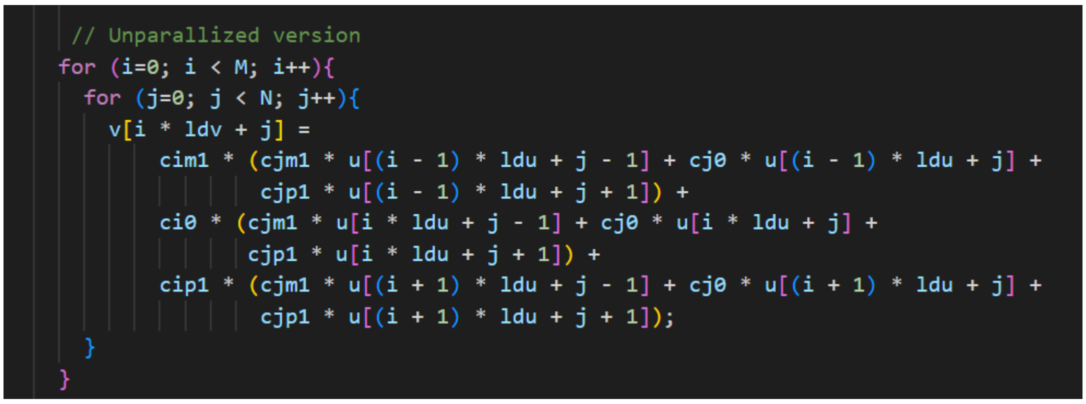
Part 1: OpenMP

Q1: Parallelization via 1D Decomposition and Simple Directives

Below is the code which needs to be parallelized(from omp_update_advection_field_1D_decomposition).



```
// Unparallized version
for (i=0; i < M; i++){
  for (j=0; j < N; j++){
    v[i * ldv + j] =
        cim1 * (cjm1 * u[(i - 1) * ldu + j - 1] + cj0 * u[(i - 1) * ldu + j] +
                cjp1 * u[(i - 1) * ldu + j + 1]) +
        ci0 * (cjm1 * u[i * ldu + j - 1] + cj0 * u[i * ldu + j] +
               cjp1 * u[i * ldu + j + 1]) +
        cip1 * (cjm1 * u[(i + 1) * ldu + j - 1] + cj0 * u[(i + 1) * ldu + j] +
                cjp1 * u[(i + 1) * ldu + j + 1]);
  }
}
```

Parallelize outer/inner loop

If we parallelize the inner loop, then each time i is incremented, we need to create parallel regions, after the parallel computation completes. The parallel regions exit and the loop moves to next iteration by incrementing i and creates another parallel region. This process repeats until i == M-1. We can see this is still an sequential program. If we parallel outer loop, then a set of parallel regions is created for each subset of i, once the parallel regions exit, the

Interchange loop order

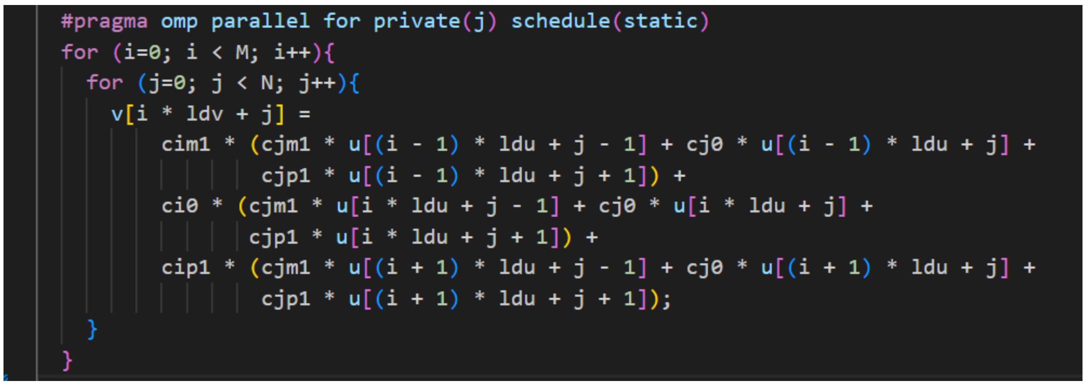
The 2D array is stored in row-major order in memory, this means that when we load v[i * ldv + j], the elements which immediately follow v[i * ldv + j], i.e., v[i * ldv + j + 1], v[i * ldv + j + 2] etc., will also be loaded and stored in cache. In the next iteration we can access v[i * ldv + j + 1] in the cache rather than memory. Hence, if we use j-i loop rather than i-j loop, cache miss rate will be high since v[(i + 1) * ldv + j + 1] is not in cache.

Schedule iterations in block or cyclic fashion

Block scheduling is a static scheduling policy where each thread is assigned a fixed number of work. Cyclic is a dynamic scheduling policy where each thread gets allocated a small piece of work initially, once a thread finishes its assigned work, the scheduler assigns it with another piece of computation work. This is best suitable when the number of computation of each thread is not known in advance. In such case, threads with "cheap work" will keep receiving work from scheduler instead of being idle. In our case, each thread needs to perform some fixed number of arithmatic operations and hence workload is known in advance. If we use cyclic scheduling, then the scheduling overhead could slow down the performance. Hence, block fashion is preferred in terms of performance.

Maximize Performance

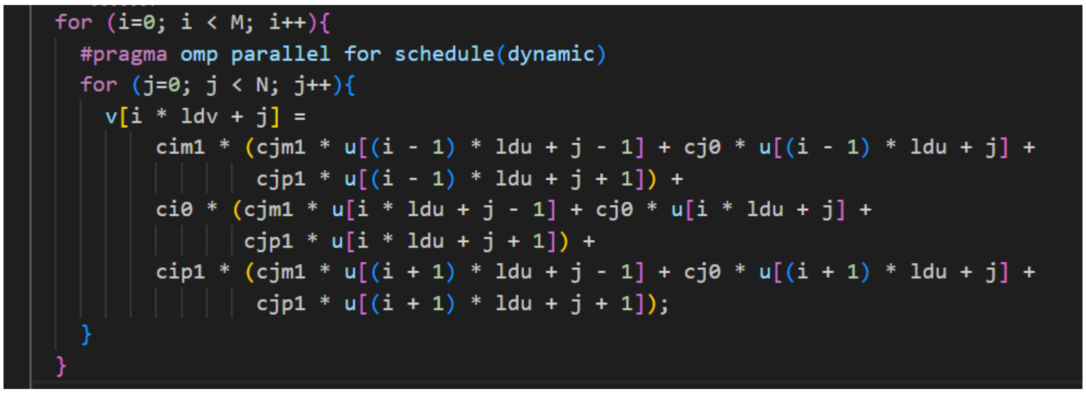
Given the discussion above, the combination which maximized performance is:



```
#pragma omp parallel for private(j) schedule(static)
for (i=0; i < M; i++){
  for (j=0; j < N; j++){
    v[i * ldv + j] =
        cim1 * (cjm1 * u[(i - 1) * ldu + j - 1] + cj0 * u[(i - 1) * ldu + j] +
                cjp1 * u[(i - 1) * ldu + j + 1]) +
        ci0 * (cjm1 * u[i * ldu + j - 1] + cj0 * u[i * ldu + j] +
               cjp1 * u[i * ldu + j + 1]) +
        cip1 * (cjm1 * u[(i + 1) * ldu + j - 1] + cj0 * u[(i + 1) * ldu + j] +
                cjp1 * u[(i + 1) * ldu + j + 1]);
  }
}
```

Maximize the number of OpenMP parallel region entry/exits

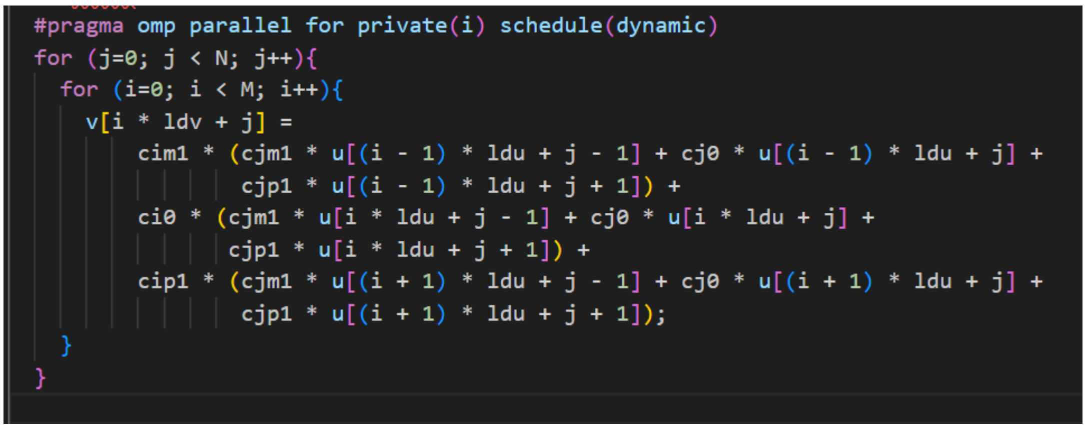
Any combinations which parallelize inner loop will maximize number of OpenMP parallel region entry/exits. Below is one example.



```
for (i=0; i < M; i++){
  #pragma omp parallel for schedule(dynamic)
  for (j=0; j < N; j++){
    v[i * ldv + j] =
        cim1 * (cjm1 * u[(i - 1) * ldu + j - 1] + cj0 * u[(i - 1) * ldu + j] +
                cjp1 * u[(i - 1) * ldu + j + 1]) +
        ci0 * (cjm1 * u[i * ldu + j - 1] + cj0 * u[i * ldu + j] +
               cjp1 * u[i * ldu + j + 1]) +
        cip1 * (cjm1 * u[(i + 1) * ldu + j - 1] + cj0 * u[(i + 1) * ldu + j] +
                cjp1 * u[(i + 1) * ldu + j + 1]);
  }
}
```

Maximize cache misses involving read operations

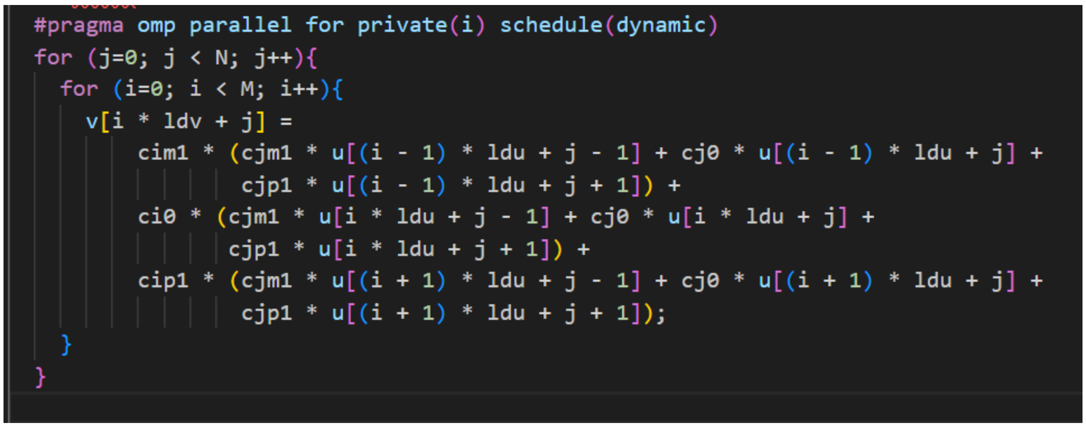
Combinations which perform j-i loop maximize cache misses. Below is one example.



```c
#pragma omp parallel for private(i) schedule(dynamic)
for (j=0; j < N; j++){
  for (i=0; i < M; i++){
    v[i * ldv + j] =
        cim1 * (cjm1 * u[(i - 1) * ldu + j - 1] + cj0 * u[(i - 1) * ldu + j] +
                cjp1 * u[(i - 1) * ldu + j + 1]) +
        ci0 * (cjm1 * u[i * ldu + j - 1] + cj0 * u[i * ldu + j] +
               cjp1 * u[i * ldu + j + 1]) +
        cip1 * (cjm1 * u[(i + 1) * ldu + j - 1] + cj0 * u[(i + 1) * ldu + j] +
                cjp1 * u[(i + 1) * ldu + j + 1]);
  }
}
```

Maximize cache misses involving write operations

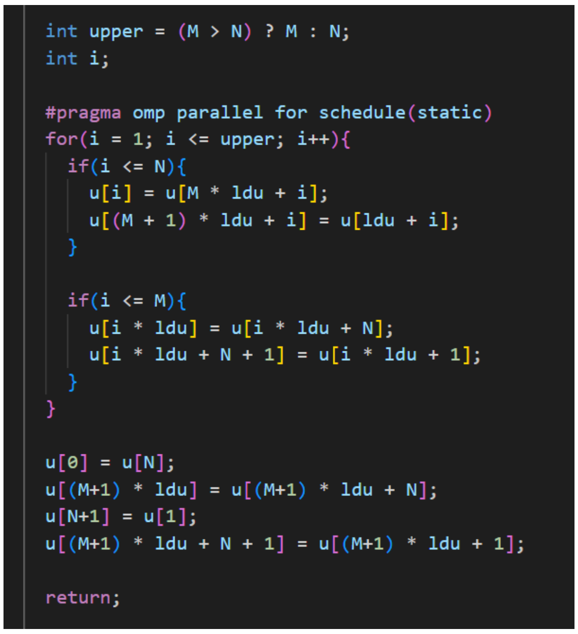
Combinations which perform j-i loop maximize cache misses. Below is one example.



```c
#pragma omp parallel for private(i) schedule(dynamic)
for (j=0; j < N; j++){
  for (i=0; i < M; i++){
    v[i * ldv + j] =
        cim1 * (cjm1 * u[(i - 1) * ldu + j - 1] + cj0 * u[(i - 1) * ldu + j] +
                cjp1 * u[(i - 1) * ldu + j + 1]) +
        ci0 * (cjm1 * u[i * ldu + j - 1] + cj0 * u[i * ldu + j] +
               cjp1 * u[i * ldu + j + 1]) +
        cip1 * (cjm1 * u[(i + 1) * ldu + j - 1] + cj0 * u[(i + 1) * ldu + j] +
                cjp1 * u[(i + 1) * ldu + j + 1]);
  }
}
```

Parallelization startegy for omp_update_boundary_1D_decomposition()

Below is the body of my omp_update_boundary_1D_decomposition() function.



```c
int upper = (M > N) ? M : N;
int i;

#pragma omp parallel for schedule(static)
for(i = 1; i <= upper; i++){
  if(i <= N){
    u[i] = u[M * ldu + i];
    u[(M + 1) * ldu + i] = u[ldu + i];
  }

  if(i <= M){
    u[i * ldu] = u[i * ldu + N];
    u[i * ldu + N + 1] = u[i * ldu + 1];
  }
}

u[0] = u[N];
u[(M+1) * ldu] = u[(M+1) * ldu + N];
u[N+1] = u[1];
u[(M+1) * ldu + N + 1] = u[(M+1) * ldu + 1];

return;
```

We parallelize it using a single loop to update the top/bottom/right/left halo area, excluding the 4 corner pointes. At the end of the function we updated these 4 corner points.

Experiments:

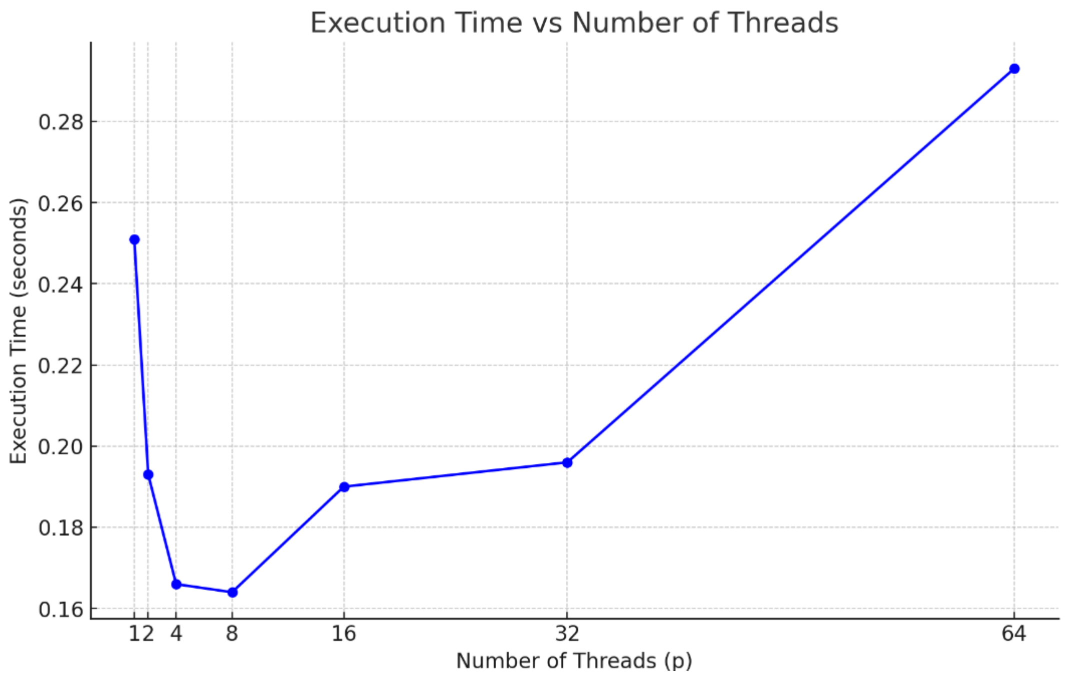
Effect of boundary parallelization

We run experiments with (M,N) = (1024, 1024), (2048,2048),(4096,4096), (8192,8192), r=5， p = 16, with omp_update_advection_field_1D_decomposition() being the max performance one described above.

(M, N)	Execution Unoptimized	Execution Optimized	Speedup
(1024, 1024)	2.41e-02s	1.80e-02s	1.34x
(2048, 2048)	8.54e-02s	6.69e-02s	1.28x
(4096, 4096)	2.89e-01s	2.32e-01s	1.25x
(8192, 8192)	8.71e-01s	8.44e-01s	1.03x

Choosing the best p

M, N = 1024, r = 100, p = [1, 2, 4, 8, 16, 32, 64]



From the above graph, we can observe that the best performing p is 8, which is a bit surprising. I think the reason this happens is as p increases, the overhead of data distribution and thread launch/exits become unneglectable compared to computation time.

Q2: Parallelization via 2D Decomposition and an Extended Parallel Region

Performance comparison with 1D parallelization

Here, we perform experiments with M, N = 1024, r = 100, P = [1, 2, 4]. The results is presented in below table. The baseline from the above is 1.64e-01s

P	Execution time	Speedup
1	1.76e-01s	0.93x
2	1.56e-01s	1.05x
4	1.35e-01s	1.21x
8	2.01e-01s	0.82x

Performance gain

We can observe that the performance is at best when p=4, I think this is due to the fact that load balancing is best achieved with p=4.

Comparison with 1D vs 2D in MPI

Similarities:

- In both MPI and OpenMP, performance can be gained by reduced workload due to 2D parallelization.

Differences:

- OpenMP threads share the same address space whereas MPI relies on message passing to distribute data, which might increase time spent on communication.
- When M, N are large, MPI can also gain performance from 2D parallelization due to smaller message size and such performance gain is not possible for OpenMP.
- Thread synchronisation needs to be done in OpenMP and the cost of maintaining cache coherence increases when threads share data.

Part 2: CUDA

Q5: Baseline GPU Implementation

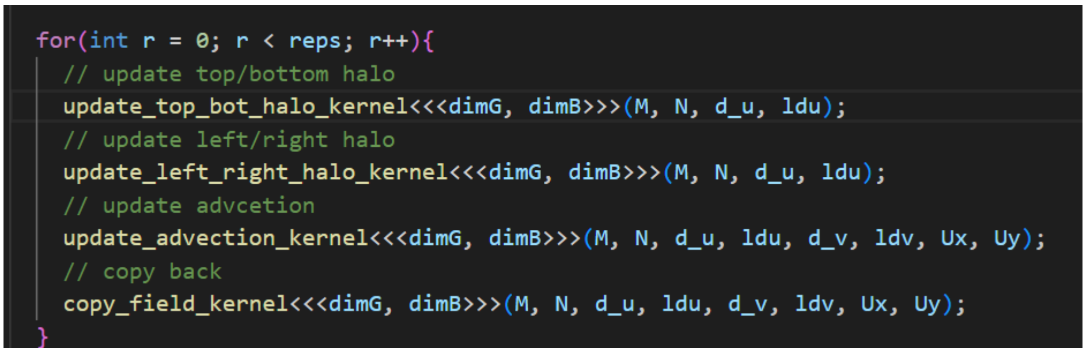
Implementation

We view the grid as a 2D thread array where there are Gx * Bx threads in x-dimention and Gy * By threads in y-dimension. For each thread, we calculate its thread index by:

$$(Idx, Idy) = (blockIdx.x * Bx + threadIdx.x, blockIdx.y * By + threadIdx.y),$$

then based on its thread index, each thread is allocated a submatrix of the orignal matrix that it needs to work on.

Below is the main part of my implementation.



```
for(int r = 0; r < reps; r++){
  // update top/bottom halo
  update_top_bot_halo_kernel<<<dimG, dimB>>>(M, N, d_u, ldu);
  // update left/right halo
  update_left_right_halo_kernel<<<dimG, dimB>>>(M, N, d_u, ldu);
  // update advcetion
  update_advection_kernel<<<dimG, dimB>>>(M, N, d_u, ldu, d_v, ldv, Ux, Uy);
  // copy back
  copy_field_kernel<<<dimG, dimB>>>(M, N, d_u, ldu, d_v, ldv, Ux, Uy);
}
```

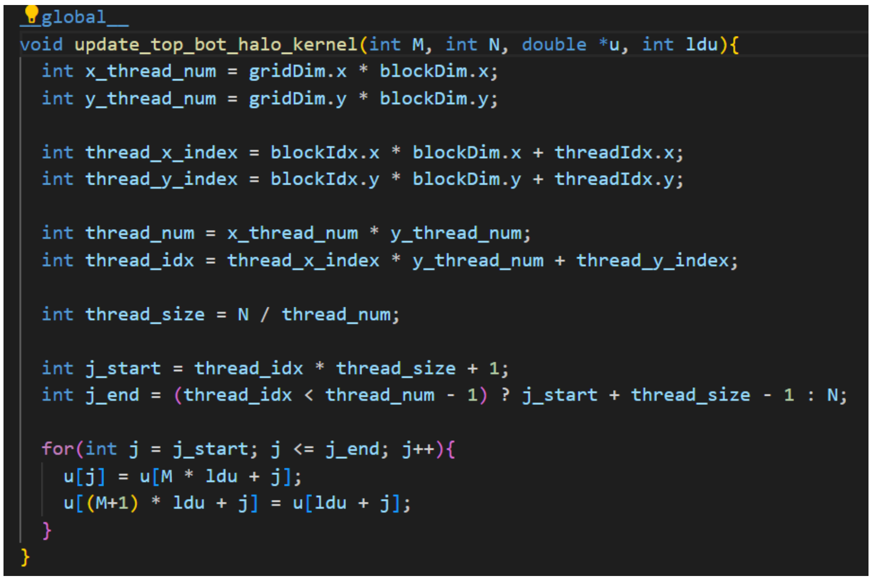
Parallelization strategy for boundary update

We write 2 kernel functions update_top_bot_halo_kernel(), update_left_right_halo_kernel() for the top/bottom halo update and left/right halo update, respectively.

For update_top_bot_halo_kernel() function, we first flatten each thread by calculating its new index value:

$$index = Idx * (Gy * By) + Idy$$

Then, based on its 1D index, each thread is allocated a subarray of the top/bottom halo array that it needs to work on. Below is the code snippet.

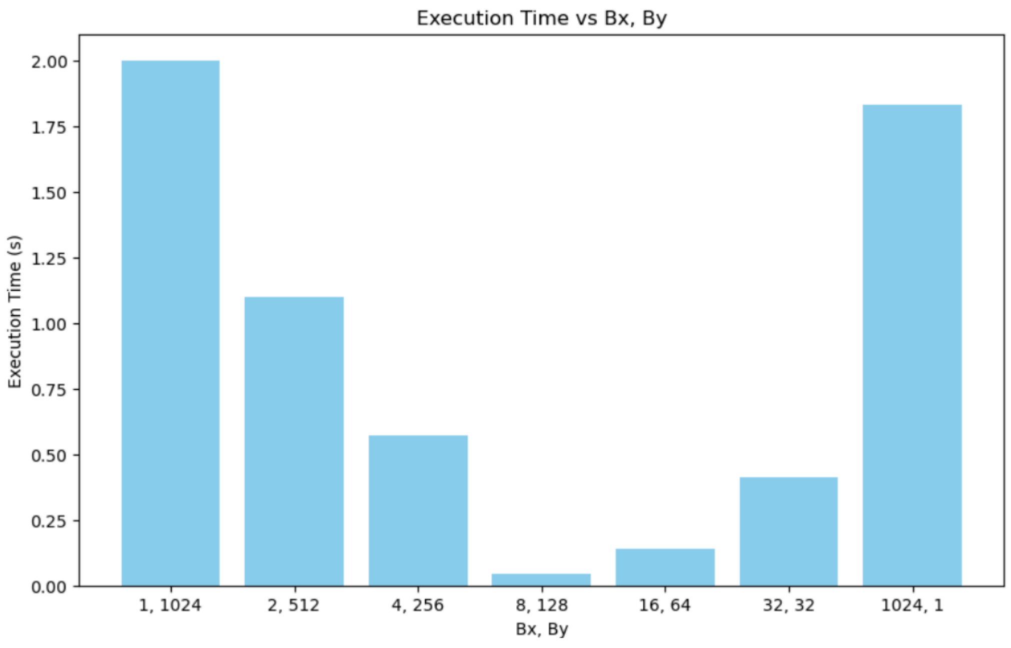


```
__global__
void update_top_bot_halo_kernel(int M, int N, double *u, int ldu){
  int x_thread_num = gridDim.x * blockDim.x;
  int y_thread_num = gridDim.y * blockDim.y;

  int thread_x_index = blockIdx.x * blockDim.x + threadIdx.x;
  int thread_y_index = blockIdx.y * blockDim.y + threadIdx.y;

  int thread_num = x_thread_num * y_thread_num;
  int thread_idx = thread_x_index * y_thread_num + thread_y_index;

  int thread_size = N / thread_num;

  int j_start = thread_idx * thread_size + 1;
  int j_end = (thread_idx < thread_num - 1) ? j_start + thread_size - 1 : N;

  for(int j = j_start; j <= j_end; j++){
    u[j] = u[M * ldu + j];
    u[(M+1) * ldu + j] = u[ldu + j];
  }
}
```

update_left_right_halo_kernel() is implemented in the same manner.

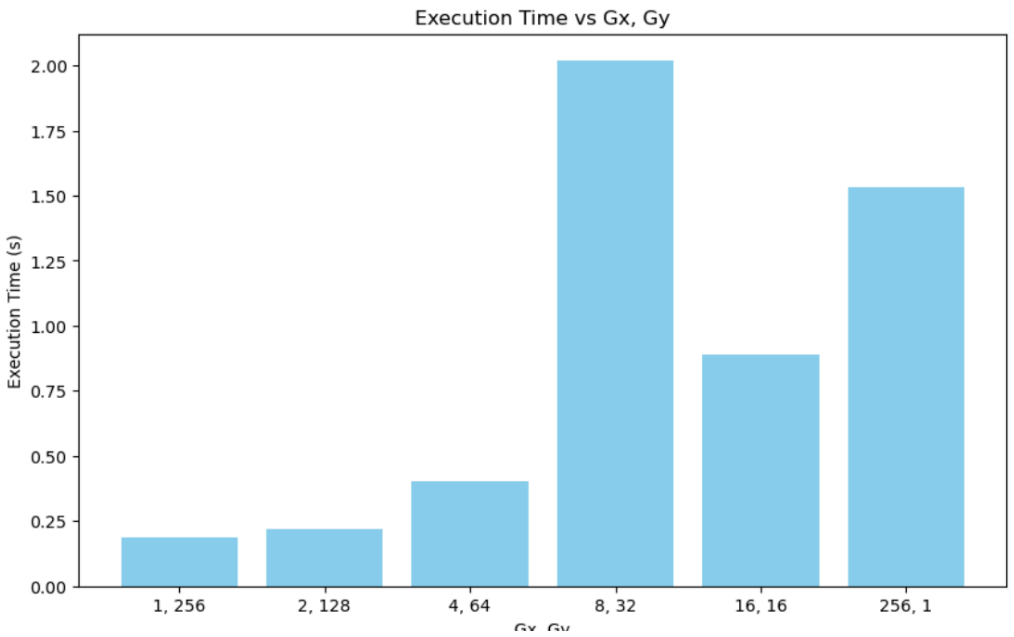
Effects of varying Gx,Gy,Bx,By

- **Effects of Bx, By**: Here, we perform experiments with M, N = 4096, r = 10, Gx, Gy = 32, with Bx, By = (1, 1024), (2, 512), (4, 256), (8, 128), (16, 64), (32, 32), (1024, 1).



I am surprised to find out that performance peaked at Bx,By level of (8,128). I expected the peak level to be (32,32). I suspect this is due to the specific architecture of studentgpu2.

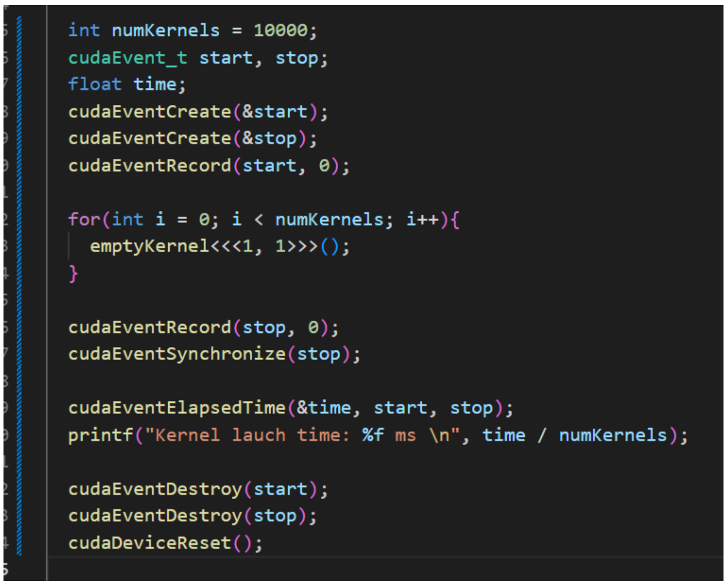
- **Effects of Gx, Gy**: Here, we perform experiments with M, N = 4096, r = 10, Bx, By = 32, with Gx, Gy= (1, 256), (2, 128), (4, 64), (8, 32), (16, 16), (32, 8), (64, 4), (128, 2), (256, 1)



It is strange to see that a Gx,Gy of 1 * 256 outperforms all other Gx, Gy combinations. What I expected is performance peaked at (Gx,Gy) = (32, 32). I suspect this is due to the specific architecture of studentgpu2.

Kernel invocation overhead

Below is the code I use to measure kernel overhead. My measured result is: 0.003134 ms.



```
int numKernels = 10000;
cudaEvent_t start, stop;
float time;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start, 0);

for(int i = 0; i < numKernels; i++){
  emptyKernel<<<1, 1>>>();
}

cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);

cudaEventElapsedTime(&time, start, stop);
printf("Kernel lauch time: %f ms \n", time / numKernels);

cudaEventDestroy(start);
cudaEventDestroy(stop);
cudaDeviceReset();
```

Speedups against the single GPU & host

Here, I only perform measurements with M,N = (2000, 2000), (4000, 4000), (8000, 8000), r = 5, Gx,Gy Bx, By = 20;

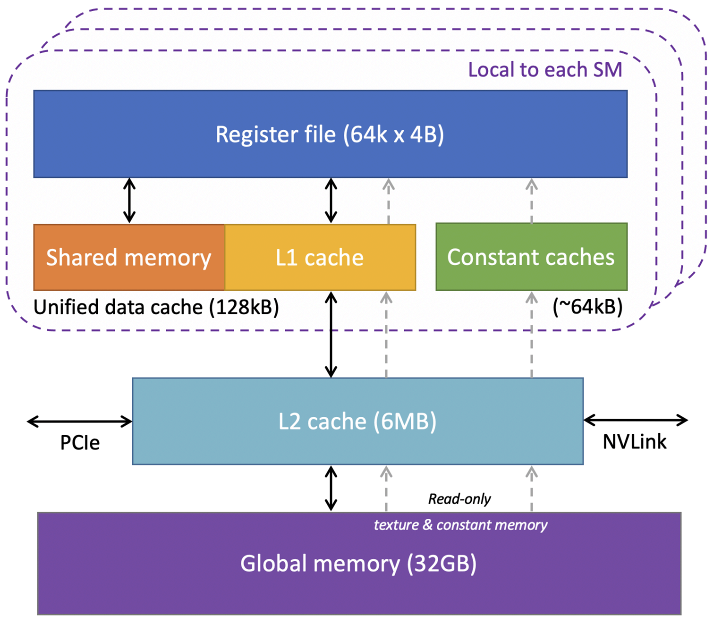
Below is the table.

(M, N)	Host	Single GPU	Multiple GPUs	Speedup vs Single GPU	Speedup vs Host
(2000,2000)	3.41e-01s	8.37e+00s	1.78e-02s	470.2x	19.1x
(4000,4000)	6.21e-01s	3.44e+01s	1.17e-01s	294.0x	5.3x
(8000,8000)	1.99e+00s	1.41e+02s	6.21e-01s	227.0x	3.2x

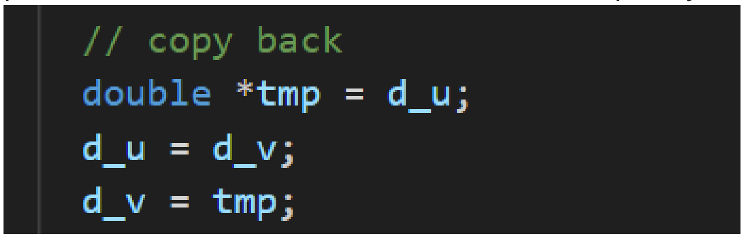
Q6: Optimized GPU Implementation

Optimization strategy:

- **Shared memory optimization**: Below is a picture of GPU memory hierachy(source: https://cvw.cac.cornell.edu/gpu-architecture/gpu-memory/memory_levels). Accessing data from global memory is expensive and can be hundreds of time slower than accessing each block's shared memory. In the baseline implementation, all the data accessing is with the global memory. Therefore, we optimize it by doing shared memory accessing. Each thread in the block first load all the date it needs from the global memory to the shared memory within the block. Then, when the thread needs to perform advection update, it loads the needed data from the shared memory within the block rather than the global memory.



- **Copy field optmization**: The copy back functionality in our baseline implementation is achieved by copy_filed_kernel() kernel function, which basically every element of v to u, and hence has O(M * N) time complexity where M, N are size of each block. This can be optimized by a simple pointer swich below, which has O(1) time complexity.



```
// copy back
double *tmp = d_u;
d_u = d_v;
d_v = tmp;
```

Optimization effects:

To compare performance, we run experiments with M,N = (4000, 4000), (8000, 8000), (16000, 16000), Bx,By = 20.

Below is the table.

(M, N)	(Gx, Gy)	Execution Time (Unoptimized)	Execution Time (Optimized)	Speedup
(4000,4000)	(80, 80)	2.91e-01s	9.46e-03s	30.76x
(8000,8000)	(160,160)	1.15e+00s	2.92e-02s	39.38x
(16000,16000)	(320,320)	4.83e+00s	9.91e-02s	48.73x