Event-Based OpenMP Tasks for Time-Sensitive GPU-Accelerated Systems

Cyril Cetre[1,3(✉)], Chenle Yu[2,4], Sara Royuela[2], Rémi Barrere[1], Eduardo Quiñones[2], and Damien Gratadour[3]

[1] Thales Research & Technology, Palaiseau, France
cyril.cetre@thalesgroup.com, remi.barrere@thalesgroup.com
[2] Barcelona Supercomputing Center, Barcelona, Spain
{chenle.yu,sara.royuela,eduardo.quinones}@bsc.es
[3] LESIA, Observatoire de Paris, Université PSL, CNRS, Sorbonne Université, Université de Paris, 5 place Jules Janssen, 92195 Meudon, France
damien.gratadour@obspm.fr
[4] Universitat Politècnica de Catalunya, Barcelona, Spain

Abstract. The throughput-centric design of GPUs poses challenges when integrating them into time-sensitive applications. Nevertheless, modern GPU architectures and software have recently evolved, making it possible to minimize overheads and interference along the critical path through advanced mechanisms, such as GPU graphs, while sustaining high throughput. However, GPU vendors provide programming ecosystems specific to their products, raising concerns about code portability. Hence, there is a need for a hardware-agnostic API capable of managing time-sensitive GPU-accelerated pipelines. In this context, we propose integrating event-based synchronizations into the high-level OpenMP programming model to, in combination with GPU graphs, notably reduce interference and overheads over the critical path. This work showcases how this combination offers significant performance improvements and time consistency. We also enable portability across several vendor ecosystems and demonstrate our work on a set of representative applications for cyber-physical systems. According to our experiments, we measured a maximum jitter below 20 µs, representing less than 5% of time variation.

Keywords: Time-sensitive systems · GPU · High-performance computing · OpenMP · CUDA · ROCm

1 Introduction

Advanced Cyber-Physical Systems (CPS) are required to support increasingly complex specifications with ever tighter deadlines. Typically composed of sensors, actuators, data transfer devices, and computing hardware, their elaboration represents a significant challenge for many application domains in academia and industry. An illustrative CPS example is the astronomical Adaptive Optics (AO) system [14], in which the algorithms in charge of processing input data from sensors require significant computing power with minimal and consistent

© The Author(s) 2024
A. Espinosa et al. (Eds.): IWOMP 2024, LNCS 15195, pp. 31–45, 2024.
https://doi.org/10.1007/978-3-031-72567-8_3

response time. AO Real-Time Controllers (RTC) like the upcoming Extremely Large Telescope (ELT) are integrating GPUs to sustain the 1.7 TB/s of required single precision memory bandwidth with a 285 µs time budget [10].

However, the modular nature of AO RTCs necessitates a software solution that can accommodate flexibility with even more complex computing pipelines in heterogeneous environments. In addition, deployed solutions may remain used for decades and maintained by researchers who are not necessarily experts in heterogeneous programming. Therefore, there is a strong need for an API that allows for (1) ensuring reliability and consistent response time, (2) avoiding dependency on specific hardware configurations, and (3) minimum learning effort.

OpenMP, a high-level directive-based programming model supporting GPU acceleration, is an excellent candidate to address these needs. Its use allows developers to integrate accelerator-agnostic code, greatly enhancing productivity through portability and modularity. However, adopting such an approach usually leads to lower performance compared to native GPU APIs, and OpenMP also lacks several critical features to enable time-sensitive GPU computations within complex CPS. This paper addresses the previously mentioned through the following contributions:

1. An extension to the OpenMP specification of event-based synchronizations to support the release of host and target tasks based on events, while still fulfilling dependencies.
2. An extension of the OpenMP taskgraph to CUDA graph framework [24] to create CUDA graphs including the proposed event-based activation of tasks and so make graphs more suitable for real-time applications.
3. An extension to the same framework to generate HIP graphs for AMD GPUs and so increase the portability of the framework.
4. An evaluation of the proposed extension to OpenMP and its implementation that analyzes the response time variability in CUDA and ROCm ecosystems. The study uses complex pipelines coming from Adaptive Optics (AO) [11] and Adaptive Beamforming (ABF) [6].

2 Background

This section briefly presents the components of our framework and its intended applicability, i.e., the graph execution model, the current mechanisms in OpenMP for event-based synchronization, and the requirements of real-time CPS.

GPU Graphs. Introduced along with CUDA 10 in 2018, the CUDA graph features [18] allow for wrapping multiple interdependent kernels as a single execution unit, requiring only one submission onto a CUDA stream to launch all inner kernels. AMD released a similar feature, namely HIP Graph [1], in 2021. The graph programming paradigm reduces kernel submission overhead, a crucial aspect for short kernels. However, the intricacies of exposing parallelism with the stream-capturing method and the tedious and error-prone nature of the explicit graph API render the implementation of CUDA graphs complex, especially when the number of graph nodes (kernels) and edges (dependencies) increases.

Host Taskgraphs. Graph-based execution has shown its benefits in reducing offloading overheads, and this paradigm has also been introduced in modern and portable programming languages like Kokkos [23] and SYCL [17] to enable a record and replay mechanism that reduces the overhead of task orchestration. OpenMP is also working towards introducing this paradigm based on a proposal that augments the API with the `taskgraph` directive [25] to expose the region of code that can be represented as a Task Dependency Graph (TDG). Furthermore, the LLVM compilation framework already offers a prototype implementation for this feature [8] in its upstream main branch. Finally, further proposals improve the interoperability between OpenMP and CUDA by statically transforming the OpenMP TDG into a CUDA graph for optimized programmability (from the former) and performance (from the latter).

OpenMP Detachable Tasks. The OpenMP specification provides event-based execution through the `detach(event)` clause, which ties the completion of a task to the fulfillment of a given `event`, and the *omp_fulfill_event(event)* runtime routine, which implements the fulfillment of an event. These features are useful, for example, to allow the interoperability of OpenMP with other languages, like CUDA, HIP and MPI. In particular, a common use case involves an OpenMP task waiting for completion until a foreign library executes a specific callback function (e.g., data transmission completed), as shown in Listing 1.1[1].

Listing 1.1. OpenMP example of event-based task synchronization

```
1 void callback(hipStream_t stream, hipError_t status, void *data)
2 { omp_fulfill_event((omp_event_t *) data); }
3
4 void task_detach_example() {
5  omp_event_t *hip_event;
6  #pragma omp task detach(hip_event) // task A
7  {
8   do_something();
9   hipMemcpyAsync(dst, src, nbytes, hipMemcpyDeviceToHost, stream);
10  hipStreamAddCallback(stream, callback, hip_event, 0);
11 }
12 #pragma omp task // task B
13 do_something_else();
14 #pragma omp taskwait
15 }
```

CPS Specifications. Fig. 1 exemplifies the AO CPS used in this paper. This use case receives an image that triggers a timer and the execution of a matrix pre-processing computation, which in turn triggers a matrix-based computation when it completes. AO requires a memory bandwidth of at least 1700 GB/s, achievable only by a few top-end GPU accelerators. In addition, the pipeline must constantly adapt to changing operational conditions through repeated reconfigurations to maintain optimal performance. Given these specifications, to provide an agnostic solution to a wide range of similar time-sensitive CPS, this work considers the following constraints:

[1] OpenMP detachable task example extracted from https://indico.euro-fusion.org/event/688/attachments/854/2752/OpenMP_Webinar_5_2021-05-25.pdf.

- **Time sensitivity**: Target GPU-accelerated systems with critical time constraints and short deadlines.
- **Productivity**: Offer a straightforward approach to express intricate synchronization mechanisms with mainstream tools and concepts that minimize the development time and cost.
- **Maintainability**: Avoid dependency with specific hardware or ecosystem.
- **Modularity/flexibility**: Facilitate on-the-fly modifications on the pipeline, like incorporating new stages or responsive system halts.

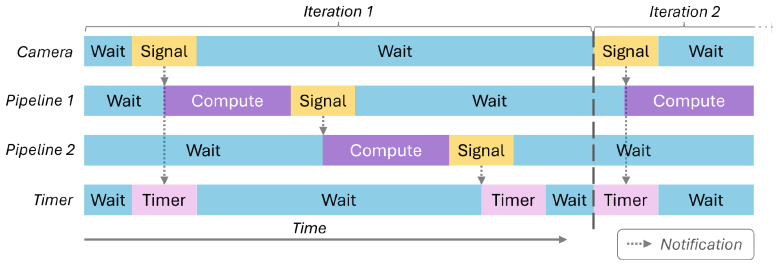


Fig. 1. Adaptive Optics (AO) pipeline.

3 Related Work

The CUDA ecosystem's closed nature complicates predicting and certifying the behavior of time-sensitive applications. Several studies work towards unveiling architecture and scheduler specificities to understand GPU behavior in discrete accelerators [16,19] and embedded platforms [2,4]. While these approaches pave the way to real-time GPU scheduling, more is needed to support them on critical systems. Examples include *persistent kernels* [13,22], which can reduce GPU latency at the cost of maintainability, and *busy-wait kernels*, first proposed by F. Ferreira et al. [12] and further explored by C. Cetre et al. [5], to mitigate the drawbacks above while maintaining latency specifications. Orthogonal to these efforts, our work aims to exploit the graph execution paradigm available on modern GPUs to further optimize the kernel submission and scheduling overhead.

Event-based synchronization among tasks has long been supported in safety-critical systems and embedded systems, for instance, by Ada language for the former and TinyGALS [9] for the latter. However, these solutions become insufficient when applied on High Performance Computing applications due to different system characteristics. Hence, there is a need to introduce a model, suitable for HPC systems, that includes event-based synchronization mechanism.

High level programming APIs, like OpenMP, provide an interesting bedrock to build such a mechanism because it already defines a task synchronization method, although limited, through events.

Accordingly, we plan to extend the existing event-based model for task launches. Furthermore, since the OpenMP's accelerator model improves productivity at the cost of performance [15], which is crucial for real-time applications

such as Adaptive Optics applications, we plan to couple event-based launches with the taskgraph framework [25], to deliver a programming model that allows to conveniently define GPU graphs synchronized with external events.

4 Event-Based Task Release with OpenMP

Contrary to the *allow-completion event* created by the `detach` clause that affects task completion, our work introduces a new mechanism that specifies the beginning of task execution. The proposition, along with its implementation overview, is detailed in the following of this section.

4.1 Proposed Extension to the OpenMP API

The new clause `attach(event)` attached to the `task` and `target` constructs describes a new restriction affecting task releases. More specifically, tasks defined with the `attach` clause will only begin the execution of their task region once their associated events are fulfilled. In order to fulfill such an event, we propose a new OpenMP directive, namely `fulfill_event`, including clauses to define host (`cpu_notify`) or device (`gpu_notify`) synchronization. Table 1 summarizes the proposed extension with brief descriptions.

Table 1. Extensions (in bold) proposed to the OpenMP API.

Directive	Associated clause	Description
`task`	**`attach(event)`**	Defer the execution of the task until the **`event`** is fulfilled
`target nowait`		Create a target task that will initiate its execution once the **`event`** is fulfilled
`fulfill_event`	**`cpu_notify(event)`**	Release **`event`** for host task waiting on it
	`gpu_notify(event)`	Release **`event`** for target task waiting on it

Syntax and Semantics. Listing 1.2 illustrates the proposal with an example that generates four interdependent target tasks, two of which are synchronized with events. The corresponding TDG is shown in Fig. 2. Upon the encountering of a `attach(event)` clause, the implementation creates a new *allow-launch event* and connects it to the beginning of the execution of the associated task region. Furthermore, the implementation updates the original *event-handle* to represent that *allow-launch event*. The generated task can only start executing its associated structured block when the *allow-launch event* is fulfilled. The *allow-launch event* is fulfilled when another thread encounters the `fulfill_event` directive with either the `cpu_notify` or `gpu_notify` clauses taking the *event-handle* corresponding to the *allow-launch event* as the argument. Once fulfilled, the *allow-launch event* will be destroyed and the *event-handle* in the argument will become disassociated from any event.

Listing 1.2. Event-based task release proposal.

```
1 // In a thread: tasks depending on events
2 int dep1, dep2, dep3, event1, event2;
3 void tdg_computation() {
4  #pragma omp target nowait  \
5      depend(out:dep1) attach(event1)
6  { /* task1 */ }
7  #pragma omp target nowait  \
8      depend(in:dep1) depend(out:dep2)
9  { /* task2 */ }
10 #pragma omp target nowait  \
11     depend(out:dep3) attach(event2)
12 { /* task3 */ }
13 #pragma omp target nowait  \
14     depend(in:dep2,dep3)
15 { /* task4 */ }
16 }
17 // In a separate thread: release events
18 void notify() {
19 #pragma fulfill_event gpu_notify(event1)
20 #pragma fulfill_event gpu_notify(event2)
21 }
```

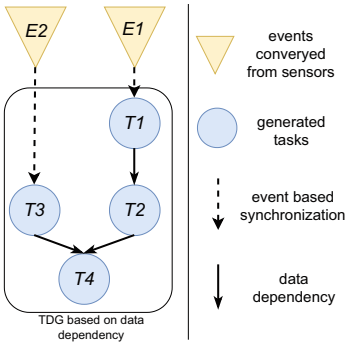


Fig. 2. TDG with events.

The Choice of a Directive Instead of a Routine. The existing *allow-completion event* is fulfilled through calling `omp_fulfill_event`, an OpenMP runtime routine. Similarly, we could define a new routine function that fulfills an *allow-launch event*. However, the possible lowering opportunities of the graph of execution (through the OpenMP taskgraph in our case) are hindered when using runtime routines, as these have a fixed implementation in each runtime system. Instead, a directive enables implementing lowering options depending on the targeted system. The lowering phase of this work is described in Sect. 4.3, which details the interaction of the `fulfill_event` construct and the `gpu_notify` and `cpu_notify` clauses with the taskgraph framework.

Current Limitations. In the current implementation, to satisfy the tight timing requirements of real-time applications, threads executing a task whose *allow-launch event* is not fulfilled are set to spin actively in an infinite loop while waiting (i.e., busy wait). However, this behavior can be modified to satisfy other requirements. For instance, a blocking wait mechanism could be used to prevent these threads from consuming CPU cycles when the implementation does not target hard real-time use cases.

4.2 Interactions with the Taskgraph Framework

The new event allows OpenMP tasks to be conveniently synchronized with external sensors, such as a camera. However, due to the restrictive timing requirements of real-time CPS, it is needed to minimize the runtime overhead, primarily related to task management, as it typically represents the main overhead in task programs [20,21]. The taskgraph framework [25] is a compelling method to mitigate such cost as it does not incur most task orchestrating operations when replaying.

Event with Taskgraph in OpenMP Runtime. Once the taskgraph is built for a region, replaying such taskgraph omits the user code within the corresponding region. Consequently, the `attach(event)` clause is not executed, and new events will not be created. As a solution, we record the tasks with their associated events, so that the fulfillment of such events still releases the corresponding tasks.

Event with Taskgraph in Foreign Runtime. GPU-accelerated real-time applications often have their device kernels tailored to the underlying GPU. The kernels automatically generated by the LLVM compiler may not be sufficiently optimized to fulfill the real-time requirements. Therefore, we leverage the device taskgraph approach proposed by C. Yu et al. [24], which enables to generate CUDA graphs through OpenMP directives using existing CUDA kernels. The device taskgraph framework creates kernel nodes for `target nowait` constructs and CUDA host nodes for `task` directives. By integrating `allow-launch events` as new CUDA graph nodes into the graph, we can further reduce the synchronization cost. Consequently, in this work, we solely focus on using the new event with taskgraph in foreign runtime.

4.3 Implementation in LLVM

Based on LLVM 15, our framework implementation in the compiler includes front-end modifications, middle-end IR transformations, and runtime support. Following, we give an overview of the implementation in the OpenMP context and then in a foreign context interacting with another runtime, such as CUDA.

Implementation in the OpenMP Context. The compiler front-end emits a runtime function call, namely `__kmpc_task_allow_launch_event`, when encountering a task with the `attach(event)` clause. The runtime function associates the created task with the event. As a result, when the task has its input data dependencies satisfied, it also evaluates whether its associated event is fulfilled before executing. For the `fulfill_event` directive, the `cpu_notify` is used to notify the OpenMP runtime while the `gpu_notify` clause is effective in foreign contexts, i.e., in GPU graphs. Hence, when the taskgraph is managed by the OpenMP runtime, host and target tasks are all orchestrated from the host library. Consequently, the fulfillment of an `allow-launch event` through the runtime function `__kmpc_fulfill_launch_event` can synchronize both host and target tasks.

Implementation in Foreign Contexts. To generate a CUDA graph with event synchronization, we are based on the implementation of taskgraphs proposed by C. Yu et al. [24]. Figure 3 shows an overview of the complete compilation process, including our alterations. This CUDA graph generation can be summarized to:

Step1. An analysis pass on the LLVM IR of the OpenMP program determines the task instances and their corresponding arguments through inspecting the task constructing runtime calls (e.g., *__kmpc_omp_target_task_alloc*).

Step2.1. A transformation pass takes the analysis result and transforms, or inserts calls to the CUDA runtime in the LLVM IR file at the right places, e.g., replacing host-to-device copies with *cudaMemcpy* calls. The figure represents the output file as *augmented LLVM file*.

Step2.2. Along with the transformation pass, a CUDA file is generated from scratch, noted in the figure as *genGraph.cu*. Based on the information gathered in the analysis pass, the file builds the CUDA graph by calling CUDA graph API functions (e.g., *cudaGraphAddKernelNode*).

Step3. Link the object files compiled from the *augmented LLVM file* and the *genGraph.cu* files to produce the final binary.

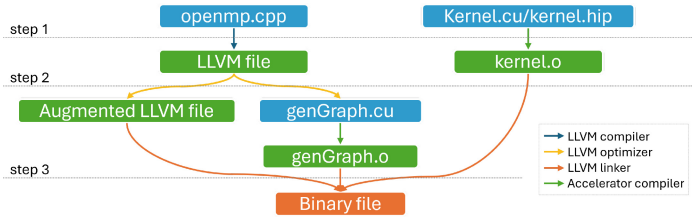


Fig. 3. Compilation steps of the GPU graph generation.

To incorporate event-based synchronization into the *genGraph.cu* file, the following additional transformations are performed:

1. Upon encountering the `attach` clause for the first time, two kernels exclusively used for synchronization are created and inserted in the generated CUDA file. One represents the active wait on the associated event (referred to as *BusyWaitKernel*), and the other is to notify the fulfillment of it (referred to as *FulfillingKernel*).
2. For each *attach* clause, two graph nodes are created: one containing the encapsulated task (referred to as *TaskNode*), and the other with *BusyWaitKernel* (referred to as *BusyWaitNode*).
3. All input dependencies of *TaskNode* are transferred to the *BusyWaitNode*.
4. A dependency is created from *BusyWaitNode* to the *TaskNode*, ensuring that the synchronization kernel finishes before the encapsulated task can start.

Additionally, the `fulfill_event` and the `gpu_notify` clause remain independent from the taskgraph, producing the following on the LLVM IR:

1. Create or retrieve a stream for the GPU notification (it must be a different stream from the one launching the GPU graph).
2. Insert the *FulfillingKernel* to such a stream so that the *BusyWaitNode* can be released.

4.4 Generation of HIP Graph

This work extends LLVM to support HIP graphs for AMD GPUs from OpenMP to support a broader range of GPUs. Targeting AMD devices can be done effortlessly by the user, who just provides an additional flag at the compilation time, namely *-hip-tdg*, to carry the following modifications to generate a HIP graph:

1. The *augmented LLVM file* in Fig. 3 is modified accordingly so that the inserted CUDA calls (data allocation and transfers from host to device and conversely) are transformed to HIP API calls.
2. The *genGraph.cu* file is translated to HIP C++ through the *hipify* script.
3. Similarly, the final binary is created by linking the object files compiled from kernel code, HIP Graph C++ file, and the *augmented LLVM file*.

Regarding step 2, it is possible to enhance the LLVM framework to generate HIP C++ directly, without translating it from CUDA. This approach could lead to a more robust implementation for AMD devices and will be addressed in future work.

5 Evaluating the Proposed OpenMP Extension

Our experiments measure the response time and time variability of real-time CPSs. The response time includes all possible interrupts, overheads, and external events. For each application tested, we evaluate three implementations: (1) the *Regular CUDA/HIP Graph*, which relies on CUDA/HIP graphs with active wait kernels for synchronizing computations and serves as the reference implementation because, although it requires more effort from the user, it typically provides the best performance; (2) *OpenMP CUDA/HIP Graph with GPU Sync.*, which uses GPU graphs generated from the proposed taskgraph lowering to CUDA graphs, including the new event synchronization; and (3) *OpenMP CUDA/HIP Graph with CPU Sync.*, which uses the taskgraph framework to generate a GPU graph but leaves the synchronizations to the CPU processes, i.e., the `fulfill_event gpu_notify` directive makes a CPU process to submit the *FulfillingKernel* to the GPU instead of translating it to a graph node.

The metrics used in the experiments are: (i) the *Mean Execution Time* (MET), (ii) the *Maximum Measured Execution Time* (MMET), and (iii) the *Max Jitter* (i.e., $MMET - MET$). The first two evaluate the response time and the last assesses the time variability. The measurements are obtained on a server with the configuration shown in Table 2. The applications are launched on isolated CPU threads through the *isolcpus* kernel boot command line.

Table 2. Environmental setup.

OS	Ubuntu 22.04 (w/o real-time patch)
Linux kernel	5.15.0-75
CPU	Intel Xeon 5220
CUDA ver.	12.0
ROCm ver.	5.6.0
NVIDIA GPU	A100 80GB
AMD GPU	MI100

Two use cases are selected:

An Adaptive Optics (AO) real-time controller, which reproduces all the steps from data acquisition to the output array [11]. As described in Fig. 1, this application combines two computational Pipeline Units (PU): a pixel processing unit that takes as input the raw data from wavefront sensor cameras and processes the pixels with a series of simple arithmetic kernels before sending the processed pixels to the second PU. The latter PU performs linear algebra operations, particularly matrix-vector multiplications, to produce a command as a vector sent to the deformable mirror actuators associated with a physical component, typically a telescope.

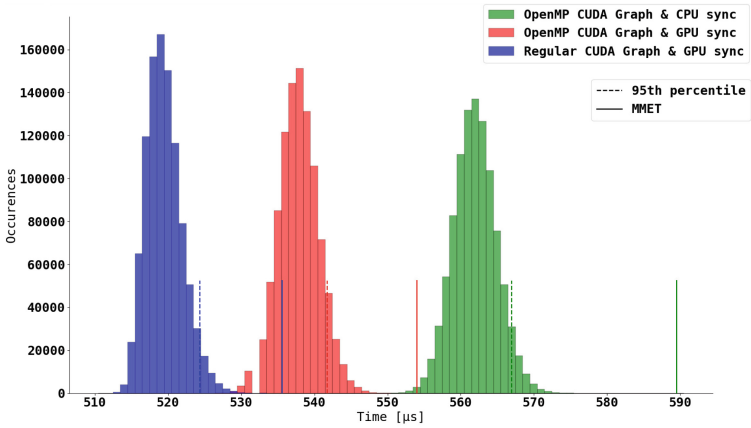
An Adaptive Beam Forming (ABF) algorithm, used in various fields, including wireless communications, radar/sonar systems, and medical imaging. It can leverage the throughput provided by the GPU to improve the quality of the received signal [3,6,7]. ABF uses weights and phases to combine signals from multiple sensors and build spatial beams by lowering signals from other directions. Building beams require high bandwidth data acquisition, low latency, and, lastly, high computing power to form and produce signals that would be unusable without ABF. The implementation used in our experimentation has one PU, which takes incoming signals as input and outputs the reconstructed signal through a combination of several arithmetic and linear algebra kernels.

These applications are typically measured in microseconds; therefore, we include a timer PU within the pipeline to obtain an accurate yet non-intrusive time measurement. This timer PU submits CUDA/HIP events to the computation stream to set an iteration's start and end points. The numbers reported in this section result from executing one million iterations.

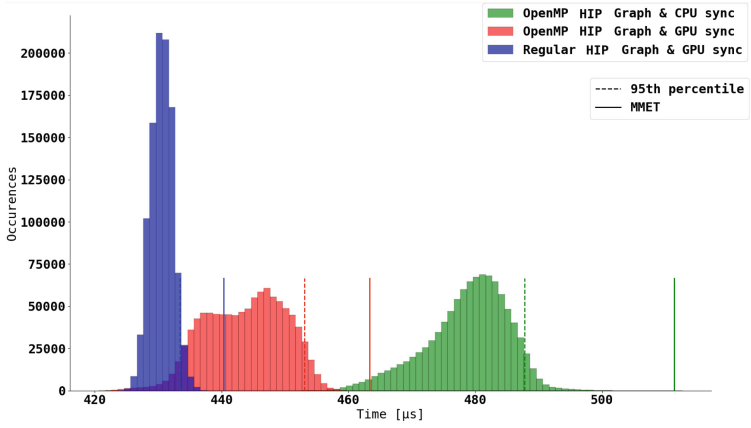
5.1 Adaptive Optics Application

Figure 4 shows the results obtained with the AO application using CUDA (in Fig. 4a) and HIP (in Fig. 4b). The reference implementation with vanilla CUDA/HIP graphs gives the best results with a maximum jitter of 16 µs (520 µs MET and 536 µs MMET) with CUDA and 11 µs (430 µs MET, 441 µs MMET) with HIP. Our method, i.e., OpenMP GPU graph with event synchronization, achieves an identical 16 µs max. jitter (538 µs MET and 554 µs MMET) with CUDA, and slightly higher max. jitter of 21 µs (443 µs MET and 464 µs MMET) with HIP, both increasing around 15 µs the average execution time. Finally, the third method using the CPU synchronization strategy showed the largest execution times and a max. jitter of 28 µs (562 µs MET and 590 µs MMET) with CUDA and 33 µs (478 µs MET and 511 µs MMET) with HIP.

The CPU synchronization strategy showed the most significant response time and lower stability. Although the maximum jitter of the third implementation is the largest, it can still be considered by the target specification as the maximum jitter is smaller than 10% of MET. However, this high jitter shows that even if the involved CPU threads are thoroughly isolated, such an approach is more



(a) Using CUDA graphs.



(b) Using HIP graphs.

Fig. 4. Occurrences of different response time of the AO application.

prone to be impacted by interrupts. Over the long term, this method is more likely to miss the application deadline, which makes it more challenging to meet our target specifications.

Our approach with OpenMP GPU Graphs delivered competitive mean execution times and jitters compared to the native CUDA/HIP implementation while maintaining a simple, directive-based programming style. This method benefits from event synchronization and GPU graphs, combining time sensitivity with productivity and maintainability. Furthermore, real-life AO and ABF applications accommodate dozens of PUs, making writing manually CUDA/HIP graphs challenging and invalidating the implementation with CPU synchronization as the jitter scales with the number of synchronizations.

5.2 Adaptive Beam Forming Application

Tables 3a and 3b present the results obtained with the ABF application. Similar to the AO application, the native CUDA/HIP implementations showed the best performance with the shortest MET. The proposed method, OpenMP CUDA/HIP graph with event synchronization, only introduces an overhead of about 10 µs compared to the native implementation (a MET of 520 µs vs. 515 µs for CUDA, and 551 µs vs. 540 µs for HIP). Whereas with the CPU synchronization, this overhead is 18 µs for CUDA and 39 µs on AMD GPUs. Again, real-life ABF applications contain multiple computing PUs, making our solution more valuable, as it provides a convenient way to define an efficient and consistent execution pattern.

Table 3. Metrics for the ABF application.

Metrics (µs)	MET	MMET	Max jitter
CUDA busy-wait	515	523	8
OpenMP CUDA graph with GPU Sync.	520	525	5
OpenMP CUDA graph with CPU Sync.	533	550	17

(a) Using CUDA graphs.

Metrics (µs)	MET	MMET	Max jitter
HIP busy-wait	540	555	15
OpenMP HIP Graph with GPU Sync.	551	564	13
OpenMP HIP Graph with CPU Sync.	579	598	19

(b) Using HIP graphs

6 Conclusion

This paper extends the OpenMP support for event-based synchronization with an *allow-launch event* through a new directive and a series of clauses detailed in Table 1. Combining with an enhancement of an existing taskgraph implementation [24,25], we create a framework that is capable of statically generating CUDA and HIP graphs containing synchronizing kernels out of an OpenMP code. These extensions enable the use of OpenMP in domains nowadays requiring HPC capabilities and real-time behaviour, from the presented adaptive optics and adaptive beamforming applications, to other domains like autonomous mobility, including automotive, railway and avionics, among others.

Although our experiments show that such graphs are particularly suitable for real-time applications, the current framework cannot be scaled dynamically due to the intrinsic, static nature of the taskgraph framework. In the future, we plan to adopt a new proposal for dynamically recording GPU graphs [26] to satisfy on-the-fly modifications required by some CPSs. Additionally, for the CUDA ecosystem, the introduction of device-launched graphs in CUDA 12 and conditional nodes in CUDA 12.3 could further enhance the flexibility of our work.

Acknowledgements. This work is supported by the RisingStars project, under the Marie Skłodowska-Curie grant agreement No 873120 from the European Union's Horizon 2020 research and innovation programme. This work is co-financed by the ASCENDER project of the UNICO I+D Cloud program that has MINECO and the EU-Next Generation EU as financing entities, within the framework of the PRTR and the MRR. This work is also partly supported by the HiPERT project, with reference PID2023-148117NA-I00, financed by MCIU/AEI/10.13039/501100011033/ FEDER, UE.

References

1. AMD: HIP documentation (2024). https://rocm.docs.amd.com/projects/HIP/en/latest/
2. Amert, T., Otterness, N., Yang, M., Anderson, J.H., Smith, F.D.: GPU scheduling on the NVIDIA TX2: hidden details revealed. In: 2017 IEEE Real-Time Systems Symposium (RTSS), pp. 104–115 (2017). https://doi.org/10.1109/RTSS.2017.00017
3. Barrere, R., Lenormand, E., Bui, D., Lee, E.A., Shaver, C., Tripakis, S.: An introduction to the pthales domain of Ptolemy II. Technical report UCB/EECS-2011-32, EECS Department, University of California, Berkeley (2011). http://www2.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-32.html
4. Capodieci, N., Cavicchioli, R., Olmedo, I.S., Solieri, M., Bertogna, M.: Contending memory in heterogeneous SoCs: Evolution in NVIDIA tegra embedded platforms. In: 2020 IEEE 26th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), pp. 1–10 (2020). https://doi.org/10.1109/RTCSA50079.2020.9203722
5. Cetre, C., Ferreira, F., Sevin, A., Barrere, R., Gratadour, D.: Real-time high performance computing using a Jetson Xavier AGX. In: 11th European Congress Embedded Real Time System (ERTS2022). Toulouse, France (2022). https://hal.science/hal-03693764
6. Chen, J., Chen, J., Min, H., Wang, X.: Real-time embedded implementation of adaptive beamforming for medical ultrasound imaging. In: 2016 Sixth International Conference on Instrumentation & Measurement, Computer, Communication and Control (IMCCC), pp. 356–360 (2016). https://doi.org/10.1109/IMCCC.2016.66
7. Chen, J., Yu, A.C.H., So, H.K.H.: Design considerations of real-time adaptive beamformer for medical ultrasound research using FPGA and GPU. In: 2012 International Conference on Field-Programmable Technology, pp. 198–205 (2012). https://doi.org/10.1109/FPT.2012.6412134
8. Chenle Yu, A.M.: Task record and replay mechanism in LLVM (2023). https://reviews.llvm.org/D146642
9. Cheong, E., Liebman, J., Liu, J., Zhao, F.: TinyGALS: a programming model for event-driven embedded systems. In: Proceedings of the 2003 ACM Symposium on Applied Computing, pp. 698–704. SAC 2003, Association for Computing Machinery, New York, NY, USA (2003). https://doi.org/10.1145/952532.952668
10. Clénet, Y., et al.: MICADO-MAORY SCAO Preliminary design, development plan & calibration strategies. In: Adaptive Optics for Extremely Large Telescopes conference, 6th edn. Québec, Canada (2020). https://hal.science/hal-03078430
11. Ferreira, F., Bernard, J., Sevin, A., Doucet, N., Gratadour, D.: Cosmic: a real-time platform for signal processing pipelines. In: 2022 IEEE Workshop on Signal Processing Systems (SiPS), pp. 1–6 (2022). https://doi.org/10.1109/SiPS55645.2022.9919251

12. Ferreira, F., et al.: Hard real-time core software of the AO RTC COSMIC platform: architecture and performance, p. 172 (2020). https://doi.org/10.1117/12.2561244
13. Gupta, K., Stuart, J.A., Owens, J.D.: A study of persistent threads style GPU programming for GPGPU workloads. In: 2012 Innovative Parallel Computing (InPar), pp. 1–14 (2012). https://doi.org/10.1109/InPar.2012.6339596
14. Guyon, O.: Extreme adaptive optics. Ann. Rev. Astron. Astrophys. **56**, 315–355 (2018)
15. Khalilov, M., Timoveev, A.: Performance analysis of CUDA, OpenACC and OpenMP programming models on TESLA V100 GPU. In: Journal of Physics: Conference Series, vol. 1740, p. 012056. IOP Publishing (2021)
16. Li, H., Yu, D., Kumar, A., Tu, Y.C.: Performance modeling in CUDA streams - a means for high-throughput data processing. In: 2014 IEEE International Conference on Big Data (Big Data), pp. 301–310 (2014). https://doi.org/10.1109/BigData.2014.7004245
17. Ltd, C.S.: SYCL Graphs (2024). https://codeplay.com/portal/blogs/2024/01/22/sycl-graphs
18. NVIDIA: CUDA 10 Features Revealed: Turing, CUDA Graphs, and More (2018). https://developer.nvidia.com/blog/cuda-10-features-revealed/
19. Olmedo, I.S., Capodieci, N., Martinez, J.L., Marongiu, A., Bertogna, M.: Dissecting the CUDA scheduling hierarchy: a performance and predictability perspective. In: 2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), pp. 213–225 (2020). https://doi.org/10.1109/RTAS48715.2020.000-5
20. Podobas, A., Karlsson, S.: Towards unifying OpenMP under the task-parallel paradigm. In: Maruyama, N., de Supinski, B.R., Wahib, M. (eds.) IWOMP 2016. LNCS, vol. 9903, pp. 116–129. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45550-1_9
21. Rico, A., Sánchez Barrera, I., Joao, J.A., Randall, J., Casas, M., Moretó, M.: On the benefits of tasking with OpenMP. In: Fan, X., de Supinski, B.R., Sinnen, O., Giacaman, N. (eds.) IWOMP 2019. LNCS, vol. 11718, pp. 217–230. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-28596-8_15
22. Todd, A.: Improving real-time performance with CUDA persistent threads (CuPer) on the Jetson TX2. Concurrent Real-Time White Paper (2018)
23. Trott, C.R., et al.: Kokkos 3: Programming model extensions for the exascale era. IEEE Trans. Parallel Distrib. Syst. **33**(4), 805–817 (2021)
24. Yu, C., Royuela, S., Quiñones, E.: OpenMP to CUDA graphs: a compiler-based transformation to enhance the programmability of NVIDIA devices. In: Proceedings of the 23th International Workshop on Software and Compilers for Embedded Systems, p. 42-47. SCOPES 2020, Association for Computing Machinery (2020)
25. Yu, C., Royuela, S., Quiñones, E.: Taskgraph: a low contention OpenMP tasking framework. IEEE Transactions on Parallel and Distributed Systems (2023)
26. Yu, C., Royuela, S., Quiñones, E.: Enhancing heterogeneous computing through OpenMP and GPU graph. In: 53rd International Conference on Parallel Processing (2024). https://doi.org/10.1145/3673038.3673050

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.