GPU Acceleration of IC Thermal Simulation

Abhishek Upadhya 24B1309

1 Introduction

Thermal analysis is a critical component of modern integrated circuit (IC) design, as increasing power densities directly affect reliability and performance. Numerical simulation of heat diffusion is commonly used to estimate steady-state and transient temperature profiles across a chip.

This project accelerates a two-dimensional IC thermal simulation using GPU programming. A CPU baseline implementation is compared against custom CUDA and Triton GPU kernels, with a focus on quantitative performance evaluation.

2 Physical Model

The simulation solves the two-dimensional heat equation with a source term:

$$T_{i,j}^{t+1} = T_{i,j}^{t} + \alpha \Delta t \nabla^2 T_{i,j} + P_{i,j}$$

where T denotes temperature, P is a power density map, α is the thermal diffusivity, and Δt is the time step.

The Laplacian operator is discretized using a standard 5-point stencil:

$$\nabla^2 T_{i,j} = T_{i+1,j} + T_{i-1,j} + T_{i,j+1} + T_{i,j-1} - 4T_{i,j}$$

Boundary cells are held fixed and excluded from updates.

3 CPU Baseline

The CPU baseline is implemented in Python using NumPy arrays and explicit nested loops over grid indices. While simple and readable, this approach suffers from severe performance limitations due to Python loop overhead and poor cache utilization.

The simulation is run on a 1024×1024 grid for 1000 time steps, resulting in approximately 10^9 stencil updates. The measured runtime for the CPU implementation is 1696.55 seconds.

4 CUDA Implementation

The CUDA implementation assigns one GPU thread to each grid cell. Threads are launched in a two-dimensional grid matching the spatial layout of the temperature field. Each thread loads its neighboring values from global memory, computes the Laplacian, and updates the temperature.

The kernel is compiled and launched using a custom PyTorch C++/CUDA extension. One kernel launch corresponds to one simulation time step. One-time compilation overhead is excluded from benchmark measurements.

5 Triton Implementation

A Triton kernel is implemented using a linearized grid representation. Each Triton program instance processes a contiguous block of grid elements. Triton provides a higher-level abstraction over CUDA while still enabling efficient memory access patterns and parallel execution.

The Triton implementation performs the same computation as the CUDA kernel and is benchmarked under identical conditions.

6 Results

All experiments are conducted on an NVIDIA GTX 1650 Ti GPU. The measured runtimes are shown in Table 1.

Method	Grid Size	Time (s)	Speedup
CPU (NumPy)	1024×1024	1696.55	$1\times$
CUDA Kernel	1024×1024	0.0888	$19100\times$
Triton Kernel	1024×1024	0.0880	$19280\times$

Table 1: Performance comparison of CPU, CUDA, and Triton implementations.

7 Analysis

The observed speedup arises from massive parallelism and significantly higher memory bandwidth available on the GPU. The kernel is primarily memory-bound, as each update requires multiple global memory loads.

Despite using different programming models, CUDA and Triton achieve nearly identical performance, indicating that both effectively map the stencil computation to GPU hardware. Triton offers improved developer productivity, while CUDA provides finer control over low-level execution details.

8 Conclusion

This project demonstrates that GPU acceleration can improve IC thermal simulation performance by over four orders of magnitude compared to a naive CPU implementation. Both CUDA and Triton are effective tools for accelerating stencil-based numerical workloads. Future work could explore shared memory tiling, multi-GPU scaling, or implicit time integration methods.