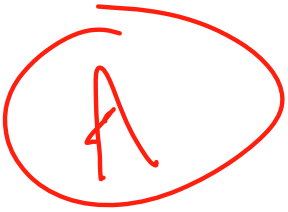
Accelerated Modeling of Laplacian Diffusion via GPU
Programming Assignment #3
CS677 High Performance Computing
November 4, 2020
Andrew Weston



Abstract: Diffusion is the movement of particles across a concentration gradient; it continues until an equilibrium has been reached. We present and analyze accelerated models of Laplacian temperature diffusion using the Compute Unified Device Architecture application programming interface. We demonstrate that general-purpose graphics processing unit programming with CUDA can achieve a 5.5x speedup of the model's computation.

1. Introduction

1.1 Physics Background

1.1.A) Diffusion

Diffusion is the movement of particles across a concentration gradient; it continues until a temperature (or pressure, or density) equilibrium has been reached. The diffusion equation is modeled via a partial differential equation (PDE) that describes density dynamics. If D, the diffusion coefficient (a term describing the rate of diffusion) varies depending on the density, then the equation is nonlinear. However, if the rate of diffusion is constant, then the diffusion equation reduces to a linear equation – the well-known heat distribution equation.

1.1.B) Heat Equation (1-Dimensional)

The heat distribution equation models temperature changes over time. Consider an insulated, conducting 1-meter metal rod (see Figure 1 below). The rod is initially at room temperature (23 C). At time t=0, one end of the rod is placed in a drum of boiling water (100 C). Over time, heat will be transferred from regions of higher temperature (the left end) to regions of lower temperature.

Figure 1

Assume that the density of the rod and its thermal conductivity are constant. Now consider an infinitely thin slice x (width = Δx) of the rod, such that its temperature u is uniform within that slice. Since the rod is insulated, the change in temperature at location x over time Δt is equal to the "heat in" from its left boundary minus the "heat out" at its right boundary. This is

modeled by the partial differential equation:

$$\frac{\partial u}{\partial t} = K\frac{\partial^2 u}{\partial x^2},$$

where K is the diffusivity constant. In this way we can compute the temperature u at point x at time t. Moreover, with judicious choice of the time scale T and the length of the rod L, the diffusivity constant vanishes.

Next, the Central Difference Theorem can be used to derive an approximate numerical solution to the second-order derivative:

$$\frac{\partial^2 u}{\partial x^2} = \frac{dt}{dx * dx}[u(x - dx) - 2u(x) + u(x + dx)]$$

Rearranging and expressing as a discrete equation gives:

$$u_{t+1}[i] = \frac{u_t[i-1] + u_t[i+1]}{2},$$

where i is the index of point x. This finite difference equation says: the temperature at point x in the next time step (t+1) is equal to the average temperature of its two neighbors in the current time step. Computed over very small differences in time (t) and very small distances along the cylinder (x), this method converges to provide an arbitrarily accurate approximation to the heat equation.

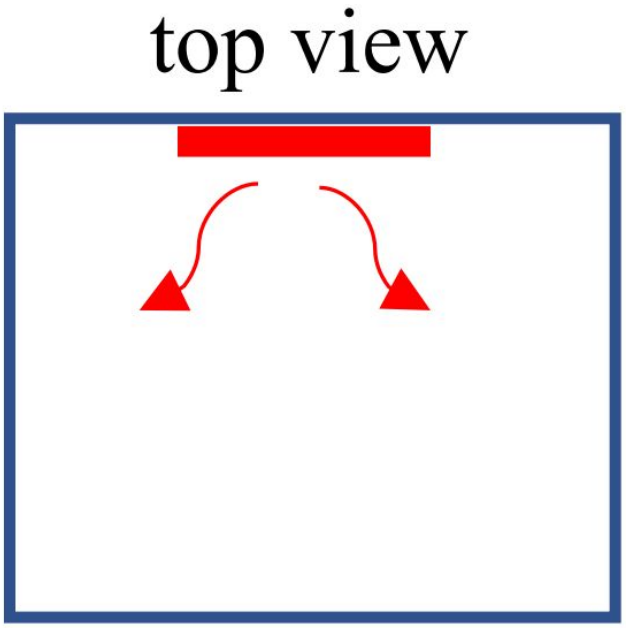
As with many numerical methods, the algorithm runs to convergence. This can be determined via a global notion of stable temperature (especially given a fixed temperature heat source), or when the temperature change over two units of time at all points is less than some small value ε. As noted, it is often used to determine the temperature at a specified point at a specified time.

1.1.C) Heat Equation (Multi-Dimensional)

This basic idea can be extended to two dimensions, assuming the simplifying use of constants as described above. Consider a room with insulated walls and an electric baseboard heater as its single heat source (see Figure 2). We strive to calculate and model the temperature distribution as heat is radiated over time (see Figure 3).

Figure 2

Figure 3



This constitutes a 2-dimensional gradient, represented by the Laplace equation:

$$\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} = 0,$$

which similarly reduces to a finite-difference equation in which the temperature of a 2-D point at the next time step is equal to the average temperature of its four neighbors at the current time. Once the basic model is implemented realistic conditions can be added to improve the accuracy of the simulation. For example, structures can be placed in the room, or an open

window can be used to introduce airflow. The model could also be extended to three dimensions, as in the case of true volumetric objects such as the atmosphere or the ocean (i.e. weather modeling).

1.2 CUDA Background

The Compute Unified Device Architecture (CUDA) is a parallel computing Application Programming Interface (API) that allows software developers to leverage a CUDA-capable graphics processing unit (GPU) for general-purpose applications. CUDA utilizes a GPU's highly parallel processing elements to concurrently perform a single instruction on many sets of data.

1.3 Accelerated Modeling of Laplacian Diffusion with CUDA

One characteristic of the finite difference solution to the diffusion problem explained in Section 1.1 is that calculation of an individual "slice" is independent of all other slices. Thus, a parallel algorithm can exploit this fact and compute many slices concurrently. At a given time step, calculations may be performed in any order and still be correct, because there are only local data dependencies (assuming old/new temperature values remain isolated).

2. Solutions

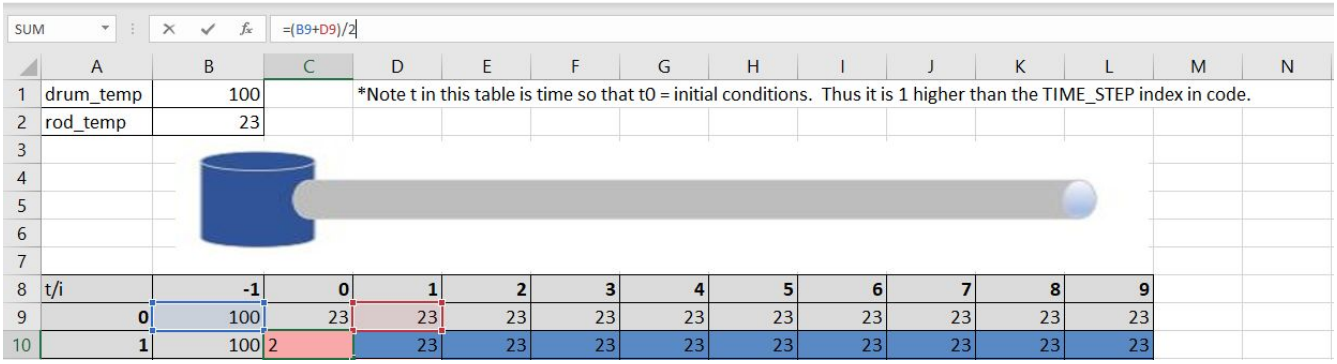
First, we discuss implementations of the 1-D model of diffusion through a rod as outlined in Section 1.1.B). Then we explain implementations for the 2-D model of diffusion through a room as outlined in Section 1.1.C). For both applications we begin by explaining the diffusion model conceptually, introduce a sequential implementation, consider a parallel solution using CUDA, and finally explain our validation strategy.

2.1 Rod

2.1.A) Rod Model

Figure 4 shows a Microsoft Excel model for one timestep of diffusion computation for a coarse-grained rod with only ten slices. The highlighted cells' coloration illustrates the computation's data dependencies. Calculating the first slice (at index 0 and time 1) is performed in cell C10 (shown bordered in green). Cell C10 is computed by averaging the temperatures of its adjacent cells in the previous timestep (shown bordered in blue and red).

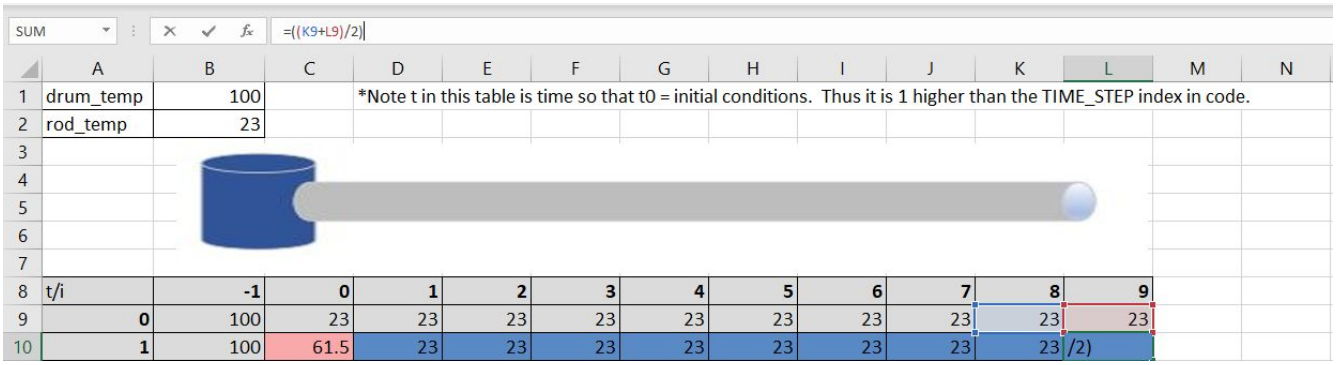
Figure 4



	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	drum_temp	100		*Note t in this table is time so that t0 = initial conditions. Thus it is 1 higher than the TIME_STEP index in code.										
2	rod_temp	23												
3														
4														
5														
6														
7														
8	t/i	-1	0	1	2	3	4	5	6	7	8	9		
9	0	100	23	23	23	23	23	23	23	23	23	23		
10	1	100	2	23	23	23	23	23	23	23	23	23		

This relationship is duplicated for every slice with the exception of the slice furthest from the boiling water. The furthest slice has no “right neighbor”. Instead, we average it’s left neighbor and itself in the previous timestep (see Figure 5).

Figure 5



	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	drum_temp	100		*Note t in this table is time so that t0 = initial conditions. Thus it is 1 higher than the TIME_STEP index in code.										
2	rod_temp	23												
3														
4														
5														
6														
7														
8	t/i	-1	0	1	2	3	4	5	6	7	8	9		
9	0	100	23	23	23	23	23	23	23	23	23	23		
10	1	100	61.5	23	23	23	23	23	23	23	23	/2)		

See Appendix A for the complete Microsoft Excel model of diffusion in a rod with 10 slices over 20 timesteps. Similarly, See Appendix B for a summary of the Microsoft Excel model of diffusion in a rod with 100 slices over 1000 timesteps.

2.1.B) Rod Sequential Implementation

Our sequential implementation of the rod model accepts parameters for the number of slices, number of timesteps, and the index of the slice of the rod to be sampled. The program makes space for two copies of the rod’s slices. Then, using nested loops for the number of timesteps and the number of slices, the program computes each slice’s “neighborhood average” as explained in Section 2.1.A). We use the two copies of array slices as a circular array. At the beginning of each timestep, the copy written in the last timestep is only read in order to account for data dependencies. Computations for “this” timestep are then written to the other rod copy overwriting the state of $timestep - 2$ which are no longer required because $timestep - 1$ has been computed. See Appendix C - Rod/Room Sequential Implementation for full details.

2.1.C) Rod Parallel Implementation

Like the sequential implementation, our parallel implementation of the rod model accepts parameters for the number of slices, number of timesteps, and the index of the slice of the rod to be sampled. However, in this implementation we model the two copies of the rod with $ArraySize = (Slices * Copies) + Copies$ space. The two additional slices give us space to initialize the bucket temperature in the first two positions of each rod copy. This approach simplifies the kernel algorithm to computing slices with only two cases: 1) First/Last slice 2) All other slices(the common case).

Next, our host helper function calc_rod_diffusion Line 96 Appendix D - Rod CUDA Implementation sets the block and grid size. Notably, our program is a preliminary parallel implementation. We simply select a block size of 32 based on the default warp size. Likewise, we choose a grid size by simply dividing the space used to store the rod models by the block size (and rounding up to ensure the last block is properly computed) see Line 116.

The host calls two helper functions to execute the diffusion computation. First the host calls cu_initialize_rod to set the rod state at timestep 0. Then for each timestep, the host invokes a cu_rod_diffusion_step kernel that computes a single timestep and updates the state of one of the rod copies. Finally, the device copies the temperature at the specified index after the specified number of timesteps back to the host. Appendix D also includes sample code (commented out) that demonstrates how the entire rod state could be copied between host and device for development, debugging, and alternative applications.

2.1.D) Rod Solution Validation

First, we developed the simple rod model explained in Section 2.1.A) using Microsoft Excel. Excel allowed us to easily visualize the data dependencies of the problem, create a simple test bench, and visualize that our model matched our intuition of diffusion occurring from left to right as time progressed. After we developed the Excel model we wrote the sequential implementation in Appendix C. Appendix E shows sample output of the sequential program calculating a rod with ten slices for 20 timesteps that matches (with rounding) the Excel model. We performed a similar validation with a rod with 100 slices for 1000 timesteps but its output is not shown for brevity.

Instead, Appendices F and G show the numpy and plotly visualization of sample output for both test cases of rod parameters. Sample code for generating these visualizations is included in Appendix H. Once we developed our Excel model, manually validated it, and confirmed our sequential implementation matched, we then verified our CUDA implementation by checking its output against the output of the sequential program during development. A simple BASH script demonstrating this validation is available in Appendix I.

2.2 Room

2.2.A) Room Model

When we extend the diffusion model to 2-D as explained in Section 1.1.C) we refer to individual elements as “tiles” instead of “slices”. Figure 6 shows a Microsoft Excel model for one timestep of diffusion computation for a coarse-grained 10x10 room with 100 tiles.

Figure 6

Time Step	0									
i/j	0	1	2	3	4	5	6	7	8	9
0	23	23	23	100	100	100	23	23	23	23
1	23	23	23	23	23	23	23	23	23	23
2	23	23	23	23	23	23	23	23	23	23
3	23	23	23	23	23	23	23	23	23	23
4	23	23	23	23	23	23	23	23	23	23
5	23	23	23	23	23	23	23	23	23	23
6	23	23	23	23	23	23	23	23	23	23
7	23	23	23	23	23	23	23	23	23	23
8	23	23	23	23	23	23	23	23	23	23
9	23	23	23	23	23	23	23	23	23	23
Time Step	1									
i/j	0	1	2	3	4	5	6	7	8	9
0	23	23	42.25	100	100	100	42.25	23	23	23
1	23	23	23	42.25	42.25	42.25	23	23	23	23
2	23	23	23	23	23	23	23	23	23	23
3	23	23	23	23	23	23	23	23	23	23
4	23	23	23	23	23	23	23	23	23	23
5	23	23	23	23	23	23	23	23	23	23
6	23	23	23	23	23	23	23	23	23	23
7	23	23	23	23	23	23	23	23	23	23
8	23	23	23	23	23	23	23	23	23	23
9	23	23	23	23	23	23	23	23	23	23

Heater

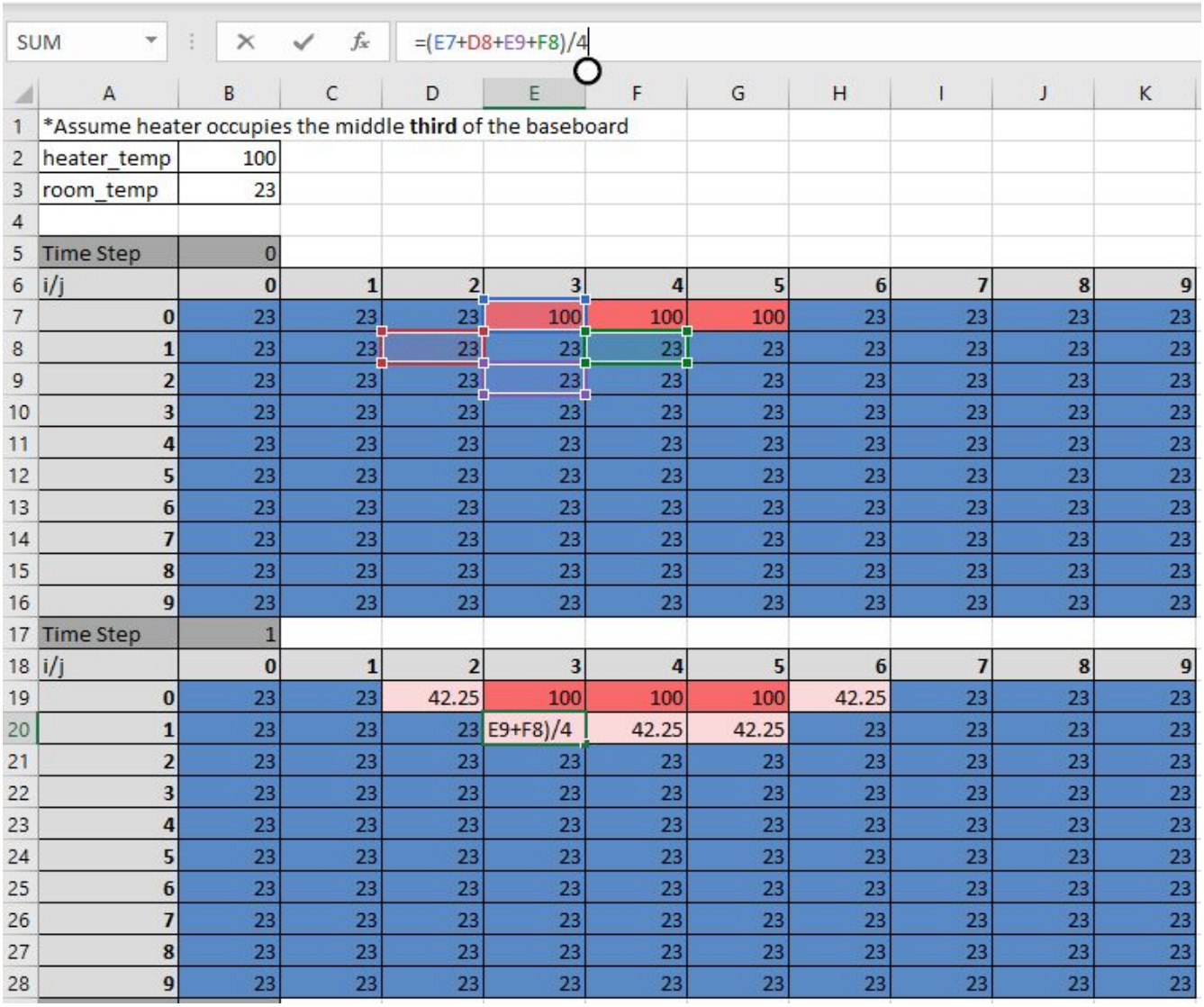
In every room model we simply assume the baseboard heater occupies the middle third of the wall and is only one tile deep.

Internal Tiles

Extending the model to 2-D requires us to consider a tile’s north and south neighbors (in addition to its east and west neighbors as in the rod model) in the common case. Figure 7 shows the computation of cell E20 in timestep one is an average operation of the cell’s north, south, east, and west neighbors in the previous timestep (shown in blue, purple, green, and

red respectively).

Figure 7

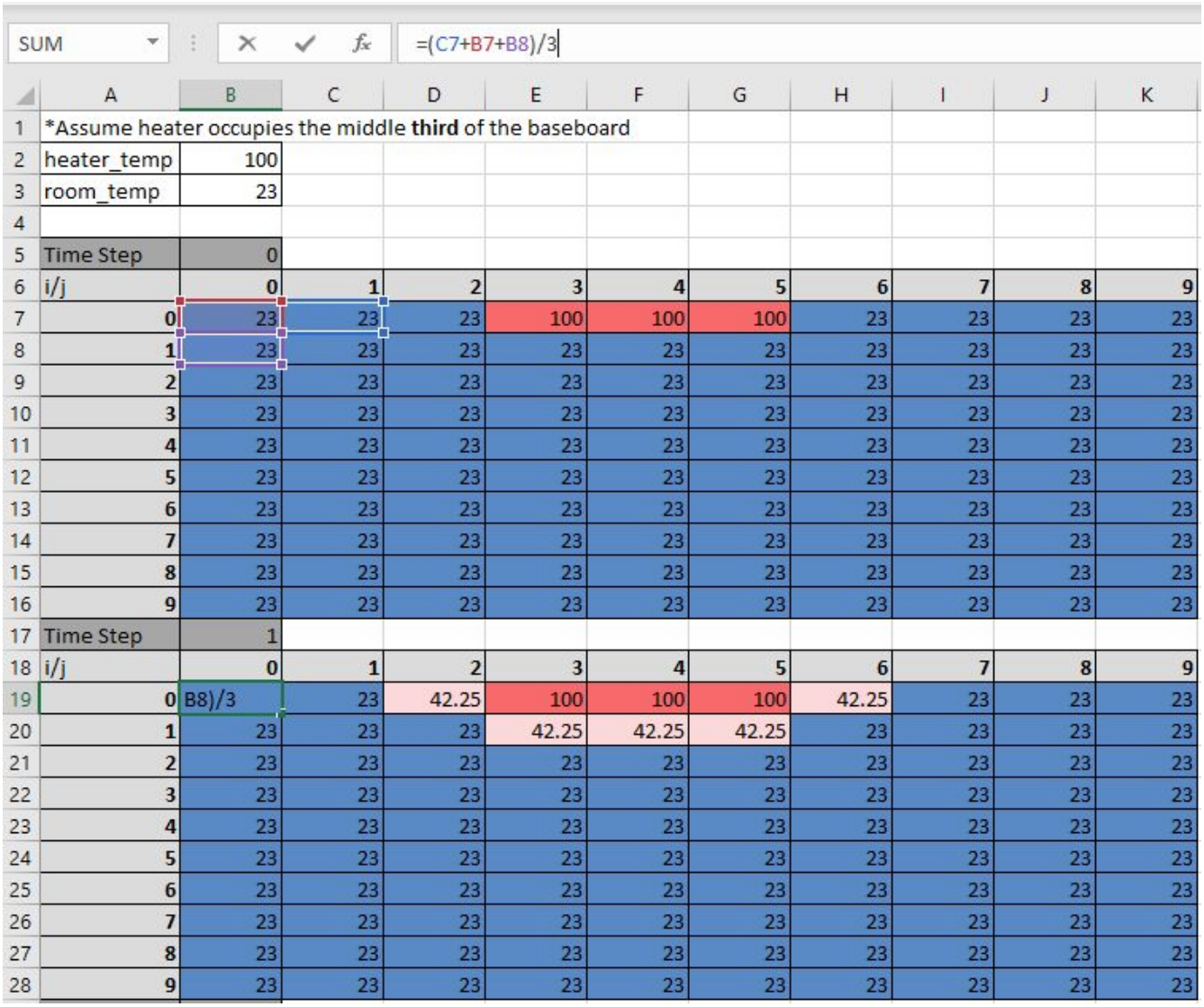


	A	B	C	D	E	F	G	H	I	J	K
1	*Assume heater occupies the middle **third** of the baseboard										
2	heater_temp	100									
3	room_temp	23									
4											
5	Time Step	0									
6	i/j	0	1	2	3	4	5	6	7	8	9
7	0	23	23	23	100	100	100	23	23	23	23
8	1	23	23	23	23	23	23	23	23	23	23
9	2	23	23	23	23	23	23	23	23	23	23
10	3	23	23	23	23	23	23	23	23	23	23
11	4	23	23	23	23	23	23	23	23	23	23
12	5	23	23	23	23	23	23	23	23	23	23
13	6	23	23	23	23	23	23	23	23	23	23
14	7	23	23	23	23	23	23	23	23	23	23
15	8	23	23	23	23	23	23	23	23	23	23
16	9	23	23	23	23	23	23	23	23	23	23
17	Time Step	1									
18	i/j	0	1	2	3	4	5	6	7	8	9
19	0	23	23	42.25	100	100	100	42.25	23	23	23
20	1	23	23	23	E9+F8)/4	42.25	42.25	23	23	23	23
21	2	23	23	23	23	23	23	23	23	23	23
22	3	23	23	23	23	23	23	23	23	23	23
23	4	23	23	23	23	23	23	23	23	23	23
24	5	23	23	23	23	23	23	23	23	23	23
25	6	23	23	23	23	23	23	23	23	23	23
26	7	23	23	23	23	23	23	23	23	23	23
27	8	23	23	23	23	23	23	23	23	23	23
28	9	23	23	23	23	23	23	23	23	23	23

Corner Tiles

As with the rod model, we perform a special average operation for all border elements where any of the tile's neighbors are missing. For corner cases, we take the average of the corner and its two existing neighbors from the previous timestep. For example, Figure 8 shows that the northwest corner of the room is the average of itself and its east and south neighbors from the previous timestep.

Figure 8



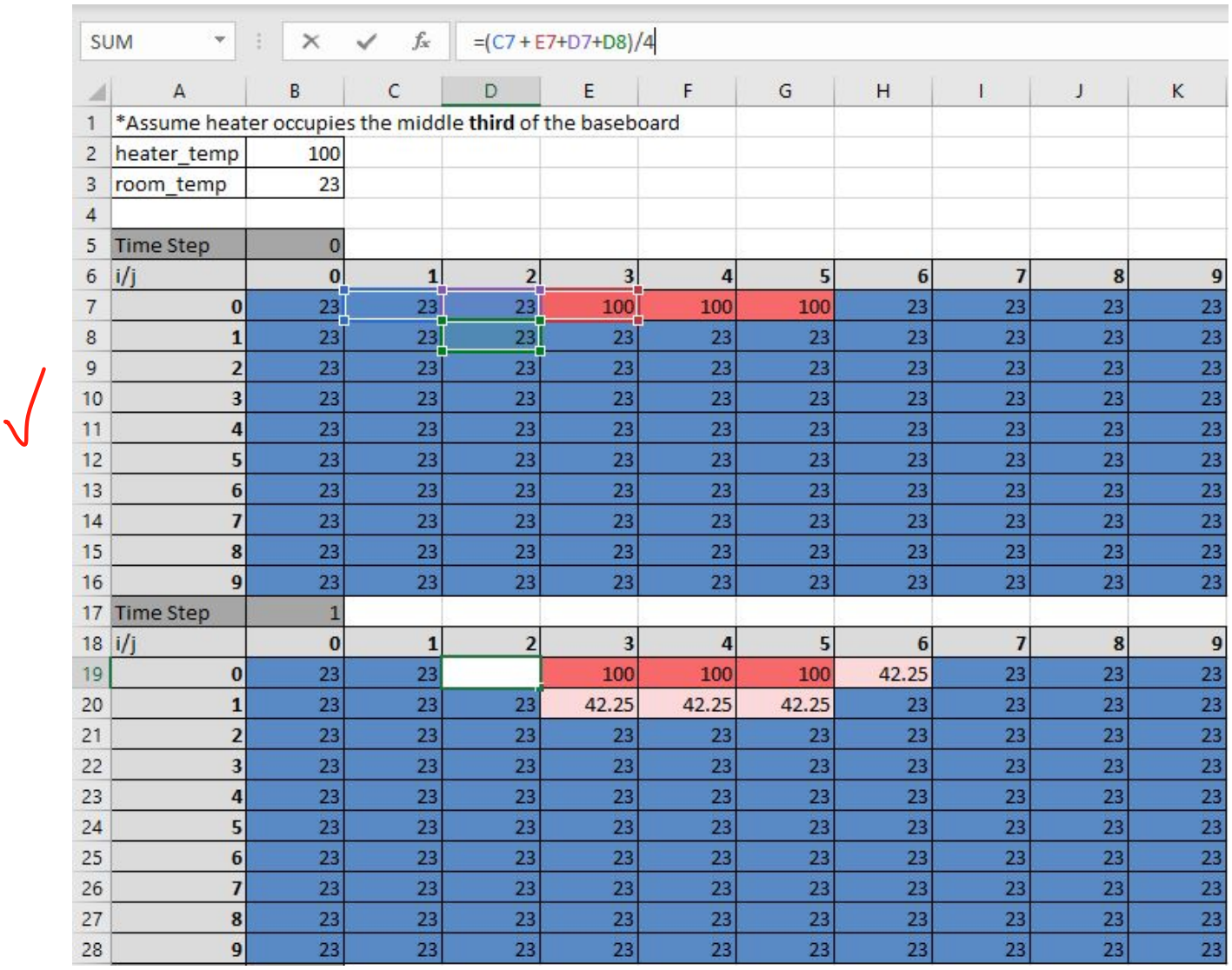
SUM | =(C7+B7+B8)/3

	A	B	C	D	E	F	G	H	I	J	K
1	*Assume heater occupies the middle **third** of the baseboard										
2	heater_temp	100									
3	room_temp	23									
4											
5	Time Step	0									
6	i/j	0	1	2	3	4	5	6	7	8	9
7	0	23	23	23	100	100	100	23	23	23	23
8	1	23	23	23	23	23	23	23	23	23	23
9	2	23	23	23	23	23	23	23	23	23	23
10	3	23	23	23	23	23	23	23	23	23	23
11	4	23	23	23	23	23	23	23	23	23	23
12	5	23	23	23	23	23	23	23	23	23	23
13	6	23	23	23	23	23	23	23	23	23	23
14	7	23	23	23	23	23	23	23	23	23	23
15	8	23	23	23	23	23	23	23	23	23	23
16	9	23	23	23	23	23	23	23	23	23	23
17	Time Step	1									
18	i/j	0	1	2	3	4	5	6	7	8	9
19	0	B8)/3	23	42.25	100	100	100	42.25	23	23	23
20	1	23	23	23	42.25	42.25	42.25	23	23	23	23
21	2	23	23	23	23	23	23	23	23	23	23
22	3	23	23	23	23	23	23	23	23	23	23
23	4	23	23	23	23	23	23	23	23	23	23
24	5	23	23	23	23	23	23	23	23	23	23
25	6	23	23	23	23	23	23	23	23	23	23
26	7	23	23	23	23	23	23	23	23	23	23
27	8	23	23	23	23	23	23	23	23	23	23
28	9	23	23	23	23	23	23	23	23	23	23

Border Tiles

Tiles on the outside of the room that are not corners are computed similarly to the corner case. Border tiles of the average of their neighbors and themself in the previous timestep. Figure 9 shows one such border tile's data dependency.

Figure 9



SUM | =(C7 + E7+D7+D8)/4

	A	B	C	D	E	F	G	H	I	J	K
1	*Assume heater occupies the middle **third** of the baseboard										
2	heater_temp	100									
3	room_temp	23									
4											
5	Time Step	0									
6	i/j	0	1	2	3	4	5	6	7	8	9
7	0	23	23	23	100	100	100	23	23	23	23
8	1	23	23	23	23	23	23	23	23	23	23
9	2	23	23	23	23	23	23	23	23	23	23
10	3	23	23	23	23	23	23	23	23	23	23
11	4	23	23	23	23	23	23	23	23	23	23
12	5	23	23	23	23	23	23	23	23	23	23
13	6	23	23	23	23	23	23	23	23	23	23
14	7	23	23	23	23	23	23	23	23	23	23
15	8	23	23	23	23	23	23	23	23	23	23
16	9	23	23	23	23	23	23	23	23	23	23
17	Time Step	1									
18	i/j	0	1	2	3	4	5	6	7	8	9
19	0	23	23		100	100	100	42.25	23	23	23
20	1	23	23	23	42.25	42.25	42.25	23	23	23	23
21	2	23	23	23	23	23	23	23	23	23	23
22	3	23	23	23	23	23	23	23	23	23	23
23	4	23	23	23	23	23	23	23	23	23	23
24	5	23	23	23	23	23	23	23	23	23	23
25	6	23	23	23	23	23	23	23	23	23	23
26	7	23	23	23	23	23	23	23	23	23	23
27	8	23	23	23	23	23	23	23	23	23	23
28	9	23	23	23	23	23	23	23	23	23	23

2.2.B) Room Sequential Implementation

The sequential room implementation is substantially similar to the sequential rod implementation in Section 2.1.B). The program reads in parameters for the number of tiles, number of timesteps, and room dimensions. Next, it creates two copies of the tiles in the room. Then it uses nested loops to compute each tile for each timestep. The key difference between the rod and room implementations is how the room models the baseboard heater. As explained in Section 2.2A), we assume the heater occupies the middle third of one wall and is only one tile "deep". See Appendix C - Rod/Room Sequential Implementation for full details.

2.2.C) Room Parallel Implementation

Like the sequential implementation, our parallel implementation of the room model accepts parameters for the number of tiles, number of timesteps, and the index of the tile of the room to be sampled.

Next, our host helper function calc_room_diffusion (Appendix J Line 110) sets the block and grid size. As with the parallel rod program in Section 2.2.C), our room application is a preliminary parallel implementation. We simply select a block size of 32 based on the default warp size. Likewise, we choose a grid size by simply dividing the space used to store the rod models by the block size (and rounding up to ensure the last block is properly computed) see Line 138.

The host calls two helper functions to execute the diffusion computation. First the host calls cu_initialize_room for both room copies to set the room state at timestep 0. Then for each timestep, the host invokes a cu_room_diffusion_step kernel that computes a single timestep and updates the state of one of the room copies. Finally, the device copies the temperature at the specified index after the specified number of timesteps back to the host. Appendix J also includes sample code (commented out) that demonstrates how the entire room state could be copied between host and device for development, debugging, and alternative applications.

2.2.D) Room Solution Validation

We validated the room model programs with a similar process to that of the rod validation. First, we developed a simple Microsoft Excel model for a 10x10 room that computed ten timesteps (introduced in Figure 6) and a larger 100x100 room that computed 100 timesteps (see Appendix K). Next, we wrote the sequential implementation in Appendix C. We manually inspected a reasonable sample of tiles to confirm the sequential program matched the Excel model (adjusting for rounding). Then, we confirmed that the numpy and plotly visualization matched our intuition of how heat would radiate from the baseboard heater (see Appendix L). Finally, we periodically checked that our parallel program output always matched the output from our sequential program with BASH scripts like that in Appendix I.

3. Experimental Results

3.1 Overview

We determined experimental results on a Grand Valley State University Architecture Laboratory machine. The system includes an AMD Ryzen 7 2700x 8-core CPU with 16 GB RAM and an NVIDIA GeForce 1050ti 768-core GPU with 4 GB of memory. Additionally, it ran Ubuntu 20.04 and CUDA 10.1.

We have run five iterations of each implementation and analyze only the average runtime for a given implementation. A sample script for running the trials as well as the full details of every trial's performance are available in Appendices M and N respectively. Additionally, we note that Speedup is defined by the equation:

$$Speedup = Time_{Sequential} / Time_{Parallel}$$

The CUDA implementation of the rod model led to 5.5x speedup (Figure 10) based on the improved performance in Figure 11.

Figure 10

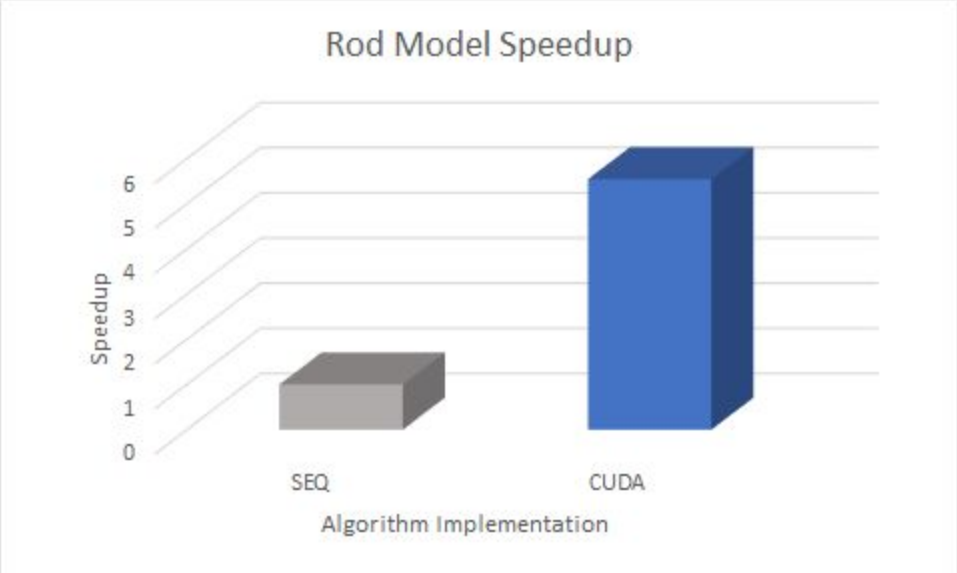
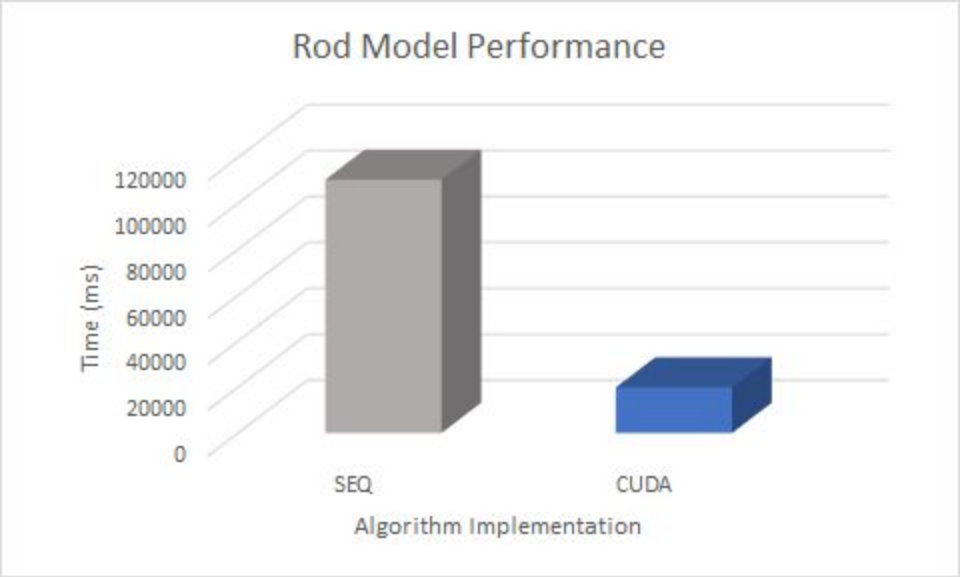


Figure 11



In contrast, the CUDA implementation of the room model led to more than a 50x slowdown compared to the sequential version (see Figures 12 and 13). Notably, our program was a nearly identical copy of its sequential equivalent. Unfortunately, failing to apply most CUDA constructs and best practices led to a severe decrease in performance.

Figure 12

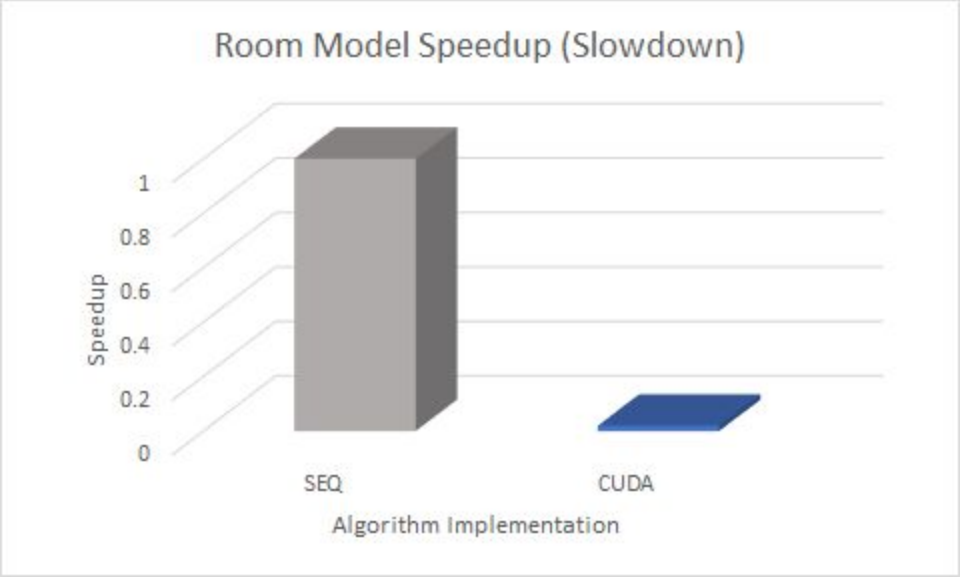
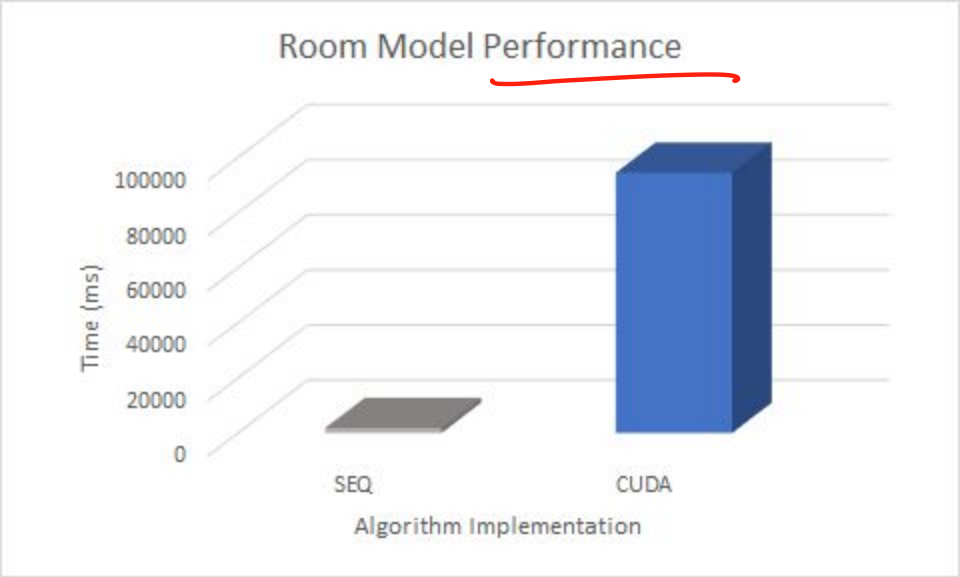


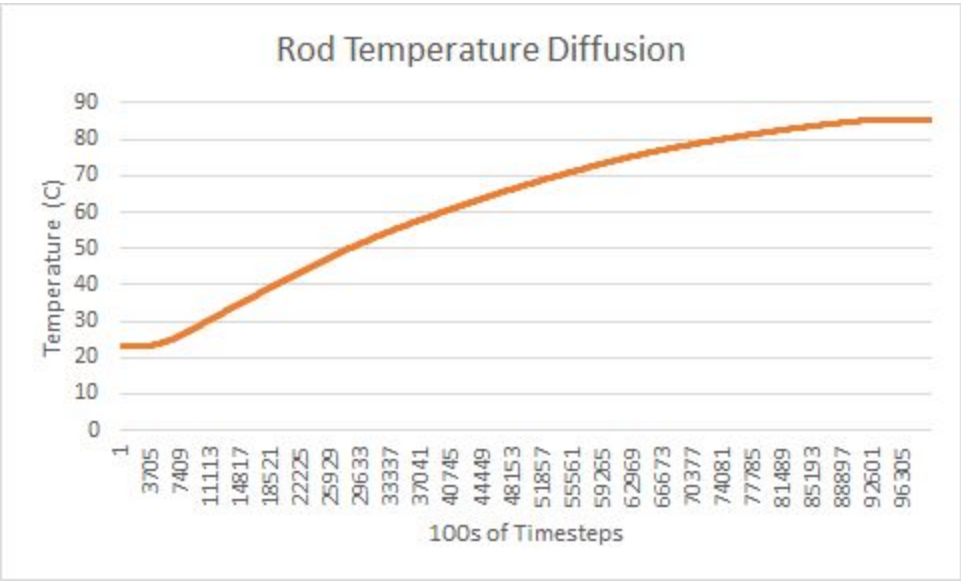
Figure 13



Full performance data for all of our trials is available in Appendix N.

Finally, we present the change in temperature at a specific point in the rod and the room models over a relatively long period of time. Figure 14 shows how the temperature at the slice 0.7m away from the bucket heater changes over ten million timesteps using 2500 slices. The temperature change initially lags as the heat diffuses to the slice at 0.7m. Then the rate of temperature increase accelerates. Last we see a plateau as the temperature begins to approach the temperature of the heater.

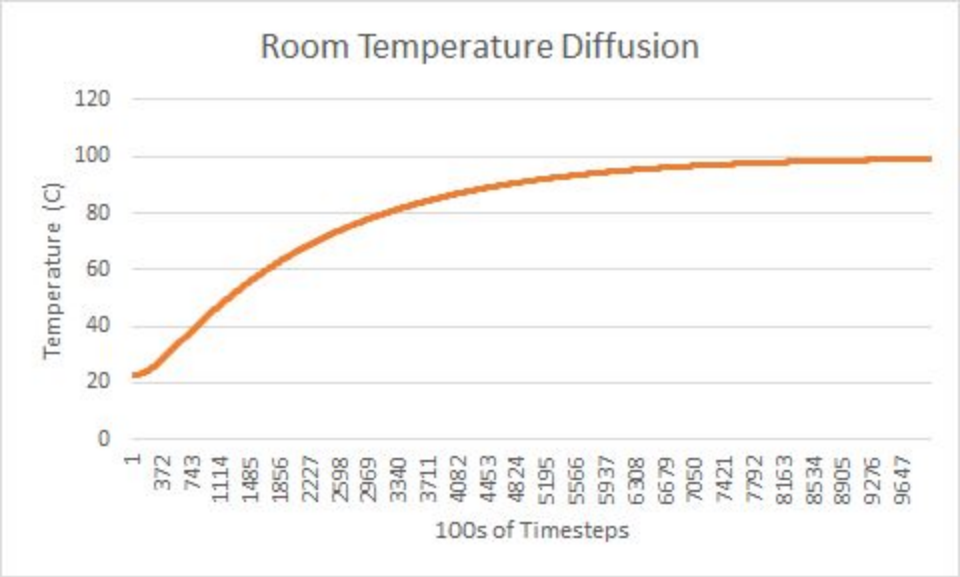
Figure 14



Likewise, Figure 15 shows the temperature change at a position 70% of the width and 70% of

the depth of a square room with 90,000 tiles over one million timesteps. As with Figure 14 the temperature change accelerates and then plateaus. Additionally, we note that Figure 15 shows that this particular tile is approaching the convergence temperature.

Figure 15



4. Problems and Solutions

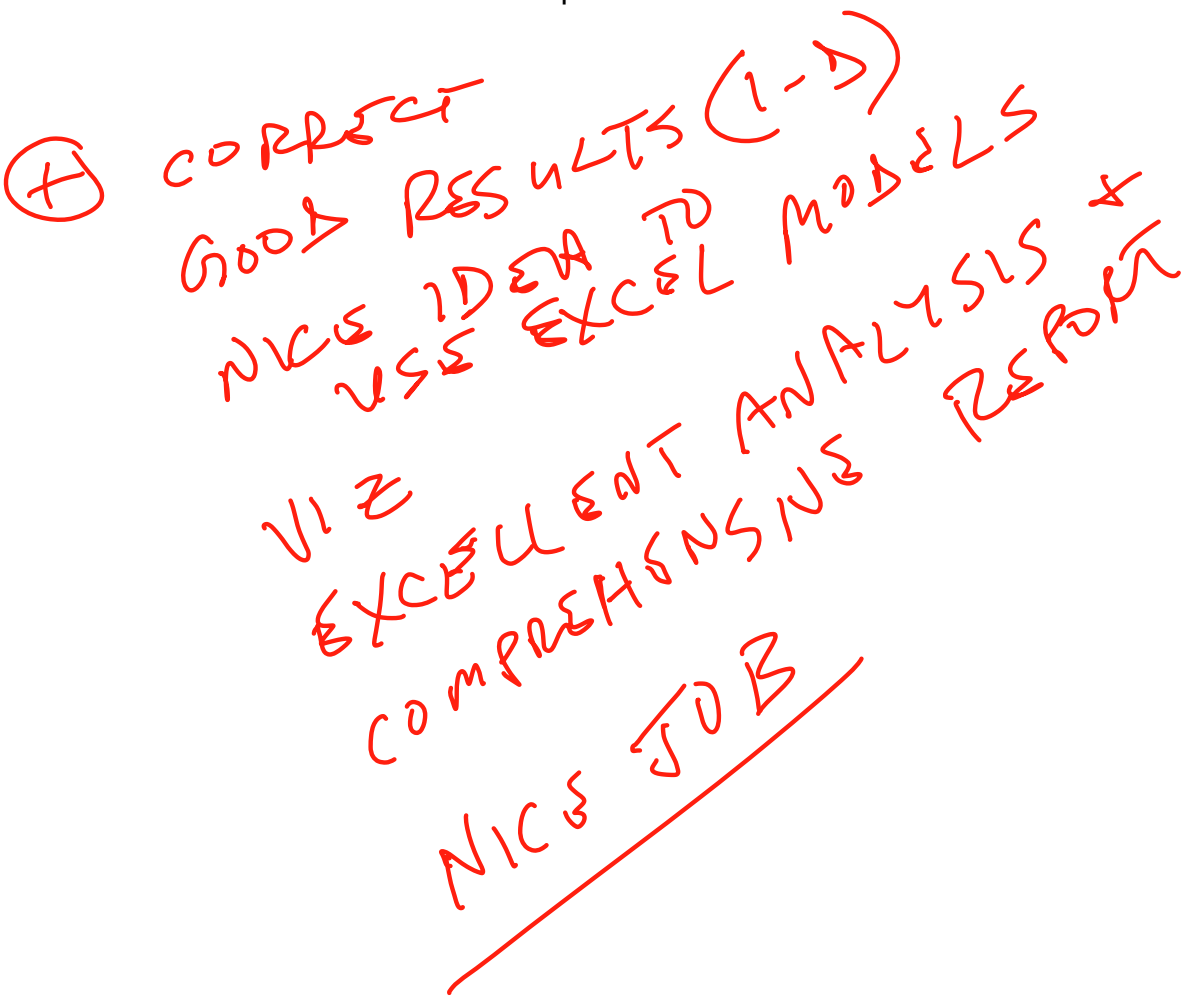
4.1 Project Management - I regret not spending more time and effort to develop and tune my CUDA implementations. First, I spent a significant amount of time on the Excel models and "test benches" for both the rod and room. The textbook emphasizes the importance of correctness to high performance computing - it is not useful to get to an inaccurate answer quickly. Regardless, my focus on correctness put me somewhat behind schedule. Next, I worked on developing the sequential implementations. I got further behind schedule with a subtle bug in the room model's sequential program. Then I prioritized working on the numpy and plotly visualization because I thought it would be useful for debugging large models over long times. Regretfully, this put me still further behind. By the time I began working on the CUDA implementations I was relatively behind schedule and wanted to ensure I had some form of CUDA programs running. The results are simple programs that utilize few CUDA constructs and best practices.

4.2 Data Visualization - My only data visualization programming experience was using gnuplot to generate simple charts and graphs for Programming Assignment #1. I understood that data visualization was an important aspect of high performance computing because it helps us understand results (and even debug). Thus, I chose to spend time researching and experimenting with different data visualization frameworks. While this probably limited the time I could spend working on the CUDA implementations, I think both are excellent uses of time.

4.3 Diffusion Model - As explained in Section 2.1A, I chose to model diffusion in the final "slice" of the rod as the average of *itself* and its left neighbor from the previous timestep. In retrospect I think this was a mistake because one of the basic assumptions of the rod model was that the rod was fully insulated. Since the final slice only has "heat in" it may have been more proper to simply copy the final slice's value from the left neighbor's value in the previous timestep.

I may have been somewhat hasty when considering this issue when I was developing the model. When I reconsidered the issue I had already finished the Excel model, the sequential rod implementation, and the majority of the sequential implementation so I was reluctant to change the model due to the project's time constraints.

Ultimately the difference between my temperature computation of the rod at location 0.7 m seems reasonable. I calculated 85.5277 compared to Dr. Wolffe's 85.531494.



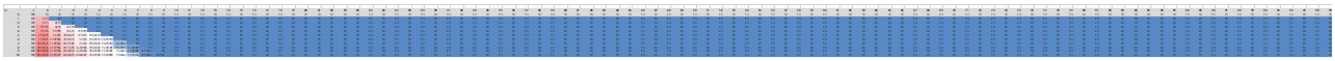
Appendix A - Microsoft Excel Model (Rod 10x1 20 Timesteps)

drum_temp	100	*Note t in this table is time so that t0 = initial conditions. Thus it is 1 higher than the TIME_STEP index in code.
rod_temp	23	

t/i	-1	0	1	2	3	4	5	6	7	8	9
0	100	23	23	23	23	23	23	23	23	23	23
1	100	61.5	23	23	23	23	23	23	23	23	23
2	100	61.5	42.25	23	23	23	23	23	23	23	23
3	100	71.125	42.25	32.625	23	23	23	23	23	23	23
4	100	71.125	51.875	32.625	27.8125	23	23	23	23	23	23
5	100	75.9375	51.875	39.84375	27.8125	25.40625	23	23	23	23	23
6	100	75.9375	57.89063	39.84375	32.625	25.40625	24.20313	23	23	23	23
7	100	78.94531	57.89063	45.25781	32.625	28.41406	24.20313	23.60156	23	23	23
8	100	78.94531	62.10156	45.25781	36.83594	28.41406	26.00781	23.60156	23.30078	23	23
9	100	81.05078	62.10156	49.46875	36.83594	31.42188	26.00781	24.6543	23.30078	23.15039	23
10	100	81.05078	65.25977	49.46875	40.44531	31.42188	28.03809	24.6543	23.90234	23.15039	23.0752
11	100	82.62988	65.25977	52.85254	40.44531	34.2417	28.03809	25.97021	23.90234	23.48877	23.11279
12	100	82.62988	67.74121	52.85254	43.54712	34.2417	30.10596	25.97021	24.72949	23.50757	23.30078
13	100	83.87061	67.74121	55.64417	43.54712	36.82654	30.10596	27.41772	24.73889	24.01514	23.40417
14	100	83.87061	69.75739	55.64417	46.23535	36.82654	32.12213	27.42242	25.71643	24.07153	23.70966
15	100	84.87869	69.75739	57.99637	46.23535	39.17874	32.12448	28.91928	25.74698	24.71304	23.89059
16	100	84.87869	71.43753	57.99637	48.58755	39.17992	34.04901	28.93573	26.81616	24.81879	24.30182
17	100	85.71877	71.43753	60.01254	48.58814	41.31828	34.05782	30.43259	26.87726	25.55899	24.5603
18	100	85.71877	72.86565	60.01284	50.66541	41.32298	35.87543	30.46754	27.99579	25.71878	25.05965
19	100	86.43283	72.8658	61.76553	50.66791	43.27042	35.89526	31.93561	28.09316	26.52772	25.38921
20	100	86.4329	74.09918	61.76686	52.51798	43.28159	37.60302	31.99421	29.23166	26.74119	25.95847

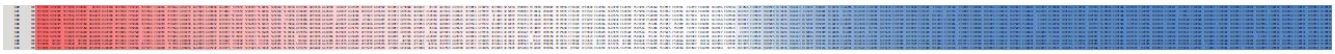
Appendix B - Microsoft Excel Model (Rod 100x1 1000 Timesteps)

Timesteps 1-10



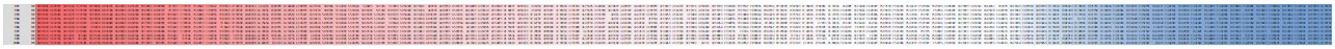
.
.
.

Timesteps 501-510



.
.
.

Timesteps 991-1000



Appendix C - Rod/Room Sequential Implementation

```cpp
#include <iostream>
#include <chrono>

/* Module for computing diffusion in a rod */
float diffusion_seq_1D(char *const *argv);

/* Module for computing diffusion in a room */
float diffusion_seq_2D(char *const *argv);

/* Helper function for printing a time step of the rod model */
void printRodTimeStep(const float *rod, int SLICES, int X_SIZE,
                      int TIME_STEP);

/* Helper function for printing all of the room model's state */
void printArray(const float *room, int TILES, int X_SIZE, int Y_SIZE);

/* Helper function for printing a time step of the room model */
void printRoomTimeStep(const float *room, int TILES, int X_SIZE,
                       int Y_SIZE, int TIME_STEPS);


using namespace std;


/**********************************************************************
 * This function serves as the program "driver" for an implementation
 * of a heat diffusion model of a rod or a room.
 *
 * @param argc is the number of command-line arguments.
 * @param v is the array of strings representing the command-line args.
 **********************************************************************/
int main(int argc, char **argv) {
    auto start_seq = chrono::high_resolution_clock::now();
    const int DIMENSIONS = stoi(argv[1]);
    float temperature;
    switch (DIMENSIONS) {
        case 1:
            temperature = diffusion_seq_1D(argv);
            break;
        case 2:
            temperature = diffusion_seq_2D(argv);
            break;
        default:
            cerr << "Invalid dimension arg must be 1 or 2)" << endl;
    }
    auto end_seq = chrono::high_resolution_clock::now();
    chrono::duration<double, milli> seq_millisec = end_seq - start_seq;
    cout << "Temp: " << temperature << endl;
    cout << "Time: " << seq_millisec.count() << " ms" << endl;
    return 0;
}


/**********************************************************************
 * This helper function serves as the program implementation
 * of a heat diffusion model for a rod.
 *
 * @param v is the array of strings representing the command-line args
 *
 * @return is the temperature value for the specified rod location.
 **********************************************************************/
float diffusion_seq_1D(char *const *argv) {
    const int SLICES = stoi(argv[2]);
```

```
 64     const int TIME_STEPS = stoi(argv[3]);
 65     const int LOC_X_INDEX = stoi(argv[4]);
 66     const int LOC_Y_INDEX = stoi(argv[5]);
 67     const int X_SIZE = stoi(argv[6]);
 68     const int Y_SIZE = stoi(argv[7]);
 69
 70     //malloc 2 rods of space for "old" and "new" vals
 71     const int COPIES = 2;
 72     const int ARRAY_SZ = SLICES * COPIES;
 73     const int INIT_TEMP = 23.0;
 74     const int HEATER = 100.0;
 75
 76     auto *rod = (float *) malloc(sizeof(float) * ARRAY_SZ);
 77     for (int i = 0; i < ARRAY_SZ; i++)
 78         rod[i] = INIT_TEMP;
 79
 80     for (int i = 0; i < TIME_STEPS; i++) {
 81         const int CP_INDEX = ((i % 2) * SLICES);
 82         //calc each rod slice element
 83         for (int j = 0; j < SLICES; j++) {
 84             // the rod slice element in "this" time step
 85             const int EL_INDEX = CP_INDEX + j;
 86             // the rod slice element in "previous" time step
 87             const int PREV_INDEX = ((((i % 2) + 1) % 2) * SLICES) + j;
 88             //special case of first slice
 89             if (j == 0) {
 90                 // avg of heater and right neighbor
 91                 rod[EL_INDEX] = (HEATER + rod[PREV_INDEX + 1]) / 2;
 92             }
 93                 //special case of last slice
 94                 // -> average me and my left neighbor
 95             else if (j == SLICES - 1) {
 96                 rod[EL_INDEX] =
 97                         (rod[PREV_INDEX - 1] + rod[PREV_INDEX]) / 2;
 98             }
 99                 // common case - average left and right neighbors
100                 // from last time step
101             else {
102                 rod[EL_INDEX] =
103                         (rod[PREV_INDEX - 1] + rod[PREV_INDEX + 1])
104                         / 2;
105             }
106         }
107         //printRodTimeStep(rod, SLICES, X_SIZE, i);
108     }
109     const int CP_INDEX = (((TIME_STEPS + 1) % 2) * SLICES);
110     return rod[CP_INDEX + LOC_X_INDEX];
111 }
112
113
114 /**********************************************************************
115  * This helper function serves as the program implementation
116  * of a heat diffusion model for a room.
117  *
118  * @param v is the array of strings representing the command-line args
119  *
120  * @return is the temperature value for the specified room location.
121  *********************************************************************/
122 float diffusion_seq_2D(char *const *argv) {
123     const int TILES = stoi(argv[2]);
124     const int TIME_STEPS = stoi(argv[3]);
125     const int LOC_X_INDEX = stoi(argv[4]);
126     const int LOC_Y_INDEX = stoi(argv[5]);
```

```
127     const int X_SIZE = stoi(argv[6]);
128     const int Y_SIZE = stoi(argv[7]);
129     if (X_SIZE * Y_SIZE != TILES)
130         cerr << "Invalid X or Y Size dimensions (X x Y must = TILES)"
131              << endl;
132     //malloc 2 grids of space for "old" and "new" vals
133     const int COPIES = 2;
134     const int ARRAY_SZ = TILES * COPIES;
135     const float INIT_TEMP = 23.0;
136     const float HEATER = 100.0;
137
138     auto *room = (float *) malloc(sizeof(float) * ARRAY_SZ);
139
140     //initialization - assume heater occupies the middle third of wall
141     const int HEATER_START = X_SIZE / 3;
142     const int HEATER_END = HEATER_START * 2;
143     for (int k = 0; k < COPIES; k++) {
144         for (int i = 0; i < Y_SIZE; i++) {
145             for (int j = 0; j < X_SIZE; j++) {
146                 //set heater tiles
147                 if ((i == 0) &&
148                     ((j >= HEATER_START) &&
149                      j < HEATER_END))
150                     room[(k * TILES) + j] = HEATER;
151                     //set common case tiles
152                 else room[(k * TILES) + (i * X_SIZE) + j] = INIT_TEMP;
153             }
154         }
155     }
156
157     //printArray(room, TILES, X_SIZE, Y_SIZE);
158     for (int k = 0; k < TIME_STEPS; k++) {
159         for (int i = 0; i < Y_SIZE; i++) {
160             for (int j = 0; j < X_SIZE; j++) {
161                 const int EL_INDEX = ((k % 2) * (TILES)) +
162                                      (i * X_SIZE) + j;
163                 const int PREV_EL_INDEX =
164                         ((((k % 2) + 1) % 2) * (TILES)) +
165                         (i * X_SIZE) + j;
166                 //case 1 - heater tiles
167                 if ((i == 0) && ((j >= HEATER_START) &&
168                                  (j < HEATER_END))) {
169                     //ignore i since we know we're in the first row
170                     room[EL_INDEX] = HEATER;
171                 }
172                     //case 2 - NW corner
173                 else if ((i == 0) && (j == 0)) {
174                     //average myself, E, and S from last time step
175                     room[EL_INDEX] = (room[PREV_EL_INDEX] +
176                                       room[(PREV_EL_INDEX) +
177                                            1] +
178                                       room[(PREV_EL_INDEX) +
179                                            X_SIZE]) / 3;
180                 }
181                     //case 3 - NE corner
182                 else if ((i == 0) && (j == X_SIZE - 1)) {
183                     //average myself, W and S from last time step
184                     room[EL_INDEX] =
185                             ((float) (room[PREV_EL_INDEX] +
186                                       room[PREV_EL_INDEX -
187                                            1] +
188                                       room[PREV_EL_INDEX +
189                                            X_SIZE])) / 3;
```

```
190                 }
191                     //case 4 - N border
192                 else if (i == 0) {
193                     //average myself, W, E, and S from last time step
194                     room[EL_INDEX] =
195                             (room[PREV_EL_INDEX] +
196                              room[PREV_EL_INDEX - 1] +
197                              room[PREV_EL_INDEX + 1] +
198                              room[PREV_EL_INDEX + X_SIZE]) /
199                             4;
200                 }
201                     //case 5 - SW corner
202                 else if ((i == Y_SIZE - 1) && (j == 0)) {
203                     //average myself, E, and N from last time step
204                     room[EL_INDEX] =
205                             (room[PREV_EL_INDEX] +
206                              room[PREV_EL_INDEX +
207                                   1] +
208                              room[PREV_EL_INDEX -
209                                   X_SIZE]) / 3;
210                 }
211                     //case 6 - W border
212                 else if (j == 0) {
213                     //average myself, E, N, and S from last time step
214                     room[EL_INDEX] =
215                             (room[PREV_EL_INDEX] +
216                              room[PREV_EL_INDEX +
217                                   1] +
218                              room[PREV_EL_INDEX -
219                                   X_SIZE] +
220                              room[PREV_EL_INDEX +
221                                   X_SIZE]) / 4;
222                 }
223                     //case 7 - SE corner
224                 else if ((i == Y_SIZE - 1) && (j == X_SIZE - 1)) {
225                     //average myself, W, and N from last time step
226                     room[EL_INDEX] =
227                             (room[PREV_EL_INDEX] +
228                              room[PREV_EL_INDEX -
229                                   1] +
230                              room[PREV_EL_INDEX -
231                                   X_SIZE]) / 3;
232                 }
233                     //case 8 - E border
234                 else if (j == X_SIZE - 1) {
235                     //average myself, W, N, and S from last time step
236                     room[EL_INDEX] =
237                             (room[PREV_EL_INDEX] +
238                              room[PREV_EL_INDEX -
239                                   1] +
240                              room[PREV_EL_INDEX -
241                                   X_SIZE] +
242                              room[PREV_EL_INDEX +
243                                   X_SIZE]) / 4;
244                 }
245                     //case 8 - S border
246                 else if (i == Y_SIZE - 1) {
247                     //average myself, W, E, and N from last time step
248                     room[EL_INDEX] =
249                             (room[PREV_EL_INDEX] +
250                              room[PREV_EL_INDEX - 1] +
251                              room[PREV_EL_INDEX + 1] +
252                              room[PREV_EL_INDEX - X_SIZE]) /
```

```
253                             4;
254                 }
255                     //case 9 - common case of center of the room
256                 else {
257                     //average W, E, N, and S from last time step
258                     room[EL_INDEX] =
259                             (room[PREV_EL_INDEX - 1] +
260                              room[PREV_EL_INDEX + 1] +
261                              room[PREV_EL_INDEX - X_SIZE] +
262                              room[PREV_EL_INDEX + X_SIZE]
263                             ) / 4;
264                 }
265             }
266         }
267         cout << room[((k%2)* TILES) + (LOC_Y_INDEX * X_SIZE) + LOC_X_INDEX
    ] << endl;
268     }
269     //printArray(room, TILES, X_SIZE, Y_SIZE);
270     //printRoomTimeStep(room, TILES, X_SIZE, Y_SIZE, TIME_STEPS);
271     const int CP_INDEX = (((TIME_STEPS + 1) % 2) * (TILES));
272     return room[CP_INDEX + (LOC_Y_INDEX * X_SIZE) + LOC_X_INDEX];
273 }
274
275
276 /**************************************************************************
277  * This helper function prints the entire room state in a "human-
278  * readable format".
279  *
280  * @param room is the pointer to the room model's temperature state.
281  * @param TILES is the number of pieces the room is divided into
282  * for temperature computations.
283  * @param X_SIZE is the width of the room.
284  * @param Y_SIZE is the depth of the room.
285  *************************************************************************/
286 void printArray(const float *room, const int TILES, const int X_SIZE,
287                 const int Y_SIZE) {
288     for (int i = 0; i < TILES * 2; i++) {
289         if (i == TILES)
290             cout << endl << endl << endl;
291         if (i % X_SIZE == 0)
292             cout << endl;
293         cout << room[i] << " ";
294     }
295     cout << endl << endl << endl;
296 }
297
298
299 /**************************************************************************
300  * This helper function prints half of the room state in a "human-
301  * readable format" (only one time step is printed).
302  *
303  * @param room is the pointer to the room model's temperature state.
304  * @param TILES is the number of pieces the room is divided into
305  * for temperature computations.
306  * @param X_SIZE is the width of the room.
307  * @param Y_SIZE is the depth of the room.
308  * @param TIME_STEPS is the time step to display.
309  *************************************************************************/
310 void printRoomTimeStep(const float *room, const int TILES,
311                        const int X_SIZE, const int Y_SIZE,
312                        int TIME_STEPS) {
313     for (int i = 0; i < TILES; i++) {
314         if (i % X_SIZE == 0 && i != 0)
```

```
315             cout << endl;
316         cout << room[(((TIME_STEPS + 1) % 2) * TILES) + i] << " ";
317     }
318     cout << endl;
319 }
320 
321 
322 /*****************************************************************
323  * This helper function prints half of the rod state in a "human-
324  * readable format" (only one time step is printed).
325  *
326  * @param rod is the pointer to the rod model's temperature state.
327  * @param SLICES is the number of pieces the rod is divided into
328  * for temperature computations.
329  * @param X_SIZE is the width of the room.
330  * @param TIME_STEP is the time step to display.
331  *****************************************************************/
332 void printRodTimeStep(const float *rod, const int SLICES,
333                       const int X_SIZE, int TIME_STEP) {
334     for (int i = 0; i < SLICES; i++) {
335         cout << rod[((TIME_STEP % 2) * SLICES) + i] << " ";
336     }
337     cout << endl;
338 }
```

Appendix D - Rod CUDA Implementation

```
 1 #include <stdio.h>
 2 #include <iostream>
 3 #include <stdlib.h>
 4 #include <chrono>
 5 #include "math.h"
 6
 7 using namespace std;
 8
 9 #define BLOCK_SIZE 32
10
11 /* Helper kernel to set initial rod state */
12 __global__ void cu_initialize_rod(float *rod_state_d,
13                                   const int ARRAY_SIZE);
14
15 /* Helper kernel to compute rod state after one additional time step */
16 __global__ void cu_rod_diffusion_step(float *rod_state_d,
17                                       const int ARRAY_SIZE,
18                                       const int START,
19                                       const int END);
20
21 /* Helper function to compute the rod's temperature */
22 void calc_rod_diffusion(float *temperature, const int SLICES,
23                         const int TIME_STEPS,
24                         const int LOC_X,
25                         const int X_SIZE, float *rod_state,
26                         const int ARRAY_SIZE);
27
28 /* Helper function to print the rod's state */
29 void print_rod_state(const float *state, const int SIZE,
30                      const int TIME_STEPS);
31
32 /* Model the bucket/heater at consistent 100 degrees */
33 const int HEATER_TEMP = 100;
34
35 /* Model the rod at 23 degrees at initialization */
36 const int INIT_TEMP = 23;
37
38
39 /*****************************************************************
40  * This function serves as the program "driver" for an implementation
41  * of a heat diffusion model of a rod.
42  *
43  * @param argc is the number of command-line arguments.
44  * @param v is the array of strings representing the command-line args.
45  *****************************************************************/
46 int main(int argc, char *argv[]) {
47     if (argc != 8) {
48         cerr << "usage: pgName slice timeSteps locX locY xSize ySize"
49              << endl;
50         exit(-1);
51     }
52
53     auto start_cuda = chrono::high_resolution_clock::now();
54
55     const int SLICES = stoi(argv[2]);
56     const int TIME_STEPS = stoi(argv[3]);
57     const int LOC_X_INDEX = stoi(argv[4]);
58     const int LOC_Y_INDEX = stoi(argv[5]);
59     const int X_SIZE = stoi(argv[6]);
60     const int Y_SIZE = stoi(argv[7]);
61     float temperature = -1.0;
62     const int COPIES = 2;
63     //COPIES 2 extra spaces to handle the bucket initialization
```

```
64      // in both copies
65      const int ARRAY_SZ = (SLICES * COPIES) + COPIES;
66
67      //FIXME rm from performance trials
68      //auto *rod_state = (float *) malloc(sizeof(float) * ARRAY_SZ);
69      auto *rod_state = (float *) malloc(sizeof(float) * 1);
70      calc_rod_diffusion(&temperature, SLICES, TIME_STEPS, LOC_X_INDEX,
71                         X_SIZE, rod_state, ARRAY_SZ);
72      //print_rod_state(rod_state, ARRAY_SZ, TIME_STEPS);
73
74      auto end_cuda = chrono::high_resolution_clock::now();
75      chrono::duration<double, milli> seq_millisec = end_cuda -
76                                                     start_cuda;
77
78      cout << "Temp: " << temperature << endl;
79      cout << "Time: " << seq_millisec.count() << " ms" << endl;
80  }
81
82
83  /*****************************************************************
84   * This helper function computes the rod's temperature at the specified
85   * location after the specified number of time steps.
86   *
87   * @param temperature is the location to update with the results of
88   * the computation.
89   * @param SLICES is the number of pieces the rod is divided into
90   * for temperature computations.
91   * @param TIME_STEPS is the total number of time steps to compute.
92   * @param LOC_X is the location on the rod to be sampled.
93   * @param rod_state is the location to copy the rod state.
94   * @param ARRAY_SIZE is the total size of the rod array.
95   *****************************************************************/
96  void calc_rod_diffusion(float *temperature, const int SLICES,
97                          const int TIME_STEPS,
98                          const int LOC_X,
99                          const int X_SIZE, float *rod_state,
100                         const int ARRAY_SIZE) {
101     float *rod_state_d;
102     cudaError_t result;
103     const int CP_INDEX = (TIME_STEPS + 1) % 2;
104     const int BASE = CP_INDEX * (ARRAY_SIZE / 2) + 1;
105
106
107     result = cudaMalloc((void **) &rod_state_d,
108                         sizeof(float) * ARRAY_SIZE);
109     if (result != cudaSuccess) {
110         fprintf(stderr, "cudaMalloc (block) failed.");
111         exit(1);
112     }
113
114     dim3 dimblock(BLOCK_SIZE);
115     dim3
116     dimgrid(ceil((double) (ARRAY_SIZE) / BLOCK_SIZE));
117
118     cu_initialize_rod <<<dimgrid, dimblock>>>(rod_state_d, ARRAY_SIZE);
119
120     for (int k = 0; k < TIME_STEPS; k++) {
121         const int START = ((k % 2) * ((ARRAY_SIZE / 2) + 1));
122         const int END = ((((k) % 2) + 1) * (ARRAY_SIZE / 2)) - 1;
123         cu_rod_diffusion_step<<<dimgrid, dimblock>>>(rod_state_d,
124                                                      ARRAY_SIZE,
125                                                      START,
126                                                      END);
```

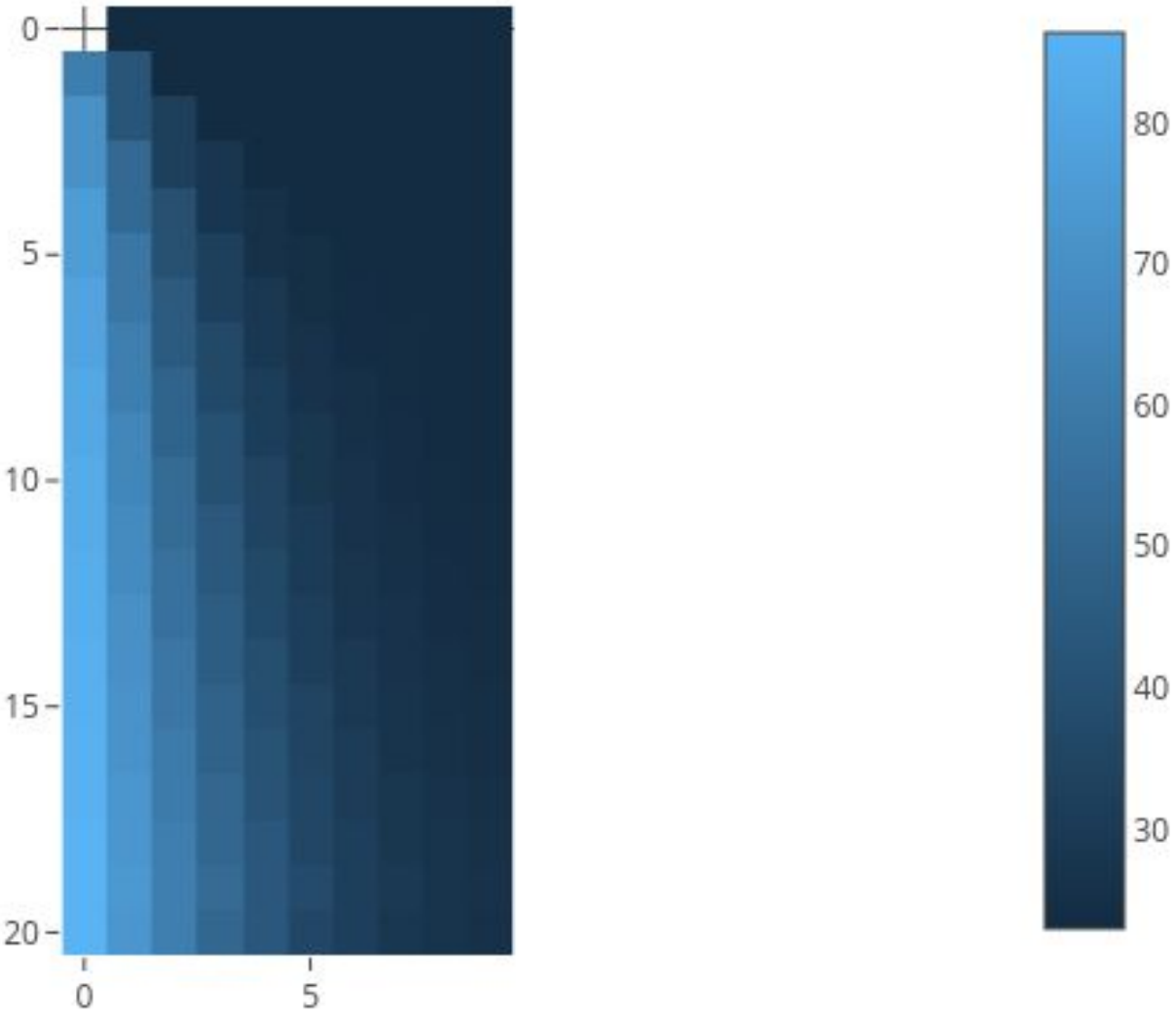
```
127     }
128
129     // transfer whole rod state back to host
130     //result = cudaMemcpy(rod_state, rod_state_d,
131     // sizeof(float) * ARRAY_SIZE, cudaMemcpyDeviceToHost);
132     // transfer only resulting temp back to host
133     result = cudaMemcpy(temperature,
134                         (void *) &rod_state_d[BASE + LOC_X],
135                         sizeof(float), cudaMemcpyDeviceToHost);
136     if (result != cudaSuccess) {
137         fprintf(stderr, "cudaMemcpy host <- dev (block) failed.");
138         exit(1);
139     }
140
141     result = cudaFree(rod_state_d);
142     if (result != cudaSuccess) {
143         fprintf(stderr, "cudaFree (block) failed.");
144         exit(1);
145     }
146 }
147
148
149 /*****************************************************************
150  * This kernel computes the rod's temperature at the specified
151  * location after one additional time step.
152  *
153  * @param rod_state_d is the location of the rod state.
154  * @param ARRAY_SIZE is the total size of the rod array.
155  * @param START is the location to begin computing the next time step.
156  * @param START is the location to end computing the next time step.
157  *****************************************************************/
158 __global__ void cu_rod_diffusion_step(float *rod_state_d,
159                                       const int ARRAY_SIZE,
160                                       const int START, const int END) {
161     int i = blockIdx.x * BLOCK_SIZE + threadIdx.x;
162     const int PREV = ((ARRAY_SIZE / 2) + i) % ARRAY_SIZE;
163     if (i >= START && i < END) {
164         //start of both copies is the 100 degree const so do nothing
165         if ((i == 0) || (i == (((ARRAY_SIZE / 2)))));
166             //common case
167         else {
168             rod_state_d[i] = ((float)
169                     (rod_state_d[PREV - 1] + rod_state_d[PREV + 1]))
170                             / 2;
171         }
172     }
173     //special case of end of the rod
174     if (i == END)
175         rod_state_d[i] = ((float)
176                 (rod_state_d[PREV - 1] + rod_state_d[PREV]))
177                         / 2;
178 }
179
180
181 /*****************************************************************
182  * This kernel initializes the rod's temperature.
183  *
184  * @param rod_state_d is the location of the rod state.
185  * @param ARRAY_SIZE is the total size of the rod array.
186  *****************************************************************/
187 __global__ void cu_initialize_rod(float *rod_state_d,
188                                   const int ARRAY_SIZE) {
189     int i = blockIdx.x * BLOCK_SIZE + threadIdx.x;
```

```
190     if (i < ARRAY_SIZE) {
191         if ((i == 0) || (i == (ARRAY_SIZE / 2)))
192             rod_state_d[i] = HEATER_TEMP;
193         else {
194             rod_state_d[i] = INIT_TEMP;
195         }
196     }
197 }
198 
199 
200 /*********************************************************************
201  * This helper function prints half of the rod state in a "human-
202  * readable format" (only one time step is printed).
203  *
204  * @param state is the pointer to the rod model's temperature state.
205  * @param SIZE is the number of pieces the rod is divided into
206  * for temperature computations.
207  * @param TIME_STEPS is the time step to display.
208  *********************************************************************/
209 void print_rod_state(const float *state, const int SIZE,
210                      const int TIME_STEPS) {
211     for (int i = 1; i < SIZE / 2; i++) {
212         cout << state[(((TIME_STEPS + 1) % 2) * (SIZE / 2)) + i]
213              << " ";
214     }
215     cout << endl;
216 }
```

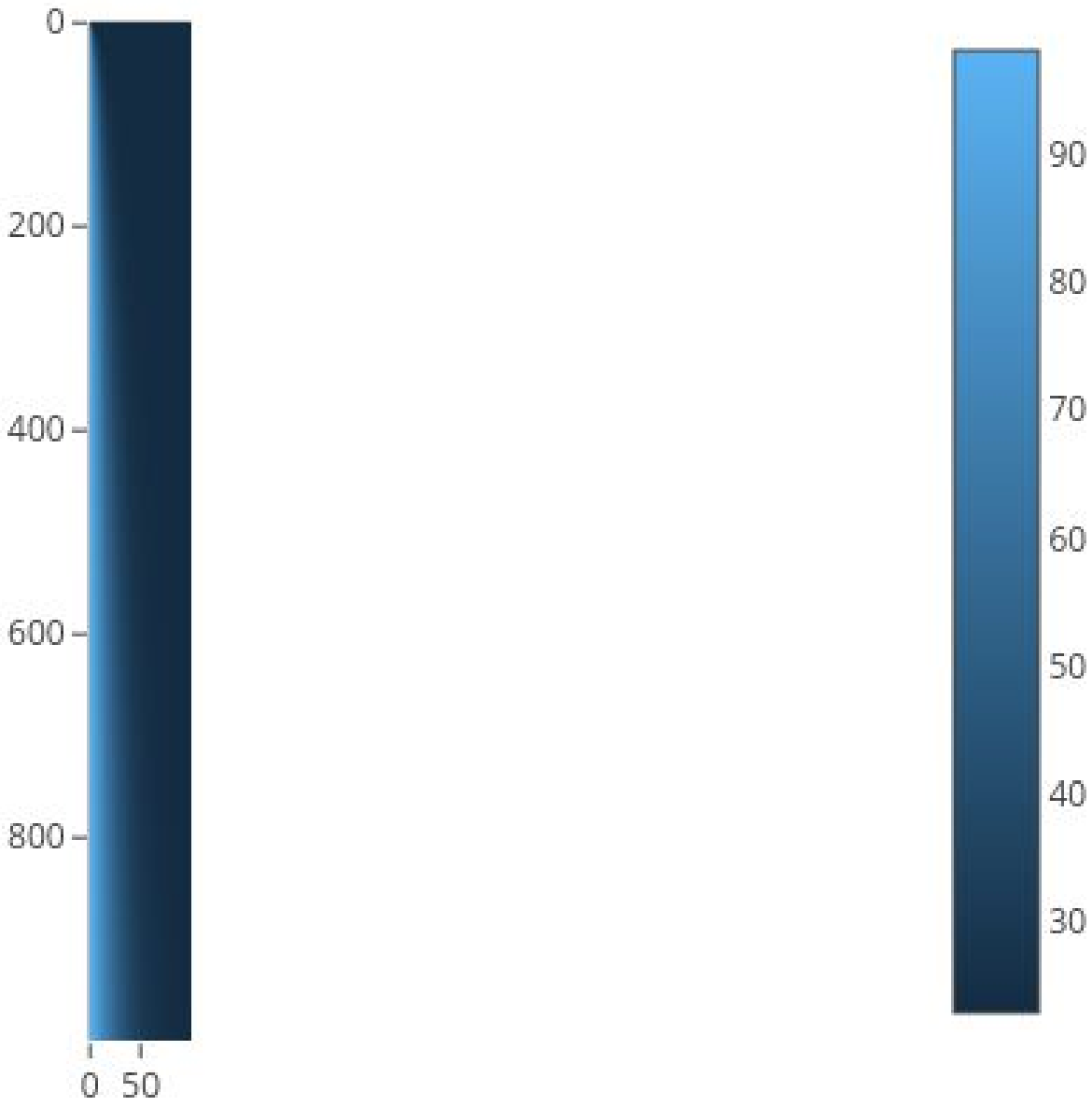
Appendix E - Sample Output of the Sequential Rod Implementation (Rod 10x1 20 Timesteps)

```
 1 61.5 23 23 23 23 23 23 23 23 23
 2 61.5 42.25 23 23 23 23 23 23 23 23
 3 71.125 42.25 32.625 23 23 23 23 23 23 23
 4 71.125 51.875 32.625 27.8125 23 23 23 23 23 23
 5 75.9375 51.875 39.8438 27.8125 25.4062 23 23 23 23 23
 6 75.9375 57.8906 39.8438 32.625 25.4062 24.2031 23 23 23 23
 7 78.9453 57.8906 45.2578 32.625 28.4141 24.2031 23.6016 23 23 23
 8 78.9453 62.1016 45.2578 36.8359 28.4141 26.0078 23.6016 23.3008 23 23
 9 81.0508 62.1016 49.4688 36.8359 31.4219 26.0078 24.6543 23.3008 23.1504 23
10 81.0508 65.2598 49.4688 40.4453 31.4219 28.0381 24.6543 23.9023 23.1504 23.0752
11 82.6299 65.2598 52.8525 40.4453 34.2417 28.0381 25.9702 23.9023 23.4888 23.1128
12 82.6299 67.7412 52.8525 43.5471 34.2417 30.106 25.9702 24.7295 23.5076 23.3008
13 83.8706 67.7412 55.6442 43.5471 36.8265 30.106 27.4177 24.7389 24.0151 23.4042
14 83.8706 69.7574 55.6442 46.2354 36.8265 32.1221 27.4224 25.7164 24.0715 23.7097
15 84.8787 69.7574 57.9964 46.2354 39.1787 32.1245 28.9193 25.747 24.713 23.8906
16 84.8787 71.4375 57.9964 48.5876 39.1799 34.049 28.9357 26.8162 24.8188 24.3018
17 85.7188 71.4375 60.0125 48.5881 41.3183 34.0578 30.4326 26.8773 25.559 24.5603
18 85.7188 72.8657 60.0128 50.6654 41.323 35.8754 30.4675 27.9958 25.7188 25.0596
19 86.4328 72.8658 61.7655 50.6679 43.2704 35.8953 31.9356 28.0932 26.5277 25.3892
20 86.4329 74.0992 61.7669 52.518 43.2816 37.603 31.9942 29.2317 26.7412 25.9585
```

Appendix F - Numpy and Plotly Visualization of Sample Output of the Sequential Rod Implementation (Rod 10x1 20 Timesteps)



Appendix G - Numpy and Plotly Visualization of Sample Output of the Sequential Rod Implementation (Rod 100x1 1000 Timesteps)



Appendix H - Sample Numpy and Plotly Visualization Source Code

```
import plotly.express as px
import plotly.io as pio
import numpy as np
import os

pio.templates.default = "ggplot2"

"""
/*****************************************************************************
 * This function serves as the program "driver" for a data
 * visualization program.  It generates a heat map based on a text
 * file.
 *****************************************************************************/
 """
if __name__ == '__main__':
    img = np.genfromtxt('output_10x1_20.txt', dtype=float, delimiter=' ')
    fig = px.imshow(img)
    fig.update_layout(template="none")
    #fig.show()
    if not os.path.exists("figures"):
        os.mkdir("figures")
    fig.write_image("figures/rod_10x1_20.png")
```

Appendix I - Sample Rod Validation BASH Script

```
rm ./test_bench/cuda_seq_comp/*.txt
g++ -Wall -Wno-unused-variable ../untitled/main.cpp -o pa3
nvcc rod.cu -o diff_rod
if [ $? -eq 0 ]
then
        for i in {1..1000}
        do
                ./pa3 1 2500 10000000 1750 -1 2500 1  >> ./test_bench/cuda_seq_comp/comp1_$i.txt
                ./diff_rod 1 2500 10000000 1750 -1 2500 1  >> ./test_bench/cuda_seq_comp/comp2_$i.txt
                diff -I "^T*" ./test_bench/cuda_seq_comp/comp1_$i.txt ./test_bench/cuda_seq_comp/comp2_$i.txt
        done
fi
```

Appendix J - Room CUDA Implementation

```cpp
#include <stdio.h>
#include <iostream>
#include <stdlib.h>
#include <chrono>
#include "math.h"

using namespace std;

#define BLOCK_SIZE 32

/* Helper kernel to set initial room state */
__global__ void
cu_initialize_room(float *room_state_d, const int ARRAY_SIZE,
                   const int HEATER_START, const int HEATER_END);

/* Helper kernel to compute room state after one additional time step */
__global__ void
cu_room_diffusion_step(float *room_state_d_curr,
                       float *room_state_d_prev, const int TILES,
                       const int X_SIZE, const int Y_SIZE,
                       const int HEATER_START, const int HEATER_END);

/* Helper function to compute the room's temperature */
void calc_room_diffusion(float *temperature, const int TILES,
                         const int TIME_STEPS,
                         const int LOC_X,
                         const int X_SIZE,
                         const int LOC_Y,
                         const int Y_SIZE,
                         float *room_state1,
                         float *room_state2);


/* Helper function to print the room's state */
void print_room_state(const float *state, const int SIZE,
                      const int X_SIZE, const int Y_SIZE);

/* Model the heater at consistent 100 degrees */
const int HEATER_TEMP = 100;

/* Model the room at 23 degrees at initialization */
const int INIT_TEMP = 23;


/**********************************************************************
 * This function serves as the program "driver" for an implementation
 * of a heat diffusion model of a room.
 *
 * @param argc is the number of command-line arguments.
 * @param v is the array of strings representing the command-line args.
 **********************************************************************/
int main(int argc, char *argv[]) {
    if (argc != 8) {
        cerr <<
             "usage: programName slice timeSteps locX locY xSize ySize"
             << endl;
        exit(-1);
    }
    auto start_cuda = chrono::high_resolution_clock::now();

    const int TILES = stoi(argv[2]);
    const int TIME_STEPS = stoi(argv[3]);
    const int LOC_X_INDEX = stoi(argv[4]);
```

```
 64     const int LOC_Y_INDEX = stoi(argv[5]);
 65     const int X_SIZE = stoi(argv[6]);
 66     const int Y_SIZE = stoi(argv[7]);
 67     if (X_SIZE * Y_SIZE != TILES)
 68         cerr << "Invalid X or Y Size dimensions (X x Y must = TILES)"
 69              << endl;
 70     float temperature = -1.0;
 71
 72     //make space for copying back room state
 73 //    auto *room_state1 = (float *) malloc(sizeof(float) * TILES);
 74 //    auto *room_state2 = (float *) malloc(sizeof(float) * TILES);
 75     auto *room_state1 = (float *) malloc(sizeof(float) * 1);
 76     auto *room_state2 = (float *) malloc(sizeof(float) * 1);
 77     calc_room_diffusion(&temperature, TILES, TIME_STEPS, LOC_X_INDEX,
 78                         X_SIZE, LOC_Y_INDEX, Y_SIZE, room_state1,
 79                         room_state2);
 80     //print the proper half of the room state
 81 //    if ((TIME_STEPS + 1) % 2 == 0)
 82 //        print_room_state(room_state1, TILES, X_SIZE, Y_SIZE);
 83 //    else
 84 //        print_room_state(room_state2, TILES, X_SIZE, Y_SIZE);
 85
 86     auto end_cuda = chrono::high_resolution_clock::now();
 87     chrono::duration<double, milli> seq_millisec = end_cuda -
 88                                                    start_cuda;
 89     cout << "Temp: " << temperature << endl;
 90     cout << "Time: " << seq_millisec.count() << " ms" << endl;
 91 }
 92
 93
 94 /*****************************************************************
 95  * This helper function computes the room's temperature at the
 96  * specified location after the specified number of time steps.
 97  *
 98  * @param temperature is the location to update with the results of
 99  * the computation.
100  * @param TILES is the number of pieces the room is divided into
101  * for temperature computations.
102  * @param TIME_STEPS is the total number of time steps to compute.
103  * @param LOC_X is the location on the room to be sampled.
104  * @param X_SIZE is the room width dimension.
105  * @param LOC_Y is the location on the room to be sampled.
106  * @param Y_SIZE is the room depth dimension.
107  * @param room_state1 is the first copy of the room state.
108  * @param room_state2 is the second copy of the room state.
109  *****************************************************************/
110 void calc_room_diffusion(float *temperature, const int TILES,
111                          const int TIME_STEPS,
112                          const int LOC_X,
113                          const int X_SIZE,
114                          const int LOC_Y,
115                          const int Y_SIZE,
116                          float *room_state1,
117                          float *room_state2) {
118     float *room_state1_d;
119     float *room_state2_d;
120     cudaError_t result;
121
122     result = cudaMalloc((void **) &room_state1_d,
123                         sizeof(float) * TILES);
124     if (result != cudaSuccess) {
125         fprintf(stderr, "cudaMalloc (block) failed.");
126         exit(1);
```

```
127     }
128
129     result = cudaMalloc((void **) &room_state2_d,
130                         sizeof(float) * TILES);
131     if (result != cudaSuccess) {
132         fprintf(stderr, "cudaMalloc (block) failed.");
133         exit(1);
134     }
135
136     dim3 dimblock(BLOCK_SIZE, BLOCK_SIZE);
137     int nBlocks = ceil(((double) (TILES)) / BLOCK_SIZE);
138     dim3 dimgrid(nBlocks, nBlocks);
139
140     //initialization - assume heater occupies middle third of  wall
141     const int HEATER_START = X_SIZE / 3;
142     const int HEATER_END = HEATER_START * 2;
143     cu_initialize_room <<<dimgrid, dimblock>>>
144             (room_state1_d, TILES, HEATER_START, HEATER_END);
145     cu_initialize_room <<<dimgrid, dimblock>>>
146             (room_state2_d, TILES, HEATER_START, HEATER_END);
147
148     for (int k = 0; k < TIME_STEPS; k++) {
149         if (k % 2 == 0)
150             cu_room_diffusion_step<<<dimgrid, dimblock>>>
151                     (room_state1_d, room_state2_d, TILES,
152                      X_SIZE, Y_SIZE,
153                      HEATER_START, HEATER_END);
154         else
155             cu_room_diffusion_step<<<dimgrid, dimblock>>>
156                     (room_state2_d, room_state1_d, TILES,
157                      X_SIZE, Y_SIZE,
158                      HEATER_START, HEATER_END);
159     }
160
161     // transfer whole room state 1 back to host during dev
162     //result = cudaMemcpy(room_state1, room_state1_d,
163     // sizeof(float) * TILES, cudaMemcpyDeviceToHost);
164
165     // transfer only resulting temp back to host
166     if (TIME_STEPS % 2 == 1)
167         result = cudaMemcpy(temperature,
168                             (void *) &room_state1_d
169                             [(LOC_Y * X_SIZE) + LOC_X],
170                             sizeof(float),
171                             cudaMemcpyDeviceToHost);
172     else
173         result = cudaMemcpy(temperature,
174                             (void *) &room_state2_d
175                             [(LOC_Y * X_SIZE) + LOC_X],
176                             sizeof(float),
177                             cudaMemcpyDeviceToHost);
178
179     if (result != cudaSuccess) {
180         fprintf(stderr, "cudaMemcpy host <- dev (block) failed.");
181         exit(1);
182     }
183     // transfer whole room state 2 back to host during dev
184 //    result = cudaMemcpy(room_state2,
185 //    room_state2_d, sizeof(float) * TILES, cudaMemcpyDeviceToHost);
186 //    if (result != cudaSuccess) {
187 //        fprintf(stderr, "cudaMemcpy host <- dev (block) failed.");
188 //        exit(1);
189 //    }
```

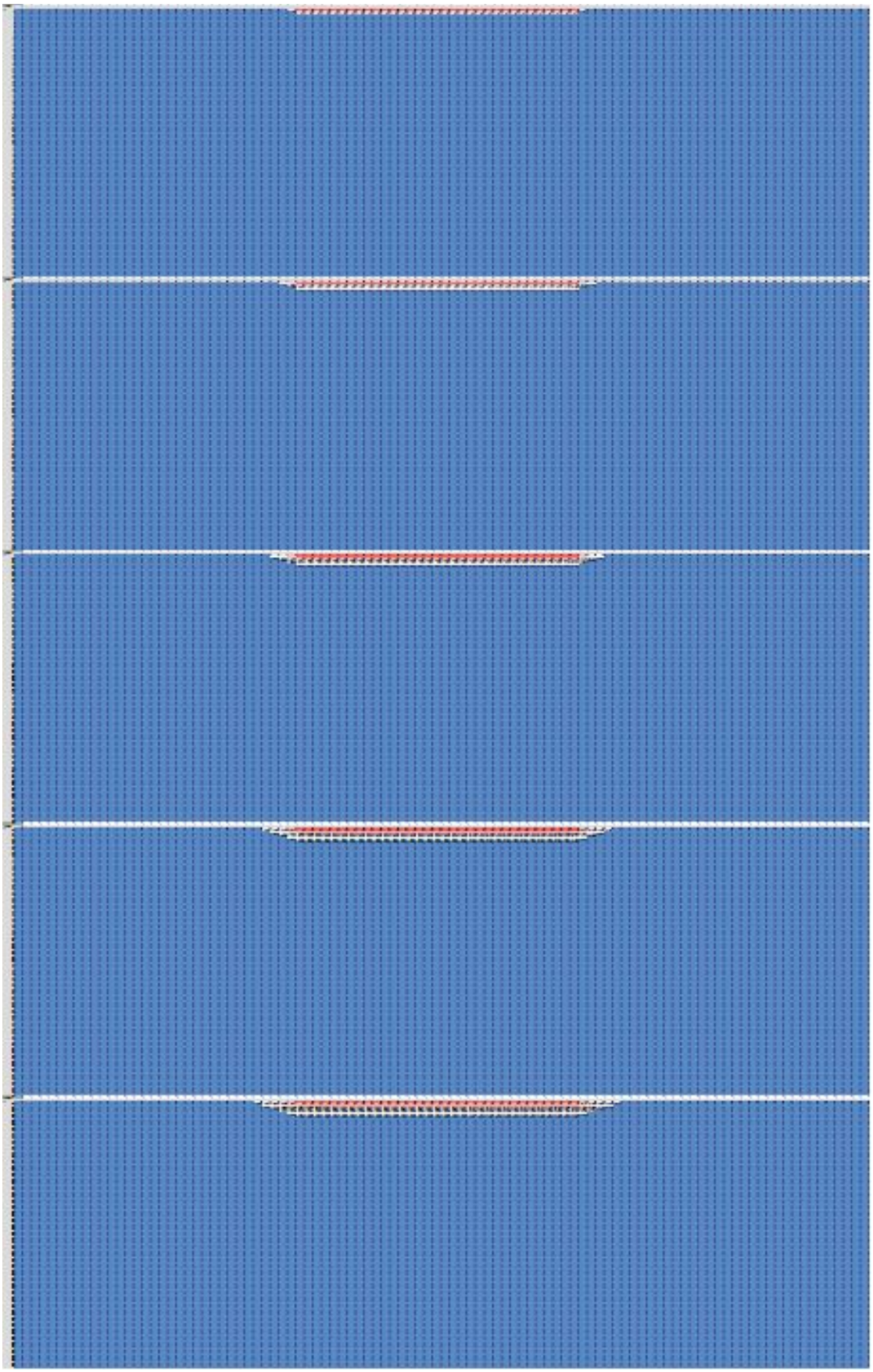
```
190     result = cudaFree(room_state1_d);
191     if (result != cudaSuccess) {
192         fprintf(stderr, "cudaFree (block) failed.");
193         exit(1);
194     }
195     result = cudaFree(room_state2_d);
196     if (result != cudaSuccess) {
197         fprintf(stderr, "cudaFree (block) failed.");
198         exit(1);
199     }
200 }
201
202
203 /*****************************************************************
204  * This kernel computes the room's temperature at the specified
205  * location after one additional time step.
206  *
207  * @param room_state_d_curr is the "next" copy of the room state.
208  * @param room_state_d_prev is the "previous" copy of the room state.
209  * @param TILES is the number of pieces the room is divided into
210  * for temperature computations.
211  * @param X_SIZE is the room width dimension.
212  * @param Y_SIZE is the room depth dimension.
213  * @param HEATER_START is the X index of the first heater tile.
214  * @param HEATER_END is the X index after the last heater tile.
215  *****************************************************************/
216 __global__ void
217 cu_room_diffusion_step(float *room_state_d_curr,
218                        float *room_state_d_prev, const int TILES,
219                        const int X_SIZE,
220                        const int Y_SIZE, const int HEATER_START,
221                        const int HEATER_END) {
222     int el = blockIdx.x * BLOCK_SIZE + threadIdx.x;
223     int i = el / X_SIZE;
224     int j = (blockIdx.x * BLOCK_SIZE + threadIdx.x) % X_SIZE;
225     const int HEATER = 100.0;
226     if (el < TILES) {
227         //case 1 - heater tiles
228         if ((i == 0) && ((j >= HEATER_START) && (j < HEATER_END))) {
229             //ignore i since we know we're in the first row
230             room_state_d_curr[el] = HEATER;
231         }
232             //case 2 - NW corner
233         else if ((i == 0) && (j == 0)) {
234             //average myself, E, and S from last time step
235             room_state_d_curr[el] = (room_state_d_prev[el] +
236                                      room_state_d_prev[(el) +
237                                                        1] +
238                                      room_state_d_prev[(el) +
239                                                        X_SIZE]) / 3;
240         }
241             //case 3 - NE corner
242         else if ((i == 0) && (j == X_SIZE - 1)) {
243             //average myself, W and S from last time step
244             room_state_d_curr[el] =
245                     ((float) (room_state_d_prev[el] +
246                               room_state_d_prev[el -
247                                                 1] +
248                               room_state_d_prev[el +
249                                                 X_SIZE])) / 3;
250         }
251             //case 4 - N border
252         else if (i == 0) {
```


WITH THIS DEGREE OF DIVERGENCE, syncthreads() = GOOD IDEA

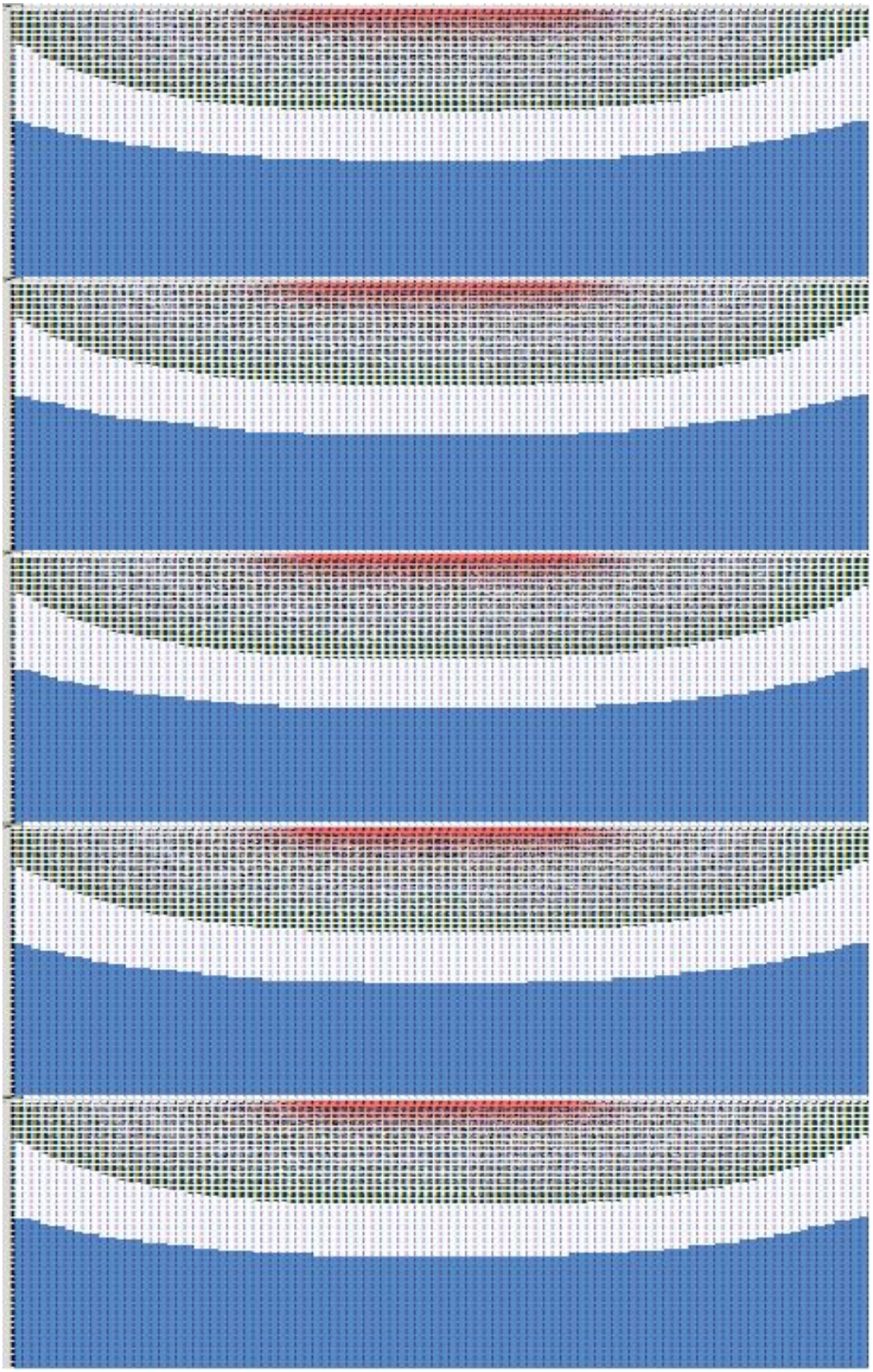
```
253             //average myself, W, E, and S from last time step
254             room_state_d_curr[el] =
255                     (room_state_d_prev[el] +
256                      room_state_d_prev[el - 1] +
257                      room_state_d_prev[el + 1] +
258                      room_state_d_prev[el + X_SIZE]) /
259                     4;
260         }
261             //case 5 - SW corner
262         else if ((i == Y_SIZE - 1) && (j == 0)) {
263             //average myself, E, and N from last time step
264             room_state_d_curr[el] =
265                     (room_state_d_prev[el] +
266                      room_state_d_prev[el +
267                                        1] +
268                      room_state_d_prev[el -
269                                        X_SIZE]) / 3;
270         }
271             //case 6 - W border
272         else if (j == 0) {
273             //average myself, E, N, and S from last time step
274             room_state_d_curr[el] =
275                     (room_state_d_prev[el] +
276                      room_state_d_prev[el +
277                                        1] +
278                      room_state_d_prev[el -
279                                        X_SIZE] +
280                      room_state_d_prev[el +
281                                        X_SIZE]) / 4;
282         }
283             //case 7 - SE corner
284         else if ((i == Y_SIZE - 1) && (j == X_SIZE - 1)) {
285             //average myself, W, and N from last time step
286             room_state_d_curr[el] =
287                     (room_state_d_prev[el] +
288                      room_state_d_prev[el -
289                                        1] +
290                      room_state_d_prev[el -
291                                        X_SIZE]) / 3;
292         }
293             //case 8 - E border
294         else if (j == X_SIZE - 1) {
295             //average myself, W, N, and S from last time step
296             room_state_d_curr[el] =
297                     (room_state_d_prev[el] +
298                      room_state_d_prev[el -
299                                        1] +
300                      room_state_d_prev[el -
301                                        X_SIZE] +
302                      room_state_d_prev[el +
303                                        X_SIZE]) / 4;
304         }
305             //case 8 - S border
306         else if (i == Y_SIZE - 1) {
307             //average myself, W, E, and N from last time step
308             room_state_d_curr[el] =
309                     (room_state_d_prev[el] +
310                      room_state_d_prev[el - 1] +
311                      room_state_d_prev[el + 1] +
312                      room_state_d_prev[el - X_SIZE]) /
313                     4;
314         }
315             //case 9 - common case of center of the room
```

```
316         else {
317             //average W, E, N, and S from last time step
318             room_state_d_curr[el] =
319                     (room_state_d_prev[el - 1] +
320                      room_state_d_prev[el + 1] +
321                      room_state_d_prev[el - X_SIZE] +
322                      room_state_d_prev[el + X_SIZE]
323                     ) / 4;
324         }
325     }
326 }
327
328
329 /*****************************************************************
330  * This helper function prints half of the room state in a "human-
331  * readable format" (only one time step is printed).
332  *
333  * @param state is the pointer to the rod model's temperature state.
334  * @param SIZE is the number of pieces the room is divided into
335  * for temperature computations.
336  * @param X_SIZE is the room width dimension.
337  * @param Y_SIZE is the room depth dimension.
338  *****************************************************************/
339 void print_room_state(const float *state, const int SIZE,
340                       const int X_SIZE, const int Y_SIZE) {
341     for (int i = 0; i < Y_SIZE; i++) {
342         for (int j = 0; j < X_SIZE; j++) {
343             cout << state[(i * X_SIZE) + j] << " ";
344         }
345         cout << endl;
346     }
347     cout << endl;
348 }
349
350
351 /*****************************************************************
352  * This kernel initializes the room's temperature.
353  *
354  * @param room_state_d is the copy of the room state.
355  * @param TILES is the number of pieces the room is divided into
356  * for temperature computations..
357  * @param HEATER_START is the X index of the first heater tile.
358  * @param HEATER_END is the X index after the last heater tile.
359  *****************************************************************/
360 __global__ void
361 cu_initialize_room(float *room_state_d, const int TILES,
362                    const int HEATER_START, const int HEATER_END) {
363     int i = blockIdx.x * BLOCK_SIZE + threadIdx.x;
364     if (i < TILES) {
365         if (i >= HEATER_START && i < HEATER_END) {
366             room_state_d[i] = HEATER_TEMP;
367         } else room_state_d[i] = INIT_TEMP;
368     }
369 }
```

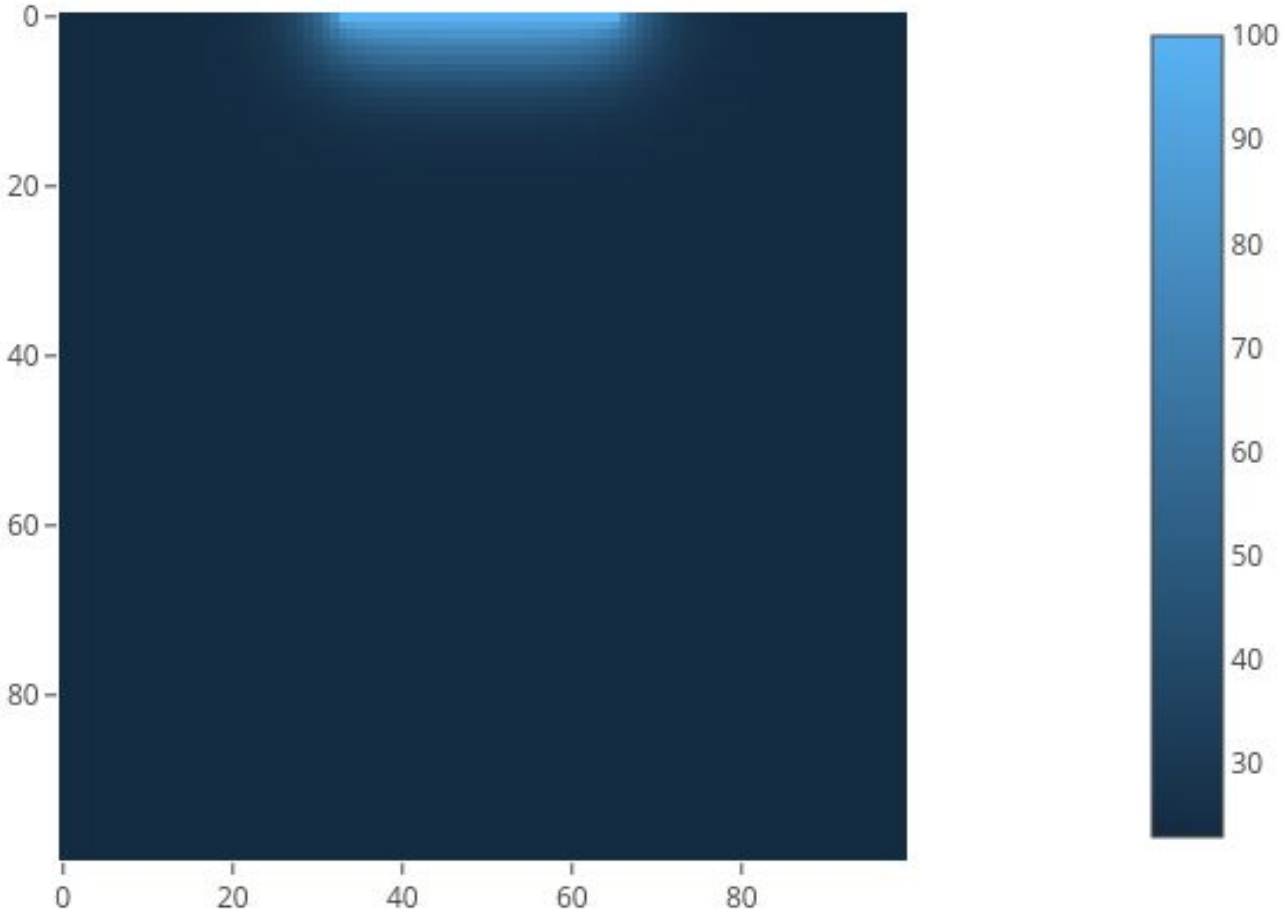
Appendix K - Microsoft Excel Model (Room 100x100 100 Timesteps)



.
.
.



Appendix L - Numpy and Plotly Visualization of Sample Output of the Sequential Room Implementation (Room 100x100 100 Timesteps)



Appendix M - Sample BASH Script for Running Performance Trials

```
 1 rm diff_rod pa3
 2 g++ -Wall -Wno-unused-variable ../untitled/main.cpp -o pa3
 3 nvcc rod.cu -o diff_rod
 4 if [ $? -eq 0 ]
 5 then
 6         for i in {1..5}
 7         do
 8         ./pa3 1 2500 10000000 1750 -1 2500 1 >> ./figures/arch_seq_rod_time_trials.txt
 9         ./diff_rod 1 2500 10000000 1750 -1 2500 1 >> ./figures/arch_cuda_rod_time_trials.txt
10         done
11 fi
```

QUICK LOOK DOES NOT REVEAL REASON FOR 2-D CUDA SLOWDOWN?

DOUBLE-CHECK RESULTS MAY REQUIRE DEEPER LOOK USING PROFILING/NSIGHT

Appendix N - Performance Trials

Sequential Rod

Trial	Time (ms)
1	110094
2	110106
3	110648
4	110416
5	110430
Average	**110338.8**

CUDA Rod

Trial	Time (ms)
1	19672.9
2	19834.9
3	19748.9
4	20183.8
5	20025.2
Average	**19893.14**

Sequential Room

Trial	Time (ms)
1	1862.75
2	1816.4
3	1843.96
4	1928.7
5	1811.59
Average	**1852.68**

CUDA Room

Trial	Time (ms)
1	93707.6
2	94113.8
3	94818.4
4	94867.4
5	94828.4
Average	**94467.12**

WOULD EXPECT PROBLEM SIZE OF ROOM > ROD HENCE GREATER TIME

OK SOMETHING IS GOING ON HERE