Enabling CHIP-SPV in Chapel GPUAPI Module

Jisheng Zhao
jisheng.zhao@cc.gatech.edu
Georgia Institute of Technology
Atlanta, Georgia, USA

Akihiro Hayashi
ahayashi@gatech.edu
Georgia Institute of Technology
Atlanta, Georgia, USA

Brice Videau
bvideau@anl.gov
Argonne National Laboratory
Argonne, Illinois, USA

Vivek Sarkar
vsarkar@gatech.edu
Georgia Institute of Technology
Atlanta, Georgia, USA

KEYWORDS

Chapel, GPUs, GPUIterator, GPUAPI, CHIP-SPV

ACM Reference Format:
Jisheng Zhao, Akihiro Hayashi, Brice Videau, and Vivek Sarkar. 2022. Enabling CHIP-SPV in Chapel GPUAPI Module. In *Proceedings of CHIUW '23: The 10th Annual Chapel Implementers and Users Workshop (CHIUW '23).* ACM, New York, NY, USA, 4 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

There has been a growing interest in utilizing GPUs in large-scale systems. While we believe PGAS languages such as Chapel [2] are suitable not only for homogeneous nodes but also for heterogeneous nodes, GPU programming with PGAS languages in practice is still limited since there is still a big performance gap between compiler-generated GPU code and hand-tuned GPU code. Additionally, hand-optimization of CPU-GPU data transfers is an important contributor to this performance gap.

In our past work, we proposed the `GPUAPI` module [5, 7], which includes a wide variety of Chapel-level GPU API that allows the user to write device memory (de)allocation and device-to-host/host-to-device data transfer in Chapel. While the user still has to write a GPU kernel manually, once the user writes a working Chapel program with the `GPUAPI` module and their GPU kernel on one platform, the Chapel+GPU program should run on another platform without any further modifications. We demonstrated that the module can be utilized in real-world Chapel applications such as ChOp [1] and CHAMPS [11] across multiple CPU-GPU platforms and it facilitates hand-optimization of CPU-GPU data transfers in CHAMPS [8].

Also, the `GPUAPI` module is designed to comply with Chapel's multi-resolution concept, where the user has the option of providing a high-level specification and also of diving into lower-level details to incrementally evolve their implementations for improved performance on multiple CPUs+GPUs nodes particularly when it is used with the `GPUIterator` module [4]. Specifically, we proposed to introduce the following spectrum of GPU programming abstraction in Chapel [6] and the `GPUAPI` module is mainly responsible for the MID-level/MID-LOW-level part:

- **HIGH-level/HIGH-MID-level**: The compiler compiles `forall` / `reduce` constructs to GPUs and generates all the host part required for GPU execution (HIGH). For the host part, the user has the option of using our GPUAPI to optimize device memory allocation and data transfer (HIGH-MID).
- **MID-level/MID-LOW-level**: The user writes 1) GPU kernels in a low-level GPU language, and 2) the host part in Chapel in either/both of two levels of abstraction: a Chapel programmer-friendly version (MID), and a thin wrapper version of raw GPU API routines (MID-LOW).
- **LOW-level**: The user writes a full GPU program in a low-level GPU language and call it from Chapel using the C interoperability feature.

In this talk, we focus on enhancing our support for Intel GPUs in the `GPUAPI` module. Essentially, we introduce the CHIP-SPV framework [3] as a backend for the module, which not only allows the user to run their hand-written CUDA/HIP kernels on Intel GPUs as-is, but also allows the runtime to perform finer-grain control of Intel GPUs through Intel Level Zero runtime.

Our key contributions include:

(1) Design and Implementation of our CHIP-SPV backend in the `GPUAPI` module.
(2) Preliminary Evaluation of the backend on two Intel GPU platforms.

2 GPUAPI

The `GPUAPI` [5, 7] module is a standalone Chapel module, which includes a wide variety of platform-independent Chapel-level GPU routines, including device memory (de)allocation routines and device-to-host/host-to-device data transfer routines, which can run on different GPU platforms.

The module mainly supports NVIDIA CUDA-supported GPUs, AMD ROCm-supported GPUs, and Intel Level Zero-supported GPUs (and FPGAs) through different vendor-provided libraries/frameworks as shown in Figure 1.

Runtime implementation: One of the interesting aspects of our implementation is that there is only a CUDA implementation of the `GPUAPI` module. We utilize the `hipify` from AMD and `dpct` from Intel to convert the CUDA implementation to a HIP and DPC++ version respectively. Additionally, on Intel platforms, we

CHIUW '23, June 1, 2023, Virtual Format

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

	CUDA	HIP	SYCL
NVIDIA	✓	✓	✓
AMD	✓(via `hipify`)	✓	✓
Intel	✓(via `CHIP-SPV` or `dpct`)	✓(via `CHIP-SPV`)	✓

Table 1: How user-written kernels work on different GPU platforms.

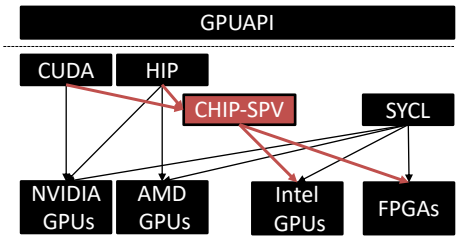

Figure 1: Multi-platform support in the `GPUAPI` module.

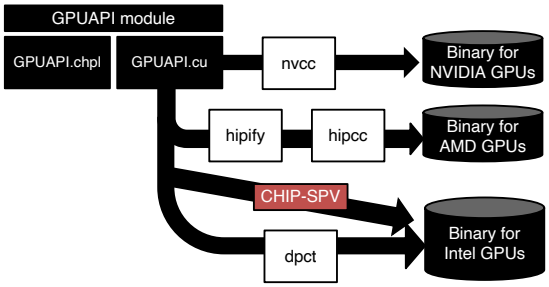

Figure 2: The implementation of the `GPUAPI` module.

used to provide another way to run the `hipify`ed runtime code with HIPLZ [12] and we have updated our Intel backend implementation as HIPLZ transitions to CHIP-SPV, which is the main topic of this work.

More specifically, at the time of installation, our cmake-based build system identifies installed GPUs and generates an appropriate static (.a) and/or shared (.so) library with the conversion - e.g., `libGPUAPICUDA.so` for NVIDIA GPUs (Figure 2).

User-written GPU kernels: The user is supposed to write the kernel part using vendor-provided GPU libraries/frameworks such as CUDA, HIP, SYCL, and so on. The user can simply write their kernels using their favorite framework and link it with the corresponding version of GPUAPI library (e.g., `libGPUAPICUDA.so`). If there is any conversion required, the user can also utilize our cmake-based build system. Table 1 summarizes how user-written kernels work on different GPU platforms.

The general applicability of this auto-conversion approach: It is also worth noting that this auto-conversion approach works very well even with real-world applications. For example, while the kernel part of ChOp and CHAMPS are originally implemented in CUDA, the `hipify` tool is able to produce the HIP version and run it on an AMD GPU flawlessly [5, 7, 8].

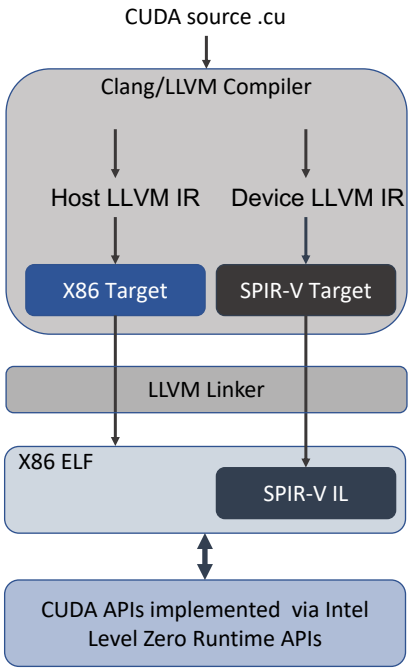

Figure 3: CHIP-SPV Code Generation.

3 CHIP-SPV

The design goal of CHIP-SPV [3] is to connect the heterogeneous programming model with low-level runtime library, especially for Intel Level Zero. CHIP-SPV is extended from an early work HIPLZ [12] that bridges the HIP programming model to Intel Level Zero. To support CUDA programming model, there should be 1) a compiler that takes CUDA source code to generate an executable binary and 2) a runtime library that implements CUDA API routines to support target CUDA GPUs. As shown in Figure 3, CHIP-SPV employs a CUDA-compatible compiler frontend based on the LLVM/Clang [10] that translates CUDA source code to two parts of LLVM intermediate representation: host IR and device IR (generated from CUDA kernel functions). The host part is processed via the legacy LLVM x86 backend to produce an x86 binary, and the device part is processed via the LLVM SPIR-V backend to produce SPIR-V IR. The x86 binary and the SPIR-V IR are then linked together to make an x86 executable binary (or shared library) that is embedded with SPIR-V (a fat binary). The linked binary interacts with a GPU via CUDA API routines that are implemented by Intel Level Zero runtime routines. The SPIR-V IR is translated GPU binary via the JIT compiler in the vendor driver (i.e., Intel GPU driver).

4 USING CHIP-SPV AS CUDA/HIP BACKEND

As mentioned in Section 2, user-written GPU kernels in user's program and device-related routines in `GPUAPI.cu` are supposed to be compiled by the `nvcc` compiler. This is where CHIP-SPV comes into play. Specifically, CHIP-SPV compiles those CUDA programs to a binary that targets an Intel GPU or any other Intel Level Zero compatible GPU. Figure 4 presents the workflow that builds the

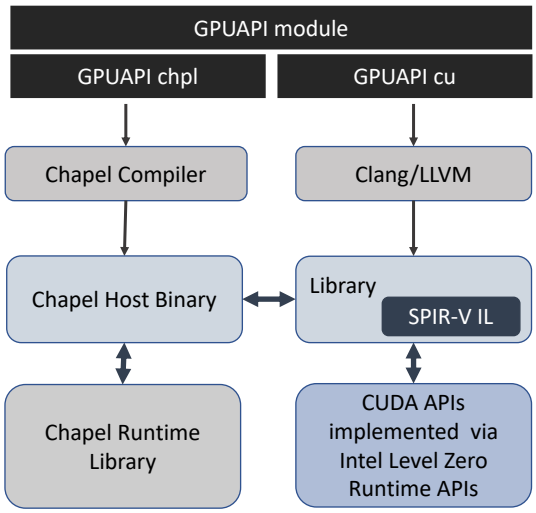

Figure 4: CHIP-SPV Code Generation.

GPUAPI module (`GPUAPI.chpl` and `GPUAPI.cu`) to Intel GPUs. As with the other existing backends in Figure 2, the Chapel part is compiled via the Chapel compiler and the CUDA part is compiled to a shared library via Clang/LLVM. At runtime, the Chapel host binary invokes the library to execute the GPU part on the target Intel GPU.

5 EVALUATION

We implemented a CHIP-SPV-based `GPUAPI` backend and validated the `GPUIterator+GPUAPI` version of different benchmarks worked on the following two Intel-integrated GPU platforms:

- Intel Gen 9: which is composed of a 4-Core Intel Xeon Processor E3-1585 v5 CPU running at 3.5GHz and Iris Pro Graphics P580 (GT4e) Gen9 GPU with a peak clock rate at 1.15GHz and 72 execution units;
- Intel UHD Graphics 770: which is composed of a 12-Core Intel i7-12700 CPU running at 2.1GHz and UHD770 with a peak clock rate of 1.45GHz and 96 execution units.

The driver libraries used in this experiment are Intel Compute Runtime NEO version: driver version 22.02 for Gen9 and 23.09 for UHD Graphic 770.

We used Chapel version 1.29 and employed five benchmarks (shown in Table 2) to evaluate the performance of the `GPUIterator` module and the `GPUAPI` module with the CHIP-SPV backend enabled. In each benchmark, the iteration spaces of `forall` loops are divided between CPU and GPU in different percentages (CPU X% and GPU Y%, where $X + Y = 100$). It is worth noting that it is safe to assume that the performance of CPU100%+GPU0% is equivalent to that of the original Chapel `forall` version as the `GPUIterator` module internally uses the same parallel iteration mechanism as the original one [4]. Also, in all the five benchmarks, we verified that the GPU variants produce the same output as what the CPU100%+GPU0% variant produces.

The experimental results obtained on Gen9 (see Figure 5) and UHD Graphics 770 (see Figure 6) show that BlackScholes, Logistic Regression and Matrix Multiplication gain performance enhancement when increasing GPU workload on the Gen9 GPU. However, due to the restriction of the integrated GPU (i.e., built on the same chip with CPU and share memory channel and processing power), the performance enhancement is limited in general.

Table 2: Benchmarks and input data size used in our evaluation .

Benchmark	Description	Data Size
Vector Copy	A simple vector kernel	n = 200×2^{20}
Stream	A simple vector kernel	n = 200×2^{20}
BlackScholes	The Black-Scholes equation	n = 200×2^{20}
Logistic Regression	A classification algorithm	f = 2^{16} s = 32
Matrix Multiplication	Matrix-Matrix multiply	n = 2^{11}

Although the performance improvement is not significant enough on the integrated GPU, it is important to note that 1) we were able to verify the end-to-end compilation and execution flow of the CHIP-SPV backend using five different applications on the Intel GPU and 2) we believe employing CHIP-SPV as a backend for `GPUAPI` is a reasonable way to help target `GPUAPI` to different GPU platforms, in particular Intel GPU platforms. Additionally, one inherent benefit of using `GPUIterator` + `GPUAPI` is that this allows the user to explore the best performing CPU-GPU ratio easily even when another GPU platform is used.

6 CONCLUSIONS

In this talk, we present the work that employs CHIP-SPV as a backend for Chapel `GPUAPI` module to support compiling and executing CUDA/HIP code on Intel Level Zero runtime-compatible GPU systems. We evaluate our prototype on two Intel-integrated GPU platforms: Gen9 and UHD. The experimental results show that using CHIP-SPV helps target GPUAPI to more GPU platforms.

Our further plan is to evaluate our module on a discrete Intel GPU (e.g. XE), on which we anticipate such a GPU gives more performance improvements over CPUs as in our previous work [4]. Also, since CHIP-SPV is extended to support OpenCL runtime in addition to Intel Level Zero runtime, we also plan to use other OpenCL-supported GPU platforms.

Furthermore, we plan to discuss the possibility of using CHIP-SPV as a general code generation target in Chapel's GPU code generator [9] to enhance its portability. In that case, the Chapel compiler translates Chapel's GPU loops as device-specific LLVM IR, and lets CHIP-SPV's compiler toolchain translate them into a fat binary that can run on various GPU architectures that supports either Intel GPUs or OpenCL.

7 ACKNOWLEDGEMENTS

This work was supported by the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357, and 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). We also gratefully acknowledge the computing resources provided and operated by the Joint Laboratory for System Evaluation (JLSE) at Argonne National Laboratory.

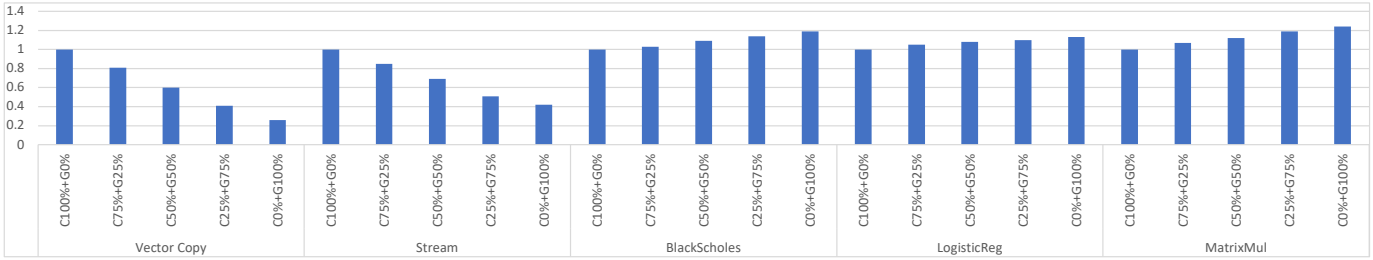

Figure 5: Performance evaluation over the original FORALL (4 workers) on the Intel Xeon E3-1585 v5 CPU + Gen9 GPU.

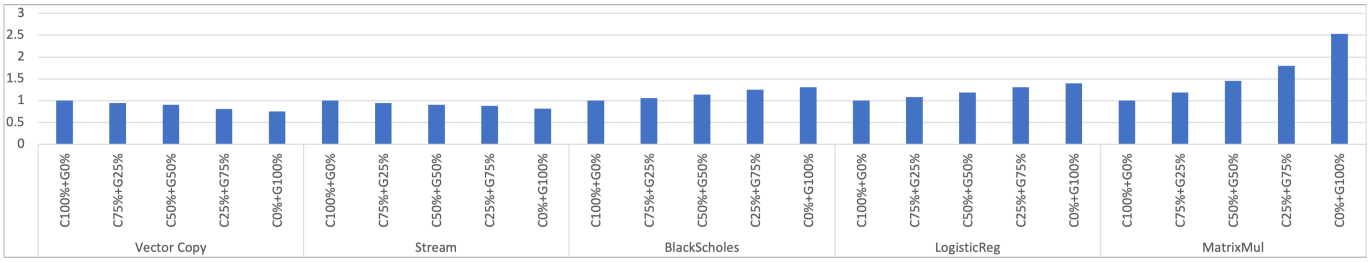

Figure 6: Performance evaluation over the original FORALL (12 workers) on the Intel i7-12700 CPU + UHD Graphics GPU.

REFERENCES

[1] Tiago Carneiro, Nouredine Melab, Akihiro Hayashi, and Vivek Sarkar. 2021. Towards Chapel-based Exascale Tree Search Algorithms: dealing with multiple GPU accelerators. In *HPCS 2020 - The 18th International Conference on High Performance Computing & Simulation*. Barcelona / Virtual, Spain. https://hal.archives-ouvertes.fr/hal-03149394

[2] Bradford L. Chamberlain. 2011. Chapel (Cray Inc. HPCS Language). In *Encyclopedia of Parallel Computing*. 249–256. https://doi.org/10.1007/978-0-387-09766-4_54

[3] CHIP-SPV. 2023. chip-spv. https://github.com/CHIP-SPV/chip-spv.

[4] Akihiro Hayashi, Sri Raj Paul, and Vivek Sarkar. 2019. GPUIterator: Bridging the Gap between Chapel and GPU Platforms *(CHIUW 2019)*. Association for Computing Machinery, New York, NY, USA, 2–11. https://doi.org/10.1145/3329722.3330142

[5] Akihiro Hayashi, Sri Raj Paul, and Vivek Sarkar. 2022. A Multi-Level Platform-Independent GPU API for High-Level Programming Models. In *High Performance Computing. ISC High Performance 2022 International Workshops*, Hartwig Anzt, Amanda Bienz, Piotr Luszczek, and Marc Baboulin (Eds.). Springer International Publishing, Cham, 90–107.

[6] A. Hayashi, S. Raj Paul, and V. Sarkar. 2020. Exploring a multi-resolution GPU programming model for Chapel. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 675–675. https://doi.org/10.1109/IPDPSW50202.2020.00117

[7] A. Hayashi, S. Raj Paul, and V. Sarkar. 2021. GPUAPI: Multi-level Chapel Runtime API for GPUs. In *The 8th Annual Chapel Implementers and Users Workshop(CHIUW)*.

[8] A. Hayashi, S. Raj Paul, and V. Sarkar. 2022. Accelerating CHAMPS on GPUs. In *The 9th Annual Chapel Implementers and Users Workshop(CHIUW)*.

[9] Engin Kayraklioglu, Andy Stone, David Iten, Sarah Nguyen, Michael Ferguson, and Michelle Strout. 2022. Targeting GPUs Using Chapel's Locality and Parallelism Features. In *The 9th Annual Chapel Implementers and Users Workshop(CHIUW)*.

[10] LLVM.org. 2023. Clang: a C language family frontend for LLVM. http://clang.llvm.org/.

[11] Matthieu Parenteau, Simon Bourgault-Cote, Frédéric Plante, Engin Kayraklioglu, and Eric Laurendeau. 2021. Development of Parallel CFD Applications with the Chapel Programming Language. In *AIAA Scitech 2021 Forum*. https://doi.org/10.2514/6.2021-0749

[12] Jisheng Zhao, Colleen Bertoni, Jeffrey Young, Kevin Harms, Vivek Sarkar, and Brice Videau. 2022. HIPLZ: Enabling Performance Portability for Exascale Systems. In *HeteroPar 2022: Euro-Par Workshop, August, 2022, Proceedings (Lecture Notes in Computer Science)*, Maciej Malawski and Krzysztof Rzadca (Eds.). Springer.