EFFECTIVENESS OF PROGRAM TRANSFORMATIONS AND COMPILERS FOR DIRECTIVE-BASED GPU PROGRAMMING MODELS

BY

SWAPNIL AVINASH GHIKE

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2012

Urbana, Illinois

Advisers:

Professor David Padua
Professor Maria Garzaran

ABSTRACT

Accelerator devices like the General Purpose Graphics Computing Units (GPGPUs) play an important role in enhancing the performance of many contemporary scientific applications. However, programming GPUs using languages like C for CUDA or OpenCL requires relatively high investment of time and the resulting programs are often fine-tuned to perform well only on a particular device. The alternative is to program in a conventional and machine independent notation and use compilers to transform CPU programs to heterogeneous form either automatically or relying on directives from the programmer. These compilers can offer the benefits of code portability and increased programmer productivity without imposing much penalty on performance.

This thesis evaluates the quality of early versions of two compilers - the PGI compiler and the Cray compiler, as tools for translation of C programs written for single or multicore CPUs to heterogeneous programs that execute on NVIDIA's GPUs. In our methodology, we apply a sequence of transformations to CPU programs that allow the compilers to form GPU kernels from loops, and then we analze the impact of each transformation on the performance of compiled programs. Our further evaluation of the performance of 15 application kernels shows that the executables produced by the PGI and Cray compilers can achieve reasonable, and in some cases equivalent performance as compared to hand-written OpenMP and CUDA codes. Our results also show that the Cray compiler managed to produce faster executables for more applications than the PGI compiler. We show that for a heterogeneous program to execute faster, the traditional analyses and optimizations needed for producing a good sequential program are equally if not more valuable compared to those needed to produce a good GPU kernel. At the end of this thesis, we also provide a set of guidelines to programmers for extracting good performance from the heterogeneous executables produced by the PGI

and Cray compilers.

To my parents, mentors and friends who cared,
for making me a better person.

ACKNOWLEDGMENTS

I would like to thank my adviser at the University of Illinois, Professor David Padua, without whose invaluable guidance this work would not have been possible. I hold his vast knowledge in the highest regard and truly appreciate the innumerable lessons and insights about research that he has taught me. It is also not often that one finds an adviser with a sense of humour as wonderful as his.

I must also thank Professor Maria Garzaran, my thesis co-adviser at the University of Illinois, for her contributions of time, advice, ideas and editing guidance. I am especially grateful for her resourcefulness and timely guidance that ensured that my research does not stop progress due to any problems.

I would like to express my immense gratitude towards Professor Ruben Gran Tejero, from Universidad De Zaragoza, who provided me valuable and constructive suggestions throughout the course of this project and helped me see through the maze of experimental data.

A very special thanks goes out to Donald Newell and Shuai Che, who are graduate students at the University of Virginia, for helping improve my understanding of Rodinia benchmarks and duly fixing any bugs in the benchmark suite that I reported.

I wish to acknowledge the assistance provided by the PGI Technical Support in solving any problems that I faced while using the PGI compiler and for fixing any bugs that I reported.

I am particularly grateful for the help provided by Dr. Brett Bode at the National Center for Supercomputing Applications in setting up my access to the JYC1 system and the Cray compiler.

Dr. Luiz DeRose at Cray Inc., equipped me with resources useful for performing experiments on the JYC1 system and also provided insights into certain functionalities of the Cray compiler. I would like to express my sincere appreciation for his help.

I would also like to extend my thanks to Albert Sidelnik, who is a graduate student at the University of Illinois, for providing me insights into the methods of measuring GPU performance.

As I submit this thesis, I could not be more appreciative of my grandfather, father and aunt who inculcated in me a curiousity about science right from my childhood. I feel also extremely lucky to have been loved and encouraged all along by my grandmother and mother.

I want to sincerely thank my family, teachers, and the many people for the time they put in me and got me to graduate school.

I must acknowledge my best friend, Priyanka, for being there through thick and thin.

I wish to acknowledge all the friends who provided me the much needed moral support and entertaining distractions. They know who they are.

I wish to acknowledge that this research is part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (award number OCI 07-25070) and the state of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign, its National Center for Supercomputing Applications, Cray, and the Great Lakes Consortium for Petascale Computation.

TABLE OF CONTENTS

LIST OF TABLES ix

LIST OF FIGURES x

CHAPTER 1 INTRODUCTION 1

CHAPTER 2 ENVIRONMENTAL SETUP 4
2.1 Target Platform 4
2.2 Compilers 4
2.3 Benchmarks 5
2.4 Performance Measurement 6

CHAPTER 3 ACCELERATOR DIRECTIVES 7
3.1 Parallelism Management 7
3.2 Data Management 10

CHAPTER 4 EFFECTIVENESS OF COMPILERS IN ANALYZING DEPENDENCES AND APPLYING LOOP TRANSFORMATIONS 12
4.1 Loop Transformations 12
4.2 Computation Patterns 18

CHAPTER 5 TRANSFORMATION OF OPENMP PROGRAMS TO HETEROGENEOUS PROGRAMS 20
5.1 Transformation Steps 20
5.2 Impact of Transformations on Performance 27

CHAPTER 6 PERFORMANCE OF RODINIA BENCHMARKS . . 32
6.1 Performance Comparison of Sequential Programs compiled by the PGI and Cray Compilers and Performance Comparison of OpenMP Programs compiled by the PGI and Cray Compilers 33
6.2 Performance Comparison of Heterogeneous Programs and their OpenMP Versions 35
6.3 Performance Comparison of Heterogeneous Programs and their CUDA Versions 39

CHAPTER 7 GUIDELINES FOR THE PROGRAMMER 49

CHAPTER 8 CONCLUSIONS 54

REFERENCES 56

LIST OF TABLES

2.1 Hardware Characteristics 4
2.2 Compilers and Optimization Flags 5
2.3 Rodinia Benchmarks 6

4.1 Abilities of the PGI and Cray compilers to apply loop transformations to micro-kernels 17
4.2 Abilities of the PGI and Cray compilers to automatically parallelize computation patterns 19

5.1 Transformations applied while using the PGI compiler 25
5.2 Transformations applied while using the Cray compiler 26
5.3 Relative times taken by the executions of heterogeneous programs compiled with the PGI compiler. 28
5.4 Relative times taken by the executions of heterogeneous programs compiled with the Cray compiler 28

6.1 Percentage of vector instructions produced by PGI and Cray compilers in sequential and OpenMP benchmarks 35
6.2 Device Kernel Metrics 45
6.3 Number of device kernels in the given occupancy range 47
6.4 Number of device kernels that allocate registers per thread in the given range 48

LIST OF FIGURES

5.1 Distribution of time spent by the heterogeneous programs after the application of transformations 29

6.1 Speedups of sequential programs compiled with the Cray compiler over their own performance when compiled with the PGI compiler and speedups of OpenMP programs compiled with the Cray compiler over their own performance when compiled with the PGI compiler 34
6.2 Speedups of OpenMP, heterogeneous and CUDA versions of Rodinia benchmarks over sequential versions, all compiled by the PGI compiler 36
6.3 Speedups of OpenMP, heterogeneous and CUDA versions of Rodinia benchmarks over sequential versions, all compiled by the Cray compiler 37
6.4 Ratio of times taken for the computation in GPU kernels in the heterogeneous versions of Rodinia benchmarks 40
6.5 Speedups of heterogeneous programs compiled by the PGI and Cray compilers over their CUDA versions 41

CHAPTER 1

INTRODUCTION

GPUs have started playing an important role in performance critical applications in computer graphics, scientific computing, gaming consoles and mobile devices, primarily because GPUs offer massive parallelism and computational power that is not available with the heavyweight multicore CPUs. GPUs have also shown better performance per watt in studies [1] conducted in the past using applications that represented varying domains and computational patterns. Many supercomputers including NCSA's upcoming Blue Waters supercomputer [2] at the University of Illinois employ GPUs. With the introduction of NVIDIA Tegra processor in tablet devices [3, 4], the applicability of heterogeneous architectures, i.e., architectures that combine a traditional general purpose CPU and specialized co-processors like the GPU, seems evident in all forms of computing.

As parallelism increases, memory bandwidth becomes a bottleneck. GPUs try to hide memory latencies by maintaining a large pool of thread contexts and switching between these contexts at virtually no cost. Thus, GPUs perform best when its multiprocessors are kept busy with hundreds or even thousands of threads that execute in parallel. However, extracting the best performance out of GPUs using APIs that are closer to hardware and C-like programming languages such as CUDA and OpenCL is a time consuming process for programmers. Correctness issues like avoiding deadlocks and race conditions, and performance issues like making optimal use of device memory bandwidth, avoiding cyclic CPU-GPU data movement and pipelining of asynchronous data movement have to be dealt with explicit instructions or algorithms by the programmer. Thus, increased performance comes at the cost of programmer productivity. On top of the existing programmer productivity drains, manually writing code using CUDA and OpenCL often leads to programs that are fine-tuned to perform only on a particular device. Performance portability thus becomes another major problem.

Loops are the natural candidates for translating to GPU kernels. A programmer can use CUDA or OpenCL to manually translate loops into GPU kernels. However, this task can be simplified by using compilers that can translate conventional CPU programs to produce heterogeneous executable code that is fine-tuned to the resources of the GPU being used. This compilation process may complete automatically or may need the programmer to provide hints or instructions in the form of directives. Providing directives usually takes less efforts than writing a CUDA or OpenCL program. The heterogeneous code can then offload loops to the GPU and take care of host-GPU data communication. Thus the heterogeneous compilers' approach has the potential to obtain the best of both worlds - good performance, if not the best as that obtained with optimized CUDA or OpenCL programs, and increased programmer productivity and code portability.

In this thesis, we analyze and evaluate the PGI compiler and the Cray compiler which can transform C/C++ loops to their heterogeneous versions and attempt to perform GPU resource optimizations. We insert directives to help the PGI and Cray compilers generate CUDA and ptx kernels respectively. These directives are useful in producing instructions that launch device kernels, manage the memory resources on the host and the GPU and initiate any data transfer between them. Dependence analysis and loop transformations play a critical role in exposing parallelism out of sequential loops. Using a set of 10 representative loops, we analyze the abilities of the compilers to detect loop dependences and transform the loops automatically to remove these dependences.

Some previous studies [5, 6, 7] proposed compilation or runtime frameworks that attempt to transform the sequential programs to heterogeneous code completely automatically, but they have not been effective and manual intervention was often required from the programmer. One of the goals of this thesis is to identify what manual intervention is required to allow the compiler to produce heterogeneous code that performs well and is portable. Our work involves proposing a sequence of manual transformations that can transform sequential or OpenMP programs to a form that can be understood by the PGI and Cray compilers. We also describe the impact of each transformation on the performance of heterogeneous programs by transforming the OpenMP versions of 15 benchmarks from the Rodinia suite to the corresponding heterogeneous versions. At the end of this thesis, we provide a set

of guidelines to programmers for extracting good performance out of these heterogeneous programs compiled using PGI and Cray compilers.

In addition, there has not been any comparison of performances obtained by using these compiling frameworks with the native CUDA implementation or with OpenMP implementation of programs to determine if using these special compiling frameworks produces good performance. The second goal of this thesis is to evaluate how well these compilers do in achieving good performance. To contribute to filling this vacuum, we compare the performances of the heterogeneous versions of programs compiled by the PGI and Cray compilers with the OpenMP and CUDA implementations of those programs. This comparison will allow us to analyze the performance capabilities of the PGI and Cray compilers, identify bottlenecks and determine the kinds of programs that are more suitable for transforming to their heterogeneous versions. Our results show that these heterogeneous versions perform within 85% of the CUDA performance in certain cases, and perform worse in other cases depending on various factors such as data communication between the host and the device, non-coalesced device memory accesses which depend on the structure of the program, etc. As discussed below, the heterogeneous compilers have scope to improve both in terms of efforts required to transform an OpenMP code and performance of the resulting heterogeneous executables.

The rest of this thesis is organized as follows: Section 2 describes the platforms, compilers and the benchmarks that we used; section 3 describes the various directives and attributes used by the heterogeneous compilers; section 4 is an evaluation of the abilities of these compilers to analyze dependences and perform loop transformations; section 5 presents the transformations required to convert OpenMP programs to their heterogeneous versions and the impact of each transformation on their performance; section 6 evaluates the performance of Rodinia benchmarks when compiled with the heterogeneous compilers and discusses the performance bottlenecks; section 7 presents guidelines to programmers for extracting good performance out of heterogeneous programs; and finally section 8 concludes. Throughout this thesis, we refer the directives-annotated programs compiled by the PGI and Cray compilers to produce heterogeneous code as the heterogeneous programs, to distinguish them from their native CUDA versions. We also refer to the PGI and Cray compilers as heterogeneous compilers.

CHAPTER 2

ENVIRONMENTAL SETUP

In this section, we describe the target platform, compilation tools, benchmarks and the methodology of performance measurement that we used for our experiments.

2.1 Target Platform

Our experiments were performed using a single Cray XK6 node that consisted of a 16-core AMD Opteron 6200 series processor [8] (formerly code-named Interlagos) and a NVIDIA Tesla X2090 GPU [9]. The specifications of these architectures are presented in table 2.1. The operating system used was SUSE Linux 2.6.32.45(.45-0.3.2_1.0400.6336-cray_gem_s).

2.2 Compilers

The sequential and OpenMP programs, and the programs containing accelerator directives were compiled using the PGI [10] and Cray [11] compilers. For the comparison of performance, we have also compiled CUDA [12] programs with the GCC/NVCC compilers. The optimization flags used by the compilers are noted in table 2.2.

Type	CPU	GPU
Name	AMD Opteron 6200	NVIDIA Tesla X2090
Cache	L2/L3 - 512kB/12MB	L2(shared) 42KB(Multiprocessor)
# cores	16 cores	512 CUDA cores
Peak Perf.	294.4GFLOPS	1331 GFLOPS
Frequency	2.1 GHz	1.3 GHz

Table 2.1: Hardware Characteristics

Specification	PGI	CRAY XK	NVCC/GCC
Version	11.10-0	CCE v 8.1.0.139	4.0.17a / 4.3.4
Baseline optimizations	-fast -fastsse -O3	-O3	-O3 / -O3 -Ofast
Platform specific optimizations	-tp=bulldozer-64	-h cpu=interlagos	-arch=sm_20 / -march=bdver1
Flags to disable all parallelism	None	-h noacc -h noomp -h noomp_acc	NA
Flags to enable only OpenMP parallelism	-mp=allcores	-h noacc -h noomp_acc	NA
Flags to enable only GPU parallelism	-ta=nvidia,keepgpu	-h noomp_acc	NA
Flags to generate compilation report	-Minfo	-hlist=m	NA

Table 2.2: Compilers and Optimization Flags

2.3 Benchmarks

We used two benchmark suites to evaluate the heterogeneous compilers. One of them is a home-made micro-benchmark suite and the other is the Rodinia benchmark suite version 2.0.1 [13, 14].

The home-made micro-benchmark suite consists of a set of loops that can test the abilities of the compilers to automatically perform certain loop transformations that can expose parallelism in the loops. A loop transformation is the modification of a loop nest that facilitates further performance optimizations while maintaining correctness. We tested the PGI and Cray compilers to check if they can perform loop transformations that remove loop carried dependences to expose the parallelism in loop iterations. These transformations are alignment, distribution, interchange, privatization, reversal and skewing [15]. Each loop nest requires a different and exactly one transformation to expose the parallelism in its iterations.

The Rodinia benchmark suite groups heterogeneous and representative applications of different domains of the engineering: data mining, bioinformatics, physics simulation, pattern recognition, image processing, medical imaging, graph algorithm, fluid dynamics and linear algebra. Table 2.3 shows the names of the benchmarks, their respective domains and the problem sizes.

The Rodinia benchmarks were available with two different implementations, OpenMP and CUDA. The CUDA implementations attempt to make the most efficient use of the GPU by exploiting the different types of memories available in the GPU: global, shared, constant and texture memories.

Out of the 16 benchmarks available in Rodinia benchmark suite, we have used 15 of them. We did not include the benchmark Mummer-GPU because of the limitations of using unions in the compilers that we studied. We have elaborated more on the use of unions in section 7 under the title Miscellaneous issues.

Benchmark	Acronym	Area	Type	Size
PathFinder	PFDR	Grid Traversal	Dynamic Progr.	400000 elements
Kmeans	KM	Data mining	Dense Algebra	819200points, 34feat.
BF Search	BFS	Graph algorithm	Graph Traversal	10 nodes
Back-Propaga.	BP	Pattern Recog.	Unstructured Grid	524288 input nodes
Hotspot	HS	Physics	Structured Grid	2048*2048 data points
LU Decomp.	LUD	Linear algebra	Dense Algebra	2048*2048 data points
k-Near Neigh.	NN	Data mining	Dense Algebra	64 files
Needle-Wunsch	NW	Bioinformatics	Dynamic Progr.	8192*8192 data points
SRAD	SRAD	Image Process.	Structured Grid	2048*2048 data points
StreamCluster	SC	Data mining	Dense Algebra	65536 data points
CFD Solver	CFD	Fluid dynamics	Unstructured Grid	200K elements
Leukocyte	LC	Medical Imaging	Structured Grid	219*640 pixel, 8frames
Particle Filter	PF	Medical Imaging	Structured Grid	1024*1024 data points
Heartwall	HW	Medical imaging	Structured Grid	609*590 data points
Lava-MD	LMD	Molecular	Structured Grid	10 boxes

Table 2.3: Rodinia Benchmarks

2.4 Performance Measurement

In this work we have used clock_gettime(CLOCK_REALTIME) functionality exposed through the time.h header file to measure the time of execution and compare the performances of different versions of programs. Our time measurement excludes any setup and initial I/O performed by the program. In case of CUDA programs and heterogeneous programs, we also measure the time spent in three separate components of the program using the CUDA profiler [16]: time spent in the GPU kernels, time taken to transfer data between CPU and GPU and the time spent in any sequential computation that is an integral part of the program plus overhead of launching GPU kernels and initiating data transfers.

CHAPTER 3

ACCELERATOR DIRECTIVES

In this section, we describe the directives and attributes provided by the PGI accelerator programming model [10] and the OpenACC standard [17] that help the compilers generate heterogeneous code from sequential programs.

We first describe *compute constructs* that manage parallelism in the device kernel and data constructs that manage data movement between the host and the device. We describe two compute constructs - parallel and loop constructs, and two clauses that can be used with these constructs to control parallelism in the device kernel. Then we explain the data constructs that are used to manage data movement between host and accelerator memories. For each construct, first we describe the OpenACC C/C++ syntax and then we describe the PGI programming model C/C++ syntax. We also use the terms *construct* and *directive* interchangeably.

A parallel construct along with the structured block following it is called a parallel region or an accelerator compute region. A data construct along with the structured block following it is called a data region.

3.1 Parallelism Management

3.1.1 Parallel Construct

The parallel construct is used to indicate that the compiler will attempt to parallelize any loops within the strutured block that follows the parallel construct. The loops are parallelized if they do not carry dependences across iterations, or if the compiler could transform them in a manner that removes the carried dependences. The dependence free iterations are executed on an accelerator device by creating threads and grouping them into thread-blocks for execution on the compute units of the device. If the device is a

NVIDIA GPU, then each streaming multiprocessor of the GPU executes one or more thread blocks simultaneously. In the OpenACC C/C++ standard, this construct typically has the following syntax:

```
#pragma acc parallel [clause [[,] clause]...]
{
structured block
}
```

Depending on the compiler implementation, any computation in the structured block that is not a part of a parallelized *for* loop is performed by one thread in each thread block and the results are shared with the rest of the block, or all the threads replicate the same computation. The compiler ensures that correct device kernel code is generated. A parallel construct along with the structured block following it is called a *parallel region*.

The equivalent construct in PGI accelerator programming model has the following syntax:

```
#pragma acc region [clause [[,] clause]...]
{
structured block
}
```

3.1.2 Loop Construct

The OpenACC loop directive is used inside the parallel region. This construct provides the programmer a fine-grained control of the parallelism generated in the subsequent *for* loop. The programmer can leverage this control by declaring loop-private variables, arrays, reduction operations, etc along with the loop directive. In the OpenACC C/C++ standard, this construct has the following syntax:

```
#pragma acc loop [clause [[,] clause]...]
for () {...}
```

The equivalent construct in the PGI accelerator programming model has the following syntax:

```
#pragma acc for [clause [[,] clause]...]
for () {...}
```

3.1.3 Independent Clause

An important clause used with the loop construct is the independent clause. This clause specifies that the iterations of the following for loop are data-independent and hence the loop can safely be parallelized. It is typically useful in cases where the compiler conservatively assumes that the loop iterations carry dependences, however the computation remains correct even if the dependences are ignored and the loop is parallelized. Such cases may arise when the compiler has limitations in dependence analysis, or multiple iterations write the same value to a variable, or when the runtime value of a variable determines the memory access pattern, etc. The OpenACC syntax for the independent clause is as follows:

```
#pragma acc loop independent
for () {...}
```

The equivalent clause in the PGI accelerator programming model has the following syntax:

```
#pragma acc for independent
for () {...}
```

3.1.4 Collapse Clause

The collapse clause takes a constant positive integer argument. This clause instructs the compiler to fuse together n tightly nested loops that immediately follow this clause, where n is the integer argument provided to the clause by the programmer. This clause is typically useful for increasing the parallelism as the trip count of collapsed set of loops is the product of trip counts of those individual loops. The PGI compiler automatically attempts to collapse any tightly nested loops that are parallelizable, the Cray compiler requires the programmer to specify the collapse clause if the loops are to be collapsed together.

3.1.5 Restrict Attribute

This C99 attribute asserts to the compiler that the arrays declare with this attribute do not overlap and hence updates to one will not affect others. For

instance, arrays A and B declared as *typename restrict *A* and *typename restrict *B* will have non-overlapping memories and updates to one array will not affect the data in the other.

3.2 Data Management

3.2.1 Data Construct

The data construct defines scalars and arrays that will be allocated in the device memory for the scope of the structured block that immediately follows the data construct. The data region can also act as the synchronization point for transferring data between the host and the device, this is achieved using data clauses. In C and C++, the OpenACC data construct has the following syntax:

```
#pragma acc data [clause [[,] clause]...]
{
structured block
}
```

The equivalent construct in the PGI accelerator programming model has the following syntax:

```
#pragma acc data region [clause [[,] clause]...]
{
structured block
}
```

3.2.2 Data Clauses

The data clauses can be combined with compute or data constructs to take a comma-separated list of variable and array names. A copy of these variables or arrays will be allocated and be visible in the device global memory. This is necessary to support the physically separate memory spaces of the accelerator and the host. We have used four OpenACC data clauses in our experiments - *copyin*, *copyout*, *copy* and *create* clauses. The *copyin* clause declares that the variables or arrays in its list have values in the host that need to be copied to the device memory at the beginning of the construct being used,

and these values dont need to be copied back to the host. The *copyout* clause is used to declare the variables or arrays in its list will have values in the device that need to be copied to the host memory at the end of the construct. The *copy* clause declares that the concerned data needs to be copied to the device at the beginning of the construct and copied back to the host at the end of the construct. The *create* clause is used to allocate memory in the device for variables and arrays that dont need values from the host and also any values assigned to them by the device are not needed on the host. Thus the data associated with create clause is not copied between host and device memories.

The PGI accelerator programming model uses the same names and syntax for the *copyin*, *copyout* and *copy* clauses. However, the clause equivalent to the OpenACC *create* clause has the name *local* in the PGI accelerator programming model.

CHAPTER 4

EFFECTIVENESS OF COMPILERS IN ANALYZING DEPENDENCES AND APPLYING LOOP TRANSFORMATIONS

Dependence analysis provides information about the order of computation between loop statements within the same or different iterations. Cross-iteration dependences prevent a loop from being parallelized. There are various loop transformations that can be applied to rearrange the order of computation in the loop to expose parallelism or to optimize the code in other ways while maintaining the correctness.

In this section, we report the results of our analysis to determine the abilities of the PGI and the Cray compilers to automatically apply transformations that remove cross-iteration dependences and expose parallelism or that optimize the availble parallelism. For this analysis, we wrote a set of nine loop nests. In six of the loop nests, application of a single transformation among *alignment*, *distribution*, *fusion*, *interchange*, *privatization*, *reversal* and *skewing* [15] will allow iterations of that loop nest to run in parallel using multiple threads. The other three loop nests perform *reduction*, *prefix scan* and *histogram reduction* each. These computational patterns, in particular *reduction*, are used often among scientific applications and they can be parallelized only by a change of algorithm, however compilers can potentially recognize these patterns and produce the parallelized algorithms automatically. When the compilers dont recognize the computation patterns, the generated code results in a naive GPU implementation that executes serially and provides low performance.

4.1 Loop Transformations

In this section, we describe the basic structure of the loops that we used and the how the application of different loop transformations changes them. We do not describe how these loop transformations preserve the correctness and

the reader is expected to understand dependence analysis and loop transformations. In order to make sure that the performance changes correctly reflect the change in computation pattern and the CPU-GPU data transfer does not become a bottleneck, we added raw flops to the loop iterations so that each loop nest would perform more than 100,000 flops in total.

1. **Loop Alignment**

In the following loop, dependence from *S1* to *S2* due the array A crosses the iterations of the i loop:

```
for (i = 1; i < N; i++){
  A[i] = B[i] + C[i];                    (S1)
  D[i] = A[i-1] * 2.0;                   (S2)
}
```

We can realign the loop to compute and use the values in the same iteration as follows. The modified loop has no carried dependences:

```
D[1] = A[0] * 2.0;
  for (i = 1; i < N-1; i++){
    A[i] = B[i] + C[i];                  (S1)
    D[i+1] = A[i] * 2.0;                 (S2)
  }
A[N-1] = B[N-1] + C[N-1];
```

2. **Loop Distribution**

In the following loop, dependence from *S1* to *S2* due the array A crosses the iterations of the j loop:

```
for (i = 0; i < N; i++){
  for (j = 0; j < N; j++){
    A[i][j] = B[i][j] + C[i][j];         (S1)
    D[i][j] = A[i][j-1] * 2.0;           (S2)
  }
}
```

Loop distribution eliminates loop carried dependences by separating the given loop into multiple loops such that the sources of all dependences execute in one loop before their sinks (if any) execute in a different loop. In the loop above, values were computed in one iteration of a loop and used in a later iteration. After distribution, the values are computed in one loop and used

in an entirely different loop. With loop distribution the dependence is no longer carried by the j loop without any loss of correctness:

```
for (i = 0; i < N; i++){
  for (j = 0; j < N; j++){
    A[i][j] = B[i][j] + C[i][j];                    (S1)
  }
  for (j = 0; j < N; j++){
    D[i][j] = A[i][j-1] * 2.0;                      (S2)
  }
}
```

We can also see that distribution changes some loop carried dependences to loop independent dependences.

3. **Loop Interchange**

In the following example, we can clearly see that the j loop can execute in parallel:

```
for (i = 0; i < N; i++){
  for (j = 0; j < N; j++){  // parallel
    A[i+1][j] = A[i][j] + B[i][j];                  (S1)
  }
}
```

Executing j loop in parallel will lead to additional overhead of N barrier synchronizations at run time one for each iteration of the i loop. Interchanging loops on traditional CPUs can reduce this overhead by spawning and joining threads exactly once:

```
for (j = 0; j < N; j++){  // parallel
  for (i = 0; i < N; i++){
    A[i+1][j] = A[i][j] + B[i][j];                  (S1)
  }
}
```

4. **Loop Privatization**

The following loop has loop carried dependences on the scalar variable t:

```
for (i = 0; i < N; i++){
  t = A[i];                                         (S1)
  A[i] = B[i];                                      (S2)
  B[i] = t;                                         (S3)
}
```

These dependences go away if each loop iteration has its copy of variable t as shown below:

```
for (i = 0; i < N; i++){
  private t;
  t = A[i];                                                    (S1)
  A[i] = B[i];                                                 (S2)
  B[i] = t;                                                    (S3)
}
```

5. **Loop Reversal**

Here the i and j loops carry dependences due to array A and the k loop can execute in parallel.

```
for (i = 1; i < N+1; i++){
  for (j = 1; j < M+1; j++){
    for (k = 0; k < L; k++){
      A[i][j][k] = A[i][j-1][k+1] + A[i-1][j][k+1];            (S1)
    }
  }
}
```

The dependence matrix of the loop nest above is $\begin{pmatrix} = & < & > \\ < & = & > \end{pmatrix}$.

There is an opportunity to improve the parallelism if the k loop could be placed at the outermost position. However, the k loop needs to be reversed before a loop interchange is performed. The new dependence matrix thus becomes $\begin{pmatrix} < & = & < \\ < & < & = \end{pmatrix}$.

Since all dependences are carried by outer loop now, running it sequentially allows the inner loops to execute in parallel. The modified form of the loop nest is as follows:

```
for (k = L-1; k >= 0; k--){
  for (i = 1; i < N+1; i++){
    for (j = 1; j < M+1; j++){
      A[i][j][k] = A[i][j-1][k+1] + A[i-1][j][k+1];            (S1)
    }
  }
}
```

6. **Loop Skewing**

In the following loop nest, clearly no loop is parallelizable.

```
for (i = 2; i < N+1; i++){
  for (j = 2; j < M+1; j++){
    for (k = 1; k < L+1; k++){
      A[i][j][k] = A[i][j-1][k] + A[i-1][j][k];          (S1)
      B[i][j][k+1] = B[i][j][k] + A[i][j][k];            (S2)
     }
  }
}
```

The dependence matrix for the following loop nest is: $\begin{pmatrix} = & < & = \\ \leq & = & = \\ = & = & < \\ = & = & = \end{pmatrix}$.

Loop skewing can change = directions into < on an inner loop. The inner can then be moved to the outermost position and run sequentially, which may make the other loops parallel. Here the innermost loop can be skewed by using $k = k + i + j$. After skewing and an interchange, we have the following dependence matrix: $\begin{pmatrix} < & = & < \\ \leq & < & = \\ = & = & = \end{pmatrix}$.

All the dependences are now carried by the outer loop, thus the inner loops can run in parallel. The modified loop nest is as follows:

```
for (k = 5; k < N+M+L+3; k++){
  for (i = MAX(2,k-M-1-L);
       i < MIN(N+1, k-3)+1; i++){
    for (j = MAX(2,k-i-L);
         j < MIN(M+1,k-i-1)+1; j++){
      A[i][j][k-i-j] = A[i][j-1][k-i-j] + A[i-1][j][k-i-j];   (S1)
      B[i][j][k-i-j+1] = B[i][j][k-i-j] + A[i][j][k-i-j];     (S2)
    }
  }
}
```

In table 4.1, we list the names of transformations in the first column. In the second column, we state state whether the transformation enables a loop nest to efficiently use GPU parallelism. In our working methodology for this section, we used the compilation reports to determine if the compiler has applied the transformations automatically. To make sure that the compilation reports are correct, for each transformation we executed both versions of loop nests described in section 4.1. The first loop nest does not expose parallelism and needs to be transformed, the second loop nest is the transformed version of the first loop nest and is parallelizable. If the compiler was not able to automatically apply the transformation, then the performance of the transformed version of the loops will perform significantly better than the first version. The third column of table 4.1 shows the speedups of the

Transformation	Essential to exploit GPU parallelism	Speedup		Can the compiler automatically apply the transformation?	
		PGI	Cray	PGI	Cray
Alignment	Yes	299.14	33.73	No	No
Distribution	Yes	212.22	15.59	No	No
Interchange	No, threads spawned only once	1.00	1.00	Not needed	
Privatization	Yes	0.99	1.00	Yes	Yes
Reversal	Yes	13.59	30.42	No	No
Skewing	No, skewed iteration space	0.72	0.28	No	No

Table 4.1: Abilities of the PGI and Cray compilers to apply loop transformations to micro-kernels

manually transformed versions of the loop nests over the original loop nests compiled by the same compiler. In the fourth column of table 4.1 we show our conclusions on whether the compilers were able to automatically perform the transformations listed in the first column.

By understanding the loop transformations, we realize that out of all the transformations that expose parallelism in loops, the transformations that enable a loop to make efficient use of GPU parallelism in particular are *alignment*, *distribution*, *privatization* and *reversal*. We will see in sections 5 and 6 that the *distribution* transformation played a vital role in improving the performance of Rodinia benchmarks (though it was not the singularly most important part of the transformation process) and thus it is important that the compilers do a better job at applying distribution transformation automatically. *Fusion*, *skewing* and *interchange* transformations are not helpful in significantly improving performance of loops on a GPU.

The *interchange* transformation helps in improving performance when an outer loop executes sequentially, and in each iteration it spawns and joins multiple threads to execute an inner loop in parallel. In this case, the inner parallel loop can be interchanged with the sequential outer loop as long as no dependences are violated, thereby spawning threads only once. However, there is another way in which threads would be spawned only once and still the interchange transformation would not be needed. This is possible if the iterations of the inner parallel loop are split across multiple threads and the sequential outer loop executes all its iterations in every thread. The compilers that we studied can produce the latter type of code and thus they eliminate the need to implement a proper interchange transformation for loops that run on a GPU.

The *skewing* transformation is not suitable for efficiently using GPU parallelism, because the code that results after applying the transformation ex-

ecutes in a skewed iteration space. This leads to execution of many conditional statements and thereby creates warp divergence which hampers the GPU performance. Also due to the skewed iteration space, many threads in the GPU perform no operations in many iterations. There are many redundant non-coalesced memory accesses too, since the data is stored in row major form and the steps that run in parallel in the skewed iteration space need to access different rows flushing the GPU caches often.

If the transformation is suitable to efficiently using the GPU parallelism, and if the performance of the transformed loop nest was better than the original loop nest then it implies that the compiler was not able to apply the transformation automatically. The compilation reports and the speedups obtained show that among the transformations that were necessary to efficiently use GPU parallelism, the compilers could not automatically apply *alignment*, *distribution* and *reversal* transformations. The only transformation that both PGI and Cray compilers could automatically apply was *privatization*. These results are recorded in table 4.1.

4.2 Computation Patterns

In this section, we show the loop nests that perform *reduction*, *prefix scan* and *histogram reduction* and their parallelized versions. These computational patterns can only be parallelized in a programming language like CUDA or OpenMP that gives the programmer a fine grained control of the code executed by each thread and certain types of memory barriers, and there don't exist any parallelizable sequential equivalent loop nests for these patterns. Table 4.2 shows the compilers were able to automatically parallelize the computational patterns. The fact that the compilers were not able to parallelize *prefix scan* and *histogram reduction* automatically indicates a need for better recognition of computational patterns or finer control over parallelism that would benefit programmers if the automatic recognition and transformation of patterns does not succeed.

1. **Reduction**

Reduction is an operation, such as addition, which is applied to the elements of an array to produce a result of a lesser rank. A typical sum reduction loop

Computation Pattern	Can the compiler automatically parallelize the loop nest?	
	PGI	Cray
Reduction	Yes	Yes
Prefix Scan	No	No
Histogram Reduction	No	No

Table 4.2: Abilities of the PGI and Cray compilers to automatically parallelize computation patterns

looks like the following:

```
for (i = 0; i < N; i++){
  sum = sum + A[i];
}
```

Reduction is a special computation pattern and a CUDA programmer can write an efficient device kernel to compute reduction in parallel [18]. Similarly, a heterogeneous compiler could generate a parallel device kernel for reduction.

2. **Prefix Scan**

A typical inclusive prefix scan loop looks like the following:

```
A[0] = B[0];
for (i = 1; i < N; i++){
  A[i] = A[i-1] + B[i];
}
```

Just like reduction, a heterogeneous compiler could potentially recognize inclusive and exclusive prefix scan operations and generate an efficient parallel kernel [19].

3. **Histogram Reduction**

A histogram reduction loop looks like the following:

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

Histogram reduction is tricky to parallelize because of the index values that are determined at runtime. An efficient parallel kernel for histogram reduction involves atomic operations and might need parameter tuning to suit the given device [20].

CHAPTER 5

TRANSFORMATION OF OPENMP PROGRAMS TO HETEROGENEOUS PROGRAMS

In this section, we describe the transformations that we performed to convert the OpenMP code to a format that the PGI and Cray heterogeneous compilers could understand and compile to produce device kernel code, the reasons for introducing each transformation, the effect of each transformation on the overall performance and the time spent in different parts of the program kernel. The record of whether a transformation was applied or not is presented in tables 5.1 and 5.2. At every step, we check for the compiler errors and performance issues, and we accordingly introduce the required transformation. Sometimes the application of a transformation is necessary for more than one reason. In such cases, the reasons for applying these transformations are provided in the tables. The last column of both tables provides the total number of benchmarks that required application of the corresponding transformation, and the bottom row of the tables provides the total number of transformations applied for each benchmark. In tables 5.3 and 5.4, we note the effect that each of these transformations has on performance. In figure 5.1, we provide breakdown of the time spent by the benchmarks into the device kernel execution, CPU-GPU data communication and the sequential part of the program kernel that is executed on the CPU in addition to the overhead of launching device kernels, for the last few steps of the transformation process that produced the most significant performance difference.

5.1 Transformation Steps

The transformations were applied in the following order:

T1 Convert the program to C99 from C++, if using the PGI compiler. The version of PGI compiler used for this study did not allow

any C++ code in the parallel regions. This also required extra work to translate any C++ STL invocations used inside parallel regions to their equivalent implementation in C.

T2 Insert *parallel regions*. Replace an OpenMP parallel region syntax with a PGI or Cray parallel region syntax. The compiler will attempt to parallelize loops within the parallel region by offloading them to the GPU. The loops are parallelized if they do not carry dependences across iterations.

T3 Insert *restrict clauses*, if using the PGI compiler. PGI compiler requires the programmer to use restrict attribute to indicate that arrays will have non-overlapping memories and updates to one array will not affect the data in the other. This transformation is not necessary when using the Cray compiler.

T4 Convert any multi-dimensional arrays that index their sub-arrays with dynamically allocated pointers to an array stored in contiguous memory. This new array could then be addressed in a single or multiple dimensions. Given a starting memory address, the compilers generate instructions that perform any data transfer between the CPU and the GPU for a fixed number of contiguous memory locations. Therefore multi-dimensional arrays with non-contiguous memory will produce incorrect results.

T5 Remove any arrays/pointers from structures/unions. Instead of using a union, allocate separate space to each of its data members. If a structure/union contains another structure/union, perform this step recursively. This transformation is necessary because the compiler versions that we used for experiments could not de-reference the arrays contained in a structure or union inside a parallel region. The compilers could also not correctly allocate space on the GPU to unions that were used inside the parallel regions.

T6 Inline procedures. The PGI compiler required all the procedures inside a parallel region to be manually inlined. The Cray compiler could inline procedures with primitive data type arguments.

T7 Insert *data clauses.* Although both compilers are able to automatically generate a CPU-GPU memory copy command to transfer data back and forth between the CPU and the GPU, they are able to do so only when the size of the array is already known at the compilation time. However most of the arrays in practical programs are dynamically allocated. The PGI compiler can determine the size of data to be transferred on the PCIE bus if the array index calculations in the loops inside the parallel region are simple enough. On the other hand, the Cray compiler makes the use of data clauses mandatory to transfer dynamically allocated arrays. When the compilers are not able to automatically determine the size of the array to be copied to/from the GPU, then the programer has to manually specify the size of the array in the data clauses.

T8 Use independent clause in the following cases to execute the loop iterations in parallel:

a. The loop iterations have an output dependence, but parallelizing the loop results only in a benign data race in which threads write the same value to a given memory location.

b. The compiler detects false dependences between loop iterations due to array index calculations that involve runtime variables, the values of which the programmer knows or can deduce but the compiler cannot.

c. Due to limitations in the internal implementation, the compiler cannot analyze the array index calculations to deduce that the loop is parallelizable, even though there is no theoretical reason why the compiler should not deduce so. In such cases, the compiler may conservatively decide to not parallelize the loop.

d. The Cray compiler generated incorrect GPU code due to a bug when it tried to automatically parallelize the outer loop of a loosely coupled multi-loop nest. A workaround was to generate the correct code was to use the independent clause above the outer loop. This bug will likely go away in the coming releases of the compiler.

T9 Insert *data regions.* Using data region helps to avoid repetitive data movement between the CPU and the GPU which often forms the per-

formance bottleneck for heterogeneous programs. Programmers can specify the variables or arrays to be copied into the GPU memory at the entry point of the data region and the variables to be copied out at the exit point of the data region. These variables or arrays are maintained in the GPU for the scope of the data region. A typical optimization achieved by a data region involves enclosing a loop that executes a parallel region multiple times. Instead of transferring data back and forth between CPU and GPU for every execution of the parallel region, the data region performs the transfer only once and keeps the data alive in the GPU for the scope of the data region.

T10 Change the size and/or the number of parallel regions as compared to the OpenMP source code in the following cases:

a. Distribute a parallel loop over any inner loops if the inner loops are parallel and contain sufficient amount of computation. The tightly coupled loop nests formed in this manner have large effective trip counts. Thus they provide an opportunity for creating large number of threads, and become a good candidate for GPU computation. Arrange data clauses accordingly. In case of parallel regions that consist of outer parallel loops using a procedure repeatedly that in turn contains more parallelizable loops, procedure inlining abilities by themselves are not enough to obtain good performance if the compiler could not perform loop distribution over long loop bodies. In this case, after the procedures have been inlined, this transformation proves particularly useful to form parallelizable loops with large trip counts.

b. Remove the reduction from an parallel region to perform it on the CPU or in a separate parallel region. Separating the reduction out makes it easier for the compiler to optimize the remaining computation. The reduction can be performed in a separate parallel region or on the CPU depending on the amount of computation and the amount of CPU-GPU data communication involved.

c. Change the boundaries and/or the number of parallel regions. Depending on the availability of any parallelizable loops, we can expand the parallel regions to absorb such additional loops and/or

generate new parallel regions for those parallelizable loops. Similarly, we should de-parallelize a loop that has a very small amount of computation and requires a large amount of data transfer between the CPU and the GPU and execute it on the CPU. Accordingly the data movement should be optimized using data regions.

d. Change loops that are parallelized while maintaining the same number of parallel regions. This is useful to parallelize the loops with larger trip counts or the loops that provide coalesced memory accesses. This is achieved by interchanging loops or by merely changing the position of parallel directive if the loops are loosely nested and interchanging those loops is not possible without substantial manual effort.

T11 Use collapse clause. PGI compiler can collapse tightly nested loops together into a single loop. The trip counts of the loops in the nest are multiplied together to constitute the effective trip count of the resulting collapsed loop. The collapsed loop can then be parallelized to generate large amount of parallelism. Cray compiler does not collapse loops by default, and it requires the use of a collapse clause. Using collapse clause for a nest of parallelizable loops leads to an increase in multiprocessor occupancy (which we have explained in section 6.3.2.1).

We can observe that most benchmarks required 9 or 10 of the transformations to produce a reasonable performance. Depending on the OpenMP source code, all these transformations require varying amount of work from the programmer.

Using the PGI compiler demands laborious work from the programmer in transformation T6, where the programmer needs to inline all the functions that are used inside the parallel region. This requires substantial manual efforts which could take similar or even more time as compared to performance tuning in transformation T10. We inlined functions used inside the parallel regions in case of 9 benchmarks while using the PGI compiler. On the other hand while using the Cray compiler, we were required to inline functions in transformation T6 only in k-Nearest Neighbours, and these procedures were library procedures. Thus the Cray compiler seems to have better function inlining abilities. Cray compiler adopts a policy of making the use of data clauses mandatory whenever there is a data transfer involved between the

T	PFDR		KM		BFS		BP		HS		LUD		NN		NW		SRAD		SC		CFD		LC		PF		HW		LMD		Total
	A	R	A	R	A	R	A	R	A	R	A	R	A	R	A	R	A	R	A	R	A	R	A	R	A	R	A	R	A	R	A
T1	✔	S	-	-	✔	S	-	-	-	-	-	-	-	-	✔	S	-	-	✔	S	✔	S	-	-	-	-	-	-	-	-	5
T2	✔	S	✔	S	✔	S	✔	S	✔	S	✔	S	✔	S	✔	S	✔	S	✔	S	✔	S	✔	S	✔	S	✔	S	✔	S	15
T3	✔	S	✔	S	✔	S	✔	S	✔	S	✔	S	✔	S	✔	S	✔	S	✔	S	✔	S	✔	S	✔	S	✔	S	✔	S	15
T4	✔	S	✔	S	-	-	✔	S	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	3
T5	-	-	-	-	-	-	✔	S	-	-	-	-	-	-	-	-	-	-	✔	S	✔	S	✔	S	-	-	✔	S	✔	S	6
T6	-	-	✔	S	-	-	✔	S	-	-	-	-	✔	S	✔	S	-	-	✔	S	✔	S	✔	S	✔	S	✔	S	-	-	9
T7	-	-	✔	CF	✔	S	✔	S	✔	S	✔	S	-	-	✔	S	✔	S	✔	S	✔	S	✔	S	✔	S	✔	CF	✔	CF	13
T8.a	-	-	-	-	✔	S	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1
T8.b	-	-	-	-	-	-	-	-	-	-	-	-	-	-	✔	S	✔	S	-	-	-	-	-	-	-	-	-	-	-	-	2
T8.c	-	-	✔	S	-	-	✔	S	✔	S	✔	S	-	-	-	-	-	-	-	-	✔	S	✔	S	✔	S	✔	CF	-	-	8
T8.d	-	-	-	-	-	-	-	S	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S	-	-	-	-	-	-	0
T9	✔	S	✔	S	✔	S	✔	S	✔	S	✔	S	-	-	✔	S	✔	S	✔	S	✔	S	✔	S	✔	S	✔	CF	✔	S	14
T10.a	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	✔	S	-	-	✔	S	-	-	2
T10.b	-	-	✔	S	-	-	✔	S	-	-	-	-	-	-	-	-	-	-	✔	S	-	-	-	-	-	-	-	-	-	-	3
T10.c	-	-	-	-	✔	S	✔	S	-	-	-	-	-	-	-	-	✔	S	-	-	-	-	-	-	-	-	-	-	-	-	3
T10.d	-	-	-	-	-	-	✔	S	-	-	-	-	-	-	-	-	-	-	-	-	-	-	✔	S	-	-	-	-	✔	S	3
Total	5		8		7		9		5		5		3		7		6		8		8		8		6		8		6		

Legend:

Sub-column	*Symbol*	*Meaning*
Sub-column A	✔	*The corresponding transformation was applied.*
Sub-column A	-	*The corresponding transformation was not applied.*
Sub-column R	*CF*	*Compilation did not finish successfully, i.e., the compiler crashed while compiling the program.*
Sub-column R	*S*	*The compilation completes successfully.*
Sub-column R	-	*The corresponding transformation was not applied.*

Table 5.1: Transformations applied while using the PGI compiler

Step	PFDR		KM		BFS		BP		HS		LUD		NN		NW		SRAD		SC		CFD		LC		PF		HW		LMD		Total
	A	R	A	R	A	R	A	R	A	R	A	R	A	R	A	R	A	R	A	R	A	R	A	R	A	R	A	R	A	R	A
T1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0
T2	✔	S	✔	S	✔	S	✔	S	✔	S	✔	S	✔	S	✔	S	✔	S	✔	S	✔	S	✔	S	✔	S	✔	S	✔	S	15
T3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0
T4	✔	S	✔	S	-	-	✔	S	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	3
T5	-	-	-	-	-	-	✔	S	-	-	-	-	-	-	-	-	-	-	✔	S	-	-	✔	S	-	-	✔	S	✔	S	5
T6	-	-	-	-	-	-	-	-	-	-	-	-	✔	S	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1
T7	✔	S	✔	CF	✔	S	✔	S	✔	S	✔	CF	-	-	✔	S	✔	S	✔	CF	✔	S	✔	CF	✔	CF	✔	CF	✔	CF	14
T8.a	-	-	-	-	✔	S	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1
T8.b	-	-	-	-	-	-	-	-	-	-	-	-	-	-	✔	S	✔	S	-	-	-	-	-	-	-	-	-	-	-	-	2
T8.c	-	-	-	-	-	-	-	S	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S	-	-	-	-	-	-	0
T8.d	-	-	✔	S	-	-	✔	S	-	-	✔	S	-	-	-	-	-	-	✔	CF	-	-	✔	CF	✔	CF	✔	CF	-	-	7
T9	✔	S	✔	S	✔	S	✔	S	✔	S	✔	S	-	-	✔	S	✔	S	✔	CF	✔	S	✔	CF	✔	CF	✔	CF	✔	CF	14
T10.a	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	✔	CF	-	-	✔	CF	-	-	2
T10.b	-	-	✔	S	-	-	✔	S	-	-	-	-	-	-	-	-	-	-	✔	S	-	-	-	-	-	-	-	-	-	-	3
T10.c	-	-	-	-	✔	S	✔	S	-	-	-	-	-	-	-	-	✔	S	-	-	-	-	-	-	-	-	-	-	-	-	3
T10.d	-	-	-	-	-	-	✔	S	-	-	-	-	-	-	-	-	-	-	-	-	-	-	✔	CF	-	-	-	-	✔	CF	3
T11	-	-	-	-	-	-	-	-	✔	S	-	-	-	-	-	-	✔	S	-	-	-	-	✔	CF	✔	CF	✔	CF	✔	CF	6
Total	4		6		5		7		4		4		2		4		5		6		3		7		5		7		6	-	-

Legend:

Sub-column	*Symbol*	*Meaning*
Sub-column A	✔	*The corresponding transformation was applied.*
Sub-column A	-	*The corresponding transformation was not applied.*
Sub-column R	*CF*	*Compilation did not finish successfully, i.e., the compiler crashed while compiling the program.*
Sub-column R	*S*	*The compilation completes successfully.*
Sub-column R	-	*The corresponding transformation was not applied.*

Table 5.2: Transformations applied while using the Cray compiler

CPU and the GPU. Our experience with transforming the Rodinia benchmarks has shown us that this policy has a good effect on performance in that it forces the programmer to explicitly think about the amount of data transfer and optimize any obvious cyclic data transfer.

The use of T8.a and T8.b is exhibited in the Breadth-First Search and the Needleman-Wunsch, SRAD benchmarks respectively. However we had to use this clause also for the cases where the PGI compiler could not simply analyze the array index calculations and the Cray compiler generated incorrect GPU code due to a bug with parallelizing more than a singly nested loop, which are the cases T8.c and T8.d respectively. Using independent clause and forcing the parallelization was a workaround in both these situations, and should not be required if the compilers performed correct analysis.

Transformation T10 is the most crucial transformation in obtaining a good performance. This step can require substantial trial-and-error work from the programmer in deciding how the boundaries of the parallel regions are to be adjusted. This transformation is applied in 8 Rodinia benchmarks.

5.2 Impact of Transformations on Performance

In this section, we will show the impact of each transformation step on the performance of heterogeneous programs. In tables 5.3 and 5.4, we show the relative performance impact of transformations mentioned in section 5.1. We recorded the relative performance only after applying transformation T7, because all the parallel regions execute in the GPU only after the application of transformation T7, that is, after the addition of data clauses. The value recorded for transformation Tn for a given benchmark is a ratio of the time taken after the application of Tn divided by the best time recorded for the benchmark at any step of the transformation process. Thus, we have essentially recorded the relative performance obtained after the application of different transformations, with a value of 1 indicating the fastest performance and a value greater than 1 indicating performance slowdown. An important point to note here is that the Cray compiler is still under development and some benchmarks (ParticleFilter, Leukocyte, Particlefilter, LavaMD and Streamcluster after the application of certain transformations) could not be compiled correctly due to internal compiler bugs. This has been indicated in

Transformation	PFDR	KM	BFS	BP	HS	LUD	NN	NW	SRAD	SC	CFD	LC	PF	HW	LMD
T10	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
T9	1.0	100.5	2.5	1.0	1.0	1.0	1.0	1.0	2.7	∞	1.0	∞	1.0	∞	∞
T8	5.3	102.4	6.7	5.0	13.7	537.1	1.0	∞	10.6	∞	14.2	∞	3.3	∞	∞
T7	5.3	∞	128.3	59.4	∞	5859.7	1.0	∞	858.6	∞	14.2	∞	3.6	∞	∞

Table 5.3: Relative times taken by the executions of heterogeneous programs compiled with the PGI compiler.

Transformation	PFDR	KM	BFS	BP	HS	LUD	NN	NW	SRAD	SC	CFD	LC	PF	HW	LMD
T11	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	CF	CF	CF	CF
T10	1.0	1.0	1.0	1.0	4.5	1.0	1.0	1.0	4.7	1.0	1.0	CF	CF	CF	CF
T9	1.0	∞	2.7	1.4	4.5	1.0	1.0	1.0	6.7	CF	1.0	CF	CF	CF	CF
T8	350.8	∞	7.5	4.3	34.1	21.1	1.0	∞	19.2	CF	50.6	CF	CF	CF	CF
T7	350.4	∞	170.2	11.2	34.1	21.1	1.0	∞	26.3	CF	50.6	CF	CF	CF	CF

CF indicates that compilation did not finish successfully, i.e., the compiler crashed while compiling the program.

Table 5.4: Relative times taken by the executions of heterogeneous programs compiled with the Cray compiler

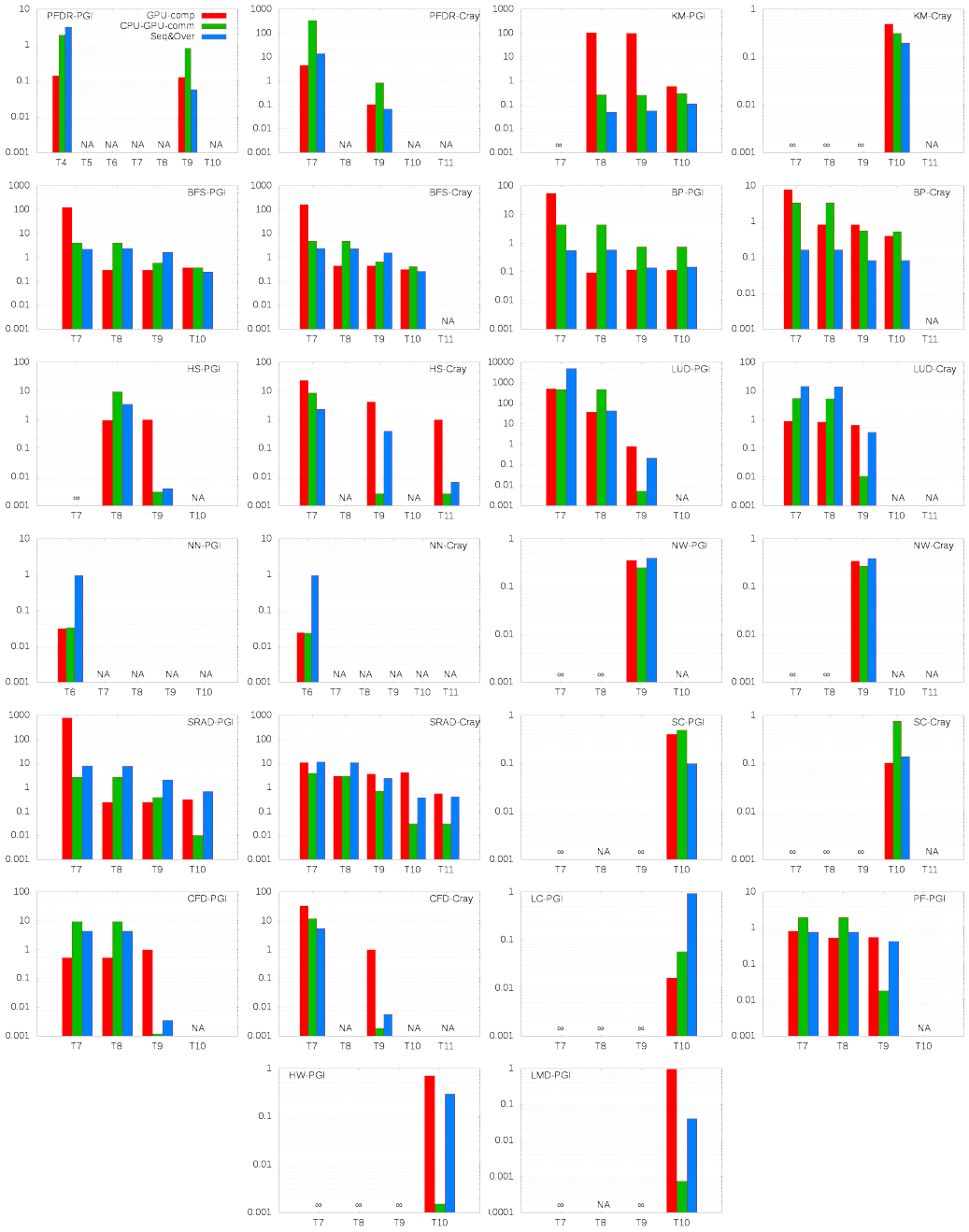
tables with CF.

The tables clearly show that the best performance is obtained after the application of T10 for heterogeneous programs compiled with the PGI compiler, and after the application of T11 for the for heterogeneous programs compiled with the Cray compiler. In general, the performance improves as we apply transformations from T7 to T11. As can be seen from the tables, all steps are critical for performance improvement and we cannot point out any one particular transformation as the most important.

In figure 5.1, we plot the breakdown of the program execution time into its three components - device kernel execution time in the GPU, time of CPU-GPU data communication, and the time of sequential computation and any overhead of launching the device commands over the PCIE bus. This information was obtained using NVIDIAs CUDA Profiler [12]. In each subfigure, the transformation steps are listed on the X axis and the bars for the three components of each program are plotted such that the sum of these three components (which equals to the overall program time) is normalized to 1.

The impact on each transformation on individual components of the overall program time can be seen from figure 5.1:

1. The application of transformation T8 (using independent clause) should reduce the time spent in device kernel execution. Indeed for every benchmark on which T8 was applied (a ✔was recorded in the row corresponding to T8 in tables 5.1 and 5.2), the magnitude of the bars



Legend:	
GPU-comp	*Time taken by device kernel execution*
CPU-GPU-comm	*Time required for CPU-GPU data transfer*
Seq&Over	*Time taken by sequential computation on the CPU plus any overhead of launching GPU commands over the PCIE bus*
NA	*The transformation was not applied*
∞	*The program executed for a very long time*

Figure 5.1: Distribution of time spent by the heterogeneous programs after the application of transformations

correspoding to device kernel execution time reduces from T7 to T8 in figure 5.1.

2. The application of T9 (using data region) should reduce the time spent in host-device data communication. For every benchmark on which T9 was applied (a ✔was recorded in the row corresponding to T9 in tables 5.1 and 5.2), the magnitude of the bars correspoding to host-device data communication time reduces from T8 to T9 in figure 5.1. The introduction of data region reduces the number of distinct data transfers between the CPU and the GPU, hence the application of T9 also reduces the overhead of launching data transfer commands, this is also clearly seen in figure 5.1.

3. The application of T10 (expand or contract the boundaries of parallel regions and data regions, or change their numbers) has an effect on all three components. The magnitude of these components can increase or decrease depending on which combination among T10.a, T10.b, T10.c and T10.d is applied and how effective each of these transformation is for that program:

   (a) T10.a generally reduces the amount of computation performed in the GPU by moving the less parallel code to the CPU. It has the side effect of increasing the sequential computation time on the CPU. The data transfer time can remain the same if the data transfer over multiple parallel regions is optimized using a data region, or it can increase if the creation of new parallel regions leads to more data transfer.

   (b) T10.b can make a reduction operation more efficient on the GPU and reduce the device kernel time. Or it can reduce the amount of computation performed in the GPU by moving a reduction operation from the GPU to the CPU, thereby again reducing the device kernel time, increasing the sequential execution time on the CPU and potentially reducing the data transfer time.

   (c) Quite clearly, T10.c can increase or decrease the times of device kernel execution, sequential computation on the CPU and host-GPU data transfer.

   (d) By changing the loops that are parallelized, T10.d can also increase or decrease the times of device kernel execution, sequential computation on the CPU and host-GPU data transfer.

Figure 5.1 shows the results of the application of T10 with the change in magnitudes of all three columns from T9 to T10. The combined effect of all of T10.a, T10.b, T10.c and T10.d is to reduce the total program kernel time as seen in tables 5.3 and 5.4.

4. The application of T11 should reduce the time spent in device kernel execution. For the Cray accelerated versions that apply T11, the device kernel execution time indeed reduces from T10 to T11 as seen in figure 5.1.

In summary, transformation T8 reduces device kernel execution time, T9 reduces the time spent in host-device data communication and the overhead of launching data communication commands, T10 can increase or decrease all three components of the program, and T11 reduces the time spent in device kernel execution.

Similar effects are observed for all the benchmarks with the application of different transformation steps. It should be noted that as we move from T7 to T11, the total time of a benchmark's execution reduces if the corresponding cell in the tables 5.1 and 5.2 has a ✔and the total time remains the same if the corresponding cell in the tables 5.1 and 5.2 has a -, and the magnitude of this change in total time has been demonstrated in tables 5.3 and 5.4.

CHAPTER 6

PERFORMANCE OF RODINIA BENCHMARKS

In the second set of experiments, we evaluated the performance of the Rodinia suite of benchmarks compiled by both the PGI and Cray compilers. For each one of the 15 benchmarks that we used, the Rodinia suite provides two different versions of the code - an OpenMP version and a CUDA version. We have transformed these different benchmarks using the transformation steps mentioned in section 5 and compiled them using the PGI and the Cray compilers for performance analysis. Since ParticleFilter, Leukocyte, Particlefilter and LavaMD could not be compiled by the Cray compiler due to bugs, the corresponding spaces in the figures have been left blank.

We note the performance by executing the benchmarks on a single machine and recording the time taken for the core computation of each benchmark. This core computation does not include initial setup and I/O with the disk. We recorded the time of execution of each benchmark. For the CUDA and heterogeneous versions, this includes time taken by the GPU computation, time required for the CPU-GPU data communication, and time taken by any sequential execution on the CPU that is part of the program and any overhead of launching kernels to the GPU. The performance measurement for sequential and OpenMP versions consists of majoring the time required to execute their program kernel on the CPU.

This section is divided into 3 sections:

1. First we show that the CPU programs compiled by the Cray compiler perform faster than the CPU programs compiled by the PGI compiler. We also discuss the possible reasons behind this performance difference.

2. Then we compare the performance of heterogeneous programs with the peformance of their OpenMP and CUDA versions. We also determine the reasons behind the slow performance produced by the heterogeneous versions of certain benchmarks. These observations provide us

some important lessons about the characteristics that are desirable from heterogeneous programs.

3. We note the difference in performance of the device kernels of heterogeneous programs and try to find the reasons behind this difference using metrics like multiprocessor occupancy, registers count, shared memory use, instructions count, etc. However, our analysis shows that the performance of device kernels is a complex function of various parameters and it is difficult to summarize the compilers strengths and weaknesses in a simple formula.

6.1 Performance Comparison of Sequential Programs compiled by the PGI and Cray Compilers and Performance Comparison of OpenMP Programs compiled by the PGI and Cray Compilers

6.1.1 Observations

We first compare the performance of the sequential versions of each benchmark compiled using the PGI and Cray compiler. The benchmarks are listed on X axis, and on the Y axis we plot the speedup of the performance obtained when the Cray compiler is used over the performance obtained when the PGI compiler is used. From figure 6.1, we can see that the programs perform better when they are compiled with the Cray compiler, except in the case of Kmeans and LavaMD.

6.1.2 Performance Analysis

We discuss two potential reasons to understand the performance difference between the executables produced by the two compilers.

First, the executables produced by the PGI compiler displayed the problem of allocating conflicting addresses to arrays. This means that the memory allocation library used by the PGI compiler provides array start addresses that use the same L1 data cache line. If the start addresses of arrays con-

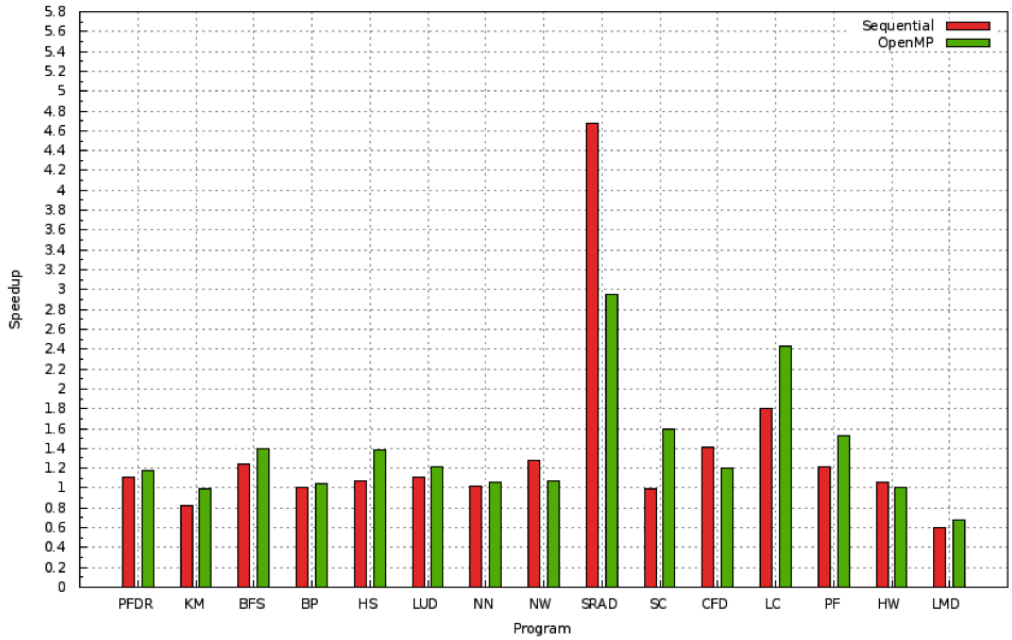


Figure 6.1: Speedups of sequential programs compiled with the Cray compiler over their own performance when compiled with the PGI compiler and speedups of OpenMP programs compiled with the Cray compiler over their own performance when compiled with the PGI compiler

flict, then it is highly likely that the addresses of multiple or all elements of those arrays will conflict. Loops very often compute the same index for addressing elements of multiple arrays in the same iteration. If these arrays elements have conflicting addresses, all of their data clearly cannot remain in the L1 cache in the same loop iteration. This causes the cache to flush more frequently and thus it reduces the performance of the executables produced by the PGI compiler. This issue could be addressed by adding some offset padding to the start addresses of arrays which would force elements of different arrays to use different L1 cache lines in the same loop iteration. We performed experiments in which we padded offsets at the start addresses of arrays, this increased the performance of the executable produced by the PGI compiler significantly. This verified that the policy used by the PGI compiler to allocate addresses to arrays reduced performance.

Second, the Cray compiler seems to be able to produce more vector instructions for most benchmarks. Since vector instructions can perform more work in data parallel fashion, a higher percentage of vector instructions should lead to performance speedup, and hence the number of vector instructions could

Benchmark	Percentage of vector instructions			
	PGI-seq	PGI-OpenMP	Cray-seq	Cray-OpenMP
PFDR	1.25	1.25	0.65	0.65
KM	3.79	3.79	10.3	5.16
BFS	1.72	1.72	0.6	0.6
BP	5.37	5.3	9.86	9.86
HS	0.58	0.58	4.73	2.61
LUD	4.92	4.92	18.3	18.4
NN	4.6	4.6	6.27	6.27
NW	0.74	0.74	0.31	0.31
SRAD	2.57	2.5	2.17	2.17
SC	4.43	4.43	10.3	10.2
CFD	0.29	0.29	3.55	3.88
LC	5.39	5.39	12.4	11.3
PF	8.4	3.68	20.9	20.9
HW	3.97	4.08	8.7	8.7
LMD	2.94	2.94	2.97	3.58

Table 6.1: Percentage of vector instructions produced by PGI and Cray compilers in sequential and OpenMP benchmarks

potentially be one of the factors that influence sequential performance. However we can see from table 6.1 that the higher percentage of vector instructions does not necessarily correlate to a better performance for all programs, therefore we can assume that there are other factors that can influence the performance more than vector instructions do.

In particular, the high speedup obtained for SRAD could be attributed to the issue of conflicting array addresses since this issue was highly pronounced in the case of SRAD, and padding offsets at the start addresses of arrays increased the performance of the executable produced by the PGI compiler significantly. The reduced performance of Kmeans and LavaMD when compiled with the Cray compilee can perhaps be attributed to loop unrolling and spilling of register file. However as discussed earlier, CPU performance is a result of multiple factors combined, and the analysis of these factors is out of scope of our work.

6.2 Performance Comparison of Heterogeneous Programs and their OpenMP Versions

6.2.1 Observations

We now evaluate the performance of each benchmarks OpenMP, CUDA and heterogeneous versions against the performance of its sequential version. This

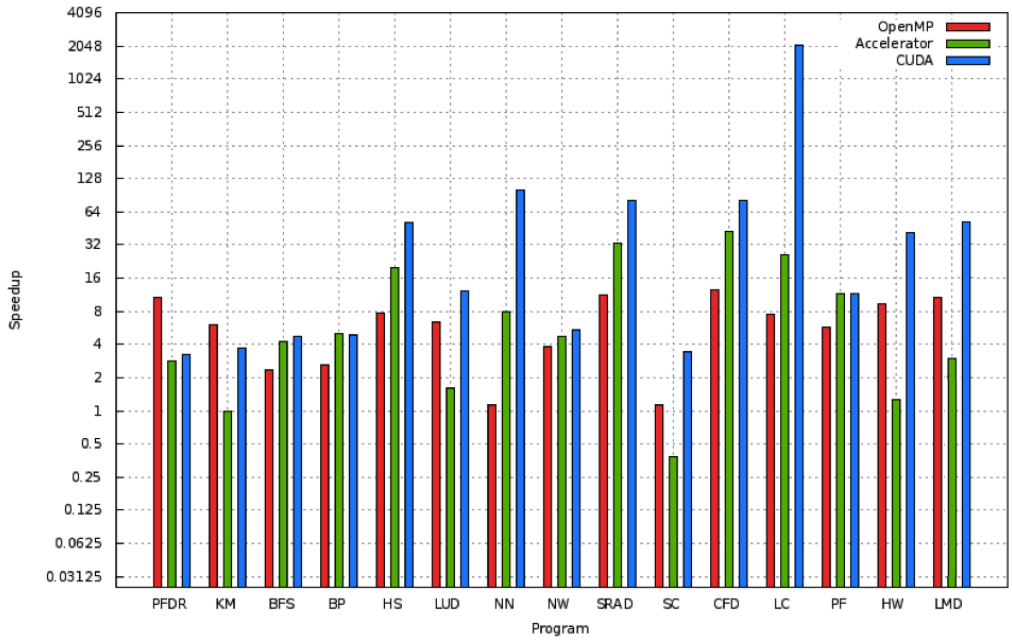


Figure 6.2: Speedups of OpenMP, heterogeneous and CUDA versions of Rodinia benchmarks over sequential versions, all compiled by the PGI compiler

will give a comparative perspective of the speedups that could be obtained using different programming techniques. The heterogeneous program versions have been produced by applying transformations to their OpenMP versions, thus this comparison will also provide us an idea about the tradeoff between obtaining performance improvement using the heterogeneous compilers versus programming using CUDA. The OpenMP versions executed 16 software threads in parallel, in case of heterogeneous programs the number of GPU threads launched in parallel was completely determined by the PGI and Cray compilers. The results are shown in figures 6.2 and 6.3.

6.2.2 Performance Analysis

Heterogeneous versions have produced good speedup over their OpenMP versions in the case of many benchmarks. However there are certain benchmarks whose heterogeneous versions dont perform as well. We discuss the reasons behind the slow performance and derive some important lessons.

1. Inefficient heterogeneous programs can perform slower than their OpenMP versions. Both the CUDA and heterogeneous versions of PathFinder and Kmeans perform slower than their OpenMP versions. This

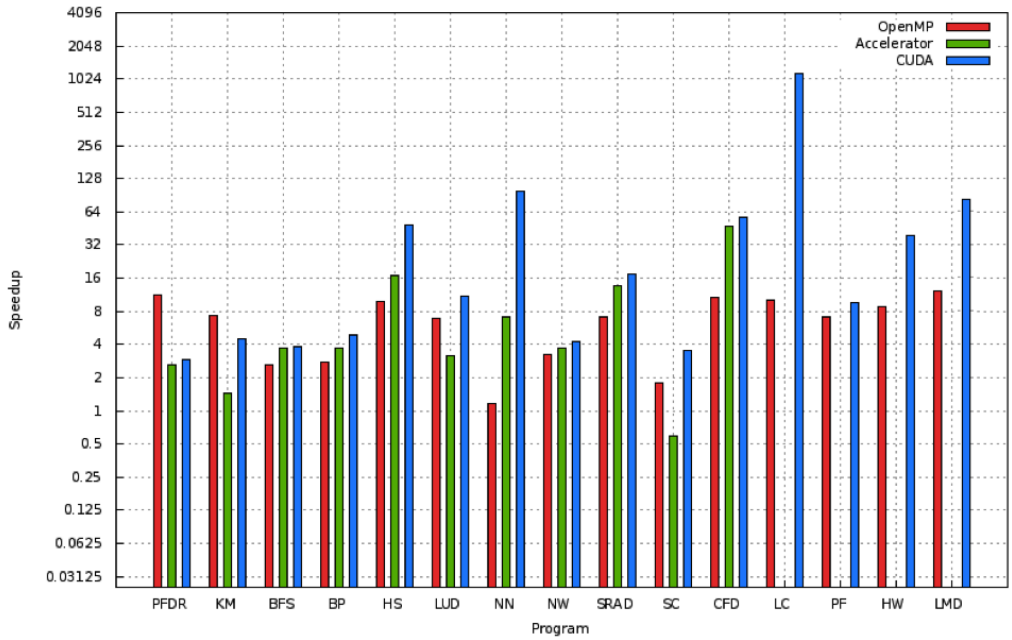


Figure 6.3: Speedups of OpenMP, heterogeneous and CUDA versions of Rodinia benchmarks over sequential versions, all compiled by the Cray compiler

can be explained by observing that in case of PathFinder, the CPU-GPU data communication itself in both the CUDA and heterogeneous versions took more time than the OpenMP versions whole kernel due to large size of the data being transferred. The heterogeneous versions of Kmeans suffer from both data transfer and non-coalesced memory accesses in the GPU kernels, hence they produce slow performance.

The heterogeneous versions of Streamcluster, LUD and LavaMD perform slower than their OpenMP versions, though their CUDA performance performs well. In the case of Streamcluster, the CPU-GPU data communication itself in the heterogeneous version takes about 2.3X more time as compared to the whole program in the OpenMP version. This happens due to repeated device kernel calls and cyclic data communication between CPU and GPU. LUD involves triangular matrix operations and has many row-wise and column-wise dependences [14]. The heterogeneous version makes a lot of non-coalesced accesses to the device memory since the device cache or shared memory is not big enough to hold the entire matrix involved in the computation, which leads to reduced performance in GPU computation. The heterogeneous version of LavaMD performs very little computation for the number of device global memory accesses it performs, which results in bad

performance.

Similar results are observed with Heartwall heterogeneous version compiled by the PGI compiler. The heterogeneous version of Heartwall spends about 0.7 second in one particular device kernel execution, which is almost 80% of the total time spent taken by the OpenMP version in the entire program. The device kernel under consideration is executed 8 times. In this kernel, each thread loads data from memory in a loop which can potentially cause a lot of non-coalesced global memory loads. The number of misses in L2 cache of the GPU in the heterogeneous version are also about 48X as compared to the CUDA version. We believe these two factors combine to make the heterogeneous version of Heartwall quite inefficient. These two factors also explain the relatively slow performance of Heartwalls heterogeneous version as compared to its CUDA version.

These observations provide us some important lessons about the characteristics that are desirable from heterogeneous programs. The observations with PathFinder, Kmeans and Streamcluster show us that to exploit GPU parallelism efficiently, it is extremely important to be able to parallelize enough part of the code and the code should perform enough number of operations in the GPU for every data byte transferred, so that the sequential part and the data transfer respectively does not become a bottleneck to performance. The performance of LavaMD heterogeneous version shows the importance of using shared memory effectively to cache data and to avoid global memory accesses in the GPU. Heartwall and LUD demonstrate the need to rearrange loops so as to avoid non-coalesced memory accesses.

Inefficient heterogeneous programs may not also produce significant performance benefit over their sequential versions. The heterogeneous versions of 3 benchmarks - Kmeans, Heartwall and Streamcluster - could not produce even a speedup of 2 over their sequential version. Out of these, two benchmarks have not been correctly compiled by the Cray compiler.

2. Providing explicit control over optimizations is important for performance tuning. CUDA gives programmers an explicit control over performing micro-optimizations. CUDA versions of many Rodinia benchmarks have been optimized to make judicious use of shared memory and reduce the performance penalty of device global memory accesses. However, the heterogeneous compilers perform such micro-optimizations during the

compilation, and these optimizations are hidden from the programmer. As can be seen from our results, heterogeneous compilers may not always be able to match the performance of CUDA. In such cases, providing clauses and directives that allow the programmer to fine-tune various micro-optimizations could help the programmer to boost the program performance while benefiting from the performance portability of these optimization clauses.

CUDA also allows the programmer to change the program algorithm. In a number of Rodinia benchmarks, the program algorithm has been changed from the OpenMP version to the CUDA version to rearrange loops, increase parallelism and reduce the overhead of runtime device kernel launches. The effort required from the programmer to change the program algorithm while using the heterogeneous compilers is substantial and not any less than that required to change the algorithm of a native CUDA program.

6.3 Performance Comparison of Heterogeneous Programs and their CUDA Versions

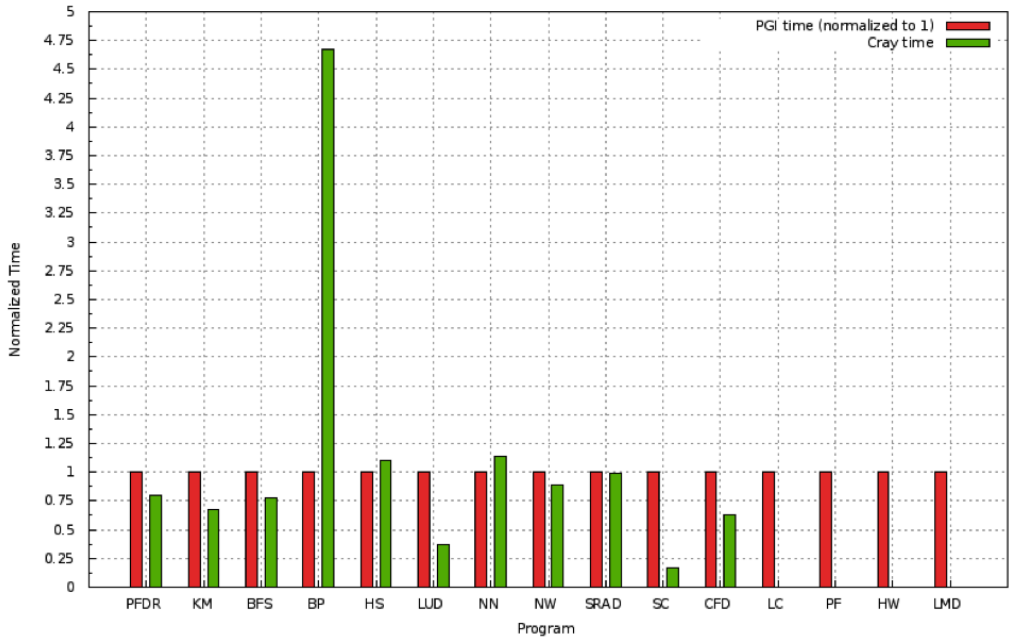
6.3.1 Observations

6.3.1.1 Performance of the Device Kernels of the Heterogeneous Programs and their CUDA Versions

Figure 6.4 shows the total times of execution of the device kernels, these values do not include the times of CPU execution, the host-GPU data communication or the overhead of launching device kernels.

6.3.1.2 Overall Performance of the Heterogeneous Programs and their CUDA Versions

Now we compare the overall performance of the entire programs of the GPU versions of the benchmarks. This will allow us to understand the trade-off between inserting accelerator pragmas into the OpenMP codes versus re-programming the entire algorithm using CUDA, and the performance implications of both these approaches. The results will also allow us to compare the PGI and Cray compilers along the lines of performance, time spent in



The time taken for the execution of GPU kernels launched by the PGI compiler is normalized to 1.

Figure 6.4: Ratio of times taken for the computation in GPU kernels in the heterogeneous versions of Rodinia benchmarks

the GPU and the usage of GPU resources, thus helping us to understand the performance bottlenecks in both heterogeneous versions.

From figure6.5, we can clearly see that CUDA outperforms the heterogeneous versions in most benchmarks. The Cray compiler crashed while compiling the heterogeneous versions of ParticleFilter, Heartwall, LavaMD and Leukocyte, so the performance results of these benchmarks using the Cray compiler are not reported.

In case of 4 benchmarks, both the heterogeneous versions perform within 75% of CUDA performance. For a total of 6 benchmarks, at least one of the heterogeneous versions managed to reach a performance within 85% of the CUDA performance. On the other hand, the heterogeneous versions of k-Nearest Neighbors, Leukocyte, Heartwall and LavaMD performed under 11% of the CUDA performance.

The performance bottlenecks for the heterogeneous versions of Heartwall and LavaMD have been explained above in section 6.2.2. The heterogeneous versions of k-Nearest Neighbors dont parallelize enough part of the kernel as can be seen in figure 5.1, thereby leading to a slower performance as compared to the CUDA version. The extraordinary performance of Leuko-

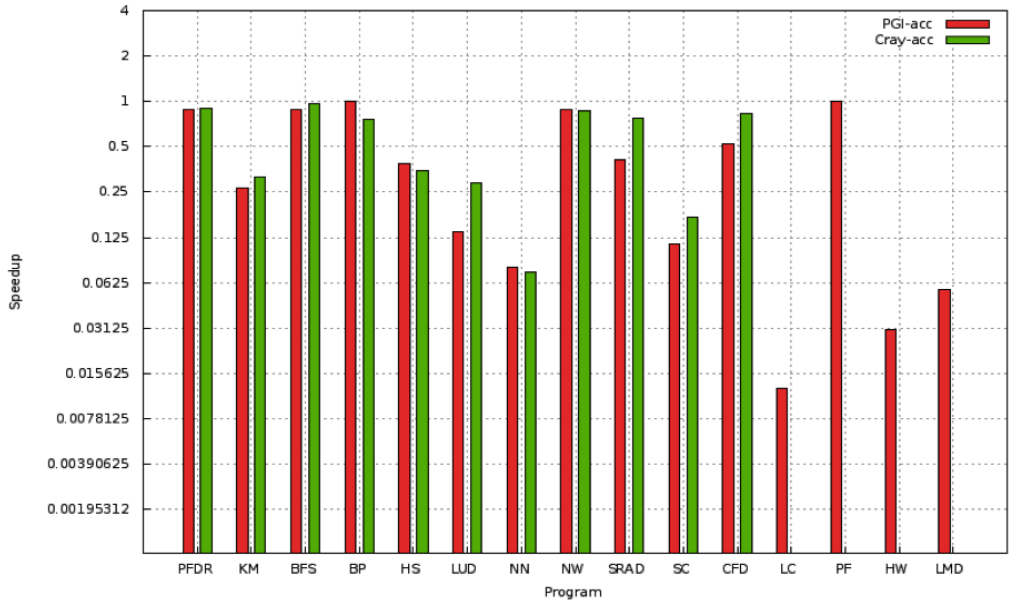


Figure 6.5: Speedups of heterogeneous programs compiled by the PGI and Cray compilers over their CUDA versions

cyte with CUDA is obtained by using a technique called persistent blocking [13, 21] wherein all the iterations are performed in a single device kernel call and all cells are processed concurrently with one thread block allocated to each cell. Similarly for many benchmarks, the CUDA version achieves a superlinear speedup as compared to the OpenMP version. Indeed superlinear speedups have been observed with GPUs [22, 23] and they are typically attributed to caches and very high memory bandwidth. These combined with the persistent blocking optimization produce a superlinear speedup in case of Leukocyte as compared to its sequential version too.

In general, CUDA is expected to provide the best performance since it allows the programmer to express tight control over parallelism and perform micro-optimizations for performance or even change the algorithm to make it more suitable for GPUs data parallelism.

6.3.2 Performance Analysis

Any performance variation between two different heterogeneous versions can be attributed to two components of the program GPU computation and CPU computation. We observed that the CPU-GPU data communication time remains the same with the same transformation steps from section 5

applied to obtain both PGI and Cray heterogeneous versions. This is expected since the both the compilers provide the programmer a fine-grained control over each data transfer step, which allows the programmer to perform the same optimizations for data transfer in both the heterogeneous versions. However, the compilers hide the code generation in the GPU kernels and CPU computation from the programmer, hence the programmer cannot perform same optimizations in these components across both heterogeneous versions. Therefore, the performance variation can be attributed to these two components of the programs.

6.3.2.1 Factors that affect device kernel time

The factors that affect the sequential performance have already been described in section 6.1. The factors that affect the GPU computation performance are multiprocessor occupancy, the usage of GPU resources such as registers allocated per thread and shared memory allocated per thread block. We provide a brief description for these factors:

1. Multiprocessor occupancy: Multiprocessor occupancy is the ratio of active warps, i.e. warps that have state on a GPU multiprocessor at any point of time, to the maximum number of warps that could be supported on a multiprocessor of the GPU. Each multiprocessor houses a finite number of registers and a finite small amount of shared memory. These registers and shared memory are a shared resource that are allocated to thread blocks executing on the multiprocessor. Due to these constraints, each multiprocessor can hold on a finite number of active thread blocks and a finite number of active threads. The size of each thread block also has a finite upper bound. Maximizing the occupancy can help to hide latency during global memory loads. The number of registers on a multiprocessor limits the total number of active threads on a multiprocessor and the shared memory used by each thread block limits the number of active thread blocks. A CUDA programmer typically needs to choose the size of thread blocks such that the resource constraints are satisfied, and enough parallelism is generated using the occupancy and instruction level parallelism (ILP) to hide instruction pipeline latency and memory latency. More specifically for each multiprocessor, required parallelism = latency of global memory accesses that needs to

be hidden (cycles) * throughput of the multiprocessor (instructions executed per cycle) and available parallelism = number of active threads on the multiprocessor * ILP (number of instructions that can execute in parallel per thread). The available parallelism should satisfy the required parallelism for completely hiding the latency of global memory accesses.

The multiprocessor occupancy is a useful measure for comparison only when the underlying programs execute the same set of statements. Hence we dont use occupancy as a measure of comparison between the heterogeneous and CUDA versions, because the CUDA version executes a different set of statements and in some cases with some differences in the algorithm. It should also be noted that occupancy is by no means sufficient to make any conclusions about the actual performance. For example despite having good occupancy, a device kernel can stall on GPU memory accesses if the ILP is not enough to hide the memory latency completely, which could lead to a slower performance as compared to that of a device kernel that has less occupancy, but incurs less overhead of memory accesses due to better use of registers.

2. Register allocation: If a compiler could allocate more registers to each thread, then the compiler can potentially generate more ILP and at the same time reduce global memory accesses by effective use of the registers.

3. Optimal use of shared memory: Similarly, making optimal use of shared memory to cache data can potentially reduce global memory accesses thereby reducing the maximum occupancy required to achieve peak performance.

The use of more registers per thread and shared memory per thread block dont guarantee better performance, but they increase the potential for optimizing global memory accesses and increasing ILP.

6.3.2.2 Performance Analysis of Device Kernels of the Heterogeneous Programs

Table 6.2 presents some important values such as the average occupancy, average number of registers allocated per thread, average shared memory used per thread block, average threads per multiprocessor for both the hetero-

geneous versions. The average is taken per device kernel call, and not per distinct device kernel. Table 6.2 also presents the total number of instructions executed in the GPU, GPU L2 cache misses incurred for each benchmark.

Figure 6.5 shows that the heterogeneous versions compiled by the Cray compiler outperform their counterparts compiled by the PGI compiler in GPU kernel execution times in case of 8 benchmarks. In these 8 benchmarks, the versions compiled by the PGI compiler execute more instructions (including replays) except in the case of Kmeans. However, the heterogeneous version of Kmeans compiled by the PGI compiler incurs high L2 cache misses due to inefficient use of shared memory and slows down.

The heterogeneous versions of Back Propagation, Hotspot and k-Nearest Neighbors compiled by the PGI compiler outperform their heterogeneous counterparts compiled by the Cray compiler. Heterogeneous version of Back Propagation when compiled with Cray compiler incurs a huge amount of L2 cache misses, hence it slows down.

In case of Hotspot, the combined effect of less number of instructions to execute and use of shared memory has likely made the PGI version faster. In case of k-Nearest Neighbors, the large number of registers allocated per thread and the possible ILP available in each thread due to more registers probably overpowered the other factors and made the heterogeneous version compiled by the PGI compiler faster.

The large difference in L2 cache misses displayed by Kmeans, Back Propagation, LU Decomposition and Streamcluster can be attributed to inefficient use of shared memory after taking a look at the source code. These device kernels exhibit memory accesses in nested loops that could be much more efficient if cached in shared memory.

Table 6.3 shows that Cray compiler generally produces kernels that have 66% occupancy. From table 6.4, we also observe that Cray compiler generally tends to allocate more registers to each thread. However, these values are not sufficient to make a concrete conclusion about any benefits of having more registers per thread and we can only speculate that this can potentially increase ILP and offset the reduced occupancy. However from the results it is clear that just like any CUDA program, it is necessary for the hetergoeneous compilers to produce device kernel code that makes efficient use of registers and shared memory.

From table 6.2, we can observe that none of the two compilers emerges

Benchmark	Difference in total instructions		Avg number of threads created per kernel call		Avg occupancy per kernel call		Avg number of registers used per kernel call per thread		Shared memory used per kernel call (kB)		Difference in L2 misses	
	Produced more instructions	Percentage of extra instructions	PGI	Cray	PGI	Cray	PGI	Cray	PGI	Cray	Produced more misses	Percentage of extra misses
PFDR	PGI	44.85	25146.06	1024	1	0.67	10	17	520	0	Cray	5.87
KM	Cray	51.99	25747.2	262211.2	0.58	0.54	20	16	4	256	PGI	98.85
BFS	PGI	32.68	47616	42442.67	0.79	0.67	11.75	15.67	516	173.33	PGI	13.15
BP	Cray	46.33	24570.12	902.1	0.49	0.57	10.39	20.86	867.53	470.09	Cray	266.15
HS	Cray	32.11	65579.75	33642.25	0.75	0.62	24	24.5	224	0	Cray	4.26
LUD	PGI	98.47	160.47	1022.25	0.4	0.66	21	24.98	4	511	PGI	99.93
NN	PGI	37.48	256	128	0.5	0.65	35	20	4	0	Cray	0.06
NW	PGI	20.56	383.98	319.96	0.92	0.63	12	18.5	4	0	Cray	5.13
SRAD	PGI	4.94	11252.54	1146.49	0.75	0.61	16.76	29.95	638.82	0	PGI	20.85
SC	PGI	55.87	2207.76	178100.72	0.58	0.67	14.77	17.67	1794.44	854.58	PGI	95.28
CFD	PGI	22.28	21851.38	22058.12	0.27	0.54	25.25	31.37	10	0	Cray	1.2
LC	NA	NA	522.46	NA	0.79	NA	24.05	NA	86.53	NA	NA	NA
PF	NA	NA	3460.4	NA	0.63	NA	16.55	NA	549.78	NA	NA	NA
HW	NA	NA	3126.4	NA	0.57	NA	18.25	NA	8	NA	NA	NA
LMD	NA	NA	1024	NA	0.83	NA	22	NA	812	NA	NA	NA

NA - Not applicable because the compilation did not succeed.

Table 6.2: Device Kernel Metrics

completely victorious in any metric and the values of these metrics vary with each benchmark. Though Cray compiler has produced better GPU kernel code in more benchmarks, the information obtained through various metrics is not sufficient to conclude with absolute certainty about the strategy used to generate blocks and threads, register allocation, shared memory optimizations and instruction counts with replays. It is difficult to generalize the compilers strengths and weaknesses without further simulations, or taking a look at the actual device kernel code produced by both the compilers which will need further study.

6.3.2.3 Performance Analysis of the Overall Heterogeneous Programs

Now we will explain the difference of performance between the two heterogeneous versions of the benchmarks for certain interesting cases. We notice that the heterogeneous version of LU Decomposition compiled by the Cray compiler outperforms its PGI counterpart by a factor of 2. From table 6.2, it can be observed the faster heterogeneous version of LUD executes almost half the number of instructions and incurs almost half the number of L2 cache misses as compared to the slower version, thereby significantly improving the time of execution on GPU. We can also observe from figure 5.1 that LUD spends more time in the GPU execution than sequential execution on the CPU, and Cray compiler mitigates this time as seen in the figure 6.1.

Similarly, the heterogeneous version of CFD Solver compiled by the Cray compiler displays more average occupancy, registers allocated per thread and also executes almost 22% less instructions, thus it mitigates the GPU execution time which is clearly the most time consuming part of CFD Solver.

SRAD spends almost the same amount of time in GPU in its both heterogeneous versions. However, figure 5.1 shows that SRAD also spends significant amount of time in the CPU part of the program kernel where the executable produced by the Cray compiler demonstrates significant speedup (figure 6.1) and makes the overall program faster (figure 6.5).

These examples show us that for a heterogeneous program to achieve good performance, the traditional analyses and optimizations for producing a good sequential program are as valuable as those needed to produce a good device kernel.

Benchmark	PGI								Cray							
	0-0.125	0.125-0.25	0.25-0.375	0.375-0.5	0.5-0.625	0.625-0.75	0.75-0.875	0.875-1.0	0-0.125	0.125-0.25	0.25-0.375	0.375-0.5	0.5-0.625	0.625-0.75	0.75-0.875	0.875-1.0
PFDR	0	0	0	0	0	0	0	2	0	0	0	0	0	1	0	0
KM	0	1	0	0	0	0	0	1	0	0	0	1	0	1	0	0
BFS	0	1	0	0	0	0	0	3	0	0	0	0	0	3	0	0
BP	3	7	1	0	0	4	0	1	3	0	0	1	1	1	0	0
HS	0	0	0	1	0	0	0	1	0	0	0	0	1	1	0	0
LUD	0	1	0	0	0	0	1	0	1	0	0	0	0	2	0	0
NN	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0
NW	0	1	0	0	0	0	0	1	1	0	0	0	0	1	0	0
SRAD	0	0	1	0	0	0	3	2	0	0	0	0	2	3	0	0
SC	0	2	0	0	0	0	0	3	0	0	0	0	0	4	0	0
CFD	0	3	0	1	0	0	0	1	0	0	1	0	0	4	0	0
LC	0	13	0	0	0	2	2	1	NA	NA	NA	NA	NA	NA	NA	NA
PF	3	1	2	1	0	0	0	10	NA	NA	NA	NA	NA	NA	NA	NA
HW	0	9	0	0	0	1	5	5	NA	NA	NA	NA	NA	NA	NA	NA
LMD	0	0	0	0	0	1	0	1	NA	NA	NA	NA	NA	NA	NA	NA

NA - Not applicable because the compilation did not succeed.

Table 6.3: Number of device kernels in the given occupancy range

Benchmark	PGI						Cray					
	0-10	11-20	21-30	31-40	41-50	51-60	0-10	11-20	21-30	31-40	41-50	51-60
PFDR	2	0	0	0	0	0	0	1	0	0	0	0
KM	0	2	0	0	0	0	0	2	0	0	0	0
BFS	2	2	0	0	0	0	0	3	0	0	0	0
BP	8	3	2	0	0	0	0	4	0	2	0	0
HS	0	1	0	1	0	0	0	1	0	1	0	0
LUD	0	0	2	0	0	0	0	2	1	0	0	0
NN	0	0	0	1	0	0	0	1	0	0	0	0
NW	0	2	0	0	0	0	0	2	0	0	0	0
SRAD	1	2	3	0	0	0	0	1	2	2	0	0
SC	0	5	0	0	0	0	0	4	0	0	0	0
CFD	2	2	0	0	1	0	1	4	0	0	0	1
LC	0	2	11	2	3	0	NA	NA	NA	NA	NA	NA
PF	2	14	0	1	0	0	NA	NA	NA	NA	NA	NA
HW	2	10	7	1	0	0	NA	NA	NA	NA	NA	NA
LMD	0	1	1	0	0	0	NA	NA	NA	NA	NA	NA

NA - Not applicable because the compilation did not succeed.

Table 6.4: Number of device kernels that allocate registers per thread in the given range

CHAPTER 7

GUIDELINES FOR THE PROGRAMMER

In this section, we will present some guidelines that can help the programmer produce efficient executables using the heterogeneous compilers. We will first mention the background knowledge that the programmer needs to have and we will explain the basic characteristics that a program should exhibit so that a heterogeneous compiler can produce an efficient executable. Although the heterogeneous compilers abstract many details of GPGPU programming like the thread block management, writing device kernel code, etc., the programmer must keep in mind important issues like warp divergence of the device kernels, coalesced memory accesses and host-device data communication while transforming the programs to their heterogeneous versions. Therefore we will also provide suggestions to improve performance of the heterogeneous programs and special cares that need to be taken while transforming the program to its heterogeneous version.

Background knowledge required:

- The programmer should have a fundamental understanding of what is parallelism, what constitutes dependences between loop iterations and the Amdahl's law. As described in section 4, since the heterogeneous compilers that we studied could not apply all loop transformations to remove cross-iteration dependences, we recommend that the programmer should also have an idea about how to transform sequential loops to parallelizable loops with independent iterations. It would help further if the programmer had a working knowledge of OpenMP and could correctly transform a sequential program to OpenMP and execute it with appropriate number of threads.

- The programmer should have sufficient understanding of the problem and the algorithm that solves it. This will allow the programmer to

understand whether the algorithm could be parallelized and if it is possible, write the OpenMP code and test it for correctness. Ideally the programmer should use a strategy that exposes as much parallelism as possible, because the code will be eventually transformed and executed in the device.

- The programmer should know that the bandwidth of the PCIE bus between the CPU and the GPU is quite low as compared to GPU memory bandwidth. Hence, it makes sense to use the GPUs only when the device kernels have enough computation to data communication ratio, otherwise the program should be executed on the CPU.

- The programmer should understand the term coalesced memory accesses and be able to optimize loop iterations to minimize the non-coalesced memory accesses.

- The programmer should understand the term warp divergence and know that divergent warps can reduce the performance.

Characteristics that the program should exhibit:

- The program's algorithm should be amenable to parallelizing, that is, the algorithm should be amenable to be transformed in such a way that the transformed algorithm has loops with independent iterations.

- The loops from the algorithm obtained in the above step should have large amount of parallelism in terms of threads that can be executed in parallel, so as to extract efficient performance from the GPU.

- A heterogeneous program can run slower than its OpenMP version (or the sequential version) if the benefit offered by GPU parallelism is offset by the amount of data communication required. The device kernels produced by the heterogeneous compilers should have a high computation to data communication ratio, so that the host-device data transfer does not become the performance bottleneck.

Issues related to performance improvement:

- We recommend transforming programs to their heterogeneous versions in the order we have transformed the Rodinia benchmarks in section 5

to obtain good performance. This performance can further be improved by using the parallel, vector and async clauses, which could be an extension of this work.

- Working in incremental steps will clearly make it easier for the programmer to analyze the impact of each transformation in the resulting compilation reports, verify the correctness of results and evaluate the performance improvement.

- The programmer should keep a close eye on the compilation reports to determine which loops are parallelized in the device and the locations of data movement in the program.

- While performing any transformation, the programmer must be aware of the data movement and use data regions to avoid any cyclic host-device data movements.

- The programmer needs to be careful with reductions that are inside a bigger device kernel. Depending on implementation, a heterogeneous compiler could choose to form a separate parallel reduction kernel or run the entire bigger kernel sequentially in the GPU due to dependences introduced by the reduction operation. If the compiler does not form a separate reduction kernel or the report does not explicitly state that it is performing reduction in parallel on the GPU, then the programmer should apply transformation 10.b.

- While performing transformation 10.c, sometimes computation that is not highly parallel can be included into parallel regions if doing so allows for optimization of data movement using data regions. The programmer should carefully evaluate the trade-off of offloading computation that is not highly parallel to GPU versus resulting optimization of data movement.

- While performing transformation 10.d, changing the loops that are parallelized may give rise to non-coalesced memory accesses. The programmer should carefully evaluate the performance difference resulting from the tradeoff of increased parallelism versus cost of memory accesses.

- Loop nests to be executed on the GPU must be rectangular. Non-rectangular loops cause warp divergence which could produce significant performance slowdowns.

- A parallel region that includes a loop nest formed by an outer sequential or parallelizable loop and an inner parallelizable loop requires careful performance tuning by trial and error.

    a. If the outer loop is not parallelizable, the programmer should keep an eye on whether the outer loop executes redundantly and only one device kernel is launched or whether a device kernel is launched multiple times in the outer loop. Clearly a single kernel launch with redundant outer loop execution will reduce overhead and give better performance. If there are multiple device kernel launches, the eventual performance will be affected by the device kernel launch overhead, overhead of any data transfer and the computation inside each kernel. We have observed that in many cases, the Cray compiler takes the former path while the PGI compiler chooses the latter.

    b. If both the inner and outer loops are parallelizable but loosely nested, then clearly the loops cannot be fused together due to loose nesting. The programmer needs to evaluate the performance obtained by parallelizing the outer loop versus the performance obtained by parallelizing the inner loop. If the inner loop accesses the device global memory then parallelizing the outer loop can cause the caches to flush very frequently because of more combined irregular memory accesses by multiple threads executing in parallel. On the other hand if the inner loop is parallelized, then the program can incur a heavy overhead of multiple kernel launches.

Miscellaneous issues:

- Pointer arithmetic is not allowed in the accelerator region. Hence, any such arithmetic should be converted to array notation or should be performed before the accelerator region.

- Variables that are used after the end of a data region cannot be declared

inside the data region, because they dont have any memory allocated outside the data region.

- Since the PGI and Cray compilers cannot work with unions while producing device kernel code, the programmer needs to inline them, that is, create a variable that holds the same amount of memory as the union. The use of different variables inside the union can be emulated by pointer casts on the CPU and by using multiple copies of data with different data types on the GPU. However, issues like of nesting of unions within unions, nesting of structures with unions etc. can significantly increase the manual work required for transformation T5 and having multiple copies of data in the GPU can make the program extremely inefficient due to increased data transfer.

- If an array is allocated memory on the GPU using data clauses on a data region, then the array should not be used on the CPU computation inside the data region. Reusing the same array inside the data region for both CPU and GPU computation will produce incorrect results and it is a mistake that programmers can easily make due to oversight.

- *while* loops are not parallelizable by default. They can be made parallelizable by converting them to rectangular *for* loops with finite trip counts.

- The programmer must be careful about the possibility of any garbage data in the GPU overwriting the correct data in the host or the correct data in the GPU not overwriting old results in the CPU. This typically happens with the incorrect use of copy clauses.

- The programmer should watch out for runtime device errors.

- We observed that the use of PGI private clause can sometimes cause heavy overhead. It does not increase the GPU execution time or the sequential execution time. However the overhead caused is so high that it changed the runtime of CFD from 10 seconds to 34 seconds. This issue can perhaps be fixed in the later releases of the PGI compiler.

CHAPTER 8

CONCLUSIONS

In this thesis, we have proposed a sequence of transformations to transform OpenMP programs to a form understood by the PGI and Cray compilers and analyzed the impact of each transformation on the performance of 15 Rodinia benchmarks. Our results show that all the transformations lead to incremental performance improvement and the extent of improvement obtained by each transformation depends on the structure of the program. We have also proposed a set of guidelines for programmers to extract good performance from heterogeneous programs and discussed the characteristics of a program that make it amenable to compiling with heterogeneous compilers.

We analysed the performance of sequential, OpenMP and heterogeneous executables produced by the PGI and Cray compilers. In 10 out of 15 Rodinia benchmarks, both sequential and OpenMP executables produced by the Cray compiler outperform those produced by the PGI compiler. Among the heterogeneous versions of 8 out of 11 Rodinia benchmarks, heterogeneous programs compiled using the Cray compiler produced faster GPU kernels than their versions compiled using the PGI compiler. Out of the same 11 benchmarks, the overall performance of 7 heterogeneous programs compiled using the Cray compiler was better than their versions compiled using the PGI compiler. These results also show that for a heterogeneous program to produce good performance, the traditional analyses and optimizations for producing a good sequential program are as valuable as those needed to produce a good device kernel.

We analysed and compared the performance of the heterogeneous benchmarks with the performance of their CUDA versions. Our analysis showed that the heterogeneous programs could achieve upto 85% of CUDA performance in 6 out of 15 benchmarks and it shows the potential of heterogeneous compilers to produce good heterogeneous code and increase programmer productivity at the same time.

Our analysis also points to some of the shortfalls of using heterogeneous compilers. Many well-known techniques that could remove dependences in loops and expose parallelism are not currently successfully employed by the PGI and Cray compilers. The information obtained from the performance analysis of Rodinia benchmarks is not sufficient to conclude with absolute certainty about various issues such as the strategy employed by the compilers to generate thread blocks, register allocation, shared memory optimizations and instruction counts, etc. It also points to the uneven behavior of compilers across different benchmarks due to which it is difficult to generalize the compilers strengths and weaknesses without further simulations, or taking a look at the actual device kernel code produced by both the compilers. Such uneven behaviour makes it difficult for programmers to make an informed decision of choosing a heterogeneous compiler that suits their needs. Previous studies have already shown that there are no universally accepted criteria to determine what transformations and optimizations are important for exploiting parallelism in CPU programs. Since a compiler has limited resources, it is important to identify the best optimization techniques, however our results show that there is a similar lack of universal criteria for the optimization of GPU kernels too.

For the heterogeneous compilers to fulfill their promise of providing performance along with reduction in manual efforts for optimizations and become a practical alternative to programming models like CUDA or OpenCL, minimal modifications should be required to sequential or OpenMP programs. The efforts required from programmers by the current heterogeneous compilers are already significantly less than their CUDA or OpenCL counterparts for simple programs. Since the development of heterogeneous compilers is an ongoing process, we can expect that bug-free implementations and better analyses will make the efforts-to-performance tradeoff offered by these compilers suitable for working with more complicated programs. In such a scenario, it will be helpful to have a set of well-formulated criteria that the programmers can use to determine whether their programs would achieve similar performance as CUDA/OpenCL or incur performance degradation when compiled with heterogeneous compilers. It remains a topic of further study whether such a set of criteria could be universally agreed upon or whether it remains an illusive magic recipe.

REFERENCES

[1] J. Enos, C. Steffen, J. Fullop, M. Showerman, G. Shi, K. Esler, V. Kindratenko, J. Stone, and J. Phillips, "Quantifying the Impact of GPUs on Performance and Energy Efficiency in HPC Clusters," in *Green Computing Conference, 2010 International*, Aug. 2010, pp. 317 –324.

[2] "Blue Waters Website," 2011. [Online]. Available: {http://www.ncsa.illinois.edu/BlueWaters/}

[3] "Bring High-End Graphics to Handheld Devices," NVIDIA Whitepaper, NVIDIA Corporation.

[4] "The Benefits of Multiple CPU Cores in Mobile Devices," NVIDIA Whitepaper, NVIDIA Corporation.

[5] T. Han and T. Abdelrahman, "hiCUDA: High-Level GPGPU Programming," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 22, no. 1, pp. 78 –90, Jan. 2011.

[6] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. Beard, and D. I. August, "Automatic CPU-GPU Communication Management and Optimization," *SIGPLAN Not.*, vol. 47, no. 6, pp. 142–151, June 2011. [Online]. Available: http://doi.acm.org/10.1145/2345156.1993516

[7] M. Amini, F. Coelho, F. Irigoin, and K. R., "Static Compilation Analysis for Host-Accelerator Communication Optimization," in *LCPC*, 2011.

[8] H. McIntyre, S. Arekapudi, E. Busta, T. Fischer, M. Golden, A. Horiuchi, T. Meneghini, S. Naffziger, and J. Vinh, "Design of the Two-Core x86-64 AMD; Bulldozer; Module in 32 nm SOI CMOS," *Solid-State Circuits, IEEE Journal of*, vol. 47, no. 1, pp. 164 –176, Jan. 2012.

[9] C. M. Wittenbrink, E. Kilgariff, and A. Prabhu, "Fermi GF100 GPU Architecture," *IEEE Micro*, vol. 31, pp. 50–59, 2011.

[10] The Portland Group, "PGI Compiler Reference Manual," 2011. [Online]. Available: http://www.pgroup.com/doc/pgiref.pdf

[11] CRAY, "Cray Compiler Environment," 2011. [Online]. Available: http://docs.cray.com/books/S-2179-52/html-S-2179-52/index.html

[12] NVIDIA Corporation, "NVIDIA CUDA Programming Guide version 4.0," 2011. [Online]. Available: http://developer.download.NVIDIA.com

[13] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron, "Rodinia: A benchmark Suite for Heterogeneous Computing," in *IISWC 2009*, Oct. 2009, pp. 44 –54.

[14] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron, "A Characterization of the Rodinia Benchmark Suite with Comparison to Contemporary CMP Workloads," in *IISWC'10*, 2010. [Online]. Available: http://dx.doi.org/10.1109/IISWC.2010.5650274 pp. 1–11.

[15] K. Kennedy and J. R. Allen, *Optimizing compilers for Modern Architectures: a Dependence-Based Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002.

[16] "Compute Command Line Profiler," NVIDIA Whitepaper, NVIDIA Corporation.

[17] CAPS Enterprise and Cray Inc. and NVIDIA and the Portland Group, "The OpenACC Application Programming Interface, v1.0," Nov. 2011. [Online]. Available: http://http://www.openacc-standard.org/

[18] Mark Harris, "Optimizing Parallel Reduction in CUDA," NVIDIA Corporation. [Online]. Available: http://people.maths.ox.ac.uk/gilesm/cuda/prac4/reduction.pdf

[19] Mark Harris, "Parallel Prefix Sum (Scan) with CUDA," NVIDIA Corporation. [Online]. Available: http://http.developer.nvidia.com/GPUGems3/gpugems3_ch39.html

[20] B. Dhanasekaran and N. Rubin, "A New Method for GPU based Irregular Reductions and Its Application to K-Means Clustering," in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, 2011. [Online]. Available: http://dl.acm.org/citation.cfm?doid=1964179.1964182 pp. 34 –41.

[21] M. Boyer, D. Tarjan, S. T. Acton, and K. Skadron, "Accelerating Leukocyte Tracking using CUDA: A Case Study in Leveraging Manycore Coprocessors," in *Proc. of the 2009 IEEE Int. Symp. on Parallel&Distributed Processing*, ser. IPDPS '09, 2009. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2009.5160984 pp. 1–12.

[22] D. G. Spampinato and A. C. Elstery, "Linear optimization on modern GPUs," in *Proceedings of the 2009 IEEE International Symposium*

on Parallel&Distributed Processing, ser. IPDPS '09. Washington, DC, USA: IEEE Computer Society, 2009. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2009.5161106 pp. 1–8.

[23] A. Leist, D. P. Playne, and K. A. Hawick, "Exploiting graphical processing units for data-parallel scientific applications," *Concurr. Comput. : Pract. Exper.*, vol. 21, no. 18, pp. 2400–2437, Dec. 2009. [Online]. Available: http://dx.doi.org/10.1002/cpe.v21:18