UNIVERSITÉ CÔTE D'AZUR

MASTER INFORMATIQUE

TER REPORT

Study of GP-GPU calculations on graphics cards

Prepared by
Enzo Ghizzardi

Supervised by
Sid Touati
Cinzia Di Giusto

Master in Computer Science
at Université Côte d'Azur
Fall semester 2024-2025

Abstract

This report explores the performance and specificities of parallel computation programming on graphics processing units (GPUs), leveraging the CUDA (NVIDIA) and OpenCL (AMD) frameworks. The study is set in a context where the use of GPUs for massively parallel computations has become essential across various fields, such as machine learning, simulations, and video games.

After an introduction to the hardware, including a comparison of the characteristics and evolution of modern graphics cards, the report analyzes the CUDA and OpenCL frameworks, highlighting their differences in terms of design, features, and hardware optimizations. To conduct performance evaluations, three massively parallel algorithms were implemented: matrix multiplication, Bellman-Ford, and thermal diffusion simulation.

Performance was measured and compared for each algorithm on an AMD RX 6750 XT GPU and an Intel i5 12600KF CPU. These tests include comparisons between CUDA and OpenCL on GPUs, as well as direct performance comparisons between the CPU and GPU.

Finally, the discussion addresses critical questions such as the fundamental differences between CPUs and GPUs, the simultaneous management of graphical interfaces and computational tasks on GPUs, and the impact of data transfers on overall performance.

Contents

Part I
Introduction

1 Context and motivations

Graphics Processing Units (GPUs) play a central role in modern computing, far exceeding their original purpose of accelerating graphical rendering. Their ability to perform massively parallel computations makes them an essential tool for various applications, including artificial intelligence, physical simulations, data analysis, and scientific computing. As computational performance demands continue to grow, GPUs have become the preferred solution for domains once dominated by Central Processing Units (CPUs).

One of the significant challenges of this transition lies in mastering GPU programming models, such as CUDA, developed by NVIDIA, and OpenCL, an open standard for programming GPUs and other accelerators. These frameworks provide distinct paradigms to efficiently leverage GPU capabilities, requiring a deep understanding of concepts like data parallelism, hierarchical memory management, and CPU¡-¿GPU data transfers.

This project is set within this context and aims to explore the hardware and software specificities of modern GPUs, compare the CUDA and OpenCL frameworks, and evaluate computational performance on several algorithms. The study also includes an analysis of the fundamental differences between CPUs and GPUs, highlighting their complementary roles in modern computing.

2 State of the Art and Evolution of Graphics Cards

2.1 Evolution and Characteristics of GPUs

Graphics cards emerged in the 1990s to meet the growing demands of multimedia and video game applications. Initially designed for fixed operations, such as 2D/3D rendering, they evolved into programmable architectures with the introduction of shaders in the 2000s. This transition allowed developers to customize graphical computations, paving the way for more diverse applications.

The introduction of NVIDIA's CUDA programming model in 2007 marked a turning point, enabling GPUs to handle non-graphical workloads. OpenCL, introduced by the Khronos Group in 2009, expanded on this idea by providing a cross-platform solution for programming GPUs and other accelerators.

With the rise of deep learning, GPUs like NVIDIA's Tesla and AMD's Radeon Pro were tailored for scientific and industrial workloads, offering massive performance boosts through specialized optimizations.

Modern GPUs, such as NVIDIA's RTX 4000 series and AMD's Radeon RX 7000 series, combine raw power, energy efficiency, and advanced software tools for AI workloads. They support libraries like CUDA, OpenCL, and Vulkan, as well as popular frameworks like TensorFlow and PyTorch.

These GPUs are distinguished by their high number of parallel cores. For example, NVIDIA's RTX 4090 features over 16,000 CUDA cores, while AMD GPUs rely on Compute Units (CUs), each containing several SIMD (Single Instruction Multiple Data) processors. This architecture enables the simultaneous processing of thousands of threads, optimizing massively parallel computations.

Recently, NVIDIA introduced Tensor Cores for machine learning computations and RT Cores for real-time ray tracing, a rendering technique that realistically simulates lighting, reflections,

refractions, shadows, and indirect illumination with physical accuracy.

Key market players include :

- **NVIDIA**, dominating the market with its proprietary **CUDA** framework.
- **AMD**, competing via **OpenCL**, and offering a competitive price-to-performance ratio for specific workloads.
- **Intel**, a recent entrant into the GPU market with its Intel Arc series, targeting AI applications and general-purpose computing.

2.2 Current Trends

GPUs have witnessed exponential growth in floating-point operations per second (FLOPS) with each generation, albeit at the cost of increased power consumption. Current architectures aim to improve performance-per-watt with innovations such as chiplet designs (AMD) and advanced cooling technologies.

GPUs are now employed in non-traditional contexts, including medical image processing, financial modeling, and physical simulations.

Additionally, the adoption of multi-backend frameworks like OpenCL enhances software interoperability across diverse hardware platforms.

3 Objectives

This report aims to:

1. Analyze the CUDA and OpenCL frameworks, their core concepts.
2. Implement representative algorithms to evaluate and compare performance on CPU and GPU (AMD).
3. Discuss the results and explore intriguing questions related to GPU capabilities.

Part II
Methods and Materials

In this chapter, we focus on the methods and tools used throughout our experiments.

4 Methods

4.1 Principle of the Experiment

The experiment aims to measure, analyze, and compare the computational performance of AMD GPUs and Intel CPUs using three benchmarks designed for massively parallel computations.

4.2 Benchmarks Used

Matrix Multiplication : Matrix multiplication is a fundamental operation in linear algebra, widely used in applications such as image processing, machine learning, and numerical simulations. The objective is to compute C=A×B, where A, B and C are matrices with appropriate dimensions.

$$C[i][j] = \sum_{k=1}^{p} A[i][k] \cdot B[k][j], \forall i \in [1,m], j \in [1,n]$$

Bellman-Ford Algorithm : The Bellman-Ford algorithm is used to find the shortest paths from a source vertex s in a weighted graph with V vertices and E edges. Unlike Dijkstra's algorithm, it can handle negative weights. At each iteration, the algorithm "relaxes" the edges by updating distances. These operations are independent for all edges, enabling edge-level parallelization.

The relaxation formula for each edge (u,v) with weight w(u,v) is:

$$d[v] = \min(d[v], d[u] + w(u,v))$$

Where d[v] is the estimated distance between the source s and vertex v. If the path through u improves the estimated distance for v, then d[v] is updated. The algorithm runs at most V-1 iterations, as the shortest path in a graph with V vertices involves at most V-1 edges.

Heat Diffusion Simulation : The heat diffusion simulation models the propagation of temperature in a 2D grid over a given number of time steps. It is governed by the heat equation and requires iterative updates to the temperature at each grid point.

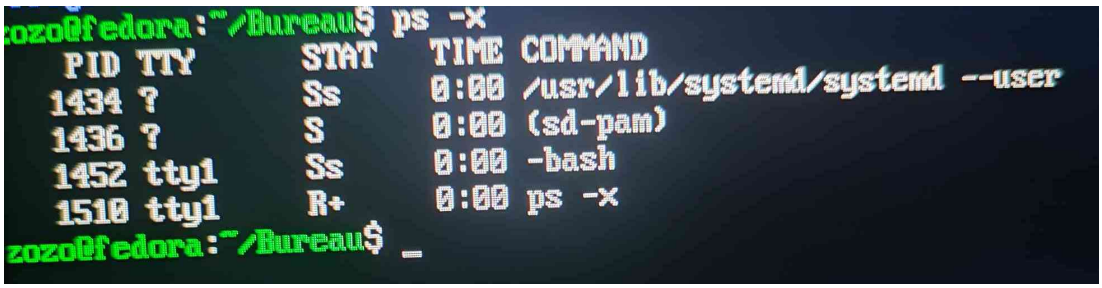
$T[i][j]=\frac{1}{4}(T[i-1][j]+T[i+1][j]+T[i][j-1]+T[i][j+1])$

At each iteration, the temperature of a point depends only on its immediate neighbors, allowing point-level parallelization.

4.3 Methodology and Experimental Environment

Each benchmark is adapted and executed on an NVIDIA GPU, an AMD GPU, and an Intel CPU. Additionally, the benchmarks are executed first on a machine with a GUI (graphical user interface) and then on a machine stripped of its GUI, running with minimal active processes and limited to a command-line interface. This setup allows us to observe any differences in performance between the two configurations.

The list of active processes for the minimal test configuration is shown below:



- `/usr/lib/systemd/systemd --user` : Manages user-specific services and sessions through `systemd` ;
- `(sd-pam)` : A PAM (Pluggable Authentication Modules) session manager handling user authentication and session settings ;
- `-bash` : A **Bash** shell session running in a virtual terminal (`tty1`), loaded with specific environmental variables ;
- `ps -x` : The process listing command executed to generate this list.

Each benchmark is implemented in C, using CUDA and OpenCL libraries to interact with GPUs, and OpenMP to leverage CPU parallelism.

We use the `clock_gettime` function to measure execution times with nanosecond precision.

For GPU benchmarks, execution times include the complete computation time, factoring in data transfer times between the CPU and GPU. Additionally, data transfer times are calculated separately to evaluate their impact.

Software Environment

1. **Software Environment :**
    - **Distribution :** Fedora Linux 41 (Workstation Edition),
    - **Kernel Version :** Linux 6.12.4-200.fc41.x86_64,

- **Firmware Version :** 4.02.

2. **Compiler :**

   - **Name :** GNU Compiler Collection (GCC),
   - **Version :** 12.3.1,
   - **Compilation Options :**
     - For CPU Benchmarks :

       `-O2` : Enables intermediate optimization for a balance of performance and compilation time

       `-fopenmp` : Activates OpenMP support in GCC

       `-lm` : Links to the standard math library `libm`
     - For GPU Benchmarks :

       `-lOpenCL` : Links to the OpenCL library (Open Computing Language)

       `-lm` : Links to the standard math library `libm`

5 Hardware

5.1 Machine Configuration

We will use two key components in our experiments :

1. **Processor :**

   - **Model :** Intel 12th Gen Core i5-12600KF,
   - **Microarchitecture :** Alder Lake, combining Performance (P) cores for demanding tasks and Efficient (E) cores for lighter tasks.
   - **Cores and Threads :**
     - **Physical Cores :** 10 (6 P-cores and 4 E-cores),
     - **Logical Threads :** 16 (Hyper-Threading enabled on P-cores)
   - **Clock Speed :**
     - Base Frequency : 3.7 GHz,
     - Boost Frequency : Up to 4.9 GHz.

2. **GPU AMD :**

   - **Model :** AMD Radeon RX 6750 XT,
   - **Architecture :** RDNA 2, optimized for energy efficiency and improved gaming/scientific performance.
   - **Compute Processors :**
     - Compute Units : 40 CU
     - Stream Processors : 2560 (analogous to CUDA cores in NVIDIA GPUs)
   - **Clock Speed :**
     - Base Frequency : $\approx$ 2150 MHz,

- Boost Frequency : Up to 2600 MHz.

3. **Compute Power (FP32) :** 13.3 TFLOPS

To interact with these GPUs, we will need to understand a little more about how OpenCL works (we will also look at the CUDA documentation).

5.2 About CUDA

CUDA is a programming platform developed by NVIDIA that enables the use of NVIDIA GPUs for general-purpose computing tasks (beyond graphical processing). Using CUDA, programs written in C/C++ (or other compatible languages) can execute computations on the GPU, leveraging its parallel architecture.

Blocks and Grids : CUDA's parallelism model is based on two fundamental units: blocks and grids.

- **Blocks** : A block is a group of threads that execute together on a multiprocessor and share resources, such as shared memory.
- **Grids** : A grid is a collection of blocks. When a CUDA function (kernel) is launched, blocks are organized into a grid.

Threads and Parallelism : Parallelism in CUDA is achieved through threads. Each thread executes independently and has its own memory space. Threads within the same block can cooperate and share data using shared memory, which can enhance efficiency.

The thread management system in CUDA is highly flexible, allowing threads to be organized into blocks and grids according to computational needs. For example, the number of threads per block and blocks per grid can be adjusted to optimize GPU resource utilization.

Streaming Multiprocessors (SMs) : A GPU consists of several Streaming Multiprocessors (SMs), which can execute multiple threads simultaneously. SMs are the computational units responsible for running threads.

Basic Syntax : CUDA code is typically written in C/C++ with CUDA-specific extensions. A function that runs on the GPU is called a kernel. Below is a simple example of launching a kernel:

```
__global__ void addition(int* a, int* b, int* c) {
    // Each thread gets a unique index based on its ID within the block.
    int index = threadIdx.x;
    c[index] = a[index] + b[index]; // Performs the addition of elements from arrays
    a and b, and stores the result in c.
}

int main() {
    const int size = 10; // Declares the size of the arrays to be added.
    int a[size], b[size], c[size]; // Declares three arrays in CPU memory.
    // Initialization of a and b... // The values of arrays a and b should be initialized here
```

```
    int *d_a, *d_b, *d_c; // Declares pointers for GPU memory.

    // Allocates memory on the GPU to store the arrays a, b, and c.
    cudaMalloc((void**)&d_a, size * sizeof(int));
    cudaMalloc((void**)&d_b, size * sizeof(int));
    cudaMalloc((void**)&d_c, size * sizeof(int));

    // Copies the arrays a and b from the CPU to the GPU.
    cudaMemcpy(d_a, a, size * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size * sizeof(int), cudaMemcpyHostToDevice);

    // Launches the addition kernel with 1 block and 'size' threads.
    addition<<<1, size>>>(d_a, d_b, d_c);

    // Copies the results from the GPU (d_c) back to the array c on the CPU.
    cudaMemcpy(c, d_c, size * sizeof(int), cudaMemcpyDeviceToHost);

    // Frees the allocated memory on the GPU to avoid memory leaks.
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);

    return 0; // Ends the program.
}
```

In this simple program, an array is processed by parallel threads. The function `addition` is a kernel executed on the GPU. The line `addition<<<1, taille>>>` launches the kernel with a grid containing 1 block and a number of threads equal to `size`.

Memory in CUDA : CUDA supports several types of memory, each with specific characteristics:

- Global Memory: Visible to all threads but with high latency.
- Shared Memory: Accessible only to threads within the same block. It is fast and useful for thread collaboration.
- Local Memory: Used for variables private to a thread.
- Constant Memory: Optimized for read-only data that does not change during execution.

Selecting the appropriate memory type is essential for performance optimization.

Performance Optimization : To maximize GPU performance, various aspects of the CUDA application need optimization:

Occupancy: Refers to the proportion of GPU resources utilized during kernel execution. Maximizing occupancy involves choosing an appropriate block size and minimizing per-thread memory

usage.

Global Memory Access: Global memory has high latency. To reduce this, memory accesses should be coalesced—organized so that threads access contiguous memory blocks..

Shared Memory Usage: Shared memory is much faster than global memory but limited in size. To improve performance, global memory accesses should be minimized and replaced with shared memory whenever possible.

CUDA documentation provides a comprehensive reference for understanding and leveraging NVIDIA GPUs' capabilities. It serves as a foundation for designing high-performance parallel programs while addressing challenges such as memory management and optimization.

5.3 About OpenCL

OpenCL (Open Computing Language) is an open standard for parallel computing across heterogeneous platforms, including GPUs. Created by the Khronos Group, OpenCL enables developers to write programs that can run on different types of processors (CPU, GPU, DSP) by exploiting their parallel architectures.

In this research project, OpenCL was used to program parallel algorithms executed on an AMD RX 6750 XT GPU. This section outlines key aspects of OpenCL documentation and explains how its concepts and methods were applied in designing the benchmarks.

Key OpenCL Concepts : To understand OpenCL programming, it is essential to grasp a few key concepts:

1. Platforms and Devices: OpenCL defines a platform as the set of available computational devices. A device can be a CPU or GPU, and each device consists of compute units responsible for executing threads (referred to as "work-items" in OpenCL).

2. Context: An OpenCL context groups a set of devices and manages resources such as memory and command queues.

3. Command Queue: Commands (e.g., program execution or memory transfers) are submitted to devices via a command queue. The command queue defines the order in which tasks are executed on the device.

4. Kernels: Kernels are functions written in OpenCL C, executed on devices such as GPUs. An OpenCL program consists of multiple kernels running in parallel on different devices.

5. Work-items and Work-groups: Work-items are individual units of kernel execution. They are organized into work-groups, enabling efficient workload distribution across GPU compute units.

Programming in OpenCL

Initialization : Start by querying available platforms using `clGetPlatformIDs`, then obtain computational devices with `clGetDeviceIDs`. Once devices are identified, create a context using `clCreateContext` to group them.

Command Queue Creation : After creating the context, initialize a command queue with `clCreateCommandQueue`. This queue is essential for sending commands to the device, such as kernel execution or memory transfers.

Writing Kernels Kernels are written using OpenCL C, a slightly restricted version of C optimized for parallel computing. A typical kernel includes function arguments (e.g., input and output memory) and contains the instructions executed by each work-item.

Example kernel for vector addition :

```
__kernel void vector_add(__global const float *a,
    __global const float *b, __global float *result) {
    int gid = get_global_id(0);
    result[gid] = a[gid] + b[gid];
}
```

This kernel takes as input two arrays a and b and stores the result of the addition element by element in `result`.

Compilation and Execution : Compile the kernel using `clCreateProgramWithSource` then `clBuildProgram`.

Exécution du Kernel Link the kernel to buffers with `clSetKernelArg`, then execute it using `clEnqueueNDRangeKernel`. This function takes as parameters the global and local size of the problem, which defines how the work-items are organized.

Attached is an example of a benchmark I used for matrix multiplication, which applies what has just been said.

OpenCL documentation, enriched with AMD extensions, provides a robust framework for GPU programming. Although more complex than CUDA, OpenCL offers unmatched flexibility for writing efficient algorithms across a wide range of hardware. The concepts described in the documentation were directly applied in the benchmarks.

5.4 Comparison of CUDA and OpenCL

CUDA and OpenCL are two parallel programming models used to harness the power of GPUs. Although they share similar goals, their designs, scopes, and uses differ significantly. This section provides a detailed comparison by analyzing their main features, strengths, weaknesses, and application domains.

CUDA is heavily optimized for NVIDIA GPUs, providing deep integration with the hardware. This optimization results in better performance on NVIDIA cards, but limits its use to NVIDIA hardware.

OpenCL takes a universal approach, compatible with various hardware (GPUs, CPUs, FPGAs). However, this flexibility comes with increased complexity and often lower performance than CUDA on NVIDIA GPUs.

CUDA compiles kernels statically, which allows for better compile-time optimization.

OpenCL compiles kernels dynamically, providing greater flexibility but longer compilation times.

CUDA and OpenCL represent two distinct approaches to GPU programming. CUDA, as a proprietary solution, offers optimal performance and simplicity on NVIDIA GPUs, but lacks portability. OpenCL, on the other hand, offers a universal standard for a wide range of hardware, at the cost of greater complexity and slightly lower performance on some devices. The choice between the two therefore depends on the specific needs of the project: optimized performance (CUDA) or cross-platform flexibility (OpenCL).

We are now ready to put the experiment into action.

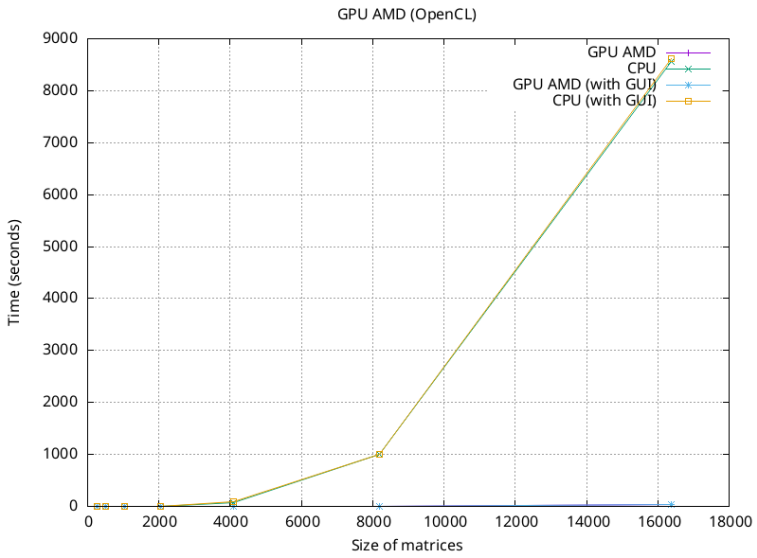
Part III
Results

This part will be devoted to the presentation of the results obtained (and organized in the form of tables and graphs) after execution of the different benchmarks.

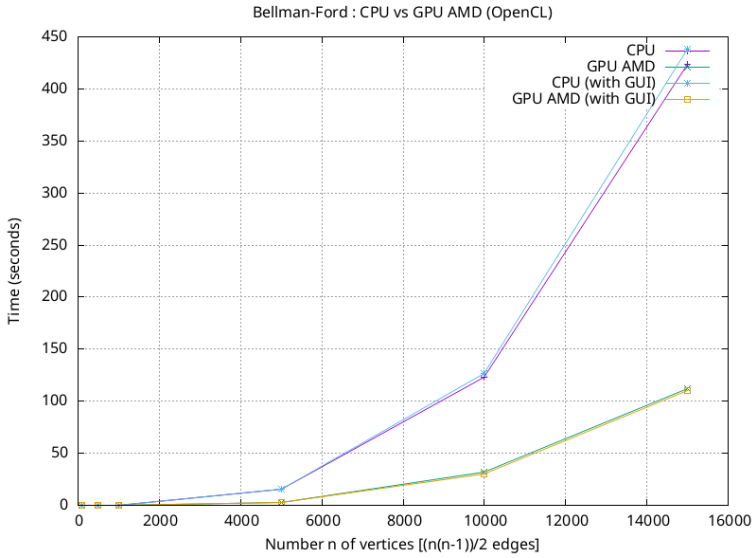
6 GPU vs CPU (with OpenMP directives)

We will be comparing the performance of the AMD GPU to my Intel CPU. The CPU benchmarks will use OpenMP to take advantage of the parallelism enabled by the multiple threads of my i5 12600kf, and thus try to compete with the RX 6750XT.

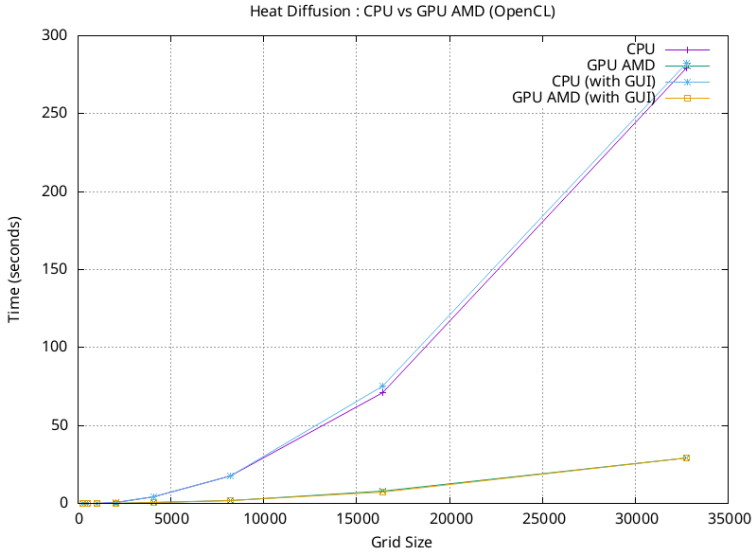
6.1 Matrix multiplication



6.2 Bellman-Ford



6.3 Heat Diffusion Simulation



Part IV
Discussions

In this section, we delve deeper into the results and explore specific topics related to this research project.

7 Discussion of Results

We first observed that running benchmarks had a minor yet slightly noticeable impact on CPU computations. However, for GPU computations, the impact was imperceptible for the dataset sizes used in this study. This is likely because, during the execution of the benchmarks, although the graphical interface (GUI) was active, no programs with significant GPU utilization were launched.

Regarding the performance comparisons between the Intel CPU and AMD GPU across the first three benchmarks, we noticed that as the dataset size increases, the GPU's performance exponentially surpasses that of the CPU. This is because GPUs are designed to handle massively parallel computations more efficiently. Conversely, for smaller dataset sizes, CPUs sometimes outperform GPUs due to their high clock frequencies.

For example, in the case of matrix multiplication with a size of 256, the CPU was more efficient than the GPU. We also observed that data transfer times (first from CPU to GPU, then back) outweighed the time required solely for computation.

This demonstrates that even in massively parallel computations, the dataset size must be sufficiently large for the benefits of GPU parallelism to outweigh the overhead of data transfers.

8 Further Insights

8.1 GPU Resource Management

An important question to address is: how does the GPU manage its tasks and resources when running a benchmark with an active graphical interface (GUI)?

The GPU is divided into Streaming Multiprocessors (SMs), which execute CUDA or OpenCL threads. GPU tasks (graphical rendering + GPU computations) are scheduled via an integrated hardware scheduler. This scheduler dynamically divides resources between graphical tasks (often prioritized to avoid visual slowdowns) and computation tasks (benchmarks, simulations). It allocates SMs dynamically based on task requirements and load.

However, if a benchmark consumes too many GPU resources, graphical performance may degrade (e.g., lag, reduced frame rates). If a benchmark uses a large number of threads or blocks (exceeding the capacity of the SMs), the GPU may become saturated.

Additionally, if your benchmark allocates a significant amount of memory (e.g., for very large matrices), it may exhaust available GPU memory, leading to allocation failures or slowdowns as data is swapped to RAM via the PCIe bus.

8.2 What calculations for GPUs?

Not all computations are efficient or even suitable for implementation on a GPU. In theory, any sequential computation can be implemented using OpenCL or CUDA on a GPU, even if executed on a single thread. Similarly, recursive computations can be adapted into iterative versions for GPU execution. However, the key question is whether a computation is appropriate for GPU execution.

Several criteria help distinguish computations that are challenging to implement on a GPU and, more importantly, those that are inefficient:

- Computations with complex dependencies :

  Algorithms with complex dependencies, such as pure sequential algorithms, are difficult to implement efficiently on a GPU. For example, non-parallelized versions of Dijkstra's algorithm or graph traversal algorithms with dynamic dependencies (e.g., DFS) are poorly suited for GPUs because they require sequential updates.

- Computations with complex conditional instructions :

  Conditional instructions (branches) can lead to thread divergence, significantly slowing down GPU computations.

- Computations requiring extensive synchronization :

  GPUs are optimized for massive parallelism with minimal synchronization. Algorithms requiring frequent communication or dependencies between threads are difficult to implement efficiently.

- Computations with arbitrary precision requirements :

  GPUs are optimized for floating-point (32-bit and 64-bit) and integer computations. Tasks requiring arbitrary precision (e.g., high-precision symbolic algorithms) are not well-suited for GPUs.

- Computations with limited parallelism :

  GPUs excel when there are thousands or millions of tasks to execute in parallel. Algorithms with limited parallelism do not leverage the architecture's potential.

8.3 The Roles of CPUs and GPUs

A natural question arises from some of the benchmarks' results: if GPUs are so efficient for certain calculations, why do we still rely heavily on CPUs?
The answer lies in understanding that CPUs and GPUs serve different and complementary roles in modern computing.

A CPU's architecture is radically different from that of a GPU. CPUs consist of a small number of powerful cores capable of executing multiple instructions in parallel using techniques such as pipelining, out-of-order execution, and hyper-threading (which multiplies logical cores per physical core). CPUs are highly versatile, capable of handling a wide range of tasks, including complex sequential instructions and conditional branches. Additionally, CPUs have low latency, making them ideal for tasks requiring rapid responses.
These characteristics make CPUs indispensable for certain tasks. For example, they are irreplaceable for managing operating systems (OS), which need to handle hardware interrupts (keyboard, mouse), execute sequential instructions, schedule multiple tasks, and manage hardware resources

efficiently.

In contrast, GPUs consist of thousands of simpler cores organized into groups (Streaming Multiprocessors). These cores are optimized for executing the same code across different datasets and have fast but limited VRAM shared among threads within the same group. GPUs are ideal for massively parallel and independent computations, such as graphical rendering (processing millions or billions of pixels, vertices, or textures in a short time) and data processing (e.g., neural networks in AI).

Rather than replacing CPUs, GPUs are designed to complement them. They work together in modern systems: the CPU handles the OS, peripherals, and sequential tasks, coordinates complex operations, and offloads parallel computations to the GPU. The GPU, in turn, handles tasks like graphical rendering and massively parallel calculations.

Part V
Conclusion

This research project allowed us to understand the characteristics and the role that GPUs play today, in our machines, and even more so in those of professionals.

GPUs are becoming increasingly significant in the modern world. Emerging fields such as AI, machine learning, and many others demand ever-growing resources and parallelism to drive their development and progress—not only in terms of performance but also efficiency. The era when graphics cards were designed solely for graphical computations is long gone. Major players in this field, such as NVIDIA, are now among the most prolific and fastest-growing companies in the world, reflecting the profound impact they will have on our lives in the coming years and already do today.

While the future of GPU technology is bright, challenges such as energy efficiency, accessibility, and integration with emerging computing paradigms like quantum computing will define the next frontier in GPU development.

A Appendix

OpenCL Benchmark for Matrix Multiplication

```
#define CL_TARGET_OPENCL_VERSION 220
#include <CL/cl.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define MATRIX_SIZE 16384

const char* kernelSource = "__kernel void matmul(__global float* A,
__global float* B, __global float* C, int n) {"
                           "    int row = get_global_id(1);"
                           "    int col = get_global_id(0);"
                           "    float value = 0.0;"
                           "    for (int k = 0; k < n; k++) {"
                           "        value += A[row * n + k] * B[k * n + col];"
                           "    }"
                           "    C[row * n + col] = value;"
                           "}";

int main() {
    int N = MATRIX_SIZE;
    size_t size = N * N * sizeof(float);
    float *A = (float *)malloc(size);
    float *B = (float *)malloc(size);
    float *C = (float *)malloc(size);

    for (int i = 0; i < N * N; i++) {
        A[i] = rand() / (float)RAND_MAX;
        B[i] = rand() / (float)RAND_MAX;
    }

    struct timespec start, end;
    cl_platform_id platform;
    cl_device_id device;
    cl_context context;
    cl_command_queue queue;
    cl_program program;
    cl_kernel kernel;
    cl_mem d_A, d_B, d_C;

    // Initialisation OpenCL
    clGetPlatformIDs(1, &platform, NULL);
    clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device, NULL);
    context = clCreateContext(NULL, 1, &device, NULL, NULL, NULL);
    queue = clCreateCommandQueueWithProperties(context, device, 0, NULL);
```

```
d_A = clCreateBuffer(context, CL_MEM_READ_ONLY, size, NULL, NULL);
d_B = clCreateBuffer(context, CL_MEM_READ_ONLY, size, NULL, NULL);
d_C = clCreateBuffer(context, CL_MEM_WRITE_ONLY, size, NULL, NULL);

// Compilation et exécution du kernel
program = clCreateProgramWithSource(context, 1, &kernelSource, NULL, NULL);
clBuildProgram(program, 1, &device, NULL, NULL, NULL);
kernel = clCreateKernel(program, "matmul", NULL);

clSetKernelArg(kernel, 0, sizeof(cl_mem), &d_A);
clSetKernelArg(kernel, 1, sizeof(cl_mem), &d_B);
clSetKernelArg(kernel, 2, sizeof(cl_mem), &d_C);
clSetKernelArg(kernel, 3, sizeof(int), &N);

// Mesure du temps total
clock_gettime(CLOCK_MONOTONIC, &start);

// Transfert CPU -> GPU
struct timespec transfer_start_1, transfer_end_1;
clock_gettime(CLOCK_MONOTONIC, &transfer_start_1);

clEnqueueWriteBuffer(queue, d_A, CL_TRUE, 0, size, A, 0, NULL, NULL);
clEnqueueWriteBuffer(queue, d_B, CL_TRUE, 0, size, B, 0, NULL, NULL);

clock_gettime(CLOCK_MONOTONIC, &transfer_end_1);

size_t globalSize[2] = {N, N};
size_t localSize[2] = {16, 16};

struct timespec kernel_start, kernel_end;
clock_gettime(CLOCK_MONOTONIC, &kernel_start);

clEnqueueNDRangeKernel(queue, kernel, 2, NULL, globalSize, localSize, 0, NULL, NULL);
clFinish(queue);

clock_gettime(CLOCK_MONOTONIC, &kernel_end);

// Transfert GPU -> CPU
struct timespec transfer_start_2, transfer_end_2;
clock_gettime(CLOCK_MONOTONIC, &transfer_start_2);

clEnqueueReadBuffer(queue, d_C, CL_TRUE, 0, size, C, 0, NULL, NULL);

clock_gettime(CLOCK_MONOTONIC, &transfer_end_2);

clock_gettime(CLOCK_MONOTONIC, &end);

// Calculs des temps
double total_time = (end.tv_sec - start.tv_sec) + (end.tv_nsec - start.tv_nsec) / 1e9;
```

```
    double kernel_time = (kernel_end.tv_sec - kernel_start.tv_sec) +
    (kernel_end.tv_nsec - kernel_start.tv_nsec) / 1e9;
    double transfer_time_1 = (transfer_end_1.tv_sec - transfer_start_1.tv_sec) +
    (transfer_end_1.tv_nsec - transfer_start_1.tv_nsec) / 1e9;
    double transfer_time_2 = (transfer_end_2.tv_sec - transfer_start_2.tv_sec) +
    (transfer_end_2.tv_nsec - transfer_start_2.tv_nsec) / 1e9;
    double transfer = transfer_time_1 + transfer_time_2;

    printf("Taille : %d, Temps total : %f secondes, Temps GPU : %f secondes, Temps transfert : %f se
           N, total_time, kernel_time, transfer);

    clReleaseMemObject(d_A);
    clReleaseMemObject(d_B);
    clReleaseMemObject(d_C);
    clReleaseKernel(kernel);
    clReleaseProgram(program);
    clReleaseCommandQueue(queue);
    clReleaseContext(context);
    free(A);
    free(B);
    free(C);

    return 0;
}
```

References

CUDA C PROGRAMMING GUIDE, by Nvidia

The OpenCL C Speciification, by Khronos OpenCL Working Group

OpenMP 4.0 API C/C++