Braiding a Million Threads: Scalable GPU Sort on Frontier

ChapelCon '24

7 June 2024

Josh Milthorpe, **Brett Eiffert**, Jeffrey S. Vetter

ORNL Advanced Computing Systems Research

ORNL is managed by UT-Battelle LLC for the US Department of Energy

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.

Path of Research

- Transition from Summit to Frontier
- Georgia Tech GPU API – Akihiro Hiyashi
- Performance on Frontier
- Chapel GPU Code Generation

This research used resources of the Oak Ridge Leadership Computing Facility (OLCF) and 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.

Accelerating Arkouda with GPUs

https://github.com/Bears-R-Us/arkouda

- Arkouda promises 'HPC-enabled exploratory data analytics'
- Compute on large data → memory bandwidth

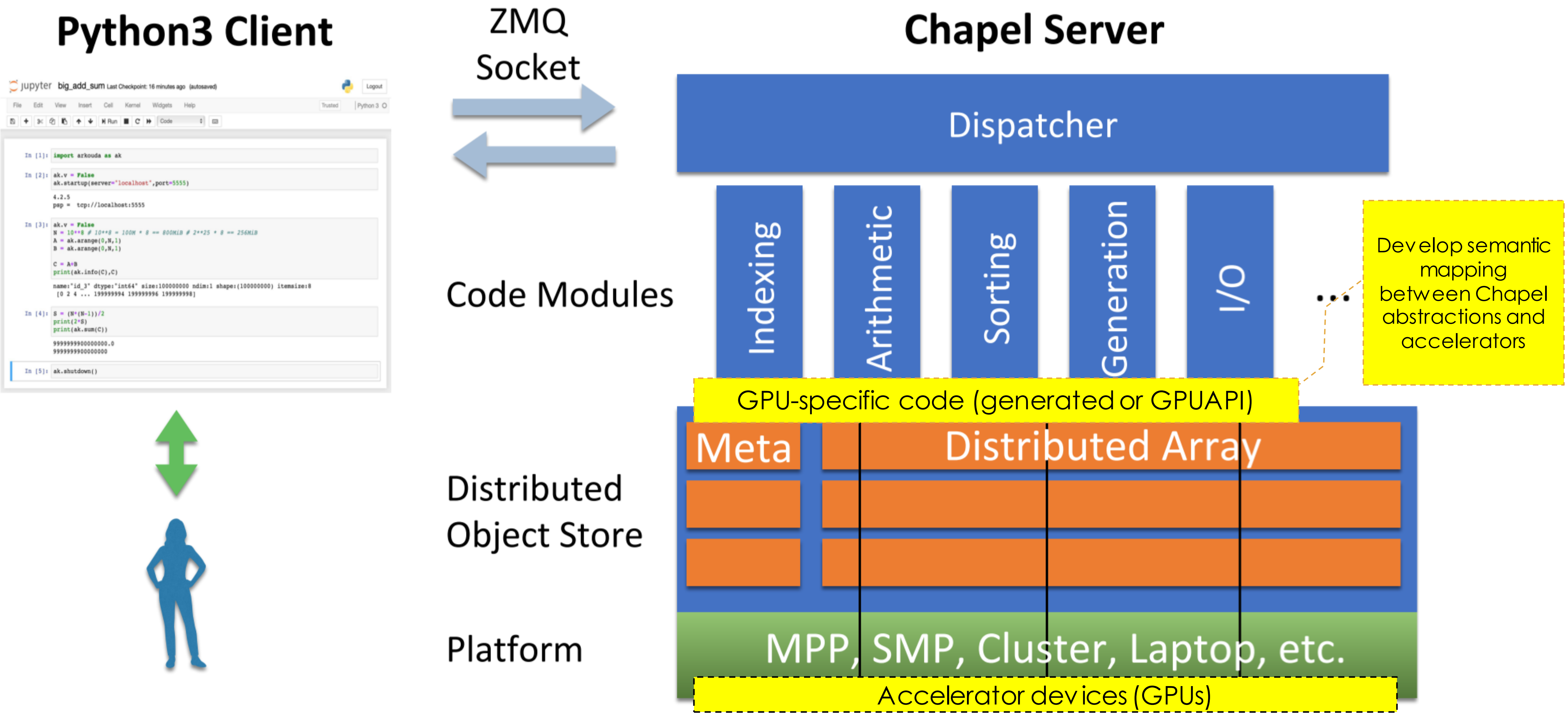
	CPU-DRAM	GPU-HBM
Summit (2018)	340 GB/s	2,700 GB/s
Frontier (2022)	205 GB/s	13,080 GB/s

- Challenges:
  - algorithmic portability
  - memory management
  - programmability

Carlos Jones/ORNL, CC BY 2.0 via Wikimedia Commons

OLCF at ORNL, CC BY 2.0 via Wikimedia Commons

How our GPU code fits into Arkouda



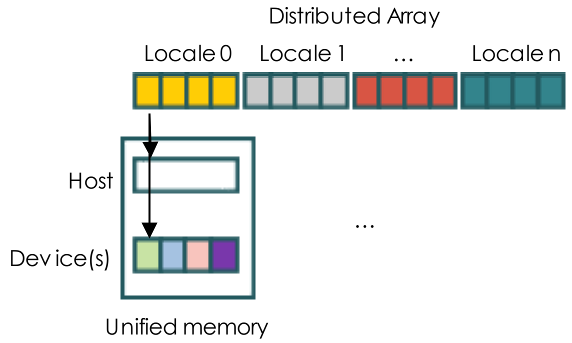
Georgia Tech Chapel GPU API

- Georgia Tech-developed framework abstracting over GPU programming models (CUDA, HIP, DPC++, SYCL)
- `GPUIterator`: exposing parallelism for kernel launch
  - low-level
- `GPUAPI`: device and memory management
  - low-mid-level: C-interoperability wrappers around device functions
  - mid-level: `GPUArray` to manage memory allocation, transfer
  - there is no high-level

GPUUnifiedDist: Arkouda Arrays in Shared Virtual Memory

- Host and device(s) share pointers to a single unified memory space
- Any access to memory that is currently in a different physical memory will result in a page fault, handled transparently with hardware support
- User-defined Chapel distribution `GPUUnifiedDist`
  - based on `BlockDist`
  - allocates memory for `LocGPUUnifiedArr` using `makeArrayFromPtr(umemPtr, …)`

```
module SymArrayDmapCompat …
proc makeDistDom(size:int, param GPU:bool = false) where GPU == true {
  select MyDmap {
    when Dmap.blockDist {
      return {0..#size} dmapped gpuUnifiedDist(…);
    }
    …
  }
}
```

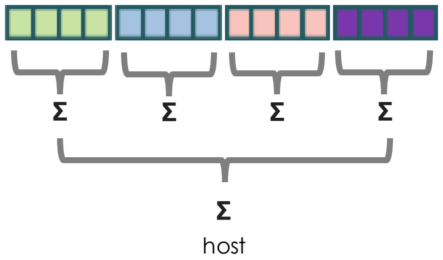


https://github.com/milthorpe/chapel-gpu
https://github.com/milthorpe/arkouda

Example: Sum on GPU (unified memory), GPU API

$$\sum_{i=0}^{n-1} A_i$$

`ak_arr.sum()`



Device memory allocation

Data transfer

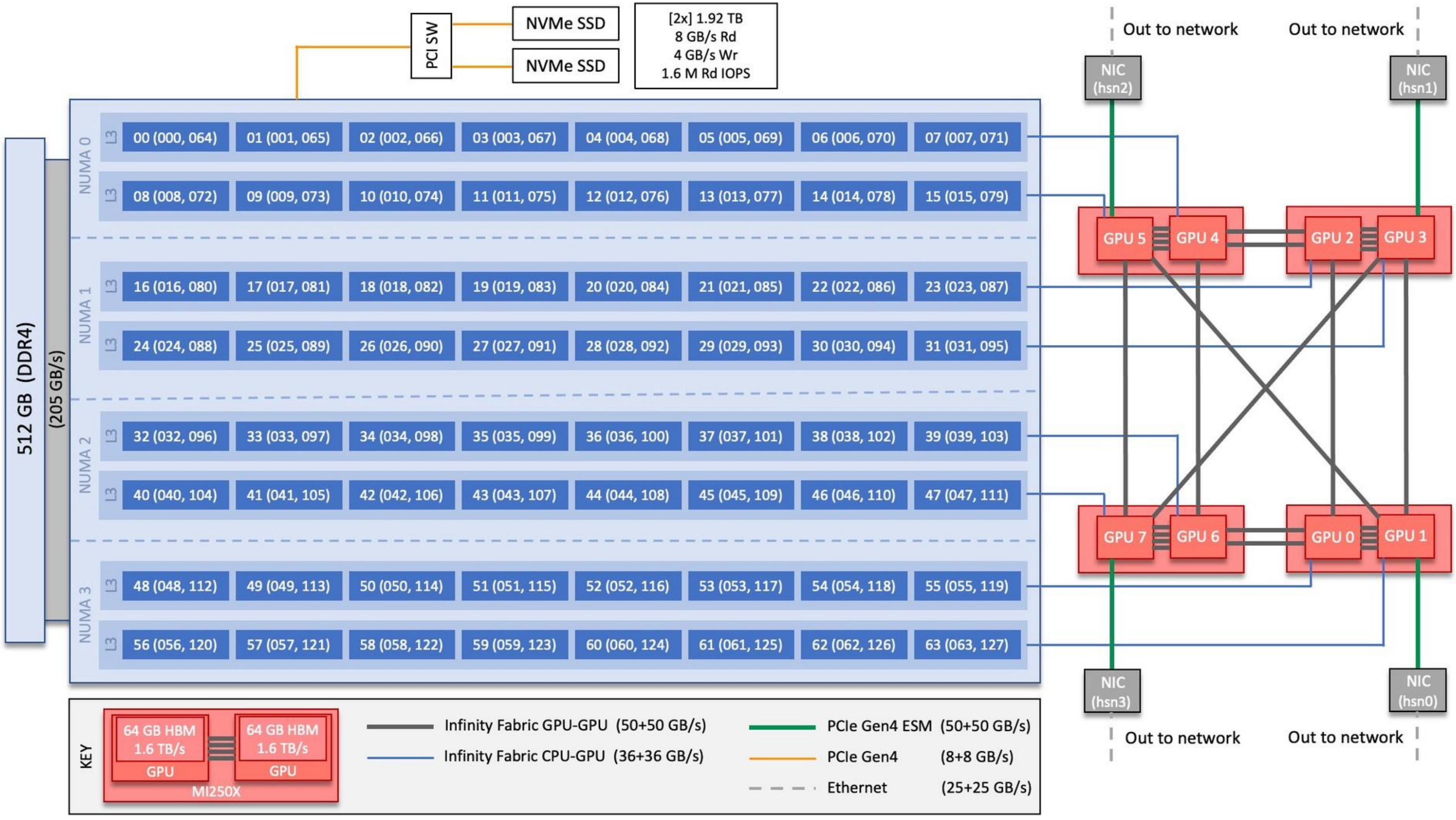
Synchronization

```
use GPUIterator;
use GPUAPI;

extern proc launchSum(devInPtr: c_void_ptr, devOutPtr: c_void_ptr, n: int): etype;

proc cubSum(ref e: SymEntry) where e.GPU == true {
  var deviceSum: [0..#nGPUs] e.etype;
  var sumCallback = lambda(lo: int, hi: int, n: int) {
    var devOut = new GPUArray(deviceSum[deviceId]);
    var deviceId: int(32);
    GetDevice(deviceId);
    e.prefetchLocalDataToDevice(lo, hi, deviceId);
    launchSum(e.c_ptrToLocalData(lo), devOut.dPtr(), n);
    DeviceSynchronize();
    devOut.fromDevice();
  };
  forall i in GPU(e.a.localSubdomain(), sumCallback) { }
  return (+ reduce deviceSum);
}
```

Frontier - Experimental Evaluation

- Evaluation platform: OLCF Frontier (+Crusher)
  - 1 × 64-core AMD EPYC 7A53 CPU @ 2.75GHz
  - 512 GiB DDR4 DRAM
  - 4 × AMD MI250X GPUs with 128 GiB HBM2E
  - Logically divided into 8 GPUs / 64 GiB HBM
  - ROCM 5.4.0
  - AMD driver version 6.1.5
- Arkouda Radix Sort LSD (Least Signficant Digit)
- Timing server-side Arkouda Chapel code directly (not from Python client)
  - Doesn't allow batching of communications

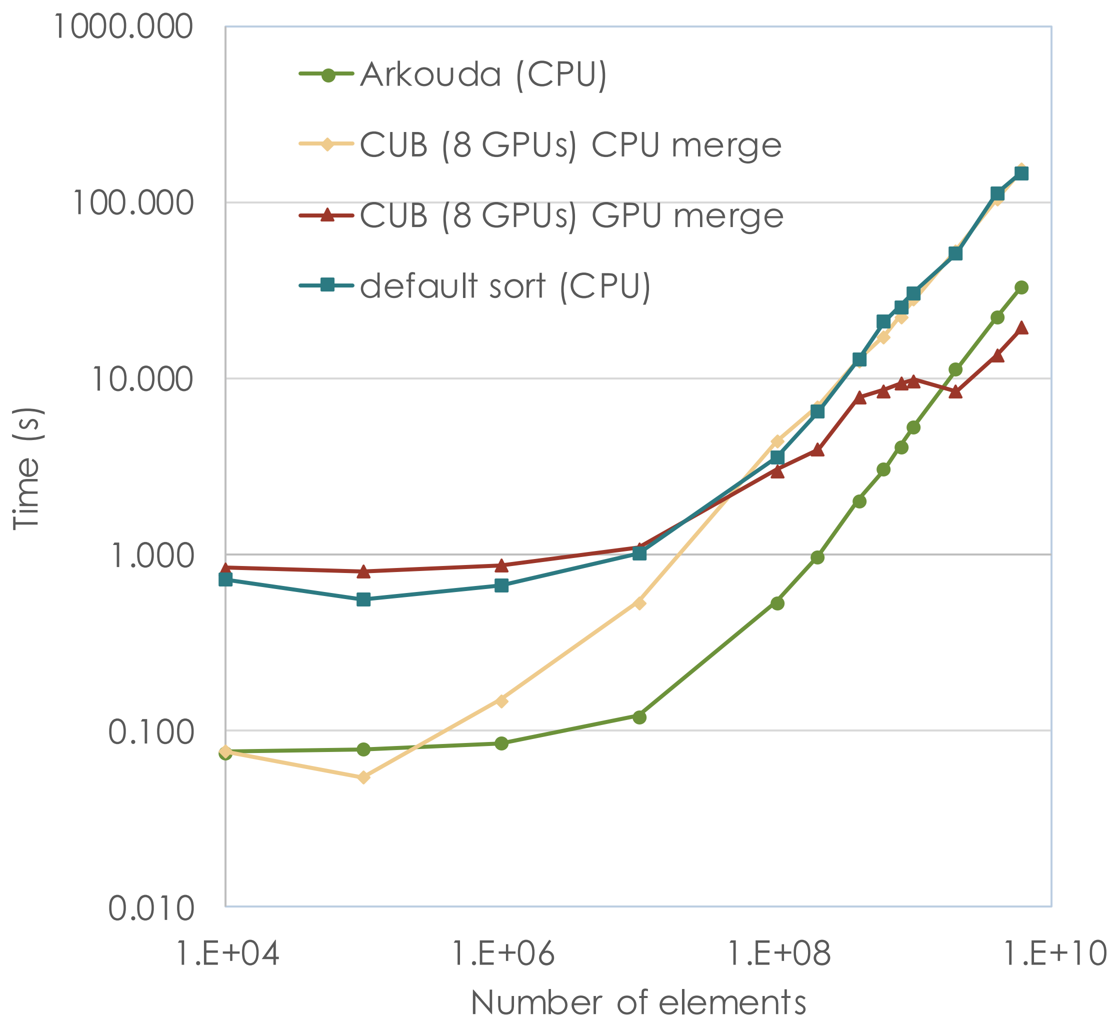
Frontier - Compute Node Schema



Sort (Frontier, single-node)

- Arkouda (CPU)
  `radixSortLSD_keys`
- Kernel:
  - CUB library
    `DeviceRadixSort::SortKeys`
  - merge on CPU:
    K-way merge
  - GPU merge:
    peer-to-peer swap and merge

Tobias Maltenberger, Ivan Ilic, Ilin Tolovski, and Tilmann Rabl. (2022) *Evaluating multi-GPU sorting with modern interconnects.* Intl. Conf. Management of Data. https://doi.org/10.1145/3514221.3517842

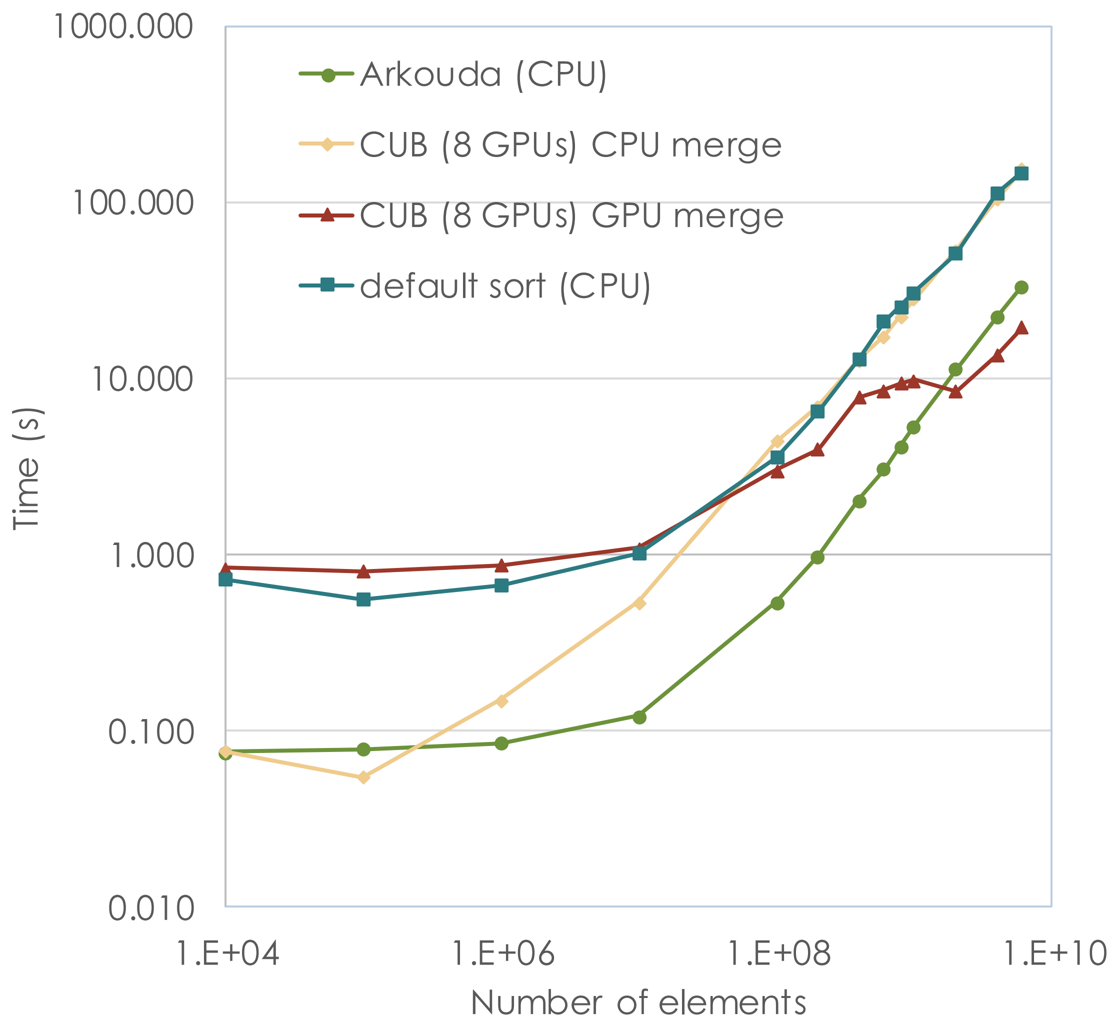



```
ak_df.sort_values()
```

Oak Ridge National Laboratory

Sort (Frontier, single-node cont.)

- GPU peer-to-peer merge beats Arkouda CPU for large data
- Approx. 60% of time is data transfer to/from GPU – can be eliminated if Arkouda arrays reside in GPU HBM



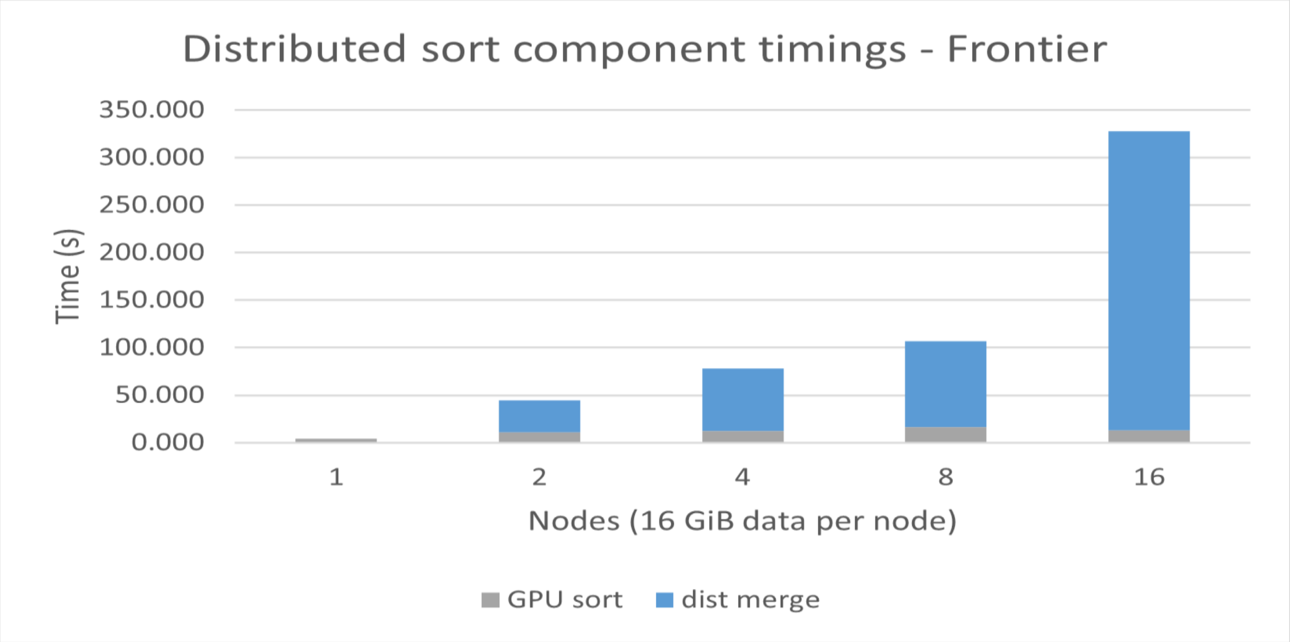

```
ak_df.sort_values()
```

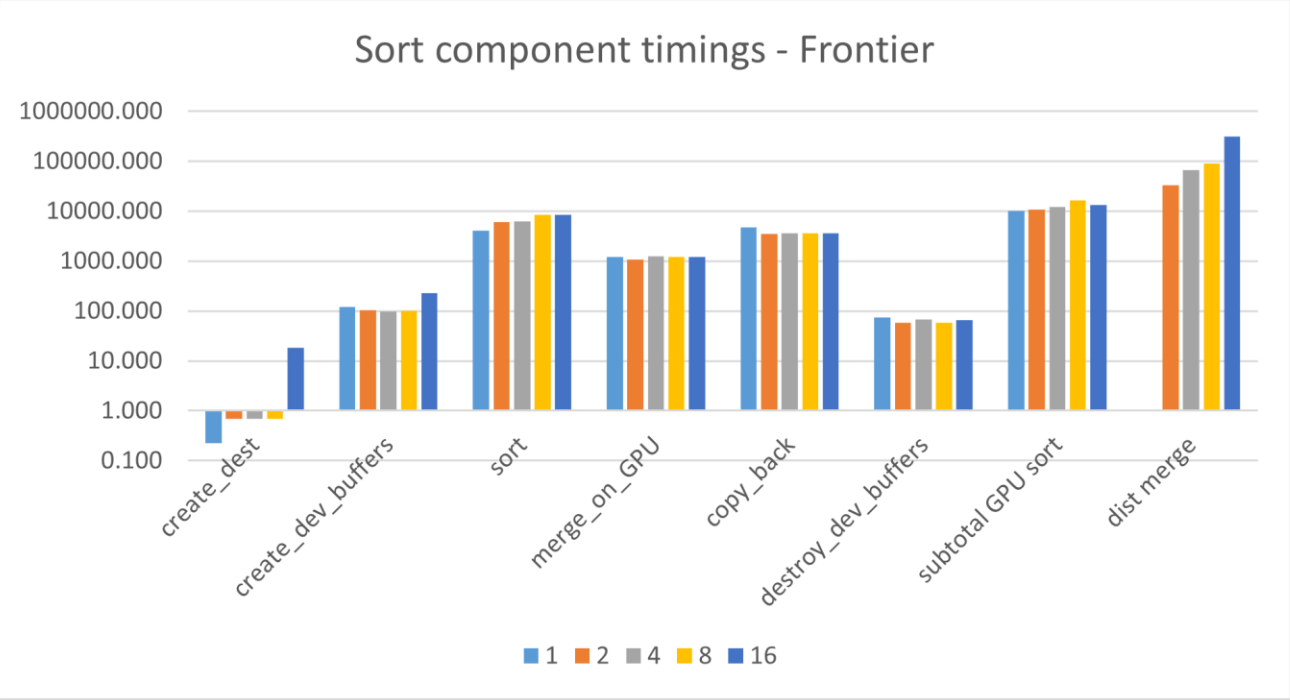
Sort (Frontier, multi-node)

- Arkouda (CPU)
  `radixSortLSD_keys`
- Kernel:
  - CUB library
    `DeviceRadixSort::SortKeys`
  - merge on CPU:
    K-way merge
  - GPU merge:
    peer-to-peer swap and merge

    Tobias Maltenberger, Ivan Ilic, Ilin Tolovski, and Tilmann Rabl. (2022) *Evaluating multi-GPU sorting with modern interconnects.* Intl. Conf. Management of Data. https://doi.org/10.1145/3514221.3517842
  - Inter-node merge:
    peer-to-peer swap and merge
    in host memory - O(n log n)

`ak_df.sort_values()`





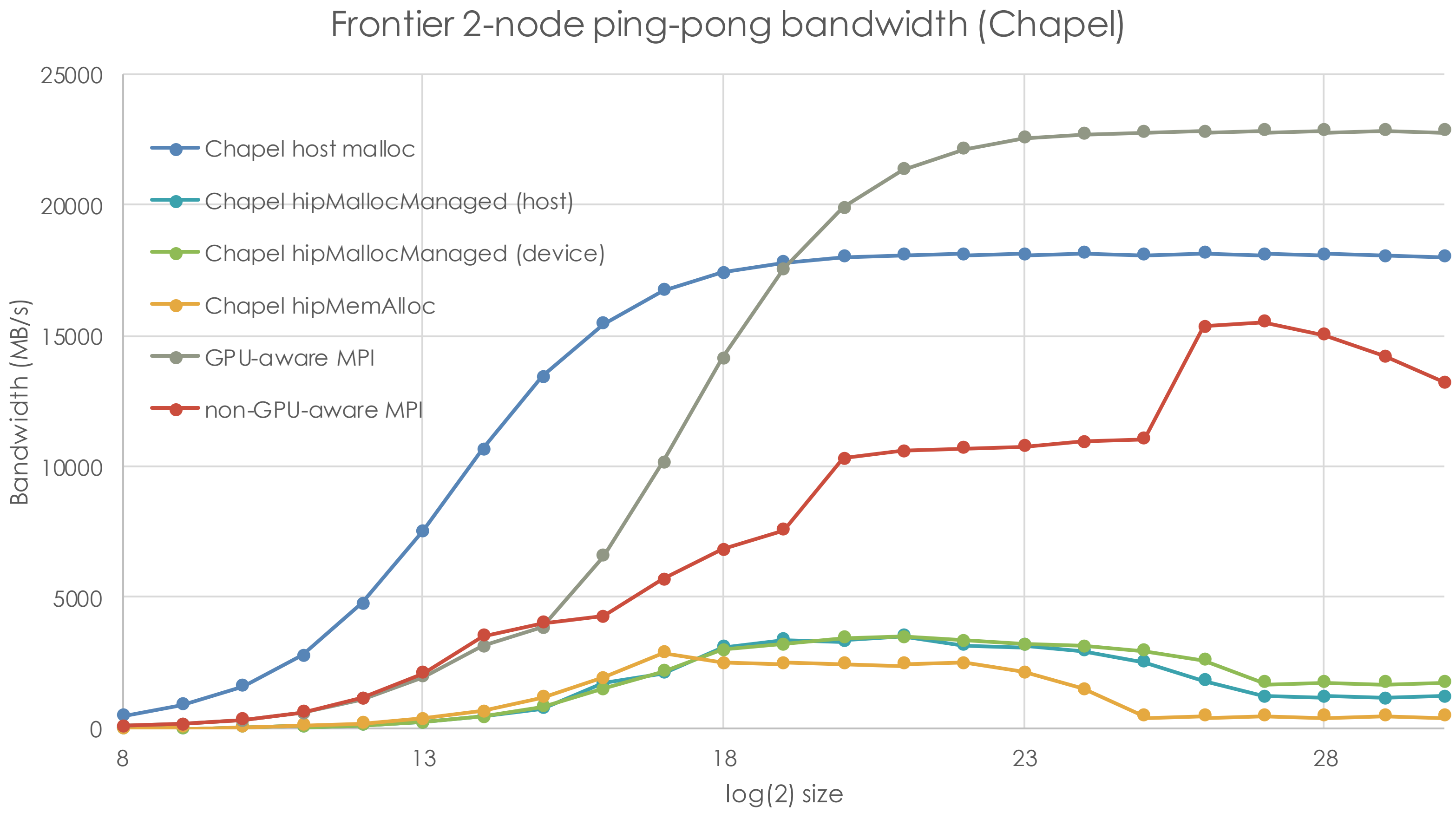
OAK RIDGE National Laboratory

Limitations

- Inter-node communications paging through host, non-GPU memory
    - Lower available bandwidth than GPU-direct comms
- Repeated communications in same session become slower! (Memory leak?)
- NUMA-awareness of GPU arrays
- Prefetch does not improve kernel time (unlike Summit/NVIDIA – is it even implemented?)

Internode Communications on GPU-connected EX

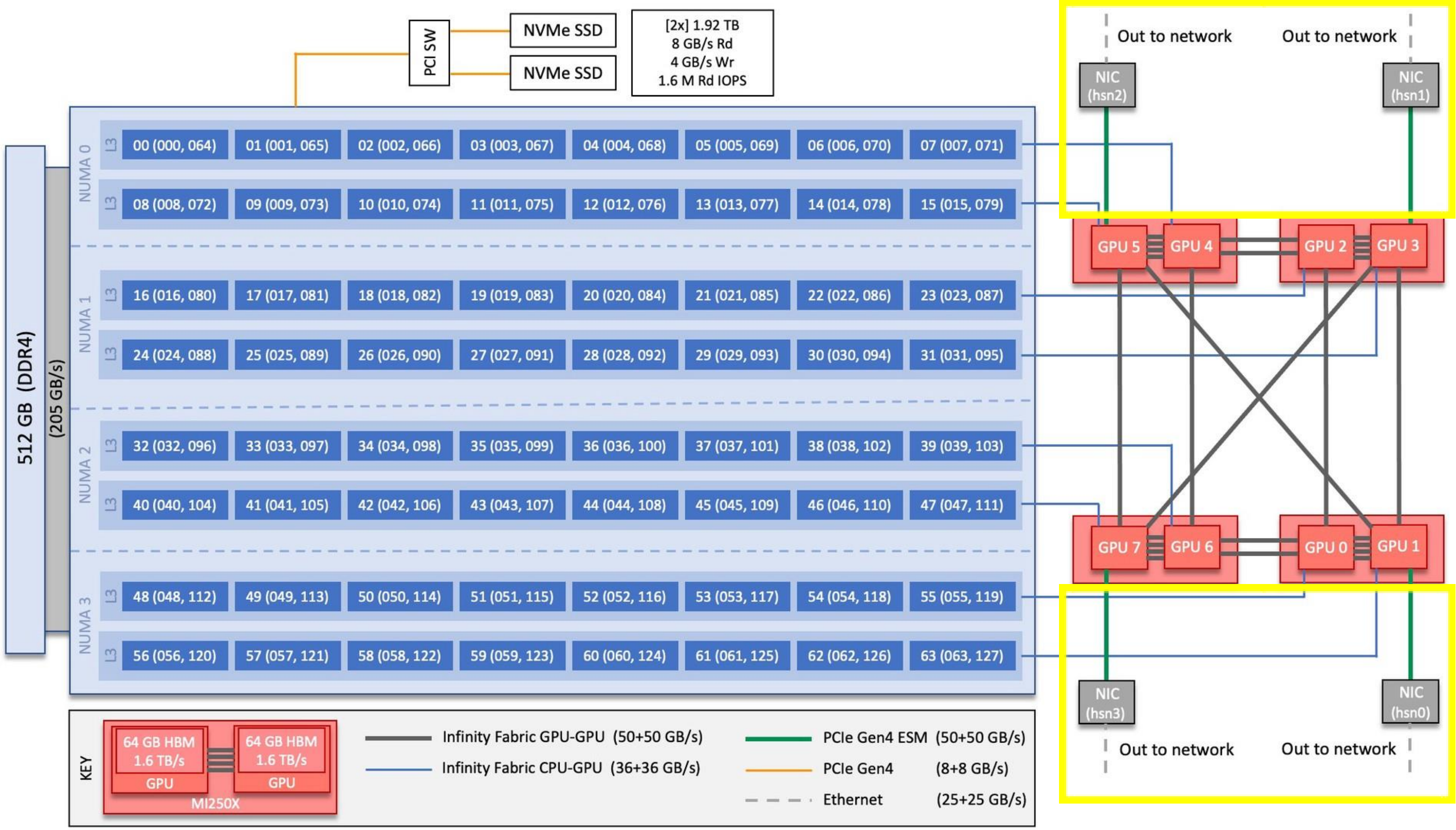
- Chapel comms bandwidth is significantly below GPU-aware MPI
  - Regardless of how memory is allocated



OAK RIDGE National Laboratory

Chapel OFI Comms Layer Changes

- Frontier network interface cards (NICs) are directly connected to GPUs
- Chapel OFI comms layer doesn't utilise libfabric FI_HMEM intranode device support
- Need to register device memory regions - new memory allocator

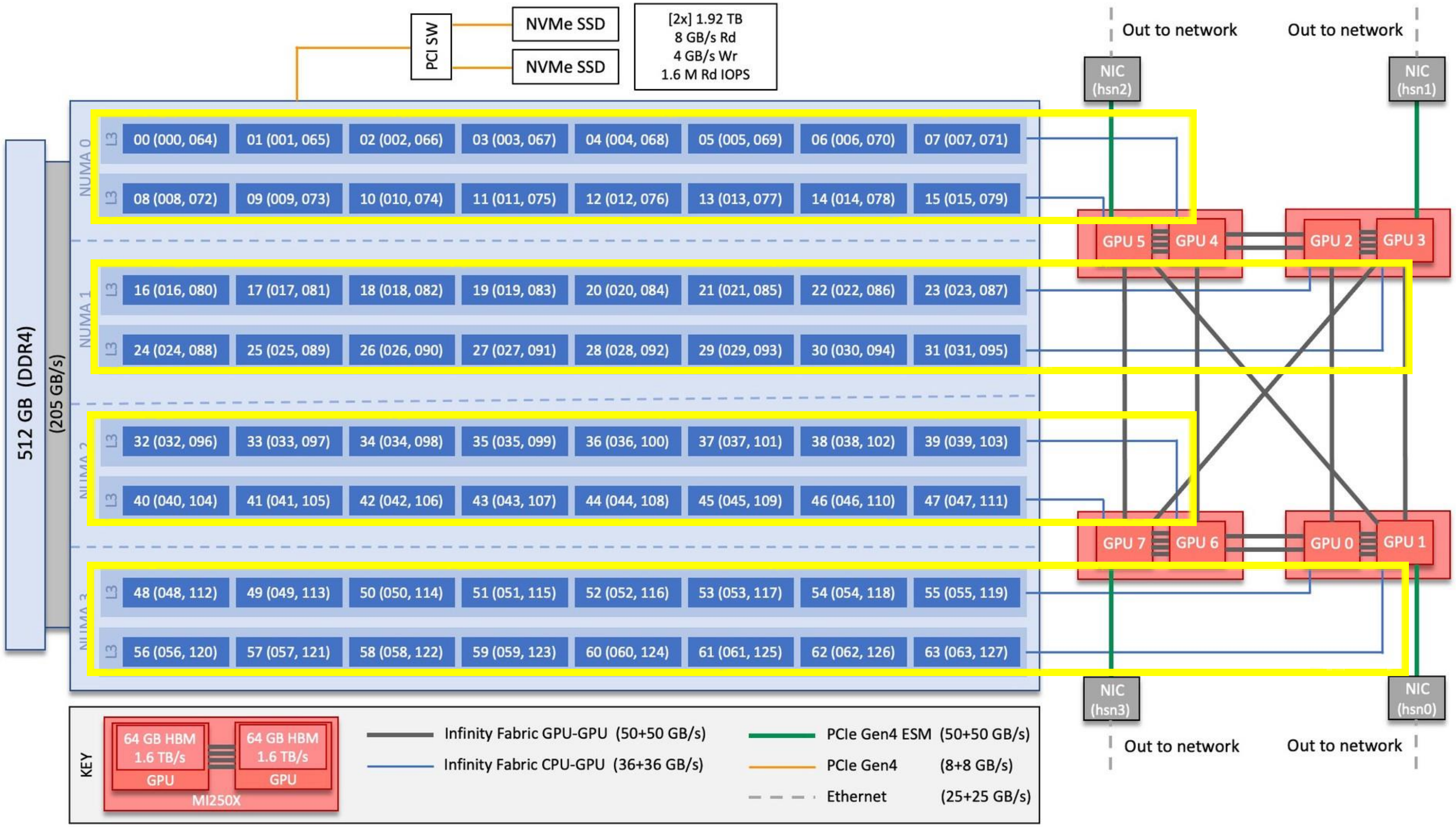


https://docs.olcf.ornl.gov/systems/frontier_user_guide.html

Oak Ridge National Laboratory

NUMA on Frontier

- Chapel distributed arrays aren't NUMA-aware
- Full host-device flood bandwidth (200GB/s) requires NUMA-aware placement of host array segments



https://docs.olcf.ornl.gov/systems/frontier_user_guide.html

Opportunities for Chapel GPU Code Generation

- With GPU code generation, some algorithms may be appropriate for both multi-GPU and distributed computation

```
proc selectPivot(A: [], targetLocs: [] locale) {
  var endLH = A.localSubdomain(targetLocs[targetLocs.size/2 - 1]).last;
  var startRH = A.localSubdomain(targetLocs[targetLocs.size/2]).first;
  var partitionSize = A.localSubdomain(targetLocs.first).size;
  var lo = 0;
  var hi = partitionSize * targetLocs.size/2;
  while lo < hi {
    var mid = hi - (hi - lo) / 2;
    if A[endLH-mid+1] <= A[startRH + mid-1] then
      hi = mid - 1;
    else
      lo = mid;
    }
  return lo;
}
```

This code is from DistMerge.chpl; currently, each node is a single locale; but it would work equally well if targetLocs referred to GPU devices

```
proc mergePartitions(ref src, ref tmp, const targetLocs: [] locale) {
  if targetLocales.size > 2 then
    coforall i in 0..1 {
      var localesSubset: [0..# targetLocs.size / 2] locale;
      localesSubset = targetLocs((i * (targetLocs.size / 2))
         .. #(targetLocales.size / 2));
      mergePartitions(src, tmp, localesSubset);
  }
  var pivot = selectPivot(src, targetLocales);
  if pivot > 0 {
    var locsToMerge = swapPartitions(src, tmp, targetLocs, pivot);
    coforall (loc, cut) in locsToMerge do on loc {
      mergeSortedKeysAtCut(src, tmp, cut);
    }
  }
  // recursive merge again
}
```

Opportunities for Chapel GPU Code Generation (2)

- This code works for inter-locale swap (doesn't work for GPUs)

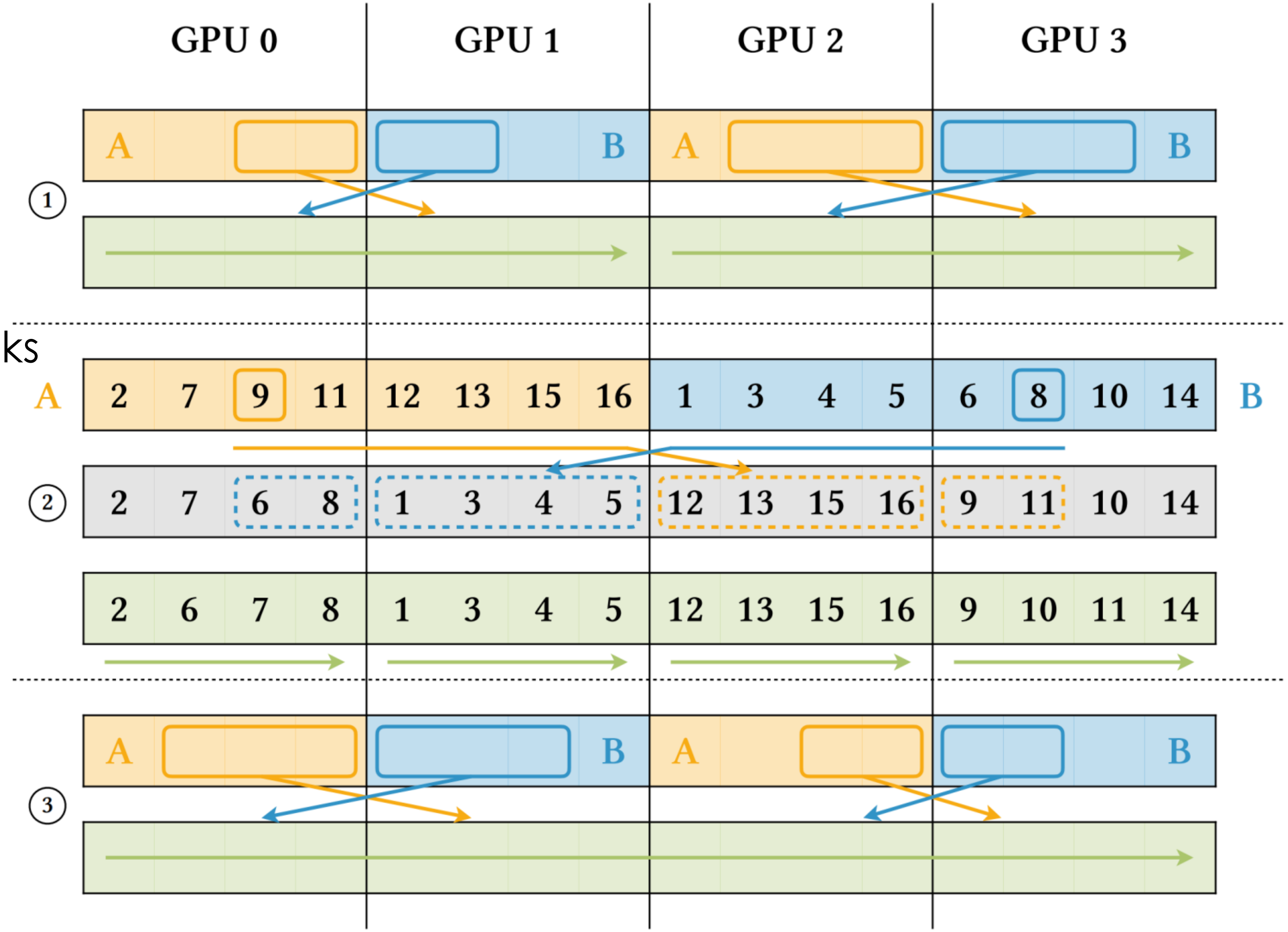
```
proc swapPartitions(ref src, ref tmp, const targetLocales: [] locale, in pivot: int) {
    var partitionSize = src.size / src.domain.targetLocales().size;
    var localesToSwap = pivot / partitionSize;
    …
    coforall i in 0..localesToSwap with (const targetLocales, ref localesToMerge) {
        const leftLocale = …;
        const rightLocale = …;
        const numElements = if (i == localesToSwap) then pivot else partitionSize;

        const leftStart = src.localSubdomain(leftLocale).first + partitionSize - numElements;
        const rightStart = src.localSubdomain(rightLocale).first;

        cobegin {
            on leftLocale do tmp.localSlice[leftStart..#numElements] = src[rightStart..#numElements];
            on rightLocale do tmp.localSlice[rightStart..#numElements] = src[leftStart..#numElements];
        }
        ...
    }
}
```

Multi-GPU Swap and Merge Algorithm

2 log2(N) -1 rounds

Need to access individual elements across all GPU chunks

and copy chunks between arrays



Tobias Maltenberger, Ivan Ilic, Ilin Tolovski, and Tilmann Rabl. (2022) *Evaluating multi-GPU sorting with modern interconnects.* Intl. Conf. Management of Data. https://doi.org/10.1145/3514221.3517842

GPU API

```
template <typename T>
inline __device__ void GetValueFromVirtualPartition(size_t partition_size, T** virtual_partition, size_t index,
                                                     T* value) {
  *value = virtual_partition[index / partition_size][index % partition_size];
}

template <typename T>
__global__ void SelectPivot(size_t partition_size, size_t num_partitions, T** local_virtual_partition,
                            T** remote_virtual_partition, size_t* result_pivot) {
  size_t low = 0;
  size_t high = partition_size * num_partitions;

  while (low < high) {
    const size_t mid = high - (high - low) / 2;
    T a;
    GetValueFromVirtualPartition<T>(partition_size, local_virtual_partition, partition_size * num_partitions - mid, &a);
    T b;
    GetValueFromVirtualPartition<T>(partition_size, remote_virtual_partition, mid - 1, &b);
    if (a <= b) {
      high = mid - 1;
    } else {
      low = mid;
    }
  }
  *result_pivot = low;
}
```

Swap partitions – CUDA/HIP - called from GPU API

```
template <typename T>
std::array<int, 2> SwapPartitions(DeviceBuffers<T>* device_buffers, size_t pivot, const std::vector<int>& devices) {
  const size_t partition_size = device_buffers->GetPartitionSize();
  size_t devices_to_swap = pivot / partition_size;
  ...
#pragma omp parallel for
  for (size_t i = 0; i <= devices_to_swap; ++i) {
    const int left_device = ...;
    const int right_device = ...;
    const size_t num_elements = (i == devices_to_swap) ? pivot : partition_size;

    CheckCudaError(cudaSetDevice(left_device));

    CheckCudaError(cudaMemcpyAsync(
        thrust::raw_pointer_cast(device_buffers->AtSecondary(left_device)->data() + partition_size - num_elements),
        thrust::raw_pointer_cast(device_buffers->AtPrimary(right_device)->data()), sizeof(T) * num_elements,
        cudaMemcpyDeviceToDevice, *device_buffers->GetPrimaryStream(left_device)));

    CheckCudaError(cudaSetDevice(right_device));

    CheckCudaError(cudaMemcpyAsync(
        thrust::raw_pointer_cast(device_buffers->AtSecondary(right_device)->data()),
        thrust::raw_pointer_cast(device_buffers->AtPrimary(left_device)->data() + partition_size - num_elements),
        sizeof(T) * num_elements, cudaMemcpyDeviceToDevice, *device_buffers->GetPrimaryStream(right_device)));
    ...
  }
```

Merge Partitions – CUDA/HIP

```
template <typename T>
void MergePartitions(DeviceBuffers<T>* device_buffers, HostVector<T>* elements, const std::vector<int>& devices,
                     size_t num_fillers) {
  if (devices.size() > 2) {
#pragma omp parallel for
    for (size_t i = 0; i < 2; ++i) {
      MergePartitions(
          device_buffers, elements,
          {devices.begin() + (i * (devices.size() / 2)), devices.begin() + ((i + 1) * (devices.size() / 2))},
          num_fillers);
    }
  }
  const size_t pivot = FindPivot<T>(device_buffers, devices);
  if (pivot > 0) {
    const std::array<int, 2> devices_to_merge = SwapPartitions<T>(device_buffers, pivot, devices);
    MergeLocalPartitions<T>(device_buffers, elements, pivot, devices_to_merge, devices, num_fillers);
  }
  // recursive merge again
```

Chapel GPU Code Generation can make our lives easier

- Note – below is fictional code that we envision as nice to haves as developers when implementing algorithms, such as ones that require system specific optimizations (Frontier)

- Allocating a distributed GPU array across all GPUs on a node

```
var Dom: domain(1) dmapped blockDist({0..n}, targetLocales=here.gpus) = {0..n};
```

- Direct copy between portions of an array allocated to different devices on a single node

```
const gpu0Dom = A.dom.localSubdomain(here.gpus[0]);
const gpu7Dom = A.dom.localSubdomain(here.gpus[7]);
A[gpu0Dom] = A[gpu7Dom];
```

- Direct copy between portions of an array allocated to devices on different nodes

```
const localGpu0Dom = A.dom.localSubdomain(here.gpus[0]);
const remoteGpu0Dom = A.dom.localSubdomain(Locales[1].gpus[0]);
A[localGpu0Dom] = A[remoteGpu0Dom];
```

Summary and Future Work

- Georgia Tech GPU API
  - Granularity met with limitations and tedious, difficult to debugs implementation
- Frontier and AMD specific issues
  - ROCM bugs
- Frontier and Chapel incompatability
  - HBM through GPUs, not host
  - Chapel Arrays are not NUMA aware
- Chapel GPU Code Gen
  - Realize is work in progress
  - Feedback on potential solutions to solve issues caused by high granularity of Georgia Tech GPU API

This research used resources of the Oak Ridge Leadership Computing Facility (OLCF) and 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.