Approach Overview

Propagation

The overall method for the finite-difference propogation was as follows:

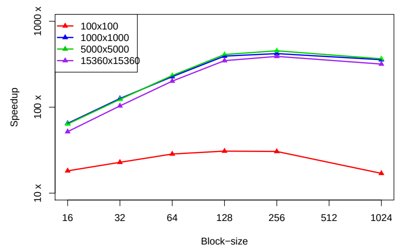
- Allocate an array of size $n \times m$ on the GPU's global memory using `cudaMallocPitch`
- Using `cudaMemcpy2D`, transferred the array containing the initial conditions initialised form the host to device
- Set block dimensions to be `1` $\times$ `block_size`. Making the x-axis of each block any greater than 1 is redundant as all communciation is done across the y-axis.
- Allocate a shared memory array of size `2*(block_size+4)`. This will be split into two arrays, `unew` and `uold`, containing `block_size` elements and the four halo values.
- Didn't use texture or constant memory as they are both constant. Didn't utilise surface memory as I deemed keeping everything on shared memory as much as possible would be optimum.
- Data from global memory transferred to shared memory array at start of kernel.
- Single propagation carried out on shared memory arrays.
- Update `unew` vals transferred back to global memory.
- Repeated `p` times at which point global memory array sent back to RAM.

Reduce

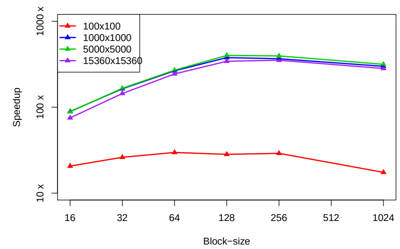
Overall method for vector reduction as follows:

- Allocate shared array of size `block_size`.
- Assign values from array in global memory to shared memory.
- Apply a binary reduction, leaving one sum for each block.
- Write the `gridDim` sums back to global memory.
- Apply previous steps recursively on the remaing values in each row.

- Once each row has been reducded to single value, write back array to RAM using `cudaMemcpy2D`.
- Didn't apply atomics as no threads were trying to access the same point in memory so no need to protect the memory address or sequentialise.



(a) Speedup using shared memory.



(b) Speedup using global memory.

Figure 1: Speedup of calculation times of finite-difference propogation on GPU, using optimisations and using only global memory.

Results

Propagation calculation times

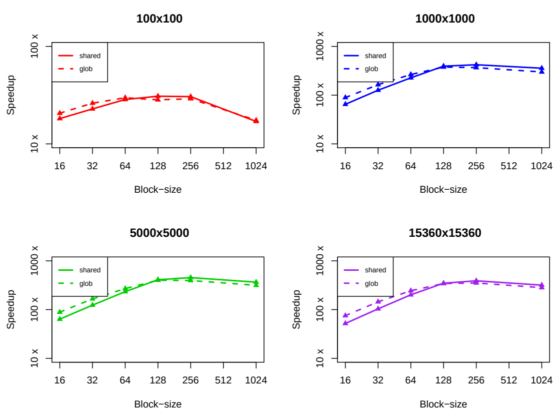
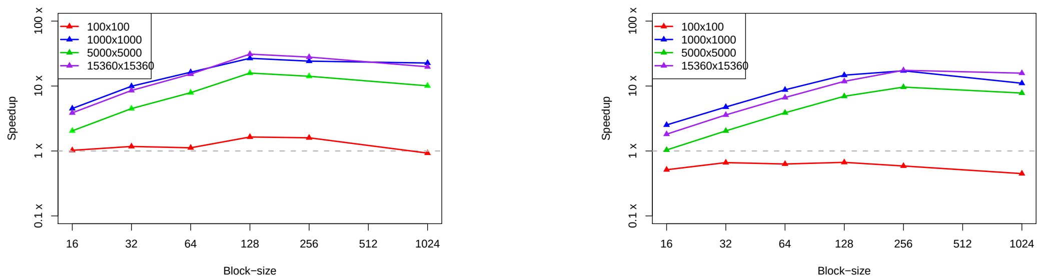


Figure 2: Speedup of calculation times of finite-difference propogation, comparing global memory and optimisations.

Figure 1a shows the speedup of the propogation calulation time using the optimised shared memory approach over the CPU's serial time. The speedup reaches up to $\sim 453\times$. It increases for the first few increases in matrix size, as expected, however, for 15360×15360, there is an evident decrease in speedup. Not clear as to why, it even appears to plateau regarding problem size. Also, interesting to note, there is a clear dip in speedup once the block-size exceeds certain points, depending on the matrix size. For 100×100, this is obvious as a block size greater than 128 will be of no benefit to the algorithm, it is not as obvious for the larger matrices but a dip past block-size of 256 is still seen to a smaller extent. You could extend this to larger block sizes but you would be limited by the max capacity of your shared memory.

The difference in this particular case between using shared memory compared to global is arguably negligible here as can bee seen from Figure 2 and 1b. Both methods appear to follow almost the exact same trend.



(a) Speedup using shared memory.

(b) Speedup using global memory.

Figure 3: Speedup of calculation times of reduce function on GPU, using optimisations and using only global memory.

Reduce calculation times

The reduce function did not experience the same levels of speedup as seen in the propagation. Presumably due to the increased synchronisation costs of performing the parallelised binary reduction. The results in this case actually started off on par or slower than the serial case, and eventually reach $\sim 30\times$ for the 15360×15360 matrix and a block size of 128. The unusual reduction in speedup seen for the larger matrix 15360×15360 is not evident in this kernel. These can be seen in Figure 3a.

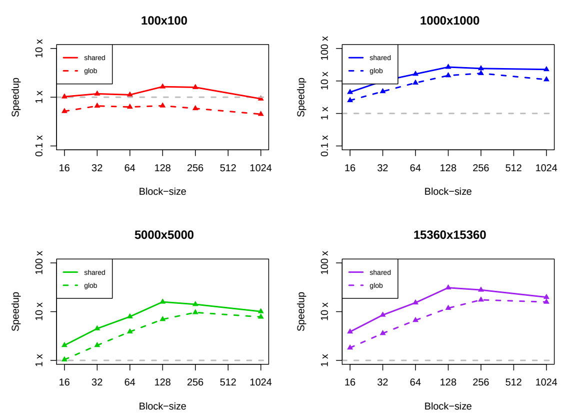
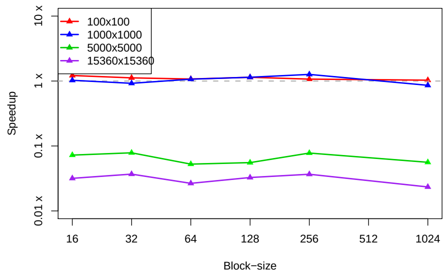
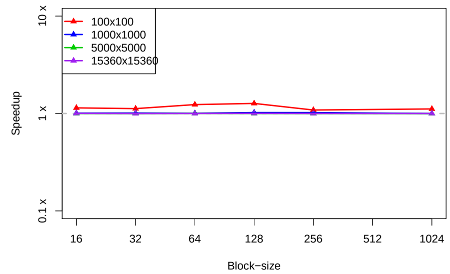


Figure 4: Speedup of calculation times of reduce function, comparing global memory and optimisations

Figure 4 illustrates that the difference in using shared memory versus global which is more obvious here. Expectedly, the shared memory outperforms the global. A dropoff in either case, due to blocksize was not as evident in the case of the reduce function. There does appear to be a slight dropoff but not as significant as seen in the propagation.



(a) Speedup of memory allocation time of `cudaMallocPitch` over `cudaMalloc`.



(b) Speedup of device-host transfer time of `cudaMemcpy2D` over `cudaMemcpy`.

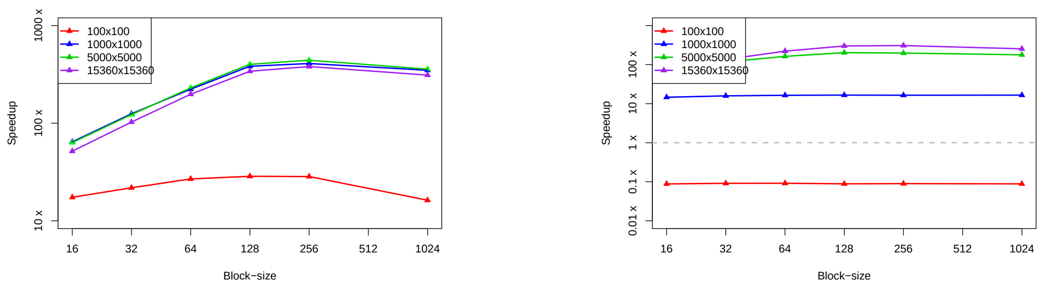
Figure 5: Speedups of allocation time on GPU, and device-host transfer time of `cudaMallocPitch` and `cudaMemcpy2D` over `cudaMalloc` and `cudaMemcpy`.

Allocation & Device-host transfer times

The allocation time for using `cudaMalloc` and `cudaMallocPitch` was tested. Figure 5a would imply that texttttcudaMalloc was orders faster, however, in reality it appeared that whatever kernel was ran second in the program actually ran slower. This is contrary to what was seen in the previous assignment where the first kernel was slower and is a result which I cannot come up with justification for.
The transfer time from device to host on the other hand was pretty much consistent for both `cudaMemcpy` and `cudaMemcpy2D`. No obvious benefits of using 2D data over standard 1D data transfer was seen.

SSE

My SSEs for both optimised and global memory kernels were both reporting 0.00 for all cases. However, I changed something just before submitting and appeared to cause a bug in my shared memory kernel and so, because of this, I am not reporting accuracy values, as for certain problem/block-sizes the values are incorrect. I did not get the chance to fix this bug.



(a) Speedup using shared memory. (b) Speedup using global memory.

Figure 6: Speedup of total calculation times on GPU, using optimisations and using only global memory.

Total time

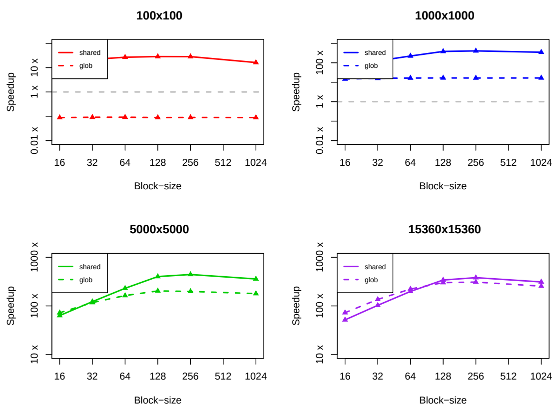
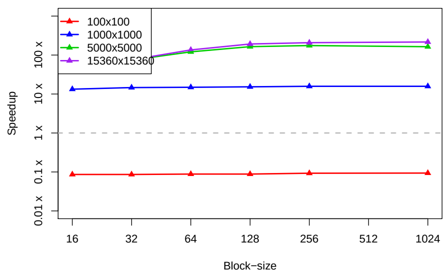


Figure 7: Speedup of total calculation time, comparing global memory and optimisations.

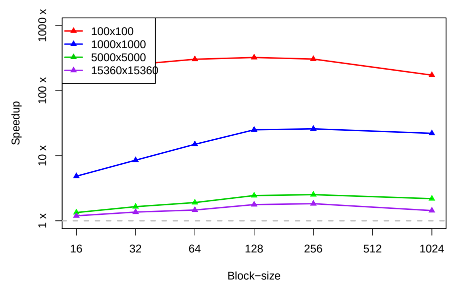
The overall speedups seen are all-in-all, comparable to the results seen in the propagation calculation. An interesting result demonstrating the benefit of using shared memory can be seen in 7, particularly looking at the 5000×5000 case. In this problem, both kernels start off about the same size, but as the block-size grows i.e. the size of the shared memory used, the optimised kernel begins to diverge away and show a more significant speedup while the global memory kernel seems to plateau. Overall speedup reached a max of $\sim 442\times$ compared to serial. These results are all illustrated in Figures 6a, 6b. My guess is this is inflated speedups due to poor serial performance.

Doubles vs floats

Testing single precision speed versus double precision is shown in Figures 8a and 8b. The speedup over CPU is far more constant in double precision rather interestingly.



(a) Speedup over CPU using double precision numbers.



(b) Speedup of single precision over double precision numbers.

Figure 8: Speedup comparisons of floats and doubles.

The single precision values to evidently outperform double precision overall but the speedup reduces by 2 orders as problem size increases, rather strangely.