GPU Performance: Benchmark Analysis

GPU Performance
Centro de Cálculo Numérico

Autor: Álvaro Martínez Collado

Madrid, Septiembre de 2024

Índice

1. Understanding the GPU

(Before starting, it is important to mention that during the study of this project, an NVIDIA GeForce RTX 3050 was used as the base graphics card. Therefore, unless explicitly mentioned in specific sections, this graphics card will be assumed as the default.)

1.1. Introduction to GPUs

A GPU is a specialized electronic circuit designed to accelerate the creation and rendering of images, animations, and videos for display output. GPUs are highly efficient at performing parallel calculations on large blocks of data, making them essential not only for graphics tasks but also for computational workloads like machine learning and scientific simulations.

How is a GPU different from a CPU?

GPUs differ from CPUs in their architecture and processing capabilities. GPUs have many small, simple cores designed for parallel processing, making them highly effective for tasks like image processing and deep learning, which require handling many operations simultaneously. In contrast, CPUs have fewer, more complex cores optimized for general-purpose tasks, including complex decision-making and single-threaded applications.

How is a GPU similar to a CPU?

Despite their differences, GPUs and CPUs both execute instructions and process data using semiconductor technology. They share fundamental principles of binary logic and data management. Additionally, both have evolved to handle a broader range of tasks: GPUs now support general-purpose computing, and CPUs include more cores and specialized instructions for parallel processing.

1.2. Vectorization and parallelism

As previously mentioned, GPUs support both vectorization and parallelization, though differently from CPUs. While CPUs have a few powerful cores optimized for sequential processing and use vectorization, GPUs consist of thousands of smaller cores designed for massive parallelism. This makes GPUs exceptionally efficient for tasks like matrix multiplication, where many operations can be performed concurrently.

Vectorization:

GPUs achieve vectorization by executing the same instruction across multiple cores organized in grids and blocks (or threads in CUDA). Each core processes a different data set simultaneously, enabling efficient large-scale data processing.

Parallelization:

GPUs excel in parallelization by leveraging their many cores to handle thousands of threads concurrently, making them ideal for tasks that can be divided into numerous parallel operations.

1.3. Memory Access

Memory access in GPUs is optimized for high throughput and parallel processing. GPUs use several types of memory with different characteristics:

- **Global Memory:** This is the largest and slowest memory on the GPU, accessible by all cores. It's used for storing large datasets and is suitable for tasks that involve significant data exchange between the CPU and GPU. However, access to global memory can be slow if not managed efficiently.

- **Shared Memory:** Shared memory is much faster than global memory and is used for communication between threads within the same block. It allows for efficient data sharing and reduces the need to repeatedly access global memory, improving performance for tasks with high inter-thread communication.
- **Local Memory:** Each thread has its own local memory, which is private and used for storing data specific to that thread. Local memory helps in storing intermediate results and temporary data.
- **Texture and Constant Memory:** These are specialized types of memory. Texture memory is optimized for spatial locality and is used for image and graphical data, while constant memory is used for read-only data that remains constant across all threads during a kernel execution.

Efficient memory access is crucial for performance, and optimizing memory use involves minimizing global memory accesses, maximizing shared memory usage, and ensuring coalesced memory accesses to reduce latency and improve throughput.

2. GPU Hardware

The hardware architecture of GPUs is designed to facilitate massive parallel processing, making them ideal for tasks that can be parallelized. Key components include:

- **Cores:** GPUs are equipped with thousands of small, simple cores designed for parallel execution. Unlike CPU cores, which are few in number but powerful, GPU cores are numerous and less complex. These cores work together to perform many operations simultaneously, which is crucial for tasks like rendering images and processing large datasets.
- **Streaming Multiprocessors (SMs):** The GPU cores are organized into Streaming Multiprocessors (SMs). Each SM contains multiple cores, a portion of shared memory, and scheduling hardware. SMs manage the execution of threads within a block, handling the scheduling and synchronization necessary for efficient parallel processing.
- **Memory Hierarchy:** The GPU has a complex memory hierarchy designed to balance speed and capacity.
- **Interconnects:** High-speed interconnects, such as NVIDIA's NVLink and AMD's Infinity Fabric, are used to connect multiple GPUs or to facilitate rapid communication between different parts of a GPU. These interconnects enhance performance by allowing for efficient data transfer and coordination among multiple GPUs in a system.

3. GPU Software

GPU software encompasses various tools and frameworks that facilitate the development and execution of parallel programs on GPUs. Major components include:

- **CUDA:** NVIDIA's Compute Unified Device Architecture (CUDA) is a parallel computing platform and API that allows developers to leverage NVIDIA GPUs for general-purpose computing. CUDA provides extensions to standard programming languages such as C, C++, and Fortran, and includes a comprehensive toolkit for developing GPU-accelerated applications. It supports features like kernel programming, memory management, and inter-thread synchronization.
- **OpenCL:** The Open Computing Language (OpenCL) is an open standard for parallel programming across heterogeneous systems, including GPUs from various vendors. OpenCL provides a unified programming model and API that enables developers to write code that can run on different types of processors (CPUs, GPUs, FPGAs) and is suitable for a wide range of applications.

- **Libraries and Frameworks:** Several libraries and frameworks provide pre-built functions and abstractions for GPU programming, making it easier to develop high-performance applications:
  1. **cuBLAS:** A library for performing basic linear algebra operations on NVIDIA GPUs, optimized for performance.
  2. **cuDNN:** A library for deep neural network operations, providing highly optimized routines for deep learning tasks.
  3. **TensorFlow and PyTorch:** Popular machine learning frameworks that utilize GPU acceleration to improve training and inference performance.
- **Driver Software:** GPU drivers are essential for enabling communication between the operating system and GPU hardware. They handle the execution of GPU-accelerated applications, manage resources, and provide support for various APIs and frameworks. Regular updates to drivers ensure compatibility with new software and hardware features.
- **Development Tools:** Various development tools and IDEs (Integrated Development Environments) support GPU programming by providing debugging, profiling, and performance analysis capabilities. Examples include NVIDIA Nsight and AMD ROCm tools.

Understanding these software components and their interaction with GPU hardware is crucial for developing efficient GPU-accelerated applications and achieving optimal performance.

4. Operators Used in Benchmark.

4.1. CUDA

Compute Unified Device Architecture (CUDA) is a parallel computing platform and programming model developed by NVIDIA. It enables developers to leverage NVIDIA GPUs for general-purpose computing tasks beyond traditional graphics rendering. CUDA provides a set of extensions to standard programming languages such as C, C++, and Fortran, allowing developers to write programs that execute on the GPU. CUDA facilitates efficient parallel processing by exposing low-level GPU architecture features, including fine-grained control over memory and execution.

Key Features of CUDA:

- **Parallel Computing Model:** CUDA allows for the execution of multiple threads in parallel on the GPU. Threads are organized into blocks, and blocks are organized into grids, providing a flexible and scalable way to manage parallelism.
- **Memory Hierarchy:** CUDA provides access to different types of memory, including global, shared, and local memory, enabling optimization of memory access patterns and efficient data management.
- **Kernel Functions:** In CUDA, functions executed on the GPU are called kernels. Kernels are written in CUDA C/C++ and are launched from the host (CPU) to run on the device (GPU).
- **Thrust Library:** CUDA includes the Thrust library, which provides high-level abstractions for common parallel algorithms, such as sorting and reduction, simplifying development.

To use CUDA with Julia, you need to install the CUDA toolkit and the necessary Julia packages.

Once the CUDA toolkit and Julia packages are installed, you can start utilizing CUDA features in your Julia code. Here are some common tasks:

- **Checking GPU Availability:** Use the ‘CUDA‘ package to check if your GPU is available and properly configured:

```
using CUDA
println(CUDA.has_cuda())
```

- **Allocating and Transferring Data:** Allocate memory on the GPU and transfer data between the host and device:

```
a = CUDA.rand(10)  # Allocate an array on the GPU
b = CUDA.fill(2.0, 10)  # Fill an array with a constant value on the GPU
c = a .+ b  # Perform element-wise addition on the GPU
```

CUDA Utilities: In addition to core CUDA functionalities, several utilities and tools are available to help with development and performance optimization:

- **NVIDIA Nsight:** A suite of tools for debugging and profiling CUDA applications, providing insights into performance and helping identify bottlenecks.
- **CUDA Profiler:** Built into the CUDA toolkit, this tool allows for detailed performance analysis of GPU applications, including memory usage and execution time.
- **CUDA Samples:** The CUDA toolkit includes sample code and examples that demonstrate various features and best practices for CUDA programming.

Understanding and utilizing CUDA effectively can significantly enhance the performance of computational tasks by leveraging the parallel processing power of NVIDIA GPUs.

5. Benchmarks

In this section, we present benchmarks with the GPU that compare the performance of various computational tasks. The goal is to provide a detailed analysis of the time processes involved and to draw meaningful conclusions about the performance characteristics of the GPU.

Benchmark Methodology

The benchmarks conducted are designed to evaluate the performance of GPU computations to see the differences against the CPU performance. To ensure a fair comparison, the same set of algorithms and operations are executed on both types of hardware, so in this section it's going to be analyzed similar benchmarks to the CPU ones. The benchmarks focus on evaluating:

- **Execution Time:** The time taken to complete the computations on each platform.
- **Scalability:** How performance scales with increasing problem size and complexity.
- **Efficiency:** The computational efficiency of the operations, including memory usage and processing power.

5.1. Simply GPU Matrix Multiplication

This section presents a simple benchmark for evaluating the performance of matrix multiplication on the GPU. The objective is to measure the time taken for matrix multiplication using GFlops with varying matrix sizes. The benchmark uses the Julia programming language and the CUDA library.

The code below performs matrix multiplications for matrices of different sizes and records the time taken for each operation. The results are printed to the console, providing an overview of how the execution time scales with matrix size.

Description of the Benchmark

1. **Matrix Size Variation:** The benchmark tests matrix sizes ranging from 100 to 10,000 in increments of 100.
2. **Matrix Initialization:** For each size N, two matrices A and B are initialized with random floating-point values using the 'CUDA.rand' function.
3. **Timing Measurement:** The time required for matrix multiplication $A \times B$ is measured using the 'CUDA.@elapsed' macro. This provides the duration of the multiplication operation.
4. **Results Output:** The matrix size N and the corresponding duration t are printed to the console. The results can be saved to a CSV file, although the line for writing to the file is commented out in this example.

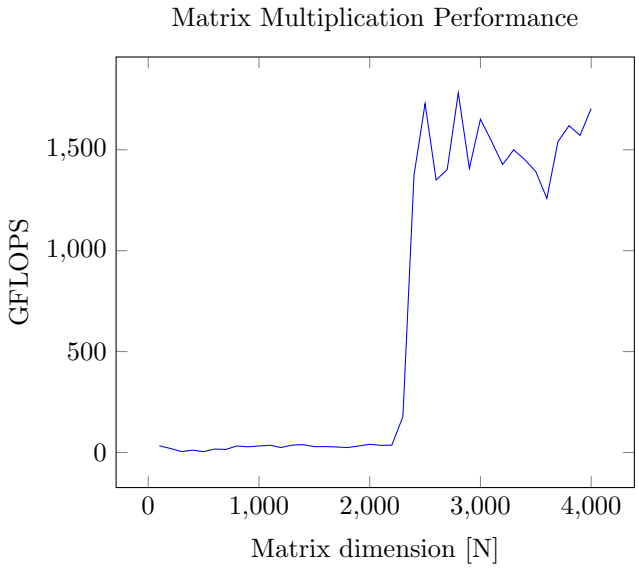

Figura 1: GPU efficiency with matrix multiplications.

5.1.1. Code

Here is the Julia code used for this benchmark:

```julia
import Pkg
Pkg.activate(".")
Pkg.add( "PGFPlotsX" )

using CUDA
using PGFPlotsX

# Initialize arrays to store N and GFLOPS values
x = Int[] # Array to store N values
y1 = Float64[] # Array to store GFLOPS values

for N = 100:100:4000 #limite superior 10000

    A = CUDA.rand(Float32, N, N)
    B = CUDA.rand(Float32, N, N)
    t = CUDA.@elapsed A * B
```

```
17
18      # Compute GFLOPS
19      gflops = (2 * N^3) / (t * 1e9)
20
21      println("N = $(N), Time = $(t), GFLOPS = $(gflops)")
22
23      # Store the values in arrays
24      push!(x, N)
25      push!(y1, gflops)
26
27  end
28
29  # Create the plot using PGFPlotsX
30  plot = @pgf Axis(
31      {
32          xlabel="Matrix dimension [N]",
33          ylabel="GFLOPS",
34          title="Matrix Multiplication Performance",
35      },
36      Plot({no_marks, "blue"}, Table(x, y1)),
37  )
38
39  PGFPlotsX.save("/ALVARO/Documentacion GPU/doc_latex/code/1/grafico_matrix_mult.tex", plot, include_preamble=
        false)
```

5.2. Matrix Multiplication Modifying N using Times

In this section, we present a benchmark that measures the performance of matrix multiplication on the GPU. Specifically, we analyze the duration required for matrix multiplication as the size of the matrices varies. The benchmark is conducted using the Julia programming language with the CUDA library.

The benchmark procedure involves varying the size of the matrices and recording the time taken for a series of matrix multiplications. The results are then saved to a CSV file for further analysis.

Description of the Benchmark

1. **Matrix Size Variation:** The benchmark varies the size of the matrices from 300 to 2499 with a step size of 25.
2. **Number of Operations:** For each matrix size N, the number of operations 'TIMES' is calculated to ensure that the total number of operations is constant for different matrix sizes. The formula used is:

$$\text{TIMES} = \frac{N_{\text{ops}}}{2 \times N^3}$$

   where N_{ops} is set to 2×10000^3.
3. **Timing Measurement:** The time taken to perform the matrix multiplications is measured using the '@elapsed' macro. The benchmark performs 'TIMES' multiplications and records the total duration.
4. **Data Recording:** The results, including matrix size, number of operations, and duration, are written to a CSV file for analysis.

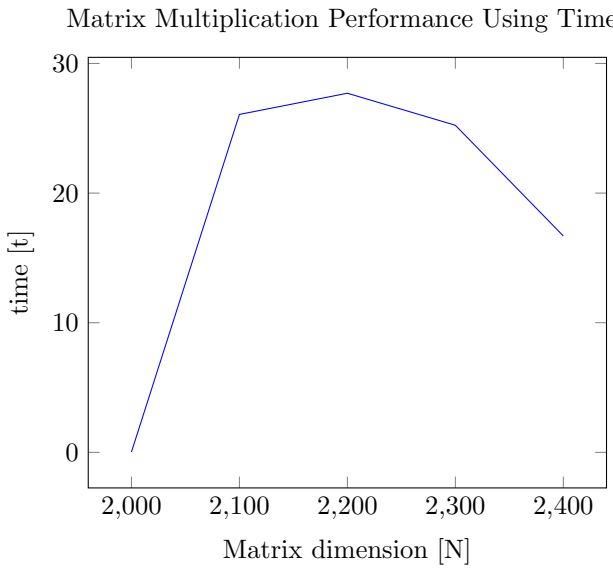

Figura 2: GPU efficiency with matrix multiplications.

5.2.1. Code

Below is the Julia code used to perform this benchmark:

```julia
import Pkg
Pkg.activate(".")
Pkg.add( "PGFPlotsX" )

using CUDA
using PGFPlotsX

# Initialize arrays to store N and GFLOPS values
x = Int[] # Array to store N values
y1 = Float32[] # Array to store GFLOPS values

N_ops = 2 * 10000.0^3
for N = 2000:100:2499 #limite inf era 300, 25 el paso y 2499 limite superior

    TIMES = div(N_ops, 2 * N^3)

    A = CUDA.rand(Float32, N, N)
    B = CUDA.rand(Float32, N, N)

    t = @elapsed begin
        for i = 1:TIMES
            A * B
        end
    end
    println("$N, $TIMES, $t")

    # Store the values in arrays
    push!(x, N)
    push!(y1, t)
end

# Create the plot using PGFPlotsX
plot = @pgf Axis(
    {
        xlabel="Matrix dimension [N]",
```

```
36          ylabel="time [t]",
37          title="Matrix Multiplication Performance Using Times",
38      },
39      Plot({no_marks, "blue"}, Table(x, y1)),
40  )
41
42  PGFPlotsX.save("/ALVARO/Documentacion GPU/doc_latex/code/2/grafico_matrix_mult_times.tex", plot,
        include_preamble=false)
```

5.3. GPU Matrix Multiplication Performance GFLOPS

This section describes a benchmark that measures the performance of matrix multiplication on the GPU using an NVIDIA GeForce GTX 1060 6GB. The objective is to evaluate the time taken for matrix multiplication across various matrix sizes and compare it to theoretical performance estimates. The benchmark uses the Julia programming language along with the CUDA and Plots libraries.

The following Julia code performs the matrix multiplication and measures the execution time. It also plots the performance results in GFLOPS (Giga Floating Point Operations per Second) to visualize how the GPU handles different matrix sizes.

Description of the Benchmark

1. **Matrix Initialization:** The matrix initialization GPU function initializes matrices A and B with random floating-point values and transfers them to the GPU as 'CuArray'.
2. **Matrix Multiplication:** The matrix multiplication function performs matrix multiplication on the CPU, while the matrix multiplication GPU function measures the time required for GPU-based matrix multiplication using the 'CUDA.@elapsed' macro.
3. **Timing Measurement:** The time matrix multilication function measures the duration of matrix multiplications for different matrix sizes N. It calculates the time per operation and compares it to theoretical performance estimates.
4. **Results Plotting:** The plot results function generates a plot of the GPU performance in GFLOPS versus matrix size N. It also includes a label indicating the maximum GFLOPS achieved.
5. **Data Analysis:** The benchmark results are printed to the console and visualized using the 'Plots' library. The maximum GFLOPS achieved is highlighted in the plot for comparison with theoretical performance.

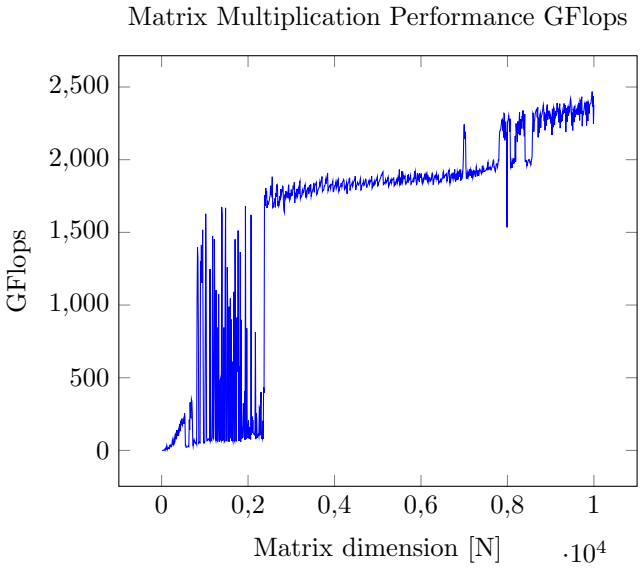

Figura 3: GPU efficiency with matrix multiplications.

5.3.1. Code

Here is the Julia code used for this benchmark:

```
import Pkg
Pkg.activate(".")
Pkg.add( "PGFPlotsX" )

using Plots
using LinearAlgebra
using CUDA

############################################################
############### INITIALIZATION OF MATRICES ###############
############################################################

function matrix_initialization_GPU(N)
  # N = 10000

  A = rand(Float32, N, N) # Initialize matrix A with random values
  B = rand(Float32, N, N) # Initialize matrix B with random values

  d_A = CUDA.CuArray(A) # Transfer matrix A to GPU
  d_B = CUDA.CuArray(B) # Transfer matrix B to GPU

  return d_A, d_B
end

############### MATRIX MULTIPLICATION ###############
function matrix_multiplication(A, B)
  return A * B # Perform matrix multiplication on CPU
end

function matrix_multiplication_GPU(A, B)
  return CUDA.@elapsed A * B # Measure time taken for matrix multiplication on GPU
end

function time_matrix_multiplication(N, N_cores, matinit, matmul)
  Time = zeros(length(N)) # Array to store computation times
  Theoretical_time = 1e9 / (4e9 * 512/32 * N_cores) # Theoretical time per operation

  for (i, n) in enumerate(N) # Loop through each matrix size
    A, B = matinit(n) # Initialize matrices
    m = length(B)

    t1 = time_ns() # Start time measurement
    matmul(A, B) # Perform matrix multiplication
    t2 = time_ns() # End time measurement

    dt = t2 - t1
    Time[i] = dt / (2 * n * m) # Calculate time per operation

    println("N=", n, " Time per operation =", Time[i], " nsec")
    println("N=", n, " Theoretical time per operation =", Theoretical_time, " nsec")
  end

  return Time, Theoretical_time
end

function plot_results(GFLOPS, GFLOPS_max, title, ymax)
  xlabel!("N") # Label x-axis
  display(plot(N, GFLOPS, ylims=(0, ymax), title=title, minorgrid=true)) # Plot GFLOPS
  display(plot!(N, GFLOPS_max * ones(length(N)), minorgrid=true)) # Plot theoretical GFLOPS
end

N_cores = 1
N = Vector(10:10:10000) # Define matrix sizes
Time, Theoretical_time = time_matrix_multiplication(N, N_cores, matrix_initialization_GPU,
```

```
    matrix_multiplication_GPU)
GFLOPS_GPU = 1 ./ Time # Compute GFLOPS
GFLOPS_max = 1 / Theoretical_time # Compute maximum theoretical GFLOPS

# Create the plot using PGFPlotsX
plot = @pgf Axis(
    {
        xlabel="Matrix dimension [N]",
        ylabel="GFlops",
        title="Matrix Multiplication Performance GFlops",
    },
    Plot({no_marks, "blue"}, Table(N, GFLOPS_GPU)),
)

PGFPlotsX.save("/ALVARO/Documentacion GPU/doc_latex/code/3/grafico_GFLOPS_GPU.tex", plot, include_preamble=
    false)
```

5.4. Performance GFLOPS NVIDIA GeForce GTX 1060 6GB

LINK GTX 1060. https://www.techpowerup.com/gpu-specs/geforce-gtx-1060-6-gb.c2862
The NVIDIA GeForce GTX 1060 6GB is a graphics card from the GTX 10 series, released in 2016, offering significant performance for parallel processing and intensive computation tasks. To evaluate its processing capability, the GFLOPS (Giga Floating Point Operations Per Second) metric is used, which measures the number of floating-point operations the GPU can perform in one second.

5.4.1. Key Specifications

To calculate the GFLOPS performance of the GTX 1060 6GB, we consider the following specifications:

- **Number of CUDA Cores:** 1,280
- **Base GPU Clock:** 1,506 MHz
- **Boost GPU Clock:** 1,708 MHz
- **Number of Floating Point Operations per Cycle per Core:** 2 (each CUDA core can perform 2 floating-point operations per cycle)

5.4.2. GFLOPS Calculation

GFLOPS performance is calculated using the formula:

$$\text{GFLOPS} = \text{Number of CUDA Cores} \times \text{GPU Clock} \times \text{Operations per Cycle} \times \frac{1}{10^6}$$

where the conversion factor changes the clock frequency from MHz to GHz. Using the boost clock frequency of 1,708 MHz for the calculation, we get:

$$\text{GFLOPS} = 1,280 \times 1,708 \text{ MHz} \times 2 \times \frac{1}{10^3}$$

$$\text{GFLOPS} = 1,280 \times 1{,}708 \times 2$$

$$\text{GFLOPS} = 4,371{,}84 \text{ GFLOPS}$$

5.4.3. Summary

The theoretical maximum performance of the NVIDIA GeForce GTX 1060 6GB is approximately **4.37 TFLOPS** (teraflops) for single-precision (FP32) floating-point calculations. This figure provides an indication of the GPU's potential to handle floating-point operations in high-performance applications such as scientific simulations, machine learning, and graphics processing.

It is important to note that actual performance may vary depending on the specific workload, software implementation efficiency, and operational conditions of the GPU.

5.5. Performance in GFLOPS of the NVIDIA GeForce RTX 2060 Laptop

LINK GTX 1060. https://www.techpowerup.com/gpu-specs/geforce-rtx-2060.c3310
The NVIDIA GeForce RTX 2060 Laptop is a high-performance graphics card designed for laptops, offering significant computational power for various demanding tasks. To assess its performance, we use the GFLOPS (Giga Floating Point Operations Per Second) metric, which measures how many floating-point operations the GPU can perform in one second.

5.5.1. Key Specifications

For calculating the GFLOPS performance of the RTX 2060 Laptop, we consider the following specifications:

- **Number of CUDA Cores:** 1,920
- **Base GPU Clock:** 1,050 MHz
- **Boost GPU Clock:** 1,440 MHz
- **Number of Floating Point Operations per Cycle per Core:** 2 (each CUDA core can perform 2 floating-point operations per cycle)

5.5.2. GFLOPS Calculation

The GFLOPS performance can be calculated using the formula:

$$\text{GFLOPS} = \text{Number of CUDA Cores} \times \text{GPU Clock} \times \text{Operations per Cycle} \times \frac{1}{10^6}$$

where the conversion factor adjusts the clock frequency from MHz to GHz. Using the boost clock frequency of 1,440 MHz for the calculation, we get:

$$\text{GFLOPS} = 1,920 \times 1,440 \text{ MHz} \times 2 \times \frac{1}{10^3}$$

$$\text{GFLOPS} = 1,920 \times 1{,}44 \times 2$$

$$\text{GFLOPS} = 5,529{,}60 \text{ GFLOPS}$$

5.5.3. Summary

The theoretical maximum performance of the NVIDIA GeForce RTX 2060 Laptop is approximately **5.53 TFLOPS** (teraflops) for single-precision (FP32) floating-point calculations. This figure provides a measure of the GPU's capability to handle floating-point operations efficiently, which is crucial for tasks such as high-resolution gaming, complex simulations, and professional creative applications.

It is important to note that the actual performance may vary based on the specific workload, the efficiency of the software implementation, and the operating conditions of the GPU.