Comparison of Sorting Algorithms Performance on GPUs vs CPUs

Oluwatosin Adeosun
adtosin@gmail.com

Jayant Bedwal
jayant21@utexas.edu

Mark Varga
mark.varga@utexas.edu

The University of Texas at Austin

1. INTRODUCTION

Sorting is an important computational block used by many programs and is one of the most studied algorithmic problems. The importance of sorting has led to the design of multiple sorting algorithms. Some of these algorithms work by moving elements to their final position, one at a time. Such algorithms iterate over an array of size N, put 1 item in place, and continue sorting an array of size N âĂŞ 1. Others put items into a temporary position, closer to their final position, then they perform a rescan, moving items closer to the final position with each iteration. Different aspects factor into the selection of an algorithm to be used with an application, including algorithmic complexity, startup costs, additional space requirements, worst-case behavior, assumptions about input data, and behavior on already-sorted or nearly-sorted data.

In this paper, we evaluate the execution time speedup of sorting algorithms on GPUs vs CPUs. GPUs are massively multithreaded processors which can support several thousand concurrent threads. By rewriting the algorithms to take advantage of this feature we could achieve very large speedups as we show later in the paper. However, some algorithms actually take longer to execute on the GPU due to synhronization overheads and other inefficiencies. We describe why these algorithms are performing better or worse. In addition, we look at locality aspects of the algorithms for both the sequential and parallel versions.

For this project we focused on four different sorting algorithms: enumeration sort, merge sort, radix sort and bubble sort. We wrote generic sequential code for the CPU using C/C++, and then we wrote a parallel form of each algorithm using Nvidia CUDA. In this report we will characterize the behavior of both implementations using various array and GPU block sizes. In addition, we looked at both integer and float types. All array elements were generated using the standard rand() function. For timing measurements, we used functions provided by CycleTimer.h similarly to Lab 2 timing measurements. We used perf and nvprof to profile the sequential and parallel CUDA implementations respectively.

We ran all benchmarks on an Intel Xeon CPU and an Nvidia Tesla K20 GPU found in TACC Stampede. The CPU features 16 cores with 32kB L1, 256kB L2, and 2MB L3 caches. The Tesla K20 features 2496 processor cores with 5GB onboard memory and 208GB/s memory bandwidth. All the algorithms on the CPU were compiled with O3 to get the best CPU performance results.

2. ENUMERATION SORT

Enumeration sort is a relatively simple algorithm that for each element counts how many of the other elements are smaller than itself. The final count gives the final position of the particular element in the sorted array. The algorithm thus iterates over all elements for each element, and gives $O(N^2)$ time complexity. Regardless of the inefficiency of the algorithm, it is still interesting to look at how much speedup can be achieved using CUDA.

2.1 Methods of Parallelizing

Our parallel CUDA code for enumeration sort consists of two kernels. The first kernel creates a thread for each element, and within the thread it iterates over all other elements in global memory, calculates the count of smaller elements, and then copies the analyzed element to its final location in a temporary array. The second kernel copies the temporary array back to the original one. This single approach itself can achieve a very substantial speedup compared to the sequential implementation. We also coded a second implementation of the first kernel which loads elements from the global memory into shared memory block by block and iterates over the elements in the shared memory. Using this second kernel provided an even larger speedup, and we decided to use this kernel for further analysis.

2.2 Runtime Benchmarks

Table 1 shows the performance of the sequential CPU algorithm for different array sizes with both integer and float point data types. Table 2 shows the runtimes of the shared-memory CUDA code with four block sizes and for int data types, and Table 3 shows the same but for float data types. For the CUDA benchmarks, we measured both the overall times (including malloc and memory copy), and compute-only times. An analysis follows the tables. Note that the CUDA tables shows benchmarks up to 1 million elements, while the sequential table below only shows results up to 100,000 elements due to larger arrays taking very long to sort.

Comparing Tables 1, 2 and 3, we immediately see that the speedup due to using Tesla K20 is very substantial. Using 100,000 elements and a block size of 256, the CUDA version with shared memory achieved a speedup of 1207x with floats. For integers, we achieved best results with a block size of

Table 1: Enumeration Sort CPU Runtimes

Element Count	Average Time for Integer (ms)	Average Time for Float (ms)
100	0.188	0.224
1000	7.936	9.86
10000	879.841	1,071.669
100000	87082.695	106,473.18

512 threads, and a speedup of 981x. Interestingly, we getter speedup with floating point arrays, possibly due to the fact that GPUs need to perform many floating point operations and thus are well-optimized for that.

We also see interesting trends when we look at how well different block sizes and data types perform. From the tables, 32 threads per block seem suboptimal for floats and very suboptimal for ints. Presumably, with 32 threads we donâĂŹt get enough SIMD and/or latency hiding. In contrast, the other block sizes perform roughly the same, with some differences depending on data type and array size. For one million ints, a block size of 512 seems to be the best choice, while for floats 256 is better.

2.3 CPU Locality and IPC Analysis

Besides benchmark analysis, we also performed locality analysis on the sequential and parallel implementations. In this section we present data we collected about the sequential enumeration sort algorithm using perf. In particular, we looked at L1 and LLC hit ratio as well as instruction per cycle metrics. Table 4 below shows the results of our benchmarks for different integer array sizes.

As the results show, the enumeration sort achieves relatively good L1 hit ratio. This is explained by the relatively small size of the arrays, with the biggest one taking roughly 400KB. As expected, increasing the array size lowers hit ratio in L1, but the Last Level Cache (LLC) is able to absorb those misses, and the algorithm has to access main memory only rarely. The Instruction Per Cycle performance remains stable throughout the tests, although with larger array sizes we achieved less IPC efficiency.

2.4 GPU Locality and Bandwidth Analysis

Using the Nvidia CUDA Profiler nvprof, we took locality and memory bandwidth measurements for the GPU version that uses shared memory. We benchmarked for all array sizes from 100 to 100,000 integers, with a fixed block size of 512. The second kernel of the code only copies the temporary array back to the final one, so we focused only on analyzing the first kernel, which performs most of the work. Table 5 below shows our measurement results.

The results show that SM efficiency starts out low at small array sizes but achieves very high efficiency for larger arrays. The same trends are true for the Instruction Per Cycle metric. Global load requests are around 100% throughout, i.e. the same amount of memory bandwidth was used as the requested amount. This is not a particularly low number, especially compared to the poor store efficiency, but we have seen load efficiency of up to 400% with the other version of the kernel that does not use shared memory (numbers not shown here). This means that with the simpler kernel memory requests are coalesced much better. However, the kernel we are analyzing here achieves much better performance due to the utilization of shared memory, which achieves a peak throughput of 576 GB/s and reduces global load throughput from around 62 GB/s with the simpler kernel to 1.13 GB/s with the shared memory kernel (shown in the last row). We achieved 100% L2 hit rate for reads throughout our tests, which is surprising but is likely explained by the relatively low problem size. Overall, based on the table above and functional unit utilization metrics provided by nvprof, the utilization of Tesla K20 seems relatively low with this algorithm, but it is unlikely that the code could be optimized much further.

3. BUBBLE SORT

The Bubble sort algorithm is a comparison algorithm. The algorithm continuously steps through the input list, swapping adjacent elements until the list is in order. It is a very simple but slow algorithm with an average and worst case performance of $O(n^2)$. When the input array is already sorted the performance improves to O(n). While it's not a practical algorithm to use in real-life scenarios, we included it because we wanted to see the behaviour of a slow algorithm on a GPU, similarly to enumeration sort. Because of the long computation time, we can only compute for a maximum of 100,000 elements.

3.1 Method of Parallelizing

To parallelize this algorithm, we created a kernel that loops through the whole array and compares and swaps the elements if it is larger. For a n sized array, we launch n threads to do the comparisons. Inside each thread, we loop n-1 times to compare each element to the next element.

3.2 Runtime Benchmarks

We collected the runtime results for both the sequential and the parallel implementations in Tables 6, 7 and 8.

From Table 6, we see that the float data type takes less execution time than the int data type for 1000, 10,000 and 100,000 data elements. This is likely because the data is generated randomly and the random input array generated for floats is probably better sorted initially than that of the int data types. Comparing Tables 6, 7 and 8, we can see some speedups on the GPU compute time compared to the CPU. For integers, at 100,000 elements, we get the best compute time on the GPU using a block size of 128 threads. At that block size, the GPU achieves a speedup of 7x compared to the CPU. For floating point numbers for the same number of elements, the best compute time is also with 128 threads per blocks and we have a speedup of 6.5x . We have slightly better speedup with CPU vs floating point. However, unlike merge and enumeration there is not much speedup with the GPU. This is expected, because there is not much fine grained parallelism with bubble sort. In order not to lose information about the order, the comparisons need to be done atomically, and this ends up worsening the performance on the GPU.

3.3 CPU Locality Benchmarks

Table 2: Enumeration Sort CUDA Runtimes for Integers

	32 Threads per Block		128 Thread per Block		256 Threads per Block		512 Thread per Block	
Element Count	Average Overall Time(ms)	Average Compute Time (ms)	Average Overall Time(ms)	Average Compute Time (ms)	Average Overall Time(ms)	Average Compute Time (ms)	Average Overall Time(ms)	Average Compute Time (ms)
100	0.28	0.042	0.502	0.272	0.272	0.038	0.488	.258
1000	0.344	0.109	0.54	0.311	0.328	0.096	0.534	0.305
10000	2.293	2.021	1.42	1.15	1.355	1.09	1.493	1.226
100000	140	138.5	72.435	71.94	72.15	71.651	70.358	69.861
1000000	13700.329	13697.388	6692.602	6689.722	6694.888	6691.993	6587.469	6584.613

Table 3: Enumeration Sort CUDA Runtimes for Floats

	32 Threads per Block		128 Thread per Block		256 Threads per Block		512 Thread per Block	
Element Count	Average Overall Time(ms)	Average Compute Time (ms)	Average Overall Time(ms)	Average Compute Time (ms)	Average Overall Time(ms)	Average Compute Time (ms)	Average Overall Time(ms)	Average Compute Time (ms)
100	0.296	0.052	0.503	0.271	0.284	0.052	0.286	.053
1000	0.463	0.228	0.818	0.585	0.447	0.214	0.445	0.211
10000	5.507	5.235	2.176	1.904	2.264	1.991	2.373	2.105
100000	363.213	362.685	110.46	109.97	88.185	87.676	88.759	88.271
1000000	34971.558	34968.103	9303.39	9300.602	8289.289	8286.471	8348.082	8345.317

Table 4: CPU Perf Results

Element Count	L1 References	L1 Misses	L1 Hit Ratio	LLC References	LLC Misses	LLC Hit Ratio	IPC
100	426,431	17,003	96.01%	4,877	3	99.94%	1.32
1000	1,440,630	17,234	98.80%	5,074	13	99.74%	1.34
10000	100,731,378	6,317,645	93.73%	6,319	9	99.86%	1.22
100000	3,336,303,553	627,439,469	81.19%	37,503,137	502	99.99%	1.23

Table 5: GPU Locality and Bandwidth Benchmarks

Element Count	SM Efficiency	Warp Execution Efficiency	IPC	Requested Global Load Throughput (MB/s)	Requested Global Store Throughput (MB/s)	Global Load Throughput (MB/s)	Global Store Throughput (MB/s)	Global Memory Load Efficiency	Global Memory Store Efficiency	Device Memory Read Throughput (MB/s)	Device Memory Write Throughput (MB/s)	L2 Hit Rate (L1 Reads)	Shared Memory Load Throughput (GB/s)
100	4.67%	81.4%	0.651	75.514	37.757	78.535	123.84	96.15%	30.49%	0	120.82	100.00%	4.8329
1,000	14.30%	97.67%	2.529	173.29	57.762	173.29	420.04	100.00%	13.75%	0	414.50	100.00%	59.148
10,000	85.66%	99.84%	3.222	934.33	44.492	934.33	352.16	100.00%	12.63%	0.115	426.34	100.00%	445.63
100,000	94.12%	100.00%	3.802	1163.67	5.7685	1163.67	46.103	100.00%	12.51%	4.894	84.511	100.00%	576.85

Table 6: Bubble Sort CPU Runtimes

Element Count	Average Time for Integers (ms)	Average Time for Float (ms)
100	0.01	0.07
1,000	0.726	0.614
10,000	64.786	61.419
100,000	6541.476	5985.909

We collected CPU locality results using perf for the sequential algorithm. Results are shown in Table 9.

As the results show, the bubble sort has very high L1 hit rates because there is a lot of temporal locality present in memory accesses due to the presence of the loops and accesses are done in a stride of a single element, thereby giving high spatial locality. The LLC has a very high hit rate because it is possible to fit the whole input array in the LLC and the algorithm has to access main memory only rarely. The Instruction Per Cycle performance varies among the different element sizes. It is highest at 1000 elements and then reduces as the number of elements increases. This is probably because we have the highest L1 hit ratio for 1000 elements so the number of cycles used is not as high compared to others. Therefore most of the elements in the 1000 element sized array can fit into the L1 cache, and for the 10000 and 100,000, there will be substantial increase in L1 misses which adds more cycles and reduces the ipc.

3.4 GPU Locality and Bandwidth Analysis

For the locality and memory bandwidth measurements of the GPU, we benchmarked for all the array sizes with a fixed blocksize of 512 threads. Table 10 shows the result of the measurement. There are no results for element count over 10000 because they take a very long time to profile on the GPU's and the job times out before they are complete.

We see in Table 10 that the SM efficiency increases as the number of elements we have increases because we can utilize more thread blocks to do the computation. The warp efficiency also increases as the number of elements increase for a similar reason as the increase in sm efficiency.. The global memory load and store efficiencies are increasing slowly with the increase in the number of elements.The global load efficiency is around average and the store efficiency at 32%. This indicates poor memory coalescing, the threads are requesting data from locations that are not close together in physical memory. The device memory read and write throughputs increases significantly as the number of elements increases. This is expected as the amount of memory reads and writes will be higher when there are more elements.

4. MERGE SORT

The merge sort algorithm is a divide and conquer algorithm. First the algorithm divides the array to be sorted into two equal subarrays recursively till each subarray contains one element. Then it merges the subarrays to produce new sorted subarrays until there is only one subarray remaining which is the sorted array. To sort n objects, merge sort has a worst case performance of O(n log n), making it one of the fastest sorting algorithms, and thus it is widely used.

4.1 Method of Parallelizing

For the parallel CUDA code that runs on GPU, we decided that the best way to design the algorithm would be to use the bottom-up merge sort technique. We start with two arrays, the input array and a temporary copy of the input array. Then we define a width, starting at 2, and during each step, the width is multiplied by 2. And while the width is less than twice the size of the array, sort each width sized chunk of the array into the temporary copy, then switch the pointers of the arrays. This avoids allocating lots of tiny arrays. The parallelism in this is that for each width iteration, each thread gets a portion of the array(the width chunks) to sort.

4.2 Runtime Benchmarks

Table 11 shows the performance of the sequential CPU algorithm for different array sizes with both int and float data types. Table **??** shows the runtimes of the CUDA code with four different block sizes and for int data types, and Table 12 shows the same but for float data types. For the CUDA benchmarks, the overall and compute-only times were almost the same so we only included overall times. An analysis follows the tables. Like the previous algorithms, the elements were generated randomly.

From Table 11, we see that the float data types take more execution time than int. Comparing Tables 11, **??** and 12, we can see the performance is actually worse on the GPU than on the CPU. This is likely because the loop to assign the width is done outside the kernel which causes a lot of memory transfer overhead. We tried to implement this phase in shared memory and we noticed speedups for 1,00 and 1,000 element arrays but could not get it to work for larger sized arrays due to time constraints.

4.3 CPU Locality and IPC Analysis

Similarly to the previous algorithms, we collected CPU locality data for the sequential implementation using perf. The results for different integer array sizes are shown Table 13.

As the results show, the merge sort has very high L1 hit rates. This is explained by the fact that there is a lot of spatial locality in the algorithm because the elements are usually accessed in order. The LLC also has a very high hit rate because it is possible to fit the whole input array in the LLC and the algorithm has to access main memory only rarely. The Instruction Per Cycle performance remains stable throughout the tests, although with larger array sizes we achieved better IPC efficiency.

4.4 GPU Locality and Bandwidth Analysis

For the locality and memory bandwidth measurements of the GPU, we benchmarked for all the array sizes with a fixed blocksize of 512 threads. Table 14 below shows the results of the measurements.

We see that the SM efficiency is very small. This is mostly because not too many thread blocks are used towards the end of the sorting where the width chunks are large. The global memory load and store efficiencies are quite low, meaning that the requested global load bandwidth is a lot less than what was actually used by the algorithm for both loads

Table 7: Bubble Sort CUDA Runtimes for Integers

	32 Thread per Block		128 Thread per Block		256 Threads per Block		512 Threads per Block	
Element Count	Average Overall Time (ms)	Average Compute Time (ms)	Average Overall Time (ms)	Average Compute Time (ms)	Average Overall Time (ms)	Average Compute Time (ms)	Average Overall Time (ms)	Average Compute Time (ms)
100	0.538	0.419	0.538	0.358	0.536	0.348	0.784	0.439
1,000	4.963	3.897	5.336	3.931	5.287	3.878	5.471	4.061
10,000	59.796	55.021	52.412	48.502	53.286	49.749	53.014	49.084
100,000	1841.152	1822.471	922.694	913.075	936.76	926.939	976.859	966.601

Table 8: Bubble Sort CUDA Runtimes for Floats

	32 Threads per Block		128 Threads per Block		256 Threads per Block		512 Threads per Block	
Element Count	Average Overall Time (ms)	Average Compute Time (ms)	Average Overall Time (ms)	Average Compute Time (ms)	Average Overall Time (ms)	Average Compute Time (ms)	Average Overall Time (ms)	Average Compute Time (ms)
100	0.543	0.361	0.536	0.349	0.536	0.418	0.536	0.353
1,000	4.958	3.995	4.973	4.347	5.064	3.961	5.223	4.078
10,000	59.694	55.19	52.305	49.463	53.264	49.504	52.8	49.075
100,000	1843.922	1825.36	922.628	913.277	936.725	927.188	976.095	966.011

Table 9: Bubble Sort CPU Perf Results

Element Count	L1 References	L1 Misses	L1 Hit Ratio	LLC References	LLC Misses	LLC Hit Ratio	IPC
100	445,226	17,251	96.13%	4893	17	99.65%	1.3
1000	2,977,483	17,161	99.42%	4903	8	99.84%	1.85
10000	251,353,566	9,269,454	96.31%	5046	8	99.84%	1.45
100000	25,005,225,964	919,580,031	96.32%	38179726	170	100.00%	1.28

Table 10: GPU Profiler Results for Bubble Sort

Element Count	SM Efficiency	Warp Execution Efficiency	IPC	Requested Global Load Throughput (MB/s)	Requested Global StoreThroughput (MB/s)	Global Load Throughput (GB/s)	Global Store Throughput (GB/s)	Global Memory Load Efficiency	Global Memory Store Efficiency	Device Memory Read Throughput (KB/s)	Device Memory Write Throughput (GB/s)	L2 Hit Rate (L1 Reads)
100	1.88%	88.58%	0.172253	86.432	41.759	0.181	0.119	46.72%	28.81%	0.00	0.116	100.00%
1000	3.59%	98.09%	0.444045	810.58	412.96	1.6711	1.0810	48.54%	32.05%	0.00	1.2822	100.00%
10000	26.49%	99.83%	0.585173	8097.39	4069.78	16.308	10.408	48.53%	32.37%	227.42	11.024	100.00%

Table 11: CPU Runtimes for Merge Sort

Element Count	Average Time for Integers (ms)	Average Time for Floats (ms)
100	0.002	0.002
1000	0.025	0.026
10000	0.286	0.309

Table 12: Merge Sort CUDA Runtimes for Floats

	32 Threads per Block	128 Threads per Block	256 Threads per Block	512 Threads per Block
Element Count	Average Overall Time (ms)	Average Overall Time (ms)	Average Overall Time (ms)	Average Overall Time (ms)
100	0.224	0.219	0.219	0.222
1000	1.786	1.556	1.547	1.542
10000	23.946	20.347	19.885	19.7
100000	233.08	181.382	174.796	172.467
1000000	2430.132	1747.565	1666.356	1639.085

and stores. This indicates poor memory coalescing, probably because concurrent threads are requesting data from far apart locations in physical memory. The device memory read throughput increases significantly as the number of elements increases. This is expected as the amount of memory reads will be higher when there are more elements. The device memory write throughput increases till 100,000 elements, then there is a steep drop off when there are 1,000,000 elements. This is likely due to less L2 hits creating more accesses and thus more contention in device memory.

5. RADIX SORT

Radix sort is a non-comparative integer sorting algorithm that sorts data with integer keys. It groups keys by the individual digits which share the same significant position and value. It assumes that the keys are D-digit numbers and sorts on one digit of the key at a time, from least to most significant. For a collection of N integers with the maximum number of digits as D, the algorithm needs to make D passes over all numbers and the complexity of the algorithm for sorting a total of N numbers is on the order of O(N).

5.1 Parallel Implementation

The parallel implementation consists of 3 CUDA kernels. Before starting the sort, the algorithm finds the largest number in the array. The number of digits D in this number determines the number of passes the algorithm has to go through. Then procedure of the algorithm is as follows:

1. The array is divided up into blocks, and elements for each block are copied into the shared memory.
2. The first kernel creates a histogram of the digit at the nth place in the numbers to be sorted out. A separate thread works on each individual element of the array, extracting the digit and incrementing the value of the corresponding histogram bin using atomic operations.
3. The partial histogram arrays are copied into the global memory and are combined to form the complete histogram array. Based on this histogram array, the next kernel calculates the location of each element in the semi-sorted array after that pass. In this kernel, each of the thread is working on each radix in the histogram.
4. The elements are copied in their respective locations as per the index array generated by the previous kernel.

This whole procedure is repeated for each of the D digits.

5.2 Results and Benchmarks

Table 15 shows the performance of sequential CPU algorithm for different array sizes for both float and int values.

Tables 16 and 17 below show the performance of CUDA implementation for both integer and floating point values.

Radix sort is one of the best-known sorting algorithms, and on sequential machines it is often the most efficient one as well. As we can see by comparing tables 15, 16, and 17 the sequential implementation is much faster than the parallel implementation. These measurements are consistent with the ones described in [1]. The results can be attributed to the following reasons:

1. The overhead of shuffling data in GPU is much more than in CPU. This is further worsened by the current implementation which performs these data movements in the global memory rather than in the shared memory. The reason to perform them in the global memory is that there is no restriction on where the element will end up after the shuffling. An element in the nth index can land up in 1st index. Since different blocks cannot access shared memory of other blocks, implementing the data movement in shared memory would require bringing in and moving data out of blocks frequently along with increasing the complexity of the implementation.
2. To perform the complete sort we implemented 6 kernels. Each kernel requires the results of the previous kernel to move forward. Thus synchronization is required between each of the kernels. This results in overhead. The more the number of digits in numbers, more the number of iterations, resulting in even more synchronization overhead.

5.3 CPU Locality and IPC Analysis

We collected CPU locality data for the integer sequential algorithm using perf. The results are shown in Table 18

The result indicates that radix sort gets a good L1 hit ratio which further helps in speeding up the sequential performance over the GPU implementation. The hit ratio remains the same for all the array sizes because for the data movement part most of the array is required in the memory. Also the LLC is also having a good hit ratio for almost all the array sizes. IPC is relatively stable but slightly increases for larger array sizes.

6. REFERENCES

[1] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, and P. Dubey. Fast sort on cpus and

Table 13: Merge Sort CPU Perf Results

Element Count	L1 References	L1 Misses	L1 Hit Ratio	LLC References	LLC Misses	LLC Hit Ratio	IPC
100	427,496	17,066	96.01%	5,269	6	99.89%	1.29
1000	539,194	17,076	96.83%	4,844	8	99.83%	1.28
10000	1,816,522	26,339	98.55%	5,113	8	99.84%	1.39
100000	15,844,444	225,534	98.58%	31,408	6	99.98%	1.37
1000000	168723601	3290669	98.05%	695079	105	99.98%	1.33

Table 14: Merge Sort GPU Profile Result

Element Count	SM Efficiency	Warp Execution Efficiency	IPC	Requested Global Load Throughput (MB/s)	Requested Global Store Throughput (MB/s)	Global Load Throughput (GB/s)	Global Memory Load Efficiency	Global Memory Store Efficiency	Device Memory Read Throughput (MB/s)	Device Memory Write Throughput (GB/s)	L2 Hit Rate (L1 Reads)
100	5.52%	45.77%	0.067	47.03	27.869	0.254	17.35%	14.92%	0	0.255	100.00%
1000	6.44%	52.59%	0.11	266.22	159.27	1.373	16.05%	14.25%	0.543	1.375	100.00%
10000	7.48	53.35%	0.09	405.09	229.33	3.24	12.50%	12.50%	0.119	3.440	100.00%
100000	7.67%	60.57%	0.09	514.90	277.78	4.119	12.50%	12.50%	138.03	3.633	98.43%
1000000	7.69%	68.89%	0.1	618.08	326.84	4.944	12.50%	12.50%	711.54	0.897	93.55%

Table 15: Radix Sort CPU Runtimes

Element Count	Average Time for Integer (ms)	Average Time for Float (ms)
100	0.146	0.162
1000	0.449	0.715
10000	3.633	5.986
100000	35.058	57.726

Table 16: Radix Sort CUDA Runtimes for Integers

Element Count	32 Threads	128 Threads	256 Threads	512 Threads
100	3.21	3.21	3.329	3.306
1000	11.92	11.848	11.908	11.992
10000	98.762	97.743	97.388	96.845
100000	970.396	958.272	956.896	955.988

Table 17: Radix Sort CUDA Runtimes for Floats

Element Count	32 Threads	128 Threads	256 Threads	512 Threads
100	11.978	12.432	12.405	12.374
1000	44.905	44.823	45.041	45.267
10000	371.082	365.806	365.447	363.855
100000	3650.012	3599.192	3592.899	3593.085

Table 18: CPU Perf Results for Radix Sort

Element Count	L1 References	L1 Misses	L1 Hit Ratio	LLC References
100	484037	17,883	96.31	5,397
1,000	993403	18,292	98.16	5,394
10,000	6088381	65,135	98.93	5,896
100,000	42858383	546,010	98.73	28,157
1,000,000	425700902	5,347,067	98.74	232181

gpus: A case for bandwidth oblivious simd sort. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 351–362, New York, NY, USA, 2010. ACM.