LLNL-CONF-834862

Breaking the Vendor Lock - Performance Portable Programming Through OpenMP as Target Independent Runtime Layer

J. Doerfert, M. Jasper, J. Huber, K. Abdelaal, G. Georgakoudis, T. Scogland, K. Parasyris

May 5, 2022

31st International Conference on Parallel Architectures and Compilation Techniques (PACT 2022)
Chicago, IL, United States
October 10, 2022 through October 12, 2022

Breaking the Vendor Lock — Performance Portable Programming Through OpenMP as Target Independent Runtime Layer

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

Marc Jasper
jasper3@llnl.gov
Lawrence Livermore National Laboratory, Livermore, USA

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

Khaled Abdelaal
khaled.abdelaal@ou.edu
University of Oklahoma, Norman, USA

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

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

Konstantinos Parasyris
parasyris1@llnl.gov
Lawrence Livermore National Laboratory, Livermore, USA

ABSTRACT

High performance computing (HPC) systems pervasively feature GPU accelerators. For maximum efficiency, these are usually programmed using vendor-specific languages, such as CUDA. However, this is not portable and leads to vendor lock-in. Existing portable proramming models require transcribing the whole application, which is tedious and often results in sub-optimal performance without necessarily avoiding the need to maintain multiple versions. Although solutions for automated translation exist, they sacrifice either features of the original model, performance, or both.

We propose a novel compiler-based approach for performance portable programming of GPUs by generating portable code from the original, vendor-specific application source. Specifically, we present LLVM/Clang extensions for performance portable CUDA by leveraging the existing LLVM/OpenMP offloading infrastructure for portable execution on different GPU architectures. Our contributions include: re-designing the compiler driver for portable toolchain generation, defining a target independent math library, and re-architecting compiler lowering from CUDA APIs to existing and new OpenMP runtime calls. We evaluate our approach using six established CUDA proxy and benchmark applications first on NVIDIA GPUs, to measure the overhead of our portability layer, then secondly on AMD GPUs, to determine the efficacy of our approach. In both experiments we compare the performance to native program versions, i.e., CUDA and HIP. Our approach has minimal overhead compared to non-portable alternatives, thus providing viable performance portability for existing code without cost to the user. We further show CUDA code debugged directly on the host.

PACT '22, Oct. 10–12, 2022, Chicago,IL
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

CCS CONCEPTS

• **Software and its engineering → Compilers**.

KEYWORDS

Performance portability, CUDA, GPGPU, OpenMP, AMDGPU, LLVM

ACM Reference Format:
Johannes Doerfert, Marc Jasper, Joseph Huber, Khaled Abdelaal, Giorgis Georgakoudis, Thomas Scogland, and Konstantinos Parasyris. 2022. Breaking the Vendor Lock — Performance Portable Programming Through OpenMP as Target Independent Runtime Layer. In *PACT '22: International Conference on Parallel Architectures and Compilation Techniques (PACT), Oct. 10–12, 2022, Chicago, IL.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Performance Portable Programming is an abstract goal in which diverse hardware can be interchanged without requiring program porting, tuning, or maintenance efforts. At the beginning of the GPGPU era the difference in programming difficulty between the two available GPGPU targets was so large that it led to the enormous (and enormously successful) CUDA ecosystem. However, as a proprietary programming language, CUDA can traditionally only be compiled to NVIDIA GPUs. With the advent of greater GPU diversity, new programming languages emerged with the promise of being as performant as CUDA while portable across different (GPU) vendors. Given the manifold of choices and their vastly varying support on actual diverse hardware, users are rightfully confused. Further, any change in programming language requires a complex and expensive porting of existing applications. While some automation exists, there are no clear answers on the horizon. A major reason is the feature divergence between the different parallel programming language options. HIP, for example, subsumes most of CUDA up to version 8, but later extensions are not available. Consequently, a port to HIP prevents the use of new hardware and software features even if the HIP code is executed on an NVIDIA GPU that would provide all required support.

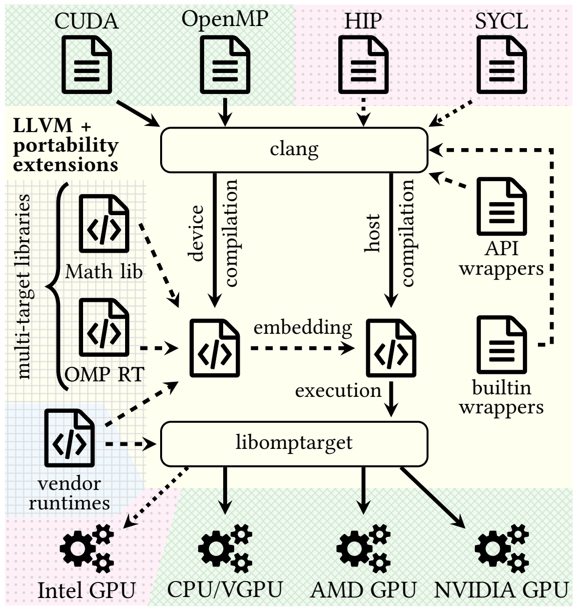


Figure 1: Overview of our LLVM compiler with portability extensions. The already supported inputs and outputs are shown with a green crosshatch background, the ones on a red dotted background can be dealt with in the same way. All parts of the compilation pipeline shown on light yellow background have been implemented or modified, the grid background on the left indicates the target dependent parts.

For this work we took a step back and started from the same place as many HPC users, with CUDA code. First off, CUDA has proven to be a great asset in GPGPU programming. While other models might provide convenient features or some degree of portability, it is arguably not always enough to justify an expensive porting effort with uncertain outcomes. For this reason, a non-trivial amount of users are interested in keeping the known-to-work CUDA code around, potentially augmented with other kernels to explore alternative hardware. If we assume CUDA is not going to be replaced, and we assume duplication of compute kernels is not a viable long term strategy either, we need to make CUDA performance portable instead. In fact, we argue that portability is only the first step and users actually want *fully interoperable* and *performance portable* parallel programming languages that can be combined as needed.

In this work we present a solution for performance portable GPU programming that is (almost) fully transparent to the user. Our prototype CUDA compiler, conceptually depicted in Figure 1, targets an augmented OpenMP offload runtime which is effectively a target independent API on top of different GPU runtimes, e.g., CUDA and HSA. Through multiple independent wrapper layers we redirect explicit CUDA API calls issued by the user, implicit CUDA API calls generated by the compiler, as well as CUDA builtin function calls to respective OpenMP versions. Together with a new inter-operable compiler driver, a portable math library for GPUs, and augmented OpenMP runtimes on the host as well as the device, we can execute CUDA programs with minimal performance penalty on AMD GPUs. Through the virtual GPU target [24], host execution of CUDA is feasible as well. Notably, our strategy is not restricted to CUDA. With wrappers for HIP and SYCL/DPC++, following the presented scheme, one can port these languages to all supported targets and further combine them with each other and CUDA alike.

In the following, we summarize the contributions and limitations of this work before we briefly explain offload compilation via LLVM/Clang in Section 3. The necessary steps to achieve portability and interoperability are described in Section 4. Additional benefits of the LLVM/OpenMP layer are introduced in Section 5. An evaluation on six HPC proxy and benchmark applications is presented in Section 6. Before we conclude in Section 9, we discuss related work in Section 7 and future extensions in Section 8.

2 CONTRIBUTIONS AND LIMITATIONS

This work introduces a compiler prototype based on LLVM/Clang which takes a CUDA program as input and compiles it for NVIDIA or AMD GPUs, or the host CPU. The main contributions are:

- The CUDA compiler prototype that can target different GPU and CPU architectures by utilizing augmented LLVM/OpenMP offloading runtimes as the target of independent wrapping layers for CUDA host and device APIs.
- Augmented LLVM/OpenMP runtimes that expose more control to the user, e.g., for kernel launch parameters, and which provide target independent accessors not required for standard OpenMP.
- A target independent standard math library for GPUs, i.e., `libm` for GPUs, that allows portability for math functions and enables more compiler optimizations for them in the process.
- A new compiler driver scheme which uses a novel embedding of device code in the host object files to allow for interoperability between offloading models, so far OpenMP and CUDA.
- The ability to use the OpenMP ecosystem for CUDA codes, including but not limited to: the OpenMP tooling interface (OMPT), and remote OpenMP offloading [23].

Our work is a research prototype with limitations, including:

- We only support a subset of the CUDA runtime API. This is not a conceptual limitation and more can be ported as required. Similarly, we do not support libraries like NVIDIA CUB or Thrust yet. Both require more than new wrappers around CUDA API calls as they contain, among other things, NVIDIA specific assembler. We describe possible solutions in Section 8.
- Our capabilities are based on, and thereby limited by, language support of LLVM/Clang and offload support in LLVM/OpenMP. In particular, community LLVM does not yet support Intel GPUs.

3 BACKGROUND

Before we approach compilation for a target independent intermediate layer, we need to understand the current compilation model LLVM/Clang uses for offloading. While we discuss CUDA in particular, the same ideas hold for HIP, OpenMP, and SYCL as well.

(1) The Clang driver pre-includes headers distributed together with LLVM/Clang. These headers define CUDA builtins, like `threadIdx.x`, and declare various functions provided by the `libdevice` library included with the CUDA installation.
(2) User includes of C++ headers, e.g., `complex`, are intercepted through additional include paths prepended by the Clang driver.

Clang headers wrap system headers with a modified environment that makes them (better) usable on the device.

(3) Clang processes the CUDA source code as it would with codes written in other languages. However, certain constructs will result in CUDA API calls not present in the user program. Of note are kernel launches via the triple chevron (`Kernel<<<GrdDim, BlckDim>>>(arg)`) which are lowered to `cudaLaunchKernel`, and the registration code emitted into the host object file for kernels and global variables exposed in the device code.

(4) The CUDA `libdevice` library is linked into the LLVM-IR generated for the device side before it is optimized and lowered to PTX. This library provides definitions for device functions, including math functions like `__nv_sin`.

We reused the same techniques but modified all steps to break the dependence on the CUDA installation unless the targeted is an NVIDIA GPU. When we target an AMD GPU we will link in the AMD counterparts to the NVIDIA/CUDA specific libraries. For host debugging we use the same libraries for host and device.

4 APPROACH

Making an accelerator offloading language portable requires severing ties to the target specific environment. All compilation steps, except code generation, should be independent of the target architecture. We describe this in general, but provide examples rooted in the changes necessary to re-target CUDA. Other languages, such as HIP and SYCL, will require similar steps but can largely reuse parts that do not deal with CUDA types or API functions explicitly.

4.1 Original Language Dependencies

There are various kinds of dependencies on particular functionality distributed as part of the original language software development kit (SDK). Conceptually, one can choose to either use (i) all, (ii) some, or (iii) none of the features distributed with the language, e.g., headers. With option (i) all symbols are available but not all of them will have been wrapped properly, resulting in hard-to-detect runtime errors when non-supported hardware is targeted. It is further not guaranteed that the portability layer and the original language will inter-operate on supported hardware, resulting in hard to debug execution time incompatibilities. The second option (ii) is a middle ground that would only use original headers. However, it is impossible to control how much is included transitively and it would require wrapping various unused functions just to make them available for the linker. While one can automate the process of providing non-functional wrappers for linking purposes, there are other complications in original headers, e.g., inline assembly, that complicate this approach. Lastly, option (iii) avoids the SDK entirely as not to mix it with the portability layer. This ensures that errors are generally flagged early in the compilation process and only the symbols used by the application are actually needed. However, since types, among other things, are now provided by the portability layer, we cannot mix in any original language parts without risking clashes, e.g., multiple definitions of the same name.

For this work, we chose option (iii) by completely cutting ties with the CUDA installation during the compilation and instead provide the necessary definitions ourselves. Said differently, we effectively passed the `-nogpulib` and `-nogpuinc` command line options to the LLVM/Clang driver as it ensures no CUDA package components are used. In the following, we discuss the different parts that require handling if the original SDK is not available.

```
inline cudaError_t cudaMalloc(void **DevPtr, size_t Size) {
  if (omp_get_default_device() == omp_get_initial_device())
    return __last_error = cudaErrorNoDevice;
  *DevPtr = omp_target_alloc(Size, omp_get_default_device());
  if (*DevPtr == nullptr)
    return __last_error = cudaErrorMemoryAllocation;
  return __last_error = cudaSuccess;
}
```

Figure 2: Definition of `cudaMalloc` in our `cuda_runtime_api.h` header. Existing OpenMP APIs are used with additional control logic. The value of `__last_error` is used in `cudaGetLastError`.

4.1.1 Headers. The first issue that appears when the original language SDK is not used is missing headers. Users include language dependent headers, e.g., `cuda.h`, in order to get access to types and functions not predefined by the compiler. We employ the same trick LLVM/Clang used to intercept includes of system headers, ref. (2) in Section 3. A custom include path is added to LLVM/Clang to expose common headers provided by the language SDK. Our implementations of these are similar to the SDK versions in that they provides type definitions (incl. `cudaError`), macros (incl. `__shared__`), and runtime function declarations. The latter are special as we do provide definitions with inline linkage right away rather than functions declarations that serve as entry points into the original language runtime. As illustrated in Figure 2 the original API functions are implemented with corresponding functionality exposed by our extended OpenMP offload runtime together with logic to account for error handling and signature differences.

4.1.2 API Calls. In Figure 2 we have already shown how user facing APIs are provided by our custom implementations of common language headers. In addition, we also need to handle compiler generated calls to the original language runtime which generally do not have a counterpart in the user facing API of OpenMP. The best example is the triple chevron syntax (`Kernel<<<GrdDim, BlckDim>>>(arg)`) which is lowered to `__cudaPushCallConfiguration` as well as `__cudaPopCallConfiguration` and finally a call to `cudaLaunchKernel`. To provide these implicitly defined functions, we pre-include headers into the application as LLVM/Clang does for regular CUDA compilation, ref. (1) in Section 3. Their implementation involves manipulation of thread-local global state and calls into the OpenMP offload runtime parts usually only targeted by LLVM/Clang to implement things like target directives (`#pragma omp target`). To support multi-dimensional kernel launches, user provided streams, and dynamic shared memory, we created a new entry point in the OpenMP offload runtime that accepts the same parameters as the regular CUDA kernel launch does. LLVM/OpenMP already supports streams when NVIDIA GPUs are targeted so their handling can be integrated easily. Once the AMD GPU plugin in LLVM/OpenMP gains support for "streams" we can implement CUDA streams with those. For the time being, streams are effectively ignored when we target AMD GPUs. For dynamic shared memory we added support into the LLVM/OpenMP offload runtime which makes the feature available in the same way as CUDA exposes it to the user, hence through an external global array in `__shared__` memory space.

4.1.3 *Builtins.*

Language builtins are pre-defined by the compiler either through explicit handling of expressions or via declarations in pre-included header files. For this work we reused the implementation of the globals `threadIdx`, `blockIdx`, `blockDim`, and `gridDim` LLVM/Clang provides in its pre-included headers. Accesses to the `x`, `y`, and `z` members are translated to function calls via the `__declspec (property(get=getter))` construct. The actual getters are the builtin functions `__nvvm_read_ptx_sreg_[n]{cta,t}id_x`. As the user, a library, or the compiler itself might call these builtin functions explicitly we provide a wrapper for them rather than a different lowering of the member accesses to the `{Thread,Block}Idx` and `{Block,Grid}Dim` globals. To get a portable device API we followed the design of the OpenMP device runtime introduced by Tian et al. [28]. The target independent interface exposed to the compiler are implemented differently for each target architecture. The entire lowering scheme for an access to `threadIdx.x` is shown in Figure 3. Other device-side builtins, such as `__syncthreads`, are implemented the same way. Some can directly be lowered to existing target independent entry points of the OpenMP device runtime. For others, like the three dimensional block and grid coordinates, we extended the API as required.

__clang_cuda_builtin_vars.h - existing - LLVM/Clang

```
// Declare a global `threadIdx` and define accesses to the x
// member as calls to `__nvvm_read_ptx_sreg_tid_x` builtin.
struct __cuda_builtin_threadIdx_t {
  __declspec(property(get = __fetch_builtin_x)) unsigned x;
  static inline __attribute__((always_inline, device))
  unsigned __fetch_builtin_x(void) {
    return __nvvm_read_ptx_sreg_tid_x;
  }
  // ...
};

extern const __attribute__((device, weak))
__cuda_builtin_threadIdx_t threadIdx;
```

__openmp_cuda_device_wrapper.h - new - LLVM/Clang

```
// The `__nvvm_read_ptx_sreg_tid_x` familiy of builtins is defined
// through new target independent OpenMP device functions.
__device__ uint32_t __kmpc_get_hardware_thread_id_in_block_x();

__attribute__((device, always_inline, flatten))
inline unsigned __nvvm_read_ptx_sreg_tid_x() {
  return __kmpc_get_hardware_thread_id_in_block_x();
}
```

DeviceRTL/src/Mapping.cpp - extended - LLVM/OpenMP

```
// The target independent entry points are implemented based on
// the target architecture the device runtime is compiled for.
#pragma omp begin declare variant match(device={arch(amdgpu)})
uint32_t __getHardwareThreadIdInBlockX() {
  return __builtin_amdgcn_workitem_id_x();
}
#pragma omp end declare variant

#pragma omp begin declare variant match(device={arch(nvptx64)})
uint32_t __getHardwareThreadIdInBlockX() {
  return __nvvm_read_ptx_sreg_tid_x();
}
#pragma omp end declare variant

extern "C" uint32_t __kmpc_get_hardware_thread_id_in_block_x() {
  return __getHardwareThreadIdInBlockX();
}
```

Figure 3: The three steps used to lower `threadIdx.x` to the respective CUDA or AMDGCN builtin that retrieves the value at runtime. The top part is already included with LLVM/Clang's CUDA implementation and results in the translation to the target dependent builtin call `__nvvm_read_ptx_sreg_tid_x`. The two lower parts have been added to map the NVIDIA builtin first to a target independent one (`__kmpc_get_hardware_thread_id_in_block_x`) and then to the proper implementation for the user chosen target architecture (e.g., `__builtin_amdgcn_workitem_id_x` for AMD GPU targets). Note that the code in the bottom part is included in the OpenMP device runtime and compiled for all supported architectures. The offload driver will link in the right version as part of the device code link and optimization step. As such, all indirections will be folded and `threadIdx.x` will result in a single builtin call, as it would with native CUDA or HIP compilation.

4.1.4 *Assembly.*

Inline assembly is inherently target specific. The presence of inline assembly for a different architecture, even in dead code, does often cause problems as the compiler frontend is unable to verify the (few) syntactic and semantic constraints assembly has to fulfill. Since most user applications, including our benchmarks, do not use target specific assembly we did not implement support for it. However, libraries, e.g., NVIDIA CUB, do use inline assembly and will consequently not compile to a different architecture seamlessly. To leverage library functions not utilizing assembly one could delay assembly-related errors in LLVM/Clang and only emit them if the code is emitted. To further allow assembly in an application one would need to replace it with appropriate code for the target architecture, or a target independent abstraction.

4.2 Portable Offloading

We already introduced different layers to implement APIs or builtins of the original language in target agnostic ways. However, creating offload binaries is conceptually more complicated than host only compilation. Both host and device code require separate compilation as well as linking steps that need to be orchestrated. Furthermore, the resulting binaries must be combined such that cross-device references are resolved properly at runtime. All that has to work without disturbing existing build infrastructure, requiring that every build step can only produce a single object file and no additional commands shall be required from the user. While all offload languages supported by LLVM/Clang already deal with these complications, they do so on a per language basis. To support re-targeting and full interoperability we needed to unify offload compilation.

4.2.1 *Offload Driver.*

The LLVM/Clang compiler driver is responsible for generating the necessary steps to create the final object file, executable, or library. For this work we replace the existing offloading drivers of OpenMP and CUDA with a truly target independent version that (mostly) unifies their respective toolchains. Our new LLVM/Clang offloading driver contains two major changes compared to the existing offloading drivers. First, we use a unified scheme to compile and embed device objects into the host object. Second, we combined all the complexity of device linking into a single stage that augments the normal host linking step. Device linking was previously handled in the clang driver itself uniquely for each toolchain, which made it impossible to compile different

```
struct __offload_entry {
  void *addr;
  char *name;
  size_t size;
  int32_t flags, reserved;
};
__offload_entry entry __attribute__((section("offloading_entries")));

// Linker defined symbols used by the runtime.
extern struct __offload_entry __start_offloading_entries;
extern struct __offload_entry __stop_offloading_entries;
```

Figure 4: Registration scheme for device globals and kernels that require host access. For each symbol an offload entry is placed in a special ELF section (`"offloading_entries"`), the section is later inspected by the OpenMP runtime.

toolchains together or handle static libraries with device code correctly. In addition to targeting the OpenMP runtime from CUDA, this new driver allows us to mix and match CUDA and OpenMP device code. We further support linking device libraries late which allows to optimize target independent code early and include target dependent implementations only after generic optimizations, e.g., for math functions, have been performed.

Code Embedding. Offloading schemes embed device code into the host object file to work seamlessly with existing build setups. Our unified embedding scheme wraps the device code in a new binary format that contains necessary metadata for interoperability, e.g., what offload target was used. This binary file is embedded into the host ELF file as a section with the `SHF_EXCLUDE` flag set. Our new augmented linker step extracts device code from all object files and (static) libraries. The necessary linking steps for the target are performed using vendor tools. Finally, runtime registration code is generated for symbols that need be accessed by the host.

Symbol Registration. Offloading languages require externally visible global symbols, which includes kernels, to be first registered by the host before they can be used, e.g., for a host-to-device memcpy or kernel launch. In order to unify the offloading languages and achieve interoperability we must be able to register globals from both CUDA and OpenMP equivalently. We accomplished this by changing LLVM/Clang's CUDA code generation to create the same offloading entries as LLVM/OpenMP uses. The scheme, sketched in Figure 4, is much simpler than the original handling for CUDA symbols. A device-side global is registered by creating a host-side `__offload_entry` in a special ELF section (`"offloading_entries"`). The linker defined symbols `__start_SECNAME` and `__stop_SECNAME` allow generic registration code at runtime to iterate all offload entries regardless their origin. The actual registration with the vendor drivers is then performed by the OpenMP offload runtimes.

4.2.2 *Target-Specific Device Libraries.*

GPU vendors provide their own device libraries that implement builtins and parts of the standard library, e.g., common math functions, in target dependent ways. Offload schemes in LLVM/Clang include these libraries early and combined them with headers that defined standard library functions, including math functions, eagerly through target dependent alternatives. As an example, CUDA and OpenMP offload provide math overlay headers that define `sin` as an inline function returning the result of `__nv_sin`. The latter is then defined in the early linked in LLVM-IR `libdevice` library, which is part of a CUDA installation.

To provide portability and allow the LLVM middle-end to optimize calls to known runtime functions, especially math functions, we delayed inclusion of target dependent definitions and linking of target dependent code. To this end, we needed a generic GPU "`libm`" math library for each supported architecture.

We create our GPU math libraries directly from the GPU math headers used for CUDA, OpenMP, and HIP compilation. Instead of linking the headers into the application source code we compile them into standalone LLVM-IR bitcode files. During late linking we pick the vendor libraries and the math library for the target architecture and merge it with the application LLVM-IR code.

4.2.3 *Optimization Pipeline.*

To facilitate target independent optimizations, especially for "known" runtime calls like math functions, we run the optimization pipeline before we link in any target dependent libraries. To eliminate potential overhead of the wrappers that translate CUDA into generic APIs which are then implemented with target dependent code, we run the optimization pipeline again after all libraries have been linked into the device code. This dual optimization is possible as we run device code optimizations during the regular compilation stage as well as during the new augmented linker stage. The final compilation flow is sketched in Figure 5. The top part shows the compile-time actions and the bottom part illustrates the link-time steps. Note that the user chosen optimization level (`-OX`) is passed to `clang` as well as `llc`.

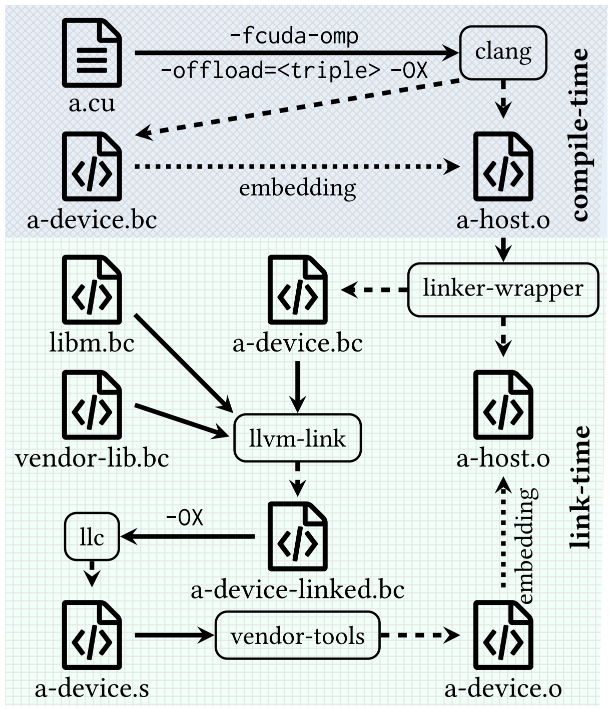


Figure 5: Compilation pipeline for CUDA inputs. The compile-time part (top, blue crosshatch background) optimizes the input without target specific code, though the device IR is target dependent. At link-time (bottom, green grid background), target specific code and vendor libraries are linked into the device code. The result is optimized again to eliminate potential indirection overheads.

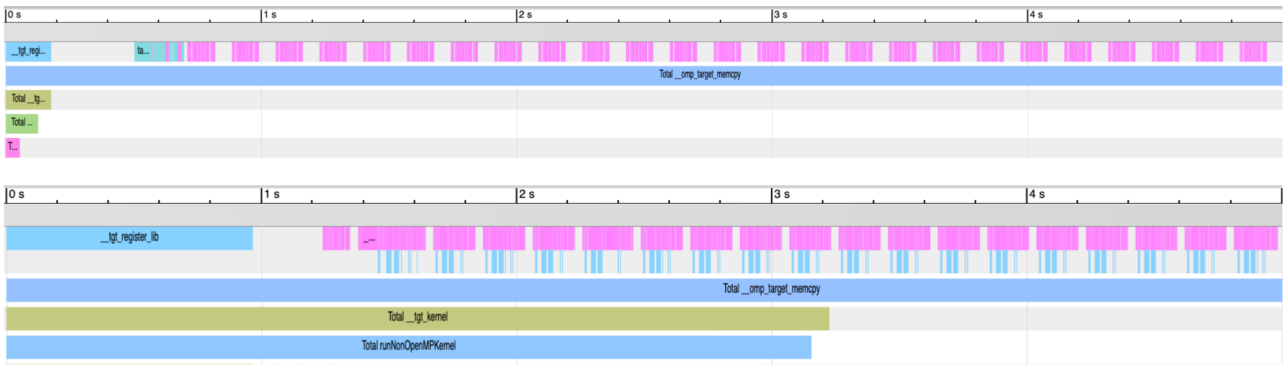


Figure 6: Screenshots of profile traces obtained via `LIBOMPTARGET_PROFILE` for the execution of Lulesh (CUDA version). The top part illustrates execution on an NVIDIA GPU (V100) and the bottom part is the same code executed on an AMD GPU (MI50).

5 LLVM/OPENMP ECOSYSTEM

Using the OpenMP runtimes as target independent runtime layer has benefits beyond portability and interoperability with OpenMP. The LLVM/OpenMP ecosystem has significantly grown in recent years and it provides unique capabilities, especially compared to other open source compiler support of parallel programming languages. Through this work, current and future development efforts in the LLVM/OpenMP space will become accessible to CUDA codes, and eventually also to other languages such as HIP and SYCL.

Features especially interesting to developers include advanced debugging support [7], a virtual GPU (VGPU) offloading target running on the host [24], as well as remote offloading [23, 31] which, combined with the VGPU, allows single-host out-of-process offloading to identify memory mapping errors on the host. CUDA executed through the LLVM/OpenMP infrastructure can be analyzed with the same environment flags used for OpenMP offloading, namely:

`LIBOMPTARGET_INFO=<bitset>` to get high-level information about memory movement and kernel execution;

`LIBOMPTARGET_DEBUG=<bitset>` to get verbose information about the inner workings of the offloading runtime; and

`LIBOMPTARGET_PROFILE=<json>` to get a Chrome tracing compatible profile file that lists the times of all device interactions.

In addition to development features, LLVM is equipped with an increasing number of OpenMP-aware optimizations performed in the middle-end compilation pipeline. The "OpenMP-opt" pass [17] in LLVM does, for example, deduplicate calls to the same OpenMP API function in a function scope if it is known that the function will always return the same value. Given that our CUDA API wrappers are implemented through OpenMP runtime functions, as shown in Figure 2, CUDA code directly benefits from this deduplication. Users can even verify this by adding `-Rpass=openmp-opt` to the `clang` invocation. Future investment, e.g., host-to-device optimizations or latency hiding transformations, will directly benefit CUDA codes as well. Similarly, currently developed runtime extension, e.g., specializing JIT compilation for OpenMP kernels, will be accessible to foreign codes executed through the same infrastructure without additional cost or involvement to the user.

To showcase capabilities that become accessible at no extra cost we present the lightweight LLVM/OpenMP profiling and host execution support for regular CUDA codes. Profiling is shown in Figure 6 for the Lulesh CUDA benchmark (ref. Table 2) executed on both NVIDIA (top) and AMD (bottom) GPUs. For the screenshots we used Chrome's visualization of the raw profile data. The first rows of each profile contain the trace while the bottom rows summarize the time spent in particular LLVM/OpenMP runtime functions. In this prototype we do not yet attribute source locations to runtime invocations but the interface is already available and can be used in the future.

Execution of CUDA code on the host is another benefit that comes with the LLVM/OpenMP environment. Through the VGPU offloading target the code is compiled and executed as if it was offloaded to a GPU, except that the actual execution is happening on the host CPU. Among the many benefits is the ability to use mature CPU tooling for device side code. An example gdb session for the SU3 benchmark (ref. Table 2) executed via the VGPU is presented in Figure 7. While the VGPU is not yet available in community LLVM, we did attach the open source prototype, which supports various GPU builtins like warp shuffles, to our CUDA compiler the same way we attached the CUDA and AMD plugin.

```
Thread 17 "su3" hit Breakpoint 1, k_mat_nn (a=0x[...]8b10, b=0x[...]
      cb20, c=0x[...]cc50, total_sites=256) at ./mat_nn_cuda.hpp:22
22 int myThread = blockDim.x * blockIdx.x + threadIdx.x;
(gdb) bt
#0 k_mat_nn (a=0x[...]8b10, b=0x[...]cb20, c=0x[...]cc50,
      total_sites=256) at ./mat_nn_cuda.hpp:22
#1 0x[...]d6dd in ?? () from /usr/lib64/libffi.so.7
#2 0x[...]9a69 in VGPUTy::VGPUTy()::{lambda()#2}::operator()() ()
   from [...]/lib/libomptarget.rtl.vgpu.so
(gdb) print myThread
$2 = 15
(gdb) next
25 if (mySite < total_sites) {
(gdb) cont
Continuing.

Thread 17 "su3" hit Breakpoint 2, k_mat_nn (a=<opt out>, b=<opt out>,
       c=0x[...]cc50, total_sites=256) at ./mat_nn_cuda.hpp:32
32 CMULSUM(a[mySite].link[j].e[k][m], b[j].e[m][l], cc);
(gdb) next
36 c[mySite].link[j].e[k][l] = cc;
(gdb) print cc
$3 = {real = 1, imag = 0}
```

Figure 7: Exemplary gdb session to debug the SU3 CUDA benchmark on the host. Compilation and execution via the VGPU target is intentionally close to GPU offloading. This hurts performance but improves debugability of GPU issues.

Table 1: CUDA API calls used by each benchmark

API call	XSBench	RSBench	LULESH	SU3	Triad	miniFE	Plugin Implementation NVIDIA	Plugin Implementation AMD
`cudaMalloc`	x	x	x	x	x	x	Stable	Stable
`cudaMallocHost`					x		Stable	Basic
`cudaMemcpy`	x	x	x	x		x	Stable	Stable
`cudaMemcpyAsync`					x		Stable	Basic
`cudaFree`	x	x	x	x	x	x	Stable	Stable
`cudaFreeHost`					x		Stable	Basic
`cudaMemset`						x	Stable	Stable
`cudaDeviceSynchronize`	x			x			Stable	Stable
`cudaThreadSynchronize`					x	x	Stable	Stable
`cudaGetDeviceProperties`	x	x					Stable	Basic
`cudaStreamCreate`						x	Stable	Basic

Table 2: Benchmarks including brief summary and inputs.

Name	Description	Command Line
XSBench (ver. 19)	Monte Carlo neutron transport algorithm	`-m event -s large`
RSBench (ver. 12)	Monte Carlo neutron transport algorithm	`-m event -s large`
LULESH (ver. 2.0)	Proxy that approximates hydrodynamic equations	`-i 100 -s 128 -r 11 -b 1 -c 1`
SU3	Lattice QCD SU(3) matrix-matrix multiply	`-i 100 -l 32 -t 128 -v 3 -w 1`
Triad	Tests data transfer speeds, part of the STREAM benchmark	`-passes 100`
miniFE (ver. 2.0)	Proxy for unstructured implicit finite element codes	`-nx 128 -ny 128 -nz 128`

Table 3: Hardware and software of the evaluation platforms.

	AMD + MI50	Power9 + V100
CPU	AMD EPYC 7401 2.00 GHz	IBM Power9
# cores	24	10
# sockets	2	4
Main Memory	256 GB	256GB
GPU	AMD Radeon Instinct MI50	NVIDIA Tesla V100-SXM2
GPU Memory	16 GB	16GB
GPU Software Stack	ROCm v5.0.2	CUDA v11.1.0
OS	RHEL v7.6	RHEL v8.5

6 EVALUATION

Our evaluation is designed to answer one central question: Can our target independent CUDA compiler prototype compete with the respective native programming models on NVIDIA and AMD GPUs without the need to change the source code in any way. To this end, we compare our prototype to (a) state-of-the-art vendor compilers for both NVIDIA and AMD devices with CUDA and HIP inputs respectively, and (b) the LLVM/Clang compiler underlying our prototype using the CUDA and HIP versions of the benchmark to target their respective native architectures.

We used the six benchmarks listed in Table 2 which we run with the shown input parameters. The source code was taken from the HecBench[1] benchmark suite. Table 1 summarizes the CUDA API calls present in each benchmark. Further descriptions are provided with performance result discussions in Section 6.2.

[1]https://github.com/zjin-lcf/HeCBench

6.1 Methodology

Our evaluation setup consists of two machines with AMD and NVIDIA GPUs, respectively. Hardware and software stack details are given in Table 3. We used the compilers distributed with the software stack, thus ROCm v5.0.2 and CUDA v11.1.0, respectively.

For each benchmark and each machine we create three executables. The first is compiled using the vendor provided compilation toolchain. So, for the NVIDIA system we use `nvcc` and for the AMD system we use `hipcc`. We refer to this version in our plots and discussion as *vendor-cc*. The second executable is always compiled through the original non-extended LLVM/Clang on which our prototype is based on. It is a development commit designated as LLVM 15. We refer to this version as *clang-cc*. Finally, the last executable is created using our prototype that extends the LLVM/ Clang toolchain. We refer to this version as *cuda-omp-cc*. Notably, when using the vendor compilers and unmodified LLVM/Clang we are required to compile the source version matching the GPU vendor of the machine. HecBench provides equivalent CUDA and HIP implementations for each benchmark, which we use for those compilations. Our prototype always uses the CUDA source of each benchmark regardless of the system.

For our performance evaluation, we measure the execution time of each executable ten times and present all results as dots in the plots. For all benchmarks, except Triad, we use the overall execution time as reported by the benchmark. For Triad we add timing code around the `RunBenchmark` function as there was no existing timer.

6.2 Experimental Results

In this section we present and discuss our experimental findings.

XSBench. XSBench is a proxy application for the Open Monte Carlo (OpenMC) project. OpenMC [26] simulates the transport of neutrons and photons using the Monte Carlo methodology. This proxy application [30] uses a memory-bound implementation to compute the continuous energy macroscopic neutron cross-section lookup when studying neutron transport.

Figure 8a shows the execution time of all six executables when run on the two tested systems. Our performance portable prototype compiler (*cuda-omp-cc*) outperforms both the vendor compiler, `nvcc` and `hipcc`, respectively, as well as the native LLVM/Clang compiler. We used `nvprof` to extract execution traces on the *Power9 + V100* system. The execution time spent on all CUDA related operations, including the kernel, is statistically the same except for the device allocation. Specifically, the *cuda-omp-cc* compiled version spends 200*ms* less on allocating device data in comparison with the other two executables. Note that the OpenMP offload runtime, which resolves the `cudaMalloc` application call when *cuda-omp-cc* is used, is employing `cuMemAlloc` internally, not `cudaMalloc`.

RSBench. RSBench is also a proxy application for the Open Monte Carlo (OpenMC) project. However, RSBench [29] provides a compute-bound alternative implementation to XSBench.

Figure 8b depicts the results of the RSBench experiments. For both architectures the *cuda-omp-cc* compiled binaries achieved significantly lower execution times. The traces of both *nvprof* and *rocproc* for all executable versions indicate that the root cause of the poor performance of the *vendor-cc* and *clang-cc* is the computation of the main kernel (`lookup`). As described in Section 4.2.3, the *cuda-omp-cc* compilation executes two distinct optimizations pipelines. In the first, math functions are only available as compiler-known declarations. In the second, their target specific implementation has been linked in. Only in this model we observed that math calls originally in the innermost loop, part of `fast_nuclear_W`, have been hoisted out. To verify that this code movement is the reason of the performance improvement we manually hoisted two completely invariant sub-expressions out of the loop. For the *vendor-cc* generated executable, execution time is reduced by 0.3 and 3.3 seconds on the *Power9 + V100* and *AMD + MI50* systems respectively. This confirms that the observed performance gains are mainly due to the dual optimization pipeline which allows math-knowledge aware transformations before complex target specific implementations are linked in.

LULESH. LULESH [16] is a hydrodynamics proxy app on hexahedral mesh that approximates the Sedov problem. LULESH runs for a number of iterations, during each a collection of node-centered and element-centered properties are updated depending on the previous values of neighboring nodes and elements.

Figure 8c presents the performance measurements of LULESH. In the *Power9 + V100* system both executables produced by LLVM/-Clang (*clang-cc* and *cuda-omp-cc*) present a slow-down of approximately 10% compared to the *vendor-cc* executable. This indicates that `nvcc` is generally more successful optimizing this benchmark. For the *AMD + MI50* system the results show a 1.14× slowdown for *cuda-omp-cc* compared to the fastest version produced by the standard LLVM/Clang. The difference stems from the fact that our prototype compiler currently issues 1.113× more asynchronous memory copies to the HSA runtime while copying the same overall amount of data. We believe the underlying cause to be in the LLVM/OpenMP offload plugin for AMD which is not as mature as the NVIDIA one. As the former plugin matures the execution time gap on the AMD system should naturally disappear.

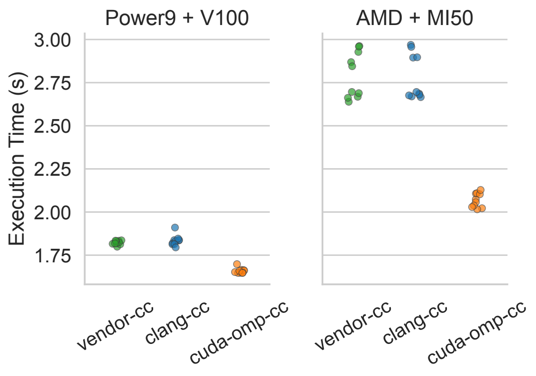
SU3Bench. SU3Bench implements a sparse matrix-matrix multiply routine for the Special Unitary group of order 3 (hence SU(3)). This kernel is extracted from MLIC-Lattice QCD [4], an application related to quantum chromodynamics (QCD) theory that studies the strong interaction between quarks and gluons. While the benchmark is available in multiple languages and frameworks we only used the CUDA and HIP versions for our evaluation.

Figure 8d shows the execution time for the SU3 benchmark. In the *Power9 + V100* systems the executables produced by LLVM/-Clang (*clang-cc* and *cuda-omp-cc*) are 1.16× faster than the `nvcc` produced version. On the *AMD + MI50* system the performance difference is smaller with the *clang-cc* compiled executable performing best. The *cuda-omp-cc* version showcases a large deviation across measurements. Observing the *rocprof* traces indicate the cause to originate in the implementation of the LLVM/OpenMP offload AMD plugin.

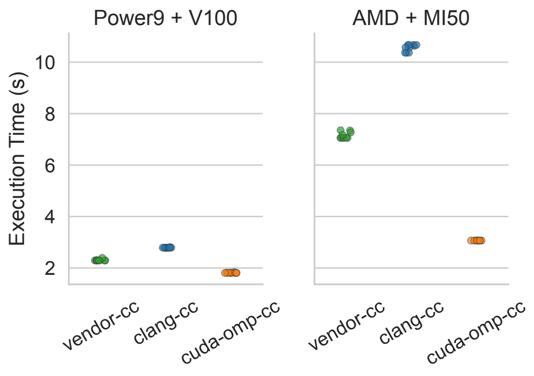
Triad. Triad is a benchmark stressing the bandwidth of the host-device subsystem communication layer. Through asynchronous CUDA APIs and CUDA streams the benchmark overlaps host-to-device transfers, computations, and device-to-host transfers.

Figure 8e illustrates the performance measurements of Triad. In the case of *Power9 + V100* all compilers perform similarly. The *cuda-omp-cc* compiled executable is the slowest version but overall performance is not impacted significantly. Specifically, the *cuda-omp-cc* executable is 4% slower than the *vendor-cc* one.

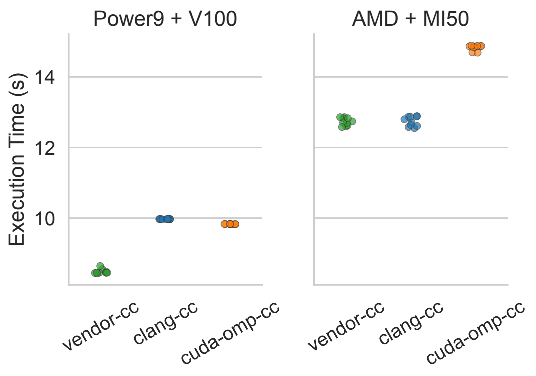
In the case of the *AMD + MI50* system, using *cuda-omp-cc* results in very poor performance. Currently, the LLVM/OpenMP offload plugin for AMD does not implement streams, or an alternative thereof. So, although we lower streams into the high-level stream abstraction used by the OpenMP offload runtime, the plugin does not make use of this information and serializes the transfers and computation. Moreover, the AMD plugin does not implement the pinned memory allocator interface as the NVIDIA plugin does. Therefore, transfers between the host and the device are significantly slower. These limitations of the current prototype will be resolved as the AMD plugin implementation becomes more robust and adds support for streams and pinned memory.



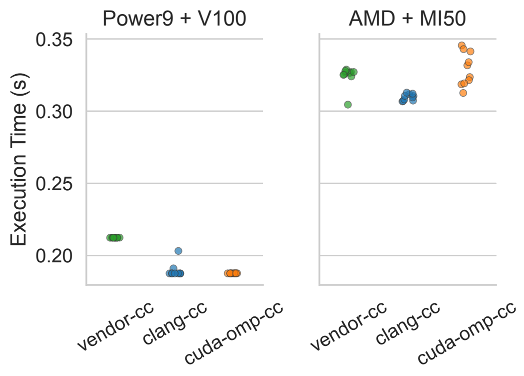
(a) Execution times of XSBench.



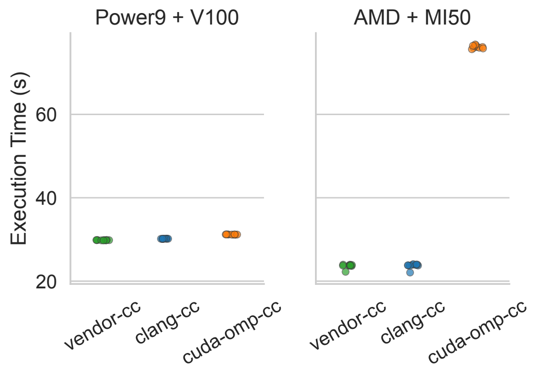
(b) Execution times of RSBench.



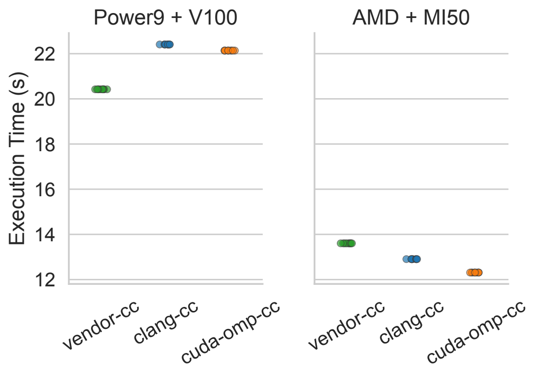
(c) Execution times of Lulesh.



(d) Execution times of SU3.



(e) Execution times of Triad.



(f) Execution times of MiniFE.

Figure 8: Evaluation results for the six benchmarks described in Table 2.

MiniFE. MiniFE is a proxy application for unstructured implicit finite element codes. It is similar to HPCCG but provides a much more complete vertical covering of the steps in this class of applications. Templates are used to implement different sparse matrix structures and different data types, e.g., single and double precision.

Figure 8f illustrates the results for MiniFE. The LLVM/Clang compiled executable on the NVIDIA machine shows an overhead of 10% in comparison to the vendor compiled version. In the *AMD + MI50* system the *cuda-omp-cc* presents a similarly modest speedup of 1.10× and 1.04× compared to *vendor-cc* and *clang-cc* respectively.

7 RELATED WORK

Performance portability solutions for GPGPU programming usually take the form of either programming models with their own compiler, or libraries that abstract over multiple programming models.

The first set includes OpenMP [22] since version 4, OpenCL [1], and SYCL [25] for compute, OpenGL [10] and Vulkan [11] for graphics and a variety of smaller competitors. These all provide at least *portability* across different hardware targets, with varying degrees of difficulty and performance tuning requirements, but given a program that has already been ported to CUDA none of them are an easy target. The HIP [3] model technically also counts as portable since it can target either AMD or NVIDIA hardware, and with some caveats SYCL with HIPSYCL [2], but it requires code to either be ported or kept compliant with the requirements of `hipify`.

The second set includes Kokkos [9], RAJA [15] and even the parallel algorithms in C++17 [19]. Because these must abstract over existing programming models, they tend to provide higher level interfaces, and incur more overhead and more porting effort coming from code that is already written in a SIMT style and often utilizes "low-level" CUDA builtins.

There have been other efforts to interpose between CUDA and the hardware as well. Some of the oldest include efforts to target CPUs with CUDA, after simulator support was removed from CUDA, in the form of Ocelot [5, 6], MCUDA [27], COX [13], CuPBoP [12], and a commercial offering PGI CUDA C/C++ for x86 [18]. All allow for (some) portability to the CPU, with some limitations in terms of performance. None can target other GPU platforms. Through the virtual GPU OpenMP target [24] our approach is also able to offload CUDA to most CPU architectures supported by LLVM/Clang, though not necessarily with the best performance.

A recent research project [14] ports parts of device side CUDA code to SPIR-V in order to execute it on RISC-V GPUs. However, this is only a partial solution for key builtins and, most importantly, it requires manual porting of host and registration code.

The RCUDA [8] project, the OpenMP Cluster programming model [31], and the remote offloading plugin for LLVM [23] allow a program on one node to transparently access resources, e.g. the NVIDIA GPUs, on another node. This allows for more flexible use of GPUs, but does not help with porting to other architectures.

The official `hipify` translator that's part of HIP could be thought of as a source-to-source translator from CUDA to HIP, though it leaves nearly all compute code alone. CU2CL [21] does a source-to-source rewrite from CUDA to OpenCL, but due to the significant differences in semantics, especially OpenCL's lack of a single-source option for host and kernel code intermixed, usually requires the code to be maintained in OpenCL after translation. Martini [20] provides configurable alternative to such source-to-source translation tools as shown by their `hipify` clone.

8 FUTURE WORK

There are four major areas for improvement towards *fully interoperable* and *performance portable* parallel programming languages. The first three can closely follow the approach taken in this work while the fourth requires innovative and new solutions.

(1) Provide wrappers for the HIP and SYCL specific APIs and builtins. While not conceptually different from the wrapped CUDA APIs and builtins, new wrappers are necessary to translate those languages to the OpenMP intermediate layer. Full interoperability with CUDA and OpenMP will follow.
(2) Provide (extended) plugins for Intel GPUs and other offload targets to allow all supported languages to offload to those. While there are open source Intel GPU plugins, they are not merged into community LLVM yet and maturity is unclear.
(3) Extend the wrappers around user facing CUDA APIs. This includes core language APIs, e.g., all functions with the `cuda` prefix, but not necessarily support libraries such as NVIDIA's CUB or Thrust. Inline assembly support might be required for certain applications but one could imagine portable function calls as alternatives for some use cases.
(4) Provide a high-level wrapper library to implement or replace the interfaces of common vendor support libraries, e.g. NVIDIA Thrust. There are various benefits of intercepting calls early. The interfaces of equivalent libraries by other vendors, e.g., ROCm Thrust, are designed to be similar but not necessarily identical drop-in replacements. To this end, we imagine calls to a wrapper library, like "OpenMP Thrust", which redirects to the appropriate vendor implementation. Alternatively, to avoid application changes, one could also imagine that the OpenMP wrapper directly exposes the same user facing functions, hence with the same name and in the same namespace, as existing libraries.

9 CONCLUSION

In this work we have shown that portability of, and interoperability between, offloading languages is possible using mostly existing compiler technology alone. By redesigning the compilation flow, deliberately choosing abstraction layers, and providing target independent implementations for language APIs and builtins, users can be freed from porting efforts and the burden to maintain multiple program versions. While initial results clearly show the potential, there is more work to do to stabilize the environment, integrate all parts into LLVM, and find suitable solutions for high-level libraries, e.g., NVIDIA's Thrust. However, we believe this work shows that *full interoperability* and *performance portability* of parallel programming languages does not require a more diverse set of new models. Instead, we need convergence the toolchains and compilation schemes that have evolved over time to unify the handling of existing offloading languages. Supporting different languages, syntaxes, and abstraction layers is only beneficial if we collapse the underlying ecosystem to benefit from advances at this level throughout the entire application world. Further, we believe unification will free effort that we can redirect towards performance optimizations, debugging solutions, and general tooling in a way that will make HPC better for any and all applications.

ACKNOWLEDGMENTS

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.

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 partially prepared by LLNL under Contract `DE-AC52-07NA27344 (LLNL-CONF-834862)` and was supported by the LLNL-LDRD Program under Project No. 20-ERD-018.

REFERENCES

[1] 2012. The OpenCL Specification. https://www.khronos.org/registry/cl/specs/opencl-1.2.pdf.

[2] Aksel Alpay. 2022. hipSYCL - a SYCL implementation for CPUs and GPUs. https://github.com/illuhad/hipSYCL.

[3] AMD. 2022. AMD HIP Programming Model. https://github.com/ROCm-Developer-Tools/HIP.

[4] Carleton DeTar, Steven Gottlieb, Ruizi Li, and Doug Toussaint. 2018. MILC Code Performance on High End CPU and GPU Supercomputer Clusters. In *EPJ Web of Conferences*. EDP Sciences.

[5] Gregory Frederick Diamos, A Kerr, and M Kesavan. 2009. Translating GPU binaries to tiered SIMD architectures with Ocelot. (2009).

[6] Gregory Frederick Diamos, Andrew Robert Kerr, Sudhakar Yalamanchili, and Nathan Clark. 2010. Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *International Conference on Parallel Architectures and Compilation Techniques*. ACM. https://doi.org/10.1145/1854273.1854318

[7] Johannes Doerfert, Joseph Huber, and Melanie Cornelius. 2021. Advancing OpenMP Offload Debugging Capabilities in LLVM. In *ICPP Workshops 2021: 50th International Conference on Parallel Processing, Virtual Event / Lemont (near Chicago), IL, USA, August 9-12, 2021*, Federico Silla and Osni Marques (Eds.). ACM, 20:1–20:8. https://doi.org/10.1145/3458744.3473358

[8] José Duato, Antonio J Pena, Federico Silla, Rafael Mayo, and Enrique S Quintana-Ortí. 2010. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In *2010 International Conference on High Performance Computing & Simulation*. IEEE, 224–231.

[9] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. 2014. Kokkos: Enabling Manycore Performance Portability Through Polymorphic Memory Access Patterns. *J. Parallel and Distrib. Comput.* 74, 12 (2014), 3202 – 3216. https://doi.org/10.1016/j.jpdc.2014.07.003 Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[10] Khronos Group. 2022. The OpenGL Specification. https://www.opengl.org/.

[11] Khronos Group. 2022. The Vulkan Specification. https://www.vulkan.org/.

[12] Ruobing Han, Jun Chen, Bhanu Garg, Jeffrey Young, Jaewoong Sim, and Hyesoon Kim. 2022. CuPBoP: CUDA for Parallelized and Broad-range Processors. *CoRR* abs/2206.07896 (2022). arXiv:2206.07896 https://doi.org/10.48550/arXiv.2206.07896

[13] Ruobing Han, Jaewon Lee, Jaewoong Sim, and Hyesoon Kim. 2021. COX: CUDA on X86 by Exposing Warp-Level Functions to CPUs. *CoRR* abs/2112.10034 (2021). arXiv:2112.10034 https://arxiv.org/abs/2112.10034

[14] Ruobing Han, Blaise Tine, Jaewon Lee, Jaewoong Sim, and Hyesoon Kim. 2021. Supporting CUDA for an extended RISC-V GPU architecture. *CoRR* abs/2109.00673 (2021). arXiv:2109.00673 https://arxiv.org/abs/2109.00673

[15] RD Hornung and JA Keasler. 2014. *The RAJA Portability Layer: Overview and Status*. Technical Report. Lawrence Livermore National Laboratory (LLNL), Livermore, CA.

[16] R. D. Hornung, J. A. Keasler, and M. B. Gokhale. 2011. Hydrodynamics challenge problem. (6 2011). https://doi.org/10.2172/1117905

[17] 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 *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2022, Seoul, Korea, Republic of, April 2-6, 2022*, Jae W. Lee, Sebastian Hack, and Tatiana Shpeisman (Eds.). IEEE, 41–52. https://doi.org/10.1109/CGO53902.2022.9741290

[18] Portland Group International. 2022. PGI CUDA C/C++ for x86. https://developer.nvidia.com/pgi-cuda-cc-x86.

[19] ISO/IEC. 2017. ISO International Standard ISO/IEC 14882:2017 - Programming Languages – C++. https://www.iso.org/standard/68564.html.

[20] Alister Johnson, Camille Coti, Allen D. Malony, and Johannes Doerfert. 2022. MARTINI: The Little Match and Replace Tool for Automatic Application Rewriting with Code Examples. In *Euro-Par 2022: Parallel Processing - 28th International Conference on Parallel and Distributed Computing, Glasgow, UK, August 22-26, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13440)*, José Cano and Phil Trinder (Eds.). Springer, 19–34. https://doi.org/10.1007/978-3-031-12597-3_2

[21] Gabriel Martinez, Mark Gardner, and Wu-chun Feng. 2011. CU2CL: A CUDA-to-OpenCL translator for multi-and many-core architectures. In *2011 IEEE 17th International Conference on Parallel and Distributed Systems*. IEEE, 300–307.

[22] OpenMP ARB. 2008. OpenMP Application Programming Interface Version 5.2. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf

[23] Atmn Patel and Johannes Doerfert. 2022. Remote OpenMP offloading. In *PPoPP '22: 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Seoul, Republic of Korea, April 2 - 6, 2022*, Jaejin Lee, Kunal Agrawal, and Michael F. Spear (Eds.). ACM, 441–442. https://doi.org/10.1145/3503221.3508416

[24] Atmn Patel, Shilei Tian, Johannes Doerfert, and Barbara M. Chapman. 2021. A Virtual GPU as Developer-Friendly OpenMP Offload Target. In *ICPP Workshops 2021: 50th International Conference on Parallel Processing, Virtual Event / Lemont (near Chicago), IL, USA, August 9-12, 2021*, Federico Silla and Osni Marques (Eds.). ACM, 24:1–24:7. https://doi.org/10.1145/3458744.3473356

[25] Ruyman Reyes and Victor Lomüller. 2016. SYCL: Single-source C++ accelerator programming. In *Parallel Computing: On the Road to Exascale*. IOS Press, 673–682.

[26] Paul K. Romano, Nicholas E. Horelik, Bryan R. Herman, Adam G. Nelson, Benoit Forget, and Kord Smith. 2015. OpenMC: A State-of-the-Art Monte Carlo Code for Research and Development. *Annals of Nuclear Energy* (2015). https://doi.org/10.1016/j.anucene.2014.07.048 Joint International Conference on Supercomputing in Nuclear Applications and Monte Carlo 2013, SNA + MC 2013. Pluri- and Trans-disciplinarity, Towards New Modeling and Numerical Simulation Paradigms.

[27] John A. Stratton, Sam S. Stone, and Wen-mei W. Hwu. 2008. MCUDA: An Efficient Implementation of CUDA Kernels for Multi-core CPUs. In *Languages and Compilers for Parallel Computing, 21th International Workshop, LCPC 2008, Edmonton, Canada, July 31 - August 2, 2008, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 5335)*, José Nelson Amaral (Ed.). Springer, 16–30. https://doi.org/10.1007/978-3-540-89740-8_2

[28] Shilei Tian, Jon Chesterfield, Johannes Doerfert, and Barbara M. Chapman. 2021. Experience Report: Writing a Portable GPU Runtime with OpenMP 5.1. In *OpenMP: Enabling Massive Node-Level Parallelism - 17th International Workshop on OpenMP, IWOMP 2021, Bristol, UK, September 14-16, 2021, Proceedings (Lecture Notes in Computer Science)*, Simon McIntosh-Smith, Bronis R. de Supinski, and Jannis Klinkenberg (Eds.). Springer. https://doi.org/10.1007/978-3-030-85262-7_11

[29] John R. Tramm, Andrew R. Siegel, Benoit Forget, and Colin Josey. 2014. Performance Analysis of a Reduced Data Movement Algorithm for Neutron Cross Section Data in Monte Carlo Simulations. In *EASC 2014 - Solving Software Challenges for Exascale*. Stockholm. https://doi.org/10.1007/978-3-319-15976-8_3

[30] John R. Tramm, Andrew R. Siegel, Tanzima Islam, and Martin Schulz. 2014. XSBench - The Development and Verification of A Performance Abstraction for Monte Carlo Reactor Analysis. *The Role of Reactor Physics toward a Sustainable Future (PHYSOR)* (2014).

[31] Hervé Yviquel, Marcio Pereira, Emílio Francesquini, Guilherme Valarini, Gustavo Leite, Pedro Rosso, Rodrigo Ceccato, Carla Cusihualpa, Vitoria Dias, Sandro Rigo, et al. 2022. The OpenMP Cluster Programming Model. In *The Second Workshop on LLVM in Parallel Processing (LLPP)*.