Performant Portable OpenMP

Guray Ozen
gozen@nvidia.com
NVIDIA Corporation
Berlin, Germany

Michael Wolfe
mwolfe@nvidia.com
NVIDIA Corporation
Hillsboro, Oregon, USA

Abstract

Accelerated computing has increased the need to specialize how a program is parallelized depending on the target. Fully exploiting a highly parallel accelerator, such as a GPU, demands more parallelism and sometimes more levels of parallelism than a multicore CPU. OpenMP has a directive for each level of parallelism, but choosing directives for each target can incur a significant productivity cost. We argue that using the new OpenMP `loop` directive with an appropriate compiler decision process can achieve the same performance benefits of target-specific parallelization with the productivity advantage of a single directive for all targets. In this paper, we introduce a fully descriptive model and demonstrate its benefits with an implementation of the `loop` directive, comparing performance, productivity, and portability against other production compilers using the SPEC ACCEL benchmark suite. We provide an implementation of our proposal in NVIDIA's HPC compiler. It yields up to 56X speedup and an average of 1.91x-1.79x speedup compared to the baseline performance (depending on the host system) on GPUs, and preserves CPU performance. In addition, our proposal requires 60% fewer parallelism directives.

CCS Concepts: • **Software and its engineering** → **Compilers**; **Parallel programming languages**; • **Computer systems organization** → **Heterogeneous (hybrid) systems**.

Keywords: Compilers, Parallel Programming, GPUs, OpenMP

ACM Reference Format:
Guray Ozen and Michael Wolfe. 2022. Performant Portable OpenMP. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction (CC '22), April 02–03, 2022, Seoul, South Korea.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3497776.3517780

1 Introduction

Heterogeneous computers are now common for High Performance Computing. Programming heterogeneous systems requires the following steps:

1. Moving Data: determining, if the device has separate memory, which data to copy to the accelerator and back, and when.
2. Identifying Parallelism: deciding which loops or code blocks can be executed in parallel.
3. Target specific optimizations: mapping program parallelism to hardware parallelism, including loop scheduling, reordering, etc.
4. Device code: generating binary code for the target device.

In low-level programming models such as CUDA [25], OpenCL [18], pthreads [4], or vector intrinsics [15], the programmer is responsible for the first three steps and the last item is done by the compiler. Although it is possible to develop very efficient programs, productivity is low.

The OpenMP [26] API is a directive-based parallel programming model developed to address productivity and performance as well as portability across target systems. It has mostly followed a prescriptive philosophy where programmers choose parallelization strategies and specify these to the compiler and runtime. In this article we call this approach the Prescriptive OpenMP Model. Early studies showed how this model can be implemented in GPUs [1, 2, 19, 21, 27, 28]. Although directives definitely simplify programming, the programmer is still responsible for the first three steps above. Moreover, these steps may need to be repeated for each target, as exploiting parallelism may be significantly different for different target devices. Furthermore, different OpenMP implementations interpret some directives differently [5, 7, 11, 23], meaning different implementations may require different directives. This can make OpenMP programming for heterogeneous systems a daunting task.

We propose a fully descriptive OpenMP model with compiler based automatic parallelism mapping to increase all three of performance, portability, and productivity of OpenMP programming. This is *not* automatic parallelization. Our model lets the programmer tell what loops to parallelize, but not how to parallelize them, shifting this burden from the programmer to the compiler. By doing so, we improve portability and productivity by reducing the number of directives and clauses. We also get a performance advantage with our approach because the model is more scalable for massive architectures such as GPUs, and allows use of an optimized Single Program Multiple Data (SPMD) code generation strategy.

Our model uses the OpenMP `loop` construct, which was introduced with OpenMP 5.0, to allow the compiler to implicitly decide how to map parallelism. We have implemented our model and here evaluate its performance. Our contributions include:

- A simpler, descriptive OpenMP programming model, where the programmer says what to parallelize instead of how to do so.
- A programming model that can always run in SPMD mode, for which the compiler can generate performant code for multicore CPUs as well as highly parallel GPUs.
- An implementation of this model in NVIDIA's HPC compiler[1].
- Evaluation of the performance of our model against other current compilers with the SPEC ACCEL[17] benchmark suite. Our implementation yields up to 56X speedup and an average of 1.91x-1.79x speedup compared to the baseline performance (depending on the host system), and outperforms other compilers on GPUs, and matches or outperforms other compilers on CPUs.
- Evaluation of the productivity of our model by counting the number of parallelization directives and clauses required for heterogeneous programming compared to the Prescriptive OpenMP Model code. Our model requires 60% fewer parallelism directives while giving the performance benefits mentioned.
- Evaluation of the portability benefits of our model by comparing the performance of the same benchmarks on both GPU and CPU targets. Our model achieves equivalent or better performance to the baseline prescriptive OpenMP code on both GPU and CPU targets.

2 Background

2.1 NVIDIA GPUs and the CUDA Programming Model

This section provides an overview of the architecture of the current NVIDIA GPUs and the CUDA execution model. A GPU has a number of Streaming Multiprocessor (SM) units, where an SM roughly corresponds to a CPU core. Each SM has a large register file, a number of integer and floating point ALUs, instruction fetch units, branch processing units, load-store units, control units, and a small level 1 data cache. An instruction is fetched, decoded, and dispatched, just as on a CPU, except GPU threads are organized into groups of 32 threads, called a *warp*; all enabled threads in the warp will execute the same instruction at the same time.

[1]Formerly the PGI Compiler

In a CUDA program, the programmer writes *kernels* to execute on the GPU, and launches the kernel with a *launch configuration* that describes how many GPU threads to create. The GPU threads are grouped into a *grid* of *thread blocks* of *threads*, where the grid and the thread blocks may be described by a 1-, 2-, or 3-dimensional rectangle. The threads in any single thread block will be launched simultaneously on a single SM where they can coordinate and share some resources.

2.2 OpenMP

The OpenMP API[26] provides a set of directives to offload the execution of code regions onto accelerators, to map loops inside those regions onto the resources available in the device architecture, and to map and move data between address spaces. The main offload directives are `target data` and `target`, which create the data environment and offload the execution of a code region to an accelerator device, respectively. In classical OpenMP programs, the program creates a single *team* of *threads*, and uses the `parallel` and `for` or `do` constructs to workshare loop iterations across those threads. To take advantage of the higher degree of parallelism in GPUs, OpenMP added a `teams` construct to create multiple teams of threads. When teams are created, only the primary thread of each team is active until it encounters a `parallel` construct, which activates the other threads in the team. The `distribute` construct specifies how the iterations of a loop are workshared across the teams, just as the `for` or `do` constructs specify how the iterations of a loop are workshared across the threads within a team.

3 Motivation

In this section, we highlight two issues related to the Prescriptive OpenMP Model's design with respect to productivity and portability. First, since Prescriptive OpenMP requires a directive or clause for every parallelization decision, the user has to specify each choice, even the simplest ones, reducing productivity. Additionally, the user may have to repeat this decision-making for different devices if the devices are fundamentally different, such as a CPU and a GPU. Second, the OpenMP model is typically fork-join, whereas the native execution model on current GPUs is SPMD. If the OpenMP region can be converted to use SPMD execution, it will run more efficiently on a GPU. If not, the implementation must use a more complex thread management runtime, resulting in inferior performance. These two important issues motivate our research.

3.1 Performance Portability in the Prescriptive OpenMP Model

When we look at OpenMP from a high level, there are three constructs to create parallelism: `teams`, `parallel`, and `simd`. The `teams` construct is a way to execute code across multiple

Table 1. The top table shows the mapping of Prescriptive OpenMP Model parallelism directives to parallelism in the hardware. The table below shows which work-sharing directives use OpenMP parallelism. The model is prescriptive; for example, the `teams` directive yields parallelism for GPUs, but no parallelism for CPUs. This means the programmer must write different directives for each device.

Directive	CPU	GPU
`teams`	Sequential	CUDA *thread blocks*
`parallel`	CPU *threads*	CUDA *threads*
`simd`	Vector Instructions	Sequential
Directive	**Workshare**	
`distribute`	teams parallelism	
`for/do`	thread Parallelism	

teams of processors, for example, *thread blocks* in the GPU. The `parallel` construct executes across the processors or threads in a single team. The `simd` construct controls vector or SIMD parallelism. Several current compilers including ours map these constructs as shown in Table 1. To leverage the massive parallelism of a GPU with this mapping, a programmer should use `teams` and `parallel`, while mapping to a CPU requires `parallel` and `simd` constructs. An OpenMP compiler, however, may choose to use a different mapping, for instance, using `simd` for thread parallelism and `parallel` for warp parallelism on a GPU. This means a programmer may have to use different directives based not only on the target system, but on the compiler implementation. This makes it difficult to achieve performance portability with a prescriptive programming model.

An example of the portability and productivity problem is when a programmer needs to use different directives for different targets. One way to do this is with a preprocessor to select different directives for each target. Figure 1 shows a snippet extracted from the *553.pclvrleaf* program in the SPEC ACCEL benchmark suite [17]. Here the programmer has written each directive set twice for different settings of the `SPEC_USE_INNER_SIMD` macro. When the macro is defined, the outer loops are parallelized across teams and threads, while the inner loops are vectorized with the `simd` directive, as would be appropriate for a CPU architecture. For a GPU, it is usually better to collapse the loops, to ensure enough parallelism in all hardware dimensions, as when the macro is not defined.

There are 892 uses of this macro in this benchmark suite. Other directives could be optimal depending on the target or implementation or loop limits, but would require even more preprocessor macros. Repeating many of the same directives or clauses in different macro paths increases development and maintenance costs. The use of macros does not allow for optimized code for both CPU and GPU, so you can't create a single program with optimized code for both CPU and GPU. This further demonstrates portability and productivity costs, as well as functionality limitations.

```fortran
!$omp target teams
#ifdef SPEC_USE_INNER_SIMD
!$omp distribute parallel do private(j,k)
#else
!$omp distribute parallel do simd collapse(2)
#endif
DO k=y_min,y_max
 #ifdef SPEC_USE_INNER_SIMD
 !$omp simd
 #endif
 DO j=x_min,x_max+1
  vol_flux_x(j,k)= 0.25_8*dt*xarea(j,k)*(xvel0(j,k)&
   + xvel0(j,k+1)+xvel1(j,k)+xvel1(j,k+1))
 ENDDO
ENDDO
#ifdef SPEC_USE_INNER_SIMD
!$omp distribute parallel do private(j,k)
#else
!$omp distribute parallel do simd collapse(2)
#endif
DO k=y_min,y_max+1
 #ifdef SPEC_USE_INNER_SIMD
 !$omp simd
 #endif
 DO j=x_min,x_max
  vol_flux_y(j,k)= 0.25_8*dt*yarea(j,k)*(yvel0(j,k)&
  + yvel0(j+1,k)+yvel1(j,k)+yvel1(j+1,k))
 ENDDO
ENDDO
!$omp end target teams
```

Figure 1. A code snippet from the *553.pclvrleaf* program in the SPEC ACCEL. The first problem is that in order to get the best performance from different compilers, it may be necessary to write new directives under a new macro. The second problem is that a single binary cannot include performant code for CPU and GPU. In the end, each device requires different binaries, and each compiler requires different macros.

With the intention of allowing a single executable to be optimized for each target, OpenMP 5.0 introduced the `metadirective`, giving programmers the ability to write separate directives for each target device in the same source without relying on preprocessing. We think that it addresses the functionality issues that are missing when using macros, but it has the same productivity costs since the programmer is still responsible for writing different directives for each device and implementation.

3.2 Performance Problems of Prescriptive OpenMP Model on GPU

A major performance challenge facing OpenMP implementations for GPUs is the *fork-join* mechanism of the `parallel` construct. OpenMP defines that when a program encounters a `parallel`, the threads will be activated. However, once a GPU kernel is launched, all GPU threads are already active. The user can dynamically select the number of threads, can set a different number of threads to each `parallel` inside the same `target` construct which generates a GPU kernel.

It is also possible to write special code for each thread using OpenMP APIs. The OpenMP model also requires serial execution of the *teams region*, that is, code in a `teams` region but not in `parallel` regions.

The native execution model for a GPU is SPMD. If a compiler can convert the program to use an SPMD model, it will run much more efficiently. However, many programs written in the Prescriptive OpenMP Model cannot be automatically optimized to use SPMD.

3.2.1 Implementing Fork-Join.

In this section, we describe various methods of implementing *fork-join* on the GPU that have been presented or implemented.

Outlining [8]: The most common method to implement the `parallel` construct on CPUs is to outline the construct and have the OpenMP runtime fork threads to run the construct. However, GPUs cannot actually fork threads. Instead, the runtime parks all GPU threads but the primary thread at a synchronization point. The primary thread executes the *teams region*. When it encounters a `parallel` construct, it wakes the other threads to execute the outlined region of `parallel` construct. One hindrance is that GPU threads cannot share local (stack) data. Therefore, shared data owned by the primary thread must be written to global or shared memory, which can be costly.

State-machine[2, 3]: Another method is to use a state-machine. At the start, the primary thread starts running the *teams region* while the other threads wait for the primary thread to determine their next state. When the next state designates a `parallel` region, all threads become active and share the work until the `parallel` region completes. At that point, all threads except the primary thread again go into a wait state. This technique suffers performance loss when encountering calls to functions that potentially contain `parallel` code.

Dynamic parallelism[3, 29]: In this method, the `teams` region starts with one primary thread. When the thread encounters a `parallel` construct, it launches a new kernel using CUDA dynamic parallelism [24], where the body of the construct has been outlined to a new kernel. Launching kernels using CUDA dynamic parallelism comes with significant overhead[12, 31]; this method is not used by any current compiler.

Efficient fork-join [16]: The implementation designates one warp as the primary-warp, and makes it responsible to execute *teams regions*. When the primary-warp sees a parallel region, it wakes the other warps to execute the parallel region. This model has better register usage than the state-machine[2, 3], however it adds a lot of synchronization. As the primary and worker threads are completely diverged, data flow analysis and optimization from the *teams region* into the `parallel` construct is defeated. All shared data written by the primary thread must be stored in GPU global or shared memory, as with outlining.

3.2.2 Optimized Code Generation.

The fork-join methods described above perform slowly. Here, we briefly describe some studies that try to generate SPMD code directly.

Broadcasting[10, 12, 20]: This method works when the entire region can be run in SPMD mode, all threads active all the time. The compiler generates conditionals so that all threads except thread zero jump around the *teams region* code. Any stores to shared variables in this region are broadcast by being saved in *CUDA shared memory* or device global memory. At the end of the *teams region*, all threads meet at a synchronization point. This method may introduce a lot of overhead due to synchronization, however it can often be optimized away. The most important advantage of this method is that it supports more classical compiler optimizations.

XL compiler [13, 30]: In this work, the IBM XL compilers [14] leverage interprocedural compiler analysis to determine whether running the code as SPMD is safe. But this optimization is only enabled if the *teams region* can be run redundantly by all threads. For this reason, it can be activated in very few places.

Tregions[9]: This method is designed for the Clang [22] project. It also tries to run the program in SPMD mode if it is safe. It postpones the decision of running as SPMD from the Clang frontend to the LLVM middle-end. It carries complex execution logic from the generated code to the runtime.

3.2.3 Challenging Scenarios to Optimize Fork-join.

***fork-join* in a function:** When a `parallel` construct is in a function called within a `target` region, it requires a dynamic *fork-join*. This case is really difficult to optimize when the compiler has no visibility of the function body while generating code for the `target` region. A compiler may be able to optimize this case with link time optimization.

Parallelism-inhibiting directives and API calls: OpenMP includes many synchronization primitives, such as `master`, `primary`, `critical`, `masked`, `single`, etc., and various runtime API calls. These hinder scalability significantly when they are used. In addition, it requires complex runtime support to implement them on GPUs. Many programs can be optimized if these features are not used, but this may not be obvious at compile-time due to function calls.

4 Our Descriptive OpenMP Model

In this section, we present our Descriptive OpenMP model. It has two advantages. First, it overcomes the performance, portability, and productivity problems we mentioned in Section 3.1. We give the compiler the freedom (or responsibility) to automatically select how to exploit parallelism in most cases, requiring fewer directives and clauses, and resulting in more performance-portable code. Second, it allows for the optimization of the *fork-join* avoiding the overhead that we described in Section 3.2.

We leverage OpenMP's `loop` construct by modifying it slightly. Our modifications are as follows:

1. **Automatic Parallelism Mapping and Creation:** The compiler can automatically map the loops to the parallelism created by `teams` or `parallel` constructs. It can also create `teams` or `parallel` parallelism if none exist.
2. **Allow `atomic` directive:** The `atomic` construct is allowed inside a `loop` region.
3. **Restrict `parallel` directive:** Because a `parallel` construct in a `loop` region would inhibit our fork-join optimization, we currently disallow this case.

The first modification is the most crucial part of our work. As mentioned in Section 3.1, productivity is significantly decreased when the user must manually choose how to exploit parallelism for each device. By shifting this burden to the compiler, we address the current performance portability problem and increase productivity by reducing the number of directives and clauses. We discuss this in detail in Section 5.

Our second modification is an extension to allow the `atomic` directive in `loop` regions. Atomics are widely used in parallel programs and do not inhibit or limit the concurrent execution model. We could find no reason to disallow them.

Our last modification is to restrict the `parallel` directive. We mentioned overhead in Section 3.2 when this directive is challenging to be optimized. Thus we currently eliminate this case to have a model that the compiler can optimize well.

The powerful part of our model is that while the compiler automatically selects parallelism, the programmer can modify or improve this selection by collaborating with the compiler using the `bind` clause as follows:

- `omp loop bind(teams)`: Map this loop to teams or teams and thread parallelism.
- `omp loop bind(parallel)`: Map this loop to thread parallelism.
- `omp loop bind(thread)`: Map this loop to a single thread for sequential execution.
- `omp loop bind(teams,parallel)`: Map this loop to teams and thread parallelism.

5 Automatic Parallelism Mapping

We propose automatic parallelism selection for each target device as a compiler technique to increase portability and productivity, without requiring the user to write different directives for each device. An early study [32] describes some of how to do automatic parallelism mapping only for the GPU, but not for CPUs and OpenMP. The compiler will adjust the parallelism for each device, and can even generate a single binary that will run on any supported device available at runtime.

5.1 Finding Parallel Loops

Every loop annotated by a `loop` construct is, by definition, a parallel loop. The `loop` construct implies that the loop iterations can be run concurrently. The compiler can safely parallelize, vectorize, or transform the loop. OpenMP's other work-sharing constructs, such as `for/do` or `distribute` do not imply truly parallel iterations. For these reasons the `loop` directive has quite a few advantages.

To discover more parallelism, our compiler also does dependence analysis[33] on each loop in the `target` region. That's not to say that the compiler depends on automatic parallelization, but by default it uses parallelization to expose more parallelism in addition to what the user has identified. Most of the remaining compiler work is to map the parallel loops to the available hardware parallelism.

5.2 Parallelism Mapping for GPUs

The compiler schedules parallel loops by essentially mapping OpenMP *teams* to *thread blocks*, and OpenMP *threads* to *threads*:

Team parallel loops: The compiler assigns the outermost loops to *thread blocks*. If the outermost loop has to be run serially, the outermost parallel loop is chosen. The user can specify `loop bind(teams)` to override the compiler selection.

Thread parallel loops: Optimal memory access in the GPU is coalesced access, that is, adjacent threads accessing consecutive memory addresses. Therefore, assigning thread parallelism is a crucial task and directly affects performance. In order for the compiler to find this, it needs to know the memory references in the loops and the induction variables of the loops. Then it classifies the memory reference access patterns as stride-1, non-stride-1, independent, or dynamic for each loop. The aim is to minimize non-stride-1 access and maximize stride-1 access across threads. Access frequency is also important, so it gives more weight to each references in nested loops. Eventually, it maps the x dimension of *thread* parallelism to one of the loops, usually the loop with the most stride-1 accesses. As a minor note, when the trip count of the loop is smaller than *warp size*, the compiler may run these loops serially to avoid underutilization of lanes. In addition, the user can specify `loop bind(parallel)` to override the compiler selection.

5.3 Parallelism Mapping for CPU

The compiler schedules parallel loops by essentially mapping OpenMP *threads* to *CPU threads*, and OpenMP *simd* parallelism to *SIMD instructions*:

Thread parallel loops: The compiler usually assigns the outermost loops to threads to achieve coarse-grain parallelism. If the outermost loop is not parallel or is marked as `bind(thread)`, it chooses the outermost available parallel loops.

SIMD parallel loops: Each loop marked with `loop` construct is a potential for SIMD parallelism. The compiler assigns them to vector instructions.

5.4 An Example

Figure 2 shows an example of automatic parallelism selection. Here, the user specifies the parallel loops in a target-agnostic way. For example, the user marks the outer loops with `loop` and the compiler finds that the inner loops are also parallel. Now we explain how the compiler automatically creates and selects parallelism differently for each target device.

GPU Parallelism: Figure 3 shows parallelism mapping for GPU. The most critical mapping is the *x* dimension of *threads* to achieve coalesced memory access. As we described above, the compiler performs stride-1 and non-stride-1 memory access analysis among the parallel loops. In this example both *loop-j* have the most stride-1 access so the compiler maps them to *thread* parallelism.

The user can expect a *barrier* after *thread* parallelism. However, *loop-j* has no external dependence here. Therefore our compiler eliminates redundant *barrier*.

GPUs have more available parallelism so the compiler maps the remaining outer loops to *thread blocks*. Here, it maps each *loop-i* to the *x* dimension of the *thread block*. If there are more loops, it can leverage the other two dimensions of the *thread block*. Alternatively, the user or the compiler can collapse the tightly-nested loops.

An important issue when mapping *thread block* parallelism is dependences between *loop-i* loops. If there are dependences, the compiler must synchronize the *thread blocks*, but this is not efficient doing on a GPU. In such a case, the compiler may create multiple kernels, one for each outer loop, effectively acting as a *barrier*. There is no such situation in this code, so a single GPU kernel is sufficient.

Another problem compilers face is that there is a limit for the number of *threads* and *thread blocks* and the loop trip count may be bigger than the limits. To handle these cases, the compiler *strip-mines* the loops into an outer, sequential *strip* loop and an inner parallel *element* loop. The strip size naturally maps to a corresponding dimension of *blockDim* or *gridDim*.

For performance, the compiler may optimize redundant parallel memory accesses across threads such as to *yvel0* and *yvel1* in this code. In the past, we have allocated space in the *CUDA shared memory*, applying loop fission, filling the the space in the first loop and using it in the second loop. But *CUDA shared memory* resides in the L1 cache, so allocating it means sacrificing space from the L1 cache, and the L1 cache is fairly small, 128KB for NVIDIA Ampere GPUs. Furthermore, this requires additional thread-level synchronization. Therefore we do not do auto-caching in our compiler, we leave it to the hardware for now. If using *CUDA shared memory* is necessary, the user can define *team private* variables and the compiler will place them in *CUDA shared memory* if they are small enough.

To increase instruction-level parallelism (ILP), the compiler can unroll the loops. In this example, our compiler

```
!$omp target teams
!$omp loop
DO k=y_min,y_max
 DO j=x_min,x_max+1
  vol_flux_x(j,k)= 0.25_8*dt*xarea(j,k)*(xvel0(j,k)&
   + xvel0(j,k+1)+xvel1(j,k)+xvel1(j,k+1))
 ENDDO
ENDDO
!$omp loop
DO k=y_min,y_max+1
 DO j=x_min,x_max
  vol_flux_y(j,k)= 0.25_8*dt*yarea(j,k)*(yvel0(j,k)&
   + yvel0(j+1,k)+yvel1(j,k)+yvel1(j+1,k))
 ENDDO
ENDDO
!$omp end target teams
```

Figure 2. Our proposal OpenMP program. The programmer indicates what to parallelize rather than how to parallelize. Our compiler achieves exactly the same parallelism as in Figure 1, with fewer directives and no target-specific directives in single binary.

```
DO tK=y_min,y_max,gridDim.x                -> Cyclic Scheduling
 DO k=tK,min(tK+gridDim.x-1,y_max)         -> Map to blockIdx.x
  DO tJ=x_min,x_max+1,blockDim.x-> Unroll, Cyclic Scheduling
   DO j=tJ,min(tJ+blockDim.x-1,x_max+1)-> Map to threadIdx.x
    ! Loop Body !
DO tK=y_min,y_max+1,gridDim.x              -> Cyclic Scheduling
 DO k=tK,min(tK+gridDim.x-1,y_max+1)       -> Map to blockIdx.x
  DO tJ=x_min,x_max,blockDim.x   -> Unroll, Cyclic Scheduling
   DO j=tJ,min(tJ+blockDim.x-1,x_max)   -> Map to threadIdx.x
    ! Loop Body !
```

Figure 3. GPU Parallelism mapping of code in Figure 2

```
W = (y_max + NumThreads) / NumThreads
DO tK=y_min,y_max,W                            -> Map to Threads
 DO k=tK,min(tK+W-1,y_max)                   -> Block Scheduling
  DO j=x_min,x_max                                -> Vectorize
   ! Loop Body !
W = (y_max+1 + NumThreads) / NumThreads
DO tK=y_min,y_max+1,W                          -> Map to Threads
 DO k=tK,min(tK+W-1,y_max+1)                 -> Block Scheduling
  DO j=x_min,x_max                                -> Vectorize
  ! Loop Body !
```

Figure 4. CPU Parallelism mapping of code in Figure 2

unrolls *loop-tJ* so it does not change the memory access pattern. As a side note unrolling comes with a trade-off. While it increases ILP, it may reduce occupancy. It increases the number of registers used in the code, and the register file size in SM is only 64K for NVIDIA Ampere GPUs. So the compilers need a heuristic to make unrolling decisions. This is outside of this article, but unrolling does not affect our parallelism selection.

CPU Parallelism: A CPU typically consists of multiple cores (currently up to 80). Though this number is small compared to a GPU, CPU cores are much more powerful, typically include SIMD units, and have much larger caches. Therefore, the compiler usually maps the outermost loops to threads to achieve coarse-grain parallelism. Here it parallelizes both of the *loop-k* using *threads*. It also applies *static block* loop scheduling to achieve better cache locality. We show CPU parallelism mapping in Figure 4.

The compiler can vectorize the loops here. As we mentioned earlier, all the loops here are data-independent, so they are all vectorizable. Our compiler vectorizes the *loop-j*. For performance, compilers could do outer loop vectorization. Although our compiler does not currently implement this, the `omp loop` construct is very suitable for supporting outer loop vectorization.

6 Code Generation

In this section, we explain the code generation strategy to implement OpenMP. We explain it in two parts: the first for the Prescriptive OpenMP Model, and the second for our proposed Descriptive OpenMP Model.

6.1 CPU Code Generation

6.1.1 Prescriptive OpenMP Model. We use the outlining technique as most compilers do. Our compiler extracts the body of the `target`, `teams`, and `parallel` constructs into outlined functions, replacing the body with a runtime call that will invoke the function with the appropriate threads. We translate loop scheduling constructs such as `for`, `do`, and `distribute` into runtime calls for work distribution. In the runtime, we launch a single team for the `teams` construct, and multiple threads for the `parallel` construct. Unlike the GPU with its thread blocks, threads are the coarsest unit of parallelism on the CPU. We know that another idea would be to bind `teams` to NUMA nodes, but this can slow down code that does not use `teams`. Therefore, we only use thread parallelism. Our OpenMP runtime has the same interface as the commonly-used LLVM OpenMP runtime, so our code generation is the same as other compilers using that library.

6.1.2 Descriptive OpenMP Model. Code generation for the Descriptive OpenMP Model is similar to that in the Prescriptive OpenMP Model for CPU targets. The key difference is we do not generate extra code for team parallelism, we just generate code for thread parallelism. Once the compiler finds the loops to parallelize, it uses thread parallelism across those iterations. Here we outline loop bodies, then we fork the threads and execute the body with the threads. If the inner loops are suitable for vectorization, the compiler will generate the vector code for them.

6.2 GPU Code Generation

6.2.1 Prescriptive OpenMP Model. To support the Prescriptive OpenMP Model, we use the outlining method we explained in Section 3.2.1. As mentioned, it is not possible to use CPU-style outlining directly for GPU targets so we have adapted this technique for the GPU. When the program encounters `teams`, we launch teams and threads and put threads to sleep. When the program sees `parallel` it executes the outlined parallel function.

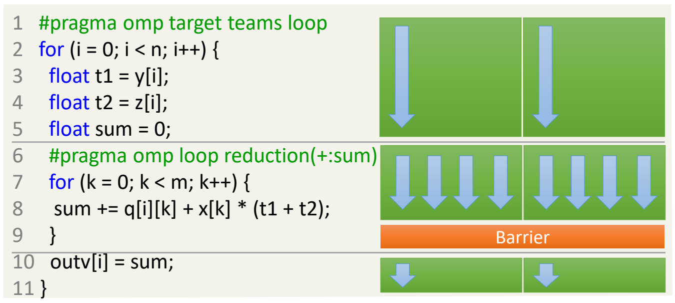
6.2.2 Descriptive OpenMP Model. GPU code generation starts with a `target` construct using a similar outlining mechanism, except the outlined function is a GPU kernel. We do not do any other outlining. We implement the optimized broadcasting method we explained in Section 3.2.1. As mentioned in Section 3.2.3, supporting fork-join parallelism is challenging on a GPU. Our model simply disallows this, so the generated code can run fully in SPMD mode.

Code in the *teams region* must be executed only by the primary thread (thread zero) of the team, so it must be protected to avoid conflicts or redundant execution. The compiler generates a *neutering* conditional branch around the *teams region*, so all threads except thread zero branch around that code (are neutered) and wait at a synchronization point. Any results computed in the *teams region* that need to be seen by the other threads in the team are broadcast to all threads using *CUDA shared memory* or, if too large, using device memory. The primary thread stores its results in the appropriate memory, then joins the other threads at the synchronization point. All threads then load the shared data as appropriate and continue.

To avoid unnecessary synchronization for the *teams region* at every such construct, the compiler does several optimizations. It delays insertion of the *neutering* conditional branch until after data flow analysis and optimization. This allows the compiler to propagate constants and perform other optimizations between the *teams region* and the body of the construct. The compiler also detects when the code in the *teams region* can be safely executed redundantly by all threads in the team, avoiding data sharing and synchronization altogether.

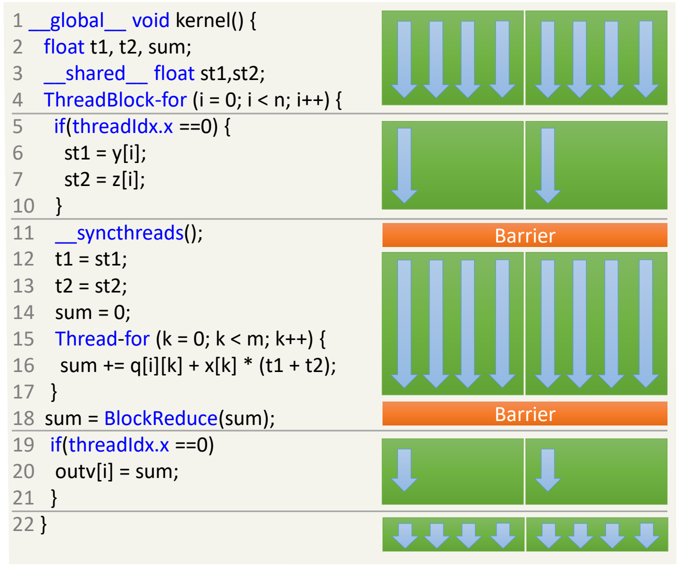
Figure 5 shows code generation example for the OpenMP code in Figure 5a. First, the compiler automatically selects the parallelism for the loops using the technique we proposed in Section 5. It maps *loop-i* to *thread blocks* and *loop-j* to the *thread* parallelism. Here lines 2-5 and 10-11 correspond to the *teams region*, and as mentioned above, a single thread should execute them. To clarify this better, we illustrate this execution model next to the code. There are two GPU *thread blocks*, their primary threads run these lines.

A naïve compiler can generate *neutering* branches to surround the *teams region*. If there are stores to the variable inside, their final value should be visible at the time other threads are activated. So the compiler broadcasts them. Figure 5b shows an example of this scenarios. Here the compiler generates the conditionals on lines 5 and 19, allocating *st1, st2* from *CUDA shared memory*. While the primary thread is running these branches, other threads wait at the barrier (*__syncthreads*). It then broadcasts the written values to the



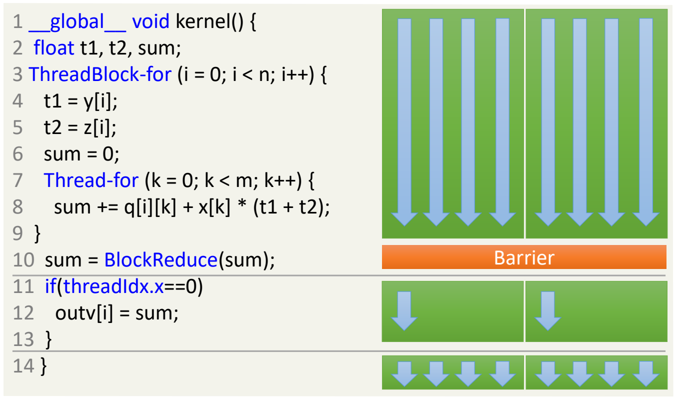
```
1  #pragma omp target teams loop
2  for (i = 0; i < n; i++) {
3   float t1 = y[i];
4   float t2 = z[i];
5   float sum = 0;
6   #pragma omp loop reduction(+:sum)
7   for (k = 0; k < m; k++) {
8    sum += q[i][k] + x[k] * (t1 + t2);
9   }
10  outv[i] = sum;
11 }
```

(a) Example OpenMP Code



```
1 __global__ void kernel() {
2  float t1, t2, sum;
3  __shared__ float st1,st2;
4  ThreadBlock-for (i = 0; i < n; i++) {
5   if(threadIdx.x ==0) {
6    st1 = y[i];
7    st2 = z[i];
10  }
11  __syncthreads();
12  t1 = st1;
13  t2 = st2;
14  sum = 0;
15  Thread-for (k = 0; k < m; k++) {
16   sum += q[i][k] + x[k] * (t1 + t2);
17  }
18 sum = BlockReduce(sum);
19 if(threadIdx.x ==0)
20  outv[i] = sum;
21 }
22 }
```

(b) Conceptually generated GPU code without broadcast optimization



```
1 __global__ void kernel() {
2 float t1, t2, sum;
3 ThreadBlock-for (i = 0; i < n; i++) {
4  t1 = y[i];
5  t2 = z[i];
6  sum = 0;
7  Thread-for (k = 0; k < m; k++) {
8   sum += q[i][k] + x[k] * (t1 + t2);
9 }
10 sum = BlockReduce(sum);
11 if(threadIdx.x==0)
12  outv[i] = sum;
13 }
14 }
```

(c) Conceptually generated GPU code with broadcast optimization

Figure 5. Code Generation Example

registers of other threads, as seen on lines 12-13. This naïve code generation uses extra *CUDA shared memory* and synchronization, which degrades performance. If there are many *teams regions* in the code, the cost increases.

To avoid these extra operations, the compiler analyzes when a *teams region* can be safely run redundantly. Figure 5c shows the generated code with this optimization. As seen here, the first *teams region* is run redundantly, all threads run this region. The compiler proves that it is safe to do so because all threads load the same data from memory and write it to their own registers. The compiler maintains the conditional for the second *teams region*. Here only the primary thread has the reduction value. Finally, the compiler eliminates extra synchronization and extra memory usage and generates GPU-optimized code.

7 Evaluation

This section describes the experimental platform and the test programs, and compares the performance of those programs with our model against production compilers on the same hardware.

7.1 Experimental Platform

Our experiments are done on two different systems to compare against different compilers. Each has an NVIDIA Tesla V100 GPU with 80 SMs, 5120 CUDA cores, compute capability 7.0, and 16GB of device memory. The systems are as follows:

1. 2 socket 20-core 2.40GHz Intel Skylake Gold 6148 CPU, 256GB of main memory, running CentOS 8.2. A CUDA 11.0 driver and toolkit was used.
2. 2 socket 20-core 3.80GHz IBM POWER9 CPU with 4 threads, 128GB of main memory, running Red Hat Pegas 7.6. A CUDA 10.2 driver and toolkit was used.

7.1.1 Compilers and Environment Variables. To understand the quality of our implementation we also compare our work in NVIDIA’s HPC compiler against three production compilers that support the same hardware, namely GCC, Clang, and the IBM XL compilers. We know that the Cray compiler supports OpenMP offload well, but we did not include it in our results because we did not have access to a Cray system. Table 2 shows information about the compilers and the flags we used. Clang only supports C and C++. To get comparable results from all compilers, we set the following OpenMP environment variables:

- `OMP_NUM_THREADS` to 80 (Intel) or 40 (IBM).
- `OMP_THREAD_LIMIT` to 80 (Intel) or 40 (IBM).
- `OMP_PROC_BIND` to true.

7.1.2 Benchmarks. We use the 15 OpenMP programs of the SPEC ACCEL benchmark suite version 1.3 [17]. The SPEC ACCEL suite was developed by the Standard Performance Evaluation Corporation (SPEC) High Performance Group (HPG) to measure the performance of compute intensive parallel applications on accelerated systems. It is comprised of seven C, six Fortran, and two combined C and Fortran application kernels.

7.1.3 Porting SPEC ACCEL Benchmark Suite to the Descriptive OpenMP Model. We ported the SPEC ACCEL benchmarks by removing the preprocessor macros and

Table 2. Compilers and flags used.

Compiler	GPU Flags	CPU Flags
NVIDIA HPC	-fast -mp=gpu -gpu=fastmath	-fast -mp
IBM XL 16.1.1	-O3 -qoffload -qsmp	-O3 -qsmp
GNU 11.1	-fopenmp -O3 -foffload="-O3 -lm"	-O3 -fopenmp
Clang 11.1	-fopenmp -O3 -fopenmp-targets=nvptx64	-O3 -fopenmp

changing the OpenMP directives to use the `omp loop` directive. No other changes were made to the source code.

To compare the productivity of using our Descriptive OpenMP Model vs. Prescriptive OpenMP Model, we counted the number of appearances of OpenMP parallelism directive keywords in each benchmark: `target`, `teams`, `parallel`, `distribute`, `for`/`do`, `simd`, or `loop`. As discussed earlier, using a preprocessor to make target-specific choices often requires the programmer to repeat the same directive keyword multiple times. Using the OpenMP 5.0 `metadirective` has exactly the same issue. Our descriptive model uses fewer directives. Figure 6 shows the number of OpenMP parallelism directive keywords in SPEC ACCEL benchmark suite, using the original OpenMP and as ported using our descriptive model. Our model needs 60% fewer OpenMP directives.

7.2 Experimental Results

We evaluated the SPEC ACCEL benchmarks by measuring the elapsed time to run the *ref* datasets on each of the two systems. These runs are SPEC *estimates*, as they were not run in the SPEC performance harness. We use NVIDIA's HPC compiler for the baseline to demonstrate that it supports the original OpenMP code, that its performance is competitive with other production compilers, and that the performance of our descriptive OpenMP model is an improvement even against our own compiler. For each system and compiler, we compile and run the program on the multicore CPU, and recompile and run the program to offload the `target` regions to the GPU. For the Descriptive OpenMP Model with NVIDIA's HPC compiler, we compile it once with GPU flags and use the same binary for both the CPU and the GPU, setting the environment variable *OMP_TARGET_OFFLOAD=DISABLED* to run on the CPU. Figure 7 shows speedup results compared to the baseline. In the rest of this section, we discuss the performance in more detail.

7.2.1 Discussion on CPU Executions.

Figure 7a and Figure 7c show the performance on Intel and IBM CPUs, respectively. The most important result here is that while the

[2]Data directive counts: 1 in pilbdc, pswim, psismic, postencil, and pep, 2 in pcs and pcg, 3 in pmniGhost and polbm, 5 in pclvrleaf and psp, 7 in pcsp and pbt, 31 in ppalm.

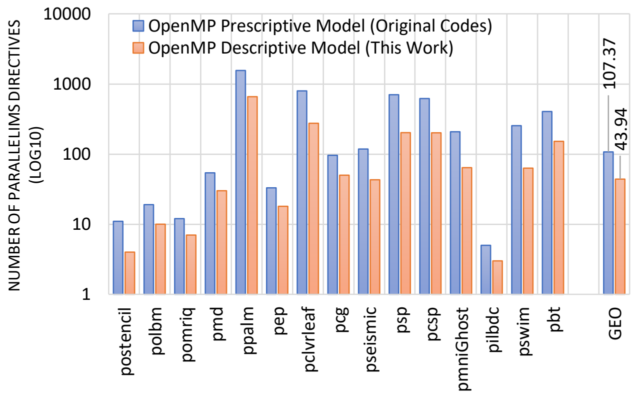


Figure 6. The number of OpenMP parallelism directive keywords in each benchmark. Data directives such as `target data`, `target enter/exit data`, `declare target` are not included. Their numbers are negligible.[2]

model we propose uses fewer directive keywords, it preserves the performance on the CPU. This demonstrates that the OpenMP `loop` construct and a good compiler can be used for higher productivity without performance loss.

The *IBM XL* compiler gives a compilation error on **pomriq**, **ppalm** and **psp**, and a verification error on **polbm** and **pclvrleaf**. The other compilers can compile and run the entire benchmark suite.

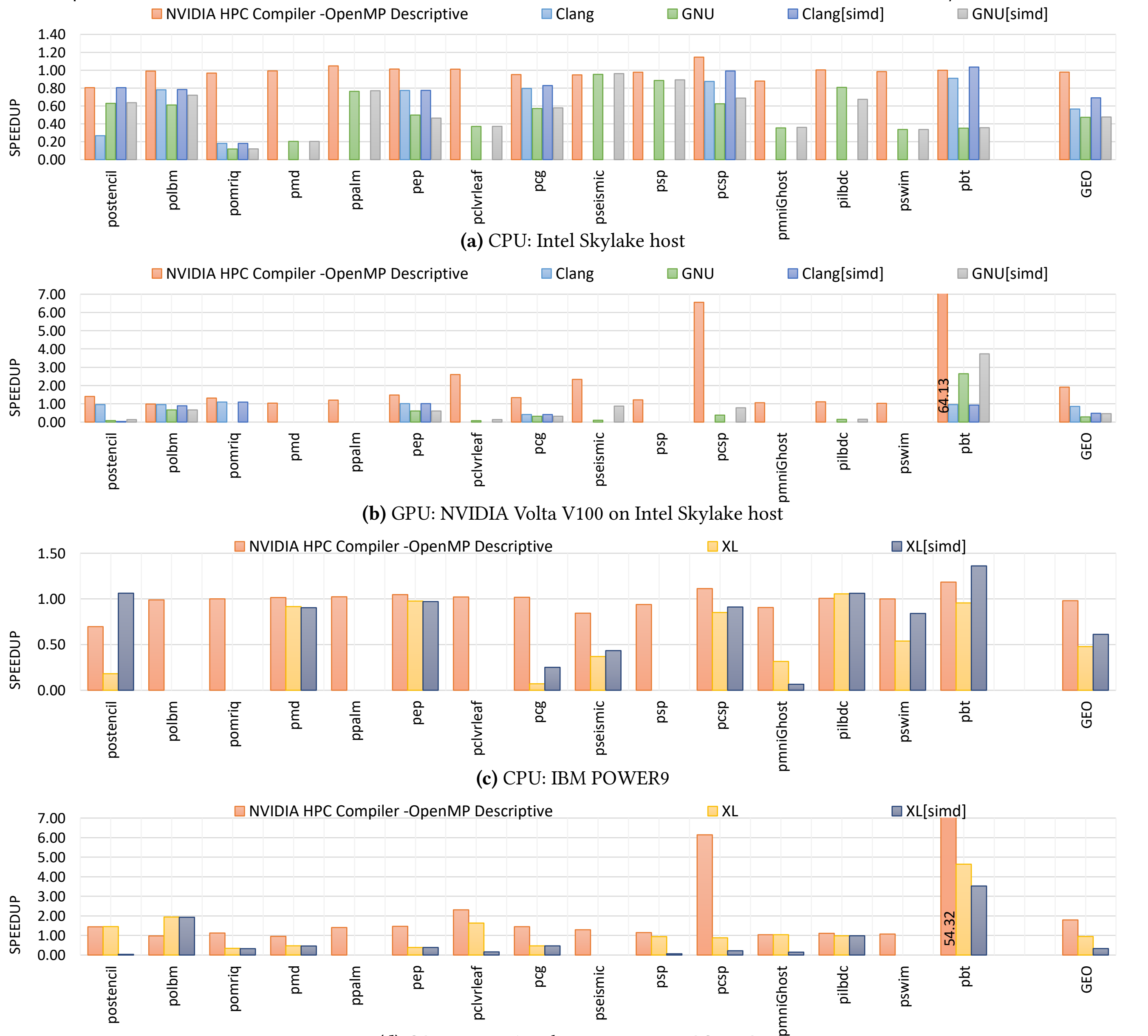
postencil runs slower in the Descriptive OpenMP Model against Prescriptive OpenMP Model. This is because of a bug in NVIDIA's HPC compiler, where the inner loop is not vectorized, which we will address in the near future.

7.2.2 Discussion on GPU Executions.

Figure 7b and Figure 7d show the performance on the GPU with Intel and IBM hosts, respectively. Our descriptive model yields an average of 1.91x and 1.79x speedup, with a maximum of 64x and 54x acceleration on Intel and IBM hosts compared to NVIDIA's HPC compiler.

NVIDIA's HPC compiler can compile every benchmark in both the Prescriptive OpenMP and Descriptive OpenMP models. The *GNU* compiler fails at runtime for **pomriq**, **pmd**, **ppalm**, **psp**, **pmniGhost**, and **pswim**. The *clang* compiler gives a verification error for **pcsp**. The *IBM XL* compiler gives a compilation error for **ppalm**.

postencil, ppalm, pclvrleaf, pep, pcg, pseismic, and **psp:** For these benchmarks, it is important to efficiently parallelize the inner loops to take advantage of the multiple levels of parallelism in the GPU. The Prescriptive OpenMP model does not make good use of inner loop parallelism. In many of these loops, only the outer loop was parallelized. With the rigid Prescriptive OpenMP model, the compiler is limited in exploiting the inner loop parallelism. Descriptive OpenMP Model with NVIDIA's HPC compiler shows its advantage here by detecting parallelism on the inner loops and



(a) CPU: Intel Skylake host

(b) GPU: NVIDIA Volta V100 on Intel Skylake host

(c) CPU: IBM POWER9

(d) GPU: NVIDIA Volta V100 on IBM POWER9

Figure 7. Figure 7a and 7c show the speedup for CPU execution on each host relative to the CPU execution baseline. Figure 7b and 7d show the speedup for GPU execution on each host relative to the GPU execution baseline. The baseline is the original Prescriptive OpenMP benchmarks compiled by NVIDIA’s HPC compiler. In each case, the leftmost bar is NVIDIA’s HPC compiler using `omp loop` with the OpenMP Descriptive Model version of the benchmark. The *Clang[simd], GNU[simd], IBM XL[simd]* bars are compiled with *-DSPEC_USE_INNER_SIMD* to enable `omp simd` usage on the inner loops. The X axis shows the benchmark from the suite. In all cases, higher is better.

efficiently parallelizing them across the threads, resulting in the best performance.

We also tried enabling the `simd` directive on the inner loop with the `SPEC_USE_INNER_SIMD` macro. Indeed, the *GNU* compiler gets a big performance boost, because we think that it uses the `simd` directive for GPU thread-level parallelism. The other compilers suffered catastrophic slowdowns, further demonstrating the portability and productivity tradeoff.

polbm: This is the only benchmark where our work does not outperform all other compilers; the *IBM XL* compiler performs better. It assumes some data is read-only and generates a read-only-caching (texture load) instruction to load the data. We believe that this optimization is unsafe unless the compiler can guarantee that there are no aliasing issues between pointers in the kernel.

pomriq: The program contains a single kernel where only the outer loop needs to be parallelized. This code shows an example where loop tuning was required. NVIDIA's Descriptive OpenMP Model compiler wanted to map thread parallelism to the inner loop, but doing so degraded performance. We added a `bind(thread)` clause to this loop so our compiler run this loop serially and moved the thread parallelism on the outer loop automatically.

pcsp, and **pbt:** For these benchmarks, our work outperforms the other compilers significantly. The big performance difference is that the thread parallelism that comes with `parallel for/do` is on the wrong loop in the original benchmark source. The memory pattern is optimized for consecutive memory accesses in a single thread, which is the right choice for CPUs. However, this is suboptimal for a GPU, which wants consecutive memory accesses across adjacent threads, and eventually results in a massive performance loss. This demonstrates that the OpenMP programming model is prescriptive and strict, and the compiler has no freedom to optimize the parallelism.

7.2.3 Evaluation Takeaways. Descriptive OpenMP with NVIDIA's HPC compiler gives the best performance for GPUs compared to all other compilers and yields 90% and 80% speedup on on Intel and IBM POWER hosts, respectively, compared even to our own compiler. The most important reason is that it can efficiently parallelize or move appropriate parallelism to the inner loops. Moreover, the performance proves to be portable and gives equivalent performance on the CPU compared to code written with prescriptive directives.

8 Conclusion

Accelerated computing has become common and the newest pre-exascale- and exascale-class supercomputers are all accelerated systems. One challenge for parallel programming in general, and with accelerated systems in particular, is to address the performance / portability / productivity tradeoff. When using OpenMP, or any other parallel programming language, parallelism must be both identified and mapped to the hardware parallelism. In classical Prescriptive OpenMP, these two steps are conflated. When writing an OpenMP program that will be run on both accelerated and non-accelerated systems, the programmer must often write different directives for the two types of systems.

We propose modifying and using the `loop` construct in the well-accepted OpenMP API to increase all three of performance, portability, and productivity. The most important modification in our model is to change the `loop` construct to be more *descriptive*, and to allow or require the compiler to choose how to map the program parallelism to the target system in most instances. This increases portability, by allowing a compiler to choose different levels of parallelism for the same loop on different target systems. This increases productivity, by obviating the need for the programmer to think about the target system details, and by reducing the number of directives and clauses required. This also affects performance. If the compiler makes bad decisions, then the programmer may want or need to insert directives or clauses to modify those decisions, perhaps even in some cases as many directives or clauses as the Prescriptive OpenMP model would require. If the compiler makes good decisions, the performance should match that of the best Prescriptive OpenMP version of that program.

Our experiments using the SPEC ACCEL OpenMP benchmarks show that the performance of programs written with our proposed model do in fact match that of Prescriptive OpenMP when compiled for and run on multicore CPUs. The experiments also show that the performance of programs written with our proposed model exceed that of Prescriptive OpenMP when compiled for and run on GPUs. The reason for the performance improvement is that our proposed descriptive model allows the compiler to generate SPMD code for the offloaded code, which is the native execution model for a GPU, and which has significantly less runtime overhead. We also counted the number of OpenMP directives and clauses for the Prescriptive OpenMP and our descriptive model, and the descriptive `loop` model uses 60% fewer directives, with no `metadirective` or preprocessor code for target-specific code generation. We realize that these results are valid only for this particular benchmark suite. Other suites may require fine-tuning with other OpenMP directives, or the algorithm may need to be restructured with other directives, for example, tasking. However, if the programmer is already writing highly parallel, scalable code for accelerators, we think it is likely there will be little need for any other directives. We claim this demonstrates that our model improves all three of performance, portability, and productivity for OpenMP programming of accelerated systems.

Even though the Prescriptive OpenMP Model does not quite fit the GPU execution model, we think it can sometimes be optimized as efficiently as the Descriptive OpenMP Model. A compiler should be able to identify when the code avoids unsafe patterns and then compile it as SPMD. But if the programmer uses any unsafe directive or API call, the compiler will fall back to the non-SPMD model, resulting in a very high performance loss. The level of these optimizations may vary from compiler to compiler, so performance may not be consistent across compilers. Therefore, we think that the Descriptive OpenMP Model is a performance safe model for GPUs.

Acknowledgments

This paper's authors would like to thank the following colleagues: Simone Atzeni, Evgeny Baskakov, Sebastien Deldon, Zahra Khatami, Mark LeAir, Razvan Lupusoru, Pallavi Mathew, Mahdi Soltan Mohammadi, Adam Smith, Annemarie Southwell, Xiaonan Tian, Rajan Walia, and Zhen Wang. This research is done partly under a non-recurring engineering contract between NERSC and NVIDIA.[6].

References

[1] Samuel F. Antão, Alexey Bataev, Arpith C. Jacob, Gheorghe-Teodor Bercea, Alexandre E. Eichenberger, Georgios Rokos, Matt Martineau, Tian Jin, Guray Ozen, Zehra Sura, Tong Chen, Hyojin Sung, Carlo Bertolli, and Kevin O'Brien. 2016. Offloading Support for OpenMP in Clang and LLVM. In *Third Workshop on the LLVM Compiler Infrastructure in HPC, LLVM-HPC@SC 2016, Salt Lake City, UT, USA, November 14, 2016*. 1–11. https://doi.org/10.1109/LLVM-HPC.2016.006

[2] Carlo Bertolli, Samuel Antão, Gheorghe-Teodor Bercea, Arpith C. Jacob, Alexandre E. Eichenberger, Tong Chen, Zehra Sura, Hyojin Sung, Georgios Rokos, David Appelhans, and Kevin O'Brien. 2015. Integrating GPU support for OpenMP offloading directives into Clang. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC, LLVM 2015, Austin, Texas, USA, November 15, 2015*. 5:1–5:11. https://doi.org/10.1145/2833157.2833161

[3] Carlo Bertolli, Samuel Antão, Alexandre E. Eichenberger, Kevin O'Brien, Zehra Sura, Arpith C. Jacob, Tong Chen, and Olivier Sallenave. 2014. Coordinating GPU threads for OpenMP 4.0 in LLVM. In *Proceedings of the 2014 LLVM Compiler Infrastructure in HPC, LLVM 2014, New Orleans, LA, USA, November 17, 2014*, Hal Finkel and Jeff R. Hammond (Eds.). IEEE Computer Society, 12–21. https://doi.org/10.1109/LLVM-HPC.2014.10

[4] David R. Butenhof. 1997. *Programming with POSIX Threads*. Addison-Wesley.

[5] Christopher Daley, Hadia Ahmed, Samuel Williams, and Nicholas Wright. 2020. A Case Study of Porting HPGMG from CUDA to OpenMP Target Offload. In *OpenMP: Portable Multi-Level Parallelism on Modern Systems*, Kent Milfeld, Bronis R. de Supinski, Lars Koesterke, and Jannis Klinkenberg (Eds.). Springer International Publishing, Cham, 37–51.

[6] Christopher S. Daley, Annemarie Southwell, Rahulkumar Gayatri, Scott Biersdorfff, Craig Toepfer, Güray Özen, and Nicholas J. Wright. 2021. Non-recurring engineering (NRE) best practices: a case study with the NERSC/NVIDIA OpenMP contract. In *SC '21: The International Conference for High Performance Computing, Networking, Storage and Analysis, St. Louis, Missouri, USA, November 14 - 19, 2021*, Bronis R. de Supinski, Mary W. Hall, and Todd Gamblin (Eds.). ACM, 31:1–31:14. https://doi.org/10.1145/3458817.3476213

[7] Joshua Hoke Davis, Christopher S. Daley, Swaroop Pophale, Thomas Huber, Sunita Chandrasekaran, and Nicholas J. Wright. 2020. Performance Assessment of OpenMP Compilers Targeting NVIDIA V100 GPUs. In *Accelerator Programming Using Directives - 7th International Workshop, WACCPD 2020, Virtual Event, November 20, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12655)*, Sridutt Bhalachandra, Sandra Wienke, Sunita Chandrasekaran, and Guido Juckeland (Eds.). Springer, 25–44. https://doi.org/10.1007/978-3-030-74224-9_2

[8] Bronis R. de Supinski, Thomas R. W. Scogland, Alejandro Duran, Michael Klemm, Sergi Mateo Bellido, Stephen L. Olivier, Christian Terboven, and Timothy G. Mattson. 2018. The Ongoing Evolution of OpenMP. *Proc. IEEE* 106, 11 (2018), 2004–2019. https://doi.org/10.1109/JPROC.2018.2853600

[9] Johannes Doerfert, Jose Manuel Monsalve Diaz, and Hal Finkel. 2019. The TRegion Interface and Compiler Optimizations for OpenMP Target Regions. In *OpenMP: Conquering the Full Hardware Spectrum - 15th International Workshop on OpenMP, IWOMP 2019, Auckland, New Zealand, September 11-13, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11718)*, Xing Fan, Bronis R. de Supinski, Oliver Sinnen, and Nasser Giacaman (Eds.). Springer, 153–167. https://doi.org/10.1007/978-3-030-28596-8_11

[10] Free Software Foundation. [n.d.]. GCC, the GNU Compiler Collection, Offload Support. https://gcc.gnu.org/wiki/Offloading.

[11] Rahulkumar Gayatri, Charlene Yang, Thorsten Kurth, and Jack Deslippe. 2018. A Case Study for Performance Portability Using OpenMP 4.5. In *Accelerator Programming Using Directives - 5th International Workshop, WACCPD 2018, Dallas, TX, USA, November 11-17, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11381)*, Sunita Chandrasekaran, Guido Juckeland, and Sandra Wienke (Eds.). Springer, 75–95. https://doi.org/10.1007/978-3-030-12274-4_4

[12] Guray Ozen. 2017. Compiler and Runtime Based Parallelization and Optimization for GPUs. Ph.D. Dissertation.

[13] Akihiro Hayashi, Jun Shirako, Ettore Tiotto, Robert Ho, and Vivek Sarkar. 2019. Performance evaluation of OpenMP's target construct on GPUs - exploring compiler optimisations. *Int. J. High Perform. Comput. Netw.* 13, 1 (2019), 54–69. https://doi.org/10.1504/IJHPCN.2019.097051

[14] IBM. [n.d.]. XL Compiler for C, C++ and Fortran. https://www.ibm.com/products/xl-cpp-linux-compiler-power.

[15] Intel Corp. 2021. *Intel C++ Compiler Classic Developer Guide and Reference.*

[16] Arpith Chacko Jacob, Alexandre E. Eichenberger, Hyojin Sung, Samuel F. Antão, Gheorghe-Teodor Bercea, Carlo Bertolli, Alexey Bataev, Tian Jin, Tong Chen, Zehra Sura, Georgios Rokos, and Kevin O'Brien. 2017. Efficient Fork-Join on GPUs Through Warp Specialization. In *24th IEEE International Conference on High Performance Computing, HiPC 2017, Jaipur, India, December 18-21, 2017*. IEEE Computer Society, 358–367. https://doi.org/10.1109/HiPC.2017.00048

[17] Guido Juckeland, William C. Brantley, Sunita Chandrasekaran, Barbara M. Chapman, Shuai Che, Mathew E. Colgrove, Huiyu Feng, Alexander Grund, Robert Henschel, Wen-mei W. Hwu, Huian Li, Matthias S. Müller, Wolfgang E. Nagel, Maxim Perminov, Pavel Shelepugin, Kevin Skadron, John A. Stratton, Alexey Titov, Ke Wang, G. Matthijs van Waveren, Brian Whitney, Sandra Wienke, Rengan Xu, and Kalyan Kumaran. 2014. SPEC ACCEL: A Standard Application Suite for Measuring Hardware Accelerator Performance. In *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation - 5th International Workshop, PMBS 2014, New Orleans, LA, USA, November 16, 2014. Revised Selected Papers*. 46–67. https://doi.org/10.1007/978-3-319-17248-4_3

[18] Khronos OpenCL Working Group. 2020. *The OpenCL Specification, version 3.0.*

[19] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. 2009. OpenMP to GPGPU: a Compiler Framework for Automatic Translation and Optimization. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2009, Raleigh, NC, USA, February 14-18, 2009*. 101–110. https://doi.org/10.1145/1504176.1504194

[20] Seyong Lee and Jeffrey S. Vetter. 2014. OpenARC: extensible OpenACC compiler framework for directive-based accelerator programming study. In *Proceedings of the First Workshop on Accelerator Programming using Directives, WACCPD '14, New Orleans, Louisiana, USA, November 16-21, 2014*, Sunita Chandrasekaran, Fernanda S. Foertter, and Oscar R. Hernandez (Eds.). IEEE Computer Society, 1–11. https://doi.org/10.1109/WACCPD.2014.7

[21] Chunhua Liao, Yonghong Yan, Bronis R. de Supinski, Daniel J. Quinlan, and Barbara M. Chapman. 2013. Early Experiences with the OpenMP Accelerator Model. In *OpenMP in the Era of Low Power Devices and Accelerators - 9th International Workshop on OpenMP, IWOMP 2013, Canberra, ACT, Australia, September 16-18, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8122)*, Alistair P. Rendell, Barbara M. Chapman, and Matthias S. Müller (Eds.). Springer, 84–98. https://doi.org/10.1007/978-3-642-40698-0_7

[22] LLVM Team. [n.d.]. 2021. [Online]. The LLVM Compiler Infrastructure. https://github.com/llvm/llvm-project.

[23] Matt Martineau and Simon McIntosh-Smith. 2017. The Productivity, Portability and Performance of OpenMP 4.5 for Scientific Applications Targeting Intel CPUs, IBM CPUs, and NVIDIA GPUs. In *Scaling OpenMP for Exascale Performance and Portability - 13th International Workshop on OpenMP, IWOMP 2017, Stony Brook, NY, USA, September 20-22, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10468)*,

Bronis R. de Supinski, Stephen L. Olivier, Christian Terboven, Barbara M. Chapman, and Matthias S. Müller (Eds.). Springer, 185–200. https://doi.org/10.1007/978-3-319-65578-9_13

[24] NVIDIA Corp. [n.d.]. CUDA Dynamic Parallelism Programming Guide, 2013. ([n. d.]).

[25] NVIDIA Corp. 2021. *CUDA C++ Programming Guide Version 11.2.*

[26] OpenMP ARB. 2020. OpenMP Application Program Interface, v. 5.1. http://www.openmp.org.

[27] Guray Ozen, Simone Atzeni, Michael Wolfe, Annemarie Southwell, and Gary Klimowicz. 2018. OpenMP GPU Offload in Flang and LLVM. In *2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*. 1–9. https://doi.org/10.1109/LLVM-HPC.2018.8639434

[28] Guray Ozen, Eduard Ayguadé, and Jesús Labarta. 2014. On the Roles of the Programmer, the Compiler and the Runtime System When Programming Accelerators in OpenMP. In *Using and Improving OpenMP for Devices, Tasks, and More - 10th International Workshop on OpenMP, IWOMP 2014, Salvador, Brazil, September 28-30, 2014. Proceedings*. 215–229. https://doi.org/10.1007/978-3-319-11454-5_16

[29] Guray Ozen, Eduard Ayguadé, and Jesús Labarta. 2015. Exploring Dynamic Parallelism in OpenMP. In *Proceedings of the Second Workshop on Accelerator Programming using Directives, WACCPD 2015, Austin, Texas, USA, November 15, 2015*. 5:1–5:8. https://doi.org/10.1145/2832105.2832113

[30] Ettore Tiotto, Bardia Mahjour, Whitney Tsang, Xing Xue, Tarique Islam, and Wang Chen. 2020. OpenMP 4.5 compiler optimization for GPU offloading. *IBM J. Res. Dev.* 64, 3/4 (2020), 14:1–14:11. https://doi.org/10.1147/JRD.2019.2962428

[31] Jin Wang and Sudhakar Yalamanchili. 2014. Characterization and Analysis of Dynamic Parallelism in Unstructured GPU Applications. In *2014 IEEE International Symposium on Workload Characterization, IISWC 2014, Raleigh, NC, USA, October 26-28, 2014*. 51–60. https://doi.org/10.1109/IISWC.2014.6983039

[32] Michael Wolfe. 2010. Implementing the PGI Accelerator model. In *Proceedings of 3rd Workshop on General Purpose Processing on Graphics Processing Units, GPGPU 2010, Pittsburgh, Pennsylvania, USA, March 14, 2010*. 43–50. https://doi.org/10.1145/1735688.1735697

[33] M. Wolfe and C.-W. Tseng. 1992. The power test for data dependence. *IEEE Transactions on Parallel and Distributed Systems* 3, 5 (1992), 591–601. https://doi.org/10.1109/71.159042