Efficient Parallelism Detection for Heterogeneous Computing

A Thesis
Presented to
The Division of Mathematics and Natural Sciences
Reed College

In Partial Fulfillment
of the Requirements for the Degree
Bachelor of Arts

Benjamin Black

May 2018

Table of Contents

List of Tables

List of Figures

Abstract

We are in the era of heterogeneous computing, where applications need to use more than one type of processor in order to improve performance. But programmers are not always taught how to best use these new processors, which results in poor use of hardware, lost performance, and lost power savings. One challenge for the programmer is that these new processors often rely on software to define parallel execution of code; unfortunately, programmers often have limited experience dealing with the problems associated with writing parallel applications. To help the programmer, we suggest building a tool that combines results from dynamic parallelism detection and workload characterization. For parallelism detection, we built a tool which detects whether parallelism exists in loops, and if not, determines why it does not. We confirm prior work that the naive implementation of the tool is memory intensive, so we implement two different methods of reducing memory consumption, one based on stride compression as introduced by SD^3. We find that combined with loop iteration size and variance data, this tool can be effective in guiding the programmer's parallelization efforts. We gather workload characterization metrics that could be combined with the parallelism information to make the tool useful in distinguishing between code that is effective on GPUs vs. CPUs, but we did not make use of these metrics due to time limitations.

Introduction

Traditional general purpose processors are designed to be easy to program while being as fast as possible at many different computation tasks. This allows the vast majority of applications to use general purpose processors exclusively.

For a long time, computer hardware tried to make the standard model of computation as fast as possible, inventing various methods such as instruction level parallelism that allowed the programmer to think of code sequentially while in actuality the code was running in parallel. But these methods could not scale well and eventually, the limits of these methods were found. To keep software performance improving, hardware moved to a many-core model where several cores executed code independently. Programming languages started including multi-threading features, and software developers were forced to learn how to reason about parallel programming.

But because each core of a many-core processor needs to execute general purpose code efficiently, each core is large, complex, and power hungry, so relatively few cores can be placed on a single chip. Today, high end many-core processors typically have at most two dozen cores.

This limitation was caused by the cores being general purpose. Hardware, however, does not need to be general purpose. Individual applications often have a specific type of procedure that consumes the vast majority of the computing power. For these applications, hardware developers can build an accelerator that runs that procedure very efficiently but cannot run other types of code. The general purpose processor running the application can use the accelerator to compute that procedure quickly and then perform the rest of the computation itself. Encryption and decryption are commonly done with accelerators.

Some hardware takes a compromise between generality and performance. Notably, Graphics Processing Units (GPUs), originally designed to accelerate graphics in video games, have become general enough to run many types of computations and now run many important numeric scientific calculations.

This concept of using different hardware processes to run a single application process is called heterogeneous computing. The computer hardware industry is starting to put several different types of processors and accelerators on a single chip, so heterogeneous computing is gaining increasing importance.

0.1 Challenges of heterogeneous computing

Heterogeneous computing offers many challenges for the programmer. Since the different hardware platforms have different characteristics and interfaces, different programming languages, libraries, and tools need to be used when coding for each platform. The languages and tools for accelerators are generally much less user friendly than ones for general purpose processors, and programmers are less familiar with them. This means that programmers often have to put substantial effort into porting their code to an accelerator like a GPU.

Even worse, there is no guarantee that an application will run faster on an accelerator. Take a GPU as an example. Because of the overhead involved in starting up work on a GPU, GPUs execute sequential code slower than CPUs. Even if the code is massively parallel, GPUs may still be unable to execute it effectively because the application depends on certain hardware characteristics of general purpose processors that are different then on GPUs. So when a programmer tries to parallelize an application and does not have a deep understanding of GPU architecture, they are taking a certain risk: all that extra work may be for nothing.

Since the benefits of heterogeneous computing are potentially so large, reducing the risks and making heterogeneous programming more accessible to non-experts could bring significant benefits.

0.2 Our solution

This thesis focuses on a small part of the larger challenge of heterogeneous computing. First, we focus our attention only on CPUs and GPUs and ignore other types of hardware. This is only because we are restricted by time, we hope that our work would easily extend to other highly programmable accelerators.

Within this focus on CPUs and GPUs, this thesis focuses only on the identification problem:

1. Determining whether certain code structures are parallel or not.

2. If so, then determining which kind of hardware would be more appropriate: CPU or GPU.

We suggest creating a tool which can help programmers understand how their code might perform on a many-core processor and a GPU assuming the code was ported to a GPU. Ideally, this tool would not only give results based on how the code currently is, but also give information to the programmer which would help them understand how the code could be changed to give better results on different types of hardware.

The results of this tool should allow average programmers to make better decisions and assume less risk when deciding how to allocate their time, potentially allowing them to better accelerate their applications.

0.3 Related work

There are two avenues of prior work which we try to combine in order to make such a tool: workload characterization and dynamic parallelism detection.

Workload characterization measures the demands applications place on hardware resources. Che et. al. created the Rodinia benchmark, to allow researchers to do workload characterization to understand differences between GPUs and CPUs.[1] In particular, the goal for this benchmark suite is to understand the general types of workloads and data structures that work well on GPUs. In a way, they are examining the characteristics of categories of applications, where we are trying to investigate characteristics of specific applications. Workload characterization and how it relates to our goals is described in more detail in Section 2.1.

Parallelism detection analyzes which parts of sequential code might be run in parallel. Historically, applications of parallelism detection have been limited to CPUs. An example commercial application of parallelism detection is the Intel® Advisor. The tool tells programmers which loops in the code might be parallelized, how they might be parallelized, and if not, where the problems are that are stopping it from being parallelized. Intel Advisor currently costs around $1600 and is closed source, so we will be building our own tool similar to this one but with fewer features.

0.4 Parts of thesis

This thesis makes several contributions to the larger goal of helping the programmer understand the presence and type of parallelism in their code:

1. Built a memory-efficient tool to detect parallelism in loops.

2. Built a tool to collect a variety of metrics for our workload characterization.

3. Analyzed the possible effectiveness of these tools to distinguish between GPU and CPU oriented applications.

Chapter 1 introduces parallelism detection and approaches to improve memory efficiency in parallelism detection. Chapter 2 describes implementation details of the tools, and Chapter 3 presents some results on how well these tools ended up working.

[1]Che et al. (2010)

Chapter 1

Dynamic parallelism detection

1.1 Introduction

We need a way of determining parallelism in order to do any useful inference on hardware compatibility. However, recall that our end goals are larger than just that. We are creating a tool that helps the programmer understand their code and helps them improve the algorithms and prototype them to perform better on different hardware without having the programmer actually deal with the hardware. Consequently, we not only want a parallelism tool that figures out which parts of the code can be parallelized or not, but we also want a tool that helps the programmer understand why code might not be parallelizable and perhaps suggest how to fix it.

We came to the conclusion that current open-source parallelism detection tools do not meet our efficiency and accuracy requirements, and so we built our own tool which does meet our needs.

This chapter goes over the necessary background for that tool. In Section 1.2, we introduce the type of parallelism we are detecting, and why it is so important. In Section 1.3, we introduce dynamic analysis of loops, a prerequisite for the base algorithm. In Section 1.4, we introduce the base algorithm for detecting it, and the problems with that base algorithm. In Section 1.5, we go over prior work in this area and discuss how it does not suit our needs completely.

1.2 Loop level parallelism

Parallelism exists in different code structures and at different levels of granularity. Programmers talk about how parallelism manifests itself in code. These include task parallelism (executing different functions in parallel) and loop parallelism (running different parts of a loop in parallel). Computer architects talk about how different kinds of parallelism can be exploited in hardware, including instruction level parallelism, thread level parallelism, and memory level parallelism.

This thesis focuses on loop-level parallelism (LLP). This type of parallelism is common, easy to detect, scalable on massively parallel processors, and simple for programmers to reason about.

1.2.1 What is loop level parallelism?

We start by introducing some terminology. A loop is a code construct that executes the same instructions over and over on different data. A loop iteration is a single execution of the code in the loop. Two loop iterations are independent if the later iteration does not use data computed by the prior iteration.

Loop level parallelism exists when every iteration of the loop can be run independently from every other iteration.

1.2.2 Examples of loop level parallelism

Here is code that performs in-place vector-scalar multiplication on a vector `B` with `n` elements.

Code Example 1 Vector-scalar multiply

```
for(int i = 0; i < n; i++){
    B[i] = B[i] * c;
}
```

This loop is parallelizable. In this loop, each iteration is simply the calculation of `B[i] = B[i] * c` for some `i`. Since the iteration is dependent only on the element of `B` that it also updates, any of these iterations can be executed before any other iteration. You can compute these iterations in any order without affecting correctness, so the loop is parallelizable.

In contrast, here is code that performs a cumulative sum.

Code Example 2 Cumulative Sum

```
for(int i = 1; i < n; i++){
    B[i] = B[i-1] + A[i];
}
```

This loop is not parallelizable. For `i > 1`, `B[i] = B[i-1] + A[i]` cannot be computed correctly until `B[i-1]` is calculated, since `B[i-1]` is used to calculate `B[i]`. To summarize, the loop must be executed in a particular order, so it is not parallelizable.

1.2.3 Importance of loop level parallelism

Loop level parallelism is important to this work for four reasons: it is common, scales well on massively parallel hardware, is simple for programmers to reason about, and is easy to detect.

Loop level parallelism is common and easy for programmers to reason about. A large amount of work in research and tool development has been devoted to exploiting loop level parallelism. Many parallelism frameworks are focused on loop level parallelism. OpenMP is a C and C++ parallelism framework that allows programmers to parallelize loops (assuming they are parallelizable) on CPUs with a single line of code.

Loop level parallelism is easy to detect compared to other types of parallelism. Restricting ourselves to this kind of parallelism will give us a simple and fast algorithm. Research in automatic parallelization has been focused mostly on loops, mostly because loops are more common and easier to analyze than other parallel code structures.

Finally, parallel loops often scale well on massively parallel hardware compared to parallel tasks or recursive calls, simply because they often have a large number of iterations.

1.3 Dynamic Analysis

There are two general methods to establish parallelism in code: static analysis and dynamic analysis. Static analysis tools parse code like compilers do, analyzing syntax trees of variable assignments and loops. They attempt to establish general properties of the code. Generally, the results of static analysis are true given any possible input. Unfortunately, as code increases in complexity and generality, it becomes impossible to derive precise results in full generality. For a complex semantic property like parallelism, static analysis returns too many false negatives to be useful for practical purposes.

Dynamic analysis tools examine running processes like an interpreter does, executing a series of instructions on a particular input, while analyzing the behavior and values of the running program. Since a dynamic analysis tool only observes the program on a particular input, it can only establish properties of the code for that specific input. Therefore, dynamic analysis can give false positives for parallelism, meaning a loop might be parallelizable on the particular input, but not on others. To reduce the number of false positives, a complex input that uses many of the features of the program and reflects production use will be more helpful than a simple, contrived input.

1.3.1 Dynamic Loop Detection

Dynamic loop level parallelism detection requires a determination of when a loop starts, ends, and iterates in a running program. This is a surprisingly challenging problem. Loops have a clean structure in high level languages like C. It is easy for both the compiler and a human to understand what code is inside the body of a loop, and when the loop starts and ends. However, this work focuses on the dynamic analysis of loops.

Doing dynamic analysis requires us to understand the structure of loops in machine code. Even if the source code is available, there is not always a precise mapping from machine code to source code, as loops may be compiled to different machine code depending on the compiler. Loops in machine code are hard to detect, as they are implemented with “go to” instructions (referred to in this thesis as goto). goto instructions allow the code to jump to fixed instructions elsewhere in the code. These instructions implement various functionality like if-else blocks, while loops, and break statements, so their presence does not guarantee a loop.

To demonstrate the difficulty of finding loops in machine code, here is the standard way of translating the above vector-scalar multiply loop into machine code:

```
1 i = 1
2 if i >= n goto line 8
3 tmp1 = B[i]
4 tmp1 = tmp1 * c
5 B[i] = tmp1
6 i = i + 1
7 goto line 2
8 ...
```

Note that even with a single simple loop, there are two goto statements, one for creating the loop and one for exiting the loop. With nested loops, branches, and break statements, it quickly becomes very difficult to understand where a particular loop starts and how to know that a particular loop terminated. So in order to understand loops in machine code, we have to somehow separate loops out from a tangle of goto instructions.

Luckily, there is prior work for doing just this. We use LoopProf [1] code in order to detect the locations of loops in code. In particular, LoopProf tells us where the loop begins and the nesting structure of loops.

LoopProf uses dynamic analysis in order to determine where loops are. At a high level, it tracks a list of instructions executed in a function. When a goto jumps back to a instruction on that list, then LoopProf concludes that there was a loop, and that its second iteration just started executing.

However, the LoopProf implementation, generously provided to us by its author, did not contain all of the functionality we needed. In particular, it did not detect the precise ends of loops, which we needed for dynamic dependence analysis. Consequently, we made some additions in order to suit our needs.

1.4 The baseline algorithm: The pairwise method

One of the goals of this thesis is to create a tool that identifies loop level parallelism. The literature has several methods to identify this type of parallelism. A particularly simple method is the naive pairwise method. This thesis's implementation will be an optimization of the pairwise method, so the naive version will be described in detail here.

Recall that loop-level parallelism occurs when each iteration of a loop can be executed independently from every other iteration, and that independence means that one iteration does not depend on data computed by another iteration. The pairwise method tracks reads and writes to memory to determine dependences. Each read is considered to be an input to the computation in the loop, and each write is an

[1] Chen et al. (2004)

output. If one loop iteration writes to one location in memory and a later iteration reads from that same location, then that is a dependence that should be recorded as it inhibits parallelism.

There is an important difference between memory and dependencies. Memory can sometimes be reused without creating dependencies. In particular, consider a location in memory that is written to before it is read in a loop iteration. The data that was previously at that location was overwritten, so the current iteration will not depend on data at that location that was written in previous iterations. In this case, we say the location is "killed." Reads of killed memory locations are not tracked in a given iteration.

The pairwise method stores two different structures for a particular loop:

- history table: A table of reads and writes of previous loop iterations.
- pending table: A table of reads and writes of the current loop iteration.

The pairwise method determines dependencies only using this data. The next section illustrates the method with examples.

1.4.1 Simple demonstration

Here is the cumulative sum procedure from the loop level parallelism example section, written to show only the reads and writes to memory.

```
1  for(int i = 1; i < n; i++){
2      tmp1 = B[i-1]          //READ B[i-1]
3      tmp2 = A[i]            //READ A[i]
4      B[i] = tmp1 + tmp2     //WRITE B[i]
5  }
```

When the loop is executed, the code reads and writes to the arrays. This read and write information is combined with information about how the loop iterates in our analysis. For example, here is how the above code would be processed when executed.

```
LOOP START
LINE 2 READ B[0]
LINE 3 READ  A[1]
LINE 4 WRITE B[1]
LOOP ITERATION
LINE 2 READ B[1]
LINE 3 READ  A[2]
LINE 4 WRITE B[2]
LOOP ITERATION
...
LOOP END
```

This combined information is the input to the pairwise method. Note that in the implementation, lines will be notated by their instruction addresses, not their line numbers.

The actual process of the pairwise method can be seen in the tables below. When the loop starts, the pending table and history table are empty. As the first iteration is processed, reads and writes are put into the pending table. Right before the first iteration ends, the pending and history tables look like this:

Pending Table		
Instr	Memory	R/W
2	B[0]	R
3	A[1]	R
4	B[1]	W

History Table		
Instr	Memory	R/W

The history table is still empty, as there were no previous iterations of the loop. But as the first iteration ends, the pending table is merged into the history table.

Pending Table		
Instr	Memory	R/W

History Table		
Instr	Memory	R/W
2	B[0]	R
3	A[1]	R
4	B[1]	W

Right before the second iteration ends, the tables look like this:

Pending Table		
Instr	Memory	R/W
2	B[1]	R
3	A[2]	R
4	B[2]	W

History Table		
Instr	Memory	R/W
2	B[0]	R
3	A[1]	R
4	B[1]	W

At the end of the second iteration, the pairwise method will look through the reads of the pending table and see if there are any memory addresses in the history table that are writes. In this case, the history table has a write at address B[1] by line 4, and the pending table has a read at B[1] by line 2. A dependence between line 4 and 2 will be logged, and the pairwise method correctly identifies that the loop is not parallel.

1.4.2 Nested loop demonstration

This method also works for nested loops. We will be using the vector-matrix multiplication procedure presented in Code Example 3 to demonstrate this. In this procedure, the inner loop is not parallel but the outer loop is. The pairwise method will be able to detect this. This section will walk through the execution of the pairwise method on that procedure. The locations of read and write instructions will be referenced using line numbers from Code Example 4.

Code Example 3 Matrix-vector multiply

```
for(int i = 0; i < output_size; i++){
    sum = 0
    for(int j = 0; j < input_size; j++){
        sum += M[i][j] * x[j]
    }
    y[i] = sum
}
```

Code Example 4 Matrix-vector multiply with expanded memory accesses

```
 1 for(int i = 0; i < output_size; i++){        //outer loop
 2     sum = 0                      //WRITE sum
 3     for(int j = 0; j < input_size; j++){     // inner loop
 4         tmp1 = x[j]              //READ  x[j]
 5         tmp2 = M[i][j]           //READ  M[i][j]
 6         tmp3 = sum               //READ  sum
 7         tmp4 = tmp1 * tmp2
 8         tmp5 = tmp3 + tmp4
 9         sum = tmp5               //WRITE sum
10     }
11     tmp6 = sum                   //READ sum
12     y[i] = tmp6                  //WRITE y[i]
13 }
```

As the outer loop starts, line 2 writes to the memory address of `sum`. This is put into the pending table of the outer loop. Then the inner loop starts. Empty pending and history tables are created for the inner loop; these tables are separate from the outer loop's tables.

The inner loop is processed very similarly to the cumulative sum example from earlier. A loop dependence on `sum` between lines 6 and 9 is logged for the inner loop. At the end of the inner loop, the history table will contain:

History Table (inner loop)

Instr	Memory	R/W
4	`x[0],...x[input-size-1]`	R
5	`M[0][0],...,M[0][input-size-1]`	R
6	`sum`	R
9	`sum`	W

When the inner loop finishes, its history table is merged into the pending table of the outer loop, meaning all of the reads of arrays `x` and `M[i]` are put in the pending table of the outer loop. In that pending table, `sum` is written to (line 2) but not read; since the first access is a write, in the outer loop, `sum` is considered killed, and no further reads or writes to `sum` will be put in the pending table, including the accesses from in the inner loop, or the access on line 11. However, the first write to `sum` on line 2 is still an important access that could cause a dependency in an outer loop, so it still needs to be stored.

At the end of the first iteration of the outer loop, the pending table will be:

Pending Table (outer loop)

Instr	Memory	R/W
2	`sum`	W
4	`x[0],...x[input-size-1]`	R
5	`M[0][0],...,M[0][input-size-1]`	R
12	`y[0]`	W

This table will be merged into the history table of the outer loop, and the process will repeat. Note that there will not be any read after write dependences logged, since `sum` and `y[i]` are only written to and `x[j]` and `M[i][j]` are only read from. Based on this analysis, the pairwise method concludes that the outer loop is parallel. Intuitively this is true; the outer loop can be parallelized trivially if the memory of `sum` is duplicated for each thread.

1.4.3 Generalized algorithm

To generalize this approach to arbitrary nested loops, the pairwise method keeps track of a global loop stack, a stack of pending and history tables for all of the active loops. With this, we can describe a more precise algorithm for the pairwise method: Algorithm 5. The significant procedures in this outline will be referenced later as specific implementations are described.

Algorithm 5 Pairwise method

```
1. When loop $L$ starts, initialize empty pending and history tables for $L$, and push
   $L$ onto the loop stack.
2. When a memory location $m$ is accessed by instruction $i$, and loop $L$ is at the
   top of the loop stack,
   (a) If location $m$ is killed in the pending table, do nothing. SUBTRACT-
       KILLED
   (b) Otherwise, put that information into the pending table of loop $L$. ADD-
       ACCESS
3. When an iteration of loop $L$ ends,
   (a) Find conflicting memory locations between reads in the pending table and
       writes in the history table. Report those conflicts. CONFLICT-CHECK
   (b) Merge the pending table into the history table. MERGE
4. When loop $L$ terminates,
   (a) If there is a loop $L'$ below $L$ on the loop stack,
       i. Remove entries in the history table of $L$ that are at killed memory
          locations of $L'$'s memory location. SUBTRACT-KILLED
       ii. Merge the history table of loop $L$ into pending table of loop $L'$ MERGE
   (b) Pop $L$ off the loop stack.
```

1.4.4 Naive implementation

An implementation of the pairwise method simply needs to define the data structures of the pending and history table and make the capitalized procedures more precise.

The naive implementation of the pairwise method uses the following data structures and procedures.

The pending and history tables are implemented as a hash table indexed by memory addresses and which stores sets of instructions. To simplify the procedure, tables for writes and reads are separate, so there are two pending tables, one for reads and one for writes. The table procedures outlined in the pairwise algorithm description are implemented as follows:

- ADD-ACCESS: Use the read pending table if the access is a read, otherwise, use the write table. If there is no entry for the memory address in that table, add an entry, and put the instruction from that new access in the set. If there is an entry, add the accessed instruction to the set.
- MERGE: Repeatedly add every access from the input table to the output table using ADD-ACCESS.
- SUBTRACT-KILLED: For a particular access in the history table, check if there is an entry at that location in the write pending table but not the read pending table. If there is, remove the entry.
- CONFLICT-CHECK: Go through every entry in the pending table, and check if it is a read, and if there is also an entry in the history table at that memory address that is a write. If so, then there is a conflict at that location between all the instructions in the two entries in the table.

1.5 Prior work on optimizing the pairwise method

Naive implementations of the pairwise method have a severe memory overhead that limits the usefulness of the approach. Kim et al.'s work mitigates this problem by reducing memory consumption through compression[2]. Our work follows the mechanism used in the SD^3 paper to reduce memory overhead. This section gives an overview of the memory considerations involved with the pairwise method and prior attempts to fix this by compressing the working memory.

1.5.1 Memory overhead of pairwise method

The pairwise method often uses more working memory than ordinary systems have, since it has to store information for each memory access in the worst case. The SD^3 paper[3] mentions commercial tools which use roughly 100x the memory of the program being profiled. The authors tested their implementation of the naive method on 17 SPEC 2006 C/C++ benchmarks. They found that 14 applications used more than 12GB of memory, exceeding the limits of the test system. Out of the 14 benchmarks that used over 12GB of memory to run, 9 used under 100MB of memory under native

[2]Kim et al. (2010)
[3]Kim et al. (2010)

execution, and memory overhead was over 120x for all 9 of these applications. This suggests that ordinary users will not be able to profile important pieces of software on production inputs, limiting the usefulness of tools using the naive pairwise method.

1.5.2 Stride compression

The SD^3 paper introduces using strides to compress the memory accesses information in the pending and history tables. We present SD^3 approach here as the thesis uses a similar approach.

What is a stride?

A stride is a finite range of integers which have a constant interval between elements. It can be stored efficiently as a triple `(first, last, interval)`. For example, the even numbers from 2 to 20 would be a stride `(first: 2, last: 20, interval: 2)`.

Memory is often accessed in a strided fashion in code. For example, here is code that accesses an array of memory in a strided fashion. Every third element of the array is written to.

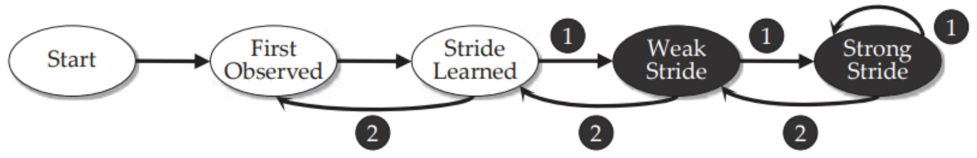
```
for(int i = 1; i < 12; i = i + 3){
    B[i] = 1;
}
```

One way to store all these memory accesses would be to store a list of memory addresses for each index `[1, 4, 7, 10]`. But information can be stored more efficiently as a stride: `(first:1, last:10, interval:3)`. This is more efficient because no matter how long the stride is, it can always be stored in just 3 values, instead of needing more memory to store each new access.

Detecting strides

The SD^3 method detects both the existence and interval of strided accesses. Some access patterns are not strided, and if so, the memory accesses should not be treated as strides. When accesses are strided, then the interval of the stride must be calculated dynamically.

Figure 1.1: Stride detector state machine[4]



SD^3 suggests using an online stride detection algorithm to calculate both the existence and length of strides. This algorithm keeps a global data structure separate

[4]Kim et al. (2010)

from the loop stack which keeps track of separate state machines for every instruction. The state machine is organized as follows:

This state machine keeps track of the stride length and position of previous accesses, inferring the stride length of the current access. The strong stride-weak stride states allow for short interruptions in an otherwise consistent stride pattern. This may be helpful, for example, when accessing the rows of a two dimensional array. In this case, the access pattern may be unpredictable when switching between rows, but the overall stride pattern will remain the same within each row.

Data structure for non-strided accesses

Some memory access patterns are not strides. When this is the case, the SD^3 method calls these accesses "points", and puts them in a "point table", which is similar in structure to the pending and history tables in the naive method. These are kept separate from the strides, which are put in the "stride table". There is both a stride and a point table inside each history and each pending table.

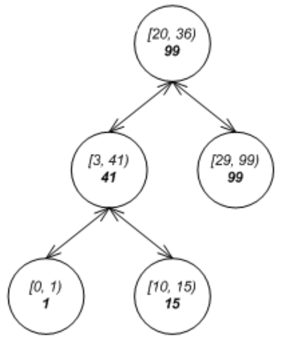
Data structure for strided accesses

Operations on non-strided accesses are efficient because the memory location of an access can be looked up quickly using hash tables. However, strides can conflict and be merged even when they don't start or end on the same address, so different data structures are needed. One possible method would just use a list of strides, but the pairwise comparisons needed to do conflict checking would take quadratic time. The SD^3 paper suggests use of interval trees to quickly prune the search space to mitigate this quadratic complexity.

An interval tree is an augmented binary search tree that allows searches for intervals, instead of single values. An interval tree can efficiently find all intervals stored inside it which overlap with another interval.

Take the tree in Figure 1.2, which stores 5 intervals, including `[20,30)`, `[10,15)`, and `[0,1)`.

Figure 1.2: Interval Tree



This tree can be queried to find all intervals that overlap with another one, say

`[40,50)` (this will return `[3,41)`, and `[29,99)`). Like an ordinary binary search tree, it is self-balancing and allows $O(\log(n))$ inserts and deletes. Finds are $O(m \log(n))$, where m is the number of intervals it finds.

In the context of strides, each stride would simply be stored as an element in the tree keyed as an interval from first to last, inclusive. Then, operations like conflict checking can find strides that overlap with efficiency relating to the number of strides found, rather than the size of the set of strides.

Merging strides

In order for the overall algorithm to use less memory than the naive version, individual strides must be merged together when merging together sets of strides in MERGE. This is because the online detection method generates strides for single accesses, which are immediately put in the pending table. If collections of strides are joined naively during MERGE, then these single-access strides will be the only type of stride in the table, and memory performance will not improve over the naive version.

Two strides are mergeable when they can be written as a single stride. For example `(first:1, last:10, interval:3)` and `(first:13, last:22, interval:3)` can be merged, since the stride `(first:1, last:22, interval:3)` covers the same elements as the two of them combined. On the other hand, `(first:2, last:8, interval:2)` and `(first:13, last:22, interval:3)` cannot be merged since simply extending the bounds of the stride cannot include all of the elements of both strides.

SD^3 uses the interval tree described above to find possible candidates for mergable strides, and then checks further. If strides for a particular instruction are consistently not found to be mergable, then no further attempts are made to merge them.

Finding overlap between sets of strides

During CONFLICT-CHECK, there are 3 different types of possible access conflicts: point-point, point-stride, and stride-stride. Point-point conflicts are found exactly as in the naive version, with hash table lookups. Possible point-stride and stride-stride conflicts are found using interval trees. Checking point-stride conflicts efficiently is simple, but checking stride-stride conflicts in constant time is more complex.

SD^3 introduces a new algorithm called Dynamic-GCD which efficiently finds overlap between strides. To understand how constant time stride overlap checking is possible, consider infinite length strides. Take two infinite length strides, with stride length a and b. Take two points, one from each stride, and calculate the distance between them. Call this distance c. Then there is an integer x and y such that $ax + by = c$ if and only if the strides conflict. This condition is in turn equivalent to $c = 0 \mod (\gcd(a, b))$, where $\gcd(a, b)$ is the greatest common denominator of a and b. This test can be done in constant time with the well known Euclidean algorithm for the GCD. Now, finite strides sometimes do not intersect when their infinite counterparts do, so the Dynamic-GCD algorithm gives a similar, but more complex test than the above that uses the Extended Euclidean algorithm to exactly compute intersection for finite strides.

1.6 Bit compression

A strided format is not the only method of compressing memory accesses. Memory accesses can also be compressed effectively using bit arrays. A bit array is simply an array of boolean variables which are stored inside individual bits, instead of in bytes, like ordinary C variables.

1.6.1 Basis of bit compression

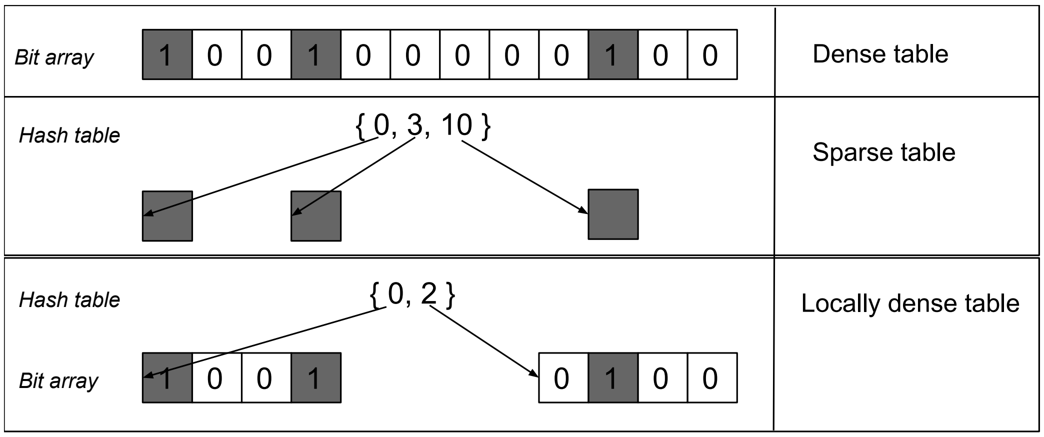
Stride compression is effective because many memory intensive procedures in real applications access memory in a strided fashion. Bit compression is effective because program memory is locally dense, meaning if memory is accessed in one location, nearby memory addresses are usually also valid program memory which will be accessed eventually. This local density property is virtually guaranteed by modern operating systems and hardware, and so it is not as dependent on the program's structure as stride compression.

1.6.2 Implementation of a bit compressed integer set

A locally dense set implementation should use the advantages of dense and sparse set implementations. A sparse set of integers is best implemented using a hash table. A completely dense set of integers is best implemented using a bit array. So a locally dense set should be implemented by a hash table where each entry is a bit array that represents a contiguous range of integers starting at a particular location. Figure 1.3 shows the the three different types of sets storing the set $\{0, 3, 10\}$.

The items in the hash table in the locally dense set are key value pairs. The key in a key value pair is a unique identifier to a location in the table. The value is other information associated with that key. In this case, the key is number/BlockSize, and the value is the bit array with length BlockSize.

Figure 1.3: Representations of the set $\{0, 3, 10\}$



Chapter 2

Implementation

This chapter will first discuss workload characterization concepts, introducing the relevant hardware concepts, and then discussing our method of collecting specific characteristics in formats which can be matched with parallelism data.

To detect parallelism and also to compare the compression methods outlined in Section 1.5, we implemented three different versions of the pairwise method: one based on stride compression, one based on bit compression, and one based on the naive implementation. This chapter will describe the stride compression and bit compression algorithms in detail.

2.1 Introduction to workload characterization

Workload characterization measures the demands application place on hardware. Hardware researchers are interested in the sorts of hardware characteristics that are important to application performance. They use workload characterization to understand how trade-offs in hardware design would affect various applications' performance. To accomplish this, they collect metrics on the applications they want to optimize. These metrics inform decisions about the hardware trade-offs that hardware engineers are considering.

We are interested in a related question: What are the application characteristics that can help the programmer choose between different established hardware platforms? We use workload characterization to understand the platform independent characteristics of single threaded CPU programs that are indicators of performance for traditional multiprocessors and GPUs.

In this work, we examine a variety of applications that are best fit for CPUs and GPUs and collect platform independent metrics that help indicate which type of hardware that parts of the application are best fit for.

2.1.1 Important hardware characteristics under consideration

In the following sections we explain some important hardware characteristics in traditional multiprocessors and GPUs and the metrics that measure some aspect of these characteristics.

Scale of Parallelism

GPUs will not be able to execute procedures faster than CPUs unless that procedure has a high degree of parallelism. GPUs have hundreds of small cores, where CPUs have at most two dozen or so larger cores. So CPUs are better at processing a small number of tasks; GPUs will not be effective unless the work can be split over hundreds of threads. To capture this, we suggest using the number of iterations in parallel loops.

In addition, using a GPU has significant overhead, meaning it takes substantial time to run any computation on a GPU, no matter how trivial. This is because when a GPU is used, the CPU typically has to send data and code to the GPU, the GPU runs the computation, and then sends back the data. On a traditional multiprocessor, the overhead of using threads is much smaller.

Since each kernel call must do substantial work to be effective on a GPU, we suggest collecting the number of instructions each loop iteration contains to make sure there is substantial work to be done by the loop.

Parallelism Model

At a low level, GPUs execute threads in groups called warps. Warps have an SIMT (single instruction multiple thread) model of execution, meaning that each thread in the warp executes instructions simultaneously in lock-step. Due to this hardware design, when GPU threads in a warp execute different instructions due to a branch, they cannot do so simultaneously, so instead the different instructions are executed serially.[1] This sequential processing of branch paths creates a performance hazard called branch divergence. Handling branch divergence is time consuming on GPUs, so performance suffers significantly when branch paths differ within a warp.

Meanwhile CPUs have a MIMD (multiple instruction multiple data) model, where each thread executes code independently. CPUs are also equipped with advanced branch prediction which often reduces the cost of branches to nearly zero. Consequently, parallel code which executes different branch segments may run better on CPUs.

To account for this performance difference, we collect a count of the number of conditional branches encountered, and the number of times the path of that branch changed. These branch changes captures the probability that different threads in a GPU warp will take different branch paths.

[1] Fung et al. (2007)

Predictability of memory accesses

GPUs have global memory that is optimized for high bandwidth and high latency. Predictable accesses such as strided memory can be streamed from GPU memory to the cores efficiently, while unpredictable memory accesses stall the instruction pipeline because they take longer to access. In contrast, CPUs have larger caches that accommodate data reuse and so are much better at handling unpredictable memory if that memory fits inside the caches.

We chose to capture this performance difference by measuring the number and length of strided accesses. We use the same stride detector code as in our pairwise method (see Section 1.5.2.2 for a design).

Inter-thread vs Intra-thread memory sharing

CPU theads have much larger cache capacity than GPU threads, since there are simply many more GPU threads active at a given time.[2] This means that GPU threads cannot rely on caches to exploit temporal or spacial locality to the same degree that CPU threads can. Instead, GPU caches effectively exploit cross-thread locality.

To capture this, we measure the amount of memory shared between loop iterations vs reaccessed within a single iteration. We construct maps of the memory footprint of the loop and each of its iterations. From these memory footprints, we calculate the number of unique memory bytes accessed in each iteration and each loop instance.

2.1.2 Loop-level Metrics

An important contribution of this thesis is creating tools which allow parallelizability information to be combined with other metrics in our workload characterization. In particular, since we evaluate loop level parallelizability, we chose to collect all other metrics at the loop level.

We chose some metrics based on the main differences between CPUs and GPUs as described previously in this section, as well as some other metrics which are common in workload characterization. Table 2.1 summarizes these metrics.

2.2 Metrics Collection

For a tool to detect loop level parallelism and collect loop level metrics, it first needs to understand where the loops are in the code. To do this, we run LoopProf (discussed in Section 1.3.1) on the application to collect information about which instructions loops start and end on.

To detect parallelism, we created a tool powered by Intel Pin[3] which uses the loop information generated by LoopProf. This tool calls methods from the parallelism detection algorithm when loops start and end and when memory is accessed. When the

[2]Jia et al. (2014)
[3]Luk et al. (2005)

Table 2.1: Summary of collected data

Metric	Relevant hardware characteristic(s)
Number of iterations	Scale of Parallelism
Instruction count of loop iterations	Scale of Parallelism
Number of writes/reads	Traditional workload characterization
Bytes accessed per instance	Traditional workload characterization
Total memory footprint of loop	Memory sharing
Shared memory footprint between loop iterations	Memory sharing
Strided accesses	Memory access predictability
Number of branch changes	Parallelism type

program finishes, the parallelism detection algorithm outputs its results of how many loop iterations conflicted and which pairs of instructions those conflicts originated from.

To collect loop level metrics we created another analysis tool powered by Intel Pin[4] which uses the loop information generated by LoopProf. Similar to the pairwise method, this tool collects loop level information by keeping a stack of data about each current active loop, and when the loop terminates, it does two things. First, it accumulates the information about the current loop instance into a summary of all the instances in the static loop. Then it simply passes its information to the loop below it on the stack, which accumulates that information with its own information.

2.2.1 Stack and register memory

One problem with our approach to parallelism detection is that it has no way of detecting inductions. An example of an induction variable is the `i` in Code Example 6. According to the method of analysis demonstrated in Section 1.4.2, the variable `i` carries a loop dependency because it is read in line 3 and then written in line 3. But this loop can be transformed into a parallel loop easily. This is a serious problem, since `i` is just a loop index that nearly every sequential loop has, including the ones that can be run in parallel.

Real world parallelism detectors use static analysis to detect these induction variables.[5] Static analysis is effective in this case because loop indices are almost always stored in a fixed location in the stack or in registers, so only fixed locations of memory need to be checked for the properties that induction variables have.

Unfortunately, we did not have time to do any static analysis. To deal with this issue, our Pin tool which records accesses simply ignores all stack and registers accesses. While this means we do detect parallel loops, it also means that some problematic memory conflicts are not detected.

[4]Luk et al. (2005)
[5]Kim et al. (2010)

Code Example 6 Loop Induction

```
int i = 0;
while(i < 1000){
    i = i + 1;
}
```

One example of a problematic dependency carried in registers is the `carry` bit in multiprecision addition. Say you have two arbitrary size integers, `A` and `B`. You add these together and store the results in `R`. These arbitrary size integers are stored as arrays where `A[i]` is the `ith` 64 bit block of `A`. Code Example 7 shows an algorithm to compute this addition. Note that the `carry` variable is a non-trivial dependence which could force this algorithm to be run sequentially.

Code Example 7 Arbitrary-precision Addition

```
carry = 0
for(int i = 0; i < max_size(A,B); i++){
    R[i] = A[i] + B[i] + carry;
    if(addition_overflowed(A[i] + B[i] + carry)) {
        carry = 1
    }
    else{
        carry = 0
    }
}
...
```

These register and stack dependencies seem to be rather rare in simple code, because finite quantities of memory rarely store crucial dependencies. But for more complex code, it seems more likely that such dependencies occur with some frequency. So when interpreting the results of this work, it is important to realize that our analysis assumes that stack and register variables do not cause problematic data conflicts.

2.3 Workloads

To distinguish between applications that are best run on CPUs or GPUs, we first needed a good sample of programs with different characteristics. Some of these should be geared towards CPUs, and others should be likely candidates for GPU acceleration. To get a wide variety of applications, we gathered applications from two different benchmarks: Rodinia and Parsec.

Rodinia

Rodina is a benchmark suite designed to compare CPU and GPU performance on different applications. The Rodinia applications are chosen to reflect the Berkeley

7 dwarves of scientific computation.[6] All of these applications are parallelizable to some degree, and most of them are effective on GPUs.

For each algorithm, Rodinia has a CPU implementation which is multi-threaded using OpenMP. Since our tool does not support multithreaded applications, we removed the OpenMP compilation flag, and removed calls to the OpenMP library such as `omp_get_thread_num()` from the source code.

Parsec

Parsec is a benchmark of multi-threaded applications designed for multiprocessors.[7] It has a more diverse range of applications than Rodinia, including computer vision and textual analysis as well as traditional scientific computing applications.

Parsec was more difficult to get running on our system. There were some applications which were incompatible with C++11 and did not compile, and many others which could not be altered to run single threaded without substantially changing the structure of the code, and so they could not be run on our analysis tools. We present results only for the applications that we could successfully compile and run on our tools.

2.4 Stride Compression Implementation

The code that Kim et al. wrote to implement the SD^3 algorithm was not made publicly available, so it is closed source, and we cannot use it to detect parallelism. To achieve the goals of this thesis, we still needed to detect parallelism without running out of system memory, so we chose to implement the algorithm ourselves. As we had limited time to implement it, we decided to make changes to the algorithm for simplicity. In particular, we thought implementing interval trees ourselves would be error-prone, and we could not find an open source C++ implementation, so we chose to use other data structures. In addition, the details of the Dynamic-GCD algorithm proved difficult to work out, so we used other techniques for finding stride conflicts. These were key features to ensure efficiency in the original SD^3 algorithm, so more changes had to be made to improve efficiency. In particular, a slow conflict-check method was used in our implementation, so optimizations were made that minimize the number of conflict-check operations that occur.

2.4.1 Algorithm and data structure overview

First, a high level overview of the main data structures and objects will be introduced, and later the specific data structures and algorithms will be detailed.

At the highest level, a program has a single LoopStack, which contains a stack of LoopInstances. LoopInstances store all information about a particular active loop,

[6]Che et al. (2010)
[7]Bienia et al. (2008)

including the pending and history tables. Separate from the LoopStack, there is a StrideDetector for each instruction that accesses memory.

Recall from Algorithm 5 that there are several main procedures in the pairwise method:

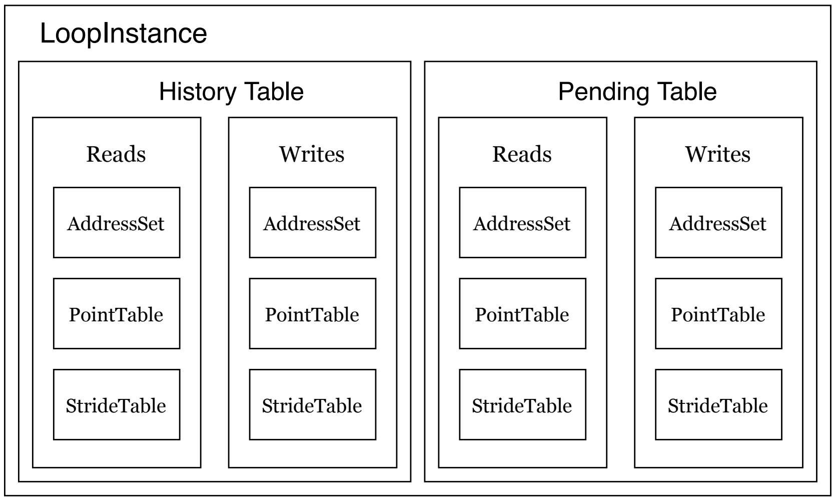
1. During a loop iteration, and when some memory is accessed, add the memory access and the instruction that accessed it to the pending table.
2. After a loop iteration finishes,
    (a) Check for conflicts between accesses in the pending and history tables. This is referred to as Conflict-Check.
    (b) Merge the pending table into the history table.
3. After an entire loop ends,
    (a) Merge all non-killed accesses in the history table into the pending table.

Our stride compression implementation has three main data structures inside each pending or history table. The purpose of these data structures is outlined here, and the details will be explained later.

- StrideTable: This table stores strided accesses and the corresponding addresses of the instructions from which the strided memory accesses originated. Strides can be added efficiently to this table.
- PointTable: This table stores non-strided accesses called points and the corresponding addresses of the instructions from which these memory accesses originate.
- AddressSet: This structure is simply a set of memory addresses. This is in contrast to Point and Stride tables which also store the instruction addresses associated with each memory access. We found it necessary to store this separate table which stores strictly less information to enable one of our optimizations.

These data structures store the content of the pending and history tables. Figure 2.1 shows how these tables fit inside the LoopInstance. Note that within a history or point table, there are two StrideTables, PointTables, and AddressSets, one table for reads, and one for writes.

Figure 2.1: LoopInstance Layout



2.4.2 Adding accesses

Algorithm 8 provides a high level description of how new memory accesses are put in the pending table. Figure 2.2 shows the steps as a flow chart. The numbers in the figure correspond to the steps in the algorithm.

Algorithm 8 New memory accesses

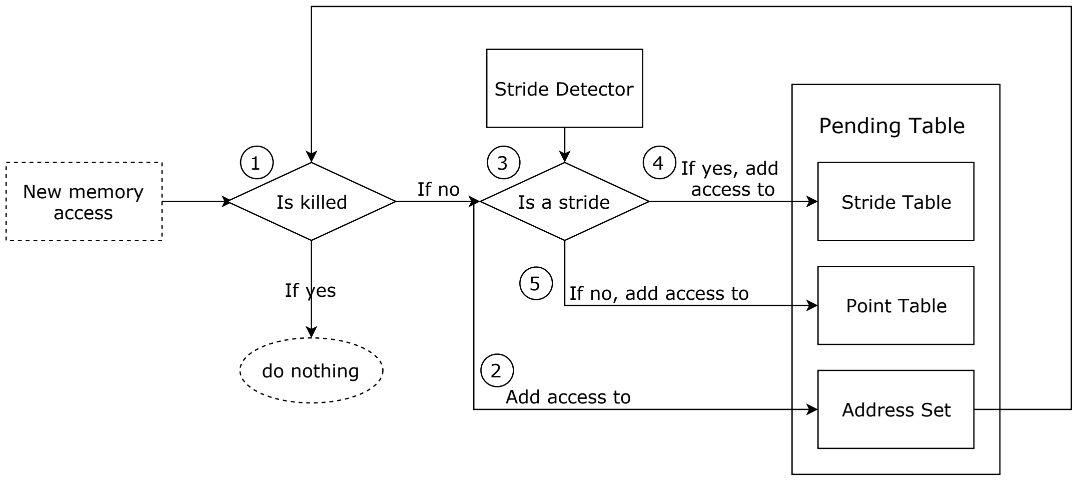
When a memory location m is accessed by instruction i, and LoopInstance L is at the top of the loop stack,

1. If the memory location m is killed according to the pending table's AddressSet, do nothing and skip the rest of the steps.
2. Add the bytes accessed by m to the appropriate AddressSet.
3. Find the loop-independent stride detector that corresponds to instruction i, and use it to calculate whether m is a stride. If it can be viewed as a stride, then look up the stride distance d which was evaluated by the stride detector.
4. If m is a stride with stride distance d, then add it to the appropriate stride table.
5. If m is not a stride, add it to the appropriate point table.

Stride Detection

The StrideDetector is implemented to match the original SD^3 design described in Section 1.5.2 as closely as possible. The state machine matches Figure 1.1.

Figure 2.2: Adding accesses

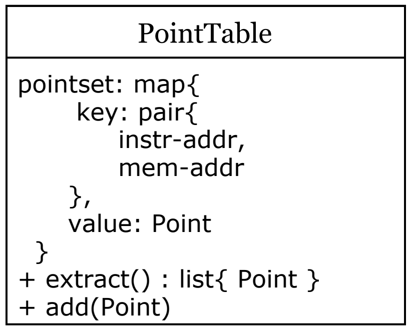


Adding points to the PointTable

Points (non-strided accesses) are added to the PointTable during new memory accesses and merges of two tables. The PointTable is designed so that identical points are not duplicated in the table.

The PointTable is simply a set (no duplicates) of points matched by instruction address and memory address (see figure 2.3). When being added to the PointTable, we check if the point already appears in the set. If there is a matching element there already, nothing happens; otherwise, the point is added to the set.

Figure 2.3: Point Table

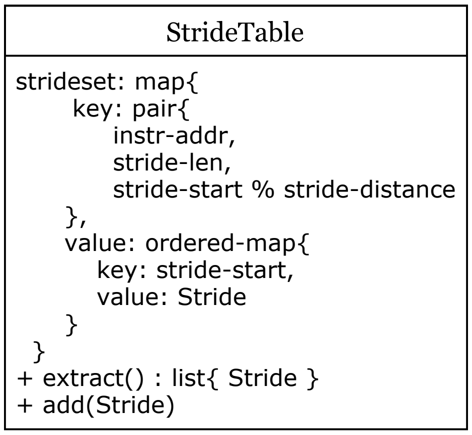


```
PointTable
pointset: map{
     key: pair{
         instr-addr,
         mem-addr
     },
     value: Point
  }
+ extract() : list{ Point }
+ add(Point)
```

Adding Strides to the StrideTable

Strides are also added to the StrideTable during new memory accesses and merges. When possible, strides need to be merged into existing strides in the table. This is critical for compression (see Section 1.5.2.5). The StrideTable is designed to make this process simple and efficient. At a high level, the StrideTable is a set of ordered sets of strides. Figure 2.4 shows a precise outline of how the table is structured.

Figure 2.4: Stride Table



```
StrideTable
strideset: map{
      key: pair{
           instr-addr,
           stride-len,
           stride-start % stride-distance
     },
     value: ordered-map{
         key: stride-start,
         value: Stride
     }
  }
+ extract() : list{ Stride }
+ add(Stride)
```

When storing an element in the StrideTable, the outer set is matched on the instruction address, the stride distance, and the stride start modulo the stride distance. Strides which have these criteria in common are mergeable if they adjoin. This is best explained with examples:

Stride #	Stride Distance	Stride start	Stride last	Merge issue with stride 1
1	2	4	8	Mergeable
2	2	0	2	Mergeable
3	3	4	13	Stride distance
4	2	5	9	Stride offset
5	2	14	20	Not adjoined

In the table above, Stride 1 is only mergeable with itself and with Stride 2. The other strides have the three types of issues which can prevent stride merging. Stride 1 is mergeable with Stride 2 because the strides have the same stride distance, the same offset (they are both on the same numbers modulo 2), and they also adjoin (if Stride 4 were one item longer, then its last element would be Stride 1's first element). Stride 3 has a different stride distance from the Stride 1 so it is not mergeable with it. Stride 4 is not mergeable with Stride 1 because it contain only addresses which are even, while Stride 4 contains only odd addresses. This merging problem can be captured mathematically with the stride start modulo the stride distance. Finally, Stride 5 has the same Stride distance and is on evens, but does not adjoin Stride 1. The strides are not contiguous since 10 and 12 are not included in either, so Stride 5 is not mergeable with Stride 1.

The type of non-mergeable accesses represented by Strides 3 and 4 are separated out with the outer map of the StrideTable, which separates strides with different stride distances or stride starts into different ordered maps.

Now, the inner map needs to merge all adjoining strides, while not merging strides like Stride 5. This problem is exactly the problem of merging overlapping intervals, which is already well understood. But to be complete, a solution outline will be detailed here.

The inside map is ordered by the first element in the interval (which just happens to be a stride). Overlapping intervals are always merged, there will never be overlapping intervals in the map. This property will be used to guarantee efficiency.

To see how this can be done, consider merging in Stride 1 into the StrideTable when Stride 2 and 5 were already added.

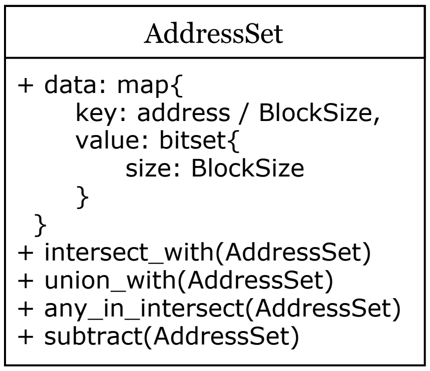
Current intervals	(0,2),(14,20)
Adding interval	(4,8)

Since intervals in the table will never overlap, the interval directly below the added interval (the (0,2) interval) is the only possible interval that starts before it, and also could overlap. The intervals above the added interval (in this case, (14,20)) can be efficiently checked because they are mergeable if and only if their first element is less than the added interval's last element.

2.4.3 Adding addresses to the AddressSet

The AddressSet is implemented as a bit compressed set as described in Section 1.6.2. That is, it is a table where each entry is a key value pair where the key is an integer which is equal to the $\lfloor \text{address/BlockSize} \rfloor$, and the value is a bit array of length BlockSize. Figure 2.5 shows a summary of this data structure.

Figure 2.5: AddressSet data structure



```
AddressSet
+ data: map{
      key: address / BlockSize,
      value: bitset{
            size: BlockSize
      }
  }
+ intersect_with(AddressSet)
+ union_with(AddressSet)
+ any_in_intersect(AddressSet)
+ subtract(AddressSet)
```

The table is implemented with a C++ `map`. The bit array is implemented with a C++ standard library's `bitset` which has fast intersection and union operations which are based on the C++ bitwise logical operators `&` and `|`. Bit access and assignment are also simple to use, treated syntactically similar to array access, but are implemented as a mix of array accesses and bit twiddling.

New addresses can be added to the set with a single line of C++:

```
bitset_table[memaddr / BlockSize][memaddr % BlockSize] = 1;
```

This line depends on a quirk of the C++ `map` class: new entries are automatically created when accessed for the first time.

Other operations of AddressSet

The AddressSet has other operations that are implemented. For example, during conflict checking, the intersection of the sets in the history and pending tables is taken, and during merging, the union of the sets is taken. These operations follow a certain pattern. In-place set intersection is shown in Algorithm 9 as a representative example of how these operations are evaluated. The general pattern is to treat the map of bitsets like a sparse set, and if the two maps have a matching entry, then treat the bitset as a dense set. This method results in linear time operations with respect to the map in the for loop.

Algorithm 9 BitSetIntersect

```
1: function BitSetIntersection(Dest,Src)
2:     for $\text{block}, \text{bitset} \in \text{Dest}$ do
3:         if $\text{blocknum} \in \text{Src}$ then
4:             bitset &= Dest.get(blocknum)
5:         else
6:             Src.remove(blocknum)
```

2.4.4 Checking conflicts

In SD^3, there is a single procedure that evaluates memory conflicts using the point and stride tables. We broke this into two procedures for performance reasons. The first procedure, IsParallel, is a fast operation that gives a binary answer: either the loop is parallel or it is not. The second, slower operation, FindConflictingInstructions, finds the particular pairs of instructions which accessed the conflicting memory. Later, these instructions are reported to the user. FindConflictingInstructions is only run when IsParallel returns false, since there would never be conflicts in a parallel loop. Figure 2.6 shows the general outline of the relevant operations and data structures, which we describe below.

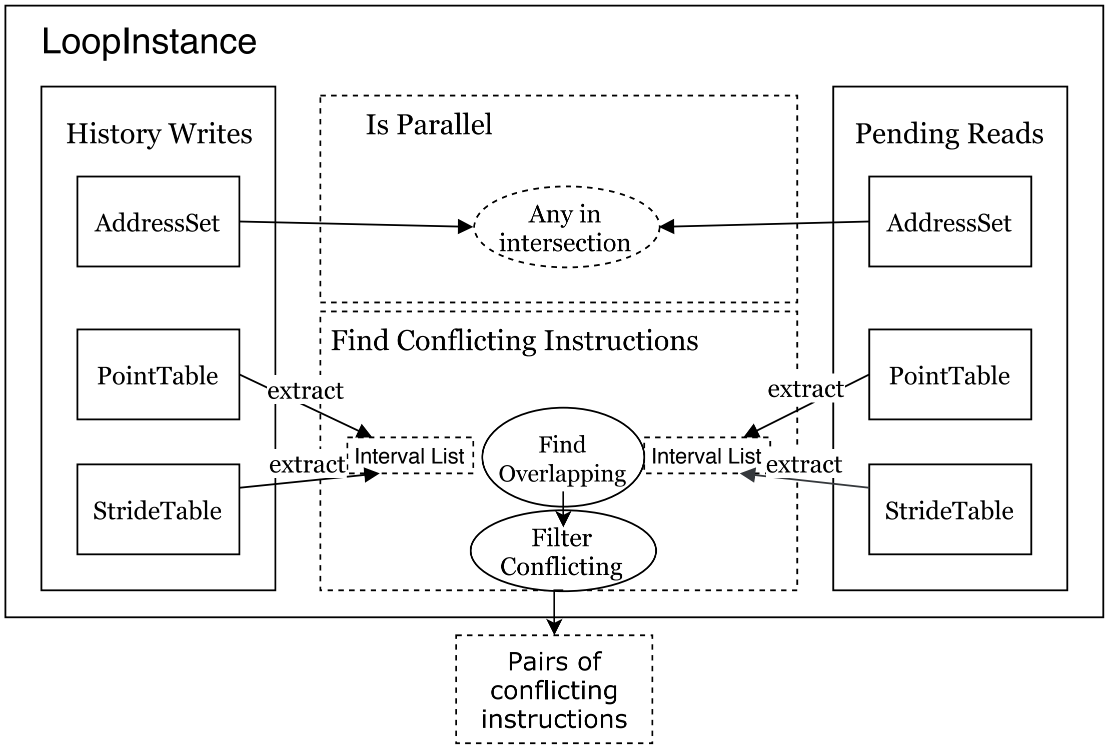
IsParallel

The AddressSet makes this operation simple and fast to compute. The intersection of the history write set and the pending read set is checked. The loop iteration is parallel if and only if that intersection is empty. If the intersection is not empty, this implies that a byte of memory has a read after write dependence which prevents the code from being parallelized.

FindConflictingInstructions

FindConflictingInstructions is broken into two steps. First, lists of all the strides and points in the stride and point tables are gathered. Then, these strides and points are treated as intervals, and the intervals that overlap are found in our FindOverlapping function. Finally, whether the strides and points actually overlap is calculated in our FilterConflicting function.

Figure 2.6: Conflict-Check outline



Find Overlapping

FindOverlapping finds pairs of overlapping intervals. This is another well known interval problem. We use a sorted list technique. The idea behind the algorithm is to sort the lists of intervals by their first element, and then go through all the intervals in one of the lists, and find every element in the other list that overlaps with it. In order to do this efficiently, information about the second list is stored. Algorithm 10 gives a precise description of this algorithm.

Filter Conflicting

Recall that the SD^3 paper uses the efficient Dynamic-GCD algorithm to check if a pair of overlapping strides really do conflict. We found it difficult to work out the details of this algorithm, so we instead decided to use a different method. It is based on a slow method of checking strides that compares the strides element by element. As an optimization, we use case specific methods of checking that are checkable by methods simpler than Dynamic-GCD and faster than element comparison. We built these case specific methods by making several observations about strides.

1. Many strides will be dense, meaning every byte in between the start and end of the interval will be accessed. Any overlapping strides are assumed to conflict if one of them is dense. This assumption is not always true and could result in false positives.

Algorithm 10 Find Overlapping

```
find_overlapping_intervals(A : interval list, B : interval list)
    sort A by first
    sort B by first
    overlap = Set()
    // before_last contains all interval from B that start before
    // the current interval from A starts and end after the current
    // interval ends
    before_last = vector()
    // ib is the first interval of B which starts after the interval
    // from A starts.
    ib = 0
    for interval in A
        // removes elements from before_last efficiently
        new_before_last = vector()
        for item in before_last:
            if item.last >= interval.first:
                new_before_last.append(item)
        before_last = new_before_last

        // Find the ib that has the property for the current interval
        // while adding in new elements to before_last
        while B[ib].first() >= interval.first()
            if B[ib].last() >= interval.first()
                before_last.append(B[ib])
            increment ib

        // report overlaps for the intevals in before_last
        for bl in before_last
            overlap.add(bl,interval)

        // report overlaps for the intervals which start after
        // interval starts
        for ib2 = ib until B[ib2].first > interval.last
            overlap.add(B[ib2],interval)
            ib2 = ib2 + 1
```

2. Many other possible conflicting strides will have the same stride distance. In this case, the strides conflict if and only if

$$(\text{stride-start}_1 - \text{stride-start}_2) = 0 \mod (\text{stride distance})$$

3. Many strides will have long interval overlap compared to the stride distances. Formally this is calculated by

$$\text{LCM}(\text{stride distance 1}, \text{stride distance 2}) < \text{interval operlap}$$

   Where LCM is the least common multiple. In this case, the ordinary GCD test described in Section 1.5.2 gives the correct answer; the strides conflict if and only if the GCD test passes.

Our method verifies if an observation holds for a particular case. For example, in the first case, it checks if one of the two strides is dense. Then it uses the properties of the case to check if the strides really do conflict. For example, in the third case, by performing the GCD test to check if the strides conflict. If none of the properties are found to hold for a given pair of strides, then the two strides are checked for conflicts with the slow algorithm which compares the strides element by element.

Truncation of FindConflictingInstructions

Whether the loop has conflicts or not, we do not want to run FindConflictingInstructions for every loop iteration or else the algorithm would run in quadratic time with respect to the number of loop iterations, which would be untenable. We reduce the number of times we run FindConflictingInstructions to a constant number per loop instance by making two observations:

1. If the loop iteration does not have any conflicts (discovered by running IsParallel), then we know that no instructions have conflicts, and we have no reason to run FindConflictingInstructions.
2. If the loop does have conflicts, then the same conflicts most likely occur in many other loop iterations. So, after we have found the locations of conflicts in some constant number of loop iterations, we simply do not check later iterations. The information reported may be incomplete, but some conflicts will always be reported.

Combined, these two truncation techniques mean that we have reasonably accurate reporting while only running FindConflictingInstructions a constant number of times for a particular loop instance.

2.4.5 Loop termination

When a loop terminates and it is not the outermost active loop, then it must be merged into its parent loop. Every Point and Stride is extracted from the outer loop's history table, and unless the point or stride has been killed, it is added to the inner loop's pending table using the add method described in Section 2.4.2.

Killing points and strides

Recall that when a history table is merged into the outer loop's pending table, killed addresses must not be added to the pending table. The SD^3 algorithm removes killed points and strides by a similar algorithm to their conflict-check method. We used AddressSets to do this instead. Given a particular point or stride, we simply use the AddressSet to check if the bytes that a point or stride describe have been marked as killed. Strides and points are only removed if every byte they reference is killed. This could result in false positives when identifying conflicts, but we do not believe that this will be a common occurrence.

2.5 Bit Compression Implementation

Stride compression introduces all sorts of complexity, creates performance issues, and does not guarantee good compression on all workloads. Bit compression (introduced in Section 1.6) has the potential to be the basis for a much simpler, faster compression system.

2.5.1 Main ideas

For our implementation, we combine the naive method's approach of listing instructions for each memory address and bit compression's approach of breaking up the address space into contiguous blocks which are stored as bit arrays. This approach will allow us to keep track of instruction information while compressing memory addresses in bit arrays. The interactions of these ideas are explained below.

Figure 2.7 shows how the naive implementation stores the accesses listed in Table 2.2. The accessed memory addresses all store a list of instructions that accessed that address. For example, memory address 3 stores the list that just contains instruction 15.

Table 2.2: Example accesses

Instruction	MemAddress
21	0
15	0
15	3
3	10

Figure 2.8 shows how the bit compression implementation stores this same set of accesses with a BlockSize of 4. Similar to the bit compressed set described in Section 2.4.3,[8] the table splits the address space into fixed width blocks. Each block stores two things. Most importantly, it stores a map of instructions to the addresses in the block which that instruction accessed (second row of the BlockInfo). In order

[8]Note that the exact memory addresses accessed are identical to the example in Figure 1.3, so that figure can be used for additional context into this implementation.

to speed up conflict-check, it also stores a bitset of overall accesses (first row of the BlockInfo).

This main data structure is used in the pending and history tables. Again, there are two of these data structures in each pending and history table, one for reads and one for writes.

Figure 2.7: Naive implementation table

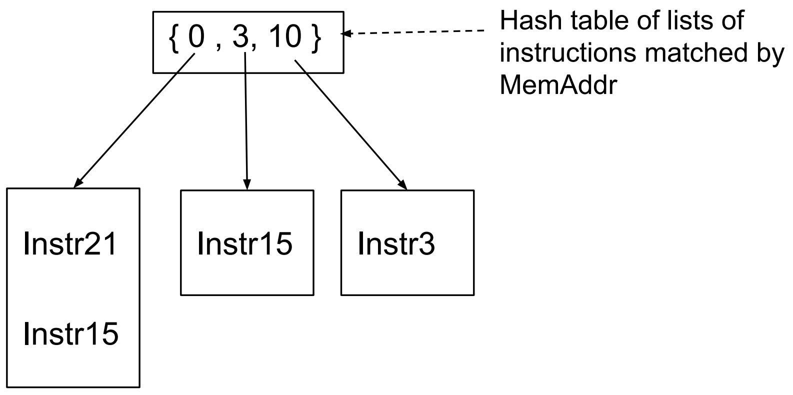
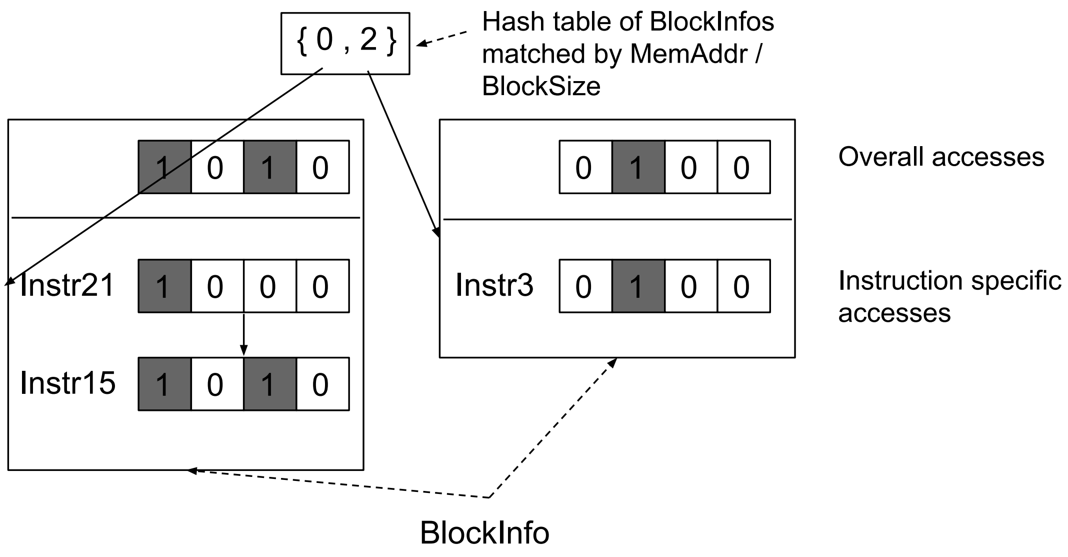


Figure 2.8: Bit compressed table



Implementation details of data structures

All key value maps in our bit compressed table are implemented with the C++ standard library `map`. Under the covers, this is implemented as a binary search tree, but we only use functionality that hash tables also have.

To implement the bit array, we use the C++ standard library `bitset`, which allows for simple bit access and assignment and also union and intersection operations. Under the covers, this is just an array of integers. The C++ `bitset`'s union, intersection, and set subtraction operations are extremely fast, because they are implemented with C++'s logical bitwise operations `|`, `&`, and `~`. These operations process all the bits in a single integer in parallel.

The address space is divided into blocks of 256 bytes, resulting in a 256 length bitset, which takes up 32 bytes of memory. This number was chosen because in the best case this allows for over 100 times the memory efficiency of the naive method, and in the worst case, it is only around 5 times worse than the naive method, since it only stores one blockset per instruction, and an instruction takes 8 bytes to store.

2.5.2 Adding new memory

When a byte of memory is accessed by an instruction, then

1. If the byte is killed (meaning it appears in the write table, but not the read table), then do nothing, and exit.
2. If there is no BlockInfo element that covers the address in the pending table, then add an empty one.
3. Add the byte accessed to the overall accesses bitset inside its BlockInfo.
4. If the instruction is not in the BlockInfo, add it.
5. Add the byte accessed to the bitset associated with the instruction address.

If multiple bytes are accessed by the same access, then this process is just repeated for each byte.

2.5.3 Conflict-check

When the pending table is merged into the history table, pairs of conflicting instructions need to be found.

Note that an access can only conflict with other accesses that have the same block identifier, MemAddress/BlockSize. If a pending table has a BlockInfo specified by a certain MemAddress/BlockSize, and the history table does not, then none of the accesses in that BlockInfo element in the pending table conflict with any elements in the history table.

The non-trivial important case is when a BlockInfo element is both in the pending and history tables. In the worst case, this is determined by comparing each pair of instructions in the respective BlockInfo elements, and checking if the intersection of their respective bitsets is empty. The overall access bitset is used to prune this computation, because if a bitset for an instruction does not conflict with the overall bitset in the other BlockInfo, then it also does not conflict with any of the instructions in that other BlockInfo. If the intersection is not empty, however, then a conflict exists between that instruction and at least one of the instructions in the other BlockInfo. Then the exact instruction(s) that conflict are found with exhaustive search.

2.5.4 Merge

Merging tables is as simple as merging the sets of instructions in the matching BlockInfo elements, and computing the union of the bitsets associated with a particular instruction in a BlockInfo. The union of the overall access bitset is also calculated.

2.5.5 Killing accesses

Recall that the accesses which are written but not read in the pending table are killed, and no further accesses to that address should be added.

So when adding a new access or merging in a BlockInfo into the pending table, a bitset of killed addresses for that block is found using the overall accesses bitset in the read and write parts of the pending table. This bitset of killed addresses removes the accesses in the BlockInfo by performing set subtraction on each bitset in the BlockInfo. Set subtraction on bitsets is easily implemented using bitwise negation and intersection.

Chapter 3

Results

3.1 Intro

Having created the tools described in Chapter 2, we now evaluate the effectiveness of these tools when run on a set of applications. First we evaluate the memory consumption of the stride and bit compression implementations. Then, we examine how important the parallel loops are to the applications' performance. We select the parallel loops that are found to be important, and examine some characteristics of these loops including the degree of parallelism, the variance of the length of the loop iterations, and the scale of the loop. Finally, we examine the workload characterization metrics in important parallel loops.

3.1.1 Running applications

All the results were obtained by running a set of applications selected from the Rodinia and Parsec benchmark suites introduced in Section 2.3. Table 3.1 lists all the applications we examined, the benchmark suites the applications came from, and the command line arguments used during our analysis.

Note that when the command line argument is a filename, it does not include the full path to the data file. The data files that are used are all ones that were included in the original benchmark suite; no new files were created.

3.2 Memory improvement of compression

Recall that the effect of stride compression on memory demands, and to a lesser extent bit compression, are contingent on program characteristics. To make sure that the compression methods we implemented are effective and to see how the two compression methods compare, we measured the memory consumption of these different approaches on the applications.

Table 3.1: Benchmark application and their inputs

Application	Suite	Command line arguments
B+tree	Rodinia	`b+tree.out core 2 file mil.txt command command.txt`
Back Propagation	Rodinia	`backprop 655360`
Breadth First Search	Rodinia	`bfs 4 graph1MW_6.txt`
Heartwall	Rodinia	`heartwall test.avi 5 4`
Hotspot	Rodinia	`hotspot 1024 1024 2 4 temp_1024 power_1024 output.out`
Hotspot3d	Rodinia	`3D 512 8 100 power_512x8 temp_512x8 output.out`
kmeans	Rodinia	`kmeans -i kdd_cup`
lavaMD	Rodinia	`lavaMD -cores 4 -boxes1d 10`
LU Decomposition	Rodinia	`lud_omp -s 2000`
Myocite	Rodinia	`myocyte.out 100 1 0 4`
Nearest Neighbor	Rodinia	`nn ./nn/filelist_4 5 30 90`
Needleman-Wunsch	Rodinia	`needle 2048 10 2`
ParticleFilter	Rodinia	`particle_filter -x 128 -y 128 -z 10 -np 10000`
PathFinder	Rodinia	`pathfinder 100000 100`
SRAD	Rodinia	`srad 2048 2048 0 127 0 127 2 0.5 2`
StreamCluster	Rodinia	`sc_omp 10 20 256 4096 4096 1000 none output.txt 4`
Black Scholes	Parsec	`blackscholes 1 in_64K.txt prices.txt`
Canneal	Parsec	`canneal 1 15000 2000 400000.nets 128`
Facesim	Parsec	`facesim -timing -threads 1`
Fluidanimate	Parsec	`fluidanimate 1 5 in_300K.fluid out.fluid`
Streamcluster	Parsec	`streamcluster 10 20 128 16384 16384 1000 none output.txt 1`
Swaptions	Parsec	`swaptions -ns 64 -sm 40000 -nt 1`

Memory evaluation setup

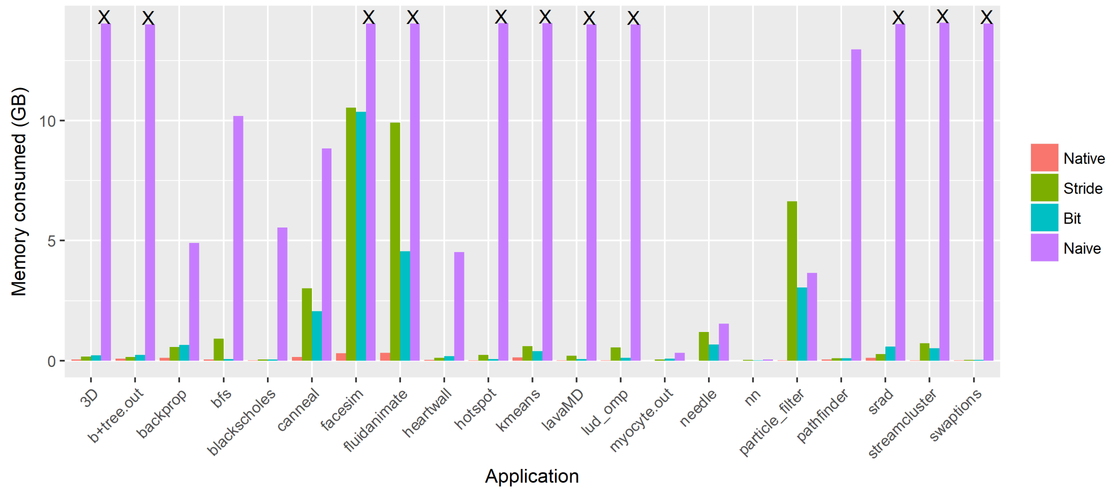
The maximum memory consumption used by our tool while running an application was measured by querying the Unix `ps` utility approximately every 20ms over the entire run of our tool. The maximum of all such queries is reported as the maximum memory consumed. As such, the resulting memory consumption is only an approximation. In particular, if the program allocated memory and immediately freed the memory before `ps` was queried again, that additional memory would not be measured. We do not believe that this measurement issue will cause serious error in measuring our tool's memory consumption because the tool runs slowly and consumes large quantities of memory throughout the run.

Inadvisablilty of the naive method

The naive method implementation often took up more memory than an ordinary desktop would have. The full results are plotted in Figure 3.1. Our system had 16GB of memory, and so the process was terminated after the program used over 14GB (on the plot, this is marked by an X). The naive method used more than 14GB of memory on 11 of the 21 applications, while neither the stride nor the bit compression methods used that much on any of the applications. This result confirms Kim et. al.'s observations that the naive method is not useful for important practical purposes, and that stride compression does help solve the problem.[1]

[1]Kim et al. (2010)

Figure 3.1: Memory consumption of parallelism detection on applications



Bit compression vs. Stride Compression

We now consider the respective memory overhead of stride compression[2] and bit compression. Recall that in theory, stride compression can also do much better than bit compression. In the best case, stride compression can make profiling memory consumption independent of input size, as the entire working memory can be compressed into a single stride. Of course, this ideal scenario is highly unlikely in real programs, and in the worst case, little memory is accessed in a strided manner, and stride compression does no better than the naive version. Bit compression claims to be less dependent on program characteristics, as it stores blocks of contiguous accesses, but it can also do as poorly as the naive method in cases where each instruction only accesses a single piece of memory in each block. Determining which method is better needs to be measured on real applications.

Overall, the performance of the methods depends on the applications. As seen in Figure 3.1, bit compression sometimes performs better and sometimes worse than stride compression. For example, stride compression gets better results on `srad`, `heartwall`, and `backprop`, while bit compression does better on `bfs` and `fluidanimate`. But when bit compression performs better, it sometimes performs much better, while stride compression only does better by a little. For example, `srad` is the application where stride compression beats bit compression by the greatest margin, using only 47% of the memory of bit compression. Meanwhile, for `bfs`, bit compression uses only 6.2% of the memory of stride compression, a huge improvement.

One explanation for why stride compression performs poorly while running `bfs` is because the main process of `bfs` is not stride friendly. In the `bfs` (breadth first search) code, the graph is implemented as an adjacency list. Code Example 11 is the inner loop of `bfs` copied from the source code. The `h_graph_edges` array stores the edge list, and the value it stores is used as an index into the `h_graph_visited` array.

[2]As a caveat, note that this stride compression implementation is not completely comparable to SD^3, since we added an AddressSet, which SD^3 does not store.

Since the locations at which the memory is accessed is dependent on the values in the input, the accesses to the elements of `h_graph_visited` will not necessarily be strided, that is, separated from one another by a fixed distance.

These non-strided accesses will then be stored as points, consuming as much memory as accesses in the naive method. Since these accesses are a significant part of overall accesses in `bfs`, memory performance as a whole will suffer.

Code Example 11 Breadth first search inner loop

```
for(int i=h_graph_nodes[tid].starting;
     i<(h_graph_nodes[tid].no_of_edges + h_graph_nodes[tid].starting);
     i++) {
    int id = h_graph_edges[i];
    if(!h_graph_visited[id]) {
        h_cost[id]=h_cost[tid]+1;
        h_updating_graph_mask[id]=true;
    }
}
```

Another surprising result is that the naive method uses comparable memory to the compressed methods on several applications, most notably `particle_filter` and `needle`. This suggests that there are some applications which neither compression method works effectively on.

3.2.1 Future work

Applications are continuing to become more and more memory intensive, so we will continue to need better memory compression methods to support that growth. We were considering using the SPEC 2017 applications for this thesis, but found that some of the applications on “train” inputs were consuming too much memory for our analysis. When running the “intspeed” benchmark (a subsection of the SPEC 2017 benchmark), the deepsjeng_s application natively used almost 7GB of memory. When running the fpspeed benchmark, the lbs_s application used 3.1GB of memory.

How much memory might the parallelism detection tool consume when run on memory intensive applications like these? The application `facesim` consumed 310MB of memory natively, and the both the compressed parallelism tools consumed over 10GB of the memory when profiling it, so the tools used 32x more memory than the native application. Assuming that an application like deepsjeng_s or lbs_s has a similar overhead ratio, both of these applications could have an overhead of over 100GB, far more than ordinary systems have.

One possibility to improve compression ratios is to combine the stride and bit compression. This can be done by implementing the point table using bit compression, and possibly by putting short strides in the point table when they cannot be merged in with other strides.

3.3 Identifying important parallelism

One of the goals of this thesis is to use parallelism detection to aid further analysis, for example estimating the effectiveness of GPU acceleration. This section evaluates whether the parallelism tool we built can be used for this purpose. In particular, we confirm that the tool does detect loops which could possibly be accelerated by parallel hardware, including massively parallel hardware like GPUs, for meaningful performance improvements.

Caveat for the correctness of results

As discussed in Section 2.2.1, the parallelism detector does not detect dependencies carried through stack and register variables. This implies that it may not give correct results on certain algorithms where dependencies are carried by local variables. While we believe that this is uncommon, this remains an unverified assumption, so all the results in this section should be evaluated in the context of that assumption.

Significance of loops

Most loops consume a tiny fraction of the applications' running times. Programmers usually want to optimize entire applications; individual loops are only optimized when significant portions of the application's running time is spent in the loop. Since the performance of these loops are unimportant to the programmer, they are not worth investigating and should be eliminated from further analysis. Figure 3.2 shows the percentage of running time spent in the loops (estimated by machine instructions executed), how many iterations these loops have, and whether the loops were found to not have data conflicts (which means they are parallelizable). This plot includes all loops from all applications.

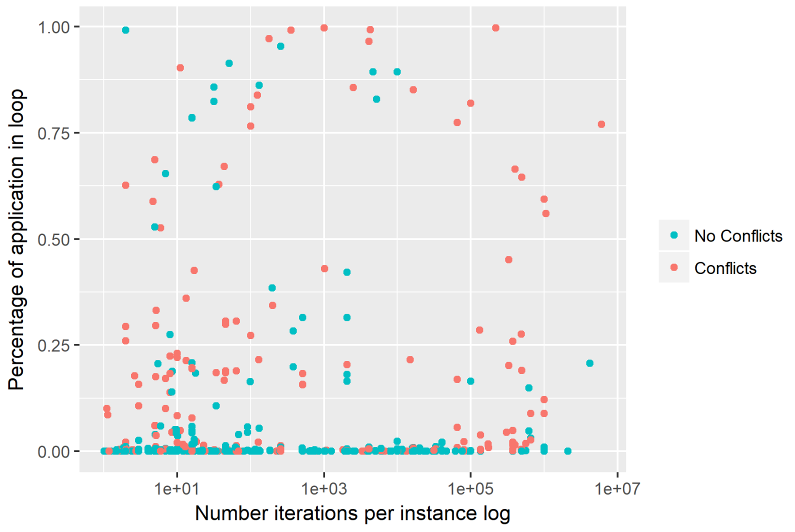
This plot shows that there are several loops which can be trivially parallelized for significant application speedup. These are the loops which are parallel (do not have conflicts) and have substantial number of iterations (blue dots, upper right). These can be reported to the programmer as targets of trivial parallelization.

The important loops that are not parallel (have conflicts) but have a large number of iterations are also important (red dots, upper right). These loops could possibly be parallelized with a reduction or with data synchronization and should be reported to the programmer for further investigation.

The dots in the lower-middle are less important than the ones at the top of the plot, but they can also substantially improve application performance. If several smaller parallel loops are present in an application, then parallelizing each of them could result in similar performance gains to parallelizing a large loop.

But the dots at the very bottom are unimportant to application performance, and so should be ignored. This eliminates most of the loops from consideration. 75% of the loops include less than 1% of the total runtime of their respective application.

Figure 3.2: Loop importance and parallelizability



Plotting all loops in all applications, loops have conflicts if any iteration conflicted with any other.

Parallelizability of applications

The parallelism detection tool will only be useful to programmers if it detects parallelism in a majority of parallel applications. We evaluate this characteristic of the parallelism tool by checking if at least one important loop in each application we evaluate has a useful degree of parallelism. This is a fair evaluation of the parallelism tool because we already know that the Parsec and Rodinia benchmarks include parallel implementations of all of the applications under consideration.

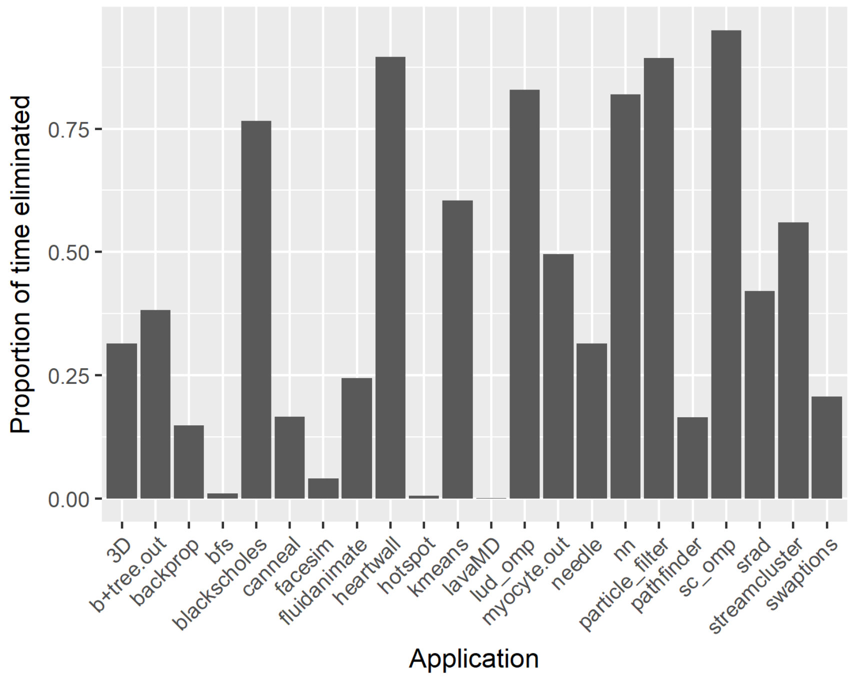
We introduce a metric for the percentage of time spent that can be eliminated from the application by parallelizing the loop. It assumes a sort of idealized parallelism. It is calculated as

$$\text{time eliminated} = \frac{(\text{instructons executed in loop}) \times \frac{\text{iterations per instance}-1}{\text{iterations per instance}}}{\text{instructions executed in application}}$$

The percent of time eliminated term, $\frac{\text{iterations per instance}-1}{\text{iterations per instance}}$ assumes no overhead, infinite cores, and uniform loop iterations. These assumptions are extremely optimistic, and so this estimate of time eliminated may also be overly optimistic. But it gives a rough estimate of parallelizability. Many of the loops which score highly are in fact very useful to parallelize, which we know since the benchmark suite successfully parallelized them. The loops which score poorly are certainly not useful to parallelize, as there is little gain found even with these generous assumptions. So this metric can be interpreted as an upper bound on how much the loop can reduce application execution time if parallelized.

This estimate of application speedup is a poor indicator of application parallelizability, but it can give us some confidence that the parallelism tool will in fact be somewhat useful for many different applications.

Figure 3.3: Performance improvement by idealized parallelization



Speedup by parallelizing the best loop in the given application assuming no overhead and infinite cores.

According to this metric, we detected some substantial parallelism in most of the profiled applications. Figure 3.3 shows speedup of entire applications assuming idealized parallelization of the loop in the program which results in the greatest time eliminated. `bfs`, `hotspot` and `lavaMD` show little potential for parallelism, as none of the parallel loops are very important to the application. In contrast, `heartwall`, `LUD`, and `blackscholes` show promise for large performance gains by parallelizing a single loop. Others applications are more ambiguous, such as `swaptions` and `fluidanimate`, because while parallelizing a single loop cannot increase application performance that much, it is possible that parallelizing several loops can in fact improve performance substantially. Since this plot only gives information about a single loop, it cannot say either way.

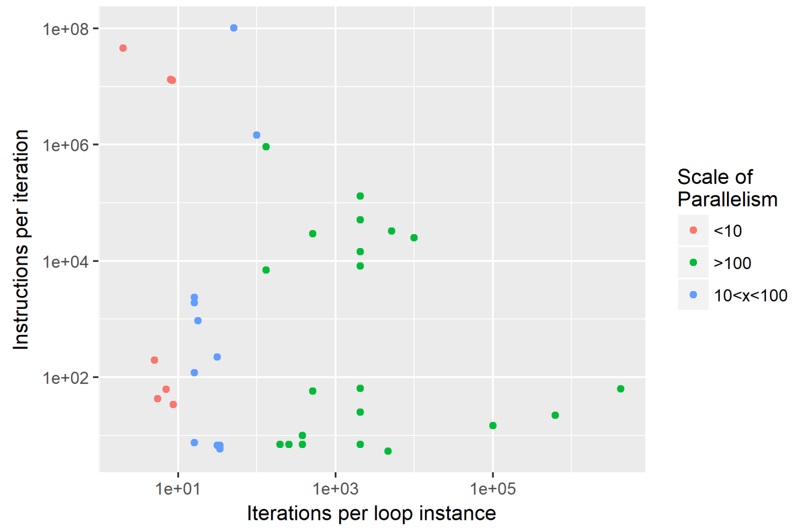
Scale of detected parallelism

The scale of the parallelism is one of the most important considerations when evaluating what sort of hardware to run the loop on. There are two aspects to scale.

First, there is the number of threads the work can be divided into. GPUs can run thousands of threads in parallel, while CPUs can at most run a few dozen. Since loop iterations are usually uniform, this can be estimated with the number of iterations in a loop instance.

Second, there is the amount of work that is done in each thread. Unlike idealized parallelism, real world parallelism on both CPUs and GPUs requires substantial time

Figure 3.4: Parallelism scale of important parallel loops



Loops which could speed up the program by 10% or more assuming idealized parallelism.

for the hardware to divide the work into parallel threads at runtime. This means that loops which do little work per iteration and have few iterations may not be effectively parallelized on either GPUs or CPUs. Figure 3.4 plots these two metrics for the important parallel loops in all the applications.

CPUs can run a small number of threads effectively and are ideal for parallelizing loops with a small number of iterations where each iteration does substantial work (upper left, colored red). GPUs are ideal for loops which have a large number of iterations (colored green). Loops which have a middling number of iterations (colored blue), and also have substantial work done per iteration (upper part of the graph), could be parallelized more effectively on a GPU or a CPU depending on additional details of the hardware under consideration.

However, the loops in the bottom left of Figure 3.4 may not be parallelizable on hardware because the overhead of distributing iterations of the loop onto either CPU or GPU threads may be greater than the benefit of parallelizing the loop. The parallelism detection tool should inform the programmer about this problem, or the programmer may be misled by the idealized analysis into thinking that the loop would be valuable to parallelize.

Uniformity of useful parallel loops

It greatly simplifies analysis to assume that important parallel loops are uniform, i.e., all the loop iterations do the same amount of work. Notably, this assumption was utilized in our analysis of parallelizability of applications and scale of parallelism. While this is true for many loops, some loops have substantial variability in iteration length that can impact analysis of parallelizability as well as the type of parallel architecture the loop is most suited for.

To see why this is a problem, consider the idealized parallel machine where an infinite number of threads can run in parallel. Consider a loop with 1000 iterations that can run in parallel, but one of those loop iterations takes as much time all the others combined. In this case, overall loop performance can at most be doubled, since the one iteration takes up half of the sequential time. But if you assume uniformity of iterations, then performance would be calculated to increase by 1000 times, so the actual and predicted speedup would be off by a factor of 500.

Luckily, for the applications we examined, most loops are roughly uniform. Figure 3.5 shows a metric of uniformity plotted over the number of iterations per loop instance. The metric chosen is the maximum iteration length over the mean length. Only loops which could speed up the program by 10% or more assuming idealized parallelism are shown.

The two dots in the top left of the graph are loops which have serious problems for parallelizability. They only have 10 loop iterations each on average, but the maximum iteration length is 10 times the mean iteration length. This means that performance may not be increased substantially by parallelization, since a single iteration dominates the runtime of the loop.

The loop that appears in the top right of the graph is not so problematic. After all, it has over 10,000 loop iterations. If the iterations are carefully distributed over threads, it is possible that the variance between iteration lengths would not have a significant impact on performance, since the smaller iterations could be coalesced together into threads comparable in length to the large thread. But since the programmer has to do extra work to accomplish this, the tool should report this non-uniformity to the programmer.

The loops which show moderate amounts of variance need to be similarly evaluated on a case by case basis. For 67% of the important loops shown, the maximum iteration length is less than 10% greater than the mean iteration length, so most of the important loops are in fact uniform.

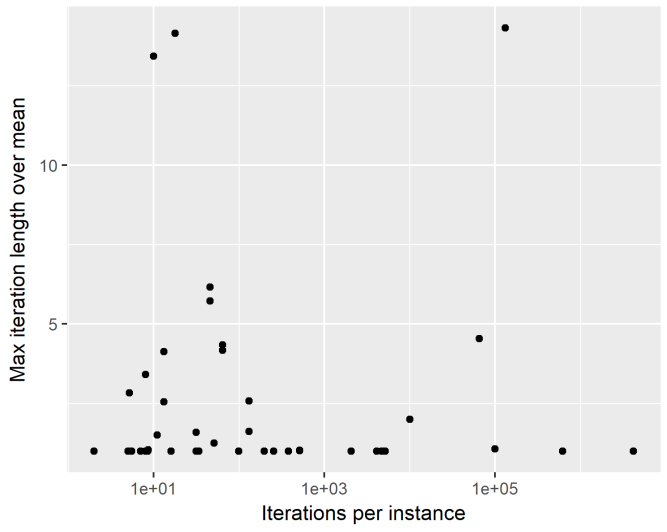
3.3.1 Future work

This section suggests that this parallelism information can be rather useful to both the programmer and any further characterization of the code. Future work could create a tool which a programmer can easily use to evaluate parallelism. The programmer would provide the inputs to the application, and the analysis tool will tell the programmer where to consider different sorts of parallelism.

A crude version of this tool was implemented to help debug the parallelism detector. It annotates the source code with comments containing the instruction addresses of the loops and with information about conflicts. Code Example 12 shows an example of this output on some matrix multiplication code. Note that the actual tool outputs more information, and the output is truncated here for better presentation.

In its current form, a developer can use this tool to find parallel loops and see how many iterations they run in. For example, the outer loop of the matrix multiplication has 0 conflicting iterations, which means it is parallelizable. This can help assure the programmer that parallelizing the loop is possible.

Figure 3.5: Uniformity of loop iterations in important parallel loops



Loops which could speed up the program by 10% or more assuming idealized parallelism.

For loops which have conflicts, the developer can track down where these conflicts are in the source code. In this case, the inner loop has a conflict. This conflict is labeled by the loop identifier so the programmer can easily track down which conflicts match with which loop.

Future work could improve the tool by:

- Adding more useful information about the loops such as the length and variance of their iterations. This should help the programmer evaluate whether the loops would be profitable to parallelize.
- Summarizing the important parallel and possibly parallel loops to investigate further. This may be important for a programmer to navigate large codebases with hundreds of loops.
- Add an overall score of GPU and CPU parallelism for each loop, perhaps with some kind of performance modeling.
- Adding static analysis that detects register and stack dependencies as discussed in Section 2.2.1.

3.4 Matching parallelism with other metrics

One goal of this thesis is to try to characterize programs that are effective on CPUs vs. GPUs. But as discussed in Section 2.1, choosing the right hardware requires investigating other program metrics than just the scale of parallelism. GPUs and CPUs have all sorts of different hardware characteristics which significantly affect the

Code Example 12 Example output of parallelism tool

```
const int size = 100;
int * A = random_array(size*size);
int * B = random_array(size*size);
int * C = random_array(size*size);
//LOOP 4006b9: ConflictedIterations: 0 TotalLoopIterations: 101
for(int i = 0; i < size; i++){
    //LOOP 4006ca: ConflictedIterations: 0 TotalLoopIterations: 10100
    for(int j = 0; j < size; j++){
        //LOOP 4006db: ConflictedIterations: 990000
        //TotalLoopIterations: 1010000
        for(int k = 0; k < size; k++){
            //PARALLEL_CONFLICT FOR 4006db: LaterPC: 400701
            //EarlierPC: 400760 NumConflictIters: 990000
            C[i*size+j] += B[i*size+k] * A[k*size+j];
        }
    }
}
```

viability of running certain loops on them. To capture some of these differences we collected the metrics described in Table 3.2 for each loop.

Variability of metrics in important parallel loops

These metrics will only be important in predicting overall performance if they are not consistent across all the loops. Figure 3.6 tries to capture single variable variance by showing how all the important parallel loops score on the different metrics. To interpret the importance of this figure, take the StridedAccessesAbove16 boxplot as an example. This boxplot shows three loops where over 50% of accesses are strided accesses where the stride length is over 16 bytes. The rest of the loops are shown to have less than 20% of this type of access. The significant difference between these groups of loops may have hardware implications that we can exploit to improve performance prediction. The most important takeaway of this plot is that it does suggest that there is substantial variance within each of these metrics. While this does not capture overall variance of these performance characteristics, it is a first cut on whether these metrics may turn out to be useful.

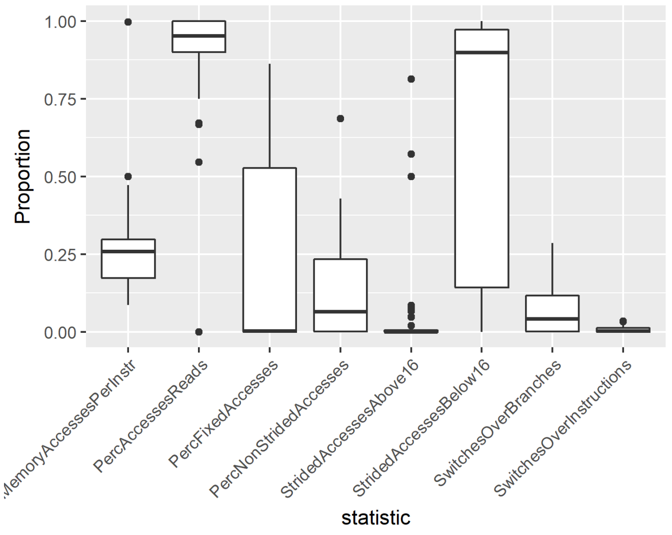
3.4.1 Future work

Unfortunately, due to time restrictions, we were not able to use the metrics to evaluate loop performance on different hardware. Future work would combine metrics similar to these with parallelism summaries discussed in Section 3.3 and build a predictive performance model which would indicate to a programmer if and how to parallelize

Table 3.2: Other metrics collected

Metric name	Metric description
SwitchesOverBranches	Number of time a branch path changes direction over the number of times a branch instruction is executed
SwitchesOverInstructions	Number of time a branch path changes direction over the number of instructions executed
PercNonStridedAccesses	Percentage of accesses which do not access memory as a stride
StridedAccessesBelow16	Percentage of accesses which are strides of length 16 bytes or less
StridedAccessesAbove16	Percentage of accesses which are strides of length 16 bytes or more
PercFixedAccesses	Percentage of accesses which are strides of length zero, i.e. they access the same location of memory location repeatedly
MemoryAccessesPerInstr	Number of memory accesses over number of instructions
PercAccessesReads	Percentage of memory accesses which are reads

Figure 3.6: Other metrics of important parallel loops



Loops which could speed up the program by 10% or more assuming idealized parallelism the loop. Such information could be displayed to the programmer by a tool like is described in Section 3.3.1.

Conclusion

This thesis set out to create a tool that helps programmers understand how their code might best utilize heterogeneous computing. It addresses several challenges involved with completing this goal and leaves other challenges open for future work.

One of the problems encountered by prior work in parallelism detection is the massive memory overhead of the naive algorithm. We confirmed that the simplest implementation uses too much memory to run real applications on real systems, and implemented two methods for reducing the memory footprint: stride compression and bit compression. We found that these two methods use comparable magnitudes of memory on most of the applications examined, and both are vast improvements over the naive method. As programmers find new ways of making use of larger quantities of system memory, these techniques will not be good enough, so we hint that future work could combine bit and stride compression into a single tool.

Another potential problem with the naive method of parallelism detection is that it only finds a certain type of parallelism: trivial loop-level parallelism. This raises the question of whether it is useful in detecting parallelism that is actually important to program performance. We found evidence that it does detect important parallelism for most of the applications studied. To the extent to which the applications we examined are representative, this tool does in fact detect real parallelism. Future work could make the tool more accessible to the programmer by producing more informative output. Future work could also produce more accurate results by incorporating static analysis.

Finally, we started addressing the challenge of how to inform the programmer about software characteristics other than parallelism and their effects on hardware performance. We collected several loop level metrics other than parallelism, and suggested that they would be important in evaluating the effectiveness of accelerating the loops on GPUs vs. CPUs. Future work can determine how to pair those metrics with parallelism, how to interpret or model them in a way that has direct implications for performance, and evaluate if different metrics are needed. Future work could also evaluate new metrics for hardware platforms other than CPUs and GPUs.

If all these challenges are addressed, and a quality tool is produced, then programmers who use the tool should be much more able to evaluate what kind of parallelism, if any, is appropriate for different parts of their code. This will allow them to make more informed decisions about investing time to adapt the code to work on heterogeneous hardware.

References

Bienia, C., Kumar, S., Singh, J. P., & Li, K. (2008). The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, (pp. 72–81). New York, NY, USA: ACM. `http://doi.acm.org/10.1145/1454115.1454128`

Borkar, S., & Chien, A. A. (2011). The future of microprocessors. *Commun. ACM*, *54*(5), 67–77. `http://doi.acm.org/10.1145/1941487.1941507`

Che, S., Sheaffer, J. W., Boyer, M., Szafaryn, L. G., Wang, L., & Skadron, K. (2010). A characterization of the Rodinia benchmark suite with comparison to contemporary cmp workloads. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'10)*, IISWC '10, (pp. 1–11). Washington, DC, USA: IEEE Computer Society. `http://dx.doi.org/10.1109/IISWC.2010.5650274`

Chen, T., Lin, J., Dai, X., Hsu, W.-C., & Yew, P.-C. (2004). Data dependence profiling for speculative optimizations. In E. Duesterwald (Ed.), *Compiler Construction: 13th International Conference, CC 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004. Proceedings*, (pp. 57–72). Berlin, Heidelberg: Springer Berlin Heidelberg. `https://doi.org/10.1007/978-3-540-24723-4_5`

Fung, W. W. L., Sham, I., Yuan, G., & Aamodt, T. M. (2007). Dynamic warp formation and scheduling for efficient gpu control flow. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, (pp. 407–420). Washington, DC, USA: IEEE Computer Society. `http://dx.doi.org/10.1109/MICRO.2007.12`

Hoste, K., & Eeckhout, L. (2007). Microarchitecture-independent workload characterization. *IEEE Micro*, *27*(3), 63–72.

Jia, W., Shaw, K. A., & Martonosi, M. (2014). Mrpb: Memory request prioritization for massively parallel processors. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, (pp. 272–283).

Kim, M., Kim, H., & Luk, C.-K. (2010). Sd3: A scalable approach to dynamic data-dependence profiling. In *Proceedings of the 2010 43rd Annual IEEE/ACM Interna-*

tional Symposium on Microarchitecture, MICRO '43, (pp. 535–546). Washington, DC, USA: IEEE Computer Society. http://dx.doi.org/10.1109/MICRO.2010.49

Landi, W. (1992). Undecidability of static analysis. *ACM Lett. Program. Lang. Syst.*, *1*(4), 323–337. http://doi.acm.org/10.1145/161494.161501

Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., & Hazelwood, K. (2005). Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, (pp. 190–200). New York, NY, USA: ACM. http://doi.acm.org/10.1145/1065010.1065034

Moseley, T., Connors, D. A., Grunwald, D., & Peri, R. (2007). Identifying potential parallelism via loop-centric profiling. In *Proceedings of the 4th International Conference on Computing Frontiers*, CF '07, (pp. 143–152). New York, NY, USA: ACM. http://doi.acm.org/10.1145/1242531.1242554

Nickolls, J., Buck, I., Garland, M., & Skadron, K. (2008). Scalable parallel programming with CUDA. *Queue*, *6*(2), 40–53. http://doi.acm.org/10.1145/1365490.1365500

Rogers, T. G., O'Connor, M., & Aamodt, T. M. (2012). Cache-conscious wavefront scheduling. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, (pp. 72–83). Washington, DC, USA: IEEE Computer Society. http://dx.doi.org/10.1109/MICRO.2012.16

Taylor, M. (2013). A landscape of the new dark silicon design regime. *IEEE Micro*, *33*(5), 8–19. http://dx.doi.org/10.1109/MM.2013.90

Zahran, M. (2016). Heterogeneous computing: Here to stay. *Queue*, *14*(6), 40:31–40:42. http://doi.acm.org/10.1145/3028687.3038873