DEGREE PROJECT IN INFORMATION AND COMMUNICATION TECHNOLOGY,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2018

Synthesis of GPU Programs from High-Level Models

ZIYUAN JIANG

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Title: Synthesis of GPU Programs from Higher-Level Models
Author: Ziyuan Jiang
Supervisor: George Ungureanu
Examiner: Ingo Sander
Thesis number: TRITA-EECS-EX-2018:5
KTH Royal Institute of Technology
School of Information and Communication Technology (ICT)

Abstract

Modern graphics processing units (GPUs) provide high-performance general purpose computation abilities. They have massive parallel architectures that are suitable for executing parallel algorithms and operations. They are also throughput-oriented devices that are optimized to achieve high throughput for stream processing. Designing efficient GPU programs is a notoriously difficult task. The ForSyDe methodology is suitable to ease the difficulties of GPU programming. The methodology encourages software development from a high level of abstraction and then transforming the abstract model to an implementation through a series of formal methods. The existing ForSyDe models support the synchronous data flow (SDF) model of computation (MoC) which is suitable for modeling stream computations and is good for synthesizing efficient stream processing programs. There also exists high-level design models named parallel patterns that are suitable to represent parallel algorithms and operations. The thesis studies the method of modeling parallel algorithms using parallel patterns, and explores the way to synthesize efficient OpenCL implementation on GPUs for parallel patterns. The thesis also tries to enable the integration of parallel patterns into the ForSyDe SDF model in order to model stream parallel operations. An automation library that helps designing stream programs for parallel algorithms targeting GPUs is purposed in the thesis project. Several experiments are performed to evaluate the effectiveness of the proposed library regarding implementations of the high-level model.

Sammanfattning

Moderna grafikbehandlingsenheter (GPU) tillhandahåller högpresterande generella syftes-beräkningsförmågor. De har massiva parallella arkitekturer som är lämpliga för att utföra parallella algoritmer och operationer. De är också stream-inriktade enheter som är optimerade för att uppnå hög streaming för streaming-behandling. Att utforma effektiva GPU-program är en notoriskt svårt uppgift. ForSyDe-metoden är lämplig för att underlätta svårigheterna med GPU-programmering. Metodiken uppmuntrar mjukvaruutveckling från en hög nivå av abstraktion för att sedan omvandla den abstrakta modellen till en implementering genom en rad formella metoder. De befintliga ForSyDe-modellerna stöder synkron dataflöde (SDF) modell av beräkning (MoC) som är lämplig för modellering av streaming-beräkningar och är bra för att syntetisera effektiv streaming-bearbetningsprogram. Det finns också högkvalitativa designmodeller som kallas parallella mönster vilka är lämpliga för att representera parallella algoritmer och operationer. Avhandlingen analyserar metoden för modellering av parallella algoritmer med parallella mönster, och utforskar sättet att syntetisera effektiv OpenCL-implementering för GPU för parallella mönster. Avhandlingen försöker även att möjliggöra integration av parallella mönster i ForSyDe SDF-modellen för att modellera streaming parallella operationer. Ett automationsbibliotek som hjälper till att designa stream-program för parallella algoritmer som riktar sig mot GPU:er är avsedda för avhandlingsprojektet. Flera experiment utförs för att utvärdera effektiviteten hos det föreslagna biblioteket avseende implementering av högnivåmodellen.

Acknowledgements

Several people have helped me and supported me during the thesis project. I would like to express my gratitude to them in this section.

First of all, I would like to thank my examiner and mentor Ingo Sander for providing many feedbacks on both the design of my program and the writing of the thesis report. The papers that Ingo recommended also helped me quickly understand the background knowledge. He also provided me with several excellent materials that helped me improve my skills on reading papers and writing reports, which are very beneficial during the thesis project. It was also very kind of him to check my progress frequently and gave me advice when I encounter problems.

I also want to thank my supervisor George Ungureanu. He pointed out several good directions that I can look into at the beginning of the thesis project. His previous works inspired me a lot. He also offered me lots of help on getting familiar with several high-level computation models. I want to thank him for his quick replies to my doubts and problems.

I want to express my gratitude to my parents for their support during my study in Sweden. The development of my knowledge and skills during the master program is simply not possible without their support. I would also like to thank my girlfriend Liping Gu. She always believes in me. Her letters and phone calls always brought me much happiness and encouragement to overcome so many difficulties during the project.

Contents

List of Terms and Acronyms

AoS array of structures. 6, 25, 35

API application programming interface. 11, 19, 20, 24, 42, 48, 53, 54, 56, 70, 77

CPU Central Processing Unit. 12, 62, 69–74, 77, 78

FFT fast fourier transform. 6, 38–40, 51

FIFO first in, first out. 29, 56

ForSyDe Formal System Design. 1, 5, 11, 12, 27–29, 31, 41, 42

FPGA field-programmable gate array. 19

GPGPU general-purpose computing on graphics processing units. 5, 11, 17, 59

GPU graphics processing unit. 1, 5, 11–13, 17–20, 23–26, 57, 59, 69–74, 77–79

MoC model of computation. 1, 12, 27, 29, 41

NDRange N-Dimensional Range. 20

P2CL Patterns-to-OpenCL (P2CL) is the library developed in this thesis project which helps create OpenCL stream programs using data parallel patterns. 7, 12, 13, 27, 31, 45, 46, 48, 53, 54, 56, 57, 59, 62, 69, 70, 73, 74, 79, 80

SDF synchronous data flow. 1, 8, 12, 29, 54, 56, 77–79

SIMT single instruction, multiple threads. 18, 19

SM streaming multiprocessor. 17, 20

SoA structure of arrays. 6, 25, 35, 36, 46, 49

SoC System on a Chip. 69, 74

SP streaming processor. 17, 18, 20

XML Extensible Markup Language. 8, 45–47, 51, 53, 54, 79

Chapter 1

Introduction

This chapter gives a brief overview of the thesis. The problem the thesis tries to solve is presented in the first section, which is followed by a short summary of the accomplished work in the thesis and its limitations. Finally, the structure of this report is presented in the last section.

1.1 Motivation

As the performance of single processor reaches its limit, the industry put more effort into multicore devices to utilize data-level parallelism and thread-level parallelism [1]. Graphics processing unit (GPU) is one of the parallel-structured devices which is originally designed to accelerate rendering of 3D graphics. Its multiprocessor structure makes it also suitable for exploiting data-level parallelism for general purpose computing.

However, designing efficient programs for general-purpose computing on graphics processing units (GPGPU) is a notoriously difficult task. Firstly, the parallelism of the target algorithm is required to be fully analyzed before designing. Secondly, programmers need to have thorough knowledge of the GPU architecture and the programming model of the selected application programming interfaces (APIs) in order to avoid enormous unnecessary overheads and apply optimizations. Besides that, although there exist APIs that provide functional portability to multiple platforms, the performance portability is not guaranteed. Additional adjustments might be necessary to achieve high performance on different devices.

The methodology of Formal System Design (ForSyDe) is suitable to ease these difficulties. The methodology encourages starting software development from a high level of abstraction and then transforming the abstract model to an implementation through a series of formal methods [2]. It allows programmers to focus on the what the system should do rather than how [3]. This transformation can be achieved through a set of synthesis and verification tools, which not only reduce development time and cost but also ensure correctness and efficiency of implementations through the approach of correct-by-construction.

1.2 Contribution

This thesis explores transformation from high-level models down to OpenCL implementations on GPUs. The existing synchronous data flow (SDF) model of computation (MoC) in the ForSyDe modeling framework is suitable for GPU programming. This is because GPUs are designed in a way that throughput is emphasized more than latency [4], and the SDF MoC naturally suits stream processing. However, in order to explicitly express parallel operations, another high-level model named parallel pattern is also used in this thesis. This thesis provides a way to add support for parallel patterns inside a SDF graph in order to utilize both the parallel structure and the throughput-oriented design of GPUs. This thesis provides a methodology for generating optimized software for GPUs from the high-level model, captured in the design of an automation tool named Patterns-to-OpenCL (P2CL). Due to the limited time frame of the thesis project, the tool currently focuses on efficient implementations of parallel patterns. It embeds operations described by parallel patterns inside one single SDF process , which can interact with data flows fed from the Central Processing Unit (CPU). The plan for supporting complete SDF networks is described in Chapter 10.

The tool has the following features:

- Recognize and parse a script that describes a stream processing system using parallel patterns.
- Instead of statically generating codes, the tool allows users to load the script describing a system at runtime, feed data flow to the system and get a sequence of results.
- Efficient kernel code and execution plan for GPUs for several parallel patterns are automatically generated and the kernel code can be exported for development of other programs.
- In the case when new input data arrives faster than the computation, several instances of computation can concurrently run on a single GPU to hide memory latencies and provide higher throughput.

1.3 Structure of the Thesis

This thesis report is divided into three parts. The first part describes the background studies including the description of parallel patterns, the ForSyDe project and several other projects that targeting designing GPU programs from high level models. An introduction to GPU programming and OpenCL is also provided in this part. The second part contains information on how P2CL should be used and the implementation details of P2CL. The last part demonstrates evaluations of the tool. The limitations and visions for future development are also discussed in this part.

Part I

UNDERSTANDING THE PROBLEM

Chapter 2

GPGPU and OpenCL

The GPU parallel programming model is different from the sequential execution model used for CPUs. In order to generate efficient code, one has to understand the programming model. Its relationship to the detailed architecture of the GPU is also vital for program optimization. This section briefly describes the GPU architecture together with the OpenCL programming model. Several common good practices and pitfalls of GPU programming are also introduced which are used as guidelines in the development of the automation tool.

2.1 GPU Architecture

GPUs are originally designed for display generation. Throughout the years, its architecture has been evolved from hard-wired graphics pipelines to massive highly-programmable processors [5]. Modern GPUs use the same type of processors to perform different stages of graphics processing as well as general purpose computing. This unified architecture allows better load balancing and scalability since all the functions can use the whole processor array [5].

As shown in Figure 2.1, a modern GPU consists of massive streaming processor (SP) cores. Each SP core is capable of managing multiple concurrent threads. Their states are managed inside the SP cores, thus no expensive register saving and restoring mechanism is performed between those threads. The SP cores are organized into several streaming multiprocessors (SMs). Besides the SP cores, each SM also includes special function units, instruction and constant caches, a multithreaded instruction unit, and a shared memory [5]. SMs are grouped into texture/processor clusters (TPC), which control SMs and provide a lower hierarchy of caches.

Although modern GPUs have a massively parallel structure, in order to manage tasks whose data set is larger than the number of processors, or to execute

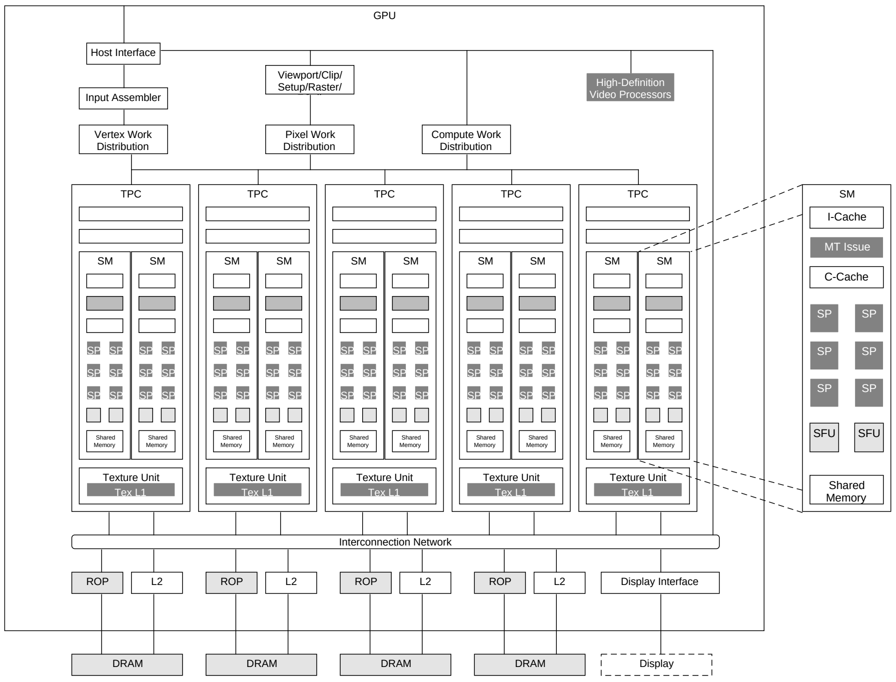


Figure 2.1: GPU Architecture Adapted from [5]

multiple different tasks at the same time, GPUs must be able to schedule concurrent threads. A single instruction, multiple threads (SIMT) mechanism is implemented in many GPUs. In this mechanism, parallel threads that execute the same instructions are grouped into warps. It is also called wavefront in AMD's terminology. The size of warps is a fixed value on a given architecture. GPUs schedule and execute several warps concurrently. At each instruction issue time, a warp that is ready to execute its next instruction is selected to be issued [5]. The instruction is then broadcast to the active threads of the warp to be executed [5]. As shown in Figure 2.2, three warps running on a GPU are illustrated. At first, warp 4 is ready to be executed. Its instruction 10 is selected and broadcasted to threads managed by several SP cores. It turns out that the instruction request memory accesses which cannot be finished in one cycle. Thus, warp 4 is not ready at the next instruction issue time. Instead, instruction 9 of warp 2 is selected to be issued. When the memory access requested by warp 4 is finished, warp 4 becomes ready again and is issued later. It is worth mentioning that although threads in one warp execute the same instruction, they are allowed to take different execution paths when conditional branch instructions are encountered. If different execution paths are taken in a warp, all threads will work through both paths and masking is applied to ensure the correct result. Therefore, stream processors can only achieve full efficiency when all the threads in a warp follow the same execution path.

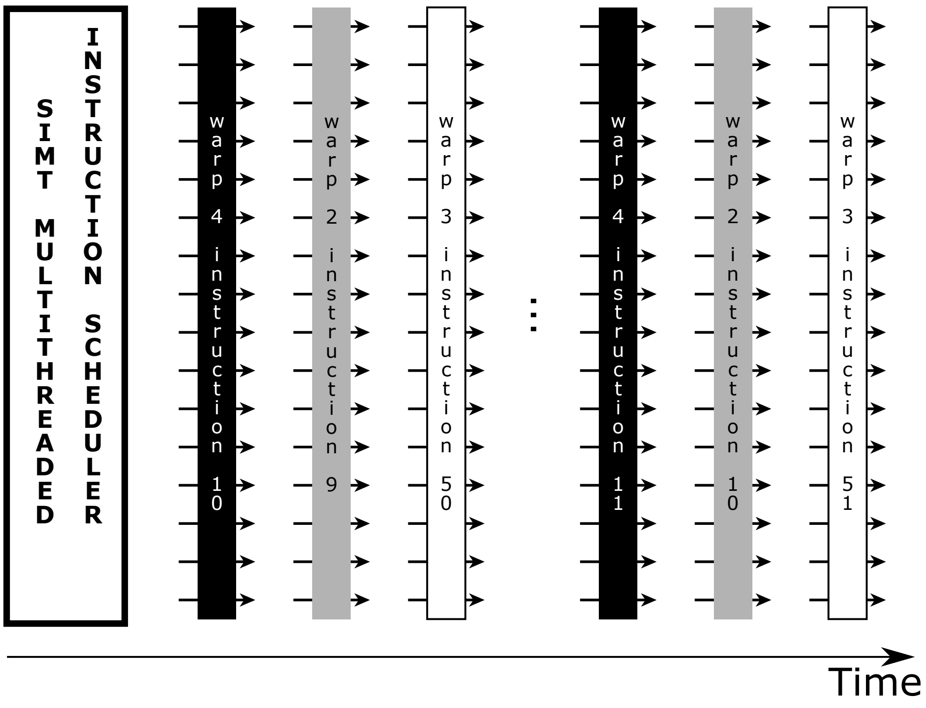


Figure 2.2: SIMT Scheduling Adapted from [5]

2.2 OpenCL Programming Model

In order to do massive parallel computing on a GPU, application programming interfaces (APIs) are provided to allow programmers to design software directly using a parallel programming model. In the earlier days of general-purpose computing on GPUs, there are only APIs featuring graphics programming. Programmers had to transform the data into graphic forms and adapt the computation into graphics operations. The creation of CUDA and OpenCL frees programmers from this conversion. While CUDA is created by NVIDIA and only used on their CUDA-enabled GPUs, OpenCL is an open standard supported by multiple vendors, targeting heterogeneous platforms which not only include GPUs from different vendors but also include field-programmable gate arrays (FPGAs) and digital signal processors (DSPs). Programming portability is emphasized in OpenCL development [6]. Thus an OpenCL program can be executed on different platforms with correct results. However, as mentioned earlier, there is no guarantee on performance portability. OpenCL hides the architecture details of platforms and provides a platform model, a memory model, and an execution model, allowing programmers to think about parallelization from the start and identify performance critical issues at an abstract level. The following paragraphs introduce the overview of these three models. Execution flow and memory consistency are subtracted and put into the later section. This thesis does not intend to include all the details of OpenCL. Readers are encouraged to read the newest OpenCL specification [7] for more information.

OpenCL Platform Model Figure 2.3 demonstrates the view of hardware in the OpenCL model. Several compute devices are connected and controlled by a host machine. Each of the compute devices may represent a GPU or other devices. The OpenCL host program needs to select devices where the compute kernel executed on at the beginning of a program. Inside each compute device, there are several compute units which can be used to model SMs inside GPUs. At the lowest level, SPs can be represented by processing elements which can process several threads of computation.

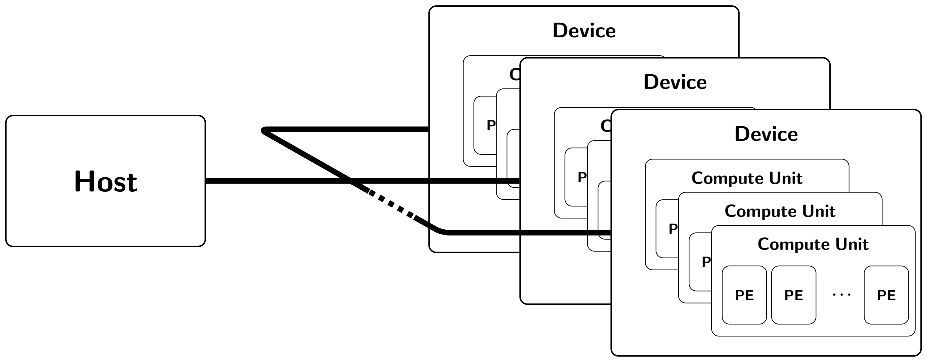


Figure 2.3: OpenCL Platform Model

OpenCL Execution Model OpenCL programs consist of the host programs that issue and manage workloads and the kernels that run on the device. The kernel programs are written in OpenCL C and are executed in parallel over a predefined N-dimensional computation domain [8]. This domain is named N-Dimensional Range (NDRange). Each element of execution in the computation domain is named a work-item which runs inside a processing element. Work-items are grouped into work-groups which fit inside compute units. At execution time, each work-item can get access to its position in the work-group and in the global domain through provided APIs. The work-group indexes are also available to be obtained. Work-items operate accordingly to that information and collectively complete the entire computation. Figure 2.4 shows the overview of the index space structure.

OpenCL Memory Model As shown in Figure 2.1, there is a hierarchy of different types of memory and caches inside a GPU. OpenCL memory model uses three levels of memory to simplify and resemble the complex memory hierarchy. The model is demonstrated in Figure 2.5. Work-items have their own private memory which is the fastest to access. Local memory is within compute units and is used to share data within work groups. At the lowest level, global and constant memory can be accessed by all the work-items. Constant memory can represent read-only memory inside a GPU, which is faster to read than global memory. Programmers are given the responsibilities to explicitly select memory regions to use and move data between regions.

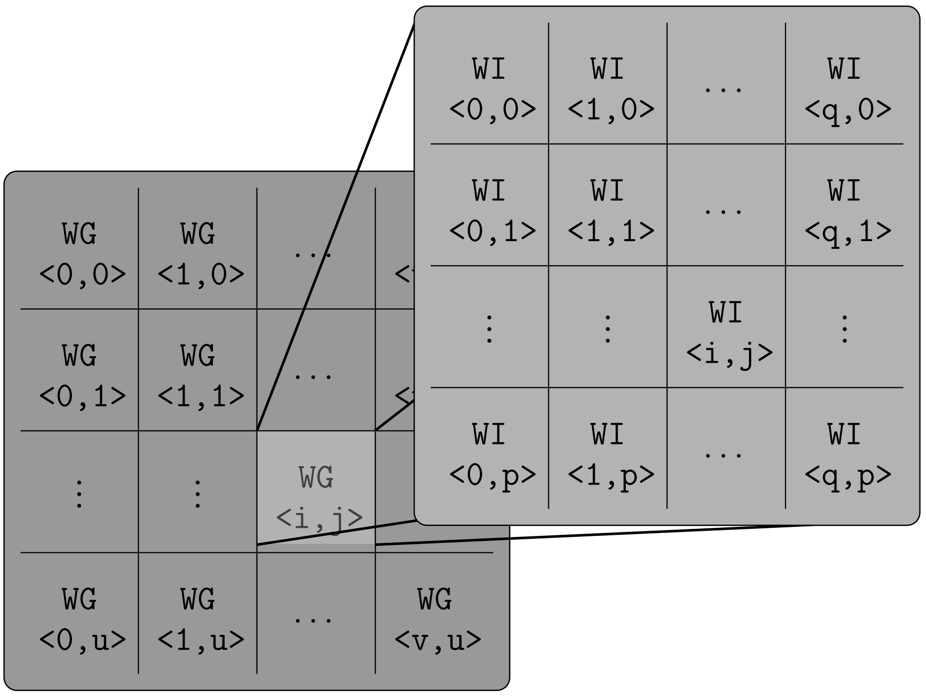


Figure 2.4: OpenCL Index Space

2.3 Communication and Synchronization

Communication and synchronization in OpenCL are only available within work-groups in a kernel invocation. Work-items are able to communicate through local or global memory with the help of synchronization functions [8]. Barrier functions ensure that all work-items within a work-group must encounter it before any of the work-items are allowed to continue [9]. The function also provides memory fences that ensure correct ordering to local or global memory. In the newer OpenCL C 2.0 standard [10], there are more communication functions along with some common parallel pattern operations that will be introduced later.

2.4 OpenCL Application Workflow and Kernel Functions

With the OpenCL programming model in mind, it is easier to understand an OpenCL program. A typical OpenCL application workflow can be summarized into the following steps:

1. Query and select the platform and devices to create a context object and

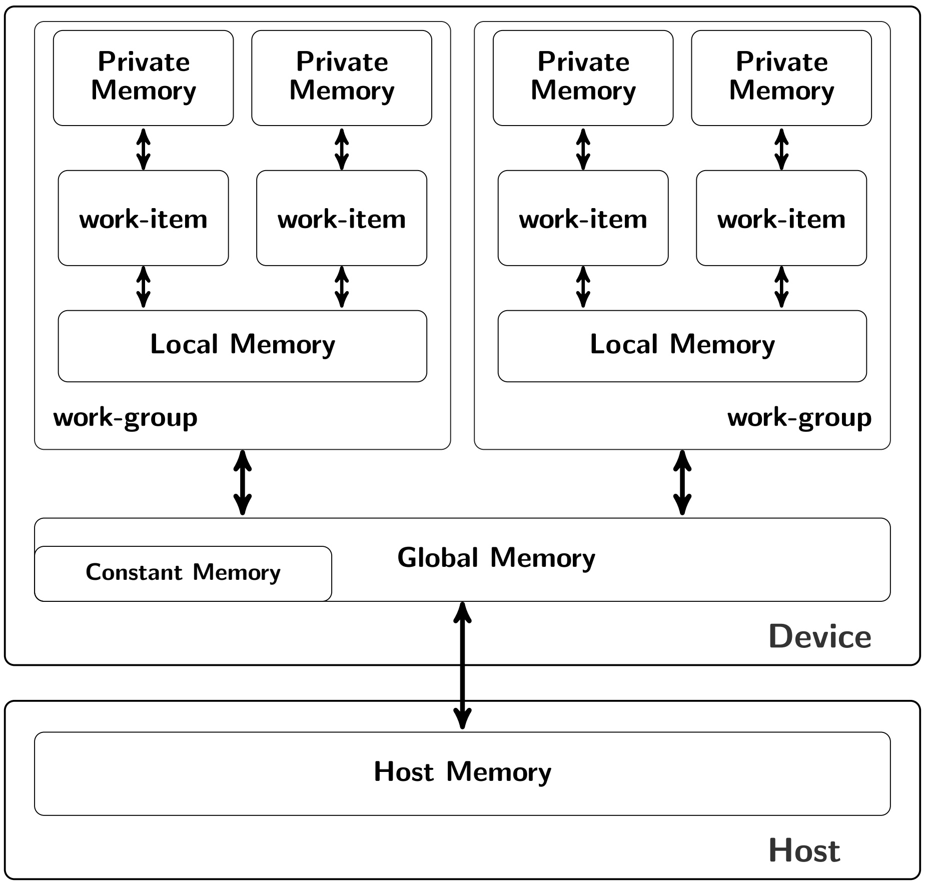


Figure 2.5: OpenCL Memory Model

command queues.

2. Build kernel programs for each device.
3. Create kernel function objects.
4. Create memory objects and assign them along with other parameters to the kernel functions.
5. Run the kernels functions in NDRange domain and collect results.

OpenCL APIs is originally in C language. An efficient C++ wrapper is also provided to allow simpler software development using C++.

A simple kernel program that performs vector addition is shown in Listing 2.1. Kernels functions that a host program may enqueue is prefixed with `"__kernel"`. Only they can be enqueued to a command queue. The `"__global"` qualifier together with `"__local"`, `"__private"`, and `"__constant"` are used to specify which memory space the buffer or array is located. A pointer cannot point to a buffer or array with different address space qualifier. The "get_global_id" function is the API for getting the global position of a work-item in the NDRange. For this kernel function, each work-item fetches the values from *i*-th position in A and B, and put the results back to the corresponding position in C.

```
__kernel void vadd(
  __global float* A,
  __global float* B,
  __global float* C)
{
  int i = get_global_id(0);
  C[i] = A[i] + B[i];
}
```

Listing 2.1: Vector Addition Kernel

2.5 OpenCL Optimization Tips

Although different GPU vendors implement OpenCL in different ways. There still exist good practices in OpenCL programming that benefit the performance on most GPUs. Several of these optimization tips are introduced here while readers are encouraged to read more about this topic from guides written by various GPU vendors.

2.5.1 Global Memory Coalescing

Global memory read and write are expensive operations on GPU. Therefore, memory coalescing is one of the most important performance considerations in GPU programming [11]. On modern GPU architectures, accesses to global memory requested by a portion of threads within a warp are grouped as one transaction if certain requirements are met. Several access patterns and their impacts on the global memory latency are illustrated below.

- The simplest pattern that guarantees coalesced memory access is the case where the k-th threads in a warp access the k-th word in a segment [11]. However, it is not required for all the threads to participate [11]. Figure 2.6 shows the pattern.
- Misaligned access refers to the situation where each thread accesses memory locations in a segment with an offset. On older GPUs such as NVIDIA GeForce GTX 8800, this type of memory requests cannot be coalesced, thus they must be serialized and cost much time [11]. However, newer GPUs like NVIDIA GeForce GTX 280 can coalesce these memory requests as long as they all fall within the same aligned segment [11]. According to NVIDIA, on devices of compute capability 1.2 or higher, memory transactions requested by half warps within a 128-byte aligned segment can be coalesced. These devices can also reduce the transaction size to maximize effective bandwidth. Figure 2.7 shows the situation. In this figure, there are 16 threads accessing a 64Bytes (16 words) aligned segment. All

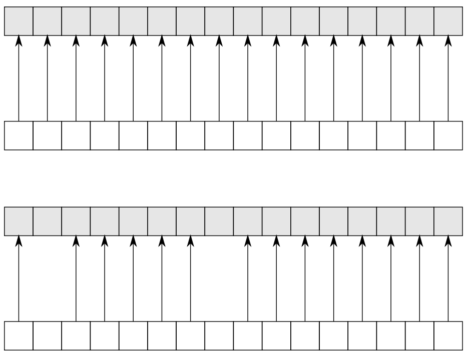


Figure 2.6: Sequential Access

the requests fall into the same 16-word segment. Thus only one 64-Byte memory transaction is required for this case on newer GPUs.

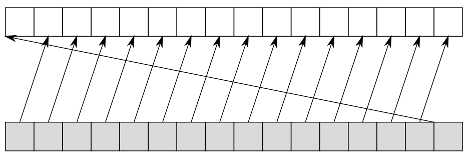


Figure 2.7: Misaligned Access

- Strided access is the case when threads access memory locations with a unified or non-unit stride. In Figure 2.8, 16 threads are accessing memory locations with a stride of 2. Like the misaligned access, as long as the memory locations fit in a 128-byte aligned segment, modern GPUs can coalesce them. However, since only one out of two words in the segment is useful, the effective bandwidth is low. In the case when each thread in the group need to access consecutive words, if the data is loaded word by word, there will be several of these coalesced strided memory access and each of them with a lot of wasted bandwidth. Vector load and store is recommended to be used by AMD OpenCL Optimization Guide [12] and Adreno OpenCL Programming Guide [13]. On AMD and Adreno GPUs, using vector APIs for 4 words leads to the requests to be coalesced. There is no benefit using vector APIs for 8 words and 16 words over a multiple of vector accesses for 4 words [13].

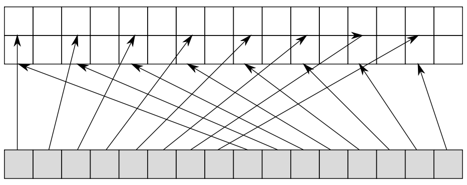


Figure 2.8: Strided Access

2.5.2 Bank Conflicts

As mentioned earlier, local memory is available for work-items to communicate within their work-groups. They are fast to access compared to the global memory. On GPUs, accesses to consecutive local memory locations are handled by different memory banks. For example, if the number of banks in a GPU is 8 and there is a block of local memory that is aligned to 8 words, accesses to word 0 is handled by bank 0 and accesses to word 1 is handled by bank 1. Word 8 is warped around and mapped to bank 0 again. This example of bank mapping is shown in Figure 2.9. For a NVIDIA GPU with compute capability 2.0, shared memory has 32 banks, with each of them mapped to a consecutive 32-bit word [14].

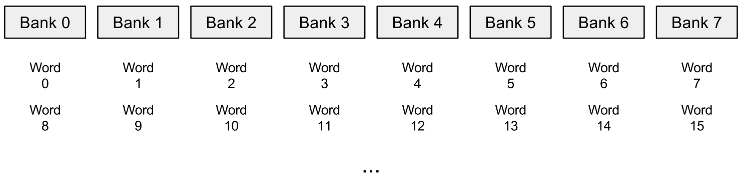


Figure 2.9: Memory Banks Mapping

These banks are able to serve local memory accesses simultaneously. However, if several work-items in a work-group try to access memory locations that mapped to the same bank, the transactions must be serialized. This is an important issue to consider when work-items are accessing a column of data in a row-major order matrix. If the width of the matrix is a multiple of the number of banks, every memory location in a column is mapped to one memory bank. The general techniques to avoid bank conflicts in this case is usually allocating one extra word for every row, so that the data in the same column mapped to different banks. Thus, bank conflicts are resolved.

2.6 Performance Portability and Autotuning

Owing to various structures and specifications of GPUs, a fixed kernel cannot achieve maximum performance on all devices. Several factors in kernel designing might affect the performance on different platforms. Lee, Joo Hwan et al. [15] evaluated effects of several aspects of OpenCL programs executed on multi-core CPUs and GPUs. The work of [16] also summarized several performance critical factors for GPU programs. They are shown as follows.

- Tiling Size for block-based algorithm
- Data Layout
  - column-major or row-major
  - AoS or SoA

- Caching and Prefetching
- Thread Data Mapping
  - work-item/ work-group size
- Operation-Specific Tuning
  - usage of instrinsic functions for specific architectures

In order to take those aspects into consideration and provide performance portability for different platforms, autotuning has been introduced to GPU computing. The idea is to generate multiple kernel versions which implement the same algorithm optimized for different architectures and heuristically select the best-performing one [6].

Chapter 3

ForSyDe

The goal of this thesis is to enable the integration of P2CL into the ForSyDe framework. This chapter briefly describes the ForSyDe framework in order for the reader to understand the methodology and the model used in this thesis. For a more detailed description of the ForSyDe framework, the reader is advised to consult [2] and [17].

3.1 Introduction

The high level of abstraction enables designers to have a clear overview of the system and data flow, allowing simpler identification of optimization and opportunities for better decisions [18]. Keutzer et al. state that a design methodology that addresses complex systems must start at high levels of abstraction in order to be efficient [19]. ForSyDe is such a methodology that provides a formal base in the form of the theory of models of compuation (MoCs), targeting designs of heterogeneous embedded and cyber-physical systems [20]. The main objective of ForSyDe is to move system design from the implementation into the functional domain [2]. This allows designers to specify the system functionality using high-level models and then transform the high-level model to an implementation model with the help of design transformational methods. Through this tranformation and refinement, the implementation model has the same semantics as the initial model but is more detailed and optimized for implementation.

ForSyDe supports two modeling languages. The functional language Haskell is an elegant choice because the language is free from side effects and it naturally supports many concepts of ForSyDe such as higher order functions and lazy evaluations. There are two libraries available in Haskell ForSyDe. The shallow-embedded library supports different types of MoCs. It is a rapid-prototyping framework used only for simulation. The deep-embedded library [21] supports both simulation and transformation. Models specified in this library can be used to synthesis VHDL code for hardware synthesis or to generate GraphML graphs that can be used for other analysis and synthesis backend tools. SystemC is another supported language of Formal System Design (ForSyDe) [22]. It is a library of C++ that provides event-driven interfaces with the purpose of co-simulating and validating hardware-software systems at a high level of abstraction [3]. To accord with ForSyDe principles, restricted features of SystemC are used. SystemC ForSyDe can generate an intermediate representation named ForSyDe-XML based on XML and C++ files. Similar to the GraphML, this intermediate representation is used by backend tools.

3.2 The Modeling Framework

Figure 3.1 shows the structure of the ForSyDe modeling framework. In ForSyDe, a system is modeled as a hierarchical network of concurrent processes [2]. There are two types of process. Leaf processes are created directly from process constructors which take side-effect-free functions and initial values as input [22]. Composite processes are created by composing processes together [22]. There is no global state in the system. Thus, processes communicate through signals [2]. Since ForSyDe supports multiple models of computation, domain interfaces are provided to allow communications between models.

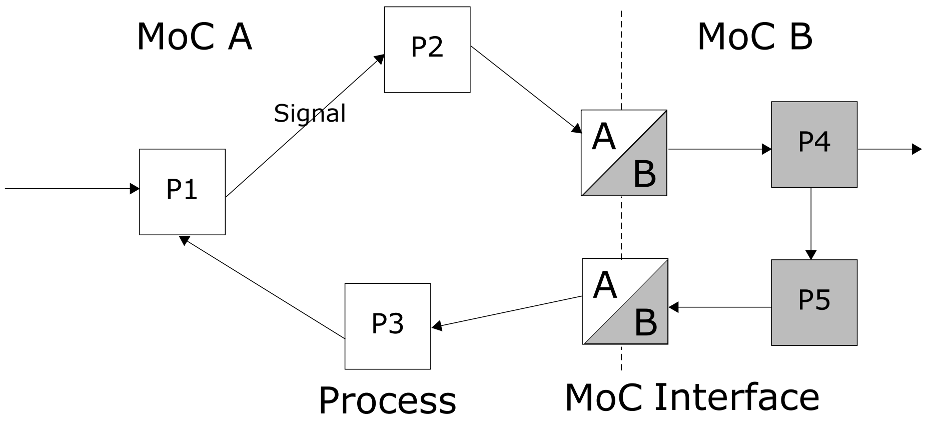


Figure 3.1: ForSyDe Modeling Framework Adapted from [20]

A signal is a sequence of events marked with a tag and a value. The tag can be used to model physical time, the order of events or other properties of the computational model [17]. The tag of the signal is either implicitly given or explicitly specified. The value type of one signal must be consistent with all the events in it.

3.3 Models of Computation

ForSyDe currently provide libraries for the several MoCs, which are Synchronous MoC, synchronous data flow (SDF) MoC, Discrete-Event MoC, and Continuous Time MoC. This thesis have particular insterests for the synchronous data flow MoC.

This ForSyDe SDF model follows the definition of synchronous data flow graph [23]. An example of SDF graphs is shown in Figure 3.2. This model consists of process (actor) nodes which represent computations and arcs which stands for FIFO buffers.

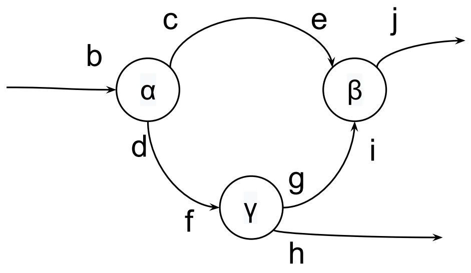


Figure 3.2: a synchronous data flow Example Adapted from [23]

This model is good for signal processing algorithms because signal processing algorithms naturally fit in data flow graphs. The synchronous data flow is a special data flow where the amount of data comsumed and produced by the data flow node is fixed for each input and output[23]. An actor can be invoked when there is enough input data. Each invocation of an actor generates fixed amount of data to output buffers. This model have a high level of analyzability [17]. Using this model, the schedules and required buffer size of a system can be determined at compile time [23]. Efficient implementations with different optimization goals can be achieved through different scheduling plan [24].

Chapter 4

Patterns

Parallel programming has several generic program structures, called skeletons or patterns[25]. They represent parallel algorithms in an abstract form and can be used as components for building parallel programs [25]. There are two types of parallel patterns namely task-parallel patterns and data-parallel patterns [25]. Task-parallel patterns are used for model execution of several tasks [25]. Data-parallel patterns partition the data and performing computations simultaneously [25]. The idea behind programming using parallel patterns is similar to ForSyDe. Programmers are encouraged to focus on the computation problem and leave the actual organization of parallelism to an automation tool [25]. This thesis mainly focuses on several data parallel patterns for lists. In order to have additional expressiveness, the thesis also uses some compositional patterns for hierarchically composing algorithms with operations created by data parallel patterns. This chapter describes those patterns that are used in P2CL. Many of the used patterns are modified from commonly known patterns. However, some constraints and extensions are introduced for the purpose of better expressiveness and simplicity. For a complete description of commonly used parallel patterns, readers may refer to [25] and [26].

4.1 Data Parallel Patterns

This section describes the data parallels used in P2CL.

4.1.1 Map Pattern

The map pattern is a fundamental data parallel pattern that applies a function f to each element of the list. It can be expressed by the following equation.

$$map\ (f)\ [a_0,\ ...,\ a_{m-1}] = [f\ a_0, ..., f\ a_{m-1}] \tag{4.1}$$

As shown in Figure 4.1, the input, and output of this pattern have a one-to-one relationship. There is no communication required between these computations.

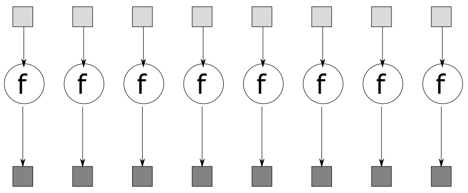


Figure 4.1: Map Pattern

4.1.2 Reduce Pattern

The reduce pattern is another commonly used pattern. It takes a binary associative operation and applies it to list elements. Assume the binary associative operation is $\oplus$. The following equation defines that pattern. A common example of this pattern is summing all the data in a list.

$$reduce(\oplus)[a_0,\ ...,\ a_{m-1}] = a_0\ \oplus ... \oplus a_{m-1} \tag{4.2}$$

This pattern shows a many-to-one relationship between the input and the output. Since the operation is binary associative, there are usually different ways of executing this pattern. Figure 4.2 shows the sequential and the parallel approaches to compute the result of a reduce pattern. The sequential approach on the left combines one element from the list at a time, while for the parallel approach, each adjacent pair of elements are combined at the same time to form a new list and recursively get the final result.

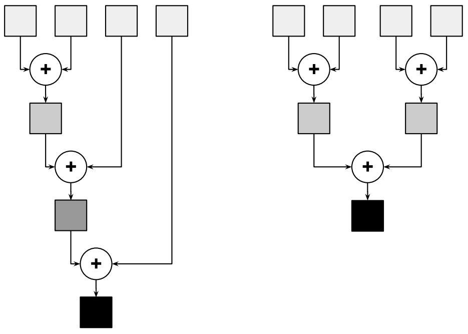


Figure 4.2: Reduce Pattern

4.1.3 Gather and Scatter Pattern

Gather and scatter are different from previously introduced patterns. They do not involve computations but mainly focus on the management of data. The gather pattern collects data from multiple locations and saves to one single location. The scatter pattern stores a collection of data to multiple places. These operations are widely used in scientific simulation and image processing applications. However, the descriptions are too general, which allows several variants [27]. Here in this thesis, the following constraints are added to these patterns so that they are easier for modeling and implementation.

1. All the locations that are used for gathering (scattering) data must be expressed as offsets relative to the location of the collective output (input).
2. These offsets must be identical between all the locations in the collective output(input).

These constraints are explained in details for each pattern in the following paragraphs.

Gather Pattern Consider a map pattern where the function is an identity function. The data in the i-th location of the input buffer is stored into the i-th location of the output buffer. This operation can also be modeled using gather pattern. Since only one element in the input buffer is gathered, the data type of the collective output is the same as the element of the input. The offset for gather, in this case, is zero, as in the i-th collective output gathers data from the i-th element of the input buffer. Take another example of gather patterns shown in Figure 4.3. There are eight data elements that are gathered into four output tuples where each of the tuples consists of two data elements. The offsets, in this case, are 0 and 4 and they are the same for all the output tuples. The first output tuple gathers data from input location 0 and 4. The index for the second output tuple is 1. Thus, the required input location is computed by adding 1 to 0 and 4, resulting 1 and 5. It is similar for the remaining output tuples.

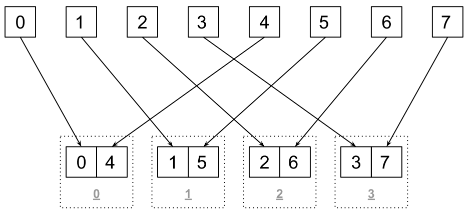


Figure 4.3: Gather Pattern

With the help of the examples above, it is easier to understand the constraints described above. The constrained gather pattern used in this thesis can be defined as the following equation.

$$\begin{aligned} &gather\ ([o_0,\ ...,\ o_{n-1}])\ [a_0,\ ...,\ a_{m-1}] \\ &= [(a_{o_0},\ ...,\ a_{o_{n-1}}),\ ...,\ (a_{m-1+o_0},\ ...,\ a_{m-1+o_{n-1}})] \end{aligned} \tag{4.3}$$

The data encaptured in a parentheses is a tuple represents one collective output.

Scatter Pattern A scatter pattern is defined as the inverse operation of a gather pattern with the same offsets. Similar to the gather pattern where the operation may not use all the data elements in the input list, an operation modeled by scatter pattern may not fill all the elements in the output list. Figure 4.4 shows a scatter operation that inverse the gather operation in Figure 4.3. Each data element in an input tuple is scattered to the location with offset 0 and 4 respectively.

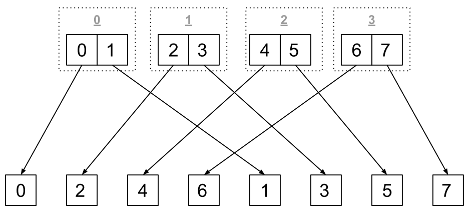


Figure 4.4: Scatter Pattern

4.1.4 Transpose Pattern

The transpose pattern is another pattern that rearranges the input data. It mimics the matrix transpose operation where a row-major matrix is transformed to be column-major. However, in this context, it is not limited to matrix operation but can also be applied to lists and other data structures [28]. Figure 4.5 shows such a situation where a transpose operation is used to separate the odd-indexed elements and the even-indexed elements. This operation is modeled as transposing a matrix with the width of two.

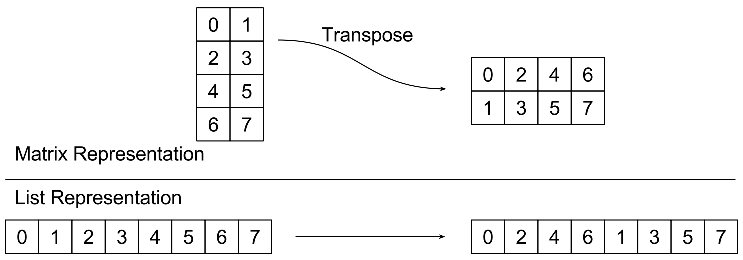


Figure 4.5: Transpose Pattern

Since this thesis only focuses on parallel patterns for lists, the definition of the transpose pattern is described as follows.

Definition Let A be a list with the length $m \times n$. Then the transpose of A with parameter m and n is another list B with the length $m \times n$. For every i and j that satisfy $i \in [0, m), j \in [0, n)$ and $i, j \in \mathbb{Z}$, the element in B with index $n \times i + j$ is equal to the element in A with index $m \times j + i$,

All the elements in the input list have an one-to-one mapping to elements in the ouput list in the transpose pattern, which is different from the gather and scatter patterns. It is also easy to see that a transpose operation can also be modeled with gather and scatter patterns. However, because the locations to read and write for gather and scatter are specified as offsets, it is convenient to use the transpose pattern for reordering of a long list.

4.1.5 Array of Structures (AoS) vs. Structures of Arrays (SoA)

Before completing the descriptions of data parallel patterns, the concept of array of structures and structure of arrays are introduced to extend the expressiveness of those patterns. Array of structures and structure of arrays are two different ways of storing sequences of structured data that contains several elements. AoS is the intuitive approach, where data structures are stored one after another. For SoA, different fields of the structured data are separated into different lists. Figure 4.6 shows such an example. The structure data type foo consists of two fields. The first field is an integer and second one is a floating point number.

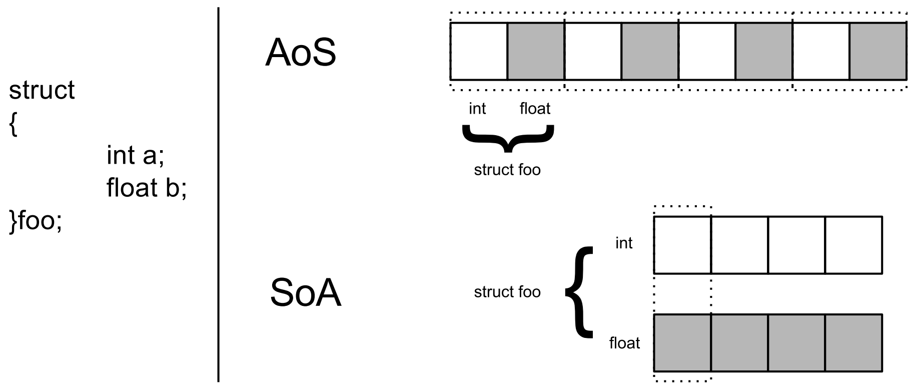


Figure 4.6: array of structures (AoS) vs. structure of arrays (SoA)

Different from some other researches [29] [30] that also study data parallel patterns, the thesis does not distinguish patterns that involve one input/output list with those that involve multiple input/output lists. One consequence is that the map pattern can take functions that have several inputs and outputs. An operation similar to Figure 4.7 can be created. It might seem contradictory to the original definition. However, since input(output) lists share the same length. They can be considered as SoA, which is functionally the same as AoS. Thus the function can be considered to apply to each element of a conceptual list, whose elements are considered as structured data that are separated in several physical lists.

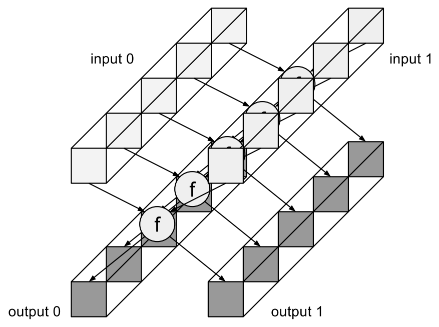


Figure 4.7: Map Pattern with Multiple Input and Output Lists

Inputs and outputs of all the data parallel patterns in this thesis follow the concept of structure of arrays, as in they are theoretically all capable of creating operations that have multiple input and output lists. However, it is currently not supported for the reduce pattern. Some additional restrictions are described in Chapter 6.

4.2 Compositional Patterns

Consider computations and data arrangements modeled by the data parallel patterns are basic operations. Then compositional patterns are patterns that are used to compose complex algorithms with these basic operations. The nesting pattern described in [26] provides a general way to create a network of operations. However, it is difficult to express a network of operations in code and GPU does not naturally support executing a network of operations. In this thesis, only several simple ways of combining operations are allowed to create a process. A general network of processes is expected to be used for the creation of more complex algorithms in future versions.

4.2.1 Operation Map Pattern

Operation map pattern is a way to extend the operations on a larger list. It takes a basic operation as a parameter and repeats the operation to form a larger composed operation. As shown in Figure 4.8, a basic operation operates on a list of 6 elements. An operation map takes this operation as a parameter and takes another parameter 3 to create a composed operation that operates on a list of 18 elements, where the same basic operation is performed on every small segment of 6 elements. In the current version of P2CL, the operation map cannot be nested, as in the only the operations created by basic data parallel patterns or a sequence of these basic operations can be used for operation map.

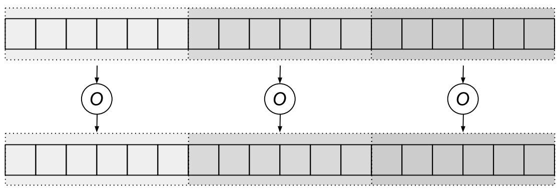


Figure 4.8: Operation Map

4.2.2 Stage-generate Pattern

The idea of the stage-generate pattern comes from the for-generate loop syntax in VHDL. The for-generate loop in VHDL is used to create compositional systems with repeated components. Within each replication of the for-generate loop, there is an identifier named generate parameter that indicates the value to be used for generation of the current component. Here in the stage-generate pattern, the same concept is applied. The stage-generate pattern takes the number of stages as a parameter. Another parameter is the operation to be replicated. The operations are repeated in sequence to create multiple stages and there is an iterator that can be used to vary operations in different stages. This pattern allows taking all kinds of operations either created by data parallel pattern or by compositional patterns. A simple repetition of one operation for three stages is illustrated in Figure 4.9

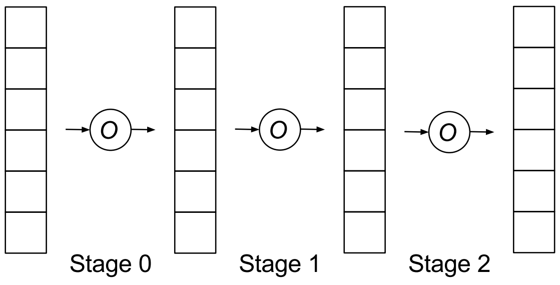


Figure 4.9: Stage Generate Pattern

4.3 Example Algorithm Modeled with Patterns

Here in this section, several algorithms are described using models that are introduced above.

4.3.1 Vector Dot Product

Vector dot product is an operation that sums the products of each dimension of two vectors. This operation takes the two vectors as inputs and generates a single number. It can be seen as a map operation followed by a reduce operation. Its structure is visualized as Figure 4.10.

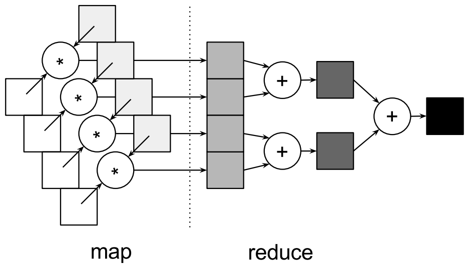


Figure 4.10: Dot Production

4.3.2 Fast Fourier Transformation (FFT)

FFT is a widely used algorithm in scientific and engineering applications. It possesses a large amount of parallelism and it also has a relatively complex structure. Thus, it is good as an example for creating algorithms with patterns. This example is inspired from the ForSyDe Atom [31] FFT example. A radix-2 decimation-in-frequency(DIF) FFT of length 4 is demonstrated here. In this thesis, only the necessary information of the algorithm is described. Additional information about the DIF FFT algorithm can be found in [32]. The purpose of this section is only to provide an introduction on how the DIF FFT algorithm can be created by patterns. A complete model that can be used by P2CL for a larger-length FFT algorithm is expressed in an extensible markup language (XML) script in Chapter 6.

As shown in Figure 4.11, the FFT algorithm consists of multiple the same computations. That is the basic building block of the DIF FFT algorithm–the butterfly operation. Figure 4.12 shows such an operation. The ω_N^l term is called a twiddle factor, which is a value that can be easily determined by the FFT length and where the butterfly function is located. The butterfly operation takes two complex values from the first and the second half of the input list and produces two intermediate complex values for the next stage.

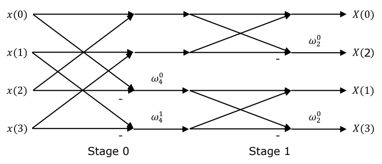


Figure 4.11: DIF FFT of length 4

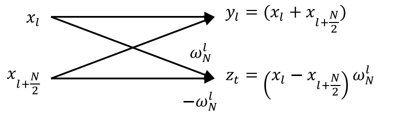

x_l
$x_{l+\frac{N}{2}}$
$y_l = (x_l + x_{l+\frac{N}{2}})$
$z_t = (x_l - x_{l+\frac{N}{2}})\,\omega_N^l$
ω_N^l
$-\omega_N^l$

Figure 4.12: DIF FFT Butterfly adapted from [32]

It is easy to see that a butterfly operation involves input/output arrangements as well as computations. If the computation of the butterfly operation is extracted as a function whose input and output are tuples of two complex numbers, then the entire butterfly operation can be expressed as a sequence of gather, map and scatter patterns. The anatomy of the first stage of the FFT algorithm is shown in Figure 4.13. The input list is first gathered into two tuples, followed by a map operation works on the two tuples. At the final step, the tuples are scattered to the output list. It is also obvious that all these gather, map and scatter operations requires half the length of the input list as a parameter. The gather and scatter operation need this length for their pattern offsets. The map operation needs the length because the butterfly function uses it to calculate the twiddle factor. It is also worth mentioning that besides the two lists of tuples, the map operation has another input list whose values are 0 and 1. It is used to indicate the position of the butterfly function.

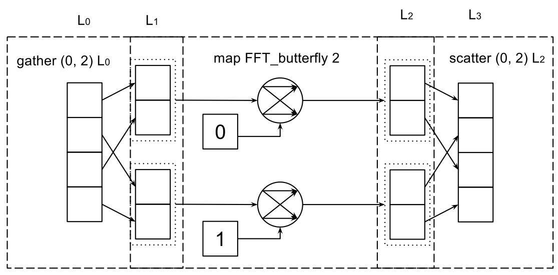


Figure 4.13: Stage 0 of the FFT Algorithm

The second stage also consists of two butterfly operations, but they are organized in a different way. They can be seen as two exactly the same operations performed on both the lower and the upper half of the list. The parameter used by gather, map, and scatter patterns for this stage is 1 and a list with one value zero is served as an additional input for the map operation. This structures can be created by the operation map pattern with the number of repetition as 2.

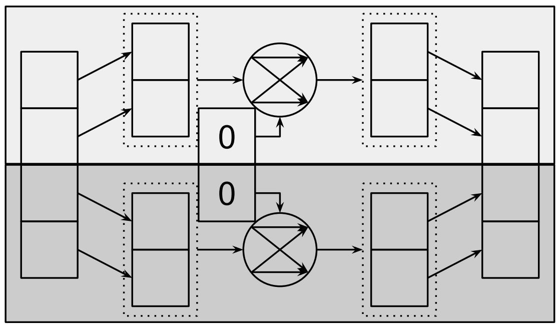


Figure 4.14: Stage 1 of the FFT Algorithm

With the analysis of two stages, it is easy to notice that the two stages have similar structures. Only some of the parameters used either by patterns or functions vary. And those parameters can all be determined by the index of the stages. Thus the entire FFT operation can be created with the following structures.

1. A generic butterfly operation is created using a sequence of gather, map, and scatter operations.

2. The butterfly operation is then extended to a larger list to form a generic stage operation of FFT algorithm.
3. The stage operation is wrapped with the stage generate pattern where the iterator of the stage pattern can be used to determine the parameters used in different stages.

A complete structure of the algorithm is shown in Figure 4.15.

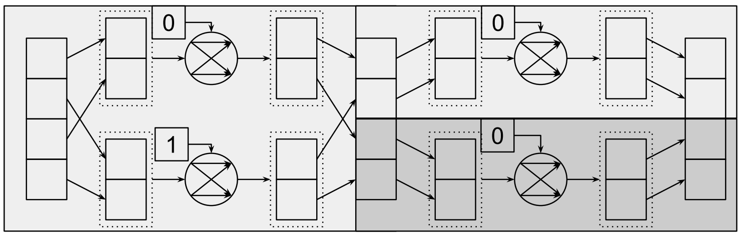


Figure 4.15: Complete FFT Algorithm Modeled by Patterns

As shown in Figure 4.11, the output of the DIF FFT is not properly ordered. Some permutation operations are necessary to be performed after the algorithm. For the radix-2 FFT, the permutation is a bit-reversal permutation, as in the indexes of the element in the input list are translated to binary format and bit-reversed to get its new indexes in the output list. This operation can be modeled by multiple stages of transpose operations. For the FFT algorithm of length 4, only one transpose operation with width 2, and height 2 is necessary. The 8-length bit-reversal permutation modeled by two stages of transpose pattern is illustrated in Figure 4.16. It can be modeled as a generic transpose operation with width 2 wrapped with the operation map pattern and then wrapped with the stage-generate pattern.

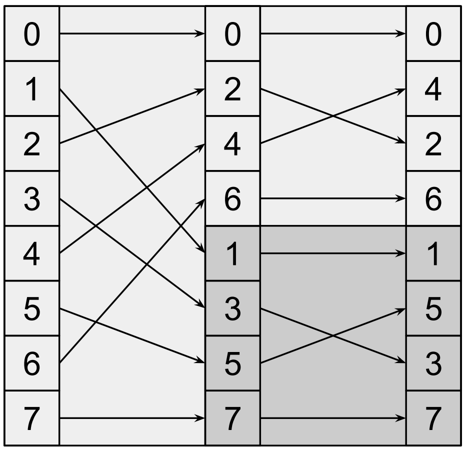


Figure 4.16: Bit-reversal Permutation

Chapter 5

Related Approaches

Exploration of using data parallel patterns or other formalisms in parallel programming has been a research topic since the creation of parallel hardware. Bird-Meertens formalism created a notation and calculus for deriving programs from specifications [33]. Its theory on lists [34] is later used as parallel model [30] and has been implemented by many projects [35] [36] for fast and efficient data parallel programming. There are also several projects use Bird-Meertens as data parallel patterns and also support for several task parallel patterns. Eden [37] and P3L [38] are such projects.

Instead of letting designers to explicitly specify patterns, there also exist researches that extract parallel patterns from other higher-order functions or from sequential programs. Examples of such researches are ParaPhrasing [39] and Busvine's PUFF compiler [40].

In this chapter, several projects that targeting developing GPU program from high-level abstraction are introduced.

5.1 F2CC

ForSyDe-to-CUDA-C (f2cc) is a software synthesis tool developped under the ForSyDe project [41]. It was developed by Gabriel Hjort Blindell in 2012 as part of his Master Thesis [3]. An experimental version with several improvements is developed under George Ungureanu's Master Thesis project in 2013 [18]. The tool generates CUDA C code from models specified with ForSyDe synchronous MoC. Since the ForSyDe synchronous MoC does not naturally exploits data parallelism. A pattern named split-map-merge is used to model a process that accepts an array as input, applies one or several functions on every element, and produces an array as output [42]. This pattern can be explicitly declared using the ParallelMapSY process constructor. It can also recognize this parallel pattern in a ForSyDe process network and combine relevant processes into a ParallelMapSY process.

In the experimental version, a general cost-based platform model used for representing execution platforms are provided. The model is provided in order to have better load balance when mapping processes to a GPU.

5.2 SkelCL and SkePU

SkelCL [43] and SkePU [44] are two independent but quite similar projects that target skeleton programming on multi-core CPUs, GPUs. They support vector and matrix as container types. The containers are where the parallel patterns can be applied on. They support map, reduce patterns which are quite similar to the definitions in Chapter 4. The map pattern in SkePU is able to take functions with any numbers of input and output, while in SkelCL, zip pattern is used for functions that take multiple inputs. They both support the scan pattern. However, scan pattern in SkePU is only capable of performing prefix sum while it is general purpose in SkelCL. Besides, SkelCL have support for stencil patterns [45] while SkePU do not. C++ is selected to be the language for both of their APIs. However, functions needed by parallel patterns are specified as function template in SkePU while they should be provided as strings in SkelCL.

In terms of implementation, SkelCL is based on OpenCL, while SkePU is capable generate codes using OpenCL, CUDA, OpenMP for different platforms. SkePU also supports autotuning so that it can have some extent of performance portability on different platforms [46].

Part II

Development and Implementations

Chapter 6

Representations of Parallel Patterns

This chapter provides a tutorial on how to specify an algorithm using parallel patterns in an XML file that are accepted by P2CL.

6.1 Supported Data Types

P2CL support several atomic types and also support composite data types with some limitations.

Atomic Types The supported atomic types are listed below.

- char
- unsigned char/ uchar
- short
- unsigned short/ ushort
- int
- unsigned int/ uint
- long
- unsigned long/ ulong
- float

Note that the double type is not supported in current version of P2CL. Support for double type is an optional feature in OpenCL. Current version discards this type because it does not detect whether the targeting hardware support double type.

Tuple Types In terms of composite data types, only tuple types are supported P2CL. They are structures that consist of elements of the same atomic type. Their names are the capitalized atomic datatype name CHAR, UCHAR, SHORT, USHORT, INT, UINT, LONG, ULONG and FLOAT followed by an integer value *n* that defines the number of elements. The supported values for *n* are within the range 2 and 32 inclusive. However, it is better for users to consider using values 2, 4, 8, 16, so that the intrinsic vector types function provided by OpenCL can be used. The elements of the tuple types can be referred to by indices that started with character 'e' and followed by the index. f.e0 refer to the first elements of a tuple named f. In order to represent more complex composite data types, the idea of SoA which is introduced in Chapter 4 should be used as a workaround.

6.2 Function Decriptions

Several parallel patterns require functions as parameters. In order for parallel patterns to use these functions, some general information of the functions needs to be specified in the input XML file. Note that the XML file only needs the declaration of functions. The definition of them is provided to P2CL separately from the XML file. Listing 6.1 shows an example of the function descriptions section. All the function descriptions are provided in the body of the function_descriptions element. Each function description is specified as a function element. Inside a function element, the function declaration and the information of the parameters are provided. The declaration should follow the syntax of C language. Some additional restrictions should also be followed.

- The function should not have return types. All the outputs should be passed through pointers provided as parameters.
- The parameter list is allowed to contain multiple inputs and output data and several integer parameters that are used to vary the computation.
- The parameters should be provided in the order of input data, output data pointers and functional parameters.
- A special input is the index. It must be the last one of all the input parameters. The name of the parameter must be "index" and its type must be int. It is used to take the corresponding value from an index list so that the function is able to know the location of the elements it operates on. It is a special input because the index list is a virtual list that should not provide by users or generated by the preceding operation. As long as the parameter is specified correctly, the synthesized kernel for map and reduce patterns will feed the index to the function.

After the function declaration, the number of the input, output, and functional parameters should also be provided in separate elements.

```
<function_descriptions>
  <function>
    <decl>void fft_bfly2(FLOAT4 fin, int index, FLOAT4* fout, int m)</decl>
    <num_input>2</num_input>
    <num_output>1</num_output>
    <num_para>1</num_para>
  </function>
</function_descriptions>
```

Listing 6.1: Function Descriptions

6.3 Process Descriptions

The process element contains the descriptions of operations modeled by patterns along with information of the input and output port. The process is specified as a sequence of parallel operations where the output lists of the previous operation are the input of the next operation. Thus, the sequence of the operations elements matters in the XML file. Basic operations modeled by data parallel patterns are specified in separate operation elements. They are wrapped by elements of compositional patterns to create more complex operations.

6.3.1 Variables and Parameters

Before getting into details of the process descriptions, it is worth mentioning that users can declare integer variables that can be used as parameters either for patterns or for the functions used by patterns. The variables can be declared inside the scope of a process or in the scope of a stage generate or an operation map pattern. The stage generates pattern also provides an iterator variable for operations inside it. The variables can either be specified as a fixed integer number or as an expression that uses other variables as parameters. Listing 6.2 shows the two ways of defining a variable. Variables that are set by value should have the value_type attribute set as "value". The default value for this attribute is also "value" so that this can also be omitted. Variables that are set by expression should set that attribute as "expr". Those variables must also specify the other variables that the expression is depended on in a space separated list in the para_list attribute. Basic $+ - */$ operators and math functions pow() and sqrt() can be used to create the expression.

```
  <shared_variable name="fft_2pow" value_type="value">9</shared_variable>
  <shared_variable name="fft_length" value_type="expr" para_list="fft_2pow">pow(2,
      fft_2pow)</shared_variable>
```

Listing 6.2: Variables Declarations

Figure 6.1 show a structure of variables. Variable a and b are declared in the global scope of the process. Variable c is in the scope of a stage generate pattern. An iterator is also provided by the stage generate pattern. The patterns inside the stage generate pattern can access those variables. There also exists a tree-structured dependency relationship between those variables.

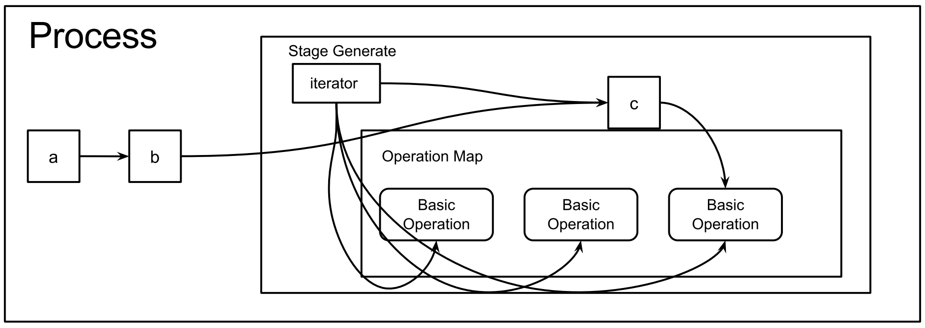


Figure 6.1: Variables Tree

6.3.2 Port Declarations

The information of ports determines the input and output list size. Listing 6.3 shows the declarations of two input ports and one output port. Each port element should specify the name, the direction, the type, and the length attributes. The name of the port is used to identify it. It is later used by P2CL APIs to query the index of the port. The direction attribute determines whether the port is an input port or an output port. The type and length of the port is used to determine the input and output list size. For simplicity, only atomic data types can be used in this type attribute.

```
  <port name="iport0" direction="in" type="int" length="4"/>
  <port name="iport1" direction="in" type="int" length="4"/>
  <port name="oport" direction="out" type="int" length="1"/>
```

Listing 6.3: Port Declarations

6.3.3 Data Parallel Patterns

Operations created by data parallel pattern are represented inside operation elements. Depend on the skeleton types (parallel pattern types), the operation elements require different child elements.

map A map operation is shown in Listing 6.4. The function elements specify the function that is used to create the operation. The func_para element contains settings for the functional parameters of that function. The element whose name is a "para" followed by an index is to set those parameters which is identified by the index. The way of setting values to the parameters is similar to declarations of variables. They can either be specified as a fixed value or an expression. The length element is used to set the number of elements in the input and output lists. The element types are depended on the input and output parameters types of the function declaration.

```
<operation>
    <skeleton_type>map</skeleton_type>
    <function>fft_bfly2</function>
    <func_para>
```

```
        <para0 value_type="expr" para_list="m">m</para0>
    </func_para>
    <length value_type="expr" para_list="m">m</length>
</operation>
```

Listing 6.4: Map Operation Representation

reduce Listing 6.5 shows an reduce operation. It is almost the same as the map representation. The only difference is the function used by the reduce pattern can only have two input parameters and one output parameter. And no index input is allowed. These restrictions guarantees the function is a binary associative function. However, as mentioned in Chapter 4, these restrictions made the reduce pattern currently does not support SoA input and output.

```
<operation>
    <skeleton_type>reduce</skeleton_type>
    <function>vector_sum</function>
    <length value_type="value">4</length>
</operation>
```

Listing 6.5: Reduce Operation Representation

gather A gather operation is described in a script snippet in Listing 6.6. The input_range is used to define the number elements of the input list. The basetype and tuple_size elements are used to determine the types of input elements. In this example, elements of the input list is seen as a tuple of two float numbers. If another basetype is set, there will be another input list whose elements is a tuple of two number of that basetype, and there will also have two output lists. It is also important to notice that only a fixed integer number can be assigned in the tuple_size element. This is different from other parameters which can accept expressions. Offset elements are used to set the offsets of the gather operation. Since the type of input elements is FLOAT2 and there are two offsets specified, the type of collective output elements is FLOAT4, as in two FLOAT2 tuples grouped together. The length elements define the number of tuples inside the output lists.

```
<operation>
    <skeleton_type>gather</skeleton_type>
    <tuple_size>2</tuple_size>
    <input_range value_type="expr" para_list="m">2 * m</input_range>
    <basetype>
      <type0>float</type0>
    </basetype>
    <offset>0</offset>
    <offset value_type="expr" para_list="m">m</offset>
    <length value_type="expr" para_list="m">m</length>
</operation>
```

Listing 6.6: Gather Operation Representation

scatter An example of operation representation is shown in Listing 6.7. The syntax is similar to the gather operation. However, the tupe_size and basetype are used to determine the type for output lists instead of input lists. The output_range also describes the number of elements in the output lists. In this way, the gather and scatter operations with the same parameters are the exact inverse operations.

```
<operation>
    <skeleton_type>scatter</skeleton_type>
    <length value_type="expr" para_list="m">m</length>
    <tuple_size>2</tuple_size>
    <basetype>
      <type0>float</type0>
    </basetype>
    <output_range value_type="expr" para_list="m">2 * m</output_range>
    <offset>0</offset>
    <offset value_type="expr" para_list="m">m</offset>
</operation>
```

Listing 6.7: Scatter Operation Representation

transpose Listing 6.8 shows an example of a transpose pattern. The basetype and tuple_size have the same meaning and syntax as the gather pattern. The width and height parameters can be set as fixed values or expressions.

```
<operation>
    <skeleton_type>transpose</skeleton_type>
    <width>2</width>
    <height>256</height>
    <tuple_size>2</tuple_size>
    <basetype>
        <type0>int</type0>
    </basetype>
</operation>
```

Listing 6.8: Transpose Operation Representation

6.3.4 Compositional Patterns

The compositional patterns are defined as container elements that can embed basic operations

Operation Map Pattern The operation map pattern is simply a map element with sequences of operations and a num_of_extensions inside it. An example is shown in Listing 6.9.

```
<map>
  <num_of_extensions value_type="expr" para_list="i">pow(2, i)</num_of_extensions>
  <!-- -...---->
  <!---basic operations--->
</map>
```

Listing 6.9: Operation Map Representation

Stage Generate Pattern The stage generate pattern element needs to specify the number of stages. The iterator_name element is used to set the name of the iterator variable. Other variables can be defined under the scope of a stage generate pattern. Listing 6.10 shows an example of stage generate patterns.

```
<stage_generate>
  <num_of_stages value_type="expr" para_list="fft_2pow">fft_2pow</num_of_stages>
  <iterator_name>i</iterator_name>
  <!-- -...---->
  <!---other compositional patterns or sequences of basic operations--->
</stage_generate>
```

Listing 6.10: Stage Generate Pattern Representation

6.4 Examples

This section shows the XML documents that represent the two examples introduced in Chapter 4. Listing 6.11 shows the description for the vector dot product.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<p2cl>
<function_descriptions>
  <function>
    <decl>void vector_mul(int a, int b, int* c)</decl>
    <num_input>2</num_input>
    <num_output>1</num_output>
    <num_para>0</num_para>
  </function>
  <function>
    <decl>void vector_sum(int a, int b, int* c)</decl>
    <num_input>2</num_input>
    <num_output>1</num_output>
    <num_para>0</num_para>
  </function>
</function_descriptions>
<process>
  <port name="iport0" direction="in" type="int" length="4"/>
  <port name="iport1" direction="in" type="int" length="4"/>
  <port name="oport" direction="out" type="int" length="1"/>

  <operation>
      <skeleton_type>map</skeleton_type>
      <function>vector_mul</function>
      <length value_type="value">4</length>
  </operation>
  <operation>
      <skeleton_type>reduce</skeleton_type>
      <function>vector_sum</function>
      <length value_type="value">4</length>
  </operation>
</process>
</p2cl>
```

Listing 6.11: Vector Dot Production Representation

Listing 6.11 shows the description for the fast fourier transform (FFT) algorithm.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<p2cl>
<function_descriptions>
  <function>
    <decl>void fft_bfly2(FLOAT4 fin, int index, FLOAT4* fout, int m)</decl>
    <num_input>2</num_input>
    <num_output>1</num_output>
    <num_para>1</num_para>
  </function>
</function_descriptions>
<process>
  <port name="iport" direction="in" type="float" length="1024"/>
  <port name="oport" direction="out" type="float" length="1024"/>
  <shared_variable name="fft_2pow" value_type="value">9</shared_variable>
  <shared_variable name="fft_length" value_type="expr" para_list="fft_2pow">pow(2,
     fft_2pow)</shared_variable>
  <stage_generate>
    <num_of_stages value_type="expr" para_list="fft_2pow">fft_2pow</num_of_stages>
    <iterator_name>i</iterator_name>
    <shared_variable name="m" value_type="expr" para_list="fft_length i">
      fft_length/pow(2, i+1)
    </shared_variable>
    <map>
      <num_of_extensions value_type="expr" para_list="i">pow(2, i)</
     num_of_extensions>
      <operation>
          <skeleton_type>gather</skeleton_type>
          <tuple_size>2</tuple_size>
          <input_range value_type="expr" para_list="m">2 * m</input_range>
          <basetype>
            <type0>float</type0>
          </basetype>
          <offset>0</offset>
          <offset value_type="expr" para_list="m">m</offset>
          <length value_type="expr" para_list="m">m</length>
      </operation>
      <operation>
          <skeleton_type>map</skeleton_type>
          <function>fft_bfly2</function>
          <func_para>
              <para0 value_type="expr" para_list="m">m</para0>
          </func_para>
          <length value_type="expr" para_list="m">m</length>
      </operation>
      <operation>
          <skeleton_type>scatter</skeleton_type>
          <length value_type="expr" para_list="m">m</length>
          <tuple_size>2</tuple_size>
          <basetype>
            <type0>float</type0>
          </basetype>
          <output_range value_type="expr" para_list="m">2 * m</output_range>
          <offset>0</offset>
          <offset value_type="expr" para_list="m">m</offset>
      </operation>
    </map>
  </stage_generate>
  <stage_generate>
    <num_of_stages value_type="expr" para_list="fft_2pow">fft_2pow</num_of_stages>
    <iterator_name>i</iterator_name>
    <shared_variable name="m" value_type="expr" para_list="fft_length i">
      fft_length/pow(2, i+1)
    </shared_variable>
    <map>
      <num_of_extensions value_type="expr" para_list="i">pow(2, i)</
     num_of_extensions>
      <operation>
          <skeleton_type>transpose</skeleton_type>
          <width>2</width>
          <height value_type="expr" para_list="m">m</height>
          <tuple_size>2</tuple_size>
          <basetype>
              <type0>float</type0>
          </basetype>
      </operation>
    </map>
  </stage_generate>
</process>
</p2cl>
```

Listing 6.12: Vector Dot Production Representation

Chapter 7

P2CL Overview

This chapter gives an overview of the P2CL library. It is split into the user guide on how to use the library and the introduction to the implementation details. The APIs of P2CL is quite simple. The content of this chapter is enough to understand them. The structure of the library is described in this chapter, whereas, the details of kernel generation and execution is described in Chapter 8.

7.1 Overview from Users' Perspective

This section provides information that is sufficient for users to use the P2CL library with given algorithms modeled in an XML file.

7.1.1 Designing Workflow

The APIs of P2CL is developed with the intention to separate the algorithm specification and the actual usage of the algorithm. With such motivation, the modeled algorithm can be easily invoked by the provided APIs. Figure 7.1 shows the overview from the users' perspective. The workflow of using an algorithm with P2CL is described as follows:

1. The algorithm is first modeled using patterns with the structure of algorithm stored in an XML file.

2. Although the functions used by the patterns are declared inside the XML file, their definitions are provided in strings as another parameter at initialization.

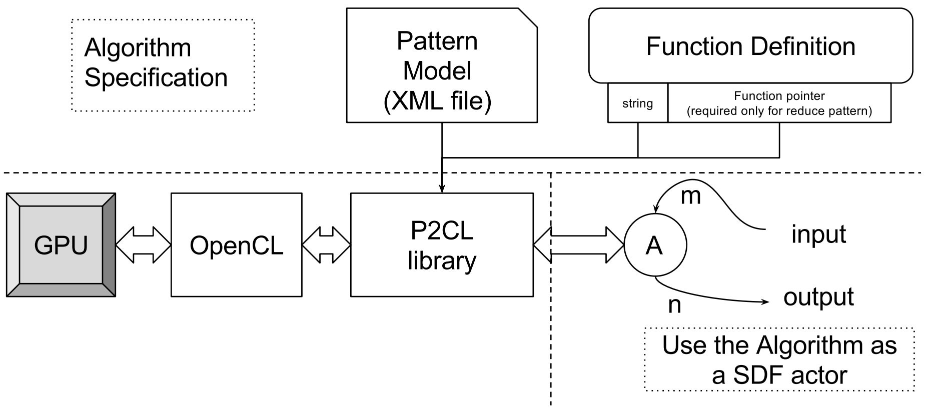


Figure 7.1: Workflow

3. For the algorithms that contain reduce patterns, function pointers of reduce functions are also required by the library so that the CPU can do the last steps of combinations. This is because the last steps of reduce operations have limited level of parallelism.

4. After the initialization, a process object is created. The programmers can easily use the object as a SDF actor. The consumption and production rate of each input and output port is determined by the input and output lists size specified in the XML file. The APIs are thread-safe, thus they can be invoked from different threads.

An example of a vector dot production program created using P2CL is demonstrated in Listing 7.1. Line 40 shows the initialization of a process object. According to the description of the constructor function shown in the text box below, the process object is created with the capability of enqueuing enough data that is suitable for three invocations of the algorithm. The path of the XML file is provided as the second parameter, and the string of the function definitions is provided as the third parameter. Since the dot production involves a reduce pattern, a function pointer that do the final combinations should be provided. The required function pointer needs to satisfy the type $void(*)(void*a, void*b, void*c)$, where the first two parameters are pointers to the inputs and the last parameter is the pointer of the output. Thus the function vector_sum is wrapped with another function and feed to the constructor.

Process Constructor

```
p2cl::Process::Process (
        int batchNum,
        std::string xmlFileName,
        std::string additionalFunc = std::string(),
        std::function<void(void *, void *, void *)> reduceFunction
        = std::function<void(void *, void*, void*)>()
  )
```

Parameters:

batchNum	the max number of computation that the process object can handle at the same time
xmlFileName	file name of the xml file that describes the algorithm

After the initialization, the pushBack function is used to enqueue the vectors. The first parameter is the size in byte that the user want to enqueue. The second parameter is the index of the input array that the data should be enqueued to. The third parameter takes the pointer of the input data. In this demonstration example, two vectors with only four dimensions are pushed into the object. However, in a real program, larger data sets should be used for achieving higher efficiency.

After pushing back the required input data, users may use the blocking pop function to pop the result. The first parameter is the index of the output array and the second parameter is the pointer to the output location. The poped data size in bytes is provided as the return value of the function.

```cpp
#include <iostream>
#include "process.hpp"

void vector_sum(int a, int b, int* c)
{
  *c = a + b;
}

void vector_sum_wrapper(void *a, void *b, void*c)
{
  int va = *((int *)a);
  int vb = *((int *)b);
  vector_sum(va, vb, (int *) c);
}

const char * functions=
"\
void vector_mul(int a, int b, int* c)\
{\
  *c = a * b;\
}\
\
void vector_sum(int a, int b, int* c)\
{\
  *c = a + b;\
}\
";

int main(int argc, char *argv[])
{
  int a[4] = {1,2,3,4};
  int b[4] = {1,2,3,4};
  int c[4];

  int iport0_index;
  int iport1_index;
  int oport_index;

  if(argc < 2)
  {
    std::cout<<"usage: "<<argv[0]<<" xmlfilename"<<std::endl;
    return 0;
  }

  p2cl::Process obj(3,argv[1], functions, vector_sum_wrapper);
  iport0_index = obj.getInPortIndex("iport0");
  iport1_index = obj.getInPortIndex("iport1");
  oport_index = obj.getOutPortIndex("oport");


  obj.pushBack(sizeof(int), iport0_index, a);
  obj.pushBack(sizeof(int), iport0_index, a + 1);
  obj.pushBack(sizeof(int), iport0_index, a + 2);
  obj.pushBack(sizeof(int), iport0_index, a + 3);
  obj.pushBack(sizeof(int), iport1_index, b);
  obj.pushBack(sizeof(int), iport1_index, b + 1);
  obj.pushBack(sizeof(int), iport1_index, b + 2);
  obj.pushBack(sizeof(int), iport1_index, b + 3);
  obj.pop(oport_index, c);
  std::cout<<c[0]<<std::endl;
}
```

Listing 7.1: Vector Addition Kernel

7.1.2 Buffer Sizes and Flow Control

The APIs of P2CL allows users to use algorithms as SDF actors. Although single invocation of the algorithm consumes fixed-length input data and generates fixed-length output data, it is recommended to enqueue more input data to the buffer before waiting for the result, so that more instances of computations can be executed concurrently to achieve higher performance. Besides that, each input or output port can be operated in different threads. Thus, it is important for users to understand the size of the input and output buffer and how flow control is achieved for multithreaded programs.

With fixed consumption and production rate, typically the size of FIFO buffers connected to a SDF actor is dependent on the consumption and production rate and the scheduling of the actor. Since the current version of P2CL does not support networks of processes, the maximum buffer size is determined by the first parameter of the process constructor. Take the vector dot product described in the last subsection as an example. The list size for both input ports is 4 float numbers. The output port size is one float number. The first parameter to the constructor is 3, which means the process object is capable of storing 12 float numbers in each input port buffer before popping any output data. The output port is also able to store 3 float numbers. If more input data tries to be pushed to the FIFO buffer, the function will block the thread. Fetching data from an empty output port also cause the thread to be blocked. Thus, a simple flow control mechanism is achieved.

7.2 Overview of the Library

Figure 7.2 illustrates the structure of the P2CL library. It can be separated into the synthesis part and the execution part. All the rounded rectangles represent the vital objects used insides the library. The parser and analyzer are surrounded with dashed lines because these two objects are deleted after the synthesis procedure to reduce memory footprint.

Figure 7.3 shows the detailed procedure of the synthesis part. The parser takes

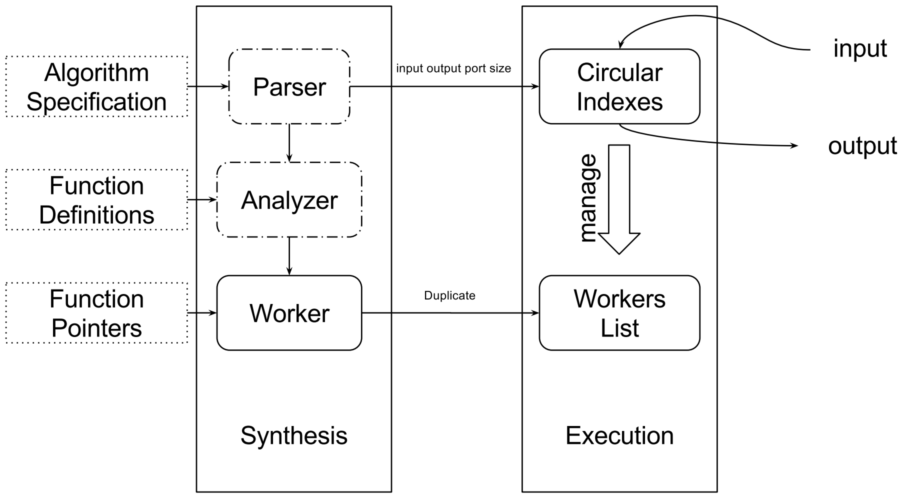


Figure 7.2: Block Diagram of P2CL library

the algorithm specification as an input and generate an intermediate operations representation which is stored in a special linked list structure. This representation is used to create the analyzer object. The parser also passes the port information to the analyzer. The analyzer is used to scan the intermediate representation, update the representation, and collect information needed for execution. After analysis, the analyzer passes the kernel program, max lists size for all operations, and the execution plan to a worker object.

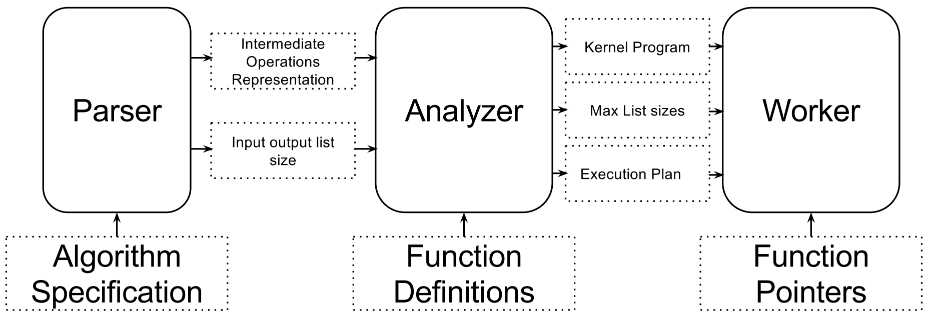


Figure 7.3: Details of the Synthesis Procedure

In the execution part, the worker object is provided with the OpenCL context. The worker object is then duplicated to a list of objects. Each worker object manages a set of OpenCL zero-copy buffers connected to the GPU device. They can independently dispatch computations to support concurrent execution.

Those workers are managed by a set of circular indexes. Suppose a program enables four workers, which are identified as worker A, worker B, worker C and worker D. Figure 7.4 shows how they are managed by a set of indexes. The worker objects are listed in a circle where indexes iterate around it. Each input port has an index points to a worker whose input buffer for that port has not been filled. The indexes for the input port also record how much data has been enqueued to the corresponding input buffer of the current worker. They advance to the next worker when the corresponding input buffers are filled. When all the input buffer of a worker has been filled, the worker is invoked to start the computation. The execution index points to the next worker that is waiting to be invoked. The output port indexes indicate the workers whose corresponding output list has not been read out. Each output list is popped out as a whole. The result index points to the last worker who has output data that has not been read out. All the input port index cannot exceed the result index since the computation of workers after that index might not complete. The input of these workers should not be changed. In a similar manner, the output port indexes cannot exceed the execution index since the input of workers beyond that index is not ready.

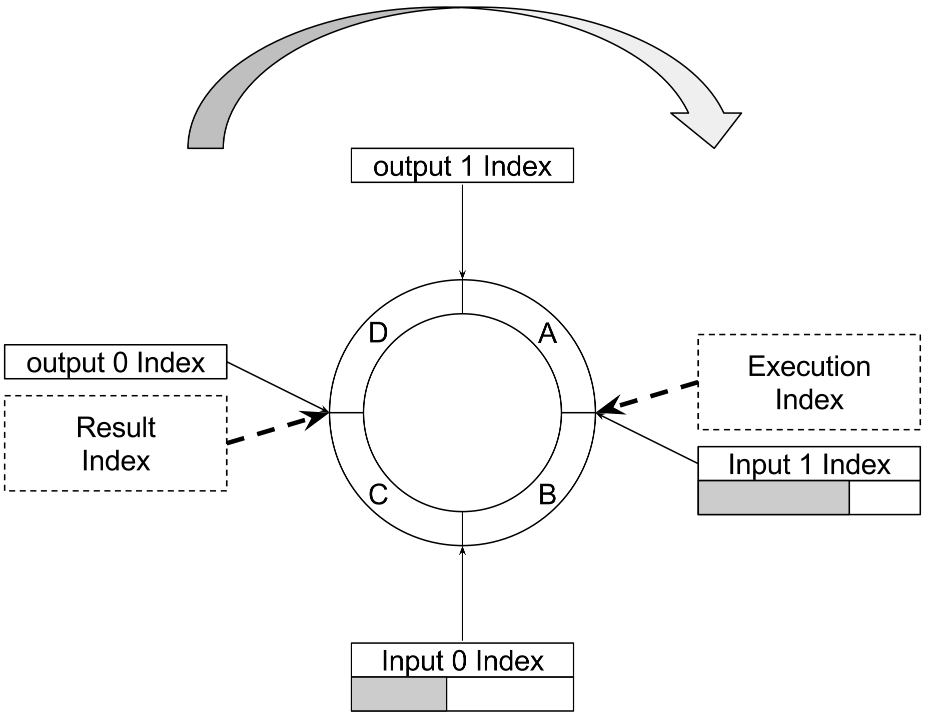


Figure 7.4: Details of the Circular Index

Chapter 8

Kernel Generation and Execution

This chapter describes how the P2CL library transforms patterns to the OpenCL kernels and how the kernels are executed. The P2CL library provides kernel templates for Map, Reduce, Transpose patterns. The gather and scatter pattern share the same template named data arrangement template.

8.1 Pattern Fusion

In the work of Sato, Shigeyuki et al. fusion transformation is applied to skeleton programming for GPGPU [36]. The idea is that operations created by data parallel patterns can be fused to a single operation according to some rules, so that the operation can fit in one kernel invocation in GPU programming. This is a beneficial optimization since the intermediate results between two successive operations do not need to fall back to the slow global memory at the end of the first kernel invocation and loaded back to private memory in the later kernel invocations. They apply a greedy fusion strategy, which fuses operations even when it involves recomputations. This is still in most cases beneficial since the computation cost in GPU programming is usually much lower than the cost of memory accesses.

In the current version of P2CL, since operations in a process can only be consecutive operations, the fusions are much simpler. The fusion rules for the patterns used in the thesis is shown is Table 8.1.

Next Operation Type / Previous Operation Type	Map	Reduce	Gather	Scatter	Transpose
Map	M	R		MF	
Reduce					
Gather	M	R		MF	
Scatter					
Transpose					

- M means the fused operation use the Map kernel template, and can still follow the fusion rule of map operation to fuse with later operations.
- MF means the fused operation use the Map kernel template, but cannot continue to fuse with other operations.
- R means the fused operation use the Reduce kernel template.

Table 8.1: Fusion Rules

8.2 Kernel Templates

Operations described with patterns can be synthesized into kernel sources with some templates. By filling in code snippets generated by operation information, a complete kernel source can be created.

8.2.1 Map Kernel

Listing 8.1 shows an example of generated map kernel sources. It is structured with several sections wrapped with curly brackets. The first section is used to shift the input and output pointers. This is an optional section that only appears for operations inside a operation map pattern. After declarations of input and output variables, the second section is used for loading input data from the global memory to local memory. The data can either be loaded according to the global index or gathered according to the information of a fused gather operation. The section of computation follows the loading section, in which a sequence of function can be nested to get the final output data. The last section of the map kernel is the storing section, the output data is either normally stored in sequence or scattered to multiple locations. The definitions of tuple types and the storing and loading function TUPLE_POINTER_LOAD and TUPLE_POINTER_STORE are automatically generated and are put in the beginning of the entire kernel source.

```
__kernel void operation0(
    __global float *  input0
    ,
    __global float *  output0
    ,
    int m
)
{
  size_t g0 = get_global_id(0);

  size_t l0 = get_local_id(0);

  {
    //shift input and output pointer for
    //high order map pattern
    size_t g1 = get_global_id(1);

    size_t g_size0 = get_global_size(0);

    input0 += g1  * (2 * m) * 2;
    output0 += g1 * (2 * m) * 2;
  }

  FLOAT4 input_variable0;
  FLOAT4 output_variable0;
  {
    //the section for loading input data
    float * p_input_var0 = (float *)(&input_variable0);
    {
      int arrange_index_base = g0;
      int range = 2 * m;
      int arrange_index;
      arrange_index = arrange_index_base + (0);
      if((arrange_index < range && arrange_index >= 0))
      {
        TUPLE_POINTER_LOAD2(p_input_var0, arrange_index, input0);

      }else{
        p_input_var0[0] = 0;
        p_input_var0[1] = 0;


      }

      arrange_index = arrange_index_base + (m);
      p_input_var0 += 2;
      if((arrange_index < range && arrange_index >= 0))
      {
        TUPLE_POINTER_LOAD2(p_input_var0, arrange_index, input0);

      }else{
        p_input_var0[0] = 0;
        p_input_var0[1] = 0;


      }

    }

  }

  {
    //the section for the mapped functions
    fft_bfly2(input_variable0, g0, &output_variable0, m);
  }

  {
    //the section for storing ouput data
    float * p_output_var0 = (float *)(&output_variable0);
    {
      int arrange_index_base = g0;
      int arrange_index;
      arrange_index = arrange_index_base + (0);

      TUPLE_POINTER_STORE2(p_output_var0, arrange_index, output0);

      arrange_index = arrange_index_base + (m);

      p_output_var0 += 2;
      TUPLE_POINTER_STORE2(p_output_var0, arrange_index, output0);

    }

  }

}
```

Listing 8.1: Map Kernel Source

8.2.2 Reduce Kernel

Before introducing the Reduce kernel template, it is necessary to describe how a reduce operation is performed in P2CL. Figure 8.1 illustrates the execution data flow. At the beginning of execution, the input data is stored in the global memory. For the example shown in the figure, there are two work-groups where each of them contains four work-items. Firstly, all the work-items load the data elements according to their global indexes. If the number of elements is more than the number of work-items, each work-item will continuously load the next corresponding element and combined it with the previously computed value until all the input data has been loaded. This approach guarantees coalesced global memory accesses. The next step is to combine values within work-groups. Each element stores their intermediate value to the local memory which is shared within its work-group. The first half of elements in the local memory is combined with the second half utilizing only half of the work-items. Iteratively, all the values in the local memory reduce to one value for each work-group. These results are stored back to the global memory. Finally, CPU will do the last step of computation since the level of parallelism at the end is limited. It is also worth mentioning that the work-group size is set to be the maximum work-group size for the reduce kernel if the input list length is higher than that. When there is a small input list, the work-group size is selected as the maximum power-of-two number that is smaller than the input length. A limit of 32 also restricts the total number of work-groups so that only limited data needs to be transferred back to the CPU.

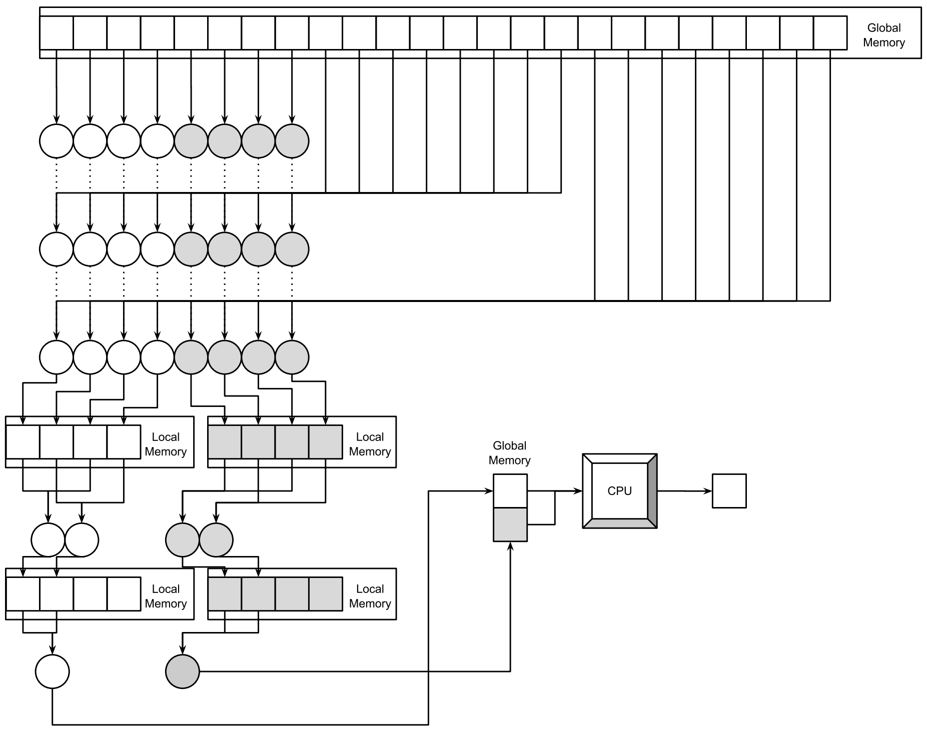


Figure 8.1: Reduce Pattern Data Flow

Listing 8.2 shows the kernel template of reduce pattern. It reflects the execution plan described above with the first for loop loading and combining data and the next while loop combining results within work-groups. Note that the input data in the reduce template can either directly loaded from the global memory or from the intermediate result of a fused map pattern.

```
__kernel void operation100(
                __global float *        input0
                ,
                __global float *        output0
                ,
                int reduce_length
                ,
                __local float*  operation100local_space0
)
{
        size_t gp0 = get_group_id(0);


        size_t l0 = get_local_id(0);

        size_t l_size0 = get_local_size(0);

        size_t g_size0 = get_global_size(0);

        size_t index0 = get_global_id(0);

        FLOAT4 input_variable0;
        FLOAT4 partial_result;
        float* p_partial_result = (float*)&partial_result;
        float* p_input_var0= (float*)&input_variable0;
        int partial_length = l_size0;
        TUPLE_POINTER_LOAD4(partial_result, index0, input0);
        int it_end = reduce_length;
        for(int it = index0 + g_size0; it < it_end; it += g_size0)
        {
                TUPLE_POINTER_LOAD4(input_variable0, it, input0);
                {
                        reduce(partial_result, input_variable0, &partial_result)
     ;
                }


        }

        TUPLE_POINTER_STORE4((p_partial_result), l0, operation100local_space0);
        barrier(CLK_LOCAL_MEM_FENCE);
        while(partial_length > 1)
        {
                partial_length >>= 1;
                if(l0 < partial_length)
                {
                        TUPLE_POINTER_LOAD4(p_input_var0, l0 + partial_length,
     operation100local_space0);
                        {
                                reduce(partial_result, input_variable0, &
     partial_result);
                        }

                        TUPLE_POINTER_STORE4(p_partial_result, l0,
     operation100local_space0);

                }
                barrier(CLK_LOCAL_MEM_FENCE);

        }
        if(l0 == 0)
        {
                TUPLE_POINTER_STORE4(p_partial_result, gp0, output0);

        }
}
```

Listing 8.2: Reduce Kernel Source

8.2.3 Data Arrangement Kernel

The data arrangement kernel is used for the gather and scatter pattern when they are not fused with other patterns. Although the gather and scatter pattern can be achieved using map pattern, gathering and scattering data directly for large tuple types in the way of the map template may break down to several global memory accesses and introduces overhead. This data arrangement template uses one work-item for loading and storing one element in a tuple. Listing 8.4 shows one kernel source code for gathering two tuples of four float numbers. Each workitem takes the element in the tuple corresponding to its local index. There are two offsets specified for this gather operation. Therefore only two coalesced global memory accesses are necessary for loading the data. In the second section of the kernel source, each work-item also stores the data to the two corresponding locations in the grouped tuple.

```
__kernel void operation100(
    __global float *  input0
    ,
    __global float *  output0
    ,
    int m
)
{
  size_t gp0 = get_group_id(0);

  size_t l0 = get_local_id(0);



  FLOAT2 input_variable0;

  {
    float * p_input_var0 = (float *)(&input_variable0);
    {
      int arrange_index_base = gp0;
      int arrange_index;
      int range = 2 * m;
      arrange_index = arrange_index_base + (0);
      if((arrange_index < range && arrange_index >= 0))
      {
        arrange_index = arrange_index * (4);
        p_input_var0[0] = input0[arrange_index + l0];

      }else{
        p_input_var0[0] = 0;

      }

      arrange_index = arrange_index_base + (m);
      if((arrange_index < range && arrange_index >= 0))
      {
        arrange_index = arrange_index * (4);
        p_input_var0[1] = input0[arrange_index + l0];

      }else{
        p_input_var0[1] = 0;

      }

    }

  }


  {
    float * p_output_var0 = (float *)(&input_variable0);
    {
      int arrange_index_base = gp0 * 8;
      arrange_index_base += l0;
      output0[arrange_index_base + 0] = p_output_var0[0] ;
      output0[arrange_index_base + 4] = p_output_var0[1] ;
    }

  }

}
```

Listing 8.3: Data Arrangement Kernel Source

8.2.4 Transpose Kernel

The kernel template for the transpose pattern is basically a matrix tranpose kernel except that local indexes of work-items are used for elements in tuple types. The input lists are separated into small segments. Each segment is loaded to the local memory which is shared within one work-group. This approach tries to coalesce global accesses by using different work-items for loading and storing one element for the differently aligned input and output list. Notice that in line 37 and 54, there is a "plus one" on the column size for computing the index for the local memory. That is used for avoiding bank conflicts on local memory.

```
 1 __kernel void operation0(
 2     __global int *  input0
 3     ,
 4     __global int *  output0
 5     ,
 6     __local int * operation0local_space0
 7     ,
 8     int width
 9     ,
10     int height
11     ,
12     int group_dimx
13     ,
14     int group_dimy
15 )
16 {
17   size_t gp0 = get_group_id(0);
18 
19 
20   size_t l0 = get_local_id(0);
21 
22   int tuple_index = l0 % 2;
23   l0 = l0 / 2;
24   int block_size_x = ceil((float)width/ group_dimx);
25   int block_x = gp0 % (block_size_x);
26   int block_y = gp0 / (block_size_x);
27 
28   int local_x = l0 % group_dimx;
29   int local_y = l0 / group_dimx;
30 
31   int in_x = mad24(block_x, group_dimx, local_x);
32   int in_y = mad24(block_y, group_dimy, local_y);
33   if(in_x < width && in_y < height)
34   {
35     int input_index = mad24(in_y, width, in_x);
36 
37     int local_input = mad24(local_y, group_dimx * 2 + 1, local_x * 2);
38 
39     local_input += tuple_index;
40     input_index = input_index * 2 + tuple_index;
41     operation0local_space0[local_input] = input0[input_index];
42 
43   }
44   local_x = l0 % group_dimy;
45   local_y = l0 / group_dimy;
46 
47   int out_x = mad24(block_y, group_dimy, local_x);
48   int out_y = mad24(block_x, group_dimx, local_y);
49 
50   barrier(CLK_LOCAL_MEM_FENCE);
51   if(out_x < height && out_y < width)
52   {
53     int output_index = mad24(out_y, height, out_x);
54     int local_output = mad24(local_x, group_dimx * 2 + 1, local_y * 2);
55 
56     local_output += tuple_index;
57     output_index = output_index * 2 + tuple_index;
58     output0[output_index] = operation0local_space0[local_output];
59 
60   }
61 }
```

Listing 8.4: Data Arrangement Kernel Source

Part III

Evaluations and Discussions

Chapter 9

Evaluations

Although current P2CL has some limitations, it manages to ease the GPU programming and provides implementations with sufficient efficiency. This chapter demonstrates the result of some experiments regarding the usability and efficiency of P2CL.

The evaluations are performed on two devices. Their specifications are listed in Table 9.1. Device One is a Laptop with one integrated graphics unit which is in the same System on a Chip (SoC) of the CPU. Device Two is a virtual private server that is equipped with one graphics card and one virtual CPU.

	Device One	Device Two
CPU Model	Intel® i7-6500U	Intel® Xeon® E5-2682 v4
GPU Model	Intel® HD Graphics 520	AMD FirePro™ S7150
OpenCL Version	2.0	1.2
Maximum Number of Compute Units	24	32
Local Memory Size	64 KiB	32 KiB
Global Memory Size	1488 MiB	7.979GiB
Maximum Work-group Total Size	256	256
Maximum Work-group Dimensions	{256 256 256}	{256 256 256}

Table 9.1: Device Specifications

9.1 Programing Simplicity

P2CL reduces the complexity of creating a OpenCL program. The following table shows the number of lines of code that should be used to create several programs using P2CL and plain OpenCL APIs. It is easy to see that writing GPU programs using P2CL requires much less effort.

Application	P2CL		OpenCL	
	function definitions & script	host program	kernel program	host program
vector addition	32	68	38	169
vector dot production	42	48	45	192
reshape	16	68	167	10

Table 9.2: Comparison of Programming Simplicity

Note that those hand-written programs using plain OpenCL APIs are naive implementations that are also used to compare throughput. Programs with optimizations usually require more code size.

9.2 Performance of P2CL over Naive OpenCL Programs

In addition to programming simplicity, P2CL also provides sufficient efficiency comparing to naive hand-written GPU programs. It is also worth mentioning that P2CL also introduces overhead that might hinder the performance. This section discusses the overhead and the performance gain of P2CL through the evaluations of several applications.

9.2.1 Elementwise Addition

Elementwise addition is a typical case of map pattern. In this application, each element in a input list is added with a scalar value for several times in order to vary the computation strength in a kernel. It is a simple and straight-forward application, which exists little space for optimizations. Therefore it is a good test case to evaluate the overhead of P2CL. The operations are performed on a list with 65536 elements. Varied number of additions are performed on each element to have different workload. Figure 9.1 shows the result of the evaluation on Device One.

The yellow line shows the performance of the hand-written OpenCL program. The execution time grows linearly with the workload. The orange line demonstrates the result when P2CL is used with only one thread and each input element is feed into the process object one by one. This approach introduces much overhead on the CPU as each call of the pushBack function involves index checking and invocations of some thread communication functions. There are two ways of reducing the latency. The first one is pushing input data in bulks instead of pushing data elementwise. The second one is pushing input data from

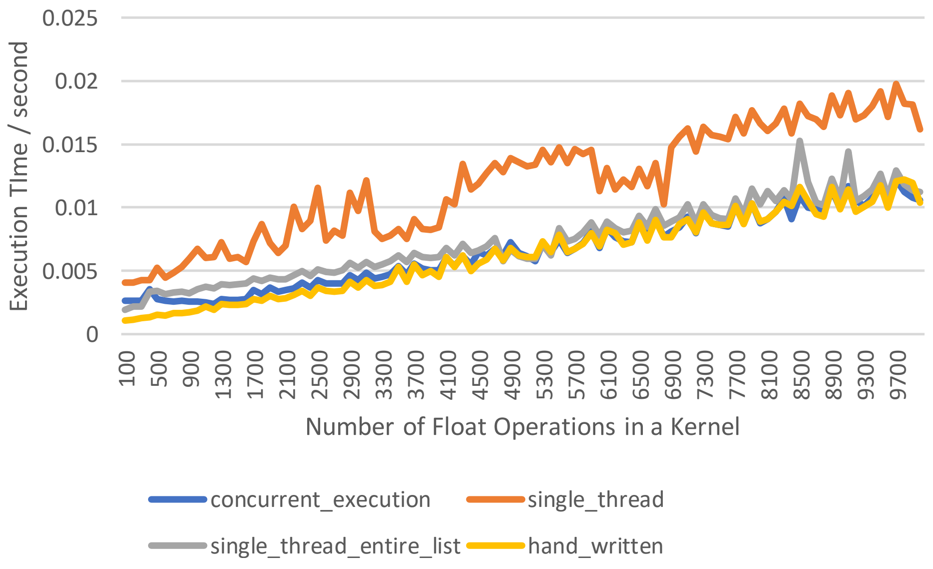


Figure 9.1: Overhead Evaluations with Elementwise Addition Application on Device One

a different thread and enabling the process object to allow concurrent execution. The first solution reduce invocations of pushBack function thus reduces index checking. This second solution fills the GPU with sufficient work while the CPU continue collecting input data. These two solutions are also illustrated in Figure 9.1. The gray line is collected with the entire input list enqueued in one pushBack function call. The blue line shows the performance of concurrent execution. Note that the execution time stays at almost the same value for current execution when the workload is small. That time can be seen as the overhead of pushing the input data elementwise to the process object since the concurrent execution on GPU is not large enough to hide the latency caused by the slow input enqueuing.

Since both the GPU and the CPU on Device two are much more powerful than those on Device one, the result is slightly different and is shown in Figure 9.2. The case that the entire input list is enqueued in one function call has similar overhead as the hand-written program. However, for the concurrent execution program, although it has better performance than the single thread version, it still has a constant overhead comparing to the hand-written version. Consider that the overhead is a constant value, it is still bearable when there are a large amount of computations in a program.

In order to test the limit of concurrent execution, another evaluation experiment is performed by toggling the allowed number of concurrent computations of a process object. The input lists in this experiment are all enqueued in only one function call. And there is only one floating point addition inside one kernel. Figure 9.3 shows the results collected on Device one. The vertical axies shows the total amount of floating point operations within one second. It reflects the throughput and performance. When only one instance of computation is allowed

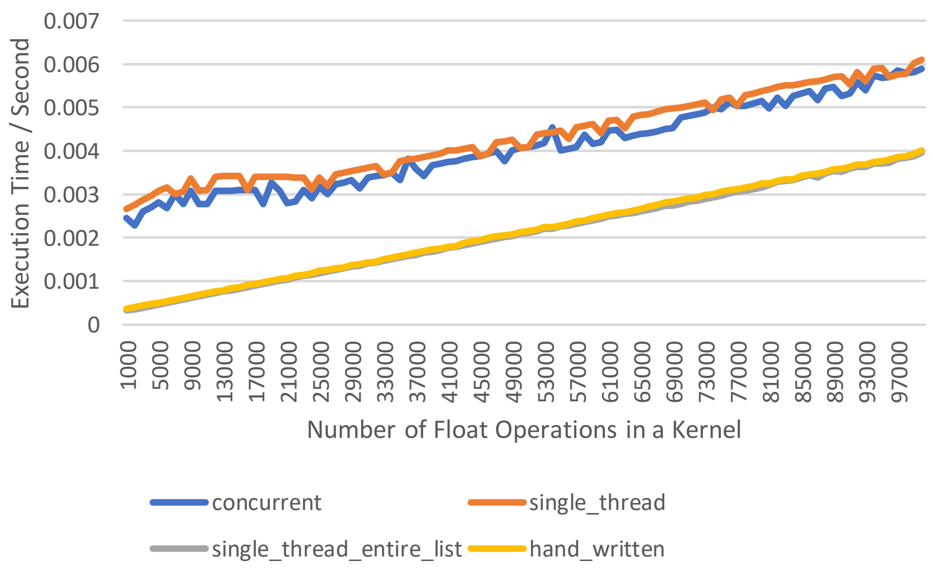


Figure 9.2: Overhead Evaluations with Elementwise Addition Application on Device Two

to be executed at a time, the throughput is very low, as in there is only one set of input buffer in this configuration. New input data has to wait until the results of the previous computation are popped out before it can be pushed to GPU. That is the reason for the significant increase in throughput when the allowed number of concurrent computations becomes two. The highest throughput is achieved when the number is five, this throughput gain is achieved because of the latency hiding between different works when they concurrently executed on GPU. However, there is no benefit to allow more concurrent computations, since the transfer speed between CPU and GPU then becomes the bottleneck. The similar results are achieved on Device Two, which are shown in Figure 9.4.

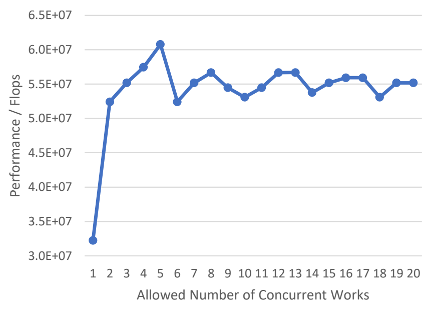


Figure 9.3: Concurrent Execution Evaluation with Elementwise Addition Application on Device One

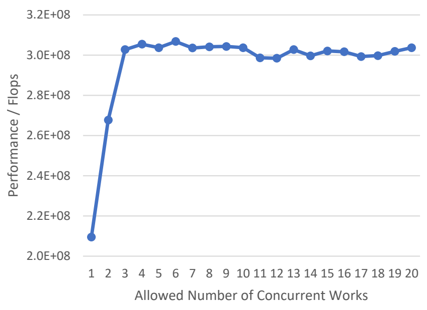


Figure 9.4: Concurrent Execution Evaluation with Elementwise Addition Application on Device Two

9.2.2 Vector Dot Production

Vector dot production is a simple application which can be composed using the map and reduce pattern. There are many optimization techniques that can be used for this computation. Here the comparison between the naive hand-written implementation and the optimized implementation created by P2CL is presented. On Device one, the application created using P2CL finishes the computation of vector production for the length of 131072 in 0.00204347 seconds. This evaluation is performed with the condition that all the input data is enqueued in one function call and the input and output are enqueued and popped in one thread. The reduce pattern in P2CL only allow a maximum of 32 elements in the last step of a reduce operation to be transferred and computed on CPU. This is a heuristic approach that tries to balance the limited parallelism in the last steps of reduce operation and the high cost of transferring between CPU and GPU, which can be improved in a future version. In order to have a good comparison, the hand-written counterpart also follows this rule to minimize the effect of the CPU speed. Different work-group sizes are toggled for the hand-written program to test the complete performance potential. The results are shown in Table 9.3.

work-group size	execution time / second
2	0.000853
4	0.000936
8	0.000887
16	0.001030
32	0.001067
64	0.001662
128	0.009099
256	0.012715

Table 9.3: Execution Time for Hand-written Vector Production on Device One

It can be seen that the P2CL program outperforms the hand-written program with work-group size 128 and 256. Consider that according to P2CL's execution plan, such application on Device one is performed with work-group size 256, the P2CL program only requires 1/6 of the execution time of the hand-written program to finish. The reason that P2CL tend to choose larger work-group is to have fewer iterations for the computation of the first map operation, which is beneficial when the mapped function requires much execution time. When adding 1000 floating point operations in the mapped operation, the P2CL program takes 0.00449777 seconds, while the results of the hand-written program shown in Table 9.4 are all higher than that.

On Device two, the unmodified program created using P2CL already shows better performance than the hand-written program with different work-group size. It takes 0.00039245 second for the P2CL program and the results of the hand-written program are shown in Table 9.5. By adding additional computations into the mapped function, the P2CL program has an even better advantage over the hand-written program. The detailed data collected for that will not be

workgroup size	execution time / second
2	0.005133
4	0.004874
8	0.004857
16	0.005598
32	0.006015
64	0.006394
128	0.010876
256	0.018389

Table 9.4: Execution Time for Hand-written Vector Production with Addtional Idle Computations on Device One

presented in this report.

workgroup size	execution time / second
2	0.000511
4	0.000492
8	0.000507
16	0.000450
32	0.000462
64	0.000540
128	0.000900
256	0.005469

Table 9.5: Execution Time for Hand-written Vector Production on Device Two

9.2.3 Transpose Operation

The transpose pattern is another pattern that many memory optimizations can be applied to. The evaluation is performed with varied length of the input list. The width and the height are set to be either the same or the closest values whose product equals the length. Figure 9.5 shows the results collected on Device Two. The program using P2CL outperforms the naive hand-written implementation starting from the very small list. As the list becomes larger, the effect of the optimizations becomes more obvious. On Device One, similar behavior can be seen in Figure 9.6. However, the program using P2CL only outperforms the hand-written program on Device One with very large list. This behavior is probably due to the special design of the integrated GPU that shares the same SoC as a CPU. The large shared last level caches might alleviate the performance penalty caused by strided accesses.

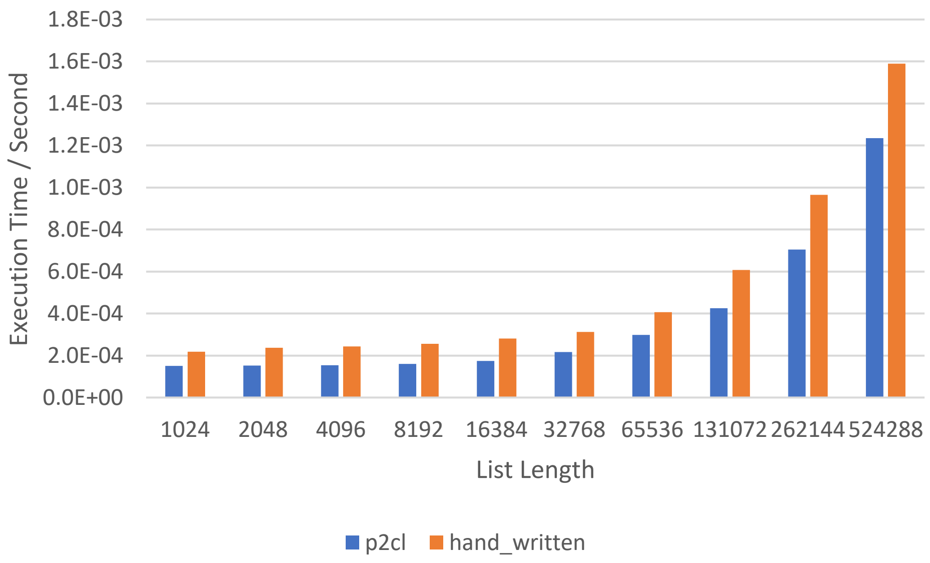


Figure 9.5: Transpose Operation Evaluation on Device Two

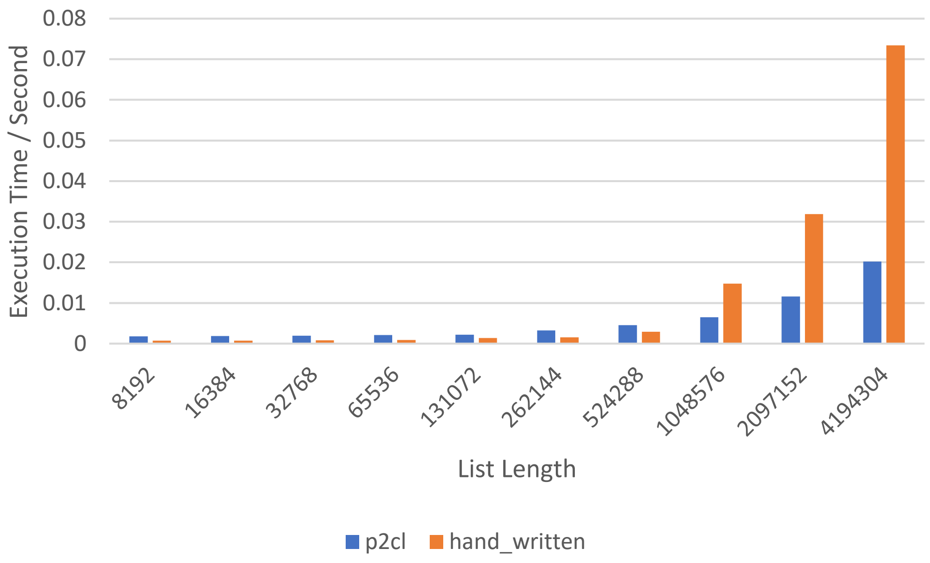


Figure 9.6: Transpose Operation Evaluation on Device One

Chapter 10

Future Works

Although currently the tool is capable of generating kernel programs and providing an easy-to-use API that allows the CPU side to interact with the GPU program in a similar way to interact with an SDF process. Much work should be done to firstly allow more complex systems and secondly make the model more intuitive to be understood and to be written.

10.1 Parallel Operations in SDF Processes

The first thing that is neccessary to improve is to add support for complete SDF graphs. Currently, in the model representation, all the parallel operations are embedded in one SDF process. There can be a sequence of operations in it like shown in Figure 10.1. The complete model that supports SDF networks should be like the one shown in Figure 10.2. The parallel operations specified by parallel patterns are still inside SDF processes. However, a sequence of parallel operations should be modeled by a sequence of SDF process, inside which there is only one parallel operation. The result of one process can feed to several other processes so that more complex systems can be modeled. In this way, the fusion optimizer can recognize more parallel operations that can be combined. Figure 10.2 (a) shows a network of processes where the result of a map operations feed to another map operation and a reduce operation in separate processes. Figure 10.2 (b) shows the model after fusion. As mentioned before, fusion should be applied whenever possible in order to reduce expensive global memory accesses. Thus, the first process that containing the map operation is coalesced to both the latter processes.

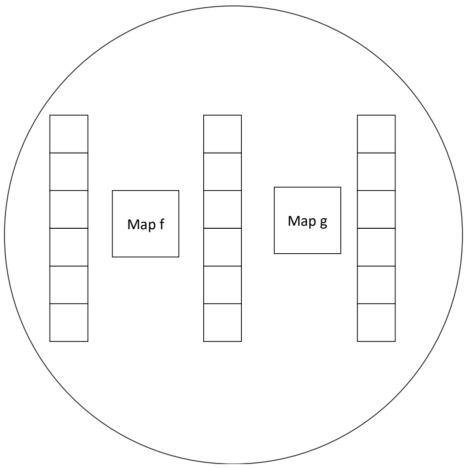
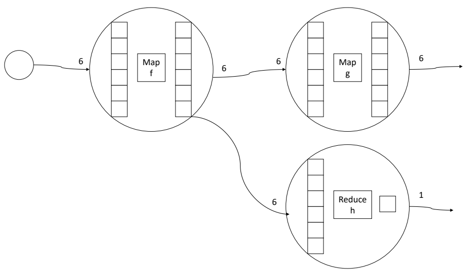
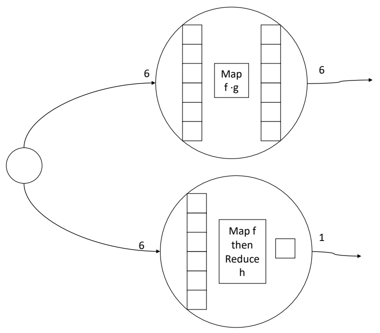


Figure 10.1: Parallel Operations inside a SDF process



(a) Before Fusion



(b) after Fusion

Figure 10.2: Parallel Operations in SDF networks

10.2 Scheduling SDF Network on Heterogeneous Platforms

Another idea regarding supporting SDF graphs is to allow the CPU and the GPU to execute different SDF processes and collectively complete the specified computations. Under this design, CPUs are assigned processes that require less parallelism. The processes containing parallel operations should be executed on GPUs. Scheduling of processes is critical to the throughput of this system. Figure 10.3 show such an example where process A and C are assigned to a CPU and process B is assigned to a GPU. Two scheduling plan can be applied to the system. One schedule is executing process A eight times which provides just enough data to one invocation of process B. After the invocation of process B, eight invocations of process C is followed. This is a typical SDF schedule plan for the minimization of buffer size. Another schedule is executing process A for twenty-four times to provide enough input data for three invocations of process B. Twenty-four times invocation of process C follows the execution of process

B. This schedule although requires more buffer size, it should show a better performance in terms of throughput, because more workloads can be concurrently executed to hide memory latencies. These considerations for scheduling can be applied to the future development of P2CL.

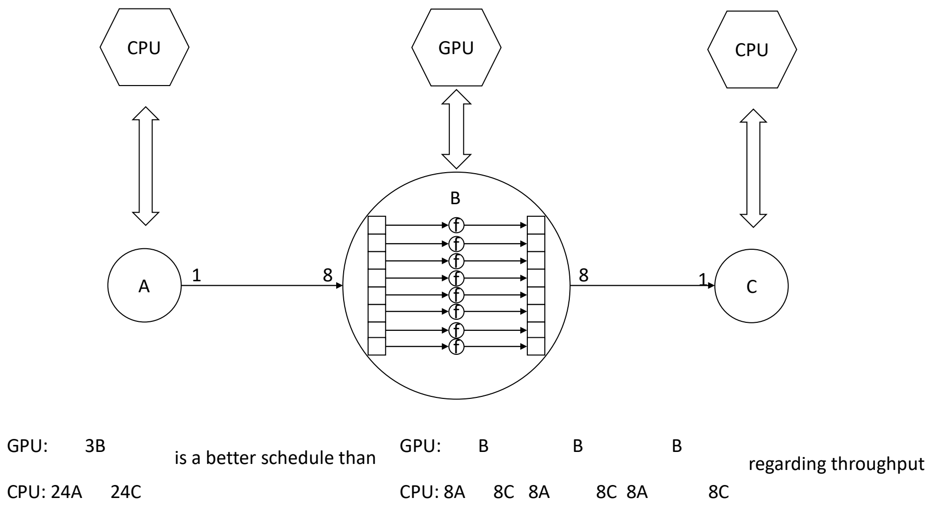


Figure 10.3: Scheduling of SDF Network

10.3 More Intuitive XML Representation

The XML representation of parallel patterns described in Chapter 6 reply on the user to specify the data types and length of an parallel operation. However, during the development of the tool, it becomes obvious that in many cases the data types and length can be derived from the input lengths and data types. Besides that, the operation map pattern can also be replaced by the divide and conquer paradigm that specify how input data can be split into equal sections instead of extending basic operations. These are differences of perspectives. The current XML representation describes a system from the perspective of operations However a representation from the perspective of data might be more intuitive for the user. This left space for improvement. A more intuitive and easy-to-written representation can be developed in the future.

10.4 Auto-Tuning

One set of kernel program and execution plan cannot achieve high performance on all GPU devices. The evaluation chapter also demonstrates that program created using P2CL cannot achieve good performance for all devices and different computation strength. In Chapter 2, the idea of auto-tuning is briefly introduced as a good approach to achieve performance portability. In the future development of P2CL, this approach may be applied to achieve better performance on different devices. One parameter that can be tuned to achieve better performance is the workgroup size. Varying it usually does not require modification of the kernel code, so it should be easier to achieve than tuning some other parameters.

Bibliography

[1] J. L. Hennessy and D. A. Patterson, "Data-level parallelism in vector, SIMD, and GPU architectures", in *Computer Architecture: A Quantitative Approach, 5th ed.* 225 Wyman Street, Waltham, MA 02451, USA: Morgan kaufmann, 2012, ch. 4, pp. 262–341.

[2] I. Sander and A. Jantsch, "System modeling and transformational design refinement in ForSyDe", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, no. 1, pp. 17–32, Jan. 2004. DOI: 10.1109/TCAD.2003.819898.

[3] G. Hjort Blindell, "Synthesizing software from a ForSyDe model targeting GPGPUs", Master's thesis, KTH Royal Institute of Technology, Stockholm, 2012.

[4] M. Garland and D. B. Kirk, "Understanding throughput-oriented architectures", *Commun. ACM*, vol. 53, no. 11, pp. 58–66, Nov. 2010, ISSN: 0001-0782. DOI: 10.1145/1839676.1839694. [Online]. Available: http://doi.acm.org/10.1145/1839676.1839694.

[5] J. Nickolls and D. Kirk, "Graphics and computing GPUs", in *Computer Organization and Design: The Hardware/Software Interface, 5th ed.* 225 Wyman Street, Waltham, MA 02451, USA: Morgan kaufmann, 2014, ch. Appendix-C, pp. C1–C83.

[6] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra, "From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming", *Parallel Computing*, vol. 38, no. 8, pp. 391–407, 2012.

[7] *The open standard for parallel programming of heterogeneous systems.* [Online]. Available: https://www.khronos.org/opencl/ (visited on 08/06/2017).

[8] J. Tompson and K. Schlachter, "An introduction to the OpenCL programming model", *Pearson Education*, vol. 49, 2012.

[9] Khronos Group *et al.*, "The OpenCL specification–version 1.2", *Khronos OpenCL Working Group.*, vol. 380, 2012.

[10] ——, "The OpenCL C specification–version 2.0", *Khronos OpenCL Working Group.*, vol. 205, 2016.

[11] *NVIDIA OpenCL best practices guide*, Aug. 2009. [Online]. Available: https://www.nvidia.com/content/cudazone/CUDABrowser/downloads/papers/NVIDIA_OpenCL_BestPracticesGuide.pdf (visited on 08/06/2017).

[12] *OpenCL optimization guide*. [Online]. Available: http://developer.amd.com/amd-accelerated-parallel-processing-app-sdk/opencl-optimization-guide/ (visited on 08/06/2017).

[13] *Adreno OpenCL Programming Guide*. [Online]. Available: https://developer.qualcomm.com/download/adrenosdk/adreno-opencl-programming-guide.pdf (visited on 08/06/2017).

[14] *Memory Statistics - Shared*. [Online]. Available: http://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/memorystatisticsshared.htm (visited on 08/18/2017).

[15] J. H. Lee, N. Nigania, H. Kim, K. Patel, and H. Kim, "OpenCL performance evaluation on modern multicore CPUs", *Sci. Program.*, vol. 2015, 4:4–4:4, Jan. 2016, ISSN: 1058-9244. DOI: 10.1155/2015/859491. [Online]. Available: https://doi-org.focus.lib.kth.se/10.1155/2015/859491.

[16] Y. Zhang, M. Sinclair, and A. A. Chien, "Improving Performance Portability in OpenCL Programs", in *Supercomputing: 28th International Supercomputing Conference, ISC 2013, Leipzig, Germany, June 16-20, 2013. Proceedings*, J. M. Kunkel, T. Ludwig, and H. W. Meuer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 136–150, ISBN: 978-3-642-38750-0. DOI: 10.1007/978-3-642-38750-0_11. [Online]. Available: https://doi.org/10.1007/978-3-642-38750-0_11.

[17] I. Sander, A. Jantsch, and S.-H. Attarzadeh-Niaki, "ForSyDe: System design using a functional language and models of computation", in *Handbook of Hardware/Software Codesign*, S. Ha and J. Teich, Eds. Dordrecht: Springer Netherlands, 2017, pp. 1–42, ISBN: 978-94-017-7358-4. DOI: 10.1007/978-94-017-7358-4_5-1. [Online]. Available: https://doi.org/10.1007/978-94-017-7358-4_5-1.

[18] G. Ungureanu, "Automatic software synthesis from high-level ForSyDe models targeting massively parallel processors", Master's thesis, KTH Royal Institute of Technology, Stockholm, 2013.

[19] K. Keutzer, A. R. Newton, J. M. Rabaey, and A. Sangiovanni-Vincentelli, "System-level design: Orthogonalization of concerns and platform-based design", *IEEE transactions on computer-aided design of integrated circuits and systems*, vol. 19, no. 12, pp. 1523–1543, 2000.

[20] *ForSyDe Website*. [Online]. Available: https://forsyde.ict.kth.se/trac (visited on 03/30/2017).

[21] A. Acosta, H. Woidt, I. Sander, and S.-H. Attarzadeh-Niaki, *ForSyDe deep*. [Online]. Available: https://github.com/forsyde/forsyde-deep (visited on 08/06/2017).

[22] S. H. A. Niaki, M. K. Jakobsen, T. Sulonen, and I. Sander, "Formal heterogeneous system modeling with SystemC", in *Specification and Design Languages (FDL), 2012 Forum on*, IEEE, 2012, pp. 160–167.

[23] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing", *IEEE Transactions on computers*, vol. 100, no. 1, pp. 24–35, 1987.

[24] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, "Synthesis of embedded software from synchronous dataflow specifications", *The Journal of VLSI Signal Processing*, vol. 21, no. 2, pp. 151–166, 1999.

[25] J. Fischer, S. Gorlatch, and H. Bischof, "Foundations of data-parallel skeletons", in *Patterns and Skeletons for Parallel and Distributed Computing*, F. A. Rabhi and S. Gorlatch, Eds. London: Springer London, 2003, pp. 1–27, ISBN: 978-1-4471-0097-3. DOI: `10.1007/978-1-4471-0097-3_1`. [Online]. Available: `https://doi.org/10.1007/978-1-4471-0097-3_1`.

[26] M. D. McCool, A. D. Robison, and J. Reinders, "Patterns", in *Structured parallel programming: patterns for efficient computation*, Elsevier, 2012, ch. 3.

[27] M. McCool, *Parallel Pattern 4: Gather*, 2010. [Online]. Available: `http://www.drdobbs.com/parallel-pattern-4-gather/222600465` (visited on 08/18/2017).

[28] Udacity, *Transpose Part 2 - Intro to Parallel Programming.* [Online]. Available: `https://youtu.be/CM4tB3NsKiY`.

[29] M. Steuwer, P. Kegel, and S. Gorlatch, "SkelCL - a portable skeleton library for high-level gpu programming", in *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, May 2011, pp. 1176–1182. DOI: `10.1109/IPDPS.2011.269`.

[30] D. B. Skillicorn, "The bird-meertens formalism as a parallel model", in *Software for Parallel Computation*, J. S. Kowalik and L. Grandinetti, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 120–133, ISBN: 978-3-642-58049-9. DOI: `10.1007/978-3-642-58049-9_9`. [Online]. Available: `https://doi.org/10.1007/978-3-642-58049-9_9`.

[31] G. Ungureanu, *Forsyde-atom*, `https://github.com/forsyde/forsyde-atom`, 2013.

[32] E. Chu and A. George, "The divide-and-conquer paradigm and two basic FFT algorithms", in *FFT Black Box: Serial and Parallel Fast Fourier Transform Algorithms*, Boca Raton, Florida, USA: CRC Press, 1999, ch. 3.

[33] R. Bird *et al.*, *A calculus of functions for program derivation.* Oxford University. Computing Laboratory. Programming Research Group, 1987.

[34] R. S. Bird, "An introduction to the theory of lists", in *Logic of Programming and Calculi of Discrete Design: International Summer School directed by F.L. Bauer, M. Broy, E.W. Dijkstra, C.A.R. Hoare*, M. Broy, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1987, pp. 5–42, ISBN: 978-3-642-87374-4. DOI: `10.1007/978-3-642-87374-4_1`. [Online]. Available: `https://doi.org/10.1007/978-3-642-87374-4_1`.

[35] S. Sankar and R. Hayes, "ADL an interface definition language for specifying and testing software", *SIGPLAN Not.*, vol. 29, no. 8, pp. 13–21, Aug. 1994, ISSN: 0362-1340. DOI: `10.1145/185087.185096`. [Online]. Available: `http://doi.acm.org.focus.lib.kth.se/10.1145/185087.185096`.

[36] S. Sato and H. Iwasaki, “A skeletal parallel framework with fusion optimizer for GPGPU programming”, in *Programming Languages and Systems: 7th Asian Symposium, APLAS 2009, Seoul, Korea, December 14-16, 2009. Proceedings*, Z. Hu, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 79–94, ISBN: 978-3-642-10672-9. DOI: 10.1007/978-3-642-10672-9_8. [Online]. Available: https://doi.org/10.1007/978-3-642-10672-9_8.

[37] R. Loogen, Y. Ortega, R. Peña, S. Priebe, and F. Rubio, “Parallelism abstractions in Eden”, in *Patterns and Skeletons for Parallel and Distributed Computing*, F. A. Rabhi and S. Gorlatch, Eds. London: Springer London, 2003, pp. 95–128, ISBN: 978-1-4471-0097-3. DOI: 10.1007/978-1-4471-0097-3_4. [Online]. Available: https://doi.org/10.1007/978-1-4471-0097-3_4.

[38] S. Pelagatti, “Task and data parallelism in P3L”, in *Patterns and Skeletons for Parallel and Distributed Computing*, F. A. Rabhi and S. Gorlatch, Eds. London: Springer London, 2003, pp. 155–186, ISBN: 978-1-4471-0097-3. DOI: 10.1007/978-1-4471-0097-3_6. [Online]. Available: https://doi.org/10.1007/978-1-4471-0097-3_6.

[39] C. Brown, K. Hammond, M. Danelutto, P. Kilpatrick, H. Schöner, and T. Breddin, “Paraphrasing: Generating parallel programs using refactoring”, in *Formal Methods for Components and Objects: 10th International Symposium, FMCO 2011, Turin, Italy, October 3-5, 2011, Revised Selected Papers*, B. Beckert, F. Damiani, F. S. de Boer, and M. M. Bonsangue, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 237–256, ISBN: 978-3-642-35887-6. DOI: 10.1007/978-3-642-35887-6_13. [Online]. Available: https://doi.org/10.1007/978-3-642-35887-6_13.

[40] D. Busvine, “Implementing recursive functions as processor farms”, *Parallel Computing*, vol. 19, no. 10, pp. 1141–1153, 1993, ISSN: 0167-8191. DOI: http://dx.doi.org/10.1016/0167-8191(93)90023-E. [Online]. Available: http://www.sciencedirect.com/science/article/pii/016781919390023E.

[41] *f2cc ForSyDe-2-CUDA C*. [Online]. Available: https://forsyde.ict.kth.se/trac/wiki/ForSyDe/f2cc (visited on 09/07/2017).

[42] G. H. Blindell, C. Menne, and I. Sander, “Synthesizing code for GPGPUs from abstract formal models”, in *Languages, Design Methods, and Tools for Electronic System Design: Selected Contributions from FDL 2014*, F. Oppenheimer and J. L. Medina Pasaje, Eds. Cham: Springer International Publishing, 2016, pp. 115–134, ISBN: 978-3-319-24457-0. DOI: 10.1007/978-3-319-24457-0_7. [Online]. Available: https://doi.org/10.1007/978-3-319-24457-0_7.

[43] *SkelCL a skeleton library for heterogeneous systems*. [Online]. Available: http://skelcl.uni-muenster.de (visited on 09/07/2017).

[44] *SkePU2 autotunable multi-backend skeleton programming framework for multicore CPU and multi-GPU systems*. [Online]. Available: http://www.ida.liu.se/labs/pelab/skepu/#publications (visited on 09/07/2017).

[45] M. Steuwer, M. Haidl, S. Breuer, and S. Gorlatch, “High-level programming of stencil computations on multi-GPU systems using the SkelCL library”, *Parallel Processing Letters*, vol. 24, no. 03, p. 1 441 005, 2014.

[46] U. Dastgeer, J. Enmyren, and C. W. Kessler, “Auto-tuning SkePU: A multi-backend skeleton programming framework for multi-GPU systems”, in *Proceedings of the 4th International Workshop on Multicore Software Engineering*, ser. IWMSE ’11, Waikiki, Honolulu, HI, USA: ACM, 2011, pp. 25–32, ISBN: 978-1-4503-0577-8. DOI: 10.1145/1984693.1984697. [Online]. Available: http://doi.acm.org.focus.lib.kth.se/10.1145/1984693.1984697.

TRITA-EECS-EX-2018:5