

Semestral Project announcement

Jiří Filipovič

Fall 2023

B-spline coefficients computation

The goal of the project is to implement an GPU-accelerated version of a code computing b-spline coefficient. We are not going to dig into mathematic too deeply, but convert C code to CUDA.

- the input is a 2D array, which is processed by rows and columns
- for each row:
 - we compute a 'sum' of the row (with some more math)
 - the sum is used to modify the first element of the row
 - we iteratively modify the row from left to right, each element uses the value of its predecessor
 - we modify the value of the rightmost element
 - we iteratively modify the row from right to left, each element uses the value of its predecessor
- the same for columns

```
void solveCPU(float *in, float *out, int x, int y) {
    const float gain = 6.0f;
    const float z = sqrtf(3.f) - 2.f;

    // process lines
    for (int line = 0; line < y; line++) {
        float* myLine = out + (line * x);

        // copy input data
        for (int i = 0; i < x; i++)
            myLine[i] = in[i + (line * x)] * gain;

        // compute 'sum'
        float sum = (myLine[0] + powf(z, x)
            * myLine[x - 1]) * (1.f + z) / z;
        float z1 = z;
        float z2 = powf(z, 2 * x - 2);
        float iz = 1.f / z;
        for (int j = 1; j < (x - 1); ++j) {
            sum += (z2 + z1) * myLine[j];
            z1 *= z;
            z2 *= iz;
        }

        // iterate back and forth
        myLine[0] = sum * z / (1.f - powf(z, 2 * x));
        for (int j = 1; j < x; ++j)
            myLine[j] += z * myLine[j - 1];
        myLine[x - 1] *= z / (z - 1.f);
        for (int j = x - 2; 0 <= j; --j)
            myLine[j] = z * (myLine[j + 1] - myLine[j]);
    }
    // process columns...
```

Implementation

You will get a framework, which does all the boring stuff:

- creates input, copies it into GPU memory
- check result of CUDA implementation against non-optimized CPU code
- benchmarks your code

Your work

- you are expected to write CUDA code (kernel and code calling the kernel in file `kernel.cu`)
- you can get inspiration (and precise specification) from unoptimized code in `kernel_CPU.C`
- compilation: `nvcc -o framework framework.cu`

Project Rules

What will be tested?

- the input size predefined in `framework.cu` can be changed (can be rectangular)
- the size will be at least 1024 in each dimension
- the size will be divisible by 128 in each dimension
- the code should run on computing capability 3.0 and newer
- your kernel can overwrite the input
- the result of GPU and CPU code has to differ at most by factor of 0.0001
- tip: you can consider limited arithmetic precision of floats and spare some operations without effect

What is forbidden?

- collaboration (discuss general questions, not your code)

Project Stages

The project has three stages:

- running parallel implementation (till Nov 5th): 25p
- efficient implementation (till Nov 26th, required performance will be announced): 25p
- the final competition (till Dec 6th): up to 20p for above average implementations

The First Stage

Write a correct implementation in C for CUDA.

- the performance is not relevant
- but must be efficiently parallelized
- till November 5th (any day time)
- the points will be assigned according to the functionality of your code (check different input sizes!), if delayed, -2 points for each day of delay
- I highly recommend to start with optimization immediately after you have a functional code

Why are you doing this?

Not an 'artificial' problem

- I will reveal what it is for at the end of the semester
- besides learning, you are also solving a practical problem, which can be used in real software :-)