Università degli Studi di Firenze
Parallel Computing

Parallelization of Kernel Processing Algorithm with C++11 and CUDA

Pietro Bongini

Anno Accademico 2017/2018

- Implementation of a parallel version of kernel convolution with C++11 and CUDA
- Performance analysis

Introduction to Kernel Processing

- Kernel Processing Algorithm is a widely used technique in the field of image processing.
- It consists in the convolution of two matrixes:
  - the image matrix
  - and a matrix called kernel or mask
- Usually the mask is a 3×3 or 5×5 or 7×7 matrix

Kernel Processing Algorithm

Kernel processing algorithm consists of the following steps

1. We set the accumulator to zero.
2. We cycle on rows and columns of kernel elements.
3. We multiply each kernel element with the corresponding element of the image and we add the result to the accumulator.
4. At the end of the cycle we assign the accumulator value to the output image pixel which corresponds to the central element of the kernel.
5. We repeats 1) 2) 3) 4) for all the pixels of the input image

C++ Kernel Processing algorithm

The way we handle the edge pixels is the following:

- We don't take into account the pixels of the image that require pixels out of the image bound

As we can see the Kernel Processing Algorithm becomes very onerous

- increasing the dimensions of the input image
- increasing the dimensions of the kernel

We have parallelized the algorithm with C++11 and CUDA

C++11 introduction

- C++11 is a version of the standard C++ which provides both low and high level facilities for multithread programming.
- It's very simple to create a thread and passing to it the arguments.
- As we can see Kernel Convolution is an embarrassingly parallel algorithm so we don't need mutex

C++11 implementations

We have create multiple parallel implementations of Kernel Processing with C++11

- parallel version on image channels
- parallel version on image rows
- parallel version on image columns
- parallel version on image blocks

```
void parallelizedRowConvolution(Image_t*inputImage,const float * kernel ,float * outputImageData,
                        int kernelSizeX, int kernelSizeY, int dataSizeX, int dataSizeY, int channels)
    static const int numThreads = 8;    //number of threads
    std::thread t[numThreads];

    for(int i = 0; i< numThreads; i++) {    //assign arguments to each thread
        t[i] = thread(rowConvolution, i * (dataSizeY / numThreads), (dataSizeY / numThreads) * (i + 1),
                      inputImage, kernel, outputImageData, kernelSizeX, kernelSizeY,
                      dataSizeX, dataSizeY, channels, numThreads, i);
    }

    for (int j = 0; j < numThreads; ++j) {  //wait for all threads
        t[j].join();
    }
}
```

Experiments

We have conducted different experiments measuring the execution time of the sequential version and the parallel version in multiple cases:

- Increasing the dimensions of the input image
- Increasing the dimensions of the kernel

The results are shown below

image dim.	seq.	chan.	block.	row	cols
508 x 493	0.06s	0.04s	0.02s	0.02s	0.02s
1920 x 1200	0.63s	0,24s	0.11s	0.11s	0.011s
3939 x 2955	2.61s	1.21s	0.36s	0.33s	0.35s
8160 x 6120	9.66s	4.36s	1.30s	1.25s	1.23s

Table: Execution times of sequential and C++11 versions using a 3x3 kernel

Experiment results

image dim.	seq.	chan.	block.	row	cols
508 x 493	0.19s	0.09s	0.045s	0.046s	0.041s
1920 x 1200	1.40s	0,62s	0.25s	0.21s	0.21s
3939 x 2955	6.44s	2.60s	0.79s	0.85s	0.83s
8160 x 6120	24.90s	10.46s	2.92s	3.05s	3.12s

Table: Execution times of sequential and C++11 versions using a 5x5 kernel

image dim.	seq.	chan.	block.	row	cols
508 x 493	0.46s	0.17s	0.08s	0.07s	0.07s
1920 x 1200	2.36s	1.08s	0.36s	0.35s	0.37s
3939 x 2955	11.45s	4.80s	1.50s	1.59s	1.44s
8160 x 6120	47.66s	19.25s	5.58s	5.76s	5.89s

Table: Execution times of sequential and C++11 versions using a 7x7 kernel

We also represent the the results in the tables on graphs.

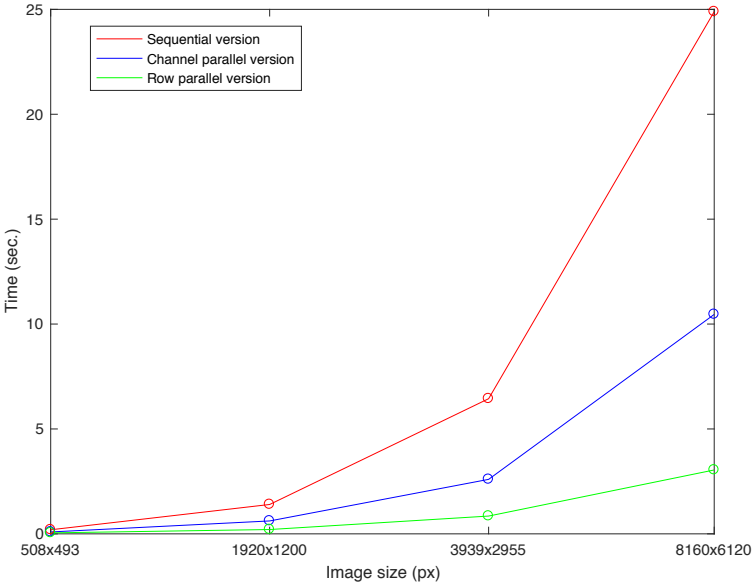

Figure: trend of the execution time (increasing the dimension of the image) of parallel per channel version, parallel per row version and sequential version (using 5×5 kernel)

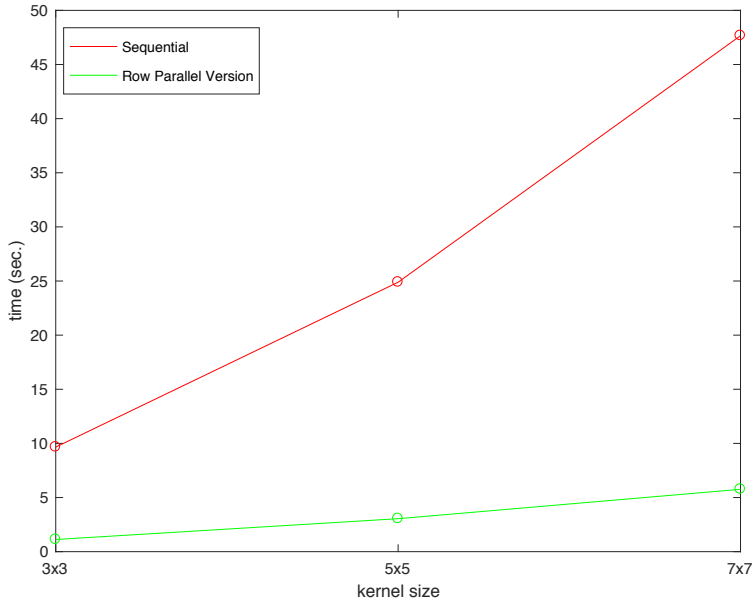

Figure: trend of execution time (increasing the size of the kernel) of sequential version and per row version (using 5×5 kernel)

Experiment results

In the following table are shown the speedups for the different C++11 implementations

kernel size	chan.	block.	row	cols
3×3	2.21	7.43	7.73	7.85
5×5	2.38	8.52	8.16	7.98
7×7	2.47	8.54	8.27	8.09

Table: Speedup results for C++11 versions

CUDA introduction

- CUDA is a parallel computing platform and application programming interface (API) model created by Nvidia.
- This platform is designed to work with C, C++,Fortran and Python.
- It enables explicit GPU memory management. The GPU runs multiple threads in parallel.
- A CUDA program is made of two parts: the host code and the device code; the host code runs on the CPU and the device code on the GPU.
- In the GPU, threads are organized in block and blocks constitute a grid.

CUDA algorithm

In the implementation of Kernel Processing with CUDA we follow these steps:

1. We allocate GPU memories.
2. We copy image matrix, mask matrix and output matrix from CPU memory to GPU memory.
3. We execute the CUDA function to compute kernel convolution.
4. We copy data back from GPU to CPU.
5. We destroy GPU memories.

CUDA algorithm code

We have developed 4 different versions of kernel processing using CUDA using

- Global Memory
- Global Memory and kernel in constant memory
- Shared Memory
- Shared Memory and kernel in constant memory

Experiments

- The experiments conducted are similar to the experiments conducted for C++11.
- We analyze the execution time of CUDA versions in comparison with the sequential version.

The results are shown below

image dim.	global	glob+const	shared	shared+const
508 x 493	0.027ms	0.026ms	0.024ms	0.023ms
1920 x 1200	0.027ms	0.027ms	0.025ms	0.024ms
3939 x 2955	0.027ms	0.027ms	0.025ms	0.025ms
8160 x 6120	0.029ms	0.028ms	0.027ms	0.026ms

Table: Execution times of CUDA versions using 3x3 kernel

Experiment results

image dim.	global	glob+const	shared	shared+const
508 x 493	0.026ms	0.026ms	0.025ms	0.025ms
1920 x 1200	0.028ms	0.027ms	0.025ms	0.025ms
3939 x 2955	0.029ms	0.028ms	0.027ms	0.026ms
8160 x 6120	0.030ms	0.028ms	0.027ms	0.026ms

Table: Execution times of CUDA versions using 5x5 kernel

image dim.	global	glob+const	shared	shared+const
508 x 493	0.029ms	0.029ms	0.026ms	0.025ms
1920 x 1200	0.030ms	0.030ms	0.029ms	0.028ms
3939 x 2955	0.031ms	0.031ms	0.030ms	0.030ms
8160 x 6120	0.031ms	0.030ms	0.029ms	0.028ms

Table: Execution times of CUDA versions using 7x7 kernel

Experiment results

- The results of CUDA versions are much faster than the other versions
- They are obtained computing a mean from multiple experiments
- There isn't a significant difference on the execution time of the CUDA versions