Università di Pisa

Computer Engineering

Computer Architecture

Parallelized Nearest Neighbor Upscaler

Group Project Report

TEAM MEMBERS:
Biagio Cornacchia
Gianluca Gemini
Matteo Abaterusso

Academic Year: 2021/2022

Contents

1 Introduction

The aim of this project is to develop an **image upscaler** taking advantage of the parallelization capabilities of the modern **CPU** and **GPU**. Image upscaling is the process that allows to increase the resolution of an image, trying to minimize the loss in image quality. There are several possible algorithms and the one chosen for the project is the **nearest neighbor interpolation**. The nearest neighbor is the simplest upscaling method in which each pixel in the upscaled image is assigned the value of its nearest neighbor in the original image. The advantages of this method are that it is simple and fast but, on the other hand, it tends to produce pixelated results.

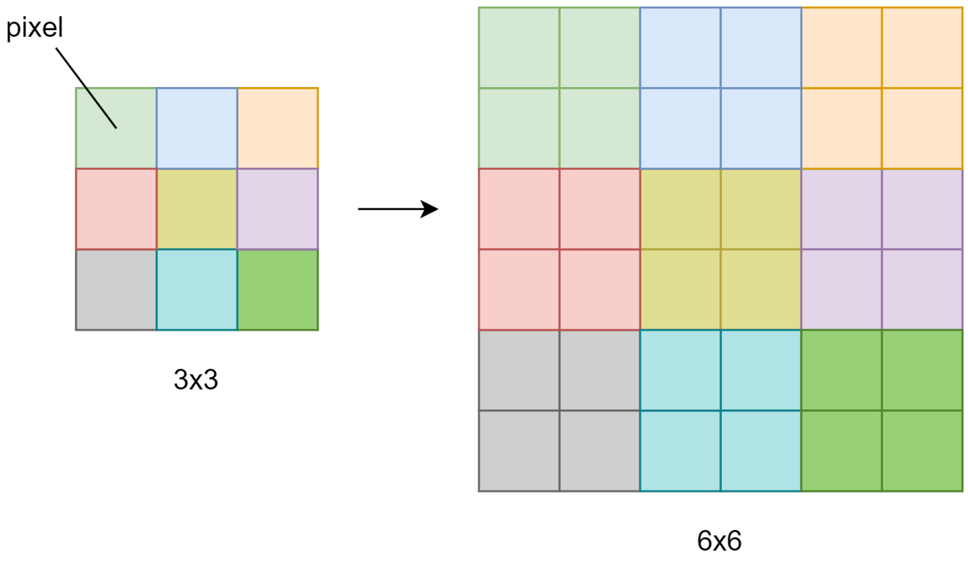


Figure 1: Example of Nearest Neighbor upscaler.

Since each pixel of the upscaled image only depends on one pixel of the original image, all the pixels of the upscaled image are **independent** from each other thus they can be computed simultaneously. This characteristic makes it possible for the algorithm to benefit from **parallelization**.

2 CPU Version

In this section will be analyzed the implementation and the performances of the algorithm for the CPU.

2.1 Implementation

To achieve parallelization, the C++11 built-in support library for threads has been used. The code executed by each thread is the following:

```cpp
void worker(const uint32_t* originalImage, uint32_t* upscaledImage,
        uint32_t start, uint32_t stop, uint8_t upscaleFactor, size_t width)
{
    uint32_t upscaledWidth = width * upscaleFactor;

    // iterate the pixels of the original image assigned to this thread
    for (size_t oldIndex = start; oldIndex < stop; oldIndex++) {
        // convert the position in a matrix notation
        uint32_t i = oldIndex / (width);
        uint32_t j = oldIndex - (i * width);

        // compute the position of the first pixel to duplicate
        // in upscaled image
        uint32_t newi = i * upscaleFactor;
        uint32_t newj = j * upscaleFactor;

        // iterate the pixel to duplicate in upscaled image
        for (int m = newi; m < newi + upscaleFactor; m++) {
            for (int n = newj; n < newj + upscaleFactor; n++) {
                // compute the pixel position in the upscaled image vector
                uint32_t newIndex = m * upscaledWidth + n;

                // copy all the channels
                upscaledImage[newIndex] = originalImage[oldIndex];
            }
        }
    }
}
```

The image to upscale (*originalImage*) is loaded into a **one-dimensional byte array**. Each pixel is represented by 4 byte (32 bit) which correspond to the 4 channels *red*, *green*, *blue* and *alpha* (transparency). Therefore, through a static cast to *uint32_t* array, each array element represents a pixel of the image.

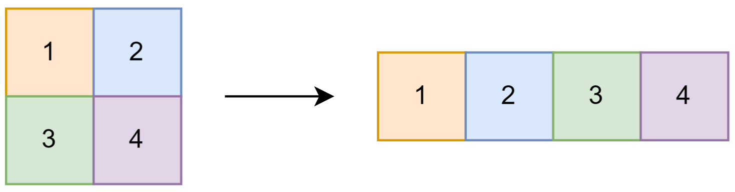


Figure 2: Example of representation of an image in memory.

The range of pixels of the original image that the thread must handle is represented by the two arguments *start* and *stop*. Each of these pixels is copied into the upscaled image, and the number of copies is defined by the **upscaling factor**.

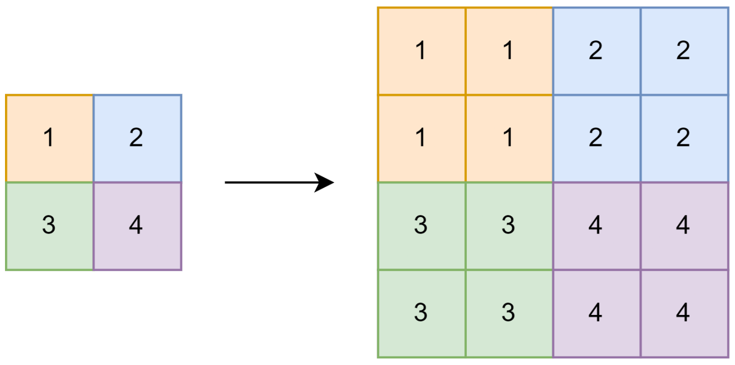


Figure 3: Example of an upscaler result.

2.2 Tests and Results

The tests have been performed on a Ryzen 7 5800H with 8 cores. The considered metrics are the **execution time** (in milliseconds) and the relative **speed-up** varying the number of scheduled threads. To sample the execution times, the *chrono* build-in library has been used. For each case study, 30 repetitions have been carried out, over which the mean and 95% confidence interval has been calculated. Regarding the speed-up, both **cumulative** and **step** has been computed. The tests have been repeated for different original image size and with an upscale factor equal to 2.

The first test has been carried out using an image of 854x480 pixels. The results obtained are the following:

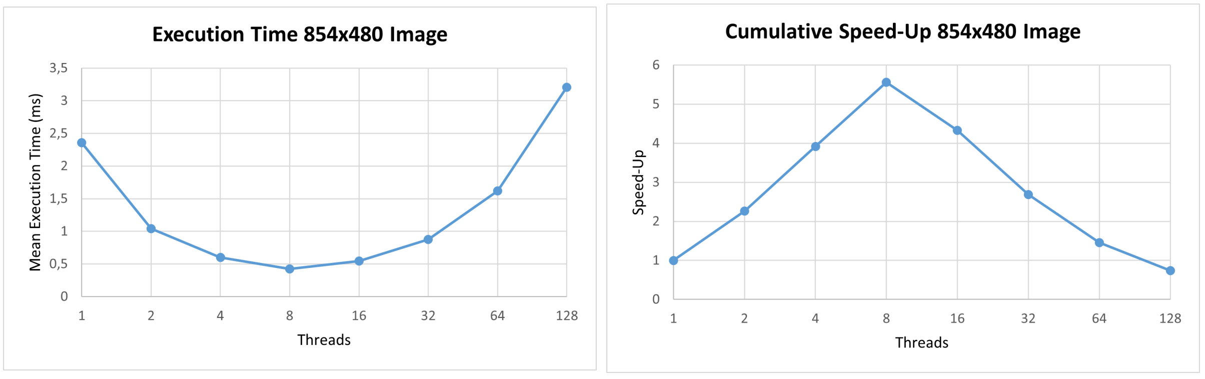


Figure 4: Test results for 854x480 pixels image.

Using a number of threads equal to the number of physical cores of the CPU, a speed-up between 5x and 6x is achieved. If the number of physical cores is exceeded, the benefits due to parallelism **decrease** because of the overhead due to **time-sharing**.

The second test has been carried out using an image of 1280x720 pixels. The results obtained are the following:

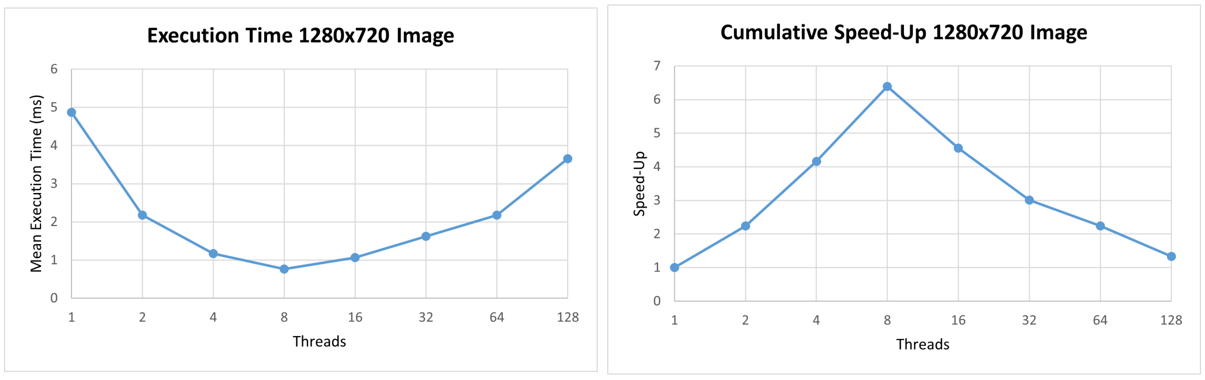


Figure 5: Test results for 1280x720 pixels image.

So, **doubling** the number of pixels with respect to the previous case, it has been obtained the **same behavior** with a maximum speed-up greater than 6x, exploiting all the CPU physical cores.

The final test has been carried out using an image of 1920x1080 pixels. The results obtained are the following:

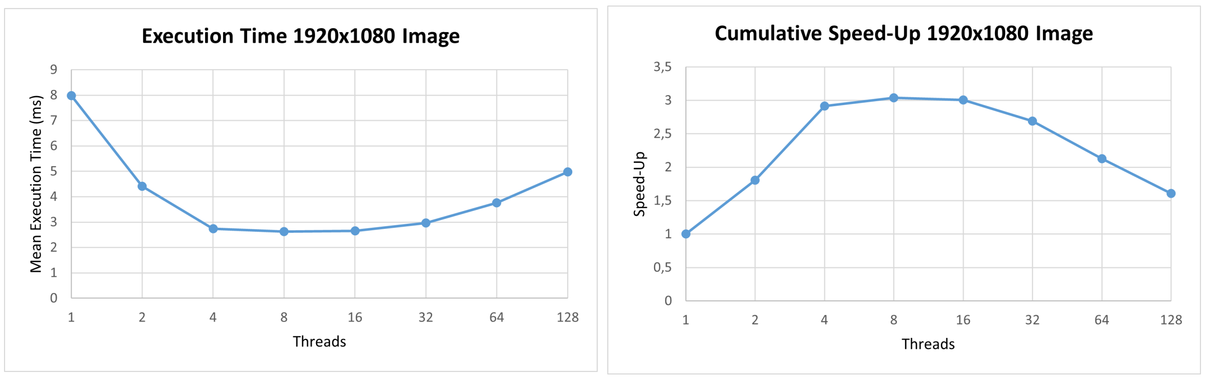
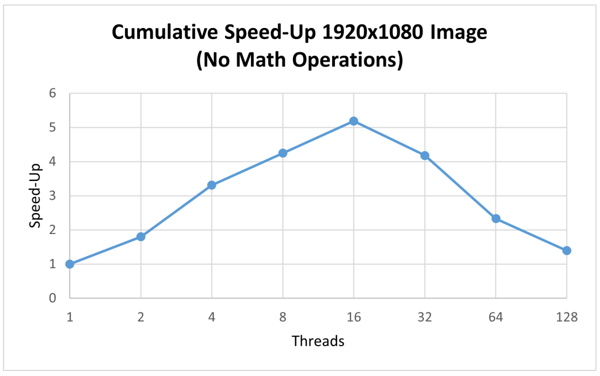
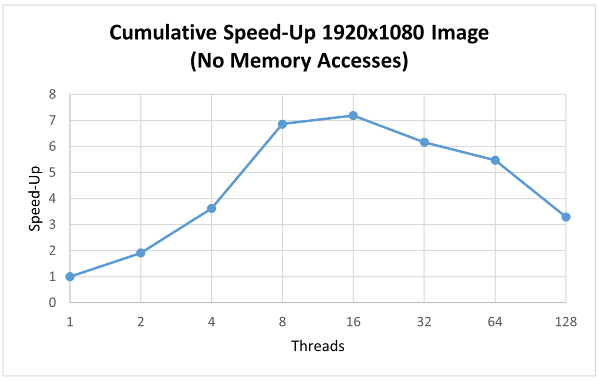


Figure 6: Test results for 1920x1080 pixels image.

Doubling again the number of pixels, the algorithm reached a **saturation point**. Indeed, using 8 threads the algorithm performs as if it were using 4 threads. The algorithm consists of a **CPU bound** part where the indexes are computed and an **I/O bound** part where read and write operations are performed. To figure out which of the two parts causes the saturation, tests have been repeated excluding one of the two parties in turn.





The results show that the **excessive** number of **memory accesses** per thread causes saturation. Therefore, in order to improve the performances, it is necessary having **more real parallel threads**.

3 GPU Version

In this section will be analyzed the implementation and the performances of the algorithm for the GPU, using CUDA. CUDA allows software to exploit the high number of NVIDIA graphics card CUDA cores achieving an high level of parallelism.

3.1 Implementation

The code executed by each CUDA core is the following:

```
__global__ void upscale(cudaTextureObject_t originalImage,
        uint8_t* upscaledImage, uint32_t pixelsHandledByThread, uint32_t width,
        uint32_t height, uint8_t bytePerPixel, uint32_t upscaleFactor)
{
    uint32_t pixelsHandledByBlock = pixelsHandledByThread * blockDim.x;
    uint32_t startNewIndex = blockIdx.x * pixelsHandledByBlock +
        + threadIdx.x * pixelsHandledByThread;
    uint32_t upscaledWidth = width * upscaleFactor;
    uint32_t upscaledSize = width * height * upscaleFactor * upscaleFactor;

    // iterate all pixels handled by this thread
    for (uint32_t i = 0; i < pixelsHandledByThread; i++) {
        // compute the coordinates of the pixel
        uint32_t newIndex = startNewIndex + i;

        if (newIndex < upscaledSize) {
            uint32_t x = newIndex / upscaledWidth;
            uint32_t y = newIndex - (x * upscaledWidth);

            // compute the coordinates of the pixel of the original image
            uint32_t oldX = x / upscaleFactor;
            uint32_t oldY = y / upscaleFactor;

            // copy the pixel
            uchar4  pixelToCopy = tex2D<uchar4>(originalImage, oldY, oldX);
            memcpy(&upscaledImage[newIndex * bytePerPixel],
                    &pixelToCopy, sizeof(uchar4));
        }
    }
}
```

The approach used in this implementation is slightly different from the CPU one. In particular, the kernel **iterates** the **upscaled image pixels** instead of the original ones. This increases the number of total threads, but it allows to better manage the **memory locality** and so the cache hit rate.

Basically, original and upscaled image can be both loaded into the GPU global memory. In this case, each kernel must read from the former and write to the latter. However, this access pattern continually causes **jumps** to **potentially distant locations** in memory, not allowing proper utilization of the cache. To avoid this problem, the original image is loaded into the **texture memory**, a read-only memory with a dedicated cache.

Differently from the CPU version, the pixels that a thread must handle are not decided a priori, but they are obtained using the *thread ID* and the *block ID* of each thread. So, the argument *pixelsHandledByThread* tells to the thread how many pixels it must handle.

3.2 Tests and Results

The tests have been performed on a NVIDIA RTX 3060 mobile with 30 multi-processors (MP) and 128 CUDA cores per MP. The warp size is 32, so each multi-processor has 4 partitions and the total of CUDA cores is 3840. Also in this case, 30 repetitions have been carried out for each case study and the considered metrics are the **execution time** and the **speed-up** (cumulative and step).

CUDA allows to define different kernel configurations, specifying the **block** and **grid size**. The upscaler has been tested using 32, 64 and 128 threads per block since 32 is the warp size and 128 is the CUDA cores associated with a multi-processor. For each of these configurations, *pixelsHandledByThread* has been made to vary to test **different levels** of **parallelism**. Indeed, the lower is the number of pixels handled and the more is the number of total threads.

The first test has been carried out using an image of 854x480 pixels. The results obtained are the following:

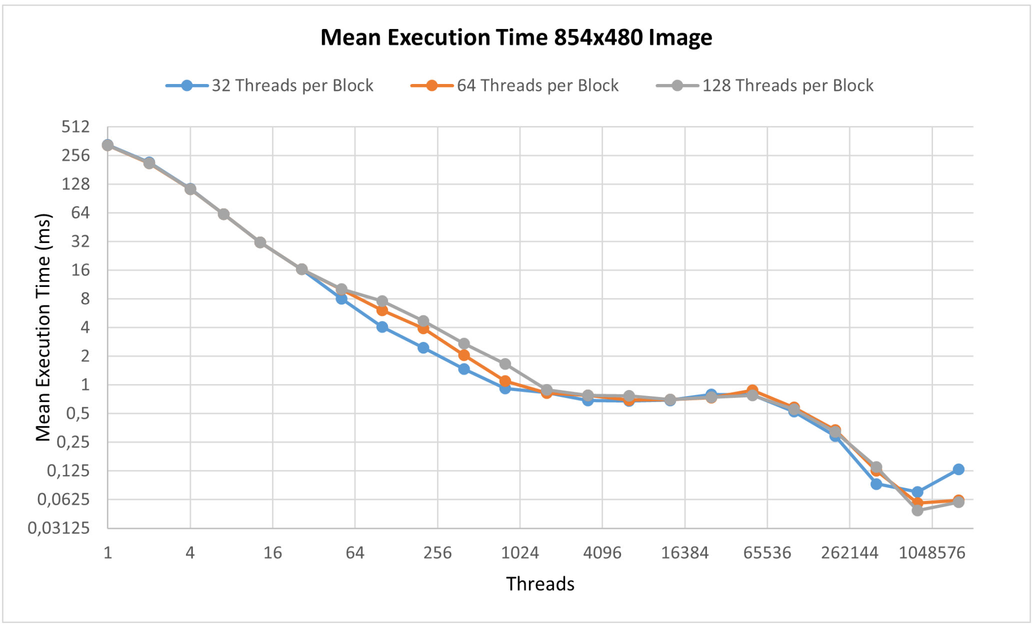


Figure 7: Execution time resulting from the test.

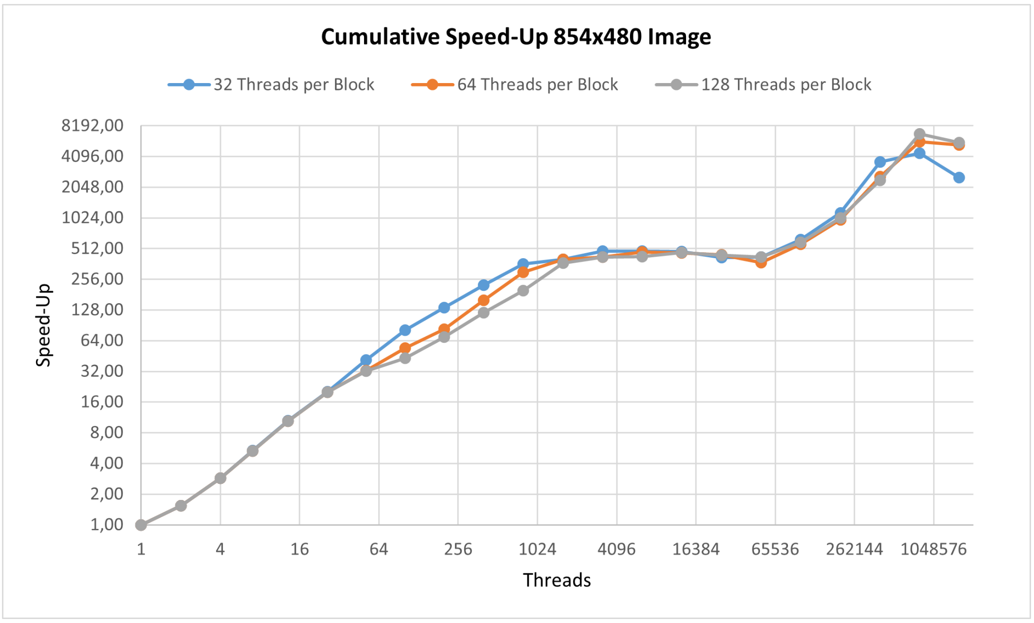


Figure 8: Cumulative speed-up resulting from the test.

The cumulative speed-up shows that after 512 threads, a **saturation point** is reached. To figure out precisely when the saturation occurs, the step speed-up graph has been analyzed.

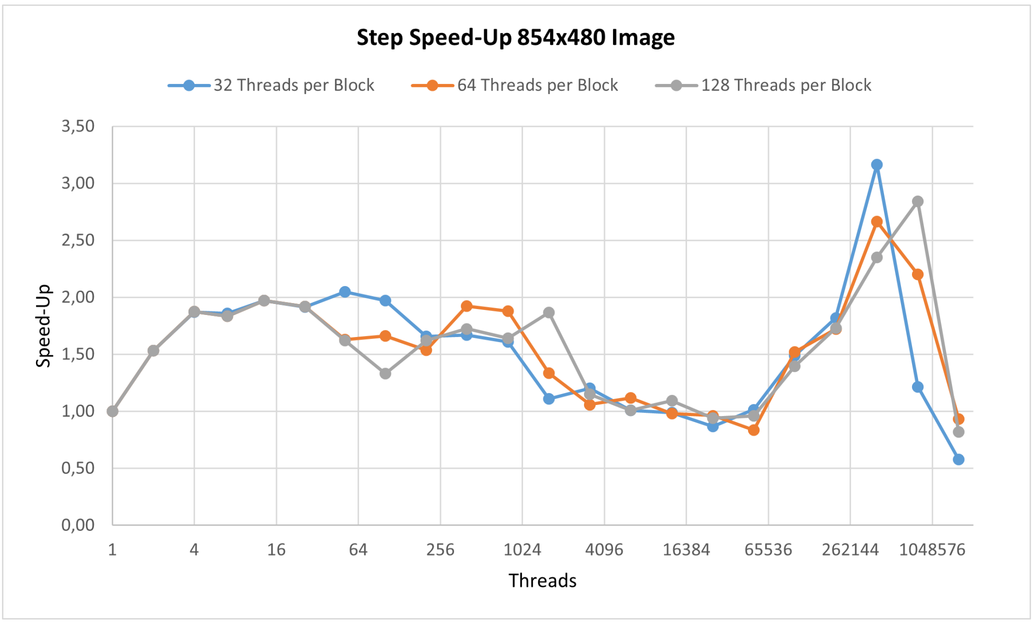


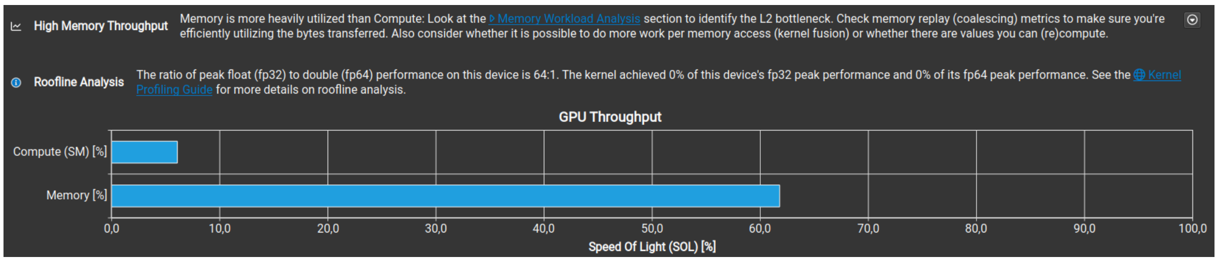
Figure 9: Step speed-up resulting from the test.

From this graph, it is possible to see that the saturation occurs for all block configurations around 4000 threads, which corresponds to the time when the system goes into **time-sharing**. In order to understand the problem and how to solve it, more in-depth analyses are needed.

3.3 Performance Analysis

A detailed analysis has been obtained by exploiting the *Nvidia Nsight Compute* profiler. The upscaler has been compiled in *release mode*, and the configuration used is 128 threads per block, 128 managed pixels per thread, and the input is an image of 854x480 pixels.

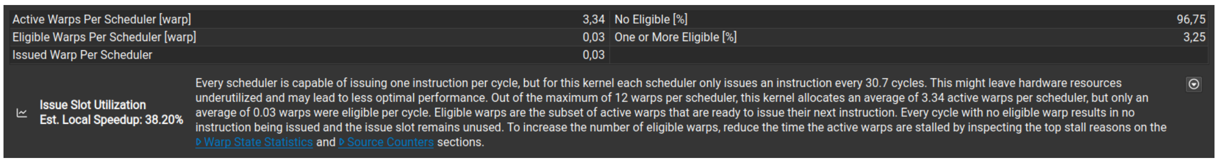
The first metric that has been analyzed is the **balance** between **compute** and **memory throughput**.


High Memory Throughput: Memory is more heavily utilized than Compute: Look at the Memory Workload Analysis section to identify the L2 bottleneck. Check memory replay (coalescing) metrics to make sure you're efficiently utilizing the bytes transferred. Also consider whether it is possible to do more work per memory access (kernel fusion) or whether there are values you can (re)compute.

Roofline Analysis: The ratio of peak float (fp32) to double (fp64) performance on this device is 64:1. The kernel achieved 0% of this device's fp32 peak performance and 0% of its fp64 peak performance. See the Kernel Profiling Guide for more details on roofline analysis.

As shown in the picture above, the memory throughput is very high, whereas the compute throughput is very low. This situation can lead to **latency problem**.

Another important metric to consider is the **percentage** of **no eligible**, that represents the number of cycles in which no instructions have been issued.

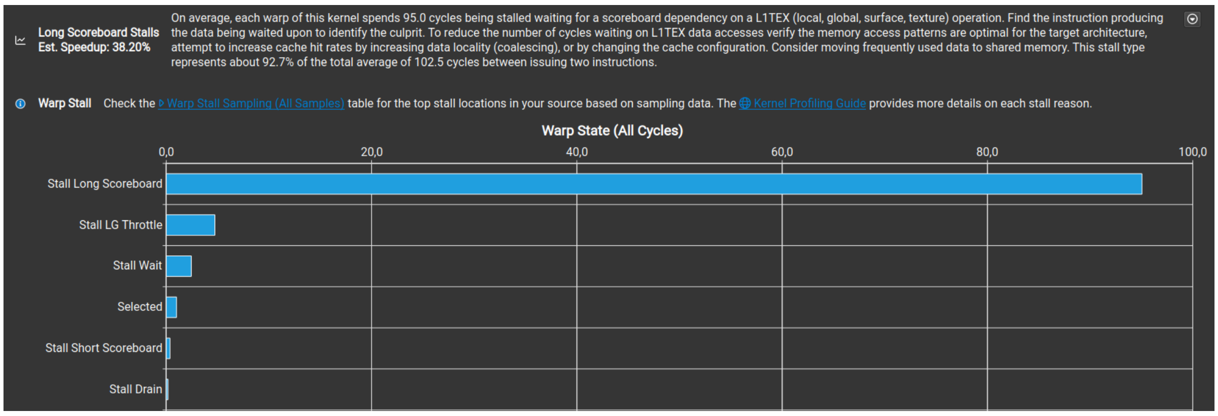


Active Warps Per Scheduler [warp]	3,34	No Eligible [%]	96,75
Eligible Warps Per Scheduler [warp]	0,03	One or More Eligible [%]	3,25
Issued Warp Per Scheduler	0,03		

Issue Slot Utilization
Est. Local Speedup: 38.20%
Every scheduler is capable of issuing one instruction per cycle, but for this kernel each scheduler only issues an instruction every 30.7 cycles. This might leave hardware resources underutilized and may lead to less optimal performance. Out of the maximum of 12 warps per scheduler, this kernel allocates an average of 3.34 active warps per scheduler, but only an average of 0.03 warps were eligible per cycle. Eligible warps are the subset of active warps that are ready to issue their next instruction. Every cycle with no eligible warp results in no instruction being issued and the issue slot remains unused. To increase the number of eligible warps, reduce the time the active warps are stalled by inspecting the top stall reasons on the Warp State Statistics and Source Counters sections.

Results show that for more than 96% of the cycles, no instructions were executed.

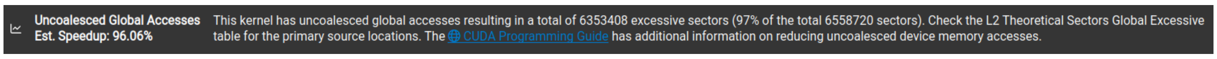
Afterwards, **warp state statistics** have been analyzed, that show the states in which all warps spent cycles during the kernel execution.


Long Scoreboard Stalls
Est. Speedup: 38.20%
On average, each warp of this kernel spends 95.0 cycles being stalled waiting for a scoreboard dependency on a L1TEX (local, global, surface, texture) operation. Find the instruction producing the data being waited upon to identify the culprit. To reduce the number of cycles waiting on L1TEX data accesses verify the memory access patterns are optimal for the target architecture, attempt to increase cache hit rates by increasing data locality (coalescing), or by changing the cache configuration. Consider moving frequently used data to shared memory. This stall type represents about 92.7% of the total average of 102.5 cycles between issuing two instructions.

Warp Stall: Check the Warp Stall Sampling (All Samples) table for the top stall locations in your source based on sampling data. The Kernel Profiling Guide provides more details on each stall reason.

On average, each warp of this kernel spends almost 100 cycles being stalled waiting for a **scoreboard dependency** on a local, global, surface or texture memory operation.

Finally, **source counters** have been analyzed.



Uncoalesced Global Accesses
Est. Speedup: 96.06%

This kernel has uncoalesced global accesses resulting in a total of 6353408 excessive sectors (97% of the total 6558720 sectors). Check the L2 Theoretical Sectors Global Excessive table for the primary source locations. The CUDA Programming Guide has additional information on reducing uncoalesced device memory accesses.

Can be seen how the **uncoalesced global accesses problem** causes an excessive number of sectors to be accessed. Considering that the profiling results show that most of the problems are due to memory, solving uncoalesced global accesses might be a good starting point for solving saturation. In fact, the speed up achievable by solving this problem according to the profiler is more than 95%.

4 GPU Optimized Version

In this section will be analyzed how the uncoalesced global accesses problem has been solved and the impact it had on performance.

4.1 Solution

Uncoalesced global accesses occur if threads belonging to the same warp perform a memory operation at the **same instant** at addresses that are **not consecutive**. The upscaler analyzed in the previous section behaves as shown below:

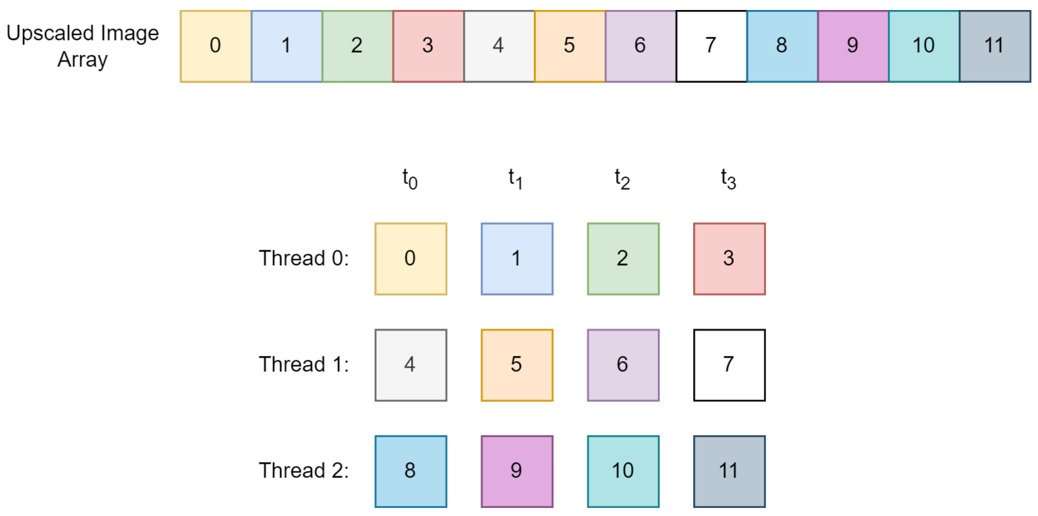


Figure 10: Example of uncoalesced global accesses.

It is possible to see that at the instant t_0 the three threads access indexes 0, 4, and 8, respectively, which are **not contiguous**. In order to achieve coalesced global accesses the upscaler should behave as shown below:

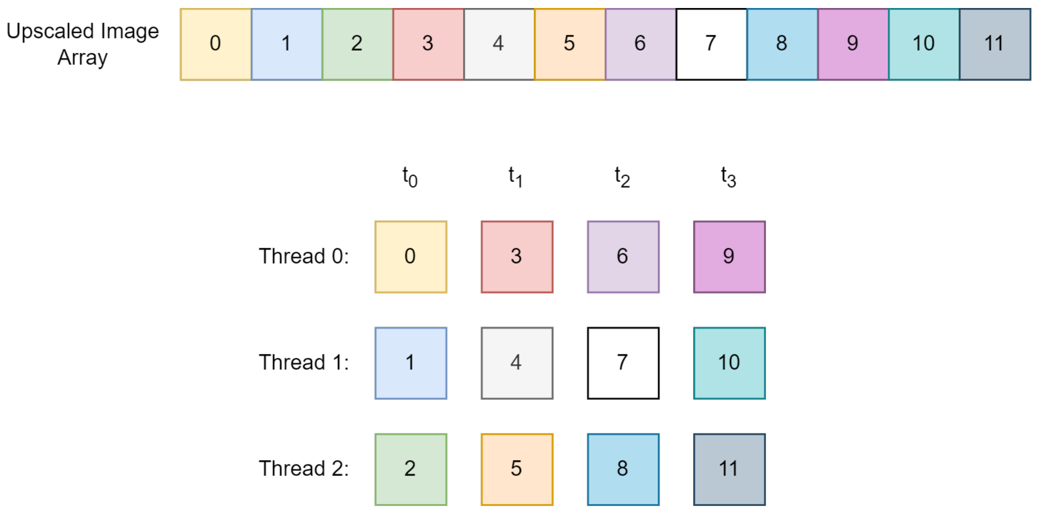


Figure 11: Example of coalesced global accesses.

4.2 Implementation

The optimized kernel code is the following:

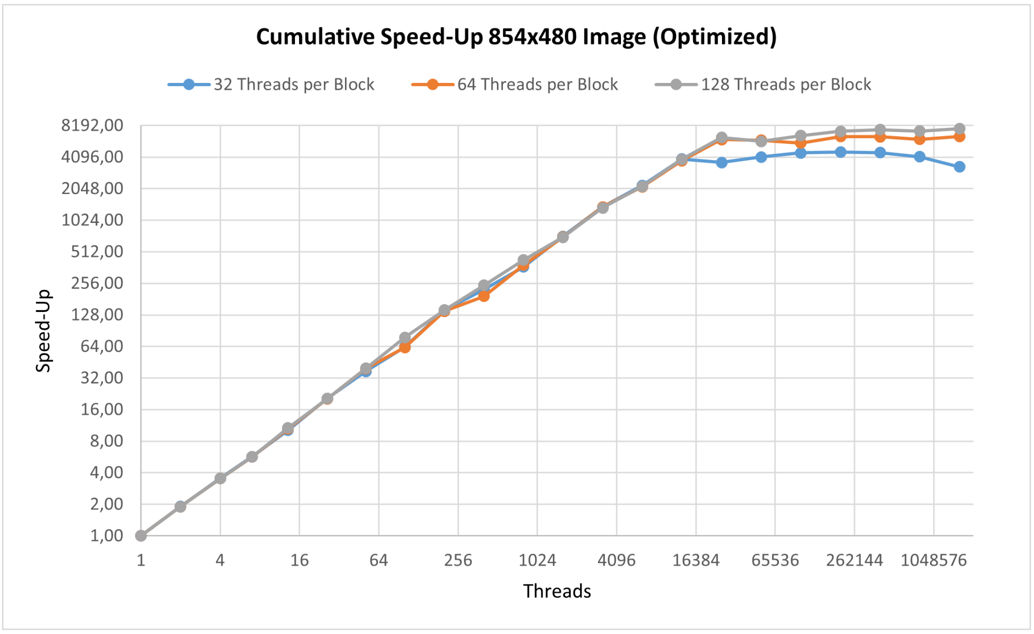
```
__global__ void upscaleOptimized(cudaTextureObject_t originalImage,
        uchar4* upscaledImage)
{
    uint32_t startNewIndex = blockIdx.x * blockDim.x + threadIdx.x;

    // iterate all pixels handled by this thread
    for (uint32_t i = 0; i < devicePixelsHandledByThread; i++) {
        // compute the coordinates of the pixel
        uint32_t newIndex = startNewIndex + (i * deviceThreadsCount);

        if (newIndex < deviceUpscaledSize) {
            uint32_t x = newIndex / deviceUpscaledWidth;
            uint32_t y = newIndex - (x * deviceUpscaledWidth);

            // compute the coordinates of the pixel of the original image
            uint32_t oldX = x / deviceUpscaleFactor;
            uint32_t oldY = y / deviceUpscaleFactor;

            // copy the pixel
            uchar4 pixelToCopy = tex2D<uchar4>(originalImage, oldY, oldX);
            upscaledImage[newIndex] = pixelToCopy;
        }
    }
}
```

In order to solve uncoalesced global accesses, the **indexes system** has been modified. In addition, two further optimizations have been implemented:

- The original kernel calculates read-only used variables for each thread, the value of which remains the same for all threads. Therefore, in the optimized version it has been decided to **compute** these variables **a priori** and load them into **constant memory**. This reduces the mathematical operations of each thread and the access time to the values.
- The *memcpy* operation in the original kernel corresponds to 4 memory accesses. Therefore, in the optimized version, the CUDA built-in structure *uchar4* has been exploited to ensure that copying a pixel (which corresponds to 4 bytes) is done through a **single memory transaction**.

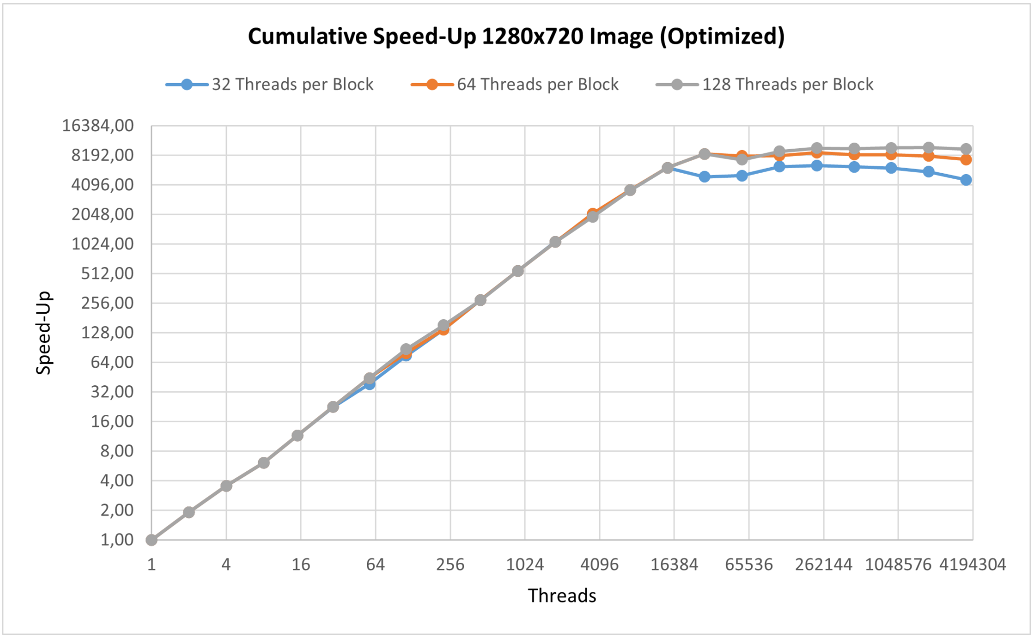
4.3 Tests and Results

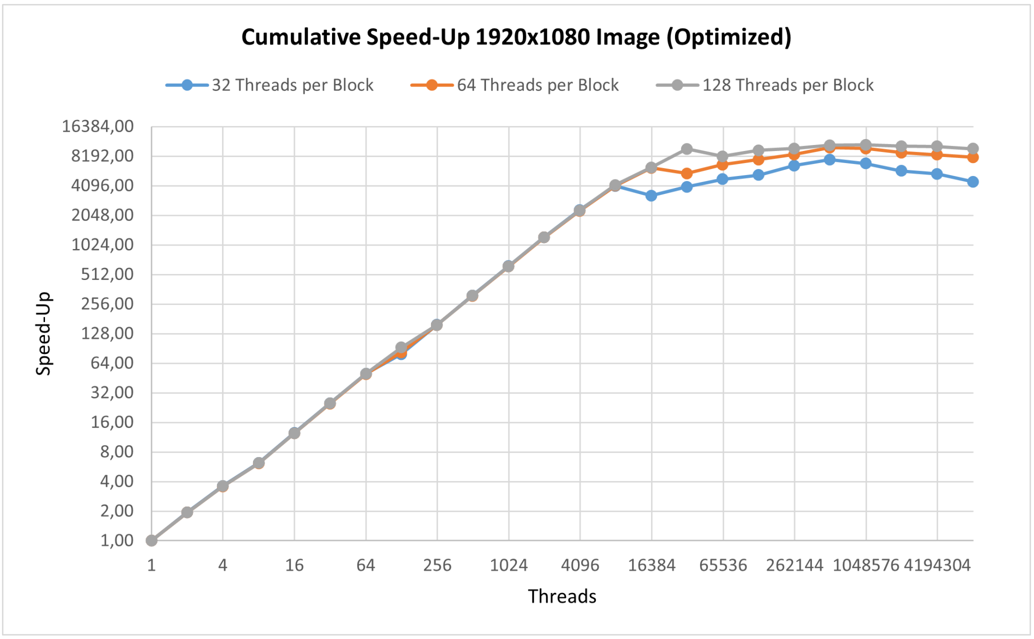
To compare the new kernel with the previous one, the same test with an image of 854x480 pixels, 32/64/128 threads per block, and 128 pixels handled by thread has been performed.



In the previous solution (figure 3.2), saturation occurred around 4000 threads. With this new solution, saturation occurs above 20000 threads. In particular, the step speed up is always greater than 1.5x, making it **reasonable** to use **more resources** until saturation.

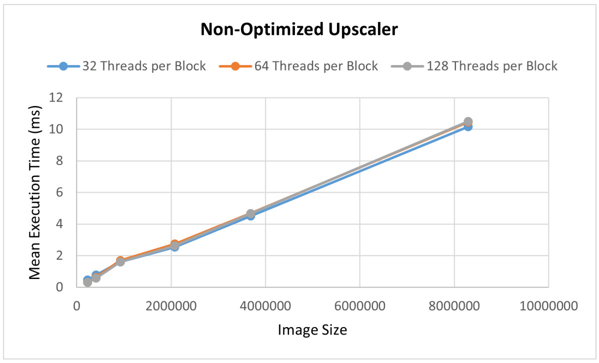
The same test has been carried out using larger images. The results are reported below.

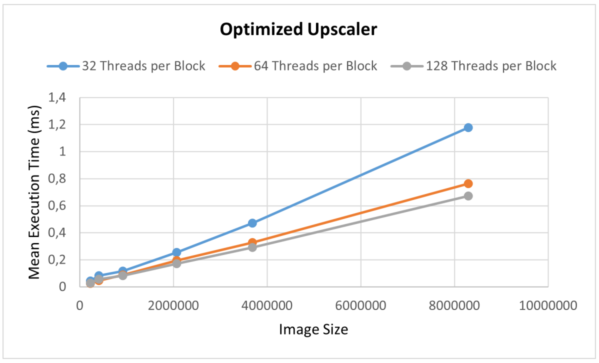




The application behaviour remains approximately the same varying the image size. More precisely, **saturation** tends to **occur later** as image size increases.

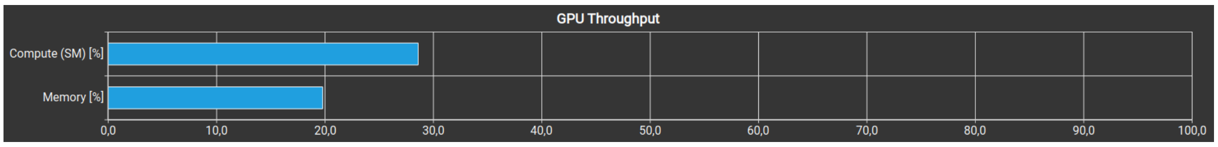
To test whether the optimized algorithm has **better scalability**, a test has been performed in which the number of pixels handled by each thread is fixed and the size of the input images is varied. Then, each thread will always perform the **same number** of **memory accesses** whereas the input load is varied.



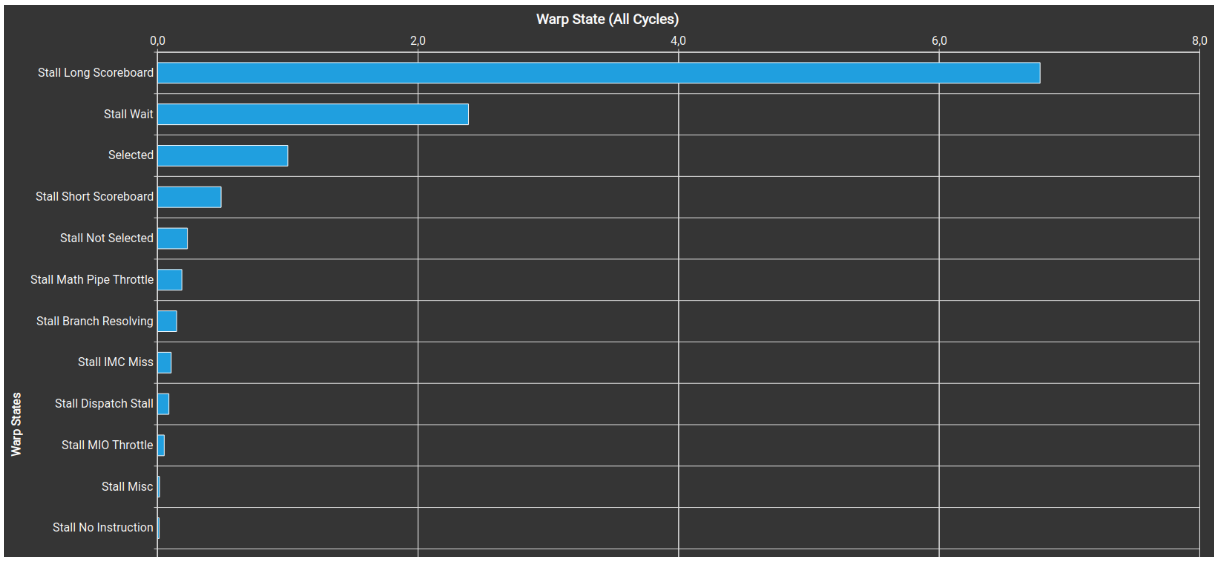


As the results show, the new algorithm **scales much better**. Indeed, in the highest load case, the execution time is **lower** by an **order** of **magnitude** than the original algorithm.

To conclude, an analysis with the profiler has been performed. As in section 3.3, the optimized upscaler has been compiled in *release mode*, and the configuration used is 128 threads per block, 128 managed pixels per thread, and the input is an image of 854x480 pixels.



The picture above shows that the compute and memory throughput are **correctly balanced**.



Instead, this final picture shows that the number of cycles in which warps stall are **reduced** by an **order** of **magnitude**.