

Politecnico di Torino

Politecnico di Torino

Master's Degree in Software Engineering

Academic Year 2025/2026

Graduation Session October 2026

Accelerated Triangle Counting for Massive Graphs On GPUs

Supervisors:
Stefano Quer
Lorenzo Cardone

Candidates:
Nicola Basciu s349485

Abstract

Triangle counting is a fundamental primitive in graph analysis, serving as a building block for higher-level metrics like the local clustering coefficient. It is fundamental in different scenarios, including social network community detection, biological protein-protein interaction, fraud detection, and cybersecurity for threat and anomaly identification.

GPUs offer massive parallelism and high memory bandwidth, representing a suitable platform for these types of algorithms, but they introduce several challenges. First, the CSR format commonly used to store sparse graphs leads to irregular, non-coalesced memory accesses when performing neighbor-list intersections, severely limiting achievable memory bandwidth. Second, real-world graphs typically exhibit highly skewed degree distributions, causing significant load imbalance across GPU threads and warps. Third, approaches based on full bitmaps to accelerate intersection tests, incur prohibitive memory overhead on large graphs, often exceeding the limited device memory available on GPUs.

This thesis introduces GetterTri, a novel GPU-based triangle counting algorithm that addresses these limitations by integrating merge-based and binary-search intersection techniques with a run-time load balancing strategy. Rather than committing to a single intersection strategy for the entire graph, GetterTri leverages CUDA Cooperative Groups to implement a dynamic intra-warp load-balancing strategy that adaptively selects between merge-based and binary-search techniques according to runtime cost estimates derived from the relative sizes of the neighbor lists being intersected.

This thesis also presents a comparative analysis of Enterprise and Consumer GPUs, highlighting the performance differences between the two types of devices.

Table of Contents

1 Introduction 1

2 Theoretical Background 4
- 2.1 Graph Theory . . . 4
  - 2.1.1 Edge Properties . . . 4
  - 2.1.2 Graph Density and Sparse Graph . . . 5
  - 2.1.3 Graph's Application in the Real World . . . 6
  - 2.1.4 Data Representation . . . 6
- 2.2 Triangle Representation . . . 8
- 2.3 Approaches to Triangle Counting . . . 9
  - 2.3.1 Algebraic Approach . . . 9
  - 2.3.2 Intersection-based Approach . . . 10

3 GPU Architecture 12
- 3.1 Graphics Processing Unit (GPU) Overview . . . 12
- 3.2 GPGPU Programming Model and CUDA . . . 14
- 3.3 Memory Hierarchy . . . 16
- 3.4 Profiling Tools . . . 17
  - 3.4.1 Nvidia Nsight Systems . . . 17
  - 3.4.2 Nvidia Nsight Compute . . . 18
- 3.5 Enterprise and Consumer Differences . . . 22
  - 3.5.1 Memory . . . 22
  - 3.5.2 Hardware Accelerators . . . 23

4 Traditional Approaches to Triangle Counting 25
- 4.1 Matrix Multiplication Approach . . . 25
- 4.2 Intersection-based Approach . . . 28
  - 4.2.1 Merge Path . . . 28
  - 4.2.2 Binary Search . . . 29
  - 4.2.3 Hash Buckets . . . 30
- 4.3 State of the Art . . . 31

5 GetterTri 35
5.1 Edge or Node Iterator . . . 35
5.2 Preprocessing . . . 36
5.3 Workload balancing and Memory Optimization . . . 41
5.4 Combination of Traditional Approaches . . . 42
5.5 Cuda Cooperative Groups and Warp Primitives . . . 43
5.6 Threshold Analysis . . . 45
5.7 GetterTri Implementation . . . 46

6 Results and Discussion 53
6.1 Experimental Environment . . . 53
6.2 CPU vs GPU Performance . . . 54
6.3 Profiling Results . . . 57
6.4 Comparison with State of the Art . . . 59
6.4.1 Performance Comparison . . . 61
6.5 Enterprise vs Consumer Hardware Results . . . 62

7 Conclusion and Future Work 67
7.1 Conclusion . . . 67
7.2 Future Work . . . 68

List of Figures 69

List of Tables 72

Bibliography 74

Chapter 1

Introduction

Beyond simple data structures, graphs function as a sophisticated mathematical language that distills complex real-world information and intricate connectivity into machine-readable models. Within this architectural paradigm, vertices serve as the fundamental data entities, while edges represent their multidimensional relationships as connective tissue. This versatile abstraction allows researchers to map nearly any system of interactions, leading to a broad spectrum of applications across diverse scientific and technical frontiers. In molecular chemistry and bioinformatics [1, 2, 3, 4], graphs are indispensable for modeling atomic structures and analyzing metabolic pathways. They are equally transformative in the digital world, where they power authoritative web search ranking algorithms [5] and map the "small world" connections within social networks [6]. Furthermore, the predictive power of graph analysis is utilized in cybersecurity to cluster malware behaviors [7], in hardware design to simulate large digital circuits [8], and in software engineering to track subsystem failures and component dependencies [9, 10].

Triangle Counting in graphs is an important primitive, used in different fields, from community detection in social networks to protein-protein interaction in biology. It is also important because it is fundamental in order to calculate other metrics, like the clustering coefficient, which measures the local density around a node, based on how much a node is connected to others. Beyond these fundamental measures, triangle counting serves as a key building block in numerous graph analytics tasks, like anomaly detection and network motif analysis. Its applications span a wide range of domains.

In social network, triangles let us detect communities that are highly interconnected.

In biological networks, graphs are used to represent interactions between cellular components, like proteins, genes, and so on, analyzing the complexity of the resulting graph. In this case, triangles are used to represent some of the Protein-Protein Interactions (PPIs) and Gene-Regulatory Interactions (GRIs), differentiating each

type of triangle, or, more generally each possible motif, by the direction of the edge [11].

In the field of cybersecurity, the use of graphs to predict an anomaly is known as Graph-Based Anomaly Detection (GBAD) [12], and is used to identify suspicious behaviour, botnets or coordinated attacks.

Over time, researchers have developed different techniques in order to find and count triangles, all of these techniques can be divided into two main groups. The first one includes those techniques based on linear algebra. While linear algebra approaches are elegant, they are often inefficient in practice for large-scale graphs and prohibitive due to the high use of memory. This is primarily because they compute the cube of the adjacency matrix, compute its diagonal, and sum its elements, and in order to do so, it is necessary to store the complete matrices. On GPU architectures, this approach is frequently prohibitive due to the device's limited memory capacity. The second strategy exploits an intersection list. Instead of calculating complex matrices, these types of algorithms search through the adjacency list of two connected nodes to find common nodes, if any. In GPU implementations, these types of algorithms assign a vertex or an intersection to a single warp, which is very efficient when the involved node is a hub, but wastes precious GPU resources in the case of a sparse graph without hub nodes. As pioneered by Schank et al. [13], these algorithms are better suited to the sparsity of real world graphs, due to their low density, in many cases less than 1%, but also bring some problems, like data locality, which results in inefficient bandwidth utilization. In order to improve data locality, expensive preprocessing techniques are used, leading to improved kernel execution time at the cost of a non-negligible preprocessing overhead that is often excluded from final results.

This thesis investigates and analyzes various intersection-based approaches to triangle counting, providing a rigorous comparison with established state-of-the-art frameworks. To ensure a comprehensive evaluation, we examine both CPU and GPU algorithms and analyze critical performance metrics, including kernel execution time and memory utilization. We introduce **GetterTri**, a novel algorithm that addresses the triangle-counting problem by integrating a hybrid search strategy that combines Merge-based and binary-search techniques. To mitigate the challenges posed by power-law degree distributions, GetterTri intra-warp communication mechanism allows dynamic workload redistribution within a warp when a thread encounters excessive computational load. The main contributions of this thesis are:

- A hybrid intersection strategy that combines Merge-based and Binary-Search, dynamically choosing the best approach based on the length of the two intersection lists
- A load-balancing mechanism that exploits intra-warp communication, estimates the workload of each intersection at runtime, and adaptively switches

between independent and cooperative execution

- A lightweight preprocessing strategy that filters edges directly during the file reading phase, avoiding the need to store both the edges in memory, achieving a low memory footprint

GetterTri was evaluated on a diverse dataset comprising 91 graphs, achieving an average speedup of 1.77× over TRUST[14], 3.31× over TRICORE[15],1.96× over H-Index[16],8.02× over GUNROCK[17], and over 31.40× over GALOIS[18] on Enterprise Hardware.

Chapter 2

Theoretical Background

2.1 Graph Theory

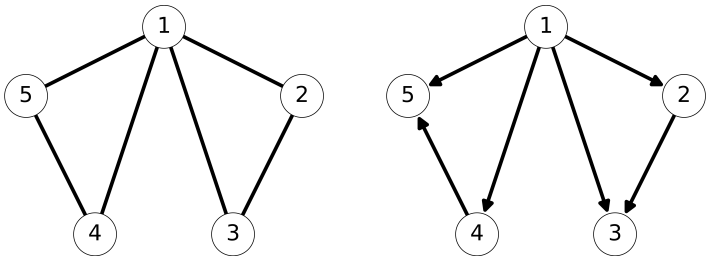
A graph $G = (V, E)$ is a set of vertices V connected through the edges in E. We indicate the number of elements in V with $|V|$, and the number of edges in E with $|E|$.

2.1.1 Edge Properties

For a vertex $v \in V$, we define its neighborhood $N(v) = \{u \in V \mid (u, v) \in E\}$ as the set of vertices adjacent to v, and its degree $deg(v) = |N(v)|$ as the cardinality of this set.

A graph is unweighted when edges do not have a label to indicate their weight. When a graph is undirected, each edge $e \in E$ connecting two vertices $v, w \in V$ is an unordered relationship, i.e., it is identical to an edge $f \in E$ connecting the same two vertices in the opposite direction $w, v \in V$. Our approach does not consider multiple edges connecting the same two vertices, as it automatically collapses them into a single edge.

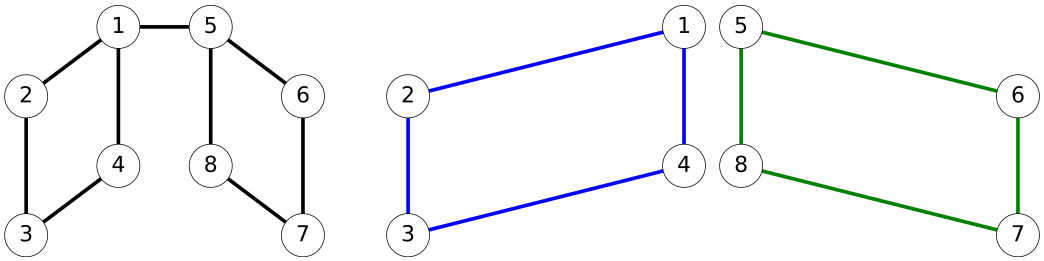
In the context of unweighted and undirected graphs, a triangle is a set of three distinct vertices that are mutually connected. In some cases, as shown in Figure 2.1, the graph is manipulated in order to optimize the structure, removing the mutual connection between two vertices and maintaining only one edge, depending on the rule applied, i.e., based on the degree of the vertices, only the edge from u to v is maintained if the degree of u is greater than the degree of v. In this way, it is possible to process and analyze the graph properties more easily, due to the smaller number of edges to analyze.



(a) *Graph* **(b)** *DAG*

Figure 2.1: Transformation from an undirected graph (a) to a DAG (b). In this case, the rule applied is based on the vertex ID, more precisely $id(u) < id(v)$.

Another widely adopted strategy consists of partitioning the primary graph into multiple subgraphs by identifying and removing a minimal set of connections, known as edge cuts. This decomposition simplifies graph analysis by reducing the number of vertices and edges in the working set for each subgraph. Furthermore, graph partitioning enables hybrid and adaptive processing models: individual subgraphs can be processed independently, allowing specific optimizations to be applied based on the unique topological properties of each partition.



(a) *Original Graph* **(b)** *Sub Graph* 1 **(c)** *Sub Graph* 2

Figure 2.2: Partitioning of the original graph(a) into two different graphs, cutting the edge from 1 to 5.

2.1.2 Graph Density and Sparse Graph

The density of a graph is defined as the ratio between the number of edges actually present and the maximum possible number of edges between $|V|$ vertices, i.e., $\frac{2|E|}{|V|(|V|-1)}$ for an undirected graph. Real-world graphs are characterized by poor density, a property that has profound implications for both the choice of data structures used to store the graph and the design of algorithms. Sparsity is particularly relevant for triangle counting, since the cost of intersecting neighbor lists is directly tied to their cardinality, and the presence of **hub** nodes, which are

a particular type of node that possesses a high number of edges with respect to other nodes, leads to a possible load imbalance in parallel implementations.

Instead, when each vertex is connected to each other, we have a Full Mesh graph, as visible in Figure 2.3.

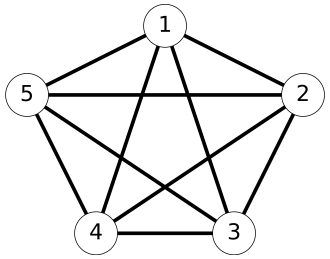


Figure 2.3: Example of Full Mesh graph

2.1.3 Graph's Application in the Real World

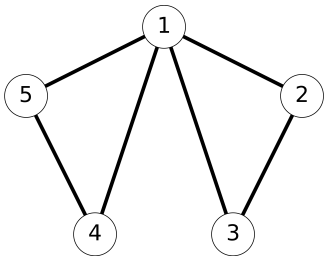
One powerful feature of graphs is the ability to represent a collection of entities in the form of vertices, with edges that serve as connections between them. This type of representation is common across various domains, such as social networks, protein structures, and the World Wide Web. A wide variety of analyses can be performed using graphs, ranging from fraud detection to protein-protein interaction networks.

Over time, the scale of real-world graphs has grown exponentially, evolving into massive datasets comprising billions of vertices and trillions of edges. Consequently, analyzing these large-scale graphs implies severe computational and architectural challenges. A wrong choice of the data structure used, combined with a large graph, is likely to lead to a high memory footprint, causing the main memory capacity of standard hardware systems to be exceeded.

As a result, traditional analytical approaches become inefficient or computationally prohibitive. To address these bottlenecks, modern research focuses on designing scalable graph processing frameworks, optimized data structures, and high-performance parallel algorithms capable of processing massive graphs efficiently.

2.1.4 Data Representation

The classical way to represent a graph is to build an adjacency matrix, where each column represents the presence or absence of a connection between two nodes (Figure 2.4), resulting in a bitmap. This type of data structure is very inefficient, due to the storage of non-existing connections.



(a) *Graph*

	1	2	3	4	5
1	0	1	1	1	1
2	1	0	1	0	0
3	1	1	0	0	0
4	1	0	0	0	1
5	1	0	0	1	0

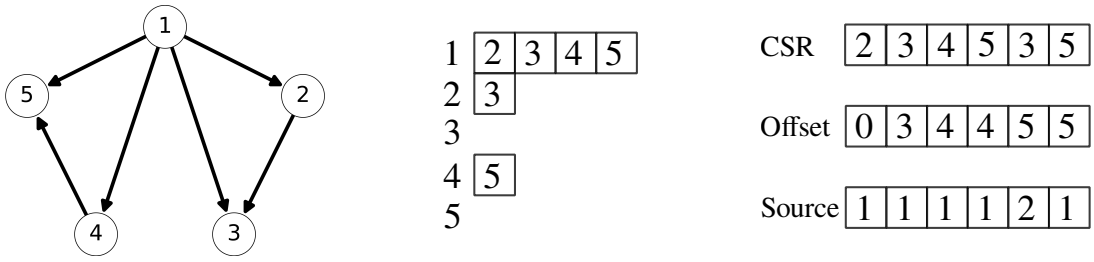
(b) *Adjacency Matrix*

Figure 2.4: Representation of a graph with an adjacency matrix.

In order to solve this issue, another type of data structure is used, the Compressed Sparse Row (CSR), where for each vertex, only the existing connections between two nodes are stored, allowing to save precious memory. For example, take a real world graph with 3M vertices and 45M edges, the space used for storing the graph is, supposing that an edge occupies 1 byte in the bitmap and 4 bytes in the CSR:

- Bitmaps $= |V|^2 \cdot 1Bytes = 3M \cdot 3M \cdot 1Bytes = 8.18TB$
- CSR $= |E| \cdot 2 \cdot 4Bytes = 2 \cdot 45M \cdot 4B = 343.32MB$

Furthermore, if we store only one edge for each pair of vertices, as previously discussed in section 2.1.1, the space needed to store the graph is halved, equal to only $\approx 180MB$. In order to use CSR, two other data structures are needed: one for storing the offset, and one for keeping track of the source node. Even if the last is not strictly needed, since it is possible to calculate the source node on the fly from the offset, it is helpful because it let us avoid calculating the value each time. With these new data structures, the final space occupied is equal to $(|E| \cdot 2 + |V| + 1 + |E|) \cdot 4B$ for undirected graphs is $\approx 552MB$, and $(|E| + |V| + 1 + |E|) \cdot 4B$ for oriented graphs, so for the previous example the total space needed is $\approx 355MB$. An example of CSR is shown in Figure2.5

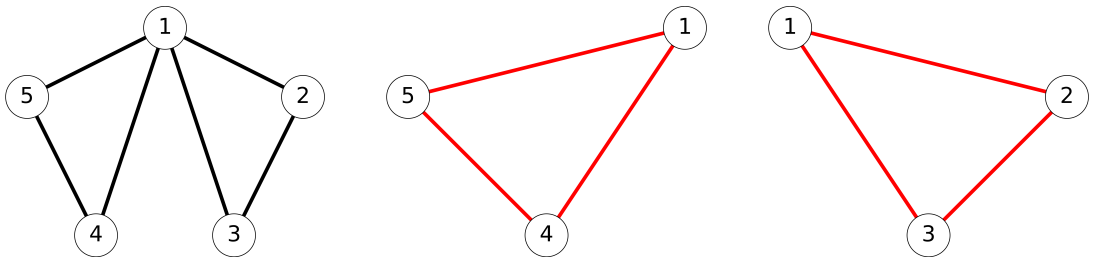


(a) Graph **(b)** CSR arrays **(c)** Complete CSR

Figure 2.5: Graph representation using the Compressed Sparse Row format

2.2 Triangle Representation

Formally, a triangle exists if for any two nodes $u, v \in V$ that are connected by an edge $e \in E$, there is a third node $w \in V$ such that w is also connected to both u and v. This structure forms a complete subgraph of three nodes, often referred to as a 3-clique (K_3). A triangle is a set of three distinct vertices that are all mutually adjacent. Figure 2.6 shows some graphical examples with some triangles.



(a) G_1 **(b)** G_2 **(c)** G_3

Figure 2.6: Given the graphs G represented in Figures (a), Figures (b), and (c) report two different admissible triangles.

A triangle constitutes the smallest motif within a network topology.

Metrics such as the clustering coefficient rely directly on triangle counts to quantify the degree to which nodes tend to group together.

Consequently, efficiently identifying and enumerating triangles is indispensable for advanced computational pipeline tasks, including community detection, spam detection, and fraud identification.

However, real-world graphs introduce significant challenges. From a computational perspective, the baseline approach for triangle counting involves computing the intersection of neighboring sets between connected pairs of vertices. While the complexity of this process is heavily influenced by the degree distribution

of the network, modern real-world networks typically exhibit power-law degree distributions characterized by a small fraction of highly connected **hub** nodes.

Furthermore, these computational demands reveal other bottlenecks. The memory footprint required to store data structures, combined with irregular memory access patterns, leads to poor data locality. In modern memory hierarchies, this irregularity manifests as frequent cache misses and memory bandwidth saturation, rendering traditional sequential and naive parallel implementations entirely inefficient for graphs with billions of edges.

Modern solutions combine sophisticated algorithmic optimizations, such as vertex re-ordering, dynamic orientation, exploiting CPU multi-threading, GPU parallelism, and heterogeneous memory systems[19].

2.3 Approaches to Triangle Counting

Over the years, different approaches for counting triangles in a graph have been developed. All these algorithms can be grouped under two main categories, the **Algebraic** approach and the **Intersection-based** approach.

2.3.1 Algebraic Approach

The algebraic approach is a very elegant way to deal with this problem, but in general, it has been shown to be very ineffective, mainly for two reasons: the huge amount of memory needed to store the matrix, and the extra time required to transform a sparse graph into a bitmap.

Given the adjacency matrix A of an undirected graph G, the total number of triangles can be computed using the classical trace formula: $TC = \frac{\text{trace}(A^3)}{6}$, where $A^3 = A \cdot A \cdot A$ is obtained through standard matrix multiplication, and the trace sums the diagonal entries of the resulting matrix.The division by 6 accounts for the fact that each triangle is counted twice for each of its three vertices.

An equivalent, and often more practical formulation relies on the Hadamard element wise product rather than the trace of a triple matrix product: $TC = \frac{\sum_{i,j} A^2_{i,j} \cdot A_{i,j}}{6}$

Here, $A^2_{i,j}$ counts the number of common neighbors shared by vertices i and j, while the Hadamard product with A masks this quantity, retaining a contribution only for vertex pairs that are themselves connected by an edge. This formulation is generally preferred over the trace in practical implementations, since it avoids computing the full triple A^3 and instead requires only a single matrix multiplication $A^2 = A \cdot A$, followed by a masked, element-wise reduction against A.

2.3.2 Intersection-based Approach

The intersection-based approach, instead of facing the problem from a point of view of matrices, exploits the adjacency lists of the nodes. There are two main categories of intersection-based approach, these methodologies are usually referred to as the **Node Iterator** and the **Edge Iterator**. They differ primarily in their starting point and traversal strategy.

The Node Iterator method adopts a vertex-centric approach. Following Hu et al. [15], Algorithm 1 presents the pseudocode for this methodology.

Algorithm 1 Node Iterator

```
Require: G = (E, V)
  triangles = 0
  for v ∈ V do
      for u ∈ Adj(v) do
          triangles+ = CheckIntersection(v, u)
      end for
  end for
```

Algorithm 1 iterates through every vertex v in the graph's vertex set V. For each vertex v, the algorithm iterates over its adjacency list to identify its neighbors. For each neighbor u, it performs a common-intersection check to determine whether v and u share any other connections. This process essentially treats each node as a local hub, searching for closed paths of length three within its immediate neighborhood. In contrast, the Edge Iterator focuses on the graph's connections, iterating over every edge (v, u) in the set E rather than the vertices. Algorithm 2 reports the pseudocode of this strategy, following the work by Hu et al. [15].

Algorithm 2 Edge Iterator

```
Require: G = (E, V)
    triangles = 0
    for (u, v) ∈ E do
        triangles+ = CheckIntersection(u, v)
    end for
```

For every pair of connected nodes, Algorithm 2 intersects their respective adjacency lists to find a third node w that connects them. If such a node exists, a triangle is confirmed.

To optimize this process, researchers often use a refined version, the **Edge Iterator Forward** proposed by Schank et al. [13].In this variant, we first assign

a unique identifier to each node. We represent the graph by storing each edge exactly once, from the vertex with the lower identifier to the one with the higher identifier. This transformation effectively turns the graph into a Directed Acyclic Graph (DAG), preventing redundant computations by ensuring that each triangle is counted only once, thereby significantly enhancing the search efficiency on high-performance architectures such as GPUs.

Algorithm 3 Edge Iterator (forward procedure).

Require: $G = (E, V)$
  $triangles = 0$
  for $(u, v) \in E$ **do**
    if $(id(u) < id(v))$ **then**
      $triangles+ = CheckIntersection(u, v)$
    end if
  end for

Chapter 3

GPU Architecture

This chapter covers the history of GPUs, from their original purpose to their current applications, and then programming language features and profiling tools will be analyzed, before analyzing the hardware differences between enterprise and consumer models.

3.1 Graphics Processing Unit (GPU) Overview

GPUs were initially designed as dedicated hardware accelerators for real-time image processing, primarily targeting the consumer gaming market, enabling the parallel evolution of both hardware and increasingly complex products. Their primary function was to convert points in space into images; a graphical representation of the pipeline is shown in Figure 3.1 (source Graphics Compendium).

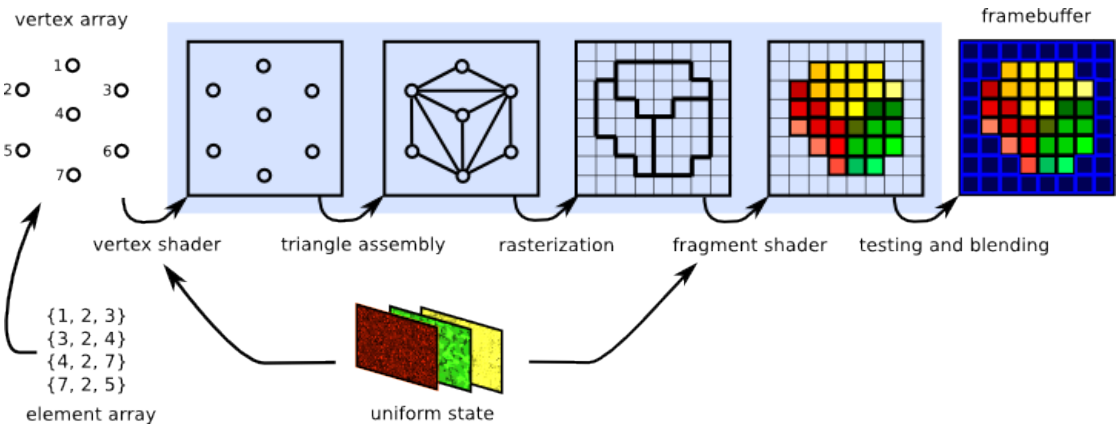


Figure 3.1: Visual representation of a complete rasterization pipeline

More generally, a GPU can be viewed as a collection of processors, much simpler

than a CPU, that simultaneously execute the same stream of instructions. In Figure 3.2 (source Cornell), two architectures are compared; on the left there is a typical CPU architecture, while on the right there is a GPU's. As is possible to see, the caches are shared among different cores, and the number of cores in a GPU is by far superior with respect to a CPU; e.g., an Nvidia RTX 5070 possesses a total of 6144 cores, whereas an AMD Ryzen 7 7800X3D has a total of 8 physical cores and 16 logical cores. Figure 3.3, demonstrates how a CPU and a GPU are connected.

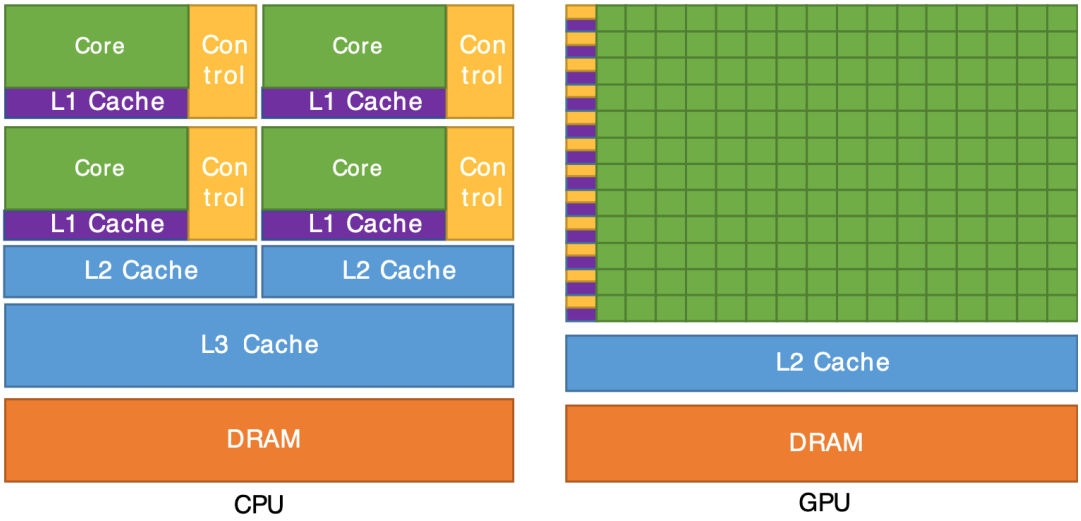


Figure 3.2: Comparison of CPU and GPU architectures

Due to their intrinsic nature for parallel computing, GPUs have evolved over time from processing only images to performing parallel computations. Various languages and standards exist for executing parallel code on GPUs, ranging from vendors such as CUDA for Nvidia and ROCm for AMD to universal options like OpenCL. This transition has enabled the porting of algorithms originally designed for CPU multithreading to GPUs. This thesis adopts CUDA as the reference language, as it is widely used in the literature due to its optimization capabilities.

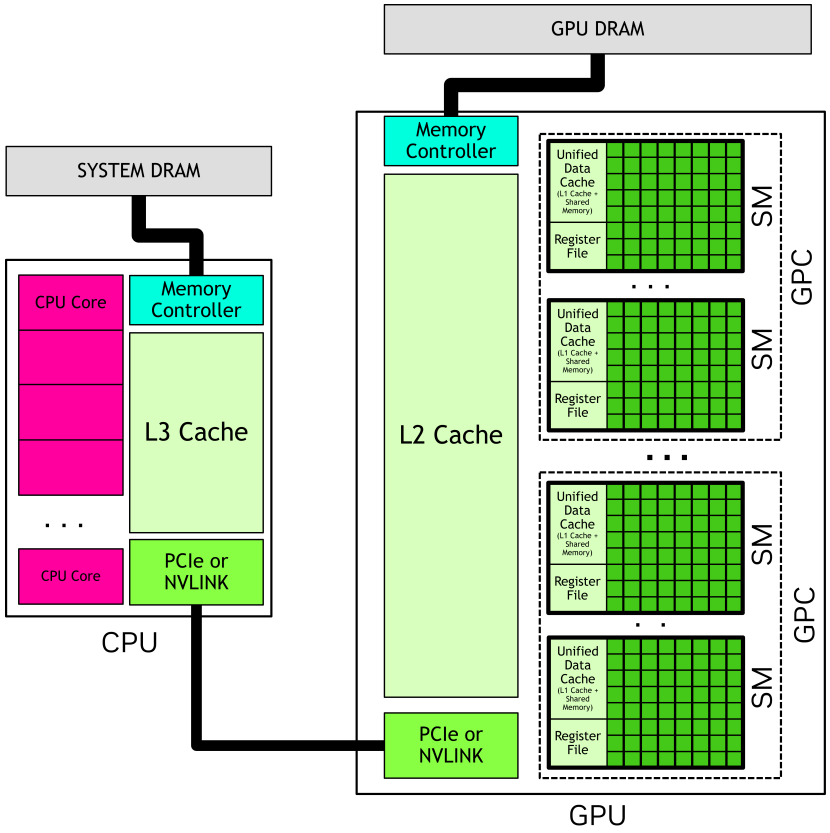


Figure 3.3: Interaction between CPU and GPU, source Nvidia

3.2 GPGPU Programming Model and CUDA

CUDA has become the standard due to its high performance and optimization with respect to other languages, but also due to its environment, having powerful profiling tools that help the programmer analyze and identify bottlenecks and other useful metrics. CUDA is presented as an extension of the C language, introducing the concept of a kernel, code that is run on the GPU device, an example of a kernel is shown in Figure 3.4. CUDA differentiates the CPU and the GPU by calling them Host and Device, respectively.

```
__global__ void myKernel()
{
    printf("Hello from GPU!\n");
}
```

Figure 3.4: Simple example kernel in CUDA.

In order to launch one or more kernels, the programmer must specify other parameters: the block and grid sizes. At first, threads are grouped into a block, whose size ranges from 1 to 1024, and then will be grouped another time inside a grid. The total number of threads executed is equal to $SizeBlocks \cdot SizeGrid$. Threads are identified by the combination of the id inside the block and the unique id of the resident block. In Figure 3.5, a visual representation of how threads are grouped into blocks and a grid is shown, while in Figure 3.6, a kernel is shown.

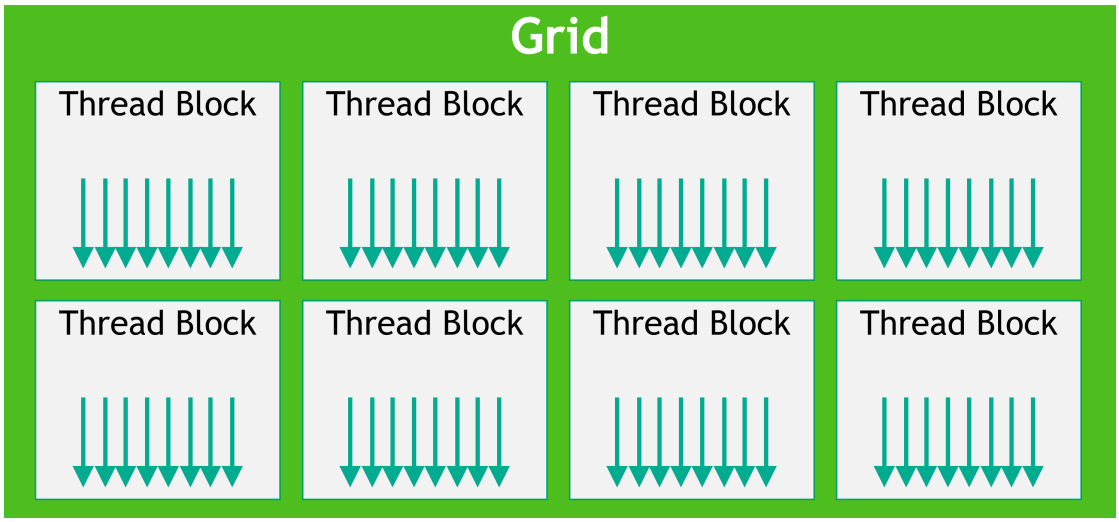


Figure 3.5: Visual representation of how grids and blocks work in CUDA (image source : Nvidia CUDA)

```
void launchMyKernel()
{
    dim3 blockDim(10);
    dim3 gridDim(2);
    myKernel<<<gridDim, blockDim>>>();
    cudaDeviceSynchronize();
}
```

Figure 3.6: Kernel Launch with a total of 20 threads.

Another fundamental concept in CUDA is warps and Streaming Multiprocessors (SM). A warp is a group of 32 threads within the same block that are executed at the same time in an SM, e.g., if the block size is equal to 128, the total number of warps is equal to 4. In the presence of branch divergences, due to an if statement like in Figure 3.7, the scheduler will apply a mask to sequentially execute threads that took different branch paths, degrading the overall performance.

```
__global__ void kernelDivergence()
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx % 2 == 0)
    {
        printf("Thread %d: Even index\n", idx);
    }
    else
    {
        printf("Thread %d: Odd index\n", idx);
    }
}
```

Figure 3.7: Example of kernel divergence

3.3 Memory Hierarchy

GPUs feature multiple levels of memory. This section explains how each level interacts with the system and within the device itself. Figure 3.8 serves as a reference throughout this section. There are four main types of memory: **Global Memory**, **Constant Memory**, **Shared Memory**, and **Registers**.

Global Memory Global memory is the main memory used by the device. It is the slowest, but offers the largest capacity. It allows the host to read from and write to the device.

Constant Memory This memory is faster than Global memory but more restrictive, as it is read-only. To maximize the efficiency of this memory type, as many threads as possible should read from the same memory location simultaneously. To declare a variable as Constant, the keyword **___constant___** must be used.

Shared Memory Shared Memory is a type of memory shared among threads within the same block. Threads in one block cannot access the shared memory of another block. It is faster than Constant memory. Allocating an excessive amount of shared memory can degrade performance, as a Streaming Multiprocessor will be able to host fewer concurrent blocks due to physical hardware limitations.

Registers Registers are private to each thread, they offer the fastest access speeds but have limited capacity. If a thread requires more registers than are available, **Register Spilling** occurs, forcing the device to use global memory, resulting in reduced performance.

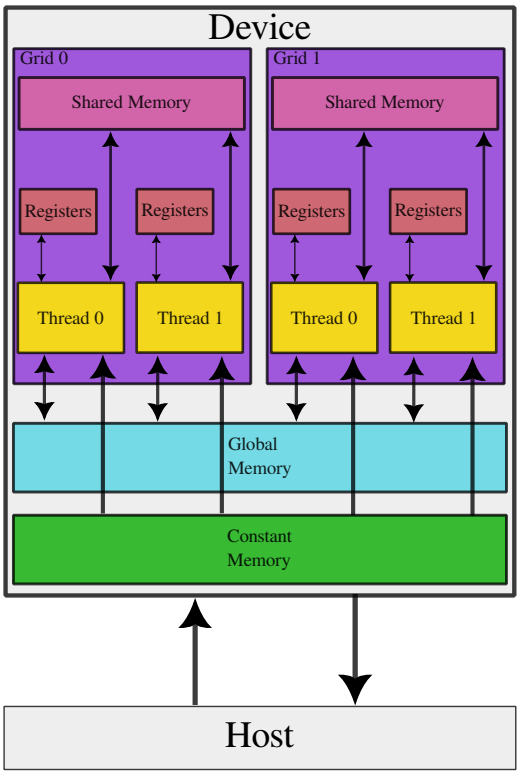


Figure 3.8: Graphic representation of how the different types of memory are accessed by the device and by the host

3.4 Profiling Tools

In order to identify possible bottlenecks and analyze how the CPU interacts with the GPU and vice versa, profiling tools are used. This section will briefly explain how these tools work and the main metrics to consider.

3.4.1 Nvidia Nsight Systems

Nvidia Nsight Systems is a system-wide tool that analyzes how the CPU interacts with the GPU device. It provides a timeline view of application execution, making it possible to observe aspects such as CUDA streams and memory transfer between Host and Device, or in the case of multi-GPU, Device-to-Device. This tool is able to track these activities with minimal overhead, allowing developers to detect and optimize critical sections of the application.

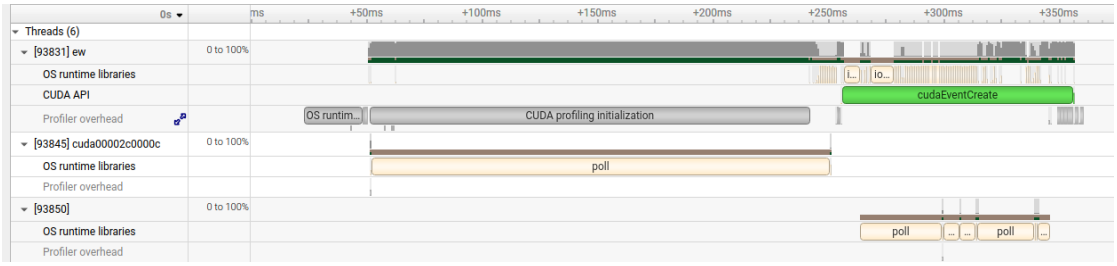


Figure 3.9: Example of an NVIDIA Nsight Systems Report

3.4.2 Nvidia Nsight Compute

Instead of analyzing the whole system, Nvidia Nsight Compute focuses more on analyzing singular aspects of each kernel, with a more detailed report. Figure 3.10 shows the initial window of an NCU report. From all the metrics reported, we are going to examine only a few of them.

ID	Estimated Speedup [%]	Function Name	Demangled Name	Duration [ns] (nan ns)	Runtime Improvement [ns] (nan ns)	Compute Throughput [%]	Memory Throughput [%]
0	86.96	niHashKernel	niHashKernel..int3..	2,624	2,281.76	2.02	12.47
1	95.00	DeviceRadixSortSin...	..DeviceRadixSortSi...	12,128	11,521.60	3.00	3.04
2	84.09	niReorderPosKernel	niReorderPosKernel..	2,624	2,206.61	2.61	15.91
3	84.87	niFindCellBoundsK...	niFindCellBoundsK...	2,400	2,036.87	2.18	12.86
4	61.48	initBoundariesKernel	initBoundariesKern...	2,016	1,239.50	11.08	10.04
5	78.53	p2gKernel_U	p2gKernel_U..float3..	72,736	57,119.22	9.33	63.74
6	79.30	p2gKernel_V	p2gKernel_V..float3..	67,744	53,721.06	10.01	67.74
7	78.90	p2gKernel_W	p2gKernel_W..float3..	72,192	56,960.95	9.68	62.97
8	60.99	p2gNormalizeKernel	p2gNormalizeKern...	4,992	3,044.71	8.85	29.41
9	50.00	dilateFluidFlags	dilateFluidFlags..	3,008	1,504.00	7.18	11.82
10	42.26	applyExternalForce...	applyExternalForce...	2,528	1,068.39	7.66	17.86
11	46.29	cell_classification_...	cell_classification_...	2,144	992.53	10.30	15.36
12	64.51	markFluidCellsKernel	markFluidCellsKern...	12,352	7,967.98	12.85	51.89
13	71.87	applyUBoundaryKer...	applyUBoundaryKer...	1,920	1,379.81	9.15	9.15
14	64.79	applyVBoundaryKer...	applyVBoundaryKer...	1,856	1,202.49	8.74	8.74
15	60.45	applyWBoundaryKe...	applyWBoundaryKe...	1,856	1,121.90	8.49	8.49

Figure 3.10: Nvidia Nsight Compute initial window; all the kernels with some basic information are listed

Occupancy The **Occupancy** measures how many warps are active per SM with respect to theoretical occupancy, or, in other words, is the percentage to which a device is able to process warps. A high difference between the Theoretical and Achieved Occupancy means that the workload is unbalanced. In Figure 3.11, both the theoretical and the achieved occupancy are shown. There are many factors that can influence occupancy. The tools provide graphs to understand how and what to change in order to increase the percentage. Figure 3.12 shows how different aspects of a kernel can vary occupancy.

Theoretical Occupancy [%]	100
Theoretical Active Warps per SM [warp]	48
Achieved Occupancy [%]	92.88
Achieved Active Warps Per SM [warp]	44.58

Figure 3.11: Screenshots of NCU's Occupancy section

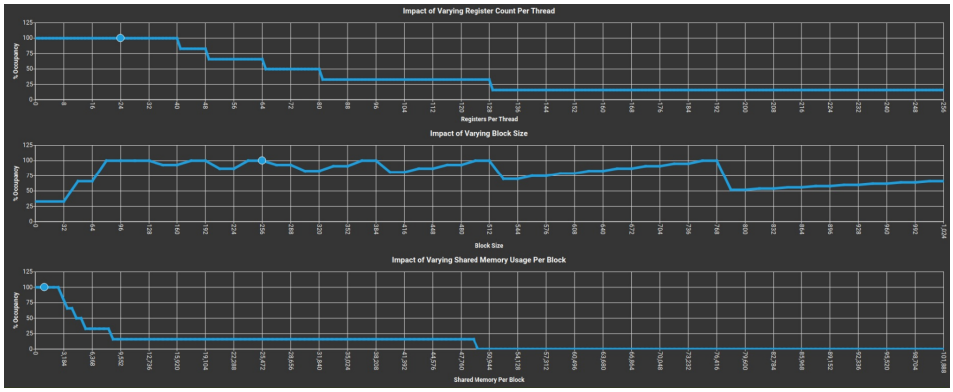


Figure 3.12: These graphs show how the occupancy can vary based on different factors, Register Per Thread, Block Size, and Shared Memory Per Block, respectively, from top to bottom.

Memory Hit Rate and Throughput This section of the tool analyzes how the device's memory is used by the kernel, measuring the Throughput and the Hit Rate of L1 and L2 cache memory, helping developers to understand how to increase performance by optimizing memory access patterns. In Figure 3.13, a few metrics were reported, indicating the percentage of hit rate for L1 and L2 cache memory, in this case 8.73% and 90.07%, respectively, and the percentage of bandwidth utilization.

Memory Throughput [Gbyte/s]	33.87
L1/TEX Hit Rate [%]	8.73
L2 Hit Rate [%]	90.07
Mem Busy [%]	50.91
Max Bandwidth [%]	63.74
Mem Pipes Busy [%]	9.33

Figure 3.13: Memory Workload Analysis

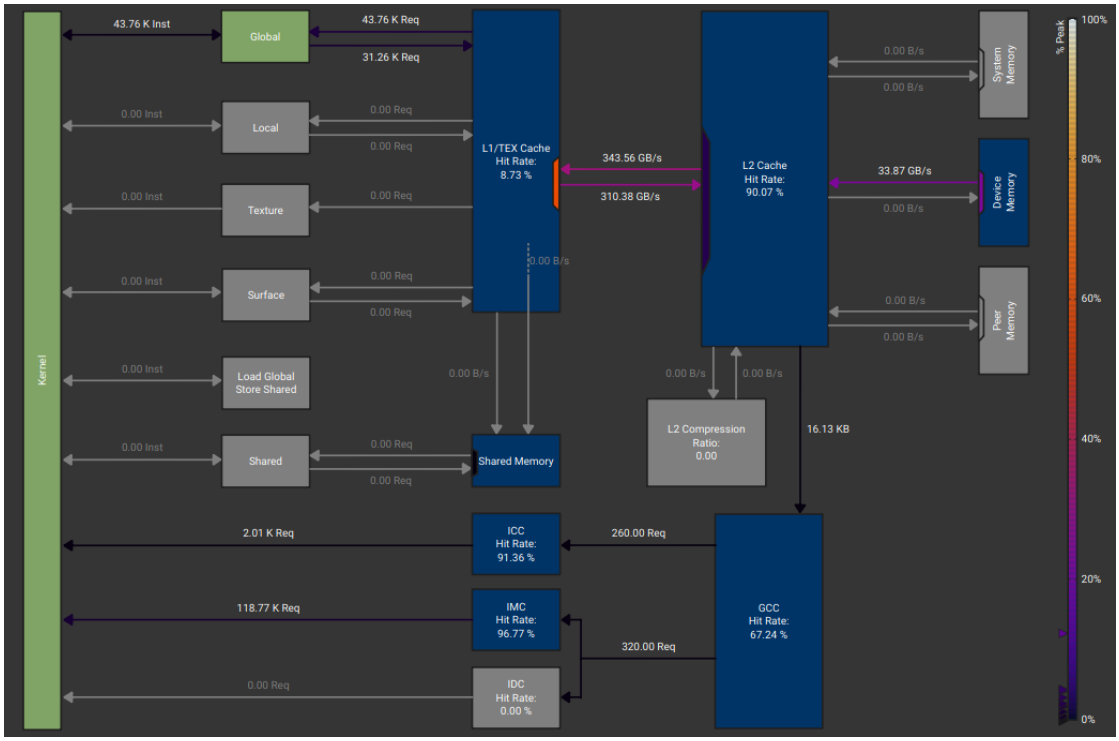


Figure 3.14: Memory Chart, graphic summary of the memory Hit Rate and Throughput

Roofline Model Before explaining the **Roofline Model**, we need to introduce the concept of Computational Intensity.

The Computational Intensity, or Operational Intensity, is the ratio of operations to the amount of data movement and is measured as FLOPs per Byte. A High Computational Intensity means that the application performs a large number of operations per unit of data transfer, while a Low Computational Intensity indicates that the application requires significant data movement relative to operations, which will cause a bottleneck and a relative performance degradation.

The **Roofline Model** is a visual performance model used by developers to optimize applications. The x-axis represents the Computational Intensity ($FLOPS/byte$), and on the y-axis is measured the Throughput/Performance (GFLOPS). A typical model is characterized by a Transition Point, also known as a knee point, which indicates that an application is memory-bound or compute-bound. When an application, or a kernel, is memory-bound, the system spends more time moving data than performing operations; on the other hand, when an application is compute-bound, any further increase in computational intensity will not yield any throughput gains. When an application reaches the plateau,

it means that it has reached the maximum theoretical computational throughput of the hardware. For example, looking at Figure 3.15, we have 3 different applications:App_1, App_2 *and* App_3. Since App_2 and App_3 are above the knee point, they are compute-bound, while App_1 is memory-bound.

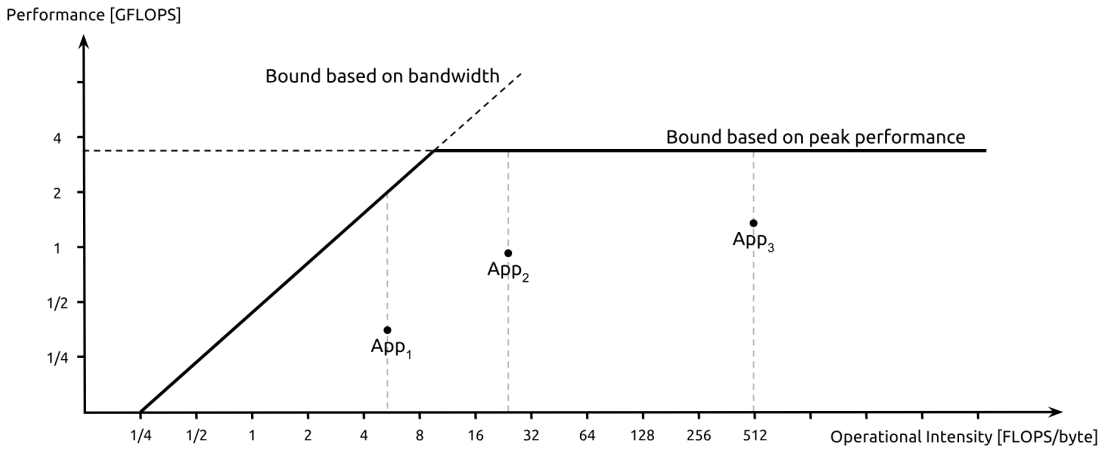


Figure 3.15: Example of Roofline Model, in the x-axis is measured the Operational Intensity, in the y-axis the Throughput

To increase computational intensity, several techniques can be used, such as leveraging shared memory or reorganizing data structures to improve data coalescing, which reduces global memory transfers. Performance can be improved by increasing Instruction-Level Parallelism (ILP) or by using compiler optimizations, such as the `-use_fast_math` flag.

Figure 3.16 shows a sequence of typical optimization steps aimed at increasing application throughput. Starting from a memory-bound state, the first step is to optimize memory access patterns to maximize effective bandwidth and increase computational intensity. This shifts the application to the right, moving it from the memory-bound region toward the compute-bound zone. Finally, by fully exploiting compute resources and ILP, the application reaches the performance plateau, achieving maximum theoretical compute throughput.

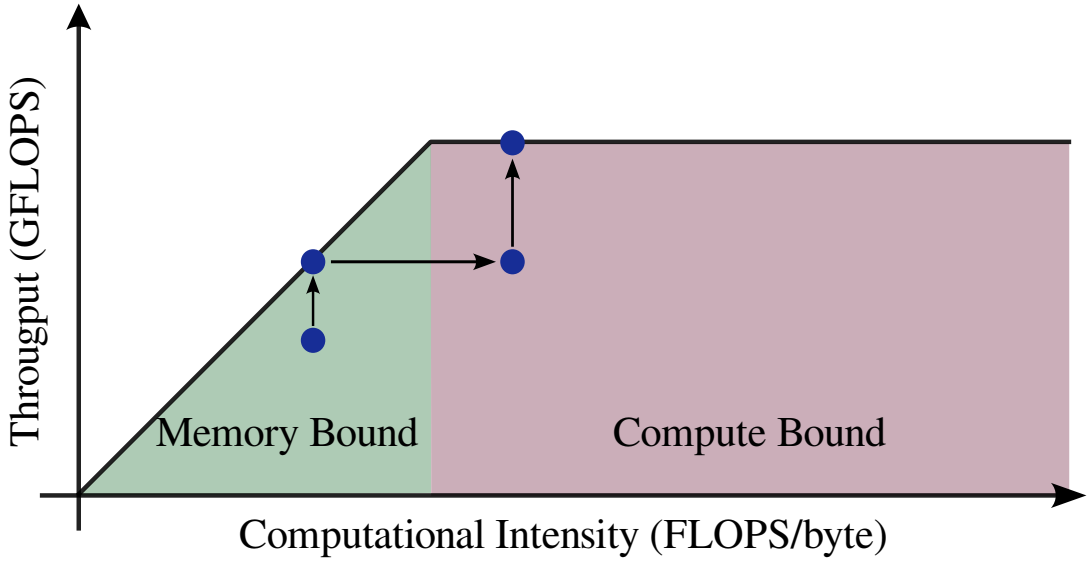


Figure 3.16: Example of optimization using the roofline model.

3.5 Enterprise and Consumer Differences

In this section, the main hardware differences between an enterprise and a consumer model will be analyzed. From now on, we will take the Nvidia A100 and RTX3070 as examples. We also have to take note that these two devices are designed with different focuses: while Consumer Hardware is used primarily for gaming and video editing, the main task of Enterprise Hardware is to process a large amount of data, in fact are installed in large numbers in data-center servers. In the field of AI, Enterprise Hardware is strongly used for its computational power. Also, one important aspect is the cost, the launch price of the RTX 3070 was 500USD, with respect to 10.000USD for an A100, and the energy consumption is 250W and 400W at full throttle, respectively, for RTX3070 and A100.

3.5.1 Memory

The main difference between Enterprise Hardware and CHW in terms of memory is, in the first place, the amount of memory with which a device is equipped, which is

significantly higher for Enterprise Hardware, and the memory bandwidth, $448GB/s$ for the RTX3070 and $1.55TB/s$ for the A100.

One important aspect of memory differences is the technology adopted by the two categories, a GDDR-like memory for a Consumer Hardware an HBM memory for Enterprise Hardware; Figure 3.17 will be used as a reference. A GDDR-like memory is a cheap memory that is placed around the chip, providing good performance. The HBM model, instead, is a peculiar type of memory, where the memory chips are stacked vertically, and each stack is placed on the same processor's package via an interposer, providing performance greater than a GDDR memory but with a major cost.

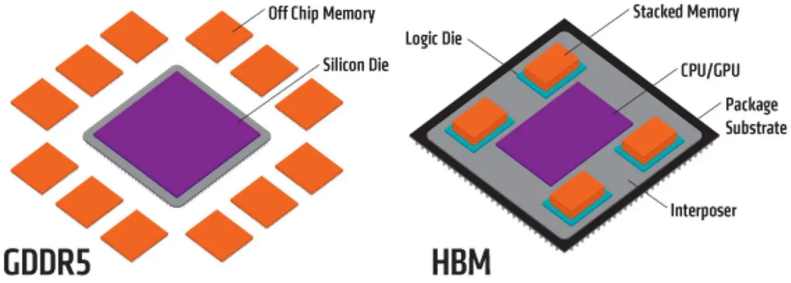


Figure 3.17: Hardware differences between DDR and HBM memories

3.5.2 Hardware Accelerators

Ray Tracing Core A Ray Tracing Core is a dedicated hardware accelerator whose main purpose is to render 3D effects in real time, such as rays of light, shadows, and reflections. In Figure 3.18 there is an example of how rays work. These cores are more present in consumer hardware, and since it is more relevant to gaming, we will not go into detail.

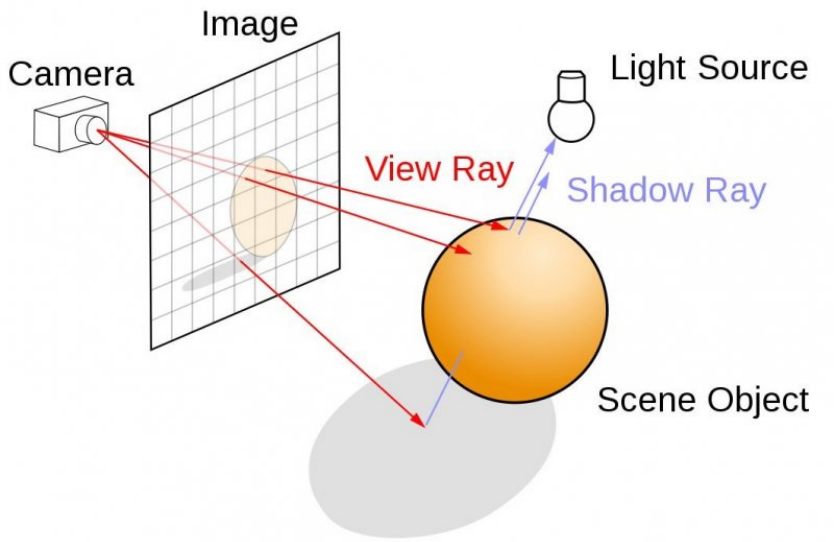


Figure 3.18: Ray-tracing example

Tensor Core A Tensor Core is a specialized hardware accelerator designed to perform Matrix-Multiply-Accumulate (MMA) operations efficiently. These units are deployed in both consumer and enterprise architectures. Recently, Tensor Cores have become an important component due to their wide use in Artificial Intelligence (AI) and Deep Learning, where matrix operations are the underlying foundation of model execution.

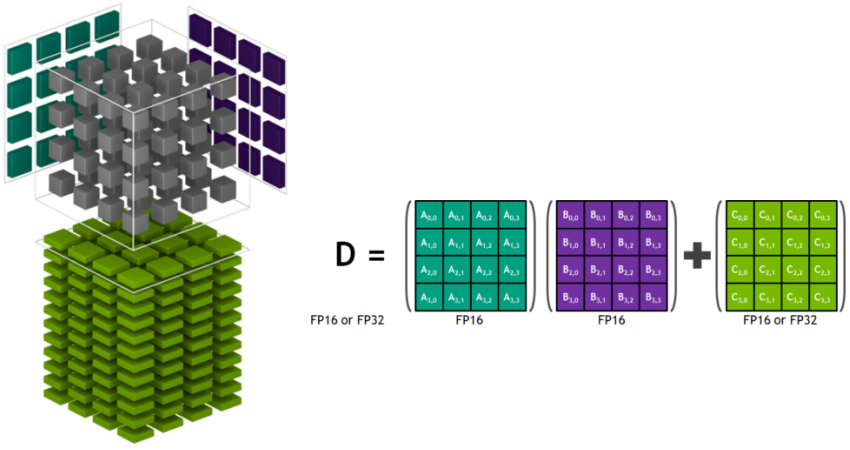


Figure 3.19: Visual representation of how Tensor Cores work

Chapter 4

Traditional Approaches to Triangle Counting

In Section 2.3, we introduced the main algorithms. In this chapter, we will first cover the main search algorithms, and then, in Section 4.3, discuss the state-of-the-art and the relevant frameworks, both CPU and GPU.

4.1 Matrix Multiplication Approach

Methods regarding the adoption of adjacency matrices are very elegant from a mathematical point of view, but are unfeasible in most cases, since allocating a complete matrix for a real-world graph, as discussed in Section 2.1.4, requires more space than a GPU can provide. This particular aspect leads to algorithms known as Sparse General Matrix-Matrix Multiplication (SpGEMM). Given two different matrices, A and B, matrix C is the result of $A \times B$. C can either be computed in the classical way, by multiplying the whole matrices together, or it can be computed step by step by decomposing the matrices into sub-matrices; in this way, it is possible to exploit the sparsity of the input matrices, avoiding unnecessary computations and reducing the overall memory footprint. Let's suppose we divide each matrix into four sub-matrices, so that
$A = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix}$, $B = \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}$, $C = \begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix}$
To calculate C_{00}, we need to compute the sum of sub-products $A_{00} \times B_{00} + A_{01} \times B_{10}$. Figure 4.1 shows a visual example. In this way, C can also be viewed as:
$C = \begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} = \begin{pmatrix} A_{00} \times B_{00} + A_{01} \times B_{10} & A_{00} \times B_{01} + A_{01} \times B_{11} \\ A_{10} \times B_{00} + A_{11} \times$

Breaking down a single large matrix into smaller sub-matrices enhances both scalability and performance, since it allows more matrix multiplications to be computed in parallel, while also avoiding empty sub-matrices in order to save precious resources. This approach is very effective when applied to real-world

graphs, where the probability of encountering an empty matrix is very high. Algorithm 4 shows an example of Hadamard Product combined with SpGEMM.

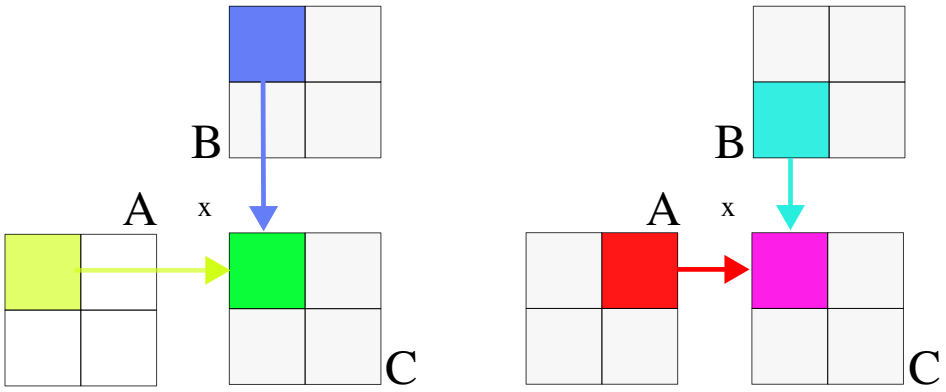


Figure 4.1: Simple explanation of sparse matrix multiplication. Given two matrices, A and B, C is the result of the multiplication between A and B.

Algorithm 4 SPGEMM combined with Hadamard Operator

Require: $G = (V, E)$, $M = ADJ(G)$, $N = [V, V], k$ ▷ M is the complete adjacency matrix for G, N is empty

$triangles = 0$

for $(int\ i = 0; i < k; i++)$ **do** ▷ k is the number of sub-blocks per row

  for $(int\ j = 0; j < k; j++)$ **do**

    for $(int\ l = 0; l < k; l++)$ **do**

      $pos1, pos2 = calculateSubBlock(i, j, l);$

      $N[i][j] += M[i][pos1] \times M[pos2][j]$

    end for

    $triangles += \sum(N[i][j] \odot M[i][j])$

  end for

end for

$triangles = \frac{triangles}{6}$

But the problem of the enormous amount of memory used to store the matrix still remains. To overcome this issue, one solution is to store only a portion of the matrix, usually the upper/lower triangular matrix. Since the original matrix is symmetric, the resulting off-diagonal sub-matrices come in symmetric pairs, which lets us store only one of each pair and apply the transpose operator when needed. For example, looking at Figure 4.2, if we want to compute B^2, we will need to compute $B \times A + C \times B + E' \times D$. In terms of space, in this case, we are only using

$\approx 55.5\%$ of the original space. More generally, increasing the number of sub-blocks reduces the amount of space needed to store the matrix. Algorithm 5 is a variant of Algorithm 4, which uses the property of a symmetric matrix, transposing the sub-block when needed, in order to save space.

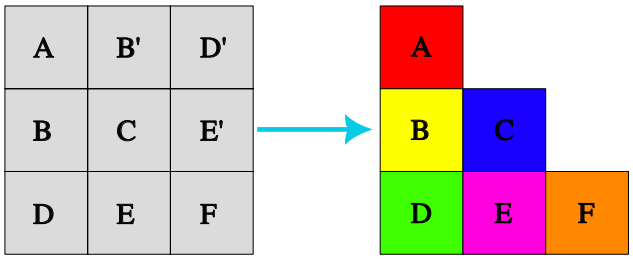


Figure 4.2: Transition from a complete symmetric matrix to a lower triangular matrix.

Algorithm 5 SPGEMM storing only the lower triangular matrix

Require: $G = (V, E),\ M = ADJ(G)\ , N = [V, V], k$ ▷ M stores only the lower triangular sub-blocks of the adjacency matrix for G, N is empty

$triangles = 0$

for $(int\ i = 0; i < k; i++)$ **do** ▷ k is the number of sub-blocks per row

  for $(int\ j = 0; j \leq i; j++)$ **do**

    for $(int\ l = 0; l < k; l++)$ **do**

      $(pos1, pos2) = calculateSubBlock(i, j, l)$

      $block1 = (i \geq pos1)\ ?\ M[i][pos1]\ :\ M[pos1][i]^T$

      $block2 = (pos2 \geq j)\ ?\ M[pos2][j]\ :\ M[j][pos2]^T$

      $N[i][j]\ +\!= block1 \times block2$

    end for

    $triangles\ +\!= \sum(N[i][j] \odot M[i][j])$

  end for

end for

$triangles = \dfrac{triangles}{3}$

4.2 Intersection-based Approach

In order to check the intersections that are in common between two different edges, different algorithms have been developed. In this section, we will analyze 3 different algorithms, which are the most used ones.

4.2.1 Merge Path

The Merge-Based Approach (also known as the Two-Finger Pointer or Linear Merge Approach) computes the intersection of two sorted adjacency lists, L_u and L_v, representing the neighbors of vertices u and v. In triangle counting, every common neighbor shared by u and v completes a triangle (u, v, w). Algorithm 6 reports the pseudocode. It uses two pointers (or "fingers"), one at the head of each list. Then, it compares the elements at the current pointer positions. If the elements are equal, a triangle is found. Both pointers advance to the next element. If the elements are unequal, the pointer to the smaller value is moved to the next element. The process continues until either list ends.

Algorithm 6 The Merge-Based Approach.

Require: N = Adj(u), M = Adj(v)
  $triangles = 0$
  $i = 0, j = 0$
  $end1 = len(N), end2 = len(M)$
  $a = N[0], b = M[0]$
  while $i < end1$ && $j < end2$ **do**
    if $(a == b)$ **then** $triangle = triangle + 1$
    if $(a \leq b)$ **then** $i = i + 1$
    if $(a \geq b)$ **then** $j = j + 1$
    if $(i < end1)$ **then** $a = N[i]$
    if $(j < end2)$ **then** $b = M[j]$
  end while

This approach strictly requires that the adjacency lists are pre-sorted, and it is most efficient when the two lists have similar lengths and a high degree of overlap. The approach is linear in the sizes of the lists; i.e., if M and N are the lengths of the two adjacency lists, the complexity is $O(M + N)$. This method is highly favored in GPU implementations because it accesses memory linearly, which is ideal for memory coalescing and predictable cache behavior.

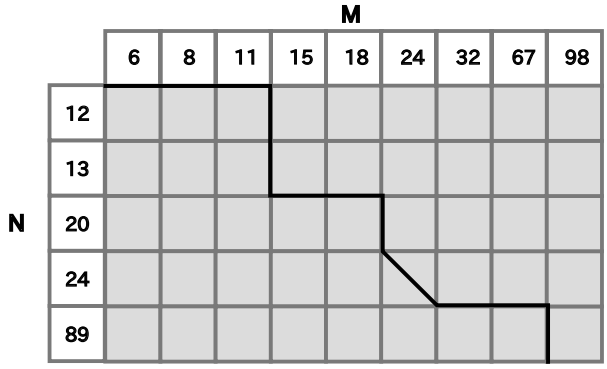


Figure 4.3: Merge-Based

4.2.2 Binary Search

The binary search approach is an intersection technique optimized for cases where the lengths of the two adjacency lists differ significantly. Algorithm 7 reports the pseudocode.

Algorithm 7 The Binary Search Approach.

Require: N = Adj(u), M = Adj(v)
  $triangles = 0$
  for $i = 0; i < len(N); i++$ **do**
    $a = N[i]$
    $j = 0,\ e = len(M)$
    while $j < e$ **do**
      $m = j + ((e - j)/2)$
      $b = M[m]$
      if $a == b$ **then**
        $triangles++$
        break
      end if
      if $(b < a)$ **then** $j = m + 1$
        else $e = m$
    end while
  end for

Unlike the merge-based approach, which scans both lists linearly, this method treats the lists asymmetrically by identifying the shorter list (L_{small}) and the longer list (L_{large}). After that, it iterates through every element e in L_{small}, and for each element e, performs a binary search within L_{large} to check for its existence. If

the binary search finds a match, a common neighbor is found, and a triangle is counted. As with the merge-based approach, L_{large} must be sorted to support $O(\log N)$ lookups.

The approach is significantly faster than a merge when L_{small} is much smaller than L_{large} (e.g., searching for a few neighbors of a "hub" vertex in the list of a low-degree vertex). If N is the size of the shorter list ($|L_{small}|$) and M the size of the longer list ($|L_{large}|$), the time complexity is $O(N \cdot \log M)$, which is often much smaller than the $O(N + M)$ required by the merge approach.

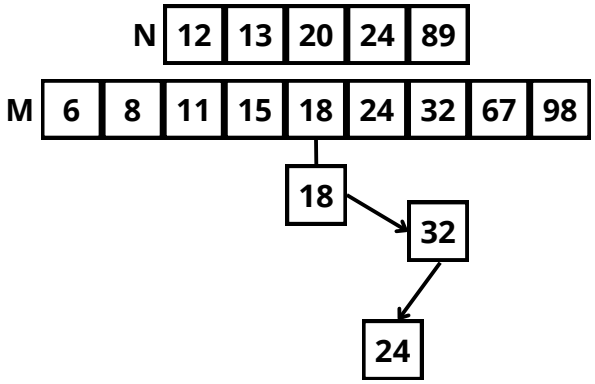


Figure 4.4: Binary Search

4.2.3 Hash Buckets

The hash bucket-based approach aims to accelerate the search process to a near-constant-time operation, $N(u)$ and $N(v)$. This method typically consists of two distinct phases:

- **Build Phase:** The algorithm selects the smaller adjacency list, e.g., $N(u)$, and maps its elements into B buckets using a hash function, e.g., $h(x) = x \pmod B$.
- **Probe Phase:** The algorithm iterates through the larger list $N(v)$ in parallel. For each neighbor $y \in N(v)$, the corresponding bucket index $i = h(y)$ is computed. The search is then restricted solely to the elements within bucket i.

While this technique effectively reduces the average-case complexity to $O(|N(v)| \cdot \frac{|N(u)|}{B})$, it incurs a nontrivial memory overhead due to the need for auxiliary storage to maintain the bucket structure. Furthermore, the performance is sensitive to the choice of the B parameter: a B that is too small leads to longer buckets and increased search times. In contrast, an excessively large B wastes global GPU memory and can reduce cache hit rates. Unlike "preprocessing-free" methods,

this approach trades off a portion of the memory budget for increased throughput during the intersection stage.

Algorithm 8 Hash-Bucket

Require: N = Adj(u), M = Adj(v), Buckets, $bucket_size$

```
triangles = 0
i = 0, j = 0
for each vertex n in N do
    pos_bucket = n mod bucket_size
    Buckets[pos_bucket].insert(n)
end for
for each vertex v in M do
    pos_bucket = v mod bucket_size
    for each element x in Buckets[pos_bucket] do
        if x == v then
            triangles++
        end if
    end for
end for
```

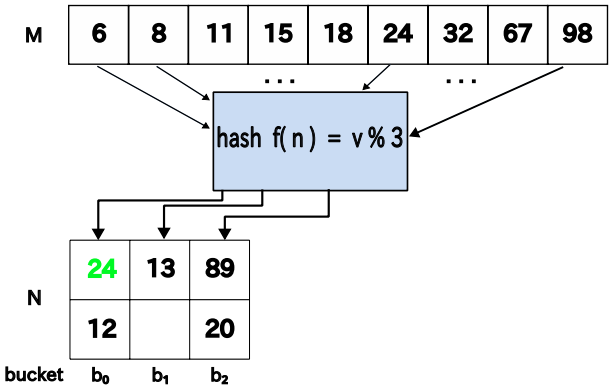


Figure 4.5: Hash Bucket

4.3 State of the Art

Il conteggio dei triangoli nei real-world graphs è oggetto di numerosi studi, sia su architetture CPU e GPU sia su architetture custom particolari; la letteratura può essere divisa in diverse sezioni a sua volta, basate sulla tipologia di hardware studiato, sulla tecnica usata e sul livello di granularità con cui il lavoro viene

distribuito tra i vari core/thread. Triangle counting in real-world graphs has been the object of studies, both on CPU and GPU architectures and on some custom architectures like [19]. The literature can be divided into two main categories, based on target hardware, technique used, and accuracy. This thesis will focus only on exact triangle counting; approximate triangle counting algorithms will not be discussed.

One of the first studies on the problem is that of Schank et al.[13], who compare different algorithms for finding and counting all triangles in large-scale graphs. This work is considered one of the most important.

Over time, many general-purpose frameworks for graph analytics have been developed, both on CPU and GPU. Galois[18] is a lightweight framework that exploits multithreading on the CPU to execute different algorithms on graphs, including triangle counting. Although Galois has not been developed specifically for triangle counting, it offers good terms of gains when moving from CPU to GPU.

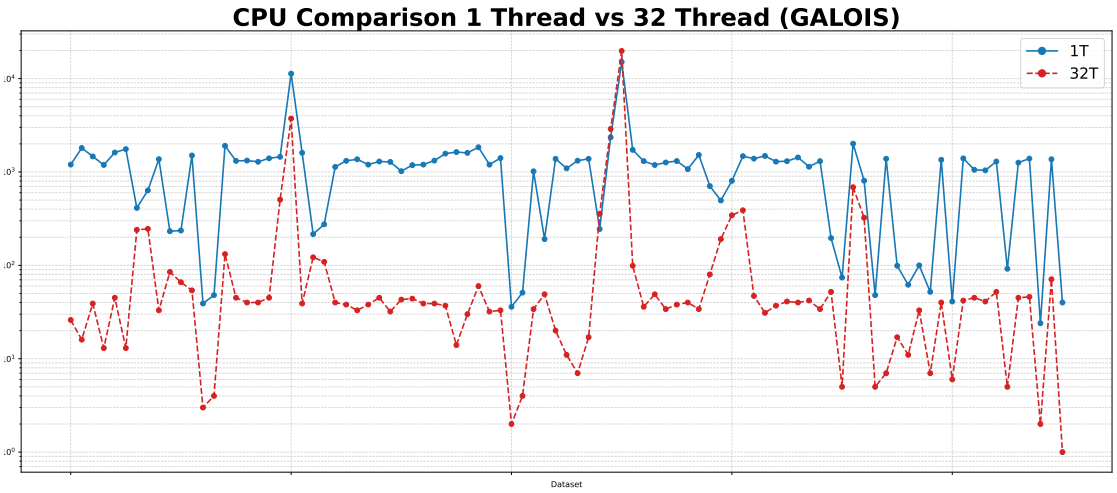


Figure 4.6: CPU triangle counting performance comparison, Single-thread vs Multi-thread configuration

The direction of research that is most relevant to this thesis is the one that addresses triangle counting specifically on GPUs. H-Index[16] is a hash-index technique for parallel triangle counting on GPUs, that relies on a hash-based intersection, previously discussed in section 4.2.3 of this thesis. Subsequently, from the same author, TRUST[14] has been developed, which revisits the problem of triangle counting on GPUs, by introducing improvements to workload balancing across the threads and to memory access pattern, which is a critical aspect as discussed before.

More recently, research has moved toward specialized GPU component and custom architectures. Su et al. [19] investigate the use of the custom architecture UpDown[20] to accelerate the triangle counting and k-truss computation. This architecture implements a memory module (Figure 4.7 shows an example) to improve data-intensive workloads, including graph processing, demonstrating important results. Instead, ToT[21] explores the use of Tensor Core, discussed in section3.5.2, to accelerate triangle counting, proposing an algorithm similar to the one presented in Section 4.1, achieving an important speedup in end-to-end time. Regarding sparse

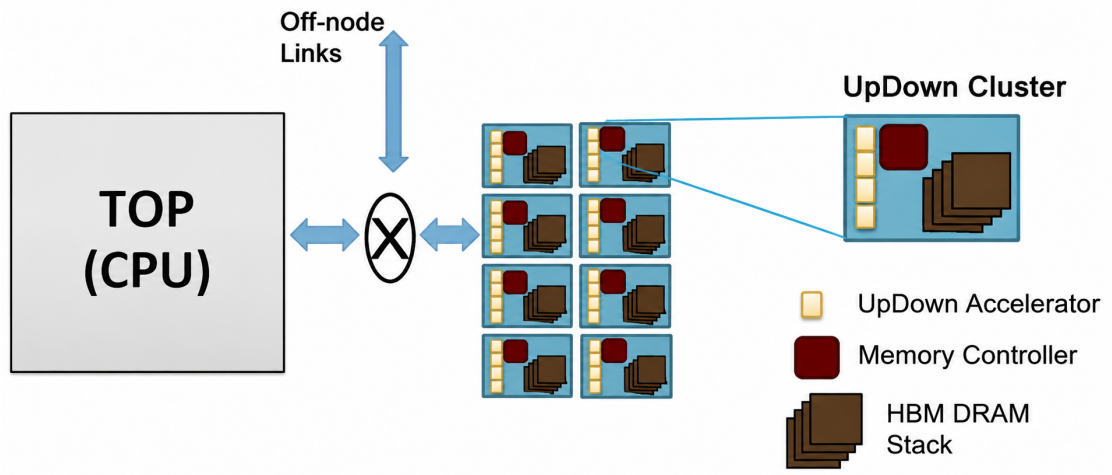


Figure 4.7: An UpDown Node

matrix multiplication-based (SpGEMM), GraphBLAST[22] implements a GPU-based graph analytics framework, based only on linear algebra. It offers primitives for different operations such as SpGEMM. cuGraph [23] is part of NVIDIA's RAPIDS suite, which provides an implementation of different algorithms, including triangle counting, that is widely used in both industry and research due to its ease of access and use. Gunrock[17] is a general-purpose GPU framework for graph analytics that offers a set of primitives that can be combined to implement different operations.

Current limits and proposed solution Two current limits can be found in existing approaches. First, the previously discussed works handle load balancing by assigning a single node/edge to a single warp. TRUST[14] balances the load at the warp level; Tricore[15] dynamically balances the load at the GPU level, through a work stealing assignment of tasks by a scheduler process. Load balancing at the warp level is not handled, since the authors rely on the fact that the binary search algorithm produces a balanced load between threads, but without explicitly calculating the strategy. None of the work discussed before introduces an explicit

estimate of the load at the single thread level within the warp, nor an adaptive decision made at runtime on how the threads within a warp must collaborate to resolve one or more intensive intersections.

Moreover, most of the work discussed adopts a single intersection technique, applied uniformly across the entire graph, without exploiting the possibility of combining multiple techniques. This choice does not allow the strengths of each technique to be fully exploited.

This thesis addresses both of the limitations described above. First, a dynamic load balancing mechanism internal to the single warp is proposed: each thread first estimates the amount of work needed to resolve its intersection, and then the total warp workload is computed by summing all the previously calculated values. If the workload exceeds a certain threshold, the warp is flagged as **heavy** and threads within the warp will collaborate to resolve the intersection; otherwise each thread will work on its own set of intersections. Unlike the previously cited approaches, which operate at GPU or warp granularities as a whole, this approach introduces an adaptive decision taken at runtime for each warp, between independent and collaborative execution, combining both binary search and merge path to check common intersections.

Chapter 5

GetterTri

GPUs have proven to be excellent for graph analysis, thanks to their parallel computational power. When a process is launched on a GPU, it is called a Kernel. On Nvidia GPUs, the threads in a kernel are grouped into blocks; these threads are, in turn, subdivided into groups of 32 threads known as Warps. Threads within a warp can communicate with each other using specific primitives[24]. This inter-thread communication can be extended using Cooperative Groups [25], which enable communication and synchronization between threads at different granularities. In our case, we opted for intra-warp communication. In this chapter, we will explain step by step how we developed GetterTri, discussing how we made the main choices.

5.1 Edge or Node Iterator

In section 2.3 we discussed the main intersection-based algorithms, the Node and Edge Iterator. The first step was to choose the base algorithm. In the previous chapter, we talked about the state-of-the-art and the algorithms behind it; one particular aspect is that these implementations are either vertex-centric or edge-centric, i.e., how parallelism is treated. An interesting work is HTC [26], which proposes an algorithm that adaptively selects one from node and edge iterators for each vertex. We ran a few tests to see which one between the node and the edge iterator was better, in Figure 5.1 shows a graph that reports the kernel time of the two paradigms. Except for a few cases, the edge iterator achieved better times with respect to the node iterator, so we opted for this one, likely because edge-centric methods distribute work more evenly across threads. We therefore opted for the edge-centric paradigm as the basis for GetterTri.

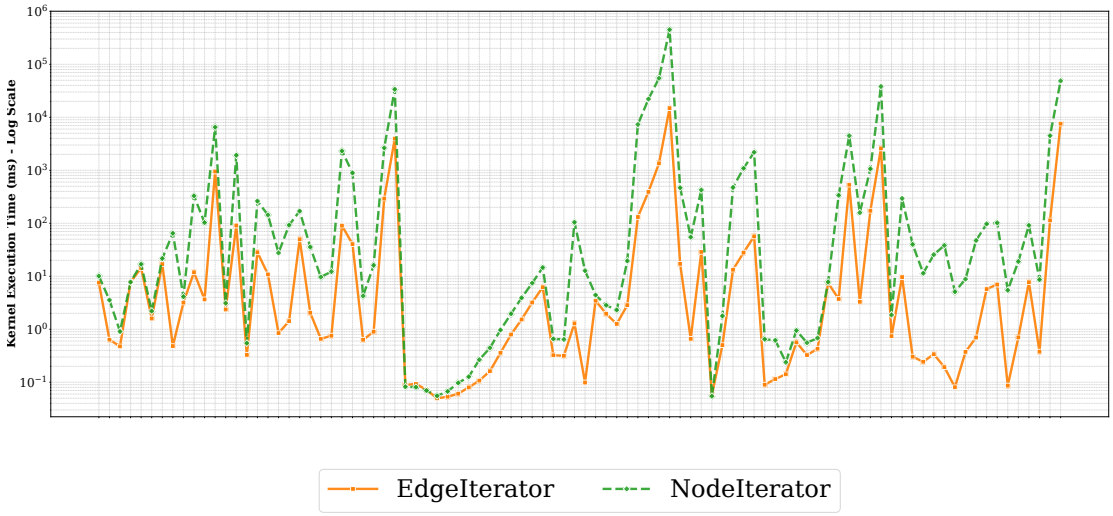


Figure 5.1: Performance comparison between Edge and Node iterators

5.2 Preprocessing

Preprocessing is a preliminary phase in which the structure of the graph is altered in order to boost the performance, and typically consists of transforming an unoriented graph into an oriented one, keeping only one edge from one vertex to another, based on some rule. Applied to triangle counting also eliminates redundant computation, since each triangle is counted only once, and not multiple times. From a memory perspective, instead, it reduces the size of the edge list, since we are keeping only one edge.

This section first covers a few common techniques used to perform preprocessing on graphs, then we will explain how we handled this problem. Figure 5.2 will be used as a reference starting graph. In this graph, a connection between two nodes also implies storing two edges, i.e., given two nodes A and B, we will store one edge from A to B and one edge from B to A. For each method, pseudocode and visual representation of the resulting graph will be provided.

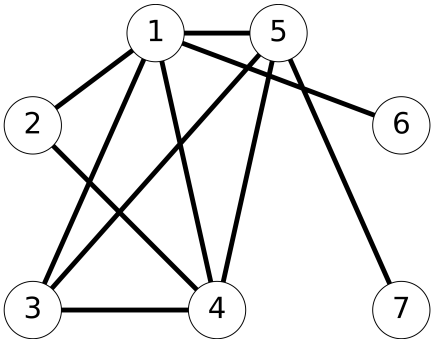


Figure 5.2: Starting Graph

ID based The orientation based on the vertex Id, is the simplest type of preprocessing among those presented. An edge between two nodes is kept only from the lower ID vertex to the higher ID vertex. Even if simple, this technique is particularly effective, and does not require any expensive computation beyond a single comparison per edge. However, this method presents a big limitation: since Ids are assigned independently of the graph's structural properties, this strategy does not guarantee the optimal orientation.

Algorithm 9 Preprocessing ID-based.

Require: $G = (V, E)$
  for each $edge(u, v) \in E$ **do**
    if $id(v) > id(u)$ **then**
      $KeepEdge$
    else
      $RemoveEdge$
    end if
  end for

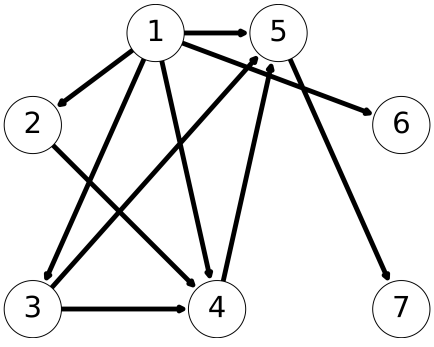


Figure 5.3: Starting graph after ID-based preprocessing

Degree based This is an alternative strategy, where each connection is oriented based on the degree, from the lower-degree vertex to the higher-degree vertex. This strategy is particularly effective for different reasons, such as improving memory coalescence and not requiring too much time to be performed. This orientation reduces the theoretical worst-case cost of the intersection step, "at the price of requiring the degree of every vertex to be computed beforehand. This type of preprocessing is very common among the state-of-the-art; one example is Tricore.

Algorithm 10 Preprocessing Degree-based.

Require: $G = (V, E)$
  for each $edge(u, v) \in E$ **do**
    if $deg(v) > deg(u)$ **then**
      $KeepEdge$
    else
      $RemoveEdge$
    end if
  end for

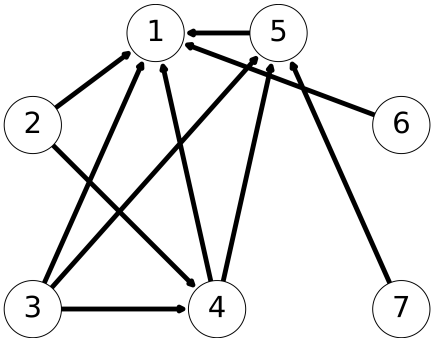


Figure 5.4: Starting graph after degree-based preprocessing

Re-Label Ordering A more complex implementation is the re-label technique, which applies a degree-based filter after having sorted and relabeled the vertices. First, the vertices are sorted based on their degree, typically in ascending order, and reassigned new identifiers according to this order. After having changed the connection from the old to the new values, a filter is applied, usually a degree-based one. This strategy is also known as **rank-by-degree** and has the same effect on the length of the neighbor list, but with additional benefits from a memory perspective: since the vertices with similar degree are **closer** to each other, this will result in cache-friendly accesses and a more coalesced data structure. However, even if this technique offers a big advantage, it requires some expensive computation, mainly

for the relabel phase, even so, frameworks like TRUST[14] perform this type of preprocessing.

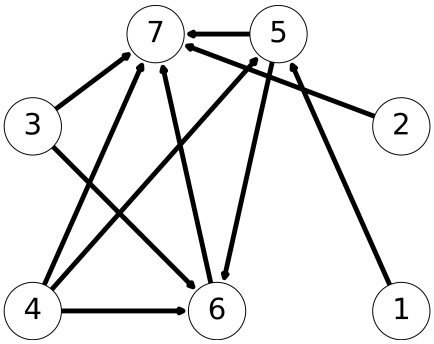


Figure 5.5: Starting graph after relabel-based preprocessing

Algorithm 11 Preprocessing with re-label.

Require: $G = (V, E)$
  $G \leftarrow SortVertex(G)$
  for each $edge(u, v) \in E$ **do**
    $(u, v) \leftarrow UpdateVerticesID(u, v)$
  end for
  for each $edge(u, v) \in E$ **do**
    if $deg(v) > deg(u)$ **then**
      $KeepEdge$
    else
      $RemoveEdge$
    end if
  end for

GetterTri preprocess Before implementing any of these strategies, two distinct doubts about preprocessing emerged : **how much time is gained, and how much time is spent?**. For both questions, the answer turned out to be substantial, but let us proceed in order.

Regarding the gain in kernel time, even a simple ID-based implementation(Algorithm 9) yields significant improvement, naturally, the more complex the applied preprocessing technique, the larger the improvements.

The other question instead regards **how much time is spent during the preprocessing**. This problem is generally observed, but in execution times and comparative tables it is always excluded, justified by the fact that the graph is

processed only once. From our point of view, this problem should not be overlooked; works such as ToT[21] do report end-to-end time.

We have sought a compromise, looking for a solution that would be as light as possible; the solution that we adopted is that of implementing an ID-based preprocessing, that discards the edges while reading the input file, in this way, we minimize the computational overhead as much as possible. In our version, instead of discarding edges based on the condition $id(u) > id(v)$, we discard them based on the opposite condition, $id(v) > id(u)$; according to our tests, reported in Figure 5.7, this choice shows better results. Table 5.1 reports the time required by each method.

The times reported under ID, Degree, and Relabel correspond to the time required to perform preprocessing; the time needed to read and parse the file is reported under the column Read and is excluded from preprocessing time; the values under the column GT correspond to the difference between Read time and the time needed for our proposed approach, which combines in a single-step parsing and preprocessing. An important consideration must be made: as visible in the table, the values under GT are negative; this means that our approach actually saves time overall. Even though our implementation performs extra steps with respect to a normal read, avoiding the need to load half of the edges into memory drastically reduces the overhead.

In light of this, we decided to adopt preprocessing combined with file reading.

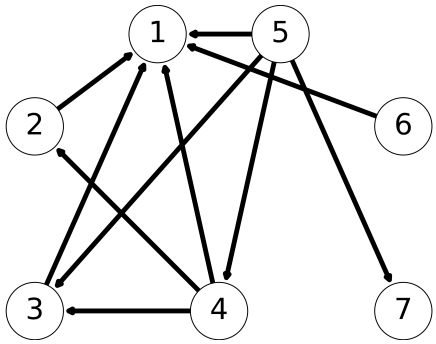


Figure 5.6: Starting graph after GT-based preprocessing. It differs from Figure 5.2 in the applied rule: instead of the classical $id(v) > id(u)$, the rule $id(u) > id(v)$ is used.

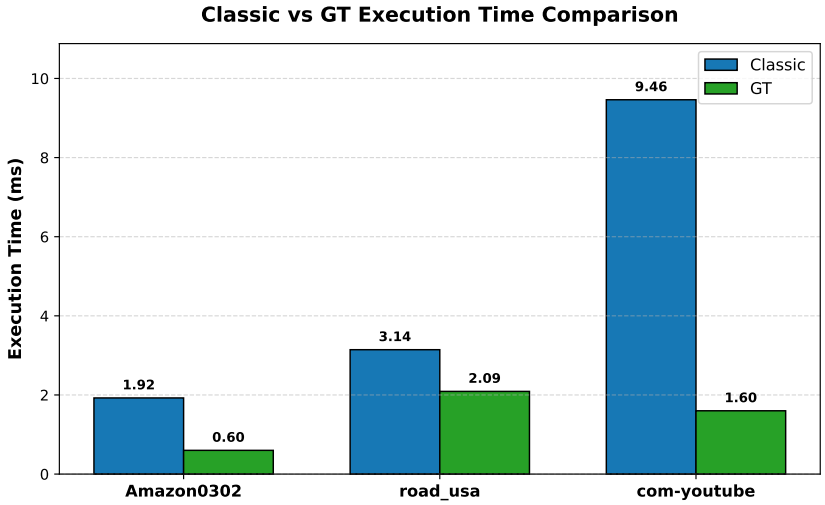


Figure 5.7: Comparison between the classic ID-based orientation ($id(v) > id(u)$) and the orientation adopted by GetterTri ($id(u) > id(v)$).

File	Read	ID	Degree	Relabel	GT
Amazon0302	22.87	8.75	13.56	26.58	-5.42
com-lj	924.03	369.08	687.62	923.51	-336.17
com-orkut	3549.15	912.03	2206.23	2256.81	-1310.43

Table 5.1: Breakdown of preprocessing times (in milliseconds) for different graphs. The **Read** column reports the time required to read and parse the input file alone. **ID**, **Degree**, and **Relabel** report the time required by each orientation strategy on top of the parsed graph. **GT** reports the difference between the classic file-reading time and the time required by GetterTri combined parsing-and-filtering approach; negative values indicate a time reduction.

5.3 Workload balancing and Memory Optimization

In triangle counting, graphs containing a large number of **hub** nodes represent an obstacle to overcome. To address this problem, we explored several different approaches. Starting from an edge-iterator design, we first attempted to assign an entire warp to each individual intersection. While this approach was particularly efficient for graphs with a high number of hub nodes, it introduced additional

overhead for the majority of edges in graphs without such hubs, degrading overall performance.

As an alternative, we implemented a version that launched two distinct kernels depending on the estimated cost of each intersection. This design reduced the overhead observed in the previous approach, but yielded only marginal performance improvements. This experiment led to the idea behind our final algorithm: a single, adaptive kernel that relies on intra-warp communication, based on the estimated workload cost, to redistribute work dynamically without interrupting the execution flow.

5.4 Combination of Traditional Approaches

The design of the algorithm behind GetterTri is the result of a deep analysis of the combination of different intersection-based techniques. Starting from the edge-iterator paradigm, we evaluated the efficiency of **binary search** relative to a **Merge-based** approach. The evidence from the tests showed that the efficiency of these techniques is closely related to the topology of the adjacency lists involved:

- **Merge Path**: provides high throughput when the adjacency lists are of similar length, as it ensures a balanced linear scan of both lists.

- **Binary Search**: exhibits superior performance when there is a significant disparity between the lengths of the adjacency lists ($|N| \ll |M|$).

The adoption of a hybrid approach enables dynamic selection of the best strategy. However, power-law graphs reveal a critical limitation: the presence of hub nodes (nodes with extremely high degrees). These nodes generate massive workloads that, if naively managed, lead to computational resource saturation and severe load-balancing issues among threads. Specifically, one thread can become stuck while other threads remain idle.

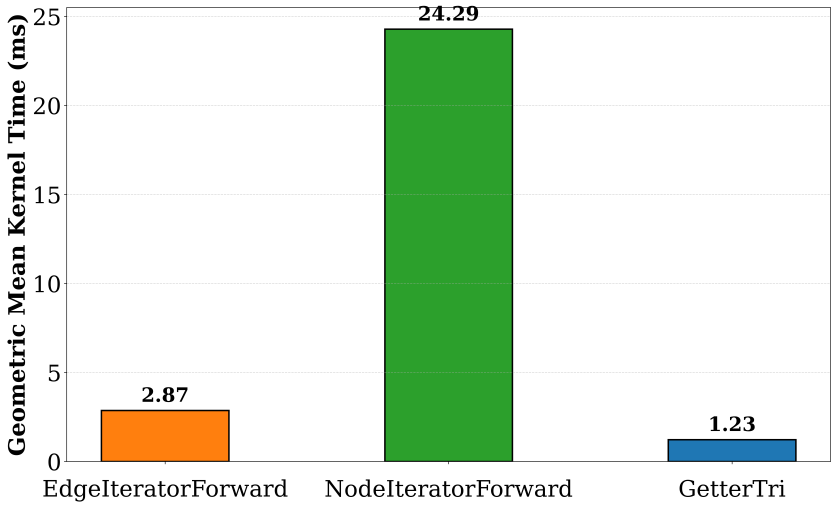


Figure 5.8: Performance comparison between Node Iterator, Edge Iterator and GetterTri

5.5 Cuda Cooperative Groups and Warp Primitives

The proposed solution leverages CUDA Cooperative Groups, which simplify intra-warp communication and synchronization and make them more robust for the dynamic load-balancing strategy. **Warp primitives** are those instructions that operate at the warp level, where threads within the same warp can write information directly to other threads' registers, the fastest memory available on the GPU as previously discussed in section 3.3. The use of warp primitives requires explicitly specifying a thread mask. Cooperative Groups provide a way to group threads, even of different blocks, in our case, we opted to use cooperative groups only at the warp level, as said before, to simplify the intra-warp communication. We evaluated the impact of multiple granularities, Figure 5.9 and Figure 5.10 show two examples. In these tests, we ran a kernel that counts the number of cooperative groups considered heavy, varying the cooperative group size from 8 to 32. As shown in the figures, the relationship between cooperative group size and the number of counted heavy groups is not always linear: looking at Figure 5.10, the number of heavy CGs grows in almost a linear way as the CG size decreases, from $\approx 20k$ to $\approx 103k$. Instead, if we look at Figure 5.9 the increase in heavy CGs when the CG size decreases from 32 to 8 is almost 10 times. As expected, increasing the number of sub-cooperative groups increases the chance that warp divergence will occur, for this reason, we kept the size of cooperative groups equal to 32.

This approach allows seamless switching between independent execution, in

which each thread processes a distinct edge, and cooperative execution, in which all 32 lanes of a warp collaborate on a single intersection that the threshold parameter classifies as heavy. Through warp-level primitives, such as register shuffle, the elected leader thread communicates the parameters of the heavy intersection to all remaining lanes instantaneously. The leader is selected via a `ballot` operation that identifies active threads with pending heavy work.

The leader also communicates which algorithm to use between Merge-based and Binary-Search.

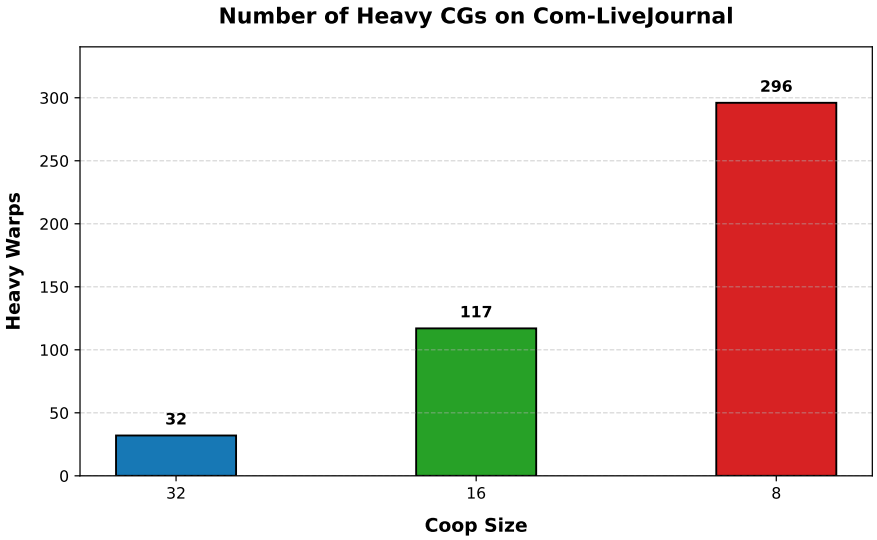


Figure 5.9: Number of cooperative groups flagged as heavy on Live-Journal

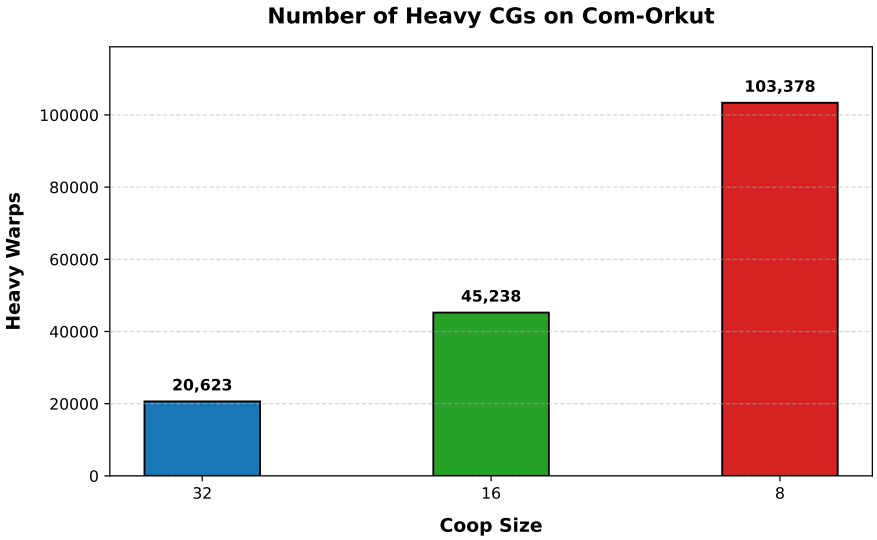


Figure 5.10: Number of cooperative groups flagged as heavy on Com-Orkut

5.6 Threshold Analysis

Through empirical analysis, we determined the threshold beyond which a warp transitions from independent processing to cooperative computation. When the estimated workload exceeds this value, all 32 lanes collaborate on a single heavy edge, distributing the intersection work evenly across threads and mitigating load imbalance on hub nodes with large adjacency lists.

The analysis shown in Figure 5.11 demonstrates that performance improves consistently with increasing threshold, stabilizing beyond 30,000. Block size has minimal impact across all tested configurations; however, block size 128 yields the best or equivalent performance in both the global geometric mean and on dense graphs such as com-orkut (Figure 5.12). We therefore select a block size of 128 and a threshold of 30,000 as the default configuration for all subsequent experiments.

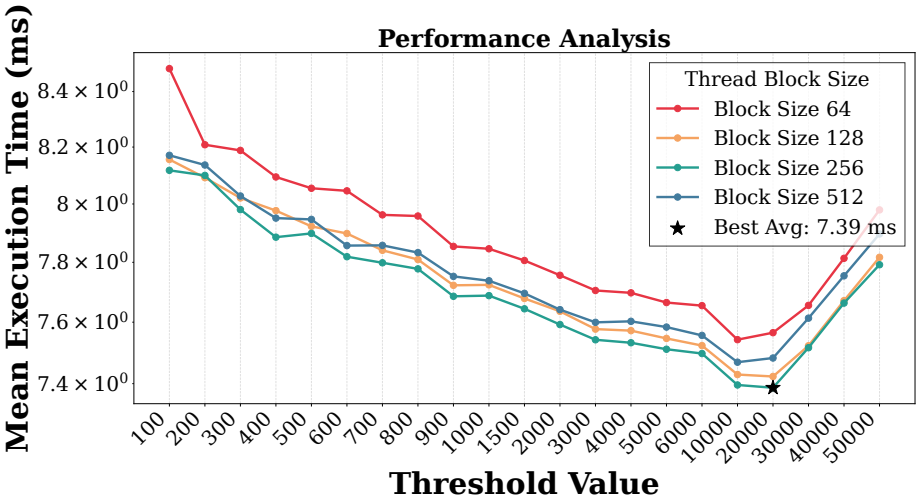


Figure 5.11: Performance evaluation on the whole dataset, the execution times are calculated using a geometric mean

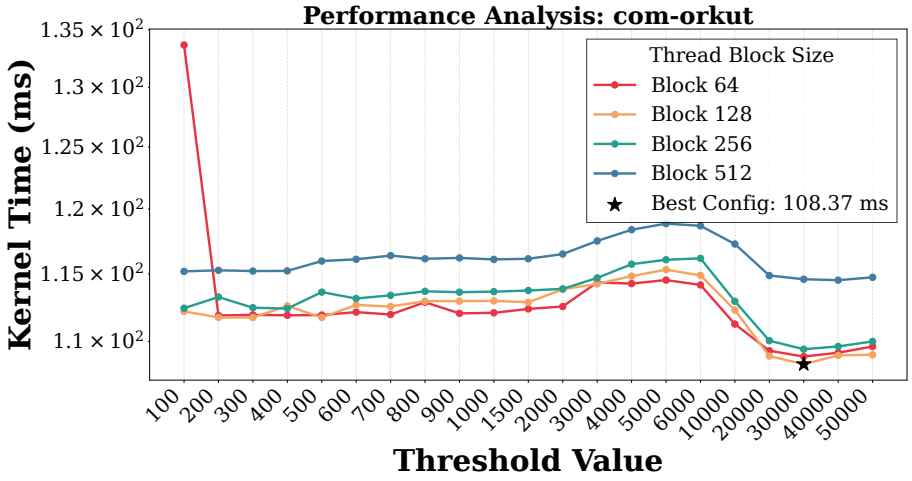


Figure 5.12: Performance evaluation on com-orkut

5.7 GetterTri Implementation

This section describes how the first version of **GetterTri** was implemented, tracing the design process that led to the final version, and providing a detailed explanation of the pseudocode reported in Algorithm 12. The idea behind this algorithm lies in intra-warp communication: within each warp, each thread is assigned to a single

intersection, and it computes the theoretical cost of its own intersection using both **Merge-based** and **Binary Search**, selecting the cheaper of the two accordingly. Once the cost is calculated, it is shared among the warp by a **shuffle** instruction. If the cost of the whole warp is above a certain threshold, the workload is not handled by single threads, but is instead extended to the entire warp, where threads, in turn, share their own information, such as the interested intersection and the method to use.

In order to develop this algorithm, we had to address these main obstacles:

- At which threshold should intra-warp communication be established?
- How to handle a workload when identified as heavy?
- How should the workload be distributed between the two search techniques?

The first question was answered in Section 5.6. The other two will be answered one at a time, justifying each design choice.

How to identify a heavy workload and how to handle it Initially the cost was calculated only at a local level, if a thread workload was below the threshold, then it will be flagged as light, otherwise it would be flagged as heavy. Under this logic, a thread with a light workload would first solve its own intersection independently, and only afterwards would it help the threads flagged as heavy, which, in turn, had to wait for all the light threads to finish their own task before the heavy workload's information could be shared and cooperatively processed. This approach improved performance on graphs with a larger number of hub nodes, but degraded performance on sparse graphs, where most threads carried a light workload and were therefore forced to wait in idle, before any cooperative work could begin. The next step was to revisit one of our earlier designs, the two kernel approach, but reformulate it so that it could be launched as a single kernel. Whenever the entire warp's workload is flagged as heavy, it would be processed and resolved immediately by all the threads, rather than being deferred until every light thread had completed its own work. This new design showed an almost negligible performance degradation on sparse graphs, due to the small overhead introduced by the additional checks, and a substantial performance improvement on hub-graphs. For this reason, we adopted this version as the basis of the final algorithm. Figure 5.13 and Figure 5.14 show an example of a graphic timeline of the first and the adopted version, respectively. In the first image, each thread works on its own intersection and then waits; the heavy threads stay unresolved until all the light threads have finished. Instead, in the second figure, the last discussed design is reported, where all the threads within a warp cooperate on the same intersection.

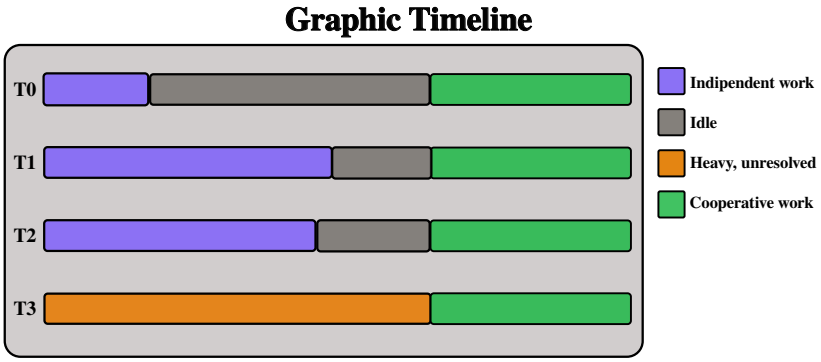


Figure 5.13: Graphic timeline showing the initial kernel design, where threads begin to cooperate only after the light-workload threads have finished their own work, resulting in wasted cycles.

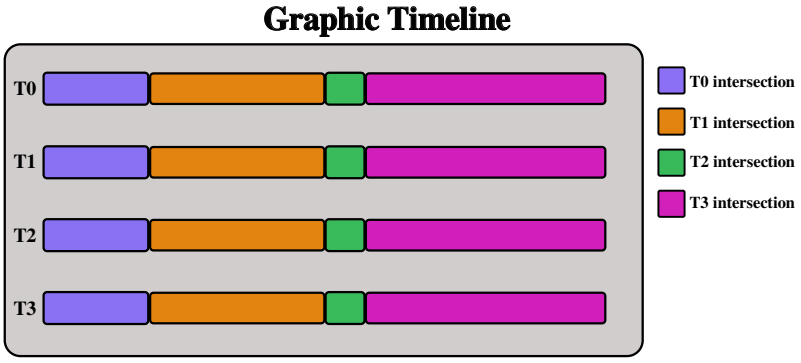


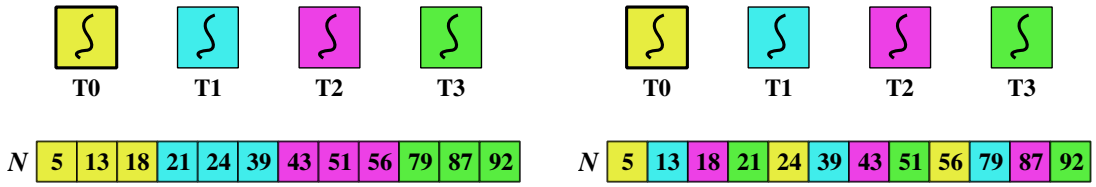
Figure 5.14: Graphic timeline showing the final kernel design, where all threads in the warp cooperate immediately on the heavy workload, eliminating the idle cycles observed in Figure 5.13.

How to partition workload between Merge-based and Binary-Search When a thread is elected as leader, it must share with the other threads in the warp the information needed to solve its assigned intersection, including the two vertices involved and the search method to be used. As discussed previously in section 5.4, we combine two search techniques, **Merge-based** and **Binary Search**. When the workload is flagged as heavy, the main challenge is how the work should be partitioned between the threads of the warp once one of these two techniques has been selected. We organized the partition as follows:

- **Merge-based**: the adjacency list is split into chunks, and each thread works on its own share.

- **Binary-Search**: each lane does not process consecutive elements but instead strides through the short list at intervals of 32, ensuring that all lanes of the warp cover the entire list without redundant work.

This partitioning ensures that the resulting workload remains balanced across all threads of the warp, regardless of which of the two techniques is used. In Figure 5.15, a graphical example of how data are partitioned is shown.



(a) Merge-based chunk division **(b)** Binary Search indices

Figure 5.15: Graphical representation of how data are handled based on the chosen search method.

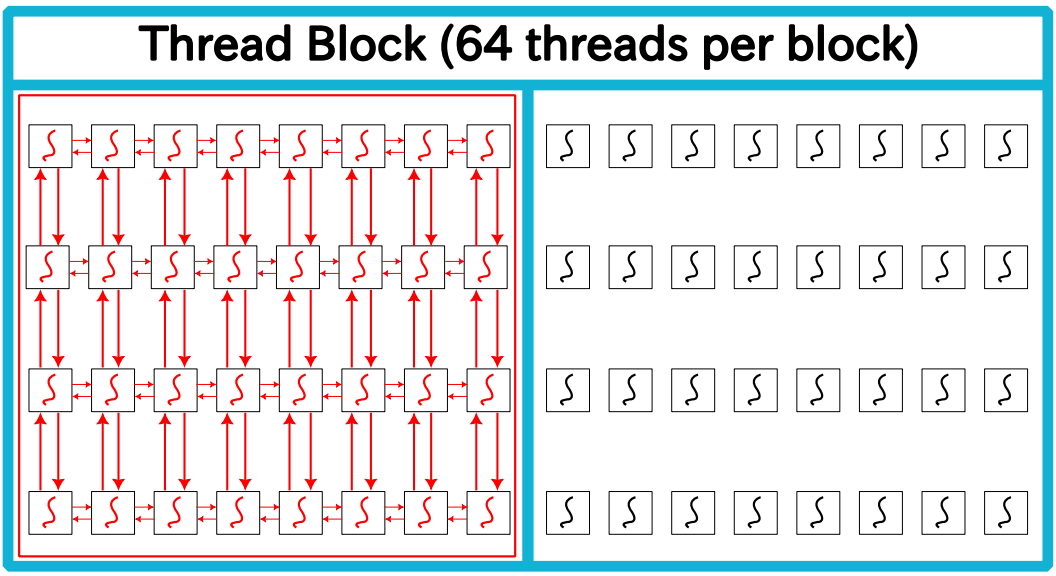
GetterTri PseudoCode In this paragraph we explain how the algorithm works in detail, Algorithm 12 will be used as reference, and the main sections will be described. The variable *triangles* is a global variable that is initialized to 0. In the CUDA implementation all the threads have a local counter that will be summed into the global variable only at the end, in order to avoid overhead; the sum is performed by means of atomic operations. The variables G and $Threshold$ are two constant global variables. For simplicity, we will refer to the graph using G, without imposing any data structure, keeping the code as neutral as possible. The variable warp, instead, exists only at the local level.

At the start of the algorithm, each thread is assigned to a specific intersection (u, v), and each thread initialize its own local variables to default values. Then, each thread will compute its personal intersection cost; in this step the algorithm to use between **Merge-based** and **Binary-Search** is also decided.

Once the intersection cost has been computed, all the threads within a warp will aggregate their own cost by means of a shuffle operation. If the estimated work of the whole warp is below a certain threshold, the local variable is_load_heavy is set to $false$, and each thread will proceed to resolve its own intersection alone; in the other case, in which the whole warp's estimated work is above the threshold, then is_load_heavy will be set to $true$ and the threads will start to cooperate in order to resolve the warp's intersection. From this moment, cooperative groups will be

used to make intra-warp communication easier. In order to start the cooperation, first of all the *ballot* function has to be called; *ballot* is a special CUDA function, that allows threads to **vote** : given 32 bits, this function allows threads to set the i-th bit to 1 or 0 based on the thread's rank, i.e., the thread with rank equals to 0 will modify the bit in position 0, the thread with rank equals to 1 will modify the bit in position 1 and so on. The order in which the thread's intersection will be processed is ID-based, starting from 0 to 31 in an incremental manner. At each new *while iteration*, a new *leader* is elected based on the current value of the mask; the *leader* will share the new intersection to resolve and the method to use, and which one of the two lists is the shorter one. Based on the chosen method, as explained in the previous paragraph, we will have two different data partitions: if **Merge-based** is used, data will be partitioned into chunks, otherwise threads will stride through the short list at intervals of 32. If new triangles have been found, the local counter will be updated. At the end of the *while iteration*, the mask will be updated with a new value, and once all the bits are set to 0, the while cycle will end. At this point, a shuffle operation will take place, aggregating all the local triangle counts, and after that, the thread with rank equal to 0, will perform an atomic operation to sum the warp's triangle value to the global variable $Triangles$, in order to save global memory transactions.

Figure 5.16 reports a visual representation of a possible CUDA block situation, in this case with a block size equal to 64, implying the existence of 2 warps, where the first one is flagged as **heavy** and threads will collaborate to resolve their tasks, instead the right one is flagged as **light**, and the threads inside this warp will work independently from each other.



(a) *Thread Block*

Figure 5.16: In this case, the block size is equal to 64. The threads in the first warp are working on a heavy task highlighted in red, and each thread, in turn, shares its own info, whereas in the second warp, each thread works independently of the others.

Algorithm 12 GetterTri Implementation.

Require: warp = CooperativeGroup(32), G(V,E), Threshold
    $triangles = 0$
    for (u,v) **do**
        $is_load_heavy = false$
        $len1 = len(u),\ len2 = len(v)$
        $num_op = calculate_operation(len1, len2)$
        $use_merge = is_merge_better(len1, len2)$
        $total_op = warp.shfl(num_op)$
        if $total_op > Threshold$ **then**
            $is_load_heavy = true$
        end if
        if $is_load_heavy = false$ **then**
            if use_merge **then**
                $triangles+ = MergePath(u, v)$
            else
                $triangles+ = BinSearch(u, v)$
            end if
        else
            $warp.has_work = warp.ballot(is_load_heavy)$
            while $warp.has_work$ **do**
                $leader = __ffs(warp.has_work) - 1$
                $\{lu, lv, s_use_merge\} = leader.Info()$
                if s_use_merge **then**
                $triangles+ = WarpMergePath(lu, lv)$
                else
                $triangles+ = WarpBinSearch(lu, lv)$
                end if
                $warp.hasWork\ \&=\ (1u << leader)$
            end while
        end if
    end for

Chapter 6

Results and Discussion

6.1 Experimental Environment

This section will cover the experimental environment used to evaluate GetterTri; after describing the machines used for both Enterprise and Consumer, we will go into implementation details.

Enterprise Hardware For the Enterprise Hardware (EHW) we conducted our experiments on a workstation equipped with an AMD EPYC 7T83 CPU (64 cores, 128 threads) and 64 GB of DDR4 RAM. The GPU implementation was evaluated on an NVIDIA A100 with 40 GB of HBM2 memory.

Consumer Hardware For the Consumer Hardware (CHW), we used an Intel Core i9-10900KF CPU (10 cores, 20 threads) and 64 GB of DDR4 RAM and an NVIDIA GeForce RTX 3070 with 8 GB of GDDR6 memory.

Implementation Details

All implementations were written in C++ and CUDA and compiled with GCC 9.4.0 and NVCC 13.1 using maximum optimization flags (e.g., `-O3`). Tests on Galois have been performed with 32 threads due to machine constraints.

GetterTri performs no heavy preprocessing on the input graph. The only transformation occurs during the file-reading phase, in which edges are selectively discarded if the destination vertex ID is greater than the source vertex ID. This approach ensures that each edge is processed in a unique direction, effectively orienting the graph with near-zero overhead.

We conducted our experiments on publicly available datasets, including various graph topologies with diverse scales in node and edge counts and density, primarily

from the Stanford Network Analysis Project (SNAP) dataset [27] and DIMACS [28]. The number of TEPS is calculated as $\frac{|E|_{undirected}}{T_{kernel}}$, where $|E|_{undirected}$ represents the number of unique undirected edges in the graph, and T_{kernel} is the execution time of the GPU kernel. In our evaluation, bi-directional edges are treated as a single connection. Excluded are reading time, time needed to copy data from the host to the device, and preprocessing time. All code and results will be made available on GitHub[1].

Framework	HW Target	Algorithm
TRUST	GPU	Hash-Based
TRICORE	GPU	Binary-Search
H-INDEX	GPU	Hash-Based
GUNROCK	GPU	Merge-Path
GALOIS	CPU	Merge-based (Default)

Table 6.1: Comparison of the frameworks used as baselines, by target hardware and intersection algorithm.

6.2 CPU vs GPU Performance

Before comparing **GetterTri** with the other frameworks, we will discuss the existing gap between the CPU-based and GPU-based algorithms. In detail, we used **Galois**[18], evaluated in multiple configurations: a single-thread configuration and a multi-threading configuration, alongside our GPU implementation, GetterTri. Figure 6.2 reports the execution time needed to count all the triangles in the dataset. As expected, the GPU implementation achieves an important speedup, Figure 6.1 reports the speedup obtained using the single-thread configuration as the baseline. Figure 6.3 shows that the higher the number of edges, the larger the gap, confirming that the extreme parallelism offered by GPUs is the dominant factor.

Power Consumption Even if the execution time is an important metric, other aspects must be considered, like power consumption. Looking at the raw number of cores, a GPU possesses hundreds of times more cores than a typical CPU, i.e., an NVIDIA A100 GPU has more than 6900 cores, whereas the AMD EPYC 7T83

[1] `https://github.com/Shuba19/GetterTri`

has 64 cores. But this high number of cores comes with a problem: **How much does a GPU consume?**

Using the already cited components as a reference, the GPU has a nominal power consumption of 400W, while the CPU has one of 280W. In our case, the CPU suffers from being blocked to a maximum of 32 threads, so we will take as nominal values a power consumption of $100W$ and $180W$ for the **1T** and **32T** configurations, respectively, since the consumed power does not scale in a linear manner with respect to the maximum nominal power, instead, there is a fixed baseline consumption, to which additional consumption is added as more cores are used intensively. In order to calculate the consumed energy, we multiply the nominal power by the time spent counting the triangles, obtaining:

- **1T** : $E = 100W \times 118.754s = 11875.4J$
- **32T** : $E = 180W \times 32.609s = 5869.62J$
- **GPU** : $E = 400W \times 0.74052s = 296.208J$

Where J denotes the **Joule**, and is used to measure energy and work. Based on previous calculations, even assuming the GPU constantly draws its maximum nominal power, which is higher than the CPU's, its estimated energy consumption remains lower than that of an optimized CPU implementation.

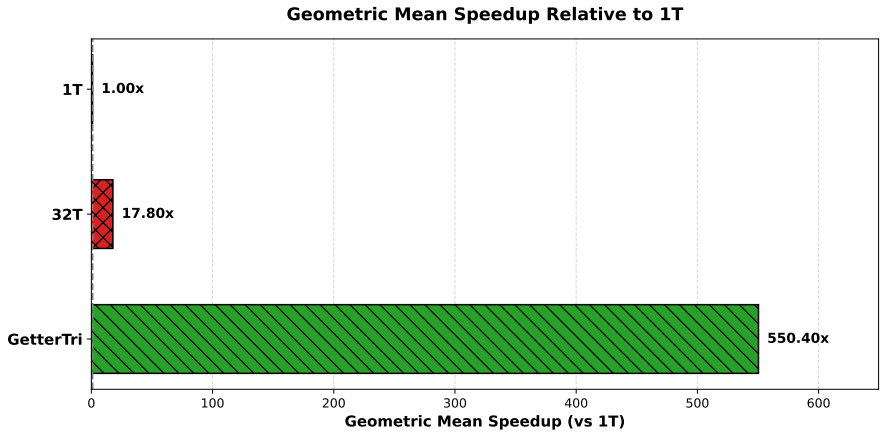


Figure 6.1: Histogram that represents the speedup achieved by other implementations using the single-thread configuration as baseline.

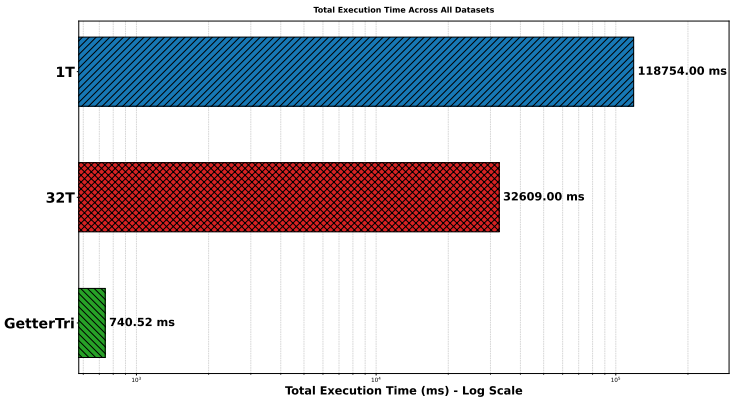
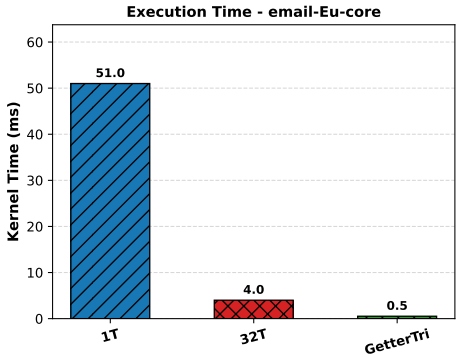


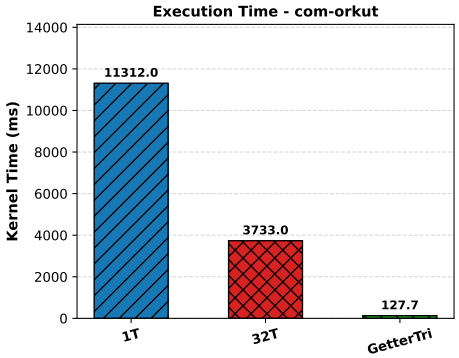
Figure 6.2: Total execution time



(a) email-Eu-core



(b) com-youtube



(c) com-orkut

Figure 6.3: Execution time comparison across different execution configurations. The graphs are ordered based on the number of edges, from (a) the lower to (c) the highest.

6.3 Profiling Results

This section covers the metrics from the reports generated by the NVIDIA Nsight Compute profiling tool. In section 3.4.2 we already discussed how the tool works and its related metrics, but for the reader's convenience, a summary of the relevant metrics will be provided at the beginning of each paragraph. The value reported in this section originated from a profiling session on Consumer Hardware.

Roofline model and SOL The Roofline model is a performance model that can be used to bound floating-point performance as a function of machine peak performance, machine peak bandwidth and arithmetic intensity of the application. Figure 6.4 reports an example of the Roofline model resulting from a GetterTri profiling session.

SOL stands for **Speed of Light** and is a high-level overview of the throughput for the GPU's resources. The SOL analysis reports the percentage of peak compute and memory throughput achieved by a kernel. In Table 6.2, the most relevant metrics collected during a representative profiling session are reported, while Table 6.3 reports the corresponding compute workload metrics. The Roofline model shows that the kernel is memory-bound, but since the kernel performs integer operations and this model analyzes floating-point operations, other values should be considered, such as **Compute Throughput** and **Memory Throughput**: these values show that the kernel is compute-bound.

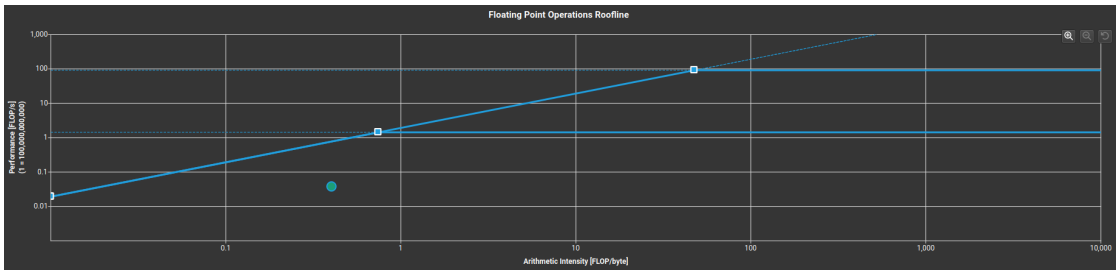


Figure 6.4: Roofline model obtained from a representative GetterTri profiling session.

The kernel achieves a Compute Throughput of 67.03%, which, together with **SM Busy**, **Issue Slots Busy**, and **Executed IPC Active** values, indicates that when a warp is scheduled, the pipeline of the Streaming Multiprocessor is kept busy.

Memory Workload Analysis and Thread Occupancy The **Memory Workload Analysis** report shows how effectively memory is utilized, reporting both

Metrics	Value
Duration (ms)	3.27
Compute (SM) Throughput (%)	67.03
Memory Throughput (%)	41.23
L1/TEX Cache Throughput (%)	44.58
L2 Cache Throughput (%)	18.91
DRAM Throughput (%)	4.71

Table 6.2: GPU Speed of Light Throughput metrics.

Metrics	Value
Executed IPC Elapsed (inst/cycle)	2.68
Executed IPC Active (inst/cycle)	2.90
Issued IPC Active (inst/cycle)	2.90
SM Busy (%)	67.03
Issue Slots Busy (%)	67.03

Table 6.3: Compute Workload Analysis metrics.

L1 and L2 cache hit rates. **Thread Occupancy** instead measures the ratio between the number of warps actively resident on a Streaming Multiprocessor (SM), reporting a comparison with the theoretical maximum supported by the device and showing how well the GPU's resources are being used. Table 6.4 presents various metrics regarding the memory workload. DRAM throughput represents only a fraction of the theoretical maximum, whereas L1 and L2 cache hit rates are high, reaching 87.38% and 93.19% respectively. These values indicate that the majority of accesses performed by the threads take place directly on-chip, rather than requiring access to global DRAM memory.

Table 6.5, on the other hand, presents warp statistics. The average number of active threads per warp is 7.99 out of a maximum of 32—approximately 25%. This value indicates the presence of warp divergence due to the intrinsic nature of GetterTri. Meanwhile, the **Long Scoreboard Stalls** value indicates that a significant portion of threads wait an average of 4 cycles per instruction due to data dependencies.

Finally, Table 6.6 compares theoretical versus achieved occupancy—showing a drop from a theoretical 100% to 77.62% and indicating that the GetterTri design causes warp divergence. However, in theory, switching to a single search algorithm would be advantageous and would reduce the presence of warp divergence, but our tests have shown us that in reality our hybrid approach turns out to be efficient. Taking the **com-orkut** graph as an example and the classic GetterTri design as a baseline, we obtain a speedup of 1.18× and 1.11× compared to using only Binary Search or only Merge-based as search algorithms, respectively.

Metrics	**Value**
Memory Throughput (GB/s)	9.04
L1/TEX Hit Rate (%)	87.38
L2 Hit Rate (%)	93.19
Mem Busy (%)	33.76
Max Bandwidth (%)	41.23
Mem Pipes Busy (%)	41.23

Table 6.4: Memory Workload Analysis metrics.

6.4 Comparison with State of the Art

This section will cover the main differences between GetterTri and the state-of-the-art, first outlining the reasons for selecting the frameworks listed in Table 6.1, highlighting similarities and differences, before presenting the test results.

Metrics	Value
Warp Cycles Per Issued Instruction	12.82
Warp Cycles Per Executed Instruction	12.83
Avg. Active Threads Per Warp	7.99
Avg. Not Predicated Off Threads Per Warp	7.48
Long Scoreboard Stalls (cycles/inst)	4.00

Table 6.5: Warp State Statistics metrics.

Metrics	Value
Theoretical Occupancy (%)	100.00
Theoretical Active Warps Per SM	48.00
Achieved Occupancy (%)	77.62
Achieved Active Warps Per SM	37.26

Table 6.6: Occupancy metrics from Nsight Compute.

Frameworks The frameworks selected were Trust [14], Tricore [15], H-index [16], Gunrock [17], and Galois [18]. **Trust** was chosen due to its significance in the literature; **Tricore** was selected for its similar approach, specifically a warp level binary search, and along with **H-Index**, for having been a Graph Challenge Champion. Finally, we decided to include two general-purpose graph analytics frameworks, one for GPUs and one for CPUs: **Gunrock** and **Galois**, respectively.

6.4.1 Performance Comparison

Tests were conducted on a dataset comprising 91 graphs; GetterTri achieved an average speedup of 1.77× over Trust, 3.31× over Tricore, 1.96× over H-index, 8.02× over Gunrock, and 31.40× over Galois on Enterprise Hardware, while on Consumer Hardware, the corresponding speedups are 2.42×, 7.97×, 2.73×, 11.00×, and 7.24×. These results demonstrate that GetterTri outperforms both dedicated GPU algorithms and GPGPU frameworks. The average was calculated using the geometric mean to ensure a balanced result unaffected by the presence of potential outliers.

Memory Footprints In addition to achieving a speedup in terms of kernel time, GetterTri also demonstrates more efficient memory usage. GetterTri does not allocate any additional auxiliary structures, relying instead solely on the basic CSR structure described in Section 2.1.4.

Across all tests, GetterTri achieved an average memory usage of 153.18 MB, followed by Gunrock (326.27 MB), Trust (466.28 MB), Tricore (674.74 MB), Galois (703.14 MB), and H-index (4113.93 MB).

Comparison with General Purpose Framework Both **Gunrock** and **Galois** were designed as general purpose graph analytics frameworks, that is, frameworks that offer a set of primitives for computing specific features, including triangle counting. Overall, GetterTri performs very well compared to these two frameworks, achieving significant speedups on both consumer and enterprise hardware; it reaches peak speedups of approximately 382× and 341× against Gunrock, and approximately 50× and 75× against Galois. It is worth noting that Galois, unlike Gunrock, exhibits more stable execution times.

Graphs where GetterTri does not achieve the best performance In several cases, however, GetterTri fails to achieve the best performance compared to other frameworks. One example is the frequently cited **com-orkut**, where TRUST achieves a speedup of 0.87× on CHW and 0.59× on EHW (values less than 1 indicate that the competitor is faster). A similar example is **com-LiveJournal**, where GetterTri achieves a speedup of 1.05× on Consumer Hardware, but is

outperformed on Enterprise Hardware. These tests demonstrate that GetterTri struggles with graphs containing a large number of hub nodes.

In the tests run on Enterprise Hardware, H-Index outperforms GetterTri on a series of graphs, specifically `p2p-Gnutella04`, `soc-Epinions1`, and `soc-Slashdot`, which share the characteristics of being small and sparse, achieving speedups of 0.51×, 0.52×, and 0.66×, respectively. A common aspect of those graphs is that the execution time is extremely low (under one millisecond). In such cases, the overhead associated with the initial computation and inter-thread information exchange is no longer negligible. Another example, again using the CHW configuration, involves the **wiki-topcats** graph, where GetterTri is outperformed by all three of its competitors.

Those results indicate that, overall, GetterTri stands out as a solution that, thanks to its intra-warp communication, delivers excellent results in most scenarios, however, it encounters some issues with the type of graphs previously mentioned, specifically those where kernel execution time is so low that the overhead impacts the final results, as well as graphs with a high number of hub nodes.

6.5 Enterprise vs Consumer Hardware Results

This section presents a subset of the experiments conducted, detailing approximately twenty tests performed on both enterprise and consumer hardware. Table 6.8 summarizes the previously mentioned data, specifically the average speedup and average memory usage for both consumer and enterprise hardware.

Each framework was evaluated under the following conditions:

- A warm-up run was performed for each test.
- Each test was executed once.
- Triangle correctness was verified through direct comparison across the different frameworks.
- The latest available version of the algorithms was used.
- Memory usage was measured differently depending on the framework: for Gunrock, an estimate was calculated; Galois reports its own memory usage; while for TriCore, TRUST, and H-Index, memory consumption was calculated directly via the code.
- All evaluations were performed on the exact same dataset.

Table 6.9 reports end-to-end results on six graphs, selected to span a range of sizes, densities, and topologies. This measured time covers the entire program

execution, from start to finish, including file reading, H2D and D2H transfers, preprocessing, and kernel time. It should be noted that TRUST and TriCore require a binary file as input, generated beforehand from an edge list; this allows the graph to be loaded directly into memory without any parsing step. GetterTri, on the other hand, reads from a plain-text METIS file, which must be parsed at load time. The time required to convert the edge list into the binary format is not included in the reported figures.

In light of this asymmetry, the results reported in the table are even more significant: despite the substantial difference in **read time**, deriving from the fact that TRUST and TriCore operate on an already-parsed file while GetterTri must parse its input from scratch, GetterTri still achieves a considerable speedup. For instance, on **com-LiveJournal**, GetterTri attains a speedup of 9.13× over TRUST and 1.75× over TriCore. An even more notable case is **thermal2**, where the speedup over TRUST grows from 2.14×, measured at the kernel-time level in Table 6.11, to 12.71× once the full end-to-end cost is accounted for, and the speedup over TriCore grows from 3.22× to 6.50×.

Despite the theoretically higher computational power of the enterprise hardware, the graph in Figure 6.5 shows that, in reality, consumer hardware achieves a lower average execution time; Table 6.7 lists the specifications for both devices. These findings suggest that in cases involving sparse graphs, consumer hardware devices can outperform enterprise ones.

For instance, with the **email-Enron** and **p2p-Gnutella04** graphs, the RTX 3070 achieves speedups of 3.63× and 3.35×, respectively, compared to the A100. This behavior may partly result from the higher clock frequency of the RTX 3070, although launch overhead and architectural differences may also contribute. However, as graph size and the number of edges to be processed increase, the A100 gains an advantage over the 3070. This is due to its higher number of Streaming Multiprocessors, 108 versus the 3070's 48, as well as the larger size of its L1 and L2 caches; a case worth mentioning is the **kron_g500-simple-logn20** graph, where the A100 achieves a 6.97× speedup over the 3070.

It should also be mentioned that available memory is another critical factor: with only 8 GB of GDDR6 memory, the RTX 3070 imposes a limit on the maximum size of processable graphs, a constraint absent in the A100, which possesses 40 GB of memory. Consequently, consumer hardware is a viable choice primarily for applications involving smaller graphs, whereas enterprise hardware remains the better choice for processing large-scale graphs.

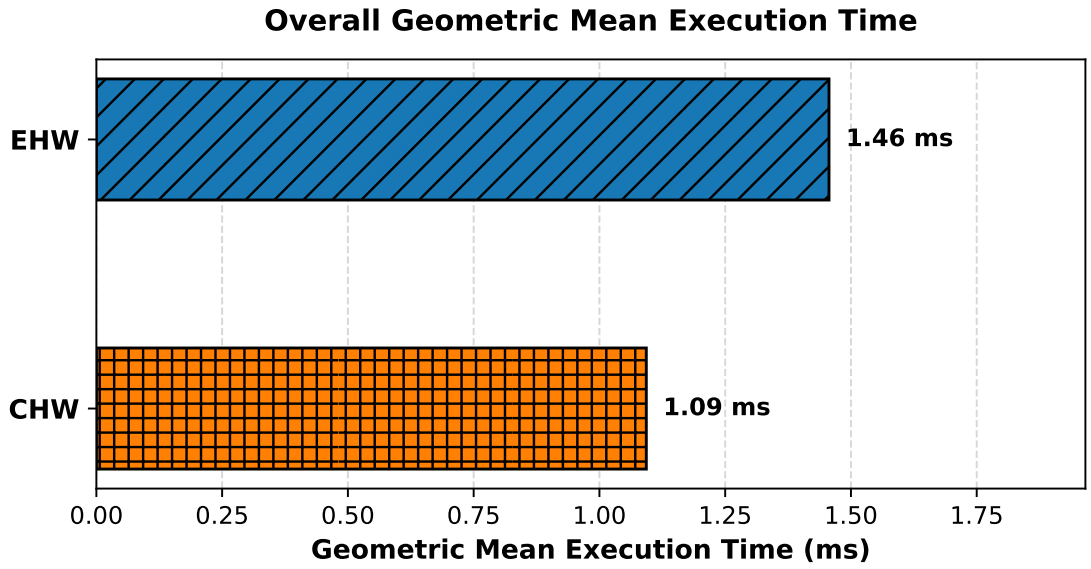


Figure 6.5: Histogram that reports the average kernel time measured on Enterprise Hardware (EHW) and on Consumer Hardware (CHW).

Specification	NVIDIA RTX 3070	NVIDIA A100
Streaming Multiprocessors (SM)	48	108
CUDA Cores	5,888	6,912
Base Clock	1.50 GHz	1.10 GHz
Boost Clock	1.73 GHz	1.41 GHz
L1 Cache / Shared Mem. (per SM)	128 KB	192 KB
L1 Cache / Shared Mem. (Total)	5.75 MB	20.73 MB
L2 Cache (Shared)	4.00 MB	40.00 MB
Memory Type & Size	GDDR6 (8 GB)	HBM2 (40 GB)
Memory Bandwidth	448.0 GB/s	1,555.0 GB/s
TDP (Power Consumption)	250 W	400 W

Table 6.7: Hardware specifications comparison between RTX 3070 (Consumer Hardware) and A100 SXM4 40GB (Enterprise Hardware).

Metrics	GetterTri	TRUST	TRICORE	H-INDEX	GUNROCK	GALOIS
Average Memory Used (MB)	153.18	466.28	674.74	4113.93	326.27	703.14
Average Speedup EHW (GT vs)	1.00x	1.77x	3.31x	1.96x	8.02x	31.40x
Average Speedup CHW (GT vs)	1.00x	2.42x	7.97x	2.73x	11.00x	7.24x

Table 6.8: Values calculated on a total of 91 graphs

Graph	GetterTri E2E (ms)	Trust Pproc (ms)	Trust E2E (ms)	Trust Speedup	Tricore Pproc (ms)	Tricore E2E (ms)	Tricore Speedup
com-orkut	2791.31	16226	17239.85	6.18	1069	3589.00	1.29
com-LiveJournal	767.15	6502	7000.83	9.13	388	1344.00	1.75
kron_g500-simple-logn20	1600.19	2822	3532.99	2.21	360	1592.00	0.99
cit-Patents	435.62	4370	4762.03	10.93	199	847.00	1.94
nlpkkt240	8586.98	28239	30722.10	3.58	3096	10708.00	1.25
thermal2	105.03	985	1334.62	12.71	178	683	6.50

Table 6.9: End-to-end performance comparison across selected graphs. Pproc stands for preprocess, E2E for end-to-end.

Dataset	GetterTri		TRUST		TRICORE		H-INDEX		GUNROCK		GALOIS	
	(ms)	TEPS	(ms)	Speedup	(ms)	Speedup	(ms)	Speedup	(ms)	Speedup	(ms)	Speedup
adaptive	0.77	17.61B	7.95	10.27x	16.25	21.00x	11.93	15.42x	2.54	3.28x	12.38	16.00x
af_shell9	4.21	2.03B	4.39	1.04x	9.94	2.36x	8.53	2.03x	5.75	1.37x	14.00	3.33x
audikw1	34.12	1.12B	22.75	0.67x	44.96	1.32x	47.22	1.38x	633.22	18.56x	190.19	5.57x
cit-Patents	6.22	2.66B	8.57	1.38x	21.55	3.47x	16.19	2.60x	59.34	9.54x	104.00	16.73x
com-amazon	0.36	2.54B	0.86	2.36x	3.60	9.89x	0.94	2.57x	1.03	2.82x	6.88	18.90x
com-LiveJournal	26.65	1.30B	27.98	1.05x	41.80	1.57x	47.02	1.76x	937.89	35.19x	388.07	14.56x
com-orkut	214.91	0.55B	187.96	0.87x	237.45	1.10x	380.77	1.77x	15006.00	69.83x	2850.81	13.27x
GAP-road	1.43	20.24B	13.22	9.27x	25.02	17.55x	24.31	17.05x	5.48	3.84x	34.68	24.33x
loc-brightkite_edges	0.25	0.85B	0.47	1.88x	3.04	12.05x	0.34	1.33x	5.12	20.28x	5.21	20.63x
nlpkkt200	5.72	37.78B	28.70	5.02x	120.64	21.10x	185.06	32.37x	60.98	10.67x	131.98	23.08x
p2p-Gnutella04	0.18	0.22B	0.29	1.61x	2.65	14.78x	0.19	1.04x	0.27	1.52x	8.88	49.54x
p2p-Gnutella31	0.20	0.72B	0.32	1.57x	2.71	13.25x	0.26	1.27x	0.29	1.40x	4.78	23.33x
roadNet-CA	0.28	9.94B	2.59	9.31x	4.59	16.51x	2.74	9.83x	0.90	3.24x	2.83	10.18x
roadNet-PA	0.24	6.36B	2.35	9.70x	3.69	15.23x	1.65	6.82x	0.69	2.85x	1.51	6.24x
roadNet-TX	0.25	7.62B	2.20	8.73x	3.96	15.71x	2.06	8.17x	0.77	3.06x	1.91	7.59x
road_usa	1.42	20.34B	15.20	10.71x	29.38	20.70x	24.26	17.10x	5.48	3.86x	34.77	24.50x
soc-Epinions1	0.58	0.70B	0.77	1.32x	4.44	7.63x	0.63	1.08x	34.16	58.66x	3.82	6.55x
soc-LiveJournal1	38.92	1.10B	38.44	0.99x	53.62	1.38x	70.50	1.81x	1722.58	44.26x	533.70	13.71x
soc-Slashdot0902	0.54	0.93B	0.72	1.32x	3.00	5.53x	0.66	1.21x	17.64	32.48x	3.26	6.00x
thermal2	0.40	9.23B	1.96	4.93x	5.73	14.40x	3.13	7.85x	1.03	2.60x	5.08	12.76x
wiki-Talk	3.46	1.34B	5.35	1.54x	8.36	2.41x	6.39	1.84x	470.67	135.85x	47.74	13.78x
wiki-topcats	48.48	0.52B	20.27	0.42x	29.57	0.61x	27.01	0.56x	18513.20	381.91x	237.78	4.91x

Table 6.10: Performance analysis on Consumer Hardware: execution time (ms) and throughput (TEPS) for GetterTri, and relative execution time (ms) and speedup for the other frameworks.

Dataset	**GetterTri**		**TRUST [14]**		**TRICORE [15]**		**H-INDEX[16]**		**GUNROCK[17]**		**GALOIS**[18]	
	(ms)	TEPS	(ms)	Speedup	(ms)	Speedup	(ms)	Speedup	(ms)	Speedup	(ms)	Speedup
adaptive	1.19	11.43B	6.80	5.70x	8.07	6.77x	10.02	8.41x	6.39	5.36x	16.00	13.43x
af_shell9	1.20	7.12B	2.21	1.84x	5.26	4.38x	6.61	5.51x	10.82	9.02x	13.00	10.83x
Amazon0302	0.60	1.50B	1.33	2.22x	1.92	3.19x	0.70	1.16x	3.76	6.26x	45.00	74.90x
audikw1	8.57	4.48B	10.85	1.27x	29.35	3.43x	37.71	4.40x	98.33	11.48x	246.00	28.72x
cit-Patents	2.75	6.01B	5.05	1.84x	9.93	3.61x	13.33	4.85x	22.07	8.02x	132.00	48.00x
com-amazon	0.61	1.52B	1.09	1.78x	1.38	2.27x	0.81	1.33x	1.28	2.09x	40.00	65.63x
com-LiveJournal	13.23	2.62B	12.53	0.95x	20.66	1.56x	34.01	2.57x	376.56	28.47x	505.00	38.18x
com-orkut	127.66	0.92B	75.31	0.59x	105.48	0.83x	248.90	1.95x	6730.07	52.72x	3733.00	29.24x
com-youtube	1.60	1.87B	1.67	1.05x	3.00	1.88x	2.56	1.61x	545.15	341.55x	39.00	24.43x
GAP-road	2.10	13.76B	12.33	5.88x	11.74	5.60x	19.40	9.25x	8.62	4.11x	49.00	23.36x
loc-brightkite_edges	0.51	0.42B	1.32	2.58x	77.63	151.43x	0.39	0.75x	14.77	28.82x	36.00	70.22x
nlpkkt200	10.52	20.54B	15.50	1.47x	46.78	4.45x	122.72	11.67x	22.66	2.15x	344.00	32.71x
p2p-Gnutella04	0.60	0.07B	0.68	1.13x	1.26	2.10x	0.31	0.51x	0.74	1.23x	31.00	51.61x
p2p-Gnutella31	0.53	0.28B	0.67	1.26x	1.27	2.41x	0.35	0.66x	0.79	1.49x	37.00	69.90x
roadNet-CA	0.66	4.20B	1.73	2.63x	2.30	3.49x	2.07	3.15x	2.21	3.35x	40.00	60.71x
roadNet-PA	0.56	2.76B	1.46	2.60x	1.95	3.48x	1.25	2.24x	1.95	3.48x	42.00	75.06x
road_usa	2.09	13.84B	12.34	5.92x	13.05	6.26x	19.30	9.25x	8.34	4.00x	52.00	24.93x
soc-Epinions1	1.06	0.38B	1.35	1.28x	1.66	1.57x	0.55	0.52x	56.77	53.69x	5.00	4.73x
soc-LiveJournal1	20.26	2.11B	16.83	0.83x	28.95	1.43x	46.26	2.28x	766.03	37.81x	687.00	33.91x
soc-Slashdot0902	0.88	0.57B	1.55	1.76x	1.37	1.56x	0.58	0.66x	21.39	24.33x	5.00	5.69x
thermal2	0.89	4.14B	1.90	2.14x	2.86	3.22x	2.81	3.17x	2.78	3.13x	7.00	7.88x
wiki-Talk	2.26	2.06B	2.66	1.17x	4.01	1.77x	5.02	2.22x	456.88	201.78x	71.00	31.36x

Table 6.11: Performance analysis on Enterprise Hardware: execution time (ms) and throughput (TEPS) for GetterTri, and relative execution time (ms) and speedup for the other frameworks.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

This thesis has examined the problem of triangle counting in graphs. Although conceptually simple, triangle counting involves several challenges, including the best search strategy, how and when to perform preprocessing and how to manage the possible presence of hub nodes.

Starting from the basics of graph theory, examining the terminology used to describe a graph and its properties and the various data structures used for graph analysis, and then describing the research algorithms used and the various distinctions of general approach, the differences between the hardware components of consumer and enterprise devices were described, and then finally analyzed the state of the art with the various methodologies used.

This thesis addresses these issues by proposing GetterTri, a GPU-based algorithm for counting triangles based on a dynamic estimated cost mechanism calculated at runtime. Unlike the state of the art, where for each node or intersection an entire warp is assigned, GetterTri associates each individual warp thread with a single intersection, and in the case that the estimated workload exceeds a certain threshold, which has been empirically chosen, the threads will begin to cooperate to resolve intersections. This mechanism allows you to dynamically choose the best strategy between the two based solely on the estimate of the work, avoiding costly preventive preprocessing or node classification operations.

A second contribution of this thesis concerns the research technique used, the combination of the two techniques of Merge-based and Binary Search, where the threads individually, will choose the most appropriate search method based on the length of the two adjacency lists.

A third contribution instead is the implementation of a preprocessing performed during the reading of the file, allowing a cost-free preprocessing, achieving a low

use of memory compared to the other frameworks.

GetterTri was tested on a dataset of 91 graphs, each of which differs in the number of nodes, edges, and density, confirming the effectiveness of the proposed approach. It has also been tested on two different hardware architectures. On Enterprise hardware, GetterTri obtains an average speedup of 1.77x, 3.31x, 1.96x, 8.02x, and 31.40x compared to Trust, Tricore, H-index, Gunrock, and Galois, respectively; on Consumer hardware, the average speedups are 2.42x, 7.97x, 2.73x, 11.00x, and 7.24x.

In the face of these results, however, it must be said that GetterTri does not consistently achieve the best performance across all graphs; in fact on graphs with highly skewed distributions and a large number of high-degree vertices, frameworks such as TRUST are more efficient, while for extremely small graphs, the overhead of estimation calculation affects the performance.

In addition, a comparison was also made between Enterprise and Consumer hardware, which highlighted several strengths for both categories: while it is true that enterprise hardware turns out to be more efficient in computing large graphs due to the high number of SM and the larger size of the L1 and L2 caches, consumer hardware, instead having a higher clock frequency, offers greater performance in smaller graphs.

7.2 Future Work

With regard to future developments, different directions can be considered for the development of GetterTri.

First, the extension of GetterTri to a multi-GPU configuration, representing a natural evolution for handling graphs whose size may exceed the memory available on a single device, through the study and implementation of new graph partitioning techniques that are as lightweight as possible while delivering the best possible results.

Another development involves revisiting the balancing mechanism via cooperative groups, moving beyond the standard 32-thread granularity to implement finer-grained or coarser-grained solutions at both the warp and block levels, respectively, while establishing new threshold levels adapted to the required granularity.

As a last development, a completely different path could be considered: exploring the SpGEMM techniques mentioned above combined with Tensor Cores along with Compact Matrix Format techniques.

List of Figures

2.1 Transformation from an undirected graph (a) to a DAG (b). In this case, the rule applied is based on the vertex ID, more precisely $id(u) < id(v)$. 5
2.2 Partitioning of the original graph(a) into two different graphs, cutting the edge from 1 to 5. 5
2.3 Example of Full Mesh graph 6
2.4 Representation of a graph with an adjacency matrix. 7
2.5 Graph representation using the Compressed Sparse Row format . . 8
2.6 Given the graphs G represented in Figures (a), Figures (b), and (c) report two different admissible triangles. 8

3.1 Visual representation of a complete rasterization pipeline 12
3.2 Comparison of CPU and GPU architectures 13
3.3 Interaction between CPU and GPU, source Nvidia 14
3.4 Simple example kernel in CUDA. 14
3.5 Visual representation of how grids and blocks work in CUDA (image source : Nvidia CUDA) 15
3.6 Kernel Launch with a total of 20 threads. 15
3.7 Example of kernel divergence 16
3.8 Graphic representation of how the different types of memory are accessed by the device and by the host 17
3.9 Example of an NVIDIA Nsight Systems Report 18
3.10 Nvidia Nsight Compute initial window; all the kernels with some basic information are listed 18
3.11 Screenshots of NCU's Occupancy section 19
3.12 These graphs show how the occupancy can vary based on different factors, Register Per Thread, Block Size, and Shared Memory Per Block, respectively, from top to bottom. 19
3.13 Memory Workload Analysis 19
3.14 Memory Chart, graphic summary of the memory Hit Rate and Throughput 20

3.15 Example of Roofline Model, in the x-axis is measured the Operational Intensity, in the y-axis the Throughput 21
3.16 Example of optimization using the roofline model. 22
3.17 Hardware differences between DDR and HBM memories 23
3.18 Ray-tracing example 24
3.19 Visual representation of how Tensor Cores work 24

4.1 Simple explanation of sparse matrix multiplication. Given two matrices, A and B, C is the result of the multiplication between A and B. 26
4.2 Transition from a complete symmetric matrix to a lower triangular matrix. 27
4.3 Merge-Based 29
4.4 Binary Search 30
4.5 Hash Bucket 31
4.6 CPU triangle counting performance comparison, Single-thread vs Multi-thread configuration 32
4.7 An UpDown Node 33

5.1 Performance comparison between Edge and Node iterators 36
5.2 Starting Graph 37
5.3 Starting graph after ID-based preprocessing 37
5.4 Starting graph after degree-based preprocessing 38
5.5 Starting graph after relabel-based preprocessing 39
5.6 Starting graph after GT-based preprocessing. It differs from Figure 5.2 in the applied rule: instead of the classical $id(v) > id(u)$, the rule $id(u) > id(v)$ is used. 40
5.7 Comparison between the classic ID-based orientation ($id(v) > id(u)$) and the orientation adopted by GetterTri ($id(u) > id(v)$). 41
5.8 Performance comparison between Node Iterator, Edge Iterator and GetterTri 43
5.9 Number of cooperative groups flagged as heavy on Live-Journal . . 44
5.10 Number of cooperative groups flagged as heavy on Com-Orkut . . . 45
5.11 Performance evaluation on the whole dataset, the execution times are calculated using a geometric mean 46
5.12 Performance evaluation on com-orkut 46
5.13 Graphic timeline showing the initial kernel design, where threads begin to cooperate only after the light-workload threads have finished their own work, resulting in wasted cycles. 48

5.14 Graphic timeline showing the final kernel design, where all threads in the warp cooperate immediately on the heavy workload, eliminating the idle cycles observed in Figure 5.13. 48
5.15 Graphical representation of how data are handled based on the chosen search method. 49
5.16 In this case, the block size is equal to 64. The threads in the first warp are working on a heavy task highlighted in red, and each thread, in turn, shares its own info, whereas in the second warp, each thread works independently of the others. 51

6.1 Histogram that represents the speedup achieved by other implementations using the single-thread configuration as baseline. 55
6.2 Total execution time 56
6.3 Execution time comparison across different execution configurations. The graphs are ordered based on the number of edges, from (a) the lower to (c) the highest. 56
6.4 Roofline model obtained from a representative GetterTri profiling session. 57
6.5 Histogram that reports the average kernel time measured on Enterprise Hardware (EHW) and on Consumer Hardware (CHW). 64

List of Tables

5.1 Breakdown of preprocessing times (in milliseconds) for different graphs. The **Read** column reports the time required to read and parse the input file alone. **ID**, **Degree**, and **Relabel** report the time required by each orientation strategy on top of the parsed graph. **GT** reports the difference between the classic file-reading time and the time required by GetterTri combined parsing-and-filtering approach; negative values indicate a time reduction. 41

6.1 Comparison of the frameworks used as baselines, by target hardware and intersection algorithm. 54
6.2 GPU Speed of Light Throughput metrics. 58
6.3 Compute Workload Analysis metrics. 58
6.4 Memory Workload Analysis metrics. 59
6.5 Warp State Statistics metrics. 60
6.6 Occupancy metrics from Nsight Compute. 60
6.7 Hardware specifications comparison between RTX 3070 (Consumer Hardware) and A100 SXM4 40GB (Enterprise Hardware). 64
6.8 Values calculated on a total of 91 graphs 64
6.9 End-to-end performance comparison across selected graphs. Pproc stands for preprocess, E2E for end-to-end. 65
6.10 Performance analysis on Consumer Hardware: execution time (ms) and throughput (TEPS) for GetterTri, and relative execution time (ms) and speedup for the other frameworks. 65
6.11 Performance analysis on Enterprise Hardware: execution time (ms) and throughput (TEPS) for GetterTri, and relative execution time (ms) and speedup for the other frameworks. 66

Acknowledgements

Ringrazio innanzitutto i miei relatori, Stefano e Lorenzo, per avermi seguito e consigliato durante il percorso della tesi.

Ringrazio tutti gli amici che a me sono stati vicini durante gli anni accedemici.

Infine ringrazio la mia famiglia per avermi supportato durante questi lunghi anni, e sono sicuro che sarà cosi anche per il futuro.

Bibliography

[1] Mariya I Skvortsova, Igor I Baskin, Ivan V Stankevich, Vladimir A Palyulin, and Nikolai S Zefirov. «Molecular Similarity. 1. Analytical Description of the Set of Graph Similarity Measures». In: *Journal of chemical information and computer sciences* 38.5 (1998), pp. 785–790 (cit. on p. 1).

[2] Maureen Heymans and Ambuj K Singh. «Deriving Phylogenetic Trees from the Similarity Analysis of Metabolic Pathways». In: *Bioinformatics* 19.suppl_1 (2003), pp. i138–i146 (cit. on p. 1).

[3] Björn H Junker and Falk Schreiber. *Analysis of Biological Networks*. Vol. 2. Wiley Online Library, 2008 (cit. on p. 1).

[4] Andrew Dalke and Janna Hastings. «FMCS: a novel algorithm for the multiple MCS problem». In: *Journal of cheminformatics* 5.Suppl 1 (2013), O6 (cit. on p. 1).

[5] Jon M Kleinberg. «Authoritative sources in a hyperlinked environment». In: *Journal of the ACM (JACM)* 46.5 (1999), pp. 604–632 (cit. on p. 1).

[6] Stanley Milgram. «The Small World Problem». In: *Psychology Today* 2.1 (1967), pp. 60–67 (cit. on p. 1).

[7] Younghee Park and Douglas Reeves. «Deriving Common Malware Behavior through Graph Clustering». In: *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ASIACCS '11. Hong Kong, China: Association for Computing Machinery, 2011, pp. 497–502. ISBN: 9781450305648. DOI: 10.1145/1966913.1966986. URL: https://doi.org/10.1145/1966913.1966986 (cit. on p. 1).

[8] Jochem H. Rutgers, Pascal T. Wolkotte, Philip K.F. Hölzenspies, Jan Kuper, and Gerard J.M. Smit. «An Approximate Maximum Common Subgraph Algorithm for Large Digital Circuits». In: *2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*. 2010 (cit. on p. 1).

[9] Thomas Zimmermann and Nachiappan Nagappan. «Predicting Subsystem Failures using Dependency Graph Complexities». In: *The 18th IEEE International Symposium on Software Reliability (ISSRE '07)*. 2007, pp. 227–236. DOI: 10.1109/ISSRE.2007.19 (cit. on p. 1).

[10] SG Shirinivas, S Vetrivel, and NM Elango. «Applications of graph theory in computer science an overview». In: *International journal of engineering science and technology* 2.9 (2010), pp. 4610–4621 (cit. on p. 1).

[11] Ngoc Hieu Tran, Kwok Pui Choi, and Louxin Zhang. «Counting motifs in the human interactome». In: *Nature communications* 4.1 (2013), p. 2241 (cit. on p. 2).

[12] Tahereh Pourhabibi, Kok-Leong Ong, Booi H Kam, and Yee Ling Boo. «Fraud detection: A systematic literature review of graph-based anomaly detection approaches». In: *Decision Support Systems* 133 (2020), p. 113303 (cit. on p. 2).

[13] Thomas Schank and Dorothea Wagner. «Finding, counting and listing all triangles in large graphs, an experimental study». In: *International Workshop on Experimental and Efficient Algorithms* (2005), pp. 606–609 (cit. on pp. 2, 10, 32).

[14] Santosh Pandey et al. «Trust: Triangle counting reloaded on GPUs». In: *IEEE Transactions on Parallel and Distributed Systems* 32.11 (2021), pp. 2646–2660 (cit. on pp. 3, 32, 33, 39, 61, 66).

[15] Yang Hu, Hang Liu, and H Howie Huang. «Tricore: Parallel triangle counting on GPUs». In: *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2018, pp. 171–182 (cit. on pp. 3, 10, 33, 61, 66).

[16] Santosh Pandey, Xiaoye Sherry Li, Aydin Buluc, Jiejun Xu, and Hang Liu. «H-index: Hash-indexing for parallel triangle counting on GPUs». In: *2019 IEEE high performance extreme computing conference (HPEC)*. IEEE. 2019, pp. 1–7 (cit. on pp. 3, 32, 61, 66).

[17] Yangzihao Wang et al. «Gunrock: GPU graph analytics». In: *ACM Transactions on Parallel Computing (TOPC)* 4.1 (2017), pp. 1–49 (cit. on pp. 3, 33, 61, 66).

[18] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. «A Lightweight Infrastructure for Graph Analytics». In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, 2013, pp. 456–471 (cit. on pp. 3, 32, 54, 61, 66).

[19] Jiya Su, Alexander Fell, Andronicus Rajasukumar, David F Gleich, and Andrew A Chien. «Scaling triangle counting and k-truss on the updown architecture». In: *2025 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE. 2025, pp. 1–7 (cit. on pp. 9, 32, 33).

[20] Andronicus Rajasukumar, Tianchi Zhang, Ruiqi Xu, and Andrew A Chien. «Updown: A novel architecture for unlimited memory parallelism». In: *Proceedings of the International Symposium on Memory Systems*. 2024, pp. 61–77 (cit. on p. 33).

[21] YuAng Chen and Jeffrey Xu Yu. «ToT: Triangle counting on tensor cores». In: *IEEE Transactions on Parallel and Distributed Systems* (2025) (cit. on pp. 33, 40).

[22] Carl Yang, Aydın Buluç, and John D Owens. «GraphBLAST: A high-performance linear algebra-based graph framework on the GPU». In: *ACM Transactions on Mathematical Software (TOMS)* 48.1 (2022), pp. 1–51 (cit. on p. 33).

[23] Alex Fender, Brad Rees, and Joe Eaton. «Rapids cugraph». In: *Massive Graph Analytics*. Chapman and Hall/CRC, 2022, pp. 483–493 (cit. on p. 33).

[24] NVIDIA Corporation. *CUDA C++ Programming Guide: Warp Shuffle Functions*. `https://docs.nvidia.com/cuda/cuda-programming-guide/index.html#warp-shuffle-functions`. Accessed: 2024-05-22. 2024 (cit. on p. 35).

[25] NVIDIA Corporation. *CUDA C++ Programming Guide: Cooperative Groups*. `https://docs.nvidia.com/cuda/cuda-programming-guide/index.html#cooperative-groups`. Accessed: 2024-05-22. 2024 (cit. on p. 35).

[26] Li Zeng, Kang Yang, Haoran Cai, Jinhua Zhou, Rongqian Zhao, and Xin Chen. «HTC: Hybrid vertex-parallel and edge-parallel Triangle Counting». In: *2022 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE. 2022, pp. 1–7 (cit. on p. 35).

[27] Jure Leskovec and Andrej Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. `http://snap.stanford.edu/data` (cit. on p. 54).

[28] David A. Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner, eds. *Graph Partitioning and Graph Clustering: 10th DIMACS Implementation Challenge Workshop*. Vol. 588. Contemporary Mathematics. American Mathematical Society, 2013. URL: `https://www.cc.gatech.edu/dimacs10/` (cit. on p. 54).