Experience Report: Writing A Portable GPU Runtime with OpenMP 5.1

Shilei Tian[1][0000−0001−6468−6839], Jon Chesterfield[2][0000−0002−8546−2014],
Johannes Doerfert[3][0000−0001−7870−8963], and Barbara
Chapman[1][0000−0001−8449−8579]

[1] Department of Computer Science, Stony Brook University, USA
{shilei.tian, barbara.chapman}@stonybrook.edu
[2] Advanced Micro Devices, UK
jchester@amd.com
[3] Mathematics and Computer Science, Argonne National Laboratory, USA
jdoerfert@anl.gov

Abstract. GPU runtimes are historically implemented in CUDA or other vendor specific languages dedicated to GPU programming. In this work we show that OPENMP 5.1, with minor compiler extensions, is capable of replacing existing solutions without a performance penalty. The result is a performant and portable GPU runtime that can be compiled with LLVM/Clang to Nvidia and AMD GPUs without the need for CUDA or HIP during its development and compilation.

While we tried to be OPENMP compliant, we identified the need for compiler extensions to achieve the CUDA performance with our OPENMP runtime. We hope that future versions of OPENMP adopt our extensions to make device programming in OPENMP also portable across compilers, not only across execution platforms.

The library we ported to OPENMP is the OPENMP device runtime that provides OPENMP functionality on the GPU. This work opens the door for shipping OPENMP offloading with a Linux distribution's LLVM package as the package manager would not need a vendor SDK to build the compiler and runtimes. Furthermore, our OPENMP device runtime can support a new GPU target through the use of a few compiler intrinsics rather than requiring a reimplementation of the entire runtime.

Keywords: OpenMP · LLVM · Portability · Target offloading · Runtimes · Accelerator

1 Introduction

In this paper, we describe how we ported the LLVM OPENMP device runtime library to OPENMP 5.1 using only minor extensions not available in the standard. The OPENMP device runtime provides the OPENMP functionalities to the user and implementation code on the device, which in this context means on the GPU. As an example, it provides the OPENMP API routines as well as routines utilized by the compiler e.g., for worksharing loops.

Our work replaced the original LLVM OpenMP device runtime implemented in CUDA to allow for code reusibility between different targets, e.g. AMD and Nvidia. It further lowers the bar to entry for future targets that only need to provide a few target specific intrinsics and minimal glue code.

The OpenMP device runtime library can now be shipped with pre-build LLVM packages as they only need LLVM/Clang to build it; neither a target device nor vendor SDKs are required, which lowers the barrier to entry for OpenMP offloading. This work is a proof of concept for writing device runtime libraries in OpenMP, with identical functionality and performance to that available from CUDA or HIP compiled with the same LLVM version.

The remainder of the paper is organized as follows. We discuss background and motivation in Section 2. Section 3 presents our approach, which is followed by an evaluation in Section 4. Finally, we conclude the paper in Section 5.

2   Background

When compiling from one language to another, there are usually constructs that are straightforward in the former and complicated or verbose in the latter. For example, a single OpenMP construct `#pragma omp parallel for` is lowered into a non-trivial amount of newly introduced code in the application, including calls into a runtime that provides certain functionality, like dividing loop iterations. In this work, the input is OpenMP target offloading code, that is the OpenMP target directive and the associated code, and the output is ultimately Nvidia's PTX or AMD's GCN assembler.

2.1   Device Runtime Library

The LLVM OpenMP device runtime library contains the various functions the compiler needs to implement OpenMP semantics when the target is an Nvidia or AMD GPU. It is basically `libomp` for the GPU. The original implementation in LLVM was in CUDA [8], compiled with Nvidia's NVCC to PTX assembler which was linked with the application code to yield a complete program. We later adapted that source to compile alternatively as HIP, which is close enough to CUDA syntax for the differences to be worked around with macros. Prior to this work the device runtime was hence comprised of sources in a common and target dependent part. In order to let the target dependent compiler recognize the code, target dependent keywords (such as `__device__` and `__shared__` in CUDA) are replaced with macros (`DEVICE` and `SHARED`), and the header where these macros are defined will be included accordingly depending on the target. The basic idea is visualized in in Listing 1.

```
// Common part
DEVICE void *__kmpc_alloc_shared(uint64_t bytes);
SHARED int shared_var;
// CUDA header
#define DEVICE __device__
```

```
#define SHARED __shared__
// AMDGCN header
#define DEVICE __attribute__((device))
#define SHARED __attribute__((shared))
```

Listing 1. Macros in current device runtime.

This strategy works. For Nvidia offloading the source is compiled as CUDA, for AMDGPU offloading it is compiled as HIP. Both produce LLVM bitcode but with different final targets, Nvidia's PTX and AMD's GCN respectively. However, if a programming model does not adequately resemble CUDA, such as OpenCL or Intel's DPC++ [3], the approach will become less straight forward.

What's more, this setup requires vendor SDKs (such as CUDA Toolkit or ROCm Developer Tools) to compile the device runtime, which creates a barrier for the package managers of Linux distributions. In practice that means the LLVM OpenMP installed from Linux distributions does not support offloading out of the box because the package would require a dependence on the CUDA or ROCm package, among other things.

2.2 Compilation Flow of OpenMP Target Offloading in LLVM/Clang

The compilation of an OpenMP program with target offloading directives contains the following two passes (as shown in Fig. 1):

Host Code Compilation. This pass includes the regular compilation of code for the host and OpenMP offloading code recognition as preparation for the second pass. Offloading regions are replaced by calls to the corresponding host runtime library functions (e.g. `__tgt_target` for the directive `target` in LLVM OpenMP) with suitable arguments, such as the kernel function identifier, base pointers of each captured variables and the number of kernel function arguments. In addition, a fallback host version of the kernel function will be emitted in case target offloading fails at runtime.

Device Code Compilation. This pass utilizes the recognized OpenMP target offload regions, as well as related functions and captured variables, and then emits target dependent device code. This includes one entry kernel function per target region, global variables (potentially in different address spaces), and device functions, as well as some target dependent metadata. As part of this compilation the OpenMP device runtime library is linked into the user code as an LLVM bitcode library (`dev.rtl.bc` in the Fig. 1).

In addition to the `target` construct (as well as its combined variants), OpenMP provides the `declare target` directive which specifies that variables and functions are mapped onto a target device, and should hence be usable in device code. The `declare variant` directive can be used to specify a context, e.g., the compilation for a specific target, in which a specialized function variant should replace the base version.
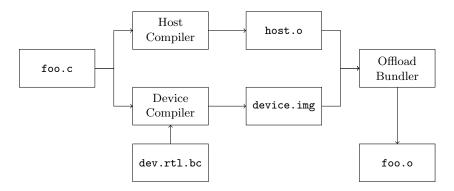
Fig. 1. Compilation flow of an OpenMP program with target offloading.

2.3   Motivation

While the OPENMP device runtime library can be implemented in any language it should be linked into the application in LLVM bitcode format for performance reasons. This setup, shown in Fig. 1, allows to optimize the runtime together with the application, effectively specializing a generic runtime as needed.

Given that the base language is irrelevant as long as we can compile to LLVM bitcode, OPENMP comes to mind as a portable and performant way to write code for different accelerators. As almost the entire device library can be interpreted as C++ code, rather than a CUDA or HIP code base, the compilation as OPENMP is feasible, in particular because LLVM/Clang is a working C++ and OPENMP compiler already.

Since OPENMP 5.1 all conceptually necessary building blocks are present in the language specification:

– The `declare target` directive can be used to compile for a device, hence to generate LLVM bitcode that is targeting Nvidia's PTX or AMD's GCN. As we do not need a host version at all, we can even use the LLVM/Clang flag `-fopenmp-is-device` to invoke only the device compilation pass described in Section 2.2.
– The `declare variant` directive can be used if a target requires a function implementation or global variable definition different from the default.
– The `allocate` directive provides access to the different kinds of memory on the GPU.

For an additional target architecture, the work done in the compiler backend to emit code for that architecture will allow to retarget an OPENMP implemented device runtime almost for free. The incremental development cost is reduced from (re)implementing the device runtime in a language that can be compiled to the new architecture to providing a few declare variant specialisations.

Finally, if the port uses compiler intrinsics instead of CUDA or HIP functions for the small target dependent part, it can be compiled without a vendor specific SDK present. This unblocks shipping offloading as part of Linux distributions.

3   Implementation

In this section, we describe the new LLVM OpenMP device runtime implemented with OpenMP 5.1. First, we talk about the common part, and then discuss how target dependent parts are implemented and why extensions were necessary. Only AMD and Nvidia platforms are discussed as other GPU architectures cannot be targeted by the community LLVM version at this time.

3.1   Common Part

Device Code
Using the `declare target` directive around all source files causes all functions and data to be emitted for the target device. Macros to indicate that functions or globals are for the device, as shown in Listing 1, are not needed.

Global Shared Variables
The implementation of the device runtime maps an OpenMP team to a thread block[1] on the target device. Therefore, a shared variable visible to all threads in the same thread block is equivalent to a variable that can be accessed within the same OpenMP team. The `allocate` directive specifies how to allocate variables in different memory spaces. Uses with an `allocator(omp_cgroup_mem_alloc)`[2] we can place global variables in local shared memory, the equivalent of the CUDA `__shared__` shown in Listing 1.

In contrast to shared CUDA or HIP variables, C++ specifies that global variables are default initialized. While we can technically do this for global shared variables defined with OpenMP, it is not supported by LLVM/Clang at this time. Furthermore, the performance is likely to suffer as the device runtime is designed to initialize these variables explicitly on demand. To this end, we extended LLVM/Clang with a variable attribute for this work: `loader_uninitialized`[1]. The effect is that annotated variables will not have a default initialized value but instead be uninitialized like the CUDA or HIP shared variables are as well.

Listing 2 shows device code and global shared variable declaration as it is used in our OpenMP device runtime.

```
#pragma omp begin declare target

// Function declaration
extern __kmpc_impl_threadfence();
// Function definition
void __kmpc_flush(kmp_Ident *loc) {
  __kmpc_impl_threadfence();
}
// Global variable
int global_var;
```

---

[1] We are using CUDA terminology here. For AMD platforms it is *wavefront*.

[2] The implementation currently uses `allocator(omp_pteam_mem_alloc)` which is equivalent given the current mapping of parallelism.

```
// Shared variable
int shared_var ;
#pragma omp allocate ( shared_var ).          \
            allocator ( omp_pteam_mem_alloc )
// Shared variable declaration
extern int other_shared_var ;
#pragma omp allocate ( other_shared_var )     \
            allocator ( omp_pteam_mem_alloc )

#pragma omp end declare target
```

<div align="center">

Listing 2. An example of new device runtime code.

</div>

Atomic Operations

The device runtime uses five atomic operations, `add`, `inc`, `max`, `exchange`, and `cas`, implemented in target dependent parts with LLVM/Clang builtin functions.

OPENMP 5.1 [4] introduces the `compare` clause, which supports conditional update statements. When combined with the `capture` clause, all of these atomic operations except `inc` can be implemented via OPENMP, as shown in Listing 3. We implemented the support of the `compare` clause and its combination with the `capture` clause for LLVM/Clang but the it has not been merged into the community version yet. With the updated requirements for flush[3], which we also implemented for this work, our OPENMP versions of atomic operations can generate LLVM-IR that is identical to the original target dependent implementation via compiler intrinsics.

```
uint32_t atomic_add ( uint32_t *X, uint32_t E ) {
  uint32_t V;
#pragma omp atomic capture seq_cst
  { V = *X; *X += E; }
  return V;
}
uint32_t atomic_max ( uint32_t *X, uint32_t E ) {
  uint32_t V;
#pragma omp atomic compare capture seq_cst
  { V = *X; if (*X < E) { *X = E; } }
  return V;
}
uint32_t atomic_exchange ( uint32_t *X, uint32_t E ) {
  uint32_t V;
#pragma omp atomic capture seq_cst
  { V = *X; *X = E; }
  return V;
}
uint32_t atomic_cas ( uint32_t *X, uint32_t E, uint32_t D ) {
```

---

[3] OPENMP 5.1 removes the requirement for a flush operation at the entry and exit of an atomic operation if `write`, `update`, or `capture` is specified and the memory ordering is `seq_cst`

```
      uint32_t V;
#pragma omp atomic compare capture seq_cst
  { V = *X; if (*X == E) { *X = D; } }
  return V;
}
```

Listing 3. Atomic operations implemented in OpenMP 5.1.

The missing atomic operation is `inc`. According to the CUDA specification [2], `inc` implements:

```
{ v = x; x = x >= e ? 0 : x + 1; }
```

and returns `v`. This atomic operation can not be represented in a form that OpenMP 5.1 supports because OpenMP 5.1 requires that the order operation be either `<` or `>`, and the alternative statement of the conditional expression statement must be `x` itself. Therefore, we still keep it in the target dependent part implemented with LLVM intrinsics as shown in Listing 4.

3.2   Target Specific Part

Target dependent global functions and variables are currently declared in a header and implemented in target dependent source files which are only compiled for the specific target, either as CUDA or HIP. A drawback of this method is that the creation of a device runtime for a new target might require us to remove a function from the common part and insert it into the target specific part if the existing (common) implementation is not suited for the new device.

Since OpenMP 5.0, the `declare variant` directive declares a specialized variant of a base function and specifies the context in which that specialized variant is used. It supports various context selector with the `match` clause, one of which is `device` selector. For example, with `match(device={arch(arch_name)})`, the code wrapped in a `begin/end declare variant` region will be only generated if the target architecture *matches* the `arch_name`.

Listing 4 shows how the atomic `inc` function is implemented with target dependent compiler intrinsics selected via the `begin/end declare variant` directive for both Nvidia and AMD GPU targets.

Note that we use the `match_any` extension for Nvidia platforms as we support two distinct architectures, `nvptx` and `nvptx64`, but we do not want to distinguish between them in the device runtime. While this can be handled by duplicating the code, our new context selector changes the semantic of the matching to produce a match if *any* architecture in `arch(nvptx, nvptx64)` is targeted. By default a match would require all architectures to be targeted. In addition to `match_any` we extended LLVM/Clang with other useful context selectors, e.g, `match_none` and `allow_templates`[4].

```
#pragma omp declare target
// Fallback version, which raises a compilation error
```

---

[4] See: `https://clang.llvm.org/docs/AttributeReference.html#pragma-omp-declare-variant`

```
uint32_t atomic_inc(uint32_t *X, uint32_t E) {
  error("target dependent implementation missing");
}
// AMDGCN implementation
#pragma omp begin declare variant                        \
           match(device={arch(amdgcn)})
uint32_t atomic_inc(uint32_t *X, uint32_t E) {
  return __builtin_amdgcn_atomic_inc32(X, E,
                                       __ATOMIC_SEQ_CST, "");
}
#pragma omp end declare variant
// NVPTX implementation
#pragma omp begin declare variant                        \
           match(device={arch(nvptx,nvptx64)},           \
                 implementation={extension(match_any)})
uint32_t atomic_inc(uint32_t *X, uint32_t E) {
  return __nvvm_atom_inc_gen_ui(X, E);
}
#pragma omp end declare variant
#pragma omp end declare target
```

Listing 4. Atomic `inc` implementation with the `match_any` clause.

Other target dependent functions are required to handle synchronization, thread hierarchy, etc. These are implemented via compiler intrinsics, function calls to the corresponding native runtime library, or inline assembly.

4 Evaluation

In this section, we evaluated our proposed method in three ways: code comparison, functional testing, and performance evaluation.

4.1 Code Comparison

The previous implementation compiled CUDA to LLVM-IR, and HIP to LLVM-IR, while our proposed method compiles OpenMP to LLVM-IR for both platforms. The accuracy of the port to OpenMP was assessed by comparing the text form of the library before and after changing over to OpenMP. If the text forms were identical, we would be certain the language change made no difference. This was not quite the case. The differences were in semantically unimportant metadata, symbol name mangling for variant functions, and the order of inlining as preferred by the language front end which had minor reordering effects on PTX and GCN generation.

4.2 Functional Testing

There are a number of OpenMP test suites and applications in use for checking the behaviour of the compiler, including SOLLVE V&V [7], and Ovo [5]. All

ran identically with the new OpenMP runtime as they had using the previous device runtime.

4.3 Performance Evaluation

Systems Configuration We evaluate the performance of our method experimentally on the Summit supercomputer. Each Summit node contains two IBM POWER9 processors and six Nvidia Volta V100 GPUs. CUDA 10.1.243 was used, which is the version loaded by default.

Benchmarks The SPEC ACCEL benchmark suite V1.3 was used to evaluate the new device runtime. Because support for Fortran is still in progress, we chose those benchmarks written in C. There are 15 OpenMP enabled benchmarks in SPEC ACCEL. Seven of them are in C, namely `503.postencil`, `504.polbm`, `514.pomriq`, `552.pep`, `554.pcg`, `557.pcsp`, and `570.pbt`. `557.pcsp` can not be compiled, therefore we only ran the other six benchmarks. We also chose a C++ proxy application, miniQMC [6].

`O2` was used when compiling the benchmarks and application. Each test case was executed five times, and the execution time was averaged. miniQMC was measured through the `miniqmc_sync_move` benchmark executed as follows: `miniqmc_sync_move -g "2 2 1"`.

Results Fig. 2 compares the execution time when the original device runtime is used with the execution time obtained using our proposed new device runtime. We can see that the execution times are almost identical, and for those cases where they are not same, the variance is less than 1% and assumed to be noise.
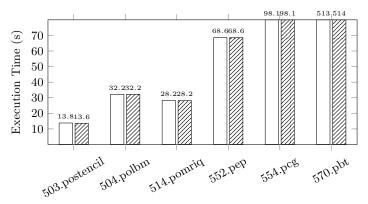

Fig. 2. Comparison between execution time of original device runtime (▢) and that of our proposed new device runtime (▨) on Nvidia platform.

The proxy application benchmark `miniqmc_sync_move` contains two target regions, `evaluate_vgh` and `evaluateDetRatios`. They are executed multiple times. Table 1 shows the profiling results (execution time) of each target region

from Nvidia's profiler `nvprof`. There is no performance difference between the two versions.

Target Region	Version	Time (ms)	# Calls	Avg (μs)	Min (μs)	Max (μs)
evaluate_vgh	Original	1374.72	64512	21.309	19.744	32.384
	New	1376.59		21.338	19.776	33.760
evaluateDetRatios	Orignal	573.46	18202	31.505	25.247	44.480
	New	573.93		31.531	24.544	47.103

Table 1. Comparison of execution time of the two target regions in `miniqmc_sync_move` on Nvidia platform.

All the results above demonstrate that our proposed portable OPENMP device runtime can provide the same performance as the current CUDA-like version on the Nvidia platform. Based on the code comparison, functional testing and some AMD internal performance testing results, the portable runtime is believed to show no performance change from its HIP predecessor either.

5   Conclusions and Future Work

OPENMP works well as a language to implementing GPU-only code libraries. The direct support for memory allocators and the precise dispatch through `declare variant` are clear advantages over C++. While minimal compiler modifications were required to match the CUDA and HIP semantics to the fullest, we expect those to be incorporated into the OPENMP standard over time.

Using OPENMP is especially suitable as the vehicle for implementing an OPENMP runtime library since the main prerequisite is an OPENMP compiler which needs to be implemented all targets in any case. Since the library ships with the LLVM repository, it can be built by any distribution which has built Clang. Vendor SDKs or compilers are no longer required.

Since the host and device runtime libraries can build as part of LLVM, we will coordinate with Linux distribution developers to ensure that people who install the distribution LLVM package onto a system that has a target device and driver available will be able to get this working "out of the box".

Acknowledgement

References

1. Attributes in clang, `https://clang.llvm.org/docs/AttributeReference.html#loader-uninitialized`
2. Cuda toolkit documentation v11.3.0, `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#ato`
3. Intel® oneapi dpc++/c++ compiler, `https://software.intel.com/content/www/us/en/develop/tools/oneapi/c`
4. Openmp application programming interface version 5.1, `https://www.openmp.org/spec-html/5.1/openmp.html`
5. Ovo: Openmp vs offload, `https://github.com/TApplencourt/OvO`
6. Qmcpack/miniqmc, `https://github.com/QMCPACK/miniqmc`
7. Sollve/sollve_vv, `https://github.com/SOLLVE/sollve_vv`
8. Jacob, A.C., Eichenberger, A.E., Sung, H., Antao, S.F., Bercea, G.T., Bertolli, C., Bataev, A., Jin, T., Chen, T., Sura, Z., Rokos, G., O'Brien, K.: Efficient fork-join on gpus through warp specialization. In: 2017 IEEE 24th International Conference on High Performance Computing (HiPC). pp. 358–367 (2017). https://doi.org/10.1109/HiPC.2017.00048