Real time denoising on Gpu

Adrien Vannson
ENS de Lyon

Internship supervised by:
Johannes Hanika
Karlsruhe Institute of Technology

8 May 2023 – 4 August 2023

Contents

1 Introduction 3

2 Denoising images 3

2.1 Rasterization, ray tracing and path tracing 3

2.2 Machine Learning Denoising 4

2.3 Training and evaluation 5

2.3.1 Training data 5

2.3.2 Evaluation 5

3 Introduction to Gpu computing 7

3.1 Architecture of a Gpu 7

3.1.1 Thread hierarchy 7

3.1.2 Memory hierarchy 8

3.2 Hardware effects 9

3.2.1 Accessing the global memory 9

3.2.2 Memory coalescing 9

3.2.3 Bank conflicts 10

3.2.4 Thread divergence 11

4 Matrix multiplication 11

4.1 State of the art 11

4.2 Tensor cores 12

4.3 Matrix multiplication algorithms 13

4.3.1 Naive algorithm 13

4.3.2 Shared memory 14

4.3.3 Cooperative matrices 14

4.3.4 Efficient use of cooperative matrices 14

4.3.5 Performance evaluation 16

5 Fast network inferencing 16

5.1 State of the art 17

5.2 Efficient convolution 17

5.3 Inferencing the network 18

5.4 Performance evaluation 18

6 Conclusion 18

Bibliography 19

A Hardware specification 21

B Performance measurement protocol 21

C Source code 21

1 Introduction

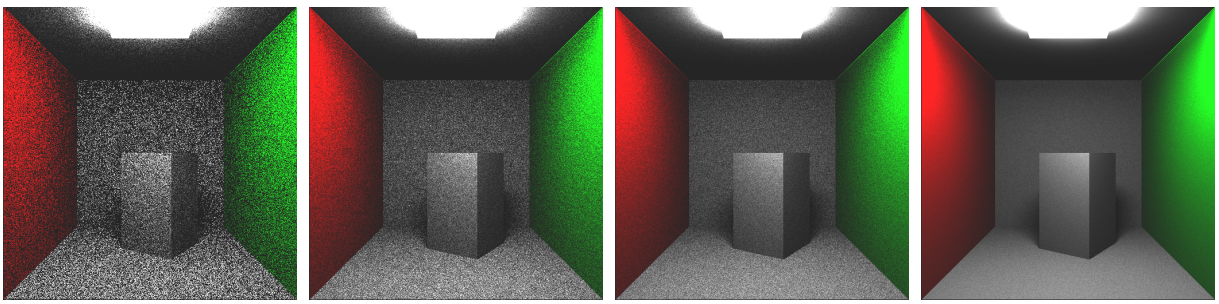
Real-time rendering – the creation of images of 3D scenes in real time – is a particularly important and active domain of computer graphics. It is, for example, useful for video games. Since it requires a lot of computations, it is usually done on a GPU. Historically, this has been mainly realized thanks to rasterization: a simple technique where primitives such as triangles are directly projected on the output image. This method is rather efficient, but presents the major drawback of producing images that may seem unrealistic. On the other hand, ray-tracing and path-tracing are methods that produce more realistic images, but take longer to be executed. While ray-tracing can now be used in real time, it is not yet the case of path tracing, a Monte Carlo algorithm that takes time to converge. When used at real-time rates, it can produce very noisy images, or even images containing almost only noise. Several methods have been suggested to reconstruct the expected image from the noisy output of the path-tracer. One promising method consists of using machine learning to do it. It can work well, but the inference of the neural network is time-consuming. Thankfully, NVIDIA recently released an extension allowing developers to use physical components of the GPU to boost the speed of machine learning tasks. The goal of the internship is to take advantage of this extension to evaluate the output of a neural network quickly, measure the evaluation time and figure out whether this technique could be used at real-time rates.

2 Denoising images

2.1 Rasterization, ray tracing and path tracing

Rasterization, ray tracing and path tracing are different algorithms that can be used to create images of 3D scenes. In short, rasterization is a fast algorithm, but it is quite limited as it can't render correctly every kind of scene. Ray-tracing produces more realistic images, but is slower even if it can now be run at real time rates. The path-tracing algorithm can produce realistic images, but it is way slower. More explanations on these different approaches can be found on NVIDIA's website [1].

The path-tracing algorithm relies on a Monte Carlo method to estimate the color of each pixel. As shown by Figure 1, the algorithm may be slow to converge: here, 10000 samples per pixel are needed to obtain an image with a reasonable quality. Even if the convergence of the algorithm can be improved with various techniques such as using more efficient sampling methods, modern graphic cards can't evaluate in real-time enough samples per pixel to obtain images that do not seem very noisy.



100 samples per pixel | 300 samples per pixel | 900 samples per pixel | 10000 samples per pixel

Figure 1: Convergence of the Monte Carlo renderer

2.2 Machine Learning Denoising

Since path tracing can produce noisy images, research has been made to find methods to reconstruct the expected image from the noisy output of the path-tracer. In 2017, Chaitanya et al. [2] focused on images rendered with a low sample count, and suggested using a machine learning approach to realize the denoising. They used a fully convolutional neural network with the U-Net architecture [3], as shown in Figure 2. The network is given the image and information about the geometry of the scene (for each pixel, the depth, the normal and a property of the material), and produces a denoised version of the image. Neural networks with this architecture are still studied and used in recent papers [4, 5, 6], so I focused on this kind of network.

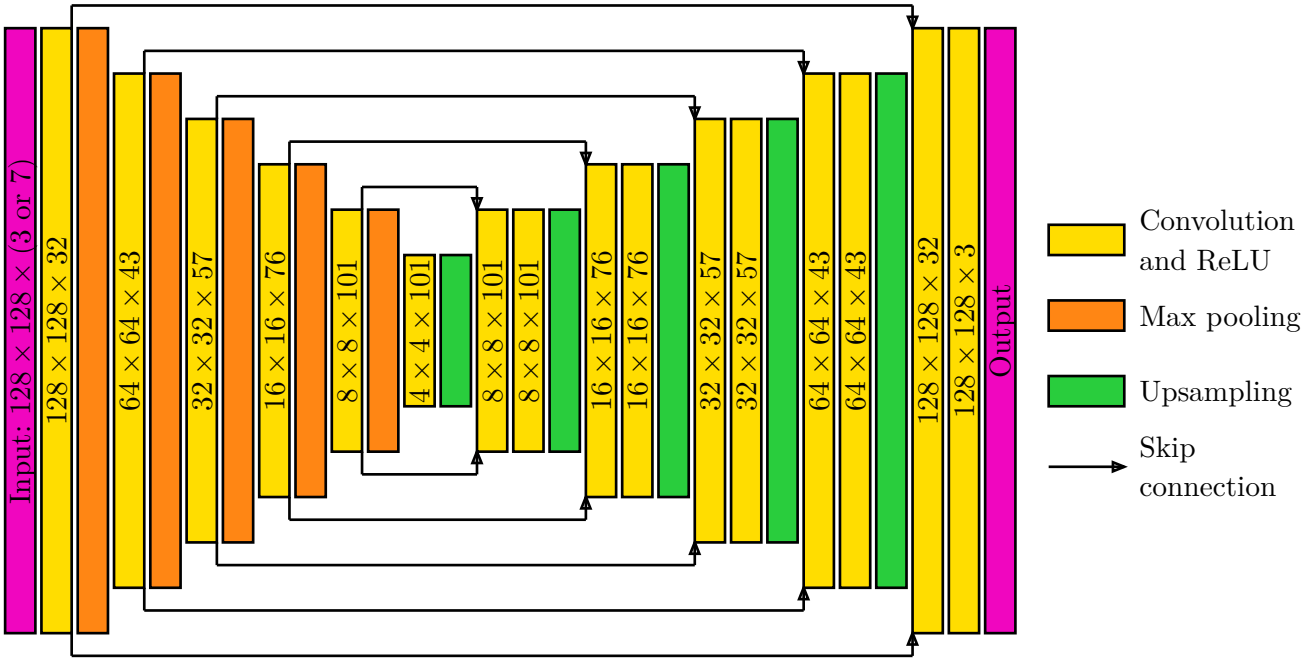


Figure 2: Architecture of the network.

On the figure, the dimensions listed in some of the layers correspond to the dimensions of the *output* of the layer when the network is given a 128×128 tile as input. As the network is fully convolutional, it can be executed on images with other sizes. In this case, the dimension of the output of each layer depends on the dimension of the input image.

The different types of layers are the following:

- **Convolution layer**. Perform a 3×3 convolution on the input. A constant bias is added to each output feature, and the `ReLU` activation function is used. As a reminder, $\text{ReLU} : x \mapsto \max(x, 0)$. Some coefficients are not well-defined since they correspond to pixels outside the image. They are replaced with zeros.
- **Max pooling**. Each 2×2 block of the input is merged into one single pixel by taking the maximum value of each component among the pixels in the block. If the input has a size of $w \times h \times d$ (a $w \times h$ image with d components per pixel), the output will have a size of $\frac{w}{2} \times \frac{h}{2} \times d$.
- **Nearest neighbor upsampling**. The output is divided into 2×2 blocks. All the pixels in the same block have the same value: the value of the pixel that has the position of the block in the original input. If the input has a size of $w \times h \times d$ (a $w \times h$ image with d components per pixel), the output will have a size of $2w \times 2h \times d$.
- **Skip connection**. Concatenates the features of two images with the same dimensions.

2.3 Training and evaluation

As the training of the network was not the goal of the internship, I did not spend too much time trying to optimize it. It is far from being optimal, but it works well-enough to demonstrate the efficiency of convolutional networks to denoise images. Plus, I used an architecture that is known to perform well at this task: it has already been shown that with correct training, these networks can work well in a large variety of scenes.

2.3.1 Training data

Thirty different scenes are generated randomly. They contain different objects (cubes, spheres, torus, teapots and monkeys) placed randomly and with a random color. The camera is also placed randomly. The lights are big rectangles placed at the top of the scene: they allow the presence of soft-shadows. Some of the full images are represented on Figure 3.

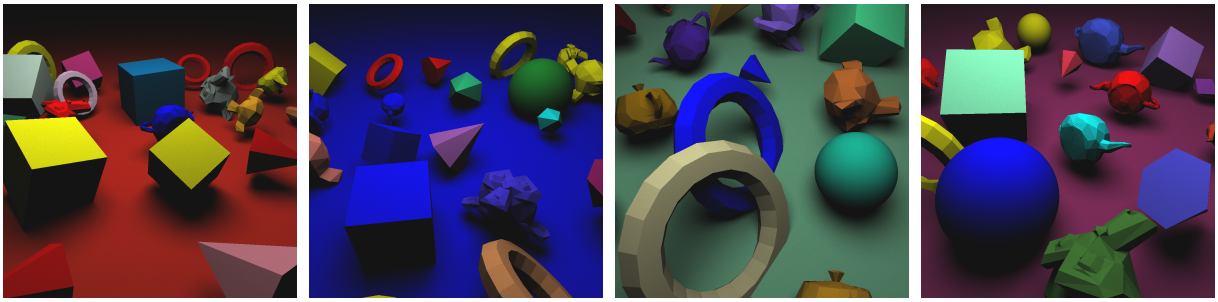


Figure 3: Some of the training scenes

The images of the different scenes are generated with my own renderer [7] using path-tracing. Each scene is rendered three times with 4 samples per pixel to obtain noisy images, and one time with 10000 samples per pixel to obtain the expected image. The network is trained on 900 128×128 tiles taken from the images. Its input consist of several values for each pixel:

- The RGB color of the input pixel.
- Optionally, information about the geometry of the scene: the normal and the depth corresponding to each pixel. This information can help the network to detect edges better.

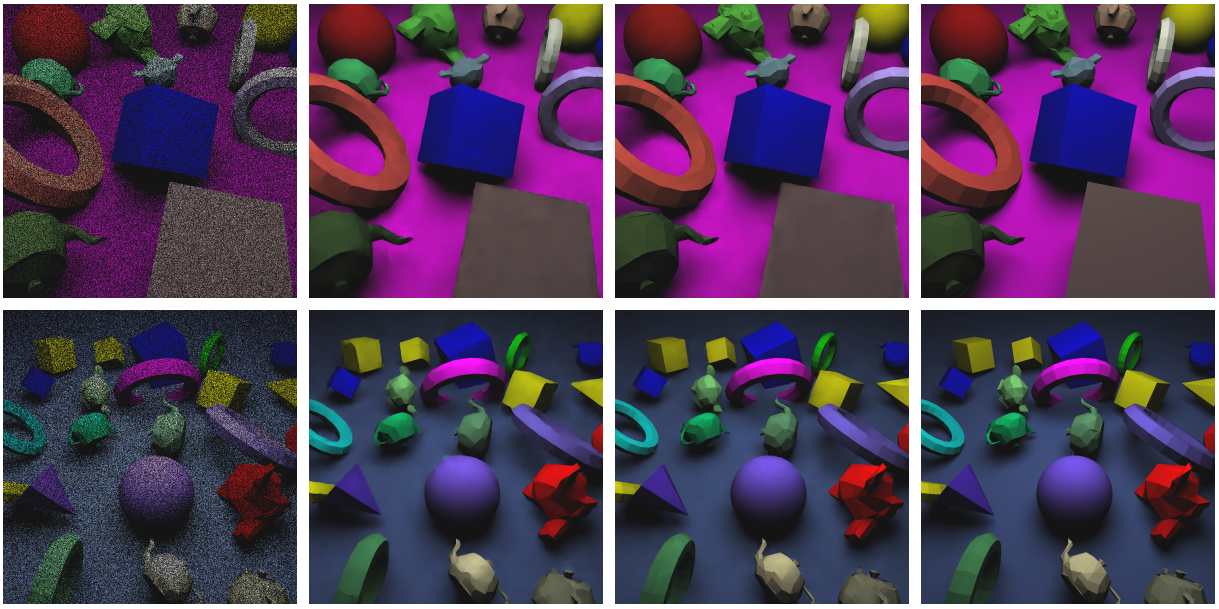
If the geometry of the scene is given, the network contains 861093 trainable parameters.

The network is implemented with Tensorflow [8], and trained for 200 iterations using the Adam algorithm.

2.3.2 Evaluation

The network is evaluated on scenes generated with the same protocol as the training scenes. Two different scenarios are tested: with and without the geometry of the scene as additional input.

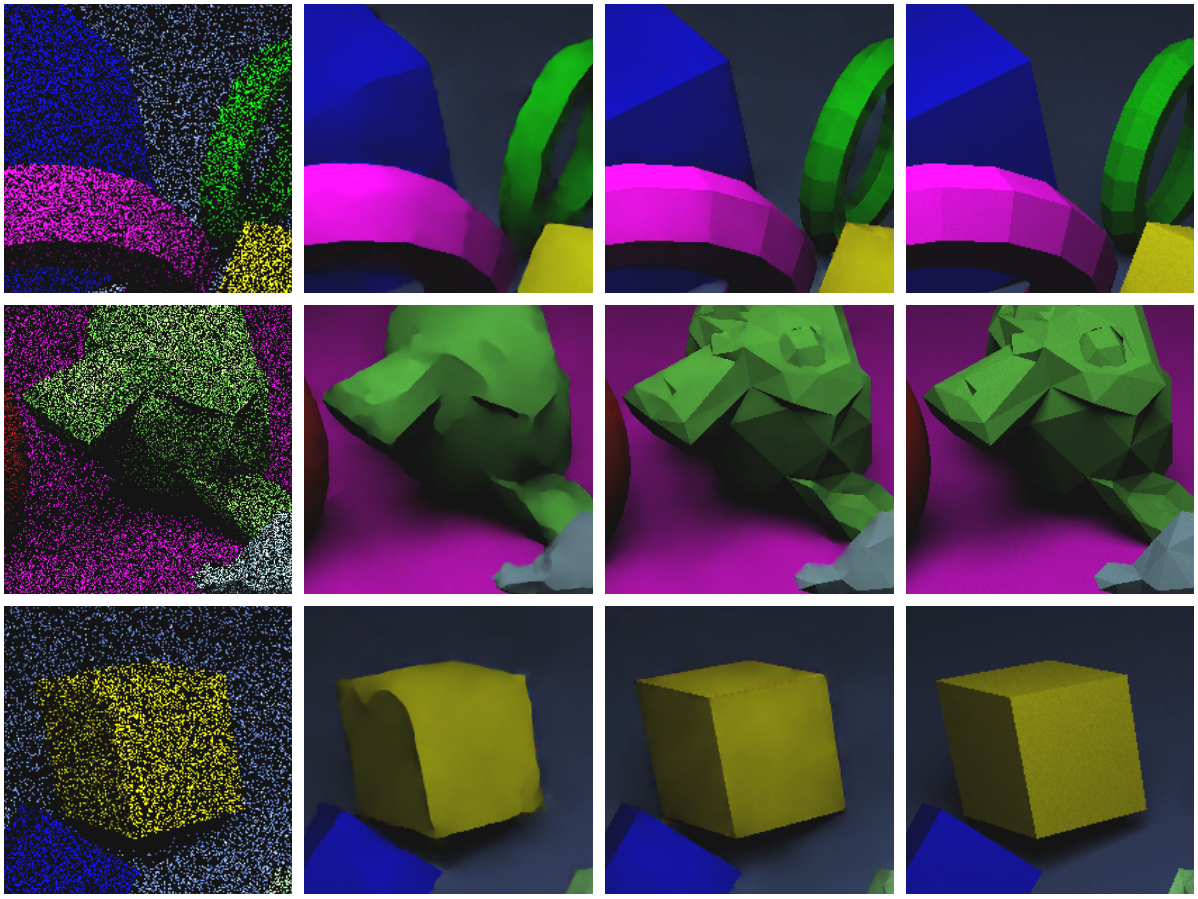
Overall, as shown in Figure 4, the results are rather good: the network is able to reconstruct the expected image from the noisy input.



Input | Without geometry | With geometry | Expected

Figure 4: Denoising output

On Figure 5, we can see how the network performs on small details of some of the scenes. The first example shows a tile where both networks perform rather well: the reconstructions look like what was expected. On the second line, there is an example where the network that does not have access to the geometry of the scene struggles to reconstruct the image: the shape looks very blurry. This is not surprising given the low quality of the input: doing the reconstruction would probably be a difficult task even for a human! The other network does it very efficiently: as it has access to the geometry of the mesh, it is able to draw the edges of the mesh at the right place. Without having access to the geometry of the scene, the first network can realize even worse predictions, as shown in the last example. It understands that two different faces must be separated by an edge, but fails to draw the boundary correctly. The cube appears really distorted.



Input | Without geometry | With geometry | Expected

Figure 5: Impact of adding the geometry of the scene as input to the network

3 Introduction to GPU computing

Before getting into all the details of GPU denoising, it is necessary to understand how GPUs work, and how to write code that can be executed quickly on a graphic card. I will first introduce the general architecture of a graphic card, and then explain some hardware effects that affect the performance of the code.

3.1 Architecture of a GPU

The architecture of a GPU is really different from what we are used to when we write code that runs on CPUs. A GPU is designed to perform similar tasks on different data very efficiently. GPUs were firstly created to perform graphic tasks, but they are now often used to perform general computations (that is known as GPGPU: General-Purpose computing on Graphics Processing Units). In the report, I will only use them to perform general computations, without using the graphic pipeline that is available.

3.1.1 Thread hierarchy

The program executed on the graphic card can be written in a language similar to C. A program is composed of several *shaders*. Each shader defines a function, the *kernel*, that can be executed many times with different data in parallel: each of these instances is called a *thread*. On Figure 6, the threads are represented by the small orange arrows.

A thread is part of a *thread block*, a programming abstraction representing a group of threads executed on the same physical component, called a *streaming multiprocessor* (SM). Threads belonging to the same thread block can communicate by sharing memory. They are

not necessarily executed at the same time, but they can be manually synchronised. However, thread blocks are completely independent: they can be executed on different multiprocessors at different times, so threads of different thread blocks can not communicate.

At a lower level, the threads of a thread group are divided into *warps*. Each warp is a group of 32 threads executed at the same time by the streaming multiprocessor. At a given time, all the 32 processors in the SM will perform the same instruction, but on different data. In the figure, the warps are represented by the big orange arrows. The warps on the right are being executed, while the warps on the left are waiting to be scheduled.

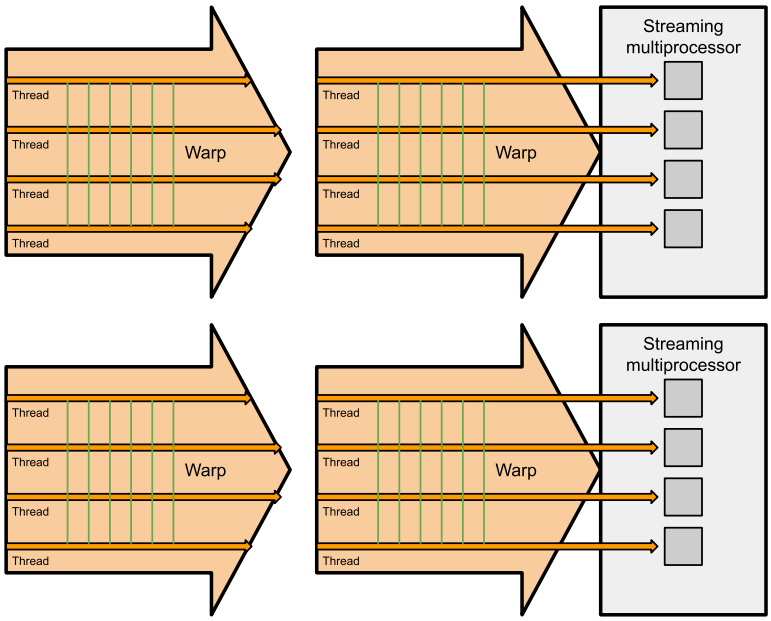


Figure 6: Thread hierarchy

3.1.2 Memory hierarchy

The data manipulated by the GPU can be stored on different memories, each type of memory having different size and access time.

- The biggest memory available is called the *global memory*. It is similar to the RAM used when writing CPU code. It can store a few Gigabytes of data, but it has the drawback of being very slow: each request can take hundreds of nanoseconds to be executed, as shown in Figure 7. The global memory is shared among the threads.
- The *shared memory* is a smaller but more efficient memory. It has a size around 100 KB [9, 16.2], and can answer each request in a few nanoseconds as shown in Figure 8. Shared memory is located on each SM, and only the warps scheduled on the SM can access it. As a consequence, only the threads of the same thread block, which are guaranteed to be executed on the same SM, can share some shared memory.
- A thread also has access to a small amount of very fast memory: *registers*. They are used to store variables . If it is not possible to store all the variables on registers, some of them are spilled to local memory. They are then stored on the device memory, which makes their access times significantly longer [9, 5.3.2].

3.2 Hardware effects

3.2.1 Accessing the global memory

Since the global memory is mainly stored on the GPU's biggest and slowest memory, it is costly to use it: each request can take several hundreds of nanoseconds to be performed. Thankfully, to reduce this time, the global memory can be cached on L1 and L2 caches.

To show this behavior, I created a simple program that, given an integer n and a permutation σ of $[\![0, n-1]\!]$, computes $\sigma^{1000000}(0)$. This prevents compiler optimizations and forces the requests to be performed serially.

```
uint pos = 0;
for (int i = 0; i < 1000000; i++) {
    pos = buffer[pos];
}
```

Figure 7 shows the time needed to read a value from the global memory as a function of n, the size of the permutation.

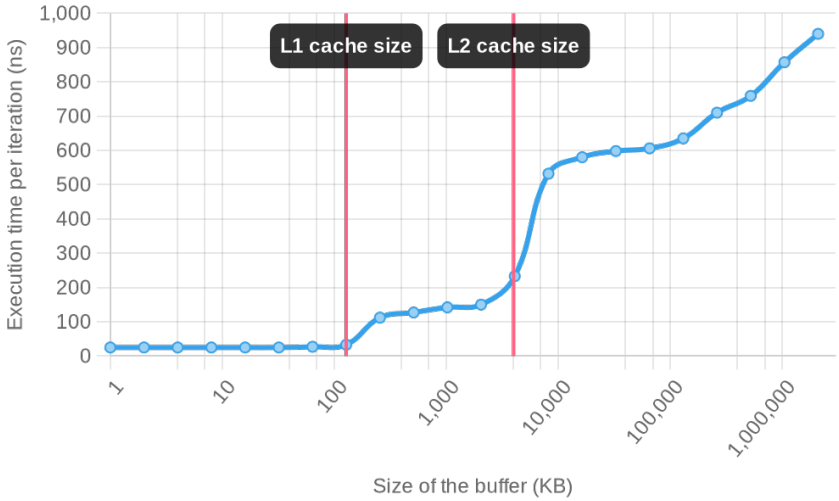


Figure 7: Access time to global memory

We can distinguish three phases:

- When n is small, all the data can be stored on the L1 cache. The access time is small.
- When n has an intermediate size, the L2 cache is used. The requests are about 100 ns slower than before.
- When n gets bigger, the caches become useless and the data is constantly read from the global memory. The access time is about 600 ns per request.
- I am not sure why the access time keeps increasing, but it is probably not important since it only concerns very big buffers.

3.2.2 Memory coalescing

As shown in the previous section, accessing the global memory can be a costly operation. To limit the overhead of accessing this part of the memory, the requests of the different threads in the warp can be coalesced: they are automatically merged into a smaller number of requests to larger portions of global memory.

The documentation of Nvidia [10, 9.2.1] describes the rule used to coalesce memory accesses: "The concurrent accesses of the threads of a warp will coalesce into a number of transactions equal to the number of 32-byte transactions necessary to service all of the threads

of the warp.". As specified later in the same document [10, 9.2.1.2], the transactions are aligned with 32-bytes segments.

As non-coalesced memory access can be really costly, it is important to avoid them when possible.

3.2.3 Bank conflicts

As we saw in the previous section, the shared memory plays a central role in the execution of a program. It allows storing data of an intermediate size and sharing it between threads of the same block without having to use the global memory which is hundred times slower.

The shared memory of an SM is stored in 32 physical components called *memory banks*. A memory bank can answer to only one request at a time: if several threads try to access different bytes stored in the same memory bank, the requests will be processed serially – this is called a *bank conflict*. As this is slower than processing the requests in parallel, bank conflict should be avoided. In the ideal case, when a request is performed, each thread should use a different memory bank than the others. It is the responsibility of the developer to avoid these conflicts. Fortunately, it is easy to know where each variable is stored: a byte at address a is stored in the $\left(\left\lfloor \frac{a}{4} \right\rfloor \bmod 32\right)^{\text{th}}$ memory bank.

To measure the impact of bank conflicts, I created a simple program that runs on the GPU. Only one thread block of 32 threads is created, so all the threads are executed together in the same warp. I tested 32 different scenarios: in the k^{th} scenario, the i^{th} thread accesses data at pseudo-random addresses belonging only to the first memory bank if $i \leq k$, or to the i^{th} memory bank otherwise. All the bank conflicts occur on the first memory bank: k threads try to access it at the same time. In particular, when $k = 1$, there is no bank conflict. The measured execution times in the different scenarios are plotted on Figure 8.

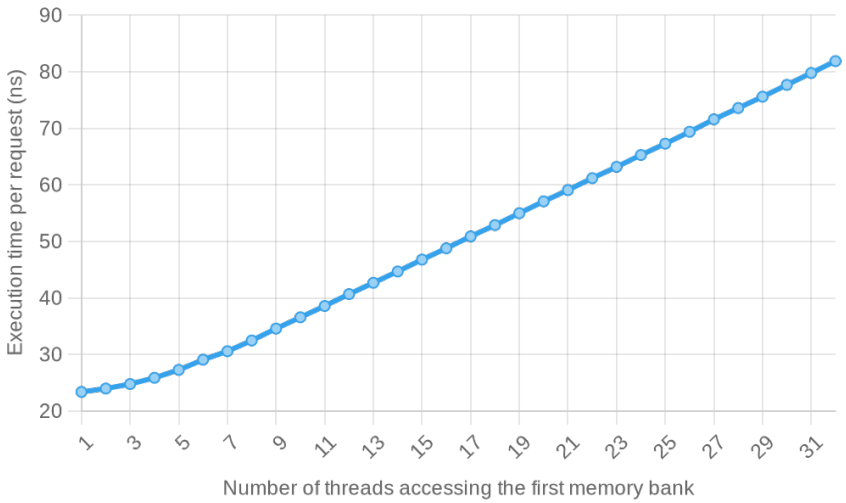


Figure 8: Overhead of bank conflicts

We can see that the execution time is an affine function of the number of threads accessing the first memory bank. Each conflict produces an overhead of about 2 ns: this is the time needed to read from and write to the shared memory.

It may be interesting to note that bank conflicts occur only when threads try to access different bytes. If two threads access the same byte, no bank conflict occurs as the memory can be sent to the two threads at the same time. This may append, for instance, if the variables that are being accessed use less than 32 bits of memory, which is the case of `float16` [9, 16.4.3].

3.2.4 Thread divergence

As described earlier, all the threads in the same warp are executed together following the *SIMT* (Single Instruction, Multiple Thread) model. Each instruction is applied by all the processors of the SM at the same time, each processor working on its own data. However, the shader code may contain `if` statements. In such a case, it is possible that a part of the code should be executed by only some of the threads in the warp. When this happens, the instructions are still enumerated and sent to all the processors, but the processors that shouldn't execute the instructions are *disabled*.

This effect can have an important impact on performances. For instance, let's consider the following program where a condition `C` is used to choose whether to execute `f` or `g`.

```
if C:
    f()
else:
    g()
```

If the condition `C` is verified by all the threads in the warp, only the code from `f` will be executed. Similarly, if none of the threads verify `C`, only the code from `g` will be executed. However, if some threads verify `C` while others do not, both `f` and `g` will have to be executed. As all the processors in the SM have to execute the same instruction at each step, the two branches will be processed serially.

To show this effect, I created a program composed of 32 threads. A parameter c given to each of them is used to select the number of threads that will execute `f`. The other threads will execute `g`. The functions `f` and `g` are very similar and take approximately the same time to be executed. The execution time of the program depends on c:

- If $c = 0$ or $c = 32$, all the threads execute the same function. The total execution time is about 137 ms.
- If $c \in [\![1, 31]\!]$, the two functions are executed. The total execution time is about 274 ms, which is about two times slower.

This phenomenon must be taken into account when trying to optimize GPU shaders. It illustrates how GPUs are different from CPUs.

4 Matrix multiplication

4.1 State of the art

As explained earlier, denoising an image can be done using machine learning techniques. To be able to use these methods in real-time, it is necessary to be able to perform matrix products efficiently on the GPU. As it is a very important problem, a lot of research has been realized in this domain.

To obtain good performance, a first approach consists of implementing the naive algorithm very efficiently. Nvidia developed Cutlass, an optimized library that can be used with Cuda, a language designed to perform general computations on GPUs. Nvidia's blog [11] describes how the multiplication is done. A three-level hierarchy is used to take advantage of the thread and memory hierarchies of GPUs. This library can use the tensor cores – see the next section – to accelerate the product. The problems with such libraries is that they are usually designed to be used with Cuda, and not with Vulkan. It is thus difficult to use them in real-time graphic applications that rely on Vulkan (Cuda does not support all the graphic pipeline available on

the GPU). According to my supervisor, using CUDA libraries would introduce an overhead of several milliseconds per frame, which is not acceptable.

To allow using the tensor cores of the GPU in VULKAN, the *cooperative matrix* extension was published by NVIDIA. Since this extension is relatively new – the GLSL extension was released in 2019 [12] –, no good library is available to perform matrix multiplication efficiently. There are a few VULKAN libraries that can be used to perform fast matrix product, such as NCNN [13]. However, they do not seem to be able to take advantage of the tensor cores.

From an algorithmic perspective, researchers also investigate the possibility of using more efficient algorithms such as STRASSEN's algorithm [14]. However, the performance is not significantly improved compared to the naive algorithm, and an efficient implementation of the naive algorithm is still needed. Thus, I chose not to focus on this approach.

4.2 Tensor cores

To improve the performance of matrix multiplication algorithms on its GPUs, NVIDIA added special components dedicated to this task: *tensor cores*. Each SM contains four tensor cores [9, 16.7.1] that can perform matrix multiplications really quickly [9, 7.24.6]. Since tensor cores are hardware components, they can not work on arbitrary types of data. NVIDIA's documentation [9] [9, 7.24.6] lists the matrix formats that are supported by the tensor cores. In the rest of the report, I will only use them with 16×16 matrices containing `float16` coefficients. The performance of the tensor cores have already been studied and evaluated [15].

I benchmarked these tensor cores with the following protocol. Only one thread block is executed, so that all the code runs on the same SM. Given $n \geq 1$, n warps (that is, $32 \times n$ threads) are started. Each warp computes three million matrix multiplications. To avoid expensive accesses to memory, only three matrices are used for the whole computation. The execution time is plotted on Figure 9.

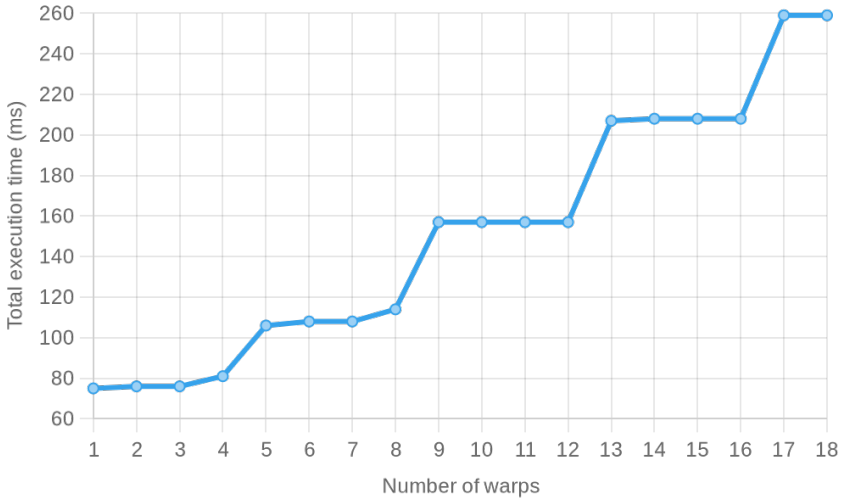


Figure 9: Benchmark of the tensor cores

The execution time depends linearly on $\lfloor \frac{n}{4} \rfloor$. It is thus important to schedule enough warps on the same SM to be able to use all the available tensor cores simultaneously. It is important to understand this behavior to obtain good performance, especially since it is not really directly documented.

We can also use these measurements to compute an estimation of the maximal throughput potentially achievable by the GPU. The graph shows that each series of $4 \times 3 \times 10^6$ multipli-

cations takes 50ms. Thus, each SM can execute 2.4×10^8 multiplications per second. Since the matrices are 16×16, this corresponds to almost 1 TFlops. The GPU contains 48 SMs, so the maximal throughput theoretically achievable is around 47 TFlops.

To multiply two matrices, two types of floating-point operations are necessary: additions and multiplications. To compute the throughput of an algorithm, I will consider the number of floating-point operations as the number of multiply-add operations. Multiplying a $n \times p$ matrix by a $p \times m$ matrix will therefore count as realizing $n \times p \times m$ operations. It is different from what is sometimes done: for example, the benchmark of the cooperative matrix extension [16] cited on the presentation [17] rather counts the real number of operations, that is twice the number of multiply-add operations. I believe that it is not coherent as additions and multiplications do not take the same time to be executed, and can sometimes even be executed together.

4.3 Matrix multiplication algorithms

Suppose that we have two matrices, $A \in \mathbb{R}^{n \times p}$ and $B \in \mathbb{R}^{p \times m}$, and we want to compute their product $C = A \times B \in \mathbb{R}^{n \times m}$. For some of the following algorithms, we will decompose these matrices in small blocks. We will suppose that the dimensions of the matrices are multiples of the size of the blocks. If it is not the case, it is possible to extend the matrices with lines or columns of zeros.

All the following approaches implement the naive $\mathcal{O}(n \times p \times m)$ algorithm. However, they have different ways to access the memory, which are the cause of different performance. Each subsection contains an estimation of the throughput of the corresponding algorithm. It is estimated with the computation of the product of 4096×4096 matrices. The last subsection compares the performances of all these algorithms.

Some of these algorithms have already been explained and evaluated in a recent presentation [17]. However, these measurements have several limitations.

- One of the goals of the presentation was to show the speed of tensor cores compared to regular cores. However, the two components are not used in the same way: the memory is used more efficiently with the tensor cores. This could produce a bias, and it made my supervisor quite sceptical about these results.
- The throughput obtained with each method is not compared to the theoretical maximal throughput achievable. The presentation does not say if the best method presented is near optimal or if there is still room for improvement.
- The choice of some parameters such as the tile size is not discussed.

4.3.1 Naive algorithm

In this first, simple version, one thread is responsible for computing one coefficient of C. Let's consider the thread at position (i, j) that computes $C_{i,j}$. To obtain the result, it simply evaluates the expression:

$$\sum_{k=1}^{p} A_{i,k} B_{k,j}$$

Since each coefficient of A and B is read p times from the global memory, this approach is really slow: the throughput is only about 0.17 TFlops. This method is implemented in the `standard.comp` shader.

4.3.2 Shared memory

To limit the number of accesses to the global memory, the matrices are divided into tiles of size 16×16. A thread block, composed of 16×16 threads, is responsible for the computation of all the coefficients of a tile of the output matrix. The algorithm is similar to the previous one, but this time, full tiles are loaded from A and B to the shared memory of the SM. Once they are loaded, the product of the tiles is computed as usual, only reading from and writing to the shared memory.

To improve the performance, the following optimisations are made:

- To coalesce the requests to the global memory, the coefficients of the 16×16 tiles of the matrices are stored contiguously.
- To avoid bank conflicts when performing the product, the tiles of B are stored transposed.

This approach is a bit faster, as the coefficients of A and B are read only $\frac{p}{16}$ times from global memory. The other accesses are from the shared memory which is much faster. This approach, which has a throughput of about 0.81 TFlops, is implemented in the `shared.comp` shader.

The tiles have a size of 16×16 because it corresponds to the size of the cooperative matrices that will be used in the following subsection.

4.3.3 Cooperative matrices

This algorithm is similar to the previous one, but instead of loading the tiles to the shared memory and doing the product on the regular cores, cooperative matrices are used to do the product on the tensor cores. As in the previous section, the coefficients of B are stored contiguously.

The method takes advantage of the physical components of the GPU that are available to perform matrix multiplications, but it is still slow: the tiles are read from global memory each time. This approach, which has a throughput of about 8.6 TFlops, is implemented in the `coopmat.comp` shader.

We can see that tensor cores greatly improve the speed of the product: the memory is loaded in the same way as before, but the execution time is ten times lower. This illustrates the efficiency of the tensor cores.

4.3.4 Efficient use of cooperative matrices

The problem with the previous version is that the coefficients are loaded frequently from the global memory. To avoid this, it is possible to load parts of columns of A and parts of lines of B, split them into 16×16 tiles and compute their outer product with cooperative matrices. This process is represented on Figure 10. On the schema, each square in a matrix represents a 16×16 submatrix.

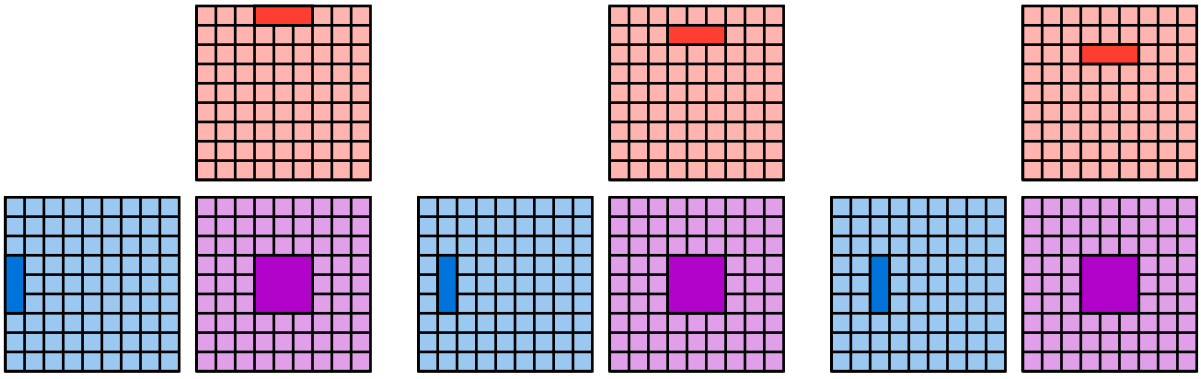


Figure 10: Efficient tiled product

In more details, if each thread block is responsible for the computation of a $a \times a$ block of the output matrix (a being a multiple of 16), $\frac{n}{a} \times \frac{m}{a}$ thread blocks are created. Each thread block has the responsibility to compute the values of a $a \times a$ block of the output matrix. To do it, it performs $\frac{k}{16}$ iterations. At each iteration, it loads a part of 16 columns of A and a part of 16 lines of B to the shared memory, computes their outer product efficiently using the tensor cores (the coefficients are divided into many 16×16 tiles) and finally adds them to the final result.

One question remains: how to choose a, the size of the blocks of the output matrix? To reduce the number of reads from and writes to the global memory as much as possible, the blocks must be as big as possible. However, two constraints must be verified:

- All the data used by a thread block must fit in the memory.
- There must be enough different blocks to avoid leaving some SM unused.

On the GPU that I used, the size of the shared memory is 100 KB, and each SM can contain 65536 32-bits registers. Since the coefficients are stored on `float16`, using 256×256 blocks was a reasonable choice. Experimentally, it gave good results compared to other sizes of matrix.

To verify the second constraint, it may be interesting to use smaller blocks when the input matrices are small enough. This is not the case here since we use 4096×4096 matrices for the benchmark.

It is important to note that with this size of block, most of the shared memory is used, so it is likely that only one thread block can be scheduled on each same SM. However, as shown in Section 4.2, to be able to use the four tensor cores available on the SM, several warps must be used. For this reason, I made each thread block contain 8 warps – that is, $32 \times 8 = 256$ threads. All these warps cooperate to load the input data to the shared memory. Once the data is loaded, each of them is responsible for the computation of a part of the product. The repartition of the thread blocks is represented on Figure 11: the square represents the submatrix of the output matrix that the thread block is computing, and each rectangle represents the submatrix computed by one of the warps.

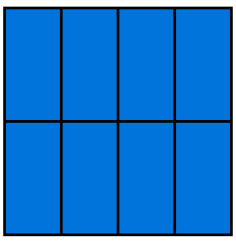


Figure 11: Submatrices of each warp

Since the warps are not independent, they have to be manually synchronised. This synchronisation must be done with caution, as over-synchronising the warps could lead to a slow execution of the program and a low usage of the tensor cores. I used a different method from the benchmark [16] provided by Nvidia: I splitted the warps into two groups that always perform different actions: when one is loading data, the other is performing products. This removes a part of the latency: otherwise, the tensor cores are left unused when the treads are loading memory. This optimization makes my implementation a little bit faster than the tensor cores benchmark, even if my code is not very optimized (especially the loading of the data).

This approach, which has a throughput of about 34 TFlops, is implemented in the `tiled.comp` shader.

4.3.5 Performance evaluation

The performances of the different algorithms are presented on the following table. All the algorithms are evaluated with the multiplication of two 4096×4096 matrices.

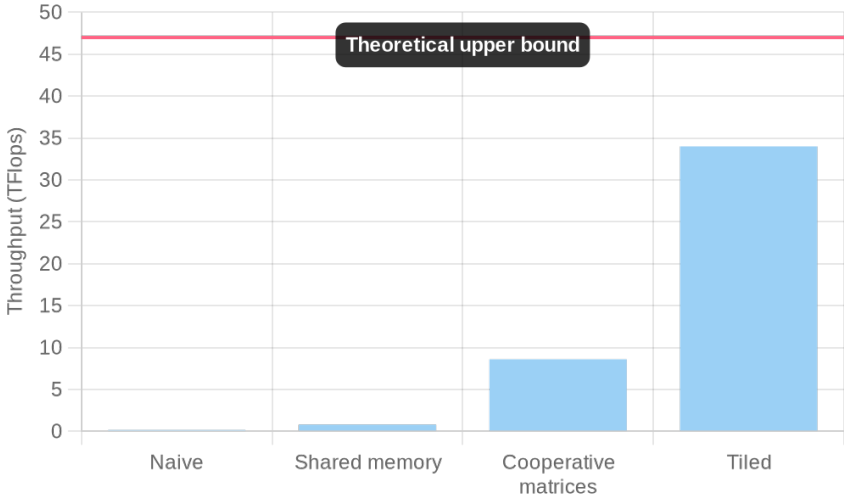


Figure 12: Performance of the different methods

We can see that the best version of the algorithm is very efficient: the throughput almost reaches the maximal possible value. This means that the tensor cores are used efficiently: we manage so send them data fast enough to keep them busy.

5 Fast network inferencing

Now that we know how to multiply matrices efficiently on the Gpu, we can start to see how to evaluate the Cnn quickly. Since the main operation of the network is the convolution, we will focus on this operation.

5.1 State of the art

Since the publication of Chaitanya's article [2] in 2017, many articles about GPU denoising in real-time have been published. They rely on variants of the U-Net architecture.

To speed up the rendering of a high-resolution image, it is generally possible to render a smaller image, and then transform it into a high-resolution image using machine learning. A network architecture has been proposed [6] to realize both the denoising and the supersampling parts at the same time. To realize the inference of the network quickly, Nvidia's TensorRT [18] is used. This library generates Cuda kernels that are executed on the GPU. These kernels are rather efficient and can take advantage of the tensor cores: the whole network can be inferred in a few milliseconds. However, the library doesn't support some operations specific to denoising, so some of the steps must be implemented manually in shaders. Plus, some time is lost converting the data in a format accepted by TensorRT, which is suboptimal [6, 4.3].

5.2 Efficient convolution

The convolutions are the operations that take most of the time of the evaluation of the network. It is necessary to implement them carefully to have a network that can be executed at real-time rates. A convolution can be performed with the same algorithm as matrix multiplication, but with different matrices.

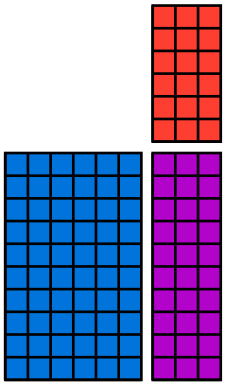


Figure 13: Reduction of the convolution to matrix product

Figure 13 shows the matrices that should be used to compute the convolution:

- The matrix A (in blue) contains the input data. Each line contains the input data for a pixel, that is the features of the pixel and the features of its 8 neighbours (for a 3×3 convolution as here).
- The matrix B (in red) contains the weights. Each column gives the weights corresponding to a given output feature.
- The matrix C (in purple) contains the output data. Each line contains the features of a pixel of the image.

We can note that the matrices are not square: if A is $n \times p$ and B is $p \times m$, we have $n \gg p \gg m$.

In practice, these matrices are never explicitly constructed: the input matrices are generated on the fly thanks to the input data, and the coefficients of the output matrix are written directly at the right place of the memory.

Unfortunately, I did not have the time to implement the efficient algorithm seen previously: I only used a simpler and less efficient variant. For this reason, the final performance is far from what could have probably been achieved.

5.3 Inferencing the network

The evaluation of the network is composed of different operations: convolutions, max-pooling, skip connections and upsampling. If these different operations are realized separately in different shaders, the data must be accessed from global memory many times, which can take some time. To avoid this overhead, the max-pooling, the skip connections and the upsampling operations are not performed in their own shader. Instead, they are realized in a shader that also performs a convolution.

To achieve this, a shader is generated for each convolution layer: the same code is compiled several times with different parameters. The shader performs a convolution. It is able to read its input data from several buffers, which is needed when the outputs of several layers must be concatenated (it happens after a skip connection). The shader also performs a max-pooling or an upsampling step if needed. This also has the advantage of allowing the compiler to better optimize the code. For instance, if the number of features is known at compile time, it allows the compiler to unroll more loops, which has an important impact on performances.

5.4 Performance evaluation

Table 1 and Table 2 show the performance of each convolution layer of the network. As in previous sections, the throughput is defined as the number of multiply-add operations per second during the convolution. All the measurements are obtained with a 1024×1024 image as input. The time taken to realize the upsampling or the max-pooling is included in the measurements, but it is negligible compared to the time of the convolution.

Convolution	0	1	2	3	4	5
Time (ms)	0.93	2.31	0.99	0.83	0.99	0.51
TFlops	2.27	1.41	1.46	0.77	0.28	0.18

Table 1: Performance of the encoder

Convolution	6	7	8	9	10	11	12	13	14	15
Time (ms)	0.84	0.35	1.17	0.82	2.82	1.2	3.87	1.68	8.16	3.32
TFlops	0.9	2.17	1.69	2.08	1.59	3.18	2.62	5.2	2.78	0.55

Table 2: Performance of the decoder

We can see that the throughput is about 10 times lower than the throughput obtained when only performing matrix multiplication. This means that the tensor cores are under-used, and that most of the time is taken by accessing the global memory.

This is due to the fact that I did not have the time to implement the efficient method seen previously, but instead used a less efficient variant. I think it should be possible to significantly improve the execution time of the convolution by implementing this method. However, the version that I implemented is only about two times slower than the version implemented with Tensorflow, which means that it is already rather efficient.

6 Conclusion

During my internship, I worked on real-time GPU denoising using convolutional neural networks. I first had to learn how GPU computing works, and how to write efficient programs that run on GPU. It took me several weeks, it was very frustrating to spend such a long time making everything work. Then, I trained a convolutional neural network to denoise images thanks to

scenes that I generated. I also worked on a fast algorithm to evaluate this network quickly: I first focused on matrix multiplication algorithms, then on convolution algorithms.

This internship gave me the occasion to learn about Gpu computing and to work with machine learning on a real-world problem, which is something I found really interesting… even if I had to spend a lot of time trying to solve very strange issues specific to Gpu computing.

Overall, this internship was a very good experience. Everyone was very welcoming and available to help me both living in Karlsruhe and solving the problems that I had related to my subject. I would like to thank them for making this internship possible.

Bibliography

[1] "What is path tracing?." https://blogs.nvidia.com/blog/2022/03/23/what-is-path-tracing/

[2] C. R. A. Chaitanya, A. S. Kaplanyan, et al., "Interactive reconstruction of monte carlo image sequences using a recurrent denoising autoencoder," *ACM Trans. Graph. (Tog)*, vol. 36, no. 4, pp. 1–12, 2017.

[3] O. Ronneberger, P. Fischer, and T. Brox, "U-net: convolutional networks for biomedical image segmentation," in *Med. Image Comput. Computer-Assisted Intervention–MICCAI 2015: 18th Int. Conference, Munich, Germany, October 5-9, 2015, Proceedings, Part III 18*, 2015, pp. 234–241.

[4] J. Hasselgren, J. Munkberg, M. Salvi, A. Patney, and A. Lefohn, "Neural temporal adaptive sampling and denoising," in *Comput. Graph. Forum*, vol. 39, 2020, pp. 147–155.

[5] N. Hofmann, J. Hasselgren, and J. Munkberg, "Joint neural denoising of surfaces and volumes," *Proc. ACM Comput. Graph. Interactive Techn.*, vol. 6, no. 1, pp. 1–16, 2023.

[6] M. M. Thomas, G. Liktor, et al., "Temporally stable real-time joint neural denoising and supersampling," *Proc. ACM Comput. Graph. Interactive Techn.*, vol. 5, no. 3, pp. 1–22, 2022.

[7] A. Vannson, "3d renderer." [Online]. Available: https://github.com/AdrienVannson/rust-renderer

[8] M. Abadi, P. Barham, et al., "Tensorflow: a system for large-scale machine learning," in *12th USENIX Symp. Operating Syst. Des. Implementation (OSDI 16)*, 2016, pp. 265–283.

[9] "Cuda programming guide." https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[10] "Cuda best practices." https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html

[11] "Cutlass matrix multiplication algorithm." https://developer.nvidia.com/blog/cutlass-linear-algebra-cuda/

[12] "Cooperative matrix glsl extension." https://github.com/KhronosGroup/GLSL/blob/master/extensions/nv/GLSL_NV_cooperative_matrix.txt

[13] H. Ni, and T. ncnn contributors, "Ncnn," 2017. [Online]. Available: https://github.com/Tencent/ncnn

[14] J. Huang, C. D. Yu, and R. A. v. d. Geijn, "Strassen's algorithm reloaded on gpus," *ACM Trans. Math. Softw. (Toms)*, vol. 46, no. 1, pp. 1–22, 2020.

[15] W. Sun, A. Li, T. Geng, S. Stuijk, and H. Corporaal, "Dissecting tensor cores via microbenchmarks: latency, throughput and numeric behaviors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 34, no. 1, pp. 246–261, 2022.

[16] J. Bolz, "Cooperative matrix performance." https://github.com/jeffbolznv/vk_cooperative_matrix_perf

[17] B. Pierre, "Cooperative matrix multiply." https://www.khronos.org/assets/uploads/developers/presentations/Cooperative_Matrix_May22.pdf

[18] Nvidia, "Tensor-rt." [Online]. Available: https://developer.nvidia.com/tensorrt-getting-started

[19] "Nvidia geforce rtx 3070 ti graphic card." https://www.nvidia.com/en-us/geforce/graphics-cards/30-series/rtx-3070-3070ti/

[20] "Nvidia geforce rtx 3070 ti specs." https://www.techpowerup.com/gpu-specs/geforce-rtx-3070-ti.c3675

A Hardware specification

All the experiments mentioned in this report were realized on the same computer. It is equipped with the graphic card Nvidia GeForce Rtx 3070 Ti [19]. The characteristics of this card are available online [20]. Among other things, this card has the following characteristics:

- 48 Streaming Multiprocessors at 1.58 GHz
- 8.6 compute capability: this indicates some properties supported by the graphic card

The 8.6 compute capability corresponds to the characteristics below. A more in-depth description can be found on Nvidia's website [9, 16.].

- 65536 32-bit registers per SM
- 65536 32-bit registers per thread block
- 255 32-bit registers per thread
- 100 KB of shared memory per SM

B Performance measurement protocol

To save energy, the GPU is able to adapt its clock speed to how much computation needs to be done. To perform reliable and reproducible measurements, it is necessary to fix this clock speed. The speeds available on the GPU can be listed with `nvidia-smi --query-supported-clocks=gpu_name,memory,graphics --format=csv`. I chose to fix the memory clock speed at 9501 MHz and the core speed at 1920 MHz. Performance measurements are realized following this protocol:

- Enable the persistence mode: `nvidia-smi -i 0 -pm 1`
- Fix the GPU core speed: `nvidia-smi --lock-gpu-clocks=1920`
- Fix the memory clock speed: `nvidia-smi --lock-memory-clocks=9501`
- Make the measurement
- Reset the settings: `nvidia-smi --reset-gpu-clocks` and `nvidia-smi --reset-memory-clocks`

C Source code

Most of the codes that I wrote during the internship are available on Github at the following address: https://github.com/AdrienVannson/gpu-denoising. A `README.md` file describes the content of this repository. The shaders that I wrote consist of modules of `vkdt`, a software currently developed by my internship supervisor. I only published the code that I wrote, without including the code of the whole software.