Implementing OpenMP's SIMD Directive in LLVM's GPU Runtime

Eric Wright
efwright@udel.edu
University of Delaware
Newark, Delaware, USA

Johannes Doerfert
jdoerfert@llnl.gov
Lawrence Livermore National Laboratory
Livermore, California, USA

Shilei Tian
shilei.tian@stonybrook.edu
Stony Brook University
Stony Brook, New York, USA

Barbara Chapman
barbara.chapman@stonybrook.edu
Stony Brook University
Stony Brook, New York, USA

Sunita Chandrasekaran
schandra@udel.edu
University of Delaware
Newark, Delaware, USA

ABSTRACT

GPUs support three levels of parallelism: thread blocks, warps (or wavefronts) within a block, and threads within a warp. Some GPU programming models allow the use of all three of these levels, such as OpenMP offloading with the *teams*, *parallel*, and *simd* directives. However LLVM/OpenMP does not support *simd* and only uses two levels, thread blocks and all threads within a block. For codes with three explicit layers of parallelism this can decrease performance and potentially require restructuring of the application. In this work we present our design and implementation of the OpenMP *simd* directive in LLVM's OpenMP GPU runtime, which includes both CPU-centric and GPU-centric execution models. We evaluate our prototype using kernels and a few proxy applications showing a performance improvement ranging from 1.3x to 3.5x depending on the benefit the kernels receives from such an optimization. Thus, this work enables real-world applications with three explicit layers of parallelism to expose to better exploit the full benefits of GPU architecture.

CCS CONCEPTS

• **Computing methodologies → Parallel programming languages**; • **Software and its engineering → Runtime environments**; *Source code generation.*

KEYWORDS

OpenMP offloading, LLVM, SIMD, GPU

ACM Reference Format:
Eric Wright, Johannes Doerfert, Shilei Tian, Barbara Chapman, and Sunita Chandrasekaran. 2023. Implementing OpenMP's SIMD Directive in LLVM's GPU Runtime. In *52nd International Conference on Parallel Processing (ICPP 2023), August 07–10, 2023, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3605573.3605640

1 INTRODUCTION

There are several ways to program GPUs: Using GPU-enabled libraries such as MAGMA [14] or AmgX [21]. Using a kernel-based programming language such as CUDA or HIP. Or, using a directive-based language such as OpenACC and OpenMP. While direct GPU programming provides the user with control and conceptual means to utilize the GPU to the fullest, implementations, especially for portable directive-based languages, have a hard job translating programming language semantics to the device. To provide full language support as early as possible, it is not uncommon that features are therefore implemented in a "conservative" way, e.g., non-observable parts are ignored. Filling such "implementation holes" becomes increasingly important as growing heterogeneity calls for portable programming models while efficient use of complex devices requires accurate low-level control. In this work we will look at OpenMP offloading by extending the LLVM/OpenMP GPU implementation with a third level of parallelism that can be controlled by the user.

OpenMP has provided `target` directives for GPU programming since version 4.0. Further, the OpenMP standard provides directives to mark three conceptually different kinds of parallelism that are often thought of as a hierarchy: The `teams` directive spawns a league of teams, each with one initial (main) thread. In the absence of a `teams` directive, e.g., in most host codes, there is one implicit team started at the program beginning. The `teams` directive is often paired with the `distribute` directive which splits the iterations of associated loops across teams. Next, the `parallel` directive creates a team of threads. The `parallel` directive is often paired with the `for` directive which splits the iterations of associated loops across the threads in the team. Lastly, the `simd` directive specifies that the associated loop iterations can be executed in lockstep, e.g., via vector instructions.

These parallel levels are not unique to OpenMP as other languages provide equivalent concepts. OpenACC [1] uses "gang", "worker", and "vector" directives, Kokkos [15] uses "team", "thread", and "vector", and alpaka [31] uses "grid", "blocks", and "threads". The underlying reason all these programming models provide these levels can be found in the (early) GPU architecture. NVIDIA GPUs provide the user with thread blocks onto which "gangs", "teams", or "grids" are mapped. Each thread block contains warps onto which "workers", "threads", or "blocks" are mapped. The lanes in a warp (used to) be executed in lockstep fashion which matches the idea of “vector” execution.

Since it is harder to manage more layers of parallelism it is often the case that the innermost vector level is simply folded into the thread layer. In this mode each warp lane is associated with a thread and the threads use vector length one. This embedding is beneficial if the user only utilizes two parallel levels but most (reg. Section 2) OpenMP implementations eliminate the SIMD layer unconditionally. For codes with three explicit layers of parallelism this can decrease performance and potentially require restructuring of the application. The performance penalty can be significant if the thread level does not provide enough parallelism, or if there is high divergence between threads. Similarly, performance suffers if data access patterns are neither uniform nor consecutive with regards to worksharing loops (*#pragma omp for*).

In this work we will discuss the complexities and opportunities of a third level of parallelism in the LLVM/OpenMP GPU runtime. We describe how synchronization, data management and the GPU thread execution need to be adjusted to support all three levels of the host-centric OpenMP execution model.

The contributions of this work include:

- An extended LLVM/OpenMP GPU runtime library with support for three distinct levels of parallelism.
- Lowering of the OpenMP `simd` directive into control code that employs multiple warp lanes for concurrent execution of the loop iterations.
- Support for two conceptually different execution modes: generic-SIMD and SPMD-SIMD. The former matches the CPU model in which vector lanes are inactive if not used while the latter is aligned with the GPU model in which all warp lanes are active (almost) all of the time.
- An evaluation of our prototype on small kernels as well as HPC proxy applications that mirror real-world science codes.

Known limitations are:

- Our prototype supports NVIDIA GPUs at this point. The gap towards AMD GPU support is discussed in detail.
- Due to the non-availability of `simd` support by OpenMP offloading compilers the selection of existing codes with three levels of parallelism is quite limited.
- Our implementation cannot yet automatically translate generic-SIMD code into SPMD-SIMD code but the conceptual optimization is still described.

2 RELATED WORK

OpenMP has supported GPU offloading since the 4.0 standard with the inclusion of new `target` and `target data` directives, and has been extended and improved in subsequent OpenMP versions. User experiences of applying OpenMP target offloading can be found in some of the recent work including using the SPEChpc2021 benchmarking suite [7], other applications including HPGMG [10], miniMD [23], UK mini-apps [20], LULESH [18], among others. Work in [9, 24] discusses at length experiences gained and practices adopted from OpenMP hackathons when applying offloading features on HPC applications and mini-apps based on different computational motifs (BerkeleyGW, WDMApp/XGC, GAMESS, GESTS, and GridMini) targeting heterogeneous systems.

The OpenMP offloading support for GPUs in LLVM can be traced back to the two works discussed in [4, 5]. The (PGI) Fortran front-end, known as Flang, supported OpenMP offloading via the LLVM OpenMP runtime [22]. Since then, researchers have been working on compiler and runtime optimization for LLVM OpenMP. The first front-end-based optimizations for NVIDIA GPUs that can avoid idle threads and reduce register usage was introduced in [3]. Work in [11] presented the TRegion interface which delays the discovery of SPMD regions to compiler middle end, contrary to the front-end based approach used before, which can support more kernels to execute in SPMD mode. Runtime support for concurrent execution of OpenMP target tasks was introduced in [26]. Results in [16] discusses OpenMP-aware program analyses and optimizations that allow efficient execution of the generic, CPU-centric parallelism model provided by OpenMP on GPUs. A co-design methodology is presented in [12] for optimizing applications using a specifically crafted OpenMP GPU runtime inducing near-zero overhead in most cases.

There also exists several works describing implementations of SIMD parallelism within OpenMP for CPU-based architectures. An extension of OpenMP to generate explicit SIMD instructions is described in [6]. Early implementations of a possible `simd` directive within OpenMP is explored in [19] and [8]. The SIMD instruction generation in various compilers, including Clang/OpenMP, is analyzed in [29]. An OpenMP SIMD implementation of the VASP code is described in [28].

3 BACKGROUND

OpenMP offloading utilizes a host-device execution model where the host (CPU) schedules and synchronizes target tasks, in the form of *kernel*s, and handles memory allocation and movement between the host and target devices (e.g. GPUs). Computational kernels are executed on the device by launching a league of *team*s. Each team has a *main thread* that will begin executing the code region contained by the `teams` directive. Additional *worker thread*s can be spawned by using the `parallel` directive. There are three worksharing constructs: `distribute`, `for` and `simd`. `distribute` schedules loop iterations across the league of teams, `for` schedules loop iterations across threads within a team and `simd` uses single instruction multiple data (SIMD) parallelism for the loop.

GPU execution models utilize a similar structure with multiple streaming multiprocessors (SM), each containing several parallel work units (warps for NVIDIA GPUs and wavefronts for AMD GPUs) that are able to execute simultaneously. A simple mapping of the OpenMP model to GPU hardware is a team per each SM, and threads within the team to hardware threads within the SM. An example mapping is shown in Fig. 1.

The `simd` construct specifies that the attached loop should be executed using SIMD parallelism. For GPUs this is typically done using single instruction, multiple thread (SIMT) parallelism, meaning that multiple threads within a work unit execute the same instruction. This means that in terms of GPU offloading a `simd` loop should be executed in parallel by a set of adjacent threads. We can achieve this by separating the threads in each team into distinct groups. These *SIMD group*s will contain a single main thread and

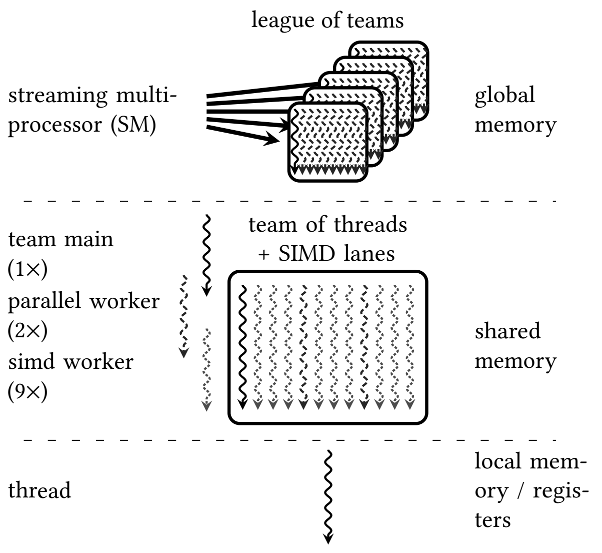

Figure 1: Simplified mapping of the OpenMP programming model to the GPU. Top row: Outermost parallelism across streaming multiprocessors (SMs) onto which OpenMP teams are mapped. All threads on this level share the global memory. Middle row: A single SM corresponding to a team of threads (team main + parallel workers) and, through this work, also SIMD vector lanes (simd workers) in OpenMP. Both shared and global memory are accessible by these threads. Bottom row: A single GPU/OpenMP thread which has exclusive access to local memory and registers. Adapted from Fig. 2 from [16]

multiple workers. The main thread executes `parallel` regions and all threads in the group execute `simd` loops.

3.1 OpenMP "Generic" CPU-centric Execution Model

The OpenMP programming model is a CPU-centric model that allows for sequential and parallel regions to be used interchangeably. This aligns well with a CPU execution where alternating between single-threaded execution and spawning additional threads is fairly easy. However, the GPU execution model generally requires all threads to begin execution at the start of the kernel and stay active until termination. In a kernel-language, sequential regions of code would require guarding to assure that only a single thread executes them, and any side-effects created during that region need to be communicated to other threads.

Prior work [5] discusses several problems in creating a portable solution to this in LLVM/Clang. In the case of parallel regions existing within branching if/then statements all threads will need to handle all potential paths that could be taken to ensure that all threads reach the correct parallel region. Handling the thread guarding needed for branching paths require extensive code generation, which is antithetical to Clang's design, and limits portability to other compilers.

An implementation that utilizes a state machine where threads alternate between idle, during sequential regions, and active, during parallel regions is described in [5]. A single thread is designated as the team main thread and will run the user code. All other threads are considered as worker threads and will enter the state machine where they will encounter a thread barrier and become idle until the main thread encounters a parallel region. This solves the problem of branching paths since only one thread needs to traverse the branching statements. When a parallel region is encountered some variables may need to be communicated to other threads if they are side-effects from the sequential code. The main thread will specify which parallel region was encountered before completing the thread barrier, signaling to the workers that they should become active and begin executing the parallel region. This implementation is what is currently used in LLVM/OpenMP.

3.2 OpenMP "SPMD" GPU-centric Execution Model

Work in [27] introduces an execution mode in the IBM XL C/C++ compiler that avoids the generic state machine when all threads can execute in parallel. This new mode is referred to as single program multiple data (SPMD) mode and has since been upstreamed into the LLVM/Clang. The key characteristic of SPMD mode is the assertion that all threads can safely execute the target region and will encounter the same parallel regions and any sequential regions of code will not produce side-effects. The simplest case for when SPMD is applicable is when all affected OpenMP regions are tightly nested, since this means there is no sequential code between the parallel regions. This allows OpenMP to behave similarly to GPU kernel-languages where all threads are active at the beginning of the kernel.

SPMD mode is further extended in [16] to be applicable to a larger variety of codes by introducing thread guarding of sequential code regions. The work introduces an inter-procedural analysis at the LLVM IR level to check sequential regions for potential side-effects that can be eliminated using thread guarding and variable broadcasting. If these guarded regions create values needed outside of the region, then these values would be broadcasted to the other threads. The *SPMDization* of these codes avoid the use of the generic state machine at the cost of additional synchronization in the guarded regions and data broadcasting.

4 LLVM CODE GENERATION FOR OPENMP LOOPS

This section will explain our methodology for generating LLVM IR for handling OpenMP worksharing loops, and the additions required specifically for `simd` loops. This code generation isolates the bodies of loops into "loop tasks," allowing these tasks to be passed as variables into LLVM's GPU runtime which then facilitates scheduling of these tasks on the appropriate threads. Such a method removes the burden of intensive parallel code generation and instead focuses on a more robust runtime library for handling GPU parallel tasks.

4.1 Interfacing with the GPU Runtime Through the OMP IR Builder

LLVM's *OpenMP IR Builder* is used to generate OpenMP target code and to interface with the LLVM/OpenMP runtime library. This tool is designed to be front-end independent and allows for a generalized approach to creating parallelism with OpenMP without requiring a compiler to do extensive parallel code generation. A compiler may interface with the OpenMP IR Builder by creating callback functions, which handle certain parts of the code generation while the IR Builder will generate the code needed for OpenMP parallelism, including runtime library function calls.

We have added new functions to the OpenMP IR Builder to generate code for OpenMP worksharing loops. This requires two callback functions: 1) to generate the trip count of the loop, and 2) to generate the body of the loop. The generated loop body will later be isolated and moved into a separate function in a process called *outlining*, which allows the body to be passed into the OpenMP runtime by using the *outlined function* as a pointer. This function represents the task that a single thread would do to execute a single iteration of the loop. Then the runtime will handle work distribution across threads and ensure that all iterations are executed.

Since the outlined function may reference variables that are no longer in the correct scope, these variables must be passed to the function as arguments. They are aggregated into a structure and passed as a singular payload to the outlined function. The payload is packed before the runtime function call and unpacked within the outlined function. Special consideration for these variables is needed for `simd` loops, since the generic execution mode requires the main thread to communicate these variables to the worker threads. In this case any variable used within the outlined region needs to exist in either shared or global memory such that it is accessible by all threads. During the outlining if any variables in the payload are local allocations from the encompassing parallel region, then those allocations are *globalized* [16], and the corresponding memory is deallocated at the end of the parallel region.

4.2 OpenMP Worksharing Loops in Clang

We have altered Clang's code generation for OpenMP `simd` loops to instead use our new function in the OpenMP IR Builder. The two key requirements that need to be met to create a `simd` loop is a callback function to generate LLVM IR for both the trip count of the loop and the body of the loop. Clang's *OpenMPSimdDirective* class is used to represent the OpenMP `simd` directive. Since this class is a loop directive it contains an *OMPCanonicalLoop* node as one of its children. The OMPCanonicalLoop has some built-in methods that are particularly useful for determining the trip count of the loop as well as resolving the loop variable.

The OMPCanonicalLoop is used to generate the LLVM IR for the loop trip count callback. Then, in the body callback a local allocation is created for the loop variable, and the OMPCanonicalLoop is used to initialize that loop variable based on the current loop iteration number. Lastly, Clang emits the *CompoundStmt* which includes the body of the loop.

While our work uses Clang as the front-end, the changes described would be applicable to any potential front-end wanting to use LLVM's GPU runtime. The front-end would have to provide code generation for the loop trip count and the loop body, similar to the methodology we have described. Then the OMP IR Builder would perform the loop task outlining and generate the appropriate runtime function calls. Loop scheduling is then performed from within the runtime.

4.3 Variable Globalization

For `simd` loops executing in the CPU-centric generic mode (reg. Section 5.3) some variables will need to be shared among threads and will be globalized. When a `simd` loop is generated, any variables used within the body of the loop (which are the variables that will be passed to the outlined function) are checked to determine which memory they reside in. If the variable is a local allocation (i.e only visible to the current thread) it will be replaced with a shared memory allocation. If the variable in untraceable (such as the case if its allocation is in another translation unit) then it will be copied to shared memory just before the `simd` loop is executed.

5 GPU RUNTIME IMPLEMENTATION

This section will discuss our implementation of three-leveled hierarchical parallelism in the LLVM/OpenMP runtime library, including new runtime functions for `simd` loops and significant changes to `teams` and `parallel` regions.

5.1 OpenMP/GPU Hardware Mapping

Fig. 2 shows an example of mapping a potential OpenMP target region onto a NVIDIA GPU. The `teams` directive spawns a league of teams and each team may contain many threads. This figure shows a single OpenMP team, but typically the league would contain several teams depending on the maximum number of concurrent threads the hardware allows. One thread within the team will be distinguished as the team main thread and will be in charge of running the code contained within the teams region.

The `parallel` directive spawns a team of threads to execute the parallel region. For this work, the team of threads is evenly divided into SIMD groups, where all threads within a group occupy the same warp. Our implementation does not allow for SIMD groups to encompass multiple warps as it extensively utilizes warp-level thread barriers. One thread in each group is designated as the *SIMD main thread* and will execute `parallel` regions while all other threads are *SIMD worker threads* and will execute `simd` loops.

The `simd` directive specifies that the attached loop should be executed using SIMD parallelism. For GPUs this is done by parallelizing across adjacent threads in a warp. For our implementation a `simd` loop distributes loop iterations across threads in the same SIMD group.

When a `teams` region is executing in generic mode an additional warp is assigned to act as the team main thread. This additional warp is needed for the purpose of thread synchronization as discussed in [17]. However, synchronization of threads within a SIMD group is done using a warp-level synchronization, which does not have the same limitations.

The following functions have been created to handle the mapping of the SIMD groups within the runtime:

- *getSimdGroup* returns which group the thread belongs to.
- *getSimdGroupId* returns the thread's ID within its group. SIMD main threads always have an ID of 0.
- *getSimdGroupSize* returns the size of the SIMD group. All SIMD groups are the same size and the size could differ between parallel regions.
- *isSimdGroupLeader* returns true if the thread is a SIMD main thread for its group.

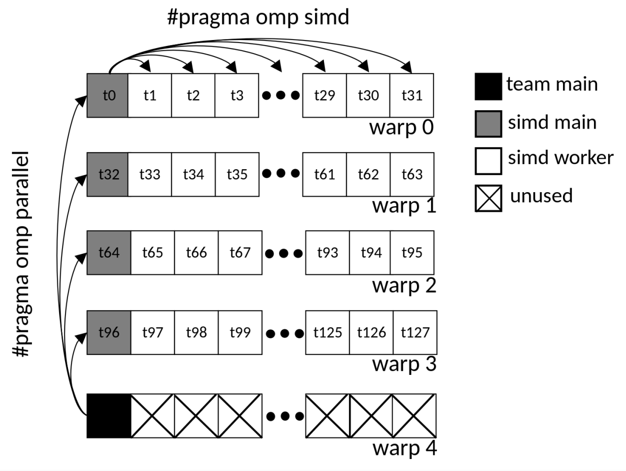

Figure 2: A possible mapping of a single OpenMP team on an NVIDIA GPU using four warps, totaling 128 threads for the computation. One SIMD group per warp, meaning one SIMD main and 31 SIMD workers per warp. One additional warp is included to act as the main thread in the team, which is required when the teams region is executing in generic mode.

- *simdmask* returns a bit-mask that represents which threads in the warp share a SIMD group with the thread.

5.2 Execution of OpenMP Offloaded Regions

At the start of an offloaded region all threads will begin by calling the *__target_init* function, which generally initializes the shared team state. It is also an important divergence point for the threads in the team. If the `teams` region executes in SPMD mode all threads will return from this function and immediately begin executing the user code. If the offloaded region will instead execute in generic mode only the team main thread will return to the user code while all the other threads will enter into a state machine where they will immediately encounter a thread barrier and remain idle until the main thread encounters an OpenMP `parallel` region.

`parallel` regions are handled through the runtime function *__parallel*. If running in SPMD mode all threads will reach the same call of *__parallel* and all threads will independently resolve the function pointer and handle the variable payload. In generic mode only the main thread will reach the *__parallel* function and the worker threads must be notified of the parallel region and any needed variables. When the main thread completes the thread barrier the worker threads will fetch the outlined function pointer and any variables used within the outlined function before executing it. Fig. 3 shows the *__parallel* function assuming the encompassing `teams` region is executing in SPMD mode.

Regardless of whether the `teams` region is SPMD or generic mode the runtime reaches another important divergence point in *__parallel* where each OpenMP `parallel` region can also be either SPMD or generic. In SPMD mode, all threads within the team will execute the `parallel` region, while in generic mode the main thread in each SIMD group will execute the `parallel` region and worker threads enter the SIMD state machine and immediately encounter a warp-level barrier and wait for a `simd` loop to be encountered. A call to the new runtime function *__simd* signifies a worksharing loop for SIMD parallelization. Fig. 4 shows this function with the two different execution modes.

```
void __parallel(
  void *fn, void **args, int64_t nargs, int32_t SPMD
) {
  if(SPMD) {
    // All threads execute region in SPMD mode.
    invokeMicrotask(fn, args, nargs);
    return;
  }
  if(isSimdGroupLeader()) {
    // Only simd mains execute region in generic mode.
    invokeMicrotask(fn, args, nargs);
    // Send termination signal to simd workers
    setSimdFn(nullptr);
    synchronizeWarp(simdmask());
  } else {
    // Simd workers enter the state machine.
    simdStateMachine();
  }
}
```

Figure 3: A portion of the *__parallel* runtime function showing the two different execution modes that parallel regions can be. If the `parallel` region is SPMD mode then all threads within the SIMD group will execute it. If it is instead generic mode only the SIMD main thread will execute the parallel region while all SIMD workers enter into a separate SIMD state machine.

```
void __simd(void *WorkFn, uint64_t TripCount,
            void **Args, uint32_t NumArgs) {
  if(isParallelSPMD()) {
    // In SPMD all threads in the SIMD group
    // execute the loop
    __workshare_loop_simd(WorkFn, TripCount, Args);
    synchronizeWarp(simdmask());
    return;
  }

  // In generic SIMD main thread sets up the
  // group state and signals the workers
  setSimdFn(WorkFn, TripCount);
  void **GlobalArgs;
  __begin_sharing_simd_args(&GlobalArgs);
  for(uint32_t i = 0; i < NumArgs; i++)
    GlobalArgs[i] = Args[i];
  synchronizeWarp(simdmask());
  __workshare_loop_simd(WorkFn, TripCount, GlobalArgs);
  synchronizeWarp(simdmask());
}
```

Figure 4: Runtime function for OpenMP `simd` loops in SPMD or generic mode. If the `parallel` region is generic mode then all variables needed within the `simd` loop must be shared from the SIMD main thread, and the SIMD workers must be notifed of what loop should be executed and for how many iterations. If the `parallel` region is instead in SPMD mode then this information is already local to each thread and no communication needs to occur.

5.3 CPU-centric Generic Model

Fig. 5 shows how each thread functions within the runtime. When the threads encounter an OpenMP `parallel` region that is executing in generic mode the threads will split into two possible paths, similar to `teams` generic mode. Threads that are designated as SIMD main will begin executing the `parallel` region user code. Fig. 2 shows a possible configuration of these SIMD mains with one main thread per warp, however it is possible to have multiple SIMD mains per warp. Threads that are designated as SIMD workers will enter the SIMD state machine and become idle while waiting for the main thread to encounter a `simd` loop. This is done through a warp-level barrier using a bit-mask to identify all threads within the same SIMD group. Fig. 6 shows the implementation for this state machine.

When the SIMD main thread encounters a call to *__simd*, it updates the SIMD group state with some information about the current `simd` loop, such as a function pointer that references the outlined function to be executed, the trip count of the loop and the addresses of all variables needed within the outlined function. When the SIMD main thread reaches the end of the current parallel region it sets the outlined function pointer in the SIMD group state as a *nullptr* which signifies a termination signal, and then notifies the workers through the warp synchronization. After this, all threads within that SIMD group will exit and run into a team-level barrier, where they will wait for all threads in all SIMD groups to finish the `parallel` region.

5.3.1 *Variable Sharing.*

When running in generic mode variables used within `parallel` regions and `simd` loops need to be shared from the main thread to all worker threads. These variables are always stored as pointers such that each variable is a consistent size. A static allocation of memory is reserved in GPU shared memory exclusively for these variables. Prior to our work the only thread that would write to this shared memory was the singular team main thread. If more variables needed to be shared than what the pre-allocated memory could hold, a global allocation is created to hold the variables instead, with that memory being deallocated at the end of the parallel region.

Now that this variable sharing space is written to by the team main thread and all SIMD main threads, the size of this space is increased and the available space is divided evenly among the SIMD groups. If a SIMD group needs more space than what is available a global memory allocation is created instead, which means that each SIMD group will have a pointer which correlates to where variables are stored (either in the shared memory or in a new global memory allocation).

Additionally, the size of a SIMD group can differ among different parallel regions. As an example using NVIDIA GPUs, if a target region is launched using 128 threads across 4 total warps, the number of total SIMD groups would be in the range of $4 \leq NumGroups \leq 64$, with the threads per group being $2 \leq ThreadsPerGroup \leq 32$ (for 32 threads in a warp). If the group size is less than two then the parallel region would run on all threads in the team and all `simd` loops would execute sequentially. Different parallel regions can use a different number of threads per group which results in a varying number of groups. In a case where a large number of SIMD groups are used the variable sharing space is less likely to be able to fit all variables.

The most noteworthy change in hardware resource usage from our work comes in the form of these shared memory changes. Originally, 1,024 bytes of shared memory were reserved as the variable sharing space. We have increased this to 2,048 bytes to help accommodate the new SIMD groups. This number is subject to change as more experimentation is done to select a size that is tailored for a typical code utilizing `simd`. Additionally, shared memory usage in general is increased for codes using our generic SIMD implementation as variables needed within the `simd` loops need to be moved to shared memory to be accessible by all threads within the SIMD group. This will vary by code, and will also be mitigated completely when using SPMD mode (reg. Section 5.4).

5.4 Optimized GPU-centric SPMD Model

A *parallel* region using SPMD mode will be executed by all threads in the team. Unlike the generic mode, there is no difference between SIMD main and SIMD workers. All threads will allocate any variables local to the parallel region, determine the trip count of the loop, load the variable payload and call the *__simd* runtime function using the outlined function pointer. Since all of this information is now local to each thread there does not need to be any communication like in generic mode and variables local to the parallel region that are needed in a *simd* loop do not need to be moved to shared memory.

In the case where the *simd* directive in unused, *parallel* regions will always execute in SPMD mode with a SIMD group size of one. This signifies that only two levels of parallelism should be used and behaves identically to the current implementation of LLVM/Clang. Fig. 7 shows how SIMD worker threads handle both SPMD and generic modes.

Similar to *teams* regions, *parallel* regions executing in SPMD mode must not produce side-effects. In the case where all *simd* loops are tightly nested within the *parallel* region then no side-effects will occur. However, in any other scenario there may need to be some level of thread guarding and variable broadcasting to eliminate side-effects, such as [16] describes for *teams* regions.

5.4.1 *Towards AMD GPU Support.*

AMD GPUs introduce some limitations to our execution model. LLVM/OpenMP does not provide an implementation for wavefront-level barriers, making our SIMD generic mode implementation incompatible with AMD GPUs. For this reason, our implementation only currently supports SPMD mode for AMD GPUs. If a parallel region would run in generic mode all *simd* loops will run sequentially. There may be some possibilities to implement generic mode on AMD GPUs using alternate methods for the thread barrier, however we do not yet know the viability of such approaches and will need to be explored as a future direction.

5.5 SIMD Worksharing Loop Execution

Fig. 8 shows the implementation for executing `simd` loops within the runtime. The *WorkFn* variable is the outlined function which contains the body of the loop that each thread will execute. The *TripCount* variable is the total number of iterations that the loop should run for. Lastly, the *Args* variable is the payload passed into

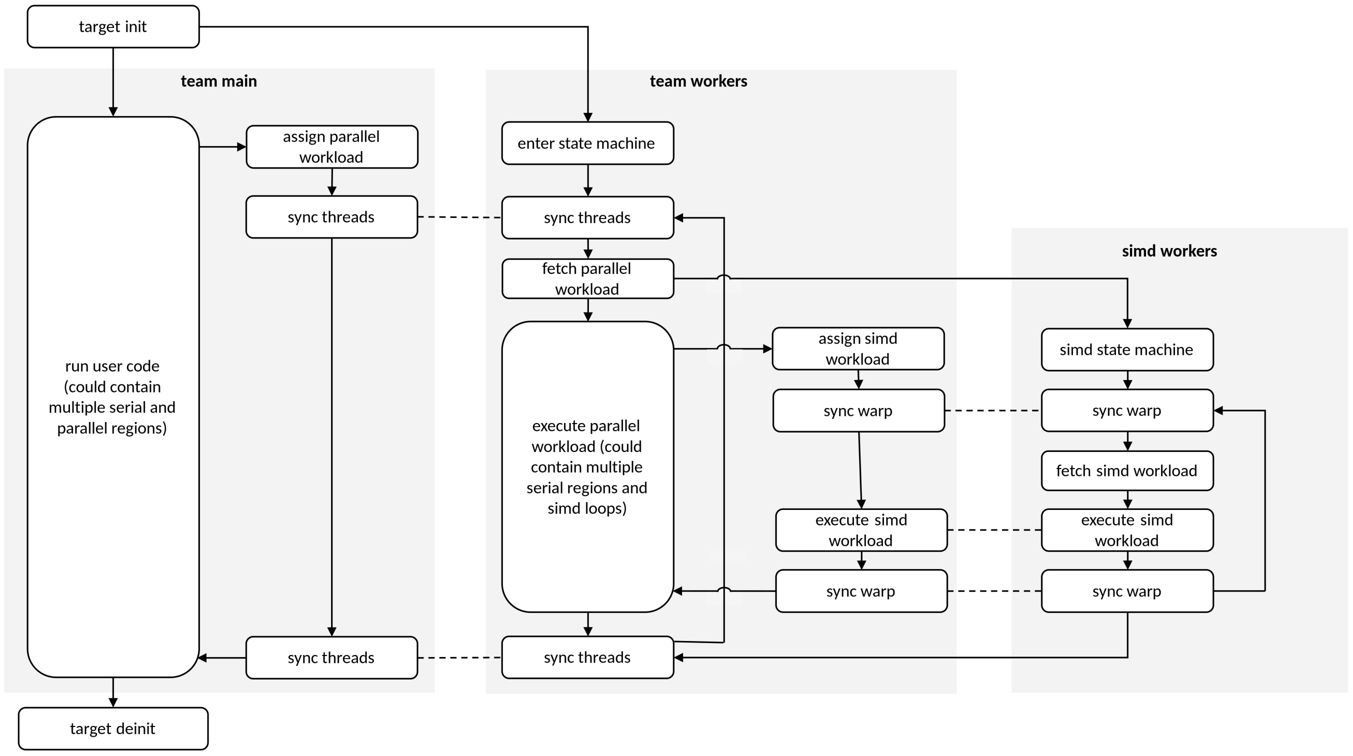

Figure 5: Program flow diagram of the GPU runtime assuming all regions are executed in generic mode.

```
void simdStateMachine() {
  do {
    void *WorkFn;
    void **FnArgs;
    uint64_t TripCount;

    // Wait for work
    synchronizeWarp(simdmask());
    getSimdFn(&WorkFn, &TripCount);
    if(!WorkFn) // Terminate at end of parallel
      return;

    // Fetch shared variables and execute loop
    getSimdArgs(&FnArgs);
    __workshare_loop_simd(WorkFn, TripCount, FnArgs);
    synchronizeWarp(simdmask());
  } while(true);
}
```

Figure 6: New state machine for SIMD workers when the containing `parallel` region executes in generic mode. Workers immediately reach a warp-level barrier. Once the SIMD main thread completes the barrier all workers will fetch the function pointer of the current `simd` loop, as well as any variables shared from the SIMD main thread. If the function pointer is a null pointer then the worker threads exit the state machine, as this signifies the end of the current `parallel` region.

the outlined function which may contain any number of pointers from global, shared or local memory.

Indirect calls using function pointers is normally costly. However, LLVM/Clang performs a front-end static analysis that creates an if/cascade, similar to a C switch statement, to compare the function pointer against known outlined regions, a methodology defined in [5]. In the case that the region is not known and cannot be placed in this if/cascade, such as regions in functions defined in other translation units, an indirect call is emitted as a fallback option.

6 RESULTS

This section will analyze performance results of our implementation on several selected codes that are known to benefit from using three levels of parallelism. Additionally, codes that do not receive any obvious benefit from this optimization are used to understand the performance penalty of our implementation if used unnecessarily to better give application developers guidance on using the `simd` directive.

6.1 Experimental Setup

All results are gathered on the Perlmutter supercomputer (NERSC-9) hosted by the National Energy Research Scientific Computer Center. Each computation node contains four NVIDIA A100 (40GB) GPU and one AMD EPYC 7763 CPU. All runs are collected using a single GPU and using the average of 10 runs. We use LLVM 16 with our custom modifications and CUDA version 11.7.

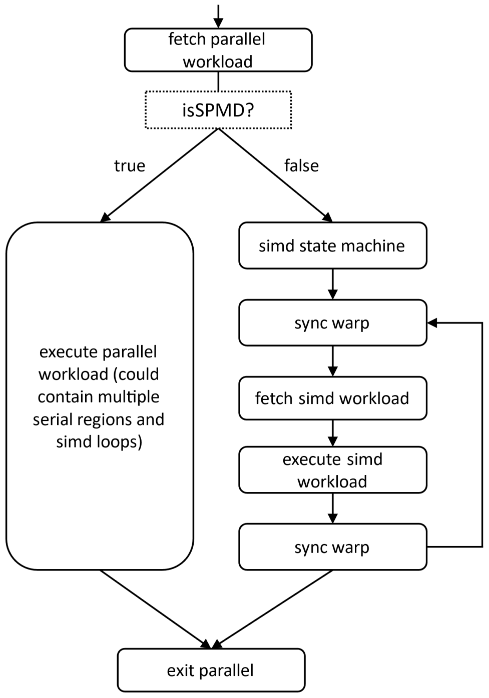

Figure 7: Flow diagram for SIMD worker threads upon encountering a parallel region. If the region should be executed in SPMD mode, worker threads will execute the entire region under the assumption that no side-effects will be produced. If the region should instead be executed in generic mode, worker threads will enter into the state machine and wait for a *simd* loop to be encountered.

```
void __simd_loop(
  void *WorkFn, uint64_t TripCount, void **Args) {
  uint64_t omp_iv = getSimdGroupId();
  synchronizeWarp(simdmask());
  while(omp_iv < TripCount) {
    WorkFn(omp_iv, Args);
    omp_iv += getSimdGroupSize();
  }
}
```

Figure 8: Function for executing `simd` loops. Each thread will execute a portion of the total iterations depending on SIMD group size.

6.2 Limitations

These results will primarily focus on the performance of the `simd` implementation in the different execution modes. We are not able to fully address shortcomings in the OpenMP runtime [25] outside of the additions made in this paper. Additionally, we cannot comment on the general performance of AMD GPUs in LLVM as it is not mature enough, and is not yet fully supported by our implementation.

SIMD is not universally useful on all codes. The codes used in these results were selected specifically with the knowledge that they are compatible with three levels of parallelism and will benefit from this optimization. Since `simd` is not a well-supported feature among OpenMP offloading compilers there are few benchmarks that utilize

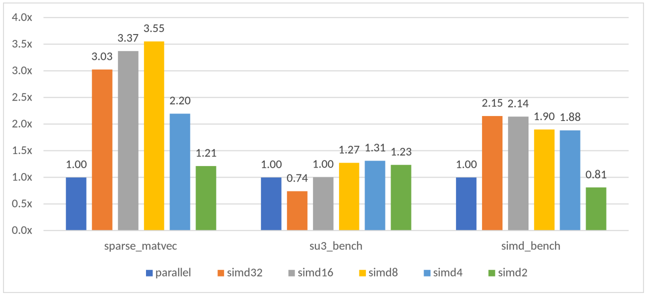

Figure 9: Results for various kernels comparing our `simd` implementation to the original two levels of parallelism. Experiments with all possible SIMD group sizes.

the `simd` directive, so several of the codes used here either had the `simd` directive added for this work or were adapted from OpenACC which has a mature three-leveled parallel implementation.

Additionally, with our new API for OpenMP worksharing loops, we do not yet have a compatible implementation for OpenMP *reductions*. Some potentially valuable experiments are impossible in the current stage of this project without the ability for reductions. Section 7 discusses the future direction we are taking with respect to the loop API and the OpenMP `reduction` clause.

6.3 SIMD Benefit Results

We observe the performance increase from several codes with known benefits from three-leveled parallelism. *sparse_matvec* is a sparse, matrix-vector product kernel adapted from and OpenACC code described in [2]. The inner-most loop of this kernel is relatively small, and varies based on the sparsity of the matrix. This kernel also originally used a data reduction on the product calculated in the inner-most loop, however reductions are not yet implemented for our new loop execution model, so instead we use a less efficient atomic update for the product.

To utilize the original two levels of parallelism we parallelize the outer loop with `teams distribute` and the inner loop with `parallel for`. With this structure the `teams` region will run in generic mode. For the three levels of parallelism we instead parallelize the outer loop with the combined `teams distribute parallel for` and the inner loop with `simd`, meaning the `teams` region will execute in SPMD mode and the `parallel` region in generic mode.

SU3_bench [13] has a small inner-loop with 36 total iterations that was originally executed serially by each thread. We now apply `simd` to this loop to allow for SIMD parallelism on the GPU. In this code both `teams` and `parallel` regions are SPMD mode.

We have also created a new benchmarking kernel that very closely fits the three levels of parallelism to gauge the performance increase that these optimizations could potentially provide in an ideal scenario. This kernel example has a small inner loop that fits into a single warp, but is not collapsible with the outer-loop nest. We parallelize the outer-loop with `teams distribute parallel for` and the inner-loop with `simd`. The `teams` region is SPMD while the `parallel` is generic mode.

Fig. 9 shows the relative speedup over the two-level parallel baseline. For *sparse_matvec* we see a maximum speedup of 3.5x. This

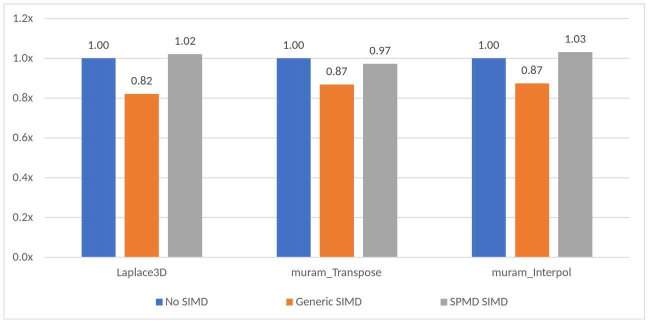

Figure 10: Relative speedup of the different SIMD execution modes. All `teams` regions are executed in SPMD mode. The performance similarity of "SPMD SIMD" and "No SIMD" suggests low performance overhead in our SPMD implementation.

is partially due to the teams region now being SPMD mode which means that extra warps are not needed for the team main thread, as well as using a much larger thread count per OpenMP team. We also see that a SIMD group size of 8 gives the best performance, likely due to it on average wasting fewer threads than other possible sizes due to the varying sparsity of the matrix, whereas the two-level parallel approach uses thread blocks of size 32, meaning that many threads may be idle.

SU3_bench sees a maximum speedup of 1.3x using a SIMD group size of 4 threads, however this is only slightly better than 2 and 8 thread group sizes. These group sizes likely performed better than other options by reducing the number of idle threads given the size of the `simd` loop. Lastly, our benchmarking kernel sees a speedup of 2.15x with a SIMD group size of 32 threads, but is very close in performance to a group size of 16.

These results highlight the possible improvement with SIMD parallelism. Not all codes will receive identical benefit from this optimization, but codes that cannot express efficient vector parallelism in a two-level parallel structure can see a speedup in the range of these example kernels.

6.4 Performance Cost of the Implementation

To understand the performance difference of the different execution modes we analyze several kernels that include three parallelizable loops. The execution modes of these kernels can be adjusted between generic and SPMD mode by changing whether or not the loops are tightly-nested. We have used three codes for this analysis: A simple three-dimensional heat diffusion kernel called laplace3d, and two kernels adapted from an OpenACC code called muram_transpose and muram_interpol [30]. We expect that these codes may see a small performance benefit from using a three-leveled parallel approach as it may slightly improve data-reuse, however, our main goal is to observe performance differences between generic and SPMD mode with our current implementation.

Fig. 10 shows the relative speedup of the different `simd` execution modes compared to the "No SIMD" version, which uses `teams` SPMD mode with two levels of parallelism. The number of teams and threads-per-team is kept consistent and the SIMD group size is 32 for all examples.

All regions executing in SPMD mode performs similarly to the "No SIMD" version, with laplace3d and muram_interpol seeing a marginal performance increase. Running in generic mode sees a roughly 15% slowdown, which is the penalty for using the state machine and the extra synchronization it needs. Overall, from these results we conclude that our SIMD implementation produces little-to-no performance overhead when executing in SPMD mode. Additionally, the overhead accrued when executing in SIMD generic mode is comparable to the overhead of the `teams` generic mode.

6.5 Developer Recommendations and Best Practices

From these results we put forward some general guidelines to aid any developers who would use the `simd` directive for their codes. In situations where SPMD mode can be utilized (i.e when `parallel` regions and `simd` loops are tightly nested) there is not a noteworthy penalty for using the `simd` directive. However, in situations where generic mode would be used instead one would have to weigh the benefit of such an optimization against the performance penalty of generic mode. In codes that truly benefit from `simd` (i.e when there exists a non-collapsible, but small inner loop) the benefit typically outweighs this cost.

Additonally, with the eventual inclusion of automatic conversion from SIMD-generic into SIMD-SPMD the cost of using `simd` in a nonoptimal case will be further minimized. However, it is still likely that even with proper SPMDization the included thread guarding and variable broadcasting would still see some amount of performance degradation, meaning that it is still likely best practice to use only two-leveled parallelism when all three levels are unneeded.

For choosing a `simdlen`, or SIMD group size, our best results were when we focused on reducing thread waste, choosing sizes that best evenly divide our loop trip count. When there are several viable group sizes we found that there still may be slight performance differences between them, depending on the code. It is likely best to experiment with the different options to see which fits the specific scenario best.

7 CONCLUSION AND FUTURE WORK

In this paper we have discussed our design and implementation of the OpenMP `simd` directive in LLVM's OpenMP GPU runtime with both CPU-centric and GPU-kernel execution models. This SIMD parallelism is an important optimization for codes that can efficiently express all three levels of available parallelism on GPUs, and we have shown a performance improvement on a variety of compatible codes in the range of 1.3-3.5x.

Many existing OpenMP offloaded applications are not able to utilize the `simd` directive as compilers do not offer support for this feature yet. It is in the immediate future direction to solicit further case studies, or alter existing codes, to continue exploring performance benefits and limitations of our model.

Additionally, we plan to extend the work from [16] to also support the SPMDization of `parallel` regions. This will make SPMD mode applicable to a wider range of codes, which will improve general performance as well as compatibility of our implementation to AMD GPUs.

We have also introduced a new API for OpenMP worksharing loops. In these results we have only explored `simd` loops, but `distribute`, `for` and combined loop constructs are supported. The API must be extended to include data reductions and loop collapsing, as these are common optimizations in OpenMP parallel codes. This will also expand the pool of applications that we can experiment with our new implementation.

ACKNOWLEDGMENTS

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. The research is also supported by the NSF under grant no. 1814609. The views and opinions of the authors do not necessarily reflect those of the U.S. government or Lawrence Livermore National Security, LLC neither of whom nor any of their employees make any endorsements, express or implied warranties or representations or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of the information contained herein. This work was in parts prepared by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-851052).

REFERENCES

[1] [n. d.]. OpenACC Specification. https://www.openacc.org/specification. Accessed: 2021-05-27.

[2] 2021. *OpenACC Programming and Best Practices Guide.* openacc-standard.org. 47–52 pages. https://www.openacc.org/sites/default/files/inline-files/OpenACC_Programming_Guide_0_0.pdf

[3] 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 *Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC).* 1–11.

[4] 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 *Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC).* 5:1–5:11.

[5] 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 *LLVM Compiler Infrastructure in HPC (LLVM-HPC).* 12–21.

[6] Con Bradley and Benedict R Gaster. 2008. Exploiting loop-level parallelism for SIMD arrays using OpenMP. In *A Practical Programming Model for the Multi-Core Era: 3rd International Workshop on OpenMP, IWOMP 2007, Beijing, China, June 3-7, 2007 Proceedings 3.* Springer, 89–100.

[7] Holger Brunst, Sunita Chandrasekaran, Florina M Ciorba, Nick Hagerty, Robert Henschel, Guido Juckeland, Junjie Li, Verónica G Melesse Vergara, Sandra Wienke, and Miguel Zavala. 2022. First Experiences in Performance Benchmarking with the New SPEChpc 2021 Suites. In *International Symposium on Cluster, Cloud and Internet Computing (CCGrid).* 675–684.

[8] Diego Luis Caballero de Gea. 2015. SIMD@ OpenMP: a programming model approach to leverage SIMD features. (2015).

[9] Barbara Chapman, Buu Pham, Charlene Yang, Christopher Daley, Colleen Bertoni, Dhruva Kulkarni, Dossay Oryspayev, Ed D'Azevedo, Johannes Doerfert, Keren Zhou, et al. 2021. Outcomes of OpenMP Hackathon: OpenMP Application Experiences with the Offloading Model (Part II). In *International Workshop on OpenMP (IWOMP).* 81–95.

[10] Christopher Daley, Hadia Ahmed, Samuel Williams, and Nicholas Wright. 2020. A Case Study of Porting HPGMG from CUDA to OpenMP Target Offload. In *International Workshop on OpenMP (IWOMP).* 37–51.

[11] Johannes Doerfert, Jose Manuel Monsalve Diaz, and Hal Finkel. 2019. The TRegion Interface and Compiler Optimizations for OpenMP Target Regions. In *International Workshop on OpenMP (IWOMP)*, Vol. 11718. 153–167.

[12] Johannes Doerfert, Atemn Patel, Joseph Huber, Shilei Tian, Jose M Monsalve Diaz, Barbara Chapman, and Giorgis Georgakoudis. 2022. Co-Designing an OpenMP GPU Runtime and Optimizations for Near-Zero Overhead Execution. In *International Parallel and Distributed Processing Symposium (IPDPS).* IEEE.

[13] Douglas Doerfler, Christopher Daley, and USDOE. 2020. SU3_bench: Lattice QCD SU(3) Matrix-Matrix Multiply Microbenchmark (SU3_bench) v1.0. https://doi.org/10.11578/dc.20200610.4

[14] Jack Dongarra, Mark Gates, Azzam Haidar, Jakub Kurzak, Piotr Luszczek, Stanimire Tomov, and Ichitaro Yamazaki. 2014. Accelerating Numerical Dense Linear Algebra Calculations with GPUs. *Numerical Computations with GPUs* (2014), 1–26.

[15] H Carter Edwards and Christian R Trott. 2013. Kokkos: Enabling Performance Portability Across Manycore Architectures. In *Extreme Scaling Workshop (XSW).* 18–24.

[16] Joseph Huber, Melanie Cornelius, Giorgis Georgakoudis, Shilei Tian, Jose Manuel Monsalve Diaz, Kuter Dinel, Barbara M. Chapman, and Johannes Doerfert. 2022. Efficient Execution of OpenMP on GPUs. In *International Symposium on Code Generation and Optimization (CGO).* 41–52.

[17] Arpith Chacko Jacob, Alexandre E Eichenberger, Hyojin Sung, Samuel F Antão, Gheorghe-Teodor Bercea, Carlo Bertolli, Alexey Bataev, Tian Jin, Tong Chen, Zehra Sura, et al. 2017. Efficient fork-join on GPUs through warp specialization. In *2017 IEEE 24th International Conference on High Performance Computing (HiPC).* IEEE, 358–367.

[18] Ian Karlin, Tom Scogland, Arpith C Jacob, Samuel F Antao, Gheorghe-Teodor Bercea, Carlo Bertolli, Bronis R de Supinski, Erik W Draeger, Alexandre E Eichenberger, Jim Glosli, et al. 2016. Early Experiences Porting Three Applications to OpenMP 4.5. In *International Workshop on OpenMP (IWOMP).* 281–292.

[19] Michael Klemm, Alejandro Duran, Xinmin Tian, Hideki Saito, Diego Caballero, and Xavier Martorell. 2012. Extending OpenMP* with Vector Constructs for Modern Multicore SIMD Architectures. *IWOMP* 7312 (2012), 59–72.

[20] Matt Martineau, Simon McIntosh-Smith, Carlo Bertolli, Arpith C. Jacob, Samuel F. Antao, Alexandre Eichenberger, Gheorghe-Teodor Bercea, Tong Chen, Tian Jin, Kevin O'Brien, Georgios Rokos, Hyojin Sung, and Zehra Sura. 2016. Performance Analysis and Optimization of Clang's OpenMP 4.5 GPU Support. In *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS).* 54–64.

[21] M. Naumov, M. Arsaev, P. Castonguay, J. Cohen, J. Demouth, J. Eaton, S. Layton, N. Markovskiy, I. Reguly, N. Sakharnykh, V. Sellappan, and R. Strzodka. 2015. AmgX: A Library for GPU Accelerated Algebraic Multigrid and Preconditioned Iterative Methods. *SIAM Journal on Scientific Computing* 37, 5 (2015), S602–S626.

[22] Güray Özen, Simone Atzeni, Michael Wolfe, Annemarie Southwell, and Gary Klimowicz. 2018. OpenMP GPU Offload in Flang and LLVM. In *Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC).* 1–9.

[23] Simon J Pennycook, Jason D Sewall, and Jeff R Hammond. 2018. Evaluating the Impact of Proposed OpenMP 5.0 Features on Performance, Portability and Productivity. In *International Workshop on Performance, Portability and Productivity in HPC (P3HPC).* 37–46.

[24] Swaroop Pophale, Dossay Oryspayev, B Chapman, B Pham, C Yang, C Daley, C Bertoni, D Kulkarni, E D'Azevedo, H He, et al. 2021. *Outcomes of OpenMP Hackathon: OpenMP Application Experiences with the Offloading Mode.* Technical Report. Brookhaven National Lab.(BNL), Upton, NY (United States).

[25] Shilei Tian, Jon Chesterfield, Johannes Doerfert, and Barbara Chapman. 2021. Experience Report: Writing A Portable GPU Runtime with OpenMP 5.1. In *International Workshop on OpenMP.*

[26] Shilei Tian, Johannes Doerfert, and Barbara M. Chapman. 2020. Concurrent Execution of Deferred OpenMP Target Tasks with Hidden Helper Threads. In *Languages and Compilers for Parallel Computing (LCPC).* 41–56.

[27] E. Tiotto, B. Mahjour, W. Tsang, X. Xue, T. Islam, and W. Chen. 2020. OpenMP 4.5 Compiler Optimization for GPU Offloading. *IBM Journal of Research and Development* 64, 3/4 (2020), 14:1–14:11.

[28] Florian Wende, Martijn Marsman, Jeongnim Kim, Fedor Vasilev, Zhengji Zhao, and Thomas Steinke. 2019. OpenMP in VASP: Threading and SIMD. *International Journal of Quantum Chemistry* 119, 12 (2019), e25851.

[29] Florian Wende, Matthias Noack, Thomas Steinke, Michael Klemm, Chris J Newburn, and Georg Zitzlsberger. 2016. Portable simd performance with openmp* 4. x compiler directives. In *Euro-Par 2016: Parallel Processing: 22nd International Conference on Parallel and Distributed Computing, Grenoble, France, August 24-26, 2016, Proceedings 22.* Springer, 264–277.

[30] Eric Wright, Damien Przybylski, Matthias Rempel, Cena Miller, Supreeth Suresh, Shiquan Su, Richard Loft, and Sunita Chandrasekaran. 2021. Refactoring the MPS/University of Chicago Radiative MHD (MURaM) model for GPU/CPU performance portability using OpenACC directives. In *Platform for Advanced Scientific Computing Conference (PASC).* 1–12.

[31] Erik Zenker, Benjamin Worpitz, René Widera, Axel Huebl, Guido Juckeland, Andreas Knüpfer, Wolfgang E Nagel, and Michael Bussmann. 2016. Alpaka–An Abstraction Library for Parallel Kernel Acceleration. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW).* 631–640.