10 December 2021 Friday

comp429-Project 2

Group Members:

- M. Musab Küçük (mkucuk18-69910)
- Yavuzhan Akyüz (yakyuz18-68646)

GPU used for testing: TESLA V100

V1: works!

V2: works!

V3: works!

V4: works!

How to run our code:

We have changed our Makefile, so just compile the code with "make" and then there will be executables named: serial, v1, v2, v3, v4 as seen on the Figure 1.

```
[[yakyuz18@login02 prj2_final_code]$ pwd
/scratch/users/yakyuz18/prj2_final_code
[[yakyuz18@login02 prj2_final_code]$ ls -1
Makefile
bin
cardiacsim.cpp
cardiacsim.o
cardiacsim_kernels.cu
cardiacsim_v1.o
cardiacsim_v2.o
cardiacsim_v3.o
cardiacsim_v4.o
proje2_complete_testt.out
runfile.sh
serial
v1
v1.cu
v2
v2.cu
v3
v3.cu
v4
v4.cu
[yakyuz18@login02 prj2_final_code]$
```

Figure 1

-Now it's possible to use "srun" for each of these 1 by 1, or you can use the "sbatch runfile.sh", but please edit the path in "runfile.sh" as seen the figure 2

runfile.sh

```
#!/bin/bash
#SBATCH --job-name=MusabTest
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --time=1:0:0
#SBATCH --output=proje2_complete_testt.out
#SBATCH -p short
#SBATCH --gres=gpu:tesla_v100:1
echo "starting musab-yavuz-test"
/kuacc/users/yakyuz18/prj2_final_code/v1 -n 1024 -t 500
/kuacc/users/yakyuz18/prj2_final_code/v2 -n 1024 -t 500
/kuacc/users/yakyuz18/prj2_final_code/v3 -n 1024 -t 500
/kuacc/users/yakyuz18/prj2_final_code/v4 -n 1024 -t 500
```

Figure 2

Implementation

Version 1:

- We parallelized the serial cpp code in a new v1.cu file. We mostly used the same code, but made critical changes in our simulate function, and our Alloc2D function(we didn't use it to create arrays). Also we made sure to implement correct memory allocations and copies for device.

Changes in Simulate:

- In order to parallelized serial program, we removed all the for loops in our simulate function, and rather we created three __global__ kernel functions: odeKernel, pdeKernel, ghostKernel. Then for our every array (E , R, E_prev) we filled our new kernels accordingly. See Figure 3 for our implementation.

```
__global__ void ghostKernel(double *E_prev, const int n, const int m) {
    int j = threadIdx.x + 1;

    E_prev[j * (n+2)] = E_prev[j * (n+2) + 2];
    E_prev[j * (n+2) + (n + 1)] = E_prev[j * (n + 2) + (n - 1)];

    E_prev[j] = E_prev[2 * (n + 2) + j];
    E_prev[(m + 1) * (n + 2) + j] = E_prev[(m - 1) * (n + 2) + j];
}

__global__ void odeKernel(double *E, double *R,
    const int n, const int m, const double kk,
    const double dt, const double a, const double epsilon,
    const double M1, const double M2, const double b) {

    int i = threadIdx.x + 1;
    int j = blockIdx.x + 1;
    int index = j * (n + 2) + i;

    E[index] = E[index] - dt * (kk * E[index] * (E[index] - a) * (E[index] - 1) + E[index] * R[index]);
    R[index] = R[index] + dt * (epsilon + M1 * R[index] / (E[index] + M2)) * (-R[index] - kk * E[index] * (E[index] - b - 1));
}

__global__ void pdeKernel(double *E, double *E_prev, const double alpha, const int n, const int m) {
    int i = threadIdx.x + 1;
    int j = blockIdx.x + 1;
    int index = j * (n + 2) + i;

    E[index] = E_prev[index] + alpha *
                                  (E_prev[index + 1] + E_prev[index - 1] - 4 * E_prev[index] + E_prev[index + m + 2] +
                                   E_prev[index - (m + 2)]);
}

void simulate(double *E, double *E_prev, double *R,
              const double alpha, const int n, const int m, const double kk,
              const double dt, const double a, const double epsilon,
              const double M1, const double M2, const double b) {

    ghostKernel<<<1, n>>>(E_prev, n, m);
    pdeKernel<<<m, n>>>(E, E_prev, alpha, n, m);
    odeKernel<<<m, n>>>(E, R, n, m, kk, dt, a, epsilon, M1, M2, b);
}
```

Figure 3

Changes in Memory Allocation:

- So far in classes, we have dealt with 1D arrays for Memcpy to device, but in this project our main arrays were allocated in Host by 2D Alloc method, but we couldn't copy these array to our Device's memory, because we think in CUDA Memcpy must do linear memory allocation. Thus we converted our Host arrays to 1D arrays, so in all of our project we work with 2D -> 1D arrays. Because of our one cell padding, we created Device arrays with size (m+2, n+2). Then we used cudaMemcpy to copy them to Device , see Figure 4.

```
cudaMalloc((void **) &d_E, sizeof(double) * (m + 2) * (n + 2));
cudaMalloc((void **) &d_E_prev, sizeof(double) * (m + 2) * (n + 2));
cudaMalloc((void **) &d_R, sizeof(double) * (m + 2) * (n + 2));

cudaMemcpy(d_E, E, sizeof(double) * (m + 2) * (n + 2), cudaMemcpyHostToDevice);
cudaMemcpy(d_E_prev, E_prev, sizeof(double) * (m + 2) * (n + 2), cudaMemcpyHostToDevice);
cudaMemcpy(d_R, R, sizeof(double) * (m + 2) * (n + 2), cudaMemcpyHostToDevice);
```

Figure 4

At last, we copy Device arrays to Host and then free the CUDA memory.

Version 2:

- In version 2, we fused PDE and ODE loops into one loop in the kernel: see Figure 5

```
__global__ void singleKernel(double *E, double *E_prev, double *R, const int n, const int m, const double kk, const double dt, const double a, const double epsilon,
    const double M1, const double M2, const double b, const double alpha) {
    int i = threadIdx.x + 1;
    int j = blockIdx.x + 1;
    int index = j * (n + 2) + i;
    E[index] = E_prev[index] + alpha * (E_prev[index + 1] + E_prev[index - 1] - 4 * E_prev[index] + E_prev[index + m + 2] + E_prev[index - (m + 2)]);
    E[index] = E[index] - dt * (kk * E[index] * (E[index] - a) * (E[index] - 1) + E[index] * R[index]);
    R[index] = R[index] + dt * (epsilon + M1 * R[index] / (E[index] + M2)) * (-R[index] - kk * E[index] * (E[index] - b - 1));
}

void simulate(double *E, double *E_prev, double *R,
            const double alpha, const int n, const int m, const double kk,
            const double dt, const double a, const double epsilon,
            const double M1, const double M2, const double b) {

    ghostKernel<<<1, n>>>(E_prev, n, m);
    singleKernel<<<m, n>>>(E, E_prev, R, n, m, kk, dt, a, epsilon, M1, M2, b, alpha);
}
```

Figure 5

Version 3:

-In version 3, we changed our singleKernel so that the threads would access their corresponding place in array (calculated by thread index), so the threads don't keep referencing the arrays for calculating new values: see Figure 6

```
__global__ void singleKernel(double *E, double *E_prev, double *R,
    const int n, const int m, const double kk,
    const double dt, const double a, const double epsilon,
    const double M1, const double M2, const double b, const double alpha) {
    int i = threadIdx.x + 1;
    int j = blockIdx.x + 1;
    int index = j * (n + 2) + i;
    double E_temp = E[index];
    double R_temp = R[index];
    E_temp = E_prev[index] + alpha * (E_prev[index + 1] + E_prev[index - 1] - 4 * E_prev[index] + E_prev[index + m + 2] + E_prev[index - (m + 2)]);
    E_temp = E_temp - dt * (kk * E_temp * (E_temp - a) * (E_temp - 1) + E_temp * R_temp);
    R_temp = R_temp + dt * (epsilon + M1 * R_temp / (E_temp + M2)) * (-R_temp - kk * E_temp * (E_temp - b - 1));
    E[index] = E_temp;
    R[index] = R_temp;
}
```

Figure 6

Version 4:

- We created a shared 2D memory, like stencil, to collect data from device accordingly to the threadID, then we used the shared variable for obtaining the host arrays. We used synchronized CUDA function to prevent the race condition. Then we call the function in the simulate with numBlock and blockSize, see Figure 7

```
__global__ void singleKernel(double *E, double *E_prev, double *R, const int n, const int m, const double kk,
    const double dt, const double a, const double epsilon,
    const double M1, const double M2, const double b, const double alpha) {

    int x_thread = threadIdx.x, y_thread = threadIdx.y, x_block = blockIdx.x, y_block = blockIdx.y, x_blockDim = blockDim.x, y_blockDim = blockDim.y;
    const int block_size = 16;

    __shared__ double device_memory_array[block_size + 2][block_size + 2];

    if(x_thread == 0) {
        int index = (y_block * y_blockDim * (n + 2)) + (x_block * x_blockDim) + ((y_thread + 1) * (n + 2));
        for (int j = 0; j < x_blockDim + 2; j++) {
            device_memory_array[y_thread + 1][j] = E_prev[index + j];
        }
        if(y_thread == 0) {
            int index = (y_block * y_blockDim * (n + 2)) + (x_block * x_blockDim);
            for (int j = 0; j < x_blockDim + 2; j++) {
                device_memory_array[0][j] = E_prev[index + j];
            }
        }
        if(y_thread == 1) {
            int index = (y_block * y_blockDim * (n + 2)) + (x_block * x_blockDim) + ((y_blockDim + 1) * (n + 2));

            for (int j = 0; j < x_blockDim + 2; j++) {
                device_memory_array[y_blockDim + 1][j] = E_prev[index + j];
            }
        }
    }
    int index = (y_block * y_blockDim * (n + 2)) + (x_block * x_blockDim) + (n + 2) + 1 + (y_thread * (n + 2) + x_thread);

    __syncthreads();
    double E_temp = E[index];
    double R_temp = R[index];

    E_temp = device_memory_array[y_thread + 1][x_thread + 1] + alpha * (
        device_memory_array[y_thread + 1][x_thread + 2] +
        device_memory_array[y_thread + 1][x_thread] - 4 * device_memory_array[y_thread + 1][x_thread + 1] + device_memory_array[y_thread + 2][x_thread + 1] +
        device_memory_array[y_thread][x_thread + 1]);
    E_temp = E_temp - dt * (kk * E_temp * (E_temp - a) * (E_temp - 1) + E_temp * R_temp);
    R_temp = R_temp + dt * (epsilon + M1 * R_temp / (E_temp + M2)) * (-R_temp - kk * E_temp * (E_temp - b - 1));

    __syncthreads();
    E[index] = E_temp;
    R[index] = R_temp;
 }

void simulate(double *E, double *E_prev, double *R,
            const double alpha, const int n, const int m, const double kk,
            const double dt, const double a, const double epsilon,
            const double M1, const double M2, const double b) {

    const dim3 block_size(16,16);
    const dim3 num_blocks(n / block_size.x, n / block_size.y);
    ghostKernel<<<1, n>>>(E_prev, n, m);
    singleKernel<<<num_blocks, block_size>>>(E, E_prev, R, n, m, kk, dt, a, epsilon, M1, M2, b, alpha);
}
```

Figure 7

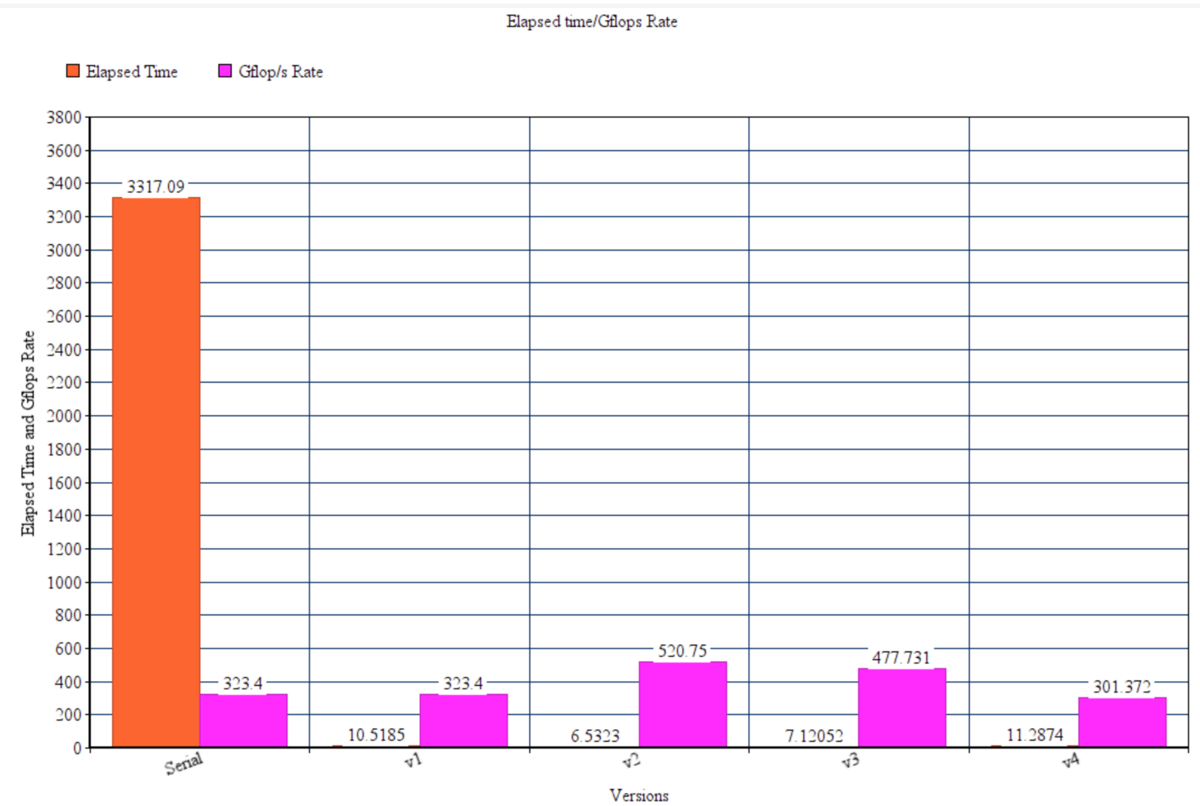
10 December 2021 Friday

Experiments

	Serial	Version 1	Version 2	Version 3	Version 4
Elapsed Time	3317.09	10.5185	6.5323	7.12052	11.2874
GFlop/s Rate	323.4	323.4	520.75	477.731	301.372

Elapsed Time and Gflops Rate comparison:

-We measured Time and Gflop/s rate of our versions without using the plotter. Here are our result: **(Version 4 Block_Size default value is 16)**



Elapsed Time and Gflops Rate graph

- We have obtained close results for v1, v2 , v3, and v4, but serial code was a lot more slower than our versions. Among the versions that have we created, v2 was always faster in every experiment that we have run. we have observed that, in the same version of the program: global memory > temp memory > shared memory in speed wise.

Bandwidth Rate Comparison Experiment:

-We used TESLA V100 GPU for all of our tests.

v100 benchmark results:

```
[CUDA Bandwidth Test] - Starting...
Running on...

 Device 0: Tesla V100-SXM2-32GB
 Quick Mode

 Host to Device Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)        Bandwidth(MB/s)
   33554432                     12108.8

 Device to Host Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)        Bandwidth(MB/s)
   33554432                     12862.3

 Device to Device Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)        Bandwidth(MB/s)
   33554432                     731888.6
```

Our version Bandwidths:

```
serial:
Sustained Bandwidth (GB/sec): 1.17201
```

```
version 1:
Sustained Bandwidth (GB/sec): 369.6
```

```
version 2:
Sustained Bandwidth (GB/sec): 595.143
```

```
version 3:
Sustained Bandwidth (GB/sec): 545.978
```

```
version 4:
Sustained Bandwidth (GB/sec): 344.425
```

- Since v2 is the fastest, we have seen that its Sustained Bandwidth is also very high. As we have anticipated, even the v2 couldn't get too close to benchmark, since additional optimization is required.

v4 Block Size Comparison:

Block size = 2

```
version 4(2x2):
Elapsed Time (sec)          : 45.4588
Sustained Gflops Rate       : 74.8302
```

Block size = 4

```
version 4(4x4):
Elapsed Time (sec)          : 18.5759
Sustained Gflops Rate       : 183.124
```

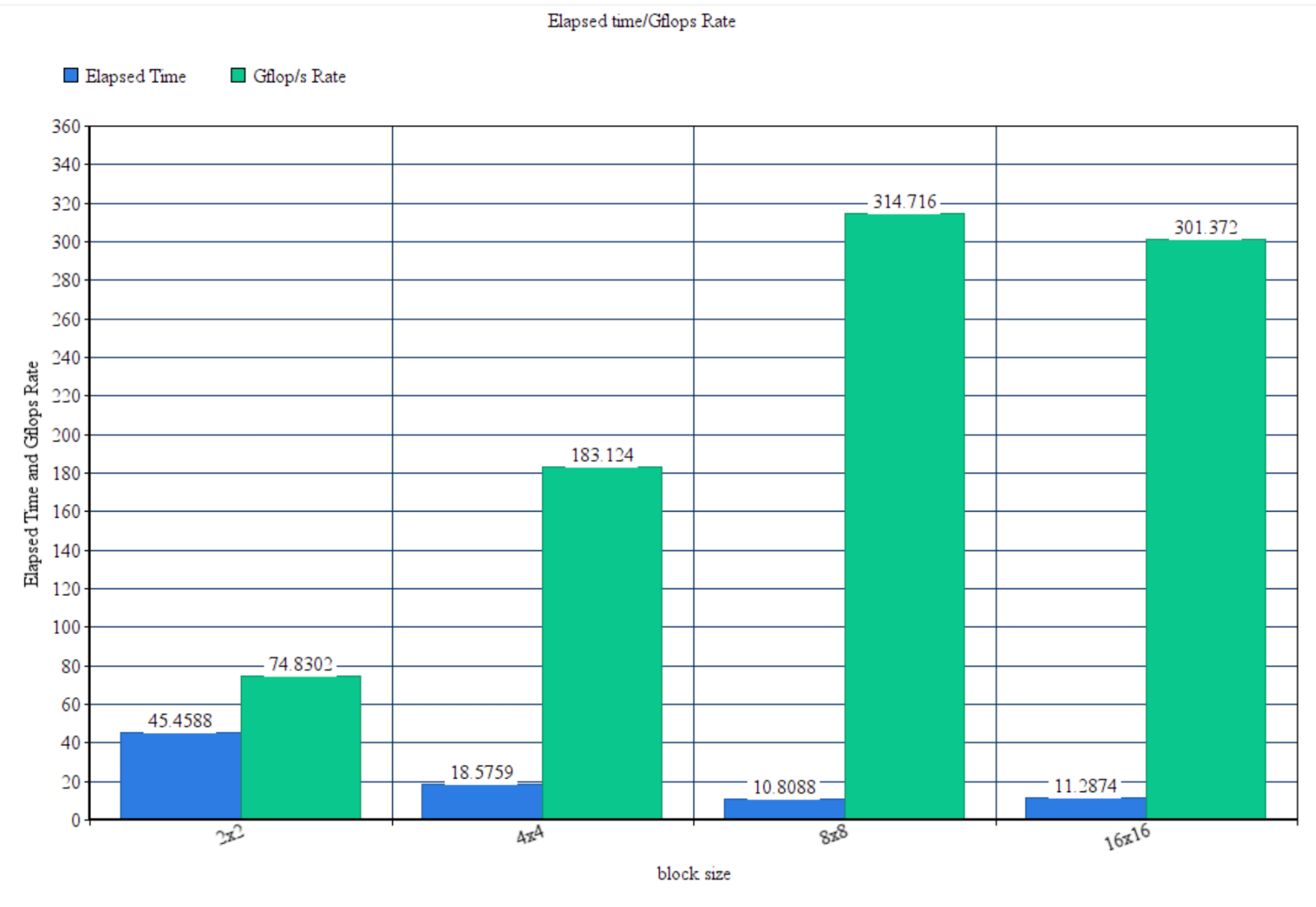
Block size = 8

```
version 4(8x8):
Elapsed Time (sec)          : 10.8088
Sustained Gflops Rate       : 314.716
```

Block size = 16

```
version 4(16x16):
```

```
Elapsed Time (sec)            : 11.2874
Sustained Gflops Rate         : 301.372
```



Elapsed Time and Gflops Rate vs Block size graph

- We observed that the best value is 8 for block size and 16 is very close to it performance wise. As we increase block size, from 2 to 8, we see performance increase because of overall less copy operation, but after 8, overhead becomes too much and performance starts to decrease again.