GMAI: A GPU Memory Allocation Inspection Tool for Understanding and Exploiting the Internals of GPU Resource Allocation in Critical Systems

ALEJANDRO J. CALDERÓN, Universitat Politècnica de Catalunya & Ikerlan Technology Research Centre
LEONIDAS KOSMIDIS, Barcelona Supercomputing Center (BSC)
CARLOS F. NICOLÁS, Ikerlan Technology Research Centre
FRANCISCO J. CAZORLA, Barcelona Supercomputing Center (BSC)
PEIO ONAINDIA, Ikerlan Technology Research Centre

Critical real-time systems require strict resource provisioning in terms of memory and timing. The constant need for higher performance in these systems has led industry to recently include GPUs. However, GPU software ecosystems are by their nature closed source, forcing system engineers to consider them as black boxes, complicating resource provisioning. In this work we reverse engineer the internal operations of the GPU system software to increase the understanding of their observed behaviour and how resources are internally managed. We present our methodology which is incorporated in GMAI (GPU Memory Allocation Inspector), a tool which allows system engineers to accurately determine the exact amount of resources required by their critical systems, avoiding underprovisioning. We first apply our methodology on a wide range of GPU hardware from different vendors showing its generality in obtaining the properties of the GPU memory allocators. Next, we demonstrate the benefits of such knowledge in resource provisioning of two case studies from the automotive domain, where the actual memory consumption is up to 5.6× more than the memory requested by the application.

ACM Reference Format:
Alejandro J. Calderón, Leonidas Kosmidis, Carlos F. Nicolás, Francisco J. Cazorla, and Peio Onaindia. 2020. GMAI: A GPU Memory Allocation Inspection Tool for Understanding and Exploiting the Internals of GPU Resource Allocation in Critical Systems. *ACM Trans. Embedd. Comput. Syst.* 19, 5, Article 34 (November 2020), 23 pages. https://doi.org/10.1145/3391896

1 INTRODUCTION

In the domain of critical real-time systems we find a wide spectrum of computer systems. On the one end of the spectrum we have safety critical systems, ranging from transportation to medical and control systems. Since human lives are at stake, such systems usually have hard real-time requirements, which means that their correct behaviour is dictated not only by correct functionality but also by their timely execution with respect to predefined deadlines. On the other end we find business and mission critical systems which although do not impose a threat to human safety, their correct and timely execution is essential to fulfil their mission, typically to provide valuable services to science, society and economy. Examples of such systems are banking and commerce services, communications and scientific space missions, which have somewhat less strict timing requirements, but they are still important for their operation and justification of their high cost.

Despite that these systems are very diverse and have very different particular requirements, all of them have a common property: they require high availability. The key to achieve high availability is the careful resource provisioning

Authors' addresses: Alejandro J. Calderón, Universitat Politècnica de Catalunya & Ikerlan Technology Research Centre; Leonidas Kosmidis, Barcelona Supercomputing Center (BSC); Carlos F. Nicolás, Ikerlan Technology Research Centre; Francisco J. Cazorla, Barcelona Supercomputing Center (BSC); Peio Onaindia, Ikerlan Technology Research Centre.

of the system, in order to guarantee that each of the tasks of the system has enough resources to be efficiently executed, without at the same time exceeding a limit that can jeopardise the entire system or impact the other tasks.

In particular, one of the most extreme cases of resource provisioning is found in avionics [14], whose operating system standard, namely ARINC653 [2] enforces strict memory and time budgets for each task. This requires that the system engineer needs to figure out the exact memory usage of each task and ensure that the total memory usage does not exceed the size of the system memory. Similarly in timing, the worst case execution time of each task has to be determined, and ensure that it is smaller than its deadline and that the overall system has enough capacity to accommodate the execution of all tasks. Automotive operating systems, AUTOSAR-compliant [3], follow a similar approach in resource allocation, as well as the operating systems in other critical domains like the Integrity RTOS which is used in industrial control systems [9].

In less critical systems built on general purpose operating systems like Unix-based ones, although the operating systems do not impose these limitations for each task, system engineers still perform the same type of analysis. For example, although these operating systems do allow the use of more memory than the one physically present in the system, based on virtual memory and disk-backed memory (a feature known as *paging* or *swap*) and/or compression, the performance of the system is severely affected when this feature is used, compromising its timing behaviour and under heavy memory pressure even the stability of the system is jeopardised. Therefore, the accurate resource provisioning allows to prevent such scenarios, guaranteeing that the total capacity of the system is not exceeded.

A recent trend in the critical domains is the introduction of GPUs, in order to satisfy the performance demand of advanced features. Probably the most well known case is in automotive, where automakers are working on autonomous driving prototype vehicles [16] powered by GPUs mainly for cognitive tasks and artificial intelligence. The medical domain and finance are also employing GPUs [28] mainly for image processing and high-computational capacity, as well as the space domain [7]. Other critical domains are expected to follow as well, especially whenever there is a need for inference based on artificial intelligence (AI) or high compute performance.

The GPU market lead vendor NVIDIA has performed significant investments in the automotive and industrial automation sector by designing embedded GPU systems meeting the temperature and reliability needs of these markets, such as the NVIDIA PX2 and its development board Jetson TX2, the NVIDIA Xavier and its latest addition NVIDIA Jetson Nano. Other vendors such as Imagination Technologies have also automotive compliant products like the PowerVR Series6XT GX6650 GPU incorporated in the Renesas R-CAR H3 platform or the latest product lines PowerVR Series8 and Series9, as well as ARM which recently announced its collaboration with Samsung to produce the GPU platform Exynos Auto V9 which will be used in Audi's cars, based on its Mali-G76 GPU.

Despite the important performance benefits provided by GPUs, they are notoriously known about their closed source nature. In particular, NVIDIA GPUs are programmed in CUDA, a proprietary programming language developed by NVIDIA. The GPU execution model in its rudimentary form follows an accelerator approach, in which the programmer has to explicitly allocate GPU memory and manage transfers between the CPU and the GPU, as well submitting code to be executed in the GPU, known as *kernel*. Although this explicit resource allocation provides the delusion of full control over the resource management, the actual resource consumption both in memory and timing is larger, hidden behind closed source layers. The reason is that the actual resource management takes place within the CUDA runtime and GPU driver, which are closed source. The GPU products from other vendors are programmed in OpenCL, which despite its name is not more open. OpenCL has a similar programming model to CUDA, which is also implemented in a closed source runtime and GPU driver provided by each vendor.

As a consequence, an accurate resource provisioning of GPU applications is complicated, leading either to under-estimation or overestimation of resource provisioning. Although this problem is not yet very evident in the existing under-utilised prototype systems, based on Unix-like operating systems e.g. Linux, it will soon be a roadblock as these systems will require the consolidation of more software functionalities in the same platform. Even more importantly, the problem will be more pronounced when these systems will be moved to operating systems for critical systems with strict and explicit resource provisioning per task like AUTOSAR and ARINC653.

In this work, we expose for the first time the internal resource allocation mechanism of a GPU system. This way, we allow the accurate resource provisioning for a GPU-based critical system. First we review the different types of memory allocation in a GPU system and we start by demonstrating the basis of our methodology with a small motivational example. Next we present some essential background on memory allocators and we describe in detail our methodology to discover the properties of the memory allocator used in a GPU-based system. Subsequently we present the implementation of GMAI, GPU Memory Allocator Inspector, a tool which allows to extract automatically the properties of the memory allocator of any GPU and allows to analyse the memory consumption of GPU applications written in both CUDA and OpenCL. Finally, we present our findings for a wide range of GPUs from different vendors and we use the information obtain from GMAI about the internals of the memory allocator to demonstrate the benefits of accurate resource provisioning with two case studies for a critical system, showing that the actual memory consumption is significantly higher than the one requested by the software.

2 MEMORY ALLOCATION IN GPUS

2.1 Memory Allocation in CUDA

Before we enter into the GPU memory allocation internals, it is essential to review the programmer's view of memory management in order to better understand its internal behaviour. As already mentioned, in the CUDA programming model, the programmer is in charge of explicitly managing memory for both the CPU and the GPU side, including allocation, deallocation and transfers between the CPU and the GPU.[1]

Regular CPU memory ie. allocated using `malloc` or `mmap` is by default *paged*, which means that the operating system can swap it out to the disk if needed, typically due to memory oversubscription. On the other hand, GPU memory, allocated with `cudaMalloc` is always non-paged, that is, it is always present in the memory. Copies between CPU and GPU memory are performed by DMA (Direct Memory Access) operations. However, as DMA transfers are asynchronous with respect to the CPU execution, they can operate only when the pages are guaranteed to be resident in the memory. Since this is not always the case for paged memory, the transfers need to pass from a staging area of non-paged memory. In other words, in a CPU to GPU transfer, memory needs to be copied first to this intermediate buffer using the CPU and therefore synchronously, before the DMA can kick in to perform the asynchronous transfer to the device. This results in additional memory, which can be shared among applications, and additional timing overhead in GPU transfers.

In order to avoid these overheads, the programmer can allocate non-paged CPU memory, also known as *pinned* memory or *paged-locked* using `cudaMallocHost`. However, this type of memory in the system is limited and its allocation is more expensive since it requires a user space to kernel space switch. This allows the use of fully asynchronous transfers using `cudaMemcpyAsync`.

[1] CUDA also provides a feature called Unified Memory, which takes this responsibility away. Despite the increase in productivity, the performance of this feature heavily depends on the application's memory access patterns and it adds even more black-box behaviour to the memory management and its timing, which makes it less suitable for critical systems. For this reason, we do not discuss this feature in the rest of this paper.

Last, there is the option to allocate another type of *pinned* memory in the CPU side, which is also memory mapped to the GPU, using `cudaHostAlloc` and specifying the flag `cudaHostAllocMapped`. This means that no explicit copies are required between the CPU and GPU, which gives the name *zero-copy*. Depending on the type of the GPU, this is implemented in a different way. In a discrete GPU, ie. GPUs with their own physical DRAM memory, the copies are performed in a fined-grained manner using the DMA engines to transfer the data over the PCIe link. On the other hand, in embedded (integrated GPUs) which share the same main memory with the CPU, the GPU directly accesses the same memory as the CPU. Of course in both cases it is up to the programmer to ensure the consistency of the shared memory between CPU and GPU. This functionality is supported by a feature known as UVA (Unified Virtual Addressing), which allows both the CPU and the GPU to operate using the same virtual address. It is worth to note that UVA is not the same with Unified Memory, which as we explained is not appropriate for critical systems and therefore is not considered in this study. On the contrary, Unified Memory is implemented using the UVA feature.

2.2 Memory Allocation in OpenCL

OpenCL follows a similar programming model with CUDA, however it is a lower level language than CUDA. This means that the same functionality is implemented with more API calls which offer finer grained control, at the expense of programming complexity.

In OpenCL, memory allocations are handled via memory objects, using the `cl_mem` type. Generic memory allocations are known as *buffer objects* and are created using the `clCreateBuffer` function. This function receives a *flags* parameter which controls how the memory will be accessed by the device. By default, the memory allocated by this function belongs to the device, and the value of flags is `CL_MEM_READ_WRITE`, which indicates that the device can read and write in this region of memory. However, it also can be configured to be read-only using the `CL_MEM_READ_ONLY` flag or write-only using the `CL_MEM_WRITE_ONLY` flag.

The flags parameter can also be used to specify which kind of memory will be allocated. The `CL_MEM_USE_HOST_PTR` flag indicates that OpenCL should use the memory referenced by a host pointer passed to the function instead of allocating a new memory region. This means that the user is responsible of allocating this region of host memory before calling the `clCreateBuffer` function. According to the standard, OpenCL implementations can cache the contents of the host memory region in device memory.

The `CL_MEM_ALLOC_HOST_PTR` flag indicates OpenCL to create the allocation using host accessible memory. Usually this is the flag used to allocate pinned memory in OpenCL implementations. The `CL_MEM_ALLOC_HOST_PTR` flag can be used with the `CL_MEM_COPY_HOST_PTR` flag to initialize the contents of the new allocated memory with the values stored in a buffer referenced by a host pointer passed to the function. To read or write data to a buffer created using the `CL_MEM_ALLOC_HOST_PTR` flag, the user should use the `clEnqueueMapBuffer` function to map the memory region, operate on the buffer and then use the `clEnqueueUnmapMemObject` function to unmap the memory region.

The zero-copy behaviour is not explicitly defined in the OpenCL standard. It is important to take into account that the OpenCL standard is just a set of abstract definitions, and each vendor is responsible for their implementation. For example, in the Intel implementation of OpenCL, the buffers created using the `CL_MEM_ALLOC_HOST_PTR` and `CL_MEM_USE_HOST_PTR` flags are by default zero-copy buffers [13], however, this may not be the case with the implementation of other vendors.

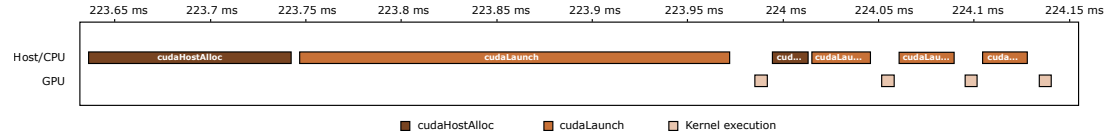

Fig. 1. Execution times for GPU related calls shown in Listing 1 with same size, using zero-copy allocations.

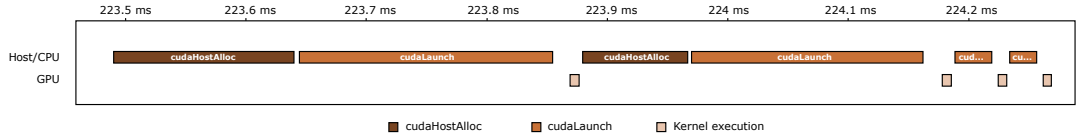

Fig. 2. Execution times for GPU related calls shown in Listing 1 with different size, using zero-copy allocations.

3 MOTIVATIONAL EXAMPLE

Now that we have a clear idea about the different memory allocation options in both CUDA and OpenCL, we can see a motivational example which explains the need for understanding the internals of GPU memory allocations. In our example we use CUDA since it is more concise.

```
Allocate X bytes;
Launch kernel;
Allocate Y bytes;
Launch kernel;
```

Listing 1. Motivational Example

We execute the code shown in Listing 1 on a Jetson TX2 platform, which is an embedded NVIDIA platform with an integrated GPU and we measure the execution time of the 4 GPU-related calls shown in the listing using `nvprof`, NVIDIA's profiler.

In Figure 1 we see the results of running the example with two allocations of the same size (1024 bytes). We notice that the first allocation takes considerable time, while the second one is shorter and the same happens on the first and second kernel launches.

However, when we allocate two chunks of memory with different sizes (1024 and 4096 bytes), we notice that always the first allocation and the first kernel launch for a given memory size take similar time (Figure 2). We notice the same trend for all the 3 different types of allocation introduced in the previous section (paged, pinned and zero-copy).

This observation indicates that the underlying memory allocator implemented in the closed source GPU runtime/-driver manages each of the memory allocations of different sizes in a separate way. The question that is raised is following: *can we determine the internals of this memory allocator, so that we can know the exact system memory allocated and predict which of the GPU related calls are expected to take longer?* In the following sections we will introduce our methodology to discover the memory allocator internals in both CUDA and OpenCL devices. However, as a first step we need to examine common characteristics of memory allocators proposed in the literature for CPUs, since we suspect that the implemented memory allocator is very probable to follow a known design instead of a novel one.

4 BACKGROUND ON MEMORY ALLOCATORS

A memory allocator provides memory to a program when requested and takes it back when the program frees it. It also keeps track of the regions of memory that have been assigned and the regions that are free to assign, using an auxiliary data structure. The main goal of an allocator is to do these tasks in the least possible amount of time, while at the same time minimising memory waste [24].

Initially the memory allocator reserves a contiguous chunk of memory which is used as *pool*, to satisfy dynamic memory requests. When the pool is full, the allocator expands by reserving a new pool. Depending on whether the allocator is implemented in the operating system or at user space, the memory for its pool is reclaimed by using a predefined range of addresses or a preallocated memory region in the former case, or using the `break` or `mmap` system calls in the latter. Custom memory allocators can also use the standard C library calls such `malloc`.

A common challenge for a memory allocator is that programs may free the allocated memory in any order, creating holes between used *blocks*. Note that for efficient representation, block sizes are usually powers of two and they have a minimum *granularity*. The proliferation of small holes leads to the creation of unusable blocks of memory, a problem known as *fragmentation*.

Fragmentation leads to memory waste, incrementing the amount of memory used by the allocator. *External* fragmentation occurs when the available free blocks are too small for the requested size or when the allocator is unable to split bigger blocks to satisfy smaller requests. *Internal* fragmentation occurs when a block larger than needed is assigned, leaving wasted memory inside the block. To avoid fragmentation, techniques like *splitting* free blocks (to satisfy smaller requests) and *coalescing* free blocks (to create larger blocks) are used in conjunction with an allocation policy.

As stated in [10, 18, 24] there are different policies and mechanisms used by memory allocators to manage memory efficiently:

Sequential fits: memory allocators in this category are based in a single linear list to manage the free blocks of memory. A *best fit* allocator searches the smallest free block in the list large enough to satisfy a request. A *first fit* allocator searches from the beginning of the list and uses the first free block large enough to satisfy the request. A *next fit* allocator begins the search from the last used position. A *worst fit* allocator looks for the largest free block in the list.

Segregated free lists: such memory allocators use an array of free lists, having one list for each block size. When a program requests memory, the allocator uses the list with the smallest block size large enough to satisfy the request. The fit of the allocations is not always perfect because the available block sizes are limited, which causes some internal fragmentation. Some segregated free lists allocators use *size classes* to put together a range of sizes in the same list.

Buddy systems: these allocators allocate memory in fixed block sizes which are split in two parts (or coalesced together) repeatedly to obtain blocks of the requested size. A free block can only be merged with it's *buddy*, so coalescing usually is fast.

Indexed fits: some memory allocators, instead of searching sequentially in a free list, use a more complex indexing data structure like a tree or a hash table to keep track of unallocated blocks. The use of this type of indexed structures leads to faster searches and allocations.

Bitmapped fits: these allocators use a *bitmap* to keep a record of the used areas of the heap. A bitmap is a vector of one-bit flags where each bit represents a word in the heap. The search in a bitmap is slower than in an indexed structure, however, the memory consumption is lower because it does not need to store the size of the blocks.

5 REVERSE ENGINEERING GPU MEMORY ALLOCATORS

5.1 Reverse Engineering CUDA Memory Allocators

After reviewing the properties of existing memory allocators, we can design a methodology in order to discover the internals of the GPU memory allocators. Note that we are interested in the key parameters of the memory allocator which affect its memory consumption and timing behaviour, but we are not after obtaining every single detail about its design ie. whether its free list is implemented using a list, tree or a bitmap, since such a task may not be entirely possible to achieve or at least not with a reasonable amount of effort. Furthermore and most importantly it does not affect resource provisioning in the same degree to the other parameters.

Without loss of generality, in this subsection we focus on the same architecture we used for the motivational example. In fact, as we show in Section 7, the same methodology is applicable to all NVIDIA GPUs we tried, ranging from old to bleeding edge GPU models. Moreover, since our methodology does not depend on CUDA, it can also be applied on non-NVIDIA GPUs programmed in OpenCL, as we are going to see in the next subsection.

Starting from the zero-copy allocation scenario, we want to identify the basic design of the memory allocator which is used in order to allocate pinned memory in the CUDA runtime and driver. The fact that the allocation for different sizes results in significantly longer execution times for the first allocation, means that the allocator follows a segregated free list design. Therefore, the next step is to identify its size classes as well as the pool size of each free list. In order to achieve our goal, we design carefully crafted memory allocation experiments and observe their behaviour in order to extract the information we are after. In Section 6 we present a tool that fully automates our methodology and can be executed in any system featuring a GPU to extract its memory allocator properties.

Algorithm 1: Pool size extraction

Output: pool_size

1 Allocate 1 byte of pinned memory
2 Capture `mmap` system call
3 Extract *len* argument from `mmap` system call
4 $pool\_size \leftarrow len$
5 Free memory allocated

Pool Size: In order to identify the pool size of each list, we first create an experiment in which we allocate the minimum amount of memory as shown in Algorithm 1. Since pinned memory has to be requested from the operating system, a user space to kernel space transition based on a system call is required. We monitor the system calls of the executing process using the `strace` utility, which intercepts the system calls as well as their parameters.

We notice that the memory allocation call generates a `mmap` system call, whose second argument corresponds to the size of the memory pool for the list. In our platform, this size is 2MB.

As a validation, running `strace` on the example of Listing 1 reveals a `mmap` only on the first allocation of each size, both with the same size of 2MB, which explains their longer execution time.

Allocation Granularity: Once we know the memory pool size, we need to identify the minimum memory size which corresponds in a single entry within the free list. We achieve this by applying Algorithm 2. The idea is simple: we try to repeatedly allocate the minimum size, until the free list is expanded, by using a new memory pool, which is indicated by a `mmap` call in the `strace`. In our platform, this happens after 4096 allocations, which means that each allocation reserved a 512 bytes entry within the free list.

Algorithm 2: Granularity calculation

Input: pool_size
Output: granularity

```
Allocate 1 byte of pinned memory
allocations ← 1
while a new mmap is not generated do
    Allocate 1 byte of pinned memory
    allocations ← allocations + 1
end while
granularity ← pool_size/allocations
Free memory allocated
```

Algorithm 3: Size classes extraction

Input: granularity
Output: size classes information

```
inferior_size ← granularity
superior_size ← granularity
size_class ← 0
while not all classes extracted do
    Allocate inferior_size bytes of pinned memory
    size_class ← size_class + 1
    while a new mmap is not generated do
        superior_size ← superior_size + granularity
        Allocate superior_size bytes of pinned memory
        Free last allocation
    end while
    Save size_class, inferior_size and superior_size − granularity
    inferior_size ← superior_size
    Free memory allocated
end while
```

Size Classes: Knowing the size of each free list and the allocation granularity, we can focus on detecting how many free lists are kept by the allocator, each corresponding to a different size class. In Algorithm 3 we start creating allocations of increasing sizes, by using the granularity as an increment factor. If a new pool is not created (no new `mmap`) we free the allocation and try the next size. This way we prevent the case that the existing pool used for the current size class is expanded and therefore generating a false positive `mmap`.

In this experiment, we also validate that the pool size and granularity obtained for the first size class using Algorithms 1 and 2 respectively, hold also for each of the other free lists corresponding to the rest of the size classes. However, this validation is not shown in Algorithm 3 for clarity. This is achieved by using the same algorithms, but instead of allocating 1 byte, we allocate minimum size corresponding to the examined size class. We confirm that in all our experiments, these values are consistent among all the size classes for the examined systems described in the Results Section.

Algorithm 4: Best fit ascending test

Input: inferior_size, superior_size
Output: Determines if the policy used is best fit

```
1  for size = superior_size to inferior_size do
2  |  Allocate size bytes of pinned memory
3  end for
4  foreach other_allocation do
5  |  Store size of other_allocation
6  |  Free other_allocation
7  end foreach
8  for size = min_stored_size to max_stored_size do
9  |  Allocate size bytes of pinned memory
10 end for
11 Check if all new allocations were assigned using best fit policy
12 Free memory allocated
```

Allocation Policy: Having obtained all the parameters of the memory allocator, it only remains to identify the policy used in a free list. For this reason, we created validation tests for each type of the four main policies: first fit, best fit, next fit and worst fit. Algorithm 4 shows one these tests checking for the best fit policy. We first create a number of allocations with a decreasing size corresponding to the entire range of allowed sizes for a given size class, so that all allocations are held in the same free list (lines 1-3). Since at this point the free list is empty, each allocation takes the next available free block, resulting in consecutive allocations in the list.

Next, we start freeing every other allocation, creating free blocks of decreasing size and keeping track of their size (lines 4-7). In the final step, we start allocating the same size of blocks that were released in the previous step, but in the reverse order (lines 8-10). That is, each new allocation best fits in the last block of the free list. If the allocator follows a best fit policy, it will result in allocating the same positions as the ones that were freed in the previous step. Otherwise, eg. if the allocator follows a first fit policy, then the allocations would be suboptimal, resulting in an expansion of the original pool.

In order to perform the validation, we use multiple measures. First we use `strace` to validate that there is no expansion of the pool during lines 8-10. Moreover, we keep track of the addresses returned by each and make sure that the new allocations correspond to their best locations, which were their old locations.

Note that the presented example is only one of the variations of the policy validation tests, which are not shown here due to the lack of space and because they are quite similar. In particular, we have versions which perform the allocations in reverse order, or applying the last step (lines 8-10) in random order, in order to check whether the policy instead of best fit follows a LIFO (Last-In First-Out, stack-like) policy. Another variation of this test uses allocations of the same size, in order to identify what is the allocation policy in the presence of multiple equal size blocks.

Coalescing: In this experiment we perform a series of allocations with arbitrary sizes which however can be rounded up to the same size in a given size class. Next, we create two neighbouring free blocks in the middle of the free list. In the following, we allocate a single block with size equal to the addition of the free blocks and we check whether the allocator merges the blocks or creates a new allocation in the free list.

Splitting: This experiment is similar to the previous one, with the difference that only one block is freed in the free list. Then a smaller size block is allocated, to check whether the allocator splits the free block, or the new allocation takes place elsewhere in the free list.

Expansion Policy: For this experiment we perform allocations for a given size class, until the pool is expanded one or multiple times. Then we check whether the pool is expanded when it is full – after allocating exactly the same size of allocations with the pool size – or earlier, when an occupancy threshold in the list is exceeded.

Pool Usage: For this experiment we create multiple pools for a given size class. Then we free a block from the first pool, and perform a new allocation. This way we can check whether the allocation policy is applied across all the pools of the same size, or whether an alternative policy is applied eg. only to the last allocated pool.

Shrinking: Finally, we check whether the memory allocated for expanded memory pools is returned to the system. This is similar to the previous experiment. We perform allocations of the same size class until the memory pool is expanded several times and then we free all the allocations of a given memory pool. We validate whether the memory pool is returned to the system by observing a `munmap` after its last block is freed. Moreover we check whether only a certain memory pool is returned eg. only the last allocated or any of them.

Timing: The methodogy we presented so far corresponded to the case of pinned memory and in particular with zero-copy. In this case, in addition to the `mmap` during memory allocation calls, we obtain also `ioctl` system calls during the kernel launches. These system calls are used in order to communicate with device drivers. We observe that in the first kernel execution after a new pool created for a new size class, the kernel invocation has an extra `ioctl` call. We attribute the longer execution time of these kernels in this additional `ioctl`, which we speculate that is responsible for performing the memory mapping of the host pinned memory to the GPU's MMU (Memory Management Unit).

Paged-memory Allocator: The previously presented methodology is also appropriate without any modifications for the conventional pinned memory allocation, in which there is an one-to-one correspondence of CPU and GPU allocated memory. However, for the memory allocator used for the paged-memory allocations we need a slightly different way to observe its internals.

In particular, the paged-memory allocations do not require a user-to-kernel switch and therefore its parameters cannot be obtained using `strace`. However, we assume that the same allocator design used for pinned memory for CUDA is also used for managed memory within CUDA, in order to reduce development and verification costs. As we comment in the Results Section, this assumption is fully validated. Since `strace` is not applicable in this case, the observation of the memory allocator's behaviour is applied by instrumenting the code with `gdb` in order to obtain the API call parameters and the returned pointers to the allocated blocks. Also, the timing behaviour is observed as previously, using NVIDIA's profiler. With these modifications, the previously presented algorithms are also used to obtain the key properties of the paged-memory allocator, too.

5.2 Reverse Engineering OpenCL Memory Allocators

As we have already mentioned, the OpenCL follows as similar programming model with CUDA. In this subsection, we adapt the algorithms presented in the previous subsection to the memory allocation calls supported by OpenCL.

A significant difference between OpenCL and CUDA is that each vendor has its own implementation of OpenCL, which can result in different memory allocators for each vendor. For illustration purposes we have selected a Mali-T860 GPU as OpenCL reference platform. Using this GPU we have applied our reverse engineering methodology trying to extract the same information we extracted from NVIDIA GPUs.

Pool Size: The same way we did with CUDA, we create an experiment in which we allocate the minimum amount of memory as shown in Algorithm 1, intercepting the generated system calls with the `strace` utility. In this case, the memory allocation also generates a `mmap` system call, whose second argument corresponds to the size of the memory pool for the list. In our OpenCL reference platform, this size is 256KB.

Allocation Granularity: To identify the minimum memory size which corresponds in a single entry within a pool, we also apply Algorithm 2. We create a memory pool and then repeatedly allocate 1 byte of memory until a new `mmap` is generated, which indicates that a new pool has been created. In our OpenCL reference platform, this happens after 4096 allocations. Having a pool size of 256KB, this means that each allocation reserved 64 bytes within the first memory pool.

Size Classes: Having the pool size and the allocation granularity within each pool, we focus on detecting whether the allocator uses size classes. In Algorithm 5 we start creating an allocation with the minimum size. Then, we create allocations of increasing sizes, by using the granularity as an increment factor, until we reach the pool size. If a new pool is not created (no new `mmap` is generated) it means that all possible sizes within a memory pool are compatible, so size classes are not used.

Algorithm 5: Use of size classes

Input: pool_size, granularity
Output: size_classes_used

```
1  inferior_size ← granularity
2  superior_size ← granularity
3  size_classes_used ← false
4  Allocate inferior_size bytes of pinned memory
5  while superior_size < pool_size do
6      superior_size ← superior_size + granularity
7      Allocate superior_size bytes of pinned memory
8      if a new mmap is generated then
9          size_classes_used ← true
10         Free last allocation
11         break
12     end if
13     Free last allocation
14 end while
15 Free first allocation
```

Allocation Policy: To determine the allocation policy used by the allocator we created validation tests for each type of the four main allocation policies, in a similar way we did with CUDA. First we create a simple test doing several allocations of different sizes and releasing some of them in specific positions. After creating the free spaces, we create a new allocation and check which space is used. This way we deduce the allocation policy used.

To validate the deduced policy we use some tests similar to the one shown in Algorithm 4 or its random variant. The main difference between the CUDA implementation and the OpenCL implementation of these tests is that with CUDA we need to check different size classes.

Coalescing: For this experiment we perform a series of allocations with the same size of the internal granularity of a pool. Next, we free two neighbouring allocations to create a continuous free space in the middle of the pool. Then, we

perform a new allocation with size equal to the double of the granularity to check whether the allocator merges the free blocks or creates a new allocation in the pool.

Splitting: This experiment is similar to the previous one, with the difference that only one block is freed in the pool. Then a smaller size block is allocated, to check whether the allocator splits the free block or the new allocation takes place elsewhere in the pool.

Expansion Policy: For this experiment we perform several allocations with the same size of the granularity until a new pool is created. Then we check whether the new pool is created when the previous one is full or when an occupancy threshold in the pool is exceeded.

Pool Usage: To determine how the pools are used when there are several pools created, we perform a series of allocations until we have two full pools and a third one with free space. Next, we create a big free space in the first pool and a smaller one in the second pool. Then, we try to make an allocation with the size of the smaller free space to check if the allocator uses the lastly created pool or if it uses the best space available in the previous pools.

Shrinking: Finally, we check how the memory allocated for the pools is returned to the system. In a similar way we did with CUDA, we perform several allocations to create several pools of memory. Then, we free all the allocations and corresponding to a pool and check when is generated the corresponding `munmap` system call. This way we determine if the pools are returned to the system immediately after freeing its last block or if they are returned at the end of the program.

6 GMAI: GPU MEMORY ALLOCATOR INSPECTOR

Based on the methodology defined in Section 5, we implemented GMAI (GPU Memory Allocator Inspector), which is a tool that can be executed in any system featuring a GPU and extract its memory allocator properties. The GPU Memory Allocator Inspector consists of two parts: the first part is a set of scripts on which we implement the experiments we defined in Section 5 to extract the properties of a GPU memory allocator. The second part is a preload library which can be used to determine the real GPU memory consumption of GPU-based applications. Figure 3 shows the GPU Memory Allocator Inspector workflow. The source code of the GPU Memory Allocator Inspector is available at [5].

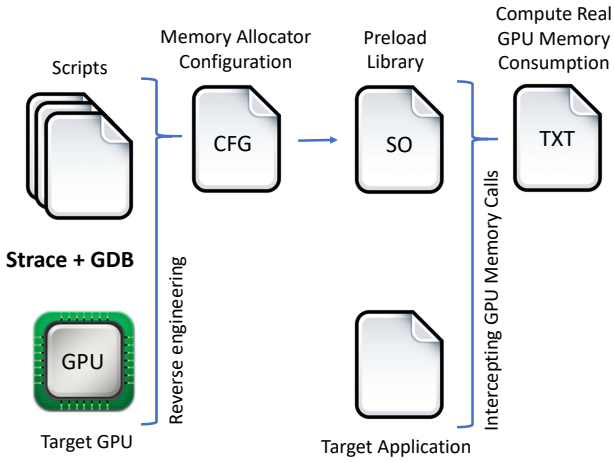

Fig. 3. GPU Memory Allocator Inspector (GMAI) workflow

GMAI can be used in two ways. In the first one we use reverse engineering techniques to extract the memory allocator properties of a target GPU. With this information we generate a configuration file which can be used to visualise the extracted properties. This information could later be used by an engineer to manually analyse the GPU memory consumption of GPU-based applications. However, the second way of using this tool consists on a preload library which can be used to automatically compute the real GPU memory consumption of a target application based on the memory allocator properties stored in the configuration file.

For the implementation of GMAI we have used different debugging techniques. For the initial analysis of the system calls generated by a GPU application we have used the `strace` utility. This way we know what are the functions and parameters we have to look at to get the information we are after for. Using this information, we have used `gdb` to execute our experiments and extract the values we needed in each of them. Having a functional set of `gdb` commands to extract the information for all the experiments, we have automated the debugging process using the `gdbinit` file. This way we automatically generate the configuration file by executing the necessary `gdb` commands over our experiments.

For intercepting the GPU memory allocation calls of a GPU application, we have used the preloading technique, which is a feature of `ld`, the dynamic linker in UNIX-like systems. With preloading we can override arbitrary function calls in a program, by using the environment variable `LD_PRELOAD` to specify a library with our own implementations of the functions we are interested in. With this technique we implemented our own versions of the GPU functions used to allocate memory which we mentioned in Section 2 for both CUDA and OpenCL. In our function versions we keep track of their parameters before calling the actual functions. This way we can intercept those functions from a GPU running application and use their parameters by combining them with the properties of the GPU memory allocator stored in the configuration file to compute the application's real GPU memory consumption.

7 RESULTS

7.1 Obtained Properties of CUDA Allocators for Various GPU Models

In this subsection, we provide the results we have obtained using our methodology on a wide range of NVIDIA GPUs, ranging from very old products with capability 1.1 to the latest NVIDIA's embedded SoC Nano, as shown in Table 1.

Table 1. Tested GPU Platforms

Device Name	Comp. Capabil.	Runtime/ Driver	Kernel Version	GPU Type
GeForce 9300M GS	1.1	6.5	3.19.0	Discrete
Quadro FX 3700	1.1	6.5	3.12.9	Discrete
GeForce GTX 960M	5.0	10.0	4.15.0	Discrete
GeForce GTX 1050 Ti	6.1	9.2	4.15.0	Discrete
GeForce GTX 1080 Ti	6.1	9.2	4.15.0	Discrete
Tegra X1 (Nano)	5.3	10.0	4.9.140	Integrated
Tegra X2 (TX2)	6.2	9.0	4.4.38	Integrated
Xavier	7.2	10.0	4.9.108	Integrated

As explained in the previous Section, we have implemented our methodology in the GMAI tool which automates completely the process. Once GMAI is executed, in a few seconds a report is generated with the information about the memory allocator. In the Listing 2 we can see the generated report about the NVIDIA TX 2 platform, which we used in

the discussion of the previous Sections. In Listing 3 we can see the generated report about the NVIDIA GeForce GTX 1080Ti, which we use as discrete GPU reference platform.

In both GPUs, we observe that the pool size is 2MB and the minimum allocation granularity is 512 bytes. The allocator is using 6 size classes, with the last one ranging up to the pool size. Larger allocations are always rounded up to the next 4KB multiple, which is the system's page size. The allocator is implementing a Segregated Lists Allocator with best fit policy. In the event of expansion, the allocator is keeping a stack of pools. Deallocations can happen to any of the pools, however new allocations are only allocated in the last created pool. Finally, the allocator frees the memory used by any pool when all its blocks are freed.

Regardless of the version of the driver or the hardware, we obtained exactly the same results with the NVIDIA GPUs GTX 1050 Ti and Xavier. For the GPUs GeForce GTX 960M and TX1 Nano we also obtained identical results but with the pool size being 1MB. For the older NVIDIA GPUs, Quadro FX 3700 and GeForce 9300M GS we obtained a pool size of 1MB but 256 bytes granularity.

Our results indicate that the same properties are followed by both the memory allocator for paged and pinned memory, including zero-copy. However, our system call and timing analysis for understanding the sources of variability in the execution time of GPU related calls has revealed that in the newer devices which support UVA (the ones with compute capability more than 2), only the zero-copy scenario is supported, regardless of whether the flag `cudaHostAllocMapped` is used.

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

Allocator policy: Best fit
Coalescing support: Yes
Splitting support: Yes
Expansion policy: When full
Pool usage: Last created
Shrinking support: Yes. Any pool deleted
```

Listing 2. NVIDIA TX2 memory allocator report

```
Device name: GeForce GTX 1080 Ti
Compute capability: 6.1
CUDA runtime version: 9.2
CUDA driver version: 9.2

Pool size: 2097152 bytes
Granularity: 512 bytes

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

Allocator policy: Best fit
Coalescing support: Yes
Splitting support: Yes
Expansion policy: When full
Pool usage: Last created
Shrinking support: Yes. Any pool deleted
```

Listing 3. GeForce GTX 1080 Ti memory allocator report

7.2 Obtained Properties of OpenCL Allocators

In this subsection, we provide results we have obtained applying our methodology to some OpenCL compatible GPUs, which are shown in Table 2. Our main reference platforms for this subsection are the ARM Mali GPUs since they are the first non-NVIDIA GPUs we analyse. We include two OpenCL compatible NVIDIA GPUs for comparison purposes.

As we did with CUDA, we also automated our methodology to extract the information about the memory allocators of OpenCL GPUs by incorporating it in the GMAI tool. Listing 4 and Listing 5 show the generated report about Mali-T860 and Mali-G72 GPUs respectively.

Table 2. Tested OpenCL GPU Platforms

Device Name	Vendor	Architecture	OpenCL Version	Kernel Version	GPU Type
Mali-T860	ARM	Midgard	1.2	4.4.154	Integrated
Mali-G72	ARM	Bifrost	2.0	4.9.78	Integrated
GeForce GTX 1050 Ti	NVIDIA	Pascal	1.2	4.15.0	Discrete
GeForce GTX 1080 Ti	NVIDIA	Pascal	1.2	4.15.0	Discrete

```
Device name: Mali-T860
OpenCL driver version: 1.2

Pool size: 262144 bytes
Granularity: 64 bytes

Size classes: Not used
Large allocations: mmap size next 4KB multiple

Allocator policy: Best fit
Coalescing support: Yes
Splitting support: Yes
Expansion policy: When full
Pool usage: Best available
Shrinking support: No. Pools deleted at the end
```

Listing 4. Mali-T860 OpenCL memory allocator report

```
Device name: Mali-G72
OpenCL driver version: 2.0

Pool size: 262144 bytes
Granularity: 128 bytes

Size classes: Not used
Large allocations: mmap size next 4KB multiple

Allocator policy: Best fit
Coalescing support: Yes
Splitting support: Yes
Expansion policy: When full
Pool usage: Best available
Shrinking support: No. Pools deleted at the end
```

Listing 5. Mali-G72 OpenCL memory allocator report

We observe that the only difference between these ARM GPUs is the granularity of the memory allocator, which is 64 bytes in the Mali-T860 GPU and 128 bytes in the Mali-G72 GPU. The pool size is 256KB in both GPUs and the allocator does not use size classes. This is a significant difference between the NVIDIA memory allocators we examined in the previous subsection. We speculate that this decision is related to the smaller size of memory available in these platforms (4GB) and therefore the memory allocator this way wastes less memory due to internal fragmentation.

Allocations larger than the pool size are rounded to the next 4KB multiple, which is the page size. The allocator implements a best fit policy and supports coalescing and splitting of free blocks. The allocator expands creating new pools when there is no enough space on previous pools. When there are multiple pools with free space, the allocator applies the best fit policy across the pools. However, even when the pools are created consecutively in memory, the

allocator does not use a free region shared by two pools to satisfy the space required by a new allocation. Another difference we observed compared to the NVIDIA allocators, is that in NVIDIA GPUs only the last created pool is used for new allocations, even when there is free space in other pools. Finally, we found out that the OpenCL allocator on ARM GPUs does not free the memory used by a pool when all its blocks are freed, unlike the CUDA allocators. Instead, the memory is released at the end of the program. However, we observed that those regions of memory are reused for the creation of new pools.

We also tested GMAI on two OpenCL compatible NVIDIA GPUs (1050 and 1080) which we analysed in the previous subsection. Repeating these experiments with OpenCL we got the same results shown in Listing 3. This means that the NVIDIA OpenCL implementation internally uses the same memory allocator used by CUDA. When doing these tests we also observed that in Mali GPUs the memory is reserved when we create the corresponding `cl_mem` object. However, in the NVIDIA implementation, the memory pools are created until we map a region of a `cl_mem` object.

7.3 Exploiting the Knowledge of GPU Allocators in Automotive Case Studies' Resource Provisioning

The ultimate purpose of exposing the internals of the GPU allocators, is this knowledge to be leveraged to compute precisely the amount of memory used by critical applications. As explained in the introduction, this will be essential when GPUs will be incorporated in avionics and automotive RTOSes. Moreover, in current general purpose operating systems it allows to make sure that the system can safely accommodate the memory and timing requirements of the application, without the use of unpredictable swap memory.

In order to demonstrate these benefits, we apply our knowledge on two automotive case studies used in modern vehicles' environment perception: a model-based generated safety-critical automotive task, implementing a sobel filter for edge detection and a pedestrian detection task [21]. The former, edge detection, is very common in both ADAS (Advanced Driving Assistance Systems) and autonomous driving for numerous tasks such as lane departure [17], sign [22] and car detection [27]. Pedestrian detection is also used for ADAS, eg. automated breaking as well as for autonomous driving.

As we described in Section 6 when we execute GMAI on a given platform, it generates a configuration file with the properties of the memory allocator. At runtime, we execute the GPU program with the GMAI preloading library which exposes the GPU memory allocation API calls. This way GMAI intercepts all memory requests and their sizes and based on the configuration file, it provides details about the actual memory consumption of the allocator, which we present in the results of the two case studies next.

7.3.1 Edge Detection. Table 3 shows the dynamically allocated memory, explicitly allocated in the program. We notice that the input is a 3-component (RGB) image 640×480 and a 3×3 filter kernel, while the output is a single component 640×480 image, containing the detected edges. Without knowing the internals of the GPU memory allocator, when the task is executed on a platform with zero-copy pinned memory a system engineer might provision 1228809 bytes memory consumption.

Edge detection allocations with CUDA

Table 4 shows the actual memory used by the memory allocator when the edge detection algorithm is executed on an NVIDIA TX2 platform with zero-copy allocations. We notice that we have allocations from two different size classes. This means that two memory pools are created, with 2MB each. Each of these creations will increase the execution time of two memory allocation calls, the first ones corresponding to these class sizes, as well as the execution time of the first kernel invocation following these allocations.

Table 3. GPU memory allocations in Edge Detection Task

Variable	Type	Size
Input Image (640×480)	int8 RGB	921600 bytes
Filter Kernel (3×3)	int8	9 bytes
Output Image (640×480)	int8	307200 bytes
Total:		1228809 bytes

Table 4. GPU Memory allocator usage in Edge Detection (NVIDIA TX2)

Variable	Size Class	Size	Occupied 512b Blocks	Occupied Size
Input Image	6	921600 bytes	1800	921600 bytes
Filter Kernel	1	9 bytes	1	512 bytes
Output Image	6	307200 bytes	600	307200 bytes
Total:				1229312 bytes

Therefore, the total memory consumption to be provisioned is 4MB for this platform and configuration, which is 3.4× more than it was expected, due to internal fragmentation. The memory allocator however is only using a fraction of those. In the first free list, the 3×3 kernel is occupying a single block of 512 bytes instead of 9 bytes due to the minimum block granularity, while in the other free list 1228800 bytes are occupied compared to the 2MB of the pool, resulting in 58% free list occupancy.

On the other hand, in an NVIDIA Nano platform, each memory pool occupies 1MB. However, the two images exceed the memory pool size for size class 6, requiring the memory pool to expand. Therefore the allocator uses 3MB for its pools, which is 2.6× larger that the memory explicitly allocated by the application. In older NVIDIA GPUs like the GeForce 9300M GS, the figures are almost identical, with the difference of the block size of 256, which slightly changes the occupied size in the pool for the filter kernel.

If the application is configured to use pinned memory but not zero-copy, the above numbers are correct, too. The only difference is that in this case both CPU and GPU memory is used, which doubles the aggregate memory consumption.

Finally, if the application is configured to use paged memory, the memory consumption is also doubled because both CPU and GPU memory are used[2]. The difference in this case is that a pinned buffer provided by the operating system is also used for performing the transfers. However, this buffer is shared among different applications and as such it does not need to be taken into account when computing the total memory consumption of the system, when multiple critical tasks are consolidated in the same platform.

Edge detection allocations with OpenCL

Table 5 shows the memory used by the OpenCL memory allocator in an ARM Mali-T860 GPU. In this platform, the memory is allocated in pools of 256KB with a granularity of 64 bytes. However, when an allocation is larger than 256KB the pool size is the next multiple of 4KB. This means that for the input image the allocator will create a pool of 925696

[2] In fact the CPU paged memory consumption in that case is closer to the explicitly allocated memory using `malloc`, since the GNU memory allocator [8] only uses 8 byte aligned blocks in 32-bit platforms and 16 byte aligned blocks in 64 bit ones and it does not use segregated lists. Moreover, the memory pool in CPU is lazily allocated, which means that the OS only reserves the pages of the heap which have been accessed. However, considering equal CPU and GPU memory consumption simplifies the CPU side memory analysis and provides a safe upper bound for a safety critical system in which lazy allocation is not used.

Table 5. GPU Memory allocator usage in Edge Detection (ARM Mali-T860)

Variable	Size	Occupied 64b Blocks	Occupied Size
Input Image	921600 bytes	14400	921600 bytes
Filter Kernel	9 bytes	1	64 bytes
Output Image	307200 bytes	4800	307200 bytes
Total:			1228864 bytes

bytes, using 921600 bytes and leaving 4096 bytes free. For the filter, we only need a block of 64 bytes which can be allocated in the free space of the previously created pool. For the output image the allocator will create a pool of 311296 bytes, which is the next 4KB multiple. The total memory consumption to be provisioned is 1236992 bytes, which is 8183 bytes more than the memory explicitly allocated by the application. However, it only represents a 0.67% of increment. Using a Mali-G72 GPU the only difference is that for the filter the allocator will reserve a block of 128 bytes, since this is the granularity in this platform. However, the results will be the same because in this case the filter can also be allocated in the free space of the pool created for the input image.

7.3.2 Pedestrian Detection. This application is significantly more complex than the previous task and it is obtained from the open source implementation of the benchmark described in [21]. In addition to the input and output images, this task uses a complex dynamically allocated cascade classifier structure. This structure consists of numerous smaller dynamically allocated structures with sizes ranging from 32 bytes to 84 bytes arranged in arrays, requiring a total of 7534 dynamic memory allocations as shown in Table 6. The order in which the memory for these structures is allocated is shown in Listing 6.

```
N_MAX_STAGES = 30;
N_MAX_CLASSIFIERS = 250;
Allocate sizeof(Struct_A) = 32 bytes;
Allocate sizeof(Struct_B) = 480 bytes;
for i = 1 to N_MAX_STAGES do
    Allocate sizeof(Struct_C) = 8000 bytes;
    for j = 1 to N_MAX_CLASSIFIERS do
        Allocate sizeof(Struct_D) = 84 bytes;
    end for
end for
Allocate 307200 bytes for input_image;
Allocate 307200 bytes for output_image;
```

Listing 6. Pseudocode of memory allocations in the pedestrian detection case study [21]

In a zero-copy scenario, the CPU and the GPU can use the same memory, therefore the complex structure can be used as is in the GPU, gaining in programmability.

Table 6 summarises the different GPU allocations of the application. Without knowing the internals of the GPU allocator, a system engineer would provision 1484912 bytes, out of which 870512 correspond to the structure of the classifier.

Table 6. GPU memory allocations in Pedestrian Detection

Variable	Allocs.	Individual Size	Total Size
Input Image (640×480)	1	307200 bytes	307200 bytes
Output Image (640×480)	1	307200 bytes	307200 bytes
Classifier			
Struct A	1	32 bytes	32 bytes
Struct B (30×16 array)	1	480 bytes	480 bytes
Struct C (250×32 array)	30	8000 bytes	240000 bytes
Struct D	7500	84 bytes	630000 bytes
Total:	7534		1484912 bytes

Table 7. GPU Memory allocator usage in Pedestrian Detection Task (TX2)

Variable	Size Class	Individual Size	Occupied 512b Blocks	Individual Occupied Size	Allocations	Total Occupied Size
Input Image	6	307200 bytes	600	307200 bytes	1	307200 bytes
Output Image	6	307200 bytes	600	307200 bytes	1	307200 bytes
Classifier						
Struct A	1	32 bytes	1	512 bytes	1	512 bytes
Struct B	1	480 bytes	1	512 bytes	1	512 bytes
Struct C	3	8000 bytes	16	8192 bytes	30	245760 bytes
Struct D	1	84 bytes	1	512 bytes	7500	3840000 bytes
Total:					7534	4701184 bytes

Pedestrian detection allocations with CUDA

Table 7 shows the actual memory consumption within the memory allocator when the pedestrian detection algorithm is executed on an NVIDIA TX2 platform. Again we notice that the allocations are rounded up to 512 byte multiples, since this is the minimum allocation granularity in the allocator, which penalises small allocations. In this task there are 3 size classes used.

In platforms like the NVIDIA TX2 where the memory pool is 2MB, a single pool is enough for class sizes 3 and 6. However, for the class 1 the total size exceeds 2MB, which requires the free list to expand to accommodate the total of 3841024 bytes required for this size class. Therefore the allocator uses 8MB in total, which is 5.6× more than the initially provisioned one.

For platforms like Nano with 1MB pool size, again the class sizes 3 and 6 can use a single pool, while the class 1 requires 4 pools. Therefore, the total consumption of the allocator is 6MB, 4.2× bigger than the memory explicitly requested by the application.

In the case of paged-memory or pinned memory without zero copy, the complex classifier structure cannot be used, since the pointers it contains are not valid across the different CPU and GPU address spaces. For this reason, the authors of [21] have used a single allocation for the entire structure, which is partitioned accordingly. This is similar to a custom GPU memory allocator, allowing a more predictable behaviour. In that case, a single 870512 bytes allocation is requested, which can fit in a single pool of either 2MB or 1MB depending on the device. Inside this pool, it will occupy 1701 blocks of 512 bytes, occupying 870912 bytes in the free list.

Table 8. GPU Memory allocator usage in Pedestrian Detection Task (Mali-T860)

Variable	Individual Size	Occupied 64b Blocks	Individual Occupied Size	Allocations	Total Occupied Size
Input Image	307200 bytes	4800	307200 bytes	1	307200 bytes
Output Image	307200 bytes	4800	307200 bytes	1	307200 bytes
Classifier					
Struct A	32 bytes	1	64 bytes	1	64 bytes
Struct B	480 bytes	8	512 bytes	1	512 bytes
Struct C	8000 bytes	125	8000 bytes	30	240000 bytes
Struct D	84 bytes	2	128 bytes	7500	960000 bytes
Total:				7534	1814976 bytes

Since only a single size class is actually used in this case (class 6), the total space used inside the free list will be 1485312, so the total memory consumption of the allocator is the 2MB of the free list (same amount divided in 2 free lists for devices with pool size of 1MB, like the Nano), which is 1.4× more than the initially provisioned one. Moreover, as in the previous task, since the application under this scenario requires both CPU and GPU memory, this amount is doubled.

Pedestrian detection allocations with OpenCL

Table 8 shows the memory used by the OpenCL memory allocator in an ARM Mali-T860 GPU. As we show in Listing 4, this allocator does not use size classes, which means that the order in which the allocations are made will dictate the order in which the memory pools will be created and used. Listing 6 shows the order in which the allocations are made. As mentioned earlier, this allocator creates pools of 256KB with a granularity of 64 bytes. For Struct A, the allocator will create a new pool of 256KB and will use only a block of 64 bytes. For Struct B the allocator will use 512 bytes of the previously created pool, leaving 261568 bytes free. On each iteration of the outer loop the allocator will allocate 8000 bytes for Struct C and 32000 bytes for Struct D. The free space in the first pool will be enough for the allocations of the first 6 iterations. To allocate the 960000 bytes required by the other 24 iterations, the allocator will create 4 extra pools of 256KB. Finally, for each image, the allocator will create a pool of 311296 bytes. The total memory consumption to be provisioned is 1933312 bytes, which is 1.3× larger than the memory explicitly allocated by the application. Using a Mali-G72 GPU, which has a granularity of 128 bytes, the difference is that the allocator will reserve 128 bytes for Struct A and 8064 bytes for each allocation of Struct C. However, even with these differences, 5 pools of 256KB are enough to allocate all the structures. For this reason, the memory consumption will be the same.

We notice that in both case studies, the amount of extra memory allocated due to the internal fragmentation in the ARM OpenCL memory allocators is lower than the one NVIDIA platforms when it is compared to the amount of memory requested by the programmer. This difference comes from the fact that the ARM implementations do not use size classes in their memory allocators, for this reason we speculate that this has been a design choice due to the limited amount of memory present in these devices.

8 RELATED WORK

In this Section we present some previous works in the literature similar to our work. We can categorise these works in articles related to resource allocation and reverse engineering techniques in GPUs and CPU memory allocators.

GPU Memory Allocators. Multi-core memory allocators like the one proposed by Berger et al. [4], has been shown not to scale well with many-core architectures like GPUs. For this reason, some authors have approached the GPU resource management topic by creating custom memory allocators suited for many-core architectures:

Huang et al. [11, 12] proposed *XMalloc*, a memory allocator based in two techniques: allocation coalescing (aggregation of memory allocation requests from SIMD-parallel threads to be handled by the CUDA allocator) and buffering of freed blocks for faster reuse using parallel queues. Results on a NVIDIA G480 GPU showed that *XMalloc* magnified the CUDA allocator throughput by a factor of 48.

Steinberger et al. [20] showed that traditional memory allocation strategies used by CPUs are not suited for the use on GPUs and proposed *ScatterAlloc*. This allocator reduces collisions by scattering memory requests using hashing. Experimental results showed that *ScatterAlloc* was about 100 times faster than the CUDA allocator and up to 10 times faster than *XMalloc*.

Widmer et al. [23] proposed *FDGMalloc*, which makes use of the SIMD parallelism present in GPUs to significantly speed-up the allocation of dynamic memory. The authors compared their implementation with the CUDA allocator and with *ScatterAlloc*, achieving a speed-up of several orders of magnitude.

A common characteristic in all these works is that they focus their analysis in comparing the performance of their allocators with the performance of the CUDA allocator, without trying to understand its internal structure or the way it works as we do in this paper. Moreover, these works obtain their memory through the CUDA memory allocator, so they are still susceptible to the timing effects of its usage.

Reverse Engineering Works on GPUs. The black box nature of the GPUs has lead to the creation of some research works oriented to the use of reverse engineering techniques to get information about their internal characteristics.

Wong et al. [25] developed a microbenchmark suite to measure various undisclosed characteristics of the processing elements and memory hierarchies of a NVIDIA GTX280 GPU. Their results validated some of the hardware characteristics publicly available and revealed some other undocumented hardware structures used for control flow and caching. Following a similar approach, Mei et al. [15] exposed previously unknown characteristics about the memory hierarchy of Fermi, Kepler and Maxwell NVIDIA GPUs.

Amert et al. [1] applied black-box experimentation to a NVIDIA TX 2 GPU. Based on results, they defined a set of rules describing the behaviour of the NVIDIA TX2 scheduler. The same group later extended their work on software and disclosed a set of non-obvious pitfalls to avoid when using CUDA-enabled GPUs for safety-critical systems [26].

All these works are based in applying reverse engineering techniques to hardware or software of GPUs, however, none of them is oriented to get information about the memory allocation system and leverage it, which is the focus of our study.

Reverse Engineering Memory Allocators. Eventhough memory allocation is an extensively researched area, the only work to our knowledge related to reverse engineering memory allocators is the *MemBrush* tool, proposed by Chen et al. [6]. The purpose of *MemBrush* is to detect the API functions of custom memory allocators in stripped binaries. *MemBrush* has been used to improve other reverse engineering tools like *Howard* [19], which is used to extract data structures from C binaries without having any symbol tables.

To the best of our knowledge, our paper is the first work oriented to extract information (real memory usage, size classes and allocation policy) about a closed source GPU memory allocator and to analyze the benefits of this information for critical systems.

9 CONCLUSIONS

In this paper we presented a methodology and an automated tool to extract information about the internals of the GPU memory allocators. We applied our method in a wide range of GPUs from different vendors, supporting both CUDA and OpenCL. We identified that there is only a slight difference between different CUDA GPUs, in the amount of memory used internally as a pool and the granularity, in particular in older GPUs. On the other hand, we found out that OpenCL memory allocators do not use different size classes, but they serve all their memory allocations from a single size class.

We have presented GMAI (GPU Memory Allocation Inspector) which allows the extraction the memory allocator properties in an automatic way and based on this information it enables the computation of the actual memory consumption of GPU applications.

We have applied GMAI in two safety critical automotive case studies, showing how a system engineer can be benefited by this information, in order to provision the correct amount of memory. In particular we have shown that the actual memory consumption of the memory allocator can be up to an order of magnitude higher than the amount requested by the application, by running our tests in several GPUs from various vendors.

ACKNOWLEDGEMENTS

This work has been partially supported by the Spanish Ministry of Science and Innovation under grant TIN2015-65316-P, the HiPEAC Network of Excellence and the European Research Council (ERC) under the European Union's Horizon 2020 Research and Innovation programme (grant agreement No. 772773). Leonidas Kosmidis is also funded by the Spanish Ministry of Economy and Competitiveness (MINECO) under a Juan de la Cierva Formación postdoctoral fellowship (FJCI-2017-34095).

REFERENCES

[1] T. Amert, N. Otterness, M. Yang, J. H. Anderson, and F. Donelson Smith. 2018. GPU Scheduling on the NVIDIA TX2: Hidden Details Revealed. In *Proceedings - Real-Time Systems Symposium*, Vol. 2018-January.

[2] ARINC. 2010. Avionics Application Software Standard Interface: ARINC Specification 653P1-3. Aeronautical Radio.

[3] AUTOSAR. [n.d.]. AUTOSAR. https://www.autosar.org Accessed April 2019.

[4] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. 2000. Hoard: A Scalable Memory Allocator for Multithreaded Applications. In *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*. 117–128.

[5] A. J. Calderón, L. Kosmidis, C. F. Nicolás, F. J. Cazorla, and P. Onaindia. [n.d.]. GMAI: GPU Memory Allocation Inspector. https://github.com/ajcalderont/gmai

[6] X. Chen, A. Slowinska, and H. Bos. 2013. Who Allocated My Memory? Detecting Custom Memory Allocators in C Binaries. In *Proceedings - Working Conference on Reverse Engineering, WCRE*. 22–31.

[7] R. L. Davidson and C. P. Bridges. 2018. Error Resilient GPU Accelerated Image Processing for Space Applications. *IEEE Transactions on Parallel and Distributed Systems* 29, 9 (2018), 1990–2003.

[8] Free Software Foundation. 2019. The GNU Allocator. https://www.gnu.org/software/libc/manual/html_node/The-GNU-Allocator.html Accessed April 2019.

[9] Green Hills Software. 1996. Integrity RTOS. https://www.ghs.com/products/rtos/integrity.html Accessed April 2019.

[10] Y. Hasan and J. M. Chang. 2006. A Tunable Hybrid Memory Allocator. *Journal of Systems and Software* 79, 8 (2006), 1051–1063.

[11] X. Huang, C. I. Rodrigues, S. Jones, I. Buck, and W. Hwu. 2010. XMalloc: A Scalable Lock-Free Dynamic Memory Allocator for Many-Core Machines. In *Proceedings - 10th IEEE International Conference on Computer and Information Technology, CIT-2010, 7th IEEE International Conference on Embedded Software and Systems, ICESS-2010, ScalCom-2010*. 1134–1139.

[12] X. Huang, C. I. Rodrigues, S. Jones, I. Buck, and W.-m. Hwu. 2013. Scalable SIMD-Parallel Memory Allocation for Many-Core Machines. *Journal of Supercomputing* 64, 3 (2013), 1008–1020.

[13] Intel Corporation. [n.d.]. Getting the Most from OpenCL 1.2: How to Increase Performance by Minimizing Buffer Copies on Intel Processor Graphics. https://software.intel.com/en-us/articles/getting-the-most-from-opencl-12-how-to-increase-performance-by-minimizing-buffer-copies-on-intel-processor-graphics Accessed October 2019.

[14] L. Kosmidis, C. Maxim, V. Jegu, F. Vatrinet, and F. J. Cazorla. 2018. Industrial Experiences with Resource Management under Software Randomization in ARINC653 Avionics Environments. In *IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers, ICCAD*.

[15] X. Mei and X. Chu. 2017. Dissecting GPU Memory Hierarchy through Microbenchmarking. *IEEE Transactions on Parallel and Distributed Systems* 28, 1 (2017), 72–86.

[16] NVIDIA Corporation. [n.d.]. Self Driving Cars. https://www.nvidia.com/en-us/self-driving-cars Accessed April 2019.

[17] U. Ozgunalp. 2018. Combination of the Symmetrical Local Threshold and the Sobel Edge Detector for Lane Feature Extraction. In *Proceedings - 9th International Conference on Computational Intelligence and Communication Networks, CICN 2017*, Vol. 2018-January. 24–28.

[18] V. Shah and A. Shah. 2019. *Proposed Memory Allocation Algorithm for NUMA-Based Soft Real-Time Operating System*. Advances in Intelligent Systems and Computing, Vol. 814. 3–11 pages.

[19] A. Slowinska, T. Stancescu, and H. Bos. 2011. Howard: A Dynamic Excavator for Reverse Engineering Data Structures. *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS'11)* (2011).

[20] M. Steinberger, M. Kenzel, B. Kainz, and D. Schmalstieg. 2012. ScatterAlloc: Massively Parallel Dynamic Memory Allocation for the GPU. In *2012 Innovative Parallel Computing, InPar 2012*.

[21] M. M. Trompouki, L. Kosmidis, and N. Navarro. 2017. An Open Benchmark Implementation for Multi-CPU Multi-GPU Pedestrian Detection in Automotive Systems. In *IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers, ICCAD*, Vol. 2017-November. 305–312.

[22] H. Vishwanathan, D. L. Peters, and J. Z. Zhang. 2017. Traffic Sign Recognition in Autonomous Vehicles Using Edge Detection. In *ASME 2017 Dynamic Systems and Control Conference, DSCC 2017*, Vol. 1.

[23] S. Widmer, D. Wodniok, N. Weber, and M. Goesele. 2013. Fast Dynamic Memory Allocator for Massively Parallel Architectures. In *ACM International Conference Proceeding Series*. 120–126.

[24] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. 1995. *Dynamic Storage Allocation: A Survey and Critical Review*. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), Vol. 986. 1–116 pages.

[25] H. Wong, M. . Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. 2010. Demystifying GPU Microarchitecture through Microbenchmarking. In *ISPASS 2010 - IEEE International Symposium on Performance Analysis of Systems and Software*. 235–246.

[26] M. Yang, N. Otterness, T. Amert, J. Bakita, J. H. Anderson, and F. D. Smith. 2018. Avoiding Pitfalls when Using NVIDIA GPUs for Real-Time Tasks in Autonomous Systems. In *Leibniz International Proceedings in Informatics, LIPIcs*, Vol. 106.

[27] R. Younis and N. Bastaki. 2018. Accelerated Fog Removal from Real Images for Car Detection. In *2017 9th IEEE-GCC Conference and Exhibition, GCCCE 2017*.

[28] X. Yu, H. Wang, W. . Feng, H. Gong, and G. Cao. 2019. GPU-Based Iterative Medical CT Image Reconstructions. *Journal of Signal Processing Systems* 91, 3-4 (2019), 321–338.