Comparative Performance Analysis of hashing on GPU and CPU

Simone Mulazzi s330471
Tommaso Gualtierotti s329534

GPU Programming

Dipartimento di Automatica e Informatica
Politecnico di Torino

Torino, June 2025

Contents

Figures

Tables

Listings

1. Introduction

Hashing algorithms are a foundational component in modern cryptography and data integrity systems. These algorithms transform input data of arbitrary size into fixed-length outputs —called hashes—that serve as unique digital fingerprints. Widely adopted hashing functions like SHA-1, SHA-256, and MD5 play a vital role in areas such as digital signatures, message authentication, password protection, blockchain validation, and file integrity verification.

As global data volumes continue to grow exponentially, the demand for rapid and repeated hashing operations has increased dramatically. Whether verifying the integrity of large datasets, processing transactions in distributed ledgers, or securing massive volumes of network traffic, the computational workload can quickly exceed the capabilities of traditional CPU-based solutions. Although modern CPUs offer multi-core parallelism, they are fundamentally limited in their ability to scale hashing tasks due to their lower core count and general-purpose architecture.

In contrast, hashing workloads exhibit a high degree of data-level parallelism, as each input can be processed independently of the others. This makes hashing particularly well-suited for execution on GPUs, which are designed to handle thousands of concurrent threads efficiently. By mapping hashing computations across many GPU cores, it is possible to achieve significant acceleration, especially in batch-oriented or high-frequency environments.

The need for parallelization becomes even more critical in security-related scenarios. In brute-force attacks—whether for password recovery, key testing, or cryptanalysis—vast numbers of inputs must be evaluated against known hash outputs. Since each candidate can be processed independently, GPUs allow attackers and researchers alike to explore extensive input spaces in significantly less time.

Parallel execution also plays an essential role in cryptographic research, especially for analyzing hash function behavior. Tasks such as detecting collisions (finding two different inputs that produce the same hash) or measuring statistical properties of hashes over large datasets require evaluating millions or even billions of inputs. GPU-accelerated hashing dramatically reduces the time needed for such analysis, enabling deeper and more comprehensive testing of algorithmic resilience.

While specialized hardware accelerators such as ASICs or FPGAs offer even greater performance for cryptographic operations, they come with trade-offs: limited availability, higher cost, and reduced flexibility. For many applications, general-purpose GPUs offer a highly accessible and scalable alternative, capable of delivering high-throughput performance without the need for custom hardware.

This project investigates the effectiveness of parallel hashing on GPU architectures, using CUDA to implement high-performance, flexible hashing workflows. The system supports widely used hash functions—SHA-1, SHA-256, and MD5—and is designed to exploit the GPU's massive parallel capabilities through batch processing, CUDA streams, and asynchronous memory transfers. One of the primary objectives is to benchmark and compare the execution times of hashing operations on GPUs versus traditional CPU-based implementations. This comparison provides valuable insights into the performance gains achievable through parallelization and helps evaluate the scalability of each platform under increasing workloads. By enabling fast, large-scale hashing operations, the project supports both practical applications —such as secure data processing and cryptographic authentication—and theoretical research, including brute-force analysis and hash collision detection. The result is a versatile and performance-aware framework that demonstrates how modern GPUs can effectively address the computational demands of today's data-intensive and security-critical environments.

2. Implementation

2.1. GPU implementation

The implementation of this project focuses on accelerating cryptographic hash functions using CUDA-enabled GPUs, with the aim of significantly improving the execution time of hashing workloads through parallel computing techniques. The design was built to be modular and extensible, supporting multiple widely used hashing algorithms — namely SHA-1, SHA-256, and MD5. The primary objective was not only to execute these algorithms in parallel on the GPU but also to provide a mechanism for benchmarking and comparing performance against a CPU-based implementation, using the same data, configurations, and constraints.

2.2. General idea

The concept behind the execution is shown in Figure 1:

1. Read data from the dataset.
2. Copy data from host (CPU) to device (GPU).
3. Execute the algorithm on the device (GPU).
4. Copy data (hashes) from device (GPU) to host (CPU).

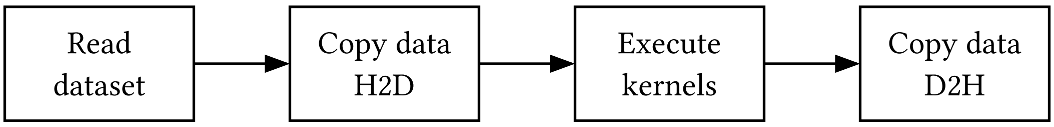

Figure 1: General GPU execution flow

2.3. Batch Processing

At the core of the system lies a batch processing mechanism, which is essential in case of very large datasets, since the whole content of the dataset cannot fit either in the VRAM of the GPU or the RAM of the CPU or both of them.In this approach the content of the dataset is read batch by batch and the memory to store the content is allocated only once.
In this way the algorithms can be executed either using only one reading batch, i.e. storing the whole dataset in RAM (when possible), or using many batches with a fixed size choosed at compile-time.
Batch reading can be used by both the CPU and GPU version of the algorithms. In Figure 2 we can see a representation of how the batch processing is achieved and how the GPU executes the algorithms.

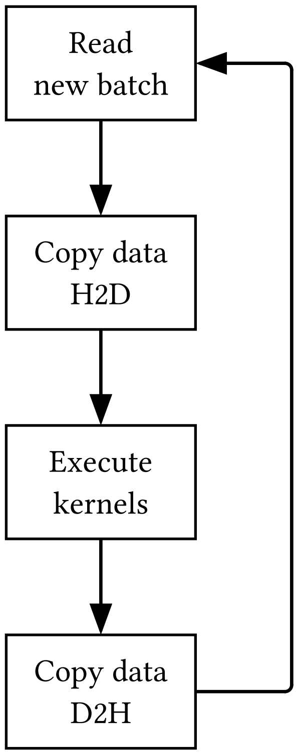

Figure 2: General GPU execution flow

Listing 1 shows how the batch reader struct is implemented and which functions provides. Here follows a brief description of both **`batch_reader_t`** struct's attributes and funcctions.

1. `batch_reader_t` attributes
   a. `File *file`: pointer to the dataset from which items are read.
   b. `size_t buffer_pos`: how many lines are read.
   c. `total_items`: total number of lines in the dataset.
   d. `batch_read_items` lines read in the current batch.
   e. `iterations`: number of batches read so far.
2. `batch_reader_init`: allocates and initializes a `batch_reader_t` structure, opens the file, counts total lines, and rewinds the file.
3. `batch_read_strings_from_file`: reads up to BATCH_NUM_LINES lines into preallocated buffers, zeroing or allocating as needed on each call.
4. `batch_reader_close`: closes the file and frees the batch_reader_t struct.
5. `batch_reader_rewind_file`: rewind the batch reader's file to the beginning.

```c
typedef struct {
    FILE *file;
    size_t buffer_pos;
    size_t total_items;
    size_t batch_read_items;
    size_t iteration;
} batch_reader_t;

batch_reader_t* batch_reader_init(const char *filename);

size_t batch_read_strings_from_file(batch_reader_t *batch_reader,
                                     char ***strings,
                                     size_t **lengths);

void batch_reader_close(batch_reader_t *batch_reader);

void batch_reader_rewind_file(batch_reader_t* batch_reader);
```

Listing 1: Batch reader typedef and utilities functions

Listing 2, shows how the batch reading mechanism is implemented. The following steps are performed:

1. A batch of strings is read from the dataset.
2. The content read from the dataset is copied into a host (CPU side) array, waiting to be copied to GPU.
3. The loop exit condition is checked. If the number of strings read is equal to the number of strings for a single batch, then there might still be strings to be read. Otherwise if a lower number a strings has been read, then the end of the dataset has been reached.

```
    do
    {
        batch_read_strings_from_file(reader, &input_strings, &data_length);
        num_lines = reader->batch_read_items;
        if (num_lines == 0) break;

        memset(h_data, 0, BATCH_SIZE);
        for (size_t i = 0; i < num_lines; i++)
        {
            memcpy(h_data + i * MAX_STRING_LENGTH,
                   input_strings[i],
                   data_length[i]);
        }

        /******************/
        /* CUDA CODE here */
        /******************/

        batch_iteration++;
    } while (reader->batch_read_items == BATCH_NUM_LINES);
```

Listing 2: C implementation of batch reading mechanism

2.4. Kernel implementation

Each hashing algorithm was ported to a CUDA kernel designed to exploit the SIMT (Single Instruction, Multiple Threads) nature of GPU threads. Each thread processes one input data item — for example, computing the SHA-1 hash of a single input in isolation. This model naturally suits hashing algorithms because hash computations are independent, and no inter-thread communication is required. The code presented in Listing 3 is used as reference.

```
__global__ void sha1_kernel(const char *data,
                            uint32_t *hashes,
                            size_t *lengths,
                            size_t num_lines)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < num_lines)
    {
        sha1_device(data + idx * MAX_STRING_LENGTH,
                    lengths[idx],
                    hashes + idx * SHA1_HASH_LENGTH);
    }
}
```

Listing 3: SHA1 kernel

In Listing 4, is shown the relevant portion of CUDA code of the wrapper function for parallel execution of SHA1 on the GPU.
Here the following are performed:

1. The dataset's strings are copied from the Host memory to the GPU memory.

2. The number of blocks, based on the number of threads, to launch enough threads to compute hashes of all the strings previously copied from host memory is computed.
3. The kernel is actually launched.
4. A check on possible errors arising from kernel execution is performed.
5. Device is *synchronized* in order to wait for all threads to finish their work, before accessing results from Device memory.
6. Hashes are copied from Device to Host.

```
CHECK_CUDA_ERROR(cudaMemcpy(d_data,                                    h_data,
total_size, cudaMemcpyHostToDevice));
CHECK_CUDA_ERROR(cudaMemcpy(d_lengths, lengths, num_lines * sizeof(size_t),
cudaMemcpyHostToDevice));

size_t blocks = (num_lines + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK;

sha1_kernel<<<blocks,  THREADS_PER_BLOCK>>>(d_data,  d_hashes,  d_lengths,
num_lines);

CHECK_CUDA_ERROR(cudaGetLastError());
CHECK_CUDA_ERROR(cudaDeviceSynchronize());

CHECK_CUDA_ERROR(cudaMemcpy(
                        hashes,
                        d_hashes,
                        num_lines * SHA1_HASH_LENGTH * sizeof(hashes[0]),
                        cudaMemcpyDeviceToHost));
```

Listing 4: Cuda code of SHA1 kernel wrapper

2.5. Streams execution

For what concerns GPU execution of the algorithms, streams execution is made available to improve the performance of the kernel execution. In this way, as can be seen in Figure 3, in case of very large datasets the three operations **`Copy from H2D`**, **`Execute kernel`**, **`Copy from D2H`** are pipelined across the different streams. The system uses multiple CUDA streams (NUM_STREAMS, configurable at compile time) to execute independent batches in parallel.
A stream in CUDA is a sequence of operations (memcpy, kernel launch, etc.) that executes in order on the GPU, but different streams can run concurrently. By associating different partitions of a batch with a separate stream, the system enables overlapping data transfer and computation. For example, while a partition of a batch is being hashed, the next one can be transferred, and the previous one's results can be copied back — all simultaneously.
This overlap effectively hides memory latency and keeps the GPU fully occupied. Empirical benchmarks showed significant speedups when streams and asynchronous memory transfers were used compared to a naive sequential implementation.
All the execution carried out have been done with a number of streams equal to ***8***. This number is shown to be the value which provides the best performance with the current configuration.

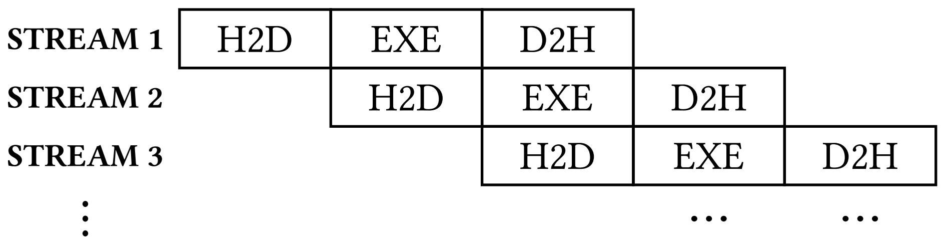

Figure 3: GPU streams execution flow

Instead of using standard **`cudaMemcpy`**, the implementation uses **`cudaMemcpyAsync`** to perform memory transfers without blocking the host thread. These transfers operate on page-locked (pinned) memory, which avoids costly intermediate copies made by the CUDA driver and enables Direct Memory Access (DMA) between host and device. Pinned memory is allocated once at program start to avoid fragmentation and reused across multiple batches (in case batch reading is used).
Cuda streams are created only once at startup and then are destroyed at the end of the execution of the algorithm. Two helper functions was used in order to handle Cuda streams. In Listing 5 the code is shown.

```
static void cuda_streams_create(cudaStream_t *streams)
{
    for (size_t i = 0; i < NUM_STREAMS; i++)
    {
        CHECK_CUDA_ERROR(cudaStreamCreate(&streams[i]));
    }
}

static void cuda_streams_destroy(cudaStream_t *streams)
{
    for (size_t i = 0; i < NUM_STREAMS; i++)
    {
        CHECK_CUDA_ERROR(cudaStreamSynchronize(streams[i]));
        CHECK_CUDA_ERROR(cudaStreamDestroy(streams[i]));
    }
}
```
Listing 5: Helper functions to handle streams creation and destruction

If streams execution is being used, the code inside the wrapper function of kernel (Listing 4), slightly changes. Listing 6 shows the code of the wrapper function for SHA1 parallel algorithm. If Listing 4 and Listing 6 are compared, it can be seen that the structure of the code is the same, the only difference lies in the fact that instead of using `cudaMemcpy` standard function, `cudaMemcpyAsync` is used in place of it, in order to be able to exploit streams execution and in order to interleave memory and compute pipelines and speedup the computation.

```
    CHECK_CUDA_ERROR(cudaMemcpyAsync(
                                d_data + off_data,
                                h_data + off_data,
                                bytes,
                                cudaMemcpyHostToDevice,
                                streams[s]));
    CHECK_CUDA_ERROR(cudaMemcpyAsync(
                                d_lengths + off_len,
                                lengths + off_len,
                                lines * sizeof(size_t),
                                cudaMemcpyHostToDevice,
                                streams[s]));

    size_t blocks = (lines + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK;

    sha1_kernel<<<blocks, THREADS_PER_BLOCK, 0, streams[s]>>>(
                                d_data + off_data,
                                d_hashes + off_len * SHA1_HASH_LENGTH,
                                d_lengths + off_len,
                                lines);
    CHECK_CUDA_ERROR(cudaGetLastError());

    CHECK_CUDA_ERROR(cudaMemcpyAsync(
                                hashes + off_len * SHA1_HASH_LENGTH,
                                d_hashes + off_len * SHA1_HASH_LENGTH,
                                lines * SHA1_HASH_LENGTH * sizeof(hashes[0]),
                                cudaMemcpyDeviceToHost,
                                streams[s]));
```

Listing 6: Streams execution SHA1 wrapper function code

Please note that in order to use streams execution, the memory allocated on the host side cannot be allocated using standard memory allocation utilities functions like `malloc` but some specific functions are to be used. Listing 7 shows the code of the utility function used to allocate non-paged (or pinned) memory on the Host side. Allocating memory in this way, it is guaranteed that the allocated memory remains in physical RAM without being swapped to disk. This allows for faster data transfers between the host and the device because the GPU can directly access the memory without the need for the operating system to intervene. If the memory was allocated in standard way (e.g. using `malloc`), then the streams execution would have no effect as no speedup is asynchronous memory copying could be exploited.

```
void allocate_cuda_host_memory(void **ptr, size_t alloc_size)
{
    CHECK_CUDA_ERROR(cudaHostAlloc(ptr,
                                   alloc_size,
                                   cudaHostAllocDefault));
    CHECK_NULL(ptr);
}
```

Listing 7: Utility function to allocate non-paged memory on Host side

2.6. Thread and Block count

The number of CUDA threads per block (`THREADS_PER_BLOCK`) can be configured statically using the `config.h` file, while the number of blocks per grid is computed dynamically based on the batch size. A typical configuration involves blocks of 64 threads and a grid size that ensures all data items in a batch are assigned to a thread. This ensures full utilization of GPU cores and enables parallel computation of thousands of hashes at once. However

2.7. CPU implementation

The CPU implementation of the hashing algorithms and the GPU code structure are mirrored to be as similar as possible, ensuring a fair baseline. This choice is lead by the will of having a baseline as similar as possible, in order to give still more meaning to the comparison between CPU and GPU algorithms execution.

3. Use cases and Applications

This design allows the system to be deployed in various scenarios:

1. **Brute-force attacks and password cracking**: where billions of candidate inputs must be hashed and compared against a target hash — a task that benefits significantly from the massive parallelism of the GPU. Large keyspaces can be explored quickly by distributing candidate passwords across threads.
2. **Collision detection and cryptanalysis**: large-scale experiments to find or study hash collisions become feasible, as the system can compute and compare vast numbers of hash values in parallel.
3. **Data integrity verification**: hashes of files or blocks can be computed in parallel for backups, distributed storage systems, or forensic analysis.
4. **Benchmarking and profiling**: researchers can study how different hash algorithms perform under parallel conditions and evaluate their resilience under high-load scenarios.

4. Comparisons and final results

This section presents the performance of CUDA kernels under varying parameters and optimization, compared with their CPU counterparts.
The results are presented using standard metrics such as **execution time (s)**, **number of operations (NOPS)**, **memory bandwith (GB/s)**.

4.1. Testbench hardware

Benchmarks are performed on a setup composed by:

- CPU: AMD Ryzen 7 3700X
- RAM: Corsair Vengeance DDR4 3200MHz
- GPU: NVIDIA GeForce GTX 1650
  - Compute capability: 7.5
  - Total global memory: 3.62 GiB
  - Warp size: 32
  - Number of SM: 14
  - Compute cores: 896
  - Clock rate: 1.78 GHz
  - Memory clock rate: 4.00 GHz
  - Memory bus width: 128 bits

4.2. Experiments

4.2.1. Thread Block configuration analysis

Optimizing the block size in CUDA applications is critical to achieve peak performance on the GPU. In this case, very small block sizes (e.g. 2, 4 or 8 threads per block) results in extremely poor performance due to low occupancy and the inability to effectively overlap computation with memory operations. As the block size increases to 64 threads per block (as shown in Figure 4), the GPU can schedule more threads concurrently, leading to better utilization of the available cores and a significant reduction in execution time. However, further increasing the block size to 128 threads per block or more does not yield additional improvements. Infact, it could sometimes introduce performance penalties likely due to resource contention, such as limited register. These observations underscore the importance of tuning the block size to balance between maximizing parallel execution and minimizing resource bottlenecks, ultimately ensuring that the GPU's computational capabilities are fully leveraged.
Figure 4 shows results based on SHA1 parallel execution on GPU using a dataset of about 17 900 000 items.

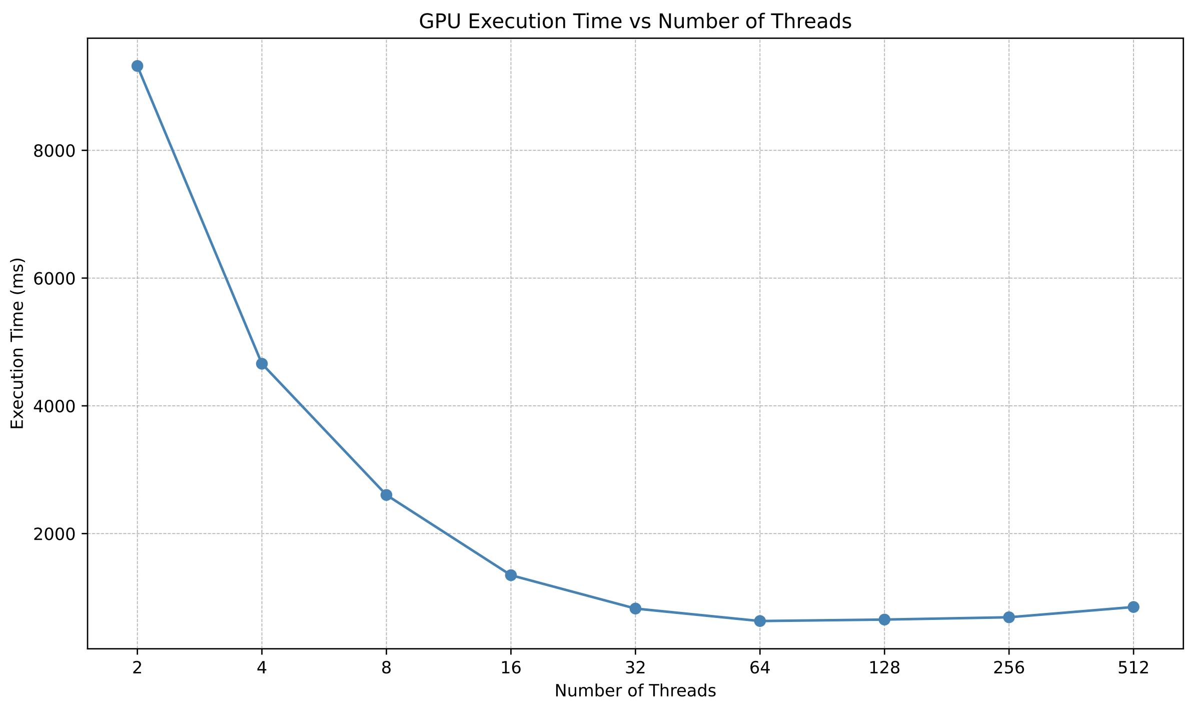

Figure 4: Thread Block size vs Execution time

4.2.2. Stream count configuration analysis

Optimizing the number of CUDA streams in GPU applications is essential to maximize concurrency and achieve the best execution time. In this dataset, running with a single stream (1) results in the highest execution time (around 700+ ms), reflecting limited concurrency and underutilization of the GPU's ability to overlap computation with memory operations. Increasing the number of streams to 2 or 4 significantly reduces the execution time (down to approximately 450–490 ms), as the GPU can schedule multiple concurrent operations, improving resource utilization and hiding latency. However, increasing the number of streams beyond 8 to 16 does not continue to improve performance; times plateau around the mid-450 ms to 480 ms range. This indicates that while multiple streams enable concurrent execution, after a certain threshold, additional streams introduce overhead such as scheduling complexity and resource contention, which limits further gains. For higher stream counts such as 64, 128, and 256, the execution time remains relatively stable with some variability. Occasionally, times increase slightly, likely due to factors such as increased context switching or memory bandwidth contention. These results highlight the importance of tuning the number of streams to balance concurrency and overhead. Using too few streams underutilizes the GPU, while too many streams can saturate internal resources, negating the benefits of parallelism. In this case, the optimal range appears to be around 4 to 16 streams, which effectively leverages the GPU's capabilities without incurring significant overhead. Figure 5 shows these results based on SHA-1 parallel execution on a GPU using a dataset of approximately 17,900,000 items.

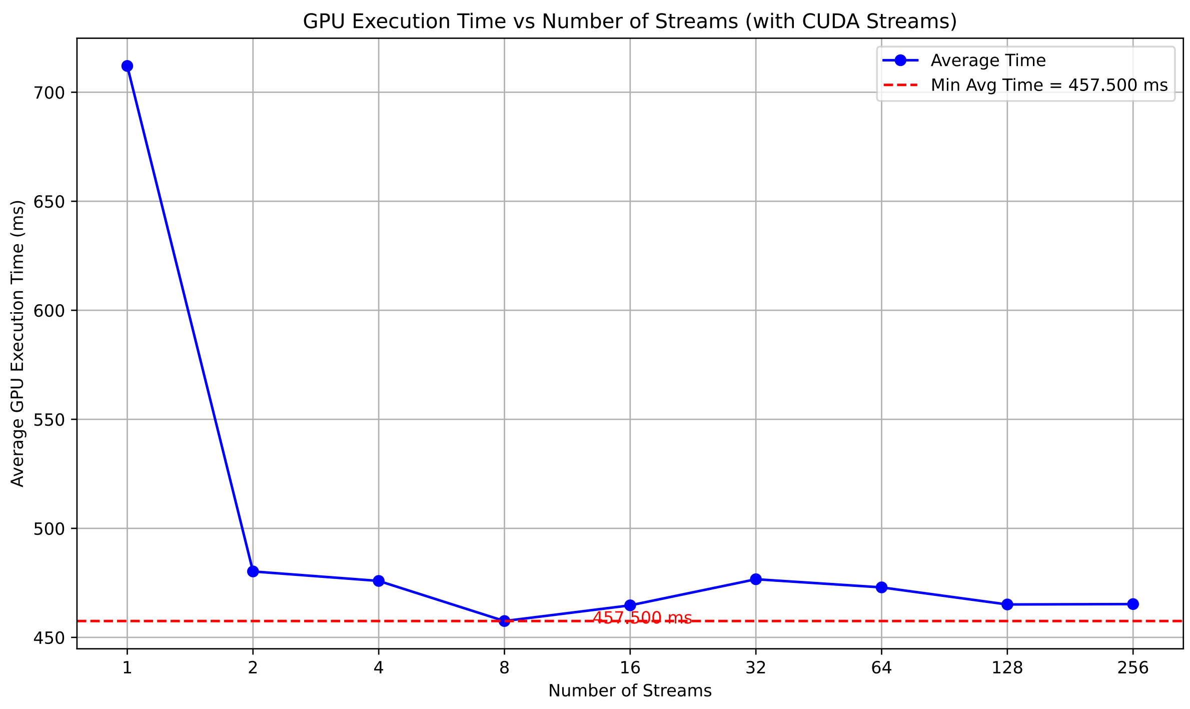

Figure 5: CUDA stream count vs Execution time

4.2.3. Roofline Analysis

The Roofline Analysis in Figure 6, illustrates the relationship between Performance (OPs/s) and Arithmetic Intensity (OPs/byte). The different dots represent the kernels' performance. All the dots are positioned below the theoretical GPU limit, indicating that the kernel is not fully utilizing the maximum throughput. The kernel appears to be **compute-bound**. However this is only an absolute model: it doesn't care how busy SMs are, only whether your kernel is being capped by compute or bandwidth theoretically, assuming ideal utilization. Since **Arithmetic Intensity** of kernels is above machine balance ($\frac{\textbf{MAX PERFORMANCE}}{\textbf{MAX BANDWIDTH}}$), which in this case is equal to $\frac{1330 \text{ GOPS}}{128 \text{ GB/s}} \approx 10.39$, this highlights the fact that kernels are compute-bound. The plot also shows very well that the kernels which exploit the streams execution, can reach an overall better performance, pushing the kernels nearer to the computation bound in the diagram. This is due to the fact that exploiting streams execution, thus interleaving memory and computing pipelines, a better throughput of the computing pipeline can be achieved.
All the results shown in Figure 6 come from CUDA kernels issued with blocks of 64 threads, since it is experimentally shown that in this case this is the number of threads which allows for the best performance.

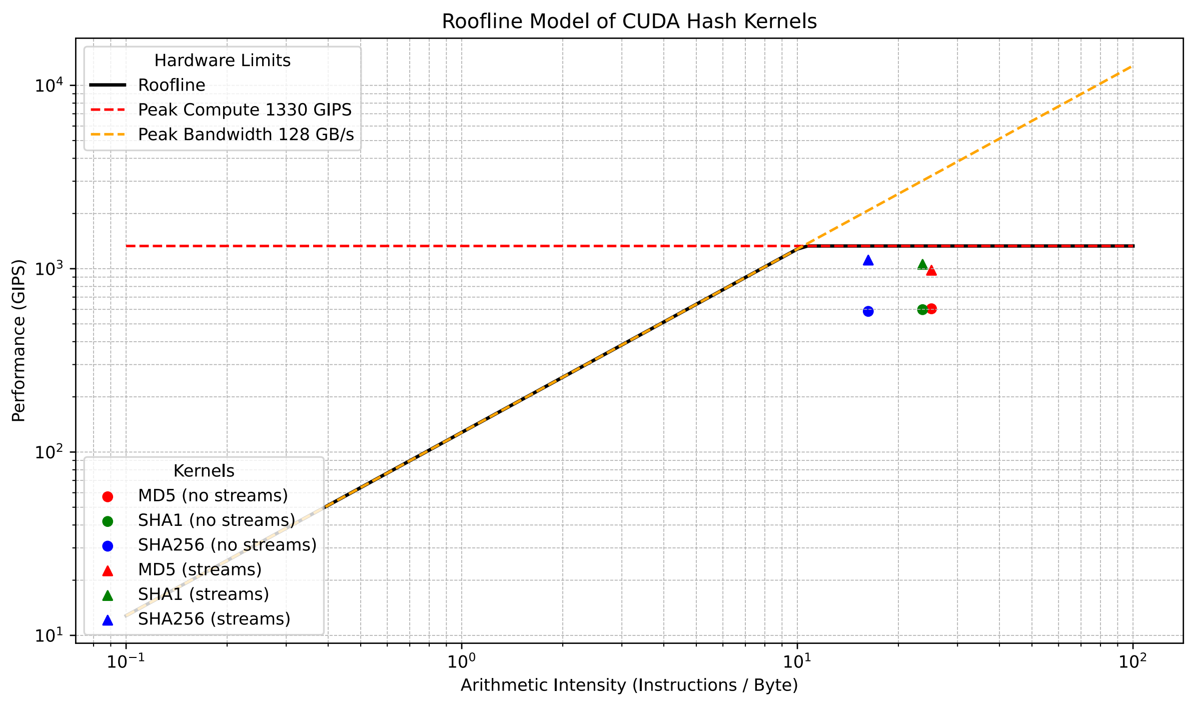

Figure 6: Roofline Diagram

4.2.4. Profiling with NVIDIA Nsight Compute

Recalling to the fact that Roofline model is only an absolute model, after profiling the execution using NVIDIA Nsight Compute, it still turns out that all the kernels are mainly compute bound. Profiling results are shown in Table 1 (please note that only non streams execution results are shown) This is in part due to the fact that the dataset is composed of small strings, but mostly due to the fact that hashing algorithms are computing intensive, therefore a lot of operations are performed to obtained the hash value. Also, in case of longer strings, the problem would still exist, since these hashing algorithms perform the same set of operations on blocks of 512 bytes and therefore the best way to maximize the memory vs computation throughput woudld be use string of 512 bytes.
This suggests that:
1. The compute units (SMs) are busier relative to the memory units.
2. The kernel is spending more time doing computation than waiting for memory.

Kernel	Memory throughput	Compute throughput
SHA1	28.17%	63.24%
MD5	27.51%	62.55%
SHA256	45.34%	70.48%

Table 1: Compute and Memory throughput of CUDA kernels profiled using NVIDIA Nsight Compute

4.2.5. GPU vs CPU performance

The main purpose of this project is to show the improving that can be obtained exploiting massively parallel capability of GPUs over CPUs.

In Table 2, execution time of different configurations running either on CPU or GPU are shown.
The picture, perfectly highlights and shows the speedup that can be obtained using GPUs.
As you might expect, with a very small dataset (first row of Table 2), the overhead of GPUs if fully dominating the execution time, therefore it goes naturally that CPU is faster overall. However it is interesting to note that using streams execution, GPU is able to drop a lot of overhead and the total execution time is very scaled compared to non streams execution.
Scrolling down the rows of the table is easy to notice that with the increasing of the dataset size, the performance of GPU with respect to the CPU are notably better, this is due to the first concept of GPU programming, i.e. GPUs beats CPUs not on single core performance but on multicore performance and on throughput. Therefore, the hash of a single string will *always* take more time on a GPU with respect to a CPU, but, whilst on a consumer CPU you can have at most tens of cores, on consumer GPU you can have hundreds of them and if filled correctly, the performance improving is remarkable.

In Table 2 only the results of SHA1 algorithms are shown, despite other algorithms with the same structure, like MD5 and SHA256 perform in a similary way.
It also worth notice that the code running, using batch processing, there is an overhead in both the executions, this is obviously due to the fact that dataset is read in fixed-size batches and the whole dataset is not fitted entirely on VRAM (GPU DRAM).

The implementation then leads to a maximum speedup of about **7 or 8 times** using standard CUDA code, while using streams execution it leads to a speedup of about **10 or 11 times**. If using batch processing the speedup is limited to **4 or 5 times** depending on whether streams execution is used or not. This is due to the fact that in batch processing part of the execution is done on the CPU (dataset reading).

Dataset lines	**CPU**	**CPU BP**	**GPU**	**GPU BP**	**GPU streams**	**GPU BP streams**
79	0.04 ms	0.09 ms	165.1 ms	226.9 ms	2.53 ms	24.3 ms
3.8×10^5	206.12 ms	237.4 ms	232.9 ms	313.5 ms	21.23 ms	70 ms
18×10^6	9467.3 ms	10912 ms	1202.7 ms	2572.7 ms	883.5 ms	2148 ms
64×10^6	✗	39186 ms	✗	8953 ms	✗	7995 ms

Table 2: Comparison of SHA1 execution time between CPU and GPU