Nate Bowerman and Dhynasah Cakir

CIS 677 – Project 3

Overview

This program implements a simulation of diffusion of a particle across a system over a given number of timesteps. The simulation we did used a single injection of the particle in a one-dimensional array, but it could be easily extended to other situations for simulation.

Sequential Algorithm

A quick overview of the steps of the sequential algorithm is:

1. Input size of grid, number of timesteps, and initial concentration
2. Generate a grid with the given size and concentration of 0 in all cells
3. Set initial concentration of index 0 to the initial concentration
4. For each index, calculate the new concentration using the previous concentration of the index and its neighbors.
5. Repeat step 4 for the number of timesteps
6. Output final array of concentrations

It is straightforward to calculate the next concentration for a non-boundary cell as the average of the cell and its two neighbors. For boundary conditions, the new concentration is calculated as twice the old boundary concentration plus the concentration of the neighbor divided by three. This ensures that the concentration of the entire system stays constant for each timepoint.

The algorithm alternates between two arrays to store the new concentrations at each timestep. This is necessary because the new concentration for index i uses the concentration at index i-1, which is updated in the previous step of the loop. By saving all new concentrations in a different array than the one used to calculate the averages, this is no longer an issue. Whichever density array was updated last is the one that the program outputs.

CUDA Algorithm

Steps 1-3 are the same as in the sequential version. Step 4 and on is where the process diverges.

Calling the kernel

The Kernel is run. When the function simulate() is ran, it is called from the host to run the cu code. The arrays, Density1 and Density2 are 2 of the parameters. The size of the array, the time steps and the initial concentration are also parameters in this function.

Cu file code

In the cu file simulate is designated as extern void. First, space is allocated in the device for 2 arrays, Density1_d and Density2_d with the size of the Density1 and Density2 arrays. In this file, the arrays that are passed through the function when called in the main function are copied to mirror arrays.

Next, the execution configuration is set. dimBlock is set to the block size. Dimgrid is set to the size of the array divided by the block size. Time steps is then incremented in a for loop. For every time step the function where the density is calculated in then updated. At the same time index 0 and n-1 are multiplied by itself and added the concentration of its neighbor and divided.

Once this process is finished the arrays are copied back to the host. And the allocated space is then released.

Comparison between the sequential and CUDA results is done is R (code given in appendix) by summing the absolute value of all pairwise differences between the sequential and CUDA arrays. If both arrays are the same, this sum will be zero.

Visualization

A visualization of the final state of the system is generated using R (code given in appendix). An example plot using an array size of 1028 and 20,000 iterations is given below. The color here indicates the final concentration of each index for the entire array. The color is scaled by the maximum and minimum concentrations. You can see the gradual dispersion of the particle throughout the system. As the iterations approach infinity, the color would become more uniform throughout the system.

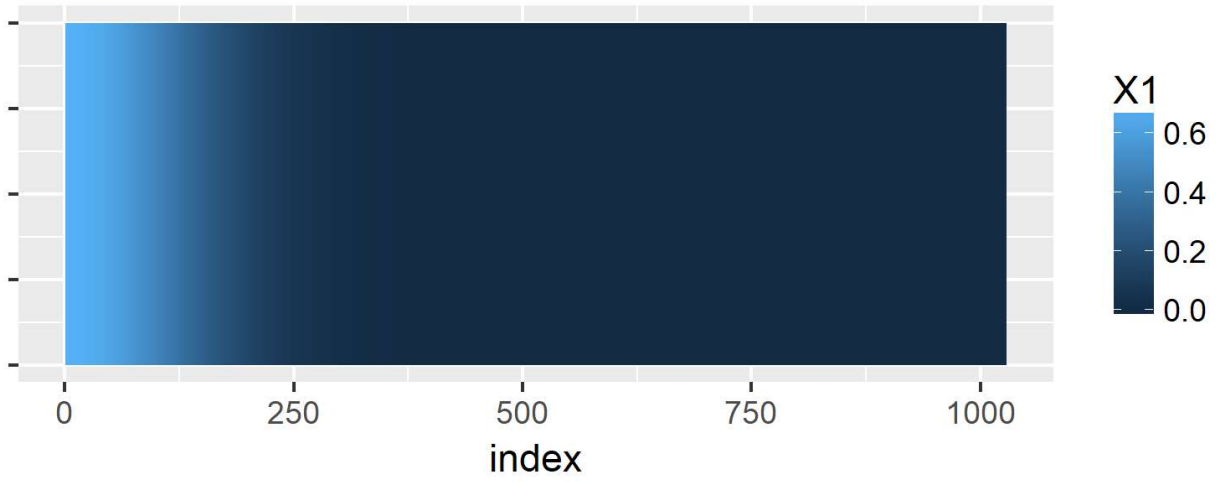

Figure 1: Visualization of final system state

Speedup Comparison

The time in milliseconds for the CUDA algorithm to run for varying block size is plotted below for an array size of 16,384 and 500,000 iterations. The speedup for 8 blocks was about 2.5 and the speedup for 64, 128, and 256 was about 1.32-1.36.

Table 1: Summary of Speedup Analysis

Blocksize	Size	iterations	msec	Speedup
8	16384	500000	220.35	2.494118
64	16384	500000	403.576	1.361773
128	16384	500000	408.733	1.344592
256	16384	500000	417.097	1.317629
seq	16384	500000	549.579	

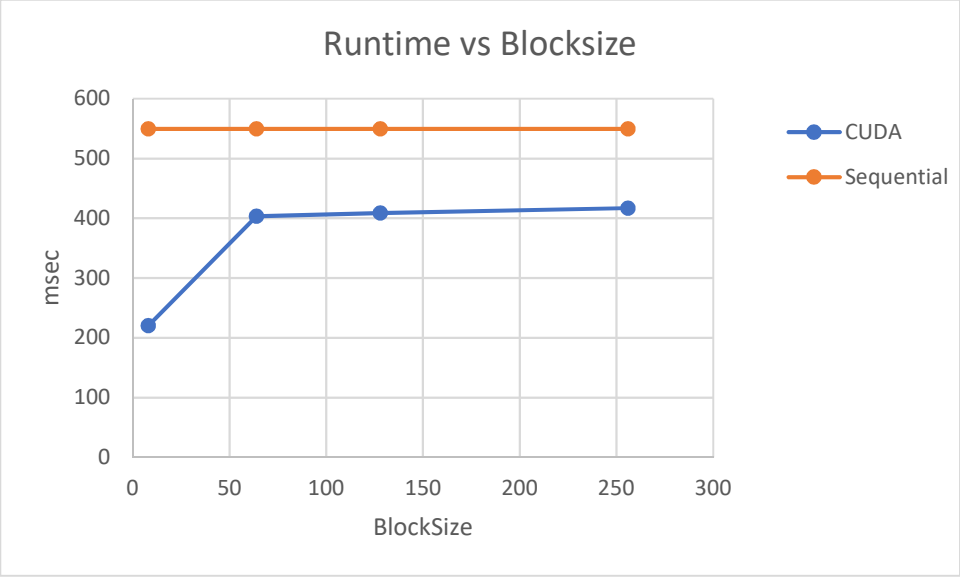

Figure 2: Plot of Speedup Analysis

In this speed-up analysis we noticed that there was not a significant speedup, which was not expected. Even though we increased the block size significantly, the speed remained consistent. We were uncertain why we were getting this outcome. One of the reasons we predicted is the way the memory is allocated on the GPU. Since allocating the space is done manually rather than dynamically it can contribute to the length of time the program runs.

Another reason for our unfavorable outcomes could be that the arrays are created on the CPU first then transferred over to the GPU to do the calculations. If the array was created on the GPU this might increase speed up because as said previously, transferring large data into the GPU is costly and contributes to the time it takes to complete the simulation.

Even with these two factors we should still not see this significant decrease in time or no speed up at all. Which means something else is going extremely wrong and we don't know exactly what it is. We would need to do a more lengthy and analysis of the code to determine the exact problem.

From the graph we can see that the speed was consistent after an increase to 64 block sizes. Whereas the sequential code performed at approximately 550 milliseconds, the parallel code performed around 400 milliseconds consistently.

Appendix

```
library(gplots)
library(ggplot2)
library(RColorBrewer)

#read in arrays
GPU <- read.table(file="Z:\\My Documents\\CIS 677\\Project 3\\ParallelDensityArray3.txt", header=FALSE, sep = " ")
sequential <- read.table(file="Z:\\My Documents\\CIS 677\\Project 3\\DensityArray3.txt", header=FALSE, sep = " ")

#remove blank final cell
GPU <- GPU[1,-dim(GPU)[2]]
sequential <- sequential [1,-dim(sequential)[2]]

# calculate sum of pairwise differences to compare both arrays
diff <- abs(GPU-sequential)
sum <- sum(diff)
sum

dim=dim(sequential)[2]
newGPU <- data.frame(index=1:dim,conc=t(GPU[1,]),y=rep(1,times=dim))

# creates a PNG
png("Z:\\My Documents\\CIS 677\\Project 3\\heatmaps_in_r.png",    # create PNG for the heat map
   width = 5*300,       # 5 x 300 pixels
   height = 2*300,
   res = 300,           # 300 pixels per inch
   pointsize = 8)       # smaller font size

# remove background and axis from plot
theme_change <- theme(
  axis.text.y = element_blank(),
  axis.title.y = element_blank()
)

ggplot(newGPU,aes(x=index,y=y,fill=X1))+geom_tile()+theme_change

dev.off()
```

Printout: project 3.c

Wednesday, November 7, 2018 3:29 PM

```
/*
*  GPU- accelerated modeling of point source pollution
*
*/

#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
//#define BLOCK_SIZE 256

extern void simulate(double*, double*, int, int);

int main ( int argc, char *argv[])
{
      // declare arrays, and input variables
      // declare arrays, and input variables
      int SIZEOFARRAY;
      int TimeSteps;
      int InitialConc;

      if(argc != 4){
            printf(" enter : SIZEOFARRAY, TimeSteps,
InitialConc");
            return 1;
      }


      SIZEOFARRAY = atoi(argv[1]);
      TimeSteps = atoi(argv[2]);
      InitialConc = atoi(argv[3]);


      if(SIZEOFARRAY < 1 || TimeSteps < 1 || InitialConc < 1) {
            printf("enter only positive numbers greater than 0 for
arguments");
            return 1;
      }



      double Density1[SIZEOFARRAY];
      double Density2[SIZEOFARRAY];

      // intialize arrays
      for (int i = 0; i < SIZEOFARRAY; i++) {
            Density1[i]= 0.0;
            Density2[i] = 0.0;
      }
      Density1[0]= InitialConc;
      Density2[0] = InitialConc;

      struct timeval start;
```

```
    gettimeofday(&start, NULL);

    simulate(Density1,Density2,SIZEOFARRAY,TimeSteps);

    struct timeval finish;
    gettimeofday(&finish, NULL);

    FILE *outfile;
    outfile = fopen("ParallelDensityArray.txt", "w");

   if (TimeSteps%2 == 0) {
       for( int i =0; i < SIZEOFARRAY; i++){
           fprintf(outfile, "%f ", Density1[i]);
       }
  } else {
       for( int i =0; i < SIZEOFARRAY; i++){
           fprintf(outfile, "%f ", Density2[i]);
       }
   }
    fclose(outfile);
    printf("time %lu\n", finish.tv_usec - start.tv_usec);
    return 0;
}
```

Printout : project 3. cu

Wednesday, November 7, 2018 3:28 PM

```
/*
 * project_3.cu
 * includes setup function called from driver program
 * includes kernel function 'cu_claculateDiffusion()'
 */

#include <stdio.h>
#include <stdlib.h>


#define BLOCK_SIZE 256



__global__ void updateDensity(double *newDensity, double
*oldDensity, int SIZEOFARRAY){
      int index;
      index = blockIdx.x *BLOCK_SIZE + threadIdx.x;
      if(index == 0){
            newDensity[index] = (oldDensity[index]*2
+oldDensity[index+1])/3;
      } else if(index == SIZEOFARRAY) {
            newDensity[index] =
(oldDensity[index-1]+oldDensity[index]*2)/3;
      } else {
            newDensity[index] =
(oldDensity[index-1]+oldDensity[index]+oldDensity[index+1])/3;
      }

}


extern "C" void simulate(double *Density1, double* Density2, int
SIZEOFARRAY, int TimeSteps)
{
      double *Density1_d;
      double *Density2_d;
      cudaError_t result;

      //allocate space in the device
      result = cudaMalloc ((void**) &Density1_d, sizeof(double) *
SIZEOFARRAY);
      if (result != cudaSuccess) {
            fprintf(stderr, "cudaMalloc (Density1) failed.");
            exit(1);
      }
      result = cudaMalloc ((void**) &Density2_d, sizeof(double) *
SIZEOFARRAY);
      if (result != cudaSuccess) {
            fprintf(stderr, "cudaMalloc (Density2) failed.");
            exit(1);
      }
```

```
      //copy the arrays from host to the device
      result = cudaMemcpy (Density1_d, Density1 , sizeof(double) *
SIZEOFARRAY, cudaMemcpyHostToDevice);
      if (result != cudaSuccess) {
            fprintf(stderr, "cudaMemcpy host->dev (Density1)
failed.");
            exit(1);
      }
      result = cudaMemcpy (Density2_d, Density2, sizeof(double) *
SIZEOFARRAY, cudaMemcpyHostToDevice);
      if (result != cudaSuccess) {
            fprintf(stderr, "cudaMemcpy host->dev (Density2)
failed.");
            exit(1);
      }

      //set exectuion configuration
      dim3 dimblock (BLOCK_SIZE);
      dim3 dimgrid (SIZEOFARRAY/BLOCK_SIZE);
      //function that calls the GPU
       int i;
      for (i=1; i<= TimeSteps; i++){
            if (i%2 == 0) {
             updateDensity<<<dimgrid,dimblock>>>(Density1_d,
Density2_d, SIZEOFARRAY);

        }
      else {
               updateDensity<<<dimgrid,dimblock>>>(Density2_d,
Density1_d, SIZEOFARRAY);
        }

      }


      result = cudaMemcpy (Density1, Density1_d, sizeof(double) *
SIZEOFARRAY, cudaMemcpyDeviceToHost);
      if (result != cudaSuccess) {
            fprintf(stderr, "cudaMemcpy host <- dev (Density1)
failed.");
            exit(1);
      }
      result = cudaMemcpy (Density2, Density2_d, sizeof(double) *
SIZEOFARRAY, cudaMemcpyDeviceToHost);
      if (result != cudaSuccess) {
            fprintf(stderr, "cudaMemcpy host <- dev (Density2)
failed.");
            exit(1);
      }

      // release the memory on the GPU
```

```
result = cudaFree (Density1_d);
if (result != cudaSuccess) {
     fprintf(stderr, "cudaFree (Density1) failed.");
     exit(1);
}
result = cudaFree (Density2_d);
if (result != cudaSuccess) {
     fprintf(stderr, "cudaFree (Density2) failed.");
     exit(1);
}
}
```

Printout project 3 seq. c

Wednesday, November 7, 2018 3:30 PM

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>


void updateDensity(double *newDensity, double *oldDensity, int
SIZEOFARRAY,int Conc ){
    newDensity[0] = (oldDensity[0]*2+oldDensity[1])/3;
    for(int i = 1; i < (SIZEOFARRAY-1); i++ ) {
        newDensity[i] =
(oldDensity[i-1]+oldDensity[i]+oldDensity[i+1])/3;
    }

newDensity[SIZEOFARRAY]=(oldDensity[SIZEOFARRAY-1]+oldDensity[SIZ
EOFARRAY]*2)/3;
}

int main (int argc, char *argv[])
{
     struct timeval start;
     gettimeofday(&start, NULL);
     int SIZEOFARRAY;
     int TimeSteps;
     int InitialConc;

     if(argc != 4){
          printf(" enter : SIZEOFARRAY, TimeSteps,
InitialConc");
          return 1;
     }


     SIZEOFARRAY = atoi(argv[1]);
     TimeSteps = atoi(argv[2]);
     InitialConc = atoi(argv[3]);


     if(SIZEOFARRAY < 1 || TimeSteps < 1 || InitialConc < 1) {
          printf("enter only positive numbers greater than 0 for
arguments");
          return 1;
     }

        double Density1[SIZEOFARRAY];
     double Density2[SIZEOFARRAY];

     for (int i = 1; i< SIZEOFARRAY; i++) {
          Density1[i]= 0.0;
          Density2[i] = 0.0;
     }
    Density1[0]=InitialConc;
    Density2[0]=InitialConc;
```

```
    for (int i=1; i<= TimeSteps; i++){
            if (i%2 == 0) {
                updateDensity(Density1,Density2,SIZEOFARRAY,
InitialConc );
            } else {

updateDensity(Density2,Density1,SIZEOFARRAY,InitialConc);
            }
    }
     struct timeval finish;
     gettimeofday(&finish, NULL);

     FILE *outfile;
     outfile = fopen("DensityArray.txt", "w");

    if (TimeSteps%2 == 0) {
        for( int i =0; i < SIZEOFARRAY; i++){
            fprintf(outfile, "%f ", Density1[i]);
        }
   } else {
        for( int i =0; i < SIZEOFARRAY; i++){
            fprintf(outfile, "%f ", Density2[i]);
        }
    }
     fclose(outfile);
     printf("time %lu\n", finish.tv_usec - start.tv_usec);
     return 0;
}
```