Automatic Parallelization for Graphics Processing Units in JikesRVM

by

Alan Chun-Wai Leung

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2008

Abstract

Accelerated graphics cards, including specialized high-performance processors called Graphics Processing Units (GPUs), have become ubiquitous in recent years. On the right kinds of problems, GPUs greatly surpass CPUs in terms of raw performance. However, GPUs are currently used only for a narrow class of special-purpose applications; the raw processing power available in a typical desktop PC is unused most of the time.

The goal of this work is to present an extension to JikesRVM that automatically executes suitable code on the GPU instead of the CPU. Both static and dynamic features are used to decide whether it is feasible and beneficial to off-load a piece of code to the GPU. Feasible code is discovered by an implementation of data dependence analysis. A cost model that balances the speedup available from the GPU against the cost of transferring input and output data between main memory and GPU memory has been deployed to determine if a feasible parallelization is indeed beneficial. The cost model is parameterized so that it can be applied to different hardware combinations.

We also present ways to overcome several obstacles to parallelization inherent in the design of the Java bytecode language: unstructured control flow, the lack of multi-dimensional arrays, the precise exception semantics, and the proliferation of indirect references.

Acknowledgements

First and foremost I would like to express my gratitude to my supervisor, Ondřej Lhoták, for all his patience, support and guidance. Without him, this work would not be possible.

I would like to thank Rose, Telson, Helen, Joanna, my family and friends who provided love and care before and during my academic studies.

I would also like to thank Ghulam Lashari for his help in the subject of GPU programming and developers of JikesRVM for their research platform as well as friendly assistance on the mailing list.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

The graphics card in a typical modern desktop personal computer has significantly more raw processing power and memory bandwidth than the general purpose Central Processing Unit (CPU). In addition to being used for displaying graphics with a monitor, newer generations of these graphics cards are capable of offloading many general-purpose computations from the CPU. For that reason, modern graphics cards are usually referred to as Graphics Processing Units (GPUs).

GPUs can also have significantly high raw performance than CPUs. For example, the NVIDIA GeForce 7800 GTX can perform 165 Giga Floating Point Operations Per Second (GFLOPS), while the theoretical peak rate of a dual-core 3.7 GHz Intel Pentium 965 is 25.6 GFLOPS [38]. Figure 1.1 shows a performance growth comparison between GPUs and CPUs in recent years. The performance gap between GPUs and CPUs has been widening and is likely to continue to increase, even as the number of CPU cores increases. Adding CPU cores requires duplicating control logic and implementing expensive cache-coherency protocols. In contrast, increasing the processing power of a GPU-like tiled SIMD architecture requires significantly fewer hardware resources.

However, the processing power provided by the GPU is unused most of the time, except in very specific applications. In recent years, a "general-purpose"

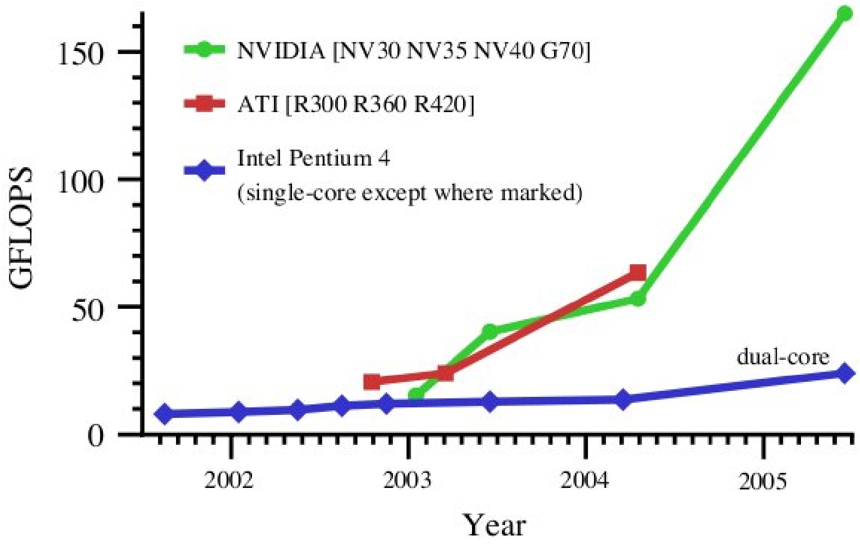


Figure 1.1: GPU performance [38]

GPU (GPGPU) community has sprung up, which applies GPUs to problems other than rendering three-dimensional scenes [38]. Despite the term "general-purpose", the GPGPU community focuses on adapting specific algorithms for execution on GPUs, although there has been some work on programming systems targetting a range of applications.

Although GPU vendors are progressively increasing the ease of use of their GPUs in general purpose computation, programming for GPUs remains a difficult task. Many application programmers for domain specific applications have not been trained to program parallel architectures.

1.2 Goal

The goal of this thesis is to describe our prototype implementation of an auto-parallelizing compiler that takes advantage of GPU resources even for code that has not been explicitly implemented with GPUs in mind. This is accomplished by extending a Java JIT compiler to detect loops which can be parallelized, and which can be executed more quickly on the GPU than on the CPU. The higher raw performance of the GPU must be weighed against the cost of transferring the input and output between main memory and GPU memory. We propose a parameterized cost model to weigh these costs and decide when it is beneficial to execute code on the GPU. The parameters are used to tune the cost model to the specific hardware on which the code runs.

1.3 Contributions

Our work makes the following contributions:

- It proposes a new loop parallelization algorithm tailored to the programming model exposed by common GPU hardware. The GPU programming model combines some characteristics of both the vector and multi-processor execution models targetted by traditional parallelization algorithms, but is distinct from both of these models.

- It describes the prototype implementation that was built on top of an existing research Java virtual machine.

- It identifies obstacles to parallelization that are specific to Java bytecode, and briefly discusses the solutions that we have implemented to overcome them. The use of Just-in-Time compilation makes it possible to overcome these difficulties with simple but effective techniques.

- It introduces a technique for minimising the number of data transfers between the system's memory and the GPU's memory.

- It proposes and evaluates a cost model for deciding whether it is profitable to run a given loop on the GPU rather than the CPU. In particular, the cost model balances the data transfer overhead against the faster computation possible on the GPU.

The rest of the thesis is organized as follows: Chapter 2 provides background on GPUs, JikesRVM, dependence analysis and related work. Chapter 3 describes the core implementation of the dependence analysis, the GPU parallelization algorithm, other challenges faced and improvements made. Chapter 4 reports on an experimental evaluation of the cost model. Lastly, Chapter 5 concludes.

Chapter 2

Background

2.1 GPU Overview

This section of the thesis will first explain the programming model presented by GPUs to programmers, and how it is intended to be used for rendering 3D images. Later, we will explain how the programming model can be used for other general purpose applications.

Under graphics application programming interfaces (APIs), the GPU programming model is organized as a pipeline of several stages, as shown in Figure 2.1. The figure shows a conceptual view of the pipeline exposed by the APIs. Actual hardware implementation can vary. The first stage of the pipeline is the geometry stage, and is implemented by the vertex processor. The input to this stage is a list of vertices in a three-dimensional local coordinate space describing the scene to be rendered. For each vertex, this stage translates it to global coordinates, calculates lighting information, and maps it to its two-dimensional position on the screen.

The second stage is the rasterizer. The rasterizer produces a bitmap with the same number of elements (called *fragments*) as the number of pixels in the image being rendered. On this bitmap, the rasterizer draws the polygons described by the two-dimensional vertices that were computed in the geometry stage. Each fragment contains a fixed amount of information such as color, texture coordinates and depth. Parameters generated by the vertex processor are also interpolated to

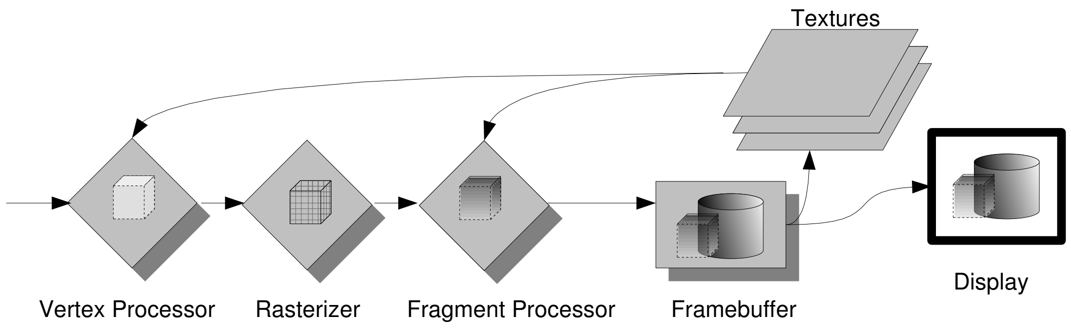


Figure 2.1: GPU Pipeline

each fragment.

The third stage is fragment processing. From the information stored in each fragment, the fragment processor computes a colour for the corresponding pixel. In addition to reading the information for the current fragment, the fragment processor may also perform random-access reads from *textures*, which are additional input arrays distinct from the fragment bitmap. The colours computed by the fragment processor may be written either to the frame buffer to be shown on the screen, or to a texture, from which they can be read again in subsequent fragment stage passes. However, it is not possible to read and write to the same texture in one pass.

In older GPUs, the transformations performed by these stages were fixed by the hardware, but in recent years, the vertex processing and fragment processing stages have become fully programmable and have been unified into a single hardware component. Non-graphical applications usually use only the fragment processor because of the convenient feedback loop provided by its ability to write to and read from textures. However, the vertex processor does expose a scatter operation.

The computation to be performed is specified to the GPU in the form of a fragment program, which traditionally performs floating point operations on a set of registers local to each fragment. Although early GPUs executed only straight-line code, current models support control flow within the fragment program, including loops. The fragment program may also perform an arbitrary number of random access reads from textures, but it may only output a fixed, small number of floating point numbers as its output, and to fixed output locations. The GPU executes the fragment program many times (i.e.,once for each output fragment), and records the output values generated by each execution in an element of the output texture. Because of the limitation on the output of a fragment program, a common GPGPU technique is to divide an algorithm into a sequence of passes, with each pass reading the texture generated by the preceding pass.

Listing 1 Matrix-vector Example

```
public static float[]
  fix(float[][] A, float[] B) {

  float[] Bn = new float[100];

  for (int k = 0; k < 10; k++) { // Loop 1
    for (int x = 0; x < 100; x++) { // Loop 2
      float s = 0;
      for (int y = 0; y < 100; y++) { // Loop 3
        s += A[x][y] * B[y];
      }
      Bn[x] = s;
    }
    float[] tmp = Bn;
    Bn = B;
    B = tmp;
  }
  return B;
}
```

We use an example to demonstrate how computations are mapped to the GPU. Figure 1 shows Java code for an example program that multiplies a matrix by a vector 10 times. There are three ways in which a loop can be implemented when using a GPU, and we will use the three nested loops in the example program to demonstrate them. Figure 2 shows a GPU fragment shader program, written in Cg [26], used to implement the computation. The outermost loop ($Loop_1$) is still executed on the CPU, and triggers the GPU program 10 times (the CPU code is not shown). The body of the middle loop ($Loop_2$) becomes the fragment program. The GPU will execute the fragment program once for each element of the output array, thereby implementing the middle loop. Neither the CPU program nor the

Listing 2 matrix-vector code for GPU

```
float iteration (
in float2 coords : TEXCOORD0,
uniform samplerRECT textureA,
uniform samplerRECT textureB,
) : COLOR {
  float s = 0.0;
  for (float y = 0; y < 100; y++) {
    float x = coords.x;
    s += texRECT(
      textureA, float2(x,y)) *
      texRECT(textureB, float2(0,y))
    );
  }
  return s;
}
```

GPU fragment program implements the middle loop; its implementation is implicit in the data-parallel programming model exposed by the GPU. The innermost loop (*Loop*$_3$) is encoded in the fragment program, since it is inside the body of the middle loop.[1] We call these three implementations of loops CPU, GPU-Implicit, and GPU-Explicit, respectively. In Section 3.3.4, we will give an algorithm to deciding how each loop in a program should be implemented.

Unfortunately, most graphics cards can only be used with a graphics API such as OpenGL [48] and shaders such as the Cg program shown in Listing 2. We must set up the graphics card so that when the fragment program is run, it will compute the desired result. First, any array data must be stored as textures. In the example shown in Listing 1, A and B should occupy two textures. Initial values of the two arrays should be copied into the texture before execution. The programmer should then instruct the graphics API to draw a `quad` primitive that is the size of B somewhere in the scene. The view should be set so that the `quad` is the only object drawn to the viewport. Finally, the programmer must also instruct the API to apply the shader given in Listing 2 after binding the textures accordingly. The end result will be that the value of the frame buffer will contain the results of one

[1]This example could be implemented more efficiently if a loop interchange transformation was first applied; however, the purpose of the example is to demonstrate the three kinds of loops.

iteration of the matrix vector multiplication. The same procedure can be repeated after storing the framebuffer as the new texture B.

This technique is the more traditional GPGPU programming model that is widely applicable to many GPUs. Some newer GPGPU specific programming languages are described in Section 2.4. However, regardless of the programming method used, the GPU can be viewed as a stream processor as shown here:

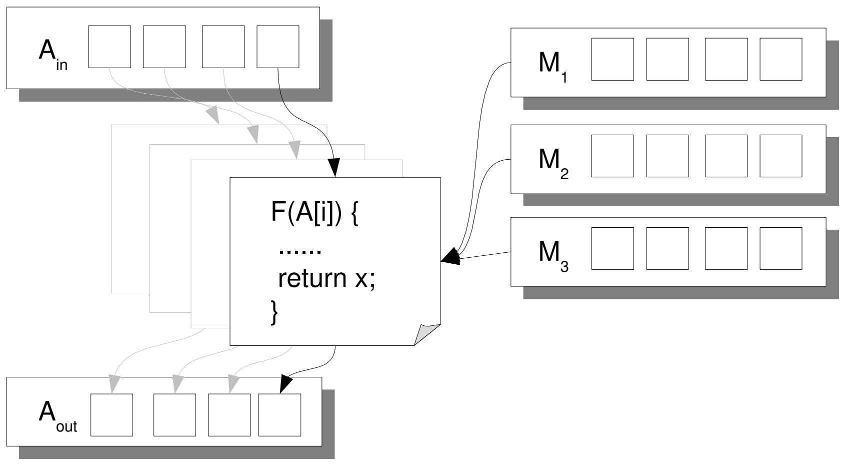


Figure 2.2: Stream Programming Model

We will call this programming model the Single Program Multiple Data (SPMD) stream programming model. A programmer can view the GPU as a machine that applies a small identical program (kernel) in parallel to every element of an array and stores the result to an output array. In figure 2.2, the program F is applied to every single element of A_{in} and output is stored in corresponding entries of A_{out}. Also, F is allowed to access other data randomly.

There are many other differences between GPUs and CPUs. Most GPUs are tailored to render graphics in real time, so certain tradeoffs are made in their design that hinder other applications.

In the standard GPU pipeline, fragment programs can perform data gathering operations (i.e., reading from arbitrary data-dependent locations in texture memory), but they cannot perform data scattering operations (i.e., writing to arbitrary data-dependent locations in texture memory). Several workarounds are possible. NVIDIA's new CUDA programming model allows arbitrary writes [2]. The latest generation of GPUs supporting the Direct3D 10 programming model have a programmable rasterizer which could be used to perform scatter operations in between passes of a fragment program [21]. However, hardware and drivers supporting these new programming models are not yet widespread. In this work, we target the GPUs that are widely available today using the ubiquitous OpenGL programming model. For that reason, our stream programming model shown in Figure 2.2 is able to read any textures available but can only to write to the output element associated with the current kernel instance.

For reasons of hardware cost, some current GPUs also do not fully respect IEEE standards when performing floating point arithmetic. Our implementation does not transform code marked with the Java `strictfp` modifier, which is intended to mark expressions which must be evaluated in strict compliance with the IEEE 754 standard. However, for code not marked with `strictfp`, it is possible that the result computed by a given GPU does not comply even with the looser Java standard for floating point arithmetic.

More information about the graphics pipeline and 3D graphics programming can be found in the OpenGL Programming Guide [48].

2.2 JikesRVM Overview

Our implementation is built on top of an existing research Java virtual machine called JikesRVM (Jikes Research Virtual Machine). JikesRVM (formally known as Jalapeño) provides a flexible open testbed to prototype virtual machine technologies and experiment with a large variety of design alternatives [6]. This subsection of the thesis will provide some background information on JikesRVM itself.

The most distinctive design feature of JikesRVM compared to other Java virtual machine is the language JikesRVM is written in. Much of the core functionality is written in Java while roughly 3% is written in C++ [11]. All applications written in Java require a VM (virtual machine) to execute. However, instead of requiring another VM to run JikesRVM, JikesRVM is actually executing on itself. There are a few special implementation choices made in order to facilitate this goal.

First, unlike most other VMs, JikesRVM never interprets the Java bytecode. Instead, it relies only on the Just-in-Time (JIT) compilation. JikesRVM has two different compilers: baseline and optimizing. The baseline compiler offers a quick bytecode to machine code compiler for most initial compilation of Java methods. The optimizing compiler, on the other hand, offers a multi-level optimizing compiler that requires more compilation time but which is suitable for recompilation of frequently executed methods. Currently JikesRVM supports the x86 and the PowerPC CPU architectures.

The build process also differs from normal Java programs. First, a static compiler is invoked to build each Java source file and generate all the required class files.

One of the JIT compilers within JikesRVM is chosen to run on an external VM to bootstrap the VM itself. Sun Microsystems' HotSpot, for example, has been used as the external VM. The bootstrapping process involves compiling a minimal subset of class files needed for elementary functions of the RVM to the machine language of the target machine. The list of class files required (called the *primordial list*) will be saved as a single image on disk. When JikesRVM runs, this image is loaded from disk, and is used to compile and run the other class files of JikesRVM and the application begin executed. More information about the bootstrapping and build system of JikesRVM can be found in *Implementing Jalapeño in Java* [10].

The self executing VM design enables many interesting optimizations. Not only does JikesRVM perform optimization of the running application, it can optimize itself while it is running if necessary. Recompilation decisions are made using the adaptive optimization system. JikesRVM collects information about the executing application by means of instrumenting and profiling branches. Based on statistics gathered by instrumenting the generated code, the VM will decide on a recompilation plan as well as use the information as an aid to optimizations. More information about the adaptive optimization system in JikesRVM can be found in *Adaptive optimization in the Jalapeño JVM* [13].

The optimizing compiler works on three intermediate representations (IR) similar to those of *Advanced Compiler Design and Implementation* [35]: High-level

Intermediate Representation (HIR), Low-level Intermediate Representation (LIR) and Machine-level Intermediate Representation (MIR). Much of our extension resides in optimization passes that work on the HIR representation. In particular, the Static Single Assignment (SSA) variation of HIR is where the core parallelization phrases are implemented.

Although not directly used in our work, there are many more interesting design decisions and implementations within JikesRVM. For example, JikesRVM deploys a virtual processor approach for threads where threads are multiplexed onto one or more virtual processors. The VM then schedules virtual processors to physical ones. A ”quasi-preemption” scheduling scheme is used to allow context switching only in specific instructions of the running program called *yield points*. This allows the implementation of parallel garbage collectors. For more information related to internals of JikesRVM and other research projects that use JikesRVM, a comprehensive list of all publications that involve JikesRVM can be found in the JikesRVM publications website [7].

2.3 Dependence Overview

This section provides an overview of data dependence analysis which is essential when determining which sections of the source code can be executed in parallel in the GPU. An overview of dependence analysis as well as numerous definitions that are key to understanding our implementation are discussed in this section. More comprehensive treatments of dependence analysis can also be found in many textbooks [51, 47, 8].

Like all optimizations, parallelization must preserve the semantics of the original program. Consider the following example:

Listing 3 Simple Sequential Example

```
int x = 0;
x = 1; // S1
System.out.println(x); // S2
```

Suppose we would like to execute both S_1 and S_2 in parallel, possibly in two different processors. Depending on which processor finishes executing its corresponding instruction first, the output can vary. We might say that S_2 depends on the execution of S_1 to be completely finished. The relative ordering of S_1 and S_2, therefore, must be preserved.

The study of the dependence relationship between instructions is called **dependence analysis**. The primary focus of dependence analysis is to identify possible control or data interference relationships between pairs of instructions such as those displayed in Listing 3. Dependence exists between specific executions of statements. Before we can define formally what data dependence is, we must be able to classify specific executions of a statement as a single statement might be executed multiple times. To do that we will use **iteration vectors**. Consider the following program:

Listing 4 Dependence across iterations

```
for(int i = 0; i < 100; i++) {
    for(int j = 0; j < 100; j++) {
    ... // S0
    }
}
```

The execution of S_0 when the induction variables have the values $i = 5$ and $j = 6$ is said to have an iteration vector of $\vec{v} = (5, 6)$. Formally we define the iteration vector as follows:

Definition 1 (Iteration Vector). *Suppose L is a set of n nested loops $L_0, L_1, L_2...L_n$ with loop induction variables $i_0, i_1, i_2...i_n$ respectively. The execution of a statement S in L when $i_0 = v_0, i_1 = v_1, i_2 = v_2, ...i_n = v_n$ is represented by $S[\vec{v}]$ where $\vec{v}$ is the* **iteration vector** *$(v_0, v_1, v_2, ...v_n) \in \mathbb{Z}^n$.*

Definition 2 (Iteration Space). *The set of all possible iteration vectors of a loop nest is called the* **iteration space** *of the loop nest.*

Definition 3 (Iteration Vector Ordering). *Given vectors $\vec{v_0} = (v_0, v_1, v_2...v_{n-1})$ and*

$\vec{v_1} = (v'_0, v'_1, v'_2 ... v'_{n'-1})$, we define the partial orders $<_c$, $<$ and $=$ as follows:

$$\begin{aligned}
\vec{v} = \vec{v'} &\iff v_0 = v'_0, v_1 = v'_1, ... v_{min(n,n')-1} = v'_{min(n,n')-1} \\
\vec{v} <_c \vec{v'} &\iff v_0 = v'_0, v_1 = v'_1, ... v_{c-1} = v'_{c-1}, v_c < v'_c \\
\vec{v} < \vec{v'} &\iff \exists_{c<n} | \; \vec{v} <_c \vec{v'}
\end{aligned}$$

We say $\vec{v} >_c \vec{v'}$ or $\vec{v} > \vec{v'}$ if $\vec{v'} <_c \vec{v}$ or $\vec{v'} < \vec{v}$ respectively.

The $<_c$ relationship between iteration vectors creates a partial ordering in $\mathbb{Z}^n$. We can now formally define the execution of a statement to be the following:

Definition 4 (Execution of Order Statements)**.** *If the execution of $S_0[\vec{v}]$ occurs before $S_1[\vec{v'}]$, we will write $S_0[\vec{v}] \ll S_1[\vec{v}]$.*

Also, given two vectors $\vec{v}$ and $\vec{v'}$, $\vec{v} < \vec{v'}$ implies that $S_0[\vec{v}]$ executes before $S_1[\vec{v'}]$ regardless of the lexical order of S_0 and S_1 in the source code [51, Theorem 4.1]. Using Definition 4, we can identify the specific execution of any statement that is nested within loops. For statements that are outside of loops, we can define the function body as a single loop that iterates only once. The execution of S_0 when the induction variables have the values $i = 5$ and $j = 6$ in Listing 4 will be written as $S[(0, 5, 6)]$. Since our work focuses on loops, we will abbreviate execution as $S[(5, 6)]$.

With a formal definition of execution of statements, we can formally define dependence.

Definition 5 (Data Dependence)**.** *A* **data dependence** *between the execution of statements $S_0[\vec{v}]$ and $S_1[\vec{v'}]$ occurs when $S_1[\vec{v}]$ accesses the same data as $S_0[\vec{v}]$ and $S_0[\vec{v'}] \ll S_1[\vec{v'}]$. We will write $S_0[\vec{v}] \rightarrow S_1[\vec{v'}]$ if such a dependence exists. If not, we will write $S_0[\vec{v}] \not\rightarrow S_1[\vec{v'}]$. Furthermore, we will write $S_0 \rightarrow S_1$ if there exist $\vec{v}$ and $\vec{v'}$ in the iteration space such that $S_0[\vec{v}] \rightarrow S_1[\vec{v'}]$. Otherwise, we will write $S_0 \not\rightarrow S_1$ if for all $\vec{v}$ and $\vec{v'}$ in the iteration space, $S_0[\vec{v}] \not\rightarrow S_1[\vec{v'}]$.*

The type of a dependence can be further classified as **true dependence**, **anti-dependence**, **output dependence** or **artificial dependence**. Table 1 shows the classifications of data dependences according to whether the accesses are reads or writes. Figure 2.3 demonstrates an example of each dependence. In the first three examples, at least one access is a write, so rearranging the relative order of S_0 and S_1 will definitely alter the program's semantics. In some memory models, two reads may be considered to interfere. However, in our work, we assume that pairs of reads are not considered a dependence. We shall redefine data dependence as follows:

Definition 6 (Data Dependence (with Interference))**.** *A* **data dependence** *between the execution of statements $S_0[\vec{v}]$ and $S_1[\vec{v'}]$ occurs when $S_1[\vec{v'}]$ accesses the same data as $S_0[\vec{v}]$,* **at least one of the instruction writes to that data***, and $S_0[\vec{v}] < S_1[\vec{v'}]$.*

Table 1 Data dependence types.

S_0	S_1	Name
WRITE x	READ x	True Dependence
READ x	WRITE x	Anti-dependence
WRITE x	WRITE x	Output Dependence
READ x	READ x	Artificial Dependence

When dealing with dependence relationships between instructions across iterations, we can further classify data dependences as **loop-carried** and **loop-independent**.

Definition 7 (Loop-carried Dependence)**.** *A data dependence is* **loop-carried** *if there exist iterations $\vec{v}$ and $\vec{v'}$ such that $S_0[\vec{v}] \rightarrow S_1[\vec{v'}]$ and $\vec{v} < \vec{v'}$.*

Definition 8 (Dependence Carried by a Loop)**.** *Let $S_0[\vec{v}] \rightarrow S_1[\vec{v'}]$ be a loop-carried dependence with $\vec{v} <_c \vec{v'}$. We denote this dependence as $S_0 \rightarrow_c S_1$ and we say that the loop-carried dependence is* **carried by** *the loop at depth c. Alternatively, we can also write the dependence as $S_0 \rightarrow_L S_1$ if loop L is the loop at depth c.*

Definition 9 (Loop Independent Dependence)**.** *A data dependence is* **loop-independent** *if there exists $\vec{v}$ such that $S_0[\vec{v}] \rightarrow S_1[\vec{v}]$ and for all $\vec{w}, \vec{w'}$ $w \neq w' \longrightarrow S_0[\vec{w}] \not\rightarrow S_1[\vec{w'}]$. To distinguish this from loop-carried dependence, we will denote this type of dependence as $S_0 \rightarrow_\infty S_1$.*

```
x = 0;
....
x = 1;                  // S0
System.out.println(x);  // S1
```

(a) True Dependence

```
x = 0;
....
System.out.println(x);  // S0
x = 1;                  // S1
```

(b) Anti Dependence

```
x = 0;
....
x = 2; // S0
x = 1; // S1
```

(c) Output Dependence

```
x = 0;
....
System.out.println(x);  // S0
System.out.println(x);  // S1
```

(d) Artificial Dependence

Figure 2.3: Examples of different dependences

Consider the following code segment:

Listing 5 Dependence across iterations

```
int x = 0;

for(int i = 0; i < 100; i++) {
    System.out.println(x);     // S0
    x = i;                     // S1
}
```

There is a loop-independent anti-dependence $S_0 \rightarrow_\infty S_1$ as well as true loop-carried dependences on $S_1[0] \rightarrow S_0[1]$, $S_1[1] \rightarrow S_0[2]$, $S_1[2] \rightarrow S_0[3] ... S_1[98] \rightarrow S_0[99]$ which can simply be denoted by $S_1 \rightarrow_1 S_0$.

Given a loop body, it is known that individual iterations of the body can be executed in parallel if each iteration does not depend on the outputs of other iterations [8, Theorem 2.8].

Theorem 1 (Dependence and Parallel Execution of Loop Iterations)**.** *Given a set of instructions $S = \{S_0, S_1, ...S_{n-1}\}$ within a loop of depth c, iterations of the loop can be executed in parallel if $\forall_{i,j \in [0:n-1]}\ S_i \not\rightarrow_c S_j$.*

Definition 10. *A dependence graph of a program is graph $G = (V, E)$ where $v \in V$ are the nodes representing instructions of the program and the labeled edges $e_\omega = (v_0, v_1) \in E$ implies $v_0 \rightarrow_\omega v_1$ may be true. ω is the loop that carries the dependence and ω is ∞ if the dependence is not loop carried.*

Theorem 1 show that a loop is parallelizable if there are no loop-carried dependences between its iterations. For that reason, our implementation must provide an analysis that will compute all possible dependence across loops. The result will be stored in form of a graph called the **dependence graph** that is defined in Definition 10. Edges of the dependence graph shows possible dependence relationship between instructions. Because these edges represent a "may be" dependent relationship, it is important to create a dependence graph that is as precious as possible.

2.4 Related Work

Most existing parallelization approaches fall into two categories, depending on the hardware features that they exploit: task-level parallelism [39, 20, 42, 15, 12] and vectorization [25, 17, 31, 19, 23, 29, 37, 36, 51, 8, 47].

Task-level parallelism is supported by multiple instances of a fully-functional processor. The overhead of creating threads can be high, but each thread can execute an arbitrary program. Generally, the outermost loop is parallelized, resulting in long tasks and few thread creations. In the context of parallelizing Java, three examples of this technique are JAVAR [18], JavaSpMT [28], and SableSpMT [41].

Vectorization, on the other hand, is supported by a SIMD architecture in which multiple computational units are controlled by a single control unit, so they execute the same instruction. Although SIMD instructions are limited to specific types of computations, they have little overhead, to the point that it is feasible to mix individual SIMD instructions with sequential computations. As a result, vectorization generally targets innermost loops. Vectorization can be categorized into two principal approaches: the traditional loop-based parallelization [19, 37, 46, 32] and the basic block approach [31, 27, 44].

The loop-based vectorization technique proceeds by stripmining the loop. A single loop will be replaced by two nested loops where the number of iterations in the inner loop is same as the vector length. Each scalar instruction in the inner loop body will then be replaced by a corresponding vector instruction. The basic block approach, on the other hand, unrolls the loop by a multiple of the vector length and packs each group of isomorphic scalar instructions into a vector instruction. The loop-based approach requires complicated loop transformations like loop fission (splitting up a single loop to multiple loops) and scalar expansion (replacing scaler variables with an array) and is inhibited by loop carried dependences, especially true data dependences shorter than the vector length. The basic block approach, on the other hand, requires simpler analyses but incurs overhead due to packing and unpacking of the operands of isomorphic statements. Vectorization in general requires very sophisticated analyses and faces numerous challenges including the difficult problem of supporting control flow in vector code [45]. In contrast, our target architecture (the GPU) requires a simple loop analysis and offers a more flexible programming model than the traditional SIMD machines.

Current GPUs cannot be decisively categorized as either multi-processor or vector processors; they share some characteristics of both. The fragment processor has traditionally been a SIMD processor with a limited instruction set. In recent years, hardware for support strictly nested control flow has been added, but it is not intended to support highly divergent control flow. The overhead required to start a computation makes the GPU more similar to a multi-processor system.

The CUDA architecture [2] moves even further towards a general multi-processor style of parallelism. The CUDA programming framework further exposes the GPU to general purpose programmers. No longer do the programmers have to express computation through a graphical API like OpenGL, thus eliminates a lot of unnecessary graphic initialization overhead. The programmers are given a lot of control of the NIVDIA 8800 GPUs in terms of GPGPU programmability. They are allowed to manage thread blocks within the GPU themselves with a C like programming language. Unlike earlier graphics cards, CUDA compatible GPUs allow synchronization like mechanism. However, much like traditional OpenGL based GPGPU techniques, the responsibility to discover parallelism is left to the programmer.

Parallelizing an inner loop would incur high kernel startup overhead, while an outer loop is likely to contain divergent control flow and computations not supported by the GPU. The hybrid nature of GPUs suggests a new kind of parallelization algorithm targeting loops in the middle of a loop nest. In this thesis, we present one such algorithm.

Another parallelization system targeting GPUs is that of Cornwall et al. [24], which performs source-to-source translations to help domain experts retarget an image processing library written in C++ to GPUs. Somewhat similar to the approach described later in this thesis, their translator aims to discover potentially-parallel assignments (PPAs) in loops by walking the abstract syntax tree of the program. Enclosing loops whose induction variables affect the index of array assignments of the PPAs are considered to be potentially-parallel loops (PPLs). Optimizations such as hoisting of parallelization-preventing instructions are then applied to the PPLs and eventually be translated into GPU executable code. This approach works well with programs that are known to have a specific structure. Our approach described here follows the more traditional approach to parallelization in which parallelization attempts occur as high as possible in the loop nest tree. This approach should facilitate extraction of more parallelism.

RapidMind is a C++ GPU metaprogramming framework which consists of two parts [3]. The front-end is a C++ template library that provides data types and overloads operators to generate code in the RapidMind intermediate representation (IR) data structures. The back-end optimizes the IR and emits code for one of the supported target architectures (GPU, Cell BE, multi-core CPU). A programmer can embed a kernel intended to run on the GPU as a suitably delimited piece of C++ code directly in the C++ program. Executing such a kernel requires two steps. In the first step, the C++ code that the programmer has written is executed on the CPU. At this stage, no computation is actually performed. Each overloaded operator, instead of performing a computation, generates the IR instruction that would perform the corresponding computation. Thus, the code that the programmer has written is code that writes the code that will run on the GPU. Once all the code has run and the entire IR has been generated, the RapidMind back-end processes the IR and generates suitable GPU code which can then be executed.

ASTEX [1] takes a run-time approach, in that it searches for hot traces at run time that are amenable to GPU execution [40]. The target program is initially instrumented with monitoring code and executed. Runtime memory access information is gathered and analyzed off-line. Favorable code segments will then be recompiled into Hybrid Multi-core Parallel Programming (HMPP) *codelets*. Similar to RapidMind, HMPP [43] aims to provide a general purpose programming environment for numerous architectures. Unlike RapidMind, however, HMPP codelets rely on compiler directives such as C pragmas.

Recently, interest in using JikesRVM for parallelization has grown. Zhao et al. [50, 49] have also implemented loop parallelization in the context of JikesRVM. However, rather than GPUs, their intended target is JAMAICA [4], a multiprocessor parallel architecture.

Chapter 3

Implementation

3.1 Overview

We have implemented GPU parallelization within an existing Java Just-In-Time (JIT) compiler, JikesRVM [9]. In order to minimize the overhead of parallelization, the compiler must focus on hot (frequently executed) methods of the program. JikesRVM uses an adaptive optimization system [13] with multiple optimization levels; optimizations at higher levels are applied only to methods observed to be hot. GPU parallelization is done at the highest optimization level (*-O2*), and only on code that is expected to be executed frequently.

Figure 3.1 shows the overall architecture of the implementation. The parallelization algorithm is preceded by two preparatory stages. The first stage, *OPT_Array-AccessAnalysis*, recovers information about multi-dimensional array accesses that is lost in Java bytecode (see Section 3.5.1). Such information will be stored within *OPT_ArrayAccessDictionary* in the form of *OPT_ArrayAccess* objects. The second stage, *OPT_GlobalDepAnalysis*, performs dependence analysis on array accesses to construct a dependence graph of type *OPT_GlobalDepGraph* (see Section 2.3). The third stage, *OPT_Parallelization*, implements GPU parallelization, which generates code that will run on the GPU (see Section 3.3).

The primary back-end used by the third stage is the RapidMind platform [3]. RapidMind is a C++ programming framework for expressing data parallel algo-

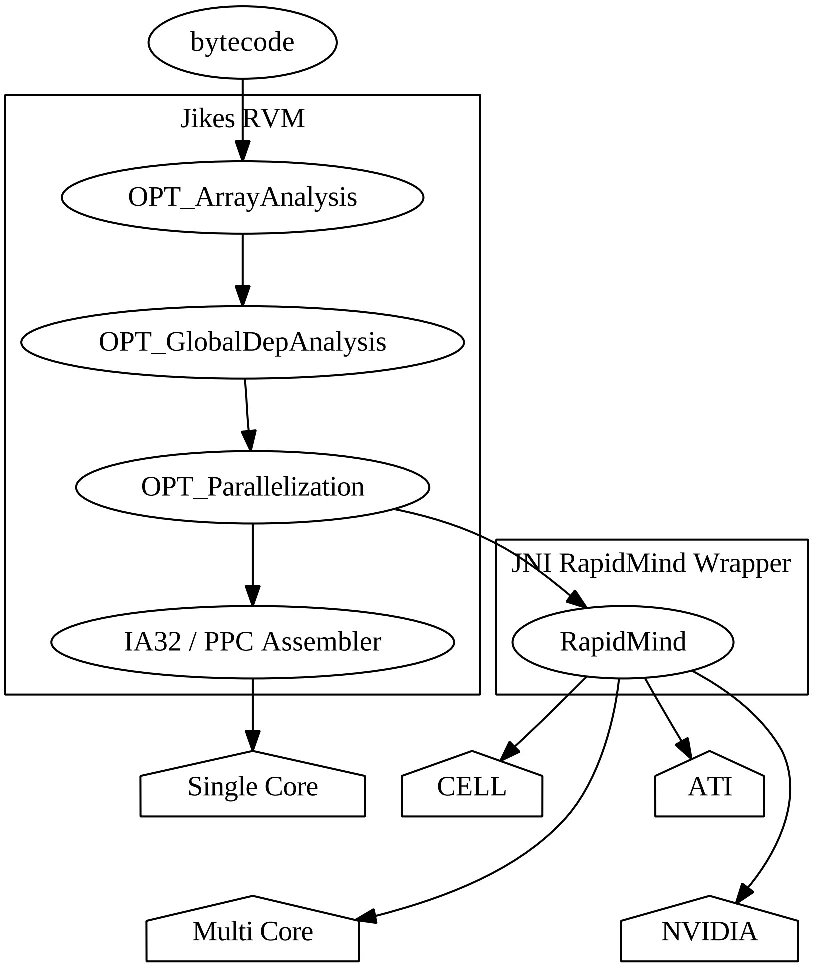


Figure 3.1: Overall Architecture

rithms in a hardware-independent way. Our code generator accesses RapidMind using a Java Native Interface (JNI) wrapper. RapidMind is designed to generate appropriate code at run-time for the chosen hardware. Currently, RapidMind can generate code for GPUs from major vendors, the Cell BE processor, and multi-core CPUs. So far, we have evaluated the system on GPUs only.

This chapter is organized as follows: Section 3.2 will describe the importance of *OPT_GlobalDepGraph* in parallelization and the implementation of *OPT_Global-DepAnalysis*. Section 3.3 will describe the parallelization process and the implementation of *OPT_Parallelization*. Section 3.4 will describe an extension to the parallelization algorithm to eliminate data transfer overhead. Section 3.5 will describe some of the Java specific challenges and how the implementation solves these challenges. Since *OPT_ArrayAccessDictionary* is a Java specific requirement, it will be discussed in section 3.5.

3.2 Dependence Analysis

This section provides an overview of the data dependence analysis implemented by *OPT_GlobalDepAnalysis*. Using the theory described in Section 2.3, this compilation phase will compute a dependence graph that will be used in the parallelization pass.

Before the core *OPT_Parallelization* phase begins, *OPT_GlobalDepAnalysis* is run to create the dependence graph of the method currently being compiled. The entire dependence graph is stored in an object of type *OPT_GlobalDepGraph*; it can be displayed with an added compiler flag (see Appendix A). Nodes of the graph (*OPT_GlobalDepGraphNode*) represent JikesRVM High-level Intermediate Representation (HIR) instructions of the current method. Given an *OPT_Instruction* the corresponding *OPT_GlobalDepGraphNode* can be retrieved by the *scratchObject* of the instruction. The *scratchObject* reference is a common way to annotate instructions to pass information between passes within JikesRVM.

Figure 3.2 shows an example dependence graph printed using the debug flag. Each node in the graph represents a JikesRVM HIR instruction. Green edges represent loop-independent dependences, red edges represent loop-carried dependences while blue edges represent the loop-carried dependences on the loops' induction variables.

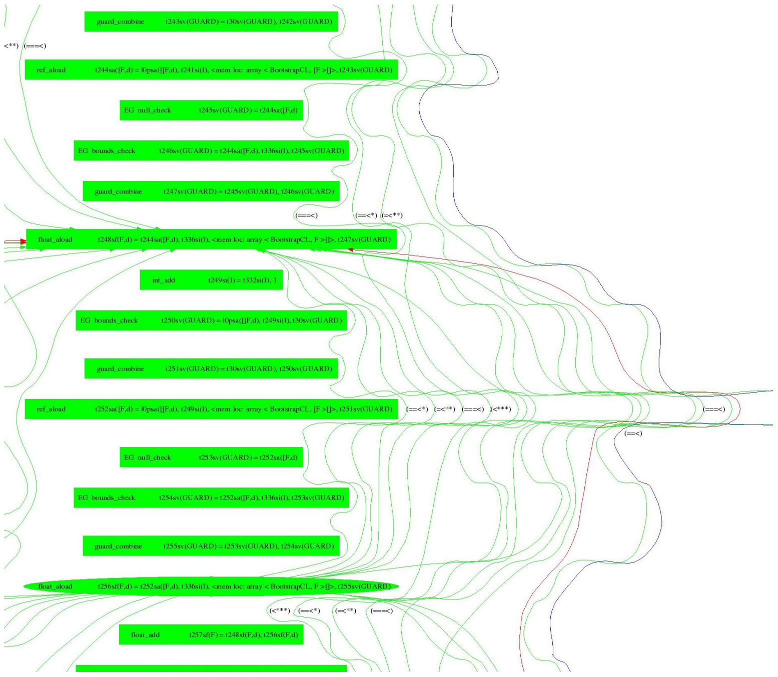

guard_combine t243sv(GUARD) = t30sv(GUARD), t242sv(GUARD)
<**) (===<)
ref_aload t244sa([F,d) = l0psa([[F,d), t241si(I), <mem loc: array < BootstrapCL, [F >[]>, t243sv(GUARD)
EG null_check t245sv(GUARD) = t244sa([F,d)
EG bounds_check t246sv(GUARD) = t244sa([F,d), t336si(I), t245sv(GUARD)
guard_combine t247sv(GUARD) = t245sv(GUARD), t246sv(GUARD)
(===<)
(==<*) (=<**)
float_aload t248sf(F,d) = t244sa([F,d), t336si(I), <mem loc: array < BootstrapCL, F >[]>, t247sv(GUARD)
int_add t249si(I) = t332si(I), 1
EG bounds_check t250sv(GUARD) = l0psa([[F,d), t249si(I), t30sv(GUARD)
guard_combine t251sv(GUARD) = t30sv(GUARD), t250sv(GUARD)
ref_aload t252sa([F,d) = l0psa([[F,d), t249si(I), <mem loc: array < BootstrapCL, [F >[]>, t251sv(GUARD)
(==<*) (=<**) (===<) (<***)
(===<)
(==<)
EG null_check t253sv(GUARD) = t252sa([F,d)
EG bounds_check t254sv(GUARD) = t252sa([F,d), t336si(I), t253sv(GUARD)
guard_combine t255sv(GUARD) = t253sv(GUARD), t254sv(GUARD)
float_aload t256sf(F,d) = t252sa([F,d), t336si(I), <mem loc: array < BootstrapCL, F >[]>, t255sv(GUARD)
(<***) (==<*) (=<**) (===<)
float_add t257sf(F) = t248sf(F,d), t256sf(F,d)

Figure 3.2: Example dependence graph output

The rest of this section will explore how dependence relationships are computed in the *OPT_GlobalDepAnalysis* phrase. One interesting fact about dependence computation is that extra spurious edges between instructions in the dependence graph will not affect the soundness of the parallelization process. However, they do prevent parallelization opportunities. The general approach we use is to start by assuming all dependences exist and remove dependence edges by disproving their existence.

Definition 11 (Spurious Edges of Dependence Graphs)**.** *An edge E_ω between S_0 and S_1 is* **spurious** *if for all plausible execution of the program $S_0 \not\rightarrow_\omega S_1$.*

Given two instructions S_0 and S_1, the first step is to examine dependences related to scalars. The use of SSA within JikesRVM HIR simplifies this process greatly. Two instructions will have a scalar data dependence if both S_0 and S_1 access the same register and one of them defines a register while the other reads it.

The dependence $S_0 \rightarrow S_1$ will be loop carried if S_1 is a SSA form ϕ instruction that reads a variable x in the header of a loop L while S_0 is a instruction that defines x within L. Otherwise, $S_0 \rightarrow S_1$ is loop independent.

A dependence that results from accessing array elements requires a more sophisticated test. Array element accesses are indexed by variables. In many cases the values of the indices are not compile-time constants. For example:

Listing 6 Array indices example

```
x = A[f(c)]; // S0
A[g(c)] = y; // S1
```

If both `f`(c) and `g`(c) are known to be distinct constants , we can prove they are independent. However, when `f`(c) and `g`(c) are not constants, it would seem that the only option is to be conservative and create $k+1$ edges between S_0 and S_1 as $S_0 \rightarrow_m S_1, m \in [1, 2, ..k, \infty]$ where k is the depth the innermost loop that contains S_0 and S_1. To conserve memory, a special *OPT_GlobalDepGraphEdge* flagged *UNKNOWN* is used to represent k edges within the *OPT_GlobalDepGraph*.

UNKNOWN edges prevent parallelization due to the fact that we are assuming the existence of a loop carried dependence. A large number of parallelization opportunities will be missed due to this limitation. Consider the following program:

Listing 7 Dependence across iterations

```
for(int i = 1; i < 100; i++) {
    for(int j = 1; j < 100; j++) {
        A[f(i,j)] = A[(g(i,j)]; //S
    }
}
```

This is a frequently seen array access pattern in which the index is a function of the loop induction variable. Because i and j are induction variables, they will not be compile-time constants and an *UNKNOWN* edge might seem unavoidable. Luckily, there are known algorithms that can be used to prove independence in many cases.

Suppose we are interested in knowing whether S has a true loop-carried dependence on itself. What we want to determine is if there exists some $\vec{v_0} = (x, y)$ and $\vec{v_1} = (x', y')$ such that $A[f(x, y)]$ is addressing the same value as $A[(g(x', y')]$ while $\vec{v_0} < \vec{v_1}$. Formally:

Theorem 2 (Loop-carried Dependence from Array Access)**.** *Let S_0 and S_1 be in a common loop nest with induction variables $i_0, i_1, ...i_{n-1}$. Also let S_0 reference $A[f(i_0, i_1, ...i_{n-1})]$ and S_1 reference $A[g(i_0, i_1, ...i_{n-1})]$, assuming that one of the references be a write and let no other common data be accessed. $S_0 \rightarrow_c S_1$ holds if and only if there exist $\vec{v_0}$ and $\vec{v_1}$ in the iteration space such that $\vec{v_0} <_c \vec{v_1}$ and $f(\vec{v_0}) = g(\vec{v_1})$.*

Proof. This is a straight application of Definition 7 in the special case where the common data is an array element indexed by a function of the induction variables. □

If f and g are pure functions, we could examine the possible values that the functions return for all possible x, x', y, y'. In practice, we need more efficient ways to determine dependence. However, in doing so, we need to make some assumptions that sacrifice completeness. First, the array addressing index value must be **admissible**.

Definition 12 (Admissible Function)**.** *A function is* **admissible** *if the only unknowns are induction variables of the loop nest.*

Second, we attempt to disprove a dependence only when the index expressions are affine functions of the induction variables of the loop nest. These two requirements appear to be restrictive at first glance. In practice, however, many loops are written in this way. The previous dependence can now be generalized as follows:

Given a loop nest with induction variables $i_0, i_1, i_2, ...i_n$ with constant lower bounds $L_0, L_1, L_2, ...L_n$, constant upper bounds $U_0, U_1, U_2, ...U_n$, and loop invariant constants $a_0, a_1, a_2, ...a_n, a_{n+1}, b_0,\ b_1,\ b_2,\ ...\ b_n,\ b_{n+1}$, the instructions S_0 accessing $A[a_0 i_0 + ...a_n i_n + a_{n+1}]$ and S_1 accessing $A[b_0 i_0 + ...b_n i_n + b_{n+1}]$ do not have a loop carried dependence if the following system of inequalities in $2n$ unknowns $(x_0, x_1, x_2, ...x_n, x'_0, x'_1, x'_2, ...x'_n)$ has no solution:

$$a_0x_0 + a_1x_1 + a_2x_2 + ...a_nx_n + a_{n+1} = b_0x'_0 + b_1x'_1 + b_2x'_2 + ...b_nx'_n + b_{n+1}$$

$$(x_0, x_1, x_2, ...x_n) < (x'_0, x'_1, x'_2, ...x'_n)$$

$$L_0 \leq x_0 \leq U_0, L_1 \leq x_1 \leq U_1, L_2 \leq x_2 \leq U_2, ..., L_n \leq x_n \leq U_n$$

To try to prove that the system has no solution, we apply a sequence of successively stronger tests. Within our implementation in JikesRVM, the initial test for the nonexistence of a solution is the **strong separability test** [51]. If the function is not strongly separable, the weak separability test will be used. Although strong separability test is the weakest of all tests due to its strong assertion, it is actually applicable in many practical applications, and is inexpensive to evaluate.

Definition 13. *(Separability) Given two linear functions $f(\vec{x}) = a_0x_0 + ...a_nx_n + a_{n+1}$ and $g(\vec{x'}) = b_0x'_0 + b_1x'_1 + b_2x'_2 + ...b_nx'_n + b_{n+1}$ where $a_0, a_1, a_2, ...a_n, a_{n+1}, b_0, b_1, b_2, ...b_n, b_{n+1}$ are constants, $f(\vec{x})$ and $g(\vec{x}')$ are* **separable** *if there exists $c \in [0 : n]$ such tvec for all $i \in [1 : n] i \neq c \longrightarrow a_i = 0, b_i = 0$.*

In this case, $f(\vec{x})$ can be written as $f(x) = a_0 + a_cx_c$ and $g(\vec{x}')$ can be written as $f(x') = b_0 + b_cx'_c$.

Furthermore, $f(\vec{x})$ and $g(\vec{x}')$ are **strongly separable** *if $a_c = 0$, $b_c = 0$ is true or if $a_c = b_c$ is true. Otherwise, they are* **weakly separable**.

Table 2 Strong separability test

Condition	No Solution
$a_c = 0, b_c = 0$	$a_0 \neq b_0$
$a_c \neq 0, b_c = 0$	$a_c \nmid (b_0 - a_0)$
$a_c = 0, b_c \neq 0$	$b_c \nmid (a_0 - b_0)$
$a_c = b_c \neq 0$	$a_c \nmid (a_0 - b_0)$ or $a_c > (a_0 - b_0)$

Table 2 provides an algorithm for proving nonexistence of a solution of two linear functions that are strongly separable. The first column specifies the values of a_c and b_c and the second column provides the sufficient conditions for the system to have no solution.

The first case is trivial. Given both $a_c = 0$ and $b_c = 0$, the array access is strictly $A[a_0]$ and $A[b_0]$ where both a_0 and b_0 are constants. Independence follows from the fact that $a_0 \neq b_0$.

The second case follows from the fact that if $a_0 + a_c x_c = b_0$, we can rewrite the equation as $a_c x_c = b_0 - a_0$, which implies that solution exists if and only if $a_c \mid (b_0 - a_0)$. The third case follows the same argument as well.

The last case can be written as $a_0 + a_c x_c = b_0 + a_c x'_c$. Again, we can rewrite the equation as $a_c(x'_c - x_c) = (a_0 - b_0)$. If a solution exists, then a_c must divide $(a_0 - b_0)$. Also, since $x_c < x'_c$ must be true, then we know that $a_c \leq (a_0 - b_0)$ has to hold as well.

If the two equations are weakly separable, we can conclude that $a_0 + a_c x_c = b_0 + a_c x'_c$ has no solution if $\gcd(a, -b) \nmid (b_0 - a_0)$ [51].

This concludes the two tests currently implemented within *OPT_GlobalDep-Analysis*. Although the tests are not comprehensive, they do cover all the applications we targetted. Many improvements could be introduced to increase the power of the analysis and further disprove dependence.

OPT_GlobalDepAnalysis treats $a_0, a_1, a_2, ...a_n, a_{n+1}, b_0, b_1, b_2, ...b_n, b_{n+1}$ of $f(\vec{x})$ and $g(\vec{x}')$ as constants only if they are actually compile time constants. In cases where these variables are not known constants, JikesRVM provides a loop analysis that can prove that they are loop invariant. In that case, we can symbolically evaluate some of the dependence tests and disprove dependence by means of a runtime check.

As shown in a later section, the whole dependence graph does not need to be fully available before the parallelization process begins. A suggested improvement to the implementation would be to improve execution time of the (just-in-time) compilation process by computing dependences lazily in a demand-driven fashion. Computation of dependence within a loop can be reserved to the cases where the loop is GPU execution favorable.

Other possibilities for future work are to add additional tests if the separability test fails. Next, the full *GCD Test* could be applied. Banerjee [16] provided many more dependence tests suitable for different types of loops. Loop iteration spaces shaped like triangles or trapezoids are not covered in our implementation and can be added. In general, a system with ranged inequalities can be difficult to solve and may require more powerful yet computationally intensive approaches such as integer programming.

3.3 Algorithm

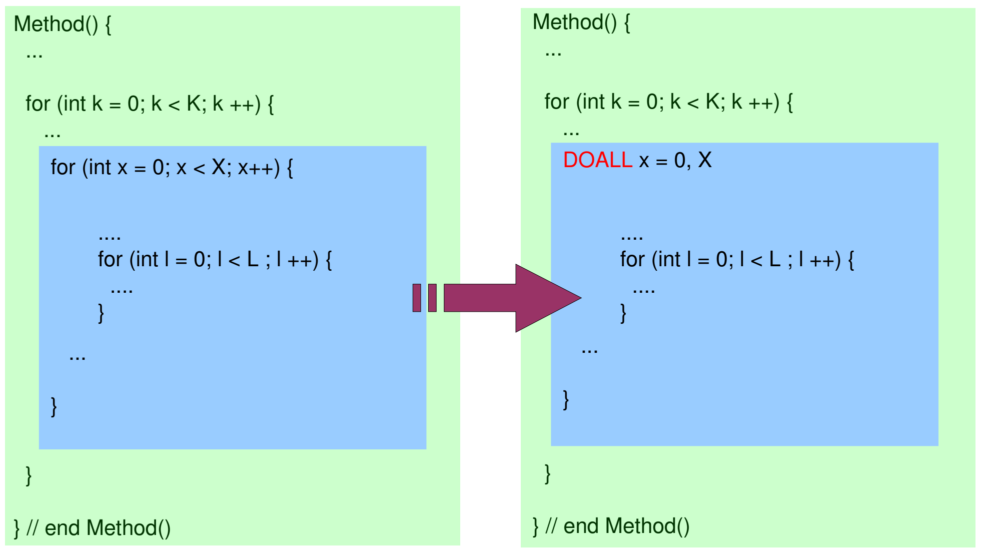
This section describes the core parallelization process of the compiler implemented in *OPT_Parallelization*. Using the dependence graph described in the previous section, we can identify independent loop iterations that can be executed in parallel. The algorithm that we use is based on a very well known algorithm (*parallelize* [51]). However, because the GPU programming model differs from that of CPUs and vector units, the existing algorithm needs to be modified.

There are two major approaches to automatic parallel execution. They are usually called **parallelization** and **vectorization**. While both focus on loop iterations of a program, they have a slightly different goal.

3.3.1 Parallelization

Parallelization is usually referred to as execution of a single program on multiple CPUs. Programmers who have a very firm understanding of the high level intention of the program, usually express explicit parallelism by means of **task-level parallelism**. Multiple threads performing identical as well as different tasks are created to operate on different data sets. However, task-level parallelism is often difficult to discover automatically since the compiler knows very little about the high level concepts of the program. In most cases, an auto-parallelizing compiler will focus on loop structures.

The basic idea is to find loop-level parallelism between iterations. Because thread creation is usually an expensive process, the compiler should always try to parallelize as many instructions as possible. Therefore, the traditional algorithm



```
Method() {
  ...

  for (int k = 0; k < K; k ++) {
    ...
    for (int x = 0; x < X; x++) {

        ....
        for (int l = 0; l < L ; l ++) {
          ....
        }

      ...

    }

  }

} // end Method()
```

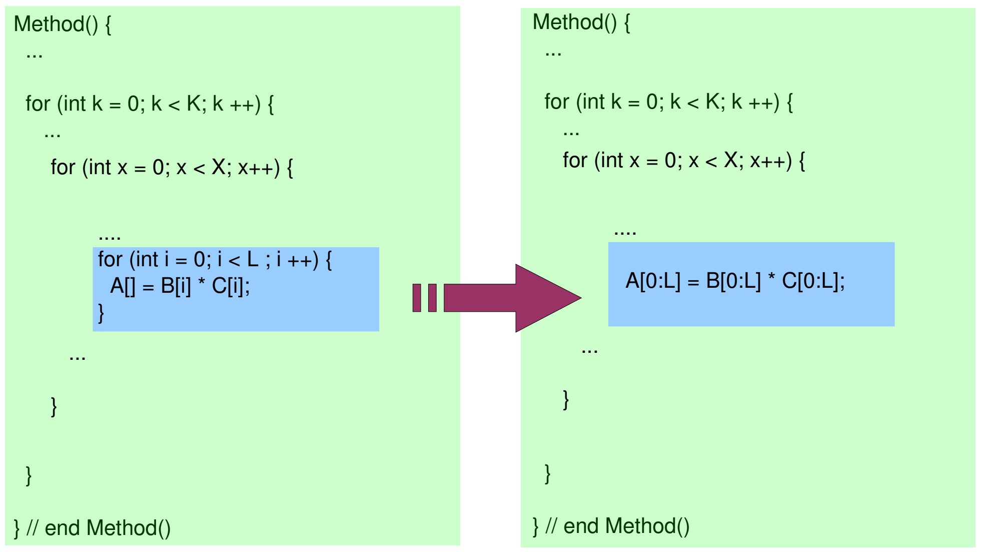
```
Method() {
  ...

  for (int k = 0; k < K; k ++) {
    ...
    DOALL x = 0, X

        ....
        for (int l = 0; l < L ; l ++) {
          ....
        }
      ...

    }

  }

} // end Method()
```

Figure 3.3: Parallelization scenario

operates on the loop tree by greedily attempting to parallelize the outer loops before recursively searching in child loops. However, aggressively distributing a large code portion to each CPU can result in a higher chance of loop-carried data dependences. Fortunately, threads can be synchronized by means of locks or barriers. Figure 3.3 shows a possible auto-parallelization scenario where the outermost *for* loop is translated into a Fortran-like *DOALL* loop that is executed in parallel.

3.3.2 Vectorization

Vectorization, on the other hand, refers to the use of vector instructions of the target machine architecture to perform otherwise scalar operations. For example, the instruction *ADDPS* from the Intel Streaming SIMD Extensions is capable of adding two vectors of size four in parallel [5]. Vectorization mainly exploits data parallelism, meaning that the programmer's intention is to perform the same operation to each element of a group of data. Vector instructions are often multiples of a single operation with no control structures. Execution is always parallel. Each single operation must be independent. Figure 3.4 demonstrates a typical loop based vectorization process performed by an auto-vectorizing compiler. Instead of iterating through each element of an array A, the whole loop has been replaced with a



```
Method() {
  ...

  for (int k = 0; k < K; k ++) {
    ...
     for (int x = 0; x < X; x++) {


            ....
            for (int i = 0; i < L ; i ++) {
              A[] = B[i] * C[i];
            }
        ...

     }


  }

} // end Method()
```

```
Method() {
  ...

  for (int k = 0; k < K; k ++) {
    ...
    for (int x = 0; x < X; x++) {


            ....
            A[0:L] = B[0:L] * C[0:L];

        ...

    }


  }

} // end Method()
```

Figure 3.4: Vectorization scenario

single vector instruction that operates on A.

3.3.3 GPU Parallelization

From the SPMD programming model programming model described earlier, we can see that current GPU architecture is not strict data parallelism as in vectorization nor is it strict task-level parallelism as in multi-threading. Current GPGPU techniques operate on data streams so one might that consider such programming model relates closely to SIMD applications. However, because the GPU is capable of control flow, it should not be viewed as a SIMD machine. While the GPU can handle complicated control flow like a threaded CPU, it is still limited by the data stream model. Much like a threaded CPU where thread creation is a significant overhead, GPU initialization is also a source of overhead. Finally, most GPUs are not capable of performing synchronization of data, making it less task oriented.

Figure 3.5 demonstrates a typical program parallelized using GPGPU techniques. In many cases, the programmer's goal is to perform similar computationally intensive operations on each element of some input data and store the result in another data set. The program on the left illustrates the programmer's intent

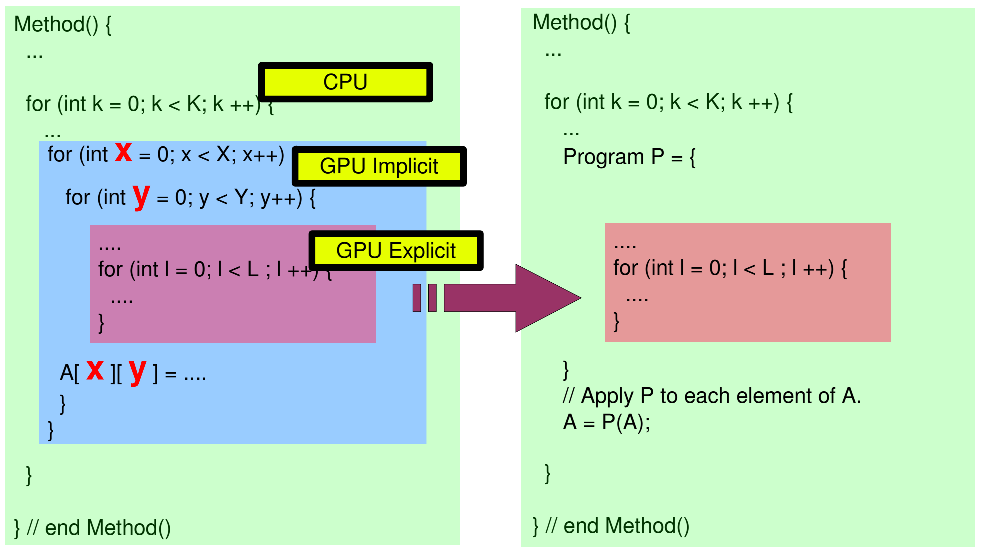

Method() {
...
for (int k = 0; k < K; k ++) {
CPU
...
for (int x = 0; x < X; x++)
GPU Implicit
for (int y = 0; y < Y; y++) {
....
for (int l = 0; l < L ; l ++) {
GPU Explicit
....
}
A[x][y] = ...
}
}
}
} // end Method()
Method() {
...
for (int k = 0; k < K; k ++) {
...
Program P = {
....
for (int l = 0; l < L ; l ++) {
....
}
}
// Apply P to each element of A.
A = P(A);
}
} // end Method()

Figure 3.5: GPGPU code pattern

when no parallel hardware is available. She must iterate through each element of the loop.

The program on the right illustrates a GPGPU version of the same program. The two data iterating loops have been removed. Instead, iterations of the loop have been replaced by a single fragment program. The program will be invoked by passing a two-dimensional input array to the program. The GPU will apply the program to all individual data elements of the array. Results will be stored in the output array. This programming model closely reflects the SPMD programming model used to describe the GPU in figure 2.1.

Almost all GPU execution favorable programs will exhibit this type of pattern. The most interesting part of this transformation is that the x and y loops no longer exist in the final program. Instead, they are implicit within the GPU program. For that reason, we are going to label such loops as *GPU-Implicit*. On the other hand, loops that are inside *GPU-Implicit* loops become loops of the GPU executable program. They will be labeled as *GPU-Explicit*. Finally, non-parallelizable outer parents of the GPU loops will be labeled as *CPU* loops.

3.3.4 Classification of Loops

A compiler targeting the GPU must not only identify parallelizable loops, but it must also decide, for each loop, whether to implement it on the CPU, to make the GPU implement it implicitly by directing it to execute a fragment program once for each iteration of the loop, or to implement it explicitly inside the fragment program. In this section, we formulate constraints that these decisions must satisfy for a collection of nested loops, and in the next section, we give an algorithm that computes a solution to these constraints.

The constraints are defined on a loop nesting tree. The root of the tree represents the whole program as a loop that is iterated exactly once. In addition, each loop in the program is represented by a tree node. For each loop, the loops nested directly within it become its children in the loop nesting tree. Each node in the tree must be classified as either *CPU*, *GPU-Implicit*, or *GPU-Explicit*.

There are no limitations on the kinds of loops that may be classified as *CPU* loops. Thus, a safe (but perhaps inefficient) solution is to classify all loops as *CPU* loops.

A *GPU-Implicit* loop, as the name suggests, is implemented by directing the GPU to execute a fragment program once for each iteration of the loop. In order for a loop L to be *GPU-Implicit*, it must fulfill the following requirements.

Restriction 1. *The parent of GPU-Implicit loop L in the nesting tree must be either CPU or GPU-Implicit.*

The outermost *GPU-Implicit* loop will be the control change from CPU to GPU. Once a *GPU-Implicit* loop starts, the GPU will be in charge of the program until the end of the loop. Within a *GPU-Implicit* loop, there can be no more *CPU* loops.

Restriction 2. *If the parent L' of L is also GPU-Implicit, L must be tightly nested within L' (i.e., L must be the entire body of L').*

Multiple loops may be implemented implicitly by the GPU, but only if all of them are tightly nested immediately within one another. This way, the body of the innermost *GPU-Implicit* loop becomes the GPU executable program used in the SPMD model.

Restriction 3. *No loop-carried true data dependence can exists between instructions of L.*

Iterations of L will be the fragment program that the GPU executes implicitly for each value of the induction variable. The order of execution is not necessarily preserved as the GPU executes iterations in parallel. Restriction 3 ensures that changing the order does not change the semantics because of Theorem 1. Surprisingly, anti-dependence between loop iterations is allowed. The reason is that before GPU execution begins, all the array data must be copied into the GPU. The data copying, which will be discussed further in Section 3.3.6, has the same effect as renaming, which breaks any anti-dependence.

Restriction 4. *For each array store $A[i_1, i_2, ...i_n]$ inside a GPU-Implicit loop, the dimension n of the store must equal the number of GPU-Implicit loops, and the i_k must be the induction variables of the GPU-Implicit loops, in order of nesting, with i_1 being the induction variable of the outermost GPU-Implicit loop.*

This final and perhaps the strongest restriction ensures that any program in the GPU will not have any scatter memory write. Every *GPU-Implicit* iteration can write only to the memory location associated with that iteration.

Finally, a *GPU-Explicit* loop is implemented explicitly in the code of the fragment program. The only requirement is that it must be nested (not necessarily tightly) inside a *GPU-Implicit* loop or an other *GPU-Explicit* loop. However, since a *GPU-Explicit* loop is part of the body of a *GPU-Implicit* loop, Restriction 4 must still hold. Within the *GPU-Explicit* loop, there should be no true data dependences carried by any of the *GPU-Implicit* loops (Restriction 3), but dependences carried by the *GPU-Explicit* loops are allowed.

3.3.5 Identifying Loop Types

The algorithm to decide whether each loop should be executed on the CPU or implicitly or explicitly on the GPU begins by identifying the index expressions occurring in stores in each loop. It applies the following definition to each loop.

Definition 14. *For a loop L in the loop nesting tree,* WriteIndices(L) *is defined as follows. If the body of L contains an instruction that cannot be supported on the GPU, then* WriteIndices(L) $= \top$. *Otherwise, if the body of L contains no array writes, then* WriteIndices(L) $= \bot$. *Otherwise, if all array writes in the body of L have the same index vector $(i_1, \ldots, i_n)$ and all the i_k are induction variables of distinct loops, then* WriteIndices(L) $= (i_1, \ldots, i_n)$. *Otherwise,* WriteIndices(L) $= \top$.

Listing 8 WriteIndices example

```
int x = 0;

for(int i = 0; i < 100; i++) {
  for(int j = 0; j < 100; j++) {
    A[i][j] = ...;
  }
}

for(int i = 0; i < 100; i++) {
  for(int j = 0; j < 100; j++) {
    A[i][j] = ...;
    B[0][j + j] = ...;
  }
}
```

In Listing 3.3.5, the first loop clearly has *WriteIndices* of (i, j). The second loop contains two array writes of different indices so the *WriteIndices* is $\top$.

A loop that cannot be implemented on the GPU because it contains unsuitable instructions or because it writes to arrays using inconsistent indices will have WriteIndices(L) $= \top$. Otherwise, WriteIndices of a loop is the unique index vector used for array writes in the loop.

Next, the algorithm computes, for each loop, the maximal set of loops that are tightly nested within it, using the following definition.

Definition 15. *For a loop L in the loop nesting tree,* TNLoops*(L) is defined* *as follows. If the entire body of L is another loop L', then* TNLoops(L) = TNLoops$(L') \cup \{L\}$*. Otherwise,* TNLoops$(L) = \{L\}$.

Finally, the algorithm traverses the loop nesting tree searching for loops that will become the outermost *GPU-Implicit* loops. When there are multiple possibilities, it is preferable to select the outermost loop possible to maximize the amount of processing moved to the GPU. Therefore, the traversal proceeds from the root of the tree to the leaves, so that it considers outer loops before inner loops. When considering a given loop, the algorithm checks that the loop and other loops tightly nested within it cover the induction variables needed for array stores occurring in the loop, and that the candidate loops do not carry dependences. The algorithm also considers the possibility of interchanging the tightly-nested loops. This makes parallelization possible even if the original nesting order is inconsistent with the array store index vector, or extra loops are nested in between those that define the induction variables used in array store indices. To determine whether loops can be interchanged, the algorithm uses the standard technique of identifying interchange-preventing dependences [51]. The overall parallelization algorithm is shown in Listing 9. it is invoked on the root of the loop nesting tree.

Listing 9 GPU parallelization algorithm

```
Algorithm Parallelize(loop $L$):
1: if WriteIndices($L$) = $(i_1, \dots, i_n)$
   and $\{i_1, \dots, i_n\} \subseteq$ TNLoops($L$)
   and no dependences are carried by loops $i_1, \dots, i_n$
   and TNLoops($L$) can be interchanged so the outermost $n$ loops are $i_1, \dots, i_n$,
   in this order then
2:   interchange TNLoops($L$) in this way
3:   generate GPU program for body of loop $i_n$
4:   replace loop $L$ with code to execute GPU program
5: else
6:   for each child loop $L'$ of $L$ in the loop nesting tree do
7:     Parallellize($L'$)
```

Figure 3.6 shows the dependence graph output after the parallelization phase. Besides the dependence information shown by the edges, each node has been color coded to represent the loop classification process. For this example, the algorithm

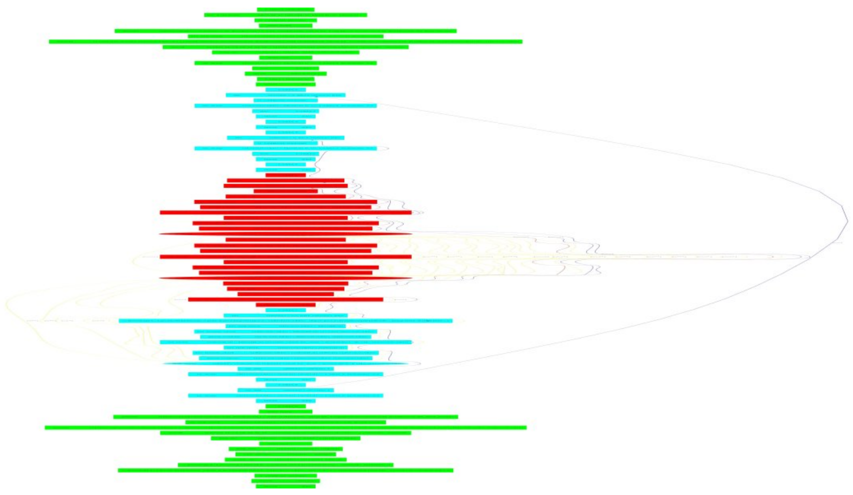


Figure 3.6: Sample output of the dependence graph after parallelization

discovered two *GPU-Explicit* loops and the sole loop nested inside them becomes the *GPU-Implicit* loop. The green colored nodes are instructions that have not been parallelized and remain a *CPU* loop. Cyan colored nodes are instructions that have been classified as *GPU-Implicit* loops. The red colored nodes are classified as the *GPU-Explicit* loops (which also are part of *GPU-Implicit* loops). Not only is this final graph representation of the parallelization process useful for debugging the compiler's parallelization phase, it is helpful for providing feedback to the users of the compiler. An Integrated Development Environment (IDE), for example, could use the information provided by this graph to inform the user which parts of their code is GPU parallelizable and which dependences are preventing parallelization.

3.3.6 Data Transfer

Graphics cards have dedicated memory with a very high transfer rate to the graphics processor. However, GPU computations cannot directly access main memory, and CPU instructions cannot directly access GPU memory. The speed-up of using the GPU may be limited by the overhead of copying data between main memory and GPU memory. This section proposes a cost model to determine whether executing code on the GPU is beneficial despite the copying overhead.

Figure 3.7 shows the execution time of a matrix multiplication benchmark. The set of plots in blue is the execution of a plain CPU execution. The set of plots in red shows a parallelized version for the GPU using our compiler. The graph shows that for matrices of size 120 $\times$ 120 or less, the CPU outperforms the GPU due to the overhead associated with using the GPU. We will propose a execution cost model to estimate the execution time needed for CPU execution and GPU execution. At runtime, we will decide whether the speed-up from the parallel GPU execution overcomes the associated overhead.

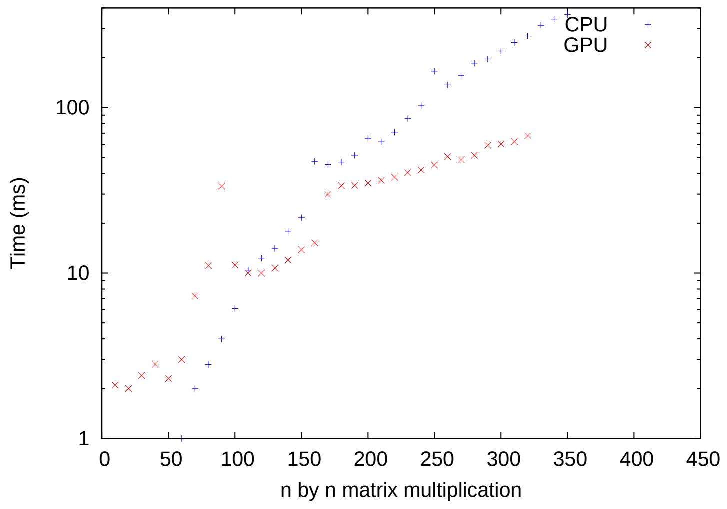


Figure 3.7: Plain matrix multiplication on GPU and CPU

The model estimates the time that a loop nest will take to execute on both the CPU and the GPU (including copying overhead). With hundreds of models of CPUs and GPUs in use today, no single formula is suitable for all configurations. Therefore, we propose a parameterized formula, in which the parameters can be tuned to the specific target hardware on which the code will execute.

The value of each of the parameters to the model becomes available at one of three different stages of compilation: when the JIT compiler is installed on the machine, when the JIT compiler compiles the loop, and whenever the compiled code executes the loop. When the JIT compiler is installed, micro-benchmarks are executed to estimate the processing power of the CPU and the GPU. These parameters remain constant for all programs. The estimated number of instructions in the body of the loop becomes known either when the loop is compiled or when the loop executes (if other loops are nested within it and their iteration counts depend on runtime values). Whenever the compiled code prepares to execute the loop, the number of iterations and the size of the input and output data become known. At that point, all the parameters are known, and the compiled code uses the model to decide whether to execute that instance of the loop on the CPU or the GPU.

Listing 10 Cost estimation

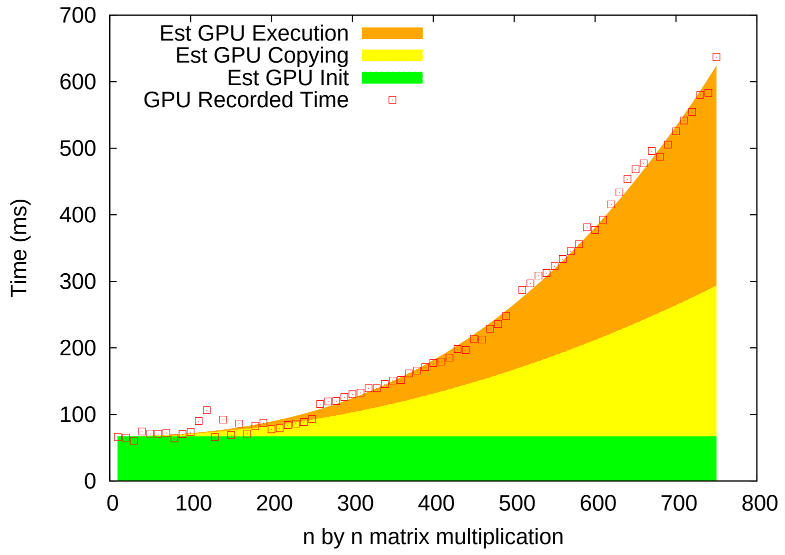
$$Cost_{cpu} = t_{cpu} \times insts \times A_{out}.size$$
$$Cost_{gpu} = t_{gpu} \times insts \times A_{out}.size \quad + \quad copy \times \sum_{A \in Ainout} A.size \quad + \quad init$$

$Cost_{cpu}$ estimates the time needed to execute all iterations of the loop on the CPU. The parameter t_{cpu} is the average time needed to execute one bytecode instruction as determined by the off-line micro-benchmarks. We assume that all instructions require the same amount of time, though a more precise model could divide instructions into different classes. The parameter *insts* is the expected number of instructions to be executed in the body of the loop. We assume that conditional branches are taken 50% of the time and that nested loops execute for ten iterations, unless their iteration count is a known constant. The parameter $A_{out}.size$, the size of the output array, becomes known when the loop is to be executed. The loop will iterate once for each element in the output array. The estimated cost is the product of these three parameters.

The GPU processing time $Cost_{gpu}$ is modelled as a product of three similar parameters, but two additional terms are added to model data transfer. The parameter *copy* estimates the time needed to copy one floating point number to or from the GPU memory, and is multiplied by the number of elements in the input and output arrays. If the same array is both read and written, it is counted twice. The parameter *init* is a constant term estimating the time needed to set up the GPU to execute a given shader program.

To determine the fixed parameters of the model (i.e. t_{cpu}, t_{gpu}, *copy*, and *init*), a benchmark is executed on both the CPU and GPU on a range of test inputs of different sizes and the actual execution times are recorded. Least squares regression is performed to determine the parameter values that most closely reflect the observed times. We will call these benchmarks the *training benchmarks*.

Figure 3.8 demonstrates a cost estimation of a matrix multiplication kernel. The green area represents the initialization cost ($init$). The yellow area represents the copying cost ($copy \times \sum_{A \in Ainout} A.size$). The orange unit represents the execution cost ($t_{gpu} \times insts \times A_{out}.size$). The sum of the three areas represents an estimate of the total running times ($Cost_{gpu}$). The points along the curve represent the actual recorded running time. As the figure shows, our estimation closely models the actual running time for this particular benchmark. This is partly due to the fact that the training benchmark is the same as the actual benchmark. Section 4 will further investigate the effects of different training benchmarks.



- $t_{gpu} = 7.81 \times 10^{-8}$ ms / instruction
- $copy = 1.01 \times 10^{-4}$ ms / element
- $init = 66.57$ ms

Figure 3.8: Estimated cost vs. actual cost of execution

3.4 Multi-pass Extension

Experience shows that data transfer occupies a significant portion of the execution time and is the main source of performance degradation. Therefore it is highly beneficial to reduce the amount of data copying between the GPU's memory and the main memory. A common pattern in which this is especially important is that of a loop that repeatedly applies some operation to a single array. For example, this pattern occurs in the Successive Over Relaxation (SOR) benchmark from the Java Grande Suite and in stencil applications [30]. The following is a simplified version of the kernel in SOR:

Listing 11 Successive over relaxation kernel

```
for (int k = 0; k < 30; k++) {
    for (int x = 1; x < size - 1; x++) {
        for (int y = 1; y < size - 1 ; y++) {
            Y[x][y] =
                (omega * X[x][y]) + (one_minus_omega * (
                            X[x-1][y] +
                            X[x+1][y] +
                            X[x][y-1] +
                            X[x][y+1]));

        }
    }
    swap(X,Y)
}
```

Each iteration of the outermost loop reads from an input array, performs a set of computations and writes to a new array. At the end of the iteration, the input reference and output reference are swapped in preparation for the next iteration. We assume that the input and output arrays are not aliased. There are no loop carried dependencies within the two inner loops. Using the algorithm described in section 3.3, we can parallelize by classifying the loops as shown in Figure 3.9.

In Figure 3.9, each execution of the loop outside the outermost *GPU-Implicit* loop requires a texture transfer from the CPU to the GPU. In this case, the array M and M' are needed and will be copied into $TEXTURE\_M$ and $TEXTURE\_M'$ accordingly. The program P will strictly operate on $TEXTURE\_M$ and $TEXTURE\_M'$. After each GPU execution, all textures that had been modified need to be copied back into the virtual machine's address space. This type of transfer is very expensive. Since the output textures are not modified between the end of the iteration and the beginning of the next GPU execution, we would like to keep those data within the GPU.

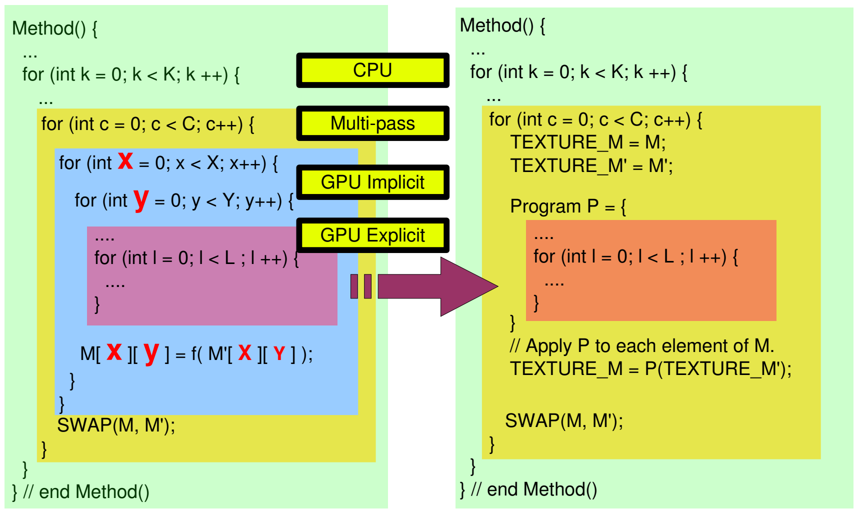
To efficiently handle this common case, we introduce a fourth kind of loop: *Multi-pass* loops. Like a CPU loop, a *Multi-pass* loop executes on the CPU, and its body may contain GPU loops. However, the following restrictions ensure that it is safe to copy the data to and from the GPU memory only once, rather than every time a GPU implicit loop executes. Restriction 5 ensures that the *Multi-pass* loop is outside all *GPU-Implicit* loops. Restriction 6 enforces that any array copied into the GPU is not used outside of the GPU. Thus, all data transfer for all loops inside the *Multi-pass* loop can be done once before the *Multi-pass* loop begins and after it finishes.

Restriction 5. *The parent of a Multi-pass loop must be a CPU loop and contain at least one child loop that is GPU-Implicit.*

Restriction 6. *All array reads or writes to an array must strictly reside inside GPU-Implicit children of the Multi-pass loop or strictly outside of all GPU-Implicit children but not both.*

We have adapted the algorithm from Listing 9 to find *Multi-pass* loops. After the algorithm finishes classifying the original three loop types, potential *Multi-pass* loops are identified by examining the parents of outermost *GPU-Implicit* loops. Loops satisfying the above two restrictions are classified as *Multi-pass* loops. In our implementation, the introduction of *Multi-pass* loops shows an average performance increase of 19.6% for the SOR benchmark with a data size of 625 and a number of iterations ranging from 1 to 100.

Figure 3.10 shows a *Multi-pass* variant of the loop classification of Figure 3.9. The texture copying has been moved outside of the loop that has been deemed as *Multi-pass*. The *Multi-pass* extension effectively eliminates $K - 1$ downloads as well as $K - 1$ uploads per texture. For large K, the speed-up is highly noticeable. Figure 3.11 shows the performance of SOR for an input of 100×100 elements after

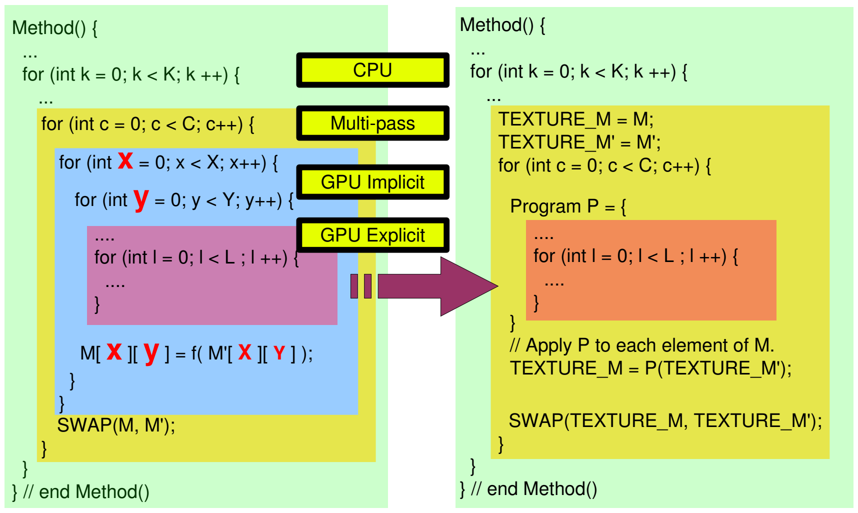


```
Method() {
 ...
 for (int k = 0; k < K; k ++) {
   ...
   for (int c = 0; c < C; c++) {
     for (int x = 0; x < X; x++) {
       for (int y = 0; y < Y; y++) {
         ....
         for (int l = 0; l < L ; l ++) {
          ....
         }
         M[ x ][ y ] = f( M'[ x ][ y ] );
       }
     }
     SWAP(M, M');
   }
 }
} // end Method()
```

```
Method() {
 ...
 for (int k = 0; k < K; k ++) {
   ...
   for (int c = 0; c < C; c++) {
     TEXTURE_M = M;
     TEXTURE_M' = M';

     Program P = {
       ....
       for (int l = 0; l < L ; l ++) {
        ....
       }
     }
     // Apply P to each element of M.
     TEXTURE_M = P(TEXTURE_M');

     SWAP(M, M');
   }
 }
} // end Method()
```

Figure 3.9: Multi-pass loop with redundant copying

1 to 10 successive relaxations. The green line shows the execution time without the *Multi-pass* extension while the red line shows the execution time with *Multi-pass*. Without *Multi-pass*, copying overhead is significant. On the other hand, with the *Multi-pass* extension, additional iterations of SOR are almost free.



```
Method() {
 ...
 for (int k = 0; k < K; k ++) {
   ...
   for (int c = 0; c < C; c++) {
     for (int x = 0; x < X; x++) {
       for (int y = 0; y < Y; y++) {
         ....
         for (int l = 0; l < L ; l ++) {
          ....
         }
         M[ x ][ y ] = f( M'[ x ][ y ] );
       }
     }
     SWAP(M, M');
   }
 }
} // end Method()
```

```
Method() {
 ...
 for (int k = 0; k < K; k ++) {
   ...
   TEXTURE_M = M;
   TEXTURE_M' = M';
   for (int c = 0; c < C; c++) {
     Program P = {
       ....
       for (int l = 0; l < L ; l ++) {
        ....
       }
     }
     // Apply P to each element of M.
     TEXTURE_M = P(TEXTURE_M');

     SWAP(TEXTURE_M, TEXTURE_M');
   }
 }
} // end Method()
```

Figure 3.10: Multi-pass loop without redundant copying

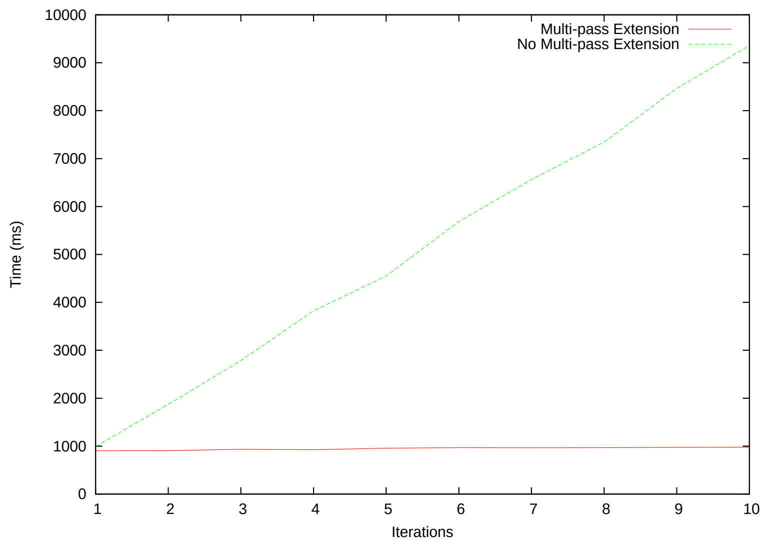


Figure 3.11: SOR performance

3.5 Java Specific Issues

The algorithm in Section 3.3 is generic in that it should be applicable to different programming languages. This section discusses obstacles specific to the choice of Java bytecode as the source language. As we will see, implementing parallelization in a just-in-time compiler makes it possible to use run-time information to effectively overcome these otherwise difficult obstacles.

3.5.1 Java Arrays

A key problem in parallelizing Java is its lack of real multi-dimensional array data structures. Java supports only one-dimensional arrays; multi-dimensional arrays are simulated as arrays of arrays. Figure 3.12 demonstrates the differences in Java's array implementation as compared to other programming languages. The left of Figure 3.12 shows how a typical programming language such as C or Fortran represent a multi-dimensional array. A four-by-four matrix is usually represented by a continuous block of memory. The right of the figure shows the Java representation of a four-by-four matrix.

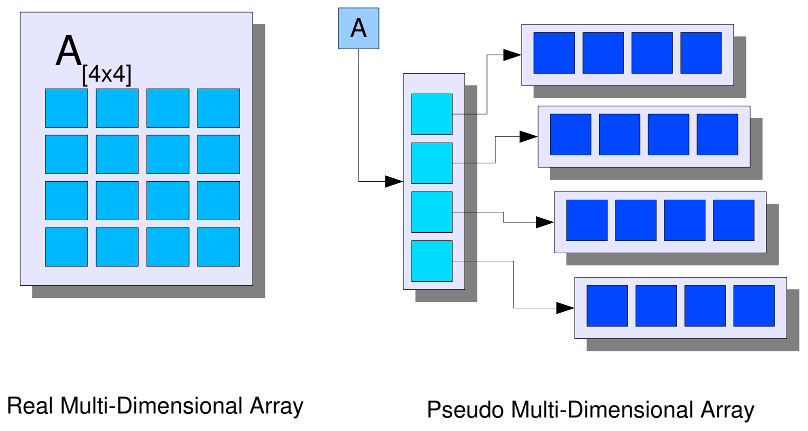


Figure 3.12: Array Implementations

The first difference is that an array in Java is a non-primitive object that is always accessed by reference. References behave very much like pointers, hence the first level of indirection implied by the first arrow from the variable A. Second, multi-dimensional arrays in Java are actually arrays of array references. Despite the complication, much of the implementation is transparent to the programmer. However, it is possible to create programs that make analysis very difficult.

Our optimizations must work on multi-dimensional arrays represented in this way; restricting them to work only on one-dimensional arrays would severely limit the programs to which they can be applied. Since the compiled Java bytecode does not have a concept of multi-dimensional arrays, the programmers' intended uses of multi-dimensional arrays are lost when the source code is compiled and are replaced with multiple dereferences. Consider the following code segment:

Listing 12 Simple Sequential Example

```
float [][][] K = .....;
.....
K[1][2][3] = 25.0f;
```

The above would be translated to the same bytecode as the following equivalent Java program:

Listing 13 Simple Sequential Example

```
float [][][] K = .....;
.....
float [][] tmp1 = K[1];
float []   tmp2 = tmp1[2];
tmp2[3] = 25.0f;
```

Finally, JikesRVM would translate the bytecode into the following (simplified) HIR code:

Listing 14 Simple Sequential Example

```
ref_aload                tmp1([[F) = K([[[F), 1
ref_aload                tmp2([F)  = tmp1([[F,d), 2
float_astore             25.0, tmp2([F), 3
```

In Listing 14, the first instruction loads the pointer stored in $K[1]$ to $tmp1$. The second instruction dereferences the value stored in $tmp1$ and store it in $tmp2$. Finally, the last instruction stores 25.0 to the memory address that $3+tmp2$ points to.

From HIR code similar to Listing 14, we would like to recover the original intention of the array element store of 25.0 to $K[1][2][3]$. The single array access now appears as several instructions, each accessing one dimension of the array. The translator must recover the original index vector from these separate instructions. Listing 3.5.1 shows a pseudo-code algorithm to recover such information. The algorithm, which performs a single pass of the code, takes advantage of the HIR being in SSA form. An `aload` or `astore` is recognized as an array *read* or *write*, respectively. If the unique definition reaching the base of load/store S_1 is also an array load S_2, the two statements are linked together. This is repeated until the definition reaching the base of the array access is no longer an array access. The chain of array accesses discovered in this way gives the full multidimensional array index vector.

The array recovery algorithm is implemented by *OPT_ArrayAnalysis*. *OPT-_ArrayAccess* objects can be of type read or write depending on the array access. These objects are stored within an *OPT_ArrayAccessDictionary* object. Consumers of the *OPT_ArrayAccessDictionary* can retrieve an array access using an *OPT_Instruction* as key. Caution must be taken when recovering array information. In cases where the definition comes from a different basic block or if the definition comes from other instructions besides a *ref_aload*, little is known about where the array originates. For such cases, those instructions will be associated with an *OPT_ArrayAccess* object with an *UNKNOWN* type. The *UNKNOWN* type will force future compilation phases to perform parallelization much more conservatively.

Listing 15 Array recovery algorithm

```
Algorithm ARRAY RECOVER(Program $P$):
for $inst \in$ Instructions($P$) do
    if $inst$ is $aload$ or is $astore$ then
        $dim \leftarrow 0$
        $\vec{I}[dim] \leftarrow$ array index of $inst$
        $var \leftarrow$ array pointer of $inst$
        $def \leftarrow$ get single reaching definition of $var$
        while $def$ is $ref\_aload$ do
            $dim \leftarrow dim + 1$
            $\vec{I}[dim] \leftarrow$ array index of $def$
            $var \leftarrow$ array pointer of $def$
            $def \leftarrow$ get single reaching definition of $var$
        return $var$, $\vec{I}$ in reverse order
```

3.5.2 Inter-array Aliasing

Even with the multi-dimensional information fully recovered, analysis on Java arrays is still difficult. One reason is that two lexically different variable references can point to the same data. This phenomenon, called **aliasing**, can drastically alter some of the results of the dependence analysis that were described in Section 2.3. Since arrays are always accessed by reference, lexically different array variables can reference the same array. Aliasing between arrays introduces dependences that are not considered in traditional dependence analysis for languages without aliasing.

The loop below demonstrates the problem. A dependence analysis that treats A and B as distinct arrays will find no dependence. However, if A and B reference the same array, **S1** has a loop-carried dependence with itself, so the loop should not be parallelized.

Listing 16 Aliasing problem.

```
for(int i = 1; i < 100; i++) { // L1
    A[i] = B[i - 1]; // S1
}
```

Static alias analysis is complicated, often imprecise, and difficult to do efficiently enough to be included in a JIT compiler. Like described in the work by Artigas et al. [14], our implementation detects aliasing using only runtime checks. Optimized code that assumes arrays are unaliased is protected by guards that check this assumption. To minimize the number of runtime checks, guards are inserted only when array aliasing would cause a loop carried dependence that prevents parallelization.

During the dependence graph creation phase, all array accesses are considered to be to the same array. However, a dependence edge caused by different array variables is marked as *AliasOnly* and annotated with the pair of Java array variables that must be aliased for the dependence to occur.

The parallelization algorithm will continue to parallelize a loop even if it carries dependences, as long as all of those dependences are marked as *AliasOnly*. When such a loop is parallelized, the variable pairs corresponding to the dependences are added to a list of pairs that must be checked for aliasing in the guard.

3.5.3 Intra-array Aliasing

The array of arrays implementation of multi-dimensional arrays mentioned in the previous section is not totally hidden from to the programmer. A programmer can create strange irregular arrays in Java that make some of the analysis very difficult. Figure 3.14 shows three array variables A, B and C. Both A and B are two dimensional arrays while C is a single dimension array.

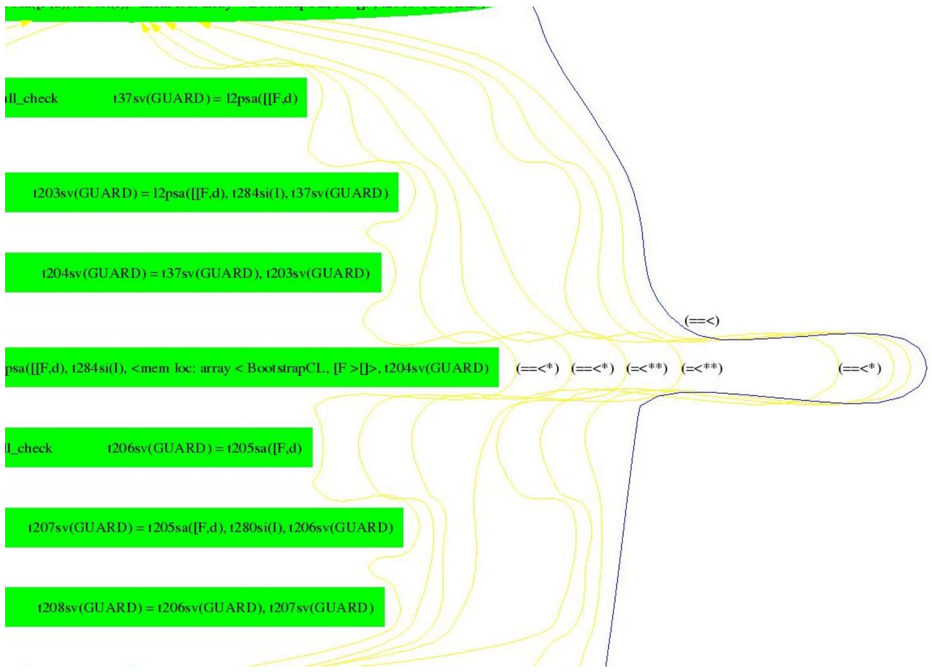

ll_check
t37sv(GUARD) = l2psa([[F,d)
t203sv(GUARD) = l2psa([[F,d), t284si(I), t37sv(GUARD)
t204sv(GUARD) = t37sv(GUARD), t203sv(GUARD)
(==<)
psa([[F,d), t284si(I), <mem loc: array < BootstrapCL, [F >[]>, t204sv(GUARD)
(==<*)
(==<*)
(=<**)
(=<**)
(==<*)
ll_check
t206sv(GUARD) = t205sa([F,d)
t207sv(GUARD) = t205sa([F,d), t280si(I), t206sv(GUARD)
t208sv(GUARD) = t206sv(GUARD), t207sv(GUARD)

Figure 3.13: AliasOnly edges in dependence graph dump

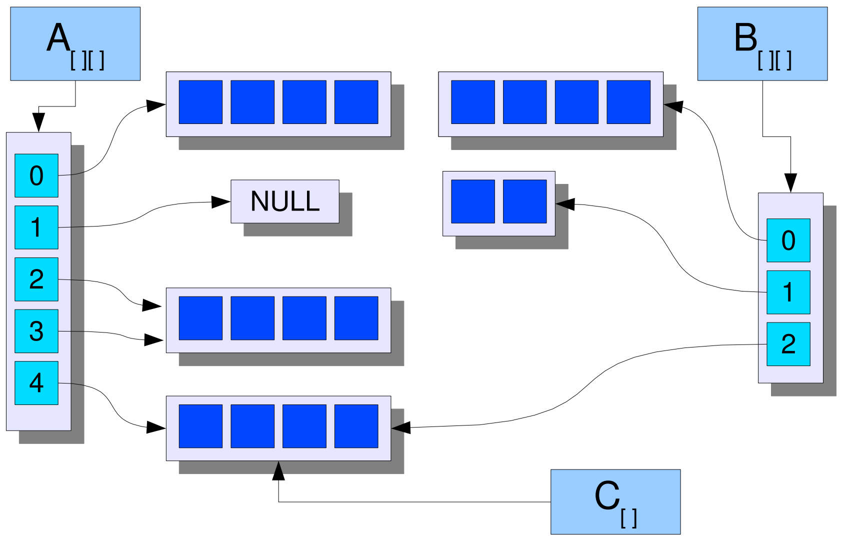


Figure 3.14: Irregularities of Java arrays

The first irregularity is **Intra-array Aliasing**. Rows 2 and 3 of A are an example of this phenomenon. Given this configuration, $A[2][i]$ and $A[3][i]$ $\forall_{i\in[0:4]}$ address the same data, which is impossible in programming with a real multi-dimensional array implementation. Intra-array aliasing can also happen between two array variables. $B[2]$ and $A[4]$ also point to the same array data, meaning that $B[2][i]$ and $A[4][i]$ $\forall_{i\in[0:4]}$ address the same array data element. To further complicate the matter, it is possible to have an array variable that is one dimension smaller to alias a sub-array of a bigger array. C is an example this phenomenon, where C points to $A[4]$. Accessing $C[i]$ and $A[4][i]$ $\forall_{i\in[0:4]}$ also address the same data.

Aliased sub-arrays introduce unexpected dependences, since a write to a given location also writes to other aliased locations. Let us consider an example where intra-array aliasing can affect data dependence analysis.

Suppose we have the following program:

Listing 17 Intra-array aliasing in parallelization

```
for(int i = 0; i < 5; i++) { // L1
    A[i][0] = A[i][0] * 2; // S
}
```

Because S only refers to the array A, we will try to apply Theorem 2. Without using the separability test, we can list out the iteration space: $0, 1, 2, 3, 4$. Every iteration accesses a different array index and according to Theorem 2, there should be no loop-carried data dependence. However, this is not the case if A points to an array like the one in figure 3.14. $A[2][0]$ and $A[3][0]$ are actually the same element, therefore $S[2] \rightarrow S[3]$ or $S \rightarrow_1 S$. Theorem 2 fails when there is intra-array aliasing.

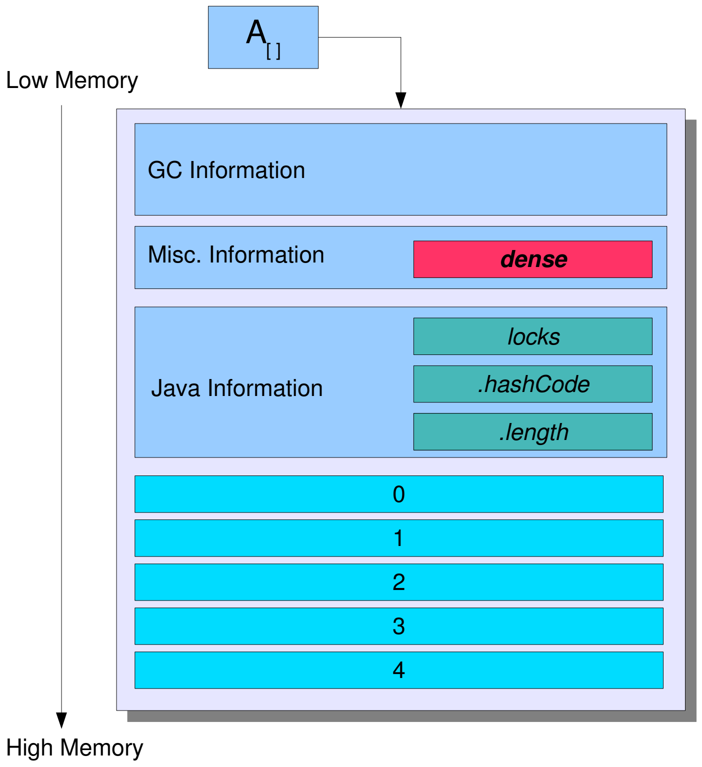


Figure 3.15: JikesRVM array layout

Since we can only apply the theorem on arrays that do not have intra-array aliasing, our analysis must detect such irregularities and avoid parallelizing the code when they occur. Fortunately, these irregularities are rare, so this restriction has little effect on the number of realistic programs that can be parallelized. To cheaply and conservatively ensure that parallelized code executes only on arrays of unaliased sub-arrays, our implementation adopts the dense flag technique of [34]. An extra one-bit flag is added to the header of every multi-dimensional array.

Figure 3.15 shows how JikesRVM represents an array object in memory. Each array object in JikesRVM is divided into four sections. The first section contains garbage collection related information such as reference counters and other markers. The second section has been alloted for research purposes. The third section stores some Java specific information such as hashcode, length of the arrays and lock information. Finally, the last section contains n pointers to each element of the array.

We are going to use an extra bit in the second section to store what we call the *dense* flag.

Definition 16 (Dense Array). *An array of arrays A is* **dense** *if the elements of A are not* `NULL` *and they are not inter-array aliased.*

The dense flag of array A is set to true if A is known to be dense. It is set to false if A might not be dense. Using a runtime detection approach, we will attempt to discover possible intra-array aliasing. When the array is created using the *multianewarray* bytecode instruction, the flag is set to true. The array returned by this instruction always has unaliased sub-arrays. When executing any instructions that could cause this property to be violated, such as overwriting one of the sub-arrays of the array, the flag is reset to false. The following example demonstrates how the flag is set:

Listing 18

```
float [][] K = new float[M][N]; // K.dense = true
K[0][0] = 1.0f; // K.dense = true
K[0] = new float[N] // K.dense = false;
```

Given a 2D array reference A, if the dense flag of A has not been set to $false$, this implies that A does not contain any intra-array aliasing. This is not true for an array of dimension greater than two. The dense flag only signifies that the immediate dimension of sub-arrays is dense. The sub-arrays themselves can be dense or not dense. Consider the following example:

Listing 19

```
float [][][] K = new float[2][2][2]; // K.dense = true
K[0][0] = new float[2];
// K[0].dense is false but K.dense remains true
```

To quickly validate that any given n-dimensional array A contains no intra-array aliasing, we must recursively check each sub-array of A. We can safely conclude A

contains no intra-array aliasing within itself if the dense flag of A is true and each sub-array element of A contains no intra-array aliasing within itself.

Given two 2D non-aliased array references A and B, the sub-array elements of A and B can not be intra-array aliased if both dense flags of A and B are true. This is true because if there is some $A[x]$ that points to some $B[x']$, a reference store to A must have been executed. This will in turn set the dense flag of A to be false. Again, for arbitrary n-dimensional array, this is not always true. Listing 20 is such an example and the dense flag check must recursively check each element's density.

Listing 20

```
float [][][] A = new float[2][2][2];
float [][][] B = new float[2][2][2];
A[0][0] = B[1][1];
```

The parallelization process will require some code splitting. Initially, *OPT_ArrayAnalysis* will create *OPT_ArrayAccess* objects by assuming that no intra-array aliasing can occur. *OPT_GlobalDepAnalysis* will proceed as usual, again, assuming that intra-array aliasing does not exist. Finally, when code is generated, run time checks are inserted to validate the assumptions. Consider the following program:

Listing 21

```
for(int i = 0; i < N; i++) {
        for(int j = 0; j < N; j++) {
                A[i][j] = B[i][j] * B[i][j];
        }
}
```

The above will be transformed as the following:

Listing 22

```
if (IsDense(A) && IsDense(B)) {

  // GPU version
  ...

} else {
  for(int i = 0; i < N; i++) {
    for(int j = 0; j < N; j++) {
      A[i][j] = B[i][j] * B[i][j];
    }
  }
}
```

Thus, the parallelized code can take advantage of the invariant that the subarrays of each array are not aliased and that they are not null. The guard that tests the dense flag ensures that, if these conditions could be violated, the array is processed on the CPU instead.

3.5.4 Bounds Checks

Every array access in Java can throw a `NullPointerException` or `ArrayIndexOutOfBoundsException`. According to the Java Language Specification, the exceptional control transfer must occur exactly at the time of the access causing the exception, and any side-effects occurring before it must be preserved. The Java exception semantics thus impose a control dependence between every pair of array accesses. To safely parallelize Java code, it is necessary to ensure that these exceptions cannot occur in the code.

For every loop compiled for the GPU, the implementation also compiles a fallback CPU version with the standard exception semantics. Before executing the loop on the GPU, the implementation performs conservative checks to ensure that all array accesses will be to non-null arrays and within the array bounds. If any check fails, the CPU version of the loop is executed instead of the GPU version. For every array reference accessed in the loop, the implementation checks that it is non-null and loop-invariant before the loop. Every array index expression must be either loop-invariant or of the form $ax + b$, where x is a loop induction variable and a and b are loop-invariant. The bounds on an index expression in this form can be determined from the bounds on x, and compared to the array size on entry to the loop. Consider the following code segment:

Listing 23 Bounds example

```
for(int i = 0; i <= 100; i++) {
                    ... = A[4 * i + 1];
}
```

Knowing that i goes from 0 to 100 in the iteration space will allow us to conclude that reads of A range from 1 to 401. The guard can be inserted like so:

Listing 24 Bounds guard example

```
if (A.length >= 401) {
          // GPU Version
     ...
} else {
          for(int i = 0; i <= 100; i++) {
                    ... = A[4 * i + 1];
          }
}
```

The checks are slightly more complicated when the access is multi-dimensional. Suppose we have the following code segment.

Listing 25 2D array bounds check example

```
for(int i = 0; i <= 10; i++) {
        for(int j = 0; j <= 100; j++) {
                ... = A[i][j];
        }
}
```

We might have to check each sub-array like so:

Listing 26 2D array bounds guard example

```
if (A.length >= 10 && A[0].length >= 100) {
        // GPU Version
    ...

} else {
        for(int i = 0; i <= 10; i++) {
                for(int j = 0; j <= 100; j++) {
                        ... = A[i][j];
                }
        }
}
```

The above check is safe except for another irregularity caused by Java's pseudo multi-dimension array. Each sub-array can be of a different length. This is demonstrated in Figure 3.14. $B[1]$ has a length of 2 as opposed to 4 like the other elements of B. A bounds check guard before the loop as shown in Listing 26 would alter the semantics of the original program if exceptions are thrown.

Definition 17 (Rectangular Array)**.** *An n dimension Java array A is* **rectangular** *if each sub-array $A[0].length = A[1].length = A[2].length... = A[A.length - 1].length$ and each sub-array $A[0], A[1], ...A[A.length - 1]$ are also rectangular.*

Surely, we can iterate the whole array and verify that each sub-array indeed has the required length. However, that would introduce a significant overhead. We would like a quick check to determine if an array is rectangular. When an multi-dimensional array is created with the *multianewarray* bytecode instruction, it is guaranteed to be rectangular. Once again, we can deploy a flag to record if an sub-array has been reassigned. This flag, however, would also contain the same value as the dense flag. Our prototype therefore relies on the dense flag to determine if the array is both dense and rectangular. The example in Listing 25 would be transformed into the following:

Listing 27 Modified 2D array bounds guard example

```
if (IsDense(A) &&
    A.length >= 10 &&
    A[0].length >= 100) {
    // GPU Version
    ...
} else {
   for(int i = 0; i <= 10; i++) {
      for(int j = 0; j <= 100; j++) {
          ... = A[i][j];
      }
   }
}
```

Although our approach is very conversative, it is applicable in most situations. Irregular shaped arrays exist but are rarely deployed by programmers. Also, our approach requires no static computation. Overhead is introduced when subarrays are being assigned but rarely do programmers use arrays in such fashion. Also, many array bound check elimination techniques exist in the literature that could be added to our compiler to rule out the bounds check. For example, ABCD [22] is a demand-driven lightweight bounds check elimination algorithm that was once part of JikesRVM. Regioning [33] is another approach that could be added to subdivide the iteration space into two sections: a *safe* section where we know for sure that no exceptions can be thrown and a *non-safe* section where exceptions might be thrown.

3.5.5 Recovering Control Flow

Control flow is expressed in Java bytecode in an unstructured form using *GOTO* and conditional *GOTO* instructions, but most shader programming languages support only structured control flow expressed using `if-then-else` and `while` constructs, as does our analysis. Therefore, the transformation must recover the structure of the control flow.

There are two fundamental control structures that can be created within the Java programming language: Loops and `if-else` structures.

JikesRVM includes a mechanism to recover loop structure in its high level intermediate representation HIR [50]. However, conditional branches are left in the form of unstructured branches.

In most cases, the compiler will be able to recreate the original structured control flow. However, there are cases where this is impossible without restructuring the program. The problem arises when `GOTO`s are used to branch into the middle of control structures. This type of control flow is impossible to express in the Java language because the lack of support for `GOTO` statements. However, the compiled bytecode operates purely on `GOTO`s. Therefore, a hand-assembled `class` file or other optimization passes can generate this type of program.

Definition 18 (Reducible loop)**.** *A loop is* **reducible** *if there is only one unique entry point.*

There are two main types of control flow that cannot be recovered directly. The first type is irreducible loops. JikesRVM's loop analysis does not support irreducible loops. At a high optimization level, the optimizing compiler will not attempt to optimize the method if it contains irreducible loops and our parallelization phase is not executed. In our work, we will focus strictly on methods in which all loops are reducible.

After loops have been identified, each loop can be considered as a single node in a control flow graph. The resulting graph will be acyclic. For example the simple control flow structure shown in Figure 3.16 can be translated to the Java equivalent in Listing 28.

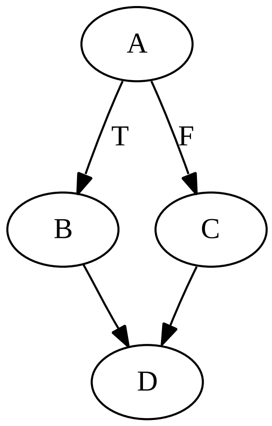


Figure 3.16: Simple GOTO control flow

Listing 28 Simple Recover Example

```
// A:
if (...) {
    // B:
} else {
    // C:
}
// D:
```

However, some control flow graphs cannot be directly translated into `if-else` constructs even though they are acyclic. Figure 3.17 is an example of such phenomenon. The node E in the figure is a common `else` block of two different `if-else` constructs. This situation has no Java source equivalent. In particular, the code within node E will have to be cloned as shown in Figure 3.18 and the generated code will look like that shown in Listing 30.

Instead of determining which blocks need to be cloned, our implementation uses a recursive graph traversal to generate code for each node. The nodes are visited in a way such that a node that needs to be cloned will be visited multiple times, creating multiple copies of that node. The pseudo-code shown in Listing 29 is used to translate an acyclic control flow graph into a structured control flow using if-then-else statements. The basic idea is to traverse the control flow graph from the two successors of a conditional branch until a basic block is reached that post-dominates the condition.

Definition 19 (Post-dominance)**.** *A basic block A* **post-dominates** *B if all paths from B to any of the function's return statements must pass through A.*

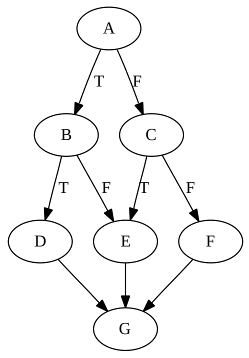


Figure 3.17: Sharing of else blocks

Listing 29 Algorithm for recovering if-then-else structure.

```
Algorithm GENERATE(BB block, BB condBl):
 1: if block = null or (condBl ≠ null and block postdominates condBl) then
 2:     return empty list
 3: ret ← GPU code for block
 4: if block ends in unconditional branch or falls through then
 5:     return ret + generate(successor of block, condBl)
 6: if block ends in conditional branch with condition cond then
 7:     if cond then
 8:         return ret + GENERATE(branch successor of block, block)
 9:     else
10:         return ret + GENERATE(fall-through successor of block, block)
```

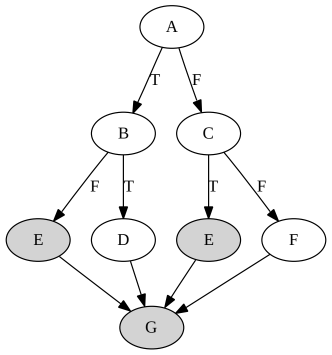


Figure 3.18: Cloning else blocks

Listing 30 Complicated Recover Example

```
// A:
if (...) {
   if (...) {
      // B:
   } else {
      // *E* (cloned):
   }
} else {
   if (...) {
      // *E* (cloned):
   } else {
      // F:
   }
}
// G:
```

3.6 Implementation Summary

In this chapter, we have provided a generic algorithm that discovers GPU executable loops. Of all the GPU executable loops, we provided a cost model that estimate performance changes if the loop is executed on the GPU. We also provided an extension that improves performance by eliminating extra data transfer. Finally, we addressed problems that arise when implementing our algorithm in a Java environment.

Chapter 4

Results

To quantify the performance improvements from executing code on the GPU and to evaluate the accuracy of the cost model, we measured the execution times of a set of benchmarks, which were chosen to vary in the ratio of computation to memory bandwidth required. The GPU parallelization algorithm was implemented in JikesRVM 2.9.0. The benchmarking system contained an Intel Pentium 4 CPU running at 3.0 GHz with 1 GB of memory, and an NVIDIA GeForce 7800 GPU with 256 MB of GPU memory. The machine was running Ubuntu 6.06.1 with Linux kernel version 2.6.15 and NVIDIA driver version 100.14.11.

The purpose of the benchmarks is not to demonstrate the raw computing power of the GPU compared to the CPU. This has already been demonstrated in Section 1.1 (in particular, Figure 1.1). Instead, our experiments are targeted to show that our prototype implementation can take advantage of the added computation power using our automated. Five benchmarks are used in our experiment: *mul*, *mandel*, *julia*, *matrix* and *raytrace*.

The *mul* benchmark is a simple loop that multiplies a number by itself n times, repeated for an array of m initial numbers ($10 \leq n \leq 250, 1000 \leq m \leq 20000$). The following is the simplified version of the kernel:

Listing 31 mul kernel

```
for (int j = 0; j < size; j++) {
  float t = a[j];
  for (int k = 0; k < i; k++) {
     t = t * C;
  }
  a[j] = t;
}
```

The *matrix* benchmark is a slightly more complicated micro benchmark. It multiplies two n by n matrices ($10 \leq n \leq 320$) in the following way:

Listing 32 matrix kernel

```
for (int x = 0; x < size; x++) {
   for (int y = 0; y < size; y++) {
      float s = 0.0f;
      for (int k = 0; k < size; k++) {
         s += B[x][k] * C[k][y];
      }
      A[x][y] = s;
   }
}
```

The *mandel* benchmark generates a fractal image from the Mandelbrot set. Given a single point c in the complex plane, the complex function $F_c(z) = z^2 + c$ is repeatedly applied to the starting point 0. If the output eventually converges, the point c is said to belong in the Mandelbrot set. A color is chosen for each converged point depending on the speed of the convergence. To compute the set, we repeatedly apply the function to a point up to 250 applications. The point is believed to not be in the set if it does not converge within 250 applications. Much like a 3D graphics application, the color of each complex coordinate is independent from the values of other coordinates. This type of computation is ideal for executing on GPUs.

The *julia* benchmark generates a fractal image from the Julia set which is closely related the Mandelbrot set. The same function ($F_c(z) = z^2 + c$) from the Mandelbrot set is used in computing the Julia set. However, c is now a fixed complex number. For each z in the complex plane, F is repeatedly applied in search of a fixed point. The identical technique and problem sizes as the mandel benchmark are examined.

Lastly, the *raytrace* benchmark is an implemenation of a ray caster. Given a set of n by n pixels as a view port, $n \times n$ rays are traced from the eye and tested for intersection with m spheres in a given scene. Like the two fractal benchmarks, rays are independent of each other and each ray can be computed in parallel. Each of our test units renders a scene of n by n pixels containing m spheres ($50 \leq n \leq 300, 25 \leq m \leq 250$).

One interesting observation to note is the difference between the output images of the CPU and GPU executions of *raytrace* and the fractals. In Section 2.1 we have mentioned the loss of precision using the GPU. In fact, the color values in the two sets of generated pictures are indeed different. Because of the chaotic nature of the fractal generating functions, slight differences between applications of F can result in a huge difference in output. However, differences in the output images were barely observable by human eye even at a high resolution. Figure 4.1 shows a 500-by-500 pixel image of the Julia set generated by the CPU. Figure 4.2 shows the same set generated by the GPU. The two images look almost identical.

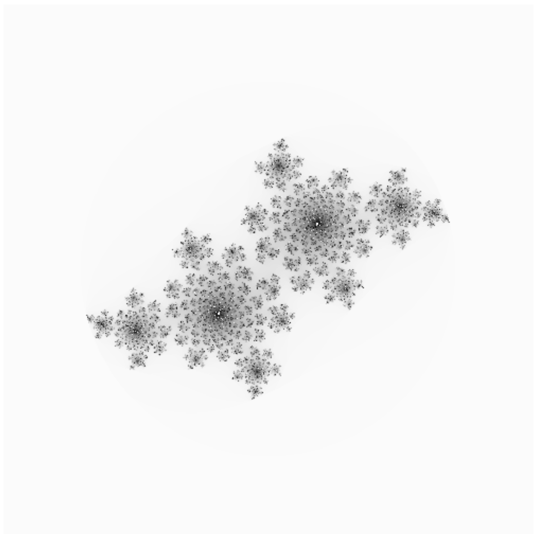


Figure 4.1: Image of Julia set by the CPU

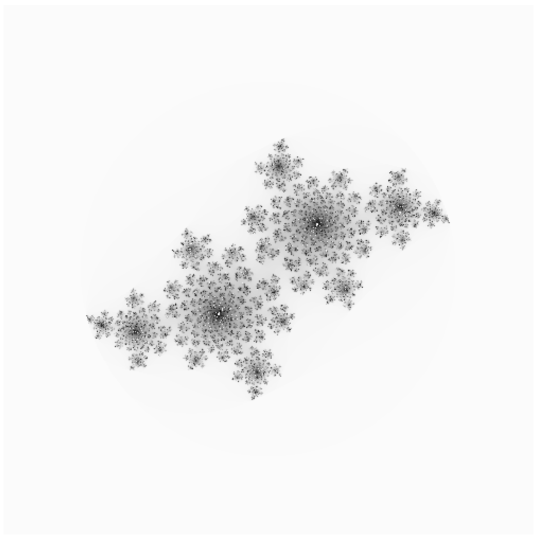


Figure 4.2: Image of Julia set by the GPU

Each input size of all the benchmarks was repeated twenty times. Each execution was separately compiled. A warm-up run is first executed, then the executed time of the actual run is recorded. Ten of the twenty runs are executed with parallelization enabled (GPU execution) and ten more runs are executed with parallelization disabled (unmodified). The averages of the ten execution times is recorded.

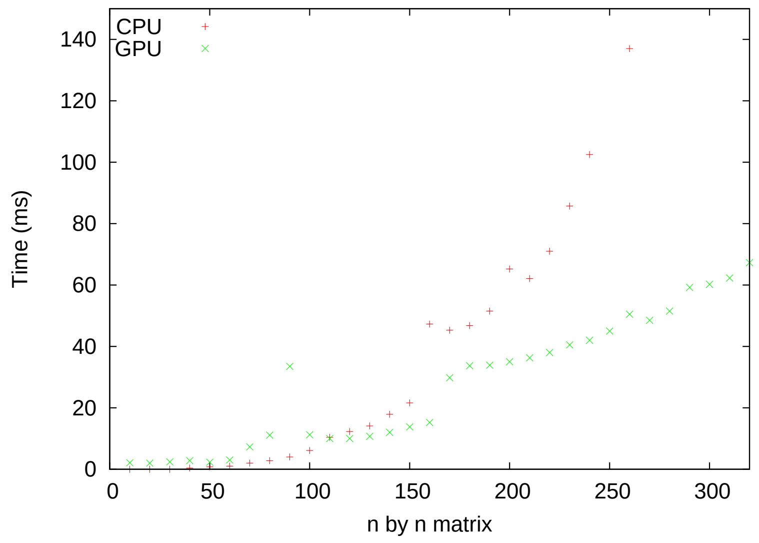


Figure 4.3: Running Time of matrix

Figure 4.3 shows the execution times of the matrix benchmark on the CPU and the GPU. For matrices of 100 by 100 elements and smaller, the copying overhead dominates GPU execution time, so the multiplication is faster on the CPU. For larger matrices, however, the GPU is faster, and the computation time increases much more slowly as the matrix becomes larger. The *mandel* and *julia* benchmarks exhibit a similar trend, as shown in Figure 4.4.

The benchmark execution times for all the benchmarks are shown in Figure 4.5. Each bar represents the total time needed to execute a benchmark on its full range of test inputs. The times are normalized to the time required to execute entirely on the CPU, with the GPU parallelization disabled; this is shown as the left-most bar for each benchmark. The right-most bar for each benchmark is the smallest time possible if the implementation made an optimal choice, for each test input, whether to use the CPU or the GPU. The bars inbetween show the execution time when the choice between CPU and GPU is made according to the model proposed in Section 3.3.6, tuned using each of the benchmarks. The second-right-most bar for each benchmark shows the execution time when the model is tuned using a combination of all benchmarks except the benchmark whose execution time is being measured.

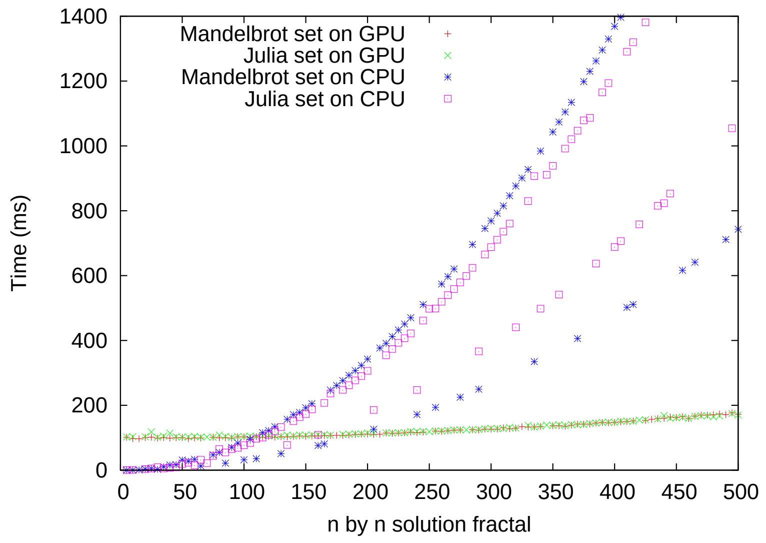


Figure 4.4: Running Time of mandel and julia

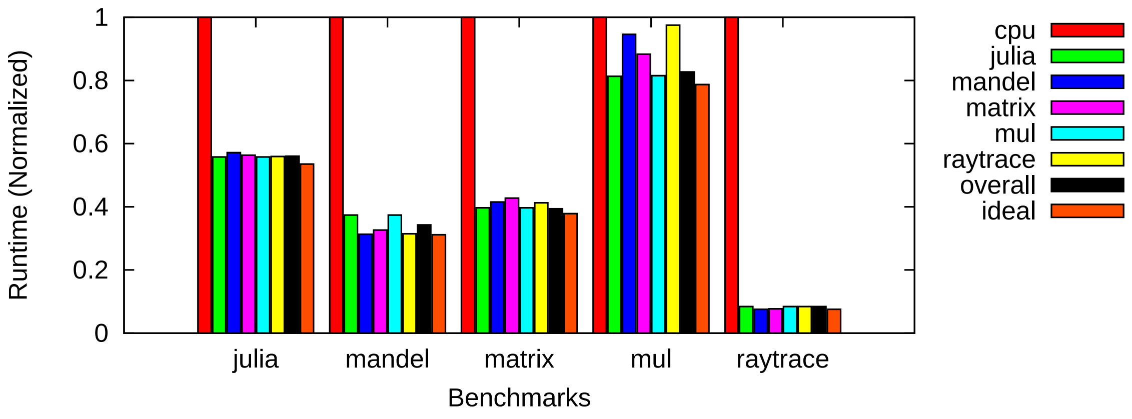


Figure 4.5: Execution Time Comparison

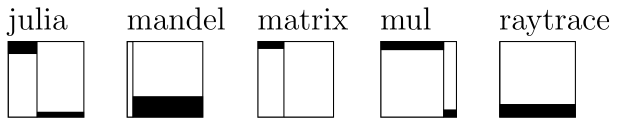


Figure 4.6: Overall cost model accuracy

The ideal speedup over the CPU ranges from 27% for *mul* to 13 $\times$ for *raytrace*. When the choice between CPU and GPU is made according to the cost model, the performance improvements are generally close to the ideal choice regardless of which benchmark is used to tune the model. When the model is tuned on all but the benchmark being measured, execution time using the cost model is within 4.7% (*julia*) to 11.5% (*raytrace*) of the ideal time.

To better understand the accuracy of the cost model, we compared the choices suggested by the model to the ideal choices. The results of this comparison are depicted in Figure 4.6, which is to be interpreted as follows. Each square represents the executions of one of the benchmarks using a cost model tuned on all but the benchmark being measured. The area of each of the squares represents the full set of test input sizes for the benchmark. The fraction of each square that is white is the fraction of test inputs for which the model makes the ideal ("correct") choice; the black area of each square represents test inputs for which the model makes the wrong choice. The area to the left of the vertical line is the proportion of inputs which can be processed faster on the CPU than the GPU, while the area to the right represents the inputs on which the GPU is faster. Thus, for example, the top-left black rectangle in each square represents the fraction of inputs for which the CPU would be faster, but the model incorrectly suggested using the GPU.

The *mul* benchmark executes faster on the CPU than the GPU on 83% of the test inputs, as shown by the square labelled mul; for the other benchmarks, the GPU is faster more often than the CPU. The *raytrace* benchmark always executes faster on the GPU than on the CPU. Most of the area of each square is white (87% on average), indicating that the model often makes the correct choice. On the *julia* and *mul* benchmarks, the model is balanced, in that it errs in both directions: it sometimes suggests using the CPU when the GPU would be faster, and vice versa. On the *mandel* benchmark, in 27% of the cases in which the GPU would be faster, the model instead suggests using the CPU. However, as Figure 4.5 shows, the effect on overall runtime is small, because the cases on which the model is incorrect are the smallest inputs, whose execution time is negligible on either processor.

We draw the following conclusions from these results. The potential performance improvement from using the GPU is very large, up to 13 $\times$ for the *raytrace* benchmark. Because of this, a large improvement is possible even when the cost model is tuned on only a single benchmark. When the cost model is tuned on a variety of benchmarks, it predicts the faster processor for 87% of the test inputs, achieving total execution times within 4.7% to 11.5% of the ideal time.

Chapter 5

Conclusions

We have presented a loop parallelization algorithm that detects loops that can be executed in parallel in the programming model exposed by modern GPU hardware. We have also addressed some of the extra overhead issues with copying data between the GPU and the CPU. In addition, we identified Java-specific obstacles to parallelization imposed by the semantics of Java, and suggested simple but effective ways to overcome those obstacles in the context of a JIT compiler. We also proposed a cost model for deciding whether it is profitable to execute a given loop on the GPU rather than the CPU. The techniques were implemented in JikesRVM, and empirically evaluated. Specifically, executing numerical code on the GPU instead of the CPU was shown to give speedups of up to 13 $\times$ on a ray casting benchmark. The cost model, when tuned on realistic benchmarks, generalizes well to other realistic benchmarks. When the cost model is used to choose between the CPU and the GPU, the resulting performance is very close to that of the ideal choice.

The choice of Java as our input language imposed lots of difficulties. Although we were able to overcome most of them, there are still cases that we fail to recognize. The ideal goal where the parallelization is totally transparent is difficult. On the one hand, the programmer can be trained to write computationally intensive code with a certain pattern in order to benefit from auto-parallelizing compilers. On the other hand, the compiler should be able to provide some feedback to the programmer to provide information such as favorable loops, parallelization-preventing dependences and failure in runtime dense flag checks. The colored dependence graph shown in

Figure 3.2 can also be incorporated within integrated development environments (IDEs) to assist programmers in performance optimization.

The GPU parallelization algorithm performs loop interchange when this is necessary to execute a loop on the GPU. In the future, we would like to increase the applicability of the parallelizer by adding some of the many other loop transformation that have been proposed [51, 8, 47] for uncovering parallelization opportunities.

Finally, GPU architectures are changing quickly. The parallelization algorithm presented in this thesis can be used as a base, and extended as necessary to take advantage of new GPU features as they are added.

Appendix A

The following is a list of extra command line options added to JikesRVM to use and debug the parallelization process.

- `-oc:parallelization=<boolean>` This is a boolean flag that enables (default) or disables GPU parallelization phrase.
- `-oc:parallel_program_implementation=<string>` This is a string option changes the code generating backend implementation. The string should be the full class name of the implementation class. The default is `jrm.rapidmind.RapidmindProgramImp`. A dummy implementation class `jrm.DebugProgramImp` will print out the pseudo shader code to standard output.
- `-oc:debug_prebuild_rm_program=<boolean>` This is a boolean flag that enables (default) or disables static ahead of time compiling of Rapidmind programs during JIT compilation.
- `-oc:debug_parallelization_analysis=<boolean>` This is a boolean flag that disables (default) or enables printing of debug information of the analysis that determines if loops are parallelizable.
- `-oc:debug_parallelization_perform=<boolean>` This is a boolean flag that disables (default) or enables printing of debug information of the analysis that translate HIR code to Rapidmind code.
- `-oc:print_global_array_dep=<boolean>` This is a boolean flag that disables (default) or enables printing of nodes and edges of the dependence graph that involve array access in text.

- `-oc:print_global_scalar_dep=<boolean>` This is a boolean flag that disables (default) or enables printing of nodes and edges of the dependence graph that involve scalar access in text.

- `-oc:print_global_carried_dep=<boolean>` This is a boolean flag that disables (default) or enables printing of nodes and edges of the dependence graph that involve loop carried dependences.

- `-oc:print_global_dep_dot=<boolean>` This is a boolean flag that disables (default) or enables printing of nodes and edges of the complete dependence graph in a Graphviz DOT file format.

Bibliography

[1] Astex. http://www.irisa.fr/caps/projects/Astex, 2007. 20

[2] NVIDIA CUDA. http://developer.nvidia.com/object/cuda.html, 2007. 9, 19

[3] Rapidmind. http://www.rapidmind.net/, 2007. 20, 21

[4] The Jamaica Project. http://intranet.cs.man.ac.uk/apt/projects/jamaica/, 2007. 20

[5] Intel 64 and ia-32 architectures software developer's manual. 2008. 30

[6] JikesRVM. http://jikesrvm.org/, 2008. 10

[7] List of all publications that use JikesRVM. http://jikesrvm.org/Publications, 2008. 12

[8] John R. Allen and Ken Kennedy. *Optimizing compilers for modern architectures: a dependence-based approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002. 12, 17, 18, 72

[9] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Syst. J.*, 39(1):211–238, 2000. 21

[10] Bowen Alpern, Dick Attanasio, John J. Barton, Anthony Cocchi, Susan Flynn Hummel, Derek Lieber, Mark Mergen, Ton Ngo, Janice Shepherd, and Stephen Smith. Implementing Jalapeño in Java. In *ACM SIGPLAN Conference on*

Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '99), 1999. 11

[11] Bowen Alpern, Maria Butrico, Tony Cocchi, Julian Dolby, Stephen Fink, David Grove, and Ton Ngo. Experiences Porting the Jikes RVM to Linux/IA32. In *2nd Java Virtual Machine Research and Technology Symposium (JVM '02)*, 2002. 10

[12] S. Amarasinghe, J. Anderson, M. Lam, and C.-W. Tseng. An overview of the SUIF compiler for scalable parallel machines. In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, San Francisco, CA, 1995. 18

[13] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 47–65, New York, NY, USA, 2000. ACM. 11, 21

[14] Pedro V. Artigas, Manish Gupta, Samuel P. Midkiff, and José E. Moreira. Automatic loop transformations and parallelization for Java. In *ICS '00: 14th Int. Conf. on Supercomputing*, pages 1–10, 2000. 48

[15] Prithviraj Banerjee, John A. Chandy, Manish Gupta, John G. Holm, Antonio Lain, Daniel J. Palermo, Shankar Ramaswamy, and Ernesto Su. The PARADIGM Compiler for Distributed-Memory Message Passing Multicomputers. In *The First International Workshop on Parallel Processing*, pages 322–330, Bangalore, India, Dec. 1994. 18

[16] Utpal Banerjee. *Dependence Analysis.* Kluwer Academic Publishers, Norwell, Massachusetts, 1997. 28

[17] Aart J. C. Bik. *Software Vectorization Handbook, The: Applying Intel Multimedia Extensions for Maximum Performance.* Intel Press, 2004. 18

[18] Aart J. C. Bik and Dennis B. Gannon. Automatically exploiting implicit parallelism in Java. *Concurrency: Practice and Experience*, 9(6):579–619, 1997. 18

[19] Aart J. C. Bik, Milind Girkar, Paul M. Grey, and Xinmin Tian. Automatic intra-register vectorization for the intel architecture. *Int. J. Parallel Program.*, 30(2):65–98, 2002. 18

[20] William Blume, Rudolf Eigenmann, Keith Faigin, John Grout, Jay Hoeflinger, David A. Padua, Paul Petersen, William M. Pottenger, Lawrence Rauchwerger, Peng Tu, and Stephen Weatherford. Polaris: Improving the effectiveness of parallelizing compilers. In *Languages and Compilers for Parallel Computing*, pages 141–154, 1994. 18

[21] David Blythe. The Direct3D 10 system. In *SIGGRAPH '06*, pages 724–734, 2006. 10

[22] Rastislav Bodik, Rajiv Gupta, and Vivek Sarkar. ABCD: eliminating array bounds checks on demand. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 321–333, 2000. 57

[23] Gerald Cheong and Monica S. Lam. An optimizer for multimedia instruction sets. In *Proceedings of the Second SUIF Compiler Workshop*, 1997. 18

[24] Jay L. T. Cornwall, Olav Beckmann, and Paul H. J. Kelly. Automatically translating a general purpose C++ image processing library for GPUs. In *Proceedings of the Workshop on Performance Optimisation for High-Level Languages and Libraries (POHLL)*, page 381, April 2006. 19

[25] A. Eichenberger. Optimizing compiler for the cell processor. In *14th Internation Conference Parallel Architectures and Compilation Techniques*, pages 161–172, 2005. 18

[26] Randima Fernando and Mark J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. 7

[27] F. Franchetti, S. Kral, J. Lorenz, and C.W. Ueberhuber. Efficient utilization of simd extensions. In *Proceedings of the IEEE*, volume 93, pages 409–425, 2005. 18

[28] Iffat H. Kazi and David J. Lilja. JavaspMT: A speculative thread pipelining parallelization model for Java programs. In *Proceedings of the 14th International Parallel & Distributed Processing Symposium (IPDPS'00), Cancun, Mexico, May 1-5, 2000*, pages 559–564, 2000. 18

[29] Andreas Krall and Sylvain Lelait. Compilation techniques for multimedia processors. *International Journal of Parallel Programming*, 28(4):347–361, 2000. 18

[30] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, Atanas Rountev, and P Sadayappan. Effective automatic parallelization of stencil computations. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 235–244, New York, NY, USA, 2007. ACM. 41

[31] Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 145–156, New York, NY, USA, 2000. ACM. 18

[32] Corinna G. Lee and Mark G. Stoodley. Simple vector microprocessors for multimedia applications. In *International Symposium on Microarchitecture*, pages 25–36, 1998. 18

[33] José E. Moreira, Samuel P. Midkiff, and Manish Gupta. From flop to megaflops: Java for technical computing. *ACM Transactions on Programming Languages and Systems*, 22(2):265–295, 2000. 57

[34] José E. Moreira, Samuel P. Midkiff, and Manish Gupta. A comparison of three approaches to language, compiler, and library support for multidimensional arrays in Java. In *JGI '01: Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, pages 116–125, 2001. 51

[35] Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997. 11

[36] Dorit Naishlos. Autovectorization in GCC. In *GCC Developer's Summit*, pages 105–118, 2004. 18

[37] Dorit Naishlos, Marina Biberstein, Shay Ben-David, and Ayal Zaks. Vectorizing for a simdd dsp architecture. In *CASES '03: Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, pages 2–11, New York, NY, USA, 2003. ACM. 18

[38] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007. vii, 1, 2

[39] Yunheung Paek and David A. Padua. Automatic parallelization for non-cache coherent multiprocessors. In *Languages and Compilers for Parallel Computing*, pages 266–284, 1996. 18

[40] Eric Petit, Sebastien Matz, and Francois. Partitioning programs for automatically exploiting GPU. *SC'06 Workshop: General-Purpose GPU Computing: Practice And Experience* *http://www.gpgpu.org/sc2006/workshop/INRIA_GPU_partitioning.pdf*. 20

[41] Christopher J. F. Pickett and Clark Verbrugge. SablespMT: a software framework for analysing speculative multithreading in Java. In *PASTE '05: The 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 59–66, 2005. 18

[42] Constantine D. Polychronopoulos, Miliand B. Gikar, Mohammad R. Haghighat, Chia L. Lee, Bruce P. Leung, and Dale A. Schouten. The structure of parafrase-2: an advanced parallelizing compiler for c and fortran. In *Selected papers of the second workshop on Languages and compilers for parallel computing*, pages 423–453, London, UK, UK, 1990. Pitman Publishing. 18

[43] Stéphane Bihan Romain Dolbeau and François. Bodin. Hmpp: A hybrid multicore parallel programming environment. In *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU 2007)*, 2007. 20

[44] J. Shin, J. Chame, and M. Hall. Compiler-controlled caching in superword register files for multimedia extension architecture, 2002. 18

[45] Jaewook Shin. Introducing control flow into vectorized code. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*, pages 280–291, Washington, DC, USA, 2007. IEEE Computer Society. 18

[46] N. Sreraman and R. Govindarajan. A vectorizing compiler for multimedia extensions. *International Journal of Parallel Programming*, 28(4):363–400, 2000. 18

[47] Michael Joseph Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. 12, 18, 72

[48] Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. 8, 10

[49] Jisheng Zhao, Matthew Horsnell, Ian Rogers, Andrew Dinn, Chris Kirkham, and Ian Watson. Optimizing chip multiprocessor work distribution using dynamic compilation. In *Proceedings of Euro-Par*, pages 28–31, 2007. 20

[50] Jisheng Zhao, Ian Rogers, Chris Kirkham, and Ian Watson. Loop parallelisation for the Jikes RVM. In *PDCAT '05: Proceedings of the Sixth International Conference on Parallel and Distributed Computing Applications and Technologies*, pages 35–39, 2005. 20, 58

[51] Hans Zima and Barbara Chapman. *Supercompilers for parallel and vector computers.* ACM Press, New York, NY, USA, 1991. 12, 14, 18, 27, 28, 29, 36, 72