Sort Hard: GPU Edition

Kyu Yeon Lee
College of Computing
Georgia Institute of Technology
kyuyeon.lee@gatech.edu

***Abstract*—How can GPUs bring the heat to sorting? While most algorithms are crafted with CPUs in mind, this report dives into how GPUs can handle the heavy lifting. We revisit and profile the performance of a suboptimal $O(n \log^2 n)$ sorting algorithm—bitonic sort—and explore how CUDA unleashes its full potential.**

***Index Terms*—GPU, sorting, CUDA, shared memory**

I. Introduction

Sorting is one of the most fundamental problems in computer science. There are CPU-optimized sorting algorithms, such as quicksort or mergesort, where the performance gains become marginal when applied to GPUs. However, bitonic sorting, which has a suboptimal performance of $O(n \log^2 n)$ on CPUs, can take advantage of the massive parallelism of modern GPU architectures. NVIDIA GPUs also offer programmable cache memory, called shared memory, which can further optimize performance by reducing memory latency. Here, the performance uplift of each implementation using shared memory is described.

The goal is, first, to write a parallelized version of bitonic sorting in CUDA, and second, to improve performance by using shared memory and lastly, apply other optimization methods including pinned memory and register allocation. The implementation for shared memory application consists of five phases: bitonic sorting using only global memory, basic usage of shared memory, using shared memory except for bitonic merge, except for intermediate merge, and full usage of shared memory. Each phase is numbered from 0 to 4.

Performance evaluation is conducted on NVIDIA L40S GPUs in the PACE-ICE environment provided by the Georgia Institute of Technology.

II. Background

A. Bitonic Sorting

Bitonic sorting has a recursive nature where it bitonically sorts the two sub arrays in ascending and descending order, and then merges the two arrays int a sorted array. This can be also written in an iterative fashion as shown in Algorithm 1. The last for loop where it iterates from 0 to n can be parallelized by a GPU kernel. [1]

This can also be visualized as a sorting network as 1. Each horizontal wire represents an element of the array to be sorted, and the arrow describes a swap operation which points to the larger element.

Algorithm 1 Pseudocode for iterative bitonic sorting

```
for k = 2; k ≤ n; k ← k × 2 do
    for j = k/2; j > 0; j ← j/2 do
        for i = 0; i < n; i ← i + 1 do
            l ← bitwiseXOR(i, j)
            if l > i then
                if bitwiseAND(i, k) == 0 then
                    if arr[i] > arr[l] then
                        swap arr[i] and arr[l]
                    end if
                else
                    if arr[i] < arr[l] then
                        swap arr[i] and arr[l]
                    end if
                end if
            end if
        end for
    end for
end for
```

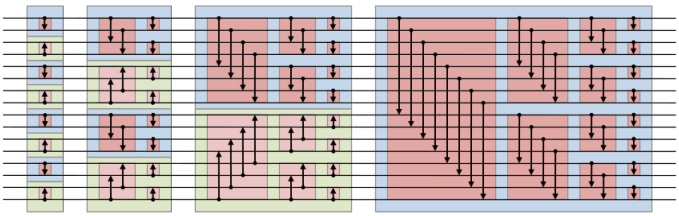

Fig. 1. Visualized sorting network of bitonic sort [1]

B. CUDA Shared Memory

Shared memory in CUDA is a programmable cache, where it has much faster access time compared to global memory. The basic idea for utilizing such memory is to keep the frequently accessed data within the shared memory to avoid costly global memory accesses. Bitonic sorting can leverage shared memory since the working set of the sorting is straightforward. The parallelizable region in algorithm 1 is applied to only a fixed, sequential region of the array. Our goal is to fit this part in shared memory as much as possible.

III. Kernel Implementation and Optimization

The following describes the implementation and performance for each phase of optimization. The result for sorting $10,000,000$ items is shown at Fig. 1. A trend in decreasing kernel time can be observed, but for phase 1.

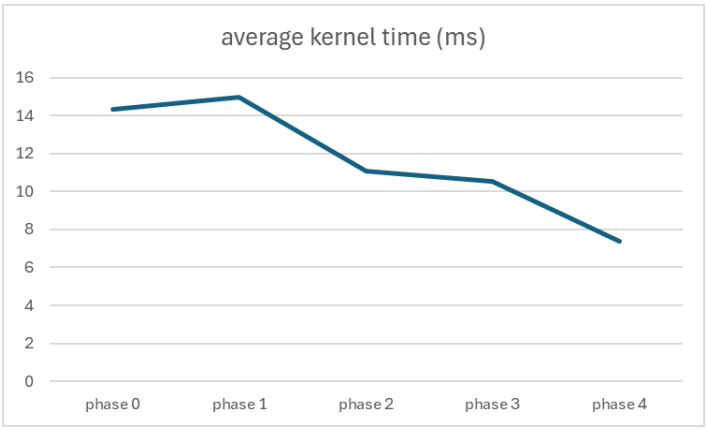

Fig. 2. Kernel time for sorting $10,000,000$ elements for each phase of implementation

A. Phase 0 - Using Only Global Memory

The initial implementation has the parallelizable region translated to CUDA code. The red boxes enclosing the swap arrows in 1 are the parallelizable regions, which can be done in a single kernel call without repetition within. This results in a complexity of $O(log^2 n)$ kernel launches, where if the size of the input is $16,777,216 (= 2^{24}$, results in 299 kernel launches. Given that global memory access is costly and that kernel launches are unnecessary overhead, this does not have optimal performance.

B. Phase 1 - Basic Usage of Shared Memory

Here, the implementation was to use shared memory in the kernel, but not changing the number of kernel launches. That is, modifying only the kernel in III-A. This simply loads the numbers to be compared into shared memory, compare the numbers, and store them into global memory. Such implementation only shown an increase in the total kernel time, where it is pure overhead. In III-B, the numbers to be compared are likely to be stored in registers, but this implementation just adds an unnecessary intermediate step in the shared memory. This results in degradation of performance, an overhead of 4% increase in kernel time.

C. Phase 2 - Using Shared Memory Except for Bitonic Merge

Given that III-B results in adding overhead, restructuring the iteration is required to reduce CUDA launches and to make a single kernel access shared memory more frequently. The first idea is to implement a CUDA kernel that performs end-to-end sorting if the given array (or subarray) fits in the shared memory. Consider the case in 1. Suppose we have shared memory of 4 integers per thread block. Our goal is to fit the first two rows of blue/green boxes in 1 in a single kernel, allowing it to not access the global memory for sorting. Also note that for this implementation, the kernel should also be able to decide if the given array is to be sorted ascending or descending. This can be identified by the `blockIdx.x` value, where if it is ascending for even and descending for odd. This implementation shows 25% less kernel time where global memory access are significantly reduced. The kernel launches are also reduced to 234 calls, where the shared memory implementation can do the work for the first 66 kernels in III-A and III-B.

D. Phase 3 - Using Shared Memory Except for Intermediate Merge

As III-C have shown promising results, shared memory can be used for the later part of the algorithm, bitonic merging. Notice the rightmost blue region in 1. Each parallelizable region is subdivided into two identical problems of smaller size. The identical ascending sort also makes the implementation easier, which does not require the decision of ascending/descending sort within the kernel. This optimization shows about 5% of reduce in kernel time and a reduction of 10 kernel launches.

E. Phase 4- Fully Utilizing Shared Memory

Considering the idea from III-D, we can also apply the shared memory usage to intermediate merges. By intermediate merges, it is visualized in 1 as the smaller, but identical shapes of the rightmost region in blue. However, the green regions indicate an ascending sort, where the decision should be addressed in the kernel. This can be evaluated by using the global thread ID masked by the size of the input that the thread block is working on. This reduces the kernel time 30% compared to III-D, which is the greatest observed, and reduces the number of kernel launches to 119 for input of size $16,777,216$. This is a speedup of nearly $2\times$ compared to the initial, global only sorting and up to $77\times$ speedup compared to CPU based quick sort.

IV. Profile Results

This workload can be divided into three kernels. The first one is a global bitonic merge kernel `bitonicMergeGPU()` which has no iterations within. Note that this is identical to the initial implementation in III-A which doesn't utilize shared memory. `bitonicSortGPUshmem()` is sorts a subarray which fits in the shared memory, and `bitonicMergeGPUshmem()` performs a bitonic merge which fits in the shared memory. We can gain some insights from the profile results.

A. Compute Throughput

The two `bitonic*shmem()` kernels show a relatively higher compute throughput, ranging from 61% to 75%, compared to `bitonicMergeGPU()`, about 0.16. This is because the shared memory kernels operate on a working set of better locality and iterates, whereas the global kernel does only a small number compare operations without any iterations.

B. Memory Throughput

Memory throughput shows the opposite of compute throughput, where the global kernel. Another interesting point is that all three kernels show similar L1, TEX and L2 hit rates. This is because shared memory does not go through the mentioned caches but has an independent path. Therefore,

most of the cache misses are compulsory misses, since this is a streaming application. In terms of memory accesses, the three kernels should not show much difference, but the shared memory kernels work on the shared memory for a longer time, not accessing the DRAM until the sorting is done.

C. Register Pressure

Register usage is another item for optimization, which has notable impact on thread occupancy because it can be a deciding factor on whether the warp can be scheduled. The initial profile result indicates that register usage for `bitonicSortGPUshmem()` requires 62 registers, which was a major bottleneck for thread occupancy. By applying optimization techniques described in the later sections, this was reduced to 20, resulting in higher $(90 + \%)$ occupancy.

D. Duration

The shared memory bitonic sorting takes the longest time overall of 1.29, and the two other kernel take approximately 0.17 and 0.10, the non shared memory taking the least due to not having iterations. `bitonicSortGPUshmem()` takes less iterations than `bitonicSortGPUshmem()` since it is a subset.

V. Optimization

A. Thread Block Size Tuning

Thread block sizes decide the size of shared memory usage, since this implementation uses 2 times the size of the number of threads. Larger thread blocks give more parallelism, however, also may limit occupancy since more threads in a block require more resources. Therefore, the size should be tuned based on the results, and thread blocks of size 256 shows the best performance balancing parallelism and occupancy.

B. Pinned Memory

Outside of kernel optimization, pinned memory [2] may be used to optimize device-host transfers. Pinned (page locked) memory, is memory such that the operation system does not swap out. This also bypasses memory management layers, which lets direct access to the system memory. Existing host memory can be pinned by `cudaHostRegister()`, and this can be placed after the kernel launches for the task to overlap with compute. This results in the runtime of ‘cudaMemcpy()‘ is significantly reduced as described in Figure 3.

C. Register Usage

Registers are limited resources that are critical for the performance of compute kernels. This includes common programming language techniques such as reducing number of intermediate variables, recalculating instead storing, reducing control logic, etc. In this example, applying such practices greatly reduced register usage from 40 to 20, which shown by Nsight Compute, contributing to higher thread occupancy.

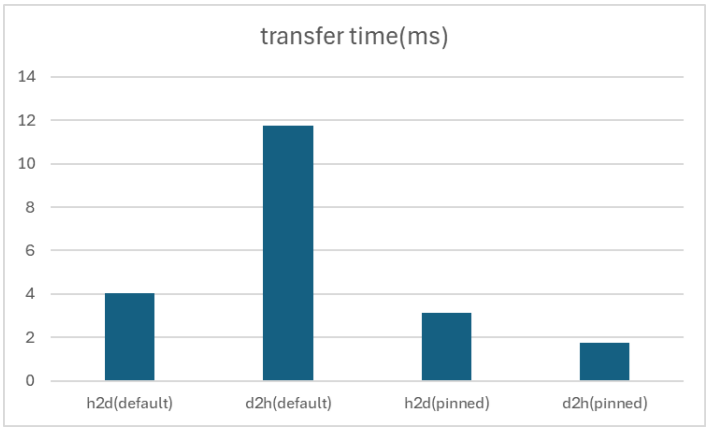

Fig. 3. Performance of pinned memory of transferring $10,000,000$ integers

VI. Conclusion

Bitonic sorting can be $100\times$ faster compared to CPU sorting, and shared memory, register optimization, and pinned memory has considerable contributions to the speedup since sorting is a memory bound workload.

VII. Further Work

Nvidia L40S GPUs provide up to 7.125KBs of shared memory per thread block. However, only 1KB of shared memory per block is used in this study. This is because each thread operates on one ascending/descending bitonic sort/merge. The 1KB shared memory is bound by the number of threads used for maximum performance, which is 512 threads per block. It would be an interesting approach to load as much as data possible in the shared memory and modify the kernel to handle multiple bitonic sorts, but the complexity of the implementation being high, this is left as a further item.

References

[1] Wikipedia Contributors, “Bitonic sorter,” Wikipedia, The Free Encyclopedia, https://en.wikipedia.org/wiki/Bitonic_sorter, accessed Sep. 22, 2024.

[2] Mark Harris, “How to Optimize Data Transfers in CUDA C/C++”, NVIDIA DEVELOPER Blog, https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/