COMP 429/529: Project 3

Berkay Barlas
54512

Naim Berk Tumer
53847

May 24, 2019

Date Performed: March 17, 2019
Instructor: Didem Unat

In this assignment we implemented CUDA versions on top of given serial version for Cardiac Electrophysiology Simulation and conducted experiments.

To compile all the necessary files run **make**
In this assignment we have completed

- Version 1: A naive that uses global (device) memory
- Version 2: All kernels fused into one kernel
- Version 3: Temporary variable references for R and E arrays
- Version 4: Optimized for CUDA using shared on-chip memory with 2D blocks

1 Implementation

CUDA Implementations

1.1 Version 1

In this version, we parallelized the simulator using single GPU with a naive implementation that makes all the references to global (device) memory.
We created 3 different kernels for ode and pde and ghost cells.

```
__global__ void ghosts(const int n, const int m, double *E_prev) {
    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 ode(const double a, const double kk, const double dt, const int n, const int m, double *E, double *R,
                    const double epsilon,
                    const double M1, const double M2, const double b) {
    /*
     * Solve the ODE, advancing excitation and recovery to the
     *      next timtestep
       */
    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 pde(const int n, const int m, double *E, double *E_prev, 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)]);
}
```

1.2 Version 2

In this version, we fused all the kernels into one kernel by can fusing the ODE and PDE loops into a single loop.

```
__global__ void pde_ode(const double a, const double kk, const double dt, const int n, const int m, double *E, double *E_prev, double *R,
    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));
 }
```

1.3 Version 3

In this version, we used temporary variables to eliminate global memory references for R and E arrays in the ODEs.

```
__global__ void pde_ode(const double a, const double kk, const double dt, const int n, const int m, double *E, double *E_prev, double *R,
  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 temp_E = E[index];
  double temp_R = R[index];

  temp_E = 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)]);
  temp_E = temp_E - dt * (kk * temp_E * (temp_E - a) * (temp_E - 1) + temp_E * temp_R);
  temp_R = temp_R + dt * (epsilon + M1 * temp_R / (temp_E + M2)) * (-temp_R - kk * temp_E * (temp_E - b - 1));

  E[index] = temp_E;
  R[index] = temp_R;
}
```

1.4 Version 4

In this version, we optimised our CUDA implementation by using shared on-chip memory on the GPU by bringing a 2D block into shared memory and sharing it with multiple threads in the same thread block.

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

  int tx = threadIdx.x, ty = threadIdx.y;
  int bx = blockIdx.x, by = blockIdx.y;

  __shared__ double block_E_prev[TILE_DIM + 2][TILE_DIM + 2];
  if(tx == 0) {
      int index = (by * blockDim.y * (n + 2)) + (bx * blockDim.x) +
     ((ty + 1) * (n + 2));
      for (int j = 0; j < blockDim.x + 2; j++) {
          block_E_prev[ty + 1][j] = E_prev[index + j];
      }
      if(ty == 0) {
          int index = (by * blockDim.y * (n + 2)) + (bx * blockDim.
   x);

          for (int j = 0; j < blockDim.x + 2; j++) {
              block_E_prev[0][j] = E_prev[index + j];
          }
      }
      if(ty == 1) {
          int index = (by * blockDim.y * (n + 2)) + (bx * blockDim.
   x) + ((blockDim.y + 1) * (n + 2));
          for (int j = 0; j < blockDim.x + 2; j++) {
              block_E_prev[blockDim.y + 1][j] = E_prev[index + j];
          }
      }
  }
  int index = (by * blockDim.y * (n + 2)) + (bx * blockDim.x) + (n
   + 2) + 1 + (ty * (n + 2) + tx);

  __syncthreads();

  double temp_E = E[index];
  double temp_R = R[index];

  temp_E = block_E_prev[ty + 1][tx + 1] + alpha * (block_E_prev[ty
   + 1][tx + 2] + block_E_prev[ty + 1][tx] - 4 * block_E_prev[ty +
     1][tx + 1] + block_E_prev[ty + 2][tx + 1] + block_E_prev[ty][
   tx + 1]);
  temp_E = temp_E - dt * (kk * temp_E * (temp_E - a) * (temp_E - 1)
     + temp_E * temp_R);
  temp_R = temp_R + dt * (epsilon + M1 * temp_R / (temp_E + M2)) *
   (-temp_R - kk * temp_E * (temp_E - b - 1));
  __syncthreads();
  E[index] = temp_E;
  R[index] = temp_R;
}
```

2 Performance Report

2.1 Performance Study without plotter

We conducted a performance study without the plotter is on. In our results version 2 performed better in terms of execution time and gflops rate. We observed that data transfer time does not affects the performance significantly as we only copy the data at begining and end of execution.

Version 2 performed better because of the required number of operations are minimum while in other version additional operations are needed such as copying data to a local variable and to a shared variable.

Version 4 performed worse than other versions because bringing block to shared memory take a lot of time. Moreover we tried different block sizes and observed version with block size 16 is best.

Device: GeForce GTX 1080 Ti

Serial :

Execution Time (sec) : 570.656

Gflops Rate : 5.96103

Version 1 :

Execution Time (sec) : 21.3603

Gflops Rate : 159.253

Version 2 :

Execution Time (sec) : 17.9579

Gflops Rate : 189.426

Version 3 :

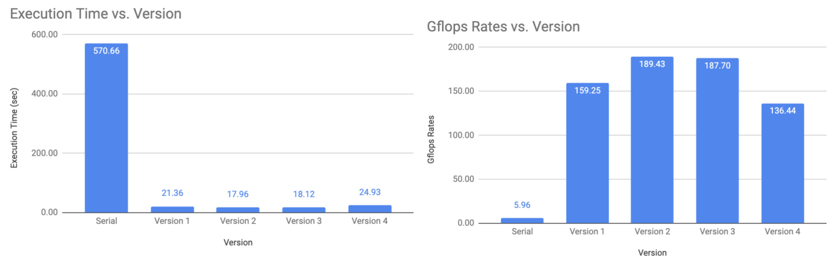
Execution Time (sec) : 18.123

Gflops Rate : 187.701

Version 4 with Block size 16 :

Execution Time (sec) : 24.9325

Gflops Rate : 136.436



(a) Execution times of different versions

(b) Gflops Rate of different versions

Figure 1

2.2 Comparison with Stream Benchmark

Stream Benchmark shows highest bandwidth rate that can be possible on the device. The highest Bandwidth rate we achieved is 216 GB/sec with Version 2 while the maximum device to device Bandwidth is 347 GB/sec.

Benchmark Results

Host to Device Bandwidth (GB/sec): 11.688

Device to Host Bandwidth (GB/sec): 12.723

Device to Device Bandwidth (GB/sec): 347.221

Our Implementation

Serial Version Sustained Bandwidth (GB/sec): 6.8126

Serial Version Sustained Bandwidth (GB/sec): 6.8126

Version 1 Sustained Bandwidth (GB/sec): 182.003

Version 2 Sustained Bandwidth (GB/sec): 216.487

Version 3 Sustained Bandwidth (GB/sec): 214.515

Version 4 with Block Size 64 Sustained Bandwidth (GB/sec): 155.927

2.3 Performance Results with Different Block Sizes

We conducted performance tests of version 4 with 4, 8, 16, 32 and 64 grid sizes. As the grid size increases the number of copy operations decreases. However, after grid size 16 we observed a decrease in performance.

The implementation with block size 16 became fastest one amongs them. The block size with 64 didn't fit to the available thread number in GPU block, therefore, it didn't work.

Version 4 with Block size 4 :

> **Execution Time (sec)** : 45.1086
>
> **Gflops Rate** : 75.4113

Version 4 with Block size 8 :

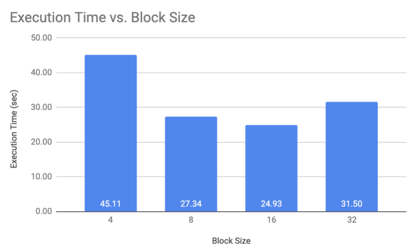
> **Execution Time (sec)** : 27.3411
>
> **Gflops Rate** : 124.417

Version 4 with Block size 16 :

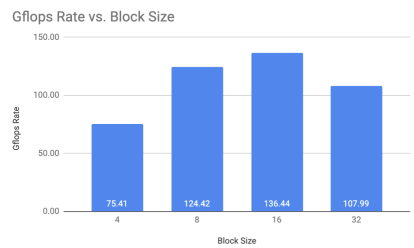
> **Execution Time (sec)** : 24.9325
>
> **Gflops Rate** : 136.436

Version 4 with Block size 32 :

> **Execution Time (sec)** : 31.5012
>
> **Gflops Rate** : 107.986



(a) Execution Times with respect to different block sizes



(b) Gflops Rate results with different block sizes

Figure 2

2.4 Serial vs CUDA

We observed a significant performance improvement with CUDA implementation. Our best CUDA implementation Version 2 is 33 times faster than serial version.

Serial Version :

> **Execution Time (sec)** : 570.656
>
> **Gflops Rate** : 5.96103

CUDA : Version 2

> **Execution Time (sec)** : 17.9579
>
> **Gflops Rate** : 189.426

2.5 MPI vs CUDA

Our CUDA implementation performed much better than our fastest MPI version with 32 thread and 8 x 4 geometry which was implemented in the second assignment.

The difference between two implementations caused by commucation overhead such as updating the ghost cells in MPI version and the total number of threads used in calculations. While there are 32 threads in CPU, there are thousands of threads in GPU.

MPI Version 32 Thread 8 x 4 :

> **Execution Time (sec)** : 448.854
>
> **Gflops Rate** : 7.57862

CUDA : Version 2

> **Execution Time (sec)** : 17.9579
>
> **Gflops Rate** : 189.426