Linköping Studies in Science and Technology

Dissertations. No. 1903

Programming Abstractions and Optimization Techniques for GPU-based Heterogeneous Systems

by

Lu Li

Department of Computer and Information Science
Linköping University
SE-581 83 Linköping, Sweden

Linköping 2018

ISBN 978-91-7685-370-2
ISSN 0345-7524
Printed by LiU Tryck 2018

URL: http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-145304

Abstract

CPU/GPU heterogeneous systems have shown remarkable advantages in performance and energy consumption compared to homogeneous ones such as standard multi-core systems. Such heterogeneity represents one of the most promising trends for the near-future evolution of high performance computing hardware. However, as a double-edged sword, the heterogeneity also brings significant programming complexities that prevent the easy and efficient usage of different such heterogeneous systems. In this thesis, we are interested in four such kinds of fundamental complexities that are associated with these heterogeneous systems: measurement complexity (efforts required to measure a metric, e.g., measuring enegy), CPU-GPU selection complexity, platform complexity and data management complexity. We explore new low-cost programming abstractions to hide these complexities, and propose new optimization techniques that could be performed under the hood.

For the measurement complexity, although measuring time is trivial by native library support, measuring energy consumption, especially for systems with GPUs, is complex because of the low level details involved such as choosing the right measurement methods, handling the trade-off between sampling rate and accuracy, and switching to different measurement metrics. We propose a clean interface with its implementation that not only hides the complexity of energy measurement, but also unifies different kinds of measurements. The unification bridges the gap between time measurement and energy measurement, and if no metric-specific assumptions related to time optimization techniques are made, energy optimization can be performed by blindly reusing time optimization techniques.

For the CPU-GPU selection complexity, which relates to efficient utilization of heterogeneous hardware, we propose a new adaptive-sampling based construction mechanism of predictors for such selections which can adapt to different hardware platforms automatically, and shows non-trivial advantages over random sampling.

For the platform complexity, we propose a new modular platform modeling language and its implementation to formally and systematically describe a computer system, enabling zero-overhead platform information queries for high-level software tool chains and for programmers as a basis for making software adaptive.

For the data management complexity, we propose a new mechanism to enable a unified memory view on heterogeneous systems that have separate memory spaces. This mechanism enables programmers to write significantly less code, which runs equally fast with expert-written code and outperforms the current commercially available solution: Nvidia's Unified Memory. We further propose two data movement optimization techniques, lazy allocation and transfer fusion optimization. The two techniques are based on adaptively merging messages to reduce data transfer latency. We show that these techniques can be potentially beneficial and we prove that our greedy fusion algorithm is optimal.

Finally, we show that our approaches to handle different complexities can be combined so that programmers could use them simultaneously.

This research has been partly funded by two EU FP7 projects (PEPPHER and EXCESS) and SeRC.

Populärvetenskaplig Sammanfattning

Vi lever i ett samhälle där vetenskap och teknik utvecklas i allt snabbare takt, och där datorer numera finns överallt. Människan har blivit beroende av datorer för att kunna sköta sitt dagliga arbete och även för underhållning och kommunikation. Ett modernt liv kan knappast föreställas utan datorer. I vårt mycket datoriserade samhälle kommer produktiviteten och välfärden drabbas signifikant om datorernas prestanda sviker. Snabbare datorer kan även öppna nya vägar i forskningen inom andra områden, såsom djupinlärning-teknologin som möjliggör självkörande bilar mm. De kan även underlätta upptäckter i andra vetenskapliga grenar. Till exempel, genom att köra större simuleringar kan precisare experimentella data genereras, vilket dock kräver en snabbare arbetsstation eller även en superdator. Eftersom datorer är så betydelsefulla och man fortsätter flytta mer uppgifter till dem så jobbar vi inom vetenskap och teknik hårt för att ytterligare förbättra datorers prestanda.

För att nå detta mål måste programvaran och hårdvaran samarbeta på ett bättre sätt. Tidigare kunde programvaran profitera automatiskt från snabbare hårdvara i varje generation och därmed själv bli snabbare. Men dessa gamla goda tider är över. Än värre: snabbare datorer förbrukar också betydligt mer energi, vilket skapar nya problem för samhället. Lösningen som hårdvaruindustrin tagit till sedan ca 2005 var övergången till fler- och mångkärniga datorarkitekturer, d.v.s. parallella, distribuerade och oftast heterogena datorsystem där vanliga processorer (CPU) kompletteras med grafikprocessorer (GPU) eller andra former av programmerbara hårdvaruacceleratorer. Dessa system erfordrar komplex programmering och noggrann, ressursmedveten optimering av programkoden för prestanda och energieffektivitet. Det är en stor utmaning för mjukvaruingenjörer att skapa snabb kod för dessa komplexa datorarkitekturer som kan möta det moderna samhällets stadigt ökande prestandakrav. Dessutom kan den snabba utvecklingstakten i hårdvaran leda till inkompatibilitet eller ineffektivitet av redan existerande programvara på nya hårdvarugenerationer.

Sammanfattningsvis så finns huvudsakligen fyra problem: (1) Det är svårt att skriva effektiv programkod. (2) För existerande prestandakritisk programkod är det svårt att garantera att den överhuvudtaget kan köras på varje ny hårdvarugeneration. (3) Även om själva koden är portabel så är det svårt att automatiskt bibehålla effektivitetsnivån på nästa hårdvarugeneration. (4) Vi behöver metoder som kan optimera inte bara programmets exekveringstid utan även dess energianvändning.

I denna avhandling utforskar vi programmeringsabstraktioner (t.ex. för programvarukomponenter) och tekniker för heterogena datorsystem som tar itu med dessa problem. Våra metoder och ramverk avlastar programmeraren från flera viktiga uppgifter utan att negativt påverka mjukvarans prestanda. (A) En av ansatserna automatiserar minneshanteringen och optimerar dataöverföringen så att programmet exekverar snabbare än hårdvarutillverkarens egen automatiserade lösning. Samma ansats gör det möjligt

för programmeraren att skriva kompaktare, mer läsbar kod som dock exekverar lika effektivt som expert-handskriven kod, och därmed ökar programmerarens produktivitet. (B) Vi utvecklade ett plattformsbeskrivningsspråk som underlättar att systematiskt beskriva komplexa datorsystem med deras hårdvaru- och systemprogramvarukomponenter, och som kan främja portabilitet, optimering och adaptivitet av programvara till exekveringsplattformen. (C) Vi utvecklade en ny mekanism för konstruktion av smarta prediktorer som kan göra programexekveringen adaptiv till exekveringsplattformen, möjliggör effektiv användning av hårdvaran, och visar signifikanta förbättringar jämfört med state-of-the-art lösningen. (D) Vi överbryggar gapet mellan prestandaoptimering och energioptimering på ett sätt som möjliggör att under vissa förutsättningar återanvända prestandaoptimeringstekniker för att få en reduktion av programmets energiförbrukning. Slutligen kan vi nyttja alla dessa metoder och ramverk samtidigt genom att integrera dem på ett lämpligt sätt.

Vi gör våra programvaruprototyper allmänt tillgängliga med öppen källkod. På det sättet kan de användas (och faktiskt redan har använts) t.ex. av andra forskare inom vårt område för att hantera vissa av de ovannämnda komplexiteterna och som byggstenar i andra forskningsprototyper.

Popular Science Summary

We live in a society where science and technology are evolving faster than ever, and computers are everywhere. People rely on computers to perform their daily jobs and get entertainment. Modern life is hard to imagine without computers.

In the heavily computerized society where we are living, it will significantly harm the society's productivity and welfare if computers run slowly. Moreover, faster computers can unlock the true power of research in other field, like deep learning technology that enables self-driving cars. They can also facilitate discoveries in other scientific areas, e.g., more precise experimental data can be obtained by running larger simulations which requires a faster work station or even a supercomputer. Since computers are so important and we keep putting more tasks on them, scientists and engineers are working hard to further improve their performance. To achieve such a goal, software and hardware must collaborate. In old times, software could rely on faster hardware in each generation, thereby making itself run faster automatically. But the good old days are gone, possibly forever. To make things worse, faster computers also bring significant more energy consumptions. The alternative is to introduce multicore/many core designs in our computers that lead to a scalable and sustainable energy increase but require parallel and distributed programming of often heterogeneous systems with GPUs and careful optimizations for performance and energy efficiency. Producing fast-running software on these complex parallel computers, to meet the insatiable needs of society, is very challenging for software engineers, not even considering that the fast-evolving hardware may break or run very inefficiently the software already produced. In summary, there are four main problems: 1) it is hard to produce fast software; 2) for already produced high performance software, it is hard to guarantee that they could still run on each generation of hardware that appears frequently; 3) it is hard to automatically maintain its efficiency on time on each new generation of hardware; 4) we need methods to lean more towards reducing energy consumption of software in addition to making it faster.

In this thesis, we explore new programming abstractions (for software components) and techniques to tackle these problems. We remove four important responsibilities (handling of measurement complexity, CPU/GPU selection, plaform complexity and data management) from software engineers without sacrificing software performance. VectorPU enables software engineers to write significantly less code still with the same efficiency as expert-written code, resulting in a productivity boost. VectorPU allows software to run significant faster than the current commercially available solution. We design a new platform description language XPDL to systematically describe a computer system, and protect software to be broken by different machines, and possibly by future computers. We design a new construction mechanism for smart predictors that can make software executions adaptive to different machines and allow efficient hardware utilization,

and that shows non-trivial advantages over the state-of-the-art solution. We bridge the gap between performance optimization and energy optimization, thus if no metric-specific assumptions related to time optimization techniques are made, we can easily reuse performance optimization techniques to reduce energy consumption instead. Finally, we gain all those benefits simultaneously by integrating them in meaningful ways. We make our designed software framework prototypes available as open source, thus these prototypes can (and already did) help other researchers to tackle these complexities, and utilize those prototypes for new knowledge generation.

Acknowledgements

To finish a PhD degree is challenging and full of hard work. Fortunately I enjoyed most of time during the long journey, learned many interesting things and survived a few dark moments of stress and frustrations. The thesis would not have been possible without the help from numerous persons.

First and foremost, I would like to thank my supervisor Prof. Christoph Kessler. He offered me a precious opportunity to work on this thesis topic and countless important help along this long journey. I can never express enough gratitude to Christoph. My secondary supervisor Welf Löwe also gave me valuable advices.

My wife Zhili Liu gave me indispensable support, encouraging me when I was down, and being happier than me whenever I achieved some progress. Thanks so much! My daughter Emma Ningxin Li was born during my PhD years. She is a gift to me, and like an angel, she always cheered me up ever since. I also thank my father Gechao Li and my mother Juxiang Zhang for their remote big support even though they were in China and we were separated by thousands of miles.

This work was financially supported by two EU FP7 projects: PEPPHER (www.peppher.eu) and EXCESS (www.excess-project.eu), and also by Swedish e-Science Research Center (SeRC). I also thank the EXCESS project members Philippas Tsigas, Phuong Ha, Anders Gidenstam, Dmitry Khabi, Paul Renaud-Goud, Dennis Hoppe and many other people for the constructive conversations and pleasant collaborations during the project. Some of the jointly achieved results finally became publications and formed the basis for the final thesis.

I thank National Supercomputer Center at Linköping University (NSC), Scientific Computing group at the University of Vienna and Arctic Green Computing Group at the University of Tromsø for letting me use their machines, to perform some of the experiments and some data from these experiments are used in this thesis.

I got quite some practical help and pleasant conversations with my fellow PhD students: Nicolas Melot, Usman Dastgeer, Erik Hansson and August Ernstsson. Other colleagues I want to thank are: Kristian Sandahl, Oleg Sysoev, Ludovic Henrio, Anne Moe, Åsa Kärrman, Mikaela Holmbäck, Eva Pelayo Danils and Nahid Shahmehri.

Master thesis project students I co-supervised and who helped me for some implementation work are Johan Ahlqvist and Ming-Jie Yang.

Many other people although not listed here also helped me, thank you all! The thesis is for all of you!

Contents

1 Introduction **1**
1.1 Research Questions . . . 2
1.2 Our Work and Contributions . . . 3
1.3 Research Methodology . . . 4
1.4 List of Publications and Technical Reports . . . 6
1.5 Thesis Outline . . . 12

2 Background **13**
2.1 Parallel Programming . . . 13
2.2 Programming Multicore CPUs . . . 15
2.3 Programming GPGPUs . . . 15
2.4 Software Components and *ceset*s . . . 17
2.5 PEPPHER Composition Tool: ComPU . . . 18
2.6 Skeleton Programming and SkePU . . . 19
2.7 Meta-programming in C++ . . . 22

3 Handling Measurement Complexity **25**
3.1 Measurement Complexity . . . 26
3.2 MeterPU Design and Implementation . . . 27
3.2.1 Library API and Example Applications . . . 27
3.2.2 Implementation . . . 29
3.2.3 More Examples . . . 30
3.2.4 MeterPU Support for Visualization . . . 32
3.2.5 Discussion . . . 32
3.3 Combining SkePU with MeterPU . . . 33
3.4 Experimental Results and Discussion . . . 34
3.4.1 Experimental Setup . . . 34
3.4.2 Tuning for Individual Skeletons . . . 37
3.4.3 Tuning for LU Decomposition . . . 37
3.4.4 MeterPU Overhead . . . 39
3.4.5 Comparison to Other Alternatives . . . 40
3.5 Related Work . . . 40
3.5.1 Measurement Abstraction . . . 40
3.5.2 Skeleton Programming . . . 41

3.6 Summary and Future Work 41

4 Handling CPU-GPU Selection Complexity 43
4.1 CPU-GPU Selection Complexity 44
4.2 Adaptive Sampling 45
4.2.1 An Motivating Example 46
4.2.2 Adaptive Sampling and Prediction 47
4.2.3 Experimental Results and Discussions 52
4.2.4 Summary and Future Work 56
4.3 Pruning Strategies of Adaptive Sampling 56
4.3.1 Three Pruning strategies 57
4.3.2 Experimental Results and Discussions 58
4.3.3 Summary 72
4.4 TunerPU Framework Design 72
4.4.1 TunerPU Overview 72
4.4.2 Unifying Views of a Set of Implementation Variants . 73
4.4.3 Generic Run-time Selector 74
4.4.4 Expressiveness 75
4.4.5 Summary 78
4.5 Related Work 78
4.5.1 Automated Performance Autotuning 78
4.5.2 Framework Design 80
4.6 Summary 81

5 Handling Platform Complexity 82
5.1 Platform Complexity 83
5.2 A Review of PEPPHER PDL 84
5.2.1 Control Relation 85
5.2.2 Interconnect Specification 86
5.2.3 Properties Concept 86
5.2.4 Modularity Issues 87
5.3 XPDL Design Principles 87
5.3.1 Basic Features of XPDL 88
5.3.2 Parameters, Constants and Constraints 89
5.3.3 Hardware Component Modeling 89
5.3.4 Power Modeling 96
5.3.5 Hierarchical Energy Modeling 99
5.4 XPDL Compiler and Interfacing with Other Software 99
5.4.1 Query API 100
5.4.2 Interfacing with Other Software 101
5.4.3 Portability by XPDL 102
5.5 Related Work 102
5.6 Summary 104

6 Handling Data Management Complexity 106
6.1 Data Management Complexity 107
6.2 Framework Design and Evaluation 108
6.2.1 VectorPU Design 108
6.2.2 Expressiveness of VectorPU 113
6.2.3 Experimental Results 120
6.2.4 To VectorPU or not to VectorPU? 126
6.2.5 Summary and Future Work 129
6.3 Lazy Allocation 129
6.3.1 Lazy Allocation 130
6.3.2 Transfer Fusion Optimization 134
6.3.3 Evaluation 137
6.3.4 Summary and Future Work 141
6.4 Related Work 143
6.4.1 Framework Design 143
6.4.2 Data Transfer Optimization 144
6.5 Summary 145

7 Put It All Together: a Case Study 146
7.1 A Meaningful Integration 146
7.2 Case Study: Matrix-matrix Multiplication 149
7.2.1 Baseline: Manual Code 149
7.2.2 Writing a XPDL Model 151
7.2.3 Connecting XPDL to MeterPU and VectorPU 151
7.2.4 Connecting XPDL to TunerPU 152
7.2.5 First Code Rewrite Using VectorPU 153
7.2.6 Second Code Rewrite with TunerPU 153
7.3 Evaluation 157
7.3.1 Programmability 157
7.3.2 Portability 159
7.3.3 Performance Portability 160
7.4 Summary 161

8 Conclusions 162

Chapter 1

Introduction

Computers, as a key infrastructure for our modern society, are integrated intensively with almost every aspect of human life. As we keep being increasingly dependent on computers, our expectations and imaginations that computers will do more and more for us never stops. As a consequence, there is a constant eagerness for higher performance of computer systems, also for improving the easiness to utilize these computers.

As an example to illustrate why high performance computers are important, the deep learning technology, which shows remarkable power compared to human intelligence, is mainly based on the structure and the algorithms of artificial neural networks (ANNs). ANNs have existed for a long time, but due to the lack of computing power, together with the lack of high volumes of training data, we could not see their real power for a long time, and support vector machines could easily beat neural networks for decades. Recently with the computing power from high performance computing systems, especially those equipped with GPUs allowing general-purpose computing, training of deep neural networks became possible and those trained ANNs achieve so surprisingly good results with large training data sets that their performance is often even better in its trained area than humans. In short, without modern computing power, deep learning's remarkable power can not be made visible.

It is extremely challenging for computer systems to achieve high performance due to hardware and software reasons. One of the hardware reasons is power consumption, because the frequency of processors can not keep increasing with affordable power consumption increase. The software reasons are mainly that computing efficient code automatically to optimally utilize hardware is either statically undecidable (dependent on unknown data input) or NP-complete[1]. To manually write efficient code can be time consuming even for experts and it is almost impossible to reach its optimal

[1]Taking one sub-problem of the optimal code generation, namely instruction scheduling alone is already NP-complete [91].

performance due to a large number of tunable parameters.

In order to continue the trend of performance improvements of computer systems, hardware has since 2005 evolved to the multi-core paradigm, as using multiple low-frequency processors allows to potentially achieve the same performance with a high-frequency processor, but the former consumes much less power than the latter. However, the challenges of writing efficient software are pushed to the next level of difficulty although it is already quite difficult even to utilize sequential machines efficiently. The new challenges for writing software include parallelism granularity, load balancing, communication, synchronization, and problems like data races and deadlocks are visible in parallel programmers' daily life.

In recent years, the hardware trend has moved more towards specialized hardware to complement general-purpose hardware. GPUs have shown significant performance advantages over CPUs. Although originally designed to process graphics data only, the extensions to general data-parallel computations (GPGPU) are remarkably successful. Additional software challenges brought by CPU/GPU heterogeneous systems include separate address spaces, co-existence of different programming models and optimization guidelines. CPU/GPU heterogeneous systems are the platforms that we are targeting in this thesis.

As the cost of software development and optimization already dominates the overall cost of high performance computer systems, program properties like programmability, portability, and performance portability (maintained efficiency when switching hardware) are undoubtedly important.

As a summary, the hardware evolves in a way forced by scalability, which brought huge challenges on the software side. On the other hand, however, even given such situations, we can not stop pursuing programmability, portability, and performance portability. In this thesis, we explore designs of programming abstractions and optimization techniques for CPU/GPU heterogeneous systems that will lead to improvements of those program properties.

1.1 Research Questions

Given the software challenges of CPU/GPU heterogeneous systems, we specify our research questions in this thesis as the following:

1. How to improve programmability by software abstractions for a CPU/GPU heterogeneous system? Does the increased abstraction come at a runtime cost? Is it possible to design zero-cost abstractions?

2. How to improve the portability across different heterogeneous systems with different hardware and software configurations, when applications are already written for a specific CPU/GPU heterogeneous system?

3. How to improve time performance portability for programs on different heterogeneous systems? For performance portability, we refer to the automated adaptivity of programs when migrated to a different heterogeneous system to reach decent performance.

4. How to take energy consumption into account as an optimization goal when optimizing programs for heterogeneous systems?

1.2 Our Work and Contributions

Our work is mainly on two aspects: one is framework design research, exploring how to design and implement software abstractions to achieve programmability, portability and to take energy into considerations, another explores new algorithms, optimization and performance modeling techniques mainly for performance portability, both of them are to address our research questions in Section 1.1.

Our contributions are summarized as follows:

1. A design and implementation of a measurement abstraction API/library to hide measurement complexity on CPU/GPU systems, and a quantitative evaluation and comparison of our implementation and other related work. The implementation is named as *MeterPU*. This work address the Research Questions 1 (programmability) and 4 (energy optimization).

2. A design and implementation of a data abstraction to hide data management complexity on CPU/GPU systems, and a quantitative evaluation including a comparison between our implementation and a commercial implementation, CUDA Unified Memory. The implementation is named as *VectorPU*. This work address the Research Question 1 (programmability).

3. A design and implementation of a platform description language for modeling heterogeneous systems both for hardware and software aspects, and a demonstration of its usefulness by our data abstraction. The language is named as *XPDL*, and the implementation of its compiler is named as *XDPL compiler*. A compiler implementation is needed to translate the XPDL modeling language into a library that can be directly integrated with other tool chains to address portability. This work address the Research Question 2 (portability).

4. A design and implementation of a tuning framework that transforms legacy programs into a tunable form, and allows the automated tuning of a run-time variant selector. The implementation is named as *TunerPU*. This work address the Research Question 1 (programmability).

Name	Purpose	Impl. language	License
MeterPU	Measurement abstraction framework	C++, CUDA	GPLv3
VectorPU	Data abstraction framework	C++, CUDA	LGPL
XPDL	Platform description language and its compiler	C++	Internal release
TunerPU	Tuning framework	C++	Internal release

Table 1.1: Research prototypes

5. A training space exploration technique enabling fast training, generating a decision-tree based tuner, that allows to control the trade-off between tuning quality, run-time overhead and training overhead, and three pruning strategies to further improve the training and prediction efficiency. We also provide a quantitative evaluation and comparison to other related work. The technique is tested by implementing it in the TunerPU framework. This work address the Research Question 3 (performance portability).

6. A practical run-time optimization technique on lazy allocation, which eliminates some data transfer latency between CPU and GPU on heterogeneous systems. The technique is implemented it in the VectorPU framework and evaluated. This work address the Research Question 3 (performance portability).

7. A practical run-time optimization technique on data transfer fusion and a proof for its optimality. The technique is implemented it in the VectorPU framework and evaluated. This work address the Research Question 3 (performance portability).

The basic properties of the research prototypes built are described in Table 1.1. Together with their embedded optimization techniques, they will be detailed in Chapters 3, 4, 5, and 6 respectively.

1.3 Research Methodology and Knowledge Generation

As our research questions are how-to questions, the research methodology is research through design [9], using a design as an answer, and a design itself is a body of knowledge. The power of this knowledge is descriptive and inspirational, rather than explanatory and predictive. As design research implies, design itself does not enforce methodological and scientific rigor [44].

Our design method mainly follows a case study approach: from simple programs we learn how to generalize, and then we design a generalization that applies to the simple programs. We add new cases to test the generalization and adjust if necessary. We continue until the generalization stabilizes. We keep the run-time overhead as a critical design constraint.

During and after design and implementation, we are faced with the question: is our design any better than the previous ones? This is a typical scientific question that does require methodological rigor, therefore we use quantitative and statistical methods for comparative analysis when alternative prototypes are available or the previous methods can be re-implemented in a reasonable amount of time.

For evaluation methodology, we measure programmability, portability, performance portability in the following ways:

- For measuring programmability, we use logical lines of code (LOC) as our main metric. Logic lines of code is a metric of code size but an arguable metric to measure programmability which indicates the easiness to program. Obviously small code size does not necessarily lead to easiness to program, as code with small size can have complex control flow which makes it hard to write correct programs. To the best of our knowledge, there is no consensus on how to measure programmability [109]. Other metrics such as cyclomatic complexity [105] are being used as well in the literature. Since the code we are interested in contains little control flow complexity, we argue that cyclomatic complexity will not add much extra information. Code churn [93] measures the amount of code change between two code states, indicating some quality factors of the code transition, however we are more interested in comparing two states of the code instead of its transitions. Other works that use LOC to measure programmability include [89, 32, 21, 121].

- For measuring code portability, we could compile and execute the code of interest on different machines where the key configurations differ. We obtained a "portable" or "non-portable" answer by this measurement.

- For measuring performance portability, we could measure the code of interest using time measurement functions supported on different machines where the key configurations differ. To increase the reliability of measurements, we measure multiple times and use the arithmetic mean or median value.

- For measuring energy, we use the measurement method by Burtscher et al. [18] to measure GPU energy, and Intel PCM [129] for measuring CPU energy. Those are energy-counter-based approaches, while using external meters such as Oscilloscope (energy measurement through voltage) may be better in measurement accuracy. Since our main focus is about how to model energy given measurement values, thus energy-counter-based approaches will suffice in this purpose. To increase the reliability of measurements, we use a similar method as for measuring performance portability.

1.4 List of Publications and Technical Reports

In this section we list all publications in chronological order, and summarize each author's contributions for each paper in the list below. The first person (I) denotes Lu Li, the author of this PhD thesis. Journal papers are prefixed with an asterisk (*). The publications that I played a core role in are used to form this thesis and prefixed with a †.

1. Dastgeer, U., Li, L., and Kessler, C. (2012b). The PEPPHER Composition Tool: Performance-Aware Dynamic Composition of Applications for GPU-based Systems. In *Proc. 2012 Int. Workshop on Multi-Core Computing Systems (MuCoCoS 2012), Nov. 16, 2012, Salt Lake City, Utah, USA, in conjunction with the Supercomputing Conference (SC12)*. IEEE

    - This paper described a composition tool to increase programming abstractions on CPU/GPU heterogeneous systems for PEPPHER components. PEPPHER component model was developed in EU FP7 PEPPHER project [104], which employs XML files to annotate its components' properties (e.g., function parameters) and deployment information (e.g., compiler flags). The PEPPHER composition tool translates the PEPPHER components into a low-level task-based representation that utilize a run time system StarPU [8] to handle low level details such as data transfer and scheduling.
    - The PEPPHER composition tool was implemented by me, Usman Dastgeer extended the tool and performed the experiments by the suggestions from Christoph Kessler. Usman Dastgeer wrote the paper.

2. † Li, L., Dastgeer, U., and Kessler, C. (2013). Adaptive off-line tuning for optimized composition of components for heterogeneous many-core systems. In *High Performance Computing for Computational Science-VECPAR 2012*, pages 329–345. Springer

    - This paper described a recursive decomposition method for smart sampling of training space, which is used to train a tuner for run-time software component implementation variant selection. The tuner performs run-time selection and invocation of implementation variants based on run-time context such as problem size.

- I developed the recursive decomposition method for training space exploration, Christoph Kessler developed the decision-tree-based prediction mechanism, I implemented the methods, I wrote the paper with proof-reading by Christoph Kessler.

3. Dastgeer, U., Li, L., and Kessler, C. (2013). Adaptive Implementation Selection in a Skeleton Programming Library. In *Proc. of the 2013 Biennial Conference on Advanced Parallel Processing Technology (APPT-2013)*, volume LNCS 8299, pages 170–183. Springer

   - This paper applied recursive decomposition of training space and the decision-tree-based prediction mechanism [79] to a skeleton programming framework SkePU [38] and evaluate its prediction accuracy.
   - Usman Dastgeer implemented the recursive decomposition of training space and the decision-tree-based prediction mechanism [79] designed by Christoph Kessler and me, to auto-tune SkePU, I gave some implementation suggestions. Usman Dastgeer wrote the paper, with proof-reading by Christoph Kessler and me.

4. † Li, L., Dastgeer, U., and Kessler, C. (2014). Pruning Strategies in Adaptive Off-line Tuning for Optimized Composition of Components on Heterogeneous Systems. In *Proc. Seventh International Workshop on Parallel Programming Models and Systems Software for High-End Computing (P2S2) at ICPP*. IEEE

   - This paper described three techniques to further improve the efficiency of our recursive decomposition method in Item 2 [79]: thresholding, oversampling and implementation pruning, and study their effects and trade-offs.
   - Christoph Kessler and I developed three ideas to further improve our recursive decomposition method, I performed the implementation, experiments and wrote the paper with proof-reading by Christoph Kessler.

5. * Dastgeer, U., Li, L., and Kessler, C. (2014). The PEPPHER Composition Tool: Performance-Aware Composition for GPU-based Systems. *Computing*, 96(12):1195–1211. doi: 10.1007/s00607-013-0371-8

   - This paper is an extended version of the previous paper [30] by adding more experimental results.
   - Usman Dastgeer performed the extra experiments in the evaluation, and wrote the paper with proof-reading by Christoph Kessler and me.

6. Kessler, C., Dastgeer, U., and Li, L. (2014a). Optimized Composition: Generating Efficient Code for Heterogeneous Systems from Multi-Variant Components, Skeletons and Containers. In *Proc. First Workshop on Resource awareness and adaptivity in multi-core computing (Racing 2014)*, pages 43–48

    - This paper is an invited survey paper based on the research work on SkePU by Usman Dastgeer and the PEPPHER composition tool by me.
    - Christoph Kessler wrote the paper.

7. † Li, L. and Kessler, C. (2015). MeterPU: A Generic Measurement Abstraction API Enabling Energy-tuned Skeleton Backend Selection. In *Proc. International Workshop on Reengineering for Parallelism in Heterogeneous Parallel Platforms (REPARA-2015) at ISPA-2015*, Helsinki. IEEE

    - This paper described a design and implementation of a measurement abstraction API and library for CPU/GPU heterogeneous systems, and study its programmability, expressiveness, extensibility, run-time overhead, etc. It also included an evaluation of the integration of the library with SkePU to allow energy-tunable capability.
    - Christoph Kessler suggested to make skeleton programming energy-tunable, and further suggested to experiment with the skeleton programming framework SkePU. I developed a measurement abstraction API and library designed so that making SkePU energy-tunable became a trivial effort. I designed and implemented MeterPU, further improved by Christoph Kessler's suggestions. I wrote the paper, which was then improved by Christoph Kessler's suggestions.

8. † Kessler, C., Li, L., Atalar, A., and Dobre, A. (2015a). XPDL: Extensible Platform Description Language to Support Energy Modeling and Optimization. In *2015 44th International Conference on Parallel Processing Workshops (ICPPW)*, pages 51–60. IEEE

    - This paper described a design of a platform modeling framework for CPU/GPU heterogeneous systems. It is modular, and specifically tailored to support optimizations on high level middleware on the software stack for CPU/GPU heterogeneous systems.
    - Christoph Kessler and I designed the platform description language for heterogeneous systems with modular design goal. Aras Atalar (Chalmers University) and Alin Dobre (Movidius) contributed examples of platform modeling using the language. Christoph Kessler and I wrote the paper.

9. * † Li, L., Dastgeer, U., and Kessler, C. (2016). Pruning Strategies in Adaptive Off-line Tuning for Optimized Composition of Components on Heterogeneous Systems. *Parallel Computing*, 51:37–45

    - This paper extended a previous paper [80] by adding visualization of the prediction process and analysis of net efficiency.
    - Christoph Kessler and I discussed how to extend our pruning strategy paper [80] to a journal paper by adding visualization and net efficiency analysis . I implemented those changes, wrote the extension part of the paper with proof-reading by Christoph Kessler.

10. Sjöström, O., Ko, S.-H., Dastgeer, U., Li, L., and Kessler, C. (2015). Portable Parallelization of the EDGE CFD Application for GPU-based Systems using the SkePU Skeleton Programming Library. In *ParCo-2015 conference*, pages 135–144. Published in: Gerhard R. Joubert, Hugh Leather, Mark Parsons, Frans Peters, Mark Sawyer (eds.): Advances in Parallel Computing, Volume 27: Parallel Computing: On the Road to Exascale, IOS Press, April 2016, pages 135-144. DOI 10.3233/978-1-61499-621-7-135

    - This paper ported parts of a real-world application EDGE CFG application to SkePU and evaluated the speedup.
    - Soon-Heum Ko, Usman Dastgeer and I co-supervised Oskar Sjöström in a master thesis project which formed the basis of this paper. Oskar Sjöström performed the implementation and experiments, and wrote the paper with some texts provided by Soon-Heum Ko and proof-reading by Soon-Heum Ko, Usman Dastgeer, Christoph Kessler and me. Oskar Sjöström presented the paper.

11. * † Li, L. and Kessler, C. (2016). MeterPU: A Generic Measurement Abstraction API Enabling Energy-tuned Skeleton Backend Selection. *Journal of Supercomputing*, pages 1–16. Springer

    - This paper extended a previous paper [82] by adding the following: quantitative comparison on code complexity using logic LOC (lines of code) between MeterPU and other alternatives, inline investigation for MeterPU overhead, demonstration of MeterPU's visualization feature, extension of MeterPU to measure time on multiple GPUs and extension of MeterPU to measure power by external measurement devices.
    - Christoph Kessler and I planned how to extend our previous MeterPU paper [82] to a journal paper. I performed the implementation and experiments, and wrote the extension part with proof-reading by Christoph Kessler.

12. * Ernstsson, A., Li, L., and Kessler, C. (2016). SkePU 2: Flexible and Type-safe Skeleton Programming for Heterogeneous Parallel Systems. *International Journal of Parallel Programming*

    - This paper improves significantly the programming interface of a skeleton programming framework SkePU, and evaluated its performance compared to the previous implementation of SkePU.
    - Christoph Kessler and I co-supervised August Ernstsson, as a master thesis project which forms the basis of this paper. August Ernstsson performs the design and implementation, the experiments and wrote the paper with proof-reading by Christoph Kessler and me. This master thesis was given the best thesis award by IDA, Linköping University in 2016.

13. Thorarensen, S., Cuello, R., Kessler, C., Li, L., and Barry, B. (2016). Efficient Execution of SkePU Skeleton Programs on the Low-power Multicore Processor Myriad2. In *Proc. Euromicro PDP-2016 Int. Conf. on Parallel, Distributed, and Network-based Processing.* IEEE

    - This paper extended SkePU to a low power processor Myriad2 and evaluated its performance.
    - Christoph Kessler and I supervised a master thesis project by Rosandra Cuello, which formed the basis for the subsequent master thesis done by Sebastian Thorarensen, who performed the design and implementation with the help of Brendan Barry. Sebastian Thorarensen wrote the paper with proof-reading by Christoph Kessler, Brendan Barry and me.

14. † Li, L. and Kessler, C. (2017b). VectorPU: A Generic and Efficient Data-container and Component Model for Transparent Data Transfer on GPU-based Heterogeneous Systems. In *Proc. 8th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and 6th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM'17).* ACM

    - This paper describes an embedded annotation language for normal CUDA programs, that allows for automatic data management without redundant data transfer. It also included a performance comparison with a commercial mechanism: CUDA Unified Memory.
    - I developed an annotation language that allows to generalize the idea of smart containers [27] from SkePU to normal CUDA programs, I designed and implemented the language with suggestions from Christoph Kessler. I performed the experiments and wrote the paper, with proof-reading by Christoph Kessler.

15. † Li, L. and Kessler, C. (2017a). Lazy allocation and transfer fusion optimization for GPU-based heterogeneous systems. In *Proc. Euromicro PDP-2018 Int. Conf. on Parallel, Distributed, and Network-based Processing*. IEEE
    - This paper described two memory transfer optimization techniques based on merging small messages. It discussed the design and implementation of the two techniques, and gave some initial evaluation of performance benefits.
    - I designed and implemented the two techniques, performed the evaluation and wrote the paper with Christoph Kessler.

I also contributed to the following technical reports:

1. Dastgeer, U., Li, L., and Kessler, C. (2012a). D1.4: Research prototype implementation. Technical report, ©The PEPPHER Consortium. June, 2011

2. † Kessler, C., Li, L., Dastgeer, U., Tsigas, P., Gidenstam, A., Renaud-Goud, P., Walulya, I., Atalar, A., Moloney, D., Hoai, P. H., and Tran, V. (2014c). D1.1 Early validation of system-wide energy compositionality and affecting factors on the EXCESS platforms. Technical Report FP7-611183 D1.1, EU FP7 Project EXCESS

3. † Kessler, C., Li, L., Dastgeer, U., Gidenstam, A., and Atalar, A. (2014b). D1.2 Initial specification of energy, platform and component modelling framework. Technical Report FP7-611183 D1.2, EU FP7 Project EXCESS

4. † Kessler, C., Li, L., Dastgeer, U., Cuello, R., Sjöström, O., Hoai, P. H., and Tran, V. (2015b). D1.3 Energy-tuneable domain-specific language/library for linear system solving. Technical Report FP7-611183 D1.3, EU FP7 Project EXCESS

5. † Kessler, C., Li, L., Hansson, E., Ahlqvist, J., Thorarensen, S., and Yang, M.-J. (2015c). D1.4 First prototype of composition tool and multi-level energy and platform modeling framework. Technical Report FP7-611183 D1.4, EU FP7 Project EXCESS

6. Dolkas, K., Sandoval, Y., Hoppe, D., Khabi, D., Umar, I., Ha, P., Moloney, D., Li, L., Kessler, C., Gidenstam, A., and Renaud-Goud, P. (2015). D5.3 Report on integrating the first designs and prototypes from technical WPs. Technical Report FP7-611183 D5.3, EU FP7 Project EXCESS

7. † Kessler, C., Li, L., Melot, N., Hansson, E., Ernstsson, A., Thorarensen, S., and Barry, B. (2016). Final specification of energy, platform and component modelling framework and final prototype. Technical Report FP7-611183 D1.5, EU FP7 Project EXCESS

1.5 Thesis Outline

The thesis starts by providing necessary background in Chapter 2, then we discuss several programming challenges: the complexity to handle measurement, CPU-GPU selection, platform diversity and data management. At the same time, we present our prototype design and optimization techniques: MeterPU, TunerPU, XPDL, and VectorPU in Chapter 3, 4, 5, and 6 respectively. These prototypes addresses programming complexities at different levels and can be combined together. In order to demonstrate how to integrate and use these prototypes together, we provide an example in Chapter 7. Finally Chapter 8 concludes and suggests possible future work.

Chapter 2

Background

In this chapter we provide some basic background knowledge that helps to understand the rest of the thesis, and the definitions of the terminology that we use later on.

The thesis is partially based on the concept of *software component*. Before introducing this concept, we first describe various ingredients that a component could consist of in the context of parallel programming. First we introduce parallel programming in general without binding to a specific piece of parallel hardware in Section 2.1, and then we introduce programming for several popular kinds of parallel hardware, such as multicore CPUs using OpenMP and GPGPUs using CUDA in Sections 2.2 and 2.3. With all the ingredients at hand, we introduce the concept of a (parallel) software component in Section 2.4, and the composition tool ComPU that utilizes existing software components for some optimizations in Section 2.5. Afterwards, we introduce skeleton programming as another way of parallel programming in a higher abstraction level compared to OpenMP, CUDA etc., and a skeleton programming framework SkePU which we will use for some of the work presented in this thesis. Finally we describe C++ meta-programming which allows compile-time computations with no run-time overhead, and makes it suitable for implementing low-overhead programming abstractions, such as those developed in Chapters 3, 5 and 6.

2.1 Parallel Programming

Programming a sequential machine efficiently is already a very complex problem. As CPU hardware evolved to multicore architectures, programmers not only need to program in parallel, but also play a critical role for performance. As Hennessy and Patterson [51] pointed out:

> "The popular version of Moore's law—increasing performance with each generation of technology—is now up to programmers."

In addition to sequential programming, parallel programming adds quite some responsibilities to programmers' shoulders. Foster's PCAM method [63] gives a conceptual method for the design of a parallel program without binding to a specific piece of parallel hardware, which illustrates a clear picture for these new responsibilities. As the name "PCAM" implies, the method includes four stages as follows:

1. **P**artitioning: this stage refers to determining how to partition the workload into multiple pieces of work, we could call each such piece a *task*. These tasks could be executed in parallel if there are no dependencies among them. Two partitioning schemes are *domain decomposition* and *functional decomposition*. Domain decomposition leads to *data level parallelism*, and functional decomposition leads to *task level parallelism*. Data level parallelism usually incurs lower overhead than task parallelism [51].

2. **C**ommunication and synchronization: some tasks may not have dependencies, but more frequently they have. Some tasks may need to communicate with other tasks by providing data that these tasks need to perform their intended computations. Such communications can be performed by message passing or through shared data with appropriate protections to avoid race conditions. To ensure that tasks only execute after they obtained the data they need, synchronizations are needed to force these tasks to wait if they are not data-ready yet.

3. **A**gglomeration: the whole workload can be partitioned in different granularities. A more coarse-grained partitioning will constrain the level of parallelism, while an overly fine-grained partitioning will lead to excessive communications and synchronizations. Agglomeration can increase the granularity of workload partitioning, and remove the need of some data communications and synchronizations. Determining the optimal granularity is usually hardware- and application-dependent, and may require empirical tuning.

4. **M**apping and scheduling: after agglomerations, we have a reduced set of tasks that can finally be mapped to and scheduled on processing elements. Mapping refers to determining where to execute a certain task, and scheduling refers to determining when and where to execute a certain task. To determine the optimal mapping and scheduling depends on applications, run-time contexts, and hardware, which usually is statically undecidable or NP-complete.

Without any programming abstractions, parallel programmers need to handle these programming issues explicitly, and handle them properly. Some of these aspects can however be abstracted away by more high-level parallel programming models and their implementations.

In next two sections, we will discuss the hardware-specific parallel programming on two popular processors.

2.2 Programming Multicore CPUs

A modern CPU consists of multiple cores that allow truly parallel executions of different programs. Different cores in a CPU could share cache at least at its last level, and main memory. A computer system could consist of multiple multicore-processors in different sockets, with each connected to its own memory. A processor could be able to access another processor's memory at a larger latency than its own, which is called *non-uniform memory access (NUMA)*. Many processors can execute vector (SIMD) operations, which implements data level parallelism. Each core in a processor could run different programs in parallel, which implements task level parallelism. The processor itself is pipelined, and compilers can transform a loop to better exploit pipelining, which is called *software pipelining*.

At programming model level, what are visible to programmers are vector (SIMD) operator intrinsics, thread API (e.g., pthreads [19]) to implement data level parallelism and task level parallelism. Pipelining is usually out of programmers' hand, but rather conspired by compilers and hardware. Different threads could exchange messages by shared memory protected by synchronization techniques, such as mutex locks, atomics etc. Higher level programming models on multicore CPUs also exist, like OpenMP [25], which only requires to annotate loops for parallelizing a sequential program.

2.3 Programming GPGPUs

A *Graphics Processing Unit (GPU)* is a kind of co-processor that can offload graphics processing tasks from CPUs in a computer system. After 2001, performing general purpose computations on some GPUs became practical by introducing programmable shaders and floating point support on GPUs, and more and more applications were ported to such GPUs ever since. Such general purpose GPUs are usually called *GPGPU*s. In this thesis, we use the terms GPU and GPGPU interchangeably. CUDA [98] and OpenCL [92] is introduced afterwards, to better support general purpose programming on Nvidia's CUDA-enabled GPUs.

GPGPUs are usually designed for high throughput, compared with CPUs which aim at short latency. Under such design philosophy, GPGPUs are usually equipped with a large number of simple cores to ensure a high level of parallelism, and those simple cores usually have limited cache size and shallow cache hierarchy without sophisticated branch predictors, out of order execution, speculation etc. These simple cores are organized into groups, each group is called a *streaming processor*[1] (*SM*). The programmers write code for a large number of threads, which are also organized into groups; each group is called a *block*. A block is usually mapped to a SM for execution at run-time. A set of blocks that run the same code is called a *grid*.

[1]We use Nvidia's terminology, for OpenCL's terminology, please see [92].

Typical GPGPUs have three levels of memories with different access latencies that are visible to programmers: global memory that could be accessed by every thread with a large latency, shared memory that could only be accessed by threads within a block with a medium latency, and private memory that could only be accessed by each individual thread with a short latency. Within a block, threads could communicate via shared memory. The inter-block communication is much harder, usually through global memory and could only be done by multiple kernels.

In order to program for GPGPUs, e.g., using CUDA, programmers write code in the *SPMD* (single program, multiple data) style, where the kernel is the same for every thread, and programmers invoke GPGPU kernels by specifying grid and block size. Programmers are recommended to target at a relatively large block and grid size, to ensure a high degree of parallelism. In the kernel, a thread id for each thread is calculated, and branching is necessary if different threads should perform different operations. Such branching may not be good for performance, as the code is branching to one arm, the threads that should branch to other arms have to wait and this causes a sequentialization, which is called *control divergence*. Thus GPGPUs fit data parallel applications better performance-wise.

Programmers are also recommended to utilize scratchpad memory such as shared memory within a block if multiple accesses to the same data from different threads within a block make it worthwhile.

In order to execute GPGPU code, programming on CPU side is necessary as well. A CPU plays the role as a master for a GPGPU in the same system. CPUs' main memory is also called host memory, and GPGPUs' global memory is also called device memory. A CPU performs management tasks such as memory allocations, memory initializations, data transfers between host memory and device memory, and GPGPU kernel invocations etc. Data should also be transferred from host memory to device memory before the invocation of the kernel that will read these data. The result data should be transferred back from device memory to host memory if host functions need them.

Open Computing Language (OpenCL) [92] is an open standard for cross-platform parallel computing that can provide a similar programming interface with similar functionalities as CUDA for programming GPGPUs. OpenCL is portable across many different processing element types such as multicore CPUs, GPGPUs etc., while CUDA only support Nvidia's CUDA-enabled GPUs. Programming using OpenCL involves more efforts than CUDA in general, as OpenCL's API is more complicated. Higher level programming models on GPGPUs also exist, like OpenACC [128], which only requires to annotate loops for parallel execution on GPGPUs.

2.4 Software Components and *ceset*s

The software component is an important concept in software engineering, and there are many definitions for software component. The most popular one is given by Szyperski et al. [120], which we will use in this thesis:

> "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties." [120]

From this definition, we can extract a few pieces of key information of our interest:

1. A component is only visible for its interface and dependency information.
2. The internal structure of a component is undefined, thus programmers could use different techniques at hand to implement a component.
    - It could be implemented sequentially or in parallel.
    - It could be implemented targeting multicore CPUs or GPGPUs.
    - It could be implemented using low level programming models, such as pthreads, CUDA, OpenCL etc., or high level programming models, such as OpenMP, OpenACC etc.
    - It could be implemented using different algorithms.
3. A component can be composed by third parties in a differently configured environment.

Although a component can consist of many different elements, such as (multiple) functions, data, images, meta-data etc, in this thesis we constrain our components to be light-weight in component structure, each only consisting of a single function, although the function can be arbitrarily complex. We use the terms component and light-weight component interchangeably. Using lightweight components is only for the convenience of reference. We only consider C/C++-based programming models as they are common on GPU-based systems, thus no heavy-weight interoperability machinery such as stubs[2] and skeletons, like in CORBA [111], is needed.

An example of lightweight component is shown in Listing 2.1. The component consists of only one function: `vector_scale()`. It exposes its interface, its implementation is not necessarily visible even if we show it here, the implementation may be in a binary library.

[2]Stubs and skeletons in this context are adapters that convert a component call to a form that can bridge programming language difference gap.

```
void vector_scale(int const * int in, int * out,
                  int const size, int scale_factor){
  for (int i=0; i<size; ++i)
    out[i]=in[i]*scale_factor
}
```

Listing 2.1: A lightweight component

Due to the undefined internal structure of a component, we could implement an interface using different design choices, which will lead to a set of components that expose the same interface, with different implementation design choices and possibly different dependency requirements. Each of them is called an *implementation variant*, and they all share the same interface, which is called the *interface* of the *ceset* Note that an implementation variant is also a component, and in this thesis we use them interchangeably. The set of all implementation variants implementing the same functionality under an interface is called a *component equivalence set*, abbreviated as *ceset* (pronounced as /*si* : *set*/).

Each element in a *ceset* is implemented by different design choices, thus its performance (and energy consumption) may differ significantly with each other in a *ceset*. For example, two components implemented for the same interface of a reduce operation, but programmed to run on multicore CPU and GPGPUs can differ in performance by 647 times [83]. Furthermore, each implementation variant may not outperform another across all its input range, thus this creates a space for performance optimization by implementation selection according to the execution of a run-time context such as its input instance, which we call the *implementation selection problem.*

A component is subject to composition by third parties, thus a third party tool could bind a call to any component in its *ceset*. Binding a call to a component means to map a call to one of the component in the *ceset*, so whenever such a call is encountered in program, we know which component in the *ceset* the call should jump to. If such binding is fixed at compile time, we call it *static composition.* If such binding is fixed at run-time, we call it *dynamic composition.*

Next, we will discuss a composition tool that first utilize static composition, and then delegate the final binding choice to a run-time system for dynamic composition.

2.5 PEPPHER Composition Tool: ComPU

Given a program consisting of calls to interfaces, each with a non-empty *ceset*, we could build tools to manipulate the static and dynamic binding from a caller to a callee (i.e., an implementation variant) from a *ceset*. Tools could prepare the code to make such bindings adaptive to its run-time context as a run-time optimization.

The composition tool ComPU [32] was developed in EU FP7 project PEPPHER [104]. It is designed to perform static composition and prepare components for further dynamic composition. ComPU requires programmers to annotate their components. Compiler analysis can not determine automatically which components belong to the same *ceset*, because components that expose the same interface do not necessarily implement the same functionality. Other annotations that programmers need to provide include the access modes (read, write or read-write) of its parameters of a component, which allows automatic memory management for operands, and the dependency constraints (e.g., a component may require a specific library) information.

ComPU parses those XML files carrying component annotations. The static composition that ComPU performs mainly involves narrowing down the *ceset* by filtering those components whose dependency constrains are evaluated as not satisfied by ComPU. This will ensure the further composition to be always executable, leading to an executable program. ComPU performs the dependency constraint evaluation by fetching platform information through the XPDL query API (see Chapter 5). In short, ComPU disables the components that can not run on a system for the further linking.

Since ComPU runs statically and the final binding requires the run-time context of an invocation, thus a run-time system is needed to perform the final binding. In the PEPPHER project, StarPU [8] is selected for ComPU to pass the narrowed *ceset* to. StarPU requires a component to be wrapped in its specified format, thus after narrowing, ComPU generate the wrapper code for the remaining components, and additionally generate a makefile for compilation. Up to this point, programmers could compile the components with their application and run it with StarPU providing for final binding based on its performance models. Using ComPU allows to generate some low level code required by StarPU thus enable programmers to write less code [32].

To summarize, ComPU reads components' XML descriptors, narrows down each *ceset* by evaluating the constraints on component dependencies, generates wrappers for the remaining component in the narrowed *ceset*, and delegates the final binding to StarPU.

2.6 Skeleton Programming and SkePU

The idea of skeleton programming comes from functional programming [54], which encourages to write programs with immutable variables and side-effect free functions, and treats a program execution as evaluating a mathematical expression. Side-effect free functions can safely run in parallel, as they do not mutate state outside their own function scope, and no data access conflict could possibly happen. Haskell is an example of popular functional languages.

Functional programming treats everything as a value, including functions. Therefore a function can take another function as an argument, and those functions that could take functions as arguments are called *high-order* *functions*. By utilizing high-order functions, programmers could write an outer function, containing some control structure and pass a function value to the outer function with the control structure to control the execution of the function. For example, programmers could write a loop, then pass different function values to it to loop on different functions, thus the loop structure is reused and abstracted away. This kind of abstraction is called *control abstraction* [100].

Skeleton programming utilizes the concepts of control abstraction and side-effect free functions. Each skeleton is a high order function that has well-defined semantics and implements a control abstraction, e.g., a high-order function only consisting of a simple loop on an array without loop-carried data dependencies is called a *map* skeleton. Programmers only need to write one or several side-effect free functions[3] to express the core computations to be performed element-wise, which are called *user functions*, and combine these functions with suitable skeletons to get the correct control for free. Since user functions are side-effect free, each skeleton has greater flexibilities to execute them, compared to functions that produce side-effects. The flexibilities include the freedom to execute them in parallel without worrying about race conditions etc., which makes skeleton programming a good fit to the parallel programming context. As each skeleton has well-defined semantics, expert programmers could implement the synchronizations required and non-expert programmers reuse it for different user functions.

SkePU [38] is a research software framework that implements the idea of skeleton programming on heterogeneous systems that may consist of sequential CPUs, multicore CPUs and GPGPUs. SkePU offers programmability, as programmers only need to write one or several sequential user-functions, and the implementation variants that are needed for execution on multicore CPUs and GPGPUs etc. are generated automatically by SkePU. SkePU supports automatic and adaptive CPU-GPU selection [31] by implementing the implementation selection technique of Section 4.2. SkePU offers data structures with automatic memory allocation and coherence management between CPU memory and GPU memory. In recent years, SkePU has been ported to the low-power multicore processor Movidius Myriad2 [124]. The second generation of SkePU, *SkePU 2* [40] redesigned SkePU's programming interface with modern C++11 language features.

SkePU has a rich support for different types of skeletons to provide the wide expressiveness for general high performance computations. The typical data-parallel skeletons are shown in Table 2.1. The unary map skeleton takes every element in an array and produces a result by applying a user function on each element in the array. The binary map skeleton does the same as a unary map skeleton except that the input consists of two arrays, and the user function takes two values, one of each of the two arrays. The *reduce* skeleton takes an input array and produces a scalar by scanning

[3]These side-effect free functions return their output values instead of writing to an pointer.

Skeleton type	Description
Map(a, b, f)	$b_i = f(a_i), i = 1, ..., n$
Map(a, b, c, f)	$c_i = f(a_i, b_i), i = 1, ..., n$
Reduce(a, f)	$d = f(a_1, a_2, ..., a_n), i = 1, ..., n$
Mapreduce(a, b, f, g)	$d = g(f(a_1, b_1), ..., f(a_n, b_n)), i = 1, ..., n$
Mapoverlap(a, t, f)	$b_i = f(a_{i-t}, ..., a_{i+t}), i = t, ..., n - t$
Maparray(a, b, c, f)	$c_i = f(b_i, a_1, ..., a_n), i = 1, ..., n$

Table 2.1: Typical SkePU skeletons and their semantics. (a, b, c: vector. d: scalar. t: positive integer constant. f, g: user function)

the input array by applying a user function. The *mapreduce* skeleton is the combination of a map and reduce skeleton, such combination allows to optimize away an intermediate array to store the result of the map skeleton execution. The *mapoverlap* skeleton implements the *stencil* computation, where the user function can access values from a specified neighborhood of each input element in an array to compute each output value. The *maparray* skeleton is similar to the mapoverlap skeleton, except that a user function can use all elements of an input array to compute each output value.

A typical SkePU program is shown in Listing 2.2. Lines 3-6 declare a user-function `mult` that performs a simple binary multiplication. Line 11 instantiates a map skeleton composed with the user function, and Line 13 invokes the skeleton instance. It could run on multicore CPUs or GPGPUs based on automatic implementation selection which is adaptive to the run-time context (e.g., the input vector's size). All low level details, such as data allocation, data transfer, hardware-specific implementation variant generation, are hidden in the skeleton implementations and reused for different user functions for different computations.

```
1  #include <skepu2.hpp>
2
3  int mult(int x, int y)
4  {
5    return x * y;
6  }
7
8  int main(int argc, char *argv[])
9  {
10   skepu2::Vector<int> vector1(100, 3), vector2(100, 7), result(100);
11   auto vecmult = skepu2::Map<2>(mult);
12
13   vecmult(result, vector1, vector2);
14   std::cout << "Map: result = " << result << "\n";
15
16   return 0;
17 }
```

Listing 2.2: Example code snippet for map skeleton using SkePU 2

2.7 Meta-programming in C++

Meta-programming is very different from traditional programming in that meta-programs run at compile time instead of run-time, thus meta-programs cause no rum-time overhead. What is a meta-program? Abrahams et al. [1] provide a definition of meta-program as follows:

> "A *meta-program* is a program that manipulates code."

Thus in principle a compiler could be viewed as a meta-program. A meta-program allows to define a new language and uses programs in that language as an input. The input language is called *domain language*. A meta-program manipulates a domain language and translate it to a *host language*. For example, the parser generation tool YACC [59] takes parser specifications as its domain language, and translates to its host language, C.

YACC illustrate a meta-program example where its domain language and host language differ. One could also design a meta-program where its domain language and host language are the same. In such a situation, the domain language is defined using the host language's syntax and semantics. We usually call such a domain language an *embedded language*. The benefits of an embedded language are three-fold: first, it does not require to learn a new set of syntax; second, the interactions between the domain language constructs and the host language constructs are smooth, since in essence the domain language constructs and the host language constructs are in the same program; third, there is no extra build step needed for the embedded language.

In this thesis, we are targeting at GPU-based heterogeneous systems, and the languages that are used to program GPGPUs are based on C/C++. Since C++ has better modularity support than C (e.g., C++ allows to modularize both functions and data compared to C), we choose C++ for meta-programming and design embedded languages whenever it applies.

Several kinds of meta-program constructs exist, such as numerical meta-functions, type meta-functions etc. Listing 2.3 gives an example of a numerical meta-function. Lines 1-3 show a definition of a numerical meta-function. The `const` keyword guarantees that the value can be used and computed at compile time, and the `static` keyword allows the usage of the value without run-time object creation. Line 6 shows how to invoke the function. Numerical meta-function is equivalent to constant values defined by macros, except that macros are expanded at preprocessing time while Numerical meta-function is expanded at compile-time, and usually numerical meta-function allows explicit typing for its constant value.

```
struct three {
  static int const value = 3;
}

//invoke by
three::value
```

Listing 2.3: Numerical meta-function example

Listing 2.4 shows an example of a type meta-function. The input is the type `T` in Line 1, and the output is the type `type` in line 3. Such a computation transforms one type to another. Since template expansion is performed at compile time, such computation is finished after compilation.

```
1 template <class T>
2 struct Func {
3   typedef type ...;
4 }
5
6 //invoke by
7 Func<Int>::type
```

Listing 2.4: An general meta-function example

A C++ *trait* is a special case of C++ type meta-function. A metaphor of a trait is that it is a type meta-function with multiple return values. Listing 2.5 shows an example of a C++ trait. It allows to group a set of types (such as `type1`, `type2`, etc. in Lines 2-4) and associate the group with a single type (type `Trait` in Line 1). Since behaviors are usually encapsulated in classes which are also types, this grouping could simplify controlling a set of behaviors by using a single type as a parameter. C++ traits are used to implement the MeterPU `Meter` class in Chapter 3.2.1.

```
1 struct Trait {
2   typedef type1 ...;
3   typedef type2 ...;
4   ...
5 }
```

Listing 2.5: A trait as a special case of a meta-function

C++ macros can be considered as a kind of C++ meta-program constructs, because they can manipulate code as well. Macros could be used to enable or disable arbitrary segments of code, and take preprocessing-time values. One should use macros with care, as it could decrease readability and make debugging difficult.

For the concern of programming abstraction implementation, C++ meta-programming for embedded languages has very interesting properties: first it allows to define a new language easily compared to implementing a compiler; secondly it causes no run-time overhead, which is good as programming abstraction implementation requires highly efficient execution; thirdly, the interaction between the new language and the host code is native and seamless, thus it allows to build language extensions such as annotations easily. On the other hand, C++ meta-programs are less powerful than a full-fledged compiler which can perform complex analysis and optimizations based on different intermediate representations (e.g., SSA [115]), and C++ meta-programs can only compute values with its input determined at compile time.

To summarize, C++ meta-programming allows to define new embedded languages with zero run-time cost and reasonable programmers' efforts, and in this thesis, we will utilize C++ meta-programming to build programming abstractions when it applies, e.g., MeterPU in Chapter 3, XPDL library in Chapter 5 and VectorPU in Chapter 6.

Chapter 3

Handling Measurement Complexity

This chapter is based on the following journal paper:

- Li, L. and Kessler, C. (2016). MeterPU: A Generic Measurement Abstraction API Enabling Energy-tuned Skeleton Backend Selection. *Journal of Supercomputing*, pages 1–16. Springer

  Which in turn, is an extended version of the workshop paper:

- Li, L. and Kessler, C. (2015). MeterPU: A Generic Measurement Abstraction API Enabling Energy-tuned Skeleton Backend Selection. In *Proc. International Workshop on Reengineering for Parallelism in Heterogeneous Parallel Platforms (REPARA-2015) at ISPA-2015*, Helsinki. IEEE

A fundamental question in empirical science is how to measure things, e.g., Newton's second law can be re-discovered by measurement and simple curve-fitting. This question is particularly important for computer science, and it is necessary to measure some metrics (e.g., time) of programs, as in general no perfect model can predict e.g., time of a program execution by just static analysis of the program. The power of static program analysis is limited, caused by statically unknown input that leads to a statically undecidable path of control flow, aliasing and imprecision of data flow analysis. To build a reasonable model to replace measurement requires measuring as a prerequisite, and one may need to calibrate or verify such models from time to time via measurements, especially when switching platforms. Measurement is the basis for optimization, as one can not optimize what can not be measured.

Is measurement simple or complex? How to measure energy as it becomes the bottleneck of HPC systems? How to deal with different kinds of measurements? Why is unifying different kinds of measurement desired and how can it be done to hide the programming complexity of measurement? This chapter will explain these questions, and serves as a basis or infrastructure to the following chapters.

Section 3.1 gives an overview of the complexity involved to measure programs. Section 3.2 describe the design and implementation of MeterPU to tackle such complexities. Section 3.3 shows how to integrate MeterPU with SkePU (cf. Section 2.6) as we evaluate MeterPU through SkePU. The evaluation is shown in Section 3.4.

3.1 Measurement Complexity

Measurement of time that programs run is simple as the existing APIs on different kinds of operating systems already hide the complexity, allowing programmers to mark a measurement region and to get back a human-readable result such as "3 microseconds". However, measurement of energy is still a challenge. Instead of only fetching some hardware counters immediately before and after the measurement region in the code, measuring energy requires constantly sampling power data (besides maintaining a separate thread for constant sampling, the overly high sampling frequency may incur significant overhead which is included in a measurement), either by built-in power sensors or by estimates. The frequency of such sampling is usually a hard decision, as higher frequency allows better precision, while it incurs its own energy overhead that affects the measurement accuracy. Using external measurement equipments such as oscilloscope requires careful deployment and some physical instrumentations on the device under measurement at the risk of circuit damage, with the later-on complexity of power data analysis.

Assuming that all these complexities are properly handled, the power data obtained may still not be what we expect. Burtscher et al. [18] studied measurement of GPU energy using built-in sensors, and found out that energy is still spent after the GPU kernel under measurement finishes. He assumes there is a capacitor of some sort, and correct the power data by a formula mimicking capacitor charging and recharging.

Some special-purpose libraries appeared to hide the complexity of measurement for some hardware components, such as the Intel Performance Counter Monitor (PCM) library [129], which allows to easily measure CPU and DRAM energy. However, no library, as far as we know, can directly yield energy consumption values for GPUs. NVIDIA Management Library (NVML) [94] allows to sample the power consumption of the whole GPU board at a low frequency (20 samples per second by microbenchmarking), however, by the study of Burtscher et al. [18], those values are not the true power usage and require correction. Different libraries expose different interfaces, to use a different library requires learning the library carefully.

As a summary, measurement of energy for program executions, especially on GPUs, are not trivial, and due to the importance of measurement, this issue must be handled and the complexity must be hidden. In addition, non-uniform interfaces exposed by different library make it hard to use them.

In the following section (Section 3.2), we propose a design and implementation as a generalization effort to unify measurements for different metrics on different hardware components on GPU-based heterogeneous systems. The benefits of such a unification are at least the following:

- The complexity of measurements for different metrics on different hardware components is hidden.
- The gaps of different interfaces of different libraries are filled, which gives programmers a unified view for different measurements. This minimizes programming efforts to perform the measurements for various purposes, such as microbenchmarking, building performance models etc.
- The switching of different metrics becomes trivial, which yields benefits like easy switching optimization goals. For example, the optimization goal could be switched from time to energy for legacy optimization frameworks that requires measurements, if the switching does not break goal-specific assumptions.

3.2 MeterPU Design and Implementation

MeterPU is a C++ template-based library prototype, which aims to provide an abstraction layer for measurement of various metrics on different platforms. Currently it supports measuring time and energy spent on CPU, on DRAM and on (single and multiple) GPU(s). It can also measure energy by different measurement techniques, such as external measurement devices (e.g. Wattsup power meter). The library can be extended easily for new metrics, such as FLOPS, cache misses etc., and for new measurement techniques.

3.2.1 Library API and Example Applications

MeterPU exposes to programmers a unified API no matter which type of supported metric is used.

Listing 3.1 shows the API that MeterPU exposes to programmers. It consists of a class called `Meter` with a template parameter to specify the Meter's type. The `Type` parameter specifies which metric is used, such as time or energy, and which hardware component to measure, such as CPU or GPU. The generic type also has a C++ trait [15] defined with it, where it can fetch all metric-related types. A C++ trait class is a class that encapsulates all related types (meta-data) together. For instance, if CPU time is used as metric (template parameter `Type` is `CPU_Time`), then `typename Meter_Traits<Type>::ResultType` will give the time unit as the

`CPU_Time`-related types, such as microsecond. The API consists of four main member functions of `Meter`: `start()`, `stop()`, `calc()`, and `get_value()`, which are used as follows:

- start(): mark the start of a measurement phase/period.
- stop(): mark the end of a measurement phase/period.
- calc(): calculate the metric value based on the measurements taken between start() and stop(). It is possible to merge calc() with stop(), but here we expose the two functions to programmers[1], and class `Meter` can be inherited if the merge is preferred.
- get_value(): return the value of the measurement between the start and stop calls.

```
template<class Type>
class Meter
{
public:
 void start();
 void stop();
 void calc();
 typename Meter_Traits<Type>::ResultType const &get_value()
    const;
private:
   ...
}
```

Listing 3.1: Main MeterPU API

Listing 3.2 shows an example application that uses MeterPU. First a meter is initialized with the type `CPU_Time`, showing that it is a time meter on CPU. Then `start()` and `stop()` are used to mark the start and end of a time measurement by the meter, and finally `calc()` and `get_value()` are called to calculate the difference in metric value between the two program points, and to print the meter reading on a terminal.

[1]Actually this separation can lead to reduced overhead if more than one meter is measured over the same time interval, as the possibly time-consuming calc() calls can be done outside the measurement interval for each metric.

```
#include <MeterPU.h>
int main()
{
    using namespace MeterPU;
    Meter<CPU_Time> meter;
    meter.start();
    //Do sth here
    sleep(2);
    meter.stop();
    meter.calc();
    my_print( meter.get_value() );
}
//Result:
//[CPU Time Meter] Time consumed is 2.00013e+06 micro seconds.
```

Listing 3.2: An example application that uses the MeterPU library

3.2.2 Implementation

The actual measurement plug-in code used with MeterPU is implemented on top of native measurement libraries. For the CPU time measurement, it may use the native `clock_gettime()`[2] function. For the CPU energy measurement, it may use the Intel PCM library. For the GPU energy measurement, it may use the Nvidia NVML library (nvmlDeviceGetPowerUsage(), reported sampling error within 5% [94]) to get power samples, and apply numerical integration to calculate energy values and compensate for the Nvidia NVML's capacitor-like effect[3] with K20c. The Nvidia power integration is only executed after the completion of the measurement by MeterPU of a code region. Thus, such power integration overhead will not influence the measurement result obtained from MeterPU.

MeterPU is not coupled with a specific measurement method, and it can be hooked with more accurate methods or devices. So far we are not aware of any library that offers power samples on the PCI Express bus on the motherboard side, thus the motherboard-side energy of GPU data transfers is not supported yet in MeterPU, and not included in the experimental results described in Section 3.4. In the future one can consider to use an external power meter between the correct power rails of a motherboard for the power samples of PCIe, in this case a spare/separate computer can act as a host for the power meters, and one can develop a library for polling samples of PCIe power from that computer, then it will integrate with MeterPU easily.

MeterPU uses C++ traits to encapsulate all type information related to a meter type, e.g. for CPU time, `Meter_Traits<CPU_Time>::ResultType`

[2]The current parameter setting of `clock_gettime()` makes the MeterPU's CPU time meter measure wall clock time.

[3]The approach by Burtscher et al. [18] to correct power values from the Nvidia NVML library might give better accuracy of GPU power measurement, see Sect. 3.2.4. On the other hand, MeterPU is not coupled with a specific compensation method.

gives the unit type for the final result of a CPU time measurement. The class `Meter` acts as a facade, all types related to its meter type will change as the template expands statically.

MeterPU uses the `Meter` class as the interface exposed to programmers. A class has the advantage that it can hide arbitrarily complex data structures and logic underneath, and keeps the exposed part simple and unified, thus MeterPU has the potential and possibility to unify the API design for all measurements of metrics of interest, and acts as an important step towards this goal. MeterPU is designed not only to make a legacy optimization framework based on empirical sampling and modeling easy to migrate to other optimization goals, but also targets at proposing a general interface standard, thus different vendors could hook their own abstracted measurement implementation if necessary. MeterPU is also designed to be extensible, thus it can adapt to metrics that may appear in the future.

3.2.3 More Examples

We provide more examples of MeterPU code to show the expressiveness of MeterPU. Listing 3.3 show how MeterPU can be used to measure CPU energy (`cpu_energy_meter`) and Nvidia GPU energy (`gpu_energy_meter`) with a unified API; only the template parameter to initialize a meter differs in these scenarios (other code to use these meters is the same as in Listing 3.2). For GPU energy measurement, there is one extra template needed, the device id of a GPU, used to distinguish between different GPUs in a multi-GPU system. For example, to initialize a meter associated with a GPU of device id 3, one can write: `"Meter< NVML_Energy<3> > meter"` and `"Meter< NVML_Energy<> > meter"` for the GPU with device id 0 as default, which is usually the case on machines with only one Nvidia GPU.

```
Meter<PCM_Energy> cpu_energy_meter;
Meter< NVML_Energy<> > gpu_energy_meter;
Meter<CUDA_Time> nvidia_gpu_time_meter;
Meter<CUDA_Multiple_Time<0,1> > gpu_cuda_time_meters;
Meter< System_Energy<GPU_0> > system_energy_meter;
Meter<Wattsup_Energy> wattsup_meter;
```

Listing 3.3: Code snippets to initialize different kinds of meters

MeterPU can initialize meters not only for individual hardware components, but also for combinations of those components, while keeping the usage of those meters the same as individual ones. For homogeneous hardware components, MeterPU can easily build a meter associated with them, e.g. `gpu_cuda_time_meters` in Listing 3.3, which can measure kernels that run on multiple GPUs, and only requires the set of GPU ids specified. This feature allows a small constant number of lines of code (LOC) to use MeterPU when increasing the number of homogeneous hardware components to be measured. For heterogeneous components, we can, for example, use combinations of energy meters for CPUs, DRAM and a GPU by `system_energy_meter`[4] in Listing 3.3.

Allowing combinations of meters brings several benefits: first, it is easier to use, as optimizing energy on a system or a combination of important hardware components is usually preferred. Second, MeterPU will enforce the sequence in which different meters start, thus possible overhead (like energy overhead for spawning a new thread of continuous sampling) is factored out. Third, by metaprogramming for building meters, some runtime overhead like a `for` loop over many homogeneous meters is removed, which may yield more accurate measurement values[5].

One limitation of the current implementation for system meters for energy measurement is the measurement overhead in terms of energy. When measuring energy of an Nvidia GPU, a CPU thread per meter is spawned to periodically poll power samples from the GPU. Although the energy overhead of thread creation and destruction is not included, the energy spent on the CPU thread to periodically poll GPU power samples is included in the system meter measurement, and leads to an overestimation of energy values consumed by the CPU and the GPU. We however expect that this part of energy is linear in the time of the polling period and can be eliminated a-posteriori in a future version of MeterPU.

MeterPU does not force to collect pairs of time-series measurements, it depends on its plugins. The current plugin to measure CPU energy just return one value as it uses Intel PCM which directly returns the final energy value of our interest. Measuring GPU energy requires to poll time-series measurements. The power integration overhead is not included in the measurement overhead. This is why we separated calc() (this function does power integration if needed) from stop() (this function only stops the thread that polls the power data if we are measuring GPU energy) in our MeterPU API design. Whether the time overhead for a thread for polling time-series measurements is included or not depends on how one measures it, if only GPU energy is measured, such overhead is not included in the measurement. Such overhead only exists when we use the MeterPU system energy plugin.

The MeterPU API can also be applied for external measurement devices. We developed a plugin for the Wattsup .Net power meter. When Wattsup is connected to a computer that runs MeterPU, we can use `wattsup_meter` in Listing 3.3 to control the measurement, and get the calculated energy value from measured power values by numerical integration. MeterPU can also be synchronized for device startup time[6] and neglect error samplings (due to Wattsup device hardware error) automatically.

[4]We call this combination system_meter. So far we do not support separate measurement for different CPUs; if necessary we can extend MeterPU in the future

[5]Metaprogramming enforces loop unrolling for initializing many homogeneous meters, thus avoid some branch instructions when initializing every next meter in the loop.

[6]For Wattsup power meter, we observed a relatively long latency before obtaining the first measurement data

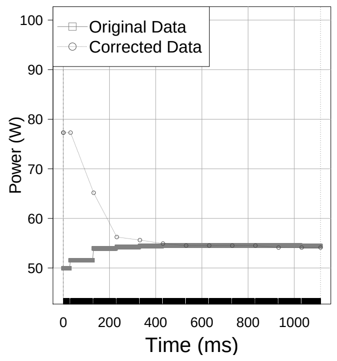


Figure 3.1: GPU power visualization by MeterPU. The vertical dashed lines denote the time when the kernel under measurement starts and ends.

3.2.4 MeterPU Support for Visualization

MeterPU can visualize the measurement samples obtained by different plugins for analysis and debugging purpose. Figure 3.1 shows a power visualization for a GPU kernel execution wrapped by one `start()` and one `stop()`. The red curve shows the original samples by MeterPU's `NVML_Energy` plugin (internally use Nvidia NVML library), and the blue curve shows the corrected power samples by Burtscher's approach [18], with redundant power samples removed. Based on those corrected values, the final energy value is calculated by numerical integration in MeterPU.

3.2.5 Discussion

Energy measurement can both be performed by hardware-based measurements (e.g., Wattsup power meter, shunts and A/D converters) and software methods using models and hardware counters. Hardware-based measurements can offer power samples at higher precision and sampling rate than software methods without overestimation by the overhead of maintaining a power-polling thread. However, from the optimization's point of view, using energy feedback by software methods is more practical and may dominate in the tuning framework use cases, because usually the energy cost is hardware- and software-dependent, and hard to predict analytically by a single general model (also hardware evolves fast). Thus measurement at deployment time is usually necessary, but deploying hardware instrumentations on every machine (where due to heterogeneity, different machines' configurations usually differ, and deployment of hardware measurement equipment requires understanding of each machine's cable connections) is not very possible, and hosting those hardware instrumentations (e.g. A/D converter) for an energy-tuning framework with on-line training often requires a separate host computer, which may cost much more energy than the savings by the tuning framework.

3.3 Combining SkePU with MeterPU

SkePU (see Section 2.6) is an open-source C++ based skeleton programming library for GPU-based systems. SkePU supports multi-platform code generation (C, OpenMP, CUDA, OpenCL, StarPU) including support for multi-GPU execution and hybrid execution, from the same high-level source code.

SkePU's adaptive off-line tuning [31] is an implementation selection methodology consisting of an efficient (time) sampling strategy, deployment time offline training by smart sampling, and generation of a compact dispatch data structure used in dynamic selection of the predicted best implementation variant for a call to a multi-variant skeleton instance. More details of this technique can be found in Chapter 4.2 and [79, 81].

With MeterPU integrated, automatic back-end selection in SkePU can migrate from time optimization to energy optimization easily. The automatic back-end selection in SkePU is based on a predictor that predict the fastest implementation variant. We are interested in migrating the predictor so that it predicts the implementation variant that consumes the least energy consumption instead of execution time. Usually we optimize for the total system energy, not the energy of a specific hardware component. Even if a CPU implementation is invoked, the GPU on the target machine is not turned off and we can not only measure CPU energy when only a CPU implementation is invoked. Thus we use a system energy meter as described in Section 3.2.3 for each of the wrappers.

In SkePU, the time sampling for different skeleton calls with user-defined code snippets is encapsulated in separate wrapper functions in back-ends for different platforms (CPU, OpenMP, CUDA etc). Then we replace the time measurement region marker code in SkePU's off-line sampling module [31] with MeterPU `start()` and `stop()` calls, calculate the energy value by `calc()`, and finally pass the value by `get_value()` to the tuning framework. Now SkePU is ready to tune for reducing energy cost. Furthermore, the code for the initialization of different types of meters can co-exist and be guarded by macros, which makes the switching between different optimization goals as easy as passing a different compiler flag. One can also easily make such switching at run-time without recompiling.

It is worth mentioning that we don't measure system power. We measure system energy by aggregating energy values from both Intel PCM and Nvidia NVML (with power integration). Calling Intel PCM routines directly returns energy value for the Intel CPU and DRAM in the target system, while Nvidia NVML yields time-series measurements for Nivida GPUs, more precisely the time stamp and power value pairs. Intel PCM and Nvidia NVML are handled as two different plugins, and a system energy meter is constructed on top of this two primitive plugins. Our interest is to reduce

Skeleton type	Description	User function
Map	$b_i = f(a_i), i = 1, ..., n$	return a*a;
	$c_i = f(a_i, b_i), i = 1, ..., n$	return a*b;
Reduce	$d = f(a_1, a_2, ..., a_n), i = 1, ..., n$	return a+b;
Mapreduce	$d = g(f(a_1, b_1), ..., f(a_n, b_n))$, $i = 1, ..., n$	return a*b; //for map return a+b; //for reduce
Mapoverlap	$b_i = f(a_{i-t}, ..., a_{i+t})$, $i = t, ..., n - t$	return (a[-2]*4 + a[-1]*2 + a[0]*1 + a[1]*2 + a[2]*4)/5;
Maparray	$c_i = f(b_i, a_1, ..., a_n), i = 1, ..., n$	//vector permute int index = (int)b; return a[index];

Table 3.1: Setup for different SkePU skeletons. (a, b, c: vector. d: scalar. t: positive integer constant.)

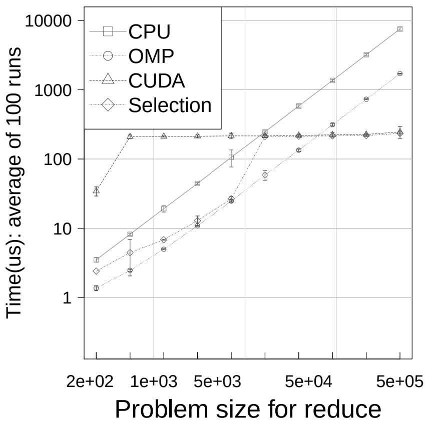
total energy consumption of a computation task, so the instant power values are less interesting in this context, although average power can be calculated by measuring time and energy at the same time.

SkePU's automatic backend selector is not a full-fledged scheduler, it only predicts which back-end runs faster and invokes that back-end implementation. SkePU has some support for hybrid computing like asynchronous calls, but it is not as general as a scheduler. Thus when we combine MeterPU with SkePU, either CPU or GPU are busy for a component call, but not both of them. The implementation variant selector in SkePU only requires to predict which implementation variant performs better. Thus predicting the absolute value, e.g., how much time or energy an implementation variant will spend, although interesting in quite many contexts, is not necessary in SkePU's case.

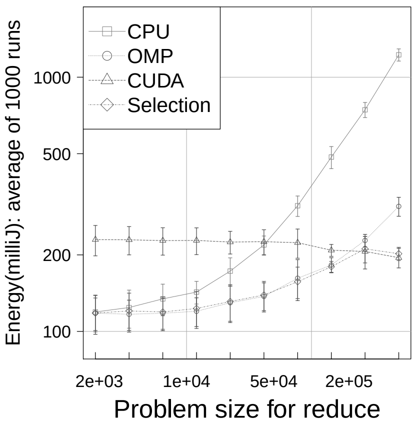
3.4 Experimental Results and Discussion

3.4.1 Experimental Setup

In order to evaluate energy-tuned SkePU with MeterPU, we use SkePU v1.1.1 on an Intel Xeon (E5-2630L v2) server with a Nvidia K20c GPU; During all experiments, the same configuration for the machine is used with all cores and the GPU switched on. We start with the training of SkePU skeletons (the skeletons used and their user functions are listed in Table 3.1), and the tuning framework will sample training runs by MeterPU to build empirical models (for details on how to build such empirical models, please see Section 4.2) for guidance of dynamic implementation switching. Then we



(a) Time tuning for reduce.

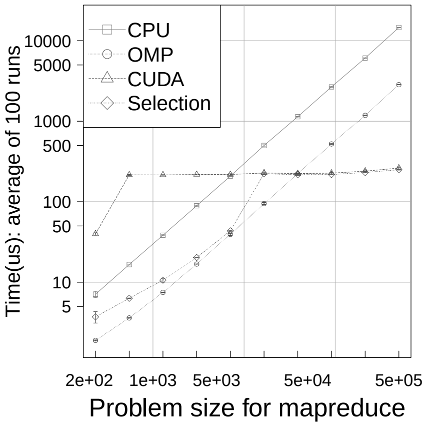


(b) Energy tuning for reduce.

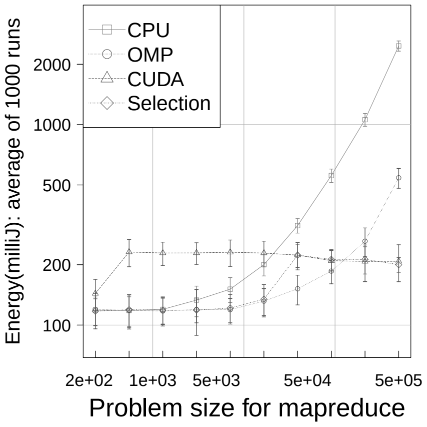
Figure 3.2: Tuning SkePU skeletons with MeterPU (1)

generate test cases for different call contexts (problem sizes), and measure the time speedups and energy savings. The dynamic selection is performed always on CPU, even if the GPU implementation is selected, which will cause some energy cost for dynamic implementation selection spent on CPU, thus we use a system meter as described in Section 3.2.3 to measure CPU and GPU energy at the same time. Meanwhile we also avoid the problem of having to initialize the correct meter statically for dynamically-known choice of component invocations, e.g., to initialize a MeterPU meter requires to know statically whether the meter measures some CPU side metric or GPU side metric, but the dynamic implementation selection choice which decides where (CPU or GPU) to measure is not statically known.

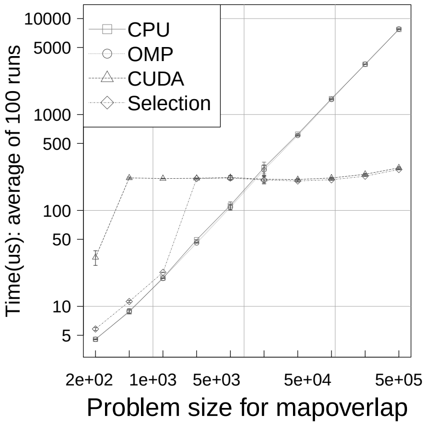
Regarding the energy cost of CPU-GPU communications, as described in Section 3.2.2, CPU side PCIe communication cost is not measured by MeterPU's currently implemented meters, only the GPU side PCIe communication cost is included. In our experiments, we ignore communication cost. By the usage of smart containers [27], data transfer only happens at the first execution of iterative computations over the same skeleton with the same operands, and we discard the measurement for the first execution. Thus in the subsequent skeleton calls, no data transfers are performed, and the energy changes by software components are well captured by the MeterPU system meters. For programs where the communication cost is significant, the energy measurement plug-in needs to be extended to include CPU-side PCIe energy.



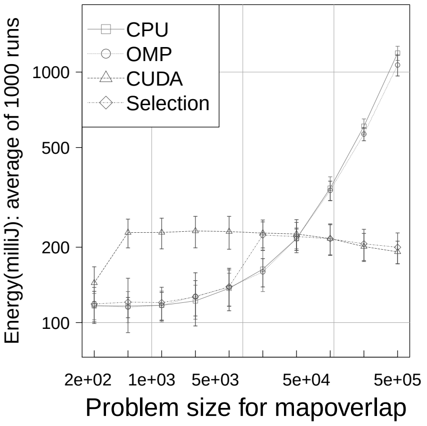
(a) Time tuning for mapreduce skeleton.



(b) Energy tuning for mapreduce skeleton.



(c) Time tuning for mapoverlap skeleton.



(d) Energy tuning for mapoverlap skeleton.

Figure 3.3: Tuning SkePU skeletons with MeterPU (2)

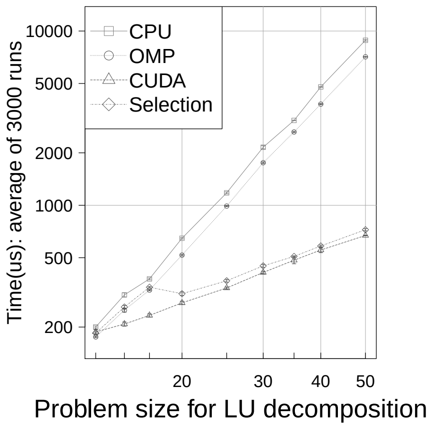
3.4.2 Tuning for Individual Skeletons

Figures 3.2 and 3.3 show the results obtained by applying the user functions of Table 3.1 on different skeletons. The left hand side shows the time for CPU (sequential), OpenMP, CUDA and time-optimizing selection, and the right hand side shows the energy of these scenarios with energy-optimizing selection. We can see that in most cases the time and energy cost of the smart selection switches to the least-cost software component as the problem size (i.e., operand size) changes. The selection is non-optimal only in few test cases, which is an artifact of a too shallow maximum depth of recursive subdivision in adaptive sampling (see Chapter 4.2 and [79, 81]), and better accuracy can be achieved by increasing the training depth [31, 81]. The time and energy savings for large problem sizes can be remarkable, e.g. for a map skeleton, time speedup can reach up to 647× (calculated by the time cost of multi-threaded CPU implementation and smart selection at the largest problem size experimented) if a more throughput-efficient processor type is chosen, and for energy the savings can be about 10×; furthermore, the time speedup and energy reduction factor will continue to increase as problem size grows even larger. The overhead of smart selections for both time and energy is negligible.

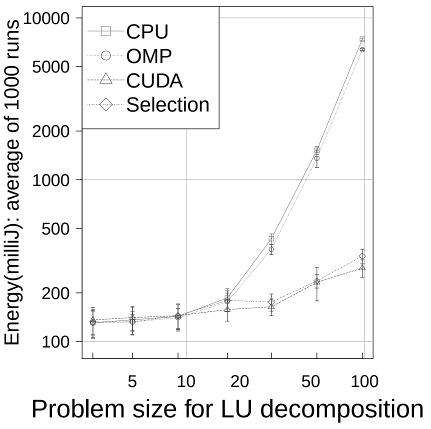
We also observe that the transition points usually differ where the most time-efficient, respectively the most energy-efficient, implementation for the same skeleton switches. This makes it necessary for separate training and sampling when switching the optimization goal, and an abstraction of sampling and measurement such as MeterPU is necessary to facilitate the construction of different prediction models. Here the general behavior of time and energy for the computations is not too dissimilar, especially on our K20c GPU where the static (idle) power is already about two thirds of the full-load power, due to power capping, and hence the majority of GPU energy is linear in GPU time.

3.4.3 Tuning for LU Decomposition

In order to show that SkePU tuning can be applied in the area of linear algebra and linear system solving for the reduction of execution time and energy, an experiment is performed on a frequently used linear system solving computation, LU decomposition, where the MapArray skeleton is heavily used. Figure 3.4 shows that in almost all cases SkePU chooses the implementation that has the highest available efficiency in time or energy. Tuned SkePU provides up to 12× in time speedup and up to 21× in energy reduction using the OpenMP implementation as a baseline, and the savings will continue to increase as problem size increases above the maximum used in our experiments.

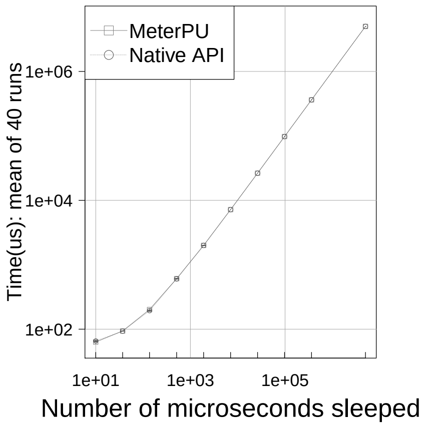


(a) Time tuning

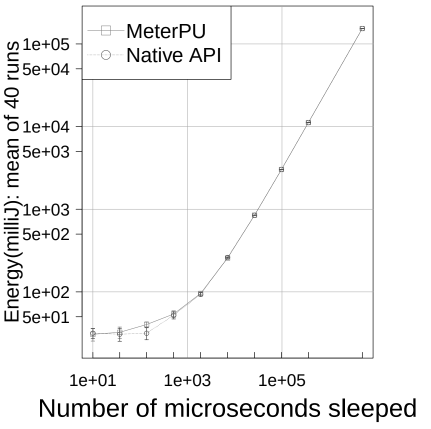


(b) Energy tuning

Figure 3.4: Time and energy tuning of LU decomposition by SkePU and MeterPU



(a) MeterPU time overhead



(b) MeterPU energy overhead

Figure 3.5: MeterPU overhead, including 95% confidence interval on mean value.

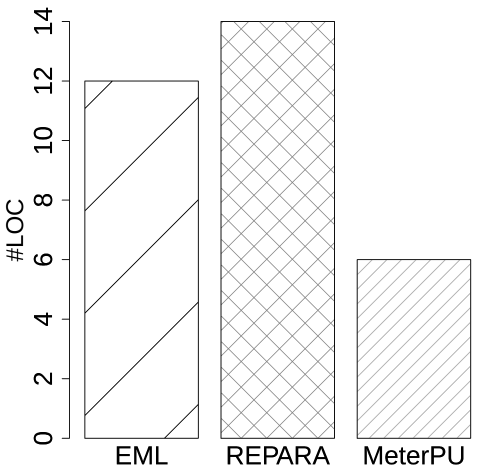


Figure 3.6: LOC (lines of code) comparison

3.4.4 MeterPU Overhead

MeterPU provides a software abstraction of measurement for various metrics and various measurement techniques/methods on different hardware components. Usually an abstraction comes at a cost; in this case, the cost may distort the measurement values. In the MeterPU implementation, when `meter.start()` is called, it delegates the call to the native API, such as `clock_gettime()` etc. Thus the overhead from MeterPU is just one function call, which can even be inlined for some meter types (e.g. `CPU_Time` meter). Figure 3.5 shows the comparison of measurement values between MeterPU and native API (time by `clock_gettime()`, and energy by Intel PCM library for CPU energy, since MeterPU runs on CPU). We can see that for large problem sizes, the overhead is too small to be observed. For small problem sizes (with less than 100 μs for time and 100 mJ for energy), we can observe that the native measurement facility is not accurate, because we measure a kernel of a sleep function, thus the measured value should be strictly linear in the problem size, but in those regions of small problem sizes, the values are above the expected linear values. To sum up, firstly MeterPU is a negligible-overhead abstraction, secondly, the resolution of those native measurement facilities is limited, and below their resolution limits the accuracy decreases to some extent.

3.4.5 Comparison to Other Alternatives

An abstraction's main purpose is to hide uninteresting details, thus we use lines of code (LOC)[7] as the main evaluation metric to compare MeterPU with its alternatives: EML [20] and REPARA [73], which also provide an abstraction measurement layer, described in section 3.5.1. We measure the logic lines of code needed for the measurement of a code region, including the calculation of a final metric value, as shown in Figure 3.6. It is clear that MeterPU requires much less code (only half compared to the second best). The reason why code using MeterPU is much shorter compared to other alternatives is that the initialization and destruction of a measurement is in a MeterPU meter class's constructor and destructor, and those code will be called automatically and completely hidden from programmers. The code for error handling during measurement is also hidden from programmers. MeterPU's advantage on LOC will be larger if several meters of the same type are required, as by variadic template a complex meter will be constructed at compile time, and MeterPU keeps its LOC constant compared to other alternatives. This helps code transformation tools to automatically transform legacy code to be energy-tuned or tuned towards other user-defined metrics supported or extended by MeterPU.

3.5 Related Work

3.5.1 Measurement Abstraction

Several software libraries have been proposed to act as an abstraction of (energy) measurement on different hardware components. EML [20] is a C library that implements a software abstraction of measurement with low overhead (1.54%), supporting dynamic discovery of measurement devices. REPARA's performance and energy monitoring library [73] implemented in C++, provides a unified interface to support both counter-based and hardware-based measurement methods, which can be discovered by their hardware description language HPP-DL. Comparing to those libraries, MeterPU provides the simplest interface while maintaining generality, which facilitates retargeting legacy tuning frameworks, and keeps the overhead minimal (only one extra function call).

Monitoring frameworks, e.g. Nagios [60] and GroundWork [49], take measurements in either intrusive or non-intrusive way while applications are running, and store performance data to file systems or databases. However, from the optimization's point of view, retrieving performance data from file systems or databases to calculate an aggregate metric value at runtime in a feedback loop adds additional overhead compared with a light-weight measurement abstraction such as MeterPU. Another monitoring framework,

[7]We also consider the metric cyclomatic complexity [105], as the control flow in this context is simple, thus using this metric will not add extra information.

PowerAPI [14], is implemented in Scala language with the goal of enhancing the capabilities for monitoring programs written in different languages. MeterPU is implemented in C++, with potentially better integration in main-stream heterogeneous programming models such as OpenCL, CUDA, etc.

3.5.2 Skeleton Programming

Besides SkePU, there are other skeleton programming frameworks targeting heterogeneous systems. SkelCL [116] provides OpenCL-based high-level skeletons and data types to ease the development of programming on heterogeneous systems. MueSLi [39] is a C++ template library with various data and task parallel skeletons where the users may specify where the skeletons are executed (CPUs or GPUs). Fastflow [46] provides layered abstractions on cache-coherent shared memory multicores, with skeleton support for typical streaming patterns. Marrow [88] is a skeleton framework for OpenCL computations, enabling combinations, nesting of skeletons and overlapping of communication and computation. Comparing to these frameworks, SkePU provides automated implementation (backend) selection for both time and energy optimization.

3.6 Summary and Future Work

We developed MeterPU as a software abstraction for measurements of various metrics on different types of processors in a heterogeneous computer system, and demonstrated that the MeterPU API can be applied for both hardware- and software-based measurement methods, and on both single hardware components and (homogeneous and heterogeneous) combinations of those components. It can facilitate both the reuse of legacy tuning frameworks and the switching among different optimization goals on those frameworks, as shown by integrating it with SkePU. We thereby also extended SkePU's individual skeletons to be energy-tunable, and showed that energy-tuned SkePU can reduce energy consumption similarly as time-tuned SkePU accelerating execution time, e.g. for computations in the area of linear algebra, as demonstrated with LU decomposition as an example.

Future work includes extending MeterPU to support measurement of the energy of PCIe communication on the motherboard side, and testing SkePU's energy tunability on more complex applications. MeterPU may also support multi-objective optimization by for example easily providing measurements of different metrics for the analysis of the Pareto-front data points.

MeterPU is publically available at
`http://www.ida.liu.se/labs/pelab/meterpu/`.

Chapter Acknowledgements

Research partially funded by EU FP7 project EXCESS and SeRC project OpCoReS. We thank Oleg Sysoev from Linköping University for suggestions on statistical data handling. We thank Dennis Hoppe from HLRS Stuttgart, Erik Hansson from Linköping University, Paul Renaud-Goud from Chalmers and all other EXCESS project members for comments on this work.

Chapter 4

Handling CPU-GPU Selection Complexity

This chapter is mostly based on the following paper:

- Li, L., Dastgeer, U., and Kessler, C. (2016). Pruning Strategies in Adaptive Off-line Tuning for Optimized Composition of Components on Heterogeneous Systems. *Parallel Computing*, 51:37–45

  which is based on

- Li, L., Dastgeer, U., and Kessler, C. (2014). Pruning Strategies in Adaptive Off-line Tuning for Optimized Composition of Components on Heterogeneous Systems. In *Proc. Seventh International Workshop on Parallel Programming Models and Systems Software for High-End Computing (P2S2) at ICPP*. IEEE

  and on

- Li, L., Dastgeer, U., and Kessler, C. (2013). Adaptive off-line tuning for optimized composition of components for heterogeneous many-core systems. In *High Performance Computing for Computational Science-VECPAR 2012*, pages 329–345. Springer

Another complexity that we will face quite early when utilizing GPU-based heterogeneous systems concerns the decision when to use GPU and when not to. We know that CPU and GPU are not Pareto-optimal over each other, GPU is advantageous in performance under at least these conditions: the computation is data-parallel or close to data-parallel, and the problem size is large enough. Switching to a most suitable type of processor or a set of most suitable processors smartly can lead significant speedups.

How do we decide precisely when to use CPU or GPU? Is the selection simple or complex? How to reuse legacy code easily? How to design the selection mechanism to be performance portable? This chapter will explore these questions.

Section 4.1 discusses the complexity of CPU-GPU selection. Section 4.2 proposes a full-phase mechanism for CPU-GPU selection, including a training mechanism and a run-time CPU-GPU selector. Section 4.3 discusses three techniques to further improve the efficiency of the methods in Section 4.2. Section 4.4 proposes a framework design to transform the legacy code into a form that is tunable by CPU-GPU selectors.

4.1 CPU-GPU Selection Complexity

CPU-GPU selection is a complex problem, due to the following reasons.

- The selection is hardware-dependent. If a system is equipped with a faster GPU (e.g., by higher clock frequency), then the threshold of using that GPU for a performance gain becomes lower, and CPU-GPU selection should favor selecting GPU in more cases.

- The selection is software-dependent. If the computation under consideration is GPU-effective, e.g., data-parallel, then the threshold to use GPU becomes lower, and GPU should be selected in more cases.

- The selection is run-time context dependent. Take problem size as an example of a run-time context property, if the problem size is larger, then better GPU occupancy [95] may be achieved, thus GPUs are likely to outperform CPUs in this and similar cases. Another important run-time context property is the current location of the most recent argument data, as data transfer cost is expensive and can sometimes dominate the CPU-GPU selection. Due to the time limit, in this thesis we have not taken data location into consideration when performing CPU-GPU selection.

Thus, CPU-GPU selection is complex and best delayed to run-time, when run-time context property values are known. In the next section we propose such a design for run-time selection which allows to reuse legacy code easily.

In order to perform CPU-GPU selection for a computation, several implementation variants that can run on CPU or (and) GPU are necessary. Thus, the CPU-GPU selection problem becomes the problem of implementation variant selection. Our main research interest is addressing how to select the implementation variants, thus we assume that those implementation variants already exist as legacy code.

4.2 Adaptive Sampling: a Training Space Exploration Technique

We propose a machine learning approach to tackle the complexity of CPU-GPU selection, or more precisely, implementation variant selection, for the following reasons:

- The logic between the input (run-time context, component type, hardware configuration) and output (a winner implementation variant) is not straight-forward for humans to extract and thus hard to program by hand. New hardware may bring new features that can affect the implementation selection decisions which can be hidden and hard to detect. Hardware features used in this thesis are the hardware architectural parameters that come along with each generation of GPUs, such as number of cores, number of SMs, cache sizes etc.

- If we select a sub-optimal variant for an input configuration, there are no catastrophic consequences (e.g. program crashes, application failures), only leading to a slowdown of program executions.

- Although possible, it is at least time-consuming to find and program the logic between the input and output. Considering that hardware is evolving fast, it is arguable that the effort spent is worthwhile. Hardware features changes for each generation and for different GPU products of the same generation. This change of such hardware changes will impact the performance of some specific implementation variants in a unclear way (e.g., it is hard to say how much will the same code run faster on new generation of GPUs), where machine learning techniques fit well. Since each generation brings a mix of such changes, it is hard to say how much a implementation variant will run faster. The speedup on each implementation variant will affect their ranking compared to other variants, thus affect the final implementation selection choice in a unclear way as well.

For the setup of our method, we ignore modeling of hardware features explicitly by assuming that hardware configurations are fixed when programs execute, this is true for usual cases. And even if hardware configurations do change during run-time, they tend to change regularly[1], thus the execution time of a computation with some problem size tends to be stable, which legitimates our assumption. We are interested in components whose execution time is stable algorithmically (not necessarily linear) thus predictable, e.g., no data-dependent time complexity algorithms and no randomized algorithms inside a component.

In order to address the type of computations on which the prediction of the best implementation variant depends, we build a predictor per *ceset*.

[1]E.g., always increase processor frequency when a certain type of workload is issued.

To address the run-time context such as problem size, we use a machine learning technique to automatically learn the function from the run-time context to the best-performing implementation variant.

The main challenge that remains is to find a good training data set to train our predictor. In contrast with supervised machine learning approaches on other domains (e.g. face recognition, natural language processing), where training data are usually labelled manually, to obtain a piece of training data that a certain implementation is best-performing for a given input configuration we could simply run all implementation variants and compare their execution latencies. However, this is where we meet the *curse of dimensionality*, where the training space blows up when the dimensionality of the input space increases. In such a large training space, it is difficult to find good training examples. Considering that the training is performed on each component of interest, sampling a large data set for each function is likely to be infeasible, while sampling a small proportion of the whole training space, is feasible.

Section 4.2.1 gives a motivating example to show why implementation selection is necessary. Section 4.2.2 describes our technique for generating good training examples efficiently and how to predict the winner implementation variant. Section 4.2.3 presents experimental results, and finally Section 4.2.4 concludes.

4.2.1 An Motivating Example

Consider a typical example where a component's implementation variants for execution on different kinds of processors show performance advantages for different variants with respect to different input sizes, as shown in Figure 4.1. In a subrange[2] of call context instance values (here, of the number of array elements) where one implementation variant runs fastest among all implementations variants we call that implementation variant the *winner* for that range of input sizes.

The performance-related parameters for a function call can be many, let us denote that we have n such parameters, and each parameter has a valid range, no matter how large or small it can be. Then we have a n-dimensional range for n such parameters. We can map such a n-dimensional range to a n-dimensional space. A specific context instance can also be considered as a point in a n-dimensional space. Some points or hyperplanes divide winning ranges of different implementations, we call those the transition points or hyperplanes. Ideally if all those points or hyperplanes can be found effectively, we can construct a compact representation which requires small overhead for both store and look-up, and it will provide 100 percent accuracy of winner prediction. We define how we measure accuracy as follows: for

[2]For example, the range [5,8] is a subrange of the 1-dimensional space [1,10], and this applies to multi-dimensional spaces as well, e.g., the 2-dimensional range [[3,7], [5,8]] is a subrange of the two dimensional space [[1,10],[1,20]]

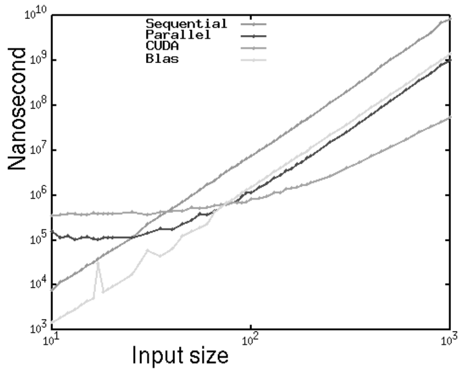


Figure 4.1: Performance for matrix-matrix multiplication variants

each tuple of performance-affecting context property values, the execution time of all implementations is measured to get the winner with respect to that parameter setting, and the prediction is run on the same vector to get the predicted winner. The prediction counts as correct if the winner and the predicted winner are the same. The accuracy is calculated as the percentage of the correct predictions out of all test cases.

One may argue that the characteristics shown in Figure 4.1 may not apply for other problems. In this section, we test three benchmark applications, and these applications surprisingly conform to the characteristics of Figure 4.1, which shows an interesting property: The winning range for each implementation variant is convex, i.e., if two points in the one-dimensional space have the same winner, then this winner wins on all points between these. Our pruning strategy is based on this convexity assumption: for a n-dimensional space, if all vertices of a subspace have the same winner, then it wins on all points in the subspace at least in majority cases. (For more discussions about the safety of the convexity assumption, please see Section 4.3.2) Based on this assumption, we construct an algorithm and data structure to approximate and represent these transition points.

4.2.2 Adaptive Sampling and Prediction

For a concise terminology, we adopt that *composition* is the selection of a specific implementation variant (i.e., callee) for a call to component-provided functionality and the allocation of resources for its execution. Composition is made *context-aware* for performance optimization if it depends on the current *call context*, which consists of selected operand properties (such as size) and currently available resources (such as cores or accelerators). Such compositions also depend on the locations of the involved components' operands which determines whether data transfer should be incurred, and whether other tasks are also executing on some hardware resources, we haven't considered these two factors in this thesis. The context properties to be considered and optionally their *value ranges* (e.g., minimum and maximum value) are declared in the TunerPU API. We refer to a call's arguments shortly as a *context instance*, which is a tuple of concrete values for context properties that might influence callee selection. Hence, composition maps context instances to implementation variants [70].

Composition can be done either statically or dynamically. *Static composition* selects a implementation variant at compile time with no run-time overhead at the cost of being not run-time context-aware. *Dynamic composition* makes such decisions at run-time with run-time overhead but can potentially make a better decision considering run-time context. The hope is that the time saved by invoking the fastest implementation variant is larger than the overhead of the dynamic selection process, and thus performance portability is increased.

Dynamic composition with on-line training by the runtime system shows some disadvantages: it requires a certain number of representative executions before it can offer acceptable selection accuracy for dynamic composition; however, it is often not guaranteed that those representative executions will happen during a sufficiently long period of time. The online training methods, as implemented e.g., in StarPU [8], build performance models incrementally at runtime when useful computations are executed and these measurements are obtained (almost) for free. However, online tuning can only perform well if enough well-distributed training examples are gathered, and it may experience a relatively long period of learning with still suboptimal selections, which we refer to as the "cold-start effect".

As an alternative, we consider off-line training and dynamic composition. In off-line training, measuring performance for every possible runtime context instance (which would offer perfect selection and precise representation of this information) is often not feasible, thus a dynamic composer is forced to make predictions based on a limited set of training examples.

Next, we discuss our adaptive sampling techniques for generating good training examples, and an example for an illustration of adaptive sampling.

4.2.2.1 Adaptive Sampling

The space $\mathcal{C} = I_1 \times ... \times I_D$ of context instances for a component with D properties in the context instances is spanned by the D context property axes with considered (user-specified or default) finite intervals I_i of discrete values, for $i = 1, ..., D$. A continuous subinterval of an I_i is called a *range*, and any cross product of such subintervals on the D axes is called a *subspace* of $\mathcal{C}$. Hence, subspaces are "rectangular", i.e., subspace borders are orthogonal to the axes of $\mathcal{C}$.

We offer a accuracy-controllable offline-trainer and dynamic composer based on ranges, i.e., the trainer tries to automatically approximate the (usually, non-rectangular and possibly non-convex) subsets in $\mathcal{C}$ where one particular implementation variant performs better than all the others, by a set of subspaces.

Our idea is to find sufficiently precise approximations by adaptively recursive splitting of subspaces by splitting the intervals I_i, $i = 1, ..., D$. Hence, subspaces are organized in a hierarchical way (following the subspace inclusion relation) and represented by a 2^D-ary tree (cf. binary space partitioning trees and quadtrees/octrees etc.).

Our algorithm for off-line measurement starts from a trivial tree $T_{\mathcal{C}}$ that has just one node, the root (corresponding to the whole $\mathcal{C}$), which is linked to its 2^D corner points (here, the 2^D outer corners of $\mathcal{C}$) that are stored in a separate table of recorded performance measurements. The implementation variants of the component under examination are run with each of the corresponding 2^D context instances, possibly multiple times for averaging, using a context instance generator provided with the metadata of the component; a variant whose execution exceeds a timeout for a context instance is aborted and not considered further for that context instance. Now we know the winning implementation variant for each corner point and store it in the performance table, too, and $T_{\mathcal{C}}$ is properly initialized.

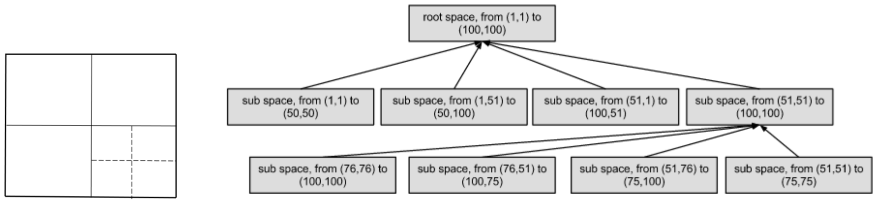

root space, from (1,1) to (100,100)
sub space, from (1,1) to (50,50)
sub space, from (1,51) to (50,100)
sub space, from (51,1) to (100,51)
sub space, from (51,51) to (100,100)
sub space, from (76,76) to (100,100)
sub space, from (76,51) to (100,75)
sub space, from (51,76) to (75,100)
sub space, from (51,51) to (75,75)

Figure 4.2: Cutting a space recursively into subspaces, and the resulting dispatch tree.

Consider any leaf node v in the current tree T_t representing a subspace $S_v = R_1^v \times ... \times R_D^v$. If the same specific implementation variant runs fastest on all context instances corresponding to the 2^D corners of S_v, we stop further exploration of that subspace and will always select that implementation whenever a context instance at run-time falls within that subspace. Otherwise, the subspace S_v may be refined further. Accordingly, the tree is extended by creating new children below v which correspond to the newly created subspaces of S_v.

By iteratively splitting the ranges in FIFO order, we generate an adaptive tree structure to represent the performance data and selection choices, which we call *dispatch tree.*

The user can specify a *maximum depth* (training depth) for this iterative refinement of the dispatch tree, which implies an upper limit on the runtime lookup time, and also a maximum tree size (number of nodes) beyond which any further refinement is cut off. Third, the user may specify a timeout for overall training time, after which the dispatch tree is considered final.

The prediction is a run-time lookup that searches through the dispatch tree starting from the root and descending into subspace nodes according to

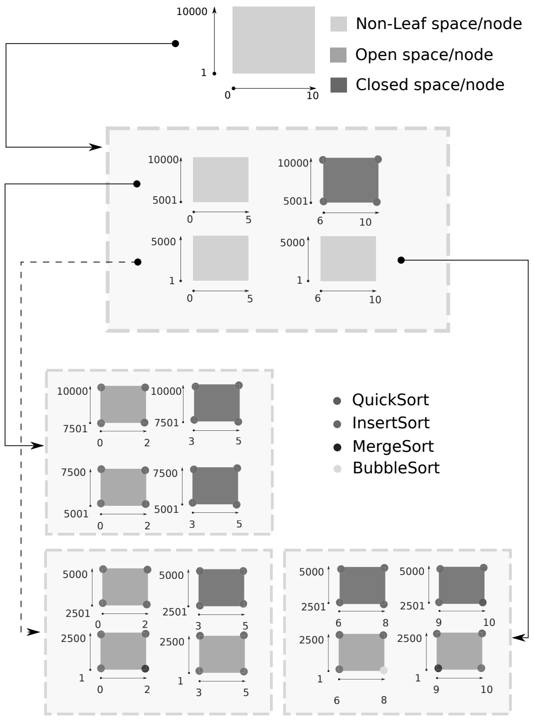


Figure 4.3: Tree visualization for a sorting example

the current runtime context instance. If the search ends at a *closed leaf*, i.e., a leaf node with equal winners on all corners of its subspace, the winning implementation variant can be looked up in the node. If the search ends in an *open leaf* with different winners on its conners (e.g., due to reaching the specified cut-off depth), we perform an approximation within that range by choosing the implementation that runs fastest on the subspace corner with the shortest Euclidean distance from the run-time context instance. The prediction result is an index number which we use to fetch the function address of the winner implementation variant in our dispatch table.

A visualization of an example tree structure is shown in Figure 4.3, which shows that a large area of the training space is pruned during the sampling process. A tree structure is built by the adaptive sampling process with training depth 2 for a sorting component with four implementation variants. Two parameters are chosen for training, the array size ranging from 1 to 10000 and the discretization of (sampled) sortedness of the operand array ranging from 0 to 10, which forms a two dimensional space. At the first recursive division step, the space is divided into four subspaces with one closed for further exploration. In the next recursive step, the three open spaces are further divided and more subspaces are recognized as closed.

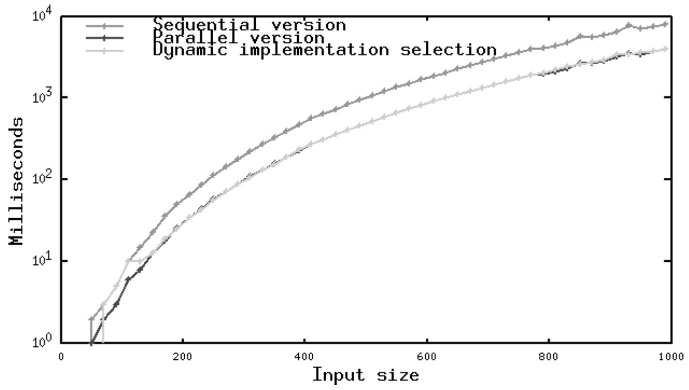


Figure 4.4: Execution time for dynamic composition of matrix-matrix multiplication with a 41-node lookup tree determined by the adaptive refinement training algorithm with cut-off depth 3. — The hardware we use is a multi-core system with 16 CPUs, where each CPU is an Intel(R) Xeon(R) CPU E5520 running at 2.27GHz with 8192 KB cache. The operating system is Linux 3.0-ARCH and the compiler is gcc 4.6.1.

The deeper the algorithm explores the tree, the better accuracy the dynamic composer can offer for the composition choice; however, it requires more off-line training time and more runtime lookup overhead as well. We give the option to let the user decide the trade-off between training time and accuracy by setting the cut-off depth, size and time in the TunerPU API.

4.2.2.2 Example for Dynamic Composition with Adaptive Off-line Training

Let us consider a matrix-matrix multiplication example with two implementation variants, the well-known sequential version and a parallel version parallelized by pthreads with a fixed number of 4 threads. In the off-line training phase, performance data is measured by one execution per context instance; at execution time of the composed code with dynamic selection, performance is averaged over 10 runs per context instance.

As the resources (here, number of threads for OpenMP) is fixed, a context instance is just a triple consisting of the three problem sizes that define the operand matrix dimensions. The training space of context instances is chosen as $[1:1000, 1:1000, 1:1000]$, i.e., comprising 10^9 possible context

Table 4.1: Platform description

Machine name	CPU type	GPU type	OS	Compiler
Fermi	Intel(R) Xeon(R) CPU E5520 @ 2.27GHz	two Tesla M2050	ARCH 3.2.1-2	gcc 4.6.2 and nvcc V0.2.1221
Cora	Intel(R) Xeon(R) CPU X5550 @ 2.67GHz	two nVidia Tesla C2050 and one Tesla C1060	RHEL 5.6	gcc 4.1.2 and nvcc V0.2.1221

instances (input sizes). As tree data structure we use an octree with simultaneous refinement of subspaces along all three dimensions. The cut-off depth for the tree is set to 3. With these settings, the off-line training time (i.e., for the tree construction including the measurements on the target system) takes 228 seconds and the constructed tree has 41 nodes, where the adaptive tree refinement is done mostly for subspaces with smaller problem sizes. By comparing the composed code at runtime with the actually fastest component for each context measured for square test matrices (see Figure 4.4), we find that the tree lookup yields a dynamic selection accuracy of 92%. From Figure 4.4 we can also see that the overhead for performing dynamic selection is rather negligible. For some context instance the dynamically selected implementation variant runs even faster than the same one without dynamic selection; such anomalies are mostly due to the operating system's interruptions during measuring; in principle, the composed code should always run slightly slower than the best individual component, due to run-time lookup overhead.

4.2.3 Experimental Results and Discussions

4.2.3.1 Platforms

We use two GPU based heterogeneous systems called Fermi and Cora. A brief description of the two platforms is shown in Table 4.1.

4.2.3.2 Benchmarks

For the evaluation we have chosen 4 benchmark problems: matrix-matrix multiplication, sorting, and two RODINIA benchmarks: path finder and backpropagation. A detailed description is shown in Table 4.2.

4.2.3.3 Methodology

We first train each benchmark problem with training depth from 0 to 4. If the training time exceeds 3 hours then we terminate the training process. Each benchmark is trained twice, with one version which prunes closed space in the tree representation and another which performs no pruning at all.

The test points are chosen evenly from the training space so that every subspace in the dispatch tree is used for performance prediction.

Table 4.2: Benchmark test settings

Benchmark	Feature modeling	Range	Space size	Implementation variants
Matrix-matrix multiplication	row size, column size of first matrix; column size of second matrix	(1, 1, 1) to (3000, 3000, 3000)	2.7E+10	Sequential implementation, CUDA implementation, BLAS implementation, Pthread implementation
Sorting	array size; discretization of array values distribution (sampled number of inversions)	(1,0) to (100000,10)	1E+6	bubble sort, insertion sort, merge sort, quick sort, CUDA Thrust sort (only on Fermi)
Path finder	row; column	(1,1) to (10000,20000)	2E+8	OMP implementation, CUDA implementation
Back propogation	array size	(1000) to (100000)	9.9E+4	OMP implementation, CUDA implementation

4.2.3.4 Experimental results on two machines

The test results for 4 benchmarks on Fermi is shown in Table 4.3. In particular, for backpropagation, the performance behavior for different training depths on Fermi are shown in Figure 4.5.

The results for the 4 benchmarks on Cora are shown in Tables 4.4.

More results using adaptive sampling to make skeletons and LU decomposition both performance- and energy-tunable are shown in Section 3.4.2 and Section 3.4.3. Matrix-matrix multiplication with more implementation variants are tested in Chapter 7. Other benchmarking results by this tuning approach are shown in [31], including kernels like Nbody, Taylor serious etc.

4.2.3.5 Discussion

For the sorting, pathfinder and backpropagation benchmarks, the accuracy grows quickly with respect to the training depth. For backpropagation the accuracy achieves 100% with only training depth 4. The result for the matrix-matrix multiplication benchmark is a little disappointing, because it has a relatively large training space. Most subspaces in its dispatch tree are open ones and for the points near their corners the Euclidean distance criterion can give a better approximation while in the large central area of these subspaces, the accuracy can not be guaranteed. Since we train on a large space, which means large input sizes, a single training execution may take a long time; for this reason, deep training depths become not practical and are not considered in this benchmark testing.

From the test results we can see that in most cases the accuracy of prediction of the winner implementations increases with the depth of the dispatch tree. This is expected because, as open subspaces can be partly closed by exploring deeper levels, the accuracy increases. This trade-off is exposed to users.

We also can see that for a relatively short training time, we get a reason-

Table 4.3: Test results for 4 benchmarks on Fermi (td: Training depth; tt: Training time; ato: average time overhead on performing dynamic selection; nn: Number of nodes generated in the tree representation)

Matrix-matrix multiplication on Fermi, 343 test points								
td	pruning closed space				no pruning for closed space			
	tt (s)	Accuracy (%)	ato (μs)	nn	tt (s)	Accuracy (%)	ato (μs)	nn
0	85	51	17	1	88	50	15.9	1
1	755	48	21	9	762	48	20.4	9
2	6118	62	23	73	6252	62	23	73
Sorting on Fermi, 110 test points								
pruning closed space					no pruning for closed space			
0	233	36	4	1	233	36	3.6	1
1	1035	61	4.9	5	1035	64	4.9	5
2	2485	80	5.5	17	4071	80	5.6	21
Back propagation on Fermi, 20 test points								
pruning closed space					no pruning for closed space			
0	7	55	9	1	6	55	9	1
1	7	80	11	3	6	80	10	3
2	8	90	13.6	5	8	90	11.8	7
3	7	95	12	7	13	95	12.5	15
4	8	100	13.1	9	18	100	14	31
Path finder on Fermi, 200 test points								
pruning closed space					no pruning for closed space			
0	36	59	12.6	1	29	59	12.7	1
1	161	77	16.5	5	122	77	14.5	5
2	371	86	16.5	17	497	86	15.8	21
3	609	95	16.8	45	1992	95	20.9	85

able prediction accuracy in total which means that pruning closed subspaces works and the assumption that we can treat all points in a closed subspace equally holds for those benchmarks. Another evidence for the assumption is the comparison between two versions of test results, one which performs closed subspace pruning and one which does not. We get almost the same results from the two sets of tests on all benchmarks we use, thus it is considered rather safe not to explore closed space in the training phase on these benchmarks.

The time overhead for run-time selection is acceptable, in the order of microseconds. Since we only explore a shallow depth of a dispatch tree, the number of nodes generated is small, too, so the memory overhead is acceptable as well.

As for the relation between accuracy and performance, we can illustrate it in Figure 4.5 for backpropagation. Comparing constant invocation of the OpenMP implementation variant with dynamic selection among all available variants, we see that for subspaces where the OpenMP variant wins, the performance of all variants only differs by a few microseconds; for the subspace where OpenMP does not win, we gain performance. The performance gain might be remarkable if some variant scaling badly is constantly invoked. From the figures we can also see that wrong decisions for points within open subspaces often happen near transition points between different winners, and often the performance difference of implementation variants at points near transition points is low, thus a wrong decision does not yield a

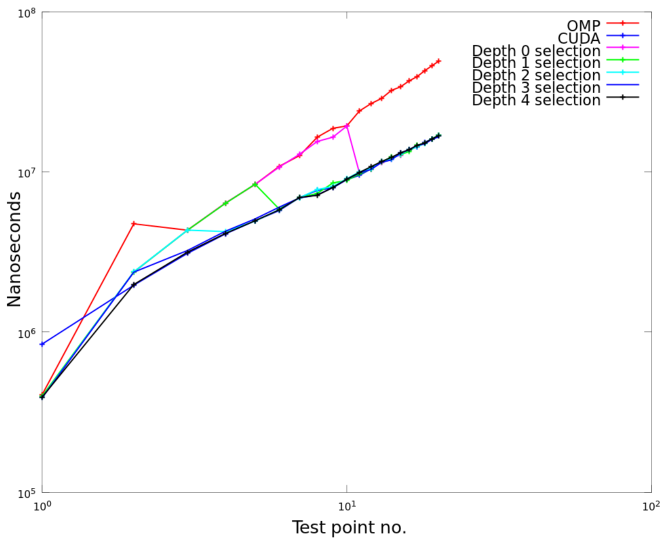


Figure 4.5: Performance with maximum depths 0 to 4 for the backpropagation benchmark on Fermi.

performance penalty as large as in other points in the subspace.

In general, our approach can pick the best implementation variant for most of the cases for the different platforms.

We observed an anomaly for the exploration of subspace in matrix-matrix multiplication on Fermi. When the depth increases from 0 to 1, with more training time, the accuracy drops. One possible explanation is that when splitting some space where the winner on one of the corners is shared by a minority of the other corners, the Euclidean distance criterion will cause a majority of points to be predicted wrongly, which with the coarser dispatch tree are predicted correctly. Continuing to refine that subspace may make the accuracy increase again; however, continuing the exploration for matrix-matrix multiplication on such large space is so time-consuming that we have to postpone further investigation of this problem to future work.

To test on more benchmarks is necessary, but we are currently constrained practically by writing a general problem instance generator to generate an arbitrary-size problem input so that we can invoke computations on it and extract the training examples whose sizes are required by adaptive sampling. Writing such problem instance generator requires application domain knowledge and component writers are in a privileged position to write such generators. This is addressed in more detail in the training_run() bullet in Section 4.4.2, and this motivates the design of TunerPU in Section 4.4.

Table 4.4: Test results for 4 benchmarks on Cora

Matrix-matrix multiplication on Cora, 343 test points								
td	pruning closed space				no pruning for closed space			
	tt (s)	Accuracy (%)	ato (μs)	nn	tt (s)	Accuracy (%)	ato (μs)	nn
0	67	48	17.6	1	63	48	18.3	1
1	634	49	22.5	9	621	49	22.2	9
2	5115	67	26.4	73	5009	68	26.2	73
Sorting on Cora, 110 test points								
pruning closed space					no pruning for closed space			
0	162	34	5.3	1	159	35	5.5	1
1	714	62	7.1	5	710	62	8.5	5
2	1747	80	8.6	17	2809	78	8.6	21
Back propagation on Cora, 20 test points								
pruning closed space					no pruning for closed space			
0	3	55	11.9	1	3	60	13	1
1	4	85	12.6	3	4	90	14.7	3
2	4	95	16.1	5	5	95	14	7
3	4	100	13.4	7	7	95	15.2	15
Path finder on Cora, 200 test points								
pruning closed space					no pruning for closed space			
0	21	39	12.5	1	21	39	12.1	1
1	97	67	14.5	5	92	67	16.1	5
2	219	82	15.6	17	400	82	16.2	21
3	367	95	15.9	45	1511	95	18.1	85

4.2.4 Summary and Future Work

We have developed an adaptive off-line training algorithm and dispatch tree representation that allows to pick the best implementation variants for most of the cases on different GPU-based heterogeneous machines, hence it improves performance portability. Our method allows to reduce training time and enables the user to trade off prediction accuracy, runtime overhead and training time.

Our approach for pruning closed space is based on the assumption that, if corners of a space show a common winner, all points in the space would have the same winner, which holds in most of our benchmark applications. The assumption needs to be further investigated with more applications, and refined prediction methods for open spaces should be developed. Note that, in cases where the user knows that the assumption does not hold, a better accuracy could then be enforced by also refining closed space within the given depth limit, at the expense of a larger dispatch tree and longer training time.

Further improvements of our method are possible and will be considered in future work. For instance, timeouts for individual measurements (applicable on CPUs) and aborting variants under measurement that exceed the current winner of a training execution can save more training time.

4.3 Pruning Strategies of Adaptive Sampling

Although adaptive sampling allows efficient generation of training examples to train our implementation variant selector, it is desirable to make the generation of training examples more efficient, because the training can possibly be performed on each *ceset*. As the number of such sets can potentially be large, training all of them requires more efficiency.

As our adaptive sampling technique heavily depends on the convexity assumption, which also can be viewed as a heuristic and brings the risk to miss interesting training examples if the assumption does not hold in some regions of the whole training space.

In this section we further propose three pruning strategies that allow to enhance the adaptive sampling technique either by higher efficiency or by more safety considering the convexity assumption. Section 4.3.1 describes our pruning strategies. Section 4.3.2 discusses our experimental results.

4.3.1 Three Pruning strategies

Thresholding is a further refinement of our adaptive sampling strategy based on the convexity assumption. Previously, a subspace is considered closed if all winners on its vertices are strictly the same; let us call it a *strictly closed subspace*. Now we will relax this condition by introducing a threshold, or tolerance level, θ: If the relative performance difference of a winning variant v_1 and another variant v_2 at a subspace vertex is within the tolerance, i.e. $|t_1 - t_2|/t_1 \leq \theta$ (where t_1 and t_2 denote execution times of two variants on the same input vector) then v_1 and v_2 are considered equivalent, i.e., both are considered winners. Hence, if all winners on a subspace's vertices are equivalent to each other, then we call the subspace as a *closed space* and thus prune it in the same way as strictly closed subspaces.

If the number of relevant context properties (i.e., space dimension) is relatively large, then the generated subspace tree will still grow relatively fast with increasing tree depth. Furthermore, if the number of implementation variants is also large, the probability that a large subspace is a closed one is deeply decreased, which increases the sampling and training cost. The thresholding approximation allows to increase the probability for closing a space in both cases and reduce the training time, at the expense of a small potential performance loss limited by θ.

Oversampling addresses the convexity assumption, that it is assumed to be statistically (but not exactly) safe to prune closed subspaces without loss of accuracy. Oversampling will explore a subspace further to gain additional confidence whether it is closed or not, by sampling for one or several points in the subspace. Oversampling is expected to improve prediction accuracy at the expense of a slight increase in sampling, training and runtime overhead. Light-weight forms of oversampling, such as light oversampling which only samples the point in the center of a subspace besides the vertices, can be considered as a trade-off between training overhead and optimization potential.

Implementation pruning is a technique that allows to prune implementation variants in the sampling process. We assume that, for a subspace,

Table 4.5: Benchmark test settings

Benchmark	Feature modeling	Range	Implementation variants
Matrix-matrix multiplication (MM)	row size, column size of first matrix, column size of second matrix	(30, 30, 30) to (300, 300, 300)	Sequential impl. and a variant by loop rearrangement, CUDA impl., BLAS impl., Pthread impl. and five of its variants from loop rearrangement
Sorting (ST)	array size; discretization of array values distribution	(1,0) to (10k,10)	bubble sort, insertion sort, merge sort, quick sort
Path finder (PF)	row; column	(1,1) to (10k,20k)	OpenMP impl., CUDA impl.
Backpropagation (BP)	array size	(1k) to (100k)	OpenMP impl., CUDA impl.

the winners of its vertices are more likely to also win on its internal smaller subspaces when split, thus we may consider only the winners of its vertices for sampling on its subspaces, in order to reduce sampling and training time further. For example, consider a one-dimensional problem with 10 implementation variants; in the first round of sampling (the two problem size interval corners), finding the winner for a specific input vector requires at least to run 10 implementation variants on the input vector. Since it is a one-dimensional problem, for the whole context property space at most 2 subspaces are generated by binary split and each subspace will only have at most 2 vertices, and hence at most 2 winners are found on the two vertices. For the next refinement step, finding a winner for a specific input vector within a subspace only requires to run 2 implementation variants on the input vector, greatly reducing the training time. Since a winner is assumed to have higher probability to win on subsequent refinement steps, implementation pruning may only show trivial effects on the relevance of the extracted training examples and the prediction accuracy.

Furthermore, combining the different techniques can be interesting. Oversampling is designed to have a positive effect on the prediction accuracy since it checks extra points to reduce the probability of wrongly closing a subspace. Thresholding and implementation pruning are designed to reduce the training time at a marginal cost of prediction accuracy. Combining them will, theoretically, provide the benefit of improved prediction with less training overhead.

4.3.2 Experimental Results and Discussions

All experiments are performed on a GPU-based heterogeneous system called Cora. Table 4.1 shows a brief description.

The benchmarks that we use are: Matrix-matrix multiplication, Sorting, and two RODINIA [22] benchmarks: Path finder and Back propagation. Table 4.5 shows a detailed description.

The benchmarks are trained with increasing training depth (maximum tree depth for cut-off). For a comparison, each benchmark is trained twice, with or without thresholding resp. oversampling. In some cases, the shallow training depth does not allow for a significant difference, thus these tests are ignored.

For testing the selection accuracy based on the training data, all test parameters are chosen evenly from the whole training space, thus all closed and open spaces are involved in prediction.

For each tuple of performance-affecting context property values, the execution time of all implementations is measured to get the winner with respect to that parameter setting, and the prediction is run on the same vector to get the predicted winner. The prediction counts as correct if the winner and the predicted winner are the same. The accuracy is calculated as the percentage of the correct predictions out of all test cases.

For each test case, we measure accuracy, training cost (training time, number of nodes which reflect space overhead) and run-time overhead. Higher accuracy and lower overhead are desirable, especially for the run-time overhead.

Discussion: Convexity Assumption Our convexity assumption enables us to close a subspace and stop sampling on the area when its vertices share the same winner. Later on, we predict the winner of the inner elements of the subspace as the same as for the vertices. However, one may argue that it may not be safe (for loss-free pruning) to close such a space. Even if it is safe to do so, the spaces closed may only take up a small amount of the total space. In order to investigate on the two questions, we perform a test on all benchmarks in the following way: we choose elements from only the closed spaces recognized by the adaptive sampling process, and test how accurate the prediction is if we treat the inner elements the same as their vertices. We also calculate the percentage of the number of elements in closed spaces related to the total number of elements, as a measure of how frequently a subspace is a closed one.

The test results are shown in Figure 4.6. For the prediction accuracy, we can see that for most benchmarks on every training depth, the prediction accuracy reaches 100%, and more than 90% for the remaining benchmarks. For the very large training spaces, we can safely prune closed spaces without or with trivial loss of optimization potential. For the percentage of the number of elements in closed space, we can see that even for a shallow training depth like depth 2, the ratio reaches more than 50% and we can predict them in a simple way correctly. Thus we can save training time from a fairly large training space.

It is interesting to visualize the real and our predicted landscape for winner spaces as a comparison. Here we choose the path finder benchmark with training depth 2, and the visualization is done at two scales: global scale in Figure 4.7a and one open space (left lower corner) in Figure 4.7b.

The OpenMP-optimal and CUDA-optimal region show the real winner landscape of the regions where OpenMP and CUDA implementation dominate respectively in performance. The region outside the CUDA-predicted-

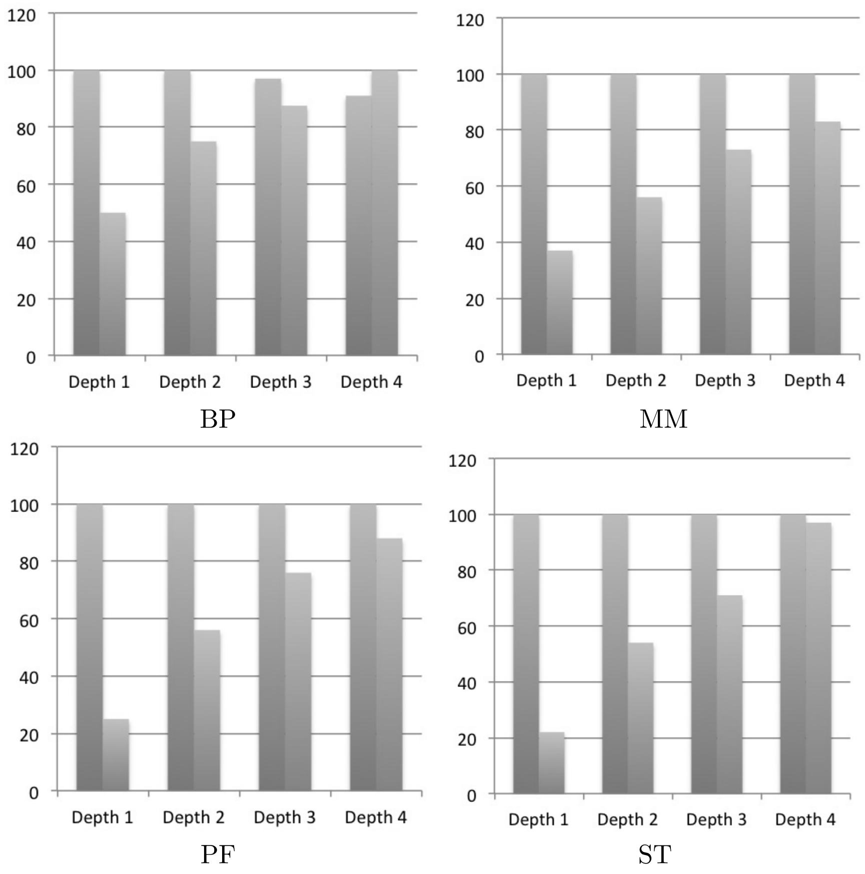
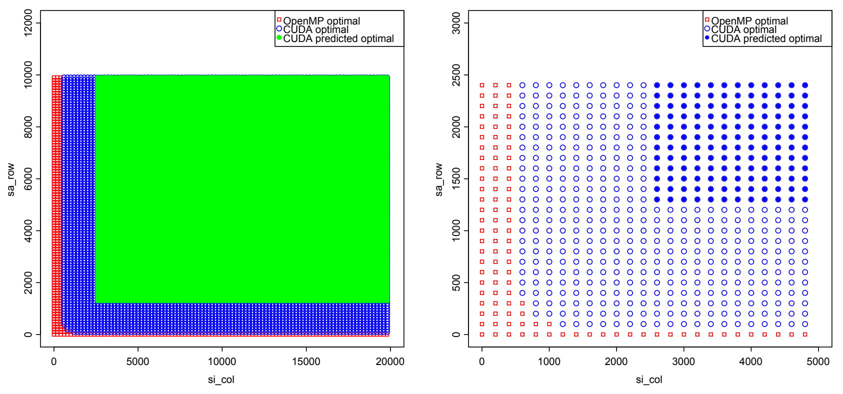


Figure 4.6: Test result of 4 benchmarks for test cases from only closed space (blue/left bar: accuracy, red/right bar: (number of elements in closed space)/(number of elements in whole space), unit: %, BP: Backpropagation, MM: Matrix multiplication, PF: Path finder, ST: Sorting)

optimal one is the OpenMP-predicted-optimal region, and in this case the CUDA-predicted-optimal region is a subset of the CUDA-optimal region. From Figure 4.7a, we can see that even with such a shallow depth of training (and thus relatively low overhead in training time), the predicted boundary is fairly close to the real boundary (with prediction accuracy 79.71%; as the training depth grows, the predicted boundary will converge to the real boundary as shown in the following experiments), and the convexity assumption holds statistically for this case.

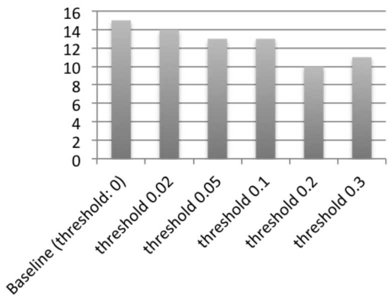
A closed space is trivial for prediction, while an open space is more complex and interesting to visualize. Since Euclidean distance is used as the criteria for prediction, we can see in Figure 4.7b that the predicted regions are the sub-squares of the subspace. The real boundary is nonlinear, though here we use a linear boundary for approximation, from which some of the prediction inaccuracies rooted. Usually, the inaccuracies happen near the real boundaries, and the performance difference in those points near the boundaries are often relatively small, thus such inaccuracies will not lead to significant performance loss. As the training depth grows, more training efforts are used only in those open regions, and the inaccuracies will decrease efficiently.



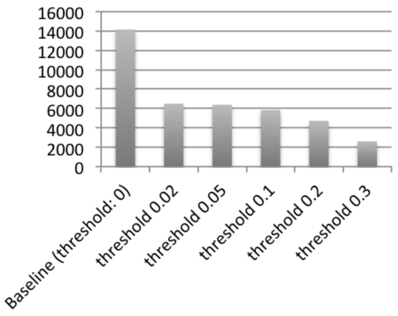
(a) The whole landscape (b) One specific open space

Figure 4.7: Overlay of real and predicted landscape for PF

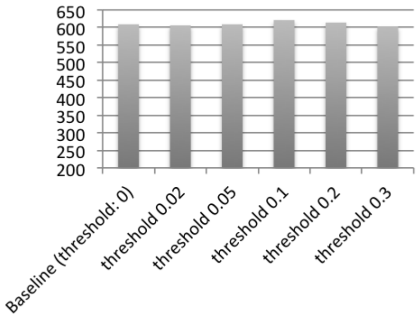
Discussion: Thresholding Thresholding is designed to reduce training time by closing spaces earlier if the user-defined threshold is met. Figure 4.8 shows the comparison of the training time for thresholding setting 0 (disable thresholding), 0.02, 0.05, 0.1, 0.2 and 0.3. We can see that for most cases the training time decreases as we enlarge the value of the threshold. However whether thresholding can help or not to reduce the training time, or how much it can reduce the training time, depends on the implementa-



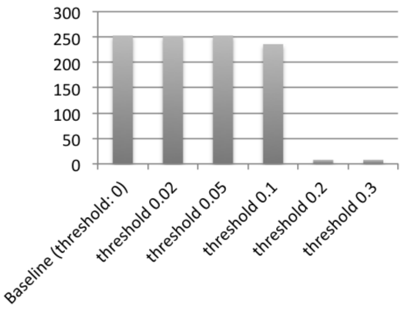
BP: Depth 4



MM: Depth 4

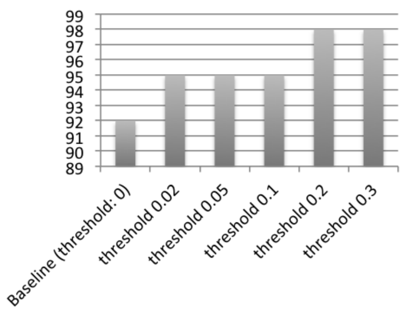


PF: Depth 1

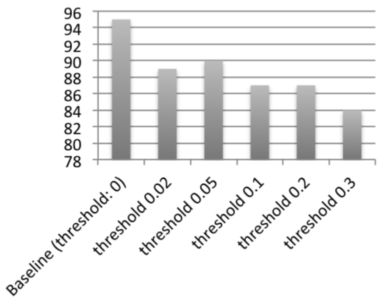


ST: Depth 4

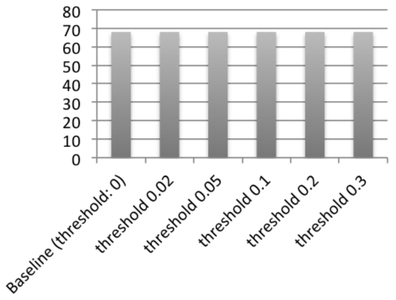
Figure 4.8: Training time (s) for thresholding with θ =0 (disable thresholding), 0.02, 0.05, 0.1, 0.2 and 0.3



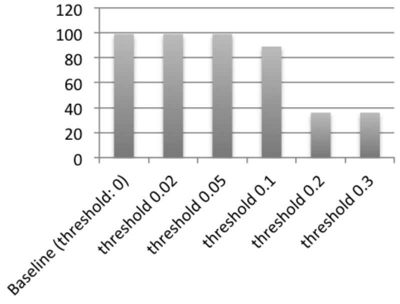
BP: Depth 4



MM: Depth 4



PF: Depth 1



ST: Depth 4

Figure 4.9: Prediction accuracy (%) for thresholding with θ =0 (disable thresholding), 0.02, 0.05, 0.1, 0.2 and 0.3

tion variants' performance curves, thus different benchmarks show different behaviors. For the Matrix multiplication benchmark, the effect is rather obvious, but for the Path finder benchmark, the effect of thresholding is trivial.

Figure 4.9 shows the comparison of prediction accuracy for thresholding setting 0, 0.02, 0.05, 0.1, 0.2 and 0.3. For most cases, as we enlarge the value of the threshold, the prediction accuracy drops, but only by a few percent. [3] For Backpropagation, as we enlarge the value of threshold, the prediction increases. This is because when a larger threshold is specified, less training examples are sampled, and a simpler prediction model or hypothesis is generated, thus sometimes the overfitting problem is avoided. Setting a large threshold may risk generating too few training examples; although a large amount of training time is saved, the underfitting problem happens and we observe a deep decrease in prediction accuracy as shown in the Sorting

[3]For the benchmark MM and the 30% threshold, 84% accuracy is achieved, which means that in the worst case 16% of the measurements are at 70% "efficiency" (performance relative to the true winner had tolerance been zero), and this responds to 95.2% average net efficiency. For the 2% and 5% tolerance, the efficiencies are about 99.8% and 99.5%, respectively.

benchmark.

Setting an appropriate threshold value reflects the trade-off between the training overhead and the prediction accuracy. It may be possible to automatically learn a reasonable threshold.

Discussion: Light oversampling Light oversampling is designed to improve the prediction accuracy by adding more training examples (sampling extra points in subspaces besides their vertices). Figures 4.10 and 4.11 show prediction accuracy and training time with and without light oversampling. The figures show that the accuracy for each benchmark improves with light oversampling, at the cost of increased training time.

As the training depth increases, the accuracy increase rate decreases. When the training depth is low, only a limited number of training samples are available, thus if some significant training samples are added, the prediction accuracy will improve significantly. When more and more training samples are added, the accuracy converges to the upper bound (100%), thus it is difficult to find training samples that can contribute to improving the prediction model, and adding new training samples may not make much sense.

Meanwhile, if we increase the training depth, the number of subspaces added increases exponentially due to the subspace tree structure's characteristic, thus the increase rate of training time change is increasing. (Note that Figure 4.10 for Matrix multiplication and Path finder uses a logarithmic-scale axis, thus the increase amounts are bigger than what the bar chart looks like.)

Thus, it is worthwhile to use this technique for lower training depth, which will lead to significant accuracy increase with low extra training time.

Note that in Figure 4.10, for the benchmark Back propagation, the prediction accuracy decreases as the training depth increases (more training samples are added). This is a well-known machine learning problem, *overfitting*, which means that the prediction model (the hypothesis) tries to fit the training set in a very complex way, and may not generalize well to new test cases. This problem is interesting for future investigation.

Discussion: Implementation pruning The implementation pruning technique is designed to reduce training time when the number of implementation variants is large for a component. Figure 4.12 compares the training time with and without implementation pruning. From the bar charts we can see that for the benchmarks Matrix multiplication and Sorting, which contain a relatively large number of implementation variants, the training time decreases significantly (note that the bar charts of Matrix multiplication and Sorting use a logarithmic scale, thus the time decrease is much larger than it looks like from the charts), especially for the case of depth 4 in Matrix multiplication, the training time is reduced by more than 39 times. For depth 0 of all benchmarks, no training time is reduced because before the

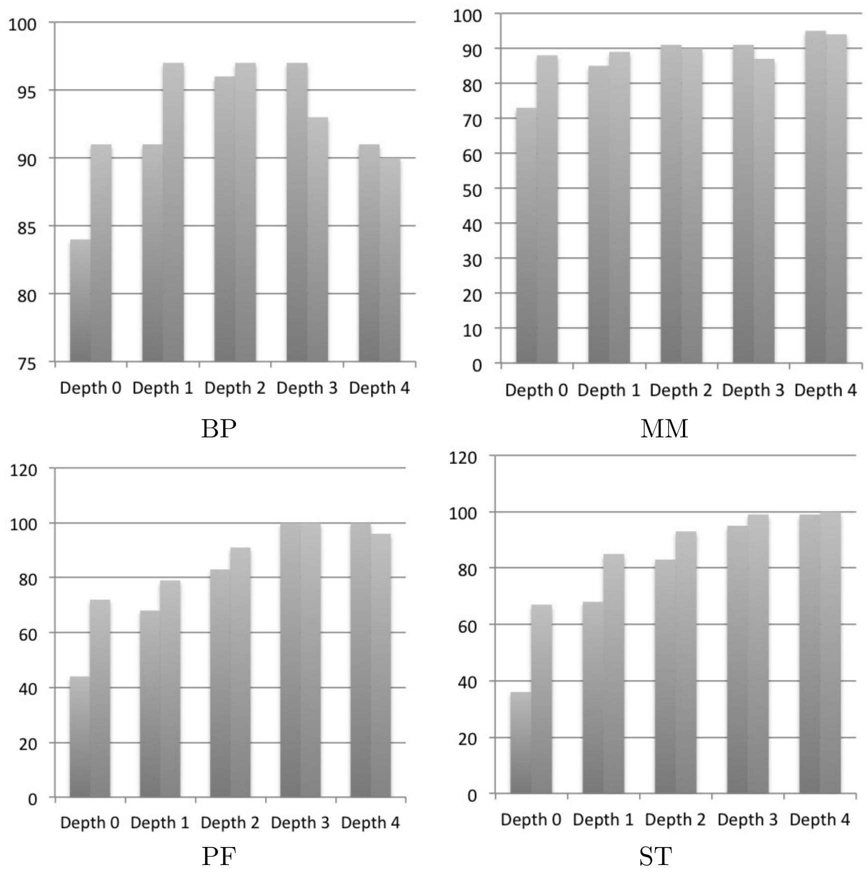


Figure 4.10: Prediction accuracy (%) for with (red/right bar) and without (blue/left bar) light oversampling

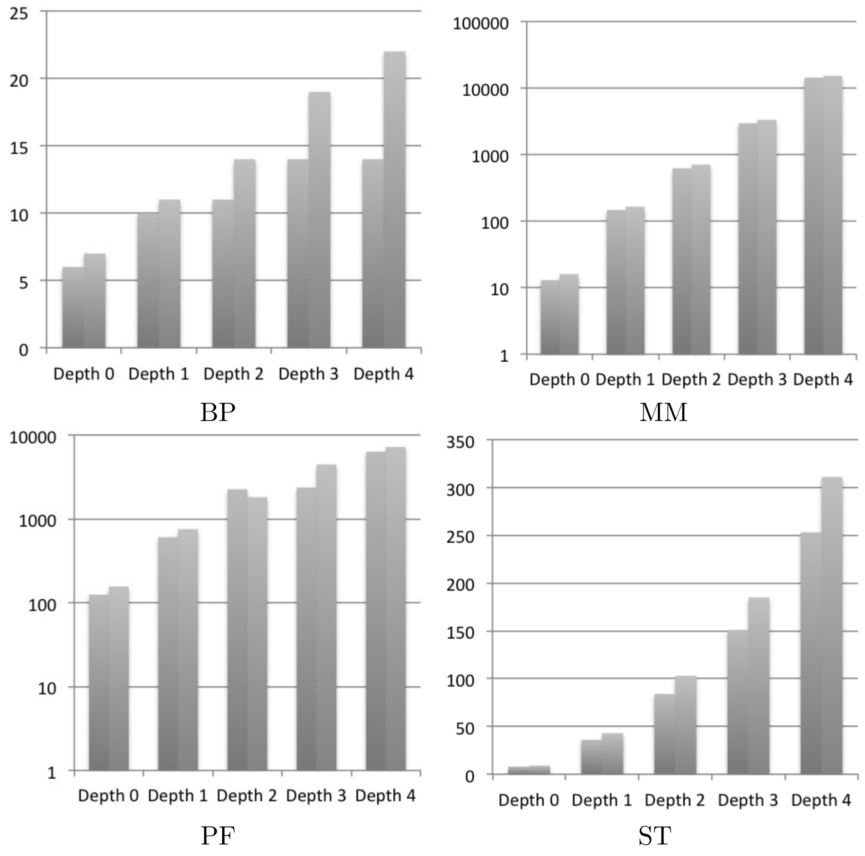


Figure 4.11: Training time (s) with (red/right bar) and without (blue/left bar) light oversampling

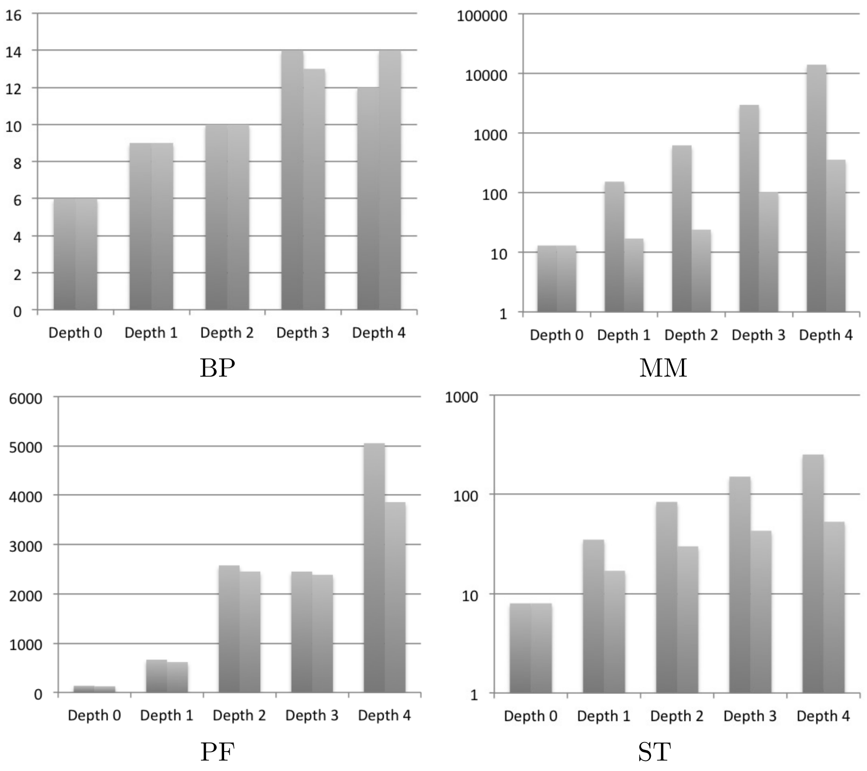


Figure 4.12: Training time (s) with (red/right bar) and without (blue/left bar) implementation pruning

first division step the sampling happens as before, and the implementation pruning is applied afterwards. For benchmarks Back propagation and Path finder, the effect is rather marginal because the number of implementation variants is small. Thus it may be more worthwhile to apply this technique when the number of implementation variants is relatively large.

When applying implementation pruning no noticeable drops on the prediction accuracy are observed; thus for all benchmarks, it is safe to perform implementation pruning.

Discussion: Combining Light oversampling and Implementation pruning Different pruning techniques are designed for different optimization objectives and have been shown with their effects in the experimental results. Light oversampling is designed for prediction accuracy, while thresholding and implementation pruning are for less training or sampling overhead. One idea is; although light oversampling enables higher prediction accuracy, it causes more overhead, can we reduce the overhead by thresholding or implementation pruning or both?

Figures 4.13 and 4.14 show the effect (accuracy and training time) of combining light oversampling with implementation pruning. For the benchmarks with a relatively large number of implementation variants, we see that reduced overhead occurs together with improved prediction accuracy. This is because the overhead caused by light oversampling is much less than the savings by implementation pruning. On one hand, implementation pruning can mitigate the overhead increase by the light oversampling; on the other hand, implementation pruning may lead to lose of optimization potential. This problem can be mitigated by light oversampling for the reason that sampling extra points may find more winners than those obtained from sampling the vertices.

Other combinations are also interesting to us and can be explored in the future.

Discussion: Comparing adaptive sampling with random sampling In order to evaluate further on the pros and cons of adaptive sampling, we compare the training time and accuracy between adaptive sampling and random sampling [58]. We feed the training examples generated by random sampling and adaptive sampling to the same predictor: C5.0 [17]. For benchmarks Matrix multiplication and Sorting which have more than two implementation variants, we test also for adaptive sampling with implementation pruning. Figure 4.15 shows the results of the comparison for 4 benchmarks.

From the charts we can see that the trend that the accuracy increases with increasing training time (the number of training examples). For Matrix Multiplication, it is quite clear to notice that as the training time increases, the accuracy quickly converges towards 100% in adaptive sampling; however, for random sampling, the accuracy only increases marginally because the

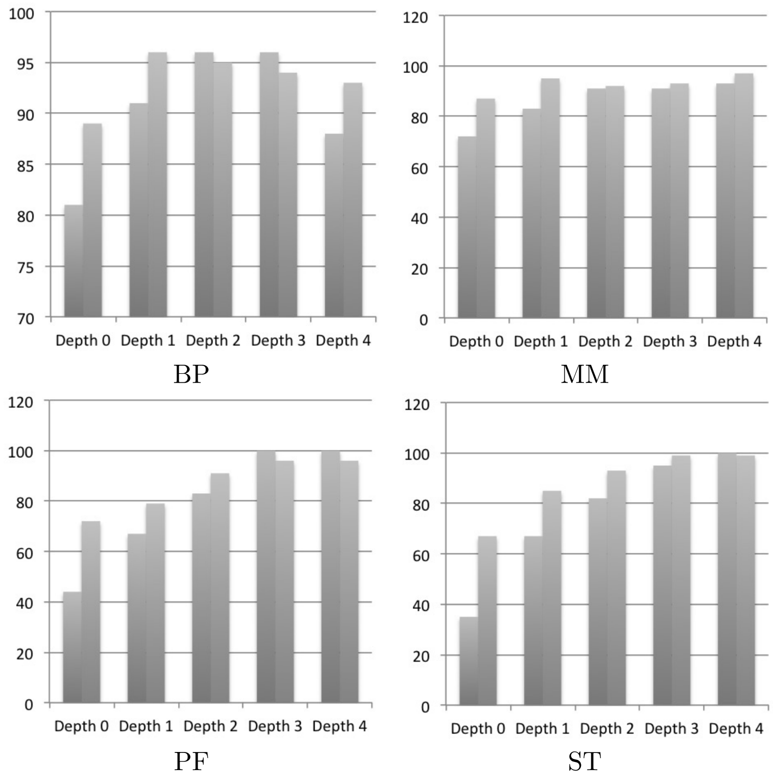
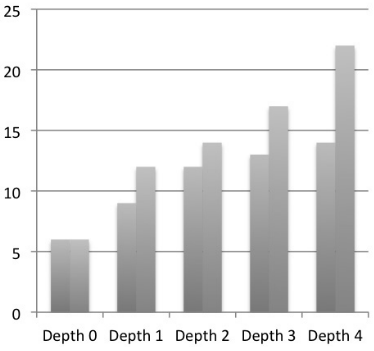
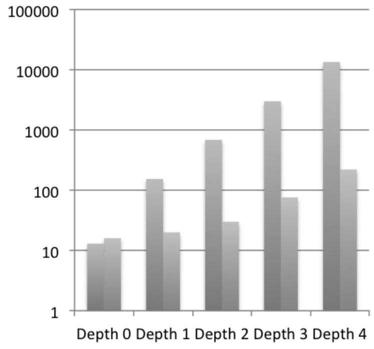


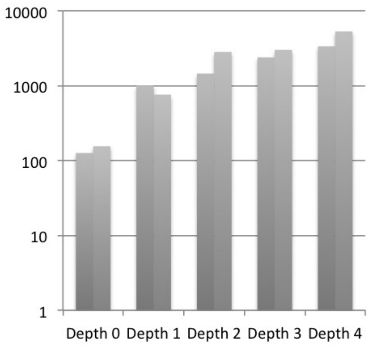
Figure 4.13: Prediction accuracy (%) with (red/right bar) and without (blue/left bar) the combination of light oversampling and implementation pruning



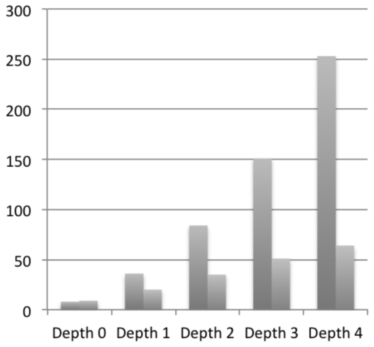
BP



MM



PF



ST

Figure 4.14: Training time (s) with (red/right bar) and without (blue/left bar) the combination of light oversampling and implementation pruning

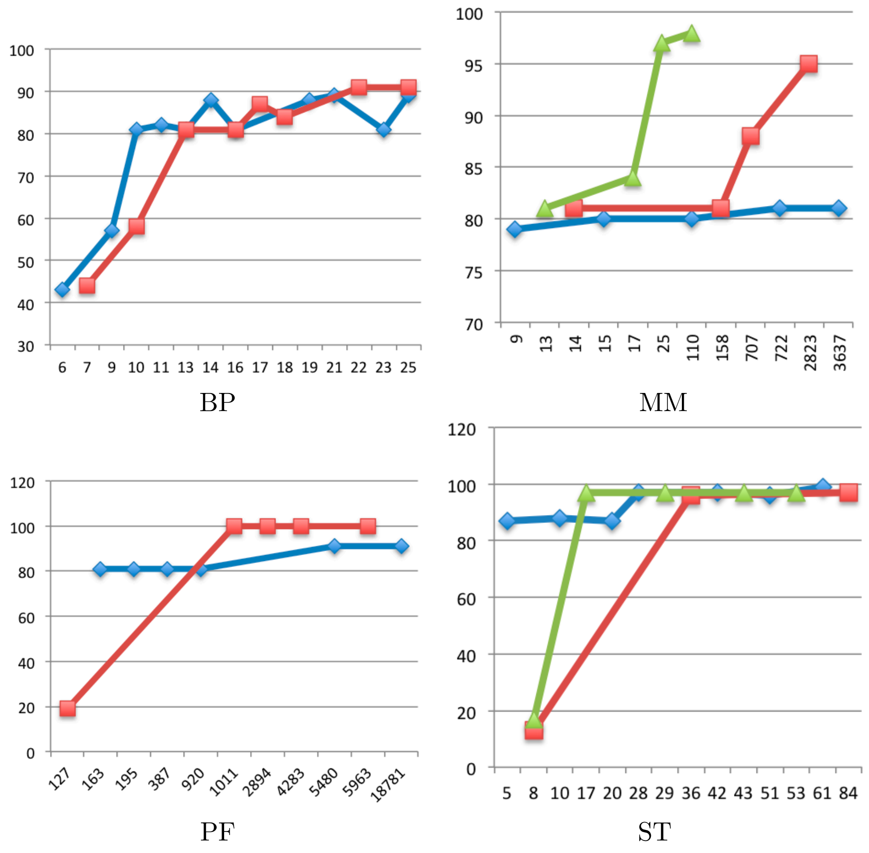


Figure 4.15: Comparison of adaptive sampling with random sampling (blue line: random sampling, red line: adaptive sampling, green line: adaptive sampling with implementation pruning; vertical axis: accuracy; unit:%. horizontal axis: training time, unit: s.)

random sampling often samples more points in the closed spaces which can not contribute to the prediction model significantly. With implementation pruning, the convergence speed is even much faster. For Path Finder, only in the first subdivision step, the adaptive sampling yields worse results, because it only samples points on the vertices of the whole context property value space. However, in quite few subdivision steps, the accuracy converges to 100%. For Sorting, adaptive sampling with implementation pruning shows similar advantage over random sampling. Only for Back propagation in which sampling executions are computationally cheap and the number of implementation variants is small, the two methods give similar results.

4.3.3 Summary

We have provided thresholding, light oversampling, and implementation pruning techniques that allow to better leverage our heuristic convexity assumption for pruning the adaptive sampling process. The combination of the different pruning techniques shows that the benefits of reducing training time and improving prediction accuracy can even be obtained at the same time. The comparison between adaptive sampling and random sampling shows that our method can converge much faster and reaches a prediction accuracy which random sampling can not reach over a sufficiently long time.

4.4 TunerPU Framework Design

In order to better support the implementation variant selection method described in the previous two sections, and also enforce component writers to provide the extra necessary ingredients (see details in Section 4.4.2) besides the components for the selection method, we propose a framework design and implementation named TunerPU, which we will discuss in this section.

Different computations or software components expose different interfaces, and part of the parameters in their interfaces are performance-related and worth tuning. How to view different components uniformly so that they can be tuned uniformly? We propose the TunerPU tuning framework, which consists of two main components: an abstraction layer and a generic API. The abstraction layer allows to unify the view of different software component (implementation variants) interfaces from the performance-tuning's angle. The generic API enables the uniform tuning of an generic implementation variant selector based on this uniform view.

Section 4.4.1 gives a brief overview of TunerPU's main components, which are detailed in the following sections. Section 4.4.2 describes the design of the abstraction layer. Section 4.4.3 describes the generic API. Section 4.4.4 gives several different examples to show the expressiveness of the TunerPU framework.

4.4.1 TunerPU Overview

TunerPU includes three main components as shown in Figure 4.16:

- `tuneit`: the main struct that programmers use to train its implementation variant selector, get predictions which implementation variant is best given a run-time context, and run an Implementation variant using a predicted implementation variant. More details is described in Section 4.4.3.

- `tunable`: the abstraction layer that unifies the view of different implementation variants implementing different interfaces, different concrete kernels could be used uniformly by tuneit if such an abstraction

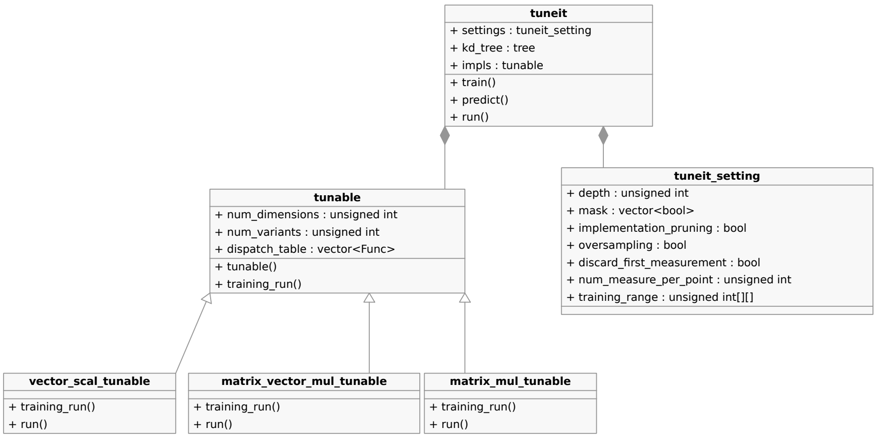


Figure 4.16: TunerPU class diagram

layer is implemented. In Figure 4.16, three kernels inherit tunable: vector scale, matrix-vector multiplication and matrix-matrix multiplication. This component is detailed in Section 4.4.2.

- `tuneit_setting`: provide some control on the tuning process, such as training depth (see Section 4.2.2.1), training range etc.

4.4.2 Unifying Views of a Set of Implementation Variants

Different objects can be indistinguishable if we neglect some aspects of the objects, which we usually achieve by abstraction. In other words, abstraction allows a unified view on different objects.

Different implementation variants that are functionally equivalent usually share the same interface or can be converted to share the same interface by wrappers. Remember such a group of implementation variants is a *ceset*. Different *ceset*s are usually not functionally equivalent and does not share the same interface. If we could not unify the view on different *ceset*s, then we have to develop specific tuning for each set. Thus, unifying the view has the great benefit to allow a uniform tuning on different *ceset*s.

The unified view is achieved by an abstraction design, which is an abstraction class `tunable` implemented in C++ shown in Figure 4.16. This class only has five elements: a constructor, a `run` method, and three member variables: `num_dimensions`, `num_variants` and `dispatch_table` Different *ceset*s now have only these five elements visible uniformly. The meanings of the 5 elements are described below:

- `constructor`: initializes a `tunable` class with appropriate configurations.

- `training_run()`: initializes the run-time environment (e.g., input arrays) for all the implementation variants in a *ceset*, runs all its implementation variants with measurement, checks the correctness of the returned results by invoking each implementation variant, and destructs the run-time environment. Its input is a `variant_mask` (showing if some variants are disabled for invocation), the repeat size of runs per each setting, and the specified run-time performance-related (not the complete list of) arguments. Its output is a vector of tuples, each mainly describing the run-time performance-related arguments and the corresponding measurement value. It is a pure virtual method and an override is enforced by component writer. The method is the key part of the abstraction class. Each *ceset* requires different initialization logic[4], different measurement methods[5], and different logic to check correctness[6]. Thus by the `training_run()` method, all those differences are unified and made invisible. Implementing the `training_run()` method requires domain knowledge (e.g., initialization, correctness checking and selecting performance-related parameters), thus component writers have the privileged position to write this method for tuning, which is why such an override is enforced by pure virtual syntax. Since the `training_run()` method uses variadic templates, it is expressive for the arbitrary number of performance-related variables.
- `num_dimensions`: number of performance-related arguments.
- `num_variants`: number of implementation variants.
- `dispatch_table`: array of function pointers to the implementation variants.

Thus, for each *ceset*, component writers only need to define a class that inherits our `tunable` class, then all the implementation variants inside the equivalent set are amenable to the tuning by our implementation selector which is described next.

4.4.3 Generic Run-time Selector

Next we introduce our tuning facility, which aims to provide a tuning mechanism and can be seamlessly applied to different *ceset*s. It consists of two components: a setting and a tuner. The setting consists of a few knobs to control the tuning, such as the training range specifying the range for each performance related parameter a training phase might use. We can see that

[4]The initialization logic usually requires domain knowledge, e.g. fluid dynamic simulation.

[5]E.g. `clock_gettime()` for CPU (Linux) implementation variants and `cudaEventRecord()` for CUDA ones, MeterPU hides this complexity, see Chapter 3.

[6]Checking correctness usually requires domain knowledge as well.

our tuner class (called `tuneit` in Figure 4.16) integrates three constructs: a `tuneit_setting`, a `tunable` (the unified-view class of Section 4.4.2), and a `kd_tree` (which readers can neglect for now, it is detailed in Section 4.2). Then the `train()` method can perform the training and the `predict()` method can give a prediction of the expected best performing implementation variant according to the run-time context passed as arguments. The prediction result is the index number in the dispatch table of its `tunable` member variable. The connection between the tuner and the unified view class is in the `train()` method, where the `training_run()` method of its `tunable` member variable is called to get training data for arbitrary context within the range specified by the `setting` of the `tuneit`.

4.4.4 Expressiveness

We demonstrate the expressiveness of our tuning framework by several examples, a 1D (only one performance-related parameter in a function signature) component and a 2D component. A 3D component is shown in Chapter 7. The framework uses variadic templates to handle the cardinality of performance-related parameters of a component, so it can accommodate arbitrary such cardinality.

Listing 4.1 shows the unified view of a vector scale component, which simply scales a vector by a scalar. It is obvious that the performance of this component is determined only by its vector size assuming it is already compiled, thus it exhibits a 1D performance tuning problem. Two implementation variants are considered: `vector_scale_cpu` and `vector_scale_gpu` (a wrapper to its kernel function). In Listing 4.1, we pass to the tunable struct two function pointers to the two implementation variants and implement the `training_run()` method.

Listing 4.2 shows the example code for training an implementation selector and getting a prediction for an array size. First we initialize a `tuneit_setting` with appropriate control arguments. Then we initialize a tuner with the setting object and the `tunable` type in Listing 4.1. Afterwards we can invoke the training and get a prediction easily.

Listings 4.3 and 4.4 show two similar pieces of code for matrix vector multiplication as a 2D tuning problem, which only differs in the number of parameters in `training_run()` and `predict()` method and training settings. This shows that the unified view does help to reuse the tuning facility.

```
void vector_scale_cpu(float *x, size_t N, float val);
void vector_scale_gpu(float *x, size_t N, float val);

typedef void (*vector_scale)(float *, size_t , float);

template <class MeasureType, class ...Tunable_Args>
struct vector_scale_tunable :
       public tunable<MeasureType, vector_scale , Tunable_Args...>
       {
             vector_scale_tunable():
              tunable<MeasureType, vector_scale , Tunable_Args...>
                     (
                       //number of dimension, 1 in this case
                       VECTOR_SCALE_NUM_DIM,
                       //number of variants,
                       //calculated automatically on XPDL model
                       VECTOR_SCALE_NUM_VARIANTS,
                       {vector_scale_cpu , vector_scale_gpu}
                     )
                     {}

             std::vector< ... > training_run(
                 //mask out some implementation variants,
                 //passed automatically from tuneit_setting
                 std::vector<bool> const& variant_mask,
                 //number of runs per each setting
                 size_t const repeat_size,
                 //the performance-related parameter,
                 //passed automatically by the tuner
                 size_t const arg1 ) const
                 {...}
       };
```

Listing 4.1: Tunable struct for vector scale component

```
size_t const depth=2;
vector<bool> const mask(4,true);

tuneit::tuneit_setting<VECTOR_SCALE_NUM_DIM> st
     {depth, mask, true, false, true, 40, { {1,200} } };

constexpr size_t num_vertices=2;

tuneit::tuneit<
                  VECTOR_SCALE_NUM_VARIANTS,
                  num_vertices,
                  vector_scale_tunable<float, size_t>,
                  float, size_t
                > mytuner(st);

mytuner.train();

cout<<"prediction is: "<<mytuner.predict(20)<<endl;;
```

Listing 4.2: Code for implementation variant selection for vector scale component

```
void matrix_vector_mul_cpu(float const * A, float const *b, float *c
    , size_t const NY,size_t const NX);
void matrix_vector_mul_gpu(float const * a,float const * b,float * c
    ,size_t const ha,size_t const wa);

typedef void (*matrix_vector_mul)(float const * ,float const * ,
    float * ,size_t const ,size_t const);

template <class MeasureType, class ...Tunable_Args>
struct matrix_vector_mul_tunable :
      public tunable<
                      MeasureType,
                      matrix_vector_mul,
                      Tunable_Args...
                    >
{
            matrix_vector_mul_tunable():
             tunable<
                            MeasureType,
                            matrix_vector_mul,
                            Tunable_Args...
                          >
                    (
                      MATRIX_VECTOR_MUL_NUM_DIM,
                      MATRIX_VECTOR_MUL_NUM_VARIANTS,
                      {matrix_vector_mul_cpu,
                       matrix_vector_mul_gpu}
                    ){}

            std::vector<...> training_run(
                                 std::vector<bool> const&
                                 variant_mask,
                                 size_t const repeat_size,
                                 //performance-related parameters
                                 //passed automatically by tuneit
                                 size_t const HA,
                                 size_t const WA) const
                                 {...}
};
```

Listing 4.3: Tunable struct for matrix vector multiplication component

```
size_t const depth=2;
vector<bool> const mask(2,true);

tuneit::tuneit_setting<MATRIX_VECTOR_MUL_NUM_DIM> st
    {depth, mask, true, false, true, 40, { {1,200}, {1,200} } };

constexpr size_t num_vertices=4;

tuneit::tuneit<
               MATRIX_VECTOR_MUL_NUM_VARIANTS,
               num_vertices,
               matrix_vector_mul_tunable<float, size_t, size_t>,
               float,
               size_t,
               size_t > mytuner(st);

mytuner.train();

cout<<"prediction is: "<<mytuner.predict(20,20)<<endl;;
```

Listing 4.4: Code for implementation variant selection for vector scale component

4.4.5 Summary

In this section we have presented our tuning framework TunerPU, which consists of an abstraction class enabling a unified view on a set of implementation variants, and a tuner which could tune the implementation selection on the unified view thus decouples with specific set of variants. We also showed the expressiveness of our tuning framework with respect to the number of performance-related parameters.

4.5 Related Work

4.5.1 Automated Performance Autotuning

Techniques for automated performance tuning have been considered extensively in previous work; they are applied e.g. in generators of optimized domain-specific libraries (such as basic linear algebra [127, 86, 122], reduction [130], sorting [87, 122] or signal transforms [43, 113, 106, 33]), iterative compilation frameworks (e.g. [103]), or for the optimized composition of general program units [72, 70, 7, 5, 126], e.g. the components in our case.

Automated performance tuning usually involves three fundamental preparatory tasks: (1) searching through the space of context property values, (2) generation of training data and measurements on the target system, (3) learning a decision function / rule (e.g. for best variant selection, decomposition, or settings for tunable parameters), or alternatively (3a) learning a predictor for performance and then (3b) decide / optimize based on that predictor among the remaining options. In our approach, these three tasks are tightly coupled to limit the amount of measurement time and representation size required, while most other approaches decouple at least two of these tasks.

Search, measurements and learning can each be performed off-line (i.e., at deployment time or compile time) or on-line (i.e., at run time), or as a combination of both. In our approach, all tasks are done off-line at component deployment time, while all are performed at runtime in the StarPU runtime system by continuously recording measurements from the running program and using these data for future decisions [7].

Kessler and Löwe [72] propose a methodology for optimized composition of multi-variant grey-box components. The component provider offers additional knowledge such as time functions for performance prediction, which might include data obtained from microbenchmarking, measuring, direct prediction or hybrid prediction. Predictions are made for a regularly sampled (dense) space of context instances, including composition of prediction functions for recursive components in a dynamic programming algorithm. Based on those predictions, a dispatch table and dynamic selection code are generated and injected into the components for run-time selection. The dispatch tables can be a-posteriori compressed using various machine learning techniques such as decision tree, decision graph, Bayesian classifier and SVM, where the decision tree was empirially found to be most effective [26]. In contrast, our current work does the compression a-priori, thus avoiding excessive prediction or measurements. Danylenko et al. [26] compares 4 different machine-learning approaches, Decision Trees, Decision Diagrams, Naive Bayes and SVM on sorting benchmark in the field of context-aware composition for a-posteriori compression of the dispatch function. Results show that Decision Diagram performs better in scalability, and almost the same in prediction accuracy and decision overhead compared with the other three approaches.

PetaBricks [5] provides a framework with language and compiler support for exposing implementation variant choices. It also contains an off-line autotuning system which starts to test with a small input size and doubles the size of the input on each later iteration. They assume that optimal choices for smaller subproblems are independent of the larger problem, so they construct new composition candidates incrementally from small input sizes to larger ones. The algorithmic choices are made off-line in the output of the compiler.

Elastic computing [126] lets the programmer transparently utilize the heterogeneous computing resources by providing a library of elastic functions. The autotuner trains itself from measurements (which are not further specified) and then uses a linear regression model for predicting performance of untested input values.

Grewe and O'Boyle [48] suggested an approach for statically choosing the best mapping between tasks and unit types (CPU, GPU). Static features such as numbers of float operations, are extracted from a set of programs, and and scheduling decisions are fed to a SVM classifer. Then at compile time, the decision for distribution of work load on different kinds of processors is made.

ABCLibScript [61] is a directive system that provides autotuning functionality on numerical computations within the FIBER framework. The choice of performance-related parameters, such as unrolling depth and block length, is specified for training execution. A performance model is also specified by the users, and generated together with training results. At run-time, best code regions are selected.

Singer and Veloso [112] applied a back-propagation neural network for performance prediction in the field of signal processing. Results show that choices of different combinations of features affect remarkably the prediction accuracy.

Thomson et al. [123] presents an unsupervised learning approach (fuzzy clustering algorithm) for a machine learning based compiler. Significant reduction in the training cost is achieved by grouping training programs into clusters using the ratio of assembly instructions to the total program instructions' as a feature vector. After clustering, they carried out training executions on one (randomly selected) representative from each cluster, recording the best execution configuration for each of the selected programs. This is an alternative approach to reduce training time.

Wang and O'Boyle [125] developed two predictor functions (data-sensitive and data-insensitive) to predict the best OpenMP execution configuration (number of OpenMP threads, scheduling policy) for an OpenMP program on a given architecture. They use two machine learning algorithms (Artificial Neural Network and Support Vector Machine) and train them using code, data and runtime features extracted via source to source instrumentation.

Zuluaga et al. [131] provide a smart sampling strategy for multi-objective optimization problems. The sampling process starts with extracting random training samples to 1% of the training space. Then smart sampling coupled with a specific prediction method, Gaussian Process, will choose the next sampling point according to the sampled data so far. Since Gaussian Process is a statistical model and involves uncertainty, they try to iteratively sample the points with the largest uncertainty to improve the prediction accuracy.

Our approach can be considered as an adaptive variant of decision tree learning. Decision tree learning, often based on C4.5 [107] or similar tools, is also used in many other approaches, e.g. in [113, 122, 130, 33].

Analytical models have been researched for many years; [42], [52] and [53] show analytical models that can predict performance of applications with reasonable accuracy, but either they are coupled with specific applications or platforms, or their applicability for a broader range of applications is not shown.

4.5.2 Framework Design

Two main approaches are used in previous work addressing the framework design for implementation selection: the first approach introduces new programming languages with explicit support for implementation selection and implement new compiler to support the new languages, such as Petabricks [5].

Another approach is to use a library for a programming model that supports implementation selection, and naturally this could allow reuse of the legacy implementation variants.

OpenTuner [6] proposes a general tuning framework targeting at different domains and using ensembles of search techniques collaboratively. It also requires users to define a `run` function to evaluate the fitness of a certain tunable configuration, which is similar to our approach. The difference lies in that our approach is focusing on run-time parameter tuning, thus in addition to the `run` function in OpenTuner which is equivalent to our `training_run` function in TunerPU, we also include interface in TunerPU to support run-time tuning for production runs. We present a clear picture to reuse the legacy code, which we call as *ceset*. Our facility is implemented in C++, comparing to the choice of Python in OpenTuner, our approach may incur less framework overhead.

Elastic computing [126] proposes an adapter to connect tuning with legacy code. The framework also has an input/output description for a software component, however it does not encapsulate or explicitly describe how to produce a valid input for performance modeling, compared to our approach. Furthermore it requires programmers to map all performance-related parameters to a single value, which is equivalent to manual feature engineering and usually considered difficult to perform by hand in deep learning research [47], and our approach does not require such a mapping from programmers.

StarPU [8] requires programmers to write explicit wrappers to their legacy code if function signatures of the legacy code do not conform to their unified function signature, which is usually the case, and programmers do have to write such wrappers frequently. Our approach only requires programmers to write a struct implementing our unified view once per *ceset*, instead of once per each component. SkePU [31] requires new code to be written in their skeleton framework and variants are automatically generated, thus hard to reuse legacy implementations.

4.6 Summary

In this chapter we showed the complexity of the CPU-GPU selection problem, or more precisely, the implementation variant selection problem. We proposed an adaptive sampling approach and further three pruning strategies to enhance the selection mechanism. We designed a framework that allows a uniform view and thus easy tuning of legacy code, and a full-phase mechanism for implementation variant selection.

Chapter Acknowledgements

This work was partly funded by EU FP7 projects PEPPHER (www.peppher.eu) and EXCESS (www.excess-project.eu) and by SeRC project OpCoReS. We also thank the Scientific Computing group at the University of Vienna, Austria, for letting us use their GPU server *Cora* for the measurements reported in this paper.

Chapter 5

Handling Platform Complexity

This chapter is based on the following paper and technical reports:

- Kessler, C., Li, L., Atalar, A., and Dobre, A. (2015a). XPDL: Extensible Platform Description Language to Support Energy Modeling and Optimization. In *2015 44th International Conference on Parallel Processing Workshops (ICPPW)*, pages 51–60. IEEE

  and EXCESS project deliverables:

- Kessler, C., Li, L., Dastgeer, U., Gidenstam, A., and Atalar, A. (2014b). D1.2 Initial specification of energy, platform and component modelling framework. Technical Report FP7-611183 D1.2, EU FP7 Project EXCESS

- Kessler, C., Li, L., Hansson, E., Ahlqvist, J., Thorarensen, S., and Yang, M.-J. (2015c). D1.4 First prototype of composition tool and multi-level energy and platform modeling framework. Technical Report FP7-611183 D1.4, EU FP7 Project EXCESS

- Kessler, C., Li, L., Melot, N., Hansson, E., Ernstsson, A., Thorarensen, S., and Barry, B. (2016). Final specification of energy, platform and component modelling framework and final prototype. Technical Report FP7-611183 D1.5, EU FP7 Project EXCESS

The next level of complexity that will inevitably come is the platform (hardware and software infrastructure) complexity. As utilizing platform efficiently is a necessary condition to achieve high performance, we can not treat the platform transparently as a black box, instead we must open that box. When we open this box, the complexity is high, as the platform is a heterogeneous set of hardware components, interconnects, and low-level software pieces. Each heterogeneous hardware component has different capabilities and requires different structures, each software piece has different versions, usually with the same interface. To achieve portability requires to understand if a piece of code has its dependencies (e.g., BLAS libraries) and constraints (e.g., enough GPU memory) satisfied by a target platform. To achieve performance portability requires to understand the hardware capabilities (e.g., is it beneficial to use GPUs?), and the performance of dependent libraries (e.g., can a library run efficiently on a machine at hand?).

Can we model the platform complexity by a unified language? How to model enough details to support interesting optimizations at higher level? Can we achieve a design of a platform description language that benefits from modern language features such as modularity? What is the run-time overhead for querying platform models? In this chapter we will explore these questions.

Section 5.1 discusses the platform complexity which motivates and helps understanding the design requirements of a platform description language. Section 5.2 reviews the platform description language PDL, and discusses its pros and cons that leads to a redesign of the language called XPDL in Section 5.3, embedded with some examples which illustrate the expressiveness of our language. Section 5.4 describes our compiler support for implementating the XDPL language and the interfacing for high level optimizations.

5.1 Platform Complexity

We refer to *platform* as the collection of hardware components and system software that are relevant for portability and performance portability for a computation task to run on. For a heterogeneous system, the platform usually consists of CPUs (either single core or, most common, multicore), GPUs and/or other accelerators, DRAM and interconnects between those hardware components. It also includes the system software such as the operating systems, libraries etc.

Why is modeling such systems not trivial? First of all, different systems usually have different configurations. Whether accelerators such as GPUs may be equipped or not depends on the kind of tasks a system is supposed to perform, as accelerators are more specifically designed than CPUs. GPUs are advantageous in data-parallel applications. FPGAs and ASICs are even more application-specific, although they are out of the scope of this thesis. Being designed for special purpose instead of general purpose allows accelerators to have different capabilities than general purpose processors. More diversity makes platform modeling complex. Furthermore, hardware evolves fast, requiring that the modeling could be adapted accordingly.

Different hardware components usually require different system software packages to provide low level support on them, exposing different APIs, separate memory spaces, programming models etc, which greatly complicates the code that depends on them. Considering that the hardware evolves fast, the system software will change accordingly. For example, new CUDA versions are released fast, about two versions per year. Newer versions bring new features, such as Nvidia's Unified Memory added from CUDA 6.0. Different system software makes platform modeling complex.

The range of different classes of computers that we need to be able to model is wide, from mobile devices with mobile processors to HPC clusters. Those computers are designed with different instruction set architectures, hardware organizations, technologies etc., which may show different performance (e.g., bandwidth), and run different compilers, etc., that are related to performance portability.

Granularity of modeling is an issue as well: one could model the hardware very coarsely (only model the main components and interconnects), or very fine-grained (to the logic level). To determine the right level of granularity and allow easy customization for different optimizations of interest requires special care. Furthermore, the organization of roles for each hardware component may change over time. For a typical system with CPUs and GPUs, CPUs usually play the role of master which control GPUs as slaves. CPUs could themselves be slaves controlled by one of the CPUs. If threads running on CPUs are not pinned to them, then the roles can change dynamically during a parallel program execution. As energy consumption becomes a dominating factor to constrain computing performance, providing some support of energy modeling is within our concerns.

5.2 A Review of PEPPHER PDL

PDL, described by Sandrieser et al. [110], has been developed in the EU FP7 project PEPPHER (2010-2012, www.peppher.eu), as a XML-based platform description language for single-node heterogeneous systems, to be used by tools to guide performance prediction, automated selection of implementation variants of annotated application software components ("PEPPHER components"), or task mapping. In particular, it establishes naming conventions for the entities in a system such that they can be referred to by symbolic names. Beyond that, it is also intended to express so-called *platform patterns*, reusable templates for platform organization that could be instantiated to a complete platform specification.

PDL models the main architectural blocks and the hierarchical execution relationship in heterogeneous systems. All other entities (e.g., installed software) are modeled as free-form properties in the form of key-value pairs.

The *main architectural blocks*, which distinguish between *types* and *instances*, include
– *Processing Units* (PU), describing the processing elements in a heterogeneous machine;
– *Memory regions*, specifying data storage facilities (such as global/shared main memory, device memories); and
– *Interconnect.*

For PDL example specifications we refer to [110] or [66, Sec. 4.1].

5.2.1 Control Relation

The overall structure of these hardware components descriptions in a PDL specification is, however, not organized from a hardware perspective (i.e., the structural organization of the hardware components in the system) but follows the programmer perspective by formalizing the *control relation* between processing units as a logic tree consisting of so-called Master, Hybrid and Worker PU. There must be one *Master PU*, a feature-rich general purpose PU that marks a possible starting point for execution of a user program, thus acting as the root of the control hierarchy; there are a number of *Worker PU*s that are specialized processing units (such as GPUs) that cannot themselves launch computations on other PUs and thus act as leaves of the control hierarchy; and *Hybrid PU*s that can act both as master and worker PU and thus form inner nodes in the control hierarchy. A system might not contain all three types of PUs (i.e., control roles) simultaneously. For instance, a standard multicore server without accelerators has no workers, and the Cell/B.E., if used stand-alone (i.e., not coupled with a host computer acting as Master), has no hybrid PUs.

In practice, which PU is or could be host and which one is a device depends not only on the hardware but mainly on the programming model used, which might be defined by the compiler, runtime system and/or further system layers. The PDL control relation usually reflects the system's native low-level programming model used (such as CUDA).

The actual workings of the launching of computation modeled by the control relation cannot be explicitly specified in PDL. In practice, this aspect is typically hardcoded in the underlying operating system scheduler or the heterogeneous runtime system, making the specification of a unique, specific Master PU questionable, for instance in a dual-CPU server.

Discussion The motivation of emphasizing the control role in PDL was to be able to define abstract platform patterns that can be mapped to concrete PUs. However, we find that using the control relation as the overarching structure of a platform description is not a good idea, especially as the control relationship may not always be fixed or even accessible to the programmer but might, for instance, be managed and hardcoded in the underlying heterogeneous runtime system.

In particular, it makes more sense to adopt a hardware-structural organization of platform descriptions, because power consumption, temperature metrics and other measurement values naturally can be attributed to coarse-grain hardware blocks, such as CPU, memory or devices, but not to software roles. Most often, the software roles are implicitly given by the hardware blocks and/or the underlying heterogeneous runtime system, hence a separate role specification is usually not necessary.

Hence, we advocate a language structure that follows the hardware composition of the main architectural blocks, that is also more aware of the concrete architectural block type (e.g., that some hardware component is actually a GPU), and that allows to optionally model control relations separately (referencing the involved hardware entities) for complex systems where the control relation cannot be inferred automatically from the hardware entities alone. This should still allow the definition of abstract platform (i.e., generic control hierarchy) patterns, but rather as a secondary aspect to a more architecture oriented structural specification. Where appropriate, the control relation could also be specified as a feature of the runtime system rather than the architectural structure itself.

In the PEPPHER software stack, the mechanisms and constraints of launching tasks to other PUs are managed entirely by the runtime system (StarPU [8]); as a consequence, the control relation information in PDL specifications was not needed and not used by the PEPPHER tools. If suitably complemented by control code specification, it might rather become relevant in scenarios of generating code that runs directly on the bare hardware without a managing runtime system layer in between.

5.2.2 Interconnect Specification

The interconnect specification in PDL is intended to model communication facilities between two or more PUs, in a style similar to xADML [11] intercluster communication specifiers. In PEPPHER this information was not used, as the communication to and from an accelerator is managed completely within the PEPPHER runtime system, i.e., StarPU.

5.2.3 Properties Concept

The possibility of specifying arbitrary, unstructured platform properties (e.g., installed software) as *key-value pairs*[1] provides a very flexible and convenient extension mechanism in PDL, as long as the user(s) keep(s) control over the spelling and interpretation of introduced property names. Properties can also be used as placeholders if the actual values are not known yet at meta-data definition time or can change dynamically, e.g. due to different possible system configurations. The existence and, where existing, values of specified properties can be looked up by a basic query language.
Discussion Properties allow to extend PDL specifications beyond the defined tags and attributes given by the fixed PDL language syntax, in an ad-hoc way. This is highly flexible but can also lead to inconsistencies and confusion due to lack of standardization of naming conventions for properties.

Properties in PDL can be mandatory or optional. We think that mandatory properties should better be modeled as predefined XML tags or attributes, to allow for static checking. For instance, a property within a

[1]Both keys and values are strings in PDL.

CPU descriptor such as `x86_MAX_CLOCK_FREQUENCY` [110] should better be specified as a predefined attribute.
Using Platform Descriptions for Conditional Composition In recent work on *conditional composition* [28] our group used PDL specifications to guide the dynamic selection of implementation variants of PEPPHER components. In the *PEPPHER composition tool* [32], which builds an adaptive executable program for a PEPPHER application from its annotated multi-variant components, the target system's PDL specification is parsed and stored in an internal representation, which can be inspected at composition time to check for selectability constraints that depend on static property values. In order to also enable constraints that involve dynamic properties or property values, the composition tool also writes the PDL representation to a file that is loaded by the PEPPHER application's executable code on startup into a run-time data structure. This data structure thus holds the platform metadata which the composition code can introspect via a C++ API, i.e. read access property values in the evaluation of constraints that guide selectability of implementation variants. In a case study with a sparse matrix vector multiply component in [28], our group used this feature to let each CPU and GPU implementation variant specify its specific constraints on availability of specific libraries (such as sparse BLAS libraries) in the target system, and to add selection constraints based on the density of nonzero elements, leading to an overall performance improvement.

5.2.4 Modularity Issues

Another limitation of PDL is its semi-modular structure of system specifications. While it is generally possible to decompose any XML specification into multiple submodules (files), PDL does not specifically encourage such modularity and tends to produce monolithic system descriptions, which limits the reuse of specifications of platform subcomponents.

5.3 XPDL Design Principles

In contrast to the PDL default scenario of a single monolithic platform descriptor file, XPDL provides a modular design by default that promotes reuse of submodels and thus avoids replication of contents with the resulting risk for inconsistencies. XPDL defines a hierarchy of submodels, organized in model libraries containing separate models for entities such as

- CPUs, cores, caches,
- Host memory,
- GPUs and other accelerators with own device memory,
- Interconnects, such as inter-node networks (e.g. Infiniband), intra-node networks (e.g. PCIe) and on-chip networks, and

- System software installed.

In principle, a XPDL descriptor is a machine-readable data sheet of the corresponding hardware or system software entity, containing information relevant for performance and energy optimization. The descriptors for the various types of CPUs, memory etc. are XPDL descriptor modules (`.xpdl` files) placed in a distributed model repository: XPDL models can be stored locally (retrieved via the model search path), but may, ideally, even be provided for download e.g. at hardware manufacturer web sites. Additionally, the model repository contains (source) code of microbenchmarks referenced from the descriptors.

In addition to the models describing hardware and system software components, there are (top-level) models for complete systems, which are (like their physical counterparts) composed from third-party provided hardware and software components, and reference the corresponding XPDL model descriptors by name to include these submodels.

Alternative Views of XPDL Models Generally, XPDL offers multiple views: XML (used in the examples in this paper), UML (see [66]), and C++ (as used for the internal and run-time representation of models). These views only differ in syntax but are semantically equivalent, and are (basically) convertible to each other. In the following we present the XML view.

5.3.1 Basic Features of XPDL

We distinguish between *meta-models* (classes describing model element *types*) and *concrete models* of the concrete hardware components (i.e., *instances*) in the modeled target system, as several concrete models can share the same meta-model. For example, several concrete CPUs in a computer system can share the same CPU type definition.

In principle, each XPDL model of a (reusable) hardware component shall be specified separately in its own descriptor file, with the extension `.xpdl`. For pragmatic reasons, it is sometimes more convenient to specify sub-component models in-line, which is still possible in XPDL. Reusing and referencing submodels is done by referencing them by a unique name[2], so they can easily be retrieved in the same model or in the XPDL model repository. Depending on whether one includes a submodel at meta-level or base-level, the naming and referencing use different attribute names. To specify an identifier of a model element, the attribute `name` is used for a meta-model, and the attribute `id` is for a model. The strings used as `name` and `id` should be unique across the XPDL repository for reference non-ambiguity, and naming is only necessary if there is a need to be referenced. The attribute `type` is used in both base-level and meta models for referencing to a specific meta-model.

Using the `extends` attribute, XPDL also supports (multiple) *inheritance* to encourage more structuring and increased reuse by defining attributes

[2]The name may contain a version number (if applies) to be unique.

and subcomponents common to a family of types (e.g., GPU types) in a supertype. The inheriting type may overscribe attribute values.

The XML element `group` can be used to group any elements. If the attribute `quantity` is used in a `group` element (as in Listing 5.1), then the group is implicitly homogeneous. In such cases, to facilitate identifier specification, attributes `prefix` and `quantity` can be used together to assign an `id` to the group member elements automatically. For example, if we specify a group with prefix `core` and quantity `4`, then the identifiers of the group members are assigned as `core0`, `core1`, `core2` and `core3`.

For a metric such as static_power, if specified as an attribute, its unit should also be specified, in *metric*`_unit` form such as `static_power_unit` for `static_power`. As an exception, the unit for the metric `size` is implicitly specified as `unit`.

Installed software is given by `installed` tags; also these refer to separate descriptors, located in the software directory.

The `<properties>` tag refers to other possibly relevant properties that are not modeled separately by own descriptors but specified flexibly by key-value pairs. This escape mechanism (which avoids having to create new tags for such properties and allows ad-hoc extensions of a specification) was also suggested in PDL [110].

Not all entities are modeled explicitly. For instance, the motherboard (which also contributes some base energy cost) may not be modeled explicitly; its static energy share will be derived and associated with the node when calculating static and dynamic energy costs for each (hardware) component in the system.

5.3.2 Parameters, Constants and Constraints

By the `param` attribute, models can define local parameters (variables) that can be set when instantiating a concrete model. Default values can be specified, too. An example can be seen in Listing 5.9.

Constants, declared using the `const` attribute, are parameters that cannot be reassigned an arbitrary new value on instantiation.

A constraint defines a boolean expression that involves C-style terms of parameters and constants, and that must be fulfilled for a model to be valid. A typical use is to describe configurable properties of an architecture or system software, as in the example in Listing 5.8. A model can define multiple `constraints`, collected in the `constraints` section.

5.3.3 Hardware Component Modeling

In the following we show examples of XPDL descriptions of hardware components. Note that these examples are simplified for brevity. More complete specifications for some of the EXCESS systems are given in [66].

Processor Core Modeling We consider a typical example of a quad-core CPU architecture where L1 cache is private, L3 is shared and L2 is shared by 2 cores. Listing 5.1 shows the corresponding XPDL model. The use of attribute `name` in the `cpu` element indicates that this specifies a name of this meta-model in XPDL, and the attribute `quantity` in the `group` element shows that the group is homogeneous, consisting of identical elements whose number is defined with the attribute `quantity`. Furthermore, the `prefix` can define a name prefix along with `quantity`, which automatically assigns identifiers to all group members that are built by concatenating the prefix string with each group member's unique rank ranging from 0 to the group size minus one.

The sharing of memory is given implicitly by the hierarchical scoping in XPDL. In the example, the L2 cache is in the same scope as a group of two cores, thus it is shared by those two cores.

```
<cpu name="Intel_Xeon_E5_2630L">
  <group prefix="core_group" quantity="2">
    <group prefix="core" quantity=2>
      <!--  Embedded definition -->
      <core frequency="2" frequency_unit="GHz" />
      <cache name="L1" size="32" unit="KiB" />
    </group>
    <cache name="L2" size="256" unit="KiB" />
  </group>
  <cache name="L3" size="15" unit="MiB" />
  <power_model type="power_model_E5_2630L" />
</cpu>
```

Listing 5.1: An example meta-model for a specific Xeon processor

Although the element `core` should rather be defined in a separate sub-metamodel (file) referenced from element `cpu` for better reuse, we choose here for illustration purposes to embed its definition into the CPU meta-model. Embedding subcomponent definitions gives us the flexibility to control the granularity of modeling, either coarse-grained or fine-grained. The same situation also applies to the cache hierarchy.

```
<!-- Descriptor file ShaveL2.xpdl -->
<cache name="ShaveL2" size="128" unit="KiB" sets="2"
       replacement="LRU" write_policy="copyback" />

<!-- Descriptor file DDR3_16G.xpdl -->
<memory name="DDR3_16G" type="DDR3"
  size="16" unit="GB"
  static_power="4" static_power_unit="W" />
```

Listing 5.2: Two example meta-models for memory modules

Memory Module Modeling Listing 5.2 shows example models for different memory components, in different files.

Interconnect Modeling The tag `interconnect` is used to denote different kinds of interconnect technologies, e.g. PCIe, QPI, Infiniband etc. Specifically, in the PCIe example of Listing 5.3, the channels for upload and download are modeled separately, since the energy and time cost for the two channels might be different [67].

```
<!-- Descriptor file pcie3.xpdl: -->
<interconnect name="pcie3">
 <channel name="up_link"
   max_bandwidth="6" max_bandwidth_unit="GiB/s"
   time_offset_per_message="?"
   time_offset_per_message_unit="ns"
   energy_per_byte="8" energy_per_byte_unit="pJ"
   energy_offset_per_message="?"
   energy_offset_per_message_unit="pJ" />
 <channel name="down_link" ... />
</interconnect>

<!-- Descriptor file spi1.xpdl: -->
<interconnect name="spi...">
  ...
</interconnect>
```

Listing 5.3: Example meta-models for some interconnection networks

Device Modeling Listing 5.4 shows a concrete model for a specific Myriad-equipped server (host PC with a Movidius MV153 development board containing a Myriad1 DSP processor), thus its name is specified with `id` instead of `name`. This Myriad server uses several interconnections to connect the host server to Myriad1, e.g. SPI and USB. For an instantiation of any kind of interconnect, the connection information must also be specified, e.g. using the `head` and `tail` attributes for directed communication links.

```
<system id="myriad_server">
  ...
  <socket>
   <cpu id="myriad_host" type="Xeon1"
        role="master"/>
  </socket>
  <device id="mv153board" type="Movidius_MV153" />
  <interconnects>
   <interconnect id="connect1" type="SPI"
      head="myriad_host" tail="mv153board" />
   <interconnect id="connect2" type="usb_2.0"
      head="myriad_host" tail="mv153board" />
   <interconnect id="connect3" type="hdmi"
      head="myriad_host" tail="mv153board" />
   <interconnect id="connect4" type="JTAG"
      head="myriad_host" tail="mv153board" />
  </interconnects>
  ...
</system>
```

Listing 5.4: A concrete model for a Myriad-equipped server

The device with the Myriad1 processor on it is a Movidius MV153 card, whose meta-model named `Movidius_MV153` is specified in Listing 5.5 and which is `type`-referenced from the Myriad server model.

```
<device name="Movidius_MV153">
  <socket>
    <cpu type="Movidius_Myriad1"
         frequency="180" frequency_unit="MHz" />
  </socket>
</device>
```

Listing 5.5: Example meta-model for Movidius MV153 board

The MV153 model in Listing 5.5 in turn refers to another meta-model named `Myriad1` which models the Myriad1 processor, see Listing 5.6.

```
<cpu name="Movidius_Myriad1">
 <core id="Leon" type="Sparc_V8" endian="BE" >
  <cache name="Leon_IC" size="4" unit="kB"
         sets="1" replacement="LRU" />
  <cache name="Leon_DC" size="4" unit="kB"
         sets="1" replacement="LRU"
         write_policy="writethrough" />
 </core>
 <group prefix="shave" quantity="8">
   <core type="Myriad1_Shave" endian="LE" />
     <cache name="Shave_DC" size="1" unit="kB"
            sets="1" replacement="LRU"
            write_policy="copyback" />
   </core>
 </group>
 <cache name="ShaveL2" size="128" unit="kB" sets="2"
        replacement="LRU" write_policy="copyback" />
 <memory name="Movidius_CMX" type="CMX"
         size="1" unit="MB" slices="8" endian="LE"/>
 <memory name="LRAM" type="SRAM"
         size="32" unit="kB" endian="BE" />
 <memory name="DDR" type="LPDDR"
         size="64" unit="MB" endian="LE" />
</cpu>
```

Listing 5.6: Example meta-model for Movidius Myriad1 CPU

```
<system id="liu_gpu_server">
  <socket>
    <cpu id="gpu_host" type="Intel_Xeon_E5_2630L"/>
  </socket>
  <device id="gpu1" type="Nvidia_K20c" />
  <interconnects>
    <interconnect id="connection1" type="pcie3"
                  head="gpu_host" tail="gpu1" />
  </interconnects>
</system>
```

Listing 5.7: A concrete model for a GPU server

GPU modeling is shown in Listings 5.7–5.10. The K20c GPU (Listing 5.9) inherits most of the descriptor contents from its supertype `Nvidia_Kepler` (Listing 5.8) representing a family of similar GPU types. The 64KB shared memory space in each shared-memory multiprocessor (SM) on Kepler GPUs can be partitioned among L1 cache and shared memory in three different configurations (16+48, 32+32, 48+16 KB). This configurability is modeled by defining constants (`const`), formal parameters (`param`) and constraints, see Listing 5.8. In contrast, a concrete K20c GPU instance as in Listing 5.10 uses one fixed configuration that overrides the generic scenario inherited from the metamodel. Note that some attributes of `K20c` are inherited from the Nvidia_Kepler supertype, while the K20c model sets some uninitialized parameters like global memory size (`gmsz`), and overwrites e.g.

the inherited default value of the attribute `compute_capability`.

```
<device name="Nvidia_Kepler" extends="Nvidia_GPU"
        role="worker">
 <compute_capability="3.0" />
 <const name="shmtotalsize"... size="64" unit="KB"/>
 <param name="L1size" configurable="true"
        type="msize" range="16, 32, 64" unit="KB"/>
 <param name="shmsize" configurable="true"
        type="msize" range="16, 32, 64" unit="KB"/>
 <param name="num_SM" type="integer"/>
 <param name="coresperSM" type="integer"/>
 <param name="cfrq" type="frequency" />
 <param name="gmsz" type="msize" />
 <constraints>
   <constraint expr=
        "L1size + shmsize == shmtotalsize" />
 </constraints>
 <group name="SMs" quantity="num_SM">
   <group name="SM">
     <group quantity="coresperSM">
       <core type="..." frequency="cfrq" />
     </group>
     <cache name="L1" size="L1size" />
     <memory name="shm" size="shmsize" />
   </group>
 </group>
 <memory type="global" size="gmsz" />
 ...
 <programming_model type="cuda6.0,...,opencl"/>
</device>
```

Listing 5.8: Example meta-model for Nvidia Kepler GPUs

```
<device name="Nvidia_K20c" extends="Nvidia_Kepler">
 <compute_capability="3.5" />
 <param name="num_SM" value="13" />
 <param name="coresperSM" value="192" />
 <param name="cfrq" frequency="706" ...unit="MHz"/>
 <param name="gmsz" size="5" unit="GB" />
 ...
</device>
```

Listing 5.9: Example meta-model for Nvidia GPU K20c

```
<device id="gpu1" type="Nvidia_K20c">
 <!-- fixed configuration: -->
 <param name="L1size" size="32" unit="KB" />
 <param name="shmsize" size="32" unit="KB" />
 ...
</device>
```

Listing 5.10: Example model for a concrete Nvidia GPU K20c

Cluster Modeling Listing 5.11 shows a concrete cluster model in XPDL.

The cluster has 4 nodes each equipped with 2 CPUs and 2 different GPUs. Within each node, the GPUs are attached with PCIe3 interconnect, while an Infiniband switch is used to connect different nodes to each other.

```
<system id="XScluster">
  <cluster>
    <group prefix="n" quantity="4">
      <node>
        <group id="cpu1">
          <socket>
            <cpu id="PE0" type="Intel_Xeon_..." />
          </socket>
          <socket>
            <cpu id="PE1" type="Intel_Xeon_..." />
          </socket>
        </group>
        <group prefix="main_mem" quantity="4">
          <memory type="DDR3_4G" />
        </group>
        <device id="gpu1" type="Nvidia_K20c" />
        <device id="gpu2" type="Nvidia_K40c" />
        <interconnects>
          <interconnect id="conn1" type="pcie3"
                        head="cpu1" tail="gpu1" />
          <interconnect id="conn2" type="pcie3"
                        head="cpu1" tail="gpu2" />
        </interconnects>
      </node>
    </group>
    <interconnects>
      <interconnect id="conn3" type="infiniband1"
                    head="n1" tail="n2" />
      <interconnect id="conn4" type="infiniband1"
                    head="n2" tail="n3" />
     ...
    </interconnects>
  </cluster>
  <software>
    <hostOS id="linux1" type="Linux_..." />
    <installed type="CUDA_6.0"
               path="/ext/local/cuda6.0/" />
    <installed type="CUBLAS_..." path="..." />
    <installed type="StarPU_1.0" path="..." />
  </software>
  <properties>
    <property name="ExternalPowerMeter" type="..."
              command="myscript.sh" />
  </properties>
</system>
```

Listing 5.11: Example of a concrete cluster machine

5.3.4 Power Modeling

Power modeling consists in modeling power domains, power states with transitions and referencing to microbenchmarks for power benchmarking. A power model (instance) is referred to from the concrete model of the processor, GPU etc.

Power domains or *power islands* are groups of cores etc. that need to be switched together in power state transitions. Every hardware subcomponent not explicitly declared as (part of) a (separate) power domain in a XPDL power domain specification is considered part of the default (main) power domain of the system. For the default power domain there is only one power state that cannot be switched off and on by software, i.e., there are no power state transitions.

For each explicitly defined power domain, XPDL allows to specify the possible *power states* which are the states of a finite state machine (the *power state machine*) that abstract the available discrete DVFS and shutdown levels, often referred to as P states and C states in the processor documentation, specified along with their static energy levels (derived by microbenchmarking or specified in-line as a constant value where known). A power state machine has power states and transitions, and must model all possible transitions (switchings) between states that the programmer can initiate, along with their concrete overhead costs in switching time and energy.

The dynamic power depends, among other factors, on the instruction type [67] and is thus specified for each instruction type or derived automatically by a specific microbenchmark that is referred to from the power model for each instruction.

With these specifications, the processor's energy model can be bootstrapped at system deployment time automatically by running the microbenchmarks to derive the unspecified entries in the power model where necessary.

```
<power_domains name="Myriad1_power_domains">
  <!-- this island is the main island -->
  <!-- and cannot be turned off       -->
  <power_domain name="main_pd"
                enableSwitchOff="false">
    <core type="Leon" />
  </power_domain>
  <group name="Shave_pds" quantity="8">
    <power_domain name="Shave_pd">
      <core type="Myriad1_Shave" />
    </power_domain>
  </group>
  <!-- this island can only be turned off      -->
  <!-- if all the Shave cores are switched off -->
  <power_domain name="CMX_pd"
                switchoffCondition="Shave_pds off">
    <memory type="CMX" />
  </power_domain>
</power_domains>
```

Listing 5.12: Example meta-model for power domains of Movidius Myriad1

```
<power_state_machine name="power_state_machine1"
                     power_domain="xyCPU_core_pd">
  <power_states>
    <power_state name="P1" frequency="1.2"
                 frequency_unit="GHz"
                 power="20" power_unit="W" />
    <power_state name="P2" frequency="1.6" ... />
    <power_state name="P3" frequency="2.0" ... />
  </power_states>
  <transitions>
   <transition head="P2" tail="P1"
               time="1" time_unit="us"
               energy="2" energy_unit="nJ"/>
   <transition head="P3" tail="P2" ... />
   <transition head="P1" tail="P3" ... />
  </transitions>
</power_state_machine>
```

Listing 5.13: Example meta-model for a power state machine of a pseudo-CPU

Listing 5.12 shows an example of a power domain meta-model. It consists of one power domain for the Leon core in Myriad 1, eight power domains for each Shave core, etc. The setting for attribute `enableSwitchOff` specifies that the power domain for the Leon core can not be switched off. The attribute `switchoffCondition` specifies that power domain `CMX_pd` can only be switched off if the group of power domains `Shave_pds` (all Shave cores) is switched off.

Listing 5.13 shows an example of a power state machine for a power domain `xyCPU_core_pd` in some CPU, with the applicable power states and transitions by DVFS.

```
<instructions name="x86_base_isa"
              mb="mb_x86_base_1" >
 <inst name="fmul"
       energy="?" energy_unit="pJ" mb="fm1"/>
 <inst name="fadd"
       energy="?" energy_unit="pJ" mb="fa1"/>
 ...
 <inst name="divsd">
   ...
   <data frequency="2.8" frequency_unit="GHz"
         energy="18.625" energy_unit="nJ"/>
   <data frequency="2.9" frequency_unit="GHz"
         energy="19.573" energy_unit="nJ"/>
   ...
   <data frequency="3.4" frequency_unit="GHz"
         energy="21.023" energy_unit="nJ"/>
 </inst>
</instructions>
```

Listing 5.14: Example meta-model for instruction energy cost

Instruction Energy The instruction set of a processor is modeled including references to corresponding microbenchmarks that allow to derive the dynamic energy cost for each instruction automatically at deployment time. See Listing 5.14 for an example. For some instructions, concrete values may be given, here as a function / value table depending on frequency, which was experimentally confirmed. Otherwise, the energy entry is set to a placeholder (?) stating that the concrete energy cost is not directly available and needs to be derived by microbenchmarking. On request, microbenchmarking can also be applied to instructions with given energy cost and will then override the specified values.
Microbenchmarking An example specification of a microbenchmark suite is shown in Listing 5.15. It refers to a directory containing a microbenchmark for every instruction of interest and a script that builds and runs the microbenchmark to populate the energy cost entries.

```
<microbenchmarks id="mb_x86_base_1"
                 instruction_set="x86_base_isa"
                 path="/usr/local/micr/src"
                 command="mbscript.sh">
 <microbenchmark id="fa1" type="fadd" file="fadd.c"
                 cflags="-O0" lflags="..." />
 <microbenchmark id="mo1" type="mov" file="mov.c"
                 cflags="-O0" lflags="..." />
 ...
</microbenchmarks>
```

Listing 5.15: An example model for instruction energy microbenchmarks

A *power model* thus consists of a description of its power domains, their power state machines, and of the microbenchmarks with deployment information.

5.3.5 Hierarchical Energy Modeling

A concrete system model forms a tree hierarchy, where model elements such as `socket`, `node` and `cluster` form inner nodes and others like `gpu`, `cache` etc. may form leaves that contain no further modeling elements as explicitly described hardware sub-components.

Every node in such a system model tree (e.g., of type cpu, socket, device, gpu, memory, node, interconnect, cluster, or system) has explicitly or implicitly defined attributes such as `static_power`. These attributes are either directly given for a node or derived (synthesized). Directly given attribute values are either specified in-line (e.g., if it is a known constant) or derived automatically at system deployment time by specifying a reference to a microbenchmark. Synthesized attributes[3] can be calculated by applying a rule combining attribute values of the node's children in the model tree, such as adding up static power values over the direct hardware subcomponents of the node.

5.4 XPDL Compiler and Interfacing with Other Software

The *XPDL compiler*, implemented in C++, runs statically with the input of the XPDL model files in XPDL language. Its final output is a C++ header file that implements our API for interfacing with other high level software that requires platform information to perform their optimizations of interest, or to barely check for portability.

The compilation are performed by the following steps:

1. It browses the XPDL model repository for all required XPDL files recursively as referenced in a concrete model tree, parses them, and generates an intermediate representation of the composed model.
2. It then generates microbenchmarking driver code, invokes runs of microbenchmarks wherever required to derive attributes with unspecified values, filters out uninteresting values, performs static analysis of the model (for instance, downgrading bandwidth of interconnections where applicable as the effective bandwidth should be determined by the slowest hardware components involved in a communication link).
3. Finally builds a light-weight data structure and meta-functions for the composed model that is finally written into a file. Other software can simply include the generated file as a header, and they have the capabilities to query all kinds of platform information.

[3]Note the analogy to attribute grammars in compiler construction.

Note that the use of meta-functions in our generated file allows zero run-time overhead for querying platform information, and thus can activate further compiler optimizations such as control flow simplifications (e.g., *if simplifications* for constant-valued conditions) [91].

5.4.1 Query API

There are three categories of XPDL Query API functions by our design:

- Attribute-querying: allows to query an attribute value for a specified hardware component.
- Aggregated-attribute-querying: allows to query an aggregated attribute value that the compiler produces after static analysis of all XPDL models for a given platform, such as the total number of GPUs equipped in a machine.
- Software-querying: allows to query if a software with a certain version is installed, usually used for portability checking.

Listing 5.16 shows an example for each category. The first example will give a numerical value for the number of cores in the first CPU, the second yields a value for the total number of GPUs without knowing the concrete name for each GPU. The last one tests if a software library called blas with version 4.2 is installed in the system for dependency checking. These functions look different compared to normal functions because they are meta-functions in C++ allowing zero run-time overhead. We also extend the XDPL compiler to generate macro[4] versions of the XPDL API. Listing 5.17 gives such an example, in which it shows the number of GPUs, and that cblas is not installed but OpenMP is.

```
//Attribute-query API:
xpdl::cpu_1::num_of_cores

//Aggregated-attribute-query API:
xpdl::system::num_of_gpus

//Software-query API:
xpdl::is_installed<xpdl::system::libraries,xpdl_blas_4_2>::value
```

Listing 5.16: Examples of XPDL query API by meta-functions

[4]C++ macro is not a good engineering practice, but in some circumstances it is more powerful than meta-function (e.g., disable some code snippets) and more easier to implement than its compiler-based equivalent.

```
#define XPDL_NUM_OF_GPUS 1

#define XPDL_CBLAS 0

#define XPDL_OPENMP 1
```

Listing 5.17: Examples of XPDL query API by C++ macro

5.4.2 Interfacing with Other Software

Next we show an example illustrating the interfacing and the usage of XPDL platform information. Another example is shown later in Section 6.2.2.11 for enhancing data abstraction.

A previous work using ComPU with PDL is presented in Dastgeer et al. [28], it presents the idea of *conditional composition*: programmers can encode the execution constraints of their components using ComPU's small description language, such as 'validIf(pdl :: getIntProperty (" numCudaSM ") >= 16)' for a CUDA component, saying that the component can only be executed if the number of SM on a CUDA-enabled GPU is larger or equal to 16. Then ComPU statically queries the PDL model and disables the component if it finds that the constraint is not satisfied on the current platform. Programmers could use logic operators such as AND and OR to express more complicated constraints.

We implement the conditional composition by combining ComPU with XPDL. Listing 5.18 shows how we express two selectability constraints in a CUDA component, saying that in order for it to run there should be a GPU and cublas 5.0 installed on the current constraints. One could also use AND and OR. In order to conform to XML syntax constraints, we use '{' instead of '<' in this XML descriptor.

Then the composition tool ComPU runs and inspects the constraints one by one by short-curcuit evaluation (if one fails, neglect the evaluations of the rest). If any of these constraints of a component are not satisfied, it will disable the components, and only hand over those components that pass their selectability tests as tasks to StarPU run-time systems.

```
<peppher:constraints>

  <peppher:constraint name="GPU_availability"
    expr="xpdl::system::num_of_gpus > 0" />

  <peppher:constraint name="Cublas_availability"
    expr="xpdl::is_installed
      {xpdl::system::libraries,xpdl_cublas_5_0}::value != 0" />

</peppher:constraints>
```

Listing 5.18: Express contraints by XDPL API

The advantage of combining ComPU with XPDL instead of PDL is that due to a full-phase compiler introduced for XPDL, we could query a much wider range of information about the current platform, such as system software, aggregated platform attributes, deployment-time known information (e.g., achievable PCIe bandwidth), etc. Due to meta-programming API, such queries are overhead-free and may activate further compiler optimization.

5.4.3 Portability by XPDL

How does XPDL maintain portability when applications are ported to a different system? Each host system has a XPDL model defined, thus the tool chain on top of XPDL could adapt programs to each system by utilizing each system's own XPDL model to gain portability. This allows programmers to express their portability requirements in their components, which is judged against XPDL model by the tool chain to determine whether it is portable on a system. Since all the components that are incompatible with a specific system are filtered out before compilation, the components survived the filtering are guaranteed to compile and run on the target system. Since there is at least one sequential CPU implementation available in the *ceset*, it is always runnable. In this way, we gain portability. If more than one components are checked as compatible with the current system, we can potentially gain performance portability by TunerPU's adaptive implementation selection.

Currently the XPDL compiler performs no cross-compiling, the XPDL compiler runs directly on the target machine, thus the XPDL compiler's micro-benchmarking is invoked on the target machine as well. The data obtained by the micro-benchmarking will reflect the characteristics of the target machine, such data will be further used by the tool chains on top of XPDL for further processing. An example of such further processing is: a micro-benchmark is designed to get an energy price tag for an instruction for a processor. When we compile a program on a target machine with such a processor, a XPDL compiler runs first, and invokes this micro-benchmark to extract such energy price tag for the processor of the target machine. This value can be further used by other tools, e.g., a tool that constructs an energy predictor for an implementation variant on top of each instruction energy price tag, and based on that perform implementation variant selection. Notice that such micro-benchmarking is invoked by the compiler thus performed at compile time, not run time.

5.5 Related Work

Architecture description languages (ADL) for describing hardware architecture[5] have been developed and used mainly in the embedded systems do-

[5]Note that the same term and acronym is also used for software architecture description languages, which allow to formally express a software system's high-level design in terms of coarse-grained software components and their interconnections, and from which

main since more than two decades, mostly for modeling single-processor designs. Hardware ADLs allow to model, to some degree of detail, either or both the *structure* (i.e., the hardware subcomponents of an architecture and how they are connected) and the *behavior* (i.e., the set of instructions and their effect on system state and resources). Depending on their design and level of detail, such languages can serve as input for different purposes: for hardware synthesis and production e.g. as ASIC or FPGA, for generating (parts of) a simulator for a processor design, or for generating (parts of) the program development toolchain (compiler, assembler, linker). Examples of such ADLs include Mimola, nML, LISA, EXPRESSION, HMDES, ISDL, see e.g. [90] for a survey and classification. In particular, languages used by retargetable code generators (such as HMDES, LISA, ISDL and xADML [11]) need to model not only the main architectural blocks such as functional units and register sets, but also the complete instruction set including resource reservation table and pipeline structure with operand read and write timing information for each instruction, such that a optimizing generic or retargetable code generator (performing instruction selection, resource allocation, scheduling, register allocation etc.) can be parameterized or generated, respectively; see also Kessler [71] for a survey of issues in (retargetable) code generation. Note that such ADLs differ from the traditional *hardware description languages* (HDLs) such as VHDL and Verilog which are mainly used for low-level hardware synthesis, not for toolchain generation because they lack high-level structuring and abstraction required from a toolchain's point of view. However, a HDL model could be generated from a sufficiently detailed ADL model.

Architecture description languages for multiprocessors have become more popular in the last decade, partly due to the proliferation of complex multicore designs even in the embedded domain, but also because portable programming frameworks for heterogeneous multicore systems such as Sequoia [41] for Cell/B.E. and PEPPHER [32, 12] for GPU-based systems respectively, need to be parameterized in the relevant properties of the target architecture in order to generate correct and optimized code. Sequoia [41] includes a simple language for specifying the memory hierarchy (memory modules and their connections to each other and to processing units) in heterogeneous multicore systems with explicitly managed memory modules, such as Cell/B.E.

MAML [74] is a structural, XML-based ADL for modeling, simulating and evaluating multidimensional, massively parallel processor arrays.

Hwloc (Hardware Locality) [16] is a software package that detects and represents the hardware resources visible to the machine's operating system in a tree-like hierarchy modeling processing components (cluster nodes, sockets, cores, hardware threads, accelerators), memory units (DRAM, shared caches) and I/O devices (network interfaces). Like XPDL, its main purpose

tools can generate platform-specific glue code automatically to deploy the system on a given system software platform. In this section we only refer to Hardware ADLs.

is to provide structured information about available hardware components, their locality to each other and further properties, to the upper layers of system and application software in a portable way.

ALMA-ADL [118] developed in the EU FP7 ALMA project is an architecture description language to generate hierarchically structured hardware descriptions for MPSoC platforms to support parallelization, mapping and code generation from high-level application languages such as MATLAB or Scilab. For the syntax, it uses its own markup language, which is extended with variables, conditions and loop constructs for compact specifications, where the loop construct is similar to the group construct in XPDL.

HPP-DL [108] is a platform description language developed in the EU FP7 REPARA project to support static and dynamic scheduling of software kernels to heterogeneous platforms for optimization of performance and energy efficiency. Its syntax is based on JSON rather than XML. HPP-DL provides predefined, typed main architectural blocks such as CPUs, GPUs, memory units, DSP boards and buses with their attributes, similarly to the corresponding XPDL classes. In comparison to XPDL, the current specification of HPP-DL [108] does not include support for modeling of power states, dynamic energy costs, system software, distributed specifications, runtime model access or automatic microbenchmarking.

What ADLs generally omit is information about system software such as operating system, runtime system and libraries, because such information is of interest rather for higher-level tools and software layers, as for composing annotated multi-variant components in PEPPHER and EXCESS. As discussed earlier, PDL [110] models software entities only implicitly and ad-hoc via free-form key-value properties. None of the ADLs considered puts major emphasis on modeling of energy and energy-affecting properties. To the best of our knowledge, XPDL is the first ADL for heterogeneous multicore systems that provides high-level modeling support also for system software entities, and that has extensive support for modeling energy and energy-affecting factors such as power domains and power state machines.

5.6 Summary

We proposed XPDL, a portable and extensible platform description language for modeling performance and energy relevant parameters of computing systems in a structured but modular and distributed way. It supports retargetable toolchains for energy modeling and optimization, and allows application and system code to introspect its own execution environment with its properties and parameters, such as number and type of cores and memory units, cache sizes, available voltage and clock frequency levels, power domains, power states and transitions, etc., but also installed system software (programming models, runtime system, libraries, compilers, ...). We observe that, beyond our own work, such functionality is needed also in other projects in the computing systems community, and we thus hope that this work will contribute towards a standardized platform description language promoting retargetability and platform-aware execution.

Chapter Acknowledgements

This research was partially funded by EU FP7 project EXCESS (www.excess-project.eu) and SeRC (www.e-science.se) project OpCoReS.

We thank Dmitry Khabi from HLRS Stuttgart for suggestions on cluster modeling. We thank Jörg Keller from FernUniv. Hagen, Anders Gidenstam from Chalmers and all EXCESS project members for comments on this work. We also thank Ming-Jie Yang for improvements of the current XPDL toolchain prototype.

Chapter 6

Handling Data Management Complexity

This chapter is based on the following paper:

- Li, L. and Kessler, C. (2017b). VectorPU: A Generic and Efficient Data-container and Component Model for Transparent Data Transfer on GPU-based Heterogeneous Systems. In *Proc. 8th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and 6th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM'17)*. ACM
- Li, L. and Kessler, C. (2017a). Lazy allocation and transfer fusion optimization for GPU-based heterogeneous systems. In *Proc. Euromicro PDP-2018 Int. Conf. on Parallel, Distributed, and Network-based Processing*. IEEE

The final type of complexity that we will address in this thesis is data management complexity. This is particularly important for GPU-based heterogeneous systems, because data transfer has long been a severe performance bottleneck ever since the introduction of traditional sequential or multi-CPU computer systems, For high performance GPU-based heterogeneous systems this issue can be a more performance-constraining factor as data transfer on PCIe bus can be even slower than data transfer through memory hierarchy on sequential or multi-CPU systems. Another reason is that a large proportion of code on heterogeneous systems can be only for data management, which put a heavy burden the programmer. Jablin et al. [57] shows that 82% of code in a simple CUDA program is the logic related to data allocation and coherence management.

Compared to sequential or multicore processor systems, the data management on GPU-based heterogeneous systems further complicates programming on such systems. Programmers may have to allocate memory blocks

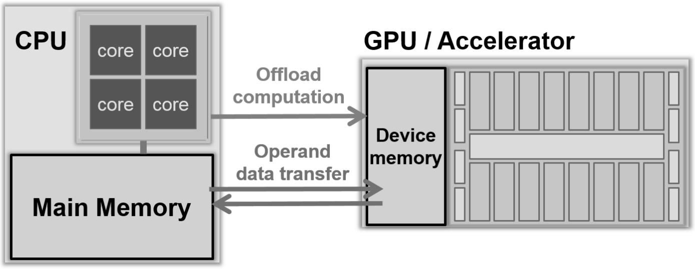


Figure 6.1: A typical GPU-based system

for operands on both sides (CPU memory and GPU memory), and deallocate them when they are no longer needed. Programmers are also required to manually transfer the data back and forth to make sure that when it is time to invoke a kernel either on CPU or GPU, the kernel's arguments are most recent if read accesses to them are performed by the kernel.

Can we abstract data management away for an arbitrary kernel? Can we achieve a design of such programming abstraction while maintaining efficiency and expressiveness? How to further optimize data transfer based on a designed data abstraction? In this chapter we will explore these questions.

Section 6.1 discusses the data management complexity in more details. Section 6.2 describes our design of data abstraction on GPU systems, which we name VectorPU. Section 6.3 proposes two data transfer optimizations. Although these optimizations are orthogonal to specific frameworks, we implement and evaluate them in VectorPU.

6.1 Data Management Complexity

The data management complexity on most high-end GPU systems[1] can be mainly attributed to the separate address spaces for CPUs and GPUs, with different programming APIs for data management on the different processor memories. This, in many cases, leads to the need to allocate and maintain at least two copies of the same data, and transfer them explicitly to keep these copies coherent. Figure 6.1 shows such a typical example, when we offload computations to a GPU, operand data is transferred back and forth between main memory and device memory. From a programmability's point of view, coding for GPU systems becomes harder, as programmers now have more responsibilities in addition to coding for the kernel computations. Many low level details are needed, such as the number of bytes for a data object. The resulting code becomes hard to understand, as the core business logic is hidden in the data management code, and as a result, the code is hard to

[1]Some GPU systems, e.g., mobile GPUs may not have their own separate address space.

maintain.

From the performance point of view, adding more responsibilities to the programmers increases the risk of careless redundant data transfers, which is a pure performance loss. Furthermore, there is a strong need to optimize data transfers under the hood, as they are often the performance bottlenecks[2] of a program on a heterogeneous computer system.

6.2 Framework Design and Evaluation

A data abstraction for heterogeneous systems can greatly improve the programmability of such systems. Specifically a vector abstraction can be preferable, as it represents a continuous chunk of memory which can be transferred between CPU and GPU memory efficiently. And more complex data structures can be built based on this basic building block, such as dense matrix (e.g., represented by a vector, if flattened in row/column major order) and sparse matrix (e.g. the CSR sparse matrix representation which consists of three vectors).

Furthermore, the STL library has been a success for data abstraction on CPU memory, and Nvidia CUDA Thrust [97] provides a STL-like data abstraction on the GPU side, and both provide high quality algorithms. A data abstraction that can be easily integrated with STL and Thrust algorithms potentially improves the programmability even further.

Therefore, we designed *VectorPU*, a C++ library that provides a vector abstraction on (Nvidia) GPU based heterogeneous system, with automatic data allocation and coherence management, and compatible with the STL and Thrust algorithm libraries. VectorPU shows 1.40× to 13.29× speedup compared to *Nvidia's Unified Memory* [50] and no slowdown compared with manual expert-written code.

Section 6.2.1 discusses our approach and the design of VectorPU, and Section 6.2.2 shows the wide applicability of our approach.

6.2.1 VectorPU Design

For a general program, we can organize its main entities as data structures and functions.

- The data structures can be arbitrarily complex, but for the interest of memory coherence, vectors are usually beneficial for efficiency reasons (data transfers are most efficient if the memory to be transferred is continuous, see also Section 6.3). Data containers, like STL vector, allow reuse of some management functionalities such as memory allocation and deallocation etc., with possibly higher execution efficiency than manual code.

[2]If not, then most programs can achieve or nearly achieve peak performance, which is obviously not true.

- The functions, no matter templated or not, usually have fixed access properties (e.g., read, write or readwrite) for their parameters. For traditional CPU programs, such properties are not vitally important to express explicitly, because all data objects of a program are loaded in the main memory of a system. In contrast, the access properties are significantly more important in (GPU-based) heterogeneous systems where different physical memory modules exist for different kinds of processing units, and such access properties can decide if a data movement is necessary. As an example, for read-only parameters of a CUDA kernel, there is no need to move the data back after the kernel call; for a write access to an invalid memory copy, making it up-to-date before the write access is also unnecessary.

While there is a clear need for such access properties of parameters of functions, unfortunately compilers are in general not able to extract such information automatically for functions by static analysis. For example, via pointer arithmetics two arrays `a` and `b` may intersect with each other, then modifying `a` will lead to modifying `b`, but as a compiler can not statically detect that `b` is modified, we can only expect a conservative "may" answer given by compilers. Furthermore, source code may not even be available. On the other hand, these properties are not so difficult for programmers or code readers to deduce. Therefore, some annotation language to encode such access properties becomes a natural solution, to allow programmers to express them while developing the software components. Furthermore, data containers should be able to process such information and perform data transfer only when necessary in order to improve execution efficiency.

6.2.1.1 Annotation DSEL

In VectorPU, we design a DSEL (Domain-Specific Embedded Language) to annotate each parameter or argument. Table 6.1 lists its typical annotations. We have pointer annotations that, when used, can translate a VectorPU data structure (described in Section 6.2.1.3) to a pointer with necessary and only necessary coherence for its pointee. For example, `RW(x)` will return a pointer of the host memory represented by `x`, and also make sure that the most recent copy is already on host memory and the memory on its device copy is marked as invalid. We can also get iterators of VectorPU data structures by iterator annotations such as `RI(x)`. For a parameter or argument that does not require coherence, such as an array size, we can use `NA` to denote "not applicable".

Access Property	Host	Device
Read pointer	R	GR
Write pointer	W	GW
Read and Write pointer	RW	GRW
Read Iterator	RI	GRI
Read End Iterator	REI	GREI
Write Iterator	WI	GWI
Write End Iterator	WEI	GWEI
Read and Write Iterator	RWI	GRWI
Read and Write End Iterator	RWEI	GRWEI
Not Applicable	NA	NA

Table 6.1: VectorPU's annotations for a parameter

6.2.1.2 Flow Signature, VectorPU Component

```
 1 void foo(const float *x, float *y,
 2   float *z, int size);
 3
 4 //(alpha signature): one-time annotations
 5 //--------Call VectorPU Component----------
 6 foo(R(x), W(y), RW(z), size);
 7
 8 //(beta signature): reusable
 9 //----VectorPU Component Definition------
10 #define bar_flow (GR)(GW)(GRW)(NA)
11 __global__
12 void bar(const float *x, float *y,
13   float *z, int size){
14   ...
15 }
16
17 //Call VectorPU Component with a beta signature
18 CALL( (bar) ((<<<32,256>>>)) ((x,y,z,N)) );
19 CALLC( (bar) ((<<<32,256>>>)) ((x,y,z,N))          (bar_flow) );
20
21 //(gamma signature): reusable and natural
22 //----VectorPU Component Definition------
23 __global__
24 void bar(const float *x[[GR]], float *y[[GW]],
25    float *z[[GRW]], int size)
26    { ...  }
27
28 //Call VectorPU component as beta signature
```

Listing 6.1: Flow signatures

In order to denote access properties for all parameters or arguments of a function signature or a function call, we define a concept called *flow signature*: an ordered sequence of annotations for every parameter's or argument's access property in a function signature or a function call. As illustrated in Listing 6.1, three forms of flow signatures can be defined:

1) α (flow) signature: we can denote the access property for each parameter at the call site of a function, as illustrated in Listing 6.1 line 6, where

`R(x)` annotates `x` to be used as a read-only parameter by the CPU function `foo()`. `R(x)` also translates `x` to a pointer required by `foo()`. Although it is simple to annotate, there is no reuse of such meta-data for another call to the same function. Annotations such as `R(x)` are, under the hood, C++ macros developed on top of the Boost library.

2) β signature: in order to reuse meta-data for the same function, we can define a flow signature separately by a C++ macro, thus there is no need to annotate multiple calls to the same function each time. As illustrated in Listing 6.1 line 10, the sequence of the annotations for parameters in the flow signature is the same as those in its function signature. The naming convention for a β signature is the function name suffixed by a string "`_flow`". We use "`G`" (GPU) to denote the parameter's location, thus `GW` signals that the argument to this parameter will be translated to a device pointer with necessary coherence knowing that it will be write-only. `NA(size)` denotes that no coherence is needed for the parameter `size`.

Line 18 shows how to call a VectorPU component with a β signature, the `CALL` macro takes a three-element tuple as input: the function name, the invocation configuration (for host function this element is empty, but double parentheseses is still required), and the run-time arguments. The `CALL` macro takes the default flow signature name, such as "`bar_flow`" in this case, thus no flow signature argument is specified. Line 19 shows another way to call a VectorPU component with a β signature. The `CALLC` (Call with a Customized flow signature) macro takes an explicit flow signature name as one extra parameter. In cases where the default macro name already exists or a function is overloaded (in such cases only one function can use the default flow signature name), `CALLC` can be used to avoid name clashes.

3) γ signature: for a clean interface we can utilize C++11 attributes to express a flow signature as illustrated in lines 24-25 in Listing 6.1. Each access property is directly adjacent to its target parameter, thus readability is improved. However, it requires a source-to-source compiler to transform γ signatures to β signatures under the hood for the native compiler to process. We built such a compiler called `vpucc` based on clang 3.9 to perform the source-to-source transformation.

It is important to notice that α and β signatures do not require the availability of the source code, thus VectorPU allows annotations for functions from binary libraries. We assume that a function's return value is trivial (e.g. a status code), and does not need coherence management, which is usually the case. If not, the function's return value can be moved to one extra parameter, and the transformed form can be expressed as a flow signature. Our flow signatures ignore return value annotations.

A call or a function that is annotated by either α, β or γ signature makes the function a *VectorPU component*.

6.2.1.3 VectorPU Vector

A *VectorPU vector* is a data container to expose a single unified continuous block of memory to programmers for heterogeneous programming, with automatic memory coherence management for host and device memory on GPU-based systems. Internally it inherits from STL and Thrust `vector`.

When a VectorPU `vector` is declared with the required number of elements, the amount of memory in bytes are calculated and allocated automatically both on host and device. This is reasonable because when we intend to use GPUs for computations, we usually first allocate host and device memory of the same size anyway, then transfer the data between them. If a programmer intends to use only CPU memory for some vector variables in a heterogeneous system, a STL vector can be used to avoid unnecessary overhead.

Regarding the supported heterogeneous data structures, in principle `list` etc. can also be added in the VectorPU library, but for efficiency reasons, `vector` is preferred because data transfer is most efficient if the data to be transfered is continuous. Other data structures can be converted from a VectorPU `vector` after the coherence action is finished, and the conversions are usually much cheaper as memory transfer between device and host is not needed anymore.

6.2.1.4 A Motivating Example

With VectorPU component and vector, one can write an accelerator application easily without programming the data transfers. Listing 6.2 shows an example with β signature. We can annotate a normal GPU function in line 1, and VectorPU guarantees the three arguments to be in correct coherent states in device memory before the function call. The flow signature also guarantees that each argument is translated to a raw device memory pointer as required by `bar()`.

```
1 #define bar_flow (GR) (GW) (GRW) (NA)
2 __global__
3 void bar(const float *r, float *s, float *t,
4          const int size) { ...  }
5
6 int main(){
7   vectorpu::vector x(10), y(10), z(10);
8   CALL( (bar) ((<<<32,256>>>)) ((x,y,z,10)) );}
```

Listing 6.2: An example application with VectorPU

6.2.1.5 Coherence Management

We use a simplified MESI coherence mechanism. Each (device or host) memory segment of VectorPU `vector` has only two states: invalid (not most recent copy) or valid (most recent copy). For different combinations of locations (CPU or GPU) and access modes (`R`, `W`, or `RW`), we have six situations to consider, only three of them (GPU cases) are listed here as the others are similar. At any time during program execution, at least one copy is valid.

1. `GR`: if the coherence flag of a GPU memory copy is invalid, then we transfer the valid copy from the CPU memory, and set the GPU coherence flag as valid.
2. `GW`: set the GPU memory coherence flag to be valid, and the CPU memory coherence flag to be invalid.
3. `GRW`: the same as the `GR` case, additionally set the CPU memory coherence flag to be invalid.

The reason why we use a simplified MESI coherence mechanism is that when we declare a VectorPU `vector`, it is already in a shared state in MESI. If programmers do not intend to use a container that is already shared, which is reasonable only in the scenario that only the CPU memory copy is needed, then a STL `vector` can be defined instead. In other words, we rely on programmers to choose the right container, which allows for a simpler coherence algorithm, and less run-time overhead. The programmers usually can decide such choices, as they already do so in normal CUDA programs by explicit invocations of `malloc()` or `cudaMalloc()`. Data transfer is performed only lazily, and in case of a write-only parameter, no data transfer to device is needed at all.

6.2.2 Expressiveness of VectorPU

In this section we show the wide range of use cases where VectorPU can apply.

6.2.2.1 Basic and Customized Data Types

VectorPU `vector` can be instantiated with a basic data type as well as a customized data type, illustrated in Listing 6.3. If it is initialized with a customized data type, the memory is arranged as array of structs on both host and device memory.

```
vectorpu::vector<int> x;
vectorpu::vector<MyType> x;
```

Listing 6.3: Vectors with different element data type

6.2.2.2 Smart Iterators

We can also use iterator annotations, with one extra "`I`" such as GRI (GPU Read Iterator) and GREI (GPU Read End Iterator), to get the starting and ending iterators from a VectorPU `vector`. Those iterators are *smart* as they guarantee lazy data movement, and they can be used either by

STL or Thrust algorithms depending on whether a host or device iterator is returned by the annotations. Thus heterogeneous programming can be simplified as in Listing 6.4. With only 4 lines of code, we declare a vector with compound data type, let the CPU generate a random sequence and the GPU sort it, and finally print it on screen. Listing 6.5 shows a hybrid computing example. Line 4 invokes a GPU kernel asynchronously, and line 5 invokes a CPU kernel, hence the CPU and GPU kernel run simultaneously.

The VectorPU iterators allow code productivity by reuse of STL and Thrust algorithms which hide tuning-relevant parameters such as grid and block size. VectorPU's annotations for translation to raw pointers allow to develop high performance code, as these raw pointers can be used by native device functions where all tunable parameters (e.g. block size) are exposed for further (manual- or auto-) tuning.

It is worth mentioning that VectorPU does not either force or change synchronizations on components. If one of a CPU and a GPU component annotated by VectorPU is asynchronous, and the call to the asynchronous component appears first followed by the call to the other component, then CPU and GPU will be busy at the same time when executing the two components.

```
vectorpu::vector<My_Type> x(N);
std::generate(WI(x), WEI(x), RandomNumber);
thrust::sort(GRWI(x), GRWEI(x));
std::copy(RI(x), REI(x), ostream_iterator<My_Type>(cout, ""));
```

Listing 6.4: Heterogeneous programming

```
1 vectorpu::vector<My_Type> v1(N), v2(N);
2 std::generate(WI(v1), WEI(v1), RandomNumber);
3 std::generate(WI(v2), WEI(v2), RandomNumber);
4 userspace::sort<<<8,256>>>(GRWI(v2),GRWEI(v2));
5 std::sort(GRWI(v1), GRWEI(v1));
6 cudaDeviceSynchronize();
```

Listing 6.5: Hybrid computation

6.2.2.3 Lambda Functions

VectorPU can not only perform coherence management for normal (host or device) functions, but also for code snippets annotated as VectorPU *lambda functions* for an analogy to C++11 lambda functions. At present this feature only applies to host code snippets, as applications usually contain such snippets to initialize data that are to be transfered to GPU for computations.

```
1  //alpha signature
2  for(std::size_t i=0; i<N; ++i) {
3    W(z)[i]=3; }
4
5  //beta signature
6  #define lambda(z) \
7        for(std::size_t i=0; i<N; ++i)  { \
8          z[i]=3; }
9  #define VECTORPU_DESCRIBE VECTORPU_META_DATA(
10           int , z , W)
11 VECTORPU_LAMBDA_GEN
```

Listing 6.6: VectorPU lambda functions

Listing 6.6 shows a code snippet that can be annotated by α or β signatures. An α signature in lambda functions is the same as in normal functions. Notice that in line 3 the annotation `W` on `z` performs the coherence management on the whole vector, for partial coherence management please see Section 6.2.2.8. A β signature allows to reuse the flow annotation for a parameter. In a β signature, we first define the lambda function by the `lambda` macro with all VectorPU `vectors` (used in the code snippet) in the parameter list, the body of the macro is the code snippet which is unchanged in its original form in non-VectorPU applications. Then we have another macro to define VectorPU flow signatures, such as type, name, access modes, and finally we have a third macro to generate necessary glue code. In code snippets where there are repetitive uses of the same VectorPU `vector`, β signatures are more compact descriptions, as only one annotation for the `vector` is needed for its multiple uses.

6.2.2.4 VectorPU Algorithms

With VectorPU annotations for iterators, we can define VectorPU algorithms as algorithms on heterogeneous systems that can take host or device data arbitrarily and under the hood only transfer necessary data. VectorPU algorithms can be implemented using VectorPU flow signatures on top of the STL and Thrust algorithms.

Listing 6.7 shows an example using `vectorpu::copy` with the same semantics but parameterized by heterogeneous iterators, thus allows to copy data between host and device. In this example the host memory of `x` will not be updated if it is already a valid copy, and the host memory of `y` will be automatically invalidated since there is new data written to its device memory. Although here an α signature is used, other flow signatures also apply.

```
1  vectorpu::copy(RI(x), REI(x), GWI(y));
```

Listing 6.7: VectorPU algorithms

6.2.2.5 Overloaded Functions

If all overloaded functions under the same name share the same flow signature, a common flow signature can be defined and reused for these functions, and at call sites all calls to these overloaded functions load the same flow signature and get the same coherence state before these calls.

If all overloaded functions have different flow signatures, each function can be equipped with its own flow signature, and at call site, callers load their own flow signature, and get their preferred coherence states.

The namespace information can be encoded in a flow signature name to avoid name clashes; in the future, compiler support can be developed, thus the flow signature name can be mangled like C++ functions.

6.2.2.6 Template Functions

Usually for a template function, an instantiation for a different template parameter does not alter its access properties (such as read, write or readwrite), and we can naturally write a reusable flow signature to all its template instantiations.

For rare cases where changing template parameters does change their access properties, different flow signatures for the same function can be defined with different names, and can be loaded at different call sites whenever necessary.

6.2.2.7 Expressing Skeletons in VectorPU

```
 1 struct my_set{
 2   template <class T>
 3   __host__ __device__
 4     void operator() (T &x) { x+=101; }
 5 };
 6
 7 int main() {
 8   vectorpu::vector<int> x(N);
 9   vectorpu::for_each<int>(GRWI(x), GRWEI(x), my_set() );
10   vectorpu::for_each<int>(GWI(x), GWEI(x),
11             [] __device__ (int &x){x=10;} );
12   vectorpu::for_each<int>(RI(x), REI(x),
13             [](int const &x) {cout<<x<<”,”;} ); }
```

Listing 6.8: Skeleton programming by VectorPU

Skeleton programming is a popular high-level programming model on heterogeneous systems. Skeletons such as map, reduce, scan, stencil, have well-defined semantics and access properties for their parameters. Listing 6.8 shows an example for a map skeleton by VectorPU, with three user functions (a *user function* is a piece of problem-specific code that is called by skeletons typically on each data element, similar to lambda functions in C++11), two to set a vector variable and one to print it. For reusable user functions, a functor can be defined (lines 1-5) that can be used for skeletons on both CPU and GPU side. The functor is invoked in line 9. For one-time use user functions, line 10-11 and 12-13 calls a map skeleton with a device and host lambda function respectively. More specifically, Line 10 is a device-side lambda (indicated by the keyword `__device__`) that set each element of the vector `x` to 10. The memory coherence is transparent and the code is compact and readable.

6.2.2.8 Partial Coherence for Vectors

VectorPU supports partial coherence for its vectors to allow flexible coherence granularity and avoid unnecessary data transfer for better performance. It provides a data container `parco_vector` (abbreviated as `pvector`) which is initialized from CPU memory iterators of a VectorPU `vector`. `pvector` only contains iterators that point to a `vector`'s CPU and GPU memory. When it is initialized with CPU memory iterators, the GPU memory iterators are set automatically for the same range. The coherence state of a `pvector` is inherited from the `vector` it points to. After initialization, programmers can use VectorPU algorithms on `pvector`s in the same way as for VectorPU `vector`s. Then the coherence only happens at the subsection that the `pvector` contains, so that only necessary data are transfered and coherence cost becomes lower. Note that the partial array coherence is supported in previous implementation for Java JIT compiler [55] and SkePU's smart containers [27].

6.2.2.9 Flow Signature Switching

Even if a reusable β or γ signature is defined at function level, at any call site, one can change the flow signature by using an α signature at call site without loading the reusable signature. This switching makes sense when we pass two `pvector`s that intersect with each other, thus the function level flow signature becomes incorrect for this situation as it in general does not assume such overlap in its parameters. The caller can reflect the peculiarity by flow signature switching.

6.2.2.10 Multi-GPU Support

VectorPU supports multi-GPU systems. VectorPU's data container `big_vector` can manage the allocation and coherence of the memory of CPU and multiple GPUs. For the parameters described by the flow signature, instead of returning a single pointer or iterator by the VectorPU vector in Section 6.2.1.3, the annotations (except `NA`) translate each parameter to a `zip_pointer` or `zip_iterator`, which is a tuple of pointers or iterators, one per each GPU. The code to define type `zip_pointer` or `zip_iterator` is generated from a model of the target system in XPDL platform description language, from which the number of GPUs is known.

For `big_vector`, at the declaration phase the memory on both CPU and multi-GPU side is allocated. For the CPU side, the memory is allocated in one big continuous chunk, thus it allows for both efficient sequential and parallel memory operations. On the multi-GPU side, the same data of memory is partitioned evenly or nearly evenly and allocated on the different GPUs' global memory by default if such as an partition is beneficial for load balance. In the future we can expose the parameter on how to cut the input data to application-specific tuning. XPDL (see Section 5) can be utilized to look up how much global memory is available on each GPU. In the future, we can also consider the global memory size on each GPU from XPDL for memory allocation for `big_vector`.

For memory coherence, VectorPU internally uses the same mechanism of Section 6.2.1.5 one by one for each GPU which is already implemented in `big_vector`. For the internal representation of memory coherence state, each GPU has its own state, thus in principle it is possible to have different coherence states for each GPU before and after each one VectorPU component invocation, but this would require more expressiveness from the flow signature. Whether it is worthwhile for an extension depends on if such pattern occurs frequently in applications, and we leave it for our future work.

Listing 6.9 shows a code example of using VectorPU's `big_vector`. We initialize a `big_vector` with specified size and constant value in line 29, use all available GPUs as accelerators to update the vector in line 30, and print its value on the screen from CPU-side memory in line 31. The data transfer is automatically managed by the help of an α flow signature. From the VectorPU component point of view, it catches a `zip_pointer`, and can extract different pointers and sizes on different GPU devices. A GPU component can internally decide the synchronization among different GPU devices, and perform performance tuning on the block size etc. To improve programmability even more, in the future we can also develop libraries of VectorPU components that can hide the extraction of `zip_iterator`s, cover common computational patterns, and adapt to different numbers of GPU devices as told by XPDL. To improve the tuning, we can also add the facility to allow tuning the number of GPUs to use for a call, instead of using all available GPUs.

```
__global__
void gpu_kernel(int *p, std::size_t N)
{
  for(std::size_t i=0; i<N; ++i)
    (*p++) +=300;
}

void gpu_func(vectorpu::zip_pointer<int> &x, std::size_t N)
{
  cudaSetDevice(0);
  gpu_kernel<<<1,1>>>(x.p0, x.size0);
  cudaSetDevice(1);
  gpu_kernel<<<1,1>>>(x.p1, x.size1);
  cudaDeviceSynchronize();
}

void cpu_func(vectorpu::zip_pointer<int> const &x, std::size_t N)
{
  //programmers can process all data from CPU memory
  //in one loop as it is continuous
  auto p=x.p0;
  for(std::size_t i=0; i<x.size; ++i)
    cout<<*(p++)<<" ";
}

int main()
{
  const std::size_t N=100;
  vectorpu::big_vector<int> x(N, 300);
  gpu_func( GW(x), N );
  cpu_func( R(x), N );
}
```

Listing 6.9: VectorPU on multi-GPU systems

6.2.2.11 VectorPU's Self-adaptive Vector

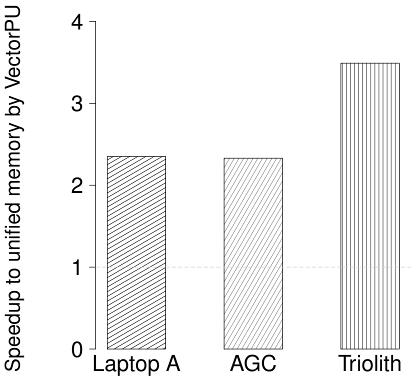
From a system model in XPDL, the number of GPUs equipped in a system can be known statically, and used to reshape VectorPU code statically as well. VectorPU contains a generalized data container: `self_adaptive_vector`.

- If no GPUs are available, `self_adaptive_vector` is barely a STL vector.
- If one GPU is available, `self_adaptive_vector` is a VectorPU `vector` in Section 6.2.1.3.
- If multiple GPUs are available, `self_adaptive_vector` is a VectorPU `big_vector` in Section 6.2.2.10.

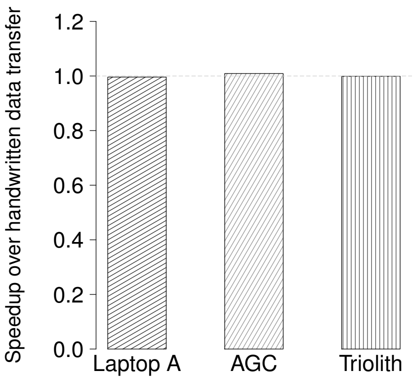
At present, `self_adaptive_vector` is only an experimental feature, because the different vectors above have different interfaces, so it is not possible yet to write VectorPU container-independent code. In the future we plan to develop a library of VectorPU components to bridge the difference, integrate our automatic back-end selection technique [81] with VectorPU,

Table 6.2: Machine configuration

Machine	CPU	GPU	CUDA
Laptop A	Intel(R) Core(TM) i7-4710MQ @ 2.50GHz	1 K2100M (Kepler)	7.5
AGC (workstation)	Intel(R) Xeon(R) E5-1620 v3 @ 3.50GHz	1 K620 (Maxwell)	7.5
Triolith n1598 (supercomputer)	Intel(R) Xeon(R) E5-2660 0 @ 2.20GHz	1 K20Xm (Kepler)	7.5



(a) Conjugate Gradient, compared with Nvidia's UM.



(b) FFT, compared with handwritten CUDA code.

Figure 6.2: Benchmark results

thus programmers can use `self_adaptive_vector` and more general functions which dynamically select CPU, GPU etc. under the hood, and let VectorPU perform data transfer when necessary.

6.2.3 Experimental Results

We use three machines of different hardware configurations for performance evaluation, ranging from a laptop, a workstation to a node in a supercomputer, and from Kepler GPUs to Maxwell GPUs, shown in Table 6.2.

We use MeterPU (see Section 3) for measurement of evaluation metrics such as time. We perform 100 repetitions of executions for each setting, and plot the arithmetic mean of the measured times.

6.2.3.1 Conjugate Gradient, Comparison to UM[3]

For performance evaluation, we use the code for a conjugate gradient solver[4] from the CUDA 7.5 SDK using Nvidia's unified memory. We rewrote it using VectorPU `vector` and component model (8 VectorPU vectors, 6 VectorPU components, 12 component invocations and 3 VectorPU lambda functions), and compared the performance with the code using Nvidia's unified memory. We use β signatures to annotate cuBLAS functions to reuse the annotations, and the lambda function feature of VectorPU to annotate the code snippets that initialize CPU-side memory of VectorPU smart vectors; thus, rewriting is trivial and not error-prone.

We observe that with VectorPU, we can gain significant speedup over Nvidia's Unified Memory in Figure 6.2a. The speedup is purely obtained from the data movement part, as this is the only difference for the two versions of the conjugate gradient solver code. As our profiling result shows, the inefficiency of unified memory lies in the page-based data transfer mechanism: transferring data page by page suffers from numerous communication setup and destruction cost.

The information on data access modes for cuSPARSE and cuBLAS functions used in the conjugate gradient solver such as read or write, on host or device memory etc., is well documented in Nvidia's online documentation [96, 99], and VectorPU enables those documentations to play an important role as executable meta-data to boost performance.

6.2.3.2 Reduction, Sorting, Comparison to UM

We use a typical GPU parallel reduction code which recursively reduces the second half of the data elementwise by a sum operator to its first half until one element is left, and write two versions using unified memory and VectorPU. We achieved 1.40× to 8.66× speedup on three different machines in Table 6.2. We also obtained 13.29× speedup on a sorting benchmark on 1M element which internally uses Thrust `sort()`, by rewriting its memory management with VectorPU instead of Nvidia's Unified Memory.

6.2.3.3 FFT, Comparison to Manual Code

We choose another code example, FFT, from CUDA 7.5 SDK with programmer-managed coherence by explicit calls to `cudaMemcpy()`. It is reasonable to assume that the FFT code from the SDK is expert-written, and no unnecessary data transfer is performed. Then we rewrite the code with VectorPU by removing all data transfer code and adding annotations for functions with α and β signatures (2 VectorPU vectors with compound data type, 4 VectorPU components and 9 component invocations).

[3]An abbreviation for Nvidia's Unified Memory

[4]In fact, this is the only benchmark using unified memory available with source code in the CUDA 7.5 SDK.

Figure 6.2b shows that VectorPU-managed coherence is as efficient as programmer-managed coherence, even though the programmer is an expert. Figure 6.2b also shows that the overhead for VectorPU's automatic memory coherence is at the noise level and negligible. The overhead only consists of checking and updating some boolean variables as coherence flags.

6.2.3.4 Programmability Improvement, Comparison to Manual Code and UM

From programmability's point of view, using VectorPU removes many responsibilities from programmers to write normal CUDA programs, such as allocating memory on both host and device, explicit moving data with the calculation of the memory size and carefulness to avoid unnecessary movements, freeing those memory copies and freeing them only once etc. The number of lines of code for the simple example vectorAdd from the CUDA SDK drops from 75 to 24 after rewriting with VectorPU (the lines that only contain a printf or brackets are not counted). The number of lines of code for a parallel reduction by unified memory drops from 21 to 17 after rewriting with VectorPU, as for a VectorPU `vector`, only one line of code can perform the allocation and initialization of both host and device memory, and there is no need to free those memories explicitly.

Thus with VectorPU it is much easier to program for heterogeneous architectures, giving the programmers the options to code for either productivity or high performance with both iterators and raw pointers at hand. With a read-only argument, a `const` pointer or iterator will be returned, thus it is less error-prone.

6.2.3.5 Comparison to OpenACC

We study the code overhead for annotating coherence by a comparison among VectorPU, OpenACC and StarPU. We annotate a convolution kernel and its caller as a case study. All kernels and their calls are annotated in a similar way for OpenACC, StarPU and VectorPU. The code size of the kernel code is not interesting and thus we don't take the kernel code size into considerations, no matter how big or small it is. We study the code overhead that integrate the kernel with each of the three frameworks.

The code overhead for annotating coherence (not including annotations for computations in the OpenACC case) is on par for the cases of VectorPU and OpenACC, as shown in the Listing 6.10 for the VectorPU case and Listing 6.11 for the OpenACC case.

VectorPU's advantage in code style is that OpenACC's annotations employ command-like style, e.g., copy, create, while VectorPU's annotations employs a property-like style. In short, using VectorPU's access properties such as read or write, the OpenACC's command annotations like copy can be automatically deduced. Access properties (like read or write) can be immutable for a region of code or a component, but commands for the code region or the component does not necessarily have this property depending on the memory organizations used, e.g., for a distributed memory system, such copy command for a code region or a component makes sense as we do need to copy, but for shared memory systems, copy is not needed anymore for the code region or component. However, the code region's or component's access properties like read or write will remain the same for both distributed and shared memory systems. This immutability usually encourages reuse of annotations. VectorPU's β signature allows such reuse of the data access properties for a function. data access patterns with pointer arithmetics.

```
#define convolution_flow (GR) (GW) (NA) (NA)
static __global__ void convolution(
    const float *A, float *Anew, int n, int m)
{
    int j=threadIdx.x+1;
    for ( int i = 1; i < m-1; i++ ) {
      Anew [j*n + i] = 0.25 * ( A [j*n+i+1] + A [j*n+i-1]
                          + A [(j-1)*n+i] + A [(j+1)*n+i]);
    }
}

CALL( (convolution) ((<<<1, n-2>>>)) ((A, Anew, m, n)) );
```

Listing 6.10: VectorPU code

```
#pragma acc data copy(A) create(Anew)
while ( iter < iter_max ) {
  #pragma acc kernels {
    #pragma acc loop independent collapse(2)
      for ( int j = 1; j < n-1; j++ ) {
        for ( int i = 1; i < m-1; i++ ) {
          Anew [j] [i] = 0.25 * ( A [j] [i+1] + A [j] [i-1] +
                              A [j-1] [i] + A [j+1] [i]);
        }
      }
  }
}
```

Listing 6.11: OpenACC code

```c
static __global__ void convolution(const float *A, float *Anew, int n, int m){
  int j=threadIdx.x+1;
  for ( int  i = 1; i < m-1; i++ )  {
    Anew [j*n + i] = 0.25 * ( A [j*n+i+1] + A [j*n+i-1]
      + A [(j-1)*n+i] + A [(j+1)*n+i]);
  }
}

void convolution_gpu(const float *A, float *Anew, int n, int m){
  convolution<<<1, n-2>>>(A, Anew, n, m);
  cudaThreadSynchronize();
}

void conv_gpu_wrapper (void *buffers[], void *_args) {
  convolution_gpu (
   (float *)STARPU_VECTOR_GET_PTR((struct starpu_vector_interface *)buffers[0]),
   (float *)STARPU_VECTOR_GET_PTR((struct starpu_vector_interface *)buffers[1]),
   ((ROA_convolution *)_args)->n,((ROA_convolution *)_args)->m);
}

typedef struct {
  int n;
  int m;
} ROA_convolution;

typedef struct {
  struct starpu_codelet cl_convolution;
  int cl_convolution_init;
} struct_convolution;

void convolution ( const float * A, float * Anew, int n, int m )
{
  static ROA_convolution arg_convolution;

  arg_convolution.n=n;
  arg_convolution.m=m;

  static struct_convolution * objSt_convolution = NULL;

  if( objSt_convolution == NULL)
  {
    objSt_convolution=(struct_convolution *)malloc(sizeof(struct_convolution));
    memset(&(objSt_convolution->cl_convolution),0
      ,sizeof(objSt_convolution->cl_convolution));
    objSt_convolution->cl_convolution_init = 0;
  }

  // codelete initialization only once, at first invocation
  if(! objSt_convolution->cl_convolution_init )
  {
    objSt_convolution->cl_convolution.where =0|STARPU_CPU|STARPU_CUDA;
    objSt_convolution->cl_convolution.cpu_funcs[0]=conv_cpu_wrapper;
    objSt_convolution->cl_convolution.cpu_funcs[1]=NULL;
    objSt_convolution->cl_convolution.cuda_funcs[0]=conv_gpu_wrapper;
    objSt_convolution->cl_convolution.cuda_funcs[1]=NULL;

    objSt_convolution->cl_convolution.nbuffers = 2;
    objSt_convolution->cl_convolution.modes[0] = STARPU_R;
    objSt_convolution->cl_convolution.modes[1] = STARPU_W;

    objSt_convolution->cl_convolution_init = 1;
  }

  starpu_data_handle_t arr_handle1, arr_handle2;

  starpu_vector_data_register(  &arr_handle1, 0,
    (uintptr_t)A, n*m , sizeof(A[0] ));
  starpu_vector_data_register(  &arr_handle2, 0,
    (uintptr_t)Anew, n*m, sizeof(Anew[0] ));

  struct starpu_task *task = starpu_task_create();

  task->synchronous = 1;

  task->cl = &(objSt_convolution->cl_convolution);

  task->handles[0] = arr_handle1;
  task->handles[1] = arr_handle2;

  task->cl_arg = &arg_convolution;
  task->cl_arg_size = sizeof(ROA_convolution);

  /* execute the task on any eligible computational ressource */
  int ret = starpu_task_submit(task);

  starpu_data_unregister(arr_handle1);
  starpu_data_unregister(arr_handle2);

}
```

Listing 6.12: StarPU Code

6.2.3.6 Comparison to StarPU

The mechanism employed by StarPU requires significantly more (for the example, more than 20 times) coding than VectorPU if we only consider the code to use both VectorPU and StarPU, although both of them are using a run-time approach. As shown in the Listing 6.10 for the VectorPU case and Listing 6.12 for the StarPU case, VectorPU requires 2 lines of code for annotations, while StarPU requires 43 lines of code. Here we neglect kernel code size, only compare the code or annotations to use StarPU and VectorPU. course, one can argue that StarPU offers data-flow driven scheduling by the significant annotation overhead, which VectorPU does not. This is true. However, VectorPU is advantageous on a category of applications where kernels are large with strong dependencies on each other and the parallelism mainly exists inside each kernel code. such applications, scheduling will not play a key role, and using VectorPU can be significantly advantageous in programmability. There are important applications in this category, such as deep learning applications, where there is massive parallelism within its forward and backward passes, while the two passes must run in order, thus there is little freedom for scheduling such component calls.

Another important difference between StarPU and VectorPU is that VectorPU allows skeleton programming[5] that is popular in HPC domain. Skeletons are higher-order functions to express common data access patterns, such as map, reduce etc. Such skeletons can compose different kernels to express different computations. VectorPU's α signature is particularly important in this context. Depending on which kernels these higher order functions compose, the access properties change, thus the annotations must happen at each call to these higher-order functions, an example is demonstrated in Listing 6.8. As far as we know, there is no other frameworks offers such level of flexibility, including StarPU.

Some other difference is that StarPU's data structure is C-style, while VectorPU's data structures are C++-style, and more specifically STL-style. In this way, VectorPU encourages and amplifies STL programming to the CPU/GPU heterogeneous systems, thus allows programmers get many data structure functions for free, such as size(), clear() etc. VectorPU also allows seamless integration with STL and Thrust algorithms, so that programmers get even more functionalities for free as components to build their applications.

[5]Skeleton programming is an important category of programming models in HPC domain, because it abstracts the low level parallelization details away and has wide-spread expressive power. There are numerous skeleton programming models in HPC domain, including SkePU, SkelCL [116], MueSLi [39], Fastflow [46], Marrow [88], Lift [117], Google's Mapreduce [34] etc.

6.2.4 To VectorPU or not to VectorPU?

One may ask in a practical sense when programmers should use VectorPU's vectors and annotations. Due to the idea-proving nature of this prototype implementation, VectorPU's programming model may be further improved. We perform an analysis for this question based on the current VectorPU implementation.

"*Annotation*s are structured information added to program source code" [100]. VectorPU's annotations such as `R`, `W`, and `GW` are structured for its vectors and affect the underlying code generation, even though the code generation is performed by macro expansion instead of a separate compiler. So they conform to the definitions of "annotation". On the other hand, these annotations can be equivalently viewed as *operator*s, as they transform a VectorPU `vector` to some other forms. For example, `R` transforms a VectorPU `vector` to a pointer with minimal coherence actions required to guarantee program execution correctness, and `RI` transforms it to a `const` iterator with the same coherence actions.

The transform starts from a VectorPU `vector`, because such a vector carries information of the size of memory that should be managed for coherence as a whole. Compared to the units of other hardware mechanisms, like page in Nivida's Unified Memory and cache line in cache systems, the VectorPU's unit is a VectorPU's vector defined by programmers, and thus is variable and more naturally reflects the different coherence units needed for different programs, and thus it is also more efficient. The target form of VectorPU's operators are either a pointer or an iterator. A pointer is the most fundamental data type to guarantee the generality of VectorPU. In addition, the target form can also be an iterator to integrate with high level programming libraries, such as STL and Thrust, for productivity.

Now we introduce two guidelines of programming with VectorPU: one guarantees the correctness of program executions, and one focuses on the efficiency of program executions.

6.2.4.1 VectorPU Programming Guideline For Correctness

The first guideline requires to understand when to use VectorPU's annotations safely, which means to understand which cases these annotations can express safely. We consider three dimensions to organize different cases to access data, as shown in Table 6.3. The first dimension is the control flow uncertainty, represented as "may" or "must". A "must" access is a unconditional access, while a "may" access is a conditional one depending on its run-time conditions, as in the example shown in Listing 6.13. The second dimension is the access mode, here we only consider read and write, since read-write is only a shorthand for "write after read", thus can be expressed by a combination of the basic access modes read and write. Since host memory and device memory are symmetric in this context, we only show an analysis on host memory. The same analysis applies for the device memory. The last dimension is the access granularity represented as "full" or "partial". Although a VectorPU `vector` specifies the unit of coherence, it may not always be accessed on every elements in the vector, which results in a partial access to it.

```
void f(float *x)
{
   //conditional read
   if(...)
     for(...)
       ...=x[...]

   //conditional write
   if(...)
     for(...)
       x[...]=...
}
```

Listing 6.13: Conditional read and write

For the cases of read accesses, the rules to use VectorPU annotations are simple, no matter what values are in the other two dimensions, we use `R` uniformly. However, only the Case 1 in Table 6.3 causes no definite performance penalties as `R` is initially designed for this case.

For the cases of write accesses, the only case where we can safely use `W` is the Case 5, when we know that we must write a full VectorPU `vector`. For the Case 6, if we only know that we may write the full vector, we need `RW` for safety instead of `W`. Let us abbreviate "valid" as `V`, and "invalid" as `I`, then three possible coherence states for a VectorPU `vector` exist before the full vector may be written: `(V,V)`, `(V,I)` and `(I,V)`, where the first element in the pair denotes the coherence state of a vector's host memory copy, and the second denotes that of its device memory copy. Note that `R`, `W` and `RW` only apply to host memory, which is the first element in the pair. For the coherence states like `(V,V)` and `(V,I)`, `W` is enough, as no matter the write access happens or not the coherent state on host memory should always be valid and preserved by `W`. But for the state `(I,V)`, using `W` when the host memory copy is not written will flip the state `I` and `V`, and result to be `(V,I)` which leads to the wrong coherence state and data loss. Using `RW` in this case can fix the issue at the cost of a performance penalty by redundant data transfer.

For Case 7, if partial data is written on the host memory copy while the device memory copy holds the most recent data in the vector, a single `W` will mark the whole device memory copy as invalid but only write parts of the host memory copy, this leads to loss of data for the part that are not written on the host memory. For Case 8, if the host memory is even not partially written, the whole data is lost.

Case no.	Control flow uncertainty	Access mode	Access granularity	VectorPU annotation
1	must	read	full	R
2	may	read	full	R
3	must	read	partially	R
4	may	read	partially	R
5	must	write	full	W
6	may	write	full	RW
7	must	write	partially	RW
8	may	write	partially	RW

Table 6.3: Expressing different cases using VectorPU annotations

6.2.4.2 VectorPU Programming Guideline For Efficiency

Using VectorPU efficiently can be achieved by transforming the cases in Table 6.3 where VectorPU's annotations may lead to a performance penalty to those that will not. Two sources that lead to the inefficiency are control flow uncertainty and partial access.

For control flow uncertainty, we could transform "may" cases to "must" cases by refactoring the conditionals out of a component. Listing 6.14 shows the refactored code using VectorPU from the basic C code in Listing 6.13. Line 15 shows that when an execution hit that line, we know we "must" read the array instead of a "may" read case. Using the corresponding annotations in Cases 1 and 5 in Table 6.3 guarantees no redundant data transfer, and thus ensures efficiency.

```
 1 void f_r(float *x){
 2      for(...)
 3        ...=x[...]
 4 }
 5
 6 void f_w(float *x){
 7      for(...)
 8        x[...]=...
 9 }
10
11 void f(vectorpu::vector &x)
12 {
13    //conditional read
14    if(...)
15       f_r(R(x));
16
17    //conditional write
18    if(...)
19       f_w(W(x));
20 }
```

Listing 6.14: Transforming components with conditional access in Listing 6.13 into ones without them.

For partial access, it might be hard in general to guarantee efficiency by ensuring no redundant data transfer[6]. For some cases, we could transform the partial-access cases into full-access ones. For example, if a partial access covers a continuous segment of a full vector, we could initialize a `pvector` (see Section 6.2.2.8) on the full vector, and use annotation on the `pvector` instead. Although the access the `vector` is partial, the access to the `pvector` is full. The creation of a new `pvector` transforms a partial access to a full access, and such creation only involves initializing two pointers thus its overhead is low. In the future, more specialized VectorPU vector types can be designed to cover more partial access patterns used in real applications.

6.2.5 Summary and Future Work

Data movement is usually the main bottleneck for many applications, thus efficient data movement management plays an important role in software optimizations on multicore systems. In this paper we show how VectorPU allows a unified memory view in the programming model, but performs much more efficiently than Nvidia's unified memory, and shows no slow-down comparing to expert-written code with manual coherence. We demonstrated the wide applicability of VectorPU. Although in this paper we used GPU, the annotation language and coherence mechanism also applies to other computing systems with the need of keeping multiple data copies and software-managed coherence. Future work includes more performance evaluation and extending the annotation language to also manage the coherence between the shared memory and the global memory on a GPU.

Section Acknowledgements

We thank NSC (National Supercomputer Center at Linköping University) and University of Tromsø for letting us use their supercomputer and workstation for part of our experiments. We also thank Ludovic Henrio from Inria, France, for constructive discussions which forms the basis of discussions of VectorPU programming guide. We thank Wen-mei Hwu from University of Illinois and David Kirk from Nvidia for their constructive comments on this work. We finally thank Timothy Lanfear at Nvidia for suggestions of benchmarks to use.

6.3 Lazy Allocation and Transfer Fusion Optimization

As mentioned earlier, for GPU-based heterogeneous systems, data transfer can be a more performance-constraining factor as data transfer on PCIe bus can be even slower than data transfer through memory hierarchy on

[6]Sometimes it might even be better for efficiency to transfer redundant data, see Section 6.3.2.

sequential or multi-CPU systems. Therefore, techniques that improve PCIe data transfer efficiency are important to achieve better performance on GPU systems.

In this section, we propose two techniques to improve data transfer efficiency on GPU systems: *lazy allocation* and *transfer fusion optimization*. The former delays dense memory allocations when memory allocation instructions are met until a kernel invocation is encountered in a program, when we know which set of array variables are used together as operands. Then we allocate those array variables of the same kind (host or device) together with one continuous memory chunk, and they can be transferred at once instead of one by one. Merged data transfer yields performance gains over the case that they are transferred separately one by one for two main reasons: less overhead and higher transfer throughput by larger payload. The latter technique merges data transfers of array variables if they are already allocated and their distance is close enough. In order for correct execution we might need to backup the data between array variables, but if they are close enough, the benefit by merging data transfer might be larger than the backup and restore overhead, and a performance increase is gained.

Section 6.3.1 proposes the design of the lazy allocation technique. Section 6.3.2 describes the transfer fusion technique. Section 6.3.3 gives the initial evaluation of the two techniques. Finally Section 6.3.4 concludes.

6.3.1 Lazy Allocation

We illustrate our technique for *lazy allocation* by a simple GPU computing program with pseudo-code in Listing 6.15. In this program, we borrow some syntax from CUDA, and we also implement the technique in CUDA, but the lazy allocation technique itself is not restricted to CUDA and could likewise be used with other types of accelerators and programming models that expose the distributed memory to the programmer.

In Listing 6.15, the program demonstrates the normal steps to implement a ternary vector addition, summing three operand vectors elementwise and storing the results into the third vector. We first allocate three vectors on CPU side and three vectors on GPU side, initialize the three CPU vectors (here we assume the initialization functions of such vectors, such as `init()` in Line 9, do not depend on some volatile global variables), transfer their data to the three GPU memory vectors which initializes them, and finally we invoke the GPU kernel function.

From this example we can see that data are transferred in three separate transfers as there is no guarantee that the three vectors are allocated continuously. On the contrary, if we can guarantee that the three vectors are allocated in one continuous block of memory, we can transfer them at once, which will save us two message overheads, according to the delay model [102]. Measuring merged data transfer alone on this program yields 3.75× speedup against transferring them separately on Laptop A in Table 6.5. For such a guarantee, we need to build up some run-time facilities for such analysis at run time, which will take some overhead. The hope is that the performance benefit of merged allocation and fusion will be larger than the overhead, so we gain a speedup. We propose a technique called lazy allocation, which will overwrite the `malloc` and `cudaMalloc` operations and not perform the allocation until data transfer time or kernel invocation time when the need of continuous memory arrangement for those vectors is known.

```
float *v1 = malloc(50);
float *v2 = malloc(50);
float *v3 = malloc(50);

cudaMalloc( &g_v1, 50 );
cudaMalloc( &g_v2, 50 );
cudaMalloc( &g_v3, 50 );

init(v1);
init(v2);
init(v3);

cudaMemcpy( v1, g_v1, cudaMemcpyHostToDevice );
cudaMemcpy( v2, g_v2, cudaMemcpyHostToDevice );
cudaMemcpy( v3, g_v3, cudaMemcpyHostToDevice );

vector_add3( g_v1, g_v2, g_v3 );
```

Listing 6.15: Pseudo-code for a typical GPU program for vector addition

The main challenge for implementing lazy allocation is that there are more than one steps from allocation points to data transfer points or invocation points. This determines that a simple temporary data structure will not be able to record the multiple previous actions until the merging allocation requirement for vectors is known. Thus we design a set of data structures for such purpose.

The main design constraint is the efficiency of the run-time analysis on the set of data structures. It must be efficient enough so that the run-time overhead is less than the benefit obtained by merging the data transfers, summed with the possible benefit of merging data allocation cost.

On the other hand, delaying the allocation also deprives the rights of compilers to arrange the arrays to possibly utilize locality if such arrays are statically allocated, but considering the expensive cost of PCIe data transfer, favoring PCIe data transfer at the cost of CPU side data locality is reasonable. Furthermore, most of the (large) arrays are dynamic arrays that compilers can not know in advance, which further legitimates our technique.

Next, we discuss our design of the auxiliary data structures for lazy allocation analysis and the process of the run-time analysis.

First we discuss two sub-problems that need to be solved for our lazy allocation technique. The first problem is the run-time overhead constraint. As the run-time overhead should be as low as possible, we choose static array allocation when an array is needed, which is less flexible as we can not change its capacity at run-time, but it is allocated on stack which runs much faster than dynamic arrays that are allocated on heaps. We could reasonably assume that there exists a limit for the amount of information that we want to store, like the maximum number of variables of array type in a program, thus we could fix the array length and use static arrays as our data structure type. There is a limit on the maximum number of formal variables that a function could take in C (128) or C++ (256), but this number is usually too large for good programming practices. So we could assume a reasonably small number as the capacity of our static auxiliary arrays.

The second problem is how to map a variable name to a static array index at run-time for run-time analysis. We consider the beauty of syntax to be less important, as our primary concern is to design a working implementation of lazy allocation and investigate its performance benefit. Thus we directly declare a variable name as an index number instead of a string.

From Listing 6.15 we can see that there are four main stages for a typical CUDA call, namely memory allocation, initialization, transfer and kernel invocation. For multi-calls some of the stages may be optional as some of the variables are reused. This work serves as an exploratory study for a more mature future design of this technique, thus we only consider one function call at present. We could use four static arrays to represent the activities for all variables at each stage. A variable name is represented by an index into these static arrays, allowing easy addressing of information for each variable. Table 6.4 shows the four static arrays used in our lazy allocation technique. The first two arrays serve as implicit key-value pairs, where the key is the variable name represented by its index in those arrays. The other two arrays serve as containers that store the variable names (indexes) involved in the process that each array represents. In the Example column of Table 6.4, the first example shows that three CPU memory variables are allocated for 50 elements each, their variable names are their indices: `0`, `1`, `2`. Then three GPU memory variables are allocated also for 50 elements each; their indices start from the second half, which are `10` (assuming the maximum number of variables on one side, either CPU or GPU, is `10`), `11`, `12`. The second row shows that variables `0`, `1` and `2` should be initialized by `1`, `3` and `5` respectively. The third row shows three data transfers: from variable `0` to `10`, from variable `1` to `11` and from variable `2` to `12`. Finally, the last row shows an invocation that passes variables `10` (the GPU memory variable), `11` and `12` as (kernel) function arguments.

Table 6.4: Data structures for lazy allocation, MAX_NUM_VARS: configurable maximum number of variables in a function

Array name	Array content	Array size	Description	Example
Allocation	allocation size	2*MAX_NUM_VARS	first half for CPU memory, second half for GPU memory	[50,50,50,..., 50,50,50,...]
Initialization	value for initialization	MAX_NUM_VARS	only for CPU memory	[1,3,5,...]
Transfer	source and destination variables	2*MAX_NUM_VARS	first half for transfer src, second half for transfer dst	[0,1,2,..., 10,11,12,...]
Invocation	variables used for invocation	MAX_NUM_VARS	only for GPU memory	[10,11,12,...]

```
vectorpu::action_recorder<float> x;

x.cpu_alloc(0,50);
x.cpu_alloc(1,50);
x.cpu_alloc(2,50);
x.gpu_alloc(10,50);
x.gpu_alloc(11,50);
x.gpu_alloc(12,50);

x.cpu_init(0,1);
x.cpu_init(1,3);
x.cpu_init(2,5);

x.transfer(0,10);
x.transfer(1,11);
x.transfer(2,12);

x.call( vector_add3, 1, 50, 10, 11, 12 );
```

Listing 6.16: Example code for vector addition with lazy allocation

The main process of lazy allocation is straight-forward: from the transfer array we can see which arrays from CPU side and GPU side are about to be transferred together, then we can decide if all the variables that form the transfer source should be preferably allocated together, and the same for all the variables that form the destination. Furthermore, from the table the bindings of variables are determined, but the layout or the order that variables should be arranged in are not fixed. For the convenience of implementation, we use the order from the `invocation` array as the layout of the binding for the destination variables, the layout source variables are determined accordingly. In this way, we can use variadic templates to automatically extract arguments and expand them as argument list to pass to the target function of the invocation. After this decision is made, we can fetch from the `allocation` table the size to allocate, from the `initialization` table how to initialize array variables, and transfer all these source arrays to their destination arrays at once.

An example of the vector addition program tailored for lazy allocation is shown in Listing 6.16. The first line initializes an action recorder which will only record actions for floating point numbers until an invocation is met. Lines 3-8 record actions that allocate 3 CPU memory chunks (variable name `0`,`1`,`2` with `50` elements each), and 3 GPU memory chunks. Lines 10-12 record actions that initialize three CPU array variables (variable name `0`,`1`,`2`) with specified values. Lines 14-16 record three data transfers, each with a source and a destination variable, e.g., from variable `0` to `10`. Until now, nothing is performed but recording. The last line records the function pointer for the invocation target, the kernel invocation configuration (number of blocks and block size), and variables used as arguments. More interestingly, the last line also performs the lazy allocation analysis to get allocation decisions, performs the allocation, initialization, and merged data transfer, and passes the appropriate pointers to the kernel function pointer and invokes it. The code in Listing 6.16 may also be generated by source-to-source compilation from normal CUDA programs.

6.3.2 Transfer Fusion Optimization

First we define a *partial_vector* or *pvector* to be a continuous memory segment of an existing vector container. A *vector* is a STL-like generic data container for elements stored continuously in memory. A pvector is useful in denoting segments used as an operand in memory coherence scenarios. For example, in a tiled matrix multiplication computation, a matrix may be represented by a 1D array, each tile consists of a small matrix which is further decomposed as a few pvectors, one per each line in its 2D interpretation. Consider that such a computation could be performed on both CPUs and GPUs, there might be a need for partial coherence (copying part of the vector or `pvector`s back and forth)

In contrast to kernel fusion optimizations, *transfer fusion optimization* (*TFO*) merges several data transfers of multiple `pvector`s into one, which can be beneficial too. Consider the scenario where there are multiple `pvector`s initialized on the same `vector`. If they all share the same state (stale or most recent), whether to perform coherent data transfer on them one by one or to group them as one data transfer is a run-time optimization choice. For any two `pvectors`, if we merge them as one data transfer, on one hand we save the cost of one communication initialization; on the other hand we pay the cost of transferring more data between the two `pvector`s if not continuous, as one data transfer only applies to a continuous chunk of memory. We also pay the cost to backup these values in between on the data transfer target memory side and recover those values after the data transfer, since these values are not supposed to be transferred in the normal settings.

Although it is a run-time choice, we can derive a formula statically for such decisions which takes the run-time information as its input.

Consider that the cost of transferring n extra bytes of data is $C_{tr}(n)$, and the cost of backup and restoration for these values is $C_{bk}(n)$. The two costs are monotonically increasing functions in the data size n between two arbitrary `pvectors`. The saving by merging data transfer is the time for communication initialization and usually constant, we denote it as S_{init}.

$$C_{bk}(n) + C_{tr}(n) < S_{init} \tag{6.1}$$

Thus, as long as Inequality 6.1 holds, we can gain performance by merging two `pvectors`. It is obvious that if two `pvectors` are adjacent to each other, then Inequality 6.1 always holds, as $C_{bk}(0) = 0$ and $C_{tr}(0) = 0$ and the TFO should always be performed.

By micro-benchmarking on a target system we can derive the limit problem size L_{max}. Since $C_{bk}(n)$ and $C_{tr}(n)$ are assumed to be monotonically increasing, there will be only one L_{max} obtained by micro-benchmarking. Thus, as long as the data size between two `pvectors` is smaller than L_{max}, it is worthwhile to perform the TFO.

Now the general problem is: if we have p `pvectors`, how to choose which `pvectors` should be fused and which should not? The *Greedy-is-good Lemma* 6.3.1 yields a method to derive the optimal solution:

Lemma 6.3.1. *Given a sequence V of* `pvectors` *$\{v_1, v_2, ..., v_p\}$ sorted by memory addresses, merging adjacent* `pvectors` *in a greedy way from either direction whenever it satisfies Inequality 6.1 will lead to an optimal solution.*

Proof: By contradiction. Let us assume that Lemma 6.3.1 does not hold. Then there exists an optimal solution that chooses some elements in V, denoted as S_{opt}: $\{..., v_k, ..., v_m, ...\}$, where at least one v_i that is immediately after v_k or immediately before v_m in V exists that is not selected in S_{opt}, and either $\{v_k, v_i\}$ or $\{v_i, v_m\}$ satisfies Inequality 6.1.

If $\{v_k, v_i\}$ satisfies Inequality 6.1, we can merge them to obtain a better solution, thus S_{opt} is not optimal, a contradiction. The same holds for the case that $\{v_i, v_m\}$ satisfies Inequality 6.1. By contradiction we prove that Lemma 6.3.1 holds. □

As long as the value L_{max} is determined statically, the code to decide transfer fusion is simple and only consists of a greedy scan of all `pvectors`. We assume that the sorting of `pvectors` is done by the programmer via a user-friendly API. The number of `pvectors` is usually much smaller than the number of elements in a `vector`, thus we can assume a nearly constant time complexity for computing TFO decisions, and thus it is suitable to be performed at run-time.

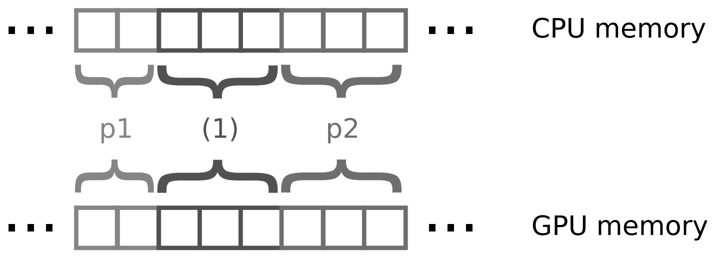


Figure 6.3: Two `pvectors` on a `vector`

6.3.2.1 State-Aware Analysis

Furthermore, in some cases the overhead and thus the threshold to apply TFO becomes lower than that in Inequality 6.1, or a TFO can bring additional benefits. Figure 6.3 shows a general case where two non-adjacent `pvectors` (`p1` and `p2`) are initialized on a `vector` with a memory chunk (1) in between. Suppose that we want to fuse the data transfers for `p1` and `p2` from the GPU memory to the CPU memory into one single coherence (transfer) action. Depending on the coherence states of the memory chunk (1) in the CPU and GPU memory, we have several cases as detailed in the following. The same analysis also applies to the other transfer direction.

1. If the coherence states of the memory chunk (1) in the CPU and GPU memory are the same, then there is no need to backup and restore the original memory values in (1). If both of them are valid and shared, then the values are the same; they can not be all invalid, as at least one copy should be valid. Thus $C_{bk} = 0$ in Inequality 6.1, and L_{max} becomes larger. This lowers the threshold to apply TFO.

2. Otherwise,

   (a) If the coherence state of the memory chunk (1) on the CPU memory is invalid and the other is valid, then there is no need to backup and restore its values, and $C_{bk} = 0$. Furthermore, the TFO has additional benefit: it brings the most recent data to the CPU side. If later (1) is used on the CPU side, the coherence is already done as a side effect of the TFO and thus considered for free.

   (b) Otherwise, we have the only case where we need to backup and restore the values in (1).

Thus a more efficient TFO algorithm can be designed accordingly, taking the coherence information into consideration.

Table 6.5: Machine configuration

Machine	**CPU**	**GPU**	**CUDA**
Laptop A	Intel(R) Core(TM) i7-4710MQ @ 2.50GHz	1 K2100M (Kepler)	7.5
Server B (supercomputer node)	Intel(R) Xeon(R) E5-2660 0 @ 2.20GHz	1 K20Xm (Kepler)	7.5

Table 6.6: Times and speedup on Laptop A for Allocation and Transfer in Ternary Vector Add

Elements	**Baseline**	**Optimized**	**Speedup**
50	43.203 μs	13.131 μs	3.307
100	43.976 μs	234.816 μs	0.187
1,000	260.623 μs	232.151 μs	1.123
10,000	316.461 μs	256.257 μs	1.235
100,000	1009.43 μs	696.911 μs	1.448
1,000,000	5510.280 μs	4295.785 μs	1.283
10,000,000	35246.25 μs	35190.65 μs	1.002

6.3.3 Evaluation

6.3.3.1 Machine configuration

The machines that we use for evaluation are listed in Table 6.5, and MeterPU [83] is used for time measurement.

6.3.3.2 Lazy Allocation

We measure the time to execute the whole code in Listing 6.15 and 6.16. Due to the abnormally expensive first call to `cudaMalloc()`, we put a dummy `cudaMalloc()` before time measurement. We use `O2` for the `nvcc` compiler optimization level. We run 1000 times and report the median: 69.807 μs obtained for the normal case and 24.46 μs for lazy allocation, thus 2.85$\times$ speedup is achieved.

Table 6.6 and the corresponding diagram in Figure 6.4a show, for the ternary vector add example of Section 6.3.1, the time for allocation and transfer (but not kernel invocation) in microseconds on Laptop A for different problem sizes (50, 100, 1000, 10000, 100000, 1M, 10M) and the resulting speedup by transfer fusion based on lazy allocation. Note the logarithmic scale on the vertical axis. The shown values are the median of 100 runs for each problem size, in order to deal with variation in time, yet the variation is fairly low, especially for the larger problem sizes. Results show that our optimization is (almost) always beneficial, even for fairly large problem sizes, e.g. still 28% performance improvement for 1 million elements as can be seen in Table 6.6. We observed one anomaly, for problem size 100, which occurs for both platforms; we assume that this phenomenon is based on CUDA

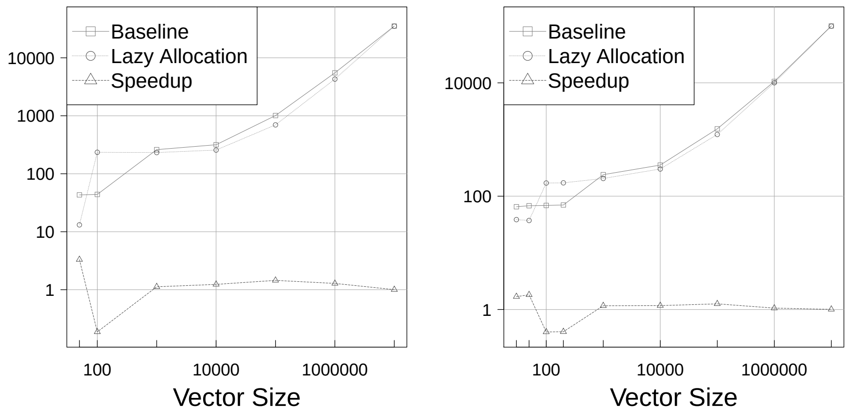
Table 6.7: Times and speedup on Server B for Allocation and Transfer in Ternary Vector Add

Elements	**Baseline**	**Optimized**	**Speedup**
30	65.112 μs	38.623 μs	1.69
50	67.8715 μs	37.2895 μs	1.82
100	68.704 μs	170.905 μs	0.402
200	69.941 μs	172.106 μs	0.406
1,000	239.722 μs	205.617 μs	1.166
10,000	354.166 μs	301.9295 μs	1.173
100,000	1543.385 μs	1229.525 μs	1.26
1,000,000	10589.4 μs	10032.3 μs	1.06
10,000,000	100928.5 μs	100104.5 μs	1.008

internally switching between different allocation mechanisms depending on data size; the fused version exceeds this threshold towards the slower mechanism already for slightly smaller problem sizes than the non-fused one. The observed behavior on Server B is very similar, see Table 6.7 and Figure 6.4b.

In order to further investigate where the speedup comes from for the lazy allocation technique, we measure the time separately for the four main stages (host and device memory allocation, host memory initialization and data transfer from host memory to device memory) of the lazy allocation technique. Figure 6.5 shows that for the case of 100K, the speedup comes from three stages: host and device memory allocation, and data transfer. The biggest time saving is from the device memory allocation, secondly the data transfer. The speedup on host memory allocation, although high, only contributes insignificantly to the total speedup as the time spent on host memory allocation is too small compared to the time spent on device memory allocation and data transfer. However, since the time savings on the device memory allocation and data transfer are non-trivial compared to the total execution time of the whole four phases, a decent overall speedup (1.448×) is achieved. For the 10M case in Figure 6.5, there are still similar speedups compared to the 100K case on the device memory allocation, but there is no noticeable speedup on the data transfer, since the time for the data transfer time and the host memory initialization dominate the total time, thus there is only 0.2% improvement on the total execution time.

Furthermore, Figure 6.6 shows the screenshots from Nvidia's Visual Profiler on the two cases in Figure 6.5 on the data transfer stage. The brown bars in the screenshots are the time period for the data transfer from host memory to device memory in the two cases of Figure 6.5. The three successive brown bars on the left part of each screenshot show the baseline of each case of Figure 6.5 where the three data transfers are not fused, and the single and bigger brown bar on the right part of each screenshot shows the fused data transfer of the three separate data transfers on its left. We can see that when the data size is at 100K, the latency of data transfer still



(a) Execution times on Laptop A (b) Execution times on Server B

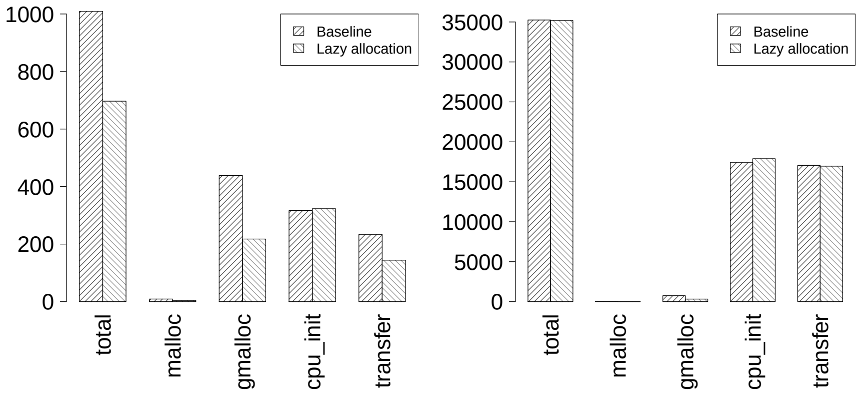
Figure 6.4: Execution times on Laptop A and Server B for allocation and transfer in microseconds for the baseline CUDA (red) and Lazy Allocation (blue) versions of elementwise ternary vector addition, and the resulting speedup (purple curve), for vector sizes ranging from 50 to 10 million. Note the logarithmic vertical axis.

takes a non-trivial portion of total data transfer time, thus merging those data transfers is still quite profitable. When the data size is large (10M), such benefits of merging data transfers become trivial compared to the total time of the four stages.

6.3.3.3 Transfer Fusion Optimization

In order to test the validity of the Transfer Fusion Optimization (TFO), we design a micro-benchmark to derive the L_{max} value in Inequality 6.1 by binary search sampling on different machines, and implement a TFO algorithm that can fuse data transfers for arbitrary number of `pvectors`. The L_{max} value differs depending on the data transfer direction (host to device or device to host) and on which of the conditions $C_{bk} = 0$ or $C_{bk} > 0$ holds as described in different cases in Section 6.3.2.1. In this micro-benchmark we consider the data transfer direction from device to host, and $C_{bk} = 0$ (as it holds for the majority of cases in the analysis in Section 6.3.2.1, which may not necessarily be the majority of cases in practice). We observe that L_{max} is constant in the lengths of the `pvectors`; the value of L_{max} is different on different machines (on Laptop A, it is 10528 (integers) while 4473 (integers) on Server B, where integers are 4 bytes large on both systems).

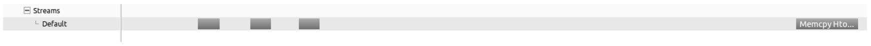
We then use 4 `pvectors` of equal size (2000 integers), keep the lengths of the gaps between these `pvectors` equal and gradually increase the gap lengths. As shown in Figure 6.7, below the L_{max} value on each machine we



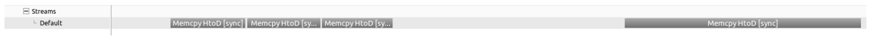
(a) Breakdown of time for lazy allocation of size 100K in Table 6.6

(b) Breakdown of time for lazy allocation of size 10M in Table 6.6

Figure 6.5: Breakdown of time for the main stages in lazy allocation on Laptop A. (total: total execution time, malloc: time for memory allocation on the host memory, gmalloc: time for memory allocation on device memory, cpu_init: time for host memory initialization, transfer: time for data transfer from host memory to device memory, y-axis unit: μs)

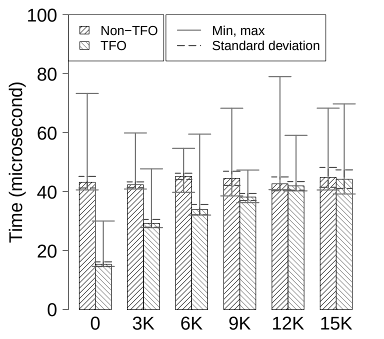


(a) Profiling for data transfer of size 100K in Table 6.6

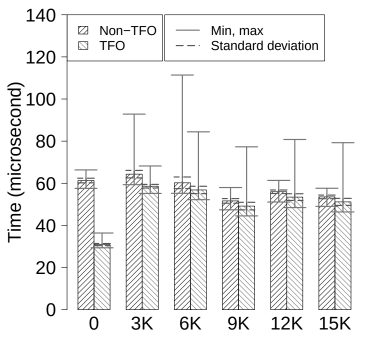


(b) Profiling for data transfer of size 10M in Table 6.6

Figure 6.6: Screenshot from Nvidia Visual Profiler for data transfer scenarios in Figure 6.5 on Laptop A



(a) Speedup on Laptop A: 1.01-2.8× (L_{max} = 10528 integers)



(b) Speedup on Server B: 1.05-1.98× (L_{max} = 4473 integers)

Figure 6.7: TFO microbenchmark speedups on 2 systems. The x-axis labels show gap lengths between `pvectors`.

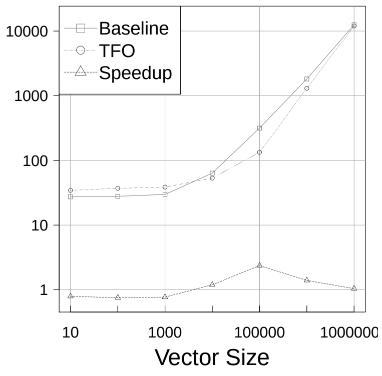
observe significant speedups, while above these values the TFO algorithm did not fuse any data transfers thus no speedup is observed. The overhead in this case is negligible, which implies that the TFO may be switched on all the time.

For the two platforms Laptop A and Server B respectively, Figures 6.8a and 6.8b show the allocation and transfer times and speedup for the transfer fusion optimization of Section 3, applied to three pvectors of exponentially increasing size separated with gaps of size 10 integers; here, the backup and restore cost is included in the measurements. On both platforms, the observed pattern is the same; in absolute times every setting runs slower on Server B because the transfer latency is larger on Server B.

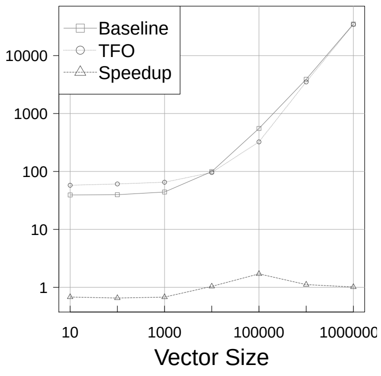
Figures 6.9a and 6.9b show the corresponding times and speedup with backup and restore cost excluded for the same experiment. This experiment models scenarios where backup and restore is not necessary as the transmitted gap contents is valid for the coherence protocol. Comparing the times and speedups to Figures 6.8a and 6.8b we see that backup and restore cost accounts for a significant overhead in TFO and that the effect of TFO increases by up to one order of magnitude for small pvectors whenever backup and restore is not required. For medium-sized vectors (around 100000 elements) the backup/restore overhead is insignificant and we still see considerable speedup.

6.3.4 Summary and Future Work

We demonstrated the feasibility of two optimization techniques for data transfer over PCIe bus for GPU-based heterogeneous systems, and showed that these techniques are possibly beneficial (significant speedups) with ini-

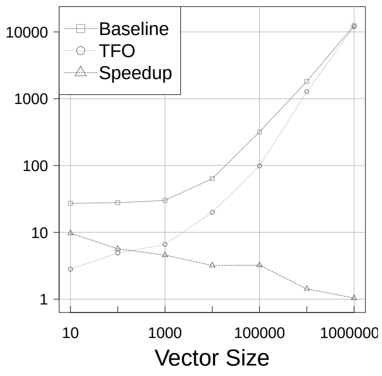


(a) TFO including backup/restore on Laptop A

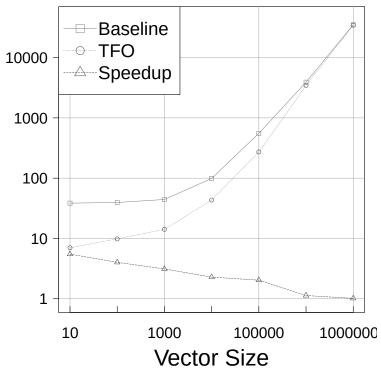


(b) TFO including backup/restore on Server B

Figure 6.8: Execution times on Laptop A (a) and Server B (b) for allocation and transfer in microseconds for the baseline CUDA (red) and Transfer Fusion optimized (blue) versions of a computation using 3 pvectors of varying size, separated by gaps of 10 integers, and the resulting speedup (purple curve). The backup and restore cost is included in the measurements. Note the logarithmic vertical axis.



(a) TFO excluding backup/restore on Laptop A



(b) TFO excluding backup/restore on Server B

Figure 6.9: Execution times on Laptop A (a) and Server B (b) for allocation and transfer in microseconds for the baseline CUDA (red) and Transfer Fusion optimized (blue) versions of a computation using 3 pvectors of varying size, separated by gaps of 10 integers, and the resulting speedup (purple curve). The backup and restore cost is excluded in the measurements. Note the logarithmic vertical axis.

tial experiments. We showed the optimality of transfer fusion optimization. We proposed our initial design of lazy allocation, which can serves as a basis for more mature designs that can work for multiple component invocations.

Our future work includes a thorough evaluation of the two techniques, a more mature design of lazy allocation, exploratory extension of the two techniques into general scratchpad memory transfer optimization, and integration of those techniques into our smart containers [85].

Section Acknowledgements

This work was partly funded by SeRC. We thank NSC (National Supercomputer Center at Linköping University) for letting us use their supercomputer for part of our experiments.

6.4 Related Work

6.4.1 Framework Design

In general, the related work for automatic data transfer management between CPU and GPU consists of three approaches: compiler-only approach, run-time approach and hybrid compiler/runtime approach.

Pure compiler approaches may remove the need to annotate the code in many cases, but can not capture runtime information, and compiler analysis can be imprecise and conservative, which may lead to redundant data transfer, such as OpenMPC [76].

Pure run-time approaches allow to capture runtime information but burden the programmers with annotations. ADSM [45], DyManD [56] and Nvidia's unified memory [75] provides a vector abstraction with automatic coherence management, but the data migration is performed page by page thus the low performance prevents its practical usage. Some works, such as SemCache [2] and SemCache++ [3], are designed for some specific high level constructs, such as matrix, although handy for matrix applications, are less generic than vector abstraction as one can build matrices on top of vectors but not vice versa. Dastgeer et al. [27] supports automatic coherence management in skeleton programming. StarPU [8] uses API calls to inform access properties instead of annotations, thus it requires more LOC (lines of code) for this purpose besides manual allocation of host memory.

Hybrid approaches that combine compiler and runtime techniques can obtain the benefits of both. Works using hybrid approaches are listed as follows: CGCM [57] is the first work that fully automates the data transfer between CPU and GPU without programmer's annotations, it inserts communication code into the source code and runs optimizations of alloca promotion, glue kernel, and map promotion in sequence. X10+AMM [101] removes the redundant data transfers for three cases: non-stale data, eager transfer data and GPU-only data. OpenARC [77] further removes redundant transfers for more situations like user-defined data clause, and dead variables, by using interactive directive-based memory transfer verification to involve programmers iteratively in selecting the optimization proposals of data transfers given by the framework, such interactiveness overcomes some limitations of compiler analysis such as dead variable detection. Ishizaki et al. [55] adds the compiler analysis for sub-array coherence. SAC [36] supports data movement management for its purely functional array programming language.

Comparing to previous work, none of them considered the case that the source code may not be available, such as cuFFT, cuBLAS, cuSPARSE etc, thus the compiler-only approach and hybrid approach can not remove the need of manual annotations. Even compared to the library-based approach, VectorPU formalizes the flow properties at function level as flow signature, thus it allows easy annotation of functions from binary libraries. Furthermore, VectorPU allows flow signature switching which is necessary in cases where run-time arguments change flow properties of a function. Other unique features include smart iterators that let programmers to use STL and Thrust algorithms naturally, and lambda functions enabling coherence management for code snippets in addition to functions. By using VectorPU, it is guaranteed that no redundant data is transferred assuming the annotations provided by programmers are correct. Compared to compiler-only or hybrid approaches, VectorPU enforces a check for the need of data transfer at every use (either read or write) of variables, thus may check at more points than necessary, however, such checks are at the granularity of the whole or sub-array and cheap [101], and show negligible overhead in our examples.

6.4.2 Data Transfer Optimization

Most of previous related work to reduce or hide the memory transfer overhead between CPU and GPU are divided into the following three categories. The first method is to hide data transfer overhead by overlapping data transfer and computation using the streaming feature of CUDA programming model, such as [119, 10]. The second method is to reduce unnecessary data transfers by keeping track of data read and write accesses, either by static analysis such as data flow analysis [78], or by run-time analysis such as [85, 27]. The third approach is to re-design an algorithm so that less memory transfer is needed, such as [4], who designs such a communication-avoiding QR factorization algorithm. Other similar work includes [35].

Data cache optimizations performed by compilers, such as tiling, that change the loop structure to sequence the memory accesses so that some of them can be in the same cache line and those transfers across the memory hierarchy are merged. This approach is more similar to our approach, while it differs with our approach in that compiler transformations are performed at compile time, while our lazy allocation and TFO are performed at run-time without suffering the compiler challenges, e.g., the aliasing and imprecision of data flow analysis. Thus our data structures in Section 6.3.1 can also be viewed as a run-time IR.

6.5 Summary

In this chapter we discussed the complexity of data management on GPU-based systems. We proposed a framework to abstract the complexity away without noticeable run-time overhead, and further proposed two data transfer optimization techniques which can potentially run under the hood.

Chapter 7

Put It All Together: a Case Study

In the previous four chapters we discussed our approaches to tackle different complexities that programmers typically face on GPU-based heterogeneous systems, and demonstrate their benefits. To serve as a foundation or as building blocks, for further research or for practical programming situations, we are interested in how we could combine or integrate those approaches, or more specifically, those frameworks together, by which the benefits of each framework could co-exist and integrating them possibly gives synergies.

These four frameworks are implemented as either libraries (stateless across calls: MeterPU[1], XPDL, TunerPU) or run-time systems (stateful across calls, VectorPU), in the same language (C++). They all exist as header files, and in principle an application could arbitrarily compose them by C++ inclusion: e.g., `"include <meterpu.h>"`. In this chapter, we describe a meaningful integration in detail in Section 7.1, hoping that it can serve as an inspiration for more ones. Then we illustrate this integration by a simple case study of matrix-matrix multiplication in Section 7.2. Section 7.3 evaluates the combined benefits.

7.1 A Meaningful Integration

Starting with a GPU-based system with some native programming modle and software (e.g., CUDA), we could directly start programming on the platform. In addition to program the kernel code, we could manually handle the measurement complexity, non-portability, data management complexity, and hard-code the computation on GPUs, which we usually do as daily programming tasks.

[1]A MeterPU meter is stateful inside one measurement instance, but it is not for multiple measurement instances, so we consider it as stateless.

Instead, we could use different prototypes built based on our approaches described in Chapter 3, 4, 5 and 6, to handle each complexity. Figure 7.1 shows a possible way to combine these approaches together. We describe each kind of interaction as follows:

- XPDL on top of hardware and system software: XPDL is designed to model hardware and system software with the purpose to support high level optimizations, thus this interaction is straightforward. XPDL support modular features, thus one could write a XPDL model for a given platform, and let the XPDL compiler fetch the model's dependent models recursively, from the XPDL central repository. Note that a XPDL model is only needed to be written once, and usable for all applications. It only requires to be revised if hardware or system software changes, which are considered to be less frequent.

- TunerPU, MeterPU and VectorPU on top of XPDL: we could make TunerPU, MeterPU and VectorPU portable based on XPDL compiler translated information, e.g., if no GPUs are found in the XPDL platform model, TunerPU could disable components based on CUDA, MeterPU could disable GPU time and energy meters, and VectorPU could disable memory allocation and coherence on GPUs.

- TunerPU on top of MeterPU and VectorPU: at training time, TunerPU uses VectorPU to manage data transfer necessary for CPU components to execute, and uses MeterPU to measure the run-time of CPU or GPU components.

- Programmers on top of TunerPU, VectorPU and XPDL: programmers write data structures[2] using VectorPU and write implementation variants using the TunerPU framework with a unified view[3] implemented. Optionally, component writers can encode their knowledge for their software components, e.g., portability requirements by binding them to the XPDL API.

- At run-time: TunerPU select which implementation variant to use given context arguments, and VectorPU ensures that the most recent data is transferred to the memory that the particular implementation variant will use as arguments.

Next, we use this integration configuration to perform a matrix-matrix multiplication.

[2]More data structures could be added to VectorPU for programmability and execution efficiency.

[3]See Section 4.4.2

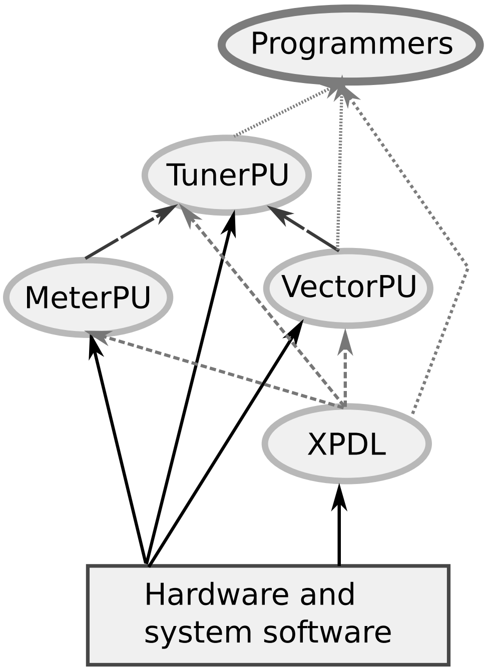


Figure 7.1: A meaningful integration.
Black straight line: interaction by execution.
Red (medium) dashed line: XPDL interactions.
Blue (big) dashed lines: TunerPU interactions.
Purple (small) dashed line: interactions by programmers.
The Arrows here denote the relation ”used by”.

7.2 Case Study: Matrix-matrix Multiplication

In this section, we will describe in detail how to apply the framework integration discussed in Section 7.1 to a concrete case: matrix-matrix multiplication, which is a widely used kernel in HPC. We will illustrate how to combine the benefits of each framework to handle multiple complexities together.

We first show the baseline code used in this case study, then we write a XPDL model, and compile it by our XPDL compiler. Afterwards we connect MeterPU, VectorPU and TunerPU to the translated platform information by the XPDL compiler. Finally we describe the two rewrites of the baseline code, which connect it to the network of these prototypes like VectorPU, TunerPU and XPDL.

7.2.1 Baseline: Manual Code

Our baseline CUDA code is straightforward as shown in Listing 7.1, we briefly explain the code as follows:

- Lines 1-18: the matrix-matrix multiplication component to run on a CUDA-enabled GPU, which is a wrapper to NVIDIA's Cublas library.
- Lines 25-36: declare handles for host and device memory, and calculate their sizes.
- Lines 38-44: memory allocation on host and device memory.
- Lines 46-47: memory initialization on host memory.
- Lines 49-50: transfer the initialized data from host memory to device memory.
- Line 52: GPU component invocation.
- Line 54: transfer the computed data from device memory to host.
- Line 56: verify the correctness of computation
- Lines 59-65: memory de-allocation on host and device memory.

This program can be compiled and executed on a CUDA-enabled GPU.

```
void matrix_mul_cublas(float const * const a,float const * const b,
    float * const c,size_t const ha,size_t const wa,size_t const wb)
{

  const float alf = 1.0f;
  const float bet = 0.0f;
  const float *alpha = &alf;
  const float *beta = &bet;

  cublasStatus_t stat;
  cublasHandle_t handle;

  stat = cublasCreate(&handle);
  assert(stat == CUBLAS_STATUS_SUCCESS);

  cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, static_cast<int>(wb)
    , static_cast<int>(ha), static_cast<int>(wa), alpha, b,
    static_cast<int>(wb), a, static_cast<int>(wa), beta, c,
    static_cast<int>(wb));

  cublasDestroy(handle);
}

int main()
{

  assert( cudaDeviceReset() == cudaSuccess );

  const size_t ha=300, wa=300, wb=300;

  float *a_h, *b_h, *c_h;
  float *a_d, *b_d, *c_d;

  const size_t size_a=ha*wa;
  const size_t size_b=wa*wb;
  const size_t size_c=ha*wb;

  const size_t raw_size_a=ha*wa*sizeof(float);
  const size_t raw_size_b=wa*wb*sizeof(float);
  const size_t raw_size_c=ha*wb*sizeof(float);

  a_h=(float *)malloc( raw_size_a );
  b_h=(float *)malloc( raw_size_b );
  c_h=(float *)malloc( raw_size_c);

  cudaMalloc(&a_d, raw_size_a );
  cudaMalloc(&b_d, raw_size_b );
  cudaMalloc(&c_d, raw_size_c );

  std::fill(a_h, a_h+size_a, 1.0f);
  std::fill(b_h, b_h+size_b, 1.0f);

  cudaMemcpy(a_d, a_h, raw_size_a, cudaMemcpyHostToDevice);
  cudaMemcpy(b_d, b_h, raw_size_b, cudaMemcpyHostToDevice);

  matrix_mul_cublas(a_d, b_d, c_d, ha, wa, wb);

  cudaMemcpy(c_h, c_d, raw_size_c, cudaMemcpyDeviceToHost);

  std::for_each(c_h, c_h+size_c, [](float const i){assert(i==300.0f)
    ;}) ;


  free(a_h);
  free(b_h);
  free(c_h);

  cudaFree(a_d);
  cudaFree(b_d);
  cudaFree(c_d);

  return EXIT_SUCCESS;
}
```

Listing 7.1: Baseline code for matrix-matrix multiplication

7.2.2 Writing a XPDL Model

Instead of writing such a program in Listing 7.1, we could use the approaches described in this thesis. First we can express our target platform as a XPDL model in Listing 7.2.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xpdl:model xmlns:xpdl="http://www.xpdl.com/system" xmlns:xsi="http
    ://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http
    ://www.xpdl.com/system system.xsd">
  <xpdl:component type="system"/>
  <xpdl:cpu id="xpdl_cpu_0" type="Intel_i7_4710MQ"/>
  <xpdl:gpu id="xpdl_gpu_0" type="Nvidia_Quadro_K2100M"/>
  <xpdl:memory id="xpdl_mem00" type="DDR3_32G" />
  <xpdl:software id="xpdl_matlab" type="matlab"></xpdl:software>
  <xpdl:interconnect id="xpdl_interconnect0" type="pcie3" head="
    xpdl_cpu_0" tail="xpdl_gpu_0" />
  <xpdl:interconnect id="xpdl_interconnect1" type="pcie3" head="
    xpdl_gpu_0" tail="xpdl_cpu_0" />
  <xpdl:library id="xpdl_cblas" type="cblas" />
  <xpdl:library id="xpdl_openmp" type="openmp" />
  <xpdl:library id="xpdl_cublas" type="cublas" />
  <xpdl:os id="xpdl_ubuntu" type="ubuntu" />
</xpdl:model>
```

Listing 7.2: The XPDL model for our target platform example

Next we use our XPDL compiler to translate[4] the model into a C++ header file. The header file is a set of C++ meta-functions and macros that implement our XPDL Query API. For the interest of this chapter, the C++ macros are more interesting in connecting XPDL to MeterPU, VectorPU and TunerPU, thus we show the macro part of the header file generated in Listing 7.3.

```cpp
...

#define XPDL_NUM_OF_GPUS 1

#define XPDL_CBLAS 1

#define XPDL_OPENMP 1

#define XPDL_CUBLAS 1

...
```

Listing 7.3: The part of XPDL compiler translated C++ header file

7.2.3 Connecting XPDL to MeterPU and VectorPU

Connecting XPDL to MeterPU and VectorPU is straightforward. By the macros generated by the XPDL compiler, the relevant MeterPU meters and VectorPU vectors can be disabled and enabled accordingly depending on the

[4]The command to use the XPDL compiler to translate a XDPL model with file name `system.xml` is `xpdl_compiler system.xml`

satisfaction of their dependencies, such as the existence of relevant hardware and system software. For example, if no GPUs are equipped in the target platform implied by XPDL, the VectorPU vector is downgraded to barely a STL vector.

7.2.4 Connecting XPDL to TunerPU

TunerPU's tuning is be portable, as it is pure C++ code, the selection is performed at run-time on the CPU side, and every GPU-based heterogeneous platform is equipped with at least one CPU.

For the abstraction class mentioned in Section 4.4.2, it needs some special care besides the implementation variants (which could simply be disabled or enabled depending on the macro interface by XPDL). We use a preprocessing-time vector to model dependencies for each implementation variant shown in line 1 of Listing 7.4. As an example, assuming we have four implementation variants, and each element in line 1 show the each implementation variant's dependency:

- The first implementation variant is the sequential implementation of matrix-matrix multiplication, thus its dependency value is always 1, as the sequential implementation of matrix-matrix multiplication can always can run without dependencies.
- The second is BLAS implementation of matrix-matrix Multiplication, thus we express its dependency as `XPDL_GSL_CBLAS`[5] using XPDL API.
- The third is OpenMP implementation of matrix-matrix Multiplication, thus we express its dependency as `XPDL_OPENMP` using XPDL API
- The fourth is cuBLAS implementation of matrix-matrix Multiplication, thus we express its dependency as `XPDL_CUBLAS` using XPDL API

From this vector we could deduce summary information (line 3) like the total number of implementation variants with their dependencies satisfied, as another macro by a macro call from Boost library[6] [13] which allows to further reshape the abstraction class for portability, such as the size of dispatch table[7] etc.

[5]Here we neglect version number, but in practice one can easily add it.

[6]The Boost library API `BOOST_PP_SEQ_FOLD_LEFT` can sum the number of 1s in a preprocessing-time vector

[7]As the number of implementation variants with their platform dependencies satisfied changes, the size of the dispatch table for dynamically binding a caller to one of these implementation variants needs to change as well.

```
#define DEPENDENCE (1)(XPDL_GSL_CBLAS)(XPDL_OPENMP)(XPDL_CUBLAS)

#define MATRIX_MUL_NUM_VARIANTS BOOST_PP_SEQ_FOLD_LEFT(SUM,
    BOOST_PP_SEQ_HEAD(DEPENDENCE), BOOST_PP_SEQ_TAIL(DEPENDENCE))
```

Listing 7.4: Code snippet to connect XPDL to TunerPU

7.2.5 First Code Rewrite Using VectorPU

After the first code rewrite using VectorPU for the code in Listing 7.1, we obtain the code in Listing 7.5, in which the matrix-matrix multiplication component in Lines 1-18 in Listing 7.1 is not repeated.

We use a VectorPU `vector` and annotations to manage the allocation/deallocation, and memory coherence management without redundant data transfer. We can directly observe a huge code size drop by this rewrite in Listing 7.5 compared to the code in Listing 7.1, which we will evaluate quantitatively in Section 7.3.

```
#include <lib/vectorpu.h>
...
int main()
{
    const size_t ha=300, wa=300, wb=300;
    vectorpu::vector<float> a(wa*ha,1), b(wa*wb,1), c(ha*wb,0);
    matrix_mul_cublas(GR(a), GR(b), GW(c), ha, wa, wb);
    std::for_each(RI(c), REI(c),
                  [](float const i){assert(i==300.0f);});
    return EXIT_SUCCESS;
}
```

Listing 7.5: Code after first rewrite using VectorPU

7.2.6 Second Code Rewrite with TunerPU

Next, we rewrite the code with TunerPU from the code rewritten using VectorPU from Listing 7.5. There are three main steps for this rewrite:

1. First we add more software components, which gives the CPU-GPU selector more freedom to select a most suitable implementation variant to use based on run-time context. Notice that we always provide a sequential CPU implementation with no dependencies, thus it could always run on any target platform. Some of them can be easily written by annotating the sequential code, like OpenMP, and some of them can be easily written by wrappers to high quality libraries, such as BLAS and cuBLAS. The resulting code snippet is shown in Listing 7.6, and we use the XPDL interface and macros to express its dependencies for running on a given platform, e.g., in Line 18.

2. Second, we implement an abstraction class to present a unified view of the *ceset* to the TunerPU generic selector. The resulting code snippet

is shown in Listing 7.7. Lines 6-30 encode the components' writers' knowledge about how to initialize data structures for a component run etc. Notice that we add an extra method `run()` (Lines 41-61) which takes an implement selection decision represented by a index value, and calls the selected implementation variant with the minimum coherence guaranteed by the flow signature. This method is used for production runs, as another interaction between TunerPU and VectorPU. There are some boilerplate code (such as the `for` loop and measurement code in Lines 14-21) in this class, which in the future could be generated automatically.

3. Third, we add several lines (Lines 5-7) of code to configure and train the TunerPU CPU-GPU selector, and then change the invocation (Line 10) of the hard-coded implementation variant to the invocation of TunerPU CPU-GPU selected implementation variant. The resulting code is shown in Listing 7.8.

```
void matrix_mul_cpu(float const * const a,float const * const b,
    float * const c,size_t const ha,size_t const wa,size_t const wb)
{
  for (size_t i = 0; i < ha; ++i)
    for (size_t j = 0; j < wb; ++j)
    {
      double sum = 0;
      for (size_t k = 0; k < wa; ++k)
      {
        double d = a[i * wa + k];
        double e = b[k * wb + j];
        sum += d * e;
      }
      c[i * wb + j] = (float)sum;
    }
}


#if XPDL_GSL_CBLAS == 1

void matrix_mul_blas(float const * const a,float const * const b,
    float * const c,size_t const ha,size_t const wa,size_t const wb)
{
  cblas_sgemm (CblasRowMajor,
      CblasNoTrans, CblasNoTrans, static_cast<int>(ha), static_cast<
    int>(wb), static_cast<int>(wa),
      1.0f, a, static_cast<int>(wa), b, static_cast<int>(wb), 0.0f,
    c, static_cast<int>(wb));
}

#endif

#if XPDL_OPENMP == 1

void matrix_mul_openmp (float const * const a,float const * const b,
    float * const c,size_t const ha,size_t const wa,size_t const wb)
{

  omp_set_dynamic(0);

#pragma omp parallel for num_threads(XPDL_NUM_OF_HW_THREADS)
  for (size_t i = 0; i < ha; ++i)
    for (size_t j = 0; j < wb; ++j)
    {
      double sum = 0;

      for (size_t k = 0; k < wa; ++k)
      {
        double d = a[i * wa + k];
        double e = b[k * wb + j];
        sum += d * e;
      }

      c[i * wb + j] = (float)sum;
    }
}
#endif


#if XPDL_CUBLAS == 1

#include <cuda_runtime.h>
#include <cublas_v2.h>

void matrix_mul_cublas(float const * const a,float const * const b,
    float * const c,size_t const ha,size_t const wa,size_t const wb)
{
  ...
}

#endif
```

Listing 7.6: More components for matrix-matrix multiplication added

```
template <class MeasureType, class ...Tunable_Args>
class matrix_mul_tunable : public tunable<MeasureType,
    matrix_mul_func, Tunable_Args...>{
  public:
    ...

    std::vector<std::tuple<size_t, size_t, MeasureType, Tunable_Args
    ...> > training_run(std::vector<bool> const& variant_mask,
    size_t const repeat_size, size_t const HA, size_t const WA,
    size_t const WB) const{
      ...

      vectorpu::vector<float> A(WA*HA,1), B(WA*WB,1), C(HA*WB,0),
    C_ref(HA*WB,WA);

      ...

      if(variant_mask[i]){
        for(size_t r=0; r<repeat_size; ++r){
          cpu_meter.start();
          //encapsulate the difference of correct invoke
          (*this->dispatch_table[i])(R(A), R(B), W(C), HA, WA, WB);
          cpu_meter.stop();
          cpu_meter.calc();
          val=cpu_meter.get_value();
          results.emplace_back(r,i,val,HA,WA,WB);
        }
      }
      ++i;

#if XPDL_GSL_CBLAS == 1
      ...
#endif

#if XPDL_OPENMP == 1
      ...
#endif

#if XPDL_CUBLAS == 1
      ...
#endif

      ...
    }

    template <class T>
    void run(size_t predicted_index, vectorpu::vector<T> &a,
    vectorpu::vector<T> &b, vectorpu::vector<T> &c, size_t const ha,
    size_t const wa,size_t const wb) const
    {
      size_t i=0;
      if(predicted_index==i)
        (*this->dispatch_table[i])(R(a), R(b), W(c), ha, wa, wb);
      i++;

#if XPDL_GSL_CBLAS == 1
      ...
#endif

#if XPDL_OPENMP == 1
      ...
#endif

#if XPDL_CUBLAS == 1
      if(predicted_index==i)
        (*this->dispatch_table[i])(GR(a), GR(b), GW(c), ha, wa, wb);
#endif
    }

};
```

Listing 7.7: The abstraction function for a unified view of the *ceset* of matrix-matrix multiplication

```
int main()
{
  const size_t ha=300, wa=300, wb=300;

  tuneit::tuneit_setting<MATRIX_MUL_NUM_DIM, MATRIX_MUL_NUM_VARIANTS
    > st{2, std::vector<bool>(4,true), true, false, true, 40,
    {{1,200}, {1,200}, {1,200}} };
  tuneit::tuneit< MATRIX_MUL_NUM_VARIANTS, 8, matrix_mul_tunable<
    float, size_t, size_t, size_t>, float, size_t, size_t, size_t>
    mytuner(st);
  mytuner.train();

  vectorpu::vector<float> a(wa*ha,1), b(wa*wb,1), c(ha*wb,0);
  mytuner.run(mytuner.predict(ha,wa,wb), a, b, c, ha, wa, wb);

  std::for_each(RI(c), REI(c), [](float const i){assert(i==200.0f)
    ;});
  return EXIT_SUCCESS;
}
```

Listing 7.8: Main function after the second rewrite

7.3 Evaluation

Next we quantitatively evaluate the rewritten code that utilize the integrated prototypes, using the normal CUDA code in Listing 7.1 as a baseline. We are interested in programmability, portability, performance portability and energy optimization, which are our main research questions (see Section 1.1).

7.3.1 Programmability

Rewriting the legacy code using the integration of frameworks as described in this chapter requires first writing a XPDL model to describe the target platform under use as in Section 7.2.2, and connecting these frameworks by code in Section 7.2.3 and 7.2.4. Since these efforts are done only once (e.g., the code connecting frameworks) or updated infrequently (e.g., a XPDL model), we do not include the code in these processes in our programmability evaluation.

Thus we are interested in comparing the baseline code (Listing 7.1) with the code from the first rewrite (Listing 7.5) and second rewrite (Listings 7.6, 7.7 and 7.8). We measure the logical lines of code (LOC), where we neglect lines consisting of only `include` or `curly brackets`.

Figure 7.2 shows the quantitative comparison on code size among the baseline code and its rewrites. We could see that for the first rewrite the code size drops significantly, by a factor of 2.47 compared to the base line code, as data allocation/deallocation and data transfers are managed automatically. For the second rewrite, we observe a significant code size increase by a factor of 3.35 compared to the base line code. 21% of the second rewrite code is for adding new implementation variants, which could usually be written conveniently by wrapping existing libraries, such as OpenMP, BLAS

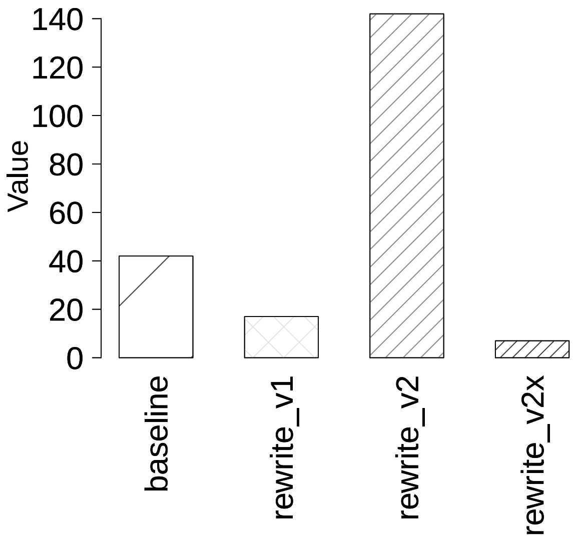


Figure 7.2: Code size comparisons among the baseline and its rewrite versions. (Rewrite_v1: code size for the first rewrite in Listing 7.5, Rewrite_v2: the total sum of the code size from the second rewrite in Listings 7.6, 7.7 and 7.8, Rewrite_v2x: the code size for the ideal solution explained in detail in this section's discussions)

and cuBLAS. 63% of the second rewrite code is for writing the abstraction class encoding component writers' knowledge for our implementation variant selector to use. Together with some coding required to allow to measure the execution time for an arbitrary run-time context, the code size increase is expected, this is a one-time effort and easy to extend to accommodate new implementation variants for the same functionality. The main function increases trivially only to invoke the training process.

If we adopt some ideas of elastic computing [126] for a library of elastic functions, and apply it to our context, where we assume a pre-trained library of components, then we would only need to write the following code in Listing 7.9 (Line 5 construct a tuner by loading its training results), and its code size is shown as `rewrite_v2x` in Figure 7.2 which is the shortest among all versions. This is the ideal case which however requires to write a library of predefined functions (like skeletons in skeleton programming [24, 23]) expressive enough for general computations.

```
1  int main()
2  {
3    const size_t ha=300, wa=300, wb=300;
4    vectorpu::vector<float> a(wa*ha,1), b(wa*wb,1), c(ha*wb,0);
5    tuneit::tuneit matrixmul;
6    matrixmul.run(matrixmul.predict(ha,wa,wb), a, b, c, ha, wa, wb);
7
8    std::for_each(RI(c), REI(c), [](float const i){assert(i==200.0f)
       ;});
9
10   return EXIT_SUCCESS;
11 }
```

Listing 7.9: Ideal code rewrite

7.3.2 Portability

Obviously the baseline code is not portable, as it can only run on CUDA-enabled GPUs. Now we test if the rewritten code could run on a CPU-only platform.

```
<?xml version="1.0" encoding="UTF-8"?>
<xpdl:model xmlns:xpdl="http://www.xpdl.com/system" xmlns:xsi="http
    ://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http
    ://www.xpdl.com/system system.xsd">
  <xpdl:component type="system"/>
  <xpdl:cpu id="xpdl_cpu_0" type="Intel_i7_4710MQ"/>
  <xpdl:memory id="xpdl_mem00" type="DDR3_32G" />
  <xpdl:os id="xpdl_ubuntu" type="ubuntu" />
</xpdl:model>
```

Listing 7.10: The XPDL model for our target platform example

We first write a XPDL model[8] shown in Listing 7.10 to describe a CPU-only machine. Then we compile the model. The key snippet of the generated

[8]The model is written to describe a CPU-only node in the supercomputer Triolith.

code is shown in Listing 7.11. Now, without further source code modification, the rewritten matrix-matrix multiplication runs and yields correct results on the CPU-only machine.

```
...

#define XPDL_NUM_OF_GPUS 0

#define XPDL_CBLAS 0

#define XPDL_OPENMP 0

#define XPDL_CUBLAS 0

...
```

Listing 7.11: Code snippet in the generated code from the model in Listing 7.10 by the XPDL compiler

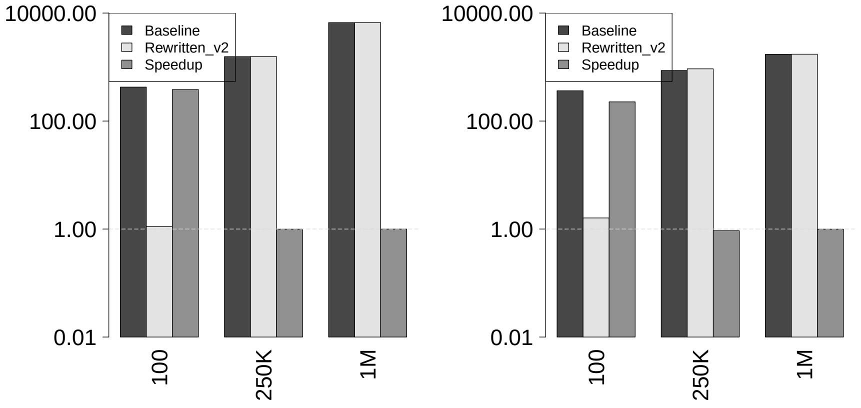
The key for portability is the existence of a sequential CPU implementation variant for a *ceset*. This variant must not have any dependencies except on an appropriate compiler. Every heterogeneous platform is assumed to have at least one CPU that can run sequential code.

One could argue that if a component is written in OpenCL, then it is automatically portable and why do we need the software stack here? The power of implementation variant selection depends on the number and the quality of implementation variants, thus to have a framework that allows to use components in multiple programming models increase the number of available components, and thus increase the potential power of implementation variant selection.

7.3.3 Performance Portability

Since the baseline code is not even portable, it will not be performance portable in general. In order to make the comparison possible, we choose two platforms both equipped with CUDA-enabled GPUs, thus the baseline code could run on both platforms: one laptop and one node with CUDA-enabled GPUs on the supercomputer Triolith. The machine configuration is shown in Table 6.5.

We measure the baseline code and the rewritten code on three problem sizes 100 times each, and plot their median values and speedup in Figure 7.3. TunerPU is trained at depth 2 with the training range from 1 to 1000 on each of the three dimensions. We can see on the two platforms that the rewritten code is adaptive to the run-time contexts: it shows performance advantages on some problem sizes while on others it is equally good. This also shows the overhead of our facility in most cases is quite low and unnoticeable in this specific setting.



(a) Laptop A (b) Server B

Figure 7.3: Performance results on two GPU-based systems.
X-axis: the number of elements in a square matrix.

7.4 Summary

In this chapter we discussed and implemented a meaningful integration of the approaches and framework prototypes that are covered in the previous four chapters. Our concern was to answer the question on how to combine those approaches and prototypes for the desired combined benefits. We then applied the integration to an example with an important kernel, matrix-matrix multiplication, for an illustration and evaluated the combined benefits.

For the matrix-matrix multiplication, rewriting a program with VectorPU leads to a significant code size drop compared to the original CUDA code. Rewriting with TunePU leads to a significant code size increase, however, the majority of the increased code is to encode the component's own knowledge for tuning, and thus worthwhile. The code written on top of the integrated frameworks is portable, and also shows performance advantages, compared to the non-portable original optimized code. Such performance advantages are also shown to be portable at least across two different GPU-systems.

Chapter 8

Conclusions

GPU-based heterogeneous systems have shown remarkable performance advantages, and such heterogeneity represents one of the most promising trends for the future high performance computing hardware evolvement. However, as a double-edged sword, we have seen various types of complexities which prevent the easy and efficient usage of different GPU-based heterogeneous systems. Our proposed abstractions and techniques in this thesis are explored to tackle these complexities.

- For the measurement complexity, we can hide it without noticeable abstraction overhead on runtime in a unified way, not only for different metrics, but also on different kinds of devices under measurement. Such a unification makes it easy for autotuning software to easily switch to different optimization goals, assuming that no goal-specific assumptions are violated. With MeterPU, we propose a clean design that shows advantage over other alternatives in the size of the code that uses our API.

- The CPU-GPU selection complexity, which is related to how to utilize the available hardware heterogeneity better, is a special case of the implementation variant selection problem. We have seen the complexity of this problem and machine learning techniques can naturally fit for this problem. We propose an adaptive sampling approach to achieve a controllable and efficiently growing prediction accuracy. We do so by extracting representative training examples in the impractically large training space. We propose three pruning strategies to further prune the search space, and by the right combination of those strategies, we could achieve better accuracies with less training time for a large *ceset*. We also show that the adaptive sampling approach is better than random sampling in our benchmarking test. We argue that there is a need to encode the component writer's domain knowledge on the component's data structures[1] only once per *ceset*, not only to make tuning the implementation variant selector possible, but also to offer a unified view to the tuning framework TunerPU thus each *ceset* can be tuned for this purpose uniformly without any modification of TunerPU.

- We have seen that heterogeneous platforms are complex and modeling them requires special care. We propose the platform modeling language XPDL with modern language features (e.g., modularity), which includes system software as a model element, and decouples hardware components with their software roles. We show the expressiveness of our language by modeling different kinds of computers ranging from single node servers to clusters, and with different kinds of heterogeneity which consists of accelerators ranging from mobile processors to GPGPUs. The language also includes support for energy modeling. We build compiler support for XPDL, and demonstrate how to interface XPDL to other software for meaningful exploitation of platform information. The runtime overhead of querying platform information is zero by meta-programming.

- In order to handle the data management complexity, we propose a design of a data management system including an embedded domain specific language, and data containers with run-time coherence management. We show that our design can abstract away explicit data management for an arbitrary kernel, no matter on CPU or GPU. We show the expressiveness of our approach. Considering performance, our approach is much faster than Nvidia's Unified Memory, and shows no noticeable slowdown even compared to expert handwritten code. We propose two data transfer optimizations: lazy allocation and transfer fusion optimization, based on merging messages between CPU and GPU over PCIe bus on typical GPU-based heterogeneous systems. We also propose prototype designs and implementations, and show that they are potentially beneficial even for large data sizes, particularly in the context of memory coherence. Our greedy transfer fusion algorithm is optimal, as shown by mathematical proof.

Our approaches and frameworks to handle different complexities are not standing alone, and they could be integrated meaningfully so that programmers can enjoy those abstractions altogether. Although we only describe one possible way to integrate our prototypes, this could play an inspirational role and by no means limits the possibilities for other ways. Possibly our frameworks could be integrated to other software packages written in C/C++/CUDA, or potentially re-implemented in other programming languages as long as some necessary language features are supported.

[1]As all elements of a *ceset* are equivalent to each other, thus their input and output data structures are the same, so any programmer that contributes an element in the *ceset* are in privileged position to encode such information.

Last but not least, we revisit our research questions of Section 1.1, and provide our answers that can be supported by the results from the previous chapters.

1. `How to improve programmability by software abstractions for a GPU-based heterogeneous system? Does the increased abstraction come at a run-time cost? Is it possible to design zero-cost abstractions?`

   - In this thesis we have offloaded a few responsibilities from the programmers' shoulders, thus programming for these heterogeneous systems becomes easier. Such responsibilities include deciding when to select CPU or GPU to use, or even when to use which algorithm, managing data transfers between host and device memories, maintaining code for portability from the component's point of view, and platform-specific coding to measure time or energy and to optimize it.
   - Abstractions usually come at a cost. However, we managed to make the prorgramming abstractions discussed in this thesis quite low, e.g., XPDL shows strictly zero overhead by metaprogramming, MeterPU and VectorPU show unnoticeable overhead, and TunerPU's overhead is controllable as a trade-off between training time, runtime overhead and prediction accuracy.

2. `How to improve the portability across different heterogeneous systems with different hardware and software configurations, when applications are already written for a specific GPU-based heterogeneous system?`

   - We propose XPDL as a systematic way to encode platform information and its queries. It serves as a basis to improve portability for both the tool chains and programmers by making platform dependencies explicit. For example, we could connect our data structure abstraction to XPDL to make it self-adaptive to different platform configuration, and we could do so to our component selector which controls which algorithms and computing device to use. Whenever programmers feel restricted by the tool chain, they could directly encode their components' portability requirements on the target platforms directly by interfacing with XPDL query API.

3. `How to improve performance portability for programs on different heterogeneous systems? For performance portability, we refer to the automated adaptivity of programs when migrated to a different heterogeneous system to reach decent performance.`

- In this thesis, we are interested in implementation variant selection in a *ceset*, and therefore in making implementation variant selection performance-portable. The performance portability is guaranteed by re-training the implementation variant selector on each different platform or an existing platform whenever a relevant configuration change happens on that platform. The re-training can be expensive due to the impractically large training space of a component, the large number of *ceset*s required for re-training, and the (possibly) frequent changes in relevant configurations on a platform, which is the main challenge. Our answer is the adaptive sampling approach with further pruning techniques, which allow efficient extraction of representative examples for re-training. More research can be directed in this direction from the author's point of view.
- Furthermore, the TFO optimization has been made performance-portable by simply re-running the microbenchmark to find the possibly different L_{max} value on a new platform, thus our fusion algorithm can merge messages adaptively. The lazy allocation optimization could be made performance-portable in the same way, and we are interested in improving its implementation and evaluation in future work.

4. `How to take energy consumption into account as an optimization goal when optimizing programs for heterogeneous systems?`

    - We answer this question by first introducing a unified view on measurements of different metrics on different devices, and then hook the unified view with our implementation selector, thus the selector can be easily re-trained to select components in favor of reducing energy consumption instead of execution time.

Finally, we would like to point out that these research questions are open-ended, hence the abstractions and techniques presented in this thesis are by no means considered optimal (unless proved) or marking the end of these questions. Due to the mysterious nature of design, we will probably observe better designs for both abstractions and optimizations in the future, and hopefully the work in this thesis can play an inspirational role.

Bibliography

[1] Abrahams, D. and Gurtovoy, A. (2004). *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond, Portable Documents*. Pearson Education.

[2] AlSaber, N. and Kulkarni, M. (2013). SemCache: Semantics-aware Caching for Efficient GPU Offloading. ICS '13, pages 421–432.

[3] AlSaber, N. and Kulkarni, M. (2015). SemCache++: Semantics-Aware Caching for Efficient Multi-GPU Offloading. ICS '15, pages 79–88.

[4] Anderson, M., Ballard, G., Demmel, J., and Keutzer, K. (2011). Communication-avoiding QR decomposition for GPUs. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 48–58. IEEE.

[5] Ansel, J., Chan, C. P., Wong, Y. L., Olszewski, M., Zhao, Q., Edelman, A., and Amarasinghe, S. P. (2009). PetaBricks: A language and compiler for algorithmic choice. In *Proc. Conf. Progr. Lang. Design Impl. (PLDI 2009)*, pages 38–49. acm.

[6] Ansel, J., Kamil, S., Veeramachaneni, K., Ragan-Kelley, J., Bosboom, J., O'Reilly, U.-M., and Amarasinghe, S. (2014). Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 303–316. ACM.

[7] Augonnet, C., Thibault, S., and Namyst, R. (2009). Automatic calibration of performance models on heterogeneous multicore architectures. In *Euro-Par Workshops 2009 (HPPC 2009)*, volume 6043 of *lncs*, pages 56–65. sv.

[8] Augonnet, C., Thibault, S., Namyst, R., and Wacrenier, P.-A. (2011). StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, 23:187–198.

[9] Bardzell, J., Bardzell, S., and Koefoed Hansen, L. (2015). Immodest proposals: Research through design and knowledge. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, pages 2093–2102. ACM.

[10] Bastem, B., Unat, D., Zhang, W., Almgren, A., and Shalf, J. (2017). Overlapping data transfers with computation on GPU with tiles. In *2017 46th International Conference on Parallel Processing (ICPP)*, pages 171–180.

[11] Bednarski, A. (2006). *Integrated Optimal Code Generation for Digital Signal Processors*. PhD thesis, Linköping University, The Institute of Technology.

[12] Benkner, S., Pllana, S., Träff, J. L., Tsigas, P., Dolinsky, U., Augonnet, C., Bachmayer, B., Kessler, C., Moloney, D., and Osipov, V. (2011). PEPPHER: Efficient and productive usage of hybrid computing systems. *IEEE Micro*, 31(5):28–41.

[13] Boost Community. Boost C++ libary. `http://www.boost.org/`. Accessed: 2017-12-04.

[14] Bourdon, A., Noureddine, A., Rouvoy, R., and Seinturier, L. (2013). PowerAPI: A Software Library to Monitor the Energy Consumed at the Process-Level. *ERCIM News*, 2013(92).

[15] Breymann, U. (1998). *Designing Components with the C++ STL*. Addison-Wesley.

[16] Broquedis, F., Clet-Ortega, J., Moreaud, S., Furmento, N., Goglin, B., Mercier, G., Thibault, S., and Namyst, R. (2010). hwloc: A generic framework for managing hardware affinities in HPC applications. In *Proceedings of the 18th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 180–186. IEEE Computer Society.

[17] Bujlow, T., Riaz, T., and Pedersen, J. M. (2012). A method for classification of network traffic based on c5. 0 machine learning algorithm. In *Computing, Networking and Communications (ICNC), 2012 International Conference on*, pages 237–241. IEEE.

[18] Burtscher, M., Zecena, I., and Zong, Z. (2014). Measuring GPU power with the K20 built-in sensor. In *Proc. Workshop on General Purpose Processing Using GPUs (GPGPU-7)*. ACM.

[19] Butenhof, D. R. (1997). *Programming with POSIX threads*. Addison-Wesley Professional.

[20] Cabrera, A., Almeida, F., Arteaga, J., and Blanco, V. (2015). Energy Measurement Library (EML) Usage and Overhead Analysis. In *23rd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 554–558.

[21] Cantonnet, F., Yao, Y., Zahran, M., and El-Ghazawi, T. (2004). Productivity analysis of the UPC language. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*

[22] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J., Lee, S., and Skadron, K. (2009). Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54. IEEE.

[23] Cole, M. (1991). *Algorithmic Skeletons: Structured Management of Parallel Computation.* MIT Press, Cambridge, MA, USA.

[24] Cole, M. I. (1989). A "skeletal" approach to the exploitation of parallelism. In *Proceedings of the Conference on CONPAR 88*, pages 667–675, New York, NY, USA. Cambridge University Press.

[25] Dagum, L. and Menon, R. (1998). OpenMP: an industry standard API for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55.

[26] Danylenko, A., Kessler, C., and Löwe, W. (2011). Comparing machine learning approaches for context-aware composition. In *Proc. 10th Int. Conference on Software Composition (SC-2011), Zürich, Switzerland*, volume 6703 of *LNCS*, pages 18–33. Springer.

[27] Dastgeer, U. and Kessler, C. (2016). Smart containers and skeleton programming for GPU-based systems. *International Journal of Parallel Programming*, 44(3):506–530.

[28] Dastgeer, U. and Kessler, C. W. (2014). Conditional component composition for GPU-based systems. In *Proc. MULTIPROG-2014 workshop at HiPEAC-2014 conference, Vienna, Austria.*

[29] Dastgeer, U., Li, L., and Kessler, C. (2012a). D1.4: Research prototype implementation. Technical report, ©The PEPPHER Consortium. June, 2011.

[30] Dastgeer, U., Li, L., and Kessler, C. (2012b). The PEPPHER Composition Tool: Performance-Aware Dynamic Composition of Applications for GPU-based Systems. In *Proc. 2012 Int. Workshop on Multi-Core Computing Systems (MuCoCoS 2012), Nov. 16, 2012, Salt Lake City, Utah, USA, in conjunction with the Supercomputing Conference (SC12).* IEEE.

[31] Dastgeer, U., Li, L., and Kessler, C. (2013). Adaptive Implementation Selection in a Skeleton Programming Library. In *Proc. of the 2013 Biennial Conference on Advanced Parallel Processing Technology (APPT-2013)*, volume LNCS 8299, pages 170–183. Springer.

[32] Dastgeer, U., Li, L., and Kessler, C. (2014). The PEPPHER Composition Tool: Performance-Aware Composition for GPU-based Systems. *Computing*, 96(12):1195–1211. doi: 10.1007/s00607-013-0371-8.

[33] de Mesmay, F., Voronenko, Y., and Püschel, M. (2010). Offline library adaptation using automatically generated heuristics. In *Int. Parallel and Distr. Processing Symp. (IPDPS'10)*, pages 1–10.

[34] Dean, J. and Ghemawat, S. (2010). Mapreduce: a flexible data processing tool. *Communications of the ACM*, 53(1):72–77.

[35] Demmel, J., Grigori, L., Hoemmen, M., and Langou, J. (2012). Communication-optimal parallel and sequential QR and LU factorizations. *SIAM Journal on Scientific Computing*, 34(1):A206–A239.

[36] Diogo, M. and Grelck, C. (2013). Towards Heterogeneous Computing without Heterogeneous Programming. In *Proc. TFP 2012, Springer LNCS 7829*, pages 279–294.

[37] Dolkas, K., Sandoval, Y., Hoppe, D., Khabi, D., Umar, I., Ha, P., Moloney, D., Li, L., Kessler, C., Gidenstam, A., and Renaud-Goud, P. (2015). D5.3 Report on integrating the first designs and prototypes from technical WPs. Technical Report FP7-611183 D5.3, EU FP7 Project EXCESS.

[38] Enmyren, J. and Kessler, C. W. (2010). SkePU: A multi-backend skeleton programming library for multi-GPU systems. In *Proc. 4th Int. Workshop on High-Level Parallel Programming and Applications (HLPP-2010), Baltimore, Maryland, USA*, pages 5–14. ACM. doi: 10.1145/1863482.1863487.

[39] Ernsting, S. and Kuchen, H. (2012). Algorithmic skeletons for multi-core, multi-GPU systems and clusters. *Int. Journal of High Performance Computing and Networking*, 7:129–138.

[40] Ernstsson, A., Li, L., and Kessler, C. (2016). SkePU 2: Flexible and Type-safe Skeleton Programming for Heterogeneous Parallel Systems. *International Journal of Parallel Programming.*

[41] Fatahalian, K., Horn, D. R., Knight, T. J., Leem, L., Houston, M., Park, J. Y., Erez, M., Ren, M., Aiken, A., Dally, W. J., and Hanrahan, P. (2006). Sequoia: Programming the memory hierarchy. In *ACM/IEEE Supercomputing*, page 83.

[42] Fraguela, B. B., Voronenko, Y., and Püschel, M. (2009). Automatic tuning of discrete Fourier transforms driven by analytical modeling. In *Parallel Architectures and Compilation Techniques (PACT)*, pages 271–280.

[43] Frigo, M. and Johnsson, S. G. (1998). Fftw: An adaptive software architecture for the FFT. In *Proc. IEEE Int. Conf. on Acoustics, Speech, and Signal Processing*, volume 3, pages 1381–1384.

[44] Gaver, B. and Bowers, J. (2012). Annotated portfolios. *interactions*, 19(4):40–49.

[45] Gelado, I., Stone, J. E., Cabezas, J., Patel, S., Navarro, N., and Hwu, W.-m. W. (2010). An Asymmetric Distributed Shared Memory Model for Heterogeneous Parallel Systems. *SIGPLAN Not.*, 45(3):347–358.

[46] Goli, M. and Gonzalez-Velez, H. (2013). Heterogeneous algorithmic skeletons for FastFlow with seamless coordination over hybrid architectures. In *Euromicro PDP Int. Conf. on Par., Distrib. and Netw.-Based Processing*, pages 148–156.

[47] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep learning.* MIT press.

[48] Grewe, D. and O'Boyle, M. F. P. (2011). A static task partitioning approach for heterogeneous systems using opencl. In *Proc. 20th int. conf. on Compiler construction*, CC'11/ETAPS'11, pages 286–305. Springer-Verlag.

[49] GroundWork Inc. GroundWork—Unified Monitoring For Real. `http://www.gwos.com/`. Accessed: 2015-01-21.

[50] Harris, M. (2013). Unified memory in cuda 6. https://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/. Accessed: 2017-11-26.

[51] Hennessy, J. L. and Patterson, D. A. (2011). *Computer architecture: a quantitative approach.* Elsevier.

[52] Hoefler, T., Gropp, W., Kramer, W., and Snir, M. (2011). Performance modeling for systematic performance tuning. In *State of the Practice Reports*, page 6. ACM.

[53] Hoisie, A., Lubeck, O., Wasserman, H., Petrini, F., and Alme, H. (2000). A general predictive performance model for wavefront algorithms on clusters of smps. In *Proc. Int. Conf. on Parallel Processing*, pages 219–228. IEEE.

[54] Hudak, P. (1989). Conception, evolution, and application of functional programming languages. *ACM Computing Surveys (CSUR)*, 21(3):359–411.

[55] Ishizaki, K., Hayashi, A., Koblents, G., and Sarkar, V. (2015). Compiling and optimizing Java 8 programs for GPU execution. In *PACT'15*, pages 419–431.

[56] Jablin, T. B., Jablin, J. A., Prabhu, P., Liu, F., and August, D. I. (2012). Dynamically Managed Data for CPU-GPU Architectures. CGO '12, pages 165–174.

[57] Jablin, T. B., Prabhu, P., Jablin, J. A., Johnson, N. P., Beard, S. R., and August, D. I. (2011). Automatic CPU-GPU Communication Management and Optimization. *SIGPLAN Not.*, 46(6):142–151.

[58] Jia, W., Shaw, K. A., and Martonosi, M. (2012). Stargazer: Automated regression-based GPU design space exploration. In *2012 IEEE International Symposium on Performance Analysis of Systems Software*, pages 2–13.

[59] Johnson, S. (1979). *YACC: Yet Another Compiler-Compiler, Unix Programmer's Manual.* Holt, Reinhart, and Winston.

[60] Josephsen, D. (2007). *Building a Monitoring Infrastructure with Nagios.* Prentice Hall PTR, Upper Saddle River, NJ, USA.

[61] Katagiri, T., Kise, K., Honda, H., and Yuba, T. (2006). Abclibscript: a directive to support specification of an auto-tuning facility for numerical software. *Parallel Computing*, 32(1):92–112.

[62] Kessler, C., Dastgeer, U., and Li, L. (2014a). Optimized Composition: Generating Efficient Code for Heterogeneous Systems from Multi-Variant Components, Skeletons and Containers. In *Proc. First Workshop on Resource awareness and adaptivity in multi-core computing (Racing 2014)*, pages 43–48.

[63] Kessler, C. and Keller, J. (2007). Models for parallel computing: Review and perspectives. *Mitteilungen-Gesellschaft für Informatik eV, Parallel-Algorithmen und Rechnerstrukturen*, 24.

[64] Kessler, C., Li, L., Atalar, A., and Dobre, A. (2015a). XPDL: Extensible Platform Description Language to Support Energy Modeling and Optimization. In *2015 44th International Conference on Parallel Processing Workshops (ICPPW)*, pages 51–60. IEEE.

[65] Kessler, C., Li, L., Dastgeer, U., Cuello, R., Sjöström, O., Hoai, P. H., and Tran, V. (2015b). D1.3 Energy-tuneable domain-specific language/library for linear system solving. Technical Report FP7-611183 D1.3, EU FP7 Project EXCESS.

[66] Kessler, C., Li, L., Dastgeer, U., Gidenstam, A., and Atalar, A. (2014b). D1.2 Initial specification of energy, platform and component modelling framework. Technical Report FP7-611183 D1.2, EU FP7 Project EXCESS.

[67] Kessler, C., Li, L., Dastgeer, U., Tsigas, P., Gidenstam, A., Renaud-Goud, P., Walulya, I., Atalar, A., Moloney, D., Hoai, P. H., and Tran, V. (2014c). D1.1 Early validation of system-wide energy compositionality and affecting factors on the EXCESS platforms. Technical Report FP7-611183 D1.1, EU FP7 Project EXCESS.

[68] Kessler, C., Li, L., Hansson, E., Ahlqvist, J., Thorarensen, S., and Yang, M.-J. (2015c). D1.4 First prototype of composition tool and multi-level energy and platform modeling framework. Technical Report FP7-611183 D1.4, EU FP7 Project EXCESS.

[69] Kessler, C., Li, L., Melot, N., Hansson, E., Ernstsson, A., Thorarensen, S., and Barry, B. (2016). Final specification of energy, platform and component modelling framework and final prototype. Technical Report FP7-611183 D1.5, EU FP7 Project EXCESS.

[70] Kessler, C. and Löwe, W. (2012). Optimized composition of performance-aware parallel components. *Concurrency and Computation: Practice and Experience*, 24(5):481–498.

[71] Kessler, C. W. (2013). Compiling for VLIW DSPs. In Bhattacharyya, S., Deprettere, E., Leupers, R., and Takala, J., editors, *Handbook of Signal Processing Systems, 2nd edition*. Springer.

[72] Kessler, C. W. and Löwe, W. (2007). A framework for performance-aware composition of explicitly parallel components. In *Parallel Computing: Architectures, Algorithms and Applications, (ParCo 2007)*, volume 15 of *Advances in Parallel Computing*, pages 227–234. IOS Press.

[73] Kiss, A., Danelutto, M., Herczeg, Z., Molnar, P., Sipka, R., Torquati, M., and Vidacs, L. (2015). D6.4: REPARA performance and energy monitoring library. Technical report, ©The REPARA Consortium.

[74] Kupriyanov, A., Hannig, F., Kissler, D., Teich, J., Schaffer, R., and Merker, R. (2006). An architecture description language for massively parallel processor architectures. In *GI/ITG/GMM-Workshop - Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Sytemen*, pages 11–20.

[75] Landaverde, R., Zhang, T., Coskun, A. K., and Herbordt, M. (2014). An investigation of Unified Memory Access performance in CUDA. In *HPEC'14*, pages 1–6.

[76] Lee, S. and Eigenmann, R. (2010). OpenMPC: Extended OpenMP programming and tuning for GPUs. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA. IEEE Computer Society.

[77] Lee, S., Li, D., and Vetter, J. S. (2014). Interactive Program Debugging and Optimization for Directive-based, Efficient GPU Computing. IPDPS'14, pages 481–490.

[78] Lee, S., Min, S.-J., and Eigenmann, R. (2009). OpenMP to GPGPU: a compiler framework for automatic translation and optimization. *ACM Sigplan Notices*, 44(4):101–110.

[79] Li, L., Dastgeer, U., and Kessler, C. (2013). Adaptive off-line tuning for optimized composition of components for heterogeneous many-core systems. In *High Performance Computing for Computational Science-VECPAR 2012*, pages 329–345. Springer.

[80] Li, L., Dastgeer, U., and Kessler, C. (2014). Pruning Strategies in Adaptive Off-line Tuning for Optimized Composition of Components on Heterogeneous Systems. In *Proc. Seventh International Workshop on Parallel Programming Models and Systems Software for High-End Computing (P2S2) at ICPP*. IEEE.

[81] Li, L., Dastgeer, U., and Kessler, C. (2016). Pruning Strategies in Adaptive Off-line Tuning for Optimized Composition of Components on Heterogeneous Systems. *Parallel Computing*, 51:37–45.

[82] Li, L. and Kessler, C. (2015). MeterPU: A Generic Measurement Abstraction API Enabling Energy-tuned Skeleton Backend Selection. In *Proc. International Workshop on Reengineering for Parallelism in Heterogeneous Parallel Platforms (REPARA-2015) at ISPA-2015*, Helsinki. IEEE.

[83] Li, L. and Kessler, C. (2016). MeterPU: A Generic Measurement Abstraction API Enabling Energy-tuned Skeleton Backend Selection. *Journal of Supercomputing*, pages 1–16. Springer.

[84] Li, L. and Kessler, C. (2017a). Lazy allocation and transfer fusion optimization for GPU-based heterogeneous systems. In *Proc. Euromicro PDP-2018 Int. Conf. on Parallel, Distributed, and Network-based Processing*. IEEE.

[85] Li, L. and Kessler, C. (2017b). VectorPU: A Generic and Efficient Data-container and Component Model for Transparent Data Transfer on GPU-based Heterogeneous Systems. In *Proc. 8th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core*

Architectures and 6th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM'17). ACM.

[86] Li, X. and Garzarán, M. J. (2005). Optimizing matrix multiplication with a classifier learning system. In *Proc. workshop on Languages and Compilers for Parallel Computing (LCPC'05)*, pages 121–135.

[87] Li, X., Garzarán, M. J., and Padua, D. (2004). A dynamically tuned sorting library. In *Proc. ACM Symp. on Code Generation and Optimization (CGO'04)*, pages 111–124.

[88] Marques, R., Paulino, H., Alexandre, F., and Medeiros, P. D. (2013). Algorithmic skeleton framework for the orchestration of GPU computations. In *Euro-Par 2013 Parallel Processing*, volume LNCS 8097, pages 874–885. Springer.

[89] Memeti, S., Li, L., Pllana, S., Kolodziej, J., and Kessler, C. (2017). Benchmarking OpenCL, OpenACC, OpenMP, and CUDA: Programming productivity, performance, and energy consumption. In *Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing*, ARMS-CC '17, pages 1–6, New York, NY, USA. ACM.

[90] Mishra, P. and Dutt, N. (2005). Architecture description languages for programmable embedded systems. *IEE Proc.-Comput. Digit. Tech.*, 152(3):285–297.

[91] Muchnick, S. S. (1997). *Advanced compiler design and implementation*. Morgan Kaufmann.

[92] Munshi, A. (2009). The OpenCL specification. In *Hot Chips 21 Symposium (HCS), 2009 IEEE*, pages 1–314. IEEE.

[93] Munson, J. C. and Elbaum, S. G. (1998). Code churn: A measure for estimating the impact of code change. In *Proceedings of International Conference on Software Maintenance*, pages 24–31. IEEE.

[94] NVIDIA Corp. NVML API reference guide. `http://docs.nvidia.com/deploy/nvml-api/index.html`. Accessed: 2017-12-04.

[95] NVIDIA Corp. (2010). GPU occupancy calculator. *CUDA SDK*.

[96] NVIDIA Corp. (2012). Cuda toolkit 4.2 cublas library. https://developer.download.nvidia.com/compute/DevZone/docs/html/-CUDALibraries/doc/CUBLAS_Library.pdf. Accessed: 2017-11-26.

[97] NVIDIA Corp. (2013). Thrust quick start guide.

[98] NVIDIA Corp. (2017a). CUDA C programming guide.

[99] NVIDIA Corp. (2017b). Cusparse library. http://docs.nvidia.com/cuda/pdf/CUSPARSE_Library.pdf. Accessed: 2017-11-26.

[100] Odersky, M., Spoon, L., and Venners, B. (2008). *Programming in Scala.* Artima Inc.

[101] Pai, S., Govindarajan, R., and Thazhuthaveetil, M. J. (2012). Fast and Efficient Automatic Memory Management for GPUs Using Compiler-assisted Runtime Coherence Scheme. PACT '12, pages 33–42.

[102] Papadimitriou, C. H. and Yannakakis, M. (1990). Towards an architecture-independent analysis of parallel algorithms. *SIAM journal on computing*, 19(2):322–328.

[103] Park, E., Kulkarni, S., and Cavazos, J. (2011). An evaluation of different modeling techniques for iterative compilation. In *Proc. Int. Conf. on Compilers, Architectures and Synthesis for Embedded Systems (CASES'11).*

[104] PEPPHER consortium. Performance portability and programmability for heterogeneous many-core architectures - PEPPHER. `http://www.peppher.eu/`. Accessed: 2017-12-04.

[105] Pfleeger, S. L. and Atlee, J. M. (1998). *Software engineering: theory and practice.* Pearson.

[106] Püschel, M., Moura, J. M. F., Johnson, J. R., Padua, D., Veloso, M. M., Singer, B. W., Xiong, J., Franchetti, F., Gacic, A., Voronenko, Y., Chen, K., Johnson, R. W., and Rizzolo, N. (2005). Spiral: Code generation for DSP transforms. *Proceedings of the IEEE*, 93(2).

[107] Quinlan, J. R. (1993). *C4.5: programs for machine learning.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[108] Sanchez, L. M. (2014). D3.1: Target platform description specification, EU FP7 REPARA project deliverable report. Technical Report FP7-611183 D1.2, EU FP7 Project REPARA.

[109] Sandahl, K., Li, L., and Kessler, C. Personal communication. 2017-12-19. Linköping University, Sweden.

[110] Sandrieser, M., Benkner, S., and Pllana, S. (2012). Using explicit platform descriptions to support programming of heterogeneous many-core systems. *Parallel Computing*, 38(1-2):52–65.

[111] Siegel, J. and Frantz, D. (2000). *CORBA 3 fundamentals and programming*, volume 2. John Wiley & Sons New York, NY, USA.

[112] Singer, B. and Veloso, M. (2000). Learning to predict performance from formula modeling and training data. In *Proc. 17th Int. Conf. on Machine Learning*, pages 887–894.

[113] Singer, B. and Veloso, M. (2002). Learning to construct fast signal processing implementations. *Journal of Machine Learning Research*, 3:887–919.

[114] Sjöström, O., Ko, S.-H., Dastgeer, U., Li, L., and Kessler, C. (2015). Portable Parallelization of the EDGE CFD Application for GPU-based Systems using the SkePU Skeleton Programming Library. In *ParCo-2015 conference*, pages 135–144. Published in: Gerhard R. Joubert, Hugh Leather, Mark Parsons, Frans Peters, Mark Sawyer (eds.): Advances in Parallel Computing, Volume 27: Parallel Computing: On the Road to Exascale, IOS Press, April 2016, pages 135-144. DOI 10.3233/978-1-61499-621-7-135.

[115] Srinivasan, H., Hook, J., and Wolfe, M. (1993). Static single assignment for explicitly parallel programs. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 260–272. ACM.

[116] Steuwer, M., Kegel, P., and Gorlatch, S. (2011). SkelCL - A portable skeleton library for high-level GPU programming. In *16th Int. Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'11)*.

[117] Steuwer, M., Remmelg, T., and Dubach, C. (2017). Lift: a functional data-parallel IR for high-performance GPU code generation. In *Code Generation and Optimization (CGO), 2017 IEEE/ACM International Symposium on*, pages 74–85. IEEE.

[118] Stripf, T., Oey, O., Bruckschloegl, T., Koenig, R., Goulas, G., Alefragis, P., Voros, N. S., Potman, J., Sunesen, K., Derrien, S., Sentieys, O., and Becker, J. (2012). A compilation- and simulation-oriented architecture description language for multicore systems. In *2012 IEEE 15th International Conference on Computational Science and Engineering*, pages 383–390.

[119] Sunitha, N. V., Raju, K., and Chiplunkar, N. N. (2017). Performance improvement of CUDA applications by reducing CPU-GPU data transfer overhead. In *2017 International Conference on Inventive Communication and Computational Technologies (ICICCT)*, pages 211–215.

[120] Szyperski, C., Gruntz, D., and Murer, S. (1998). Component software: Beyond object-oriented programming. *Addison-Wesley*.

[121] Teijeiro, C., Taboada, G. L., Touriño, J., Fraguela, B. B., Doallo, R., Mallón, D. A., Gómez, A., Mouriño, J. C., and Wibecan, B. (2009). Evaluation of UPC programmability using classroom studies. In *Proceedings of the Third Conference on Partitioned Global Address Space Programing Models*, PGAS '09, pages 10:1–10:7, New York, NY, USA. ACM.

[122] Thomas, N., Tanase, G., Tkachyshyn, O., Perdue, J., Amato, N. M., and Rauchwerger, L. (2005). A framework for adaptive algorithm selection in STAPL. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 277–288. acm.

[123] Thomson, J., O'Boyle, M. F. P., Fursin, G., and Franke, B. (2009). Reducing training time in a one-shot machine learning-based compiler. In Gao, G. R., Pollock, L. L., Cavazos, J., and Li, X., editors, *LCPC*, volume 5898 of *Lecture Notes in Computer Science*, pages 399–407. Springer.

[124] Thorarensen, S., Cuello, R., Kessler, C., Li, L., and Barry, B. (2016). Efficient Execution of SkePU Skeleton Programs on the Low-power Multicore Processor Myriad2. In *Proc. Euromicro PDP-2016 Int. Conf. on Parallel, Distributed, and Network-based Processing*. IEEE.

[125] Wang, Z. and O'Boyle, M. F. (2009). Mapping parallelism to multicores: a machine learning based approach. *SIGPLAN Not.*, 44(4):75–84.

[126] Wernsing, J. R. and Stitt, G. (2010). Elastic computing: A framework for transparent, portable, and adaptive multi-core heterogeneous computing. In *Proceedings of the ACM SIGPLAN/SIGBED 2010 conference on Languages, compilers, and tools for embedded systems (LCTES)*, pages 115–124. ACM.

[127] Whaley, R. C., Petitet, A., and Dongarra, J. (2001). Automated empirical optimizations of software and the ATLAS project. *pc*, 27(1-2):3–35.

[128] Wienke, S., Springer, P., Terboven, C., and an Mey, D. (2012). OpenACC–first experiences with real-world applications. *Euro-Par 2012 Parallel Processing*, pages 859–870.

[129] Willhalm, T., Dementiev, R., and Fay, P. (2016). Intel performance counter monitor - a better way to measure CPU utilization. https://software.intel.com/en-us/articles/intel-performance-counter-monitor. Accessed: 2017-12-19.

[130] Yu, H. and Rauchwerger, L. (2006). An adaptive algorithm selection framework for reduction parallelization. *IEEE Trans. on Par. and Distr. Syst.*, 17(10):1084–1096.

[131] Zuluaga, M., Krause, A., Milder, P., and Püschel, M. (2012). Smart design space sampling to predict pareto-optimal solutions. *SIGPLAN Not.*, 47(5):119–128.

Department of Computer and Information Science
Linköpings universitet

Dissertations

Linköping Studies in Science and Technology
Linköping Studies in Arts and Science
Linköping Studies in Statistics
Linköping Studies in Information Science

Linköping Studies in Science and Technology

No 14 **Anders Haraldsson:** A Program Manipulation System Based on Partial Evaluation, 1977, ISBN 91-7372-144-1.

No 17 **Bengt Magnhagen:** Probability Based Verification of Time Margins in Digital Designs, 1977, ISBN 91-7372-157-3.

No 18 **Mats Cedwall**: Semantisk analys av processbeskrivningar i naturligt språk, 1977, ISBN 91- 7372-168-9.

No 22 **Jaak Urmi:** A Machine Independent LISP Compiler and its Implications for Ideal Hardware, 1978, ISBN 91-7372-188-3.

No 33 **Tore Risch:** Compilation of Multiple File Queries in a Meta-Database System, 1978, ISBN 91- 7372-232-4.

No 51 **Erland Jungert:** Synthesizing Database Structures from a User Oriented Data Model, 1980, ISBN 91-7372-387-8.

No 54 **Sture Hägglund:** Contributions to the Development of Methods and Tools for Interactive Design of Applications Software, 1980, ISBN 91-7372-404-1.

No 55 **Pär Emanuelson:** Performance Enhancement in a Well-Structured Pattern Matcher through Partial Evaluation, 1980, ISBN 91-7372-403-3.

No 58 **Bengt Johnsson, Bertil Andersson:** The Human-Computer Interface in Commercial Systems, 1981, ISBN 91-7372-414-9.

No 69 **H. Jan Komorowski:** A Specification of an Abstract Prolog Machine and its Application to Partial Evaluation, 1981, ISBN 91-7372-479-3.

No 71 **René Reboh:** Knowledge Engineering Techniques and Tools for Expert Systems, 1981, ISBN 91-7372-489-0.

No 77 **Östen Oskarsson:** Mechanisms of Modifiability in large Software Systems, 1982, ISBN 91- 7372-527-7.

No 94 **Hans Lunell:** Code Generator Writing Systems, 1983, ISBN 91-7372-652-4.

No 97 **Andrzej Lingas:** Advances in Minimum Weight Triangulation, 1983, ISBN 91-7372-660-5.

No 109 **Peter Fritzson:** Towards a Distributed Programming Environment based on Incremental Compilation, 1984, ISBN 91-7372-801-2.

No 111 **Erik Tengvald:** The Design of Expert Planning Systems. An Experimental Operations Planning System for Turning, 1984, ISBN 91-7372- 805-5.

No 155 **Christos Levcopoulos:** Heuristics for Minimum Decompositions of Polygons, 1987, ISBN 91-7870-133-3.

No 165 **James W. Goodwin:** A Theory and System for Non-Monotonic Reasoning, 1987, ISBN 91-7870-183-X.

No 170 **Zebo Peng:** A Formal Methodology for Automated Synthesis of VLSI Systems, 1987, ISBN 91-7870-225-9.

No 174 **Johan Fagerström:** A Paradigm and System for Design of Distributed Systems, 1988, ISBN 91-7870-301-8.

No 192 **Dimiter Driankov:** Towards a Many Valued Logic of Quantified Belief, 1988, ISBN 91-7870-374-3.

No 213 **Lin Padgham:** Non-Monotonic Inheritance for an Object Oriented Knowledge Base, 1989, ISBN 91-7870-485-5.

No 214 **Tony Larsson:** A Formal Hardware Description and Verification Method, 1989, ISBN 91-7870-517-7.

No 221 **Michael Reinfrank:** Fundamentals and Logical Foundations of Truth Maintenance, 1989, ISBN 91-7870-546-0.

No 239 **Jonas Löwgren:** Knowledge-Based Design Support and Discourse Management in User Interface Management Systems, 1991, ISBN 91-7870-720-X.

No 244 **Henrik Eriksson:** Meta-Tool Support for Knowledge Acquisition, 1991, ISBN 91-7870-746-3.

No 252 **Peter Eklund:** An Epistemic Approach to Interactive Design in Multiple Inheritance Hierarchies, 1991, ISBN 91-7870-784-6.

No 258 **Patrick Doherty:** NML3 - A Non-Monotonic Formalism with Explicit Defaults, 1991, ISBN 91-7870-816-8.

No 260 **Nahid Shahmehri:** Generalized Algorithmic Debugging, 1991, ISBN 91-7870-828-1.

No 264 **Nils Dahlbäck:** Representation of Discourse-Cognitive and Computational Aspects, 1992, ISBN 91-7870-850-8.

No 265 **Ulf Nilsson:** Abstract Interpretations and Abstract Machines: Contributions to a Methodology for the Implementation of Logic Programs, 1992, ISBN 91-7870-858-3.

No 270 **Ralph Rönnquist:** Theory and Practice of Tense-bound Object References, 1992, ISBN 91-7870-873-7.

No 273 **Björn Fjellborg:** Pipeline Extraction for VLSI Data Path Synthesis, 1992, ISBN 91-7870-880-X.

No 276 **Staffan Bonnier:** A Formal Basis for Horn Clause Logic with External Polymorphic Functions, 1992, ISBN 91-7870-896-6.

No 277 **Kristian Sandahl:** Developing Knowledge Management Systems with an Active Expert Methodology, 1992, ISBN 91-7870-897-4.

No 281 **Christer Bäckström:** Computational Complexity of Reasoning about Plans, 1992, ISBN 91-7870-979-2.

No 292 **Mats Wirén:** Studies in Incremental Natural Language Analysis, 1992, ISBN 91-7871-027-8.

No 297 **Mariam Kamkar:** Interprocedural Dynamic Slicing with Applications to Debugging and Testing, 1993, ISBN 91-7871-065-0.

No 302 **Tingting Zhang:** A Study in Diagnosis Using Classification and Defaults, 1993, ISBN 91-7871-078-2.

No 312 **Arne Jönsson:** Dialogue Management for Natural Language Interfaces - An Empirical Approach, 1993, ISBN 91-7871-110-X.

No 338 **Simin Nadjm-Tehrani:** Reactive Systems in Physical Environments: Compositional Modelling and Framework for Verification, 1994, ISBN 91-7871-237-8.

No 371 **Bengt Savén:** Business Models for Decision Support and Learning. A Study of Discrete-Event Manufacturing Simulation at Asea/ABB 1968-1993, 1995, ISBN 91-7871-494-X.

No 375 **Ulf Söderman:** Conceptual Modelling of Mode Switching Physical Systems, 1995, ISBN 91-7871-516-4.

No 383 **Andreas Kågedal:** Exploiting Groundness in Logic Programs, 1995, ISBN 91-7871-538-5.

No 396 **George Fodor:** Ontological Control, Description, Identification and Recovery from Problematic Control Situations, 1995, ISBN 91-7871-603-9.

No 413 **Mikael Pettersson:** Compiling Natural Semantics, 1995, ISBN 91-7871-641-1.

No 414 **Xinli Gu:** RT Level Testability Improvement by Testability Analysis and Transformations, 1996, ISBN 91-7871-654-3.

No 416 **Hua Shu:** Distributed Default Reasoning, 1996, ISBN 91-7871-665-9.

No 429 **Jaime Villegas:** Simulation Supported Industrial Training from an Organisational Learning Perspective - Development and Evaluation of the SSIT Method, 1996, ISBN 91-7871-700-0.

No 431 **Peter Jonsson:** Studies in Action Planning: Algorithms and Complexity, 1996, ISBN 91-7871-704-3.

No 437 **Johan Boye:** Directional Types in Logic Programming, 1996, ISBN 91-7871-725-6.

No 439 **Cecilia Sjöberg:** Activities, Voices and Arenas: Participatory Design in Practice, 1996, ISBN 91-7871-728-0.

No 448 **Patrick Lambrix:** Part-Whole Reasoning in Description Logics, 1996, ISBN 91-7871-820-1.

No 452 **Kjell Orsborn:** On Extensible and Object-Relational Database Technology for Finite Element Analysis Applications, 1996, ISBN 91-7871-827-9.

No 459 **Olof Johansson:** Development Environments for Complex Product Models, 1996, ISBN 91-7871-855-4.

No 461 **Lena Strömbäck:** User-Defined Constructions in Unification-Based Formalisms, 1997, ISBN 91-7871-857-0.

No 462 **Lars Degerstedt:** Tabulation-based Logic Programming: A Multi-Level View of Query Answering, 1996, ISBN 91-7871-858-9.

No 475 **Fredrik Nilsson:** Strategi och ekonomisk styrning - En studie av hur ekonomiska styrsystem utformas och används efter företagsförvärv, 1997, ISBN 91-7871-914-3.

No 480 **Mikael Lindvall:** An Empirical Study of Requirements-Driven Impact Analysis in Object-Oriented Software Evolution, 1997, ISBN 91-7871-927-5.

No 485 **Göran Forslund:** Opinion-Based Systems: The Cooperative Perspective on Knowledge-Based Decision Support, 1997, ISBN 91-7871-938-0.

No 494 **Martin Sköld:** Active Database Management Systems for Monitoring and Control, 1997, ISBN 91-7219-002-7.

No 495 **Hans Olsén:** Automatic Verification of Petri Nets in a CLP framework, 1997, ISBN 91-7219-011-6.

No 498 **Thomas Drakengren:** Algorithms and Complexity for Temporal and Spatial Formalisms, 1997, ISBN 91-7219-019-1.

No 502 **Jakob Axelsson:** Analysis and Synthesis of Heterogeneous Real-Time Systems, 1997, ISBN 91-7219-035-3.

No 503 **Johan Ringström:** Compiler Generation for Data-Parallel Programming Languages from Two-Level Semantics Specifications, 1997, ISBN 91-7219-045-0.

No 512 **Anna Moberg:** Närhet och distans - Studier av kommunikationsmönster i satellitkontor och flexibla kontor, 1997, ISBN 91-7219-119-8.

No 520 **Mikael Ronström:** Design and Modelling of a Parallel Data Server for Telecom Applications, 1998, ISBN 91-7219-169-4.

No 522 **Niclas Ohlsson:** Towards Effective Fault Prevention - An Empirical Study in Software Engineering, 1998, ISBN 91-7219-176-7.

No 526 **Joachim Karlsson:** A Systematic Approach for Prioritizing Software Requirements, 1998, ISBN 91-7219-184-8.

No 530 **Henrik Nilsson:** Declarative Debugging for Lazy Functional Languages, 1998, ISBN 91-7219-197-X.

No 555 **Jonas Hallberg:** Timing Issues in High-Level Synthesis, 1998, ISBN 91-7219-369-7.

No 561 **Ling Lin:** Management of 1-D Sequence Data - From Discrete to Continuous, 1999, ISBN 91-7219-402-2.

No 563 **Eva L Ragnemalm:** Student Modelling based on Collaborative Dialogue with a Learning Companion, 1999, ISBN 91-7219-412-X.

No 567 **Jörgen Lindström:** Does Distance matter? On geographical dispersion in organisations, 1999, ISBN 91-7219-439-1.

No 582 **Vanja Josifovski:** Design, Implementation and Evaluation of a Distributed Mediator System for Data Integration, 1999, ISBN 91-7219-482-0.

No 589 **Rita Kovordányi:** Modeling and Simulating Inhibitory Mechanisms in Mental Image Reinterpretation - Towards Cooperative Human-Computer Creativity, 1999, ISBN 91-7219-506-1.

No 592 **Mikael Ericsson:** Supporting the Use of Design Knowledge - An Assessment of Commenting Agents, 1999, ISBN 91-7219-532-0.

No 593 **Lars Karlsson:** Actions, Interactions and Narratives, 1999, ISBN 91-7219-534-7.

No 594 **C. G. Mikael Johansson:** Social and Organizational Aspects of Requirements Engineering Methods - A practice-oriented approach, 1999, ISBN 91-7219-541-X.

No 595 **Jörgen Hansson:** Value-Driven Multi-Class Overload Management in Real-Time Database Systems, 1999, ISBN 91-7219-542-8.

No 596 **Niklas Hallberg:** Incorporating User Values in the Design of Information Systems and Services in the Public Sector: A Methods Approach, 1999, ISBN 91-7219-543-6.

No 597 **Vivian Vimarlund:** An Economic Perspective on the Analysis of Impacts of Information Technology: From Case Studies in Health-Care towards General Models and Theories, 1999, ISBN 91-7219-544-4.

No 598 **Johan Jenvald:** Methods and Tools in Computer-Supported Taskforce Training, 1999, ISBN 91-7219-547-9.

No 607 **Magnus Merkel:** Understanding and enhancing translation by parallel text processing, 1999, ISBN 91-7219-614-9.

No 611 **Silvia Coradeschi:** Anchoring symbols to sensory data, 1999, ISBN 91-7219-623-8.

No 613 **Man Lin:** Analysis and Synthesis of Reactive Systems: A Generic Layered Architecture Perspective, 1999, ISBN 91-7219-630-0.

No 618 **Jimmy Tjäder:** Systemimplementering i praktiken - En studie av logiker i fyra projekt, 1999, ISBN 91-7219-657-2.

No 627 **Vadim Engelson:** Tools for Design, Interactive Simulation, and Visualization of Object-Oriented Models in Scientific Computing, 2000, ISBN 91-7219-709-9.

No 637 **Esa Falkenroth:** Database Technology for Control and Simulation, 2000, ISBN 91-7219-766-8.

No 639 **Per-Arne Persson:** Bringing Power and Knowledge Together: Information Systems Design for Autonomy and Control in Command Work, 2000, ISBN 91-7219-796-X.

No 660 **Erik Larsson:** An Integrated System-Level Design for Testability Methodology, 2000, ISBN 91-7219-890-7.

No 688 **Marcus Bjäreland:** Model-based Execution Monitoring, 2001, ISBN 91-7373-016-5.

No 689 **Joakim Gustafsson:** Extending Temporal Action Logic, 2001, ISBN 91-7373-017-3.

No 720 **Carl-Johan Petri:** Organizational Information Provision - Managing Mandatory and Discretionary Use of Information Technology, 2001, ISBN 91-7373-126-9.

No 724 **Paul Scerri:** Designing Agents for Systems with Adjustable Autonomy, 2001, ISBN 91-7373-207-9.

No 725 **Tim Heyer:** Semantic Inspection of Software Artifacts: From Theory to Practice, 2001, ISBN 91-7373-208-7.

No 726 **Pär Carlshamre:** A Usability Perspective on Requirements Engineering - From Methodology to Product Development, 2001, ISBN 91-7373-212-5.

No 732 **Juha Takkinen:** From Information Management to Task Management in Electronic Mail, 2002, ISBN 91-7373-258-3.

No 745 **Johan Åberg:** Live Help Systems: An Approach to Intelligent Help for Web Information Systems, 2002, ISBN 91-7373-311-3.

No 746 **Rego Granlund:** Monitoring Distributed Teamwork Training, 2002, ISBN 91-7373-312-1.

No 757 **Henrik André-Jönsson:** Indexing Strategies for Time Series Data, 2002, ISBN 917373-346-6.

No 747 **Anneli Hagdahl:** Development of IT-supported Interorganisational Collaboration - A Case Study in the Swedish Public Sector, 2002, ISBN 91-7373-314-8.

No 749 **Sofie Pilemalm:** Information Technology for Non-Profit Organisations - Extended Participatory Design of an Information System for Trade Union Shop Stewards, 2002, ISBN 91-7373-318-0.

No 765 **Stefan Holmlid:** Adapting users: Towards a theory of use quality, 2002, ISBN 91-7373-397-0.

No 771 **Magnus Morin:** Multimedia Representations of Distributed Tactical Operations, 2002, ISBN 91-7373-421-7.

No 772 **Pawel Pietrzak:** A Type-Based Framework for Locating Errors in Constraint Logic Programs, 2002, ISBN 91-7373-422-5.

No 758 **Erik Berglund:** Library Communication Among Programmers Worldwide, 2002, ISBN 91-7373-349-0.

No 774 **Choong-ho Yi:** Modelling Object-Oriented Dynamic Systems Using a Logic-Based Framework, 2002, ISBN 91-7373-424-1.

No 779 **Mathias Broxvall:** A Study in the Computational Complexity of Temporal Reasoning, 2002, ISBN 91-7373-440-3.

No 793 **Asmus Pandikow:** A Generic Principle for Enabling Interoperability of Structured and Object-Oriented Analysis and Design Tools, 2002, ISBN 91-7373-479-9.

No 785 **Lars Hult:** Publika Informationstjänster. En studie av den Internetbaserade encyklopedins bruksegenskaper, 2003, ISBN 91-7373-461-6.

No 800 **Lars Taxén:** A Framework for the Coordination of Complex Systems´ Development, 2003, ISBN 91-7373-604-X.

No 808 **Klas Gäre:** Tre perspektiv på förväntningar och förändringar i samband med införande av informationssystem, 2003, ISBN 91-7373-618-X.

No 821 **Mikael Kindborg:** Concurrent Comics - programming of social agents by children, 2003, ISBN 91-7373-651-1.

No 823 **Christina Ölvingson:** On Development of Information Systems with GIS Functionality in Public Health Informatics: A Requirements Engineering Approach, 2003, ISBN 91-7373-656-2.

No 828 **Tobias Ritzau:** Memory Efficient Hard Real-Time Garbage Collection, 2003, ISBN 91-7373-666-X.

No 833 **Paul Pop:** Analysis and Synthesis of Communication-Intensive Heterogeneous Real-Time Systems, 2003, ISBN 91-7373-683-X.

No 852 **Johan Moe:** Observing the Dynamic Behaviour of Large Distributed Systems to Improve Development and Testing – An Empirical Study in Software Engineering, 2003, ISBN 91-7373-779-8.

No 867 **Erik Herzog:** An Approach to Systems Engineering Tool Data Representation and Exchange, 2004, ISBN 91-7373-929-4.

No 872 **Aseel Berglund:** Augmenting the Remote Control: Studies in Complex Information Navigation for Digital TV, 2004, ISBN 91-7373-940-5.

No 869 **Jo Skåmedal:** Telecommuting’s Implications on Travel and Travel Patterns, 2004, ISBN 91-7373-935-9.

No 870 **Linda Askenäs:** The Roles of IT - Studies of Organising when Implementing and Using Enterprise Systems, 2004, ISBN 91-7373-936-7.

No 874 **Annika Flycht-Eriksson:** Design and Use of Ontologies in Information-Providing Dialogue Systems, 2004, ISBN 91-7373-947-2.

No 873 **Peter Bunus:** Debugging Techniques for Equation-Based Languages, 2004, ISBN 91-7373-941-3.

No 876 **Jonas Mellin:** Resource-Predictable and Efficient Monitoring of Events, 2004, ISBN 91-7373-956-1.

No 883 **Magnus Bång:** Computing at the Speed of Paper: Ubiquitous Computing Environments for Healthcare Professionals, 2004, ISBN 91-7373-971-5.

No 882 **Robert Eklund:** Disfluency in Swedish human-human and human-machine travel booking dialogues, 2004, ISBN 91-7373-966-9.

No 887 **Anders Lindström:** English and other Foreign Linguistic Elements in Spoken Swedish. Studies of Productive Processes and their Modelling using Finite-State Tools, 2004, ISBN 91-7373-981-2.

No 889 **Zhiping Wang:** Capacity-Constrained Production-inventory systems - Modelling and Analysis in both a traditional and an e-business context, 2004, ISBN 91-85295-08-6.

No 893 **Pernilla Qvarfordt:** Eyes on Multimodal Interaction, 2004, ISBN 91-85295-30-2.

No 910 **Magnus Kald:** In the Borderland between Strategy and Management Control - Theoretical Framework and Empirical Evidence, 2004, ISBN 91-85295-82-5.

No 918 **Jonas Lundberg:** Shaping Electronic News: Genre Perspectives on Interaction Design, 2004, ISBN 91-85297-14-3.

No 900 **Mattias Arvola:** Shades of use: The dynamics of interaction design for sociable use, 2004, ISBN 91-85295-42-6.

No 920 **Luis Alejandro Cortés:** Verification and Scheduling Techniques for Real-Time Embedded Systems, 2004, ISBN 91-85297-21-6.

No 929 **Diana Szentivanyi:** Performance Studies of Fault-Tolerant Middleware, 2005, ISBN 91-85297-58-5.

No 933 **Mikael Cäker:** Management Accounting as Constructing and Opposing Customer Focus: Three Case Studies on Management Accounting and Customer Relations, 2005, ISBN 91-85297-64-X.

No 937 **Jonas Kvarnström:** TALplanner and Other Extensions to Temporal Action Logic, 2005, ISBN 91-85297-75-5.

No 938 **Bourhane Kadmiry:** Fuzzy Gain-Scheduled Visual Servoing for Unmanned Helicopter, 2005, ISBN 91-85297-76-3.

No 945 **Gert Jervan:** Hybrid Built-In Self-Test and Test Generation Techniques for Digital Systems, 2005, ISBN 91-85297-97-6.

No 946 **Anders Arpteg:** Intelligent Semi-Structured Information Extraction, 2005, ISBN 91-85297-98-4.

No 947 **Ola Angelsmark:** Constructing Algorithms for Constraint Satisfaction and Related Problems - Methods and Applications, 2005, ISBN 91-85297-99-2.

No 963 **Calin Curescu:** Utility-based Optimisation of Resource Allocation for Wireless Networks, 2005, ISBN 91-85457-07-8.

No 972 **Björn Johansson:** Joint Control in Dynamic Situations, 2005, ISBN 91-85457-31-0.

No 974 **Dan Lawesson:** An Approach to Diagnosability Analysis for Interacting Finite State Systems, 2005, ISBN 91-85457-39-6.

No 979 **Claudiu Duma:** Security and Trust Mechanisms for Groups in Distributed Services, 2005, ISBN 91-85457-54-X.

No 983 **Sorin Manolache:** Analysis and Optimisation of Real-Time Systems with Stochastic Behaviour, 2005, ISBN 91-85457-60-4.

No 986 **Yuxiao Zhao:** Standards-Based Application Integration for Business-to-Business Communications, 2005, ISBN 91-85457-66-3.

No 1004 **Patrik Haslum:** Admissible Heuristics for Automated Planning, 2006, ISBN 91-85497-28-2.

No 1005 **Aleksandra Tešanovic:** Developing Reusable and Reconfigurable Real-Time Software using Aspects and Components, 2006, ISBN 91-85497-29-0.

No 1008 **David Dinka:** Role, Identity and Work: Extending the design and development agenda, 2006, ISBN 91-85497-42-8.

No 1009 **Iakov Nakhimovski:** Contributions to the Modeling and Simulation of Mechanical Systems with Detailed Contact Analysis, 2006, ISBN 91-85497-43-X.

No 1013 **Wilhelm Dahllöf:** Exact Algorithms for Exact Satisfiability Problems, 2006, ISBN 91-85523-97-6.

No 1016 **Levon Saldamli:** PDEModelica - A High-Level Language for Modeling with Partial Differential Equations, 2006, ISBN 91-85523-84-4.

No 1017 **Daniel Karlsson:** Verification of Component-based Embedded System Designs, 2006, ISBN 91-85523-79-8

No 1018 **Ioan Chisalita:** Communication and Networking Techniques for Traffic Safety Systems, 2006, ISBN 91-85523-77-1.

No 1019 **Tarja Susi:** The Puzzle of Social Activity - The Significance of Tools in Cognition and Cooperation, 2006, ISBN 91-85523-71-2.

No 1021 **Andrzej Bednarski:** Integrated Optimal Code Generation for Digital Signal Processors, 2006, ISBN 91-85523-69-0.

No 1022 **Peter Aronsson:** Automatic Parallelization of Equation-Based Simulation Programs, 2006, ISBN 91-85523-68-2.

No 1030 **Robert Nilsson:** A Mutation-based Framework for Automated Testing of Timeliness, 2006, ISBN 91-85523-35-6.

No 1034 **Jon Edvardsson:** Techniques for Automatic Generation of Tests from Programs and Specifications, 2006, ISBN 91-85523-31-3.

No 1035 **Vaida Jakoniene:** Integration of Biological Data, 2006, ISBN 91-85523-28-3.

No 1045 **Genevieve Gorrell:** Generalized Hebbian Algorithms for Dimensionality Reduction in Natural Language Processing, 2006, ISBN 91-85643-88-2.

No 1051 **Yu-Hsing Huang:** Having a New Pair of Glasses - Applying Systemic Accident Models on Road Safety, 2006, ISBN 91-85643-64-5.

No 1054 **Åsa Hedenskog:** Perceive those things which cannot be seen - A Cognitive Systems Engineering perspective on requirements management, 2006, ISBN 91-85643-57-2.

No 1061 **Cécile Åberg:** An Evaluation Platform for Semantic Web Technology, 2007, ISBN 91-85643-31-9.

No 1073 **Mats Grindal:** Handling Combinatorial Explosion in Software Testing, 2007, ISBN 978-91-85715-74-9.

No 1075 **Almut Herzog:** Usable Security Policies for Runtime Environments, 2007, ISBN 978-91-85715-65-7.

No 1079 **Magnus Wahlström:** Algorithms, measures, and upper bounds for Satisfiability and related problems, 2007, ISBN 978-91-85715-55-8.

No 1083 **Jesper Andersson:** Dynamic Software Architectures, 2007, ISBN 978-91-85715-46-6.

No 1086 **Ulf Johansson:** Obtaining Accurate and Compre-hensible Data Mining Models - An Evolutionary Approach, 2007, ISBN 978-91-85715-34-3.

No 1089 **Traian Pop:** Analysis and Optimisation of Distributed Embedded Systems with Heterogeneous Scheduling Policies, 2007, ISBN 978-91-85715-27-5.

No 1091 **Gustav Nordh:** Complexity Dichotomies for CSP-related Problems, 2007, ISBN 978-91-85715-20-6.

No 1106 **Per Ola Kristensson:** Discrete and Continuous Shape Writing for Text Entry and Control, 2007, ISBN 978-91-85831-77-7.

No 1110 **He Tan:** Aligning Biomedical Ontologies, 2007, ISBN 978-91-85831-56-2.

No 1112 **Jessica Lindblom:** Minding the body - Interacting so-cially through embodied action, 2007, ISBN 978-91-85831-48-7.

No 1113 **Pontus Wärnestål:** Dialogue Behavior Management in Conversational Recommender Systems, 2007, ISBN 978-91-85831-47-0.

No 1120 **Thomas Gustafsson:** Management of Real-Time Data Consistency and Transient Overloads in Embedded Systems, 2007, ISBN 978-91-85831-33-3.

No 1127 **Alexandru Andrei:** Energy Efficient and Predictable Design of Real-time Embedded Systems, 2007, ISBN 978-91-85831-06-7.

No 1139 **Per Wikberg:** Eliciting Knowledge from Experts in Modeling of Complex Systems: Managing Variation and Interactions, 2007, ISBN 978-91-85895-66-3.

No 1143 **Mehdi Amirijoo:** QoS Control of Real-Time Data Services under Uncertain Workload, 2007, ISBN 978-91-85895-49-6.

No 1150 **Sanny Syberfeldt:** Optimistic Replication with Forward Conflict Resolution in Distributed Real-Time Databases, 2007, ISBN 978-91-85895-27-4.

No 1155 **Beatrice Alenljung:** Envisioning a Future Decision Support System for Requirements Engineering - A Holistic and Human-centred Perspective, 2008, ISBN 978-91-85895-11-3.

No 1156 **Artur Wilk:** Types for XML with Application to Xcerpt, 2008, ISBN 978-91-85895-08-3.

No 1183 **Adrian Pop:** Integrated Model-Driven Development Environments for Equation-Based Object-Oriented Languages, 2008, ISBN 978-91-7393-895-2.

No 1185 **Jörgen Skågeby:** Gifting Technologies - Ethnographic Studies of End-users and Social Media Sharing, 2008, ISBN 978-91-7393-892-1.

No 1187 **Imad-Eldin Ali Abugessaisa:** Analytical tools and information-sharing methods supporting road safety organizations, 2008, ISBN 978-91-7393-887-7.

No 1204 **H. Joe Steinhauer:** A Representation Scheme for De-scription and Reconstruction of Object Configurations Based on Qualitative Relations, 2008, ISBN 978-91-7393-823-5.

No 1222 **Anders Larsson:** Test Optimization for Core-based System-on-Chip, 2008, ISBN 978-91-7393-768-9.

No 1238 **Andreas Borg:** Processes and Models for Capacity Requirements in Telecommunication Systems, 2009, ISBN 978-91-7393-700-9.

No 1240 **Fredrik Heintz:** DyKnow: A Stream-Based Know-ledge Processing Middleware Framework, 2009, ISBN 978-91-7393-696-5.

No 1241 **Birgitta Lindström:** Testability of Dynamic Real-Time Systems, 2009, ISBN 978-91-7393-695-8.

No 1244 **Eva Blomqvist:** Semi-automatic Ontology Construction based on Patterns, 2009, ISBN 978-91-7393-683-5.

No 1249 **Rogier Woltjer:** Functional Modeling of Constraint Management in Aviation Safety and Command and Control, 2009, ISBN 978-91-7393-659-0.

No 1260 **Gianpaolo Conte:** Vision-Based Localization and Guidance for Unmanned Aerial Vehicles, 2009, ISBN 978-91-7393-603-3.

No 1262 **AnnMarie Ericsson:** Enabling Tool Support for Formal Analysis of ECA Rules, 2009, ISBN 978-91-7393-598-2.

No 1266 **Jiri Trnka:** Exploring Tactical Command and Control: A Role-Playing Simulation Approach, 2009, ISBN 978-91-7393-571-5.

No 1268 **Bahlol Rahimi:** Supporting Collaborative Work through ICT - How End-users Think of and Adopt Integrated Health Information Systems, 2009, ISBN 978-91-7393-550-0.

No 1274 **Fredrik Kuivinen:** Algorithms and Hardness Results for Some Valued CSPs, 2009, ISBN 978-91-7393-525-8.

No 1281 **Gunnar Mathiason:** Virtual Full Replication for Scalable Distributed Real-Time Databases, 2009, ISBN 978-91-7393-503-6.