Targeting GPUs Using Chapel's Locality and Parallelism Features

Engin Kayrakliogl
engin@hpe.com
Hewlett Packard Enterprise

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

David Iten
david.iten@hpe.com
Hewlett Packard Enterprise

Sarah Nguyen
sarahelizabeth.nguyen@hpe.com
Hewlett Packard Enterprise

Michael Ferguson
michael.ferguson@hpe.com
Hewlett Packard Enterprise

Michelle Strout
michelle.strout@hpe.com
Hewlett Packard Enterprise

ABSTRACT

Targeting GPUs using native Chapel features has been one of the most sought after features by the community. In this talk, we will discuss our vision towards targeting GPUs, the current status of the implementation, and future work towards achieving this vision.

1 INTRODUCTION

Graphical Processing Units (GPU) are ubiquitous in HPC systems. As such, supporting GPUs natively has been one of the major requests from Chapel users. Related studies resulted in prototypes that demonstrate targeting GPUs from Chapel is feasible and can deliver significant performance improvements over CPU-only computations [1–3]. However, none of these studies resulted in any native GPU support in Chapel releases.

Other efforts relied on Chapel's C interoperability support to call CUDA kernels from Chapel code via regular extern functions [4, 5]. While that direction is also promising in terms of performance, it precludes using Chapel's high-level constructs and creates dependencies on external libraries and other languages.

In the last couple of releases, the Chapel team at HPE/Cray have been designing and implementing native GPU support in Chapel. The overarching strategy for our effort is to use the existing capabilities of the languages where possible and expand the language to express GPU-based operations intuitively.

In this talk, we will first give more details about our general strategy and vision. Second, we will demonstrate the current state of the implementation. And, finally, we will go over some of the near- and long-term goals in targeting GPUs in Chapel language.

2 GPU PROGRAMMING USING A GENERAL-PURPOSE LANGUAGE

In this section, we outline our vision in supporting GPU programming in Chapel. For the current status of the implementation please refer to Section 3.

2.1 SIMT Parallelism and Vectorization

The GPU architecture and the execution model is unique. Massive parallelism leads to scheduling granularity that is coarser than the hardware parallelism. For example, in a GPU, an instruction is dispatched for groups cores (i.e., warps). This is similar to parallelism via vectorization, where an instruction is dispatched for groups of vector lanes.

Chapel's `forall` loops are the most common data-parallel loops. While `forall` guarantees that its body can be executed in an order-independent fashion, whether the parallel tasks created by the `forall` vectorize their parts of iteration is not guaranteed. This is because the parallel tasks created by the `forall`'s iterator can use `for` loops internally, which do not have any order-independence guarantees. To express order-independent loops in iterators (and elsewhere), we introduced the `foreach` loop. A `foreach` loop does not create parallel tasks. However, it signals that the loop's body can be iterated in an order-independent fashion. Therefore, if a `forall`/`foreach` loop is used with an iterator that's order-independent (i.e., an iterator that uses `foreach` internally), that loop will be vectorized if there is hardware support.

`foreach` loops that are order-independent and vectorizable can also be transformed into GPU kernels. As a result, data parallel loops in Chapel can be translated either to GPU kernels or to vectorized CPU code. In our current design and implementation, `foreach` loops are launched as GPU kernels in contexts where the code indicates it should run on a GPU sublocale if possible.

2.2 GPUs as Units of Locality

The concept of *locale* is key for locality in Chapel. Typically, a locale corresponds to a compute node in a cluster or a supercomputer. The `on` statements are used to "move" the execution from one locale to another. Or, arrays' data can be distributed across locales. Moving execution from one processing unit to another and allocating data on processing units other than the one executing are key capabilities in programming GPUs.

With this observation, we represent GPUs with *sublocales*. Sublocales are nested locales within top-level locales. To control where memory allocations occur and order-independent loops are run (either as a loop on the CPU, or a GPU kernel) we use `on` statements that target GPU sublocales.

2.3 Higher-Level Features

Based on the low-level features above, Chapel's high-level features can be used to target GPUs even more expressively. First, allocations GPU memory can be done via *domain maps*. Domain maps are used to distribute a domain across multiple locales. In most of the standard distributions included in Chapel releases, the locales onto which the data should be distributed can be set using the `targetLocales` field in the domain map initializer. As we discussed above, allocation (and execution) on the GPU is controlled by `on` statements on GPU sublocales. Similarly, a distributed array's data can be partly or wholly allocated on GPUs by setting the `targetLocales` field to use GPU sublocales.

`forall` loops are commonly used for data parallelism and locality in Chapel. How a `forall`'s distribution space is decomposed onto locales is controlled by the iterator that it traverses. These iterators can use on statements to move parts of the execution to different locales. For example, a distributed domain's default parallel iterators typically use on statements to make sure that indices are yielded on locales where they exist. Similarly, if a distributed domain (or array) is distributed on GPU sublocales, its parallel iterators can seamlessly move the execution to GPUs as appropriate.

Just as the iterator implementation can drive the locality of `forall` iterations by using on statements; it can also determine whether or not the loop is a candidate for vectorization or GPU kernel generation. Only `forall` and `foreach` loops that work with iterators that support data parallelism (possibly by using `foreach`) are candidates for vectorization or GPU kernel generation. In summary, based on the lower-level features we outlined above, higher-level features like domain maps and `forall` loops can also be used to target GPUs in Chapel.

This vision for targeting GPUs from native Chapel code follows from Chapel's multiresolution design philosophy: There are lower-level concepts that users can use for more control. On the other hand, higher-level concepts are built on top of the lower-level ones and also available to the user for less effort and potentially higher portability.

3 CURRENT STATUS

As of version 1.26[1], Chapel has limited support for targeting GPUs following the design roughly outlined above. This support can be enabled by setting the environment variable `CHPL_LOCALE_MODEL` to gpu. See the "GPU Programming" tech note [6] for more details on how to setup for GPU execution. For GPU support, the LLVM back-end must be used. We use LLVM to generate PTX assembly code and generate binary from that using CUDA toolchain.

In this version, an on statement can be used to move data allocation and execution to GPUs, `foreach` loops are analyzed for their suitability for GPU execution and are turned into kernels automatically. Below are some examples that the current implementation supports.

3.1 HPCC STREAM on GPU

This example shows how HPCC STREAM can be run on a GPU.

```
1 // NOTE: here.getChild(1) interface is subject to change
2 on here.getChild(1) {
3   var A, B, C: [1..n] int;
4
5   B = 1;
6   C = 2;
7
8   A = B + alpha * C;
9 }
```

Listing 1: HPCC STREAM as of Chapel 1.26

In line 2, the on block starts the GPU region. In this block, array data is allocated on the Unified Virtual Memory (UVM). The operations in lines 5, 6 and 8 are promoted expressions and they typically lower into `forall`/`foreach` loops. Therefore, all of these expressions will launch as individual GPU kernels. A full HPCC STREAM benchmark implemented similar to Listing 1 performs at more than 96% efficiency compared to the corresponding C+CUDA implementation. It is important to note that we have not worked on performance profiling and tuning so far.

[1]Released in March 2022

3.2 Data Copy Between the CPU and the GPU

In this example, the data is initialized on the host, copied onto GPU, STREAM is run, and the result is copied back.

```
1  var AHost, BHost, CHost: [1..n] int;
2
3  BHost = 1;
4  CHost = 2;
5
6  // NOTE: here.getChild(1) interface is subject to change
7  on here.getChild(1) {
8    var A: [1..n] int;
9
10   var B = BHost;
11   var C = CHost;
12
13   A = B + alpha * C;
14
15   AHost = A;
16 }
```

Listing 2: HPCC STREAM with Data Offload

Before the on statement, the arrays are allocated on the main memory as usual. Within the on statement, corresponding arrays are created on the device memory. Lines 10 and 11 relies on the existing bulk array transfer capabilities to copy the arrays' data from the host memory to device memory. Line 13 is run as a GPU kernel as in the previous listing. Finally, Line 15 copies the result back to the host memory.

3.3 A Basic Work-sharing Idiom Between the CPU(s) and the GPU(s)

In this final example, a basic operation is decomposed in a way that half of the operation is done by the CPU(s) and the other half by the GPU(s). We use a simple increment operation for brevity.

```
var A: [0..#n] int;
// assign half the work to CPU, the rest to GPUs. Assume divisibility
// NOTE: here.getChildCount interface is subject to change
const numGPUs = here.getChildCount()-1;
const cpuSize = n/2;
const gpuSize = (n/2)/numGPUs

// cobegin will spawn two concurrent tasks:
cobegin {  // implicit sync at the end
  // Task 1: Do the CPU part
  A[0..#cpuSize] += 1;
  // Task 2: Iterate over all GPUs, spawning concurrent tasks on each
  coforall subloc in 1..numGPUs {  // implicit sync at the end
```

```
      // NOTE: here.getChild(subloc) interface is subject to change
      on here.getChild(subloc) {
        const myShare = cpuSize+gpuSize*(subloc-1)..#gpuSize;

        // copy to device memory, increment, copy the result back:
        var AonThisGPU = A[myShare];
        AonThisGPU += 1;
        A[myShare] = AonThisGPU;
      }
    }
}
```

Listing 3: A Basic Increment Operation with Work Shared Amongst CPU(s) and GPU(s)

As can be seen from the example, the GPU support can be used in conjunction with the existing data- and task-parallel features such as array slices, bulk array operations, `cobegin` and `coforall` statements to implement complicated operations in an expressive and portable way.

4 LIMITATIONS AND NEXT STEPS

The current status of the implementation is still in the early phases. Key features that we discussed above and are implemented in version 1.26 are as follows.

- GPU sublocales represent GPU-based allocation and execution, and on statements can be used to make allocations and launch kernels on GPU.
- Vectorizable loops and promoted expressions are transformed and launched as GPU kernels.

There are many known limitations and unimplemented features as of version 1.26. We plan to address these in the next several releases before we can call the GPU support ready for production-grade use. Among such limitations are:

- Only NVidia GPUs are supported.
- GPUs are only supported in single-locale builds (i.e., `CHPL_COMM=none`).
- GPU-based distributed arrays as outlined in Section 2.3 are not supported.
- Fine-grained host/device and device-to-device communication is not supported.
- GPU support implicitly enables the flag `--no-checks`, which is the default with `--fast`. This implies that bounds checking is disabled and can result in segmentation faults instead of user-facing errors.

Besides addressing these implementation limitations, we are also developing the finer details of the design. To that end, we are in process of revising the interface for locales to make GPU programming more intuitive and portable [7]. We are also considering extensions to `forall` and `foreach` capabilities to support SIMT programming better [8].

Finally, we are actively investigating existing Chapel applications to determine the key features that they can benefit from. We believe this can help discover other limitations and prioritize what matters most for Chapel users in terms of GPU support.

REFERENCES

[1] https://github.com/chapel-lang/chapel/blob/main/doc/rst/developer/chips/17.rst
[2] https://github.com/chapel-lang/chapel/blob/main/doc/rst/developer/chips/22.rst
[3] https://github.com/rocmarchive/chapel/tree/chpl-hsa-master
[4] Akihiro Hayashi, Sri Raj Paul, and Vivek Sarkar. 2019. GPUIterator: bridging the gap between Chapel and GPU platforms. In Proceedings of the ACM SIGPLAN 6th on Chapel Implementers and Users Workshop (CHIUW 2019). Association for Computing Machinery, New York, NY, USA, 2–11. DOI:https://doi.org/10.1145/3329722.3330142
[5] 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.
[6] https://chapel-lang.org/docs/master/technotes/gpu.html
[7] https://github.com/chapel-lang/chapel/issues/18529
[8] https://github.com/chapel-lang/chapel/issues/16405