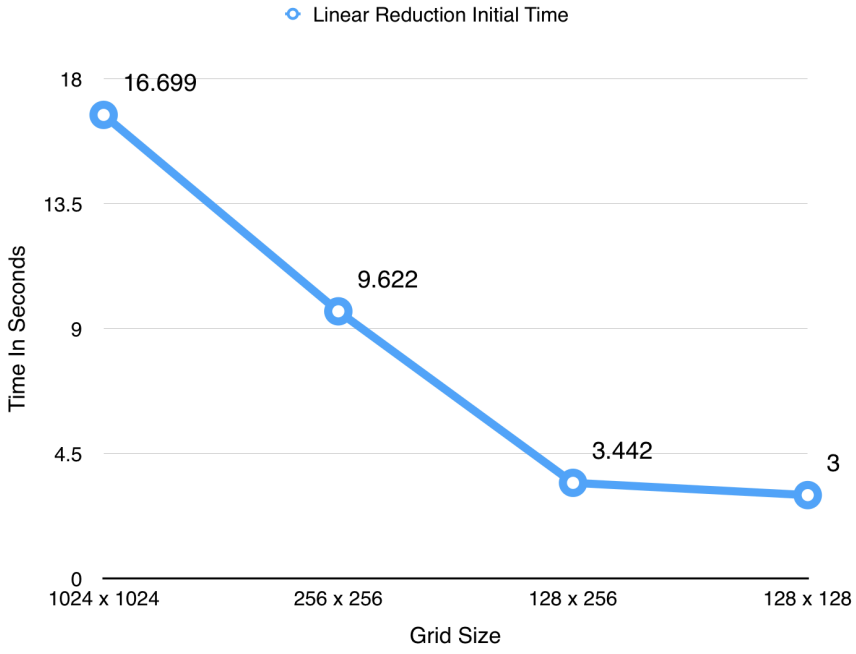
Oliver Goldstein: OG14775

OpenCL Assignment

Linear Reduction

Initially, the first optimisation, was to implement a basic linear time reduction over the variable tot_u. I initially ignored tot_cells which does not change over the course of the program. The loop_fusion held all functions except for accelerate flow. In addition a struct of arrays was implemented in order to allow for efficient vectorisation over the loop level. The vectorisation capabilities of the code was checked before placing them in the kernels with icc's vectorisation report tool. The initial timings can be seen in Fig 1.0. The cost of this operation is extremely large as on each time-step, for the 1024 x 1024 grid, with a work group size of 128, over 8000 partial reductions occur with each of these reductions taking linear time which gives about one million operations occurring. Even worse, these operations will be memory bound operations.

Fig 1.0 Initial Timings



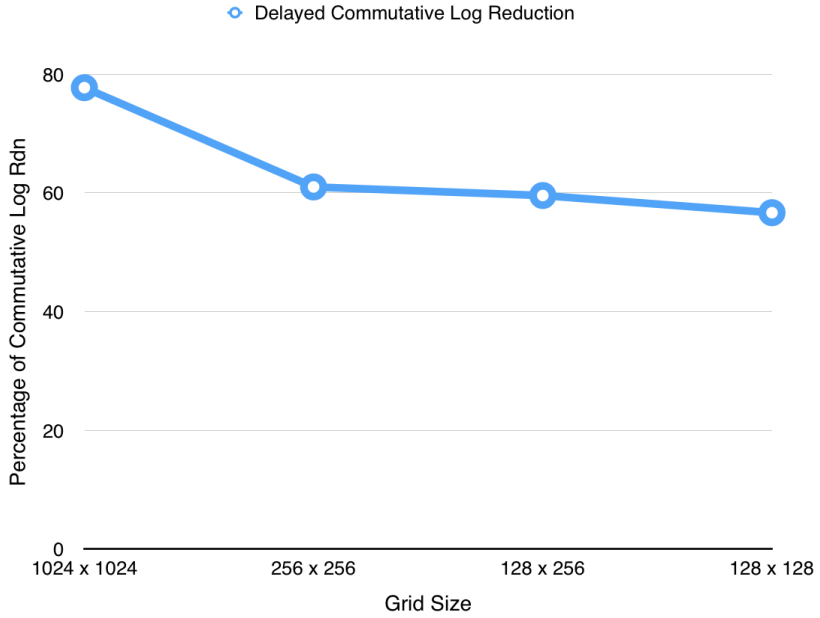
The partial reductions within each work group themselves, used local memory, which will exploit the memory hierarchy available on the GPU.

Log Time Reduction

In order to reduce the amount of time the reductions would take, it seemed sensible to implement a reduction in log time. This can be done using a binary tree structure, such that every other processing element "folds" and passes its value to the processing element next to it.

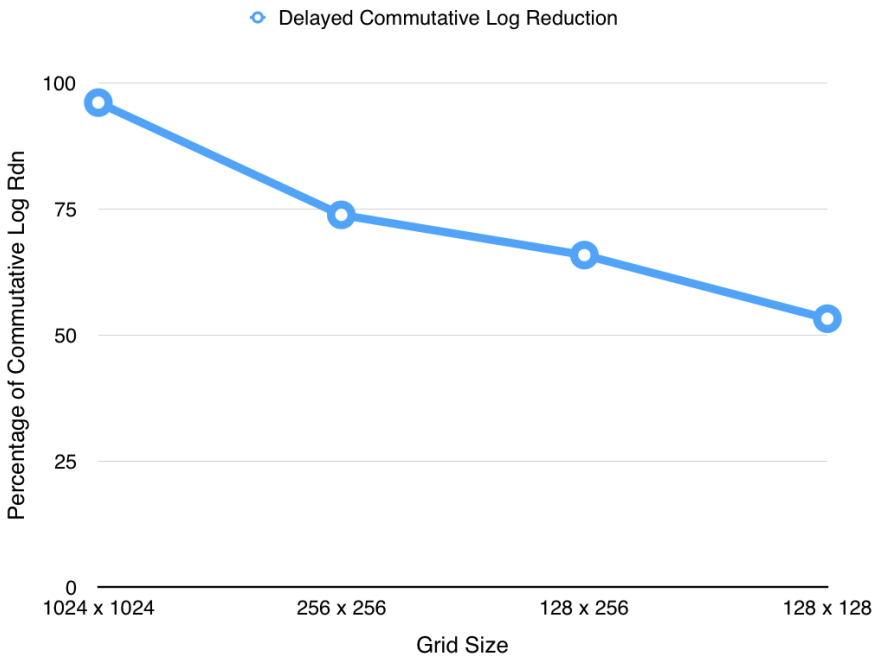
A hidden problem comes in the form of the exploitation of SIMD execution. An abstraction of the cores themselves can be the idea that they are similar to SIMD lanes within a CPU. With the conventional associative binary tree reduction, because data accesses are separated by more than one element, SIMD utilisation is poor. Within a commutative reduction, where the array itself is split in half with elements on one half accessing elements on the other half, rather than next to each other in the tree, active work items are compacted into a contiguous memory block as execution continues. After implementing this reduction, the timings can be seen in Fig 1.1. The smallest grid took almost half as long to complete as before which shows the power of this approach. The main limitation of this approach is that the size of the work groups must be a power of two.

Fig 1.1 Log Reduction Timings



"Delayed" Log Time Reduction

Fig 1.2

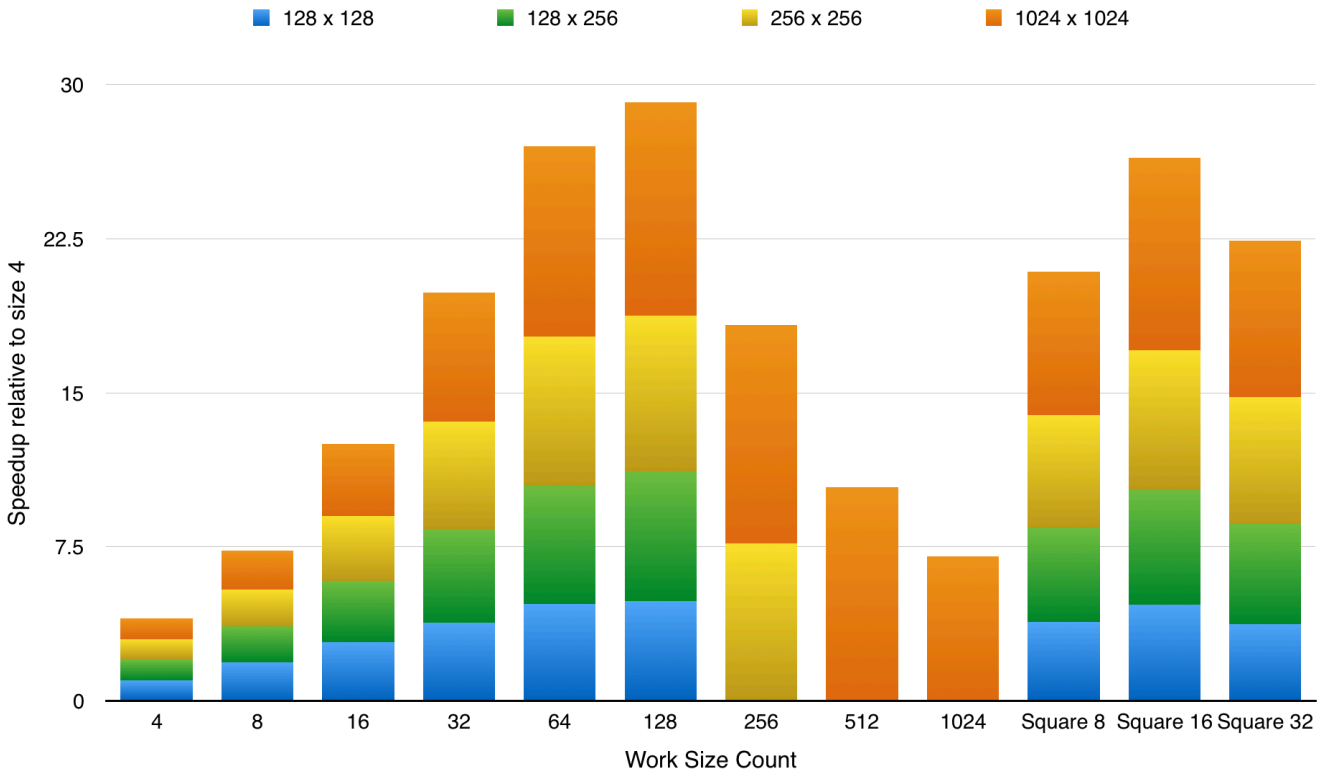


After completing this step, it seemed sensible to reduce the number of global reductions that were occurring at each time-step. This is due to each read operation containing an overhead. By doing the reads once every 200 intervals, (beyond which no further reduction took place), this overhead was reduced. Instead, it was faster to do the reading in one step. At this point I also removed the waiting step that the host performed, further removing half a second from all times.

Analysis

This reduced the execution time by the amount seen in Fig 1.2. The work group size at this point was 128 for all grids. Once again, a reduction in execution time of almost a factor of 2 occurs for the smallest grid against the commutative log reduction, with a steadily decreasing reduction factor as the grid grows larger. After doing testing on which interval (or delay) was optimal, I found that there was a steady decrease of execution time at the value of 200, after which it flatlined and in some cases increased. At this point, I tested different work group sizes. I found that very small work sizes were the worst for performance as can be seen by linear performance on size 4. The fastest work group sizes were those that approached a linear layout of size 128.

Fig 1.3 - Work Group Sizes



One can see from the speedup in Fig 1.3 that this is the highest speed for all of the different work sizes. Interestingly, square work groups did not benefit, even though one may have expected that perhaps there could have been some intangible surface area to volume improvement occurring. The linear sizes greater than 128 were not tested on the smaller grids as they did not work. An obvious question to ask here is the

amount of memory bandwidth that is being used. At this point, the largest grid of size 1024 x 1024 took 12.4 seconds. The amount of memory being transferred to and from the K20 GPU was therefore:

1024 * 1024 * 4 (float size) * 20000 (iterations) * 9 (speeds) * 2 + 1024 * 1024 (obstacles) + 200 * 4 * 1024 * 1024 for the reduction (151,834,853,376) bytes = 152 GB. The K20 has 208 GB/S at most = 2579 GB for the total time. I only used 6% of the memory bandwidth of the GPU.

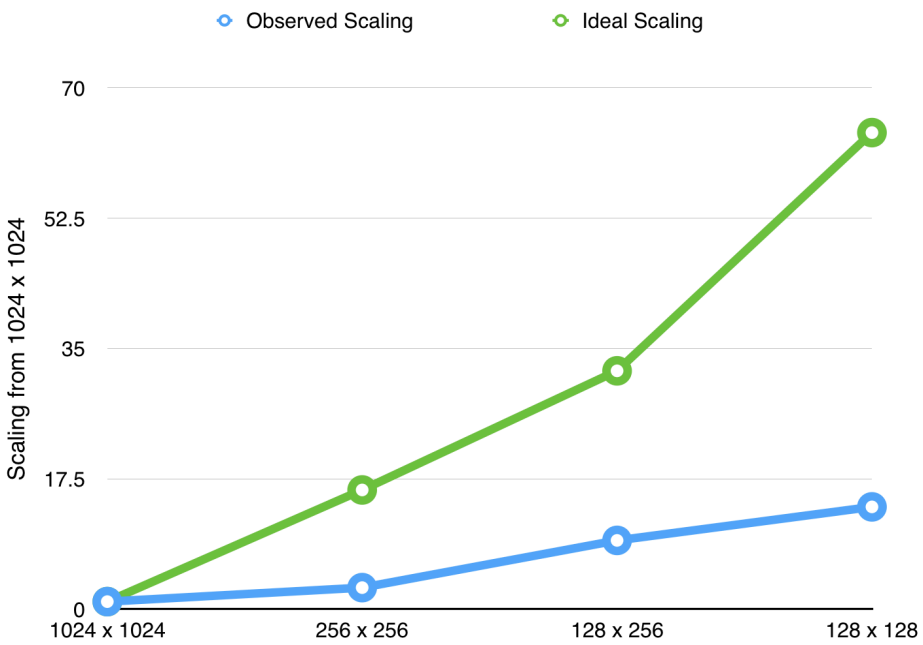
For the smallest grid this was:

128 * 128 * 4 * 40000 * 9 * 2 + 128 * 128 + 200 * 4 * 128 * 128 = 47.2 GB. In 0.89 seconds the maximum memory bandwidth is 185 GB. Therefore on the smallest grid we achieve 26% of the peak memory bandwidth. In the final times, I cannot be sure that the smallest grid is proportionally faster or whether the largest grid is proportionally slower. I suspect due to the memory bandwidth in the smallest grid, that it is faster.

Further work & Final Times

Given more time, I would have combined MPI and openCL and used both GPUs allocated. I would have implemented a basic halo exchange system. I could have also compared the difference to CPU performance. Perhaps, the scaling would have been closer to the ideal scaling had the memory hierarchy been exploited further. I would make a point of researching this.

Fig 1.4 Final Times



Grid Size	128 x 128	128 x 256	256 x 256	1024 x 1024
Time in seconds	0.89	1.35	4.33	12.47