Approach Overview

Propagation

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

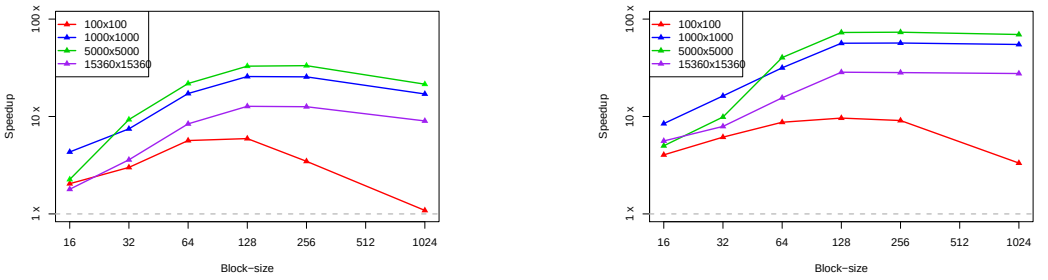
- Allocate an array of size $n \times m$ on the GPU's global memory using `cudaMallocPitch`
- Using `cudaMemcpy2D`, transferred the array containing the initial conditions initialised form the host to device
- Set block dimensions to be `1` $\times$ `block_size`. Making the x-axis of each block any greater than 1 is redundant as all communciation is done across the y-axis. **Note:** Experienced strange bug where my global memory/non-optimised kernel wouldn't execute for some matrix sizes when `dimBlock.x` $\neq 1$.
- 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 2: Speedup of calculation times of finite-difference propogation on GPU, using optimisations and using only global memory.

Results

Propagation calculation times

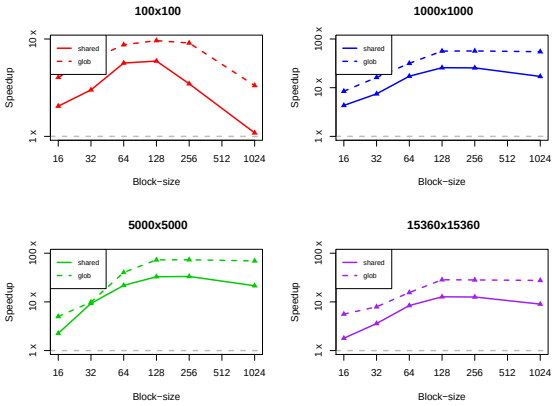
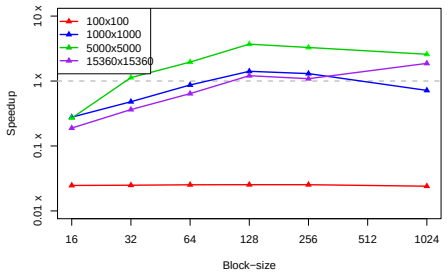


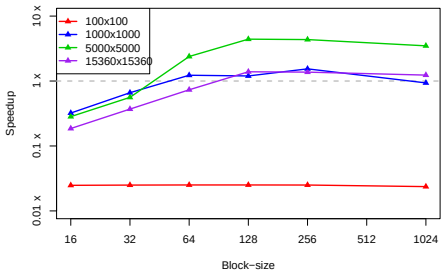
Figure 1: Speedup of calculation times of finite-difference propogation, comparing global memory and optimisations.

Figure 2a 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 $\sim 33\times$. 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. 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.

A rather interesting result, is that using the global memory and no optimisations was actually faster than using the shared memory as is seen in Figure 2b and Figure 1. It can also be seen that for all bar once case, the global memory doesn't experience a slow down as block-size increases, but rather just plateaus. This makes sense as the block-size would have less of an effect when not using shared memory.



(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 significantly slower than the serial case, $\sim 10^{-2}\times$, and eventually reach $\sim 3.5\times$ for the 5000×5000 matrix and a block size of 128. Again, the unusual reduction in speedup is seen for the larger matrix 15360×15360. These can be seen in Figure 3a.

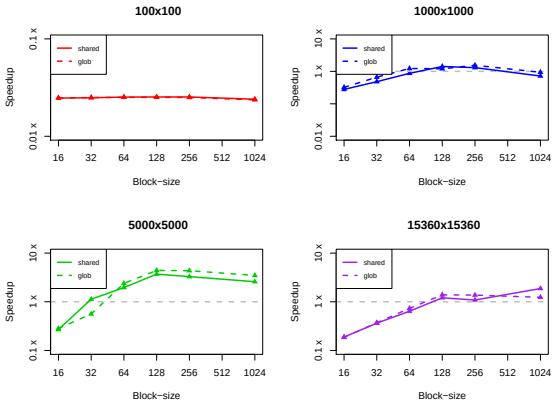
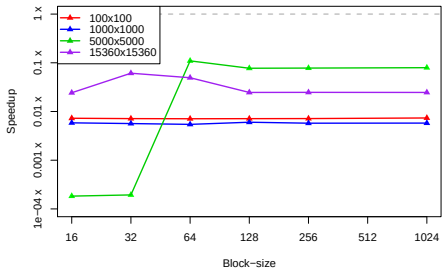
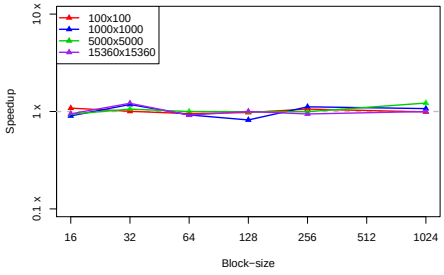


Figure 4: Speedup of calculation times of reduce function, comparing global memory and optimisations

Figure 4 illustrates that the difference in using shared memory versus global is quite negligible. A dropoff in either case, due to block-size was not as evident in the case of the reduce function. For the 5000×5000 the shared memory approach appears to actually keep rising in speedup. However, this would be limited by the maximum size of your shared memory.



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



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

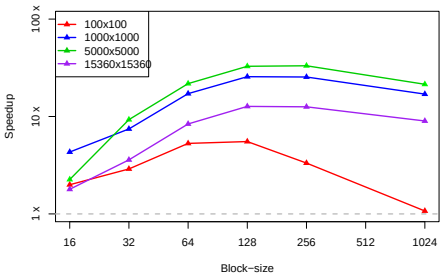
Figure 5: Speedups of allocation time on GPU, and device-host transfer time of `cudaMallocPitch` and `cudaMemcpy2D` over `cudaMalloc` and `cudaMemcpy`.

Allocation & Device-host transfer times

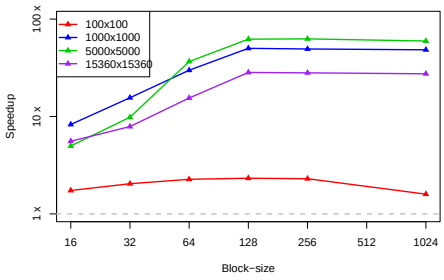
The allocation time for using `cudaMalloc` and `cudaMallocPitch` was tested. Figure 5a would imply that texttttcudaMalloc was orders faster, however, in reality it appeared that whatever kernel was ran second in the program actually ran slower. This is contrary to what was seen in the previous assignment 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

Both of my CUDA kernels were running succesfully and getting SSEs of 0.00. However, I changed something in my shared memory kernel and caused a bug which I couldn't find before handing this report in. Due to this, I haven't reported any accuracy values.



(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

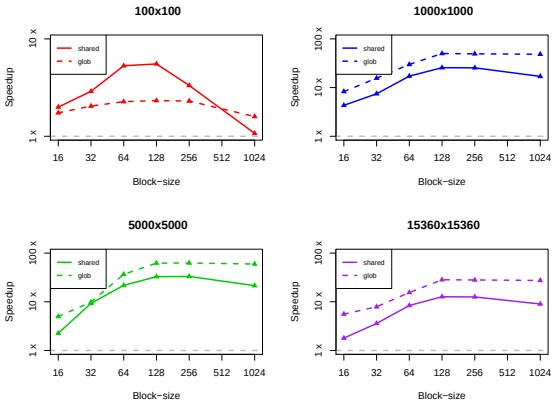
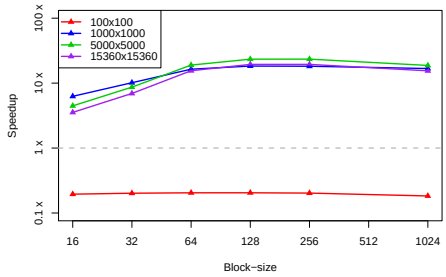


Figure 7: Speedup of total calculation time, comparing global memory and optimisations.

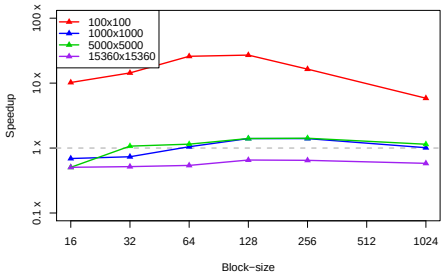
The overall speedups seen are all-in-all, comparable to the results seen in the propagation calculation. The times plateau for global memory and decrease when using shared memory. There is also a reduction for the 15360×15360 matrix and global memory proved faster. Overall speedup reached a max of $\sim 33\times$ compared to serial. These results are all illustrared in Figures 6a, 6b and 7.
Not entirely sure why using shared memory didn't get more succesful results. Was it possibly due to divergent warps? I struggled to deduce other possible reasons from my code as to why the global memory would be faster.

Doubles vs floats

Testing single precision speed versus double precision showed that for 3 or the 4 cases, single precision was slower. Another unexpected result. Shown in Figures 8a and 8b.



(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.