Experiences of Using the OpenMP Accelerator Model to Port DOE Stencil Applications

Pei-Hung Lin[1(✉)], Chunhua Liao[1], Daniel J. Quinlan[1], and Stephen Guzik[2]

[1] Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, USA
{lin32,liao6,dquinlan}@llnl.gov

[2] Mechanical Engineering Department, Colorado State University, Fort Collins, USA
stephen.guzik@colostate.edu

Abstract. The Department of Energy has a wide range of large-scale, parallel scientific applications running on cutting-edge high-performance computing systems to support its mission and tackle critical science challenges. A recent trend in these high-performance computing systems is to add commodity accelerators, such as Nvidia GPUs and Intel Xeon Phi coprocessors, into computer nodes so we can achieve increased performance without exceeding the limited power budget. However, it is well-known in the high-performance computing community that porting existing applications to accelerators is a difficult task given the numerous set of unique hardware features and the general complexity of software. In this paper, we share our experiences of using the OpenMP Accelerator Model to port two stencil applications to exploit Nvidia GPUs. Introduced as part of the OpenMP 4.0 specification, the OpenMP accelerator model provides a set of directives for users to specify semantics related to accelerators so that compilers and runtime systems can automatically handle repetitive and error-prone accelerator programming tasks, including code transformations, work scheduling, data management, reduction, and so on. Using a prototype compiler implementation based on the ROSE source-to-source compiler framework, we report the problems we encountered during the porting process, our solutions, and the obtained performance. Productivity is also evaluated. Our experience shows that the existing OpenMP Accelerator Model can effectively help programmers leverage accelerators. However, complex data types and non-canonical control structures can pose challenges for programmers to productively apply accelerator directives.

LLNL-CONF-670941. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. This work was also supported by the National Science Foundations Computer Research Infrastructure program under Award No. CNS-1205708. The rights of this work are transferred to the extent transferable according to title 17 §105 U.S.C.

C. Terboven et al. (Eds.): IWOMP 2015, LNCS 9342, pp. 45–59, 2015.
DOI: 10.1007/978-3-319-24595-9_4

1 Introduction

The Department of Energy (DOE) has a wide range of large-scale, parallel scientific applications to support its mission and tackle critical research and development challenges in multiple science disciplines. Many of these scientific applications have a lifespan of multiple decades so it is essential to port them to current mainstream high-performance computing (HPC) systems deployed in DOE in a timely fashion. A recent trend in the HPC systems is to add commodity accelerators, such as Nvidia GPUs and Intel Xeon Phi coprocessors, into computer nodes so we can achieve increased performance within a limited power budget. However, it is well-known in the HPC community that porting existing applications to accelerators is a difficult task given the numerous unique set of hardware features of accelerators and the complexity of software.

Although low-level programming models, such as CUDA [2] and OpenCL [10], can often help deliver competitive performance for certain applications, they are not productive porting solutions for large-scale parallel applications due to the extreme and comprehensive changes required in the original source code. On the other hand, high-level programming models such as OpenMP 4.0 [14] and OpenACC [4] provide language annotations in the form of directives and clauses for users to incrementally specify the semantics for porting to an accelerator. Compilers and runtime systems then automatically take care of repetitive and error-prone code transformations, thread scheduling, data management, and so on. Therefore, it is more productive for users to use high-level directive-based programming models to test the feasibility and profitability of using accelerators.

The OpenMP Accelerator Model, introduced as part of the OpenMP 4.0 specification, is a representative high-level directive-based programming model aimed to simplify the programming for accelerators. In a previous study [12], we created a prototype compiler for the OpenMP Accelerator Model and obtained an early assessment. We extend our work by applying the model to port two non-trivial DOE scientific applications: lattice-Boltzmann method and Compressible Navies-Stokes equation. Both applications conduct a stencil computation, an important category of scientific computing done in DOE facilities. However, they have very different stencil sizes so they represent a spectrum of stencil applications. However, they represent a spectrum of stencil applications by their difference in stencil sizes. Our goal is to discover problems developers may face when using the OpenMP Accelerator Model to port real applications. We also share our solutions to the problems, including suggestions to improve the programming model itself. Our contributions include: (1) providing the first study using the OpenMP Accelerator Model in OpenMP 4.0 to port non-trivial scientific applications, (2) illustrating the obstacles for porting real applications and possible solutions and workarounds, and (3) suggesting improvements, including new language features, of the OpenMP Accelerator Model to increase expressiveness and performance for accelerators.

The remainder of this paper is organized as follows. Section 2 gives an overview of the accelerator support in the OpenMP 4.0 specification. Section 3 describes the two applications. Porting experiences and performance results are given in Sect. 4. Section 5 summarizes related work and Sect. 6 presents the conclusion and future work.

2 OpenMP 4.0's Accelerator Support

OpenMP is a representative high-level directive-based programming model originally designed to address shared-memory programming. Starting from OpenMP 4.0, it has a set of language directives and runtime routines aimed at simplifying the programming for accelerators. Collectively, the accelerator support is often called the OpenMP Accelerator Model. The OpenMP accelerator model assumes that a computation node has a host device connected with one or multiple target devices. A target device, which can be any logical execution engine defined by an implementation, has threads that behave almost the same as threads on the host device. The OpenMP memory model is extended so that the code region has its own data environment. A device appears to have an independent memory, although it is allowed to share memory among devices.

The execution model is host-centric: a host device "offloads" data and code regions to accelerators for execution. In particular, the `target` construct is introduced for specifying a computation and the associated data to be offloaded to a device. Initially, only a single thread starts on a device to run an implicit task region. This single thread can fork more threads later when it encounters parallel constructs. Data-mapping attributes, specified using the `map` clause, define how variables are handled for the device data environments. Data mapping often involves data movement as host and device are commonly in different memory spaces in modern accelerator architectures. To avoid repetitive creation and cancellation of device data environments, the `target data` directive defines a device data region, in which multiple target regions can share the same device data.

Accelerators are often massively parallel architecture devices that support many concurrent threads with a hierarchical organization. OpenMP 4.0 provides the `teams` and `distribute` constructs to manage a two-level thread hierarchy. `teams` creates a league of thread teams, and the master thread of each team executes the region. `distribute` is closely nested in a teams region to share work among master threads of teams. Other features in the OpenMP accelerator model include a `target update` directive to make specified items in the device data environment consistent with their original list items, a `target declare` directive to specify the variables or functions to be mapped to a device, some combined constructs to simplify the programming, and an environment variable (`OMP_DEFAULT_DEVICE`) to indicate the default device number, and a set of runtime library routines to set and detect information related to accelerators.

3 Applications

Stencil computations are used in many large DOE scientific applications to solve partial differential equations on structured grids. In this paper, we chose two stencil applications, one using the lattice-Boltzmann method (`LBM`) and the other solving the compressible Navier-Stokes equation (`CNS`), to represent non-trivial scientific applications. The chosen `LBM` and `CNS` algorithms have very different stencil sizes (0-point vs. 25-point) leading to different computational characteristics. The `LBM` method operates in a *streaming* mode; memory is read once to perform the computation in the 0-point grid site. In the CNS method, memory from a grid site is repeatedly used in all the stencils that include that grid site. Hence, effective caching is extremely important. With effective caching, the arithmetic intensity (FLOPS per unit byte) can be quite high. The performance of the LBM algorithm is often limited by bandwidth whereas the performance of the CNS algorithm is often limited by arithmetic resources. These different characteristics can lead to different implementation strategies when porting the applications to a GPU device. We list a high level comparison between two applications in Table 1.

Table 1. Comparison between LBM and CNS applications

	Language	AMR library	Stencil	Components	Lines in codes
LBM	C++	Chombo	0-point	19	4670 (12879 w/Chombo code)
CNS	Fortran90	BoxLib	1D: 9-point 3D: 25-point	11	1242 (25967 w/BoxLib code)

In the LBM, hydrodynamics are described by a discrete kinetic equation for a single-particle distribution function [5].

$$\underbrace{f_i(\boldsymbol{j}+\boldsymbol{e}_i\Delta t, t+\Delta t) = \hat{f}_i(\boldsymbol{j},t)}_{\text{Streaming}} = \underbrace{f_i(\boldsymbol{j},t) + \mathcal{L}_{ik}\left(f_k(\boldsymbol{j},t) - f_k^{\text{eq}}(\boldsymbol{j},t)\right)}_{\text{Collision}} . \tag{1}$$

The chosen LBM application uses Chombo [6], a parallel adaptive mesh refinement (AMR) library used to solve partial differential equations. The domain size selected in the experiment is a 64^3 Cartesian grid structure partitioned into *boxes*, each of size 32^3. A total of 8 boxes cover the problem domain and 8000 time steps are performed in a single experiment. Figure 1 shows the pseudo code for the LBM computation. In the experimental setup, a loop in the application iterates over 8 boxes and performs computations to update the grid cells in each box (represented in line 8). Parallelization can be applied to the loop over boxes (line 8) or loops over grid cells (line 11 and line 16). Multi-level parallelization is feasible only if it is supported in the implementation.

The CNS algorithm is based on finite-difference methods and the equations are:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \boldsymbol{u}) = 0, \tag{2}$$

$$\frac{\partial \rho \boldsymbol{u}}{\partial t} + \nabla \cdot (\rho \boldsymbol{u}\boldsymbol{u}) + \nabla p = \nabla \cdot \boldsymbol{\tau}, \tag{3}$$

$$\frac{\partial \rho E}{\partial t} + \nabla \cdot [(\rho E + p)\boldsymbol{u}] = \nabla \cdot (\lambda \nabla T) + \nabla \cdot (\boldsymbol{\tau} \cdot \boldsymbol{u}), \tag{4}$$

where ρ is the density, $\boldsymbol{u}$ is the velocity, p is the pressure, E is the specific energy density (kinetic energy plus internal energy), $\boldsymbol{\tau}$ is the viscous stress tensor, λ is the thermal conductivity, and T is the temperature. The problem domain in CNS is represented by BoxLib [1], an AMR library very similar to Chombo. The domain size of the CNS experiment is 64^3 and partitioned into "Fabs" (Fortran array boxes), each of size 32^3. 50 time steps are performed and 5 output files are

```
 1 fi(cells, 19, boxes) = initial data;
 2 fiUpdate(cells, 19, boxes) = 0;
 3 U(grid, 4, boxes);
 4 Macroscopic(U, fi);
 5 for (int iTS = 0; iTS != nTimeStep; ++iTS)
 6 {
 7   int iBox;
 8   for (every box)
 9   {
10     {  // Advance function
11       for (every cell)
12         Collision(fi, U);
13       Exchange(fi);
14       BC(fi);
15       Stream(fiUpdate, fi);
16       for (every cell)
17         Macroscopic(U, fiUpdate);
18       swap(fi, fiUpdate);
19     }
20   }
21 }
```

Fig. 1. LBM algorithm pseudo-code

```
 1 fi(cells, 19, boxes) = initial data;
 2 fiUpdate(cells, 19, boxes) = 0;
 3 U(grid, 4, boxes);
 4 Macroscopic(U, fi);
 5 for (int iTS = 0; iTS != nTimeStep; ++iTS)
 6 {
 7   int iBox;
 8   for (every box)
 9   {
10     {  // Advance function
11       for (every cell)
12         Collision(fi, U);
13       Exchange(fi);
14       BC(fi);
15       Stream(fiUpdate, fi);
16       for (every cell)
17         Macroscopic(U, fiUpdate);
18       swap(fi, fiUpdate);
19     }
20   }
21 }
```

Fig. 2. CNS application pseudo-code

generated during the computation. An outer loop iterates over all available Fabs in the "multi-Fab" data structure (shown in line 5 in Fig. 2). Similar to the LBM, multi-level parallelization is applicable if it is supported in the implementation.

4 Porting to GPUs

Our porting process starts with obtaining baseline performance of OpenMP versions of the applications. We incrementally add additional accelerator directives and clauses to show the programming effort and performance impact. In particular, we experiment with directives and clauses for data reuse, loop collapsing, loop scheduling and hierarchical thread mapping.

The hardware platform has 132 GB memory, two 8-core Intel E5-2670 CPUs, and two Nvidia K20x GPUs. We use a prototype implementation of the OpenMP Accelerator Model, HOMP (Heterogeneous OpenMP) [12], which is built on the ROSE source-to-source compiler infrastructure [15] developed at Lawrence Livermore National Laboratory. The built-in OpenMP implementation in ROSE supports OpenMP 3.0 directives for C, C++ and a subset of Fortran. Leveraging ROSE's flexibility to experiment with new language extensions, HOMP adds the OpenMP accelerator support [12], including parsing and code transformations for target, target data, map and so on. HOMP generates CUDA code for the growing demands in GPU programming. The original OpenMP runtime library (referred to as XOMP) for ROSE has been extended to support thread configuration, loop scheduling, data management, reduction and many other required operations on GPUs. We use the GNU Compiler Collection (gcc-4.4.6), Nvidia 6.0 SDK, nvcc compiler, and the Nvidia Visual Profiler [3] in this study.

4.1 Baseline Performance on CPU and GPU

The default setup in the LBM application has OpenMP directives inserted into the loop for boxes (line 8 in Fig. 1). The OMP_NUM_THREADS environment variable

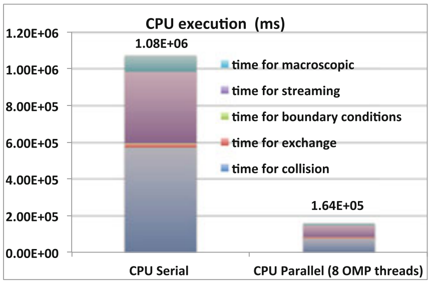

Fig. 3. LBM CPU baseline performance

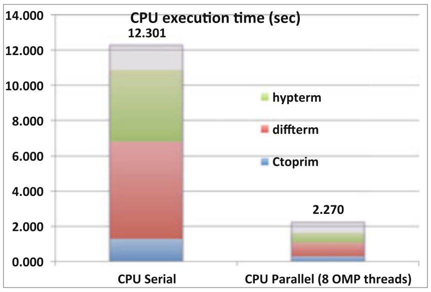

Fig. 4. CNS CPU baseline performance

is set to 8 to assign at most 8 OpenMP threads to update the 8 boxes in the loop. We assign at most 8 OpenMP threads to update the 8 boxes in the loop. Each OpenMP thread will then update 32^3 cells inside a box, a strategy that works well for boxes of this size [13]. The OpenMP parallel region terminates at the end of the loop to form an implicit synchronous barrier between time steps. Figure 3 shows the CPU's serial and parallel performance. The parallel execution with 8 OpenMP threads delivers a 6.76× speedup compared to the serial execution on the testing system.

The CNS application by default has OpenMP directives at the loops for grid cells (line 9, 11, and 13 in Fig. 2). These loops are 3-level nested loops that iterate through the cubical structure in a Fab. The whole application consists of 14 such OpenMP parallel loops. In the configured testing case, loop iterations in the outermost loop are evenly distributed into 8 OpenMP threads for 8 boxes. Figure 4 shows the comparison between serial and parallel execution using 8 threads. The parallel execution delivers a 5.42× speedup on the testing machine.

Before the porting, we discovered a few obstacles to adding OpenMP accelerator directives. We had to modify a subset of code from both applications to make the porting feasible. For example, the current HOMP only supports C/C++ input code to generate CUDA code for the GPU. We used a Fortran-to-C translator implemented in ROSE to translate the functions in the CNS into C language versions for the porting. In the LBM application, several variables used in the target loops were not mappable by the OpenMP 4.0 specification because they are part of other C++ class objects. We copied those variables to temporary variables and mapped the temporary variables as a workaround. The baseline implementations on the GPU simply reuse the OpenMP parallel directives without any optimization involved. Minimal OMP target and OMP map directives are used to identify the target region and data to be mapped onto the device.

For the LBM application, the location of OpenMP directives in the CPU implementation is not an ideal start location for the GPU implementation since it contains multiple kernels in the loop body. Using an incremental approach, we ported individual kernels first and moved the OpenMP directives to the locations of loops to the grid cells inside Collision, Macroscopic and Stream functions (shown at line 11, 15 and 16 in Fig. 1). These three functions consume the majority of execution time (47 % in Collision, 40 % in Stream and 7 % in Macroscopic)

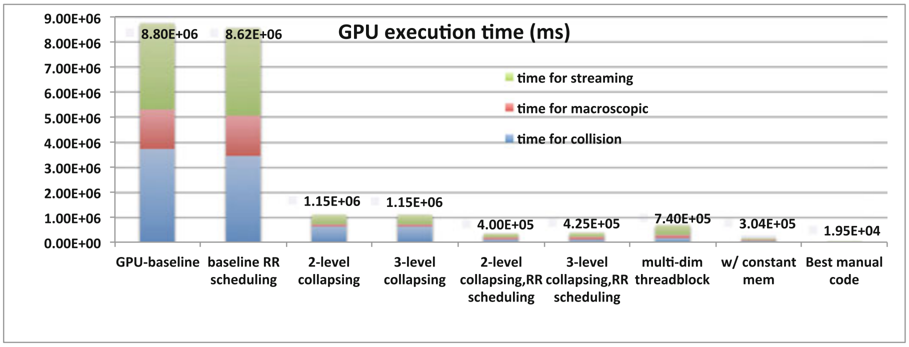

Fig. 5. `LBM` performance on GPU

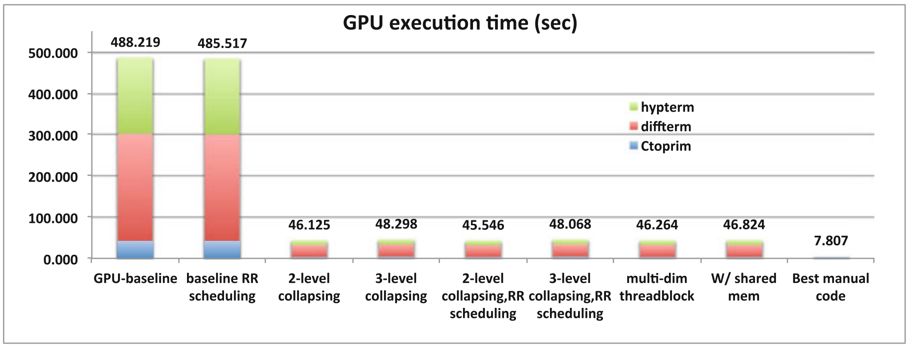

Fig. 6. `CNS` performance on GPU

on the parallel CPU execution. The GPU baseline implementation for the `CNS` application has OpenMP directives inserted into 1 loop in the ctoprim function, 3 loops in the hypterm function, and 7 loops in the diffterm function. Those are the same locations that have OpenMP directives in the parallel CPU implementation. Diffterm function takes the greatest portion (34 %) portion of total execution in the `CNS` application. Hypterm and ctoprim take 24 % and 13 %, respectively.

The baseline GPU performance in both applications were not competitive compared to their corresponding CPU version performance (shown in Figs. 5 and 6). After inspection with the Nvidia Visual Profiler [3], we found that the baseline GPU implementations have extremely low achieved GPU occupancy (<2 %). The baseline GPU implementations have extremely low achieved GPU occupancy (<2 %). This is due to the nested loops, identified by the OpenMP directives, which have only small loop iteration sizes in their outermost loop. The translated CUDA codes exploit at most 40 GPU threads to perform the computation and result in low parallelism and performance. The next step in porting was to improve the GPU utilization by increasing the parallelism.

4.2 Increasing Parallelism

Achieving high parallelism is the key for a GPU device to get high computing performance. In addition to optimizing applications for high parallelism, the porting process needs to take into account that the maximum parallelism in the real execution is subject to certain CUDA limitations. These are the limitations for K20X GPU used in this paper:

– At most 1024 threads in a thread block.
– At most 64 warps (32 threads/warp) in a SMX.
– A thread can have up to a 63 register usage.
– Each SM has up to 48 KB shared memory shared by multiple thread blocks.

We describe two feasible approaches to increasing parallelism for the chosen applications.

The first approach is loop collapsing. Loop collapsing is a transformation that converts multiple perfectly nested loops into a single loop. Compared to the original outermost loop, the collapsed loop has a larger iteration size with potential to expose higher parallelism. We apply the directive #pragma omp for collapse (n) to perform loop collapsing. However, loop structure in the `LBM` application has statements between the nested loops and does not form a perfectly nested loop. Collapsing non-perfectly nested loops is not allowed by the OpenMP specification. After reviewing the nested loop structure, we manually moved statements between loops in `LBM` application into the innermost loop body since this change causes no side effect and can form a perfectly nested loop. After collapsing, we could exploit more GPU threads to perform parallel execution on the collapsed loop. Therefore, more GPU threads could be assigned to perform parallel execution on the collapsed loop. The XOMP runtime incorporates the CUDA runtime to maximize the utilization of the GPU threads. Compared with the baseline GPU implementations, there are about 5× and 10× speedups delivered for the `LBM` and `CNS` applications respectively (shown in Figs. 5 and 6).

The second option to increase parallelism is to use the multi-dimensional thread structure supported in CUDA. In the `LBM` application, we can seamlessly allocate 32×32 threads to a thread block and have 32 thread blocks mapped to the outermost loop. This can achieve 100 % occupancy in the execution if only 32 registers are given to each GPU thread. But there are only two concurrent thread blocks in the setup due to the limitation in the allowed warp number. In the `CNS` application, we can have the same allocation if ghost cells are not involved in the computation. Otherwise, the loop iteration size becomes 40 (32 and 4 ghost cells on both sides) in the three-level nested loop. To fulfill the CUDA limitation discussed earlier, we allocate only 40 threads in a thread block and have multiple thread blocks mapped to the loop iteration space. 16 concurrent thread block are allowed in executions, and it is also the maximum allowed number in this GPU model. This configuration has lower theoretical occupancy (50 %) and the computation is inefficient due to the usage of partial-warp. The performance is reported in histograms marked with multi-dim threadblock in Figs. 5 and 6. Compared with the collapsing variants, a 1.5× speedup is achieved in the `LBM` application but a marginal difference is shown for the `CNS` application.

4.3 Loop Scheduling

OpenMP supports multiple loop scheduling policies, including static, dynamic, guided, auto, and runtime. For regular loops running on CPUs, statically and evenly dividing loop iterations among threads using a `schedule(static)` clause (referred to as static-even schedule in this paper) often leads to the best performance with minimal scheduling overhead. On the GPU, we need to perform coalesced memory access for high performance. The static-even schedule will have one GPU thread accessing multiple successive words in memory and lead to multiple memory transactions. A round-robin scheduling using `schedule(static,1)` will fulfill the need to perform coalesced memory access on the GPU device. We apply the round-robin schedule and compare only the kernel execution times in the `CNS` application. Round-robin scheduling delivers the highest (76 %) improvement in one kernel in the hypterm function and an average of 26.4 % improvement for all kernels. Performance reports show modest improvement for total execution time in the `CNS` application (1 %) and a larger improvement in the the `LBM` application (2.8$\times$). The performance analysis reports high overhead due to memory movement between the host and device memories.

4.4 Exploiting Memory Hierarchy

Nvidia GPUs provide multiple specialized memories, including on-chip software controllable cache shared within a thread block (referred to as shared memory) and constant memory accessible by all threads for read-only global data. The current OpenMP 4.0 lacks support to exploit the specialized memories. We propose to extend the OpenMP Accelerator Model to have a `cache` clause to allow users to hint such opportunities. The clause has a form of `cache (var_list)`, in which each variable listed can be further prepended by an optional `const` modifier. For example `cache (array1[0:10], const array2[5:10])` tells the compiler that there are two arrays which should be cached in the memory hierarchy of the accelerator. One of the arrays is a read-only subarray. Similar to the `map` clause, the `cache` clause can only be used with `target` or `target data` directives. Variables shown in the `cache` clause must also show up in the `map` clause affecting the same code region. With this clause, compilers translate the code to exploit either the shared memory or the constant memory of GPUs.

After evaluating the two applications, the `LBM` gained more benefits from the constant memory than the shared memory. We can store many constant coefficients, stride distances, and an array storing discrete velocity directions and an array storing weights in the constant memory space. Figure 7 extracts the comparison (execution time includes memory copying overhead) with only two kernels in the `LBM` application to demonstrate the performance with constant memory usage. A 1.32$\times$ speedup is achieved for the overall execution time from the implementation with constant memory. Higher speedups, from 1.74$\times$ to 2.44$\times$, were observed in the execution times for these three functions individually.

On the other hand, the `CNS` has relatively low constant data referenced by multiple functions. But the `CNS` application uses a 25-point stencil in the 3D

Table 2. Shared memory usage and GPU occupancy

Shared memory report			
Kernel	Size/block (byte)	Threads/block	Occupancy
Hypterm original	1920	40	50 %
Tiled 2 iterations	3840	80	56 %
Tiled 3 iterations	5760	120	50 %
Tiled 4 iterations	7680	160	47 %
Diffterm original	3520	40	41 %
Tiled 2 iterations	7040	80	28 %
Tiled 3 iterations	10620	120	25 %
Tiled 4 iterations	14080	160	23 %

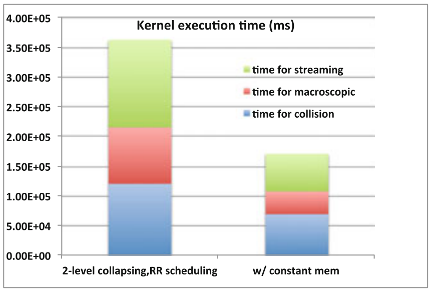

Fig. 7. LBM with constant memory

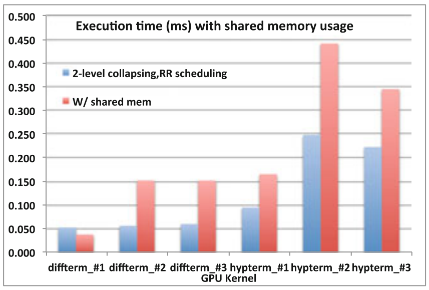

Fig. 8. CNS with shared memory

computation. Stencil data can therefore be stored in the shared memory space to gain the benefit of the fast memory. We used shared memory for six kernels (3 in Hypterm and 3 in Diffterm) in the GPU implementation for the CNS application. Table 2 shows the details of the required shared memory size, thread assignment and the achievable highest GPU occupancy. This implementation doesn't deliver higher performance compared to our earlier implementation with the best performance (shown in Fig. 8) due to a much lower GPU occupancy. To increase the active thread number in each thread block, loop tiling can be performed in the loop for the second dimension in the 3D nested loop. We can exploit more GPU threads after loop tiling but it also proportionally increases the required shared memory size for each thread block. Table 2 also shows the changes in GPU occupancy by tiling both kernels with different tiled sizes. The GPU occupancy will be limited by the allowed 48KB shared memory size. We conclude that exploiting shared memory in our implementation for the CNS application does not improve performance. It would require other optimizations to achieve efficient shared memory usage.

4.5 Reducing Memory Movement Between Host and Device

We observed several variables and arrays are copied repetitively to the GPU's memory in different kernels. Using `target data` directives with `map` clauses can usually reduce repetitive memory allocations and transferring. However, we found that this is not a trivial task for the two chosen applications due to language restrictions. OpenMP 4.0 defines a set of restrictions for variables listed in the `map` clause, such as (1) data must have a complete type for C/C++, (2) a variable that is part of another variable (e.g. a field of a struct) is not allowed unless it is an array element or array section, (3) C++ class types mapped must not contain static data or virtual members, and (4) pointer types are allowed but the memory block to which the pointer refers to is not mapped. Chombo (used in the `LBM` application) and BoxLib (used in the `CNS` application) share a data structure called Fortran array box (Fab). Fab is a structure of arrays that can store multiple components and it provides a high-level data abstraction. Information, such as loop bounds, stencil size, and a data pointer to the component array, is packaged inside the Fab. Members in Fab contain primitive arrays, scalar variables, and some static data. An ideal strategy in the porting process is to copy the entire Fab structure to the GPU's memory space. However, the Fab structure is not mappable according to OpenMP 4.0. A workaround task is to extract and store all the members of Fab in primitive arrays. Then the temporary arrays can be mapped and copied to the GPU memory. This will involve a significant code modification in the porting process.

4.6 Manual Tuning for GPU Performance

We provide manual implementations for both applications to evaluate the achievable performance through manual performance tuning. We manually implement the chosen applications with the CUDA language and consider the possibilities to involve OpenMP 4.0 standards and compiler transformations to automate the process. The manual implementations serve as a reference to study the transformation obstacles in the design of the OpenMP accelerator model. Several manual optimizations require good understanding in the application design to perform code modifications and they are not implemented as automatic transformations in this study.

The manually-tuned GPU implementation for the `LBM` application significantly simplifies the Fab structure, restructures the code, and consolidates all the memory copying. Other optimizations include hand-tuned kernels (including BoxLib's exchange function), exploiting constant memory, and several code modifications specifically for the GPU implementation. A simplified Fab structure on GPU code is designed to store only the essential data members in the CPU's Fab structure. Data is allocated and copied to GPU memory once and reused by all the kernels listed in the pseudo code in Fig. 1. This optimized implementation delivers the best performance between the CPU's and GPU's implementations (shown in Fig. 5).

The manual tuning processes for the CNS application minimize memory copying between the host and device, exploit efficient usage of shared memory, and maximize GPU occupancy. A 4^3 thread block is chosen based on the ghost cell size in the computation to avoid the partial warp usage. The code was modified to have only minimal memory transfers between host memory and device memory. All initialized data stored in the Fab data structure is copied to the device memory before the computation. There are infrequent data movements which send only a subset of computed data back to the host memory for boundary exchange performed by the BoxLib library and visualization dumps. The manual code delivers the best GPU performance with about 6× speedup compared to the best implementation with the OpenMP accelerator model (shown in Fig. 6). However, the delivered performance is not superior to the performance on the CPU due to overheads in allocating, copying and freeing memory on the GPU. Eliminating that overhead for the CNS application, the GPU execution time for the three kernels is at a comparable level to the CPU execution time.

4.7 Productivity

We briefly discuss the productivity benefit by using the OpenMP accelerator model. We choose the line number as the metric to evaluate the gain in productivity. Table 3 lists the essential information for the study. The number of accelerator directives inserted, lines in source code being ported, lines in the transformed code on the CPU (host code), and the line of the generated CUDA code on the GPU (device code), are collected in the table. Besides the code generated by the HOMP compiler, each runtime function packs a series of low-level CUDA function calls and additional codes to perform the designated task. Without the runtime support, manual implementation needs to perform the same series of CUDA function calls repetitively. For both transformed host and device codes, Table 3 lists two counts with and without including the line numbers packaged by the runtime functions. The count with lines performed in the runtime functions provides an estimation for the code size in a manual implementation. As shown in the table, using a few lines of directives can essentially save the efforts of writing hundreds or even thousands of lines of generated code. Accelerator

Table 3. Productivity study using lines of code (LOC)

Functions	Source LOC	Directives	Host LOC		Device LOC		Ratio (LOC/directives)	
			A	B	A	B	A	B
LB collision	45	2	57	464	48	58	52.5	261.0
LB macroscopic	46	2	52	421	45	55	48.5	238.0
LB stream	21	2	53	460	35	45	44.0	252.5
CNS ctoprim	14	2	27	205	30	40	28.5	122.5
CNS hypterm	57	6	81	793	123	153	34.0	157.7
CNS diffterm	82	14	335	2647	206	276	38.6	208.8

A: Lines of code without counting in runtime;
B: Line sof code with counting in runtime

directives supported by the OpenMP 4.0 can greatly simplify the porting process and improve productivity. On the other hand, programming using the OpenMP accelerator model does require additional domain knowledge, analysis, or optimization to achieve high performance on the target platform. Occasional manual code changes are needed also to workaround some language restrictions or expose more parallelism. However, the efforts of learning low-level CUDA or OpenCL would be more significant.

5 Related Work

Many previous studies [8,9,11,17] have evaluated the performance and productivity of OpenACC using a range of kernels or applications. For example, Wienke et al. [17] presented their experiences with OpenACC using two real-world applications. OpenACC helped them reach 80 % of the best-effort OpenCL version in a moderately complex simulation kernel. They reported that the inability to exploit local memory of the GPUs could contribute to the loss of performance of other complex OpenACC applications. Herdman et al. [8] used a hydrodynamics mini-application to compare OpenACC, OpenCL and CUDA. They found that OpenACC was extremely viable but their OpenCL and CUDA versions were not optimized. Hoshino et al. [9] used both kernels and a real-world computational fluid dynamics applications to compare CUDA and OpenACC. They reported that some complex Fortran data types such as arrays of derived types and derived types with variable-length arrays are not supported by OpenACC, but extensively used in the code.

The application experience of using the OpenMP accelerator support is rare due to the lack of compiler support. Dietrich et al. [7] presented an approach to measure the performance of applications utilizing OpenMP offloadings. Their focus is at performance analysis on the Intel Xeon Phi coprocessor. Silva et al. [18] compared OpenACC and OpenMP for accelerator computing. A set of parallel programming patterns, not real applications, were used to compare language features. No performance experiments were done due to the lack of compiler support. Unat et al. [16] presented a domain-specific OpenMP-like programming model for stencil methods. For small kernels, they realized up to 80 % of the performance of optimized CUDA versions. Our work provides the first study of the performance and programmability of the OpenMP accelerator model using the HOMP compiler [12]. OpenACC [4] provides a cache (var_list) directive to support cache memory on accelerators. However, this directive may only appear inside loops. By contrast, our proposed cache() is a clause which can be used with one or multiple code regions. Besides leveraging the highest level of cache, the additional const modifier in our design can support the read-only semantics to exploit constant memory.

6 Discussion and Future Work

We have found that the OpenMP Accelerator Model is a productive approach for porting existing applications to GPUs. The porting strategy can be straightforward. Users should prepare a baseline OpenMP version running on CPUs.

Then the target directive can be inserted around parallel regions. There are only a limited set of accelerator directives and clauses in OpenMP 4.0 to improve parallelism, scheduling, and data reuse, among others. So a strategy is to incrementally apply them and check the effect by performance analysis tools.

However, real applications pose unique challenges to effectively apply directive-based programming models. (1) A scientific application often has complex data types which may not be supported by the language specifications. A common workaround is to manually copy a portion of the complex data object into a variable of a simpler, supported type. (2) An application may have non-perfectly nested loops, which can be a candidate for collapsing after simple transformations. One possible way to improve productivity is to extend the collapse(n) clause to accept a flag, like collapse(n:force), to force collapsing across multiple non-perfectly nested loops when applicable. Compilers could enforce a transformation to form a perfectly nested loop, but users have to ensure the correctness of the code movements. (3) Large-scale DOE applications usually leverage many third-party libraries to increase productivity. Porting such an application may involve a challenging task to port the underlying libraries. (4) In an ideal world, users should be able to simply insert directives into existing codes to port to new platforms. However, non-trivial code restructuring may be needed to expose the right granularity of parallelism. (5) Our attempt to exploit special caches on GPUs generated some interesting results. Using constant memory for `LBM` resulted in significant performance improvements. On the other hand, using shared memory for `CNS` does not deliver higher performance in our study. The intuitive implementation to exploit special caches on GPUs may degrade the performance. Additional analysis and optimization support will be helpful to achieve good performance on GPU devices.

Our future research directions are in the following: (1) testing extensions to port complex data types and non-canonical control structures (e.g. non-perfectly nested loops). (2) using more scientific applications to find improvements to the directive-based programming models, (3) further investigation of ways of exploiting shared memory for better performance in real applications, and (4) exploring extensions to express semantics related to managing multiple accelerator devices.

References

1. BoxLib. https://ccse.lbl.gov/BoxLib/
2. CUDA Zone - The resource for CUDA developers. http://www.nvidia.com/cuda
3. Nvidia visual profiler. https://developer.nvidia.com/nvidia-visual-profiler
4. OpenACC: Directives for Accelerators. http://www.openacc-standard.org/
5. Chen, S., Doolen, G.D.: Lattice Boltzmann method for fluid flows. Annu. Rev. Fluid Mech. **30**, 329–364 (1998)
6. Colella, P., Graves, D.T., Keen, N., Ligocki, T.J., Martin, D.F., McCorquodale, P., Modiano, D., Schwartz, P., Sternberg, T., Straalen, B.V.: Chombo software package for amr applications - design document. Technical report, Lawrence Berkeley National Laboratory (2009)

7. Dietrich, R., Schmitt, F., Grund, A., Schmidl, D.: Performance measurement for the OpenMP 4.0 offloading model. In: Lopes, L., Žilinskas, J., Costan, A., Cascella, R.G., Kecskemeti, G., Jeannot, E., Cannataro, M., Ricci, L., Benkner, S., Petit, S., Scarano, V., Gracia, J., Hunold, S., Scott, S.L., Lankes, S., Lengauer, C., Carretero, J., Breitbart, J., Alexander, M. (eds.) Euro-Par 2014, Part II. LNCS, vol. 8806, pp. 291–301. Springer, Heidelberg (2014)
8. Herdman, J., Gaudin, W., McIntosh-Smith, S., Boulton, M., Beckingsale, D., Mallinson, A., Jarvis, S.: Accelerating hydrocodes with OpenACC, OpeCL and CUDA. In: 2012 SC Companion: High Performance Computing, Networking, Storage and Analysis (SCC), pp. 465–471, November 2012
9. Hoshino, T., Maruyama, N., Matsuoka, S., Takaki, R.: CUDA vs OpenACC: performance case studies with Kernel benchmarks and a memory-bound CFD application. In: 2013 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), pp. 136–143, May 2013
10. Khronos OpenCL Working Group: The OpenCL Specification - Version 1.0. Technical report, The Khronos Group (2009)
11. Levesque, J.M., Sankaran, R., Grout, R.: Hybridizing S3D into an exascale application using OpenACC: an approach for moving to multi-petaflops and beyond. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC 2012, pp. 15:1–15:11. IEEE Computer Society Press, Los Alamitos, CA, USA (2012)
12. Liao, C., Yan, Y., de Supinski, B.R., Quinlan, D.J., Chapman, B.: Early experiences with the OpenMP accelerator model. In: Rendell, A.P., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2013. LNCS, vol. 8122, pp. 84–98. Springer, Heidelberg (2013)
13. Olschanowsky, C., Guzik, S.M.J., Loffeld, J., Hittinger, J., Strout, M.M.: A study on balancing parallelism, data locality, and recomputation in existing PDE solvers. In: The International Conference for High Performance Computing, Networking, Storage and Analysis (2014)
14. OpenMP Architecture Review Board: The OpenMP API Specification for Parallel Programming. http://www.openmp.org/
15. Quinlan, D.J., et al.: ROSE compiler project. http://www.rosecompiler.org/
16. Unat, D., Cai, X., Baden, S.B.: Mint: realizing CUDA performance in 3D stencil methods with annotated C. In: Proceedings of the International Conference on Supercomputing, ICS 2011, pp. 214–224. ACM, New York, NY, USA (2011)
17. Wienke, S., Springer, P., Terboven, C., an Mey, D.: OpenACC — first experiences with real-world applications. In: Kaklamanis, C., Papatheodorou, T., Spirakis, P.G. (eds.) Euro-Par 2012. LNCS, vol. 7484, pp. 859–870. Springer, Heidelberg (2012)
18. Wienke, S., Terboven, C., Beyer, J.C., Müller, M.S.: A pattern-based comparison of OpenACC and OpenMP for accelerator computing. In: Silva, F., Dutra, I., Santos Costa, V. (eds.) Euro-Par 2014 Parallel Processing. LNCS, vol. 8632, pp. 812–823. Springer, Heidelberg (2014)