Chapter 4: GPU programming

Version 3: Reuse data in shared memory

So far we have not really made any direct use of the concept of a "block". We have merely made sure that when the GPU splits a block in warps, the threads of each warp are accessing nice continuous parts of the memory. Now we will start to look at collaboration and coordination between threads in a block.

Shared memory

The GPU has a very small amount of very fast "shared memory". For example, in the GPUs that we are using, the total amount of shared memory is 5 × 64 KB.

Each block of threads can allocate some part of the shared memory. It is important to keep in mind that there is not that much shared memory available overall, and we would like to have lots of blocks available for execution simultaneously in parallel, so the amount of shared memory that we allocate per block can only be a few kilobytes, preferably less than that.

We can allocate shared memory for e.g. 100 floats by using the following line in the kernel code:

```
__shared__ float x[100];
```

If we have this declaration in the kernel code, the GPU will allocate 400 bytes of shared memory for each block that is currently being executed. (Once we run out of shared memory, additional blocks will have to wait for their turn.)

Note that this is a static declaration; the compiler will hard-code the information "this kernel will need 400 bytes of shared memory" at compilation time, so the GPU already knows about it when we launch the kernel.

When we refer to e.g. `x[0]` in the kernel code, all threads of the same block refer to the same piece of memory. We can use this to share information between threads, but as usual in multithreaded code, we will need to take care of proper synchronization. To achieve this we will use the following primitive that acts as a barrier:

```
__syncthreads();
```

Different threads in the same block may proceed at different speeds, but `__syncthreads()` will wait until all threads are synchronized: no thread will execute any part of code that come after this instruction until all threads have completed all parts of code that come before this instruction. A very typical pattern of code looks like this:

```
int i = threadIdx.x;

__syncthreads();

// Now thread i can safely modify x[i],
// as no other thread will touch the same element

x[i] = ...;

__syncthreads();

// Now all threads can safely read any element,
// as no thread is writing anything

float y = x[something];
float z = x[something];
...

__syncthreads();
```

(Note that the threads of a warp are always synchronized, but a block consists of multiple warps, and synchronization is therefore necessary.)

Shared memory as a cache

We will now use shared memory explicitly as a small cache memory that we control. The overall idea on the level of a block is this — we are trying to reuse each input element 64 times by keeping it in the shared memory:

- Each block computes 64 × 64 values of the output.
- Each block repeats these steps n times:
  - Read 64 × 64 values of the input to shared memory.
  - Do 64 × 64 calculations with these values.

We must make sure all threads in each block participate not only in calculations but also in the process of copying data to shared memory; moreover, we still need to worry about the memory access pattern. On a more fine-grained level, we proceed as follows:

- Each block consists of 8 × 8 threads.
- Each thread computes 8 × 8 values of the output.
- Each thread repeats these steps n times:
  - Synchronize.
  - Read 1 + 1 values of the input to shared memory.
  - Synchronize.
  - Read 8 + 8 values of the input from shared memory to registers.
  - Do 8 × 8 calculations with these values.

Note the convenient numbers that we use: we happen to have 8 × 8 = 64 threads, and we are trying to read 64 + 64 values from input to shared memory, so it is easy to split this work evenly among the threads.

This idea almost works. However, from the perspective of a thread, we would have two synchronization points per two memory reads, which is a relatively high number. We slightly modify the scheme so that instead of reading just one floating point number from each row or column to shared memory, we read 4 elements. This decreases the number of synchronization operations by a factor of 4 (which is good), but increases shared memory consumption by a factor of 4 (which is bad, but not too bad).

Memory access pattern

We are aiming at the following memory access pattern.

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).
- Top: array `d`.
- Bottom: array `t`.

The key observation is that the blue dots always span continuous regions, which is good. Moreover, we are maximizing the amount of useful work done per warp: all threads of a warp access different memory addresses.

Implementation

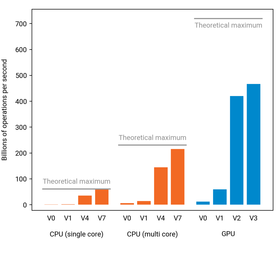
In comparison with version 2, we only need to modify the kernel code to implement these ideas:

```
__global__ void mykernel(float* r, const float* d, int n, int nn) {
    int ia = threadIdx.x;
    int ja = threadIdx.y;
    int ic = blockIdx.x;
    int jc = blockIdx.y;

    const float* t = d + nn * nn;

    __shared__ float xx[4][64];
    __shared__ float yy[4][64];

    float v[8][8];
    for (int ib = 0; ib < 8; ++ib) {
        for (int jb = 0; jb < 8; ++jb) {
            v[ib][jb] = HUGE_VALF;
        }
    }
    for (int ks = 0; ks < n; ks += 4) {
        int ija = ja * 8 + ia;
        int i = ic * 64 + ija;
        int j = jc * 64 + ija;
        for (int f = 0; f < 4; ++f) {
            int k = ks + f;
            xx[f][ija] = t[nn*k + i];
            yy[f][ija] = d[nn*k + j];
        }

        __syncthreads();

        #pragma unroll
        for (int f = 0; f < 4; ++f) {
            float y[8];
            for (int jb = 0; jb < 8; ++jb) {
                y[jb] = yy[f][jb * 8 + ja];
            }
            for (int ib = 0; ib < 8; ++ib) {
                float x = xx[f][ib * 8 + ia];
                for (int jb = 0; jb < 8; ++jb) {
                    v[ib][jb] = min(v[ib][jb], x + y[jb]);
                }
            }
        }

        __syncthreads();
    }
    for (int ib = 0; ib < 8; ++ib) {
        for (int jb = 0; jb < 8; ++jb) {
            int i = ic * 64 + ib * 8 + ia;
            int j = jc * 64 + jb * 8 + ja;
            if (i < n && j < n) {
                r[nn*i + j] = v[ib][jb];
            }
        }
    }
}
```

The CUDA compiler is usually happy to unroll most loops automatically, but looking at the SASS assembly code with `cuobjdump -sass` we noticed that one of the loops was not unrolled. An explicit `#pragma unroll` helped with the performance here.

Results

With the help of shared memory we improved the running time slightly, from 1.2 seconds to 1.1 seconds. We are now using 65 % of the theoretical maximum performance of the GPU, and we are doing much better than any of the CPU solutions on the same computer.



Are we happy now?

How much better could we still do if we tried to improve e.g. data reuse in the kernel code?

Let us conduct a small experiment. First, we replace all "min" operations with additions to make sure that we do not have arithmetic bottlenecks related to them. Then we run the code again under `nvprof` for 10 iterations, and look at the time it takes to run `mykernel`. The total time spend in the kernel turns out to be 8.13 seconds, which translates to 615 billion operations per second. This is more than 85 % of the theoretical maximum performance of the GPU, and hence there seems to be fairly little room for performance improvement related to the way we read data from memory in the kernel.

Preprocessing and data copying are responsible for less than 10 % of the total running time, so big gains are not to be found there, either.

However, it turns out that there might be some room for improvements related to the arithmetic operations: on the GPU that we use, perfectly interleaved minimum and addition operations seem to achieve the same performance as a sequence of addition operations, but in this application replacing minimum operations with additions improves the kernel running time by more than 15 %. Maybe we could interleave the addition and minimum operations better if we wrote SASS code directly?

We stop now, but the reader is encouraged to keep exploring!