# Final Project - Parallel Image Processing

Luca Cazzola
*Student-ID: 248716*
Trento, Italy
luca.cazzola-1@studenti.unitn.it

Christian Dalvit
*Student-ID: 249988*
Trento, Italy
christian.dalvit@studenti.unitn.it

## I. Introduction

The objective of this final project is to parallelize image filtering algorithms and to measure and analyze various metrics of the algorithm. This report provides a description of the problem setting, algorithms, and experimental results of the implementation. The code used for this project is made available through a public Github repository.

## II. Problem Description

When applying a filter to an $n$-dimensional signal ($n \in \mathbb{N}^+$) one of the most common and effective operation is the convolution. In a 2-dimensional scenario, which is the case for **images**, operations such as like blurring (Fig. 1), sharpening and edge extraction can be performed using convolutions [2, 3]. Some deep learning architectures also massively take advantage of convolution to automatically extract significant features from images [4, 7]. The wide-ranging applications of image convolution underscore the need for an efficient implementation of the convolution operation. For a filter $w$ of size $n \times m$ and an image $f$ the convolution operation is defined as :

$$w(x,y) * f(x,y) = \sum_{s=0}^{m-1} \sum_{t=0}^{n-1} w(s,t) \cdot f(x-s, y-t)$$

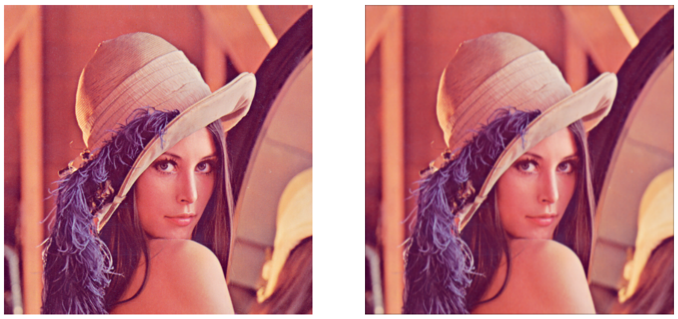

(a) Original (b) Blurred

Fig. 1: Example of Gaussian blur filter applied with convolution operator

### A. Design choices & assumptions

We assume $m, n$ to be odd numbers and that $m = n$. Hence, we are only considering square filters with odd dimensions. We further assume that the filter $w$ has only one channel and that the image $f$ can have multiple channels. In case of a multichannel image $f$, the filter $w$ is applied to each channel separately. Since the convolution is not well-defined along the image boundary, a boundary-handling strategy needs to be implemented. We decided to implement zero padding for all algorithms in this project. This means that we assume all pixels outside the image to be zero. The computation for a single output pixel $(w * f)(x, y)$ depends only on the filter $w$ and the neighborhood of $f(x, y)$. This makes the convolution operation suitable for parallelization. As most images have multiple channels as in the case of RGB domain, the way channels are stored in the memory matters.

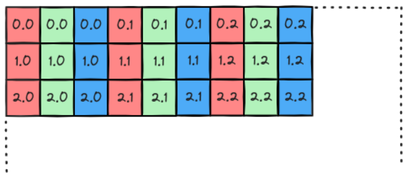


Fig. 2: Interleaved Format

Storing closely channels corresponding to the same pixel is generally referred to as **interleaved format**. It is effective when performing operations across multiple channels due to their spatial locality. It is the most used format when it comes to general image processing (not specifically convolution). Popular frameworks such as OpenCV as well as formats such as PNG adopt such structure.

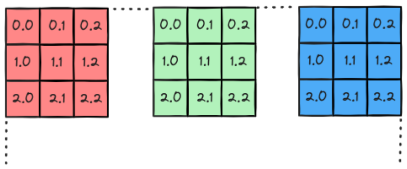


Fig. 3: Planar Format

A radically different choice is to separate channels, so that spatial locality is preserved at channel level. This is called **planar format** and is more effective when operations are independent between different channels, which is generally true in the case of convolution. It's worth noting that there exist cases such as the $(1 \times 1)$ convolution [4], which is largely used in deep learning and aims to specifically compute over the channel dimension compact the number of channels. For this reason and for the sake of simplicity, **we adopted an interleaved format**, being aware it will probably worsen performances.

## III. State of the art

Most used 2D convolution strategies mainly fall into 3 categories :

1) **Overlap & Add** : follows the divide and conquer principle. The input image is divided into smaller patches which are processed independently. Once patches are evaluated, results are aggregated [1]. Solutions falling into this category try to make clever use of GPU memory sub-system to parallelize processing as much as possible.

2) **Convolution on Fourier domain** : Leveraging on the fact convolution becomes element-wise multiplication in the Fourier domain, both the image and the given kernel can be transformed in Fourier domain, apply convolution and then transform back to the original space. The main issue with such method is the inputs dimensionalities, which might be too large.

3) **GEMM based convolution** : Motivated by the fact GPUs are heavily optimized for matrix multiplication, the convolution is reframed to fit such paradigm and is computed as GEMM [6].

**All our implemented solutions fall into the overlap & add category**, which is probably the most simplistic, but yet can be effective.

## IV. Implementations

In the following, the different algorithms implemented during this project are described and their memory access pattern is discussed. Note that we dropped the `for`-loops for the channels of the image from the pseudocode for better readability.

### A. CPU Algorithm

The CPU implementation is used as a baseline for benchmarking the various GPU algorithms. (Algo. 1) is a straightforward implementation of the mathematical convolution definition. The filter $K$ is applied sequentially to every pixel in the image of $I$. The obvious drawback of this implementation is that no parallel processing is leveraged for a more efficient computation.

**Algorithm 1** CPU implementation

```
Input: Image $I$, Kernel $K$
Output: Image $O$
 1: $k_c \leftarrow (\dim(K)-1)/2$
 2: for $u = 0$ to height$(I)-1$ do
 3:     for $v = 0$ to width$(I)-1$ do
 4:         $sum \leftarrow 0$
 5:         for $i = 0$ to $\dim(K)-1$ do
 6:             for $j = 0$ to $\dim(K)-1$ do
 7:                 $p_u \leftarrow u - k_c + i$
 8:                 $p_v \leftarrow v - k_c + j$
 9:                 if $p_u, p_v \in I$ then
10:                     $sum \leftarrow sum + K(i,j) \cdot I(p_u, p_v)$
11:                 end if
12:             end for
13:         end for
14:         $O(u,v) \leftarrow sum$
15:     end for
16: end for
```

### B. GPU Naive

(Algo. 2) is a naive implementation for parallel convolution computation. The idea behind the algorithm is the same as in (Algo. 1). But instead of processing each pixel sequentially, all pixels are computed in parallel. Each thread applies the kernel to all image channels. Although this algorithm should compute the convolution more efficiently, there is still room for improvement. For example, the (Algo. 2) stores the whole image and the kernel in the global memory of the GPU. This is not optimal in terms of memory access time.

**Algorithm 2** Naive CUDA kernel

```
Input: Image $I$, Kernel $K$
Output: Image $O$
 1: $u \leftarrow blockIdx.x \cdot blockDim.x + threadIdx.x$
 2: $v \leftarrow blockIdx.y \cdot blockDim.y + threadIdx.y$
 3:
 4: if $u, v \in I$ then
 5:     $sum \leftarrow 0$
 6:     for $i = 0$ to $\dim(K)-1$ do
 7:         for $j = 0$ to $\dim(K)-1$ do
 8:             $p_u \leftarrow u - k_c + i$
 9:             $p_v \leftarrow v - k_c + j$
10:             if $p_u, p_v \in I$ then
11:                 $sum \leftarrow sum + K(i,j) \cdot I(p_u, p_v)$
12:             end if
13:         end for
14:     end for
15:     $O(u,v) \leftarrow sum$
16: end if
```

**Note:** If a variable is not defined consider the last declaration among previous algorithms

### C. GPU Shared Memory

The GPU Shared Memory implementation in (Algo. 3) leverages the shared memory of the GPU. Computation requires many repeated accesses to the kernel and the image patch. Faster memory access time for the kernel and the image patch should benefit the performances. In (Algo. 3) an image patch and the kernel are copied into shared memory. An image patch is composed of an inner patch plus a padding component.

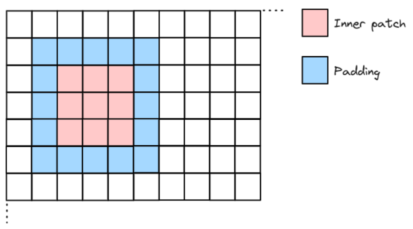


Fig. 4: Patch structure example (3×3 kernel)

The kernel is centered on all inner patch pixels. Padding is needed when evaluating the inner patch edges, as they're required by the kernel. Although (Algo. 3) takes advantage of the shared memory, there are some drawbacks. The workload is not distributed equally among the threads, as the first thread of each block must copy the kernel. Furthermore, the kernel gets copied into the shared memory once per thread block. This is redundant work, because the kernel is static and does not change during the computation.

**Algorithm 3** Shared memory CUDA kernel

**Input:** Image $I$, Kernel $K$
**Output:** Image $O$

```
1: $\texttt{OUT\_DIM} \leftarrow blockDim.x - 2k_c$
2: $col \leftarrow blockIdx.x \cdot \texttt{OUT\_DIM} + threadIdx.x - k_c$
3: $row \leftarrow blockIdx.y \cdot \texttt{OUT\_DIM} + threadIdx.y - k_c$
4:
5: __shared__ $I_s$
6: $K_s \leftarrow$ get_kernel_address($blockDim.x$)
7: if $threadIdx.x = 0 \land threadIdx.y = 0$ then
8:     kernel_to_shared_mem($K$, $K_s$)
9: end if
10: image_to_shared_mem($I$, $I_s$, $threadIdx$, $row$, $col$)
11: __syncthreads()
12:
13: $row_t \leftarrow threadIdx.y - k_c + i$
14: $col_t \leftarrow threadIdx.x - k_c + j$
15: if $row, col \in I \land col_t, row_t \in [0, \texttt{OUT\_DIM}]$ then
16:     $sum \leftarrow 0$
17:     for $i = 0$ to $\dim(K) - 1$ do
18:         for $j = 0$ to $\dim(K) - 1$ do
19:             $sum \leftarrow sum + K_s(i,j) \cdot I_s(row_t + i, col_t + j)$
20:         end for
21:     end for
22:     $O(u,v) \leftarrow sum$
23: end if
```

**Note:** If a variable is not defined consider the last declaration among previous algorithms

## D. GPU Shared Memory using Constant Memory

Since the kernel values do not change during the computation and the kernel size is relatively small, the kernel can be placed in the GPU's constant memory. Constant memory is a special part of the GPU's global memory, that is cached for efficient accesses [5]. (Algo. 4) implements the same strategy as (Algo. 3), but instead of moving the kernel into shared memory, the kernel is copied into constant memory before the kernel execution. Note that the kernel is not part of the input, because in CUDA the kernel stored in constant memory $K_c$ acts as a global variable. (Algo. 4) resolves some drawbacks of the previous implementation. The workload is distributed more equally among the threads, and the kernel is copied only once to the GPU. Since all threads have the same access pattern for $K_c$, the access of values in $K_c$ should be efficient [5].

**Algorithm 4** Shared + constant memory CUDA kernel

**Input:** Image $I$
**Output:** Image $O$

```
1: __shared__ $I_s$
2: image_to_shared_mem($I$, $I_s$, $threadIdx$, $row$, $col$)
3: __syncthreads()
4:
5: $col_t \leftarrow threadIdx.x - k_c$
6: $row_t \leftarrow threadIdx.y - k_c$
7: if $row, col \in I \land col_t, row_t \in [0, \texttt{OUT\_DIM}]$ then
8:     $sum \leftarrow 0$
9:     for $i = 0$ to $\dim(K) - 1$ do
10:         for $j = 0$ to $\dim(K) - 1$ do
11:             $sum \leftarrow sum + K_c(i,j) \cdot I_s(row_t + i, col_t + j)$
12:         end for
13:     end for
14:     $O(u,v) \leftarrow sum$
15: end if
```

**Note:** If a variable is not defined consider the last declaration among previous algorithms

## E. GPU Shared Memory using Cache

(Algo. 5) implements a hybrid approach, by using both global and shared memory, while the kernel is in constant memory. Instead of using a smaller output dimension of each thread block as in (Algo. 3) and (Algo. 4), (Algo. 5) has the same output dimension as the thread block. When computing the convolution, image pixels which are inside the image patch processed by the thread block are loaded from shared memory. The other pixels are loaded from global memory. One can observe that in (Algo. 4) the padding of an image patch overlaps with the inner patch of its neighboring patches. Therefore, there is a significant probability, that the padding of an image patch is already in L2 cache [5]. Hence, the padding can be efficiently accessed, without extra coping the padding into shared memory for every image patch.

**Algorithm 5** Cached shared + constant memory CUDA kernel

**Input:** Image $I$
**Output:** Image $O$

```
1: $col \leftarrow blockIdx.x \cdot blockDim.x + threadIdx.x$
2: $row \leftarrow blockIdx.y \cdot blockDim.y + threadIdx.y$
3:
```

```
4: __shared__ $I_s$
5: image_to_shared_mem(I, $I_s$, $threadIdx$, $row$, $col$)
6:
7: __syncthreads()
8:
9: if $col, row \in I$ then
10:     $sum \leftarrow 0$
11:     for $i = 0$ to $\dim(K) - 1$ do
12:         for $j = 0$ to $\dim(K) - 1$ do
13:             $p_u \leftarrow threadIdx.x - k_c + i$
14:             $p_v \leftarrow threadIdx.y - k_c + j$
15:             if $p_u, p_v \in [0, blockDim.x]$ then
16:                 $sum \leftarrow sum + K_c(i,j) \cdot I_s(p_u, p_v)$
17:             else
18:                 if $p_u, p_v \in I$ then
19:                     $sum \leftarrow sum + K_c(i,j) \cdot I(p_u, p_v)$
20:                 end if
21:             end if
22:         end for
23:     end for
24:     $O(u,v) \leftarrow sum$
25: end if
```

**Note:** If a variable is not defined consider the last declaration among previous algorithms

## V. Experimental Results

### A. Experimental Setup

Benchmarks are done on the University of Trento DISI department cluster

**CPU setup** :

- **Model name** : Intel(R) Xeon(R) Gold 6238R CPU @ 2.20GHz, cores : 28
- **Cache** : 32K (L1), 1024K (L2), 39424K (L3)

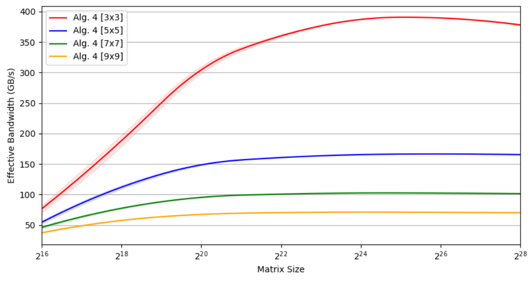
**GPU setup** :

- **Model name** : NVIDIA A30
- **Architecture** : Ampere - 8.0 compute capability
- **L2 cache size** : 25165824 bytes
- **Peak memory bandwidth** : 933 GB/s
- **Peak FP32** : 10.3 TF

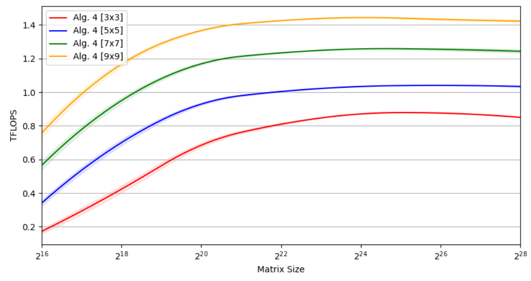
### B. Results

We're interested in measuring both the effective bandwidth (GB/s) and FLOPS (TF/s) of our implementations. Each algorithm is tested multiple times to capture mean and standard deviations of metrics at varying input size and kernel sizes. Each input to the algorithms is a randomly generated FP32 matrix of size Matrix size $\times$ $C$, with $C$ fixed to 3 to simulate real image's number of channels. Each kernel is executed with fixed block size of (16$\times$16).

Benchmark results (Fig. 6) show clear trends: as expected, the CPU implementation is totally out of scale w.r.t. the GPU ones and not worth discussing. All GPU kernels reach a stationary point as the L2 cache is filled in between $2^{20}$ and $2^{22}$ matrix size steps. The best performing implementation among all is (Algo. 4) which unexpectedly performs very close to (Algo. 3). It's true that loading the kernel multiple times is unnecessary overhead as it's fixed, but considering the filter is constantly being accessed it is going to be almost permanently available on L2 cache. Another consideration is that kernel filters on many applications (especially deep learning [4]) are generally small, which makes the number of values to load tiny. Convolution becomes more and more expensive to compute as the kernel size increases, as shown in (Fig. 5). Small kernels tilt the problem towards memory management efficiency, larger kernels instead makes it more arithmetically challenging:



(a) Effective bandwidth



(b) FLOPS

Fig. 5: Cost of kernel sizes on (Algo. 4)

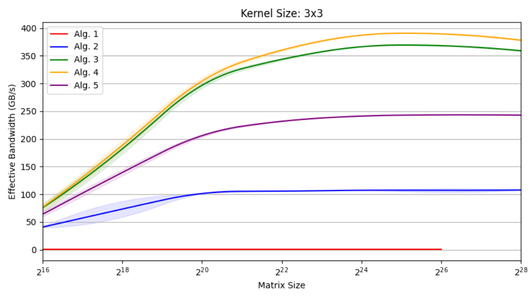
***N.B.** The X axis doesn't include the C factor for readability purposes*

When having a wide kernel becomes necessary in order to capture wider domain regions, a trade-off consists into using small to medium dilated kernels [8] which allow to "look further" while saving computational resources.

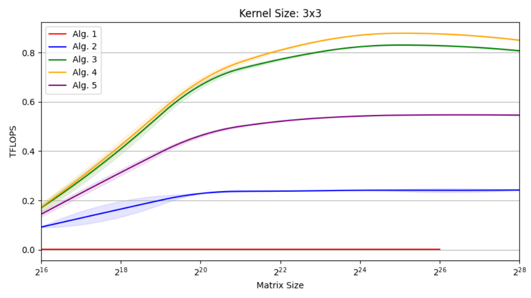
Going back to (Fig. 6), underwhelming but not unexpected are the performances of (Algo. 5) which likely heavily suffers from our choice of using an interleaved format (Fig. 2).

## VI. Conclusion

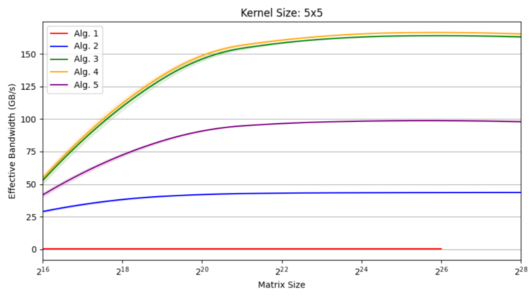
Considering our best configurations for both effective bandwidth ($\sim$400 GB/s) and FLOPS ($\sim$1.4 TF/s) it's evident we're still far from theoretical peaks, which suggests there's very much room for improvement. If we had to further continue this project, we would change our image storage formats to a planar one (Fig. 3) and test some GEMM based implementations, which seems promising.



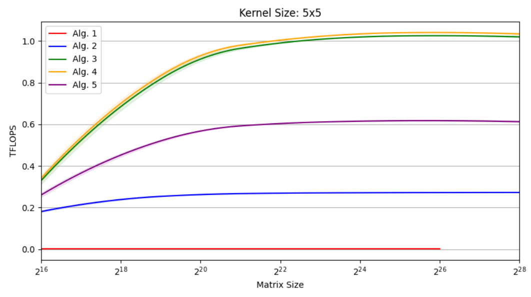
(a) Kernel (3×3) - Effective bandwidth



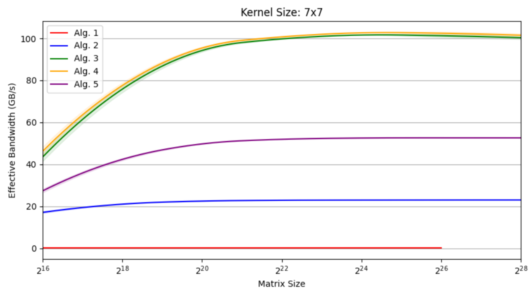
(b) Kernel (3×3) - FLOPS



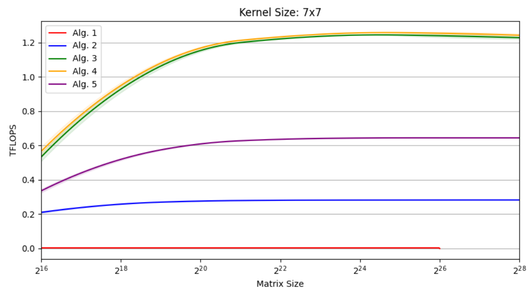
(c) Kernel (5×5) - Effective bandwidth



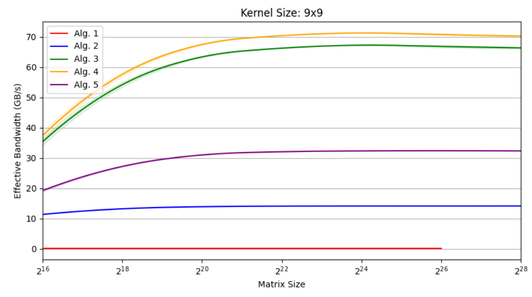
(d) Kernel (5×5) - FLOPS



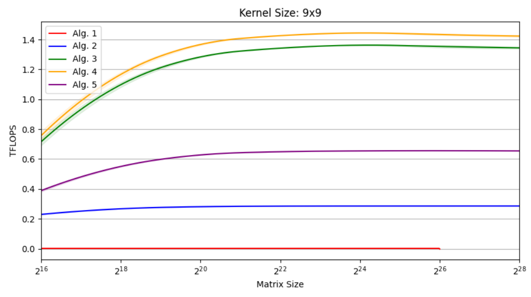
(e) Kernel (7×7) - Effective bandwidth



(f) Kernel (7×7) - FLOPS



(g) Kernel (9×9) - Effective bandwidth



(h) Kernel (9×9) - FLOPS

Fig. 6: Benchmark

***N.B.*** *The X axis doesn't include the $C$ factor for readability purposes*

## VII. Contributions

The team started by splitting some of the core tasks :

Luca Cazzola (248716)

- Backbone structure supporting the project (taken partially from previous homeworks)
- Functions to handle PNG files based on the official libpng library
- Benchmarking & results plotting components

Christian Dalvit (249988)

- Research of theoretical material supporting the project
- Implementation of convolution algorithms (1, 2, 3)

Even though the team split the workload, members constantly reached one with another to share opinions, make design choices and resolve issues. The report compilation and non-listed parts have been performed equally by both team members.

## References

[1] Karel Adámek et al. “GPU Fast Convolution via the Overlap-and-Save Method in Shared Memory”. In: *ACM Transactions on Architecture and Code Optimization* 17.3 (Aug. 2020), pp. 1–20. ISSN: 1544-3973. DOI: 10.1145/3394116. URL: http://dx.doi.org/10.1145/3394116.

[2] D. Forsyth and J. Ponce. *Computer Vision: A Modern Approach*. Pearson, 2012. ISBN: 9780136085928.

[3] R.C. Gonzalez and R.E. Woods. *Digital Image Processing*. Addison-Wesley world student series. Addison-Wesley, 1992. ISBN: 9780201508031.

[4] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV]. URL: https://arxiv.org/abs/1512.03385.

[5] Wen-mei W. Hwu, David B. Kirk, and Izzat El Hajj. *Programming Massively Parallel Processors*. 4th Edition. Elsevier, 2023. ISBN: 9780323984638.

[6] Mickael Seznec et al. “Computing Large 2D Convolutions on GPU Efficiently with the im2tensor Algorithm”. In: *Journal of Real-Time Image Processing* 19 (2022), pp. 1035–1047. DOI: 10.1007/s11554-022-01240-0. URL: https://doi.org/10.1007/s11554-022-01240-0.

[7] Sanghyun Woo et al. *ConvNeXt V2: Co-designing and Scaling ConvNets with Masked Autoencoders*. 2023. arXiv: 2301.00808 [cs.CV]. URL: https://arxiv.org/abs/2301.00808.

[8] Fisher Yu, Vladlen Koltun, and Thomas Funkhouser. *Dilated Residual Networks*. 2017. arXiv: 1705.09914 [cs.CV]. URL: https://arxiv.org/abs/1705.09914.