Technical report, IDE0931, June 2009

High-Level Parallel Programming of Computation-Intensive Algorithms on Fine-Grained Architecture

Master’s Thesis in Computer System Engineering

Fahad Islam Cheema

School of Information Science, Computer and Electrical Engineering
Halmstad University
High-Level Parallel Programming of Computation-Intensive Algorithms on Fine-Grained Architecture

Master’s thesis in Computer System Engineering

School of Information Science, Computer and Electrical Engineering
Halmstad University
Box 823, S-301 18 Halmstad, Sweden

June 2009

© 2009
Fahad Islam Cheema
All Rights reserved
Description of cover page picture:

Design diagram of width-6 and degree-2 MKLP in bi-cubic interpolation.
Preface

This project is the concluding part of Master’s in Computer System Engineering with specialization in Embedded Systems from Halmstad University, Sweden. Firstly, I would like to thank my supervisors Professor Bertil Svensson and Zain-ul-Abdin for continuous encouragement and feedback. I am very thankful to Mr. Bertil for providing concrete ideas and Mr. Zain for cooperating both in terms of time and frequency.

I also want to express my gratitude to my advisor Anders Åhlander from SAAB Microwave Systems, who was main source of information about interpolation kernels. Also I would like to thank many people from Mitrionics Inc for providing email correspondence and continuous feedback. Especially I would like to thank Stefan Möhl who is co-founder and chief scientific officer of Mitrionics Inc, for reviewing my thesis.

Finally, I would like to thank all those people who helped me to get that level of understanding of real world.

Fahad Islam Cheema
Halmstad University, June 2009
Abstract

Computation-intensive algorithms require a high level of parallelism and programmability, which make them good candidate for hardware acceleration using fine-grained processor arrays. Using Hardware Description Language (HDL), it is very difficult to design and manage fine-grained processing units and therefore High-Level Language (HLL) is a preferred alternative.

This thesis analyzes HLL programming of fine-grained architecture in terms of achieved performance and resource consumption. In a case study, highly computation-intensive algorithms (interpolation kernels) are implemented on fine-grained architecture (FPGA) using a high-level language (Mitrion-C). Mitrion Virtual Processor (MVP) is extracted as an application-specific fine-grain processor array, and the Mitrion development environment translates high-level design to hardware description (HDL).

Performance requirements, parallelism possibilities/limitations and resource requirement for parallelism vary from algorithm to algorithm as well as by hardware platform. By considering parallelism at different levels, we can adjust the parallelism according to available hardware resources and can achieve better adjustment of different tradeoffs like gates-performance and memory-performance tradeoffs. This thesis proposes different design approaches to adjust parallelism at different design levels. For interpolation kernels, different parallelism levels and design variants are proposed, which can be mixed to get a well-tuned application and resource specific design.
## List of Figures

| Figure 2.1 | Mitrion Software Development Flow [3] | 6 |
| Figure 2.2a | List Multiplication Example in Mitrion-C | 6 |
| Figure 2.2b | Syntax Error version of List Multiplication Example in Mitrion-C | 7 |
| Figure 2.3 | GUI Simulation of list multiplication example | 7 |
| Figure 2.4 | Batch Simulation results of list multiplication example | 8 |
| Figure 2.5 | List Multiplication Example specific to XD1 with Vertex-4 platform in Mitrion-C | 9 |
| Figure 2.6 | Resource analysis of list multiplication example | 9 |
| Figure 3.1 | Matlab program of one-dimensional interpolation | 16 |
| Figure 3.2 | Plot of one-dimensional interpolation in Matlab | 16 |
| Figure 3.3 | Bi-Cubic interpolation in terms of Cubic Interpolation | 17 |
| Figure 3.4 | Matlab program for Bi-Linear interpolation | 17 |
| Figure 3.5 | Plot of Bi-Linear and Bi-Cubic interpolation in Matlab | 18 |
| Figure 3.6 | Plot of Bi-Linear and Bi-Cubic interpolation in Matlab | 19 |
| Figure 3.7 | Cubic interpolation by Neville's Algorithm | 20 |
| Figure 3.8 | Calculating difference of interpolating point in cubic interpolation | 20 |
| Figure 3.9 | Cubic interpolation in Matlab | 21 |
| Figure 3.10 | Cubic interpolation for equally distant points in Matlab | 21 |
| Figure 3.11 | Bi-Cubic interpolation in Matlab | 22 |
| Figure 3.12 | Image interpolation using Bi-Cubic in Matlab | 23 |
| Figure 4.1 | Design diagram of sequential implementation of cubic interpolation | 28 |
| Figure 4.2 | Automatic-parallelization (APZ) of cubic interpolation in Mitrion-C | 29 |
| Figure 4.3 | Design Diagram of KLP in cubic interpolation | 29 |
| Figure 4.4 | Design Diagram of MKLP in cubic interpolation | 31 |
| Figure 4.5 | KLP in cubic interpolation using Mitrion-C | 31 |
| Figure 4.6 | KLP-LROP in cubic interpolation | 33 |
| Figure 4.7 | Design diagram of sequential implementation of bi-cubic interpolation | 34 |
| Figure 4.8 | APZ of bi-cubic interpolation using Mitrion-C | 35 |
| Figure 4.9 | Design Diagram of KLP in bi-cubic interpolation | 36 |
| Figure 4.10 | MKLP in bi-cubic interpolation | 36 |
| Figure 4.11 | Design view of Loop-roll-off parallelism of bi-cubic interpolation | 37 |
# List of Tables

Table 5.1. Results for non-equidistant Cubic Interpolation ............................................................. 40
Table 5.2. Results for equidistant Bi-cubic Interpolation .............................................................. 43
TABLE OF CONTENTS

PREFACE .......................................................................................................................................................................... VI

ABSTRACT .................................................................................................................................................................... VIII

LIST OF FIGURES ........................................................................................................................................................ IX

LIST OF TABLES .......................................................................................................................................................... X

TABLE OF CONTENTS ................................................................................................................................................ XI

1 INTRODUCTION ............................................................................................................................................................ 1
   1.1 MOTIVATION ....................................................................................................................................................... 1
   1.2 PROBLEM DEFINITION ..................................................................................................................................... 2
   1.3 THESIS CONTRIBUTION .................................................................................................................................... 3
   1.4 RELATED WORK .................................................................................................................................................. 3
   1.5 THESIS ORGANIZATION .................................................................................................................................... 3

2 MITRION PARALLEL ARCHITECTURE .......................................................................................................................... 5
   2.1 MITRION VIRTUAL PROCESSOR .......................................................................................................................... 5
   2.2 MITRION DEVELOPMENT ENVIRONMENT ....................................................................................................... 6
      2.2.1 Mitrion-C Compiler ......................................................................................................................................... 6
      2.2.2 Mitrion Simulator ............................................................................................................................................ 7
      2.2.3 Processor Configuration Unit .......................................................................................................................... 10
   2.3 MITRION-C LANGUAGE SYNTAX AND SEMANTICS ........................................................................................ 10
      2.3.1 Loop Structures of Mitrion-C ......................................................................................................................... 10
      2.3.2 Type System of Mitrion-C ............................................................................................................................. 11
   2.4 HARDWARE PLATFORMS SUPPORTED FOR MITRION .................................................................................... 12

3 INTERPOLATION KERNELS ........................................................................................................................................... 15
   3.1 INTRODUCTION .................................................................................................................................................. 15
      3.1.1 One-Dimensional Interpolation .................................................................................................................... 15
      3.1.2 Two-Dimensional Interpolation .................................................................................................................... 17
   3.2 MATHEMATICAL OVERVIEW ............................................................................................................................ 19
   3.3 IMPLEMENTATION IN MATLAB ........................................................................................................................ 20
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3.1</td>
<td>Cubic Interpolation Kernel in Matlab</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Bi-Cubic Interpolation Kernel in Matlab</td>
</tr>
<tr>
<td>3.4</td>
<td>IMAGE INTERPOLATION BY 2D INTERPOLATION</td>
</tr>
<tr>
<td>4</td>
<td>IMPLEMENTATION</td>
</tr>
<tr>
<td>4.1</td>
<td>IMPLEMENTATION SETUP AND PARAMETERS</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Hardware Platform details</td>
</tr>
<tr>
<td>4.2</td>
<td>IMPLEMENTATION OF CUBIC INTERPOLATION</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Kernel-Level Parallelism (KLP)</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Problem-Level Parallelism (PLP) in Cubic Interpolation</td>
</tr>
<tr>
<td>4.3</td>
<td>IMPLEMENTATION OF BI-CUBIC INTERPOLATION</td>
</tr>
<tr>
<td>4.3.1</td>
<td>KLP in Bi-Cubic Interpolation</td>
</tr>
<tr>
<td>4.3.2</td>
<td>PLP in Bi-Cubic Interpolation</td>
</tr>
<tr>
<td>4.4</td>
<td>OTHER MEMORY BASED IMPLEMENTATION VARIANTS</td>
</tr>
<tr>
<td>5</td>
<td>RESULTS</td>
</tr>
<tr>
<td>5.1</td>
<td>RESULTS FOR CUBIC INTERPOLATION</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Performance Analysis</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Resource Analysis</td>
</tr>
<tr>
<td>5.2</td>
<td>RESULTS FOR BI-CUBIC INTERPOLATION</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Performance Analysis</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Resource Analysis</td>
</tr>
<tr>
<td>6</td>
<td>CONCLUSIONS</td>
</tr>
<tr>
<td>6.1</td>
<td>SUGGESTIONS AND IDEAS</td>
</tr>
<tr>
<td>6.2</td>
<td>FUTURE WORK</td>
</tr>
<tr>
<td>7</td>
<td>REFERENCES</td>
</tr>
</tbody>
</table>

xii
1 Introduction

1.1 Motivation

Computation demands of embedded systems are increasing continuously, resulting in more complex and power consuming systems. Due to clock speed and interconnection limitations, it is becoming hard for single processor architectures to satisfy performance demands. Hardware improvements in uni-processor systems like superscalar and Very Long Instruction Word (VLIW) require highly sophisticated compiler designs but are still lacking in fulfilling continuously increasing performance demands.

To achieve better performance, application specific hardware was another requirement which highlighted the importance of reconfigurable architectures resulting in the emergence of fine-grained architectures like Field Programmable Gate Arrays (FPGA). On these reconfigurable architectures, von-Neumann architecture based processors could also be designed. These are called coarse-grained reconfigurable architectures which introduce another concept, Multiprocessor System on Chip (MPSOC).

On the software side, sequential execution and pipelining are unable to fulfil response time requirements, especially for real-time application. This has resulted in a new trend of hardware software co-design. By hardware software co-design, concurrent part of the design (having no data dependence) is considered as hardware and implemented in HDL. A sequential part of the design is considered as software and implemented in HLL. Due to high reconfigurability demands, most of the hardware software co-design architectures were built on fine-grained architectures. However, this requires complete understanding of both hardware designing and software programming, resulting in tremendous increase in development time and a need of skilled labour.

Although fine-grain processor arrays can achieve better parallelism, coarse-grain architectures are more common and are suitable for most applications, mainly due to their simple architecture and easy programmability. It is easier to design an HLL and compiler for coarse-grain architectures
High-Level Parallel Programming of Computation-Intensive Algorithms on Fine-Grained Architecture

than for fine-grained architecture. On the other hand, use of HLL for fine-grained architecture can achieve better parallelism.

Computation-intensive algorithms are becoming a bottleneck for achieving high performance. Parallelising the whole system creates more complexities in system design, wastage of resources and a tremendous increase in design time and cost. One common approach to deal with this problem is to design a separate, application-specific hardware, only for these computation-intensive algorithms, to accelerate the performance of the whole system.

Numerical computing and image processing algorithms in real-time environments like radar systems are highly computation-intensive. Also, these algorithms require a high-level of parallelism as well as programmability, to adjust parallelism according to hardware resources and memory characteristics.

1.2 Problem Definition

For High Performance Computing (HPC) and supercomputing systems, computation-intensive algorithms are becoming bottleneck for achieving the required performance. Computation-intensive algorithms require high-level of parallelism, but also programmability to achieve flexibility. By fine-grained architectures, high-level of parallelism and programmability could be achieved, but they are difficult to program.

Using HDL, it is very difficult to design and manage fine-grained processing units and HLL is a better solution for that. HLL like Mitrion-C for producing parallelism on fine-grained architectures could resolve this problem. It should be evaluated that how much programmability and parallelism is possible by using these HLL for fine-grained architectures.

Demand of parallelism is always application-specific and maximum possible parallelism changes according to available hardware resources like hardware gates, memory interface and memory size. It is highly desired to develop techniques to adjust parallelism according to available
hardware resources (gates or logic) and memory characteristics. Therefore, design approaches to be used in performance and resource tuning are also treated.

1.3 Thesis Contribution

Computation-intensive algorithms (interpolation kernels) are implemented on fine-grained architecture (FPGA) using a high-level language (Mitrion-C). Different sequential and parallel implementations of interpolation kernels are designed and evaluated in terms of performance and resource consumption. On the basis of these implementations and analyses, we have proposed different design levels to adjust parallelism according to hardware resources and memory characteristics.

1.4 Related Work

A number of parallel architectures have been developed recently. Some famous examples are Ambric [10] and RAW [11]. Also, there have been many MPSOC architectures on FPGA, such as Microblaze and PowerPC based SOC. From the perspective of the programming model, many parallel programming models are available for coarse grained parallel architectures but for fine-grained parallel architectures, programming models are very rare.

Fine-grained architectures are difficult to program and, due to that, most of the research orientation in parallel computing is towards coarse grained architectures. However, with the availability of HLL (High-Level Languages) for fine-grained architectures, these architectures will become more common in parallel applications. The idea of using fine-grained architecture for data interpolation and trade-off between memory size and performance were proposed in [1].

1.5 Thesis Organization

The remainder of this thesis is organized as follows. Mitrion Parallel architecture, including Mitrion Virtual Processor (MVP), Mitrion-C and Mitrion-SDK are discussed in Chapter 2. Chapter 3 discusses interpolation kernels. Also, simple sequential interpolation implementations are described in this chapter.
High-Level Parallel Programming of Computation-Intensive Algorithms on Fine-Grained Architecture

Design and implementations of different parallelism approaches are discussed in Chapter 4. Also, different parallelism levels are described which can parameterize parallelism according to application and available resources, to adjust different hardware resource-to-performance tradeoffs.

Analysis and results of our implementation are discussed in Chapter 5. On the basis of implementation approaches and analysis, trade-off adjustment techniques and other ideas are concluded in Chapter 6.
2 Mitrion Parallel Architecture

The main goal of Mitrion parallel architecture is to allow the software programmer to design hardware circuits in HLL without learning HDL and circuit designing concepts [4]. It is rather different from simple C-to-RTL translators as it allows the software programmer to attain fine-grain parallelism in HLL.

For non-floating-point operations like integer and fixed-point operations FPGAs are much faster than traditional Central Processing Units (CPUs) and General Purpose Graphical Processing Units (GPGPUs) but for floating-point operations GPGPUs are better. The future is expected to be heterogeneous and different types of hardware accelerators will be in high demand for of-chip hybrid computing. Mitrion parallel architecture is mainly designed for developing hardware accelerators for HPC and supercomputing systems.

2.1 Mitrion Virtual Processor

The Mitrion platform is based on the Mitrion Virtual Processor (MVP) which is a fine-grained, massively parallel, soft-core processor. There are almost 60 IP tiles which are combined in different ways to configure MVP. These tiles are designed in HDL to perform different functions like arithmetic, IO and control operations. This tile set is Turing complete which means there will always be a configuration for any possible code segment [2] [4].

Any algorithm designed in Mitrion-C is first compiled, and then an MVP configuration is generated. This configuration is normally an IP core which is passed through the place-and-route procedure and then implemented on FPGA. MVP behaves as the abstraction layer between software and hardware. Also, it takes care of all circuit designing, and allows the software designer to be unaware from the hardware details [3].
2.2 Mitrion Development Environment

Mitrion Software Development Kit (SDK) consists of Mitrion-C Compiler, Mitrion simulator and processor configuration unit. To produce parallelism, the Mitrion development environment relies on data dependence between program segments instead of order-of-execution. The Mitrion software development flow is shown in Figure 2.1.

![Diagram](image)

Figure 2.1. Mitrion Software Development Flow [3]

2.2.1 Mitrion-C Compiler

Compilation is the first step of software development. Mitrion-C compiler applies a syntax check and other compilation steps on the Mitrion-C program. The Mitrion-C compiler itself reveals parallelism to the designer, e.g. if some program segment does not have any data dependence with another segment of the program, then these segments could be run in parallel. However, if the programmer mistakenly implements these programs as sequential then the Mitrion-C compiler shows a syntax error to specify parallelism [4]. To demonstrate this, a simple Mitrion-C program which multiplies two lists of numbers is shown in Figure 2.2 a.

```mitrion-c
int:16<100> main(int:16<100> list1, int:16<100> list2)
{
    result = foreach(element_list1, element_list2 in list1, list2)
        result = (element_list1 * element_list2);
} result;
```

Figure 2.2 a. List Multiplication Example in Mitrion-C
If we change this program to a sequential version, as shown in Figure 2.2 b, then the Mitrion-C compiler will display syntax error. Mitrion-C language details are described in section 2.3.

```c
int:16<100> main(int:16<100> list1, int:16<100> list2)
{
    result = for(element_list1, element_list2 in list1, list2)
    result = (element_list1 * element_list2);
} result;
```

Figure 2.2 b. syntax Error version of List Multiplication Example in Mitrion-C

### 2.2.2 Mitrion Simulator

A Mitrion simulator is used for functional verification of the Mitrion-C program. Also, for simulation, it can get data from files. The Mitrion simulator can operate in three different ways: Graphical User Interface (GUI), and batch or server simulation mode.

- **GUI mode of simulator:** This demonstrates data dependency and flow of program in graphical form.

![Figure 2.3. GUI Simulation of list multiplication example](image)
Also GUI simulation provides a step-by-step execution of code. It is not cycle accurate, so is mostly used for functional verification of programs. A GUI simulation of list multiplication example is shown in Figure 2.3.

- **Batch simulation**: This is cycle accurate and directly produces results. This is the reason it is normally used to verify results and performance analysis. Batch simulation results for list multiplication example are shown in Figure 2.4.

```
Starting simulator
Reading simulation data from files
Completed simulation in 103 steps
Writing results of simulation run to file.
Done
```

Figure 2.4. Batch Simulation results of list multiplication example

- **Server simulation** is used to simulate the interaction between the host program and FPGA. This is called Virtual Server Simulation. Virtual server simulation is very useful when we want to simulate complex tasks like image processing. One good example for server simulation is the sobel example (provided with Mitrion SDK) which gets an image from the server and, after applying sobel filtering, creates another image.

- Another good use of server simulation is to access Mitrion Remote Server, which is accessible through the internet. The main purpose of Mitrion Remote Server is to get resource analysis information of design without actually implementing design on actual hardware. To perform resource analysis, the design must be changed according to a specific hardware platform.

Mitrion-C changes some language semantics according to the hardware platform, like the parameters to the ‘main’ function. By that time, Mitrion is supporting four hardware platforms. We must change our program according to one of those hardware platforms. In this thesis, we did implementations for a “Cray XD1 with Vertex-4 LX160 FPGA” hardware
platform. Platform relevant changes in Mitrion-C language and other hardware platform details are described in section 2.4. A platform-specific implementation of a list multiplication example is shown in Figure 2.5.

```mitron-c
(EXTRAM, EXTRAM, EXTRAM, EXTRAM)

main(EXTRAM mem_a_00, EXTRAM mem_b_00, EXTRAM mem_c_00, EXTRAM mem_d_00)
{
   (vectorY_lv, vectorX_lv, mem_a_02, mem_b_02) = foreach (i in <0 .. 3>)
   {
      (vectorY_v, mem_a_01) = memread(mem_a_00, i); // Reading from memory bank a
      (vectorX_v, mem_b_01) = memread(mem_b_00, i); // Reading from memory bank b
   } (vectorY_v, vectorX_v, mem_a_01, mem_b_01);

   (result_lv) = foreach (vectorY_v, vectorX_v in vectorY_lv, vectorX_lv)
   {
      result_v = vectorY_v*vectorX_v;
   } result_v;

   mem_c_02 = foreach (result_v in result_lv by i)
   {
      mem_c_01 = memwrite(mem_c_00, i, result_v); //Writing to memory bank c
   } mem_c_01;

   (mem_a_02, mem_b_02, mem_c_02, mem_d_00);
}
```

Figure 2.5. List Multiplication Example specific to XD1 with Vertex-4 platform in Mitrion-C

Unlike GUI and Batch simulators, the Mitrion Remote Server simulator also verifies different resource limitations and reports errors if some resource limitations are violated. Results produced by resource analysis are shown in Figure 2.6.

----------[ Estimated execution time: 10.0ns, 1 steps@100MHz
----------[ Estimated executions per second: 100000000.0
----------[ Creating Mitrion Virtual Processor for platform Cray XD1 LX160
    Target FPGA is Xilinx Virtex-4 LX xc4vlx160-10-ff1148
    Target Clock Frequency is 100MHz
    20062 Single Flip Flops
    + 1224 16-bit shiftregisters
    = 21286 Flip Flops out of 152064 = 13% Flip Flop usage
    17 BlockRAMs out of 288 = 5% BlockRAM usage
    64 MULT18X18s out of 96 = 66% MULT18X18 usage
----------[ Done

Figure 2.6. Resource analysis of list multiplication example
2.2.3 Processor Configuration Unit

MVP is based on a different processor architecture which is specifically designed to support parallelism on FPGA. Unlike traditional von-Neumann architecture, MVP architecture does not have any instruction stream [2]. The logic of MVP which performs instructions is called ‘program element’.

The process configuration unit adopts the MVP, according to custom application, by creating a point-to-point, or switched network connection with an appropriate processing element. Also, these processing elements are adaptive to bus width from 1-bit to 64-bits [4]. This results in a processor that is fully adapted to high-level description, and also it is parallel at the single instruction level.

2.3 Mitrion-C Language Syntax and Semantics

To utilize full parallelism from MVP, an instruction level fine-grained programming language was required. Due to lack of research on fine-grained programming languages, Mitrion designed their own parallel HLL, which is very similar to C language in syntax. Assuming that the reader is aware of ANSI-C, only the differences in Mitrion-C are highlighted.

2.3.1 Loop Structures of Mitrion-C

Loop parallelism is the main source of parallelism for Mitrion-C which is very suitable to deal with performance-intensive loops. In addition to simple ANSI-C loop structures Mitrion-C have another loop structure, named ‘foreach’. **Foreach loop** is very much like concurrent loop structures in concurrent languages, like ADA, or like ‘process’ and ‘procedure’ in HDL.

Unlike for loop, every statement inside foreach loop executes concurrently with every other statement. Order-of-execution within foreach is not guaranteed, and all the instructions inside a foreach loop must be data independent, as discussed in section 2.2.1.
Considering the list multiplication example in Figure 2.2 a, if we have more statements inside this ‘foreach’ loop then all of these statements will be running in parallel. Inside the ‘foreach’ loop, all of the statements must be data independent as described in previous section.

Another noticeable thing about the Mitrion-C loop structure is that it has a return type, so both functions and loops in Mitrion-C have return types but unlike functions. Also, loops can return more than one value. We must, therefore, specify which values we want to return from the loop and where we want to store them after completion of the loop. Only loop dependent variables could be returned from ‘for’ and ‘while’ loops. Also these loop dependent variables must be initialized outside the loop.

2.3.2 Type System of Mitrion-C

Mitrion-C has **scalar data types** like ‘int’, ‘float’ and ‘bool’ etc [2]. Except float these scalar data types are similar to ANSI-C. Float data type is defined in terms of mantissa and exponent so that different floating-point precisions could be defined according to application requirements. Also this definition of float is helpful to draw fix-point operations.

Mitrion-C also has **collection data types** like list, vector and memory which are slightly similar to collective data types of HDLs.

- **List data type**: This behaves as a ‘singly link list’, which means we can only access single element in the order that it exists in memory [4]. List data type is represented with “<>” and normally used to produce pipeline execution.

- **Vector data type**: All elements of vector data type will be executed simultaneously so it is normally used to produce full parallelism or loop-roll-off. Vector data type is denoted by “[]”, and it is a great source of parallelism in Mitrion-C.

- **Memory Data Type**: Similar to list data type, memory data type is also sequential but it could be accessed randomly. Memory data type is used when we want to read/write data
from/to physical memory, which is similar to HDL, type conversion between different collection types is also possible in Mitrion-C.

By using ‘foreach’ loop with ‘vector’, all loop iterations will run in parallel. Simply, it will roll off the whole loop which will cost high in terms of hardware resources. A simple example could be to replace list ‘<>’ with vector ‘[]’ in Figure 2.2a. On the other hand, by using ‘list’ with ‘foreach’ loop, only instructions inside a single iteration will run in parallel but all iterations will run in sequence. In that case, we will design hardware for only a single iteration and will reuse it for all iterations.

2.4 Hardware Platforms Supported for Mitrion

Mitrion support four hardware platforms. The detailed description of these hardware platforms is available from their relevant websites.

- **RC100**: A platform of SGI Inc having LX200 Vertex-4 FPGA with 2 memory banks of 128MB each.
- **XD1 with Vertex-4**: A platform of Cray Inc having LX160 Vertex-4 FPGA with 4 memory banks of 64MB each.
- **XD1 with Vertex-2**: A platform of Cray Inc having VP50 Vertex-2 FPGA with 4 SRAM memory banks of 64MB each.
- **H101**: A platform of Nallatech Inc having LX100 Vertex-4 FPGA with 4 SRAM memory banks of 64MB each and an SDRAM memory bank.

All implementations performed in this thesis are directed to “Cray XD1 with Vertex-4 LX160” hardware platform. The reasons for using this platform are described in chapter 4. The hardware platform not only affects implementation but also performance and resource consumption. Some
platform-specific changes are required to implement a design on actual hardware or to perform resource analysis using Mitrion remote server. Mainly the difference is due to different memory architecture in different platforms.

Assuming that the reader is familiar with FPGA and its resource details, only relevant resource details of Xilinx Virtex-4 LX xc4vlx160-10-ff1148 [5] are listed below.

- Number of Flip-Flops = 152064
- Number of Block-RAM = 288
- Number of 18X18 Multipliers = 96
High-Level Parallel Programming of Computation-Intensive Algorithms on Fine-Grained Architecture
3 Interpolation Kernels

3.1 Introduction

If some data values are available at some discrete data points and we want to construct new data points within the range of available data points then this process is called interpolation. Interpolation is also used for curve fitting. Curve fitting is a technique for finding a function to draw a curve that passes through all or maximum data points [6].

3.1.1 One-Dimensional Interpolation

Interpolation kernels can be categorized by dimension. More than 2-dimension interpolation methods are very uncommon due to high computation demands.

There are many types of 1D interpolation kernels like Linear, Nearest-Neighbour, Cubic and Spline interpolation which are used to interpolate data-points in one-dimension.

- **Linear interpolation:** This is one of the simplest interpolation methods. It simply joins all available data points linearly with each other, and draws all interpolation points on that curve. Linear interpolation is relatively less computation-intensive but it is highly inaccurate.

- **Nearest-Neighbour interpolation:** This interpolates the new points to the nearest neighbour. It is also efficient in terms of computation but has high interpolation error rates, which makes it unsuitable for applications demanding high-level of accuracy, like radar systems.

To illustrate the difference between the various one-dimensional interpolation methods, a simple Matlab program is shown in Figure 3.1. In this section, we have used Matlab built-in functions like `interp1` and `interpl2` to draw interpolations.
High-Level Parallel Programming of Computation-Intensive Algorithms on Fine-Grained Architecture

\[
\begin{align*}
  x &= 0:10; \\
  y &= \tan(x); \\
  x_{\text{int}} &= 0:.1:10; \\
  y_{\text{int\_NN}} &= \text{interp1}(x, y, x_{\text{int}}, '\text{nearest}'); \\
  y_{\text{int\_L}} &= \text{interp1}(x, y, x_{\text{int}}, '\text{linear}'); \\
  y_{\text{int\_C}} &= \text{interp1}(x, y, x_{\text{int}}, '\text{cubic}'); \\
\end{align*}
\]

\[
\text{plot}(x, y, 'o', x_{\text{int}}, y_{\text{int\_NN}}, '-', x_{\text{int}}, y_{\text{int\_L}}, '+', x_{\text{int}}, y_{\text{int\_C}}, '-'); \\
\text{legend}('\text{raw data}', '\text{Nearest Neighbour Interpolation}', '\text{Linear Interpolation}', '\text{Cubic Interpolation}');
\]

**Figure 3.1. Matlab program of one-dimensional interpolation**

- **Cubic interpolation** is a type of polynomial interpolation which constructs cubic polynomial using the four nearest points to determine the interpolation point. The result of cubic interpolation is much better than linear and nearest-neighbour which can be seen in Figure 3.2. Other polynomial interpolations, like quadratic interpolation, are also possible. Cubic interpolation is, nevertheless, a good compromise in terms of computation demands and interpolation error rate. It is for this reason that cubic interpolation is more common than other interpolation kernels.

![Figure 3.2. Plot of one-dimensional interpolation in Matlab](image)

Nearest neighbour interpolation behaves as a step function as shown in Figure 3.2. Linear interpolation linearly joins all interpolation points which are relatively closer to cubic interpolation.
3.1.2 Two-Dimensional Interpolation

Interpolation strategies could be used for two-dimensions, like Bi-Linear and Bi-Cubic interpolation. 2D interpolation kernels are obviously more computation expensive than 1D, but for applications that require computation in 2D, like image-processing applications, 2D interpolation is necessary. One simple method of 2D interpolations is to apply 1D interpolation in one dimension, to extract interpolation points in one dimension and then apply the same 1D interpolation on these interpolation points. Bi-cubic interpolation in terms of 1D interpolation is shown in Figure 3.3. On the 4*4 grid first we apply cubic interpolation on each vertical vector to calculate interpolation points which are highlighted as ‘+’ in figure. Then again we apply cubic interpolation on these interpolation points final bi-cubic Interpol highlighted as ‘O’ in the Figure.

![Figure 3.3. Bi-Cubic interpolation in terms of Cubic Interpolation](image)

In Figure 3.4, Matlab code for Bi-linear interpolation is shown. By exchanging ‘linear’ with ‘nearest’ and ‘cubic’ in interp2 function, the implementation was changed for Bi-nearest-neighbour and Bi-cubic interpolations respectively.

```matlab
[x, y] = meshgrid(-1:.1:1);
z = tan(x);
[xi, yi] = meshgrid(-1:.05:1);
zi = interp2(x, y, z, xi, yi, 'linear');
surf(xi, yi, zi), title('Bi-Linear interpolation for tan(x)')
```

![Figure 3.4. Matlab program for Bi-Linear interpolation](image)
To illustrate the difference between Bi-linear and Bi-nearest-neighbour interpolation, $\tan(x)$ function was implemented for both. By $\tan(x)$ function, we are only changing x dimension for both interpolations. Figure 3.5 illustrates that the bi-nearest-neighbour is behaving as a step function and has a crest in the diagram, while bi-linear interpolation does not have this ladder effect.

**Bi-Cubic interpolation** just applies cubic interpolation in 2D. By changing data only in one-dimension, we can easily realize the difference between Bi-linear and Bi-nearest-neighbour interpolation. This is not true for realizing the difference between Bi-Linear and Bi-Cubic. The difference of Bi-Cubic and Bi-linear interpolation is illustrated by drawing both for the $\tan(x+y)$ function in Figure 3.6. As with cubic, bi-cubic is more common than other 2D interpolations due to lower interpolation error at nominal computation cost. This thesis will mainly focus on Cubic and Bi-cubic interpolation.

As Bi-linear interpolation linearly interpolates in both dimensions so it has more sharp edges than bi-cubic interpolation as shown in Figure 3.6. Due to this, bi-linear interpolation loses much more data at edge points which increases the interpolation error.
One purpose of interpolation is to adopt the change and when sudden changes occur it cause more interpolation error. A good interpolation kernel is expected to adopt changes and with minimum interpolation error, describe the change to maximum possible detail. Bi-cubic interpolation better adopt the change with less interpolation error than bi-linear interpolation as clear from Figure 3.6.

![Figure 3.6. Plot of Bi-Linear and Bi-Cubic interpolation in Matlab](image)

### 3.2 Mathematical Overview

Cubic interpolation is a type of polynomial interpolation. The concept behind cubic interpolation is that, for any four points there exists a unique cubic polynomial to join them [8]. Neville’s algorithm is mostly used for implementing polynomial interpolation on digital systems as it is easier to implement. Neville’s algorithm is based on Newton’s method of polynomial interpolation [7] and is a recursive way of filling values in tableau [8].

For calculating cubic interpolation, we apply Neville’s algorithm to four data points which are closest to interpolation point. Now we have four x and y values which we arrange in the form of four polynomial equations. From these four equations, we can calculate the four unknown values in different ways, like Gaussian elimination. Neville’s algorithm is preferred as it is relatively less complex and computation-intensive. Step-by-step, we resolve these equations as shown in Figure 3.7 and calculate four values which we use to calculate the interpolation point.
Polynomials of Figure 3.7 like $p_{01}$, $p_{12}$, $p_{23}$ are illustrated in Figure 3.8. For interpolating a point in cubic interpolation, we select the four nearest neighbours of that point, and calculate the difference of the interpolating point with these points. These differences are used in cubic polynomials as shown in Figure 3.8. Let us say that we want to interpolate value at 1.5,

\[
\begin{align*}
d_0 &= 1.5 - x_0 \\
d_1 &= 1.5 - x_1 \\
d_2 &= 1.5 - x_2 \\
d_3 &= 1.5 - x_3 \\
p_{01} &= (y_1*d_1 - y_2*d_0) / (x_1 - x_2); \\
p_{12} &= (y_2*d_2 - y_3*d_1) / (x_2 - x_3); \\
p_{23} &= (y_3*d_3 - y_4*d_2) / (x_3 - x_4); \\
p_{02} &= (p_{01}*d_2 - p_{12}*d_0) / (x_1 - x_3); \\
p_{13} &= (p_{12}*d_3 - p_{23}*d_1) / (x_2 - x_4); \\
p_{03} &= (p_{02}*d_3 - p_{13}*d_0) / (x_1 - x_4);
\end{align*}
\]

3.3 Implementation in Matlab

3.3.1 Cubic Interpolation Kernel in Matlab

Following Figure 3.7 and Figure 3.8, a simple Matlab implementation of cubic interpolation is shown in Figure 3.9. First, we are calculating interpolation difference and then we are using it to draw cubic interpolation using Neville’s algorithm.
function Cubic()  
  x = [0 1 2 3];  
  y = [0 1 2 3];  
  x_int = 1.5;  
  sqrt_nof_interpols = 300;  
  for i = 1:(sqrt_nof_interpols * sqrt_nof_interpols),  
    d0 = x_int - x(1);  
    d1 = x_int - x(2);  
    d2 = x_int - x(3);  
    d3 = x_int - x(4);  
    p01 = (y(1)*d1 - y(2)*d0) / (x(1) - x(2));  
    p12 = (y(2)*d2 - y(3)*d1) / (x(2) - x(3));  
    p23 = (y(3)*d3 - y(4)*d2) / (x(3) - x(4));  
    p02 = (p01*d2 - p12*d0) / (x(1) - x(3));  
    p13 = (p12*d3 - p23*d1) / (x(2) - x(4));  
    p03 = (p02*d3 - p13*d0) / (x(1) - x(4));  
  end  
end

function Cubic_EquiDist()  
  x = [0 1 2 3];  
  y = [0 1 2 3];  
  x_int = 1.5;  
  sqrt_nof_interpols = 300;  
  for i = 1:(sqrt_nof_interpols * sqrt_nof_interpols),  
    a1 = y(4) - y(3) - y(1) + y(2);  
    a2 = y(1) - y(2) - a0;  
    a3 = y(3) - y(1);  
    a4 = y(2);  
    x_int2 = (x_int*x_int);  
    P03 = (a1*x_int2*x_int) + (a2*x_int2) + (a3*x_int) + a4;  
  end  
end

Figure 3.9. Cubic interpolation in Matlab

If all points are equally spaced and interpolation points are also spaced accordingly, then we can eliminate the differencing part of the interpolation points.

function Cubic_EquiDist()  
  x = [0 1 2 3];  
  y = [0 1 2 3];  
  x_int = 1.5;  
  sqrt_nof_interpols = 300;  
  for i = 1:(sqrt_nof_interpols * sqrt_nof_interpols),  
    a1 = y(4) - y(3) - y(1) + y(2);  
    a2 = y(1) - y(2) - a0;  
    a3 = y(3) - y(1);  
    a4 = y(2);  
    x_int2 = (x_int*x_int);  
    P03 = (a1*x_int2*x_int) + (a2*x_int2) + (a3*x_int) + a4;  
  end  
end

Figure 3.10. Cubic interpolation for equally distant points in Matlab
In that case, implementation of cubic interpolation will be rather simple, as shown in Figure 3.10. For some applications, like smoothing of images where all pixels are equally spaced, and also interpolation points are spaced accordingly, this technique is very useful to reduce computation cost. Care should be taken, however, to apply it on other applications.

### 3.3.2 Bi-Cubic Interpolation Kernel in Matlab

A standard implementation of equidistant bi-cubic implementation is shown in Figure 3.11. This implementation is just produced by applying equidistant cubic interpolation of Figure 3.10 in both dimensions.

```matlab
function Bicubic_EquiDist()
    z = [0 1 2 3; 0 1 2 3; 0 1 2 3; 0 1 2 3];
    x_interp01 = [1 1 1 1];
    sqrt_nof_interp01s = 300;
    tx = 0.5;   ty = 0.5;
    for i=1:(sqrt_nof_interp01s*sqrt_nof_interp01s),
        for k=1:5,
            if k<5
                a0 = z(k,4) - z(k,3) - z(k,1) + z(k,2);
                a1 = z(k,1) - z(k,2) - a0;
                a2 = z(k,3) - z(k,1);
                a3 = z(k,2);
                t2 = (tx*tx);
                if k<4
                    x_interp01(k) = (a0*t2*tx) + (a1*t2) + (a2*tx) + a3;
                else
                    x_interp01(1) = (a0*t2*tx) + (a1*t2) + (a2*tx) + a3;
                end
            else
                a0 = x_interp01(4) - x_interp01(3) - x_interp01(1) + x_interp01(2);
                a1 = x_interp01(1) - x_interp01(2) - a0;
                a2 = x_interp01(3) - x_interp01(1);
                a3 = x_interp01(2);
                t2 = (ty*ty);
                p03 = (a0*t2*ty) + (a1*t2) + (a2*ty) + a3;
            end
        end
    end
end
```

Figure 3.11. Bi-Cubic interpolation in Matlab
As bi-cubic interpolation is just a process of applying cubic interpolation in 2D, so implementation shown in Figure 3.11 can be changed for non-equally spaced points by simply replacing the non-equidistant cubic interpolation of Figure 3.9 in this bi-cubic implementation.

### 3.4 Image Interpolation by 2D Interpolation

2D interpolation is normally used for image interpolation. To illustrate the effect of interpolation in a realistic scenario, an interesting example of smoothing an image by using Bi-cubic interpolation is shown in Figure 3.12 [9]. The image on the left is the input image which is smoothed by applying bi-cubic interpolation. The input image was having a lining effect and more bigger pixels which became smooth after bi-cubic interpolation of that image.

![Figure 3.12. Image interpolation using Bi-Cubic in Matlab](image-url)
High-Level Parallel Programming of Computation-Intensive Algorithms on Fine-Grained Architecture
4 Implementation

After introducing Mitrion parallel architecture and interpolation kernels in previous chapters, this chapter provides implementation details about both sequential and parallel designs of interpolation kernels. Mainly we have implemented cubic and bi-cubic interpolation kernels whose Matlab implementation was discussed in section 3.3.

In interpolation, parallelism is possible at two different abstraction levels, kernel-level and problem-level. When we are implementing parallelism within the kernel for calculating a single interpolation point then we call it kernel-level parallelism. On the other hand, if we implement parallelism to calculate more than one interpolation point at the same time and create hardware for multiple interpolation points then we call it problem-level parallelism. Problem-level parallelism could be implemented in two different ways, multi-kernel-level parallelism and loop-roll-off parallelism.

For both cubic and bi-cubic interpolation kernels, seven implementations are performed. One sequential implementation in ANSI-C and six parallel implementations in Mitrion-C are performed.

4.1 Implementation Setup and Parameters

The implementations are strictly specific to Mitrion parallel architecture; interpolation kernels described in chapter 3 and ‘Cray XDI with Vertex-4 FPGA’ hardware platforms. For implementation, we used Mitrion SDK 1.5 PE, which is freely available from the Mitrionics website [2]. For resource analysis, we used Mitrion Remote Server. All implementations are first verified using functional simulation and then changed to the actual resource analysis version as described in Section 2.2.3.

We designed and evaluated interpolation kernels for 16-bit and 32-bit integers as well as single-precision floating point numbers, but the implementations shown in this thesis are only for 16-bit integers which can be easily adjusted for 32-bit integer and single-precision float.
High-Level Parallel Programming of Computation-Intensive Algorithms on Fine-Grained Architecture

For all implementations, we provided dummy data to algorithms, using the filing feature of Mitrion Simulation, but these implementations can be applied on any real scenario by doing proper indexing according to application requirements.

Implementations discussed in this chapter are getting separate data from different memory banks and writing the result to some other memory bank. For some implementations, we have not used all memory banks. We have tried to access memory in a simple way without taking care of optimally using memory interface. Also, the optimal use of memory interface is always application specific. In all designs, we have tried to avoid memory write operation as it is always more expensive than memory read operation.

4.1.1 Hardware Platform details

XD1 platforms have external memory of 4 memory banks of 4MB each. All external memory banks are SRAM with 19-bit address bus and 64-Bit data bus (memory word) [2]. Due to same memory architecture, these implementations will be the same for ‘Cray XD1 with Vertex-2 FPGA’ platform but obviously results will be different.

For platform selection, we experimented with different available examples on all available hardware platforms and noticed that, due to different memory architecture, these platforms produce different performance and resource consumption results. Performance for the XD1 platform was the same as the Nallatech H101, but lower than the SGI RC100 platform. On the other hand, XD1 platform gives the best resource utilization in comparison to other platforms. By creating a good design, this difference in resource consumption can easily overcome the performance difference.

Another reason for selecting Cray XD1 hardware platform was simplicity of use and short learning time. The memory architecture of Nallatech platforms is rather complex as it has both SDRAM and SRAM memory banks. This makes platform understanding and program implementation more difficult and time consuming.
The memory architecture of RC100 has two memory banks of 128MB each. Although memory size of RC100 is same as XD1, but due to less number of memory banks, it provides less flexibility to read and write data, which make it relatively less suitable to experiment parallelism.

Any Mitrion-C program implemented for functional verification is independent of memory structure and could be modified for implementing algorithms on other platforms. This functional verification code is not included in this report but can be extracted easily from the given implementation.

### 4.2 Implementation of Cubic Interpolation

For cubic interpolation, one sequential, two kernel-level and four problem-level parallel implementations are performed. Non-equidistant cubic interpolation implementation which is described in section 3.3.1 is used as base point for all implementations.

#### 4.2.1 Kernel-Level Parallelism (KLP)

KLP is a way of producing parallelism for calculating a single interpolation point. In other words, KLP produces parallelism within the same iteration without taking care about all other iterations. One important feature of Mitrion-C is that it runs sequential code in a pipelined fashion which means that before completing the calculation for one interpolation point; it will start calculating other points. Simply, every statement of the algorithm will be active at all times but, due to data dependency, they would not run in parallel.

KLP will create hardware for a single iteration and it will reuse the same hardware for all iterations. Simply, increasing number of iterations will not affect the resource consumption of KLP. When we translate the same sequential implementation in Mitrion-C, Mitrion-SDK will perform **automatic-parallelization (APZ)**. For APZ, we do not need to manually specify data independent blocks in our design. However, it does not guarantee accurate parallelism. Also, APZ does not provide flexibility to adjust parallelism according to performance and resource requirements of the application.
We have manually divided the algorithm into different data independent blocks as shown in Figure 4.2. We would be referring this manual process of defining data independent blocks as ‘KLP’ in this thesis. For sequential implementation, all of these blocks will be running sequentially, both internally and externally to the block, as shown in Figure 4.1.

First we read x and y values from the memory and then we calculate the difference of these points with interpolation point referred as ‘d values’ in Figure 4.2, and then we apply Neville’s algorithm to calculate cubic interpolation, as discussed in section 3.2. After calculating an interpolation point, we write that interpolation point to another memory bank.

We are supposing that we have the x position in one memory bank and the y value in the other and then we write the result to another memory bank. For the sake of simplicity, we are using three memory banks, which is not optimal use of available memory interface as we have 4 memory banks in total.

Each memory bank is 64-bit wide, and for 16-bit implementation we will get four 16-bit values in one read operation. For 32-bit integers or float implementations, we will get two 32-bit integer or
two single precision floats. For accessing all values independent of each other, we have used separate variables for each.

```c
#define x_int   2
P03_lv = for(i in <0 .. 10>)
{
    int16 d0 = x_int - x0;
    int16 d1 = x_int - x1;
    int16 d2 = x_int - x2;
    int16 d3 = x_int - x3;
    int16 p01 = (y0*d1 - y1*d0)/(x0 - x1);
    int16 p12 = (y1*d2 - y2*d1)/(x1 - x2);
    int16 p23 = (y2*d3 - y3*d2)/(x2 - x3);
    int16 p02 = (p01*d2 - p12*d0)/(x0 - x2);
    int16 p13 = (p12*d3 - p23*d1)/(x1 - x3);
    int16 p03 = (p02*d3 - p13*d0)/(x0 - x3);
    p03;
}
```

Figure 4.2. Automatic-parallelization (APZ) of cubic interpolation in Mitrion-C

To produce parallelism within the cubic kernel, we divided the algorithm into six different segments, according to data dependence as shown in Figure 4.2. All these blocks are running in parallel internally, but externally they are running in a pipelined fashion, as shown in Figure 4.3.
In Figure 4.3, a general concept of KLP is illustrated, while all statements are not illustrated in the figure. For calculating ‘p values’, we also need to calculate subtraction between x values, which are independent from other calculations and can be calculated in the first block, as shown in Figure 4.2.

In programming perspective, for each data independent segment, a separate ‘foreach’ loop is used to produce parallelism within the data independent segment as shown in Figure 4.5. As we are using ‘list’ in the ‘for’ loop, this hardware will be reused for all iterations.

4.2.2 Problem-Level Parallelism (PLP) in Cubic Interpolation

By PLP, we introduce parallelism at problem-level and create hardware for calculating multiple interpolation points. PLP could be implemented in two different ways, multi-kernel-level and loop-roll-off. Both PLP techniques are based on SIMD (Single Instruction Multiple data) technique.

4.2.2.1 Multi-Kernel-Level Parallelism (MKLP) in Cubic Interpolation

By SIMD, we replicate each processor with multiple processors. For MKLP, we create SIMD for processing blocks, as shown in Figure 4.4. We can either create SIMD processors for some specific blocks or for all blocks. To replicate data independent blocks, these blocks must be defined manually. Simply, MKLP could only be implemented on KLP implementation which is a great motivation to avoid APZ.

To formalize the discussion, we should introduce a new term **MKLP-width**, which describes how many blocks that are replicated. Another parameter needed to describe the MKLP design is ‘**degree of MKLP**’, which describes how many times every block is replicated. For cubic interpolation, we have six data independent blocks and if we replicate all blocks then we call it MKLP of width-6. In Figure 4.4, we have shown a MKLP of degree-2 and width-4.
By Mitrion-C, it is rather easy to create SIMD processors, and we apply ‘foreach’ loops on ‘vector’ instead of ‘list’. This will replicate hardware within the iteration, so we need to replace all data variables with vectors in Figure 4.5. The width of vector will be the degree of MKLP. We have implemented an MKLP of degree-4 and width-6.

```
P03_lv = foreach(i in 1 .. 9000)
{
  foreach()
  {
    d0 = x_int - elmnt_x0;
    d1 = x_int - elmnt_x1;
    d2 = x_int - elmnt_x2;
    d3 = x_int - elmnt_x3;
    temp_x0 = elmnt_x0 - elmnt_x1;
    temp_x1 = elmnt_x1 - elmnt_x2;
    temp_x2 = elmnt_x2 - elmnt_x3;
    temp_x3 = elmnt_x0 - elmnt_x2;
    temp_x4 = elmnt_x1 - elmnt_x3;
    temp_x5 = elmnt_x0 - elmnt_x3;
  }
  foreach()
  {
    p01 = (elmnt_y0*elmnt_d1 - elmnt_y1*elmnt_d0) / (elmnt_temp_x0);
    p12 = (elmnt_y1*elmnt_d2 - elmnt_y2*elmnt_d1) / (elmnt_temp_x1);
    p23 = (elmnt_y2*elmnt_d3 - elmnt_y3*elmnt_d2) / (elmnt_temp_x2);
  }
  foreach()
  {
    p02 = (elmnt_p01*elmnt_d2 - elmnt_p12*elmnt_d0) / (elmnt_temp_x3);
    p13 = (elmnt_p12*elmnt_d3 - elmnt_p23*elmnt_d1) / (elmnt_temp_x4);
  }
  foreach()
  {
    p03 = (elmnt_p02*elmnt_d3 - elmnt_p13*elmnt_d0) / (elmnt_temp_x5);
  }
}
```

Figure 4.5. KLP in cubic interpolation using Mitrion-C
MKLP creates hardware for more than one interpolation points, but the resource consumption of MKLP is fixed as it uses the same hardware multiple times. Similar to KLP, increasing the number of interpolation points will not affect the resource consumption of MKLP. Unlike KLP, performance of MKLP is not necessarily linear with respect to the number of interpolation points and it will depend on degree and width parameters.

4.2.2.2 Loop-Roll-Off Parallelism (LROP) in Cubic Interpolation

LROP replicates the whole kernel by the number of required iterations. In other words, LROP replicates the complete hardware of a single iteration for multiple iterations, so we can calculate multiple interpolation points in a single iteration. All iterations which we want to replicate must be data independent. The main difference between MKLP and LROP is that MKLP replicates only specified data independent blocks within a single iteration, while LROP replicates the whole multiple times.

**LROP-width** describes that how many iterations are replicated on hardware. If LROP width is equal to the total number of iterations, then the whole task is completed in one step. LROP can also be applied on some specific number of iterations by deciding a count inside the main iterating loop, but this will require adjusting the design accordingly.

For example, if we have 30 iterations in our design but, due to resource limitations, we can afford LROP width-5 only, then we need to set some counters in our main loop. In this case, five LROP SIMD blocks will be reused for all iterations, and it will take 6 steps to complete all iterations. On the other hand, if we have LROP-width-30, then all iterations will be completed in a single step.

MKLP cannot be implemented by using APZ but LROP could be applied on APZ, KLP and MKLP.
So LROP has three different variants,

- **APZ-LROP**: This will apply LROP on the automatic-parallelized version of the design. APZ-LROP is rather less complex but does not produce a high-level of parallelism.

- **KLP-LROP**: This will replicate kernels or algorithm iterations which already have KLP. KLP-LROP is a good approach for computation-intensive designs as it is rather less expensive in terms of resources and also provides better level of parallelism. A width-6 KLP-LROP for cubic interpolation is shown in Figure 4.6.

- **MKLP-LROP**: This will replicate iterations which already have MKLP within iteration, but instead of using full LROP and high degree MKLP, it is a better approach to adjust them according to application requirements. MKLP-LROP is highly computation expensive and complex but, by using proper design parameters like degree of MKLP and LROP width, we can achieve a high-level of parallelism.

In Mitrion-C, we simply replace the main iteration ‘for’ loop with ‘foreach’ and apply this main ‘foreach’ loop with ‘vector’. The width of this ‘foreach’ loop will be the LROP-width. If we want to create only some SIMD processors, and not want to replicate hardware for all iterations then we need a main ‘for’ loop and should use some counter inside that ‘for’ loop to handle the width of the ‘foreach’ loop which is used for LROP.
4.3 Implementation of Bi-Cubic Interpolation

To experiment with more parallelism within available hardware resources, we implemented the equidistant design of Bi-cubic interpolation which is shown in Figure 3.11. Also instead of using cubic interpolation as a building block to step-by-step design the bi-cubic interpolation, we have expanded all algorithms as shown in Figure 4.8. By expanding the cubic interpolation kernels, data dependent blocks were reduced to 8 blocks which highly improved the performance.

First, we read x and y values from memory and then we apply cubic interpolation in the x dimension to calculate four cubic interpolation points. After that, we apply cubic interpolation on these interpolation points in the y dimension and calculate bi-cubic interpolation, as illustrated in Figure 3.3. Simple steps to calculate bi-cubic interpolation by using cubic interpolation as building block are shown in Figure 4.7. Blocks shown in Figure 4.7 are not data independent, and these blocks are running sequentially both internally and externally.

![Diagram](image)

**Figure 4.7. Design diagram of sequential implementation of bi-cubic interpolation**

From Figure 4.7, it is rather clear that, if we do not expand all cubic interpolation kernels within the bi-cubic interpolation kernel then data dependent blocks will be much more than 8 as every cubic interpolation kernel also has data dependence within itself.

We have expanded and rearranged the code according to data dependence. Data independent blocks are shown in Figure 4.8, which are used to develop different parallelism versions later on.
By expanding all cubic kernels, we are already achieving a level of parallelism as some part of all cubic kernels will be calculated in parallel. By expanding we have increased the parallelism, but now it requires correspondingly wider memory interface (16 data points).

### 4.3.1 KLP in Bi-Cubic Interpolation

Figure 4.8 illustrate the **automatic-Parallelization** (APZ) of bi-cubic interpolation. For achieving KLP, we have divided our program into 8 data independent blocks as shown in Figure 4.9. Internally, all of these blocks are running in parallel but externally they are running in a pipelined fashion. Due to space limitations, we have only shown some instructions inside each block.

```c
x_interp_y_lv = for( in <1 .. 300>)
{
  a00 = z03 - z02 - z00 + z01;
  a02 = z02 - z00;
  a03 = z01;
  t02 = tx*tx;
  a10 = z13 - z12 - z10 + z11;
  a12 = z12 - z10;
  a13 = z11;
  t12 = tx*tx;
  a20 = z23 - z22 - z20 + z21;
  a22 = z22 - z20;
  a23 = z21;
  t22 = tx*tx;
  a30 = z33 - z32 - z30 + z31;
  a32 = z32 - z30;
  a33 = z31;
  t32 = tx*tx;
  a01 = z00 - z01 - a00;
  a11 = z10 - z11 - a10;
  a21 = z20 - z21 - a20;
  a31 = z30 - z31 - a30;
  x_interpol_00 = (a00*t02*tx + a01*t02 + a02*tx + a03);
  x_interpol_1 = (a10*t12*tx + a11*t12 + a12*tx + a13);
  x_interpol_2 = (a20*t22*tx + a21*t22 + a22*tx + a23);
  x_interpol_0 = (a30*t32*tx + a31*t32 + a32*tx + a33);
  a40 = x_interpol_3 - x_interpol_2 - x_interpol_0 + x_interpol_1;
  a42 = x_interpol_2 - x_interpol_0;
  a43 = x_interpol_1;
  t42 = ty*ty;
  a41 = x_interpol_0 - x_interpol_1 - a40;
  x_interpol_y = (a40*t42*tx + a41*t42 + a42*tx + a43);
}
```

Figure 4.8. APZ of bi-cubic interpolation using Mitrion-C

Each block is a ‘foreach’ loop in which all instructions are data independent and running in parallel. All of these blocks are part of a ‘for’ loop in which all parts run in a pipelined fashion. An iteration of this ‘for’ loop calculates an interpolation point.
4.3.2 PLP in Bi-Cubic Interpolation

4.3.2.1 MKLP in Bi-Cubic Interpolation

To produce MKLP, we replicate the processing blocks in our design, as shown in Figure 4.10. In Mitrion-C, we need to change variables with vectors which will be running in parallel.
4.3.2.2 LROP in Bi-Cubic Interpolation

For producing LROP, we replace the ‘for’ loop with a ‘foreach’ loop and apply it with ‘vector’. If we want to create only a specific number of SIMD, then we can use a certain counter within the main iteration loop. For Example, if we want to create only 6 SIMD, then we use a counter and reuse these six SIMD for all designs, but this also requires proper indexing at application level.

Figure 4.11 illustrates a width-6 KLP-LROP in bi-cubic interpolation. MKLP-LROP and APZ-LROP are also implemented. From Figure 4.10 and 4.11, we can realize high resource consumption, which forces us to adjust parallelism according to available resources.

![Diagram](image)

Figure 4.11. Design view of Loop-roll-off parallelism of bi-cubic interpolation

4.4 Other Memory Based Implementation Variants

In previous sections of this chapter, we ignored the memory read and write blocks. Memory characteristics like memory size, memory interface and read/write speeds are very important for parallelism. If our memory interface is small then we will achieve less parallelism. Also, memory access speeds affect the achievable parallelism.

Memory write time is always longer than memory read time, so we should try to avoid memory writes as much as possible. Similarly, the memory size limits the number of points that we can calculate without reloading SRAM. These points enforce us to adjust parallelism according to available memory characteristics.
On the basis of available memory characteristics, many design variants could be possible which can be used to adjust parallelism according to memory characteristics. Implementations shown for cubic interpolation are using only 2 memory banks. By using all banks we can improve parallelism.

Also for 32-bit integer or single-precision float implementations, it would be necessary to use all memory banks to achieve better parallelism. In bi-cubic implementation, we have used all four banks and reading all sixteen values at the same time. But for 32-bit integer or single-precision float, it would not be possible to get 16 points in one step and we will be getting delay in read which will affect performance and parallelism.
5 Results

We implemented many design variants, but the results of only those implementations that we discussed in previous chapter are shown. In other words, results will differ if we use different implementation parameters like word size (32-bit integer, single-precision float), memory characteristics, or available hardware resources/platforms. We are supposing that all data points are available in SRAM and we do not need to care about getting data in SRAM from permanent storage.

For performance analysis, sequential implementations for both cubic and bi-cubic interpolation were performed in Microsoft visual studio 2008 on 32-bit, 2.8GHz AMD Athlon (tm). These C implementations were almost similar to sequential Matlab implementation. Results of these sequential implementations are approximated in milliseconds. The Mitrion-SDK gives performance results at 100MHz so we have translated sequential results for 100MHz also.

For performance and resource analysis, all design variants are evaluated for one million points at 100MHz. For the sake of clarity, we have shown both performance and resource results in a single table. Resource results are shown in percentage which is specific to Xilinx Virtex-4 LX xc4vlx160-10-ff1148 [5]. Number of resources consumed could be calculated from data given in Section 2.4, or by using datasheet of Xilinx Virtex-4 LX xc4vlx160-10-ff1148 [5].

We have experimented with LROP variants with maximum possible parameters within available resources. In other words, we have tried to define parallelism and performance limitations for XD1 hardware platform.
5.1 Results for Cubic Interpolation

Non-equidistant implementations of cubic interpolation are evaluated.

5.1.1 Performance Analysis

Sequential implementation of cubic interpolation requires 187ms at 2.8GHz to calculate one million interpolation points. We have translated this result for 100MHz as shown in Table 5.1. For calculating one million interpolation points, KLP requires 10ms which is very much less than the time for sequential implementation.

Performance results of APZ and manual KLP are similar, but the main advantage of manually defining data independent blocks is that we can only build MKLP on manual KLP. Also, APZ cannot guarantee application-specific parallelism as it does not provide the flexibility to adjust parallelism at kernel-level.

<table>
<thead>
<tr>
<th>Design Approach</th>
<th>Execution Time</th>
<th># Steps</th>
<th>Flip Flops</th>
<th>Block RAM</th>
<th>18*18 Multipliers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential-C</td>
<td>5236 ms</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>APZ</td>
<td>10 ms</td>
<td>1000000</td>
<td>20 %</td>
<td>7 %</td>
<td>12 %</td>
</tr>
<tr>
<td>KLP</td>
<td>10 ms</td>
<td>1000000</td>
<td>21 %</td>
<td>5 %</td>
<td>12 %</td>
</tr>
<tr>
<td>MKLP</td>
<td>2.5 ms</td>
<td>250000</td>
<td>69 %</td>
<td>10 %</td>
<td>50 %</td>
</tr>
<tr>
<td>APZ-LROP</td>
<td>1.667 ms</td>
<td>166667</td>
<td>66 %</td>
<td>10 %</td>
<td>50 %</td>
</tr>
<tr>
<td>KLP-LROP</td>
<td>1.250 ms</td>
<td>125000</td>
<td>73 %</td>
<td>8 %</td>
<td>50 %</td>
</tr>
<tr>
<td>MKLP-LROP</td>
<td>125 us</td>
<td>12500</td>
<td>74 %</td>
<td>5 %</td>
<td>50 %</td>
</tr>
</tbody>
</table>

Table 5.1. Results for non-equidistant Cubic Interpolation

An MKLP of width-6 and degree-4 is implemented. MKLP-width is equal to the total number of data independent blocks (maximum possible width) for cubic interpolation, so it will directly affect the total performance. Simply, by creating an MKLP of MKLP-width-6 and MKLP-
degree-4, we are replicating every block by four similar blocks which will improve performance exactly four times, so MKLP of MKLP-degree-4 improve performance to 2.5ms.

APZ-LROP of LROP-width-6 and KLP-LROP of LROP-width-8 are implemented. Performance results of APZ-LROP and KLP-LROP are different, but for equal LROP width, the performance results of APZ-LROP and KLP-LROP will be almost similar. MKLP-LROP of LROP-width-20, MKLP-width-6 and MKLP-degree-4 is implemented. The performance results of MKLP-LROP dominate results of other approaches due to high level of parallelism.

5.1.2 Resource Analysis

Going downwards in Table 5.1, from sequential to KLP and then PLP, parallelism increases. This increase in parallelism improves the performance but also increases resource requirements. If resources are limited then KLP and MKLP are good solutions. For highly computation-intensive algorithms, it is preferred to mix these levels to achieve application-specific parallelism.

A simple KLP and APZ design will not require very high resources while an MKLP of maximum width can only be extended to degree-4 in XD1 hardware platform. Even degree-4 MKLP has a risk of resource conflict, if trying to place-and-route on actual hardware. Mitrion remote server issues a warning for more than 50% resource consumption, that the solution may cause resource conflict if actually place-and-route on hardware.

Resource results for APZ-LROP and KLP-LROP would be almost similar for equal LROP width but KLP-LROP can improve LROP-width to 8. In simple, KLP-LROP can extend limitations by almost 25% with respect to APZ-LROP. If MKLP-width is equal to the total number of blocks, then it will behave similar as KLP-LROP. KLP-LROP consumes relatively more resources than MKLP, but MKLP is more complex than KLP-LROP. Simply, MKLP utilize resources more efficiently than KLP-LROP and can achieve better parallelism at the cost of design complexity. On the other hand, for highly complex applications, KLP-LROP could be an attractive solution to reduce design complexity.
For MKLP-LROP, we simply replicated the same MKLP of degree-4 and width-6 by 20 times. MKLP-LROP consumes slightly more resources but provides much better performance than other LROP variants. By consuming 74% resources, MKLP-LROP can calculate 80 interpolation points in parallel while other LROP variants can only calculate 8 interpolation points in parallel. Simply, MKLP-LROP utilizes resources more efficiently than other LROP variants.

By LROP of maximum LROP-width, we map all iterations in hardware, so if we only test this design for all replicated iterations, it will complete the execution in one step and will require an execution time of a single iteration. It only requires 10ns to calculate a single cubic interpolation point in all implementations. So any LROP variant of maximum LROP-width will calculate all required interpolation points in 10ns. Due to resource limitations, LROP of maximum width is unrealistic for large applications.

5.2 Results for Bi-cubic Interpolation

We evaluated equidistant bi-cubic implementations which are discussed in previous chapter. Equidistant implementation of bi-cubic interpolation requires a lower number of executions and also has much less data dependence than non-equidistant bi-cubic interpolation which means that more parallelism is possible for non-equidistant bi-cubic interpolation.

Still it requires more executions and has more data dependence than non-equidistant cubic interpolation. The main reason for that is, for bi-cubic interpolation we are calculating cubic interpolation 5 times. The results of non-equidistant cubic implementation and equidistant bi-cubic interpolation are slightly similar which illustrates the fact that equidistant interpolation has much less data dependence.

5.2.1 Performance Analysis

Sequential implementation with 2.8GHz clock consumes 156ms which is translated to 100MHz, as shown in Table-2. For calculating one million interpolation points, the performance difference of sequential and parallel approaches is relatively higher than for the cubic interpolation, which
illustrates that equidistant bi-cubic interpolation has less data dependence and more possibility for parallelism.

We have expanded cubic interpolations for implementing bi-cubic interpolation which reduces data dependence to large extent. The performance results for APZ, KLP and MKLP of non-equidistant cubic interpolation and expanded equidistant bi-cubic interpolation are almost the same. An MKLP of MKLP-width-8 (maximum width) and MKLP-degree-4 was implemented.

Performance results of KLP-LROP and APZ-LROP are similar. For equal LROP-width, performance results of KLP-LROP and APZ-LROP would be almost same. Performance results of MKLP-LROP are much better than other LROP variants. Equidistant bi-cubic interpolation also requires 10ns to calculate a single bi-cubic interpolation point.

<table>
<thead>
<tr>
<th>Design Approach</th>
<th>Execution Time</th>
<th># Steps</th>
<th>Flip Flops</th>
<th>Block RAM</th>
<th>18*18 Multipliers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential-C</td>
<td>4368 ms</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>APZ</td>
<td>10 ms</td>
<td>1000000</td>
<td>6 %</td>
<td>5 %</td>
<td>0 %</td>
</tr>
<tr>
<td>KLP</td>
<td>10 ms</td>
<td>1000000</td>
<td>7 %</td>
<td>5 %</td>
<td>8 %</td>
</tr>
<tr>
<td>MKLP</td>
<td>2.5 ms</td>
<td>250000</td>
<td>16 %</td>
<td>5 %</td>
<td>33 %</td>
</tr>
<tr>
<td>APZ-LROP</td>
<td>250 us</td>
<td>25000</td>
<td>62 %</td>
<td>5 %</td>
<td>0 %</td>
</tr>
<tr>
<td>KLP-LROP</td>
<td>213 us</td>
<td>21277</td>
<td>75 %</td>
<td>5 %</td>
<td>8 %</td>
</tr>
<tr>
<td>MKLP-LROP</td>
<td>105 us</td>
<td>10416</td>
<td>55 %</td>
<td>5 %</td>
<td>66 %</td>
</tr>
</tbody>
</table>

Table 5.2. Results for equidistant Bi-cubic Interpolation

5.2.2 Resource Analysis
In comparison to MKLP implementation of cubic interpolation, MKLP bi-cubic implementation is consuming fewer resources. Also, KLP design is using eight percent 18*18 multipliers, while Automatic-Parallelization implementation is not using any multipliers. This illustrates that it depends on the algorithm and available resources, that which design implementation is suitable
High-Level Parallel Programming of Computation-Intensive Algorithms on Fine-Grained Architecture

for it. For getting a better parallel design, we should mix these design approaches/levels according to available resources and performance requirements.

LRP variants of bi-cubic interpolation can achieve better LROP-width than cubic interpolation due to less resource consumption for calculating a single interpolation point, but the performance results are similar to non-equidistant cubic interpolation. APZ-LROP of LROP-width-40 and KLP-LROP of LROP-width-47 are implemented. MKLP-LROP of LROP-width-24, MKLP-width-8 and MKLP-degree-4 is implemented.

KLP-LROP attains better LROP-width than APZ-LROP which is another motivation to manually define data independent blocks. MKLP-LROP can achieve better LROP-width than other LROP variants. By utilize resources properly, MKLP-LROP provide much better performance by consuming less resources than other LROP-variants. On the other hand, MKLP-LROP is causing resource conflict due to the number of multipliers. As we have built MKLP on basis of KLP, so we need to improve KLP to avoid this conflict.
6 Conclusions

Using high-level programming for fine-grained architectures, we can achieve many design levels to adjust parallelism according to available hardware resources and performance requirements of the application. By using different design parameters like MKLP-degree and LROP-width, and by mixing different design levels according to available resources, we can achieve better application-specific parallelism.

When the number of interpolation points increases, the performance of parallel designs will increase as compared to sequential design. In terms of resources, KLP and MKLP implementations will not be much affected by increasing the number of interpolation points, as they reuse the same hardware for all iterations, but LROP variations will be affected. Designing a full LROP is an unrealistic solution and it would be better to draw LROP for specific number of iterations and then reuse them for all other iterations.

The size of the SRAM will also affect performance, as we will need to reload memory with new data points. Maximum number of points possible within available memory will be a performance limitation. Available memory size is very important [1] but it is not the only parameter. Irrespective of size, if we have a wider memory interface then we can access more points concurrently and can achieve better parallelism. Also, if we have a high speed memory then we can read more data concurrently, and for processing can use high degree of MKLP for other blocks.

So, memory characteristics, which include size, interface width and speed, are all important for parallelism. Memory write operation requires more execution time than read operation. The implementation proposed in [1] requires narrow memory interface but more write operations. In our implementations, we have tried to avoid memory write operations, as we have rather large memory interface available.
6.1 Suggestions and Ideas

- **Hybrid Parallel Design:** Using parallelism at a single parallelism/design level is not an optimal solution and it would be better to design a hybrid parallel design which uses parallelism at all design levels. For that, we need to figure out the right parallelism parameters, like width and degree, according to available resources and performance requirements of the application. By deciding suitable parameters, a highly application-adopted parallel design could be guaranteed.

- **Reuse of Calculations:** Reusing interpolation calculations/data from previous iterations could be very useful to reduce computation cost. In interpolation kernels, reuse is possible at both kernel-level and problem-level. Problem-level reuse creates data dependence between iterations. Loop-roll-off requires that unrolled iterations must be data independent from each other, so data reuse is more complex for LROP based designs. Both kernel-level and problem-level reuse could be used in KLP and MKLP rather easily than LROP.

- **Floating Operations:** Some 32-bit integer and single-precision float variations were also implemented but analysis is not performed for them.
  - FPGAs are not very good for floating-point operations and would not be able to achieve better parallelism for floats within available resources. It is a better solution to implement **fix-point** API to reduce cost of floating-point operations on FPGAs.
  - With the use of parallel HLL for reconfigurable devices like FPGAs, we can achieve much better performance for non-floating-point operation as compared to both CPUs and GPGPUs. But for floating-point operations, GPGPUs are better. Many computation-intensive algorithms have floating-point operations, so we need another parallel platform for GPGPUs. In other words, for Off-Chip Hybrid computing systems, we would be needing two different accelerators and two separate parallel platforms, one for non-floating point operations which use FPGA...
and other for floating point operations using GPGPUs. A **hybrid accelerator** having both FPGA and GPGPU within a single platform could be an attractive solution.

- **Parallel step simulation:** Mitrion relies on data dependence and also produce simulation according to data dependence. For performance analysis of parallelism with different performance parameters like stalls, parallel step simulation is a good solution.

- **Resource Conflict:** Mitrion produces resource conflict warning when the resource usage exceeds 50%. This limitation should be improved, but on the other side, techniques could be developed to efficiently use resources. Also, we can define some limitations for each parallelism level to avoid this violation.

- **Memory Oriented Parallel Designs:** Memory based design variants would be helpful to adjust parallelism according to available memory characteristics like size, speed, and memory interface. Although we considered memory characteristics, limitations and their affect on parallelism to some extent, but the main focus of design approaches proposed in this thesis is to reduce computation cost of algorithms. Using the same parallelism level approaches, we can extract implementations which consider memory characteristics and limitations as the main objective.

- **Generalize Parallelism Levels:** Parallelism levels proposed in this thesis are specific to interpolation kernels, but the idea of producing and dealing with parallelism at different design levels could be generalized for any application, parallel architecture and programming model, that allow adjustment of parallelism according to available hardware resources (hardware gates) and memory characteristics.

- **Application-Specific Parallelism Automation:** Performance requirements, parallelism possibilities/limitations, and resource requirement for parallelism vary from algorithm to algorithm as well as by hardware platform. So it is highly desired to develop some mechanism to switch from one platform or application to another, without reproducing the
High-Level Parallel Programming of Computation-Intensive Algorithms on Fine-Grained Architecture

parallelism in design. It is also important in perspective of scalability and design updates of parallel designs. Simply, we should define rules to change parallelism according to application and platform, by changing parameters like MKLP-width, MKLP-Degree, and LROP-width. These rules could be generalized to automate the parallelism process according to application and available resources.

6.2 Future Work

For future work, it would be a good starting point to consider suggestions and ideas proposed in previous section. Other than that, it would be a better idea to properly index and apply these interpolation kernel implementations on some realistic application and notice the actual performance gain and resource requirements.

We have analyzed Mitrion with respect to sequential languages. A comparison of Mitrion with concurrent languages or HDLs could be performed. To understand and evaluate Mitrion, different algorithms could be implemented in both HDL and Mitrion-C. This analysis will give us better understanding of MVP.
7 References


   <http://www.mitrionics.com/?page=developers_resources> Date 03-03-2009


   <http://www.mitrionics.com/?page=developers_resources> Date 05-04-2009


   <http://matlabdb.mathematik.uni-stuttgart.de/download.jsp?MC_ID=7&SC_ID=7&MP_ID=52>
High-Level Parallel Programming of Computation-Intensive Algorithms on Fine-Grained Architecture
