Employing Directive Based Compression Solutions on Accelerators Global Memory under OpenACC

by

Ebad Salehi

B.Sc., Sharif University of Technology, 2012

A Thesis Submitted in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF APPLIED SCIENCE

in the Department of Electrical and Computer Engineering

Employing Directive Based Compression Solutions on Accelerators Global Memory under OpenACC

by

Ebad Salehi

B.Sc., Sharif University of Technology, 2012

Supervisory Committee

Dr. Amirali Baniasadi , Supervisor

(Department of Electrical and Computer Engineering)

Dr. Kin Fun Li, Departmental Member

(Department of Electrical and Computer Engineering)

ABSTRACT

Supervisory Committee

Dr. Amirali Baniasadi , Supervisor

(Department of Electrical and Computer Engineering)

Dr. Kin Fun Li, Departmental Member

(Department of Electrical and Computer Engineering)

Programmers invest extensive development effort to optimize a GPU program to achieve peak performance. Achieving this requires an efficient usage of global memory, and avoiding memory bandwidth underutilization. The OpenACC programming model has been introduced to tackle the accelerators programming complexity. However, this models coarse-grained control on a program can make the memory bandwidth utilization even worse compared to the version written in a native GPU languages such as CUDA. We propose an extension to OpenACC in order to reduce the traffic on the memory interconnection network, using a compression method on floating point numbers. We examine our method on six case studies, and achieve up to 1.36X speedup.

Contents

Supervisory Committee **ii**

Abstract **iii**

Table of Contents **iv**

List of Tables **vii**

List of Figures **viii**

Acknowledgements **x**

Dedication **xi**

1 Introduction **1**

1.1 Introduction 1

2 Background **4**

2.1 GPU Architecture 4

2.1.1 SIMD Architecture 5

2.1.2 Memory Subsystem 5

2.2 CUDA 9

2.3 OpenACC Model 9

2.4 IPMACC 10

2.4.1 IPMACC Infrastructure 10
2.5 Floating Point 12

3 Related Works **14**

4 Proposed Compression Clauses and Implementation **18**
4.1 Proposed Compression Clauses 18
4.1.1 Data Transfer Clauses 19
4.1.2 Compression Clause 20
4.2 Compression Implementation 22
4.2.1 Compression 22
4.2.2 Decompression 24
4.3 Runtime Library Extension 26

5 Methodology **28**
5.1 Benchmarks 28
5.2 OpenACC Compiler 28
5.3 Performance Evaluations 29
5.4 Platform 29

6 Experimental Results **30**
6.1 Microbenchmarks 30
6.1.1 Simple Data Copy 31
6.1.2 Simple Data Copy in Reverse Order 33
6.1.3 Strided Access Pattern 34
6.1.4 Random Access Pattern 35
6.2 Real World Benchmarks 37
6.2.1 Vector Product 37

6.2.2 Matrix Addition 37
6.2.3 Matrix Multiplication 39
6.2.4 HotSpot 41
6.2.5 Nearest Neighbor 46
6.2.6 Dyadic Convolution 48
6.3 Discussion 49

7 Conclusions **50**
7.1 Conclusion and Future Work 50
7.2 Future Work 51

Bibliography **52**

List of Tables

Table 4.1 Compression Copy Clauses 20

Table 4.2 Runtime Routines for Compression 27

List of Figures

Figure 2.1 Ideal Memory Access 7

Figure 2.2 Nonsequential Memory Access 8

Figure 2.3 Unaligned Memory Access 8

Figure 2.4 IPMACC Compilation Flow 12

Figure 2.5 IEEE 754 Floating Point Format 13

Figure 4.1 Illustration of Compression and Decompression methods . . . 26

Figure 6.1 Simple Data Movement (each data element is a 8 bit double precision floating point number) 32

Figure 6.2 Simple Data Movement in Reverse Order (Each data element is a 8 bit double precision floating point number) 34

Figure 6.3 Strided Access Pattern (Each data element is a 8 bit double precision floating point number) 35

Figure 6.4 Data shuffling (Each data element is a 8 bit double precision floating point number) 36

Figure 6.5 Vector Product Kernel Time Improvement 38

Figure 6.6 Matrix Addition Kernel Time Improvement 39

Figure 6.7 Matrix-Matrix Multiplication Kernel Time Improvement . . . 40

Figure 6.8 Matrix-Matrix Multiplication Application Run-Time Improvement 40

Figure 6.9 Compression-Enhanced MM Multiplication Breakdown 41

Figure 6.10 Baseline MM Multiplication Breakdown 41
Figure 6.11 HotSpot Kernel Performance Improvement 42
Figure 6.12 HotSpot Application Run-Time for First 700 Iterations 43
Figure 6.13 NN Kernel Time Improvement 47
Figure 6.14 Dyadic Convolution Kernel Time Improvement 48

ACKNOWLEDGEMENTS

I would like to give my warm regards to my supervisor Dr. Amirali Baniasadi for his guidance and support throughout my research, and giving me the opportunity to pursue my graduate studies at the University of Victoria. I would also like to thank all the faculty members and staff who provided a favorable setting for my studies and research, especially Dr. Nikitas Dimopoulos for his valuable technical feedbacks.

I am blessed to have had supportive friends who helped me through the struggles that one faces when moving to the other side of the world, as well as with my graduate studies by providing me with an extraordinary environment where I was able to thrive. My gratitude goes to Mohammad, Farhang, Azadeh, Behnam, Ali, Babak, Salma and Alireza to name a few.

I would also thank my laboratory partner, roommate and close friend Ahmad for sharing his expert technical knowledge and precious help.

Finally, I would like to express my deepest gratitude to my parents and sister for their continuous support and unconditional love, without whom this work would not have been possible.

DEDICATION

To my parents and my beloved sister for their patience, support and love

Chapter 1

Introduction

1.1 Introduction

Graphics Processing Units (or simply GPUs) can potentially provide very high peak performance. This high performance, however, has proven to be very difficult to achieve. There are many obstacles facing designers in the path to achieve this performance. In particular, in order to make use of the high amount of computation power in GPU computational resources, the required data should be accessed from memory in a very timely fashion. One way to achieve lower memory access time is to use the memory bandwidth more efficiently. To this end, many studies have suggested both hardware and software solutions [7, 13, 21].

Despite all past efforts, developing a GPU program which utilizes the memory bandwidth effectively and efficiently calls for strong skills and vast experience.

Programming models such as CUDA(Compute Unified Device Architecture) allow for fine-grained control over both data and parallelism. CUDA's flexibility allows programmers to fully optimize their applications, albeit by extensive development effort. The OpenACC programming model [5] was designed to alleviate the complexity of

accelerator programming. OpenACC provides programmers with a directive-based API so that they can accelerate compute intensive parts of codes by offloading the computations to the accelerator. Although OpenACC may decrease the development effort by a factor of 6.5X[20], this is at cost of performance loss.

There are many computation domains in which variation in results are tolerable to some extent. For example applications such as image, video, and audio processing, dynamical simulations, machine learning algorithms are domains for which approximation can be used to trade accuracy for other factors including but limited to performance, power, and storage. Some previous works have been done to investigate the feasibility of approximation and how it affects the outcome in aforementioned areas. We talk about this in the next chapter more.

In this work we enhance GPU performance by proposing a set of clauses for OpenACC to allow programmers to achieve faster runtime at the expense of negligible accuracy cost. These clauses make it possible for programmers to mark the datasets for which they do not need full precision and therefore can exploit compression. Once such data sets are identified, then our run-time framework performs a fast analysis on the data and applies the compression method. It also stores the necessary information later required for decompression.

In summary, we make the following contributions:

- We introduce a new double-precision floating-point compression technique for GPUs to save global memory bandwidth. Our technique reduces the memory bandwidth demand by half.

- We propose a novel OpenACC clause that allows programmers to exploit our compression technique readily in a high-level language. We show that with a minor modification in existing OpenACC code, we can enhance OpenACC applications to take advantage of our compression technique.

- We evaluate our proposed compression solution for six real world OpenACC benchmarks. In each benchmark, we investigate various data set sizes and compare the performance of our proposed solution to the baseline OpenACC.

- We investigate the run-time performance of our compression technique, reporting the breakdown of time in i) kernel execution, ii) kernel launch, and iii) compression overhead.

The remainder of this document is as follows. In Chapter 2, we review the background of accelerator programming platforms and present a brief summary of the floating point IEEE 754 Standard. In Chapter 3, we review related works. In Chapter 4, we introduce our proposed OpenACC clauses and discuss how to use them. We also detail the compression method scheme we use in this work in that chapter. In Chapter 5, we present our methodology. In Chapter 6, we evaluate our solution using many case studies, and discuss issues regarding the clauses applicability and performance. Finally, in Chapter 7 we offer conclusions, and end this thesis with future work.

Chapter 2

Background

The focus of this thesis is applying compression on floating point data transfered between GPU memory and its cores which is controlled by OpenACC directives. To understand this work thoroughly the reader should know the underlying basis of discussed topics. This chapter provides an overview of the fundamental concepts of GPU architecture, IEEE floating point standards, and OpenACC programming model.

2.1 GPU Architecture

This section gives a summary of a GPU general architecture focusing on the units that matter in this work. The architecture which is explained in this section is based on the NVIDIA Kepler GPU architecture. We mention the differences with other architectures if it is necessary. We also use NVIDIA and CUDA terminology to refer various GPU concepts in this document.

2.1.1 SIMD Architecture

GPUs execution units are located in Streaming Multiprocessors (SM). Each SM consists of a number of execution lanes, namely, Streaming Processor (SP). SPs in an SM use Single-Instruction, Multiple-Data (SIMD) architecture to execute the instructions in order to improve the computation efficiency. Therefore, threads assigned to an SM are grouped into 32 thread wavefronts (Warp) which execute in lockstep manner on the SIMD hardware. In the case that threads follow different execution paths, the hardware uses a stack-based mechanism to handle the control flow divergence by serializing the execution of threads that have diverged to other paths.

2.1.2 Memory Subsystem

There are many data storage resources on GPUs which are used for different purposes. In this subsection we introduce the most important ones which are also used in our work.

DRAM (global memory) is the main data storage on the GPU. It is accessed by SMs through a 2-level cache hierarchy. Each SM contains an L1 cache (64 KB in Tesla K20) which can also be statically configured to perform as a software managed cache. There is also an L2 cache shared between all SMs. In the case that data is not available in neither of aforementioned caches the global memory is accessed.

Due to memory and interconnection network latency threads in a warp should wait some cycles when they access global memory to receive the requested data. In the case that there are ample number of warps running on an SM, the memory latency does not degrade performance, since other warps can execute their instructions in the meantime. So enough parallelism can prevent SM under-utilization which, unfortunately, is not possible in all applications.

The global memory access latency cycles vary for different memory access patterns

and sizes. Ideally, all the threads in a warp access aligned sequential addresses in the memory as shown in figure 2.1. Therefore, all the memory requests are coalesced into one memory transaction. On the K20 GPUs, every successive 128 bytes which can be accessed as 32 single precision or integer words by a warp (32 consecutive threads) in a single memory transaction. However, the following conditions may cause uncoalesced memory accesses which are handled in serial transactions, and increase the memory latency.:

- Non-sequential Memory Access
- Unaligned Memory Access
- Sparse Memory Access

Non-sequential Memory Access: Threads in a warp may access words in memory in different orders as illustrated in figure 2.2, and it may cause serialization in handling the memory requests. However, in K20 GPUs as long as all the accessed words belong to a memory block, they can be combined into a single memory transaction. The earlier NVIDIA GPUs with older compute capabilities are not able to handle these kind of memory access in a single transaction though.

Unaligned Memory Access: The memory access pattern shown in figure 2.3 is sequential but unaligned. The last requested word is in a different memory block and must be accessed in a different memory transaction. Therefore, in this case two memory accesses are required.

Sparse Memory Access: If threads in a warp access words in different memory blocks, then more than one memory transaction is needed to handle all threads' requests. Depending on how sparse the access addresses are, the number of memory transactions vary.

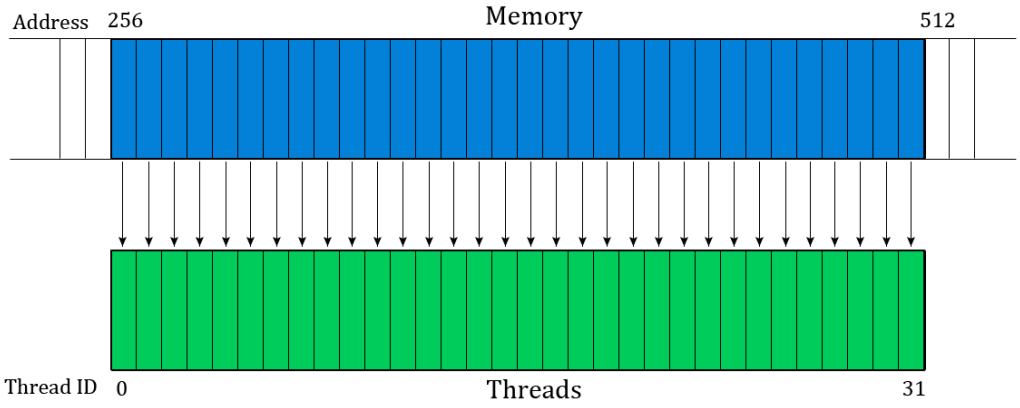

Figure 2.1: Ideal Memory Access

To use the memory bandwidth more efficiently, NVIDIA GPUs with compute compatibility of 1.2 and higher are capable of performing transactions of three different sizes; 128, 64 and 32 byte transactions. This capability is useful to avoid wasted bandwidth when the memory access pattern is very sparse.

Constant memory is part of the global memory (64 KB in Tesla K20), and physically resides on the dram. It is cached in per streaming multiprocessor constant caches (8 KB in Tesla K20). In the cache miss cases its access latency is as long as global memory. But in the case that the requested data exists in the constant cache, the access latency can be as low as accessing registers, depending on the number of unique requested addresses. Since constant memory is optimized for the situations when a warp of threads reads the same location, the lowest access latency is achieved when a single data element should be broadcast between all threads of a warp. If different addresses are accessed then the memory requests get handled serially.

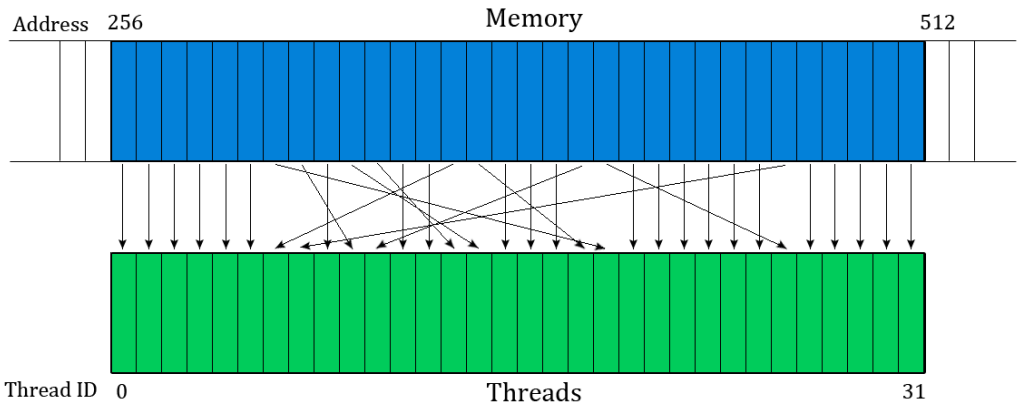

Figure 2.2: Nonsequential Memory Access

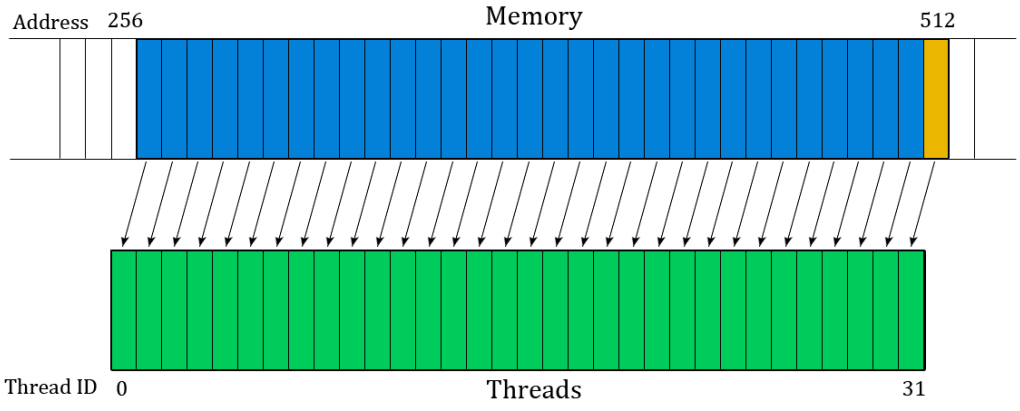

Figure 2.3: Unaligned Memory Access

2.2 CUDA

CUDA programs consist of host and device codes which execute on CPU and GPU respectively. Parallel portions of the code which are executed on the device are referred to as kernels. Thousands of lightweight threads execute the same CUDA kernel. These threads are identified by their unique ID and access different bytes of the off-chip DRAM memory which is shared among them. This results in a huge number of memory requests. Depending on how memory intensive an application is, the memory traffic can saturate the memory bandwidth and make it a bottleneck. Matrix multiplication is an example of such applications. One solution to this problem is to tune CUDA programs and employ computation overlapping techniques to hide the memory latencies. However, this is a tedious process and is not always possible.

2.3 OpenACC Model

OpenACC is an API by which programmers can offload regions of a serial code from CPU to an accelerator. This API provides two types of compiler directives to facilitate the process of accelerating serial codes. The two API classes are data management and parallelism control. Directives can be accompanied by multiple clauses to control different parameters.

Data management directives allow programmers to transfer data between host and accelerator devices as well as allocating device memory. Parallelism control directives specify regions of the code which are intended to be executed on the accelerator (e.g., mostly work-sharing loops). They also enable controlling parallelism at different granularities, variable sharing/privatization, and variable reduction. In OpenACC there are four levels of parallelism: gang, worker, vector, and thread. The equivalent terms in CUDA terminology are kernel, thread block, warp, and thread, respectively.

2.4 IPMACC

IPMACC is a framework for translating OpenACC programs to CUDA or OpenCL. To generate the final executables for accelerators the framework makes use of system compilers such as nvcc. In this work we use IPMACC to extending the OpenACC model. In this section we detail the features of IPMACC.

2.4.1 IPMACC Infrastructure

IPMACC accepts C/C++ files enhanced by OpenACC API to execute on accelerators. The output of the compilation process can be object code, binary executable or equivalent CUDA/OpenCL codes. Figure 2.4 illustrates the IPMACC compilation flow which consists of 4 steps.

Pre-process: In this stage the syntax of the input file is verified. The code should comply with C/C++ and OpenACC API syntax. IPMACC also normalizes the C/C++ syntax notations to simplify the following compilation stages. For instance after this stage all control and loop regions (if, while, for) will be fully-bracketed which makes it easier to locate the scope of each OpenACC region.

Intermediate XML Form: During this step the code is categorized into segments, each of which enclosed with one of the following XML tags: *C code*, *OpenACC pragma*, and *for loop*. The codes in the C code tags will not be touched during the compilation, and remain as they are. OpenACC pragma tags are replaced with function calls from run-time system to implement the desired functionality, such as data movement or memory allocations. The codes tagged by *for loop* remain to be analyzed for parallelization based on the directive used in the OpenACC code.

Intermediate XML format to CUDA/OpenCL: Translating the intermediate XML format to CUDA/OpenCL source code is performed in nine steps in

IPMACC.

1. In this step OpenACC information is extracted from code and stored in the process. The OpenACC related tags are replaced with dummy function calls. At the end of the compilation these function calls are replaced with the codes executing the computation on the accelerator (ex. CUDA kernel launch), data transfers codes, and host-accelerator synchronization operations. After this step XML tags are omitted, and the code returns back to C/C++ syntax. Then the code is split into two codeblocks: i)**Regions code** which includes OpenACC data clauses and kernels regions, and ii)**Original code** which is non-OpenACC related codeblock.

2. By calculating the syntax tree of C/C++ code, IPMACC keeps track of variable, types and functions defined out of the OpenACC regions. This information is necessary for generating the accelerator code.

3. In this step, which is performed in parallel to the previous one, global scope is searched as well as the parent scope of each Regions code for declarations/prototypes that are referred to. Dummy function calls which represent single Regions code are used as entry points for the search.

4. The kernel code which is targeted to be executed on the accelerator is generated in this step. To do so, the available parallelism should be specified, loop iterations should be shared between threads, and reductions should be performed on shared variables. In this step kernel arguments also are defined, and out-defined declarations/prototypes get regenerated.

5. In parallel to step 4 the dummy functions calls associated with OpenACC memory management clauses are replaced with real functions that perform the intended memory operations including host-accelerator pointer exchange, data copy in/out, and memory allocation.

6. In parallel to step 4 user-defined types and functions corresponding to each

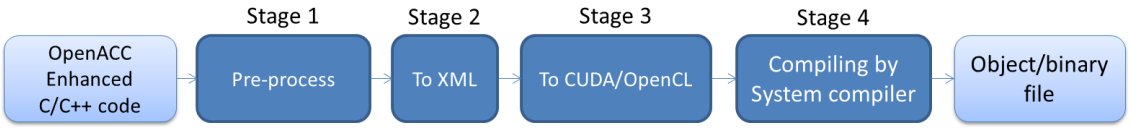

Figure 2.4: IPMACC Compilation Flow

Regions code are found by searching the AST of original code created in the 2nd stage of the compilation. These declarations will be modified and added to the kernel code.

7. Now that the kernel is constructed it should be invoked. In this step dummy OpenACC kernel region calls are replaced with the kernel invocations. In case there is a reduction clause in loop directive an extra function call will be added to as well.

8. By appending forward declaration to the code, it will be able to execute on the target accelerator.

9. The generated accelerated code is stored on the disk. The file extension is specified based on the target language (CUDA/OpenCL).

Generating the binary file using system compiler: In this stage, the system compiler (nvcc or g++) is invoked to generate the final binary file. This invocation uses the compilation flags passed to ipmacc command at the beginning of the compilation process. The output can be either an object file or an executable binary.

2.5 Floating Point

Floating point standard is defined in the IEEE 754 Standard so that different platforms can exchange floating point data consistently. Two types of floating point numbers, which are commonly used are 32 and 64 bit numbers. 32 bit encoding is referred to as single-precision and the 64 bit standard is called double-precision. The `float` and `double` are the equivalent C data types respectively. Any floating point

Figure 2.5: IEEE 754 Floating Point Format

number consists of a sign bit, exponent part, and mantissa. Figure 2.5 illustrates how many bits are dedicated to each part of floating point numbers in IEEE 754 standard. (The numbers in brackets corresponds to double precision.)

Floating point representation covers a large range of real numbers. The precision of a number stored in this format is inversely proportional to its magnitude. Small numbers have higher precisions and precision drops as the numbers grow. Formula 2.1 decodes binary floating point numbers in IEEE 754 format. One can realize that the floating point numbers between two consequent powers of two have equal exponents and differ only at their mantissa. Hence, the number of numbers that can be encoded in floating-point between consequent powers of two is constant and limited to the mantissa's different values. We exploit this fact and map a range of numbers to a range of same-exponent numbers and omit the redundant exponent part.

$$(-1)^{sign} \times 2^{exponent-bias} \times (1.mantissa)_2 \tag{2.1}$$

In this chapter we provided the necessary background information one should know to understand the work in this document. However, depending on the readers expertise they may need additional information. We suggest reading the referenced documents for more details on the concepts discussed in this thesis.

Chapter 3

Related Works

To the best of my our knowledge, our work is the first attempt to add compression capabilities to compiler directive based frameworks. Samadi et al. [17] developed SAGE, a transparent approximation system, which accepts CUDA programs and improves performance at the cost of losing accuracy to the level programmers specify. Their static compiler automatically generates multiple kernels with various degrees of approximation and a run-time system chooses one of the kernels according to the feedbacks. They applied data compression to reduce DRAM memory bandwidth usage as an approximation technique.

Ratanaworabhan et al. [16, 8] introduced lossless floating-point value compression algorithms suitable for situations that fast real-time compression and decompression operations are needed. These algorithms are inspired by VPC, a value prediction-based compression algorithm for program execution traces. These predictor algorithms are used in hardware to predict the content of CPU registers, including floating-point registers. To achieve this goal the algorithms are designed to perform a high volume of predictions per second. However, the proposed compression algorithms do not maintain a fixed compression rate, and depending on the input data

the compression ratio changes. The suggested compression algorithms are designed to work with streams of data, and to process the data sequentially which makes it less applicable in parallel devices such as GPU.

Sathish et al. [18] show the efficiency of hardware-based lossless and lossy compression methods to improve the performance of the off-chip dram memory in GPUs. They present modifications needed to the GPU microarchitecture to implement lossless and lossy compression methods for data transfer through the GPU memory I/O links. Their proposed programming model and unique memory controller architecture is transparent to the workloads and compilers. The lossy compression technique exploits the fact that floating-point numbers' precision is usually higher than necessary, and truncating the least-significant bits can be considered as a low-risk compression method. They show that GPUs equipped by their proposed compression hardware can improve the performance of many memory-bound workloads by 26% on average for GPUs with L1 caches but without L2 caches, at the cost of negligible power and area overheads compared to those of a GPU. In the case using the compression technique and adopting an L2 cache of size 768KB the performance improvement elevates to 37% compared to that of a GPU with the same size L2 caches but without the compression technique.

Hoshino et al. [12] investigate the impact of memory layout on the performance of NVIDIA Kepler, Intel XeonPhi, and Intel Xeon processors, under directive-based programming languages. They found that having structure-of-arrays is much more efficient than array-of-structures under Kepler and XeonPhi, while it has minor impact on the performance of Xeon. They explain this by the relatively smaller cache employed by Kepler (110 Bytes per hardware thread) and XeonPhi (128 KBytes per hardware thread), compared to Xeon (1048 KBytes per hardware thread). They also introduce a new directive allowing the programmer to change the data layout of

multi-dimensional arrays.

Wienke et al. [20] compare the performance and development effort of two OpenACC applications to their equivalent OpenCL implementation. They measured the development effort by considering the modified code lines and found that OpenACC requires 6.5X lower development effort compared to OpenCL. They also reported the best-effort performance gap is of 2.5X. They found that this large performance gap is due to OpenACC's inability to exploit software-managed cache.

Kraus et al. [14] investigated the opportunity to improve the performance of CFD workloads through OpenACC. They applied several CUDA-like optimizations at the OpenACC level, including texture cache and occupancy optimizations. They apply texture memory optimization by declaring variables as constant. They alter the streaming multiprocessor's occupancy by specifying vector length (or thread-block size). They found that the optimal occupancy is the point with higher cache hit rate, since the CFD workloads tend to work on large working sets. They also transform array-of-structures to structure-of-arrays to optimize memory layout (returned nearly 52% performance improvement).

Govett et al. [10] compare the performance of three different OpenACC implementations under NIM work- load. They perform three optimizations in their own implementation, called F2C-ACC. Among these optimizations, they found that variable demotion technique can improve performance significantly. Variable demotion avoids transferring the entire dimension of an array when only certain indices are accessed. This can decrease the memory transfer time and also allow generation of more efficient kernel code. For instance, variable demotion on a 1D array, where possible, can replace global memory array accesses with scalar or register accesses.

Hoshino et al. [11] studied the performance of two OpenACC microbenchmarks and one real world CFD application. They examined the common and application-

specific optimization techniques for OpenACC and CUDA. They found that the current OpenACC compilers achieve about 50% to 98% of performance of the CUDA versions depending on the compiler.

Chapter 4

Proposed Compression Clauses and Implementation

In this chapter, we introduce our new OpenACC clauses by which programmers can reduce the memory bandwidth load of their program. Using this programming interface programmers can take advantage of a set of under-the-hood API functions and kernel extensions which will be explained thoroughly later in this chapter.

4.1 Proposed Compression Clauses

Applying the proposed data transfer clauses on a data set compresses it on the host, and transfers the compressed data set to the device. Consequently, less traffic is generated on memory interconnection network when the kernel tries to access the compressed data. Also transferring data from host to device (and vice versa) becomes much less time consuming.

The proposed clauses should be used together and in two different stages of the OpenACC code; **i. Data transfer**, **ii. Kernel**.

4.1.1 Data Transfer Clauses

Any data set used in the kernel regions can be transferred to the accelerator device by one of the OpenACC clauses listed in the left column of table 4.1. To use compression on a data set, the data must be first copied by one of the proposed clauses in the right column of table 4.1 instead of the regular data transfer clauses in the left column. Using `ccopyin` and `ccopy` makes the compiler call an alternative API function, which compresses the data on the host, before copying it to the device. In the meantime other modifications may be done on the data (e.g. changing the array of structs to struct of arrays) so that it makes the compression more effective. If `ccopy` or `ccopyout` is used, the data set is copied back to the host memory and gets decompressed there.

In addition to the data transfer clauses' predefined parameters, we also introduce two optional input parameters (Min and Max) for data transfer clauses. The compression method we use needs the maximum and minimum element of the data set to operate. Calculating these two parameters causes an overhead. However, in order to eliminate it, programmers can specify the range of the data set by passing these two parameters, and therefore prevent the unnecessary calculations. Passing these parameters is mandatory if there is a possibility that the range of the data set can vary during the application execution, or if `ccopy` is being used. Since for the correct functionality, the compression method needs to know the range of the data set throughout the program execution. The programmer is responsible to make sure that the program is informed of the accurate range by passing maximum and minimum with safe margin. Note that the more tight this range is, the more precise the compression method. Hence, the programmer should specify the narrowest possible range. Below we show an example of applying compression on a data set, named *cdata*:

compression_copy(cdata[0:SIZE:MIN:MAX])

In this example MIN and MAX are the minimum and maximum of the data set.

Table 4.1: Compression Copy Clauses

Regular Data	**Compressed Data**
`copyin`	`compression_copyin(ccopyin)`
`present_or_copyin` (`pcopyin`)	`present_or_compression_copyin` (`pccopyin`)
`copy`	`compression_copy(ccopy)`
`present_or_copy` (`pcopy`)	`present_or_compression_copy` (`pccopy`)
`copyout`	`compression_copyout(ccopyout)`
`present_or_copyout` (`pcopyout`)	`present_or_compression_copyout` (`pccopyout`)

4.1.2 Compression Clause

After the compressed data is copied to the device, the compiler must be instructed to generate kernels which work properly with the compressed data. In fact, kernels need to decompress the compressed data before the first use. Therefore, we introduce `compression` clause for this purpose. Pointers to the compressed data sets must be marked by the `compression` clause on the `kernels` or the `parallel` directive. While `compression` can be used on `kernels` directive, the data transfer clauses can be used on both `kernels` and `data` directives. For instance, listing 4.1 shows the OpenACC implementation of matrix-matrix multiplication using our proposed clauses. `pccopyin` is used on `data` directive to transfer matrices a and b. Also the `kernel` directive is annotated by `compression` clause with a and b matrix pointers.

There is a reason why we introduce a separate clause (`compression`) for the `kernels` directive. In an application, the codes related to data transfer and the codes of the kernel may be listed in two different files. Hence, compilation of those files may be done separately. In order to compile those files correctly, the compiler should be hinted about the compression enhanced data sets in each file distinctly.

Listing 4.1: OpenACC Matrix-Matrix Multiplications Using Compression

```
#pragma acc data pccopyin(a[0:SIZE*SIZE],b[0:SIZE*SIZE])\
    pcopyout(c[0:SIZE*SIZE])
#pragma acc kernels compression(a,b)
#pragma acc loop independent
for (i = 0; i < SIZE; ++i) {
#pragma acc loop independent
   for(j=0; j<SIZE; j++){
      float sum=0;
      for(l=0; l<SIZE; l++){
         sum += a[i*SIZE+l]*b[l*SIZE+j];
      }
      c[i*SIZE+j]=sum;
   }
}
```

4.2 Compression Implementation

In this section we present the compression method we use in this work. By using the programming interface introduced in the previous section a compression scheme is initiated which can be applied on either 32 or 64 bit floating point data sets. What follows elaborates the underlying compression mechanisms which is implemented as an extension to IPMACC.

4.2.1 Compression

To compress the floating point numbers we use a flexible method which can pack any floating point number into an arbitrary small set of bits. In the first step we find the number which has the maximum absolute value in the data set. This step is done only if the programmer does not specify the data set range. We use the specified range to calculate the maximum absolute value in the data set if it is available. By dividing all the numbers in the data set by the maximum absolute value multiplied by two, we map them to the (-0.5,+0.5) range. We then add each number to 1.5 and change the range to (1,2). Accordingly, the exponent part of all these floating point numbers will be equal to 01111111 in case of single precision and 01111111111 in case of double precision. Therefore the exponent part can be omitted from the number representation, without losing any information. Then, we only keep a limited number of mantissa's significant bits to compress data. For instance, to apply a compression ratio of 2 to a single precision floating point number, the seven least significant bits of mantissa should be thrown away. At the end, 16 bits remain which are stored as the compressed format of a 32-bit floating point number. Code listing 4.2 shows the compression method implementation for compression ratio of 2.

Figure 4.1 is an example of compressing a 32-bit floating number which in this

case is Pi. Figure 4.1.1 shows the most accurate binary representation of Pi in single precision floating point format. Assuming that the maximum absolute value of the numbers is 4.0, numbers of the data set must be divided by 8.0 (4.0×2) and added to 1.5, so that they are in the range (1,2). Figure 4.1.2 shows the Pi mapped to (1,2) in binary format. After shifting the bits 7 times and truncating the exponent and the sign bits, 16 bits remain as the compressed number (Figure 4.1.3).

Listing 4.2: Compression Implementation in C

```
for ( i = 0; i < arraySize/sizeof(float); i++) {
   float temp = floatData[i]/(2*maxAbs) + 1.5;
   twoByteDataPtr[i] =( (*((unsigned int*)&temp) ) >>7) & 0x0000FFFF;
 }
```

This compression method is also implemented as a device function which is called in kernels to compress data before it is written to the global memory. Memory writes can be detected by assignment operators in the code. For writing compressed datasets elements to memory in kernel codes, the right-hand side expression is passed to the device function in code listing 4.3 prior to the writes, and the returned value which is compressed is written to the memory. These function calls are added to the code by the IPMACC code generator.

Listing 4.3: Compression Implementation in CUDA

```
__device__ __inline__ unsigned short compress_float(float fNum, float*
    coef){
   float temp;
   temp = fNum*coef[2]+1.5;
   return (unsigned short)(((*((unsigned int*)&temp))>>7)&0x0000FFFF);
}
```

4.2.2 Decompression

In order to decompress a compressed number, we shift its bits to left so that it forms 23 or 52 bit mantissa for single and double precision floating points, respectively. The new bits that fill the least significant part are a 1 followed by 0s. The reason why we do so is that each compressed data represents a range of uncompressed floating point numbers. If we fill all of the least significant bits with zeros, the decompressed number would be the first number of the range. However, now that we start the least significant part with 1, the decompressed number is the center of the range and is probabilistically a better approximation of the original uncompressed number. After filling the exponent part by the values which have been omitted in the compression stage we have a floating point number in the range between one to two. By subtracting the number by 1.5 the numbers range changes back to (-0.5,0.5). In the last step, we multiply this by the data set maximum absolute value multiplied by two. The result is an approximation of the original number. Figures 4.1.4 and 4.1.5 illustrate the decompression of Pi. The white colored bits in figure 4.1.4 show the lost least significant bits are replaced. Finally, figure 4.1.5 shows the fully decompressed Pi number. The 8 bits (orange colored) are the ones in which decompressed Pi and the original single precision Pi differ.

In order to optimize the decompression function we change the order of operations. Instead of performing subtraction by 1.5 and multiplication we distribute the multiplication as indicated in equation 4.1. This way the compiler replaces the subtraction and multiplication instructions with a single fused multiply-add instruction [19]. Hence, two constant values, named key_1 and key_2, are needed for decompression as indicated in equation 4.2. We calculate them on the host at the compression stage and copy them on the constant memory. There is a total of 64 KB constant memory on a GPU device which is accessed through an 8KB cache on each SM. A 4-byte data

in constant cache can be broadcast among threads with a very low latency.[3, 2]

$$2MaxAbs \times (X - 1.5) = 2MaxAbs \times X - 2MaxAbs \times 1.5 \tag{4.1}$$

$$= key_1 \times X + key_2 \tag{4.2}$$

where

$$key_1 = 2MaxAbs,$$

$$key_2 = -3MaxAbs$$

Code listing 4.4 is the serial implementation of the decompression in C. It is part of the IPMACC runtime API and decompresses the compressed datasets after kernel execution and being copied back to CPU memory. Code listing 4.5 shows a device function that performs the decompression on a 4-byte floating point number compressed in a 2-byte format. In the compilation stage, all the memory accesses which have a reference to a compressed dataset are replaced by a device function call. Hence, before using the compressed data element, it is decompressed and returned by this function. Then the returned value can be used in the rest of the kernel code. All of these modifications are performed by the IPMACC code generator in the compilation stage.

Listing 4.4: Decompression Implementation in C

```
for(i = 0; i< arraySize/sizeof(float); i++){
   unsigned int temp = ((unsigned short*)twoByteDataPtr)[i];
   temp = (temp<<7) | 0x3F800000;
   floatData[i] = *((float*)&temp) * key[0] + key[1];
}
```

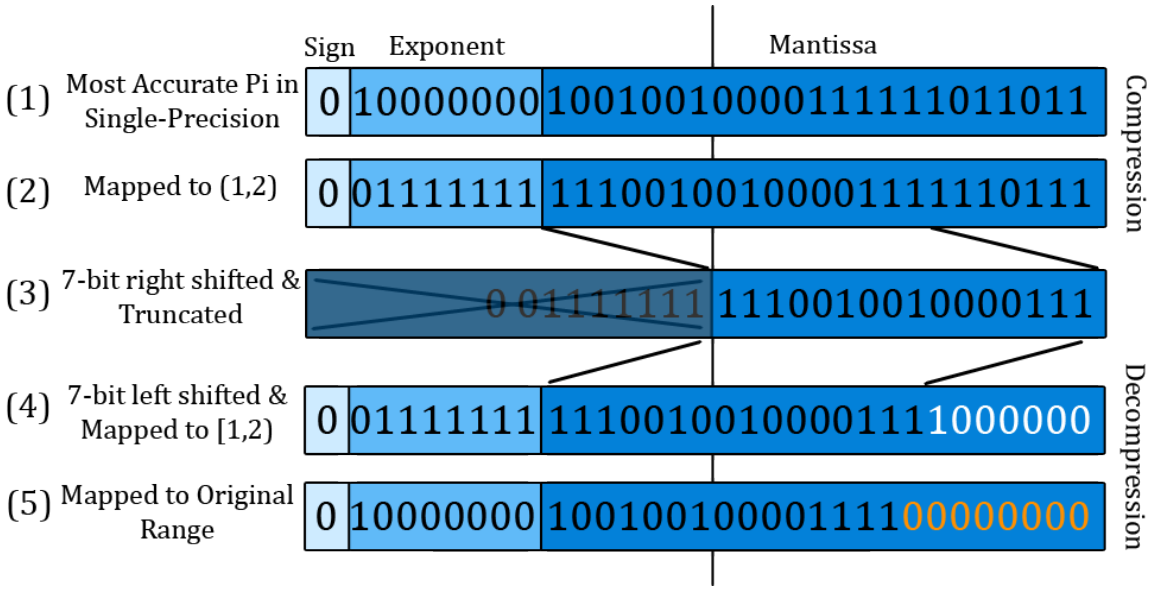

Figure 4.1: Illustration of Compression and Decompression methods

Listing 4.5: Decompression Implementation in CUDA

```
__device__ __inline__ float decompress(void *ptr,int index,float* coef){
   unsigned int temp = ((unsigned short*)ptr)[index];
   temp = (temp<<7) | 0x3F800000;
   return *((float*)&temp) * key[0] + key[1];
}
```

4.3 Runtime Library Extension

We extended the OpenACC runtime library in order to add our intended compression feature to the OpenACC model. To this end, we added new routines to the library which handle the compression and decompression processes while transferring a dataset. In fact the new routines are evolved versions of existing routines which should be used in case of working with a dataset that is marked to be compressed. The table 4.2 contains the introduced routines and their regular counterparts.

Table 4.2: Runtime Routines for Compression

Regular Data	**Compressed Data**
`acc_copyin`	`acc_compress_copyin`
`acc_present_or_copyin`	`acc_compress_present_or_copyin`
`acc_create`	`acc_compress_create`
`acc_present_or_create`	`acc_compress_present_or_create`
`acc_copyout`	`acc_decompress_copyout`

Our compression method is implemented in IPMACC framework, and the source code is available on a GitHub repository[6]. The file named *openacc.c* in *src* directory contains all the implementations of the new routines.

Chapter 5

Methodology

5.1 Benchmarks

We use benchmark applications from Rodinia benchmark suit [9]. Rodinia benchmark suite consists of many scientific and engineering applications implemented for heterogeneous platforms in CUDA and OpenMP. A third party OpenACC implementation is also available [1]. In addition to Rodinia benchmark sute, we designed a set of microbenchmarks for further in detail investigations.

5.2 OpenACC Compiler

We use our framework, IPMACC[15], which is composed of a source to source compiler and a runtime system. The compiler translates OpenACC to either CUDA or OpenCL codes. Applications are executed over IPMACC runtime, which is built on CUDA and OpenCL runtime (e.g. NVIDIA GPUs or AMD GPUs).

5.3 Performance Evaluations

In order to validate the result correctness of programs compiled by our framework we compare the outputs by the CUDA and serial versions. We considered three metrics in performance evaluations; total kernel execution time, application run-time and compression overhead time. We use *nvprof* [4] for measuring kernel execution time while system time is used for the rest of time measurements through POSIX time library. Reported results are based on average of multiple runs of the applications.

5.4 Platform

We evaluated our method using NVIDIA Tesla K20c as the accelerator. This system uses NVIDIA CUDA 6.0 [3] as the CUDA implementation backend. The other specifications of this system are as follows: CPU: Intel Xeon CPU E5-2620, RAM: 16 GB, and operating system: Scientific Linux release 6.5 (Carbon) x86 64.We use GNU GCC 4.4.7 for compiling C/C++ files.

Chapter 6

Experimental Results

In this chapter we show how our solution improves performance of different case studies. Firstly, in order to have a better understanding of hardware architecture and its impact on the efficiency of our method, we created a set of synthetic benchmarks. In these microbenchmarks we investigated the efficiency of our compression method on different data sizes and access patterns. We also compare the performance of baseline OpenACC and compression-enhanced OpenACC implementations of some real world applications. We use speedup as a metric to show the efficiency of our compression method in this chapter. Equation 6.1 shows the definition of speedup.

$$Speedup = \frac{NormalProgramExecutionTime}{EnhancedProgramExecutionTime} \tag{6.1}$$

6.1 Microbenchmarks

Prior to the real applications, we test our approach on some synthetic benchmarks. First of all by running these simple test cases, we confirm that the time saved by bandwidth usage reduction caused by data compression can outweigh its computation overheads. These test cases also examine the efficiency of reduced-size data sets over

different memory access patterns. Moreover, we investigated how data set size affects the impact of the compression. By analyzing the results of these experiments we can conceive the hardware behavior which helps us explain the results we get for real applications.

6.1.1 Simple Data Copy

In the first microbenchmark we simply copy a chunk of data from one part of GPU global memory to another part. Each thread is responsible for copying a single data element. So threads bring the data into the cores, and then copy it back to another address of the memory. We apply the compression clause on both source and destination data sets, and measure how it affects performance. Code listing 6.1 is the kernel region code used for this test case. We perform these experiments on different data sizes to understand how the compression method's efficiency changes over different data set sizes. Figure 6.1 compares the kernel execution time of the normal microbenchmark and its compression enhanced implementation. As the figure shows data compression does not have a significant impact on the kernel execution time up to a certain data size. But after that point there is a jump in the normal kernel time, and we can see a huge gap between normal and compression enhanced kernel times. At a data size twice the normal kernel jump we can see a change in the slope of the line representing the compression kernel time. Although this change is not as drastic as the normal kernel jump, it is caused by the same reason. Taking into account the size of the data sets we figure out that the drastic changes happen when the size of the data the kernel is dealing with, becomes larger than the capacity of the L2 cache. At the data copy stage, when the data is being copied from the host to the device, it goes through the L2 cache. So a portion of data is cached during the data copy stage. Although each data element is accessed only once in the kernel in this

microbenchmark, cold misses happen only to the data which is not cached during the copy stage.

To ensure that the L2 cache causes the observed phenomena, we tweak the benchmark, so that it copies an enough large dummy data into the GPU memory to flush the L2 cache. By doing this all the kernel's memory accesses result in cold misses. The impact of the L2 flushing on the kernel execution time can be seen in figure 6.1.

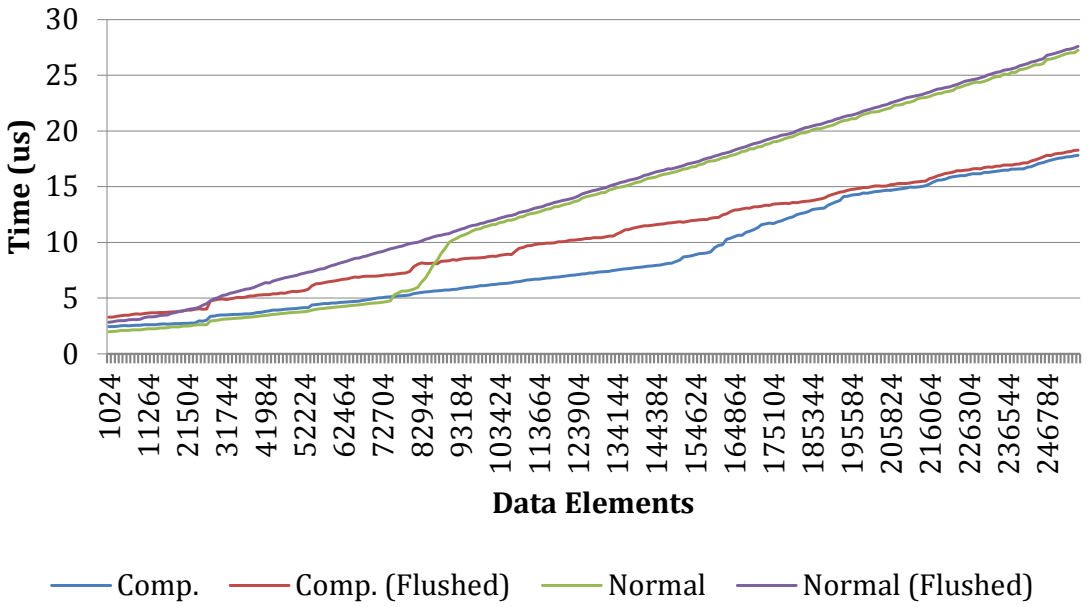

Figure 6.1: Simple Data Movement (each data element is a 8 bit double precision floating point number)

Listing 6.1: Simple Data Movement

```
#pragma acc data pccopyin(dsrc[0:size]) pccopyout(ddst[0:size:min:max])
#pragma acc kernels present(dsrc,ddst) compression(dsrc,ddst) \
        pcopyin(flush[0:10000000])
#pragma acc loop independent
     for(int i=0;i<size;i++)
              ddst[i]=dsrc[i];
```

6.1.2 Simple Data Copy in Reverse Order

By increasing the data size, shortly after the L2 cache size, we no longer can see the impact of L2 cache even in the test cases without the dummy data copy. From results shown in figure 6.1 we can infer that the cached portion of data is never used, and instead the global memory is accessed for all the memory requests. The uncached chunks of data are accessed before the cached data in the L2, which evicts the initial cached data in the L2 cache. To explain this behavior we assume that in the copy stage the head of the data is copied first, and the tail is copied at the end. So if the data size is less than the L2 cache size, the whole data is cached. But if the size of the data, which is being copied, is larger than the L2 capacity, only a chunk of data with the same size as the L2 remains cached. If we assume that the active threads access the data set in the same order, we can explain the eviction of the initially cached data.

To prove our assumptions we modify the previous benchmark so that threads access the memory in the reverse order. For example if a thread was supposed to access the ith data element in the previous benchmark, now it has to access the *size-i* th data element in the modified benchmark. In figure 6.2 we compare kernel execution times of the reverse ordered kernels. As can be seen, after the point that the data set sizes become larger than the L2 cache size, the slope of the line increases. But the execution time of unflushed kernels never reaches the flushed kernels, either in compression enhanced kernels or normal kernels. It proves the assumption that the threads which access the head of the data execute at beginning of the kernel. Our modifications make the threads to start the kernel from data set's tail which is cached in the L2, and this prevents wasting the initial data in the L2 which is cached at the copy stage.

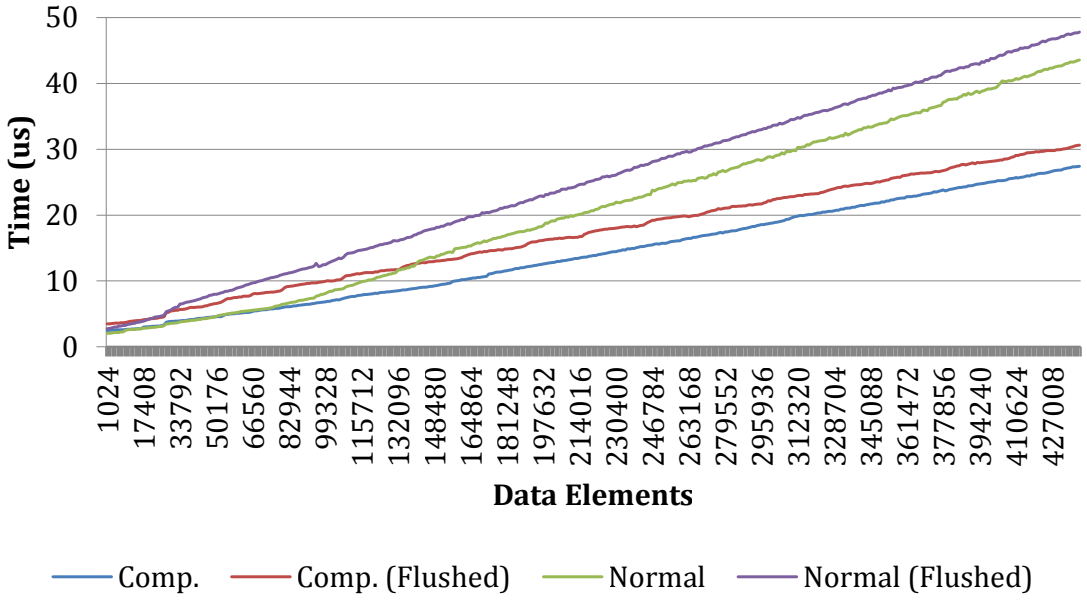

Figure 6.2: Simple Data Movement in Reverse Order (Each data element is a 8 bit double precision floating point number)

6.1.3 Strided Access Pattern

In this microbenchmark we test compression enhanced kernels with strided memory access pattern. We try different cases with different stride lengths. Subsequent threads access data elements distanced as much as the stride length, and copy it back to the destination data set. We apply compression to both source and destination data sets. Figure 6.3 compares the performance improvement achieved for different stride lengths, over different data sizes. As can be seen as the stride length grows the compression gain declines. The maximum speedup for each stride length happens at the point in which the uncompressed data sets do not fit in the L2 cache, but still the compressed data sets size is smaller than that.

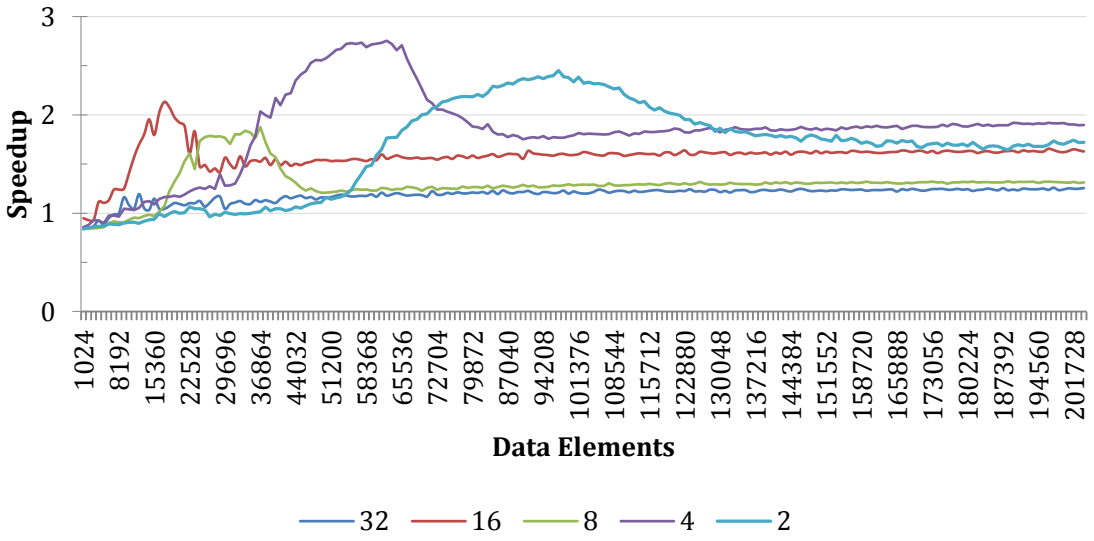

Figure 6.3: Strided Access Pattern (Each data element is a 8 bit double precision floating point number)

6.1.4 Random Access Pattern

In the previous microbenchmarks the data was simply copied from one address to another location in the memory, and the order of elements was preserved. In the real world applications the data access patterns are not always as simple. To test our compression method against complex memory access patterns we modify the previous benchmarks so that each thread accesses a random data element in the source data set, and copy it to ith location (i is the thread id). To implement this test case we initialize an array of integers with random numbers between 0 to the source array size minus one, and copy it to the accelerator memory. Threads should first read their corresponding random number, and based on that number they pick up a random element in the source array.

As can be seen in figure 6.4 up to a certain point there is no significant improvement over the normal kernel time which indicates that most of the requested data is found in the L2 cache. Our experiments also show that flushing the L2 cache does

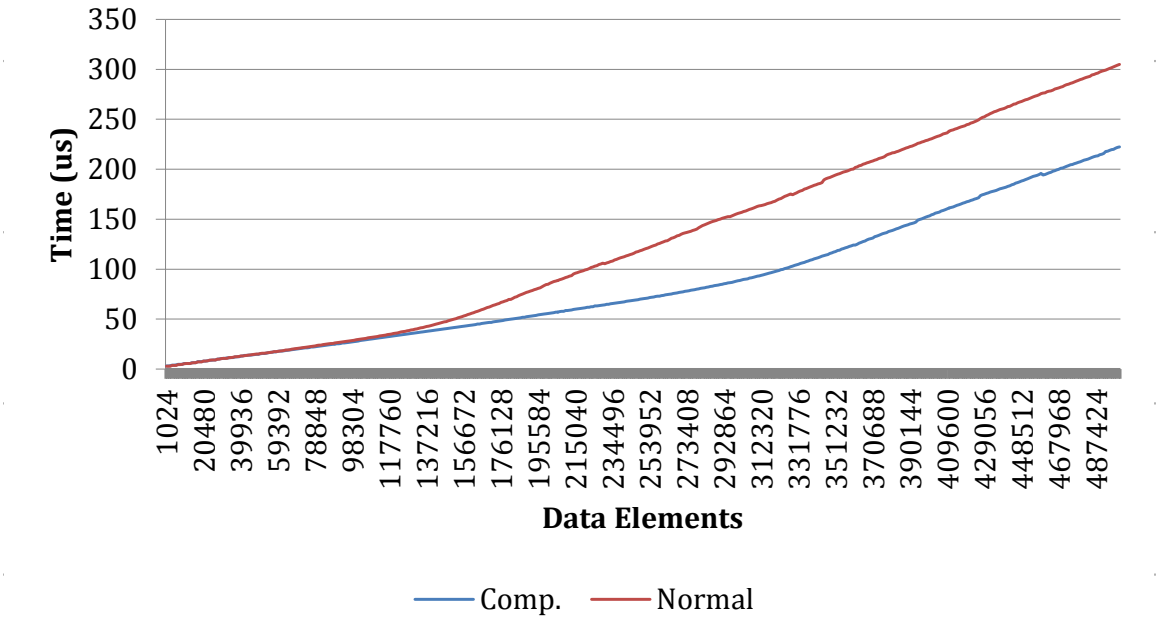

Figure 6.4: Data shuffling (Each data element is a 8 bit double precision floating point number)

not have any negative effect on this high hit rate. Hence, we can infer that the data is cached during the kernel execution by the pioneer threads, and the other threads use the cached data. The reason why only a few pioneer threads can bring almost all the data into the cache can be explained by the access pattern which is scattered randomly all over the data set, and the fact that the smallest memory transaction brings at least 32 bytes of data to the L2 cache. As long as the source data set is smaller than the size of cache, accessing it does not cause cache eviction. The lines of the L2 cache start being evicted only after the point that the size of the source data set becomes larger than the L2 cache size. Then the hit rate decreases, and we can see the execution time starts increasing with a steeper slope. As could be predicted this change happens to the compression enhanced kernel when the number of elements it is dealing with is twice greater than the number of elements which causes the slope change to the normal kernel.

6.2 Real World Benchmarks

6.2.1 Vector Product

Vector product is a simple benchmark in which corresponding elements of two vectors are multiplied, and the result is written in a new array. Code listing 6.2 shows the accelerator enhanced part of this benchmark. We applied compression to all vectors.

Listing 6.2: Vector Product

```
#pragma acc data pccopyin(a[0:SIZE],b[0:SIZE]) pccopyout(c[0:SIZE:MIN:MAX])
#pragma acc kernels compression(a,b,c)
#pragma acc loop independent
for (i = 0; i < SIZE; ++i){
        c[i] = a[i] * b[i];
}
```

In figure 6.5 we report the kernel time improvement achieved by using data compression. As the figure illustrates for the matrices with dimension sizes of 256 and larger the compression can help improving the performance up to 2.07x. The reason why we see a speedup of higher than 2x is that in the test case with vector size of 131072 elements, the L2 cache can not store all the vectors' data. However, by using the compression method the size of the vectors data shrinks to half which fits into the L2 cache. Hence, compression kernel takes advantage of the low-latency L2 cache accesses as well as less bandwidth usage.

6.2.2 Matrix Addition

Matrix addition is the operation of adding two matrices by summing the corresponding entries. We apply the compression to both input matrices and also the result

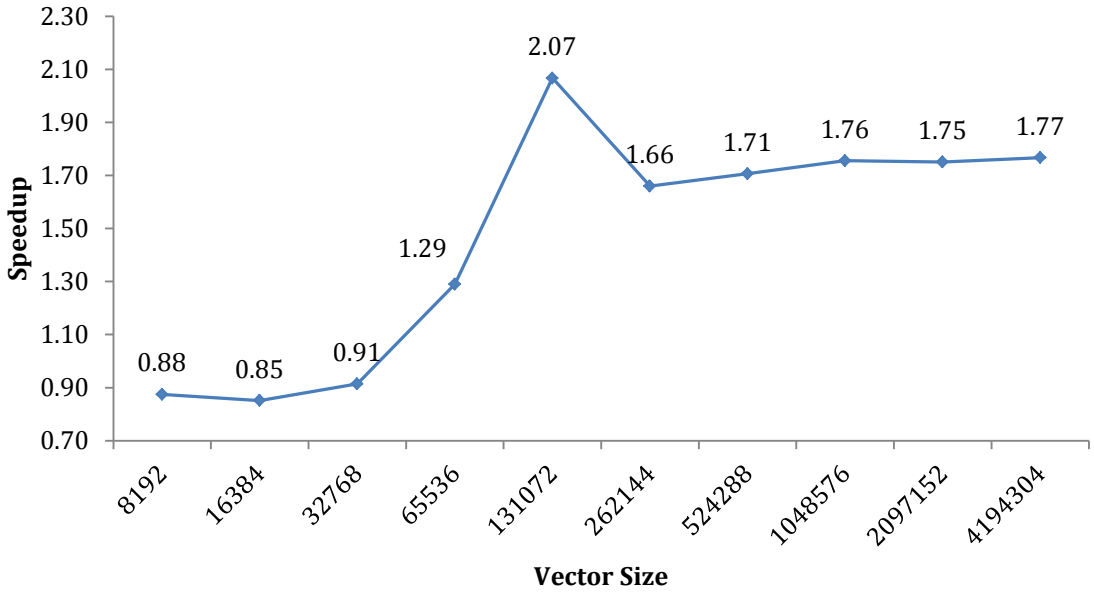

Figure 6.5: Vector Product Kernel Time Improvement

matrix. Code listing 6.3 contains the compression enhanced code we use. Since there is no dependency between the iterations of the loops, we marked both of them with `independent` clause. Therefore, each iteration is done by a single thread.

Listing 6.3: Matrix Addition

```
#pragma acc data pccopyin(a[0:SIZE],b[0:SIZE]) pccopyout(c[0:SIZE:MIN:MAX])
#pragma acc kernels compression(a,b,c)
#pragma acc loop independent
for (i = 0; i < SIZE; ++i){
#pragma acc loop independent
        for (j = 0; j < SIZE; ++j){
                c[i * SIZE + j] = a[i * SIZE + j] + b[i * SIZE + j];
        }
}
```

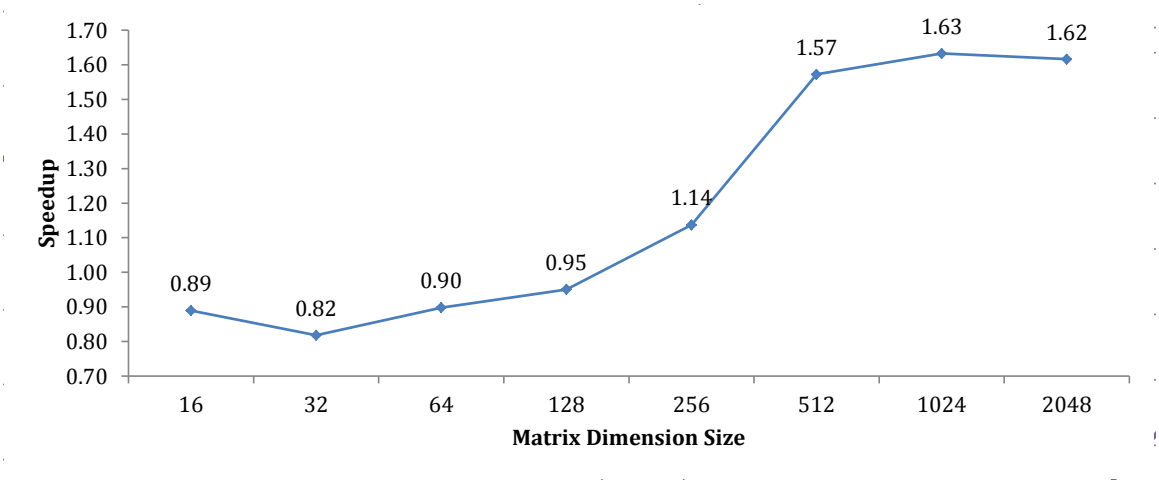

Figure 6.6: Matrix Addition Kernel Time Improvement

In figure 6.6 we report the kernel time improvement achieved by using the data compression. As the figure illustrates for the matrices with dimension sizes of 256 and larger the compression can help improving the performance up to 1.63x.

6.2.3 Matrix Multiplication

Matrix multiplication is one of the basic operations in scientific computations. We simply add OpenACC annotations to the serial code and apply our compression method to the program. The computational phase in the serial code consists of three nested *for* loops which are marked by the `kernels` directive to be executed on the accelerator. Two outer *for* loops are also annotated by the `loop` directive so that the work of each iteration is shared between threads. The `independent` clause, which is used with the `loop` directive, is a hint to the compiler not to check for dependencies between iterations of the loops. The inner *for* loop iterations are executed sequentially by a single thread.

In figure 6.7 we report the kernel time improvement achieved over the baseline OpenACC implementation by using the compression method. Compression becomes highly effective on matrix dimensions of larger than 64. Figure 6.8 illustrates the total

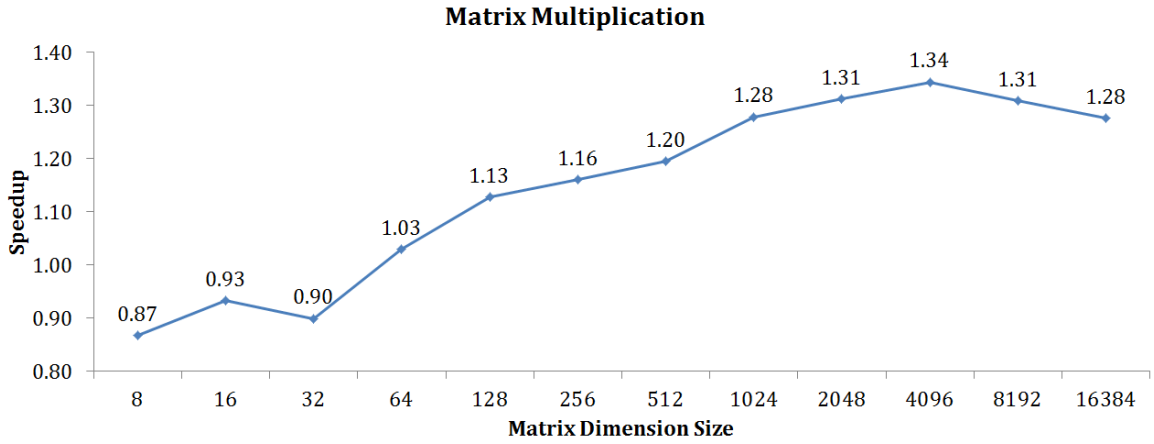

Figure 6.7: Matrix-Matrix Multiplication Kernel Time Improvement

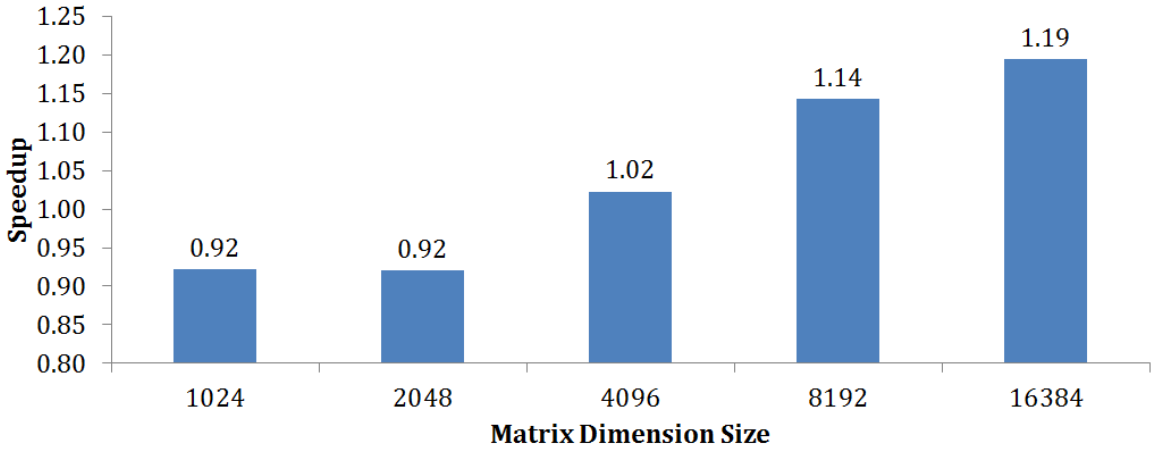

Figure 6.8: Matrix-Matrix Multiplication Application Run-Time Improvement

application run-time improvement. Total run-time consists of memory allocations, data initializations and transfers, compression time, and kernel time. We can see that for matrices with dimensions greater than 4096 the compression overhead is compensated. The breakdown of total baseline and compression-enhanced application run-time is reported in figures 6.10 and 6.9. Kernel time constitutes a larger portion of application run-time as the matrices sizes grow.

We also measured the error caused by compression. Different matrices composed of random numbers of various ranges were generated to test the accuracy of the com-

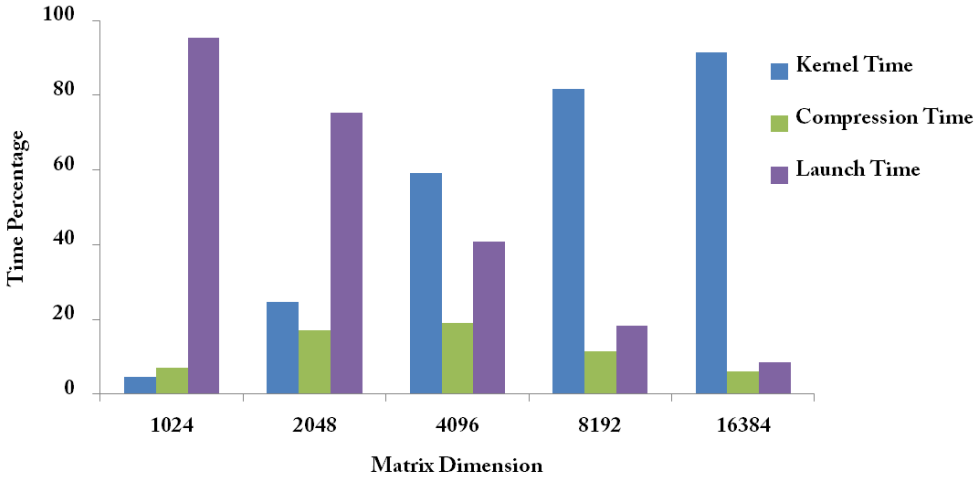

Figure 6.9: Compression-Enhanced MM Multiplication Breakdown

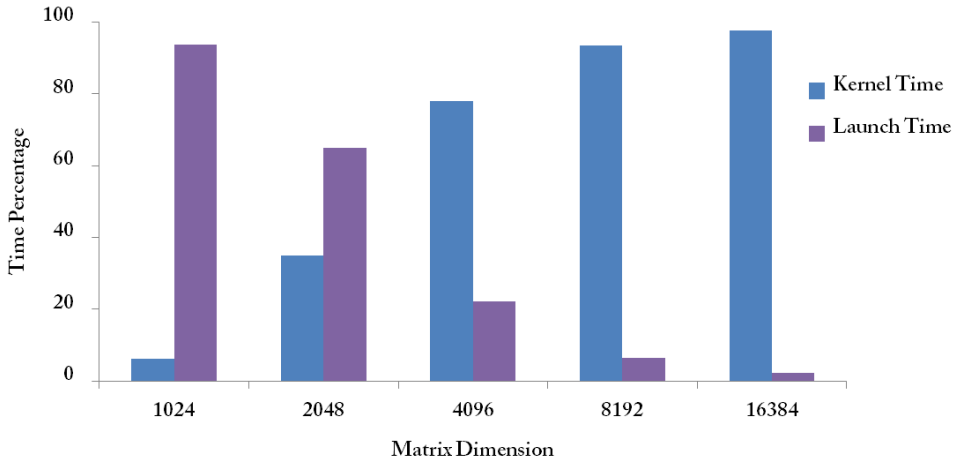

Figure 6.10: Baseline MM Multiplication Breakdown

pression method. We compared the results for our application and its compression-enhanced version. The geometric mean of relative errors is 0.006% with the standard deviation of 3^{-4}%, and the maximum measured error is equal to 0.01%. These errors are independent of input numbers ranges and size of input matrices.

6.2.4 HotSpot

HotSpot is commonly used to measure processor temperature under different architectural features and power consumptions. HotSpot performs a thermal simulation

by solving a set of differential equations for a block of cells iteratively. Each cell's temperature in the computational grid is associated with the average temperature value of the chip area it represents.

We perform our experiments on grid sizes ranging from 64 cells in each dimension to 1024. This benchmark contains two data sets, cells' temperatures and cells' powers, both stored in the double precision floating point format. We evaluated compression applying it on both data sets. But we ended up using compression only on the powers data set. We further discuss in the section 6.3. We used the compression clause on the computation phase kernel. Figure 6.11 illustrates performance improvement for the compression-enhanced kernel.

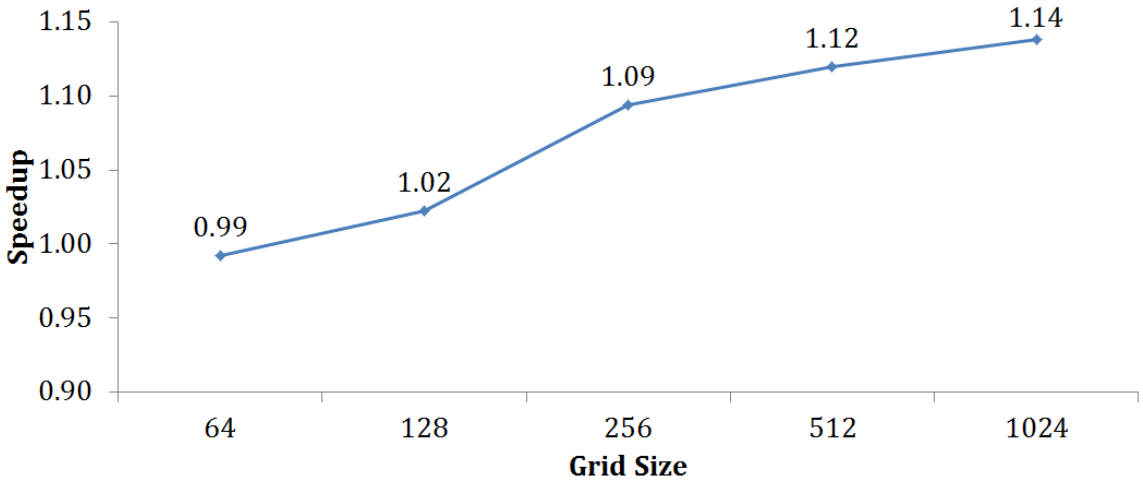

Figure 6.11: HotSpot Kernel Performance Improvement

In each iteration of the program the kernel is launched once. Figure 6.12 compares the run-time of the application, simulating a grid size of 512, in the course of the first 700 iterations. The reported time includes data transfer times and device initializations. The compression overhead explains why the compression-enhanced application run-time is initially above the baseline. The accuracy loss of the final result caused by data compression is negligible. The maximum relative error is $10^{-8}\%$ after 4000 iterations.

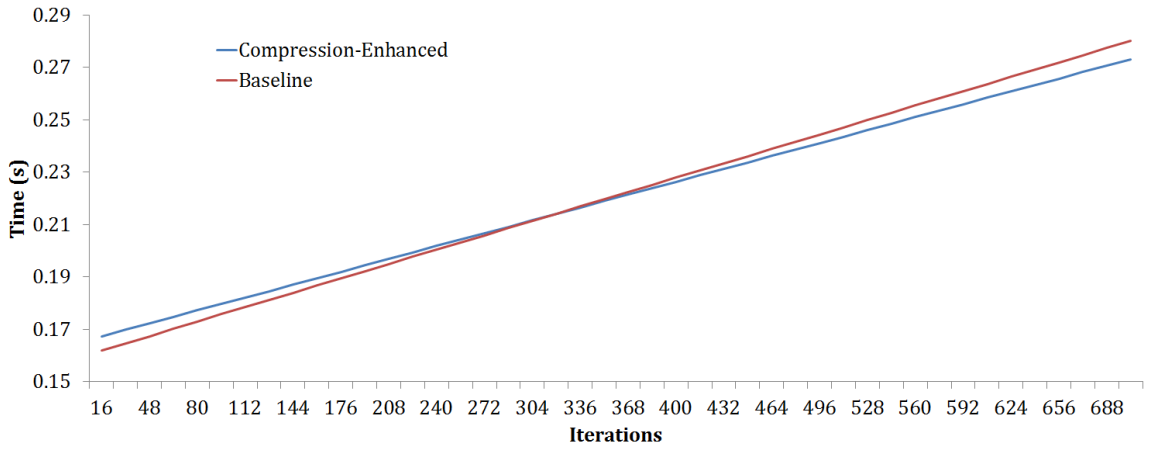

Figure 6.12: HotSpot Application Run-Time for First 700 Iterations

Listing 6.4: HotSpot Kernel

```
#pragma acc kernels present(temp, power, result)
    compression(temp,power,result)
#pragma acc loop independent
for (r = 0; r < row; r++) {
#pragma acc loop independent
        for (c = 0; c < col; c++) {
                /*      Corner 1        */
                if ( (r == 0) && (c == 0) ) {
                        delta = (step / Cap) * (power[0] +
                                        (temp[1] - temp[0]) / Rx +
                                        (temp[col] - temp[0]) / Ry +
                                        (amb_temp - temp[0]) / Rz);
                }       /*      Corner 2        */
                else if ((r == 0) && (c == col-1)) {
                        delta = (step / Cap) * (power[c] +
                                        (temp[c-1] - temp[c]) / Rx +
                                        (temp[c+col] - temp[c]) / Ry +
```

```
                        (amb_temp - temp[c]) / Rz);
}       /*      Corner 3        */
else if ((r == row-1) && (c == col-1)) {
        delta = (step / Cap) * (power[r*col+c] +
                        (temp[r*col+c-1] - temp[r*col+c]) / Rx
                           +
                        (temp[(r-1)*col+c] - temp[r*col+c]) /
                           Ry +
                        (amb_temp - temp[r*col+c]) / Rz);
}       /*      Corner 4        */
else if ((r == row-1) && (c == 0)) {
        delta = (step / Cap) * (power[r*col] +
                        (temp[r*col+1] - temp[r*col]) / Rx +
                        (temp[(r-1)*col] - temp[r*col]) / Ry +
                        (amb_temp - temp[r*col]) / Rz);
}       /*      Edge 1 */
else if (r == 0) {
        delta = (step / Cap) * (power[c] +
                        (temp[c+1] + temp[c-1] - 2.0*temp[c])
                           / Rx +
                        (temp[col+c] - temp[c]) / Ry +
                        (amb_temp - temp[c]) / Rz);
}       /*      Edge 2 */
else if (c == col-1) {
        delta = (step / Cap) * (power[r*col+c] +
                        (temp[(r+1)*col+c] + temp[(r-1)*col+c]
                           - 2.0*temp[r*col+c]) / Ry +
                        (temp[r*col+c-1] - temp[r*col+c]) / Rx
```

```
                        +
                    (amb_temp - temp[r*col+c]) / Rz);
}       /*      Edge 3 */
else if (r == row-1) {
        delta = (step / Cap) * (power[r*col+c] +
                    (temp[r*col+c+1] + temp[r*col+c-1] -
                        2.0*temp[r*col+c]) / Rx +
                    (temp[(r-1)*col+c] - temp[r*col+c]) /
                        Ry +
                    (amb_temp - temp[r*col+c]) / Rz);
}       /*      Edge 4 */
else if (c == 0) {
        delta = (step / Cap) * (power[r*col] +
                    (temp[(r+1)*col] + temp[(r-1)*col] -
                        2.0*temp[r*col]) / Ry +
                    (temp[r*col+1] - temp[r*col]) / Rx +
                    (amb_temp - temp[r*col]) / Rz);
}       /*      Inside the chip */
else {
        delta = (step / Cap) * (power[r*col+c] +
                    (temp[(r+1)*col+c] + temp[(r-1)*col+c]
                        - 2.0*temp[r*col+c]) / Ry +
                    (temp[r*col+c+1] + temp[r*col+c-1] -
                        2.0*temp[r*col+c]) / Rx +
                    (amb_temp - temp[r*col+c]) / Rz);
}
/*      Update Temperatures   */
result[r*col+c] =temp[r*col+c]+ delta;
```

```
        }
}
```

6.2.5 Nearest Neighbor

Nearest Neighbor (NN) finds the k-nearest neighbors of a point from a data set in a two dimensional space. The serial version calculates distances to all the records and finds the k nearest neighbors. The OpenACC version performs distance calculations on the accelerator in parallel and the host's master thread selects the k nearest neighbors. The input data set consists of many records and is in fact an array of structures (AoS). Each record is an object of a C *struct* which has two attributes, latitude and longitude.

In order to maximize bus utilization, all the memory requests from a warp should access neighbor bytes of memory. If each thread requests a 4-byte data and the address of the first thread's requested data is aligned, then all requests can coalesce into one memory transaction. In this case the bus utilization is 100%. To this end, it is highly recommended to change the arrangement of arrays of structures to structure of arrays (SoA) in SIMD architectures. This enhances spatial locality significantly.

Our compiler provides a transparent and low overhead data transformation on arrays of structures. This data transformation can be applied to an array while it is being compressed. In the compression stage each floating-point object is copied to a new smaller array after being compressed. Array transformation only changes the location of the compressed data in the new array, and, therefore, adds a minimal overhead to compression. Figure 6.13 demonstrates the kernel time improvement using compression with and without array transformation. We can achieve 1.36X speedup if we apply compression and data transformation. Since, the kernel time

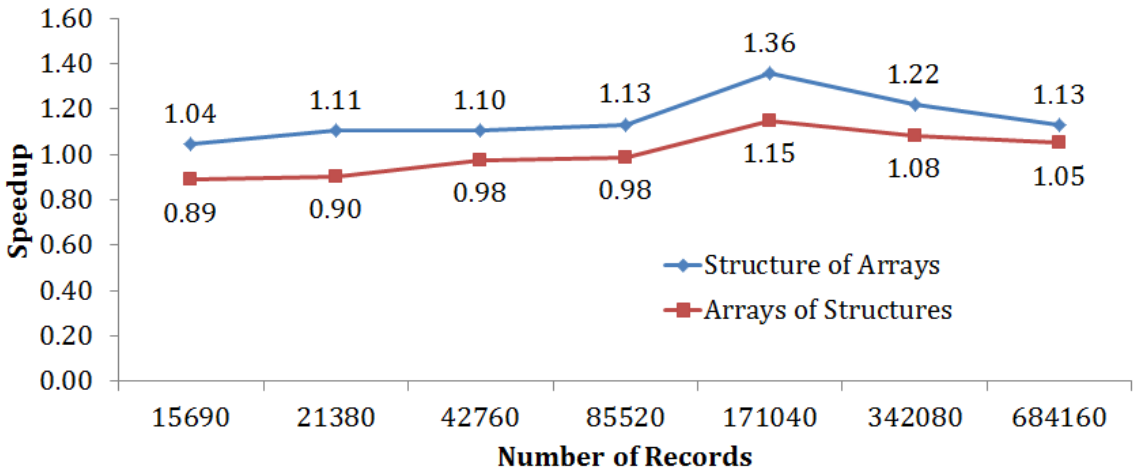

Figure 6.13: NN Kernel Time Improvement

constitutes an insignificant portion of the application run-time, we do not report speedup over that.

We measured the geometric mean of errors through all the calculated distances as 0.1% and the maximum relative as 1%. The standard deviation of the errors were calculated as 3^{-1}%.

Code listing 6.5 illustrates the code we use to apply the compression on the kernel region.

Listing 6.5: Nearest Neighbor Kernel

```
#pragma acc kernels ccopyin(locations[0:numRecords])
    copyout(distances[0:numRecords]) compression(locations)
#pragma acc loop independent
for (i=0; i<numRecords; i++) {
        LatLong latlong = locations[i];
        distances[i] = (float)sqrt((lat-latlong.lat)*(lat-latlong.lat)
        +(lng-latlong.lng)*(lng-latlong.lng));
}
```

6.2.6 Dyadic Convolution

Dyadic Convolution is an algebra operation calculating the XOR-convolution of two sequences. The OpenACC implementation parallelizes output calculations, where each thread calculates one output element. Although this implementation is fast to develop, it exhibits a high number of irregular memory accesses. We applied the compression clause on the main data set. Figure 6.14 shows the kernel time improvement for different data set sizes, using compression. Code listing 6.6 illustrates how we apply compression clauses on *h_Data* data set.

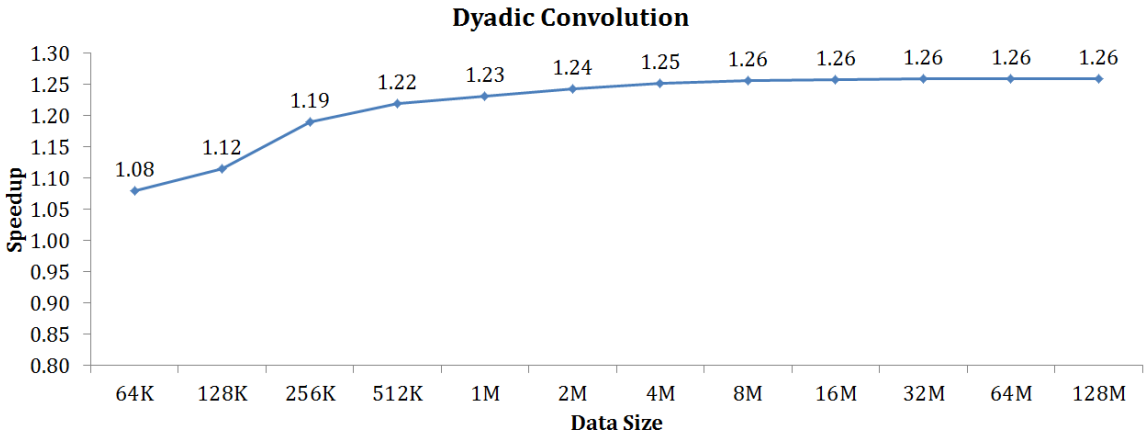

Figure 6.14: Dyadic Convolution Kernel Time Improvement

Listing 6.6: Dyadic Convolution Kernel

```
#pragma acc kernels pccopyin(h_Data[0:dataN]) pcopyin(h_Kernel[0:kernelN])
    pcopyout(h_Result[0:dataN]) compression(h_Data[0:dataN])
#pragma acc loop independent
for (int i = 0; i < dataNL; i++){
        double sum = 0;
        for (int j = 0; j < kernelNL; j++){
                sum += h_Data[i ^ j] * h_Kernel[j];
        }
```

```
        h_Result[i] = (float)sum;
}
```

6.3 Discussion

According to our evaluations applying compression clauses is effective in reducing memory access latencies and traffic in many cases. However, there are a few cases in which the compression or decompression overhead negates the improved memory latencies, and therefore no speedup can be achieved. For instance, applying compression on cells' temperatures data set in HotSpot benchmark is not effective. The reason is that each thread is responsible to calculate the new temperature of a cell using neighbor cells temperatures. Therefore, a cell temperature may be accessed by many threads. Once a block of data is accessed, it is copied to the caches so that further accesses to that block hit in the caches. Using compression decreases the memory latency of bringing data from memory. But it does not help in reading data from the caches, and it only imposes extra compression and decompression overhead. So except for the first thread that accesses a cell temperature, the others cause overheard without any gain, so long the cell data is not evicted in the cache.

Chapter 7

Conclusions

7.1 Conclusion and Future Work

In this thesis we introduced a set of OpenACC clauses to enable programmers to accelerate their codes at very low development effort. By using these facilities programmers are able to make use of a transparent compression method implemented in the compiler and the run-time system. They only need to mark the data sets by these clauses, and accordingly they reduce the memory latencies by decreasing the bandwidth usage. Although, our initial goal was decreasing the GPU global memory bandwidth usage, later we figured out the advantages of data compression is not limited to that. Reducing the size of the data results in having more data elements in the L2 cache which increases its hit ratio.

We applied our technique on a set of real world applications as well as the synthetic microbenchmarks which we designed to study our compression method efficiency on different memory access patterns. We achieve speedups up to 2X and 1.36X on GPU kernels from synthetic and real world applications respectively.

7.2 Future Work

There are many areas for further investigations as extensions of this work. One of these areas is the compression method. We proposed a low overhead compression method which is only applicable on floating point data type. There are many compression methods with particular characteristics which can be applied on specific data types. Each type of compression method is most effective, in terms of compression ratio, speed and accuracy, on a specific data pattern. Another possible extension to this work could be employing compression dedicated hardware. Compression dedicated hardware units can help achieving higher performance by performing compression and decompression calculations and not using the general purpose GPU cores. An adaptive system which is able to detect different data patterns, and use the best compression method dynamically during the program execution, can be a more robust solution.

We have observed that in cases that the requested data is available in the cache the access latency is not high enough to accommodate room for improvement. In cases such as the temperature data set in HotSpot benchmark, which most of the compressed data accesses hit in the cache, we did not observe any performance improvement. One challenging issue is that the cache is a transparent unit in the memory hierarchy and we do not have control over its behavior. In such cases using shared memory could be a solution. Data sets can be decompressed and copied to the shared memory, after being fetched from memory and before being used for the calculations in the threads. In this scheme the decompression is done once for each element of the data set and extra overhead is prevented. Implementing this solution can be the next step to cover more applications, and achieve a higher performance.

Bibliography

[1] Modified rodinia benchmark suite, 2013. [online]. available: https://github.com/pathscale/rodinia.

[2] Nitin gupta, what is constant memory in cuda. available: http://cuda-programming.blogspot.ca/2013/01/what-is-constant-memory-in-cuda.html.

[3] Nvidia corporation, .cuda toolkit 6.0,. 2014. available: https://developer.nvidia.com/cuda-downloads.

[4] Nvidia corporation, profiler's user guide, 2014. available: http://docs.nvidia.com/cuda/profiler-users-guide/.

[5] The openacc application programming interface, 2013. [online]. available: http://www.openacc-standard.org.

[6] Alireza Majidi Ahmad Lashgar and Ebad Salehi. IPMACC. https://github.com/lashgar/ipmacc, 2016.

[7] Michael Bauer, Henry Cook, and Brucek Khailany. Cudadma: Optimizing gpu memory bandwidth via warp specialization. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 12:1–12:11, New York, NY, USA, 2011. ACM.

[8] M. Burtscher and P. Ratanaworabhan. Fpc: A high-speed compressor for double-precision floating-point data. *IEEE Transactions on Computers*, 58(1):18–31, Jan 2009.

[9] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 44–54, Washington, DC, USA, 2009. IEEE Computer Society.

[10] Mark Govett, Jacques Middlecoff, and Tom Henderson. Directive-based parallelization of the nim weather model for gpus. In *Proceedings of the First Workshop on Accelerator Programming Using Directives*, WACCPD '14, pages 55–61, Piscataway, NJ, USA, 2014. IEEE Press.

[11] T. Hoshino, N. Maruyama, S. Matsuoka, and R. Takaki. Cuda vs openacc: Performance case studies with kernel benchmarks and a memory-bound cfd application. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, pages 136–143, May 2013.

[12] Tetsuya Hoshino, Naoya Maruyama, and Satoshi Matsuoka. An openacc extension for data layout transformation. In *Proceedings of the First Workshop on Accelerator Programming Using Directives*, WACCPD '14, pages 12–18, Piscataway, NJ, USA, 2014. IEEE Press.

[13] Yoongu Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*, pages 65–76, Dec 2010.

[14] Jiri Kraus, Michael Schlottke, Andrew Adinetz, and Dirk Pleiter. Accelerating a c++ cfd code with openacc. In *Proceedings of the First Workshop on Accelerator Programming Using Directives*, WACCPD '14, pages 47–54, Piscataway, NJ, USA, 2014. IEEE Press.

[15] A. Lashgar, A. Majidi, and A. Baniasadi. Ipmacc: Open source openacc to cuda/opencl translator. arxiv:1412.1127v1 [cs.pl].

[16] P. Ratanaworabhan, Jian Ke, and M. Burtscher. Fast lossless compression of scientific floating-point data. In *Data Compression Conference, 2006. DCC 2006. Proceedings*, pages 133–142, March 2006.

[17] Mehrzad Samadi, Janghaeng Lee, D. Anoushe Jamshidi, Amir Hormati, and Scott Mahlke. Sage: Self-tuning approximation for graphics engines. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 13–24, New York, NY, USA, 2013. ACM.

[18] Vijay Sathish, Michael J. Schulte, and Nam Sung Kim. Lossless and lossy memory i/o link compression for improving performance of gpgpu workloads. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 325–334, New York, NY, USA, 2012. ACM.

[19] Nathan Whitehead and Alex Fit-Florea. Precision & performance: Floating point and ieee 754 compliance for nvidia gpus.

[20] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. Openacc: First experiences with real-world applications. In *Proceedings of the 18th International Conference on Parallel Processing*, Euro-Par'12, pages 859–870, Berlin, Heidelberg, 2012. Springer-Verlag.

[21] George L Yuan, Ali Bakhoda, and Tor M Aamodt. Complexity effective memory access scheduling for many-core accelerator architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 34–44. ACM, 2009.