Chapter 4: GPU programming

Version 0: Baseline

Recall that the baseline CPU solution for the shortcut problem looked like this:

```
void step(float* r, const float* d, int n) {
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            float v = std::numeric_limits<float>::infinity();
            for (int k = 0; k < n; ++k) {
                float x = d[n*i + k];
                float y = d[n*k + j];
                float z = x + y;
                v = std::min(v, z);
            }
            r[n*i + j] = v;
        }
    }
}
```

We will now turn this into a CUDA program.

Threads and blocks

There are 3 nested loops and in total we do n × n × n units of work. We will need to somehow split this in blocks and threads. We make the following choices:

- Each thread does 1 × 1 × n units of work: it just calculates one unit of the result.
- There are 16 × 16 threads per block. Hence, each block does 16 × 16 × n units of work overall.
- Therefore we will need approximately (n/16) × (n/16) blocks to do all the work.

GPU side

Our kernel looks near-identical to the innermost loop of the CPU solution:

```
__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;
}
```

The only new ingredient is the `return` statement that is needed if `n` is not a multiple of 16.

Here `blockDim` is something we have not introduced yet, but it simply indicates the size of a block; in our case we will always have `blockDim.x = blockDim.y = 16`.

CPU side

The kernel cannot directly access the main memory of the CPU; it can only access the memory of the GPU. Hence, we will need to go through the following steps:

- Allocate some GPU memory for the input and output.
- Copy the input from the CPU memory to the GPU memory.
- Launch the kernel.
- Copy the result back from the GPU memory to the CPU memory.
- Release the GPU memory that we allocated.

Using the CUDA memory management functions, the end result looks like this:

```
void step(float* r, const float* d, int n) {
    // Allocate memory & copy data to GPU
    float* dGPU = NULL;
    CHECK(cudaMalloc((void**)&dGPU, n * n * sizeof(float)));
    float* rGPU = NULL;
    CHECK(cudaMalloc((void**)&rGPU, n * n * sizeof(float)));
    CHECK(cudaMemcpy(dGPU, d, n * n * sizeof(float), cudaMemcpyHostToDevice));

    // Run kernel
    dim3 dimBlock(16, 16);
    dim3 dimGrid(divup(n, dimBlock.x), divup(n, dimBlock.y));
    mykernel<<<dimGrid, dimBlock>>>(rGPU, dGPU, n);
    CHECK(cudaGetLastError());

    // Copy data back to CPU & release memory
    CHECK(cudaMemcpy(r, rGPU, n * n * sizeof(float), cudaMemcpyDeviceToHost));
    CHECK(cudaFree(dGPU));
    CHECK(cudaFree(rGPU));
}
```

Note that the `cudaMemcpy` function will automatically wait for the kernel to finish, so no additional synchronization is needed here. For error checking we use the following utilities:

```
#include <cstdlib>
#include <iostream>
#include <cuda_runtime.h>

static inline void check(cudaError_t err, const char* context) {
    if (err != cudaSuccess) {
        std::cerr << "CUDA error: " << context << ": "
            << cudaGetErrorString(err) << std::endl;
        std::exit(EXIT_FAILURE);
    }
}

#define CHECK(x) check(x, #x)
```

The "divide and round up" function `divup` that we will need often is defined as follows; we also define another function for rounding up that we will need soon:

```
static inline int divup(int a, int b) {
    return (a + b - 1)/b;
}

static inline int roundup(int a, int b) {
    return divup(a, b) * b;
}
```

Results

The program works just fine, all heavy lifting is now done by the GPU, and the CPU is mostly just idle waiting for the GPU to finish its work. However, the running time is not that impressive yet: the program takes 42 seconds to complete for n = 6300, which is something we could easily achieve with a single-core CPU.

We are doing only 12 billion operations per second or 10.7 operations per clock cycle. For a device with 640 parallel units this is a very low number. We can do much better…