CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. (2015)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.3648

SPECIAL ISSUE PAPER

Compiler transformation of nested loops for general purpose GPUs

Xiaonan Tian*,†, Rengan Xu, Yonghong Yan, Sunita Chandrasekaran, Deepak Eachempati and Barbara Chapman

Department of Computer Science, University of Houston, Houston, TX, 77204, USA

SUMMARY

Manycore accelerators have the potential to significantly improve performance of scientific applications when offloading computationally intensive program portions to accelerators. Directive-based high-level programming models, such as OpenACC and OpenMP, are used to create applications for accelerators through annotating regions of code meant for offloading. OpenACC is an emerging directive-based programming model for programming accelerators that typically enable inexperienced programmers to achieve portable and productive performance within applications. In this paper, we present our research in developing challenges and solutions when creating an open-source OpenACC compiler in an industrial framework (OpenUH as a branch of Open64). We then discuss in detail techniques we developed for loop scheduling reduction operations on general purpose GPUs. The compiler is evaluated with benchmarks from the NAS Parallel Benchmarks suite and self-written micro-benchmarks for reduction operations. This implementation has been designed to serve as a compiler infrastructure for researchers to explore advanced compiler techniques, extend OpenACC to other programming models, and build performance tools used in conjunction with OpenACC programs. Copyright © 2015 John Wiley & Sons, Ltd.

Received 23 April 2014; Revised 2 December 2014; Accepted 17 July 2015

KEY WORDS: OpenUH; OpenACC; loop scheduling; reduction; compiler

1. INTRODUCTION

Heterogeneous architectures that are comprised of commodity Central Processing Units (CPUs)s and computational accelerators, such as General Purpose Graphics Processing Units (GPGPUs), have been increasingly adopted in large-scale supercomputers, workstations, and desktops for engineering and scientific applications. These accelerators provide additional massively parallel computing capabilities to users while preserving the flexibility provided by CPUs for different workloads. However, effectively exploiting GPUs' full potential may become difficult due to the programming challenges faced when mapping computational algorithms to hybrid and heterogeneous architectures.

Programming models, such as CUDA [1] and OpenCL [2], for GPGPUs offer programming interfaces with execution models that closely match GPGPU architectures. Effectively utilizing these interfaces to create highly optimized applications requires programmers to thoroughly understand the underlying architectures. In addition, they must be able to significantly change and adapt program structures and algorithms. This affects both productivity and performance. An alternative approach would be to use high-level directive-based programming models to achieve the same goal, for example, HMPP [3], OpenACC [4], and OpenMP [5]. These models allow the user to insert both directives and runtime calls into existing source code, indicating that a portion of the Fortran or

*Correspondence to: Xiaonan Tian, Department of Computer Science, University of Houston, Houston, TX, 77204, USA.
†E-mail: xtian2@uh.edu

C/C++ code should execute on accelerators. Using directives, programmers may give hints to compilers to perform certain transformations and optimizations on the annotated code regions. The user can insert directives incrementally to parallelize and optimize the application, enabling a productive migration path for legacy code.

In this paper, we present our experiences in constructing an OpenACC compiler in the OpenUH open-source compiler framework [6]. Our goals are to enable a broader community participation and dialog related to this programming model and the compiler techniques that support it. We also design and specify this implementation to serve as compiler infrastructure for researchers that are interested in improving OpenACC, extending the OpenACC model to other programming languages, and/or building tools that support the development of OpenACC programs. To the best of our knowledge, we are the first to design and implement an open-source OpenACC compiler to support comprehensive reduction algorithms and read-only data cache (RODC) optimization.

We also propose the design and parallelization of reduction operations in parallel loops for GPGPU accelerators. Using OpenACC as a high-level directive-based programming model, we discuss how reduction operations are parallelized when appearing in each level of the loop nest and thread hierarchy, for example, the outer loop with `gang` (coarse-grain), mid-level with `worker` (fine-grain), and inner level with `vector` parallelism. Then the manner in which mapping of the loops and parallelized reduction to single- or multiple-level parallelism of GPGPU architectures is further detailed. We implemented these algorithms in OpenUH [7] and created a test suite that provides different use cases of reduction operations for performance evaluation. We then compare our results for these cases with two other commercial OpenACC compilers. OpenUH passed all reduction usage cases and delivered competitive performance to other compilers. OpenUH even passed multiple tests where the commercial OpenACC compilers failed.

This paper makes the following contributions:

- We deliver an open-source OpenACC research compiler based on the robust Open64 industry-level compiler framework.

  Thus, the experiences could be applicable to other compiler implementation efforts. The OpenUH compiler adopts a source-to-source approach and generates readable CUDA source code for GPGPUs. This gives users an opportunity to understand the applications of loop mapping mechanisms, which can be used to further manually optimize the code, if need be. It also allows the user to leverage the advanced optimization features offered by CUDA in the back end.
- We propose a rich set of loop scheduling strategies within the compiler to efficiently distribute kernels or parallel loops to the threading architectures of GPGPU accelerators. Our findings provide guidance for users to adopt suitable loop schedules depending on the application requirements.
- We present the compile-time read-only array/pointer detection for the read-only cache optimization in the latest Nvidia Kepler architecture.
- We provide a comprehensive solution for reduction operations by covering all reduction operator and operand data types.

  While evaluating our complete implementation strategies for reduction operations in OpenUH compiler against other vendor compilers, we observed that not all vendor compilers provide a thorough successful solution for this operation.
- We evaluate our novel strategies and its implementations in OpenUH OpenACC compiler using NAS parallel benchmarks (NPBs) [8]. Then comparisons against CUDA code version [9] and PGI [10] compiler are presented. The results show that OpenUH generates competitive performance to CUDA and modest performance gains over PGI.

The organization of this paper is as follows: Section 3 provides an overview of the OpenACC model and compiler design. Section 4 demonstrates in detail how to transform sequential loops into CUDA code for massively parallel GPGPU architecture. Section 5 discusses the parallelization strategies for reduction operations in parallel loops. Section 6 explains the compile-time read-only data optimization for offload regions. Performance results are discussed in Section 7. Section 8 highlights the related work in this area. We then conclude our work in Section 9.

DOI: 10.1002/cpe

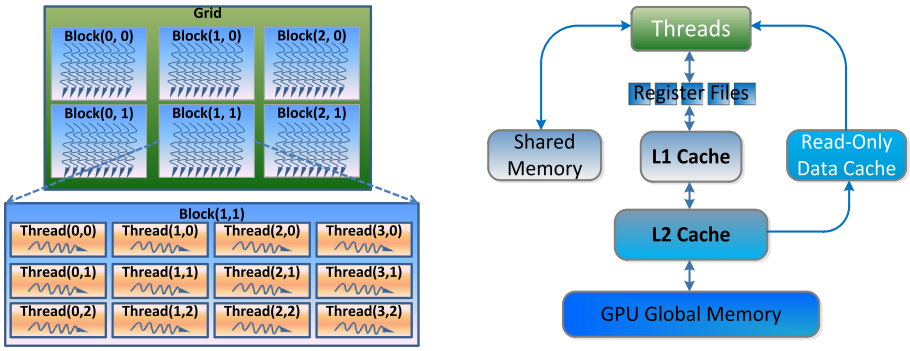
2. BACKGROUND

2.1. Nvidia GPGPU architecture

The processor architecture of GPUs and CPUs are fundamentally different. In this paper, GPU is interchangeably referred to as GPGPU, or general purpose GPU. Nvidia's GPU consists of multiple streaming multiprocessors (SMs), and each SM consists of many scalar processors (referred to as cores). In the latest Kepler architecture, SM is updated to the next-generation streaming multiprocessor, or SMX. Each GPU supports the concurrent executions of hundreds to thousands of threads following the single-program multiple-data programming model, and each thread is executed by a core. The smallest scheduling and execution unit is called a warp, which has 32 threads. Warps of threads are grouped together into a thread *block*, and blocks are grouped into a *grid*. Figure 1(a) [11] shows how the blocks can be organized into a two dimensional grid of thread blocks. Table I provides CUDA terminology for thread, block, and grid topology information. Each thread has its own unique thread ID, which can be identified by threadIdx.x, threadIdx.y, and threadIdx.z in each block. Thread blocks cannot synchronize with each other, while the threads within a block can do so.

Figure 1(b) gives an overview of the memory hierarchy in Nvidia GPUs. The GPU has a global memory space that is accessible by all threads in the grid, and this is the space that the CPU memory can communicate with. Shared memory is allocated per thread block, whose memory size can be configured by the programmer. Because the shared memory is on-chip, the latency is much lower than global memory. The L1 cache in the Kepler architecture is reserved only for local memory accesses, such as register spilling and stack data. Global loads are cached in L2 only. Read-only data cache was introduced in the latest Kepler architecture. Each SMX has a 48 KB read-only data cache. Each core in SMX accesses data via read-only cache when the data is read-only for the lifetime of the CUDA kernel. Our compiler, OpenUH, uses the on-chip shared memory for performing OpenACC reduction operations, discussed in Section 5. OpenUH also includes OpenACC optimizations for exploiting the GPU cache, discussed in Section 6.

The use of GPGPU programming for achieving greater performance has been made possible by programming models/languages such as CUDA and OpenCL. However, the challenge is that the



(a) Nvidia GPGPU Thread Block Hierarchy

(b) Nvidia GPGPU Memory Hierarchy

Figure 1. Nvidia Kepler general purpose GPU (GPGPU) architecture. (a) Nvidia GPGPU thread block hierarchy; (b) Nvidia GPGPU memory hierarchy.

Table I. CUDA terms.

Term	Description
threadIdx.x(y/z)	Thread index in $x(y/z)$ dimension of a thread block
blockDim.x(y/z)	Number of threads in $x(y/z)$ dimension of a thread block
blockIdx.x(y/z)	Block index in $x(y/z)$ dimension of the grid
gridDim.x(y/z)	Number of blocks in $x(y/z)$ dimension of the grid

DOI: 10.1002/cpe

programmers are required to thoroughly understand the GPUs. These models also require rewriting of significant portions of the application source code. This is time-consuming and error-prone. OpenACC, an emerging high-level and directive-based parallel programming standard, addresses some of these challenges by providing a more viable approach for improving the productivity and simplifying the process of porting or creating applications for using accelerators.

2.2. OpenACC programming model

OpenACC is a high-level programming model that can be used to port High Performance Computing (HPC) applications to different types of accelerators such as Nvidia GPUs, AMD GPUs and Accelerated Processing Units (APUs), and Intel Xeon Phi. It provides directives, runtime routines, and environment variables as its programming interfaces. The execution model assumes that the main program runs on the host, while the compute-intensive regions of the program are offloaded to the attached accelerator. The accelerator and the host may have separate memories, and the data movement between them may be controlled explicitly. OpenACC provides a rich set of data transfer directives, clauses, and runtime calls as part of its standard. To minimize the performance degradation due to data transfer latency, OpenACC also allows asynchronous data transfer and asynchronous computation with the CPU code, thus enabling overlapping data movement and computation.

OpenACC uses the `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 the distribution of iterations. 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. Some additional data directives and runtime routines to control the unstructured data lifetime have been added in the 2.0 specification.

We will briefly distinguish between the *kernel* in CUDA and *kernels* directive in OpenACC. CUDA `kernel` identifies a kernel function. The OpenACC `kernels` construct instructs the compiler to analyze the code and determine which code regions may be executed as kernels on the accelerator. We denote both parallel and kernels regions as 'offload' region in this paper.

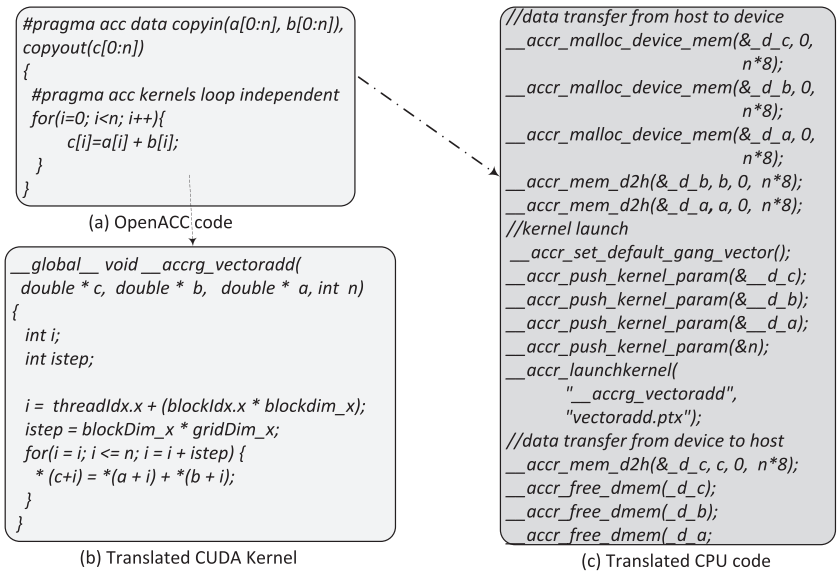


Figure 2. OpenACC vector addition example. (a) OpenACC code; (b) translated CUDA kernel; (c) translated CPU code.

DOI: 10.1002/cpe

Figure 2(a) shows a simple example of OpenACC vector addition. The `acc data` directive, which identifies a data region, will create `a` and `b` in device memory and then copy the respective data into device memory at the beginning of the data region. The array `c` will be copied out after finishing the code segment of the region. The `acc kernels` directive means that the following block should be executed on the device. The `acc loop` directive results in the distribution of loop iterations among the threads on the device.

3. THE DESIGN OF OPENACC COMPILER

The creation of an OpenACC compiler requires innovative research solutions to meet the challenges of mapping high-level loop iterations to low-level threading architectures of the hardware. It also requires support from the runtime to handle data movement and scheduling of computation on the accelerators. In this work, we have used OpenUH, a branch of the open-source Open64 compiler suite, for our compiler framework. The components of the OpenUH framework is shown in Figure 3. The compiler is comprised of several modules, with each module operating on a level of the compiler's multi-level Intermediate Representation (IR), WHIRL. From the top, each module translates the current level of WHIRL to a lower level.

We have identified the following challenges that must be addressed to create an effective implementation of the OpenACC directives. First, it is very important that we create an extensible parser and IR system to facilitate the addition of new features as and when the language evolves and to support aggressive compiler optimizations. Fortunately, the extensibility of OpenUH framework and WHIRL IR allowed us to add these extensions without too much difficulty. Second, we need to design and implement an effective strategy to distribute the loop nest across the GPGPU thread hierarchy. We discuss our solutions in more detail in Section 4.

As shown in Figure 3, we use a source-to-source translation technique to translate OpenACC offload region into CUDA code. OpenUH directly generates an object code for ×86 host CPU. Consider the OpenACC code in Figure 2(a) as an example. Figure 2(b) and (c) shows the translated CUDA kernel and the equivalent host CPU pseudocode. We have created a WHIRL2CUDA tool that can produce NVIDIA CUDA kernels after the transformation of offloading code regions. Compared with binary code generation, the source-to-source approach provides much more flexibility to users. It allows users to leverage the advanced optimization features in the back-end compilation step performed by the CUDA compiler *nvcc*. It also gives users some options to manually optimize the generated CUDA code for further performance improvement.

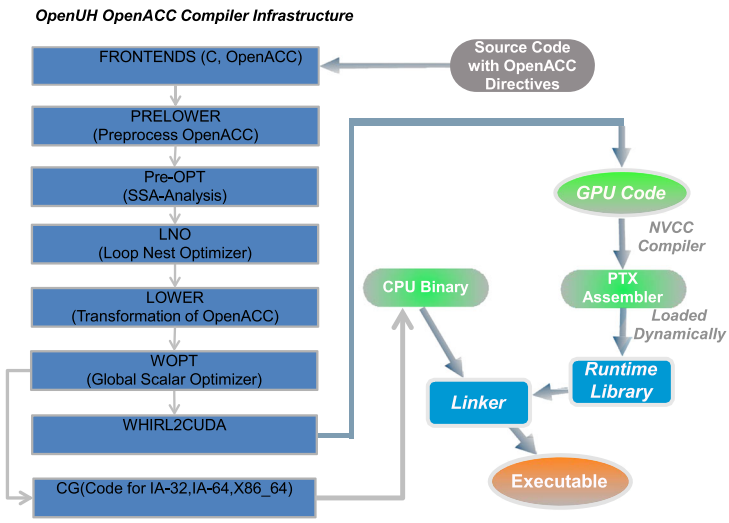


Figure 3. OpenUH compiler framework for OpenACC.

DOI: 10.1002/cpe

4. LOOP TRANSFORMATION

Programmers usually would want to offload computationally intensive loop nests for execution on massively parallel accelerator devices. One of the major challenges of compiler transformation is to create a uniform loop distribution mechanism that can effectively map loop nest iterations across the GPU parallel system. As an example, Nvidia GPGPUs has two levels of parallelism: block level and thread level. Blocks can be organized as multi-dimensional within a grid, and threads in a block can also be multi-dimensional. Distributing iterations of multi-level loop nests across these multi-dimensional blocks and threads is a nontrivial task.

OpenACC provides three levels of parallelism for mapping loop iterations onto the accelerators' thread structures: 'gang' for coarse-grain parallelism, 'worker' for fine-grain parallelism, and 'vector' for vector parallelism. The OpenACC standard allows the compiler some flexibility of interpretation and code generation for these levels. For Nvidia GPU, some compilers map each gang to a thread block, vector to threads in a block, and ignore worker [10]; other compilers map gang to the *x*-dimension of a grid block, worker to the *y*-dimension of a thread block, and vector to the *x*-dimension of a thread block [12]. There are also compilers that map each gang to a thread block, worker to warp, and vector to Single Instruction Multiple Threads (SIMT) group of threads [13].

In our design, we propose two different loop scheduling strategies for an OpenACC offload region: kernels and parallel loop scheduling. The difference between parallel and kernels loop scheduling is that we provide more options for the compiler to fine-tuning scheduling on GPGPUs in kernels region. Both types of loop scheduling can be used for a single loop, double-nested loop, and triple-nested loop as shown in Figure 4. If the depth of a nested loop is more than 3, the OpenACC `collapse` clause can be used to increase the parallelism. In Figure 4, OpenACC loops do not need to specify loop scheduling clauses such as gang, worker, and vector. We discuss these cases in detail in the following two sections.

4.1. Parallel loop scheduling

In the OpenACC specification, loop scheduling cannot use nested gang, nested worker, and nested vector; that is, a gang can only contain worker and vector, and a worker can only include 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 more levels of parallelism provided by the GPGPU device.

Table II shows the CUDA terminology that we use in our OpenACC implementation. In OpenUH, `gang` maps to a thread block, `worker` maps to the *y*-dimension of a thread block, and `vector`

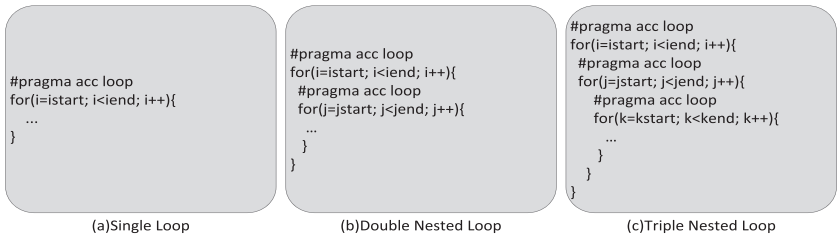


Figure 4. OpenACC loop. (a) Single loop; (b) double-nested loop; (c) triple-nested loop.

Table II. OpenACC and CUDA terminology mapping in parallel.

OpenACC clause	CUDA	Description
gang	block	Loop iterations are distributed across blocks in grid.
worker	y-dim threads	Loop iterations are distributed into *y*-dimension threads in a block.
vector	x-dim threads	Loop iterations are distributed into *x*-dimension threads in a block.

DOI: 10.1002/cpe

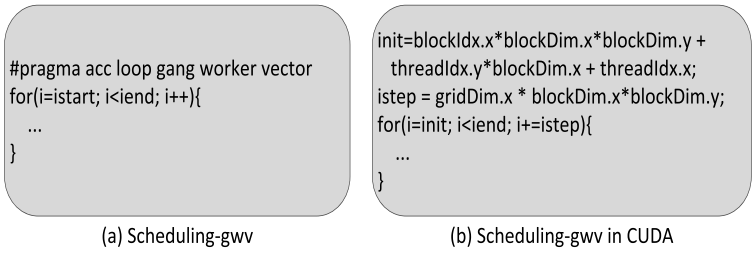


Figure 5. Single loop transformation in parallel region. (a) Scheduling-gwv; (b) scheduling-gwv in CUDA. g, gang; w, worker; v, vector.

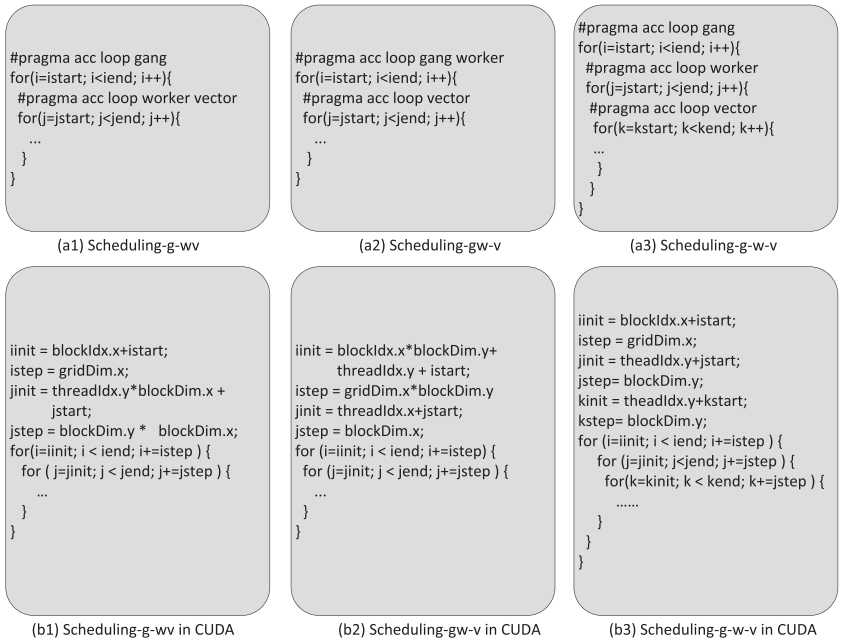


Figure 6. Double-nested and triple-nested loop transformation in parallel region. The scheduling name indicates which loop scheduling clauses are used. For example, scheduling-gw-v means the outer loop is distributed across gang and worker, and inner loop is carried by vector in a double-nested loop. The kernels loop scheduling names follow the same format. (a1) Scheduling-g-wv; (a2) scheduling-gw-v; (a3) scheduling-g-w-v; (b1) scheduling-g-wv in CUDA; (b2) scheduling-gw-v in CUDA; (b3) scheduling-g-w-v in CUDA. g, gang; w, worker; v, vector.

maps to the *x*-dimension of a thread block. Based on these definitions, the implementation details for the loop nest is shown in Figures 5 and 6. This mapping strictly follows the OpenACC standard.

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 there being a limited number of threads available in the hardware platform. Our implementation is designed in a way that is independent of the number of threads used in each loop level.

In the loop nest example, we assume that all the iterations are independent, but most of the time, the loop nest may contain a reduction operation, and the reduction may appear anywhere in the loop nest. In Section 5, we will discuss such cases and how they are parallelized in OpenUH.

The limitation of the parallel loop mapping is that the total threads (the number of workers multiplied by the number of vectors) must be less than or equal to 1024 in one gang (block). The number 1024 here is the maximum allowed number of threads in one block in an Nvidia GPU; therefore, the scalability becomes a critical issue. In Figure 7(a), the maximum number threads we can create in OpenACC is 20*1024. We could overcome this limitation by utilizing the multi-dimensional

DOI: 10.1002/cpe

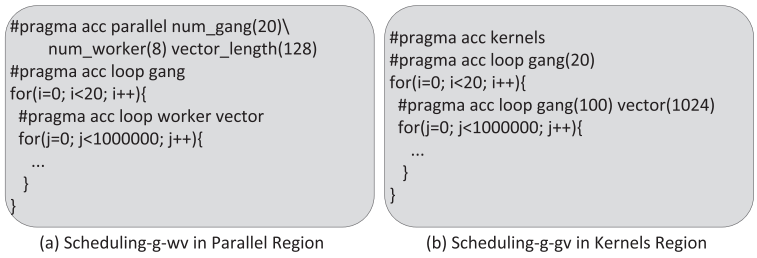


Figure 7. Parallel loop scheduling limitation and its solution. (a) Scheduling-g-wv in parallel region; (b) scheduling-g-gv in kernels region. g, gang; w, worker; v, vector.

topology of the thread block and grid. The solution is to create some extension in kernels loop scheduling to create more threads, in order to increase the thread-level parallelism. Figure 7(b) is one of the kernels loop schedules that can solve the scalability issue because it can create 20*100*1024 threads.

4.2. Kernels loop scheduling

In order to take advantage of multi-dimensional grid and thread block in Nvidia GPGPUs, we propose eight loop scheduling strategies to efficiently distribute loop iterations in an OpenACC kernels region. However, the kernels loop scheduling strategies are not limited to these eight methods; there could be more ways to take advantage of the massively parallel threading that the GPGPU offers. Table III shows the mapping terminology we used from OpenACC to CUDA for kernels directives.

Single Loop Loop iterations are distributed among gangs and vectors. Both gang and vector are one-dimensional. Each thread takes one iteration at a time and then moves ahead with *blockDim.x * gridDim.x* stride. Figure 8 show the OpenACC and translated CUDA loop for this single loop.

Double-Nested Loop Notations: g, gang; v, vector. There are four different double-nested loop cases, and the mapping algorithms are different for these. Figure 9 shows the loop scheduling for each case.

- Scheduling-g-v (in Figure 9(a1)): both gang and vector are one-dimensional. The outer loop is distributed across the gang, and the inner loop is executed among threads in each gang. The thread stride in i and j axis are *gridDim.x* and *blockDim.x*. The translated CUDA kernel is shown in Figure 9(b1).

Table III. OpenACC and CUDA terminology mapping in kernels.

OpenACC clause	CUDA	Description
gang (integer expression)	block	If there is an integer expression for this gang clause, it defines the number of blocks in one dimension of grid.
vector (integer expression)	thread	If there is an integer expression for this vector clause, it defines the number of threads in one dimension of block.

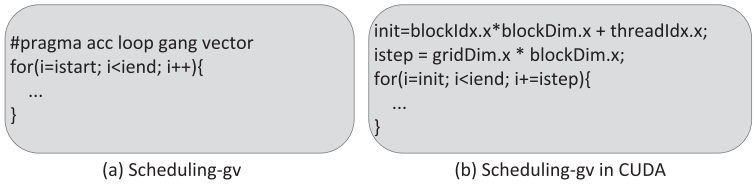


Figure 8. Single loop transformation in kernels region. (a) Scheduling-gv; (b) scheduling-gv in CUDA. g, gang; v, vector.

DOI: 10.1002/cpe

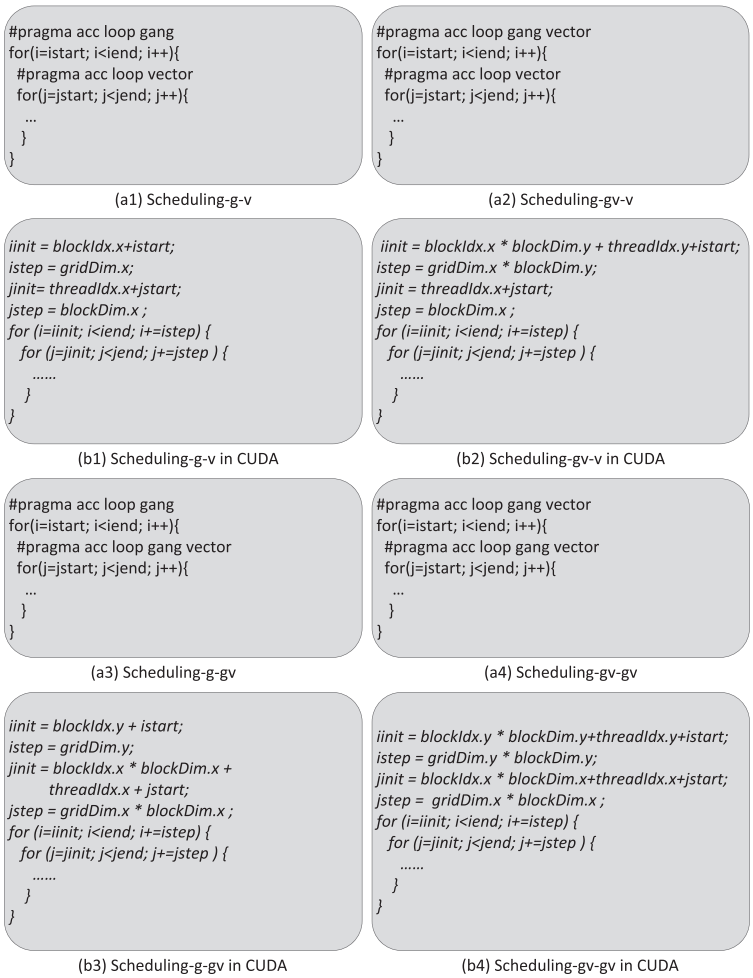


Figure 9. Translation of double-nested loops in kernels region. (a1) Scheduling-g-v; (a2) scheduling-gv-v; (b1) scheduling-g-v in CUDA; (b2) scheduling-gv-v in CUDA; (a3) scheduling-g-gv; (a4) scheduling-gv-gv; (b3) scheduling-g-gv in CUDA; (b4) scheduling-gv-gv in CUDA. g, gang; v, vector.

- Scheduling-gv-v (in Figure 9(a2)): one-dimensional gang, two-dimensional vector. After the mapping, the outer loop thread stride is *gridDim.x*∗*blockDim.y*, and the inner loop thread stride is *blockDim.x*. The translated CUDA kernel is shown in Figure 9(b2).
- Scheduling-g-gv (in Figure 9(a3)): two-dimensional gang an d one-dimensional vector. After the mapping, the outer loop thread stride is *gridDim.y*, and the inner loop thread stride is $gridDim.x * blockDim.x$. The translated CUDA kernel is shown in Figure 9(b3).
- Scheduling-gv-gv (in Figure 9(a4)): both gang and vector are two-dimensional. After the mapping, the outer loop thread stride is $gridDim.y * blockDim.y$, and the inner loop thread stride is $gridDim.x * blockDim.x$. The translated CUDA kernel is shown in Figure 9(b4).

Triple-Nested Loop For the three different triple-nested loops, Figure 10 shows the mapping.

- Scheduling-g-gv-v (in Figure 10(a1)): both gang and vector are two-dimensional. After the mapping, the thread stride in i, j, and k loop are *griddim.x*, $griddim.y * blockDim.y$, and *blockDim.x*. The translated CUDA kernel is shown in Figure 10(b1).
- Scheduling-v-gv-gv (in Figure 10(a2)): two-dimensional gang and three-dimensional vector. After the mapping, the thread stride in i, j, and k loops are *blockDim.z*, $blockDim.y *$

DOI: 10.1002/cpe

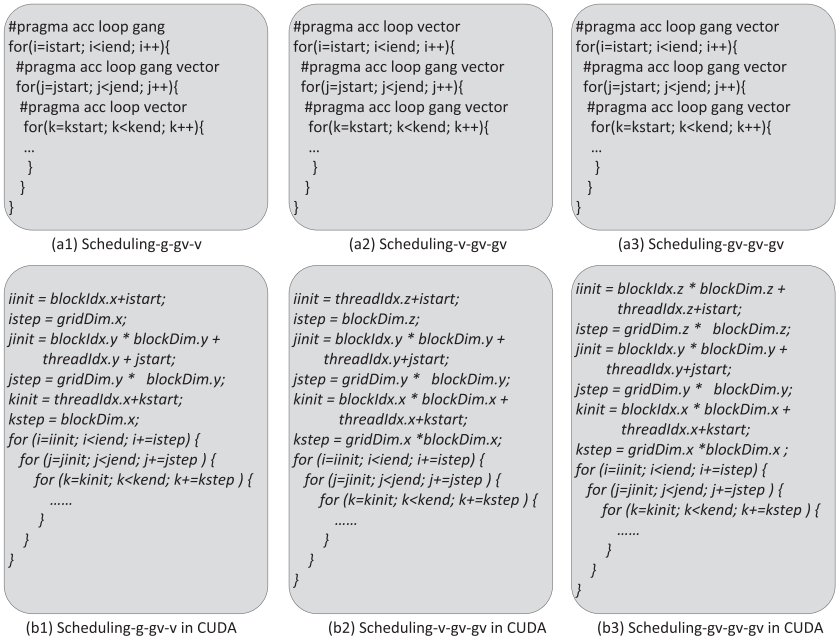


Figure 10. Translation of triple-nested loops in kernels region. (a1) Scheduling-g-gv-v; (a2) scheduling-v-gv-gv; (a3) scheduling-gv-gv-gv; (b1) scheduling-g-gv-v in CUDA; (b2) scheduling-v-gv-gv in CUDA; (b3) scheduling-gv-gv-gv in CUDA. g, gang; v, vector.

griddim.y, and *blockDim.x* $*$ *griddim.x*, respectively. The translated CUDA kernel is shown in Figure 10(b2).

- Scheduling-gv-gv-gv (in Figure 10(a3)): both gang and vector are three-dimensional. After the mapping, the thread stride in i, j, and k axes are *griddim.z* $*$ *blockDim.z*, *griddim.y* $*$ *blockDim.y*, and *gridDim.x*$*$*blockDim.x*, respectively. The translated CUDA kernel is shown in Figure 10 (b3).

The loop scheduling strategies proposed for kernels regions are not valid in OpenACC. However, the intention of the kernels construct is to let the compiler analyze and automatically parallelize loops, as well as select the best loop scheduling strategy. So far, OpenUH requires the user to explicitly specify these loop schedules, but we are planning to automate this work in the compiler so that the chosen loop scheduling is transparent to the user and the source code also follows the OpenACC standard. Another issue lies with the reduction operation in kernels computation regions. If the reduction is used in the inner loop of scheduling-gv-gv, it would require synchronization across thread blocks, and such synchronization is not supported in Nvidia GPGPUs. Hence, reduction is not supported in a kernels computation region. The work-around is to use the reduction clause in a parallel region.

5. 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 copy of a segment of the input array when the loop is distributed among threads. The operation that consolidates the results from the thread-local copies of the segments using the reduction operation is the topic that we addressed in [14]. The approach to performing parallel reductions depends on how the loop nests are mapped to the GPGPU's thread hierarchy. Moreover, the 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.

DOI: 10.1002/cpe

5.1. *Reduction in single-level thread parallelism*

The loop nest where a reduction operation is applied could be mapped to one or multiple levels in the thread hierarchy – gang, worker, and vector. We will first discuss the case where the reduction operation appears in only one level of the parallelism.

5.1.1. Reduction only in vector. Figure 11(a) shows an example of reduction occurring only in vector, where the worker and gang loops (*k* and *j*) can be executed in parallel while the vector loop (*i*) needs to perform the reduction. To parallelize vector reduction, Figure 12(a) shows the data, worker, and vector thread layout in one gang before performing the reduction operation. Each row is one worker that includes multiple vector threads. Because vector reduction happens in each worker, each row needs to do the reduction, and finally, each worker should have one reduction result. In this example, there are four workers, and each worker has four vector threads, so four vector reduction results should be generated. Because NVIDIA GPUs provide very low latency shared memory access and the reduction needs to fetch the data in multiple iterations, the reduction can be moved to the shared memory to reduce memory access latency. The parallelization strategy for vector reduction is shown in Figure 12(b). In this implementation, each vector thread first creates a private variable and does the partial reduction itself, and then all the partial private reduction values are moved into the shared memory. Notice that the data layout and thread layout in the shared memory still remain the same as that in the global memory. The reason that we retain the data and thread layouts is that the shared memory is composed of multiple banks (32 banks in Kepler), and this implementation can avoid bank conflict issues. A bank conflict occurs when threads in a warp

```
#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

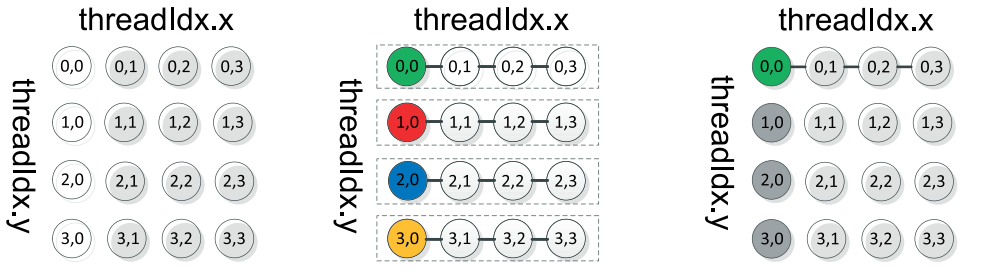
```
#pragma acc loop gang
for(k=0; k<NK; k++){
 int j_sum = k;
 #pragma acc loop worker \
              reduction(+:j_sum)
 for(j=0; j<NJ; j++){
  #pragma acc loop vector
  for(i=0; i<NI; i++)
    temp[k][j][i] =  input[k][j][i];
 j_sum += temp[k][j][0];
 }
 temp[k][0][0] = j_sum;
}
```

(b)Reduction in Worker

```
sum = 0;
#pragma acc 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 11. Reduction in single-level thread parallelism example. (a) Reduction in vector; (b) reduction in worker; (c) reduction in gang.



(a) Data and threads layout in global memory

(b) Data and threads layout in shared memory for vector reduction

(c) Data and threads layout in shared memory for worker reduction

Figure 12. Parallelization comparison for vector and worker reduction. (a) includes the original thread layout in a thread block. In (b), each group of vector threads works on row data and the reduction results are stored in the first column. The data inside the dashed rectangle are distributed into different shared memory banks to avoid bank conflict issues. In (c), four worker values are only in the first row, and the following three row threads are inactive; the final reduction is stored in the first element of the first row.

DOI: 10.1002/cpe

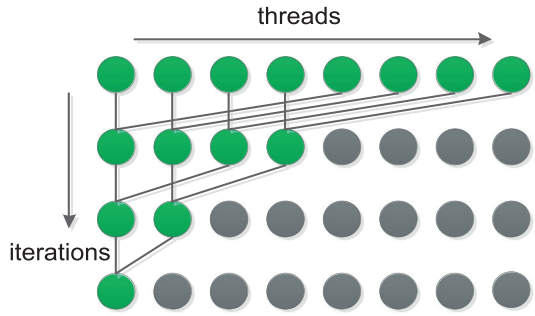


Figure 13. Interleaved log-step reduction. Synchronization is inserted after each iteration and before the next iteration. Green represents active threads while gray represents inactive threads in each iteration.

request different words (one word is 64-bit in Kepler) from the same bank in a single request. In our vector reduction implementation, the vector reduction happens in each row, and the final reduction values are stored in the first column of the shared memory. Therefore, the threads performing the reduction request data from multiple banks rather than a single bank, thus avoiding the bank conflict. The reduction algorithm used by the vector threads in shared memory is called interleaved log-step reduction [15], which is shown in Figure 13. Our implementation of this algorithm uses sequential addressing, loop unrolling, and a algorithm cascading optimization as discussed in [15]. 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 11(a) is *j*, but the initial value for the private copy of the variable *i_sum_priv* for each thread is 0. In most of implementations, the initial value is processed after the vector reduction operation is finished. For instance, the initial value is summed for '+' reduction or multiplied for '*' reduction.

5.1.2. Reduction only in worker. Figure 11(b) shows an example of reduction occurring only in worker, where the gang and vector loops (*k* and *i*) can be executed in parallel while the worker loop (*j* loop) has to do the reduction. The parallelization strategy for worker reduction is shown in Figure 12(c). In this implementation, each worker creates a private variable and does the private partial reduction first; then the first vector thread of each worker stores the partial reduction into the shared memory; finally, all the original vector threads of workers use the interleaved log-step reduction algorithm to generate the final reduction result. Using this approach requires less threads and less shared memory so that we can leave more shared memory for other computations. For instance, in the example of Figure 12(c), only four threads are required to perform 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 six iterations when only the last warp threads need to perform the reduction.

5.1.3. Reduction only in gang. The example of reduction only in gang is shown in Figure 11(c), where the inner worker and vector loops (*j* and *i*) are executed in parallel while the gang loop (*k* loop) has a reduction. Because we map each gang to each thread block in CUDA, and there is no efficient synchronization mechanism to synchronize all thread blocks, the strategy of OpenUH is to (1) create a temporary buffer with the size equal to the number of gangs, (2) have each block write its partial reduction into the specific entry of the buffer, and (3) another kernel (the same reduction kernel as the one in vector addition) is launched to do the reduction within only one block.

We mentioned partial reduction in all three cases of vector, worker, and gang reduction. OpenUH uses a sliding window technique to implement all of these partial reductions. Consider gang reduction for example. OpenUH considers all gangs as a sliding window, and this window slides through the iteration space. An alternative blocking algorithm is to divide the iterations among all gangs equally, then each gang works on the assigned iteration chunk. Essentially, there is no difference

DOI: 10.1002/cpe

between these two algorithms in gang partial reduction, but the sliding window technique is superior than a blocking algorithm in vector partial reduction because it can enable memory coalescing. Memory coalescing is impossible in worker and gang partial reductions, but we still use the window sliding technique in both cases in order to make the implementation consistent.

5.2. Reduction across multi-level thread parallelism

Section 5.1 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 parallelism can occur within different loops or within the same loop. Next, we will discuss all these possibilities in detail.

5.2.1. Reduction across multi-level thread parallelism in different loops. Figure 14(a) 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 an incorrect result is generated. This is also a favorable step because reduction occurs in both worker and vector level parallelism. The OpenUH compiler, however, can automatically detect the position of the reduction variable, and the user just needs to add the 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 the vector loop. The CAPS compiler, at times, also just needs to add the reduction clause to the outermost loop, but only when all the inner loops are sequential. With respect to the implementation, OpenUH creates a buffer with the size equal to the number of all threads that needs to perform 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-level parallelism can be realized in three ways: gang & worker, gang & worker & vector, and worker & vector. For the former two cases, a temporary buffer is created, and all threads performing the 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 because 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.

5.2.2. Reduction across multi-level thread parallelism in the same loop. Figure 14(b) shows an example of a 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 perform the reduction (gangs * workers * vector threads in this example), then each thread does its own partial

```
#pragma acc loop gang
for(i=istart; i<iend; i++){
  #pragma acc loop worker reduction(+: j_sum)
  for(j=jstart; j<jend; j++){
   #pragma acc loop vector
    for(k=kstart; k<kend; k++){
              j_sum += input[k][j][i];
    }
  }
}
```

(a) Example of RMP in the Different Loop

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

(b) Example of RMP in the Same Loop

Figure 14. Examples of reduction across multi-level thread parallelism (RMP). (a) Example of RMP in the different loop; (b) example of RMP in the same loop.

DOI: 10.1002/cpe

reduction, and finally, another kernel is launched 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 using gang parallelism. As long as gang parallelism is involved, the buffer must be in global memory.

5.3. Special reduction considerations

Apart from the cases listed in Sections 5.1 and 5.2, 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 approach 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 because too many reduction variables may require more shared memory than the hardware limit. OpenUH, however, creates a shared memory space with the size being the same as the largest required shared memory for a particular data type. For instance, if there is an 'int' type reduction and a 'double' type reduction in the same reduction clause, then we 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 reductions can share the same shared memory space.

We implemented the different cases of reduction operations 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.

6. READ-ONLY DATA CACHE OPTIMIZATION

The read-only data cache (RODC) was introduced in the latest Nvidia Kepler architecture. For each SMX, the 48 KB RODC is available for the lifetime of a CUDA kernel. The RODC has higher bandwidth and lower latency than the read-write cache and supports full-speed unaligned memory access patterns. However, the user cannot control how the hardware actually caches the read-only data. In the NVIDIA CUDA compiler, the user can give hints by using the 'const' modifier to designate certain read-only data and the '__restrict__' keyword to indicate no aliasing. The CUDA compiler will generate code to help the hardware to cache the corresponding data into RODC. We propose and implement in OpenUH two ways to effectively take advantage of RODC in high-level programming models.

Figure 15 gives an example of how the proposed clause is used and the translated CUDA code. First, a new data clause is introduced to explicitly identify the read-only data by users. We named it

```
#pragma acc data copyin(a[0:n], b[0:n]),
copyout(c[0:n])
{
 #pragma acc kernels loop independent const(a,b)
  for(i=0; i<n; i++){
        c[i]=a[i] + b[i];
  }
}
```

(a) OpenACC code

```
__global__ void __accrg_vectoradd(
  double * __restrict__ c,
 const double * __restrict__  b,
 const  double * __restrict__  a, int  n){
  int I, istep;
  i =  threadIdx.x + (blockIdx.x * blockdim_x);
  istep = blockDim_x * gridDim_x;
  for(i = i; i <= n; i = i + istep) {
    * (c+i) = *(a + i) + *(b + i);
  }
}
```

(b) Translated CUDA Kernel

Figure 15. Read-only data cache optimization. (a) OpenACC code; (b) translated CUDA kernel.

DOI: 10.1002/cpe

as ‘const’, which is only valid in the kernels/parallel construct. Users can determine which offload region needs the ‘const’ clause. Second, the OpenACC compiler can do some analysis for the offloaded region, check if an array/pointer buffer will be written, and then generate a read-only array/pointer variables list. In this scenario, aliasing can prevent the compiler from making the optimal decision. However, such aliasing issues can be easily solved by using the OpenACC loop clause ‘independent’, which tells the compiler that iterations are independent from each other. This is a better solution compared with the new ‘const’ clause solution as the code still follows the OpenACC standard and the RODC optimization is used by the compiler automatically. Because this optimization is specific to the Nvidia Kepler architecture, the compiler will bypass when targeting another architecture. This optimization has been implemented in OpenUH, and we evaluate it for seven NAS benchmarks.

7. EVALUATION

The experimental platform we used has an Intel Xeon E5-2640 processor, 32 GB main memory, and an Nvidia Kepler GPU card (K20c) with 5 GB global memory. For a comparative analysis, we used commercial OpenACC compilers from CAPS (3.4.3) and PGI (14.3). CUDA 5.5 is used for all the three compilers. GCC 4.4.7 is used as the back-end host compiler for the CAPS source-to-source compiler. We disable fused multiply add [16] so that we could make a fair CPU and GPU results comparison. 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 uses ‘-fopenacc -nvcc,-arch=sm_35,-fmad=false -Wb,-rodc’ flag to compile the given OpenACC program. The flags following ‘-nvcc’ are passed to the CUDA 5.5 compiler. The flag ‘-Wb,-rodc’ enables RODC in OpenUH. To obtain reliable results, we took an average of 10 execution runs for every benchmark.

There are no case studies that could cover all reduction cases; hence, we designed and implemented a test suite that validates all possible cases of reduction, including different reduction data types and operations. This helped evaluate our reduction operations. The test suite checks for a pass or a fail by verifying an OpenACC result with that of a CPU result. If the values do not match, it implies there is an implementation issue. Time for each of the reduction cases is also measured; in the event that the compilers under evaluation pass a test, we compare their performances as well. When reduction occurs in one of the levels of parallelism, the other levels of parallelism contain instructions being executed in parallel.

Only the reduction across multi-level thread parallelism (RMP) in the same loop uses one loop; the other reduction tests use triple-nested loop. When one loop level needs to perform a 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 the `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.

Table IV depicts the performance gain with OpenUH over PGI and CAPS compilers using the reduction test suite. OpenUH compiler can pass all the test cases [14]. The CAPS compiler failed gang reduction and all tests of RMP in either different loops or the same loop. The PGI compiler failed the sum reduction in worker, vector, and RMP in gang & worker & vector. It also failed to compile the RMP in gang & worker. It is observed that in gang or vector reduction, the performance achieved with OpenUH is similar to performance with the CAPS compiler, and only in worker reduction is it slightly less efficient than the CAPS compiler. The performance with OpenUH is better than with the PGI compiler for all reduction cases. Because we do not have access to implementation details of commercial compilers, we could not delve deeper. We determine that two features of our reduction algorithm helped to improve the performance: (1) using high bandwidth and low latency shared memory exploits the full power of parallelism during local reduction in the gang; and (2) sequential addressing, loop unrolling, and algorithm cascading [15] are applied during the OpenACC optimization phases in order to improve the instruction-level parallelism.

We use the best hand-tuned CUDA code to analyze the results for OpenUH. We chose the same problem size for the analysis and considered seven NAS benchmarks. They are Embarrassingly

DOI: 10.1002/cpe

Table IV. Using Micro-reduction test suite, performance comparison of OpenUH OpenACC compilers with PGI and CAPS: speedup = $\frac{ExeTime_{OtherCompiler}}{ExeTime_{OpenUH}}$.

		Data type			
		Int		Double	
Reduction position	Reduction operator	Speedup over PGI	Speedup over CAPS	Speedup over PGI	Speedup over CAPS CAPS
gang	+	3.11	F	1.86	F
	*	2.81	F	1.96	F
worker	+	F	0.71	F	0.83
	*	1.85	0.71	1.63	0.84
vector	+	F	0.94	F	0.96
	*	1.97	0.94	1.53	0.96
gang worker	+	CE	F	CE	F
	*	CE	F	CE	F
worker vector	+	4.74	F	3.29	F
	*	5.24	0.99	3.00	0.98
gang worker vector	+	F	F	F	F
	*	28.35	F	28.29	F
same line gang worker vector	+	37.08	F	23.84	F
	*	33.53	F	25.75	F

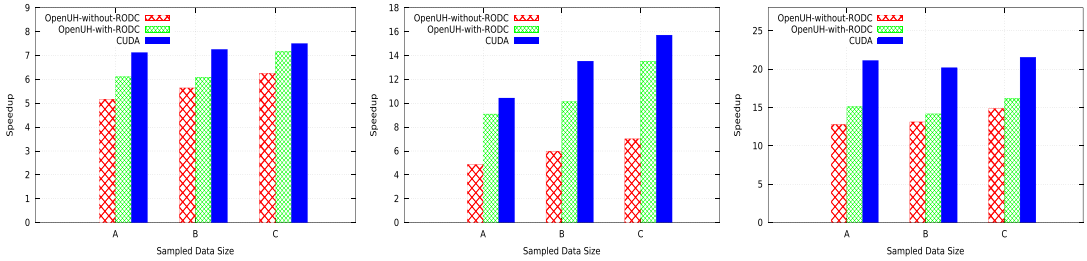
'F' stands for test FAILED, and 'CE' stands for compile-time error. The 'same line gang worker vector' row uses only one loop; the other reduction tests used triple-nested loop where the outermost, middle, and the innermost loops are gang, worker, and vector, respectively. The first column implies the reduction position. For instance, 'gang worker' means gang and worker loops need to do reduction while the vector loop executes in parallel. Speedup larger than 1.0 implies efficient code generation by OpenUH.

Parallel (EP), Conjugate Gradient (CG), MultiGrid (MG), Scalar Pentadiagonal (SP), Lower-Upper symmetric Gauss–Seidel (LU), Block Tridiagonal (BT), and Fast Fourier Transform (FT). We have contributed EP, CG, BT, and SP benchmarks to SPEC ACCEL V1.0 [17]. We chose NPB for evaluation purposes because these codes are large and complex enough to simulate the behavior of real applications. We found CUDA versions [9] for only BT, LU, and SP.

Our comparative analysis is organized into two parts. First up, we compare BT, LU, and SP OpenACC performance with hand-tuned CUDA versions. Secondly, we compare results using OpenUH and PGI for EP, CG, MG, and FT. The CAPS compiler generates incorrect results for most of the cases, so we did not consider results from CAPS for this analysis. The PGI compiler can successfully compile BT, LU, and SP benchmarks; however, they crashed during program execution, so we did not compare them with OpenUH. In NAS OpenACC benchmarks, we removed all the loop scheduling clauses and let the PGI compiler auto-tune the best loop scheduling for each kernel and parallel region. For OpenUH, we manually specified the loop scheduling for each kernel and parallel region. OpenUH generates two versions of results: with and without RODC optimization.

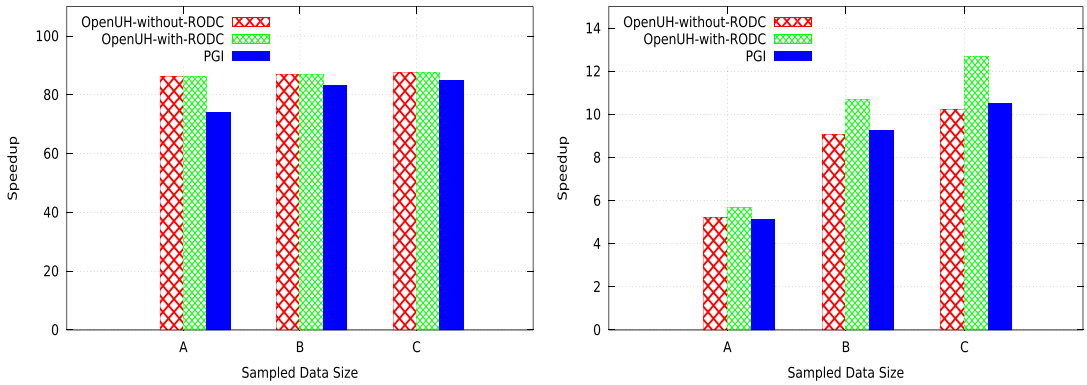
In Figures 16–18, the usage of RODC shows improvement in performance, especially for BT, LU, CG, and MG. A common feature of these applications is that all of them use a large number of read-only buffers/arrays and thus benefit from using the RODC optimization. Figure 16(a), (b), and (c) shows the comparison of speedup using OpenUH and CUDA. The structure of the CUDA code is very different from that of the OpenACC structure, because the latter uses a sequential version of the code as the starting point. However, OpenACC still yields competitive performance compared with the tuned CUDA code. The performance range mentioned later corresponds to the performance obtained for different classes (Classes A through C of NPB). When the BT benchmark uses the RODC optimization, OpenACC reaches 85~96% of performance over CUDA. Without using RODC optimization, BT achieved only 72~83% of performance over CUDA. For LU, with RODC optimization, 72~87% of performance over CUDA was achieved. For SP case, the RODC does not help much with performance improvement, but it still reaches 70~75% of performance over CUDA.

DOI: 10.1002/cpe



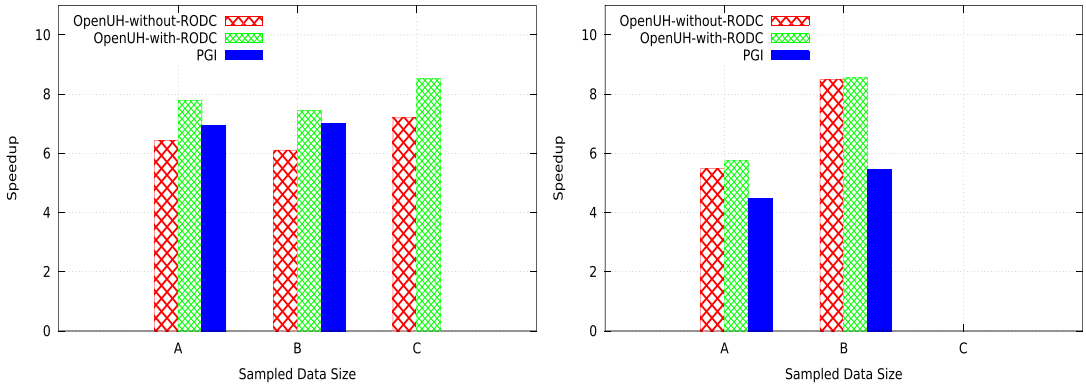
(a) BT (b) LU (c) SP

Figure 16. Performance comparison of OpenUH and CUDA: speedup = $\frac{ExeTime_{Sequential}}{ExeTime_{Parallel}}$. A, B, and C are problem size: A < B < C. (a) Block Tridiagonal (BT); (b) Lower-Upper symmetric Gauss–Seidel (LU); (c) Scalar Pentadiagonal (SP). RODC, read-only data cache.



(a) EP (b) CG

Figure 17. Performance Comparison of OpenACC applications using OpenUH and PGI compiler: speedup = $\frac{ExeTime_{Sequential}}{ExeTime_{Parallel}}$. A, B, and C are problem size: A < B < C. (a) Embarrassingly Parallel (EP); (b) Conjugate Gradient (CG). RODC, read-only data cache.



(a) MG (b) FT

Figure 18. Performance comparison of OpenACC applications using OpenUH and PGI compiler: speedup = $\frac{ExeTime_{Sequential}}{ExeTime_{Parallel}}$. A, B, and C are problem size: A < B < C. (a) PGI failed in MultiGrid (MG) CLASS C due to out of memory in device and OpenUH succeeds. (b) Both OpenUH and PGI failed in Fast Fourier Transform (FT) CLASS C due to out of memory in device. RODC, read-only data cache.

DOI: 10.1002/cpe

If the OpenACC code structure of SP and LU followed that of the CUDA code, performance of SP and LU can still be improved. However, this will involve a deep understanding of the algorithms and extensive rewriting of the codes. This will result in sacrificing portability to a large extent.

Figures 17 and 18 show the speedup of the OpenUH and PGI OpenACC compilers for EP, CG, MG, and FT benchmarks. Figure 17(a) shows that the speedup of EP is not affected with or without RODC optimization. This is because EP is compute-bound and uses very little read-only data. Also, OpenUH performs better than PGI. Figure 17(b) shows that OpenUH performs much better than PGI with RODC optimization. However, without RODC, PGI performs slightly better than OpenUH. Figure 18(a) shows that MG is memory-bound, as a result of larger data sizes, that is, Class C; PGI failed the execution at runtime. However, OpenUH was able to compile and run successfully. The potential reason could be that in OpenUH, data once created resides in the GPU and can be reused. Therefore, the data need not be re-created again, even if an access to a sub-array is made. Figure 18(b) shows the result for FT. FT has a loop distributed across vector threads and uses reduction. PGI shows poorer results because the parallel region is executed sequentially because of the failed vector reduction. This scenario is consistent with a similar test case that we wrote as part of our reduction microbenchmark suite. In this case, the data are transferred from GPU to CPU before sequential execution, and it is transferred back to the GPU after sequential execution. This workaround introduces a redundant data transfer. As a result, PGI does not perform well at all compared with OpenUH. It indicates that the reduction operation not only has an impact on parallelism but also helps remove redundant data movement. Note that both PGI and OpenUH ran out of memory for Class C.

8. RELATED WORK

In this section, we will discuss some of the existing implementations of OpenACC compilers.

There are both commercial academic OpenACC compiler efforts to support high-level programming models for GPGPUs. The CAPS compiler [18] uses the source-to-source translation approach for both CPU and GPU. The PGI OpenACC accelerator compiler [19] uses a binary code generation approach. The nested gang and vector loop scheduling we implemented in OpenUH are also supported in the PGI compiler. PGI implements aggressive implicit optimization for offload kernels region and overrides user decisions, resulting in less user control over optimizations. The Cray compiler [13] is another OpenACC compiler that can only be used on Cray platforms.

In the academic arena, OpenARC [20] from Oak Ridge National Labs is another effort toward developing on open-source OpenACC compiler. OpenARC is based on the Cetus source-to-source framework. However, it has not been released yet.

OpenMP [5] is another directive-based programming standard and provides accelerator support in 4.0 version, which was released in 2013. Liao [21] verified a prototype of OpenMP using the ROSE compiler, called Heterogeneous OpenMP. Heterogeneous OpenMP is still not mature yet to handle the NAS benchmarks. We will leave the performance evaluation as future work.

Other directive-based approaches for GPGPUs include HiCUDA [22], HMPP [12], and the PGI Accelerator Model [23]. HiCUDA is a low-level directive-based programming model. Because HiCUDA only targets NVIDIA GPGPU, it provides a set of directives for users to manually control the data cache, loop scheduling, and data movement between CPU and GPU. It is fully the programmers' responsibility to control everything, which makes it still difficult to program. In addition to its OpenACC implementation, CAPS also provides another directive-based programming model called HMPP. HMPP and PGI accelerator models are similar in that the compiler can help users auto-analyze and auto-tune the annotated computation kernels. Users also can use directives to choose their own optimization decisions. However, none of them are standard, and therefore, their code is not as portable as OpenACC.

9. CONCLUSION

In this paper, we presented a robust OpenACC implementation in the OpenUH compiler. We describe our compiler framework, which provides a rich set of loop scheduling strategies, and com-

DOI: 10.1002/cpe

prehensive solutions for reduction operations. We also discuss the read-only data cache optimization for the Kepler architecture. The loop scheduling strategies and efficient reduction algorithms we developed help to generate high-quality CUDA source code with OpenUH. We also deliver compile-time detection mechanisms to recognize read-only array/buffer use in the application, taking advantage of the read-only data cache in the hardware for OpenACC parallel and kernels regions.

We used the NPB applications to evaluate our design and implementation. A number of accomplishments were presented. First, we provided compiler support for read-only data cache optimization, which dramatically improved cases when there was a large number of read-only data/buffer in the offloaded compute regions. Second, our compiler support for a directive-based model, OpenACC, not only helped maintain source code consistency but also achieved performance close to that of a well-tuned CUDA code for NPB applications (BT, SP, and LU). Third, our implementation of reduction algorithms provided robust and efficient results compared with other vendor compilers. Evaluation for these solutions demonstrated that our compiler could yield competitive performance compared with that of some of the existing vendor compilers.

As part of the future work, we will extend the back-end support for accelerators other than GPUs, such as Intel Xeon Phi, AMD APUs, and so on. We will also take advantage of aggressive loop nest optimizations that OpenUH offers to generate an even more optimized accelerator code.

ACKNOWLEDGEMENTS

We are grateful to NVIDIA for providing us with the necessary hardware. We are thankful to PGI and CAPS for providing a copy of their compilers and necessary technical support.

REFERENCES

1. CUDA, 2013. (Available from: http://www.nvidia.com/object/cuda_home_new.html.) [Accessed on 2 April 2014].
2. OpenCL Standard, 2013. (Available from: http://www.khronos.org/opencl.) [Accessed on 2 November 2013].
3. Dolbeau R, Bihan S, Bodin F. HMPP: a hybrid multi-core parallel programming environment. *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU'07)*, Boston, MA, 2007.
4. OpenACC, 2013. (Available from: http://www.openacc-standard.org.) [Accessed on 17 March 2014].
5. OpenMP, 2013. (Available from: http://www.openmp.org.) [Accessed on 20 November 2013].
6. Liao C, Hernandez O, Chapman B, Chen W, Zheng W. OpenUH: an optimizing, portable OpenMP compiler. *Concurrency and Computation: Practice and Experience* 2007; **19**(18):2317–2332.
7. Tian X, Xu R, Yan Y, Yun Z, Chandrasekaran S, Chapman B. Compiling a high-level directive-based programming model for accelerators. *LCPC 2013: The 26th International Workshop on Languages and Compilers for Parallel Computing*, San Jose, CA, USA, 2013; 105–120.
8. Bailey DH, Barszcz E, Barton JT, Browning DS, Carter RL, Dagum L, Fatoohi RA, Frederickson PO, Lasinski TA, Schreiber RS, Simon HD, Venkatakrishnan V, Weeratunga SK. The NAS parallel benchmarks. *International Journal of High Performance Computing Applications* 1991; **5**(3):63–73.
9. NPB CUDA benchmarks, 2013. (Available from: http://www.tu-chemnitz.de/informatik/PI/forschung/download/npb-gpu/index.php.en.) [Accessed on 14 February 2014].
10. Leback B, Wolfe M, Miles D. The PGI Fortran and C99 OpenACC compilers. *Cray User Group* 2012.
11. CUDA C PROGRAMMING GUIDE, 2013. (Available from: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.) [Accessed on 19 August 2013].
12. CAPS OpenACC Parallism Mapping, 2013. (Available from: http://kb.caps-entreprise.com/what-gang-workers-and-threads-correspond-to-on-a-cuda-card.) [Accessed on 16 June 2013].
13. Cray C. C++ reference manual, 2003.
14. Xu R, Tian X, Yan Y, Chandrasekaran S, Chapman B. Reduction operations in parallel loops for GPGPUs. *Proceedings of Programming Models and Applications on Multicores and Manycores*, ACM, Orlando, Florida, 2014; 10.
15. Harris M. Optimizing parallel reduction in CUDA. *NVIDIA Developer Technology* 2007; **6**.
16. Whitehead N, Fit-Florea A. Precision & performance: floating point and IEEE 754 compliance for NVIDIA GPUs. *nVidia Technical White Paper* 2011.
17. SPEC ACCEL, 2014. (Available from: http://www.spec.org/auto/accel/Docs.) [Accessed on 2 April 2014].
18. CAPS Enterprise OpenACC Compiler Reference Manual, 2013. (Available from: http://www.openacc.org/sites/default/files/HMPPOpenACC-3.2_ReferenceManual.pdf.) [Accessed on 21 October 2013].
19. PGI Accelerator Compilers, 2013. (Available from: http://www.pgroup.com/resources/accel.htm.) [Accessed on 22 October 2013].

DOI: 10.1002/cpe

20. OpenARC:Open Accelerator Research Compiler, 2013. (Available from: http://ft.ornl.gov/research/openarc.) [Accessed on 2 April 2014].
21. Liao C, Yan Y, de Supinski BR, Quinlan DJ, Chapman B. Early experiences with the OpenMP accelerator model. In *OpenMP in the Era of Low Power Devices and Accelerators*. Springer: Canberra, AUSTRALIA, 2013; 84–98.
22. Han TD, Abdelrahman TS. hiCUDA: high-level GPGPU programming. *IEEE Transactions on Parallel and Distributed Systems* 2011; **22**(1):78–90.
23. Wolfe M. Implementing the PGI accelerator model. *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, *(GPGPU '10)*, ACM, New York, NY, USA, 2010; 43–50.

DOI: 10.1002/cpe