Student Name: **Jeremiah Hsing**
Student ID: **u7439274**

COMP 4300 Parallel System

Assignment 2

Part 1: OpenMP

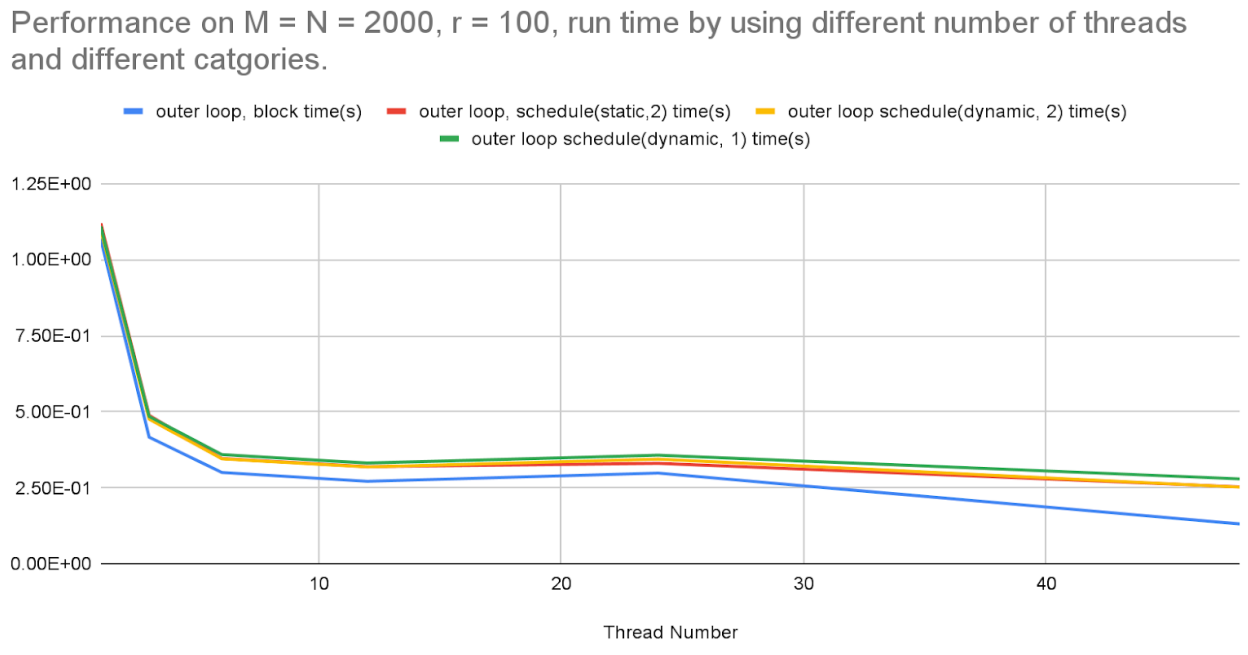
1. Parallelization via 1D Decomposition and Simple Directives

This is a run-time table with different strategies on the function ***omp1dUpdateAdvectField()*** with a different number of threads (M=N=2000, r=100).

Thread Number	1	3	6	12	24	48
outer loop, schedule(static) time(s)	1.07E+00	4.17E-01	3.01E-01	2.72E-01	2.99E-01	1.32E-01
outer loop, schedule(static, 1) time(s)	1.12E+00	4.89E-01	3.47E-01	3.20E-01	3.31E-01	2.54E-01
outer loop, schedule(dynamic, 2) time(s)	1.10E+00	4.76E-01	3.46E-01	3.19E-01	3.45E-01	2.53E-01
outer loop, schedule(dynamic, 1) time(s)	1.11E+00	4.85E-01	3.60E-01	3.32E-01	3.58E-01	2.80E-01
inner loop, schedule(static) time(s)	1.67E+00	1.54E+00	2.72E+00	3.07E+00	1.65E+01	4.11E+00

We can easily see that the inner loop parallel is significantly slower than other categories.

For better visualization, here is a graph of all the outer loop parallel categories.



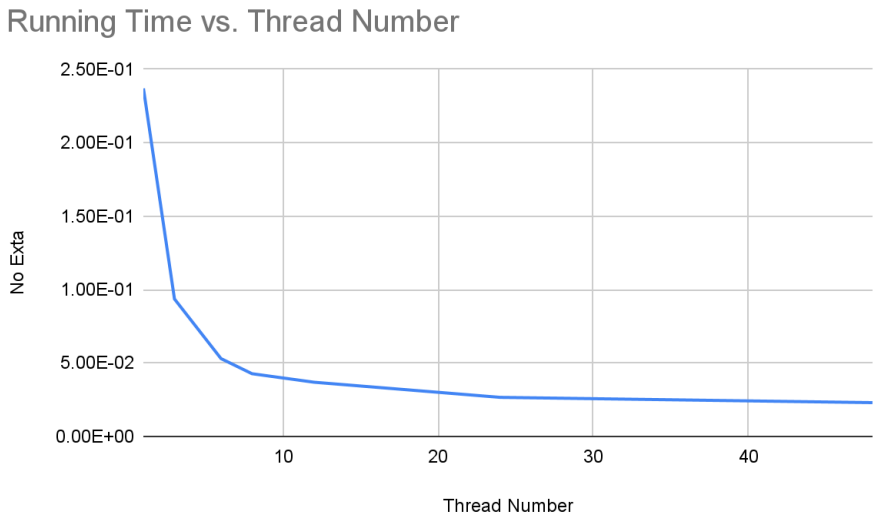
Analysis:

1) According to the graph, we can get that parallel outer for loop using ***schedule(static)*** with the default chunk size (by block fashion) can maximize the performance.

2) A unusual result is that when OMP_NUM_THREADS = 24, the program is slower than 12 threads. One possible reason is that when OMP_NUM_THREADS = 24, the program is also in the constraint of "***numactl --cpunodebind=0 --membind=0***" parameters. These bind the CPU and memory of the program. And when the number of threads is large enough, the context switch between each thread will be influenced by the CPU and memory binding and become slower.

   To verify this, I also conduct an experiment without using any CPU or memory bind (remove "***numactl --cpunodebind=0 --membind=0***") for all threads, and here is the result:

Thread Number	Running Time
1	2.37E-01
3	9.36E-02
6	5.30E-02
8	4.27E-02
12	3.69E-02
24	2.66E-02
48	2.30E-02



   As we can see the running time of the 24-thread program becomes normal.

For ***omp1dUpdateAdvectField()*** according to the cases:

a. maximize performance

   Approaching the max performance includes 2 main parts:

1) Reduce the thread spawning overhead by reducing the time of parallel region entry/exit. We can do this by modifying the ***omp1dAdvect()*** function, we can allocate the parallel region outside the ***for(r…)*** loop (wrapping ***for(r…)*** loop in ***#pragma omp parallel default(none) private(r) shared(u, ldu, v, ldv, reps){}***).

   This will make sure we only allocate the threads once when we run the program.

2) For parallelizing the ***omp1dUpdateAdvectField()*** for loop, we can parallelize the outer for loop with ***schedule(static)*** directive. The schedule(static) will use the default chunk size which is loop_count/number_of_threads (dreamcrash, 2020). This default chunk size is also verified by myself.

   In this way, we can read and write data as much memory-continuously as we can to maximize the cache hit in each thread.

Here are the code implementations:

```
void omp1dAdvect(int reps, double *u, int ldu) {
 int r, ldv = N+2;
 double *v = calloc(ldv*(M+2), sizeof(double)); assert(v != NULL);
 #pragma omp parallel default(none) private(r) shared(u, ldu, v, ldv, reps)
 {
   for (r = 0; r < reps; r++) {
     omp1dUpdateBoundary(u, ldu);
     omp1dUpdateAdvectField(&V(u,1,1), ldu, &V(v,1,1), ldv);
     omp1dCopyField(&V(v,1,1), ldv, &V(u,1,1), ldu);
   } //for (r...)
 }
 free(v);
} //omp1dAdvect()
```

omp1dAdvect()

```
static void omp1dUpdateAdvectField(double *u, int ldu, double *v, int ldv) {
 int i, j;
 double Ux = Velx * dt / deltax, Uy = Vely * dt / deltay;
 double cim1, ci0, cip1, cjm1, cj0, cjp1;
 N2Coeff(Ux, &cim1, &ci0, &cip1); N2Coeff(Uy, &cjm1, &cj0, &cjp1);
 #pragma omp for private(j) schedule(static)
   for (i=0; i < M; i++)
     for (j=0; j < N; j++)
       V(v,i,j) =
         cim1*(cjm1*V(u,i-1,j-1) + cj0*V(u,i-1,j) + cjp1*V(u,i-1,j+1)) +
         ci0 *(cjm1*V(u,i  ,j-1) + cj0*V(u,i,  j) + cjp1*V(u,i,  j+1)) +
         cip1*(cjm1*V(u,i+1,j-1) + cj0*V(u,i+1,j) + cjp1*V(u,i+1,j+1));
} //omp1dUpdateAdvectField()
```

omp1dUpdateAdvectField()

b. maximize the number of OpenMP parallel region entry/exit (without significantly increasing coherent reads/writes).

To approach this:

1) First we cannot allocate a parallel region outside the ***for(r…)*** loop in ***omp1dAdvect()*** function, because otherwise, it will reduce the time of parallel region entry/exit to 1 when we run the program.

2) We cannot significantly increase coherent reads/writes. So in the ***omp1dUpdateAdvectField()*** function, we still parallelize the outer for loop. This will make sure the read/write operations in the for loop are as much memory-continuous as possible to maximize the cache hit in each thread.

Here are the code implementations:

```
void omp1dAdvect(int reps, double *u, int ldu) {
 int r, ldv = N+2;
 double *v = calloc(ldv*(M+2), sizeof(double)); assert(v != NULL);
 for (r = 0; r < reps; r++) {
     omp1dUpdateBoundary(u, ldu);
     omp1dUpdateAdvectField(&V(u,1,1), ldu, &V(v,1,1), ldv);
     omp1dCopyField(&V(v,1,1), ldv, &V(u,1,1), ldu);
   } //for (r...)
  free(v);
} //omp1dAdvect()
```

omp1dAdvect()

```
static void omp1dUpdateAdvectField(double *u, int ldu, double *v, int ldv) {
 int i, j;
 double Ux = Velx * dt / deltax, Uy = Vely * dt / deltay;
 double cim1, ci0, cip1, cjm1, cj0, cjp1;
 N2Coeff(Ux, &cim1, &ci0, &cip1); N2Coeff(Uy, &cjm1, &cj0, &cjp1);

 #pragma omp parallel for schedule(static)
   for (i=0; i < M; i++)
     for (j=0; j < N; j++)
       V(v,i,j) =
         cim1*(cjm1*V(u,i-1,j-1) + cj0*V(u,i-1,j) + cjp1*V(u,i-1,j+1)) +
         ci0 *(cjm1*V(u,i  ,j-1) + cj0*V(u,i,  j) + cjp1*V(u,i,  j+1)) +
         cip1*(cjm1*V(u,i+1,j-1) + cj0*V(u,i+1,j) + cjp1*V(u,i+1,j+1));
} //omp1dUpdateAdvectField()
```

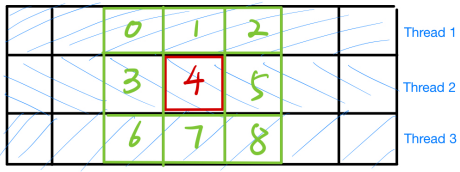
omp1dUpdateAdvectField()

c. maximize cache misses involving coherent reads (without significantly increasing parallel region entry/exits or coherent writes).

To approach this:

1) We cannot significantly increase parallel region entry/exits, so we should allocate the parallel region inside ***omp1dAdvect()*** wrapping the ***for(r…)*** loop.
2) We cannot significantly increase coherent writes, so we need to update(write) the element memory-continuously. However, for each element update, we need to read its adjacent 8 elements and itself, so it's clear that we can maximize the coherent read by assigning each row of elements to a thread (cyclic fashion).

   Here is an example of how it works:



   There are 3 rows of data, and each is assigned to a different thread (***thread 1, 2, 3***). When we need to update element 4, we need to read element [0, 1, 2, 3, 4, 5, 6, 7, 8]. Among those elements we need to read, for ***thread 2***, [0, 1, 2] and [6, 7, 8] are in different threads' L1 cache. So ***thread 2*** will trigger cache miss when reading these elements and increase the coherent read between threads.

   We can use ***schedule(static, 1)*** on the outer for loop in ***omp1dUpdateAdvectField()*** to achieve that.

Here are the code implementations:

```
void omp1dAdvect(int reps, double *u, int ldu) {
 int r, ldv = N+2;
 double *v = calloc(ldv*(M+2), sizeof(double)); assert(v != NULL);
 #pragma omp parallel default(none) private(r) shared(u, ldu, v, ldv, reps)
 {
   for (r = 0; r < reps; r++) {
     omp1dUpdateBoundary(u, ldu);
     omp1dUpdateAdvectField(&V(u,1,1), ldu, &V(v,1,1), ldv);
     omp1dCopyField(&V(v,1,1), ldv, &V(u,1,1), ldu);
   } //for (r...)
 }
 free(v);
} //omp1dAdvect()
```

omp1dAdvect()

```
static void omp1dUpdateAdvectField(double *u, int ldu, double *v, int ldv) {
 int i, j;
 double Ux = Velx * dt / deltax, Uy = Vely * dt / deltay;
 double cim1, ci0, cip1, cjm1, cj0, cjp1;
 N2Coeff(Ux, &cim1, &ci0, &cip1); N2Coeff(Uy, &cjm1, &cj0, &cjp1);
 #pragma omp for schedule(static, 1)
   for (i=0; i < M; i++)
     for (j=0; j < N; j++)
       V(v,i,j) =
```

```
          cim1*(cjm1*V(u,i-1,j-1) + cj0*V(u,i-1,j) + cjp1*V(u,i-1,j+1)) +
          ci0 *(cjm1*V(u,i  ,j-1) + cj0*V(u,i,  j) + cjp1*V(u,i,  j+1)) +
          cip1*(cjm1*V(u,i+1,j-1) + cj0*V(u,i+1,j) + cjp1*V(u,i+1,j+1));
} //omp1dUpdateAdvectField()
```

omp1dUpdateAdvectField()

d. maximize cache misses involving coherent writes (without significantly increasing parallel region entry/exits).

To approach this:

1) We cannot significantly increase parallel region entry/exits, so we should allocate the parallel region inside ***omp1dAdvect()*** wrapping the ***for(r…)*** loop.
2) To maximize the coherent writes we can parallelize the inner for loop in ***omp1dUpdateAdvectField()*** in a cyclic fashion using ***schedule(static, 1)***. This will make the cache line useless as adjacent elements in a row will be updated(written) by different threads. So ideally, each update of them will have a cache miss.

Here are the code implementations:

```
void omp1dAdvect(int reps, double *u, int ldu) {
 int r, ldv = N+2;
 double *v = calloc(ldv*(M+2), sizeof(double)); assert(v != NULL);
 #pragma omp parallel default(none) private(r) shared(u, ldu, v, ldv, reps)
 {
   for (r = 0; r < reps; r++) {
     omp1dUpdateBoundary(u, ldu);
     omp1dUpdateAdvectField(&V(u,1,1), ldu, &V(v,1,1), ldv);
     omp1dCopyField(&V(v,1,1), ldv, &V(u,1,1), ldu);
   } //for (r...)
 }
 free(v);
} //omp1dAdvect()
```

omp1dAdvect()

```
static void omp1dUpdateAdvectField(double *u, int ldu, double *v, int ldv) {
 int i, j;
 double Ux = Velx * dt / deltax, Uy = Vely * dt / deltay;
 double cim1, ci0, cip1, cjm1, cj0, cjp1;
 N2Coeff(Ux, &cim1, &ci0, &cip1); N2Coeff(Uy, &cjm1, &cj0, &cjp1);
    for (i=0; i < M; i++)
    #pragma omp for schedule(static, 1) // max corherent write
     for (j=0; j < N; j++)
       V(v,i,j) =
         cim1*(cjm1*V(u,i-1,j-1) + cj0*V(u,i-1,j) + cjp1*V(u,i-1,j+1)) +
         ci0 *(cjm1*V(u,i  ,j-1) + cj0*V(u,i,  j) + cjp1*V(u,i,  j+1)) +
         cip1*(cjm1*V(u,i+1,j-1) + cj0*V(u,i+1,j) + cjp1*V(u,i+1,j+1));
} //omp1dUpdateAdvectField()
```

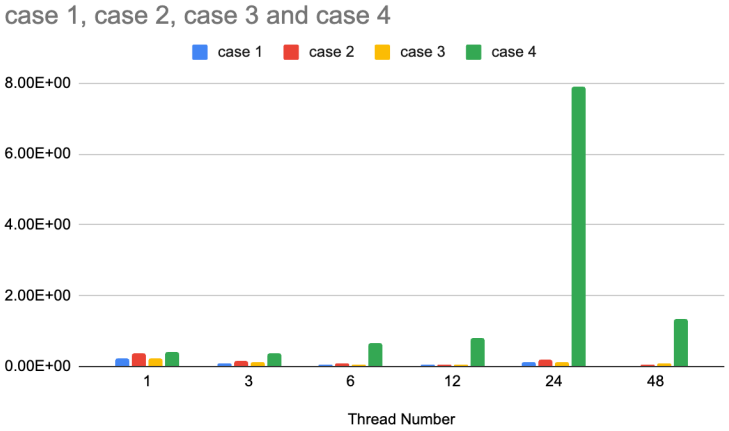
omp1dUpdateAdvectField()

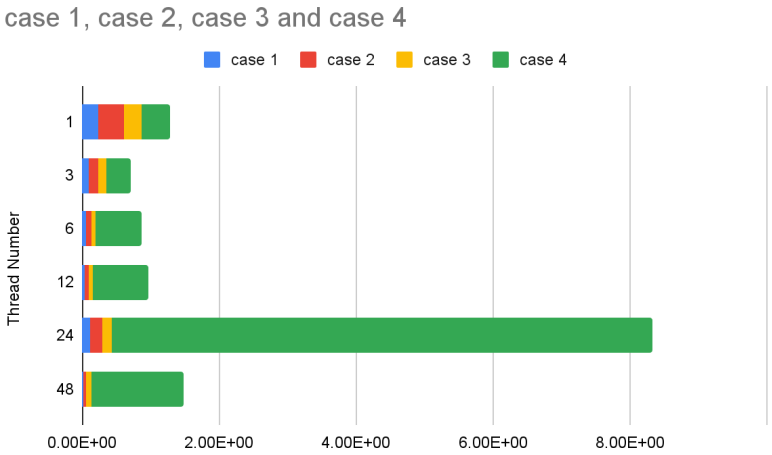
Experiments on 4 cases above

case 1: max performance;
case 2 max parallel region entry/exits;
case 3 max coherent read;
case 4 max coherent write.

Thread Number	case 1	case 2	case 3	case 4
1	2.41E-01	3.80E-01	2.49E-01	4.08E-01
3	9.26E-02	1.44E-01	1.11E-01	3.69E-01
6	5.21E-02	8.33E-02	6.86E-02	6.61E-01
12	3.97E-02	5.44E-02	5.82E-02	8.20E-01
24	1.09E-01	1.96E-01	1.28E-01	7.89E+00
48	2.39E-02	3.48E-02	7.24E-02	1.35E+00

Thread Number and Running Time of 4 Cases.





2. Performance Modelling of Shared Memory Programs

a.

3. Parallelization via 2D Decomposition and an Extended Parallel Region

a. **Explanation of Implementation**:

```
int thread_id = omp_get_thread_num();
int P0 = thread_id / Q;
int Q0 = thread_id % Q;

int M0 = (M / P) * P0;
int M_loc = (P0 < P - 1) ? (M / P) : (M - M0);

int N0 = (N / Q) * Q0;
```

```
int N_loc = (Q0 < Q - 1) ? (N / Q) : (N - N0);
```

sub-array distribution

We can use the code above to divide the advection field into sub-arrays for each thread. This operation happens inside the parallel section to make sure each thread has its own ***P0, Q0, M0, N0, M_loc,*** and ***N_loc.***

The boundary update can be done by using ***#pragma omp for schedule(static),*** this will organize the threads to parallel the boundary update for loop.

```
    updateAdvectField(M_loc, N_loc, &V(u, M0+1, N0+1), ldu, &V(v, M0+1,
N0+1), ldv);
     #pragma omp barrier
     copyField(M_loc, N_loc, &V(v, M0+1, N0+1), ldv, &V(u, M0+1, N0+1),
ldu);
     #pragma omp barrier
```

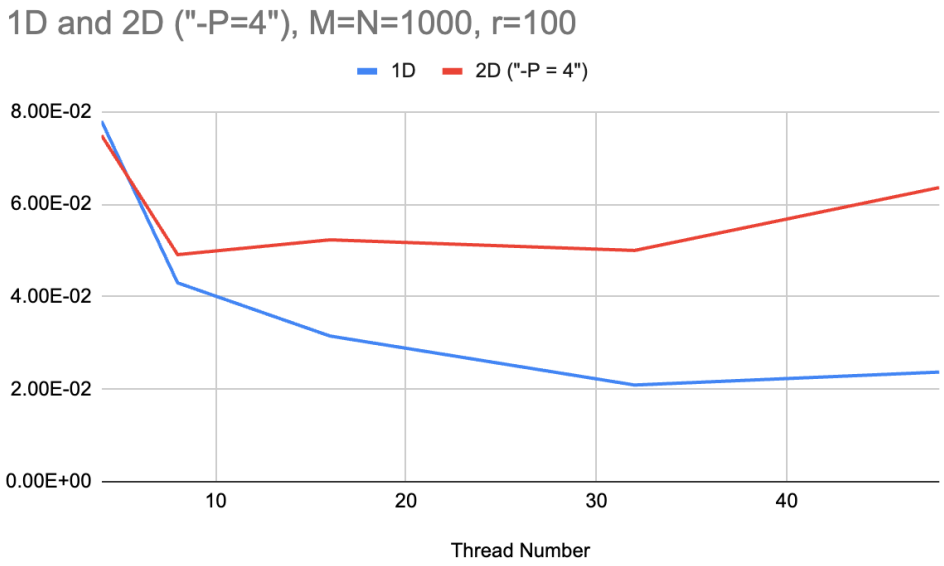
Then we can let each thread calculate and copy its own sub-array. Remember to use ***#pragma omp barrier*** for threads synchronization.

b. **Performance Analysis**:

i. First, I conduct an experiment to compare 1D and 2D grids performance.

I set M=N=1000, r=100, for 2D program, I set "-P 4" and here is the result:

Thread Number	1D	2D ("-P = 4")
4	7.80E-02	7.49E-02
8	4.30E-02	4.91E-02
16	3.15E-02	5.23E-02
32	2.09E-02	5.00E-02
48	2.37E-02	6.36E-02



Analysis:

1) When the number of threads is small (thread_num=4), 2D program is slightly faster than 1D

2) However, as the thread number grows, the 2D program becomes even slower than 1D.
    a) So to answer the question of whether there is any gain from the 2D distribution. Generally, there's no gain of 2D unless the number of threads is small.
    b) Analysis of why this happens:
        i) In my implementation, both 1D and 2D use a single parallel region, which means 2D has no advantage on thread spawning overhead against 1D.
        ii) In 2D implementation, we use ***#pragma omp barrier*** to synchronize each thread to let them keep on the same page and don't miss up on the sub-array update and copy.
        While, in 1D we use ***#pragma omp for schedule(static)***. Using ***#pragma omp for*** might not mean those threads don't need to be synchronized at all. However, 1D implementation don't really have to focus on sub-array synchronization as it doesn't have sub-arrays but just uses for loop to upgrade the entire advection field.
        Also a possible situation is that using ***#pragma omp for*** this synchronization and scheduling are organized by the system or other which is more low-level and more efficient than explicitly using ***#pragma omp barrier.***

3) Compare this with the 1D vs 2D case for the MPI version of the solver, and explain any differences with the relative advantages.
    a) Differences: in OpenMPI implementation, 2D is better in performance than 1D distribution. However, in OpenMP 2D even worse, especially when the thread number is large.

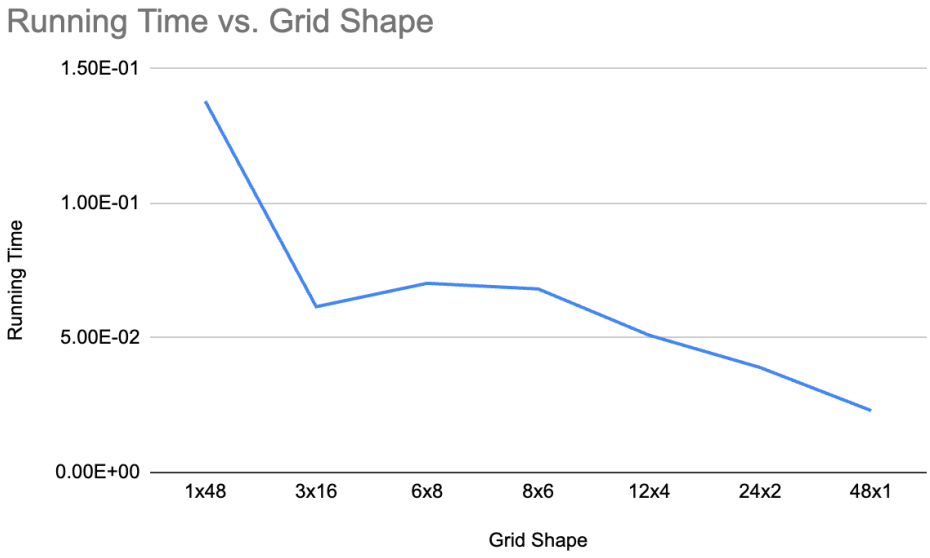
b) Reason

i) For OpenMPI implementation, using 2D grid does improve the performance, as 2D grid distribution can i) reduce the passing message's size; ii) improve the cache hit rate if the distribution is proper.
Moreover, both 1D and 2D in OpenMPI rely on message passing. That's the disadvantage of a separate memory system. So we can say that in OpenMPI, both 1D and 2D will have a period in that some processes are waiting for others' messages. Which can be considered as a kind of synchronization for data dependency.

ii) In OpenMP, as it's for a shared memory system. So there is no message passing. Even 2D distribution somethings can make the sub-array divided uniformly and improve the cache usage. The cost of using 2D in OpenMP is more threads synchronization for sub-arrays coherence. This defuses the improvement of 2D and makes it even worse.

ii. Then I run the experiment to see the relationship between various grid shapes(P * Q) and performance. I set M=N=1000, and r=100 for all kinds of grids.

Grid Shape (P x Q)	Running Time
1x48	1.38E-01
3x16	6.16E-02
6x8	7.03E-02
8x6	6.82E-02
12x4	5.10E-02
24x2	3.90E-02
48x1	2.30E-02



As we can see, as P grows the program become faster. That's probably because in this M, N case, large P within a certain range can distribute the sub-array more properly for cache usage.

c. Was there any advantage in having a single parallel region?

The answer is yes. As the previous task conclusion and experiments outputs. It seems that using fewer parallel region (such as a single one) can improve the performance. I think this is because every time program enters into a new parallel region, it needs to allocate threads accordingly. Which may lead to context switches and other overheads.

4. Additional optimization

a. **Explanation of Implementation**:
According to Vizitiu et al.(2014), at the conclusion of each time step, we can swap the buffers' pointers in order to avoid the need for a memory copy from one buffer to another.

So at in the ***ompAdvectExtra()*** at the end of each iteration, where previously ***copyField()*** is called, instead of calling ***copyField()*** function, we can swap the pointers of buffer ***u*** and ***v***; and if the iteration times ***reps*** is even, then the final buffer pointers are in the right order with the according virtual memory address so we don't need to do anything more. Otherwise, if the ***reps*** is odd the pointers are in wrong order (****u*** has buffer v's address and ****v*** has buffer u's address), so in this case we need to swap them again and copy the data from ***v*** to ***u's*** address.

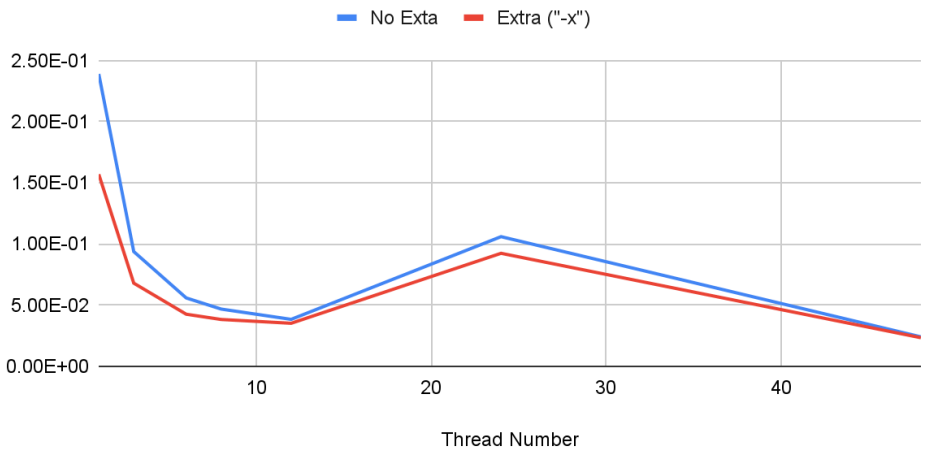
Additionally, the pointer swapping operation in an ***omp single*** section is wrapped by ***#pragma omp barrier*** to make sure the pointer swapping is only conducted by one thread in each iteration.

The pointer swapping optimization will reduce the time call ***copyField()*** from **O(r)** to **O(1)**. Which is useful if **r** is large.

b. Performance Measurement

i. No Extra and Extra ("-x"), with M=N=1000, 1D grid; CPU and memory bind on 1-24 threads case, no bind on 48 threads case:

Thread Number	No Extra	Extra ("-x")
1	2.39E-01	1.57E-01
3	9.38E-02	6.80E-02
6	5.59E-02	4.26E-02
8	4.68E-02	3.83E-02
12	3.84E-02	3.52E-02
24	1.06E-01	9.24E-02
48	2.41E-02	2.34E-02

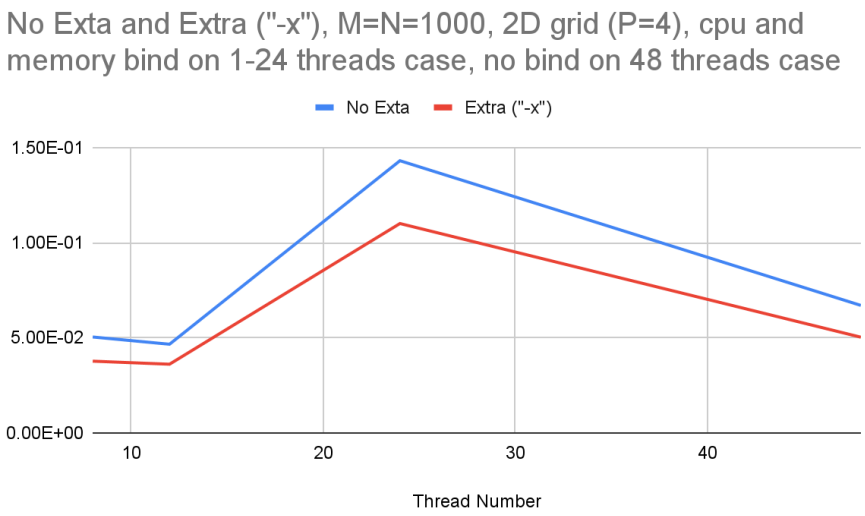


Analysis:

1) From the chart we can see the optimization works well when the thread number is small. However, when the thread number gets bigger, the optimization effect is not obvious, that's probably because in the ***ompAdvectExtra()*** we need to use ***#pragma omp barrier*** but not in the 1D non-optimized implementation. So the ***#pragma omp barrier*** will let the thread waiting time dominate when the number of threads is large.

ii. To defuse the influence of ***#pragma omp barrier***, I conduct an experiment that is: No Extra and Extra ("-x"), M=N=1000, 2D grid (P=4), CPU and memory binding on 1-24 threads case, no bind on 48 threads case.

Thread Number	No Extra	Extra ("-x")
8	5.04E-02	3.77E-02
12	4.66E-02	3.61E-02
24	1.43E-01	1.10E-01
48	6.70E-02	5.03E-02



Analysis:

1) From the data in this experiment, we can see a more obvious effect of the pointer swapping optimization.

Part 2: CUDA

5. Baseline GPU Implementation

a. **Overall Explanation**:

i. According to the parameters, there will be ***Gx * Bx * Gy *By*** threads running each parallel kernel function on device.

```
void cuda2DAdvect(int reps, double *u, int ldu) {
 double Ux = Velx * dt / deltax;
 double Uy = Vely * dt / deltay;
 int ldv = N + 2;
 double *v;
 HANDLE_ERROR( cudaMalloc(&v, ldv*(M+2)*sizeof(double)) );

 dim3 block(Bx, By);
 dim3 grid(Gx, Gy);

 for (int r = 0; r < reps; r++) {
   updateBoundaryNSKernel<<<grid, block>>>(M, N, u, ldu);
   updateBoundaryEWKernel<<<grid, block>>>(M, N, u, ldu);
   updateAdvectFieldKernel<<<grid, block>>>(M, N, u, ldu, v, ldv, Ux, Uy);
   copyFieldKernel <<<grid, block>>> (M, N, v, ldv, u, ldu);
 } //for(r...)

 HANDLE_ERROR( cudaFree(v) );
} //cuda2DAdvect()
```

The idea of distributing the workload to threads in ***updateAdvectFieldKernel()*** and ***copyFieldKernel()*** is similar to the sub-array distribution in OpenMP

implementation. Basically, according to ***threadIdx, blockIdx, blockDim, gridDim,*** we can calculate the sub-array coordinates and their local field size for each thread. Then each thread can operate on the thread's sub-array.

```
int xDim = blockDim.x * gridDim.x;
int yDim = blockDim.y * gridDim.y;
int x = blockIdx.x * blockDim.x + threadIdx.x;
int y = blockIdx.y * blockDim.y + threadIdx.y;

int M0 = (M / xDim) * x;
int M_loc = (x < xDim - 1) ? (M / xDim) : (M - M0);

int N0 = (N / yDim) * y;
int N_loc = (y < yDim - 1) ? (N / yDim) : (N - N0);
```

*sub-array distribution in **updateAdvectFieldKernel()** & **copyFieldKernel()***

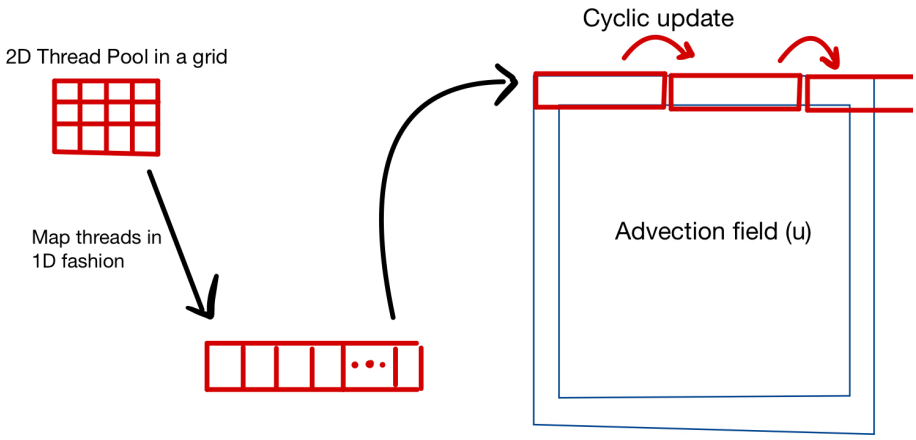
However, the parallel idea of updating the boundary (***updateBoundaryNSKernel()*** and ***updateBoundaryEWKernel()***) is different than the above idea, and I'll talk about it later.

b. **Parallelization Strategy for Boundary Updates:**

i. Naturally, in this program, all threads can be easily indexed in a 2D fashion. However, when we update the boundary, we do it in a 1D fashion(rows only when doing ***updateBoundaryNSKernel()*** and columns only when doing ***updateBoundaryEWKernel()***)

So to make full use of all threads, we can first map the thread index in 1D fashion. Then we can use the 1D index to let threads update boundary cells in a cyclic fashion.

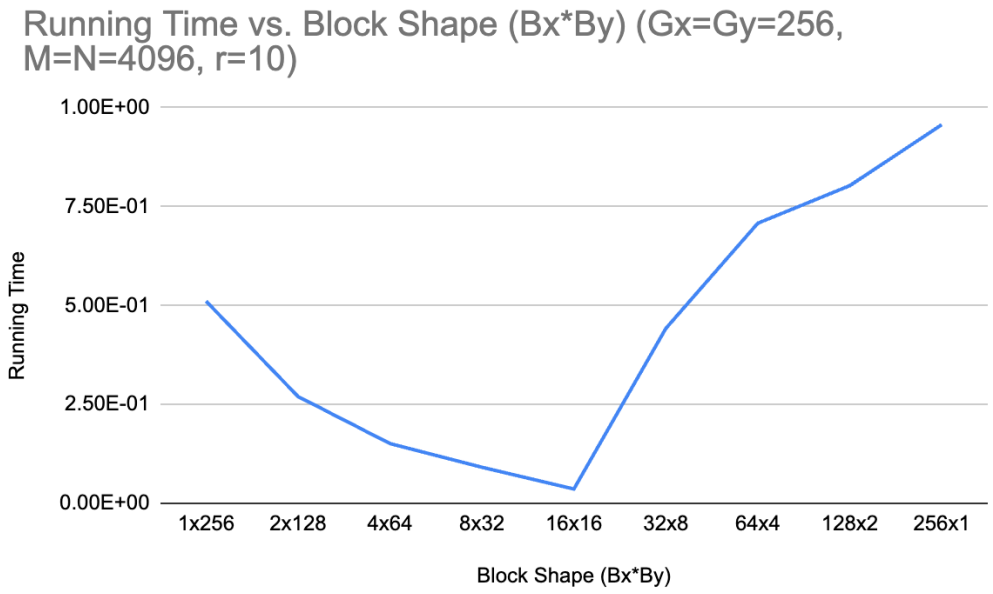
The detail of the implementation of the boundary update is already in the according file in the GitLab repo. And here's a figure to help you understand it.



c. **Perform experiments to determine the effects of varying Gx, Gy, Bx, and By**:

i. Experiment on different block shapes with fixed M=N=4096, r=10, Gx=Gy=256, this makes the total thread number fixed.

Block Shape (Bx*By)	Running Time
1x256	5.11E-01
2x128	2.70E-01
4x64	1.51E-01
8x32	9.11E-02
16x16	3.61E-02
32x8	4.42E-01
64x4	7.08E-01
128x2	8.03E-01
256x1	9.57E-01



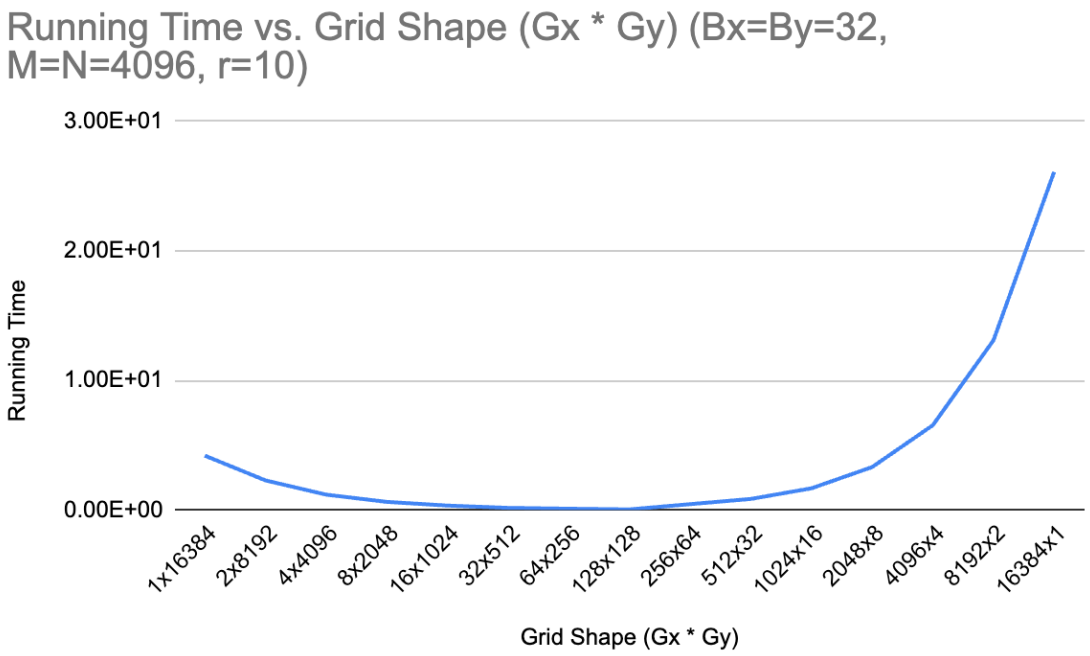
Analysis:

1) When the block shape is near to a square, the performance is best.

2) When the block columns(Bx) are large than the rows(By), the performance becomes worse.
   For example, according to the figure above, when Bx * By = 256*1 is much slower than Bx * By = 1*256.
   This is probably caused by the column-large block that might make the data access in a less memory-contiguous way, which leads to the increase in cache miss.

ii. Experiment on different grid shapes with fixed M=N=4096, r=10, Bx=By=32, this makes the total thread number fixed.

Grid Shape (Gx * Gy)	Running Time
1x16384	4.19E+00
2x8192	2.28E+00
4x4096	1.18E+00
8x2048	6.21E-01
16x1024	3.36E-01
32x512	1.59E-01
64x256	9.44E-02
128x128	4.03E-02
256x64	4.46E-01
512x32	8.48E-01
1024x16	1.67E+00
2048x8	3.31E+00
4096x4	6.54E+00
8192x2	1.31E+01
16384x1	2.61E+01



Analysis:

1) When the grid shape is near to a square, the performance is best.

2) When the grid columns (Gx) are large than the rows (Gy), the performance becomes worse.
   For example, according to the figure above, when Gx * Gy = 16384*1 is much slower than Gx * Gy = 1*16384.
   This is probably caused by the column-large block that might make the data access in a less memory-contiguous way, which leads to the increase in cache miss.

d. **Performance using Cuda2D, Host, and Cuda Serial**.

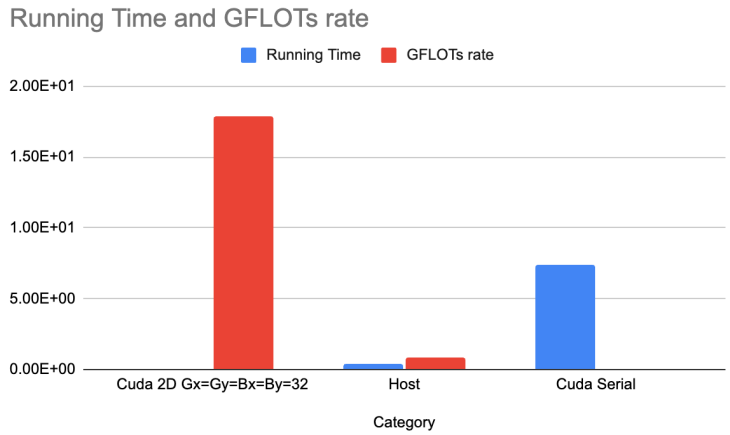
Using this shell script for testing: (M=N=4096, r =1)

```
./testAdvect -g 32,32 -b 32,32 -d 3 4096 4096 1

./testAdvect -h  -d 3 4096 4096 1

./testAdvect -s  -d 3 4096 4096 1
```

Category	Running Time	GFLOTs rate
Cuda 2D Gx=Gy=Bx=By=32	1.87E-02	1.79E+01
Host	3.79E-01	8.84E-01
Cuda Serial	7.40E+00	4.54E-02



Speed up:

1) Cuda 2D against Host speed up: 0.379 / 0.0187 = 20.267
2) Cuda 2D against single GPU speed up: 7.40 / 0.0187 = 395.721

e. **Perform suitable experiments to determine the overhead of a kernel invocation**.

i. According to Harris'(2023) technical blog shared on Nivida Developer. I write this test program to measure the overhead of calling a kernel function:

This program will call an empty kernel function 1,000,000 times.

```
#include <stdio.h>
#define REPs 1e6
__global__
void test()
{
 //this is a test function.
}

int main(void)
{
 cudaEvent_t start, stop;
```

```
 cudaEventCreate(&start);
 cudaEventCreate(&stop);
 cudaEventRecord(start);

 for (int i = 0; i < REPs; ++i)
   test<<<1, 1>>>();

 cudaEventRecord(stop);

 cudaEventSynchronize(stop);
 float milliseconds = 0;
 cudaEventElapsedTime(&milliseconds, start, stop);

 printf("Time (ms): %f\n", milliseconds/REPs);
}
```

The test program is in ***q2_5test.cu*** you can run it by:

```
nvcc -o q2_5test ./q2_5test.cu
./q2_5test > q2_5_test.txt
```

According to the testing result, the overhead of a kernel invocation is 0.003106 ms (3.106e-6s).

6. Optimized GPU Implementation

a. **Describe My Optimizations and Their Rationale**:

i. Pointer swapping to reduce memory copy time:

According to Vizitiu et al.(2014), at the conclusion of each time step, we can swap the buffers' pointers in order to avoid the need for a memory copy from one buffer to another.

```
void cudaOptAdvect(int reps, double *u, int ldu, int w) {
  ......
  for (int r = 0; r < reps; r++) {
   Compute the advection field.....
   cudaDeviceSynchronize();
   double *tmp = u;
   u = v;
   v = tmp;
 } //for(r...)
 if (reps % 2 == 1) {
   double *tmp = u;
   u = v;
   v = tmp;
   HANDLE_ERROR( cudaMemcpy(u, v, ldv*(M+2)*sizeof(double),
```

```
cudaMemcpyDeviceToDevice) );
 }
 HANDLE_ERROR( cudaFree(v) );
} //cudaOptAdvect()
```

The pseudocode of pointer swapping is shown above. The pointer swapping optimization will reduce the time to call ***cudaMemcpy()*** from **O(r)** to **O(1)**. Which is useful if **r** is large.

The pointer swapping optimization will be activated by setting parameter w to be 1 ("-w 1"), w = 0 by default, so the pointer swapping is deactivated by default.

ii. In-block Shared Memory Optimization:

According to Vizitiu et al.(2014), in the context of kernels utilizing 2D thread blocks, shared memory can be employed to decrease the number of accesses to global memory.

The optimization works as following steps:

1) Set Shared Memory Size: The shared memory array's size, designated for this version of the kernel, is calculated by the formula (blockXDim + 2) * (blockYDim + 2). The additional space is for the halo that we need for each block's update

2) Copy Field: Initially, each thread reads the grid point value it's managing and stores it in shared memory. Subsequently, threads at the block's boundary load the remaining values.

3) Update Advection: After that, when the Copy Field is finished (we may need to use in-block synchronization), each thread updates a cell accordingly, and they fetch the adjacent cells' data from shared memory but not from buffer ***u*** which is in the global memory. This can reduce the memory access time.

iii. Parameter Optimization:

The parameter(Gx, Gy, Bx, By) that passed in might not be optimal, just as what we discussed in **Question 5**, good gridDim and blockDim parameters can take the advantage of GPU memory architecture and the SIMT computation module. Also, the gap between using bad parameters and using good parameters is significantly large(like the gap between the case when Bx*By is 16*16 and the case Bx*By is 256*1).

So as the reason is given above, to maximize the power of GPU, each time before the ***updateAdvectFieldOpt()*** function is called, the program will hardcore the blockDim to 16*16. Also, a proper gridDim will be calculated according to the M, N.

This will make sure that each thread only handles one cell in the advection field, which is also the same situation Vizitiu et al.(2014) mentioned in Method(4) section.

Here is the code implementation about parameter optimization:

```
int blockDimX = 16;
int blockDimY = 16;
int gridDimX = (M + blockDimX - 1) / blockDimX;
int gridDimY = (N + blockDimY - 1) / blockDimY;
dim3 block(blockDimX, blockDimY);
dim3 grid(gridDimX, gridDimY);
size_t sharedMemSize = (blockDimX+2) * (blockDimY+2) * sizeof(double);
```

b. Experiments

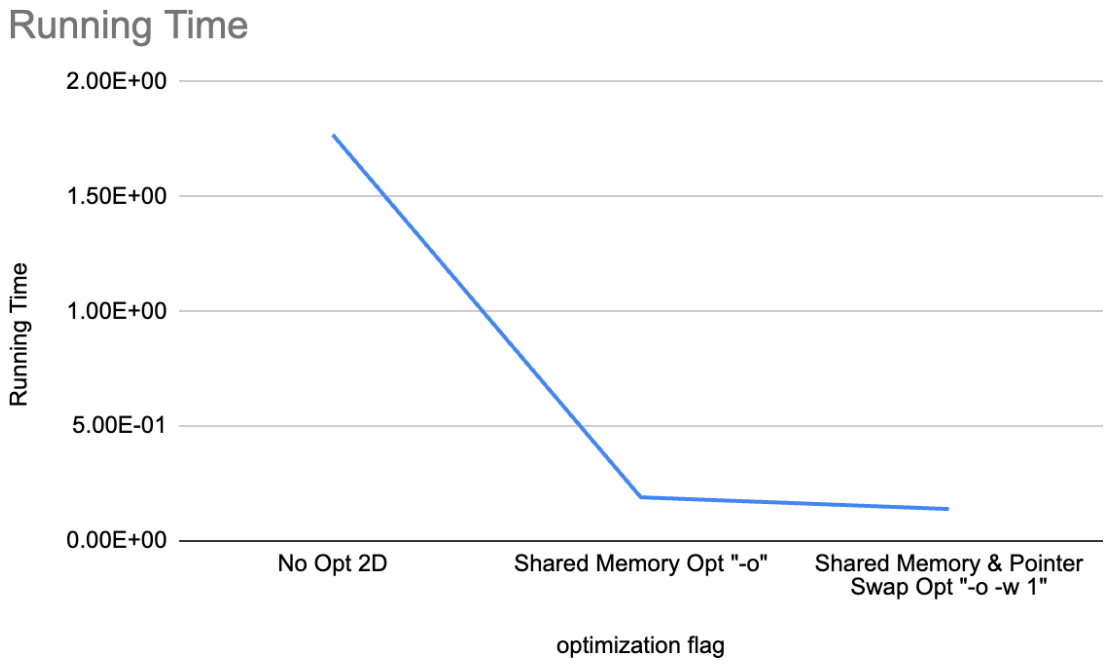
Conduct experiments with Gx=Gy=Bx=By=32, M=N=4096, r=100, using different optimization categories.

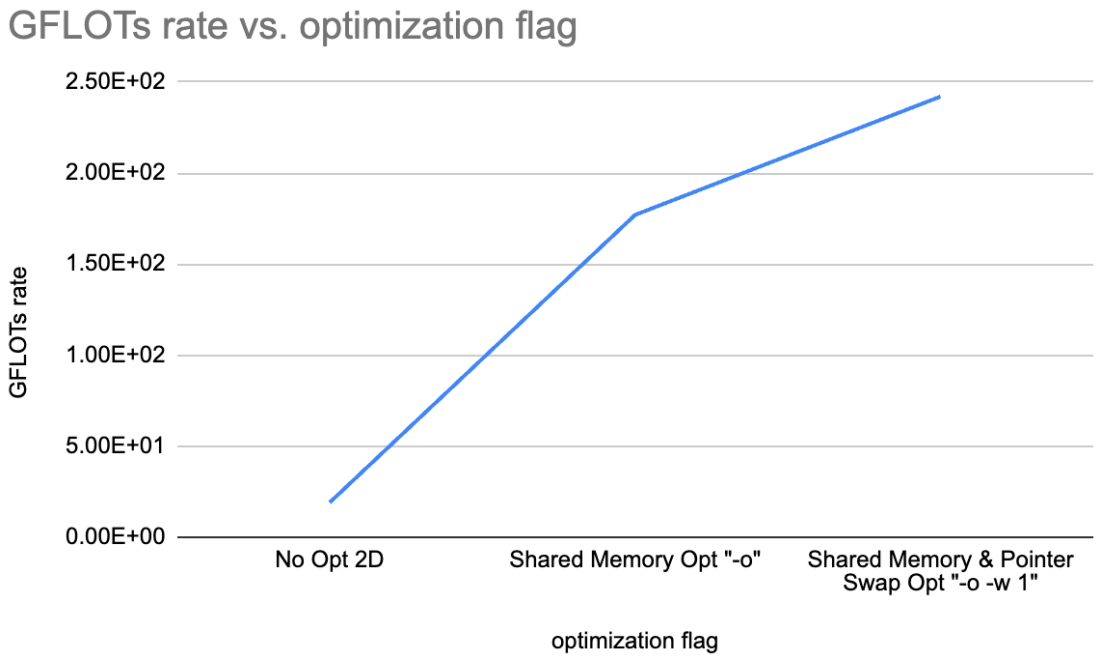
```
# No Opt 2D
./testAdvect -g 32,32 -b 32,32 -d 3 4096 4096 100

# Shared Mem Opt 2D, No Pointer Swapping
./testAdvect -g 32,32 -b 32,32 -d 3 -o 4096 4096 100

# Shared Mem Opt 2D, With Pointer Swapping
./testAdvect -g 32,32 -b 32,32 -d 3 -o -w 1 4096 4096 100
```

optimization flag	Running Time	GFLOTs rate
No Opt 2D	1.77E+00	1.90E+01
Shared Memory Opt "-o"	1.90E-01	1.77E+02
Shared Memory & Pointer Swap Opt "-o -w 1"	1.39E-01	2.42E+02





We can that all optimizations kick in well. The pointer-swapping optimization doesn't have too much effect on reducing the running time but it increases the GFLOTs rate decently. The fully optimized program can have a **1.77 / 0.139 = 12.733x** speedup against the non-optimized 2D cuda.

7. Comparison of the Programming Models

a. **OpenMP**:
   i. Design and Implementation: OpenMP is the simplest and most straightforward of the three for design and implementation. It allows for incremental parallelism, meaning you can take a serial program and gradually parallelize it. It uses compiler

directives, making it less intrusive to the code. The OpenMP API handles many low-level details such as thread creation and synchronization.

ii. Debug: Debugging OpenMP programs can be difficult due to the complexities of parallel programming, including race conditions and deadlocks. However, there are robust tools and resources available for debugging.

iii. Performance: OpenMP can provide substantial speedups on multicore and many-core systems, but its performance is limited by the architecture of shared-memory systems. For many common types of calculations, it offers sufficient performance.

b. **MPI**:

i. Design and Implementation: MPI provides a high degree of control and is highly scalable, but this comes at the cost of increased complexity. Designing and implementing MPI programs can be challenging, especially for large-scale, distributed problems.

ii. Debug: Debugging MPI programs can be difficult due to the asynchronous nature of message passing and the complexity of distributed memory systems. The distributed nature of MPI can make it hard to replicate and identify bugs.

iii. Performance: MPI can offer significant speedups for large-scale, distributed problems. It is particularly effective on distributed-memory systems and can be scaled across multiple nodes in a cluster, enabling it to handle extremely large computations. However, for small-scale problems, the advantage of MPI is not that clear

c. **CUDA**:

i. Design and Implementation: CUDA is designed for programming NVIDIA GPUs. It provides low-level access to the hardware, making the design and implementation more complex than OpenMP and MPI. Understanding the GPU architecture, memory management, and thread hierarchy is important for effective CUDA programming.

ii. Debug: Debugging CUDA programs is a complex task due to the need to handle GPU-specific issues like thread divergence, memory access issues, and proper synchronization. Specialized debugging techniques and tools are required.

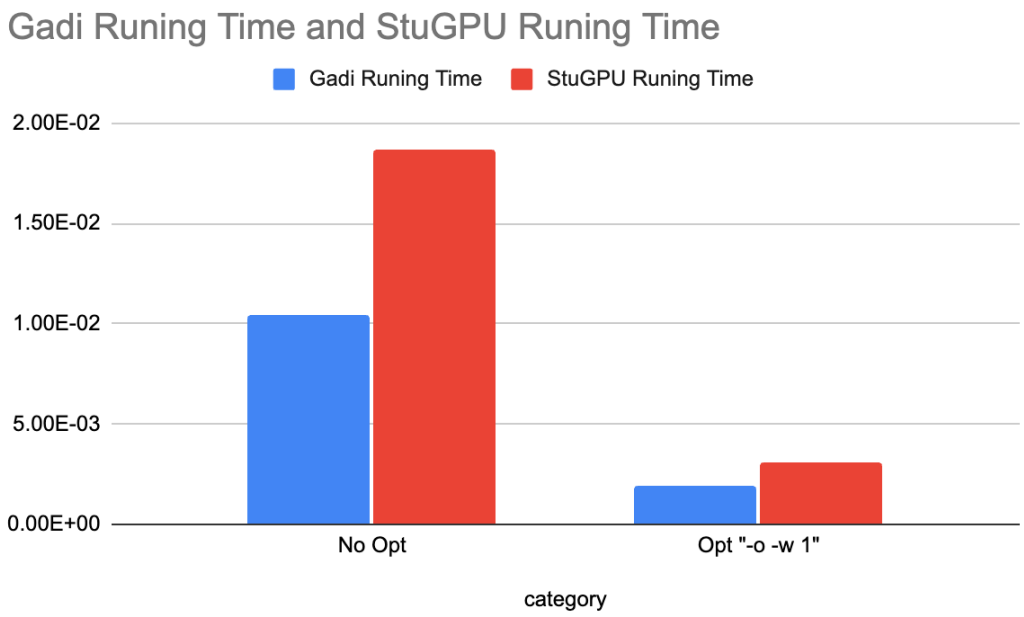
iii. Performance: The performance gain from CUDA can be high for certain types of computations, especially those that are computationally intensive and can be broken down into many small, independent tasks which can take advantage of the SIMT pattern. CUDA is particularly well-suited for tasks that require a high degree of fine-grained parallelism.

8. Optional

Here is the experiment I run on both Gadi server using Tesla V100-SXM2-32GB GPU and stugpu server using GeForce RTX 2080 Ti.

Gx=Gy=Bx=By=32, M=N=4096, r=1

category	Gadi Running Time	StuGPU Running Time
No Opt	1.04E-02	1.87E-02
Opt "-o -w 1"	1.90E-03	3.08E-03



Two GPUs' info:

```
Name:  GeForce RTX 2080 Ti
Compute capability:  7.5
Clock rate:  1665000
Device copy overlap:  Enabled
Kernel execution timeout :  Disabled
    --- Memory Information for device 0 ---
Total global mem:  11554717696
Total constant Mem:  65536
Max mem pitch:  2147483647
Texture Alignment:  512
    --- MP Information for device 0 ---
Multiprocessor count:  68
Shared mem per block:  49152
Registers per block:  65536
Threads in warp:  32
Max threads per block:  1024
Max thread dimensions:  (1024, 1024, 64)
Max grid dimensions:  (2147483647, 65535, 65535)
```

```
Name:  Tesla V100-SXM2-32GB
Compute capability:  7.0
Clock rate:  1530000
Device copy overlap:  Enabled
Kernel execution timeout :  Disabled
    --- Memory Information for device 0 ---
Total global mem:  34079637504
Total constant Mem:  65536
Max mem pitch:  2147483647
Texture Alignment:  512
    --- MP Information for device 0 ---
Multiprocessor count:  80
Shared mem per block:  49152
Registers per block:  65536
Threads in warp:  32
Max threads per block:  1024
Max thread dimensions:  (1024, 1024, 64)
Max grid dimensions:  (2147483647, 65535, 65535)
```

Analysis:

We can see that code running on Gadi is around 2 times faster than on stugpu server.

This is probable because:

1) The Tesla V100 has 80 multiprocessors, compared to 68 on the RTX 2080 Ti. More multiprocessors can mean higher performance, especially for code that scales well with the number of multiprocessors.
2) The Tesla V100 has a total global memory of approximately 32 GB, while the RTX 2080 Ti has around 11 GB. For memory-bound computations or if the data processed is large, having more memory can significantly improve performance.
3) RTX 2080 Ti indeed has a higher clock rate and computer capability, however, I think that the number of multiprocessors is a more crucial factor that influences the performance of this problem size. Given that the V100 has more multiprocessors, it could more than offset the slightly lower clock rate and compute capability.

9. Reference List

[1]dreamcrash (2020) *Chunksize in openMP static scheduling*, *Stack Overflow*. Available at: https://stackoverflow.com/questions/65180820/chunksize-in-openmp-static-scheduling/65182945#65182945 (Accessed: 22 May 2023).

[2]Harris, M. (2023) *How to implement performance metrics in Cuda C/C++*, *NVIDIA Technical Blog*. Available at: https://developer.nvidia.com/blog/how-implement-performance-metrics-cuda-cc/ (Accessed: 25 May 2023).

[3]Vizitiu, A. *et al.* (2014) 'Optimized three-dimensional stencil computation on Fermi and Kepler gpus', *2014 IEEE High Performance Extreme Computing Conference (HPEC)* [Preprint]. doi:10.1109/hpec.2014.7040968.