This work is accomplished by Heming Zhu with the help of course lectures and tutors.

1D omp Parallelization

Universal setup:

The problem size is set at 2000*2000 repeating 20 times in this chapter of the report about omp. The number of cores is limited to 20 during the time of measuring region entry or exit, coherent write miss, and coherent read miss so that each process always can fill all the information about their responsible pixels in their L2 cache. Which minimizes the noise from cache capacity miss.

All the experiments are conducted under the assumption of &V(u,1,1) and &V(v,1,1) are a multiple of one cache line, 8*n.

Maximizing Performance:

```
int blockSize = N/omp_get_num_procs()

update(){

#pragma omp parallel for schedule(static, blockSize)

for(i){

        for(j){}

}

}

copy(){

#pragma omp parallel for schedule(static, blockSize)

for(i){

        for(j){}

}

}
```

Under the universal setup, the mean advection time is 1.2925e-1s.

Horizontal even distribution to gain optimized cache behavior and work balance. Totally 19 rows are overlapped which produces coherent write miss. Totally 2 omp starting up time is experienced per iteration.

Below are the recording values of advection time versus number of processors.

P = 1, t = 3.43e-1s; P = 3, t = 1.38e-1s; P = 6, t = 9.02e-2; P = 12, t = 6.86e-2

P = 24, t = 1.37e-1; P = 48, t = 3.58e-2s;

Maximizing Coherent read miss:

```
update(){
#pragma omp parallel for schedule(static, 8)
        for(j){for(i){read u, write to v}}
}
copy(){
#pragma omp parallel for schedule(static, 1)
        For(i){for(j){read v, write to u}}
}
```

Under the universal setup and four repetitive experiments, the mean advection time for maximizing coherent read miss is 1.41e-1s.

For function update(), the omp clause divides the work zone vertically into blocks with a width of 8, precisely of the size of one cache line, so that each process won't be working on cache lines that are previously read and stored by other processors, in order to minimize coherent cache write miss in this function.

For function copy, the omp clause divides the work zone horizontally into blocks with a height of 1, in other words, each processor writes into one row. While thread 1 is doing so on one row, it will invalidate the other 19 threads' cache lines, causing 19 coherent cache read misses in the next iteration. The same story happens at every row, which means 19*2000 = 38000 misses happened at one iteration in total.

Maximizing Coherent Write Miss

```
update(){
#pragma omp parallel for schedule(static, 7)
        for(j){for(i){read u, write to v}}
}
copy(){
#pragma omp parallel for schedule(static, 7)
        For(j){for(i){read v, write to u}}
}
```

Under universal setup, the mean advection time is 1.955e-1s.

The scheme here syncs both functions into vertically divided blocks, with the same width as 7, precisely one double less than the cache line. In this case, within one row of one execution of the update() function, thread 2 will be working one V(v, 1, 7), invalidating thread 1's cache line which got loaded when thread 1 was working on V(v,1,0). The same story happens to thread 3, 4…. Therefore, for one row in update(), 19 coherent write misses can happen, concluding in a total number of 19*2000 = 38000 coherent write misses. The exact same story happens at copy, causing other 38000 misses.

Maximizing Region entry and exit.

```
int blockSize = N/omp_get_num_procs()

update(){

for(j){#pragma omp parallel for schedule(static, blockSize)

        for(i){}

}

}

copy(){

for(j){#pragma omp parallel for schedule(static, blockSize)

        for(i){}

}

}
```

Under universal setup, the meantime is 5.3075e-1s.

Both functions divide the workload horizontally, and evenly so that the cache behavior is optimal. However, there would be parallel region entry or exit for every j's increment, i.e., the width of the whole work zone. Results to 4000 omp invocations per iteration.

Time Modelling

t_s can be calculated by dividing the difference of time between best performance and most entry or exit by the difference of a number of entry or exit, within 20 iterations, ttotal4000*20 entries or exits, solves to t_S = 5.01875e-6s.

The same logic goes to t_{wr}, the difference on time is 0.1175e-1s, and in total 38000*20 read misses, solves to $t_{wr} = 1.546e - 8s$.

For t_{ww}, the difference on time is 0.6625e-1, but ttotal38000*2*20 write misses (one 38000 for update, one 38000 for copy), solves to t_{ww} = 4.358e-8s.

Compares to MPI, which has startup time and communication time measured as $ts = 4.56e-6, tw = 4.400e-10$. The communication time via cache coherence and global memory is larger, while the star-up time is similar.

The time model of one iteration for the best performance version here is:

$$expectation = 2 * t_s + t_w * (M/8) * p + (N * M * t_f)/p$$

The reason of calculating communication time in the manner shown above is due to the reason of cache coherence communication happens in a unit of the cache line.

2d OMP

Under the universal setup, at a thread dimension of 2*10, time is 5.04e-2s,

Dimension of 4*5, time is 3.10e-2s

Dimension of 5*4, time is 2.89e-2s.---B

Dimension of 20*1, time is 2.27e-2s---A

Comparing B and A, the advantage gained from 2d parallelization against 1d is not vast here, frankly, going 2d and making process-wise work zone a square will further damage the performance. The reason of such is due to the high cost of cache communication, and adding 2d processor block are only increasing the overlap borders between processors.

Comparing A with the previous 1d parallelization, 1.2925e-1s, the improvement is exciting, we are talking about removing 20 start-up times here, it is reasonable for such phenomenon to occur.

Extra OMP

One can find a walkaround to completely remove the copy function, before each iteration, base on whether odd iter or even iter, the program decides whether to read from u and write to v, or read from v and write u. In such a method, the copy function can be removed.

Primitive 2d Advection program on GPU

Code Description: The code used for this section will divide the whole workload along the X axis and Y axis, based on the number of total threads along the X axis and Y axis (i.e.., blockDim*gridDim), each thread would need to operate integerDivision(M, T_x) pixels on X axis (T_x means the number of threads along X), for those at the very end, they would need to take care integerDivision(M, T_x)+mod(M, T_x) pixels.

The same optimization strategy goes for boundary exchange. One difference is that the whole workload is (X+2) * (Y+2), including the boundaries.

Experiments: All experiments here were operated on a problem with a size of 8192*8192, and each advection is iterated 8 times. Only advection time is compared since the time has an inversely proportional relationship with the GFLOPs rate under all occasions here. No standard

errors need to be recorded for the following experiments, all the experiments have a consistent behavior with stand errors smaller than 0.01.

E1: Number of Blocks' influence on performance, the size of blocks is kept either low or high for variable control.

A.1: g(64,64), b(8,4), mean advection time: 0.6367s;

B.1: g(64,32), b(8,8), mean advection time: 0.818s;

C.1: g(128, 64), b(4,4), mean advection time: 0.436s;

C.1: g(64, 128), b(4,4), mean advection time: 0.361s;

A.2: g(64, 64), b(16,16), mean advection time: 0.5607s;

B.2: g(64, 32), b(16, 32), mean advection time: 0.5403s;

C.2: g(128, 64), b(8,16), mean advection time:0.525s;

E1 analysis: When the thread number is kept low, it is observed that, while holding the total threads' number to be static, increasing the number of blocks will result in better performance. In the experimenter's understanding, this phenomenon is caused by exposing more parallelism, since setting blocks number to be large, means more SMs are intuited to do parallel works.

A second cluster of experiments was also conducted but with a larger thread size. Although it can still be observed to have a proportional relationship between grid size and performance, the speed-up is small.

This phenomenon reminds the experimenter that the code he is currently using suffers from thread divergence. The work distribution system is too simple, which might cause load imbalances. For example, if X's size is 10, and it is required to distribute it among 6 threads, following the scheme described above, the first five threads will get 1 element each, while the last thread acquires 5 elements. This feature embedded in the code is adding so much noise to the experiment, that makes him hard to control the suitable problem size and parameters among experiments.

E2:

A: g(256, 1), b(16,8) advection time: 7.68e-2s

B: g(1, 256), b(16,8) advection time: 8.41e-1s

C: g(512,1), b(16,8) advection time: 8.07e-2s

D: g(256, 8), b(16,16) advection time 7.47e-2s;

D.2: same parameters with D, but the workload is 2048*2048, advection time: 5.88e-3s

E2 analysis:

Other than the influence of thread divergence, the cause of the above could also be the difference between their warp occupancy. While A has sufficient blocks to always keep schedulers busy, B will only have one block to feed the 4 schedulers, thus, causing 3 schedulers to be idle, and harming the warp occupancy. Further increasing on the Gx will not harvest more efficiency, but cause more thread divergence,

The best performance accumulated so far is recorded in D.

E3:

A: host advection time: 3.55s, speed up against best GPU cases: 47.523 times.

B: upon working on 2048*2048 pixels, single thread advection time: 9.96s, speed up against best GPU cases: 1693.878 times

Optimizing GPU Advection Program via Shared Memory

Code Description: This optimization is looking for a better memory access scheme by utilizing shared memory. In the kernel where each thread attempts to do updates, they will first load a few pixels' info to an array located in block-wise shared memory, before doing the real calculations. Unlike the previous version which all blocks are flattened, and the workload is distributed directly by global thread indexes, the workload distribution scheme here will first use modulo distribution by the dimension of grids, then let each thread use their block-wise thread indexes to decide the zone of work they need to take care of.

During the phase of loading into shared memory, each thread will check whether the pixel they are taking care of now is on the border of one block, if so, then it would need to load one more pixel outside the block as its "halo".

The optimized advection updater has the name of updateAdvectField2d_sharedMem() in the experimenter's code.

Optimization Rationale:

OR.1: The device used in this project is GeForce RTX 2080Ti GPU, it has 49152 Byte of memory available for each block which has exceptionally smaller access delays than global memory. There are no direct documents on the internet that measure the exact latency cycles of shared memory or global memory on GPUs labeled with 2080Ti but concluding from the existing statistics of Tesla 20K GPU (440 cycles on global mem, 48 cycles on shared Mem) or V100 GPU (1029 cycles on global mem, 19 cycles on shared Mem), shared Mem access is about 5 to 10 times faster than global mem access, the reward of replacing one global mem access by shared mem is vast.

The moral of shared memory is that when one thread performs a global-shared memory transaction on location A, other threads in its block can also enjoy low-latency memory access to A, since it is already loaded into shared memory.

In the case of program attempts to allocate more shared memory size than hardware limit, a garbage solution will be issued.

OR.2: To calculate one new value on one pixel, it requires 9 memory reads, 1 mem write, and 21 flops. These flops are all dependent on those 9 mems reads, causing bottlenecks. Assume one block's dimension of workload is 3*3 pixels and a total of 3*3 threads per block, without block-wise cache, each thread needs to do 9 memory reads, assumes there is no memory access collision, and every thread finishes its job at the same time, a latency of 9 memory reads is introduced.

Now, with shared memory optimization, the largest number of MEM reads in one thread needs to be 4 (for those on the corner, they need to load 1 pixel in the block, 2 on the border, and 1 on the corner). Results in a situation with half the number of MEM read than the previous. The experimenter expects the performance to be increased by a scale of 2 under this optimization.

Experiments: All experiments below are performed against a problem size of 8192*8192, iteration is 8.

A.primitive: g(256,64) b(16,16) mean advection time: 0.488s.

A.sharedMem: g(256,64) b(16,16) mean advection time: 0.243s.

B.primitive: g(256,64) b(8,32) mean time: 0.486s;

B.sharedMem: g(256,64) b(8,32) mean time: 0.250s

C.primitive: g(256,64) b(8,8) mean time 0.211s

C.sharedMem: g(256,64) b(8,8) mean time 0.14s

Experiment Analysis:

Firstly, the experimenter was experiencing some "clustered" inconsistency, for one cluster of time, advection time measured from one same program execution is consistent, but if he wait a while and conduct the same experiment again, the advection time would change to another scale, but sustain at that level. Therefore, it is meaningless for him to compare the difference between A or B, or C, since they are measured under different time "clusters". Sometimes, C.primitive results in around 0.5, sometimes, it results in around 0.2. However, luckily, in the same time span, C.sharedMem's results are also scaling up or down, and sustain a similar ratio with C.primitive.

The following discussion will compare the efficiency rewarded by shared memory, under the condition of the same grid dimension, but with different block dimensions.

Comparing A against B, the scale-ups of efficiency are both around 2, while they have different block dimensions, but sustain the same block size.

Comparing A or B against C, C's scale-up of efficiency is around 1.5, while C is having a smaller block size than A or B.

The above observation is reasonable to occur, due to the logic of more threads per block means fewer global-shared memory transactions distributed among threads. An extreme example here is, if the block size is 1*1, and the block workload is 9*9, it means one thread needs to do a total of 11*11 mem access, no matter whether applied shared mem or not. Because there are no other threads to share their loaded memory with it, it can neither share it's with others.

Comparison

Talking about ease of design, omp is the go-to selection, CUDA is the most delicate model I encountered so far, each step taken in CUDA requires a sound understanding of the hardware, and MPI is harder than omp and has a close difficulty with CUDA.

CUDA would also be hard to debug if one wants to utilize its shared memory architecture, this memory is neither a global memory nor a private memory, but a pseudo global and private memory. If one wants to use it, he or she must come up with a delicate mapping system between shared memory and global memory.

MPI has the potential to tackle more complicated jobs, rather than only throughput-oriented subset questions since it is using CPU. MPI can also provide more consistent and cheaper processor synchronization, allowing some more dazzling algorithms like pipelining to be possible,

Although CUDA is hard to master, currently it is providing the best speed up than anyone else, omp at its best is running on one CPU node and provides a 10 times speed up, MPI with four nodes can went 40 times, CUDA is able to push the scale up to 48.