ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADO EN INGENIERÍA INFORMÁTICA

PROCESAMIENTO DE IMÁGENES CEREBRALES EN GPU

NEUROIMAGE PROCESSING ON GPU USING CUDA

Realizado por
Javier Cabero Guerra
Tutorizado por
Manuel Ujaldón Martínez
Departamento
Arquitectura de Computadores

UNIVERSIDAD DE MÁLAGA
MÁLAGA, SEPTIEMBRE DE 2015

Fecha defensa:
El Secretario del Tribunal

Resumen: A lo largo del tiempo, el paradigma de computación de propósito general ha evolucionado, produciendo diferentes arquitecturas hardware cuyas caracteristicas son muy distintas. En este trabajo, trataremos de demostrar, a través de distintas aplicaciones pertenecientes al campo del Procesamiento de Imágenes, la diferencia existente entre tres plataformas hardware de Nvidia: dos de la serie de tarjetas gráficas GeForce, la GTX 480 y la GTX 980 y una plataforma de bajo consumo cuyo propósito es el permitir la ejecución de aplicaciones embebidas a la vez que proporcionar una eficiencia extrema: la Jetson TK1.

Respecto a los programas de prueba usaremos cinco ejemplos sacados de los CUDA Samples de Nvidia. Estas aplicaciones tienen una relación directa con el procesamiento de imágenes, dado que los algoritmos implicados en ellas tienen similitudes con los aplicados en el campo del registro de imágenes médico. Tras las pruebas, se mostrará cómo la GTX 980 es tanto el dispositivo con mayor rendimiento como el que mayor consume, se verá que la Jetson TK1 es el dispositivo más eficiente de los tres, se enseñará cómo la GTX 480 es la plataforma que más calor produce y aprenderemos otros efectos producidos por la diferencia entre las arquitecturas que hay entre los dispositivos.

Palabras claves: CUDA, GPGPU, Jetson TK1, GTX 480, GTX 980, Rendimiento, Consumo, Eficiencia, Procesamiento Imágenes Cerebrales

Abstract: As time has passed, the general purpose programming paradigm has evolved, producing different hardware architectures whose characteristics differ widely. In this work, we are going to demonstrate, through different applications belonging to the field of Image Processing, the existing difference between three Nvidia hardware platforms: two of them belong to the GeForce graphics cards series, the GTX 480 and the GTX 980 and one of the low consumption platforms which purpose is to allow the execution of embedded applications as well as providing an extreme efficiency: the Jetson TK1.

With respect to the test applications we will use five examples from Nvidia CUDA Samples. These applications are directly related to Image Processing, as the algorithms they use are similar to those from the field of medical image registration. After the tests, it will be proven that GTX 980 is both the device with the highest computational power and the one that has greater consumption, it will be seen that Jetson TK1 is the most efficient platform, it will be shown that GTX 480 produces more heat than the others and we will learn other effects produced by the existing difference between the architecture of the devices.

Keywords: CUDA, GPGPU, Jetson TK1, GTX 480, GTX 980, Performance, Power Drawback, Efficiency, Neuroimaging

Contents

Introduction

New computing technologies put an imperative effort on reducing power consumption. The search of low power platforms derives from an older perspective which focused the increment of computers performance. This idea continued until too many resources were necessary to feed the machine. At this moment, an inflection point occurred in the device targets: instead of computational power they started to concentrate on efficiency. Having more performance is not a forgotten objective but it is now driven by a reasonable power budget.

One of the main reasons for improving energy efficiency refers to the mobile market. Battery technologies are stuck and cannot improve their energy capacity using the same size at the same cost [45]. The need of saving the little energy available raises, causing a transition from top performance devices to more efficient ones.

On the other hand, we have supercomputers, which have an extraordinary energy usage. These machines need an electric power supply that can reach easily thousands of kilowatts. Here, energy delivering is a problem as important as cooling. The more power usage the more heat will be generated.

The efficient power management task is defined under the term *green computing*. Having less power consumption is a form of achieving this goal, but efficiency is more commonly measured in floating point operations per second by watts of power (FLOPS/W). The most energy efficient supercomputers in the world appear in The Green 500 list [1], which is periodically updated and contains, following this metric, those computers at the top of efficiency. In the November of 2014 update, 8 of the top 10 supercomputers used NVIDIA graphic cards.

1.1 The Testbed

This work aims to demonstrate how new technologies walk through higher processing capacity taking good efficiency into account. Three platforms are going to be used (they will be explained in detail in Section 4.1): GTX 480, Jetson TK1 and GTX 980. Going through these NVIDIA technologies it will be proven how it is possible with less hardware but better architecture optimize power consumption. One important fact is the big gap between the devices, not only in terms of hardware (quantity and architecture) but also in time. Heat generation, energy draw and performance differ widely between them.

The applications used in this work belong to the field of image processing and are similar to some neuroimaging algorithms in the way they manage data. The results obtained for this examples can, thus, be extrapolated to the medical case.

GPU-Z [49] is the application used for measuring power and heat in the GTX devices. It detects voltage and amperage in the GTX 480 and Thermal Design Power (TDP) percentage in the GTX 980. How accurate these sensors are depends on a number of variables but for the sake of this experiment the numbers are stable enough and represent in a reliable way how the program resources usage scales for different configurations.

All the tests performed are compiled in Release mode. Those corresponding to the GeForce graphics cards generate more heat and because of that they are more aware of the room temperature. These tests were computed in middle July in Malaga, Spain, at daytime with 28.5 degrees. Degrees are all measured in Celsius scale.

About the OS, Windows is used to perform the tests on the GeForce graphics cards and Ubuntu is used for the Jetson. For the first example, the GTX 980 test is provided both in Windows and Ubuntu.

Lastly, the CUDA Samples Imaging section programs are used as test applications [10]. We consider that the techniques used in this module are of special interest because they apply in common fields like medicine, robotics and mobile applications.

2 The GPGPU movement

2.1 The GPU Streaming Processor

Graphics Processing Units (GPU) were conceived as specialized processors in computer graphics. These devices free the CPU from tasks related to graphics processing. The reason why this dedicated hardware platform exists is the high computational cost of these tasks, due to the large amount of data that has to be processed in short time intervals.

Since its beginning, the CPU, based on the Von Neumann architecture, has given more importance to the instructions that manipulate data than to the data itself. Because of that, processors are not efficient when accessing to multiple data simultaneously.

The high performance offered by the GPU against the CPU when huge amounts of data are provided comes from the big change in the way information is handled.

From an historical point of view, the sequential pattern has evolved into a new data-centric model. In this new model, data is grouped into streams, making possible the performance of calculations on each of the data elements at the same time.

With the model came a programming paradigm that resulted into the development of a processor specialized in streams, commonly referred to as Streaming Processor.

2.1.1 Advantages and Drawbacks

The architecture of the streaming-based processor is what has mainly defined several advantages and drawbacks.

Its main advantage is scalability, which is the ability to handle a growing amount of work in a capable manner. Since this benefit comes directly from the hardware model nature, the expectations for the future are very high. Actually, it has resulted in the GPU performance doubling every six months since its invention, much faster than the CPU.

However, we have to point out that not all applications benefit from its architecture. Applications that make heavy use of conditional statements are hard to parallelize, as GPU systems operate on the data at the same time and this programs apply different operations depending on the values of the given data.

2.2 Evolution to a General Purpose Architecture

Over the past few years, the use of GPUs has increased in order to speed up codes that originally ran on the CPU. Its original approach, rendering graphics, has evolved into the idea of a flexible and programmable computer (General Purpose GPU or GPGPU) where every program that manages large amounts of data benefits from the GPU hardware capabilities.

Despite being a relatively recent technology, it had a great acceptation, firstly because of the continuous evolution that GPUs have experimented into GPGPUs, and secondly, due to the obtained results against the CPU and its future expectations.

Since the arrival of the first graphic platforms, a number of improvements have allowed to build more efficient devices. In the following sections we are going to examine in a deeper way the most important stages of this evolution.

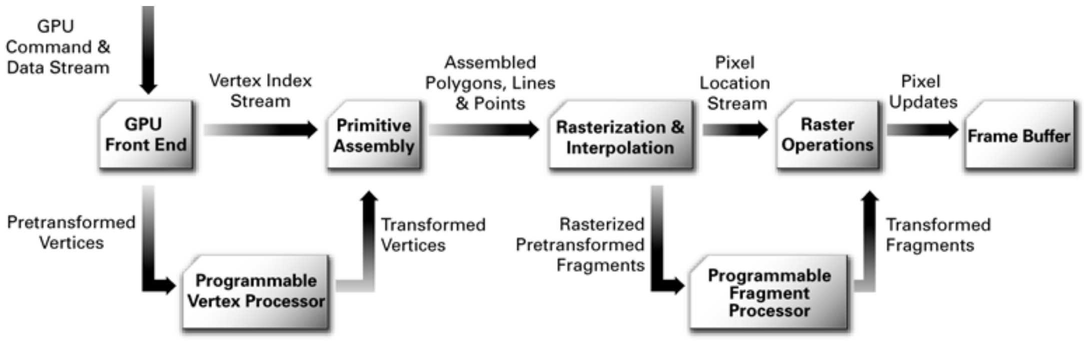

GPU Command & Data Stream
GPU Front End
Vertex Index Stream
Primitive Assembly
Assembled Polygons, Lines & Points
Rasterization & Interpolation
Pixel Location Stream
Raster Operations
Pixel Updates
Frame Buffer
Pretransformed Vertices
Programmable Vertex Processor
Transformed Vertices
Rasterized Pretransformed Fragments
Programmable Fragment Processor
Transformed Fragments

Figure 2.1: *Graphics pipeline after shaders inclusion.*

2.2.1 Starting Point

Since its inception, the GPU has executed parallel algorithms. However, these algorithms always performed the different stages of the rendering process (graphics pipeline), so there was a fixed programming model.

During the 90s, GPU programming began to normalize since the boom of graphical programming gave birth to programming interfaces (including OpenGL and later DirectX) which allowed developers to work with the GPU in a more transparent and efficient manner.

While the software was evolving, hardware companies modified the graphics pipeline introducing two programmable steps in which you could execute self-made programs called shaders, making GPUs more versatile (see Figure 2.1). However, these programms had to be created in assembler code. Because of that, and in order to popularize this new flexible capabilities of the GPU, the creation of new tools was necessary.

Thus, in 2002 HLSL (High-Level Shading Language) was born as a Microsoft initiative. This language provided higher level of abstraction than the assembler, but required the programmer to know the GPU architecture.

In late 2002, Cg (C for graphics) appeared. It was developed by NVIDIA in collaboration with Microsoft and was very similar to HLSL. The language was based on C programming language with some elements adapted to GPUs. Faced with HLSL, Cg had all the features of a high level language, more functions for the programmer and it also allowed to create code that was less dependent on the hardware.

Finally, GLSL (OpenGL Shading Language) was released as an alternative to the OpenGL Architecture Review Board. It was also based on C, allowing developers to make cross-platform applications that took advantage from most of the new features of GPUs. It was initially introduced as an extension to OpenGL 1.4 and it was officially included in OpenGL 2.0 in 2004.

2.2.2 GPGPU First Steps

At the beginning of the 21st century, GPUs had incredibly increased their programming. However, they had only been used for programming graphics applications up to that moment.

The first time they were used as general purpose devices was when some researches from the scientific sphere tried to compute more common algorithms with this platforms. In contrast to the conventional implementation of a CPU algorithm, GPU algorithms need some program layers to restructure incoming data, instructions and operators into geometry such that they behave as rendering graphics information. This way, the problem can be computed by the programmable graphics processors.

Unfortunately, developers must check that no side effects or changes occur within the graphics pipeline, as it was not designed for this purpose. These tasks required knowledge of the internal architecture, high skill and previous experience.

Algorithms	**Improvement**
Particle systems Physic simulations Molecular dynamics	2-3
Database queries Data mining Reduction operations	5-10
Signal processing Volume rendering Image processing Biocomputing	10-20
Raytracing 3D visualization	+20

Tabla 2.1: *Improvement when executing different kinds of parallel algorithms.*

Since 2003, it was possible to see codes taking advantage of GPU features.

	2008	2015
CUDA GPUs	100.000.000	600.000.000
Supercomputers in top500.org	1	75
University courses	60	840
Scientific articles	4.000	60.000

Tabla 2.2: *Evolution of CUDA.*

These programs made a clear difference between the CPU and the GPU, which would increase in the next years because of the developers that gained experience and improved their algorithms. Table 2.1 shows the differences that were observed.

2.2.3 The Arrival of CUDA

In 2003, a team of researchers from outside NVIDIA, led by Ian Buck, announced the first programming model that allowed the development of programs on a GPU using a high level language as if it were a general purpose processor. This not only meant facilities when developing GPU code, but also improved performance.

NVIDIA knew his incredibly fast hardware had to be accompanied by a software that was at the cutting edge of technology, so they invited the team to join the company and start developing the next big step in the GPGPU paradigm. As an union of hardware and software, NVIDIA released CUDA in 2006 as the first global solution for general purpose computing on GPUs. Some of the improvements were:

- Code readability.
- Easy to program and shorter development time.
- Easy to debug and optimize code.
- Independent code of the GPU.
- Complex mathematical operations and accurate results.

CUDA computing platform provided developers with a C/C++ based system along with several extensions that allowed programmers to implement parallel applications. It also offered alternatives that gave programmers the ability to express parallelism using other high level languages (Fortran, Python ...) and open standards (such as OpenACC directives).

The release of CUDA was widely accepted by scientific, academic and developer communities in general. The new parallel programming paradigm brought a

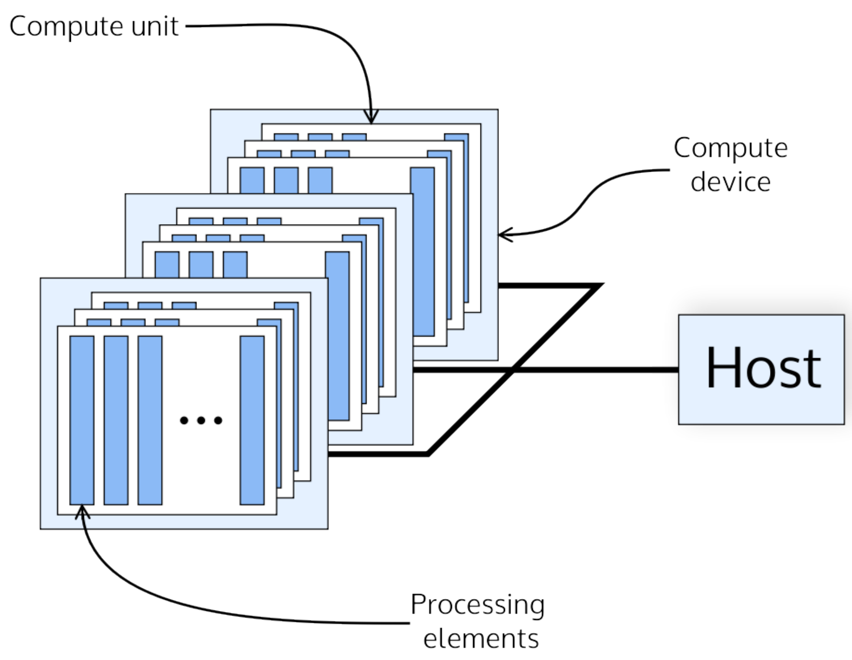


Figure 2.2: *The OpenCL model.*

number of improvements that eliminated all the difficulties encountered in the past. In fact, since its arrival day until the present, the CUDA platform has been used in more than 600.000.000 GPUs and 60.000 research applications (see 2.2).

2.2.4 OpenCL

At the end of 2008, OpenCL was released as an open alternative to propietary solutions for GPGPU. OpenCL was the product of many years of development by an open software consortium. It was originally conceived by Apple and developed in conjunction with AMD, IBM, Intel and NVIDIA; then it was given to the Khronos Group, who converted it into an open, royalty-free standard.

Unlike CUDA, OpenCL is defined as a general purpose programming standard in heterogeneous systems that can run on different architectures such as CPUs, GPUs and FPGAs. OpenCL provides an API for parallel computing and a programming language based on ISO C99 with extensions for data parallelism.

The way OpenCL operates is based on a host machine that distributes the workload between all system devices, which are called computational units. This computational units are then divided into multiple processing elements.

Although OpenCL is a valid alternative to CUDA, the distance between both is

	June 2011	June 2012	June 2013	June 2014
NVIDIA Fermi	12	53	31	18
NVIDIA Kepler	0	0	8	28
Intel Xeon Phi	0	1	11	21
ATI Radeon	2	2	3	3
IBM Cell	5	2	0	0
Hybrid	0	0	1	4
Total	19	58	54	74

Tabla 2.3: *Evolution of GPUs in TOP500.*

sometimes tremendous. If the implementation and distribution of work is perfectly adjusted to the target architecture, OpenCL performance should not be much less than CUDA. However, CUDA has not the portability of OpenCL.

2.2.5 Last Years and the Future of GPGPU

The programming of GPUs has evolved a lot in the recent years. However, its evolution needed one more step: the increment in scalability of the GPU itself.

To do that, clusters of computers arise and more devices interconnect, operating in groups that act as one graphics device. This led to the emergence of the GPGPU movement to gain momentum in the field of high performance computing.

The enhancement was not only limited to the appearance of servers and workstations: it also allowed the raise of the number of heterogeneous supercomputers that incorporated the latest generation of GPUs as coprocessors that were in charge of part of the processing work. Table 2.3 shows the evolution of graphics coprocessors in the TOP500 supercomputers list in the last four years.

The change to the GPGPU model is relatively recent, so there is still a long way to go. GPUs offer several orders of magnitude greater performance than the CPU when large amounts of data have to be processed, so they are positioned as an alternative to traditional processors and could be considered as the computing engine for the future.

Programming on Architecture Graphics Using CUDA

Once seen the increase in the popularity of the graphics programming and the importance of the GPGPU (General-Purpose computation on Graphics Processing Units), we are going to focus on the main model, CUDA, and the hardware platform that executes it.

Therefore, this chapter presents the main concepts of the graphical programming with CUDA and the highlight parts of the hardware along with the close relationship between hardware and software. Finally, the evolution of the architecture by generation is explained too.

3.1 CUDA (Compute Unified Device Architecture)

CUDA [39] is a parallel computing platform and programming model invented by NVIDIA that allows the programmer to deploy task and data parallelism in three different levels: software, firmware and hardware.

3.1.1 Software

The first level, software, has diferent ways of writing code and executing it on the GPU.

- **Programming language:** Although C/C++ is the most usual high-level language for developing code on CUDA, there are also APIs (Application Programming Interface) for other popular languages like Fortran, Java and Python.
- **Optimized libraries:** There are many libraries that allow us to perform GPU-accelerated code with just a few lines of code (cuBLAS, cuFFT, Thrust, etc.).
- **Compiler directives:** Another possibility for accelerating applications is to use standard directives with an open initiative called OpenACC. Programmers identify the data parallelism within the code through simple compiler directives, moving the bulk of the parallelization effort to the compiler. However, this automatic approaches have always a performance payoff.

3.1.2 Firmware

NVIDIA offers a driver that is compatible with the one responsible for rendering. This driver has simple APIs for controlling the memory, the device and more.

3.1.3 Hardware

Lastly, CUDA provides the programmer with the possibility of using the GPU for general purpose programming by means of a large amount of heterogeneous cores inside multiprocessors which are enveloped by a memory hierarchy. This point is explained in more details in section 3.3.

3.2 Programming Model

In the next paragraphs, the CUDA programming model is presented, taking C as the baseline language. CUDA is an extension of the C language which supplies tools for parallel programming on GPU. In this model, the GPU acts like a coprocessor and only executes a minimal fraction of code, the rest is handled by the CPU. This process is transparent for the developer due to the CUDA compiler driver (NVCC) and divides the code in two sections:

1. With the **GPU fraction** generates *PTX* [1] code files. This code is compatible with different devices so that it is decoupled from hardware implementation.
2. The **CPU part** is parsed to C compiler code in order to create object files. On Linux, we can use GCC (GNU Compiler Collection). On the other hand, CL (the Microsoft Visual Studio compiler) can be used on Windows platform.

Then the linker builds a CPU-GPU executable with the files of both parts. For NVCC to be able to divide the code, it is necessary to introduce new syntax elements are used by the programmer to define kernels. **Kernels** are C functions that contain code for one thread only, then this code is executed on multiple threads in the graphics device automatically. These threads are very thin and the context switch is immediate.

3.2.1 Processing Levels

One of the syntax elements used to define kernels is the `__global__` declaration specifier. In addition, the number of threads of each kernel is indicated within «`<...`»> through two parameters. A thread is identified within the kernel in response to the following hierarchy.

1. The **threads are organized in blocks**. Each thread has an identifier that is accessible within the kernel by means of the built-in `threadIdx` variable.
2. Likewise, these **blocks are grouped within a grid** and, like the threads, to each block is given a unique identifier within the kernel, `blockIdx`.

Both grid and thread blocks can be unidimensional, bidimensional or tridimensional and their size is indicated by the programmer under certain limitations.

[1] PTX is a low-level Parallel Thread eXecution virtual machine and instruction set architecture (ISA).

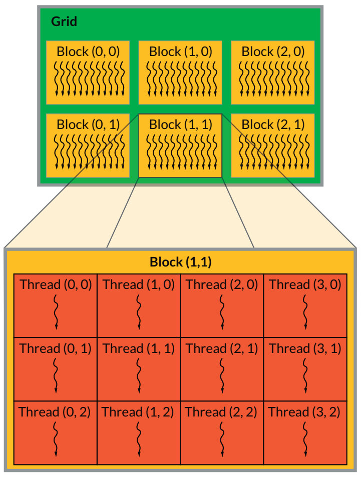


Figure 3.1: *Graphical representation of a grid with six thread blocks, each one composed of 12 threads. NVIDIA Corporation [39]*

The dimension of the thread block and the grid are accessible within the kernel through variables `blockDim` and `gridDim` respectively. This hierarchy gives to CUDA an important feature: the ***scalability***. [2]

In addition, threads are grouped within 32 elements groups called ***warp*** [3], that is the **atomic execution unit**, and they are executed in unpredictable order although they could be synchronized if this is necessary.

A warp executes **one common instruction at a time for all threads**, therefore to obtain the maximum efficiency is necessary that all threads within the warp have the same execution path. If due to data-dependence the execution path of a **warp is bifurcated**, the execution of **each branch is serialized** disabling the threads that doesn't participate on each branch. When all paths complete, the threads converge back to the same execution path. **This serialization** of the execution **only occurs within a warp**, two differents warps are able to execute distinct paths simultaneously. In the same way **blocks are executed in free order** too but in contrast they **can't be synchronized**. In addition, a thread is able to communicate only with other threads within the same thread block. All those details have to be handled with care by the programmer to guarantee the corretness of the parallel code.

[2]The code is able to run on any number of cores without recompiling.
[3]The number of threads per warp could change on future generations of GPUs.

3.2.2 Streams

Since the appearance of the second generation of graphics cards, CUDA allows to execute **kernels concurrently by means of streams**. A stream is a sequence of commands that execute in order. The execution of these commands are out of order with respect to other streams, although CUDA provides functions to synchronize them.

By default, all kernels are executed within the **same stream**. To create a new stream, CUDA C offers a new data type, `cudaStream_t`, and a new constructor, `cudaStreamCreate()`. The next code is an example of an array with three streams:

```
cudaStream_t stream[2];
for (int i = 0; i < 2; ++i)
        cudaStreamCreate(&stream[i]);
```

Kernels are **assigned to a stream** through the ***fourth parameter of the kernel launch***. The four parameters are:

1. Amount of **thread blocks** into a grid.
2. Number of **threads** within a thread blocks.
3. **Shared memory** allocation size per thread block in bytes.
4. **Stream ID**.

The maximum amount of concurrently streams depends on the generation (16 streams for Fermi and 32 for Kepler). The details of stream concurrence is explained in Section 3.4.3.2 with more detail.

3.2.3 Processing Flow

As already mentioned in section 3.2, on CUDA, the GPU (device) acts like a coprocessor of the CPU (host) but with its own memory. Because of that, it is necessary to move the data from host memory to device memory and vice versa. As a result CUDA model has a simple processing flow composed of three steps [17]:

1. Copy the **input data from host memory** to device memory.
2. Load the **program on GPU** and run, the **data are placed in cache memory** to enhance the performance.
3. Move the **results from GPU memory** to CPU memory.

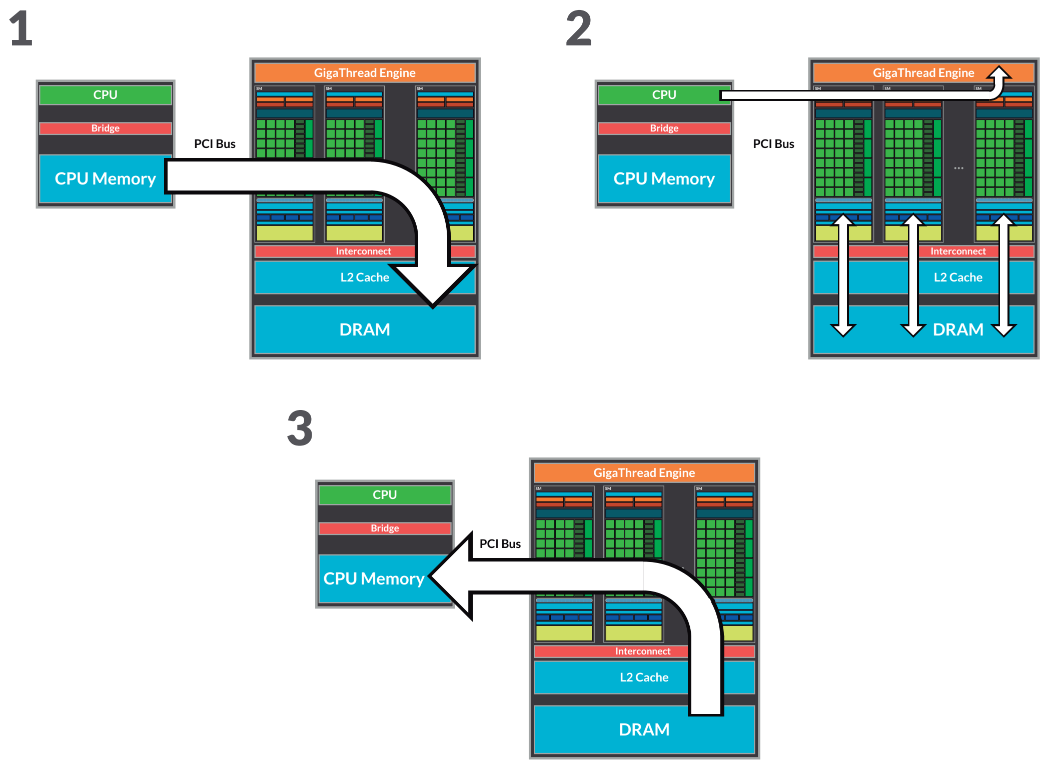


Figure 3.2: *CUDA processing flow.*

Although this schema is still valid, now it can be simplified due to unified memory, calls recently introduced in the CUDA API.

3.3 Hardware Model

To handle the thread hierarchy, CUDA implements a new hardware model that is kept on successive generations with small but powerful modifications. This model is described in this section in a general way, the specific features of each architecture generation is explained in section 3.4.

The NVIDIA GPU architecture follows a SIMT paradigm (Single-Instruction, Multiple- Thread). This architecture is akin to SIMD (Single Instruction, Multiple Data), where the instructions specify the execution and branching behavior of a single thread. This device is consisted of an **array of Streaming Multiprocessors** (SMs) with each one having **many cores**. As a result, this model along with the software model allows fine grained parallelism.

In addition, this structure is enveloped by a **memory hierarchy**. The GPU

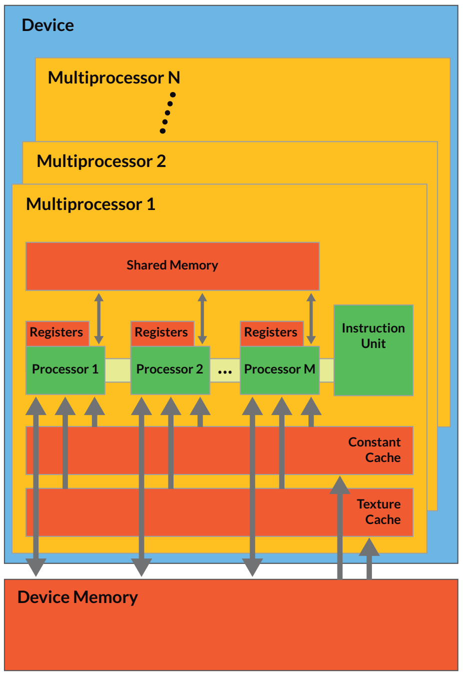


Figure 3.3: *Hardware model of CUDA. NVIDIA Corporation [39]*

has three memory layers on-die for each multiprocessor. From fastest to slowest, we have:

- A **register bank** replicated on each of the multiprocessors, which is distributed among the active threads.
- The **shared memory** is as fast as register bank. It is shared by all cores within each multiprocessor, and executions threads can communicate using it.
- In addition, the GPU has two read only memories not so popular: the constant cacge and the texture cache.

If this memory is insufficient, the GPU counts with the presence of **global memory** common to all multiprocessors. This memory is a SGRAM (Synchronous Graphics Random Access Memory), in particular it is a GDDR5 (Graphics Double Data Rate type 5).

It is three times fastest than the RAM memory of the CPU although is **500 times slower than shared memory**.

Arquitectura	G80	GT200	GF100	GK110 (k20)	GK110 (k20x)	GK110 (k40x)	GK210 (k80)	GM204 (GTX 980)
C.C.C.	1.0	1.2	2.0	3.5	3.5	3.5	3.7	5.2
Launch year	2006	2008	2010	2012-13	2013	2013-14	2014	2015
TPC	8	10	4	-	-	-	-	-
SM [SM/TPC]	16 [2]	30 [3]	16 [4]	13	14	15	(2x) 15	16
Int and fp32 [cores/SM]	128 [8]	240 [8]	512 [32]	2496 [192]	2688 [192]	2880 [192]	(2x) 2880 [192]	2048[128]
Fp64 [cores/SM]	0[0]	30 [1]	256 [16]	832 [64]	896 [64]	960 [64]	(2x) 960 [64]	64 [4]
LSU [cores/SM]	0 [0]	0 [0]	256 [16]	416 [32]	448 [32]	480 [32]	(2x) 480 [32]	512 [32]
SFU [cores/SM]	32 [2]	60 [2]	64 [4]	416 [32]	448 [32]	480 [32]	(2x) 480 [32]	512 [32]
Warp Scheduler per SM	1	1	2	4	4	4	4	4
32-bit register per SM	8K	16K	32K	64k	64k	64k	128k	64k
Shared memory per SM	16KB	16KB	16KB + 48KB	16KB + 32KB + 48KB	16KB + 32KB + 48KB	16KB + 32KB + 48KB	16KB + 32KB + 48KB + 80KB + 96KB + 112KB	96KB (2x48KB)
Cache L1 per SM	None	None						None
Cache L2	None	None	768KB	1.5MB	1.5MB	1.5MB	(2x) 1.5MB	2MB

Tabla 3.1: *Summary table with the main features of several models from each hardware generation.*

3.4 Evolution of the Architecture by Generations

To identify the different models of the architecture, NVIDIA gives an internal version number for each device generation. This number, called CUDA Compute Capability (C.C.C.), is used by applications to determine at runtime which hardware features and/or instructions are available on the present GPU. The C.C.C. is formed by two numbers (x.y):

- The first is the **major version number** and it determines the **generation**: 1 for Tesla architecture, 2 for Fermi, 3 for Kepler and 5 for Maxwell.
- The **incremental improvement** to the core **architecture** is represented by the second number or **minor version number**.

In order to select the best core architecture for the problem, the main features of the different generations are explained below.

3.4.1 The First Generation: Tesla (G80 y GT200)

This is **the first** generation that unified the vertex shader with the pixel shader for their usage on GPGPU applications. This **CUDA architecture** was launched in

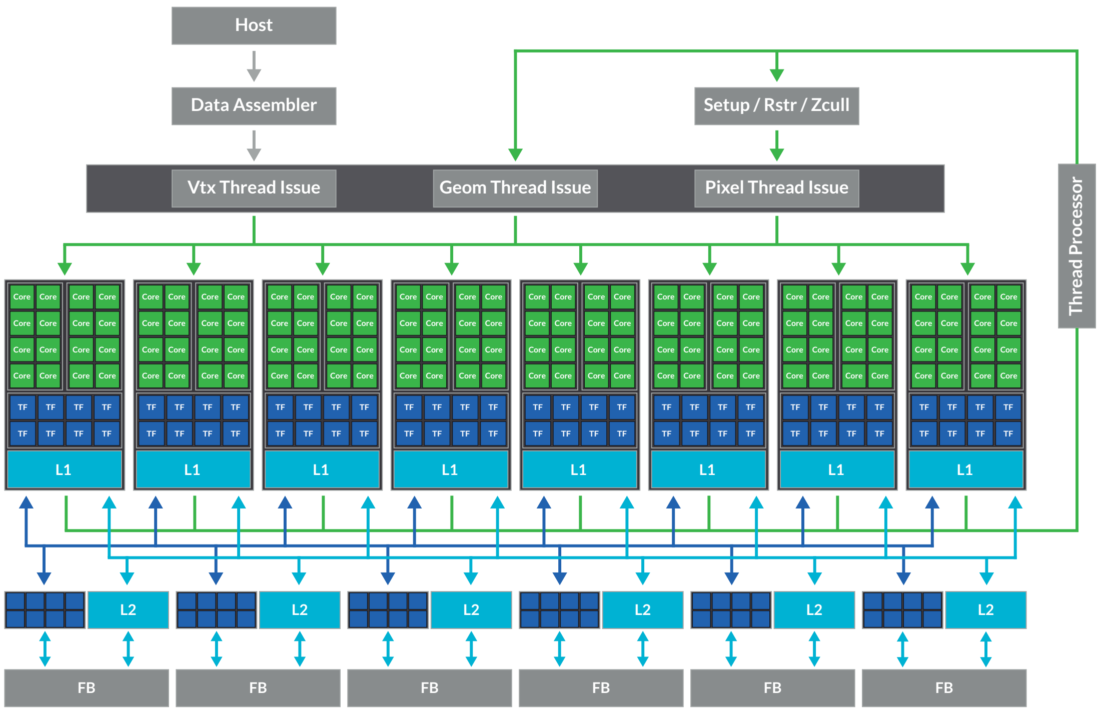


Figure 3.4: *GeForce 8800 GTX (G80) block diagram. NVIDIA Corporation [28, pg. 02]*

2006 and caused by that time a dramatic change in the graphics pipeline, going over from a lineal pipeline to a loop one. To make this possible, NVIDIA buildt the architecture described below.

Each graphic card Tesla G80 has eight Thread Processing Cluster (TPC), these in turn have two SMs with eight cores each, for a total of **128 scalar processing cores**. In addition, this cores support the dual issue of a scalar MAD and a scalar MUL operation. In the memory section the G80 is equipped with **8K registers** of 32-bits and **16KB of shared memory per SM**. This architecture can be seen in Figure 3.4, where we illustrate the GeForce 8800 GTX (G80) block diagram.

Inside of this generation NVIDIA improves the architecture creating the GT200. The main enhancements are listed below:

- **A rise in the amount of cores.** The number of blocks TCP is raised from 8 to 10, as well as increasing the amount of SMs per TCP to three. Due to this, the number of cores increased to 240.

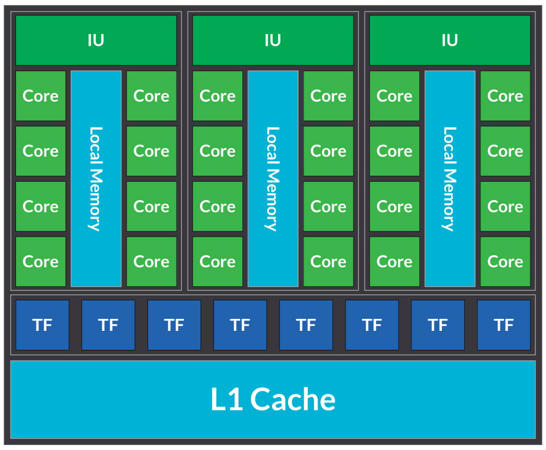


Figure 3.5: *Thread Processing Cluster of GT200. NVIDIA Corporation [29, pg. 13]*

- **More threads per chip.** The software limitation on G80 only allows 768 threads per SM whereas the GT200 accepts until 1024 threads.
- **Bigger register file size.** The register bank is doubled to 16K registers per SM.
- **Double-precision floating-point support.** One core for fp64 operation is added on each SM.
- **Shared memory improved.** Hardware memory access coalescing was added to improve memory access efficiency.

In Figure 3.5 is visible the three SMs inside a TCP of a GT200 revealing that in this time the increment of cores is produced by rising TCPs instead of the number of cores per SM.

3.4.2 The Second Generation: Fermi (GF100)

The TCP disappears and Nvidia makes a new hardware block, called Graphics Processing Clusters (GPC), which encapsulates all key graphics processing units. Inside of this hardware block there are four stream multiprocessors.

In contrast to the intergenerational enhancements of Tesla, on Fermi, NVIDIA decided to reduce the number of SMs and grow in the number of cores per multiprocessor. Thus, Fermi has three distinct types of cores:

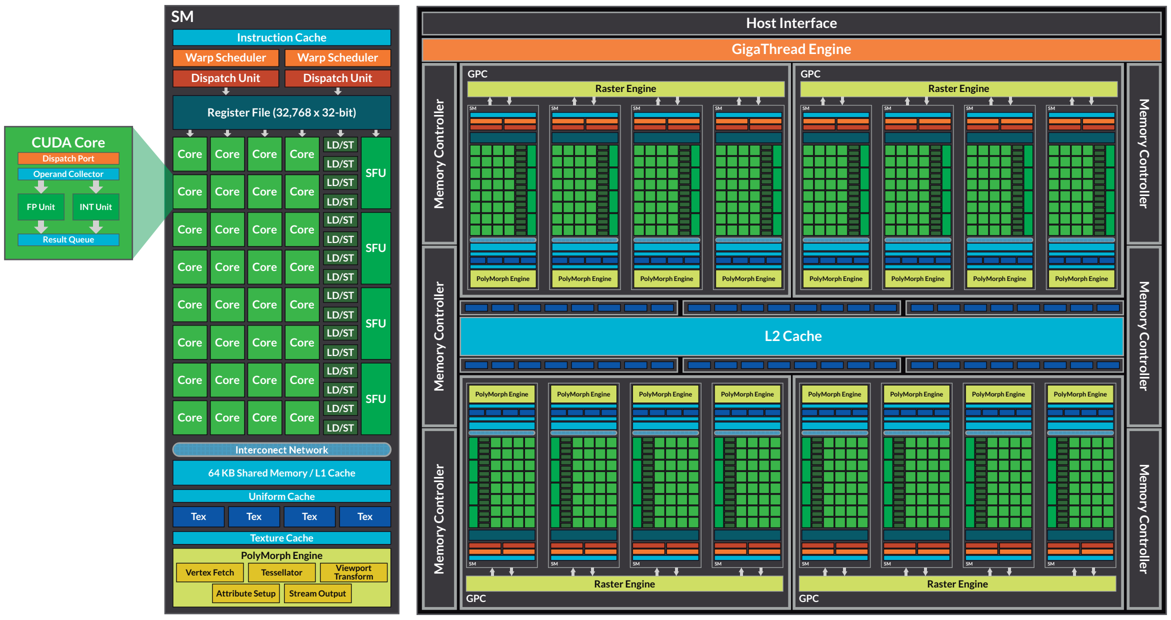


Figure 3.6: *GF100 block diagram and Stream Multiprocessor detail. [31, pg. 11 and 16]*

1. **Int and floating points units**. 32 cores per SM redesigned for optimizing 64-bit int operations. These cores are used for both simple and *double precision* operations [4].

2. **Load/Store units**. For Load/Store operations 16 cores are incorporated allowing source and destination addresses to be calculated for sixteen threads per clock.

3. **Special Functions Unit (SFU)**. Four cores are added for quick calculation of complex functions such as sin, cos, reciprocal and square root, although they are not as accurate as their CPU versions.

In addition, the GF100 has **two warp schedulers, each with an instruction dispatching unit**. This configuration allowed to launch two concurrent warps, and the schedulers did not need to check for dependencies within the instructions in the stream.

One of the main improvements over the previous generation is the **memory hierarchy**. Each Fermi's SM has 64KB of on-die memory that it is configurable in two modes: **16KB of shared memory and 48KB of L1 cache and vice versa**. The first mode optimize the algorithms where data addresses are not known beforehand, while the second is the best mode for algorithms with well defined memory

[4]In this case Fermi can run 16 fp64 operation only.

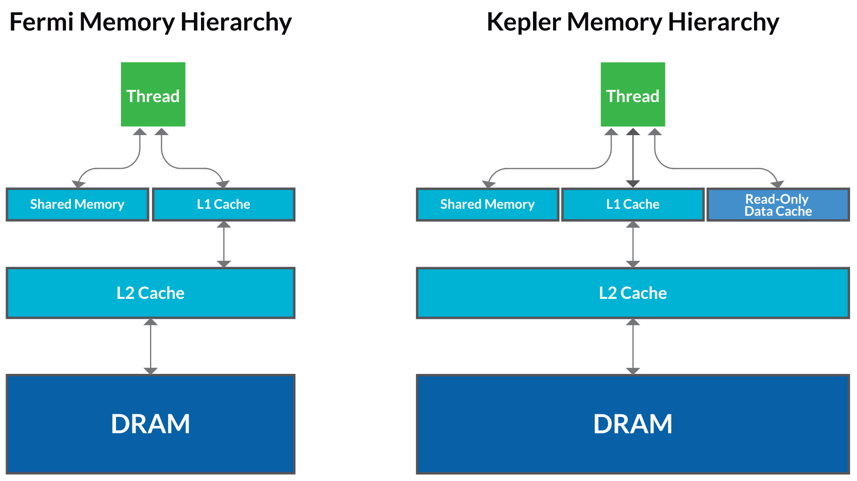


Figure 3.7: *Fermi and Kepler memory hierarchy. NVIDIA Corporation [31, pg. 19] and [36, pg. 13]*

accesses. Moreover, this generation incorporates **768KB of L2 cache shared by all stream processors**. In the left side of Figure 3.7 the diagram of this hierarchy is visible.

3.4.3 The Third Generation: Kepler (GK110 y GK210)

Following the trend introduced by Fermi, **Kepler increases the number of cores per SM** and reduces the amount of multiprocessors. Even though the GK110 is not the first chip of Kepler architecture, this section is centered in the GK110 and newer ones as they are the most widely used.

The quantity of cores per SM is the same in the distinct incremental improvement of the architecture, although the number of stream multiprocessors changes from one to another. Thus, Table 3.1 shows the differents versions and its main features.

The Kepler's SMs (called SMXs) have **192 single precision CUDA cores**, and each core has fully pipelined floating-point and integer arithmetic logic units. In addition, these SMs increase the **double-precision** computation capacity with **64 dedicated units**. More over, the GK110 has **32 LD/ST units**, doubling the amount of load and store units available in the Fermi architecture. Finally, the SMXs have **32 Special Function Units** (SFU).

Each SMX has **four warp schedulers with two dispatch instruction units** each. This allows up to eight warps to be issued and executed concurrently.

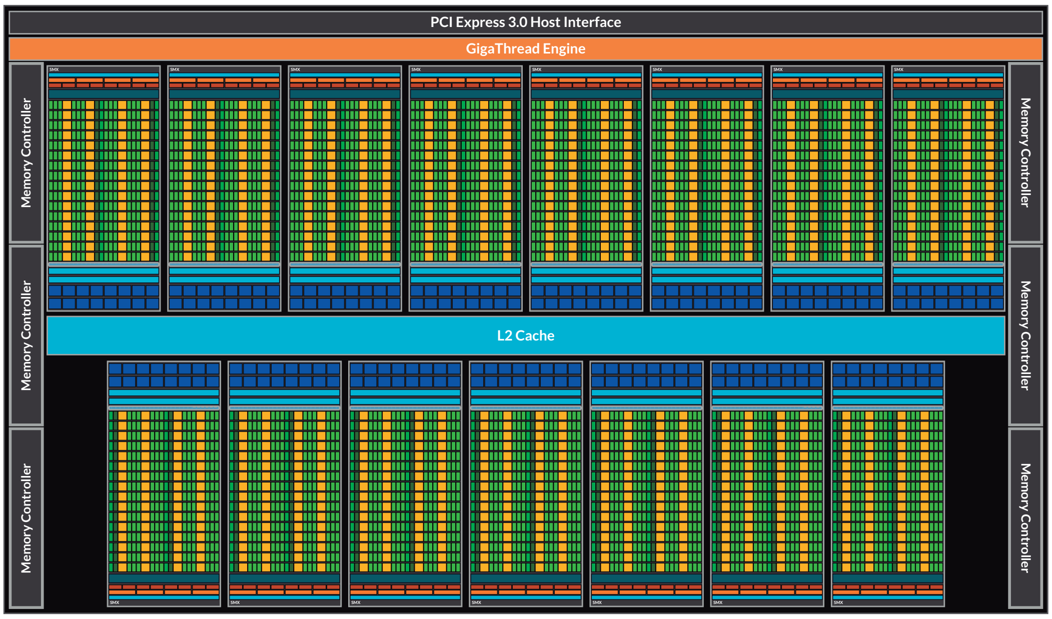


Figure 3.8: *Kepler GK110 full chip block diagram. NVIDIA Corporation [36, pg. 06]*

Kepler also follows the memory hierarchy of Fermi, although the **texture memory** is now accessible for GPGPU as **only-read memory of 48KB**. In addition, this generation improve all layers of memory:

- **Register Bank**. The amount of 32-bit register per multiprocessor grows until **64K**.
- **Shared Memory and L1 cache**. Apart from to the configuration modes of shared memory were seen in Section 3.4.2, a **new mode** is added in this generation: **32KB for both**.
- **L2 cache**. The amount of memory in this layer is doubled to **1536KB**. Additionally, the L2 cache on Kepler offers up to **2x of the bandwidth per clock** available on Fermi. [36]

The **GK210** and GK110 have their features explained in Section 3.4.3.1 and Section 3.4.3.2. One as much as the other are Kepler architectures but the GK210 has more resource on-chip than GK110. Thus, both chips share the **same amount of core per SMX** but the GK210 has **128K register of 32-bit per SMX** and **128KB of shared memory/L1 cache** with the configurations below:

- 112KB shared memory + 16KB L1 cache

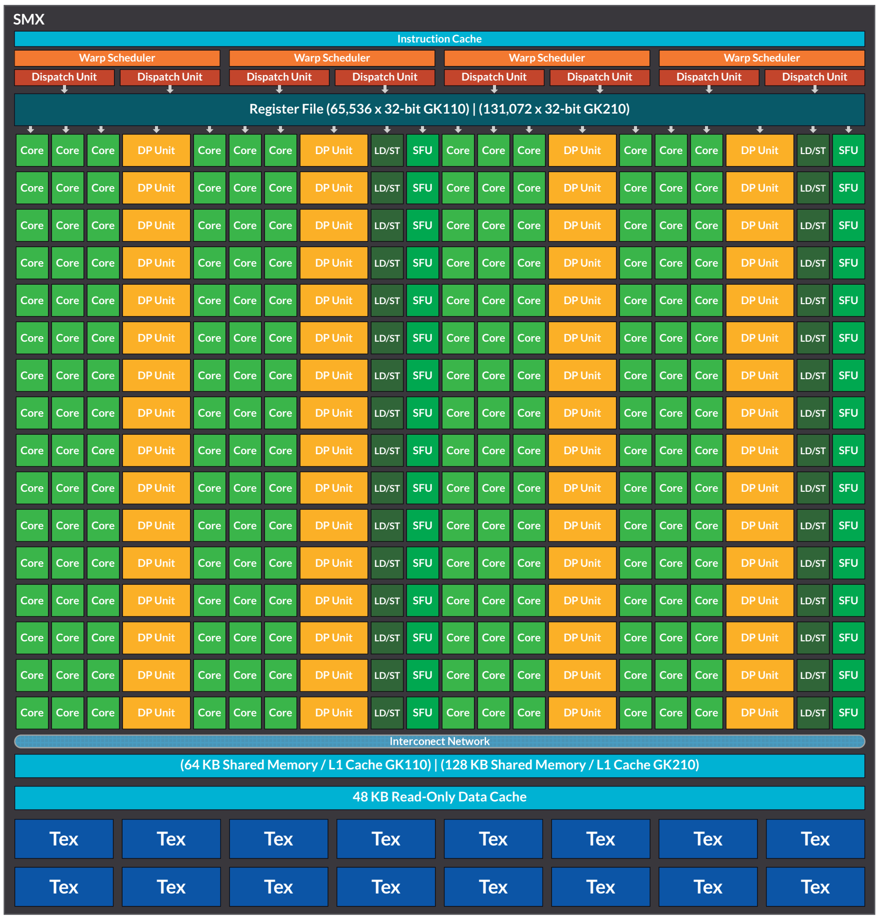

SMX
Instruction Cache
Warp Scheduler
Dispatch Unit
Register File (65,536 x 32-bit GK110) | (131,072 x 32-bit GK210)
Core
DP Unit
LD/ST
SFU
Interconect Network
(64 KB Shared Memory / L1 Cache GK110) | (128 KB Shared Memory / L1 Cache GK210)
48 KB Read-Only Data Cache
Tex

Figure 3.9: *SMX with 192 single-precision CUDA cores, 64 double-precision units, 32 SFU and 32 LD/ST units. NVIDIA Corporation [36, pg. 08]*

- 96KB shared memory + 32KB L1 cache
- 48KB shared memory + 80KB L1 cache
- The anterior amounts reversed.

3.4.3.1 Dynamic Parallelism

Until the GK110 was created, the GPU acted like CPU a coprocessor with high speed-up factors, but low autonomy. Now, with **dynamic parallelism, the GPU can generate new work for itself**. It does not need to interrupt and wait the launch of new kernels in the CPU, create the events and threads required to control dependencies, synchronize the results and control the task scheduling [3]. Figure 3.10 shows an example about how dynamic parallelism behaves releasing work

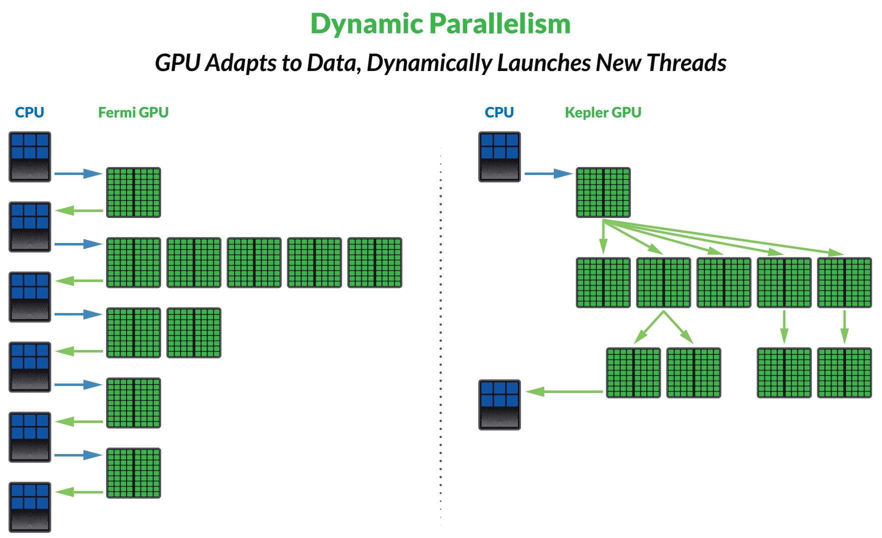


Figure 3.10: *With Dynamic Parallelism the GPU can generate new work for itself. NVIDIA Corporation [36, pg. 15]*

from the CPU.

This new feature allows the programmer to use recursive techniques in its algorithms. Due to this, the developer is able to make **algorithms that were impossible to achieve on FERMI** such as quicksort, nested loops with differing amounts of parallelism or even dynamically setting up a grid for a numerical simulation focusing in the interesting zones without an expensive pre-processing.

On Fermi, Only the host sends a grid to the CUDA Work Distributor (CWD) and this distributes the blocks among the differents SM. On Kepler, it is necessary a new unit for **the management of both device and host grids**. This component, called **Grid Management Unit (GMU)**, processes the grids received from CPU and GPU and sends them to CWD. Then, the work distributor, which accepts up to 32 grids, sends the blocks to the SMX. In addition, the GMU can pause the dispatching of new grids due to the two-ways link. In Figure 3.11 it can seen both Fermi and Kepler workflow.

3.4.3.2 Hyper-Q

On Fermi until 16 streams could be launched concurrently, but they are implemented underneath using a single queue, only the end of a stream and the start of other could be executed at the same time. On Kepler until **32 streams can be really executed concurrently** due to that each stream is managed independently on

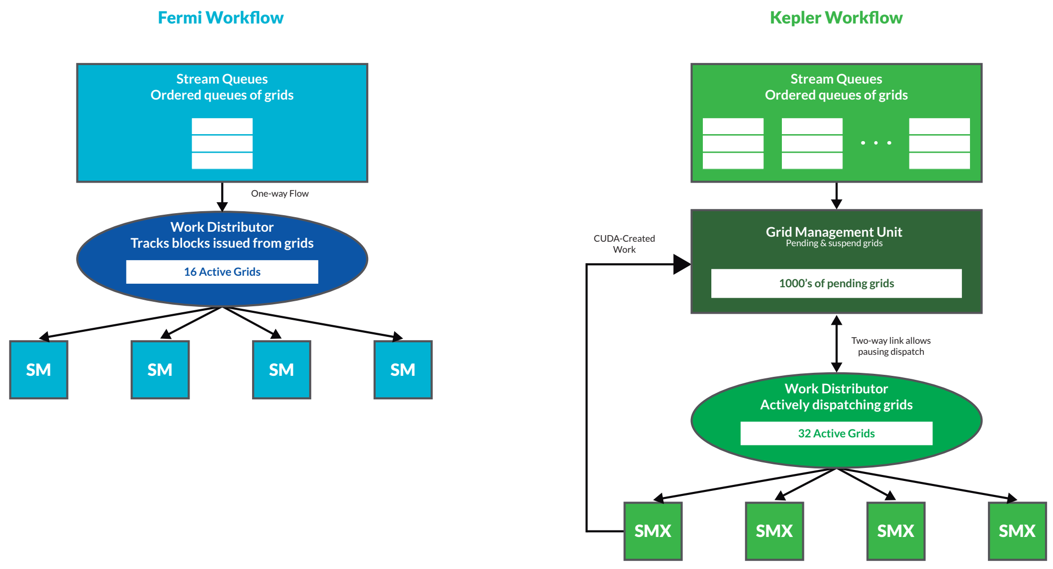


Figure 3.11: *Fermi (left side) and Kepler (right side) workflow. NVIDIA Corporation [36, pg. 19]*

a different hardware queue. In addition, this allow for executing a streams in parallel that other stream coming from the same or other CUDA program, MPI process or POSIX thread.

3.4.4 The Fourth Generation: Maxwell (GM204)

The new generation is focused on maximizing the performance per consumed watt. Thus, NVIDIA has reorganized the internal components of the multiprocessors (SMMs). Now, these are splited in four part. Each CUDA cores processing block contains:

1. **32 int and floating points units** (128 per SMM).
2. **1 double precision unit** (4 per SMM).
3. **8 Load/Store units** (32 per SMM).
4. **8 Special Functions Unit (SFU)** (32 per SMM).

In addition, each split contains a warp scheduler, which is capable of dispatching two instruction per warp at every clock. This configuration aligns with warp size, making it easier to use efficiently.

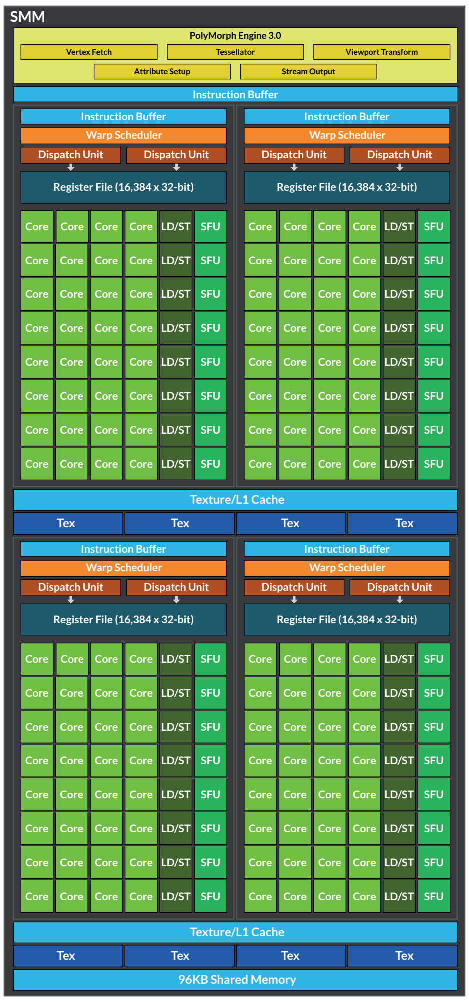


Figure 3.12: *GM204 SMM Diagram (GM204 also features 4 DP units per SMM, which are not depicted on this diagram). NVIDIA Corporation [41, pg. 08]*

3.4.4.1 Memory improvement

The memory hierarchy has changed too, now the shared memory doesn't share the block with the L1 cache. The L1 caching function is now shared with the texture catching function. The size of shared memory grows to 96KB, although this is limited to 48KB per thread block [23]. Finally, the size of L2 cache is 2MB on GM204.

Other improvement which is implemented on Maxwell is the memory compression. To reduce DRAM bandwidth demands, NVIDIA GPUs make use of lossless compression techniques as data is written out to memory. This profit is doubled when clients, such as the Texture Unit, read later the data.

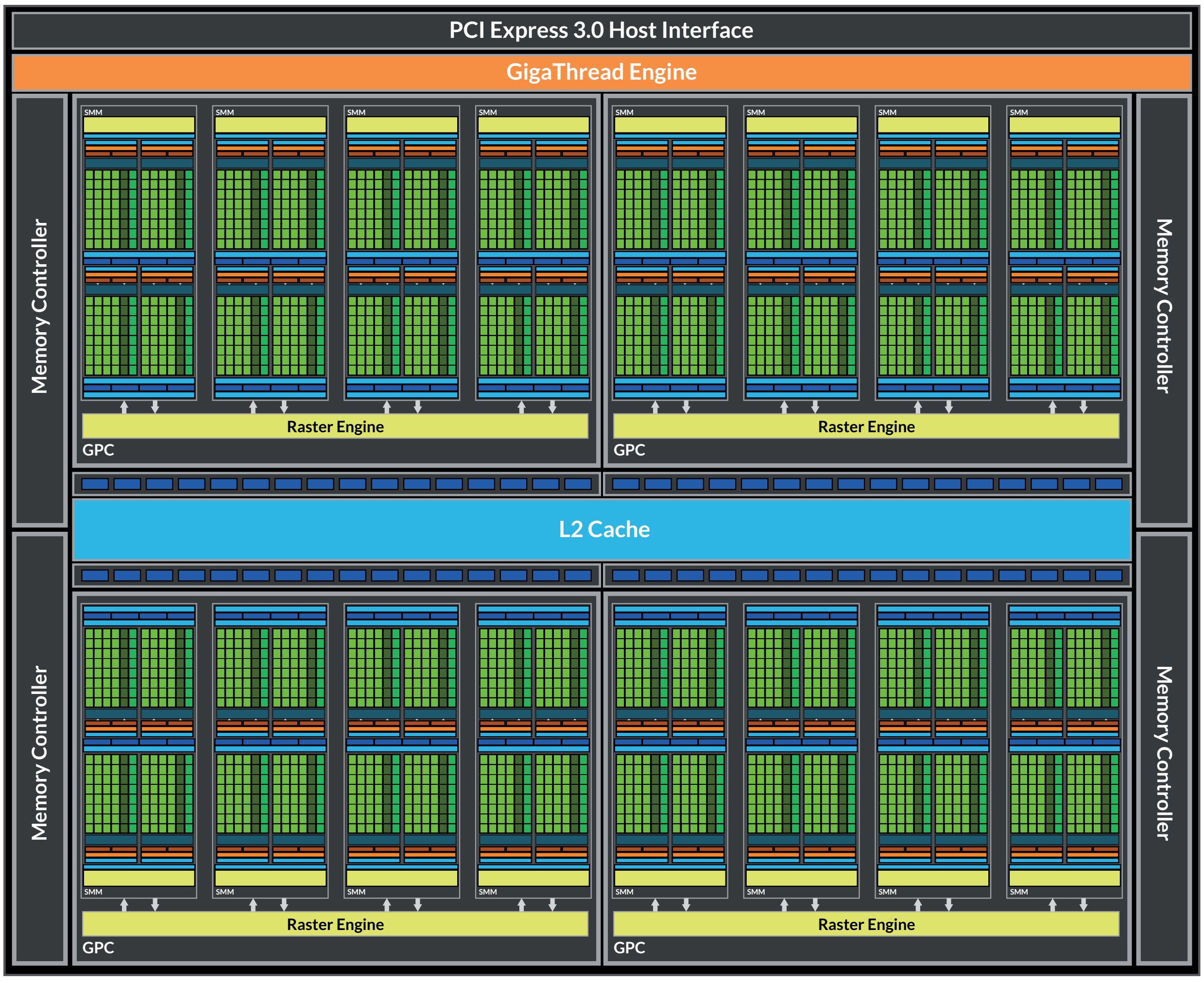


Figure 3.13: *Maxwell GK204 full chip block diagram.NVIDIA Corporation [41, pg. 06]*

3.4.4.2 Atomic operations

Maxwell introduces native shared memory atomic operations for 32-bit integers and native shared memory 32-bit and 64-bit compare-and-swap (CAS), which can be used to implement other atomic functions with reduced overhead compared to the Fermi and Kepler methods. This should make it much more efficient to implement things like list and stack data structures shared by the threads of a block [41].

GTX 480 vs Jetson TK1 vs GTX 980

4.1 Introduction

4.1.1 Dissertation Overview

NVIDIA Corporation has made a huge step into green computing. Kepler was the first generation that defines itself as an architecture concerned about efficiency. His antecessor, Fermi, was presented as a powerful parallel computing architecture. The predecessor of Kepler is Maxwell generation, which endorses the focus on efficiency.

In this section, we ilustrate this transition with a comparison between Fermi, Kepler and Maxwell. However, the devices corresponding to each generation are not only graphics cards. The Kepler one is an embedded platform designed to be extremely efficient. Next sections will introduce them and their features.

4.1.2 GeForce GTX 480

This graphics card is part of the GeForce 400 series, the one that introduced Fermi GPU architecture in early 2010 [11]. It is the oldest of the presented platforms. Fermi architecture allowed not only the raise of the performance limits of GPUs, it also added more flexibility on the programming of applications for these devices.

4.1.1 GeForce GTX 480 card.

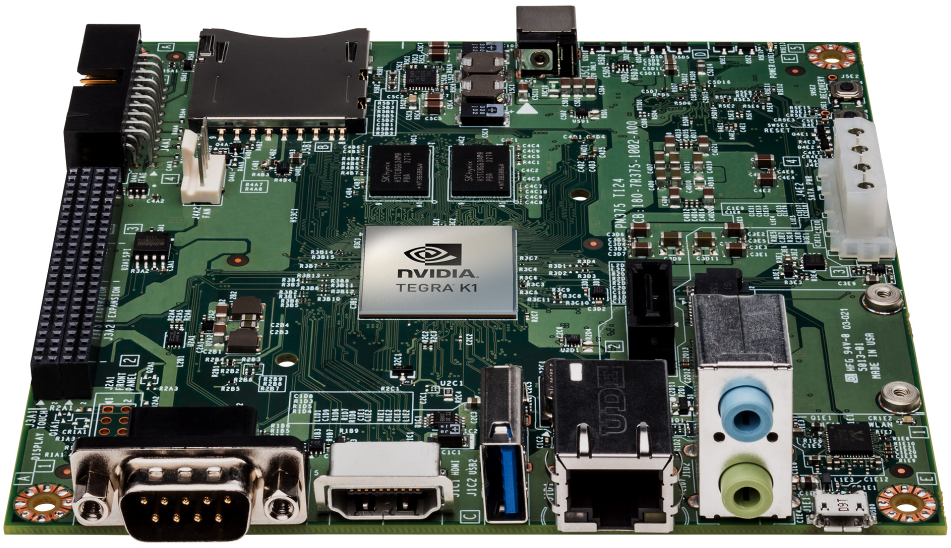
The GeForce GTX 480 is one of the most powerful devices of the GeForce 400 series. 480 CUDA cores at 1401 MHz and 1536 MB GDDR5 at 1848 MHz. The maximum temperature for the GPU is 105 degrees [12]. Theoretical peak performance is 1345 GFLOPS.

Our tests showed that, on idle, it consumes around 35 W and aproximately 65 W on a normal execution of random CUDA Samples. Some applications showed a 100 W power consumption, but it is less frequent. Minimal system power supply is set at 550 W. The idle temperature for this device is 62 degrees.

4.1.3 Jetson TK1

Jetson TK1 is described as the world's first embedded supercomputer. It stands as a computer vision platform that many has applications (autonomous robotics, mobile medical imaging, Intelligent Video Analytics, etc) [15]. Jetson TK1 aims to provide an efficient and powerful platform for embedded and mobile applications [18].

It was released on April 2014. It is powered by a Tegra K1, which consists



4.1.2 Jetson TK1 board.

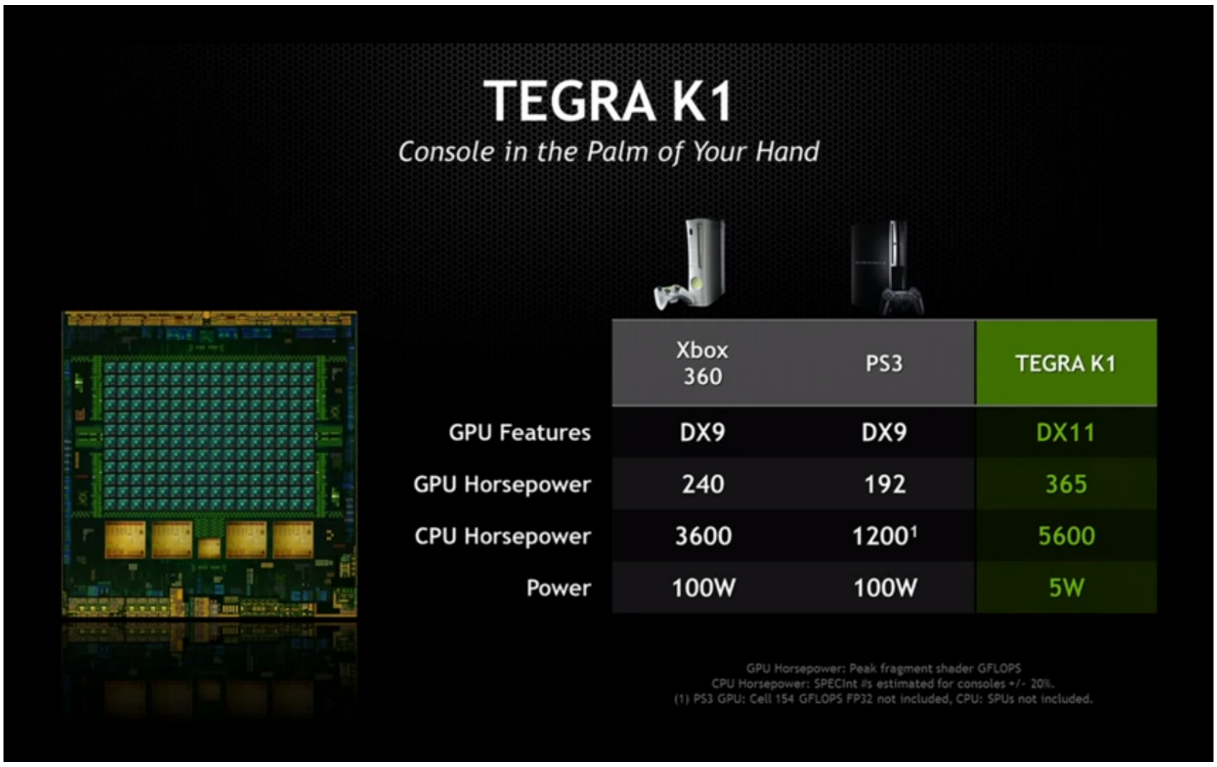
of a SoC (System-on-a-Chip) architecture integrating a GPU with 192 Kepler CUDA cores and NVIDIA CUDA 6 support and a 4-PLUS-1 Cortex-A15 ARM processor, that has better performance and is more power efficient compared to the previous generation. GPU clock can reach 852 MHz. 2GB of RAM and several ports like HDMI, USB 3.0, serial, mini PCI-e and GPIO. Peak performance is about 300 GFLOPS. The Operating System is a Linux distribution based on Ubuntu [37].

The following picture compares Jetson processor with the known PS3 and Xbox 360 gaming platforms [13]. Tegra K1 shows a power consumption of 5 watts, a remarkable difference with competitors.

On Jetson TK1 wikia it is said that the board consumes 2.2 W when it is idle, approximately 4 or 5 watts when using CPU cores and 11 watts enabling GPU cores. It also can be read that this number can raise up to 30 W when connecting peripherals.

To measure this, we used a hand-built PCB with a Shunt resistance that allows to get the electric consumption from the board power connector. This PCB was then connected to an DS-5 ARM energy probe that reads the consumption value and sends it to an application called Streamline. In addition, the Jetson TK1 Linux kernel was recompiled to add a module that reports a few more measures such as temperature and CPU activity to Streamline.

As the test applications are all graphics programs that use a window to show their results, we used a computer monitor connected to the HDMI of the Jetson and a keyboard in the USB. In the end, the power reader reported 1.12 W on idle and no



TEGRA K1

Console in the Palm of Your Hand

	Xbox 360	PS3	TEGRA K1
GPU Features	DX9	DX9	DX11
GPU Horsepower	240	192	365
CPU Horsepower	3600	1200[1]	5600
Power	100W	100W	5W

GPU Horsepower: Peak fragment shader GFLOPS
CPU Horsepower: SPECInt #s estimated for consoles +/- 20%.
(1) PS3 GPU: Cell 154 GFLOPS FP32 not included, CPU: SPUs not included.

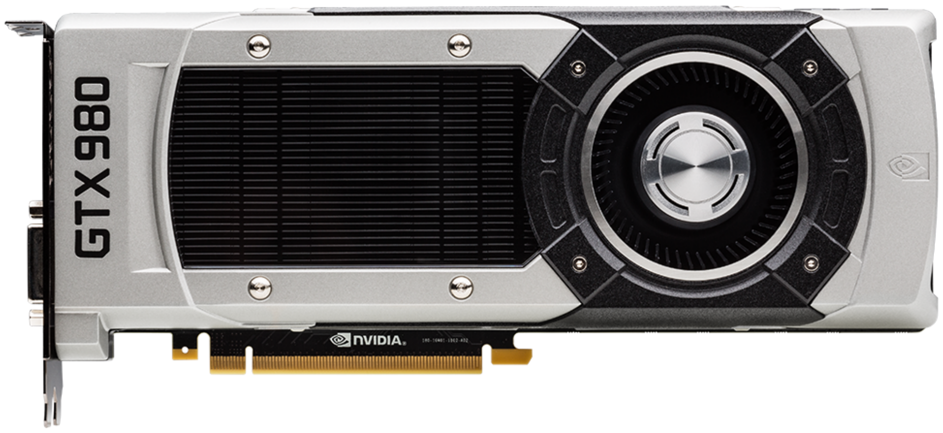
4.1.3 Jetson TK1 vs Xbox & PS3.

more than 5 W on any application execution. These numbers and the ones showed in the charts of the samples section refer all to the total consumption of the board. The tests also showed that the maximum temperature is 51 degrees [2]. Though these programs are not designed to stress the device, they provide a good example of general applications executed on the Jetson TK1 and how it would response to it. The power consumption results prove the veracity of the slide table that compares Jetson TK1 against the gaming consoles.

4.1.4 GeForce GTX 980

This graphics card belongs to the NVIDIA's GeForce 900 series. The GeForce 900 series was released as part of the Maxwell architecture in early 2014 [14]. GTX 980 came out at the end of that year and is the most modern device in our experiment.

In terms of hardware, GTX 980 has 2048 CUDA cores at a clock rate of 1126MHz (can boost to 1216MHz) and 4GB of global memory. Thus, the GTX 980 is the platform that comprises the biggest amount of hardware of all the three devices. The graphics card power maximum consumption is 165 W, which is called Thermal Design Power, and requires 500 W of system power supply. Theoretically,



4.1.4 GeForce GTX 980 card.

it can reach 98 degrees [16]. Idle power is 13.2 W, far less than the GTX 480. Our test will show the higher efficiency of the GTX 980 against the old GeForce device.

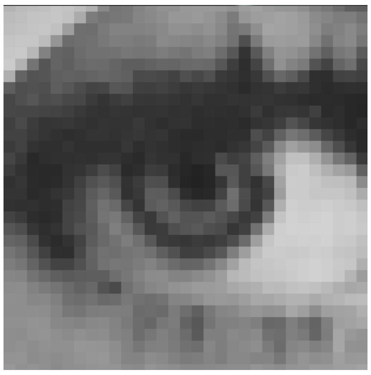
4.2 Texture filtering

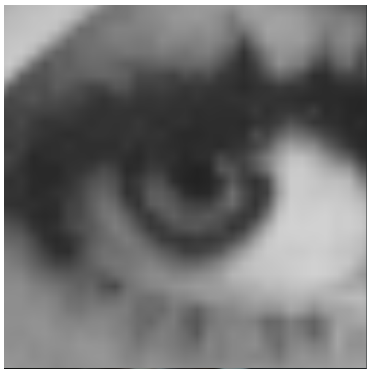
4.2.1 Description

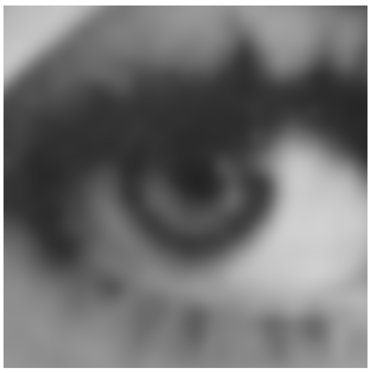
Texture filtering is a commonly known technique from the field of computer graphics which purpose consists of smoothing out textures and images. To achieve this, the algorithm averages each pixel with its neighbourhood such that the differences fade away.

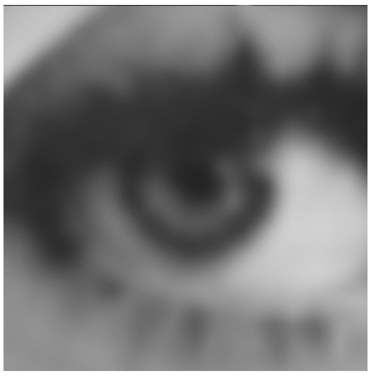
There are several algorithms to perform this task. As always, there is a trade-off between effectiveness and efficiency and each of these algorithms play with this balance betting for the first or the second part. The ones covered in this experiment are: nearest filtering, bilinear filtering, bicubic filtering, fast bicubic filtering and Catmull-Rom filtering.

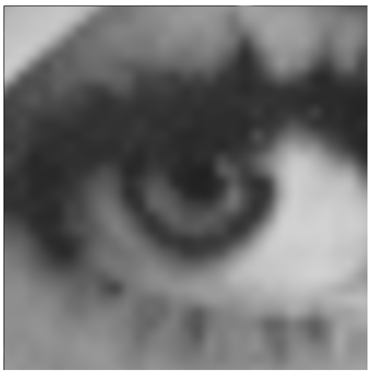
A visual example using the black and white eye part of the picture of Lena (the popular computer graphics data set) shows the differences on filtered images.


4.1.5 Nearest filtering.


4.1.6 Bilinear filtering.


4.1.7 Bicubic filtering.

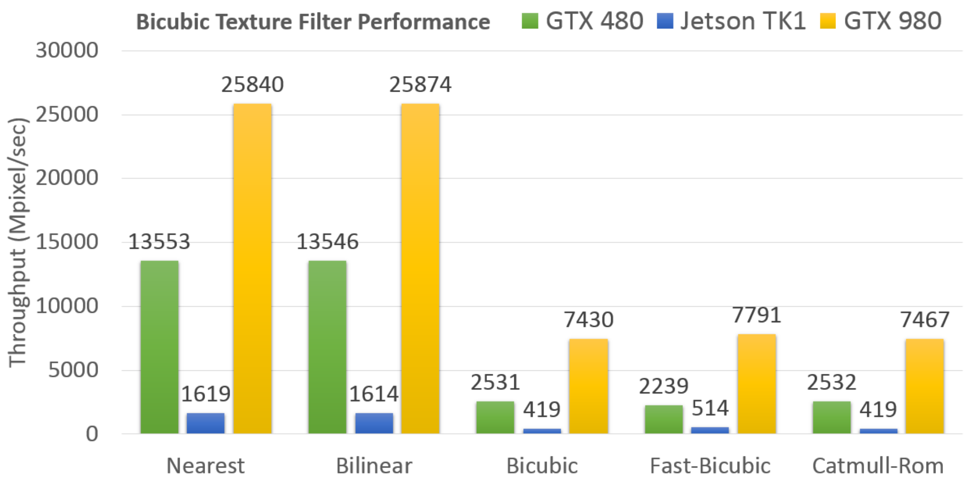

4.1.8 Fast-Bicubic filtering.


4.1.9 Catmull-Rom filtering.

Theoretically, Bicubic and Catmull-Rom produce the best results but are also the most computationally expensive. Fast-Bicubic is an optimized version of the Bicubic algorithm that has a bit less resource usage. On the other hand, the fastest algorithms are the nearest and bilinear filtering but do not perform quite as well as the others.

4.2.2 Performance

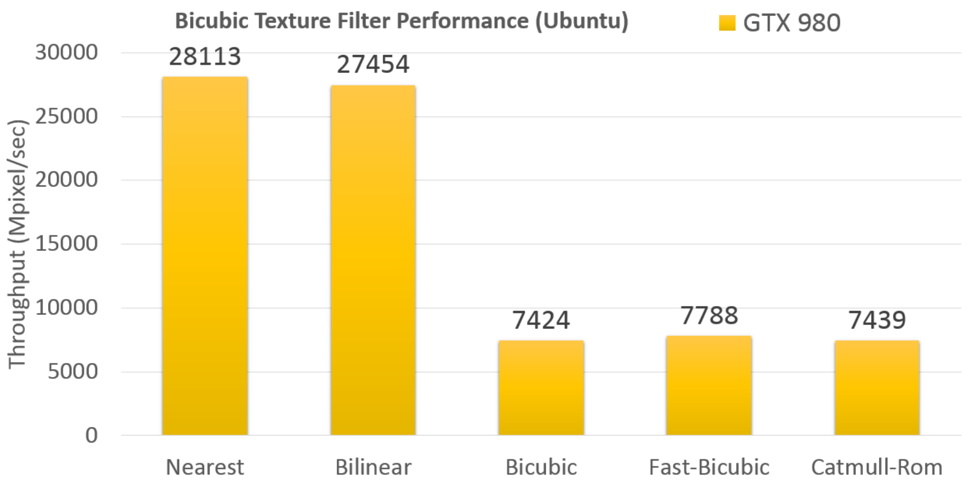
The performance of these algorithms is measured in Mpixels/sec. As previously said, the last three algorithms use a lot of resources and thus have less pixel throughput. The NVIDIA Texture Filtering sample application provides a benchmark mode that performs the computation of the selected filter a number of times. The default value for this count is 500. In practice, the benchmark mode reported unstable pixel rates in sequential benchmark tests. Because of that, the number was increased to 1.000.000 and executed 10 times. An arithmetic mean of the results was then computed. Conclusions of the experiment are contained in Figure 4.14.1.10.



4.1.10 Texture Filtering Performance (Windows).

As expected, the performance order from lower to higher values starts with the device that has less consumption (Jetson TK1) and ends with the most powerful one (GTX 980). There are two clear subsets in the chart, one containing the first two algorithms and the other one with the last three. Bicubic, Fast-Bicubic and Catmull-Rom filters performance compared to the Nearest and Bilinear filters is, in the GTX 480, approximately a 18%. This percentage is a bit higher in the GTX 980, managing to mantain a 28% of the pixel throughput. However, the Jetson TK1 has a 41%

performance in the expensive algorithms relative to the cheap ones. This makes the Jetson TK1 the device with higher scalability, as the decay in performance on work overload is superior to the GeForce devices for this example.



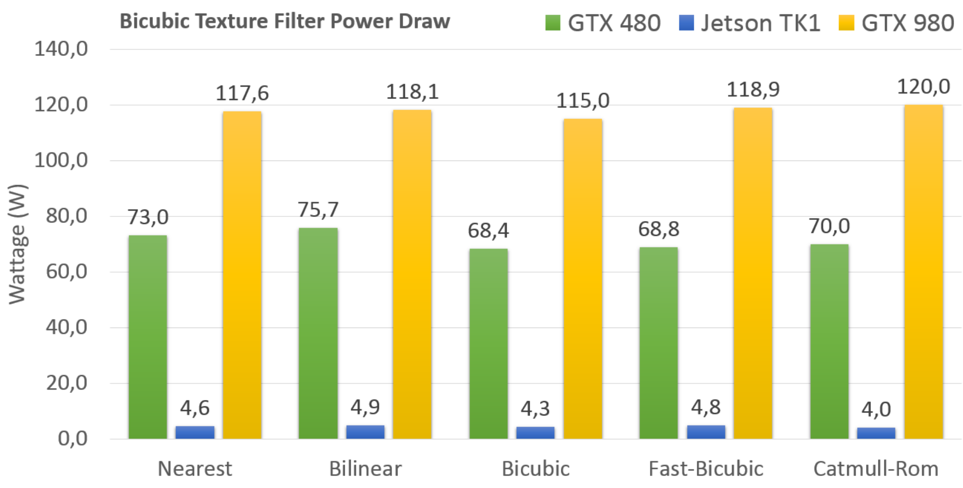
4.1.11 Texture Filtering Performance (Ubuntu).

The same CUDA sample was run in Ubuntu using the GTX 980 (see Figure 4.14.1.11). The results shown a 8% and 6% higher performance for Nearest and Bilinear filters, respectively. Despite of this, the expensive filters do not improve their throughput, having less than 1% lower performance or approximately the same pixels per second. This difference is not enough to question the importance of using the same Operating System to compare the results, as the numbers do not vary widely with the OS but more with the underlying hardware.

4.2.3 Power Draw and Heat Generation

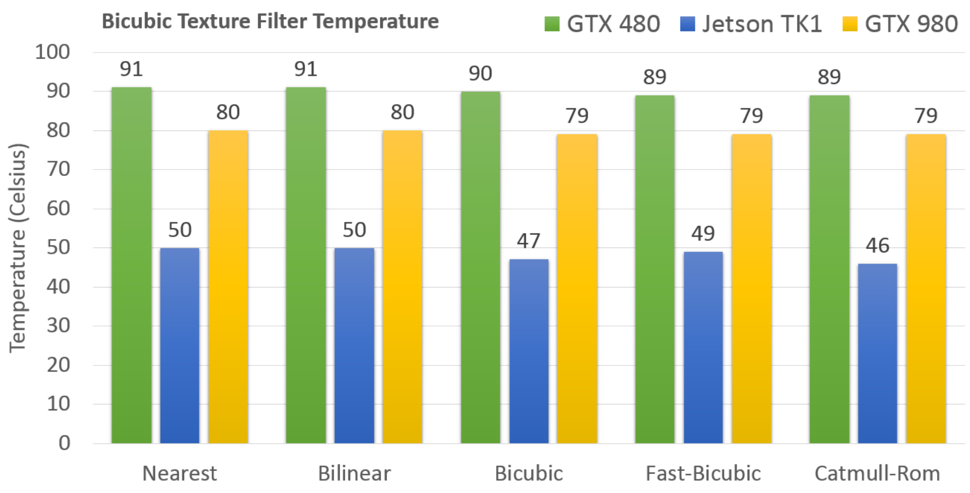
In the GTX 980, Bicubic filter has the least power consumption with 115 W. Nearest and Bilinear filters are more stable around 118 W, as well as Fast-Bicubic. Lastly, Catmull-Rom is the highest with 120 W. These differences in wattage are not very important (they are in a range of 5 W wide), meaning that the graphics card operates, in terms of power, more or less the same with each of the filters. However, the small differences reveal how expensive filters consume the most.

Bicubic filter is an exception to this rule as it has the lowest power drawback of all. It also has the least pixel processing rate (in GTX 480 and GTX 980 devices). These two indicators prove that the graphics card is not able perfectly fit the work load of the algorithm. In the GTX 480, the first two algorithms have the greatest power draw.



4.1.12 Texture Filtering Power Draw.

All the three platforms have small power differences among the different filters. In the GTX 980, the window size is 5 W, the GTX 480 has 7.3 W and the Jetson TK1 0.9 W. The device that required more power is the GTX 980 and the one consuming less is the Jetson TK1.



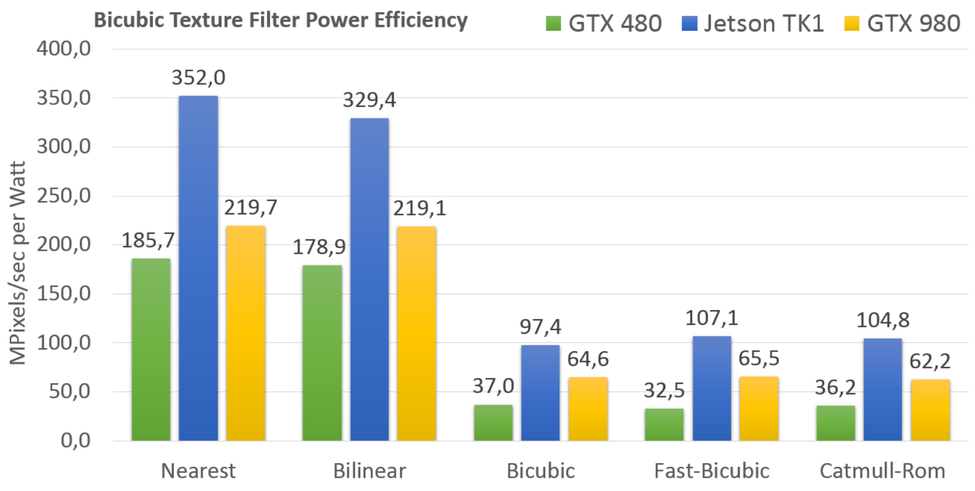
4.1.13 Texture Filtering Heat Generation.

Heat generation in this example showed that filters with less throughtput are the ones that require less power. The harder the algorithm is, the lower the temperature gets.

GTX 980 has around 12% less heat generation than the GTX 480. Both devices are cooler when computing the last filters. In general, a strong correlation occurs between the power consumption and the heat generation. Despite of this, the graphics card fan could cause the heat to perform differently, as it dynamically increases its speed depending on the work load. This assumption of the correlation between the heat and the power reflects on GTX 480 and Jetson TK1, but not on the Maxwell graphics card.

4.2.4 Algorithm Efficiency

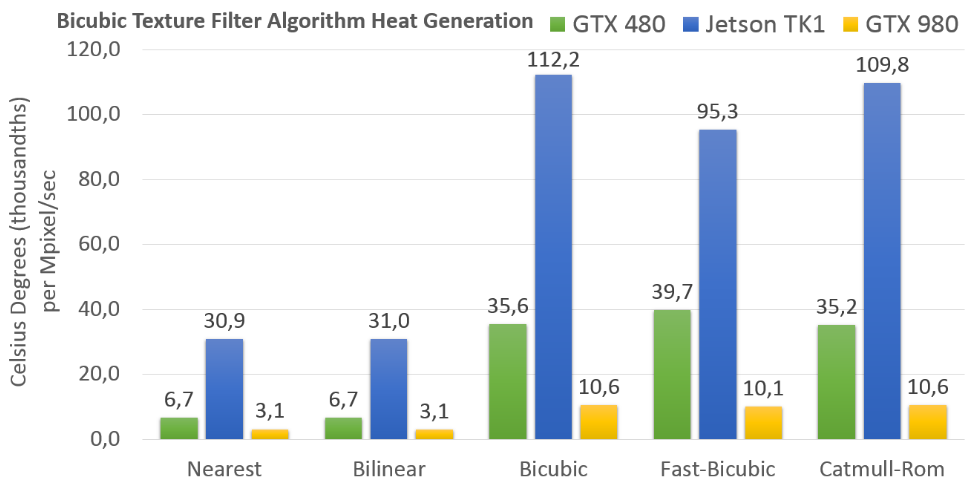
We now compare the efficiency of the devices (see Figure 4.14.1.14). GTX 980 has approximately 220 MPixel/sec per Watt for the first two algorithms while the GTX 480 gets a 180 MPixel/sec per Watt rate. The GTX 980 has higher efficiency also in the expensive algorithms, around 64 MPixel/sec per Watt against the 36 MPixel/sec per Watt of the other device. The conclusion is that, for the cheap algorithms, the GTX 980 is 20% more efficient than the GTX 480 and for the expensive ones, this difference windens to 70%.



4.1.14 Texture Filtering Power Efficiency.

Jetson TK1 has the greatest power efficiency in all filters, with a 60% and 50% higher efficiency than the GTX 980 in the first two filters and approximately a 50%, 60% and a 68% in the last three.

In the previous section, heat generation for the three devices was shown. The GTX 980 was proven to generate less heat than the GTX 480 and the Jetson TK1 again less than the GTX 980. Jetson TK1 has a very low power consumption, being just 6% and 4% of the power drawn by the GTX 480 and the GTX 980, respectively. However, the heat the device generates doesn't hold this proportions but much larger ones. Jetson TK1 generates around 53% and 60% of the heat GTX 480 and GTX 980 generates, respectively. The performance increment do not compare to this proportion. Because of this, the heat generation efficiency chart was made to show the heat generated by each performance unit.



4.1.15 Texture Filtering Heat Generation Efficiency.

Heat generation efficiency for the Jetson TK1 is now the worst of all devices. It is more efficient in terms of power, but it is fairly warmer proportionally, specially on the last three filters, with 1 degree generated for each 0.11 MPixel/sec that it achieves. Figure 4.14.1.15 shows that the most efficient device is the GTX 980, having 1 degree generated for each 0.01 MPixel/sec. Thus, the GTX 980 has a 9% of the Jetson TK1 heat generation rate (percentage of the Bicubic filter).

The most powerful device is the GTX 980, the one that generates more heat is the GTX 480 and the more power efficient is the Jetson TK1. These assertions should be true for all the CUDA samples provided here. The configurations and roofline models will show how hardware systems have evolved to increase their performance and to do a more efficient computation.

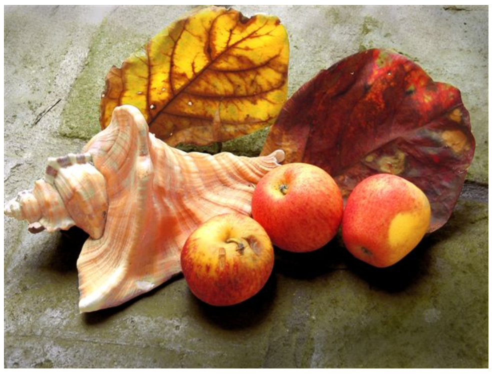
Further information and examples about this section can be found in [5].

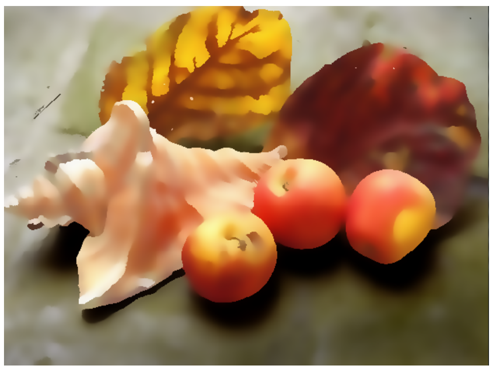
4.3 Bilateral filtering

4.3.1 Description

This is a special type of non-linear filter that smooths out textures but preserves any edges found on it. It is based on four parameters: gaussian delta, euclidean delta, the filter radius and the number of filter iterations. If the value of the second one, the euclidean delta, is high, most parts of the output texture will be filtered away as the edge-preserving nature declines. When this parameter tends to infinite the resulting filter is a gaussian one. Blur effect intensifies with a larger gaussian delta. Having a small euclidean number while incrementing the number of iterations will produce flatter colors without blurring edges, creating a cartoon effect.

We use an example of a still life paint, giving the unfiltered and filtered output using the just described technique, to demonstrate the filter result.

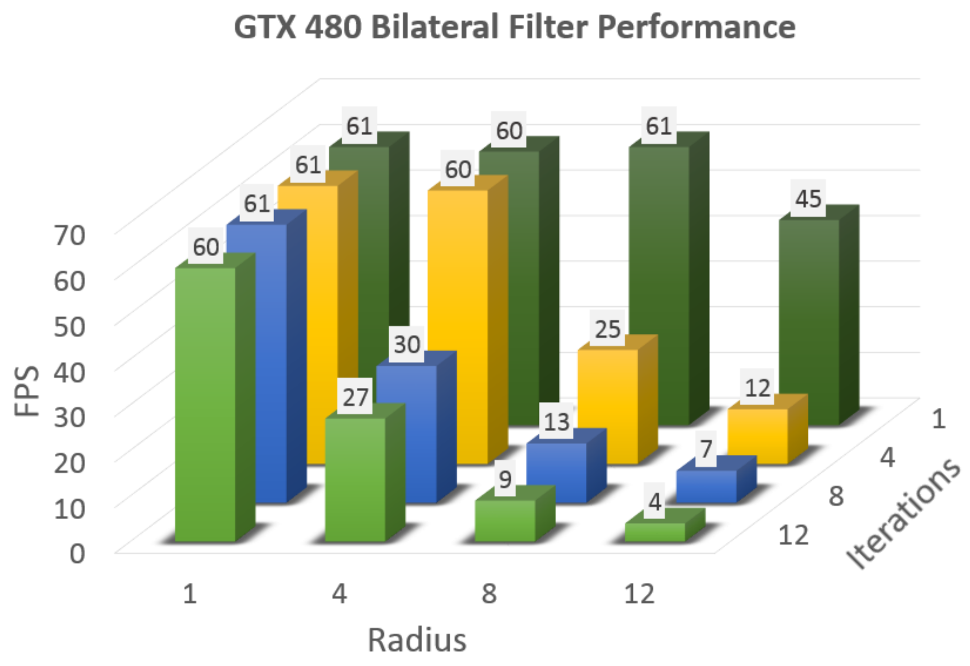

4.1.16 Original image.


4.1.17 Filtered image.

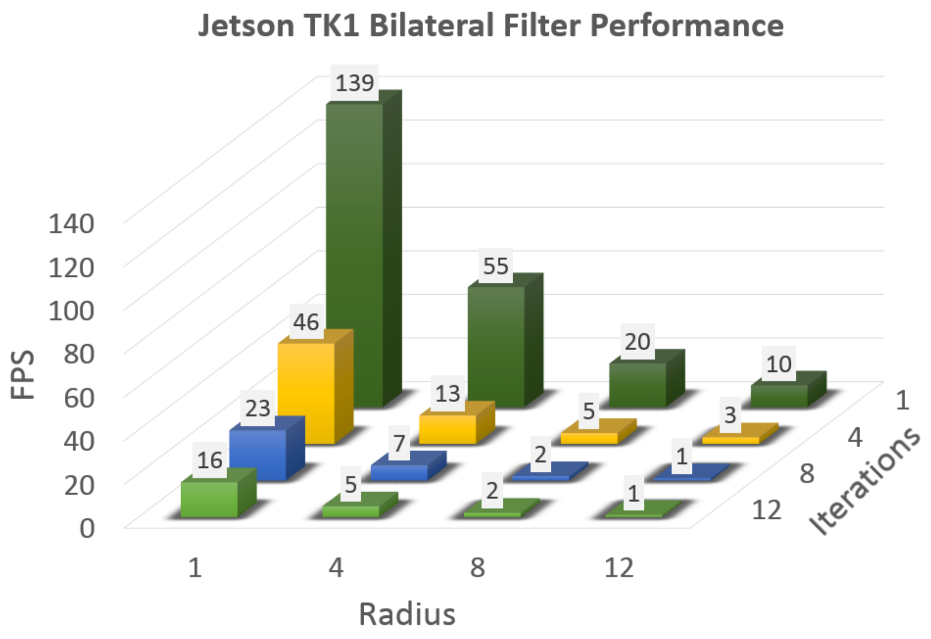
Note how, in the second image, contours are preserved while all other colors flatten. This is why it is called cartoon effect. The euclidean value for the filtered example is 0.12, a low one to make it very edge-preserving. Gaussian was set to 4 and the filter iteration count and radius size were 5.

4.3.2 Performance

To perform the tests, the euclidean value was fixed around 1 and the gaussian at 2. These parameters won't affect performance as hard as the radius and the iteration count. From 1 to 12, step size 4, values for both radius and iterations were combined into a cartesian product. The performance of the Bilateral filtering is measured using the application framerate or Frames Per Second (FPS). All results can be found in Figures 4.14.1.18, 4.14.1.19 and 4.14.1.20, respectively, for GTX 480, Jetson TK1 and GTX 980 (the iterations axis is inverted for clarity).

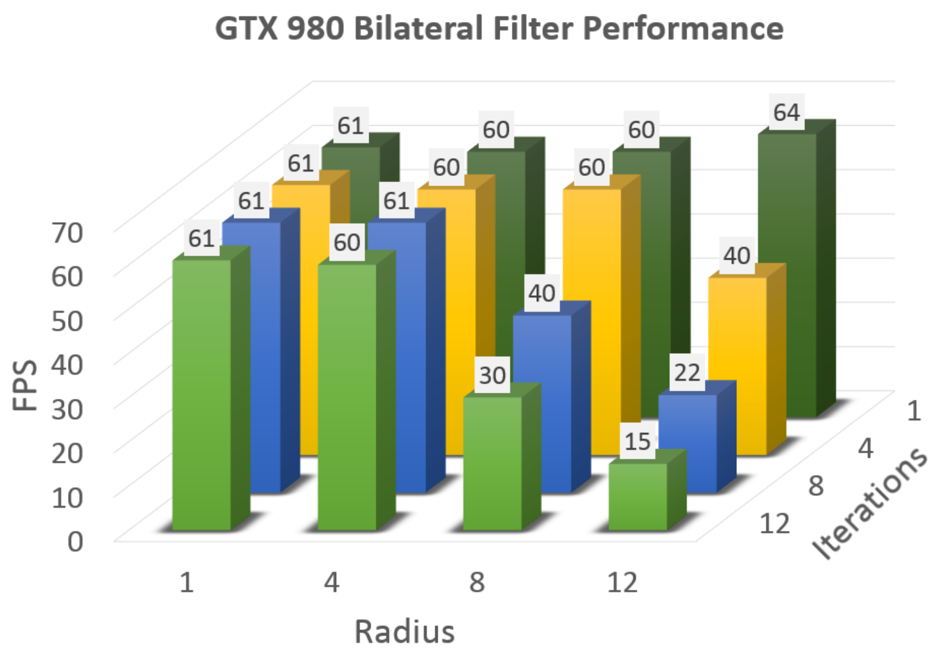


4.1.18 GTX 480 performance.



4.1.19 Jetson TK1 performance.

GTX 480 performance (see Figure 4.14.1.18) drops quickly when increasing the number of iterations. The decay simulates a logarithmic curved surface. GTX



4.1.20 GTX 980 performance.

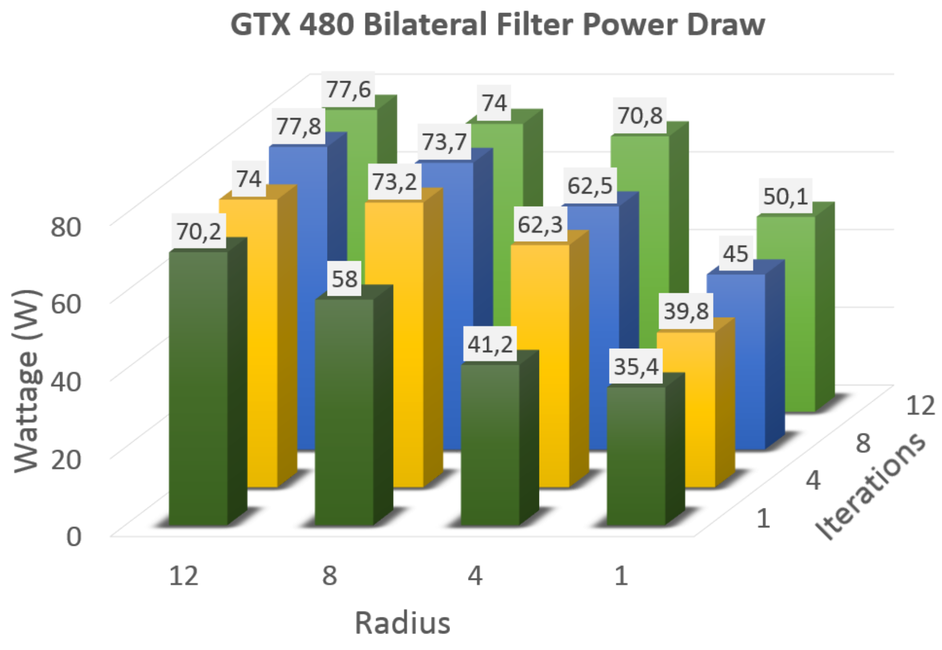
980 manages to keep up the 60 FPS (framerate limit) further than the GTX 480, with half of the configurations computed at that framerate. The lowest value for this device is the radius 12, 12 iterations configuration which is 15 FPS. In the same configuration, GTX 480 drops below the mark down to 4 FPS. This hardest case makes the GTX 980 to be 3.75x faster than the GTX 480.

On Jetson TK1, the 1-1 parameter configuration provides further performance than the limit, as there wasn't any on the application. It has the least performance and scalability of all the devices, as it drops to less than 10 FPS in most of the configurations.

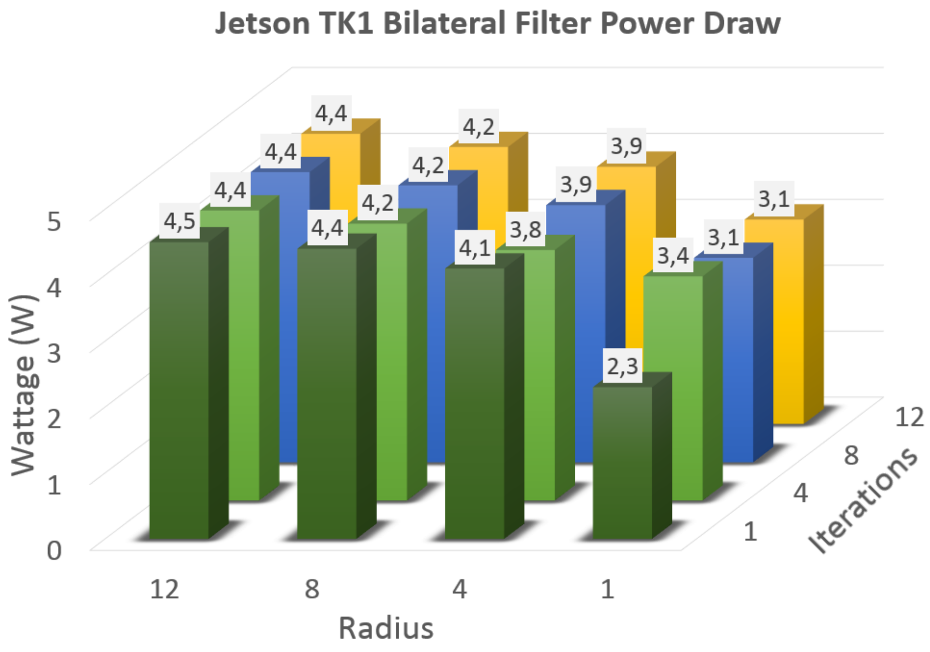
4.3.3 Power Draw and Heat Generation

From previous executions, we obtained power consumption along with generated heat. The depth axis sequence has its standard order and the radius axis is inverted, as power/heat data is better visualized this way.

In the GTX 480, the least wattage is 35.4 W and the maximum is 77.6 W. The window size for the power consumption is then 42.2 W. The minimum wattage is close to the idle value and the maximum to the limit achieved in these samples. Because of this, we say that the current sample is *complete* in terms of power consumption.

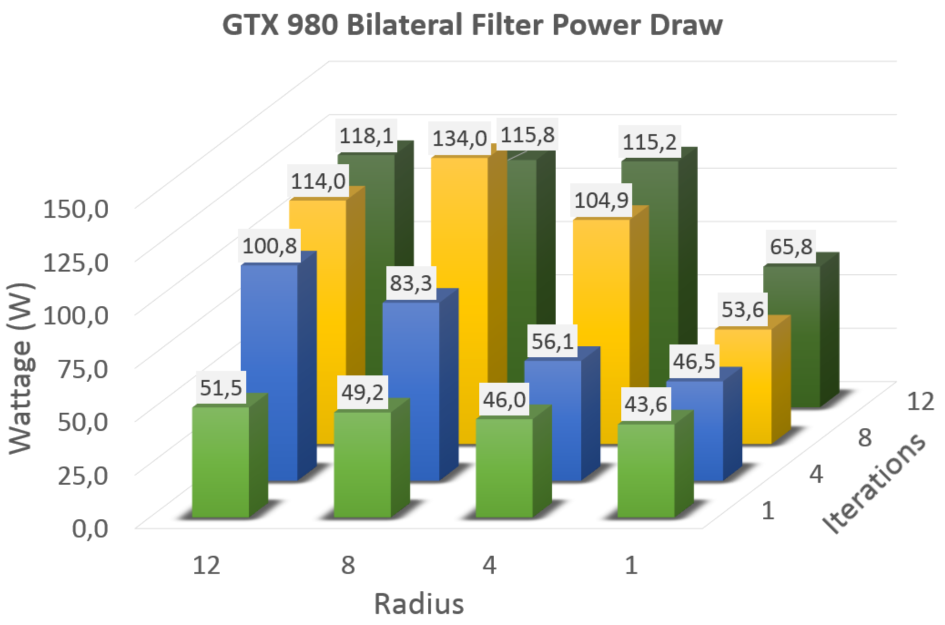


4.1.21 GTX 480 Power Draw.



4.1.22 Jetson TK1 Power Draw.

Jetson TK1 is, again, the device that consumes the least. Despite the board is connected to a monitor and a keyboard, the power draw never raises higher than 4.5 W. The minimum power consumption for this device in the sample execution is 2.3 W. The window size is then at 2.2 W.



4.1.23 GTX 980 Power Draw.

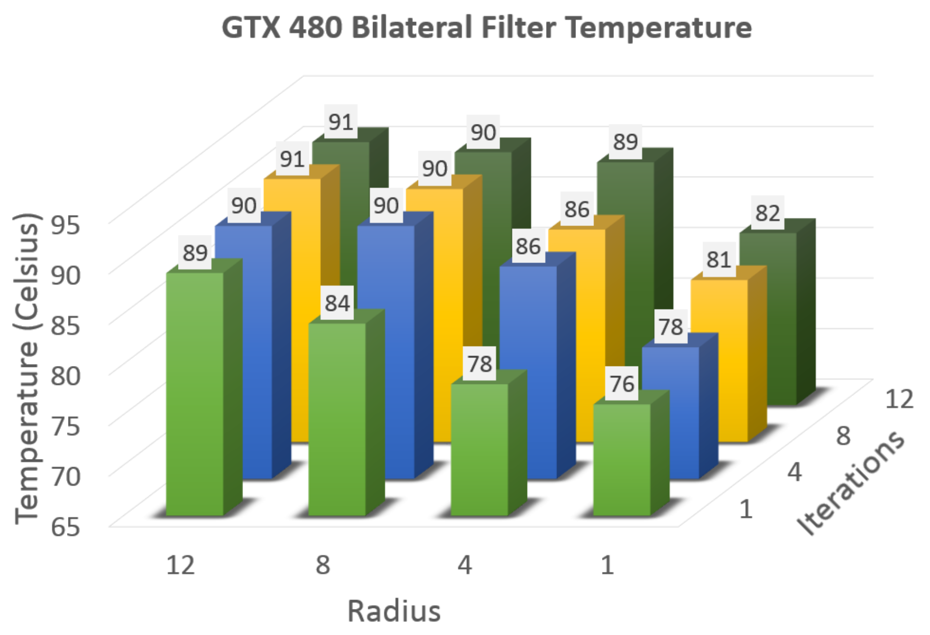
For the GTX 980, the power window size is wider: 90.4 W. The parameter configuration that has the maximum consumption is not the 12-12 but the 8-8. This configuration will have also a great heat generation relative to its position in the configurations set.

GTX 980 is having more power draw than any of the devices, not only in the hardests cases but in all the other ones, same as in the previous CUDA sample. The fact that GTX 980 consumes more is because of the hardware quantity: it has 2048 CUDA cores compared to the 480 GTX 480 has. Is it worth it to have higher power consumption? In section 4.3.4 we will discuss this question in more detail.

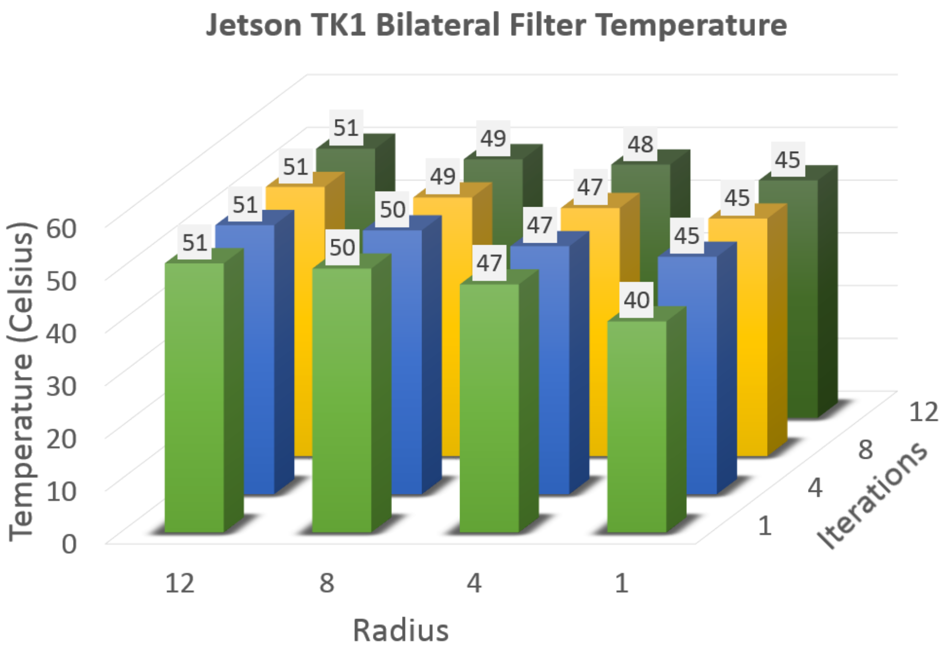
As this CUDA sample is able to get the best of the devices in all the metrics provided here, the temperature will show the greatest values of each of the platforms. Remember that all the tests were performed on summer with almost 30 degrees air temperature. The cooling system for the GeForce graphics cards is the one from a personal computer tower, with one fan in front, another in the back and the CPU one. Jetson TK1 performed all test on its own.

GTX 480 has high heat generation (see Figure 4.14.1.24). On the 12-12 configuration, the sensor marked 91 degrees. All the values that have at least one of the parameters set to 1 have considerable less heat generation than the rest, which ranges from 86 to 91 degrees while most configurations of the first have less than 84 degrees.

Jetson TK1 degrees go from 40 to 51 (see Figure 4.14.1.25). The window size



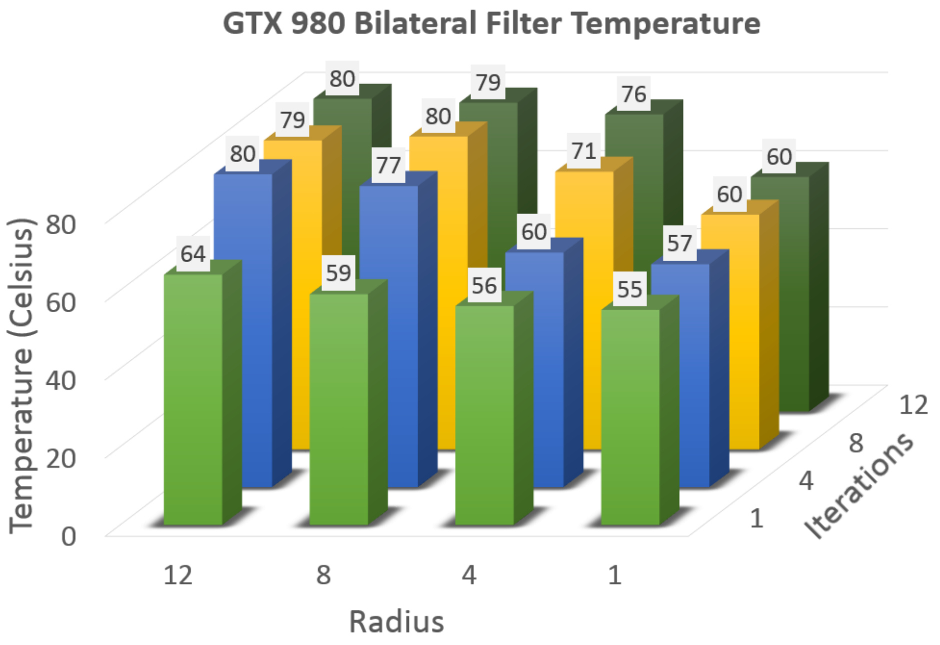
4.1.24 GTX 480 Temperature.



4.1.25 Jetson TK1 Temperature.

is smaller than the one from GTX 480, just 11 degrees (GTX 480 window had a size of 16). Neither of the GeForce devices have a temperature value less than the maximum Jetson TK1 achieves.

For GTX 980, the minimum value is 55 and the maximum 80 (see Figure



4.1.26 GTX 980 Temperature.

4.14.1.26). The rows that have at least one parameter set to 1 behave similar to those of the GTX 480, but in this case the 4-4 configuration has also a relative low value.

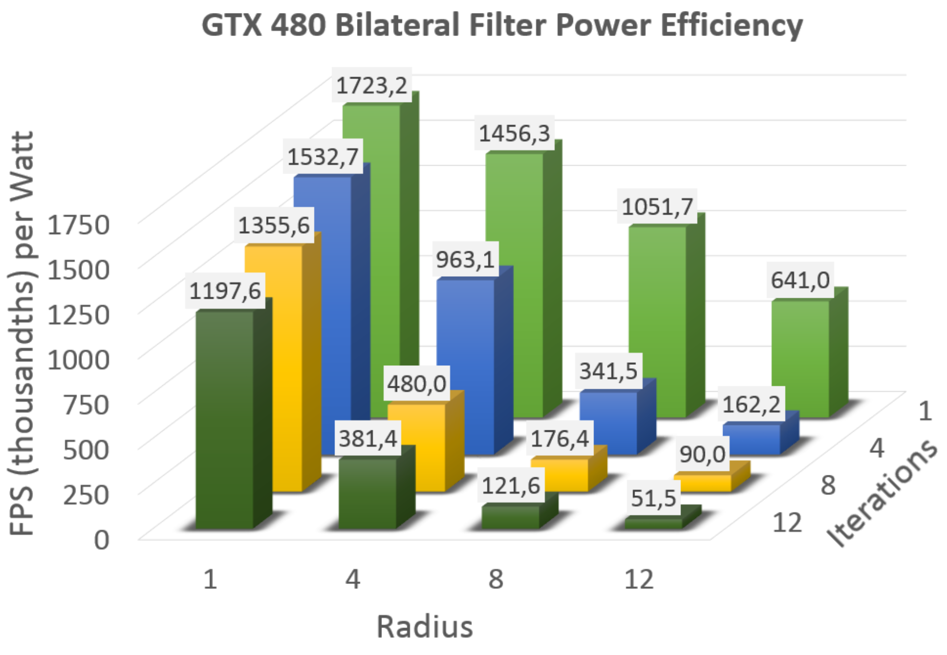
In this bilateral filter example, the more power is consumed the more heat is produced. Thus, temperature and power consumption are directly proportional. GTX 480 is on the top of the temperature scale, followed by GTX 980 and Jetson TK1, which consumes the least.

4.3.4 Algorithm Efficiency

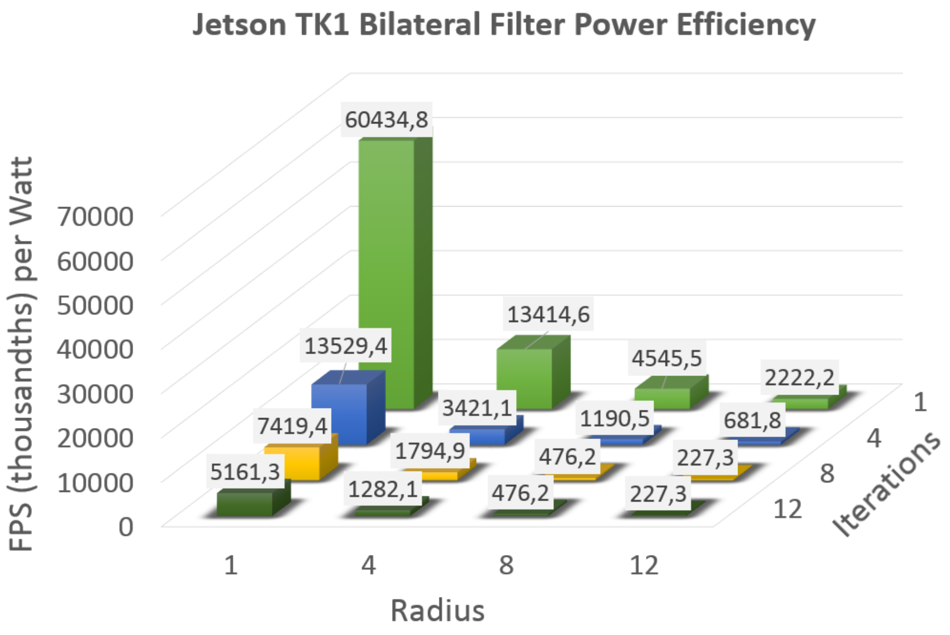
Power efficiency is now shown in thousandths of FPS per Watt to better appreciate the values. In the executions, higher efficiency is given by those configurations with low work overload. This is, as the iterations and radius size increase the device efficiency decreases.

1.7 FPS per Watt is the efficiency GTX 480 achieves for the 1-1 configuration (see Figure 4.14.1.27). The decrement in efficiency is not linear. In the 12-12, a direct linear decrement for both axis should set efficiency 144 times (12*12) lower than the 1-1. However, it is 51.5 thousandths FPS per Watt, which is 33 times less than 1-1. The graphics card shows a good scalability for efficiency.

For Jetson TK1 (see Figure 4.14.1.28), the decay is not as good as the one



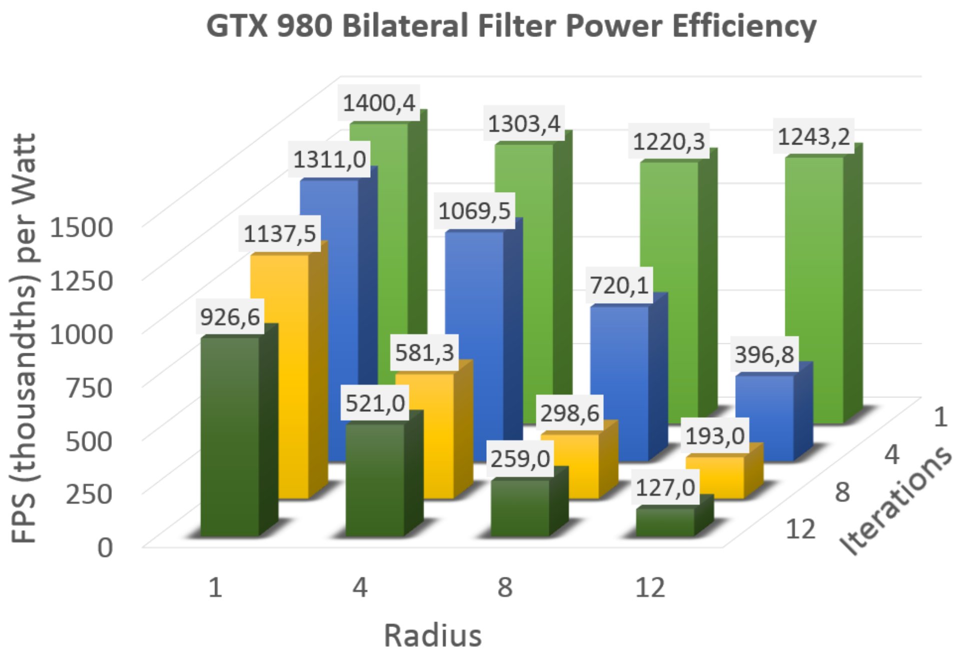
4.1.27 GTX 480 Power Efficiency.



4.1.28 Jetson TK1 Power Efficiency.

from GTX 480, having a proportion between the 12-12 and the 1-1 of 265 times less. Also, in the 1-1 configuration, the platform gets 60 FPS per Watt, a record in the efficiency scale.

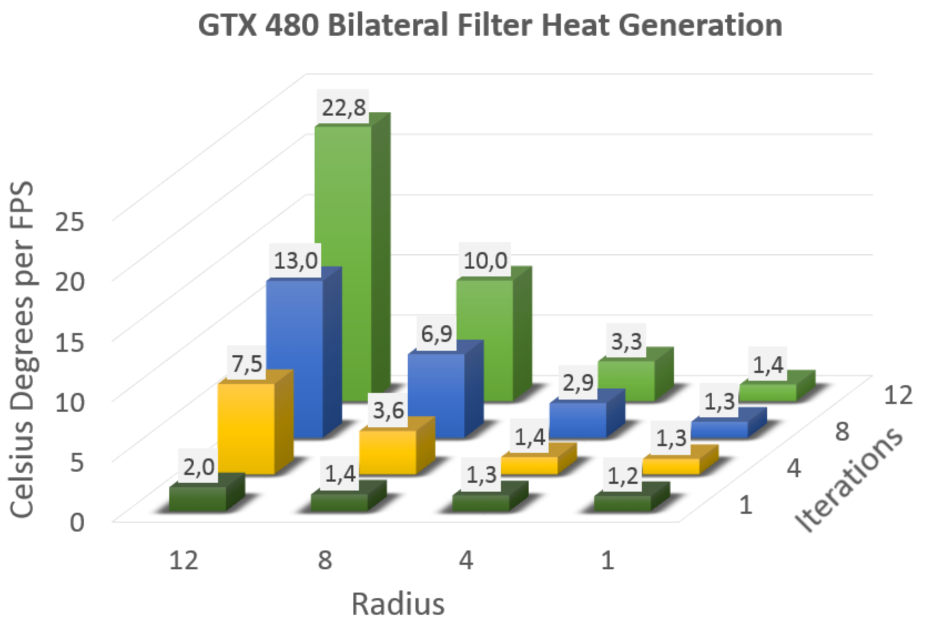
GTX 980 (see Figure 4.14.1.29) starts with 1.4 FPS per Watt and continues in the diagonal configurations (1-1, 4-4, 8-8 and 12-12) with 1069.5, 298.6 and 127.0 thousandths FPS per Watt. The differences or steps are approximately 330, 770 and 170. In GTX 480, they are 760, 680 and 120. As a matter of fact, GTX 480 begins in 1-1 with 1.7 FPS per Watt, which is higher efficiency than the GTX 980 one and ends with a lower value in 12-12. Furthermore, in the radius 1 row GTX 480 is more efficient than GTX 980, but not in the 1 iteration row, as GTX 980 mantains there almost all the initial efficiency. The last value (radius 12) of that iteration row is 1.24 FPS per Watt (GTX 980 lost only 0.16 FPS per Watt lower when the radius value increments).



4.1.29 GTX 980 Power Efficiency.

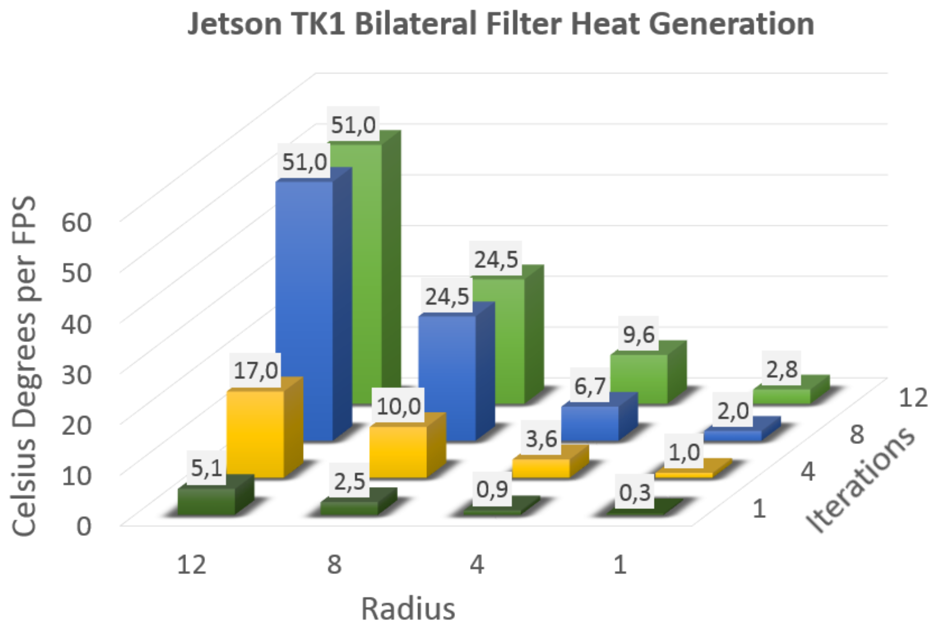
The fact that GTX 480 is more efficient on the first configurations and loses on the last ones says that having 2048 CUDA cores can be good if you have a lot of work to do. However, when the machine has lighter work, you could be wasting away watts of power because you have a lot of resources to mantain warm, even though they are not all in use.

Respect to heat generation efficiency, the more iterations and radius size, the worse this efficiency will be. Both efficiencies tend to decline when more work is



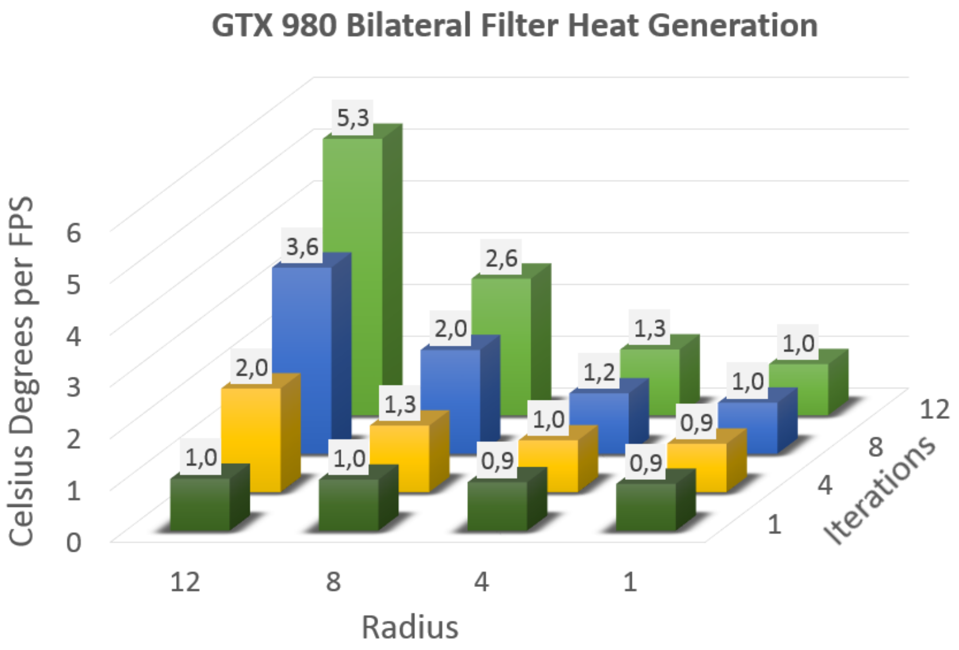
4.1.30 GTX 480 Heat Generation Efficiency.

performed, in a way that is independent from the underlying hardware. For clarity, both axis are inverted respect to the previous charts.



4.1.31 Jetson TK1 Heat Generation Efficiency.

GTX 480 produces 1.2 degrees per FPS in the 1-1 configuration and 22.8 degrees per FPS in 12-12 (see Figure 4.14.1.30). This graph is not similar to the power efficiency one, as here, the rows corresponding to configurations that at



4.1.32 GTX 980 Heat Generation Efficiency.

least have one parameter set to 1 keep the efficiency high with low heat generation rates.

Jetson TK1 (see Figure 4.14.1.31) increases its heat generation proportionally faster than GTX 480. It starts with 0.3 degrees per FPS and ends with 51 degrees per FPS. Despite it starts with better efficiency, it ends much worse.

Finally, GTX 980 (see Figure 4.14.1.32) has the best average efficiency among all the devices. Again, the most efficient device for the first configurations is the Jetson TK1, but the way GTX 980 manages to keep degrees per FPS as low as possible, keeping its efficiency until 5.3 degrees per FPS in the 12-12 configuration, makes it the device with the highest heat generation efficiency.

4.4 Box filter

4.4.1 Description

Box filter is a linear filter that performs the smoothing of an image by converting each pixel in a weighted average of its neighbourhood. It is one of the simplest filters that can be used for blur effects. By definition, this filtering technique will smooth out the parts of an image whether or not they have any contours or edges. It can be easily parallelizable because each pixel computation is independent from the rest.

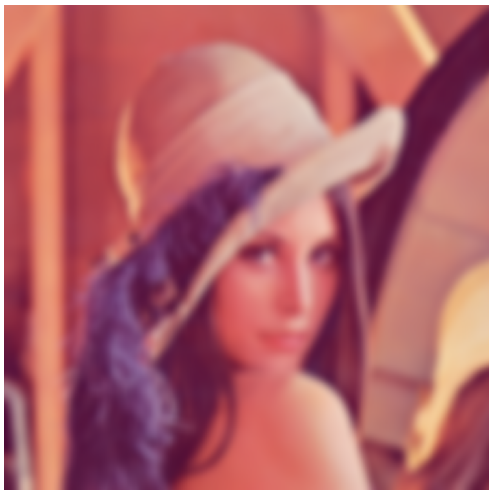
Among all the possible implementations, the matrix one is the most common. It is used to ponderate each of the pixels in the neighbourhood for the mixture (filter box). This implementation provides a wide range of possible filters. An important difference between this and some of the previous examples is that this one is linear.

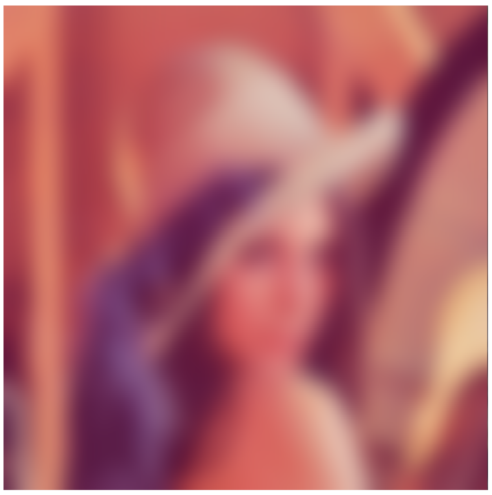
An advantage here is that it can be used to approximate to a Gaussian filter, as stated by the Central Limit Theorem. Applying it a certain number of times results in the approximation of a quadratic convolution. Furthermore, we can perform more interesting blurrings by setting negative values in the matrix, but we will not go this further.

In this example, two parameters that will sound familiar modify the result of the filter: the radius size and number of passes. This time, because the radius size is not as relevant as the iteration count for the performance, we chose to fix it at 4 and only change the second one.

Figures 4.14.1.33, 4.14.1.34 and 4.14.1.35 show the original image and two filter configurations result, one of them with radius of 10 and passes count 2 and the other with twice these parameters. The effect of the filter should be clear at first sight.

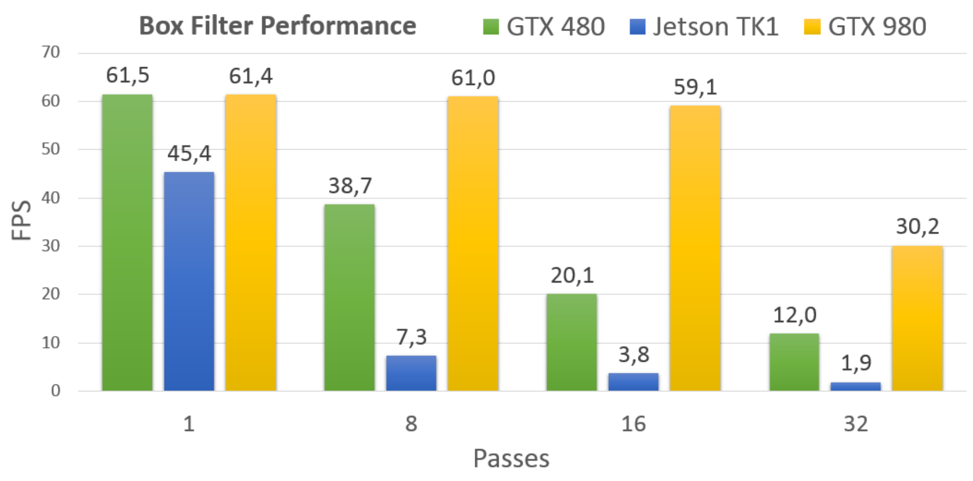
4.1.33 Original Lenna.


4.1.34 Radius: 10 Passes: 2.


4.1.35 Radius: 20 Passes: 4.

4.4.2 Performance

To test the performance of the algorithm four configuration were used (all with radius size 4) in which the number of passes had the values of 1, 8, 16 and 32. Figure 4.14.1.36 show the graph with the obtained results.



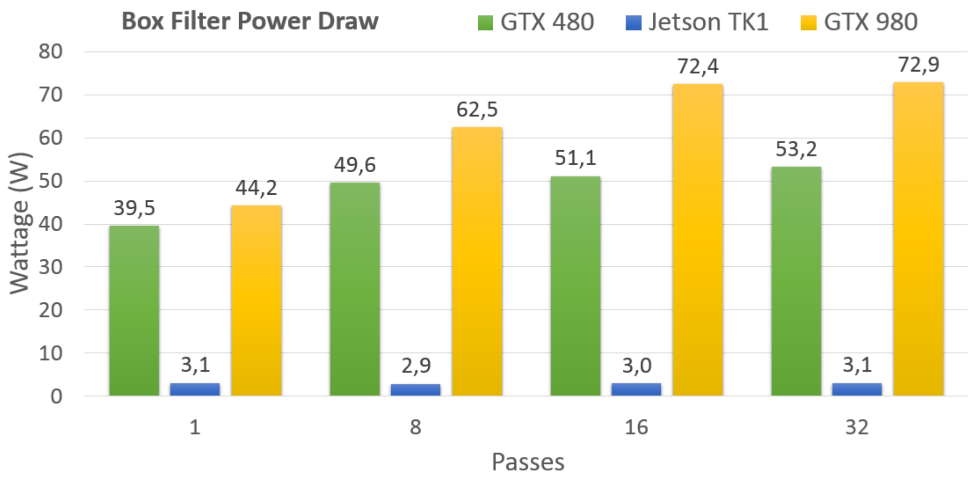
4.1.36 Box Filter Performance.

Maxwell architecture has again the best performance related to its predecesors. The lowest performance is the one of the Jetson TK1. When there is only one filter pass the framerate values of the devices are more balanced. Additionaly, the framerate of the GTX 980 is only dropped to 30.2 FPS in the last configuration, which is an acceptable value for a visual application (it is 2.51 times the performance of the GTX 480). The executions performance in the rest of platforms fall down the 30.2 FPS at 8 and 16 iterations. Jetson TK1 will never achieve the 60 FPS limit in this application.

4.4.3 Power Draw and Heat Generation

Both power consumption and heat generated in this example are lower than in the previous cases. Jetson TK1 has the same power consumption in the least passes count than in the highest, meaning that the application status doesn't affect too much to its power draw. GeForce graphics cards increase their consumption with higher number of passes as usual (see Figure 4.14.1.37).

In the GTX 480, the power ranges from 30.5 W to 53.2 W. For the GTX 980, it goes from 44.2 W to 72.9 W. We observe how GTX 480 energy consumption is far more stable than the one of the GTX 980. This second device draws more energy

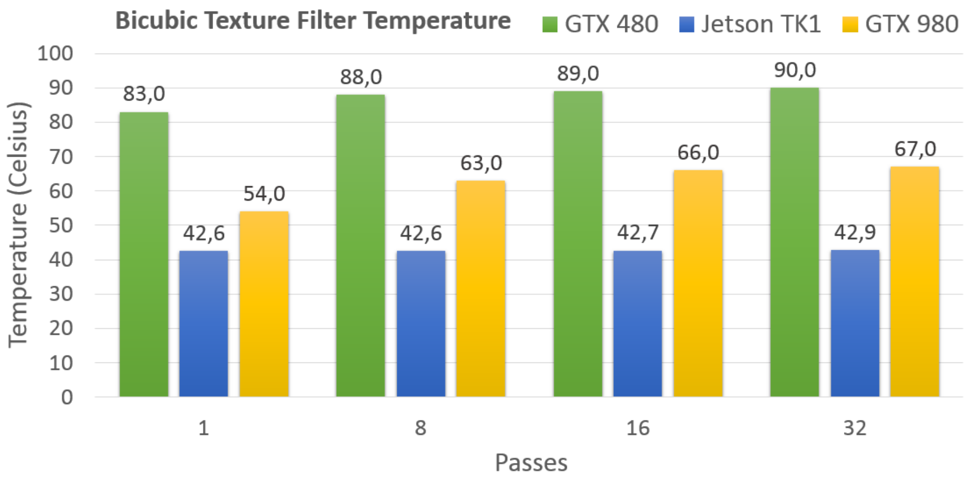


4.1.37 Box Filter Power Draw.

than the other platforms in all configurations.

GTX 980 has quite small power increment from configuration 16 to 32. Same event occurs on the GTX 480 starting from configuration 8. The point where consumption doesn't increase as fast as in the first configurations denotes how the device fullfils its main power requirements and doesn't need to warm up for the rest of the work to be done: the biggest step is the one that makes the machine change from idle to fully working.

The temperature values shown in Figure 4.14.1.38 behave similar to those of power drawback from Figure 4.14.1.37. Here, Jetson TK1 does show an increment of temperature when it has to perform more filter passes. The increment is, however, not very significant (0.3 degrees starting from 42.6).

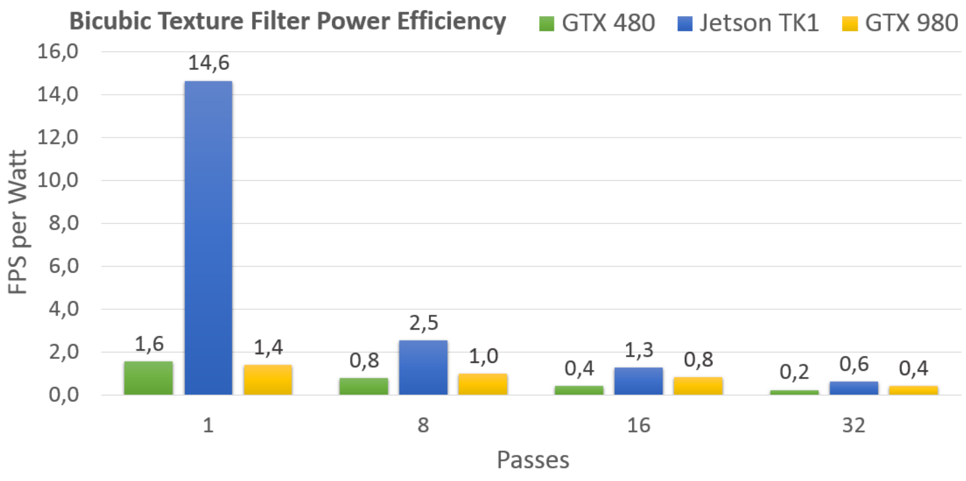


4.1.38 Box Filter Temperature.

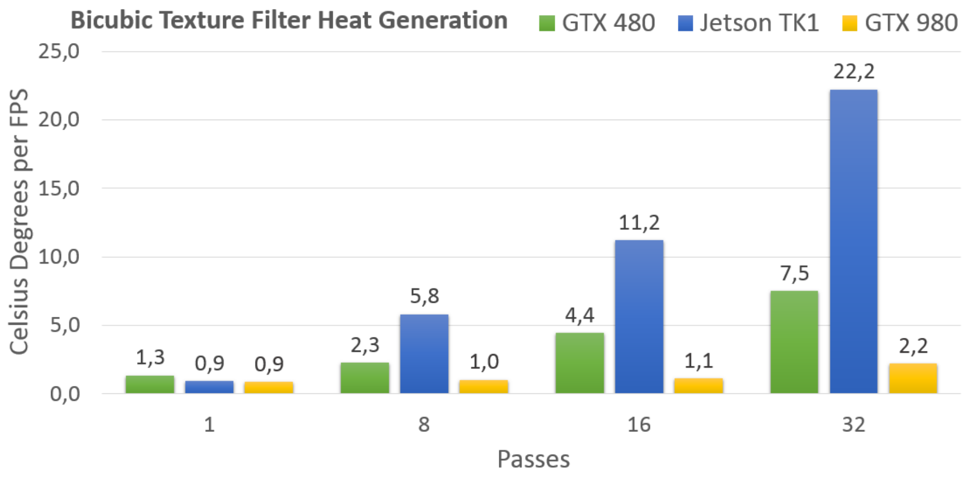
GTX 480 varies from 83 to 90 degrees. As in the power consumption chart, the greater increment in heat generation is from the 1 pass configuration to the 8 one. The rest are all 1 degree increments. GTX 980 behaves the same way, incrementing from 54 to 63 and then from 63 to 66 in the first configurations, but just to 67 in the last one.

4.4.4 Algorithm Efficiency

In Figure 4.14.1.39, similar results to those of the bilateral filter are shown: the best power efficiency is provided by the lighter configurations and the worst by the hardest ones.



4.1.39 Box Filter Power Efficiency.



4.1.40 Box Filter Heat Generation Efficiency.

GTX 480 has again higher efficiency than the GTX 980 on the first configuration and lower in the rest. The highest efficiency is provided by the Jetson TK1 in the 1 pass configuration, with 14.6 FPS per Watt, a much higher value than the GeForce graphics cards.

About the efficiency on heat generation (see Figure 4.14.1.40), Jetson TK1 has not the lowest in every of the configurations: on the first one, GTX 480 has the highest heat generation. In the rest of configurations, the order from better to worse efficiency is GTX 980, GTX 480 and Jetson TK1.

GTX 480 and GTX 980 start in the first configuration with similar efficiency values (1.3 and 0.9, respectively). The difference between them is 0.4, but as the number of passes increases, the different also does. At the end, this value is 5.3 degrees per FPS. Jetson TK1 decreases its heat generation efficiency much faster than the GeForce devices, reaching 22.2 degrees per FPS in the last configuration.

4.5 Image denoising

4.5.1 Description

Now we attend at a common problem in image processing known as *image denoising*. An image that travels through a network can have errors as some pixels may be corrupted during the data transmission (data noise). Hard disks can also make read/write mistakes, but this is less frequent. Situations were these errors tend to occur are those in which the message is not travelling through a wire (e.g. air) and the signal is too weak for the receiver. Denoising algorithms become critical in space programs or military scenarios where perfect information about orders or numbers is required. In the context of our example we will concentrate on the denoising of a picture.

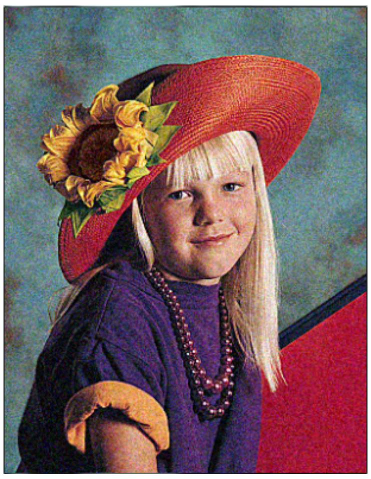
These errors manifest in the picture as abnormal color dots because of the radical pixel data change. By looking at it, it is not hard to realise that some of the pixels are incorrect, as they may be very different from their neighbourhood (e.g. red dots in the purple jersey of picture 4.14.1.41).

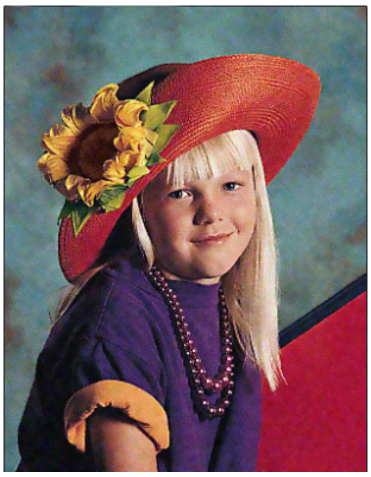
In this case, we could use a box filter to approximate to the original image, as corrupted pixels can be more or less recovered by the information of the healthy surrounding ones. Despite of this, we will use two algorithms called K Nearest Neighbors (KNN) (see Figure 4.14.1.42) and Non Local Means (NLM) (see Figure 4.14.1.43) that fit better in this problem. Again, we won't go into the details of each of them but it has to be said that the second one is a more complex variation that has greater resource usage. An optimized version of NLM called Quick NLM or NLM2 is also available (see Figure 4.14.1.44).

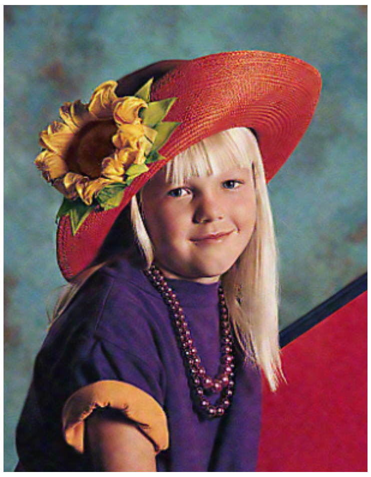
4.5.2 Performance

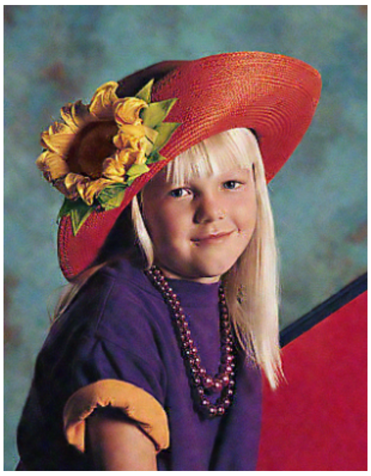
This example will clearly show the difference between the hardware platforms. The denoising algorithms were executed on all the devices and the obtained data is presented on Figure 4.14.1.45.

GTX 480 and GTX 980 have much greater performance than Jetson TK1. On the first denoising algorithm, Jetson TK1 performance is 9% of the GTX 480 one and 5% of the GTX 980. With NLM algorithm, the proportion against the GTX 480 shortens, being Jetson TK1 framerate 15% of the one of that device, but remains around the same in the rate with the GTX 980.


4.1.41 Original Noisy Image.


4.1.42 Applying KNN.

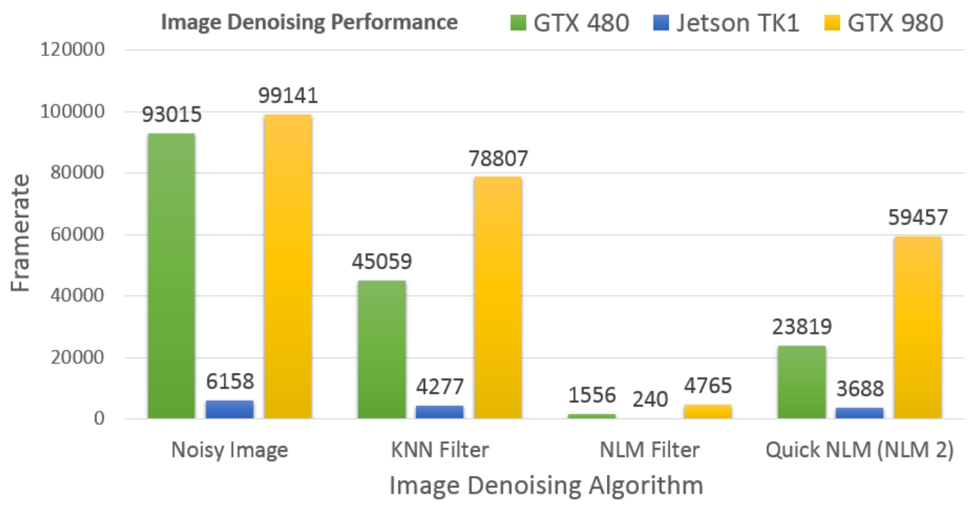

4.1.43 Applying NLM.


4.1.44 Applying Quick NLM (NLM2).

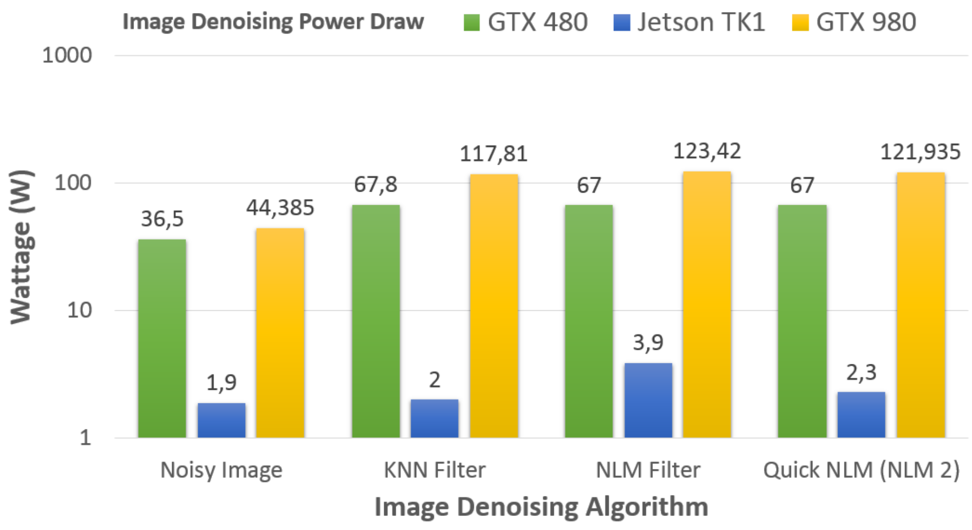
4.5.3 Power Draw and Heat Generation

The power draw consumed in this example is not the highest one nor the lowest. In Figure 4.14.1.46, the vertical axis presents a logarithmic scale. The maximum power consumption for GTX 480 is 67.8 W in the KNN filter. NLM and Quick NLM have almost the same power draw. GTX 980 goes higher to more or less 120 W and Jetson TK1 doesn't go further than 3.9 W.

For temperature (see Figure 4.14.1.47), GTX 480 almost reaches the maximum temperature of all the samples with 90 degrees in the NLM filter. Jetson TK1 stops at 48 degrees and GTX 980 at 79 (NLM). Quick NLM has 1 degree less than the NLM in the GeForce devices. The optimized algorithm has much higher fram-



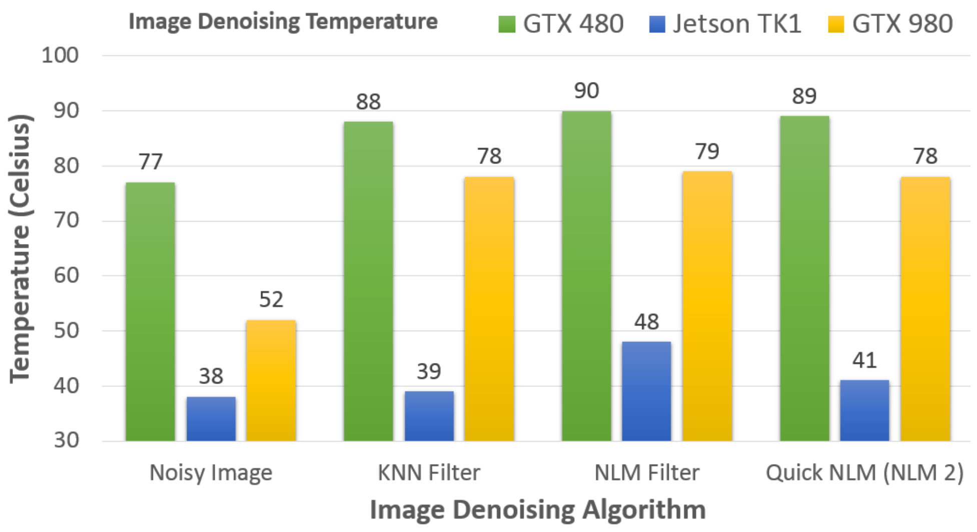
4.1.45 Image Denoising Performance.



4.1.46 Image Denoising Power Draw.

erate and its power drawback and heat generation is a bit lower.

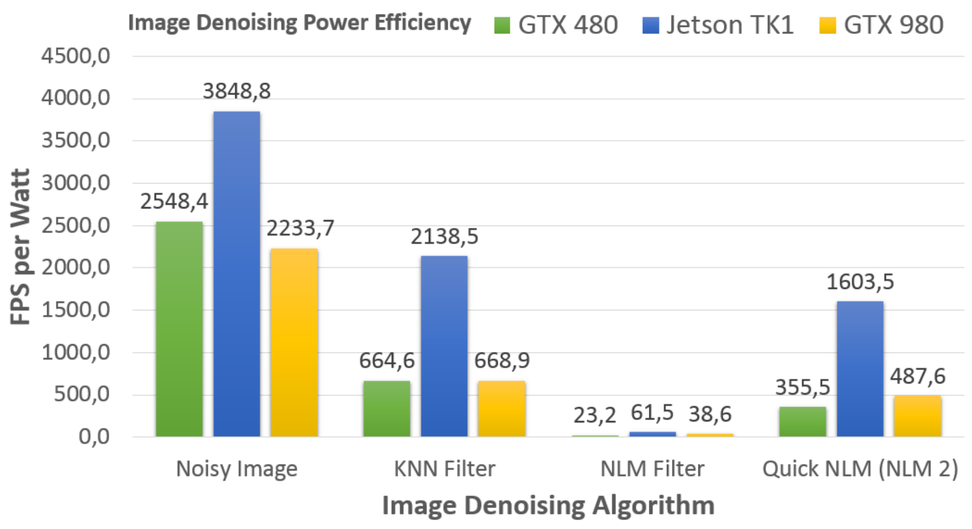
As expected, the least temperature values are generated when no algorithm is applied (noisy image) and the higher ones when using NLM denoising algorithm. GTX 980 is the platform that raises higher, changing from 52 degrees to 78 in noisy to KNN swap. The temperature step is 26 degrees. GTX 480 has higher temperature, but as it starts from 77 degrees, the step is smaller.



4.1.47 Image Denoising Heat Generation.

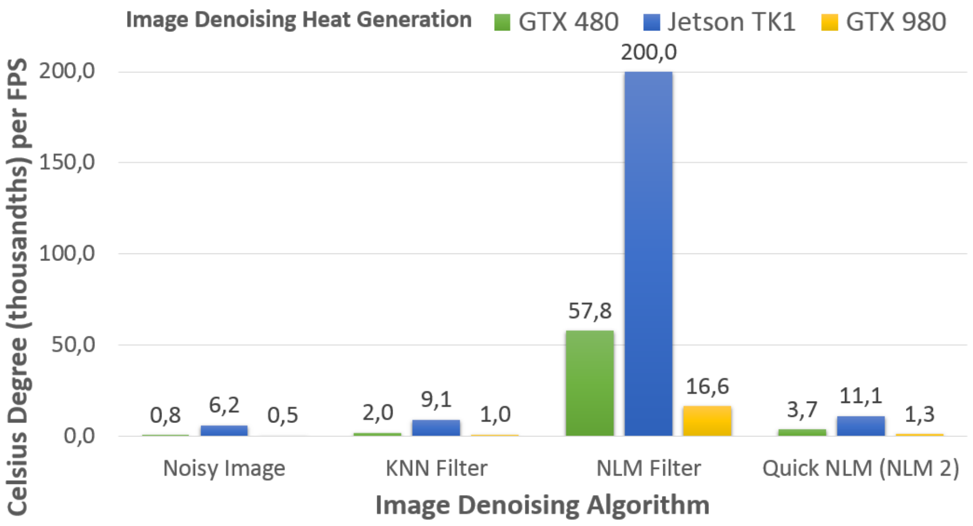
4.5.4 Algorithm Efficiency

The efficiency behaves as in the previous case: GTX 480 beats GTX 980 in the first configurations and on harder tasks the GTX 980 is more efficient (see Figure 4.14.1.48). Jetson TK1 is the most efficient platform with more than twice the efficiency in KNN and Quick NLM than the other devices. In NLM, it is more than twice just with respect to GTX 480.



4.1.48 Image Denoising Power Efficiency.

Speaking of heat generation efficiency (see Figure 4.14.1.49), GTX 480 has less than GTX 980 but is better than Jetson TK1 in all cases. On NLM, Jetson TK1 generates 0.2 degrees per each FPS it achieves, being the lowest heat generation efficiency in the graph. GTX 980 has again the best scalability in this metric with 0.016 degree per FPS on NLM filter.



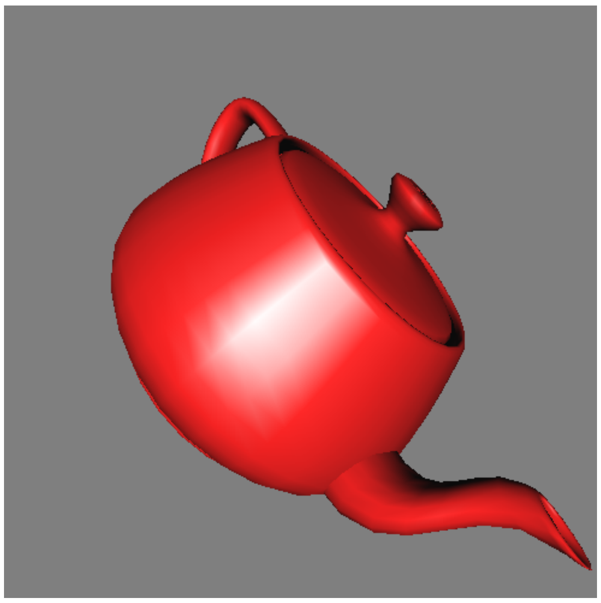
4.1.49 Image Denoising Heat Generation Efficiency.

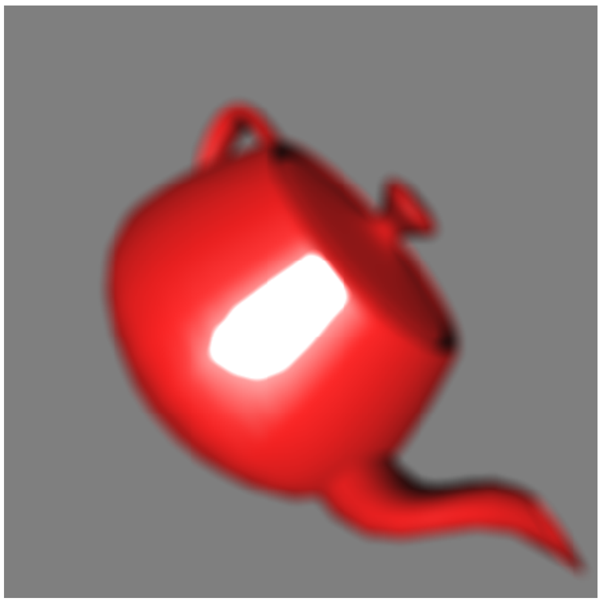
4.6 Post-process GL

4.6.1 Description

Post-processing refers to the processing that is done after the rendering phase. An example of a post-processing application is motion blur in videogames: blurring the screen when the camera is suddenly rotated.

In this example, we will post-process a 3D model of a teapot using OpenGL. As always, snapshots in which the post-processing is turned off (see Figure 4.14.1.50) and on (see Figure 4.14.1.51) are provided to appreciate the effect of the algorithm. In the processed image, the smoothing algorithm has a radius of 8. The teapot is constantly rotating in all axis when the application is executing.

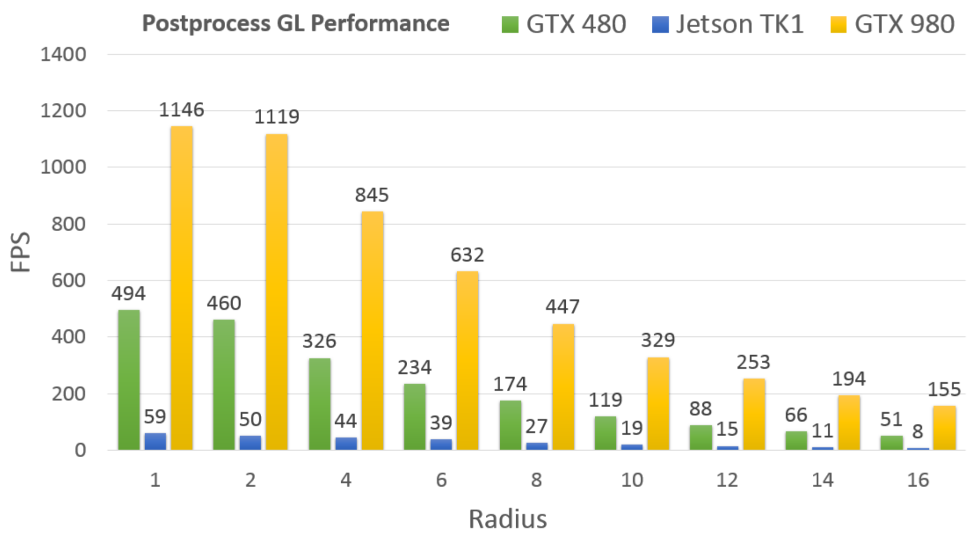

4.1.50 Original Teapot


4.1.51 Post-processed Teapot

4.6.2 Performance

Results measured started with radius 1 and then 2 until 16 incrementing 2 units in each step (see Figure 4.14.1.52). This time, no framerate limit was active on the application. With higher values of the radius, the framerate declined for all three devices. The next charts show the conclusions of this test.

GTX 980 has the highest performance value at 1 radius size with 1146 FPS. The lowest value is given by Jetson TK1 at the maximum radius size with 8 FPS.



4.1.52 Postprocess GL Performance.

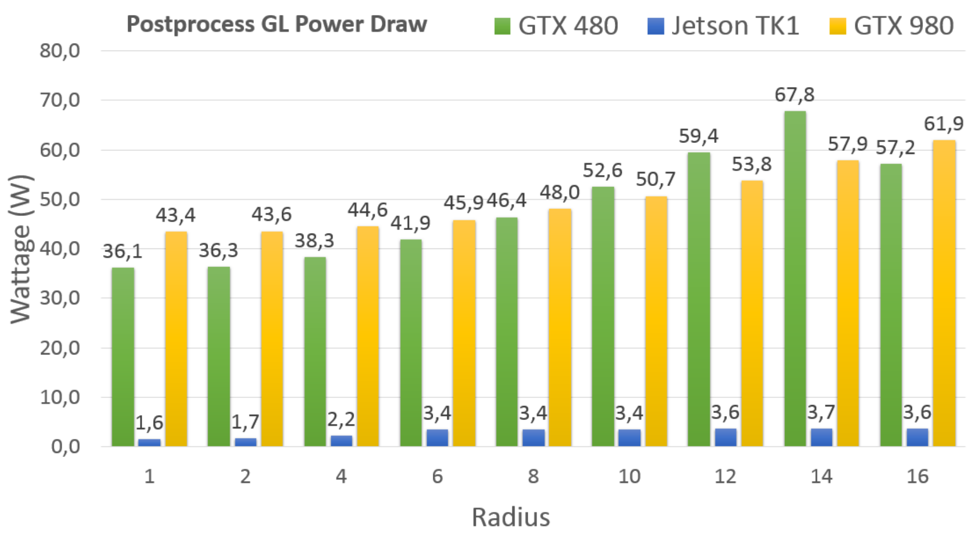
Again, an acceptable framerate is that equal or above 30 FPS. GTX 980 never drops below 155 FPS and GTX 480 keeps its performance above 50. Jetson TK1 is the only device that loses performance to below that quality mark, being the 8 radius size value, 27 FPS, the first one to drop below 30.

On the hardest case, GTX 480 has 6 times the performance of the Jetson TK1 and GTX 980 3 times the one from GTX 480. The performance drops in this case for GTX 980 and Jetson TK1 down to 13% of the initial value. GTX 480 drops to 10%.

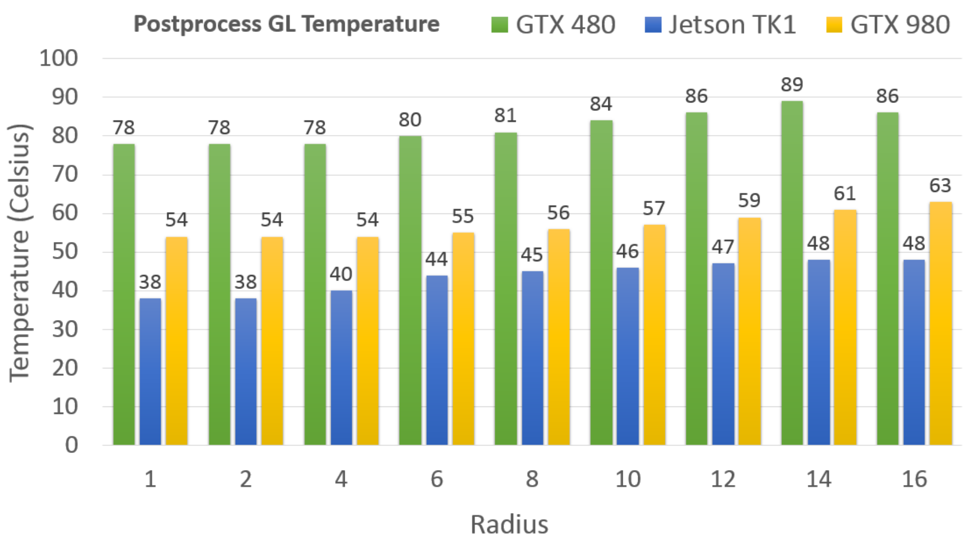
4.6.3 Power Draw and Heat Generation

Power consumption values range greatly in this example (see Figure 4.14.1.53). On GTX 480, the minimum value is 36.1 W. At the end of the test, the value was 67.8 W (not achieved on radius 16 but on 14). This won't happen to GTX 980 which has its power draw increased in each iteration. Jetson TK1 starts with 1.6 W and ends with 3.6 W. The 14 radius value is 3.7W, giving it as in GTX 480 a higher one than in the last configuration.

Temperature values in the post-processing example range from 78 degrees to 89 in the GTX 480 (see Figure 4.14.1.54). GTX 980 has a smaller variation, going from 54 degrees to 63. Finally, Jetson TK1 goes from 38 degrees to 48, the least heat generation in the graph. The direct relationship between the power consumption and the temperature makes the 16 radius size configuration for the GTX 480 have less temperature than the 14 one.



4.1.53 Postprocess GL Power Draw

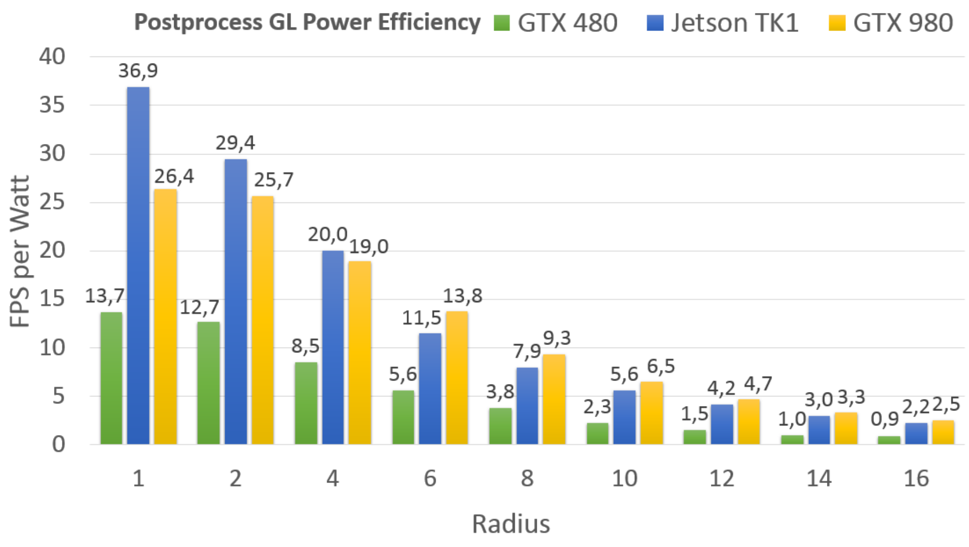


4.1.54 Postprocess GL Heat Generation

4.6.4 Algorithm Efficiency

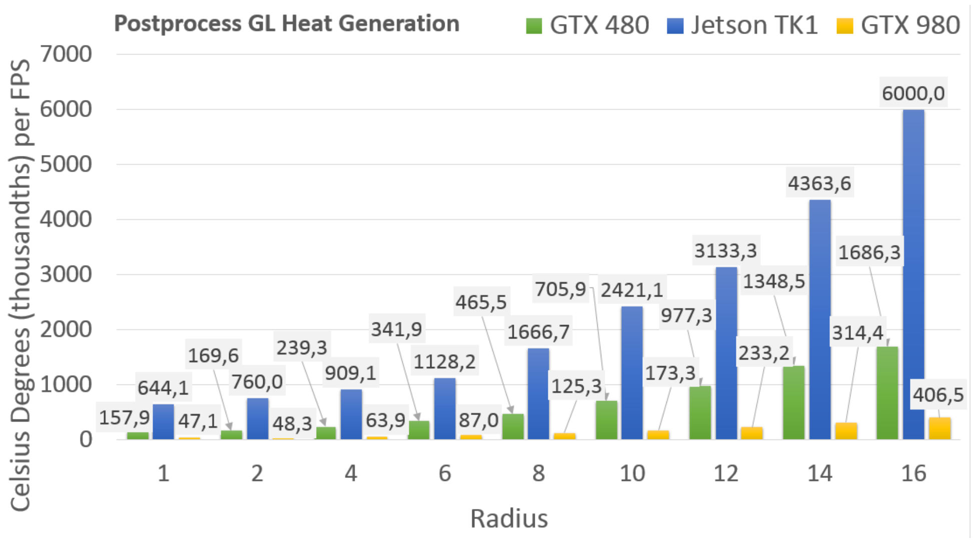
On this example, the GTX 980 overpasses in efficiency Jetson TK1 (see Figure 4.14.1.55). In the chart values, the efficiency of the Jetson decreases faster and has less values than the GTX 980 for the first time in the 6 radius size configuration. The difference in efficiency between these two devices won't increase much further, as with it shortens up to 0.3 FPS per Watt in the last configuration. GTX 480 is far away below in efficiency respecting to the other devices.

The last of the charts, the heat generation efficiency for the post-processing



4.1.55 Postprocess GL Power Efficiency.

example, sets GTX 980 as the device with the highest efficiency in terms of temperature (see Figure 4.14.1.56). It has the lowest value in the temperature scale and a 24% of the GTX 480 heat generation for 16 radius size. Jetson TK1 heat generation for that configuration is 14 times higher than the GTX 980 value.



4.1.56 Postprocess GL Heat Generation Efficiency.

Conclusions

4.6.5 English

Performed tests have proved the existing differences between the hardware platforms. The distance in terms of performance, consumption, heat generation and efficiency between the devices is directly related to the distance in time, their architecture and the hardware amount. Thanks to the experiments, it has been also possible to appreciate how the nature of applications defines how well they fit into one device or another.

To sumarize the test results, GTX 980 has the highest performance in almost all configurations, being the GTX 480 better for some low work applications in which it fits better than the Maxwell device. The temperature generation is usually directly proportional to the power draw of the current device, and the harder the work is, the more temperature and power consumption the device has.

In some applications, the efficiency of the platforms may differ. Generally, Jetson TK1 has the highest efficiency, though in the last example was overpassed by GTX 980 from an early configuration up to the last one. Both heat generation and power consumption efficiency decrease with more expensive tasks. Jetson TK1 may have the highest power draw efficiency because of the integrated CPU-GPU design, but it has the lower heat generation efficiency. The numbers speak by themselves, showing the strong points each device has (see Table 4.2).

Depending on the application, one could think of using a device because of its computational horsepower, its power or temperature efficiency or just because it generates less heat, even though its heat generation efficiency is not very good. For example, if there is no need for high computational power, then the best is to use GTX 480 as it has less power draw than GTX 980. However, for greater executions the GTX 980 is more efficient and it will be cheaper to use it despite of its higher wattage per second as it will finish ahead of the Fermi device.

Another scenario could be the one where heat generation is important: maybe

	Least	Greatest
Computational Power	Jetson TK1	GTX 980
Power Drawback	Jetson TK1	GTX 980
Heat Generation	Jetson TK1	GTX 480
Power Drawback Efficiency	GTX 480	Jetson tK1
Heat Generation Efficiency	Jetson TK1	GTX 980

Tabla 4.1: *Final Results.*

there is a high number of devices and the accumulated heat of all together could reduce the lifetime of some system components. Then, a device with higher heat generation efficiency is desired. Whether or not we need performance or efficiency, the different possible platforms will have different features that will make the choice of chosing one or another dependent on the task to be performed, its computational and energetic needs, and the conditions of the target system, both in terms of heat generated and economic cost produce by the time of usage.

4.6.6 Spanish

Las pruebas realizadas han demostrado las diferencias existentes entre las distintas plataformas hardware. La diferencia en rendimiento, consumo, generación de calor y eficiencia entre los dispositivos está directamente relacionada con la distancia en el tiempo, la arquitectura y la cantidad de hardware. Gracias a los experimentos, ha sido posible apreciar también cómo la naturaleza de las aplicaciones hace que encajen mejor en un dispositivo o en otro.

En resumen, la GTX 980 ha resultado ser el dispositivo que más rendimiendo proporciona en casi todas las configuraciones de todos los ejemplos. No obstante, la GTX 480 ha producido mejores resultados en algunos casos que requerían menor trabajo debido a la mejor adapción del programa al hardware. La generación de calor suele ser directamente proporcional al consumo del dispositivo y cuanto más trabajo se le asigna, más temperatura y consumo se produce.

En algunas aplicaciones, la eficiencia de las plataformas podía ser diferente. Generalmente, la Jetson tK1 tiene la eficiencia más alta, aunque en el último ejemplo fue superada por la GTX 980 desde una configuración más primeriza hasta la última. Tanto la eficiencia en generación de calor como en consumo decrementan con tareas más intensas. Aunque la Jetson TK1 tenga la mayor eficiencia en cuanto a consumo gracias a su diseño integrado GPU-CPU, hemos que visto que es la que peor eficiencia tiene en cuanto a eficiencia en la generación de calor. Los números hablan por si mismos, mostrando los puntos fuertes de cada dispositivo (ver Tabla 4.2).

Dependiendo de la aplicación, uno puede pensar en usar un dispositivo por su poder computacional, su eficiencia en consumo o generación de calor o simplemente porque genera menos calor, aunque su eficiencia en generación de calor no sea muy buena. Por ejemplo, si no hay necesidad de un alto rendimiento, es mejor usar la GTX 480 dado que consume menos que la GTX 980. Sin embargo, para ejecuciones de mayor importancia la GTX 980 es más eficiente por lo que será un dispositivo más barato ya que, a pesar de consumir más vatios por segundo, finalizará antes.

Otro escenario podría ser aquel en el que el calor generado importa: quizás hay un alto número de dispositivos y el calor acumulado de todos ellos puede afectar a la durabilidad de algunos componentes del sistema. Entonces, un dispositivo con más eficiencia en lal generación de calor es deseable. Tanto si se necesita o no rendimiento o eficiencia, las diferentes posibles plataformas tendrán diferentes características que harán que la elección de la plataforma dependa de la tarea que

	Menor	Mayor
Poder Computacional	Jetson TK1	GTX 980
Consumo Energético	Jetson TK1	GTX 980
Generación de Calor	Jetson TK1	GTX 480
Eficiencia en Consumo Energético	GTX 480	Jetson tK1
Eficiencia en Generación de calor	Jetson TK1	GTX 980

Tabla 4.2: *Resultados finales.*

se deba hacer, de sus necesidades computacionales y energéticas, y de las condiciones en las que el sistema destino se encuentre, tanto en lo que respecta al calor como en el coste económico que representa su uso en el tiempo.

Bibliography

[1] The Green 500 List. URL http://www.green500.org/.

[2] Q&A Jetson TK1 FAQ, 2014. URL http://developer.download.nvidia.com/embedded/jetson/TK1/docs/Jetson_TK1_FAQ_2014May01_V2.pdf.

[3] Antonio Ruiz, Manuel Ujaldón. Exploiting Kepler Capabilities on Zernike Moments. 2015.

[4] Lisa Gottesfeld Brown. A survey of image registration techniques. ACM Computing Surveys 24:325–376, 1992.

[5] C. Tomasi, R. Manduchi. Bilateral Filtering for Gray and Color Images. *Proceedings of the 1998 IEEE International. Conference on Computer Vision. Bombay. India*, 1998.

[6] Lin-Ching Chang, Esam El-Araby, Vinh Q. Dang, and Lam H. Dao. GPU accel-

eration of nonlinear diffusion tensor estimation using CUDA and MPI. *Neurocomputing*, pages 328–338, 2014.

[7] Chris McClanahan. History and Evolution of GPU Architecture. 2010.

[8] Christos Kyrkou. Stream Processors and GPUs: Architectures for High Performance Computing. Unknown.

[9] Centre for Medical Image Computing CMIC. NifTK description at the CMIC Software and Open Source Database page, 2015. URL http://cmic.cs.ucl.ac.uk/home/software/.

[10] Nvidia Corporation. CUDA Imaging Samples. URL http://docs.nvidia.com/cuda/cuda-samples/#imaging.

[11] Nvidia Corporation. Fermi White Paper, 2009. URL http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.

[12] Nvidia Corporation. GeForce GTX 480 Specifications, 2009. URL http://www.nvidia.es/object/product_geforce_gtx_480_es.html.

[13] Nvidia Corporation. Nvidia announcements at Consumer Electronics Show (CES), 2014. URL http://www.nvidia.com/object/ces2014.html.

[14] Nvidia Corporation. GeForce GTX 980 Whitepaper, 2014. URL http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF.

[15] Nvidia Corporation. Meet the Jetson Embedded Platform, 2014. URL https://developer.nvidia.com/meet-jetson-embedded-platform.

[16] Nvidia Corporation. GeForce GTX 980 Specifications, Late 2014. URL http://www.nvidia.es/object/geforce-gtx-980-es.html#pdpContent=2.

[17] Mark Harris. Introduction to CUDA C, 2013.

[18] Mark Harris. Jetson TK1: Mobile Embedded Supercomputer Takes CUDA Everywhere, 2014. URL http://devblogs.nvidia.com/parallelforall/jetson-tk1-mobile-embedded-supercomputer-cuda-everywhere/.

[19] Ian Buck. *Stream Computing on Graphics Hardware*. PhD thesis, September 2006.

[20] John Michael McNamee. A Comparison Of Methods For Accurate Summation. *ACM SIGSAM Bulletin*, 38, March 2004.

[21] José A. Lachiondo, Manuel Ujaldón, Regina Berretta, Pablo Moscato. Legendre Moments as High Performance Bone Biomarkers: Computational Methods and GPU Acceleration. *Computer Methods in Biomechanics and Biomedical Engineering*, 2015.

[22] William B. Langdon, Marc Modat, Justyna Petke, and Mark Harman. Improving 3D Medical Image Registration CUDA Software with Genetic Programming. *Proceedings of the 2014 conference on Genetic and evolutionary computation (GECCO 2014)*, pages 951–958, 2014.

[23] Mark Harris. Maxwell: The Most Advanced CUDA GPU Ever Made, 2014. URL `http://devblogs.nvidia.com/parallelforall/maxwell-most-advanced-cuda-gpu-ever-made/`.

[24] Sparsh Mittal and Jeffrey S. Vetter. A survey of methods for analyzing and improving GPU energy efficiency. *CoRR*, abs/1404.4629, 2014. URL `http://arxiv.org/abs/1404.4629`.

[25] Marc Modat, Zeike A. Taylor, Josephine Barnes, David J. Hawkes, Nick C. Fox, and Sebastien Ourselin. Fast free-form deformation using the normalised mutual information gradient and graphics processing units. *Med Phys*, pages 278–284, 2010.

[26] Modat, M., Cash, D. M., Daga, P., Winston, G. P., Duncan, J. S., and Ourselin, S. Global image registration using a symmetric block-matching. *JOURNAL of Medical Imaging*, 1(2):024003–024003, 2014.

[27] Modat, M., Ridgway, G. R., Taylor, Z. A., Lehmann, M., Barnes, J., Hawkes, D. J., Fox, N. C., et al. Fast free-form deformation using graphics processing units. *Computer Methods And Programs In Biomedicine*, 98(3):278–284, 2010.

[28] NVIDIA Corporation. NVIDIA GeForce 8800 GPU architecture overview. Technical report, November 2006.

[29] NVIDIA Corporation. NVIDIA GeForce GTX 200 GPU architectural overview. Technical report, May 2008.

[30] NVIDIA Corporation. NVIDIA's Next Generation CUDA Compute Architecture: Fermi Whitepaper. 2009.

[31] NVIDIA Corporation. NVIDIA GF100 Whitepaper. Technical report, 2010.

[32] NVIDIA Corporation. TESLA K10 GPU Accelerator Board Specification. Technical report, November 2012.

[33] NVIDIA Corporation. NVIDIA Tesla KSeries Data Sheet, October 2012.

[34] NVIDIA Corporation. TESLA K20X GPU Accelerator Board Specification. Technical report, July 2013.

[35] NVIDIA Corporation. TESLA K20 GPU Accelerator Board Specification. Technical report, July 2013.

[36] NVIDIA Corporation. NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110/210 Whitepaper. Technical report, 2014.

[37] NVIDIA Corporation. NVIDIA Jetson TK1 Development Kit. Technical report, May 2014.

[38] NVIDIA Corporation. NVIDIA Embedded Computing Website, 2015. URL https://developer.nvidia.com/embedded-computing.

[39] NVIDIA Corporation. CUDA C Programming Guide, 2015. URL http://docs.nvidia.com/cuda/cuda-c-programming-guide.

[40] NVIDIA Corporation. Official Wiki for NVIDIA's Tegra and Jetson, 2015. URL http://elinux.org/Jetson_TK1.

[41] NVIDIA Corporation. NVIDIA GeForce GTX 980 Whitepaper. Technical report, 2015.

[42] NVIDIA Corporation. NVIDIA Tegra K1 Whitepaper. Technical report, 2015.

[43] NVIDIA Corporation. Maxwell Tuning Guide: 1.4.2.1. Unified L1/Texture Cache, 2015. URL http://docs.nvidia.com/cuda/maxwell-tuning-guide/#l1-cache.

[44] Ourselin, S., Roche, A., Subsol, G., Pennec, X., and Ayache, N. Reconstructing a 3d structure from serial histological sections. *Image and Vision Computing*, 19(1-2):25–31, 2001.

[45] Rana Mohtadi. Magnesium batteries: Current state of the art, issues and future perspectives, 2014. URL http://www.beilstein-journals.org/bjnano/single/articleFullText.htm?publicId=2190-4286-5-143.

[46] Rueckert, D., Sonoda, L. I., Hayes, C., Hill, D. L. G., Leach, M. O., and Hawkes, D. J. Nonrigid registration using free-form deformations: Application to breast mr images. *IEEE Transactions on Medical Imaging*, 18(8):712–721, 1999.

[47] Samuel Williams, Andrew Waterman, David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52:65–76, April 2009. doi: 10.1145/1498765.1498785.

[48] Stephan Soller. GPGPU origins and GPU hardware architecture. 2011.

[49] TechPowerUp. GPU-Z Official Page. URL `http://www.techpowerup.com/gpuz/`.

[50] UC London Translational Imaging Group. NiftyReg: Open-source software for efficient medical image registration, 2015. URL `http://cmictig.cs.ucl.ac.uk/research/software/22-niftyreg`.

[51] Manuel Ujaldón. VIII Curso Avanzado de GPU: Programación y Rendimiento frente a la CPU, 2014.

[52] Vasily Volkov. Better performance at lower occupancy. UC Berkeley Lecture, 2010. URL `http://www.cs.berkeley.edu/~volkov/volkov10-GTC.pdf`.

[53] Barbara Zitová and Jan Flusser. Image registration methods: a survey. *Image and Vision Computing*, 21:977–1000, 2003.

A mi padre, mi madre y mis hermanos, ya que forman parte de mí ahora y siempre.

A Izan y Ade, mi inspiración y alegría más duraderos.

A Nur y Sanjay, con los que pude superar mis límites año tras año en la facultad.