2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing

CUDA vs OpenACC: Performance Case Studies with Kernel Benchmarks and a Memory-Bound CFD Application

Tetsuya Hoshino*, Naoya Maruyama†, Satoshi Matsuoka‡, and, Ryoji Takaki§
*Department of Mathematical and Computing Sciences, Tokyo Institute of Technology, Tokyo, Japan
†RIKEN Advanced Institute for Computational Science, JST CREST, Kobe, Japan
‡Global Scientific Information and Computing Center, Tokyo Institute of Technology, Tokyo, Japan
§Institute of Space and Astronautical Science, Japan Aerospace Exploration Agency, Tokyo, Japan

***Abstract*—OpenACC is a new accelerator programming interface that provides a set of OpenMP-like loop directives for the programming of accelerators in an implicit and portable way. It allows the programmer to express the offloading of data and computations to accelerators, such that the porting process for legacy CPU-based applications can be significantly simplified. This paper focuses on the performance aspects of OpenACC using two microbenchmarks and one real-world computational fluid dynamics application. Both evaluations show that in general OpenACC performance is approximately 50% lower than CUDA. However, for some applications it can reach up to 98% with careful manual optimizations. The results also indicate several limitations of the OpenACC specification that hamper full use of the GPU hardware resources, resulting in a significant performance gap when compared to a highly tuned CUDA code. The lack of a programming interface for the shared memory in particular results in as much as three times lower performance.**

***Keywords*-GPU, OpenACC, CUDA**

I. Introduction

Parallel programming with compiler directives, such as OpenMP, is widely used in scientific computing, where parallelism often appears in regular repetition constructs such as Fortran DO loops. OpenACC is a new specification for compiler directives that allow for annotation of the compute and data regions that are offloaded to accelerators such as graphics processing units (GPUs) [1]. In contrast to current mainstream GPU programming, such as CUDA [2] and OpenCL [3], where more explicit compute and data management is necessary, porting of legacy CPU-based applications with OpenACC requires only code annotations without any significant structural changes in the original code, which allows considerable simplification and productivity improvement when hybridizing existing applications.

Programming with OpenACC directives, while greatly simplified, is not as flexible as using CUDA or OpenCL. For example, both CUDA and OpenCL provide fine-grained synchronization primitives, such as thread synchronization and atomic operations, whereas OpenACC does not. Efficient implementations of applications may depend on the availability of software-managed on-chip memory, which can be used directly in CUDA and OpenCL, but not in OpenACC. These differences may prevent full use of the available architectural resources, potentially resulting in greatly inferior performance when compared to highly tuned CUDA and OpenCL code.

To understand the performance implications of programming accelerators with OpenACC, this paper presents case studies of porting and optimization of kernel benchmarks and a real-world application code. As kernel benchmarks, we use matrix multiplications and a 3-D stencil and compare their performances on NVIDIA GPUs using both OpenACC and CUDA. As an application case study, we use a computational fluid dynamics (CFD) application, called Unified Platform for Aerospace Computational Simulation (UPACS) [4]. The code was originally written in Fortran and was parallelized with MPI, so we first identify the bottleneck loops and then rewrite part of the application for GPU execution in both CUDA and OpenACC. Also, because the performance of UPACS is usually bound by the memory throughput, we apply a series of typical memory access optimizations, including several blocking transformations and loop fusions. Throughout this optimization study, we attempt to implement each optimization in both CUDA and OpenACC and compare their applicability and effectiveness in a fair setting.

Experimental results measured using NVIDIA GPUs with multiple production OpenACC compilers show that in the microbenchmark studies, the performances of the OpenACC versions are on average approximately half of the corresponding CUDA versions when the same manual optimizations are applied, with performance reaching up to 98% depending on the compiler. A similar trend in average performance is also observed for the UPACS application, but the OpenACC performance relative to CUDA reaches only 64% at best. We also find that the highly tuned versions have a larger performance gap, as some of the optimizations, particularly those based on shared memory, cannot be applied because of the limitations of the programming interface of the current OpenACC specification. Specifically, the highly tuned CUDA versions of matrix multiplication, a 7-point stencil, and UPACS are faster than the best performing OpenACC results by factors of 2.7, 1.2, and 2.4, respectively.

978-0-7695-4996-5/13 $26.00 © 2013 IEEE
DOI 10.1109/CCGrid.2013.12

II. GPU Programming Interfaces

Programming the GPU requires the use of language extensions since no standard programming language natively supports GPU programming. CUDA is currently one of the most widely used extensions, although it is specific to NVIDIA GPUs. It provides explicit, flexible programming constructs that allow fine-grained control for GPU computing. OpenACC is a relatively new extension for C and Fortran and defines directives for offloading of computation and data to accelerators. This section gives a brief overview of OpenACC and its comparison with CUDA.

A. OpenACC

OpenACC provides OpenMP-like directives to define compute and data regions in standard C and Fortran programs. Compute regions can be defined with either `parallel` or `kernels` directives. The code inside a compute region is shipped to and executed on an accelerator. C and Fortran programs with OpenACC directives can be compiled into a hybrid code by OpenACC-compliant compilers. Similar to OpenMP, in which the directives can be safely ignored, non-supporting compilers can still generate correct CPU-only code without interpretation of the directives. The directives that are used in our work are described below, along with some of their options.

Similar to CUDA, a hybrid parallelism of SIMD and SPMD is used in OpenACC. The code inside the compute region is executed in parallel by multiple *workers*, each of which can also have SIMD operations. Similar to the thread blocks in CUDA, a group of workers is managed as a *gang* in OpenACC. The numbers of workers and gangs as well as the length of the SIMD operations can be configured through the `loop` directive and its options.

Parallel loops can also have `cache` directives, which specify that data objects should be kept on the highest level of the cache memory hierarchy for the body of the loop. Because OpenACC does not have synchronization primitives for the parallelized loop iterations, such as `barrier` in OpenMP, the directive can only be used for read-only data.

The host and accelerator memory spaces are assumed to be separate, and the data objects used in the compute regions must be resident on the accelerator memory. However, unlike CUDA, they are automatically copied between the host and accelerator memories as necessary by the compilers such that their coherency is obtained at the region boundaries. Explicit control of the data transfer is also possible with directive options such as `copyin` and `present`.

B. Comparison of OpenACC and CUDA

Productivity: CUDA has been the most widely used programming interface for scientific computing on GPUs, but its explicit, low-level programming abstraction results in relatively low programming productivity. Porting of existing CPU-based programs to CUDA requires identification of the bottleneck regions, which must then be rewritten into CUDA kernel code, which often causes significant structural changes in the original code. In contrast, although OpenACC still requires the programmer to identify the bottleneck regions, the original code can be reused with much fewer changes than required with CUDA, because the minimum porting requirement is annotation of the regions with the OpenACC directives. This simplification is particularly important when porting large legacy applications.

Portability and Performance: Unlike CUDA, OpenACC is designed to be portable across devices from multiple vendors. However, this design decision limits the use of vendor-specific architectural features. For example, OpenACC does not have software-addressable on-chip memory, which is available as shared memory in CUDA. The `cache` directive may be used to fetch data to the shared memory, but the lack of synchronization with respect to the shared memory effectively limits the use of this directive to read-only data. This limitation prohibits some manual code transformations for memory access optimizations, such as temporal blocking using the shared memory [5], leaving them completely to the compiler.

III. Evaluation Methodology

The goal of this paper is to understand the performance of OpenACC-based hybrid codes on GPU accelerators. To that end, we develop two OpenACC-based kernel benchmarks, a matrix multiplication and a 7-point stencil, and compare them with the CUDA-based versions. We choose these two kernels to study the performance implications of both compute and memory intensive code, respectively. We also study an application performance by extending UPACS with both OpenACC and CUDA.

For each of the benchmarks and application, starting from a CPU-based reference code, we incrementally develop multiple versions that run on the GPU with varying degrees of optimizations. The *baseline* version has minimum extensions to enable it to run on the GPU. The rest of this section describes each of the benchmarks and their baseline implementations in OpenACC and CUDA, followed by Section IV, which describes the optimized versions.

A. Kernel Benchmarks

1) Matrix Multiplication: We use a double-precision multiplication of $C = A \times B$, where each dimension is n. The OpenACC baseline implementation is shown in Figure 1. A `kernels` directive is used in this implementation. We assume that all arrays have already been made available on the GPU device memory by a data region directive that is not shown here. The routine is parallelized by using the two `loop` directives associated with the `i` and `j` loops.

Our CUDA implementation of the matrix multiplication kernel is very straightforward. We map multiple threads and thread blocks to the `i` and `j` loops and let each thread

```
! a, b, c: 2-D n*n matrices of double values
!$acc kernels present(a, b, c)
!$acc loop
do j = 1, n
!$acc loop
  do i = 1, n
    cc = 0
    do k = 1, n
      cc = cc + a(i,k) * b(k,j)
    end do
    c(i,j) = cc
  end do
end do
!$acc end kernels
```

Figure 1. OpenACC baseline version of matrix multiplication.

```
// f1, f2: 3-D nx*ny*nz arrays of float values
!$acc kernels present(f1, f2)
!$acc loop
do z = 1, nz
!$acc loop
  do y = 1, ny
!$acc loop
    do x = 1, nx
      w = -1; e = 1; n = -1;
      s = 1; b = -1; t = 1;
      if(x == 1)  w = 0
      if(x == nx) e = 0
      if(y == 1)  n = 0
      if(y == ny) s = 0
      if(z == 1)  b = 0
      if(z == nz) t = 0
      f2(x,y,z) = cc*f1(x,y,z) + cw*f1(x+w,y,z)
        + ce*f1(x+e,y,z) + cs*f1(x,y+s,z)
        + cn*f1(x,y+n,z) + cb * f1(x,y,z+b)
        + ct*f1(x,y,z+t)
    end do
  end do
end do
!$acc end kernels
```

Figure 2. OpenACC baseline version of 7-point stencil.

compute the innermost `k` loop. We selected a thread block size of 16×16.

2) 7-point Stencil: We use a single-precision 7-point stencil kernel with the Dirichlet boundary condition. The OpenACC baseline implementation is illustrated in Figure 2. As we did in the matrix multiplication, we again assume that all necessary data exist on the GPU memory, having been transferred by a data region directive that is not shown here. We annotate all the three loops with the `loop` directive.

Our CUDA implementation of the stencil divides the `x` and `y` loops by all of the threads across the thread blocks, but lets each thread compute the `z` loop sequentially. We use a thread block size of 64×4, which typically performs efficiently in stencil kernels on NVIDIA GPUs.

B. Application Case Study

We use the CFD software package, called UPACS, which has been under development at the Japan Aerospace Exploration Agency since late 1990's [4]. UPACS consists of nearly one hundred thousand lines of Fortran 90 code, providing a large number of CFD solvers and their supporting components. It uses a multi-block and overset structured grid system with an underlying data structure that is a collection of loosely-connected rectangular 3-D grids. The size of each grid is typically 60^3, but can be chosen adaptively depending on the simulation settings and the execution environment.

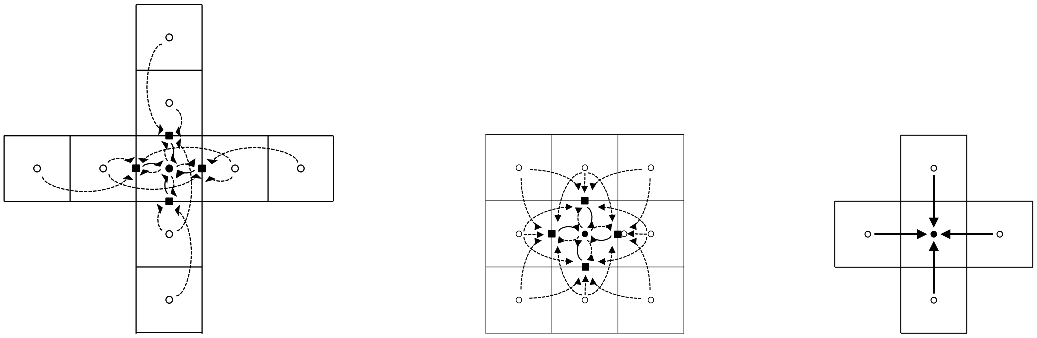
In the original UPACS, a block-wise parallelization scheme is implemented with MPI, where each MPI process sequentially processes the assigned blocks with optional compiler-based automatic parallelization of the nested DO loops inside the blocks.

In this application case study, we selected a flux-based Navier-Stokes solver, which is one of the most important components in UPACS, and consists of three major computation phases: *convection*, *viscosity*, and *time integration*. Each phase performs a standard stencil computation for each 3-D block, as illustrated in Figure 3. The convection and viscosity phases have no loop carried dependencies, which allows all of the grid points to be computed in parallel, whereas the time integration phase has a diagonal data dependency, which requires wavefront parallelization. As shown in Table I, the code size of each phase is approximately six hundred lines, consisting of up to 10 Fortran subroutines and 7 loop nests. Each phase updates 5 data objects using 20 to 33 read-only objects and 5 to 25 temporary objects. Note that each data object is a double-precision 3-D array or multiples of them aggregated to an array of structures.

The three phases are iteratively executed a given number of times, which is the most time-consuming part of the solver. For example, a sequential run with 120^3 blocks using a recent Intel Xeon processor spends approximately 25% of the total time in the convection phase, 37% in the viscosity phase, and 28% in the time integration phase; as a result, 90% of the total time is consumed by just these three phases. Therefore, in this case study comparing CUDA and OpenACC, we only port the three phases to the GPU without modifying the remaining part. Details of the baseline versions are as follows.

1) OpenACC Version: Unlike the porting of microbenchmarks, where addition of the OpenACC directives to the CPU code is the only necessary change, we found that the original UPACS code uses Fortran features that are not currently supported by any of the OpenACC compilers used in this work. The development of the baseline version therefore consists of the following two steps.

As a first step, we replace any use of non-supported features in the original code with an equivalent representation. We found that two features that are both related to Fortran derived data types are not supported: arrays of derived data types and derived data types with variable-length arrays, both of which are used extensively in UPACS. More specifically, all data defined at cell faces, such as fluxes, were originally expressed as a single object of a derived data type. We change the variable declarations to a set of separate 3-D arrays in the OpenACC version. Also,



(a) Convection stencil

(b) Viscosity stencil

(c) Time integration stencil

Figure 3. UPACS computation phases, illustrated in 2-D for simplicity. Each filled circle or rectangle represents a grid point or cell face updated by the stencil. Arrows are used to express read-write data dependencies from source to destination.

Table I
DETAILS OF EACH MAJOR PHASE IN UPACS.

				Number of 3-D data		
Phase	LoC	Number of subroutines	Number of loop nests	Read	Updated	Temporary
Convection	682	10	7	29	5	15
Viscosity	599	5	5	33	5	25
Time integration	622	5	5	20	5	5

because each block originally had a derived-type object with variable-length arrays that represent the data defined at the cell center, we copy each array out of the object into a separate array with the same length but defined independently. There are eleven such arrays in the original code, and we copy all of them to separate arrays and change their references as appropriately. In addition to these changes, we found that one subroutine did not yield correct results with any of the used compilers. It defines a stencil on each of the six boundary planes of a 3-D grid with a different stencil shape parameterized with a subroutine argument. We replace it with six unique subroutines defined for each of the six boundary planes with the non-parameterized, equivalent stencil.

Once the removal of unsupported features is complete, we offload the three phases to the GPU by defining the OpenACC compute and data regions as we did in the stencil case study. To offload computation, we add a `kernels` directive to each stencil loop nest that does not have loop carried dependencies and annotate each loop with a `loop` directive. For the time integration phase, which has a loop-carried diagonal dependency, we first change the loop nest so that the innermost loop calculates the stencil of a wavefront, and annotate the loop with a `loop` directive to parallelize it.

To save data transfers between the host and GPU memories as much as possible, we define a data region that encompasses the main loop of the three phases. We use the `copyin` option for each array variable that is used in the loop but not after the loop, and the `copy` option for variables that need the latest copy after the loop. Also, for each variable that required MPI communication for boundary exchange, we transfer the whole array before and after each exchange by using the `update` directive. Although this method is in fact suboptimal, because only the thinner subregion of the boundary and halo requires the host-GPU data transfer, we transfer the entire array for simplicity in this work. In UPACS, there are five such 3-D data: the density, the momentum for each dimension, and the energy.

2) CUDA Version: In the baseline CUDA version, we create a new CUDA kernel for each of the loop nests with no data dependency by following the same parallelization scheme as used in the 7-point stencil benchmark. For the loop nest with a loop carried dependency in the time integration phase, we again use the wavefront parallelization scheme as per the OpenACC version. We use the thread block size of 64×4, which is the same as that for the 7-point stencil, for all kernels in the baseline version. Also, to make the performance comparison with OpenACC consistent, we change the array of structure data to a structure of array form in the baseline CUDA version as well.

Although CUDA does not allow direct reuse of the original loop nest, our baseline kernel implementation is relatively simple because it does not have the limitations of OpenACC. Instead, the separation of the host and GPU memory spaces has a greater impact on the porting cost, because all variables used in the loop nests must be identified and must have their copies explicitly transferred to the GPU memory. It also effectively doubles the number of variable declarations, since CUDA requires a separate variable for each of the host and GPU memories, whereas in OpenACC, the same variable can be used transparently in both memories. We will compare the code sizes of the OpenACC and CUDA implementations in Section V.

Note that in this case study, we develop the kernel functions and related host control programs in CUDA C and used them from the original Fortran code. We see that the same implementation strategy can also be used with CUDA Fortran.

IV. Optimizations

For both CUDA and OpenACC, we apply a series of manual optimizations to evaluate their effectiveness and programming costs. This section describes both the CUDA and OpenACC versions with manual optimizations, while the next section assesses their effectiveness by comparing them against the baseline versions.

A. Matrix Multiplication

1) Thread Mapping: Our baseline OpenACC versions do not explicitly control the mapping of the available hardware parallelism to the application-level parallelism; instead, they only have a `loop` directive on each parallel loop to instruct the compilers to parallelize the loop. Since the `loop` directive also allows explicit mapping through its options, we create a modified version with manually-tuned thread mapping in OpenACC. The tuning process involves the selection of parallelism and its degree, which is performed by using a semi-automated method that searches exhaustively for the most efficient mapping among a predefined set. We apply this optimization to both OpenACC and CUDA. Note that since CUDA always requires a mapping specification, a specification chosen on the basis of known heuristics is used in the baseline CUDA version.

2) Shared Memory Blocking: We optimize the baseline CUDA version by using the shared memory, as described in [6]. Specifically, in the optimized version, all threads in the same thread block cooperatively load input sub matrices into the shared memory by all threads in the same thread block and use the on-chip data in the loop to compute the inner products. This scheme reduces the number of global memory accesses, if no hardware caching is performed, by a factor of T_x for the multiplicand matrix and a factor of T_y for the multiplier, where T_x and T_y are the dimensions of the 2-D thread block used in our benchmark implementation. Although our code is not fully optimized as the DGEMM routine in the CUBLAS library, we use the performance number as a reference to evaluating the optimization effect.

The same optimization cannot be expressed directly in OpenACC since it does not have programming constructs to access the shared memory. One indirect way would be to use the `cache` directive, which can be used to request data objects to be cached on the shared memory. Among the compilers used in this work, only the latest PGI compiler (version 12.10) claims to fully support the directive. We were, however, unsuccessful in using the directive because of compilation errors, and thus our optimization evaluation for the matrix multiplication only includes the CUDA case.

B. 7-Point Stencil

1) Thread Mapping: As in the matrix multiplication, we evaluate the effectiveness of thread mapping optimization.

2) Branch Hoisting: As shown in Figure 2, the stencil loop has six branches inside the innermost loop. Since the branches using variables y and z are loop invariant, we simply move them outside of the loop body. To completely eliminate the branches from the innermost loop, we move the remaining loops by peeling the first and last iterations of the inner loop.

3) Register Blocking: Since the slowest changing dimension is updated by a single thread and there is a data reuse along the dimension, we can use three local variables to hold the elements of the input grid at coordinates $(x, y, z-1)$, (x, y, z), and $(x, y, z+1)$. We implement this optimization in both OpenACC and CUDA.

C. UPACS

In UPACS, we also apply the thread mapping and register blocking optimizations to both OpenACC and CUDA. We also evaluate the following code transformations that are not applicable to the microbenchmarks.

1) Kernel Specialization: A common pattern in the original UPACS code defines a stencil for each of the three dimensions with a similar computation pattern. The original UPACS has a common stencil subroutine for these stencils, and indirectly accesses neighbor elements by using a subroutine parameter that specifies the stencil offsets. While this coding practice is desirable from a software engineering perspective, we found that it increased the register pressure in both the OpenACC and the CUDA code. Our kernel specialization optimizes each stencil subroutine with this pattern by creating a separate subroutine for each dimension.

2) Loop Fusion: Because stencils are typically memory bound, minimization of memory accesses is often a effective optimization method. As shown in Table I, there are multiple loop nests in each UPACS phase; therein, some pairs have a producer-consumer data flow with temporary 3-D array variables. To reduce the memory access pressure, we fuse these loop pairs and allocate their temporary variables on registers or shared memory, depending on the existence of an inter-thread data dependency. In CUDA, we manually apply this code transformation to six loop pairs in the convection phase. Since OpenACC does not have inter-thread data communication methods, such as the shared memory in CUDA, only two of the loops are fused in OpenACC.

3) Fine-Grained Parallelization in the Matrix Free Gauss-Seidel Method: The main bottleneck routine in the time integration phase uses a Matrix Free Gauss-Seidel method (MFGS), where each point uses six neighbor points as illustrated in Figure 3c. The MFGS method has a computation defined at each of the neighbor points, which is computed serially in the original UPACS code as well as in our baseline versions. The neighbor results are then used to update the central point. From our code inspection, we speculate that the computations for the six neighbor points are actually expensive enough to compute in parallel by allocating one thread per neighbor point. This fine-grained parallelization requires inter-thread data communications to update the central point, which is possible in CUDA by using the shared memory, but not in OpenACC.

V. Performance Evaluation

In this section, we give performance results using a single GPU with three OpenACC production compilers, including the PGI, HMPP, and Cray compilers. Although UPACS is parallelized with MPI, and thus is able to run on multiple GPUs over distributed nodes, we use only a single GPU for comparing CUDA and OpenACC performances since inter-GPU communications should have the same effect on the overall performance of each version. We also present the number of modified or inserted lines of code in each version as a rough approximation of the programming costs.

We use a single node of the TSUBAME2.0 supercomputer, which consists of compute nodes with two Intel Xeon Westmere-EP 2.9 GHz CPUs and three NVIDIA M2050 GPUs. We use PGI CUDA Fortran 12.10 and Intel Compiler 11.1 for the CUDA and host codes. To compile the OpenACC codes, we use PGI Compiler 12.10, HMPP 3.2.4, and Cray Compiler 8.1.0.165 with CUDA 4.1. However, because the HMPP and Cray compilers were unable to compile the UPACS code, even without our OpenACC extensions, only the PGI compiler is used for the application case study. Optimization option -O3 is used with all compilers as well as -static -xP -openmp with the Intel compiler.

A. Matrix Multiplication

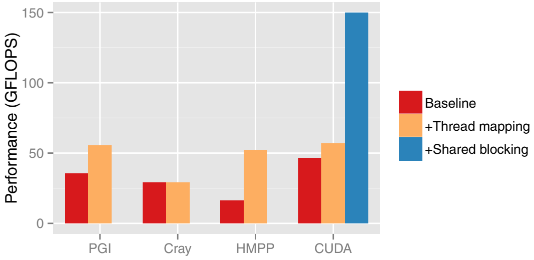


Figure 4. Performance of multiplication of 2048^2 matrices.

Table II
Number of modified lines of code in matrix multiplication.

	Baseline	Thread mapping	Shared blocking
OpenACC	9	11	
CUDA	26	26	45

Figure 4 shows the performance of double-precision matrix multiplication of two 2048^2 matrices on an M2050 GPU without counting the PCI transfer overhead. We use Fortran in both OpenACC and CUDA. Table II shows the number of modified or inserted lines of code of each version to the original CPU code. From these results, we see that the baseline OpenACC versions with the three OpenACC compilers achieve approximately 50% to 85% of performance of the baseline CUDA version with less than half of code changes. The thread mapping optimization is effective for the PGI and HMPP compilers, and improves the baseline performance by 1.6 and 3.2 times, respectively. These results indicate that thread mapping should be carefully tuned when using these two compilers. The HMPP compiler in particular generates a poorly performing mapping by default: Only sixteen thread blocks of 256 threads are spawned. Because there are fourteen SMs in Tesla M2050, this configuration would cause a load imbalance.

The shared memory blocking optimization implemented in the optimized CUDA version significantly improves the performance by a factor of three. As discussed in Section IV-A2, all three OpenACC compilers were unable to successfully compile the benchmark program with the `cache` directive. While the performance of the optimized CUDA version is still far behind that of the fully tuned DGEMM, these results indicate that a relatively simple optimization can still have a large performance impact and is not yet automated in the OpenACC compilers.

B. 7-point Stencil

Figure 5 shows the performance in terms of achieved memory throughput for a single-precision 7-point stencil on a 256^3 grid. The number of modified or inserted lines of code is shown in Table III. We use the same GPU and compiler configuration as used in the matrix multiplication experiment. The best performance achieved is 81.79 GB/s when the CUDA version is fully optimized. An experiment using the bandwidthTest program included in the CUDA SDK shows that the on-memory data throughput is approximately 108 GB/s, indicating that our optimized CUDA stencil is well tuned.

Among the three OpenACC compilers, the PGI compiler performs best with a performance 19% lower than the corresponding CUDA versions. The thread mapping optimization does not yield much of a performance improvement with PGI and Cray, whereas it produces a substantial improvement with HMPP. This result is consistent with the matrix multiplication results on the thread mapping effect. Both the branch hoisting and register blocking optimizations have mixed effectiveness. This result indicates that the manual optimizations that have been known to be effective in CUDA

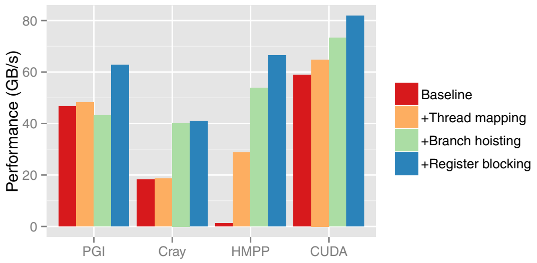


Figure 5. Throughput of 7-point stencil with 256^3 grids.

Table III
NUMBER OF MODIFIED LINES OF CODE IN 7-POINT STENCIL.

	Baseline	Thread mapping	Branch hoisting	Register blocking
OpenACC	7	10	18	29
CUDA	35	35	45	56

can also be important in OpenACC, but their effectiveness depends on the actual compilers used to generate the final code. In addition, as expected, the number of modified or inserted lines of code in OpenACC is significantly smaller than that of CUDA.

C. UPACS

Figure 6 shows the overall performance of the GPU UPACS versions when using the PGI OpenACC and CUDA Fortran compilers relative to OpenMP executed on a six-core Xeon CPU. The GPU performances include the overhead of host-device data transfer. The phase-wise performance is shown in Figure 7. Note that the versions where each optimization is not applicable are left blank in the figures. As shown in Figure 6, the baseline GPU performance with the PGI OpenACC compiler is 1.4 times higher than that of the CPU version. Similar to the 7-point stencil with the PGI Compiler, the thread mapping optimization slightly improves the performance, achieving 1.48 times faster performance when compared to the CPU performance. With the remaining optimizations in place, the final performance is 1.8 times faster than the CPU performance.

When compared to OpenACC, even the baseline CUDA version is faster than the fully optimized OpenACC version, reaching more than 2.1 times faster performance than that of the CPU version. Among the five optimizations, the shared-memory based MFGS optimization is particularly effective, further improving the performance by a factor of 1.5.

Table IV shows the number of modified or inserted lines of code. The manual optimizations significantly increase the code changes in both OpenACC and CUDA, although the former is significantly smaller than the latter. Note that because of the minimum changes required in UPACS with the PGI OpenACC compiler, we see that even the baseline OpenACC version has almost the same degree of modifications as that of the original CPU UPACS code.

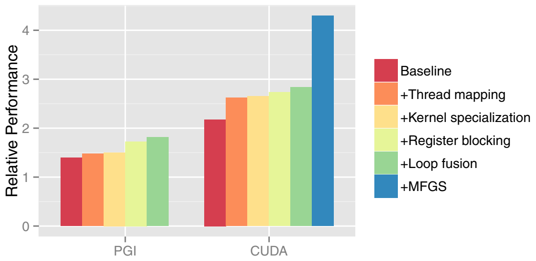


Figure 6. UPACS performance relative to the CPU performance. Non-applicable optimizations are left blank.

Overall, both the OpenACC and CUDA versions successfully achieve faster performances when compared to the original CPU version. However, the fully tuned CUDA version is faster than the corresponding version of OpenACC by a factor of 2.7, although with a much larger degree of code changes. The results indicate that OpenACC still has a considerable room for performance improvement.

VI. RELATED WORK

The PGI Accelerator Model is also an extension of OpenMP for accelerator programming, and is a precursor of the current OpenACC specification [7]. It basically presents the same architecture model to the programmer, i.e., multiple levels of different parallelism with the memory space separated from the host memory. The directives provided by the PGI Accelerator model also resemble the OpenACC directives, with several minor differences. Another precursor of the OpenACC specification is the CAPS HMPP Workbench [8], which is also a directive-based accelerator programming framework.

In contrast to the high-level abstractions provided in the work discussed above, hiCUDA prioritizes achievable performance with relatively lower level abstractions [9]. It specifically targets the NVIDIA GPU architectures, and by doing so allows the programmer to directly control the data movement between the various CUDA-specific memories, including the global, constant, shared memories. As shown in our evaluation, this level of flexibility can sometimes be critical to enable optimized performance to be achieved in real-world applications, especially for experienced programmers. It would be useful if a compiler-based automated approach such as OpenACC and a lower-level explicit model could coexist within the same unified programming model.

Another extension of OpenMP for OpenACC is presented by Bayer et al. [10]. The extension is quite similar to the OpenACC specification, and a preliminary performance evaluation with matrix multiplication shows a similar performance result to that of our matrix multiplication evaluation.

Performance of an earlier version of the Cray OpenACC compiler are reported by Wienke et al. [11]. Similar to our

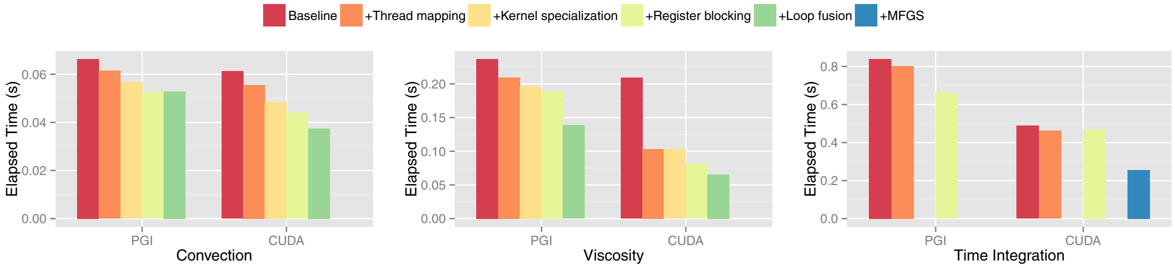


Figure 7. Elapsed time of each UPACS phase. Non-applicable optimizations are left blank.

Table IV
NUMBER OF MODIFIED LINES OF CODE IN UPACS.

	Baseline	Thread mapping	Kernel specialization	Register blocking	Loop fusion	MFGS
OpenACC	1788	1809	2601	2940	3296	
CUDA	5607	5743	6614	7641	8656	8870

work, their results indicate that OpenACC performance is generally similar to or lower than that of OpenCL, partly because of less efficient memory access. Our work presents more detailed performance studies with multiple levels of optimizations, and identifies that the lack of interface to access the on-chip memory can severely limit the performance when compared to hand-tuned CUDA code.

VII. CONCLUSION

This paper studied the performance implications of OpenACC for two microbenchmarks and one real-world CFD application. We first explored the baseline porting strategies of matrix multiplication, a 7-point stencil, and the UPACS application. We then studied the effectiveness of the common and application-specific optimization techniques in both OpenACC and CUDA. The evaluations indicate that the current OpenACC compilers achieve approximately 50% of performance of the CUDA versions, reaching up to 98% depending on the compiler. However, the limitation on the on-chip memory causes a significant performance gap when compared to the shared-memory optimized CUDA code.

The lack of programmable control of the on-chip memory in OpenACC may be alleviated by introducing low-level abstractions, such as those explored in the hiCUDA project [9]. However, low-level interfaces are typically specific to particular architectures, and may lose program portability. The question of how performance and program portability can coexist remains an open question. One promising approach would be to improve compiler-based optimizations by autotuning as is partially done in [12].

ACKNOWLEDGMENTS

This project was partially supported by JST, CREST through its research program: "Highly Productive, High Performance Application Frameworks for Post Petascale Computing."

REFERENCES

[1] "The OpenACC Application Programming Interface, Version 1.0," November 2011.

[2] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable Parallel Programming with CUDA," *ACM Queue*, vol. 6, no. 2, pp. 40–53, Mar. 2008.

[3] Khronos OpenCL Working Group, "The OpenCL Specification, Version 1.2," November 2011.

[4] R. Takaki, K. Yamamoto, T. Yamane, S. Enomoto, and J. Mukai, "The Development of the UPACS CFD Environment," in *High Performance Computing*, ser. Lecture Notes in Computer Science, A. Veidenbaum, K. Joe, H. Amano, and H. Aiso, Eds., 2003, vol. 2858, pp. 307–319.

[5] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey, "3.5-D Blocking Optimization for Stencil Computations on Modern CPUs and GPUs," in *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10)*, 2010, pp. 1–13.

[6] NVIDIA Corporation, "NVIDIA CUDA C Programming Guide Version 4.2," April 2012.

[7] M. Wolfe, "Implementing the PGI Accelerator model," in *Proceedings of Workshop on General-Purpose Computation on Graphics Processing Units*, 2010, pp. 43–50.

[8] R. Dolbeau, S. Bihan, and F. Bodin, "HMPP: A Hybrid Multicore Parallel Programming Environment," in *Proceedings of the Workshop on General Purpose Processing on Graphics Processing Units (GPGPU)*, October 2007.

[9] T. D. Han and T. S. Abdelrahman, "hiCUDA: a high-level directive-based language for GPU programming," in *Workshop on GPGPU*, 2009, pp. 52–61.

[10] J. C. Beyer, E. J. Stotzer, A. Hart, and B. R. Supinski, "OpenMP for Accelerators," in *7th International Workshop on OpenMP (IWOMP'11)*, 2011, pp. 108–121.

[11] S. Wienke, P. Springer, C. Terboven, and D. Mey, "OpenACC – First Experiences with Real-World Applications," in *Euro-Par*, 2012, pp. 859–870.

[12] S. Lee and R. Eigenmann, "OpenMPC: Extended OpenMP Programming and Tuning for GPUs," in *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10)*, 2010, pp. 1–11.