ECSE 420 Parallel Computing

Lab 3 – CUDA Musical Instrument Simulation

Daniel Chernis 260707258
David Gilbert 260746680

Introduction

In this lab, we will write code for a musical instrument simulation (in this case a drum), parallelize it using CUDA, and write a report summarizing our experimental results. We will synthesize drum sounds using a two-dimensional grid of finite elements which will then be

PART 1

1. **Description**

   To set up the synthesys in serial, we set u1[N/2,N\2] = 1 (drum hit), where N = dimension of the drum grid. Then we iterate over a 1D array calculating the middle, then the sides, then corners of the current drum state into array u. The row and column are induced from the current index of iteration the following way:

   - row = index / N
   - Col = index % N

   To compute the middle, we use the previous grid state u1 and the state before that one, u2. We plug the values into the formula as described in the lab specifications.

$$u(i,j) = \frac{\rho[u1(i-1,j) + u1(i+1,j) + u1(i,j-1) + u1(i,j+1) - 4u1(i,j)] + 2u1(i,j) - (1-\eta)u2(i,j)}{1+\eta}$$

$$1 \leq \mathrm{i} \leq \mathrm{N}-2,\ 1 \leq \mathrm{j} \leq \mathrm{N}-2$$

   To compute the sides, we follow the following equations:

$$u(0,i) := Gu(1,i)$$

$$u(N-1,i) := Gu(N-2,i)$$

$$u(i,0) := Gu(i,1)$$

$$u(i,N-1) := Gu(i,N-2)$$

$$1 \leq i \leq N-2$$

   To compute the corners, we follow the following equations:

$$u(0,0) := Gu(1,0)$$

$$u(N-1,0) := Gu(N-2,0)$$

$$u(0,N-1) := Gu(0,N-2)$$

$$u(N-1,N-1) := Gu(N-1,N-2)$$

Finally, at the end of the iteration, we set u2 equal to u1 and u1 equal to u. And repeat the whole process for every iteration.

2. Results

Iterations	u[N/2, N/2]
1	0.000000
2	-0.499800
3	0.000000
4	0.281025
5	0.046828
6	-0.087785
7	-0.321815
8	-0.741367
9	-0.388399
10	0.665226

PART 2

1. **Description**

   The core logic including the physics related formulas remain the same. The code structure was changed, with all the operations happening on a 1D array.

   First off, the execution of the Middle, Sides , and Corners of the grid need to be executed separately so as to avoid threads overwriting each other unintentionally. For each iteration, executed on the CPU, 2 consecutive Cuda calls where made: one for calculating the middle and another for the sides. Then, sequentially, the corners were updated as described in PART 1.

   For calculations of the middle of the grid, the logic remains the same, all that has changed is that instead of iterating over the whole grid, each corresponding thread would start at its respective offset and each iteration is incremented by the amount of threads (row decomposition):

```
int threadOffset = (blockIdx.x * blockDim.x + threadIdx.x);
for (int i = threadOffset; i < (DIM) * (DIM); i += THREAD_COUNT) { … }
```

   For the sides and corners, the calculation is exactly the same, it simply needed to be executed as a whole operation rather than a combination of threads to avoid conflicting grid value modifications.

2. **Results**

Table for Execution Time Given Elements/Thread for Grid Size 4 and 1 Block and 2000 Iterations

Element per Thread	**1**	**2**	**4**	**8**	**16 (*sequential)**
Time (millisec)	6024.259277	6245.333008	6530.942383	6985.452637	7024.748535

* Graphs below

PART 3

Results
The value at position u[N/2, N/2] is:

1. 0.000000
2. 0.000000
3. 0.000000
4. 0.249800
5. 0.000000
6. 0.000000
7. 0.000000
8. 0.140400
9. 0.000000
10. -0.000000
11. 0.000000
12. 0.097422
13. 0.000000
14. -0.000000
15. 0.000000
16. 0.074529
17. 0.000000
18. -0.000000
19. 0.000000
20. 0.060320
21. 0.000000
22. -0.000000
23. 0.000000
24. 0.050645
25. 0.000000
26. -0.000000
27. 0.000000
28. 0.043634
29. 0.000000
30. -0.000000

* Each row is a the next iteration, from 1 to 30

Tables:

- E per T : Element per thread,Thread Count : number of Thread per block
- Times are in milliseconds
- All data computed for 2000 iterations

E per T	thread count	block count	dim	time
8192	4	8	512	3289.625
4096	8	8	512	2340.623
2048	16	8	512	1846.124
1024	32	8	512	1494.495
512	64	8	512	1367.628
256	128	8	512	1263.997
128	256	8	512	1275.655
64	512	8	512	1232.63

E per T	thread count	block count	dim	time
2048	4	32	512	3359.818
1024	8	32	512	2391.795
512	16	32	512	1880.332
256	32	32	512	1559.131
128	64	32	512	1431.35
64	128	32	512	1383.481
32	256	32	512	1381.731
16	512	32	512	1409.874

E per T	thread count	block count	dim	time
1024	4	64	512	3370.809
512	8	64	512	2341.869
256	16	64	512	1779.892
128	32	64	512	1477.921
64	64	64	512	1405.745
32	128	64	512	1393.053
16	256	64	512	1429.592
8	512	64	512	1486.155

E per T	thread count	block count	dim	time
16384	16	1	512	1849.263
8192	32	1	512	1751.031
4096	64	1	512	1459.669
2048	128	1	512	1299.464
1024	256	1	512	1274.922
512	512	1	512	1240.586

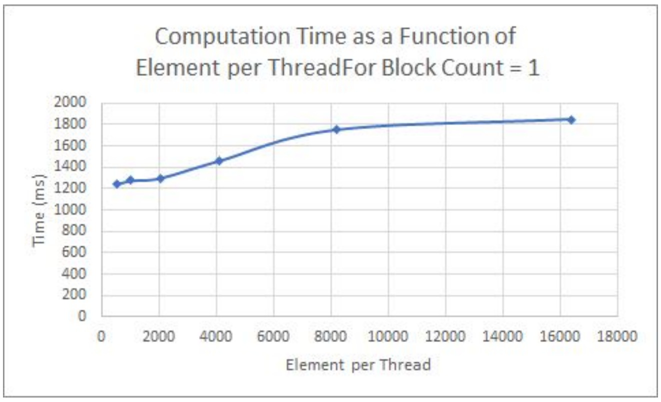
E per T	thread count	block count	dim	time
8192	16	2	512	1787.693
4096	32	2	512	1516.685
2048	64	2	512	1329.033
1024	128	2	512	1300.696
512	256	2	512	1261.806
256	512	2	512	1305.46

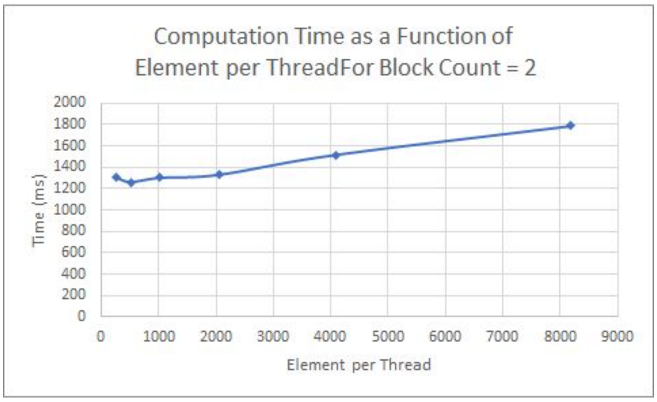
E per T	thread count	block count	dim	time
16384	4	4	512	3371.312
8192	8	4	512	2372.754
4096	16	4	512	1874.705
2048	32	4	512	1606.457
1024	64	4	512	1422.21
512	128	4	512	1355.058
256	256	4	512	1322.063
128	512	4	512	1327.824

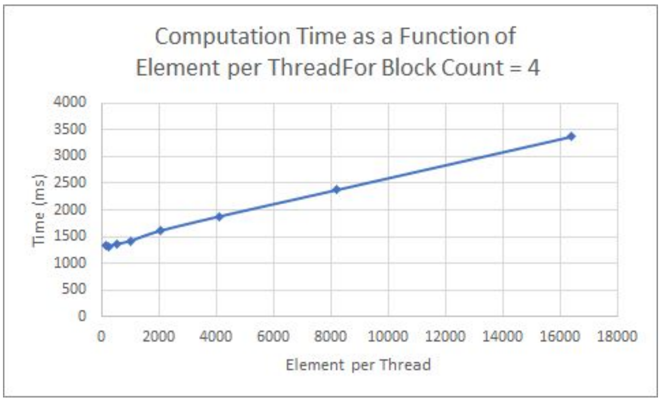
Graphs:

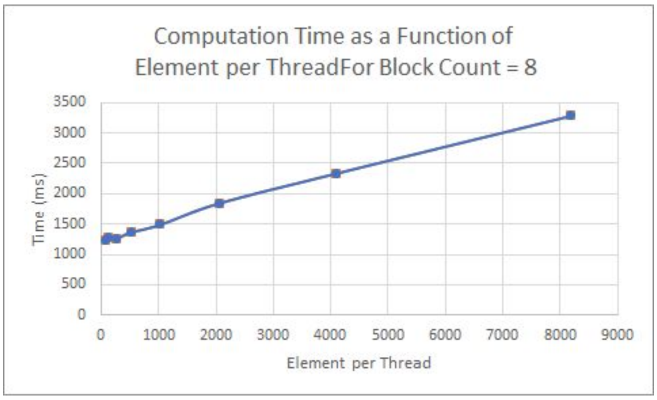
*The number of Thread for each point can be acquired from :

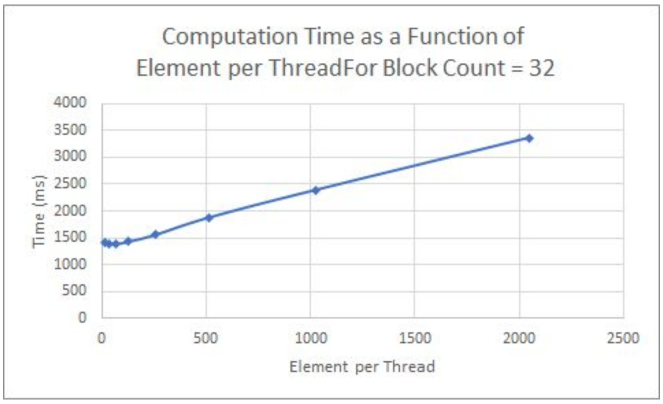
$$Thread\ Count\ =\ Dim^2/(Block\ Count * Elements\ per\ Thread)$$

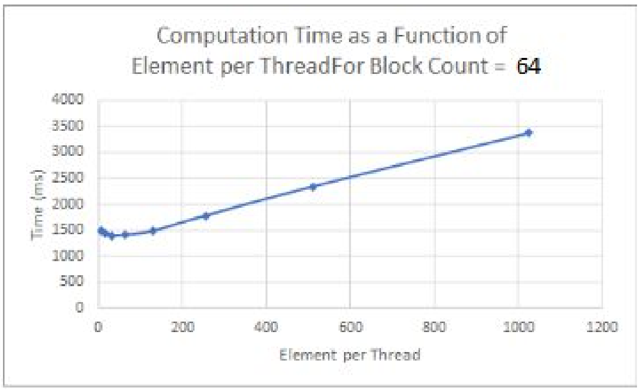












Discussion
The parallelization scheme involves varying the number of threads and blocks allowed. This affects the total number of elements each thread works with. The less elements per thread there are, the more parallelized the program is and the faster it performs.

The general trend is fewer *'elements per thread'* increases overall speed of execution. But when we reach too few elements per thread, the execution time is stalled i.e. from 128 to 64 elements per thread, the runtime decreases by ~72ms but from 64 to 8, the runtime actually increases by ~80 ms (probably due to too much overhead, thread has too few tasks to run to be worth creating overhead for).

The pair of *'thread per block'* and *'block count'* did not seem to affect speed up in general i.e. having 512 threads and 1 blocks yielded similar results to 256 Threads with 2 blocks (1240ms vs 1261ms) this seems true as long as the product of the two and the dimension are constant, so both have the same number of elements per thread. Although there seems to be a considerable decrease in speedup when using 32 or more blocks when compared to the two previous cases (for 32 block and 16 threads we get 1880 which is considerably longer).

Appendix

```
void synthesisSerial(float* u, float* u1, float* u2, int iterations)
{
    u1[DIM / 2 * DIM + DIM / 2] = 1.0; // drum hit

    for (int k = 0; k < iterations; k += 1)
    {
        for (int i = 0; i < (DIM) * (DIM); i += 1)
        {
            int row = i / (DIM);
            int col = i % (DIM);
            int offset = row * DIM + col;
            if (row == 0 || col == 0) continue;

            //Update Inner
            u[offset] =
                RHO * (u1[(row - 1) * DIM + col] +
                    u1[(row + 1) * DIM + col] +
                    u1[row * DIM + col - 1] +
                    u1[row * DIM + col + 1] -
                    4 * u1[offset]) +
                2 * u1[offset] -
                (1 - ETA) * u2[offset];

            u[offset] = u[offset] / (1 + ETA);
        }

        //Update Sides
        for (int j = 1; j < DIM - 1; j++)
        {
            u[0 * DIM + j] = G * u[1 * DIM + j];
            u[(DIM - 1) * DIM + j] = G * u[(DIM - 2) * DIM + j];
            u[j * DIM + 0] = G * u[j * DIM + 1];
            u[j * DIM + (DIM - 1)] = G * u[j * DIM + (DIM - 2)];
        }

        //Update Corners
        u[0] = G * u[1 * DIM + 0];
        u[(DIM - 1) * DIM] = G * u[(DIM - 2) * DIM + 0];
        u[(DIM - 1)] = G * u[DIM - 2];
```

```
        u[(DIM - 1) * DIM + (DIM - 1)] = G * u[(DIM - 1) * DIM + (DIM - 2)];

        // Grid update step
        arrayCopy(u1, u2);
        arrayCopy(u, u1);
    }
}

__global__ void synthesisMiddleParallel(float* u, float* u1, float* u2)
{
    float ETA = 0.0002;
    float RHO = 0.5;
    int threadOffset = (blockIdx.x * blockDim.x + threadIdx.x);

    for (int i = threadOffset; i < (DIM) * (DIM); i += THREAD_COUNT)
    {
        int row = i / (DIM);
        int col = i % (DIM);
        int offset = row * DIM + col;
        if (row == 0 || col == 0) continue;

        u[offset] =
            RHO * (u1[(row - 1) * DIM + col] +
                u1[(row + 1) * DIM + col] +
                u1[row * DIM + col - 1] +
                u1[row * DIM + col + 1] -
                4 * u1[offset]) +
            2 * u1[offset] -
            (1 - ETA) * u2[offset];

        u[offset] = u[offset] / (1 + ETA);
    }
}

__global__ void synthesisSidesParallel(float* u, float* u1, float* u2)
{
    float G = 0.75;
    //Update Sides
    for (int j = 1; j < DIM - 1; j++)
    {
        u[0 * DIM + j] = G * u[1 * DIM + j];
        u[(DIM - 1) * DIM + j] = G * u[(DIM - 2) * DIM + j];
```

```
        u[j * DIM + 0] = G * u[j * DIM + 1];
        u[j * DIM + (DIM - 1)] = G * u[j * DIM + (DIM - 2)];
    }
}

void synthesisParallel(float* d_u, float* d_u1, float* d_u2, int iterations)
{
    d_u1[DIM / 2 * DIM + DIM / 2] = 1.0; // drum hit

    for (int k = 0; k < iterations; k += 1)
    {
        synthesisMiddleParallel << < BLOCK_COUNT, THREAD_COUNT >> > (d_u, d_u1, d_u2);
        cudaDeviceSynchronize();
        synthesisSidesParallel << < BLOCK_COUNT, THREAD_COUNT >> > (d_u, d_u1, d_u2);
        cudaDeviceSynchronize();

        //Update Corners
        d_u[0] = G * d_u[1 * DIM + 0];
        d_u[(DIM - 1) * DIM] = G * d_u[(DIM - 2) * DIM + 0];
        d_u[(DIM - 1)] = G * d_u[DIM - 2];
        d_u[(DIM - 1) * DIM + (DIM - 1)] = G * d_u[(DIM - 1) * DIM + (DIM - 2)];

        // Grid update step in serial
        for (int i = 0; i < DIM * DIM; i++)
            d_u2[i] = d_u1[i];

        for (int i = 0; i < DIM * DIM; i++)
            d_u1[i] = d_u[i];
    }
}
```