Initial Experiences in Porting a GPU Graph Analysis Workload to Chapel

Paul Sathre
Computer Science
Virginia Tech
Blacksburg, Virginia, USA
sath6220@cs.vt.edu

Atharva Gondhalekar
Electrical & Computer Engg.
Virginia Tech
Blacksburg, Virginia, USA
atharva1@vt.edu

Wu-chun Feng
Computer Science
Electrical & Computer Engg.
Virginia Tech
Blacksburg, Virginia, USA
feng@cs.vt.edu

EXTENDED ABSTRACT

We present our initial experiences in porting a GPU graph-analysis workload to Chapel. This endeavor is part of a broader study to characterize the performance-productivity tradeoffs of Chapel's new native support for compiling loops for GPU execution. We discuss the motivation for migrating to Chapel, provide an introduction to our proxy workload known as *edge-connected Jaccard similarity*, and briefly discuss code migration issues and preliminary performance observations.

Introduction

The plateauing of frequency scaling and the increasing commonality of computationally-intensive workloads, such as machine learning and big-data analysis, have driven an explosion of both multi- and many-core parallelism, not just in the datacenter but also under the desk, at home, and on the go. Most cell phones are not only multi-core, but include additional accelerator engines for different workloads including graphics and digital signal processing. Even the smallest, $15 USD Raspberry Pi micro-computer is a quad-core! [2]. Laptops, desktops, workstations, and servers have been multi-core with various available accelerator engines for decades.

Such pervasive parallelism suggests a need to rethink how we write new software, maintain old software, and teach new programmers. At present, the most popular programming languages, according to the 2022 IEEE Spectrum analysis [7], are Python, C, C++, C#, and Java, none of which include fundamental tokens to express parallelism natively. In the high-performance computing (HPC) community, the predominant languages pre-date ubiquitous parallelism, namely C, C++ and Fortran. However, they do support opt-in parallelism through some combination of libraries, pragmas, extensions, or third-party frameworks.

Given the ubiquity of parallelism in computing, it behooves us to start using natively parallel languages, where parallelism is *not* an optional feature (e.g., library / pragma / extension) but rather a fundamental *language token.* A fundamental element is available from *day zero* of using a language, not a separate element gate-kept by a complicated installation or a tome of documentation, but as adjacent as simply adding "`all`" or "`each`" to your existing `for` loops. Natively parallel languages like Chapel provide this opportunity to re-envision the development process with parallelism as a first-class citizen, instead of something tacked on later.

GPU Programming

Even with natively parallel languages, there are limits to the performance achievable with traditional multi-core CPUs, which deliver performance through instruction-level pipelining (ILP). speculative execution, and modest parallelism. Thus, accelerators, whether as discrete chips or integrated with the CPU, have surged in popularity. In particular, general-purpose GPU (GPGPU) computations have become commonplace in both datacenter- and user-scale workloads.

However, with special-purpose accelerators comes the question of how to program them. Shader languages for GPUs have existed for decades, but expressing computational algorithms in them is cumbersome. In the late 2000s, Nvidia introduced the CUDA language extensions and toolkit [1] for C and Fortran, which has garnered dominant market share over the subsequent years. The OpenCL standard [18] emerged shortly after, to provide a *portable* abstraction across accelerator vendors and platforms, i.e., not just GPU but also CPU, FPGA, and others. In addition, there have been attempts to extend pragma-based parallelism to accelerators, such as OpenMP 4+ [4] and OpenACC [3]. More recently, the SYCL standard has emerged to provide a *portable*, modern C++-based abstraction for parallel computations. However, these are fundamentally *add-ons* to existing *natively serial* languages. Broadly speaking, one does not encounter parallel language elements, like collaborative thread blocks, until you get inside the GPU *kernel*, well *after* investing manual effort to install the SDK, select and initialize a device, allocate memory on it, and stage data into its memory.

GPUs in Chapel. In contrast, what could GPU computing look like within a language that is already natively parallel? Chapel has strong C interoperability, so one approach leverages CPU-side wrappers around traditional C/C++-based GPGPU languages and APIs [12–14]. However, this requires departing the Chapel ecosystem to install and learn the lower-level languages, discarding many of the productivity benefits of Chapel. Thus, we focus on efforts to support native Chapel codes' execution on GPUs.

One of the earliest attempts to demonstrate CUDA code generation from Chapel [21] dates as far back as 2012. More recently, AMD demonstrated efforts to support Chapel execution for their GPUs via the ROCm compute layer and Chapel-to-OpenCL conversion [9]. An independent effort also generated OpenCL from Chapel for just-in-time compilation in support of a *GPUArrays* library [11].

Chapel introduced native GPU support in version ~1.26 [17] (with some features available earlier). Chapel's GPU support has continued to improve with each subsequent release. More recently, Chapel 1.30 delivered further performance improvements, resulting in nearly identical performance to CUDA for some workloads [8]. This work is primarily based on the same 1.30 release.

Graph Workload

To evaluate the efficacy of Chapel 1.30's native GPU support, we consider a graph analysis workload called *edge-connected* Jaccard similarity (JS), which serves as proxy for more complicated, neighbor-intersection algorithms.

Jaccard similarity relates the intersection of two sets to their union, in order to measure the *sharedness* of the two sets [15]. That is, for sets A and B, their JS is given by the equation $JS(A, B) = \frac{intersect(A,B)}{union(A,B)}$. A JS of 1 means the two sets are identical, whereas 0 means the two sets are disjoint, and anything in between signifies partial overlap. When applied to graphs, we consider a batch of sets which consist of the 1-hop *neighborhoods* of individual vertices. Figure 1 provides a sketch of the relevant components on a trivial graph. This formulation gives $|vertices|$ neighbor sets, from which one can compute the JS of different pairs of vertices depending on the end goal.

- List-based methods can be useful for querying the relatedness of an arbitrary subset of vertex pairs.
- All-pairs could be used to infer *new* edges between vertices with highly-similar neighborhoods, which could be leveraged for iterative community building.
- The *edge-connected* variant we are interested in computes JS for all pairs of vertices that are connected by a 1-hop edge, which can be used for considering the *strength* of existing connections or synthesizing edge weights for subsequent algorithms.[1]

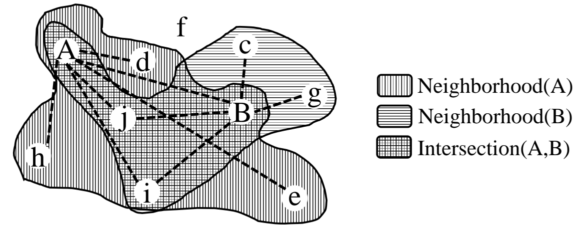

Figure 1: Components of the Edge-Connected Jaccard similarity

Graphs encompass a major modern workload underlying artificial intelligence and data analysis pipelines and typically contain significant latent parallelism. However, it is a challenge to express and exploit that parallelism to improve workload performance. Currently, parallel graph analysis is often performed through library frameworks, which serve a valuable functional purpose but are rigid and difficult to adapt if your algorithmic or hardware needs diverge from what is provided.

Chapel is a good candidate for a higher-level graph processing language as it is both natively parallel and natively distributable. Cluster- and datacenter-scaling with Chapel is much easier than augmenting a historically-serial language like C/C++ or Python with *both* an intra-node parallelism approach (CUDA, OpenMP, etc.) *and* an inter-node distribution approach (MPI, sockets, etc.). Chapel has the performance benefits of static compilation with intuitive distributable array abstractions and *fundamental tokens* for expressing parallelism and concurrency/distribution – `foreach/forall` and `coforeach/coforall`, respectively. Further, Chapel's built-in GPU support leverages *the same* parallel abstractions that already exist in the language, which makes introducing heterogeneous parallelism (CPU + GPU) easier than tacking on new libraries and toolchains from traditional GPU programming paradigms like CUDA, OpenCL, HIP, or SYCL.

Porting to Chapel

To consider Chapel's efficacy as a high-performance GPU graph analysis language, we need to evaluate its productivity benefits against any potential GPU performance implications. As a first step, towards this evaluation we manually port the CUDA precursor of our SYCL implementation [20]. This precursor includes a bespoke embarrassingly-parallel *edge-centric* (EC) implementation and an isolated fork of the *legacy* `cuGraph` implementation [5, 10]. The isolated

[1]The edge-connected formulation can also derive a *triangle count* as a byproduct, as any shared neighbor (non-zero numerator) necessarily implies edges from the two endpoint vertices to the shared neighbor.

cuGraph pipeline is predominantly *vertex-centric* (VC) and leverages CUDA's 2D and 3D kernel parallelism to collaboratively search for pairwise intersections.[2]

While other formulations of Jaccard similarity, such as sparse matrix multiplication [6], have been shown to be GPU- and distribution-amenable, we utilize our direct implementation for its familiarity and algorithmic transparency, which will allow us to perform productivity and performance comparisons against a known implementation. Provided our initial foray into GPU-enabled Chapel demonstrates that reasonable productivity and performance is achievable, then improving wall-clock performance and scalability through profiler-driven optimization and/or algorithmic adaptation or refactorization remains as future work.

GPU-ization. We have fully ported both kernel pipelines to Chapel and found the GPU sub-locale abstraction and `forall` intuitive for expressing embarrassingly parallel portions of the workload. The EC pipeline's 1D embarrassingly parallel kernels "GPU-ize" straightforwardly to Chapel 1.30 In contrast, we encountered numerous roadblocks when GPU-izing the VC pipeline. First, the intersection kernel utilizes a 3D `grid/block` configuration, mapping the Z-dimension to source vertices, Y-dimension to destination vertices, and X-dimension to a collaborative binary search for pairwise intersections. Our original Chapel implementation utilized nested `forall` statements to map each of the original CUDA grid/block dimensions, but because Chapel 1.30 supports only 1D kernels, this nesting of `forall` statements forced the compiler to map to CPU. We worked around this by manually linearizing the thread index space, which, in turn, required a patch to the Chapel runtime system to support ultra-high GPU thread counts, which HPE rapidly implemented and integrated upstream [16].

Additionally, the original CUDA uses intra-thread loops in each dimension to support scaling to arbitrary graph and neighbor-list sizes in excess of the configured `grid/block`. Due to un-GPU-izable safety checks behind the scenes of Chapel's optional `by` clause to specify the `forall`'s stride, we replaced these inner loops with equivalent manually-counted `while` loops. These two changes provided significant performance gain to the transparently CPU-mapped VC intersection but still did not allow GPU-ization.

Finally, the collaborative X-dimension intersection search within the VC intersection kernel needs atomic writes to a global intersection array – but *non-atomic* reads in a later kernel. Chapel's native *type-qualified* atomics are not yet GPU-izable, but with guidance from HPE we leveraged Chapel's C interoperability to directly call CUDA's native *access-qualified* atomics from within the `forall`. This provided the last key to unlocking GPU-ization of the VC pipeline. We were pleased that throughout this process that Chapel was able to transparently map the GPU-intended `forall` to the CPU, supporting validation of their *functional* portability,[3] while we worked to remove barriers to GPU-ization.

Host Code. We also ported the mini-application that wraps the two kernel pipelines and utilizes a custom binary compressed sparse row (CSR) file format. The format supports variable-width vertex/edge indices and Jaccard weight array members (32- vs. 64-bit), described via a fixed-sized header. Implementing equivalent file I/O in Chapel actually required more effort than the kernels due to this need to mix runtime-defined type widths with a statically-compiled parameterized CSR class representation.

The format's fixed-size header uses a C++ bitfield of flags describing the member arrays' element widths and other properties. As of 1.30, Chapel lacks a direct analog to C++'s bitfield, but we emulated that behavior by reading the bitfield as a 64-bit integer, itemizing its members in an enumerator and converting to/from a runtime *type descriptor* record via binary AND/OR, respectively. A sketch of this is provided in Fig. 2a. These conversions were encapsulated within class copy initializers and I/O methods, which ameliorated the additional code verbosity. The resulting runtime type descriptor is used to create a *concrete* instance of a generic CSR class that is parameterized with respect to the member arrays' element widths, shown in Fig. 2b.

However, as Chapel is statically typed, we needed to generate all possible instantiations of the generic CSR class and supporting generic functions at compile time in order to support any widths that might be read at runtime. We achieved this via a "ladder" of overloaded functions that incrementally translate runtime flags to the correct compile-time concrete specializations, as shown in Figure 2d.

Finally, in order to pass CSR instances through code regions that *could not know* their concrete type at compile time, we implemented a C-like opaque handle type. This type, shown in Fig. 2c, contains the runtime type descriptor, used to select the correct specialized functions, and a C void pointer to the concrete instance. Void pointers are excluded from the Chapel language *by design*, but usable through Chapel's C interoperability module. Combined, these approaches achieved a rudimentary explicit form of runtime type information (RTTI).

[2]The master branch of CuGraph has evolved since our fork – at version ~0.18, commit SHA `3f13ffcdf`– and we make no claim of commonality with or against any current implementation(s) of JS that it may provide.

[3]We consider *functional* portability to be the minimum baseline of producing semantically-correct results *without regard for achieved wall-clock performance Performance* portability is the harder goal of semantically-correct results *with similar wall-clock performance / achieved percentage of peak.*

```
1 enum CSR_header_flags {
2   isWeighted= 1 << 0,
3   ... // other flags
4   isWeightT64= 1 << 6,
5 };
6 private param CSR_BIN_FMT_VER: int(64)= 2;
7 record CSR_file_header { //disk format
8   var binFmtVer: int(64)= CSR_BIN_FMT_VER;
9   var numVerts: int(64)= 0;
10  var numEdges: int(64)= 0;
11  var flags: int(64)= 0; //can't binOr Chapel enums
12  proc init=(rhs: CSR_descriptor) {
13    this.numVerts= rhs.numVerts;
14    this.numEdges= rhs.numEdges;
15    ... // binORs to convert bools to flags int
16  }
17  ... // other req'd operators and casts
18 }
19 record CSR_descriptor {   //Runtime type descriptor
20  var isWeighted: bool= false;
21  ... // other flags
22  var isWeightT64: bool= false;
23  var numEdges: int(64)= 0;
24  var numVerts: int(64)= 0;
25  proc init=(rhs: CSR_file_header) {
26    assert(rhs.binFmtVer == CSR_BIN_FMT_VER, ``msg'');
27    ... // binANDs to convert the flags int to bools
28    this.numEdges= rhs.numEdges;
29    this.numVerts= rhs.numVerts;
30  }
31  ... //other req'd operators and casts
32 }
```

(a) Chapel enumerator and CSR *type-descriptor* to emulate a C++ bitfield and provide runtime type information

```
1 class CSR {
2   var numEdges: int(64);
3   var numVerts: int(64);
4   param isWeighted: bool;
5   ... // other flags
6   param isWeightT64: bool;
7   //CSR element arrays
8   var iDom: domain(1)= {0..numEdges-1};
9   var indices: [iDom] int(if isVertexT64 then 64 else 32);
10  var oDom: domain(1)= {0..(numVerts)};
11  var offsets: [oDom] int(if isEdgeT64 then 64 else 32);
12  var wDom: domain(1)= {0..(if isWeighted then numEdges-1 else 0)
    ↪ }; //Degenerate w/o weights
13  var weights: [wDom] real(if isWeightT64 then 64 else 32);
14  proc getDescriptor(): CSR_descriptor {
15    ... //runtime copy of param flags
16  }
17  ... //reader/writer funcs
18 }
```

(b) Chapel parameterized generic CSR type

```
1 record CSR_handle { //Opaque type
2   var desc : CSR_descriptor;
3   var data : c_void_ptr;
4   ... // reader/writer funcs
5 }
```

(c) Chapel CSR handle *opaque* type

```
1 proc NewCSRHandle(type CSR_type : CSR(?), in desc : CSR_descriptor
  ↪ ): CSR_handle {
2   assert( ...); //that desc matches CSR_type
3   var retHandle : CSR_handle;
4   local { // avoid issues w/ GPU-req'd wide pointers
5     var retCSR = new unmanaged CSR_type(...);
6     ... //Assign all the non-param, non-array fields
7     var retCast = (retCSR : c_void_ptr);
8     retHandle.data = retCast;
9     retHandle.desc = desc;
10  }
11  return retHandle;
12 }
13 //``Ladder'' of progressive runtime-->param resolution
14 //Fully-parameterized final ``rung''
15 private proc NewCSRHandle(in desc : CSR_descriptor, ... ) :CSR_handle {
16  return NewCSRHandle(CSR(isWeighted, isVertexT64, isEdgeT64,
    ↪ isWeightT64), desc);
17 }
18 ... // intermediate rungs for vertex, edge, and weight widths
19 //entrypoint rung for deducing weight presence
20 proc MakeCSR(in desc : CSR_descriptor) :CSR_handle {
21  return (if desc.isWeighted then MakeCSR(desc, true) else MakeCSR
    ↪ (desc, false));
22 }
23 //used inside funcs that know the concrete type
24 proc ReinterpretCSRHandle(type CSR_type: unmanaged CSR(?), in
  ↪ handle : CSR_handle) : CSR_type {
25  ... // cast handle.data back to CSR_type w/ type checking
26 }
```

(d) Operations on CSR opaque type.

Figure 2: Supporting a flexible CSR binary format in Chapel with C-style explicit runtime type information

Preliminary Results

We measured the performance of both JS pipelines on 18 graph workloads, the same set as our prior SYCL work [19, 20]. They cover a range of sizes and sparsities but all fit in the VRAM of our test GPU, a Nvidia RTX 3090 consumer-grade card, supported by CUDA 11.6, driver 510.108.03. The GPU kernels, both CUDA and Chapel GPU-ized, were measured using Nvidia Nsight Compute for driver-level profiling. Un-GPU-izable intermediate kernels from the VC pipeline were measured using Chapel's `stopwatch` timers around `foralls` when executed on the CPU. We considered both Chapel 1.30's default `unified_memory` (GPU-accessible, CPU-resident) and CUDA-like `array_on_device` (GPU-resident) memory strategies, but there is enough inherent indirect access latency that unified memory does not expose significant slowdown. The CUDA code is exclusively GPU-resident.

Preliminary results indicate the EC pipeline performs strongly and has a high degree of *performance* portability. On some sparser graphs CUDA significantly outperforms Chapel; but as the density increases, Chapel approaches *performance parity*. Unexpectedly, there are a few graphs where Chapel outperforms CUDA, something for future study.

The partially GPU-ized VC pipeline is expectedly underwhelming due to the CPU mapping of the intersection and final Jaccard weight kernels. However, once fully GPU-ized, the VC pipeline's performance was similar to EC's: Chapel slightly underperforms CUDA on the sparsest data but approaches performance parity at the denser end and even outperforms CUDA on one dataset!

Future Directions

We plan to continue working to understand and close performance gaps between GPU-enabled Chapel and CUDA using a profiler-driven optimization approach and to feed insights from that process back to the Chapel development team and community. Chapel 1.30 introduced preliminary AMD GPU support, which would allow us to explore tradeoffs on their hardware compared to HIP. We also plan to use our existing SYCL implementation from [20] to directly compare vendor-agnostic languages on both GPU and CPU. Finally, we intend to investigate scalability to multiple GPUs/nodes using Chapel's built-in support for distributed computing and insights from existing literature.

CODE

`https://github.com/vtsynergy/Chapel-Examples`

ACKNOWLEDGEMENTS

The work detailed herein has been supported in part by NSF I/UCRC CNS-1822080 via the NSF Center for Space, High-performance, and Resilient Computing (SHREC).

REFERENCES

[1] [n. d.]. CUDA Toolkit. https://developer.nvidia.com/cuda-toolkit

[2] [n. d.]. Raspberry Pi Zero 2 W. https://www.raspberrypi.com/products/raspberry-pi-zero-2-w/

[3] 2011. The OpenACC(TM) Application Programming Interface. https://www.openacc.org/sites/default/files/inline-files/OpenACC_1_0_specification.pdf

[4] 2013. OpenMP Application Program Interface. https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf

[5] RapidsAI 2023. *Legacy Jaccard Similarity Implementation from CuGraph*. RapidsAI. https://github.com/rapidsai/cugraph/blob/3f13ffcdf2b272fd6aba99eb529ff55c2a43d5d1/cpp/src/link_prediction/jaccard.cu

[6] Maciej Besta, Raghavendra Kanakagiri, Harun Mustafa, Mikhail Karasikov, Gunnar Rätsch, Torsten Hoefler, and Edgar Solomonik. 2020. Communication-Efficient Jaccard similarity for High-Performance Distributed Genome Comparisons. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 1122–1132. https://doi.org/10.1109/IPDPS47924.2020.00118

[7] Stephen Cass. 2022. Top Programming Languages 2022. Blog. https://spectrum.ieee.org/top-programming-languages-2022

[8] Brad Chamberlain. 2023. Announcing Chapel 1.30.0! Blog. https://chapel-lang.org/blog/posts/announcing-chapel-1.30/

[9] Michael L Chu, Ashwin M Aji, Daniel Lowell, and Khaled Hamidouche. [n. d.]. GPGPU support in Chapel with the Radeon Open Compute Platform. ([n. d.]). https://chapel-lang.org/CHIUW/2017/chu-abstract.pdf

[10] Alexandre Fender, Nahid Emad, Serge Petiton, Joe Eaton, and Maxim Naumov. 2017. Parallel Jaccard and Related Graph Clustering Techniques. In *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems* (Denver, Colorado) *(ScalA '17)*. Association for Computing Machinery, New York, NY, USA, Article 4, 8 pages. https://doi.org/10.1145/3148226.3148231

[11] Rahul Ghangas and Josh Milthorpe. 2020. Chapel on Accelerators. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 679–679. https://doi.org/10.1109/IPDPSW50202.2020.00121

[12] 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* (Phoenix, AZ, USA) *(CHIUW 2019)*. Association for Computing Machinery, New York, NY, USA, 2–11. https://doi.org/10.1145/3329722.3330142

[13] Akihiro Hayashi, Sri Raj Paul, and Vivek Sarkar. 2021. GPUAPI: Multi-level Chapel Runtime API for GPUs. In *CHIUW '21: The 8thAnnual Chapel Implementers and Users Workshop*. ACM, 9. https://chapel-lang.org/CHIUW/2021/Hayashi.pdf

[14] Akihiro Hayashi, Sri Raj Paul, and Vivek 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

[15] Paul Jaccard. 1912. THE DISTRIBUTION OF THE FLORA IN THE ALPINE ZONE.1. *New Phytologist* 11, 2 (1912), 37–50. https://doi.org/10.1111/j.1469-8137.1912.tb05611.x arXiv:https://nph.onlinelibrary.wiley.com/doi/pdf/10.1111/j.1469-8137.1912.tb05611.x

[16] Engin Kayraklioglu. [n. d.]. Selectively use 64-bit ints for GPU kernel index computation. Github. https://github.com/chapel-lang/chapel/pull/22259

[17] Engin Kayraklioglu, Andy Stone, David Iten, Sarah Nguyen, Michael Ferguson, and Michelle Strout. [n. d.]. Targeting GPUs Using Chapel's Locality and Parallelism Features. ([n. d.]).

[18] Khronos OpenCL Working Group. [n. d.]. The OpenCL Specification. https://registry.khronos.org/SYCL/specs/sycl-2020/pdf/sycl-2020.pdf

[19] Paul Sathre. 2022. Datasets for SYCL-Jaccard. https://chrec.cs.vt.edu/SYCL-Jaccard/HPEC22-Data/index.html

[20] Paul Sathre, Atharva Gondhalekar, and Wu-chun Feng. 2022. Edge-Connected Jaccard Similarity for Graph Link Prediction on FPGA. In *2022 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–10. https://doi.org/10.1109/HPEC55821.2022.9926326

[21] Albert Sidelnik, Saeed Maleki, Bradford L. Chamberlain, María J. Garzar'n, and David Padua. 2012. Performance Portability with the Chapel Language. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. 582–594. https://doi.org/10.1109/IPDPS.2012.60