Introduction

This report implements an odd-even sort in serial program using C and parallel programs using OpenCL. This report compares their performance differences and analyzes the factors that affect their performances.

Odd-even algorithm

Odd-even sorting algorithm is similar to bubble sorting algorithm. Given an array, the array is sorted in two stages. At the first stage, elements of every odd-even indexed pair are sorted. At the second stage, elements of every even-odd indexed pair are sorted. These two stages alternate until the entire array is sorted.

Experiments setting

The parallel program and the serial program perform the odd-even sort on integer arrays. Eight different array sizes are investigated: 2^{14}, 2^{15}, 2^{16}, 2^{17}, 2^{18}, 2^{19}, 2^{20}, 2^{21}. For the parallel program, this report investigates four different local work sizes: 1, 256, 512, 1024. Programs are executed five times and only the median runtime is adopted.

Analysis

All the experiment results are presented in table 1, figure 1 and figure 2. The following paper will discuss the factors that affect the program performances. Noted that in the following paper, “work-item” refers to all the work-items being created by OpenCL, and “local work-item” refers to the work-items within a workgroup.

Effects of using a serial or a parallel program

Parallel program using more than 256 local work items run much faster than the serial program. It means that a parallel program with a proper number of local work items outperforms a serial program.

In the parallel program, enough number of work-items are created to fit the number of array elements. Each work item has a global id and uses this id as an index to access two elements of the array. Values of these two elements are compared and swapped when they are not in ascending order. All the executions are done in parallel. But for the serial program, it must traverse the entire array using a for loop. Therefore, the serial program is more time-consuming than the parallel program.

Effects of the number of local work sizes

Parallel program running with 1 local work item has the worst performance. It spent about 3640 seconds sorting an array of size 2^{20} which is 300 seconds longer than the serial program. This is probably due to multiple accesses to global memory degrading the program performance. In OpenCL, data are always moved from host memory to

global memory, from global memory to local memory and back. So, when there is only one work item in each workgroup, a large number of workgroups will be generated, and the number of accesses to global memory will be significantly large.

Parallel programs running with 256 and 512 local work items took almost the same amount of time, while programs running with 1024 local work items took slightly longer. Their similar performances are due to the way to implement sorting algorithm in the kernel function. As mentioned before, OpenCL always creates a certain number of work items (as much as the value of array size), regardless of how much the local work size is. And each work item will be assigned completely the same amount of workload. In other words, the parallel runtime relies more on the array size instead of the local work size. Therefore, parallel programs with different local work sizes have similar performances for a given array.

Table 1. Runtimes of serial and parallel programs in milliseconds

array size	serial runtimes	parallel runtimes			
		local work size = 1	local work size = 256	local work size = 512	local work size = 1024
2^{14}	123.66	420	111	110	114
2^{15}	489.585	1222	227	224	232
2^{16}	1961.342	4231	494	498	519
2^{17}	7814.116	15915	1069	1061	1118
2^{18}	31425.9	62316	2303	2308	2405
2^{19}	126473	247120	5708	5951	6512
2^{20}	507822.1	982970	19568	20230	22425
2^{21}	2025589	3639350	128626	129617	134409

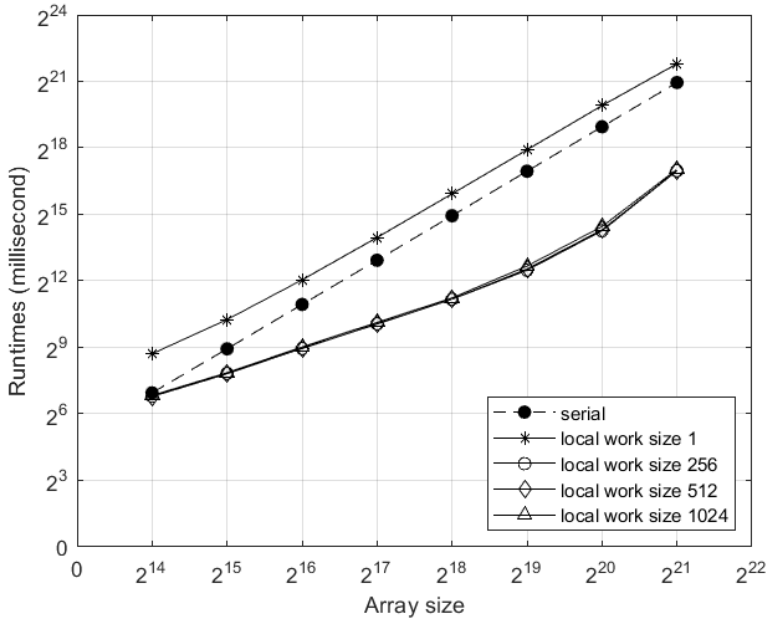

Figure 1. Runtimes of Serial and Parallel programs for each array size

Effects of the array size

Figure 2 shows the speedup of the parallel program for different array sizes. When the array size is less than 2^{16}, the speedup grows steadily. But when the array size is greater than that value, the speedup increases significantly.

Speedup is the ratio of serial runtimes and parallel runtimes, so the increasing speedup can result from the inferior performance of the serial program. As the array size increases to 2^{18} or more, traversing the entire array becomes extremely time-consuming. This is indicated in figure 1 that the slope of the serial runtime is considerably higher than the parallel runtime. Therefore, as the array size increases, the performance difference between serial and parallel programs becomes more pronounced.

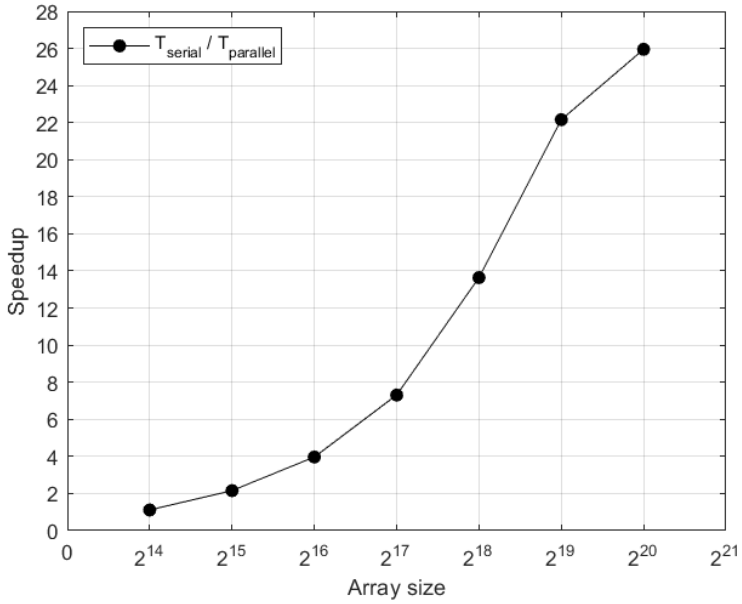

Figure 2. The Speedup of the Parallel programs (local work size = 256)

Conclusion

This report describes the performance difference between serial and parallel program and discusses how the local work sizes and the array sizes affect the performance. Programs with different local work sizes show a similar performance due to the special way to realizes the odd-even sort in the kernel function. This report also notices that there is a dramatic increase in speedup because of the degrading serial performance when sorting arrays with huge elements.