Image Blurring

Cynthia Berenice Castillo Millán

September 2018

1. Introduction

OpenCV, Open Source Computer Vision, is a library of programming functions aimed for real-time computer vision, designed for computational efficiency and multi-core processing. For the purpose of this project, we're going to take advantage of these characteristics to develop three different implementations of a 5 x 5 image blurring, evaluating its performance exploiting CPU and GPU capabilities and its speedup with different threads arrangement.

2. Development

For this project, three types of implementations were done: CPU without threads, CPU with openMP threads and using CUDA threads/blocks. Before the actual development of the codes, computer characteristics were checked to guide the testing limitations.

CPU Implementation

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                8
On-line CPU(s) list:   0-7
Thread(s) per core:    2
Core(s) per socket:    4
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 60
Model name:            Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz
Stepping:              3
CPU MHz:               1137.406
CPU max MHz:           3900.0000
CPU min MHz:           800.0000
BogoMIPS:              6800.20
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              8192K
NUMA node0 CPU(s):     0-7
```

Figure 1. Computer characteristics

For our project, the important information are the **threads per core, cores per socket, socket number**, and **CPUs.** This information tell us the maximum thread number: 8 CPU x 4 cores, 2 threads x 1 socket = **64 threads.** Since we are only using one CPU, our thread maximum will be **8.**

The results for our blurring the image regular CPU implementation after 100 iterations:

```
Input image step: 7560 rows: 3600 cols: 2520
Average image blurring elapsed 1336.871704 ms
```

Figure 2. CPU with no threads implementations result.

Since this basic implementation required 4 nested loops (2 for the complete image matrix, 2 for the blurring matrix), we expected a significant improvement by implementing OMP.

The first implementation **collapsed** the outer for loops and **8** threads, our maximum, which resulted in no improvement at all.

```
Input image step: 7560 rows: 3600 cols: 2520
Average image blurring elapsed 1336.984497 ms
```

Figure 3. First OMP implementation result

The second implementation modified, instead, the inner loops and the assignments with the instruction: **#pragma omp parallel for collapse(2) default(shared) reduction (+:out_blue, out_green, out_red).** The result, still, did not show any significant improvement:

```
Average OMP image blurring elapsed 1334.613892 ms
```

Figure 4. Second OMP implementation result

GPU – CUDA implementation

```
Detected 1 CUDA Capable device(s)

Device 0: "GeForce GTX 670"
  CUDA Driver Version / Runtime Version          9.0 / 7.5
  CUDA Capability Major/Minor version number:    3.0
  Total amount of global memory:                 1996 MBytes (2093023232 bytes)
  ( 7) Multiprocessors, (192) CUDA Cores/MP:     1344 CUDA Cores
  GPU Max Clock rate:                            980 MHz (0.98 GHz)
  Memory Clock rate:                             3004 Mhz
  Memory Bus Width:                              256-bit
  L2 Cache Size:                                 524288 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 1 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 1 / 0
  Compute Mode:
     < Exclusive Process (many threads in one process is able to use ::cudaSetDevice() with this device)
>

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 9.0, CUDA Runtime Version = 7.5, NumDevs = 1, De
vice0 = GeForce GTX 670
Result = PASS
```

Figure 5. CUDA characteristics

For our project, the information to take into account is:

```
Maximum number of threads per block:           1024
Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
```

Figure 6. Threads, Blocks, and grids dimensions

The tests were made varying the number of threads per block (without exceeding the 1024 limit) and calculating the enough number of blocks to allocate the image matrix. Warps were took into account, but tests using partial warps were used too for performance testing. Ordered by time:

Block dim X	Block dim Y	Total thread number	Time (ms)
64	2	128	13.54
64	4	256	13.62
32	32	1024	14.34
1024	1	1024	14.85
512	2	1024	14.89
16	16	256	19.71
8	8	64	32.26
4	64	256	54.41
2	64	128	98.89
2	512	1024	101.62
1	1024	1024	191.43

Figure 7. GPU tests results

From the previous chart, the first thing we can notice is that the total number of threads is **not** the most important variable performance wise since results vary largely, but what indeed made a difference was the way in which the block dimensions were setup. My first approach could be

that the incomplete warps messed up with the performance, but apparently, there were other factors going on too.

For instance, the worst results have an opposing party, which didn't performed as bad overall:

Block dim X	Block dim Y	Total thread number	Time (ms)
512	2	1024	**14.89**
2	**512**	1024	**101.62**
1024	1	1024	**14.85**
1	**1024**	1024	**191.43**

Figure 8. Threads dimension performance comparison

Lets now compare the speedup between the implementations:

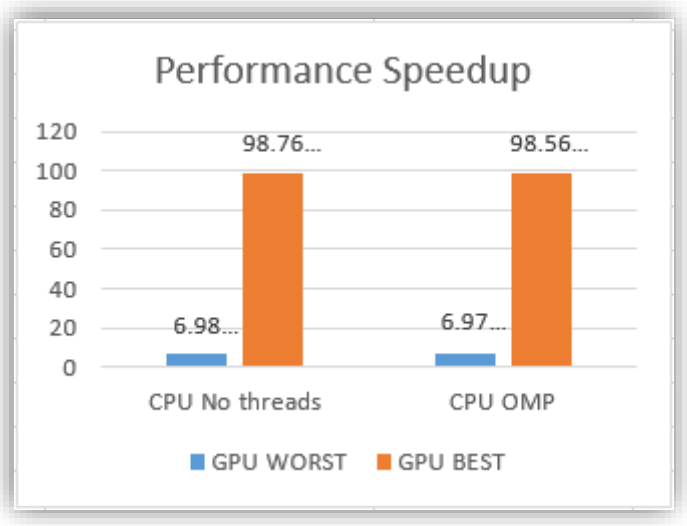

Figure 9. CPU vs GPU performance speedup

When compared to the best GPU timing, the CPU performance improvement was near to the **99%** but when the GPU is not correctly used, it just reached a **7%,** hence the importance of a careful thread and block implementation.

3. Results analysis and conclusions

The main key for a good performance is to correctly balance the work between processors, in a way that hardware utilization is maximized. Occupancy is hence an important metric since it describes the number of active warps on a multiprocessor.

One factor that determines occupancy is the register availability, which keeps local variables nearby for low latency; however, the set of registers is limited. Registers are allocated to an entire block so, if each block uses many registers, the number of registers that can be used in the multiprocessor it is reduced.

Block dim X	Block dim Y	Time (ms)	Total numer of threads
8	8	32.26	64
2	64	98.89	128
64	2	13.54	128
4	64	54.41	256
16	16	19.71	256
64	4	13.62	256
1	1024	191.43	1024
2	512	101.62	1024
32	32	14.34	1024
512	2	14.89	1024
1024	1	14.85	1024

Figure 10. GPU best performance

In this case, the best results are achieved using either 128 or 256 threads, but the block X dimension have a great impact in its performance. I can assume that the number of registers used in my program are impacting the results. Nevertheless, better Occupancy does not equate to better performance. Which may explain why increasing the number of threads doesn't have a major time improvement.

In Nvidia CUDA Documentation, it's said that some experimentation is required in order to select the best block size, but some rules of thumb that should be followed are:

- Threads per block should be a multiple of warp size (to avoid wasting computation).
- A minimum of 64 threads per block should be used.
- Between 128 and 256 threads.

After some experimentation, the results confirmed the rules.

References

[1] Marco Bertini. Parallel Computing, Universitá Degli Studi Firenze. https://www.micc.unifi.it/bertini/download/parallel/2016-2017/11_gpu_cuda_2.pdf

[2] CUDA ToolKit Documentation, Developer Zone. Thread and Block Heuristics https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#optimizing-cuda-applications

[3] Daniele Loiacono. Performance optimization with CUDA. Politecnico Di Milano. http://home.deib.polimi.it/loiacono/uploads/Teaching/CP/CP_08_CUDA_Advanced.pdf