ECE-415 Lab 4 Report

George Koffas gkoffas@uth.gr

Konstantinos Tsivgiouras ktsivgiouras@uth.gr

27 December 2020

This report reviews the various optimizations to our previous implementation of **2D Convolution** in the CPU and the GPU, and the subsequent time measurements from the GPU kernels. The measurements were taken for a variety of scenarios, which are listed below. The experiments were conducted on a Remote Host (csl-artemis) with the following specifications:

- `CPU: Intel(R) Xeon(R) CPU E5-2683 v4 @ 2.10GHz`
  - `L1d Cache: 32 KiB`
  - `L1i Cache: 32 KiB`
  - `L2 Cache: 256 KiB`
  - `L3 Cache: 4 MiB`
  - `Configuration: 2 socket, 16-way multicore, with 2-way SMT`
- `GPU: NVIDIA Tesla K80 @ 824 MHz`
  - `Configuration: 2 chip, 13 SM with 192 CUDA Cores/MP`
  - `Memory Clock: 2505 MHz`
  - `Total Global Memory: 11.411 GB`
- `Compiler: nvcc V11.1.105`

Compilation options for our experiments were:

- `nvcc -Xptxas -v --resource-usage -Xcompiler -O4`

Step 0 - Device Query: We first had to run the `deviceQuery` utility on our target system. The results of the query were used in our further experimentation as useful insight. The query results can be seen below:

```
./deviceQuery Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 2 CUDA Capable device(s)

Device 0: "Tesla K80"
  CUDA Driver Version / Runtime Version          11.1 / 10.1
  CUDA Capability Major/Minor version number:    3.7
  Total amount of global memory:                 11441 MBytes (11996954624 bytes)
  (13) Multiprocessors, (192) CUDA Cores/MP:     2496 CUDA Cores
  GPU Max Clock rate:                            824 MHz (0.82 GHz)
  Memory Clock rate:                             2505 Mhz
  Memory Bus Width:                              384-bit
  L2 Cache Size:                                 1572864 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 2 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Enabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Compute Preemption:            No
  Supports Cooperative Kernel Launch:            No
  Supports MultiDevice Co-op Kernel Launch:      No
  Device PCI Domain ID / Bus ID / location ID:   0 / 6 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

Device 1: "Tesla K80"
  CUDA Driver Version / Runtime Version          11.1 / 10.1
  CUDA Capability Major/Minor version number:    3.7
  Total amount of global memory:                 11441 MBytes (11996954624 bytes)
  (13) Multiprocessors, (192) CUDA Cores/MP:     2496 CUDA Cores
  GPU Max Clock rate:                            824 MHz (0.82 GHz)
  Memory Clock rate:                             2505 Mhz
  Memory Bus Width:                              384-bit
  L2 Cache Size:                                 1572864 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 2 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Enabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Compute Preemption:            No
  Supports Cooperative Kernel Launch:            No
  Supports MultiDevice Co-op Kernel Launch:      No
  Device PCI Domain ID / Bus ID / location ID:   0 / 7 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >
> Peer access from Tesla K80 (GPU0) -> Tesla K80 (GPU1) : Yes
> Peer access from Tesla K80 (GPU1) -> Tesla K80 (GPU0) : Yes

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 11.1, CUDA Runtime Version = 10.1, NumDevs = 2
Result = PASS
```

Step 1 - Optimization The optimizations we introduced to our code are listed below:

- We first made a very obvious optimizations by allocating the **filter** array in *constant memory*. Since it's accessed very frequently, and has a relatively small size compared to the matrix used for the convolution, we thus avoid the overhead of reading the filter's elements from global memory.

- Secondly, we introduced shared memory in our kernels to again reduce overhead from reading the matrix elements from global memory.

For our implementation, the only constraint on the filter radius was

$$filter\_radius = \{1\} \cup \{2k\}, k \in [1, 16], k \in \mathbb{N}$$

Step 2 - Time measurements To evaluate our optimized code, we measured the execution time on the device, both total execution time and memory and kernel operations seperately. The results can be seen in Figures 1 and 2. We can observe that kernel execution time goes up together with the filter radius. This can be attributed to the fact that the amount of padding we add to the original matrix is determined by the filter radius, and thus the size of the shared memory storing tiles of the original matrix is also increased, leading to more reads from global memory to store the appropriate elements to shared memory. We can also observe that the memory transfers times are stable, and the only variable element in these operations is the size of the filter array. However, since we store the filter array in constant memory, the transfer time overhead in this case is insignificant.

Step 3 - Bigger Images: Next, we needed to implement a method to process images of size bigger than 16384×16384. For that reason, we used blocks to store submatrices and then operate on those on the GPU. For experimental measurements we tested our implementation for both 16384×16384 and 32768×32768 matrix sizes. The results can be seen in Figures 3 and 4. For our implementation, block sizes follow the following constraint:

$$block\_size = \{2^k\}, k \in [1, 8], k \in \mathbb{N}$$

Step 4 - Streams: Upon observing the execution times for images bigger than the previously larger size (16384), we had to think of a way to overlay memory transfers and kernel operations in order to decrease the total execution time and better utilize the PCI bus. Thus we used 4 streams to increase

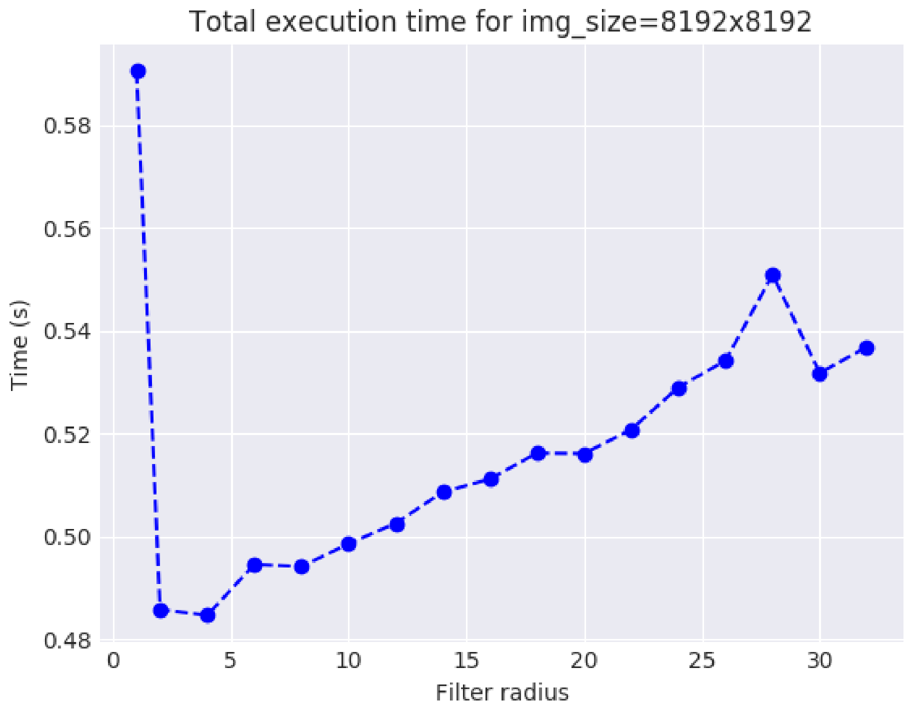


Figure 1: Total device execution time for different filter_radius sizes and 8192×8192 image size.

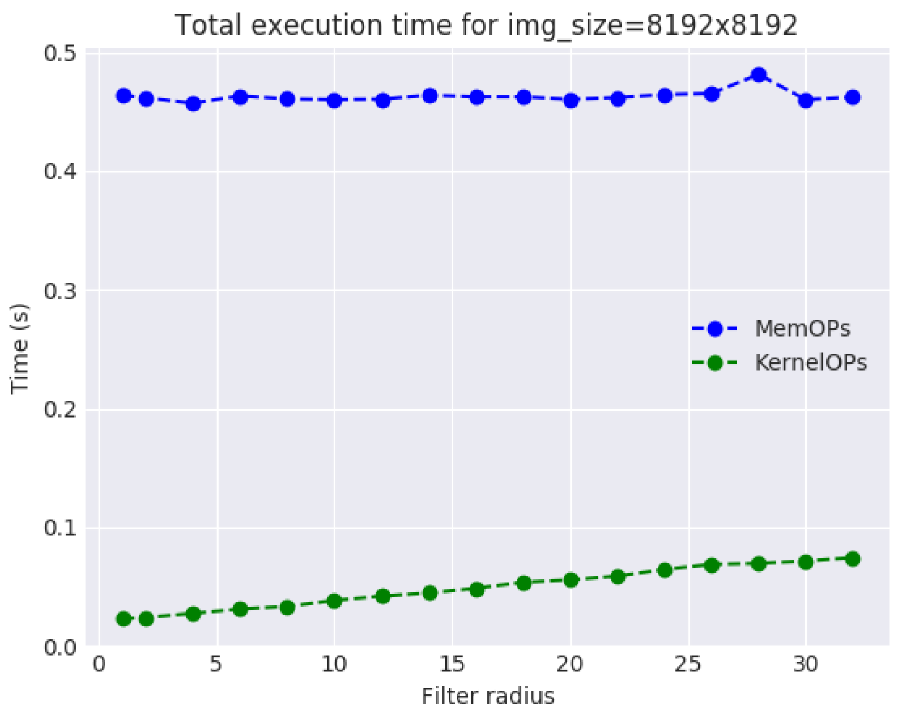
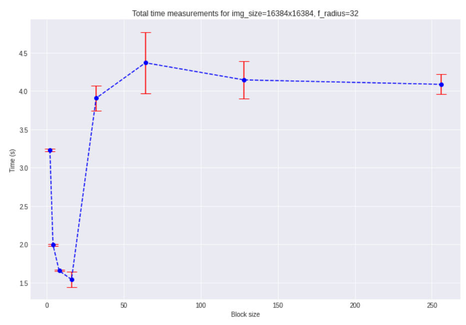
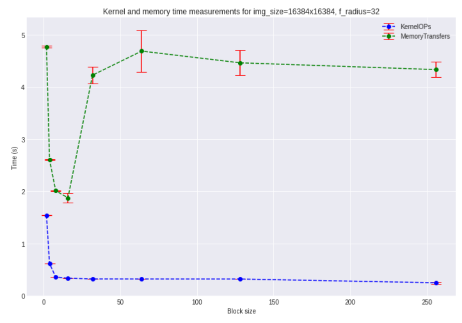


Figure 2: Kernel and Memory operation execution times for different filter_radius sizes and 8192×8192 image size.

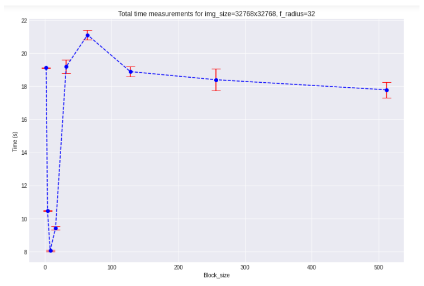


(a) GPU total execution time.

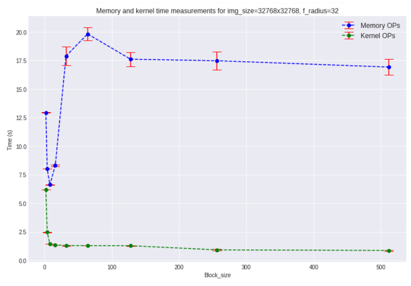


(b) Kernel and Memory time in GPU.

Figure 3: GPU execution times for image size 16384×16384, filter radius 32 and variable block size.



(a) GPU total execution time.



(b) Kernel and Memory time in GPU.

Figure 4: GPU execution times for image size 32768×32768, filter radius 32 and variable block size.

parallelism and overlap between memory transfers and kernel operations. In order to have true parallelization, we had to use pinned memory in the host system via `cudaMallocHost`. The different timelines from implementations in steps 3 and 4 can be seen in Figures 5 and 6.

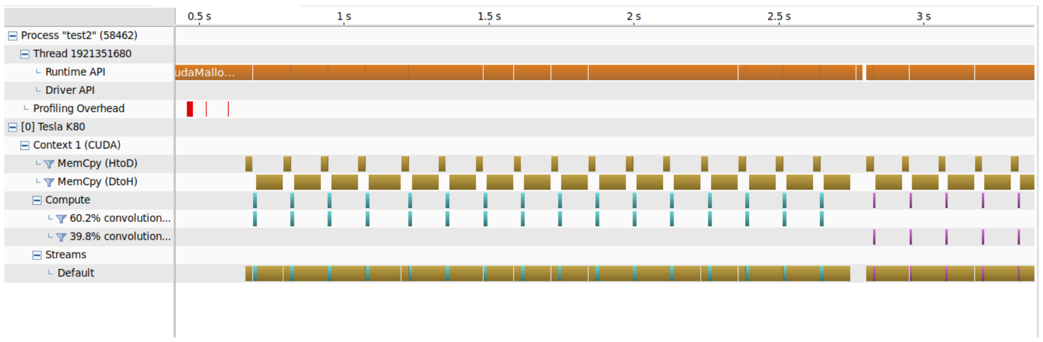


Figure 5: Timeline of blocked implementation.

Step 5 - Conclusion: The maximum image size we could get before we ran out of memory was 32768×32768, and our bottleneck this time is the host machine's RAM instead of the device's global memory.

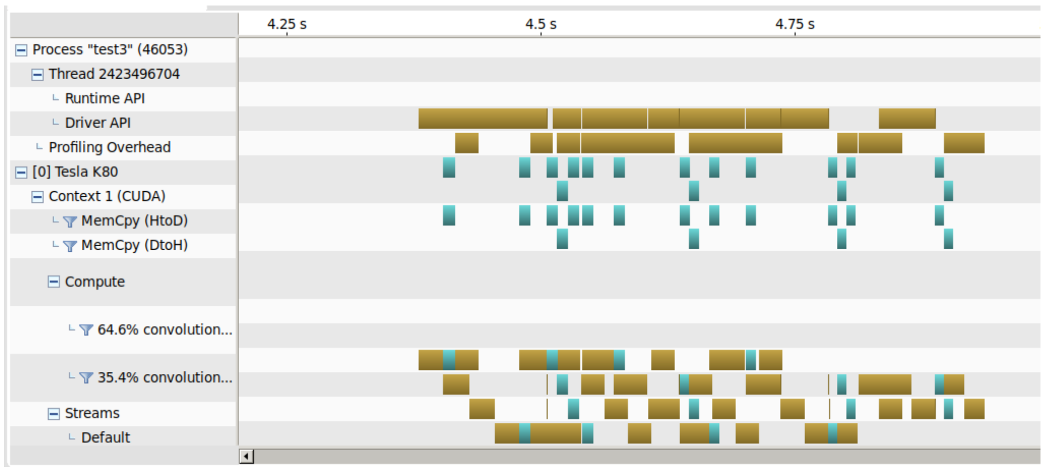


Figure 6: Timeline of "streams" implementation.