Bitonic Sort goes fast

SENAD-LEANDRO LEMES GALERA, TECHNICAL UNIVERSITY MUNICH

1 OVERVIEW

In this project we will look deeply into how to optimize bitonic sort. First, we will optimize it with SIMD and Multi-Threading. Secondly, we will look at how to optimize it for a GPU. In the end, we will combine both approaches to a hybrid one and look at how we can utilize the different types of hardware.

2 BACKGROUND

Bitonic sort is a compare-swap based sorting algorithm. It works on bitonic sequences. A bitonic sequence is a set of numbers which monotonically increases until a certain value and then monotonically decreases (or the other way around). By recursively merging these bitonic sequences the input array will be sorted. The algorithmic complexity is higher than for other sorting algorithms with $O(nlog^2n)$. While quick sort e.g. only has a runtime complexity of $O(nlogn)$. However, bitonic sort can be exploited due to its high parallelization. The compare and swap operations are mostly independent and can be executed in parallel.

3 SIMD AND MULTI-THREADING

To exploit SIMD and Multi-Threading, we implemented two ways of bitonic sort. Based on two different papers. While the first implementation is based on Cell-Sort [2], which follows the actual bitonic sorting rules. The second implementation is rather an advanced merge sort, which exploits only small bitonic sorts and then uses a classical merge-sort approach on these small presorted runs [1].

3.1 SIMD-MERGE

In this section we will explain in more depth on how to implement the SIMD-MERGE algorithm. Our implementation consists of mainly two parts: an in-register sorting kernel and merging kernel. The main idea is to first presort small size runs using SIMD bitonic sort and then merging these small runs together until the array is sorted. All information provided in this section is based on the Paper from Chhugani et. al[1].

3.1.1 In-register sorting kernel. An in-register sorting kernel, is a way to exploit SIMD instructions to sort a set of values in parallel. In principal it is just a bitonic sort implementation on SIMD level using min/max and shuffle operations. In the paper a 4-wide SIMD register is used which can be seen in Fig. 1. Since our machine already can exploit 8-wide SIMD registers (AVX2), we extended the algorithm to an 8-wide SIMD. This 8-wide implementation increased our performance even further by 1.2x (comp. to 4-wide).

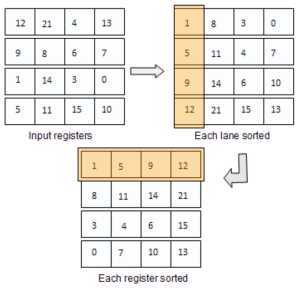


Fig. 1. 4-wide SIMD[1]

Author's address: Senad-Leandro Lemes Galera, Technical University Munich.

3.1.2 Merging kernel. We implemented the merging phase using a bottom-up approach. It can be seen as a width-doubling merge sort. We start with 8-wide sorted runs talked about in section 3.1.1. These runs are then recursively merged until the last two big runs are merged together such that the array is properly sorted.

3.2 Cell-Sort

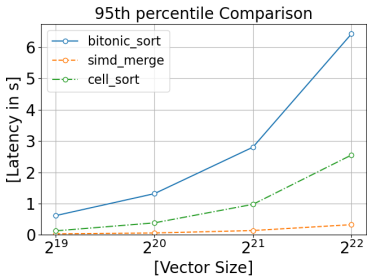
Our Cell-Sort implementation is based on the paper [2]. It follows the proposed two-tier approach. First, each core sorts a run which fits into its cache properly. Secondly, we exploit a distributed bitonic sort network to merge these phases.

3.2.1 Local phase: SIMD optimized sort. In this phase, we first split the input array into #Cores chunks. Then we use a 8-wide SIMD bitonic kernel to sort the runs. Threads with an even ID are sorted ascending and threads with an odd ID are sorted descending.

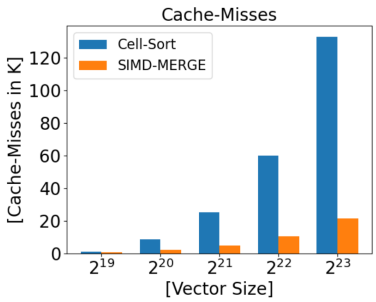
3.2.2 Global phase: Merge. The merging phase, is based on the classic bitonic merge network. For each element we find its partner (using xor) and swapping either ascending or descending.

3.3 Evaluation

In Fig. 2 (a) we can see the performance of the different approaches. We used the classic recursive bitonic sort implementation as a baseline. We can see that our SIMD-MERGE implementation outperforms the Cell-Sort by a magnitude of 7x. This is based on the fact that our SIMD-MERGE implementation is way more cache-friendly, which can be seen in Fig. 2 (b).



(a) Branch_free vs. Scalar vs. Vectorized



(b) SpeedUp vectorized against branch_free/scalar

Fig. 2. Cell-Sort vs. SIMD Merge

4 GPU OPTIMIZING

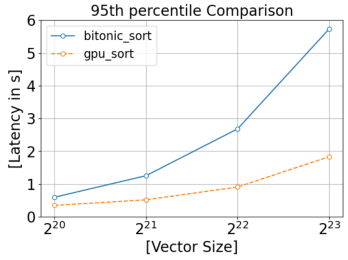
We also implemented a version which implements bitonic sort on the GPU. We implemented the code using opencl on a intel GPU based on a standard bitonic merge network.

4.1 Implementation

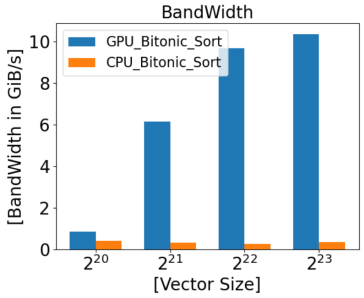
This implementation follows the idea of the merge network. It can be understood as a three step algorithm. (1) Launch kernel and sort based on size. (2) increase size. (3) go to (1) again. We repeat these steps until the whole array is sorted.

4.2 Evaluation

As a baseline we again used the recursive non-optimized bitonic sort implementation. We can see that our GPU implementation outperforms the basic approach by a lot, especially for larger vector sizes our performance becomes way better (Fig. 3 (a)). Additionally, we can see that it correlates with the bandwidth throughput, while for smaller vector sizes our bandwidth on CPU and GPU are fairly similar (Fig. 3 (b)), for higher vector sizes the GPU implementation has a much higher throughput, which also explains the speedup increase for larger vector sizes.



(a) GPU implementation vs. Baseline



(b) BandWidth CPU vs. GPU

Fig. 3. Cell-Sort vs. SIMD Merge

5 HYBRID APPROACH

In this section we will look at our hybrid approach. This hybrid approach uses our Multi-threaded CPU implementations for the small chunks and then uses our GPU implementation for the larger chunks.

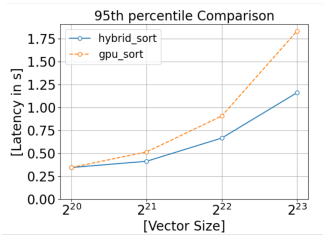


Fig. 4. Hybrid Approach

5.1 Evaluation

To evaluate our Hybrid approach with SIMD-MERGE as CPU Implementation, we used the normal GPU-Sort as a baseline. We can see that combining the SIMD-MERGE, with our GPU implementation improves performance (Fig. 4). A big problem, which occured was that our SIMD-MERGE implementation was way too fast such that our GPU implementation couldn't hold the pace. However, we can still see that our hybrid approach properly implements the switching since it outperforms the GPU implementation, even though the management overhead implemented for switching between the CPU and GPU.

REFERENCES

[1] Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, William Macy, Mostafa Hagog, Yen-Kuang Chen, Akram Baransi, Sanjeev Kumar, and Pradeep Dubey. 2008. Efficient implementation of sorting on multi-core SIMD CPU architecture. *Proc. VLDB Endow.* 1, 2 (Aug. 2008), 1313–1324. https://doi.org/10.14778/1454159.1454171

[2] Buundefinedra Gedik, Rajesh R. Bordawekar, and Philip S. Yu. 2007. CellSort: high performance sorting on the cell processor. In *Proceedings of the 33rd International Conference on Very Large Data Bases* (Vienna, Austria) *(VLDB '07)*. VLDB Endowment, 1286–1297.