Solving the Heat Equation:

A Case Study in CPU Bottlenecks vs. GPU Throughput

Dipanka Tanu Sarmah

January 21, 2026

1 The Physical Problem: 2D Heat Diffusion

To visualize the difference between CPU and GPU execution, we use the 2D Heat Equation (Laplace Diffusion). We model a metal plate where the edges are held at different temperatures. The temperature u at a point (i, j) at time $t + 1$ is the average of its four neighbors:

$$u_{i,j}^{t+1} = \frac{1}{4}(u_{i+1,j}^{t} + u_{i-1,j}^{t} + u_{i,j+1}^{t} + u_{i,j-1}^{t})$$

2 The CPU Bottleneck

In a standard 2000×2000 grid, the CPU must iterate through 4 million cells. For every time step, the CPU fetches data from RAM, performs the addition, and writes it back. Because the CPU has few cores, it must process these cells in small batches or one by one. This serial dependency creates a massive information bottleneck.

3 The Implementation: Python (NumPy vs. CuPy)

In Python, we observe a dramatic shift by moving from the CPU based NumPy to the CUDA powered CuPy.

```
import numpy as np
import cupy as cp
import time

# Simulation Parameters
size, steps = 2000, 500
plate = np.zeros((size, size))
plate[0, :] = 100.0  # Hot edge

# CPU Execution
start = time.time()
for _ in range(steps):
    plate[1:-1, 1:-1] = 0.25 * (plate[2:, 1:-1] + plate[:-2, 1:-1] +
                                plate[1:-1, 2:] + plate[1:-1, :-2])
print(f"CPU Total Time: {time.time() - start:.4f}s")

# GPU Execution (Moving data to VRAM)
plate_gpu = cp.asarray(plate)
start = time.time()
for _ in range(steps):
    plate_gpu[1:-1, 1:-1] = 0.25 * (plate_gpu[2:, 1:-1] + plate_gpu[:-2, 1:-1]
        +
                                    plate_gpu[1:-1, 2:] + plate_gpu[1:-1,
                                       :-2])
cp.cuda.Stream.null.synchronize()
print(f"GPU Total Time: {time.time() - start:.4f}s")
```

4 The Implementation: MATLAB (Parallel Toolbox)

MATLAB handles this by casting the array as a `gpuArray`.

```
T = zeros(2000, 2000);
T(1, :) = 100;

% CPU Loop
tic;
for k = 1:500
    T(2:end-1, 2:end-1) = 0.25 * (T(3:end, 2:end-1) + T(1:end-2, 2:end-1) +
        ...
                                  T(2:end-1, 3:end) + T(2:end-1, 1:end-2));
end
toc;

% GPU Loop
Tg = gpuArray(T);
tic;
for k = 1:500
    Tg(2:end-1, 2:end-1) = 0.25 * (Tg(3:end, 2:end-1) + Tg(1:end-2, 2:end-1) +
         ...
                                   Tg(2:end-1, 3:end) + Tg(2:end-1, 1:end-2));
end
wait(gpuDevice);
toc;
```

5 The Implementation: R (gpuR)

R's functional nature allows for elegant GPU offloading using the `vapply` or matrix arithmetic on `gpuMatrix` objects.

```
library(gpuR)
T_cpu <- matrix(0, 2000, 2000)
T_cpu[1,] <- 100

# GPU Allocation
T_gpu <- gpuMatrix(T_cpu, type="double")

system.time({
  for(i in 1:500){
    # Internal kernel mapping
    T_gpu[2:1999, 2:1999] <- 0.25 * (T_gpu[3:2000, 2:1999] + ...)
  }
})
```

> **Insight: Memory Locality vs. Calculation**
> The GPU version is not just faster because it has more cores. It is faster because the grid stays in the high bandwidth VRAM for all 500 steps. In the CPU version, data often travels between the CPU Cache and the slower System RAM every single step.

6 Conclusion

By treating the 2D grid as a single manifold of information rather than 4 million individual points, the GPU reduces a problem that takes seconds into one that takes milliseconds.