Real-time high performance computing using a Jetson Xavier AGX

Cyril Cetre, Florian Ferreira, Arnaud Sevin, Rémi Barrere, Damien Gratadour

Real-time high performance computing using a Jetson Xavier AGX

Cyril Cetre[1, 2], Florian Ferreira[2], Arnaud Sevin[2], Rémi Barrere[1] and Damien Gratadour[2]

[1]Thales Research & Technology
[2]LESIA, Observatoire de Paris, Université PSL, CNRS, Sorbonne Université, Université de Paris, 5 place Jules Janssen, 92195 Meudon, France

Abstract

While general purpose graphics processing units now embark tremendous amount of computing power, their use in real time applications is still a challenge. The COSMIC platform, developed in the context of adaptive optics control for giant astronomical telescopes is a demonstrated solution to perform real time computations using discrete GPUs while maintaining a high level of abstraction and modularity. An implementation on embedded platforms with the goal to reach an acceptable level of time-determinism would enable new real-time use cases in other application domains. In this regard, NVIDIA is offering a broad range of embedded systems on chip delivering great performance and compatible with the CUDA ecosystem. However, specific hardware and software features bring uncertainties regarding real-time performance. The approaches presented in this paper rely on COSMIC recipes to expose part of the underlying unconventional GPU programming model to reach real-time performance. It shows how Jetson Xavier performs on sub-millisecond complex pipelines made of several compute kernels, considering the limitations engendered by missing CUDA features as compared to discrete devices, leveraging unified memory to work around these hurdles and enabling several strategies for implementing real-time workflows on embedded GPU platforms.

Keywords: Real-time systems, graphics processing units, High Performance computing, embedded software, CUDA

Introduction

Real-time computing for adaptive optics

Adaptive Optics (AO) systems are used to compensate aberrations in real-time on optical systems. They are a core component of extremely large telescopes for astronomy. They aim at making partial compensation of image distortions induced by atmospheric turbulence in real-time using a set of computer controlled actuators, under the reflective surface of so-called deformable mirrors, to observe astronomical objects at high angular resolution and high contrast. As distortions and external constraints on AO are fluctuating on very short time scales, AO controllers, working in closed loop, need to infere the best actuators commands with minimum and stable time-to-solution (in the range of 1 ms or less). In addition, the complexity of AO systems exacerbates the need for a modular solution which provides the ability to realize flexible computing pipelines.

The COSMIC [1] platform was designed to cover the requirements of a wide variety of AO instruments considering these constraints. This platform relies on off-the-shelf high performance libraries and a modular approach, in order to minimize implementation cost and complexity while maximizing performance and throughput of applications requiring efficient GPU computations. It provides abstraction layers handling data transfers, inter-process communications and synchronisations between computation units of a given pipeline. Each computation of the pipeline is turned into an independent process, allowing individual monitoring and real-time pipeline modifications. The multi-process mechanisms are thoroughly optimized to obtain the best possible performance given the hardware configuration.

While COSMIC was originally designed for AO, its application to other domains remains perfectly valid as it is aiming at providing a framework for real-time computation regardless the pipeline. Therefore, evaluating the use of its programming methods on embedded platforms is meaningful, although opening the door to new challenges. An embedded implementation will have to ensure that performance is preserved with scarce resources while proposing possible solutions to overcome technical limitations. A stable implementation on systems on chip (SoC) could prove useful to implement simpler pipelines with specific use cases such as smart cameras working as wavefront sensors.

Embedded systems and real-time computing

With the exponential growth of the use of deep learning in various domains, SoC with integrated GPU (iGPU) are becoming very popular thanks to their rather high performance per watt. In particular, NVIDIA provides a broad range of consumer products using custom ARM CPU and iGPU. These SoC have many interesting features such as supporting the CUDA ecosystem. This makes cross-platform programming pretty straightforward with few differences between SoC and discrete GPUs (dGPU) [2]. As opposed to a dGPU, which is a separate device from the processor with dedicated memory, one of the main features of iGPU is to share memory with the CPU, which allows to reduce the need for data transfers, although it opens the doors to a number of challenges as well considering the CPU activity

may interfere negatively with the GPU computations.

In any case, achieving real-time GPU computing can be challenging since this technology offers very few safeguards on execution time determinism. That is why average jitter and Maximum Measured Execution Time (MMET) must be assessed carefully. Ensuring such features, combined with a high throughput is a significant hurdle to overcome before bringing GPU into time critical applications.

In addition, enabling workflows involving multiple processes on a GPU brings uncertainties regarding the end-to-end response time. Some functionalities like CUDA Multi Process Service (MPS) are offered to allow kernels from different processes to execute concurrently. However, MPS and some other features are not available on embedded NVIDIA SoC, making multi-process application portability unpredictable on such platforms.

Unconventional programming model for GPU

Ensuring time-to-solution repeatability with very low jitter for a complex pipeline can be particularly tough when relying on multi-process asynchronous GPU computations. While achieving a high throughput is possible when taking into account intrinsic overheads of GPU computing, such as memory transfers, kernel launches or synchronizations between the host and the device, reaching low performance jitter usually requires unconventional programming approaches, such as using persistent kernels [3] [4]. As the name implies, these kernels are not terminated between each iteration and just wait on the arrival of more data to be triggered again. However, such technique heavily depends on hardware features and the CUDA grid/block dimensioning must be handled within the kernel. As a consequence, persistent kernels must be redesigned when either the pipeline or the targeted hardware changes. A proposed trade-off between persistent kernels and traditional programming models is the use of a combination of GPU busy wait kernels and look ahead jobs scheduling, relying on the GPU scheduler to launch new kernels while avoiding CPU/GPU synchronization overheads [1].

This paper exposes this approach through a set of out-of-the box use cases, comparing the discrete and embedded behaviour in order to provide guidance about how to perform real-time multi-process computations with complex applications in an embedded environment.

Focus of this paper

This paper relies on previous work done with the COSMIC platform and extends the best practices implemented therein to provide a suitable solution for critical real-time applications running on embedded platforms in terms of average jitter, worst observed execution time and throughput.

Porting these recipes on Jetson Xavier AGX provides an overview of how architectural and software differences between embedded platforms and integrated GPUs can be worked around to achieve real-time performance.

We highlight these differences and show why it could be a serious impediment to determinism. We also propose workarounds using multi-thread or multi-process implementations of GPU inter-process communication through busy-waiting.

Finally, we evaluate through benchmarking the performance and overall behaviour obtained using different methods for efficient inter-process communication

Related Work

Using GPUs for real-time applications is not straightforward, as it has not necessarily been designed to minimise MMET, although modern GPUs are slowly overcoming technological limitations with increased features and capabilities. [5, 6].

Unlike dGPUs, the CPU and the iGPU share the same SoC DRAM on Tegra devices. As a consequence, CPU activity may interfere negatively with GPU computation and conversely [7]. That is why some authors have proposed to improve determinism by protecting GPU applications from memory throttling through custom schedulers [8, 9]. In addition, several studies are aiming to unveil closed-source details of GPU schedulers to get a better understanding of their behaviour. [10, 11].

On a positive note, shared DRAM implies that data transfers from host to device are avoidable. In such circumstances, pinned host memory accessed from GPU will not have copy overheads and will be bounded by the same bandwidth as memory allocated from device code. Combined with CPU shared memory, it is used in this contribution as a workaround to bypass the lack of CUDA Inter-Process Communication (IPC) features. In addition, previous work has shown that using pinned host memory on embedded systems can be a way to increase determinism by reducing memory requirements of GPU programs [12].

Real-time multi-process computing on GPU

Inherited from a project having strong requirements for scalability, maintainability and modularity [3], COSMIC was designed as a framework able to overcome the underlying limitations by supporting separate kernels that can run either concurrently or sequentially, with an efficient synchronization mechanism. However, a complex software architecture does not necessarily scale well when GPU computations are involved depending on the exact setting.

For instance, the default behaviour of CUDA is not suited for real-time multi-process applications. As every process has its own CUDA context only one context can run at the same time on the GPU, the device has to switch constantly between them, leading to extra jitter. In addition, running kernels with a small number of threads will lead to poor GPU performance as another kernel could have run concurrently. For a single process (and single context) application, the CUDA streams were created to overcome this issue and enable the overlapping of kernels executions.

In the case of multi-process applications, NVIDIA has made various tools available to improve the device management. In this regard, the next sub-section offers an overview of these CUDA features providing a good understanding of their implication and why they are critical for real-time.

CUDA MultiProcess Service (MPS)

CUDA MPS is a binary-compatible client-server runtime implementation of the CUDA API. MPS is especially useful when multiple processes are making use of the GPU, in particular when these processes underutilize GPU resources. Its main benefits are the following :

- Kernels and memcopy operations from different processes may overlap on the GPU.
- MPS handles only one GPU context and set of scheduling resources between its clients instead of one context for each process. This removes the need for context switching between two GPU kernels.

CUDA Interprocess Communication

Part of the CUDA Toolkit and available since compute capability 2.0, CUDA IPC enables sharing device buffers between multiple processes. While the COSMIC memory manager implementation relies on CUDA IPC to share device buffer between processes, these calls are not supported on Tegra device.

GPU busy wait synchronization

At some point, a process will have to wait for another to complete, thus needing to synchronize. Previous work on the COSMIC framework provided an assessment on which kind of synchronization gives the best response time in order to maximize performance [1]. From this study, it appears that busy wait synchronization (meaning different processes actively wait for data to be written on the GPU through memory polling) shows a significant latency improvement compared to a regular CPU based POSIX semaphore synchronization. It has several benefits :

- The pipeline processing is now fully asynchronous from the CPU, allowing the user to hide kernel launch latency by accumulating jobs on the GPU scheduler in advance.
- A CPU, even isolated is still more likely to get non preemptible interruption request from the OS kernel compared to a GPU which is dedicated to computations.

On the other hand, each busy wait process is using a GPU streaming multiprocessor to spin on a given location in memory which is increasing system occupancy and resource consumption.

Figure 1 and 2 shows how busy waiting with CUDA MPS enabled performs compared to POSIX semaphore implementation, revealing non negligible latency improvement on a NVIDIA DGX server (286 μs of average execution time for semaphore and 235 μ s for GPU Busy wait). Even though the semaphore synchronization shows a better average jitter (2.1 μs for semaphore and 4.4 μs for GPU busy wait), this synchronization process is more prone to jitter peaks (semaphore : 35 μs of peak to valley and 30 μs for GPU busy wait). In addition, the NVIDIA DGX server performs especially well to minimize this kind of interference, which tends to show higher peak to valley in the case of semaphore synchronization as compared to other GPU equipped servers.

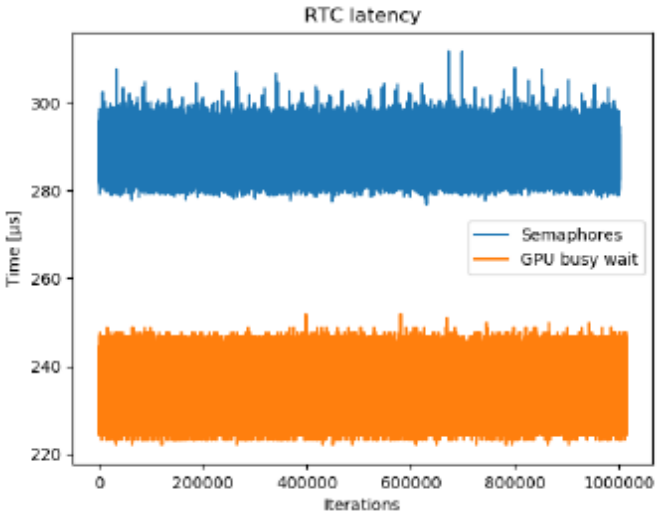

Figure 1 Time-to-solution execution profiles obtained with COSMIC in the context of the AO application, using 2 different synchronization mechanisms over 1M iterations [1]

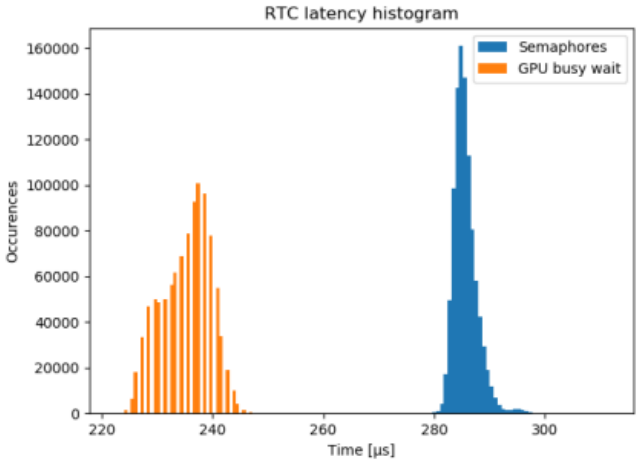

Figure 2 Time-to-solution histogram obtained with COSMIC in the context of the AO application, using 2 different synchronization mechanisms over 1M iterations [1]

As a consequence, the GPU busy wait is preferable both in terms of latency and jitter stability.

Considering the sub-millisecond response time requirement, such gains represents a great asset and are hardly dispensable. Reproducing such results on an embedded SoC would bring us closer to hard real-time embedded GPU computing, although the road ahead is full of obstacles.

The challenge of embedded GPU computing

As GPUs are growing in complexity with features including dedicated cores for specific operations including tensor cores for matrix multiplication or RT cores for raytracing, NVIDIA is regularly proposing new SoCs featuring the lat-

est innovations. As a consequence, each SoC has a specific hardware architecture (CPU and GPU) which makes it difficult to predict performance on the targeted board without comprehensive testing. That is why this paper will focus on NVIDIA Jetson Xavier AGX, which is providing some features (for instance I/O coherency) which were not available on previous NVIDIA boards.

The Jetson Xavier AGX

Released in 2018, the Jetson Xavier AGX is currently the most powerful embedded GPU platform available on the market. It is featuring within its Xavier Tegra SoC a Volta GPU with 512 CUDA cores and 8 streaming multiprocessors.

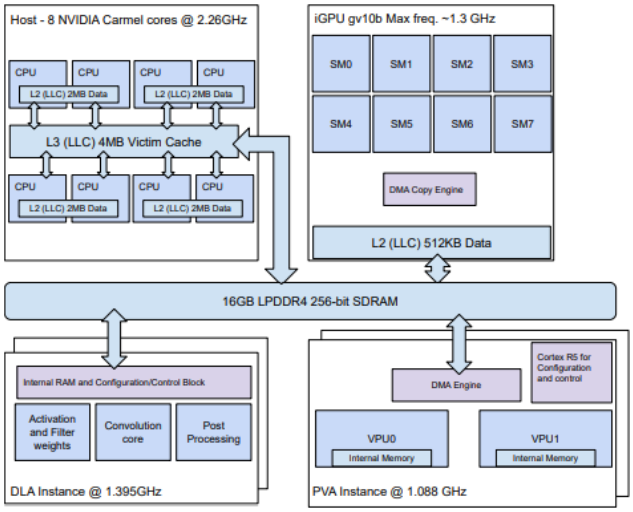

Figure 3 Block diagram of Jetson Xavier complex (credits : [13]

The CPU is composed of 8 ARM 8.2 architecture compliant cores. Regarding the cache hierarchy, it is important to notice that the CPU complex is composed of 4 islands. Although each core has its own private L1 cache, the CPU L2 cache is shared between two threads. Two cores from the same island will interfere if both are executing highly bandwidth dependant tasks. As a consequence, the combined bandwidth of two CPUs of the same island is worse compared to a single core doing heavy bandwidth computations paired with an idle one. [13].

This is why it is recommended to always isolate CPUs by island to avoid extra interference from shared L2 cache on this platform.

Unlike discrete GPU (dGPU), CPU and integrated GPU (iGPU) share 16GB of DRAM clocked at most at 2133 MHz, reaching a theoretical bandwidth of 137 GB/s.

Shared CPU/GPU memory

From a software point of view, shared memory does not mean that a CPU pointer is accessible from GPU and conversely. Depending on how memory is allocated, the cache behaviour differs and memory might not be accessible from both the host and the device. Similarly to dGPU, memory allocated with *cudaMalloc* is not accessible from the host and a buffer allocated with *cudaMallocManaged* will return a buffer accessible from both sides. The main difference with dGPU is that there is no hidden memory copies between the host and the device and the source code should be adapted to make copies only when unavoidable. NVIDIA provides detailed documentation about the cache behaviour depending on how the memory was allocated [2]. Starting with the Xavier architecture, Tegra boards are featuring I/O Coherency which allow I/O devices such as the GPU to read the latest updates in CPU caches. This is beneficial in our case as it allows to cache CPU pinned host memory and to register an existing host memory range for use by CUDA, which is crucial in our proposal to bypass the lack of CUDA IPC.

CUDA IPC/MPS unavailable on Jetson

At the time this paper was written, aforementioned multi-process features are not available on NVIDIA embedded systems. As a consequence, multi-process GPU busy waiting is not viable as it is introducing strong jitter interference due to GPU context switching. We will see in the next sections various options we explored to get performance similar to the classical multi-process approach, followed by experimental results.

Multi-process real-time computing on Jetson embedded platforms

GPU buffer sharing on Jetson Xavier

When trying to reproduce complex GPU pipelines on Jetson, the first obstacle lies in being able to share GPU buffers among processes which is usually done on a discrete device through CUDA IPC. So far, this feature is not available on Tegra SoC although sharing device memory is a core feature of the platform.

Fortunately, on Tegra devices with I/O coherency [2] (starting with the Jetson Xavier, not applicable on previous devices) it is possible to register an existing host memory range for use by CUDA. The identified workaround takes advantage of the CPU and GPU unified memory to keep decent performance while using a mix of POSIX shared memory and zero-copy mechanism which allows a GPU to fetch pinned host data.

Using GPU kernels to access pinned host memory is coming with drawbacks. First, it will result in separated pointers for host and device calls, although using the same memory space. Second, pinned host memory accessed by the GPU will not be cached. As a consequence, user must take extra care when accessing memory to avoid any performance drop.

GPU busy-waiting efficiency on Jetson Xavier

Although the aforementioned shared GPU buffers enables GPU busy wait synchronization, results showed really poor performance using a straightforward implementation. Recording sub-milliseconds computations through a non-intrusive timer (meaning that it is using a separate process in

order to avoid interference with the main pipeline) requires responsive measurements.

Unfortunately, measuring those events on GPU turns out impossible as the GPU timer is unable to start and stop at the right moment while performing sub-millisecond measurements. Considering that this timer has its own process and CUDA context, the GPU scheduler will not necessarily give the hand back to the timer process right after the computation process sends the stopping signal, thus providing misleading results.

Although CUDA preemption mechanism and context loading are not publicly disclosed, we know that the GPU is handling multiple clients (i.e., multiple processes requesting GPU resources) with a time-slicing behaviour. In addition, a GPU always serialise two kernels from different processes, even though the resources to run them in parallel are available.

As a consequence, the scheduler regularly interrupts the pipeline computations to perform GPU busy waiting from other computing units, leading to unintended performance and jitter degradation. Considering that multi-process busy-wait synchronization relies heavily on launching small kernels from different processes, it is impossible to achieve a satisfactory level of determinism in highly constrained environments using regular busy-waiting without CUDA MPS.

Possible workarounds

Thanks to unified memory, potential workarounds are available although they don't completely overcome the lack of CUDA MPS.

Multiple-context CPU active-wait strategy

It is possible to keep going with multiple CUDA contexts. In this case, the important aspect is to avoid doing active waits on GPU at all costs as explained in the previous subsection. Active waiting with CPU on a GPU data buffer is working, although forcing CPU thread spinning. This avoids a synchronization between the host and the GPU as notifications are sent through the device. However, the GPU won't be able to schedule kernels ahead of time as the CPU is now blocking the execution. Adding context switching, this strategy will introduce some overhead compared to straightforward GPU busy-wait.

Semaphore strategy

While having an overhead compared to active wait strategy, implementing POSIX semaphore based signals remains perfectly viable and will suffice in most cases.

Multi-thread strategy for real-time computing

After establishing that the principal constraint is to keep only one CUDA context to get best results when using the GPU busy waiting technique, it seems that an approach relying on both multi-thread and multiple streams is an approach to consider. CUDA streams were specifically designed to allow concurrency on a single process, which is a requirement for this kind of busy-wait.

On the positive side, a multi-threaded implementation removes software dependency to inter-process features (CUDA MPS and CUDA IPC). As a consequence, it is deployable on both discrete and integrated GPU environments as opposed to the multi-process approach.

However, it must be pointed out that the GPU behaviour is not strictly identical. For instance, considering the GPU busy-wait mechanism, it is very important to synchronize the CPU with the CUDA execution at some point. A CUDA context has a limited queue depth in terms of kernel scheduling. When this limit is reached, a forced synchronization occurs on the CPU before it can be able to enqueue more kernels. It is thus very easy to end up with a deadlock, with busy wait kernels exceeding the queue depth leading to the impossibility to send notifications. This limitation must be taken into account by the implementation.

The described behaviour may not be the only difference between multi-thread and multi-process strategies. Unfortunately, as for the multi-process preempting behaviour, CUDA stream scheduling is scarcely documented by NVIDIA although several studies are working toward exposing it in more details [14]. Thus, it is likely that new behaviour specificities will be empirically discovered in the future.

Regarding performance, results with a multi-thread implementation appeared to be very similar to regular MPS multi-processing, which is encouraging.

Environment setup and results

Use Case

For the sake of reproducibility and modularity, the decision was taken to implement a simplified pipeline while keeping only COSMIC core synchronization mechanisms. It allows us to propose multiple test cases, with both embedded and iGPUs whenever necessary.

Figure 4 shows an example of how a given sample test behaves regardless of the synchronization approach and its kind of parallelization, being either multi-threaded or multi-processed. The basic architecture of the pipeline is the following :

- A camera emulator which sends notifications at a given framerate. The notification to start computation is sent to both the timer and the first computation unit.
- A pipeline that can be composed of one or more compute units. In our test environment, it essentially performs matrix vector multiplications (MVM). It sends a notification to the timer once it is done.
- A timer which gets notifications from the camera emulator to start and from the last computation unit to stop.

Each test is performed first in a multi-processed environment, meaning each unit (timer, camera, first MVM, second MVM and so on) are launched as separated processes. The only way these units have to communicate is through shared memory and signals to synchronize, which is done either

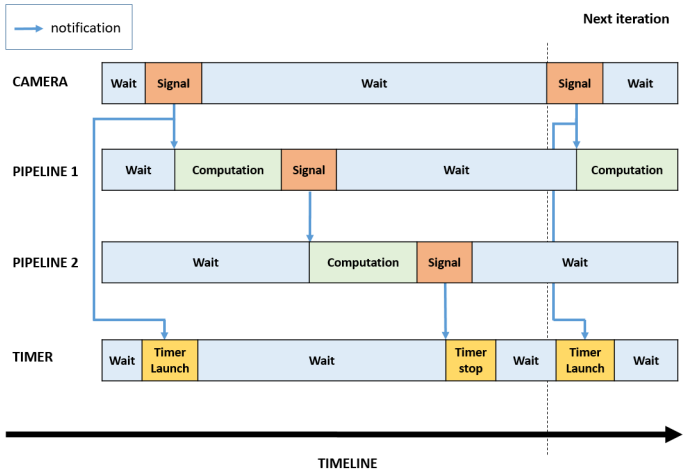

Figure 4 Timeline of a test with two computing units

through POSIX semaphores or GPU busy-wait. In a second experiment, the test is performed with the multi-threaded approach : the computation units are launched in the same process. However, each unit is given a distinct CUDA stream which allows kernel execution overlapping. Both synchronization methods are performed as well.

WCET and MMET

The Worst Case Execution Time is a term typically used in critical real-time systems where reliability and safety is paramount. As of today, it is very unlikely that an embedded environment such as a Jetson can allow a demonstrated WCET such as it is meant in avionics. In order to avoid misconception about the testing process, this paper is addressing Maximum Measured Execution Time (MMET), meaning the worst case measured in the test session with a representative number of executions (1 million). Figure 5 shows how MMET does not necessarily shows the worse possible case, compared to the WCET.

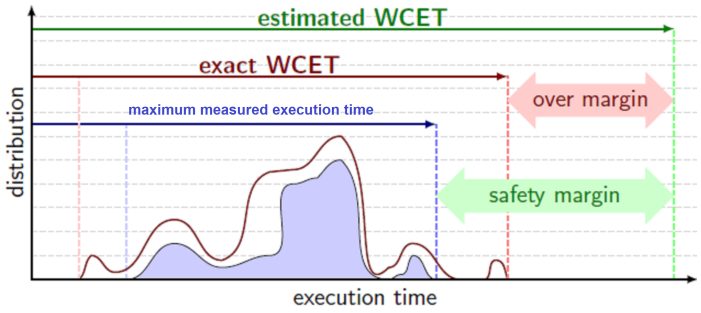

Figure 5 This paper uses maximum measured execution time (MMET) figure credit : Thales Research Technology

Timing measurement

Timing events with asynchronous CPU and GPU interactions can be surprisingly difficult. In order to record overheads between a computation unit sending a signal and another receiving it, the timer must stay in a separate thread. It is also a great asset as a timer needs to synchronize with CPU and GPU in order to get execution time, which is unavoidably introducing overhead. This kind of timer needs the first pipeline unit to send a notification to launch measurement and the last one to send a another one to stop it. The downside is that the timer is not bounded to computation anymore and might miss some iterations at some point if is not awaiting for a signal when a notification is sent.

Making the timer independent of computation units will not introduce interference to computation or forcing unintended CPU/GPU synchronization, ensuring measurement closest to the actual computation time.

The semaphore synchronization approach is measured through the C++ high_resolution_clock API while the GPU busy wait approach is using CUDA events.

Results will be shown through two different displays that highlights different kind of information :

- Histograms are helpful to get a grasp of an approach latency and jitter.
- Execution profiles include each iteration execution time. It is useful to show how an approach is affected by jitter. The wider the plot "line", the more jitter this approach gets. it also shows infrequent outliers, that would not appear on histograms.

Environment

Table 1 and 2 show both configuration used in the following tests.

OS	CentOS 8
Linux kernel	4.18.0-193.19.1.el8 2.x86 64
CPU	15 Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz
CUDA version	11.5
GPU	Tesla V100

Table 1 Benchmark environment setup with dGPU

OS	Ubuntu 18.04
Linux kernel	4.9.253-rt168-tegra
CPU	8-core ARM v8.2 64-bit
CUDA version	10.2
GPU	512-core Volta integrated GPU

Table 2 Benchmark environment setup on Jetson Xavier AGX

Optimizations for real-time

Several optimisations are done to achieve real-time performance. This includes :

- CPU isolation at boot.
- Processes scheduling set to FIFO with high priority.
- Running CUDA MPS server (for dGPU only).

In order to get comparable results across all experiments for a given setup (dGPU or Jetson), MVM are based on a fixed size. The framerate of the camera is set to always let the pipeline finish its job before notifying for the arrival of a new image.

To reach sub-millisecond computation time with both settings, the matrix size is reduced with the Jetson Xavier, giving the following sizes :

- 4096 X 12500 with the Tesla V100.
- 4096 x 1562 with the jetson Xavier.

Experimental results with the Tesla V100

The first step requires to study how different implementations behave with dGPUs compared to known results (Multiprocess with Active wait synchronization and CUDA MPS). Figures 6, 7 and Table 3 shows an experiment with 1M iterations of the same test performed with different parallelization and synchronization techniques. The pipeline is the simplest possible, composed of one camera, one processing unit performing a matrix-vector multiplication (MVM) and a non intrusive timer.

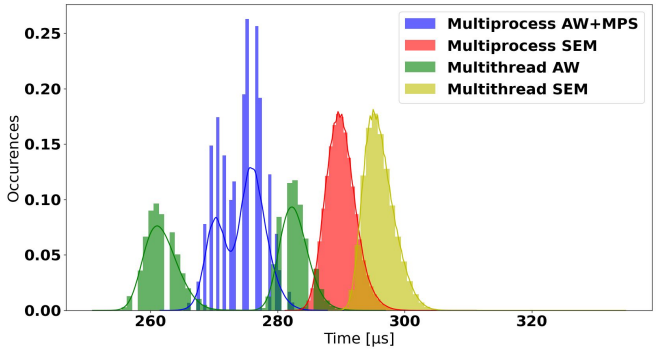

Figure 6 Normalized histograms with their estimated probability density function (solid lines) of a pipeline composed of one MVM unit with various synchronization approaches (SEMaphore and Active Wait) over 1M iteration on DGX server

The best results in terms of latency come from active wait synchronization (AW) regardless if using multi-thread or multi-process approach with 274.3 μs of Mean Execution Time (MET) for multi-process and 272.4 μs for multi-thread. Even with this extremely simple pipeline, it still shows noticeable improvement regarding the worse case compared to semaphore based synchronization (gap < 25µs between MMET and MET). However, the histogram of both active

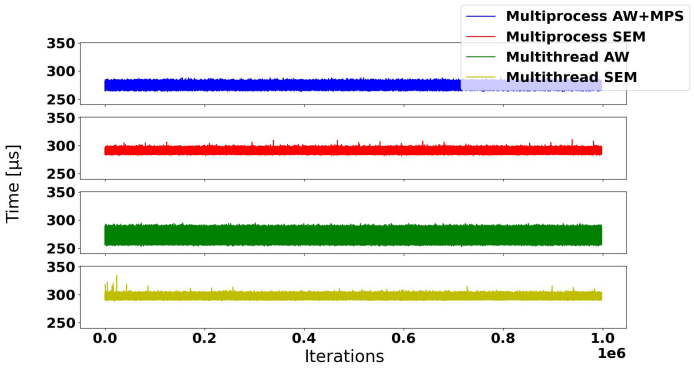

Figure 7 Execution profiles over 1M iterations for various scheduling and synchronization approaches on the DGX server

Compute type	Mean execution time (μs)	MMET (μs)	Average jitter (μs)	Jitter Peak-to-Valley (μs)
Multiprocess AW + MPS	274.3	288.8	3.5	25.6
Multiprocess SEM	290.0	310.8	2.3	29.2
Multithread AW	272.4	294.9	10.7	42.0
Multithread SEM	296.0	334.3	2.3	45.9

Table 3 Summarised results of different synchronization approaches on DGX server

wait synchronization approaches shows that results are scattered across two Gaussian distributions. That is why the average jitter of multi-thread active wait is considerably higher compared to the three other approaches. This behaviour is not yet explained as it doesn't look like an external interference, which would have been highlighted with semaphore approach as well. The multi-thread approach shows slightly higher Peak-To-Valley jitter (P2V – MMET minus the minimum measured time) with 42.0 μs with AW synchronization and 45.9 μs with semaphore synchronization. The multi-process approach (25.6 μs for AW and 29.2 μs) is slightly more stable.

With these considerations, the active-wait based synchronization is still outperforming semaphores in terms of latency and MMET and remains the preferred option with a dGPU. The multi-process approach shows slightly more stable results, although these gains are not sufficient to discard the multi-threaded approach.

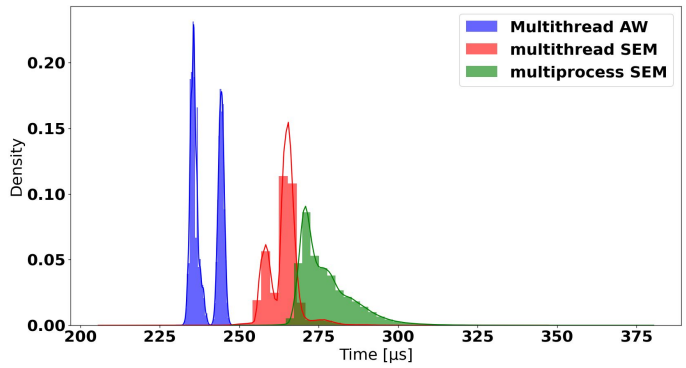

Figure 8 Normalized histograms with their estimated probability density function of a pipeline composed of one MVM unit with various synchronization approaches (SEMaphore and Active Wait) over 1M iterations on the Jetson Xavier

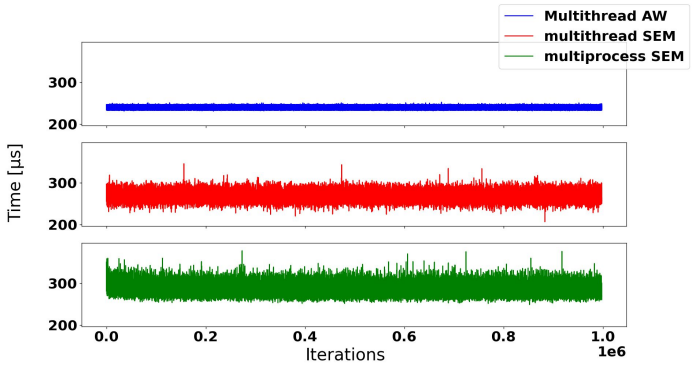

Figure 9 Execution profiles over 1M iterations for various scheduling and synchronization approaches on the Jetson Xavier

Compute type	Mean execution time (μs)	MMET (μs)	Average jitter (μs)	Jitter P2V (μs)
Multithread AW	239.4	252.5	4.4	21.5
Multithread SEM	263.9	346.4	4.8	139.8
Multiprocess SEM	277.4	378.8	8.4	129.7

Table 4 Summarised results of synchronization techniques on the Jetson Xavier

Experimental results on Jetson Xavier

As MPS is not available on Jetson Xavier, only 3 cases can be assessed. We performed the exact same experiment on this hardware and the results are presented in figures 8, 9 and Table 4.

When comparing results obtained on both the dGPU and iGPU platforms, a noticeable difference is that semaphore synchronization approaches lead to less stable results on the iGPU case with a up to 101.4μs gap between the MET and the MMET for multi-process and 82.5 for multi-threaded based approach. The multi-threaded active wait synchronization is performing best with only 13.1 μs difference. The P2V jitter shows the same kind of outcome with 21.5 μs for multi-thread AW, 139.8 μs for multi-thread SEM and 129.7 μs for multi-process SEM. Even though CPU cores are isolated on Jetson, semaphore synchronization is more prone to interference compared to the dGPU case. Multi-thread results are still considered satisfactory results, both in terms of MMET and average time to solution.

Regarding semaphore performance, it appears that multi-thread performs slightly better compared to multi-process. When comparing to the dGPU (Figure 6), the reverse occurs with multi-process semaphore delivering slightly better execution time. It may be caused by how CPU contexts are handled on both platforms (introducing CPU context switch) and probably some CPU interference, more pronounced on Jetson Xavier. We will investigate further such behavior in future work.

Multiprocess with hybrid active wait

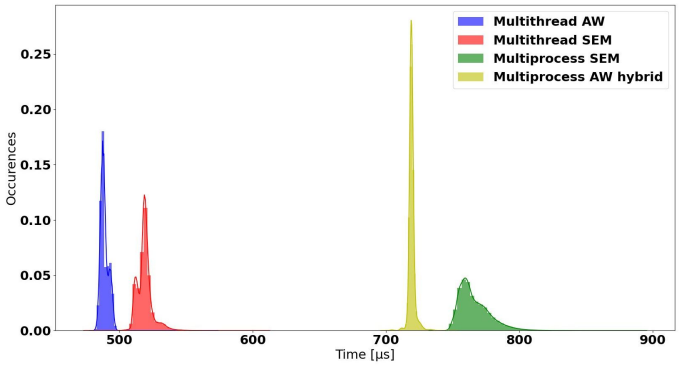

Figure 10 Normalized histograms with their estimated probability density function of a pipeline composed of two MVM units with the proposed hybrid (CPU/GPU) active wait compared to other synchronization approaches (Semaphore and active wait) on the Jetson Xavier

As aforementioned, the way the GPU handles multiple processes and CUDA contexts is not suitable for GPU busy waiting synchronization. A proposed workaround is to take advantage of shared memory to avoid busy waiting with GPU threads.

This hybrid active-wait strategy relies on letting the CPU do the busy waiting on data shared with the GPU while notifications are sent through GPU kernels. This allows the GPU scheduler to stop constantly switching between multiple CUDA contexts requiring GPU usage.

Although this implementation should not benefit from the latency hiding obtained with look ahead kernel launch and

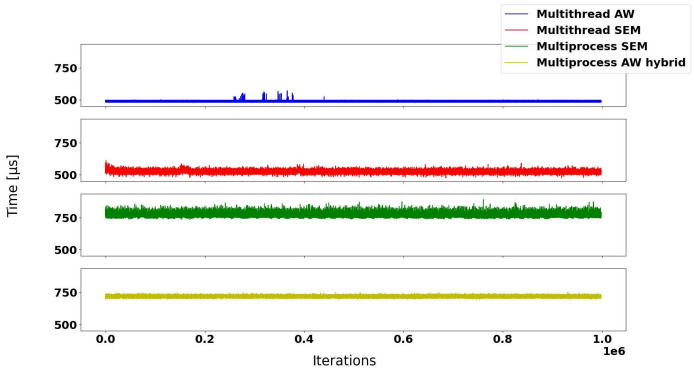

Figure 11 Execution profiles over 1M iterations of the hybrid active wait compared to other synchronization approaches on the Jetson Xavier

Compute type	Mean execution time (μs)	MMET (μs)	Average jitter (μs)	Jitter P2V (μs)
Multithread AW	488.9	572.4	3.0	93.2
Multithread SEM	519.0	611.1	5.8	116.9
Multiprocess SEM	766.16	893.3	11.6	158.9
Multiprocess AW hybrid	719.3	752.5	2.4	56.1

Table 5 Summarised results of hybrid synchronization techniques on Jetson

still partly suffers from context switching overheads, using the GPU for notification may help to reduce noxious interference coming from the CPU, plus avoiding some CPU/GPU synchronizations compared to semaphore approach.

It is important to note that such implementation is possible only with a sequential pipeline as concurrent kernels have to be implemented through CUDA streams, otherwise the resulting performance and determinism will be severely affected.

In this test case (Figure 9, 10 and Table 5), the new mechanisms described are implemented (results named Multiprocess AW hybrid), resulting in a new approach for multi-process synchronization. In order to expose how different kinds of synchronization affect performance, two sequential computing units are instantiated thus reproducing exactly the setting for Figure 4. The first noticeable thing is that the gap between multi-threaded and multi-processed synchronization increases as the pipeline complexity grows. Regarding latency, the multi-thread approach shows best results regardless the kind of synchronization (488.9 μs for active wait and 519.0 μs for semaphore) where the multi-process approach is taking an extra 200 μs for the exact same computations (766.16 μs for semaphore and 719.3 μs for the new hybrid synchronization). This brings to light the cost of GPU context switch which never happens in a multi-threaded application.

However, it also shows that synchronization on the GPU is an environment less prone to interference. The difference between the MMET and MET of the hybrid active wait approach is 33,2 μs where the second best being multi-thread AW with a difference of 83,5 μs. Finally, semaphore synchronization with multi-thread gives a difference of 92,1 μs and the multi-process approach comes last with a 127,14 μs difference. The P2V jitter confirms this result with 56.1 μs for the multi-process hybrid AW, 93.2 μs for multi-thread AW, 116.9 μs for multi-thread SEM and finally 158.9 μs for multi-process SEM.

As a consequence, the hybrid active waiting, although showing marginal latency improvement compared to using multi-process semaphore is still far from reaching multi-threaded approaches performance and cannot be proposed as a universal replacement for active wait with CUDA MPS. Still, it is a great replacement for pipelines that need to communicate through multi-process and is a good solution in terms of determinism.

Conclusion

The best practices introduced in the COSMIC framework enable the realization of complex compute intensive pipelines while keeping a high level of modularity thanks to its efficient synchronization mechanism.

It delivers latency improvements while keeping a stable jitter with discrete GPUs, but its use on embedded platforms needs to be adapted in order to work around missing features on such hardware and supporting ecosystem. Best results are obtained when reproducing the same mechanism with a multi-thread implementation, instead of multi-process. This allows the CUDA environment to keep a single context, avoiding both kernel preemption and context switching. However, the behaviour may differ between multiple processes and multiple streams implementations depending on how the scheduler handles kernels.

The embedded Jetson CUDA package is still missing some critical features of multi-process programming. However, it is possible to get it working using using the hybrid active wait strategy detailed in this paper. While the user needs to be aware of the corresponding performance trade-off, making sure only one context is executed at a given point in time, a pipeline can be built with processes sending notifications using the GPU by taking advantage of shared memory between CPU and GPU. Such approach performs better than synchronization through CPU semaphores both in terms of jitter and latency.

Future work will focus on further testing these new approaches to get a better understanding of embedded platforms behaviour in order to get closer to true GPU real-time determinism.

Acknowledgements

This work is sponsored through a grant from project 873120, a.k.a. Rising STARS, funded by European Commission under program H2020-EU.1.3.3 coordinated in H2020-MSCA-RISE-2019.

References

[1] F. Ferreira. Hard real-time core software of the AO RTC COSMIC platform: architecture and performance. *Proceedings of SPIE - The International Society for Optical Engineering*, 2020.

[2] NVIDIA. Cuda for tegra. https://docs.nvidia.com/cuda/cuda-for-tegra-appnote/, 2021.

[3] Julien Bernard. Design and performance of a scalable GPU-based AO RTC prototype. *Proceedings of SPIE - The International Society for Optical Engineering*, 2018.

[4] Allen Todd. Improving Real-Time Performance with CUDA Persistent Threads (CuPer) on the Jetson TX2. *Concurrent Real-Time White Paper*, 2018.

[5] Yang Ming. Avoiding Pitfalls when Using NVIDIA GPUs for Real-Time Tasks in Autonomous Systems. *ECRTS 2018*, 2018.

[6] Paweł Czarnul. Investigation of parallel data processing using hybrid high performance CPU+GPU systems and CUDA streams. *Computing and informatics*, 2020.

[7] Ali Waqar. Protecting Real-Time GPU Kernels on Integrated CPU-GPU SoC Platforms. *ECRTS 2018*, 2018.

[8] Kenjić Dušan. One Solution for Deterministic Scheduling on GPU for Automotive Algorithms. *2021 Zooming Innovation in Consumer Technologies Conference*, 2021.

[9] Jason Baietto. Real-Time Linux: The RedHawk Approach. *Concurrent Real-Time whitepaper*, 2019.

[10] Tanya Amert. Gpu scheduling on the NVIDIA TX2: hidden details revealed. *IEEE Real-Time Systems Symposium*, 2017.

[11] Ignacio Sanudo Olmedo. Dissecting the CUDA scheduling hierarchy: a Performance and Predictability Perspective. *RTAS*, 2020.

[12] Vance Miller. Determinism in GPU Programs, Real Time Applications on the NVIDIA Jetson TK1. *Senior Honors Thesis, University of North Carolina*, 2016.

[13] Nicola Capodieci. Contending memory in heterogeneous SoCs: Evolution in NVIDIA Tegra embedded platforms. *IEEE Real-Time Systems Symposium*, 2020.

[14] Nathan Otterness. Inferring Scheduling Policies of an Embedded CUDA GPU. *Department of Computer Science, University of North Carolina*, 2017.