Automatic Parallelization for Graphics Processing Units

Alan Leung   Ondřej Lhoták   Ghulam Lashari
D. R. Cheriton School of Computer Science
University of Waterloo
Waterloo, ON, Canada
{acleung,olhotak,glashari}@uwaterloo.ca

ABSTRACT

Accelerated graphics cards, or 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, because they are difficult to program, GPUs are used only for a narrow class of special-purpose applications; the raw processing power made available by GPUs is unused most of the time.

This paper presents an extension to a Java JIT compiler that 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 on the GPU. The paper presents 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. The cost model is parameterized so that it can be applied to different hardware combinations. The paper also presents 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.

1. INTRODUCTION

The GPU in a typical desktop PC has significantly more raw processing power and memory bandwidth than the CPU. For example, the NVIDIA GeForce 7800 GTX can perform 165 GFLOPS, while the theoretical peak rate of a dual-core 3.7 GHz Intel Pentium 965 is 25.6 GFLOPS [29]. The performance gap between GPUs and CPUs 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 SIMD unit 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" GPU (GPGPU) community has sprung up, which applies GPUs to problems other than rendering three-dimensional scenes [29]. Despite the term "general-purpose", the GPGPU community focuses on adapting specific algorithms for execution on GPUs.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PPPJ '09, August 27–28, 2009, Calgary, Alberta, Canada.
Copyright 2009 ACM 978-1-60558-598-7 ...$10.00.

The goal of our work is to make available the computational power of GPUs to people solving numerically-intensive but non-graphical problems. There is significant commercial interest from domain experts in financial modelling, oil and gas exploration, and media processing for cheap computation. Although these people are experts in their domain, they are often novice programmers, perhaps having taken one or two programming courses in Java. In some cases, they have existing prototype implementations in a high-level language running on the CPU. It is not feasible to require these domain experts to learn parallel programming, GPU architecture, C++, or template metaprogramming. However, they are willing to follow some conventions in their Java code in exchange for large speedups. We would like to empower these domain experts to experiment with tentative solutions without requiring programming experts to implement their prototypes.

To achieve this, we extend a Java virtual machine (VM) to detect loops which can be parallelized and which can be executed more quickly on the GPU than on the CPU. Thus, our solution can correctly execute any Java bytecode without modification or recompilation. In addition, if the code contains parallelizable loops, the modified VM will execute them on the GPU when that is profitable. The higher raw performance of the GPU must be weighed against the cost of transferring the input and output data 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.

We have implemented GPU parallelization in the JikesRVM [6] Java virtual machine. Like other Java VMs, JikesRVM reads Java bytecode at run time, translates it to the machine language of the target CPU, and executes it. Our extension enables the VM to generate GPU code when appropriate. Unlike some other VMs, JikesRVM does not include an interpreter: all code is compiled, first using a fast, simple compiler, and later using more sophisticated compilers for code that turns out to be significant.

The need to perform the transfomation at run time in the virtual machine is motivated by the dynamic nature of Java, compared to languages such as Fortran more traditionally used for numeric computation. In Java, most function calls are dynamically dispatched, and all arrays are dynamically allocated on the heap. Thus, purely static analyses often fail to determine exactly which code will be executed and which arrays it will manipulate. Therefore, just-in-time compilation has become the dominant technique for efficiently implementing such dynamic languages. Modern just-in-time compilers perform many of the same analyses and optimizations as static compilers, but can additionally exploit dynamic information and speculative optimization [20].

Implementing the parallelization inside an existing VM allows us to take advantage of existing key infrastructure. The adaptive optimization system [8] determines which methods use significant amounts of execution time; we focus our parallelization efforts on only those methods. Within these hot methods, JikesRVM inlines calls to other methods; thus, our analysis does not need to consider method calls. We also take advantage of other standard transformations such as loop-invariant code motion and static single assignment form.

The primary back-end we use is the RapidMind framework [4]. RapidMind is a C++ programming framework for expressing data-dependent parallel algorithms 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 using the GPU backend.

This paper 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 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 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 paper is organized as follows. Section 2 provides background on GPUs and RapidMind. Section 3 presents the GPU parallelization algorithm. Section 4 discusses obstacles specific to Java bytecode. Section 5 reports on an experimental evaluation of the cost model. Section 6 reviews related work, and Section 7 concludes.

2. BACKGROUND

2.1 GPUs

GPU architecture and associated programming models have undergone many changes in recent years, mainly in the direction of increased generality and programmability. We briefly outline these developments, and explain the overall programming model at a high level.

Traditional GPUs were organized as a fixed pipeline of dedicated stages, which operated in parallel on many data points. The tasks of the traditional GPU were to map three-dimensional vertices to two-dimensional positions on the screen, to rasterize the polygons in the scene onto a bitmap, and to colour each individual pixel of the bitmap based on lighting and texture information.

Later hardware began to allow custom programs to be executed in these stages. In particular, the fragment stage (the last stage, which colours each pixel) was most commonly used for GPGPU applications because it was one of the first to become programmable, and because it was the stage with the most raw computational power. Initially, a fragment program was a short, straight-line sequence of instructions. Predication was allowed, but no control flow. In a graphical application, the fragment program would execute once for each pixel, and its output value would determine the colour of that pixel. In a GPGPU application, the fragment program could produce an array of values in parallel, each value begin the output of one instance of the fragment program. Fragment programs could not write to arbitrary memory locations (scatter), but they could read from arbitrary locations in so-called textures (i.e. input arrays in memory). Still later hardware provided increasing levels of support for control flow in fragment programs, implemented using predication. A common way to overcome the limitations on control flow and on scatter was to divide the overall algorithm into multiple fragment programs that could each be run repeatedly.

Modern interfaces to GPUs such as CUDA [3] present a single-program multiple-data (SPMD) programming model. The GPU executes a large number of threads. Although each thread executes the same program, it is possible for the individual threads to follow different control flow paths through the program, and to read from and write to arbitrary locations in shared memory. Despite the generality of the programming model, the hardware implementing it is still composed of single-instruction multiple-data (SIMD) processors that, at any point in time, execute the same instruction on multiple pieces of data. Thus maximum performance is achieved when all of the threads follow the same or similar control flow paths. Also, the programming model provides no guarantees about the ordering of parallel writes to the same memory location. Thus, for predictable results, the programmer must ensure that no two threads will write to the same location. One way to ensure this is to forego the ability to perform scatters to arbitrary memory locations, and assign each thread a distinct set of array elements for its output, the same technique that was used by fragment programs. Use of data-dependent scatters requires domain specific knowledge about the data to guarantee the disjointness of the data-dependent write targets.

2.2 RapidMind

RapidMind is a C++ GPU metaprogramming framework which consists of two parts. 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.

In our work, we use only the RapidMind back-end, bypassing the front-end completely. Thus, no C++ code is generated, no C++ compiler is used, and no template metaprogramming is done. The parallelization phase in the VM directly generates the RapidMind IR data structures. These are passed to the RapidMind back-end to generate the GPU code.

An overview of RapidMind and how the modified JikesRVM uses it is shown in Figure 1. The right side shows the normal con-

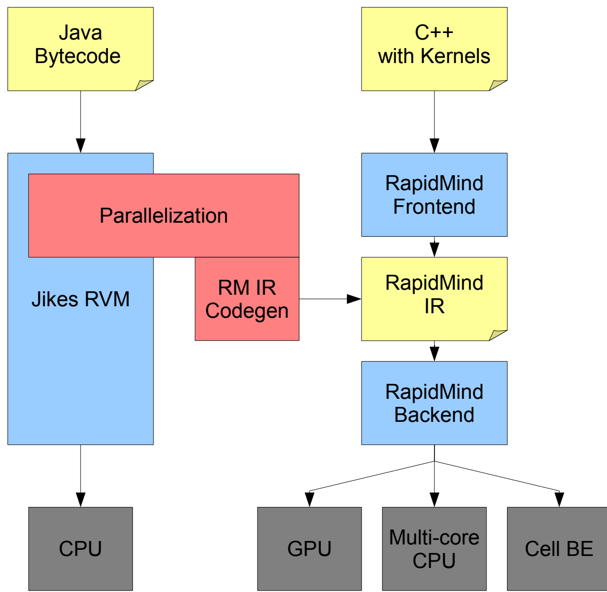


Figure 1: System Overview

figuration of RapidMind suitable for running C++ code with embedded kernels. The left side shows JikesRVM, which takes Java bytecode as input and generates machine code that executes on the CPU. We add the parallelization stage that bridges these two systems, by generating the RapidMind IR directly from the JikesRVM IR.

3. GPU PARALLELIZATION ALGORITHM

Our implementation of parallelization proceeds in three stages. The first stage recovers information about multi-dimensional array accesses that is lost in Java bytecode. This stage will be discussed later, in Section 4.2. The second stage performs dependence analysis on array accesses to construct a dependence graph [41, 38, 5]. For applications we tested, the separability test [41] was sufficient to disprove dependences. The third stage implements GPU parallelization, which generates code that will run on the GPU. We present this stage in the remainder of this section.

3.1 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. We illustrate these alternatives using the example in Figure 2. The code in part (a) of the figure is a sequential program that multiplies a matrix by a vector 10 times, producing the vector $M^{10}V$. A reasonable way to implement this algorithm using a GPU would be to write the body of the middle loop (with index `x`) as a GPU program (as shown in part (b) of the figure), and ask the GPU to execute the iterations of this loop body in parallel. This is possible because no dependences are carried by the middle loop. We call the loop a GPU-Implicit loop because the GPU implicitly executes its body multiple times when it is asked; the control flow of the loop does not appear explicitly anywhere in the generated code. We call the innermost loop (with index `y`) a GPU-Explicit loop because it appears explicitly in the GPU program in part (b). Note that the innermost loop could not have been executed in parallel because the variable `s` has a dependence carried by this loop.

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

  float[] Bn = new float[100];

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

(a)

```
float kernel(
    float x, float[][] A, float[] B) {

  float s = 0;
  for (float y = 0; y < 100; y++) {
    s += A[x][y] * B[y];
  }
  return s;
}
```

(b)

Figure 2: Repeated matrix-vector multiplication example in (a) sequential and (b) GPU code

Execution of the GPU program is triggered by code running on the CPU. Since execution of the GPU program is equivalent to performing all iterations of the `x` loop in parallel, the CPU code that should be generate should be as in part (a), but with the `x` loop replaced by a call to trigger the GPU program. The outermost loop (with index `k`) remains part of the code implemented on the CPU, so we call it a CPU loop. Note again that the outermost loop could not have been parallelized as a GPU-Implicit loop because it carries the dependence on `B`.

In the rest of this subsection, we formulate constraints that the loop classification must satisfy for a correct implementation of the input program. In the next subsection, we give an algorithm that computes a solution of the 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 L in the nesting tree must be either CPU or GPU-Implicit.*

The outer-most GPU-Implicit loop will be the control change from CPU to GPU.

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. Restrictions 1 and 2 ensure that the nesting order of the loops from outermost to innermost is CPU loops followed by GPU-Implicit loops followed by GPU-Explicit loops.

RESTRICTION 3. *No loop carried true data dependence exists between instructions of L.*

Iterations of L will be the fragment program that the GPU executes implicitly for each fragment. The order of execution is not necessarily preserved as the GPU executes iterations in parallel. Restriction 3 enforces that changing the order does not change the semantics. Anti-dependence between loop iterations, surprisingly, 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, has the same effect as renaming which breaks any anti-dependence. Loop-carried output data dependences are ruled by the following restriction.

RESTRICTION 4. *For each array store $A[i_1, i_2, ...i_n]$ inside a GPU-Implicit loop, the dimension of the store n 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 most restrictive restriction ensures that any program in the GPU not have any scatter memory write. Every GPU-Implicit iteration will 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 other GPU-Explicit loop. However, since a GPU-Explicit 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.

3.2 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 1. *For a loop L in the loop nesting tree, define* WRITEINDICES(L) *as follows:*

- *If the body of L contains an instruction that cannot be implemented 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$(L) = \top$.

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 2. *For a loop L in the loop nesting tree, define* TNLOOPS*(L) 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 the loop that will become the outer-most GPU-Implicit loop. 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 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 [41]. Listing 1 shows the overall parallelization algorithm; it is invoked on the root of the loop nesting tree.

Listing 1 GPU parallelization algorithm

```
Algorithm PARALLELIZE(loop L):
1: if WRITEINDICES(L) = (i_1, ..., i_n)
   and {i_1, ..., i_n} ⊆ TNLOOPS(L)
   and no dependences are carried by loops i_1, ..., i_n
   and TNLOOPS(L) can be interchanged so the outermost n
   loops are i_1, ..., 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 i_1 with code to execute GPU program
5: else
6:    for each child loop L' of L in the loop nesting tree do
7:       PARALLELIZE(L')
```

3.3 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.

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.

Each of the parameters to the model becomes known 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 2 Cost estimation

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

$Cost_{cpu}$ 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.

Experience shows that data transfer occupies a significant portion of the execution time and it 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 SOR benchmark from the Java Grande Suite and in stencil applications.

To efficiently handle this common case, we introduce a fourth kind of loop: *Multi-pass* loops. Like a CPU loop, a *Multi-pass* 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 the 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 A must strictly reside inside GPU-Implicit children of that loop or strictly outside of all GPU-Implicit children but not both.*

We have adapted the algorithm from listing 1 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 outer most 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 SOR benchmark with data size of 625 and number of iterations ranging from 1 to 100.

4. JAVA-SPECIFIC ISSUES

The algorithm in Section 3 was generic enough to 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.

4.1 Aliasing

In Java, arrays are always accessed by reference. Lexically different array variables can reference the same array. Aliasing between arrays introduces dependences 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.

```
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 [9], 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 array variables that must be aliased for the dependence to occur. The parallelization algorithm is modified to parallelize a loop even if it carries dependences, as long as all of those dependences are marked 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.

4.2 Multidimensional Arrays

Java supports only one-dimensional arrays; multi-dimensional arrays are simulated as arrays of arrays. 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. However, in an array of arrays, subarrays need not be of uniform length, and it is even possible for subarrays to alias each other. Aliased subarrays introduce unexpected dependences, since a write to a given location also writes to other aliased locations. To preserve behaviour, the 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 which can be parallelized.

To cheaply and conservatively ensure that parallelized code executes only on rectangular arrays of unaliased subarrays, our implementation adopts the dense flag technique of [26]. An extra one bit flag is added to the header of every multi-dimensional array. When the array is created using the multianewarray bytecode instruction, the flag is set to true. The array returned by this instruction is always rectangular and its subarrays are unaliased. When executing any instruction that could cause this property to be violated, such as overwriting one of the subarrays of the array, the flag is reset to false. Thus, the parallelized code can take advantage of the invariant that the subarrays of each array are not aliased, that they are not null, and that they are of uniform length. A guard that tests the dense flag ensures that, if these conditions could be violated, the array is processed on the CPU instead.

A related obstacle is that an access into a multi-dimensional array, when encoded in Java bytecode, appears as several instructions each accessing one dimension of the array. The translator must recover the original index vector from these separate instructions. This is done using a single pass of the code, which JikesRVM has already transformed to 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.

4.3 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 fall-back 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.

Of course, Java exceptions can also be triggered explicitly using throw instructions. The implementation does not attempt to execute a loop on the GPU if the loop contains an explicit throw.

4.4 Recovering Control Flow

Control flow is expressed in Java bytecode in an unstructured form using goto instructions. Structured control flow (using if-then-else and while constructs) is required for two reasons: first, the loop analyses depend on it, and second, shader programming languages, including the RapidMind intermediate representation, support only structured control flow. Therefore, the transformation must recover structure in the control flow.

JikesRVM includes a mechanism to recover loop structure in its high level intermediate representation HIR. However conditional branches are left in the form of unstructured branches. After loops have been identified, each loop can be considered as a single node in an acyclic control flow graph. The pseudocode shown in Listing 3 is used to translate an acyclic control flow graph into 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.

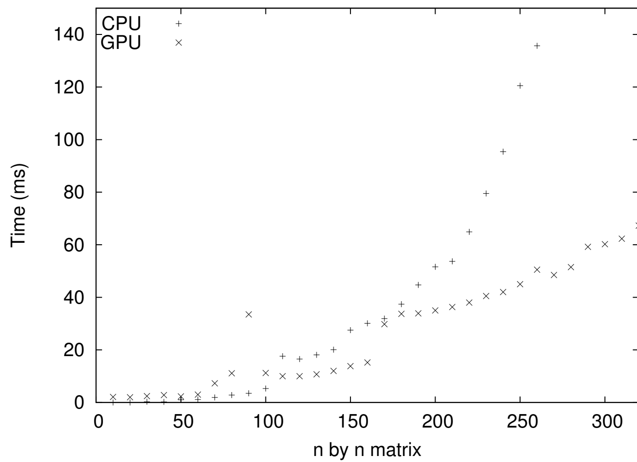
Listing 3 Algorithm for recovering if-then-else structure.

Algorithm GENERATE(BB *block*, BB *condBl*):
1: **if** *block* = null or (*condBl* $\neq$ null and *block* postdominates *condBl*) **then**
2:     **return** empty list
3: *ret* $\leftarrow$ 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:     **return** *ret* ++ if *cond* then GENERATE(branch successor of *block*, *block*) else GENERATE(fall-through successor of *block*, *block*)

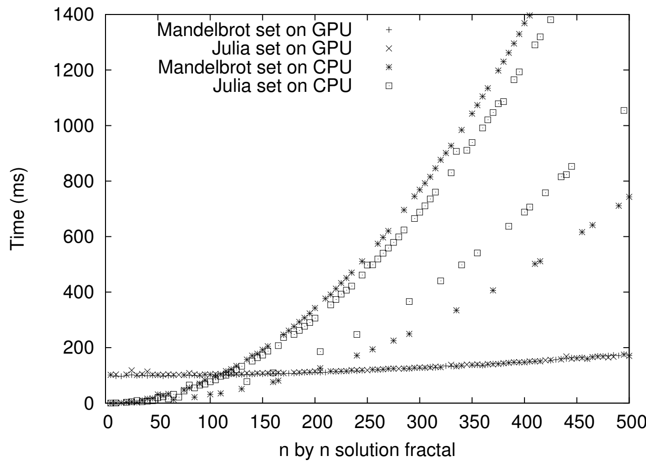
5. EMPIRICAL RESULTS

To quantify the performance improvements from executing code on the GPU and to evaluate the accuracy of the cost model, we measured the execution time 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 optimization level of the JikesRVM was set to -O2 for methods targeted for parallelization. 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.

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 matrix benchmark multiplies two n by n matrices ($10 \leq n \leq 320$). The julia and mandel benchmarks compute membership in the Mandelbrot and Julia sets, respectively, for a set of n by n complex numbers ($5 \leq n \leq 500$), using up to 250 iterations for each point. The raytrace benchmark is a ray caster that renders a scene of n by n pixels containing m spheres ($50 \leq n \leq 300, 25 \leq m \leq 250$).



9(a) Running Time of matrix



9(b) Running Time of mandel and julia

Figure 3: Comparison of CPU and GPU Execution Times

Figure 3(a) 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 3(b).

The benchmark execution times for all the benchmarks are shown in Figure 4. 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 in between show the execution time when the choice between CPU and GPU is made according to the model proposed in Section 3.3, 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.

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.

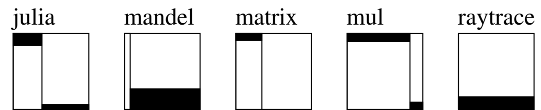


Figure 5: Overall cost model accuracy

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 5, 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 shows, the effect on overall runtime is small, because the cases on which the model is incorrect are the inputs for which the execution times are similar on both processors, so an incorrect decision incurs little performance degradation.

To summarize, 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. Although the cost model is simple (for example, it does not distinguish different instructions), it is sufficiently precise on important cases that it achieves almost the same overall performance as an ideal cost model.

6. RELATED WORK

Most existing parallelization approaches fall into two categories, depending on the hardware features that they exploit: task-level parallelism [30, 14, 33, 10, 7] and vectorization [18, 11, 23, 13, 16, 22, 28, 27, 41, 5, 38].

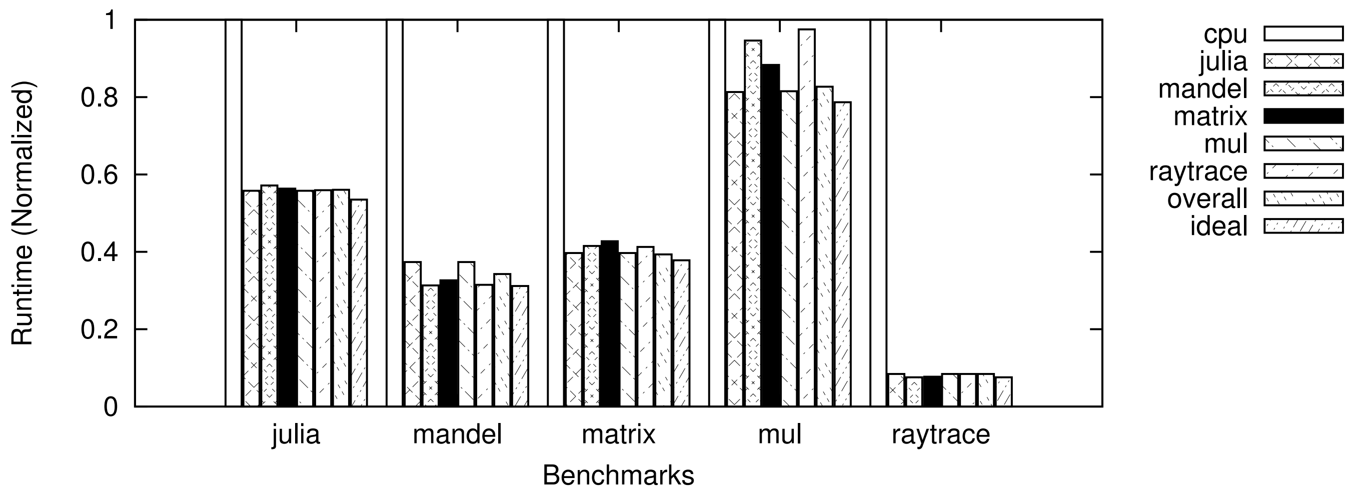


Figure 4: Execution Time Comparison

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 outer-most loop is parallelized, resulting in long tasks and few thread creations. In the context of parallelizing Java, three examples of this technique are JAVAR [12], JavaSpMT [21], and SableSpMT [32].

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 inner-most loops. The vectorization or simdization can be categorized into two principal approaches: the traditional loop-based parallelization [13, 28, 36, 24] and the basic block approach [23, 19, 35].

The loop-based vectorization technique proceeds by stripmining the loop by a factor of the vector length and then replacing each scalar instruction in the loop body by a corresponding vector instruction. The basic block approach, on the other hand, unrolls the loop by a factor of the vector length and packs each group of scalar isomorphic instructions into a vector instruction. The loop-based approach requires complicated loop transformations like loop fission and scalar expansion 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 control flow [34]. 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 multiprocessor 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 cyclic 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 [3] architecture moves even further towards a general multi-processor style of parallelism.

The hybrid nature of GPUs suggests a new kind of parallelization algorithm targeting loops in the middle of a loop nest; parallelizing an inner loop would incur high kernel startup overhead, while an outer loop is likely to contain computations not supported by the GPU and divergent control flow. We have presented one such algorithm. Another parallelization system targeting GPUs is that of Cornwall et al. [17], which performs source-to-source translations to help domain experts retarget an image processing library written in C++ to GPUs. ASTEX [1] takes a run-time approach, in that it searches for hot traces at run time that are amenable to GPU execution [31].

Zhao et al. [40, 39] have also implemented loop parallelization in the context of JikesRVM. However, rather than GPUs, their intended target is JAMAICA [2], a multi-processor parallel architecture.

Many languages and systems have been devised to provide a high-level programming model to allow developers to take advantage of the computing power provided by GPU hardware. The most closely related ones include CUDA [3], RapidMind [4], Brook [15], CGiS [25], and Accelerator [37]. In these languages, the programmer designates the loop to be executed on the GPU, and writes its body in a dedicated language. These systems provide more control over the GPU hardware, but require the programmer to learn the dedicated language. In our approach, on the other hand, the programmer writes only Java code, following conventions that allow the compiler to generate code to run on the GPU.

7. CONCLUSIONS AND FUTURE WORK

This paper has presented a loop parallelization algorithm that detects loops that can be executed in parallel in the programming model exposed by modern GPU hardware. In addition, it 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. The paper 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 one benchmark, generalizes well to

other 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 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 [41, 5, 38] for uncovering parallelization opportunities.

When the output of a loop is only read in another parallelizable loop, and both loops are implemented on the GPU, we plan to investigate whether it is possible to keep the intermediate results on the GPU, rather than copying them to the CPU and back again.

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

8. REFERENCES

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

[2] The Jamaica project. http://intranet.cs.man.ac.uk/apt/projects/jamaica/.

[3] NVIDIA CUDA. http://developer.nvidia.com/object/cuda.html.

[4] RapidMind. http://www.rapidmind.net/.

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

[6] 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.

[7] 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.

[8] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 47–65. ACM Press, 2000.

[9] 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.

[10] 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.

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

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

[13] 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.

[14] 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.

[15] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, 2004.

[16] G. Cheong and M. Lam. An optimizer for multimedia instruction sets. In *Proceedings of the Second SUIF Compiler Workshop*, 1997.

[17] 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.

[18] Alexandre E. Eichenberger, Kathryn M. O'Brien, Kevin O'Brien, Peng Wu, Tong Chen, Peter H. Oden, Daniel A. Prener, Janice C. Shepherd, Byoungro So, Zehra Sura, Amy Wang, Tao Zhang, Peng Zhao, and Michael Gschwind. Optimizing compiler for the CELL processor. In *14th International Conference on Parallel Architecture and Compilation Techniques (PACT 2005), 17-21 September 2005, St. Louis, MO, USA*, pages 161–172. IEEE Computer Society, 2005.

[19] 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.

[20] Michael Hind. Dynamic compilation and adaptive optimization in virtual machines, tutorial presented at PLDI 2004. `http://www.cs.umd.edu/~pugh/pldi04/tutorials.html`.

[21] 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.

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

[23] 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.

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

[25] Philipp Lucas. *CGiS: High-Level Data-Parallel GPU Programming*. PhD thesis, Universität des Saarlandes, August 2007.

[26] 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.

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

[28] 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.

[29] 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.

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

[31] 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`.

[32] 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.

[33] 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.

[34] 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.

[35] Jaewook Shin, Jacqueline Chame, and Mary W. Hall. Compiler-controlled caching in superword register files for multimedia extension architectures. In *2002 International Conference on Parallel Architectures and Compilation Techniques (PACT 2002), 22-25 September 2002, Charlottesville, VA, USA*, pages 45–55. IEEE Computer Society, 2002.

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

[37] David Tarditi, Sidd Puri, and Jose Oglesby. Accelerator: using data parallelism to program GPUs for general-purpose uses. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 325–335, New York, NY, USA, 2006. ACM Press.

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

[39] 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.

[40] 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.

[41] Hans Zima and Barbara Chapman. *Supercompilers for parallel and vector computers*. ACM Press, New York, NY, USA, 1991.