# Efficient Fork-Join on GPUs through Warp Specialization

Arpith C. Jacob, Alexandre E. Eichenberger, Hyojin Sung, Samuel F. Antao, Gheorghe-Teodor Bercea, Carlo Bertolli, Alexey Bataev, Tian Jin, Tong Chen, Zehra Sura, Georgios Rokos, Kevin O'Brien
IBM T.J. Watson Research Center, 1101 Kitchawan Rd., Yorktown Heights, NY, USA
{acjacob,alexe,hsung,sfantao,Gheorghe-Teodor.Bercea,cbertol,alexey.bataev,tjin,chentong,zsura,grokos,caomhin}@us.ibm.com

***Abstract*—Graphics Processing Units (GPUs) are increasingly used to accelerate portions of general-purpose applications. Higher level language extensions have been proposed to help non-experts bridge the gap between a host and the GPU's threading model. Recent updates to the OpenMP standard allow a user to parallelize code on a GPU using the well known fork-join programming model for CPUs.**

**Mapping this model to the architecturally visible threading model of typical GPUs has been challenging. In this work we propose a novel approach using the technique of Warp Specialization. We show how to specialize one warp (a unit of 32 GPU threads) to handle sequential code on a GPU. When this master warp reaches a user-specified parallel region, it awakens unused GPU warps to collectively execute the parallel code. Based on this method, we have implemented a Clang-based, OpenMP 4.5 compliant, open source compiler for GPUs.**

**Our work achieves a $3.6\times$ (and up to $32\times$) performance improvement over a baseline that does not exploit fork-join parallelism on an NVIDIA k40m GPU across a set of 25 kernels. Compared to state-of-the-art compilers (Clang-ykt, GCC-OpenMP, GCC-OpenACC) our work is $2.1$ - $7.6\times$ faster. Our proposed technique is simpler to implement, robust, and performant.**

***Keywords*-OpenMP, Fork-Join, GPU, Warp Specialization**

## I. Introduction

Graphics Processing Units (GPUs) are increasingly used to accelerate portions of general-purpose applications in a variety of domains. Their compute power, combined with their ability to hide long memory latencies, make them suitable for accelerating highly parallel, compute-intensive algorithms with a large memory footprint. As a result, they are increasingly found in supercomputers, including Tianhe-1A, TITAN [1], Piz Daint [2], Summit [3], and Sierra [4].

As GPUs become more ubiquitous, programmers face the daunting challenge of tuning their kernels for GPU architectures. Historically, specialized language extensions such as CUDA [5] or OpenCL [6] have been popular. They allow an expert programmer to extract most of the available performance for their algorithms.

Exploiting parallelism on typical GPU architectures is fundamentally different compared to exploiting parallelism on a traditional CPU. One dissimilarity is in the amount of parallelism, where typical CPU parallelism ranges in the 16-256 thread count whereas ranges on GPUs can go upward of $1M$ threads organized as a two-level hierarchy. A fundamental challenge is that, unlike on the host, where additional parallel threads can be requested on an as-needed basis, threads on a GPU must be requested upfront at kernel launch time. Moreover, all requested GPU threads are active upon executing the first statement of the kernel.

Higher level language extensions [7], [8] have been proposed to help non-experts bridge the gap between a host and an accelerator such as a GPU. OpenMP provides an offloading model that allows a user to parallelize code on a GPU using the very same constructs used on a CPU. In this model, code on a GPU can start with a sequential thread and opportunistically recruit GPU threads as needed by the application.

Listing 1: Snippet from the Heartwall benchmark

```
 1 #pragma omp declare target
 2 void kernel(public_struct public, private_struct
       private) {
 3   ...
 4   if (public.frame_no != 0) {
 5     ...
 6     #pragma omp parallel for collapse(2)
 7     for(col=0; col<public.in2_cols; col++)
 8       for(row=0; row<public.in2_rows; row++)
 9         ...
10
11     in_final_sum = 0;
12     #pragma omp parallel for reduction(+: in_final_sum)
13     for(i = 0; i<public.in_mod_elem; i++)
14       in_final_sum = in_final_sum + d_in[i];
15
16     // expensive serial code.
17   }
18 }
19 #pragma omp end declare target
20
21 int main(...) {
22   #pragma omp target teams distribute
23   for(i=0; i<public.allPoints; i++)
24     kernel(public, private[i]);
25 }
```

Listing 1 shows the Heartwall benchmark from the Rodinia suite [9] that uses OpenMP to offload code to an accelerator. The ***target teams distribute*** directive at Line 22 offloads execution of the associated loop and distributes iterations across one dimension of GPU threading resources. The generic ***fork-join*** programming model is used to exploit nested parallelism at the loops on lines 6 and 12. While the rest of the code is executed by a team's master thread alone, worker threads are only activated on arriving at the ***parallel for*** directives.

The focus of this paper is on a novel approach to bridge the gap between such a generic ***fork-join*** programming model familiar to all parallel programmers and the architecturally visible threading model of typical GPUs.

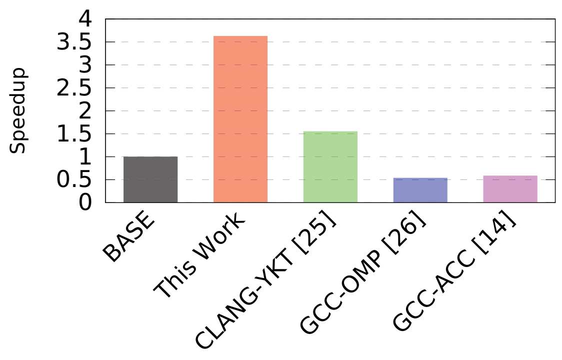


Fig. 1: Geometric mean of speedups across a set of 25 kernels that exploits nested parallelism as compared to a baseline that only exploits one level of parallelism.

A majority of OpenMP compilers for the GPU [10], [11], [12], [13] only exploit one level of parallelism. In the aforementioned example, they parallelize the outer loop on Line 23 across GPU threads but can only execute the inner loops serially. In contrast, our compiler is also able to efficiently exploit nested parallelism. Consequently, as shown in Figure 1 our work achieves a 3.6× (and up to 32×) performance improvement on an NVIDIA k40m GPU across a benchmark of 25 kernels over a baseline that does not exploit nested parallelism (§ VI-B).

We are aware of two techniques that can exploit nested parallelism on the GPU. Basic Block Neutering [14], [15] introduces additional control flow that causes unused GPU threads to bypass basic blocks in a serial region. However, even after optimizations [14], [15], there is non-negligible communication overhead to the unused threads to inform them of control flow decisions taken by the serial (master) GPU thread (§ IV-A). Another approach uses a Master-Controlled State Machine [16], where the serial thread controls a state machine that guides the unused threads to their next parallel region (§ IV-B). This technique suffers performance loss when encountering calls to functions that potentially contain parallel code. As Figure 1 shows, these state-of-the-art compilers achieve no more than a 1.56× speedup over the baseline, with several exhibiting a slowdown on our benchmark suite. In contrast to these approaches, our proposed technique is simpler to implement, robust, and performant. On the same benchmark our compiler is 2.1-7.6× faster than these state-of-the-art OpenMP and OpenACC compilers (§ VI-C).

The contribution of this paper is a novel technique to handle the ***fork-join*** programming model on GPU architectures using a technique known as Warp Specialization [17]. We show how to specialize one warp (a unit of 32 GPU threads on NVIDIA architectures) to handle all sequential code on a GPU. When this master warp reaches a parallel region, it awakens unused GPU warps for the duration of the parallel region.

While this approach does not exhibit drawbacks of past implementations, it required us to draw on lower-level barrier constructs not available in CUDA. In addition, as the state of the sequential thread needs to migrate to distinct GPU threads during parallel execution, we engineered a globally addressable stack to store shared data between sequential and parallel parts of the application.

The remainder of the paper is organized as follows. After background information (§ II) we elucidate the challenges in mapping ***fork-join*** parallelism onto the CUDA programming model (§ III) and how existing approaches tackle this problem (§ IV). We then describe our approach (§ V) and perform a detailed performance evaluation (§ VI).

## II. Background

### A. GPU Hardware and the CUDA Programming Model

A GPU organizes execution contexts into groups called ***warps*** with typically 32 in each. A warp executes on a Single-Instruction, Multiple-Thread (SIMT) unit that issues a single instruction from the warp every cycle. One or more of the 32 SIMT threads in a warp that have the same program counter as the issued instruction are executed in parallel. Divergent threads in a warp serialize execution.

A group of warps are organized within a contention group known as a Cooperative Thread Array (CTA). A GPU kernel may consist of thousands of CTAs that are scheduled in turn on tens of processors within a GPU. The hardware provides primitives that allow warps within a CTA to synchronize efficiently and communicate via low-latency shared memory.

A program written in the CUDA [5] language consists of one or more kernels launched from a host process and mapped onto this GPU hardware. The CUDA programming model is a threaded, fine-grained Single Program Multiple Data (SPMD) model [18] where a kernel launch initiates multiple concurrent instances of the same program segment. Each instance is executed by a distinct CUDA thread, typically operating on different data. A CUDA kernel starts execution with a predetermined hierarchy of CUDA threads organized as a grid of CTAs and threads in a CTA. Threads within a CTA synchronize using the barrier primitive ***__syncthreads*** .

The CUDA programming model allows a compiler to map one CUDA thread to one of the 32 SIMT threads in a warp while concealing the presence of SIMT hardware from the programmer. However, the SPMD model forces the user to expose all available parallelism at the kernel entry point. Additional threads cannot be activated on demand, as required by the ***fork-join*** model, to exploit nested parallelism.

### B. The OpenMP Programming Model

OpenMP [8] is a portable, shared-memory programming model implemented as a set of directives and runtime calls. The ***fork-join*** model is used to exploit parallelism.

An OpenMP program begins with a single master thread on a host device. On encountering a ***parallel*** directive, execution forks and the code region associated with the directive is executed in parallel by a team of threads. At the end of the parallel region the threads in the team join at an implied barrier and only the master thread continues execution. Worksharing directives such as ***for*** and ***simd*** partition iterations of loops across threads or SIMD lanes.

OpenMP includes an offload model implemented using the ***target*** directive. When a host thread encounters this directive, its associated code region is offloaded to an accelerator that begins execution in a single thread. When a ***teams*** directive is associated with a target construct, a league of teams is launched on the accelerator (one per CTA), each with a single thread of execution. The ***distribute*** construct allows worksharing of loop iterations across teams. These two directives allow the user to exploit one dimension of parallelism. A second dimension can be exploited by each team master using the ***fork-join*** model of the ***parallel*** directive.

An accelerator's data environment is set up using ***map*** clauses, which may allocate storage and transfer data to and from device storage. For a complete list of OpenMP directives and clauses we refer the reader to the OpenMP specifications [8].

## III. Challenges in Mapping the Fork-Join Model to CUDA

The familiar ***fork-join*** model enables flexible extraction of parallelism from kernels with serial code, greatly enhancing productivity. However, the burden now shifts to the compiler to efficiently map this model to GPU hardware that is tuned for the SPMD model of CUDA. We illustrate the challenges of this problem using two benchmarks, Heartwall and Histo, that we ported to OpenMP 4.5.

Listing 2: Image histogram benchmark

```
 1 #pragma omp target teams
 2 {
 3   unsigned int private_histo[BINS];
 4   #pragma omp parallel num_threads(BINS)
 5   {
 6     #pragma omp for
 7     for (int i=0;i<BINS;i++)
 8       private_histo[i] = 0;
 9
10     int lb = omp_get_team_num() * omp_get_num_threads()
          + omp_get_thread_num();
11     int st = omp_get_num_teams() * omp_get_num_threads
          ();
12     for (int i = lb; i < size; i+=st) {
13       #pragma omp atomic
14       private_histo[(data[i] * BINS) >> 12]++;
15     }
16
17     #pragma omp barrier
18
19     #pragma omp for
20     for (int i=0;i<BINS;i++)
21       #pragma omp atomic
22       histo[i] += private_histo[i];
23   } // parallel
24 } // target teams
```

Consider how a compiler may map the OpenMP program in Listing 1 to the CUDA model. A host thread that encounters the target region at Line 22 launches a GPU kernel with a default number of CTAs and CUDA threads (e.g., 1024 per CTA). Recall that a GPU kernel launch initiates one instance of the target region per CUDA thread.

*a) Limitation 1: OpenMP kernels with varying degrees of parallelism must be mapped to an SPMD model with a fixed degree of parallelism:* Many programs contain sequential code to be executed by the team master alone that is interspersed between parallel regions. Other kernels may have multiple parallel regions, each with a different number of active threads. Mapping these programs to SPMD hardware requires coordination of SIMT threads through regions of varying degrees of parallelism with only the appropriate number activated.

Since only the team master is active at Line 23 of Listing 1, 1023 threads must be deactivated. They must, however, *follow* the same control flow path as the master thread through the function *kernel* and arrive together at the parallel regions at Lines 6 and 12.

*b) Limitation 2: The CUDA model does not allow synchronization of subsets of threads in a CTA:* Consider the OpenMP 4.5 port of the Histogram benchmark shown in Listing 2. We again assume that the GPU kernel for this target region is launched with the default number of 1024 threads per CTA. A user-requested number of worker threads, as specified by the variable *BINS*, which may in general be less than 1024, are activated when the master encounters the enclosed parallel region. The user barrier on Line 17 may erroneously be implemented with the CUDA ***__syncthreads*** primitive. However, this can result in a program that deadlocks since only a subset of the 1024 CUDA threads in a CTA will ever arrive at the barrier. Indeed, OpenMP user barriers cannot be implemented in a straightforward manner with the simple CUDA barrier primitive since not all threads in a CTA may participate in a parallel region.

*c) Limitation 3: Only a single thread synchronization primitive is available in the CUDA model:* A general OpenMP program requires two distinct barriers to implement a parallel construct: one for active and inactive threads to wait on at the end of the parallel region, and another to implement implicit or explicit barriers within the parallel section. For the Histo benchmark in Listing 2, inactive threads wait on a barrier at Line 23 until all workers complete execution of the parallel region. Workers in the parallel region may simultaneously wait on another barrier at Line 17 or at the end of the worksharing loops on Lines 6 and 19.

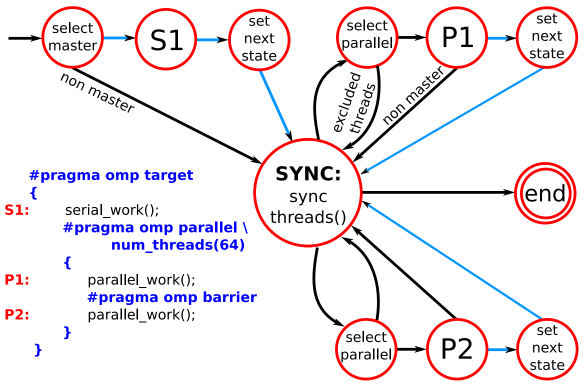
The limitations we have outlined in this section make it challenging to map OpenMP programs onto a GPU using the CUDA programming model. Significant effort has been expended to bypass these limitations by developing code transformations that enable a legal mapping onto CUDA.

## IV. Related Work

In this section we describe existing code transformation techniques to map the OpenMP model onto GPUs.

### A. Basic Block Neutering

This approach introduces control flow that causes workers to bypass a basic block in a serial region [14], [19]. While active threads execute statements in the basic block, inactive threads simply follow the control flow path as decided by the master thread.



```
    #pragma omp target
    {
S1:     serial_work();
        #pragma omp parallel \
               num_threads(64)
        {
P1:         parallel_work();
            #pragma omp barrier
P2:         parallel_work();
        }
    }
```

Fig. 2: An OpenMP 4.5 program and the state machine generated for its execution on a GPU by Bertolli et al. [16].

The master computes the condition expression of a branch, writes to a common memory location, and synchronizes with the workers. The workers read this result and branch to the appropriate target, thus following the master. The short-circuit evaluation rules of boolean operators in languages like C and C++ contribute to the inefficiency of this approach. In addition, memory synchronization primitives may inhibit traditional compiler optimizations.

Lee [15] and others [14] present code transformation algorithms that minimize the number of introduced control-flow statements. For example, an entire single-entry single-exit region can be skipped by worker threads if it can be determined that there is no parallel region within it. However, this can be challenging to conclusively prove, for example, in the presence of function calls.

None of these published works propose a solution to the last two limitations listed in § III.

## B. *Master-Controlled State Machine*

Bertolli et al. [16] suggest a scheme that defines a state machine from an OpenMP target region. The master thread guides workers through states, with periodic thread synchronization at a well-defined *barrier* state ***SYNC***.

Figure 2 shows an example, with states ***S1***, ***P1***, and ***P2*** representing code sections. Before entering each of these states a predicate activates the appropriate number of workers. After executing a code section the master sets the next state for all workers. All threads synchronize at the ***SYNC*** state, implemented with the standard CUDA primitive, before progressing to the next OpenMP region.

The state machine is constructed in this manner to overcome the limitations in § III. Unfortunately, the introduction of control flow circuits through the single ***SYNC*** state disrupts data flow analysis, which may affect the quality of generated code. Finally, it is challenging to handle orphaned OpenMP directives in user functions with this scheme.

## V. OpenMP Code Generation

We believe the complexity of existing work arises because they have always abstracted GPU hardware as an SPMD execution device. Prior work models a CTA as a collection of CUDA threads onto which OpenMP threads are mapped. This mapping is challenging to do correctly on the underlying SIMT hardware. The key idea of our work is to treat a CTA as a collection of warps that can be specialized [17] for distinct roles within the OpenMP execution model.

We designate one warp in a CTA as the master and all others as workers. The master warp is solely responsible for executing serial sections while only worker warps execute parallel regions.

Existing work, which makes no such separation of concerns, maps OpenMP threads to CUDA's SPMD model and forces all CUDA threads in a CTA to navigate through serial and parallel regions. This leads to the complex, intertwined code generation strategies described in the previous section. Instead, by assigning a warp to execute either a serial or a parallel region but not both, code generation can be tailored to each task.

In the following sections we first describe a model for programming warps in a CTA and then present our mapping for the OpenMP ***fork-join*** model.

## A. *Warp Specialized Programming*

Our approach exploits the fact that two different warps can execute distinct code regions without suffering a performance penalty as long as SIMT threads within each warp do not diverge. To coordinate warps we require a new barrier primitive that can synchronize warps.

To synchronize the master and worker warps we define the CTA-level synchronization primitive ***cta.sync $0, $1***. The first argument to this primitive is an integer that identifies a barrier. The second argument indicates the number of warps participating in the barrier. This definition permits a subset of the total number of warps in a CTA to participate in a barrier. Additionally, different subsets of warps in a CTA may wait on different barriers, as determined by the identifier argument.

A warp is said to *arrive* at a barrier when *any* of its SIMT threads execute the primitive. Different warps may arrive at the barrier via different syntactic locations. However, for correct operation, the SIMT threads of a warp that is participating in a barrier must execute the primitive exactly once, i.e., the primitive must be issued exactly once per warp. A collection of SIMT threads arriving at a barrier is blocked until the barrier completes. Warps are released through a barrier when all participating warps arrive.

*a) Model Implementation:* Our model can be implemented on modern SIMT hardware from NVIDIA and AMD by directly using assembly-level PTX and HSA instructions respectively. Warp-level synchronization can be achieved with *named barriers*[1] on NVIDIA GPUs and *fine-grain barriers*[2] on AMD GPUs.

## B. *A Pool-of-Warps for a Parallel Region*

With our model defined, we can now introduce our approach to exploiting fork-join parallelism. When an OpenMP target

[1]http://docs.nvidia.com/cuda/parallel-thread-execution/#parallel-synchronization-and-communication-instructions-bar

[2]http://www.hsafoundation.com/html_spec11/HSA_Library.htm#PRM/Topics/09_Parallel/fine_grain_barrier.htm

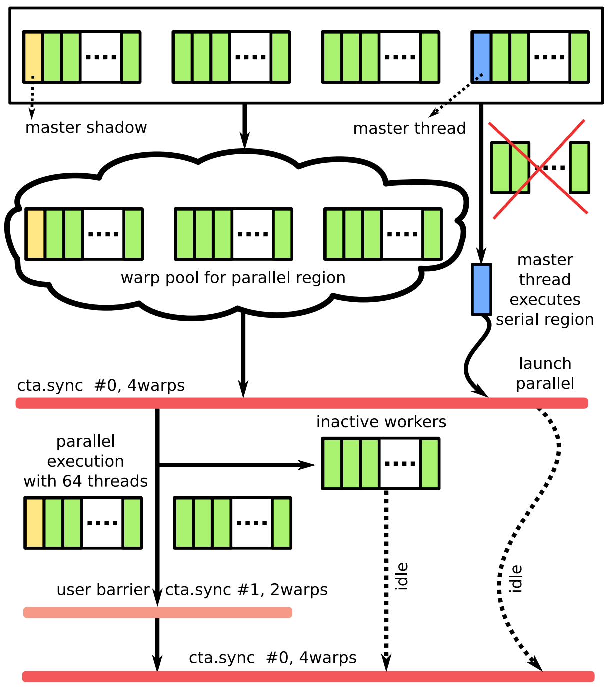


Fig. 3: Our approach to exploiting fork-join parallelism on GPUs for the program in Figure 2. The GPU kernel is launched with the first three warps dedicated for parallel regions and the last for serial execution. Time flows from top to bottom.

region is launched as a GPU kernel the last warp of a CTA is designated as the master. All other warps are placed in a pool. Within the master warp we deactivate all but one of its SIMT threads[3]. The OpenMP *master* thread is mapped to the one SIMT thread in this warp and executes serial regions.

When the master encounters a parallel region, it activates the worker warps and suspends execution. We map one OpenMP worker thread to each of the SIMT threads in worker warps. This allows us to fully exploit the SIMT hardware for regular OpenMP programs.

In the parallel region, the execution context of the master is resurrected on the first SIMT thread of the first worker warp as a *shadow*. When worker warps reach the end of the parallel region they are suspended, the master thread is reincarnated in the master warp, and single-threaded execution resumes.

Figure 3 illustrates this process for the OpenMP program listed in Figure 2. Upon kernel launch worker warps immediately enter a pool, awaiting work from the master. This is implemented by waiting on barrier #0.

When the master thread encounters a parallel region it *arrives* at barrier #0 to release workers. One or more of the worker warps, as controlled by an optional ***num_threads*** clause, execute the parallel region.

An OpenMP user barrier in a parallel region is implemented with our synchronization primitive on barrier #1. Only warps actively executing the parallel region participate in this barrier.

[3]This is typically done by causing the deactivated SIMT threads to immediately return from the kernel and is required for correct operation of our synchronization primitive.

At the end of the parallel region the workers wake up the master thread (barrier #0 can be recycled for this purpose) and return to the pool.

**Pseudocode 1** Fork-Join codegen for SIMT hardware

```
 1: function TARGET()  ▷ Kernel launched with num_warps warps per CTA
 2:   if ISMASTERWARP() then  MASTERFN()
 3:   else  WORKERFN()
 4:
 5: function MASTERFN()                 ▷ Executed by the master warp
 6:   if ISNOTFIRSTSIMTTHREAD() then  return
 7:   $Serial\ Region\ S1$                 ▷ Single threaded execution
 8:   $WorkFn \leftarrow$ PARALLELFN   ▷ Encountered a parallel region
 9:   $OmpThreads \leftarrow user\ requested\ num\_threads$
10:   cta.sync #0, $num\_warps$                     ▷ Activate workers
11:                 ▷ Implicit barrier at the end of a parallel region
12:   cta.sync #0, $num\_warps$
13:
14:   $WorkFn \leftarrow \varnothing$                      ▷ Termination condition
15:   cta.sync #0, $num\_warps$
16:
17: function WORKERFN()                ▷ Executed by worker warps
18:   loop
19:     cta.sync #0, $num\_warps$        ▷ Wait for parallel work
20:     if $WorkFn = \varnothing$ then break    ▷ Termination condition
21:
22:     if CUDATHREADID() $< OmpThreads$ then
23:       $WorkFn()$                     ▷ Execute parallel region
24:
25:                 ▷ Implicit barrier at the end of a parallel region
26:     cta.sync #0, $num\_warps$
27:   end loop
28:
29: function PARALLELFN()               ▷ Outlined Parallel Region
30:   Parallel Code P1
31:   $ActiveWarps \leftarrow \left\lceil \frac{OmpThreads}{WarpSize} \right\rceil$
32:   cta.sync #1, $ActiveWarps$                       ▷ User barrier
33:   Parallel Code P2
```

Pseudocode 1 shows the code generated by our compiler for the OpenMP program in Figure 2. Statements 2 and 3 specialize execution by warp type. To preserve the semantics of our synchronization primitive Statement 6 deactivates all but the first SIMT thread in the master warp.

Our compiler outlines code regions associated with a parallel directive into functions (see PARALLELFN). When the master thread encounters a parallel directive it assigns the associated outlined function to worker warps through the communicator *WorkFn*. The requested number of active OpenMP threads in the parallel region is also communicated to worker warps. Only active OpenMP worker threads call the outlined function to execute the parallel region.

Note that the parallel region need not be nested in the same lexical scope as the target directive; the master may encounter it in a separate translation unit and the workers execute the region by making an indirect call. However, for efficiency reasons, we convert the indirect call to direct calls when possible using the following pattern[4]:

```
if $WorkFn$ = ParallelFn1 then ParallelFn1()
else if $WorkFn$ = ParallelFn2 then ParallelFn2()
else if $WorkFn$ = $\cdots$ then $\cdots$
```

Our scheme implements user barriers using the synchronization primitive for warps (removing Limitation 2). The availability of two distinct barriers enables the concurrent implementation of the implicit barrier at the end of a parallel region and the user barrier within it (lifting Limitation 3).

## C. Optimizations for High Performance

We have implemented the proposed code generator in Clang and the LLVM toolchain. In this section we briefly summarize several optimizations that are important to achieve high performance.

*a) Sharing the Master's Stack with Workers:* Modern GPUs store the stack of a SIMT thread in a local address space that cannot be referenced outside its context. In the OpenMP model, variables declared on the master thread's stack may be live-in (live-out) to (from) a parallel region. We therefore implement a soft stack in globally addressable memory to permit sharing of variables with worker threads.

The stack is organized as a list of *slots*, with each slot pointing to a chunk of fixed-size storage in off-chip memory. Storage for the first slot is preallocated. Only if there is insufficient storage (e.g. when calling an external function that is not visible at the call site) is the slot closed and a new one dynamically allocated on the GPU.

*b) Managing OpenMP State:* Internal state variables that control the scheduling of OpenMP threads along with the soft stack are of considerable size and must be stored off-chip. The challenge is to manage this storage for the large number of concurrent teams initiated on a GPU kernel launch[5]. It is impractical to preallocate storage for all teams, and undesirable for users to limit the number of concurrent teams.

We exploit the fact that only a small number of teams are resident simultaneously on a GPU's processors (16 to 32 per processor). We preallocate up to 32 buffers in a per-processor queue from which the team master checks out a buffer for the duration of its execution[6]. High throughput queues for GPUs [20] minimize overhead and contention is limited to teams running on the same processor.

*c) Inlining and Eliding the OpenMP Runtime:* Our OpenMP runtime is implemented in a library that is translated to LLVM bitcode and fully inlined within user programs. This avoids the overhead of function calls on the GPU.

Nevertheless, the overhead of the runtime is often evident, particularly for simple kernels. Many offloaded kernels do not require the entire capabilities of the OpenMP runtime. For example, most worksharing loops are statically not dynamically scheduled; SPMD constructs do not require the soft stack; and when other constructs do require it, the stack can often be implemented statically in the GPU's shared memory scratchpad.

Our compiler completely elides the runtime for many commonly used offload patterns. This removes the overhead of state initialization/deinitialization and uses a simplified code path that references special purpose registers on the GPU[7] instead of off-chip memory.

[4]This allows the low-level GPU compiler to more accurately assess the resource requirements of the kernel.

[5]GPU kernels execute with billions of concurrent teams [5].

[6]The queue is guaranteed to be non-empty if its size is equal to the maximum number of resident teams per processor.

## VI. Performance Evaluation

In this section we evaluate the performance of our proposed OpenMP code generator and compiler. We start with a detailed description of our evaluation environment. Our first experiment measures the overheads of common OpenMP directives used to exploit ***fork-join*** parallelism on the GPU. Next, we focus on the performance advantage of exploiting nested parallelism across a diverse set of benchmarks. Finally, we compare our work against the state-of-the-art.

*a) Experimental Setup:* We run our experiments on an OpenPower node with Power 8 processors (model PowerNV 8247-42L) attached via PCIe to NVIDIA Kepler GPUs (K40m, CC 3.5) with 12 GiB RAM each. The GPU has 15 multiprocessors clocked at 875 MHz and runs with ECC enabled.

We use Nvidia's profiler to collect and report the execution times for kernels of interest. This helps us isolate active execution time on the GPU from data transfer time, which is consistent regardless of the code generation technique or compiler that is used. Each benchmark is run five times and the average runtime is calculated with the $99\%$ confidence interval within $5\%$ of the reported means. We summarize speedups across benchmarks using the geometric mean, having ensured that the distribution is log-normal with the Shapiro-Wilk test.

*b) Benchmarks:* To understand the performance limits of our compiler we evaluate a number of micro-benchmarks and benchmarks from standard suites.

Several of our benchmarks are part of the recently released SPEC ACCEL suite (*pomriq, pep, pcsp, pbt*). These are the C/C++ codes of the suite that contain nested parallelism. *imghisto* [21] computes the $N$-bin histogram of a 12-bit monochrome image. *HACCmk* [22] is a short-force evaluation routine part of the CORAL benchmark suite. *fginv* and *tnsrtrns* are respectively a batched fine-grained GJE matrix inversion kernel and a tensor transpose kernel.

We also use several applications from the Rodinia [9] suite that are written in OpenMP, focusing on those that contain nested parallelism (*bfs, kmeans, particlefilter, b+tree, lavamd, streamcluster, heartwall, backprop, lud*). We did not include two benchmarks, *mummergpu* and *leukocyte*. The former uses CUDA APIs and kernels, while the latter uses a large number of heap allocations that cause an out-of-memory error on the GPU. The Rodinia suite is originally written in OpenMP 3.1 for CPUs. We ported it to OpenMP 4.5 to enable offloading to accelerators. Care was taken to remain faithful to the original algorithm and data layout.

[7]For many constructs the OpenMP thread number, team number and number of teams map directly to threadId, blockId, and blockDim.

*c) Compilers:* We have implemented our OpenMP 4.5 compiler and runtime in Clang/LLVM [23] based on the 4.0.0 toolchain in the mainline repository (updated January 2017). We have extended the OpenMP code generator in Clang to implement our proposal. After an OpenMP program is optimized by LLVM, the NVPTX backend generates PTX assembly that is compiled and linked with *ptxas* and *nvlink* respectively (CUDA Toolkit v8.0.61). Our contributions are open source and we are in the process of submitting patches to the LLVM community. The development branch of our compiler and runtime are available on GitHub [24].

To understand the quality of our implementation we also compare our work against state-of-the-art compilers from academia and industry. Several OpenMP 4.5 compilers for GPUs have been published in the literature. Unfortunately we found many of them to be unstable: several [10], [12], [13] failed to compile or successfully run our suite. Ones that are more robust, and which we use in our comparison, are Clang-ykt (v3.8.0) [25] and GCC-OMP (v7.1.0) [26]. We also ported our suite to OpenACC and tried the ACCULL [27], CAPS, and OpenUH [28] compilers but found them to be brittle, failing on our suite, and only succeeding with GCC-ACC (v7.1.0) [26], [14]. Table I summarizes this information, showing the flags used and those kernels from our suite that nevertheless failed to compile or run successfully with each compiler.

| Compiler | Flags | Failed Kernels |
|---|---|---|
| This work | *-O3 -ffast-math -ffp-contract=fast -fopenmp -fopenmp-targets=nvptx64-nvidia-cuda -fopenmp-nonaliased-maps* | None |
| CLANG-YKT [25] | *-O3 -ffast-math -ffp-contract=fast -fopenmp -target powerpc64le-ibm-linux-gnu -mcpu=pwr8 -omptargets=nvptx64sm_35-nvidia-linux* | kmeans, pbt, btree-nested |
| GCC-OMP [26] | *-fopenmp -O3 -ffast-math -foffload=-lm* | imghisto, fginv-spmd, tnsrtrns-spmd, streamcluster-spmd, btree-nested |
| GCC-ACC [26], [14] | *-fopenacc -O3 -ffast-math -foffload=-lm* | kmeans, tnsrtrns-nested, btree-nested, imghisto-nested |

TABLE I: Compilers studied in our experiment.

In all of our experiments we prefer to let the compiler and offload library select the number of teams and threads for each kernel. However, we found Clang-ykt and GCC-ACC to select suboptimal launch parameters. For a fair comparison, we use clauses to manually set the team size and the number of teams. Note that these launch parameters are automatically selected by our compiler and runtime.

*d) SPMD vs. Nested Parallelism:* We implement two versions of each benchmark. The first, our baseline, places the construct ***target teams distribute parallel for collapse(...)*** at the outermost loop(s) of a kernel. This is a proxy for the majority of OpenMP compilers for the GPU [10], [11], [12], [13] that only exploits one level of parallelism. Our second version uses the ***target teams distribute*** directive to exploit outer parallelism across teams and the ***parallel for*** directive on an inner loop to exploit nested parallelism within a team.

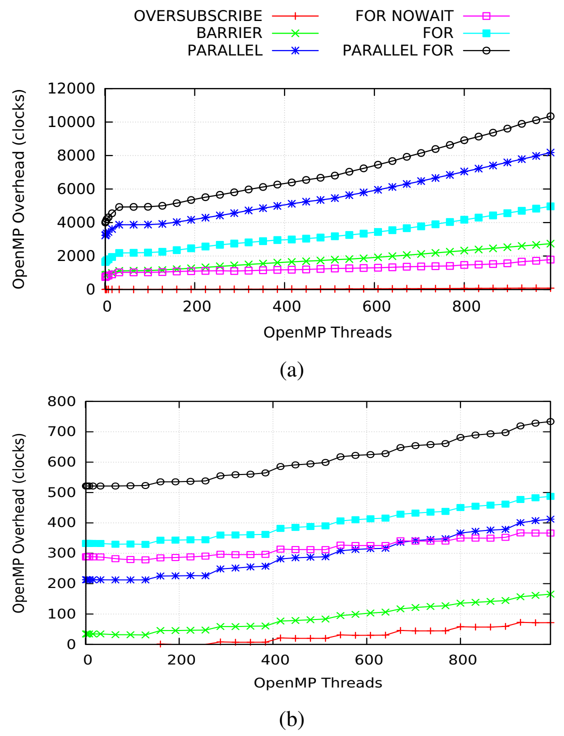


Fig. 4: Overheads of various OpenMP constructs with the runtime (top) and the runtime elision optimization (bottom).

## A. Overhead of Parallelization with OpenMP

In our first experiment, we use the well-known EPCC OpenMP micro-benchmark suite [29] to measure overhead of OpenMP constructs. We study the quality of our implementation by porting this benchmark to OpenMP 4.5 and running it on the GPU with our compiler.

We compare the execution time $T_s$ of a serial code region running on the master thread of one team to the parallel execution time $T_p$ with $p$ workers in the team of the code section enclosed by a given directive. The overhead of the directive is calculated as $T_p - \frac{T_s}{p}$. We refer the reader to Bull [29] for complete details of the methodology.

Figure 4 plots the overhead, measured for several directives, as the number of workers in a parallel region scales to 992 (recall that the master warp does not participate in the parallel region). The graph on the top shows the overhead in the worst case, when the entire capabilities of the OpenMP runtime is required. Overhead is measured to be thousands of cycles and is almost entirely due to loads and stores to data structures residing in external memory. Since GPUs have very limited caches they expose threads to the long latency of accesses to off-chip memory.

Figure 4b shows the overhead with our runtime elision optimization that applies to many real-world programs. For example, if the parallel directive does not have a ***num_threads*** clause we can directly read the default team size from the special-purpose register blockDim instead of referencing external memory. Overhead is reduced to just 200-300 cycles and is dominated by the cost of warp specialization and the signaling

| Kernel | Loop Itr. Ratio (Outer:Inner) | Inner Loop Size | Reduction/ Atomics | Coa-lescing |
|---|---|---|---|---|
| kmeans | 10,000:1 | VL | OR, IR | F |
| bfs | 10,000:1 | L | N/A | P |
| particlefilter | 100:1 | L | IR | P |
| pbt:K3 | 20:1 | G | N/A | F |
| pbt:K4 | 20:1 | G | N/A | F |
| pbt:K5 | 20:1 | G | N/A | F |
| pomriq | 10:1 | M | IR | P |
| pcsp:K1 | 5:1 | L | N/A | P |
| pcsp:K2 | 5:1 | L | N/A | N |
| b+tree | 3:1 | M | IR | F |
| lavamd | 1:10 | M | N/A | P |
| haccmk | 1:20 | M | IR | F |
| heartwall | 1:20 | VG | IR | P |
| streamcluster | 1:25 | L | OR, IR, OA | F |
| tnsrtrns | 1:50 | L | N/A | F |
| pcsp:K3 | 1:80 | M | N/A | F |
| pep:K1 | 1:100 | L | N/A | F |
| fginv | 1:100 | M | N/A | P |
| pep:K2 | 1:150 | L | OR, IR | N |
| pbt:K1 | 1:500 | G | N/A | P |
| pbt:K2 | 1:500 | G | N/A | P |
| lud:K1 | 1:500 to 500:1 | VL | IR | P |
| lud:K2 | 1:500 to 500:1 | VL | IR | P |
| backprop | 1:4,000; 4,000:1 | VL | IR | P |
| imghisto | 1:6,000 | VL | OA, IA | P |

TABLE II: Kernel Metrics. Key: Inner loop size is either very low (**VL**), low (**L**), moderate (**M**), large (**G**), or very large (**VG**); Reduction/Atomics are Inner Loop Reduction (**IR**), Outer Loop Reduction (**OR**), Inner Loop Atomics (**IA**), and Outer Loop Atomics (**OA**); Coalesced accesses are either none (**N**), partial (**P**) or full (**F**).

mechanism for the pool of warps. The low overhead can be attributed to the use of efficient hardware barriers for thread synchronization. Similar optimizations dramatically reduce overhead of all directives to just hundreds of cycles.

Overhead scales very well until around 200 threads and then increases gradually. We surmise the increase is due to oversubscription of warps from the team, as is demonstrated by the curve marked *oversubscribe*, which is the reference program running on multiple threads written so as to only include the hardware cost of oversubscription. The ***for nowait*** directive closely tracks the gradient of this curve; other directives, all of which include a barrier, scale slightly worse due to the increased cost of synchronizing a larger number of warps.

## B. Exploiting Nested Parallelism

In our second experiment we quantify the advantage of exploiting nested parallelism on the GPU. Figure 5 shows the speedup of various OpenMP kernels exploiting nested parallelism as compared to their SPMD versions. Except *bfs, kmeans* and *pomriq*, all benchmarks show similar or improved performance when exploiting nested parallelism (3.6× on average). *backprop* benefits the most with a 32× speedup, while *particlefilter* and *heartwall* performed about the same for both versions. For *bfs, kmeans* and *pomriq*, SPMD versions outperform the nested parallel versions by 30% to 4×.

We identified the major sources of the performance differences between SPMD and nested parallelism versions as follows: (1) outer-loop vs. inner-loop iteration counts, (2) the amount of work in the inner loop, (3) memory coalescing, and (4) reduction or atomic clauses in the outer/inner loop. Table II shows measurements of these metrics for all the benchmarks.

Nested parallel versions with sufficient work within the inner loop outperform SPMD versions as it efficiently exploits the additional level of intra-team parallelism. The level of intra-team parallelism is decided mainly by the iteration count of the inner loop and the size of the loop body. Among the benchmarks evaluated, those with an inner loop count less than 100 all performed worse with the nested parallelism version (*bfs, kmeans, particlefilter, heartwall*). These benchmarks fail to amortize the OpenMP overhead of invoking worker threads.

For benchmarks that have higher inner loop iteration counts, the ratio between available inter-team (outer loop) and intra-team (inner loop) parallelism is a more precise predictor of performance. SPMD versions perform well with enough inter-team parallelism, reducing the potential performance gain from exploiting an additional level of parallelism. *b+tree, lud, pomriq, pcsp* and *pbt* have outer-loop iteration counts higher than inner loops. These benchmarks show relatively low speed-up or even slow down compared to SPMD versions (except for *pbt* which has large inner loop bodies). *pomriq* suffers the most because its outer loop has 10× more iterations than the inner loop and its inner loop has a very small loop body.

On the other hand, all benchmarks with higher inner loop iteration count than the outer loop (ratio > 1) show better performance when exploiting nested parallelism. Nevertheless, the measured speedup between the two versions varies significantly across the benchmarks. Next, we identify two factors that determine the actual speedup realized.

**Atomic and Reduction Overheads:** *Atomic* and *reduction* operations within target regions add additional synchronization or communication overheads, slowing down thread execution. For example, *streamcluster* has an atomic directive in the outer loop, causing all threads in SPMD versions to perform the atomic operation while only team masters perform it in the nested parallel version. The *reduction* operation adds a sequential gather phase at the end of parallel execution. We confirmed that reduction clauses slow down inner or outer loop execution by comparing a modified version without reduction clauses to the original version for *particlefilter, b+tree, heartwall, streamcluster, hacckmk, backprop, lud, pomriq*, and *pep*.

**Coalesced Accesses:** The GPU issues a much larger number of memory accesses in parallel from active threads than a CPU so coalescing memory accesses to reduce the number of memory requests and bandwidth pressure is critical. Memory accesses can be coalesced only when they are contiguous in address space across threads in the same warp. We observed in general that benchmarks with moderately higher intra-team parallelism only exhibited a significant speedup when memory accesses are coalesced.

## C. Comparison to Related Work

In our final experiment we compare our compiler to existing work. Figure 6a plots comparative performance of the SPMD pattern that only exploits parallelism at the outermost level. Speedup is measured as the execution time of a benchmark

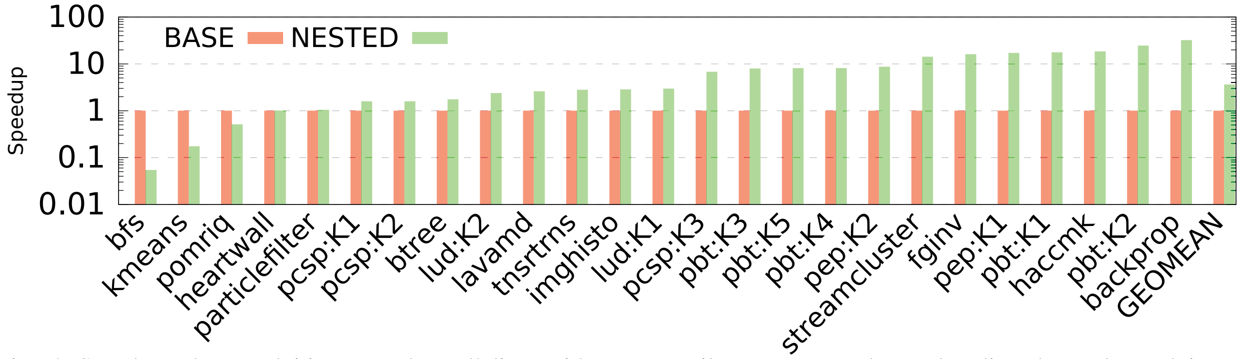
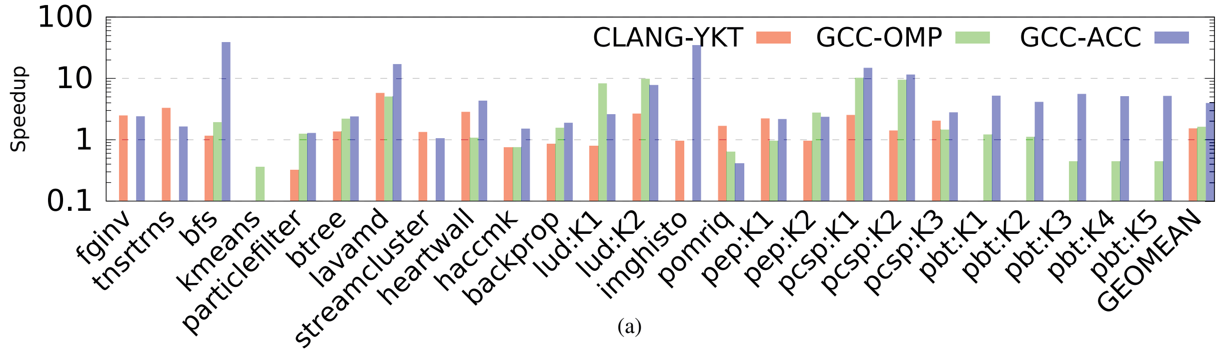
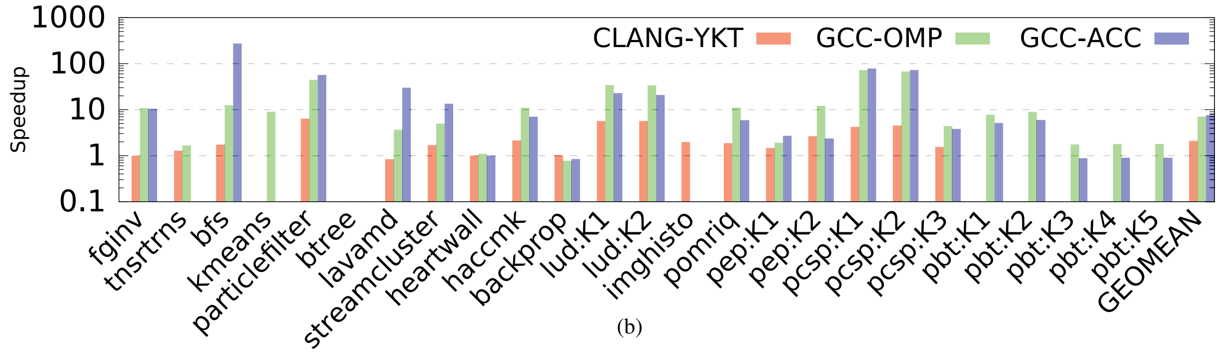


Fig. 5: Speedup when exploiting nested parallelism with our compiler as compared to a baseline that only exploits outer parallelism.



(a)



(b)

Fig. 6: Benchmark speedup of our compilers vs. existing work on (a) the baseline SPMD pattern and (b) the nested parallel pattern.

when compiled with our work versus CLANG-YKT, GCC-OMP, and GCC-ACC. Almost all benchmarks run faster with our compiler, with an average speedup of 1.5×, 1.6× and 4.0× respectively. These results show that our baseline is highly performant.

Next we compared the speedup against these compilers on the nested pattern. Our compiler is 2.1×, 7.0× and 7.6× faster than CLANG-YKT, GCC-OMP, and GCC-ACC respectively. None of the benchmarks are significantly slower when compiled with our compiler.

Across these experiments the Master-Controlled State Machine approach of CLANG-YKT performs better than the Basic Block Neutering approaches implemented in GCC. The latter approach shows significant performance degradation as compared to the SPMD pattern. It is possible that this is a quality-of-implementation issue but we believe the poor performance is an indicator of the difficulties in forcing ***fork-join*** parallelism onto the CUDA programming model.

Our results show that it is possible to achieve significant performance improvement when exploiting nested parallelism using the ***fork-join*** model but it requires going beyond the CUDA model and using the techniques of warp specialization and named barriers that we have proposed. Moreover every compiler we evaluated fails to successfully compile and run all benchmarks, which we believe is due to the brittle nature and complexity of their mapping techniques.

## VII. Conclusions

We have presented an open source OpenMP 4.5 compliant compiler for GPUs. As part of this work we have proposed a novel code generation technique to map the ***fork-join*** programming model to GPUs. Compared to existing work our approach is simpler to implement, robust, and produces high performing GPU code. The ideas presented in this paper are general and can be applied to other SIMT architectures. The insights in this paper should also enable further research into parallel programming models that is richer than the narrowly focused CUDA model.

## References

[1] O. R. L. C. Facility, "TITAN: Built for science," http://www.olcf.ornl.gov/titan/, 2013.

[2] S. N. S. Center, "Piz Daint," http://www.cscs.ch/computers/piz_daint/index.html, 2016.

[3] O. R. L. C. Facility, "SUMMIT," https://www.olcf.ornl.gov/summit/, 2017.

[4] L. L. N. Laboratory, "CORAL/Sierra," https://asc.llnl.gov/coral-info, 2017.

[5] NVIDIA, "The CUDA C programming guide," https://docs.nvidia.com/cuda/cuda-c-programming-guide/, 2017.

[6] K. Group, "Open computing language," https://www.khronos.org/opencl/, 2017.

[7] OpenACC.org, "The OpenACC® API," http://www.openacc.org/sites/default/files/OpenACC_2pt5.pdf, 2015.

[8] O. ARB, "The OpenMP® Application Program Interface," http://www.openmp.org/mp-documents/openmp-4.5.pdf, 2017.

[9] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, 2009.

[10] C. Shen, X. Tian, D. Khaldi, and B. Chapman, "Assessing one-to-one parallelism levels mapping for OpenMP offloading to GPUs," in *Proceedings of the 8th International Workshop on Programming Models and Applications for Multicores and Manycores*, 2017, pp. 68–73.

[11] J. Kim, Y.-J. Lee, J. Park, and J. Lee, "Translating OpenMP device constructs to OpenCL using unnecessary data transfer elimination," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '16, 2016.

[12] C. Liao, Y. Yan, B. R. de Supinski, D. J. Quinlan, and B. Chapman, "Early experiences with the openmp accelerator model," in *OpenMP in the Era of Low Power Devices and Accelerators*. Springer, 2013, pp. 84–98.

[13] S. Lee, S.-J. Min, and R. Eigenmann, "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*, 2009.

[14] N. Sidwell, "OpenACC implementation in GCC." https://www.youtube.com/watch?v=SBX6_K1AD7s, 2015.

[15] S. Lee and J. S. Vetter, "OpenARC: Extensible openacc compiler framework for directive-based accelerator programming study," in *Proceedings of the First Workshop on Accelerator Programming Using Directives*, ser. WACCPD '14, 2014.

[16] C. Bertolli, S. F. Antao, A. E. Eichenberger, K. O'Brien, Z. Sura, A. C. Jacob, T. Chen, and O. Sallenave, "Coordinating GPU threads for OpenMP 4.0 in LLVM," in *Proceedings of the 2014 LLVM Compiler Infrastructure in HPC*, ser. LLVM-HPC '14, 2014, pp. 12–21.

[17] M. Bauer, H. Cook, and B. Khailany, "CudaDMA: Optimizing GPU memory bandwidth via warp specialization," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11, 2011.

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

[19] Y. Yang and H. Zhou, "CUDA-NP: Realizing nested thread-level parallelism in GPGPU applications," in *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '14, 2014, pp. 93–106.

[20] T. R. W. Scogland and W.-c. Feng, "Design and Evaluation of Scalable Concurrent Queues for Many-Core Architectures," in *International Conference on Performance Engineering (ICPE)*, Austin, TX, USA, January 2015.

[21] J. Gómez-Luna, J. M. González-Linares, J. I. Benavides, and N. Guil, "An optimized approach to histogram computation on gpu," *Machine Vision and Applications*, vol. 24, no. 5, pp. 899–908, 2013.

[22] C. Benchmarks, "Haccmk," https://asc.llnl.gov/CORAL-benchmarks/Summaries/HACCmk_Summary_v1.0.pdf, 2017.

[23] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis and transformation," San Jose, CA, USA, Mar 2004, pp. 75–88.

[24] I. Research, "Clang/llvm compiler for gpus," https://github.com/clang-ykt, 2017.

[25] ——, "Clang-ykt compiler for gpus," https://github.com/clang-ykt/clang/tree/version_01, 2016.

[26] G. D. Community, "Gcc 7.1," https://gcc.gnu.org/gcc-7/, 2017.

[27] R. Reyes, I. López-Rodríguez, J. J. Fumero, and F. de Sande, *accULL: An OpenACC Implementation with CUDA and OpenCL Support*, 2012, pp. 871–882.

[28] X. Tian, R. Xu, Y. Yan, Z. Yun, S. Chandrasekaran, and B. Chapman, *Compiling a High-Level Directive-Based Programming Model for GPGPUs*, 2014, pp. 105–120.

[29] J. M. Bull, "Measuring synchronisation and scheduling overheads in OpenMP," in *In Proceedings of First European Workshop on OpenMP*, 1999, pp. 99–105.