MetaFork: A Compilation Framework for Concurrency Models Targeting Hardware Accelerators

Xiaohui Chen
The University of Western Ontario

Supervisor
Marc Moreno Maza
The University of Western Ontario

Graduate Program in Computer Science

A thesis submitted in partial fulfillment of the requirements for the degree in Doctor of Philosophy

© Xiaohui Chen 2017

Follow this and additional works at: http://ir.lib.uwo.ca/etd

Part of the Algebra Commons, Computer and Systems Architecture Commons, Programming Languages and Compilers Commons, and the Software Engineering Commons

Recommended Citation
http://ir.lib.uwo.ca/etd/4429

This Dissertation/Thesis is brought to you for free and open access by Scholarship@Western. It has been accepted for inclusion in Electronic Thesis and Dissertation Repository by an authorized administrator of Scholarship@Western. For more information, please contact tadam@uwo.ca.
Abstract

Parallel programming is gaining ground in various domains due to the tremendous computational power that it brings; however, it also requires a substantial code crafting effort to achieve performance improvement. Unfortunately, in most cases, performance tuning has to be accomplished manually by programmers. We argue that automated tuning is necessary due to the combination of the following factors. First, code optimization is machine-dependent. That is, optimization preferred on one machine may be not suitable for another machine. Second, as the possible optimization search space increases, manually finding an optimized configuration is hard. Therefore, developing new compiler techniques for optimizing applications is of considerable interest.

This thesis aims at generating new techniques that will help programmers develop efficient algorithms and code targeting hardware acceleration technologies, in a more effective manner. Our work is organized around a compilation framework, called MetaFork, for concurrency platforms and its application to automatic parallelization. MetaFork is a high-level programming language extending C/C++, which combines several models of concurrency including fork-join, SIMD and pipelining parallelism. MetaFork is also a compilation framework which aims at facilitating the design and implementation of concurrent programs through four key features which make MetaFork unique and novel:

1. Perform automatic code translation between concurrency platforms targeting multi-core architectures.
2. Provide a high-level language for expressing concurrency as in the fork-join model, the SIMD paradigm and the pipelining parallelism.
3. Generate parallel code from serial code with an emphasis on code depending on machine or program parameters (e.g. cache size, number of processors, number of threads per thread block).
4. Optimize code depending on parameters that are unknown at compile-time.

Keywords: source-to-source compiler, pipelining, comprehensive parametric CUDA kernel generation, concurrency platforms, high-level parallel programming.
Acknowledgments

The work discussed in this dissertation would not have been possible without the constant encouragement and insight of many people.

I must express my deepest appreciation and thanks to my supervisor Professor Marc Moreno Maza, for his enthusiasm and patience during the course of my PhD research. I also would like to thank my friends Ning Xie, Changbo Chen, Yuzhen Xie, Robert Moir and Colin Costello in University of Western Ontario. My appreciation also goes to my IBM colleagues in compiler group, Wang Chen, Abdoul-Kader Keita, Priya Unnikrishnan and Jeeva Paudel, for their discussions on developing compiler techniques.

Many thanks to the members of my supervisory committee Professor John Barron and Professor Michael Bauer for their valuable feedbacks. Also my sincere thanks and appreciation go to the members of my examination committee Professor Robert Mercer, Professor Robert Webber, Professor David Jeffrey and Professor Jeremy Johnson for their comments and inspiration.

Last but not the least, I am also greatly indebted to my family who deserves too many thanks to fit this page.
## Contents

**Abstract**

**Acknowledgments**

**List of Figures**

**List of Tables**

**List of Appendices**

### 1 Introduction

1.1 Dissertation Outline ........................................... 2
1.2 Contributions .................................................. 3
1.3 Thesis Statement .............................................. 3

### 2 Background

2.1 Concurrency Platforms ........................................... 4
  2.1.1 CilkPlus .................................................. 4
  2.1.2 OpenMP ................................................... 5
  2.1.3 GPU ....................................................... 5
2.2 Performance Measurement ........................................ 7
  2.2.1 Occupancy and ILP ......................................... 8
  2.2.2 Work-Span Model ......................................... 9
  2.2.3 Master Theorem .......................................... 9
2.3 Compilation Theory ............................................. 10
  2.3.1 The State of the Art of Compilers .......................... 10
  2.3.2 Source-to-Source Compiler ................................ 11
  2.3.3 Automatic Parallelization ................................. 12

### 3 A Metalanguage for Concurrency Platforms Based on the Fork-Join Model

3.1 Basic Principles and Execution Model .......................... 14
3.2 Core Parallel Constructs ....................................... 14
3.3 Variable Attribute Rules ....................................... 18
3.4 Semantics of the Parallel Constructs in MetaFork ............ 20
3.5 Supported Parallel APIs of Both CilkPlus and OpenMP ...... 21
  3.5.1 OpenMP ................................................... 23
  3.5.2 CilkPlus .................................................. 26
List of Figures

2.1 GPU hardware architecture ........................................... 6
2.2 Heterogeneous programming with GPU and CPU .................... 7

3.1 Example of a MetaFork program with a function spawn ............ 16
3.2 Example of a MetaFork program with a parallel region .......... 17
3.3 Example of a meta_for loop ........................................ 18
3.4 Various variable attributes in a parallel region .................. 19
3.5 Example of shared and private variables with meta_for ........... 19
3.6 Parallel fib code using a function spawn .......................... 20
3.7 Parallel fib code using a block spawn ............................. 20
3.8 OpenMP clauses supported in the current program translations ... 23
3.9 A code snippet of an OpenMP sections example .................. 25
3.10 A code snippet showing how to exclude declarations which come from included header files ........................................... 28
3.11 A code snippet showing how to translate a parallel for loop from CilkPlus to MetaFork .............................................. 28
3.12 A code snippet showing how to insert a barrier from CilkPlus to MetaFork ......................................................... 28
3.13 A code snippet showing how to handle data attribute of variables in the process of outlining ........................................... 32
3.14 A code snippet showing how to translate parallel for loop from MetaFork to CilkPlus .................................................. 32
3.15 A general form of an OpenMP construct ............................. 32
3.16 Translation of the OpenMP sections construct .................... 34
3.17 An array type variable .............................................. 36
3.18 A code snippet translated from the codes in Figure 3.17 ........... 36
3.19 A scalar type variable .............................................. 36
3.20 A code snippet translated from the codes in Figure 3.19 ......... 36
3.21 A code snippet showing the variable m used as an accumulator ............................................................................ 36
3.22 A MetaFork code snippet (right), translated from the left OpenMP codes .............................................................. 37
3.23 A code snippet showing how to avoid adding redundant barriers when translating the codes from OpenMP to MetaFork ............................................................................ 38
3.24 Example of translation from MetaFork to OpenMP .................. 39
3.25 Example of translating a parallel function call from MetaFork to OpenMP ................................................................. 40
3.26 Example of translating a parallel for loop from MetaFork to OpenMP ................................................................. 40
3.27 A code snippet showing how to generate a new OpenMP main function ................................................................. 40
3.28 Parallel mergesort in size $10^8$ ........................................ 43
6.6 A general command-line interface of MetaFork tools ........................................ 85
6.7 Overall work-flow of the MetaFork analysis and transformation chain ................. 86
6.8 Overall work-flow of the CLANG front-end ...................................................... 88
6.9 A code snippet showing how to create a MetaFork user definedPragmaHandler ........ 88
6.10 A code snippet showing how to register a new user definedPragma handler instance to CLANG preprocessor ............................................................. 89
6.11 Convert MetaFork Keywords toPragma directives ............................................ 91
6.12 A code snippet showing how to annotate a parallel for loop with MetaForkPragma directive ................................................................. 91
6.13 A code snippet showing how to annotate a parallel for loop with MetaForkPragma directive ................................................................. 91
6.14 The code snippet after preprocessing the code in Figure 6.13 ............................. 92
6.15 Setting the tok::eod token ........................................................................... 92
6.16 A code snippet showing how to annotate a parallel region with MetaForkPragma directive ................................................................. 92
6.17 A code snippet showing how to annotate a parallel region with MetaForkPragma directive ................................................................. 92
6.18 The code snippet after preprocessing the code of Figure 6.17 ............................. 93
6.19 Consuming the shared clause ........................................................................ 93
6.20 A code snippet showing how to annotate a join construct with MetaForkPragma directive ................................................................. 93
6.21 A code snippet showing how to annotate a join construct with MetaForkPragma directive ................................................................. 93
6.22 The code snippet after preprocessing the code of Figure 6.21 ............................. 94
6.23 A code snippet showing how to annotate device code with MetaForkPragma directive ................................................................. 94
6.24 A code snippet showing how to annotate device code with MetaForkKeywords ................................................................. 95
6.25 The code snippet after preprocessing the code of Figure 6.24 ............................. 95
6.26 Convert CilkPlus Keywords toPragma directives ............................................ 95
6.27 A code snippet showing how to annotate a parallel for loop with CilkPlusPragma directive ................................................................. 95
6.28 A code snippet showing how to annotate a parallel for loop with CilkPlusPragma directive ................................................................. 95
6.29 The code snippet after preprocessing the code Figure 6.28 ................................. 95
6.30 A code snippet showing how to annotate a parallel function call with CilkPlusPragma directive ................................................................. 96
6.31 A code snippet showing how to annotate a parallel function call with CilkPlusPragma directive ................................................................. 96
6.32 The code snippet after preprocessing the code of Figure 6.31 ............................. 96
6.33 A code snippet showing how to annotate a sync construct with CilkPlusPragma directive ................................................................. 96
6.34 A code snippet showing how to annotate a sync construct with CilkPlusPragma directive ................................................................. 96
6.35 The code snippet after preprocessing the code of Figure 6.34 ............................. 96
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>BPAS timings with 1 and 16 workers: original CilkPlus code and translated OpenMP code</td>
<td>50</td>
</tr>
<tr>
<td>3.2</td>
<td>Timings on AMD 48-core: underlined timings refer to original code and non-underlined timings to translated code</td>
<td>50</td>
</tr>
<tr>
<td>4.1</td>
<td>Speedup comparison of reversing a one-dimensional array between PPCG and MetaFork kernel code</td>
<td>65</td>
</tr>
<tr>
<td>4.2</td>
<td>Speedup comparison of 1D Jacobi between PPCG and MetaFork kernel code</td>
<td>65</td>
</tr>
<tr>
<td>4.3</td>
<td>Speedup comparison of 2D Jacobi between PPCG and MetaFork kernel code</td>
<td>66</td>
</tr>
<tr>
<td>4.4</td>
<td>Speedup comparison of LU decomposition between PPCG and MetaFork kernel code</td>
<td>68</td>
</tr>
<tr>
<td>4.5</td>
<td>Speedup comparison of matrix transpose between PPCG and MetaFork kernel code</td>
<td>68</td>
</tr>
<tr>
<td>4.6</td>
<td>Speedup comparison of matrix addition between PPCG and MetaFork kernel code</td>
<td>70</td>
</tr>
<tr>
<td>4.7</td>
<td>Speedup comparison of matrix vector multiplication among PPCG kernel code, MetaFork kernel code and MetaFork kernel code with post-processing</td>
<td>70</td>
</tr>
<tr>
<td>4.8</td>
<td>Speedup comparison of matrix multiplication between PPCG and MetaFork kernel code</td>
<td>72</td>
</tr>
<tr>
<td>4.9</td>
<td>Timings (in sec.) of quantifier elimination for eight examples</td>
<td>73</td>
</tr>
<tr>
<td>6.1</td>
<td>MetaFork constructs and clauses</td>
<td>90</td>
</tr>
<tr>
<td>6.2</td>
<td>CilkPlus constructs and clauses</td>
<td>95</td>
</tr>
<tr>
<td>7.1</td>
<td>Optimization strategies with their codes</td>
<td>116</td>
</tr>
</tbody>
</table>
List of Appendices

Appendix A Code Translation Examples ........................................... 141
Appendix B Examples Generated by PPCG ..................................... 153
Appendix C The Implementation for Generating Comprehensive MetaFork Programs . 158
Chapter 1

Introduction

In the past fifteen years, the pervasive ubiquity of multi-core processors has stimulated a constantly increasing effort in the development of concurrency platforms, such as CilkPlus [22, 98, 78], OpenMP [117, 14] and TBB [77]. While those programming languages are all based on the fork-join concurrency model [23], they largely differ in their way of expressing parallel algorithms and scheduling the corresponding tasks. Therefore, developing software code combining libraries written with several of those languages is a challenge.

Nevertheless there is a real need for facilitating interoperability between concurrency platforms. Consider for instance the field of symbolic computation. The DMPMC library[1] provides sparse polynomial arithmetic and is entirely written in OpenMP, meanwhile the BPAS library[2] provides dense polynomial arithmetic and is entirely written in CilkPlus. Polynomial system solvers require both sparse and dense polynomial arithmetic and thus could take advantage of a combination of the DMPMC and BPAS libraries. However, CilkPlus and OpenMP have different run-time systems. In order to achieve interoperability between them, an automatic source-to-source translation mechanism was desirable, yielding the original objective for this thesis work.

Another motivation for such a software tool is comparative implementation with the objective of narrowing performance bottlenecks. The underlying observation is that the same multi-threaded algorithm, based on the fork-join parallelism model, implemented with two different concurrency platforms, say CilkPlus and OpenMP, could result in very different performance, often very hard to analyze and compare. If one code scales well while the other does not, one may suspect an inefficient implementation of the latter as well as other possible causes such as higher level of parallelism overheads. Translating the inefficient code to the other language can help narrowing the problem. Indeed, if the translated code still does not scale, one can suspect an implementation issue (say the programmer missed to parallelize one critical portion of the algorithm) whereas if the translated code does scale, then one can suspect a parallelism overhead issue in the original code (say the grain-size of a parallel for-loop is too small).

In the past decade, the introduction of low-level heterogeneous programming models, in particular CUDA [5, 116], has brought super-computing to the level of the desktop computer.

---

1 From the TRIP project [www.imcce.fr/trip] developed at the Observatoire de Paris
2 From the Basic Polynomial Algebra Subprograms [www.bpaslib.org] developed at the University of Western Ontario
puter. However, these models bring notable challenges, even to expert programmers. Indeed, fully exploiting the power of hardware accelerators, in particular Graphics Processing Units (GPUs) \[106\], with CUDA-like code often requires significant code optimization effort. While this development can indeed yield high performance \[12\], it is desirable for some programmers to avoid the explicit management of device initialization and data transfer between memory levels. To this end, high-level models for accelerator programming have become an important research direction. With these models, programmers only need to annotate their C/C++ code to indicate which code portion is to be executed on the device and how data maps between host and device.

As a consequence, it is desirable for our proposed source-to-source translation framework not to restrict itself to programming languages based on the fork-join concurrency model, but also to include the low-level heterogeneous programming model of CUDA which mainly relies on the Single Instruction Multiple Data (SIMD) paradigm, yielding the second motivation of this thesis work.

As of today, OpenMP and OpenACC \[6, 144\] are among the most developed accelerator programming models. Both OpenMP and OpenACC are built on a host-centric execution model. The execution of the program starts on the host and may offload target regions to the device for execution. The device may have a separated memory space or may share memory with the host, so that memory coherence is not guaranteed and must be handled by the programmer. In OpenMP and OpenACC, the division of the work between thread blocks within a grid and, between threads within a thread block can be expressed in a loose manner, or even ignored. This implies that code optimization techniques may be applied in order to derive efficient CUDA-like code. Of course, this is a non-obvious task for a variety of reasons. First of all, for portability reasons, the hardware characteristics of the targeted GPU device should not be assumed to be known in a source code written with a high-level models for accelerator programming. Secondly, and partially as a consequence, program parameters (like grid and thread block formats) should not be assumed to be known either in that same source code. Therefore, this source code should depend on parameters which are symbolic entities with unknown values at compile time.

Being able to generate (say from annotated C/C++) and optimize such parametric CUDA-like code is the third objective of this thesis work. As we shall see, achieving this objective leads to manipulate systems of non-linear polynomial constraints, which makes this process complex and challenging.

### 1.1 Dissertation Outline

In this thesis, we propose M\(\text{etaFork}\) which is both a language accommodating several models of concurrency (fork-join, pipelining, SIMD) and a compilation framework (performing automatic translation from C\textsc{ilk}P\textsc{lus} to OpenMP, from OpenMP to C\textsc{ilk}P\textsc{lus}, from annotated C/C++ to CUDA, etc.)

In Chapter 3, we present M\(\text{etaFork}\) as a metalanguage for multithreaded algorithms based on the fork-join parallelism model and targeting multi-core architectures. By its parallel programming constructs, the M\(\text{etaFork}\) language is currently a super-set of C\textsc{ilk}P\textsc{lus} and offers
counterparts for widely used parallel constructs of OpenMP. In Chapter 5, we show how the MetaFork language supports pipelining, allowing interoperability with Cilk-P [95].

The implementation of MetaFork as a compilation framework is presented mainly in Chapter 6. However, the software framework that allows automatic generation of CUDA code from annotated MetaFork programs is discussed in Chapter 4. Finally, Chapter 7 is dedicated to the question of optimizing parametric CUDA kernels and more generally, optimizing programs depending on parameters whose values are unknown at compile-time.

1.2 Contributions

MetaFork is a high-level programming language extending C/C++, which combines several models of concurrency including fork-join, SIMD and pipelining parallelism. MetaFork is also a compilation framework which aims at facilitating the design and implementation of concurrent programs through four key features which make MetaFork unique and novel:

1. Perform automatic code translation between concurrency platforms targeting multi-core architectures.
2. Provide a high-level language for expressing concurrency as in the fork-join model, the SIMD paradigm and the pipelining parallelism.
3. Generate parallel code from serial code with an emphasis on code depending on machine or program parameters (e.g. cache size, number of processors, number of threads per thread block).
4. Optimize code depending on parameters that are unknown at compile-time.

As of today, the publicly available and latest release of MetaFork, see www.metafork.org, offers the first three features stated above. The latter is implemented as a proof-of-concept and will be integrated in the public version in a near future.

1.3 Thesis Statement

Chapter 3 is a joint work with Marc Moreno Maza, Sushek Shekar and Priya Unnikrishnan, published as [35].

Chapter 4 is a joint work with Changbo Chen, Abdoul-Kader Keita, Marc Moreno Maza and Ning Xie, published as [29].

Chapters 5 and 6 are essentially my work under the supervision of Marc Moreno Maza and Abdoul-Kader Keita.

Chapter 7 is a joint work with Marc Moreno Maza and Ning Xie.
Chapter 2

Background

This chapter presents the technical background of our topic. Section 2.1 introduces the concurrency platforms that are relevant to better understand our work. Section 2.2 explains the most commonly used technologies and concepts to give insight into the performance of a parallel algorithm. Finally, a brief overview of directions for compiler research is covered in Section 2.3.

2.1 Concurrency Platforms

Nowadays, more and more promising parallel computing platforms are available for high performance computing. To gain speedup, programmers are expected to identify parallelism and choose suitable underlying platforms; otherwise, the algorithm developed might not display speedup, or even run slower than the sequential version. In the following sections, we summarize the basic properties of several concurrency platforms.

2.1.1 CilkPlus

CilkPlus is a language extension to the C/C++ programming language, designed for task-based parallelism in a multithreaded environment. With the keywords introduced by CilkPlus, a serial program can be easily converted into a parallel program which models a fork-join parallel control pattern that splits control flow into multiple parallel flows that re-join together later. Precisely, a CilkPlus program is derived from a serial program annotated with CilkPlus keywords indicating where parallelism is allowed. To boost applications, programmers only concentrate on developing the algorithm with ample parallelism, while leaving the underlying CilkPlus run-time system with the responsibility of efficiently scheduling the computation on the executing processors. CilkPlus uses a work stealing strategy to schedule the computational tasks. This method can be efficiently implemented to map a large quantity of parallel tasks onto physical hardware resources at run-time. Moreover, the authors of [59] show that with its work stealing strategy, CilkPlus programs are guaranteed to be time and space efficient. In particular, the computational tasks are adaptively load-balanced between worker threads in the CilkPlus scheduler [99].

Note that CilkPlus keywords only denote the opportunity for parallel computation, but parallel computation is not mandatory. For example, the cilk_spawn keyword which permits
2.1. CONCURRENcy PLATFORMs

The caller running in parallel with the callee, may not actually result in concurrency execution. To carry out such concurrency execution, there must be at least one thread that is idle and looks for computational tasks to steal from other threads. This implies that when a parallel CilkPlus program runs on a single core machine, the serial semantics of that program is retained.

To ease parallel programming efforts, there are two performance analysis tools, CilkProf [72, 133] and CilkScreen [79], designed for programmers that use CilkPlus. The CilkProf scalability and performance analyzer is used to collect the parallel performance data of a CilkPlus program. For example, CilkProf profiles the work, span and burdened span, which allows the developers to diagnose performance bottlenecks in CilkPlus programs. The CilkScreen race detector provides the programmers with any data races which are extremely hard to debug with serial programming tools.

2.1.2 OpenMP

OpenMP (Open Multi-Processing) is the de-facto standard for programming multithreaded applications, which is primarily designed for shared memory systems before the release of version 4.0. By annotating a serial program with various OpenMP compiler directives, programmers can specify fork-join parallelism. Originally, OpenMP only supports implicit tasks for static parallelism which does not fit well into irregular applications which employ recursive algorithms, pointer chasing or load imbalances computation. To efficiently parallelize applications similarly to CilkPlus, OpenMP 3.0 extends its programming model, referred as the OpenMP tasking model [143], by allowing programmers to dynamically create asynchronous units of work to be scheduled by the run-time. The adoption of this tasking model broadens OpenMP to parallelize task-based applications as flexibly as CilkPlus.

Furthermore, beyond developing code for single address space parallelism, the OpenMP language committee has been working on a set of extensions to support heterogeneous computation model using both CPU and accelerators, and this work leads to the release of version 4.0 of the OpenMP specification in July, 2013. This accelerator model of OpenMP builds on a host-centric execution model. That is, the execution of an OpenMP program starts on the host (i.e. CPU) and the host may offload a piece of code to an attached accelerator device (e.g. GPU) for execution according to the OpenMP directives. The accelerator may have its separated memory space or may share memory space with the host, so that memory coherence [121] is not guaranteed.

The accelerator model in OpenMP is similar to OpenACC which also accelerates computation on an external accelerator by adding OpenACC compiler directives. OpenACC has been developed for several years, so the programming features of OpenACC are rich and stable. OpenMP, however, is just starting in the heterogeneous programming field, but it is catching up.

2.1.3 GPU

In recent years, the use of GPUs as accelerators to offer more opportunities for performance tuning, has increased exponentially [52, 111, 45]. Contrary to the traditional CPU which is designed for serial processing, the GPU provides massive parallelism which is specialized for speeding up intensive computations. Programmers can take advantage of this programming
model to combine the power of CPU and GPU in order to explore massive parallelism. However, this model also introduces notable challenges [108], even for expert programmers, due to the complexity of GPU architectures, including memory hierarchy, thread hierarchy and hardware resource limitations. To exploit the computation capabilities efficiently, programmers are required to commit significant efforts in optimizing their algorithms with respect to specific hardware features [135, 84].

Figure 2.1: GPU hardware architecture

**GPU Hardware Model.** A GPU is made up of an array of streaming multiprocessors (SMs) that perform computations independently of each other, as well as off-chip device memory, as shown in Figure 2.1. As GPU architecture is evolving, the configuration and the number of SMs, and the amount of the device memory vary. Roughly, each SM consists of Scalar Processors (SPs), on-chip shared memory and register files. Register files and shared memory of each SM are statically and evenly allocated among thread blocks as long as they are active, and when context switching happens, the states of those registers and shared memory do not need to be saved and restored. This can be viewed as a key basis in achieving high throughput. A thread block which is composed of threads, is mapped to an SM and the execution of each thread block is completely independent of every other. Within a thread block, the SM splits the threads into warps of 32 threads which run in SIMD manner on SPs. In fact, from the perspective of GPUs, a warp is the basic scheduling and execution unit for an SM. Different warps within a thread block are scheduled asynchronously, but can be synchronized if needed. When one warp is stalled, the SM can quickly switch to other warps if they are ready, thereby effectively tolerating the long latency operations such as memory loads, and maximizing resource utilization.

**GPU Programming Model.** Programming GPUs for general-purpose applications is accomplished by the Compute Unified Device Architecture (CUDA) programming model. CUDA allows programmers to define functions, called kernels, which are executed with the SIMD mode by a large amount of threads on GPUs. Figure 2.2 illustrates the heterogeneous programming model of CUDA through hybrid CPU/GPU computing approaches. That is, the execution of a CUDA program starts on the host (i.e. CPU) and the parallel kernel code is executed on the device (i.e. GPU) either synchronously or asynchronously with respect to the host. After the
2.2. Performance Measurement

Completion of kernel calls, control is returned back to the CPU thread. In order to carry out the computations on the device, the programmer is responsible for organizing the data transfers between the host and device memory, transfers in both directions being handled by the host.

CUDA abstracts execution on the GPU in the form of thousands of concurrent threads which are organized into a two-level thread hierarchy composed of the grid as the top level and thread blocks as the bottom level. The size of the grid and the thread blocks can be configured in multiple dimensions (one, two, or three dimensions) by the special parameters provided at kernel launch time. Note that, the configurations are chosen manually by programmers without the guarantee that they are optimal. CUDA kernels written as a single function, are executed by all threads in the grid; hence, each thread needs to distinguish itself from others and to access unique data portions to operate on. To this end, threads are able to query CUDA built-in variables (e.g. blockId, gridDim and threadIdx) and combine them to compute a unique identifier.

In addition, to achieve high memory bandwidth [150], GPUs offer a region of memory, called shared memory, which is private to every SM and can be used to share data at thread block scope. Compared to the slow off-chip GPU device memory which can be accessed by all the threads, the on-chip shared memory provides a very fast memory access speed which is comparable to the speed of register access. From the perspective of programming, programmers need to explicitly declare variables with the CUDA __shared__ keyword to make variables resident in shared memory.

2.2 Performance Measurement

Capturing analytically the parallelism overheads (e.g. scheduling costs) that a concurrency platform imposes on an executing program is a crucial question which has received little attention in the literature. However, establishing the leading cause of a performance issue often remains a very challenging task in many practical situations. Indeed, low parallelism or high cache-complexity at the algorithm level, ineffective implementation, hardware limitations and scheduling overheads are very different possible causes of poor performance. This section is devoted to discussing research directions to help with this challenge, targeting standard concurrency platforms on multi-core and many-core architectures.
2.2.1 Occupancy and ILP

A key motivation [36] of GPUs for massive parallelism is to handle a large mount of active warps, i.e. groups of threads executing in SIMD fashion, on each SM. An SM maximizes utilization of its SIMD lanes [129] by multiplexing those warps according to their states. That is, when a warp which is running, stalls due to long memory or ALU operations latency, this warp is switched out and marked as pending, while another warp which is eligible will be switched in for execution. When the long latency operations complete, the pending warp that was switched out early becomes eligible for execution. Ideally, if there are sufficiently many warps at each cycle to tolerate latency, full computation resource utilization of the SM is achieved. Hence, utilization of an SM is directly associated with the number of active warps on that SM. The occupancy metric on GPUs is used to formulate this concept. Occupancy for an SM is defined as the ratio of the number of active warps to the maximum number of warps. A CUDA occupancy calculator tool, included in the CUDA toolkit, helps programmers to compute the occupancy for a kernel, depending on the execution configurations and the nature of the kernel algorithms.

In practice, to improve occupancy, programmers usually configure more thread blocks than SMs, as well as more threads than SPs. However, not all those thread blocks are concurrently active on SMs. To be active, the resources, such as register and shared memory usage, requested by a thread block, must be available. Thus, occupancy for each SM is tightly constrained by the limited amount of SM resources. As reported in [84, 88, 94], even a slight increase in resource usage, like per-thread register usage and per-block shared memory usage, could deliver a sharp occupancy degradation.

Empirically, occupancy poses a distinctive impact on performance according to the kernel computation patterns. For memory-intensive applications [51, 112, 145], larger occupancy is better due to the fact that instruction level parallelism (ILP) [149] within each thread is not sufficient to mask long memory access latency, and maximizing occupancy allows hiding the memory access latency by multiplexing the active warps. Nevertheless, optimizing for occupancy does not always equate to gain better performance. As revealed in [114, 149], in the context of computation-intensive applications, better performance can also be achieved at a relatively lower occupancy by exploiting more independent work per thread, i.e. ILP. Keep in mind that GPU threads do not stall on memory access and stall only when any operand is not ready. So under this circumstance, the memory access latency can be hidden at the overlapping execution of independent instructions. This idea is widely applied by code optimization. For example, loop unrolling [63, 114] creates a larger number of available independent instructions inside the loop body at the expense of increasing register pressure which may lead to the degradation of occupancy; however, the compiler has more opportunities in scheduling those instructions to improve ILP which maybe sufficient to offset the loss of thread-level parallelism. Hence, the balance between the number of active warps and the hardware resources available for a thread (or a thread block), is directly related to performance, and by choosing optimal kernel execution configurations as well as re-designing the kernel functions, the inter-convertibility between ILP and occupancy can be leveraged for performance tuning.
2.2.2 Work-Span Model

In the work-span model, a program’s execution can be seen as a directed acyclic graph (DAG) composed of nodes and edges. A DAG node represents a sequence of instructions without any parallel control instructions, and edges represent dependencies between nodes. The execution of a DAG follows the prescribed order, where a node of the DAG is eligible to run only if all its predecessor nodes have completed. In order to measure the parallelism using the work-span model, two fundamental metrics, work and span, are derived from DAGs. The work, denoted \( T_1 \), is the total time needed to execute the DAG serially. The span, denoted \( T_\infty \), is also called the critical path length and is the execution time of the DAG on an ideal machine with an infinite number of processors.

In fact, the work-span model [23] has been used to support the development of the CilkPlus programming language and its implementation. Two complexity measures, the work \( T_1 \) and the span \( T_\infty \), and one machine parameter, the number \( P \) of processors, are combined in results like the Graham-Brent theorem [23] or the Blumofe-Leiserson theorem (Theorems 13 & 14) [24] in order to compare algorithm running time estimates. We recall that the Graham-Brent theorem states that the running time \( T_P \) on \( P \) processors satisfies

\[
T_P \leq \frac{T_1}{P} + T_\infty.
\]

When designing a parallel algorithm, it is desirable to systematically integrate the expected scheduling costs that a given concurrency platform imposes on this algorithm. To this end, a refinement of the fork-join model was proposed by He et al. [72] together with an enhanced version of the Graham-Brent theorem, which actually supports the implementation (on multicore architectures) of the parallel performance analyzer called Cilkview. In this context, the running time \( T_P \) is bounded in expectation by \( \frac{T_1}{P} + 2\Delta \hat{T}_\infty \), where \( \Delta \) is a constant (called the span coefficient) and \( \hat{T}_\infty \) is the burdened span. Frigo et al. [59] describe the implementation of the CilkPlus work-stealing scheduler. They introduce the notions of work overhead and span overhead. Spoonhower et al. [139] present a theoretical concept of deviation on which we could rely in order to perform parallelism overhead analysis in the context of schedulers based on other principles than the randomized work-stealing. In particular, and as mentioned by Haque [70], analyzing the burden span may help us discover that an algorithm is not appropriate for an architecture.

2.2.3 Master Theorem

The divide-and-conquer technique, as a classical approach for designing algorithms (either serial or parallel), is widely used [62, 15]. It applies in the case that a task with input size \( n \), can be partitioned into smaller sub-tasks recursively, each with input size \( n/b \), until a base-case problem is reached which is simple enough to be solved directly. Algorithms using this computational pattern can be parallelized efficiently with fork-join platforms, like CilkPlus and the tasking model in OpenMP, see [60].

The cost of divide-and-conquer algorithms illustrated above, can be represented as a recurrence relationship which is given by

\[
T(n) = aT(n/b) + f(n)
\]

where,
1. $n$ is the input size of the original task.

2. $a \geq 1$ and $b > 1$ are constants. $a$ and $n/b$ present the number of sub-tasks and the size of each sub-task, respectively.

3. $f(n)$ is the cost of combining the answers of the sub-tasks.

4. $T(n)$ is the cost of the task with input size $n$.

The complexity of the recurrence relationship in 2.1 can be easily solved by the master theorem [43] in the form of asymptotic notation. More precisely, $T(n)$ is asymptotically bounded in three common cases as follows:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.

2. If $f(n) = \Theta(n^{\log_b a \log k n})$ for some $k \geq 0$, then $T(n) = \Theta(n^{\log_b a \log k+1 n})$.

3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and $af(n/b) \leq cf(n)$ for some $c < 1$ and sufficiently large $n$, then $T(n) = \Theta(f(n))$.

Furthermore, the authors in [9] present a method for solving more generalized divide-and-conquer recurrences which can not be handled by master theorem.

### 2.3 Compilation Theory

Computer hardware is constantly evolving in order to boost the performance. However, by improving compiler techniques, we could also gain many benefits. Generally, compilers serve two purposes: translators which translate programs written in a source programming language into an equivalent program in an object language, and optimizers which perform optimization to make programs more efficient to enable high performance and productivity. In the following sections, we describe a broad range of research into compilers.

#### 2.3.1 The State of the Art of Compilers

We survey two open source compiler frameworks, which are more relevant to industry and academia.

**LLVM/CLANG.** LLVM (Low Level Virtual Machine) [93] compiler infrastructure is a research project started in 2000 at the University of Illinois. LLVM is designed modularly and consists of a front-end, a middle-end optimizer and a target-specific back-end. The front-end translates high-level language programs to a low-level LLVM intermediate representation (IR) (in static single assignment (SSA) form [44]) which is not supposed to actually run on a real machine. The middle-end optimizer which takes the LLVM IR as input, applies on it most of the optimization in terms of passes [156, 92, 38], then outputs an optimized equivalent LLVM IR. The LLVM optimizer provides lifetime long optimizations for a program at any possible stages of optimization [93]. In particular, the LLVM optimizer differs from traditional compilation systems with its own distinguishing features at link-time and run-time [105, 68], which
allow it to perform more sophisticated and aggressive inter-procedural optimizations. Finally, the LLVM back-end transforms the LLVM IR into processor specific binary code, or can be used as just in time compilation.

Today, LLVM employs Clang \(^{[1]}\) as its front-end for the C, C++, Objective-C and Objective-C++ programming languages. Clang aims to deliver fast compiles, useful error and warning messages, and to provide a platform for building great source code level tools, such as source code analysis tools or source-to-source transformation tools. It is designed to offer a complete replacement to the GNU Compiler Collection (GCC)\(^{[1]}\). A major design concept for Clang is its use of a library-based architecture. In this design, various parts of the front-end can be cleanly divided into separate libraries which can then be mixed up for different needs and uses. In addition, the library-based approach encourages good interfaces and reduces the work for new developers to understand and extend the Clang framework.

ROSE. Unlike traditional compilers, ROSE \(^{[2, 125]}\) is designed as a source-to-source compiler infrastructure, developed at Lawrence Livermore National Laboratory for building arbitrary tools for static analysis, optimization and transformation. ROSE has increasing support in its front-end for multiple programming languages and extensions, including C/C++, Fortran, Python, UPC, OpenMP, PHP and Java. Given input programs, the front-end is responsible for constructing an Abstract Syntax Tree (AST) which is a graph representation of the programs and lives in memory for fast operations on it. Contrary to Clang, ROSE provides mechanisms to directly change its AST to facilitate complex code transformation and optimization. Finally, the back-end generates source code from the transformed AST and calls a vendor’s compiler to run the generated source code.

2.3.2 Source-to-Source Compiler

Source-to-source is an area of compilation technology dedicated to the automatic translation of programs written in a high-level language to another such language. The transformation of an un-optimized C program into an optimized C program is an example of source-to-source compilation. The translation of \LaTeX\ document into HTML is another one.

A source-to-source compiler can be used under various situations. One example is with legacy code. Some legacy code is well developed and optimized through many years. Developers can benefit a lot from legacy code if it adapts to modern implementation platforms \(^{[118]}\). Manually rewriting this code is time consuming; what is worse, it may bring potential bugs. Sometimes fully automatic translation is difficult to achieve, which means some extra work by hand is needed; however, one can still take advantage of source-to-source compilation for fragments of code. For example, the authors of \(^{[120]}\) designed PoCC, a source-to-source compiler, for parallelizing code fragments in static control parts (SCoP) format for multi-core architecture using OpenMP. With hiCUDA \(^{[69]}\), a high-level directive-based sequential code can be translated to a CUDA program. In \(^{[138]}\), the authors propose a source-to-source C compiler, called Panda, which can automatically generate hybrid MPI + CUDA + OpenMP code that uses concurrent CPU + GPU computing from annotated sequential stencil C codes.

\(^{1}\)The GNU compiler collection https://gcc.gnu.org/
A second case is debugging multithreaded code [91]. Nowadays programmers have access to many concurrency platforms, but debugging parallel code is often a challenge: if any threads are running concurrently and competing for computer resource acquisition, tracing each thread may not be realistic. The translation of a parallel program into a serial counterpart may help in detecting issues such as data races. Thus, source-to-source compilation can be of the great benefit in such situation.

During the past, source-to-source compilation has been studied by a lot of researchers. For example, several projects offer automatic one-way translation from a concurrency platform running on one hardware architecture to another concurrency platform running on another hardware architecture, e.g. OpenMP shared-memory code to MPI distributed-memory code as in the papers [19] [49] (HOMPI Project). An example of another application of source-to-source compilation appears in an article by Lee et al. [97], where a compiler framework for automatic translation of OpenMP shared-memory programs into CUDA-based GPGPU programs is presented, thus greatly relieving the programming effort, since coding efficient CUDA program can be quite arduous. Other projects offer extensions of a concurrency platform from one hardware architecture to another hardware architecture, like HOMP [104] or OpenMPC [96] which allow extended OpenMP code to run on NVIDIA GPUs.

2.3.3 Automatic Parallelization

The polyhedron model [65] [55] [81] [17] [25] [16] is a powerful geometrical tool for analyzing the relations (w.r.t. data locality or parallelization) between the iterations of nested loop programs. Let us consider the case of parallelization. Once the polyhedron representing the iteration space of a loop nest is calculated, techniques in linear algebra and linear programming, can transform it into another polyhedron encoding the loop steps in a coordinate system based on time and space (processors). From there, a parallel program can be generated. To be practically efficient, one should avoid a too fine-grained parallelization; this is achieved by grouping loop steps into so-called tiles, which are generally trapezoids [75]. These extensions lead, however, to the manipulation of system of non-linear polynomial equations and the use of techniques like quantifier elimination by the authors of [67]. They observe, that classical algorithms for QE are not suitable, since they do not always produce conjunctions of atomic formulas, while this format is required in order to generate code automatically. This issue is addressed by recent algorithm for computing cylindrical algebraic decomposition [33]. Indeed, this algorithm supports QE in a way that the output of a QE problem has the form of a case discussion: this is appropriate for code generation.

Developing automatic parallelization compilers brings notable challenges in order to produce optimized parallel programs. A key reason is that peak performance of parallel programs is often dependent on both the hardware and program characteristics, which generally cannot be estimated based on static code analysis at compile time. Hence, to realize portable performance, the produced parallel programs should be presented in a generic and portable way for adapting performance relevant parameters [90] [113] [64]. With such programs in hand, it is possible to use auto-tuning [42] [132] [63] which provides many benefits across different hardware and problem configurations, to pick the optimal parameters with the best performance.
Chapter 3

A Metalanguage for Concurrency Platforms Based on the Fork-Join Model

In this chapter, we present MetaFork as a metalanguage for multithreaded algorithms based on the fork-join parallelism model and targeting multi-core architectures. By its parallel programming constructs, the MetaFork language is currently a super-set of CilkPlus and offers counterparts for the widely used OpenMP parallel constructs detailed in Section 3.5.

However, MetaFork does not make any assumptions about the run-time system, in particular about scheduling strategies (work sharing, work stealing). In fact, MetaFork is not designed to be a target language, but rather as the internal intermediate representation (IR) of a source-to-source compiler framework for multithreaded languages.

Section 3.1 is going to explain the execution model of MetaFork based on the fork-join concurrency model. The syntax and the semantics of MetaFork’s parallel constructs are specified in Sections 3.2 and 3.4. Since MetaFork is a faithful extension of the C/C++ language, this is actually sufficient to completely define MetaFork. Further, data attributes of the parallel constructs in MetaFork are discussed in Section 3.3.

Recall that a driving motivation of the MetaFork project is to facilitate automatic translation of programs between concurrency platforms. To date, our experimental framework includes translators between CilkPlus and MetaFork (both ways) and, between OpenMP and MetaFork (both ways). Hence, through MetaFork, we perform program translations between CilkPlus and OpenMP (both ways).

Despite the fact that it does not support all features of OpenMP, the MetaFork language is rich enough to capture the semantics of large bodies of OpenMP code, such as the Barcelona OpenMP Tasks Suite (BOTS) [53] and translate faithfully to CilkPlus most of the BOTS test cases. In the other direction, we could translate the BPAS library to OpenMP.

In Section 3.6, we briefly explain how the translators of the MetaFork compilation framework are implemented. In particular, we specify which OpenMP data-sharing clauses are captured by the MetaFork translators. Simple examples of code translation are provided.

In Section 3.7, we evaluate the benefits of the MetaFork framework through a series of experiments. First, we show that MetaFork can help narrow down performance bottlenecks in multithreaded programs by means of comparative implementation, as discussed above. Secondly, we observe that, if a native CilkPlus (resp. OpenMP) program has little parallelism overhead, then the same holds for its OpenMP (resp. CilkPlus) counterpart translated by
MetaFork. We tested more than 20 examples in total for which experimental results can be found in the technical report [34] and for which code can be found on the website of the MetaFork project. Moreover, the source code of the MetaFork translators can be downloaded from the same website at http://www.metafork.org.

3.1 Basic Principles and Execution Model

We summarize in this section a few principles that guided the design of MetaFork. First of all, MetaFork extends both the C and C++ languages into a multithreaded language based on the fork-join concurrency model. Thus, concurrent execution is obtained by a parent thread creating and launching one or more child threads so that the parent and its children execute a so-called parallel region. An important example of parallel regions is for-loop bodies. MetaFork has the following natural requirement regarding parallel regions: control flow cannot branch into or out of a parallel region.

Similarly to CilkPlus, the parallel constructs of MetaFork grant permission for concurrent execution but do not command it. Hence, a MetaFork program can execute on a single core machine.

As mentioned above, MetaFork does not make any assumptions about the run-time system, in particular about task scheduling. Along the same idea, another design intention is to encourage a programming style limiting thread communication to a minimum so as to

- prevent from data-races while preserving a satisfactory level of expressiveness and,
- minimize parallelism overheads.

To some sense, this principle is similar to one of CUDA’s principles [116] which states that the execution of a given kernel should be independent of the order in which its thread blocks are executed.

To understand the implication of that idea in MetaFork, let us return to our concurrency platforms targeting multi-core architectures: OpenMP offers several clauses which can be used to exchange information between threads (like threadprivate, copyin and copyprivate) while no such mechanism exists in CilkPlus. Of course, this difference follows from the fact that, in CilkPlus, one can only fork a function call while OpenMP allows other code regions to be executed concurrently. MetaFork has both types of parallel constructs. But, for the latter, MetaFork does not offer counterparts to the above OpenMP data attribute clauses.

3.2 Core Parallel Constructs

MetaFork has four parallel constructs: function call spawn, block spawn, parallel for-loop and synchronization barrier. The first two use the keyword meta_fork while the other two use respectively the keywords meta_for and meta_join. We emphasize the fact that meta_fork allows the programmer to spawn a function call (like in CilkPlus) as well as a block (like in OpenMP).

As mentioned, the keyword meta_fork is used to express the fact that a function call or a block is executed by a child thread, concurrently to the execution of the parent thread. If the
3.2. Core Parallel Constructs

program is run by a single processor, the parent thread is suspended during the execution of the child thread; when this latter terminates, the parent thread resumes its execution after the function call (or block) spawn.

If the program is run by multiple processors, the parent thread may continue its execution after the function call (or block) spawn, without being suspended, meanwhile, the child thread executes the function call (or block) spawn. In this latter scenario, the parent thread waits for the completion of the execution of the child thread, as soon as the parent thread reaches a synchronization point.

**Spawning a function call with meta_fork.** Spawning a call to the function \( f \), with the argument sequence \( \text{args} \), is done by

\[
\text{meta_fork } f(\text{args})
\]

The semantics is similar to that of the CilkPlus counterpart

\[
cilk\text{spawn } f(\text{args})
\]

In particular, all the arguments in the sequence \( \text{args} \) are evaluated before spawning the function call \( f(\text{args}) \). However, the execution of \( \text{meta_fork } f(\text{args}) \) differs from that of \( \text{cilk\_spawn } f(\text{args}) \) on one feature. While there is an implicit \( \text{cilk\_sync} \) at the end of the Cilk block [78] surrounding this latter \( \text{cilk\_spawn} \), no such implicit barriers are assumed with \( \text{meta\_fork} \). This feature is motivated by the fact that, in addition to the fork-join parallelism, we plan to extend the MetaFork language to other forms of parallelism such as parallel futures [139, 21].

Figure 3.1 illustrates how \( \text{meta\_fork} \) can be used to define a function spawn. The underlying algorithm in this example is the classical divide and conquer quicksort procedure.

**Spawning a block with meta_fork.** The other usage of the \( \text{meta\_fork} \) construct is for spawning a basic block \( B \), which is done as follows:

\[
\text{meta_fork } \{ \ B \ \}
\]

If \( B \) consists of a single instruction, then the surrounding curly braces can be omitted. We also refer to this construction as a *parallel region*. There is no equivalent in CilkPlus while it is offered by OpenMP. Similarly to a function call spawn, this parallel region is executed by a child thread (once the parent thread reaches the \( \text{meta\_fork} \) construct) meanwhile the parent thread continues its execution after the parallel region. Similarly also to a function call spawn, no implicit barrier is assumed at the end of the surrounding region. Hence synchronization points have to be added explicitly, using \( \text{meta\_join} \); see the examples of Figures 3.2.

A variable \( v \) which is not local to \( B \) may be shared by both the parent and child threads; alternatively, the child thread may be granted a private copy of \( v \). Precise rules about data attributes, for both parallel regions and parallel for-loops, are stated in Section 3.3.

Figure 3.2 below illustrates how \( \text{meta\_fork} \) can be used to define a parallel region. The underlying algorithm is one of the two subroutines in the work-efficient parallel prefix sum due to Guy Blelloch [20].

\[1\]In fact, the parent thread does not participate to the execution of a function call (or block) spawn, but will participate to the execution of the iterations of a parallel for-loop.
#include <algorithm>
#include <iostream>
using namespace std;

void parallel_qsort(int * begin, int * end)
{
    if (begin != end) {
        --end; // Exclude last element (pivot) from partition
        int * middle = std::partition(begin, end,
            std::bind2nd(std::less<int>(), *end));
        using std::swap;
        swap(*end, *middle); // move pivot to middle
        meta_fork parallel_qsort(begin, middle);
        parallel_qsort(++middle, ++end); // Exclude pivot and restore end
        meta_join;
    }
}

int main(int argc, char* argv[])
{
    int n = 10;
    int *a = (int *)malloc(sizeof(int)*n);
    srand( (unsigned)time( NULL ) );
    for (int i = 0; i < n; ++i)
        a[i] = rand();
    parallel_qsort(a, a + n);
    return 0;
}

Figure 3.1: Example of a MetaFork program with a function spawn
long int parallel_scanup (long int x [], long int t [], int i, int j) {
    if (i == j) {
        return x[i];
    }
    else{
        int k = (i + j)/2;
        int right;
        meta_fork {
            t[k] = parallel_scanup(x,t,i,k);
        }
        right = parallel_scanup (x,t, k+1, j);
        meta_join;
        return t[k] + right;
    }
}

Figure 3.2: Example of a MetaFork program with a parallel region

Parallel for-loops with meta_for. Parallel for-loops in MetaFork have the following format

\[
\text{meta}\_\text{for} \ (I, \ C, \ S) \ { \ B } \]

where I is the initialization expression of the loop, C is the condition expression of the loop, S is the stride of the loop and B is the loop body. In addition:

- the initialization expression initializes a variable, called the control variable which can be of type integer or pointer,
- the condition expression compares the control variable with a compatible expression, using one of the relational operators \(<, \leq, >, \geq, \neq\),
- the stride uses one the unary operators \(+, -, +=, -=\) (or a statement of the form \(cv = cv + incr\) where incr evaluates to a compatible expression) in order to increase or decrease the value of the control variable cv,
- if B consists of a single instruction, then the surrounding curly braces can be omitted.

The parent thread will share the work of executing the iterations of the loop with the child threads. An implicit synchronization point is assumed after the loop body. That is, the execution of the parent thread is suspended when it reaches meta_for and resumes when all children threads (executing the loop body iterations) have completed their execution. As one can expect, the iterations of the parallel loop \(\text{meta}\_\text{for} \ (I, \ C, \ S) \ { \ B } \) must execute independently of each other in order to guarantee that this parallel loop is semantically equivalent to its serial version \(\text{for} \ (I, \ C, \ S) \ { \ B } \).

Figure 3.3 displays an example of meta_for loop, where the underlying algorithm is the naive (and cache-inefficient) matrix multiplication procedure.

Synchronization point with meta_join. The construct meta_join indicates a synchronization point (or barrier) for a parent thread and its children tasks. More precisely, a parent thread
void multiply_iter_par(int ii, int jj, int kk, int* A, int* B, int* C)
{
    meta_for(int i = 0; i < ii; ++i)
    for (int k = 0; k < kk; ++k)
        for(int j = 0; j < jj; ++j)
            C[i * jj + j] += A[i * kk + k] + B[k * jj + j];
}

Figure 3.3: Example of a meta_for loop

reaching this point must wait for the completion of its children tasks but not for those of the subsequent descendant tasks. When the parent thread resumes, execution starts at the point immediately after the meta_join construct.

Typically, this construct is used as a communication mechanism to exchange information between different threads. In particular, a meta_join construct ensures that all writes to shared memory performed by the threads preceding this meta_join construct are seen in the same view by the code after it. This natural property of the meta_join construct solves the problem of data inconsistency and contention [54] caused by shared memory programming models.

3.3 Variable Attribute Rules

Variables that are non-local to the block of a parallel region may be either shared by or private to the threads executing the code paths where those variables are defined. After a terminology review, we specify the rules that MetaFork uses in order to decide whether such a non-local variable is shared or private.

Shared and private variables. Consider a parallel region with block Y (or a parallel for-loop with loop body Y). X denotes the immediate outer scope of Y. We say that X is the parent region of Y and that Y is a child region of X. A variable v which is defined in Y is said to be local to Y; otherwise we call v a non-local variable for Y. Let v be a non-local variable for Y. Assume v gives access to a block of storage before reaching Y. We say that v is shared by X and Y if its name gives access to the same block of storage in both X and Y; otherwise we say that v is private to Y. In particular, if Y is a parallel for-loop we say that a local variable w is shared by Y whenever the name of w gives access to the same block of storage in any loop iteration of Y, which means that all the threads that execute this parallel for-loop share the same variable w; otherwise we say that w is private to Y.

Value-type and reference-type variables. In the C/C++ programming language, a value-type variable contains its data directly as opposed to a reference-type variable, which contains a reference to its data. Value-type variables are either of primitive types (char, float, int, double, void) or user-defined types (enum, struct, union). Reference-type variables are pointers, arrays, functions and reference.
3.3. Variable Attribute Rules

static and const type variables. In the C/C++ programming language, a static variable is a variable that has been allocated statically and whose lifetime extends across the entire run of the program. This is in contrast to automatic variables (local variables are generally automatic) whose storage is allocated and deallocated on the call stack and, other variables (such as objects) whose storage is dynamically allocated in heap memory. When a variable is declared with the qualifier const, the value of that variable cannot typically be altered by the program during its execution.

/* This file starts here ... */
#include<stdio.h>
#include<time.h>
#include<stdlib.h>
int a;
long par_region(long n){
int b;
int *c = (int *)malloc(sizeof(int)*10);
int d[10];
const int f=0;
static int g=0;
meta_fork{
int e = b;
subcall(c,d);
}
/* ... and continues here ... */
void subcall(int *a,int *b){
for(int i=0;i<10;i++)
printf("%d %d\n",a[i],b[i]);
}
int main(int argc,char **argv){
long n=10;
par_region(n);
return 0;
}
/* ... and finishes here. */

Figure 3.4: Various variable attributes in a parallel region

Variable attribute rules of meta_fork. A non-local variable v which gives access to a block of storage before reaching Y is shared between the parent X and the child Y whenever v is: (1) a global variable, (2) a file scope variable, (3) a reference-type variable, (4) declared static or const, or (5) qualified shared. In all other cases, the variable v is private to the child. In particular, value-type variables (that are not declared static or const, or qualified shared, and that are not global or file scope variables) are private to the child. In Figure 3.4, the variables a, c, d, f and g are shared, meanwhile the b and e are private.

/* To illustrate variable attributes, three files (a headerfile "a.h" and two source files "a.cpp" and "b.cpp") are used. This file is a.cpp */
#include<stdio.h>
extern int var;
void test(int *array)
{
int basecase = 100;
meta_for(int j = 0; j < 10; j++)
{
static int var1=0;
int i = array[j];
if( i < basecase )
array[j]+=var;
}
}
/* This file is b.cpp*/
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include"a.h"
int var = 100;
int main(int argc,char **argv)
{
int var1=0;
for(int i=0;i<10;i++)
{
int a=(int*)malloc(sizeof(int)*10);
rand((unsigned)time(NULL));
for(int i=0;i<10;i++)
a[i]=rand();
test(a);
return 0;
}
/* This file is a.h*/
void test(int *a);

Figure 3.5: Example of shared and private variables with meta_for
Chapter 3. A Metalanguage for Concurrency Platforms Based on the Fork-Join Model

```c
long fib_parallel(long n)
{
    long x, y;
    if (n < 2)
        return n;
    else{
        x = meta_fork fib_parallel(n-1);
        y = fib_parallel(n-2);
        meta_join;
        return (x+y);
    }
}
```

Figure 3.6: Parallel fib code using a function spawn

```c
long fib_parallel(long n)
{
    long x, y;
    if (n < 2)
        return n;
    else{
        meta_fork shared(x)
        {
            x = fib_parallel(n-1);
        }
        y = fib_parallel(n-2);
        meta_join;
        return (x+y);
    }
}
```

Figure 3.7: Parallel fib code using a block spawn

Variable attribute rules of meta_for. A non-local variable which gives access to a block of storage before reaching $Y$ is shared between parent and child. A variable local to $Y$ is shared by $Y$ whenever it is declared static, otherwise it is private to $Y$. In particular, loop control variables are private to $Y$. In the example of Figure 3.5, the variables array, basecase, var and var1, are shared by all threads while the variables $i$ and $j$ are private.

The shared keyword. Programmers can explicitly qualify a given variable as shared by using the shared keyword in MetaFork. In the example of Figure 3.6, the variable $n$ is private to fib_parallel(n-1). In Figure 3.7, we specify the variable $x$ as shared and the variable $n$ is still private. Notice that the programs in Figures 3.6 and 3.7 are semantically equivalent.

3.4 Semantics of the Parallel Constructs in MetaFork

Parallel programming allows any order of the program’s execution, which increases the complexity of reasoning about their behavior. A few research [26] [100] [82] [131] has been undertaken in the area of modeling the semantics of parallel programming languages so as to avoid non-deterministic. The basic idea is that if a program executed sequentially satisfies a given criterion, parallel execution of that program must meet the same criterion.

The goal of this section is to formally define the semantics of each of the parallel constructs in MetaFork. To do so, we introduce the serial C-elision of a MetaFork program $M$ as a C program $C$ whose semantics define those of $M$. The program $C$ is obtained from the program $M$ by a set of rewriting rules stated through Algorithm 1 to Algorithm 4.

As mentioned before, spawning a function call in MetaFork has the same semantics as spawning a function call in CilkPlus. More precisely: meta_fork $f$ (args) and cilk_spawn $f$ (args) are semantically equivalent.

Next, we specify the semantics of the spawning of a block in MetaFork. To this end, we use Algorithms 2, 3 and 4 which reduce the spawning of a block to that of a function call:

- Algorithm 2 takes care of the case where the spawned block consists of a single instruction of where a variable is assigned to the result of a function call,
3.5 Supported Parallel APIs of Both CilkPlus and OpenMP

- Algorithms 3 and 4 take care of all other cases.

Note that, in the pseudo-code of those algorithms the generate keyword is used to indicate that a sequence of string literals and variables are written to the medium (file, screen, etc.) where the output program is being emitted.

A meta_for loop allows iterations of the loop body to be executed in parallel. By default, each iteration of the loop body is executed by a separate thread. However, using the grainsize compilation directive, one can specify the number of loop iterations executed per thread:

```
#pragma meta grainsize = expression
```

Nevertheless, in order to obtain the serial C-elision of a MetaFork for-loop, we require that the meta_for construct could be replaced by the C-language for - whatever is the grainsize of this MetaFork for loop - without changing the initialization expression, condition expression and stride. (Of course, the loop-body must be replaced with its serial C-elision.)

---

**Algorithm 1: Fork_region** *(P, S, R)*

**Input:** R is a statement of the form:

```
meta_for [shared(Z)] B
```

where Z is a sequence of variables, B is a piece of code, P and S are lists of shared variables and private variables to B, respectively, determined by the rules in Section 3.3

**Output:** The serial C-elision of the above MetaFork statement.

1. if B consists of a single statement which is a function call without left-value then
2. \hspace{1cm} generate(B);
3. else if B consists of a single statement which is a function call with left-value then
4. \hspace{1cm} Function_with_lvalue(P, S, B);
5. else
6. \hspace{1cm} Block_call(P, S, B);

---

3.5 Supported Parallel APIs of Both CilkPlus and OpenMP

Through MetaFork, we perform program translations between CilkPlus and OpenMP (both ways). Currently, MetaFork is a super-set of CilkPlus, however, MetaFork does not offer counterparts to all the OpenMP constructs. This section provides an overview of the APIs of both CilkPlus and OpenMP in our implementation.

---

2 The loop iterations of a thread are then executed one after another by that thread.
Algorithm 2: Function with lvalue($P, S, B$)

**Input:** $B$ is a statement of the form:

\[ G = F(A); \]

where $G$ is a left-value, $A$ is an argument sequence, $F$ is a function name, $P$ and $S$ are as in Algorithm 1.

**Output:** The serial C-elson of the MetaFork statement below:

\[
\text{meta\_fork [shared(Z)] } G = F(A)
\]

1 /* $L$ is a list consisting of all variables appearing in $G$ */
2 $L \leftarrow [G]$
3 if $L$ is not a sub-set of $S$ then
4 \[ \text{Block\_call}(P, S, B); \]
5 else
6 \[ \text{generate}(B); \]

Algorithm 3: Block\_call($P, S, B$)

**Input:** $P$, $S$ and $B$ are as in Algorithm 1.

**Output:** The serial C function call of the above input.

1 $(E, H, T, P, O) = \text{Outlining\_region}(P, S, B);$  
2 /* declare the outlined function */
3 \[ \text{generate}(E); \]
4 /* declare an object, say $O$, to wrap all the parameters */
5 \[ \text{generate}(T, O); \]
6 /* initialize all the members (i.e. parameters) in the object $O$ */
7 \[ \text{generate}(P); \]
8 /* a call to the outlined function */
9 \[ \text{generate}(H, \text{“ ( ”, O, \text{“ ) ”}); \]
3.5.1 **OpenMP**

**Data-Sharing attribute clauses.** Some OpenMP constructs accept clauses to control the data-sharing attributes of variables referenced in the OpenMP construct. Figure 3.8 describes the clauses supported in the current program translations. Note that for the each OpenMP construct discussed below, we specify the valid clauses on it.

1. *The private clause.* The private clause declares variables in its list to be private to each thread. This clause has the following format:

   ```
   private (list)
   ```

2. *The collapse clause.* The collapse clause is used to specify how many loops are associated with the OpenMP loop construct. This clause has the following format:

   ```
   collapse (expression)
   ```

3. *The shared clause.* The shared clause declares one or more list variables to be shared by OpenMP tasks. This clause has the following format:

   ```
   shared (list)
   ```

4. *The default clause.* The default clause explicitly determines the data-sharing attributes of variables that are referenced in a construct and would otherwise be implicitly determined. This clause has the following format:

   ```
   default (shared | none)
   ```

5. *The firstprivate clause.* The firstprivate clause combines the behavior of the private clause with automatic initialization of the variables in its list. This clause has the following format:

   ```
   firstprivate (list)
   ```

6. *The num_threads clause.* The num_threads clause sets the number of threads to use for the next parallel region. This clause has the following format:

   ```
   num_threads(integer-expression)
   ```

   Figure 3.8: OpenMP clauses supported in the current program translations

**Parallel construct.** This is the fundamental OpenMP parallel construct that starts parallel execution by defining a parallel region. It has the following format:
#pragma omp parallel [clause[, ]clause] ... ] new-line
structured-block

When a thread reaches a parallel directive, it creates a team of threads and becomes the master of the team. The master thread has the thread number 0. The code is duplicated and all threads will execute that code defined in the parallel region. There is an implicit barrier at the end of a parallel region provided by the OpenMP compiler. Only the master thread continues execution past this point. The parallel directive has five optional clauses that take one or more arguments: private, shared, firstprivate, default and num_threads.

Worksharing constructs. The OpenMP defines three worksharing constructs for C/C++, that are the loop, sections and single construct.

1. The loop construct. The loop directive specifies that the iterations of the loop immediately following it will be executed in parallel. The loop construct has the following format:

   #pragma omp for [clause[[,] clause] ... ] new-line
   for-loops

   It has six optional clauses: private, shared, firstprivate, schedule, collapse and nowait.

2. The sections construct. The sections construct contains a set of enclosed section/sections of code that are to be divided among the threads. Independent section constructs are nested within a sections construct. Each section is executed once by a thread. It is possible for a thread to execute more than one section. The format of this construct is as follows:

   #pragma omp sections [clause[[,] clause] ... ] new-line
   {
     [ #pragma omp section new-line]
     structured-block
     [ #pragma omp section new-line
     structured-block ]
     ...
   }

   There are three clauses: private, firstprivate and nowait. There is an implied barrier at the end of a sections construct unless the nowait clause is used. Figure 5.9 gives an example of OpenMP sections.

3. The single construct. The single construct specifies that the enclosed code is to be executed by only one thread in the team. Threads in the team that do not execute the single construct, wait at the end of the enclosed code block unless a nowait clause is specified. The format of this construct is as follows:
3.5. Supported Parallel APIs of Both CilkPlus and OpenMP

```c
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        for (i=0; i < N; i++)
            c[i] = a[i] + b[i];

        #pragma omp section
        for (i=0; i < N; i++)
            d[i] = a[i] * b[i];
    } /* end of sections */
} /* end of parallel section */
```

Figure 3.9: A code snippet of an OpenMP sections example

```c
#pragma omp single [clause[[], clause] ...] new-line
structured-block
```

This directive has three optional clauses: private, nowait and firstprivate.

**Task construct.** This is a new construct with OpenMP version 3.0. The task construct defines an explicit task, which may be executed by the encountering thread, or deferred for execution by any other thread in the team. The format of this construct is as follows:

```c
#pragma omp task [clause[[], clause] ...] new-line
structured-block
```

It has four optional clauses: default, private, firstprivate and shared.

**Master and synchronization constructs.** OpenMP realizes several synchronization means to ensure the consistency of shared data and to coordinate parallel execution among threads. Currently, we support the following three synchronization constructs.

1. The master construct. The master construct specifies a region that is to be executed only by the master thread of the team. All other threads on the team skip this section of code. There is no implied barrier associated with this construct. The format of this construct is as follows:

```c
#pragma omp master new-line
structured-block
```

2. The barrier construct. The barrier construct synchronizes all threads in the team. When a barrier construct is reached, a thread will wait at that point until all other threads have reached that barrier. All threads then resume executing in parallel the code that follows the barrier. All threads in a team (or none) must execute the barrier region. The format of this construct is as follows:
### 3. A Metalanguage for Concurrency Platforms Based on the Fork-Join Model

```c
#pragma omp barrier
```

3. The `taskwait` construct. This construct is new with OpenMP version 3.0. The `taskwait` construct specifies a wait on the completion of child tasks generated since the beginning of the current task. The format of this construct is as follows:

```c
#pragma omp taskwait
```

**Combined constructs.** OpenMP provides a set of combined constructs for specifying one construct immediately nested inside another construct. The semantics of these constructs are identical to an individual `parallel` directive being immediately followed by a separate `worksharing` construct. Currently, we support the following two combined constructs.

1. The `parallel` loop construct. This construct specifies a `parallel` construct containing one or more associated loops. The format of this construct is as follows:

```c
#pragma omp parallel for [clause[,] clause] ...
```

The clause can be any of the clauses accepted by the `parallel` or `for` directives, except the `nowait` clause.

2. The `parallel sections` construct. This construct illustrates a `parallel` construct containing one `sections` construct and no other statements. The format of this construct is as follows:

```c
#pragma omp parallel sections [clause[,] clause] ...
{
    [#pragma omp section]
    structured-block
    [#pragma omp section]
    structured-block
    ...
}
```

The clause can be any of the clauses accepted by the `parallel` or `sections` directives, except the `nowait` clause.

### 3.5.2 CilkPlus

CilkPlus offers the following three keywords which are supported by MetaFork translators to the C/C++ language.
The cilk_for keyword. A cilk_for loop allows loop iterations to run in parallel. The general cilk_for syntax is:

\[
\text{cilk_for (declaration and initialization; conditional expression; increment expression)}
\]

loop-body

The cilk_for statement divides the loop into chunks containing one or more loop iterations. Each chunk is executed serially and is spawned as a chunk during the execution of the loop. At the end of the cilk_for loop, there is an implicit barrier.

The cilk_spawn keyword. The cilk_spawn keyword informs the compiler that the preceded function by cilk_spawn may run asynchronously with the caller. A cilk_spawn statement can take one of the following forms:

\[
\begin{align*}
\text{type var} & = \text{cilk_spawn } \text{func(args);} \\
\text{var} & = \text{cilk_spawn } \text{func(args);} \\
\text{cilk_spawn } \text{func(args);} \\
\end{align*}
\]

The cilk_sync keyword. This specifies that all spawned calls in a function must complete before execution continues. After the spawned calls all complete, the current function can continue. An implicit cilk_sync at the end of a function is executed to synchronize the previous cilk_spawn statement if any, after the caller is assigned or initialized to the return value. The syntax is as follows:

\[
\text{cilk_sync;}
\]

The cilk_sync only syncs with children spawned by this function. Children of other functions are not affected.

3.6 Translation

In this section, we briefly explain how the translators of the MetaFork compilation framework are implemented. Obviously, for each translator, the semantics of each input program are preserved into the output program. However, scheduling strategies (like an OpenMP clause schedule(static, chunksize)) are ignored by our translators. In addition, in the case of the code we want to analyze with include directives, we are not interested in the declarations from included files, especially from system headers. However, by default, our visitors detailed in Section 6.4.2, run for all declarations. Thus, the code in Figure 3.10 is realized to avoid performing our translation algorithms on declarations from included header files.
const auto& FileID = SourceManager.getFileID(Decl->getLocation());

if (FileID != SourceManager.getMainFileID())
    continue;

Figure 3.10: A code snippet showing how to exclude declarations which come from included header files

3.6.1 Translation from CilkPlus to MetaFork

Translating code from CilkPlus to MetaFork is easy in principle since, up to the vectorization constructs of CilkPlus, the MetaFork language is a super-set of CilkPlus. For example, a cilk_for loop is faithfully translated to a meta_for loop, as can be seen in Figure 3.11. However, implicit CilkPlus barriers need to be explicitly inserted in the target MetaFork code, see Figure 3.12. This implies that, during translation, it is necessary to trace the instruction stream DAG of the CilkPlus program in order to properly insert barriers in the generated MetaFork code. To be precise, a meta_join construct is manually inserted before each return statement (explicit or implicit), if there has been any spawn invocation since the last sync.

```c
void function()
{
    cilk_for ( int j = 0; j < ny; j++ )
    {
        int i;
        for ( i = 0; i < nx; i++ )
        {
            u[i][j] = unew[i][j];
        }
    }
}
```

Figure 3.11: A code snippet showing how to translate a parallel for loop from CilkPlus to MetaFork

```c
void test()
{
    int x;
    x = cilk_spawn test1();
}
```

```c
void test()
{
    int x;
    x = meta_fork test1();
    meta_join;
}
```

Figure 3.12: A code snippet showing how to insert a barrier from CilkPlus to MetaFork
3.6.2 Translation from MetaFork to CilkPlus

Since CilkPlus has no constructs for spawning a parallel region (which is not a function call) we use the widely used outlining technique \[102, 103, 80, 132\] to wrap the parallel region as a function, namely outlined function, and then replace that parallel region by a call to the outlined function concurrently. In addition to the definition of the outlined function, we refer to the function which encloses a parallel region in a lexical scope, as a host function. Indeed, the problem of translating code from MetaFork to CilkPlus is equivalent to that of defining the serial elision of a MetaFork program.

We have designed and implemented our outlining algorithm by taking advantage of Clang AST, which is used to analyze the data attribute of variables and patch the code in the parallel region. Algorithm 4 depicts our algorithm in detail.

To clarify, as mentioned in Section 3.4, \(P\) and \(S\) are lists of shared variables and private variables to \(B\), respectively, determined by the rules in Section 3.3. The variables in \(P\) and \(S\) are candidates to be passed as parameters to the outlined function. However, in order to minimize the overheads introduced by outlining, we select the minimal candidates as parameters. For instance, global variables which can be seen by all the functions including the outlined function, are not passed as arguments. In addition, a variable which is defined inside the parallel region, is not worth being a candidate on the parameter list due to the fact that it is not live after the parallel region. Moreover, in order to preserve the original semantics after outlining, a parameter must be passed explicitly either by address or by value, depending on its data attributes as discussed in Section 3.3. With this view, it is natural that:

1. the shared variables are accessed via pointers by the outlined function. This gives the visibility of shared variables to the host function as well, so that the intrinsically shared semantics are kept. On the other hand, it can be profitable to minimize the usage of pointers to simplify the code complexity for the subsequent program analysis [71, 13, 28]. For that purpose, the C++ reference variables, classified as shared variables in MetaFork, are treated in an optimized way, that are, reference variables are passed by reference in C++ conventions. For example, Listing A.5 shows the outlined function, i.e. \texttt{taskFunc4}, corresponding to the parallel region in Figure 3.13, and the concurrent call to the outlined function in the host function at Line (42). The reference variable \texttt{ref} which is used within the parallel region in Figure 3.13 is access by reference at Line (18) in the outlined function. But as a contrast, the shared variable \texttt{m} is accessed via a pointer at Line (20) in the outlined function.

2. the private variables, such as the variable \(j\) at Line (40) in Listing A.5 are passed by value. Thus, private variables are stored on the stack of the outlined function and the private semantics are preserved.

Instead of passing parameters one by one to the outlined function, all of them are wrapped into a struct type (say, \(T\), as defined in Algorithm 4) which will be passed to the outlined function as the only parameter in the form of a pointer. Then \(T\) is re-declared in the outlined function and is broken into a sequence of variable declarations, see the outlined function in Listing A.5. This method solves the following issue we observe. For example, a new date type (say, \(A\)) is declared locally within the host function, and then \(A\) is used within the parallel
Algorithm 4: Outlining_region(P, S, B)

Input: P, S and B are as in Algorithm[1]

Output: (E, H, T, P, O) where E is a function definition of the form

\[
\text{static void * H(void *F) \{ T + U + D \}}
\]

H is a function name, F is an void pointer name, T is a struct data type, O is an object of T, D is a function body, P is a sequence of initialization, U is a sequence of declaration, C is a sequence of formal constructor parameters of T, I is a sequence of actual constructor parameters of T.

1 Initialize each of T, U, D, P, U, C, I to the empty string
2 /* convert parameter F to type T */
3 append (T + "*" + O + "=" + "(" + T + "*" + ")" + F) to U
4 foreach variable v in S do
5     /* reference variables are added to the struct constructor */
6         if v is reference variable then
7             append (v.type + v.name) to T
8             append (v.type + v.name + "=" + T \rightarrow v.name) to U
9             append (v.name + "(" + v.name + ")") to I
10            append (v.type + v.name) to C
11         else
12             append (v.type + "*" + v.name) to T
13             append (v.type + "*" + v.name + "=" + S \rightarrow v.name) to U
14             append (O \rightarrow v.name + "=" + "&" + v.name) to P
15     endforeach variable v in P do
16         append (v.type + v.name) to T
17         append (v.type + v.name + "=" + T \rightarrow v.name) to U
18         append (O \rightarrow v.name + "=" + v.name) to P
19     /* allocate a new object O of type T */
20     if I is empty then
21         append P to (O + "=" + "(" + T + "*" + ")" + "malloc" + "sizeof(" + T + "))")
22     else
23         append P to (O + "=" + "new" + T + "(" + I + ")")
24     /* add a constructor to T */
25         append (T + "(" + C + ")" + ";") + I + "[ ]") to T
26     Translate B verbatim except /* PASS 2 */
27     foreach right-value R within B do
28         if (R is in S) and R is not reference variable then
29             append ("*" + R.name) to D
region. In fact, without special treatment of \( A \), \( A \) is not visible within the scope of the outlined function. We admit that there are several different ways to stress the above issue, but with our method, i.e. by copying \( A \) to the outlined function before declaring \( T \) in the outlined function, we are able to keep correct grammar with minor changes of the source program.

Additionally, the value of each struct member is initialized either by its address, e.g. the variable \( \text{arry} \), or by its value, e.g. the variable \( j \). In particular, reference variables are initialized via the struct constructor, e.g. the variable \( \text{ref} \), as noted in Listing A.5.

Another key aspect of outlining is to patch the code within the parallel region. This allows the variables to access the correct memory location according to the parameter passing format, i.e. by value or by address. We accomplish this in the second pass as shown in Algorithm 4. The code patching involves simply replacing the variables (except reference variables) which are passed by address, by its pointer format. To be precise, the accesses to a variable \( x \) are translated to accesses to \((^*x)\). Lastly, the generated outlined function is placed immediately in front of the host function, either being a global function or a member function within a C++ class.

Another problem is related to handle nested parallelism during outlining. As discussed in Section 6.4.2, the immutable design of CLANG AST creates many challenges, as developers can only perform on the AST textual changes which are not structured. This design does not work well in the case of complex code transformations, such as an implementation which requires a chain of transformations, that is the transformation phase \( t_n \) requires analysis on the output of the transformation phase \( t_{n-1} \).

To deal with this, a multiple parsing approach which greatly simplifies the design of the outlining algorithm, is introduced to let the outlining procedure only work on the innermost parallel region during every single parsing iteration. At the end of each iteration, the back-end of META Fork compiler, detailed in Section 6.4.3, generates an output file serving as the input file for the next iteration in turn, until the out-most parallel region is outlined. The code (translated from BOTS benchmark) in Listing A.2, taken as an example here, illustrates our multiple parsing approach. Listing A.3 shows the results of the first running of the outlining procedure over the code in Listing A.2 and we observe that only the innermost parallel region at Line (38) in Listing A.2 is outlined. Using the code in Listing A.3 as input, we can now perform outlining procedure once more and as seen in the results in Listing A.4, the out-most parallel region at Line (78) in Listing A.3 is replaced by a call to the outlined function (i.e. _taskFunc5) concurrently.

Lastly, a meta for loop is faithfully translated to a cilk for loop, as can be seen in Figure 3.14.

### 3.6.3 Translation from OpenMP to META Fork

We start with providing an abstraction algorithm scheme that models the translated program as a sequence of subroutines that are invoked when relevant OpenMP constructs are encountered. With this scheme, we could formalize the extensive translation methods throughout this section in a more concise way which is easy to follow for readers.

Algorithm 5 visualizes this abstraction algorithm scheme which is sufficiently flexible to translate the OpenMP constructs as shown in Figure 3.15. The observation is that in the case that there is no direct one-to-one mapping between OpenMP and META Fork constructs, we
Chapter 3. A Metalanguage for Concurrency Platforms Based on the Fork-Join Model

```c
void outlining_example(void)
{
    int m, j;
    int &ref = m;
    int arry[10];

    meta_fork shared(m)
    {
        arry;
        ref++; // ref++;
        int local_j = j;
        m;
    }
}
```

Figure 3.13: A code snippet showing how to handle data attribute of variables in the process of outlining

```c
void function()
{
    meta_for ( int j = 0; j < ny; j++ )
    {
        int i;
        for ( i = 0; i < nx; i++ )
        {
            u[i][j] = unew[i][j];
        }
    }
}
```

void function()
{
    cilk_for ( int j = 0; j < ny; j++ )
    {
        int i;
        for ( i = 0; i < nx; i++ )
        {
            u[i][j] = unew[i][j];
        }
    }
}

Figure 3.14: A code snippet showing how to translate parallel for loop from MetaFork to CilkPlus

```c
#pragma omp directive_name [clause[ [, ]clause] ...]
    structured-block
```

Figure 3.15: A general form of an OpenMP construct
### Algorithm 5: Translation scheme(R)

| Input: R which represents an OpenMP construct as defined in Figure 3.15. |
| Output: (E, C, D, B, J) where |
| E is data environment variables placed before C which is a MetaFork construct, D is a set of new generated local declarations within C, B is the statement body of C, and J is a MetaFork synchronization construct or null. |
| P is the set of private variables, S is the set of shared variables and F is the set of firstprivate variables. |

\[
\begin{align*}
1 & \quad (P, S, F) = \text{DataAttributeAnalysis}(R); \\
2 & \quad \text{OuterEnvControl}(R); \quad \text{/* outputs E */} \\
3 & \quad \text{MetaConstruct}(R, S); \quad \text{/* outputs C */} \\
4 & \quad \text{InitLocalDataEnv}(P, F); \quad \text{/* outputs D */} \\
5 & \quad \text{Transform} \text{(structured-block)}; \quad \text{/* outputs B */} \\
6 & \quad \text{PotentialSync}(R); \quad \text{/* outputs J */}
\end{align*}
\]

need to abstract away the differences among several OpenMP constructs so that they could be mapped to a single MetaFork construct. This, in turn, proves the compact design of MetaFork in the context of a programming language, without losing its diversity for expressing parallelism.

To continue our algorithm scheme, in order to preserve correct program behavior during transformations, data-sharing attributes of variables referenced within OpenMP constructs are analyzed and determined according to the rules in Chapter 2.14 of OpenMP application program interface [117]. This is done by subroutine DataAttributeAnalysis which is called initially for each OpenMP construct. Secondly, programmers may use a set of OpenMP clauses (e.g. num_threads) which carry information that controls the behavior of an OpenMP construct before its execution. Those clauses are translated by OuterEnvControl subroutine, see Algorithm 6 and the translated codes are placed immediately before the execution of the parallel region. This situation is illustrated in Figure 3.22. In essence, num_threads is the only clause supported so far in this category. The OpenMP directive then is translated to its counterpart MetaFork construct (including optional clauses) by calling subroutine MetaConstruct, see Algorithm 7. Examples of this subroutine can be found through Listing A.2. Figures 3.22, 3.23 and 3.16.

In connection with the results obtained from the DataAttributeAnalysis function, in the case of that, there are variables which are local to each thread, we invoke InitLocalDataEnv subroutine as shown in Algorithm 8 to declare them as local variables at the beginning of the parallel region. Depending on the type (either private or firstprivate) of each variable, a specific method is dispatched to handle it. To be precise, the variables of private data attribute are simply declared as local variables without initialization; in the case of firstprivate variables, as detailed in Algorithm 8 we translate local declarations and generate the initialization codes to initialize the firstprivate variables with their original value; regarding the shared
void test()
{
    int sum_a=0, sum_b=0;
    int a[5] = {0,1,2,3,4};
    int b[5] = {0,1,2,3,4};
    #pragma omp parallel sections
    {
        #pragma omp section
        {
            for(int i=0; i<5; i++)
                sum_a += a[i];
        }
        #pragma omp section
        {
            for(int i=0; i<5; i++)
                sum_b += b[i];
        }
    } /* an implicit barrier here*/
}

void test()
{
    int sum_a=0, sum_b=0;
    int a[5] = {0,1,2,3,4};
    int b[5] = {0,1,2,3,4};
    meta_fork shared(sum_a)
    {
        for(int i=0; i<5; i++)
            sum_a += a[i];
    }
    meta_fork shared(sum_b)
    {
        for(int i=0; i<5; i++)
            sum_b += b[i];
    }
    meta_join;
}

Figure 3.16: Translation of the OpenMP sections construct

---

**Algorithm 6: OuterEnvControl(R)**

**Input:** R is as in Algorithm 5.

**Output:** (E) where

- E is a statement which sets the data environment variable controlling the behavior of an OpenMP program.

1. Initialize d to a set consisting of all clauses of R
2. if a "num_threads" clause is in d then
3.    extract sub-expression, namely s, from the num_threads clause;
4.    /* a function call which sets the number of threads */
5.       generate("meta_set_nworkers(", s, ")");
Algorithm 7: MetaConstruct(R, S)

Input: R and S are as in Algorithm 5.
Output: C where C is a MetaFork construct (without body).

1. Initialize t to the construct type of R
2. Initialize b to the structured block of R
3. Initialize d to the clauses appearing in R
4. switch t do
   case single
      if a "nowait" clause is in d then
         generate("meta_fork ");
         if S is non-empty then
            generate("shared( S )");
   case loop or parallel loop
      if a "collapse" clause is in d then
         initialize n to the number of the associated loops;
      else
         n = 1;
      for i = 1 to n do
         generate("meta_for") replacing the $i_{th}$ "for" C/C++ keyword appearing in b;
   case task or master or section
      if S is non-empty then
         generate("meta_fork " + "shared( S )");
      else
         generate("meta_fork ");
   case barrier or taskwait
      generate("meta_join");
   case parallel or sections or parallel sections
      do nothing;
variables, they are enclosed in the MetaFork shared clause. For instance, a variable, say v, is categorized into firstprivate data attribute within an OpenMP construct. If v is an array type as defined in Figure 3.17, v will be translated to the MetaFork codes in Figure 3.18; however, if v is a scalar variable as declared in Figure 3.19, its translated counterpart is illustrated in Figure 3.20.

```
int x[10];
int (*_fip_x)[10]=&x;
memcpy(x,_fip_x,sizeof(x))
```

Figure 3.17: An array type variable

```
int x;
int _fip_x=x;
int x = _fip_x;
```

Figure 3.19: A scalar type variable

```
int a[6] ={2,3,4,5,6,7},m=0;
#pragma omp parallel num_threads(3)
{
    #pragma omp for private(m)
    for(int i=0;i<6;i++)
    {
        m = m + a[i];
    }
}
```

Figure 3.21: A code snippet showing the variable m used as an accumulator

Note that if R is a loop construct, a limit is placed on the behavior of the variables which are in P or F, in order to preserve the correctness in the target MetaFork code. Such variables can not be used as an accumulator among different iterations, since the concept that each thread participating the execution of a loop owns a copy of a variable, is not employed by meta_for. For example, the OpenMP codes in Figure 3.21 describe that each thread participating the execution of the parallel loop, gets a copy of the variable m, and this piece of codes could not be translated to MetaFork codes. In contrast, the OpenMP and MetaFork codes of Figure 3.22 are semantically equivalent and produce expected execution.

Further, unlike the outlining technique shown in Algorithm 4, our compiler ensures that there is no need to modify any references to any variables within the structured-block.

To clarify the meaning of the function Transform, it is worth mentioning that as seen in Section 6.4.2, we use standard RecursiveASTVisitor method to traverse the AST. This implies that the translation of an OpenMP construct is deferred until completion of all the translations, say T, corresponding to its inner OpenMP constructs. Thus, the Transform function
3.6. **Translation**

```c
int a[6] = {2, 3, 4, 5, 6, 7}, b[6], m=0;
#pragma omp parallel num_threads(3)
{
    #pragma omp for private(m)
    for(int i=0;i<6;i++)
    {
        m = a[i];
        b[i] = m * m;
    }
}
```

```c
meta_set_nworkers(3);
meta_for(int i=0;i<6;i++)
{
    int m;
    m = a[i];
    b[i] = m * m;
}
```

**Figure 3.22:** A MetaFork code snippet (right), translated from the left OpenMP codes

---

**Algorithm 8: InitLocalDataEnv(P, F)**

**Input:** P and F are as in Algorithm 5

**Output:** D where D is a set of new generated local variable declarations.

1. Initialize `t` to the construct type of `R`
2. **foreach variable `v` in `F` do**
   3. **if `v` is an array type then**
      4. create a pointer (say `_fip_v`) declaration to `v`, namely `tp`;
      5. generate(`tp`);
      6. generate(`tp`, “`=`”, `&v`);
      7. create a declaration of `v`, namely `tv`;
      8. generate(`tv`);
      9. generate(“memcpy(“, `v`, “,”, `_fip_v`, “,”, “sizeof(“, `v`, “))”);
   10. **if `t` is loop or parallel loop then**
       11. **if variable `v` is not the control variable of the associated parallel loop then**
           12. create the same declaration of `v` with a different variable name (say `_fip_v`), namely `tp`;
           13. generate(`tp`, “`=`”, `v`);
           14. create a declaration of `v`, namely `tv`;
           15. generate(`tv`, “`=`”, `_fip_v`);
    16. **foreach variable `v` in `P` do**
       17. create a declaration of `v`, namely `tv`;
       18. generate(`tv`);
represents the changes happened during T. More precisely, these changes are indirectly introduced by other subroutines, i.e. OuterEnvControl, MetaConstruct, InitLocalDataEnv and PotentialSync, rather than directly made to the C/C++ statements inside the structured-block.

**Algorithm 9: PotentialSync(R)**

**Input:** R is as in Algorithm[5]

**Output:** J where J is a MetaFork synchronization construct or null.

1. Initialize t to the construct type of R
2. Initialize M to the set of {parallel, sections, parallel sections, single}
3. **if** t is in M **then**
   4. **if** there was at least one meta_fork call since the last explicit synchronization point **then**
   5. generate("meta_join");

Lastly, a barrier after finishing execution of a parallel region, may be required, because some OpenMP constructs imply an implied barrier at its end. This is achieved by making a call to PotentialSync which aims to minimize the synchronization points by eliminating redundant barriers. As shown in Figure[3.23], the translation process of the second OpenMP single construct introduces an explicit meta_join construct in the translated codes; while the implicit barrier at the end of the OpenMP parallel construct is ignored to avoid allowing two consecutive meta_join constructs in the translated MetaFork codes.

```c
void imp_barrier()
{
    int tmp1, tmp2, tmp3;
    #pragma omp parallel
    {
        #pragma omp single nowait
        {
            tmp1 = tmp1 * tmp2;
        }
        #pragma omp single
        {
            tmp3 = tmp2 * tmp3;
        } /* an implicit barrier here*/
    } /* an implicit barrier here*/
}

void meta_fork shared( tmp2, tmp1)
{
    tmp1 = tmp1 * tmp2;
}

void meta_join;
```

Figure 3.23: A code snippet showing how to avoid adding redundant barriers when translating the codes from OpenMP to MetaFork
3.6.4 Translation from MetaFork to OpenMP

This is easy in principle since the MetaFork language can be regarded as a subset of the OpenMP language. The meta_fork construct can have the following forms:

- `type var = meta_fork function();`
- `var = meta_fork function();`
- `meta_fork [shared(var)]
  block`

We note that function calls spawned with the meta_fork construct are translated using the task construct of OpenMP. Whereas several concerns related to data attribute guide when to implement the translations. Adherence to the rules defined in Section 3.3 and in Chapter 2.14 of OpenMP application program interface, array, and reference variables have to be explicitly qualified in a shared clause in the translated OpenMP program to ensure the safety of the semantics. In Figures 3.24 and 3.25 there are examples which show how the above forms are translated to the OpenMP task construct. The array variable `arry` and the reference variable `ref` are explicitly qualified as shared in the translated OpenMP codes, as shown in Figure 3.24.

The translation from MetaFork parallel for loops to OpenMP is straightforward and simple

```c
void foo(int i) {
    int m = 0, arry[10];
    int &ref = m;
    int f;
    #pragma omp task shared(f)
    f = test();
    #pragma omp task shared(ref, arry, f)
    {
        f = ref;
        arry[m] = f;
    }
}
```

Figure 3.24: Example of translation from MetaFork to OpenMP

as shown in Figure 3.26. In addition, to allow the parallel execution of the translated OpenMP program, the creation of a new main function is needed as can be seen in Figure 3.27. The function _taskFunc0 replaces the original main function (on the left) and a new main function (on the right) is generated. The new main function supports nested parallelism in OpenMP by invoking a call to omp_set_nested(1).

3.7 Experimentation

In this section, we evaluate the performance and the usefulness of the four MetaFork translators (MetaFork to CilkPlus, CilkPlus to MetaFork, MetaFork to OpenMP, OpenMP to
long fib(long n)
{
    long x, y;
    if (n<2) return n;
    else if (n<BASE)
        return fib_serial(n);
    else
    {
        x = meta_fork fib(n-1);
        y = fib(n-2);
        meta_join;
        return (x+y);
    }
}

long fib(long n)
{
    long x, y;
    if (n<2) return n;
    else if (n<BASE)
        return fib_serial(n);
    else
    {
        #pragma omp task shared(x)
        x = fib(n-1);
        y = fib(n-2);
        #pragma omp taskwait
        return (x+y);
    }
}

Figure 3.25: Example of translating a parallel function call from MetaFork to OpenMP

void parfor()
{
    int n = 10, a[n];
    meta_for(int i = 0; i < n; i++)
    {
        a[i] = i;
    }
}

void parfor()
{
    int n = 10, a[n];
    #pragma omp parallel for
    for(int i = 0; i < n; i++)
    {
        a[i] = i;
    }
}

Figure 3.26: Example of translating a parallel for loop from MetaFork to OpenMP

int _taskFunc0(int argc, char **argv)
{
    /* body*/
}

int main(int argc, char **argv){
    /* body*/
    omp_set_nested(1);
    #pragma omp parallel
    #pragma omp single
    _taskFunc0(argc, argv);
    return 0;
}

Figure 3.27: A code snippet showing how to generate a new OpenMP main function
3.7. Experimentation

To this end, we run these translators on various input programs written either in CilkPlus or OpenMP, or both. We emphasize the fact that our purpose is not to compare the performance of the CilkPlus or OpenMP run-time systems. The reader should notice that the codes used in this study were written by different persons with different levels of expertise. In addition, the reported experimentation is essentially limited to one architecture (Intel Xeon) and one compiler (GCC). Therefore, it would be delicate to draw any clear conclusions comparing CilkPlus and OpenMP.

3.7.1 Experimentation Setup

We conducted three sets of experiments. In the first one, we compared the performance of hand-written codes. The motivation, specified in the introduction, is comparative implementation. In the second one, motivated by the interoperability question raised in the introduction, is dedicated to automatic translation of highly optimized code. Now, for each test-case, we have either a hand-written-and-optimized CilkPlus program or a hand-written-and-optimized OpenMP program. Our goal is to determine whether or not the translated programs have similar serial and parallel running times as their hand-written-and-optimized counterparts. In the last experiment, we compared the parallelism overheads measured the original codes (either CilkPlus or OpenMP) and their translated counterparts.

For all experiments, apart from student’s code, we use codes from the following sources:

1. The BPAS library [http://www.bpaslib.org],
2. John Burkardt’s Home Page [http://people.sc.fsu.edu/~jburkardt/c_src/openmp/openmp.html],
3. the BOTS [53] and
4. the CilkPlus distribution examples [http://sourceforge.net/projects/cilk/].

The source code of those test cases (except BOTS) was compiled as follows:

1. CilkPlus code with GCC 4.8 using -O2 -g -lcilkrts -fcilkplus
2. OpenMP code with GCC 4.8 using -O2 -g -fopenmp

We run all our programs on

1. AMD Opteron 6168 48-core nodes (with 256GB RAM and 12MB L3).
2. Intel Xeon 2.66GHz/6.4GT with 12-cores nodes.

The two main quantities that we measure are:

1. scalability by running our compiled OpenMP and CilkPlus programs on \( p = 1, 2, 4, 6, 8, \ldots \) processors; speedup curves (or running time) data are shown in Figures 3.28, 3.29, 3.30, 3.31, 3.32, 3.33, 3.34, 3.35, 3.36a, 3.36b, 3.37a, 3.38, 3.37b, 3.39a, and 3.39b are mainly collected on Intel Xeon’s nodes and repeated/verified on AMD Opteron nodes.
2. **parallelism overheads** by running our compiled OpenMP and CilkPlus programs on \( p = 1 \) against their serial elisions. This is shown in Table 3.2.

Note that through Section 3.7, the speedup of a code being executed using \( x \) threads is defined by

\[
s(x) = \frac{t_{\text{serial}}}{t_{\text{parallel}}(x)}
\]

where \( t_{\text{serial}} \) is the execution time of the sequential code version and \( t_{\text{parallel}}(x) \) is its parallel counterpart using \( x \) threads.

### 3.7.2 Correctness

Validating the correctness of our translators was a major requirement of our work. Depending on the test-case, we could use one of the following strategies.

1. Assume that the original program, say \( P \), contains both a parallel code and its serial elision (manually written). When program \( P \) is executed, both codes run and compare their results. Let us call \( Q \) the translated version of \( P \). Since serial elisions are unchanged by our translation procedures, then \( Q \) can be verified by the same process used for program \( P \). This first strategy applies to the Cilk++ distribution examples and the BOTS examples.

2. If the original program \( P \) does not include a serial elision of the parallel code, then the translated program \( Q \) is verified by comparing the output of \( P \) and \( Q \). This second strategy had to be applied to the FSU (Florida State University) examples.

### 3.7.3 Comparative Implementation

For this first purpose, we use a series of test-cases, each of them consisting of a pair of hand-written programs: one written in OpenMP and the other in CilkPlus. Within each pair, a program \( S \), written by a student, has a performance bottleneck; meanwhile its counterpart \( E \), written by an expert does not. For each pair, we translate one program (either \( S \) or \( E \)) to the other language. For these two programs (expressed in the same concurrency platform) we measure the running time on \( p \) processors, for \( 1 \leq p \leq 48 \), and compare the resulting data so as to narrow down the performance bottleneck in the inefficient program. Figures 3.28, 3.29, 3.30 and 3.31 illustrate four test-cases.

**Parallel mergesort.** The original OpenMP code (written by a student) misses to parallelize the merge phase and simply spawns the two recursive calls using OpenMP sections while the original CilkPlus code (written by an expert) does parallelize the merge phase. On Figure 3.28 the running time curve of the translated OpenMP code is as theoretically expected while the curve of the original OpenMP code shows a limited scalability. This suggests that the original hand-written OpenMP code should expose more parallelism.
3.7. Experimentation

Matrix inversion. The two original parallel programs are based on different serial algorithms for inverting a dense matrix. The original OpenMP code uses Gauss-Jordan elimination while the original CilkPlus code uses a divide-and-conquer approach based on Schur’s complement. Figure 3.29 shows that the code translated from CilkPlus to OpenMP is more appropriate for fork-join multithreaded languages targeting multi-cores. In other words the Schur’s complement approach should be preferred in this context.

Matrix transposition. The two original parallel programs are based on different algorithms for matrix transposition which is a challenging operation on multi-core architectures. Without doing complexity analysis, discovering that the OpenMP code (written by a student) runs in $O(n^2 \log(n))$ complexity instead of $O(n^2)$ as the CilkPlus (written by Matteo Frigo) is very subtle. Figure 3.30 shows the running time of both the original OpenMP and the translated OpenMP code from the original CilkPlus code and it suggests that the code translated from CilkPlus to OpenMP is more appropriate for fork-join multithreaded languages targeting multi-cores because of the algorithm used in CilkPlus code.

Naive matrix multiplication. This test-case is the naive three-nested-loops matrix multiplication algorithm, where two loops have been parallelized. The parallelism is $O(n^2)$ as for DnC MM in Section 3.7.4. However, the ratio work-to-memory-access is essentially equal to 2, which is much less that for DnC MM. This limits the ability to scale and reduces performance overall on multi-core processors. As you can see from Figure 3.31 both CilkPlus and OpenMP scale poorly because of algorithmic issues.

![Figure 3.28: Parallel mergesort in size 10^8](image1)

![Figure 3.29: Matrix inversion of order 4096](image2)

3.7.4 Interoperability

Our second experiment is dedicated to automatic translation of highly optimized libraries. The motivation, presented in the introduction, is to facilitate interoperability between libraries/

---

3CilkPlus code was borrowed from [https://computing.llnl.gov/tutorials/openMP/samples/C/omp_mm.c](https://computing.llnl.gov/tutorials/openMP/samples/C/omp_mm.c)
codes developed for different concurrency platforms, namely CilkPlus and OpenMP. For this question, we want to determine whether or not the translated programs have similar serial and parallel running times as their hand-written-and-optimized counterparts.

**Fibonacci number computation.** This classical example is often used in the literature. The algorithm computes the integer \( F_n \) (given by \( F_n = F_{n-1} + F_{n-2}, F_1 = 1, F_0 = 0 \)) for a non-negative integer \( n \). Results of intermediate recursive calls are not remembered. Thus, the algorithm is a divide-and-conquer “5-line procedure” with high parallelism and no data traversal. So, a very easy test case for concurrency platforms based on the fork-join parallelism model. In this example, we have translated the original CilkPlus code to OpenMP. The speedup curve for computing Fibonacci with inputs 40 and 45 are shown in Figure 3.32(a) and Figure 3.32(b) respectively. As you can see from Figure 3.32, CilkPlus (original) and OpenMP (translated) codes scale well.
3.7. Experimentation

**Divide-and-conquer matrix multiplication.** This matrix multiplication algorithm [58] (DnC MM) is another easy case for the fork-join parallelism model. It computes the product of two dense matrices by means of a divide-and-conquer procedure which recursively divides the input matrices into blocks, until a base case is reached. When this latter scenario happens, the naive three-nested-loops matrix multiplication algorithm is applied. DnC MM has a high theoretical parallelism (namely $\Theta(n^2)$ in our code) and it is cache-complexity optimal [58] among all dense matrix multiplication algorithms. To be precise, the ratio work-to-memory-access is $\Theta(\sqrt{ZL})$ where $Z$ and $L$ are the cache size and cache line size, respectively. In this example, we have translated the original CilkPlus code to OpenMP code. The speedup curve after computing matrix multiplication with inputs 4096 and 8192 are shown in Figure 3.33(a) and Figure 3.33(b) respectively. As you can see from Figure 3.33 CilkPlus (original) and OpenMP (translated) codes scale well.

**Parallel prefix sum.** This other classical example [20] is often used in research articles dealing with scheduling. Given $n$ items (which could be numbers, matrices, etc.) $x_1, \ldots, x_n$, this divide-and-conquer procedure computes all prefixes $x_1 + \cdots + x_i$ for $i \in \{1, \ldots, n\}$. The theoretical parallelism is in $\Theta(n/\log(n))$ and the ratio work-to-memory-access is constant. This algorithm is, therefore, more challenging to implement on multi-cores than the previous two. In this example, we have translated the original CilkPlus code to OpenMP code. The speedup curves with inputs $5 \cdot 10^8$ and $10^9$ are shown in Figure 3.34(a) and Figure 3.34(b) respectively. As you can see from Figure 3.34 CilkPlus (original) and OpenMP (translated) codes scale well at almost the same rate.

**Quick sort.** This is the classical quick-sort algorithm where the division phase has not been parallelized, on purpose. Consequently, the theoretical parallelism drops to $(\log(n))$. The ratio of work-to-memory-access is constant. The speedup curve for quick sort is shown in Figure 3.35(a) and Figure 3.35(b). As you can see from Figure 3.35 CilkPlus (original) and OpenMP (translated) codes scale at almost the same rate.
Figure 3.34: Speedup curve on Intel node

(a) Prefix_sum : $n = 5 \cdot 10^8$
(b) Prefix_sum : $n = 10^9$

Figure 3.35: Speedup curve on Intel node

(a) Quick Sort: $n = 2 \cdot 10^8$
(b) Quick Sort: $n = 5 \cdot 10^8$
3.7. Experimentation

(a) Mandelbrot set

(b) Linear system solving (dense method)

Figure 3.36: Running time of Mandelbrot set and Linear system solving

Mandelbrot set. This algorithm does not traverse a large set of data and is compute-intensive. The running time after computing the Mandelbrot set with grid size of $500 \times 500$ and 2000 iterations is shown in Figure 3.36a. As you can see from Figure 3.36a both OpenMP (original) and CilkPlus (translated) codes scale well.

Linear system solving (dense method). In this example, different methods of solving the linear system $A \times x = b$ are compared. In this example there is a standard sequential code and slightly modified sequential code to take advantage of OpenMP. The algorithm in this example uses Gaussian elimination. This algorithm has lots of parallelism. However, minimizing parallelism overheads and memory traffic is a challenge for this operation. The running time of this example is shown in Figure 3.36b.

FFT (FSU version). This example demonstrates the computation of a Fast Fourier Transform in parallel. The algorithm used in this example has low work-to-memory-access ratio which is challenging to implement efficiently on multi-cores. The running time of this example is shown in Figure 3.37a. As you can see from the Figure 3.37a both OpenMP (original) and CilkPlus (translated) codes scale well up to 8 cores.

FFT (BOTS). This example computes the one-dimensional Fast Fourier Transform over $n$ complex values using the Cooley-Tukey [41] algorithm. It’s a divide and conquer algorithm that recursively breaks down a Discrete Fourier Transform (DFT) into many smaller DFTs. The speedup curve for this example is shown in Figure 3.37b. It is clear that the translated code scales better.

http://en.wikipedia.org/wiki/Mandelbrot_set
Chapter 3. A Metalanguage for Concurrency Platforms Based on the Fork-Join Model

![Graph of FFT test-cases: FSU version and BOTS version](image)

(a) FFT (FSU) over the complex in size $2^{25}$  
(b) FFT (BOTS)

Figure 3.37: FFT test-cases: FSU version and BOTS version

![Graph of Speedup curve of Protein alignment - 100 Proteins](image)

(a) OpenMP Single version  
(b) OpenMP For version

Figure 3.38: Speedup curve of Protein alignment - 100 Proteins

Protein alignment (BOTS). This algorithm is implemented with the Myers and Miller [115] method. It exposes relatively high parallelism but high communication/synchronization costs. The original code was heavily tuned to address these latter costs. The speedup curve for this example is shown in Figure 3.38. As you can see from the Figure 3.38, both OpenMP (original) and CilkPlus (translated) codes scale at almost the same rate.

Sparse LU matrix factorization (BOTS). This example computes an LU matrix factorization over sparse matrices. The challenge to efficiently implement this algorithm is to deal with the load imbalance issue. The speedup curve for this example is shown in Figure 3.39a. As you can see from the Figure 3.39a, both OpenMP (original) and CilkPlus (translated) codes scale at almost the same rate.
3.7. Experimentation

Figure 3.39: Speedup curve of Sparse LU and Strassen matrix multiplication

**(a) Sparse LU (OpenMP single version) (b) Strassen matrix multiplication**

### Strassen matrix multiplication (BOTS)

Strassen algorithm uses hierarchical decomposition of a matrix for multiplication of large dense matrices. Decomposition is done by dividing each dimension of the matrix into two sections of equal size. As you can see from the Figure 3.39, both OpenMP (original) and CilkPlus (translated) codes scale at almost the same rate.

### BPAS library

For this test-case, we have used the BPAS library which counts more than 150,000 lines of CilkPlus code. Half of those lines are dedicated to polynomial multiplication and we translated those to OpenMP. In Table 3.1, we report on timings of two of the main algorithms for polynomial multiplication, namely 8-way Toom-Cook and divide-and-conquer plain multiplication. One can see that the original and translated codes have similar running times on 1 and 16 cores, for all input data sizes that we tested. Therefore, the OpenMP version of the BPAS library retains the good performance of the original version written in CilkPlus.

#### 3.7.5 Parallelism Overheads

Our third experiment is devoted to the following question: do the MetaFork translators add extra parallelism overheads to the generated code w.r.t. the original code? We focus here on work overhead. By work overhead, we mean the time ratio between a multithreaded program run on one core and its serial elision. For this experiment, we have considered original programs using different parallelism patterns (divide-and-conquer, parallel for-loops) and written in both OpenMP and CilkPlus. Our results are collected in Table 3.2. For all the examples that we tested, we could observe that, if the original program has little work overhead, then the same holds for the translated program.

Chapter 3. A Metalanguage for Concurrency Platforms Based on the Fork-Join Model

Table 3.1: BPAS timings with 1 and 16 workers: original CilkPlus code and translated OpenMP code

<table>
<thead>
<tr>
<th>Test</th>
<th>Input size</th>
<th>CilkPlus</th>
<th></th>
<th>OpenMP</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$T_1$</td>
<td>$T_{16}$</td>
<td></td>
<td>$T_1$</td>
</tr>
<tr>
<td>8-way</td>
<td>2048</td>
<td>0.423</td>
<td>0.231</td>
<td>0.421</td>
<td>0.213</td>
</tr>
<tr>
<td>Toom-Cook</td>
<td>4096</td>
<td>1.849</td>
<td>0.76</td>
<td>1.831</td>
<td>0.644</td>
</tr>
<tr>
<td></td>
<td>8192</td>
<td>9.646</td>
<td>2.742</td>
<td>9.241</td>
<td>2.774</td>
</tr>
<tr>
<td></td>
<td>16384</td>
<td>39.597</td>
<td>9.477</td>
<td>39.051</td>
<td>8.805</td>
</tr>
<tr>
<td></td>
<td>32768</td>
<td>174.365</td>
<td>34.863</td>
<td>172.562</td>
<td>33.032</td>
</tr>
<tr>
<td>DnC</td>
<td>2048</td>
<td>0.874</td>
<td>0.259</td>
<td>0.867</td>
<td>0.299</td>
</tr>
<tr>
<td>Plain</td>
<td>4096</td>
<td>3.95</td>
<td>1.264</td>
<td>3.925</td>
<td>1.123</td>
</tr>
<tr>
<td>Polynomial</td>
<td>8192</td>
<td>18.196</td>
<td>3.335</td>
<td>18.154</td>
<td>4.428</td>
</tr>
<tr>
<td>Multiplication</td>
<td>16384</td>
<td>77.867</td>
<td>12.778</td>
<td>75.885</td>
<td>12.674</td>
</tr>
<tr>
<td></td>
<td>32768</td>
<td>331.351</td>
<td>55.841</td>
<td>332.126</td>
<td>55.925</td>
</tr>
</tbody>
</table>

Table 3.2: Timings on AMD 48-core: underlined timings refer to original code and non-underlined timings to translated code

<table>
<thead>
<tr>
<th>Test</th>
<th>Input size</th>
<th>CilkPlus</th>
<th></th>
<th>OpenMP</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Serial</td>
<td>$T_1$</td>
<td>Serial</td>
<td>$T_1$</td>
</tr>
<tr>
<td>Protein alignment (for)</td>
<td>100</td>
<td>24.82</td>
<td>24.82</td>
<td>25.16</td>
<td>25.15</td>
</tr>
<tr>
<td>quicksort</td>
<td>$5 \cdot 10^8$</td>
<td>89.72</td>
<td>91.94</td>
<td>89.83</td>
<td>91.61</td>
</tr>
<tr>
<td>prefixsum</td>
<td>$5 \cdot 10^8$</td>
<td>13.04</td>
<td>14.71</td>
<td>13.19</td>
<td>14.92</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>45</td>
<td>8.661</td>
<td>8.666</td>
<td>8.87</td>
<td>8.88</td>
</tr>
<tr>
<td>DnC_MM</td>
<td>4096</td>
<td>754.11</td>
<td>756.55</td>
<td>754.57</td>
<td>759.09</td>
</tr>
<tr>
<td>Mandelbrot</td>
<td>$500 \times 500$</td>
<td>0.64</td>
<td>0.64</td>
<td>0.64</td>
<td>0.65</td>
</tr>
</tbody>
</table>

3.8 Summary

MetaFork allows for rapidly mapping algorithms written for one concurrency platform to another. As we have seen in Section 3.7, MetaFork can be applied for (1) comparing algorithms written with different concurrency platforms and (2) porting more programs to systems that may have a highly optimized run-time for one paradigm (say divide-and-conquer algorithms, or producer-consumer).

The MetaFork translation framework may also avoid the negative interferences of having multiple interfaces between the different components of a large solver written with various concurrency platforms. Along the same idea, the MetaFork translators can be used to transform legacy code into a more adequate concurrency platform.

The experimental results of Section 3.7, as well as those reported in the technical report [34], suggest that our translators can be used to narrow performance bottlenecks down. By translating a parallel program with low performance, we could suspect the cause of inefficiency whether this cause was a poor implementation (in the case of Parallel mergesort, where
not enough parallelism was exposed) or an algorithm inefficient in terms of data locality (in the case of *Matrix inversion*) or an algorithm inefficient in terms of work (in the case of *Matrix transposition*).

For the second part of our experimentation, our results show that the speedup curves of the original and translated codes either match or have similar shape. Nevertheless, in some cases, either the original or the translated program outperforms its counterpart. For John Burkardt’ programs, the speedup curves of the original programs are close to those of the translated *CilkPlus* programs. We observe that the only parallel construct used in his source examples are parallel for-loops. In addition, the results of Table 3.2 show that if a *CilkPlus* (resp. *OpenMP*) program has little work overhead, the same would hold for its *OpenMP* (resp. *CilkPlus*) counterpart translated by *MetaFork*. Last but not least, we think that a great benefit of *MetaFork* is the abstraction that it provides. It can be useful for parallel language design (for example in designing parallel extensions to C/C++) as well as a good tool to teach parallel programming.

**Acknowledgments.** This work was supported in part by NSERC of Canada and in part by an IBM CAS Fellowship in 2013 and 2014. We are also grateful to Abdoul-Kader Keita (IBM Toronto Lab) for his advice and technical support.

www.metafork.org
Chapter 4

Applying \textsc{MetaFork} to the Generation of Parametric CUDA Kernels

In this chapter, we present the accelerator model of \textsc{MetaFork} together with the software framework that allows automatic generation of CUDA code from annotated \textsc{MetaFork} programs. One of the key properties of this CUDA code generator is that it supports the generation of CUDA kernel code where program parameters (like number of threads per block) and machine parameters (like shared memory size) are allowed. These parameters need not to be known at code-generation-time: machine parameters and program parameters can be respectively determined and optimized when the generated code is installed on the target machine.

The need for CUDA programs (more precisely, kernels) depending on program parameters and machine parameters is argued in Section 4.1. In Section 4.2, following the authors of [67], we observe that generating parametric CUDA kernels require the manipulation of systems of non-linear polynomial equations and the use of techniques like quantifier elimination (QE). To this end, we take advantage of the \texttt{RegularChains} library of \textsc{Maple} [32] and its \texttt{QuantifierElimination} command which has been designed to efficiently support the non-linear polynomial systems coming from automatic parallelization.

Section 4.3 is an overview of the \textsc{MetaFork} language constructs for generating SIMD code. Section 4.4 reports on a preliminary implementation of the \textsc{MetaFork} generator of parametric CUDA kernels from input \textsc{MetaFork} programs. In addition to the \texttt{RegularChains} library, we take advantage of PPCG, the polyhedral parallel code generation for CUDA [147] that we have adapted so as to produce parametric CUDA kernels. Finally, Section 4.5 gathers experimental data demonstrating the performance of our generated parametric CUDA code. Not only these results show that the generation of parametric CUDA kernels helps optimizing code independently of the values of the machine parameters of the targeted hardware, but also these results show that automatic generation of parametric CUDA kernels may discover better values for the program parameters than those computed by a tool generating non-parametric CUDA kernels.

52
4.1 Optimizing CUDA Kernels Depending on Program Parameters

Estimating the amount of computing resource (time, space, energy, etc.) that a parallel program, written in a high-level language, required to run on a specific hardware is a well-known challenge. A first difficulty is to define models of computation retaining the computer hardware characteristics that have a dominant impact on program performance. That is, in addition to specify the appropriate complexity measures, those models must consider the relevant parameters characterizing the abstract machine executing the algorithms to be analyzed. A second difficulty is, for a given model of computation, to combine its complexity measures so as to determine the “best” algorithm among different algorithms solving a given problem. Models of computation which offer those estimates necessarily rely on simplification assumptions. Nevertheless, such estimates can deliver useful predictions for programs satisfying appropriate properties.

In [71], the authors propose a many-core machine (MCM) model for multithreaded computation combining the fork-join and SIMD parallelisms; a driving motivation in this work is to estimate parallelism overheads (data communication and synchronization costs) of GPU programs. In practice, the MCM model determines a trade-off among work, span and parallelism overhead by checking the estimated overall running time so as to (1) either tune a program parameter or, (2) compare different algorithms independently of the hardware details.

We illustrate the use of the MCM model with a very classical example: the computation of Fast Fourier Transforms (FFTs). Our goal is to compare the running times of two of the most commonly used FFT algorithms: that of Cooley & Tukey [41] and that of Stockham [140].

Let $f$ be a vector over a field $\mathbb{K}$ of coefficients, say the complex numbers. Assume that $f$ has size $n$ where $n$ is a power of 2. Let $U$ be the time (expressed in clock cycles) to transfer one machine word between the global memory and the private memory of any SM \(^1\), that is, $U > 0$. Let $Z$ be the size (expressed in machine words) of the private memory of any SM, which sets up an upper bound on several program parameters. Let $\ell \geq 2$ be a positive integer. For $Z$ large enough, both Cooley & Tukey algorithm and Stockham algorithm can be implemented by calling $\log_2(n)$ times a kernel using $\Theta(\frac{n}{\ell})$ SMs each of those SMs executing a thread block with $\ell$ threads, with a respective work \(^2\) of $W_{ct} = n \left(34 \log_2(n) \log_2(\ell) + 47 \log_2(n) + 333 - 136 \log_2(\ell)\right)$ and $W_{sh} = 43 n \log_2(n) + \frac{n^2}{2} + 12 \ell + 1 - 30 n$, with a respective span \(^3\) of $S_{ct} = 34 \log_2(n) \log_2(\ell) + 47 \log_2(n) + 2223 - 136 \log_2(\ell)$ and $S_{sh} = 43 \log_2(n) + 16 \log_2(\ell) + 3$, with a respective overhead \(^4\) of $O_{ct} = 2 n U \left(\frac{4 \log_2(n)}{\ell} + \log_2(\ell) - \frac{\log_2(\ell)+15}{\ell}\right)$ and $O_{sh} = \ldots$

---

\(^1\) In the MCM model, the number of SMs is unbounded as well as the size of the global memory, whereas each SM has a private memory of size $Z$.

\(^2\) In the MCM model, the work of a thread block is the total number of local operations (arithmetic operation, read/write memory accesses in the private memory of an SM) performed by all its threads; the work of a kernel is the sum of the works of all its thread blocks.

\(^3\) In the MCM model, the span of a thread block is the maximum number of local operations performed by one of its threads; the span of a kernel is the maximum span among all its thread blocks.

\(^4\) In the MCM model, the parallelism overhead (or overhead, for short) of a thread block accounts for the time to transfer data between the global memory of the machine and the private memory of the SM running this thread.
\[
\frac{5nU \log_2(n)}{\ell} + \frac{5nU}{4\ell}
\]
See [71] for details on the above estimates. From those, one observes that the overhead of Cooley & Tukey algorithm has an extraneous term in \(O(nU\log_2(\ell))\) which is due to higher amount of non-coalesced accesses. In addition, when \(n\) escapes to infinity (while \(\ell\) remains bounded over on a given machine since we have \(\ell \in O(Z)\)) the work and span of the algorithm of Cooley & Tukey are increased by a \(\Theta(\log_2(\ell))\) factor w.r.t their counterparts in Stockham algorithm.

These theoretical observations suggest that, as \(\ell\) increases, Stockham algorithm performs better than the one of Cooley & Tukey. This has been verified experimentally\(^5\) by the authors of [71] as well as by others, see [109] and the papers cited therein. On the other hand, it was also observed experimentally that for \(\ell\) small, Cooley & Tukey algorithm is competitive with that of Stockham. Overall, this suggests that generating kernel code, for both algorithms, where \(\ell\) is an input parameter, is a desirable goal. With such parametric codes, one can choose at run-time the most appropriate FFT algorithm, once \(\ell\) has been chosen.

The MCM model retains many of the characteristics of modern GPU architectures and programming models, like CUDA [116] and OpenCL [141]. However, in order to support algorithm analysis with an emphasis on parallelism overheads, the MCM abstract machines admit a few simplifications and limitations with respect to actual many-core devices.

To go further in our discussion of CUDA kernel performance, let us consider now the programming model of CUDA itself and its differences w.r.t. the MCM model. In CUDA, instructions are issued per warp; a warp consists of a fixed number \(S_{\text{warp}}\) of threads. Typically \(S_{\text{warp}}\) is 32 and, thus, executing a thread block on an SM means executing several warps in turn. If an operand of an executing instruction is not ready, then the corresponding warp stalls and context switch happens between warps running on the same SM.

Registers and shared memory are allocated for a thread block as long as that thread block is active. Once a thread block is active it will stay active until all threads in that thread block have completed. Context switching is very fast because registers and shared memory do not need to be saved and restored. The intention is to hide the latency (of data transfer between the global memory and the private memory of an SM) by having more memory transactions in fly. There is, of course, a hardware limitation to this, characterized by (at least) two numbers:

- the maximum number of active warps per SM, denoted here by \(M_{\text{warp}}\); A typical value for \(M_{\text{warp}}\) is 48 on a Fermi NVIDIA GPU card, leading to a maximum number of \(32 \times 48 = 1536\) active threads per SM.
- the maximum number of active thread blocks per SM, denoted here by \(M_{\text{block}}\); A typical value for \(M_{\text{block}}\) is 8 on a Fermi NVIDIA GPU card.

One can now define a popular performance counter of CUDA kernels, the occupancy of an SM: it is given by \(A_{\text{warp}}/M_{\text{warp}}\), where \(A_{\text{warp}}\) is the number of active warps on that SM. Since resources (registers, shared memory, thread slots) are allocated for an entire thread block (as long as that block is active) there are three potential limitations to occupancy: register usage, shared memory usage and thread block size. As in our discussion of the MCM model, we denote the thread block size by \(\ell\). Two observations regarding the possible values of \(\ell\):

\(^5\)Our algorithms are implemented in CUDA and publicly available with benchmarking scripts from [http://www.cumodp.org/](http://www.cumodp.org/)
4.2 Automatic Generation of Parametric CUDA Kernels

The total number of active threads is bounded over by $M_{\text{block}} \ell$, hence a small value for $\ell$ may limit occupancy.

A larger value for $\ell$ will reduce the amount of registers and shared memory words available per thread; this will limit data reuse within a thread block and, thus, will potentially increase the amount of data transfer between global memory and the private memory of an SM.

Overall, this suggests again that generating kernel code, where $\ell$, and other program parameters are input arguments, is a desirable goal. With such parametric code at hand, one can optimize at run-time the values of those program parameters (like $\ell$) once the machine parameters (like $S_{\text{warp}}$, $M_{\text{warp}}$, $M_{\text{block}}$, $Z$ (private memory size) and the size of the register file) are known.

4.2 Automatic Generation of Parametric CUDA Kernels

The general purpose of automatic parallelization is to convert sequential computer programs into multithreaded or vectorized code. Following the discussion of Section 4.1, we are interested here in the following more specific question.

Given a theoretically good parallel algorithm (e.g. divide-and-conquer matrix multiplication) and given a type of hardware that depends on various parameters (e.g. a GPGPU with $Z$ words of private memory per SM and a maximum number $M_{\text{warp}}$ of warps supported by an SM, etc.) we aim at automatically generating CUDA kernels that depends on the hardware parameters ($Z$, $M_{\text{warp}}$, etc.), as well as program parameters (e.g. number $\ell$ of threads per block), such that those parameters need not to be known at compile-time, and are encoded as symbols in the generated kernel code. For this reason, we call such CUDA kernels parametric.

In contrast, current technology requires that machine and program parameters are specialized to numerical values at the time of generating the GPGPU code, see [69, 16, 76, 147].

For example, for the following code computing the product of two univariate polynomials $a$ and $b$, both of degree $n$, and writing the result to $c$,

\begin{verbatim}
for(i=0; i<=n; i++) {c[i] = 0; c[i+n] = 0;}
for(i=0; i<=n; i++) {
    for(j=0; j<=n; j++)
        c[i+j] += a[i] * b[j];
}
\end{verbatim}

...
two represent a thread block index and a thread index (within a thread block). This brings the following additional relations:

\[
\begin{align*}
0 &\leq b \\
0 &\leq u < B \\
p &= bB + u,
\end{align*}
\] (4.1)

To the previous system

\[
\begin{align*}
0 &< n \\
0 &\leq i \leq n \\
0 &\leq j \leq n \\
t &= n - j \\
p &= i + j.
\end{align*}
\] (4.2)

To determine the target program, one needs to eliminate the variables \( i \) and \( j \). In this case, Fourier-Motzkin elimination (FME) does not apply any more, due to the presence of non-linear constraints. If all the non-linear constraints appearing in a system of relations are polynomial constraints, the set of real solutions of such a system is a semi-algebraic set. The celebrated Tarski theorem [18] tells us that there always exists a quantifier elimination algorithm to project a semi-algebraic set of \( \mathbb{R}^n \) to a semi-algebraic set of \( \mathbb{R}^m, m \leq n \). The most popular method for conducting quantifier elimination (QE) of a semi-algebraic set is through cylindrical algebraic decomposition (CAD) [40]. Implementation of QE and CAD can be found in software such as Qepcad [27], Reduce [73], Mathematica [153], as well as the RegularChains library of Maple [32]. Using the function QuantifierElimination (with options ‘precondition’=‘AP’, ‘output’=’rootof’, ‘simplification’=’L4’) in the RegularChains library, we obtain the following:

\[
\begin{align*}
B &> 0 \\
n > 0 \\
0 &\leq b \leq 2n/B \\
0 &\leq u < B \\
0 &\leq u \leq 2n - Bb \\
p &= bB + u, \\
0 &\leq t \leq n, \\
n - p &\leq t \leq 2n - p,
\end{align*}
\] (4.3)

from where we derive the following program:

```c
for (int p = 0; p <= 2*n; p++) { c[p]=0; }
parallel_for (int b = 0; b <= 2*n/B; b++) {
  for (int u = 0; u <= min(B-1, 2*n-B*b); u++) {
    int p = b * B + u;
    for (int t = max(0, n-p); t <= min(n, 2*n-p); t++)
      c[p] += a[t+p-n] * b[n-t];
  }
}
```

An equivalent CUDA kernel to the parallel_for part is as below:
We remark that the polynomial system defined by (4.1) and (4.2) has some special structure. The authors in [66] have exploited this structure to deduce a special algorithm to solve it and similar problems by implementing some parametric FME. Although the system (4.3) can be directly processed by QuantifierElimination, we found that it is much more efficient to use the following special QE procedure. We replace the product $bB$ in system (4.1) by a new variable $c$, and thus obtain a system of linear constraints. We then apply FME to eliminate the variables $i, j, t, p, u$ in sequential. Now we obtain a system of linear constraints in variables $c, b, n, B$. Next we replace $c$ by $bB$ and have again a system of non-linear constraints in variables $b, n, B$. We then call QuantifierElimination to eliminate the variables $b, n, B$. The correctness of the procedure is easy to verify.

4.3 Extending the MetaFork Language to Support Device Constructs

We enhance the MetaFork language with constructs allowing the programmer to express the fact that a function call can be executed on an external (or remote) hardware component. This latter is referred as the device while the hardware component on which the MetaFork program was initially launched is referred as the host and this program is then called the host code. Both the host and the device maintain their own separate memory spaces. Such function calls on an external device are expressed by means of two new keywords: meta_device and meta_copy.

We call a statement of the form

```
meta_device (variable declaration)
```

a device declaration; it is used to express the fact that a variable is declared in the memory address space of the device.

A statement of the form

```
meta_device (function call)
```

is called a device function call; it is used to express the fact that a function call is executed on the device concurrently (thus in a non-blocking way) to the execution of the parent thread on the host. All arguments in the function call must be either device-declared variables or values from primitive types (char, float, int, double).

A statement of the form

```
meta_copy ((range), (variable), (variable))
```

```c
int b = blockIdx.x;
int u = threadIdx.x;
if (u <= 2 * n - B * b) {
    int p = b * B + u;
    for (int t = max(0, n-p); t <= min(n, 2*n-p); t++)
        c[p] += a[t+p-n] * b[n-t];
}
```
void foo()
{
    int arry_host[N];
    initialize(arry_host, N);
    meta_fork bar(arry_host, N);
    work();
}

void foo()
{
    int arry_host[N];
    initialize(arry_host, N);
    // declare an array on the device
    meta_device int arry_device[N];
    // copy 24 bytes of host array
    // from host to device
    meta_copy(arry_host+8, arry_host+32,
        arry_host, arry_device);
    meta_device bar(arry_device, N);
    work();
}

Figure 4.1: MetaFork examples

copies the bytes whose memory addresses are in range (and who are assumed to be data referenced by the first variable) to the memory space referenced by the second variable. Moreover, either one or both variables must be device-declared variables.

The left part of Figure 4.1 shows a MetaFork code fragment with a spawned function call operating on an array located in the shared memory of the host. On the right part of Figure 4.1 the same function is called on an array located in the memory of the device. In order for this function call to perform the same computation on the device as on the host, the necessary coefficients of the host array are copied to the device array. In this example, we assume that those coefficients are arry_host[2], ..., arry_host[8].

Several devices can be used within the same MetaFork program. In this case, each of the constructs above is followed by a number referring to the device to be used. Therefore, these function calls on an external device can be seen as one-sided message passing protocol, similar to the one of the Julia programming language.

We stress the following facts about function calls on an external device. Any function declared in the host code can be invoked in a device function call. Moreover, within the body of a function invoked in a device function call, any other function defined in host code can be: (1) either called (in the ordinary way, that is, as in the C language) and then executed on the same device, or (2) called on another device. As a consequence, device function calls together with spawned function calls and ordinary function calls form a directed acyclic graphs of tasks to which the usual notions of the fork-join concurrency model can be applied.

The mechanism of function call on an external device can be used to model the distributed computing model of Julia as well as heterogeneous computing in the sense of CUDA. The latter is, however, more complex since, as in discussed in Section 4.1, it borrows from both the fork-join concurrency model and SIMD parallelism. In particular, a CUDA kernel call induces how the work is scheduled among the available SMs. Since our goal is to generate efficient CUDA code from an input MetaFork program, it is necessary to annotate MetaFork code in a more precise manner. To this end, we introduce another keyword, namely meta_schedule.

Any block of MetaFork code (like a meta_for-loop) can be the body of meta_schedule statement. The semantic of that statement is that of its body and meta_schedule is an in-

---

6 The difference between the lower end of the range and the memory address of the source array is used as offset to write in the second variable.

7 Julia web site: [http://julialang.org/](http://julialang.org/)
4.3. Extending the MetaFork Language to Support Device Constructs

dication to the MetaFork-to-CUDA translator that every meta_for-loop nest of the meta_schedule statement must translate to a CUDA kernel call.

A meta_schedule statement generating a one-dimensional grid with one-dimensional thread block has the structure in Figure 4.2, where the grid (resp. thread block) dimension size is extracted from the outer (resp. inner) meta_for loop upper bound. Similarly, a meta_schedule statement generating a two-dimensional grid with two-dimensional thread blocks has the structure in Figure 4.3, where the first two outer meta_for loops correspond to the grid and the inner meta_for loops to the thread blocks. We skip the other possible configurations since they have not been implemented yet in the MetaFork compilation framework.

```
meta_schedule {
    // only for loops are supported here
    meta_for (int i = 0; i < blockDim.x; i++) {
        ... // nested for-loop body
    }
}
```

Figure 4.2: Using meta_schedule to define one-dimensional CUDA grid and thread block

```
meta_schedule {
    // only for loops are supported here
    meta_for (int u = 0; u < blockDim.y; u++) {
        meta_for (int i = 0; i < blockDim.x; i++) {
            // only for loops are supported here
            meta_for (int v = 0; v < blockDim.y; v++) {
                meta_for (int j = 0; j < blockDim.x; j++) {
                    ... // nested for-loop body
                }
            }
        }
    }
}
```

Figure 4.3: Using meta_schedule to define two-dimensional CUDA grid and thread block

In order to obtain the serial C-elision of MetaFork programs containing device constructs, we apply the following rules: (1) similarly to a cilk_spawn function call, meta_schedule and meta_device keywords are replaced by null, and (2) meta_copy keyword is replaced by null along with its parameters.

We conclude this section with an example, a one-dimensional stencil computation, namely Jacobi. The original (and naive) C version is shown in Figure 4.4. From this C code fragment, we apply the tiling techniques mentioned in Section 4.2 and obtain the MetaFork code shown in Figure 4.5. Observe that the meta_schedule statement has two meta_for loop nests.
for (int $t = 0; t < T; ++t) {
    for (int $i = 1; $i < N-1; ++$i)
        $b[i] = (a[i-1] + a[i] + a[i+1]) / 3;

    for (int $i = 1; $i < N-1; ++$i)
        $a[i] = $b[i];
}

Figure 4.4: Sequential C code computing Jacobi

int $ub_v = (N - 2) / B;

meta_schedule {
    for (int $t = 0; $t < T; ++$t) {
        meta_for (int $v = 0; $v < $ub_v; $v++) {
            meta_for (int $u = 0; $u < B; $u++) {
                int $p = $v * B + $u + 1;
                int $y = $p - 1;
                int $z = $p + 1;
                $b[p] = (a[y] + a[p] + a[z]) / 3;
            }
        }
        meta_for (int $v = 0; $v < $ub_v; $v++) {
            meta_for (int $u = 0; $u < B; $u++) {
                int $w = $v * B + $u + 1;
                $a[w] = $b[w];
            }
        }
    }
}

Figure 4.5: Generated MetaFork code from the code in Figure 4.4
4.4 The MetaFork Generator of Parametric CUDA Kernels

In Section 4.2, we illustrated the process of parametric CUDA kernel generation from a sequential C program using MetaFork as an intermediate language. In this section, we assume that, from a C program, one has generated a MetaFork program which contains one or more meta_schedule blocks. Each such block contains parameters like thread block dimension sizes and is meant to be translated into a CUDA kernel.

For that latter task, we rely on PPCG [147], a C-to-CUDA code generator, that we have modified in order to generate parametric CUDA kernels. Figure 4.6 illustrates the software architecture of our C-to-CUDA code generator, based on PPCG. Since the original PPCG framework does not support parametric CUDA kernels the relevant data structures like non-linear expressions in the for-loop lower and upper bounds, are not supported either. Consequently, part of our code generation is done in a post-processing phase.

Among our adaptation of the PPCG framework, we have added a new node in the PET (Polyhedral Extraction Tool) [148] phase of PPCG in order to represent the information of a meta_for loop; this is, in fact, very similar to a C for-loop except that we tag the outer meta_for loops as blocks and the inner meta_for loops as threads directly.

Taken Figure 4.5 as an example, our MetaFork-to-CUDA translator produces two kernel functions, a header file (for those two kernels) and a host code file where those kernels are called. Those two kernel functions are shown in Figure 4.7. In each kernel, we use the shared memory for those arrays read and use the global memory for those arrays written only once. Observe that kernel0 and kernel1 take a program parameter, the thread block format B, as an
argument, whereas non-parametric CUDA kernels usually take parameters $a, b, c, N, T, c0$ only. Correspondingly, the generated host code replacing `meta_schedule` and its body is shown in Figure 4.8. Data transfers between the CPU and GPU global memories are done before those two kernels launched and after those two kernels completed, respectively. In the case that the number of thread blocks per grid, aka `ub_v` in the MetaFork code, exceeds the hardware limit, which is 32768 shown in the host code, each kernel uses 32768 as the grid dimension size, while inside the kernel code, the amount of work per thread block is incremented via a serial loop.

```c
__global__ void kernel0(int *a, int *b, int N, int T, int ub_v,
                         int B, int c0) {

    int b0 = blockIdx.x;
    int t0 = threadIdx.x;
    int private_p;
    __shared__ int shared_a[BLOCK_0+2]; // BLOCK_0 = B

    for (int c1 = b0; c1 < ub_v; c1 += 32768) {
        for (int c2 = t0; c2 <= min(B + 1, N - B * c1 - 1); c2 += B)
            shared_a[c2] = a[B * c1 + c2];
        __syncthreads();
        private_p = (((c1) * (B)) + (t0));
        b[private_p + 1] = (((shared_a[private_p - B * c1] +
                          shared_a[private_p - B * c1 + 1]) +
                          shared_a[private_p - B * c1 + 2]) / 3);
        __syncthreads();
    }
}
__global__ void kernel1(int *a, int *b, int N, int T, int ub_v,
                         int B, int c0) {

    int b0 = blockIdx.x;
    int t0 = threadIdx.x;
    int private_w;
    __shared__ int shared_b[BLOCK_0]; // BLOCK_0 = B

    for (int c1 = b0; c1 < ub_v; c1 += 32768) {
        if (N >= t0 + B * c1 + 2)
            shared_b[t0] = b[t0 + B * c1 + 1];
        __syncthreads();
        private_w = (((c1) * (B)) + (t0));
        a[private_w + 1] = shared_b[private_w - B * c1];
        __syncthreads();
    }
}
```

Figure 4.7: Generated parametric CUDA kernel for 1D Jacobi

Since the MetaFork code is obtained after computing affine transformation and tiling (via
if (T >= 1 && ub_v >= 1 && B >= 0) {
    #define cudaCheckReturn(ret) \    do { \        cudaError_t cudaCheckReturn_e = (ret); \        if (cudaCheckReturn_e != cudaSuccess) { \            fprintf(stderr, "CUDA error: %s\n", \                cudaGetErrorString(cudaCheckReturn_e)); \            fflush(stderr); \        } \        assert(cudaCheckReturn_e == cudaSuccess); \    } while(0)
    #define cudaCheckKernel() \    do { \        cudaCheckReturn(cudaGetLastError()); \    } while(0)

    int *dev_a;
    int *dev_b;

    cudaCheckReturn(cudaMalloc((void **) &dev_a, (N) * sizeof(int)));
    cudaCheckReturn(cudaMalloc((void **) &dev_b, (N) * sizeof(int)));

    if (N >= 1) {
        cudaMemcpy(dev_a, a, (N) * sizeof(int), cudaMemcpyHostToDevice);
        cudaMemcpy(dev_b, b, (N) * sizeof(int), cudaMemcpyHostToDevice);
    }

    for (int c0 = 0; c0 < T; c0 += 1) {
        dim3 k0_dimBlock(B);
        dim3 k0_dimGrid(ub_v <= 32767 ? ub_v : 32768);
        kernel0 <<<k0_dimGrid, k0_dimBlock>>> (dev_a, dev_b, N, T, ub_v, B, c0);
        cudaCheckKernel();

        dim3 k1_dimBlock(B);
        dim3 k1_dimGrid(ub_v <= 32767 ? ub_v : 32768);
        kernel1 <<<k1_dimGrid, k1_dimBlock>>> (dev_a, dev_b, N, T, ub_v, B, c0);
        cudaCheckKernel();
    }

    if (N >= 1) {
        cudaMemcpy(a, dev_a, (N) * sizeof(int), cudaMemcpyDeviceToHost);
        cudaMemcpy(b, dev_b, (N) * sizeof(int), cudaMemcpyDeviceToHost);
    }
}

cudaCheckReturn(cudaFree(dev_a));
cudaCheckReturn(cudaFree(dev_b));

Figure 4.8: Generated host code for 1D Jacobi
quantifier elimination), we had to bypass the process of computing affine transformation and tiling that PPCG is performing. By doing this, our prototype C-to-CUDA code generator could not fully take advantage of PPCG, which explains why post-processing was necessary. Of course, improving this design is work in progress so as to completely avoid post-processing.

4.5 Experimentation

In this section, we present experimental results on an NVIDIA Tesla M2050. Most of them were obtained by running times of CUDA programs generated with our preliminary implementation of our MetaFork-to-CUDA code generator described in Section 4.4, and the original version of the PPCG C-to-CUDA code generator [147]. We use eight simple examples: array reversal (Figure 4.9, Table 4.1), 1D Jacobi (Table 4.2), 2D Jacobi (Figure 4.10, Table 4.3), LU decomposition (Figure 4.11, Table 4.4), matrix transposition (Figure 4.12, Table 4.5), matrix addition (Figure 4.13, Table 4.6), matrix vector multiplication (Figure 4.14, Table 4.7) and matrix matrix multiplication (Figure 4.15, Table 4.8). In all cases, we use dense representations for our matrices and vectors.

For both the PPCG C-to-CUDA and our MetaFork-to-CUDA code generators, Tables 4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.7 and 4.8 give the speedup factors of the generated code, as the timing ratio of the generated code to their untiled C code. Since PPCG determines a thread block format, the timings in those tables corresponding to PPCG depend only on the input data size. Meanwhile, since the CUDA kernels generated by MetaFork are parametric, the MetaFork timings are obtained for various formats of thread blocks and various input data sizes. Indeed, recall that our generated CUDA code admits parameters for the dimension sizes of the thread blocks. This generated parametric code is then specialized with the thread block formats listed in the first column of those tables.

Figures 4.9, 4.5 (with 4.4 and 4.7), 4.10, 4.11, 4.12, 4.13, 4.14 and 4.15 show the MetaFork code of eight examples with their untiled serial C programs and automatically generated CUDA kernels. In order to allocate unit sizes of shared memory in the kernel code, we predefined BLOCK_0 and BLOCK_1 (if applicable) as macros and specify their values at compile time. The tiled code for each MetaFork program is done by the quantifier elimination (QE) from the RegularChains library of Maple. These eight examples generated by PPCG are shown in Appendix B.

Array reversal. Both MetaFork and PPCG generate CUDA code that uses a one-dimensional kernel grid and the shared memory. We specialize the MetaFork generated parametric code successively to the thread block size $B = 16, 32, 64, 128, 256, 512$; meanwhile, PPCG by default chooses 32 as the thread block size. As we can see in Table 4.1, based on the generated parametric CUDA kernel, one can tune the thread block size to be 256 to obtain the best performance.

1D Jacobi. Our second example is a one-dimensional stencil computation, namely 1D Jacobi. The kernel generated by MetaFork uses a 1D kernel grid and the shared memory, while the kernel generated by PPCG uses a 1D kernel grid and the global memory. PPCG by default chooses a thread block format of 32, while MetaFork preferred format is 64.

2D Jacobi. Our next example is a two-dimensional stencil computation, namely 2D Jacobi. Both the CUDA kernels generated by MetaFork and PPCG use a 2D kernel grid and the global
4.5. Experimentation

Table 4.1: Speedup comparison of reversing a one-dimensional array between PPCG and MetaFork kernel code

<table>
<thead>
<tr>
<th>Speedup (kernel)</th>
<th>Input size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread block size</td>
<td>(2^{23})</td>
</tr>
<tr>
<td>PPCG</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>8.312</td>
</tr>
<tr>
<td>MetaFork</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>4.035</td>
</tr>
<tr>
<td>32</td>
<td>7.612</td>
</tr>
<tr>
<td>64</td>
<td>13.183</td>
</tr>
<tr>
<td>128</td>
<td>19.357</td>
</tr>
<tr>
<td>256</td>
<td>\textbf{20.451}</td>
</tr>
<tr>
<td>512</td>
<td>18.768</td>
</tr>
</tbody>
</table>

Serial code

```c
for (int i = 0; i < N; i++)
    Out[N - 1 - i] = In[i];
```

MetaFork code

```c
int ub_v = N / B;
meta_schedule {
    meta_for (int v = 0; v < ub_v; v++) {
        meta_for (int u = 0; u < B; u++) {
            int private_inoffset;
            int private_outoffset;
            __shared__ int shared_In[BLOCK_0]; // BLOCK_0 = B

            for (int c0 = b0; c0 < ub_v; c0 += 32768) {
                if (N >= t0 + B * c0 + 1)
                    shared_In[t0] = In[t0 + B * c0];
                __syncthreads();
                private_inoffset = (((c0) * (B)) + (t0));
                private_outoffset = (((N) - 1) - private_inoffset);
                Out[private_outoffset] = shared_In[private_inoffset - B * c0];
                __syncthreads();
            }
        }
    }
}
```

Figure 4.9: Serial code, MetaFork code and generated parametric CUDA kernel for array reversal

Table 4.2: Speedup comparison of 1D Jacobi between PPCG and MetaFork kernel code

<table>
<thead>
<tr>
<th>Speedup (kernel)</th>
<th>Input size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread block size</td>
<td>(2^{13} + 2)</td>
</tr>
<tr>
<td>PPCG using the global memory</td>
<td></td>
</tr>
<tr>
<td>32, 32</td>
<td>1.416</td>
</tr>
<tr>
<td>MetaFork</td>
<td></td>
</tr>
<tr>
<td>16, 16</td>
<td>1.274</td>
</tr>
<tr>
<td>32, 32</td>
<td>1.967</td>
</tr>
<tr>
<td>64, 64</td>
<td>\textbf{2.122}</td>
</tr>
<tr>
<td>128, 128</td>
<td>1.787</td>
</tr>
<tr>
<td>256, 256</td>
<td>1.789</td>
</tr>
<tr>
<td>512, 512</td>
<td>2.193</td>
</tr>
</tbody>
</table>
memory. PPCG by default chooses a thread block format of $16 \times 32$, while MetaFork preferred format varies based on input size.

Table 4.3: Speedup comparison of 2D Jacobi between PPCG and MetaFork kernel code

<table>
<thead>
<tr>
<th>Speedup (kernel)</th>
<th>Input size</th>
<th>PPCG</th>
<th>MetaFork</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$(2^{12} + 2)^2$</td>
<td>$(2^{13} + 2)^2$</td>
<td>$(2^{14} + 2)^2$</td>
</tr>
<tr>
<td>Thread block size</td>
<td>$16 \times 32$</td>
<td>$11.230$</td>
<td>$11.303$</td>
</tr>
<tr>
<td>$8 \times 4$</td>
<td>5.000</td>
<td>5.256</td>
<td>4.666</td>
</tr>
<tr>
<td>$16 \times 4$</td>
<td>7.867</td>
<td>8.724</td>
<td>7.962</td>
</tr>
<tr>
<td>$32 \times 4$</td>
<td>11.607</td>
<td>11.143</td>
<td>9.726</td>
</tr>
<tr>
<td>$8 \times 8$</td>
<td>7.209</td>
<td>7.776</td>
<td>6.704</td>
</tr>
<tr>
<td>$16 \times 8$</td>
<td>10.499</td>
<td>10.502</td>
<td>7.442</td>
</tr>
<tr>
<td>$32 \times 8$</td>
<td>12.236</td>
<td>11.487</td>
<td>9.182</td>
</tr>
<tr>
<td>$8 \times 16$</td>
<td>8.859</td>
<td>8.825</td>
<td>5.637</td>
</tr>
<tr>
<td>$16 \times 16$</td>
<td>10.774</td>
<td>10.709</td>
<td>7.694</td>
</tr>
<tr>
<td>$32 \times 16$</td>
<td>11.969</td>
<td>11.442</td>
<td>10.469</td>
</tr>
</tbody>
</table>

**LU decomposition.** MetaFork and PPCG both generate two CUDA kernels: one with a 1D grid and one with a 2D grid, both using the shared memory. The default selected thread block formats for PPCG are $32$ and $16 \times 32$; meanwhile, the preferred formats by MetaFork are $128$ and $16 \times 16$. Tuning the number of threads per thread block in our parametric code allows MetaFork to outperform PPCG.

**Matrix transpose.** Both the CUDA kernels generated by MetaFork and PPCG use a 2D grid and the shared memory. PPCG by default chooses a thread block format of $16 \times 32$, while MetaFork preferred format is $8 \times 32$. For MetaFork, the allocation unit size of shared memory for the input matrix is the same as thread block format. However, for PPCG, the allocation unit size of shared memory for the input matrix is $32 \times 32$, while the thread block format is $16 \times 32$. Thus, PPCG code transposes two coefficients of the matrix within each thread.

**Matrix addition.** Both the CUDA kernels generated by MetaFork and PPCG use a 2D grid and the global memory. The default chosen thread block format for PPCG is $16 \times 32$, while MetaFork preferred format is $32 \times 8$.

**Matrix vector multiplication.** For both MetaFork and PPCG, the generated kernels use a 1D grid and the shared memory. The thread block size chosen by PPCG is $32$. Table 4.7 shows the speedup factors obtained with the kernels generated by PPCG and MetaFork with post-processing, respectively. One can see that the performance of the parametric kernel with coalesced accesses is twice as good as that of the automatically generated kernels by MetaFork and PPCG.

**Matrix matrix multiplication.** For both MetaFork and PPCG, the generated kernels use a 2D grid and the shared memory. The thread block size chosen by PPCG is $16 \times 32$, while MetaFork preferred thread block size varies based on input sizes. For MetaFork, the allocation unit size of shared memory for each input matrix is the same as thread block format. However, for PPCG, the allocation unit size of shared memory for each input matrix is $32 \times 32$, respectively.
while the thread block format is 16 × 32. In fact, PPCG code computes two coefficients of the output matrix within each thread, thus increasing index arithmetic amortization and occupancy.

We conclude this section with timings (in seconds) for the quantifier elimination (QE) required to generate MetaFork tiled code, see Table 4.9. Our tests are based on the latest version of the RegularChains library of Maple, available at www.regularchains.org. These results show that the use of QE is not a bottleneck in our C-to-CUDA code translation process, despite the theoretically high algebraic complexity of quantifier elimination.

4.6 Summary

In this chapter, we have presented enhancements of the MetaFork language so as to support device constructs, in particular, the SIMD paradigm. For the latter, our objective is to facilitate automatic code translation from high-level programming models supporting hardware accel-
Chapter 4. Applying MetaFork to the Generation of Parametric CUDA Kernels

Table 4.4: Speedup comparison of LU decomposition between PPCG and MetaFork kernel code

<table>
<thead>
<tr>
<th>Speedup (kernel)</th>
<th>Input size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread block size</td>
<td>2^{10} * 2^{10}</td>
</tr>
<tr>
<td>PPCG</td>
<td></td>
</tr>
<tr>
<td>32, 16 * 32</td>
<td>10.712</td>
</tr>
<tr>
<td>MetaFork</td>
<td></td>
</tr>
<tr>
<td>128, 4 * 4</td>
<td>3.063</td>
</tr>
<tr>
<td>256, 4 * 4</td>
<td>3.077</td>
</tr>
<tr>
<td>512, 4 * 4</td>
<td>3.095</td>
</tr>
<tr>
<td>32, 8 * 8</td>
<td>10.721</td>
</tr>
<tr>
<td>64, 8 * 8</td>
<td>10.604</td>
</tr>
<tr>
<td>128, 8 * 8</td>
<td>10.463</td>
</tr>
<tr>
<td>256, 8 * 8</td>
<td>10.831</td>
</tr>
<tr>
<td>512, 8 * 8</td>
<td>10.416</td>
</tr>
<tr>
<td>32, 16 * 16</td>
<td>14.533</td>
</tr>
<tr>
<td>64, 16 * 16</td>
<td>14.457</td>
</tr>
<tr>
<td>128, 16 * 16</td>
<td>14.877</td>
</tr>
<tr>
<td>256, 16 * 16</td>
<td>14.803</td>
</tr>
<tr>
<td>512, 16 * 16</td>
<td>14.479</td>
</tr>
</tbody>
</table>

Table 4.5: Speedup comparison of matrix transpose between PPCG and MetaFork kernel code

<table>
<thead>
<tr>
<th>Speedup (kernel)</th>
<th>Input size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread block size</td>
<td>2^{13} * 2^{13}</td>
</tr>
<tr>
<td>PPCG</td>
<td></td>
</tr>
<tr>
<td>16 * 32</td>
<td>62.656</td>
</tr>
<tr>
<td>MetaFork</td>
<td></td>
</tr>
<tr>
<td>8 * 4</td>
<td>28.626</td>
</tr>
<tr>
<td>16 * 4</td>
<td>40.381</td>
</tr>
<tr>
<td>32 * 4</td>
<td>28.728</td>
</tr>
<tr>
<td>8 * 8</td>
<td>51.889</td>
</tr>
<tr>
<td>16 * 8</td>
<td>44.759</td>
</tr>
<tr>
<td>32 * 8</td>
<td>37.586</td>
</tr>
<tr>
<td>8 * 16</td>
<td>70.716</td>
</tr>
<tr>
<td>16 * 16</td>
<td>64.812</td>
</tr>
<tr>
<td>32 * 16</td>
<td>36.109</td>
</tr>
<tr>
<td>8 * 32</td>
<td><strong>77.327</strong></td>
</tr>
<tr>
<td>16 * 32</td>
<td>62.268</td>
</tr>
</tbody>
</table>

(like OpenMP and OpenACC) to low-level heterogeneous programming models (like CUDA). As illustrated in Section 4.3, MetaFork has language constructs to help generating efficient CUDA code. Moreover, the MetaFork framework relies on advanced techniques (quantifier elimination in non-linear polynomial expressions) for code optimization, in particular tiling.
4.6. Summary

Serial code

for (int k = 0; k < n; ++k) {
    for (int i = 0; i < n-k-1; i++) {
        // column major representation
        // of L and U
        int p = i + k + 1;
        L[k][p] = U[k][p] / U[k][k];
        for (int j = k; j < n; j++)
            U[j][p] -= L[k][p] * U[j][k];
    }
}

MetaFork code

int ub = n / B, ut = n / Sqrt_T;
meta_schedule {
    for (int k = 0; k < n-1; k++) {
        meta_for (int bx = 0; bx < ub; bx++)
            for (int ux = 0; ux < B; ux++)
                if ((k + 1 - bx * B < B) &&
                    (-B * bx + k < ux) &&
                    (ux < n - B * B)) {
                    int l = bx * B + ux;
                    L[k][l] = U[k][l] / U[k][k];
                }
    }
    for (int bx = 0; bx < ut; bx++)
        for (int by = 0; by < ut; by++)
            for (int ux = 0; ux < Sqrt_T;
                ux++)
                if (t0 == 0)
                    shared_U_1[0][0] = U[c0 * n + c0];
                __syncthreads();
                for (int c1 = b0; c1 < ub; c1 += 32768) {
                    if (((((c0) + 1) - ((c1) * (B))) < (B)) &&
                        (((B) * (c1)) + (c0)) < (t0))
                        &&
                        ((t0) < ((n) - ((c1) * (B))))
                        {
                            private_l = ((c1) * (B)) + (t0);
                            L[c0 * n + private_l] =
                                (U[c0 * n + private_l] /
                                    shared_U_1[0][0]);
                        }
                        __syncthreads();
                }
            }
}

__global__ void kernel0(double *L, double *U, int n, int ut,
                        int Sqrt_T, int ub, int B, int c0) {
    int b0 = blockIdx.x;
    int t0 = threadIdx.x;
    int private_l;
    __shared__ double shared_U_1[1][1];
    {
        if (t0 == 0)
            shared_U_1[0][0] = U[c0 * n + c0];
        __syncthreads();
        for (int c1 = b0; c1 < ub; c1 += 32768) {
            if (((((c0) + 1) - ((c1) * (B))) < (B)) &&
                (((B) * (c1)) + (c0)) < (t0))
                &&
                ((t0) < ((n) - ((c1) * (B))))
                {
                    private_l = ((c1) * (B)) + (t0);
                    L[c0 * n + private_l] =
                        (U[c0 * n + private_l] /
                            shared_U_1[0][0]);
                }
                __syncthreads();
        }
    }
}

__global__ void kernel1(double *L, double *U, int n, int ut,
                        int Sqrt_T, int ub, int B, int c0) {
    int b0 = blockIdx.y, b1 = blockIdx.x;
    int t0 = threadIdx.y, t1 = threadIdx.x;
    int private_i;
    int private_j;
    // BLOCK_0 = BLOCK_1 = Sqrt_T
    __shared__ double shared_L[1][BLOCK_0];
    __shared__ double shared_U_1[BLOCK_0][1];
    for (int c1 = b0; c1 < ut; c1 += 256) {
        if (t1 == 0 && n >= t0 + c0 + Sqrt_T * c1 + 1)
            shared_U_1[t0][0] = U[(t0 + c0 + Sqrt_T * c1) * n + c0];
        for (int c2 = b1; c2 < ut; c2 += 256) {
            if (t0 == 0 && n >= t1 + c0 + Sqrt_T * c2 + 2)
                shared_L[0][t1] =
                    L[c0 * n + (t1 + c0 + Sqrt_T * c2 + 2)];
            __syncthreads();
            private_i = (((c2) * (Sqrt_T)) + (t1));
            if (private_i < ((n) - (c0) - 1)) {
                private_j = ((c1) * (Sqrt_T)) + (t0);
                if (private_j < ((n) - (c0))) {
                    U[(private_j + c0) * n + (private_i + c0 + 1)] =
                        (shared_L[0][private_i - Sqrt_T * c2] *
                            shared_U_1[private_i - Sqrt_T * c2][0];
                }
                __syncthreads();
            }
        }
    }
}

Figure 4.11: Serial code, MetaFork code and generated parametric CUDA kernel for LU decomposition

The experimentation reported in Section 4.5 shows the benefits of generating parametric CUDA kernels. Not only this feature provides more portability but it helps obtaining better performance with automatically generated code.
Chapter 4. Applying MetaFork to the Generation of Parametric CUDA Kernels

Serial code

for (int v0 = 0; v0 < n; v0++)
for (int v1 = 0; v1 < n; v1++)
c[v0][v1] = a[v1][v0];

MetaFork code

int dim0 = n / B0, dim1 = n / B1;
meta_schedule {
    meta_for (int v0 = 0; v0 < dim0; v0++)
    meta_for (int v1 = 0; v1 < dim1; v1++)
    meta_for (int u0 = 0; u0 < B0; u0++)
    meta_for (int u1 = 0; u1 < B1; u1++)
    {
        int i = u0 + v0 * B0;
        int j = u1 + v1 * B1;
        c[j][i] = a[i][j];
    }
}

__global__ void kernel0(int *a, int *c, int n, int dim0, int dim1, int B0, int B1) {
    int b0 = blockIdx.y, b1 = blockIdx.x;
    int t0 = threadIdx.y, t1 = threadIdx.x;
    int private_i;
    int private_j;
    __shared__ int shared_a[BLOCK_0][BLOCK_1];
    for (int c0 = b0; c0 < dim0; c0 += 256)
        for (int c1 = b1; c1 < dim1; c1 += 256) {
            if (n >= t0 + B0 * c0 + 1 && n >= t1 + B1 * c1 + 1)
                shared_a[t0][t1] = a[(t0 + B0 * c0) * n + (t1 + B1 * c1)];
        __syncthreads();
        private_i = ((t0) + ((c0) * (B0)));
        private_j = ((t1) + ((c1) * (B1)));
        c[private_j * n + private_i] =
            shared_a[private_j - B0 * c0][private_j - B1 * c1];
        __syncthreads();
    }
}

Figure 4.12: Serial code, MetaFork code and generated parametric CUDA kernel for matrix transpose

Table 4.6: Speedup comparison of matrix addition between PPCG and MetaFork kernel code

<table>
<thead>
<tr>
<th>Speedup (kernel)</th>
<th>Input size</th>
<th>Thread block size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>2&lt;sup&gt;12&lt;/sup&gt;</td>
</tr>
<tr>
<td>PPCG</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16 * 32</td>
<td>13.024</td>
<td>9.750</td>
</tr>
<tr>
<td>MetaFork</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 * 4</td>
<td>19.520</td>
<td>20.329</td>
</tr>
<tr>
<td>16 * 4</td>
<td>32.971</td>
<td>35.227</td>
</tr>
<tr>
<td>32 * 4</td>
<td>54.233</td>
<td>49.734</td>
</tr>
<tr>
<td>8 * 8</td>
<td>28.186</td>
<td>30.221</td>
</tr>
<tr>
<td>16 * 8</td>
<td>44.783</td>
<td>42.008</td>
</tr>
<tr>
<td>32 * 8</td>
<td>&lt;b&gt;56.650&lt;/b&gt;</td>
<td>&lt;b&gt;50.547&lt;/b&gt;</td>
</tr>
<tr>
<td>8 * 16</td>
<td>33.936</td>
<td>32.793</td>
</tr>
<tr>
<td>16 * 16</td>
<td>45.015</td>
<td>41.606</td>
</tr>
<tr>
<td>32 * 16</td>
<td>54.426</td>
<td>47.930</td>
</tr>
</tbody>
</table>

Table 4.7: Speedup comparison of matrix vector multiplication among PPCG kernel code, MetaFork kernel code and MetaFork kernel code with post-processing

<table>
<thead>
<tr>
<th>Speedup (kernel)</th>
<th>Input size</th>
<th>Thread block size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>2&lt;sup&gt;11&lt;/sup&gt;</td>
</tr>
<tr>
<td>PPCG</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>3.954</td>
<td>3.977</td>
</tr>
<tr>
<td>MetaFork with post-processing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>4.976</td>
<td>6.260</td>
</tr>
<tr>
<td>32</td>
<td>&lt;b&gt;8.698&lt;/b&gt;</td>
<td>&lt;b&gt;6.911&lt;/b&gt;</td>
</tr>
<tr>
<td>64</td>
<td>4.260</td>
<td>5.567</td>
</tr>
</tbody>
</table>
4.6. Summary

Serial code

```c
for (int v0 = 0; v0 < n; v0++)
    for (int v1 = 0; v1 < n; v1++)
        c[v0][v1] = a[v0][v1] + b[v0][v1];
```

MetaFork code

```c
int dim0 = n / B0, dim1 = n / B1;

meta_schedule {
    meta_for (int v0 = 0; v0 < dim0; v0++)
        meta_for (int v1 = 0; v1 < dim1; v1++)
            meta_for (int u0 = 0; u0 < B0; u0++)
                meta_for (int u1 = 0; u1 < B1; u1++)
                    { int i = u0 + v0 * B0;
                      int j = u1 + v1 * B1;
                      c[i][j] = a[i][j] + b[i][j];
                    }
}
```

Figure 4.13: Serial code, MetaFork code and generated parametric CUDA kernel for matrix addition

Serial code

```c
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        c[i] += a[i][j] * b[j];
```

MetaFork code

```c
int dim = n / B;

meta_schedule {
    meta_for (int v = 0; v < dim; v++)
        for (int i = 0; i < n / 16; ++i)
            for (int j = 0; j < n; ++j) {
                int p = v * B + u;
                c[p] += a[p][i*16+j]*b[i*16+j];
            }
}
```

Figure 4.14: Serial code, MetaFork code and generated parametric CUDA kernel for matrix vector multiplication

```c
__global__ void kernel10(int *a, int *b, int *c, int n, int dim0, int dim1, int B0, int B1) {
    int b0 = blockIdx.x, b1 = blockIdx.y;
    int t0 = threadIdx.x, t1 = threadIdx.y;
    int private_i;
    int private_j;

    for (int c0 = b0; c0 < dim0; c0 += 256)
        for (int c1 = b1; c1 < dim1; c1 += 256) {
            private_i = ((t0) + ((c0) * (B0)));
            private_j = ((t1) + ((c1) * (B1)));
            c[private_i * n + private_j] =
                (a[private_i * n + private_j] +
                b[private_i * n + private_j]);
            __syncthreads();
        }
}
```

```c
__global__ void kernel0(int *a, int *b, int *c, int n, int dim0, int dim1, int B0, int B1) {
    int b0 = blockIdx.x, b1 = blockIdx.y;
    int t0 = threadIdx.x, t1 = threadIdx.y;
    int private_i;
    int private_j;

    for (int c0 = b0; c0 < dim0; c0 += 256)
        for (int c1 = b1; c1 < dim1; c1 += 256) {
            private_i = ((t0) + ((c0) * (B0)));
            private_j = ((t1) + ((c1) * (B1)));
            c[private_i * n + private_j] =
                (a[private_i * n + private_j] +
                b[private_i * n + private_j]);
            __syncthreads();
        }
}
```

```c
__global__ void kernel10(int *a, int *b, int *c, int n, int dim, int B) {
    int b0 = blockIdx.x;
    int t0 = threadIdx.x;
    int private_p;
    // BLOCK_0 = B
    __shared__ int shared_a[BLOCK_0][BLOCK_0];
    __shared__ int shared_b[BLOCK_0];
    __shared__ int shared_c[BLOCK_0];

    for (int c0 = b0; c0 < dim; c0 += 32768) {
        if (n >= t0 + B * c0 + 1)
            shared_c[t0] = c[t0 + B * c0];
        for (int c1 = 0; c1 < n / BLOCK_0; c1 += 1) {
            if (n >= t0 + B * c0 + 1)
                for (int c3 = 0; c3 < BLOCK_0; c3 += 1)
                    shared_a[c3][t0] = a[(c3 + B * c0) * n + (B * c1 + t0)];
                shared_b[t0] = b[t0 + B * c1];
            __syncthreads();
        }
        __syncthreads();
    }
    if (n >= t0 + B * c0 + 1)
        c[t0 + B * c0] = shared_c[t0];
    __syncthreads();
}
```
Table 4.8: Speedup comparison of matrix multiplication between PPCG and MetaFork kernel code

<table>
<thead>
<tr>
<th>Speedup (kernel)</th>
<th>Input size</th>
<th>PPCG</th>
<th>MetaFork</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>129.853</td>
<td>393.851</td>
</tr>
<tr>
<td></td>
<td></td>
<td>54.578</td>
<td>171.621</td>
</tr>
<tr>
<td></td>
<td></td>
<td>60.358</td>
<td>182.557</td>
</tr>
<tr>
<td></td>
<td></td>
<td>87.919</td>
<td>287.002</td>
</tr>
<tr>
<td></td>
<td></td>
<td>84.057</td>
<td>289.930</td>
</tr>
<tr>
<td></td>
<td></td>
<td>100.521</td>
<td>299.228</td>
</tr>
<tr>
<td></td>
<td></td>
<td>100.264</td>
<td>330.965</td>
</tr>
<tr>
<td></td>
<td></td>
<td>85.928</td>
<td>247.220</td>
</tr>
</tbody>
</table>

Serial code

```c
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        for (int k = 0; k < n; ++k)
            c[i][j] += a[i][k] * b[k][j];
```

MetaFork code

```c
int dim0 = n / B0, dim1 = n / B1;
meta_schedule {
    meta_for (int i = 0; i < dim0; i++)
        meta_for (int j = 0; j < dim1; j++)
            meta_for (int k = 0; k < n/4; k++)
                meta_for (int v = 0; v < B0; v++)
                    meta_for (int u = 0; u < B1; u++)
                        {
                            int p = i * B0 + v;
                            int w = j * B1 + u;
                            for (int z = 0; z < 4; z++)
                                c[p][w] += a[p][4*k+z] * b[4*k+z][w];
                        }
        }
    }
```

**Figure 4.15:** Serial code, MetaFork code and generated parametric CUDA kernel for matrix matrix multiplication
Table 4.9: Timings (in sec.) of quantifier elimination for eight examples

<table>
<thead>
<tr>
<th>Example</th>
<th>Timing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array reversal</td>
<td>0.072</td>
</tr>
<tr>
<td>1D Jacobi</td>
<td>0.948</td>
</tr>
<tr>
<td>2D Jacobi</td>
<td>7.735</td>
</tr>
<tr>
<td>LU decomposition</td>
<td>4.416</td>
</tr>
<tr>
<td>matrix transposition</td>
<td>1.314</td>
</tr>
<tr>
<td>matrix addition</td>
<td>1.314</td>
</tr>
<tr>
<td>matrix vector multiplication</td>
<td>0.072</td>
</tr>
<tr>
<td>matrix matrix multiplication</td>
<td>2.849</td>
</tr>
</tbody>
</table>
Chapter 5

MetaFork: A Metalanguage for Concurrency Platforms Targeting Pipelining

On-line algorithms [10, 37] which are suitable for applications, like signal and image processing, often involve data parallelism. In such algorithms, computations start as soon as part of the input data is available for processing, and results are returned as soon as part of the output data becomes available. Pipelining [87, 61, 39] has a basic pattern of on-line algorithms. A pipelining is a sequence of processing stages operating on tasks by partitioning them into separated units. Stages work independently in the producer-consumer manner.

Pipelining is poorly suited for data parallelism. Indeed, in this model, processing elements perform simultaneously the same operation on different data regions. Pipelining is also poorly suited for the fork-join model. Indeed, in case that processing a subset of elements is distributed to each task, synchronization must take place whenever a processing element needs to be read or written. With such synchronization in the fork-join model indicated by the implicit or explicit barrier operation, this could lead to severe parallelism overheads.

In addition, stencil computations [47, 83] are a major pattern in scientific computing. Stencil codes perform a sequence of sweeps (called time-steps) through a given array and each sweep can be seen as the execution of a pipelining. When expressed with concurrency platforms based (and limited) to the fork-join model, parallel stencil computations incur excessive parallelism overheads. This problem is studied in [134] together with a solution in the context of OpenMP by proposing new synchronization constructs to enable doacross parallelism. In Section 5.1 we briefly explain how the pipelining techniques are implemented in the real life. The syntax and the semantics of MetaFork pipelining parallel constructs are specified in Sections 5.2 and 5.3.

5.1 Execution Model of Pipelining

As mentioned above, a pipelining is a sequence of processing stages through which data items flow from the first stage to the last stage. If each stage can process only one data item at a time, then the pipelining is said to be serial and can be depicted by a (directed) path in the sense of...
5.2. Core Parallel Constructs

We enhance the MetaFork language with three constructs: meta_pipe, meta_wait and meta_continue to express pipelining parallelism.

The meta_pipe construct. The synopsis of meta_pipe is

\[
\text{meta\_pipe}([I;]C[:S][:D])\{B\}
\]

where \(I\) is the initialization expression, \(C\) is the condition expression, \(S\) is the stride, \(D\) is the dependencies, and \(B\) is the pipelining body. In addition:

- the initialization expression initializes variables, called the control variables, which can be of an integer or pointer type,
- the condition expression compares the control variables with a compatible expression, using one of the relational operators \(<, \leq, >, \geq, \neq\),
- the stride is used to increase or decrease the value of the control variables,
- the dependencies are used to specify the execution order of the data flow, and
- if \(B\) consists of a single instruction, then the surrounding curly braces can be omitted.

The above meta_pipe statement specifies that its serial counterpart induces pipelining, that is, a DAG whose vertices are processing stages through which data items flow from the first stage to the last stage. Unless dependencies are specified or meta_continue is used, the execution of the loop is sequential, that is, the DAG is linear. If dependencies are used, then they specify the immediate predecessors of each vertex of the DAG.

Each time before the execution of the pipelining body, the condition expression is evaluated. If it is true, the body is executed; otherwise, the body terminates. The execution of each iteration of the meta_pipe loop starts from stage zero in the serial manner and the pipelining
body consists of several stages whose boundaries are delimited by the meta_wait or meta_continue construct. We refer to the stages in each iteration as monotonically increasing non-negative integers as the iteration executes. The dependencies are a set of dependency which is defined as the form of \((d_1, d_2, ..., d_n)\), where \(n\) is the nest-level and \(d_i\) denotes the loop index of the \(i_{th}\) nested loop in the meta_pipe C-elson counterpart (see Section 5.3).

Using meta_wait and meta_continue into a meta_pipe body turns each iteration of the loop into a sub-DAG, as discussed in the following.

**The meta_continue construct.** meta_continue has the following format

\[
\text{meta\_continue}(s)
\]

meta_continue indicates that the execution advances to stage \(s\) without waiting for the stage \(s\) in the previous iteration of the loop to terminate. This implies that stage \(s\) could process more than one computation at a time in parallel.

**The meta_wait construct.** Analogous to the meta_continue construct, meta_wait is defined as follows:

\[
\text{meta\_wait}(s)
\]

meta_wait also indicates that the execution advances to stage \(s\), but waits for the completion of stage \(s\) in the previous iteration of the loop.

Specifically, unlike traditional pipelining technique that a stream of data flows through the pipelining at every stage, with our framework, programmers could skip some stages by taking advantage of the stage argument provided for the meta_continue and meta_wait constructs, and even the execution order between stages may be deduced dynamically. This intelligent design has the potential of exploiting complicated applications. In addition, if the argument of meta_wait and meta_continue is not explicitly specified, the execution implicitly flows into the next stage.

```plaintext
//for-loop or while-loop
meta_pipe(cond) {
  Read data;
  meta_continue();
  Processing data;
  meta_wait();
  Store data;
}
```

Figure 5.1 illustrates how to use these constructs on a simple example (given as pseudo-code) for which the corresponding execution DAG is also depicted. On the left side of Figure 5.1, the code is divided into three stages: reading data, processing data and storing data.
5.3 Semantics of the Pipelining Constructs in MetaFork

The meta_pipe indicates that the subsequent body represents a pipelining. Then, meta_continue advances work to the next stage without waiting for the completion of the previous iteration. At last, meta_wait moves to the next stage, but waits for the previous iteration to finish. As shown on the right side of Figure 5.1, stage 1 provides an opportunity for concurrent execution.

The semantics of each of the parallel constructs introduced in MetaFork pipelining model, is defined in this section, following the concept of serial C-elision discussed in Section 3.4.

The meta_pipe construct has the same semantics as the sequential execution of nested for-loops where the $i_{th}$ for-loop is denoted by the $i_{th}$ control variable appearing in the condition expression. On the other hand, in the C-elision code, the dependencies are safe to be removed without changing the semantics. Finally, meta_continue as well as meta_wait is converted into a goto statement that points to the position of the code according to its stage argument. In particular, if the argument indicates the execution from stage $s$ to stage $s + 1$, meta_continue and meta_wait are removed directly.

Figures 5.2 and 5.4 give insight into the above ideas. On the left hand side of Figure 5.2 a valid MetaFork code follows the pipelining pattern defined in Section 5.2. One the right hand of Figure 5.2, the serial counterpart of that MetaFork pipelining code is displayed by removing meta_wait, meta_continue and converting meta_pipe to a single for loop. It is obvious to use the DAG comparing the performance between the pipelining code and its serial code. On the right side of Figure 5.3, the nodes will be executed sequentially without any parallelism, while on the left side, stage 2 could run in parallel, which exposes massive parallelism.

Figure 5.4 shows a case of stencil computation code (on the left) written with meta_pipe. In this example, dependencies are specified by listing the adjacent vertices (i.e. iteration) on the vertex $(t,i,j)$. The serial counterpart of that MetaFork pipelining code is shown on the right of Figure 5.4. Note that dependencies are ignored and meta_pipe is replaced with nested for-loops.
5.4 Summary

In this chapter, we have enhanced the MetaFork framework with extensions to express pipelining parallelism. To this end, we introduced three constructs, i.e. meta_pipe, meta_continue and meta_wait, whose syntax and semantics were detailed in Sections 5.2 and 5.3. With these high-level constructs, programmers can specify a pipelining program with little programming efforts. Furthermore, compared to the traditional pipelining technique, our methods could model applications whose execution DAG is determined during the program’s execution. So far, we have focused on the syntax and semantics design, but the implementation is not yet completed.
Chapter 6

METAORK: The Compilation Framework

The objective of this chapter is to explain the design and the implementation details of the METAORK compilation framework. The key concepts of this chapter are arranged into sections and are presented as follows. In Section 6.1 we present the motivation of the METAORK compilation framework and how we will achieve our goal. In Section 6.2 we discuss the programming specification of METAORK as a high-level parallel programming language. In Section 6.3 we provide a high-level view of each software component of the METAORK compilation framework. In Sections 6.4 and 6.5 we present the key contributions of this chapter, which are the implementation of each component of the METAORK compiler, and the details of the extensions of the METAORK front-end, respectively. Finally, in Section 6.6 we detail the implementation of parsing METAORK and CilkPlus constructs.

6.1 Goals

Over the last two decades, there has been significant progress in the development of promising parallel platforms for high-performance computing targeting homogeneous and heterogeneous architectures. CilkPlus, OpenMP and CUDA are examples of these parallel platforms. Applications that run on these platforms have the potential to gain a huge boost in performance; however, they incur the cost of a significant programming effort, even for expert programmers. The challenges of parallel computing arise from the following well-known traits: large amount of concurrent threads, hierarchical memories, inter-processor communication, and synchronization. In order to take full advantage of the tremendous computing power made available through hardware accelerators, intricate knowledge of the parallel software platform and the underlying system hardware is mandatory. Additionally, due to the rapid architectural innovation and the increasing software complexity [83 [110 [46], it is not easy for developers to parallelize and optimize their programs manually. Therefore, in order to maximize performance, productivity, and portability, high-level parallel programming models that transparently adapt to a wide range of parallel architectures are needed.

In this context, we propose the METAORK compilation framework which is characterized by three significant goals:

1. offering high-level parallel programming models with language-level constructs aiming
to exploit parallelism for homogeneous and heterogeneous architectures in a simple manner as well as,

2. offering productive platforms for researchers to develop new compiler techniques to improve software in terms of performance and resource utilization, and

3. facilitating interoperability between concurrency platforms so as to realize the merits of the MetaFork compilation framework.

The first goal is driven by the observed advantages of high-level parallel programming models. Based on past experiences [130, 123, 48], parallel programming models with low-level APIs, like CUDA, offer a great opportunity for performance tuning but limit portability, scalability, and productivity. For instance, the burden of explicitly managing memory hierarchy (i.e. shared memory, global memory) and multi-level parallelism (e.g. thread blocks within a grid, threads within a thread block) placed by CUDA on programmers makes the manual development of high performance CUDA code rather complicated. Programmers are required to put much time and efforts to optimize those details.

The development of high-level parallel programming models which provide powerful abstraction, has the potential to free the programmers from the burden of handling program details. Such models abstract details of how a computation will actually be implemented. The programmers only need to supply annotations which capture the nature of the computation in the sequential code and the final decision of how to implement the computation rests with the back-end compiler by means of those annotations. A major feature of these models is to abstract away from the notion of a thread in the program as well as to make the underlying hardware features transparent to programmers. For example, these models avoid the explicit management of the hardware accelerator, i.e. thread and memory management, and the mapping of work units to threads; thus these models ease programming.

Moreover, programming with such models also leads to more portability for modern and future architectures due to the high abstraction which hides the complex details of the parallel hardware from the programmers. Consider this situation: sometimes implicit assumptions are made when programs are developed in low-level programming languages. It is possible that such programs sustain losses in performance if migrated to an architecture for which those assumptions no longer hold. For instance, hardware resources, like registers, have a crucial impact on the performance of CUDA kernels while the maximum number of registers available to each thread differs between GPU generations. Excessive usage of registers per thread, which causes register spilling, results in substantial performance degradation. By contrast, achieving high performance for an application written by high-level programming languages becomes the responsibility of the underlying compiler by performing optimization which should be focused on by researchers.

Hence, it is convincing that, instead of programming with low-level parallel programming APIs at the expense of a huge investment of time and efforts, programming with high-level languages with only minimal efforts, is the most promising technology to increase productivity and face radical changes in hardware design in the future [155]. On the other hand, observing the high cost of designing architecture specific back-end optimization from scratch, the MetaFork compilation framework is designed to make use of existing optimizing compilers.
In order to carry out such work, source-level compiler infrastructures must be available to perform the ability of interoperability between different concurrency platforms. We stress the fact that our work focuses on optimizing the use of user defined high-level (source code level) abstraction rather than on lower-level (machine instruction level) optimization associated with code generation for specific platforms.

We propose the following tasks so as to meet the goal of the MetaFork compilation framework:

1. defining a high-level language for expressing concurrency,
2. providing source-to-source transformation tools to take advantage of contemporary concurrency platforms, and
3. developing platforms which are capable of analyzing multiple programming languages.

At the time of writing, CilkPlus, OpenMP and the MetaFork language are supported by the MetaFork compilation framework.

6.2 MetaFork as a High-Level Parallel Programming Language

As described in Section 6.1, it is desirable for the MetaFork compilation framework to be built around a high-level parallel programming language. Such a language needs to satisfy the following requirements:

1. enhancing the C/C++ language with minimal language extensions while supporting popular schemes of parallelism,
2. increasing programming productivity.
3. programmers need to explicitly identify opportunities for concurrent execution.

The MetaFork language extends the C/C++ language with parallel language constructs that the programmer can express in the form of compiler-directives (i.e. Pragma’s) or keywords, see Section 6.6.1. Hereafter, we focus on Pragma directives, which are widely supported by mainstream compilers. MetaFork treats these Pragma directives in the host code as a set of declarative annotations that guide compilers towards carrying out the concrete code generation. The design of MetaFork’s Pragma directives is inspired by those of OpenMP, as illustrated by Figure 6.1 which specifies syntax.

```
#pragma mf directive-name [clause[ [,] clause] ... ] new-line
statement
```

Figure 6.1: MetaFork Pragma directive syntax

Basically, the syntax of MetaFork Pragma directives follows the convention of C/C++ standards for compiler directives. Directives are case-sensitive and each directive starts with

---

1The C Preprocessor [https://gcc.gnu.org/onlinedocs/cpp/Pragmas.html#Pragmas](https://gcc.gnu.org/onlinedocs/cpp/Pragmas.html#Pragmas)
#pragma mf to represent itself as a MetaForkPragma directive. For some directive, a clause which contains a list of variables may be used. It is the programmer’s responsibility to ensure that each variable is valid, which implies that not only it is a valid C/C++ identifier, but also that the variable has been declared before being used. The clauses can appear in any order in a directive and also the clauses in a directive can be repeated if needed, see Figure 6.2. A Pragma directive can be attached to any statement within a function body and must be terminated by a new-line character.

```c
1 void region(int *a, int *b, int N)
2 {
3   int sum_a=0, sum_b=0;
4
5   #pragma mf fork shared(sum_a) shared(a)
6   for(int i=0; i<N; i++)
7     sum_a += a[i];
8
9   for(int i=0; i<N; i++)
10      sum_b += b[i];
11
12   #pragma mf join
13 }
```

Figure 6.2: A code snippet showing a MetaFork program with Pragma directives

In Figure 6.2, the code snippet is used to illustrate the use of MetaForkPragma directives expressing parallelism at the source code level. The Pragma directive shown at Line (5) allows the for-loop statement at Line (6) to be executed concurrently to the for-loop statement at Line (9). Details on the meaning of each Pragma directive (e.g. at Lines (5) and (12)) are presented in Section 3.2.

```c
1 void region(int *a, int *b, int N)
2 {
3   int sum_a=0, sum_b=0;
4
5   meta_fork shared(sum_a) shared(a)
6   for(int i=0; i<N; i++)
7     sum_a += a[i];
8
9   for(int i=0; i<N; i++)
10      sum_b += b[i];
11
12   meta_join;
13 }
```

Figure 6.3: A code snippet showing a MetaFork program with keywords
6.3 Organization of the MetaFork Compilation Framework

In addition to the Pragma directive mechanism, MetaFork also features its language with keywords (similar to CilkPlus) as construct for parallelism. As an example, the programs annotated with Pragma directives in Figure 6.2 and that annotated with keywords in Figure 6.3 are semantically equivalent; but, unlike Pragma directives which are required to be placed in a standalone line, the keywords flavor is more flexible. MetaFork keywords could be located on the same line or on different lines with the statement that they precede. In fact, MetaFork keywords are turned into Pragma directives with macro definitions. Additional details are presented in Section 6.6.1.

6.3 Organization of the MetaFork Compilation Framework

The MetaFork compilation framework, as a source-to-source transformation compiler, takes programs written in high-level parallel programming languages (i.e. CilkPlus, MetaFork and OpenMP) as input and produces equivalent programs written in another parallel programming languages (i.e. CilkPlus, MetaFork, OpenMP and CUDA). Figure 6.4 depicts an overview of the design of the MetaFork compilation framework which highlights the MetaFork compiler as its core component from the perspective of a source-to-source transformation compiler.

The entire execution of input programs happens in two steps: at the beginning of the process, the MetaFork compiler translates the input programs into another format of parallel programs which are readable by users. Then the generated programs are passed straightly to a third-party compiler, such as GCC, CLANG and NVCC, which compiles the generated programs into an executable binary file shown as the last step of Figure 6.4.

6.4 MetaFork Compiler

The MetaFork compiler has three main components: (1) the front-end module, (2) the analysis & transformation module and (3) the back-end module. With the MetaFork compiler, input programs are converted into an abstract syntax tree (AST) by the first component. This AST is then used as a common internal representation (IR) format in the next phase, that is, by the second component, which performs numerous analyses and optimization. Afterward, the third component translates the IR obtained from the previous phase into output files. More than one type of input program is handled by the MetaFork compiler. So far, our research work has focused on applications and libraries written in C/C++ with language extensions (i.e.
In this section, we describe the design and the implementation of the MetaFork compiler, particularly, the front-end on which Section 6.4.1 gives a complete introduction. Following it, the other two modules are covered in Sections 6.4.2 and 6.4.3.

### 6.4.1 Front-End of the MetaFork Compiler

The purpose of the MetaFork compiler front-end is to parse the input programs and build the corresponding AST’s. To avoid the cumbersome implementation of a family of C-based languages front-ends and favor a widespread use of the MetaFork framework, we have taken advantage of the CLANG framework as a basis for developing the MetaFork front-end. CLANG is designed with clear interfaces and also considers itself extensible, so as to allow developers to customize their own tools to interact with CLANG. Recall the fact that MetaFork, as a high-level parallel programming language, usesPragma directives to expose parallelism. During the preprocessing phase, the parser of CLANG simply treatsPragma directives as sequences of lexical tokens so as to allow them to be handled by extensions. This feature of CLANG fits our MetaFork project well since it offers the opportunity to implement the MetaFork compiler as a standalone tool without applying any modifications to the core files or extending any C/C++ grammars of the CLANG framework. Moreover, the CLANG framework at version 3.6.2, on which MetaFork is built, includes a complete implementation of OpenMP 3.0 specification. It implies that we only need to focus on enhancing the CLANG front-end to parse thePragma directives of MetaFork and CilkPlus. Based on the above considerations, we chose to use LibTooling.

```cpp
1 int main(int argc, const char **argv) {
2   llvm::sys::PrintStackTraceOnErrorSignal();
3
4   std::unique_ptr<CompilationDatabase> Compilations(  
5     tooling::FixedCompilationDatabase::loadFromCommandLine(argc, argv));
6
7   cl::ParseCommandLineOptions(argc, argv);
8
9   tooling::RefactoringTool Tool(*Compilations, SourcePaths);
10  return
11      Tool.runAndSave(newFrontendActionFactory<OpenMPtoMetaFrontendAction>().get());
12 }
```

Figure 6.5: A code snippet showing how to create tools based on CLANG’s LibTooling

a CLANG library that allows users to develop standalone tools as well as to use CLANG’s parsing function. The interfaces among different tasks composed of building tools upon LibTooling are clearly separated and are detailed as follows:

1. **Compilation database setup.** To allow a compiler front-end to run properly, apart from the input programs, it is necessary to pass various kinds of flags that the input programs are compiled with, such as language-version flags (-std), macro definitions (-D), include
paths (-I), etc. In particular, without the include paths information, it’s not even possible to parse the input programs as a complete translation unit. CLANG figures out all these specific options by configuring a compilation database. A compilation database retrieves a collection of exact compilation options, that programs are compiled with. For instance, Line (4) of Figure 6.5 declares a compilation database variable and the ParseCommandLineOptions function at Line (7) will parse all options following the double dash "--" on the command line. For instance, the three options in Figure 6.6, including -include, -D, and -I, are stored in a compilation database.

2. ClangTool instantiation. In the second step, we need to create a ClangTool object which is the utility to run a FrontendAction discussed in step 3 over input programs. As illustrated at Line (9) of Figure 6.5, we use a specific version called RefactoringTool which is a subclass of ClangTool that offers a nice way to manipulate source-to-source transformations. Internally, such a class implements all the logic such that all involved components of a MetaFork source-to-source compiler can work in coordination with each other, such as parsing the input programs, performing an action when a match occurs (in our case, matching the Pragma directives), running the AST visitor and applying all program modifications. This explains why we end our main function in Figure 6.5 with a call to Tool.runAndSave(); in fact, it will perform all these tasks automatically. To create a Refactoring object, the compilation database variable declared in step 1 and the source programs, for instance, input.cpp in Figure 6.6, are passed as parameters to initialize its constructor.

3. FrontendAction instantiation. The FrontendAction class, which is a basic abstract class for several sub-classes, provides various interfaces for performing actions over input source programs. We use ASTFrontendAction which is a sub-class of FrontendAction because we want to analyze the AST representation of the source programs. Moreover, this is the place where we can interact with the CLANG preprocessor so as to handle ourPragma directives because building the AST cannot be done by bypassing the preprocessor. For example, at Line (12) of Figure 6.5 OpenMPToMetaFrontendAction is an instantiation of ASTFrontendAction. It serves as a parameter to the newFrontendActionFactory function which creates a new FrontendAction instance for a ClangTool object.

```bash
metatoopenmp input.cpp -o output.cpp -- -include header.h -Dmacro -I/path/
```

Figure 6.6: A general command-line interface of MetaFork tools

### 6.4.2 Analysis & Transformation of the MetaFork Compiler

Besides the purpose of serving as a high-level parallel programming language, the MetaFork compilation framework also provides tools for source-to-source transformations via the analysis performed on the AST. One of the basic functionalities required to navigate the AST is visiting each of its nodes in a predefined order. As a result, our intention is to modify and

Figure 6.7: Overall work-flow of the MetaFork analysis and transformation chain

Rewrite the AST. There are two methods to traverse the AST in C:\lang, by using Matchers or RecursiveASTVisitors. In our implementation, we use the latter which not only provides access to the nodes of the AST, which is what Matchers does, but also allows us to change the predefined order in which the AST nodes are traversed.

In this section, we present a general scheme of FrontEndAction operations which perform recursive visit and editing over the AST. To this end, programmers are responsible for manipulating four procedures, that are: ASTFrontEndAction, ASTConsumer, RecursiveASTVisitor and Rewriter, as depicted in Figure 6.7:

1. **ASTFrontEndAction.** As illustrated in Section 6.4.1, ASTFrontEndAction is a C:\lang interface which is used for writing tools that operate on the AST. In our implementation, all it does is to provide a method CreateASTConsumer() to create an ASTConsumer and set up a C:\lang Rewriter instance which is the main interface to the C:\lang rewrite buffers for code transformations.

2. **ASTConsumer.** ASTConsumer is an abstract interface that should be implemented by developers. This interface is the entrance to access the AST produced by the C:\lang parser. ASTConsumer provides several virtual methods that are called at various points when a certain type of an AST node has been parsed in the compilation process. For instance, HandleTranslationUnit, which opts in our project, is called only after C:\lang finishes building the AST for entire input programs. HandleTranslationUnit performs a traversal of the whole AST using RecursiveASTVisitor scheme.

3. **RecursiveASTVisitor.** The RecursiveASTVisitor which uses curiously recurring template pattern\(^3\) is the parent class of all our node visitors. It allows developers to visit any type of AST nodes, such as FunctionDecl and Stmt, simply by overriding a function with that name, e.g. VisitFunctionDecl and VisitStmt respectively. To be more precise, the RecursiveASTVisitor does a preorder depth-first traversal of the entire C:\lang AST. Three groups of methods are supplied to visit each node of the AST in a hierarchical manner. Traverse method group, as the first group, dispatches to a concrete Traverse method which matches the dynamic type of the AST node. From there the WalkUpFrom method, as the second group, is called which goes up the class hierarchy of the current node until the top-most class, e.g. Decl, Stmt or Type, is reached. Meanwhile, the WalkUpFrom method calls Visit method to actually visit the node. Those three methods are assigned to a priority from highest to low: Traverse \(\rightarrow\) WalkUpFrom \(\rightarrow\) Visit, in which a method can only call methods from the same, or lower levels.

To activate RecursiveASTVisitor method, we invoke its TraverseDecl function from the HandleTranslationUnit method of our ASTConsumer. This function will

\(^3\)https://en.wikipedia.org/wiki/Curiously_recurring_template_pattern
traverse all the declarations of a CLANG translation unit. More details of the RecursiveASTVisitor class could be found in the comments of file[4].

4. Rewriter. Due to the immutable design of the CLANG AST once created, direct AST manipulations (e.g. insert or delete an AST node) is not allowed so far. Instead, CLANG offers a Rewriter interface which enables developers making textual changes, such as inserting, removing or replacing text, to the source programs. For developers who are interested in code refactoring or source-to-source code transformations, the Rewriter interface is a key component coupled with the above classes.

Using the Rewriter interface, we perform necessary analysis on the input programs over the AST and make changes accordingly. By taking advantage of CLANG’s outstanding preservation of source locations for all AST nodes, this method could take as inputs CLANG SourceLocations (or SourceRanges) which indicates the location (or the range) in the input programs where that particular AST node is located, and change the input programs at specific places to perform the transformations precisely.

6.4.3 Back-End of the MetaFork Compiler

The goal of the MetaFork back-end is to produce programs written in parallel programming languages (i.e. CilkPlus, MetaFork, OpenMP and CUDA) other than to generate an executable binary file. In CLANG, this is achieved by RewriteBuffer class. RewriteBuffer is a method of the Rewriter class and stores the source programs. It preserves lots of high-level information within the source programs, like comments, formatting and so on, aiming at being capable of accurately reproducing the input. It is crucial to point out that in order to perform code transformations, an analysis is performed on the CLANG AST other than the RewriteBuffer object. However, CLANG manages a close correspondence between the RewriteBuffer object and the CLANG AST to mirror the modifications in the input programs with the help of Rewriter. So as modifications are applied on the input programs, a new RewriteBuffer is generated. This new buffer captures these textual modifications as information used to map between source locations obtained from the AST in the input programs and outputs in the new RewriteBuffer. In our work, we redirect this new RewriteBuffer to new files as CLANG back-end’s output. For instance, in Figure 6.6, output.cpp file is the output of the CLANG back-end’s output.

6.5 User DefinedPragma Directives in MetaFork

The front-end of the MetaFork compiler is required to be able to preprocess and parse the input programs written by different parallel programming languages (e.g. CilkPlus, MetaFork and OpenMP) based on Pragma directives. In order to be recognized as an user defined directive and not an ordinary comment or another directive, a specification is introduced to take care of matching Pragma directives for each occurrence in the input programs against the specification pre-defined by the developers. If a matching between a Pragma directive and the specification is successfully made, then the message contained in the Pragma directive is recorded and

associated with the corresponding CLANG statement to which that Pragma directive is referring, otherwise, a syntax error is announced. This section outlines how we deal with our high-level abstractions (i.e. Pragma directives) during the overall processing phase of the MetaFork compiler.

6.5.1 Registration of Pragma Directives in the MetaFork Preprocessor

Figure 6.8 illustrates the whole process of the CLANG front-end in a more delicate manner. We focus on how it deals with preprocessor directives, in particular, Pragma directives. The Lexer reads from the input programs and divides the input stream into an individual token for the Parser. More accurately, the input stream needs to be processed by the Preprocessor that provides the ability for expanding preprocessor directives while creating tokens. Over the course of processing the input stream, the Preprocessor is called by the Lexer once preprocessor directives are detected. The Preprocessor reads all tokens from the Lexer by the end of that directive. At this point, it is obvious that the Preprocessor is the entry point of the MetaFork compiler to interact with the CLANG front-end since Pragma directives belong to preprocessor directives.

```cpp
1 struct PragmaMetaHandler : public PragmaHandler {
2   Sema &S;
3   PragmaLocList &List;
4
5   PragmaMetaHandler(Sema &S, PragmaLocList &List) :
6     PragmaHandler("mf"), List(List), S(S) {} 
7
8   void HandlePragma(Preprocessor &PP, PragmaIntroducerKind Introducer,
9     Token &FirstToken) override;
10 };
```

Figure 6.9: A code snippet showing how to create a MetaFork user defined PragmaHandler

The first step is to define user defined Pragma directives which can be recognized by the CLANG front-end. The connection between the user defined Pragma directives and the CLANG front-end is implemented with the help of PragmaHandler class\(^5\) which is the handler in-

\(^5\)http://clang.llvm.org/doxygen/classclang_1_1PragmaHandler.html
6.5. User Defined Pragma Directives in MetaFork

stance being registered to the Clang front-end to be invoked when Pragma directives are encountered in input programs. For the sake of brevity, we take the implementation of MetaFork Pragma directives as a sample to illustrate how a new Pragma handler is specified and the same method is applied to realize the implementation of CilkPlus Pragma directives. As shown in Figure 6.9, a MetaFork Pragma handler class derived from PragmaHandler, namely PragmaMetaHandler, is defined. Each Pragma handler optionally has a PragmaNamespace to represent the base name of the Pragma directive. As required of class PragmaHandler, this base name must be a textual string which immediately follows the #pragma in the same line. In the example, the base name is defined to be mf at Line (6). The most important function provided by the Pragma handler is a virtual function HandlePragma which is a callback function on the preprocessor action. Once a Pragma directive with the base name is being matched, HandlePragma function is invoked automatically by the Clang Preprocessor.

Then we need to register a Pragma handler instance to the Clang front-end such that it can take effect. In Figure 6.10 an instance of the PragmaMetaHandler class is registered by adding it to the Clang Preprocessor (PP) which can be retrieved from the CompilerInstance instance in the ASTFrontendAction instance. From now on, the Clang Preprocessor is aware of our user defined Pragma directives and will trigger user defined actions through HandlePragma function as soon Pragma directives are matched.

PP.AddPragmaHandler(new PragmaMetaHandler(S, List));

Figure 6.10: A code snippet showing how to register a new user defined Pragma handler instance to Clang preprocessor

6.5.2 Parsing of Pragma Directives in the MetaFork Front-End

The Pragma directives are processed in the preprocessing phase ahead of the complete Clang AST generation. Keep in mind that, within MetaFork framework, Pragma directives are implemented without touching any core classes/functions in Clang/LLVM source files. However, a minor defect of this method is that the Clang AST can not represent any parallelism information carried by Pragma directives. This implies the necessity of building a bridge to easily map a Pragma directive to the AST node it precedes. Hence, the objective of the Preprocessor comprises two aspects: (1) parsing the Pragma directives correctly and (2) offering an interface to match between a Pragma directive and its corresponding AST node.

The procedure to achieve the first objective is straightforward. In the preprocessing phase, Clang Preprocessor takes ownership of the Lexer in the presence of tok::ppPragma token, then it consumes the next token which leads to two scenarios. If the token is correctly matched with mf (i.e. the base name of MetaFork Pragma directives), the Lexer consumes the input stream until the end of the Pragma directive, that is the tok::eod token is encountered, and this Pragma directive is stored in a list data structure. Reversely, if the matching cannot be performed, then the default behavior of the Clang front-end is applied.

Regarding the second objective, the observation is that by the time the PragmaHandler is called, the corresponding statement attached to that Pragma directive has not been parsed yet. Thus, the link between the Pragma directive and the AST node associated to that statement
is futile. To overcome this limitation imposed by the lack of any AST information when the PragmaHandler is being invoked, we retrieve the only context information available at that time, i.e. locations. The key point is that, immediately after the `tok::eod` token, the Lexer peeks ahead one token without consuming it. The location of this peeked token, namely context location, is tied to its preceding Pragma directive and as a basic property, this location information will also be captured by the AST during the construction. However, if a Pragma directive does not refer to a statement, like a `#pragma mf join` directive detailed in Section 6.6.1, context location is ignored automatically by the MetaFork front-end.

6.5.3 Attaching Pragma Directives to the Clang AST

As stated before, the Clang AST is hidden behind the MetaFork high-level environment annotated at programs through Pragma directives. Without the association between a Pragma directive and its related AST node, a pure AST representation is meaningless from the perspective of source-to-source compilers, because we can not figure out where to start out the transformations. Our solution is to use RecursiveASTVisitor to visit each node of the Clang AST. We query the location of the AST node from where this node was created. If it is identical to the context location of a Pragma directive, then this Pragma directive gets associated to that AST node.

6.6 Implementation

To date, the MetaFork compilation framework supports two flavors of annotation in programs to express parallelism: keywords and Pragma directives. Both flavors are designed to achieve the same goal to aid the development of compiler supported accelerator technologies. In fact, keywords flavor, as a supplementary to Pragma directives flavor, is no more than a syntactic sugar to programmers due to the fact that keywords constructs are turned into Pragma directives by Preprocessor. In Sections 6.6.1 and 6.6.2, we depict the way in which various MetaFork and CilkPlus constructs (both keywords and Pragma directives) are handled by the MetaFork front-end by the means of discussing a list of example cases, respectively.

6.6.1 Parsing MetaFork Constructs

Table 6.1: MetaFork constructs and clauses

<table>
<thead>
<tr>
<th>Keywords</th>
<th>Pragma directives</th>
<th>optional clauses</th>
</tr>
</thead>
<tbody>
<tr>
<td>meta_fork</td>
<td><code>#pragma mf fork</code></td>
<td>shared</td>
</tr>
<tr>
<td>meta_for</td>
<td><code>#pragma mf parallel for</code></td>
<td></td>
</tr>
<tr>
<td>meta_join</td>
<td><code>#pragma mf join</code></td>
<td></td>
</tr>
<tr>
<td>meta_schedule</td>
<td><code>#pragma mf schedule</code></td>
<td>cache</td>
</tr>
</tbody>
</table>

Table 6.1 summarizes the MetaFork constructs, including both Keywords and Pragma directives flavors, and clauses, which could be recognized by the MetaFork front-end cur-
6.6. Implementation

Currently. In addition, the first column (resp. second column) in conjunction with the third column denote the legal combination of Keywords (resp.Pragma directives) and clauses. Finally, the MetaFork front-end substitutes the MetaFork Keywords with the correspondingPragma directives defined in Figure 6.11 when preprocessing the input programs.

```c
#define meta_fork _Pragma(" mf fork ")
#define meta_schedule _Pragma(" mf schedule ")
#define meta_for _Pragma(" mf parallel for for ")
#define meta_join _Pragma(" mf join ")
```

Figure 6.11: Convert MetaFork Keywords to Pragma directives

**Parsing meta_for construct.** In Figure 6.12 the MetaFork Pragma directive at Line (1) annotates the for loop at Line (2) as a parallel for loop. After consuming the tok::pp pragma token, as usual, the Preprocessor reads the next token mf which indicates that this Pragma directive is introduced by the MetaFork programming language. Then the following two tokens (parallel and for) are consumed to denote this Pragma directive as a parallel for loop directive. Immediately, a new-line token, tok::eod, terminates the parsing of the whole Pragma directive at Line (1). In the last step, the Lexer uses a look-ahead technique to collect the location information of the for token at Line (2). The entire procedure is faithfully coherent with the steps discussed in Section 6.5.2.

```c
1 #pragma mf parallel for
2 for(int i=0;i<10;i++)
3 {
4   
5 }
```

Figure 6.12: A code snippet showing how to annotate a parallel for loop with MetaFork Pragma directive

On the other hand, the Keywords version in Figure 6.13 of its Pragma directives counterpart in Figure 6.12 is obtained using the meta_for keyword. The code in Figure 6.14 is obtained after preprocessing the code in Figure 6.13. Note the difference between Figures 6.12 and 6.14. Due to the substitution of the meta_for construct, the last for token at Line (1), which should be a part of the C/C++ language keyword at Line (2) to format a valid code, appears in the same line with the Pragma directive. By default Clang Preprocessor will consume that last for token as part of the Pragma directive, thus, from the Clang Parser’s

```c
1 #pragma mf parallel for for
2   (int i=0; i<10; i++)
3 {
4   
5 }
```

Figure 6.14: The code snippet after preprocessing the code in Figure 6.13

```c
1 //comsume the first for
2 PP.Lex(Tok);
3
4 // we end Clang Preprocessor and set end of directive
5 PP.getCurrentLexer()->setParsingPreprocessorDirective(false);
6 Tok.setKind(tok::eod);
```

Figure 6.15: Setting the tok::eod token

perspective, this results in an invalid code from Line (2). To address this problem, a trick is applied by the MetaFork Preprocessor. After the Preprocessor consumes the first for token at Line (1), we set it as a tok::eod token by force. As a result, the Preprocessor ends and the Lexer advances to the next token, that is the second for, which is associated with Line (2) to constitute a valid for loop statement now. This trick is done by the code in Figure 6.15.

**Parsing meta_fork construct.** In Figure 6.16, the MetaForkPragma directive at Line (1)

```c
1 #pragma mf fork shared(....)
2 statement
```

Figure 6.16: A code snippet showing how to annotate a parallel region with MetaForkPragma directive

annotates the statement at Line (2) as a parallel region. The equivalent Keywords version is displayed in Figure 6.17 using the meta_fork keyword and the shared clause. The preprocessed code of Figure 6.17 is listed in Figure 6.18. The issue arisen from the code in Figure 6.18 is that the shared clause, as a component of thePragma directive, is on a separate line, while by default the Preprocessor can not reach this point. We use a unified strategy to handle the cases in Figures 6.18 and 6.16. After the fork token is parsed, we set it as a tok::eod token by force and terminate the Preprocessor. The Lexer continues and looks ahead to the next token. If it is a shared token, the Lexer will consume until the end of the token "")" as described in Figure 6.19 and then collect the context location information. Otherwise, the Lexer will merely extract the context location information.

**Parsing meta_join construct.** In Figure 6.20, Line (1) annotates a join directive and the counterpart Keywords version is shown in Figure 6.21 using the meta_join keyword. The preprocessed code of Figure 6.21 is denoted in Figure 6.22. The parsing procedure is almost coherent with the steps discussed in Section 6.5.2 except that a context location is not needed since a join directive is a standalonePragma directive.

**Parsing meta_schedule construct.** In Figure 6.23, the MetaForkPragma directive at Line (1) indicates that the statement at Line (2) could be executed on an external device and the
6.6. Implementation

1 meta_fork shared(...)
2 statement

Figure 6.17: A code snippet showing how to annotate a parallel region with MetaFork keyword

1 #pragma mf fork
2 shared(...)
3 statement

Figure 6.18: The code snippet after preprocessing the code of Figure 6.17

1 // consume "fork" token
2 PP.getCurrentLexer()->setParsingPreprocessorDirective(false);
3 Tok.setKind(tok::eod);
4
5 while (PP.LookAhead(0).is(tok::identifier) &&
6 PP.LookAhead(0).getIdentifierInfo()->isStr("shared"))
7 {
8 // consume shared
9 PP.Lex(Tok);
10 // consume '('
11 PP.Lex(Tok);
12 bool LexID = true;
13
14 while (true) {
15 PP.Lex(Tok);
16
17 if (LexID) {
18 if (Tok.is(tok::identifier)) {
19     Vars.push_back(Tok.getIdentifierInfo()->getName());
20     LexID = false;
21     continue;
22 }
23 }
24 // We are execting a ')') or a ','.
25 if (Tok.is(tok::comma)) {
26     LexID = true;
27     continue;
28 }
29 if (Tok.is(tok::r_paren)) {
30     enddirective = Tok.getLocation();
31     break;
32 }
33 }
34 }

Figure 6.19: Consuming the shared clause

1 #pragma mf join

Figure 6.20: A code snippet showing how to annotate a join construct with MetaForkPragma directive
1 meta_join;

Figure 6.21: A code snippet showing how to annotate a join construct with MetaFork Keywords

1 #pragma mf join
2 ;

Figure 6.22: The code snippet after preprocessing the code of Figure 6.21

counterpart Keywords version is shown in Figure 6.24 using the meta_schedule keyword together with the cache clause. The preprocessed code of Figure 6.24 is denoted in Figure 6.25. The parsing method that parses the meta_schedule construct is the same method parsing the meta_fork construct.

1 #pragma mf schedule cache(...)  
2  statement

Figure 6.23: A code snippet showing how to annotate device code with MetaForkPragma directive

### 6.6.2 Parsing CilkPlus Constructs

Table 6.2 summarizes the CilkPlus constructs, including both Keywords and Pragma directives flavors, which could be recognized by MetaFork front-end currently. Note that no clause is supported yet. In the preprocessing phase, the MetaFork front-end substitutes the CilkPlus Keywords constructs with the corresponding Pragma directives defined in Figure 6.26. Compared to MetaFork constructs, CilkPlus constructs are defined with another base name, namely cilk, while the parsing method is almost identical to the way of parsing MetaFork constructs.

**Parsing cilk_for construct.** In Figure 6.27, the CilkPlus Pragma directive at Line (1) annotates the for loop at Line (2) as a parallel for loop and the counterpart Keywords version is shown in Figure 6.28 using the cilk_for keyword. The preprocessed code of Figure 6.28 is denoted in Figure 6.29. The parsing method that parses the cilk_for construct is the same method parsing the meta_for construct as shown in Section 6.6.1.

**Parsing cilk_spawn construct.** In Figure 6.30, the CilkPlus Pragma directive at Line (1) annotates the statement at Line (2) as a parallel function call. The equivalent CilkPlus Keywords version is displayed in Figure 6.31 using the cilk_spawn keyword. The preprocessed code of Figure 6.31 is listed in Figure 6.32. The parsing method that parses the cilk_spawn construct is almost the same method parsing the meta_fork construct as shown in Section 6.6.1 except that no clause is parsed here.

**Parsing cilk_sync construct.** In Figure 6.33, Line (1) annotates a sync directive and the counterpart Keywords version is shown in Figure 6.34 using the cilk_sync keyword. The preprocessed code of Figure 6.34 is denoted in Figure 6.35. The parsing method that parses the cilk_sync construct is the same method parsing the meta_join construct as shown in Section 6.6.1.
6.6. Implementation

```plaintext
1  meta_schedule cache(...) 
2  statement
```

Figure 6.24: A code snippet showing how to annotate device code with MetaFork Keywords

```plaintext
1  #pragma mf schedule 
2  cache(...) 
3  statement
```

Figure 6.25: The code snippet after preprocessing the code of Figure 6.24

Table 6.2: CilkPlus constructs and clauses

<table>
<thead>
<tr>
<th>Keywords</th>
<th>Pragma directives</th>
<th>optional clauses</th>
</tr>
</thead>
<tbody>
<tr>
<td>cilk_spawn</td>
<td>#pragma cilk spawn</td>
<td></td>
</tr>
<tr>
<td>cilk_for</td>
<td>#pragma cilk parallel for</td>
<td></td>
</tr>
<tr>
<td>cilk_sync</td>
<td>#pragma cilk sync</td>
<td></td>
</tr>
</tbody>
</table>

```plaintext
#define cilk_spawn _Pragma(" cilk spawn ")
#define cilk_for _Pragma(" cilk parallel for for ")
#define cilk_sync _Pragma(" cilk sync ")
```

Figure 6.26: Convert CilkPlus Keywords to Pragma directives

```plaintext
1  #pragma cilk parallel for 
2  for(int i=0;i<10;i++) 
3  { 
4  
5  }
```

Figure 6.27: A code snippet showing how to annotate a parallel for loop with CilkPlus Pragma directive

```plaintext
1  cilk_for(int i=0;i<10;i++) 
2  { 
3  
4  }
```

Figure 6.28: A code snippet showing how to annotate a parallel for loop with CilkPlus keyword

```plaintext
1  #pragma cilk parallel for for 
2  (int i=0;i<10;i++) 
3  { 
4  
5  }
```

Figure 6.29: The code snippet after preprocessing the code Figure 6.28
#pragma cilk spawn
function call;

Figure 6.30: A code snippet showing how to annotate a parallel function call with CilkPlusPragma directive

cilk_spawn function call;

Figure 6.31: A code snippet showing how to annotate a parallel function call with CilkPlus keyword

#pragma cilk spawn
function call;

Figure 6.32: The code snippet after preprocessing the code of Figure 6.31

#pragma cilk sync

Figure 6.33: A code snippet showing how to annotate a sync construct with CilkPlusPragma directive

cilk_sync;

Figure 6.34: A code snippet showing how to annotate a sync construct with CilkPlus Keyword

#pragma cilk sync
;

Figure 6.35: The code snippet after preprocessing the code of Figure 6.34
6.7 Summary

In this chapter, we have presented our MetaFork compilation framework with two objectives: (1) offering high-level parallel programming constructs, and (2) offering source-to-source transformation tools to utilize this framework. With the first objective, our framework addresses major challenges, such as performance, portability and scalability, imposed by low-level parallel programming APIs, and with the later, it builds a bridge between our framework and contemporary concurrency platforms. Developing such a framework from scratch is strenuous. Hence, the MetaFork compilation framework is built on top of modern compiler product Clang for parsing and generating AST from input programs. To be precise, we could manipulate mainstream C/C++ language extensions, OpenMP and CilkPlus, as well as MetaFork.

As a part of the MetaFork compilation framework, we defined a general scheme to handle user defined Pragma directives, since MetaFork, as a parallel programming language, opts Pragma directives to explicitly expose parallelism in source programs. This scheme lifts our framework as a standalone tool without editing any Clang source files and eases the burden on integrating new Pragma directives into our framework if needed. Additionally, we outlined the methods to steer the Clang AST in an efficient manner for the reason that developing source-to-source transformation tools needs to query the AST frequently.
Chapter 7

Towards Comprehensive Parametric CUDA Kernel Generation

In Chapter 4, we demonstrated that, from an annotated C code, it was possible to generate CUDA kernels that depend on program parameters considered unknown at compile-time. Our experimental results in Chapter 4 suggest that those parametric CUDA kernels could help with increasing portability and performance of CUDA code.

In the present chapter, we enhance this strategy as follows. First, we propose an algorithm for comprehensive optimization allowing us to optimize C code (and in particular CUDA code) depending on unknown machine and program parameters. Then, we use this algorithm to generate optimized parametric CUDA kernels, in the form of a case distinction based on the possible values of the machine and program parameters [137]. We call comprehensive parametric CUDA kernels the resulting CUDA kernels, see Section 7.2.

In broad terms, this is a decision tree, where each edge holds a Boolean expression, given by a conjunction of polynomial constraints, and each leaf is either a CUDA kernel or the symbol $\emptyset$, such that for each leaf $K$, with $K \neq \emptyset$, we have:

1. $K$ works correctly under the conjunction of the Boolean expressions located between the root node and the leaf, and
2. $K$ is semantically equivalent to a common input annotated C code $P$.

In each Boolean expression, the unknown variables represent machine parameters (like hardware resource limits), program parameters (like dimension sizes of thread-blocks) or data parameters (like input data size). The symbol $\emptyset$ is used to denote a situation (in fact, value ranges for the machine and program parameters) where no CUDA kernel equivalent to $P$ is provided.

The intention, with the concept of comprehensive parametric CUDA kernels, is to automatically generate optimized CUDA kernels from an annotated C code without knowing the numerical values of some, or all, of the machine and program parameters. This naturally yields a case distinction depending on the values of those parameters. Indeed, some optimization techniques (like loop unrolling) can only be applied when enough computing resources are available, while other optimization techniques (like common sub-expression elimination) can be applied to reduce computing resource consumption. These case distinctions can be handled by techniques from symbolic computation, for which software libraries are available, in particular, the RegularChains library freely available at www.regularchains.org.

Other research groups have approached the questions of code portability and code opti-
mization in the context of CUDA code generation from high-level programming models. They use techniques like auto-tuning [63,86], dynamic instrumentation [89] or both [136]. Rephrasing [86], “those techniques explore empirically different data placement and thread/block mapping strategies, along with other code generation decisions, thus facilitating the finding of a high-performance solution.”

In the case of auto-tuning techniques, which have been used successfully in the celebrated projects ATLAS [151], FFTW [57], and SPIRAL [124], part of the code optimization process is done off-line, that is, the input code is analyzed and an optimization strategy (i.e. a sequence of composable code transformations) is generated, and then applied on-line (i.e. on the targeted hardware). We propose to push this idea further by applying the optimization strategy off-line, thus, even before the code is loaded on the targeted hardware.

Let us illustrate, with an example, the notion of comprehensive parametric CUDA kernels, along with a procedure to generate them. For computing the sum of two matrices a and b of order N, our input is the meta_for-loop nest within the meta_schedule statement on the right-hand portion of Figure 7.1, whereas the serial code without tiling is provided on the left-hand portion of Figure 7.1.

```c
for (int i = 0; i < N; i++)
   for (int j = 0; j < N; j++)
      c[i][j] = a[i][j] + b[i][j];
```

```c
(int dim0 = N/B0, dim1 = N/(2*B1);
meta_schedule {
   meta_for (int v = 0; v < dim0; v++)
      meta_for (int p = 0; p < dim1; p++)
         meta_for (int u = 0; u < B0; u++)
            meta_for (int q = 0; q < B1; q++)
               {
                  int i = v * B0 + u;
                  int j = p * B1 + q;
                  if (i < N && j < N/2) {
                     c[i][j] = a[i][j] + b[i][j];
                     c[i][j+N/2] =
                        a[i][j+N/2] + b[i][j+N/2];
                  }
               }
}
```

(a) Before tiling, the C program     (b) After tiling, the MetaFork program

Figure 7.1: Matrix addition written in C (the left-hand portion) and in MetaFork (the right-hand portion) with a meta_for loop nest, respectively

We make the following simplistic assumptions for the translation of this meta_for-loop nest to a CUDA program.

1. The target machine has two parameters: the maximum number \( R_2 \) of registers per thread, and the maximum number \( R_1 \) of threads per thread-block; moreover, all other hardware limits are ignored.
2. The generated kernels depend on two program parameters, \( B_0 \) and \( B_1 \), which define the format of a 2D thread-block.
3. The optimization strategy (w.r.t. register usage per thread) consists in reducing the work per thread via removing the 2-way loop unrolling.
The possible comprehensive parametric CUDA kernels are given by the pairs \((C_1, K_1)\) and \((C_2, K_2)\), where \(C_1, C_2\) are two sets of algebraic constraints on the machine and program parameters and \(K_1, K_2\) are two CUDA kernels that are optimized under the constraints, respectively, given by \(C_1, C_2\), see Figure 7.2. The following computational steps yield the pairs \((C_1, K_1)\) and \((C_2, K_2)\).

(S1) Tiling techniques, based on quantifier elimination (QE), are applied to the meta_for loop nest of Figure 7.1 in order to decompose the matrices into tiles of format \(B_0 \times B_1\), see [29] for details.

(S2) The tiled MetaFork code is mapped to an intermediate representation (IR) say that of LLVM, or alternatively, to PTX\(^1\) code.

(S3) Using this IR (or PTX) code, one can estimate the number of registers that a thread requires; thus, using LLVM IR on this example, we obtain an estimate of 14.

(S4) Next, we apply the optimization strategy [50], yielding a new IR (or PTX) code, for which register pressure reduces to 10. Since no other optimization techniques are considered, the procedure stops with the result shown in Figure 7.2.

Based on the above steps, Figure 7.3 shows the decision tree for generating these two pairs, each of them consisting of a system of polynomial constraints and a CUDA kernel for matrix addition. Note that, strictly speaking, the kernels \(K_1\) and \(K_2\) on Figure 7.2 should be given by PTX code. But for simplicity, we are presenting them by the CUDA code counterpart.

\[
C_1 : \begin{cases} B_0 \times B_1 \leq R_1 \\ 14 \leq R_2 \end{cases}
\]

\[
\begin{aligned}
\text{__global__ void } & K1(\text{int } *a, \text{int } *b, \text{int } *c, \text{int } N, \\
& \quad \text{int } B0, \text{int } B1) \{ \\
& \quad \text{int } i = \text{blockIdx.y } * B0 + \text{threadIdx.y}; \\
& \quad \text{int } j = \text{blockIdx.x } * B1 + \text{threadIdx.x}; \\
& \quad \text{if } (i < N \&\& j < N/2) \{ \\
& \quad \quad \text{a}[i*N+j] = b[i*N+j] + c[i*N+j]; \\
& \quad \quad \text{a}[i*N+j+N/2] = b[i*N+j+N/2] + c[i*N+j+N/2]; \\
& \quad \} \\
& \} \\
& \text{dim3 dimBlock(B1, B0);} \\
& \text{dim3 dimGrid(N/(2*B1), N/B0);} \\
& \text{K1 <<<<dimGrid, dimBlock>>> (a, b, c, N, B0, B1);}
\end{aligned}
\]

\[
C_2 : \begin{cases} B_0 \times B_1 \leq R_1 \\ 10 \leq R_2 < 14 \end{cases}
\]

\[
\begin{aligned}
\text{__global__ void } & K2(\text{int } *a, \text{int } *b, \text{int } *c, \text{int } N, \\
& \quad \text{int } B0, \text{int } B1) \{ \\
& \quad \text{int } i = \text{blockIdx.y } * B0 + \text{threadIdx.y}; \\
& \quad \text{int } j = \text{blockIdx.x } * B1 + \text{threadIdx.x}; \\
& \quad \text{if } (i < N \&\& j < N) \\
& \quad \quad \text{a}[i*N+j] = b[i*N+j] + c[i*N+j]; \\
& \} \\
& \text{dim3 dimBlock(B1, B0);} \\
& \text{dim3 dimGrid(N/B1, N/B0);} \\
& \text{K2 <<<<dimGrid, dimBlock>>> (a, b, c, N, B0, B1);}
\end{aligned}
\]

Figure 7.2: Comprehensive translation of MetaFork code to two kernels for matrix addition

\(^1\)The Parallel Thread Execution (PTX) [7] is the pseudo-assembly language to which CUDA programs are compiled by NVIDIA’s NVCC compiler. PTX code can also be generated from (enhanced) LLVM IR, using nvptx back-end [3], following the work of [128].
One can observe that loop unrolling is applied to $K_1$ so as to increase arithmetic intensity. This code transformation increases register pressure, which is possible under the constraints of $C_1$ but not under those of $C_2$.

In general, to achieve a comprehensive translation of the annotated C program $\mathcal{P}$ into CUDA kernels, one could be tempted to proceed as follows:

(S1) Perform a comprehensive optimization of $\mathcal{P}$, as a MetaFork program, by applying the comprehensive optimization algorithm demonstrated in Section 7.1.

(S2) Apply the MetaFork-to-CUDA code generator introduced in Chapter 4 to each MetaFork program generated in the first step.

However, since the system of polynomial constraints associated with each optimized MetaFork program is determined by an IR representation of that program, this system would not be accurate for the CUDA code generated by a source-to-source MetaFork-to-CUDA code generator.

In fact, if PTX is used as IR in Step (S1) then Step (S2) is no longer necessary. However, Step (S2) will produce code readable by a human being that will give her/him some sense of the source code transformations performed at Step (S1).

The reason that we choose MetaFork programs as our input programs is because the MetaFork language can be used to write both “high-level” (in the spirit of OpenMP) and “lower-level” (closer to CUDA) parallel programs. Indeed, MetaFork is another high-level heterogeneous programming model like OpenMP, OpenACC and C++ AMP [154]. Hence, turning an unoptimized MetaFork program into optimized MetaFork programs can be seen as a first approximation of our final goal, that is, generating optimized parametric CUDA kernels from input unoptimized annotated C programs. The presented comprehensive optimization algorithm combined with our previous work in Chapter 4 can be used to complete a comprehensive translation of a MetaFork program into parametric CUDA kernels.

Section 7.1 proposes a comprehensive optimization algorithm for optimizing an input code fragment depending on unknown machine and program parameters. In Section 7.2, we describe the procedure of comprehensive translation of an annotated C program, namely MetaFork, into parametric CUDA kernels. An implementation of the comprehensive optimization algorithm is presented in Section 7.3 so as to generate optimized MetaFork programs from a given MetaFork program. We conduct the experimentation in Section 7.4 for optimizing six simple test cases: array reversal, matrix vector multiplication, 1D Jacobi, matrix addition, matrix transpose and matrix matrix multiplication.
This work is a joint project with Ning Xie and Marc Moreno Maza.

7.1 Comprehensive Optimization

We consider a code fragment written in the C language or in one of its linguistic extensions targeting a computer device, which can be, for instance, a hardware accelerator or a desktop CPU. We assume that some, or all, of the hardware characteristics of this device are unknown at compile time. However, we would like to optimize our input code fragment w.r.t. prescribed resource counters (e.g. memory usage) and performance counters (e.g. clock-cycle per instruction). To this end, we treat the hardware characteristics of this device as symbols and generate polynomial constraints (with those symbols as indeterminate variables) ensuring when such and such code transformation is valid.

Section 7.1.1 states the hypotheses made on the input code fragment. Section 7.1.2 specifies the notations for the hardware characteristics of the targeted device. In Section 7.1.3, we describe the evaluation of resource and performance counters. In Section 7.1.4, we define the optimization strategies that can reduce resource counters or increase performance counters. Section 7.1.5 formally gives the definition of comprehensive optimization of an input code fragment. Section 7.1.6 specifies the data structures that are used in our algorithm. Finally, in Section 7.1.7, we demonstrate our algorithm for comprehensive optimization.

7.1.1 Hypotheses on the Input Code Fragment

We consider a sequence $S$ of statements from the C programming language and introduce the following.

**Definition 1** We call parameter of $S$ any scalar variable that is

(i) read in $S$ at least once and

(ii) never written in $S$.

We call data of $S$ any non-scalar variable (e.g. array) that is not initialized but possibly overwritten within $S$. If a parameter of $S$ gives a dimension size of a data of $S$, then this parameter is called a data parameter; otherwise, it is simply called a program parameter.

**Notation 1** We denote by $D_1, \ldots, D_u$ and $E_1, \ldots, E_v$ the data parameters and program parameters of $S$, respectively.

**Hypothesis 1** We make the following assumptions on $S$.

(H1) All parameters are assumed to be non-negative integers.

(H2) We assume that $S$ can be viewed as the body of a valid C function having the parameters and data of $S$ as unique arguments.

**Example 1** $S$ can be the body of a kernel function in CUDA. Recall that the kernel code for computing matrix vector multiplication in Figure 4.14 of Chapter 4.4. This kernel code multiplies a square matrix $a$ of order $n$ with a vector $b$ of length $n$ and stores the result to a vector $c$ of length $n$. We note that $a$, $b$ and $c$ are the data, and that $n$ is the data parameter. Moreover, the grid and thread-block dimensions of this kernel are specified as $\text{dim}$ and $B$, respectively, which are then the program parameters.
7.1.2 Hardware Resource Limits and Performance Measures

We denote by $R_1, \ldots, R_s$ the hardware resource limits of the targeted hardware device. Examples of these quantities for the NVIDIA Kepler micro-architecture are:
- the maximum number of registers to be allocated per thread,
- the maximum number of shared memory words to be allocated per thread-block,
- the maximum number of threads in a thread-block.

We denote by $P_1, \ldots, P_t$ the performance measures of a program running on the device. These are dimensionless quantities typically defined as percentages. Examples of these quantities for the NVIDIA Kepler micro-architecture are:
- the ratio of the actual to the maximum number of words that can be read or written per unit of time from the global memory,
- the ratio of the actual to the maximum number of floating point operations that can be performed per unit of time by all streaming processors (SMs),
- the SM occupancy, that is, the ratio of active warps to the maximum number of active warps,
- the cache hit rate in an SM.

For a given hardware device, $R_1, \ldots, R_s$ are positive integers, and each of them is the maximum value of a hardware resource. Meanwhile, $P_1, \ldots, P_t$ are rational numbers between 0 and 1. However, for the purpose of writing code portable across a variety of devices with similar characteristics, the quantities $R_1, \ldots, R_s$ and $P_1, \ldots, P_t$ will be treated as unknown and independent variables. These hardware resource limits and performance measures will be called the machine parameters.

Each function $K$ (and, in particular, our input code fragment $S$) written in the C language for the targeted hardware device has resource counters $r_1, \ldots, r_s$ and performance counters $p_1, \ldots, p_t$ corresponding, respectively, to $R_1, \ldots, R_s$ and $P_1, \ldots, P_t$. In other words, the quantities $r_1, \ldots, r_s$ are the amounts of resources, corresponding to $R_1, \ldots, R_s$, respectively, that $K$ requires for executing. Similarly, the quantities $p_1, \ldots, p_t$ are the performance measures, corresponding to $P_1, \ldots, P_t$, respectively, that $K$ exhibits when executing. Therefore, the inequalities $0 \leq r_i \leq R_i, \ldots, 0 \leq r_s \leq R_s$ must hold for the function $K$ to execute correctly. Similarly, $0 \leq p_i \leq 1, \ldots, 0 \leq p_t \leq 1$ are satisfied by the definition of the performance measures.

Remark 1 We note that $r_1, \ldots, r_s, p_1, \ldots, p_t$ may be numerical values, which we can assume to be non-negative rational numbers. This will be the case, for instance, for the minimum number of registers required per thread in a thread-block. The resource counters $r_1, \ldots, r_s$ may also be polynomial expressions whose indeterminate variables can be program parameters (like the dimension sizes of a thread-block or grid) or data parameters (like the input data sizes). Meanwhile, the performance counters $p_1, \ldots, p_t$ may further depend on the hardware resource limits (like the maximum number of active warps supported by an SM). To summarize, we observe that $r_1, \ldots, r_s$ are polynomials in $\mathbb{Q}[D_1, \ldots, D_u, E_1, \ldots, E_v]$ and $p_1, \ldots, p_t$ are rational functions where numerators and denominators are in $\mathbb{Q}[D_1, \ldots, D_u, E_1, \ldots, E_v, R_1, \ldots, R_s]$. Moreover, we can assume that the denominators of those rational functions are positive.

Example 2 For computing the product of a dense square matrix of order $N$ by a dense vector of length $N$, consider the serial C code on the left-hand portion of Figure 7.4 and the corresponding MetaFork code on the right-hand portion of Figure 7.4. The meta_schedule statement yields
the CUDA kernel shown in Figure 4.14 of Chapter 4. The grid of this kernel is one-dimensional of size \( \text{dim} \), and its thread-blocks are also one-dimensional, each counting \( B \) threads.

```c
for (int p = 0; p < \( N \); p++)
for (int q = 0; q < \( N \); q++)
c[p] += a[p][q] * b[q];
```

```c
int \( \text{dim} = N / B \); meta_schedule {
meta_for (int v = 0; v < \( \text{dim} \); v++)
for (int i = 0; i < \( \text{dim} \); ++i)
meta_for (int u = 0; u < \( B \); u++)
for (int j = 0; j < \( B \); ++j) {
int p = v * \( B \) + u;
int q = i * \( B \) + j;
c[p] += a[p][q] * b[q];
}
}
```

(a) Before tiling, the C program  
(b) After tiling, the MetaFork program

Figure 7.4: Matrix vector multiplication written in C (the left-hand portion) and in MetaFork (the right-hand portion), respectively

Observe that, in the process of generating this CUDA kernel, some optimization techniques (like using the shared memory for arrays \( a, b \) and \( c \) in Figure 4.14 of Chapter 4) are applied, while other optimization techniques remain to be applied (like the granularity of threads as used in Figure 7.9). If \( R_2 \) and \( R_3 \) are two machine parameters that define, respectively, the maximum number of threads in a thread-block and the maximum number of shared memory words per thread-block, then the following constraints must hold:

\[
B \leq R_2 \quad \text{and} \quad B^2 + 2B \leq R_3.
\]

Note that, in Figure 4.14 of Chapter 4 each thread-block allocates a unit size \( B^2 \) of shared memory words for the array \( a \), and a unit size \( B \) of shared memory words, respectively, for each of arrays \( b \) and \( c \). Thus, this leads to the inequality \( B^2 + 2B \leq R_3 \).

### 7.1.3 Evaluation of Resource and Performance Counters

Let \( G_C(S) \) be the control flow graph (CFG) of \( S \). Hence, the statements in the basic blocks of \( G_C(S) \) are C statements, and we call such a CFG the source CFG. We also map \( S \) to an intermediate representation, which, itself, is encoded in the form of a CFG, denoted by \( G_L(S) \), and we call it the IR CFG. Here, we refer to the landmark textbook [8] for the notion of the control flow graph and that of intermediate representation.

We observe that \( S \) can trivially be reconstructed from \( G_C(S) \); hence, the knowledge of \( S \) and that of \( G_C(S) \) can be regarded as equivalent. In contrast, \( G_L(S) \) depends not only on \( S \) but also on the optimization strategies that are applied to the IR of \( S \).

Equipped with \( G_C(S) \) and \( G_L(S) \), we assume that we can estimate each of the resource counters \( r_1, \ldots, r_s \) (resp. performance counters \( p_1, \ldots, p_t \)) by applying functions \( f_1, \ldots, f_s \) (resp. \( g_1, \ldots, g_t \)) to either \( G_C(S) \) or \( G_L(S) \). We call \( f_1, \ldots, f_s \) (resp. \( g_1, \ldots, g_t \)) the resource (resp. performance) evaluation functions.
For instance, when \( S \) is the body of a CUDA kernel and \( S \) reads (resp. writes) a given array, computing the total amount of elements read (resp. written) by one thread-block can be determined from \( G_C(S) \). Meanwhile, computing the minimum number of registers to be allocated to a thread executing \( S \) requires the knowledge of \( G_L(S) \).

### 7.1.4 Optimization Strategies

In order to reduce the consumption of hardware resources and increase performance counters, we assume that we have \( w \) optimization procedures \( O_1, \ldots, O_w \), each of them mapping either a source CFG to another source CFG, or an IR CFG to another IR CFG. Of course, we assume the code transformations performed by \( O_1, \ldots, O_w \) preserve semantics.

We associate each resource counter \( r_i \), for \( i = 1 \cdots s \), with a non-empty subset \( \sigma(r_i) \) of \( \{O_1, \ldots, O_w\} \), such that we have

\[
f_i(O(S)) \leq f_i(S) \text{ for } O \in \sigma(r_i).
\]  

(7.1)

Hence, \( \sigma(r_i) \) is a subset of the optimization strategies among \( O_1, \ldots, O_w \) that have the potential to reduce \( r_i \). Of course, the intention is that for at least one \( O \in \sigma(r_i) \), we have \( f_i(O(S)) < f_i(S) \). A reason for not finding such \( O \) would be that \( S \) cannot be further optimized w.r.t. \( r_i \). We also make a natural idempotence assumption:

\[
f_i(O(O(S))) = f_i(O(S)) \text{ for } O \in \sigma(r_i).
\]  

(7.2)

Similarly, we associate each performance counter \( p_i \), for \( i = 1 \cdots t \), with a non-empty subset \( \sigma(p_i) \) of \( \{O_1, \ldots, O_w\} \), such that we have

\[
g_i(O(S)) \geq g_i(S) \text{ and } g_i(O(O(S))) = g_i(O(S)) \text{ for } O \in \sigma(p_i).
\]  

(7.3)

Hence, \( \sigma(p_i) \) is a subset of the optimization strategies among \( O_1, \ldots, O_w \) that have the potential to increase \( p_i \). The intention is, again, that for at least one \( O \in \sigma(p_i) \), we have \( g_i(O(S)) > g_i(S) \).

### 7.1.5 Comprehensive Optimization

Let \( C_1, \ldots, C_e \) be semi-algebraic systems with \( P_1, \ldots, P_t, R_1, \ldots, R_s, D_1, \ldots, D_u, E_1, \ldots, E_v \) as indeterminate variables. Let \( S_1, \ldots, S_e \) be fragments of C programs such that the parameters of each of them are among \( D_1, \ldots, D_u, E_1, \ldots, E_v \).

**Definition 2** We say that the sequence of pairs \( (C_1, S_1), \ldots, (C_e, S_e) \) is a comprehensive optimization of \( S \) w.r.t.

- the resource evaluation functions \( f_1, \ldots, f_s \),
- the performance evaluation functions \( g_1, \ldots, g_t \), and
- the optimization strategies \( O_1, \ldots, O_w \)

if the following conditions hold:

(i) [constraint soundness] Each of the semi-algebraic systems \( C_1, \ldots, C_e \) is consistent, that is, admits at least one real solution.
We say that the quintuple $Q$ has been applied so far to either $S$ and data, there exists a code fragment $S_i$ such that $(h_1, \ldots, h_e, x_1, \ldots, x_r, y_1, \ldots, y_w, z_1, \ldots, z_v)$ is a solution of $C_i$, then the code fragment $S_i$ produces the same output as $S$ on any data that makes $S$ execute correctly.

To summarize Definition 2 in non-technical terms:
- Condition (i) states that each system of constraints is meaningful.
- Condition (ii) states that as long as the machine, program and data parameters satisfy $C_i$, the code fragment $S_i$ produces the same output as $S$ on whichever data that makes $S$ execute correctly.
- Condition (iii) states that as long as $S$ executes correctly on a given set of parameters and data, there exists a code fragment $S_i$ for suitable values of the machine parameters, such that $S_i$ produces the same output as $S$ on that set of parameters and data.
- Condition (iv) states that for each resource counter $r_j$ (performance counter $p_i$), there exists at least one code fragment $S_i$ for which this counter is optimal in the sense that it cannot be further optimized by the optimization strategies from $\sigma(r_j)$ (resp. $\sigma(p_i)$).

### 7.1.6 Data-Structures

The algorithm presented in Section 7.1.7 computes a comprehensive optimization of $S$ w.r.t. the evaluation functions $f_1, \ldots, f_s, g_1, \ldots, g_t$ and optimization strategies $O_1, \ldots, O_w$.

Hereafter, we define the main data-structure used during the course of the algorithm. We associate $S$ with what we call a quintuple, denoted by $Q(S)$ and defined as follows:

$$Q(S) = (G_C(S), \lambda(S), \omega(S), \gamma(S), C(S))$$

where

1. $\lambda(S)$ is the sequence of the optimization procedures among $O_1, \ldots, O_w$ that have already been applied to the IR of $S$; hence, $G_C(S)$ together with $\lambda(S)$ defines $G_I(S)$; initially, $\lambda(S)$ is empty,
2. $\omega(S)$ is the sequence of the optimization procedures among $O_1, \ldots, O_w$ that have not been applied so far to either $G_C(S)$ or $G_I(S)$; initially, $\omega(S)$ is $O_1, \ldots, O_w$,
3. $\gamma(S)$ is the sequence of resource and performance counters that remain to be evaluated on $S$; initially, $\gamma(S)$ is $r_1, \ldots, r_s, p_1, \ldots, p_t$,
4. $C(S)$ is the sequence of the constraints (polynomial equations and inequalities) on $P_1, \ldots, P_r$, $R_1, \ldots, R_s, D_1, \ldots, D_u, E_1, \ldots, E_v$ that have been computed so far; initially, $C(S)$ is $1 \geq P_1 \geq 0, \ldots, 1 \geq P_r \geq 0, R_1 \geq 0, \ldots, R_s \geq 0, D_1 \geq 0, \ldots, D_u \geq 0, E_1 \geq 0, \ldots, E_v \geq 0$.

We say that the quintuple $Q(S)$ is processed whenever $\gamma(S)$ is empty; otherwise, we say that the quintuple $Q(S)$ is in-process.
Remark 2 For the above \( Q(S) \), each of the sequences \( \lambda(S) \), \( \omega(S) \), \( \gamma(S) \) and \( C(S) \) is implemented as a stack in Algorithms 10 and 11. Hence, we need to specify how operations on a sequence are performed on the corresponding stack. Let \( s_1, s_2, \ldots, s_N \) be a sequence.

- Popping one element out of this sequence returns \( s_1 \) and leaves that sequence with \( s_2, \ldots, s_N \).
- Pushing an element \( t_1 \) on \( s_1, s_2, \ldots, s_N \) will update that sequence to \( t_1, s_1, s_2, \ldots, s_N \).
- Pushing a sequence of elements \( t_1, t_2, \ldots, t_M \) on \( s_1, s_2, \ldots, s_N \) will update that sequence to \( t_M, \ldots, t_2, t_1, s_1, s_2, \ldots, s_N \).

7.1.7 The Algorithm

Algorithm 10 is the top-level procedure. If its input is a processed quintuple \( Q(S) \), then it returns the pair \( (G_C(S), \lambda(S)) \) (such that, after optimizing \( S \) with the optimization strategies in \( \lambda(S) \), one can generate the IR of the optimized \( S \)) together with the system of constraints \( C(S) \). Otherwise, Algorithm 10 is called recursively on each quintuple returned by \( \text{Optimize}(Q(S)) \). The pseudo-code of the \( \text{Optimize} \) routine is given by Algorithm 11.

![Algorithm 10: ComprehensiveOptimization (Q(S))](image)

Remark 3 We make a few observations about Algorithm 11.

(R1) Observe that at Line (5), a deep copy of the input \( Q(S') \) is made, and this copy is called \( Q(S'') \). This duplication allows the computations to fork. Note that at Line (6), \( Q(S') \) is modified.

(R2) In this forking process, we call \( Q(S') \) the accept branch and \( Q(S'') \) the refuse branch. In the former case, the relation \( 0 \leq v_i \leq R_i \) holds thus implying that enough \( R_i \)-resources are available for executing the code fragment \( S' \). In the latter case, the relation \( R_i < v_i \) holds thus implying that not enough \( R_i \)-resources are available for executing the code fragment \( S'' \).

(R3) Observe that \( v_i \) is either a numerical value, a polynomial in \( \mathbb{Q}[D_1, \ldots, D_u, E_1, \ldots, E_v] \) or a rational function where its numerator and denominator are in \( \mathbb{Q}[D_1, \ldots, D_u, E_1, \ldots, E_v, R_1, \ldots, R_v] \).

(R4) At Lines (18-20), a similar forking process occurs. Here again, we call \( Q(S') \) the accept branch and \( Q(S'') \) the refuse branch. In the former case, the relation \( 0 \leq v_i \leq P_i \) implies that the \( P_i \)-performance counter may have reached its maximum ratio; hence, no
Algorithm 11: Optimize

Input: A quintuple \( Q(S') \)
Output: A stack of quintuples

1. Initialize an empty stack, called result;
2. Take out from \( \gamma(S') \) the next resource or performance counter to be evaluated, say \( c \);
3. Evaluate \( c \) on \( S' \) (using the appropriate functions among \( f_1, \ldots, f_s, g_1, \ldots, g_t \)) thus obtaining a value \( v_i \), which can be either a numerical value, a polynomial in \( \mathbb{Q}[D_1, \ldots, D_u, E_1, \ldots, E_v] \) or a rational function where its numerator and denominator are in \( \mathbb{Q}[D_1, \ldots, D_u, E_1, \ldots, E_v, R_1, \ldots, R_s] \);
4. if \( c \) is a resource counter \( r_i \) then
   5. Make a deep copy \( Q(S'') \) of \( Q(S') \), since we are going to split the computation into two branches: \( R_i < v_i \) and \( 0 \leq v_i \leq R_i \);
   6. Add the constraint \( 0 \leq v_i \leq R_i \) to \( C(S') \) and push \( Q(S') \) onto result;
   7. Add the constraint \( R_i < v_i \) to \( C(S'') \) and search \( \omega(S'') \) for an optimization strategy of \( \sigma(r_i) \);
   8. if no such optimization strategy exists then
      9. return result;
   else
      10. Apply such an optimization strategy to \( Q(S'') \) yielding \( Q(S''') \);
      11. Remove this optimization strategy from \( \omega(S''') \);
      12. if this optimization strategy is applied to the IR of \( S'' \) then
          13. Add it to \( \lambda(S'') \);
          14. Push \( r_1, \ldots, r_{i-1}, r_i \) onto \( \gamma(S''') \);
          15. Make a recursive call to Optimize on \( Q(S''') \) and push the returned quintuples onto result;
   if \( c \) is a performance counter \( p_i \) then
      17. Make a deep copy \( Q(S'') \) of \( Q(S') \), since we are going to split the computation into two branches: \( 0 \leq v_i \leq P_i \) and \( P_i < v_i \leq 1 \);
      18. Add the constraint \( 0 \leq v_i \leq P_i \) to \( C(S') \) and push \( Q(S') \) onto result;
      19. Add the constraint \( P_i < v_i \leq 1 \) to \( C(S'') \) and search \( \omega(S'') \) for an optimization strategy of \( \sigma(p_i) \);
      20. if no such optimization strategy exists then
          21. return result;
      else
          22. Apply such an optimization strategy to \( Q(S'') \) yielding \( Q(S''') \);
          23. Remove this optimization strategy from \( \omega(S''') \);
          24. if this optimization strategy is applied to the IR of \( S'' \) then
              25. Add it to \( \lambda(S'') \);
              26. Push \( r_1, \ldots, r_s, p_i \) onto \( \gamma(S''') \);
              27. Make a recursive call to Optimize on \( Q(S''') \) and push the returned quintuples onto result;
      Remove from result any quintuple with an inconsistent system of constraints;
30. return result;
optimization strategies are applied to improve this counter. In the latter case, the relation $P_i < v_i \leq 1$ holds thus implying that the $P_i$-performance counter has not reached its maximum value; hence, optimization strategies are applied to improve this counter if such optimization strategies are available. Observe that if this optimization strategy does make the estimated value of $P_i$ larger then an algebraic contradiction would happen and the branch will be discarded.

(R5) Line (30) in Algorithm 11 requires non-trivial computations with polynomial equations and inequalities. The necessary algorithms can be found in [30] and are implemented in the RegularChains library of MAPLE.

Remark 4 We make a few remarks about the handling of algebraic computation during the execution of Algorithm 11.

(R6) Each system of algebraic constraints $C$ is updated by adding a polynomial inequality to it at either Lines (6), (7), (19) or (20). This incremental process can be performed by the RealTriangularize algorithm [30] and implemented in the RegularChains library.

(R7) Each of these inequalities can be either strict (using $>$) or large (using $\leq$); the left-hand side is a polynomial of either $\mathbb{Q}[D_1, \ldots, D_n, E_1, \ldots, E_v, R_1, \ldots, R_s]$, and the right-hand side is either one of the variables $R_1, \ldots, R_s$ or a polynomial of $\mathbb{Q}[D_1, \ldots, D_n, E_1, \ldots, E_v, R_1, \ldots, R_s]$ times one of the variables $P_1, \ldots, P_t$.

(R8) Because of the recursive calls at Lines (16) and (29) several inequalities involving the same variable among $R_1, \ldots, R_s$, $P_1, \ldots, P_t$ may be added to a given system $C$. As a result, $C$ may become inconsistent. For instance if $10 \leq R_1$ and $R_1 < 10$ are both added to the same system $C$. Note that inconstancy is automatically detected by the RealTriangularize algorithm.

(R9) When using RealTriangularize, variables should be ordered. We choose a variable ordering such that
- any of $P_1, \ldots, P_t$ is greater than any of the other variables,
- any of $R_1, \ldots, R_s$ is greater than any of $D_1, \ldots, D_n, E_1, \ldots, E_v$.

(R10) Then, the RealTriangularize represents the solution of $C$ as the union of the solution sets of finitely many regular semi-algebraic systems. Each such regular semi-algebraic system $\Upsilon$ consists of
- polynomial constraints involving $D_1, \ldots, D_n, E_1, \ldots, E_v$ only,
- polynomial constraints involving $D_1, \ldots, D_n, E_1, \ldots, E_v, R_1, \ldots, R_s$ only and of positive degree in at least one of $R_1, \ldots, R_s$,
- constraints that are linear in $P_1, \ldots, P_t$ and polynomial in $D_1, \ldots, D_n, E_1, \ldots, E_v, R_1, \ldots, R_s$.

Let us denote by $\Upsilon_{D,E}$, $\Upsilon_{D,E,R}$ and $\Upsilon_{D,E,R,P}$ these three sets of polynomial constraints, respectively. It follows from the properties of regular semi-algebraic systems that (a) $\Upsilon_{D,E}$ is consistent,
(b) for all real values $y_1, \ldots, y_n, z_1, \ldots, z_v$ of $D_1, \ldots, D_n, E_1, \ldots, E_v$ such that $(y_1, \ldots, y_n, z_1, \ldots, z_v)$ solves $\Upsilon_{D,E}$, there exists real values $x_1, \ldots, x_s$ of $R_1, \ldots, R_s$ such that $(y_1, \ldots, y_n, z_1, \ldots, z_v, x_1, \ldots, x_s)$ solves $\Upsilon_{D,E,R}$ and real values $h_1, \ldots, h_t$ of $P_1, \ldots, P_t$ such that $(y_1, \ldots, y_n, z_1, \ldots, z_v, x_1, \ldots, x_s, h_1, \ldots, h_t)$ solves $\Upsilon_{D,E,R,P}$. 
Remark 5 Figure 7.5 illustrates how Algorithm 11, applied to \( Q(S') \), generates the associated tree \( T(Q(S')) \). The cases for a resource counter and a performance counter are distinguished in the sub-figures (a) and (b), respectively. Observe that, in both cases, the accept edges go...
south-east, while the refuse edges go south-west.

Lemma 7.1.1 The height of the tree $T(Q(S))$ is at most $w(s + t)$. Therefore, Algorithm 10 terminates.

Proof Consider a path $\Gamma$ from the root of $T(Q(S))$ to any node $N$ of $T(Q(S))$. Observe that $\Gamma$ counts at most $w$ refuse edges. Indeed, following a refuse edge decreases by one the number of optimization strategies to be used. Observe also that the length of every sequence of consecutive accept edges is at most $s + t$. Indeed, following an accept edge decreases by one the number of resource and performance counters to be evaluated. Therefore, the number of edges in $\Gamma$ is at most $w(s + t)$.

Lemma 7.1.2 Let $U := \{U_1, \ldots, U_z\}$ be a subset of $\{O_1, \ldots, O_w\}$. There exists a path from the root of $T(Q(S))$ to a leaf of $T(Q(S))$ along which the optimization strategies being applied are exactly those of $U$.

Proof Let us start at the root of $T(Q(S))$ and apply the following procedure:
1. follow the refuse edge if it uses an optimization strategy from $\{U_1, \ldots, U_z\}$,
2. follow the accept edge, otherwise.
This creates a path from the root of $T(Q(S))$ to a leaf with the desired property.

Definition 3 Let $i \in \{1, \ldots, s\}$ (resp. $\{1, \ldots, t\}$). Let $N$ be a node of $T(Q(S))$ and $Q(S_N)$ be the quintuple labelling this node. We say that $r_i$ (resp. $p_i$) is optimal at $N$ w.r.t. the evaluation function $f_i$ (resp. $g_i$) and the subset $\sigma(r_i)$ (resp. $\sigma(p_i)$) of the optimization strategies $O_1, \ldots, O_w$, whenever for all $O \in \sigma(r_i)$ (resp. $\sigma(p_i)$) we have $f_i(O(S_N)) = f_i(S_N)$ (resp. $g_i(O(S_N)) = g_i(S_N)$).

Lemma 7.1.3 Let $i \in \{1, \ldots, s\}$ (resp. $\{1, \ldots, t\}$). There exists at least one leaf $L$ of $T(Q(S))$ such that $r_i$ (resp. $p_i$) is optimal at $L$ w.r.t. the evaluation function $f_i$ (resp. $g_i$) and the subset $\sigma(r_i)$ (resp. $\sigma(p_i)$) of the optimization strategies $O_1, \ldots, O_w$.

Proof Apply Lemma 7.1.2 with $U = \sigma(r_i)$ (resp. $U = \sigma(p_i)$).

Lemma 7.1.4 Algorithm 10 satisfies its output specifications.

Proof From Lemma 7.1.1 we know that Algorithm 10 terminates. So let $(C_1, S_1), \ldots, (C_e, S_e)$ be its output. We shall prove $(C_1, S_1), \ldots, (C_e, S_e)$ satisfies the conditions (i) to (iv) of Definition 2. Condition (i) is satisfied by the properties of the RealTriangularize algorithm. Condition (ii) follows clearly from the assumption that the code transformations performed by $O_1, \ldots, O_w$ preserve semantics. Observe that each time a polynomial inequality is added to a system of constraints, the negation of this inequality is also to the same system in another branch of the computations. By using a simple induction on $s + t$, we deduce that Condition (iii) is satisfied. Finally, we prove Condition (iv) by using Lemma 7.1.3.
7.2 Comprehensive Translation of an Annotated C Program into CUDA Kernels

Given a high-level model for accelerator programming (like OpenCL [141], OpenMP, OpenACC or MetaFork), we consider the problem of translating a program written for such a high-level model into a programming model for GPGPU devices, such as CUDA. We assume that the numerical values of some, or all, of the hardware characteristics of the targeted GPGPU device are unknown. Hence, these quantities are treated as symbols. Similarly, we would like that some, or all, of the program parameters remain symbols in the generated code.

In our implementation, we focus on one high-level model for accelerator programming, namely MetaFork. However, we believe that an adaptation to another high-level model for accelerator programming would not be difficult. One supporting reason for that claim is the fact that automatic code translation between the MetaFork and OpenMP languages can already be done within the MetaFork compilation framework, see [35].

The hardware characteristics of the GPGPU device can be the maximum number of registers to be allocated per thread in a thread-block and the maximum number of shared memory words to be allocated per thread in a thread-block. Similarly, the program parameters can be the number of threads per thread-block and the granularity of threads. For the generated code to be valid, hardware characteristics and program parameters need to satisfy constraints in the form of polynomial equations and inequalities. Moreover, applying code transformation (like optimization techniques) requires a case distinction based on the values of those symbols, as we saw with the example in the introduction.

In Section 7.2.1 we specify the required properties of the input code fragment S from the given MetaFork program, so that the comprehensive optimization algorithm, demonstrated in Section 7.1 can handle this MetaFork program. Section 7.2.2 discusses the procedure of comprehensive translation of the MetaFork program into parametric CUDA kernels, which yields the definition of comprehensive parametric CUDA kernels.

7.2.1 Input MetaFork Code Fragment

Consider a meta_schedule statement $M$ and its surrounding MetaFork program $P$. In this process of code analysis and transformation, we focus on the meta_schedule statement $M$ and assume that the rest of the program $P$ is serial C code. Hence, our examples, like the matrix vector multiplication and matrix addition examples, consist simply of a meta_schedule statement $M$ together with a few (possibly none) statements located before $M$ and initializing variables used in $M$.

Consider the meta_schedule statement $M$, that is, a statement of the form

\[ \text{meta_schedule } A \]

where $A$ is a compound statement of the form \{ $A_0A_1 \ldots A_\ell$ \} and each of $A_0, A_1, \ldots, A_\ell$ is a for-loop nest, such that:

1. each for-loop nest contains 2 or 4 meta_for loops; hence, it can be executed in a parallel fashion,
2. the body of the innermost loop can be any valid sequence of C statements; in particular, such a statement can be a for-loop, and

In practice, a parameter (in the sense of Definition [1]) of the meta_schedule statement $M$ is either a data parameter (that is, related to data being processed, like a number of rows or columns in a matrix) or a program parameter (that is, related to the division of the work among the threads executing the parallel for-loops). In Example [2], the variable $N$ is a data parameter, whereas the variables dim and $B$ are the program parameters.

Moreover, for the sake of clarity, we shall assume that the meta_schedule statement $M$ counts a single meta_for loop nest $A$. Extending the present section to the case where $M$ counts several meta_for loop nests can be done by existing techniques as we briefly explain now. Indeed, each meta_for loop nest can be handled separately. Then, “merging” the corresponding results can be done by techniques from symbolic computation, see [31, 32]. Therefore, we consider the serial elision (as defined in [35]) of $A$ in $M$ as the code fragment $S$.

Turning our attention back to Example [2], Figure 7.6 shows the serial elision of the MetaFork program on the right-hand portion of Figure 7.4.

```c
int dim = N / B, v, u;
// v is corresponding to the thread-block index
// u is corresponding to the thread index in a thread-block

// The following code is the serial elision
for (int i = 0; i < dim; ++i)
    for (int j = 0; j < B; ++j) {
        int p = v * B + u;
        int q = i * B + j;
        c[p] += a[p][q] * b[q];
    }
```

Figure 7.6: The serial elision of the MetaFork program for matrix vector multiplication

7.2.2 Comprehensive Translation into Parametric CUDA Kernels

Now, applying the comprehensive optimization algorithm (described in Section 7.1) on the serial elision $S$ of the meta_schedule statement $M$ (with prescribed resource evaluation functions, performance evaluation functions and optimization strategies), we obtain a sequence of processed quintuples of meta_schedule statements $Q_1(M), Q_2(M), \ldots, Q_\ell(M)$, which forms a comprehensive optimization in the sense of Definition 2.

If, as mentioned in the introduction of this chapter, PTX is used as intermediate representation (IR) then, for each $i = 1, \ldots, \ell$, under the constraints defined by the polynomial system associated with $Q_i(M)$, the IR code associated with $Q_i(M)$ is the translation in assembly language of a CUDA counterpart of $M$. In our implementation, we also translate to CUDA source code the MetaFork code in each $Q_i(M)$, since this is easier to read for a human being.

Therefore, in broad terms, a comprehensive translation of the meta_schedule statement $M$ into parametric CUDA kernels is a decision tree, where each edge holds a Boolean expres-
sion (given by a polynomial constraint) and each leaf is either a CUDA program in PTX form, or the symbol $\emptyset$, such that for each leaf $K$, with $K \neq \emptyset$, we have:

1. $K$ works correctly under the conjunction of the Boolean expressions located between the root node and the leaf, and
2. $K$ is semantically equivalent to $\mathcal{P}$.

The symbol $\emptyset$ is used to denote a situation (in fact, value ranges for the machine and program parameters) where no CUDA program equivalent to $\mathcal{P}$ is provided.

### 7.3 Implementation Details

In this section, we present a preliminary implementation of the comprehensive optimization algorithm demonstrated in Section 7.1. This implementation takes a given MetaFork program as input and is dedicated to the optimization of meta_schedule statements in view of generating parametric CUDA kernels.

For the algorithm stated in Section 7.1.7 to satisfy its specifications, one should use the PTX language for the IR. However, for simplicity, in our proof-of-concept implementation here, we use the IR of the LLVM compiler infrastructure, since the MetaFork compilation framework is based on Clang.

Two hardware resource counters are considered: register usage per thread and local/shared memory allocated per thread-block. No performance counters are specified; however, by design, the algorithm tries to minimize the usage of hardware resources. Four optimization strategies are used: (i) reducing register pressure, (ii) controlling thread granularity, (iii) common sub-expression elimination (CSE), and (iv) caching\(^2\) data in local/shared memory \([101]\). Details are given hereafter.

Figure 7.7 gives an overview of the software tools that are used for our implementation. Appendix C shows the implemented algorithms with these two resource counters and these three optimization strategies.

\[\text{Figure 7.7: The software tools involved for the implementation}\]

---

\(^2\text{In the MetaFork language, the keyword cache is used to indicate that every thread accessing a specified array a must copy in local/shared memory the data it accesses in a.}\]
Conversion between source code and CFG. **Clang** is used as the front-end for parsing the *MetaFork* source code and generating the source CFG. This latter is converted into a *Maple* DAG in order to take advantage of *Maple*’s capabilities for algebraic computation. Conversely, given a *Maple* version of a CFG, we can generate the corresponding *MetaFork* code by traversing this CFG.

**Register pressure.** Given the *MetaFork* source code, we use LLVM to generate the low-level machine instructions in the intermediate representation (IR), which are in a static single assignment (SSA) form. A benefit of using the SSA form is that one can calculate the live-ness sets without data flow analysis. Once the lifetime information is computed, we use the classical linear scan algorithm to estimate the register usage.

**Thread granularity.** A common method to improve arithmetic intensity and instruction level parallelism (ILP) is through controlling the granularity of threads, that is, each thread computing more than one of the output results. One can achieve this goal by adding a serial for loop within a thread. However, this method may increase the register usage as well as the amount of required shared memory (see the example in Figure 7.9). In the case that adding the granularity loop causes the needed resources to exceed the hardware limits, our algorithm applies an optimized strategy, “Granularity set to 1,” to remove that serial loop from the generated kernel code. We implement this strategy during the translation phase from the CFG to the *MetaFork* code by not generating this loop.

**Common sub-expression elimination (CSE).** For each basic block in the CFG, built from the *MetaFork* source code, we consider the basic blocks with more than one statement. Then, we use the `codegen[optimize]` package of *Maple*, such that CSE is applied to those statements and a sequence of new statements is generated. Finally, we update each basic block with those new statements. Moreover, the optimization technique has two levels: one using *Maple*’s default CSE algorithm and the other using the try-harder option of `codegen[optimize]`.

**Cache amount.** We take advantage of the PET (polyhedral extraction tool) to collect information related to the index expression of an array: occurring program parameters (defined as in Section 7.2), loop iteration counters and inequalities (that give the lower and upper bounds of those loop iteration counters). We now illustrate with an example for computing the amount of words that a thread-block requires. Consider the *MetaFork* program with the granularity for reversing a 1D array as shown on the left-hand portion of Figure 7.8. Note that w.r.t the *MetaFork* code, iteration counters **i**, **j** and **k** of for- (and meta_for-) loops are counted as neither program nor data parameters; thus, we shall consider them as bounded variables. However, in order to calculate the amount of words required per thread-block in the `RegularChains` library of *Maple*, we treat the iteration counter **i**, which indicates the thread-block index of the CUDA kernel, as a program parameter. The function call `ValueRangeWithConstraintsAndParameters` to the `RegularChains` library, as shown on the right-hand portion of Figure 7.8 is used to calculate the required words per thread-block for vector **c**. As a result, a range `[1, s*B+1]` is returned, such that we can determine that vector **c** accesses $s*B$ words per thread-block.

---

https://en.wikipedia.org/wiki/Live_variable_analysis
Towards Comprehensive Parametric CUDA Kernel Generation

// Data parameters: a, c, N
// Program parameters: B, s
int dim = N / (B * s);
meta_schedule {
  meta_for (int i = 0; i < dim; i++)
    meta_for (int j = 0; j < B; j++)
      for (int k = 0; k < s; ++k) {
        int x = i * B * s + k * B + j;
        int y = N - 1 - x
        c[y] = a[x];
      }
}

(a) MetaFork program with the granularity
(b) function call to RegularChains library

Figure 7.8: Computing the amount of words required per thread-block for reversing a 1D array

7.4 Experimentation

We present experimental results for the implementation described in Section 7.3. We consider six simple test cases: array reversal, matrix vector multiplication, 1D Jacobi, matrix addition, matrix transpose and matrix matrix multiplication. Recall that we consider two machine parameters: the amount \( Z_B \) of shared memory per streaming multiprocessor (SM) and the maximum number \( R_B \) of registers per thread in a thread-block.

Three scenarios of optimized MetaFork programs based on different systems of constraints are generated by our implementation of the comprehensive optimization algorithm. The first case of optimized MetaFork programs uses the shared memory and a granularity parameter \( s \). The second case uses the shared memory but sets the granularity parameter \( s \) to 1. The third case removes the cache keyword and sets \( s \) to 1. In this latter case, the amount of words read and written per thread-block is more than the maximum amount \( Z_B \) of shared memory per SM. However, the cache keyword is not implemented in the MetaFork compilation framework yet, so that we manually process editing for this keyword. For each of these optimization strategies, we use a shortened code shown in Table 7.1.

<table>
<thead>
<tr>
<th>Strategy name</th>
<th>Its code</th>
<th>Strategy name</th>
<th>Its code</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Accept register pressure”</td>
<td>(1)</td>
<td>“CSE applied”</td>
<td>(2)</td>
</tr>
<tr>
<td>“No granularity reduction”</td>
<td>(3a)</td>
<td>“Granularity set to 1”</td>
<td>(3b)</td>
</tr>
<tr>
<td>“Accept caching”</td>
<td>(4a)</td>
<td>“Refuse caching”</td>
<td>(4b)</td>
</tr>
</tbody>
</table>

Array reversal. Our comprehensive optimization algorithm applies to the source code the following optimization strategy codes (1) (4a) (3a) (2) (2). For the first case, it generates the optimized MetaFork code shown in Figure 7.9.

Applying optimization strategy codes in a sequence either (1) (3b) (4a) (3a) (2) (2) or (2) (2) (3b) (1) (4a) (3a), the second case generates the optimized MetaFork code shown in Figure 7.10.

---

4 In the MetaFork language, the intention of using the keyword cache is to indicate that every thread accessing a specified array a must copy in local/shared memory the data it accesses in a.
7.4. Experimentation

Constraints:

\[
\begin{align*}
2sB & \leq Z_B \\
4 & \leq R_B
\end{align*}
\]

strategies (1) (4a) (3a) (2) (2) applied

int \(N, s, B, \text{dim} = \frac{N}{(s^*B)}\);
int \(a[N], c[N]\);
meta_schedule cache(a, c) {
    meta_for (int i = 0; i < \text{dim}; i++)
    meta_for (int j = 0; j < B; j++)
    for (int k = 0; k < s; ++k) {
        int x = (i*s+k)*B+j;
        int y = N-1-x;
        c[y] = a[x];
    }
}

Figure 7.9: The first case of the optimized MetaFork code for array reversal

Constraints:

\[
\begin{align*}
2B & \leq Z_B < 2sB \\
3 & \leq R_B
\end{align*}
\]

strategies (1) (3b) (4a) (3a) (2) (2) applied

or

\[
\begin{align*}
2B & \leq Z_B < 2sB \\
3 & \leq R_B < 4
\end{align*}
\]

strategies (2) (2) (3b) (1) (4a) (3a) applied

Figure 7.10: The second case of the optimized MetaFork code for array reversal

Applying optimization strategy codes in a sequence either (1) (3b) (2) (2) (4b) or (2) (2) (3b) (1) (4b), the third case generates the optimized MetaFork code shown in Figure 7.11

Constraints:

\[
\begin{align*}
Z_B & < 2B \\
3 & \leq R_B
\end{align*}
\]

strategies (1) (3b) (2) (2) (4b) applied

or

\[
\begin{align*}
Z_B & < 2B \\
3 & \leq R_B < 4
\end{align*}
\]

strategies (2) (2) (3b) (1) (4a) (3a) applied

Figure 7.11: The third case of the optimized MetaFork code for array reversal

**Matrix vector multiplication.** Applying optimization strategy codes in a sequence either (1) (4a) (3a) (2) (2) or (2) (1) (4a) (3a) (2), the first case generates the optimized MetaFork code shown in Figure 7.12.

Applying the optimization strategy codes in a sequence either (1) (3b) (4a) (3b) (2) (2), (2) (1) (3b) (4a) (3a) (2) or (2) (2) (3b) (1) (4a) (3a), the second case generates the optimized MetaFork code shown in Figure 7.13.

Applying the optimization strategy codes in a sequence either (1) (3b) (2) (2) (4b), (2) (1)
Constraints:  
\[
\begin{aligned}
& sB^2 + sB + B \leq Z_B \\
& 8 \leq R_B
\end{aligned}
\]
strategies (1) (4a) (3a) (2) (2) applied 

or  
\[
\begin{aligned}
& sB^2 + sB + B \leq Z_B \\
& 8 \leq R_B < 9
\end{aligned}
\]
strategies (2) (1) (4a) (3a) (2) applied

```c
int N, s, B, dim0 = N/(s*B), dim1 = N/B;
int a[N][N], b[N], c[N];
meta_schedule cache(a, b, c) {
    meta_for (int v = 0; v < dim0; v++)
        meta_for (int i = 0; i < dim1; i++)
            for (int j = 0; j < B; ++j)
                for (int k = 0; k < s; ++k) {
                    int p = (v*s+k)*B+u;
                    int q = i*B+j;
                    c[p] = a[p][q]*b[q]+c[p];
                }
}
```

Figure 7.12: The first case of the optimized MetaFork code for matrix vector multiplication

Constraints:  
\[
\begin{aligned}
& B^2 + 2B \leq Z_B < sB^2 + sB + B \\
& 7 \leq R_B
\end{aligned}
\]
strategies (1) (3b) (4a) (3a) (2) (2) applied 

or  
\[
\begin{aligned}
& B^2 + 2B \leq Z_B < sB^2 + sB + B \\
& 7 \leq R_B < 9
\end{aligned}
\]
strategies (2) (1) (3b) (4a) (3a) (2) applied 

or  
\[
\begin{aligned}
& B^2 + 2B \leq Z_B < sB^2 + sB + B \\
& 7 \leq R_B < 8
\end{aligned}
\]
strategies (2) (2) (3b) (1) (4a) (3a) applied

Figure 7.13: The second case of the optimized MetaFork code for matrix vector multiplication

(3b) (2) (4b) or (2) (2) (3b) (1) (4b), the third case generates the optimized MetaFork code shown in Figure 7.14.

1D Jacobi. Given 1D Jacobi source code written in MetaFork, shown in Figure 7.15, the CSE strategy is applied successfully for all cases of optimized MetaFork programs. This example requires post-processing for calculating the total amount of required shared memory per thread-block, due to the fact that array \( a \) has multiple accesses and that each access has a different index.

Applying the optimization strategy codes in a sequence (1) (4a) (3a) (2) (2), the first case generates the optimized MetaFork code shown in Figure 7.16.

Applying the optimization strategy codes in a sequence (1) (3b) (4a) (3a) (2) (2), the second case generates the optimized MetaFork code shown in Figure 7.17.

Applying the optimization strategy codes in a sequence (1) (3b) (2) (2) (4b), the third case generates the optimized MetaFork code shown in Figure 7.18.

Matrix addition. Due to the limitation in the codegen[optimize] package of Maple, the CSE optimizer could not handle a two-dimensional array on the left-hand side of assignments.
7.4. Experimentation

Constraints:
\[
\begin{align*}
Z_B < B^2 + 2B \\
7 \leq R_B
\end{align*}
\]
strategies (1) (3b) (2) (2) (4b) applied

or

\[
\begin{align*}
Z_B < B^2 + 2B \\
7 \leq R_B < 9
\end{align*}
\]
strategies (2) (1) (3b) (2) (4b) applied

or

\[
\begin{align*}
Z_B < B^2 + 2B \\
7 \leq R_B < 8
\end{align*}
\]
strategies (2) (2) (3b) (1) (4b) applied

Figure 7.14: The third case of the optimized MetaFork code for matrix vector multiplication

```
int N, s = 1, B, dim0 = N/(s*B), dim1 = N/B;
int a[N][N], b[N], c[N];
meta_schedule {
    meta_for (int v = 0; v < dim0; v++)
        for (int i = 0; i < dim1; i++)
            for (int j = 0; j < B; ++j) {
                int p = v*B+u;
                int q = i*B+j;
                c[p] = a[p][q]*b[q]+c[p];
            }
}
```

Figure 7.15: The MetaFork source code for 1D Jacobi

Thus, we use a one-dimensional array to represent the output matrix. Applying the optimization strategy codes in a sequence (1) (4a) (3a) (2) (2), the first case generates the optimized MetaFork code shown in Figure 7.19.

Applying the optimization strategy codes in a sequence either (1) (3b) (4a) (3a) (2) (2) or (2) (2) (3b) (1) (4a) (3a), the second case generates the optimized MetaFork code shown in Figure 7.20.

Applying the optimization strategy codes in a sequence either (1) (3b) (2) (2) (4b) or (2) (2) (3b) (1) (4b), the third case generates the optimized MetaFork code shown in Figure 7.21.

Matrix transpose. Applying the optimization strategy codes in a sequence (1) (4a) (3a)
int T, N, s, B, dim = (N-2)/(s*B);
int a[2*N];
for (int t = 0; t < T; ++t)
    meta_schedule cache(a) {
        meta_for (int i = 0; i < dim; i++)
        meta_for (int j = 0; j < B; j++)
            for (int k = 0; k < s; ++k) {
                int p = j+(i*s+k)*B;
                int t16 = p+1;
                int t15 = p+2;
                int p1 = t16;
                int p2 = t15;
                int np = N+p;
                int np1 = N+t16;
                int np2 = N+t15;
                if (t % 2)
                    a[p1] = (a[np]+a[np1]+a[np2])/3;
                else
                    a[np1] = (a[p]+a[p1]+a[p2])/3;
            }
    }

Constraints:
\[
\begin{align*}
2sB + 2 & \leq Z_B \\
9 & \leq R_B
\end{align*}
\]

strategies (1) (4a) (3a) (2) (2) applied

Figure 7.16: The first case of the optimized MetaFork code for 1D Jacobi

(2) (2), the first case generates the optimized MetaFork code shown in Figure 7.22.

Applying the optimization strategy codes in a sequence either (1) (3b) (4a) (3a) (2) (2) or (2) (2) (3b) (1) (4a) (3a), the second case generates the optimized MetaFork code shown in Figure 7.23.

Applying the optimization strategy codes in a sequence either (1) (3b) (2) (2) (4b) or (2) (2) (3b) (1) (4b), the third case generates the optimized MetaFork code shown in Figure 7.24.
7.4. Experimentation

Constraints:
\[
\begin{align*}
Z_B & < 2B + 2 \\
9 & \leq R_B
\end{align*}
\]
strategies (1) (3b) (2) (2) (4b) applied

```cpp
int T, N, s = 1, B, dim = (N-2)/(s*B);
int a[2*N];
for (int t = 0; t < T; ++t)
    meta_schedule {
        meta_for (int i = 0; i < dim; i++)
            meta_for (int j = 0; j < B; j++) {
                int p = j+i*B;
                int t16 = p+1;
                int t15 = p+2;
                int pl = t16;
                int p2 = t15;
                int np = N+p;
                if (t % 2)
                    a[p1] = (a[np]+a[np1]+a[np2])/3;
                else
                    a[np1] = (a[p]+a[p1]+a[p2])/3;
            }
    }
```

Figure 7.18: The third case of the optimized MetaFork code for 1D Jacobi

```cpp
int N, B0, B1, s, dim0 = N/B0, dim1 = N/(B1*s);
int a[N][N], b[N][N], c[N*N];
meta_schedule cache(a, b, c) {
    meta_for (int v0 = 0; v0 < dim0; v0++)
        meta_for (int v1 = 0; v1 < dim1; v1++)
            meta_for (int u0 = 0; u0 < B0; u0++)
                meta_for (int u1 = 0; u1 < B1; u1++)
                    for (int k = 0; k < s; ++k) {
                        int i = v0*B0+u0;
                        int j = (v1*s+k)*B1+u1;
                        c[i*N+j] = a[i][j] + b[i][j];
                    }
}
```

Figure 7.19: The first case of the optimized MetaFork code for matrix addition

**Matrix matrix multiplication.** Applying the optimization strategy codes in a sequence (1) (4a) (3a) (2) (2), the first case generates the optimized MetaFork code shown in Figure 7.25.

Applying the optimization strategy codes in a sequence either (1) (3b) (4a) (3a) (2) (2) or (2) (2) (3b) (1) (4a) (3a), the second case generates the optimized MetaFork code shown in Figure 7.26.

Applying the optimization strategy codes in a sequence either (1) (3b) (2) (2) (4b) or (2) (2) (3b) (1) (4b), the third case generates the optimized MetaFork code shown in Figure 7.27.
Constraints:
\[
\begin{align*}
3B_0B_1 &\leq Z_B < 3sB_0B_1 \\
6 &\leq R_B
\end{align*}
\]

strategies (1) (3b) (4a) (3a) (2) (2) applied

or

\[
\begin{align*}
3B_0B_1 &\leq Z_B < 3sB_0B_1 \\
6 &\leq R_B < 7
\end{align*}
\]

strategies (2) (2) (3b) (1) (4a) (3a) applied

Figure 7.20: The second case of the optimized MetaFork code for matrix addition

Constraints:
\[
\begin{align*}
Z_B &< 3B_0B_1 \\
6 &\leq R_B
\end{align*}
\]

strategies (1) (3b) (2) (2) (4b) applied

or

\[
\begin{align*}
Z_B &< 3B_0B_1 \\
6 &\leq R_B < 7
\end{align*}
\]

strategies (2) (2) (3b) (1) (4b) applied

Figure 7.21: The third case of the optimized MetaFork code for matrix addition

Constraints:
\[
\begin{align*}
2sB_0B_1 &\leq Z_B \\
6 &\leq R_B
\end{align*}
\]

strategies (1) (4a) (3a) (2) (2) applied

Figure 7.22: The first case of the optimized MetaFork code for matrix transpose

7.5 Conclusion

In this chapter, we proposed a comprehensive optimization algorithm that optimizes the input code fragment depending on unknown machine and program parameters; meanwhile, we re-
7.5. Conclusion

Constraints:
\[
\begin{cases}
2B_0B_1 \leq Z_B < 2sB_0B_1 \\
5 \leq R_B
\end{cases}
\]

strategies (1) (3b) (4a) (3a) (2) (2) applied

or

\[
\begin{cases}
2B_0B_1 \leq Z_B < 2sB_0B_1 \\
5 \leq R_B < 6
\end{cases}
\]

strategies (2) (2) (3b) (1) (4a) (3a) applied

int N, B0, B1, s = 1, dim0 = N/B0, dim1 = N/(B1*s);
int a[N][N], c[N*N];

meta_schedule cache(a, c) {
    meta_for (int v0 = 0; v0 < dim0; v0++)
        meta_for (int v1 = 0; v1 < dim1; v1++)
            meta_for (int u0 = 0; u0 < B0; u0++)
                meta_for (int u1 = 0; u1 < B1; u1++) {
                    int i = v0*B0+u0;
                    int j = v1*B1+u1;
                    c[j*N+i] = a[i][j];
                }
}

Figure 7.23: The second case of the optimized MetaFork code for matrix transpose

Constraints:
\[
\begin{cases}
Z_B < 2B_0B_1 \\
5 \leq R_B
\end{cases}
\]

strategies (1) (3b) (2) (2) (4b) applied

or

\[
\begin{cases}
Z_B < 2B_0B_1 \\
5 \leq R_B < 6
\end{cases}
\]

strategies (2) (2) (3b) (1) (4b) applied

int N, B0, B1, s = 1, dim0 = N/B0, dim1 = N/(B1*s);
int a[N][N], c[N*N];

meta_schedule {
    meta_for (int v0 = 0; v0 < dim0; v0++)
        meta_for (int v1 = 0; v1 < dim1; v1++)
            meta_for (int u0 = 0; u0 < B0; u0++)
                meta_for (int u1 = 0; u1 < B1; u1++) {
                    int i = v0*B0+u0;
                    int j = v1*B1+u1;
                    c[j*N+i] = a[i][j];
                }
}

Figure 7.24: The third case of the optimized MetaFork code for matrix transpose

alized a proof-of-concept implementation for generating comprehensive parametric MetaFork programs, in the form of a case distinction based on the possible values of the machine and program parameters.

The comprehensive optimization algorithm that we proposed takes optimization strategies, resource counters and performance counters into account; we implemented two resource counters and four optimization strategies in this comprehensive optimization algorithm.

With this preliminary implementation, experimentation shows that given a MetaFork program, three scenarios of optimized MetaFork programs are generated, each of them with a system of constraints specifying when the corresponding code is valid.

In addition, from the experimental results, we observe that different sequences of the optimization strategies yield the same optimized MetaFork program. However, since some optimization strategies are applied to the intermediate representation of the source code, the corresponding improvements are not shown in Section 7.4.
Towards Comprehensive Parametric CUDA Kernel Generation

Constraints:

\[
\begin{align*}
& \{ sB_0B_1 + sBB_1 + B_0B \leq Z_B \\ & 9 \leq R_B \}
\end{align*}
\]

strategies (1) (4a) (3a) (2) (2) applied

Figure 7.25: The first case of the optimized MetaFork code for matrix matrix multiplication

Constraints:

\[
\begin{align*}
& \{ B_0B_1 + BB_1 + B_0B \leq Z_B < sB_0B_1 + sBB_1 + B_0B \\ & 8 \leq R_B \}
\end{align*}
\]

strategies (1) (3b) (4a) (3a) (2) (2) applied

\[
\begin{align*}
& \{ B_0B_1 + BB_1 + B_0B \leq Z_B < sB_0B_1 + sBB_1 + B_0B \\ & 8 \leq R_B < 9 \}
\end{align*}
\]

strategies (2) (2) (3b) (1) (4a) (3a) applied

Figure 7.26: The second case of the optimized MetaFork code for matrix matrix multiplication
Constraints:
\[
\begin{cases}
Z_B < B_0B_1 + BB_1 + B_0B \\
8 \leq R_B
\end{cases}
\]

strategies (1) (3b) (2) (2) (4b) applied

or

\[
\begin{cases}
Z_B < B_0B_1 + BB_1 + B_0B \\
8 \leq R_B < 9
\end{cases}
\]

strategies (2) (2) (3b) (1) (4b) applied

int N, B0, B1, s = 1, dim1 = N/(B1*s);
int dim0 = N/B0, B = min(B0, B1), dim = N/B;
int a[N][N], b[N][N], c[N*N];

meta_schedule {
  meta_for (int v0 = 0; v0 < dim0; v0++)
    meta_for (int v1 = 0; v1 < dim1; v1++)
      for (int w = 0; w < dim; w++)
        meta_for (int u0 = 0; u0 < B0; u0++)
          meta_for (int u1 = 0; u1 < B1; u1++)
          {
            int i = v0*B0+u0;
            int j = v1*B1+u1;
            for (int z = 0; z < B; z++)
              { 
                int p = B*w+z;
                c[i*N+j] = c[i*N+j] + a[i][p] * b[p][j];
              }
          }
  }
}

Figure 7.27: The third case of the optimized MetaFork code for matrix matrix multiplication
Chapter 8

Concluding Remarks

As of today, the publicly available and latest release of MetaFork, see www.metafork.org, offers all the features stated in the abstract. To be more specific, MetaFork is a meta-language for concurrency platforms, based on the fork-join model and pipelining parallelism, which targets multi-core architectures. This meta-language forms a bridge between actual multi-threaded programming languages and we use it to perform automatic code translation between those languages, which, currently consist of CilkPlus and OpenMP. The experimental results in Section 3.7 show that, first of all, the translators can faithfully translate programs written in one supported language to another. Secondly, translation can preserve (or improve) the performance of the translated programs for fork-join programs. Third, translators are robust in the sense that they can translate large projects and not just a simple test file. Indeed, we can successfully translate the BOTS benchmark.

In Chapter 4, from an input MetaFork program we generate parametric CUDA kernels, that is, CUDA kernels for which program parameters (like number of threads per block) and machine parameters (like shared memory size) are symbols. Hence, the values of these parameters need not to be known at code-generation-time: machine parameters and program parameters are determined when the generated code is installed on the target machine. The experimental results show that these parametric CUDA programs can yield higher performance than non-parametric CUDA programs obtained by other tools (namely PPCG) for C-to-CUDA automatic parallelization.

Moreover, for the fourth feature as mentioned in the abstract, a proof-of-concept implementation dedicated to the generation of comprehensive optimized MetaFork programs from an input MetaFork program is realized. While going from MetaFork to MetaFork is certainly not our final goal, turning an un-optimized MetaFork program into optimized MetaFork programs can be seen as a first approximation of our final goal, that is, generating optimized parametric CUDA kernels from input un-optimized annotated C/C++ programs, in the form of a case discussion, based on the possible values of the machine and program parameters. In future, we shall integrate this mechanism into the MetaFork compilation framework.

Finally, in Chapter 6 we have addressed the problem of developing portable high-level programming language extensions, by using directive based programming, that take advantage of novel accelerators. We illustrate the implementation of our MetaFork compilation framework which is built on top of the modern compiler product CLANG for parsing and generating AST from input programs. We emphasize the fact that the compilation of our MetaFork frame-
work is independent of that of LLVM/CLANG. In other words, our MetaFork framework is a standalone tool without modifying any LLVM/CLANG source code.
Bibliography


Appendix A

Code Translation Examples

Listing A.1: A parallel region of OpenMP code

```c
int pairalign()
{
  int i, n, m, si, sj;
  int len1, len2, maxres;
  double gg, mm_score;
  int *mat_xref, *matptr;

  matptr = gon250mt;
  mat_xref = def_aa_xref;
  maxres = get_matrix(matptr, mat_xref, 10);
  if (maxres == 0) return(-1);

  bots_message("Start aligning ");

  #pragma omp parallel
  {
    #pragma omp single private(i,n,si,sj,len1,m)
    for (si = 0; si < nseqs; si++) {
      n = seqlen_array[si+1];
      for (i = 1, len1 = 0; i <= n; i++) {
        char c = seq_array[si+1][i];
        if ((c != gap_pos1) && (c != gap_pos2)) len1++;
      }
      for (sj = si + 1; sj < nseqs; sj++)
      { m = seqlen_array[sj+1];
        if ( n == 0 || m == 0 ) {
          bench_output[si*nseqs+sj] = (int) 1.0;
        } else {
          #pragma omp task untied \ private(i,gg,len2,mm_score) firstprivate(m,n,si,sj,len1) \ shared(nseqs, bench_output,seqlen_array,seq_array,gap_pos1,
```


```c

int se1, se2, sb1, sb2, maxscore, seq1, seq2, g, gh;
int displ[2*MAX_ALN_LENGTH+1];
int print_ptr, last_print;

for (i = 1, len2 = 0; i <= m; i++) {
    char c = seq_array[sj+1][i];
    if ((c != gap_pos1) && (c != gap_pos2)) len2++;
}

if ( dnaFlag == TRUE ) {
    g = (int) ( 2 * INT_SCALE * pw_go_penalty * gap_open_scale );  // gapOpen
    gh = (int) (INT_SCALE * pw_ge_penalty * gap_extend_scale);  // gapExtend
}
else {
    gg = pw_go_penalty + log((double) MIN(n, m));  // temporary value
    g = (int) ((mat_avscore <= 0) ? (2 * INT_SCALE * gg) : (2 * mat_avscore * gg * gap_open_scale ));  // gapOpen
    gh = (int) (INT_SCALE * pw_ge_penalty);  // gapExtend
}

seq1 = si + 1;
seq2 = sj + 1;

forward_pass(&seq_array[seq1][0], &seq_array[seq2][0], n, m, &se1, &se2, &maxscore, g, gh);
reverse_pass(&seq_array[seq1][0], &seq_array[seq2][0], se1, se2, &sb1, &sb2, maxscore, g, gh);

print_ptr = 1;
last_print = 0;

diff(sb1-1, sb2-1, se1-sb1+1, se2-sb2+1, 0, 0, &print_ptr, &last_print, displ, seq1, seq2, g, gh);
mm_score = tracepath(sb1, sb2, &print_ptr, displ, seq1, seq2);

if (len1 == 0 || len2 == 0) mm_score = 0.0;
else mm_score /= (double) MIN(len1, len2);

bench_output[si*nseqs+sj] = (int) mm_score;
}
```

```c
}  // end task
```

```c
}  // end if (n == 0 || m == 0)
```

```c
}  // for (j)
```

```c
}  // end parallel for (i)
```

```c
}  // end parallel
```
Listing A.2: A parallel region of MetaFork code translated from Listing A.1

```c
int pairalign()
{
    int i, n, m, si, sj;
    int len1, len2, maxres;
    double gg, mm_score;
    int *mat_xref, *matptr;

    matptr = gon250mt;
    mat_xref = def_aa_xref;
    maxres = get_matrix(matptr, mat_xref, 10);
    if (maxres == 0) return(-1);

    bots_message("Start aligning ");

    {  
        meta_fork shared( gap_pos2, gap_pos1, bench_output, nseqs, seq_array,
                         seqlen_array) {
            int len1;
            int i;
            int m;
            int n;
            int sj;
            int si;

            for (si = 0; si < nseqs; si++) {
                n = seqlen_array[si+1];
                for (i = 1, len1 = 0; i <= n; i++) {
                    char c = seq_array[si+1][i];
                    if ((c != gap_pos1) && (c != gap_pos2)) len1++;
                }
                for (sj = si + 1; sj < nseqs; sj++)
                {
                    m = seqlen_array[sj+1];
                    if ( n == 0 || m == 0 ) {
                        bench_output[si*nseqs+sj] = (int) 1.0;
                    } else {
                        meta_fork shared( gap_open_scale, pw_ge_penalty, gap_pos2,
                                         dnaFlag, bench_output, gap_extend_scale, gap_pos1,
                                         mat_avscore, nseqs, seq_array, seqlen_array, pw_go_penalty)
                    }
                }
            }
        }
    }
}
```
```c
for (i = 1, len2 = 0; i <= m; i++) {
    char c = seq_array[sj+1][i];
    if ((c != gap_pos1) && (c != gap_pos2)) len2++;
}

if ( dnaFlag == TRUE ) {
    g = (int) ( 2 * INT_SCALE * pw_go_penalty * gap_open_scale ) ; // gapOpen
    gh = (int) (INT_SCALE * pw_ge_penalty * gap_extend_scale); //gapExtend
} else {
    gg = pw_go_penalty + log((double) MIN(n, m)); // temporary value
    g = (int) ((mat_avscore <= 0) ? (2 * INT_SCALE * gg) : (2 * mat_avscore * gg * gap_open_scale) ); // gapOpen
    gh = (int) (INT_SCALE * pw_ge_penalty); //gapExtend
}

seq1 = si + 1;
seq2 = sj + 1;

forward_pass(&seq_array[seq1][0], &seq_array[seq2][0], n, m, &se1, &se2, &maxscore, g, gh);
reverse_pass(&seq_array[seq1][0], &seq_array[seq2][0], se1, se2, &sb1, &sb2, maxscore, g, gh);

print_ptr = 1;
last_print = 0;
diff(sb1-1, sb2-1, se1-sb1+1, se2-sb2+1, 0, 0, &print_ptr, &last_print, displ, seq1, seq2, g, gh);
mm_score = tracepath(sb1, sb2, &print_ptr, displ, seq1, seq2);
if (len1 == 0 || len2 == 0) mm_score = 0.0;
else mm_score /= (double) MIN(len1,len2);
bench_output[si*nseqs+sj] = (int) mm_score;
```

static void * _taskFunc4(void * __tdata2)
{
    struct __taskenv__0 {
        int len1;
        int i;
        double gg;
        double mm_score;
        int len2;
        int n;
        int m;
        int si;
        int sj;
    } ;

    struct __taskenv__0 * _tenv1 = ( struct __taskenv__0 *) __tdata2;

    int len1 = _tenv1->len1;
    int i = _tenv1->i;
    double gg = _tenv1->gg;
    double mm_score = _tenv1->mm_score;
    int len2 = _tenv1->len2;
    int n = _tenv1->n;
    int m = _tenv1->m;
    int si = _tenv1->si;
    int sj = _tenv1->sj;

    {
        int se1, se2, sb1, sb2, maxscore, seq1, seq2, g, gh;
        int displ[10001];
        int print_ptr, last_print;
        for (i = 1 , len2 = 0; i <= m; i++) {
            char c = seq_array[sj + 1][i];
            if ((c ! = gap_pos1) && (c ! = gap_pos2))
                len2++;
        }
    }
}
```c
if (dnaFlag == 1) {
    g = (int)(2 * 100 * pw_go_penalty * gap_open_scale);
    gh = (int)(100 * pw_ge_penalty * gap_extend_scale);
} else {
    gg = pw_go_penalty + log((double)((n) < (m) ? (n) : (m)));
    g = (int)((mat_avscore <= 0) ? (2 * 100 * gg) : (2 * mat_avscore * gg * gap_open_scale));
    gh = (int)(100 * pw_ge_penalty);
}
seq1 = si + 1;
seq2 = sj + 1;
forward_pass(&seq_array[seq1][0], &seq_array[seq2][0], n, m, &se1, &se2, &maxscore, g, gh);
reverse_pass(&seq_array[seq1][0], &seq_array[seq2][0], se1, se2, &sb1, &sb2, maxscore, g, gh);
print_ptr = 1;
last_print = 0;
diff(sb1 - 1, sb2 - 1, se1 - sb1 + 1, se2 - sb2 + 1, 0, 0, &print_ptr, &last_print, displ, seq1, seq2);
mm_score = tracepath(sb1, sb2, &print_ptr, displ, seq1, seq2);
if (len1 == 0 || len2 == 0)
    mm_score = 0.;
else
    mm_score /= (double)((len1) < (len2) ? (len1) : (len2));
bench_output[si * nseqs + sj] = (int)mm_score;
}

int pairalign()
{
    int i, n, m, si, sj;
    int len1, len2, maxres;
    double gg, mm_score;
    int *mat_xref, *matptr;
    matptr = gon250mt;
    mat_xref = def_aa_xref;
    maxres = get_matrix(matptr, mat_xref, 10);
    if (maxres == 0) return(-1);
    bots_message("Start aligning ");
    {
        meta_fork shared( gap_pos2, gap_pos1, bench_output, seqlen_array, nseqs,
```
seq_array) {
int i;
int len1;
int sj;
int si;
int m;
int n;

for (si = 0; si < nseqs; si++) {
    n = seqlen_array[si+1];
    for (i = 1, len1 = 0; i <= n; i++) {
        char c = seq_array[si+1][i];
        if ((c != gap_pos1) && (c != gap_pos2)) len1++;
    }
    for (sj = si + 1; sj < nseqs; sj++) {
        m = seqlen_array[sj+1];
        if (n == 0 || m == 0) {
            bench_output[si*nseqs+sj] = (int) 1.0;
        } else {
            struct __taskenv__0 {
                int len1;
                int i;
                double gg;
                double mm_score;
                int len2;
                int n;
                int m;
                int si;
                int sj;
            } * _tenv1;

            _tenv1 = (struct __taskenv__0*) malloc(sizeof(struct __taskenv__0));
            _tenv1->len1 = len1;
            _tenv1->i = i;
            _tenv1->gg = gg;
            _tenv1->mm_score = mm_score;
            _tenv1->len2 = len2;
            _tenv1->n = n;
            _tenv1->m = m;
            _tenv1->si = si;
            _tenv1->sj = sj;
            cilk_spawn _taskFunc4(_tenv1);
        }
    }
}
// end task
Listing A.4: A parallel region of CilkPlus code translated from Listing A.3

```c
static void *_taskFunc4(void *__tdata2) {
struct __taskenv__0 {
   int len2;
   int sj;
   double mm_score;
   double gg;
   int si;
   int m;
   int n;
   int len1;
   int i;
};
struct __taskenv__0 * _tenv1 = (struct __taskenv__0 *)__tdata2;
int len2 = _tenv1->len2;
int sj = _tenv1->sj;
double mm_score = _tenv1->mm_score;
double gg = _tenv1->gg;
int si = _tenv1->si;
int m = _tenv1->m;
int n = _tenv1->n;
int len1 = _tenv1->len1;
int i = _tenv1->i;
{
    if (n == 0 || m == 0) {
        for (j) {
            cilk_sync ;
        }
    }
    bots_message(" completed!
");
    return 0;
}
```

Listing A.4: A parallel region of CilkPlus code translated from Listing A.3

```c
static void * _taskFunc4(void * __tdata2) {
struct __taskenv__0 {
   int len2;
   int sj;
   double mm_score;
   double gg;
   int si;
   int m;
   int n;
   int len1;
   int i;
};
struct __taskenv__0 * _tenv1 = ( struct __taskenv__0 * ) __tdata2;
int len2 = _tenv1->len2;
int sj = _tenv1->sj;
double mm_score = _tenv1->mm_score;
double gg = _tenv1->gg;
int si = _tenv1->si;
int m = _tenv1->m;
int n = _tenv1->n;
int len1 = _tenv1->len1;
int i = _tenv1->i;
{
    if (n == 0 || m == 0) {
        for (j) {
            cilk_sync ;
        }
    }
    bots_message(" completed!
");
    return 0;
}
```
int se1, se2, sb1, sb2, maxscore, seq1, seq2, g, gh;
int displ[10001];
int print_ptr, last_print;
for (i = 1, len2 = 0; i <= m; i++) {
    char c = seq_array[sj + 1][i];
    if ((c != gap_pos1) && (c != gap_pos2))
        len2++;
}
if (dnaFlag == 1) {
    g = (int)(2 * 100 * pw_go_penalty * gap_open_scale);
    gh = (int)(100 * pw_ge_penalty * gap_extend_scale);
} else {
    gg = pw_go_penalty + log(((double)((n) < (m) ? (n) : (m))));
    g = (int)((mat_avscore <= 0) ? (2 * 100 * gg) : (2 * mat_avscore *
        gg * gap_open_scale));
    gh = (int)(100 * pw_ge_penalty);
}
seq1 = si + 1;
seq2 = sj + 1;
forward_pass(&seq_array[seq1][0], &seq_array[seq2][0], n, m, &se1, &se2,
    &maxscore, g, gh);
reverse_pass(&seq_array[seq1][0], &seq_array[seq2][0], se1, se2, &sb1,
    &sb2, maxscore, g, gh);
print_ptr = 1;
last_print = 0;
diff(sb1 - 1, sb2 - 1, se1 - sb1 + 1, se2 - sb2 + 1, 0, 0, &print_ptr,
    &last_print, displ, seq1, seq2, g, gh);
mm_score = tracepath(sb1, sb2, &print_ptr, displ, seq1, seq2);
if (len1 == 0 || len2 == 0)
    mm_score = 0.;
else
    mm_score /= (double)((len1) < (len2) ? (len1) : (len2));
bench_output[si * nseqs + sj] = (int)mm_score;
}
}
static void * _taskFunc5(void * __tdata2)
{
    struct __taskenv__0 {
        double gg;
        double mm_score;
        int len2;
    } ;
    struct __taskenv__0 * _tenv1 = ( struct __taskenv__0 *) __tdata2;
    double gg = _tenv1->gg;
    double mm_score = _tenv1->mm_score;
```c
int len2 = _tenv1->len2;

{ int len1;
int i;
int m;
int n;
int sj;
int si;

for (si = 0; si < nseqs; si++) {
    n = seqlen_array[si + 1];
    for (i = 1, len1 = 0; i <= n; i++) {
        char c = seq_array[si + 1][i];
        if ((c != gap_pos1) && (c != gap_pos2))
            len1++;
    }
    for (sj = si + 1; sj < nseqs; sj++) {
        m = seqlen_array[sj + 1];
        if (n == 0 || m == 0) {
            bench_output[si * nseqs + sj] = (int)1.;
        } else {
            struct __taskenv__0 {
                int len2;
                int sj;
                double mm_score;
                double gg;
                int si;
                int m;
                int n;
                int len1;
                int i;
            } *_tenv1;
            _tenv1 = (struct __taskenv__0 *)malloc(sizeof(struct __taskenv__0));
            _tenv1->len2 = len2;
            _tenv1->sj = sj;
            _tenv1->mm_score = mm_score;
            _tenv1->gg = gg;
            _tenv1->si = si;
            _tenv1->m = m;
            _tenv1->n = n;
            _tenv1->len1 = len1;
            _tenv1->i = i;
            cilk_spawn _taskFunc4(_tenv1);
        }
    }
}
```
int pairalign()
{
    int i, n, m, si, sj;
    int len1, len2, maxres;
    int *mat_xref, *matptr;
    matptr = gon250mt;
    mat_xref = def_aa_xref;
    maxres = get_matrix(matptr, mat_xref, 10);
    if (maxres == 0) return(-1);
    bots_message("Start aligning ");
    {
        struct __taskenv__0 {
            double gg;
            double mm_score;
            int len2;
        } * _tenv1;
        _tenv1 = ( struct __taskenv__0*) malloc(sizeof(struct __taskenv__0));
        _tenv1->gg = gg;
        _tenv1->mm_score = mm_score;
        _tenv1->len2 = len2;
        cilk_spawn _taskFunc5(_tenv1);
        cilk_sync;
        // end parallel for (i)
    }
    // end parallel
    bots_message(" completed!
");
    return 0;
}
static void * _taskFunc4(void * __tdata2)
{
    struct __taskenv__0 {
        int (*m);
        int j;
        int (*arry)[10];
        int &ref;
    } __taskenv__0 ( int &ref ) : ref(ref) { }
};

    struct __taskenv__0 * _tenv1 = ( struct __taskenv__0 *) __tdata2;
    int (*m) = _tenv1->m;
    int j = _tenv1->j;
    int (*arry)[10] = _tenv1->arry;
    int &ref = _tenv1->ref;
    {
        ( *arry);
        ref++;
        int local_j = j;
        ( *m);
    }
}

void outlining_example(void)
{
    int m,j;
    int &ref=m;
    int arry[10];

        struct __taskenv__0 {
            int (*m);
            int j;
            int (*arry)[10];
            int &ref;
        } __taskenv__0 ( int &ref ) : ref(ref) { }
    } * _tenv1;

    _tenv1 = new __taskenv__0( ref);
    _tenv1->m = &m;
    _tenv1->j = j;
    _tenv1->arry = &arry;
    cilk_spawn _taskFunc4(_tenv1);  
}  
}
Appendix B

Examples Generated by PPCG

We present PPCG code with generated CUDA kernels for eight examples: array reversal (Figure B.1), 1D Jacobi (Figure B.3), 2D Jacobi (Figure B.4), LU decomposition (Figure B.5), matrix transposition (Figure B.7), matrix addition (Figure B.2), matrix vector multiplication (Figure B.6), and matrix matrix multiplication (Figure B.8).

```c
#pragma scop
for (int i = 0; i < N; i++)
  Out[N - 1 - i] = In[i];
#pragma endscop

__global__ void kernel0(int *In, int *Out, int N) {
  int b0 = blockIdx.x;
  int t0 = threadIdx.x;
  __shared__ int shared_Out[32];
  #define floord(n,d) (((n)<0) ? -((-(n)+(d)-1)/(d)) : (n)/(d))
  for (int c0 = 32 * b0; c0 < N; c0 += 1048576) {
    __syncthreads();
    if (N >= t0 + c0 + 1)
      shared_Out[-t0 + 31] = In[t0 + c0];
    __syncthreads();
    if (N + t0 >= c0 + 32)
      Out[N + t0 - c0 - 32] = shared_Out[t0];
  }
}

Figure B.1: PPCG code and generated CUDA kernel for array reversal
```

```c
#pragma scop
for (int v0 = 0; v0 < n; v0++)
  for (int v1 = 0; v1 < n; v1++)
    c[v0][v1] = a[v0][v1] + b[v0][v1];
#pragma endscop

__global__ void kernel0(int *a, int *b, int *c, int n) {
  int b0 = blockIdx.y, b1 = blockIdx.x;
  int t0 = threadIdx.y, t1 = threadIdx.x;
  #define floord(n,d) (((n)<0) ? -((-(n)+(d)-1)/(d)) : (n)/(d))
  #define min(x,y) ((x) < (y) ? (x) : (y))
  for (int c0 = 32 * b0; c0 < n; c0 += 8192)
    if (n >= t0 + c0 + 1)
      for (int c1 = 32 * b1; c1 < n; c1 += 8192)
        for (int c3 = 32 * t0; c3 < n; c3 += 8192)
          if (n >= t0 + c3 + 1)
            c[(t0 + c0) * n + (c1 + c3)] =
              a[(t0 + c0) * n + (c1 + c3)] +
              b[(t0 + c0) * n + (c1 + c3)];
}

Figure B.2: PPCG code and generated CUDA kernel for matrix addition
```
#pragma scop
for (int t = 0; t < T; ++t) {
    for (int i = 1; i < N-1; ++i)
        b[i] = (a[i-1] + a[i] + a[i+1]) / 3;
    for (int i = 1; i < N-1; ++i)
        a[i] = b[i];
}
#pragma endscop

__global__ void kernel0(int *a, int *b, int N, int T, int c0)
{
    int b0 = blockIdx.x;
    int t0 = threadIdx.x;
    #define floord(n,d) (((n)<0) ? -((-(n)+(d)-1)/(d)) : (n)/(d))
    for (int c1 = 32 * b0; c1 < N - 1; c1 += 1048576)
        if (N >= t0 + c1 + 2 && t0 + c1 >= 1)
            b[t0 + c1] = (((a[t0 + c1 - 1] + a[t0 + c1]) +
                a[t0 + c1 + 1]) / 3);
}

__global__ void kernel1(int *a, int *b, int N, int T, int c0)
{
    int b0 = blockIdx.x;
    int t0 = threadIdx.x;
    
    __global__ void kernel0(int *a, int *b, int N, int T, int c0)
    {
        int b0 = blockIdx.y, b1 = blockIdx.x;
        int t0 = threadIdx.y, t1 = threadIdx.x;
        
        #define floord(n,d) (((n)<0) ? -((-(n)+(d)-1)/(d)) : (n)/(d))
        #define min(x,y) ((x) < (y) ? (x) : (y))
        for (int c1 = 32 * b0; c1 < N - 2; c1 += 8192)
            if (N >= t0 + c1 + 3)
                for (int c2 = 32 * b1; c2 < N - 2; c2 += 8192)
                    for (int c4 = t1; c4 <= min(31, N - c2 - 3); c4 += 16)
                        b[(t0 + c1 + 1) * N + (c2 + c4 + 1)] =
                            (((a[(t0 + c1) * N + (c2 + c4 + 1)]) +
                                a[(t0 + c1 + 1) * N + (c2 + c4 + 1)]) +
                            a[(t0 + c1 + 1) * N + (c2 + c4 + 2)]) / 4;
    }
}

Figure B.3: PPCG code and generated CUDA kernel for 1D Jacobi

__global__ void kernel0(int *a, int *b, int N, int T, int c0)
{
    int b0 = blockIdx.y, b1 = blockIdx.x;
    int t0 = threadIdx.y, t1 = threadIdx.x;
    #define floord(n,d) (((n)<0) ? -((-(n)+(d)-1)/(d)) : (n)/(d))
    #define min(x,y) ((x) < (y) ? (x) : (y))
    for (int c1 = 32 * b0; c1 < N - 2; c1 += 8192)
        if (N >= t0 + c1 + 3)
            for (int c2 = 32 * b1; c2 < N - 2; c2 += 8192)
                for (int c4 = t1; c4 <= min(31, N - c2 - 3); c4 += 16)
                    a[(t0 + c1 + 1) * N + (c2 + c4 + 1)] =
                        b[(t0 + c1 + 1) * N + (c2 + c4 + 1)];
}

Figure B.4: PPCG code and generated CUDA kernel for 2D Jacobi
__global__ void kernel0(double *L, double *U, int n, int c0)
{
    int b0 = blockIdx.x;
    int t0 = threadIdx.x;
    __shared__ double shared_U_1[1][1];
    #define floord(n,d) (((n)<0) ? -((-(n)+(d)-1)/(d)) : (n)/(d))
    if (t0 == 0)
        shared_U_1[0][0] = U[c0 * n + c0];
    __syncthreads();
    for (int c1 = 32 * b0; c1 < n - c0 - 1; c1 += 1048576)
    if (n >= t0 + c0 + c1 + 2)
        L[c0 * n + (t0 + c0 + c1 + 1)] =
        (U[c0 * n + (t0 + c0 + c1 + 1)] - shared_U_1[0][0]);
}
__global__ void kernel1(double *L, double *U, int n, int c0)
{
    int b0 = blockIdx.y, b1 = blockIdx.x;
    int t0 = threadIdx.y, t1 = threadIdx.x;
    __shared__ double shared_L[1][32];
    __shared__ double shared_U_1[32][1];
    #define min(x,y) ((x) < (y) ? (x) : (y))
    #define max(x,y) ((x) > (y) ? (x) : (y))
    if (n + 30 >= ((32 * b1 + 8191 * c0 + 31) % 8192) + c0)
    for (int c1 = 32 * b0; c1 < n - c0 - 1; c1 += 8192) {
        if (t0 == 0)
            for (int c3 = t1; c3 <= min(31, n - c0 - c1 - 2); c3 += 16)
                shared_L[0][c3] = L[c0 * n + (c0 + c1 + c3 + 1)];
        __syncthreads();
        for (int c2 = 32 * b1 + 8192 * (-32 * b1 + c0 + 8160) / 8192; c2 < n; c2 += 8192) {
            if (t1 == 0 & n >= t0 + c2 + 1)
                shared_U_1[t0][0] = U[(t0 + c2) * n + c0];
            __syncthreads();
            if (n >= t0 + c0 + c1 + 2)
                for (int c4 = max(t1, t1 + 16 * floord(-t1 + c0 - c2 - 1, 16) + 16);
                    c4 <= min(31, n - c2 - 1); c4 += 16)
                    U[(c2 + c4) * n + (t0 + c0 + c1 + 1)] -=
                    (shared_L[0][t0] * shared_U_1[c4][0]);
            __syncthreads();
        }
    }
    __syncthreads();
}

Figure B.5: PPCG code and generated CUDA kernel for LU decomposition
__global__ void kernel0(int *a, int *b, int *c, int n)
{
    int b0 = blockIdx.x;
    int t0 = threadIdx.x;
    __shared__ int shared_a[32][32];
    __shared__ int shared_b[32];
    int private_c[1];

    #define floord(n,d) (((n)<0) ? -((-(n)+(d)-1)/(d)) : (n)/(d))
    #define min(x,y) ((x) < (y) ? (x) : (y))

    for (int c0 = 32 * b0; c0 < n; c0 += 1048576) {
        for (int c1 = 0; c1 < n; c1 += 32) {
            if (n >= t0 + c1 + 1) {
                for (int c2 = 0; c2 <= min(31, n - c0 - 1); c2 += 1)
                    shared_a[c2][t0] = a[(c0 + c2) * n + (t0 + c1)];
                shared_b[t0] = b[t0 + c1];
            }
            __syncthreads();
            if (n >= t0 + c0 + 1 && c1 == 0)
                private_c[0] = 0;
            if (n >= t0 + c0 + 1)
                for (int c3 = 0; c3 <= min(31, n - c1 - 1); c3 += 1)
                    private_c[0] += (shared_a[t0][c3] * shared_b[c3]);
            __syncthreads();
            if (n >= t0 + c0 + 1)
                c[(t0 + c0) * n + (c1 + c3)] = shared_a[c3][t0];
            __syncthreads();
        }
    }
}

Figure B.6: PPCG code and generated CUDA kernel for matrix vector multiplication

__global__ void kernel0(int *a, int *b, int *c, int n)
{
    int b0 = blockIdx.y, b1 = blockIdx.x;
    int t0 = threadIdx.y, t1 = threadIdx.x;
    __shared__ int shared_a[32][32];

    #define floord(n,d) (((n)<0) ? -((-(n)+(d)-1)/(d)) : (n)/(d))
    #define min(x,y) ((x) < (y) ? (x) : (y))

    for (int c0 = 32 * b0; c0 < n; c0 += 8192) {
        for (int c1 = 32 * b1; c1 < n; c1 += 8192) {
            if (n >= t0 + c1 + 1) {
                for (int c3 = t1; c3 <= min(31, n - c0 - 1); c3 += 16)
                    shared_a[t0][c3] = a[(t0 + c1) * n + (c0 + c3)];
            }
            __syncthreads();
            if (n >= t0 + c0 + 1) {
                for (int c3 = t1; c3 <= min(31, n - c1 - 1); c3 += 16)
                    c[(t0 + c0) * n + (c1 + c3)] = shared_a[c3][t0];
            }
            __syncthreads();
        }
    }
}

Figure B.7: PPCG code and generated CUDA kernel for matrix transpose
#pragma scop
for (int i = 0; i < n; i++)
for (int j = 0; j < n; j++)
for (int k = 0; k < n; ++k)
c[i][j] += a[i][k] * b[k][j];
#pragma endscop

#define floord(n,d) (((n)<0) ? -((-(n)+(d)-1)/(d)) : (n)/(d))
#define min(x,y) ((x) < (y) ? (x) : (y))
for (int c0 = 32 * b0; c0 < n; c0 += 8192)
for (int c1 = 32 * b1; c1 < n; c1 += 8192) {
if (n >= t0 + c0 + 1 && n >= t1 + c1 + 1) {
private_c[0][0] = c[(t0 + c0) * n + (t1 + c1)];
if (n >= t1 + c1 + 17)
private_c[0][1] = c[(t0 + c0) * n + (t1 + c1 + 16)];
}
for (int c2 = 0; c2 < n; c2 += 32) {
if (n >= t0 + c2 + 1)
for (int c4 = t1; c4 <= min(31, n - c2 - 1); c4 += 16)
shared_a[t0][c4] = a[(t0 + c0) * n + (c2 + c4)];
if (n >= t0 + c2 + 1)
for (int c4 = t1; c4 <= min(31, n - c1 - 1); c4 += 16)
shared_b[t0][c4] = b[(t0 + c2) * n + (c1 + c4)];
__syncthreads();
if (n >= t0 + c0 + 1 && n >= t1 + c1 + 1)
for (int c3 = t0; c3 <= min(31, n - c2 - 1); c3 += 1) {
private_c[0][0] +=
(shared_a[t0][c3] * shared_b[c3][t1]);
if (n >= t0 + c1 + 17)
private_c[0][1] +=
(shared_a[t0][c3] * shared_b[c3][t1 + 16]);
}
__syncthreads();
}
if (n >= t0 + c0 + 1 && n >= t1 + c1 + 1) {
c[(t0 + c0) * n + (t1 + c1)] = private_c[0][0];
if (n >= t1 + c1 + 17)
c[(t0 + c0) * n + (t1 + c1 + 16)] = private_c[0][1];
}
__syncthreads();
}

Figure B.8: PPCG code and generated CUDA kernel for matrix matrix multiplication
Appendix C

The Implementation for Generating Comprehensive MetaFork Programs

In this appendix, we exhibit the pseudocode of the preliminary implementation of the comprehensive optimization algorithm demonstrated in Chapter 7. Algorithm 12 is the implemented algorithm for generating comprehensive MetaFork programs from a given MetaFork program, while Algorithm 13 and Algorithm 14 comprise the implemented Optimize procedure. In this implementation, we consider two resource counters: register usage per thread and data amount per thread block to be cached in the shared memory; meanwhile, we apply three optimization strategies, including reducing register pressure, controlling thread granularity, and common sub-expression elimination.

Algorithm 12: MultiParametricCodeOptimizer(fileName)

Input: fileName, giving the location of the input program
Output: optimized versions of the input program in the form of a case discussion (depending on the hardware resource limits)

1. plans := [Create_Optimization_Plan(fileName)];
2. results := [];
3. while the number of plans <> 0 do
   4. plan := plans[1]; plans := plans[2..-1];
   5. task := ExtractTask(plan) [1];
   6. new.plans := Optimize(plan, task);
   7. for new.plan in new.plans do
      8. if IsCompleted(new.plan) then
         9. results := [new.plan, op(results)];
      else
         10. plans := [new.plan, op(plans)];
   11. return results;
Algorithm 13: Optimize\((plan, task)\)

**Input:** \(plan\), encoding a program being optimized, and \(task\), an optimization task of \(plan\)

**Output:** A list of new plans obtained by optimizing \(plan\) according to \(task\)

1. local caching\(_{task}\), granularity\(_{task}\), register\(_{task}\), new\(_{plans}\), optimized\(_{plans}\), current\(_{vars}\), alternative;
2. new\(_{plans} := []\);
3. if \(task[NAME] = \text{"Register\_Pressure\_Control" and } task[\text{CURRENTLEVEL}] <=\) task[\text{FINALLEVEL}] then
4. alternative := Copy\_Optimization\_Plan\((plan)\);
5. \(r := \text{RegisterPressure}(plan, task[\text{CURRENTLEVEL}])\);
6. /* Accept case */
7. plan[\text{CONSTRAINTS}] := \([\text{"<"} (r, R.B), \text{op(plan[CONSTRAINTS])}]\);
8. if IsConsistent(plan) then
9. register\(_{task} := \text{FindTask}(plan, \text{"Register\_Pressure\_Control"})\);
10. register\(_{task}[\text{CURRENTLEVEL}] := \text{register\(_{task}[\text{FINALLEVEL}] + 1\)};
11. plan[\text{LOG}] := \[\text{"Accept register pressure"}, \text{op(plan[LOG])}\];
12. caching\(_{task} := \text{FindTask}(plan, \text{"Caching"})\);
13. if caching\(_{task}[\text{CURRENTLEVEL}] >\) caching\(_{task}[\text{FINALLEVEL}] then
14. granularity\(_{task} := \text{FindTask}(plan, \text{"Granularity\_Control"})\);
15. granularity\(_{task}[\text{CURRENTLEVEL}] := \text{granularity\(_{task}[\text{FINALLEVEL}] + 1\)};
16. plan[\text{LOG}] := \[\text{"No granularity reduction"}, \text{op(plan[LOG])}\];
17. new\(_{plans} := [plan, \text{op(new\(_{plans})}]\);
18. /* Refuse case */
19. alternative[\text{CONSTRAINTS}] := \[\text{"<"}(R.B, r), \text{op(alternative[CONSTRAINTS])}]\);
20. if IsConsistent(alternative) then
21. register\(_{task} := \text{FindTask}(alternative, \text{"Register\_Pressure\_Control"})\);
22. if (register\(_{task}[\text{CURRENTLEVEL}] <\) register\(_{task}[\text{FINALLEVEL}]\) then
23. new\(_{plans} := [alternative, \text{op(new\(_{plans})}]\);
24. else
25. optimized\(_{plans} := \text{Optimize}(alternative, \text{FindTask}(alternative, \text{"CSE"}))\);
26. if evalb(nops(optimized\(_{plans}) <> 0) then
27. new\(_{plans} := [\text{op(optimized\(_{plans})}, \text{op(new\(_{plans})}]\);
28. else
29. optimized\(_{plans} := \text{Optimize}(alternative, \text{FindTask}(alternative, \text{"Granularity\_Control"}))\);
30. if evalb(nops(optimized\(_{plans}) <> 0) then
31. new\(_{plans} := [\text{op(optimized\(_{plans})}, \text{op(new\(_{plans})}]\);
32. /* No "else" case since we tried everything we could */
33. /* to reduce register pressure and we failed! */

/* To be continued in Algorithm \[14\] */
Algorithm 14: Optimize(plan, task)

Input: plan, encoding a program being optimized, and task, an optimization task of plan

Output: A list of new plans obtained by optimizing plan according to task

/* continuing Algorithm 13 */
1 else if task[NAME] = “Caching” and task[CURRENTLEVEL] <= task[FINALLEVEL] then
2 alternative := Copy_Optimization_Plan(plan);
3 z := CacheAmount(plan);
4 /* Accept case */
5 plan[CONSTRAINTS] := [‘<=’(z, Z_B), op(plan[CONSTRAINTS])];
6 current_vars := { op((plan[RING])[variables]) };
7 current_vars := (current_vars union indets(z)) minus R_B, Z_B;
8 plan[RING] := RegularChains:-PolynomialRing([R_B, Z_B, op(current_vars)));
9 if IsConsistent(plan) then
10 plan[LOG] := [“Accept caching”, op(plan[LOG])];
11 task[CURRENTLEVEL] := task[FINALLEVEL] + 1;
12 register_task := FindTask(plan, “Register_Pressure_Control”);
13 if register_task[CURRENTLEVEL] > register_task[FINALLEVEL] then
14 granularity_task := FindTask(plan, “Granularity_Control”);
15 granularity_task[CURRENTLEVEL] := granularity_task[FINALLEVEL] + 1;
16 plan[LOG] := [“No granularity reduction”, op(plan[LOG])];
17 new_plans := [plan, op(new_plans)];
8 /* Refuse case */
17 alternative[CONSTRAINTS] := [‘<’(Z_B, z), op(alternative[CONSTRAINTS])];
18 alternative[RING] := plan[RING];
19 if IsConsistent(alternative) then
20 optimized_plans := Optimize(alternative,FindTask(alternative, “Granularity_Control”));
21 if evalb(nops(optimized_plans) <> 0) then
22 new_plans := [op(optimized_plans), op(new_plans)];
23 else
24 optimized_plans := Optimize(alternative,FindTask(alternative, “CSE”));
25 if evalb(nops(optimized_plans) <> 0) then
26 new_plans := [op(optimized_plans), op(new_plans)];
27 else
28 task := FindTask(alternative, “Caching”);
29 task[CURRENTLEVEL] := task[FINALLEVEL] + 1;
30 new_plan := AbandonCaching(alternative);
31 new_plan[LOG] := [“Refuse caching”, op(new_plan[LOG])];
32 new_plans := [new_plan, op(new_plans)];
33 else if task[NAME] = “CSE” and task[CURRENTLEVEL] <= task[FINALLEVEL] then
34 task[CURRENTLEVEL] := task[CURRENTLEVEL] + 1;
35 new_plan := ApplyCSE(plan, task[CURRENTLEVEL]);
36 new_plan[LOG] := [“CSE applied”, op(new_plan[LOG])];
37 new_plans := [new_plan, op(new_plans)];
38 else if task[NAME] = “Granularity_Control” and task[CURRENTLEVEL] <= task[FINALLEVEL] then
39 task[CURRENTLEVEL] := task[FINALLEVEL] + 1;
40 new_plan := SetGranularityToOne(plan);
41 new_plan[LOG] := [“Granularity set to 1”, op(new_plan[LOG])];
42 new_plans := [new_plan, op(new_plans)];
43 return (new_plans);
Curriculum Vitae

Name: Xiaohui Chen

Education

Degrees:
- Doctor of Philosophy in Computer Science
  University of Western Ontario, 2012.09 - 2016.09
- Master in Computer Science and Technology
  University of Science and Technology of China, 2009.09-2012.05
- Bachelor in Computer Science and Technology
  Northwestern Polytechnical University, 2005.09 - 2009.07

Related Work

Experience:
- Internship in compiler group, IBM Toronto Lab
  2014.09-2015.01, 2015.09-2016.01, 2016.07-2016.11

Publications:
