IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received 22 April 2024, accepted 22 May 2024, date of publication 29 May 2024, date of current version 6 June 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3406893

RESEARCH ARTICLE

XeroZerox: Analysis and Optimization of GPU Memory Management for High-Integrity Autonomous Systems

ALEJANDRO J. CALDERÓN[1], (Member, IEEE), LEONIDAS KOSMIDIS[2], (Member, IEEE), CARLOS-F. NICOLÁS[1], AND FRANCISCO J. CAZORLA[2]

[1]Ikerlan Technology Research Centre, Arrasate, 20500 Basque Country, Spain
[2]Barcelona Supercomputing Center (BSC), Barcelona, 08034 Catalonia, Spain

Corresponding author: Alejandro J. Calderón (ajcalderon@ikerlan.es)

This work was supported in part by European Union's Horizon Europe Program through the Modular Model-Based Design and Testing for Applications in Satellites (METASAT) Project under Grant 101082622, in part by Spanish Ministry of Economy and Competitiveness (Spanish State Research Agency/Agencia Española de Investigación (AEI)/http://dx.doi.org/10.13039/501100011033) under Grant PID2019-107255GB-C21 and Grant IJC-2020-045931-I, and in part by the Department of Research and Universities of the Government of Catalonia with a Grant to the Computer Architecture and Operating Systems (CAOS) Research Group under Code 2021 SGR 00637.

ABSTRACT Autonomous systems require high-performance processing capabilities, which demand the use of powerful accelerators such as GPUs. However, the use of GPUs in critical systems presents several challenges, since GPU programming models rely on explicit dynamic memory management. Traditionally, dynamic memory allocation in such systems is restricted to certain controlled scenarios, which require programs to be rewritten so that all the required memory is allocated at the beginning of the program and released at its end. However, many GPU applications do not follow this approach. Moreover, when dynamic memory allocation is used, it is critical to compute the exact amount of memory that will be used as well as to minimize it, to guarantee that it fits in the physical system memory. In this paper we present XeroZerox, an open-source tool that automatically converts the traditional dynamic memory allocations of GPU applications into allocations served from a centralized and optimally sized memory pool, which is managed in a way better suited for critical systems. XeroZerox allows legacy GPU applications to be used in a critical setup without rewriting them, minimizing at the same time their memory consumption and memory management runtime overhead.

INDEX TERMS Graphics processing units, dynamic memory management, autonomous systems.

I. INTRODUCTION

As the capabilities of autonomous systems continue to grow, there is an increasing demand for sophisticated computational processes to drive their functionalities. Systems such as unmanned aerial systems (UAS) and autonomous driving (AD) vehicles require high-performance processing capabilities to handle complex software-controlled functionalities and make real-time decisions [1], [2]. However, these systems are mostly deployed on resource-constrained embedded platforms that are restricted by Size, Weight, and Power-consumption (SWaP). In this context, the robust processing capabilities of Graphics Processing Units (GPUs) make them an attractive choice for autonomous systems [1]. GPUs, originally designed to render complex graphics, have evolved into general-purpose parallel processors capable of handling a variety of computational problems. As a result, they are frequently utilized in various domains, including systems that require real-time processing [3]. Specifically, embedded architectures that integrate CPU cores and a GPU in the same System on Chip (SoC), are gaining preference for the implementation of embedded autonomous systems due to their SWaP advantages. In this direction, the GPU market leader vendor NVIDIA has made significant investments

The associate editor coordinating the review of this manuscript and approving it for publication was Cristian Zambelli.

© 2024 The Authors. This work is licensed under a Creative Commons Attribution 4.0 License.
For more information, see https://creativecommons.org/licenses/by/4.0/

in the autonomous systems sector by designing the Jetson family of embedded GPU platforms, which offer a balance between SWaP and parallel processing capabilities [4], [5].

However, despite the SWaP advantages offered by embedded CPU-GPU platforms, the integration of GPUs into critical systems presents several challenges. One of the most significant challenges arises from the inherent GPU programming models that rely heavily on explicit dynamic memory management [6]. Dynamic memory allocation, while offering flexibility and adaptability, presents inherent uncertainties in real-time safety-critical applications. In such systems, unanticipated memory allocations can lead to unpredictable behaviors which can compromise the integrity and reliability of the system. Consequently, most critical systems traditionally adhere to strict memory management paradigms, under which dynamic memory allocation is not used, or where dynamic allocations are only allowed during the system initialization, maintaining the corresponding memory allocated during the whole program execution, and releasing it at the program termination. In fact, Khronos Safety Critical (SC) APIs, such as OpenGL SC 2.0 and Vulkan SC, do not offer calls to free dynamically allocated objects [7]. Furthermore, when this restricted type of dynamic memory allocation is allowed in critical systems, it is necessary to accurately compute the exact amount of memory that the program will use, to ensure it fits in the available physical system memory.

This conservative approach offers a degree of certainty, ensuring that a program will not suddenly run out of memory during execution, nor its performance will be affected by frequent memory allocations and deallocations. As has been demonstrated in [8], the allocation of dynamic GPU memory involves the creation of memory pools, which negatively impact the launch time of GPU kernels. However, this approach poses a significant challenge for many existing GPU applications. The vast majority of these applications are designed without the constraints of critical systems in mind. Consequently, they are not compatible with the memory management model that such systems demand. Re-engineering these applications to fit within these constraints would be a time-consuming and resource-intensive task.

In industry, tools like Valgrind are typically used to analyze and debug memory management issues in software applications [9]. However, it is important to note that Valgrind does not work well with GPU applications, as it was primarily designed for CPU-based memory analysis. This limitation presents a challenge when it comes to ensuring the memory integrity and reliability of GPU-accelerated critical systems [10]. Additional research and development are necessary to address the unique memory management challenges posed by GPU integration in these systems. There are alternative approaches to manage GPU memory in safety-critical applications, such as the use of static memory allocation strategies and memory over-provisioning techniques. These approaches aim to provide a balance between predictable memory usage and the parallel processing capabilities of GPUs in critical systems, offering potential solutions for the integration of GPUs in high-integrity applications.

In this work we present XeroZerox, an open-source tool designed to address this scenario. XeroZerox transforms the dynamic allocations of the traditional GPU memory model into allocations served from a centralized memory pool. For this, XeroZerox analyzes the memory use of a target GPU application and determines the optimal size of the centralized memory pool, which is created at the beginning of the application and released at its end. XeroZerox acts as a sub-allocator, merging redundant allocations of the traditional GPU memory model into a single zero-copy or unified memory allocation. This approach allows legacy GPU applications to fit within the memory management constraints of critical setups without the need for code modifications. Additionally, XeroZerox ensures that the memory consumption of these applications is minimized, reducing both the memory footprint and the associated runtime overhead of memory management. The research presented in this work constitutes a refinement of the ideas and findings originally presented in the doctoral thesis of the main author [11].

The remainder of this article is organized as follows. Section II provides background information on the use of GPUs in high-integrity systems and on the field of dynamic memory management for embedded GPUs. Section III describes the design and implementation of XeroZerox, detailing its key features and functionalities. Section IV describes the experimental results of memory optimization of various legacy GPU applications using XeroZerox, and also presents some performance considerations when using different memory models on embedded GPU platforms. Section V provides an overview of related work in the field of GPU memory management and optimization. Finally, Section VI presents the conclusions of this work.

II. BACKGROUND

A. GPUS IN HIGH-INTEGRITY AUTONOMOUS SYSTEMS

The use of GPUs in high-integrity systems, particularly in automotive and industrial control systems, is gaining traction due to their high computational power and efficiency. Trompouki and Kosmidis [12] have demonstrated a methodology to develop and certify the entire GPU software stack, including its toolchain, up to ASIL-D for automotive applications, highlighting the importance of functional safety certification at all levels of the system. This is critical as autonomous driving and Advanced Driving Assistance Systems (ADAS) require a high degree of resilience against faults to meet the requirements of the ISO-26262 functional safety standard [1]. However, the integration of GPUs into high-integrity systems presents significant challenges, particularly with regard to reliability, fault tolerance, and

security. To address these challenges, extensive research and development efforts have been directed towards enhancing GPU reliability and security. An approach involves conducting cross-layer GPU reliability evaluations, combining high-energy neutron beam experiments, fault simulation campaigns, and application profiling to unveil and mitigate GPU vulnerabilities [13]. Security has also been a focal point, with designs aiming to reduce the attack surface through low-overhead secure GPU memory, leveraging value locality to minimize authentication metadata and optimize memory bandwidth usage [14] [15] [16]. Furthermore, the concept of utilizing GPUs in System-on-Chip (SOC) architectures for industrial applications has been explored, proposing additional hardware and software diversity to improve safety integrity [17]. Overall, the use of GPUs in high-integrity systems is characterized by a multifaceted approach to enhancing reliability, fault tolerance, and security, ensuring these systems can meet the stringent requirements of their respective applications. In this work, we focus on reducing the uncertainty of GPU memory management to improve the integrity and reliability of GPU-based autonomous systems.

B. CUDA MEMORY MANAGEMENT

The traditional CUDA programming model requires the programmer to manage memory explicitly for both the CPU and GPU, which includes tasks such as allocation, deallocation, and transfers between host memory and device memory. CPU memory, allocated with `malloc`, is by default *paged*, meaning it can be swapped out to the disk by the operating system due to memory oversubscription. Conversely, GPU memory allocated with `cudaMalloc` is always *non-paged*, ensuring that it is always present in the memory.

Transferring data between CPU and GPU memory involves DMA (Direct Memory Access) operations, which are asynchronous with respect to CPU execution. However, DMA can only work when the pages are present in memory, and this is not always guaranteed for paged CPU memory. Therefore, the transfers need to pass through an intermediate buffer of non-paged memory, requiring synchronous copying from CPU before DMA can perform asynchronous transfers to the device. This results in additional memory overhead and timing delays in GPU transfers.

To avoid these overheads, programmers can allocate non-paged CPU memory, known as *pinned* or *paged-locked* memory, using `cudaMallocHost`. However, this type of memory is limited and more expensive to allocate due to user space to kernel space switches. This type of memory allows fully asynchronous transfers using `cudaMemcpyAsync`.

An alternative option is to allocate another type of *pinned* memory on the CPU side, which is memory-mapped to the GPU, using `cudaHostAlloc` with the `cudaHostAllocMapped` flag. This approach eliminates the need for explicit copies between CPU and GPU memory, which gives the name *zero-copy*. The implementation varies depending on the GPU type. Discrete GPUs use DMA engines to transfer data over the PCIe link, while embedded GPUs, sharing the main memory with the CPU, directly access the same memory as the CPU. In both cases, the programmer must ensure consistency in the shared memory between CPU and GPU. This functionality is supported by a feature known as UVA (Unified Virtual Addressing), which allows both CPU and GPU to use the same virtual address [18].

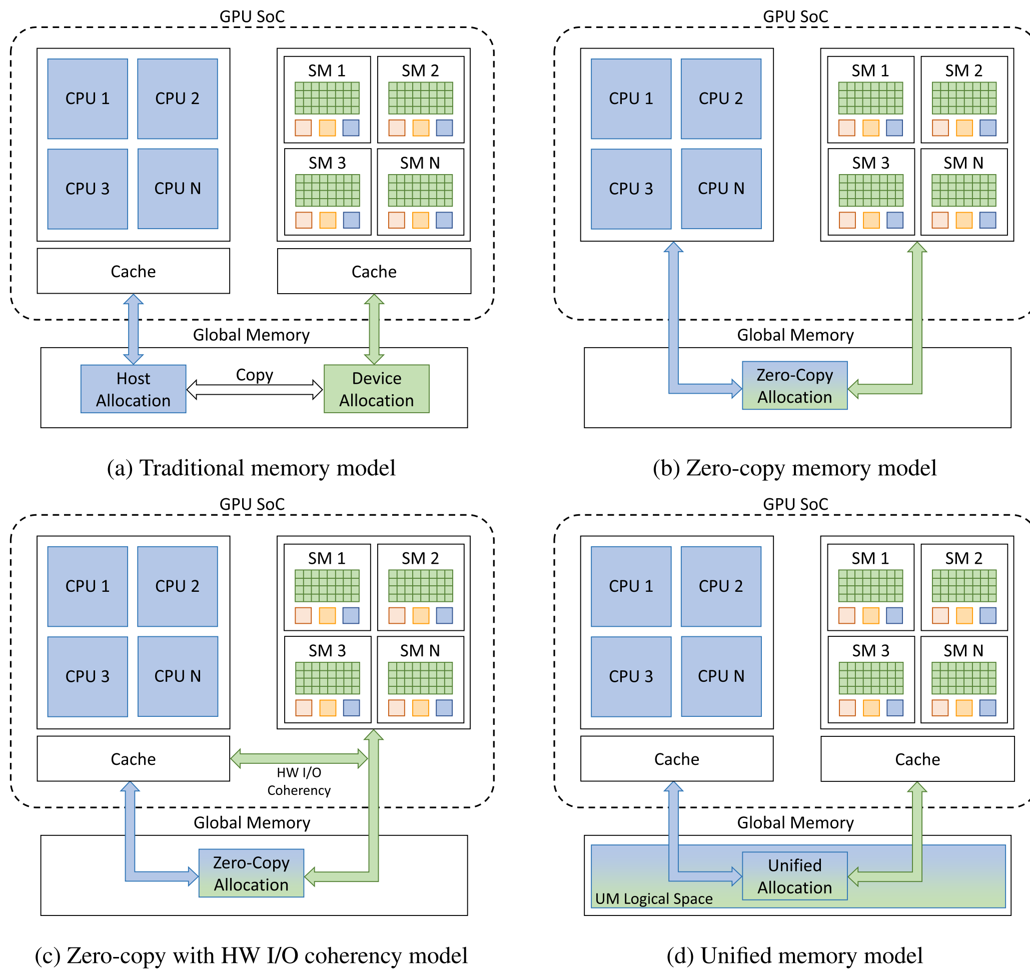
CUDA also provides Unified Memory, a feature that removes the responsibility of manually transferring data between CPU and GPU [19]. Unified Memory allows the creation of a single allocation for both CPU and GPU using `cudaMallocManaged`, eliminating the need for memory copies and simplifying code writing. Unified Memory uses the UVA feature, enabling a single memory pointer accessible from both the host side and the device side. Data transfers between CPU and GPU are handled internally by the CUDA runtime, which migrates memory pages on demand. Despite increasing productivity, the performance of Unified Memory is heavily dependent on the memory access patterns of each application and introduces more black-box behavior to memory management.

C. MEMORY MODELS IN EMBEDDED GPUS

In embedded GPU platforms, the CPU cores and the integrated GPU share the same physical memory, which can be managed using different memory models. It becomes essential to carefully select the appropriate memory model for optimal performance and efficient memory utilization when executing embedded GPU applications. The diagram shown in Figure 1 illustrates the operation of distinct CUDA memory models within embedded GPU platforms.

When using the traditional memory model (shown in Figure 1a), the physical memory is divided into two logical domains: one for host allocations and the other for device allocations. Before executing a kernel, data gets transferred from the host logical space to the device logical space. After the kernel execution, the output values are copied back from the device logical space to the host logical space. In this setup, both the CPU and GPU caches remain enabled, which can accelerate data transfers. Additionally, the synchronization of data access between the CPU cores and the integrated GPU is guaranteed by design. However, in most cases, the use of caches is not enough to completely hide the data transfer overhead. Furthermore, memory consumption remains suboptimal, as both CPU and GPU allocations are served from the same physical memory.

The zero-copy memory model provides a more effective strategy. Sharing physical memory between the CPU and GPU allows the CPU to share pointers to pinned host allocations directly accessible by the GPU, eliminating the necessity for DMA transfers. This further eliminates the need for memory copies and reduces memory consumption compared to the traditional memory model. Nevertheless, a shared memory space requires ensuring cache coherence



(a) Traditional memory model

(b) Zero-copy memory model

(c) Zero-copy with HW I/O coherency model

(d) Unified memory model

FIGURE 1. CUDA memory models in embedded GPUs.

between CPU and GPU. Since cache coherency through software-based mechanisms could introduce additional overhead, certain platforms, like NVIDIA embedded GPUs with compute capability below 7.2, tackle the cache coherency challenge by disabling last-level caches in both CPU and GPU (as illustrated in Figure 1b). This approach can negatively impact the performance of cache-dependent GPU applications. Consequently, NVIDIA suggests implementing zero-copy memory for small buffers, where the caching impact is minimal. In cases involving larger buffers, zero-copy memory proves beneficial when memory access patterns do not rely on caches.

To mitigate this constraint, the most recent NVIDIA embedded platforms featuring a compute capability of 7.2 or higher incorporate a hardware-based I/O coherency mechanism (as shown in Figure 1c) [20]. I/O coherency enables an I/O device, such as a GPU, to access the latest updates in CPU caches, eliminating the need for CPU cache management actions when CPU and GPU share the same physical memory. Nevertheless, GPU cache management tasks remain necessary due to the unidirectional nature of the I/O coherency mechanism. Consequently, the GPU cache remains inactive when using zero-copy memory on these platforms.

In embedded GPU platforms, unified memory allocations consist of pointers to a single unified logical domain accessible by both the CPU and the GPU (as shown in Figure 1d). However, the implementation of unified memory in embedded GPUs differs from that of discrete GPUs. Instead of dynamically migrating pages between host memory and device memory, the embedded unified memory model mirrors the zero-copy memory model, featuring single allocations for both CPU and GPU to reduce memory consumption. Nevertheless, unlike the zero-copy memory model, the unified memory model maintains active CPU and GPU caches, requiring the runtime system to perform software-based cache coherence management tasks. The overhead caused by these operations is higher in embedded platforms with a compute capability less than 7.2, as they lack hardware I/O coherency [20]. It is important to note that, according to NVIDIA, software-managed coherency is by nature non-deterministic and not recommended in a safe

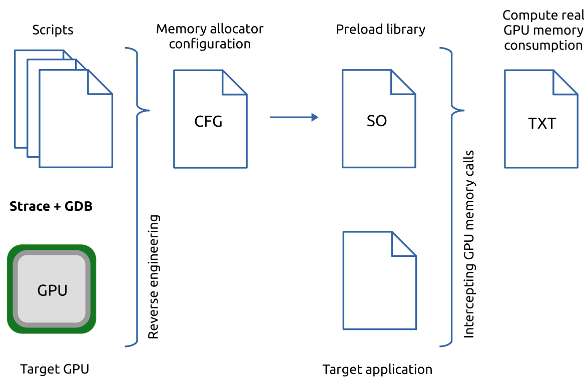


FIGURE 2. GMAI workflow [21].

```
Device name: NVIDIA Tegra X2
Compute capability: 6.2
CUDA runtime version: 9.0
CUDA driver version: 9.0

Pool size: 2097152 bytes
Granularity: 512 bytes

Size classes
1: 1   to 2    blocks of 512 [1      to 1024 bytes   ]
2: 3   to 8    blocks of 512 [1025   to 4096 bytes   ]
3: 9   to 32   blocks of 512 [4097   to 16384 bytes  ]
4: 33  to 128  blocks of 512 [16385  to 65536 bytes  ]
5: 129 to 512  blocks of 512 [65537  to 262144 bytes ]
6: 513 to 3583 blocks of 512 [262145 to 1834496 bytes]
Larger allocations: mmap rounded to next 4KB multiple

Allocator policy: Best fit
Coalescing support: Yes
Splitting support: Yes
Expansion policy: When full. Use last created.
Shrinking support: Yes. Any pool deleted.
```

LISTING 1. Example GMAI report for NVIDIA TX2 [21].

context. Therefore, in these applications, it is preferable to use the zero-copy memory model [20].

D. GMAI: GPU MEMORY ALLOCATOR INSPECTOR

GMAI (GPU Memory Allocator Inspector) is a tool that can be run on any Linux system equipped with a GPU to extract its memory allocator properties [21]. GMAI consists of two parts: the first part is a set of reverse engineering scripts, based on the microbenchmarks defined in [8], to extract the properties of a GPU memory allocator. The second part is a preload library designed to obtain the real memory consumption of GPU applications, using the properties extracted in the previous part. Figure 2 shows the workflow followed by GMAI to extract the properties of a GPU memory allocator and to use this information to compute the real memory use of GPU applications. The source code is available at [22].

When GMAI is executed, it extracts key properties of the underlying GPU memory allocator. These properties include the size of the memory pools created by the allocator, the internal granularity of these memory pools, the size ranges for which the pools are created (also known as size classes), and the memory allocation policy. This information is presented in a report similar to the one illustrated in Listing 1. Additionally, the information is saved in a configuration file that can be utilized later by the preload library to automatically determine the actual GPU memory usage of a target application. In this work, we use the information extracted with GMAI to calculate the real amount of memory used by GPU applications. Based on this information, XeroZerox can create a single, optimally sized memory pool for each target application.

III. DESIGN AND IMPLEMENTATION

In this section, we present the design and implementation of XeroZerox. The main objective of XeroZerox is to adapt the memory management of legacy GPU applications to a memory model more suitable for critical systems, minimizing at the same time their memory consumption. Legacy GPU applications normally use the traditional CUDA memory model, in which the programmer must define and initialize a set of allocations on the host side using dynamic pageable memory, and also define an equivalent set of allocations on the GPU side using dynamic device memory. Furthermore, before executing a kernel, the programmer must explicitly copy the initial values from host memory to device memory, and after the kernel execution, copy the results back from device memory to host memory. As explained in Section II-C, in embedded GPU platforms, both host and device allocations are served from the same physical memory. Therefore, when an application that uses the traditional GPU memory model is executed on an embedded GPU platform, its memory consumption is not optimal and it performs unnecessary memory transfers.

XeroZerox has been designed to automatically detect the memory allocations of the traditional memory model and replace them with memory allocations served from a centralized memory pool. This memory pool is created using either zero-copy or unified memory, and its size is calculated based on the amount of memory used by the target application. This way, XeroZerox reduces memory consumption and eliminates the need for memory transfers.

XeroZerox interacts with the CUDA runtime system at the beginning of the application to create the centralized memory pool and then at the end of the application to release it. All memory allocations requested by the application are served internally by XeroZerox using the centralized memory pool. This is how XeroZerox also minimizes the overhead produced when interacting with the CUDA runtime memory management system. Furthermore, since all memory allocations are served from the centralized memory pool, XeroZerox avoids memory leaks by releasing the entire memory pool when the application finishes.

The functionality of XeroZerox is divided into two different libraries: the analysis library and the optimization library. As shown in Figure 3, XeroZerox is applied in two different phases. In the first phase, the XeroZerox analysis library must be included in the compilation process of the target application. The analysis library contains wrapper

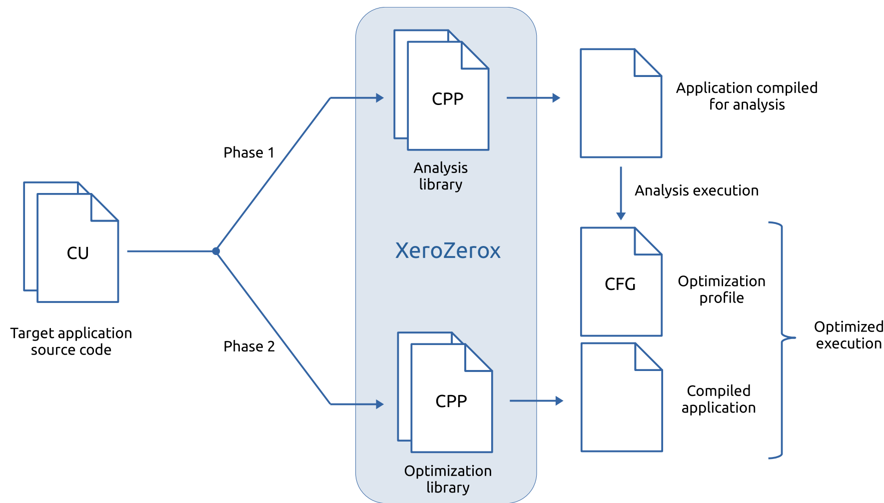


FIGURE 3. XeroZerox analysis and optimization workflow.

functions for all key functions required for dynamic memory management in CUDA applications. Each wrapper function gathers the information required for the analysis before calling the corresponding real function. When the resulting compiled application is executed, it generates a file that we call the *optimization profile*, which is used in the next phase. The details related to the analysis phase are discussed in Section III-A.

In the second phase, the target application must be compiled again, this time including the XeroZerox optimization library in the compilation process. This library contains alternative function replacements for all functions used for dynamic GPU memory management in CUDA applications. The alternative functions implement optimized versions of the original functions. In order to work properly, these functions require the information included in the optimization profile generated in the previous phase. The details related to the optimization phase are discussed in Section III-B.

A. XEROZEROX ANALYSIS PHASE

The functionality required for the XeroZerox analysis phase is implemented in the analysis library. The objective of the analysis library is to gather information about the memory-related function calls of the target GPU application, to characterize how the different types of dynamic memory are used, and to create a memory optimization profile. The results of the analysis phase are a memory characterization report, an interactive memory plot showing the evolution of the different types of dynamic memory used in the application and the location of the related function calls in the source code, and a memory characterization profile that stores information needed for the optimization phase. Figure 4 shows an example of a memory characterization plot. The interactive memory characterization plot allows the user to observe the dynamic memory behavior of a target GPU application and trace back interesting function calls to their location in the source code. To implement this functionality, it is necessary to capture all GPU-related dynamic memory calls and extract data related to the requested memory size, assigned addresses, and location of the function calls in the source code.

GPU applications can use different types of dynamic memory. For each type of memory, there are specific functions that the analysis library needs to capture. For device memory, it must keep track of `cudaMalloc` and `cudaFree` calls. For pinned memory, it must capture calls to `cudaMallocHost`, `cudaHostAlloc` and `cudaFreeHost`. On the host side, GPU applications can also use dynamic pageable memory, which is usually allocated with `malloc` and deallocated with `free`. However, pageable memory allocations can also be done with other functions such as `calloc`, so the analysis library must keep track of those function calls as well.

In order to capture the target memory functions, the analysis library uses a technique known in the literature as *function interposition* [23], which consists in replacing target functions with user-defined wrapper functions. These wrapper functions can then be used to add extra functionality to the original functions, in our case, to extract information from the arguments. In UNIX-like systems, function interposition can be done at run time using the environment variable `LD_PRELOAD` [24] which is used by GMAI [21], at load time using the `--wrap` argument of the `ld` linker [25], or at compile time, replacing the target function calls at the source code level.

Since GMAI uses `LD_PRELOAD` to implement function interposition at run time, our first approach was to extend the GMAI preload library to add the functionality required for the analysis phase. However, with this approach, the extraction of source code information is not optimal. To implement it, it must be assumed that the target application has been compiled with debugging information. Then, for each

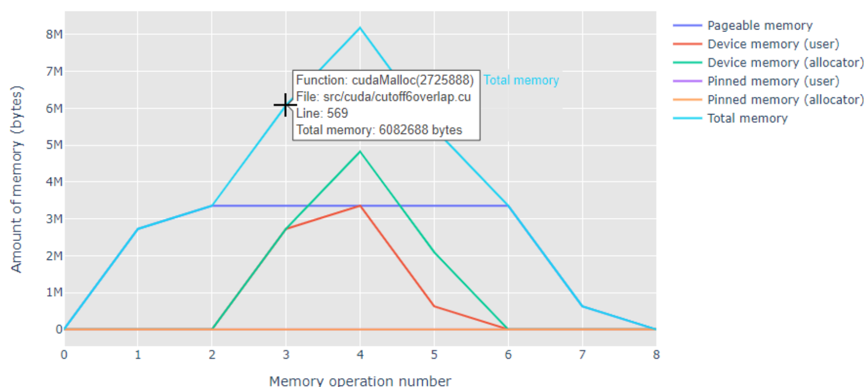


FIGURE 4. Example interactive memory characterization plot.

```
#define malloc(size) chr_malloc(size,__FILE__,__LINE__)
#define free(ptr) chr_free(ptr,__FILE__,__LINE__)
#define cudaMalloc(devPtr,size) chr_cudaMalloc(devPtr,size,__FILE__,__LINE__)
#define cudaFree(devPtr) chr_cudaFree(devPtr,__FILE__,__LINE__)
```

LISTING 2. Examples of function interposition at compile time.

function call, the debugging symbols must be extracted from the compiled binary. We created a prototype using this approach and found that the extraction of source information added much processing overhead on function calls. Moreover, while this approach works well with CUDA functions, for more generic functions such as `malloc` it generates recursion problems, since `malloc` is used internally by functions like `dlsym`, which is used inside the wrapper functions to invoke the real functions.

Due to the limitations of the `LD_PRELOAD` approach, we opted to use function interposition at compile time for the implementation of the analysis library. With this approach, the calls of the target function are replaced in the source code using `#define` directives. One of the advantages of this approach is that extraction of source code information is easier to implement. In C/C++, we have access to the `__FILE__` and `__LINE__` macros which expand to the full path of the current file and the current line number, respectively [26]. In the analysis library, these macros are added as extra arguments to the wrapper functions to get the source information for every function call. Listing 2 shows some examples.

As part of the analysis library, we created a header file with `#define` directives to replace the target memory management functions with wrapper functions. In order to replace the original function calls with the wrapper functions, this header file must be included in the source files of the target application by adding a simple `#include` directive. When the application is compiled, the memory characterization functionality is added to the function calls.

The analysis library uses as input the configuration file generated by GMAI. From this file, it gets information such as the pool size and the size classes used by the GPU memory allocators. Using these values, it keeps track of the creation of memory pools and it calculates the real amount of device memory and pinned memory reserved by the CUDA allocators.

When the compiled application is executed, each wrapper function gathers information about its corresponding memory function. At the end of the execution, the analysis library creates a report with information such as the maximum amount of dynamic memory used by the application, the number of memory pools created, the number of memory transfers, and if there are memory leaks. The library also creates an interactive memory characterization plot, such as the one shown in Figure 4.

The interactive memory characterization plot shows the evolution of pageable host memory, pinned host memory, and device memory through the execution of the application. For device and pinned memory, it shows the values requested by the user as *Device memory (user)* and *Pinned memory (user)* respectively. It also shows the real amounts of memory reserved by the CUDA memory allocators as *Device memory (allocator)* and *Pinned memory (allocator)*, based on the information extracted with GMAI. At each point, the *Total memory* is calculated as the sum of pageable host memory plus the amount of device memory and pinned memory reserved by the CUDA allocators. For each memory operation, the plot shows the current state of all types of memory and the information of each function call. As an illustrative example, Figure 4 shows that the third memory operation of the analyzed application was a `cudaMalloc` call to reserve 2,725,888 bytes of device memory, the function call is at line 569 of the cutoff6overlap.cu source file, and the total amount of GPU-related dynamic memory used at this point of the program execution is 6,082,688 bytes.

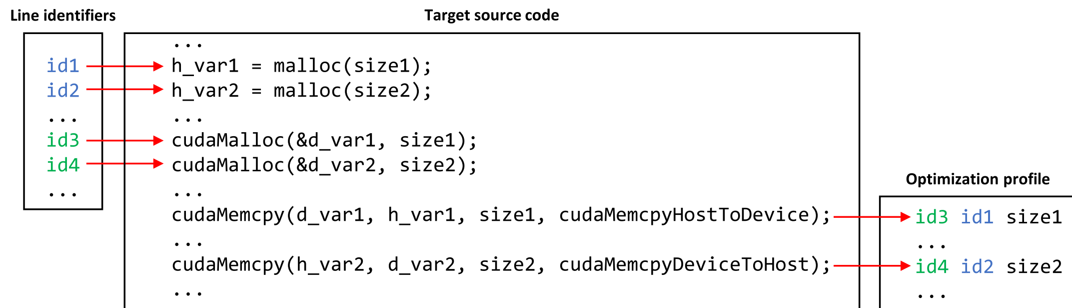


FIGURE 5. Generation of the optimization profile.

In addition to the interactive memory characterization plot, the analysis library creates the optimization profile, as shown in Figure 5. The optimization profile is a file that contains information on the correspondence between host and device allocations, the sizes requested for each allocation, and the maximum amount of memory needed by the application. This information is required to carry out the optimization phase. To create the optimization profile, first, the host memory allocation wrappers (`malloc/calloc/cudaMallocHost`) and the `cudaMalloc` wrapper gather information about the allocation sizes requested by the application and assign an identifier to each allocation. Then, it is necessary to know which host allocation is related to which device allocation. For this purpose, we take advantage of the `cudaMemcpy` function. The `cudaMemcpy` wrapper identifies the correspondence between a host allocation and a device allocation and creates the corresponding match in the optimization profile.

To calculate the maximum amount of memory needed by the application, the analysis library keeps track of the amounts of memory requested on each individual allocation using the `malloc`, `calloc`, `cudaMallocHost` and `cudaMalloc` wrappers. It also keeps track on how these allocations are released, using wrappers for the `free`, `cudaFreeHost` and `cudaFree` functions. With that information, the analysis library continuously updates the current amount of memory used and keeps track of the maximum value reached. At the end of the execution, the maximum registered value is stored in the optimization profile as the maximum amount of memory needed by the application. It is worth noting that the analysis phase is only needed to be executed once per application.

B. XEROZEROX OPTIMIZATION PHASE

The functionality required to apply the XeroZerox optimizations is implemented in the optimization library. For this library, we also created a header file with `#define` directives to replace the original memory management functions with alternative functions served by XeroZerox. To carry out the optimization, the header file must be included in the source code of the target application. This way, the memory management functions are replaced at the source code level. Then, the target application must be linked together with the optimization library. In the compilation process, the optimization library must be configured to use either zero-copy memory or unified memory to create the centralized memory pool. When the resulting compiled application is executed, it uses the XeroZerox functions instead of the traditional memory model functions to serve the allocations using the centralized memory pool.

As shown in Figure 6, when using the traditional memory model in an embedded GPU platform, the memory is partitioned into one logical space for the host allocations and one logical space for the device allocations. The host allocations are served by a host allocator in the system, while the device allocations are served by an NVIDIA allocator. As previously stated, the memory consumption in this scenario is not optimal since both CPU and GPU allocations are served from the same physical memory. Also, it requires the application to perform memory transfers between both logical spaces, which can negatively impact performance. Furthermore, the NVIDIA allocator can create multiple memory pools according to predefined size classes, which can negatively impact both memory consumption and performance [8].

XeroZerox proposes an alternative memory model, which we show in Figure 7. When an application optimized with XeroZerox is executed, it first reads the optimization profile and loads the information required for optimization. The first value it reads is the maximum amount of memory needed, which is used as a reference to create the centralized memory pool. For this task, XeroZerox requests the NVIDIA allocator to reserve either zero-copy memory or unified memory, according to the memory type configured in the compilation process. It is worth noting that the minimum amount of memory reserved for the centralized memory pool still depends on the NVIDIA allocator and the configured memory type. For zero copy memory, the minimum amount of memory reserved by the NVIDIA allocator is the pool size, which can be 1 MB or 2 MB depending on the platform [8]. In the case of unified memory, the minimum amount of memory reserved by the NVIDIA allocator in the Jetson platforms is 4 KB, which is the DRAM page size. For sizes larger than the pool size in zero-copy, or the page size in

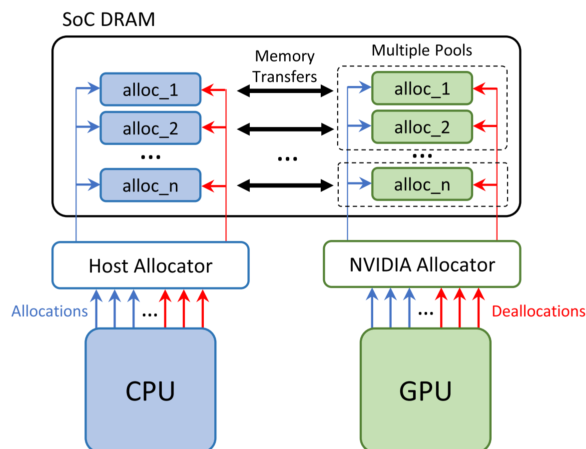


FIGURE 6. Traditional memory model.

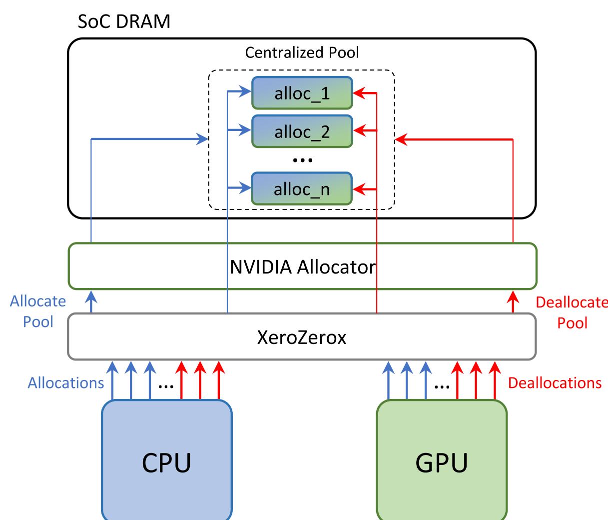


FIGURE 7. XeroZerox memory model.

unified memory, the amount of memory reserved by the NVIDIA allocator is the next multiple of the page size.

After creating the centralized memory pool, XeroZerox works as a sub-allocator, attending the allocation requests from the application based on the matches loaded from the optimization profile. Since each match represents a host allocation and a device allocation, XeroZerox only allocates memory for the first of these two allocation requests that it receives. When it receives the second allocation request, it returns a pointer to the region of memory that has been already allocated for the first one. In this way, XeroZerox transforms two matching host and device allocations into a single allocation served from the centralized memory pool. Similarly, XeroZerox deallocates a memory region only when it receives the two corresponding deallocation requests. At the end of the application execution, XeroZerox deallocates the centralized memory pool using the corresponding CUDA function.

It is worth noting that, while the zero-copy and unified memory models allow the reduction in memory consumption, the performance of the applications when using these

TABLE 1. Platforms used for XeroZerox evaluation.

Jetson platform	GPU architecture	Compute capability	SMs	GPU cores	HW I/O coherency
TX2	Pascal	6.2	2	256	No
Xavier NX	Volta	7.2	6	384	Yes

memory models will depend on the memory access patterns of each application and the coherency mechanisms of the underlying platform, as explained in Section II-C. Even when XeroZerox can improve the performance of an application by eliminating the need for memory transfers and reducing the interaction with the runtime system, this performance improvement can be overshadowed by the limitations of the zero-copy and unified memory models. The source code of XeroZerox is available at [27].

IV. EVALUATION

This section presents the results we obtained after applying the XeroZerox analysis and optimization to the benchmarks of the Rodinia benchmark suite [28], one of the most widely used GPU benchmarking suites. The Rodinia benchmark suite contains 23 benchmarks with CUDA, OpenMP and OpenCL implementations. Due to compilation issues, we do not include the *mummergpu* benchmark in the evaluation. The *cfd*, *particlefilter* and *srad* benchmarks have multiple versions, but we consider that selecting one version of each of these benchmarks is enough for the purposes of this work. For *cfd* we selected the *euler3d* version, for *particlefilter* we selected the *particlefilter_float* version, and for *srad* we selected the *srad_v1* version.

In order to carry out the evaluation, we first compiled the CUDA version of the benchmarks, including the XeroZerox analysis library in the compilation process. Then, we executed the compiled benchmarks to obtain the information for memory characterization and the optimization profiles. Finally, we compiled the benchmarks again using the XeroZerox optimization library, and generated one version optimized with zero-copy memory and one version optimized with unified memory for all the benchmarks. We used two different platforms from the NVIDIA Jetson family to perform the evaluation. The details of the selected platforms are provided in Table 1. We selected one platform with hardware I/O coherency and another one without hardware I/O coherency, with the purpose of illustrating the different behaviors we can expect in the performance of the applications when using zero-copy and unified memory.

A. MEMORY ANALYSIS AND CHARACTERIZATION

For the analysis phase, we compiled the benchmarks including the XeroZerox analysis library in the compilation process, and we executed the resulting binaries to obtain the information for memory characterization and the optimization profiles. Table 2 shows a summary of the information obtained in the memory characterization reports.

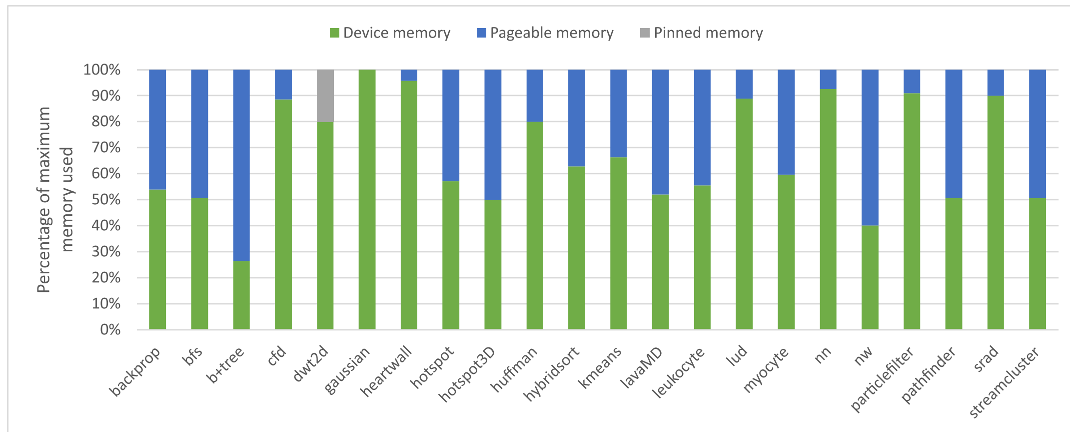


FIGURE 8. Rodinia benchmarks distribution of maximum total memory used.

TABLE 2. Rodinia benchmarks characterization results.

Benchmark	Max. memory used (bytes)	Pools created	H2D transfers	D2H transfers
backprop	20455632	3	5	3
bfs	79202120	5	18	13
b+tree	84762324	5	15	3
cfd	17094144	6	3	1
dwt2d	36700160	12	1	3
gaussian	2097296	1	3	3
heartwall	46035288	21	24	4
hotspot	7340032	2	2	1
hotspot3D	50331648	3	2	1
huffman	10487808	5	4	2
hybridsort	90218496	20	12	12
kmeans	205642200	3	5	2
lavaMD	16359296	3	4	1
leukocyte	3779072	6	32	7
lud	2359296	1	1	1
myocyte	3517200	1	7800	7800
nn	2268208	1	1	1
nw	83976204	2	2	1
particlefilter	2305544	1	6	3
pathfinder	82097280	2	2	1
srad	9315952	4	5	201
streamcluster	147656756	524	3223	3222

Regarding memory transfers, Table 2 shows the number of host-to-device and device-to-host memory transfers (H2D and D2H respectively) for each benchmark. The benchmarks with the highest number of memory transfers are *myocyte*, *srad* and *streamcluster*. When executing applications like these benchmarks on embedded GPU platforms, replacing the pageable allocations and their corresponding device allocations with zero-copy or unified allocations could help to improve performance, since all related memory transfers could be avoided.

In terms of memory consumption, Table 2 shows the maximum dynamic memory used for each benchmark during execution. The *kmeans* and *streamcluster* benchmarks are the ones with more memory consumption. In the case of *streamcluster*, the allocation activity is more intensive, creating device memory pools up to 524 times. The *maximum memory used* value is composed by different types of dynamic memory. To better understand its composition, Figure 8 shows what percentage of the maximum value corresponds to each type of dynamic memory in all the benchmarks.

As we can see, in almost all the benchmarks, the maximum memory used value is composed only by pageable host memory and device memory. The only benchmark using pinned host memory is *dwt2d*. There are some benchmarks like *backprop*, *bfs*, *hotspot*, *hotspot3D*, *lavaMD*, *leukocyte*, *nw*, *pathfinder*, and *streamcluster* where the memory is almost evenly distributed between host memory and device memory. When executing applications like these benchmarks on embedded GPU platforms, replacing the pageable allocations and their corresponding device allocations with zero-copy or unified allocations could help to reduce the memory consumption approximately to the half.

To analyze the behavior of the Rodinia benchmarks in terms of how memory is allocated and deallocated, Figure 9 shows the memory characterization plots obtained for all the benchmarks during the XeroZerox analysis phase. In typical GPU applications, memory for different variables is allocated gradually, presenting an increasing behavior until reaching the peak of maximum memory used by the application. Then, these allocations are gradually released after use, presenting a decreasing behavior until the total memory used is zero. In this category we have benchmarks like *backprop*, *bfs*, *gaussian*, *hotspot*, *hotspot3D*, *kmeans*, *lavaMD*, *lud*, *nn*, *nw*, *particlefilter*, *pathfinder* and *srad*. This is the desired behavior for most GPU applications. We just need to make sure that all allocations are grouped at the beginning of the application and that all deallocations are done in the end. This way, we can avoid the creation of memory pools in the middle of the execution, which can have a negative timing impact when launching GPU kernels [8].

On the other hand, there are applications like *hybridsort*, *leukocyte*, and *streamcluster* which have a repetitive pattern in allocation and deallocation along their execution. This

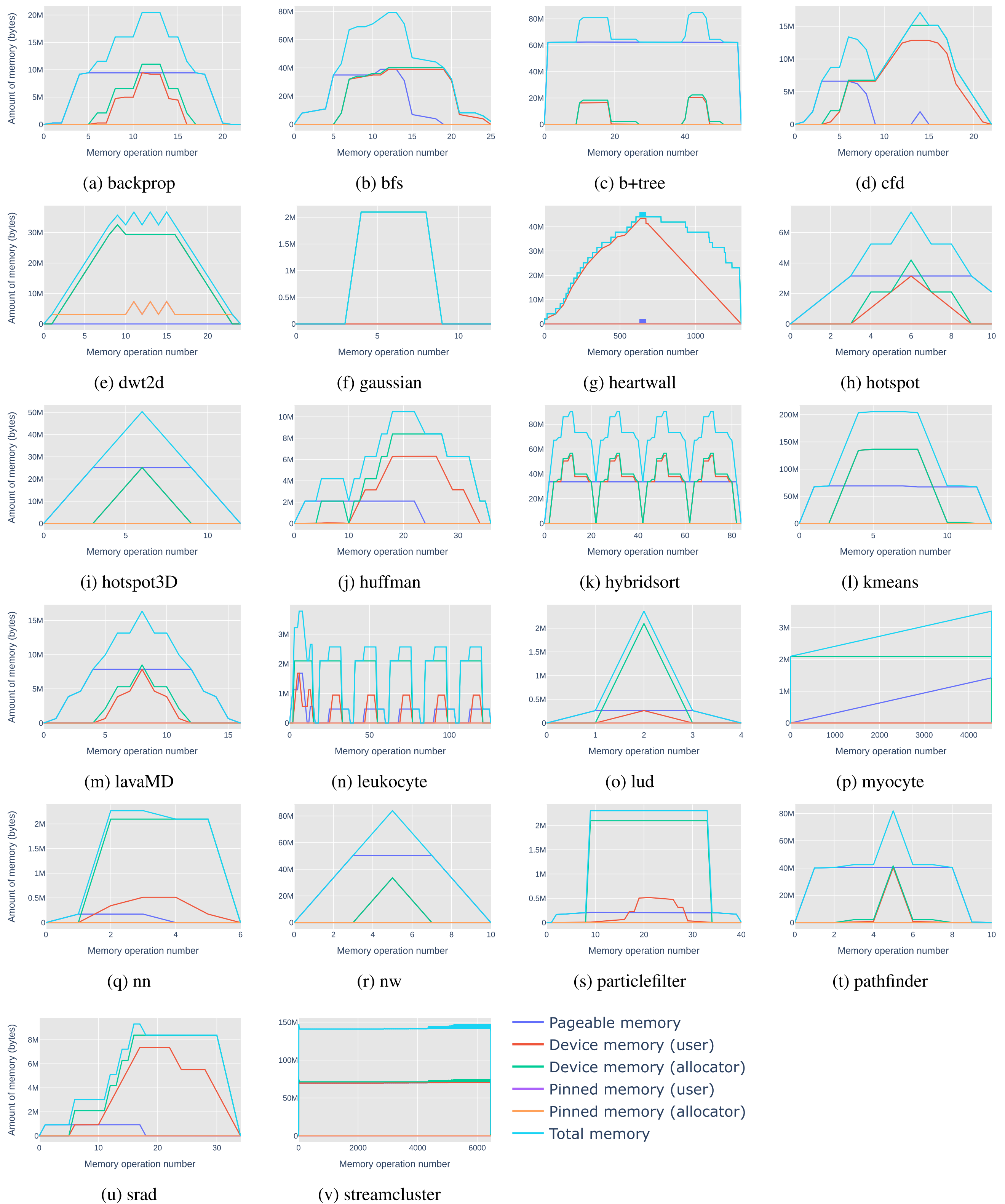


FIGURE 9. Memory characterization plots of Rodinia benchmarks.

behavior usually corresponds to iterative blocks of code with allocations and deallocations inside a loop. It is common in applications that execute the same GPU kernel several times, such as applications that process images or frames

TABLE 3. Memory consumption results for Rodinia benchmarks.

Benchmark	Memory consumption (bytes)				Reduction	
	Original		XeroZerox			
	Required	Allocated	ZC	UM	ZC	UM
backprop	9437460	20455632	9441280	9441280	53.85 %	53.85 %
bfs	38999881	79202120	39002112	39002112	50.76 %	50.76 %
b+tree	62377684	84762324	62377984	62377984	26.41 %	26.41 %
cfd	12824064	17094144	12824576	12824576	24.98 %	24.98 %
dwt2d	32505856	36700160	32505856	32505856	11.43 %	11.43 %
gaussian	144	2097296	2097152	4096	0.01 %	99.80 %
heartwall	43350996	46035288	43352064	43352064	5.83 %	5.83 %
hotspot	3145728	7340032	3145728	3145728	57.14 %	57.14 %
hotspot3D	25165824	50331648	25165824	25165824	50.00 %	50.00 %
huffman	6301704	10487808	6303744	6303744	39.89 %	39.89 %
lavaMD	7856000	16359296	7856128	7856128	51.98 %	51.98 %
leukocyte	1681920	3779072	2097152	1683456	44.51 %	55.45 %
lud	262144	2359296	2097152	262144	11.11 %	88.89 %
myocyte	812	3517200	2097152	4096	40.37 %	99.88 %
nn	513168	2268208	2097152	516096	7.54 %	77.25 %
nw	50380812	83976204	50384896	50384896	40.00 %	40.00 %
particlefilter	516392	2305544	2097152	520192	9.04 %	77.44 %
pathfinder	40400000	82097280	40402944	40402944	50.79 %	50.79 %
srad	7364992	9315952	7368704	7368704	20.90 %	20.90 %
streamcluster	72941620	147656756	72945664	72945664	50.60 %	50.60 %

in a video. The problem with applications like these is not the iterative behavior, but to include the allocations and deallocations inside the processing loop. For each allocation and deallocation, the application needs to interact with the CUDA runtime, which adds processing overhead on each iteration. Moreover, some allocations can trigger the creation of new memory pools, which can add timing overhead to the execution time of the next GPU kernel launch [8].

B. MEMORY CONSUMPTION OPTIMIZATION

In order to evaluate XeroZerox in terms of reduction in memory consumption, we compiled the benchmarks using the XeroZerox optimization library and the optimization profiles we obtained in the previous phase. We generated zero-copy and unified memory versions of the benchmarks and executed them on the selected embedded GPU platforms. Since the memory page size is the same in all Jetson platforms and both selected platforms share the same pool size, the memory consumption results are identical for both platforms. When performing the evaluation, we realized that the *hybridsort* and the *kmeans* benchmarks were not compatible with XeroZerox. By design, XeroZerox works with legacy GPU applications in which the correspondence between host allocations and device allocations is one-to-one. If an application has one-to-many correspondences between host allocations and device allocations, applying XeroZerox can result in unexpected behavior. For this reason, these benchmarks are not included in the evaluation. Table 3 shows the results we obtained for the rest of the benchmarks.

In the table, the *Allocated* column shows the amount of memory allocated in the original Rodinia benchmarks using the traditional memory model. The *Required* column shows the maximum amount of memory required by each benchmark, which has been calculated in the XeroZerox analysis phase. The *XeroZerox* columns present the size of the centralized memory pool in both zero-copy (ZC) and unified memory (UM) versions of the benchmarks. Finally, the *Reduction* columns present the percentage of reduction in memory consumption for both ZC and UM versions with respect to the memory allocated in the original benchmarks.

As shown in the table, the ZC and UM versions present an identical percentage of reduction in the cases where the required amount of memory is larger than the pool size, which in the selected platforms is 2 MB. This is an expected result since, in the Jetson platforms, all unified memory allocations and the zero-copy allocations larger than the pool size are reserved using the same rule: reserve the next multiple of 4 KB. On the other hand, for those benchmarks that require small amounts of memory, like *gaussian*, *lud*, *myocyte*, *nn* and *particlefilter*, the percentage of reduction of the UM version is higher than the percentage of the ZC version. This is so because of the minimum pool size that can be allocated for each type of memory. For example, to serve the 144 bytes required for the *gaussian* benchmark, it is more efficient the minimum 4 KB that can be reserved with unified memory than the minimum 2 MB that can be reserved with zero-copy memory.

The highest reduction in memory consumption is obtained in benchmarks like *backprop*, *bfs*, *hotspot*, *hotspot3D*, *lavaMD*, *leukocyte*, *nw*, *pathfinder* and *streamcluster*, for which the percentage of reduction is around 50% with both ZC and UM versions. As shown in Figure 8, in these benchmarks, the memory used is almost evenly distributed between host memory and device memory, which makes possible the reduction in memory consumption to approximately the half. On the contrary, for benchmarks like *cfd*, *dwt2d*,

TABLE 4. Maximum execution times of Rodinia benchmarks.

Benchmark	Jetson TX2			Jetson Xavier NX		
	Original	XeroZerox		Original	XeroZerox	
		ZC	UM		ZC	UM
backprop	0.199 s	0.282 s	0.214 s	0.223 s	0.222 s	0.227 s
bfs	3.811 s	4.225 s	3.846 s	3.624 s	3.655 s	3.058 s
b+tree	3.144 s	3.534 s	3.328 s	2.824 s	2.913 s	2.799 s
cfd	12.604 s	41.396 s	13.618 s	6.878 s	7.590 s	6.662 s
dwt2d	0.275 s	0.287 s	0.163 s	0.177 s	0.194 s	0.230 s
gaussian	0.087 s	0.083 s	0.083 s	0.129 s	0.129 s	0.131 s
heartwall	2.037 s	64.415 s	2.064 s	0.763 s	0.817 s	0.846 s
hotspot	0.691 s	0.776 s	0.736 s	0.788 s	0.671 s	0.686 s
hotspot3D	17.880 s	21.860 s	21.405 s	9.714 s	9.920 s	9.433 s
huffman	0.225 s	0.459 s	0.235 s	0.222 s	0.210 s	0.211 s
lavaMD	1.173 s	1.236 s	1.177 s	0.979 s	0.989 s	0.985 s
leukocyte	0.759 s	0.813 s	0.760 s	0.676 s	0.614 s	0.670 s
lud	0.115 s	0.130 s	0.121 s	0.153 s	0.164 s	0.160 s
myocyte	1.130 s	0.436 s	0.548 s	0.971 s	0.517 s	0.535 s
nn	0.120 s	0.173 s	0.135 s	0.198 s	0.190 s	0.184 s
nw	0.244 s	0.268 s	0.220 s	0.196 s	0.293 s	0.290 s
particlefilter	0.130 s	0.288 s	0.138 s	0.202 s	0.224 s	0.226 s
pathfinder	2.897 s	2.904 s	2.921 s	2.566 s	2.302 s	2.675 s
srad	0.307 s	0.982 s	0.334 s	0.299 s	0.319 s	0.322 s
streamcluster	15.553 s	49.666 s	14.371 s	9.001 s	7.785 s	10.306 s

heartwall and *srad*, for which the memory is not evenly distributed, we obtain less than 25% of reduction in memory consumption.

C. PERFORMANCE CONSIDERATIONS

As explained in Section II-C, the performance of applications using the zero-copy and the unified memory models depends on the memory access patterns of the application and the coherency mechanisms used in the underlying platform. Therefore, to evaluate the performance of the Rodinia benchmarks after applying the XeroZerox optimizations, we executed several times the original, the zero-copy, and the unified memory versions of the benchmarks on both embedded GPU platforms and registered their maximum execution times. The results are shown in Table 4.

The performance of the *cfd*, *heartwall* and *streamcluster* benchmarks is affected when we use zero-copy memory in the Jetson TX2 platform. If we compare these results with the results obtained when we use unified memory on the same platform, we can see that the performance is not affected in the same way. This means that these applications are cache-dependent. In embedded NVIDIA platforms without hardware I/O coherency, like the Jetson TX2, the last level caches of both CPU and GPU are disabled when using zero-copy memory but remain enabled when using unified memory. In the case of the Jetson Xavier NX, there are no significant changes in the performance of the applications when using zero-copy or unified memory instead of the traditional memory model. Due to the presence of hardware I/O coherency in this platform, the performance of the benchmarks is no longer affected when using zero-copy memory.

It should be noted that the performance of the *myocyte* benchmark improves on both platforms when using either zero-copy or unified memory. As shown in Table 2, this benchmark performs 7800 host-to-device and 7800 device-to-host memory transfers, which are completely removed when using XeroZerox. This means that the improvement in performance when using a centralized memory model is more evident in applications that have a high number of memory transfers.

V. RELATED WORK

This section provides an overview of the existing literature about optimization of CPU-GPU communication and memory management for embedded platforms. Otterness et al. [29] perform a comparison among the three CUDA memory management policies and conclude that unified memory can benefit applications only in some scenarios. Dashti and Fedorova [30] compare the performance of applications adopting different programming frameworks under the unified memory mechanism in integrated CPU-GPU systems. They also propose a new configuration scheme called *sharedalloc*, which allows programmers to manually flush CPU and GPU caches to allow concurrent accesses to data. Lin et al. [31] present vectorization and scheduling methods to effectively exploit multiple forms of parallelism for throughput optimization on hybrid CPU-GPU platforms, while conforming to system-level memory constraints. They show that their methods can significantly improve system throughput compared to previous approaches. Aghilinasab et al. [32] propose a memory bandwidth allocation scheme to protect real-time GPU kernels from memory-intensive, best-effort CPU tasks. Bateni et al. [33] explore the performance implications of

GPU memory management methods in integrated CPU-GPU architectures. Through experiments on micro-benchmarks and real-world workloads, they find that the performance under different memory management methods may vary depending on application characteristics. Lumpp et al. [34] present a framework based on a performance model, micro-benchmarks, and a zero-copy communication pattern to accurately estimate the potential speedup of applications considering different communication models in embedded platforms. Related to this work, De Marchi et al. [35] present efficient techniques to implement CPU-GPU communication, in compliance with the Robot Operating System (ROS). To the best of our knowledge, XeroZerox is the first GPU memory optimization tool that uses accurate information extracted from the memory allocator properties to adapt the memory model of embedded GPU applications to work with critical setups, minimizing at the same time their memory consumption.

VI. CONCLUSION

In this article, we presented XeroZerox, a tool designed to minimize the memory consumption and memory management overhead of legacy GPU applications when executed in embedded GPU platforms. To evaluate XeroZerox, we applied it to the benchmarks of the Rodinia benchmark suite and executed the resulting applications in two different embedded GPU platforms. The results show that XeroZerox can reduce to approximately 50% the memory consumption in applications where the use of memory is evenly distributed between host allocations and device allocations. In terms of performance, we provided results showing the impact of zero-copy and unified memory on the performance of the applications, depending on the coherency mechanisms used by the underlying platform.

In the Jetson TX2, we observed that zero-copy memory could negatively affect the performance of cache-dependent applications due to the absence of hardware I/O coherency mechanisms. On the other hand, in the Jetson Xavier NX, which includes hardware I/O coherency, the performance of the benchmarks was not significantly affected. Finally, we observed that even when XeroZerox can improve the performance of an application by eliminating the memory transfers and reducing the interaction with the runtime system, the performance improvement can be negligible in applications with a small number of memory transfers.

REFERENCES

[1] S. Alcaide, L. Kosmidis, C. Hernandez, and J. Abella, "High-integrity GPU designs for critical real-time automotive systems," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2019, pp. 824–829.

[2] A. Sigala and B. Langhals, "Applications of unmanned aerial systems (UAS): A Delphi study projecting future UAS missions and relevant challenges," *Drones*, vol. 4, no. 1, p. 8, 2020.

[3] T. Yang, Q. Xu, F. Meng, and S. Zhang, "Distributed real-time image processing of formation flying SAR based on embedded GPUs," *IEEE J. Sel. Topics Appl. Earth Observ. Remote Sens.*, vol. 15, pp. 6495–6505, 2022.

[4] NVIDIA Corporation. *Embedded Systems With Jetson*. Accessed: Sep. 2023. [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/

[5] NVIDIA Corporation. *Autonomous Machines: The Future of AI*. Accessed: Sep. 2023. [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/

[6] NVIDIA Corporation. *CUDA C++ Programming Guide*. Accessed: Aug. 2023. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[7] OpenGL SC Version 2.0.1 (Full Specification). (2019). *OpenGL SC Version 2.0.1 (Full Specification)*. [Online]. Available: https://registry.khronos.org/OpenGL/specs/sc/sc_spec_2.0.1.pdf

[8] A. J. Calderón, L. Kosmidis, C. F. Nicolás, F. J. Cazorla, and P. Onaindia, "Understanding and exploiting the internals of GPU resource allocation for critical systems," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design (ICCAD)*, Nov. 2019, pp. 1–8.

[9] Valgrind Developers. *Valgrind*. Accessed: Mar. 2024. [Online]. Available: https://valgrind.org/

[10] T. M. Baumann and J. Gracia, "Cudagrind: Memory-usage checking for CUDA," in *Proc. 7th Int. Workshop Parallel Tools for High Perform. Comput.*, Dresden, Germany, Sep. 2013, pp. 67–78.

[11] A. J. Calderón, "Real-time high-performance computing for embedded control systems," Ph.D. thesis, Dept. Comput. Archit., Universitat Politècnica de Catalunya, Barcelona, Spain, 2022.

[12] M. M. Trompouki and L. Kosmidis, "BRASIL: A high-integrity GPGPU toolchain for automotive systems," in *Proc. IEEE 37th Int. Conf. Comput. Design (ICCD)*, Nov. 2019, pp. 660–663.

[13] M. B. Sullivan, N. Saxena, M. O'Connor, D. Lee, P. Racunas, S. Hukerikar, T. Tsai, S. Kumar, S. Hari, and S. W. Keckler, "Characterizing and mitigating soft errors in GPU DRAM," in *Proc. 54th Annu. IEEE/ACM Int. Symp. Microarchitecture*, Oct. 2021, pp. 641–653.

[14] R. Abdullah, H. Zhou, and A. Awad, "Plutus: Bandwidth-efficient memory security for GPUs," in *Proc. IEEE Int. Symp. High-Performance Comput. Archit. (HPCA)*, Feb. 2023, pp. 543–555.

[15] S. Na, S. Lee, Y. Kim, J. Park, and J. Huh, "Common counters: Compressed encryption counters for secure GPU memory," in *Proc. IEEE Int. Symp. High-Perform. Comput. Archit. (HPCA)*, Feb. 2021, pp. 1–13.

[16] S. Yuan, A. Awad, A. W. B. Yudha, Y. Solihin, and H. Zhou, "Adaptive security support for heterogeneous memory on GPUs," in *Proc. IEEE Int. Symp. High-Perform. Comput. Archit. (HPCA)*, Apr. 2022, pp. 213–228.

[17] F. Reichenbach, J. Endresen, and S.-E. Ellevseth, "Maximizing diversity in CPUs: Using GPUs as coprocessors to achieve safety integrity," in *Proc. 12th IEEE Int. Conf. Ind. Informat. (INDIN)*, Jul. 2014, pp. 182–187.

[18] NVIDIA Corporation. *Virtual Memory Management*. Accessed: Nov. 2023. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html?highlight=uva

[19] NVIDIA Corporation. *Unified Memory Programming*. Accessed: Nov. 2023. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html?highlight=uva

[20] NVIDIA Corporation. *CUDA for Tegra*. Accessed: Jul. 2023. [Online]. Available: https://docs.nvidia.com/cuda/cuda-for-tegra-appnote/index.html

[21] A. J. Calderón, L. Kosmidis, C. F. Nicolás, F. J. Cazorla, and P. Onaindia, "GMAI: Understanding and exploiting the internals of GPU resource allocation in critical systems," *ACM Trans. Embedded Comput. Syst.*, vol. 19, no. 5, pp. 1–23, 2020.

[22] A. J. Calderón, L. Kosmidis, C. F. Nicolás, F. J. Cazorla, and P. Onaindia. *GMAI: GPU Memory Allocation Inspector*. Accessed: Sep. 2023. [Online]. Available: https://github.com/ajcalderont/gmai

[23] A. Chatterjee. *Function Interposition in C With an Example of User Defined Malloc()*. Accessed: Nov. 2023. [Online]. Available: https://www.geeksforgeeks.org/function-interposition-in-c-with-an-example-of-user-defined-malloc/

[24] M. Kerrisk. (2021). *Linux Programmer's Manual—ld.so(8)*. Accessed: Aug. 2023. [Online]. Available: https://man7.org/linux/man-pages/man8/ld.so.8.html

[25] S. Chamberlain. (1994). *Using LD, The GNU Linker*. Accessed: Aug. 2023. [Online]. Available: https://ftp.gnu.org/old-gnu/Manuals/ld-2.9.1/html_node/ld_3.html

[26] Free Software Foundation. (2021). *The C Preprocessor—Standard Predefined Macros*. Accessed: Aug. 2023. [Online]. Available: https://gcc.gnu.org/onlinedocs/cpp/Standard-Predefined-Macros.html

[27] A. J. Calderón, L. Kosmidis, C. F. Nicolás, and F. J. Cazorla. *XeroZerox*. Accessed: Sep. 2023. [Online]. Available: https://github.com/ajcalderont/xerozerox

[28] 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 *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*, Oct. 2009, pp. 44–54.

[29] N. Otterness, M. Yang, S. Rust, E. Park, J. H. Anderson, F. D. Smith, A. Berg, and S. Wang, "An evaluation of the NVIDIA TX1 for supporting real-time computer-vision workloads," in *Proc. IEEE Real-Time Embedded Technol. Appl. Symp. (RTAS)*, Apr. 2017, pp. 353–364.

[30] M. Dashti and A. Fedorova, "Analyzing memory management methods on integrated CPU-GPU systems," in *Proc. 2017 ACM SIGPLAN Int. Symp. Memory Manage.*, 2017, pp. 59–69.

[31] S. Lin, J. Wu, and S. S. Bhattacharyya, "Memory-constrained vectorization and scheduling of dataflow graphs for hybrid CPU-GPU platforms," *ACM Trans. Embedded Comput. Syst. (TECS)*, vol. 17, no. 2, pp. 1–25, 2018.

[32] H. Aghilinasab, W. Ali, H. Yun, and R. Pellizzoni, "Dynamic memory bandwidth allocation for real-time GPU-based SoC platforms," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 11, pp. 3348–3360, Nov. 2020.

[33] S. Bateni, Z. Wang, Y. Zhu, Y. Hu, and C. Liu, "Co-optimizing performance and memory footprint via integrated CPU/GPU memory management, an implementation on autonomous driving platform," in *Proc. IEEE Real-Time Embedded Technol. Appl. Symp. (RTAS)*, Apr. 2020, pp. 310–323.

[34] F. Lumpp, H. D. Patel, and N. Bombieri, "A framework for optimizing GPU-iGPU communication on embedded platforms," in *Proc. 58th ACM/IEEE Design Autom. Conf. (DAC)*, Dec. 2021, pp. 685–690.

[35] M. De Marchi, F. Lumpp, E. Martini, M. Boldo, S. Aldegheri, and N. Bombieri, "Efficient ROS-compliant CPU-iGPU communication on embedded platforms," *J. Low Power Electron. Appl.*, vol. 11, no. 2, 2021.

ALEJANDRO J. CALDERÓN (Member, IEEE) received the M.Sc. degree in embedded systems from the University of Mondragon, Basque Country, Spain, in 2016, and the Ph.D. degree in computer architecture from the Polytechnic University of Catalonia, Barcelona, Spain, in 2022, with a focus on the use of embedded GPUs in real-time control systems. In 2018, he joined the Real-Time Systems Team, Ikerlan Technology Research Centre, to develop the Ph.D. thesis. Currently, he is a Researcher with the Dependable Embedded Systems Department, Ikerlan Technology Research Centre.

LEONIDAS KOSMIDIS (Member, IEEE) is currently a Senior Researcher with Barcelona Supercomputing Center (BSC) and a Faculty Member with Universitat Politècnica de Catalunya (UPC). He is leading the research on embedded GPUs and accelerators for safety critical systems both at the hardware and system software level within the Computer Architecture/Operating Systems (CAOS) Group, BSC. He is the Principal Investigator of several projects funded by European Space Agency (ESA), European Commission and Industry, which focus on the adoption of GPUs in space and avionics systems, including their certification. He is also participating in standardization efforts regarding GPU programming in safety-critical systems within Khronos.

CARLOS-F. NICOLÁS received the degree in physics from the University of Valladolid, Spain, in 1988, and the Ph.D. degree in engineering from the University of Mondragon, Basque Country, Spain, in 2017. In 1988, he joined the Ikerlan Technology Research Centre, collaborating on several automation-related projects. He is currently a Senior Researcher with the Dependable Embedded Systems Department, Ikerlan Technology Research Centre, where he works on the application of model-based design and virtualization techniques to efficiently develop and commission programmable real-time control systems on heterogeneous computing platforms.

FRANCISCO J. CAZORLA is currently the Co-Head of the group on interaction between the Computer Architecture/Operating Systems (CAOS), Barcelona Supercomputing Center (BSC). He has coordinated three European projects of the FP7 and H2020 Program in the area of software timing analysis in critical computing systems and several projects with European Space Agency. He is an ERC Consolidator Grant Holder. He has authored over 200 papers in top conferences and journals in the area of computer architecture, high performance, and real-time computing systems.