Accelerating Data Analytics with Arkouda on GPUs

Josh Milthorpe
Brett Eiffert
Jeffrey S. Vetter
Oak Ridge National Laboratory
Oak Ridge, TN, USA

1 INTRODUCTION

Arkouda [7] is a framework for large-scale interactive data analytics that combines a Python front-end with a distributed server implemented using Chapel to run on parallel architectures ranging in size from a single node to an entire high-performance computing system. While Arkouda is capable of exploiting traditional high-performance compute capabilities, it is currently unable to use the powerful GPU accelerators available in many HPC systems.

In this talk, we will demonstrate how the Chapel GPUAPI can be used to accelerate Arkouda operations, which is most beneficial when a chain of operations is executed on the same data. We extend the GPUAPI to support shared virtual memory using CUDA unified memory and use this support to implement a custom domain map for Arkouda arrays. Our preliminary performance results show that GPU-accelerated operations in unified memory perform better than explicit memory management while simplifying the programming task for complex Arkouda operations.

2 ACCELERATING ARKOUDA OPERATIONS USING CHAPEL GPUAPI

The Chapel GPUAPI [5] provides abstractions over GPU programming models, including CUDA, HIP, DPC++, and SYCL. The included `GPUArray` class supports explicit allocation of device memory and transfer of data to/from device, while the `GPUIterator` supports launching kernels implemented in a GPU-native programming model (e.g. CUDA) on one or more local devices using a Chapel data-parallel **`forall`** loop. We used the GPUAPI for the NVIDIA architecture to implement Arkouda operations using optimized CUDA implementations from the CUB [2] and NCCL [3] libraries. For sort, we used the Onesweep algorithm for least-significant-digit radix sort implemented in NVIDIA's CUB library [1]. For merging the individual sorted GPU chunks, we compare two algorithms: 1) transferring back to host memory and merging on the host, and 2) a peer-to-peer merge-and-swap algorithm proposed by Tanasic et al. [9] and extended for an arbitrary number of devices by Maltenberger et al. [6]. As a design principle, we maintain the existing Python client interface to Arkouda and focus on accelerating the server-side (Chapel) implementation of each operation.

2.1 Device Cache: Explicit Memory Management for Arkouda Arrays

Given a suite of GPU-accelerated versions of Arkouda functions that can operate on data in GPU device buffers, we require a method to move the operands for these functions (Arkouda arrays) to and from the GPU. To support explicit memory management, we extended Arkouda's `SymEntry` class by adding a *device cache*, which is a set of `GPUArray` instances that distribute an Arkouda array in the available device memories. The device cache includes operations to transfer data to and from devices, which may be called before and after kernel execution.

Figure 1: API for the Arkouda `SymEntry DeviceCache`

```
class DeviceCache {
  var isCurrent = false;
  var deviceArrays: [gpuDevices] shared GPUArray?;

  proc createDeviceArrays(a: [?aD] ?etype) { ... }
  proc toDevice(deviceId: int(32)) { ... }
  proc fromDevice(deviceId: int(32)) { ... }
}
```

The device cache tracks the allocation, update, and transfer of data between device and host, meaning that a call to transfer data to a device/host that already has the most up-to-date version does not result in any data movement. This enables each Arkouda server operation to be programmed to ensure data are in place before kernel execution, while also allowing multiple operations on the same data to be chained without the need for host-device transfers.

2.2 Arkouda Arrays in Shared Virtual Memory

While explicit management of memory transfers through the GPUAPI can support efficient utilization of resources, it adds to the programming effort for developers of server operations, who need to define transfer points from host to device(s) before each accelerated operation, and from device(s) to host for each client-visible operation. Furthermore, in the context of an interactive workflow where the user is operating on multiple Arkouda arrays, it can be hard to predict access patterns so as to optimize use of device memory. An alternative approach is to use *shared virtual memory* (also known as unified [4] or managed memory), which is a shared address space that is accessible from both host and device. For NVIDIA GPUs with Pascal and later architectures, accesses to shared virtual memory are handled by a hardware Page Migration Engine, allowing memory to migrate transparently to a device from anywhere in the system, on demand [8].

We propose a new Chapel distribution, `GPUUnifiedDist`, which is a block distribution that allocates each local portion of the distributed array in CUDA managed memory using `cudaMallocManaged`. This distribution replaces the standard `BlockDist` used in the `SymArrayDmap` class which determines the domain map type to use when creating a new Arkouda array represented as an instance of `SymEntry`. With this modification, an Arkouda array in unified memory behaves like an ordinary Arkouda array, except that array elements may be transparently migrated

when they are accessed on a device or on the host. We also propose a new operation `SymEntry.prefetchLocalDataToDevice` that initiates asynchronous prefetching of a region of an Arkouda array to a specified device, to enable overlap of GPU kernel computation with data transfer. Note that it is still necessary to synchronize between device and host, for example, by calling the GPUAPI function `DeviceSynchronize()`, to ensure that all data are consistent before they are migrated.

3 EVALUATION

3.1 Experimental Configuration

Single-node experiments were conducted on an NVIDIA DGX workstation which has two 20-core Intel Xeon E5-2698s at 2.2 GHz clock speed and 256 GiB of DRAM and four Tesla V100 GPUs each with 32 GiB of high-bandwidth memory. For the GPUs, NVHPC toolkit v22.11 (CUDA v11.8) and CUDA driver version 530.30.02 were used.

3.2 Benchmark Operations

3.2.1 Reduction. We measured the time taken for a simple sum reduction over variously-sized arrays of `real(64)` values. We compared Chapel's default sum implementation running on the CPU (which is used by the Arkouda server) against a Chapel GPUAPI implementation using the `DeviceReduce::Sum` function from NVIDIA's CUB library that combines partial sums in a GPU collective communication using `ncclReduce`. For the GPU implementation, we also compared the performance where the Arkouda array is allocated in unified memory using `GPUUnifiedDist` against a version using explicit memory management via `GPUArray`.

As shown in Figure 2, GPU reduction using unified memory is approximately 50% faster than explicit memory management with `GPUArray` for large data sizes. This may be because kernel execution can begin before prefetching of data from host to device is complete. A CPU-based reduction in host memory is fastest for all problem sizes when the costs of data transfer to and from GPU devices are included. However, considering only the kernel time for the sum operation and collective reduction (discounting data transfer to device), the GPU kernel is faster than the CPU time. This suggests that there may be some benefit to offloading even a simple reduction operation if it is part of a chain of operations on the same data, as that would allow the cost of data transfer to be amortized over multiple operations.

3.2.2 Histogram. To evaluate a more complex map-reduce operation, we measured the time taken to compute a histogram over variously sized arrays of `real(64)` values. We compared Arkouda's histogram implementation for the CPU against a GPU implementation using the `DeviceHistogram::HistogramEven` function from NVIDIA's CUB library that combines partial histograms in a GPU collective communication using `ncclAllReduce`.

As shown in Figure 3, Arkouda's CPU histogram implementation is faster for small data sizes ($<10^6$), while the GPU is faster for larger sizes. For the GPU implementation, unified memory was approximately 45% faster than explicit memory management using `GPUArray` for larger data sizes.

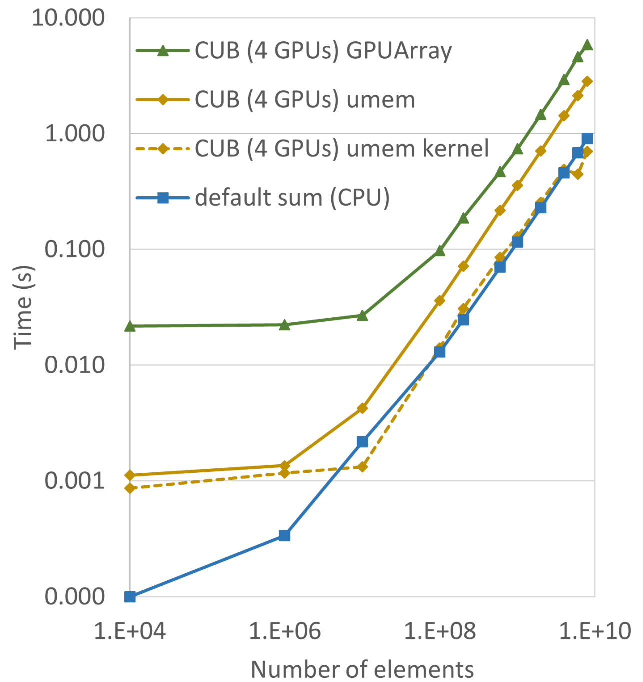

Figure 2: Performance of reduction using multi-GPU CUB/NCCL vs. default Chapel reduction on host.

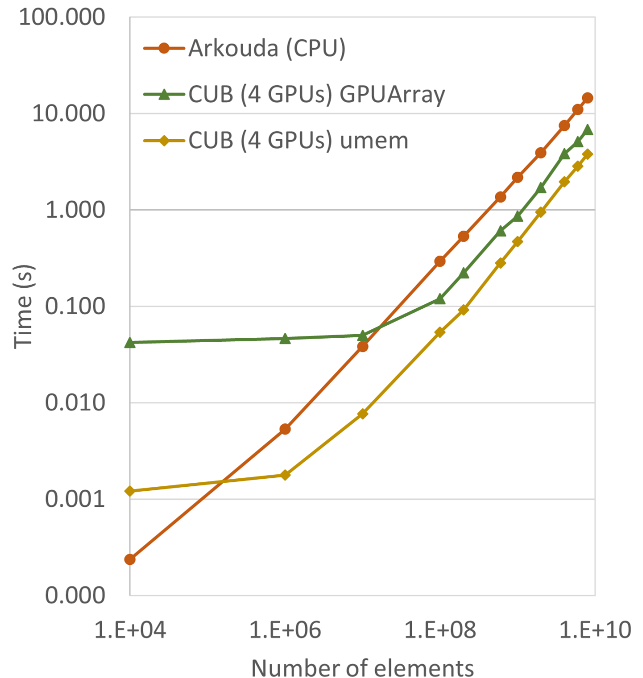

Figure 3: Performance of histogram using multi-GPU CUB/NCCL vs. Arkouda histogram on host.

3.2.3 Sort. We also measured the time taken to sort variously sized arrays of `real(64)` values. We compared Arkouda's sort implementation against the default Chapel sort and two GPU unified-memory implementations using the `DeviceRadixSort::SortKeys` function from NVIDIA's CUB library. The time for each GPU implementation includes the transfer of the sorted data from GPU back to host.

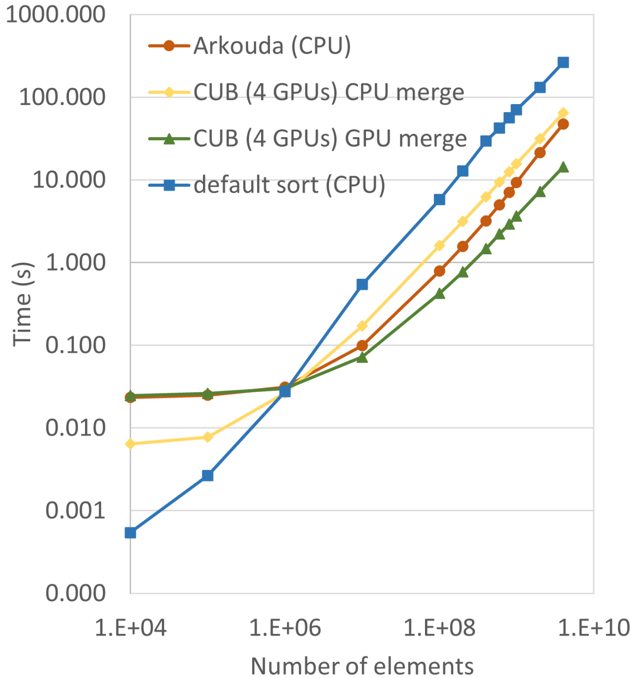

Figure 4: Performance of Arkouda sort using GPU unified memory (merging either on GPU or on CPU) vs. Arkouda sort on CPU and default Chapel sort on host.

As shown in Figure 4, the default Chapel CPU sort is fastest for small data sizes ($<10^6$), while Arkouda's sort implementation is faster for larger sizes. We also compared a pure-GPU sort implementation, where sorted chunks are merged via Maltenberger et al. [6]'s peer-to-peer swap and merge algorithm, against a heterogeneous algorithm where sorted chunks are copied back from host to device for the final merge. Merging on the GPU provides approximately a 75% performance improvement compared to merging on the host for array sizes $\geq 10^8$, and is 70% faster than Arkouda's sort for the largest data size (4×10^9).

3.2.4 Chained Operations. Finally, we measured the end-to-end time taken to perform a sequence of Arkouda operations (histogram, sum, min, max) on the same Arkouda array, which remains resident in GPU memory between operations. We compared the default Arkouda CPU implementations against GPU implementations using CUB/NCCL and either `GPUArray` or unified memory with `GPUUnifiedDist`, for variously-sized arrays of `real(64)` values.

As shown in Figure 5, a chain of operations performed on GPU is significantly faster than CPU. The unified memory implementation

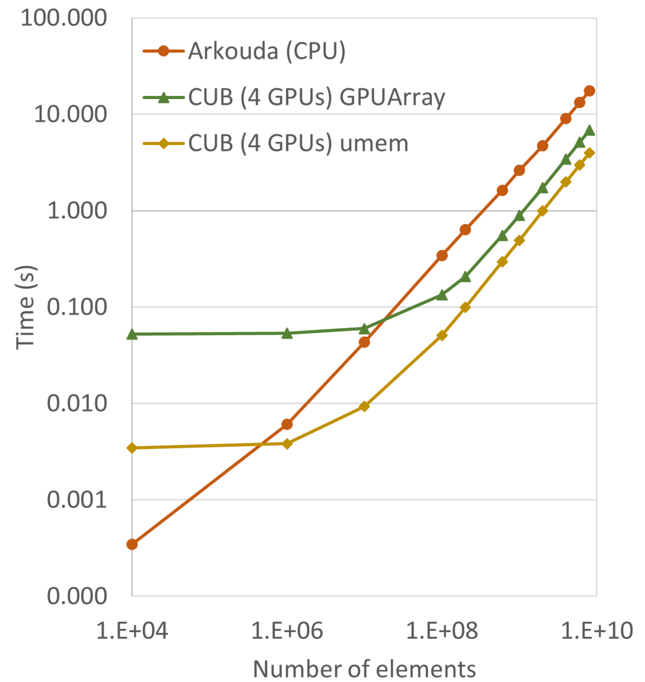

Figure 5: Performance of a chain of Arkouda operations using multi-GPU CUB/NCCL vs. CPU.

is approximately 40% faster than the `GPUArray` implementation for larger problem sizes ($> 10^9$).

This chained operation benchmark demonstrates the potential performance benefit of GPU acceleration for Arkouda operations, assuming that unnecessary data movement can be avoided. In this talk we hope to present preliminary performance results for multi-node (multi-locale), multi-GPU accelerated operations over Arkouda arrays using unified memory.

Acknowledgment

This research used resources of the Experimental Computing Laboratory (ExCL) at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

REFERENCES

[1] Andy Adinets and Duane Merrill. 2022. Onesweep: A Faster Least Significant Digit Radix Sort for GPUs. https://doi.org/10.48550/arXiv.2206.01784 arXiv:2206.01784 [cs.DC]
[2] NVIDIA Corporation. 2023. CUB: Cooperative primitives for CUDA C++. https://github.com/NVIDIA/cub
[3] NVIDIA Corporation. 2023. NVIDIA Collective Communication Library (NCCL). https://developer.nvidia.com/nccl Accessed: 2023-04-16.
[4] Mark Harris. 2013. Unified Memory in CUDA 6. https://developer.nvidia.com/blog/unified-memory-in-cuda-6/
[5] Akihiro Hayashi, Sri Raj Paul, and Vivek Sarkar. 2022. A Multi-Level Platform-Independent GPU API for High-Level Programming Models. In *ISC High Performance 2022 Workshops*. https://doi.org/10.1007/978-3-031-23220-6_7
[6] Tobias Maltenberger, Ivan Ilic, Ilin Tolovski, and Tilmann Rabl. 2022. Evaluating multi-GPU sorting with modern interconnects. In *International Conference on Management of Data*. 1795–1809. https://doi.org/10.1145/3514221.3517842

[7] Michael Merrill, William Reus, and Timothy Neumann. 2019. Arkouda: interactive data exploration backed by Chapel. In *ACM SIGPLAN 6th Chapel Implementers and Users Workshop*. 28–28. https://doi.org/10.1145/3329722.3330148

[8] Nikolay Sakharnykh. 2016. Beyond GPU Memory Limits with Unified Memory on Pascal. https://developer.nvidia.com/blog/beyond-gpu-memory-limits-unified-memory-pascal/

[9] Ivan Tanasic, Lluís Vilanova, Marc Jordà, Javier Cabezas, Isaac Gelado, Nacho Navarro, and Wen-Mei Hwu. 2013. Comparison based sorting for systems with multiple GPUs. In *6th Workshop on General Purpose Processor Using Graphics Processing Units*. https://doi.org/10.1145/2458523.2458524