CS 8803: GPU Hardware & Software

Project 2 Report

Introduction

This project explores the implementation and optimisation of bitonic sort on the NVIDIA CUDA platform. The goal is to leverage the GPU's computational power and memory hierarchy to develop a sorting solution that significantly outperforms sequential CPU-based sorting methods for large datasets.

Implementation Details

The algorithm begins by having each thread sort a `BLOCK_SIZE` segment of the array entirely within shared memory using the `bitonicSortInitialShared` kernel. For subsequent stages where comparison distances exceed the block size, the `bitonicSortGlobal` kernel is used to perform compare-exchange operations directly on data in global memory. Once a stage's comparison distance becomes small enough to again fit within a block, the `bitonicSortShared` kernel is called to complete the stage's remaining steps in the faster shared memory.

The shared memory kernels implement a branchless compare-exchange by having inactive threads operate on a separate dummy array to ensure all threads in a warp follow the same execution path. The `bitonicSortGlobal` kernel is structured to have each thread manage multiple data elements. A thread calculates the indices of several distant elements, loads them into registers, performs a sequence of compare-exchange operations on the register values, and then writes the sorted results back to global memory in a single pass.

Concurrency is managed by dividing the input array among `NUM_STREAMS` CUDA streams. Initially, and for any stage where comparisons are contained within a single stream's data partition, these streams execute their respective kernel launches in parallel. When a sorting stage requires comparisons between elements residing in different streams, a synchronisation scheme takes place. The streams are grouped, and within each group, a designated "leader" stream waits for all other "follower" streams to finish their prior work by using CUDA events. The leader then executes the `bitonicSortGlobal` kernel on the entire group's data.

Performance Optimisation

The following are the most important optimisations that were done –

1. **Using shared memory:** If all data needed for several stages/steps fit into shared memory, the data is loaded from global memory once, written back once, but used several times. This is possibly the most important optimisation and it made a big difference to the performance of the implementation.

2. **Running initial stages of bitonic sort in a single kernel:** This eliminates multiple kernel launches for each stage or step. Shared memory usage makes this optimisation even more powerful, and it made a big difference.
3. **Reducing branches in kernels:** Branchless programming techniques were used, to reduce warp divergence and hence improve occupancy. There was moderate performance improvement from this optimisation.
4. **Performing multiple steps of global memory sorting in a single kernel:** The global memory kernel was by far the most problematic to optimise as it is difficult to coalesce the memory accesses. Therefore, running this kernel fewer times by doing more work per launch significantly improved performance.
5. **Using registers in global memory sorting:** In the global memory sort kernel, multiple data elements are loaded into registers, all compare-exchange operations are performed on these registers, and the results are written back once. There was moderate performance improvement.
6. **Streams:** Multiple CUDA streams were used to overlap data transfers with various computation and maximise GPU utilisation. This on its own did not significantly improve performance, but it enabled other optimisations that did.
7. **Pinned memory/registering input host buffer in chunks:** `cudaHostRegister` was used to pin the input buffer. This results in faster transfer times as the pages for the buffer are guaranteed to not be swapped to disk. However, the call takes too long to run, so the trick here is to pin the buffer chunk-by-chunk, overlapping the pinning of the next chunk with the transfer of the current chunk to the device. This significantly improved H2D transfer times.
8. **Defining and registering result host buffer while sort is running:** The host buffer for the final sorted result is allocated and registered while the GPU is busy performing the sort, overlapping this CPU-side memory preparation with the GPU computation. There was significant improvement in H2D transfer times.

Performance Analysis

The sort was run on an Nvidia H100 GPU.

For an array of 10 million elements -

1. **Global memory accesses:** 230,686,720
2. **Local memory accesses:** 0
   This means there are no register spills to memory, which is good for performance.
3. **Divergent branches:** 0
   This means all threads in a warp execute in lockstep. This is expected because the only branches present in the kernels are loops (which are executed the same way for all threads) and branches based on input arguments to the kernels (which is the same for all threads, therefore execution happens the same way as well). Importantly, the ternary operators used in the kernels are optimised to predicated code.
4. **Memory throughput:** 64.50%
   Could be better – a major reason why this is not great is due to the global kernel

naturally lending itself to strided (and thus non-coalesced) access patterns. Improving this is future work.

5. **Occupancy:** 88.03%
   Very good due to zero branch divergence, as discussed.

For an array of 100 million elements, the sort speed is 1266 million elements per second (233x faster than CPU sort on the same PACE server). H2D transfer time is 15.4ms, D2H transfer time is 7.2ms and kernel time is 56.3ms.

Conclusion

The implementation achieves most of its desired objectives. Future work would be primarily focused on improving memory throughput, as that is where the bottleneck is. Research would be done on ways to improve memory coalescing for the global sort kernel.

References

1. Bitonic sort (Wikipedia) - https://en.wikipedia.org/w/index.php?title=Bitonic_sorter&oldid=1234823357

2. The implementation and optimization of Bitonic sort algorithm based on CUDA - https://arxiv.org/pdf/1506.01446