Parallel Computing Project: Kernel Image Processing

Giovanni Burbi
giovanni.burbi@stud.unifi.it

Abstract

Many image processing software make use of masks to apply filters that modify images. Depending on which mask has been used we can perform blurring, sharping, embossing and more. In order to do this, we must perform a convolution between the selected mask and an image. This kind of process consist of adding each element of the image to its local neighbors, weighted by the mask. The convolution operation is embarrassingly parallel, meaning that we can perform the operation on each element at the same time without incurring in race conditions. In this project, we will implement a sequential version of kernel image processing in c++ and then we will improve the performance of the program using parallel programming techniques and frameworks. We will use implicit and explicit threading frameworks such as OpenMP and CUDA and we will compare the results obtained from the experiments carried out with all our implementations on images of different sizes.

Key words: Parallelism, OpenMP, Kernel image processing, SIMD, Vectorization, CUDA, Cuda Async, Pinned memory, Array of structures, Structure of array.

1. Introduction

Kernel image processing is a technique to manipulate and extract features from images. There are many applications that make use of this kind of process to elaborate images. Depending of which kernel or **mask** (we will use this term to not confuse with *CUDA kernels*) has been used we can obtain different results such as blurring, edge detection, sharpening and more. In *figure 1* there is an example of kernel image processing with an outline mask.

We can obtain a new processed image by applying a mathematical operation, the **convolution**, between an image and the selected mask. A mask is a two-dimensional squared matrix of numbers. We can choose the size of this matrix; usually it's small. Here, we will use a 3x3 dimensional mask. We will treat an image as a two-dimensional matrix of pixels, each pixel is represented with three numeric values for each of the three channels: red, green and blue. In this project, we will consider only RGB images. During the convolution, the mask will slide over the image and we will perform the computations using the convolution function:

$$w * f(x,y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s,t) f(x-s, y-t) \qquad (1)$$

with *w* the mask function and *f(x, y)* the image.

For each pixel, we center the mask over that pixel, multiply the mask values with the corresponding pixel neighbors values of the center pixel and add everything together to obtain the new value of the current pixel of the new image.

2. Pseudo-code

A pseudo-code of the convolution is shown in *algorithm 1*. In the algorithm, a valid position is when the mask and the image overlap. We will not perform the multiplication if the element of the mask goes outside the boundaries of the image. Therefore, we will have corrupted results on the border that will make the edges appear different from the rest of the image. Alternatively, we can skip any pixel in the input image which would require values from beyond the edge by starting from inside the image and ending before the opposite edge. This method will result in the output image being slightly smaller because we will lose *radius of mask* pixels of the border of the image, with the edges having been cropped.

Algorithm 1 Convolution algorithm

```
procedure CONVOLUTION(image, mask)
    for each pixel in image do
        accum ← 0
        for each element in mask do
            if valid position then
                accum += pixel * element
        outputPixel ← accum
    return output
```

FIGURE 1: *Confrontation between the original image and a processed image obtained by performing a convolution with an outline mask*

3. Sequential implementation

The sequential implementation has been written in c++ and it follows the pseudo-code shown in *algorithm 1*, where we choose to crop the edges of the image. The images used for this project are in **ppm format**. This format represents an image as a text file without any compression. It has a header with some metadata about the image, such as the image's dimensions and the number of channels, and a payload with every pixel represented as a sequence of r, g, b values written in chars, assuming the image to have RGB channels.

The image information are then stored in a struct that represents the pixels with an array using the pointer notation of c. Therefore we will have a long sequence of values in the form r, g, b, r, g, b... each triplet represents a pixel. This kind of layout is called **Array of Structure** (AoS) and it is the standard practice when we need to store many of the same type of objects, but it's not cache friendly. In fact, every time we read a certain value, we will also bring in the cache some of the following values thanks to the memory burst. With the AoS layout it can occur that a certain pixel will not be entirely inside the cache line, so we would need to read once more in order to have all the values of that pixel. We can be cache friendlier using the **Structure of Arrays** (SoA) layout, where we have inside a struct, three arrays for the red, green and blue channel. This way we won't have problems of partial object in a cache line and we will have a **more efficient memory access**.

This will be particularly important for CUDA to allow consecutive threads access consecutive memory addresses, in other words we will have a **coalesced memory access**. The device coalesces global memory loads and stores issued by threads of a warp into as few transactions as possible to minimize DRAM bandwidth. This layout will result in better performance in each implementation on CPU and GPU.

The values of each channel were normalized in a range of $[0, 1]$ based on a depth parameter. In this way we can have different color representation changing only the depth value. We created a mask file where we create and return different kinds of masks based on a value of an enum type.

Lastly, we created a convolution file where we defined different implementations of the procedure that performs the convolution operation on a certain image with a certain mask passed as parameters.

We implemented a **naive sequential version** using the **AoS layout** and another using the **SoA layout**. Then we tried to improve the performance of the sequential version by implementing a version of the procedure that performs manual **loop unrolling** , making some assumptions about the inputs. In fact, given that we will use only 3x3 masks, we can obtain a boost in performance by **replacing the loop over the mask** with the explicit formula, adding all the terms in a single line of code. This will allow the compiler to do **vectorization**, meaning using **SIMD instructions** that exploit the CPU hardware parallel mechanism to perform the additions simultaneously, *figure 2* shows an example of the process. To do this, the pointers to the sequence of pixels values are made **restrict** to limit the effects of **pointer aliasing**, hinting to the compiler that for the lifetime of the pointer, no other pointer will be used to access the object to which it points. This allows the compiler to make **optimizations** such as vectorization, that would not otherwise have been possible. When we are working with a three channel image, like RGB, we can also **unroll the channels loop**.

We note one aspect about the SoA layout. As we said before, using the SoA data organization means using a struct with three arrays for each channel that we need to update separately. Therefore, the **SoA layout**, as we have implemented it, **involves an unroll over the channels loop**. This will give better performance in terms of cache utilization and vectorization with respect to the AoS layout implementations.

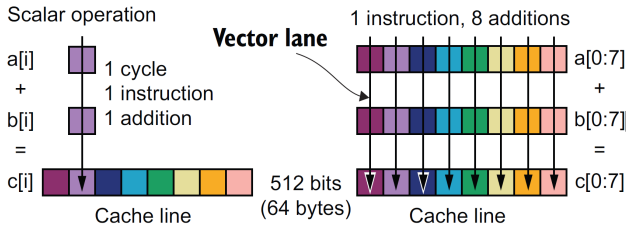

FIGURE 2: *Example: a representation of vectorization in the CPU*

4. Parallel implementation: OpenMP

The convolution operation is **embarrassing parallel**, meaning, it can be done simultaneously without the need to synchronize anything in order to perform this operation with more threads. Each thread will receive its chunk of data to elaborate and fill the corresponding positions on the result array when the new values have been computed.

We will use a framework called **OpenMP** that allows us to write parallel programs without explicitly defining the threads. In fact, **OpenMP** is an **implicit threading framework** that make use of compiler directives to create implicitly a certain number of threads and assign a chunk of data to each of them. This framework uses the *fork-join pattern* to create the threads when they are needed in certain sections of the code. We have little flexibility with OpenMP, but we also **need to change very little, if any, of the sequential implementation** to have a working parallel version with good performance.

A **naive implementation** was obtained inserting a OpenMP directive, a *parallel for*, in the outermost for loop of the sequential implementation. This will divide the image in vertical chunks for each thread. But, for the convolution, it would be best to divide the image in **squared chunks**. To do this, we can add *collapse(2)* to the previous directive in order to collapse the first two outermost for loop, corresponding to the loop over the width and height of the matrix that represents the image. Therefore, each thread will receive a portion of the row and columns of the matrix as a data chunk. The workload will be around the same for each thread because there is no computational difference based on which data a certain thread is responsible for in the convolution process. For this reason, to minimize useless overhead and coordination between threads we chose a **static scheduling** of the chunks assigned to the threads.

For this project, we have implemented a **naive version** as described above using a **AoS layout** and another version with the **SoA layout** for images with three channels.

Then we tried to improve the performance of the naive implementation with **three other versions using the AoS layout**:

- **Unroll Channels loop** (UC): Assuming only images with three channels, we manually performed a loop unrolling over the channels loop.
- **Unroll Mask loop and SIMD Channels** (UMSC): Assuming only 3x3 masks, and images with any number of channels, we performed a manual loop unrolling over the mask loop and added a directive *#pragma omp simd* that hints to the compiler to vectorialize the channels loop.
- **Unroll Mask and Channels loops** (UMC): Assuming only 3x3 masks and images with three channels, we performed a manual loop unrolling over the mask and the channels loops.

Lastly, we implemented a version with the **SoA layout** doing **loop unrolling over the mask loop**, assuming 3x3 masks. This is the equivalent of the UMC version described early but with the SoA layout.

5. Parallel implementation: CUDA

The convolution operation can be potentially performed simultaneously on every pixel of the image. Therefore we would like to **massively parallelize** this process to obtain even better performance. To do this, we used an **explicit threading framework**, CUDA.

CUDA is an architecture hardware and API for parallel programming on NVidia GPUs. The GPUs are called **device**, while the computer that make use of the GPU is called **host**. The device is where we will perform the massive parallel computation. CUDA works at its best when there is not much synchronization involved, if any. Therefore it is perfect for our case, to perform the convolution operation over an image.

The standard procedure to work with NVidia GPUs is:

- **Allocate** GPU memory
- **Transfer** data from **host to device**
- **Perform computation** inside the device invoking **CUDA kernels**
- **Transfer** results data from **device to host**
- **Free** GPU memory

The method that performs the computation inside the device is called **kernel**, we can specify how many threads will work on the data passed to the kernel by defining the **layout of the grid**.

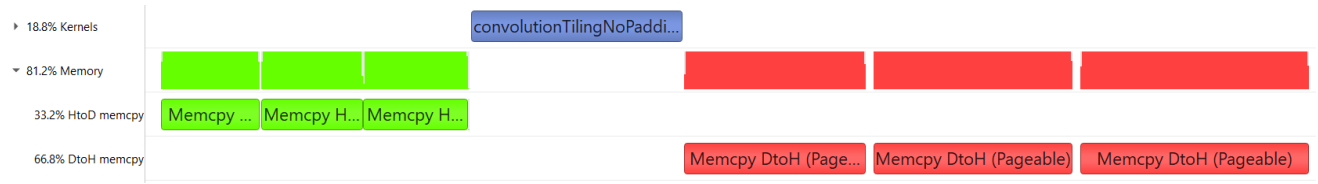

FIGURE 3: *Synchronous data transfer to and from the device, single kernel*

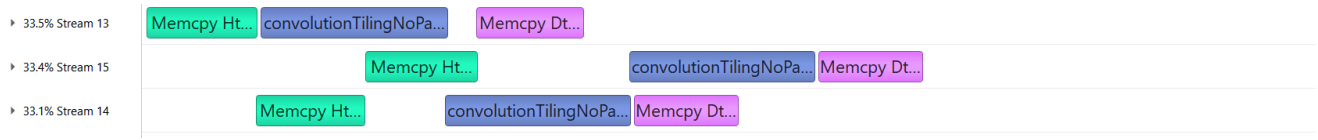

FIGURE 4: *Asynchronous data transfer on three streams, one for each channel of a RGB image. Pinned memory required. CUDA allows only one Memcpy and one kernel execution at a time. We can perform the convolution to one channel at a time thanks to using the SoA layout of data*

FIGURE 5: *Nsight Systems stream analysis; confrontation SoA tiling implementation with blocking memory transfer, using the default stream 0, and asynchronous transfer host to/from device using a stream for each channel of a RGB image.*

In fact, to define the grid we must decide the number and layout of the **blocks** that compose this grid and the **threads** that form these blocks. We want to map the grid to the input, or the output image, that we will use to perform the convolution in a way that the grid can cover all of the image. Each thread is characterized by an **index** that identifies it inside a certain block in the grid. This index will be necessary to decide what type of role a certain thread needs to fulfill in the CUDA kernel. Some threads will need to compute the convolution of a pixel or load data in the shared memory, others, outside the image, don't need to do anything.

$$dimGrid = (ceil(outputWidth/BLOCK\_WIDTH), ceil(outputHeight/BLOCK\_WIDTH)) \quad (2)$$

$$dimBlock = (BLOCK\_WIDTH, BLOCK\_WIDTH); \quad (3)$$

BLOCK_WIDTH is a constant that we can define with the only constrain, imposed by CUDA, that the **total number of threads per block is less than 1024**.

We won't perform the convolution on the image edges because we decided to **crop these out**. Therefore, we have mapped the threads over the **output image dimensions**, that is a sub-image of the input. In CUDA kernels we will have to consider this aspect when we compute the coordinates of threads over the input image.

Another aspect to note is that NVIDIA GPUs divide threads of blocks in **warps**, groups of 32 threads. All threads in a warp execute the same instruction at the same time. It's the **smallest unit of concurrency**. For this reason, it's best to define the *BLOCK_WIDTH* constant in such a way as to have a **multiplier of the warp size** as the number of threads in blocks. This is to maximize the utilization of the device hardware.

We implemented a **naive version** with **AoS layout** and **SoA layout**, then we implemented a more performing version, called **Tiling**, by exploiting the **constant and shared memory** of the NVidia GPU, using a technique called **tiling** that in turn use another technique, called **two-batch loading**, to load the data required to perform the convolution on a tile in the shared memory.

To explain these techniques, we need to introduce another important aspect of GPUs: the **memories**.

5.1. Memories

There are several levels of memory on the GPU device, each one with distinct characteristics.

The slowest one is the **global memory**, that can be read and written by every thread in the grid. This type of memory has been used by the naive implementations of the convolution.

Then, we have a small, read-only, memory called **constant memory** that can be used to store the weights of the mask. This memory is second only to registers in terms of speed, and it's useful when we have to read many times the same values like in our case. It can be accessible from all threads of the device.

Then we have **registers**, the fastest memory in the device, that are dynamically assigned to each thread and it's where local variables are stored.

Finally, every thread in a block has access to a unified **shared memory**. It's a read-and-write memory much smaller than the global memory but also faster (it is bigger but slower respect to the constant memory).

5.2. Tiling

The convolution require to read many times the same values near a certain pixel, so, instead of reading many times the same values from the global memory, which is the slowest memory we can use, we can load these values in the **shared memory** so that threads of the same block can access these much faster with only one access to the global memory.

The shared memory is small compared to the global memory, and **using it too much can worsen the performance** of the program, so we used a technique called **tiling** to partition the data into subsets called tiles so that we can store them in the shared memory. The convolution can be performed on these tiles independently of each other.

But to perform the convolution we need **some data of adjacent tiles** to compute the values at the edges of a certain tile, so, to load all the data we need from the global memory to the shared memory efficiently, we have to use use another technique called **two-batch loading**.

Essentially a tile of *TILE_WIDTH * TILE_WIDTH* elements will need *(TILE_WIDTH + MASK_WIDTH - 1) * (TILE_WIDTH + MASK_WIDTH - 1)* data to be stored in the shared memory. We mapped the tile to the **output image**, so we took the dimension of the blocks of the grid equal to that of the tile. Therefore, each thread will move **at the most two elements** from global memory to shared memory in a two stage process. There will be some threads that will read in memory only once but all the threads will be responsible to perform the convolution on one pixel if they are mapped inside the boundaries of the image, excluding the edges because they will be cropped out.

5.3. Asynchronous loading

Finally, we created other versions for the naive and the tiling implementations with SoA layout performing the data transfer from and to the device **asynchronously**, adding in this way, another layer of concurrency to the program. In fact, while launching a CUDA kernel, it doesn't block the calling host thread, the data transfer does.

Using the asynchronous transfer from a non default stream, if the host memory is **pinned** and if the device has a free DMA copy engine, we let the copy operation happen while the GPU **simultaneously** performs another operation such as a kernel execution.

In this asynchronous version, in one stream we will load a first batch of data consisting of the red channel values of all pixels, then when this first transfer is completed, we perform the convolution on these data in the device's memory while we load the data of channel green from the host to the device in another stream and so on. This can be done also to transfer the results of one of the image's channels from the device to the host. *Figure 5* shows the profiling of the program execution on the device with *NVidia Nsight Systems*.

NVidia's GPUs use streams to allow this kind of asynchronous behaviour. To enable not-defualts stream we need to introduce the *–default-stream per-thread* option to the nvcc compiler. We need to copy the data in the host, stored in a pageable memory, to a pinned memory location always on the host in order to allow the asynchronous transfer between host and device using a free DMA copy engine.

Pinned memory means that the memory is locked in the RAM memory, so the device can fetch it without help from CPU using **DMA**. The operating system can't touch this locked memory space. Not-locked memory (pageable) can generate a page fault on access, and it can be stored in memory or be in swap, so the driver needs to access every page of non-locked memory, copy it into pinned buffer and pass it to DMA. This is what happens in a synchronous transfer.

Therefore, we first allocate the pinned memory on the host, then we perform a **transfer host-to-host** into this new allocated memory from a pageable memory space and only then, we can perform the asynchronous transfer. This results in a reduction of the overhead to transfer data between host and device.

Algorithm 2 shows the in depth steps to perform asynchronous data transfer with CUDA. For space reasons only the stream over the red channel of the image is shown. We can see that we first allocate the memory in the device and define the pinned memory in the host. Then we copy the data with a host-to-host transfer from a pageable location to a pinned location in memory. At this point, we can perform the asynchronous data transfer from the host to the device on a stream. On that same stream, we then call the kernel with the convolution function and, after it finishes its job, we copy asynchronously the results from device to host. The last steps are about the copying of the results on the pinned memory of the host to a pageable location with another host-to-host transfer and the freeing of all the memory locations on the device and the pinned memory of the host used to perform the asynchronous data transfer.

6. Experiments

We have done several experiments to test the different versions implemented to perform kernel image processing. We have used three types of images:

- A small image, 480p
- a full hd image, 1920x1080
- a 4k image

For all the experiments we have:

- The mask as a 3x3 matrix
- All images have RGB color model
- Execution times results from experiments are the average over 15 runs of the application.

All tests have been performed on a computer with a **Intel© Core™ i7-8750H CPU @ 2.20GHz with 6 cores / 12 threads** and a **NVIDIA GeForce GTX 1070 Mobile 8GB**.

Algorithm 2 Asynchronous data transfer

$CudaMalloc(imageR_d)$
$CudaMalloc(resultR_d)$

allocate pinned memory on host
$CudaMallocHost(imageR_h)$
$CudaMallocHost(resultR_h)$

copy data in pinned memory H2H
$CudaMemcpy(imageR, imageR_h)$
perform async transfer H2D
$CudaMemcpyAsync(imageR_h, imageR_d, StreamR)$

Execute kernel on grid
$kernel < B, T, StreamR > (imageR_d, resultR_d)$

perform async transfer D2H
$CudaMemcpyAsync(resultR_d, resultR_h, StreamR)$
copy data from pinned memory H2H
$CudaMemcpy(resultR_h, resultR)$

destroy allocated memory
$CudaFree(imageR_d)$
$CudaFree(resultR_d)$
$CudaFreeHost(imageR_h)$
$CudaFreeHost(resultR_h)$

7. Speedup

We will compare all the different implementations using the **speedup** as the metric to evaluate the performance. The speedup is defined as the ratio between the completion time of the sequential algorithm t_s and the completion time of the parallel algorithm t_p:

$$S = \frac{t_s}{t_p} \tag{4}$$

8. Results

Now are shown the results obtained from the various experiments carried out.

8.1. OpenMP

We first see the results obtained from the **OpenMP** (OMP) implementations compared to the sequential version. In *figure 6* we can see the **speedup** of the parallel implementations over the sequential on a 4k image using the **AoS layout** with an increasing number of threads. In this plot we compare the results obtained from a naive but general implementation to a more performing one that requires specific inputs. In other words, we show that we can obtain **better results losing generality over the inputs** of the convolution algorithm.

The lowest curve is about the **OpenMP naive implementation**; it consists of only adding the *# pragma omp parallel for* directive over the outermost loop of the convolution.

We can obtain better results assuming an image with three **channels**, this way, we can manually **unroll the channels loop** so that the compiler can perform vectorization thanks to SIMD instructions. This effect is even more evident in the other two curves where the trend is more than linear in a big part of the curves. This is thanks to the hardware parallel mechanisms of the CPU that **increment the effective concurrency** of the program beyond the parallelism given by the threads created by OpenMP.

In particular, the **UMSC curve** (Unrolling Kernel loop and SIMD Channels) is obtained **assuming a 3x3 mask** and a image with any number of channels. In this implementation, we unroll the mask loop and we add the SIMD directive that hints to the compiler to vectorize the channels loop. This implementation gives better performance than the previous ones, reaching a more than a linear trend when using less than 6 threads and after that, a sub-linear increase of the speedup with respect to the number of threads. In other words, the vectorization has the effect of producing more parallelism than what we would have with a certain number of threads.

Lastly, when we **assume an image with three channels and a 3x3 mask**, we can have the best performance using the mechanism just explained before by performing the manual unrolling on both the mask and the channels loop, as we can see from the top curve in figure 6.

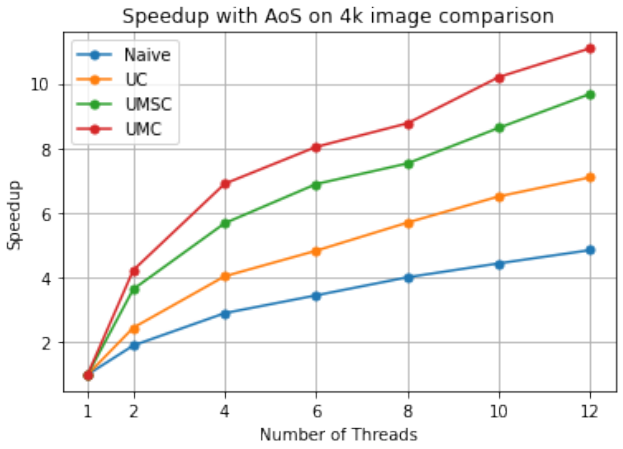

FIGURE 6: *Speedup comparison OpenMP implementations with AoS layout on 4k image varying the number of threads. Results obtained from times taken as the average over 15 runs of the algorithms. UC, Unrolling Channels. UMC, Unrolling Mask and Channels loop. UMSC, Unrolling Mask and SIMD channels loop*

As we did above, making the assumptions about the channels of the image or/and the size of the mask, we can also perform the manual unrolling even without a threading framework. In fact, the sequential implementation can obtain a good speedup exploiting the hardware parallelism given by the vectorization on the CPU.

In table 1 and table 2 we can see the times and the speedup with respect to the sequential version on all the image sizes used in the experiments, using 12 threads in the parallel implementations. We can confirm what we have said before about the relation between the losing of generality and the gain in performance.

Another aspect that we can use to obtain even better performance is the **organization, or layout, of the data in memory**. The results seen until now made use of the AoS layout, while now, in table 3 and table 4, we see the results using the **SoA layout**.

To use the SoA layout we must assume an **image with three channels** because the SoA's struct that contains the data is composed by **three arrays, one per channel**. To use this data struct we need to read or write on the three arrays separately so we won't have the loop over the channels. In other words, we must perform a sort of **loop unrolling over the channels**. Therefore, using the SoA layout will result in **performance gains** with respect to the AoS layout

Version	480p(ms)	Full HD(ms)	4k(ms)
Sequential	51.64	195.51	713.24
Sequential UC	36.54	135.72	549.71
Sequential UMC	18.07	99.69	312.55
OMP Naive	10.19	47.88	146.95
OMP UC	7.75	30.27	100.47
OMP UMSC	4.44	21.45	73.70
OMP UMC	3.87	18.09	64.32

TABLE 1: ***Times*** *obtained as average over 15 runs of the OpenMP implementations with AoS layout on all image sizes. Parallel implementations are done with 12 threads. UC, Unrolling Channels. UMC, Unrolling Mask and Channels loop. UMSC, Unrolling Mask and SIMD channels loop*

Version	480p	Full HD	4k
Sequential UC	1.41	1.44	1.30
Sequential UMC	2.86	1.96	2.28
OMP Naive	5.07	4.08	4.85
OMP UC	6.66	6.46	7.10
OMP UMSC	11.63	9.12	9.68
OMP UMC	13.35	10.81	11.09

TABLE 2: ***Speedup*** *obtained from the OpenMP implementations with AoS layout on all image sizes. Parallel implementations are done with 12 threads. UC, Unrolling Channels. UMC, Unrolling Mask and Channels loop. UMSC, Unrolling Mask and SIMD channels loop*

implementation from two sources: the forced unrolling over the channels and the **cache friendlier** data organization in memory. Even the sequential implementation with the SoA layout results in great performance improvement in terms of execution times compared to the AoS implementation.

The speedup in table 4 is calculated with respect to the times obtained from the **SoA sequential implementation**.

We called it sequential because we didn't use any threading framework to implement it, but, as we said above, this sequential version with the SoA layout benefits from the **hardware parallelism** of the CPU exploited by the vectorization implicit in the loop unrolling over the channels, as we explained before. Therefore, we will obtain **smaller speedups** of the OpenMP parallel implementations with SoA layout compared to the ones obtained with the AoS layout that where calculated using the AoS sequential implementation. In fact, the AoS sequential implementation is **much slower** than the SoA sequential one because it doesn't exploit the hardware parallelism given by the vectorization of the manual unrolled loops.

Lastly, in *table 5* and *figure 7*, we confront the two data layouts in term of execution times and in term of speedup.

Version	480p(ms)	Full HD(ms)	4k(ms)
Sequential	31.87	123.04	472.53
Sequential UM	14.25	99.69	281.89
OMP Naive	6.02	27.03	94.54
OMP UM	3.57	15.73	58.81

TABLE 3: *Times obtained as average over 15 runs of the OpenMP implementations with SoA layout on all image sizes. Parallel implementations are done with 12 threads. UM, Unrolling Mask loop.*

Version	480p	Full HD	4k
Sequential UM	2.24	1.23	1.68
OMP Naive	5.29	4.55	5
OMP UM	8.93	7.82	8.04

TABLE 4: *Speedup with respect to SoA sequential version obtained from the OpenMP implementations with SoA layout on all image sizes. Parallel implementations are done with 12 threads. UM, Unrolling Mask loop.*

Figure 7 shows the comparison in terms of speedup of AoS and SoA layouts implementations with respect to the sequential AoS implementation. We chose to compare implementations equals in term of loops unrolling and that differ only in term of data layout. We can see that using the **SoA layout** results in better performance overall in terms of speedup compared to the AoS layout. This, as we said before, is because the SoA layout allows a more efficient cache utilization.

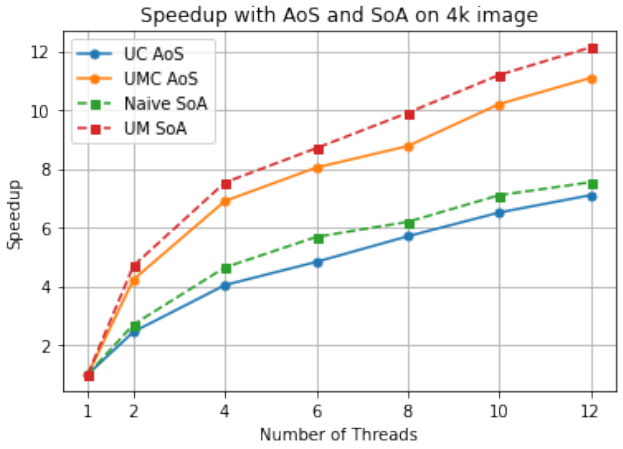

FIGURE 7: *Speedup obtained from the implementations with SoA and AoS layouts on 4k image. The speedup is calculated with respect to the AoS sequential implementation. UC, Unrolling Channels loop. UMC, Unrolling Mask and Channels loop. UM, Unrolling Mask loop.*

8.2. CUDA

Now we show the results obtained from CUDA implementations.

From *figure 8*, we see the trend of GPU times **varying the tile width**. Increasing the tile width, up to 32 (because 32x32 = 1024, the maximum number of threads that a block can contain in CUDA), we obtain better results. If not otherwise indicated, for all the subsequent experiments the tile width of the tiling implementation will be set to 32 as it is the configuration that gives the best times.

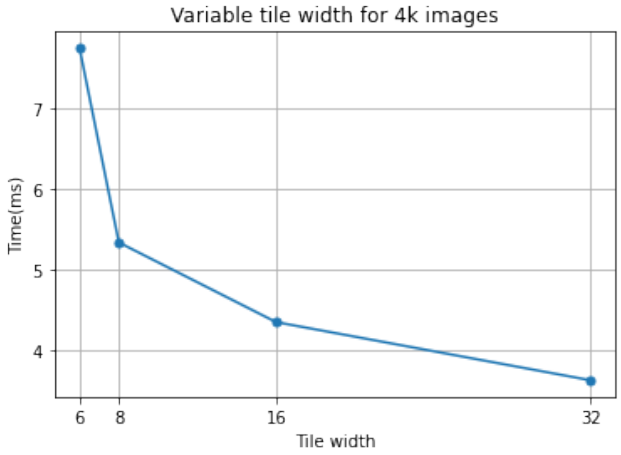

FIGURE 8: *Computation times (in milliseconds) varying the tile width for 4k image. Times taken as the mean over 15 runs of the CUDA algorithm*

In *table 6* we **compare the naive and the tiling implementations** with the speedup calculated between them. We can see the benefit of exploiting the shared memory with the tiling technique. The times reported don't include the data transfer between host and device.

In *table 7* we show the comparison between the **sequential SoA implementation and the tiling SoA implementation** with the respective speedup on all image sizes. We can see that the GPU speedup is much higher than the one obtained from the CPU implementation with OpenMP.

Using the GPU, differently from the CPU, requires to **transfer data** from the host to the device where the computation will happen and vice versa for the results. Therefore, to make a complete comparison we must **include the time to perform the data transfer**. From *table 8* we can see that the data transfer requires much more time compared to only performing the computation in the device. In the case of 4k image the complete cycle with data transfer and computation requires **29 times more time** than perform only the computation on the data.

As we have seen in a previous section regarding **asynchronous loading**, we can improve the performance of the

Number of threads	Naive AoS UC(ms)	Naive SoA(ms)	UMC AoS(ms)	UM SoA(ms)
2	291.28	264.51	168.73	151.64
4	176.43	153.71	103.42	94.78
6	147.72	125.67	88.74	81.97
8	125.07	115.25	81.32	72.06
10	109.62	100.44	69.92	63.72
12	100.47	94.54	64.32	58.81

TABLE 5: *Times (in milliseconds) obtained from the OpenMP implementations with SoA and AoS layouts on 4k image. UC, Unrolling Channels loop. UMC, Unrolling Mask and Channels loop. UM, Unrolling Mask loop.*

Image	Naive(ms)	Tiling(ms)	Speedup
480p	0.55	0.41	1.33
Full HD	2.52	1.92	1.31
4k	4.43	3.62	1.22

TABLE 6: *"Computation only" confrontation between CUDA naive and tiling implementation for all image sizes with tile width of 32, SoA layout on 4k image. The speedup is computed between Naive and Tiling version*

Image	Sequential(ms)	Tiling(ms)	Speedup
480p	53.89	0.41	131.25
Full HD	191.11	1.92	99.33
4k	472.53	3.62	130.39

TABLE 7: *Comparison between sequential and CUDA "computation only" tiling implementations with SoA layout for all image sizes. Tiling version with tile width of 32. The results are the average over 15 runs of the algorithm.*

Image	Tiling DT(ms)	Tiling CO(ms)
480p	79.61	0.41
Full HD	88.14	1.92
4k	103.84	3.62

TABLE 8: *Times comparison between CUDA with data transfer (DT) synchronous and CUDA "computation only" (CO) tiling implementations with SoA layout for all image sizes. Tiling version with tile width of 32. The results are the average over 15 runs of the algorithm.*

CUDA implementations by performing the data transfer on more streams.

Assuming an image with three channels and using the SoA layout, we can define **one stream per each image channel**. This way, while a stream is loading its chunk of data, another, that has already completed its loading, can perform the computation. We obtain an **overlap of the data transfer and the kernel execution**.

We compare the times of the synchronous and the asynchronous data transfer implementations and their speedup with respect to the sequential SoA implementation in *table 9*. We can notice that the tiling implementation with the asynchronous data transfer is **1.46 times faster than with the synchronous transfer**.

Version	Time(ms)	Speedup
Sequential	472.53	-
CUDA Naive Sync	104.85	4.50
CUDA Tiling Sync	103.84	4.55
CUDA Naive Async	72.37	6.53
CUDA Tiling Async	71.08	6.65

TABLE 9: *Effect of asynchronous data loading on CUDA tiling implementations with tile width of 32, SoA layout on 4k image. The speedup is computed with respect to the sequential SoA version*

In *figure 9*, we compare the results, in terms of speedup over the sequential AoS implementation, obtained from the CUDA implementations with AoS and SoA layouts without considering the data transfer. We can see that the **coalesced memory access** obtained thanks to the **SoA data layout** improves the performance of the algorithm very much. We can obtain a **speedup up to 196** with respect to the AoS sequential implementation on 4k image.

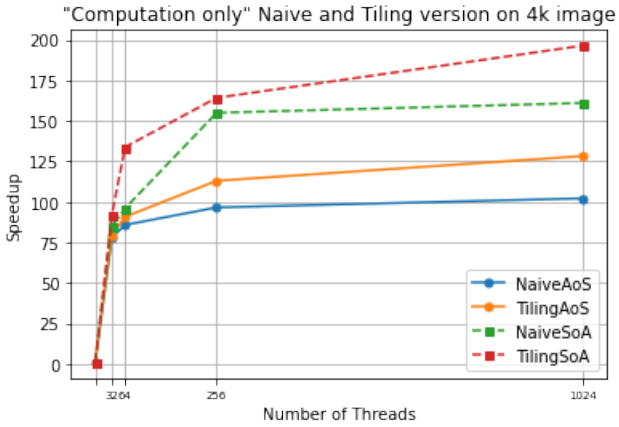

FIGURE 9: *Speedup without considering the data transfer with the device between the naive and tiling implementations with AoS and SoA layouts on 4k image varying the number of threads per block.*

9. Conclusions

As we have seen until now, programming using threading frameworks for CPUs and GPUs is quite different.

With GPUs we have to make the most of the hardware of the device, exploiting all the different levels of **memories** using the techniques we have explained earlier, making sure to have a **coalesced memory access** and being careful to **map threads** of the grid over the input or output data.

We have seen the **enormous speedup** that can be obtained using the GPU to perform **embarrassingly parallel** computations.

Unlike CPUs, the GPU requires to **transfer data** between host and device because the hardware is not meant for code with much control logic but only for **massive computation**. Therefore, we have to perform on the host sections of code with control logic and only when we have to perform heavy computations we use the device to execute CUDA kernels.

As we have seen, in our case the data transfer takes much more time than the computation of the kernels we need to perform on the device, but in other situations, like with neural networks and other applications, this may not always be the case; with the time required for the data transfer negligible with respect to the computation.

We have seen that the data transfer can be improved with the **asynchronous mechanism** of CUDA: **the streams**. Thanks to them we can **overlap** the transfer data between host and device and the execution of a CUDA kernel.

All of the results shown in this paper are obtained from a mobile GPU that has a limited amount of stream multiprocessors and allows for only one data transfer at a time, even on different streams.

An important aspect regarding GPUs is that we can obtain better performance by simply **employing a more performing GPU hardware**, without changing anything in the code. In fact, with better hardware, we could run more blocks simultaneously in different stream multiprocessors, we could have larger shared memories and we could also perform data transfer simultaneously in different streams.

With CPU's implicit parallel frameworks we have seen how easy is to **convert** a sequential version of an algorithm into a parallel one. With **OpenMP** we only need a compiler directive before a loop to obtain good speedup.

We have seen how we can drastically improve the performance of the algorithms at the cost of **losing generality** over the inputs.

We have seen how we can obtain performance improvements, even without a threading framework, by exploiting the **hardware parallel mechanisms** of the CPU when we have simple loops with simple vector computations and without pointers aliasing.

Performing **manual loop unrolling** together with a threading framework can result in more than linear speedups varying the number of threads as we have seen in some experiments.

Finally, we have seen that even with CPUs, using a cache friendlier data organization in memory, the structure of arrays layout, helps **vectorization** and allows an **efficient** use of the cache.

10. Resources

The code is available on **GitHub**:

- Sequential and parallel CPU with OpenMP: `https://github.com/GiovanniBurbi/kernel_image_processing`
- parallel GPU with CUDA: `https://github.com/GiovanniBurbi/kernel_image_processing_CUDA`

11. bibliography

1. CUDA: https://developer.nvidia.com/cuda-zone
2. OpenMP: https://www.openmp.org/
3. https://en.wikipedia.org/wiki/Kernel_(image_processing)
4. https://setosa.io/ev/image-kernels/
5. https://on-demand.gputechconf.com/gtc/2014/presentations/S4158-cuda-streams-best-practices-common-pitfalls.pdf
6. https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/
7. https://developer.nvidia.com/blog/gpu-pro-tip-cuda-7-streams-simplify-concurrency/
8. https://leimao.github.io/blog/CUDA-Stream/