WiDS Week 1

Task 1: Identify and Analyse a GPU-Accelerable Workload

Workload: 2D Thermal Simulation of an IC

1.1 Operation Breakdown

1.1.1 Description

There is both static and dynamic power dissipation in an IC. Transistors on an IC behave differently in different temperatures. In order for us to efficiently map transistors onto the IC, we need to first simulate the heat map on the IC. This helps us understand where we can and cannot place components on the IC.
We can model the heat diffusion across the chip by representing the floorplan as a 2D grid. The temperature of a grid cell depends on only two things: whether it has a power source or not and the temperature of the surrounding cells. Thus, we can compute the temperature of each cell independently.

1.1.2 Formula & Pseudocode

$$\frac{\partial T}{\partial t} = \alpha \left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \right)$$

$$T_{i,j}^{(t+1)} = T_{i,j}^{(t)} + \alpha \Delta t \left(\frac{T_{i+1,j}^{(t)} - 2T_{i,j}^{(t)} + T_{i-1,j}^{(t)}}{\Delta x^2} + \frac{T_{i,j+1}^{(t)} - 2T_{i,j}^{(t)} + T_{i,j-1}^{(t)}}{\Delta y^2} \right) + P_{i,j}$$

Where,

- alpha is the thermal diffusivity
- T is the temperature of a given cell
- P is the power dissipated at that grid cell

Below is the pseudocode for the above formula (CPU implementation)

```
for t in range(T_max):
    for i in range(1, H-1):
        for j in range(1, W-1):
            laplacian = T[i+1][j] + T[i-1][j] + T[i][j+1]
                + T[i][j-1] - 4 * T[i][j]
            T_new[i][j] = T[i][j] + alpha * dt * laplacian + P[i][j]
    swap(T, T_new)
```

1.1.3 Specifications

Input size would be the size of the 2D grid. We can have a grid size of 1024 x 1024 for ~ 1M cells. The loop would probably run for 1k iterations but that would depend on when we get the actual answer. The actual number can only be calculated after testing.

1.2 Compute vs Memory Analysis

1.2.1 Operation Bounds

The calculation of the temperature in each cell is not computation heavy. The bound comes from the thread having to load the temperature values of the adjacent cells. Thus this is memory bound.

1.2.2 Arithmetic Intensity

Arithmetic intensity is the ratio of arithmetic operations to memory operations in a kernel. We are not computing much per grid cell. Thus, the arithmetic intensity is quite low.

1.2.3 Reuse Opportunities

There is a high opportunity for using shared memory. The temperature stored in a single grid cell is used by four other threads. Having a shared memory can be extremely beneficial.

1.2.4 Limitations

Threads must be synchronised before a time step starts. This is not much of an issue since all threads are doing the same amount of computation. Only the threads handling cells at the boundary require condition handling.

1.3 Expected Behaviour on a GPU

1.3.1 Mapping the Threads

1 CUDA thread is mapping onto a single grid cell. A thread block can map to a small tile on the chip.

1.3.2 Scaling

For a 1024 x 1024 grid, there are ~ 1M cells. This requires 1 million threads per time step which can be easily handled by modern GPUs. It is only limited by memory.

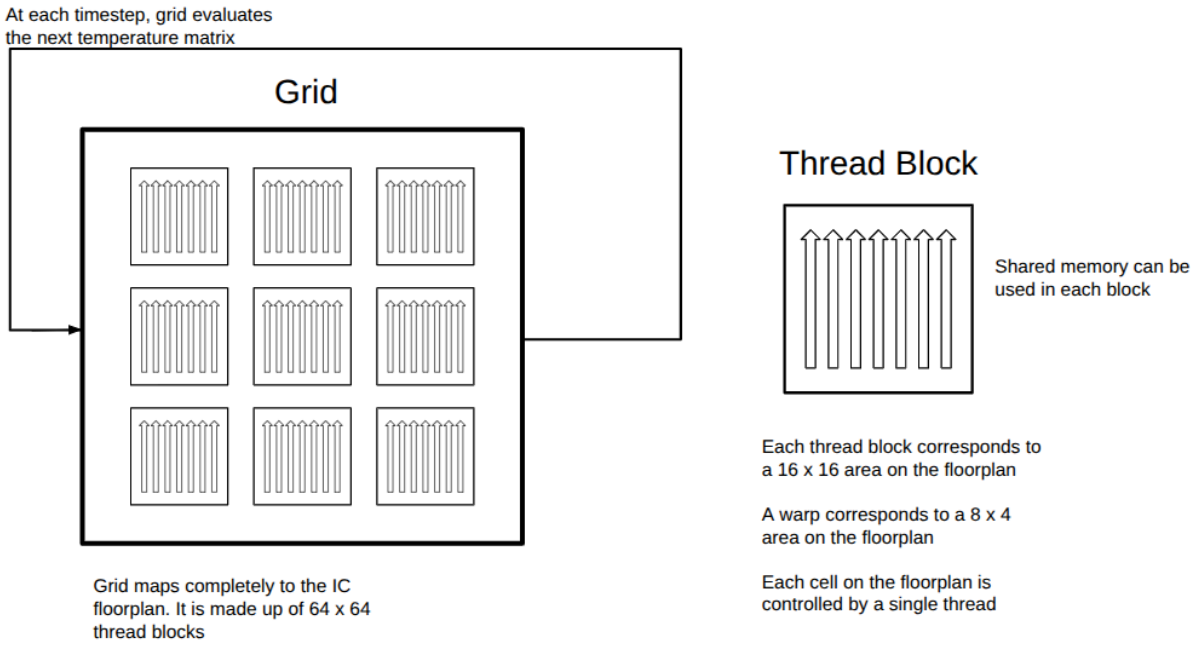
1.3.3 Challenges

There is a low chance of warp divergence since every thread runs the same instructions. Only the boundary checks can create divergence. The only problem would arise in incorrectly using the shared memory since it is limited by the GPU's capacity.

1.4 CPU Baseline

The CPU Baseline for 1k timesteps is **`1323.24s`**

Task2: CUDA Execution Model Mapping Diagram



WiDS Week 2

Task 1: Implement Basic CUDA Kernels

Three basic CUDA kernels vector addition, elementwise multiply-and-scale, and ReLU were implemented. Each kernel allocates GPU memory with cudaMalloc, transfers data using cudaMemcpy, launches a one-dimensional grid with explicit bounds checking, and copies results back to the host. Correctness was verified by comparing GPU outputs against a CPU reference implementation, and all programs compiled cleanly with nvcc without warnings.

Task 2: Grid and Block Configuration Exploration

gridSize = (n + blockSize - 1) / blockSize

Vector size n = 1000

blockDim.x = 32	gridDim.x = 32	threads launched = 1024
blockDim.x = 128	gridDim.x = 8	threads launched = 1024
blockDim.x = 256	gridDim.x = 4	threads launched = 1024
blockDim.x = 512	gridDim.x = 2	threads launched = 1024

Vector size n = 100000

blockDim.x = 32	gridDim.x = 3125	threads launched = 100000
blockDim.x = 128	gridDim.x = 782	threads launched = 100096
blockDim.x = 256	gridDim.x = 391	threads launched = 100096
blockDim.x = 512	gridDim.x = 196	threads launched = 100352

Vector size n = 10000000

blockDim.x = 32	gridDim.x = 312500	threads launched = 10000000
blockDim.x = 128	gridDim.x = 78125	threads launched = 10000000
blockDim.x = 256	gridDim.x = 39063	threads launched = 10000128
blockDim.x = 512	gridDim.x = 19532	threads launched = 10000384

2.1 What happens if the total number of threads exceeds n?

When the total threads launched exceeds n, we have threads which do not do any computation whatsoever. They just exit without performing any memory access.

2.2 Why is bounds checking required inside the kernel?

Bounds checking is required to prevent threads with thread indices greater than n from accessing memory beyond the original array. Since CUDA launches threads in blocks, it is possible for the total number of threads to exceed the size of the array making it necessary to check the bounds for correctness.

2.3 Which configurations failed or behaved unexpectedly, if any?

Larger block sizes caused more threads to be launched than what was required. Smaller block sizes actually ended up launching the exact number of threads. All configurations worked though.

2.4 Based on Task 2, when might you choose a smaller block size vs. a larger one? What trade-offs do you anticipate?

Smaller block size may be preferred when we need to minimise the number of excess threads. Larger block sizes can be useful for better resource utilisation.

WiDS Week 3

Task 1: Global Memory Access Patterns

For a vector with a size of 1.6M elements, the execution times are as follows:

	Coalesced	Non-coalesced
Execution Time	1.2 ms	30.2 ms

Both kernels performed the same operation on the same amount of data. The non-coalesced version was approximately 25 times slower than the coalesced version.

For the coalesced kernel, consecutive threads within a warp access consecutive memory locations. Thus, all 32 memory requests in a warp can be combined into a smaller number of global memory transactions. In the non-coalesced kernel, consecutive threads access memory elements separated by a stride. The GPU is unable to combine these requests and has to make 32 separate global memory transactions instead of one. This leads to a large increase in execution time. The observed 25x slowdown directly reflects the loss of coalescing efficiency. Without any global memory access patterns, the GPU is unable to merge memory requests from threads in a warp.

Task 2: Shared Memory Optimization

For a vector with a size of 1.6M elements, the execution times are as follows:

	Baseline Kernel	Shared memory
Execution Time	1.07 ms	1.56 ms

Both kernels performed the following operation:

$$b[i] = \sum_{k=-5}^{5} a[i+k]$$

In theory, the global kernels need to load 11 elements from the global memory each whereas in the case of the shared memory kernel, each kernel loads only one element from the global memory onto the shared memory on average. Therefore, shared memory should be faster than the baseline kernel by at least 11 times. However, this is not true.

In practice, the baseline kernel also loads memory onto the L1 cache which is a considerable optimization. Each kernel does not load 11 elements from the global memory. Most of the elements are already present in the L1 cache. Therefore, the baseline kernel is already highly optimized. In case of the shared memory, data is first loaded onto the shared memory and then loaded by the thread again for computation which is worse than just using the cache. It is quite difficult to find an example where using a shared memory is faster as modern GPUs are highly optimised and use caches super efficiently. One could argue that the cache is the shared memory.

Shared memory improves performance only when it reduces global memory traffic or latency that the hardware cache cannot already hide. Shared memory helps when reuse is real and caches are insufficient or unpredictable.

Shared memory in this example shouldn’t suffer from bank conflicts. A bank conflict is a performance penalty that happens when multiple threads in the same warp access different addresses that live in the same shared-memory bank at the same time. Here, consecutive threads map to consecutive banks. Thus, there aren’t any bank conflicts.

Task 3: CPU Baseline

Baseline was submitted in week 1 itself.

Execution time: 1323 s

Week 4: Real GPU Kernels for Machine Learning

Abhishek Upadhya, 24B1309

1 Task 1: Naive CUDA Kernel

We implement a numerically stable softmax operation:

$$\text{softmax}(x_i) = \frac{e^{x_i - \max(x)}}{\sum_j e^{x_j - \max(x)}}$$

The naive kernel uses:

- Global memory accesses
- Atomic operations for reductions
- One block per row

Observations:
The kernel was validated by the python script. The output matches the pytorch reference within $\sim 10^{-10}$ precision.

2 Task 2: Optimized Kernel

The optimized kernel improves performance by:

- Reusing data via shared memory
- Eliminating global atomics
- Performing tree-based reductions

Shared Memory Usage:
All row elements are loaded once and reused for max, sum, and normalization.

Observations:

The optimized kernel achieved a higher performance. The kernel was validated by the correctness script. Shared memory reuse and the elimination of atomic operations significantly reduce memory latency, resulting in improved throughput.

3 Task 3: Benchmarking and Performance Comparison

3.1 Benchmarking Methodology

Benchmarking was performed using CUDA events to obtain accurate GPU execution times. Each implementation was run for 200 iterations after warm-up runs, with CUDA synchronization enforced to ensure correct timing. All experiments used identical input tensors of size 1024×512 with single-precision floating-point values.

3.2 Benchmark Results

Table 1 reports the average execution time per iteration for each implementation.

Implementation	Time (ms)	Relative Speed
PyTorch Softmax	0.026	1.0×
Naive CUDA Kernel	0.321	0.08×
Optimized CUDA Kernel	0.068	0.38×

Table 1: Performance comparison of softmax implementations

3.3 Performance Analysis

The naive CUDA kernel performs poorly due to repeated global memory accesses and the use of atomic operations, which introduce serialization and high memory latency. As a result, it underperforms the PyTorch baseline despite parallel execution.

The optimized kernel achieves a 4.7× speedup over the naive version by reusing data through shared memory and eliminating atomic operations. PyTorch remains faster due to its highly optimized, architecture-specific kernels that use warp-level primitives and additional low-level optimizations.

3.4 Bottleneck Identification

The naive kernel is primarily memory-bound and limited by atomic serialization. The optimized kernel reduces global memory traffic, shifting the bottleneck toward computation and block-level synchronization. PyTorch further reduces these overheads through advanced kernel design.

4 Task 4: Reading Production GPU Code

4.1 tiny-cuda-nn

tiny-cuda-nn emphasizes kernel fusion and fine-grained tiling to maximize arithmetic intensity and minimize global memory accesses. Its kernels are highly specialized for fixed tensor shapes, whereas our implementation prioritizes generality and clarity.

4.2 FlashAttention

FlashAttention improves performance through algorithmic restructuring that avoids materializing large intermediate tensors, allowing computation to remain within on-chip memory. This contrasts with our kernel-level optimizations, which focus on memory reuse within a single kernel.

5 Conclusion

This assignment highlights the impact of memory-aware kernel design on GPU performance. While the naive kernel prioritizes correctness, it suffers from inefficient memory access patterns. The optimized kernel demonstrates how shared memory reuse and reduction restructuring can significantly improve performance while maintaining correctness.

Week 5 Assignment: CUDA vs Triton Softmax

Abhishek Upadhya 24B1309

1 Task 1: Triton Softmax Implementation

The chosen kernel is a row-wise, numerically stable softmax. For each row, the maximum element is subtracted before exponentiation to avoid numerical overflow, followed by normalization using the sum of exponentials. In Triton, each program instance processes one row of the input tensor using block-level parallelism.

Correctness was verified by comparing the Triton output against PyTorch's reference softmax implementation. The maximum absolute error observed was 7.45×10^{-9}, confirming numerical equivalence.

2 Task 2: Benchmarking and Performance Comparison

2.1 Experimental Setup

- GPU: NVIDIA GTX 1650 Ti
- Frameworks: PyTorch, Triton
- Input tensor: 2D tensor with softmax applied row-wise
- Timing method: wall-clock time with CUDA synchronization
- Metric: average runtime over multiple iterations

2.2 Results

Implementation	Time (ms)	Relative Speed
CUDA / PyTorch Softmax	0.1979	1.00×
Triton Softmax	0.1936	1.02×

Table 1: Runtime comparison between CUDA (PyTorch) and Triton softmax implementations.

The Triton implementation achieves performance comparable to the optimized CUDA-based PyTorch softmax on the tested input size, despite using a higher-level abstraction and minimal manual tuning.

3 Task 3: CUDA vs Triton — Design Reflection

The softmax kernel is easier to express in Triton due to its high-level handling of indexing, launch configuration, and memory access. Compared to CUDA, Triton provides less explicit control over warp-level behavior, shared memory usage, and fine-grained synchronization. For research and rapid prototyping, Triton is preferable due to faster development and readability, while CUDA remains better suited for scenarios requiring maximum performance tuning and architectural control.

4 Conclusion

This work demonstrates that Triton can achieve competitive performance with significantly reduced implementation complexity. While CUDA offers unmatched low-level control, Triton provides a productive alternative for writing correct and efficient GPU kernels in a research setting.

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.