Scalable Scientific Computing
CMDA 4634

Heat Equation
Simulated on CUDA and Rendered on OpenGL
Fall 2024

Ishana Garuda

December 18, 2025

Contents

I Final projects **5**

1 Heat Equation Simulated on CUDA and Rendered on OpenGL **7**

1.1 Summary . . . 7

1.2 Software Installation and Setup . . . 8

1.2.1 NVIDIA Drivers and CUDA Toolkit Installation . . . 8

1.2.2 OpenGL Libraries Installation . . . 9

1.2.3 Additional Notes . . . 9

1.3 Mathematical Background . . . 10

1.3.1 General Forms of the Heat Equation . . . 10

1.3.2 Discretization . . . 12

1.3.3 Boundary Conditions . . . 14

1.3.4 Stability Considerations . . . 15

1.4 Code Implementation . . . 15

1.4.1 Data Structures & Constants. . . . 16

1.4.2 CUDA Kernels with Performance Statistics . . . 16

1.4.3 Profiling Results . . . 19

1.4.4 CUDA and OpenGL Interoperability . . . 22

1.5 Conclusion & Future Work . . . 23

1.5.1 Future Works . . . 23

1.6 References . . . 24

Part I

Final projects

Lecture 1

Heat Equation Simulated on CUDA and Rendered on OpenGL

Ishana Garuda

Contents

1.1 Summary . . . 7
1.2 Software Installation and Setup . . . 8
1.2.1 NVIDIA Drivers and CUDA Toolkit Installation . . . 8
1.2.2 OpenGL Libraries Installation . . . 9
1.2.3 Additional Notes . . . 9
1.3 Mathematical Background . . . 10
1.3.1 General Forms of the Heat Equation . . . 10
1.3.2 Discretization . . . 12
1.3.3 Boundary Conditions . . . 14
1.3.4 Stability Considerations . . . 15
1.4 Code Implementation . . . 15
1.4.1 Data Structures & Constants. . . . 16
1.4.2 CUDA Kernels with Performance Statistics . . . 16
1.4.3 Profiling Results . . . 19
1.4.4 CUDA and OpenGL Interoperability . . . 22
1.5 Conclusion & Future Work . . . 23
1.5.1 Future Works . . . 23
1.6 References . . . 24

1.1 Summary

This project showcases how optimizing CUDA code can enhance real-time applications where rendering images is essential. By leveraging the power of GPUs for both computation and rendering, we can

achieve high-performance simulations suitable for physics-based applications or game development, where efficient low-level code is crucial.

The focus is on solving the heat equation in one, two, and three dimensions using CUDA for computation and OpenGL for rendering. The simulation demonstrates how heat diffuses over time through a medium, with interactive features such as adding heat sources via mouse input.

1.2 Software Installation and Setup

This section assumes that you are running on the latest Ubuntu LTS (24.04) at the time of writing.

1.2.1 NVIDIA Drivers and CUDA Toolkit Installation

To utilize CUDA and OpenGL, it's essential to have the correct NVIDIA drivers and CUDA Toolkit installed.

NVIDIA Drivers Installation

To install the latest NVIDIA proprietary drivers and kernel modules (version 560.35.03 at the time of writing) you can do the following that is a subset of the instructions given by NVIDIA's Driver Installation Guide on Linux

```
# Add NVIDIA CUDA repository key
wget https://developer.download.nvidia.com/compute/cuda/repos/ubuntu2404/x86_64/cuda-keyring_1.1-1
    _all.deb

# Install repository key
sudo dpkg -i cuda-keyring_1.1-1_all.deb

# Install NVIDIA drivers
sudo apt-get update
sudo apt-get install cuda-drivers -y

# Reboot system
sudo reboot
```

Do note that from the previous version of NVIDIA driver installations and Ubuntu NVIDIA Drivers Installation guide this is a bit different. I would lean more towards doing what NVIDIA directly gives rather than what Ubuntu has in their documentation.

Note: Ensure you reboot after installation to load the proprietary NVIDIA kernel modules. Moreover, while the open source driver could be installed I have not used and tested it out for this project.

CUDA Toolkit Installation

For CUDA development, the NVIDIA CUDA Toolkit version 12.6 was used. The toolkit provides tools and libraries for GPU programming. I would recommend going though and fully reading as much of the installation details that can be found on NVIDIA CUDA Installation Guide for Linux. In my opinion do not just blindly run parts of the guide, try and go through the guide.

NVIDIA & CUDA Verification

After installation, verify that the drivers and toolkit are correctly set up:

```
# Check NVIDIA driver status
nvidia-smi

# Verify CUDA version
nvcc --version
```

Note: Both commands should display the appropriate versions and indicate the system is ready for CUDA development.

1.2.2 OpenGL Libraries Installation

For CUDA-OpenGL interoperability or general OpenGL development, you need to install the necessary OpenGL libraries. The following steps detail the installation process on Ubuntu 24.04:

```
# Update package index
sudo apt-get update

# Install OpenGL development libraries
sudo apt-get install -y mesa-utils libgl1-mesa-dev libglu1-mesa-dev

# Install GLFW (for creating OpenGL contexts)
sudo apt-get install -y libglfw3-dev

# Install GLEW (for handling OpenGL extensions)
sudo apt-get install -y libglew-dev
```

Verifying OpenGL Installation

To verify that OpenGL is correctly installed, you can use the following commands:

```
# Check OpenGL version
glxinfo | grep "OpenGL"

# Run a simple OpenGL test
glxgears
```

Note: The `glxgears` command should display a spinning gear animation, confirming that OpenGL is functioning.

1.2.3 Additional Notes

The libraries listed above provide the essential components for OpenGL development on Linux:

- `mesa-utils`: Includes utilities for OpenGL testing and information.
- `libgl1-mesa-dev`: Provides the Mesa OpenGL library for development.
- `libglu1-mesa-dev`: Adds the OpenGL Utility Library (GLU) for compatibility with older OpenGL applications.

- `libglfw3-dev`: A library for managing OpenGL contexts and windowing.
- `libglew-dev`: Handles extensions, ensuring compatibility with modern OpenGL features.

These libraries are particularly important when developing applications that leverage both CUDA and OpenGL for rendering or interoperability.

1.3 Mathematical Background

The Fourier heat equation (also known as the diffusion equation) is a partial differential equation (PDE) that describes the distribution of heat (or variation in temperature) in a given region over time. It is fundamental in the study of heat transfer and diffusion processes over space and time.

1.3.1 General Forms of the Heat Equation

We consider the heat equation in one, two, and three dimensions. The given equations in continuous form, and we are assuming non-homogenous heat sources.

- **1D Heat Equation** in the general form looks like:

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2} + Q(x,t) \tag{1.1}$$

  where:

  - u is actually shorted from $u(x,t)$ with temperature at position x and time t.
  - $x \in [0, W]$ where W is the maximum width in the x dimension.
  - A boundary condition of either:
    - Dirichlet:
$$u(0,t) = b_1,\ u(W,t) = b_2$$
    - Neumann:
$$\frac{\partial}{\partial x}u(0,t) = b_1,\ \frac{\partial}{\partial x}u(W,t) = b_2$$
  - $u(x,0) = 0$ the starting conditions of the space is just 0 temperature for simplicity. However, this does not need to be the case.
  - α: Thermal diffusivity constant.
  - $Q(x,t) = \begin{cases} c \text{ if heat is being added} \\ 0 \text{ otherwise} \end{cases}$. The heat source term with some constant c that indicates how much heat is being added to the system externally at position x and time t.

- **2D Heat Equation** in the general form looks like:

$$\frac{\partial u}{\partial t} = \alpha \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) + Q(x,y,t) \tag{1.2}$$

  where:

- u is actually shorted from $u(x, y, t)$ with temperature at point (x, y) and time t.
- $x \in [0, W], y \in [0, H]$ where W is the maximum width in the x dimension, and H is the maximum height in the y dimention
- A boundary condition of either:
  - Dirichlet:
$$u(0, y, t) = b_1,\ u(W, y, t) = b_2,$$
$$u(x, 0, t) = b_3,\ u(x, H, t) = b_3$$
  - Neumann:
$$\frac{\partial}{\partial x}u(0, y, t) = b_1,\ \frac{\partial}{\partial x}u(W, y, t) = b_2,$$
$$\frac{\partial}{\partial x}u(x, 0, t) = b_3,\ \frac{\partial}{\partial x}u(x, H, t) = b_4$$
- $u(x, 0) = 0$ the starting conditions of the space is just 0 temperature for simplicity. However, this does not need to be the case.
- α: Thermal diffusivity constant.
- $Q(x, t) = \begin{cases} c \text{ if heat is being added} \\ 0 \text{ otherwise} \end{cases}$. The heat source term with some constant c that indicates how much heat is being added to the system externally at position (x, y) and time t.

- **3D Heat Equation** is again extending this to give:
$$\frac{\partial u}{\partial t} = \alpha\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}\right) + Q(x, y, z, t) \tag{1.3}$$

where:

- u is actually shorted from $u(x, y, z, t)$ with temperature at point (x, y, z) and time t.
- $x \in [0, W], y \in [0, H], z \in [0, D]$ where W is the maximum width in the x dimension, H is the maximum height in the y dimension, and D is the maximum depth in the z dimension.
- A boundary condition of either:
  - Dirichlet:
$$u(0, y, z, t) = b_1,\ u(W, y, z, t) = b_2,$$
$$u(x, 0, z, t) = b_3,\ u(x, H, z, t) = b_3,$$
$$u(x, y, 0, t) = b_4,\ u(x, y, D, t) = b_5$$
  - Neumann:
$$\frac{\partial}{\partial x}u(0, y, z, t) = b_1,\ \frac{\partial}{\partial x}u(W, y, z, t) = b_2,$$
$$\frac{\partial}{\partial x}u(x, 0, z, t) = b_3,\ \frac{\partial}{\partial x}u(x, H, z, t) = b_4$$
$$\frac{\partial}{\partial x}u(x, y, 0, t) = b_5,\ \frac{\partial}{\partial x}u(x, y, D, t) = b_6$$
- $u(x, 0) = 0$ the starting conditions of the space is just 0 temperature for simplicity. However, this does not need to be the case.
- α: Thermal diffusivity constant.
- $Q(x, t) = \begin{cases} c \text{ if heat is being added} \\ 0 \text{ otherwise} \end{cases}$. The heat source term with some constant c that indicates how much heat is being added to the system externally at position (x, y, z) and time t.

1.3.2 Discretization

To solve the heat equation numerically, we discretize the spatial and temporal domains using Finite Difference methods.

Some ongoing useful notation that I will be using is the labels for where the time step, and spacial dimension indexing is going to be. For a given vector u that represents the spacial discretization of space is the sub script next to u, where the i index represents the x axis/dim, j index represents the y axis/dim, and k index represents the z axis/dim. As for the time dimension, it will be given as a super script for the vector u where n and $n+1$ are going to be the common states that represent the current time step, and next time step respectively.

Explicit method

The discretization method we are going over here can be looked a bit more at from Wikipedia's Explicit Method. The basic idea is to try to use the current point at some coordinate (i, j, k in the 3D case, for example), and explicitly compute the next time step at that coordinate using the adjacent spatial points (such as the forward, backward, left, right, up, down points in the 3D case again). Such a scheme is also a Forward Time-Centered Space (FTCS) scheme. Other methods such as the Implicit Method or Crank–Nicolson Method could be implemented, however they have their own set of trade-offs regarding temporal step sizes, computational errors, and accuracy against the true solution of the continuous space-time PDE it is trying to solve.

Spatial Discretization

We can approximate the second derivatives using central differences as discussed above. For that we introduce grid spacings Δx, Δy, and Δz for each spatial dimension; these in essence are step sizes in each dimension. For the $u(x, y, z, t)$ function we can move by such a grid spacing, and show it as moving in the x-dim by a positive step size of Δx as: $u(x + \Delta x, y, z, t) = u^n_{i+1,y,z}$

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u^n_{i+1,j,k} - 2u^n_{i,j,k} + u^n_{i-1,j,k}}{(\Delta x)^2} \tag{1.4}$$

$$\frac{\partial^2 u}{\partial y^2} \approx \frac{u^n_{i,j+1,k} - 2u^n_{i,j,k} + u^n_{i,j-1,k}}{(\Delta y)^2} \tag{1.5}$$

$$\frac{\partial^2 u}{\partial z^2} \approx \frac{u^n_{i,j,k+1} - 2u^n_{i,j,k} + u^n_{i,j,k-1}}{(\Delta z)^2} \tag{1.6}$$

Do note that as we move down in dimentions we can just get rid of indeices and

Temporal Discretization

Use a forward difference for the time derivative:

$$\frac{\partial u}{\partial t} \approx \frac{u_{i,j,k}^{n+1} - u_{i,j,k}^{n}}{\Delta t} \tag{1.7}$$

Discretized Heat Equation

Combining the spatial and temporal discretizations, we derive the update formula in all the 3 dimensions we discussed above. Note that for all three of the final equations we will try and optimize the formulation itself to reduce the use of floating point instructions by doing some algebra to simplify the equation as much as possible by hand, and try and optimize as much for Floating-Point Multiply-Add/Accumulate instructions `FMA`.

Some constant changes we will use to all the equations is assume that $h = \Delta x = \Delta y = \Delta z$. Then let $r = \frac{\Delta t\,\alpha}{h^2}$. Also since we are not going to be dealing with the adding of non-homogenous heat in the main discretized as we can just think of it as a separate check and addition that we just add some heat at certain spaces of u, so lets remove it for now. But for the sake of completion, the basic equation would look like: $u_{i,j,k}^{n+1} = u_{i,j,k}^{n} + \Delta t\, Q_{i,j,k}$ meaning that either at that certain coordinate there is some heat being added by c if we decide that.

So for the simplified, and looking at this heat equation PDE in a more homogenous setting, we can say that the update formulations are going to look like the following.

- **1D Update Formulation:** Given equation (1.4) and (1.7) we can combine them and solve for u_i^{n+1}.

$$\begin{aligned} u_i^{n+1} &= u_i^n + \Delta t\,\alpha \left(\frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{(\Delta x)^2} \right) \\ &= u_i^n + r(u_{i+1}^n - 2u_i^n +) \\ &= (1-2r)u_i^n + r(u_{i+1}^n + u_{i-1}^n) \end{aligned}$$

  We can also redefine some of these neighboring u_i variables to make it easier to see how it all fits together. We can assign:

  - $u_i \implies u_c$ for center
  - $u_{i+1} \implies u_r$ for right
  - $u_{i-1} \implies u_l$ for left

$$u_c^{n+1} = (1-2r)u_c^n + r(u_r^n + u_l^n) \tag{1.8}$$

- **2D Update Formulation:** Similarly we can use equation (1.4), (1.5) and (1.7) to combine and solve for $u_{i,j}^{n+1}$.

$$\begin{aligned} u_{i,j}^{n+1} &= u_{i,j}^n + \Delta t\,\alpha \left(\frac{u_{i+1,j}^n - 2u_{i,j}^n + u_{i-1,j}^n}{(\Delta x)^2} + \frac{u_{i,j+1}^n - 2u_{i,j}^n + u_{i,j-1}^n}{(\Delta y)^2} \right) \\ &= u_{i,j}^n + r(-4u_{i,j}^n + u_{i+1,j}^n + u_{i-1,j}^n + u_{i,j+1}^n + u_{i,j-1}^n) \\ &= (1-4r)u_{i,j}^n + r(u_{i+1,j}^n + u_{i-1,j}^n + u_{i,j+1}^n + u_{i,j-1}^n) \end{aligned}$$

  Again lets simplify some of the neighbors of $u_{i,j}$ variables to make it easier to see how it all fits together.

- $u_{i,j} \implies u_c$ for center
- $u_{i+1,j} \implies u_r$ for right
- $u_{i-1,j} \implies u_l$ for left
- $u_{i,j+1} \implies u_u$ for up
- $u_{i,j-1} \implies u_d$ for down

$$u_c^{n+1} = (1-4r)u_c^n + r(u_r^n + u_l^n + u_u^n + u_d^n) \tag{1.9}$$

- **3D Update Formulation:** Finally the we use all the equations (1.4), (1.5), (1.6) and (1.7) to combine and solve for $u_{i,j}^{n+1}$.

$$\begin{aligned} u_{i,j,k}^{n+1} &= u_{i,j,k}^n + \Delta t\,\alpha\left(\frac{u_{i+1,j,k}^n - 2u_{i,j,k}^n + u_{i-1,j,k}^n}{(\Delta x)^2} + \frac{u_{i,j+1,k}^n - 2u_{i,j,k}^n + u_{i,j-1,k}^n}{(\Delta y)^2} + \frac{u_{i,j,k+1}^n - 2u_{i,j,k}^n + u_{i,j,k-1}^n}{(\Delta z)^2}\right) \\ &= u_{i,j,k}^n + r(-6u_{i,j,k}^n + u_{i+1,j,k}^n + u_{i-1,j,k}^n + u_{i,j+1,k}^n + u_{i,j-1,k}^n + u_{i,j,k+1}^n + u_{i,j,k-1}^n) \\ &= (1-4r)u_{i,j}^n + r(u_{i+1,j,k}^n + u_{i-1,j,k}^n + u_{i,j+1,k}^n + u_{i,j-1,k}^n + u_{i,j,k+1}^n + u_{i,j,k-1}^n) \end{aligned}$$

Simplifying for neighbors of $u_{i,j,k}$.

- $u_{i,j} \implies u_c$ for center
- $u_{i+1,j,k} \implies u_r$ for right
- $u_{i-1,j,k} \implies u_l$ for left
- $u_{i,j+1,k} \implies u_u$ for up
- $u_{i,j-1,k} \implies u_d$ for down
- $u_{i,j,k+1} \implies u_f$ for forward
- $u_{i,j,k-1} \implies u_b$ for backward

$$u_c^{n+1} = (1-6r)u_c^n + r(u_r^n + u_l^n + u_u^n + u_d^n + u_f^n + u_b^n) \tag{1.10}$$

1.3.3 Boundary Conditions

Boundary conditions are essential to solving the heat equation. In this simulation I am trying to showcase we will use two types of boundary conditions. Note that while each point/edge/face of the different dimensions can be different, for the sake of simplicity we are setting them all to the same value. So, that is just saying for the different cases above in Section 1.3.1 we are setting $b = b_1, \ldots, b_6$. Now we can look at what the conversion of the continuous to discrete/discretized looks like for the two boundary conditions. Also, for simplicity we will just look at the 1D case; as the same basic math will apply in all the cases.

- **Dirichlet Boundary Condition** is going from being set up as

$$u(0,t) = b,\ u(W,t) = b,$$

to look like

$$u_0^n = b,\ u_W^n = b.$$

However, note that we will simplify this further by setting $b = 0$ in the Dirichlet case.

- **Neumann Boundary Condition** is going from being set up as

$$\frac{\partial}{\partial x}u(0,t) = b,\ \frac{\partial}{\partial x}u(W,t) = b$$

to look like either a Forward Difference or Backward difference. We have an example of a forward difference for time in equation (1.7). The main difference is that we are taking the partial with respect to a spacial derivative. So for the left boundary at 0 we can see that it will use a forward difference

$$\begin{aligned} &\frac{\partial}{\partial x}u(0,t) = b \\ \implies &\frac{u(0+\Delta x,t) - u(0,t)}{\Delta x} = b \\ \implies &\frac{u_1^n - u_0^n}{h} = b \\ \implies &u_0^n = h\,b + u_1^n \end{aligned}$$

And for the right boundary at W we are going to use a backward difference

$$\begin{aligned} &\frac{\partial}{\partial x}u(W,t) = b \\ \implies &\frac{u(W,t) - u(W-\Delta x,t)}{\Delta x} = b \\ \implies &\frac{u_W^n - u_{W-1}^n}{h} = b \\ \implies &u_W^n = h\,b + u_{W-1}^n \end{aligned}$$

1.3.4 Stability Considerations

The choice of time step Δt is critical for the stability of the explicit finite difference method. The following condition must be satisfied:

$$\Delta t \leq \frac{1}{2\alpha}\left(\frac{1}{(\Delta x)^{-2} + (\Delta y)^{-2} + (\Delta z)^{-2}}\right) \tag{1.11}$$

In the code, `TIME_STEP` is calculated based on this criterion.

1.4 Code Implementation

The code is structured to efficiently simulate the heat equation using CUDA for computation and OpenGL for rendering. The actual executable can also be modified between the different spacial dimentions, and boundary conditions. For profiling tools such as NVIDIA Nsight Compute and Nsight Systems were used to analyze performance and guide optimizations both in command line interfaces, and their UI-UX applications.

1.4.1 Data Structures & Constants.

I use flattened 1D arrays to store multi-dimensional grids rather than subsetting multiple times and allocating memory is much easier too. Moreover, this helps to grab pages of the matrix in memory together in theory and can help to move data. The use of indexing helper functions like `IDX_2D` and `IDX_3D` can be used to map multi-dimensional indices to linear indices given a set of x, y, z coordinates, and their respective width (w) and height (h). Moreover these functions were defined via CUDA macros rather than true function calls so that it could be recompiled as much as possible during compilation time, rather than being calculated at each point in runtime.

```
#define IDX_2D(x, y, width) ((y) * (width) + (x))
#define IDX_3D(x, y, z, width, height) ((z) * (width) * (height) + (y) * (width) + (x))
```

Similar regard was given to setting up the box, width, height, depth, and other constants needed by the simulation kernels.

```
#define WIDTH 1000
#define HEIGHT 1000
#define DEPTH 100
#define HEAT_RADIUS 5
#define HEAT_SOURCE 5.0f
#define H 1.0f
#define H2 (H * H)
#define DIFFUSIVITY 1.0f
#define TIME_STEP (H2 * H2 * H2 / (2 * DIFFUSIVITY * (H2 + H2 + H2)))
#define RATIO (TIME_STEP * DIFFUSIVITY / H)
```

Where we can see parts of these are the constants are what we discussed in the math background section.

1.4.2 CUDA Kernels with Performance Statistics

The CUDA implementation is done via kernels that is divided into 3 major kernels, each targeting a specific aspect of the simulation:

- One in each dimension for simulation, `heat_kernel_*d_sim`. This is the most complex kernel as it is trying to optimize the actual math behind simulating a PDE and data movement the most.
- One in each dimension to dynamically add heat into the GPU allocated memory `add_heat_kernel_*d`.
- One in each dimension to convert a set of 32 bit floating point numbers into a `char4` which is a type defined by CUDA that holds 4 16-bit integers that define a color, as each red, green, blue, and alpha (transparency) values are between 0 and 255.

Shared Memory Optimization in the 2D Kernel

All of the simulation (`heat_kernel_*d_sim`) kernels use shared memory to store data required to compute their respective next time step for a given spacial coordinate. This helps to reduce global memory accesses and store memory for threads to share between each other. Below is the shared memory declaration and data loading segment. The reason for `extern` is so that when the kernel is launched the amount of memory needed can be allocated.

```
extern __shared__ float s_u[];
const int shared_bs_x = blockDim.x + 2;

int x = blockIdx.x * blockDim.x + threadIdx.x;
int y = blockIdx.y * blockDim.y + threadIdx.y;

int s_x = threadIdx.x + 1;
int s_y = threadIdx.y + 1;

// Load central cell
s_u[IDX_2D(s_x, s_y, shared_bs_x)] = u0[IDX_2D(x, y, WIDTH)];
```

Borders are then loaded conditionally, ensuring only threads on the block's edge perform additional work. This does lead to some warp divergence, and while there are a few different work around such as launching the kernel with more threads, allocating more memory and having duplicate boundaries and other options, this is what I ended up with.

```
// Left border
if (threadIdx.x == 0 && x > 0)
    s_u[IDX_2D(0, s_y, shared_bs_x)] = u0[IDX_2D(blockIdx.x*blockDim.x - 1, y, WIDTH)];
// Right border
if (threadIdx.x == blockDim.x - 1 && x < WIDTH - 1)
    s_u[IDX_2D(blockDim.x + 1, s_y, shared_bs_x)] = u0[IDX_2D((blockIdx.x + 1)*blockDim.x, y,
        WIDTH)];
// Top border
if (threadIdx.y == 0 && y > 0)
    s_u[IDX_2D(s_x, 0, shared_bs_x)] = u0[IDX_2D(x, blockIdx.y*blockDim.y - 1, WIDTH)];
// Bottom border
if (threadIdx.y == blockDim.y - 1 && y < HEIGHT - 1)
    s_u[IDX_2D(s_x, blockDim.y + 1, shared_bs_x)] = u0[IDX_2D(x, (blockIdx.y + 1)*blockDim.y,
        WIDTH)];
// Synchronize threads after loading
__syncthreads();
```

This approach tries to minimize the need of extra true global memory, and minimizing the amount of threads that are used and launched. Nevertheless, we can look at the `ncu-ui` which will display a diagram of the source code.

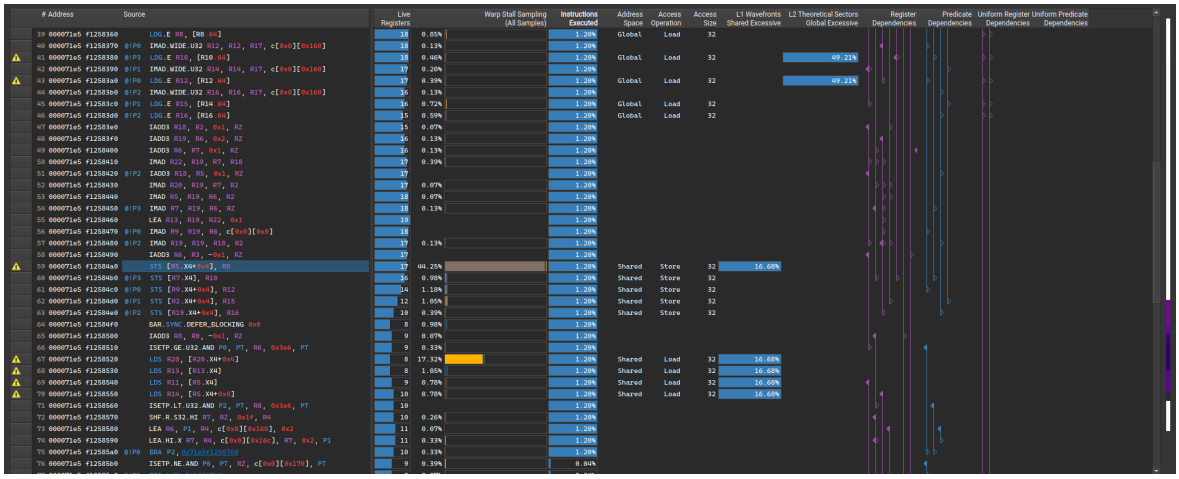

Figure 1.1: Source of 2D Heat Simulation Kernel Showcased for Shared Memory Access from NVIDIA Nsight Compute

From this we can see that there is some warp stalling, and there are further optimizations that Nsight Compute suggests to look at from CUDA documentation and best practices. However, this is graph

and information is much harder to digest and try and understand what it is doing/suggesting how to optimize my code further.

Optimized 3D Heat Kernel

The 3D kernel processes the grid in slices, reducing register pressure and improving memory access patterns. Techniques from Paulius Micikevicius's work were adopted to balance shared memory and register usage.

We can see this from the summary statistics of ncu on how many registers were used.

- For `heat_kernel_2d_sim` we see that the usage of registers look like:

```
Section: Launch Statistics
-------------------------------- --------------- ------------ ------------ ------------
Metric Name                          Metric Unit      Minimum      Maximum      Average
-------------------------------- --------------- ------------ ------------ ------------
Block Size                                             256.00       256.00       256.00
Grid Size                                            3,969.00     3,969.00     3,969.00
Registers Per Thread             register/thread        25.00        25.00        25.00
Shared Memory Configuration Size           Kbyte        32.77        32.77        32.77
Driver Shared Memory Per Block       Kbyte/block         1.02         1.02         1.02
Dynamic Shared Memory Per Block      Kbyte/block         1.30         1.30         1.30
Static Shared Memory Per Block        byte/block         0.00         0.00         0.00
# SMs                                        SM        34.00        34.00        34.00
Threads                                  thread 1,016,064.00 1,016,064.00 1,016,064.00
Uses Green Context                                     0.00         0.00         0.00
Waves Per SM                                          19.46        19.46        19.46
-------------------------------- --------------- ------------ ------------ ------------
```

- For `heat_kernel_3d_sim` we see that the usage of registers look like:

```
Section: Launch Statistics
-------------------------------- --------------- ------------ ------------ ------------
Metric Name                          Metric Unit      Minimum      Maximum      Average
-------------------------------- --------------- ------------ ------------ ------------
Block Size                                             256.00       256.00       256.00
Grid Size                                            3,969.00     3,969.00     3,969.00
Registers Per Thread             register/thread        29.00        29.00        29.00
Shared Memory Configuration Size           Kbyte        32.77        32.77        32.77
Driver Shared Memory Per Block       Kbyte/block         1.02         1.02         1.02
Dynamic Shared Memory Per Block      Kbyte/block         1.30         1.30         1.30
Static Shared Memory Per Block        byte/block         0.00         0.00         0.00
# SMs                                        SM        34.00        34.00        34.00
Threads                                  thread 1,016,064.00 1,016,064.00 1,016,064.00
Uses Green Context                                     0.00         0.00         0.00
Waves Per SM                                          19.46        19.46        19.46
-------------------------------- --------------- ------------ ------------ ------------
```

From this we can see that we only use 4 more registers to simulate the 3D heat equation after the compiler helps optimize the true low level code that is given to the GPU.

Trying to Optimize `FMAs`

From the math background when we did all the algebra to try and simplify stuff, we can see that most of the FMAs that we can try and squeeze out isn't much. As in the 3D case this is the instruction each thread is doing to calculate the next time step.

Figure 1.2: Source of 3D Heat Simulation Kernel Showcased for FMAs from NVIDIA Nsight Compute

Do note that I set h which are the spacial distances $(\Delta x, \dots)$ to 1, so that might be the reason we just see a ton of ADDs, rather than FMAs.

1.4.3 Profiling Results

We can look at roofline models for all the kernels I implemented for the 3 different spacial dimensions running on a local Desktop with a RTX 4060 Ti 16 GB card.

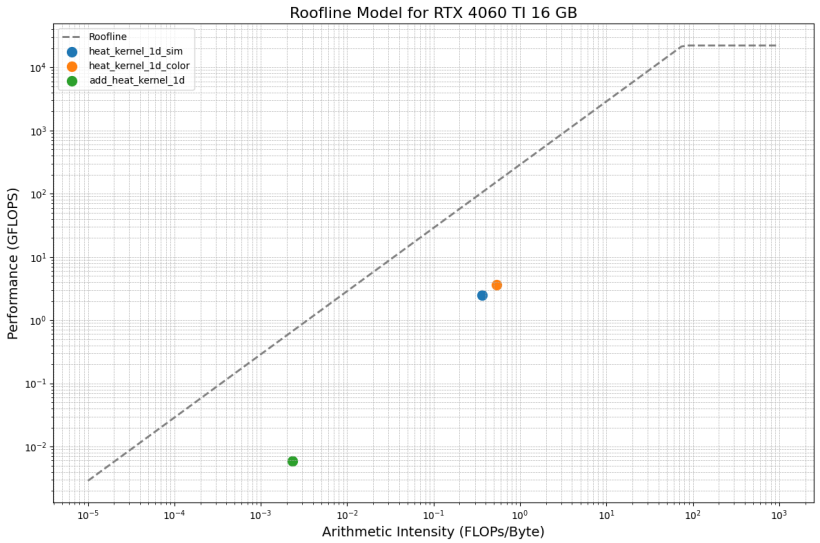

Figure 1.3: 1D Roofline Model

From this Figures 1.3 we can see that only in the 1D case the actual data size was not large enough of a problem size to fully use the GPU. From the below `ncu` output for the `heat_kernel_1d_sim` we can see the same idea too that the problem size is too small. It hardly uses 3% of the memory throughput on average, and not even 1% of the compute throughput in max.

```
heat_kernel_1d_sim(float *, float *, BoundaryCondition) (2, 1, 1)x(512, 1, 1), Device 0, CC 8.9,
      Invocations 250
  Section: GPU Speed Of Light Throughput
  ---------------------- ----------- -------- -------- --------
  Metric Name            Metric Unit  Minimum  Maximum  Average
  ---------------------- ----------- -------- -------- --------
  DRAM Frequency                 Ghz     8.41     9.07     8.76
  SM Frequency                   Ghz     2.21     2.29     2.25
  Elapsed Cycles               cycle 4,133.00 6,211.00 4,327.23
  Memory Throughput                %     0.96    19.93     3.04
  DRAM Throughput                  %     0.96    19.93     2.98
  Duration                        us     1.82     2.72     1.92
```

```
    L1/TEX Cache Throughput           %      3.80     18.19     17.49
    L2 Cache Throughput               %      0.46      3.40      0.93
    SM Active Cycles              cycle    115.15    551.74    121.18
    Compute (SM) Throughput           %      0.29      0.51      0.50
    ----------------------- ----------- -------- -------- --------
```

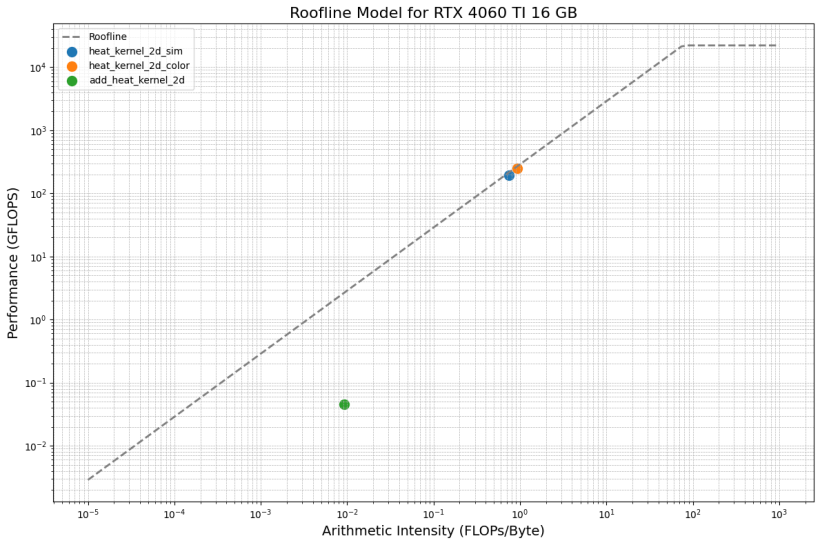

Figure 1.4: 2D Roofline Model

Comparatively we can see the summary of `heat_kernel_2d_sim` is much more intersting with it using 90.15% of memory throughput, and 71.66% of L1 Cache Throughput as the problem size is not too big yet, so most of the data can fit in L1.

```
heat_kernel_2d_sim(float *, float *, BoundaryCondition) (63, 63, 1)x(16, 16, 1), Device 0, CC
    8.9, Invocations 250
  Section: GPU Speed Of Light Throughput
  ----------------------- ----------- --------- ---------- ---------
  Metric Name             Metric Unit   Minimum    Maximum   Average
  ----------------------- ----------- --------- ---------- ---------
  DRAM Frequency                  Ghz      8.96       8.99      8.98
  SM Frequency                    Ghz      2.21       2.28      2.25
  Elapsed Cycles                cycle 52,045.00 190,277.00 70,194.12
  Memory Throughput                 %     40.62      92.52     90.15
  DRAM Throughput                   %     40.62      92.52     90.15
  Duration                         us     23.49      83.26     31.21
  L1/TEX Cache Throughput           %     39.04      83.99     71.66
  L2 Cache Throughput               %     14.19      44.38     34.56
  SM Active Cycles              cycle 39,826.79  85,667.50 49,335.74
  Compute (SM) Throughput           %     28.63      79.18     64.66
  ----------------------- ----------- --------- ---------- ---------

  Section: Occupancy
  ------------------------------ ----------- ------- ------- -------
  Metric Name                    Metric Unit Minimum Maximum Average
  ------------------------------ ----------- ------- ------- -------
  Block Limit SM                       block   24.00   24.00   24.00
  Block Limit Registers                block    8.00    8.00    8.00
  Block Limit Shared Mem               block   13.00   13.00   13.00
  Block Limit Warps                    block    6.00    6.00    6.00
  Theoretical Active Warps per SM       warp   48.00   48.00   48.00
  Theoretical Occupancy                    %  100.00  100.00  100.00
  Achieved Occupancy                       %   70.97  436.51  128.03
  Achieved Active Warps Per SM          warp   34.06  209.52   61.45
  ------------------------------ ----------- ------- ------- -------
```

As for occupancy and what ncu was able to record the achieved values are seem wrong, as it goes above 100%. On the Nsight Compute UI interface it does not give a summary of all the kernels together, but after going through a few of them I saw that many of them get close to 95% $\sim$ 100% achieved occupancy.

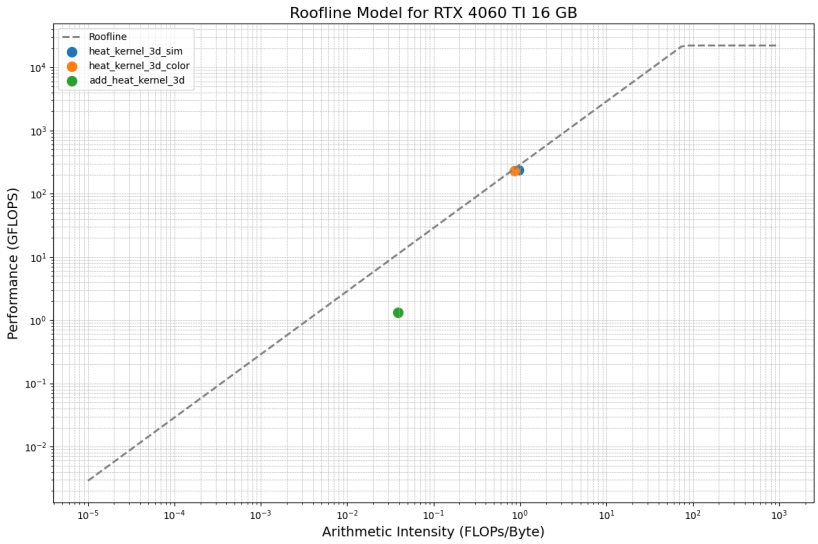

Figure 1.5: 3D Roofline Model

Do note that in the other two cases (2D and 3D), we see from Figures 1.4 and 1.5, the main computational heavier kernels reach the memory bound. So, we can tell that the PDE simulation is a memory bound function rather than a compute bound on the 4060 GPU this was run on with the explicit time stepping method used with the talked about finite differences.

Finally we can also look at the statistics of the 3D kernel.

```
heat_kernel_3d_sim(float *, float *, BoundaryCondition) (63, 63, 1)x(16, 16, 1), Device 0, CC
    8.9, Invocations 20
  Section: GPU Speed Of Light Throughput
  ----------------------- ----------- ------------ ------------ ------------
  Metric Name             Metric Unit      Minimum      Maximum      Average
  ----------------------- ----------- ------------ ------------ ------------
  DRAM Frequency                  Ghz         8.99         8.99         8.99
  SM Frequency                    Ghz         2.30         2.31         2.30
  Elapsed Cycles                cycle 7,169,031.00 7,676,084.00 7,419,719.00
  Memory Throughput                 %        85.68        88.78        87.25
  DRAM Throughput                   %        85.68        88.78        87.25
  Duration                         ms         3.11         3.34         3.22
  L1/TEX Cache Throughput           %        35.52        38.56        36.88
  L2 Cache Throughput               %        30.72        32.54        31.79
  SM Active Cycles              cycle 7,110,203.47 7,718,797.35 7,440,148.71
  Compute (SM) Throughput           %        35.27        38.38        36.59
  ----------------------- ----------- ------------ ------------ ------------

  Section: Occupancy
  ------------------------------ ----------- ------- ------- -------
  Metric Name                    Metric Unit Minimum Maximum Average
  ------------------------------ ----------- ------- ------- -------
  Block Limit SM                       block   24.00   24.00   24.00
  Block Limit Registers                block    8.00    8.00    8.00
  Block Limit Shared Mem               block   13.00   13.00   13.00
  Block Limit Warps                    block    6.00    6.00    6.00
  Theoretical Active Warps per SM       warp   48.00   48.00   48.00
  Theoretical Occupancy                    %  100.00  100.00  100.00
  Achieved Occupancy                       %   93.26  106.48   98.43
```

```
    Achieved Active Warps Per SM           warp    44.77    51.11    47.25
    ------------------------------ ----------- -------- -------- --------
```

Here we see we get to maximum occupancy for the 3D simulation kernel too, and it seems to say that we somehow got above the theoretical max too, which doesn't make total sense. Regardless, we have a high memory throughput like the 2D version, but we have a lower compute throughput per SM as each layer of the z dimension is mapped to a for loop on a thread rather that getting its own thread. So, in that regard, it might be better to work with a 3D block and have a bit higher register usage to gain a better bump in compute throughput, and possibly even get a higher bump in L1 cache throughput.

1.4.4 CUDA and OpenGL Interoperability

By leveraging CUDA-OpenGL interoperability, we perform both computation and rendering on the GPU, minimizing data transfer overheads.

Pixel Buffer Objects (PBOs)

CUDA graphics resources are mapped to OpenGL Pixel Buffer Objects or a OpenGL 3D Volume Texture, allowing CUDA kernels to write simulation results directly into buffers used for rendering after they are done executing.

Unified GPU Computation and Rendering

By performing both computation and rendering on the GPU, we eliminate the need for data transfer over the PCIe bus, reducing latency.

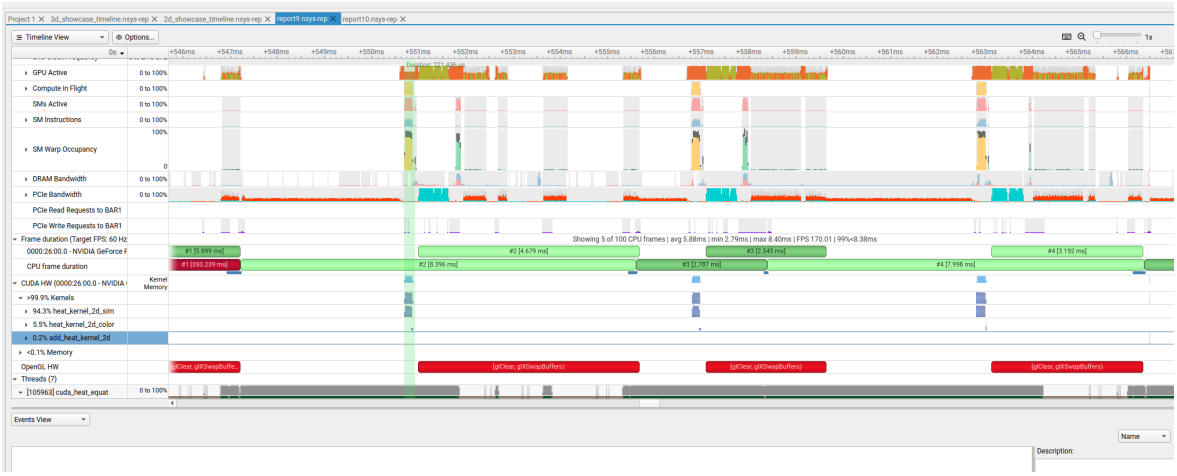

Figure 1.6: 2D Nsight Systems Timeline of CUDA Kernel and OpenGL API Execution Time

Here we can see that clearly the CUDA code and running it 10 time is beneficial as there is a lot of

time we can mask the kernel calls with the OpenGL rendering.

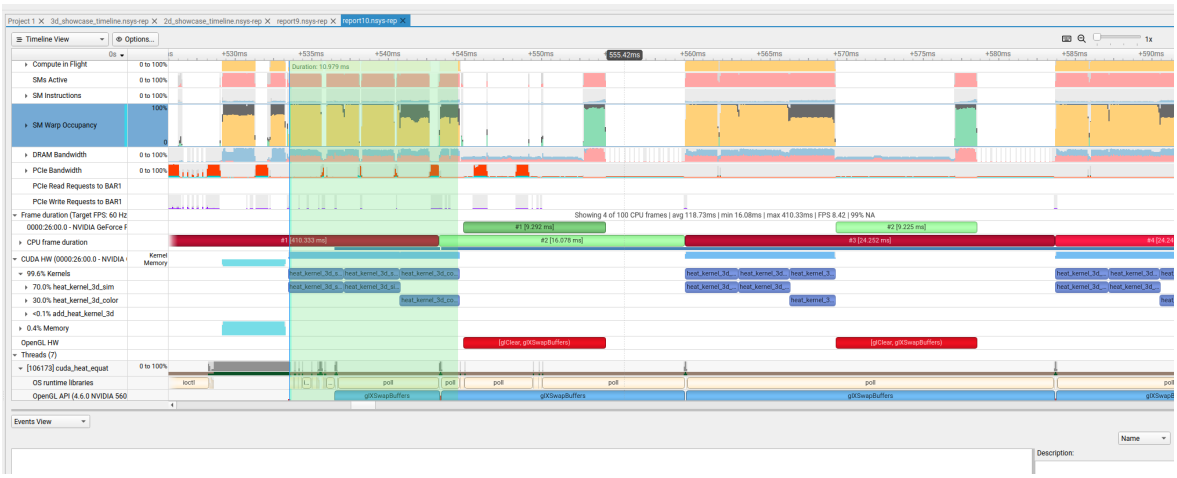

Figure 1.7: 3D Nsight Systems Timeline of CUDA Kernel and OpenGL API Execution Time

However, from this we can see that the time it takes to execute the kernel is much longer and is the reason why we can see the FPS drop when running the code from 2D to 3D.

1.5 Conclusion & Future Work

This project demonstrates how optimized CUDA code can significantly enhance performance in real-time simulations that require both computation and rendering. By leveraging CUDA and OpenGL interoperability and implementing strategic optimizations, we achieve real-time visualization of heat diffusion in one, two, and three dimensions with not just help in understanding how the diffusion process looks like, it also helps debug CUDA code and lets us see if the simulation is reacting how we expect. There were many points where for example shared memory usage was not well written and I could just compile and run my code to see where the issue was. Nevertheless, it does not help actually fix the bug but can help invaluable to try and help guess where the bug might be at.

1.5.1 Future Works

- Implement more complex boundary conditions and heat source terms.
- Explore advanced color mapping techniques for rendering.
- Extend the simulation to other types of partial differential equations such as incorporating fluid mechanics with diffusion to see how heat might move in a room with regard to some kind of mixing terms.
- Look at ways the OpenGL rendering itself could be optimized so that it takes less time to render the images.
- The 3D kernel could still be further optimized and possibly we could try and care less about accuracy for more frames generated per second. Or the discretization for a given space could

be made larger, where h increases, but using a more compute bound function and less data we could get a similar level of accuracy.

- The 3D kernel itself has a possibility to be looked at further to see if there are more optimizations that can be done, as cache throughput is low.

1.6 References

- NVIDIA CUDA Samples: https://github.com/NVIDIA/cuda-samples
- CUDA Heat Equation Guide: https://enccs.github.io/OpenACC-CUDA-beginners/2.02_cuda-heat-equation/
- 3D Finite Difference Computation on GPUs using CUDA by Paulius Micikevicius: https://developer.download.nvidia.com/CUDA/CUDA_Zone/papers/gpu_3dfd_rev.pdf
- Ashwin Srinath's Implementation: https://github.com/shwina/cuda_3Dheat