COM6521 Parallel Computing with Graphical Processing Units (GPUs) Assignment

Eylul Lara CIKIS- acp22elc - lara.cikis@gmail.com

OpenMP

Stage 1

Description and Justification

This part is very straight forward to parallelise. I chose to parallelise the outer loop so the algorithm would run in parallel for each particle. I have tried different approaches before settling on the current one including doing nested parallels. However the overhead was not worth it at all.

```
for (x = x_min; x <= x_max; ++x) {
    for (y = y_min; y <= y_max; ++y) {
        x_ab = (float)x + 0.5f - omp_particles[i].location[0];
        y_ab = (float)y + 0.5f - omp_particles[i].location[1];
        pixel_distance = sqrtf(x_ab * x_ab + y_ab * y_ab);
        if (pixel_distance <= omp_particles[i].radius) {
            const unsigned int pixel_offset = y * omp_output_image.width + x;
            #pragma omp atomic
            ++omp_pixel_contribs[pixel_offset];
        }
    }
}
```

I think it not being worth it is due to the fact that this second part which contains the inner loops does a few very simple things and then an atomic line which we have to use to avoid race conditions. That means creating new forks for these inner loops actually makes our program run slower in some cases where the overhead becomes too much to create forks that will wait to go one by one when they reach the atomic part anyways. So the design that worked the best for this part for me was one outer parallel loop with atomic increase for the race condition. The scheduling I have selected for part1 is schedule(dynamic, 8). All of the scheduling was selected by testing each of them one by one. All of my results will be in the table at the end of the OpenMP part and there is also a google sheets link that will hold these test results as well as other ones.

Performance

Particle Count	Output Image Size	CPU Reference Timing (ms)	OpenMP Stage 1 Timing (ms)
10000	1024x1024	17.51	8.06
10000	2048x2048	25.32	12.15
10000	4096x4096	37.93	24.91
100000	1024x1024	163.80	67.37
100000	2048x2048	213.26	78.64
1000000	1024x1024	1608.48	629.35

For all of my benchmarks I chose to do scale both particle count and output image size both independently and few tests where they both are scaled.

For this part we can clearly see that for all of the tests there is a marked improvement. However when we increase the image size this improvement stays around only 2x faster. Compared to that when we start to increase the particle count this parts improvement gets almost as close to 3x the speed. This makes sense since this part is parallelised on the outer loop which uses particle count.

Stage 2

Description and Justification

Stage 2 has 4 parts that we must tackle.
First part is the prefix sum which I decided to leave for the OpenMP part because OpenMP unlike CUDA does not provide us with the necessary tools to handle this operation efficiently. I have tried it with many different things and this is just a more complicated operation than this library is equipped to handle. As the second part is clearly a straightforward serial operation I will skip over that.

The main part we are parallelising is part 3 which this time I have decided to use nested parallels. This was actually again harder to decide if using it was worth it or not because the outer loop again is working with particle count while the inner loop runs on the pixels of each particle. The decision to use parallel loops was made after doing some tests with both having it and not having it. I saw close to 10% increased performance when scaling with image resolution and the downside of the overhead generated affecting performance was negligible. I have also solved the race condition in this part using “#pragma omp atomic capture” which requires

“-openmp:llvm “ additional option to be present when compiling on visual studio. I have talked more in depth about that in compilation part at the end in the appendix.

```
omp_set_nested(1);
int i = 0;
#pragma omp parallel for schedule(dynamic,8)
for (i = 0; i < omp_particles_count; ++i) {
    // Compute bounding box [inclusive-inclusive]
    int x_min = (int)roundf(omp_particles[i].location[0] - omp_particles[i].radius);
    int y_min = (int)roundf(omp_particles[i].location[1] - omp_particles[i].radius);
    int x_max = (int)roundf(omp_particles[i].location[0] + omp_particles[i].radius);
    int y_max = (int)roundf(omp_particles[i].location[1] + omp_particles[i].radius);
    // Clamp bounding box to image bounds
    x_min = x_min < 0 ? 0 : x_min;
    y_min = y_min < 0 ? 0 : y_min;
    x_max = x_max >= omp_output_image.width ? omp_output_image.width - 1 : x_max;
    y_max = y_max >= omp_output_image.height ? omp_output_image.height - 1 : y_max;
    // Store data for every pixel within the bounding box that falls within the radius
    int x = x_min;
    int y = y_min;
    #pragma omp parallel for private(x, y) schedule(dynamic,8)
    for (x = x_min; x <= x_max; ++x) {
        for (y = y_min; y <= y_max; ++y) {
            const float x_ab = (float)x + 0.5f - omp_particles[i].location[0];
            const float y_ab = (float)y + 0.5f - omp_particles[i].location[1];
            const float pixel_distance = sqrtf(x_ab * x_ab + y_ab * y_ab);
            if (pixel_distance <= omp_particles[i].radius) {
                const unsigned int pixel_offset = y * omp_output_image.width + x;
                // Offset into cpu_pixel_contrib buffers is index + histogram
                // Increment cpu_pixel_contribs, so next contributor stores to correct offset
                unsigned int storage_offset;
                #pragma omp atomic capture
                    storage_offset =  omp_pixel_contribs[pixel_offset]++;
                storage_offset += omp_pixel_index[pixel_offset];
                // Copy data to cpu_pixel_contrib buffers
                memcpy(omp_pixel_contrib_colours + (4 * storage_offset), omp_particles[i].color, 4 * sizeof(unsigned char));
                memcpy(omp_pixel_contrib_depth + storage_offset, &omp_particles[i].location[2], sizeof(float));
            }
```

The last part of this stage is the pair sorting which i have decided to leave it with the original cpu sorting solution. I have tried parallelising the loop and calling it for each run however this quickly gets out of hand when resolution is increased and halts the program.

I have once again chosen to use dynamic 8 scheduling for these parallels from my tests done on them.

Performance

Particle Count	Output Image Size	CPU Reference Timing (ms)	OpenMP Stage 2 Timing (ms)
10000	1024x1024	82.34	57.73
10000	2048x2048	83.83	53.50
10000	4096x4096	128.00	116.88
100000	1024x1024	1747.51	1451.35

100000	2048x2048	1178.30	890.23
1000000	1024x1024	24001.35	18970.56

This part is a bit more difficult to understand where the improvement comes from. And I will explain why. As we can see we see improvement in the range of 35-70%. However this improvement decreases especially when we scale with resolution. At first when I was doing the assignment this surprised me however once I got to the CUDA part and saw similar results I understood it better. I did some profiling tests and found that almost 50% of stage 2's time on parallel systems was taken by the cpu sorting algorithm. As that algorithm runs for each pixel of the image as resolution increases that last part takes up more and more time of this stage so while part 3 which is the most important part might only take up 20% sorting may take up to 70% of the total time of the stage.

Still even with that caveat this stage shows improvement over the CPU solution and if i had more time I would have liked to come up with a solution for the cpu sorting algorithm at the end.

Stage 3

Description and Justification

Stage 3 compared to previous two stages is much easier as it is free of race conditions there is no need for atomics or critical sections.
For this part what I have done is parallelising the loops. One thing that is different from the first 2 stages is, stage 3 runs over for each pixel instead of each particle.

Another thing that differentiates between this and other stages is. This stage scales so much worse with dynamic scheduling. From my testing if dynamic scheduling is used on stage 3 it can actually go slower than the cpu bound solution. So at this stage I have elected to use static 8 scheduling which was the by far best result from my tests. You can check out the full results of different schedulings on my test results sheet.

Performance

Particle Count	Output Image Size	CPU Reference Timing (ms)	OpenMP Stage 3 Timing (ms)
10000	1024x1024	21.37	8.39
10000	2048x2048	23.18	13.58

10000	4096x4096	35.72	34.34
100000	1024x1024	216.23	41.39
100000	2048x2048	214.81	51.512
1000000	1024x1024	2153.62	321.70

From our results we can see that this stage can show much better results compared to CPU timings. However a certain outlier scenario is also present. While if the particle count is sufficiently high if output image is scaled everything is alright. If the particle count is very low (100-500) and output image is very big (4096x4096) this part will be slower than the CPU solution. I actually could not find the cause of this issue. However I suspect it is due to the static scheduling I have decided to use. What I am guessing the issue is that a lot of threads are not doing much operations since there are very few particles so without dynamic scheduling this creates a lot of empty time for threads.

CUDA

Stage 1

Description and Justification

The device code for stage 1 of cuda looks very similar to the OpenMP version. A few key differences are;
Firstly we took advantage of the read only cache property of using CUDA. By using “const Particle* __restrict__ d_particles” we have signaled the compiler that we want the particles to be pulled through the read-only cache. This actually provided us with more than 10% increased performance in some cases for stage 1 which otherwise did not have many improvements to be gained.
Another thing different is using cuda function atomicAdd() which is much faster than anything OpenMP can do as CUDA functions are specialized on what they do.

For my blocks per grid and threads per block. I have chosen to get the device properties and reading the multiprocessor count and maxThreadsPerBlock from those properties. After getting those I then experimented with different combinations of these and standard block sizes like (32/64/128/256) to find out which ones performed best on my device for a certain stage. For this stage using the device properties proved the best. Which in my case was 1024 on my Nvidia 1660 Laptop GPU.

Performance

Particle Count	Output Image Size	CPU Reference Timing (ms)	CUDA Stage 1 Timing (ms)
10000	1024x1024	17.51	3.84
10000	2048x2048	25.32	5.28
10000	4096x4096	37.93	5.82
100000	1024x1024	163.80	38.02
100000	2048x2048	213.26	52.15
1000000	1024x1024	1608.48	379.09

As we can also see from the results CUDA is much faster compared to both CPU reference timings and the other OpenMP implementation. We are getting 5-7x the speed compared to CPU implementation. And one additional gain we have is CUDA implementation is much more scalable. As we even see increased performance when output image size is bigger here. Now this gain is due to the fact that when the amount of calculations done is small the overhead takes up a bigger portion of runtime. So this supposed gain when scaled will hit a wall eventually when GPU performance is fully saturated. Even with all of that I am very satisfied with the results of stage 1 for CUDA.

Stage 2

Description and Justification

Just like with the OpenMP implementation I will go over each part of stage 2 separately and parts other than part 3 have more interesting things happening this time around.

```
thrust::device_ptr<unsigned int> d_pixel_index_thrust(d_pixel_index);
thrust::device_ptr<unsigned int> d_pixel_contribs_thrust(d_pixel_contribs);
thrust::exclusive_scan(d_pixel_contribs_thrust, d_pixel_contribs_thrust + cuda_output_image_width * cuda_output_image_height+1, d_pixel_index_thrust);


unsigned int TOTAL_CONTRIBS;
cudaMemcpy(&TOTAL_CONTRIBS, d_pixel_index + cuda_output_image_width * cuda_output_image_height, sizeof(unsigned int), cudaMemcpyDeviceToHost);

if (TOTAL_CONTRIBS > cuda_pixel_contrib_count) {
    cuda_pixel_contrib_count = TOTAL_CONTRIBS;
}

if (cpu_pixel_contrib_colours) free(cpu_pixel_contrib_colours);
if (cpu_pixel_contrib_depth) free(cpu_pixel_contrib_depth);
cpu_pixel_contrib_colours = (unsigned char*)malloc(cuda_pixel_contrib_count * 4 * sizeof(unsigned char));
cpu_pixel_contrib_depth = (float*)malloc(cuda_pixel_contrib_count * sizeof(float));
```

Above we can see part 1 and 2 and this time for part 1 I have decided to use the thrust library to calculate the prefix sum operation. This is done by copying the device pointers into thrust device

pointers (this isn't a heavy operation since only the memory address is copied). Then an exclusive scan operation is done.
After this I had a few different ways I could handle the part2. First I chose to keep everything in device memory to not lose any time to copy anything back and forth from host. However this turned out to be slower since allocating memory in GPU is a very very expensive operation. Because of that I have decided to copy back the result of the prefix sum to host and allocate memory there. This by only copying 1 integer to host saves a lot of time that would be wasted.

For part 3 of stage2 like stage1 we are using read only cache to pull d_particless. A big improvement compared to OpenMP is we are again using atomicAdd function which is faster than the OpenMP Implementation.

```
const unsigned int pixel_offset = y * D_OUTPUT_IMAGE_WIDTH + x;
const unsigned int index_offset = d_pixel_index[pixel_offset] + atomicAdd(&d_pixel_contribs[pixel_offset], 1);
// Copy data to d_pixel_contrib buffers
d_pixel_contrib_colours[4 * index_offset + 0] = p.color[0];
d_pixel_contrib_colours[4 * index_offset + 1] = p.color[1];
d_pixel_contrib_colours[4 * index_offset + 2] = p.color[2];
d_pixel_contrib_colours[4 * index_offset + 3] = p.color[3];

//atomicExch(&d_pixel_contrib_depth[index_offset], p.location[2]);
d_pixel_contrib_depth[index_offset] = p.location[2];
```

The big gain on this part is however done by unrolling the memcpy operation in the cpu and OpenMP implementations. This small difference of doing it as array operations instead of memcpy calls gains CUDA implementation a lot of time.

Performance

Particle Count	Output Image Size	CPU Reference Timing (ms)	CUDA Stage 2 Timing (ms)
10000	1024x1024	82.34	58.32
10000	2048x2048	83.83	52.36
10000	4096x4096	128.00	87.33
100000	1024x1024	1747.51	1175.90
100000	2048x2048	1178.30	830.27
1000000	1024x1024	24001.35	16419.62

We do see again much better results than the reference CPU timings. However a close look would show that these gains are very similar to the OpenMP results in certain particle counts and output sizes.

This is unfortunately due to the fact that we are once again using the cpu sorting algorithm. Which has the additional downside when using in combination with CUDA that we have to copy all the data we need to the host to run the sorting and copy all of it back for the next stage to the device again. Which loses a good chunk of time. Only the sorting if we time it takes upwards of 60% of our stage 2 calculation time. Even with all of that time lost CUDA still outperforms both CPU and OpenMP in every category.

Stage 3

Description and Justification

This stage like the OpenMP once again is very simple to parallelise but there are still a few tricky things I had to take into account.

First, unlike the other 2 stages we do not use the prop device thread counts in this stage. This was decided after testing it with all of them using 128 threads per block gave the best result unlike 1024 which was my device property.
Second thing we did was use __ldg to pull the values out of pixel index since we never interact with these values other than reading them using read only cache is no problem. And unlike particles which are structures pixel indexes are just integers which allow us to use __ldg instead of const __restrict__.

```
unsigned int first = __ldg(&d_pixel_index[i]);
unsigned int last = __ldg(&d_pixel_index[i + 1]);
```

Performance

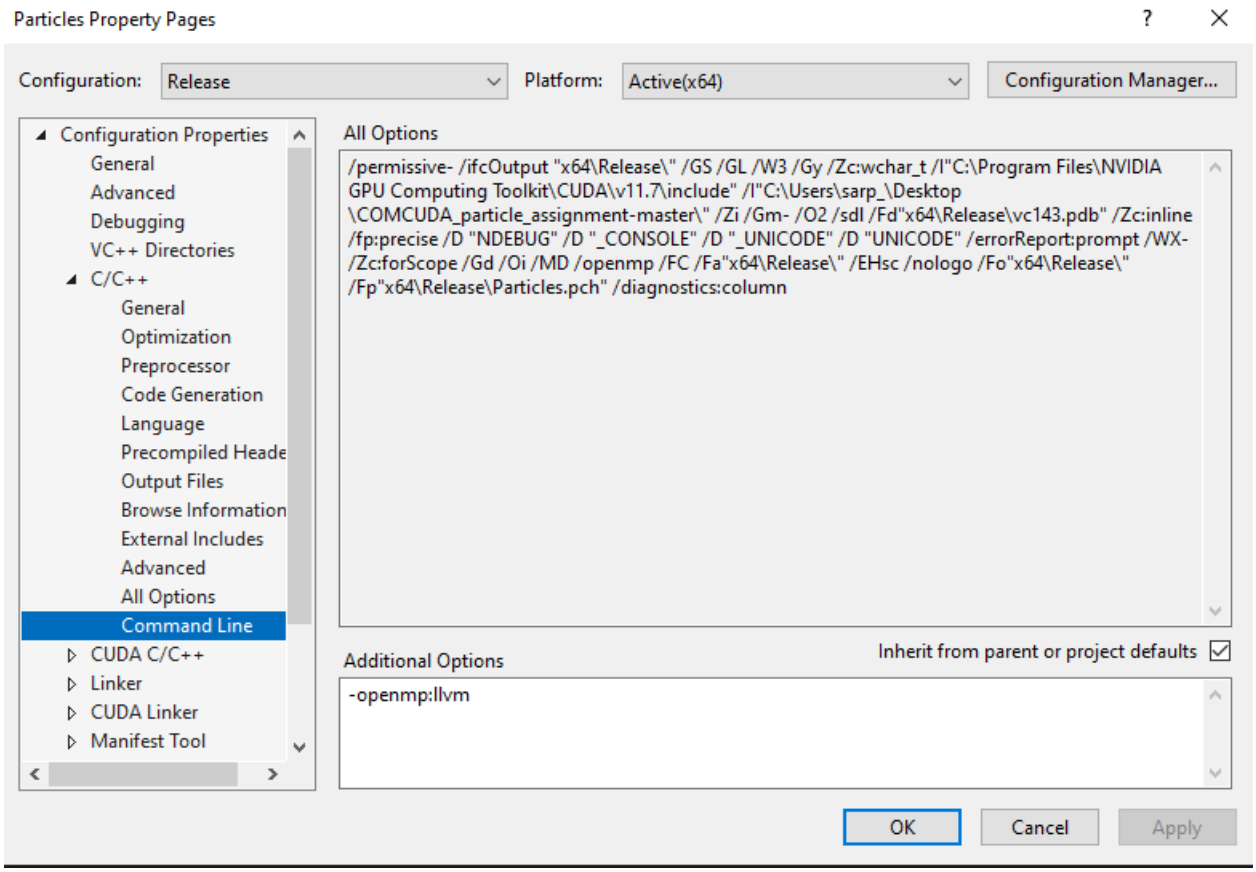
Particle Count	Output Image Size	CPU Reference Timing (ms)	CUDA Stage 3 Timing (ms)
10000	1024x1024	21.37	0.23
10000	2048x2048	23.18	0.36
10000	4096x4096	35.72	0.88
100000	1024x1024	216.23	35.42
100000	2048x2048	214.81	1.85
1000000	1024x1024	2153.62	793.42

Now we are entering the real performance gain. Because stage3 runs on image size instead of particle count. This allows CUDA to shine by using all of the available resources to parallelise. We see 10-30x performance gain in this stage which is incredible. This gain of course decreases when particle count increases the amount of operations needed to be done without changing the parallelisation amount at all. This stage in my opinion shows us what can be done in terms of performance when using CUDA the best.

Appendix

Compilation Notes

The CUDA code completely is ok and complies without problem in diamond machines (tested as of 19/05/2023)
As I have used “#pragma omp atomic capture” in OpenMP stage which is sometimes problematic with very old OpenMP 2.0 installations which Visual Studio in the Diamond computers use. This can be resolved by using a newer OpenMP installation or by adding “-openmp:llvm” as an additional option in the visual studio command line. I asked Robert in the lab if this was ok and he said it was fine to use even if it wasn’t viable on diamond computers, however I wanted to add an appendix about it regardless. It should be ok as long as atomic capture is a valid option.



```
unsigned int storage_offset;
#pragma omp atomic capture
    storage_offset =  omp_pixel_contribs[pixel_offset]++;
storage_offset += omp_pixel_index[pixel_offset];
```

This problem can be solved by changing the above code as such as below in worst case

```
unsigned int storage_offset;
#pragma omp critical
{
    storage_offset = omp_pixel_contribs[pixel_offset]++;
    storage_offset += omp_pixel_index[pixel_offset];
}
```

However this will completely kill the performance of this stage and slow it to a halt which avoids the spirit of this assignment.

Full testing results

Full testing results can be found on the link below

https://docs.google.com/spreadsheets/d/11jNGOLcKRKmtKfmQu5bVgcaWV8A2awObvMjZcj8gh44/edit?usp=sharing