Introduction to CUDA and OpenCL

Łukasz Gut, Grzegorz Litarowicz

Lab 06, 16.04.2020

Introduction

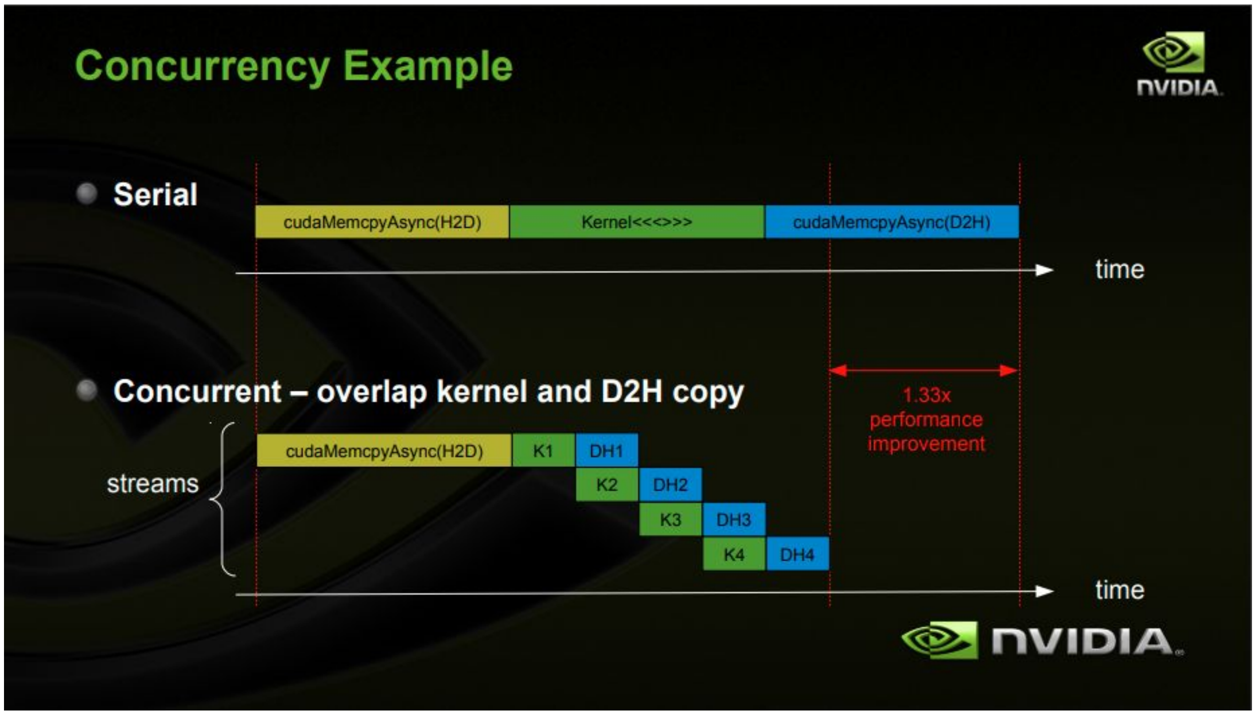
In this report, we first take a look at CUDA Streams and analyze them. We also try to improve the performance of SAXPY operation on the GPU. Lastly, we take a look at the single-threaded implementation of 2D heat conduction equation and parallelize it using CUDA to improve performance.

Streams

A stream in CUDA language is a way to define a sequence of commands that execute in order. Different streams may execute their commands concurrently or out of order with respect to each other.

This is important because it lets us further parallelize our code to improve performance even more. It is especially useful when dealing with multiple kernel launches. In such a scenario streams let us perform data movement and kernel launches concurrently.

The idea can be easily seen on the slide below which was provided by Nvidia in a presentation on Streams and Concurrency.



SAXPY

SAXPY stands for "Single-Precision A*X Plus Y". This is a very common vector operation which includes vector scaling and vector addition. Mathematically this can be written as follows:

$$y = ax + y$$

This simple operation can be implemented using different techniques. Our goal was to make SAXPY calculations of vectors of size 2048×2048 take less time than 25 ms.

We tried a few different techniques to make this happen:

- Unified Memory
- Memory prefetching
- Data initialization on GPU instead of CPU
- Stride technique inside a kernel
- Loop unrolling

Unfortunately, we did not manage to speed up the program running on the GPU to take less time than 25 ms. However, we found that our program runs about 38% faster on GTX 780Ti than on RTX 2060. This is very interesting.

Also, we did not completely fail. We actually managed to make this program run within 25ms time threshold, though it was on the CPU side.

Vector size	GTX 780ti time [ms]	RTX 2060 time [ms]	CPU time [ms]
2048x2048	54.524	87.622	13.758

2D Heat Conduction

The heat conduction is a well-known problem that has lots of solutions. In this report, we solve the 2D version of the beforementioned problem. Mathematically it can be described as follows:

$$\frac{dT}{dt} = a\left(\frac{d^2T}{dx^2} + \frac{d^2T}{dy^2}\right)$$

Where T is the temperature, t is time, α is the thermal diffusivity and both x and y are variables.

After discretization the problem can be written as follows:

$$\frac{\Delta T}{\Delta t} = a\left(\frac{T_{i+1,j} - 2T_{i,j} + T_{i-1,j}}{\Delta x^2} + \frac{T_{i,j+1} - 2T_{i,j} + T_{i,j-1}}{\Delta y^2}\right)$$

To solve it, a simple Euler method will be used.

Let's see how this can be implemented using the CUDA framework. First the function that spawns kernel:

```
void step_kernel_mod(float* temp_in, float* temp_out)
{
	constexpr int threads_per_block = 32;

	const dim3 block_size(threads_per_block, threads_per_block);
	const int block_x = (ni + threads_per_block - 1)/threads_per_block;
	const int block_y = (nj + threads_per_block - 1)/threads_per_block;
	const dim3 grid_size = dim3(block_x, block_y);

	Calculate_temp <<<grid_size, block_size>>> (temp_in, temp_out);
	cudaDeviceSynchronize();
}
```

This function defines a 2D grid and spawns kernel. We chose a 2D grid in order to make memory access very easy and natural, which let us avoid "for loop" in the kernel. After calling kernel we have to Synchronize the device because our next step depends on this one results.

Next lets take a look at the kernel code:

```
__global__
void calculate_temp(const float* temp_in, float* temp_out) {
	const int i = blockIdx.x*blockDim.x + threadIdx.x;
	const int j = blockIdx.y*blockDim.y + threadIdx.y;

	if(i < ni-1 && j < nj-1 && i > 0 && j > 0) {
		const int i00 = I2D(ni, i, j);
		const int im10 = I2D(ni, i-1, j);
		const int ip10 = I2D(ni, i+1, j);
		const int i0m1 = I2D(ni, i, j-1);
		const int i0p1 = I2D(ni, i, j+1);

		const float d2tdx2 = temp_in[im10]-2*temp_in[i00]+temp_in[ip10];
		const float d2tdy2 = temp_in[i0m1]-2*temp_in[i00]+temp_in[i0p1];

		temp_out[i00] = temp_in[i00]+tfac*(d2tdx2 + d2tdy2);
	}
}
```

This code is basicly the same as from one iteration of for loop from one thread implementation. We added if condition to ensure we don't read values that are out of range.

The table below shows the results of a simple benchmark we did in order to validate our approach.

Grid size	CPU time [s]	GPU time [s]
200 x 100	0.03496	0.013997
1000x1000	1.811374	0.069304

Conclusion

Streams are another way of optimizing our code on GPU. This tool lets us solve multiple smaller tasks that require only small grids in a more efficient way. If we would run them synchronously then a lot of resources on GPU would be wasted. Thanks to streams we can run more than one of these problems asynchronously and use all of the resources that are available on our machines.

Although SAXPY operation is easily parallelizable it is really hard to get better performance on the GPU than on the CPU. This might be due to the fact, that the CPU on the server is quite powerful and we've seen this kind of behaviour on previous labs. Highly parallel computing starts to shine when the data access pattern is easily mappable to the grid layout, but on the other hand, the CPU has a hard time of fitting all the data it needs in its caches.

GPU lets us solve 2D Heat conduction very efficiently because this problem involves a lot of elements even for a small dimension of matrix and requires simple operations on them. As we see calculations on GPU are 26 times faster than the CPU.