# Homework Assignment 5: Heat Diffusion Using Multi-GPU CUDA Programming

Haohan Tsao B09607009

## 1 Introduction

This report presents the implementation and analysis of solving the heat diffusion equation on a two-dimensional Cartesian grid using CUDA multi-GPU parallel programming. The heat diffusion equation is a fundamental partial differential equation that describes the distribution of heat (or variation of temperature) in a medium over time. In the steady-state condition, this reduces to Laplace's equation.

The steady-state heat diffusion equation is given by:

$$\nabla^2 u = 0 \tag{1}$$

Where:

- $u$ represents the temperature distribution
- $\nabla^2$ is the Laplacian operator

The assignment required solving this equation on a $1024 \times 1024$ Cartesian grid with specific boundary conditions:

- Top edge: 400 K (heated boundary)
- Other three edges: 273 K (cooled boundaries)

We were asked to investigate the optimal block size for this problem and compare the performance between single GPU and dual GPU implementations.

## 2 Theoretical Background

### 2.1 Heat Diffusion to Laplace Equation

The time-dependent heat diffusion equation is:

$$\frac{\partial u}{\partial t} = \alpha \nabla^2 u \tag{2}$$

Where $\alpha$ is the thermal diffusivity. In steady-state conditions ($\frac{\partial u}{\partial t} = 0$), this becomes Laplace's equation. However, we can use the time-dependent formulation as an iterative method to reach the steady-state solution.

### 2.2 Discretization on a 2D Grid

On a two-dimensional grid with spacing $\Delta x = \Delta y = 1$, the Laplacian operator can be discretized using the finite difference method:

$$\nabla^2 u(x, y) \approx u(x+1, y) + u(x-1, y) + u(x, y+1) + u(x, y-1) - 4u(x, y) \tag{3}$$

### 2.3 Iterative Solution Method

For the steady-state solution, we use the iterative Jacobi method:

$$u^{n+1}(x, y) = \frac{1}{4}[u^n(x+1, y) + u^n(x-1, y) + u^n(x, y+1) + u^n(x, y-1)] \tag{4}$$

This iterative approach converges to the solution of Laplace's equation with the given boundary conditions.

## 3 Experimental Setup

### 3.1 Implementation Approach

The implementation utilizes CUDA multi-GPU programming with the following key features:

1. **Multi-GPU Architecture**: Using OpenMP to manage multiple GPUs, with one CPU thread per GPU
2. **Domain Decomposition**: The 2D grid is partitioned among GPUs for parallel processing

3. **P2P Communication**: GPU-to-GPU communication for boundary data exchange
4. **Double Buffering**: Two arrays are used to avoid memory copying during iterations

### 3.2 Testing Methodology

To determine the optimal configuration, we performed experiments with:

**GPU Configurations:**

- Single GPU (1×1 arrangement)
- Dual GPU (2×1 arrangement)

**Block Sizes:**

- 8×8 threads per block
- 16×16 threads per block
- 32×32 threads per block

**Problem Size:**

- Grid size: 1024×1024
- Convergence criterion: $\epsilon = 10^{-10}$
- Maximum iterations: 10,000,000

## 4 Implementation Details

### 4.1 Source Code Structure

The implementation consists of the following key components:

- `heat_diffusion.cu`: Main CUDA source file containing host code and device kernels
- `generate_experiments.sh`: Automated experiment generation script
- `submit_all.sh`: Batch job submission script
- `collect_results.sh`: Results collection and analysis script
- `Makefile`: Compilation configuration

## 4.2 Key CUDA Implementation Features

```
__global__ void heat_diffusion(float* phi0_old, float* phiL_old,
                               float* phiR_old, float* phiB_old,
                               float* phiT_old, float* phi0_new,
                               float* C)
{
    // Calculate thread indices
    int x = blockDim.x*blockIdx.x + threadIdx.x;
    int y = blockDim.y*blockIdx.y + threadIdx.y;
    int site = x + y*Lx;

    // Skip boundary points
    if (skip == 0) {
        // Apply 4-point stencil
        phi0_new[site] = 0.25*(b+l+r+t);
        diff = phi0_new[site] - phi0_old[site];
    }

    // Parallel reduction for error calculation
    cache[cacheIndex] = diff*diff;
    __syncthreads();
    // ... reduction logic
}
```

Listing 1: Core heat diffusion kernel implementation

## 4.3 Multi-GPU Communication Strategy

For multi-GPU implementation, P2P communication is established between neighboring GPUs:

```
// Enable P2P access between neighboring GPUs
if (NGPU > 1) {
    int cpuid_r = ((cpuid_x+1)%NGx) + cpuid_y*NGx;
    cudaDeviceEnablePeerAccess(Dev[cpuid_r], 0);
    int cpuid_l = ((cpuid_x+NGx-1)%NGx) + cpuid_y*NGx;
    cudaDeviceEnablePeerAccess(Dev[cpuid_l], 0);
}
```

Listing 2: P2P communication setup

# 5 Results and Analysis

## 5.1 Performance Comparison

The following table summarizes the performance results for all tested configurations:

| GPU Count | Block Size | Time (ms) | Gflops | Iterations |
|---|---|---|---|---|
| 1 | 8×8 | 124,809 | 43.13 | 736,246 |
| 1 | 16×16 | 96,977 | 55.51 | 736,246 |
| 1 | 32×32 | 120,626 | 44.63 | 736,246 |
| 2 | 8×8 | 80,869 | 66.56 | 736,246 |
| 2 | 16×16 | 57,593 | 93.47 | 736,246 |
| 2 | 32×32 | 66,007 | 81.55 | 736,246 |

Table 1: Performance comparison for different GPU and block size configurations

## 5.2 Optimal Block Size Analysis

The results clearly demonstrate that **16×16 block size is optimal** for this problem:

| Block Size | Single GPU Gflops | Dual GPU Gflops | Performance Ratio |
|---|---|---|---|
| 8×8 | 43.13 | 66.56 | 1.54× |
| 16×16 | 55.51 | 93.47 | 1.68× |
| 32×32 | 44.63 | 81.55 | 1.83× |

Table 2: Performance comparison across block sizes

## 5.3 Multi-GPU Scaling Analysis

The multi-GPU implementation shows excellent scaling efficiency:

- **Best Configuration**: 2 GPUs with 16×16 blocks achieving 93.47 Gflops
- **Speedup**: 1.68× improvement over single GPU
- **Efficiency**: 84% of ideal 2× speedup
- **Consistent Convergence**: All configurations converged in exactly 736,246 iterations

## 6 Discussion

### 6.1 Block Size Optimization

The superior performance of 16×16 blocks can be attributed to several factors:

- **Memory Coalescing**: 16×16 provides optimal memory access patterns for the 2D grid
- **Occupancy**: Balances thread count (256 threads) with resource utilization
- **Shared Memory Usage**: Efficient utilization without bank conflicts
- **Warp Alignment**: Better alignment with GPU warp architecture (32 threads)

The poor performance of 32×32 blocks (1024 threads) suggests resource limitations, while 8×8 blocks underutilize the GPU cores.

### 6.2 Multi-GPU Efficiency

The multi-GPU implementation demonstrates strong scaling:

- **Load Balancing**: Even distribution of computational work
- **Communication Overhead**: Minimal impact from P2P boundary exchange
- **Synchronization**: Efficient OpenMP barriers between iterations
- **Memory Bandwidth**: Effective utilization of aggregate memory bandwidth

### 6.3 Convergence Behavior

All configurations converged to the same solution in identical iteration counts, confirming:

- **Mathematical Consistency**: GPU count and block size do not affect convergence
- **Numerical Accuracy**: Proper implementation of boundary conditions
- **Algorithm Stability**: Robust iterative solver implementation

## 7 Conclusion

This study successfully demonstrates the implementation and optimization of a multi-GPU CUDA solver for the heat diffusion equation. The key findings are:

1. **Optimal Configuration**: 2 GPUs with 16×16 block size provides the best performance at 93.47 Gflops
2. **Block Size Impact**: 16×16 consistently outperforms other block sizes across all GPU configurations
3. **Multi-GPU Scaling**: Achieves 84% efficiency with 1.68× speedup using dual GPUs
4. **Robust Implementation**: Consistent convergence behavior regardless of parallelization strategy

The implementation showcases the effectiveness of multi-GPU computing for large-scale numerical simulations, with proper domain decomposition and P2P communication enabling near-linear scaling. The 16×16 block size emerges as the optimal choice, balancing computational efficiency with hardware resource utilization.

For production use, the dual-GPU configuration with 16×16 blocks is recommended for maximum performance, while the single-GPU implementation with 16×16 blocks provides an efficient solution for memory-constrained scenarios.

## References

- CUDA Programming Guide, NVIDIA Corporation
- OpenMP Specification, OpenMP Architecture Review Board
- Numerical Methods for Partial Differential Equations, Morton & Mayers