Analysis of GPU Memory Allocation Characteristics

Marcos Rodriguez
Ikerlan Technology Research Center, Mondragón, Spain
Universitat Politècnica de Catalunya, Barcelona, Spain

Irune Yarza
Ikerlan Technology Research Center, Mondragón, Spain

Leonidas Kosmidis
Barcelona Super Computing Centre (BSC), Spain

Alejandro J. Calderón
Ikerlan Technology Research Center, Mondragón, Spain

Abstract

The number of applications subject to safety-critical regulations is on the rise, and consequently, the computing requirements for such applications are increasing as well. This trend has led to the integration of General-Purpose Graphics Processing Units (GPGPUs) into these systems. However, the inherent characteristics of GPGPUs, including their black-box nature, dynamic allocation mechanisms, and frequent use of pointers, present challenges in certifying these applications for safety-critical systems.

This paper aims to shed light on the unique characteristics of GPU programs and how they impact the certification process. To achieve this goal, several allocation methods are rigorously evaluated to determine which one is best suited to an application, regarding the program characteristics within the safety-critical domain.

By conducting this evaluation, we seek to provide insights into the complexities of GPU memory accesses and its compatibility with safety-critical requirements. The ultimate objective is to offer recommendations on the most appropriate allocation method based on the unique needs of each application, thus contributing to the safe and reliable integration of GPGPUs into safety-critical systems.

2012 ACM Subject Classification Computer systems organization → Parallel architectures; Software and its engineering → Real-time schedulability; Software and its engineering → Parallel programming languages

Keywords and phrases CUDA, Memory allocation, Rodinia, Embedded

Digital Object Identifier 10.4230/OASIcs.PARMA-DITAM.2025.1

Supplementary Material
Dataset (Experiment results): `https://github.com/marcosrc92/MARS-data` [20]
archived at `swh:1:dir:0b2fbba6fbdfa60cb3d84175a2dd102bfb293ff2`

Funding The research presented throughout this paper has received funding from the European Commission's Horizon Europe programme under the METASAT project (grant agreement 101082622), the Basque Government through the ELKARTEK programme within the framework of the AUTOTRUST project (grant number KK-2023/00019) and by the CERVERA programme within the framework of the MEDUSA project (grant number CER-20231011).

© Marcos Rodriguez, Irune Yarza, Leonidas Kosmidis, and Alejandro J. Calderón;
licensed under Creative Commons License CC-BY 4.0
16th Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures and 14th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2025).
Editors: Daniele Cattaneo, Maria Fazio, Leonidas Kosmidis, and Gabriele Morabito; Article No. 1; pp. 1:1–1:15

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Safety-critical systems have long been integral to various sectors, including aviation, nuclear power generation, healthcare, and more. In today's world, these systems are even more prevalent in our daily lives with the development of autonomous systems. Therefore, it is crucial to ensure that the development and deployment prioritise safety and adhere to rigorous standards to protect occupants, pedestrians, and workers on industrial environments.

High Performance Computing (HPC) platforms are those which are designed to accelerate data processing to solve complex and computationally intensive problems, some architectures include the use of Graphics Processing Units (GPUs), Field-Programmable Gate Arrays (FPGAs) or Tensor Processing Units (TPUs). These accelerators are increasingly establishing their presence on these sectors by delivering accelerated computations within real-time constraints. However, these heterogeneous systems have inherent limitations, with GPU schedulers exhibiting a black-box behaviour. Additionally, a well-known issue is the bottleneck that arises in the data transfer process between the host system (CPU side) and the device (accelerator side), leading to unpredictable data access times. Over the years, independent authors and companies have dedicated their efforts to develop memory allocation algorithms and architectures specifically aimed at achieving faster data accessing and deterministic behaviour of GPUs.

To ensure the reliability and uniformity of outcomes, this paper conducts an evaluation of various allocation methods using the Rodinia benchmark suite [10, 11, 18, 1]. The assessment encompasses both static and dynamic attributes extracted from the benchmarks, employing tools offered by NVIDIA and other contributors. Static metrics, such as data size, the number of allocations, copies, kernel launches, cache hit rate, and memory coalescence, are taken into account. Moreover, the dynamic aspects inherent in any code, such as allocation, data copying, kernel launching API usage, and overall execution time, are meticulously documented to discern potential connections with the static ones.

A significant enhancement to the benchmark analysis includes a desirable feature for safety-critical systems, the ability to control the timing of memory allocations. Leveraging the *XeroZerox* a tool introduced by [8], that allows that the entire memory allocation is executed only once and exclusively at the beginning of the execution for the NVIDIA allocation methods, we were able to achieve that behaviour. That tool has support to create this memory pool using Unified Memory (UM) and zero-copy (ZC). In this work, we add support to the traditional allocation method to grant an equal comparison between all results gathered from each memory configuration. Our analysis aims to provide conclusions regarding the most suitable allocation method based on the identified program characteristics, regarding mean measurements for those metrics to reveal the swiftest method, reinforced with assessments of standard deviation and histogram representations to gauge predictability, thereby providing valuable insights for informed decision-making.

The organisation of this paper is as follows: Section 2 introduces the most relevant previous works that inspired this study. Section 3 presents the memory managing methods that have been selected to conduct the time analysis on memory accesses that is presented in this work. Section 4 describes *XeroZerox* highlighting our modifications. Section 5 describes the static characteristics and dynamic metrics used for evaluating how they are related and affect the timing response. Section 6 exposes the data treatment, the value extracted from executing every benchmark with different memory configuration and a comparative analysis extracted from the results. Finally, Section 7 summarises the most important ideas and outline future work to be undertaken.

2 Related Work

In this section we present the prior work in the field, aiming to contextualise our research, identify gaps, and build on established frameworks. We identify two categories of memory management, a) studies which aim into new allocation strategies and b) studies which extract conclusions from reverse engineering the GPU. Our work is influenced by two streams: studies focused on new allocation strategies and those aimed at understanding memory through reverse engineering. We employ these techniques to get metrics, using tools designed to alter memory behaviour, to highlight which characteristics are relevant during program execution.

2.1 Allocation strategies

A. Calderón introduces a tool named *XeroZerox*, which is specifically designed for embedded platforms [8]. This open source tool carries out a two-phase analysis and modification process on the source code. In the first phase, it intercepts explicit GPU memory allocation calls and copies creating a mapping between CPU and GPU variables. In the second phase, leveraging this information, it strategically replaces the allocation method sections, opting for UM or ZC allocation instead of the default GPU memory allocation method. *XeroZerox* also creates a pool of memory to allocate all the data at once at the beginning of the process and free it at the end, being a desirable behaviour for safety-critical applications, first for avoiding the timing overhead and non-time deterministic nature of the memory allocations which can impact the worst case execution time of the program, as well as to ensure that the size of the memory allocations is fixed, and therefore can always be satisfied at program deployment.

Another strategy is proposed by Sven Widmer et al. [23]. They developed an allocator focused on enhancing SIMD scalability for small, frequent memory allocations, minimising branch divergence. The system-wide default allocator performs well with few simultaneous requests, and this approach optimises data accesses by utilising one superblock shared among warp threads. A voting mechanism selects a worker thread, reducing simultaneous memory requests and invocations. This design eliminates the need for superblock header data, streamlining memory allocation and minimising synchronisation and memory overhead. Aggregating memory requests within a warp ensures efficient cache utilisation, aligning with the goal of minimising the use of the default allocator for improved performance.

Another approach is described by Andrew Adinetz [3]. HAlloc is a statically sized memory pool, which is subdivided into chunks during initialization, while the handling of large allocations relies on the CUDA dynamic memory allocator. For each allocation size, only one active bin from which to allocate is kept by HAlloc. When a configurable threshold for usage within a bin is reached, it is replaced with a new active bin, maximizing the chances of subsequent allocations finding an available block in the active bin. Lists of bins that are almost-exhausted and almost-empty are also kept by HAlloc for each size. Bins are moved between these two lists during free operations, and bins in the almost-empty list are used to select new active bins when needed. Per-size bins are also maintained by their fine-grained allocator, but a linked list is used to track all active bins, avoiding costly active bin replacement operations.

Zaid Qureshi et al. [19] develop the BaM System (Big accelerator Memory). This tool allows programmers to access big data sets which exceed the GPU memory available in the system, by accessing data stored in storage devices in an on-demand and fine-grained manner, while improving the access time. The authors call this "accelerator-centric" architecture. Threads on GPU can bring data wherever it is stored, either on CPU or any other storage system. This reduces the use of page-faulting mechanism from CPU and it is demonstrated using NVMe SSDs.

In terms of predictability, Björn Forsberg et al. [14] show how to enhance cache hit rate through the adept management of prefetching and evicting using Predictable Execution Models (PREM). To enhance predictability, they introduce a division into a memory phase and a compute phase, a strategy akin to that of XeroZerox. They leverage a "replacement policy" that "selects which data to evict when new data is requested".

2.2 Reverse engineering

On reverse engineering, Jake Choi et al. [13] performed a comparative analysis between UM and ZC in relation to the traditional *cudaMalloc* over a NVIDIA Jetson TX2 SoC, an embedded platform based on Pascal architecture. By evaluating those allocators against three benchmarks of the Rodinia suite, they extract and compare metrics such as memory usage and execution time. Their study concludes that the traditional method should not be the default choice for programmers. In fact, the optimal allocator depends on the specific application being considered.

Calderón et al. [6] created a tool for the reverse engineering of the default CUDA memory allocator in terms of functionality and timing behaviour. They showed that similar to CPU allocators, the CUDA allocator works with power of two bin sizes and that GPU memory allocations which fall into a newly allocated or reallocated bin affects the execution time of the subsequent GPU kernel call. Their open source tool is able to extract the bin sizes and memory pools of NVIDIA GPUs and has been demonstrated with both desktop and embedded GPUs. Moreover, later the tool was extended to OpenCL and non-NVIDIA GPUs [7].

3 Allocation Methods

Over time, there has been a concerted effort from both industry players and academic researchers to devise quicker algorithms, all with the common goal of enhancing data access. In the following we provide a succinct overview of the algorithms under evaluation. It's worth noting that these particular algorithms were selected for their seamless integration and straightforward implementation on our embedded GPU platform i.e. open source availability of their code and compatibility with embedded NVIDIA GPUs. Interestingly, the first three algorithms hail from the research and development efforts of NVIDIA, while the fourth stems from an independent researchers' work.

CUDA traditional allocation is the native allocator of CUDA C programming language [12]. It is used through *cudaMalloc*, which allocates device global memory of a specified size. The operating system looks for space in the memory pool using one of the following policies: first-fit, best-fit, worst-fit or next-fit. Still, data movements must be explicitly done using *cudaMemcpy*.

Zero-Copy. In order to transfer data from host (CPU) to device (GPU) or in the reverse direction, the memory has to be copied to a page-locked buffer. This process can be avoided by allocating paged-locked (aslo known as pinned memory) with *cudaMallocHost* or *cudaHostAlloc* calls, so data can be directly accessed by the device using DMA (Direct Memory Access). Enabled by Unified Virtual Addressing (UVA) released on CUDA 4, it makes data accessible through PCI-e avoiding *cudaMemcpy* calls.

Unified Memory. Supported since CUDA 6, this allocation method [15] joins both Central Processing Unit (CPU) and GPU memory address spaces in a single one. Using *cudaMallocManaged* function, data is allocated into that space, returning a pointer accessible both from host and device. Similar to ZC there is no need of explicit copy declaration because the system migrates data automatically.

ScatterAlloc. This algorithm [22] organises a memory pool into a structure called *Super Block*. This structure has a fixed size of memory that is also split into equal size pages. Its author also proposes a method to keep track of free memory in two levels of hierarchy. At higher level a *Page usage table* is used that stores which pages are in use and freed. At lower level, inside each page, there is another table which keeps track of chunks within a page. In order to make allocations faster, the algorithm changes the storing region when 90% of the pages are filled. In addition, the author introduces an approach to reduce simultaneous access from different threads to the same memory region.

4 Memory analysis and reconfiguration

To introduce a safety framework and enable consistent comparisons between allocation methods, this work uses the *XeroZerox* tool [8] developed by A. J. Calderón mentioned in Section 2, extended with support to traditional memory allocation to fit the needs of our analysis. This tool offers numerous benefits: it allows us to modify memory models without altering the original benchmark code. Additionally, it supports the use of a single, preallocated memory pool at the program start-up. This is a highly desirable feature for safety-critical systems, as it enables control over memory reservation and the ability to calculate the worst case execution time of this process. In the utilisation of the *XeroZerox* tool for assessing allocation methods, a critical observation emerges: the original tool initially overlooked the case of the traditional allocation model. This discrepancy poses a substantial challenge to achieving a comprehensive comparison among allocation strategies. In this work we address this issue in order to perform a fair comparison. Next, we delineate our methodology for utilising the *XeroZerox* tool and address the gap it initially presents concerning the traditional allocation model. To enhance a robust comparison between allocation methods, we have devised an approach to incorporate this gap within our analysis framework.

XeroZerox analyses GPU applications and determines the size of a centralised memory pool that can serve all its needs. This pool is allocated at the beginning of the application using zero-copy or unified memory allocation and it is released upon its completion. Serving as a sub-allocator, *XeroZerox* intercepts traditional memory allocations and substitutes them with allocations served consecutively – i.e. similar to a bump allocator – from the centralised memory pool, with minimal and constant runtime cost. This approach accommodates legacy GPU applications within critical setups' memory management constraints, without any code modifications. Moreover, *XeroZerox* priorities minimising memory consumption, effectively reducing both the memory footprint and the runtime overhead associated with memory management for these applications.

To intercept the target memory functions, the analysis library employs a technique referred to in the literature as *interposition* [9]. This method involves substituting the target functions with user-defined wrapper functions. These wrappers serve to augment the original functions with additional functionality, such as extracting information from their arguments, which is particularly pertinent to our objectives. The analysis library is executed only the first time of executing the application and generates a comprehensive report including details like the maximum memory utilisation, the count of memory pool instances generated, the

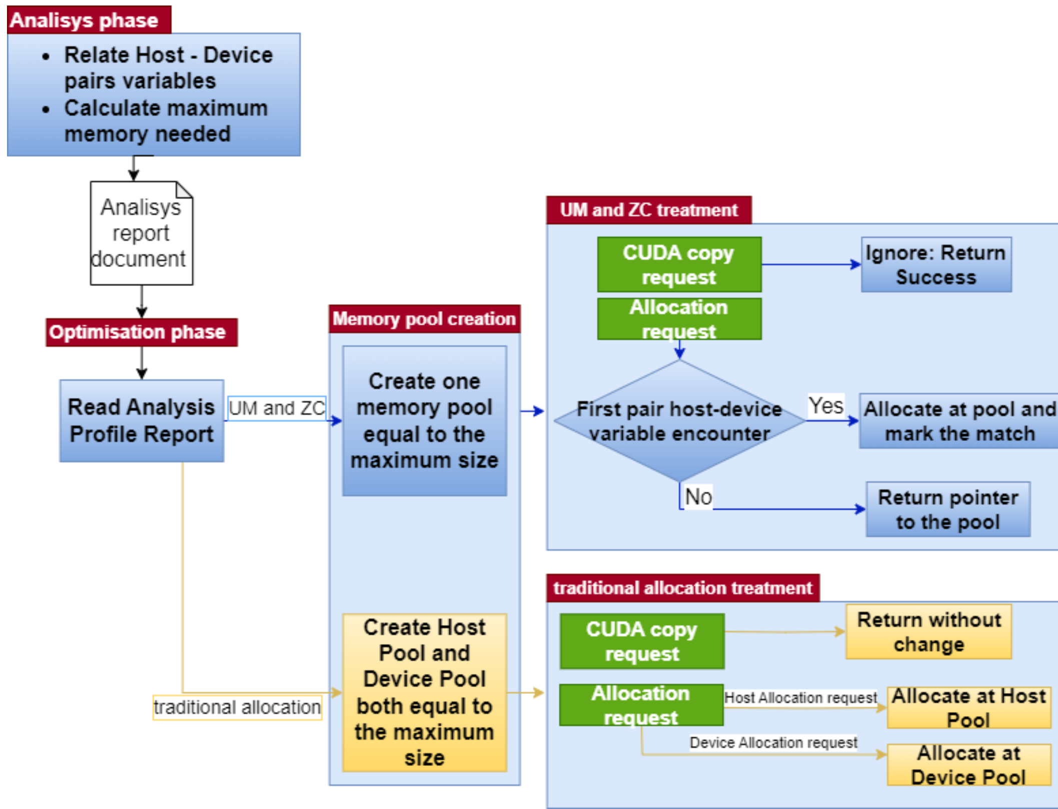

Figure 1 XeroZerox tool behaviour.

frequency of memory transfers and the detection of any memory leaks. On a second phase, which is the only one used for subsequent executions, eg. at deployment, once the centralised memory pool is established in the initialisatiton of the GPU application, *XeroZerox* assumes the role of a sub-allocator, handling allocation requests from the application in accordance with the matches loaded from the optimisation profile. Notably, *XeroZerox* adopts a strategy where it fulfils memory allocation requests solely for the initial request it receives. Subsequent allocation requests prompt *XeroZerox* to return a pointer to the memory region already allocated for the preceding request, effectively optimising memory utilisation by reusing allocated space. This approach minimises redundant allocations and contributes to more efficient memory management within the application.

In this paper, we extended *XeroZerox* with support for the traditional allocation method. We took advantage from the analysis phase, to identify the CUDA calls. At this point, instead of creating just one memory pool to be allocated using ZC (Zero Copy) or UM (Unified Memory) method at the beginning of the optimisation phase, host and device pools are created separately as it is performed when the default CUDA allocation method is used. During an allocation call, the tool discerns whether it is a GPU or CPU allocation, storing the data in the corresponding pool through a linked list. Additionally, the incorporation of support for copy calls has been essential due to the existence of two distinct allocated pools. This ensures that whenever such a call is activated, the tool can reference the variables to the pre-allocated pools, facilitating the migration of data.

As illustrated in Figure 1, the dark blue boxes denote components of the original *XeroZerox* tool, representing the baseline framework. Whereas, the yellow boxes highlight additional elements we added to ensure a fair comparison among different allocation methods.

5 Benchmark's characterisation

The Rodinia benchmark suite is a collection of parallel applications designed to evaluate the performance and scalability of computer systems, particularly those with multicore processors or GPU accelerators. It was developed by researchers at the University of Virginia as a resource for evaluating and comparing different parallel computing architectures.

This benchmarking suite is one of the most widely used GPU bencmarking suites used in the literature. Since our purpose is to analyse the general memory allocation behaviour of GPU software and find out the optimal memory allocation method for use in safety critical systems, we intentionally avoid using a GPU benchmarking suite targeting explicitly safety critical systems like GPU4S Bench and OBPMark, wich have a predictable, easy to analyse memory allocation behaviour, similar to the one achieved by *XeroZerox*.

Rodinia contains 23 benchmarks, characterised by their computation patterns, named *Dwarfs* by Paul Springer [21] and introduced by Asanovic et al. [4]. They refer to recurring structures or strategies used to solve problems and perform tasks in computational systems. These patterns provide standardized ways to approach common computational problems, making it easier to design, implement, and understand. Despite their utility in classification, our analysis indicates that these patterns do not have any relevance on our results. Furthermore, during the execution of 13 out of 23 Rodinia benchmarks, runtime errors or system crashes were encountered, causing the hardware to reboot. To maintain the integrity of the results, we opted not to modify the benchmark code to force compatibility with the ARM-based embedded system used. Those evaluated are presented in Table 1 along with the name of the benchmark and the data loaded is provided by Rodinia's developers.

Table 1 Evaluated Rodinia benchmarks.

Applications	Dwarfs	Datasets
Back Propagation	Unstructured Grid (UG)	65536 elements
Gaussian Elimination	Dense Linear Algebra (DLA)	3x3 matrix and 1024x1024 matrix
LU Decomposition	Dense Linear Algebra (DLA)	64x64 matrix and 2048x2048 matrix
Kmeans	Dense Linear Algebra (DLA)	100 and 819200 elements x 34 columns
Breadth-First Search	Graph Traversal (GT)	1 million nodes and 4096 nodes
SRAD_v1	Structured Grid (SG)	512x512 image
Hotspot	Structured Grid (SG)	Two square matrices of 64x64, 512x512 and 1024X1024 each
Heart Wall	Structured Grid (SG)	104 frames: 609x590 pixels
Leukocyte	Structured Grid (SG)	600 frames: 640x480 pixels
Myocyte	Structured Grid (SG)	16 parameters 1 instance 100 ms

Taking into consideration prior research, we have curated a set of characteristics that define a program for our experimental setup, categorized into static and dynamic metrics. Static metrics provide intrinsic insights into program structure, encompassing cache hit rates, shared memory usage, coalescence, memory access frequency, and data transfer sizes between CPU and GPU. On the other hand, dynamic metrics focus on timing operations, including overall program execution time, kernel execution time, CUDA APIs execution time, and memory operations execution time. This comprehensive approach enables a thorough evaluation of program performance across various dimensions, facilitating informed comparisons between different allocation methods and program configurations. Crossing results from both categories is the key to obtain conclusions of how program characteristics influence its timing results. The following subsections give a deeper understanding of these categories.

5.1 Static metrics

These metrics provide intrinsic insights into program structure, highlighting how a program is coded and its inherent characteristics. These metrics are crucial for understanding the efficiency and resource usage of a program. The following points explain each static metric in detail:

1. **Cache Hit Rates:** This metric assesses the efficiency of data retrieval from cache memory. A high cache hit rate indicates that most data requests are satisfied by the cache, leading to faster data access and improved performance. Analysing cache hit rates helps in optimizing memory hierarchy and reducing latency.
2. **Shared Memory Usage:** This metric evaluates the utilization of shared memory resources within the system. Efficient use of shared memory can reduce global memory accesses and increase the speed of data processing. Understanding shared memory usage is key to optimizing memory allocation and parallel processing capabilities.
3. **Coalescence:** This metric examines the degree to which memory accesses are coalesced, which optimizes data transfer efficiency by grouping multiple memory requests into a single transaction. High coalescence reduces the number of memory transactions, improving bandwidth utilization and reducing latency.
4. **Memory Access Frequency:** This metric quantifies the frequency of memory accesses, indicating the demand for data retrieval during program execution. High memory access frequency can highlight potential bottlenecks and guide optimizations to minimize redundant memory operations and enhance overall performance.
5. **Data Transfer Sizes Between CPU and GPU:** This metric measures the volume of data exchanged between the central processing unit (CPU) and the graphics processing unit (GPU). Large data transfers can introduce significant overhead and latency. Understanding and optimizing data transfer sizes are crucial for improving inter-device communication and overall program efficiency.

5.2 Dynamic metrics

On the other hand, dynamic metrics focus on timing operations, providing insights into various aspects of program execution. The following points explain each dynamic metric in detail:

1. **Overall Program Execution Time**: This metric captures the total duration from the start to the end of the program's execution.
2. **Kernel Execution Time:** This metric measures the time taken to execute computational kernels, which represent the core processing tasks of the program. Analysing kernel execution time helps in understanding the efficiency of the computational workload and identifying areas for optimisation within the kernels.
3. **CUDA APIs Execution Time:** This metric examines the time spent executing CUDA Application Programming Interface (API) calls. These calls manage GPU resources and operations, so their execution time reflects the overhead associated with GPU management. Specifically, this includes the time taken for kernel launches, memory allocations, data transfers, and, in some cases, synchronization operations.
4. **Memory Operations Execution Time:** This metric measures the duration required for memory read and write operations. It is indicative of data transfer efficiency and memory access latency. Optimizing memory operations execution time can significantly enhance overall program performance, particularly in data-intensive applications.

6 Experimental Results

In this section, we present the results of testing various allocators on the Rodinia benchmark suite using their standard input set. Subsequently, in the following sections of this section, graphical representations of the data are provided to facilitate a time comparison between the selected allocators for each metric across all benchmarks. This analysis examines the performance characteristics, strengths, and weaknesses of each allocator in relation to the benchmarks' nature.

Due to the extensive volume of collected data, we have opted to store it in a dedicated GitHub repository [2]. Detailed information from each of the 500 iterations, as well as summarised characteristics in Excel files, can be accessed through this repository. This approach allows for transparency and facilitates access to the comprehensive dataset for those interested in further analysis or replication of the study.

6.1 Experimental setup

For each allocation method outlined in section 3, we performed multiple iterations of each benchmark on an NVIDIA Jetson Orin AGX, an embedded GPU platform certified for use in the automotive sector. Due to the inherent variability in GPU execution times, as discussed in section 5, each benchmark was executed 500 times with the same data inputs. This number of runs allows for a reliable calculation of the mean and standard deviation for each selected parameter, ensuring statistically meaningful results. The variability in execution times was verified through time histograms, which support the assumption of a Gaussian distribution. These histograms are available for review on our GitHub repository [2]. In contrast, obtaining static metrics was more straightforward, as these remain consistent regardless of when the data collection tool was launched or the memory configuration selected.

Only memory access methods interacting with *XeroZerox* have been studied, using the unmodified benchmarks as baseline. This approach was chosen due to a key feature of *XeroZerox* that aligns it with safety-critical systems: memory allocation is managed by ensuring that all allocations occur at the start of the execution.

Using the *NVIDIA NSight Systems* and *NVIDIA NSight Compute* tools alongside each benchmark, we generated a timing report containing dynamic metrics and a characterization report featuring static metrics. The benchmarks from Rodinia were compiled using CUDA version 11.4 and automated with Python scripts version 3.8.10. Metric extraction was performed using *NVIDIA NSight Systems* version 2022.4.2.1 and *NVIDIA NSight Compute* version 2023.2.0.0 build 32895467.

6.2 Profiling

The initial set of static characteristics was extracted from the profiling tool *NVIDIA NSight Systems*, and the results are presented in Table 2. This table displays basic static characteristics, such as the number of allocations and deallocations performed using *cudaMalloc* and *cudaFree*, the number of copies made by *cudaMemcpy*, the quantity of kernels launched, and the size of the data moved. The data related to other native allocation instructions, like *cudaBindTexture* or *cudaMemcpyToSymbol*, has been retained in its original form. This decision is grounded in the belief that these instructions offer essential functionality required by programmers. To aid interpretation, certain cells in the table have been colour-coded and star-marked: blue cells (*) are the value believed to be the edge on the allocator choice, while a green background (**) represent values that exceeded the threshold established for influencing allocator selection, this is founded on the conclusions presented in the following subsection.

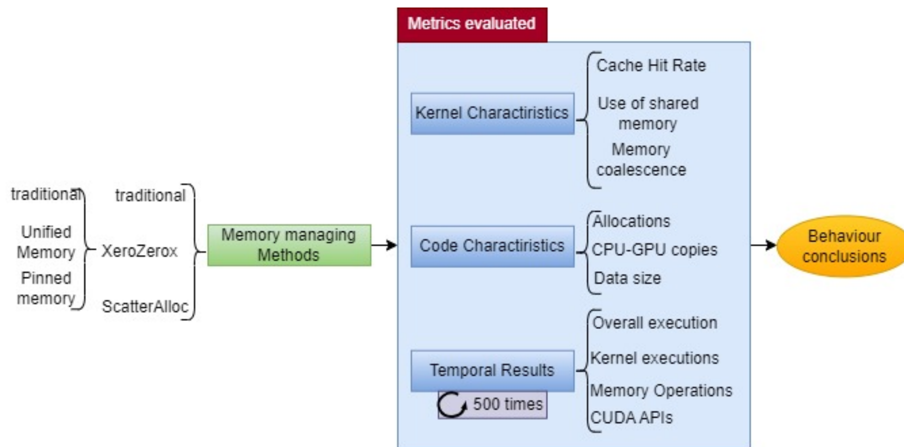

Figure 2 Metrics extracted for every evaluated benchmark.

During our work, the necessity of acquiring a deeper understanding of the behaviour of each benchmark was recognised. For this reason, we have made use of *NVIDIA NSight Compute* to acquire a second set of static metrics related to kernels. To make sure that the allocation method does not interfere with these results by comparing the kernels' characteristics, binaries run from a clean build and from binaries interfered with *XeroZerox*, which are shown in Table 3. In three benchmarks (*gaussian* with 1024 size matrix, *srad_v1* and *myocyte*) the NVIDIA's tool could not complete the analysis. For these two cases the report was empty despite how many times we run the test. We suspect that this might be due to the high number of kernels launched, due to this limitation we could not provide information about these tests.

We observed an anomalous L2 cache hit rate for kernel 2 in the *leukocyte* benchmark, consistently reported by the *NVIDIA NSight Compute* tool, despite multiple reruns of the benchmark. Since we lack access to the internal workings of this tool, we are unable to provide a definitive explanation for this irregularity.

At the core of our analysis lies a meticulous process of data collection of the dynamic metrics during each iteration of the benchmarking procedure. This data, intricately tied to the associated process and variable, serves as the foundation for subsequent calculations of mean and standard deviation. An example of results from those executions is illustrated in Figure 3 for API's executions and in Figure 4 for memory and kernel operations.

Figure 3 represents a benchmark with a specific input set size. Vertical axis (y) represents the normalised time consumed on running every API taking as baseline benchmarks without being modified with a value of 1.00, called *traditional alloc* at the picture. The x axis represents each allocation method evaluated, from left to right: traditional alloc, traditional *cudaMalloc* with XeroZerox optimisation, *ScatterAlloc*, *ZC* with XeroZerox optimisation and *UM* with XeroZerox optimisation. The z axis represents every API call considered relevant, from front to rear: Kernel launch time, allocation API (every method has its own), copy API, synchronisation API (not every benchmark uses this instruction) and overall execution time.

On the other hand, Figure 4 illustrates kernel and memory operations execution times. Similar to the representation used in the APIs figures, these figure differ in that one of the horizontal axes has been adjusted to show which operations are being evaluated. Each kernel call and the normalised time it takes to transfer data from the CPU to the GPU and vice versa are depicted, using the traditional allocation method as a baseline with a value of 1.00. For UM and ZC, the transfer time is not included due to the unique behaviour of these methods, as explained in Section 3.

Table 2 Benchmark Characteristics.

Dwarf	Benchmark	Data	Allocations	Copies	Kernels launched	Data sizes
DLA	Gaussian	Matrix 3x3	3	6	4	$\leq 64kB$
		Matrix 1024x1024	3	6	2046 **	4 MB, $\leq 64kB$ *
	LUD	Matrix 64x64	1	2	10	$\leq 64kB$
		Matrix 2048x2048	1	2	382 **	16,777 MB **
	Kmeans	List 100x34 elements	4	7	3	$\leq 64kB$
		List 819200x34 elements	4	7	3	11.4 MB, 3,277 MB **
SG	srad_v1	512x512 image	12	206 **	502 **	0.920 MB , $\leq 64kB$
	hotspot	Two squared matrix 64x64	3	3	1	$\leq 64kB$
		Two squared matrix 512x512	3	3	1	1.049 MB
		Two squared matrix 1024x1024	3	3	1	4.149 MB *
	heartwall	104 frames: 609x590 pixels	623	50	20	1.952 MB , $\leq 64kB$
	leukocyte	600 frames: 640x480 pixels	34	39	7	0.561 MB , 0.472 MB , $\leq 64kB$
	myocyte	16 parameters 1 instance 100 ms	4	16500 **	3900 **	$\leq 64kB$
UG	backpropagation	65536 elements	6	8	2	4.457 MB , 0.262 MB , $\leq 64kB$ *
GT	BFS	4096 nodes	7	23	16	98 kB , $\leq 64kB$
		1 million nodes	7	31	24	24 MB , 8 MB , 4 MB , 1 MB , $\leq 64kB$ **

Complete information is contained in a Github repository [2], where Excel documents contain detailed information of the mean and standard deviation arranged in folders ordered by DWARFS alongside with 3D chart representations to condense data.

6.3 Comparative Analysis

In this subsection we discuss the results extracted from the correlation between static and temporal analysis, taking two distinct scenarios. The first revolves around an analysis just concerning the overall execution time. In contrast, the second is oriented to those systems characterised by continuous operation, like safety-critical systems responsible for monitoring the environment from startup to the moment the machine is shutdown. These systems typically entail a single allocation at initialisation, followed by recurrent data movement and kernel launches throughout operation, culminating in memory deallocation upon shutdown.

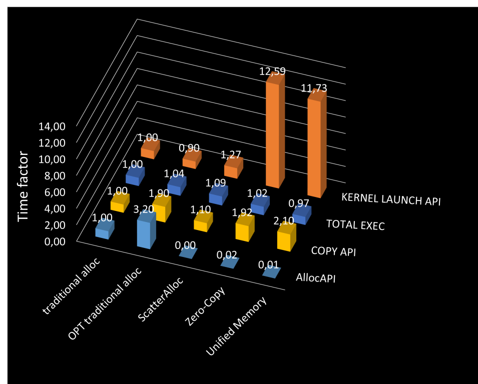

Figure 3 APIs-hotspot 64x64 matrix.

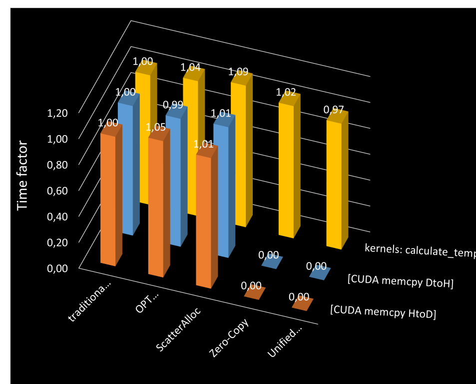

Figure 4 Mem/Kernel Ops-hotspot 64x64.

Table 3 Kernels Characteristics.

Dwarf	Benchmark	Data	L1 cache hit rate	L2 cache hit rate	Coalescent memory	Excesive sectors accessed
DLA	Gaussian	Matrix 3x3	kernel 1: 33,33% kernel 2: 60%	kernel 1: 56,77% kernel 2: 66,47%	Uncoalesced Global Accesses	kernel 1: no kernel 2: 2 (20%)
		Matrix 1024x1024	no data	no data	no data	no data
	LUD	Matrix 64x64	kernel 1: 48,39% kernel 2: 39,24% kernel 3: 25%	kernel 1: 40,51% kernel 2: 42,87% kernel 3: 61,14%	Uncoalesced Shared Accesses	kernel 1: 562 (41%) kernel 2: 6216 (70%) kernel 3: no
		Matrix 2048x2048	kernel 1: 48,39% kernel 2: 58,22% kernel 3: 53,29%	kernel 1: 40,51% kernel 2: 49,38% kernel 3: 64,42%	Uncoalesced Shared Accesses	kernel 1: 562 (41%) kernel 2: 263144 (70%) kernel 3: no
	Kmeans	List 100x34 elements	kernel 1: 82,08% kernel 2: 86,49%	kernel 1: 54,79% kernel 2: 27,73%	Uncoalesced Global Accesses	kernel 1: 3009 (70%) kernel 2: no
		List 819200x34 elements	kernel 1: 52,78% kernel 2: 41,07%	kernel 1: 71,98% kernel 2: 67,62%	Uncoalesced Global Accesses	kernel 1: 24371200 (78%) kernel 2: no
SG	srad_v1	512x512 image	no data	no data	no data	no data
	hotspot	Two squared matrix 64x64	16,13%	62,54%	Uncoalesced Global Accesses	864 (27%)
		Two squared matrix 512x512	4,65%	68,69%	Uncoalesced Global Accesses	67192 (31%)
		Two squared matrix 1024x1024	3,54%	68,67%	Uncoalesced Global Accesses	273184 (31%)
	heartwall	104 frames: 609x590 pixels	95,48%	97,14%	Uncoalesced Global Accesses	9958 (19%)
	leukocyte	600 frames: 640x480 pixels	kernel 1: 99,23% kernel 2: 98,78% kernel 3: ≈ 98%	kernel 1: 22,51% kernel 2: 172,91% kernel 3: ≈ 80%	Uncoalesced Global Accesses	kernel 1: 3051 (19%) kernel 2: 122584 (88%) kernel 3: ≈ 580000 (18%)
	myocyte	16 parameters 1 instance 100 ms	no data	no data	no data	no data
UG	backprop	65536 elements	kernel 1: 58,30% kernel 2: 71,18%	kernel 1: 52,52% kernel 2: 54,17%	Uncoalesced Global Accesses in both kernels	kernel 1: 73728 (18%) kernel 2: 163847 (15%)
GT	BFS	4096 nodes	kernel 1: 19,40% kernel 2: 7,19%	kernel 1: 50,19% kernel 2: 49,48%	No coalescence warning	
		1 million nodes	kernel 1: 0,04% kernel 2: 0,01%	kernel 1: 0,91% kernel 2: 0,93%	No coalescence warning	

6.3.1 Analysis regarding overall execution time

By correlating static results (explained in section 5) and dynamic results from the time analysis, we derived programming guidelines for GPU usage, presenting distinct conclusions based on overall execution time and considerations more pertinent to safety-critical systems. Generally speaking, the traditional method with the *XeroZerox* optimisation out stands over the other allocation methods regarding the total execution time of the benchmark. Additionally, there are other characteristics that had been seen relevant to choose a method of memory management. It appears that the most relevant one is the cache hit rate. This can be observed in the cases of both the *heartwall* and *leukocyte* benchmarks, where the hit rates for both L1 and L2 caches exceed 90%. In such cases, the traditional allocation method without optimisation yields the lowest overall execution times.

In general, when a substantial number of kernels is launched, and some data involved in copies that exceed 4 MB, the zero-copy method with *XeroZerox* proves to be the optimal allocation method, resulting in the lowest overall execution times. Conversely, when the number of copies and kernel launches is small, and the data copied does not exceed 4 MB, both zero-copy and unified memory yield similar results, with unified memory demonstrating the lowest execution times for sizes below that threshold.

There are, however, exceptions. For instance, the *BFS* benchmark should have a clear allocator preference. When using 4096 nodes, unified memory is preferred, while when loading 1 million nodes, zero-copy performs better. Remarkably, the execution times are

very similar for both allocators, with the size having only a minor impact. This benchmark is unique in that it exhibits coalescent memory, thereby minimising excessive sector accesses, which could explain the observed behaviour.

Given these observations, benchmarks such as *gaussian* (loading a 3x3 matrix), *LUD* (loading a 64x64 matrix), *kmeans* (loading 100x34 elements), and *hotspot* (loading a 64x64 matrix) were expected to behave similarly. However, temporal results reveal that *hotspot* and *LUD* perform better with unified memory, while *kmeans* and *gaussian* yield better overall execution times with zero-copy. It was found that the first couple make use of shared memory and experience a higher number of warps stalled compared to the latter two benchmarks.

Below, observations are presented concerning overall execution time, categorized by their respective levels of importance, with the highest level of importance listed first. Each observation is paired with a recommended allocator:

1. L1 and L2 cache hit exceed 90% -> traditional allocation method
2. Coalescent memory -> zero-copy + XeroZerox or Unified Memory + XeroZerox
3. When a substantial number of kernels (measured at 2 in blue or green) are launched & data size greater or equal than 4 MB -> zero-copy + XeroZerox
4. When number of kernels launched is small & data size less or equal than 4 MB -> Unified Memory + XeroZerox
5. Use of shared memory -> Unified Memory + XeroZerox
6. High number of warps stalled -> zero-copy + XeroZerox

6.3.2 Analysis regarding kernels and copies execution times

Another interpretation of the results can be made if the programmer is looking to adapt the code into safety-critical systems. One important recommendation is to perform memory allocations only once at the beginning, a behaviour achieved automatically with the *XeroZerox* optimisation. Moreover, memory freeing is not a relevant characteristic in this case, because the expected execution is that the system runs continuously, reading and writing data, moving it between CPU and GPU and executing functions and kernels. So considered metrics here are the time that the copy and kernel launch APIs are active, and the kernel and copy execution times.

Upon examining the active time of the copy API and the kernel launch API, it's observed that **conventional method without XeroZerox intervention and ScatterAlloc** typically emerge as the faster options. However, there are instances where other methods yield similar times. The observed behaviour aligns closely with the previous findings, with one notable exception found in the *backpropagation* benchmark. Here, both ZC and UM methods show comparable performance to the traditional method.

When looking at copying operations, the first thing noticed is that there is no data when using ZC or UM, presumably because there is no explicit copy between CPU and GPU, so the delay depends on the related API. On the other cases, the time grows proportionally with the size copied being in any case in the same magnitude order, no special correlation is found here.

To verify the accuracy of the mean and standard deviation as appropriate measures, histograms were generated for each process, also available on the mentioned GitHub repository [2]. Notably, the traditional allocation method exhibited considerable variation in results, suggesting a lack of consistency. As a result, **it is advisable to avoid the traditional allocation method** in systems targeting safety-critical or real-time scheduling, regardless of the mean results. Also when employing the traditional allocation method in any form, the

histograms for copying and allocating APIs exhibited a multi-modal distribution, while ZC and UM usually exhibit one gaussian bell shape, so in terms of predictability it is advised to use these last two methods in case that *ScatterAlloc* is not the preferable method.

7 Conclusion

In this work, we analysed the dynamic memory behaviour of GPU programs for safety critical systems, targeting the widely used suite GPU benchmark suite, Rodinia. Studying all this data has revealed some reasons behind the allocator's behaviour have been identified through a static analysis of the benchmarks.

The specific characteristics and requirements of each benchmark and kernel influence the choice of GPU memory allocation method. Factors such as cache hit rates, data sizes, and the number of kernel launches may play a crucial role in determining which allocation method is the most suitable for a given scenario.

For future research and validation of the conclusions presented in this work, the following avenues can be explored:

1. Microbenchmarks for Specific Scenarios: Conducting microbenchmarks designed to test each specific scenario and characteristic identified in this research can provide a more detailed and comprehensive validation of the conclusions. This can help in fine-tuning allocation methods for precise use cases.
2. GPU Direct RDMA [16] and GPUDirect Storage [17]: Investigating the use of NVIDIA's GPUDirect RDMA and GPUDirect Storage methods represents a promising direction. These technologies facilitate direct GPU access to data stored in storage units such as SSDs, circumventing CPU involvement. This approach holds the potential to deliver substantial performance benefits and minimize latency in data-intensive applications. The work of J. Bakita et al.[5] is directly relevant to this topic and can be utilised to propel advancements in this area.
3. Evaluation on other platforms: Consider how these allocation methods perform on different GPU architectures and platforms, as compatibility and performance can vary.

References

1 Rodinia: Accelerating compute-intensive applications with accelerators, 2018. URL: `https://rodinia.cs.virginia.edu/doku.php`.

2 Mars-data, 2024. URL: `https://anonymous.4open.science/r/MARS-data-568D/README.md`.

3 Andrew V Adinetz. Halloc: a high-throughput dynamic memory allocator for gpgpu architectures, 2014.

4 Krste Asanovic. The landscape of parallel computing research: A view from berkeley. Report, Electrical Engineering and Computer Sciences University of California at Berkeley, 2006.

5 Joshua Bakita. Enabling GPU memory oversubscription via transparent paging to an NVMe SSD*. *Real-Time Systems Symposium*, 2022.

6 Alejandro J. Calderón, Leonidas Kosmidis, Carlos F. Nicolás, Francisco J. Cazorla, and Peio Onaindia. Understanding and exploiting the internals of GPU resource allocation for critical systems. In David Z. Pan, editor, *Proceedings of the International Conference on Computer-Aided Design, ICCAD 2019, Westminster, CO, USA, November 4-7, 2019*, pages 1–8. ACM, 2019. `doi:10.1109/ICCAD45719.2019.8942170`.

7 Alejandro J. Calderón, Leonidas Kosmidis, Carlos F. Nicolás, Francisco J. Cazorla, and Peio Onaindia. GMAI: understanding and exploiting the internals of GPU resource allocation in critical systems. *ACM Trans. Embed. Comput. Syst.*, 19(5):34:1–34:23, 2020. `doi:10.1145/3391896`.

8 Alejandro J. Calderón. *Real-Time High-Performance Computing for Embedded Control Systems*. Thesis, Universitat Politècnica de Catalunya, 2022.

9 A. Chatterjee. Function interposition in c with an example of user defined malloc, 2017. URL: https://www.geeksforgeeks.org/function-interposition-in-c-with-an-example-of-user-defined-malloc.

10 Shuai Che. Rodinia: A benchmark suite for heterogeneous computing, 2009. doi:10.1109/IISWC.2009.5306797.

11 Shuai Che. A characterization of the rodinia benchmark suite with comparison to contemporary CMP workloads, 2010. doi:10.1109/IISWC.2010.5650274.

12 John Cheng. *Professional CUDA C Programming*. John Wiley & Sons, Inc., 2014.

13 Jake Choi. Comparing unified, pinned, and host/device memory allocations for memory-intensive workloads on Tegra SoC. *Concurrency and Computation: Practice and Experience*, 2020. doi:10.1002/cpe.6018.

14 Björn Forsberg, Luca Benini, and Andrea Marongiu. *Taming Data Caches for Predictable Execution on GPU-based SoCs*. IEEE, 2019. doi:10.23919/DATE.2019.8715255.

15 NVIDIA. Unified memory in cuda for beginners, 2017. URL: https://developer.nvidia.com/blog/unified-memory-cuda-beginners/.

16 NVIDIA. Gpudirect rdma, 2020. URL: https://docs.nvidia.com/cuda/gpudirect-rdma/index.html.

17 NVIDIA. Gpudirect storage, 2020. URL: https://developer.nvidia.com/gpudirect-storage.

18 System Optimization and Riverside Computer Architecture Laboratory at the University of California. Complete rodinia benchmark suite v3.1, 2017. URL: https://github.com/socal-ucr/Rodinia/tree/3.1.

19 Zaid Qureshi, Vikram Sharma Mailthody, Isaac Gelado, Seungwon Min, Amna Masood, Jeongmin Brian Park, Jinjun Xiong, Chris J. Newburn, Dmitri Vainbrand, I-Hsin Chung, Michael Garland, William J. Dally, and Wen-mei W. Hwu. Gpu-initiated on-demand high-throughput storage access in the bam system architecture. In Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift, editors, *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, pages 325–339. ACM, 2023. doi:10.1145/3575693.3575748.

20 Marcos Rodriguez. marcosrc92/MARS-data. Dataset, swhId: swh:1:dir:0b2fbba6fbdfa60cb3d84175a2dd102bfb293ff2 (visited on 2025-01-13). URL: https://github.com/marcosrc92/MARS-data, doi:10.4230/artifacts.22756.

21 Paul L. Springer. Berkeley's dwarfs on cuda, 2012. URL: https://api.semanticscholar.org/CorpusID:44643311.

22 Markus Steinberger. Scatteralloc: Massively parallel dynamic memory allocation for the gpu, 2012.

23 Sven Widmer, Dominik Wodniok, Nicolas Weber, and Michael Goesele. Fast dynamic memory allocator for massively parallel architectures. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units, GPGPU-6, Houston, Texas, USA, March 16, 2013*, GPGPU-6, pages 120–126, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2458523.2458535.