Characterizing CPU-GPU Collaboration in Modern Heterogeneous Systems

Ferran Llorà

February 9, 2026

Abstract–

Heterogeneous systems are ubiquitous, present in systems as diverse as mobile phones, laptops, and high-end supercomputers. While traditional CPUs cannot provide sufficient throughput for compute-intensive workloads, GPUs alone are not suitable for all tasks. Thus, demanding applications require integrating both types of devices into a single heterogeneous system. As such, the efficiency of communication and the level of collaboration between these devices have become increasingly important in recent years. Computer architects need mechanisms and methodologies that accurately assess the integration of heterogeneous systems. This level of integration can be measured by carefully studying a system's behaviour under different collaboration patterns. However, these collaboration patterns are challenging to study because they depend on many variables.
The CHAI application suite is a set of program kernels designed to measure the degree of integration between CPU and GPU through collaborative workloads that test these collaboration patterns. However, CHAI was introduced in 2015 to target modest mobile phone architectures with few CPUs and a small GPU. Nowadays, even supercomputers are heterogeneous, integrating tens of CPUs and large GPUs. This TFG aims to refactor the original applications to leverage features of more recent CUDA versions and to rewrite them for AMD's HIP programming extensions. We also want to gather performance measurements on modern hardware, ranging from consumer to large HPC systems. Since these applications were designed for much less capable hardware, this work presents a scaling with the problem sizes used by the original applications to ones that yield meaningful results on modern CPU-GPU systems.
Keywords- Performance engineering, GPU, Heterogeneous systems.

✦

1 INTRODUCTION

CPUs are the foundation of all modern computing, but they fall short when dealing with problems that require high throughput. This limitation can be mitigated through the use of accelerators, which allow more complex tasks to be offloaded to specialized hardware tailored to the problem at hand. These are known as heterogeneous systems, which typically consist of a CPU and some type of accelerator.

In this TFG, we focus on CPU-GPU systems, which have become increasingly common in recent years, especially in the field of high-performance computing (HPC). An example of this trend can be observed in the TOP500 list, which ranks the 500 most powerful supercomputers in the world [1].

Nevertheless, heterogeneous systems are not a perfect solution. Broadly speaking, GPUs are programmable processors specialized for regular and massively parallel applications, while CPUs are better suited for more irregular workloads with moderate levels of parallelism, relying on strategies aimed at reducing operation latency. This specialization makes each device highly effective in its respective domain and can provide significant speedups when used in the appropriate context. However, many applications cannot be executed efficiently on either the CPU or the GPU alone and, therefore, require collaboration between both in order to fully exploit the system's capabilities.

This collaboration, however, is not free. The overhead associated with it is often closely tied to the level of hardware support provided by the system. Such hardware support typically takes the form of communication and synchronization mechanisms, including cache coherence between CPU and GPU memories, as well as buses or interconnects offering high bandwidth and/or low latency.

As a result, the presence or absence of these mechanisms, together with their performance characteristics, ultimately determines whether certain types of collaboration are worthwhile. Efficient systems with cache coherence and low-latency communication enable more fine-grain and frequent collaboration patterns, whereas traditional systems

• E-mail de contacte: ferran.llora@autonoma.cat
• Menció realitzada: Enginyeria de Computadors
• Treball tutoritzat per: Victor Soria (BSC) i Juan Carlos Moure Lopez (DACSO)
• Curs 2026/27

lacking such support force the adoption of coarser collaboration schemes with reduced synchronization.

The problem is that measuring how different hardware support mechanisms affect the efficiency of various collaboration schemes is both complex and time-consuming. For this reason, CHAI [2] was developed: a suite of heterogeneous applications designed to capture different types of collaboration and evaluate their performance on heterogeneous systems. CHAI was introduced in 2015, targeting heterogeneous mobile devices, and since then, there have been significant advances in the field, offering new levels of hardware support and new software features.

In this work, we present an updated version of the original applications, adapted to modern hardware and focused on HPC environments. We also provide an analysis of the performance of these applications on three selected systems.

2 Objectives and Contributions

This TFG presents an updated version of the CHAI applications suitable for modern high-performance computing (HPC) systems. Our update focuses on scaling the benchmarks by using larger input datasets and fixing bugs present in the original applications. In addition, this project provides a port of the CUDA-based implementations to HIP, AMD's GPU programming framework, to improve portability across different hardware platforms.

We also propose an analysis of the performance results obtained on the three proposed systems. This analysis is based on execution times and execution timelines obtained using Nvidia's and AMD's profiling tools. In short, this work presents the following contributions.

- Fixing bugs and correcting inconsistencies in the original CHAI CUDA applications.
- Scaling the benchmarks to larger problem sizes suitable for modern CPU-GPU heterogeneous systems.
- Porting the CUDA-based CHAI applications to HIP in order to support AMD GPUs.
- Testing the CHAI applications' functionality and performing a systematic performance characterization of CPU-GPU collaboration across consumer and HPC platforms.

3 Methodology

The development of this TFG follows a *Spiral Iterative Model*, which is particularly well suited for performance engineering and experimental research on heterogeneous systems. In this context, early experimental results frequently reveal unexpected bottlenecks or architectural effects that require revisiting both the implementation and the evaluation methodology.

Each iteration of the spiral consists of four main stages. First, during the *planning* phase, a specific subset of CHAI applications or an architectural feature of interest (such as Unified Memory behavior or system-wide atomics) is selected for study. This is followed by the *implementation* phase, where bugs in the original CHAI codebase are fixed, problem sizes are scaled to better match modern hardware capabilities, and applications are adapted to recent CUDA versions and ported to AMD's HIP framework.

In the *experimentation* phase, applications are executed on different heterogeneous platforms using multiple configurations of CPU threads and GPU work-groups. Profiling tools are used to collect execution times and execution timelines, allowing observation of synchronization points, idle periods, and overlap between CPU and GPU execution. Finally, during the *analysis* phase, the collected data is examined to identify scalability limits, load imbalance, synchronization overheads, and the impact of hardware support for memory coherence and interconnects.

Results obtained during the analysis phase often motivate changes in the implementation or experimental setup, leading to a new iteration of the spiral. This iterative process allows both the benchmark suite and the evaluation methodology to progressively converge towards an accurate characterization of CPU–GPU collaboration on modern heterogeneous systems.

4 State of the Art

The characterization of heterogeneous CPU-GPU systems has been an active research topic for more than a decade, driven by the widespread adoption of accelerators in both consumer and high-performance computing (HPC) environments. Early research primarily focused on GPU offloading and comparisons between CPU-only and GPU-only executions, with limited emphasis on fine-grain collaboration and synchronization between devices.

Several benchmark suites have been proposed to evaluate heterogeneous architectures. One of the most widely used is the *Rodinia* benchmark suite [3], which provides a collection of applications representative of workloads in scientific computing, data mining, and machine learning. Rodinia includes both CPU and GPU implementations and is designed to expose parallelism, memory access patterns, and scalability across architectures.

While Rodinia is highly effective for evaluating performance portability and raw accelerator efficiency, its applications are typically designed to execute either on the CPU or on the GPU. As a result, Rodinia does not explicitly stress frequent synchronization, dynamic load balancing, or fine-grain collaboration between CPUs and GPUs during execution.

Other research efforts have explored programming models and execution strategies aimed at improving CPU-GPU interaction, such as persistent kernels [4], unified memory systems [5, 6], and heterogeneous pipelines [7]. These works analyze the trade-offs between kernel launch overhead, synchronization frequency, and load balancing, often highlighting the importance of hardware support for cache coherence and low-latency interconnects.

The *CHAI* benchmark suite was introduced to specifically address this gap by focusing on collaborative heterogeneous applications. Unlike Rodinia, CHAI applications are explicitly designed to exercise different collaboration patterns, including data partitioning and fine- and coarse-grain task partitioning.

Similarly, *Hetero-Mark* [8] is another heterogeneous application suite that targets CPU–GPU cooperative execution, emphasizing communication, synchronization, and workload partitioning in real-world applications. Its focus is to reflect real-world practical scenarios, whilst CHAI is meant to study collaboration strategies and the programming model implications of heterogeneous execution.

5 Background

In this section, we present a series of concepts necessary to understand the contents of this work. We first explain key technical concepts in GPU architecture and programming models, then describe the collaboration patterns presented in the original CHAI paper.

5.1 GPU

For this Bachelor's Thesis, GPUs from both NVIDIA and AMD are used. Although the overall architecture and execution model of GPUs from these two vendors are conceptually very similar, each company employs its own terminology to describe hardware components and parallel execution abstractions. In order to avoid confusion, this work adopts NVIDIA's terminology as the reference, since it is more widely used in the literature and documentation [9]. In this section, the equivalent AMD terminology is also provided to facilitate comparison and ensure clarity.

GPU programming follows a hierarchical parallel execution model that directly reflects the underlying hardware organization. At the hardware level, a GPU consists of multiple Streaming Multiprocessors (SMs), referred to as Compute Units (CUs) in AMD terminology. Each SM/CU contains a number of arithmetic units capable of executing many threads concurrently. Unlike CPU cores, which independently execute instruction streams, SMs execute groups of threads in a coordinated manner to maximize instruction throughput.

At the programming level, a kernel launch defines a large number of parallel threads. These threads are grouped into thread blocks (called work-groups on AMD platforms), which are scheduled onto individual SMs/CUs. All threads within a block execute on the same SM and can cooperate via fast on-chip shared memory and explicit synchronization. Blocks are independent of one another, allowing the GPU to schedule them in any order and scale execution across available hardware resources.

Within each block, threads are further subdivided into warps (or wavefronts in AMD terminology), which represent the basic unit of execution. Threads within a warp execute the same instruction simultaneously in a SIMD-like fashion, while maintaining separate registers and control flow. Individual threads (or work-items in AMD terminology) are identified by unique indices that determine which portion of the data they operate on. If threads within a warp diverge due to conditional branches, the warp serially executes each branch path while masking out threads that are not on that path, which can reduce parallel efficiency until the threads reconverge.

5.2 Unified Memory (UM)

Historically, GPUs are devices separated from the CPU, having their own memory. This naturally led to a programming model where memory is managed manually, meaning that the programmer must explicitly copy the data from/to the GPU memory to/from the CPU memory.

Unified Virtual Memory (UVM) is a mechanism implemented via software that provides the illusion that the CPU and GPU have a single coherent shared memory, which can manage data movement automatically. It does this by creating a single virtual address space shared between the CPU and GPU. UVM transparently migrates pages in this shared address space when either the CPU or GPU accesses data hosted in the other's memory [10]. This greatly simplifies memory management by the programmer, and is called the Unified Memory (UM) programming model.

With proper hardware support, its performance can be close to that of manually managed memory [5, 6]. However, as we will see later, without adequate hardware support, this can degrade performance.

As noted previously, heterogeneous systems typically have separate physical memories for the CPU and GPU; however, this is not always the case. Architectures supporting Unified Physical Memory (UPM), as the name implies, contain a single physical memory shared by the CPU and the GPU. This comes with lots of benefits, mainly performance and memory usage, derived from the elimination of unnecessary data transfers and data duplication [11]. It also has some downsides, for instance, the bandwidth has to be shared among the CPU and GPU.

5.3 Persistent Kernel Model

The traditional CUDA programming model states that the application workload must launch as many work-groups as required. Each work-group is a team of threads, possibly but not necessarily collaborating on the same task, with all work-groups containing the same number of threads, typically a multiple of 32. However, for some applications that require frequent synchronization between the CPU and the GPU, a different application model, called *persistent* [4], is used.

The persistent kernel model states that, instead of launching as many work-groups as there are tasks, only as many work-groups as the GPU can simultaneously execute are launched. These work-groups keep processing tasks until no work remains, at which point they are terminated. In this way, the overhead of launching work-groups for small tasks can be reduced by using global atomics for all communication [12].

5.4 Collaboration Patterns

The applications from the original CHAI paper are divided into categories based on how work is partitioned and distributed between the CPU and GPU. These categories are *data partitioning* and *task partitioning*, the latter being further divided into fine-grain and coarse-grain task partitioning. Figure 1 graphically shows these collaboration patterns.

Figure 1 (a) shows a typical application's structure. As shown in the legend, an application is a hierarchy of tasks, containing data dependences and requiring synchronization. Depending on the amount of work of a certain group

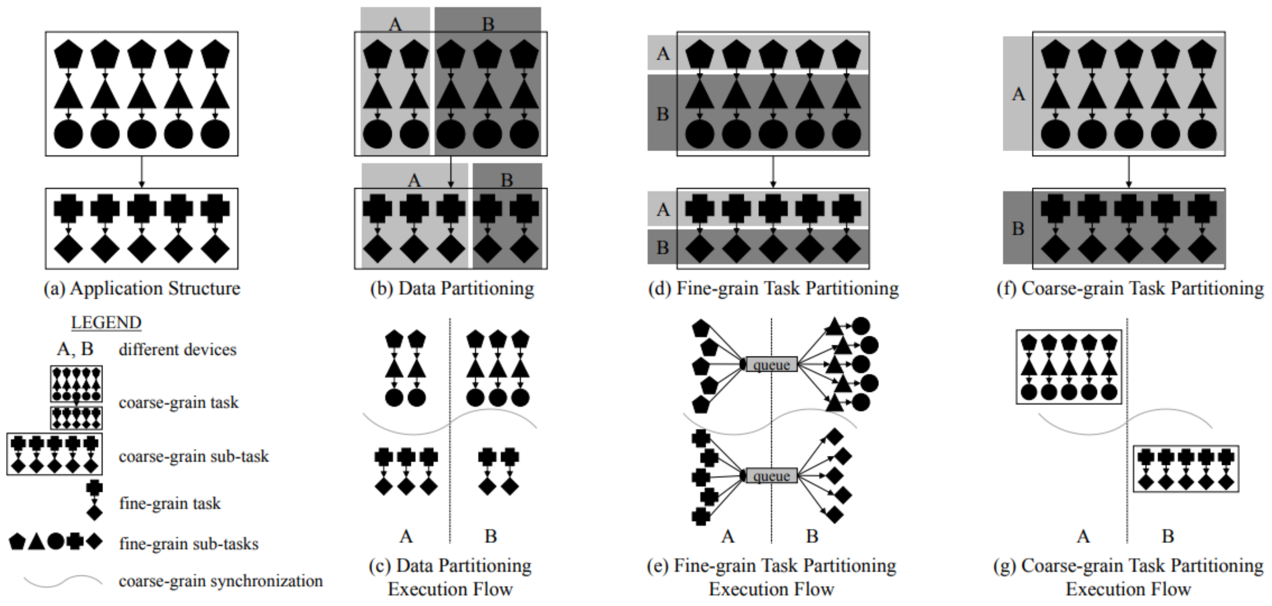
(a) Application Structure
A
B
(b) Data Partitioning
(d) Fine-grain Task Partitioning
(f) Coarse-grain Task Partitioning
LEGEND
A, B different devices
coarse-grain task
coarse-grain sub-task
fine-grain task
fine-grain sub-tasks
coarse-grain synchronization
(c) Data Partitioning Execution Flow
queue
(e) Fine-grain Task Partitioning Execution Flow
(g) Coarse-grain Task Partitioning Execution Flow

Figure 1: Collaboration patterns

of tasks, we can classify the group as fine-grain or coarse-grain. The classification may depend on the characteristics of the hardware. In the example, the application consists of small fine-grain sub-tasks, containing data dependencies that create a chain that can also be interpreted as a fine-grain task. Some groups of similar tasks, usually doing the same operations on different data items, can be aggregated into bigger groups of tasks to form coarse-grain sub-tasks. These, in turn, can form a coarse-grain task when multiple of them are chained together.

5.4.1 Data Partitioning

In *data partitioning*, the input is split between the two devices, such that they both will do the same work on different parts of the input. Figure 1 (b) shows how this can be applied to the previous example in Figure 1 (a).

Applications that use the data partitioning scheme can distribute work statically or dynamically. Partitioning work statically means predefining which percentage of the work is assigned to the CPU or GPU, so no communication is required during the execution. If the work is partitioned dynamically, a work pool stored in UVM and shared by CPU and GPU will be necessary. This work pool is managed through system-wide atomics, and both the CPU and GPU grab work on demand.

Data partitioning can be useful in Heterogeneous Systems because the storage of input data and output data in SVM can help avoid redundant data copies and movement. Moreover, SVM greatly eases programmability by managing memory automatically.

However, data partitioning may be counterproductive when applied to more fine-grain tasks in highly discrete systems, since the overhead of managing small tasks can be relatively higher.

5.4.2 Task Partitioning

In *task partitioning*, rather than the input being split, the application is partitioned into various segments so that they can be executed in a pipelined fashion. The different stages of this pipeline are then distributed to either the CPU or GPU, with the objective that each will execute the tasks more suitable in their case. Our applications can achieve this in two ways; Fine-grain and Coarse-grain task partitioning, represented graphically in Figure 1 (d) and Figure 1 (f).

As shown in Figure 1 (e), fine-grain task partitioning uses a queue system that is implemented in UVM and managed via system-wide atomics. This mechanism is well-suited for handling small tasks and frequent synchronization.

In the case of coarse-grain task partitioning, the application waits for all the subtasks to be finished before moving on to the next coarse-grain task, as seen in Figure 1 (g). This can be implemented in a variety of ways, so it will be clarified when explaining each application.

Heterogeneous systems can benefit from task partitioning because general-purpose and latency-oriented tasks can be executed on the CPU, whilst more regular and throughput-demanding tasks can be executed on the GPU. This enables potential speed-ups on applications and improved power efficiency.

6 Applications

The original CHAI application suite consists of 14 applications. However, two of them are the same application implemented using different collaboration patterns. In this work, these implementations are treated as a single application, yielding a total of 12. In this section, we analyse 4 of them, two of which (CEDD/CEDT and RSCD/RSCT) are the aforementioned cases. These are BS and CEDD/RSCD for data partitioning, BFS and CEDT for coarse-grain task partitioning, and finally RSCD for fine-grain task partitioning.

We decided to use BS for data partitioning because it is a fairly simple example to understand, and the simplest case of all. On top of this, it highlights that collaboration on data partitioning schemes is mostly only fruitful on smaller, well-integrated systems.

Next, we use BFS because it represents the "traditional" model of leaving the execution of entire tasks to the GPU. In addition, it highlights the importance of hardware support for system-wide atomics.

Finally, we opted for RSC and CED because we believe that the comparison between task and data partitioning is interesting, and moreover, RSCD/RSCT cover the remaining collaboration pattern, fine-grain task partitioning. Not only that, but RSCT shows the true potential of collaborative applications, where, on sufficiently integrated architectures, we can obtain significant speedups for certain applications.

6.1 Bézier Surface (BS)

Bézier Surface is an algorithm for creating a smooth surface by blending a set of control points. These control points can be represented as an input matrix, which our application takes as input. The output is a bigger matrix corresponding to the 2D surface.

This application uses data partitioning on the input matrix by splitting it into square tiles, which are then assigned to CPU threads or GPU work-groups. This matrix, along with the output matrix, is stored in UVM.

6.2 Canny Edge Detection (CED)

Canny edge detection is a common algorithm used to detect edges in a given input image. It does this by applying 4 filters that result in an image in black and white, outlining the edges of the image. We have two implementations for this application.

Our first implementation (CEDD) takes as input a number of frames. This implementation uses data partitioning, such that for each frame, either the entire CPU or the entire GPU computes the 4 filters. UVM is used to store the input frames and the intermediate results between filters.

The second implementation (CEDT) performs task partitioning on the filters. The first two are assigned to the CPU because they contain flow-control statements that may cause threads to diverge on the GPU, whereas the other two are more regular and thus assigned to the GPU. These filters are then executed in a pipelined manner.

6.3 Random Sample Consensus (RSC)

Random Sample Consensus (RANSAC) is an algorithm used to estimate the parameters of a mathematical model from random samples. Each iteration consisting on selecting a subset of samples and evaluating the model is called a trial. Similar to Canny edge detection, this application comprises two stages in every trial: first, an inherently sequential model-fitting stage, followed by a massively parallel model-evaluation stage. We provide two implementations: one based on data partitioning and the other on task partitioning.

The data partitioning version of this application (RSCD) performs data partitioning on the trials. Each individual trial can be assigned to one CPU thread or GPU work-group. In the case of the GPU version, since the first stage is fully sequential, one thread of the work-group does all the work, whilst the rest wait for it to finish and join it on the second massively parallel stage. UVM is used to store the input samples and the outlier count.

In the task partitioning version (RSCT), the first stage, which is inherently sequential, is executed on the CPU, whilst the second massively parallel stage is executed on the GPU. This version of task partitioning is considered fine-grain, so it uses a task queue system and system-wide atomics. Just as before, UVM is used to store the input samples and the outlier count. Additionally, it is used to store intermediate values between stages.

6.4 Breadth-First Search (BFS)

Breadth-First Search is a common algorithm that computes the shortest path between two nodes on an unweighted graph. It explores the graphs by steps. This means that on the first step, the root nodes are explored, on the second step, the root node's neighbours are explored, and so on.

Before exploring any nodes, our implementation counts the number of nodes that have to be explored, and if it exceeds a manually defined threshold, they are explored on the GPU. Otherwise, they are explored on the CPU.

This application is considered coarse-grain task partitioning, but unlike Canny Edge Detection, it uses persistent kernels. At the end of each step, it uses system-wide atomics to wait for CPU threads and GPU work-groups to finish their work. Its goal is to leave more latency-oriented tasks to the CPU, while tasks with enough work to benefit from massive parallelism go to the GPU.

7 Experimental Methodology

In this section, we describe the experimental choices and the conditions under which all experiments were conducted. We begin by detailing the platforms used. Next, we present the criteria used for the CPU-only and GPU-only tests, and finally, we specify the metrics used in the performance evaluation.

7.1 Platforms Definitions

We have selected three representative platforms for evaluating CPU-GPU collaboration in modern HPC systems. The first are the nodes of the accelerated partition at BSC's MareNostrum 5. This system consists of an Intel Xeon Platinum 8460Y+ CPU, with 40 cores and 2 hardware threads per core, and an Nvidia H100 GPU, consisting of 132 Streaming Multiprocessors. The two are connected via a PCIe Gen 5 bus. We refer to this system as "MN5".

The second system is AMD's Ryzen AI 9 HX 370. This system consists of a CPU with four standard cores and eight performance cores, totalling 12, coupled with an AMD Radeon 890M integrated GPU, with 16 Compute Units. Although this system allows memory to be configured in a manner that resembles discrete allocation between the CPU and GPU, in this work, we use a fully unified memory configuration, where both processors operate on the same physical memory space. As a result, no explicit data copies are required when transferring data between the CPU and the GPU. Additionally, this system provides support for cache coherence between the CPU and GPU. This is supported by the results presented later in the results section. We refer to this system as "AMD".

The final system is an Nvidia GH200 Superchip. It consists of an Nvidia Grace CPU with 72 cores and no hardware multithreading, and a Hopper GPU with 132 Streaming Multiprocessors, each with its own physical memory. The two are connected via an NVLink Chip-to-Chip bus, a high-bandwidth link that maintains memory and cache coherence between the CPU and GPU. This bus also offers lower latency than traditional PCIe buses and hardware support for system-wide atomics. We will refer to this system as "GH200".

7.2 Experimental Setup

Next, we will describe the experiments to measure scalability on the CPU and GPU, and the experiments to obtain performance results for collaborative executions.

7.2.1 CPU-only Tests

For the CPU-only executions, the single-threaded version (1 thread) is used as a baseline, progressively increasing the number of threads until reaching the version with n threads, where n corresponds to the number of physical cores in the system. At each step, the number of threads is doubled.

Although most of the evaluated systems support *hardware multithreading*, for the sake of simplicity in the analysis, we limited execution to the exclusive use of physical cores.

7.2.2 GPU-only Tests

GPU-only executions present higher complexity. Unlike CPU cores, GPU cores can host multiple resident *work-groups* and execute them in parallel. However, the size of these work groups can both affect how many of them can be executed in parallel and their individual performance. For example, some kernels may have sequential sections that underutilize larger blocks. As a consequence, the best performing work-group size both varies across applications and has a significant impact on each application's performance.

For this reason, the number of work-groups is not an appropriate metric for evaluating the scalability of GPU-only versions, as it does not allow fair comparisons across different applications.

Instead, we use the total number of threads as a metric, defined as the optimal number of threads per work-group multiplied by the number of work-groups. This decision is based on the observation that applications with fewer threads per work-group will have more work-groups in total, and vice versa, providing a common framework for comparing scalability across applications.

7.2.3 Collaborative Versions Tests

In the case of unified versions, several parameters can be varied, including the number of CPU threads, GPU work-groups, and application-specific coefficients.

Unlike CPU-only or GPU-only executions, unified versions do not always benefit from using more CPU cores. On the contrary, there are several cases where the best performance is achieved by using fewer resources. For this reason, we evaluated most of the possible configurations and selected the best one for each application in terms of execution time. Tables 1, 2, and 3 show the best configurations for collaborative versions of each application across all systems. Tables 4, 5, and 6 show the configurations used for every application across all systems, such as input used, and application-specific parameters.

7.3 Experimental Setup

All experiments were conducted exclusively on the respective systems, with the goal of obtaining results free from external interference. During these experiments, execution times were collected for different sections of each application. In general, these sections include Allocation time, Initialization time, Kernel time, and Deallocation time. In this work, we focus on kernel execution time, as it provides the most relevant information regarding application behaviour in a collaborative CPU-GPU environment. To obtain this value, several *warm-up* executions are first performed, followed by a set of final executions, of which we obtain an average.

To evaluate performance in greater detail, we use Nvidia and AMD profiling tools to obtain execution timelines that include CPU task start and end times. In the case of the GPU, technical limitations stop us from gathering real execution times of tasks, so an estimation will be made by obtaining the average time of a work-group based on the execution time of the entire grid and the number of work-groups.

Finally, for all the applications, technical limitations impede that we capture task execution times inside the GPU kernels, in the same way as we did with the CPU. For this reason, the execution times of tasks for individual GPU work-groups were calculated by assuming all GPU work-groups do the same amount of work.

8 PERFORMANCE RESULTS

To analyse collaborative versions, we first examine the best cases for both CPU-only and GPU-only versions. For this reason, we assess the scalability of each proposed application for each system. Note that the task-partitioning versions of Canny Edge Detection and Random Sample Consensus (CEDT and RSCT) are not included because they require collaboration.

8.1 CPU Scalability

Figure 2 shows the speed-up gained on each application when using more CPU threads on each system individually. We observe that BS and RSCD scale almost perfectly across all systems, with high efficiencies on all systems except AMD.

This is expected, since RSCD, and BS have no strong dependencies and allow their inputs to be split into chunks that can be processed independently. One outlier, however, is CEDD on the GH200 and MN5. We can see performance stops improving significantly after 32 threads. This is due to individual frames not containing sufficient work for the

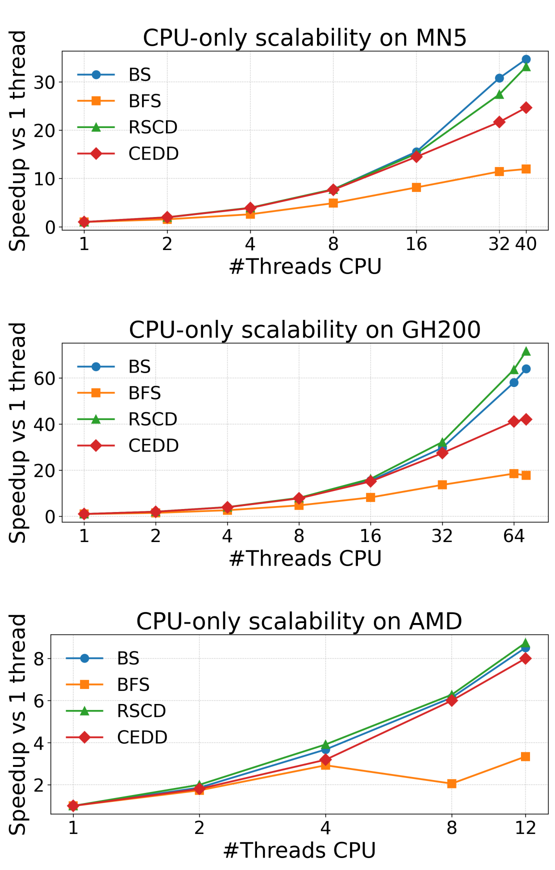

Figure 2: CPU Speedup vs 1 thread version on: (a) MN5, (b) GH200, and (c) AMD.

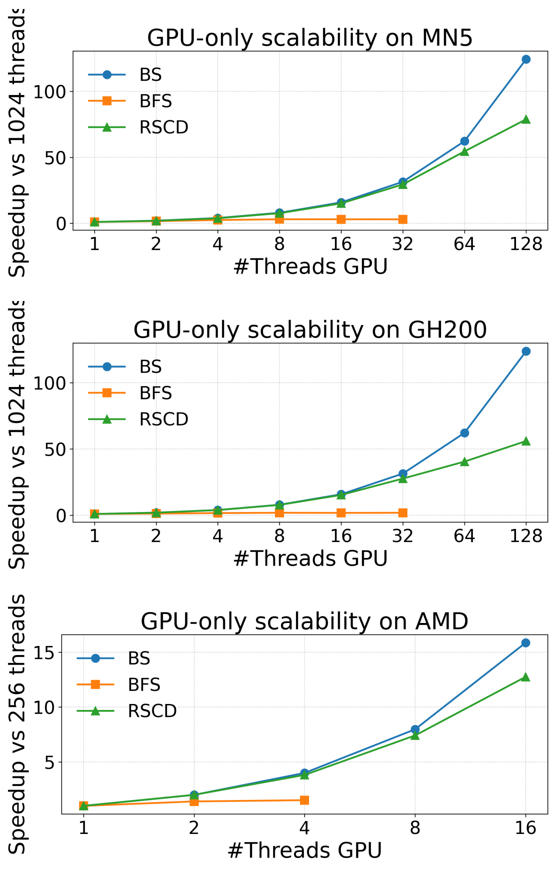

Figure 3: GPU Speedup vs 1024 threads on MN5 (a) and GH200 (b), and vs 256 threads version on AMD (c).

high number of cores in GH200 and MN5 compared with AMD. A solution was to increase the size of the problem, however, the input frames became too big and were causing issues with the amount of available disk memory.

In contrast, BFS scales much worse than the other applications. This behavior is not inherent to the application, but rather a consequence of the limited input size available.

Finally, we observe atypical scaling behavior on the AMD system, where performance decreases when doubling the number of threads. This effect appears to be configuration-specific, as it is only observed on this platform for this particular input size and thread count, and requires further investigation.

8.2 GPU Scalability

Next, we look at the performance of these same applications on the GPU, partitioning the graphs in the same way as before. We also note that GPU scalability does not account for CEDD, since, as previously mentioned, it does not use the persistent kernel model. Figure 3 shows the speed-up achieved from using different numbers of GPU threads for the GPU-only versions of the previously mentioned applications.

Across all three systems, we observe that RSCD scalability stagnates as thread counts increase due to the increase in synchronization that comes with the extra work-groups. Aside from these cases, for the most part, applications on the GPU scale almost directly with the number of threads.

Similar to the CPU case, BFS is notable for its limited scalability. The input graph is too small to fully occupy the GPU, and additional experiments were omitted once this behaviour became evident. For the remaining applications, GPU-only executions scale nearly linearly with the number of threads across all platforms, indicating that these kernels are generally well-suited to massive parallel execution.

8.3 Collaboration Analysis

After analysing CPU-only and GPU-only scalability, we now focus on collaborative executions to assess how different CPU-GPU collaboration patterns affect performance on each platform. In summary, Figure 4 shows the speedup of the best collaborative configuration over the best CPU/GPU-only configuration on each application. Below is an analysis of these results.

8.3.1 Bézier Surface (BS)

In this case, we observe that MN5 and GH200 perform poorly, whereas tasks on the AMD systems run quite quickly. This application employs data partitioning, which means that the CPU and GPU perform the same work. However, this workload contains no flow-control statements and is highly regular, making it well-suited for GPU execution. Moreover, these executions contain few tasks, which means that collaborative versions perform worse because they must wait for a single CPU task.

This occurs only on MN5 and GH200 because these systems are highly unbalanced in terms of CPU and GPU com-

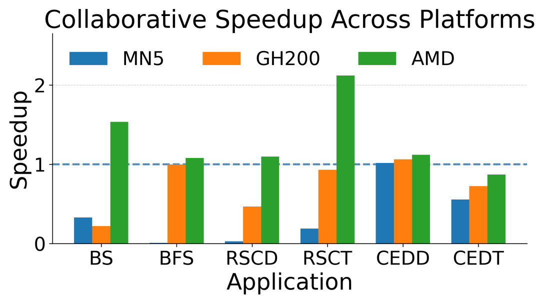

Figure 4: Speedup of best collaborative version vs best CPU/GPU-only version.

putational capacity. On the AMD system, which has a more modest GPU, the CPU-to-GPU computation ratio is more balanced, so this problem does not appear. Furthermore, performance improves because CPU and GPU tasks take roughly the same amount of time, so the execution time is effectively the sum of both, and optimizing one meaningfully affects the total runtime.

8.3.2 Breadth First Search (BFS)

First, we observe that collaborative BFS on MN5 lags behind the best CPU/GPU-only performance. On the other two machines, collaborative performance is comparable to that of the non-unified version and even improves on the AMD system. We believe could be due to the lack of hardware support for system-wide atomics in this system as well as the data movement between the CPU and GPU. As we mentioned earlier, BFS uses system-wide atomics to wait for CPU threads and GPU work-groups to finish their work at the end of each step. This means that even if the CPU is waiting for the GPU to finish, it uses atomics frequently. When the GPU finishes its work, it also performs an atomic operation to signal the task's end. This results in a high number of page faults that exchange data between the CPU and the GPU via the PCIe bus, drastically reducing performance.

We measured time spent in this section together with getting work from the global work pool, which are both atomic-intensive sections, and got that, for MN5, a total of 90% and 24% of execution time was spent on both sections on GPU and CPU, respectively. Since the GPU performs 99,9% of iterations, this results in a massive loss of performance. To illustrate this, the best GPU-only version spent 14% of time on the synchronization stage.

In the case of the GH200, these page migrations are not a problem because the NVlink C2C alleviates the bottleneck, with time spent on the synchronization phase being 0,8% and 1,5% for GPU and CPU, respectively. The relatively good performance on AMD suggests optimized hardware support for system-wide coherent atomics. On AMD, the previous percentages are 3,6% and 5,4% of execution time for GPU and CPU, respectively.

Another point to note is that across all systems, the best configurations for collaborative workloads (Tables 1, 2, and 3) use a single CPU thread, and the best configuration for the GPU-only version is also single-threaded. The reason is that BFS takes advantage of the CPU's latency-oriented approach to quickly execute the small steps, while leaving the bigger ones to the GPU; thus, more threads would increase this latency.

8.3.3 Random Sample Consensus (RSCD/RSCT)

Finally, this data partitioning version (RSCD) is performed on fine-grain tasks, and synchronization is managed through atomics. On top of this, on every iteration, atomics are used to update a global counter. Like BFS, since the use of atomics is relevant to understanding the performance, we also measured the time spent on fetching work from the work pool and doing the task.

Similarly to BFS, this means a high usage of atomics and UVM, which really hurts performance on the MN5 system with no optimised hardware support, which spends 69% and 18% of execution time on the synchronization section on GPU and CPU respectively. On the other hand, GH200 performs much better in comparison, which is again due to the NVlink C2C, however, it still spends 35% and 12% on GPU and CPU synchronization respectively, compared to the best non-collaborative version, 5,7% on GPU-only.

Finally, the AMD system almost achieves some speedup, with a 1,1% and 1,4% of execution time spent on synchronization on the GPU and CPU respectively. This is the primary reason why we believe that this system has some kind of hardware support for system-wide atomics, since this kind of performance would not be possible if every atomic meant a page fault, like on MN5.

Finally, the task partitioning version (RSCT) is an improvement in performance. This is because, in this instance, even though atomics are used, they are used on a queue system, which means that the CPU/GPU only read/write, generating fewer page faults on MN5. On top of this, the global counter is only increased on the GPU, which traduces in a lowered use of system-wide atomics.

The final factor to take into account is that the section executed on the GPU was inherently sequential, which meant that on the data partitioning version, only one GPU thread was working on it, whilst the rest diverged. This means that even on unbalanced systems, the CPU can perform this section of the application much faster than the GPU does, giving us this dramatic improvement in performance on the AMD system.

The reason this performance uplift is not observed on the GH200 and MN5, however, is that system-wide atomics are still used very frequently. On systems with so many CPU threads and GPU work-groups active, this results in a performance loss due to synchronization.

8.3.4 Canny Edge Detection (CEDD/CEDT)

We observe that this application's data-partitioning version (CEDD) performs well, with only slight improvements over the non-unified version across all systems. There are two reasons for this behaviour.

First, this application consists of very coarse-grain tasks, specifically, 200 frames. Since each task is independent of the others, the synchronization overhead is negligible.

Secondly, GPU kernels contain control-flow statements that cause GPU threads to diverge; thus, although MN5 and GH200 are unbalanced and the GPU processes them faster, the difference is 5×, rather than the 25× of BS, so the GPU is not stuck waiting for the CPU.

However, this is not the case on the Task partitioning version. In this version, the GPU executes the more regular filters, while the CPU executes the more irregular ones. However, the GPU was already better than the CPU even for the irregular tasks, meaning that across all systems, the GPU is stuck waiting for the CPU, since both have to process the entire input.

9 Conclusions

In this work, we present an updated version of the CHAI application suite for characterizing CPU-GPU collaboration on modern heterogeneous systems, ranging from consumer-grade APUs to large-scale HPC platforms. We refactored the original applications, corrected existing bugs, scaled input sizes, and ported CUDA implementations to HIP to support AMD GPUs.

Through an experimental evaluation, we analyse how different collaboration patterns behave under varying levels of hardware support for unified memory, cache coherence, and system-wide atomics. The results demonstrate that CPU-GPU collaboration is highly dependent on the underlying architecture. Fine-grain collaboration can provide performance benefits on systems with strong coherence and low-latency interconnects, such as the GH200, but can severely degrade performance on traditional PCIe-based systems. Coarse-grain collaboration and data partitioning approaches are generally more robust, but may fail to fully exploit highly unbalanced CPU-GPU configurations.

The analysis also shows that collaborative execution is not universally beneficial. In most cases, CPU-only or GPU-only executions outperform unified versions due to synchronization overhead, workload imbalance, or insufficient task granularity. These results highlight the importance of carefully matching collaboration strategies to both application characteristics and architectural features.

Overall, this TFG demonstrates that modern collaborative benchmarks, such as CHAI, are a valuable complement to established suites such as Rodinia. While Rodinia excels at evaluating raw performance and portability, CHAI provides deeper insight into the practical costs and benefits of CPU-GPU collaboration. As future work, this study could be extended by incorporating additional applications, exploring adaptive workload partitioning strategies, and evaluating emerging architectures with even tighter CPU-GPU integration.

10 Use of AI

In this TFG, generative AI was used solely for editorial purposes, such as writing and figure plotting. It was not used for the analysis or for deciding what to write, but rather to better explain the contents of this work.

ACKNOWLEDGMENTS:

I would like to thank Santiago Marco for helping through the entire process of this bachelor's thesis. We would also like to thank NVIDIA for granting us access to their GH200 superchip, which was key in the making of this work.

References

[1] "Top 500." `https://top500.org/lists/top500/2025/06/`. Accessed: 2025-11-14.

[2] J. Gómez-Luna, I. El Hajj, V. Chang, Li-Wen Garcia-Flores, S. Garcia de Gonzalo, T. Jablin, A. J. Pena, and W.-m. Hwu, "Chai: Collaborative heterogeneous applications for integrated-architectures," in *Performance Analysis of Systems and Software (ISPASS), 2017 IEEE International Symposium on*, IEEE, 2017.

[3] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, pp. 44–54, IEEE, 2009.

[4] K. Gupta, J. A. Stuart, and J. D. Owens, "A study of persistent threads style gpu programming for gpgpu workloads," in *2012 Innovative Parallel Computing (InPar)*, pp. 1–14, 2012.

[5] N. Fujita, T. Boku, T. Yoshida, T. Shirai, and M. Tsuji, "Gpu-cpu shared memory performance analysis on nvidia gh200," in *2025 IEEE International Conference on Cluster Computing Workshops (CLUSTER Workshops)*, pp. 1–2, 2025.

[6] Kortelainen, Matti J., Kwok, Martin, and on behalf of the CMS Collaboration, "Performance of cuda unified memory in cms heterogeneous pixel reconstruction," *EPJ Web Conf.*, vol. 251, p. 03035, 2021.

[7] D. Gerzhoy and D. Yeung, "Pipelined cpu-gpu scheduling to reduce main memory accesses," in *Proceedings of the International Symposium on Memory Systems*, MEMSYS '21, (New York, NY, USA), Association for Computing Machinery, 2023.

[8] Y. Sun, X. Gong, A. K. Ziabari, L. Yu, X. Li, S. Mukherjee, C. Mccardwell, A. Villegas, and D. Kaeli, "Hetero-mark, a benchmark suite for cpu-gpu collaborative computing," in *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 1–10, 2016.

[9] NVIDIA Corporation, *CUDA C++ Programming Guide*, 2023. Accessed: 2026-02-09.

[10] H. Kim, J. Sim, P. Gera, R. Hadidi, and H. Kim, "Batch-aware unified memory management in gpus for irregular workloads," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, (New York, NY, USA), p. 1357–1370, Association for Computing Machinery, 2020.

[11] J. Wahlgren, G. Schieffer, R. Shi, E. A. León, R. Pearce, M. Gokhale, and I. Peng, "Dissecting cpu-gpu unified physical memory on amd mi300a apus," 2025.

[12] S. Kim, C. Jung, and Y. Kim, “Comparative analysis of gpu stream processing between persistent and non-persistent kernels,” in *2022 13th International Conference on Information and Communication Technology Convergence (ICTC)*, pp. 2330–2332, 2022.

11 Appendix

In this section we provide additional information regarding the configurations to run all applications. Tables 1, 2 and 3 show the best-case parameters for all collaborative versions of our applications. These include CPU threads, GPU blocks and GPU threads/block. CEDD and CEDT do no include number of GPU blocks since they do not use the persistent kernel model. In addition, Tables 4, 5 and 6 provide the best application-specific configurations and inputs used.

Table 1: Best configs unified MN5

	BFS	BS	CEDD	CEDT	RSCD	RSCT
CPU Threads	1	1	40	24	1	8
GPU Blocks	32	132	-	-	792	66
GPU Threads/Block	128	1024	784	784	128	128

Table 2: Best configs unified GH200

	BFS	BS	CEDD	CEDT	RSCD	RSCT
CPU Threads	1	1	64	32	1	64
GPU Blocks	32	132	-	-	528	528
GPU Threads/Block	128	1024	784	784	128	128

Table 3: Best configs unified AMD

	BFS	BS	CEDD	CEDT	RSCD	RSCT
CPU Threads	12	12	8	8	12	2
GPU Blocks	16	32	-	-	32	64
GPU Threads/Block	64	256	1024	1024	128	128

Table 4: Application specific configurations for MN5

Application	**Configuration**
BS	tasksize: 32x32, n: 300, m: 1000
BFS	input graph: USA-road, switching limit: 16
RSCD	max iterations: 4000000
RSCT	max iterations: 4000000
CEDD	input frames: Rick Astley (4K)
CEDT	input frames: Rick Astley (4K)

Table 5: Application specific configurations for GH200

Application	**Configuration**
BS	tasksize: 32x32, n: 300, m: 3000
BFS	input graph: USA-road, switching limit: 16
RSCD	max iterations: 4000000
RSCT	max iterations: 4000000
CEDD	input frames: Rick Astley (4K)
CEDT	input frames: Rick Astley (4K)

Table 6: Application specific configurations for AMD

Application	**Configuration**
BS	tasksize: 32x32, n: 300, m: 300
BFS	input graph: USA-road, switching limit: 32
RSCD	max iterations: 400000
RSCT	max iterations: 400000
CEDD	input frames: Rick Astley (4K)
CEDT	input frames: Rick Astley (4K)