A Virtual GPU as Developer-Friendly OpenMP Offload Target

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

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

Johannes Doerfert
jdoerfert@anl.gov
Argonne National Laboratory
Lemont, Illinois, United States

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

ABSTRACT

While parallel programming is hard, programming accelerators has always been even more complicated. One fundamental reason is the lack of mature tooling that can be used to inspect a program that executes on two different architectures. As GPU software stacks of different vendors provide vastly different experience for developers, it is clear that the gold standard for debugging is still host (CPU) execution with its myriad of mature tooling options.

In this work we present a *virtual GPU* (VGPU) OpenMP offloading target that allows to emulate a GPU execution environment on the host. In contrast to classical "host offloading", the VGPU target reuses the same execution model, compilation paths, and runtimes as a physical GPU. While this execution mode is not able to perform as good as host-specific compilation, runtimes, and execution, it provides the developor with a more accurate stand-in for GPU offloading that is still amendable to existing host tooling.

KEYWORDS

LLVM, OpenMP, accelerator offloading, GPU, debugging

ACM Reference Format:
Atmn Patel, Shilei Tian, Johannes Doerfert, and Barbara Chapman. 2021. A Virtual GPU as Developer-Friendly OpenMP Offload Target. In *LLPP '21: The First Workshop on LLVM in Parallel Processing (LLPP), August 9th, 2021, Chicago (Argonne National Lab), Illinois, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3458744.3473356

1 INTRODUCTION

Given the omnipresence of accelerators in modern compute systems it is unreasonable to develop high-performance applications without accelerator support. In addition to classical parallel programming, which usually utilizes homogeneous processing units in a shared memory system, accelerator programming requires the developer to manage memory and execution across heterogeneous devices that operate through different program execution models. While the former is hard by itself, the latter introduces a myriad of potential errors that complicate programming, and debugging, even further. On top of the increased complexity comes an often immature software stack that makes development and debugging much harder than it would be on the host.

In this work we introduce a new way to develop and debug OpenMP offloading code that is ultimately intended for a GPU device through the mature software stack available on the host. By combining the GPU compilation paths, runtime libraries, and execution model with execution on the host CPU we provide a unique development environment: closely resembling the execution on the GPU but almost as manageable as any other CPU program.

Our *virtual GPU* (VGPU) offloading target is integrated into the LLVM/OpenMP implementation and the LLVM/Clang compiler. While the usage is no different from any other offloading target, e.g., the host or a physical GPU, the effect is a unique combination of GPU offloading and host execution. The user compiles the OpenMP offloading program specifying the virtual GPU as a target as they would a real GPU. At runtime, the VGPU executes (almost) the same code as a native GPU execution would, except that it is a host binary executing on the host (CPU). As such, the user is (conceptually) able to utilize any existing host (CPU) tooling, like debuggers, profilers, and sanitizers, regardless of the maturity of the software stack for the ultimate physical GPU target.

The remaining of the paper is organized as follows. We briefly introduce LLVM/OpenMP target offloading in Section 2. Then, Section 3 and Section 4 show the design and implementation details: how the virtual GPU device environment is implemented such that it looks like a real device from the perspective of the device runtime, and the support from compiler and device runtime. Next, we evaluate our method in Section 5, and talk about related works in Section 6. Finally, the paper is concluded in Section 7.

2 BACKGROUND

In this section, we will briefly introduce OpenMP target offloading support in LLVM. We will first talk about the compilation of an OpenMP target offloading program, and then discuss the runtime support.

2.1 OpenMP Target Offloading Compilation

The compilation of an OpenMP program with target offloading directives contains the following two passes (as shown in Figure 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`) 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, which will be introduced next, is linked into the user code as an LLVM bitcode library (`vgpu.rtl.bc` in the Figure 1).

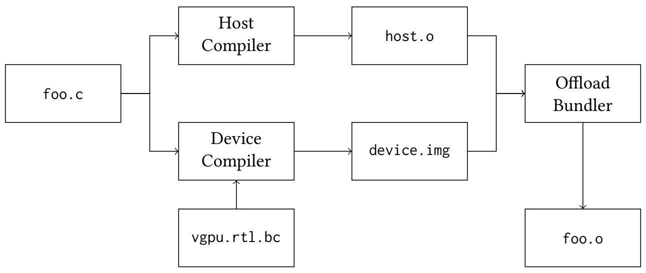

Figure 1: Compilation flow of an OpenMP program with target offloading.

2.2 OpenMP Target Offloading Runtimes

In order to support OpenMP target offloading, at runtime, the host code is executed, which dynamically links LLVM's `libomptarget` library, which dynamically loads the available target offloading plugins, and executes the device code using the device plugin, as demonstrated in Figure 2.

`libomptarget` implements OpenMP 5.X user-level runtime library functions as well as the compiler-level runtime library routines that Clang emits as part of its code generation. Clang's code generation emits device-agnostic calls to compiler-level routines which are then implemented by `libomptarget` via the target-specific plugins.

There are a few levels of interfaces that are used in LLVM's implementation of OpenMP target offloading: the interface between the compiled fat binary and `libomptarget`, the interface between `libomptarget` and its target device plugins, and the interface between the binary and the device runtime library. The external interface of `libomptarget` abstracts the internals by providing the definitions for the type of data reference in a target region, requires flags, type of allocation, the data structures used to hold the data and the target region, the data structures used to manage asynchronous calls, and the functions used to execute various kinds of target regions, as well as the most device-agnostic of the OpenMP user-level functions such as `omp_get_num_devices`, `omp_get_initial_device`, and `omp_target_alloc`.

The interface between `libomptarget` and its device plugins provide encapsulation of the device-specific implementations of routines such as memory copies to and from the device (synchronous and asynchronous), executing target regions on the device, explicit synchronization for the asynchronous executions, and determining the number of available devices. `libomptarget` supports offloading onto the following device architectures via the plugins in `openmp/libomptarget/plugins/`:

- AArch64
- AMDGCN
- CUDA
- PPC64[LE]
- Remote
- VE
- x86_64

Lastly, Clang's code generation will generate calls to the device runtime library for device-specific routines. The device runtime libraries for AMD and Nvidia GPUs are implemented in `openmp/libomptarget/DeviceRTL`, providing the device specific implementations of the interface defined in `openmp/libomptarget/DeviceRTL/Interface.h` which contains OpenMP user-level routines such as `omp_get_thread_limit` as well as OpenMP runtime-level routines such as `__kmpc_barrier` that clang emits as part of its codegeneration of OpenMP constructs.

3 VIRTUAL GPU DEVICE ENVIRONMENT

The VGPU plugin implements a single[1] VGPU through a `VGPUTy` object that contains a `std::thread` array. Conceptually, and similar to a real GPU, the VGPU consists of a number of Cooperative Thread Arrays (CTAs), each of which contains a number of warps (aka. wavefronts), each of which contains a number of threads. This organization ensures that we are able to provide a virtual device that "natively" supports all operations that a real GPU can support in a way that does not require (conceptual) changes to the device runtime that is linked with the application and executed on the device. The virtual GPU architecture is visualized with accompanying environment types in Figure 3.

The total number of threads (N) of the VGPU defaults to the number of processing elements of the host. The number of threads in a warp, and the number of warps in a CTA are by default chosen to get balanced degrees of parallelism on every level (intra warp, inter warp/intra CTA, inter CTA). That means, each level is given $\approx (N)^{1/3}$ threads. While we belive this is a sensible default, each value can be individually configured via the environment variables `VGPU_NUM_THREADS`, `VGPU_NUM_THREADS_PER_WARP`, and `VGPU_NUM_WARPS_PER_CTA`.

3.1 Initialization

When the VGPU plugin is loaded at runtime, a global `VGPUTy` object is constructed. Upon construction, this object will initialize the thread array used for execution, the execution queue used to manage asynchronous execution, as well as a number of environment types used to handle these threads at various levels of abstraction: at the GPU level, CTA level, and warp level.

[1]There is no conceptual restriction to a single VGPUs, the plugin could similarly expose a dynamic number of VGPUs that can look alike or be configured differently.

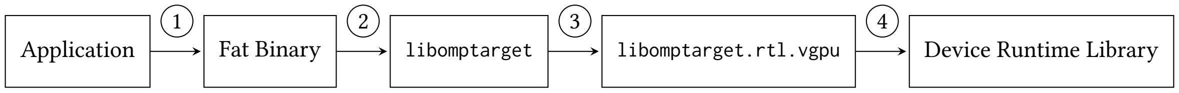

Figure 2: (1) Clang is used to compile the OpenMP target offloading application into a fat binary consisting of the host binary as well as the target binary, as shown in Figure 1. (2) This binary dynamically links in `libomptarget` and (3) `dlopen`'s `libomptarget.rtl.vgpu`. Within the target image, (4) the device runtime library is used to provide the implementation of device-specific routines such as atomic increments, memory fences, and thread synchronization.

In order to manage warp-level operations, `WarpEnvironmentTy` objects are constructed and stored in a global vector, and likewise for `CTAEnvironmentTy` objects. These objects are accessed via pointers in the thread-local `ThreadEnvironmentTy` object that is created for each thread in the thread-array. This thread-specific object is accessed by the device runtime library via the exposed accessor function `getThreadEnvironment()`. Its role is to emulate common native GPU functionalities, e.g., the hardware registers that contain the thread id or warp size. All threads are initially set to wait upon the `WorkAvailable` condition variable and are awakened as noted in the next subsection.

3.2 Scheduling

When `libomptarget` is calling for the (a)synchronous execution of a target region, the VGPU plugin will insert the associated code (aka. kernel) into a queue, analogous to a CUDA stream. More specifically, `libomptarget` will provide the plugin with a `__tgt_async_info` object that contains a VGPU execution queue/stream which records kernels and memory operations that need to be executed in the order they have been inserted. All operations associated with a single OpenMP target directive share the same `__tgt_async_info` object while different directives (usually) have different objects. This existing scheme [13] allows for concurrent and interleaved execution of operations associated with different target directives on a physical and our virtual GPU. Once the kernel has been added to a VGPU stream, all threads waiting for work are notified through the `WorkAvailable` condition variable. The kernels in the execution queue are then executed in-order, with the appropriate number of threads and teams as required by the kernel.

In order to synchronize, the thread calling synchronize is set to wait on the condition variable `WorkDone` through the `VGPUTy` member function `await` which is notified when the thread array finishes execution on a particular VGPUStream. Of course, the call to `await` must check if indeed the VGPU stream that it has been waiting on is empty, otherwise, it resumes waiting.

If the execution of the target region is to be done synchronously, a `__tgt_async_info` object is constructed within the plugin, a VPGU stream is constructed for it, and this stream is awaited upon before returning from the call to run the target region.

3.3 Execution Environment Information

Each `ThreadEnvironmentTy` object contains information such as the ID of the thread within the warp, thread block, as well as within the global VGPU thread array. In order to access `WarpEnvironmentTy` and `CTAEnvironmentTy` information, the `ThreadEnvironmentTy` object contains pointers to them and can access their environment information from accessor functions. When executing a kernel, each target region must know which team is being executed and the total number of teams. Therefore, a `ThreadBlockEnvironmentTy` object is set up for each thread block before the kernel is executed and destroyed after the kernel is executed. The `WarpEnvironmentTy` object contains the ID of the Warp environment within the global array of Warps, as well as the number of threads in the Warp.

Each CTA is handled by a `CTAEnvironmentTy` object. This object contains the number of threads in the CTA, the number of warps in the CTA, and the ID of the CTA within the global array of CTA environments as member fields read by accessor functions `getId()`, `getNumThreads()`, `getNumBlocks()`.

3.4 Synchronization

The synchronization primitives required by the device runtime library are memory fencing on the team, on the kernel, synchronizing threads within the warp, as well as providing a named barrier to synchronize threads within the CTA as well as a generic named barrier. These primitives are called from each individual thread so these calls are forwarded through the thread-local `ThreadEnvironmentTy` to the relevant Warp/CTA environment. In order to support fences in the kernel, we simply use `std::atomic_thread_fence` with the `std::memory_order` that is requested by the device runtime.

The warp environment contains a single barrier for synchronizing the the threads in the warp, and two additional barriers to synchronize calls to `shuffle` and `shuffleDown` appropriately. The details of how the shuffle-related barriers are used are explained in Section 3.5.

The CTA Environment contains three barriers, one for implementing a memory fence through a barrier, one to synchronize all threads in the CTA, and a last one to provide a named barrier to the device runtime library.

All barriers are implemented via `std::barrier`, which is only available in C++20. Luckily, `libc++` [10], part of the LLVM project, already supports the feature, while `libstdc++` [6] support is still underway. As a result, the VGPU plugin, and the OpenMP program offloading to VGPU, need to be compiled with `libc++` until support of the required features in `libstdc++` matures.

3.5 Shuffle

Parallel reduction is a common building block for many parallel programs. Earlier implementations used shared memory, which involves writing data to shared memory, synchronizing, and then reading the data back from shared memory. Modern GPUs support efficient parallel reductions exchange data between threads within the same thread block via shuffle instructions, such as `__shfl` in Nvidia and AMD instruction sets [11]. However, shuffle instructions are not supported by nearly all host architectures, although some

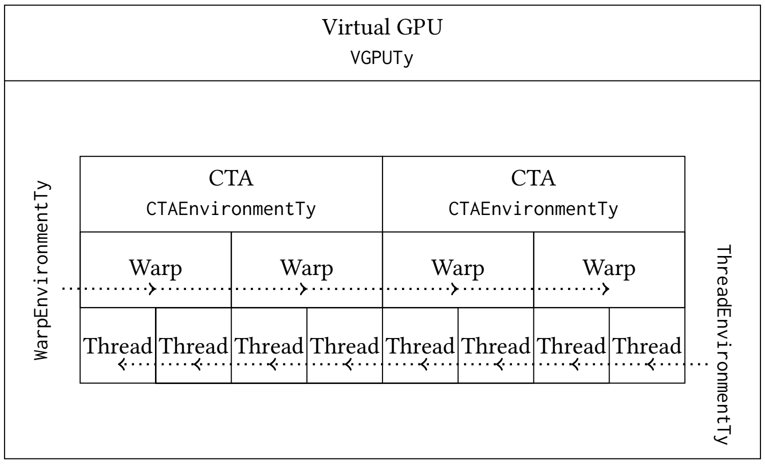

Figure 3: The architecture of the Virtual GPU with its environment type hierarchy.

do provide instructions with similar names for different purposes, such as the x86 SSE instruction `shufps` [16].

The used LLVM/OpenMP device runtime requires two warp shuffle instructions, `__shfl` and `__shfl_down`. Although [16] proposed a `shuffle` clause to OpenMP, it is not standardized or implemented, and could only be used within a `parallel` and `teams` region, which is not sufficient for our use case. In order to support the native GPU shuffle semantics in the VGPU we implemented them explicitly, as shown in Listing 1. In each `WarpEnvironmentTy` object, there is a memory buffer with 4 × number of threads bytes. During a shuffle, the buffer is used as temporary storage that emulates the registers involved in a native GPU shuffle operation. In addition, as mentioned in Section 3.4, two barriers are set for synchronization. Threads write their data to the right slots in the buffer and reach the first barrier to ensure all writes have been performed. Afterwards, threads read from their respective slot the "register value" of another thread in the warp before a second barrier ensures all reads have been performed. Both, `shuffle` and `shuffleDown` are member functions of the `WarpEnvironmentTy` and will be called by each thread participating in the shuffle. It is worth to note that we leverage the fact that the OpenMP device runtime ensures only entire warps will participate in a shuffle operation, this is also the case for execution on a native GPU.

```
int32_t shuffle(int32_t V, uint64_t Src) {
  uint64_t Id = getThreadIdInWarp();
  WarpEnv->writeShuffleBuffer(V, Id);
  WarpEnv->waitShuffleBarrier();
  V = WarpEnv->getShuffleBuffer(Src);
  WarpEnv->waitShuffleBarrier();
  return V;
}

int32_t shuffleDown(int32_t V, uint32_t D) {
  uint64_t Id = getThreadIdInWarp();
  WarpEnv->writeShuffleBuffer(V, Id);
  WarpEnv->waitShuffleDownBarrier();
  V = WarpEnv->getShuffleBuffer(
    (Id + D) % getWarpSize());
  WarpEnv->waitShuffleDownBarrier();
  return V;
}
```

Listing 1: Implementation of `shuffle` and `shuffleDown` as part of the `WarpEnvironmentTy`.

4 COMPILER AND RUNTIME SUPPORT

An OpenMP program with target offloading is typically compiled via the following command (assume offloading to Nvidia GPU):

```
clang -fopenmp -fopenmp-targets=nvptx64 ...
```

where `nvptx64` is the target architecture type. If a specific GPU architecture is to be targeted, it can be specified by the additional flags such as `-Xopenmp-target -march=sm_75` for targeting the Nvidia Turing architecture.

4.1 Virtual Target Triple(s)

VGPU is taken as a new set of targets for OpenMP offloading, depending on the actual underlying host architecture. We could potentially set `-fopenmp-targets=vgpu` and let the clang driver to detect the host architecture. However, it lacks the ability of cross compilation, which is arguably important for some super computers, such as Fugaku supercomputer [4], whose login nodes and compute nodes are different architectures and there is no compiler in compute nodes. Instead of setting a fake target, we add a new vendor `vgpu`. The command argument is `-fopenmp-targets={arch}-vgpu` where `{arch}` is the target architecture type. For example, program compiled with `-fopenmp-targets=x86_64-vgpu` can run on `x86_64` platform. In this way, users can still build the program on one architecture and execute it on another one, like before.

4.2 Code Generation

At the time of this writing, most of code generation for target devices are implemented in the class `CGOpenMPRuntimeGPU`. Target specific functions, such as `getGPUWarpSize` and `getGPUNumThreads`, are implemented in CGOpenMPRuntimeAMDGCN and CGOpenMPRuntimeNVPTX for AMD and Nvidia GPUs respectively, where the two classes inherit the class CGOpenMPRuntimeGPU. If the target device is not Nvidia's or AMD's, it will fall back to the version in the class CGOpenMPRuntime.

One of the key differences between VGPU and classical host offloading is that we reuse code generation as a physical GPU. We set the code generation to a new class CGOpenMPRuntimeVGPU, which inherits CGOpenMPRuntimeGPU as well, if the vendor is vgpu. Since the device runtime already has interface functions to get device information such as warp size, those target dependent functions are implemented by simply emitting corresponding function calls.

4.3 Shared Variable Expansion

GPU memory hierarchy contains more levels compared to CPU memory. Shared memory is one of them, which is a per-thread-block trunk of memory and can only be accessed by threads within the thread block. It can usually improve performance if shared memory is used appropriately. In the used LLVM/OpenMP device runtime, a number of objects are allocated in shared memory space, such as team state and smart shared memory manager. However, in the generated LLVM IR, the address spaces that are not supported by the target are usually ignored by the back end, leading to data races at runtime.

In order to tackle the difference, we implemented a code transformation in middle end to correct the accesses of shared variables. As shown in Listing 2, for each global variable in shared address space, a new global array of size max number of thread blocks is created whose element type is same as the shared variable. Each access of the shared variable will be replaced by an access to the element of the array indexed by the thread ID in the thread block. This method is transparent to the device runtime, thus does not require any change in the device runtime.

```
for (auto &G : M.globals()) {
  // If the global is not a shared variable,
  // move on.
  if (G.getAddressSpace() != SHARED)
      continue;
  // Create a new global variable which is
  // an array of G->getType().
  auto *T = ArrayType::get(G->getType(),
                 MaxNumBlocks);
  auto *A = M.getOrInsertGlobal(T);
  // For each use of G, create a function call
  // to get the block id, get the pointer to
  // the element indexed by block id, and set
  // the use to the element.
  for (auto &U : G.uses()) {
    auto *Id = CallInst::Create(BlockIdFn);
    auto *V = GetElementPtrInst::Create(A,
                  {Zero, Id});
    U.set(V);
  }
}
```

Listing 2: Simplified shared variable expansion.

4.4 Device Runtime Support

The LLVM/OpenMP device runtime we used for this work is an experimental rewrite of the existing one that relies on C++ and OpenMP, except for some target dependent functions that are implemented with LLVM/Clang intrinsics [12]. Those functions are for execution environment information query, synchronization, and misc utilities (such as time, and shuffle). In order to support a new target VGPU, all target dependent functions need to be implemented accordingly inside of begin/end declare variant regions. Since there is no actual hardware, and the execution model of a GPU is different from CPU, no existing intrinsics can meet the requirement. Note that we could implement VGPU support also in the current LLVM/OpenMP device runtime but it would require more involved infrastructure due to the runtime design.

For the VGPU, target specific intrinsics are emulated via the per-thread ThreadEnvironmentTy object described in Section 3.3. By calling the accessor function getThreadEnvironment, the thread can get a pointer to its own ThreadEnvironmentTy object. Thus, all target dependent functions are implemented by calling corresponding member functions of the ThreadEnvironmentTy object. For example, getThreadEnvironment()->getThreadIdInBlock() returns the thread ID in a thread block, equivalent to threadId.x in CUDA or the LLVM-IR llvm.nvvm.read.ptx.sreg.tid.x intrinsic available when targeting NVPTX.

In order to conditionally declare the functions relevant to the VGPU runtime, an OpenMP declare variant clause [2] is used as shown in Listing 3.

```
#pragma omp begin declare variant.           \
        match(device={kind(cpu)})
uint32_t getThreadIdInBlock() {
  return getThreadEnvironment()->
           getThreadIdInBlock();
}
#pragma omp end declare variant
```

Listing 3: Using OpenMP's declare variant to conditionally declare device runtime functions akin to the vendor-specific GPU device runtime functions.

5 EVALUATION

In order to test the VGPU design, we compared the execution time of OpenMP target offloading applications when ran with the classical x86_64 host offloading and on the the virtual GPU. While the benefit of the VGPU offloading is in development and debugging, it is crucial for user experience that the performance is comparable to native host execution. For the benchmarks, our VGPU was initialized with the following configurations (threads per warp/warps per CTA): 16/1, 8/2, 4/4, 2/8, and 1/16, since the benchmark machine had a maximum hardware concurrency limit of 16 threads. For our benchmarks, we used a machine with AMD Ryzen 7 3800X and 32 GB of DDR4 RAM, running a Linux distribution. We used our fork of the trunk Clang/LLVM that contains the new device runtime library, the required clang changes, as well as the new libomptarget plugin required to provide the virtual GPU.

Benchmark	x86_64 Absolute Time (s)	16/1	8/2	4/4	2/8	1/16
bfs-graph65536.txt	0.0106	1.44	1.46	1.50	1.40	1.48
bfs-graph1MW_6.txt	0.05145	1.87	1.99	1.98	1.93	1.90
nw	0.3847	1.12	1.13	1.14	1.12	1.12
hotspot-64	0.2885	1.60	1.53	1.47	1.57	1.47
hotspot-512	0.2832	1.53	1.50	1.54	1.52	1.43
hotspot-1024	0.2890	1.53	1.45	1.53	1.55	1.68
RSBench-small	4.522	1.93	-	-	-	-
RSBench-large	19.190	1.73	-	-	-	-
XSBench-small	1.995	1.16	-	-	-	-
XSBench-large	9.090	1.10	-	-	-	-

Table 1: The absolute running time of x86_64 host offloading for each benchmark run and the slow-down factor of running the benchmark on each of the aforementioned VGPU configurations in (threads per warp/warps per CTA) form.

5.1 Benchmarks

The Rodinia Benchmark Suite [3] is a benchmark suite developed for heterogeneous programming models such as OpenMP, OpenCL, and CUDA. Within this suite, there are number of OpenMP benchmark programs, and a small subset of them contain OpenMP target offloading: bfs, hotspot, cfd, nw, and lud. Of these, we ran bfs - a simple breadth first search application on two of the benchmark input graphs of different sizes: `graph65536.txt`, and `graph1MW_6.txt`. The third provided input for bfs in the Rodinia suite contained a graph too small for us to accurately compare the running time between the virtual GPU and the classical x86 host offloading, so it has been excluded from benchmarking. For nw - a nonlinear global optimization program for DNA sequence alignments, we conducted a run occurred with the arguments `100000/100000 3 16`. Other argument sets for nw runs were either infeasible within the benchmark environment or would result in inaccurate comparisons. For hotspot - a program used to estimate processor temperature, we ran the following arguments as inputs: `10 10 1000000 16` with the temperature and power files being appropriately paired i.e. `temp_64` with `power_64`. We also ran the RSBench [14] and XSBench [15] mini-apps that contain a key kernel of the Monte Carlo neutron transport algorithm. Each benchmark was run a hundred times and only benchmarks with an appropriately small variance are discussed.

5.2 Results

From Table 1, we see that the VGPU induces up to a 2× slowdown on the subset of Rodinia benchmarks with statistically meaningful running times. This initial implementation of a virtual GPU indicates that there is an acceptable class of OpenMP target offloading programs that can be tested and debugged with the virtual GPU target device rather than using classical host offloading which does not use Clang's GPU code generation nor the device runtime. While we expect the slowdown to be reduced as we improve the implementation, the VGPU already provides developers a practical alternative to develop and debug code intended for physical GPU targets such as AMD, NVIDIA, or Intel GPUs.

5.3 Debugging Support

We were also able to run OpenMP target offloaded to VGPU programs through debuggers such as gdb [5] and lldb [7]. In addition to being able to set breakpoints within the target regions to debug, we were also able to access stack traces in case of segmentation faults within a target region, as an example, see Listing 4. This ensures that the primary requirement of the virtual GPU plugin providing access to host-only debugging tools while utilizing the OpenMP (GPU) target offloading toolchain is satisfied.

```
* thread #2, name = 'XSBench', stop reason =
    signal SIGSEGV: invalid address (fault address
    : 0x0)
  * frame #0: 0x... tmpfile_gmU3b1`
      fast_forward_LCG(seed=1070, n=0) at
      Simulation.c:371:20
    frame #1: 0x... tmpfile_gmU3b1`
        __omp_outlined___debug__.1(...) at
        Simulation.c:59:10
  ...
```

Listing 4: Partial stacktrace from XSBench that offloads onto the VGPU target

6 RELATED WORKS

OpenMP 4.0 provides mechanisms to offload regions of code to accelerators. Antao et al. [1] introduces an OpenMP offloading implementation to LLVM targeting Nvidia GPUs. Later the implementation was extended to support AMD GPUs. In addition, LLVM infrastructure also supports host offloading to `x86_64`, `AArch64`, `PPC64[LE]` [8]. Like we discussed before, the support is totally different from our design, and the programs run like a regular OpenMP program. Recently the device runtime was rewritten with OpenMP, making it possible to compile the device runtime and run on host [12]. For other programming models providing offloading support, similar solutions have been explored, such as [9] for OpenCL. To the best of our knowledge, our work is the first to emulate GPU execution for OpenMP offloading programs.

7 CONCLUSION AND FUTURE WORK

In this work, we propose a novel way to develop and debug OpenMP offloading program via a virtual GPU (VGPU) that emulates GPU execution on the host. To make sure that VGPU can provide a same development environment as a native GPU, we implemented a virtual GPU device environment that can provide same execution model and functionalities, combined with the GPU compilation paths and runtime libraries. The evaluation results show that OpenMP target offloading programs can be tested and debugged on host with our proposed VGPU with reasonable slowdowns compared to host-tuned execution.

We intend to improve the performance of our VGPU implementation and to add more features, such as custom scheduling policies, separate process space (to support data movement debugging), as well as sanitizer support.

ACKNOWLEDGMENTS

This work 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.

REFERENCES

[1] Samuel F. Antao, Alexey Bataev, Arpith C. Jacob, Gheorghe-Teodor Bercea, Alexandre E. Eichenberger, Georgios Rokos, Matt Martineau, Tian Jin, Guray Ozen, Zehra Sura, Tong Chen, Hyojin Sung, Carlo Bertolli, and Kevin O'Brien. 2016. Offloading Support for OpenMP in Clang and LLVM. In *The Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*. Salt Lake City, UT, USA, 1–11.

[2] OpenMP Architecture Review Board. 2020. OpenMP Application Programming Interface Version 5.1. https://www.openmp.org/spec-html/5.1/openmp.html

[3] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE international symposium on workload characterization (IISWC)*. Ieee, 44–54.

[4] Fujitsu. 2021. Supercomputer Fugaki. https://www.fujitsu.com/global/about/innovation/fugaku/

[5] GNU. 2021. GDB: The GNU Project Debugger. https://www.gnu.org/software/gdb/

[6] GNU. 2021. libstdc++ - GCC. https://gcc.gnu.org/onlinedocs/libstdc++/manual/status.html#status.iso.2020

[7] GNU. 2021. The LLDB Debugger. https://lldb.llvm.org/

[8] LLVM Developer Group. 2021. OpenMP Support — Clang 13 documentation. https://clang.llvm.org/docs/OpenMPSupport.html

[9] Pekka Jääskeläinen, Carlos Sánchez de La Lama, Erik Schnetter, Kalle Raiskila, Jarmo Takala, and Heikki Berg. 2015. pocl: A performance-portable OpenCL implementation. *International Journal of Parallel Programming* 43, 5 (2015), 752–785.

[10] LLVM. 2021. "libc++" C++ Standard Library. https://libcxx.llvm.org/

[11] Justin Luitjens. 2014. Faster Parallel Reductions on Kepler. https://developer.nvidia.com/blog/faster-parallel-reductions-kepler/

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

[13] Shilei Tian, Johannes Doerfert, and Barbara Chapman. 2020. Concurrent Execution of Deferred OpenMP Target Tasks with Hidden Helper Threads. In *The Workshop on Languages and Compilers for Parallel Computing (LCPC)*. Stony Brook, NY, USA.

[14] 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

[15] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. 2014. XS-Bench - The Development and Verification of a Performance Abstraction for Monte Carlo Reactor Analysis. In *PHYSOR 2014 - The Role of Reactor Physics toward a Sustainable Future*. Kyoto. https://www.mcs.anl.gov/papers/P5064-0114.pdf

[16] Anjia Wang, Xinyao Yi, and Yonghong Yan. 2020. Supporting Data Shuffle Between Threads in OpenMP. In *OpenMP: Portable Multi-Level Parallelism on Modern Systems*, Kent Milfeld, Bronis R. de Supinski, Lars Koesterke, and Jannis Klinkenberg (Eds.). Springer International Publishing, Cham, 98–112.