Recent GPU Programming Improvements in Chapel

Engin Kayraklioglu
engin@hpe.com
Hewlett Packard Enterprise

Andy Stone
andy.stone@hpe.com
Hewlett Packard Enterprise

Daniel Fedorin
daniel.fedorin@hpe.com
Hewlett Packard Enterprise

ABSTRACT

Chapel's emerging native GPU programming support has improved considerably in the last year. In this talk, we will highlight some of the improvements and discuss our next steps.

1 INTRODUCTION

GPUs continue to gain popularity and importance in high-performance computing. In November 2005, none of the compute power of the supercomputers in TOP500 [1] came from accelerators; whereas in 2022, 60% of it did. Uses of GPUs are more prevalent near the top of the most recent list, where 7 out of top 10 systems are equipped with GPUs.

In this landscape, providing native GPU programming support is one of the key priorities of the Chapel team at Hewlett Packard Enterprise. We have been working towards that goal for some time with increased intensity over the last couple of years. This talk will focus on recent improvements in portability, performance and features. We will also discuss near- and long-term goals. Whereas the talk itself will give a quick recap on how to program GPUs natively in Chapel, the details are omitted here for brevity. The reader is encouraged to peruse our submission in CHIUW 2022 [2], notes from the last year [3, 4], and the GPU technote [5] for learning more about GPU programming with Chapel.

2 RECENT UPDATES

In the talk we will focus on the following updates as we believe they are the most important for the users. However, we are planning to enumerate other noteworthy improvements.

2.1 Portability

2.1.1 AMD Support. Until Chapel 1.30, only NVIDIA GPUs could be targeted. Version 1.30 features our initial support for targeting AMD GPUs. This support is built on ROCm, where the runtime GPU layer is implemented in HIP. The compiler implementation for GPU support is mostly common between NVIDIA and AMD targets, differing only in the final steps of generating the binary. There are two known limitations specific to the AMD target: (1) it can only be used in single-locale builds, (2) a subset of 64-bit math functions are not supported.

2.1.2 The Simulated GPU layer. We have been working on a new simulated GPU layer. This is a development/debugging-oriented layer that allows the GPU locale model to be used in systems without GPUs. With this layer, programmers do not need to run on systems with GPUs to use features like `assertOnGpu` to understand whether their loops will be transformed into kernel launches. Further diagnostic features from the `GpuDiagnostics` module can be used to count kernel launches or verbosely report when a launch occurs. As of today, this is still a work-in-progress, but we are aiming to include it in version 1.31.

2.1.3 Next Steps. In the near-term, we are focusing on extending AMD support to enable multi-locale execution. We target achieving feature parity with NVIDIA within a few releases. We are also planning to target Intel GPUs as they become more available.

2.2 Performance

We have started looking into the performance of our native GPU programming support. In this section of the talk, we will talk about two of the key performance optimizations that are in Chapel 1.30.

2.2.1 Kernel Launch Improvements. In the early prototypes, our runtime would to load the GPU binary every time a kernel launch occurs. This resulted in easier implementation. However, it was also a source of considerable overhead per kernel launch. This was especially noticeable in short-running kernels. We have fixed this performance issue by eagerly loading the GPU binary during application startup, which resulted in around 300x faster kernel launch times. See the "1.30 Prerelease" curve in Figure 1 for the performance improvement in HPCC Stream with varying vector sizes.

2.2.2 Kernel Execution Improvements. We have also made progress in eliminating costs coming from kernel execution. Chapel arrays have metadata that typically needs to be accessed/used per access. Execution time overheads for that are typically not noticeable on CPUs. However, we've observed that not to be the case for GPUs, potentially due to limited cache per thread. Our initial step to eliminate that overhead is to leverage the existing loop-invariant code motion (LICM) optimization in the Chapel compiler. Previously, GPU kernels were generated before this pass, which precluded GPU kernels from benefiting from the optimization. With some modifications to the GPU transformation pass in the compiler, we were able to move it after LICM. This resulted in performance improvements both in benchmarks like HPCC Stream Triad, and user applications like ChOp [7]. See the "1.30" curve in Figure 1 for the performance improvement caused by this optimization.

2.2.3 Next Steps. We are aware of some overheads on CPU execution with `CHPL_LOCALE_MODEL=gpu`, which are even more pronounced in the non-default `CHPL_GPU_MEM_STRATEGY=array_on_device` mode. This is a mode in which the array data is allocated strictly on device global memory rather than managed memory. As such, the `array_on_device` mode improves performance of data copies between device and host significantly. We are planning to make this the default mode, but first, we will focus on resolving the CPU performance issues caused by it.

Furthermore, in the near term, we want to (1) investigate abstract syntax tree (AST) specialization early in the compiler to make some static optimizations for the GPU code paths, and (2) improve LICM to remove array metadata from GPU kernels in more cases.

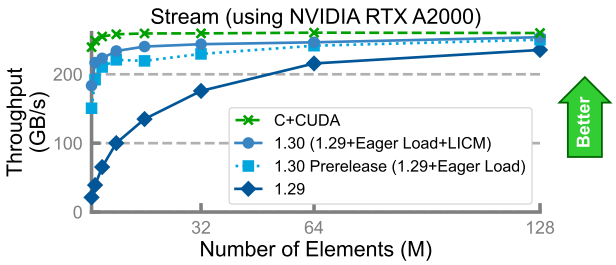

Figure 1: STREAM Performance Progress with Optimizations

2.3 Features

We recommend browsing the `GPU` module documentation [6] to learn more about the current API supporting GPU programming. This API has been improved significantly in the last year, which we will summarize during the talk. Additionally, we will highlight our nascent profiler support.

2.3.1 Initial Support for Profilers with NVIDIA GPUs. Profilers are especially important in the GPU performance analysis workflow, where timers and `writeln`-based introspection are more cumbersome to use than with CPU performance analysis. In earlier versions, we have noticed that while you can use NVIDIA's Nsight Compute profiler to profile kernel execution performance, enabling line number generation thwarted performance optimizations performed by the PTX assembler. In version 1.30, we have added a new flag, `--gpu-ptxas-enforce-optimization`, which ensures that the generated binary is optimized even with the `-g` flag that adds line number information in the emitted PTX. We expect to overhaul how `-g` works with the LLVM backend, and to deprecate this flag in the long term.

2.3.2 Next Steps. We are actively designing and prototyping new features for `forall` and `foreach`. We aim for these features to replace some of the API in the `GPU` module (e.g., standalone functions to set the kernel block size, create block-shared arrays) with portable means that can be used across vendors and across systems with and without GPUs. The current API already provides the former but not the latter. This is one of the major design efforts of the GPU team and expected to span several releases.

REFERENCES

[1] https://www.top500.org/
[2] https://chapel-lang.org/CHIUW/2022/Kayraklioglu.pdf
[3] https://chapel-lang.org/releaseNotes/1.27-1.28/04-ongoing-gpus.pdf
[4] https://chapel-lang.org/releaseNotes/1.29-1.30/04-gpus.pdf
[5] https://chapel-lang.org/docs/main/technotes/gpu.html
[6] https://chapel-lang.org/docs/main/modules/standard/GPU.html
[7] Tiago Carneiro, Jan Gmys, Nouredine Melab, Daniel Tuyttens, Towards ultra-scale Branch-and-Bound using a high-productivity language, Future Generation Computer Systems, Volume 105, 2020, Pages 196-209, ISSN 0167-739X, https://doi.org/10.1016/j.future.2019.11.011.