GPGPU - Merge Sort

Péter Dávid Podlovics

May 27, 2019

1 Parallel merge sort

This document summarizes the results of the GPGPU assignment *Parallel Merge Sort.* The task is to implement a parallel version of merge sort on floats using general purpose GPU programming, and compare the CPU and GPU based implementations.

1.1 Overview of the implementation

Besides the sequential CPU and purely GPU based algorithms, I also implemented a parallel CPU, and two hybrid algorithms. The hybrid implementations try to combine the CPU's and GPU's processing power to yield a more optimal solution for the problem.

All implementations use the bottom-up version of merge sort, where *insertion sort* is used to sort the initial ranges; but the merging part differs based on whether the program is executed on the CPU or on the GPU. The CPU programs use the standard version, where an additional array is allocated for the merging; but the GPU based programs have to sort in place, so they utilize a variant where the merging is done by shifting.

The standard algorithm runs in $\mathcal{O}(n \log n)$ time and uses $\mathcal{O}(\frac{n}{2})$ space The in-place variant uses no additional space, but runs in $\mathcal{O}(n^2 \log n)$ time. Hopefully, we will be able to circumvent the additional time overhead by exploiting the power of the massively parallel execution.

1.2 Sequential CPU

This implementation uses the standard algorithm, and executes it sequentially.

1.3 Parallel CPU

This implementation also uses the standard algorithm, but executes the initial insertion sorts, and the subsequent merges in parallel using `C++ threads`. The program can be parameterized by the length of the initial ranges. These ranges will be sorted by insertion, then they will be merged together.

1.4 Pure GPU

This implementation uses the in-place variant of merge sort, but executes the insertion sorts and the merges on the GPU. The considerable disadvantage of this implementation is that it only uses the GPU. As a consequence, during the last few iterations, the ranges to be merged can become very long.

This not only means a GPU thread will have to do *more* work, but also that very few threads will have to do *all* the work, since there will be only a very small amount of very long ranges. In the most extreme situation, a sinlge GPU thread will have to merge two ranges of length $\frac{n}{2}$, while all the other threads are completely idle.

1.5 Sequential hybrid

This implementation uses the standard version of the algorithm. It tries to circumvent the problem detailed in the above section by combining GPU parallel and CPU sequential execution. The insertion sorts are executed by GPU threads, but the merging is executed in a hybrid fashion. The execution will start on the GPU, but when the length of the ranges reach a certain threshold, the entire array will be copied back to the CPU, and the rest of the algorithm will be executed sequentially there.

It is important to note, that the merges executed on the GPU use the in-place variant of the merging algorithm, but the rest executed on the CPU use the more time-efficient version.

1.6 Parallel hybrid

This is implementation is very similar to the sequential hybrid one, but instead of executing the merges sequentially on the CPU, it utilizes `C++ threads` to parallelize the computation.

2 Statistics

This section presents the results of the implemented algorithms. The programs were fed both randomly generated data, and initially sorted data as well. The statistics can be seen in the tables below. The header column contains the names of the different implementations, and the the header row contains the size of the input in ”thousand elements”.

The table entries show the time of the execution in milliseconds. The execution for the GPU implementation were measured starting from after copying the data to the GPU, and ending after copying the data back to the CPU. In other words, both the execution time of the algorithm, and the time of retrieving the data from the GPU were measured, but putting the data on the GPU was not.

Table 1: Random input statistics

	5	10	25	50	100	250	500	1000	2000	5000	10000
cpu_seq	4.5	9.2	17.5	27.3	53.1	116.2	240.9	497.8	1030.9	-	-
cpu_par	2.2	4.1	6.6	12.1	25.8	71.3	140.8	277.4	561.3	1374	-
gpu	50	175.5	1100	-	-	-	-	-	-	-	-
hs_seq	2.0	3.2	7.2	14.0	27.5	67.1	127.2	251.3	538.3	1337	-
hs_par	2.4	3.2	7.3	13.1	25.4	55.7	98.7	174.2	347.1	823	1750

The initial length of the ranges for the parallel CPU implementation was 400 elements. The maximal workload for a GPU thread (i.e.: the switching threshold) in the hybrid implementations was 180. Both of these values were chosen on an empirical basis.

Table 2: Initially sorted input statistics

	5	10	25	50	100	250	500	1000	2000	5000	10000
cpu_seq	1.9	3.9	9.8	22.1	43.6	110.1	222.4	455.5	943.7	2378	-
cpu_par	1.4	2.2	6.1	11.1	23.8	67.4	133.8	271.9	546.0	1279	-
gpu	2.1	3.7	12.1	23.8	49.5	97.7	186.3	373.4	-	-	-
hs_seq	1.1	1.7	3.9	7.6	17.1	44.3	90.8	193.0	410.9	1071	-
hs_par	1.2	1.8	3.7	7.5	14.3	33.5	67.0	136.8	275.3	661	1401

When the execution time of a program reached 1 second, it was ”disqualified” from any further measurements. Also the current implementation of the purely GPU based algorithm does not support more than one million threads, so no measurements were taken for the five and ten million cases.

3 Conclusions

As we can see from the statistics, the best performing algorithm is the parallel hybrid one. Then comes the sequential hybrid implementation, followed by the parallel CPU variant. The second to last one is the sequential implementation, and the worst one is the purely GPU based variant.

It is a bit surprising to see the GPU based implementation to finish last, but merging a long range of elements on a single GPU core is highly inefficient. This is why the hybrid implementations performed so well. They combined the best aspects of CPU and GPU execution. They yielded not only great results, but stable ones as well. The variance of the measurements of the hybrid implementations were considerably lower than those of the other implementations.

Also, it is important to note, that the parallel CPU implementation performed pretty well for smaller inputs. However, if we were to process huge amounts of data, the parallel hybrid algorithm trumps all other approaches.