Employing Compression Solutions under OpenACC

Ebad Salehi*, Ahmad Lashgar†, Amirali Baniasadi‡
Department of Electrical and Computer Engineering, University of Victoria
*Email: *ebads67@uvic.ca, †lashgar@uvic.ca, ‡amiralib@uvic.ca,*

***Abstract*—For GPUs to achieve their peak performance, effective and efficient usage of memory bandwidth is necessary. To this end, programmers invest extensive development effort to optimize a GPU program, specially its memory bandwidth usage. The OpenACC programming model has been introduced to tackle the accelerators programming complexity. However, this model's coarse-grained control on a program can make the memory bandwidth utilization even worse than the utilization achieved under 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 three case studies and achieve up to 1.36X speedup.**

***Keywords*-OpenACC; Compression; Accelerators**

I. 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 [6], [17], [11]. 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 allow for fine-grained control over both data and parallelism. CUDAs 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[16], it can harm performance.

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 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 three 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 the paper is as follows. In Section 2, we review the background of accelerator programming platforms and present a brief summary of the floating point IEEE 754 Standard. In Section 3, we introduce our proposed OpenACC clauses and discuss how to use them. In Section 4, we present our methodology. In Section 5, we evaluate our solution using three case studies. In Section 6, we discuss issues regarding the clauses applicability and performance. In Section 7, we review related works. Finally, in Section 8 we offer conclusions.

II. Background

A. 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 an 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.

B. *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 device 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.

C. *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` types are the equivalent C data types respectively. Any floating point number consists of a sign bit, exponent part, and mantissa. Figure 1 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 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

Sign	Exponent	Mantissa
1	8(11)bits	23(52)bits

Figure 1. IEEE 754 Floating Point Format

a range of numbers to a range of same-exponent numbers and omit the redundant exponent part.

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

III. Proposed Compression Clauses

In this section, we introduce our new OpenACC clauses by which programmers can reduce the memory bandwidth load of their program. Applying these clauses to 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.

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

A. *Data Transfer Clauses*

Any data set used in the kernel regions can be transferred to the accelerator device by one of the clauses listed in table I. To use compression on a data set, the data must first be copied by one of the proposed clauses in the right column of table I instead of the regular data transfer clauses in the left column. This makes the compiler call an alternative API function, which compresses the data 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.

In addition to the predefined parameters, we also introduce two optional input parameters (Min and Max) for data transfer clauses. Programmers can specify the range of the data set by these two parameters, and therefore reduce the compression overhead. However, passing these parameters is mandatory if there is a possibility that the range of the data set can vary during the application execution. Since for correct functionality, the compression method needs to know the range of the data set, the programmer is responsible to make sure that program is informed of the accurate range. 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.

Listing 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;
  }
}
```

Table I
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)

B. 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, 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 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.

C. 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 2 shows the compression method implementation for compression ratio of 2.

Figure 2 is an example of compressing a 32-bit floating number which in this case is Pi. Figure 2.1 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 2.2 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 2.3).

Listing 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;
}
```

D. 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 2.4 and 2.5 illustrate the decompression of Pi. The white colored bits in figure 2.4 show the lost least significant bits that are replaced. Finally, figure 2.5 shows the fully decompressed Pi number. The 8 rightmost bits 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 2. This way the compiler replaces the subtraction and multiplication instructions with a single fused multiply-add instruction [15]. Hence, two constant values, named key_1 and key_2, are needed for decompression as indicated in equation 3. 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]

.

Code listing 3 shows the decompression of a 4-byte floating point number compressed in a 2-byte format.

$$2MaxAbs \times (X - 1.5) = 2MaxAbs \times X - 2MaxAbs \times 1.5 \quad (2)$$

$$= key_1 \times X + key_2 \quad (3)$$

where

$$key_1 = 2MaxAbs,$$

$$key_2 = -3MaxAbs$$

Listing 3. Decompression Implementation in CUDA

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

IV. Methodology

A. Benchmarks

We use benchmark applications from Rodinia benchmark suit [7]. 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].

B. OpenACC Compiler

We use our framework, IPMACC[13], 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).

C. 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.

D. 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.

V. Experimental Results

In this section we show how our solution improves performance in three case studies. We compare the performance of baseline OpenACC and OpenACC compression-enhanced implementations.

A. 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 among 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 3 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 4 illustrates the total application run-time improvement. Total run-time consists of memory allocations, data initializations and transfers, compression time, and kernel time. We can

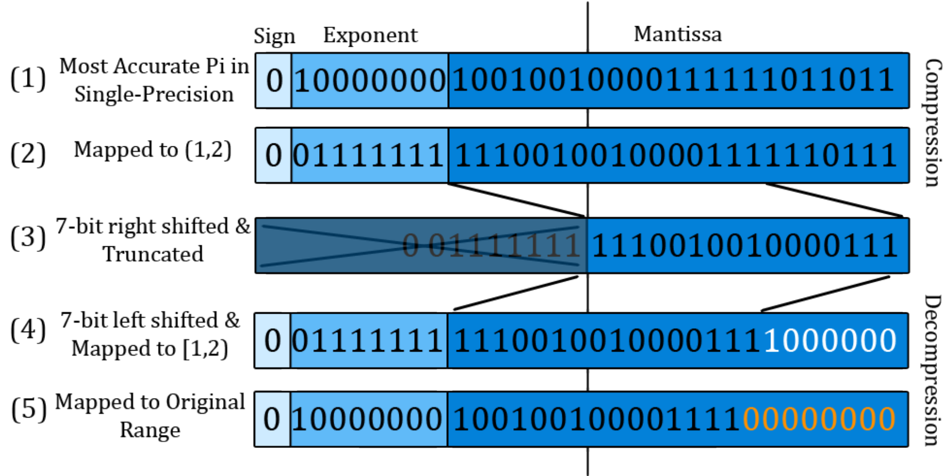

Figure 2. Illustration of Compression and Decompression methods

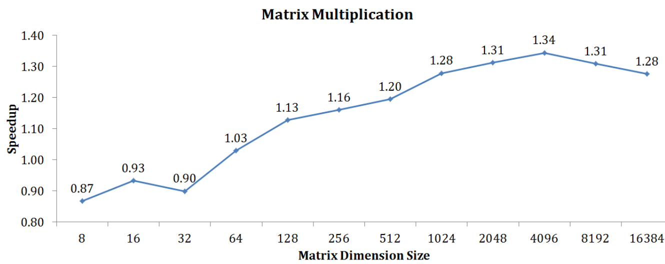

Figure 3. Matrix-Matrix Multiplication Kernel Time Improvement

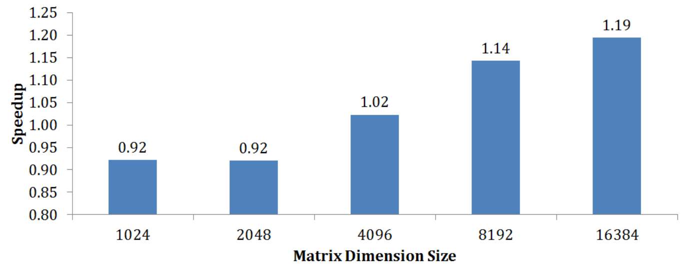

Figure 4. Matrix-Matrix Multiplication Application Run-Time Improvement

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 and 5, respectively. These figures show that the Kernel time constitutes a larger portion of application run-time, and the compression and the memory transfer time become less important as the matrices sizes grow. Hence, achieving 25% less kernel time is worth extra compression calculations before the kernel launch which may even double the application launch time.

Using compression also helps reducing the data transfer time between CPU and GPU. However, this transfer time is not a noticeable fraction of the total application time, specially in comparison with the compression time and the kernel time. Hence, we do not separate it from application launch time.

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 compression method. We compared the results for our application and its compression-enhanced version. The geometric mean of relative errors is 0.006% and the maximum error is equal to 0.01%. These errors are independent of input numbers ranges and size of input matrices.

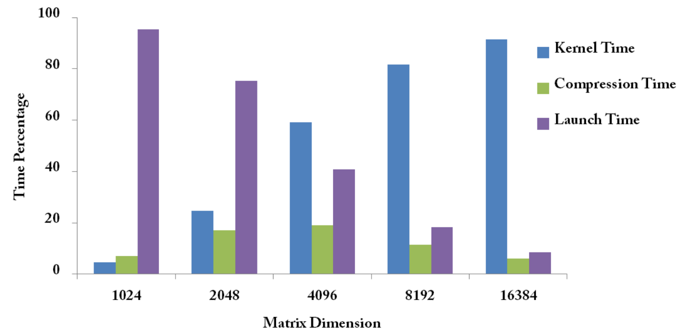

Figure 5. Compression-Enhanced MM Multiplication Breakdown

B. 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

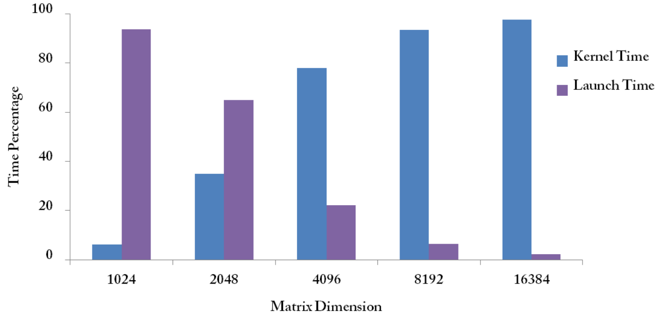

Figure 6. Baseline MM Multiplication Breakdown

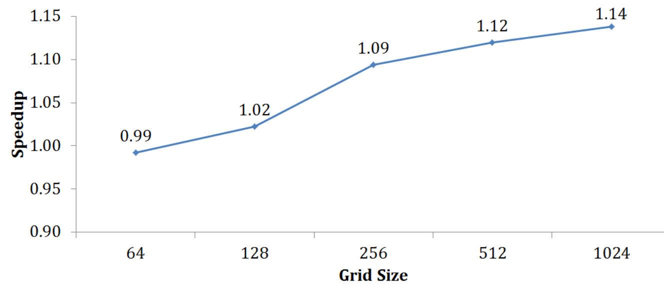

Figure 7. HotSpot Kernel Performance Improvement

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 this in the section VI. We used the compression clause on the computation phase kernel. Figure 7 illustrates performance improvement for the compression-enhanced kernel.

In each iteration of the program the kernel is launched once. Figure 8 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.

C. 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.

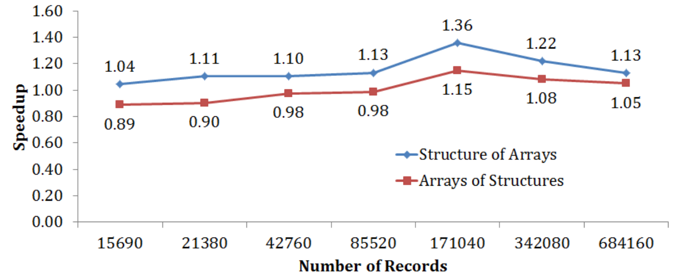

Figure 9. NN Kernel Time Improvement

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 9 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 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 0.003% and the maximum relative as 0.018%.

D. 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. Figure10 shows the kernel time improvement for different data set sizes, using compression.

VI. 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

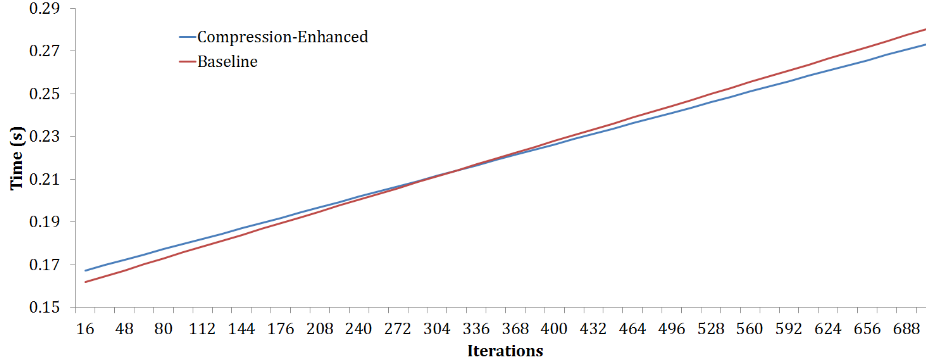

Figure 8. HotSpot Application Run-Time for First 700 Iterations

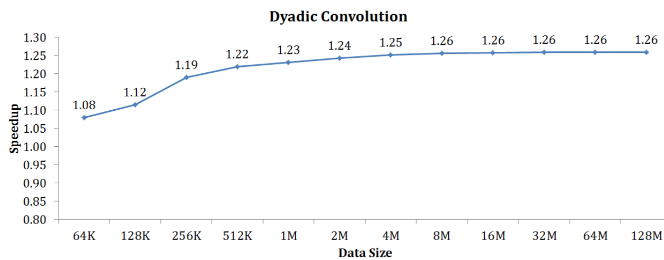

Figure 10. Dyadic Convolution Kernel Time Improvement

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 L1 cache so that further accesses to that block hit in the cache. Using compression decreases the memory latency of bringing data from memory. But it does not help in reading data from the L1 cache, 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 from the L1 cache.

This problem arises because cache is a transparent unit in the memory hierarchy and we do not have control on its behavior. A solution to this issue is using the shared memory. Data sets can be decompressed 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. This is part of our ongoing work.

We performed many experiments to figure out the reason why from certain data set sizes the performance gain declines (eg. Figure 3). This happens because of the L2 cache behavior. Compressing the data not only does lead to less memory bandwidth usage but also allows the memory system able to fit more data elements into the cache. Having this in mind, we can explain why, for example, in Figure 3 we see a reduction in the speedup for matrices larger than 4096. With this data element number, the kernels enhanced with the compression method can fully fit their data into the L2 cache while normal kernels with the same data element number need to use global memory as the data size is beyond the capacity of the L2 cache. As the data element number grows, compression enhanced kernels also need to use global memory, and this lessens the gap between compression enhanced kernels performance and the normal ones. As a result the speedup gained by the compression method shrinks.

VII. Related Work

To the best of our knowledge, our work is the first attempt to add compression capabilities to compiler directive based frameworks. Samadi et al. [14] 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 between them according to the feedbacks. They applied data compression to reduce DRAM memory bandwidth usage as an approximation technique.

Hoshino et al. [9] 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. [16] 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 OpenACCs inability to exploit software-managed cache.

Kraus et al. [12] 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 multiprocessors 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. [8] compare the performance of three different OpenACC implementations under NIM work-load. They perform three optimizations on 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 reduce 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. [10] 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.

VIII. Conclusion

In this paper we introduced a set of OpenACC clauses to enable programmers to accelerate their codes with very low development effort. Our compiler and run-time system performs a transparent compression on the data sets marked by these clauses and hence reduces the memory latencies. We achieve speedups up to 1.36X on GPU kernels from real world applications.

References

[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] M. Bauer, H. Cook, and B. 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.

[7] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. 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.

[8] M. Govett, J. Middlecoff, and T. 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.

[9] T. Hoshino, N. Maruyama, and S. 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.

[10] 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.

[11] Y. 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.

[12] J. Kraus, M. Schlottke, A. Adinetz, and D. 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.

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

[14] M. Samadi, J. Lee, D. A. Jamshidi, A. Hormati, and S. 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.

[15] N. Whitehead and A. Fit-Florea. Precision & performance: Floating point and ieee 754 compliance for nvidia gpus.

[16] S. Wienke, P. Springer, C. Terboven, and D. 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.

[17] G. L. Yuan, A. Bakhoda, and T. 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.