Computer Architecture Homework 4: Sorting Optimizations

Team Members: Sri Iyengar (si2468), Harrison James Riley (hjr2128), Elijah Retzlaff (er3211), Noah Hartzfeld (nah2178)

Methodology:

Algorithm Selection:

- Our project explores two sorting algorithms: radix sort and merge sort, to try and find the most cost-efficient algorithm. We picked these two algorithms for their stability, performance on large datasets, as well as their differences relative to each other. MergeSort is a general-case sorting algorithm, while RadixSort is optimized for sorting numerical values, integers specifically. Because of this, we expect to see RadixSort perform better, but set out to build an optimized version of MergeSort that would outperform the more specialized RadixSort.

Data Structure Designs:

- For consistency and comparability, we used the same data structure (array) for both algorithms.

Processor and Memory System Optimization:

- We optimized both algorithms for performance using parallelization.
- We experimented with different GCP configurations.
- Additionally, we performed an exploratory CUDA implementation of merge sort combined with bitonic sort parallelized on a GPU.

Results and Discussion:

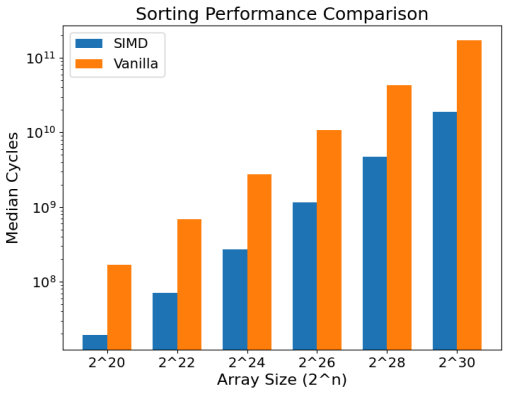
SIMD Vectorization of Radix Sort Using Intel AVX2 Intrinsics:

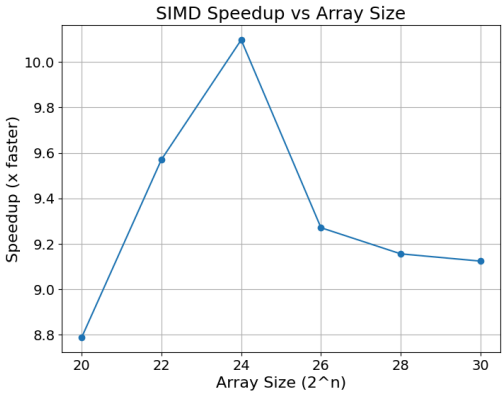
Algorithmic Improvements Made: Since the digit-level passes of radix sort are inherently non-parallelizable, we reduced the number of digit-level passes by using base-256 radix instead of base-10 for the sorting. This allowed us to radix sort by byte rather than by digit, working with the natural structure of our 32-bit binary numbers such that we only have to do 4 sequential digit-level passes rather than the the variable number of passes needed for base-10 which can be up to 10 passes for a full 32-bit number. Next we parallelized the extraction of bytes (not digits anymore since we are using base-256) in the histogram creation process, using SIMD intrinsics to load 8 uint32_t integers from the array at a time, mask them, and extract the current byte from them. We parallelized the extraction of bytes in the same way again in the sorting/scattering of elements into their corresponding buckets.

Data Structure Improvements Made: In the vanilla radix sort the histogram used is a regular array while in the SIMD radix we used 32-byte aligned arrays both for better compatibility with AVX2 intrinsics (which operate on 256-bit/32-byte vectors) and for improved memory access patterns. Since cache lines are 64 bytes, aligning our data to 32-byte boundaries ensures that our SIMD operations never split across cache lines, reducing the number of cache loads needed and improving performance.

These improvements lead to a roughly 9x speedup for sizes of approximately 4GB. The data in the SIMD Speedup graph was collected from 10 runs of each radix sort (SIMD and vanilla) at each array size from 2^20 to 2^30. The speedup between the vanilla and SIMD radix sort starts at about 8.8x and then peaks to about 10.1x at a size of 2^24 elements. It then drops down to about 9x at a size of 2^30. This drop is likely due to the use of two arrays for tracking histogram and then the mapping of buckets in the SIMD radix sort versus only using one in the algorithmically simpler vanilla radix sort. This means

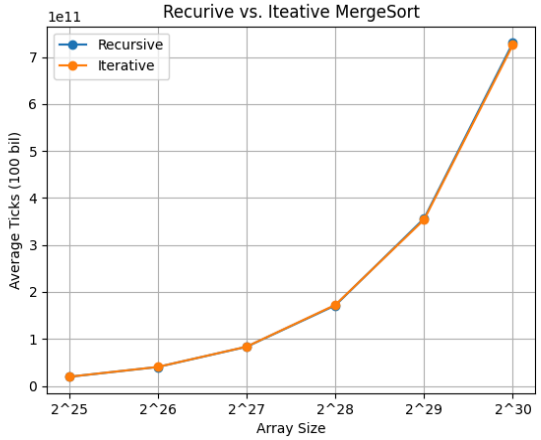
that the SIMD sort incurs slowdown from cache-line conflicts, which become more pronounced as dataset size exceeds L2/L3 cache limits, sooner than the vanilla radix. Data was collected on AMD EPYC 2 vCPU (2 cores) Memory 16 GB.

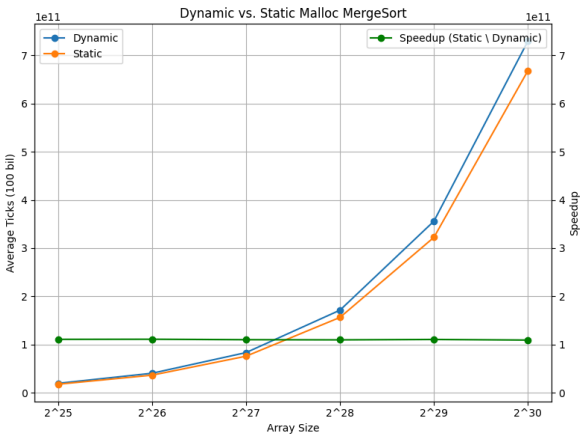




MergeSort (CPU):

Data collected on AMD EPYC 4 vCPU, 2 core, 16 GB memory
We started with a basic recursive implementation of MergeSort. Once we had this baseline, we created an iterative version. We wanted to implement both techniques, to reduce the slowdown caused by the overhead of repeated recursive function calls. We ultimately saw identical performances from the two. While this result was underwhelming, it makes sense. The algorithms are similar, the iterative function still needs to call merge many times, and modern compilers have limited the overhead of function calls. Cache efficiency and memory bandwidth are likely bigger factors especially at larger array sizes, so focusing on those may be better.



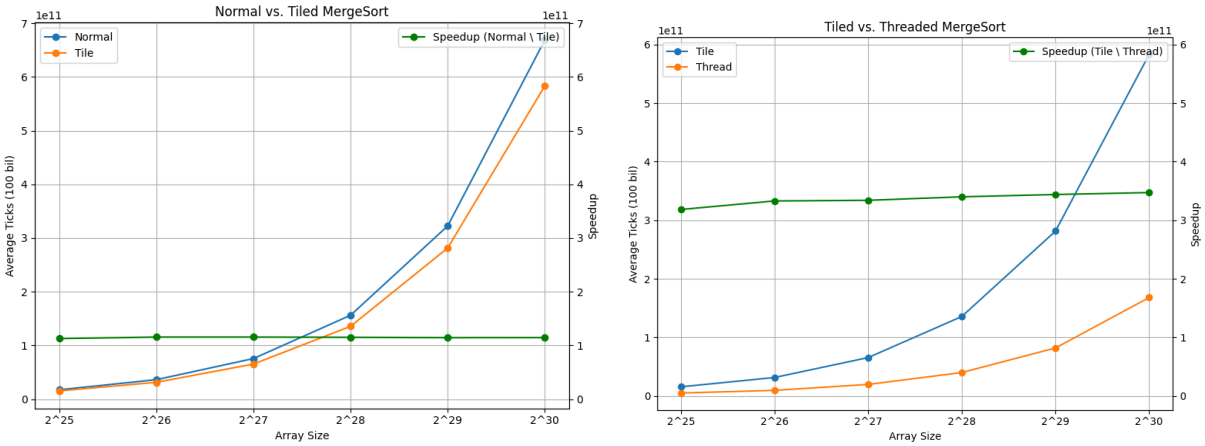


With performance so similar, and the recursive version being simpler, we chose to enhance this.

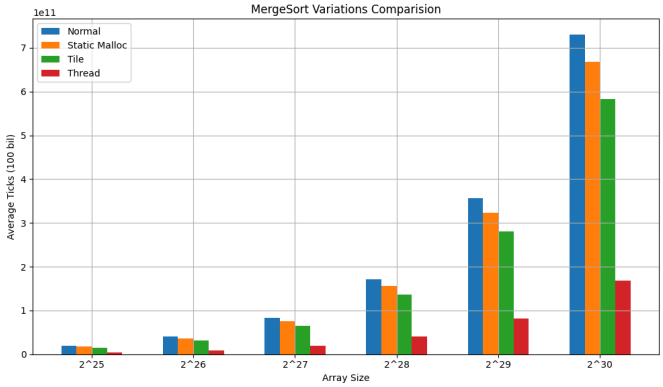
To improve performance, we reduced memory allocation by replacing dynamic allocation inside the merge function, which called malloc 2n times, with a single preallocated temporary array. This approach provides sufficient space for sorting subarrays and the final merge, avoiding the overhead of frequent allocations while maintaining efficiency for large arrays.

The next way we can improve performance is by utilizing the cache more efficiently. Tiling is a technique where we can break the array into small “tiles”, that are small enough to fit into a single cache line. Within these tiles, sorting is sequential, which allows for optimal data access. A normal implementation of MergeSort continues dividing into even smaller subarrays and can cause cache

conflicts. We found the best results using a tile size of 32, so this is likely the cache block size of the gcp.



Finally using inspiration from tiling we implemented a threaded MergeSort. Similar to our tiling approach we split the array into smaller chunks and let a single thread sort each chunk, before merging these. We knew that 32 was the optimal tile size, so we wanted to use a large number of threads to act on each tile in parallel. However, because our machine only used two cores, it could not efficiently utilize enough threads to split the array into such small chunks. We found that the optimal number of threads was 16, this number may be different if the machine had more cores. This meant we had to modify our implementation to no longer use insertion sort since the subarrays were much larger. Instead, each thread performs its own recursive MergeSort on its assigned subarray. This one far outperforms the rest because of its ability to merge simultaneously. While we see a constant speedup here using a fixed amount of threads, on a machine with more cores, we may be able to increase the number of threads with the size of the array. With this improvement, we were able to get a similar time as the vanilla radix sort, achieving our goal. However, a merge sort run on CPU is no match for a vectorized radix sort, and we needed to try something different to beat it.



Mergesort on GPU (NOT stable)

Having not been able to improve the speed of MergeSort to overtake RadixSort, we pivoted to a new strategy for speeding MergeSort. We parallelized using CUDA on a machine with an NVIDIA L4 GPU and an Intel Cascade Lake (8 vCPU with 32GB mem) with GCP. The goal was to enhance sequential mergesort by utilizing GPU parallelism. Local chunks of the array were sorted independently and then merged to produce the final sorted array. We set the chunk size to 1024, matching the GPU's threadblock thread limit, since this is the granularity at which threads are assigned to streaming multiprocessors. Using bitonic sort, well-suited for GPU parallelism, each threadblock sorted a chunk, leaving multiple sorted chunks of size 1024 to be merged into a single array.

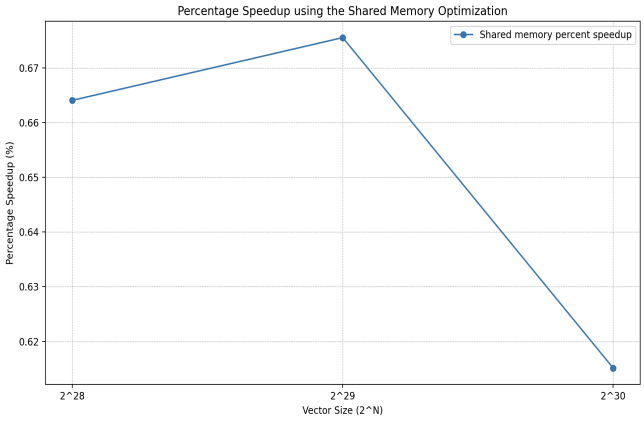
We performed the merge operation in stages, where each stage merged k sorted chunks two at a time. Starting with k sorted chunks of size L, each stage performed k/2 merges to produce k/2 sorted chunks of size 2L, continuing until one fully sorted chunk remained. The merges in each stage will happen in parallel with **respect to other merges** but also **within itself.** We parallelize same-stage merges by assigning different threadblocks to perform each merge operation in parallel. We internally parallelize each merge operation with the **co-rank** function. This function relies on the following observations:

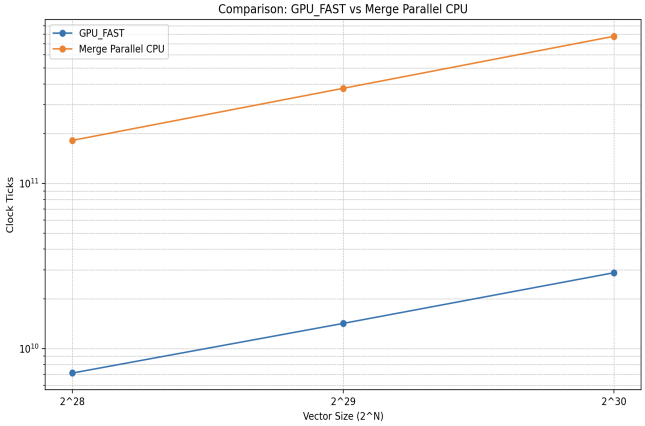
1. When merging two arrays A and B into C, each thread can take complete, unique responsibility for a portion of the output array C that it will populate.
2. For each element to be merged, its index in C can be calculated by summing (index in the input array that it is from) + (number of elements in the other input array that are smaller than it). Thus, since the index in the origin input array is available trivially, all that needs to be calculated is the number of elements in the other input array that are lower than this element. Since both input arrays are sorted, this can be done using binary search, in logarithmic time. Thus, this theoretically reduces merge timing from O(n) to a O(log(n)). ** Note that this simplified explanation doesn't account for duplicates, but the co_rank function does account for this.
3. Each thread can therefore compute (for each element it is responsible for), the range of indices in the input arrays that it will have to merge and be able to perform this operation in parallel.

An additional architectural consideration was to reduce global memory accesses, which have a high latency, and move to **shared memory** accesses which have a lower latency. Thus, we include two implementations, one with the global memory accesses, and another that uses a **tiling** approach to put blocks of the array into shared memory for reduced memory latency.

The data points below were calculated by taking the mean of 10 runs on that specific configuration. The shared memory optimization had a ~0.65% speedup compared to the global memory GPU implementation for large arrays (1 GB, 2GB, 4GB). We also see that comparing the shared memory optimization implementation to the fastest CPU mergesort (mergesort with threads) shows about a ~25x speedup for large arrays. Using the average time per GB for each of the implementations, as well as the **$0.87** per hour for the cost of machine, yields about **$0.00078** cost per hour for the tiled implementation and **$0.00079** for the slower implementation.

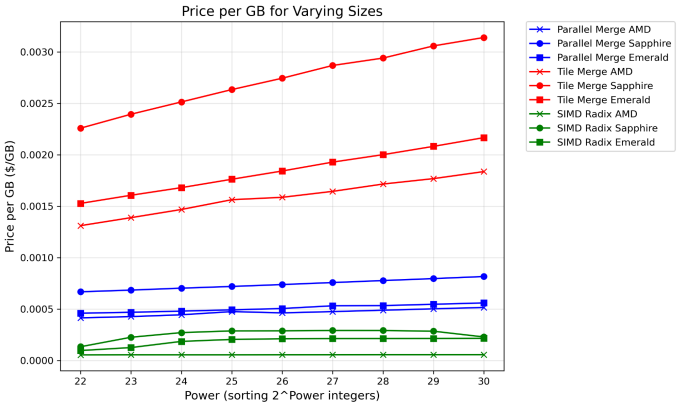
It is important to note that due to the nature of bitonic sort and this implementation is **not stable**. This could be fixed by attaching some sort of tag to each element that contains its position in the original array, so that duplicates can be properly considered by looking at their tag. Since the merging portion is stable, the bitonic sort could also be changed to a sort that is stable on its own, and that would enforce stability within our overall sort. **Since, this lacks stability, we do not include it in the graphs below.**

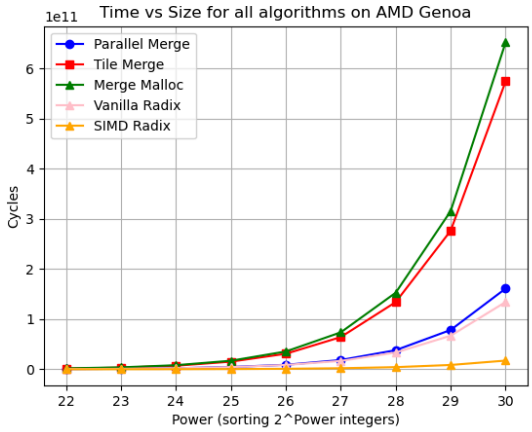




Cost Analysis of Different CPUs:

To calculate the cost per GB, we first obtained the total cost of the sort using the cost per month from GCP, converting to cost per second, then dividing by processor frequency. The cost per GB is the total cost of the sort divided by the size of the sort in GB.





For all platforms and algorithms tested, SIMD Radix Sort demonstrated the lowest cost per GB, achieving up to a 10x speedup over Vanilla Radix Sort and maintaining a consistent cost advantage across dataset sizes. On the AMD Genoa platform, SIMD Radix Sort averaged approximately $0.0004/GB, significantly outperforming Tiled Merge Sort ($0.0015/GB) and Parallel Merge Sort ($0.0008/GB). This cost efficiency stems from its optimized digit-level passes (base-256) and memory access patterns leveraging AVX2 intrinsics. AMD Genoa consistently outperformed Intel Sapphire and Emerald, likely due to its higher clock frequency (3.7 GHz), which provided an edge in CPU-bound sorting tasks.

Conclusion:

While Tiled Merge Sort improved cache utilization and GPU-based Merge Sort achieved up to a 25x speedup over CPU implementations, their cost efficiency lagged behind SIMD Radix Sort, particularly on large datasets. The diminishing speedup of SIMD Radix Sort for datasets exceeding cache sizes highlights opportunities for future optimization to mitigate cache-line conflicts. Additionally, GPU MergeSort's reliance on bitonic sort made it unstable for datasets with duplicate elements, a limitation that could be addressed in future work by incorporating a stable sort variant or tagging elements with their original positions. Exploring iterative MergeSort further, particularly with tiling, could also yield better results through additional testing and refinement.

We would also like to explore running Parallel Merge Sort on a machine with more cores to benchmark its performance with increased thread counts. Further investigation into in-place Merge Sort could reduce memory usage, potentially benefiting both Merge Sort and Radix Sort. Addressing these areas for improvement, including SIMD Radix Sort's performance on datasets exceeding cache limits, would allow us to refine these implementations and further optimize their cost efficiency for large-scale sorting.

While we weren’t able to optimize Merge Sort on CPU to the point of overtaking Radix Sort, we did ultimately work around this with our GPU MergeSort implementation, which leveraged CUDA parallelism to achieve a ~25x speedup over CPU Merge Sort for large datasets. This demonstrates the potential of GPU acceleration to close performance gaps in CPU-dominated algorithms.