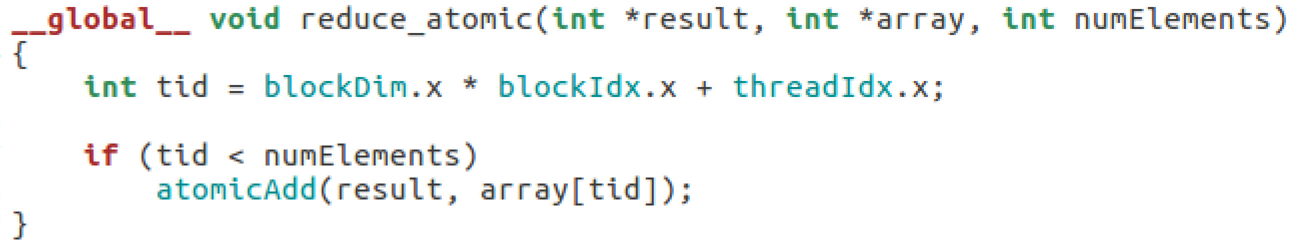
CMSC 691
High Performance Distributed Systems

CUDA reduction and sorting

Dr. Alberto Cano
Assistant Professor
Department of Computer Science
acano@vcu.edu

Naïve reduction 1

- Atomic instructions
- Every single thread increments the atomic result

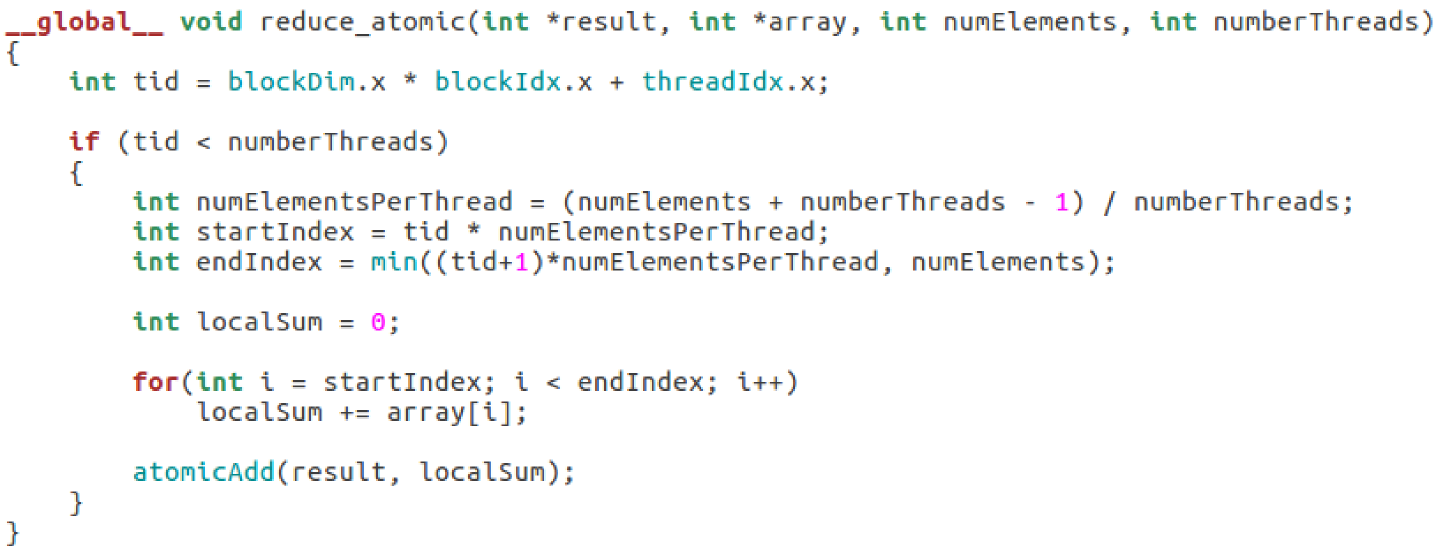


```
__global__ void reduce_atomic(int *result, int *array, int numElements)
{
    int tid = blockDim.x * blockIdx.x + threadIdx.x;

    if (tid < numElements)
        atomicAdd(result, array[tid]);
}
```

- There’s no actual parallelization
- Threads in the warp & block compete, serializing the execution
- Reads from global memory are coalesced

Naïve reduction 2

- Reduce the number of threads
- Assign each thread the reduction of a subset of the array
- Add the partial results using atomic instructions
- Multi-core CPU style

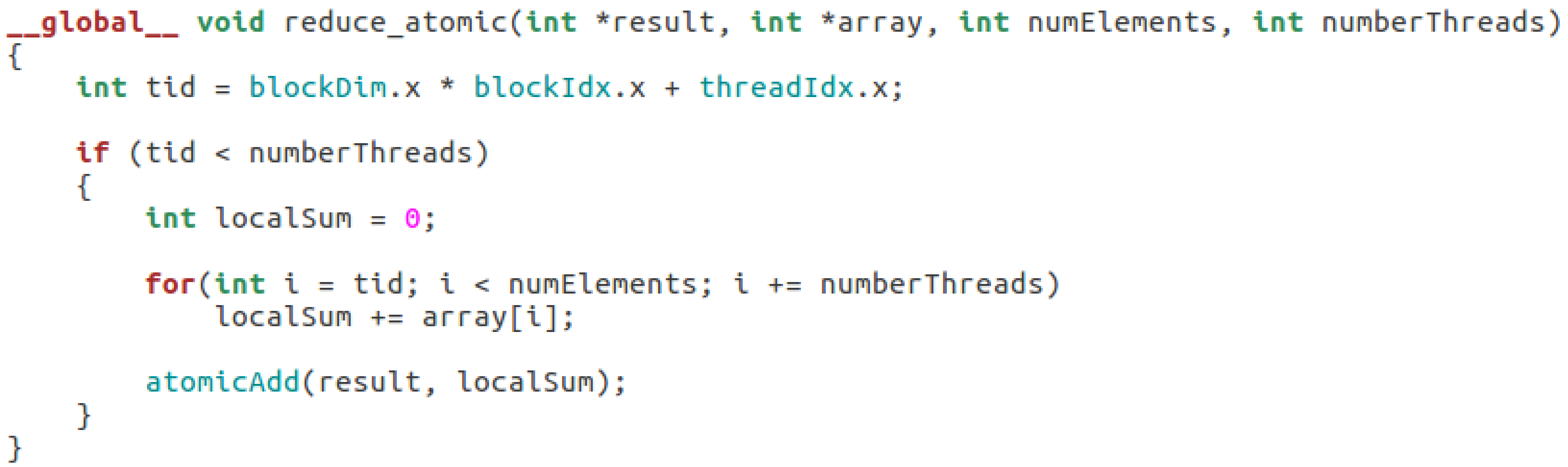


```
__global__ void reduce_atomic(int *result, int *array, int numElements, int numberThreads)
{
    int tid = blockDim.x * blockIdx.x + threadIdx.x;

    if (tid < numberThreads)
    {
        int numElementsPerThread = (numElements + numberThreads - 1) / numberThreads;
        int startIndex = tid * numElementsPerThread;
        int endIndex = min((tid+1)*numElementsPerThread, numElements);

        int localSum = 0;

        for(int i = startIndex; i < endIndex; i++)
            localSum += array[i];

        atomicAdd(result, localSum);
    }
}
```

- Memory access pattern not coalesced!

Naïve reduction 3

- Same methodology but using a coalesced memory access pattern
- Every iteration the displacement is *numberThreads* positions

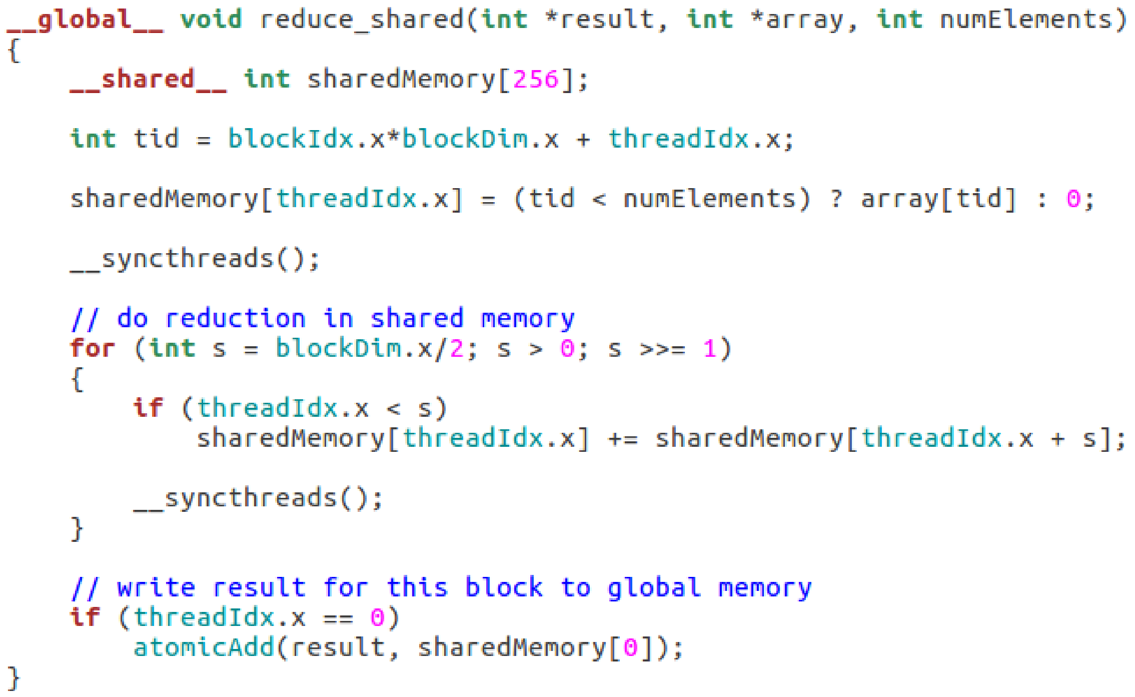


```
__global__ void reduce_atomic(int *result, int *array, int numElements, int numberThreads)
{
    int tid = blockDim.x * blockIdx.x + threadIdx.x;

    if (tid < numberThreads)
    {
        int localSum = 0;

        for(int i = tid; i < numElements; i += numberThreads)
            localSum += array[i];

        atomicAdd(result, localSum);
    }
}
```

- Alternative: reduce local results within the thread block

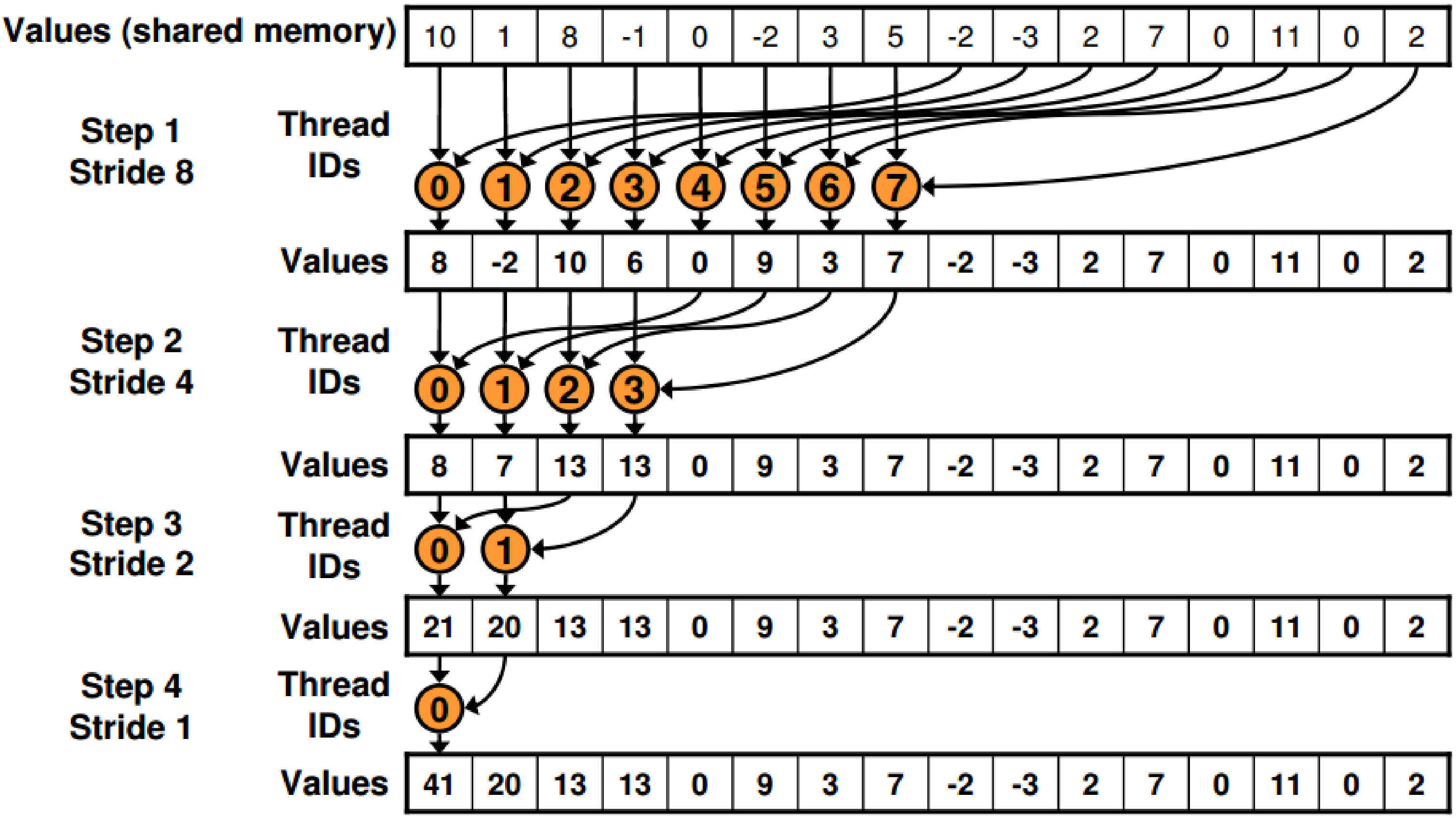
Shared memory reduction

- Load the data into shared memory
- Perform local reduction per thread block
- Finally, only one thread per block run the atomic add



```
__global__ void reduce_shared(int *result, int *array, int numElements)
{
    __shared__ int sharedMemory[256];

    int tid = blockIdx.x*blockDim.x + threadIdx.x;

    sharedMemory[threadIdx.x] = (tid < numElements) ? array[tid] : 0;

    __syncthreads();

    // do reduction in shared memory
    for (int s = blockDim.x/2; s > 0; s >>= 1)
    {
        if (threadIdx.x < s)
            sharedMemory[threadIdx.x] += sharedMemory[threadIdx.x + s];

        __syncthreads();
    }

    // write result for this block to global memory
    if (threadIdx.x == 0)
        atomicAdd(result, sharedMemory[0]);
}
```

Shared memory reduction (sequential addressing)

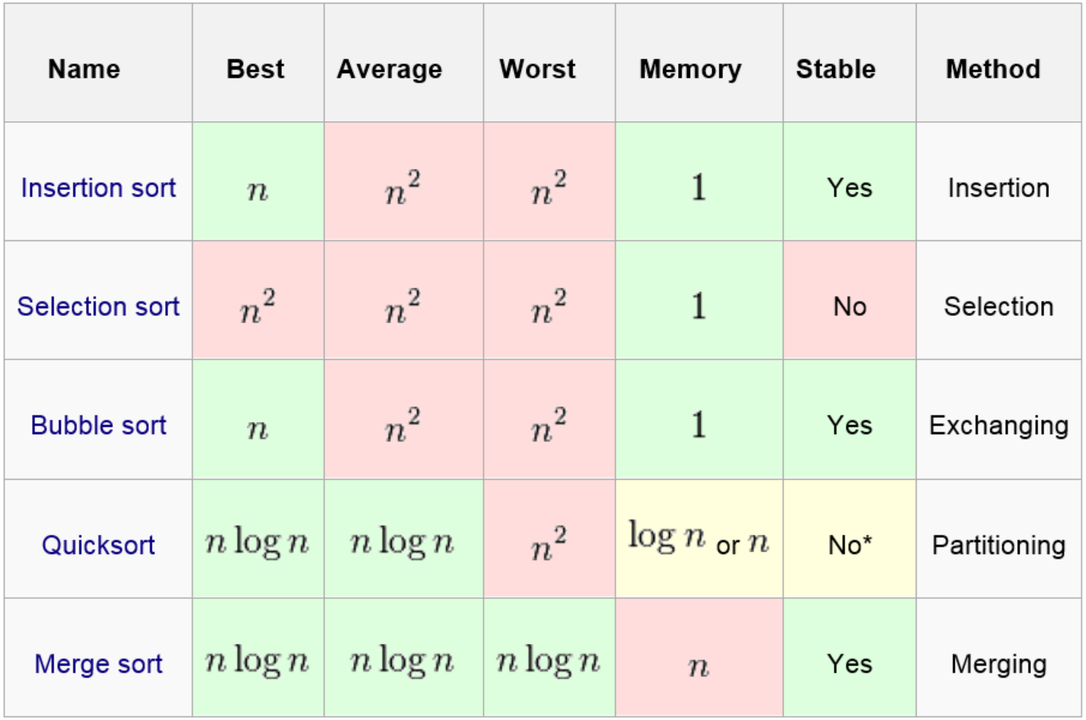


	Values (shared memory)	10	1	8	-1	0	-2	3	5	-2	-3	2	7	0	11	0	2
Step 1 Stride 8	Thread IDs	0	1	2	3	4	5	6	7								
	Values	8	-2	10	6	0	9	3	7	-2	-3	2	7	0	11	0	2
Step 2 Stride 4	Thread IDs	0	1	2	3												
	Values	8	7	13	13	0	9	3	7	-2	-3	2	7	0	11	0	2
Step 3 Stride 2	Thread IDs	0	1														
	Values	21	20	13	13	0	9	3	7	-2	-3	2	7	0	11	0	2
Step 4 Stride 1	Thread IDs	0															
	Values	41	20	13	13	0	9	3	7	-2	-3	2	7	0	11	0	2

- How to combine the partial results from different shared memories_?

Sorting: order an array of keys whose elements are comparable

- Internal (in-place) vs external (require extra memory)
- Stable: maintain the relative order for equal keys
- Recursion: divide and conquer

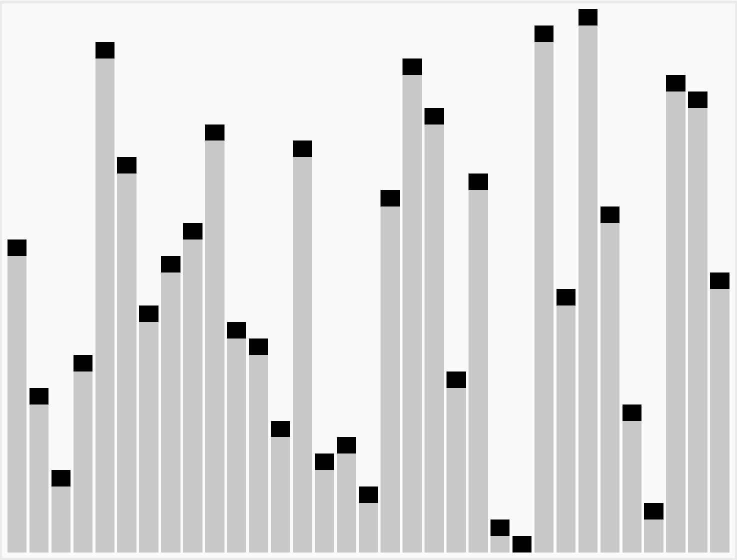


Name	Best	Average	Worst	Memory	Stable	Method
Insertion sort	n	n^2	n^2	1	Yes	Insertion
Selection sort	n^2	n^2	n^2	1	No	Selection
Bubble sort	n	n^2	n^2	1	Yes	Exchanging
Quicksort	$n \log n$	$n \log n$	n^2	$\log n$ or n	No*	Partitioning
Merge sort	$n \log n$	$n \log n$	$n \log n$	n	Yes	Merging

Quicksort

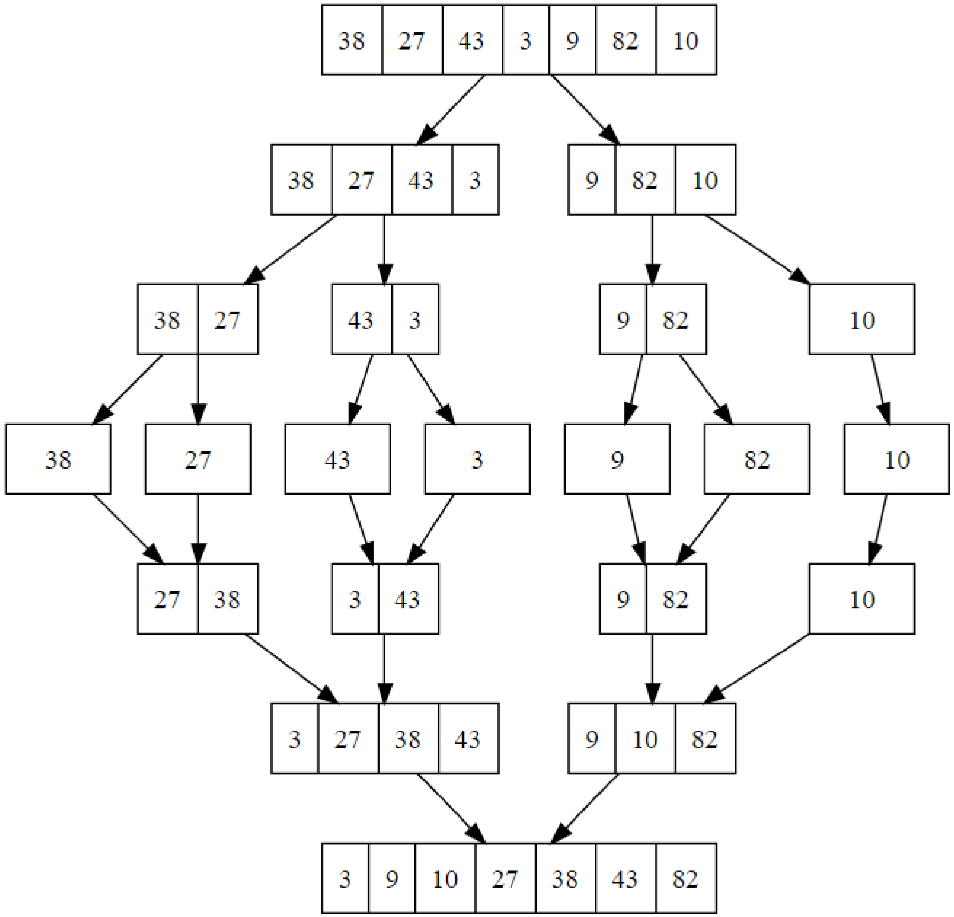
- Divide and conquer, completely parallelizable!

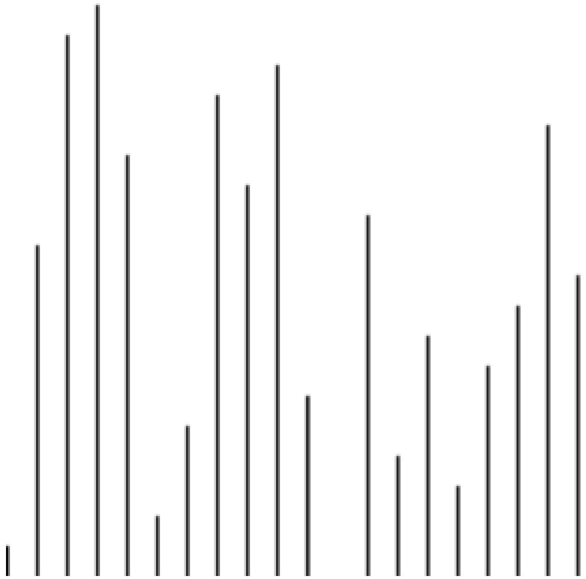
```
quicksort(A, lo, hi)
  if lo < hi
    p ← partition(A, lo, hi)
    quicksort(A, lo, p - 1)
    quicksort(A, p + 1, hi)

partition(A, lo, hi)
  pivot ← A[hi]
  i ← lo
  for j ← lo to hi - 1
    if A[j] <= pivot
      swap A[i] and A[j]
      i ← i + 1
  swap A[i] and A[hi]
  return i
```

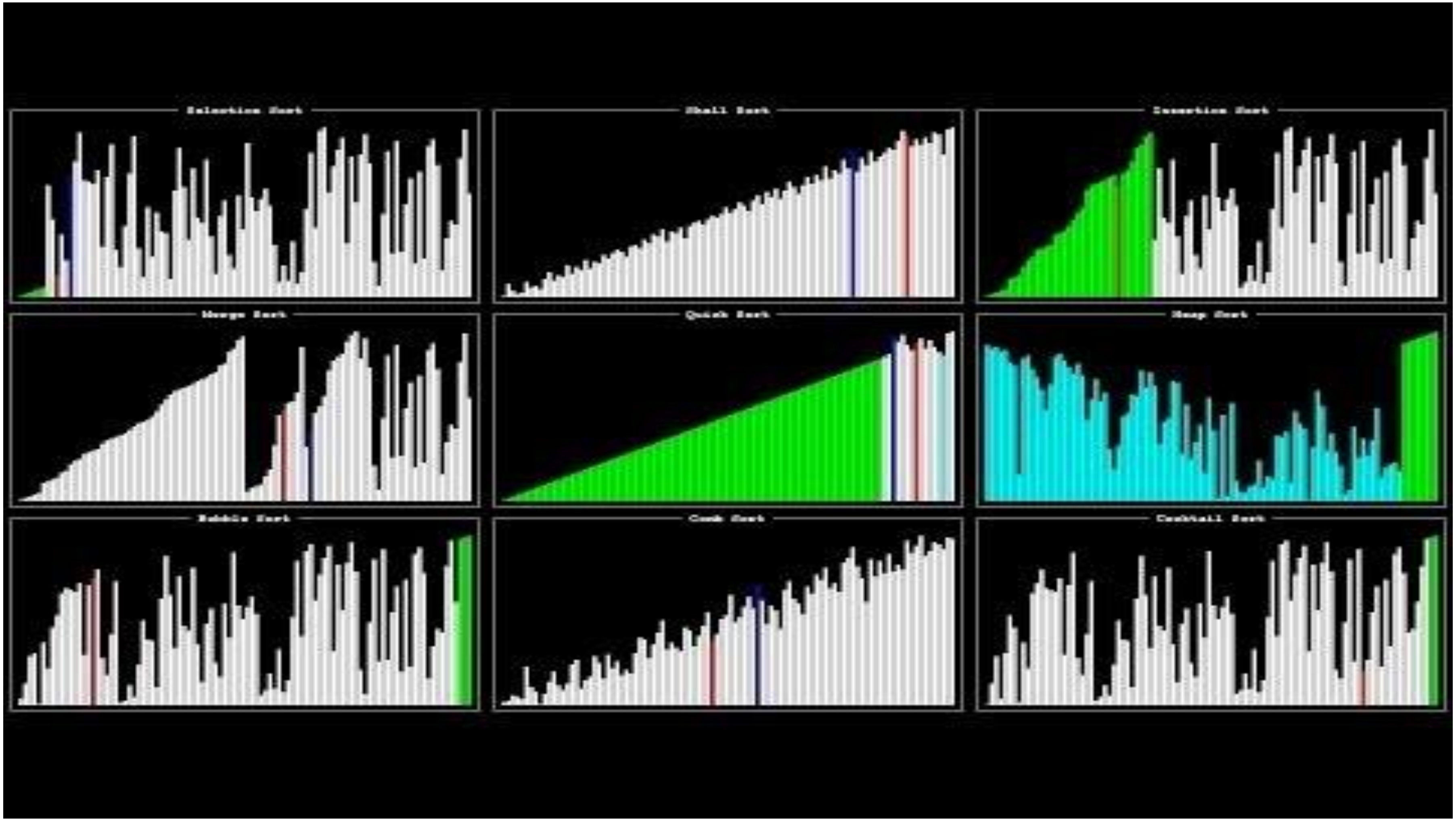


Mergesort

```
mergesort(A, lo, hi)
  if lo+1 < hi
    mid = (lo + hi) / 2
    fork
      mergesort(A, lo, mid)
      mergesort(A, mid, hi)
    join
    merge(A, lo, mid, hi)
```

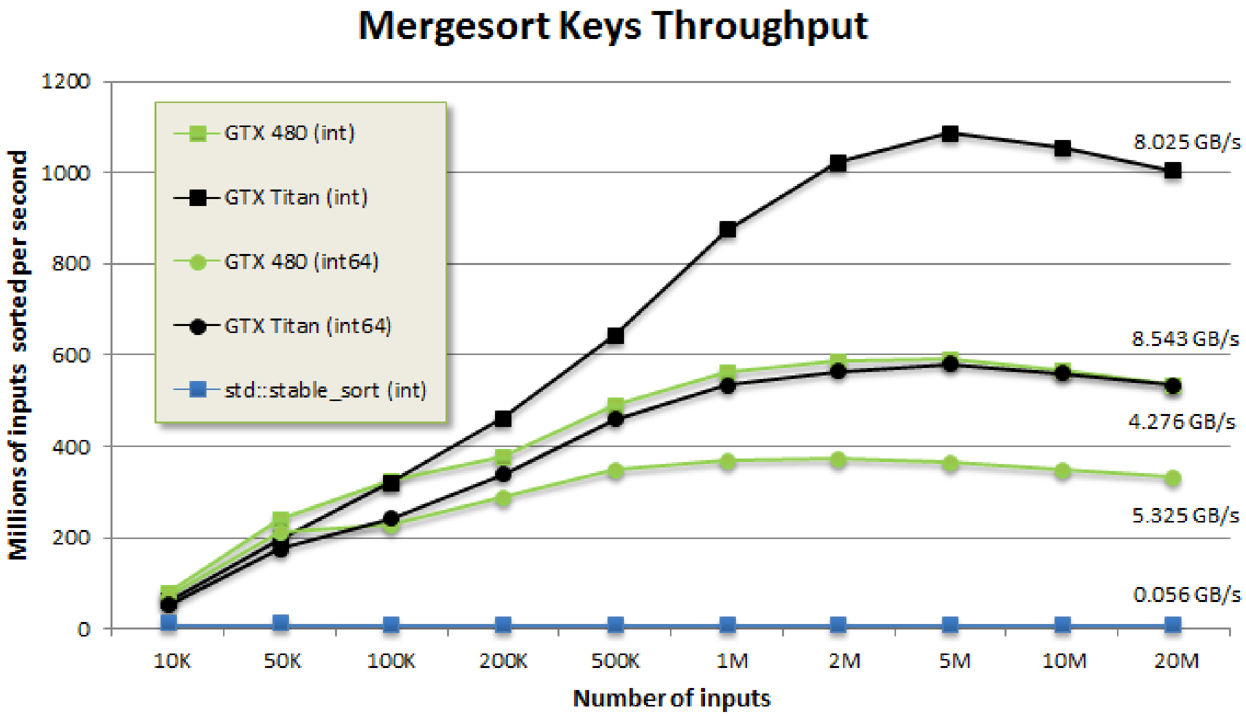





Selection Sort
Shell Sort
Insertion Sort
Merge Sort
Quick Sort
Heap Sort
Bubble Sort
Comb Sort
Cocktail Sort

GPU parallel sorting

- Mergesort, Bitonic sort, Radix sort for LSB O(kn)
- Youtube: Radix Sort Part 1 - Intro to Parallel Programming
- Don’t panic, we won’t have to implement this


Mergesort Keys Throughput

Thrust library

- Thrust parallel template library to implement high-performance applications with *minimal* programming effort
- Based on the C++ Standard Template Library (STL)
- Provides containers for *host_vector* and *device_vector*

```
thrust::device_vector<int> v(size);
```

- Allows casting raw pointers to device pointer

```
cudaMalloc((void **) &raw_ptr, N * sizeof(int));
thrust::device_ptr<int> dev_ptr(raw_ptr);
```

- Algorithms: binary search, reduce, count, min/max, sort, sortbykey

CMSC 691
High Performance Distributed Systems

CUDA reduction and sorting

Dr. Alberto Cano
Assistant Professor
Department of Computer Science
acano@vcu.edu