Advanced High Performance Computing - OpenCL

Riley Evans (re17105)

Starting Point The runtimes of the Lattice-Boltzmann (D2Q9) program can be seen in Table 1. They have been generated using CUDA 10.0 with GCC 5.4.0.

Table 1: Runtimes of the unoptimised program

Size	Runtime (s)
128x128	45.633
256x256	331.826
1024x1024	1277.081

Porting to OpenCL Not all kernels have been ported to OpenCL. This means that there is an unnecessary amount of copying data between the host and the device. By porting all the kernels to OpenCL this can be prevented. Reducing the copying to just one write and one read, for the cells grid.

The reduction used will be reasonably simple. It will use one work-item in a work-group to sum up all the values inside that work-group. The values will be copied back to the host and summed up every iteration. Alternative reduction techniques will be experimented with after all the basic optimisations have been made. The speedup achieved after porting to OpenCL can be seen in Table 2.

Table 2: Runtimes after porting all kernels to OpenCL

Size	Runtime (s)	Speedup
128x128	4.831	9.45x
256x256	18.130	18.30x
1024x1024	78.133	16.34x

Now that all kernels are running in OpenCL there is no longer need for `clFinish` after each kernel has been enqueued: this is because the queue is in-order, meaning that the kernels will be executed in the order that they were queued in. Having `clFinish` prevents the next kernel from being queued until the current one has finished.

Table 3: Runtimes after removing `clFinish`

Size	Runtime (s)	Speedup
128x128	3.693	1.31x
256x256	15.393	1.18x
1024x1024	77.465	1.01x

Fusing Kernels The program has four kernels, that all access the same cells for a single instance of the kernel. They are: `propagate`, `rebound`, `collision`, and `av_velocity`. It would make sense to fuse these kernels together. This will reduce the number of times the same cell is fetched from memory. The speedup achieved by doing this is shown in Table 4.

Coalesced Memory Access Another step that can be taken to improve the speed of the program is to change the memory access pattern. Currently, the grid of cells is stored in an array of structures. This, however, is not the optimal way to store it. When `cells[ii + jj * nx].speed[5]` is accessed for different `ii` and `jj` values, it will be striding over memory. This means that the program is not making optimal use of cache lines, nor will each work-group be able to load a sequential block of memory for each instruction. A structure of arrays (SoA) format is best suited to SIMD architectures.

To do this, however, poses a problem: the program cannot simply pass a structure of pointers into the kernel as an argument. This is because the `cl_mem` values stored on the host may not be a pointer to the memory on the device. Passing each buffer as an argument to the kernel, will avoid this problem. The speedup created by this transformation can be seen in Table 5.

Table 4: Runtimes after fusing the `propagate`, `rebound`, `collision` and `av_velocity` kernels

Size	Runtime (s)	Speedup
128x128	2.811	1.31x
256x256	8.811	1.75x
1024x1024	24.911	3.11x

Table 5: Runtimes after using coalesced memory access

Size	Runtime (s)	Speedup
128x128	1.916	1.47x
256x256	4.651	1.89x
1024x1024	4.682	5.32x

Slow Host Code The program is now using an SoA format. The runtimes, however, are unexpected: the time to run 128x128 and 128x256 are almost equal, similarly with 256x256 and 1024x1024. It would be expected that the time of the first size would be noticeably faster, especially in the latter case. After investigating further why this may be the case, one suspicion could be that the host code is slowing down the execution of the program rather than kernels.

The runtime of the host code can be tested, by removing all computation from the kernels. If the program still runs with similar runtimes, it would suggest that the host code is slowing down the program. The times generated from this experiment, can be seen in Table 6.

Table 6: Runtimes after removing the computation

Size	Runtime (s)
128x128	1.922
256x256	3.887
1024x1024	3.036

These results are not below the ballpark times given. This would suggest that the host code could be optimised further. This leads to the question: *Is enqueuing kernels an expensive operation? If it is can it be optimised in some way?* After experimenting with the queuing of kernels one thing becomes apparent. There is a linear relationship between the number of arguments and the time needed to queue them. This can be seen in Figure 1. This effect is noticed when the arguments are buffers much sooner than if the arguments are integers. Therefore, it would make sense to try to minimise the number of arguments to a kernel that are buffers.

Reducing the number of kernel arguments There are two ways to reduce the number of arguments that are buffers. One method is to initialise a single buffer to store the whole grid and access it as a 3D array. Another is to create a structure on the device, which stores pointers to the buffers. The latter has some caveats: a new kernel would need to be created to convert the buffers to pointers on the device. Another kernel would be needed to perform a pointer swap.

The results from attempting both of these methods can be seen in Table 7. From the results it appears that the

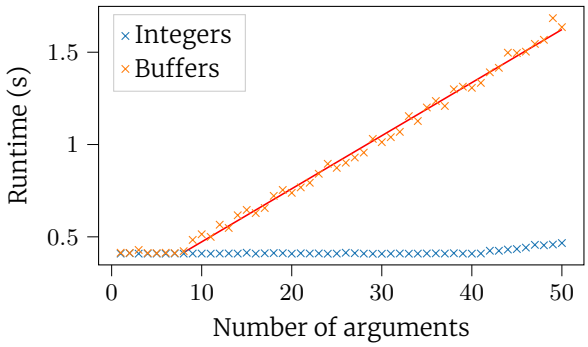

Figure 1: Time to queue 50000 kernels with n arguments.

second option is faster. However, in the next section, when optimising the reduction, the first option produces faster results.

Table 7: Results from reducing the number of arguments

Size	Runtime (s)	
	3D Buffer	Device Struct
128x128	1.377	1.039
256x256	3.565	2.505
1024x1024	4.617	4.740

Reductions The current reduction is not the most optimal choice. Rather than a single work-item summing up all the values, this could be done by multiple work-items. Using a binary sum allows us to do this. An example of this can be seen in Figure 2

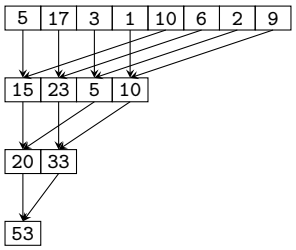

Figure 2: Binary Reduction.

Additionally, every iteration `tot_cells` is reduced. However this is constant throughout all iterations. Therefore this only needs to reduced this once. The speeds achieved for each type of reduction, with only `tot_u` reduced can be seen in Table 8.

There is a further optimisation that can be made. The reduction values are only required at the end of the simulation. The work-group sums can be stored for all iterations. These can all be summed up on the device after all iterations have completed, then copied back to the host as an array of `av_vels`. This helps since partial sums no longer need to be copied back from the device every iteration. The `clEnqueueReadBuffer` command is a blocking command. This produces a problem that is similar to the issue with `clFinish` previously identified. This stops the next kernel from being queued until the memory has been copied from the device. The speedup gained from this can be seen in Table 9.

This has reduced the memory transfer between the device and host, however it still leaves one possible area for improvement: the binary reduction. Although it reduces the amount of idle work-items, there is still a large number of work-items that do not do any work. Half the work-items do no more work after calculating the initial `tot_u` value for their cell. If there was a way to ensure that all work-items performed the same work this

Table 8: Runtimes of both types of kernels, without the `tot_cells` reduction.

Size	Runtime (s)	
	Local Memory	Binary Reduction
128x128	0.990	0.958
256x256	2.664	2.529
1024x1024	3.696	3.655

Table 9: Runtime from reducing memory transfer

Size	Runtime (s)	Speedup
128x128	0.522	1.84x
256x256	1.059	2.52x
1024x1024	3.087	1.18x

would surely lead to a speed increase in the reduction. This could be done by storing copies of board for multiple iterations. Then using a single work-item to compute the reduction for each iterations board. This ensures that all work-items stay busy during the reduction.

This, however, has one major drawback, it will use a significant amount of memory to store every value of `tot_u` for every iteration. For the 128x128 input, this would require 2.6GB of memory on the device. This will be even higher for the bigger sizes. To avoid this iterations can be chunked into groups of 3584 iterations. Then perform a reduction on those iterations and copy the values back to the host. 3584 is significant as it is the number of cores on the Tesla P100. This ensures that all cores stay busy for the duration of the reduction. The results from trying this method can be seen in Table 10.

Table 10: Runtime from attempting to reduce the number of idle work-items

Size	Runtime (s)	
	1792 Chunk	3584 Chunk
128x128	0.520	0.514
256x256	1.498	1.217
1024x1024	5.464	-

The results in Table 10, show that this optimisation does not help to speed up the reduction. It is marginally faster than the previous method for the 128x128 size. However it also causes 1.77x slowdown in the 1024x1024 size. The chunk size of 3584 could not be used for the largest sized grid, due to not being able to allocate a buffer large enough. Although the device has enough memory, it is unable to allocate a buffer larger than 4GB – 1/4 of the GPUs memory. This method will not be used for the reduction as it does improve the performance of the program.

Branching Branching on a GPU tends to be bad: branch instructions have a high latency, meaning it is best to avoid them wherever possible. The `ast` kernel has a divergent-branch, when checking if a cell is an obstacle. This will mean that there could be many Processing Elements that are performing no computation. Masking and selection can be used to avoid branching. This ensures that all work-items will be carrying out the same work until the point where the final values are stored into memory. The results from implementing this can be seen in Table 11.

The results do not show a major improvement. This may not be have been expected. There could be one main reason why this is the case: the obstacle files that were given are sparse. They are only present on the border

Table 11: Runtime and speedup after reducing the amount of branching in the `ast` kernel

Size	Runtime (s)	Speedup
128x128	0.523	1.00x
256x256	0.989	1.07x
1024x1024	3.138	0.98x

of the grid. This means that in most work-groups the branch would have been uniform. With only a few work items around the edges producing divergent branches. This means that the extra computation that is needed to have a straight line code, does not pay off.

Masking could also be used to merge the `accelerate_flow` kernel into `ast`. This will help to reduce the bottle neck observed earlier, when trying to queue many kernels quickly. Fusing this kernel leads to the results in Table 12.

Table 12: Runtime and speedup after fusing the `accelerate_flow` kernel

Size	Runtime (s)	Speedup
128x128	0.257	2.04x
256x256	0.882	1.12x
1024x1024	3.027	1.04x

Other Minor Optimisations There are a couple of other minor optimisations that can be made. Firstly, the calculation of `y_n`, `y_s`, `x_e` and `x_w` can be optimised. All the grid sizes are powers of two therefore these values can be calculated with Boolean logic and bit operations rather than a ternary operator. An example of this is `y_n = (ny - 1) & (jj + 1)`. This gives on average a 1.01x speedup. Secondly, some calculations can be extracted from the kernels. These calculations are all constant for every iteration and work-item. This will prevent calculating the same value multiple times when it is not necessary. An example of this could be `w3` and `w4`. This also gives on average a 1.01x speedup.

Work Group Sizes Previously, to test the program a work group size of 64x1 (cols x rows) has been used. However now that all other optimisations have been applied to the program different sizes can be evaluated to find the best. The times for some possible work-groups can be seen in Figures 3, 4 and 5.

Figure 3: Runtimes of work-groups on size 128x128

The results from this experiment demonstrate a key point; the size of a work-group is not the only determining factor affecting runtimes. The *shape* of a work-group is also crucial. Using 1x16 and 16x1 as an example, this is clear. The reasons for this are likely to be down to the way each work-group accesses memory. Since the data is stored in a row-major format, the 1x16 would be striding over memory to access cells, whereas 16x1 would not,

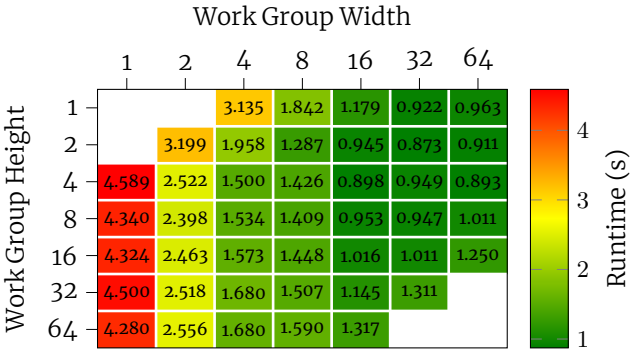

Figure 4: Runtimes of work-groups on size 256x256

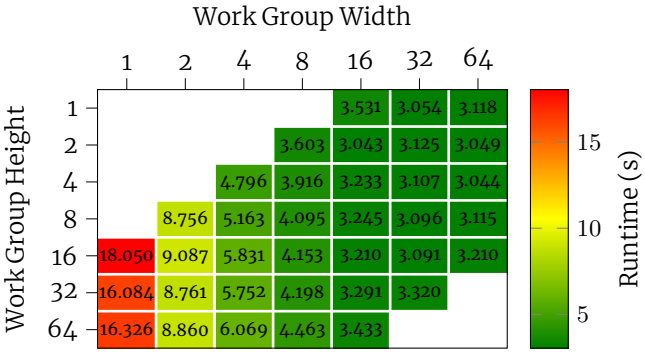

Figure 5: Runtimes of work-groups on size 1024x1024

meaning it makes better use of cache lines. A work-group of 32x1 will be used for the final program.

Roofline Analysis One method that can be used to evaluate the performance of the program is to use a roofline model. This requires calculating some metrics of the device being used - NVIDIA Tesla P100. It has a Peak TFLOP/s of 9.7 and a peak memory bandwidth of 730 GB/s. This allows the Operational Intensity of the ridge point to be calcualted as 13.3 = 9700/730. This can be used to decide if the program is memory or compute bound.

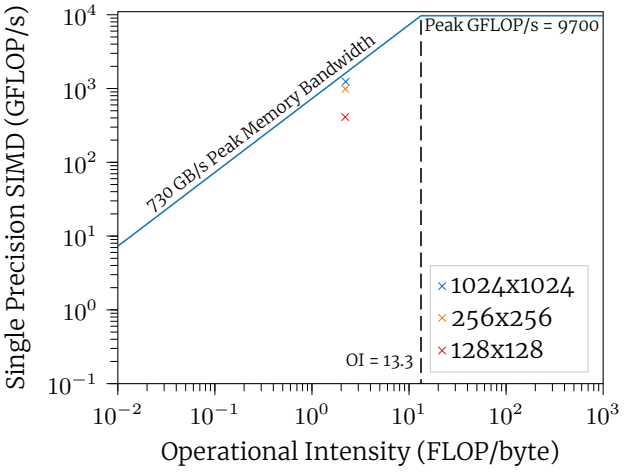

Figure 6: Roofline model of NVIDIA Tesla P100

Using Oclgrind to measure the instructions used inside the kernels for a single iteration, some metrics can be calculated about the program. Each iteration of the 128x128 sized board load/stores 1206896 bytes and uses 2658888 floating point operations.

This gives an operational intensity of 2.20. Therefore the program is memory bound, which would be expected of a lattice type program. The FLOP/s for the program can be calculated using the following formula: $Achieved\ FLOP/s = \frac{FLOPs\ per\ iteration * Iterations}{Runtime}$. This

gives 410.6 GFLOP/s. This can be repeated for the other sizes.

Figure 6, shows the four sizes performance on the roofline model. The 1024x1024 and 256x256 grids both achieved over 60% peak memory bandwidth. They achieved bandwidths of 511.9 GB/s and 443.5 GB/s respectively. The 128x128 grid however has a significantly lower achieved bandwidth of just 186.6 GB/s. It could be possible that the previous effect of slow host code is occurring here as well.

Different Devices OpenCL is also able to run on CPUs, therefore it would make sense to compare this implementation with OpenMP. The results of these tests can be seen in Table 13. The results for OpenCL on a CPU are significantly slower than with OpenMP. One possible reason for this could be the overhead added by OpenCL needed to queue kernels. It is also unlikely that OpenCL will allocate NUMA aware buffers. The OpenCL code on a GPU is also significantly faster than on the CPU. This is because GPUs are designed for SIMD, whilst offering a higher memory bandwidth to the memory on the device.

Table 13: Comparing runtimes of OpenCL and OpenMP

	OpenCL Runtime (s)		OpenMP Runtimes (s)
Size	Tesla P100	Intel Xeon E5-2680	Intel Xeon E5-2680
128x128	0.259	3.524	0.992
256x256	0.863	16.547	4.680
1024x1024	2.997	50.830	20.744

OpenCL allows for kernels to be run across different devices, therefore testing performance on different devices would be sensible. The two extra GPUs that will be tested are the NVIDIA Tesla K20m and GeForce 1660Ti. The results from testing this can be seen in Table 14. The results for the Tesla K20m are as expected, it is the slowest of the three GPUs and produces the slowest runtimes. The GeForce 1660Ti however, which is slower than the Tesla P100, has one surprise result: the 128x128 size is faster than on the Tesla P100. One credible reason why this could be the case, is that consumer processors are designed for shorter burst jobs.

Table 14: Comparing runtimes of different GPUs

	Runtime (s)			
Size	Tesla P100	Tesla K20m	Tesla V100	GeForce 1660Ti
128x128	0.259	0.633	0.206	0.194
256x256	0.863	3.795	0.520	1.837
1024x1024	2.997	11.779	1.987	6.552

OpenACC Another method for running the program on a GPU is OpenACC. It allows for programs to be make parallel through the use of pragmas. A structured data directive can be added to the program. It specifies how memory is allocated on the device and when it should be copied to/from the device. As an example `#pragma acc data copy(cells[:nx * ny * 9]) copyin(obstacles[:nx * ny]) create(tmp_cells[:nx * ny * 9])` states that `cells` and `obstacles` are copied to the device when the block is entered and `cells` is copied back to the host when the block is left. A scratch space called `tmp_cells` is created, which does not get copied to or from. A parallel loop pragma can be added to the loops iterating over `nx` and `ny`.

Table 15: Runtime of OpenACC version

Size	Runtime (s)
128x128	2.651
256x256	3.764
1024x1024	5.068

The times achieved using OpenACC can be seen in Table 15. These times are much slower than the times seen using OpenCL. Investigating using NVVP, one problem with the program is that it copies back the average velocity every iteration. This does not allow the program to utilise the full bandwidths when transferring data from the device to the host, causing the program to slow down. This can be avoided by using a kernel pragma, to inform the compiler that computation should stay on the GPU. This however, did not help, instead increasing the runtime.

Host Languages Previously throughout this report, there have been many occasions where the host code has been the limiting factor when running the program. Therefore two further host languages will be tested: Python and Haskell. These make for two interesting choices. The Python API for OpenCL is significantly more simple than the C API. This means writing the host code for OpenCL will likely be easier, however it could come with some performance hits. The Haskell API is a middle ground for usability, however unlike Python, it is a compiled language meaning it will likely be fast.

Table 16: Runtimes using different host languages

	Runtime (s)		
Size	C	Python	Haskell
128x128	0.259	2.857	0.451
256x256	0.863	5.797	0.978
1024x1024	2.997	3.105	3.025

The results from this experiment can be seen in Table 16. For the larger size grid 1024x1024 all languages have a similar runtime. Each of their runtimes are in the order that would have been expected. There are however some unexpected runtimes on the smaller sizes using Python. The 256x256 time is slower than the 1024x1024 time. The 128x128 is also 10x slower than the C version. Conclusions can be drawn by inspecting the number of iterations used on both of the sizes. 256x256 has twice as many iterations as 128x128, whilst having a runtime that is double 128x128's. This is further evidence to suggest that these sizes are bound by the time it takes to schedule the kernel for each iteration, rather than the runtime of the kernels.

These results would suggest that Python is a sensible language choice, if the kernel execution time is greater than the time taken to queue it. In most other cases, Haskell or C would be a sensible choice as they are able to queue kernels faster. Meaning you are able to use kernels that have a lower runtime without the overall time being bound by the host code.

Final Runtimes

Table 17: Final runtimes of the program

	Runtime (s)		
Size	Given program	Optimised OpenCL	Speedup
128x128	45.633	0.259	176.2x
256x256	331.826	0.863	384.5x
1024x1024	1277.081	2.997	426.1x