COMP4300 Assignment 2

Jackson Kilrain-Mottram (u6940136)

May 2023

Just before you begin reading this, I wanted to note a few things about this assignment. I've had little to no time to actually work on this during the vast majority of the assignment duration (due to overlapping work and uni assessments that take up my time, hence not grounds for an extension). What has been implemented, evaluated, discussed and presented here is essentially just a week and a half worth of work. Is it optimal? No. Is it the best representation of my work? No. That being said, I think the ideas and techniques presented are well thought through. Cheers for your consideration, hopefully this isn't too painful to read.

1 Question 1

1.1 Performance

In order to properly quantify the performance impacts that the various configurations of OpenMP directives used in each case have, we will utilise a Linux tool called `perf`. This captures event statistics for the compute stack, from the OS level, to kernel and hardware. We are particularly interested in hardware, at the L1 data cache misses and LLC load misses. To give some context to these quantities we will examine, let's first mention that the program has roughly 5.4×10^10 instruction total to be executed for 100 iterations of a 1000×1000 grid.

```c
void omp1dUpdateBoundary(...) {
    // ... snip ...
    #pragma omp parallel for private(j)
    for (j = 1; j < N + 1; j++) { ... }
    #pragma omp parallel for private(i)
    for (i = 0; i < M + 2; i++) { ... }
}

void omp1dUpdateAdvectField(...) {
    // ... snip ...
    #pragma omp parallel for private(i)
    for (i = 0; i < N; i++)
        for (j = 0; j < M; j++)
            V(u, i, j) = ...;
}

void omp1dCopyField(...) {
    // ... snip ...
    #pragma omp parallel for private(i)
    for (i = 0; i < N; i++)
        for (j = 0; j < M; j++)
            V(u, i, j) = V(v, i, j);

}
```

Utilising the performance counters from `perf stat -d`, we see that on average `L1-dcache-load-misses` is approximately 0.72% and `LLC-load-misses` fluctuate between $55 - 62.08\%$. Quantitatively, these equate to 5.6×10^7 in L1 dcache misses and 1.4×10^6 in LLC misses. This implies that there are few misses in the cache for loads, as the locality and alignment of data shared between threads is well situated. The overhead seen here can be attributed to the edge cases between the outside (starting and ending point) of the contiguous part of the advection field accessed for a thread. These are less ideal in their alignment and block saturation, leading to the minimum overhead seen in the L1 data cache load misses.

1.2 Parallel Region Entry/Exits

```c
void omp1dUpdateBoundary(...) {
    // ... snip ...
    #pragma omp parallel for private(j)
    for (j = 1; j < N + 1; j++) { ... }
    #pragma omp for parallel private(i)
    for (i = 0; i < M + 2; i++) { ... }
}

void omp1dUpdateAdvectField(...) {
    // ... snip ...
    for (i = 0; i < N; i++)
        #pragma omp parallel for private(j)
        for (j = 0; j < M; j++)
            V(u, i, j) = ...;
}

void omp1dCopyField(...) {
    // ... snip ...
    for (i = 0; i < N; i++)
        #pragma omp parallel for private(j)
        for (j = 0; j < M; j++)
            V(u, i, j) = V(v, i, j);

}
```

1.3 Cache Misses Coherent Reads

```
void omp1dUpdateBoundary(...) {
    // ... snip ...
    #pragma omp parallel for private(j)
    for (j = 1; j < N + 1; j++) { ... }
    #pragma omp parallel for private(i)
    for (i = 0; i < M + 2; i++) { ... }
}

void omp1dUpdateAdvectField(...) {
    // ... snip ...
    #pragma omp for parallel private(i) schedule(static,16)
    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            V(u, i, j) = ...;
}

void omp1dCopyField(...) {
    // ... snip ...
    #pragma omp for parallel private(j) schedule(static,16)
    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            V(u, i, j) = V(v, i, j);
}
```

1.4 Cache Misses Coherent Writes

```
void omp1dUpdateBoundary(...) {
    // ... snip ...
    #pragma omp parallel for private(j)
    for (j = 1; j < N + 1; j++) { ... }
    #pragma omp parallel for private(i)
    for (i = 0; i < M + 2; i++) { ... }
}

void omp1dUpdateAdvectField(...) {
    // ... snip ...
    #pragma omp for parallel private(i) schedule(dynamic)
    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            V(u, i, j) = ...;
}

void omp1dCopyField(...) {
    // ... snip ...
    #pragma omp for parallel private(i) schedule(dynamic)
    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            V(u, i, j) = V(v, i, j);
}
```

Using `OPENMP_NUM_THREADS=48 perf stat -d ./testAdvect 1000 1000 100` we can keep track of performance counters of hardware behaviour while the advection solution is executing. More specifically, paying attention to the `L1-dcache-load-misses` and `LLC-load-misses`. Form the execution profile, we note that `L1-dcache-load-misses` sits at $6.80 - 14.60\%$ miss rate consistently across multiple runs. For `LLC-load-misses`, this tends to fluctuate heavily between 75.73% overall. From this we can see that cache misses occur frequently for loads that are localised to blocks that overlap between processors. This causes frequent invalidation due to coherence reads between the processors to ensure up to date cache block data. Qualitatively, there is approximately 5.9×10^8 total L1 dcache misses, compared to the

5.6×10^7 for the optimise variation in case 1. Similar,y the LLC misses are an order of magnitude greater in quantity, at 1.35×10^7 on average compared to 1.4×10^6.

1.5 Performance Analysis

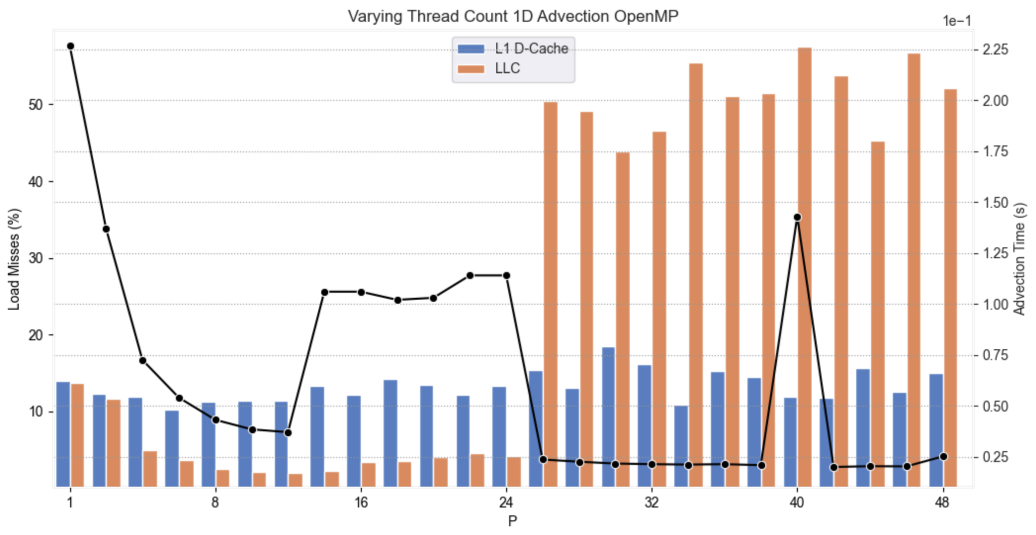

Figure 1: 1D advection time and cache misses (L1 data and LLC) for varying thread counts. Line graph uses right y-axis, bar plots use left y-axis.

Aspect Ratio	Advection Time	Aspect Ratio	Advection Time
1	2.27×10^{-1}s	26	2.36×10^{-2}s
2	1.37×10^{-1}s	28	2.25×10^{-2}s
4	7.25×10^{-2}s	30	2.16×10^{-2}s
6	5.4×10^{-2}s	32	2.13×10^{-2}s
8	4.31×10^{-2}s	34	2.11×10^{-2}s
10	3.84×10^{-2}s	36	2.13×10^{-2}s
12	3.7×10^{-2}s	38	2.08×10^{-2}s
14	1.06×10^{-1}s	40	1.43×10^{-1}s
16	1.06×10^{-1}s	42	1.99×10^{-2}s
18	1.02×10^{-1}s	44	2.03×10^{-2}s
20	1.03×10^{-1}s	46	2.02×10^{-2}s
22	1.14×10^{-1}s	48	2.51×10^{-2}s
24	1.14×10^{-1}s		

Table 1: 2D advection time for varying aspect ratios. From left to right: square, rectangular (one NUMA node, 24 threads), rectangular (two NUMA nodes, 48 threads).

The most optimal configuration detailed in section 1.1, is graphed in figure 1. We can see that for for a single NUMA node, the most optimal configuration was to use a 12×2 thread decomposition. This yielded the lowest advection time, paired with efficient cache performance. Comparing this to the 48 thread configuration across two NUMA nodes, we can see that the performance is better at the cost of more coherence read operations, seens in the increase in L1 data cache and L3 cache behaviour. This makes sense considering there is move movement between nodes and synchronisation between the cores in the nodes. Note that in terms of volume, we have more coherence related misses with the two NUMA node configuration, however a similar percentage of those were misses. This implies that the program behaviour is consistent though quantitatively may still have some room for improvement to scale more optimally in multi-NUMA configurations. One could theorise optimising reads to effectively use the full cache hierarchy to avoid overheads of full writes/reads.

2 Question 2

In order to formulate a model, we need to establish a few points on the hardware properties and problem decomposition. Firstly, using `cat /proc/cpuinfo | grep cache_alignment` we can determine the cache block size for the Xeon 8274

CPUs being utilised. Using this on the Gadi nodes, we see that there is 64 bytes in a cache block. To set the scene, an $M \times N$ grid, requires $\lceil \frac{M \cdot N}{64} \rceil$ cache blocks to fully contain it, these are distributed across the contiguous grid (figure 2).

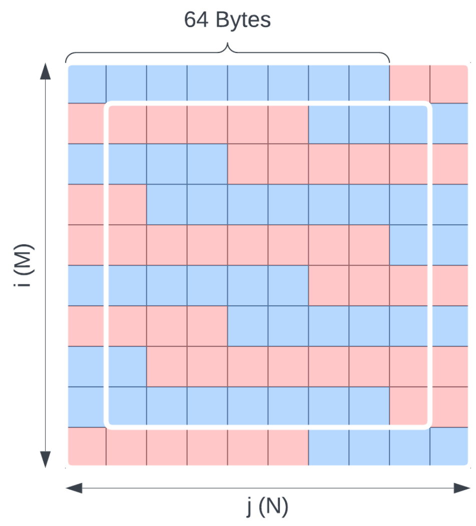

Figure 2: Cache block distribution over an example 2D advection grid.

In order to determine cache coherence behaviour on read and write operations for the solution, we need to first define a function $f : \mathbb{Z}^2 \to \mathbb{Z}$ that maps a 2D grid index (i, j) to the associated cache block it resides in.

$$f(i, j) \overset{\text{def}}{=} \left\lfloor \frac{j + (i \cdot N)}{64} \right\rfloor$$

2.1 Field Update

For the `omp1dUpdateAdvectField` method, there are two sets of operations to consider for coherent-write cache misses and coherent-read cache misses.

2.1.1 Write Misses

Each row starts at a given cache block and ends at the same or another cache block. Using this we can determine how many cache lines (and which they are) are involved. For each write, we need to invalidate these cache lines on each of the other processors. Given that each of the i indices are in parallel, we need to find the maximal cost line update across all processors. Therefore the total can be expressed as follows

$$M_L = \left\lfloor \frac{M}{P} \right\rfloor$$

$$U_w = t_{w,W} \cdot \max_{p \in [0..P]} \left(\max \left\{ 1, \sum_{i=1}^{M_L} [f((p \cdot M_L) + i, N) - f((p \cdot M_L) + i, 1)] \right\} \right)$$

2.1.2 Read Misses

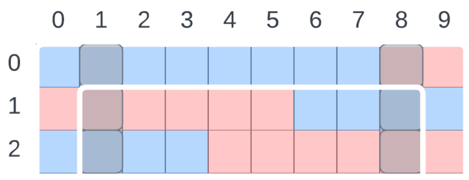

Figure 3: First and last cache block encounters over a subset of the example advection grid.

Each row element has a convolution applied over the previous, current and next row elements. The rows start at a given cache block and end at the same or different cache block. For each transition between cache blocks (as highlighted in figure 3), we will see a coherent read miss. Given we load these rows in chunks for each thread, we consider misses for each row, summed together.

$$U_r = t_{w,R} \cdot \max_{p \in [0..P]} \left(\max \left\{ 1, \sum_{i=1}^{M_L} \left(\sum_{l=-1}^{1} [f((p \cdot M_L) + i + l, N) - f((p \cdot M_L) + i + l, 1)] \right) \right\} \right)$$

2.2 Copy Field

When copying the field from v to u to allow for the next iteration or finalisation to occur, we have both read and write operations occurring simultaneously. Note that these are orthogonal to each other, as the read operations are performed over v and the write operations over u. We can abstract the common max into a single formulation applied to both the read and write coefficients as they equate equivalently.

$$C_{\max} = \max_{p \in [0..P]} \left(\max \left\{ 1, \sum_{i=1}^{M_L} [f((p \cdot M_L) + i, N) - f((p \cdot M_L) + i, 1)] \right\} \right)$$
$$C_w = t_{w,W} \cdot C_{\max}$$
$$C_r = t_{w,R} \cdot C_{\max}$$

2.3 Full Model

Combining the above operational breakdowns, we can achieve the full model over r iterations with an entry/exit cost for $U_{r,w}$ and $C_{r,w}$, ignoring the boundary update (as stipulated by the question):

$$t = r \cdot (2t_s + U_r + U_w + C_r + C_w)$$

2.4 Coefficient Evaluation

Considering the measurement for t_s, we can use two times, one that captures the startup overhead and the second that captures the teardown overhead. For a given parallelised loop, this equates to computing the time to the first loop iteration and the time from the last loop iteration. The following pseudo code illustrates this.

```
time startup_t1, startup_t2;
time teardown_t1, teardown_t2;
startup_t2 = getCurrentTime();
#pragma omp parallel for
for (int i = 0; i < N; i++) {
    if (i == 0) {
        startup_t2 = getCurrentTime();
    } else if (i == N) {
        teardown_t1 = getCurrentTime();
    } else {
        // do stuff
    }
}
teardown_t2 = getCurrentTime();
```

Performing this within the advection solver, we get an average value of 6.74×10^{-4}s for the total startup + teardown overhead. Comparing the to value measured in assignment 1 ($2.951666666667 \times 10^{-4}$), it is indeed close to the cost of startup for an MPI communication. More specifically, it is close to the two way communication startup cost. If we consider only the startup, it is comparatively close to the one way cost at 1.6803624×10^{-5} against 1.47583×10^{-5} (MPI).

We can easily inspect the values of $U_{r,w}$ and $C_{r,w}$ by splitting up each inner most loop into two distinct operations. The first performs reads, and the latter performing writes, where the measured times are written into thread-local array entries. We can then average these to find the coefficients for each of the operation overheads between the iterations that require coherence updates and the ones that don't (reads for instance will require an update on the first iteration indexing a new cache line and writes on every iteration).

	Read ($t_{w,R}$)	**Write** ($t_{w,W}$)
C	6.26285×10^{-6}	2.220465×10^{-5}
U	2.125284×10^{-5}	5.465016×10^{-5}
Average	1.3757845×10^{-5}	3.8427405×10^{-5}

Table 2: Measurements of t_w without timer overheads for update U and copy C parallelised operations with 24 threads and $p = 12$. Note that the startup and teardown time t_s is not included in these measurements as they are from within the iterations.

Compared to the value of t_w from assignment 1 ($6.469449853897094 \times 10^{-10}$), the value calculated here is three orders of magnitude greater. Given that the behaviour of cache coherency is limited to hardware effects on the same node, this is unusually. Especially considering the communication cost for a single double for the MPI implementation is over a network. It is possible that these measurements are skewed by hardware behaviour or low-level OS behaviour such as context switches and other of the like.

3 Question 3

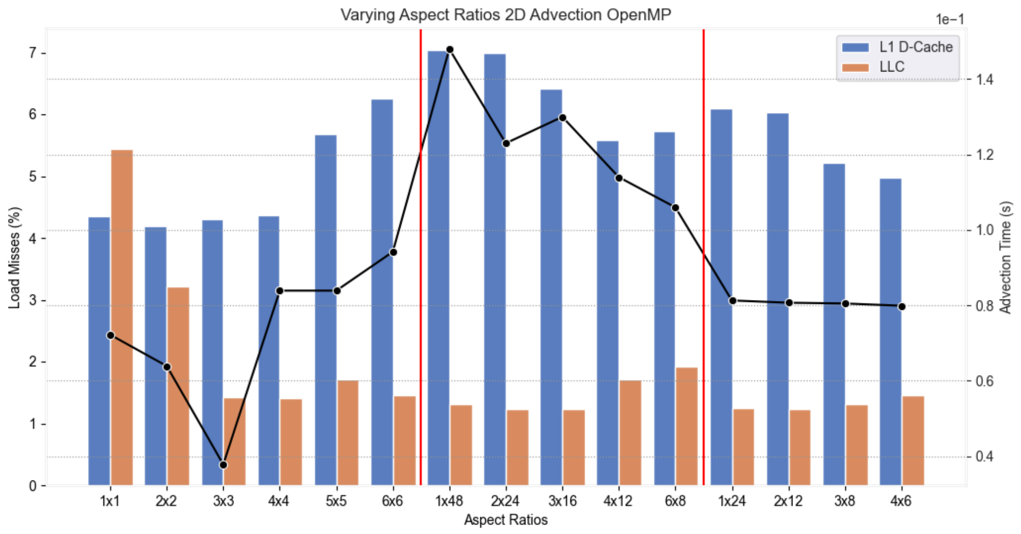

Figure 4: 2D advection time and cache misses (L1 data and LLC) for varying aspect ratios. The first group (left of left red line), are square ratios, the second group (in between red lines) are ratios for 48 threads and the last right group (right of right red line) are for 24 threads. Line graph uses right y-axis, bar plots use left y-axis.

Aspect Ratio	Advection Time
1x1	7.22×10^{-2}s
2x2	6.39×10^{-2}s
3x3	3.78×10^{-2}s
4x4	8.39×10^{-2}s
5x5	8.39×10^{-2}s
6x6	9.42×10^{-2}s

Aspect Ratio	Advection Time
1x48	1.48×10^{-1}s
2x24	1.23×10^{-1}s
3x16	1.3×10^{-1}s
4x12	1.14×10^{-1}s
6x8	1.06×10^{-1}s

Aspect Ratio	Advection Time
1x24	8.13×10^{-2}s
2x12	8.07×10^{-2}s
3x8	8.05×10^{-2}s
4x6	7.99×10^{-2}s

Table 3: 2D advection time for varying aspect ratios. From left to right: square, rectangular (one NUMA node, 24 threads), rectangular (two NUMA nodes, 48 threads).

Relative to the 1D implementation in section 1, we see that the performance of the 2D solver has either tied or slightly worsened when compared to the 1D solution. Interestingly, the cache behaviour is substantially more performant with all configurations within a 7% miss rate for both L1 data cache and L3. This coherence behaviour scales well as the grid scales. Note that since we are working on a 2D grid, the locality of cache lines is consistent with the logical ordering of thread contexts over the grid. This implies well-matched thread-to-cache behaviour, reducing the overheads as the decomposition scales. The 24 core as opposed to 48 core performance difference is more pronounced here, indicating that inter-NUMA node coherence is a significant overhead.

3.1 MPI Comparison

Comparing against the performance improvements of the 2D implementation with MPI from the first assignment, we can see that there is a noticable difference in outcomes between the two programming models. The MPI implementation saw a much greater performance improvement than seen here. Take for instance the two NUMA node configuration from assignment 1:

Aspect Ratio	**Advection Time (A1)**	**Advection Time (A2)**
1x48	3.70×10^{-2}s	1.48×10^{-1}s
2x24	2.73×10^{-2}s	1.23×10^{-1}s
3x16	2.38×10^{-2}s	1.3×10^{-1}s
4x12	3.10×10^{-2}s	1.14×10^{-1}s
6x8	2.62×10^{-2}s	1.06×10^{-1}s

Table 4: Comparison of 2D advection time for assignment 1 MPI implementation against the 2D OpenMP implementation here.

We can see (in table 5) that there is almost consistently an order of magnitude difference in performance. One of the key differences to point out is the ability to hide latency with MPI is much easier than that of OpenMP. In this implementation the latencies for memory operations are consistently present for every operation. The optimisations made could be said to be below what could theoretically be done to improve this difference.

Aspect Ratio	**Advection Time (A1)**	**Advection Time (A2)**
1	2.28×10^{-1}s	2.27×10^{-1}s
2	1.16×10^{-1}s	1.37×10^{-1}s
4	5.87×10^{-2}s	7.25×10^{-2}s
8	3.01×10^{-2}s	4.31×10^{-2}s
16	1.40×10^{-2}s	1.06×10^{-1}s
32	9.42×10^{-3}s	2.13×10^{-2}s
48	1.38×10^{-2}s	2.51×10^{-2}s

Table 5: Comparison of 2D advection time for assignment 1 MPI implementation against the 2D OpenMP implementation here.

Similar performance differences can be observed between the 2D solutions of MPI and OpenMP models. There is less of a difference seen compared to that of the 1D differences previously discussed. This could indicate that the performance of the approach taken in assignment 1 has limitations in its scalability that are comparable to the OpenMP model even with more pronounced latencies (harder to hide). Another point to mention is the volume of communication for the MPI model may well be more of a bottleneck at this stage when compared to shared memory operations. Specifically when considering relative overheads involved.

3.2 Single Parallel Region

It's hard to say whether the single parallel region provided much in the way of improvements. This is mostly due to the 2D implementation being at the same performance or slightly below the 1D for some cases. It would seem that the single region does make a difference, but the cost of the parallel region handling against the cache effects is offset such that little to no difference is observable. This may be more pronounced with an implementation that hides the cache latencies more effectively.

4 Question 4

4.1 Solution Description

For convolution problems that compute time step evolutions, an approach for very large scale matrices is to utilise Fourier transforms (Ahmad et al. 2021). In assignment 1, this was also presented and translated into an MPI implementation based on the original algorithm in the paper by Ahmad et al. (2021). Here, the same algorithm has been implemented, utilising OpenMP as the parallelisation medium. There were several issues with the initial implementation in assignment 1 that have been fixed; missing final rotation of result matrix, incorrect forward DFT parameters and a few other minor points.

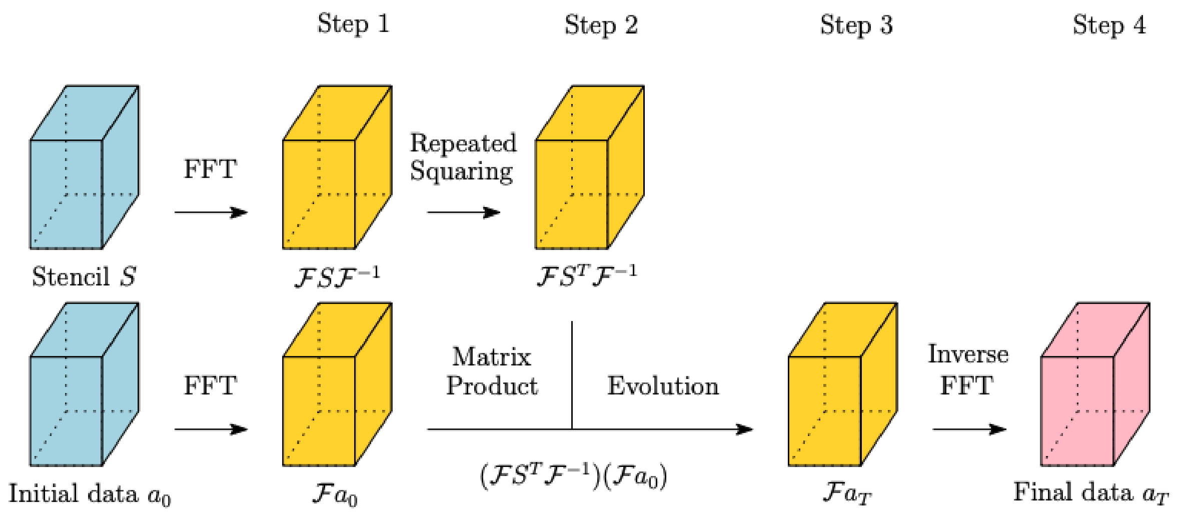

Figure 5: Outline of applying a FFT convolution kernel to an iterated stencil computation tile. Image sourced from Ahmad et al. (2021).

As an overview, this approach applies time step evolution as repeated squaring to the FFT transformed core convolution kernel (figure 5). Given the sparse nature of the matrix, this vastly simplifies the total quantity of matrix-matrix operations required to compute the full evolution. Computing a single matrix-matrix product with the FFT evolved kernel and the FFT evolved data matrix, the full evolution can be computed with an overall smaller computational overhead. Lastly, the inverse FFT is computed and the result rotated $T \mod M$ (where T is the time steps and M is the matrix width).

We can utilise a convolution kernel derived from the Lax-Wendroff method using the coefficient functions of the Courant number used in the definition as described in ?. Essentially, we end up with the following kernel (see the assignment 1 report for further details and motivation on this):

$$c_y = a\frac{\Delta t}{\Delta y}$$

$$b^y_{-1} = \frac{c_y}{2}(1 + c_y)$$

$$b^y_0 = 1 - c_y^2$$

$$b^y_1 = -\frac{c_y}{2}(1 - c_y)$$

$$\mathcal{M} = \begin{bmatrix} b^x_{-1} \cdot b^y_{-1} & b^x_{-1} \cdot b^y_0 & b^x_{-1} \cdot b^y_1 \\ b^x_0 \cdot b^y_{-1} & b^x_0 \cdot b^y_0 & b^x_0 \cdot b^y_1 \\ b^x_1 \cdot b^y_{-1} & b^x_1 \cdot b^y_0 & b^x_1 \cdot b^y_1 \end{bmatrix}$$

$$\forall i \in [0, m], j \in [0, n] \quad u^{n+1}_{i,j} = u^n_{i,j} * \mathcal{M}$$

NOTE: In order to compile and use this, provide the `FFT_CONV_KERNEL=1` *flag alongside the make command, e.g.* `module load fftw && make clean FFT_CONV_KERNEL=1 all`.

4.2 Performance Evaluation

The numerical error for this model was not completely ironed out. The operations are correct in the algorithmic sense, as such the complexity and behaviour of the algorithm is as it should be. However, I was unable to find out what the error was due to time constraints.

As mentioned in the previous sub-section, this FFT method is designed to work with very large evolutionary convolution problems. In order to properly illustrate this, let us consider a matrix of dimensions $10,000 \times 10,000$ over 100 time steps. For both the optimised evaluation and the baseline, we consider 24 threads using a 6×4 decomposition on a single NUMA node to ensure minimal overhead in cache coherence behaviour (as explored in section 2 and 3).

For the baseline we see that the advection time is 1.19×10^2s, which is substantially larger than any of the previous results in this report. Utilising the FFT optimisation, this drops to 2.57×10^1s, equating to a change of -78.4% overall. One could argue that this is still a long time to compute the advection of 100 time steps, however considering that each thread is responsible for a chunk of $\sim 1667 \times 2500$, this is a huge advantage that indicates the standard approach is heavily compute bound in quantity of arithmetic operations.

5 Question 5

To preface this section, I'd like to point out that I was intending to include profiling using the `nv-nsight-cu-cli` *or* `ncu` *tooling, however these would not function for any kernel invocations despite the solver being able to run perfectly well as a standalone program. Given the aforementioned time constraints on working on this within essentially a week, I had to spend my time better elsewhere. I had brought this up on Piazza without a response in time. It's unfortunate, there's nothing I can really do about it. Thanks for your consideration.*

5.1 Performance Evaluation

Comparing the host and GPU execution for a single thread, we can see that a single GPU core is substantially slower than that of a CPU core at two orders of magnitude higher duration.

CPU	**GPU**
9.10×10^0s	7.36×10^2s

In order to determine the performance characteristics, we will use a 4096×4096 grid over 100 time steps. This allows us to scale the grid and block sizes by powers of 2 which fit neatly to the 32 threads in a warp and higher grouping abstractions.

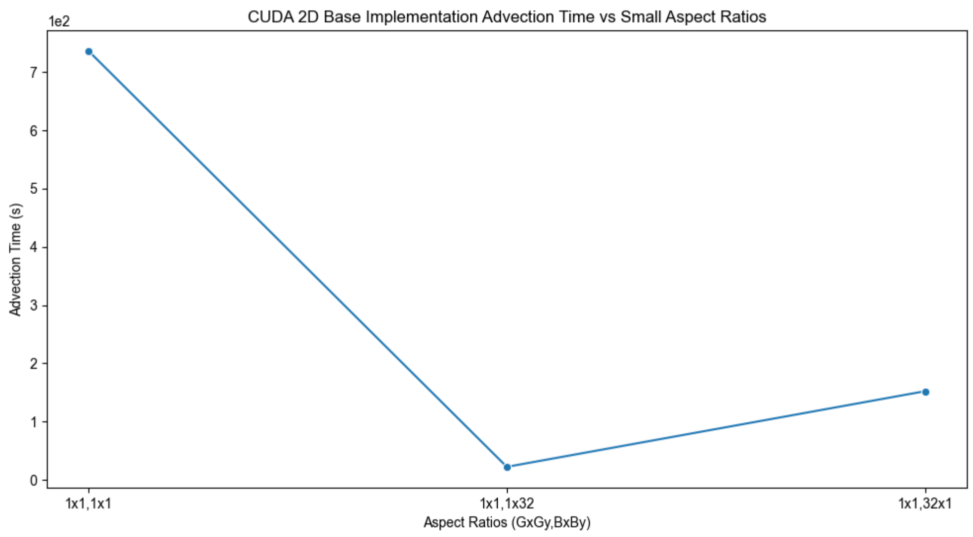

Figure 6: Baseline CUDA implementation, contextualising the small grid/block threading combinations.

Noting the behaviour of the baseline CUDA implementation against the small grid/block sizes, we can see that there is an immediate speedup from the single core execution. However, it is not as significant as one might expect, due to the heavily serial and interdependent nature of the kernels. It is interesting to note the difference in performance between 1×32 and 32×1 block sizes. This can be accounted for in the bias in boundary updates, specifically due to the directional indexing scheme used. The second kernel invocation for the EW basis is more efficient overall when using the 1×32 variant, slightly outperforming the other configuration.

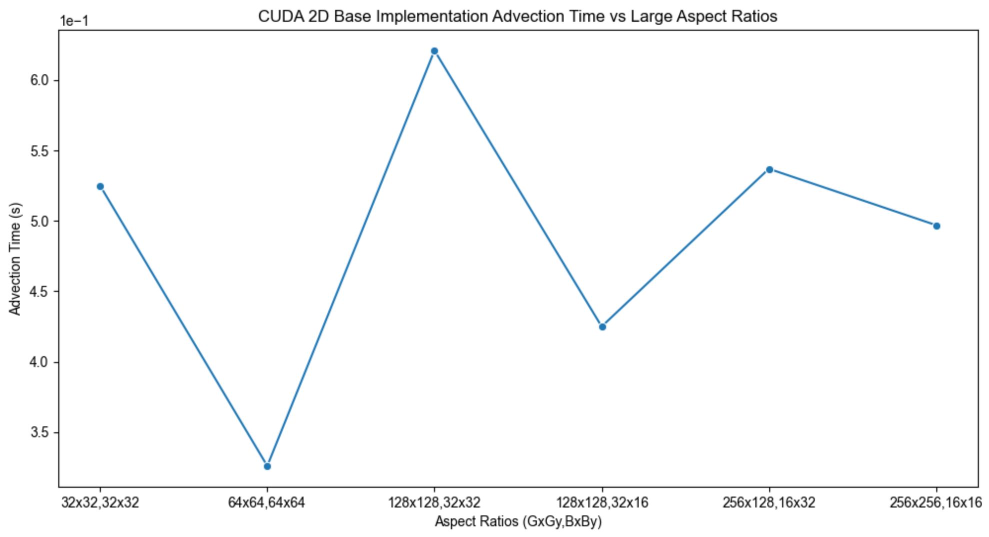

Figure 7: Baseline CUDA implementation, detailing the larger grid/block combinations.

Aspect Ratio	**Advection Time**
G: $(32, 32)$ B: $(32, 32)$	5.25×10^{-1}s
G: $(64, 64)$ B: $(32, 32)$	3.26×10^{-1}s
G: $(128, 128)$ B: $(32, 32)$	6.21×10^{-1}s
G: $(128, 128)$ B: $(32, 16)$	4.25×10^{-1}s
G: $(256, 128)$ B: $(16, 32)$	5.37×10^{-1}s
G: $(256, 256)$ B: $(16, 16)$	4.97×10^{-1}s

Table 6: Raw values for advection times at various aspect ratios for grid and block configurations.

Using larger thread decomposition for the solver, the performance gains are more noticeable. Though it is clear that there is an asymptote of some kind around the 4.7×10^{-1}s mark, as any variance in the grid and block configurations seem to level at this point. Given the nature of the implementation as previously mentioned, this is a bottleneck as there is not enough of the problem parallelised to fully utilise GPU resources efficiently.

Note that the implementation seems to achieve better results with the second and fourth configuration. Given the threading allocations, this would imply that matching 1:1 in terms of elements to threads does not yield effective performance gains. This is mostly likely due to a lack of throughput despite effective occupancy (given the thread values and lack of conditionals to cause divergence).

5.2 Kernel Invocation Overhead

Kernel invocations come in two flavours, depending on how they are used. On one hand they may have synchronisation barriers before ad after and on the other hand, no synchronisation barriers. Both of these exhibit different behaviours. To examine them, we will invoke a series of empty kernel launches for a single grid/block/thread and measure both as an average over the iterations.

```
#define R 10
#define N 1000

__global__ void emptyKernel() {}

double avg_sync = 0.0;
for (int r = 0; r < R; r++) {
    cudaDeviceSynchronize();
    double start_sync = gettime();
    for i in 0..N {
        emptyKernel<<<1,1>>>();
    }
    cudaDeviceSynchronize();
    avg_sync += gettime() - start_sync;
}
double avg_no_sync = 0.0;
for (int r = 0; r < R; r++) {
    cudaDeviceSynchronize();
    double start_no_sync = gettime();
    for i in 0..N {
        emptyKernel<<<1,1>>>();
    }
    avg_no_sync += gettime() - start_no_sync;
}
print(avg_sync, avg_no_sync);
```

Per-Kernel Sync	**No Sync**
3.128223×10^{-6}s	7.989457×10^{-6}s

After measuring the overhead, it seems that synchronised launches incur and addition overhead, being 155.4% more expensive than their non-synchronised counterparts. This overhead will likely have little to no effect on the vast majority of programs being executed. However, in the case that a program utilises many kernel invocations these overheads could become present with a large enough timescale.

Note: There is a post (dscerutti et al. 2022) that is floating around on the NVIDIA forums that talks about these two kinds of overhead characterisations with essentially the same implementation. The idea for computing the overhead of syncrhonised kernel launches was inspired by this article.

6 Question 6

In order to accelerate the baseline implementation, the advection solver problem decomposition needs to fit to the GPU architecture. The first section is to concurrently execute two kernels via CUDA streams for the boundary update and inner field update. This interleaves the execution of warps between both kernels allowing for overlapped execution, provide resource utilisation allows for both kernels to execute simultaneously. A synchronising call to `cudaDeviceSynchronise()` ensures a barrier is held before the next iteration proceeds.

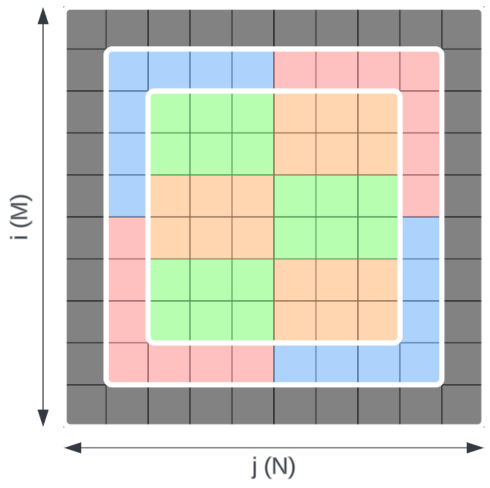

Figure 8: Example grid/block distribution over an advection grid. Boundary updates are denoted as red and blue blocks. Inner field updates are denoted by green and orange blocks. Dark outline is the halo exchange extension to the base advection grid.

The next optimisation to consider is the use of arithmetic based conditional logic using modular-conditionals. Further detail on this can be found in the source `parAdvect.cu` with a large block comment.Another optimisation prevents requiring fully copies between the output and input buffer at each iteration. Instead, a device pointer swap is used to exchange the pointers of `u` and `v` to swap the targets for each buffer base address.

Lastly, the working chunk from the global input field is copied into shared memory for each block. This is a 2D chunk copied by each thread, responsible for its own 2D block inside this chunk (via `cudaMemcpy2D(...)`). This could be further improved to alter how the grid/block/thread indexing is used to ensure that each block maps to a contiguous section of the input buffer. (Due to time constraints I did not get around to refactoring to support this completely, see `updateInnerFieldDeviceSharedMem` in `parAdvect.cu`).

6.1 Performance Evaluation

For consistency, we will use the same parameterisation for evaluating the optimised variant of the CUDA advection solution.

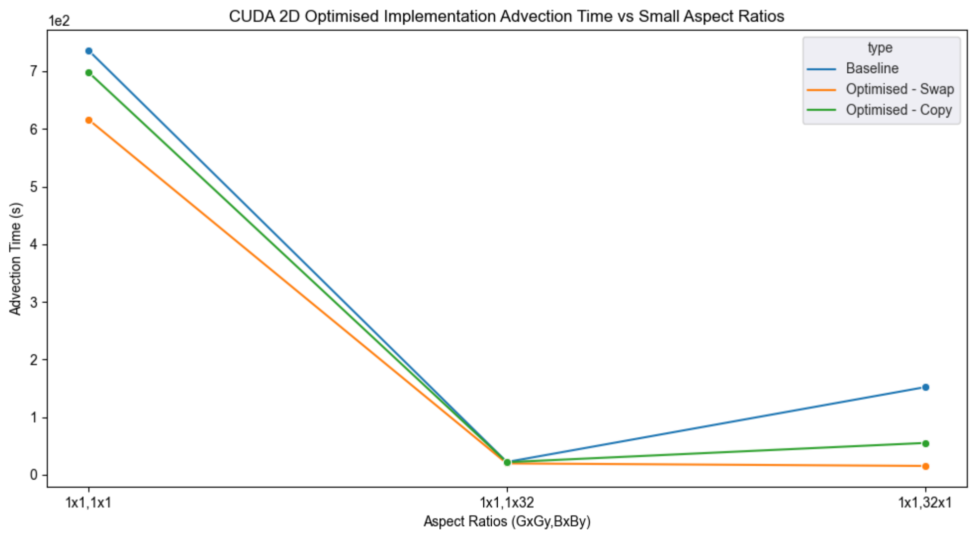

Figure 9: Optimised CUDA implementation, contextualising the small grid/block threading combinations.

Comparing the small grid/block configurations against that of the baseline, we see a definitive improvement over the baseline in all but the $1 \times 1, 1 \times 32$ configuration, where the performance is remarkably similar. This could be chalked up to the limited improvements that parallelism yields for such large matrices with small parallelisation factor. Additionally, we see that the behaviour of the $1 \times 1, 32 \times 1$ configuration has changed to being the better performing in this case. This is attributable to the uniform parallelisation of the entire boundary as a single kernel with modular-conditionals.

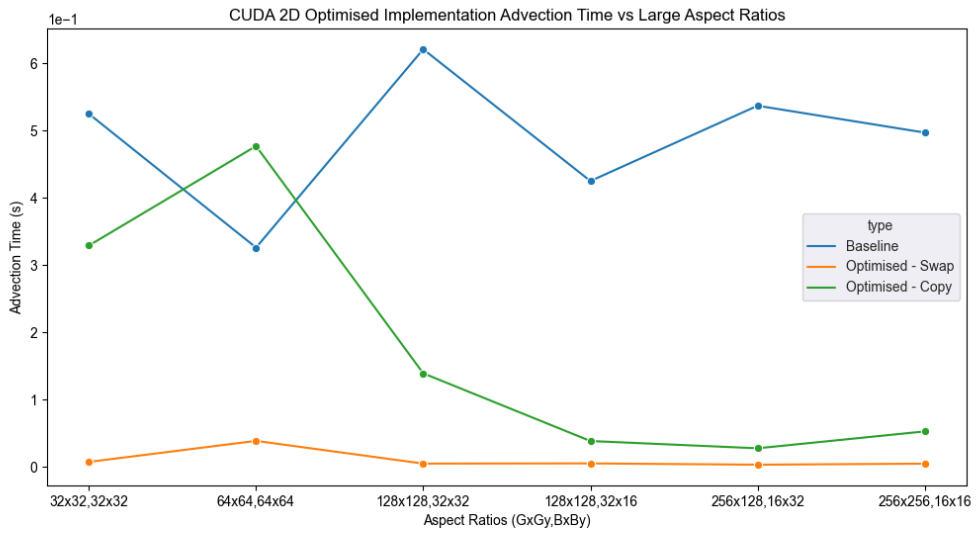

Figure 10: Optimised CUDA implementation, detailing the larger grid/block combinations.

Aspect Ratio / Advection Time	**Baseline**	**Optimised - Copy**	**Optimised - Swap**
G: $(32, 32)$ B: $(32, 32)$	7.22×10^{-3}s	3.29×10^{-1}s	7.22×10^{-3}s
G: $(64, 64)$ B: $(32, 32)$	3.85×10^{-2}s	4.77×10^{-1}s	3.85×10^{-2}s
G: $(128, 128)$ B: $(32, 32)$	4.72×10^{-3}s	1.39×10^{-1}s	4.72×10^{-3}s
G: $(128, 128)$ B: $(32, 16)$	5.01×10^{-3}s	3.83×10^{-2}s	5.01×10^{-3}s
G: $(256, 128)$ B: $(16, 32)$	3.08×10^{-3}s	2.75×10^{-2}s	3.08×10^{-3}s
G: $(256, 256)$ B: $(16, 16)$	4.64×10^{-3}s	5.26×10^{-2}s	4.64×10^{-3}s

Table 7: Raw values for advection times at various aspect ratios for grid and block configurations.

Using larger configurations, the performance improvement is clear across all cases. Optimising memory usage via shared memory at a block level, in conjunction with the alleviation of warp divergence cases that may be present has provided a substantial boost in performance. A major change to the structure of the algorithm was the removal of the copy from the output buffer to the input buffer, in favour of a pointer swap. This was a significant time sink, up until the

$128 \times 128, 32 \times 16$ configuration where the performance benefits taper off. From this point on, it's clear that the other optimisations implemented assisted in reducing the advection time.

7 Question 7

Between the three models of MPI, OpenMP and CUDA, design wise CUDA provides the greatest diversity for design of highly parallel tasks. More specifically, with regards to problems that have thread independent discretisation models, such as matrix evolutions (Heat equation, advection, etc). However, when it comes to approaches that require interaction between threads (not necessarily synchronisation), it introduces complexity that is otherwise more manageable in the MPI model. As for OpenMP, it yields similar benefits that CUDA does, however in much more significant context given the restrictions on compiler driven threading models as opposed to the more explicit nature of CUDA.

Implementation wise, OpenMP has the most approachable interfaces for both new and adaptive implementation. This extends to understanding the behaviour of how execution, compaction and grouping function relative to the behaviour of the pragma directives. That being said, it has a high-end implementation overhead when it comes to optimising it for fine-grained control of execution behaviour due to the high-level of abstraction. MPI has provides a well rounded SPI from simplistic interaction models for control over exchange behaviour and optimisations to a per-thread level to the point of fine-grained memory behaviour.

Over CUDA, it is substantially easier to control complex overlapping behaviour where the implementation does not follow SIMD/SIMT structuring well (diversity in condition evaluation for instance). This allows for reduced algorithmic complexity when handling these kinds of diverse behaviour on a thread-to-thread basis. Lastly, for CUDA, implementation has a high-entry barrier in terms of understanding how to appropriately structure your implementation and understand the resource decomposition required for appropriate allocation and usage. This requires more work to structure the implementation differently, especially when considering how conditional overheads (warp divergence/convergence) are easy to introduce with easy-to-overlook implementation aspects. Substantially more effort and understanding is required for the CUDA model, however, even with this overhead the tool set it provides for programmability is far and above the others allowing full control over the behaviour and program structure across threads when compared to OpenMP.

During development, in any of the three paradigms/models, the debugging of the core algorithm with regards to structure, is essentially the same. This is specific to the arithmetic operations and memory operations. However, considering the halo exchanges and the different approaches for managing these, it differs quite a bit ween the communication model versus the shared memory model. For MPI, exchange debugging relies on logging tagged entries communicated between threads for message issues. With contention issues much of the debugging hinges on manual checking and verification of parameters for communication handlers.

On the other hand, in the land of shared memory models, OpenMP is straightforward to debug for most cases. In the most difficult of cases (that I had encountered), it comes down to data ownership between threads and the global program space. Missing a marker for shared or private variables can lead to undefined behaviour that is hard to track down, however, due to the simplicity of compiler directives for simple cases (such as advection solution), these tend to be easy to find. Compare that to CUDA, which has improvement with regards to scoping as mentioned previously, as runtime errors occur for segmentation violations and improper host/device pointer usage. These become much clearer, however, sourcing the problem behind them can be arduous for complex logic. With proper alteration of kernel execution and isolation, it does however provide an edge over OpenMP with the clarity of debugging techniques.

References

Ahmad, Z., Chowdhury, R., Das, R., Ganapathi, P., Gregory, A. & Zhu, Y. (2021), 'Fast stencil computations using fast fourier transforms'.

dscerutti, njuffa, Greg & Crovella, R. (2022), 'Any way to measure the latency of a kernel launch?'.
URL: *https://forums.developer.nvidia.com/t/any-way-to-measure-the-latency-of-a-kernel-launch/221413/4*