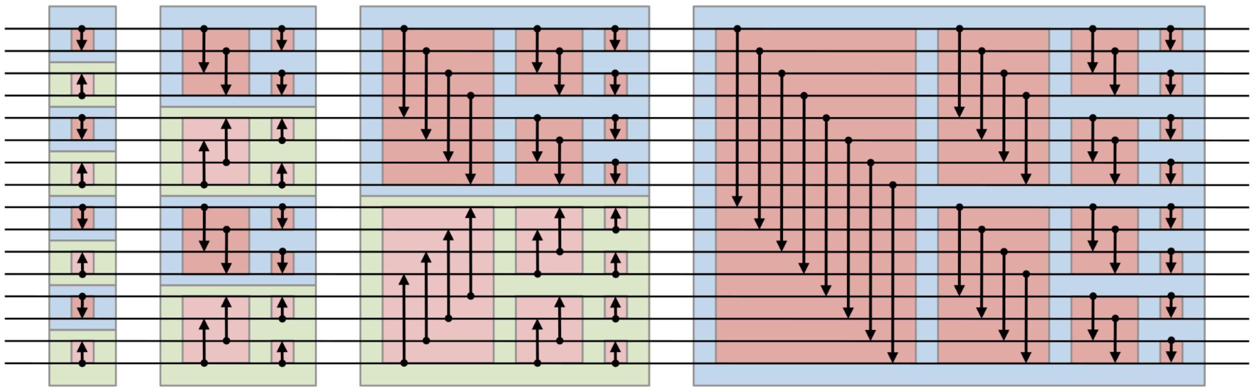
CS 8803 GPU Hardware and Software Project 2 - Parallel Sorting

Abhishek Sharma
asharma779@gatech.edu

In this assignment, I was tasked with implementing a fast parallel sorting algorithm. Bitonic sort is an example of a sorting algorithm that can be parallelized. It compares pairs of elements and swaps them to achieve an increasing or descending order depending on their position in the overall array. The graphic below shows each phase of the sorting algorithm, with blue and green colors denoting whether the comparison is happening to achieve ascending/descending order. The red boxes and arrows within show which elements are being compared with each other.


Source 1: Bitonic Sort Visual

Each vertical stack of red rectangles represents an operation which can be parallelized. This is because each pair of elements within a stack that are being compared is disjoint from the other pairs in that stack. This means that the number of sequential operations is proportional to $1 + 2 + ... + log_2 N$, where N is the length of the array. This can also be written as $O(log^2 N)$.

Implementation Details

My implementation initially followed the bitonic sort pseudocode from the project spec, but included a few optimizations.

1. Thread/Block Level Parallelism

For starters, I implemented block and thread level parallelism. This was done by taking the inner loop of the bitonic sort algorithm and setting k to the current thread id.

```
for i=1 to (log n) do
  for j=i-1 down to 0 do
    k = threadid.x #parallelize looping through array
    a = arr[k]; b = arr[k XOR 2j];
    if (k XOR 2j) > k then # (a,b) are compared so skip (b,a) case
      if (2i & k) is 0 then
        Compare_Exchange↑ with (a, b)
      else
        Compare_Exchange↓ with (a, b)
  endfor
endfor
```

This would ensure each thread was focused on a single comparison operation at each step of the bitonic sort. The way I made sure there was an exact mapping between each thread and each element of the array was by doing a few things:

- I padded the array with zeros so its length would be a power of 2.
- I used a predefined number of threads, 512.

- I calculated the number of blocks by dividing (Length of array) / 512. Since the size was a power of 2, any size 512 or more would perfectly be divided by 512. For smaller sizes, I set the number of threads directly to the length of the array and used 1 block.

This 1:1 thread:array element mapping helped me ensure every thread was focused on a unique array element. As a result, I didn't need to use any conditional statements to check if the current thread and block id composed a valid index, my pre-calculations guaranteed this.

2. Using Shared Memory As Much as Possible

For the middle loop (j = i-1 down to 0), we can use shared memory whenever j is less than $\log_2 S$, where S is the shared memory limit for a block. This is because each element will only be compared with elements that are at most $\log_2 S$ indices away. So, we can break the array up into chunks that are $\log_2 S$ long and perform the corresponding step of the bitonic sort independently for each chunk. We can also load each chunk into a block's shared memory so that it can be used by all of the threads from that block. This provides a significant speedup since shared memory accesses and updates are a lot faster on GPUs than those with global memory.

3. Using Pinned Memory for Host Arrays

The initial arrays allocated on host memory (arrCpu and arrSortedGpu) were initially allocated using *malloc*. By changing this to use the *cudaMallocHost* method, I was able to significantly improve data transfer times between host and device. When we use *cudaMallocHost* instead of *malloc*, we allocate the memory as pinned or page-locked, which is done in the background when we *malloc* and then transfer from host to device. This saves us time when transferring the memory from host to device, since the memory is already pinned.

4. Parallelized Padding

Though I was initially creating a large array using *memset* to pad it on the host, I realized I could reduce transfer times by performing padding on the device and only transferring back the initial elements from the original array without the padding. I did this by implementing a simple padding kernel function. This would allow each thread to set a different element of the input array on the device to zero (in the indices where padding was needed).

Throughput and Occupancy Optimizations

In order to maximize occupancy and throughput, I took a few extra steps to optimize my program:

1. Reducing Divergent Branches

I made sure to combine and remove if statements where possible. For example, when deciding to swap elements, I wrote the conditional as a single if statement instead of breaking it up into two separate ones like the pseudocode does. I was also able to remove any if statements that check if the current index (blockIdx*blockDim + threadIdx) is in bounds by setting the number of blocks and threads to add up to the exact number of elements in the array.

2. Coalesced Memory Accesses

I ensured that the threads in a block were accessing adjacent array indices when loading and writing values from/to global memory.

3. Experimenting with Thread/Block Dimensions

I initially set the number of threads per block to 1024, a higher bound for GPUs like the NVIDIA H100. After reducing the number to 512, I noticed an increase in memory throughput. This is likely due to the resulting higher number of blocks, which would increase the number of SMs active at a time and therefore more memory bandwidth utilization.

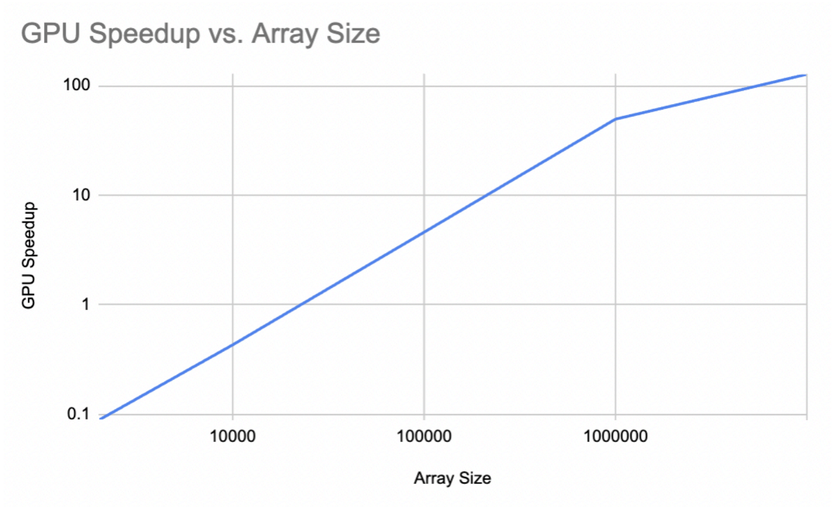
Performance Results

When running grade.py, my code achieved the following results:

- Achieved Occupancy: **74.81%**
- Million elements per second: **830.2029148756421**
- Memory Throughput: **55.58%**

Some more performance metrics from a test run with 10M elements:

- CPU Sort Time (ms) : 1538.909058
- GPU Sort Time (ms) : 12.545025
- GPU Sort Speed : **797.128723 million elements per second**
- GPU Sort is **122x** faster than CPU !!!
- H2D Transfer Time (ms): 1.006048
- Kernel Time (ms) : 10.556192
- D2H Transfer Time (ms) : 0.982784



The chart above shows the speedup by array size on a log scale. A linear chart wouldn't do justice to the comparative speedups from an array size of 2 thousand to 10 million. As the array size grows, the benefit of using a GPU for sorting becomes larger and larger.

Extra Optimizations

In addition to major optimizations listed above, I made a few smaller optimizations to gain performance improvements:

1. Replace Powers/Multiplication with Optimized Operations

There were some cases where I needed to do multiplication or exponent operations, like computing array memory sizes and padded sizes. Whenever possible, I replaced multiplications and exponent operations with bit shifts.

2. Builtin Methods for Logarithm

I used the *__builtin_clz* method to get the $\log_2$ of the array length (supposed to be faster than math.log for powers of 2).

3. CUDA min function for padding index

To ensure the padding function was using an in-bounds index, I used the CUDA min function instead of an if statement

Sources

1. By Bitonic at English Wikipedia - Own work, CC0, https://commons.wikimedia.org/w/index.php?curid=21961929
2. Ed Discussion Comment by Scott allowing changing the memory allocation. https://edstem.org/us/courses/61750/discussion/5189186?comment=12422753