OpenMP Kernel Language Extensions for Performance Portable GPU Codes

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

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

Tom Scogland  
scogland1@llnl.gov  
Lawrence Livermore National Laboratory  
Livermore, CA, USA

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

ABSTRACT
In contemporary high-performance computing architectures, the integration of GPU accelerators has become increasingly prevalent. To harness the full potential of these accelerators, developers often resort to vendor-specific kernel languages, such as CUDA. While this approach ensures optimal efficiency, it inherently compromises portability and engenders vendor dependency. Existing portable programming models, such as OpenMP, while promising, demand extensive code rewriting due to their fundamental difference from kernel languages.

In this work, we introduce extensions to LLVM OpenMP, transforming it into a versatile and performance portable kernel language for GPU programming. These extensions allow for the seamless porting of programs from kernel languages to high-performance OpenMP GPU programs with minimal modifications. To evaluate our extension, we implemented a proof-of-concept prototype that contains a subset of extensions we proposed. We ported six established CUDA proxy and benchmark applications and evaluated their performance on both AMD and NVIDIA platforms. By comparing with native versions (HIP and CUDA), our results show that OpenMP, augmented with our extensions, can not only match but also in some cases exceed the performance of kernel languages, thereby offering performance portability with minimal effort from application developers.

CCS CONCEPTS
• Software and its engineering → Compilers: Parallel programming languages.

KEYWORDS
OpenMP, GPU, CUDA, HIP

ACM Reference Format:

1 INTRODUCTION
In the evolving domain of high-performance computing, the integration of GPUs for general-purpose computing has emerged as a pivotal trend. These GPUs are often programmed using vendor-specific “kernel languages” such as CUDA and HIP. While these languages offer tailored environments for GPU-centric tasks, ensuring optimal performance, they inherently lack cross-vendor portability. As the GPU ecosystem becomes increasingly diverse, there arises a compelling need for programming frameworks that meld the robust performance of kernel languages with the adaptability of cross-vendor compatibility.

OpenMP [17] has emerged as a prominent, performance-portable heterogeneous programming model. It provides developers with a suite of intuitive directives and interfaces, facilitating code execution on both host CPUs and GPUs. This adaptability ensures that applications can seamlessly harness the computational capabilities of a wide range of hardware architectures. Yet, the intrinsic disparities between OpenMP and kernel languages often demand significant code adaptations. While OpenMP allows developers to write single instruction multiple threads (SIMT) style code, similar to kernel languages, to a degree, thereby potentially easing the transition, it falls short in certain areas. Notably, the absence of support for specific functionalities, such as granular synchronization levels, makes a complete transition challenging.

In this work, we introduce novel extensions to LLVM OpenMP, designed to enable users to craft their GPU code in SIMT style, while simultaneously leveraging the performance portability intrinsic to OpenMP. These extensions simplify the transition from kernel languages to OpenMP, often reducing the porting process to text replacement. Moreover, our extensions integrate with host OpenMP tasking support, adeptly managing dependencies and ensuring efficient execution across both host CPUs and GPUs. An added advantage is the ability to blend traditional and kernel-like OpenMP code on GPUs, providing developers with the flexibility to preserve existing codebases while selectively incorporating GPU-specific optimizations.
To evaluate the efficacy of our proposal, we implemented a proof-of-concept prototype that contains a subset of the proposed extensions. We ported six established proxy and benchmark applications from their CUDA versions and evaluated their performance on both AMD and NVIDIA platforms, where we compared the performance to native program versions (HIP and CUDA). The results demonstrate that OpenMP, augmented with our proposed extensions, can not only match the performance of kernel languages but also, in some cases, outperform them, offering performance portability with minimal effort from application developers.

The rest of this paper is structured as follows: Section 2 explains the core principles of GPU programming via kernel languages and OpenMP; Section 3 details our extensions and design decisions; Section 4 presents the evaluation results of our proof-of-concept implementation; Section 5 discusses related works; and Section 6 concludes the paper.

2 BACKGROUND

In this section, we delve into the core principles of GPU programming with kernel languages and OpenMP target offloading. While we use CUDA as our primary example, the concepts are broadly applicable to most kernel languages. Figure 1 shows a simple CUDA program, and its OpenMP equivalent is shown in Figure 2. We will highlight the difference between the two approaches and identify the gaps in OpenMP that have driven our proposed extensions.

2.1 Kernel

A kernel serves as the GPU program’s entry point, similar to the main function in host programs. This program is initiated once launched from the host. CUDA kernels are defined with the __global__ keyword, as seen by the kernel function in Figure 1. Conversely, OpenMP does not need an explicit kernel definition; the body of a target region is outlined as a kernel by the OpenMP compiler.

2.2 Global Symbol

A global symbol, either a function or a global variable, is accessible on GPUs only if it is a device global symbol. In CUDA, functions can only call device functions and access device global variables. These are identified by the __device__ keyword, as shown by the use function in Figure 1. In OpenMP, symbols used within a target region do not require explicit annotation if they are in the same translation unit (TU). However, for symbols that are not directly used within a target region of the current TU but visible to others, they must be defined within the declare target directive.

2.3 Kernel Launch

CUDA employs the chevron syntax <<<...>> for invoking kernels, specifying grid size, block size, dynamic shared memory, and stream. For OpenMP, the compiler automatically generates the host code for a target region to launch the corresponding kernel. In LLVM OpenMP, an OpenMP team is mapped to a thread block, with grid and block sizes specified via num_teams and thread_limit clauses, respectively. However, OpenMP currently lacks support for multidimensional grid and block. In addition, the target construct in OpenMP is synchronous by default. This means that, unlike CUDA’s kernel launch, there is no need for explicit synchronization post the launch.
target region. OpenMP ensures that the program progresses only after all operations associated with the target region are complete.

2.4 Dependence and Stream

A stream is essentially an ordered queue of operations, facilitating asynchronous tasks and establishing dependencies. Operations within a single stream are executed sequentially while operations across different streams may execute concurrently. In CUDA, a majority of operations, including both kernel launches and runtime APIs, utilize a stream object to designate the queue in which they are enqueued.

OpenMP offers asynchronous operations through the `nowait` clause applied to relevant directives [26]. Dependencies can be established using the `depend` clause, while synchronization is managed via the `taskwait` construct. However, the `depend` clause comes with inherent limitations, making its application less intuitive compared to the approach of streams.

2.5 Memory Hierarchy

GPUs possess a diverse memory hierarchy, each with distinct characteristics like latency, bandwidth, and access modes. CUDA supports four commonly used computational memory spaces\(^1\): private, shared, global, and constant, with respective keywords like `__private__` and `__shared__`. In OpenMP, the allocate directive, combined with the appropriate allocator, serves a similar purpose\(^2\).

2.6 Data Mapping

Managing data between host and device typically involves: 1) device memory allocation, 2) host-to-device data transfer, and 3) post-kernel execution, device-to-host data transfer. In CUDA, these steps are explicitly executed using runtime APIs, e.g., `cudaMalloc` and `cudaMemcpy`. OpenMP offers two data management strategies: directive-based and API-based. Directives like `target data` and `target update` automate these steps, while APIs such as `omp_target_alloc` and `omp_target_memcpy` offer explicit control.

2.7 Synchronization

CUDA offers a multi-level synchronization mechanism, encompassing warp, block, and kernel levels. Additionally, it furnishes primitives like shuffle for efficient thread communication. In contrast, while OpenMP provides the `barrier` directive for explicit synchronization and mandates implicit barriers at the end of specific regions, it currently falls short in providing the granularity of control that CUDA offers.

2.8 Workload Distribution

In CUDA, the onus is on the users to manually distribute workloads among threads, typically by determining memory access offsets using thread and block identifiers, as exemplified by the variable `id` in Figure 1. OpenMP, however, leans towards automation. Commonly, developers employ work-sharing constructs like `distribute` and `for directives to distribute workloads. Yet, there is flexibility; one can craft OpenMP code in SIMT style, as illustrated in Figure 3. This approach, however, comes with limitations:

- As highlighted in Section 2.3, OpenMP’s lack of support for multi-dimensional grid/block configurations necessitates the translation of workloads into a one-dimensional space.
- The aforementioned lack of granular control in Section 2.7 makes it challenging to implement certain functions.

```c
#pragma omp target teams num_teams(gsize) thread_limit(bsize)
#pragma omp parallel
{
    int shared[128];
    #pragma omp groupprivate(team: shared)
    int threadId = omp_get_thread_num();
    /* initialize shared */
}
#pragma omp barrier
int blockIdx = omp_get_team_num();
int blockDim = omp_get_team_size();
int id = blockIdx * blockDim + threadIdx;
if (id < n)
{
    b[id] = use(a[id], tmp[threadIdx]);
}
```

Figure 3: Write the target region of Figure 2 in SIMT style.

3 EXTENSION DESIGN

In this section we will describe our proposed extentions and the design decisions incorporated into LLVM OpenMP, enabling users to craft OpenMP programs in a SIMT style.

3.1 ompx_bare Clause

Figure 3 illustrates that, to invoke a SIMT style target region, users must use the `target teams construct followed by a parallel construct\(^3\). Despite of the limitation mentioned before, such a structure may appear convoluted to those unacquainted with OpenMP’s concepts.

To address this, we introduce the `ompx_bare` clause that can be used on the `target teams construct. This is not simply a "syntax sugar" for the previous nested constructs. When this clause is present, the target region operates in a "bare metal" mode, similar to the SIMT model of kernel languages. The front end will not generate code to initialize the OpenMP device runtime that ensures the compliance with the OpenMP execution model [5]. We retain the `teams clause to preserve the semantic consistency, especially when using `num_teams` and `thread_limit` to define grid/block sizes. The `ompx_bare clause also changes the scope of variables defined in the region such that they will not be globalized according to OpenMP semantics [9]. Now a bare metal OpenMP target region can be launched in the following approach:

\(^1\)Texture memory is not widely used in computation.

\(^2\)Technically this is a compiler work around. The OpenMP language committee is proposing a new approach to define variables in different memory space. For example, `#pragma omp groupprivate(team: var)` can be used to define a variable `var` in shared memory. We will use the new syntax in the rest of the paper.

\(^3\)While the `target teams parallel construct is currently disallowed, OpenMP 6.0 is set to remove this restriction.
While OpenMP’s comprehensive set of directives can automate numerous tasks—such as memory allocation and data movement—as discussed in Section 2.6, there remains a need for direct API interactions with the device. To address this, we expand the APIs by adapting the user-facing API implementations from Doerfert et al. [4]. These enhanced APIs can be integrated under the ompx umbrella, such as ompx_malloc for cudaMalloc.

3.2 Multi-Dimensional Grid and Block

The num_teams and thread_limit clauses are augmented to accept a list of integers to support multi-dimensional grid and block. For instance, a three-dimensional CUDA grid size represented as dim3 gridSize(128, 64, 32) can be equivalently expressed using num_teams(128, 64, 32). While we do not impose a dimensionality constraint at the OpenMP level, any dimensions exceeding a device’s capability will be disregarded.

3.3 Device API

Our extensions furnish APIs that enable users to interact with a device in the same way as a kernel language. To get a portable device API we follow the design of the OpenMP device runtime introduced by Tian et al. [25]. We provide two sets of APIs: C APIs prefixed with ompx_, and C++ APIs encapsulated within the ompx namespace. They can be extended to support Fortran. In the following we will briefly introduce two sets of widely used APIs.

3.3.1 Thread Indexing

Both CUDA and HIP provide four built-in thread indexing variables: threadIdx, blockIdx, blockDim, and gridDim. Each of these has three data members, \((x, y, z)\), corresponding to their respective dimensions. In our extension, this information is retrieved through OpenMP-like APIs. For instance, the C API ompx_thread_id_x() and its C++ counterpart ompx::thread_id(x) are equivalent to threadIdx.x.

3.3.2 Synchronization

In our design, we provide APIs to bridge the gap discussed in Section 2.7. Examples include ompx_sync Warp and ompx sync thread block. Furthermore, we also introduce APIs for warp-level primitives, such as ompx_shfl Sync.

3.4 Host API

While OpenMP’s comprehensive set of directives can automate numerous tasks—including memory allocation and data movement, as discussed in Section 2.6, there remains a need for direct API interactions with the device. To address this, we expand the APIs by adapting the user-facing API implementations from Doerfert et al. [4]. These enhanced APIs can be integrated under the ompx umbrella, such as ompx_malloc for cudaMalloc.

3.5 depend Clause

OpenMP 5.1 introduced the interop construct, facilitating interoperability with foreign runtime environments. This allows users to retrieve a stream from the OpenMP runtime and use it for operations in other environments.

As discussed in Section 2.4, the depend clause in OpenMP typically orchestrates dependencies for asynchronous operations. While it might seem logical to use streams directly into the depend clause to manage the execution of the associated construct, the current design of the depend clause does not allow it because only the location of the item, not its semantics, is taken when resolving dependencies. To bridge this gap, our extension proposes an enhancement to the depend clause. We introduce a new dependence type, interopobj, where the associated item is an interop object. The semantics of this interop object dictate the handling of the corresponding operations. For instance, the kernel corresponding to the target region in Figure 5 would be dispatched into the stream linked with obj.

```
#pragma omp target teams ompx_bare nowait depend(interopobj: obj)
#pragma omp interop init(targetsync: obj)
omp_interop_t obj = omp_interop_none;
#pragma omp interop init(teamsync: obj)
#pragma omp target teams ompx_bare nowait depend(interopobj: obj)
{
    ...}
```

3.6 Interoperability with Vendor Libraries

Vendor-specific libraries are prevalent and offer high efficiency, but they often adhere strictly to their proprietary programming models. For instance, cuBLAS [16] is exclusively tailored for CUDA. For OpenMP, crafting a performance-portable library with the same capabilities as vendor libraries from the ground up is not feasible. To address this, our extension introduces a lightweight wrapper layer. This layer boasts function signatures similar to those in vendor libraries, enabling users to effortlessly transition from vendor-specific implementations. Under the hood, this wrapper layer invokes the appropriate vendor library based on the offloading target determined at compile time.

4 Evaluation

The primary objective of our evaluation is to ascertain whether OpenMP, when augmented with the kernel language extensions proposed in this work, can rival the performance of native programming models on NVIDIA and AMD GPUs. To address this question, we implemented a proof-of-concept prototype based on LLVM 18 that contains extensions described in Sections 3.1 and 3.3.

4.1 Methodology

Our evaluation setup included both AMD and NVIDIA systems, with the hardware and software configurations detailed in Figure 7.
We chose six benchmarks from HeCBench [11] benchmark suite. Figure 6 shows the details of each benchmark and the command line arguments we used when running the benchmark. Each benchmark in the suite has four versions: CUDA, HIP, OpenMP target offloading, and SYCL. We ported the OpenMP kernel language version from the CUDA version and measured the performance of the following four versions:

- OpenMP kernel language: This version is compiled with our prototype. We refer to it as ompx in our plots and discussion.
- OpenMP target offloading: This version is compiled with LLVM/Clang, and we refer to it as omp.
- Native with LLVM/Clang: This version is compiled with LLVM/Clang, and we refer to it as cuda and hip respectively.
- Native with vendor compiler: This version is compiled with the corresponding vendor compiler. It is referred as cuda-nvcc and hip-hipcc respectively.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Command Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>XSBench</td>
<td>Monte Carlo neutron transport algorithm</td>
<td>-m event</td>
</tr>
<tr>
<td>RSBench</td>
<td>Monte Carlo neutron transport algorithm</td>
<td>-m event</td>
</tr>
<tr>
<td>SU3</td>
<td>Lattice QCD SU3 matrix multiply</td>
<td>-i 1000 -l 32 -t 128 -v 3 -w 1</td>
</tr>
<tr>
<td>AIDW</td>
<td>Adaptive inverse distance weighting</td>
<td></td>
</tr>
<tr>
<td>Adam</td>
<td>Adaptive moment estimation</td>
<td></td>
</tr>
<tr>
<td>Stencil 1D</td>
<td>1D version of stencil computation</td>
<td></td>
</tr>
</tbody>
</table>

Figure 6: Benchmarks including brief summary and the command line arguments.

4.2 Experiment Results

Figure 8 presents the execution times for each benchmark version on both the NVIDIA A100 system (Figure 8a to Figure 8f) and AMD MI250 system (Figure 8g to Figure 8l). The benchmarks themselves reported all execution times.

4.2.1 XSBench. XSBench [28] serves as proxy applications for the Open Monte Carlo (OpenMC) project [19], which employs the monte carlo methodology to simulate neutron and photon transport. XSBench focuses on a memory-intensive implementation, computing the continuous energy macroscopic neutron cross-section lookup, a critical component in neutron transport studies. The execution times for all versions on both systems are depicted in Figures 8a and 8g. Notably, the ompx version consistently outperforms the native versions compiled with both LLVM/Clang and the vendor compiler across both systems. The results from the omp version were excluded due to the benchmark reporting an invalid checksum, rendering the results non-comparable.

4.2.2 RSBench. Similar to XSBench, RSBench [27] is also a proxy applications for the OpenMC project. However, it distinguishes from XSBench by offering a compute-bound implementation. The execution times for all versions on both systems are shown in Figures 8b and 8h. Similar to the XSBench results, the ompx version exceeds the performance of the native version compiled with LLVM/Clang on both systems. Interestingly, on the NVIDIA A100 system, the omp version outperforms the CUDA version. Profiling results suggest that, despite a higher register usage (162), the omp version leverages 2KB of shared memory. This indicates that the heap-to-shared optimization in OpenMP [9] effectively moves certain data to shared memory, thereby enhancing performance.

4.2.3 SU3. SU3 implements a sparse matrix-matrix multiplication routine specific to the Special Unitary group of order 3, denoted as SU(3). The core computation is derived from MLIC-Lattice QCD [3]. This application delves into the quantum chromodynamics (QCD) theory, which studies the strong interactions between quarks and gluons. When evaluated on the NVIDIA A100 system, as shown in Figure 8c, the ompx variant lags behind the CUDA version by approximately 9%. A deeper dive into the profiling data shows that the CUDA version is more register-efficient, utilizing 24 registers compared to the 26 used by the ompx version. A detailed examination of the PTX code generated for ompx version further reveals that despite functions inlined into the kernel, these functions are not eliminated, leading to a substantial device binary size difference (29KB for ompx versus an only 3.9KB for CUDA). On the AMD system, as shown in Figure 8i, the ompx version outperforms the HIP version by 28%. Across both systems, the ompx version consistently demonstrates better performance over the omp version.

4.2.4 AIDW. Adaptive inverse distance weighting (AIDW) in the benchmark suite is an efficient parallel AIDW algorithm by adopting fast k-nearest neighbors search [15]. As depicted in Figures 8d and 8j, on the AMD MI250 system, the OpenMP version aligns closely with the performance of the native version, irrespective of the compiler used. On the NVIDIA A100 system, while it matches the performance of the CUDA version compiled with nvcc, it is slightly slower by about 5% when compared to the version compiled using LLVM/Clang. A detailed examination of the PTX code generated for both the ompx and CUDA versions highlighted that shared variables within the kernel are demoted in the CUDA version.

4.2.5 Adam. Adaptive moment estimation (Adam) [13] is an important optimization algorithm in machine learning that computes adaptive learning rates for each parameter. As shown in Figures 8e and 8k, the ompx version matches the performance of the CUDA
Figure 8: Performance comparison of the six benchmarks on NVIDIA A100 system (Figure 8a to Figure 8f) and AMD MI250 system (Figure 8g to Figure 8l). Each figure has four bars: "ompx" stands for the OpenMP kernel language version, "omp" is the original OpenMP target offloading version, "cuda" and "hip" represent the native version compiled with LLVM/Clang, and "cuda-nvcc" and "hip-hipcc" are the native version compiled with vendor compiler. The dotted line is the baseline, which is "cuda" and "hip" version respectively.
version on NVIDIA A100 system, and 16.6% faster than the HIP version on AMD MI250 system. The omp version is 8x slower due to an issue in LLVM OpenMP that results in the launch of only 32 threads per thread block.

4.2.6 Stencil 1D. Stencil is a classical numerical data processing computation which updates array elements according to some fixed pattern. The CUDA version was adapted from a CUDA tutorial demonstrating the use of shared memory. Figures 8f and 8i shows that with our extension, the ompx version can outperform the native version on both systems. The omp version is significantly slow due to the inability to rewrite the generic state machine [9].

5 RELATED WORKS

Solutions aiming for performance portability in GPU programming typically fall into two main categories: programming models equipped with dedicated compilers and libraries that provide an abstraction layer over multiple programming paradigms. The first category includes models like OpenMP (from version 4 onwards) [17], OpenCL [12], and SYCL [23] for computational tasks, as well as OpenGL [20] and Vulkan [24] for graphical tasks. While these models offer portability across diverse hardware platforms, they come with their own challenges in terms of ease of use and performance tuning. Notably, for programs already adapted to kernel languages, transitioning to any of these models is seldom straightforward. HIP [1] can technically be counted as portable since it can target both AMD and NVIDIA GPUs. The second category features solutions like Kokkos [6], RAJA [2], and the parallel algorithms introduced in C++17 [10]. Given their need to abstract over pre-existing programming models, these solutions often present higher-level interfaces. This abstraction can lead to increased overhead and a more intensive porting process, especially for code originally crafted in a SIMT style.

Furthermore, there have been works aimed at enabling proprietary programming models to function on diverse platforms. Examples include MCUDA [22], COX [8], and CuP2P [7]. While these efforts facilitate some degree of portability to CPUs, they often come with performance trade-offs and do not extend support to other GPU platforms.

Since the introduction of target offloading, OpenMP has emerged as a promising performance-portable programming model for GPUs. This capability was further realized in LLVM through the efforts of Doerfert et al. [5], Huber et al. [9] by optimizing OpenMP for efficient execution on GPUs. Beyond targeting accelerators, Lu et al. [14], Patel and Doerfert [18], Shan et al. [21] expanded the LLVM OpenMP target offloading support to encompass multi-node computation, pushing OpenMP’s capabilities beyond just intra-node computation. Building on the performance portability of OpenMP and the existing LLVM/OpenMP target offloading infrastructure, Doerfert et al. [4] introduced a compiler-based method to convert CUDA code into portable OpenMP code. Our work draws from their user-facing APIs, integrating them into our proposed extension.

6 CONCLUSION AND FUTURE WORK

In this work, we introduced novel extensions to LLVM OpenMP that enable developers to write GPU code in a SIMT style that is similar to kernel languages, while still benefiting from the performance portability that OpenMP offers. These extensions aim to make the transition from kernel languages to OpenMP more straightforward, often reducing the process to simple text replacements. Our proof-of-concept implementation and subsequent evaluations underscored the efficiency of our approach. Not only does our solution simplify the porting process, but it also ensures that performance is on par with native kernel languages.

In the future, we will fully implement all the proposed extensions within the LLVM framework. Beyond that, we plan to refine these extensions further. A significant avenue of exploration for us will be the potential integration of these extensions with code rewriting tools. This integration aims to simplify the transition from kernel languages to OpenMP, further reducing the burden on developers.

ACKNOWLEDGMENTS

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-854558). We also gratefully acknowledge the computing resources provided and operated by the Joint Laboratory for System Evaluation at Argonne National Laboratory.

REFERENCES


registry.khronos.org/OpenCL/specs/3.0-unified/pdf/OpenCL_API.pdf

mization. In International Conference on Learning Representations (ICLR) arXiv,

[14] Wenbin Lu, Baodi Shan, Eric Raut, Jie Meng, Mauricio Araya-Polo, Johannes
Doerfert, Abid Muslim Malik, and Barbara M. Chapman. 2022. Towards Efficient
Remote OpenMP Offloading. In International Workshop on OpenMP (IWOMP),
3-031-15922-0_2

Adaptive IDW Interpolation Algorithm Using Fast kNN Search. CoRR abs/1601.05904


High Performance (ISC) Lecture Notes in Computer Science, Vol. 13289. Springer,
Hamburg, Germany, 315–333. https://doi.org/10.1007/978-3-031-07312-0_16

[19] Paul R. Romano and Benoit Forget. 2013. The OpenMC Monte Carlo Particle


2023. MPI-based Remote OpenMP Offloading: A More Efficient and Easy-to-
use Implementation. In International Workshop on Programming Models and
Applications for Multicores and Manycores (PMAM). ACM, Montreal, QC, Canada,
50–59. https://doi.org/10.1145/3582514.3582519

Implementation of CUDA Kernels for Multi-core CPUs. In International Workshop
on Languages and Compilers for Parallel Computing (LCPC) Lecture Notes in
Canada, 16–30. https://doi.org/10.1007/978-3-540-89740-8_2


registry.khronos.org/vulkan/specs/1.3-extensions/pdf/vkspec.pdf

Experience Report: Writing a Portable GPU Runtime with OpenMP 5.1. In Inter-
national Workshop on OpenMP (IWOMP), September 14-16, 2021, Vol. 12870.

Execution of Deferred OpenMP Target Tasks with Hidden Helper Threads. In
Languages and Compilers for Parallel Computing (LCPC), October 14-16, 2020,
Vol. 13149. Springer, Stony Brook, NY, USA, 41–56. https://doi.org/10.1007/978-
3-030-95953-1_4

[27] John R. Tramm, Andrew R. Siegel, Benoit Forget, and Colin Josey. 2014. Per-
formance Analysis of a Reduced Data Movement Algorithm for Neutron Cross
Section Data in Monte Carlo Simulations. In International Conference on Exascale
Sweden, 39–56. https://doi.org/10.1007/978-3-319-15976-8_3

XSbench - The Development and Verification of a Performance Abstraction
for Monte Carlo Reactor Analysis. In International Conference on Physics of
Reactors (PHYSOR), September 28 - October 3, 2014 JAEA, Kyoto, Japan, 1–12.
http://dx.doi.org/10.11484/jaea-conf-2014-003