Reduction Operations in Parallel Loops for GPGPUs

Rengan Xu, Xiaonan Tian, Yonghong Yan,
Sunita Chandrasekaran, and Barbara Chapman
Dept. of Computer Science, University of Houston, Houston, TX, 77004, USA
{rxu6,xtian2,yyan3,schandrasekaran,bchapman}@uh.edu

ABSTRACT

Manycore accelerators offer the potential of significantly improving the performance of scientific applications when offloading compute intensive portions of programs to the accelerators. Directive-based programming models such as OpenACC and OpenMP are high-level programming model for users to create applications for accelerators by annotating region of code for offloading with directives. In these programming models, most of the offloaded kernels are data parallel loops processing one or multiple multi-dimensional arrays, and it is often that scalar variables are used in the parallel loop body for reduction operations. Since reduction operation itself has loop-carried dependency preventing the parallelization of the loops, this could have a significant impact on the performance if not handled properly.

In this paper, we present the design and parallelization of reduction operations in parallel loops for GPGPU accelerators. Using OpenACC as the high-level directive-based programing model, we discuss how reduction operations are parallelized when appearing in each level of the loop nest and thread hierarchy. We present how we handle the mapping of the loops and parallelized reduction to single- or multiple-level parallelism of GPGPU architectures. These algorithms have been implemented in the open source OpenACC compiler OpenUH. We compare our implementation with two other commercial OpenACC compilers using test cases and applications, and demonstrate better robustness and competitive performance than others.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*Reduction*

General Terms

Design, Algorithms, Measurement

PMAM'14 February 15 - 19, 2014, Orlando, FL, USA

http://dx.doi.org/10.1145/2560683.2560692.

Keywords

OpenUH, OpenACC, Reduction, Compiler

1. INTRODUCTION

Heterogeneous architectures that comprises of commodity CPUs and computational accelerators such as GPGPUs have been increasingly adopted in both large scale supercomputers, workstations and desktops for engineering and scientific applications. These accelerators provide additional massively parallel computing capabilities to the users, while preserving the flexibilities of CPU for different workloads. However, effectively tapping their full potential may become difficult, due to the programmability challenges faced by users when mapping computation algorithms to such hybrid and heterogeneous architectures.

Programming models such as CUDA[1] and OpenCL[4] for GPGPUs offer users programming interfaces with execution models that closely matches the GPGPU architectures. Effectively using these interfaces for creating highly optimized applications require programmers to thoroughly understand the underlying architecture, as well as to significantly change the program structures and algorithms. This affects both productivity and performance. An alternative approach is to use the high-level directive-based programming models, e.g. HMPP[9], OpenACC [2] and OpenMP [5]. These models allow the user to insert directives and runtime calls into the existing source code, making partial or full portion of Fortran or C/C++ codes to be executed on accelerators. By using directives, programmers can give hints to compilers to perform certain transformation and optimizations on the annotated code region. The user can insert directives incrementally to parallelize and optimize the application, enabling a productive migration path for legacy codes.

The region that is offloaded to the accelerator is referred to a computational kernel. Most of the offloaded kernels are data parallel loops processing one or multiple multi-dimensional arrays, and the loops will be executed in parallel on accelerator devices such as GPGPUs. It is also very common that scalar variables are used in the parallel loop body for reduction operations. Reduction is an operation that uses a binary associative operator to operate an input array and generate a single output value. Since reduction operation itself has loop-carried dependency preventing the parallelization of the loops, this could have a significant impact on the performance if not handled properly. However, they can be computed out of order thanks to the associativity feature of binary operator, thus enabling parallelization to a certain degree.

In this paper, we present the design and parallelization of reduction operations in parallel loops for GPGPU accelerators. Using OpenACC as the high-level directive-based programing model, we discuss how reduction operations are parallelized when appearing in each level of the loop nest and thread hierarchy, e.g. the outer loop with `gang`(coarse-grain), mid-level with `worker`(fine-grain) and inner level with `vector` parallelism. We present how we handle the mapping of the loops and parallelized reduction to single- or multiple-level paralellisim of GPGPU architectures. We implemented these algorithms in the open source OpenACC compiler OpenUH [15], and created a testsuite that provides different use cases of reduction operations for performance evaluation. We compare our results to those using two other commercial OpenACC compilers, and our algorithms passed all reduction usage cases and delivered competitive performance to other compilers.

The main contributions of this paper include:

- We propose several algorithms to parallelize the reduction operation in parallel loops for GPGPUs and implement these algorithms in an open source OpenACC compiler.
- Our algorithms cover all possible cases of reduction operations in three levels of parallelism, all reduction operator types and operand data types.
- We also provided a testsuite and three well-known reduction benchmarks in this work. The evaluations show competitive performance compared to commercial OpenACC compilers.

The organization of this paper is as follows: Section 2 discusses the mapping strategies of high-level parallel loops to the multiple-level paralelism of GPGPU architecture. Section 3 discusses the pallelization strategies for reduction operations of parallel loops. Performance results are discussed in Section 4. Section 5 highlights related work in this area. We conclude our work in Section 6.

2. PARALLELISM MAPPING

The processor architecture of GPGPUs and CPUs are fundamentally different. For example, NVIDIA's GPGPU consists of multiple streaming multiprocessors (SMs), and each SM consists of many scalar processors (SPs, also referred to as cores). Each GPGPU supports the concurrent executions of hundreds to thousands of threads, and each thread is executed by an SP. The smallest scheduling and execution unit is called a warp that has 32 threads. Warps of threads are grouped together into a thread block, and blocks are grouped into a grid. Figure 1 shows how the blocks can be organized into a one or two or three-dimensional grid of thread blocks. Thread blocks cannot synchronize with each other while the threads within a block can.

The GPU has a global memory space that is accessible by all threads in the grid and this is the only space that the CPU memory can communicate with. Shared memory is allocated per thread block,whose memory size can be configurable by the programmer. Because the shared memory is on-chip, the latency is much lower than global memory.

2.1 Programming using OpenACC

Programming GPGPUs has been made possible by CUDA and OpenCL, however programmers typically need to not only thoroughly understand the GPU architecture but also need to rewrite the application source code completely. To that end, OpenACC, an emerging directive-based parallel programming standard, provides a viable approach that primarily aims to improve programmer productivity by simplifying parallel programming for systems with accelerators.

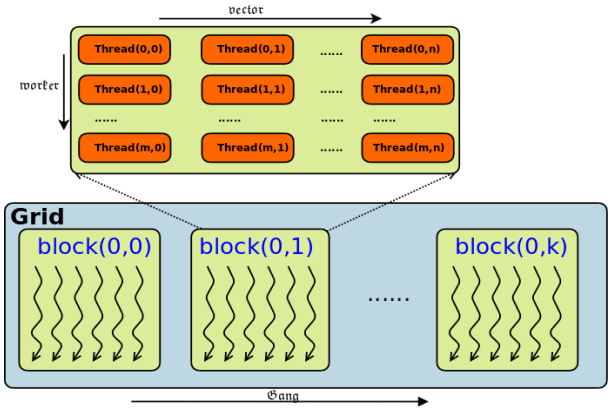


Figure 1: GPGPU Thread Block Hierarchy

OpenACC uses `parallel` or `kernels` constructs to define a compute region that will be executed in parallel on the accelerator device. The `loop` construct is used to specify how the loop iterations to be distributed. The purpose of using `parallel` and `kernels` is that `parallel` construct provides more control to the user while the `kernels` provides more control to the compiler. The `reduction` clause is allowed on a loop construct. The execution model of OpenACC assumes that the main program runs on the host, while the compute-intensive regions of the main program are offloaded to the attached accelerator. In the memory model, usually the accelerator and the host CPU consist of separate memory addresses to prevent conflicts between CPU and accelerators. OpenACC 1.0 discusses different types of data transfer clauses. Possible runtime routines to control data lifetimes have been proposed in the 2.0 specification.

The hardware independence makes OpenACC more attractive for codes that need to run on different architectures. However mapping loops to a different architecture is an interesting challenge. Compilers are relied upon to provide support for mapping onto more than one type of an accelerator chip and provide necessary optimizations for the chosen hardware.

2.2 Mapping Parallel Loops onto GPGPU architectures

OpenACC supports three levels of parallelism: coarse-grained parallelism "gang", fine-grained parallelism "worker" and vector parallelism "vector". The programmer can create several gangs and a single gang may contain several workers and a single worker may contain several vector threads. The iterations of a loop can be executed in parallel by distributing the iterations among one or multiple levels of parallelism of GPGPU architectures. Mapping loops to the hardware in OpenACC is implementation dependent. Figure 2 shows a triple nested loop example that exhibits all the three levels of parallelism. In this example, assume each loop can be executed in parallel, then k loop is distributed across all gangs, j loop is distributed across all workers in a single gang, and

```
#pragma acc loop gang
for(k=k_start; k<k_end; k++){
  #pragma acc loop worker
  for(j=j_start; j<j_end; j++){
    #pragma acc loop vector
    for(i=i_start; i<i_end; i++){
        ...
    }
  }
}
```

Figure 2: Loop Nest Example with OpenACC Parallelisms

Table 1: CUDA Terminology in OpenACC Implementation

Term	Description
threadIdx.x	thread index in X dimension of a thread block
threadIdx.y	thread index in Y dimension of a thread block
blockDim.x	no. of threads in X dimension of a thread block
blockDim.y	no. of threads in Y dimension of a thread block
blockIdx.x	block index in X dimension of the grid
gridDim.x	no. of blocks in X dimension of the grid

i loop is distributed across all vector threads of one worker.

Our OpenACC implementation is built on top of the SIMT execution model of CUDA. Table 1 shows the CUDA terminologies that is used in our OpenACC implementation. In OpenUH compiler, `gang` maps to a thread block, `worker` maps to the Y-dimension of a thread block and `vector` maps to the X-dimension of a thread block. Based on these definitions, the implementation details for the loop nest discussed in Figure 2 is shown in Figure 3. Here we add the start offset to the index of each level of threads so that the working threads start from 0 that effectively reduce the thread divergence. Another possible implementation is to get the thread id and then determine whether the id is greater than the start position and lesser than the end position in each loop level. In that case, the threads whose ids are lesser than the start position will not participate in the computation and will be idle all the time leading to thread divergence. In our implementation, the threads in each loop level increase along with their own stride size, so that each thread processes multiple elements of the input data. This solves the issue of limited number of threads availability in the hardware platform. Our implementation is designed in a way that it is independent of the number of threads used in each loop level. However, the appropriate number of threads may enable coalesced memory access and improve performance.

In the loop nest example, we assume that all the iterations are independent, but most time the loop nest may contain reduction operation and the reduction may appear anywhere on the loop nest. In the next section, we will discuss such cases and how they are implemented in our OpenUH compiler.

3. PARALLELIZATION OF REDUCTION OPERATIONS FOR GPGPUS

The reduction operation applied to a parallel loop uses a binary operator to operate on an input array and generates a single output value for that array. Each thread has its own local segments copy of the input array when the loop is distributed among threads. The operation that consolidates

```
k = blockIdx.x + k_start;
while(k < k_end){
  j = threadIdx.y + j_start;
  while(j < j_end){
    i = threadIdx.x + i_start;
    while(i < i_end){
      ...
      i += blockDim.x;
    }
    j += blockDim.y;
  }
  k += gridDim.x;
}
```

Figure 3: Implementation of Example Loop Nest

the results from the thread-local copies of the segments using the reduction operation is the issue that we are addressing in this paper. The approach to performing parallel reductions depends on how the loop nests are mapped to the GPGPU's thread hierarchy. Moreover, reduction operation always implies a barrier synchronization, this may introduce runtime overhead, hence we need to be cautious to only include the synchronization when necessary.

Although most reduction operations are inherently not parallel, for those that have the properties of associativity and commutativity [3], we are able to apply the divide and conquer method to achieve parallel execution. That is, let us assume there are three input variables, *a1*, *a2* and *a3*, and the reduction operator 'sum' does $a1+a2+a3$. The associativity of a binary operator is a property that determines how operators of the same order of operations are grouped without using parentheses. Since each reduction uses only one operator and this operator has equal precedence, the operation can be grouped differently. For instance, $(a1+a2)+a3$ and $a1+(a2+a3)$ will deliver the same output as $a1+a2+a3$. The commutativity of a binary operator is a property that changes the order of operations and does not change the result. For instance, $a3+a1+a2$ and $a2+a3+a1$ will deliver the same output as $a1+a2+a3$. All of the OpenACC reduction operators satisfy both associativity and commutativity properties. So the reduction operations can be performed in any order as long as it uses a single operator and includes all of the input data. These are the vital properties that we will be applying in our implementation.

3.1 Reduction in Single-level Thread Parallelism

The loop nest where a reduction operation is applied could be mapped to one or multiple level thread hierarchy. For example, OpenACC includes three-level of parallelisms: gang, worker and vector, reduction can appear within any of these levels. Let us first discuss the case where the reduction operation appears in only one level of the parallelism.

3.1.1 Reduction only in vector

Figure 4(a) shows an example of reduction occurring only in vector, where the worker and gang loops (k and j) can be executed in parallel, whereas the vector loop (i) needs to perform reduction. There are different strategies to parallelize this case, as shown in Figure 6. Figure 6 (a) shows the data and worker and vector threads laid out in one gang before doing reduction. Each row is one worker that includes multiple vector threads. Since vector reduction happens in each worker, each row needs to do reduction and finally each worker should have one reduction result. In this example, there are 4 workers and each worker has 4 vector

```
#pragma acc parallel \
            copyin(input) \
            copyout(temp)
{
  #pragma acc loop gang
  for(k=0; k<NK; k++)
  {
    #pragma acc loop worker
    for(j=0; j<NJ; j++)
    {
      int i_sum = j;
      #pragma acc loop vector \
              reduction(+:i_sum)
      for(i=0; i<NI; i++)
        i_sum+=input[k][j][i];
      temp[k][j][0] = i_sum;
    }
  }
}
```

(a) Reduction in vector

```
#pragma acc parallel \
            copyin(input) \
            copyout(temp)
{
  #pragma acc loop gang
  for(k=0; k<NK; k++)
  {
    int j_sum = k;
    #pragma acc loop worker \
            reduction(+:j_sum)
    for(j=0; j<NJ; j++)
    {
      #pragma acc loop vector
      for(i=0; i<NI; i++)
        temp[k][j][i]=input[k][j][i];
      j_sum += temp[k][j][0];
    }
    temp[k][0][0] = j_sum;
  }
}
```

(b) Reduction in worker

```
sum = 0;
#pragma acc parallel \
            copyin(input) \
            create(temp)
{
  #pragma acc loop gang \
          reduction(+:sum)
  for(k=0; k<NK; k++)
  {
    #pragma acc loop worker
    for(j=0; j<NJ; j++)
    {
      #pragma acc loop vector
      for(i=0; i<NI; i++)
        temp[k][j][i]=input[k][j][i];
    }
    sum += temp[k][0][0];
  }
}
```

(c) Reduction in gang

Figure 4: Reduction in Single-level Thread Parallelism Example

```
k = blockIdx.x;
while(k < NK){
  j = threadIdx.y;
  while(j < NJ){
    i = threadIdx.x;
    int i_sum = j;
   /* private for each vector */
    int i_sum_priv = 0;
    while(i < NI){
      i_sum_priv += input[k][j][i];
      i += blockDim.x;
    }
    sbuf[threadIdx.x+threadIdx.y*
        blockDim.x]=i_sum_priv;
    __syncthreads();
    i_sum = reduce_vector(sbuf, j);
    temp[k][j][0] = i_sum;
    j += blockDim.y;
  }
  k += gridDim.x;
}
```

(a) Reduction in vector

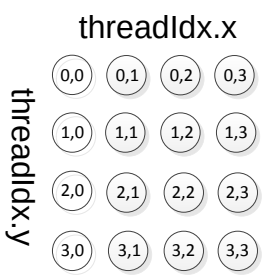
```
k = blockIdx.x;
while(k < NK){
  j = threadIdx.y;
  int j_sum = k;
 /* private for each worker */
  int j_sum_priv = 0;
  while(j < NJ){
    i = threadIdx.x;
    while(i < NI){
      temp[k][j][i]=input[k][j][i];
      i += blockDim.x;
    }
    j_sum_priv += temp[k][j][0];
    j += blockDim.y;
  }
  if(threadIdx.x == 0)
    sbuf[threadIdx.y] = j_sum_priv;
  __syncthreads();
  j_sum = reduce_worker(sbuf, k);
  k += gridDim.x;
}
```

(b) Reduction in worker

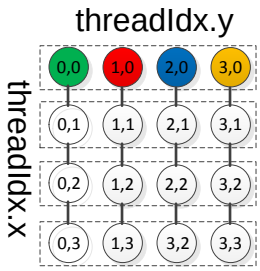
```
k = blockIdx.x;
sum = 0;
/* private for each gang */
int sum_priv = 0;
while(k < NK){
  j = threadIdx.y;
  while(j < NJ){
    i = threadIdx.x;
    while(i < NI){
      temp[k][j][i]=input[k][j][i];
      i += blockDim.x;
    }
    j += blockDim.y;
  }
  sum_priv += temp[k][0][0];
  k += gridDim.x;
}
if(threadIdx.x == 0 &&
   threadIdx.y == 0)
  partial[blockIdx.x]=sum_priv;
```

(c) Reduction in gang

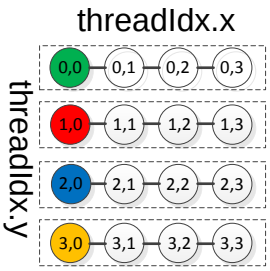
Figure 5: Implementation for Reduction Examples in Figure 4



(a) Data and threads layout in global memory



(b) One type data and threads layout in shared memory



(c) Another type data and threads layout in shared memory

Figure 6: Parallelization Comparison for Vector Reduction. (a) includes the original threads layout in a thread block. In (b), each vector thread works on each column data and the reduction results are stored in the first row. In (c), each vector thread works on each row data and the reduction results are stored in the first column. The data inside the dashed rectangle are contiguous in memory.

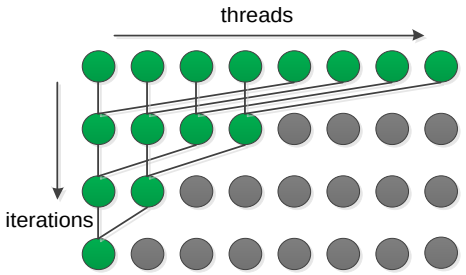


Figure 7: Interleaved Log-step Reduction. Synchronization is inserted after each iteration and before the next iteration. Green represents active threads while the grey represents inactive threads in each iteration.

threads, so four vector reduction results should be generated. Since NVIDIA GPU provides very low latency shared memory, the reduction can be moved to the shared memory to reduce memory access latency. We present two different implementation strategies in Figure 6 (b) and (c). In both these strategies, each vector thread first creates a private variable and does the partial reduction itself and then all the partial private reduction values are stored into the shared memory. But how these data are stored and which thread works on which data can have a significant impact on the performance.

Figure 6(b) uses a strategy where the data and the threads layout are transposed in the shared memory, so the reduction in every row of the original thread block becomes the reduction in every column of the thread block and the final four reduction results are stored in the first row. This approach increases memory divergence since the data that needed to use reduction are not stored contiguously in the shared memory. This is because a warp is the smallest execution unit for GPGPUs and instructions are SIMD-synchronous within a warp.

Figure 6(c) shows yet another approach which is implemented in OpenUH. In this approach, the thread layout is the same as the data in the global memory and the layout of the threads working on these data still keep the same. Therefore the vector reduction happens in each row and the final reduction values are stored in the first column of the shared memory. Thus, the data that needs to be reduced are stored contiguously in the shared memory. Although memory divergence may still happen, this issue could be solved by unrolling the last 6 iterations where the reduction data size are 64, 32, 16, 8, 4 and 2. Actually in our implementation, we unroll all iterations since the thread block size is limited to less or equal to 1024 threads in the hardware we are using. Vector size should be a multiple of the warp size.

For both Figure 6(b) and (c), the vector threads in the shared memory do the reduction using the interleaved log-step reduction algorithm [10] shown in Figure 7.

A point to note is that the initial value of the variable that needs to be reduced may have a different value for the private copy of that variable. For example, the initial value of *i_sum* in Figure 4(a) is *j*, but the initial value for the private copy of the variable *i_sum_priv* for each thread is 0 (shown in Figure 5(a)). In most of the implementations, the initial value is processed after the vector reduction algorithm is done. For instance, the initial value is summed for + reduction or multiplied for * reduction.

3.1.2 Reduction only in worker

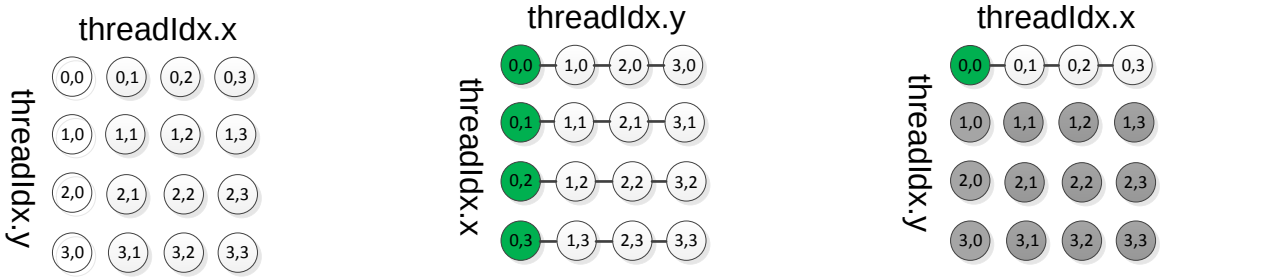
Figure 4 (b) shows an example of reduction occurring only in worker, where the gang and vector loops (k and i) can be executed in parallel, whereas the worker loop (j loop) has to do the reduction. Again we present two parallelization strategies in Figure 8(b) and (c). In both implementations, each worker creates a private variable and does the private partial reduction first, then all the private reduction data computed by all workers will be stored into the shared memory for the final reduction. But the next step they go to different paths.

In Figure 8(b), the original vertically layout workers are placed into the shared memory horizontally. That is, the transposed threads work on the transposed data elements in the shared memory. Only the first row in the shared memory contains the useful data while the other rows have duplicate data as the first row. All rows need to do the same interleaved log-step reduction algorithm as vector reduction, final worker reduction results are stored in the first column of the shared memory. Since all rows are the same, actually only the first element of the first row has the useful final reduction result. The advantage of this approach is that the implementation follows the worker reduction concept very strictly. But the disadvantage is that it consumes a lot of shared memory which is a scarce resource in the GPGPU architecture, and it needs to insert synchronization between each iteration in the reduction algorithm because the workers are not consecutive or they are not warp threads.

The OpenUH compiler, however, employs another different strategy shown in Figure 8(c). First, the workers also need to do partial reduction by creating private variable. Next the first vector thread of each worker stores the partial reduction into the shared memory, then all the original vector threads of workers use the interleaved log-step reduction algorithm to generate the final reduction result. Note that the threads working on the shared memory data are the same as the threads working on the global memory data, so no transpose happens here. Using this approach requires less threads and less shared memory so that we can leave more shared memory for other more important computations. For instance, in the example of Figure 8(c), only 4 threads are required to do the reduction and only the first row of the shared memory is occupied. Also the advantage of this approach is that the vector threads are warp threads, so we do not need synchronization in the last 6 iterations when only the last warp threads need to the reduction.

3.1.3 Reduction only in gang

The example of reduction only in gang is shown in Figure 4(a), where the inner worker and vector loops (j and i) are executed in parallel while the gang loop (k loop) has reduction. Since we map each gang to each thread block in CUDA and there is no synchronization mechanism to synchronize all thread blocks, the strategy of OpenUH is to create a temporary buffer (*partial* in Figure 5 (c)) with the size equal to the number of gangs, each block writes its partial reduction into the specific entry of the buffer, then another kernel (the same reduction kernel as the one in vector addition) is launched to do the reduction within only one



(a) Data and threads layout in global memory (b) One type data and threads layout in shared memory (c) Another type data and threads layout in shared memory

Figure 8: Parallelization Comparison for Worker Reduction. In (a), Threads in each row is one worker, so four workers need to do reduction. In (b), four worker values are in every row, so four rows have duplicate values and the final reduction is stored in the first column. In (c), four worker values are only in the first row and following three row threads are inactive, the final reduction is stored in the first element of the first row.

block. How to implement the partial reduction within each gang may be different in different compilers. One approach is to divide the iterations all gangs need to compute among all gangs equally, then each gang works on the assigned iterations. OpenUH does not use such blocking algorithm, instead OpenUH considers all gangs as a window, and this window slides through the iteration space. This is similar to the round-robin scheduling algorithm. Essentially there is no difference between blocking algorithm and window sliding techniques in gang partial reduction, but the window sliding technique is superior than blocking algorithm in vector partial reduction since it can enable memory coalescing. Memory coalescing is impossible in worker and gang partial reduction, but we still use window sliding technique in both cases in order to make the implementation consistent.

3.2 Reduction across Multi-level Thread Parallelism (RMP)

Section 3.1 focused on reduction occurring only in single level parallelism. Although we discuss each of the cases individually, there could be several combinations of these cases, so some or all of these single cases could be grouped together. For instance, in a triple nested loop, the outermost, the middle and the innermost loops use gang, worker and vector reduction, respectively. Reduction can also occur on different variables within different levels of parallelism. Multiple levels of parallelisms can happen within different loops or within the same loop. Next we will discuss all these possibilities in detail.

3.2.1 RMP in Different Loops

Figure 9 shows an example of the same reduction spanning across different levels of parallelism in different loops. In this example, the *j_sum* needs to perform reduction on both the worker and vector loops. The CAPS compiler adds the reduction clause to both the worker and vector loops, failing which incorrect result is generated. This is also a favorable step since reduction occurs in both worker and vector level parallelisms. The OpenUH compiler, however, is smarter since it can automatically detect the position of the reduction variable and the user just needs to add the

```
#pragma acc parallel copyin(input) \
                     copyout(temp)
{
  #pragma acc loop gang
  for(k=0; k<NK; k++)
  {
    int j_sum = k;
    #pragma acc loop worker reduction(+:j_sum)
    for(j=0; j<NJ; j++)
    {
      #pragma acc loop vector
      for(i=0; i<NI; i++)
        j_sum += input[k][j]i];
    }
    temp[k] = j_sum;
  }
}
```

Figure 9: Example of RMP in Different Loops

reduction clause to the loop that is the closest to the next use of that reduction variable. In this case, *j_sum* is assigned to *temp[k]* after the worker loop, so we add the reduction in the worker loop. If *j_sum* is used after the vector loop and inside the worker loop, then we add the reduction clause in vector loop. CAPS compiler at times, also just needs to add the reduction clause to the outer most loop, but only when all the inner loops are sequential. With respect to the implementation, OpenUH compiler creates a buffer with the size equal to the number of all threads that needs to do the reduction (workers * vector threads in this example) and the buffer is stored in the shared memory. Each thread writes its own partial reduction result into this buffer and continues the reduction operation in the shared memory. The multi-levels of parallelisms can be of three scenarios: gang & worker, gang & worker & vector, and worker & vector. For the former two cases, a temporary buffer is created and all threads performing reduction operation write their own private reduction into this buffer based on the unique id of each thread. The buffer is allocated in the global memory since the reduction spans across gangs and all gangs do not have the mechanism to synchronize. Another kernel that takes this temporary buffer as input is launched and this kernel performs the vector reduction to generate the final reduction

value. Note that the reduction cannot span across gang & vector without going through the worker.

An alternative approach is to perform the reduction in multi-level parallelism following the order of the parallelisms the reduction appear in. In the example of Figure 9, each vector thread performs partial reduction and populates its own private variable, then all vector threads perform the vector reduction with their private reduction values. As a next step, each worker does the partial reduction and populate its own private variable, then all workers do the worker reduction with their private reduction values. Each worker's private reduction value is the reduction value of all vector threads within that worker. The final worker reduction value is the private value for each gang. Since each gang has multiple workers, the final reduction value would be different for each gang. Using this approach, the private variables are different in worker and vector, the vector and worker reduction algorithms could reuse the algorithms discussed in Section 3.1. This implementation converts the same reduction in different loop levels into different reductions in different loop levels so that the algorithms in Section 3.1 can be reused. OpenUH does not use this implementation since this approach needs to perform reduction in multiple times and therefore more synchronizations are required.

3.2.2 RMP in the Same Loop

```
sum = 0;
#pragma acc parallel copyin(input)
{
  #pragma acc loop gang worker vector \
                reduction(+:sum)
  for(i=0; i<NI; i++)
    sum += input[i];
}
```

Figure 10: Example of RMP in the Same Loop

Figure 10 shows an example of reduction across multi-level parallelism in the same loop. In the implementation, OpenUH creates a buffer with the size equal to the size of the all threads that need to reduction (gangs * workers * vector threads in this example), then each thread does its own partial reduction, and finally launch another kernel to do the reduction for all values in the buffer. Again whether the buffer is stored in global memory or shared memory depends on whether the reduction happens in gang parallelism. As long as gang parallelism is involved, the buffer must be in global memory.

3.3 Special Reduction Considerations

Apart from the cases listed in Section 3.1 and Section 3.2, there are some other special reduction cases. One of them is that the same reduction clause includes multiple reduction variables and these variables have different data types (e.g. int and float). In this case, one way is to create a large shared memory space and different sections of the shared memory are reserved for different reduction data types. This implementation may face the shared memory size issue since too many reduction variables may required more shared memory than the hardware limit. OpenUH compiler, however, creates a shared memory space with the size the same as the largest required shared memory for a particular data type. For instance, if there are "int" type reduction and the "double" type reduction in the same reduction clause, then we just need to create a shared memory for the double type reduction because the required int type reduction memory space is smaller than the required shared memory for double type reduction and these two reduction can share the same shared memory space.

We implemented the different cases of reduction operation in both global and shared memory. Although the implementation in global memory is similar to that of the shared memory's, the main difference is the memory access latency. We created an implementation in the global memory primarily because the shared memory is sometimes reserved for other computation, therefore there is not enough memory space for performing reduction operations. Take the blocked matrix multiplication for example, the matrix is divided into multiple blocks and the computation for each block occurs inside the shared memory. Therefore if a reduction has to happen at the same time, then we would need to move the reduction operation to the global memory.

Another issue is the size of the iteration space and the size of threads. The algorithm in [10] requires that both the iteration space and thread size be power of 2. We remove such a restriction in OpenUH. The restriction of the iteration space size is removed in the algorithm as shown in Figure 3, because the threads window slides through the iteration space. Although when the iteration space is not power of 2, there will be some memory divergence within the iterations in the last window. Therefore, the iteration space size in the interleaved log-step reduction algorithm is decided by the thread size rather than the original iteration space size itself. Because the log-step reduction algorithm inherently requires that the iteration space must be power of 2, we need some additional steps before we could consider the algorithm, when the threads size is not power of 2. For instance, if the threads size is 96, first we need to get the previous power-of-2 number 64, then the first 32 threads will do the reduction on the first 32 elements and the last 32 elements. Then the first 32 threads will work on the first 32 elements and the middle 32 elements which is 64 elements which has already satisfied the requirement of the log-step reduction algorithm. The recommended vector threads size is multiple of warp size (32). Although the vector threads size also could not be multiple of 32, the correctness will not be affected but the performance will degrade significantly.

4. EVALUATION

The experimental platform has 24 Intel Xeon x86_64 cores with 32GB main memory, and an NVIDIA Kepler GPU card (K20c) with 5GB global memory. We use OpenUH compiler to evaluate our OpenACC reduction implementation. For a comparative analysis, we also uses commercial OpenACC compilers CAPS 3.4.0 and PGI 13.10 compilers. CUDA 5.5 is used for all the three compilers. GCC 4.4.7 is used as the host compiler for CAPS compiler. To easily compare the CPU result and GPU result, we disable Fused Multiply Add (FMA) [16]. We use "-O3, -acc -ta=nvidia,cc35,nofma" for the PGI compiler and "–nvcc-options -Xptxas=-v,-arch,sm_35,-fmad=false gcc -O3" for the CAPS compiler. OpenUH compiler uses "-fopenacc" flag to compile the given OpenACC program. And since OpenUH uses a source-to-source technique, CUDA nvcc compiler is used to compile the generated kernel files and the flag passed to nvcc compiler is "-arch=sm_35 -fmad=false". The number of vector size is set to 128 since Kepler architecture has

quad warp scheduler that allows to issue and execute four warps (32 threads) simultaneously. Threads within a thread block is limited to 1024 threads due to which the number of workers is set to 8. All thread blocks are scheduled on all streaming multiprocessors (SMs). Kepler has 13 SMs and one of them is likely to be disabled [8], also each SM can support at most 16 thread blocks. To keep all SMs busy we choose the number of gangs to be 12x16=192. We can set the gang, worker and vector using `num_gangs`, `num_workers` and `vector_length` clauses in OpenACC.

Since there are no existing benchmarks that could cover all the reduction cases, we have designed and implemented a testsuite to validate all possible cases of reduction including different reduction data types and reduction operations. The testsuite will check if a given reduction implementation passed or failed by verifying the OpenACC result with the CPU result. If the values do not match, it implies there is an implementation issue. We also measure the execution time of each of the reduction cases, so if the compilers under evaluation can pass the test, we compare their performances too. For all the test cases, we perform reduction using OpenACC first and then on the CPU side, after which we compare if their results are the same. When reduction occurs in one of the levels of parallelism, the other levels of parallelisms has instructions being executed in parallel. Only the RMP in the same loop uses one loop, the other reduction tests use triple nested loop. When one loop level needs to do reduction, that loop iteration size is up to 1M and the other two loops are 2 and 32 because of the memory limit of the hardware. Although we used triple nested loop in experiments, the user can use `collapse` clause in OpenACC if the loop level is more than three. We discuss the results of the most commonly used reduction operators "+" and "*"; the implementation of other reduction operators are almost the same. We also use a real world application to demonstrate "max" reduction intrinsic.

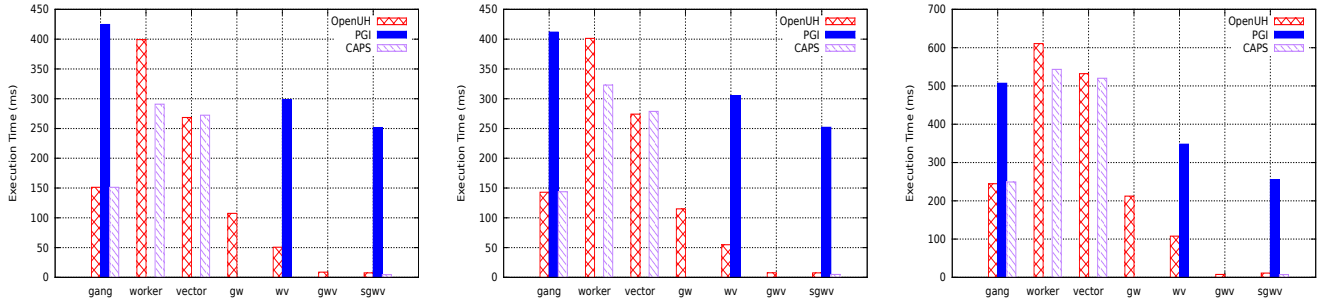
Table 2 discusses the performance results of OpenUH, PGI and CAPS compilers while using the reduction testsuite. We see that only OpenUH compiler passed all of the reduction tests. CAPS compiler failed some of the tests of RMP in different loops. PGI compiler failed the summation reduction in worker, vector and RMP in gang & worker. It even failed to compile the RMP in gang & worker & vector. Figure 11 shows the performance comparison of the three compilers. It is observed that in gang or vector reduction, the performance of OpenUH compiler is more or less the same as the CAPS compiler, and only in worker reduction it is slightly less efficient than CAPS compiler. The performance of OpenUH is better than PGI compiler for all reduction cases. We could not dive deeper into the analysis for the obvious reason that CAPS and PGI are commercial compilers and we do not have access to their underlying implementation details. Although the execution time here is only several hundred milliseconds, it can still have an impact on a real-world application. We discuss this later.

Although the testsuite only includes the reduction cases in triple nested loop and one loop, they can be used in any levels of the loop. Apart from the testsuite, we also used some real-world benchmark applications: 2D Heat Equation, matrix multiplication and Monte Carlo PI. Let us look into the evaluation details.

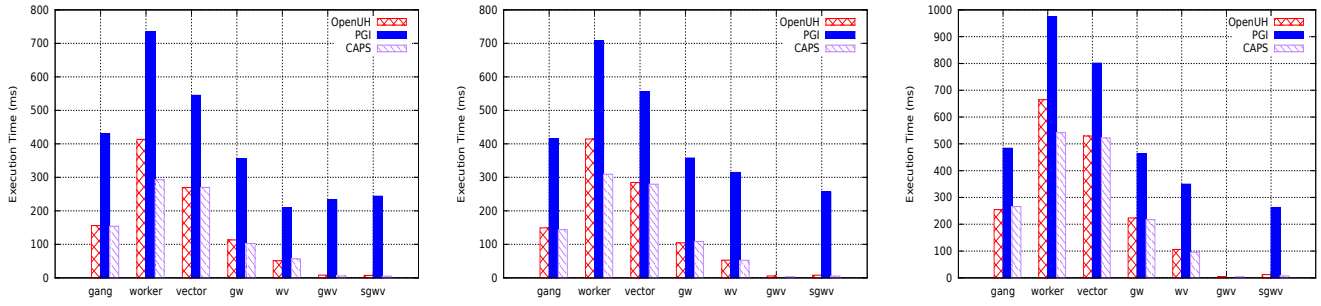
2D Heat Equation is a type of stencil computation. The formula to represent the 2D heat equation is explained in [14]. In this application, there is a grid that has boundary points and inner points. Boundary points have an initial temperature and the temperature of the inner points need to be updated over iterations. Each inner point updates its neighboring points and itself. The temperature updating operation for all inner points needs to last long enough to obtain the final stable temperature. We added the temperature converge code in [14] so that we can know when the convergence happens. The temperature is stable when the maximum temperature difference for all data points in the grid between two consecutive iterations gradually decreases from a large value until 0. So in every iteration, the program needs to compute the maximum difference for all data points in the current iteration and all data points in the previous iteration, which is a max reduction in OpenACC. The code snippet is shown in Figure 13 (a), where *temp1* is the temperature in the previous iteration and *temp2* is the temperature in the current iteration. All data points in the grid are traversed to get the maximum error. We have prior experience working on the parallelization of the temperature updating kernel [17], but in this paper we only focus on the maximum reduction. Figure 12 (a) shows the performance comparison between OpenUH and other two compilers. The performance of the CAPS compiler is missing since the temperature difference generated by this compiler increases gradually rather than a decrease, so the application can never converge. We increase the grid size from 128x128 to 512x512 and we find that OpenUH compiler is always better than PGI compiler. This demonstrates that the performance of the reduction implementation will accumulate in an iterative algorithm; we do not observe the signicant performance improvement in only one iteration.

Matrix Multiplication is a classic example in parallel programming. We consider a naive matrix multiplication case. Most developers usually only parallelize the outer two loops and lets the third loop execute sequentially since the third loop has data dependence. However we can also parallelize the third loop because essentially it just includes the "sum" reduction operations. The code snippet is shown in Figure 13 (b) and the performance comparison in shown in Figure 12 (b). Different matrix sizes are chosen and the result shows that the performance of OpenUH is more than 2x better than CAPS compiler. Since the reduction here happens only in vector and PGI compiler failed the vector reduction as indicated in Figure 11 (c), the PGI performance bar is not shown.

Monte Carlo PI is another example of using reduction. PI (π) can be computed in different ways and one of them is to use the Monte Carlo statistical method. Given a circle of the radis 1 is inscribed inside a square with side length 2, then the area of the circle and the square are π and 4, respectively. Therefore the ratio of the area of the circle to the area of the square (ρ) is $\pi/4$. The program picks points within the square randomly and check whether the point is also inside the circle. This can be determined by the formula $x^2 + y^2 < 1$, where x and y are the coordinates of the point. Assume the number of data points within the circle is m and the number of data points within the square is n, then $\rho = m/n$ and we can obtain $\pi = 4.0 * m/n$. The more data points sampled within the square, the more accurate the value of π can be obtained. In the program, n is the total number of iterations of a loop and inside the loop the coordinates of a data point x and y are randomly

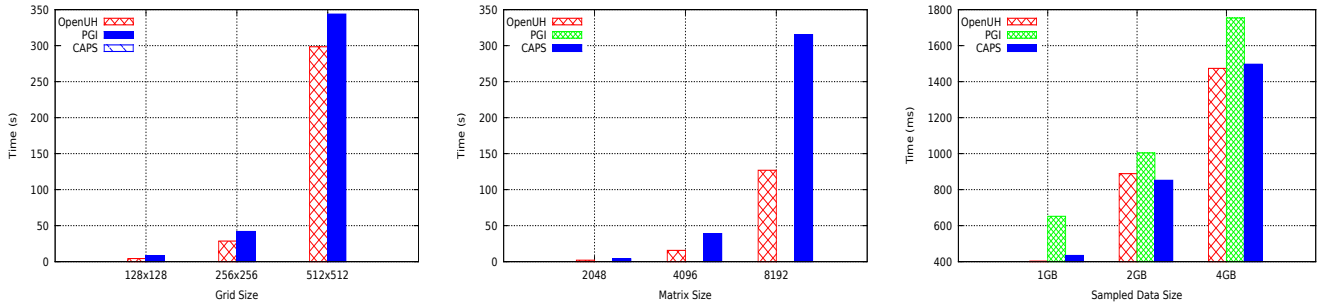


(a) Int [+] (b) Float [+] (c) Double [+]



(d) Int [*] (e) Float [*] (f) Double [*]

Figure 11: Performance Comparison of OpenACC Compilers using Reduction Testsuite. Missing bars imply that the test failed. The symbol in square brackets indicates the reduction operator. gw: gang worker; wv: worker vector; gwv: gang worker vector; sgwv: same line gang worker vector



(a) 2D Heat Equation [max] (b) Matrix Multiplication [+] (c) Monte Carlo PI [+]

Figure 12: Performance Comparison for three Applications. CAPS bar in (a) and PGI bar in (b) are missing because they failed. The symbol in square brackets is the reduction operator.

```
#pragma acc loop gang
reduction(max:error)
for (j=1; j < nj-1; j++) {
    #pragma acc loop vector
    for (i=1; i < ni-1; i++) {
        i00 = j*ni + i;
        error = fmax(error,
    fabs(temp1[i00] - temp2[i00]));
    }
}
```

(a) 2D Heat Equation

```
#pragma acc loop gang
for (i = 0; i < n; i++){
    #pragma acc loop worker
    for (j = 0; j < n; j++){
        c = 0.0;
        #pragma acc loop vector
            reduction(+:c)
        for (k = 0; k < n; k++)
            c+=A[i*n+k]*B[k*n +j];
        C[i*n+j] = c;
    }
}
```

(b) Matrix Multiplication

```
#pragma acc loop gang vector
    reduction(+:m)
for(i=0; i<n; i++)
{
//x[i]=2.0*rand()/(RAND_MAX+1.0)-1.0;
//y[i]=2.0*rand()/(RAND_MAX+1.0)-1.0;
    if(x[i]*x[i] + y[i]*y[i] < 1.0)
        m++;
}
```

(c) Monte Carlo PI

Figure 13: Code Snippet for Three Applications

Table 2: Performance Results of OpenACC Compilers using reduction testsuite. Time is in milliseconds. "F" stands for test FAILED, and "CE" stands for compile time error. Except the "same line gang worker vector" uses only one loop, the other reduction tests used triple nested loop where the outermost, middle and the innermost loops are gang, worker and vector respectively. The first column implies the reduction position. For instance, "gang worker" means gang and worker loops need to do reduction while the vector loop executes in parallel.

Reduction Position	Reduction Operator	Data Type								
		Int			Float			Double		
		OpenUH	PGI	CAPS	OpenUH	PGI	CAPS	OpenUH	PGI	CAPS
gang	+	151.27	424.64	151.29	142.99	411.76	143.78	244.61	507.02	249.13
	*	156.01	430.77	153.92	249.27	415.91	144.12	254.90	483.59	266.09
worker	+	399.25	F	290.83	401.33	F	322.97	610.61	F	543.20
	*	413.35	734.35	292.86	414.50	708.29	309.25	664.87	973.88	541.80
vector	+	268.47	F	272.32	274.06	F	278.74	532.01	F	520.14
	*	269.80	544.71	269.98	284.68	555.58	279.58	529.37	800.89	522.11
gang worker	+	107.49	F	F	115.10	F	F	212.31	F	F
	*	113.36	356.00	102.50	104.44	357.50	108.53	223.30	463.34	217.15
worker vector	+	50.58	298.33	F	54.85	304.82	F	107.75	347.90	F
	*	51.20	209.72	56.82	52.75	314.60	52.46	105.97	349.44	95.78
gang worker vector	+	8.77	CE	F	7.66	CE	F	7.65	CE	F
	*	8.15	232.84	5.50	5.61	CE	3.09	4.87	CE	3.82
same line gang worker vector	+	7.55	251.67	4.60	7.57	251.98	4.94	11.24	255.12	7.26
	*	7.25	243.63	5.21	7.86	256.18	5.361	11.90	262.49	6.90

generated by calling rand() in C, and then check whether $x^2 + y^2 < 1$. If yes, then m is increased by 1. Therefore the computation of m is actually a reduction operation. Since at the time of writing most compilers do not support function call inside an OpenACC kernel region , we pre-generate the x and y values on the host and then transfer them to the device to for m reducion. The code is in Figure 13 (c) where the loop is one level and the computation is distributed to gang and vector threads. For more accurate PI value, we try to sample as many points as we can. In our Kepler architecture, the maximum global memory is 5GB, so we use different sampled data size 1GB, 2GB and 4GB memory for this application. The results comparison among different compilers is shown in Figure 12 (c). It is observed that the performance of OpenUH is slightly better than CAPS compiler and much better than PGI compiler. This result is consistent with the performance difference while using the reduction testsuite, although we just used gang and vector in one loop instead of using gang, worker and vector in the testsuite.

5. RELATED WORK

Reduction is a well known topic, in this section, we will discuss some of the existing implementations of reduction operation in different programming models. Performing reduction is a challenge, different models may adopt different implementation strategies.

Reduction in OpenMP: In OpenMP programming model, the threads are one dimensional, hence there are not many use cases for reduction. Liao et al. [12] implemented the OpenMP reduction in two steps in the OpenUH compiler. In the first step, a reduction variable is substituted with a local copy in each thread to participate in the reduction operation computation. In the second step, values of local copies are summarized into the original reduction variable protected by a critical section. Usually the critical section is implemented via POSIX threads [7] mutex lock and unlock functions before and after a critical section. Mutex is one of the expensive locks in terms of performance. So Nanjegowda et al. [13] tried to eliminate the use of locks by using tree barrier and tournament barrier algorithms to improve the performance, but at the cost of additional data and temporary variables. Reduction in GCC compiler [6] is implemented by creating an array of the type of the reduction variable so that it can be indexed by the thread id, then each thread stores its reduction value into the array, finally the master thread iterates over the array to collect all private reduction values and generate the final reduction value. These approaches cannot be applied to OpenACC since threads in OpenMP are one dimensional but threads in OpenACC are three dimensional and OpenACC has no lock mechanism and cannot identify threads by any thread id.

Reduction in CUDA and OpenCL: Harris et al. [10] discussed seven different reduction algorithms in CUDA. Their optimizations include global memory coalescing to reduce memory divergence, shared memory optimization avoiding shared memory bank conflict, partial and full loop unrolling and algorithm cascading. They analyzed the cost of each algorithm and compared the performance of all reduction algorithms and achieved bandwidth close to the theoretical bandwidth. We have leveraged some of these algorithms in our work. OpenCL uses different reduction strategies. Catanzaro [3] took advantage of the associativity and commutativity properties of reduction operation to restructure a sequential loop into reduction trees and then used several strategies for building efficient reduction trees. They observed that most of their parallel reduction trees are very inefficient because of a number of communication and synchronization required among threads. Better performance can be achieved if much of the reduction is done serially.

Reduction in OpenACC: Komoda et al. [11] proposed a new directive in OpenACC to solve the array reduction problem and to overcome the limitation of supporting only scalar reduction in current OpenACC specification. The array reduction means that every element of an array needs to do reduction. Their array reduction implementation can work on both single GPU and multi-GPU platform. However, they do not mention the complexity of scalar reduction in OpenACC and all the possible reduction usages. Our work in this paper focuses on the scalar reduction in OpenACC standard.

6. CONCLUSION

In this paper, we present all possible reduction cases in OpenACC programming model, and the underlying parallelization implementations in an open-source OpenACC compiler OpenUH. We evaluat our implementation with a self-written reduction test suite and also use three real-world applications. We observed competitive performance while comparing OpenUH with the other two commercial OpenACC compilers. Unlike one of the commercial compilers that needed to add the reduction clause in multiple-level parallelism, OpenUH could detect the position where the reduction has to occur intelligently and the user is only required to add the reduction clause once. A similar reduction methodology can also be applied to other programming models such as OpenMP 4.0. OpenMP demonstrates two levels of parallelism and it just needs to ignore the worker if our implementation strategy is used.

7. ACKNOWLEDGMENTS

This work was supported in part by the NVIDIA and Department of Energy under Award Agreement No. DE-FC02-12ER26099. We would also like to thank PGI and CAPS for providing the compilers and support for the evaluation.

8. REFERENCES

[1] CUDA. `http://www.nvidia.com/object/cuda_home_new.html`, October 2013.

[2] OpenACC. `http://www.openacc-standard.org`, June 2013.

[3] OpenCL Reduction. `http://developer.amd.com/resources/documentation-articles/articles-whitepapers/opencl-optimization-case-study-simple-reductions/`, November 2013.

[4] OpenCL Standard. `http://www.khronos.org/opencl`, October 2013.

[5] OpenMP. `http://www.openmp.org`, October 2013.

[6] The GNU OpenMP Implementation. `http://gcc.gnu.org/onlinedocs/libgomp.pdf`, November 2013.

[7] D. Butenhof. *Programming with POSIX (R) Threads*. Addison-Wesley Professional, 1997.

[8] S. Cook. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Newnes, 2012.

[9] R. Dolbeau, S. Bihan, and F. Bodin. HMPP: A Hybrid Multi-core Parallel Programming Environment. In *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU 2007)*, 2007.

[10] M. Harris. Optimizing Parallel Reduction in CUDA. *NVIDIA Developer Technology*, 6, 2007.

[11] T. Komada, S. Miwa, H. Nakamura, and N. Maruyama. Integrating Multi-GPU Execution in an OpenACC Compiler. In *ICPP '13: Proceedings of the 42nd International Conference on Parallel Processing*, pages 260–269, 2013.

[12] C. Liao, O. Hernandez, B. M. Chapman, W. Chen, and W. Zheng. OpenUH: An Optimizing, Portable OpenMP Compiler. *Concurrency and Computation: Practice and Experience*, 19(18):2317–2332, 2007.

[13] R. Nanjegowda, O. Hernandez, B. Chapman, and H. H. Jin. Scalability Evaluation of Barrier Algorithms for OpenMP. In *Evolving OpenMP in an Age of Extreme Parallelism*, pages 42–52. Springer, 2009.

[14] G. Pullan. Cambridge cuda course 25-27 may 2009. `http://www.many-core.group.cam.ac.uk/archive/CUDAcourse09/`.

[15] X. Tian, R. Xu, Y. Yan, Z. Yun, S. Chandrasekaran, and B. Chapman. Compiling A High-Level Directive-based Programming Model for Accelerators. In *LCPC 2013: The 26th International Workshop on Languages and Compilers for Parallel Computing*, 2013.

[16] N. Whitehead and A. Fit-Florea. Precision & Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs. *nVidia technical white paper*, 2011.

[17] R. Xu, S. Chandrasekaran, B. Chapman, and C. F. Eick. Directive-based Programming Models for Scientific Applications-A Comparison. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion*, pages 1–9. IEEE, 2012.