CS 8803 HW 2 Report

Kirin Kambon (kkambon3)
Georgia Institute of Technology

I. Introduction

A. Problem Description

Bitonic sort is a sorting algorithm that leverages a series of compare-and-swap operations arranged in stages to create and then sort bitonic subsequences. These are sequences that only increase until a point and then only decrease, where either side is allowed to be empty. The algorithm is traditionally done recursively, but when unrolled into a serial version, it is an incredible candidate for parallelization. When mapped onto a GPU you can take advantage of parallel execution, although there are various optimizations required in the implementation to fully utilize the GPU abilities.

My approach consists of the following steps:

- **Host setup and memory transfers**: Allocate and initialize the data on the CPU, then copy to the GPU.
- **Shared memory kernel**: Sort the segments within each block using shared memory to reduce global memory traffic.
- **Global memory kernel**: Complete the Bitonic steps across the entire array of sub-Bitonic arrays.
- **Comparison with CPU**: Measure and compare performance with `std::sort`.

This combination of shared-memory sorting for each tile, followed by a global merge phase, results in a fully sorted array on the GPU, which I then transfer back to the host to validate correctness.

B. Implementation

I define a macro `group` that determines how many elements each thread processes in shared memory. The kernel named `bitonicSortShared16` copies chunks of `group * tileSize` elements into shared memory, performs an in-block bitonic sort, and writes data back out to global memory. This reduces repeated global memory accesses.

After the per-block sorting step, the kernel `bitonicSortStepBranchless` completes the merging phase of the bitonic sort across the entire array. It is launched multiple times, increasing the distance (`k`) and reducing `j` until the entire array is fully sorted. This kernel avoids conditional branches by using `min` and `max`.

II. Performance - Base and Optimized

A. Implementation of Optimizations

I tuned various parameters in my implementation. This included group macro (4, 8, 16) which meant balancing how much sorting is done per thread in shared memory. Another optimization was to implement Pinned (Page-Locked) Memory on the host as this allowed faster transfers between CPU and GPU, since page-locked buffers avoid OS page faults. I also changed the types to `uint16_t`. This switch from `int` to `uint16_t` reduced the overall data usage, which in turn sped up both host-to-device and device-to-host copies.

To improve transfer efficiency, I also used *asynchronous* memory copies (`cudaMemcpyAsync`) in a CUDA stream. This enables overlapping data transfers with kernel execution, thus hiding part of the communication overhead behind computation. I also used compiler hints like `__restrict__` on pointer parameters to inform the compiler that these pointers do not alias other memory references. I also used unrolling loops (e.g., via `#pragma unroll`) to reduce loop overhead sections.

An important optimization I implemented was the use of `i` and `ixj` indices:

```
unsigned int i = t / j * 2 * j + t % j;
       unsigned int ixj = i ^ j;
```

This computes paired indices for the compare-and-swap steps without extra condition checks, thus merging multiple arithmetic operations into a single formula. Then, by using `min` and `max` instead of if-statements (i.e., a *branchless* approach), I was able to remove divergent branches that could harm warp execution efficiency. This also resulted in half as many threads needing to be launched.

I also tuned the Thread block size (256, 512, 1024) which determines the best occupancy and performance in the global merge kernel. Table I provides the results of this experiment. 256 most likely underutilized SMs, and 1024 had fewer blocks, potentially resulting in mismatched scheduling. 512 threads gave the best overall performance, properly balancing occupancy and overhead on this particular GPU.

TABLE I
Global Kernel Performance for Different Thread Block Sizes

Threads/Block	Time (ms)
256	15.44
512	8.76
1024	13.974

B. Results Performance

To analyze the performance I collected Nsight (`ncu`) data as well as ran it on the H100 GPU for varius array sizes: **100,000**, **1,000,000**, and **10,000,000**. The results are shown in tables below.

The following table shows the performance data when run on the H100 GPU. The larger the array size the larger the speedup was for the GPU sort in ms. This demonstrates that the Bitonic sort algorithm is optimal for large input data arrays when compared with the GPU sort.

TABLE II
PERFORMANCE DATA ON H100 GPU

Array Size	CPU Sort (ms)	GPU Sort (ms)	Speedup
100k	12.127	4.799	2x
1M	138.363	5.718	24x
10M	1463.613	11.429	128x

Table III below shows a comparison using Nsight for the The Shared Memory Kernel (`bitonic Sort Shared 16`). It shows that when launched with 4096 blocks of 1024 threads each, totaling over 4 million threads, it achieved approximately 98% occupancy, in contrast to around 80% for 1 million elements and 50% for 100,000 elements. The larger problem size provides enough workload to keep nearly all SMs fully busy. Memory throughput is about 76%, with L1/TEX cache usage around 78%. Similar to the 1M case, this indicates heavy reliance on shared memory and L1 caching, limiting frequent, high-volume DRAM accesses (reported at only about 1%).

TABLE III
COMPARISON OF `bitonicSortShared16` FOR DIFFERENT INPUT SIZES (FROM NSIGHT COMPUTE).

Input Size	Grid Size	Block Size	Achieved Occupancy	Mem. Throughput	Duration
100k	32	1024	49.70%	16.04%	49.73 µs
1M	256	1024	80.05%	73.65%	90.02 µs
10M	4096	1024	98.09%	76.19%	1.30 ms

Table IV below shows the results for the Global Merge Phase (`bitonic Sort Step Branchless`). It was invoked 26 times with 16,384 blocks of 512 threads each, totaling over 8 million threads. It achieved occupancy around 76%, slightly lower than the 80% seen at 1 million elements, but still high enough to feed the GPU with plenty of concurrent work. The memory throughput is about 50%, and DRAM throughput is similarly measured at 50%. The merge step requires more global memory operations than the shared-memory kernel, but high occupancy helps sustain efficiency.

TABLE IV
COMPARISON OF `bitonicSortStepBranchless` FOR DIFFERENT INPUT SIZES (FROM NSIGHT COMPUTE).

Input Size	Grid Size	Block Size	Invocations	Mem. Throughput	Duration (avg)
100k	128	512	19	10.33%	3.69 µs
1M	1024	512	18	18.07%	5.10 µs
10M	16384	512	26	50.00%	9.10 µs

Comparing all three sizes (100k, 1M, 10M) reveals a classic scaling pattern: 100k had insufficient blocks/threads to saturate the GPU, leading to low occupancy (24–50%) and modest throughput. 1M had enough parallel workload to reach about 80% occupancy, substantially higher throughput, and efficient cache usage. 10M had near-max GPU utilization (up to 98%) in the shared-memory sorting phase and high concurrency overall. Although global-memory-intensive steps still occur, the GPU is kept highly active throughout sorting.

III. CONCLUSION AND FINAL OBSERVATIONS

The tuning of `group` and thread block sizes is crucial: larger `group` values and extremes in block sizes can negatively impact performance. For shared memory, in-block sorting reduced global memory bandwidth usage but needed careful tuning. 512 threads per block yielded consistently high occupancy and performance, outperforming both 256 and 1024 in testing. For smaller problem sizes, the overhead of kernel launches and data transfers can exceed the benefit of GPU parallelism. By combining shared memory bitonic sorting with global memory merges, I achieve a fully sorted array on the GPU.

IV. CITATIONS

No external citations were required for this project beyond course materials.

V. FUTURE WORK

Some options for next steps could include a dynamic approach, use CPU for smaller arrays, GPU for larger arrays and Warp-Level Primitives, including shuffle and warp intrinsics. Another option could be a Multi-GPU approach which would include scaling out to multiple GPU's. I could also attempt to Integration with Thrust to compare performance and overheads against `thrust::sort`.