Optimized GPU-Based Thermal Simulation of CPU Floorplans

Alec Roessler

August 2, 2025

Contents

1 Introduction

The purpose of this report is to develop and implement a parallel program (using GPU's) to perform temperature estimation of a CPU chip by applying a simple heat transfer algorithm to simulate heat spreading. As chips become more powerful and densely packed, the ability to manage and dissipate heat efficiently is a major consideration early in the design process. Chip designers use real world tools such as HotSpot developed by the University of Virginia, among others to model the thermal capabilities and limitations of chip designs before they are physically manufactured. However, the software these tools use can be relatively slow running on standard CPU's given the inherent parallel nature of the algorithm.

This report will first explain the process of generating a synthetic power map resembling the Alpha 21264/EV6 CPU which will serve as the input for the algorithm. Then, a basic serial implementation of the heat spreading algorithm will be developed and used for benchmark purposes. Finally, the program will be ported to CUDA and optimized by a series of improvements to further improve the performance of the parallel program over that of the serial program.

1.1 GitHub and Presentation Video

GitHub: `https://github.com/alecroessler/MPP_project_temp_simulation`
YouTube presentations link: `https://www.youtube.com/watch?v=a8sxd8ccC_Q`

2 Poisson Steady-State Heat Equation Using Jacobi Iteration

To model the heat conduction of a system, the Poisson heat equation can be used for a discrete, steady-state, time-independent, 2D system. For our purposes, we are aiming to determine the steady state temperature of the CPU not through time but with iterations that can be later extrapolated to time. Given the discrete input of the power map (to be described later), the algorithm will be put into its discretized form and iterated using Jacobi iteration. Below is the derivation of such from the general form.

To model heat conduction in a system, the Poisson heat equation is used for a discrete, steady-state, time-independent, two-dimensional domain. Our goal is to determine the steady-state temperature distribution of the CPU by iterating towards equilibrium using the Jacobi method. The derivation follows from the general form.

2.1 General Poisson Heat Equation

The Poisson heat equation for a continuous, time-independent system is given by [1]:

$$\alpha \nabla^2 T + \frac{q}{\rho c_p} = 0 \tag{1}$$

where:

- $\alpha = \frac{k}{\rho c_p}$ is the thermal diffusivity,
- q is the volumetric heat generation rate ($\mathrm{W/m^3}$),
- ρ is the material density,
- c_p is the specific heat capacity,
- k is the thermal conductivity,
- T is the temperature field.

Rearranging and substituting α, the equation becomes:

$$\frac{k}{\rho c_p}\nabla^2 T = -\frac{\dot{q}_v}{\rho c_p} \tag{2}$$

$$k\nabla^2 T = -\dot{q}_v \tag{3}$$

$$\nabla^2 T = -\frac{\dot{q}_v}{k} \tag{4}$$

Recall the definition of the Laplacian operator in two dimensions:

$$\nabla^2 T = \frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \tag{5}$$

Substituting the 2D Laplacian into the Poisson equation yields the following.

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = -\frac{q(x,y)}{k} \tag{6}$$

2.2 Discrete Jacobi Relaxation of Poisson Heat Equation

To discretize the 2D domain for numerical computation, the finite difference method is used to approximate the second derivatives using the neighboring elements [2]:

Define the neighboring temperatures around point (x, y):

$$T_{\text{top}} = T(x, y+h), \quad T_{\text{bottom}} = T(x, y-h), \quad T_{\text{right}} = T(x+h, y), \quad T_{\text{left}} = T(x-h, y)$$

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \approx \frac{T_{\text{top}} + T_{\text{bottom}} + T_{\text{right}} + T_{\text{left}} - 4T(x,y)}{h^2} \tag{7}$$

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \approx \frac{T(x,y+h) + T(x,y-h) + T(x+h,y) + T(x-h,y) - 4T(x,y)}{h^2} \tag{8}$$

Here, h denotes the uniform spacing between adjacent grid points in both the x and y directions. Including the volumetric heat generation term $q(x, y)$, the discrete Poisson heat equation becomes:

$$T(x,y+h) + T(x,y-h) + T(x+h,y) + T(x-h,y) - 4T(x,y) = -\frac{h^2}{k} q(x,y) \tag{9}$$

Solving for temperature at the point (x, y) yields the Jacobi iteration formula:

$$T^{(k+1)}(x,y) = \frac{1}{4}\left(T^{(k)}(x,y+h) + T^{(k)}(x,y-h) + T^{(k)}(x+h,y) + T^{(k)}(x-h,y) + \frac{h^2}{k} q(x,y)\right) \tag{10}$$

where:

- $T(k)$ is elements current temperature,
- $T(k+1)$ is elements next predicted temperature,
- $q(x, y)$ is the elements volumetric heat generation rate (W/m^3) from the power map,
- k is the thermal conductivity,
- h is the spacing between elements

This formula represents the relaxation of the Poisson equation using the Jacobi iterative method [3], where the superscript (k) denotes the iteration number. This allows for the approximation of the next temperature value for each element in the 2D array as a function of its current temperature value and the power density of each element allowing a programmable and implementable algorithm [4].

3 Floorplan Extraction and Power Map Creation

A power map is a 2-D array of values that represent the volumetric heat generation rate (W/m^3) of the CPU. This is the input q in the algorithm in Equation 10 necessary for modeling heat transfer. Locating a real-world power map proves difficult as this is normally proprietary information that chip manufacturers and design firms do not release publicly. For the purposes of this project, a synthetic power map will be generated using a publicly available floorplan and power trace file for the Alpha 21264/EV6 CPU.

A floorplan of a CPU is a 2D model used by design engineers to describe the chip's physical layout. It defines the positions, dimensions, and descriptions of major functional blocks on the chip. For example, functional blocks would include L1 and L2 caches, arithmetic logic units (ALU's), and registers, among others. A power trace file describes the power usage of the functional blocks in a floorplan. The floorplan and power trace of the Alpha 21264/EV6 CPU are shown below [5].

3.1 Floorplan for Alpha EV6 processor

```

# Floorplan close to the Alpha EV6 processor
# Line Format: <unit-name>\t<width>\t<height>\t<left-x>\t<bottom-y>\t[<specific-heat>]\t[<resistivity
    ↪ >]
# all dimensions are in meters
# comment lines begin with a '#'
# comments and empty lines are ignored

L2_left 0.004900 0.006200 0.000000 0.009800
L2 0.016000 0.009800 0.000000 0.000000
L2_right 0.004900 0.006200 0.011100 0.009800
Icache 0.003100 0.002600 0.004900 0.009800
Dcache 0.003100 0.002600 0.008000 0.009800
Bpred_0 0.001033 0.000700 0.004900 0.012400
Bpred_1 0.001033 0.000700 0.005933 0.012400
Bpred_2 0.001033 0.000700 0.006967 0.012400
DTB_0 0.001033 0.000700 0.008000 0.012400
DTB_1 0.001033 0.000700 0.009033 0.012400
DTB_2 0.001033 0.000700 0.010067 0.012400
FPAdd_0 0.001100 0.000900 0.004900 0.013100
FPAdd_1 0.001100 0.000900 0.006000 0.013100
FPReg_0 0.000550 0.000380 0.004900 0.014000
FPReg_1 0.000550 0.000380 0.005450 0.014000
FPReg_2 0.000550 0.000380 0.006000 0.014000
FPReg_3 0.000550 0.000380 0.006550 0.014000
FPMul_0 0.001100 0.000950 0.004900 0.014380
FPMul_1 0.001100 0.000950 0.006000 0.014380
FPMap_0 0.001100 0.000670 0.004900 0.015330
FPMap_1 0.001100 0.000670 0.006000 0.015330
IntMap 0.000900 0.001350 0.007100 0.014650
IntQ 0.001300 0.001350 0.008000 0.014650
IntReg_0 0.000900 0.000670 0.009300 0.015330
IntReg_1 0.000900 0.000670 0.010200 0.015330
IntExec 0.001800 0.002230 0.009300 0.013100
FPQ 0.000900 0.001550 0.007100 0.013100
LdStQ 0.001300 0.000950 0.008000 0.013700
ITB_0 0.000650 0.000600 0.008000 0.013100
ITB_1 0.000650 0.000600 0.008650 0.013100
```

3.2 Floorplan Visualization

The Python script provided by Hotspot (UVA) [5] was used to visualize the aforementioned floorplan.

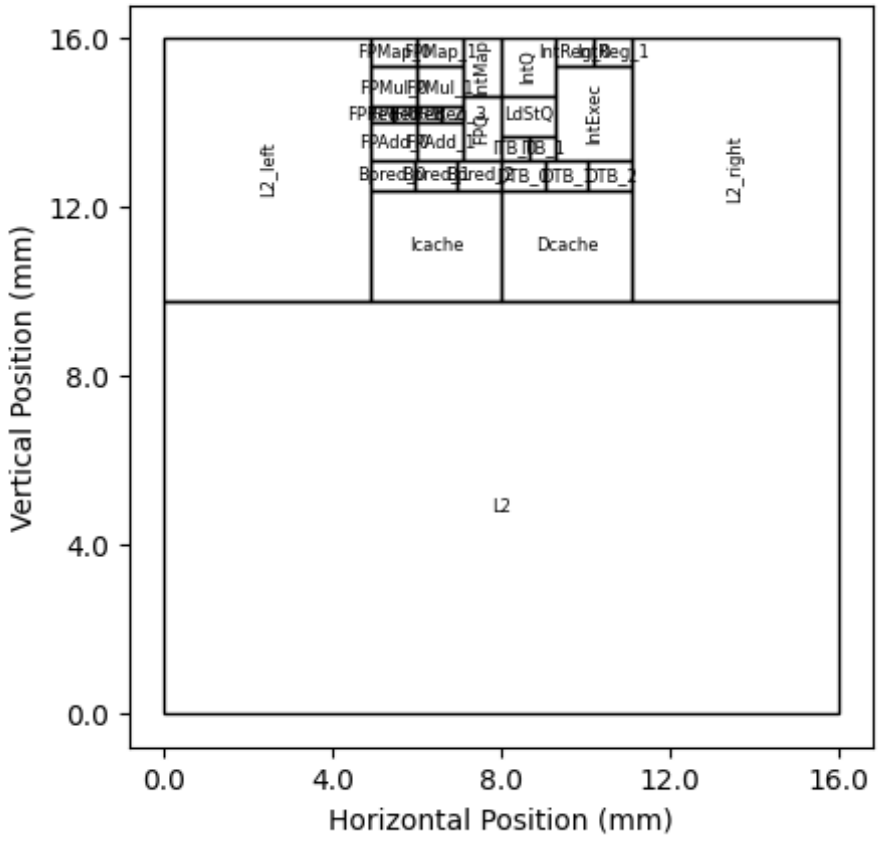

Figure 1: EV6 Processor Visualized Floorplan

3.3 Power Trace File for Alpha EV6 Processor

```
L2_left L2 L2_right Icache Dcache Bpred_0 Bpred_1 Bpred_2 DTB_0 DTB_1 DTB_2 FPAdd_0 FPAdd_1 FPReg_0
    ↪ FPReg_1 FPReg_2 FPReg_3 FPMul_0 FPMul_1 FPMap_0 FPMap_1 IntMap IntQ IntReg_0 IntReg_1 IntExec
    ↪ FPQ LdStQ ITB_0 ITB_1
1.44 7.37 1.44 8.27 14.3 1.5167 1.5167 1.5167 0.0597 0.0597 0.0597 0.62 0.62 0.1938 0.1938 0.1938
    ↪ 0.1938 0.665 0.665 0.0236 0.0236 1.07 0.365 2.585 2.585 7.7 0.0354 3.46 0.2 0.2
```

Listing 1: Power Trace File (First Two Time Steps)

3.4 Power Density Map Creation

Creating the power map involves breaking down the floorplan into finer grid elements (here we use 256×256) to generate a more realistic and non-uniform power distribution. Each block in the floorplan is related to its corresponding power consumption from the first iteration of the power trace file. This involves calculating the total elements in the 256×256 power map of each block and each element's contribution to the total power consumed in each block. For our synthetic power map, random deviations and Gaussian noise in power consumption are added, and then the total is normalized so that the sum of all the blocks still equals the total power consumption from the power trace file (approximately 58 Watts).

After this is done, the power map can be converted to a power density map by dividing the power for each element by the element's volume, calculated as:

$$\text{Element width} = \text{Element height} = \frac{\text{Die width (16 mm)}}{256} = 0.0625 \text{ mm} = 6.25 \times 10^{-5} \text{ m}$$

The element volume is then given by:

$$\text{Element volume} = \text{Element width} \times \text{Element height} \times \text{Thickness}$$

Assuming a thickness of 0.5 mm $= 5 \times 10^{-4}$ m, the volume is:

$$6.25 \times 10^{-5} \times 6.25 \times 10^{-5} \times 5 \times 10^{-4} = 1.95 \times 10^{-12} \text{ m}^3$$

After converting the power map to a power density map (serving as the input q in Equation 10 from Section 2.2), the resulting map can be both visualized and saved as a CSV file for subsequent loading into the algorithm. It is important to note that the large L2 cache region exhibits very low power density (approximately zero), despite consuming roughly 10 watts of power, because its power is distributed over a much larger area compared to other processor regions. In contrast, `Int_Reg0` and `Int_Reg1` display the highest power densities, as indicated by the yellow and white areas located near the top middle of the visualization. The Python script for this

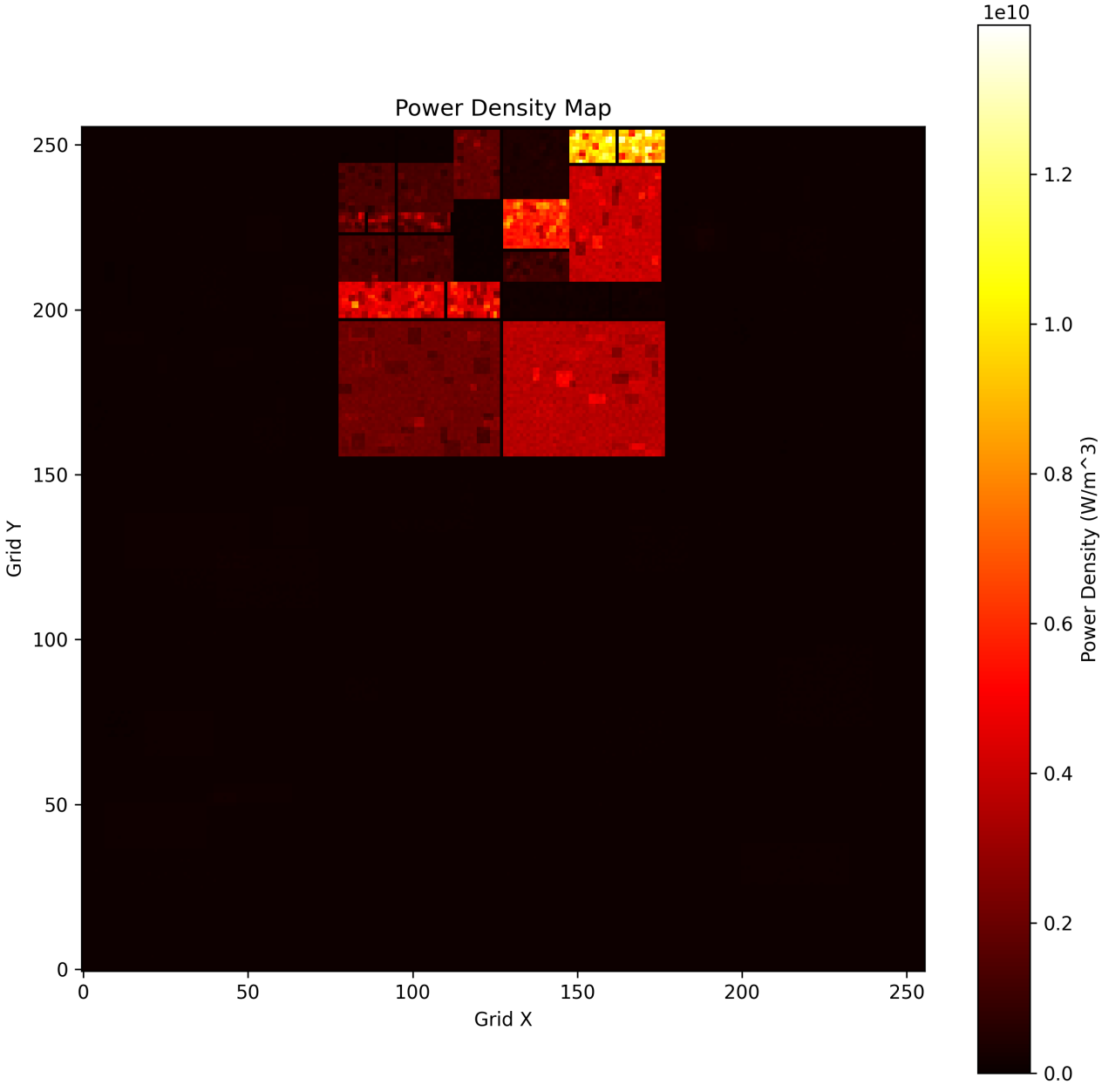

Figure 2: Power Density Map

4 Implementation (Serial Version)

4.1 Simulation Implementation

To implement the algorithm, `simulation.c` was written. In it, the following processes and functions are performed:

1. Initialize arrays of size 256×256 for `q`, `T`, and `T_new`.
   - `T` stores the previous temperature.
   - `T_new` is the updated temperature array.
   - `q` is the power density map.
2. Load the power map (`q`) from a CSV file and store it in the `q` array.
3. Initialize `T` with ambient temperature: 25 degrees Celsius.
4. Implement the Jacobi-based discrete Poisson heat equation algorithm:
   - `T_new` is updated repeatedly in a loop up to a defined maximum number of iterations (`ITERATIONS`).
   - Each element in `T_new` is computed using Equation 10.
   - Dirichlet boundary conditions are applied, fixing all boundary elements at ambient temperature (25 degrees Celsius).
   - The maximum element-wise change between `T` and `T_new` is calculated each iteration.
   - If this change falls below a defined convergence threshold (0.001), the loop terminates early.
   - Every 100 iterations, the maximum temperature change is printed.
   - `T_new` is copied into `T` for the next iteration.
5. After convergence or reaching the iteration limit, the maximum, minimum, and average temperatures of the final `T_new` array are printed.
6. The total execution time of the program is printed.

4.2 simulation.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

// Set up parameters
const int GRID_SIZE = 256;
const char* POWER_MAP_FILE = "../data/power_map_256.csv";
const double T_amb = 25; // Ambient temperature in Celcius
const int ITERATIONS = 50000;
const double DIE_WIDTH_M = 0.016; // 16 mm
const double h = DIE_WIDTH_M / GRID_SIZE;
const double k = 150.0; // thermal conductivity (using silicon)

// load power map q from CSV file
int load_power_map(const char* filename, double q[GRID_SIZE][GRID_SIZE]) {
    // Confirm file opens
    FILE* file = fopen(filename, "r");
    if (!file) {
        printf("Error opening file:\n");
        return 1;
    }

```

```
    // Read file data
    for (int i = 0; i < GRID_SIZE; i++) {
        for (int j = 0; j < GRID_SIZE; j++) {
            fscanf(file, "%lf,", &q[i][j]);
        }
    }

    fclose(file);
    return 0;
}

// Compute the maximum absolute difference between two temperature grids (element by element)
double max_abs_diff(double a[GRID_SIZE][GRID_SIZE], double b[GRID_SIZE][GRID_SIZE]) {
    double max_diff = 0.0;
    for (int i = 0; i < GRID_SIZE; i++) {
        for (int j = 0; j < GRID_SIZE; j++) {
            double diff = fabs(a[i][j] - b[i][j]);
            if (diff > max_diff) {
                max_diff = diff;
            }
        }
    }
    return max_diff;
}

// Compute the maximum, minimum, and average temperatures in a grid
double max_temp(double arr[GRID_SIZE][GRID_SIZE]) {
    double max_val = arr[0][0];
    for (int i = 0; i < GRID_SIZE; i++) {
        for (int j = 0; j < GRID_SIZE; j++) {
            if (arr[i][j] > max_val) max_val = arr[i][j];
        }
    }
    return max_val;
}
double min_temp(double arr[GRID_SIZE][GRID_SIZE]) {
    double min_val = arr[0][0];
    for (int i = 0; i < GRID_SIZE; i++) {
        for (int j = 0; j < GRID_SIZE; j++) {
            if (arr[i][j] < min_val) min_val = arr[i][j];
        }
    }
    return min_val;
}
double avg_temp(double arr[GRID_SIZE][GRID_SIZE]) {
    double sum = 0.0;
    for (int i = 0; i < GRID_SIZE; i++) {
        for (int j = 0; j < GRID_SIZE; j++) {
            sum += arr[i][j];
        }
    }
    return (sum / (GRID_SIZE * GRID_SIZE));
}









int main() {
    clock_t start_time = clock();

    // Initialize arrays for power map and temperatures
    double q[GRID_SIZE][GRID_SIZE];
    double T[GRID_SIZE][GRID_SIZE];
    double T_new[GRID_SIZE][GRID_SIZE];

```

```
    // Load the power map from the CSV file
    if (load_power_map(POWER_MAP_FILE, q) != 0) {
        printf("Failed to load power map.\n");
        return 1;
    }

    // Initialize T with T_amb
    for (int i = 0; i < GRID_SIZE; i++) {
        for (int j = 0; j < GRID_SIZE; j++) {
            T[i][j] = T_amb;
            T_new[i][j] = T_amb;
        }
    }

    clock_t setup_time = clock();
    double setup_elapsed = (double)(setup_time - start_time) / CLOCKS_PER_SEC;
    printf("Setup and initialize time: %.2f seconds\n", setup_elapsed);


    // Jacobi discrete heat equation (propogation simulation loop)
    for (int iter = 0; iter < ITERATIONS; iter++) {
        // Update internal grid points
        for (int i = 1; i < GRID_SIZE - 1; i++) {
            for (int j = 1; j < GRID_SIZE - 1; j++) {
                T_new[i][j] = (T[i+1][j] + T[i-1][j] + T[i][j+1] + T[i][j-1] + (h*h / k) * q[i][j]) /
                    ↪ 4.0;
            }
        }

        // Apply Dirichlet boundary conditions (fixed ambient temperature of 25 degrees Celsius)
        for (int i = 0; i < GRID_SIZE; i++) {
            T_new[i][0] = T_amb;
            T_new[i][GRID_SIZE - 1] = T_amb;
            T_new[0][i] = T_amb;
            T_new[GRID_SIZE - 1][i] = T_amb;
        }

        // Check for convergence and exit
        double max_change = max_abs_diff(T, T_new);
        if (max_change < 1e-3) {
            printf("Converged after %d iterations\n", iter);
            break;
        }

        double max_temperature = max_temp(T_new);
        if (iter % 1000 == 0) {
            printf("Iteration %d: max change = %.5f, max temp = %.5f\n", iter, max_change,
                ↪ max_temperature);
        }


        // Copy T_new to T for next iteration
        for (int i = 0; i < GRID_SIZE; i++) {
            for (int j = 0; j < GRID_SIZE; j++) {
                T[i][j] = T_new[i][j];
            }
        }
    }

    clock_t simulation_time = clock();
    double simulation_elapsed = (double)(simulation_time - setup_time) / CLOCKS_PER_SEC;
    printf("Simulation time: %.2f seconds\n", simulation_elapsed);

    printf("Max Temp: %.2f C\n", max_temp(T_new));
    printf("Min Temp: %.2f C\n", min_temp(T_new));
    printf("Avg Temp: %.2f C\n", avg_temp(T_new));
```

```
154
155     clock_t end_time = clock();
156
157     double elapsed_time = (double)(end_time - start_time) / CLOCKS_PER_SEC;
158     printf("Total Execution time: %.2f seconds\n", elapsed_time);
159
160
161     // Save results to csv file
162     FILE* file = fopen("/home/roesslera/code/MPP_project_temp_simulation/data/results.csv", "w");
163     if (!file) {
164         printf("Error opening results file for writing.\n");
165         return 1;
166     }
167
168     for (int i = 0; i < GRID_SIZE; i++) {
169         for (int j = 0; j < GRID_SIZE; j++) {
170             fprintf(file, "%.6f", T[i][j]);
171             if (j < GRID_SIZE - 1) {
172                 fprintf(file, ",");
173             }
174         }
175         fprintf(file, "\n");
176     }
177
178     fclose(file);
179
180     return 0;
181 }
```

4.3 Kodiak Serial Setup

To run the serial implementation (C file) on Kodiak, we cannot do the same as we did for our previous labs nor should we run directly on the login node. To run on a separate CPU node, the following PBS file was created.

```
#!/bin/bash
#PBS -q batch
#PBS -l nodes=1:ppn=1
#PBS -l walltime=00:30:00
#PBS -N test_job
#PBS -j oe
#PBS -o output.log

cd $PBS_O_WORKDIR

echo "Running on node: $(hostname)"

./simulation
```

To run, build the executable with: `gcc -o simulation simulation.c -lm` where `-lm` links the math module. The executable can be run with the following command: `qsub run_cpu.pbs`

4.4 Results

The results from the Kodiak serial implementation output file are as follows.

```
Running on node: n007
Setup and initialize time: 0.02 seconds
Iteration 0: max change = 0.09110, max temp = 25.09110
Iteration 1000: max change = 0.01534, max temp = 44.52675
Iteration 2000: max change = 0.01226, max temp = 57.09434
Iteration 3000: max change = 0.00975, max temp = 67.55969
```

```
Iteration 4000: max change = 0.00793, max temp = 76.19184
Iteration 5000: max change = 0.00661, max temp = 83.37226
Iteration 6000: max change = 0.00562, max temp = 89.42951
Iteration 7000: max change = 0.00486, max temp = 94.60199
Iteration 8000: max change = 0.00426, max temp = 99.07686
Iteration 9000: max change = 0.00376, max temp = 102.98722
Iteration 10000: max change = 0.00336, max temp = 106.42706
Iteration 11000: max change = 0.00302, max temp = 109.47416
Iteration 12000: max change = 0.00272, max temp = 112.19540
Iteration 13000: max change = 0.00247, max temp = 114.62837
Iteration 14000: max change = 0.00225, max temp = 116.81220
Iteration 15000: max change = 0.00205, max temp = 118.78867
Iteration 16000: max change = 0.00188, max temp = 120.57765
Iteration 17000: max change = 0.00173, max temp = 122.19867
Iteration 18000: max change = 0.00159, max temp = 123.67093
Iteration 19000: max change = 0.00146, max temp = 125.01086
Iteration 20000: max change = 0.00135, max temp = 126.23569
Iteration 21000: max change = 0.00124, max temp = 127.36137
Iteration 22000: max change = 0.00115, max temp = 128.39181
Iteration 23000: max change = 0.00106, max temp = 129.33582
Converged after 23754 iterations
Simulation time: 30.85 seconds
Max Temp: 130.00 C
Min Temp: 25.00 C
Avg Temp: 52.32 C
Total Execution time: 30.87 seconds
```

The resulting data was then graphed to visualize the max temperature and convergence rate vs iteration using `data_analysis.py` which is located in the appendix.

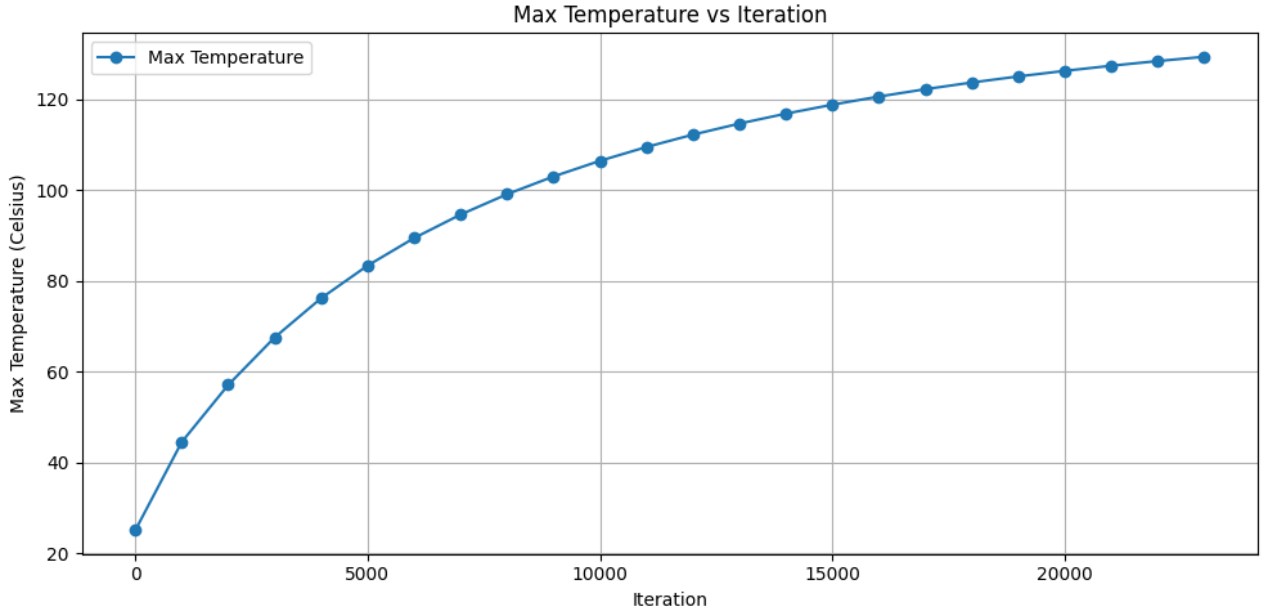

Figure 3: Max Temperature vs Iteration

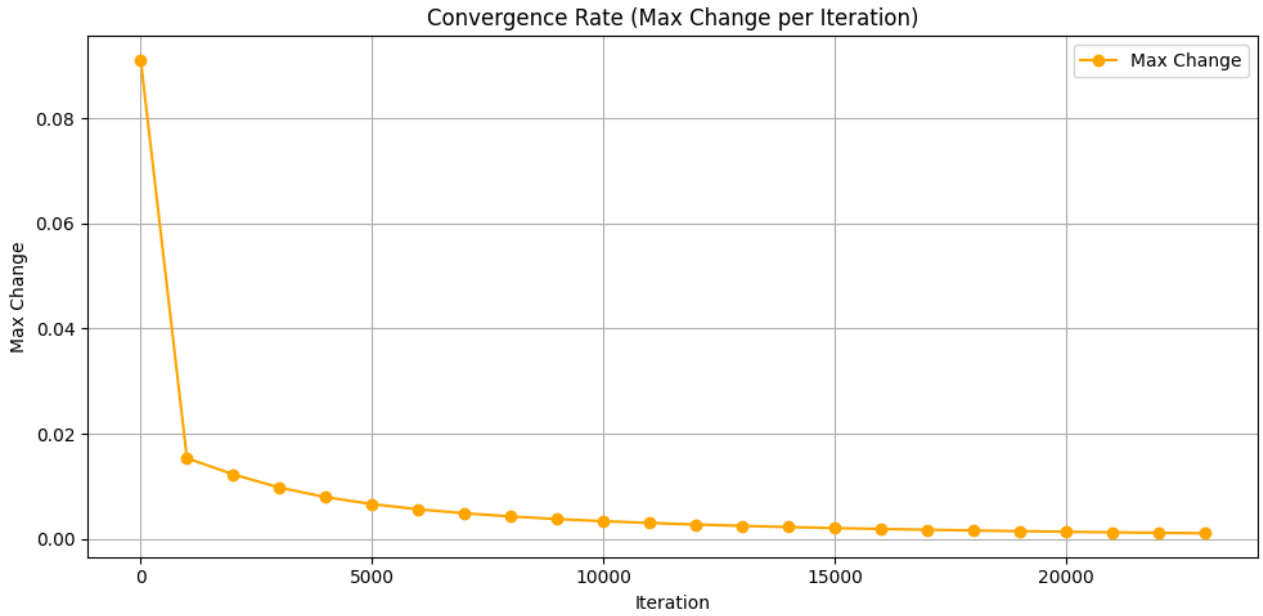

Figure 4: Convergence Rate

The Python script `visualize_temp.py` (located in the appendix) was used to generate a heatmap of the resulting temperature grid.

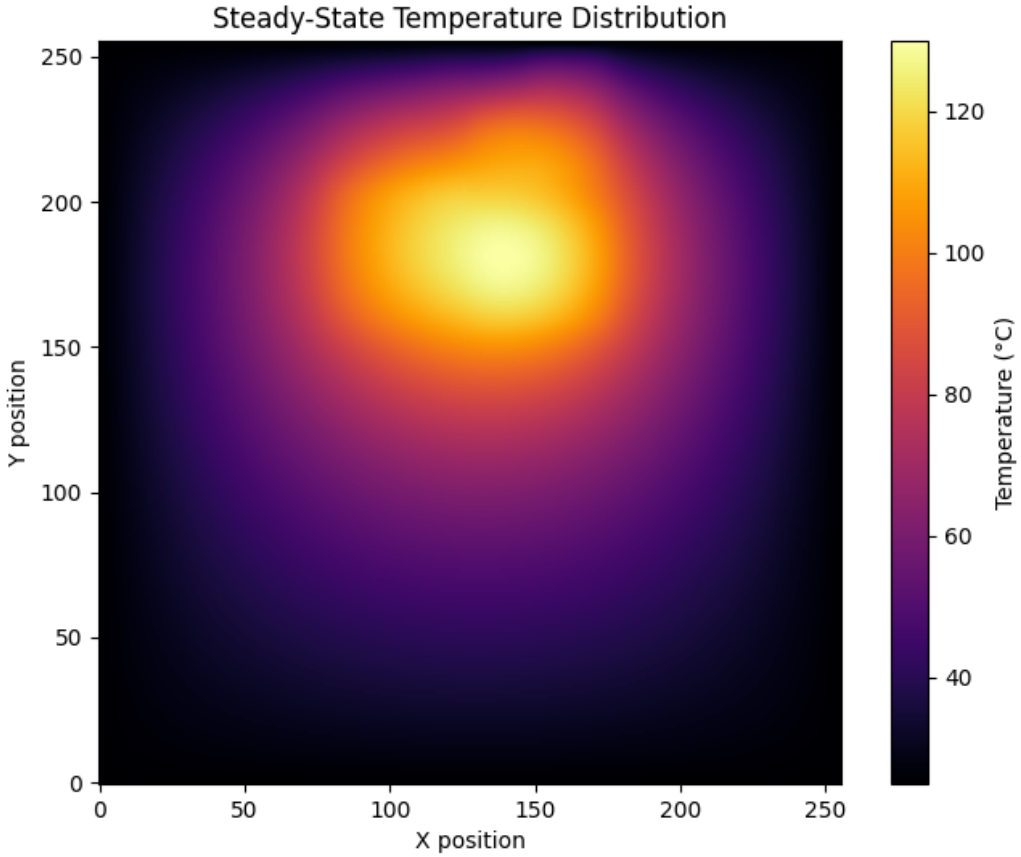

Figure 5: Steady-State Temperature Distribution

The results indicate that the simulation reached convergence after 23,754 iterations with the final

maximum temperature seen being 130 degrees Celsius and an average temperature of 52.32 degrees Celsius. From examining Figure 3, the max temperature rises faster initially and then levels out resulting in steady state. This follows the data in Figure 4 indicating a diminishing maximum change element wise in temperatures after each iteration. Initially, there is a steep decline in change until the value diminishes to below the 0.001 threshold.

Figure 5 shows the heat map displaying the final temperature distribution of the chip. Higher power density areas in the upper middle see the highest temperature while elements along the border remain at ambient temperature. Realistically this would not happen as the edges (Dirichlet's boundary conditions) are not held at ambient temperature and every element in the chip will heat up. Additionally this is only considering the lateral propagation of temperatures through the silicon material from element to element. It is not including the temperature spread and dissipation introduced through a heat sink. This results in an unrealistic heat distribution but does give insight as to how it may propagate and reach steady state. For a more realistic model, the boundary conditions can be become variable, a heat sink can be accounted for, and the equation used can be adjusted to be a function of time instead of simple iteration constants. Additionally, a real power map would be used using lab measured data of the chip. However, these additions are out of the scop of this report as it is focused on GPU implementation and optimization of a parallel computation program.

5 Parallel (CUDA) Implementation

To port the serial code to CUDA (for parallel execution on a GPU), several changes were made. The implementation was split into two files: `main.cu` and a supporting kernel file `kernel.cu`.

5.1 Naive Version

The resulting naive (basic working) version of the CUDA implementation is described below.

1. Assign constant parameters (grid size, thermal variables).
2. Initialize and allocate host arrays `q_h`, `T_h`, and `T_new_h` of size 256×256. These arrays are flattened into one-dimensional form, as CUDA kernels typically operate on linear memory. This flattening allows for easier indexing and efficient memory access within device code.
3. Allocate corresponding device arrays: `q_d`, `T_d`, and `T_new_d`.
4. Copy data from host arrays to device arrays using `cudaMemcpy`.
5. Define grid and block dimensions using the explicit ceiling division method.
6. Launch the CUDA kernel within an iteration loop:
    - After each kernel call, copy `T_new_d` back to host memory.
    - Compute the maximum change between the old and new temperature arrays on the host.
    - If the maximum change is below the convergence threshold (0.001 degrees C), the simulation returns.
    - Swaps the device pointers `T_d` and `T_new_d` to prepare for the next iteration.
7. Once the simulation completes, the final device temperature is copied to host memory (T_new_h)
8. Computes and prints the maximum, minimum, and average final temperatures.
9. Calls verification function to check the correctness of the output.
10. Frees all allocated host and device memory.
11. Prints the total execution time of the program.

5.1.1 main.cu

```
#include <stdio.h>
#include <stdint.h>
#include "support.h"
#include "kernel.cu"

// Set up parameters
const int GRID_SIZE = 256;
const int total_size = GRID_SIZE * GRID_SIZE;
const char* POWER_MAP_FILE = "/home/roesslera/code/MPP_project_temp_simulation/data/power_map_256.csv"
    ↪ ;
const double T_amb = 25; // Ambient temperature in Celcius
const int ITERATIONS = 50000;
const double DIE_WIDTH_M = 0.016;
const double h = DIE_WIDTH_M / GRID_SIZE;
const double k = 150.0; // thermal conductivity (using silicon)

int load_power_map(const char* filename, double* q) {
    // Confirm file opens
    FILE* file = fopen(filename, "r");
    if (!file) {
        fprintf(stderr, "Error opening file:\n");
        return 1;
    }

    // Read file data
    for (int i = 0; i < total_size; i++) {
        fscanf(file, "%lf,", &q[i]);
    }

    fclose(file);
    return 0;
}


int main(int argc, char* argv[])
{
    Timer timer, total_timer, timer_copy, timer_max, timer_kernel;
    float t_copy = 0, t_max = 0, t_kernel = 0;
    startTime(&total_timer);
    cudaError_t cuda_ret;

    // Initialize host variables ----------------------------------------------

    printf("\nSetting up the problem..."); fflush(stdout);
    startTime(&timer);

    double *q_h, *T_h, *T_new_h;

    q_h = (double*) malloc( sizeof(double) * total_size );
    T_h = (double*) malloc( sizeof(double) * total_size );
    T_new_h = (double*) malloc( sizeof(double) * total_size );

    for (unsigned int i=0; i < total_size; i++) { T_new_h[i] = T_amb; T_h[i] = T_amb; }
    if (load_power_map(POWER_MAP_FILE, q_h) != 0) {
        fprintf(stderr, "Failed to load power map.\n");
        return 1;
    }

    stopTime(&timer); printf("%f s\n", elapsedTime(timer));

    // Allocate device variables ----------------------------------------------

    printf("Allocating device variables..."); fflush(stdout);
    startTime(&timer);
```

```
64
65      double *q_d, *T_d, *T_new_d;
66
67      // CUDA device variables for q, T, and T_new
68      cuda_ret = cudaMalloc((void**)&q_d, sizeof(double)* total_size);
69      if(cuda_ret != cudaSuccess) FATAL("Unable to allocate device memory for q_d");
70
71      cuda_ret = cudaMalloc((void**)&T_d, sizeof(double)* total_size);
72      if(cuda_ret != cudaSuccess) FATAL("Unable to allocate device memory for T_d");
73
74      cuda_ret = cudaMalloc((void**)&T_new_d, sizeof(double)* total_size);
75      if(cuda_ret != cudaSuccess) FATAL("Unable to allocate device memory for T_new_d");
76
77
78
79      cudaDeviceSynchronize();
80      stopTime(&timer); printf("%f s\n", elapsedTime(timer));
81
82      // Copy host variables to device ------------------------------------------
83
84      printf("Copying data from host to device..."); fflush(stdout);
85      startTime(&timer);
86
87      // Copy q, T, and T_new from host to device
88      cuda_ret = cudaMemcpy(q_d, q_h, sizeof(double)*total_size, cudaMemcpyHostToDevice);
89      if(cuda_ret != cudaSuccess) FATAL("Unable to copy q from host to device");
90      cuda_ret = cudaMemcpy(T_d, T_h, sizeof(double)*total_size, cudaMemcpyHostToDevice);
91      if(cuda_ret != cudaSuccess) FATAL("Unable to copy T from host to device");
92      cuda_ret = cudaMemcpy(T_new_d, T_new_h, sizeof(double)*total_size, cudaMemcpyHostToDevice);
93      if(cuda_ret != cudaSuccess) FATAL("Unable to copy T_new from host to device");
94
95      cudaDeviceSynchronize();
96      stopTime(&timer); printf("%f s\n", elapsedTime(timer));
97
98      // Launch kernel ----------------------------
99      startTime(&timer);
100
101     // Define grid and block dimensions
102     dim3 blockDim(16, 16);
103     dim3 gridDim((GRID_SIZE + blockDim.x - 1) / blockDim.x, (GRID_SIZE + blockDim.y - 1) / blockDim.y)
        ↪ ;
104
105     // Launch the kernel
106     int iter;
107     for (iter = 0; iter < ITERATIONS; iter++) {
108         startTime(&timer_kernel);
109         compute_temperature<<<gridDim, blockDim>>>(T_d, T_new_d, q_d, k, GRID_SIZE, h, T_amb);
110         stopTime(&timer_kernel); t_kernel += elapsedTime(timer_kernel);
111         cuda_ret = cudaGetLastError();
112         if(cuda_ret != cudaSuccess) FATAL("Unable to launch kernel");
113
114
115         // Copy T and T_new to host to check convergence
116         startTime(&timer_copy);
117         cudaMemcpy(T_h, T_d, sizeof(double) * total_size, cudaMemcpyDeviceToHost);
118         cudaMemcpy(T_new_h, T_new_d, sizeof(double) * total_size, cudaMemcpyDeviceToHost);
119         stopTime(&timer_copy); t_copy += elapsedTime(timer_copy);
120
121         startTime(&timer_max);
122         double max_change = max_abs_diff(T_h, T_new_h, total_size);
123         stopTime(&timer_max); t_max += elapsedTime(timer_max);
124
125
126         if (max_change < 1e-3) {
127             printf("Converged after %d iterations\n", iter);
128             break;
```

```
        }

        // Swap T and T_new pointers
        double* temp = T_d;
        T_d = T_new_d;
        T_new_d = temp;
    }

    cuda_ret = cudaDeviceSynchronize();

    // Copy device variables from host ----------------------------------------

    printf("Copying data from device to host..."); fflush(stdout);
    startTime(&timer);

    cuda_ret = cudaMemcpy(T_new_h, T_new_d, sizeof(double)*total_size, cudaMemcpyDeviceToHost);
    if(cuda_ret != cudaSuccess) FATAL("Unable to copy memory from device");

    cudaDeviceSynchronize();
    stopTime(&timer); printf("%f s\n", elapsedTime(timer));

    double max_temp_T = max_temp(T_new_h, GRID_SIZE);
    double min_temp_T = min_temp(T_new_h, GRID_SIZE);
    double avg_temp_T = avg_temp(T_new_h, GRID_SIZE);

    printf("Max Temp: %.2f C\n", max_temp_T);
    printf("Min Temp: %.2f C\n", min_temp_T);
    printf("Avg Temp: %.2f C\n", avg_temp_T);

    // Verify correctness -----------------------------------------------------

    printf("Verifying results..."); fflush(stdout);
    verify(iter, max_temp_T, min_temp_T, avg_temp_T);

    // Free memory ------------------------------------------------------------

    free(q_h);
    free(T_h);
    free(T_new_h);

    // Free device variables
    cudaFree(q_d);
    cudaFree(T_d);
    cudaFree(T_new_d);

    printf("Temperature array copy time to host for convergence check: %.5f s\n", t_copy);
    printf("Time for finding maximum difference for convergence check: %.5f s\n", t_max);
    printf("Kernel execution time: %.5f s\n", t_kernel);

    stopTime(&total_timer); printf("Total Execution Time: %f s\n", elapsedTime(total_timer));

    return 0;
}
```

5.1.2 kernel.cu

Additionally, a `kernel.cu` file was written to contain the CUDA kernel and supporting functions.

`compute_temperature` kernel:

- Passes in `T`, `T_new`, `q`, and equation parameter variables.
- Calculates the correct `x` and `y` coordinates based on block index, block dimension, and thread index.

- Verifies `x` and `y` coordinates are within valid bounds.
- Applies Dirichlet boundary conditions (sets boundaries to ambient temperature).
- Calculates neighboring elements accounting for the 1D array indexing.
- Applies Equation 10 and updates the corresponding `T_new` array element.

`max_abs_diff`: Computes the maximum absolute difference between the current and previous temperature arrays, element by element.

`max_temp`, `min_temp`, `avg_temp`: Compute the maximum, minimum, and average temperatures of the array, respectively.

```
// Kernel algorithm
__global__ void compute_temperature(double* T, double* T_new, double* q, double k,
    int grid_size, double h, double T_amb) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    if (x >= grid_size || y >= grid_size) return;

    int idx = y * grid_size + x;

    // Apply Dirichlet boundary conditions
    if (x == 0 || x == grid_size - 1 || y == 0 || y == grid_size - 1) {
        T_new[idx] = T_amb;
        return;
    }

    // Compute 1D indices for neighbors
    int top = (y - 1) * grid_size + x;
    int bottom = (y + 1) * grid_size + x;
    int left = y * grid_size + (x - 1);
    int right = y * grid_size + (x + 1);

    double coeff = (h * h / k) * q[idx];

    T_new[idx] = (T[top] + T[bottom] + T[left] + T[right] + coeff) / 4.0;
}

// Compute the maximum absolute difference between two arrays
double max_abs_diff(double* a, double* b, int size) {
    double max_diff = 0.0;
    for (int i = 0; i < size; ++i) {
        double diff = fabs(a[i] - b[i]);
        if (diff > max_diff) max_diff = diff;
    }
    return max_diff;
}


// Compute the maximum, minimum, and average temperature in the grid
double max_temp(double* arr, int grid_size) {
    double max_val = arr[0];
    for (int i = 0; i < grid_size * grid_size; i++) {
        if (arr[i] > max_val) max_val = arr[i];
    }
    return max_val;
}
double min_temp(double* arr, int grid_size) {
    double min_val = arr[0];
    for (int i = 0; i < grid_size * grid_size; i++) {
        if (arr[i] < min_val) min_val = arr[i];
    }
    return min_val;
```

```
}
double avg_temp(double* arr, int grid_size) {
    double sum = 0.0;
    for (int i = 0; i < grid_size * grid_size; i++) {
        sum += arr[i];
    }
    return (sum / (grid_size * grid_size));
}
```

5.1.3 Results

```
Setting up the problem...0.008331 s
Allocating device variables...0.156288 s
Copying data from host to device...0.000162 s
Converged after 23754 iterations
Copying data from device to host...0.000049 s
Max Temp: 130.00 C
Min Temp: 25.00 C
Avg Temp: 52.32 C
Verifying results...TEST PASSED

Temperature array copy time to host for convergence check: 2.46523 s
Time for finding maximum difference for convergence check: 0.93031 s
Kernel execution time: 0.07622 s
Total Execution Time: 3.641117 s
```

The parallel CUDA version of this program executes in 3.64 seconds, compared to the 30.87 seconds it took for the serial version, resulting in a speedup of approximately 8.5x. However, it can be seen that the bulk of the execution time was spent on the convergence check—copying the resulting temperature arrays to host memory and computing the maximum temperature increase. Below is a table representing each portion of the algorithm's corresponding execution time expressed as a percentage.

Task	**Time (s)**	**% of Total Time**
Setting up the problem	0.00833	0.23%
Allocating device variables	0.15629	4.29%
Copying data from host to device	0.00016	0.0044%
Copying data from device to host	0.00005	0.0013%
Temp array copy time to host for conv. check	2.46523	67.68%
Time for finding max difference for conv. check	0.93031	25.54%
Kernel execution time	0.07622	2.09%

Table 1: Execution time breakdown as percentage of total execution time (3.64 s)

The actual computation kernel for the temperature updates accounts for only 2.09% of the total time while the convergence check takes 93.22% of the time. Given a very small kernel execution time of 0.07622 seconds for a large number of iterations (23,754), it is then worthwhile to focus on improving the convergence check portion.

5.2 Device Side Convergence Check

The reason the convergence check is taking a large portion of the programs execution time is due to the memory copies and serial maximum change math on the host. This is because for each of the 23,754 iterations it must copy both temperature arrays to the host and compute the maximum difference for element-wise in the 256 x 256 array while recording the current maximum. For two arrays of this size

comprised of doubles, this means for every iteration (2 * 256 * 256 * 8) = 1 MB of memory transfer. This is very inefficient and makes for a great use case of a device side reduction algorithm.

The next sections contains the necessary changes for implementing a partial reduction algorithm on the device. Each of the 256 blocks is responsible for calculating its own maximum change value which is then copied to the host for a much smaller serial global maximum check. This reduces the amount of memory transfers significantly and will lead to a much faster executing program. By reducing the copied array to 256 elements, this means that only 256 * 8 = 2 kB of data will have to be transferred per iteration.

5.2.1 Changes Made

For main.cu:

At beginning:

```
// For reduction kernel: hardcoded since we know the values and can keep structure of program in tact
const int THREADS_PER_BLOCK = 256; // blockDim.x * blockDim.y
const int BLOCKS = 256; // (GRID_SIZE * GRID_SIZE + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK;
```

In initialize host variables:

```
double *max_diff_h;
max_diff_h = (double*) malloc(sizeof(double) * BLOCKS);
```

In timer setup (beginning of main):

```
Timer timer_max_device, timer_max_host
float t_max_host = 0, t_max_device = 0;
```

In allocate device variables:

```
double *max_diff_d;
cudaMalloc(&max_diff_d, BLOCKS * sizeof(double));
```

In free memory section:

```
free(max_diff_h);
cudaFree(max_diff_d);
```

In kernel launch loop for convergence check:

```
// Launch reduction kernel to compute maximum difference
startTime(&timer_max_device);
max_diff_reduction<<<BLOCKS, blockDim>>>(T_d, T_new_d, max_diff_d, total_size);
cudaDeviceSynchronize();
stopTime(&timer_max_device); t_max_device += elapsedTime(timer_max_device);

// Copy max_diff_d from device to host
startTime(&timer_copy);
cudaMemcpy(max_diff_h, max_diff_d, sizeof(double) * BLOCKS, cudaMemcpyDeviceToHost);
stopTime(&timer_copy); t_copy += elapsedTime(timer_copy);

// Complete reduction on host
startTime(&timer_max_host);
double max_change = 0.0;
for (int i = 0; i < BLOCKS; i++) {
    if (max_diff_h[i] > max_change) {
        max_change = max_diff_h[i];
    }
}
stopTime(&timer_max_host); t_max_host += elapsedTime(timer_max_host);
```

in kernel.cu added new CUDA kernel

```
__global__ void max_diff_reduction(double* T, double* T_new, double* max_diff, int total_size) {
    __shared__ double data[256];
    int local_index = threadIdx.y * blockDim.x + threadIdx.x;
```

```
    int global_index = blockIdx.x * blockDim.x * blockDim.y + local_index;


    // Compute difference for each thread
    double difference = 0.0;
    if (global_index < total_size) {
        difference = fabs(T_new[global_index] - T[global_index]);
    }

    data[local_index] = difference;
    __syncthreads();

    // Max reduction
    for (int stride = 128; stride > 0; stride /= 2) {
        if (local_index < stride) {
            data[local_index] = fmax(data[local_index], data[local_index + stride]);
        }
        __syncthreads();
    }

    // Return the maximum difference at index 0
    if (local_index == 0) {
        max_diff[blockIdx.x] = data[0];
    }
}
```

5.2.2 Results

```
Setting up the problem...0.008513 s
Allocating device variables...0.180910 s
Copying data from host to device...0.000163 s
Converged after 23754 iterations
Copying data from device to host...0.000053 s
Max Temp: 130.00 C
Min Temp: 25.00 C
Avg Temp: 52.32 C
Verifying results...TEST PASSED

Reduce Temperature kernel time for convergence check: 0.21038 s
Copy reduced Temperature to host for convergence check: 0.14220 s
Reduce Temperature further on host for convergence check: 0.01013 s
Kernel algorithm for temperature computation execution time: 0.05454 s
Total Execution Time: 0.610913 s
```

The total execution time decreased from 3.64 to 0.61 seconds from the naive version resulting resulting in a ≈ 5.97 speedup.

5.3 Further Optimizations

While the optimized version significantly improved the execution time, further changes can be made to further improve the performance. One small optimization that can be done is moving the coefficient calculation to the host and passing it in every time. This will reduce the arithmetic operations per thread leading to 2 fewer than the previous implementation. Aside from this, the computation kernel itself was determined to be optimal.

In the optimized version, the reduction kernel still takes a majority of the time along with the subsequent host side copy. This kernel uses much of the same variables as the temperature kernel meaning it could they could be merged together leading to less kernel launch time.

Task	Time (s)	% of Total Time
Setting up the problem	0.00851	1.39%
Allocating device variables	0.18091	29.61%
Copying data from host to device	0.00016	0.03%
Copying data from device to host	0.00005	0.01%
Reduce Temperature kernel time for convergence check	0.21038	34.44%
Copy reduced Temperature to host for convergence check	0.14220	23.28%
Reduce Temperature further on host for convergence check	0.01013	1.66%
Kernel algorithm for temperature computation execution time	0.05454	8.93%

Table 2: Execution time breakdown as percentage of total execution time (0.61 s).

5.3.1 Changes Made

To implement these optimizations, the following changes were implemented.

In main.cu: At the beginning of the program-

```
double coeff = (h * h / k);
```

Altered kernel launch-

```
compute_temperature<<<gridDim, blockDim>>>(T_d, T_new_d, q_d, coeff, GRID_SIZE, T_amb, max_diff_d);
```

In kernel.cu: The new merged kernel becomes-

```
__global__ void compute_temperature(double* T, double* T_new, double* q, double coeff,
    int grid_size, double T_amb, double* max_diff_per_block) {

    // Indicies
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    int idx = y * grid_size + x;
    int local_idx = threadIdx.y * blockDim.x + threadIdx.x;

    // Shared memory for block reduction
    __shared__ double s_data[256];

    double difference = 0.0;

    // Bundle temperature computations, boundary checks, and difference calculation
    if (x < grid_size && y < grid_size) {
        if (x == 0 || x == grid_size - 1 || y == 0 || y == grid_size - 1) {
            // Boundary condition
            T_new[idx] = T_amb;
        } else {

            // Extract neighbors
            int top = (y - 1) * grid_size + x;
            int bottom = (y + 1) * grid_size + x;
            int left = y * grid_size + (x - 1);
            int right = y * grid_size + (x + 1);

            // Perform temperature calculation
            coeff *= q[idx];
            T_new[idx] = (T[top] + T[bottom] + T[left] + T[right] + coeff) / 4.0;
        }
        difference = fabs(T_new[idx] - T[idx]); // Calculate the difference compared to previous
    }

    s_data[local_idx] = difference;
    __syncthreads();

    // Parallel reduction to find max diff per block
    for (int stride = blockDim.x * blockDim.y / 2; stride > 0; stride /= 2) {
```

```
        if (local_idx < stride) {
            s_data[local_idx] = fmax(s_data[local_idx], s_data[local_idx + stride]);
        }
        __syncthreads();
    }

    // Write max diff of this block to global memory
    if (local_idx == 0) {
        int block_id = blockIdx.y * gridDim.x + blockIdx.x;
        max_diff_per_block[block_id] = s_data[0];
    }
}
```

5.3.2 Results

```
Setting up the problem...0.008379 s
Allocating device variables...0.171717 s
Copying data from host to device...0.000160 s
Converged after 23754 iterations
Copying data from device to host...0.000052 s
Max Temp: 130.00 C
Min Temp: 25.00 C
Avg Temp: 52.32 C
Verifying results...TEST PASSED

Copy reduced Temperature to host for convergence check: 0.13767 s
Reduce Temperature further on host for convergence check: 0.01091 s
Kernel execution time: 0.19992 s
Total Execution Time: 0.531946 s
```

The new total execution time is now 0.53 seconds resulting in a speedup of $\approx 13\%$.

6 Analysis of GPU Optimizations

6.1 Arithmetic Intensity

The arithmetic intensity of the kernel can be defined as: Arithmetic Intensity $= \frac{\text{FLOPs}}{\text{Bytes accessed}}$

For our combined kernel loads are:

- 6 Doubles loaded (5 from `T` array and 1 from `q` array)
- 1 write to `T_new` array

Arithmetic operations:

- 5 adds
- 1 multiplication
- 1 division
- 1 subtraction
- 1 absolute value

Resulting in 7 doubles in memory transfer and 9 arithmetic operations meaning the arithmetic intensity becomes:

$$\text{arithmetic intensity} = \frac{9}{7 \times 8 \text{ bytes per double}} \approx 0.16$$

This results in the kernel being considered memory bound. To increase the arithmetic intensity, three optimizations were attempted. First, the temperature arrays were defined as floats instead of doubles (the q array has to be a double to hold its data). However, the float precision proved to not be sufficient resulting in a incorrect result and all array were kept as doubles.

Next, shared memory tiling and updating multiple elements per thread was attempted. However, neither approach showed any improvement in total execution time likely due to the additional boundary checks and halo cell logic leading to divergence and additional time for logic.

6.2 Time Complexity

The optimizations that did acheive an improvement in performance were in relation to the convergence check logic. This included implementing a reduction kernel along with merging the kernels together (included in this is the coefficient host simplification). The resulting time improvements compared to the original serial version can be seen below.

Method	**Time (s)**	**Speedup**
Serial	30.870000	NA
Naive	3.641117	NA
Reduction	0.610913	496%
Merged Kernel	0.531946	584%

Table 3: Execution times and speedup percentages relative to Naive

This means that the optimizations employed successfully improved the execution time relative to the Naive version with the final optimized version exhibiting almost a 7x speedup. Additionally, the power of parallel computing for applicative programs can be seen when comparing each CUDA implementation to the initial serial version.

7 References

References

[1] "Poisson's Equation - Steady-State Heat Transfer," *Nuclear Power*. Available: `https://www.nuclear-power.com/nuclear-engineering/heat-transfer/thermal-conduction/heat-conduction-equation/poissons-equation-steady-state-heat-transfer/`. [Accessed: Aug. 7, 2025].

[2] F. P. Incropera, *Fundamentals of Heat and Mass Transfer*, Hoboken, NJ: John Wiley, 2007.

[3] University of Cambridge, "NST Part II – Mathematical Methods: Chapter 2." Available: `https://www.damtp.cam.ac.uk/user/reh10/lectures/nst-mmii-chapter2.pdf`. [Accessed: Aug. 7, 2025].

[4] J. Burkardt, "HEAT_MPI," Florida State University. Available: `https://people.sc.fsu.edu/~jburkardt/c_src/heat_mpi/heat_mpi.html`. [Accessed: Aug. 7, 2025].

[5] UVA Hotspot Team, "Uvahotspot/hotspot: Hotspot v7.0," *GitHub*. Available: `https://github.com/uvahotspot/hotspot`. [Accessed: Aug. 7, 2025].

[6] NVIDIA, "CUDA-SAMPLES/SAMPLES/1_UTILITIES/DEVICEQUERY/DEVICEQUERY.CPP at master - Nvidia/Cuda-samples," *GitHub*. Available: `https://github.com/NVIDIA/cuda-samples/blob/master/Samples/1_Utilities/deviceQuery/deviceQuery.cpp`. [Accessed: Aug. 8, 2025].

8 Appendix

8.1 Final main.cu

```
#include <stdio.h>
#include <stdint.h>
#include "support.h"
#include "kernel.cu"

// Set up parameters
const int GRID_SIZE = 256;
const int total_size = GRID_SIZE * GRID_SIZE;
const char* POWER_MAP_FILE = "/home/roesslera/code/MPP_project_temp_simulation/data/power_map_256.csv"
    ↪ ;
const double T_amb = 25; // Ambient temperature in Celcius
const int ITERATIONS = 50000;
const double DIE_WIDTH_M = 0.016;
const double h = DIE_WIDTH_M / GRID_SIZE;
const double k = 150.0; // thermal conductivity (using silicon)
double coeff = (h * h / k);

// For reduction kernel: hardcoded since we know the values and can keep structure of program in tact
const int THREADS_PER_BLOCK = 256; // blockDim.x * blockDim.y
const int BLOCKS = 256; // (GRID_SIZE * GRID_SIZE + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK;

int load_power_map(const char* filename, double* q) {
    // Confirm file opens
    FILE* file = fopen(filename, "r");
    if (!file) {
        fprintf(stderr, "Error opening file:\n");
        return 1;
    }

    // Read file data
    for (int i = 0; i < total_size; i++) {
        fscanf(file, "%lf,", &q[i]);
    }

    fclose(file);
    return 0;
}


int main(int argc, char* argv[])
{
    Timer timer, total_timer, timer_copy, timer_max_device, timer_max_host, timer_kernel;
    float t_copy = 0, t_kernel = 0, t_max_host = 0, t_max_device = 0;
    startTime(&total_timer);
    cudaError_t cuda_ret;

    // Initialize host variables ----------------------------------------------

    printf("\nSetting up the problem..."); fflush(stdout);
    startTime(&timer);

    double *q_h, *T_h, *T_new_h, *max_diff_h;

    q_h = (double*) malloc( sizeof(double) * total_size );
    T_h = (double*) malloc( sizeof(double) * total_size );
    T_new_h = (double*) malloc( sizeof(double) * total_size );
    max_diff_h = (double*) malloc(sizeof(double) * BLOCKS);


    for (unsigned int i=0; i < total_size; i++) { T_new_h[i] = T_amb; T_h[i] = T_amb; }
    if (load_power_map(POWER_MAP_FILE, q_h) != 0) {
        fprintf(stderr, "Failed to load power map.\n");
```

```
        return 1;
    }

    stopTime(&timer); printf("%f s\n", elapsedTime(timer));

    // Allocate device variables ----------------------------------------------

    printf("Allocating device variables..."); fflush(stdout);
    startTime(&timer);

    double *q_d, *T_d, *T_new_d, *max_diff_d;

    // CUDA device variables for q, T, and T_new
    cuda_ret = cudaMalloc((void**)&q_d, sizeof(double)* total_size);
    if(cuda_ret != cudaSuccess) FATAL("Unable to allocate device memory for q_d");

    cuda_ret = cudaMalloc((void**)&T_d, sizeof(double)* total_size);
    if(cuda_ret != cudaSuccess) FATAL("Unable to allocate device memory for T_d");

    cuda_ret = cudaMalloc((void**)&T_new_d, sizeof(double)* total_size);
    if(cuda_ret != cudaSuccess) FATAL("Unable to allocate device memory for T_new_d");

    // Allocate device variable for max_diff
    cudaMalloc(&max_diff_d, BLOCKS * sizeof(double));

    cudaDeviceSynchronize();
    stopTime(&timer); printf("%f s\n", elapsedTime(timer));

    // Copy host variables to device ------------------------------------------

    printf("Copying data from host to device..."); fflush(stdout);
    startTime(&timer);

    // Copy q, T, and T_new from host to device
    cuda_ret = cudaMemcpy(q_d, q_h, sizeof(double)*total_size, cudaMemcpyHostToDevice);
    if(cuda_ret != cudaSuccess) FATAL("Unable to copy q from host to device");
    cuda_ret = cudaMemcpy(T_d, T_h, sizeof(double)*total_size, cudaMemcpyHostToDevice);
    if(cuda_ret != cudaSuccess) FATAL("Unable to copy T from host to device");
    cuda_ret = cudaMemcpy(T_new_d, T_new_h, sizeof(double)*total_size, cudaMemcpyHostToDevice);
    if(cuda_ret != cudaSuccess) FATAL("Unable to copy T_new from host to device");

    cudaDeviceSynchronize();
    stopTime(&timer); printf("%f s\n", elapsedTime(timer));

    // Launch kernel ----------------------------
    startTime(&timer);

    // Define grid and block dimensions
    dim3 blockDim(16, 16);
    dim3 gridDim((GRID_SIZE + blockDim.x - 1) / blockDim.x, (GRID_SIZE + blockDim.y - 1) / blockDim.y)
        ↪ ;


    // Launch the kernel
    int iter;
    for (iter = 0; iter < ITERATIONS; iter++) {
        startTime(&timer_kernel);
        compute_temperature<<<gridDim, blockDim>>>(T_d, T_new_d, q_d, coeff, GRID_SIZE, T_amb,
            ↪ max_diff_d);
        cuda_ret = cudaDeviceSynchronize();
        stopTime(&timer_kernel); t_kernel += elapsedTime(timer_kernel);
        cuda_ret = cudaGetLastError();
        if(cuda_ret != cudaSuccess) FATAL("Unable to launch kernel");

        // Copy max_diff_d from device to host
        startTime(&timer_copy);
```

```
126         cudaMemcpy(max_diff_h, max_diff_d, sizeof(double) * BLOCKS, cudaMemcpyDeviceToHost);
127         stopTime(&timer_copy); t_copy += elapsedTime(timer_copy);
128 
129         // Complete reduction on host
130         startTime(&timer_max_host);
131         double max_change = 0.0;
132         for (int i = 0; i < BLOCKS; i++) {
133             if (max_diff_h[i] > max_change) {
134                 max_change = max_diff_h[i];
135             }
136         }
137         stopTime(&timer_max_host); t_max_host += elapsedTime(timer_max_host);
138 
139         // Check for convergence
140         if (max_change < 1e-3) {
141             printf("Converged after %d iterations\n", iter);
142             break;
143         }
144 
145         // Swap T and T_new pointers
146         double* temp = T_d;
147         T_d = T_new_d;
148         T_new_d = temp;
149     }
150 
151     cuda_ret = cudaDeviceSynchronize();
152 
153     // Copy device variables from host ----------------------------------------
154 
155     printf("Copying data from device to host..."); fflush(stdout);
156     startTime(&timer);
157 
158     cuda_ret = cudaMemcpy(T_new_h, T_new_d, sizeof(double)*total_size, cudaMemcpyDeviceToHost);
159     if(cuda_ret != cudaSuccess) FATAL("Unable to copy memory from device");
160 
161     cudaDeviceSynchronize();
162     stopTime(&timer); printf("%f s\n", elapsedTime(timer));
163 
164     double max_temp_T = max_temp(T_new_h, GRID_SIZE);
165     double min_temp_T = min_temp(T_new_h, GRID_SIZE);
166     double avg_temp_T = avg_temp(T_new_h, GRID_SIZE);
167 
168     printf("Max Temp: %.2f C\n", max_temp_T);
169     printf("Min Temp: %.2f C\n", min_temp_T);
170     printf("Avg Temp: %.2f C\n", avg_temp_T);
171 
172     // Verify correctness -----------------------------------------------------
173 
174     printf("Verifying results..."); fflush(stdout);
175     verify(iter, max_temp_T, min_temp_T, avg_temp_T);
176 
177     // Free memory ------------------------------------------------------------
178 
179     free(q_h);
180     free(T_h);
181     free(T_new_h);
182     free(max_diff_h);
183 
184     // Free device variables
185     cudaFree(q_d);
186     cudaFree(T_d);
187     cudaFree(T_new_d);
188     cudaFree(max_diff_d);
189 
190     printf("Copy reduced Temperature to host for convergence check: %.5f s\n", t_copy);
191     printf("Reduce Temperature further on host for convergence check: %.5f s\n", t_max_host);
```

```
    printf("Kernel execution time: %.5f s\n", t_kernel);

    stopTime(&total_timer); printf("Total Execution Time: %f s\n", elapsedTime(total_timer));

    return 0;
}
```

8.2 Final kernel.cu

```
// Kernel algorithm for temperature computation
__global__ void compute_temperature(double* T, double* T_new, double* q, double coeff,
    int grid_size, double T_amb, double* max_diff_per_block) {

    // Indicies
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    int idx = y * grid_size + x;
    int local_idx = threadIdx.y * blockDim.x + threadIdx.x;

    // Shared memory for block reduction
    __shared__ double s_data[256];

    double difference = 0.0;

    // Bundle temperature computations, boundary checks, and difference calculation
    if (x < grid_size && y < grid_size) {
        if (x == 0 || x == grid_size - 1 || y == 0 || y == grid_size - 1) {
            // Boundary condition
            T_new[idx] = T_amb;
        } else {

            // Extract neighbors
            int top = (y - 1) * grid_size + x;
            int bottom = (y + 1) * grid_size + x;
            int left = y * grid_size + (x - 1);
            int right = y * grid_size + (x + 1);

            // Perform temperature calculation
            coeff *= q[idx];
            T_new[idx] = (T[top] + T[bottom] + T[left] + T[right] + coeff) / 4.0;
        }
        difference = fabs(T_new[idx] - T[idx]); // Calculate the difference compared to previous
    }

    s_data[local_idx] = difference;
    __syncthreads();

    // Parallel reduction to find max diff per block
    for (int stride = blockDim.x * blockDim.y / 2; stride > 0; stride /= 2) {
        if (local_idx < stride) {
            s_data[local_idx] = fmax(s_data[local_idx], s_data[local_idx + stride]);
        }
        __syncthreads();
    }

    // Write max diff of this block to global memory
    if (local_idx == 0) {
        int block_id = blockIdx.y * gridDim.x + blockIdx.x;
        max_diff_per_block[block_id] = s_data[0];
    }
}


// Compute the maximum, minimum, and average temperature in the grid
```

```
double max_temp(double* arr, int grid_size) {
    double max_val = arr[0];
    for (int i = 0; i < grid_size * grid_size; i++) {
        if (arr[i] > max_val) max_val = arr[i];
    }
    return max_val;
}
double min_temp(double* arr, int grid_size) {
    double min_val = arr[0];
    for (int i = 0; i < grid_size * grid_size; i++) {
        if (arr[i] < min_val) min_val = arr[i];
    }
    return min_val;
}
double avg_temp(double* arr, int grid_size) {
    double sum = 0.0;
    for (int i = 0; i < grid_size * grid_size; i++) {
        sum += arr[i];
    }
    return (sum / (grid_size * grid_size));
}
```

8.3 power_map_generation.py

```
import numpy as np
import matplotlib.pyplot as plt
import csv

np.random.seed(42) # Set seed for consistent data

# EV6 floorplan units: name, width (m), height (m), x (m), y (m)
floorplan = [
    ("L2_left", 0.004900, 0.006200, 0.000000, 0.009800),
    ("L2", 0.016000, 0.009800, 0.000000, 0.000000),
    ("L2_right", 0.004900, 0.006200, 0.011100, 0.009800),
    ("Icache", 0.003100, 0.002600, 0.004900, 0.009800),
    ("Dcache", 0.003100, 0.002600, 0.008000, 0.009800),
    ("Bpred_0", 0.001033, 0.000700, 0.004900, 0.012400),
    ("Bpred_1", 0.001033, 0.000700, 0.005933, 0.012400),
    ("Bpred_2", 0.001033, 0.000700, 0.006967, 0.012400),
    ("DTB_0", 0.001033, 0.000700, 0.008000, 0.012400),
    ("DTB_1", 0.001033, 0.000700, 0.009033, 0.012400),
    ("DTB_2", 0.001033, 0.000700, 0.010067, 0.012400),
    ("FPAdd_0", 0.001100, 0.000900, 0.004900, 0.013100),
    ("FPAdd_1", 0.001100, 0.000900, 0.006000, 0.013100),
    ("FPReg_0", 0.000550, 0.000380, 0.004900, 0.014000),
    ("FPReg_1", 0.000550, 0.000380, 0.005450, 0.014000),
    ("FPReg_2", 0.000550, 0.000380, 0.006000, 0.014000),
    ("FPReg_3", 0.000550, 0.000380, 0.006550, 0.014000),
    ("FPMul_0", 0.001100, 0.000950, 0.004900, 0.014380),
    ("FPMul_1", 0.001100, 0.000950, 0.006000, 0.014380),
    ("FPMap_0", 0.001100, 0.000670, 0.004900, 0.015330),
    ("FPMap_1", 0.001100, 0.000670, 0.006000, 0.015330),
    ("IntMap", 0.000900, 0.001350, 0.007100, 0.014650),
    ("IntQ", 0.001300, 0.001350, 0.008000, 0.014650),
    ("IntReg_0", 0.000900, 0.000670, 0.009300, 0.015330),
    ("IntReg_1", 0.000900, 0.000670, 0.010200, 0.015330),
    ("IntExec", 0.001800, 0.002230, 0.009300, 0.013100),
    ("FPQ", 0.000900, 0.001550, 0.007100, 0.013100),
    ("LdStQ", 0.001300, 0.000950, 0.008000, 0.013700),
    ("ITB_0", 0.000650, 0.000600, 0.008000, 0.013100),
    ("ITB_1", 0.000650, 0.000600, 0.008650, 0.013100),
]
# Power values in Watts for each unit "block" of the floorplan
```

```
power_values = {
    "L2_left": 1.44,
    "L2": 7.37,
    "L2_right": 1.44,
    "Icache": 8.27,
    "Dcache": 14.3,
    "Bpred_0": 1.51666666666667,
    "Bpred_1": 1.51666666666667,
    "Bpred_2": 1.51666666666667,
    "DTB_0": 0.0596666666666667,
    "DTB_1": 0.0596666666666667,
    "DTB_2": 0.0596666666666667,
    "FPAdd_0": 0.62,
    "FPAdd_1": 0.62,
    "FPReg_0": 0.19375,
    "FPReg_1": 0.19375,
    "FPReg_2": 0.19375,
    "FPReg_3": 0.19375,
    "FPMul_0": 0.665,
    "FPMul_1": 0.665,
    "FPMap_0": 0.02355,
    "FPMap_1": 0.02355,
    "IntMap": 1.07,
    "IntQ": 0.365,
    "IntReg_0": 2.585,
    "IntReg_1": 2.585,
    "IntExec": 7.7,
    "FPQ": 0.0354,
    "LdStQ": 3.46,
    "ITB_0": 0.2,
    "ITB_1": 0.2,
}

GRID_SIZE = 256

max_x = 0
max_y = 0
for _, w, h, x, y in floorplan:
    max_x = max(max_x, x + w)
    max_y = max(max_y, y + h)

def scale_to_grid(x_m, y_m):
    return int(x_m / max_x * GRID_SIZE), int(y_m / max_y * GRID_SIZE)

def scale_size_to_grid(w_m, h_m):
    return max(1, int(w_m / max_x * GRID_SIZE)), max(1, int(h_m / max_y * GRID_SIZE))

power_map = np.zeros((GRID_SIZE, GRID_SIZE))

for name, w_m, h_m, x_m, y_m in floorplan:
    px, py = scale_to_grid(x_m, y_m)
    pw, ph = scale_size_to_grid(w_m, h_m)

    x_end = min(px + pw, GRID_SIZE)
    y_end = min(py + ph, GRID_SIZE)

    base_power = power_values.get(name, 0)

    block_width = x_end - px
    block_height = y_end - py

    # Start with ones for equal distribution
    block_power_map = np.ones((block_height, block_width))

    # Add random multiplicative variations in fine blocks
    num_fine_blocks = np.random.randint(20, 40)
```

```
    for _ in range(num_fine_blocks):
        fw = np.random.randint(2, max(3, block_width // 6))
        fh = np.random.randint(2, max(3, block_height // 6))
        fx = np.random.randint(0, block_width - fw + 1)
        fy = np.random.randint(0, block_height - fh + 1)
        deviation = np.random.uniform(0.6, 1.4)
        block_power_map[fy:fy+fh, fx:fx+fw] *= deviation

    # Add small multiplicative noise (around 1)
    pixel_noise = np.random.normal(loc=1.0, scale=0.05, size=(block_height, block_width))
    block_power_map *= pixel_noise

    # Clip negative values to zero
    block_power_map = np.clip(block_power_map, 0, None)

    # Normalize so sum equals the base power of the block
    block_power_map *= base_power / np.sum(block_power_map)

    power_map[py:y_end, px:x_end] = block_power_map



# Compute power density map
DIE_WIDTH_M = 0.016
DIE_HEIGHT_M = 0.016
pixel_area = (DIE_WIDTH_M * DIE_HEIGHT_M) / (GRID_SIZE * GRID_SIZE) # m^2

# Power density = Power / Area per pixel
pixel_volume = pixel_area * 0.001 # Assuming thickness of 1 mm
pixel_volume = pixel_area * 0.0005 # FOR 256
power_density_map = power_map / pixel_volume # Units: W/m^3

print("Max power density (W/m^3):", np.max(power_density_map))
print("Min power density (W/m^3):", np.min(power_density_map))
print("Total power (W):", np.sum(power_map))

# Visualize power density
plt.figure(figsize=(10, 10))
plt.imshow(power_density_map, cmap='hot', interpolation='nearest', origin='lower')
plt.colorbar(label="Power Density (W/m^3)")
plt.title("Power Density Map")
plt.xlabel("Grid X")
plt.ylabel("Grid Y")

# Save the figure as a PNG file
plt.savefig("power map generation/power_density_map.png", dpi=300, bbox_inches='tight')


plt.show()

# Save power density map to CSV
with open("power_map_simplified_256.csv", "w", newline="") as f:
    writer = csv.writer(f)
    for row in power_density_map:
        writer.writerow([f"{val:.8f}" for val in row])
```

8.4 visualize_temp.py

```
import numpy as np
import matplotlib.pyplot as plt

# Load temperature data
T = np.loadtxt("data/results.csv", delimiter=",")

```

```
# Plot
plt.figure(figsize=(8, 6))
heatmap = plt.imshow(T, cmap='inferno', origin='lower')
plt.colorbar(heatmap, label='Temperature (C)')
plt.title('Steady-State Temperature Distribution')
plt.xlabel('X position')
plt.ylabel('Y position')
plt.savefig("temperature_heatmap.png", dpi=300)
plt.show()
```

8.5 data_analysis.py

```
import matplotlib.pyplot as plt
import pandas as pd

data = {
    "iteration": [0, 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000,
                  10000, 11000, 12000, 13000, 14000, 15000, 16000, 17000,
                  18000, 19000, 20000, 21000, 22000, 23000],
    "max_change": [0.09110, 0.01534, 0.01226, 0.00975, 0.00793, 0.00661,
                   0.00562, 0.00486, 0.00426, 0.00376, 0.00336, 0.00302,
                   0.00272, 0.00247, 0.00225, 0.00205, 0.00188, 0.00173,
                   0.00159, 0.00146, 0.00135, 0.00124, 0.00115, 0.00106],
    "max_temp": [25.09110, 44.52675, 57.09434, 67.55969, 76.19184, 83.37226,
                 89.42951, 94.60199, 99.07686, 102.98722, 106.42706, 109.47416,
                 112.19540, 114.62837, 116.81220, 118.78867, 120.57765,
                 122.19867, 123.67093, 125.01086, 126.23569, 127.36137,
                 128.39181, 129.33582]
}

df = pd.DataFrame(data)

# Plot Max Temperature
plt.figure(figsize=(10, 5))
plt.plot(df["iteration"], df["max_temp"], marker='o', label='Max Temperature')
plt.xlabel("Iteration")
plt.ylabel("Max Temperature (Celsius)")
plt.title("Max Temperature vs Iteration")
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.show()

# Plot Max Change
plt.figure(figsize=(10, 5))
plt.plot(df["iteration"], df["max_change"], marker='o', color='orange', label='Max Change')
plt.xlabel("Iteration")
plt.ylabel("Max Change")
plt.title("Convergence Rate (Max Change per Iteration)")
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.show()
```