# Chapter 4: GPU programming



## Version 1: Better memory access pattern

The main reason for the poor performance of the previous version is a poor memory access pattern. To fix it, we need to understand a little bit of the GPU hardware.

### What does not work

Recall that the kernel looked like this:

```
__global__ void mykernel(float* r, const float* d, int n) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    int j = threadIdx.y + blockIdx.y * blockDim.y;
    if (i >= n || j >= n)
        return;
    float v = HUGE_VALF;
    for (int k = 0; k < n; ++k) {
        float x = d[n*i + k];
        float y = d[n*k + j];
        float z = x + y;
        v = min(v, z);
    }
    r[n*i + j] = v;
}
```

Based on what we discussed about memory access patterns on the CPU, we might expect that the right solution would be to transpose the matrix and try out something like this, in order to make sure that each thread is reading memory in a linear fashion:

```
__global__ void mykernel(float* r, const float* d, const float* t, int n) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    int j = threadIdx.y + blockIdx.y * blockDim.y;
    if (i >= n || j >= n)
        return;
    float v = HUGE_VALF;
    for (int k = 0; k < n; ++k) {
        float x = d[n*i + k];
        float y = t[n*j + k];
        float z = x + y;
        v = min(v, z);
    }
    r[n*i + j] = v;
}
```

This does not work! The performance of the new version is as poor as the original version!

### Problem

Everything in the GPU happens in warps of 32 threads. In our code we have blocks of dimensions 16 × 16, so they get divided in 8 warps with 32 threads in each warp. For example, the first warp corresponds to threads with these (x, y) indexes:

```
(0,0), (1,0), ... (15,0),
(0,1), (1,1), ... (15,1).
```

All threads of a warp operate in a fully synchronized fashion. If one thread is reading `d[n*i + k]`, all threads of the warp are reading it simultaneously.

Let us look at the line `float x = d[...]`. To have nice round numbers, let us assume that n = 1000, and let us focus on the first warp of the first block; all other warps are similar. To set the value of `x`, the 32 threads in the warp will try to read the following array elements simultaneously:

```
d[0], d[1000], ..., d[15000],
d[0], d[1000], ..., d[15000].
```

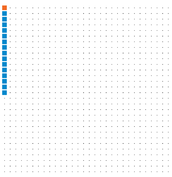
This is bad from the perspective of the memory controller. The memory reads are likely to span 16 different cache lines, and we are only using 1 element per cache line.

Incidentally, the memory access pattern is much better for the line `float y = d[...]`:

```
d[0], d[0], ..., d[0],
d[1], d[1], ..., d[1].
```

We are just accessing 2 different elements, both likely to be on the same cache line, so the memory system will need to do much less work here. The hardware is clever enough to realize that we are reading the same element many times.

Note that the solution with a transpose does not help with the problematic memory access pattern at all.



The above animation illustrates what happens in the first block of threads, for the first 6 iterations of the loop:

- Orange: which elements are accessed by the first thread.
- Blue: which elements are accessed by the first warp of threads.
- Black: which elements are accessed by the first block of threads (assuming that they are executing in a synchronized manner, which is not necessarily the case).

The key observation is that the blue dots often span very many rows, which is bad.

### Solution

There is a very simple solution: just swap the role of the indexes `i` and `j`:

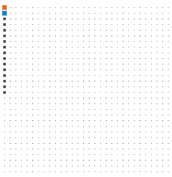
```
__global__ void mykernel(float* r, const float* d, int n) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    int j = threadIdx.y + blockIdx.y * blockDim.y;
    if (i >= n || j >= n)
        return;
    float v = HUGE_VALF;
    for (int k = 0; k < n; ++k) {
        float x = d[n*j + k];
        float y = d[n*k + i];
        float z = x + y;
        v = min(v, z);
    }
    r[n*j + i] = v;
}
```

Now on the line `float x = d[...]` the memory addresses that the warp accesses are good; we are accessing only two distinct elements:

```
d[0],    d[0],    ..., d[0],
d[1000], d[1000], ..., d[1000].
```

And on the line `float y = d[...]` the memory addresses that the warp accesses are also good; we are accessing one continuous part of the memory:

```
d[0], d[1], ..., d[15],
d[0], d[1], ..., d[15].
```



The above animation illustrates what happens in the first block of threads, for the first 6 iterations of the loop:

- Orange: which elements are accessed by the first thread.
- Blue: which elements are accessed by the first warp of threads.
- Black: which elements are accessed by the first block of threads (assuming that they are executing in a synchronized manner, which is not necessarily the case).

The key observation is that the blue dots are either continuous memory addresses or they only span two rows, which is good.

### Results

This simple change was enough to improve the running time from 42 seconds to 8 seconds.

We are now doing 52.5 operations per clock cycle.