Heterogenerous D2Q9 Lattice Boltzmann Simulation

Jonas Osborn JO14944

	128	256	1024	2048	4096
CPU	0.51	0.66	2.66	32.7	112.3
GPU	1.97	1.98	3.54	7.71	19.67

Table 1: Final Runtimes

The lattice Boltzmann method is a method for fluid simulation that is very well suited for parallelization. I have implemented several optimisations to try and let this algorithm scale as well as possible across both CPUs and GPUs. The runtime for my flat MPI code and MPI + OpenCL code can be seen in Table 1 and I will analyse these results at the end.

1 The Algorithm

The first change I made to the lattice Boltzmann program we were supplied with was to change the algorithm, I switched from the two-step two-grid system with separate propagation and collision to a one-step two-grid algorithm where collision and propagation are fused. The algorithm reads from grid A, collides and writes back to grid B and after iterating through all the cells, swaps A and B. The one-step system uses half the memory bandwidth which I anticipated to be the main bottleneck for performance alongside communication. With the two-step algorithm there are two choices for when to perform propagation: either before, whereby neighbouring cell's PDFs are pulled into the computed cell and collided, or after, where the cell is collided and it's PDFs are pushed to neighbours. As unaligned data access was unavoidable I opted for the pull order as I believed unaligned memory reads would be less damaging than unaligned writes.

After algorithm changes, the next most important change for performance is usually data layout, so I examined the possibilities. There were two main ideas: the array of structures (AoS) layout where the pdfs of a single cell are stored contiguously in memory, and the structure of arrays (SoA) layout where all the pdfs in a particular direction are contiguous. They both have advantages and disadvantages as the AoS layout provides memory access with a stride of 1 when reading or storing pdfs in the same cell, while the SoA layout provides more spatial data locality if you can keep all 9×2 cache lines in cache. The deciding factor for me in choosing the SoA layout was how the GPU accesses memory. As the GPU runs in warps and likes to coalesce memory access, it was ideally suited to the SoA layout, where for each load in the GPU kernel each thread in the warp will be looking for the physically next location in memory.

A significant pitfall with using the SoA layout is that by making a dimension of the grid the smallest step in the array there is a large danger of cache thrashing. This occurs for only certain dimensions but these include all the powers of two. This is due to the set-associative nature of the caches, so pdfs for cells in the same X location but different Y location will map to the same location in cache. This leads to many cache evictions and cache misses and a significant performance hit in the CPU version of my code. I solved it by adding a buffer ghost extension of 8 cells to the x dimension, altering it so that it no longer thrashes the cache, but retaining 32 byte alignment.

2 Optimising Communication

When splitting the work of the program up between multiple cores and nodes these systems obviously need a method to communicate to work on the same problem, this is implemented using a halo exchange. The program sends the outermost region of its portion of the grid and receives the single layer surrounding it necessary for the next stage of computation. An important choice is how the grid is broken down into sections: either into rows, columns or tiles. I dismissed columns as they do not conform to the way I have allocated memory and instead examined tiles vs rows. Tiles presents less memory to transfer for any situation where the number of processes is greater than 4, but does so at the cost of the overhead of communicating with more processes. In this case, a simple tile halo exchange would have each tile communicating with the 8 surrounding tiles which could lead to much more overhead. When we have 4 nodes with 112 cores in total to use however, the smaller bandwidth outweighs the

disadvantage of the overhead, as splitting a grid into 112 tiles rather than 112 rows leads to around 80% less data transfer for each section. Splitting by tiles lead to a 2.72x speed increase over splitting by rows.

The overhead of communicating with the 4 extra corner tiles to only receive and send a single node seemed like an opportunity for optimisation. I changed the communication scheme to run all the horizontal transfers before vertical ones and transmit all corner cells during the horizontal transfers so that they then could be passed on to their intended recipient in the vertical communication. This communication scheme was 3.27x faster than communicating by row only.

Because of the distributed nature of the SoA data layout, the boundary layers could not be transferred immediately as a single unit, they had to be either gathered into a buffer or transferred in 9 different pieces. With overhead involved, transferring nine times will not be efficient, so I looked at how to buffer my messages. For my CPU program I accomplished this by using MPI datatypes, which provided a 1.05x speed increase over manually gathering the data in to a buffer. While for my GPU program I found gathering into a buffer on the GPU was the most efficient method, it could be done during kernel execution for very little overhead.

I experimented with overlapping communication and computation in both my CPU programs and GPU programs, but for both cases found it to be less efficient. It did not improve my runtimes for my CPU program as the asynchronous message sending didnt fit with my vertical communication being dependent on horizontal communication. I had to start transmitting corner cells again to remove the dependency. Overlapping the communication with the computation produced a 1.06x speed benefit here, but this was not as fast as transferring the corners with the edges with no overlap. I also did not find any success trying to overlap with GPU as computing an inner kernel while waiting for MPI transfers to complete, and then computing an outer kernel was slower than simply waiting for transfers and only computing one kernel. This was due to the significant overhead when enqueuing a kernel.

3 Optimising for the CPU

Because of the large amount of variables and floating point operations, along with the shifts in loading or storing in the main loop of the Lattice Boltzmann code, the compiler has trouble vectorising it. To make sure that everything was vectorised in the most effective way, I wrote the main loop of the CPU code in AVX intrinsics, leading to a 4.3x speedup. To assist with easy vectorization I changed the obstacle array into an array of floats. This allowed me load it as a vector and therefore multiply using it to mask results rather than relying on control flow inside vectorised portions to determine if cells are fluid or not. This also helped for when the code was transposed to an OpenCL kernel as they do not do well with brnaching.

This also allowed me to design the arithmetic to utilize much of the newer FMA instruction set, which include the instructions that combine multiplication and addition into one single instruction. Rearranging the arithmetic to make use of this led to a small 1.05x speedup.

Loop distribution is the opposite of loop fusion and is a technique I used to improve my cache access in my main loop. I encountered multiple cache-misses trying to compute propagation, collision and average velocity calculations for all 9 directions in one loop, so distributed the inner loop into 4 separate loops. The first calculates component velocities and the density, the second propagates and performs relaxation on the first 5 directions, the third does the same on the other four directions and finally there is a fourth loop where average velocities are calculated. This optimisation lead to much less frequent cache-misses and a speed up of 1.2x.

I experimented with using non-temporal (NT) stores which use intrinsics to write data that does not have to obey normal cache coherency rules. To maximise the cache use they effectively instruct the processor to write straight to memory. After calculating the new values for a cell you should not need to read those values again until you are processing the next iteration of the matrix, by which time the cache line will have been evicted anyway. Unfortunately the necessity to save the average velocities each timestep complicated this, as part of the loop splitting my program would read back the saved velocities soon after they were stored to compute the average velocity. I found that the slowdown from rearranging average velocity to accomodate NT stores outweighed the benefits and so did not implement this in my final version.

4 Optimising for the GPU

The main bottleneck for my OpenCL implementation of the lattice Boltzmann method was enqueuing kernels and transferring data between host and device. I alleviated this by performing as few transfers as possible. I save all my average velocity calculations until the end of iteration to transfer back to host and only transfer the minimum pre-packed for halo exchange. I

experimented with two different methods of improving bandwidth between host and device, namely using pinned memory and zero copy buffers. Pinned memory allows the drivers to heavily optimise transfers to and from the GPU as it has a guarantee that the memory will remain in the same location, while Zero-Copy should be an exact map of memory from the device onto the host. I expected the zero-copy memory to be the fastest, and while it was a little bit faster (1.03x) than normal paged memory, pinned memory was a little faster still (1.05x), possibly because MPI interfaces better with pinned memory.

For both pinned and zero-copy memory I tried switching MPIs communication method from two-way sends and receives to remote memory access (RMA), using the active target mode and the post-start-complete-wait synchronisation method, as each process only communicated with two others. I found this to be slightly (0.97x) slower than the two-way system of sends and receives. I again tried overlapping communication and computation with RMA but this produced no difference.

I determined my local work size by testing all possible values on the 1024x1024 grid. The global work had to be divisible by local work groups and I wanted both the local work size and local work X dimension to be a multiple of 32, which is the warp size. This gave me 64x8 as the ideal work size.

I implemented a parallel reduction for calculating the average velocity in the kernel in local memory, making sure that threads aim to avoid bank conflicts by not reducing symmetrically. The program with this reduction implemented was 1.13x faster than a program with just one thread of the work group summing average velocities.

I also experimented with both rewriting the kernel to re-use registers as much as possible and not re-using registers at all. I tried re-using them to minimise register spilling and enable high occupancy, and tried not re-using registers to enable register dependency latency hiding. I found negligible difference between these two kernels at all.

5 Results

The dimensions of the supplied parameters meant some of the results I received were surprising at first, but made sense once I tested and compared against the runtime of input with larger dimensions. I have included some of these 2048x2048 and 4096x4096 results in this section, the input for these sizes was just the same lid driven cavity grid scaled up.

The scaling of the CPU program shown in Figure 1 is the most typical result we have here. There is an obvious trend of the algorithm to process faster as the number of cores increases. The nodes can be quite clearly seen as significant drops when they are added, apart from the fourth, while adding additional sockets often do not decrease the runtime a lot. This implies that the communication between the nodes is the bottleneck, as increasing the bandwidth by splitting the communication among more of them produces a faster runtime, while adding more computing power and memory to a node already saturating it's internet connection is not too beneficial.

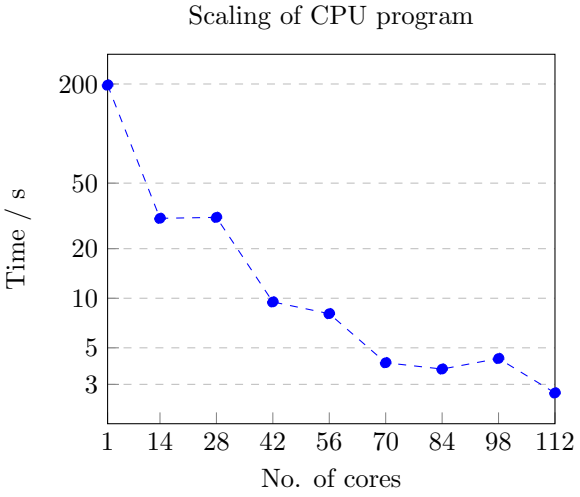

Figure 1: The runtime of my CPU program running a 1024x1024 simulation on different numbers of cores.

The results we see for the GPU at various sizes in Figure 2 is less what we expected to see, as the slowest result with 4 GPUs is the small problem size of 256x256. This is because we have hit a very definite bottleneck at these low problem dimensions where extra computing power is not necessary.

The trend of the 256x256 problem size is that the runtime gets slower as the number of GPUs increases. This is due to adding significantly more time in communication that it removes by making the simulation easier to compute. The 2048x2048 simulation size is where we see a more typical trend of more computing power leading to faster runtimes, the 1024x1024 size is left somewhere between the two.

With the much larger memory bandwidth and parallel computing power of the GPU accelerators, we would expect to see the GPU program run at much faster speeds than the CPU program, but as we can see in Figure 3 this is not completely the case. The GPU version only becomes faster for the same problem size once we are past the 1024x1024 simulation size. This is because what we are actually timing in these small

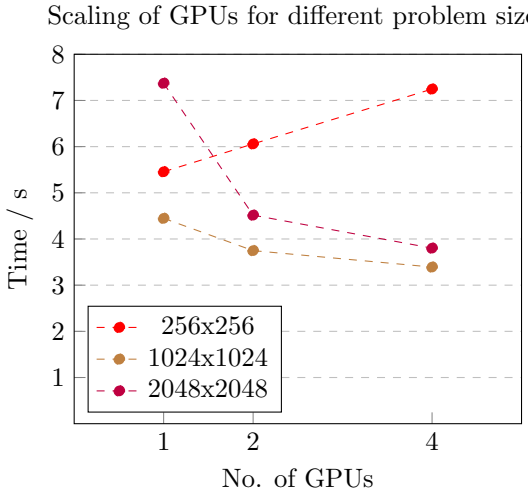

Figure 2: The runtime of my GPU program running on different numbers of GPUs

sizes is predominantly communication time, libraries and interfaces between the components. The simulation is now only a small part. The CPU is faster for low problem sizes as the only main overhead we have is that of MPI message passing. While on the GPU we have the MPI overhead, but also the transfer of memory between device and host and the overhead of queueing kernels. Once we start running the larger problems overhead matters less, it is clear to see the GPU is faster with it simulating the 4096x4096 grid at 5x the speed of the CPU version.

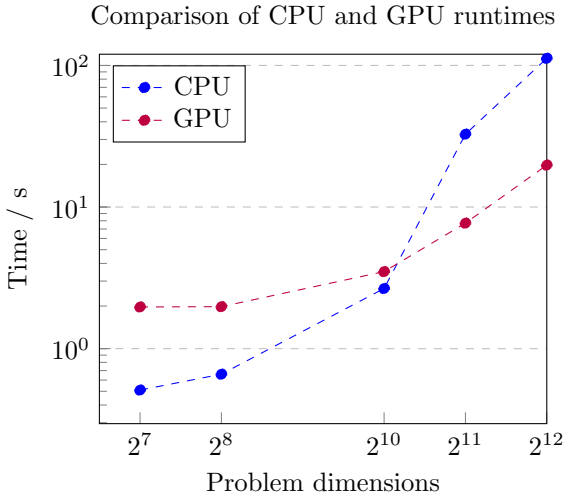

Figure 3: The different runtimes for both CPU and GPU programs running at different dimensions

6 Considered Optimisations

In my research surrounding the lattice Boltzmann method and methods we can use to optimise it, I encountered several techniques that I would have like to have tried but did not have the time to try and implement.

The one-step two grid algorithm is by no means the most efficient implementation of the lattice Boltzmann kernel. There are several that can provide the same benefits with only one grid worth of storage space required. Algorithms suchs as the AA-pattern and Esoteric twist are the most memory efficient ways to compute the lattice Boltzmann method but neither were very well suited to this problem. The Esoteric twist and other algorithms like the 'Swap' algorithm rely upon processing the cells of a lattice in a particular order. This does not fit well with GPU computing where multiple work items are processed at the same time. The AA-pattern does not require an order of execution however it splits propagation and collision in to two alternating kernels. I attempted to reconcile these with the task of producing average velocities for each timestep but could not finish this in time.

The main bottleneck with my GPU program is time spent communicating and copying memory through the host. Porting the code to CUDA would enable use of technologies such as GPUDirect that would enable Remote Direct Memory Access between the devices over infiniband. This would remove a lot of the overhead when running the GPU code at small problems sizes.