Co-Designing an OpenMP GPU Runtime and Optimizations for Near-Zero Overhead Execution

Johannes Doerfert
Argonne National Laboratory
Lemont, USA
jdoerfert@anl.gov

Atemn Patel
University of Waterloo
Waterloo, Ontario, Canada
atmn.patel@uwaterloo.ca

Joseph Huber
Oak Ridge National Laboratory
Oak Ridge, USA
huberjn@ornl.gov

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

Jose M Monsalve Diaz
Argonne National Laboratory
Lemont, USA
jmonsalvediaz@anl.gov

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

Giorgis Georgakoudis
Lawrence Livermore National Laboratory
Livermore, USA
georgakoudis1@llnl.gov

***Abstract*—GPU accelerators are ubiquitous in modern HPC systems. To program them, users have the choice between vendor-specific, native programming models, such as CUDA, which provide simple parallelism semantics with minimal runtime support, or portable alternatives, such as OpenMP, which offer rich parallel semantics and feature an extensive runtime library to support execution. While the operations of such a runtime can easily limit performance and drain resources, it was to some degree regarded an unavoidable overhead.**

In this work we present a co-design methodology for optimizing applications using a specifically crafted OpenMP GPU runtime such that most use cases induce near-zero overhead. Specifically, our approach exposes runtime semantics and state to the compiler such that optimization effectively eliminating abstractions and runtime state from the final binary. With the help of user provided assumptions we can further optimize common patterns that otherwise increase resource consumption.

We evaluated our prototype build on top of the LLVM/OpenMP GPU offloading infrastructure with multiple HPC proxy applications and benchmarks. Comparison of CUDA, the original OpenMP runtime, and our co-designed alternative show that, by our approach, performance is significantly improved and resource consumption is significantly lowered. Oftentimes we can closely match the CUDA implementation without sacrificing the versatility and portability of OpenMP.

***Index Terms*—OpenMP, gpu, offloading, compiler optimization**

I. Introduction

Efficiently and effectively utilizing different HPC system becomes more complex as such systems grow in heterogeneity. Software stacks that are increasingly specialized towards vendor hardware make the situation worse. While some parallel programming models offer portable acceleration across various platforms, this portability can be associated with a price. A common source of overheads is a rich parallel semantic model and the complex runtime system needed to support it. Kernel languages, like CUDA and HIP, are very lean in an effort to avoid potentially slow features. OpenACC, a portable parallel programming model, generally follows this design principle too. OpenMP, on the other hand, has basically included its entire (naturally grown) host model into the target offloading realm. While this can simplify porting efforts and provides flexibility, it puts a heavy burden on the compiler and runtime system to offer complex features with minimal cost.

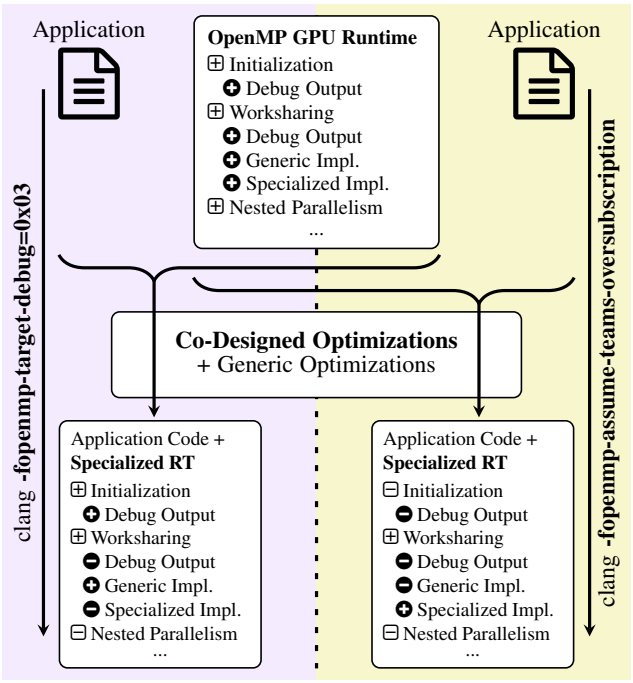


Fig. 1: Overview featuring the new OpenMP GPU Runtime as well as co-designed optimizations. Two separate compilations are shown (left and right) to illustrate how the application code but also the command line options will impact the features, and consequently overheads, that make it into the final binary.

In this work we present a novel OpenMP GPU runtime that was co-designed together with compiler optimizations. We show that one can close the gap between the performance and resource utilization of OpenMP compared to native GPU programming models for most common use cases. Our approach provides bare-metal performance for a "common core" of OpenMP functionality. The idea, as illustrated in Figure 1,

is that the model and the runtime can stay rich and extensive but *you only pay for what you actually use*. Less common features and use cases are still supported but they might result in decreased performance and increased resource consumption. Depending on the application code and the compilation flags, different features of the runtime will effectively be picked and included in the final binary. Functionality that was not used, or is not needed after optimization were performed, is statically pruned and consequently does not induce execution cost.

On top of efficient support of OpenMP features we also provide enhanced debugging and specialization options. The former allows to compile an application in debug or release mode as one would on the host. In debug mode runtime invariants are verified, user assertions are checked, and elaborate debugging options are available to be selected at execution time. When compiled in release mode these features are eliminated but with them also their cost. To further improve performance we provide the user with OpenMP extensions in form of pragmas, runtime calls, and compiler flags. As their usefulness is proven in practice we expect them to be adopted, in one form or another, into the OpenMP standard as well.

The main contributions of this work include:

- An open source implementation of an LLVM/OpenMP compatible GPU runtime in modern C++ and OpenMP 5.1 that is portable and resource efficient across various real world use cases.
- A collection of compiler optimizations integrated into the LLVM core optimization pipeline capable of eliminating or significantly reducing the overhead of OpenMP on GPUs in many common use cases.
- An evaluation of the remaining overheads, user facing assumptions to remedy some of them, and analysis on how others could be eliminated by future work.
- A detailed evaluation of our runtime, optimizations, and assumptions using multiple HPC proxy applications and benchmarks together with a discussion of the differences to their CUDA implementation.

Non-conceptual limitations of our work include:

- We did not redesign or optimize the reduction implementation in the LLVM/OpenMP GPU runtime. Reduction performance is therefore not expected to be impacted (much) by this work, nor is it evaluated.
- We are still in the process of integrating our work, both the runtime and the optimizations, into the LLVM community version. While some parts are available, current and future performance of the "new" LLVM/OpenMP GPU runtime can consequently vary.

The remainder of the paper is organized as follows. We discuss background in Section II, and introduce the design of our OpenMP GPU runtime in Section III. Next, we talk about the corresponding optimization passes and how they interact with the runtime. Evaluation with multiple HPC proxy applications and benchmarks is provided in Section V, followed by related works in Section VI. Before we conclude in Section VIII we discuss remaining challenges in Section VII.

II. Background

In this section, we will briefly introduce the LLVM/OpenMP GPU runtime, compilation and execution model.

A. LLVM/OpenMP GPU Runtime

When compiling from one language to another, there are usually constructs that are straightforward in the former and complicated or verbose in the latter. For example, a single OpenMP construct `#pragma omp parallel for` is lowered into a non-trivial amount of newly introduced code in the application, including runtime library call for certain functionality, such as dividing loop iterations.

The LLVM OpenMP GPU runtime library contains various functions the compiler needs in order to implement OpenMP semantics when the target is an Nvidia or AMD GPU. The original implementation in LLVM was written in CUDA [1] and compiled with Nvidia's NVCC to PTX assembler. Later to support AMD GPUs, the source was adapted to compile alternatively as HIP, which is close enough to CUDA syntax for the differences to be worked around with macros. Recently Tian et al. [2] ported the LLVM OpenMP GPU runtime library to OpenMP 5.1 using only minor extensions not available in the standard. Such a design allows for maximum re-usability between different targets, and further lowers the barrier to entry for future targets that only need to provide a few target specific intrinsics. While our OpenMP GPU runtime is largely built from scratch, it follows the same design principles.

B. LLVM/OpenMP GPU Compilation

When compiling an OpenMP offloading program using LLVM/Clang, the GPU runtime library is first linked into the user code as an LLVM bitcode library and then optimized together with the user application in LLVM IR. The application is then compiled to the target architecture and passed to the vendor tool-chain to create an application binary. Finally, the resulting binary kernel is inserted into the host code as a named symbol that will be loaded by the offloading plugin when the program executes.

C. LLVM/OpenMP GPU Execution Model

OpenMP offloading uses a host-centric mode of execution. The host (CPU) coordinates scheduling and synchronization of target tasks (i.e. kernels), as well as memory allocation and movement between the host and GPUs. The `teams` directive controls the creation of a *league of teams*. Each team begins with a single *main thread* that can spawn other *worker threads* using the `parallel` directive. The work-sharing directives `for` and `distribute` allow scheduling loop iterations across threads in a team and teams in a league, respectively.

GPU execution models have a similar hierarchical level of parallelism. GPUs are composed of multiple *streaming multiprocessors (SM)*, each capable of executing several threads grouped into scheduling units that execute at the same time (e.g. a *warp* in Nvidia GPUs and *wavefront* for AMD GPUs).

When lowering OpenMP onto GPUs it is common to map a team to an SM and the threads of a team to hardware threads

within the SM. Figure 2 shows this mapping between the GPU execution model and the OpenMP execution model used by LLVM. The figure also illustrates which memory kinds in the GPU are accessible by the thread(s) on each level.

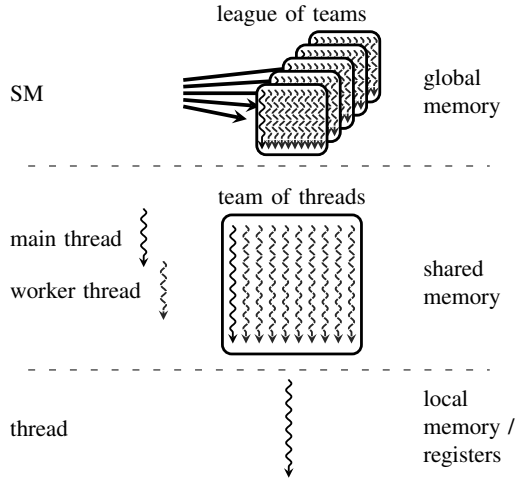


Fig. 2: Simplified mapping of the LLVM/OpenMP programming model to the GPU. Top row: Outermost parallelism across SMs onto which OpenMP teams are mapped. All threads on this level share the global memory. Middle row: A single SM corresponding to a team of threads in OpenMP. Both shared and global memory are accessible by these threads. Bottom row: A single GPU/OpenMP thread which has exclusive access to local memory and registers.

OpenMP uses a standard fork-join parallelism model where the main thread executes alone until an explicit parallel region is reached. This is in contrast with the GPU where all threads are active simultaneously. In order to maintain OpenMP semantics, a state machine first introduced in Bertolli et al. [3, 4] is used to schedule work from the main thread to the parallel worker threads. This execution mode is called generic mode. A separate single program multiple data (SPMD) mode is used when it is known that all threads are active for the entire target region, completely bypassing the need for a state machine.

III. OpenMP GPU Runtime

The new OpenMP GPU runtime was implemented almost entirely from scratch with portability and modularity in mind. It consists of roughly 2000 lines of C++ code with OpenMP pragmas, e.g., for memory placement. It does not include any system headers and can be compiled for both Nvidia and AMD GPUs. Similarly, it can be compiled for the CPU and executed in a "virtual GPU" [5]. Whenever target specific implementations are necessary we followed methodology outlined in Tian et al. [2] and utilized modern OpenMP 5.1 features, such as `begin declare variant`, for versioning.

A. SPMD-Mode Flag

We use a single boolean flag that is allocated in static shared memory to indicate if the kernel is executed in SPMD or generic-mode. The flag is set as part of the runtime initialization by the main thread of each team and will never be changed. The content is passed by-value to initialization functions to avoid reading the flag from memory before the first synchronization barrier, located after the initialization code, is reached by all threads.

B. Team ICV State

Each team has a shared internal control variable (ICV) state allocated in static shared memory. Each ICV state is initialized by the main thread during runtime initialization. It is later used by all threads that do not require an individual ICV state. The expectation is that (most) target regions do not require thread specific state at all. Hence, accessing the team state should be fast, or preferably completely optimized out at compile time.

C. Thread States and the Thread ICV State

Each thread in a team has a pointer allocated in static shared memory that indicates where the most recent ICV state for that thread can be found. During runtime initialization, all threads initialize their pointers with `NULL` to indicate they do not require a thread specific state, but rather utilize the shared team ICV state. If at any point a thread enters a nested data environment alone, or modifies its state such that the modification is not valid for all other currently active threads, a new thread specific ICV state is created. In contrast to the pre-allocated team ICV state and the array holding the thread state pointers, individual thread ICV states are allocated on demand via the shared memory stack (ref. Section III-D). A new individual state is copied from the most recent one, which might be the team ICV state, and appended to a list of individual states representing nested data environments. Allocating and tracking ICV states is in general costly, but required unless it can be statically determined that the state is not used and can be eliminated (ref. Section IV-B).

D. Shared Memory Stack

The shared memory stack is allocated in shared memory and initialized to track the usage of the stack by each thread. The stack is used whenever the runtime requires memory that can be shared between the threads in a team. Global memory, allocated via `malloc`, is used as a fallback when the stack is full. In particular there are two uses cases right now: (1) Variable globalization performed by LLVM/Clang to allow multiple threads access to "local" variables (ref. Section IV-A). (2) Individual thread ICV states that are allocated at runtime. If all uses of the shared memory stack are eliminated through optimizations (ref. Section IV) the shared memory stack can be removed entirely. The runtime also supports the use of dynamic shared memory to change the amount of shared memory at runtime.

E. Example Team and Thread ICV States

The interplay between the thread states array, the team ICV state, and potential thread ICV states is illustrated in Figure 3. The code in Figure 4 is an example input that would result in the depicted state. While the nested parallel regions

are sequential, they create a new data environment which is tracked through individual thread ICV states. It is important to note that nesting parallel regions in the current version of the runtime is strongly discouraged as it not only causes runtime allocation of the thread ICV states but also prevents state elimination (ref. Section IV-B).

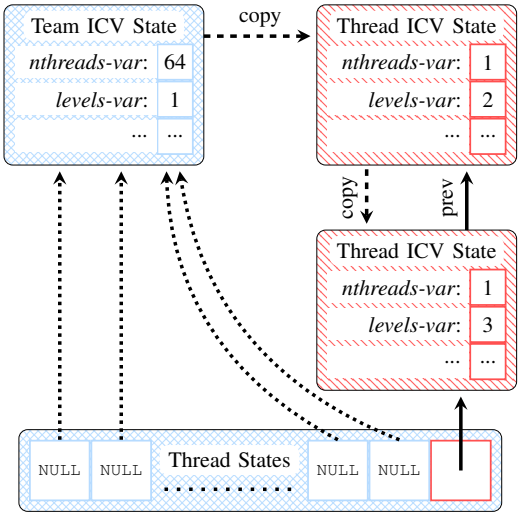


Fig. 3: Interplay between states that keep OpenMP internal control variables (ICVs) available for all threads. Any GPU state is tracked per team and located in shared memory. The team state and thread states array ▨ are pre-allocated while the thread state instances ▨ are allocated on-demand. Lookups of states without an explicit thread state instance (indicated as `NULL`) are redirected transparently to the team state.

F. Distribute and Work-sharing Loops

The runtime uses new implementations for work-sharing loops designed to resemble the execution of CUDA-style thread distributions. Typically, the compiler will generate work-sharing runtime calls individually for `distribute` and `for` work-sharing directives. The new implementation uses a combined scheme to better control the scheduling between the threads and teams in the kernel. The new implementation for the worksharing constructs is shown in Figure 5.

This implementation breaks compatibility with the OpenMP host runtime API, but also reduces the overhead through compiler optimizations. For example, if the number of iterations is known to be less than the number of threads in a team (for work-sharing), or the number of threads across all teams (for `distribute parallel for`), that is, each team/thread will

```
#pragma omp parallel num_threads(64)
if (omp_get_thread_num() == omp_get_team_size()-1){
 #pragma omp parallel num_threads(32)
 if (omp_get_thread_num() == omp_get_team_size()-1){
   #pragma omp parallel num_threads(16)
   ... // GPU state as shown in Figure 3.
 }
}
```

Fig. 4: Example input with nested parallel regions that will trigger the creation of thread ICV states at runtime. The state of the runtime when the innermost parallel region is reached is sketched in Figure 3.

```
template<typename T> void
noChunkImpl(void (*LoopBody)(T, void *), void *Args,
     T NumBlocks, T BId, T NumThreads, T TId, T
    NumIters, bool OverSubscriptionAssumption) {
  T TotalThreads = NumBlocks * NumThreads;
  // Start index in the normalized space.
  T IV = BId * NumThreads + TId;
  // Cover the entire iteration space.
  if (IV < NumIters) {
    do {
      // Execute the loop body.
      LoopBody(IV, Args);
      // Every thread executed one iteration now.
      IV += TotalThreads;
      // User assumptions to avoid the loop.
      if (OverSubscriptionAssumption) break;
    } while (IV < NumIters);
  }
}
```

Fig. 5: Pseudocode for the core of the new runtime implementation of worksharing in the absence of user-defined chunks.

execute at most one iteration/block, then the loops within the runtime function can be removed. This compile-time assumption can be enabled via two new command-line flags `-fopenmp-assume-teams-oversubscription` and `-fopenmp-assume-threads-oversubscription` which emit constant globals that the runtime will "read" at compile time (via constant propagation) and subsequently break the loops after asserting that the condition actually holds at runtime.

G. Debugging, Assertions, and Assumptions

We provide a single OpenMP GPU runtime that exposes debugging features with zero overhead for release builds. Debugging capabilities can be enabled at compile time via a command line flag and later activated at runtime using an environment variable. If debugging is not enabled at compile time, the debugging code paths will be trivially dead and removed statically. This is similar to the over-subscription assumptions discussed in Section III-F.

The runtime supports fine-grained debugging through the use of a bit-field that specifies which debugging features are to be enabled. Currently, two debugging modes are supported: runtime assertions and runtime call function tracing. Standard C/C++ assertions, i.a., `assert(cond && "message")` are supported in target regions as we provide an `__assert_fail` implementation which is usually found in libc on the host. The runtime internally uses `__assert_assume` that provides the same functionality as a regular assertion if those are enabled. If not, thus in release mode, the condition will automatically become an assumption via `__builtin_assume`.

Assumptions are used extensively by the runtime to inform the compiler of certain invariants, such as the expected ICV states (ref. Section IV-B3). Through the described mechanism they are implicitly checked in debug runs to verify correctness. Additionally, through the `omp assumes` directive, we attach high-level properties to code regions. This is particularly useful to inform our optimization pass (ref. Section IV) about the behavior of inline assembly. An example is shown in Figure 6 where we use the `aligned_barrier` and `no_call_asm` extensions to mark the inline assembly as an aligned barrier that will not transfer execution to another function.

```
#pragma omp begin assumes ext_aligned_barrier     \
                          ext_no_call_asm
void syncThreadsAligned() {
  constexpr int BarrierNo = 8;
  asm volatile("barrier.sync.aligned␣%0;" ::       \
               "r"(BarrierNo) : "memory");
}
#pragma omp end assumes
```

Fig. 6: An aligned barrier in our runtime that uses LLVM extensions for OpenMP 5.1 assumptions. These assumptions tell the compiler that the assembly is part of an aligned barrier and it will not transfer execution to another function.

IV. The LLVM OpenMP Optimization Pass

The OpenMP-Optimization pass (aka. openmp-opt), is enabled by default since LLVM 12 and runs multiple times at optimization level O1 or higher. It uses OpenMP-specific domain knowledge to optimize OpenMP code with inter-procedural analysis and code transformation. This section covers the various optimizations both existing (Section IV-A) and added for this work (Section IV-B - Section IV-D).

A. Globalization Elimination and SPMDzation

The OpenMP standard was designed with CPU parallelism in mind. However, architecture differences between the CPU and GPU make common patterns, such as sharing of local (aka. "stack") variables or forking and joining threads, costly on the latter. The OpenMP optimization pass in LLVM is already able to transform idiomatic OpenMP code into a form closer to that of a traditional GPU kernel. We briefly describe three key optimizations necessary to put our work into context.

1) Internalization: The OpenMP optimization pass performs aggressive internalization. In essence, it duplicates functions with external linkage[1] to create an internal only copy, used when invoked from a kernel within the translation unit, to aid in inter-procedural analysis.

2) Globalization Elimination: The OpenMP standard implicitly allows threads to share their local variables with other threads in a team. This is straightforward on a CPU where the stack is accessible by all threads, but on a GPU this is not true. In order to maintain OpenMP semantics the front-end (here LLVM/Clang) allocates potentially shared variables in "shareable" memory, e.g., shared or global memory. The existing LLVM OpenMP optimization pass can optimize out these runtime allocations in parts by determining if they are executed only by the main thread of a team in isolation or are private to a single thread. We later utilize this analysis as described in Section IV-C.

3) SPMDzation: An Attributor-based optimization transforms eligible generic mode kernels to SPMD mode. First, all sequentially executed code is analyzed inter-procedurally for any side effects not in compliance with the local guarding scheme. Instructions executed by the main thread with no side-effects are simply recomputed while others are guarded for single threaded execution. These optimizations allow us to first transfer the majority of OpenMP regions to closely match the structure of a standard GPU kernel.

[1]Not all linkage types allow this. An optimization remark is emitted if internalization fails.

B. Inter-Procedural Conditional Value Propagation

The most important optimization we implemented is fundamentally an inter-procedural conditional value propagation pass that handles certain kinds of memory accesses. In contrast to the existing implementation in LLVM, namely the inter-procedural sparse conditional constant propagation pass (IP-SCCP), our Attributor-based implementation can deal with various complexities that arise from the GPU design and real world use cases. Our pass handles common constant folding situations like operations with (assumed) constant operands or phi nodes with (assumed) dead predecessors. Furthermore, it follows values communicated via memory if the underlying object is properly analyzable. We generally require it to be an internal global variable, a stack allocation, or the result of a known memory allocation function, e.g., `malloc`. In the following subsection we will detail other functionalities that are not available anywhere else in LLVM but required to eliminate the runtime state entirely.

1) Field-Sensitive Access Analysis: To manage the ICV state, which is the most complex part of the GPU resident state of the OpenMP runtime, we employ structures that are allocated in shared memory. Since each field is initialized and read separately throughout the application, we perform an analysis that categorizes accesses into bins based on their relative (constant) offset in bytes and access size. Unknown offsets or users are binned separately. This effectively gives us field-sensitivity but also captures the fact that LLVM-IR does not have any semantically relevant restrictions with regards to structure fields. As with C/C++ code, structure fields are mostly used for offset calculation but do not provide semantic guarantees about access sizes or bounds.

Users of the analysis can query potentially interfering accesses to determine if the value of a load can be predicted or a store is effectively dead. The query results are implicitly filtered to hide accesses that are known to not affect the given instruction, e.g., due to their offset and size or due to reachability and dominance (ref. Section IV-B2). This is especially important when the underlying object has unanalyzable uses but we can show that the user provided access cannot be affected by those.

A potentially interfering access is presented as "exact" if it matches the offset and size of the given load or store. For most purposes we are only interested in exact matches and most users will conservatively give up for non-exact ones. There is an exception however if a range of memory is only initialized with zero-bytes, e.g., `NULL` values. Even if we cannot predict the offset of each access precisely we still can deduce that a load from anywhere in the entire underlying memory region is effectively resulting in a zero value. This is important to replace loads of the thread states array (ref. Figure 3) at a statically unknown index, namely the thread id, with a zero value if all non-initialization writes to the array are (assumed

to be) dead. Thus, if we can statically show that no individual thread ICV state is ever created, we can eliminate loads.

2) Lifetime-Aware Reachability and Dominance Analysis: It is common practice to use reachability and dominance to reason about the content of memory. Assuming a single threaded environment or no synchronization events between accesses we can, for example, perform the following deductions. If a write cannot reach a load it will not affect the loaded value. If two writes dominate each other and also a load we know that the writes are executed in order and can exclude the first from the reasoning about potentially loaded values. If a write dominates a load and each path from a second write to the load needs to contain the first write, the second write will not affect the loaded value.

To justify the absence of synchronization events between writes we use information about aligned barriers and single threaded execution as described in Section IV-C. Together with existing LLVM capabilities this suffices to perform intra-procedural reachability and dominance deductions. Our optimizations provide these functionalities inter-procedurally. To identify a reference point in the generally multi-entry call graph of an application we look at the lifetime of the underlying memory. That means we will first determine if there is a common ancestor function for the accesses in question such that the memory would not be available in the caller of that function. These ancestor points can be identified for stack allocations and shared memory at the moment. The former is not available in the caller of the function that contains the stack allocation, the latter is not available in the (unknown) caller of a GPU kernel. If no common ancestor is found the accesses are unrelated, otherwise we perform reachability and dominance queries in the ancestor using the call sites that lead to the accesses in questions. Unknown callers and callees, recursion, loops, and irreducible control flow are among the things the analysis has to account for.

3) Assumed Memory Content: In GPU programming, it is common to have conditional writes that are performed only by a single thread in the team as a form of broadcast. Most parts of the OpenMP GPU runtime initialization and state updates work this way. Namely, the main thread writes state and then synchronizes with the team such that all threads can read it.

Figure 7 shows different ways in which conditional writes can be implemented. In our OpenMP runtime we choose to use conditional pointers, shown in Figure 7b, to avoid the conditional execution illustrated in Figure 7a. However, both schemes share the problem that domain knowledge is required to justify the write actually happened. If there is no thread zero there will also not be a write of `value` to `state`. In the conditional execution case the written location is known but the write does not dominate the aligned barrier that effectively broadcasts the result to all threads. In the conditional pointer case the write dominates the aligned barrier but the written location is not known.

In practice, conditional writes prevent us from using (inter-procedural) dominance relationship (ref. Section IV-B2) to track the content of shared state. A simplified example is shown in Figure 8a. These situations most often arise whenever we have state updates in a structured code region, e.g., a parallel. To keep our analysis generic we decided to place assumptions into the runtime code as shown in Figure 8b. However, domain knowledge could be used in the future to guarantee at least, or exactly, one thread will have id zero. Either allows the analysis to determine the value of the state at the respective location and to employ dominance reasoning to filter out non-interfering accesses.

4) Invariant Value Propagation: In addition to constants we propagate values that are known to be invariant or re-computable from invariant values. The most notable example for such invariant values are the intrinsics that read the grid dimensions from the GPU registers. We further can propagate instructions and function arguments through memory if we can show the same dynamic instance of those values is read by a load. Sufficient conditions include the absence of recursion in the function that allocates the underlying memory or global memory that can only exists once.

C. Exclusive and Aligned Execution of Code

Reasoning about memory is difficult in sequential contexts already but concurrent execution of memory accesses makes it arguably more challenging. Due to synchronization between threads it is possible that memory content changes between a write and a read in the same straight line sequence of code. An example is sketched in Figure 9a. This is even true if the accesses themselves are not atomic or volatile in nature.

To be able to reason about memory accesses in a concurrent program we employ two related analyses. First, we determine if an access is only executed by the main thread of a team. This special reasoning was already present in the OpenMP optimization pass in LLVM to improve the handling of globalized memory (ref. Section IV-A). Second, we inter-procedurally track the "aligned context" in which threads have encountered, or will encounter, an *aligned* barrier without other potentially synchronizing instruction on any path. An aligned barrier indicates that all threads in the team will arrive at the same barrier instruction. If there is no potentially synchronizing instruction between the access and the previous

```
if (get_thread_id() == 0)
  *state = value;
aligned_barrier();
```

(a) Conditional write using conditional execution of the write.

```
template<typename T>
void write(T *P, T &V, bool C) {
  // get uninitialized shared memory (ref. [2])
  static T SHARED(Dummy);
  *(C ? P : &Dummy) = V;
}
write(state, value, get_thread_id() == 0);
aligned_barrier();
```

(b) Conditional write using a dummy location and conditional pointer.

Fig. 7: Two possible implementations of conditional writes. For neither we can know that `state` holds `value` afterwards without domain knowledge that guarantees at least one thread will have thread id zero.

```
write(&TeamICVState.levels_var, 0, tid == 0);
...
// begin of a #pragma omp parallel
write(&TeamICVState.levels_var, 1, tid == 0);
...
use_in_parallel(TeamICVState.levels_var);
...
// end of a #pragma omp parallel
write(&TeamICVState.levels_var, 0, tid == 0);
...
use_after_parallel(TeamICVState.levels_var);
```

(a) Chain of effectively dominating accesses to the *levels-var* ICV value that are not statically known to dominate each other due to the conditional nature of the writes (ref. Figure 7).

```
write(&TeamICVState.levels_var, 0, tid == 0);
aligned_barrier();
__builtin_assume(TeamICVState.levels_var == 0);
```

(b) Assumption employed after broadcast barriers that can be effectively seen as unconditional writes of the state.

Fig. 8: Simplified chain of effectively dominating writes (top) which cannot be used to predict the memory state without domain knowledge or additional assumptions (bottom).

```
*state = value;
synchronize();
synchronize();
use(*state);
```

(a) Straight line code with synchronization effects that prevent reasoning about memory content.

```
*state = value;
aligned_barrier();
aligned_barrier();
use(*state);
```

(b) Straight line code with "aligned" synchronization effects that do not prevent reasoning about memory content.

Fig. 9: Examples to illustrate how aligned synchronization effects can be utilized to reason about execution order while non-aligned synchronization allows complex interleaving of threads which can invalidate common assumptions.

as well as succeeding aligned barrier we denote it to be in an "aligned epoch". If it is only preceded or succeeded by an aligned barrier without interfering synchronizing event it is an "aligned access".

Two aligned non-atomic stores or stores that are executed by the same (=main) thread can utilize dominance reasoning (ref. Section IV-B2). The dominance relationship, together with the aligned or same thread restriction (and the assumption accesses cannot race) guarantees the accesses are executed in "control flow order". The idea is that we can choose to stall threads that reach the dominated store just before it is executed and until all other threads have stopped making progress. This ensures that the dominated store is observed last before a load dominated by both of them. Note that recursion and loops still need to be taken into account.

If a load and a store are accessed in an "aligned epoch" we can further utilize reachability between them to exclude write effects that may not impact the given read. This is sound because the two accesses are either between the same two aligned barriers without other synchronization effects, in which case the store has to follow the load and there is no path back for any thread. Or, alternatively, the two accesses are separated by at least the aligned barrier after the store which has to be reached by all threads. Since that barrier cannot reach the load, as the store could not either, no thread can reach the load and the store has no effect on it.

D. Aligned Barrier Elimination

The GPU runtime and compiler generated code defensively use barriers to ensure threads have a consistent view of the memory. When transformations, such as de-globalization and removal of runtime state, eliminate all non-thread-local side-effects between barriers it may render them redundant. However, only aligned barriers, thus those executed by all threads in the team, are trivially removable in this case. Non-aligned barriers might synchronize with threads that diverged earlier, e.g., as part of the state machine implementation for generic mode execution. To avoid having to reason about thread divergence we utilize specific barriers, annotated as aligned for the compiler, whenever possible.

Our barrier elimination pass detects consecutive aligned barriers in the same basic block that do not have non-thread-local side-effects in between them. During this identification process we also consider the kernel entry and exit as implicit aligned barriers. Whenever two such aligned barriers are found we can eliminate one of them. This is especially useful if one of the barriers is implicit, hence kernel entry or exit.

V. Evaluation

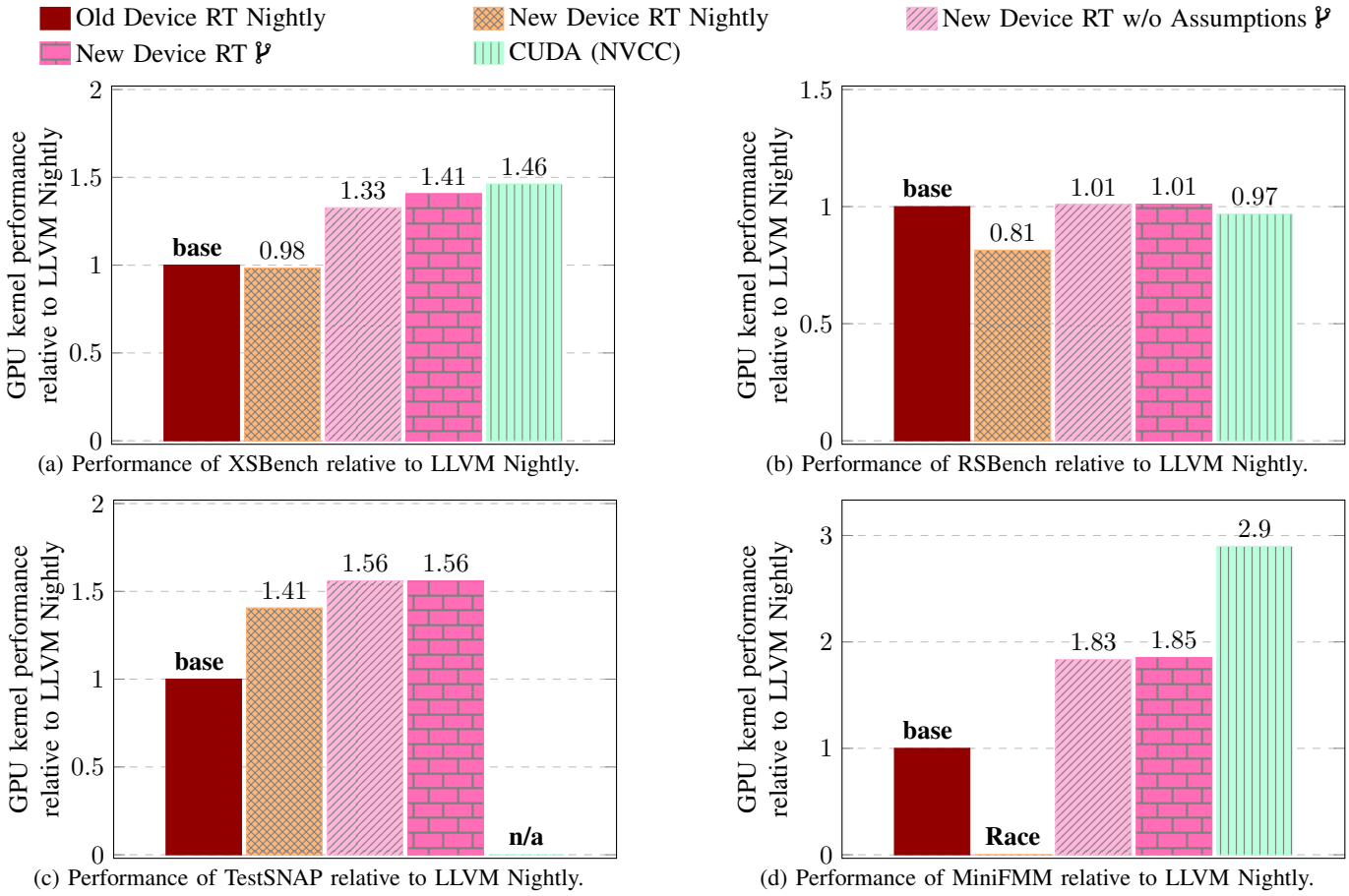
For our experiments we used an NVIDIA A100 GPU in a Gigabyte G242-Z11 Server containing an AMD EPYC 7532 (2.4Ghz) CPU, bundled with 256 GB DDR4 RAM. We used CUDA 11.4.0 for all experiments and collected kernel times with Nsight Compute CLI. Benchmarks were compiled with NVCC 11.4 (GCC 7.5.0), LLVM/Clang (Nightly) based on **git**: `81e9c90`, and a development version (ڤ) based on **git**: `a4ae55c` available at https://github.com/jdoerfert/llvm-project/tree/IPDPS22.

A. Benchmarks

For our performance study we looked at five scientific proxy applications and compared the performance of their main GPU kernel in different configurations. Our results are presented relative to the performance of the OpenMP version compiled with LLVM/Clang Nightly using the default GPU runtime.

XSBench and RSBench are two proxy applications for the Open Monte Carlo (OpenMC) project. OpenMC [6] simulates the transport of neutrons and photons using the Monte Carlo methodology. Both proxies compute the continuous energy macroscopic neutron cross-section lookup when studying neutron transport and both are available in multiple programming languages and frameworks. While XSBench [7] extracts one of the main kernels in OpenMC, which is in our setup memory bound, RSBench [8] provides a compute bound alternative implementation. Note that for OpenMP we moved the reduction out of the timed kernel to match the CUDA version.

GridMini is a proxy application for Lattice Quantum Chromodynamics (QCD). Lattice QCD simulates the strong interactions of quarks and gluons on a four-dimensional discrete



(a) Performance of XSBench relative to LLVM Nightly.

(b) Performance of RSBench relative to LLVM Nightly.

(c) Performance of TestSNAP relative to LLVM Nightly.

(d) Performance of MiniFMM relative to LLVM Nightly.

Fig. 10: Performance evaluation for the four proxy applications (ref. Section V) on an NVIDIA A100 GPU normalized to performance of LLVM Nightly. Values over 1 indicate improvements and values below denote slowdowns.

space-time grid, and provides crucial input to theoretical nuclear and particle physics. GridMini is a substantially reduced version of Grid [9], a new C++ lattice QCD library developed for highly parallel computer architectures. We modified the application to load the loop upper bound by value to more closely match the CUDA version.

TestSNAP is a proxy for the SNAP force calculation in the LAMMPS molecular dynamics package [10, 11], which contains synthetic inputs (neighbor atom positions) and reference outputs (forces on atoms) for several different SNAP models. It performs the force calculation repeatedly, checking the results against the reference data. At the end it reports the grind time (msec/atomstep) and RMS force error (eV/A).

MiniFMM is a proxy application developed by the University of Bristol for Fast Multipole Method (FMM) [12]. It solves the Laplace equation in a three-dimensional polar coordinate plane by applying the FMM, which uses a dualtree traversal method and is a test case for dynamic task parallelism [13].

B. Performance and Resource Usage

For benchmarks such as RSBench (Figure 10b) that already exhibited CUDA-like performance, there is essentially no difference between the performances of the new runtime implementations and CUDA. The new runtime, as available in the nightly build at the moment of writing this paper, created a performance regression. However, the most recent development branch solves this issue. For the XSBench (Figure 10a) and MiniFMM (Figure 10d), we see considerable improvement in performance in the latest development versions of the new runtime, getting close to the CUDA implementation. When using the new device runtime with assumptions enabled, the XSBench application closes the gap to the cuda performance by 5% difference. Although, there is still a difference between CUDA and the new runtime for MiniFMM, this difference is considerably reduced, with 1.85x improvements over the old runtime, reducing it to approximately a factor of 0.5x. Improvements in performance are directly proportional to the reduction in kernel time, as seen in Figure Figure 11. For TestSNAP, the supplied CUDA implementation used Kokkos for which a one-to-one kernel mapping between OpenMP kernels and Kokkos kernels could not be determined. For GridMini, we are able to match the CUDA performance in terms of GFlops, representing a substantial performance improvement over LLVM nightly in relation to the new device runtime as well as the old RT as shown in Figure 12. Note that MiniFMM when compiled with LLVM nightly with the new device runtime had a race condition which prevented

Build	Kernel Time	# Regs	SMem
MiniFMM	`./fmm.{cuda,omptarget}`		
Old RT (Nightly)	1.237 ms	70	8,288B
New RT (Nightly)	n/a	n/a	n/a
New RT - w/o Assumptions	0.674 ms	52	8,300B
New RT	0.667 ms	43	8,300B
CUDA (NVCC)	0.426 ms	64	4,096B
RSBench	`./rsbench -s large -m event`		
Old RT (Nightly)	0.992 s	162	2,336B
New RT (Nightly)	1.220 s	172	11,304B
New RT - w/o Assumptions	0.984 s	156	0B
New RT	0.982 s	153	0B
CUDA (NVCC)	1.025 s	102	0B
XSBench	`./xsbench -s large -m event`		
Old RT (Nightly)	0.202 s	86	2,336B
New RT (Nightly)	0.205 s	94	11,304B
New RT - w/o Assumptions	0.152 s	80	0B
New RT	0.143 s	66	0B
CUDA (NVCC)	0.138 s	50	6B
TestSNAP	`./test_snap.exe`		
Old RT (Nightly)	0.291 s	109	2,336B
New RT (Nightly)	0.207 s	117	11,304B
New RT - w/o Assumptions	0.187 s	92	3,076B
New RT	0.187 s	80	3,076B

Fig. 11: GPU kernel execution times (highest), shared memory and register usage, and execution command for each app.

profiling. Absolute times for the longest running kernel in each benchmark as well as the shared memory usage and register counts are shown in Figure 11. Most performance benefits can be traced to reducing and/or eliminating the shared memory and register usage compared to LLVM Nightly.

Loop over-subscription assumptions have two effects in the generated code. First, they reduces the live register count as there is no loop carried state. Second, they remove control flow edges which can enable generic compiler optimizations. For XSBench, we observe a considerable reduction in register usage (14) which comes with significant lower kernel execution time (-5.6%). Other benchmarks also require less registers and control flow conditions, due to the eliminated loop carried state, but they are missing secondary effects. Consequently, the kernel execution time is not affected much.

C. Optimization Effects

In order to differentiate the effect of each optimization introduced in Section IV we evaluated GridMini, XSBench, and MiniFMM with one optimization disabled at a time. Section IV-B1 introduces the field-sensitive access optimization. It is then extended in Section IV-B2, IV-B3, and IV-B4. Therefore, removing the first part implies removing all optimizations in Section IV-B, while removing the other parts still allows some field-sensitive access analysis to occur.

Improvements in XSBench and MiniFMM are directly traceable to the base field-sensitive access optimization in Section IV-B1. In the case of MiniFMM no other optimization has any effects on performance. XSBench, on the other hand, improves performance by 20% due to field-sensitive access optimizations and an additional 10% from assumed memory content optimization introduced in Section IV-B3.

Figure 13 shows the effect of the different optimizations on GridMini. Field-sensitive access analysis optimization, and its deviates, provides most of the performance boost, while exclusive and align execution of code, and aligned barrier elimination, Section IV-C and Section IV-D respectively, still play an important role in achieving CUDA-like performance.

VI. Related Work

In LLVM/Clang, OpenMP offloading support for GPUs was first presented by [3, 4], proposing a control loop to coordinate GPU threads for executing sequential and parallel regions within an offloaded region, by implementing a state machine to track execution semantics. Later [14] introduced the first front-end based optimizations for Nvidia GPUs in LLVM/Clang, related to choosing the number of teams and threads for parallel loops to avoid idle threads and reduce register usage. Recently [15] introduced a lowering of OpenMP 4.5 in the IBM XL C/C++ compiler that executes without the control loop state machine in a mode where all threads execute in parallel, deemed SPMD mode of execution, when the target offloaded region encloses a single parallel construct. Since IBM XL C/C++ compiler is derived from Clang/LLVM, those changes have been upstreamed to LLVM/Clang too. The comparative analysis of multiple OpenMP compilers by [16, 17] demonstrated severe performance issues in the LLVM/OpenMP implementation among other compilers.

Regarding compiler-based optimizations on OpenMP, [18] presented the TRegion interface which delayed the discovery of SPMD regions into LLVM, by contrast to the Clang-based approach, which enabled more kernels to execute in SPMD mode. For host OpenMP, [19] demonstrated that LLVM/Clang's outlining approach hinders the application of existing compiler optimizations. By analyzing the semantics of OpenMP runtime functions, the authors re-enabled such optimizations in the presence of OpenMP parallelism. In addition to some ideas shared with [15], such as the device function de-virtualization, they also mention SPMD code generation for the `distribute parallel` loop constructs in a target team region. [20, 21] proposed extensions to the LLVM IR, in the form of intrinsics, to represent parallelism present in OpenMP directives. However, those extensions are hard to integrate in the existing LLVM/Clang infrastructure and provide little benefit in semantic analysis in comparison to other approaches [18, 19] that directly analyze OpenMP runtime functions. Further, tools that translate between OpenACC and OpenMP [22, 23] include optimizations designed to resolve execution execution model mapping mismatches, both on CPUs and GPUs, such as using nested parallelism and the SIMD clause.

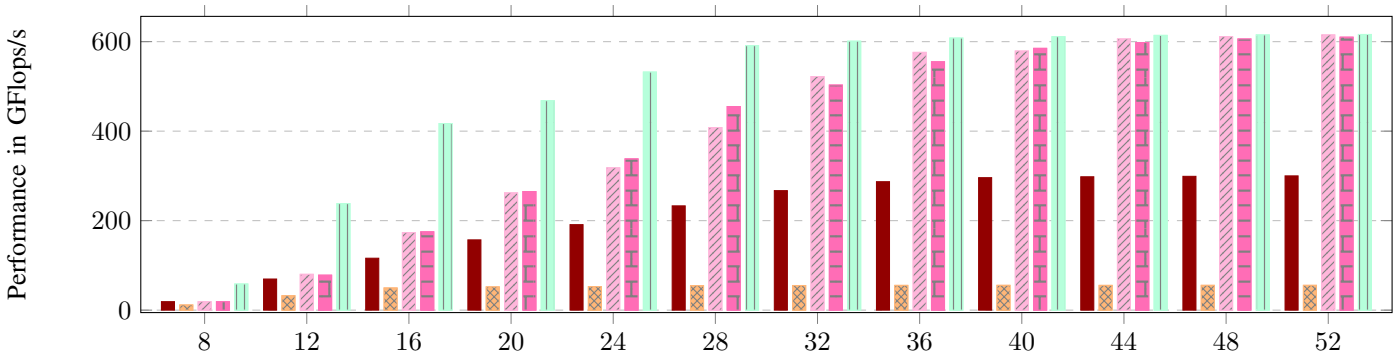


Fig. 12: Performance of GridMini in GFlops/s as reported by the application per grid size L. Bars as shown in Figure 10.

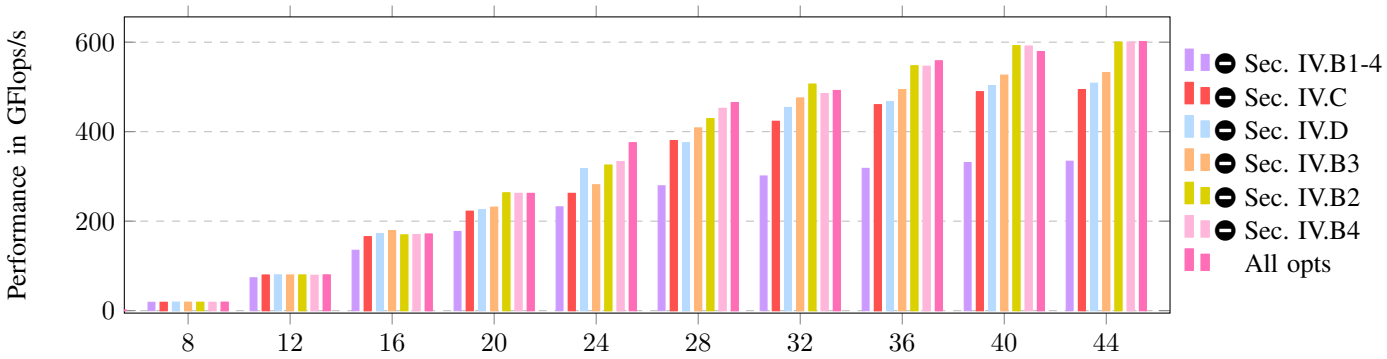


Fig. 13: Effect of optimizations on performance of GridMini in GFlops/s as reported by the application per grid size L.

VII. Remaining Challenges: OpenMP vs CUDA

Some challenges remain that prevent OpenMP from achieving complete performance parity with CUDA. Currently, aggregate data types, like the configuration struct used in XSBench, can only be passed to the device kernel by-reference. This requires an extra load of the base pointer to access the elements compared to passing the struct to the kernel by-value. Additionally, certain constructs, e.g., `parallel` require barriers at the start and end to synchronize the thread state. These cannot always be removed especially if a work-shared or distributed loop uses bounds loaded from memory. While such bounds might be known to be invariant, their side-effect will currently cause barrier elimination (ref. Section IV-D) to consider the barrier as essential when it is in fact not. Note that we addressed the loop bound issue manually for GridMini prior to our evaluation by passing in the loop bound into the target region; this matches the CUDA version. For XSBench the same issue occurs and if addressed the OpenMP register count (ref. Figure 11) will match the CUDA version as well.

Additionally, missed optimizations can cause leftover abstractions in the runtime. The optimizations mentioned in IV-A are responsible for transforming an OpenMP region into a kernel-like form if it is not in one already. If either of these optimizations fail, the OpenMP region will necessarily require additional overhead for the state-machine or the data-sharing stack. For this reason, we provide compiler diagnostics for missed optimizations using `-Rpass-missed=openmp-opt` and `-Rpass-analysis=openmp-opt`.

VIII. Conclusion

In this paper, we proposed a co-design methodology for optimizing parallel code targeting accelerators, GPUs in particular, that re-designs the runtime API and its internals for a near-zero overhead execution through compiler optimization. Specifically, our approach exposes runtime semantics and state to the compiler to perform aggressive optimization, removing or folding runtime state, and simplifying the control flow of parallel loops, in parts with the help of user provided assumptions. The evaluation results show that our prototype implementation in LLVM/OpenMP GPU offloading can significantly improve the performance over previous LLVM versions, and oftentimes match CUDA performance without sacrificing the versatility and portability of OpenMP.

IX. Acknowledgements

This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725. We also gratefully acknowledge the computing resources provided and operated by the Joint Laboratory for System Evaluation (JLSE) at Argonne National Laboratory. Part of this research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering,

and early testbed platforms, in support of the nation's exascale computing imperative. Part of this research was supported by the Lawrence Livermore National Security, LLC ("LLNS") via MPO No. B642066. This work was partially supported by LLNL under Contract DE-AC52-07NA27344 (LLNL-CONF-826728) through the LLNL-LDRD Program Project No. 21-ERD-018.

REFERENCES

[1] A. C. Jacob, A. E. Eichenberger, H. Sung, S. F. Antao, G.-T. Bercea, C. Bertolli, A. Bataev, T. Jin, T. Chen, Z. Sura, G. Rokos, and K. O'Brien, "Efficient Fork-Join on GPUs Through Warp Specialization," in *HiPC*, 2017.

[2] S. Tian, J. Chesterfield, J. Doerfert, and B. M. Chapman, "Experience Report: Writing a Portable GPU Runtime with OpenMP 5.1," in *IWOMP*, 2021.

[3] C. Bertolli, S. Antão, 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 *LLVM*, 2014.

[4] C. Bertolli, S. F. Antao, G.-T. Bercea, A. C. Jacob, A. E. Eichenberger, T. Chen, Z. Sura, H. Sung, G. Rokos, D. Appelhans, and K. O'Brien, "Integrating GPU Support for OpenMP Offloading Directives into Clang," in *LLVM-HPC*, 2015.

[5] A. Patel, S. Tian, J. Doerfert, and B. M. Chapman, "A Virtual GPU as Developer-Friendly OpenMP Offload Target," in *ICPP*, 2021.

[6] P. K. Romano, N. E. Horelik, B. R. Herman, A. G. Nelson, B. Forget, and K. Smith, "OpenMC: A State-of-the-art Monte Carlo Code for Research and Development," *Annals of Nuclear Energy*, 2015.

[7] J. R. Tramm, A. R. Siegel, T. Islam, and M. Schulz, "XSBench - The Development and Verification of a Performance Abstraction for Monte Carlo Reactor Analysis," in *PHYSOR*, 2014.

[8] J. R. Tramm, A. R. Siegel, B. Forget, and C. Josey, "Performance Analysis of a Reduced Data Movement Algorithm for Neutron Cross Section Data in Monte Carlo Simulations," in *EASC*, 2014.

[9] P. A. Boyle, G. Cossu, A. Yamaguchi, and A. Portelli, "Grid: A Next Generation Data Parallel C++ QCD Library," in *LATTICE*, 2016.

[10] A. P. Thompson, L. P. Swiler, C. R. Trott, S. M. Foiles, and G. J. Tucker, "Spectral Neighbor Analysis Method for Automated Generation of Quantum-Accurate Interatomic Potentials," *J. Comput. Phys.*, 2015.

[11] T. Deakin, S. McIntosh-Smith, J. Price, A. Poenaru, P. Atkinson, C. Popa, and J. Salmon, "Performance Portability across Diverse Computer Architectures," in *P3HPC@SC*, 2019.

[12] P. Atkinson and S. McIntosh-Smith, "On the Performance of Parallel Tasking Runtimes for an Irregular Fast Multipole Method Application," in *IWOMP*, 2017.

[13] T. Odajima, Y. Kodama, M. Tsuji, M. Matsuda, Y. Maruyama, and M. Sato, "Preliminary Performance Evaluation of the Fujitsu A64FX Using HPC Applications," in *CLUSTER*, 2020.

[14] S. F. Antao, A. Bataev, A. C. Jacob, G.-T. Bercea, A. E. Eichenberger, G. Rokos, M. Martineau, T. Jin, G. Ozen, Z. Sura, T. Chen, H. Sung, C. Bertolli, and K. O'Brien, "Offloading Support for OpenMP in Clang and LLVM," in *LLVM-HPC*, 2016.

[15] E. Tiotto, B. Mahjour, W. Tsang, X. Xue, T. Islam, and W. Chen, "OpenMP 4.5 Compiler Optimization for GPU Offloading," *IBM J. Res. Dev.*, 2020.

[16] J. Monsalve Diaz, K. Friedline, S. Pophale, O. Hernandez, D. E. Bernholdt, and S. Chandrasekaran, "Analysis of OpenMP 4.5 Offloading in Implementations: Correctness and Overhead," *Parallel Computing*, 2019.

[17] J. H. Davis, C. Daley, S. Pophale, T. Huber, S. Chandrasekaran, and N. J. Wright, "Performance Assessment of OpenMP Compilers Targeting NVIDIA V100 GPUs," in *WACCPD*, 2021.

[18] J. Doerfert, J. M. M. Diaz, and H. Finkel, "The TRegion Interface and Compiler Optimizations for OpenMP Target Regions," in *IWOMP*, 2019.

[19] J. Doerfert and H. Finkel, "Compiler optimizations for OpenMP," in *IWOMP*, 2018.

[20] X. Tian, H. Saito, E. Su, A. Gaba, M. Masten, E. Garcia, and A. Zaks, "LLVM Framework and IR Extensions for Parallelization, SIMD Vectorization and Offloading," in *LLVM-HPC*, 2016.

[21] X. Tian, H. Saito, E. Su, J. Lin, S. Guggilla, D. Caballero, M. Masten, A. Savonichev, M. Rice, E. Demikhovsky, A. Zaks, G. Rapaport, A. Gaba, V. Porpodas, and E. Garcia, "LLVM Compiler Implementation for Explicit Parallelization and SIMD Vectorization," in *LLVM-HPC*, 2017.

[22] J. Lambert, S. Lee, J. S. Vetter, and A. D. Malony, "CCAMP: An Integrated Translation and Optimization Framework for OpenACC and OpenMP," in *SC*, 2020.

[23] J. E. Denny, S. Lee, and J. S. Vetter, "CLACC: Translating OpenACC to OpenMP in Clang," in *LLVM-HPC*, 2018.