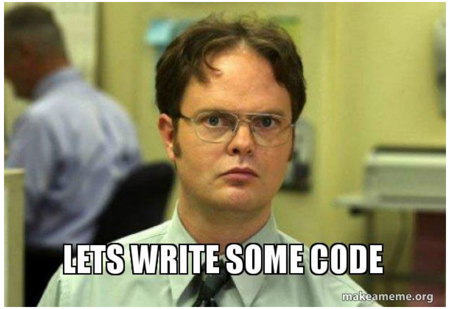
CMSC 603
High-Performance Distributed Systems

CUDA reduction and sorting

College of Engineering

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

Reduction: CPU vs GPU

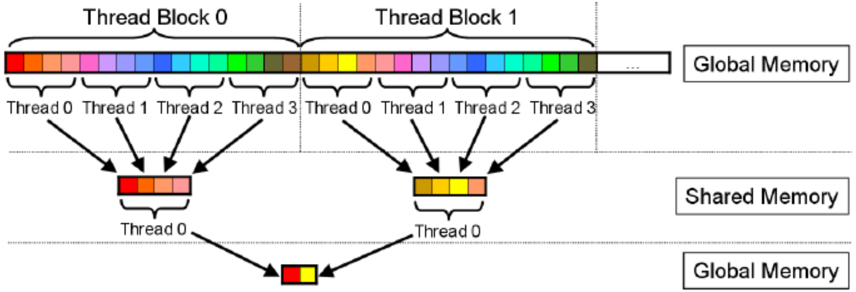


CPU code:

```
float sum = 0.0;
for (int i = 0; i < size; i++)
    sum += array[i];
```

GPU code:

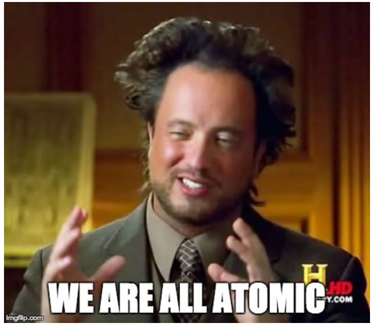
Assign, allocate, initialize device and host memory
Create threads and assign indices for each thread
Assign each thread a specific **region** to get a sum over
Wait for all threads to finish running
Combine all thread sums for final solution



Atomic operations

- CUDA provides built in atomic operations but **DO NOT** use them unless strictly necessary. Rewrite the code to avoid them
- Introduce long overheads due to guarantee the mutual exclusion
- Use the functions: atomic<operator>(float *address, float val);
- Operators: Add, Sub, Exch, Min, Max, Inc, Dec, And, Or, Xor

*atomicAdd(float *address, float val)*



Naïve reduction 1

- Using atomic instructions
- Every single thread increments the result atomically

```
__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 but the opposite
- Threads in the warp, block, and grid **compete** then serializing the execution
- Reads from global memory are coalesced

See: reduction_atomic_1.cu

Naïve reduction 2

- Reduce the number of threads to a reasonable number
- Assign each thread the reduction of a subset of the array
- Add the partial results using atomic instructions
- Multi-thread 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! stride > 1

See: reduction_atomic_2.cu

Naïve reduction 3

- Same methodology but using a coalesced memory access pattern
- Every iteration the memory leap 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);
    }
}
```

- Still can do better? reduce local results within the thread block

See: reduction_atomic_3.cu

Naïve reduction 4

- Reduce the localSum per block and perform a single atomicAdd
- However, only one thread is active doing the reduction for blockDim.x results

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

    __shared__ int sharedMemory[128];

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

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

        sharedMemory[threadIdx.x] = localSum;

        __syncthreads();

        if (threadIdx.x == 0)
        {
                for(int i = 1; i < blockDim.x; i++)
                        localSum += sharedMemory[i];

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

See: reduction_atomic_4.cu

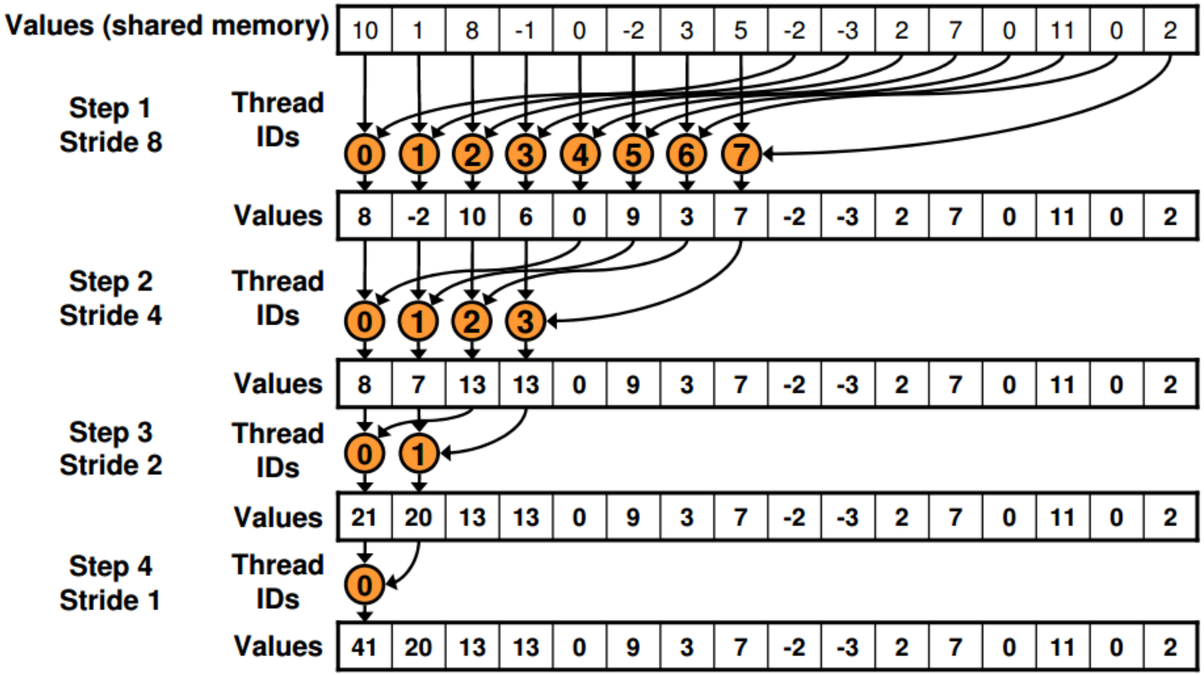
Shared memory reduction

1. Load the data into shared memory
2. Perform local reduction per thread block, keeping many **active** threads
3. 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]);
}
```

See: reduction_shared_1.cu

Shared memory reduction (sequential addressing)



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

Shared memory reduction

1. Reduce subset of elements per thread
2. Reduce shared memory with log performance
3. Single atomic add in global memory

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

    __shared__ int sharedMemory[256];

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

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

        sharedMemory[threadIdx.x] = localSum;

        __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]);
    }
}
```

See: reduction_shared_2.cu

Sorting: order an array of keys whose elements are comparable

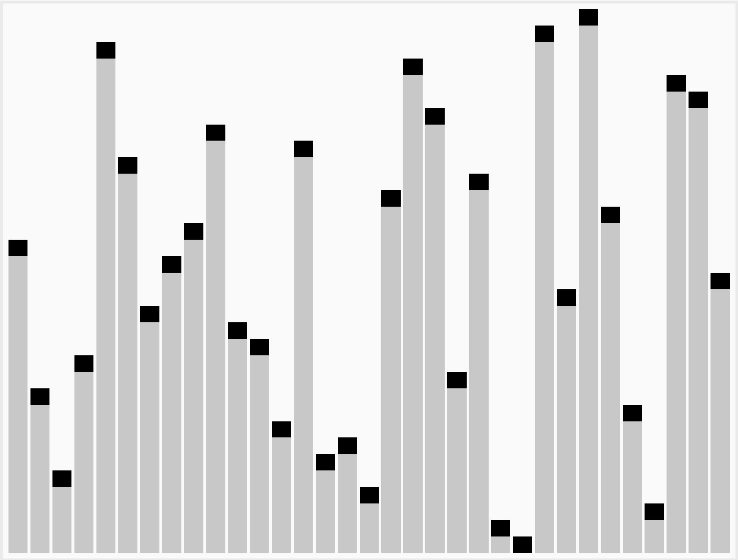
- Internal (in-place) vs external (require extra memory)
- Stable: maintain the relative order for equal keys
- Recursion: divide and conquer

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!

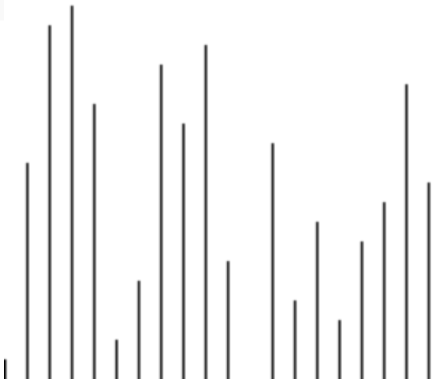
```
quicksort(A, lo, hi)
  if lo < hi
    p ← partition(A, lo, hi)
    quicksort(A, lo, p - 1)
    quicksort(A, p + 1, hi)

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
```

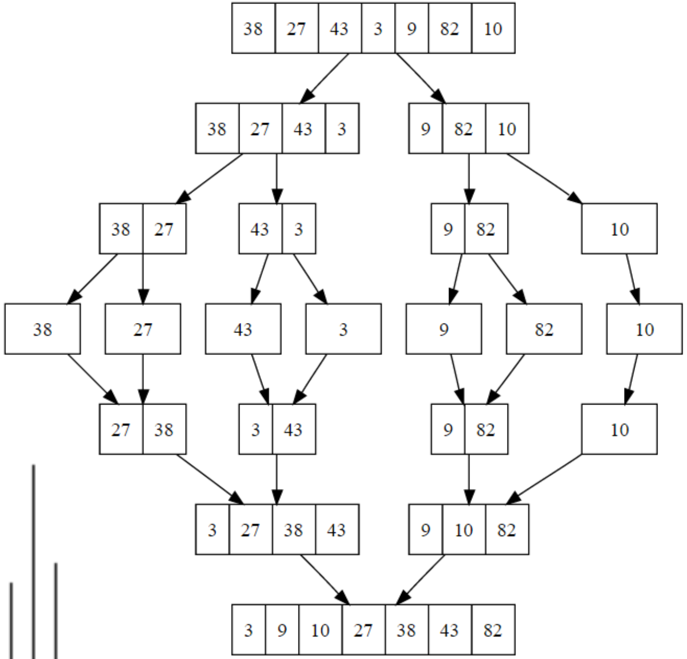


Merge sort

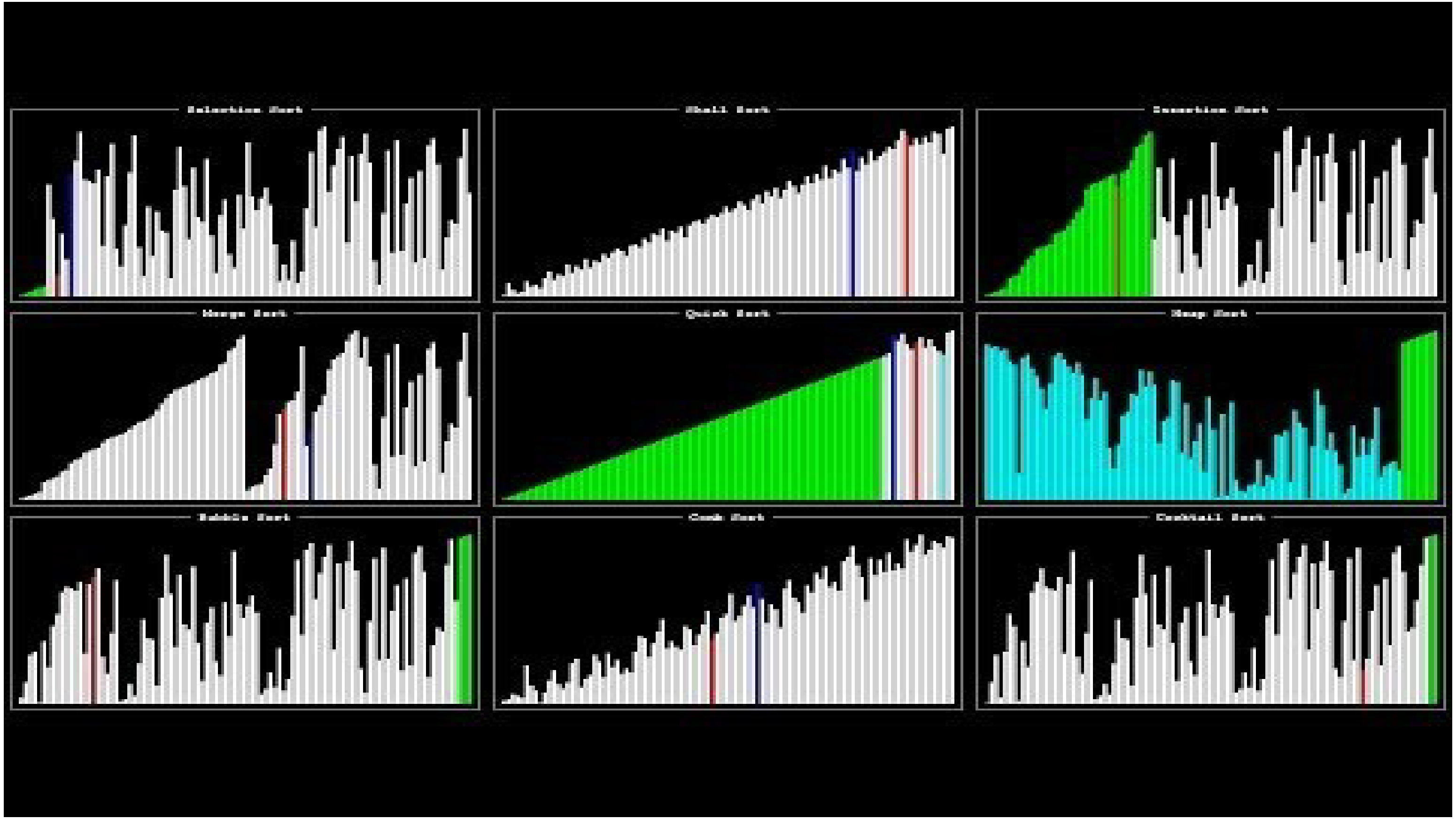
- Divide and conquer, completely parallelizable!

```
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)
```



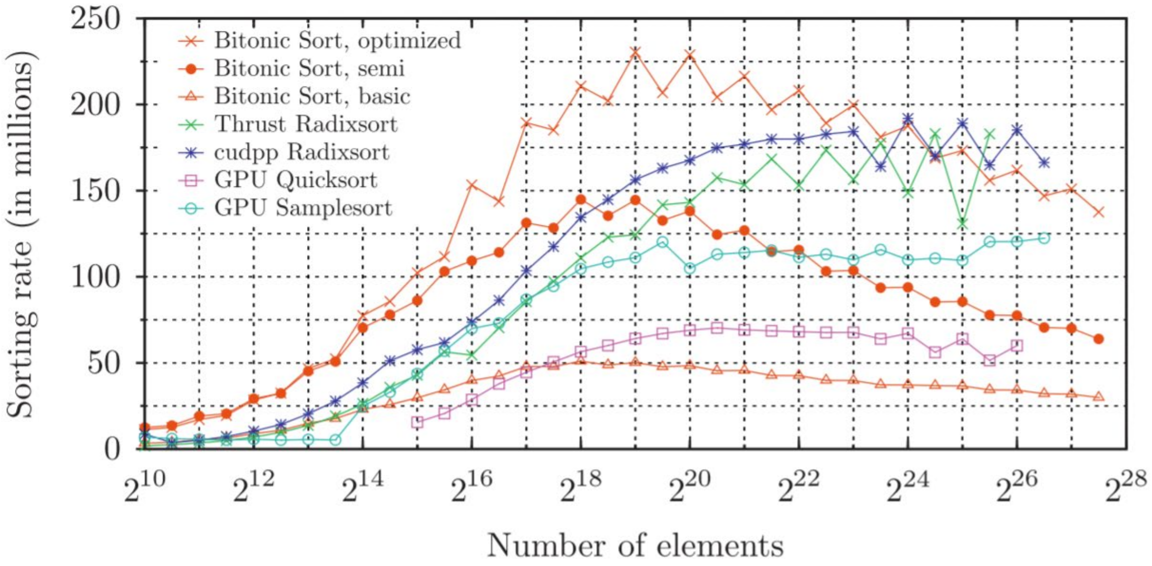


Comparison of sorting methods



GPU parallel sorting

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



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
- **See**: thrust_reduce.cu, thrust_sort.cu, thrust_sort_by_key.cu

Assignment 2

- Implement the KNN algorithm on a GPU using CUDA
- Compare the performance of the sequential, threaded, MPI and GPU versions
- Evaluate the speedup according to the data size
- Results must be the same in the CPU and GPU version
- Dataset must be copied into the GPU memory only once in the beginning
- Use float arrays to represent the data (do not use C++ templates / classes)
- If you have additional time, compare the performance when the dataset is transposed or multi-GPU using MPI

CMSC 603 High-Performance Distributed Systems

CUDA reduction and sorting

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

VCU College of Engineering