Analysis of Parallel Programming Models

Doc. No.: RS-WP2-D2.1

Issue: 6.0

Date: 14.12.2021

Author: Eduardo Quiñones (BSC)

Contributors: Cyril Cetre (TRT), Remi Barrere (TRT), Damien Gratadour (OdP)

Rising STARS Consortium

Horizon 2020 R&I programme under the Marie Skłodowska-Curie grant agreement No 873120

CHANGE RECORD

ISSUE	DATE	SECTION/PARAGRAPH AFFECTED	REASON/INITIATION DOCUMENTS/REMARKS
0.1	27.09.2022	All	New document (first Draft)
0.2	28.12.2022	All	Document completed by BSC
0.3	17.01.2023	All	Document reviewed by OdP/TRT
0.4	23.01.2023	All	Document updated by BSC
1.0	24.01.2023	All	Document ready to be submitted

Table of Contents

1 Introduction 4
1.1 Scope 4
1.2 Definitions, Acronyms and Abbreviations 4
2 Related documents 6
2.1 Applicable documents 6
2.2 Reference documents 6
3 Parallel Programming Models 7
3.1 CUDA 7
3.2 OpenCL 8
3.3 OpenMP 9
3.4 Analysis of the Programming models productivity 10
4 OpenMP Support for functional and non-functional requirements 13
4.1 Functional safety and correctness 13
4.2 Time predictability 13
4.3 Energy 14
5 Suitability of OpenMP on RisingSTARS Adaptive Optics Real-Time Controller use-cases 15
6 Conclusions 18
7 Bibliography 19

Horizon 2020 R&I programme under the Marie Skłodowska-Curie grant agreement No 873120

1 Introduction

1.1 Scope

The scope of this document is to present the first analysis of the parallel programming models taken into account within the context of Rising STARS research activities.

This document contains the following information:

- Section 3 provides a description and comparison of the parallel programming models OpenMP, CUDA and OpenCL.
- Section 4 provides an analysis on the impact of functional safety, time predictability and energy efficient on the OpenMP programming model.
- Section 5 provides an analysis of the type of applications considered within the project and the suitability of the selected parallel programming models.

1.2 Definitions, Acronyms and Abbreviations

This document employs several abbreviations and acronyms to refer concisely to an item, after it has been introduced. The following list is aimed to help the reader in recalling the extended meaning of each short expression:

AO	Adaptive Optics
API	Application Programming Interface
DDS	Data Distribution Service
DMA	Direct Memory Access
ELT	European Extremely Large Telescope
ESO	European Southern Observatory
GPU	Graphical Processing Unit
IF	Interface
IPC	Inter-Process Communication
N/A	Not Applicable
RTC	Real-Time Controller
SCAO	single-conjugate adaptive optics
SHM	Shared Memory
SE	System Engineering
TBC	To Be Confirmed
TBD	To Be Defined
TRL	Technology Readiness Level
WP	Work Package
FPGA	Field-Programmable Gate Array

Horizon 2020 R&I programme under the Marie Skłodowska-Curie grant agreement No 873120

OpenMP	Open Multi-Processing
OpenCL	Open Computing Language
CUDA	Compute Unified Device Architecture
OpenACC	Open Accelerators
OpenGL	Open Graphics Library
DSP	Digital Signal Processing
TDG/DAG	Task Dependency Graph/Direct Acyclic Graph
HRTC	Hard Real-Time Controller
SRTC	Soft real-time Controller

2 Related documents

2.1 Applicable documents

The following Applicable Documents (AD) of the exact issue contain provisions (statements, instructions, recommendations or requirements) that are ruling over - and implicate provisions in the present document.

AD	Doc Nr	Doc Title	Issue	Date
[AD1]	RS-DoA-V1	Rising STARS Description of Action v1	1.0	14.12.2021

2.2 Reference documents

The following Reference Documents (RD) contains useful information relevant to the subject of the present document.

RD	Doc Nr	Doc Title	Issue	Date
[RD1]			1.0	

Horizon 2020 R&I programme under the Marie Skłodowska-Curie grant agreement No 873120

3 Parallel Programming Models

This section provides an analysis related to the execution and memory models of the parallel programming models supported by the processor architectures considered within the Rising STARS project, featuring two different kinds of acceleration devices: i.e., GPU-based acceleration provided by NVIDIA, and FPGA-based acceleration provided by Altera and Xillinx.

3.1 CUDA

CUDA [1] is a parallel programming model designed to naturally map the parallelism within an application to the massive parallelism of the stream multiprocessors (SMs) implemented in NVIDIA devices. The CUDA platform is accessible through CUDA-accelerated libraries, compiler directives (e.g., OpenACC), and extensions to industry-standard programming languages (e.g., C, C++). Additionally, interfaces including OpenCL and OpenGL are also supported.

A CUDA program is a serial program that calls parallel *kernels*, i.e., functions or full programs. Each kernel executes across a set of parallel threads organized in a hierarchy of grids of thread blocks. A *thread block* is a set of concurrent threads that can cooperate through synchronization and shared access to a memory space private to the block. A *grid* is a set of threads block that may execute in parallel with other grids. Parallelism is determined explicitly by specifying the dimensions of a grid and its thread blocks when launching a kernel.

Parallel execution and thread management are automatic. All thread creation, scheduling and termination are handled by the underlying system, mostly directly in hardware. Per block thread synchronization is accomplished calling the *__syncthreads()* intrinsic.

Threads may access data from multiple memory spaces: per-thread *local* memory, per-block *shared* memory and *global* memory. Global memory is managed via the *cudaMalloc()* and *cudaFree()* runtime calls. To support a heterogeneous system architecture combining a CPU and a GPU, each with its own memory system, CUDA programs must copy data between the *host memory* and the *device memory*. Unified memory is a component of CUDA that provides *managed memory* to bridge the host and device memory spaces by defining a memory space in which all processors see a single coherent memory image with a common address space.

```
/* kernel.cu */
__global__ void matrix_mul(float A[N][N], float B[N][N], float C[N][N], int wA, int wB)
{
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    float th_value = 0;
    for (int k = 0; k < wA; ++k)
       th_value += A[row * wA + k] * B[k * wB + col];
    C[row * wA + col] = th_value;
}
/* main.c */
int main() {
    …
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N/threadsPerBlock.x, N/threadsPerBlock.y);
    matrix_mul<<numBlocks, threadsPerBlock>>(A, B, C, wA, wB);
    …
}
```

Listing 1. CUDA matrix multiplication example.

Listing 1 shows a simple example of a CUDA parallel program. There, the kernel matrix_mul is specified by means of the *__global__* specifier. The *threadIdx*, *blockIdx* and *blockDim* are built-in variables that allow accessing each thread, block and block dimension respectively.

The concurrency among kernels is managed using CUDA streams. A stream is a sequence of operations that execute in issue-order on the GPU. Using several streams allows the concurrent and synchronous or asynchronous execution of kernels.

Newer versions of CUDA include a new paradigm, CUDA graphs, offering two main characteristics: (1) it allows expressing work as graphs rather than single operations, and (2) it enables a *define-once-run-repeatedly* execution flow. A graph consists of a series of operations, such as memory copies and kernel launches, connected by dependencies and defined separately from its execution.

3.2 OpenCL

OpenCL [2] is an open standard for writing programs that execute across heterogeneous platforms including CPUs, GPUs, DSPs, FPGAs and other accelerators. Naturally, OpenCL pursues portability while considering programmability.

The OpenCL architecture consists of one *host* (CPUD-based) that controls multiple *compute devices* (CPUs and GPUs). Each of these consists of multiple *compute units* (equivalent to stream multiprocessors in NVIDIA, and stream cores or SIMD engines in AMD) and the latter contain multiple *processing elements*, each of them executing OpenCL *kernels*. So, the kernel is the basic unit of parallelism. Kernel bodies are instantiated once per *work item* (equivalent to a *CUDA thread*), and each work item gets a unique global id. Work-items are wrapped in *work-groups* (equivalent to a *CUDA thread block*).

OpenCL offers fine-grained data- and thread-parallelism (at the work-item level) nested within coarse-grained data- and task-parallelism (at the work-groups level). Synchronization in the form of memory fences is possible within threads in a work-group, as well as synchronization barriers for threads at the work-item level. Additionally, the host can use blocking API operations to wait for completion of certain events.

Listing 2 shows a simple example of a OpenCL parallel program. There, the kernel matrix_mul is specified by means of the *__global__* specifier. The *threadIdx*, *blockIdx* and *blockDim* are built-in variables that allow accessing each thread, block and block dimension respectively.

OpenCL has an advantage over CUDA, and is that it can be executed, not only in any GPU including the library, but also in the host. Hence, schedulers could decide to execute a given task with a unique OpenCL implementation in the host based on the availability of the resources or the performance expected for the given device [3].

```
/* kernel.cl */
__kernel void matrix_mul(__global float A*, __global float B*, __global float C*, int
wA, int wB)
{
   int col = get_global_id(0);
   int row = get_global_id(1);
   float th_value = 0;
   for (int k = 0; k < wA; ++k)
      th_value += A[row * wA + k] * B[k * wB + col];
   C[row * wA + col] = th_value;
}
/* main.c */
int main() {
    …
    clGetDeviceIDs(platform_ids[0],
       gpu ? CL_DEVICE_TYPE_GPU : CL_DEVICE_TYPE_CPU,
       1, &device_id, NULL);
    context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
    commands = clCreateCommandQueue(context, device_id, 0, &err);
    lFileSize = LoadOpenCLKernel("kernel.cl",
       &KernelSource, false);
    program = clCreateProgramWithSource(context, 1,
       (const char **) & KernelSource, NULL, &err);
    err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
    kernel = clCreateKernel(program, "matrix_mul", &err);
    …
    err = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&d_C);
    err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&d_A);
    err |= clSetKernelArg(kernel, 2, sizeof(cl_mem), (void *)&d_B);
    err |= clSetKernelArg(kernel, 3, sizeof(int), (void *)&wA);
    err |= clSetKernelArg(kernel, 4, sizeof(int), (void *)&wB);
    err = clEnqueueNDRangeKernel(commands, kernel, 2,
        NULL, globalWorkSize, localWorkSize, 0, NULL, NULL);
    err = clEnqueueReadBuffer(commands, d_C, CL_TRUE, 0,
        mem_size_C, h_C, 0, NULL, NULL);
}
```

Listing 2.OpenCL matrix multiplication example.

3.3 OpenMP

OpenMP [4] is an API aiming at facilitating parallel programming in shared-memory systems first, and also in heterogeneous systems based on the extensions of later specifications. The model allows expressing parallelism in a *fork-join* fashion. Parallelism is spawned when a parallel construct is reached, creating a team of threads, and joined when the implicit barrier at the end of a parallel region in found. Furthermore, parallelism can be spawned following two different

paradigms: the *thread-based model*, which allows for data parallelism, and the *task-based model*, which allows for task parallelism.

Synchronization of threads occurs when a barrier construct is found, and also based on flush directives. Synchronization of tasks occurs based on taskwait and taskgroup directives, and also on task dependencies, hence enabling a data-flow model.

OpenMP offers a relaxed-consistency, shared-memory model that defines three different views of the memory: a shared space accessible to all threads called *memory*; a *temporary view* of memory for each thread; and a *threadprivate* memory, private to each thread that cannot be accessed by any other thread.

The *accelerator model*, based on the tasking model, is an extension that pursues portability between devices with different ISAs, as well as programmability, by easing the burden of defining data movements between the host and the accelerator, and performance boosted by inserting accelerated parts in the applications. This is a host-centric model where a host device offloads computation to one or more target devices with their own local storage.

The OpenMP accelerator model defines a thread hierarchy where *OpenMP threads* (equivalent to *CUDA threads*), are wrapped into *teams* (equivalent to CUDA *thread blocks*), which are in turn wrapped into *leagues* (equivalent to CUDA *grids*). The parallel for construct can be used to exploit the former, while the teams and distribute constructs are used to exploit the two latter.

```
/* main.c */
int matrix_mul(float A[N][N], float B[N][N], float C[N][N]) {
   #pragma omp target device(0) map(to:A[0:N*N], B[0:N*N])
                    map(from:C[0:N*N]
   #pragma omp teams distribute parallel for private(i,j,k)
   for (i=0; i<n; i++)
      for (j=0; j<n; j++)
         for (k=0; k<n; k++)
            C[i][j] += A[i][k]*B[k][j];
}
int main() {
   …
   matrix_mul(A, B, C);
   …
}
```

Listing 3. OpenMP accelerator model matrix multiplication example.

Listing 3 shows an example of the computation of a matrix multiplication using the OpenMP accelerator model. The user code remains the same as for a sequential version of the benchmark. Just additional directives are inserted so the compiler knows how to do the transformation to exploit that computation in the accelerator.

3.4 Analysis of the Programming models productivity

The programming models used within the Rising STARS project must facilitate the expression of the required levels of parallelism to exploit all hardware resources in the underlying heterogeneous architecture, as well as describing the program data-flow. Table 1 shows a comparison in terms of forms of parallelism and architecture abstraction features available in the parallel programming models just presented. Further, Table 2 compares the models based on synchronization, mutual exclusion, language binding, error handling and tool support. Overall, OpenMP provides the most

comprehensive set of features to support a wide range of parallelism patterns, synchronizations and architectures, on both the host and the device, by allowing modelling the memory hierarchy. Additionally, OpenMP offers two advantages over the rest of programming models. First, it defines an emerging error model that includes features for cancelling parallel execution, i.e., aborts an OpenMP region and causes executing tasks to proceed to the end of the cancelled region. Proposals to extend this model with further extend this model with support for call-backs, and other resiliency mechanisms already exist [5]. Second, it allows binding the computation with the data by defining a binding policy (i.e., *master*, to assign all threads to the same place[1] as the master thread; *close*, to assign threads close to its parent; and spread, to create a sparse distribution of the threads of a team among the set of places of the parent's place partition[2]) to a parallel region.

Table 1. Comparison of the presented parallel programming models based on parallelism patterns and architecture abstraction (extended from [6]).

	Parallelism			Architecture abstraction		
Parallel Programming Model	**Data parallelism**	**Asynchronous task parallelism**	**Host/device**	**Abstraction of memory hierarchy**	**Data and computation binding**	**Explicit data mapping host/device**
OpenMP	parallel for simd	task/taskloop	Host and device (target)	OMP_PLACES, teams and distribute	proc_bind	map(to|from|tofrom|alloc)
CUDA	<<<…>>>	Async kernel launch and memcpy, CUDA graphs	Device only	Blocks/ xthread shared memory	-	cudaMemcpy
OpenCL	kernel	clEnqueueTask	Host and device	Work-group and work-item	-	bufferWrite

Table 2. Comparison of the presented parallel programming models based on synchronizations, mutual exclusions, language binding, error handing and tool support (extended from [6]).

	Synchronizations							
Parallel Programming Model	**Barrier**	**Reduction**	**Join**	**Point to point (data-flow)**	**Mutual exclusion**	**Language library**	**Error handling**	**Tool support**
OpenMP	barrier	reduction	taskwait	depend	Locks, critical, atomic,	C/C++ and Fortran	cancel	OMPT interface/ Extrae [7]

[1] In OpenMP, a *place* is an unordered set of processors on a device.

[2] In OpenMP, a *place partition* describes the places currently available to the execution environment for a given parallel región.

					single, master	based directives		
CUDA	_syncthreads	-	-	cudagraph	atomic	C/C++ extensions	-	NVIDIA profiling tools
OpenCL	work_group barrier	work_group reduction	-	-	atomic	C/C++ extensions	exceptions	System/vendor tools

Nonetheless, for the exploitation of the accelerator devices, OpenMP might not provide the best performance compare to dedicated languages such as CUDA for NVIDIA GPUs. In these cases the programmability might be harmed by the difficulty of using CUDA. In this regard, there is a proposal [8] that use the OpenMP programming language to define workflows including the description of data-flow and data movement host-device that can later be transformed into CUDA graphs for enhanced performance opportunities. In a similar line, a very similar programming model to OpenMP, the OmpSs programming model developed at BSC, also runs on GPUs and FPGAs; for the former, it uses kernels written with OpenCL and CUDA, and for the latter it uses high-level OmpSs C/C++ and compiler transformations [9] for lowering the code to the target device. OmpSs supports the OpenMP programming model and so it will be considered within the OpenMP programming model.

Overall, OpenMP is very suitable for parallelizing applications thanks to the features it includes to fine-tune the parallelization process, as well as its support for both host and accelerator execution and data-flow description and data movement. OpenMP offers great programmability because it is based on compiler directives that can incrementally be inserted in the sequential source code to achieve better performance. Moreover, OpenMP include a nice interoperability with CUDA, OpenCL and FPGA kernels, and has several mechanisms to control the runtime behavior, exploiting the performance capabilities of heterogeneous computing required by Rising STARS use-cases.

4 OpenMP Support for functional and non-functional requirements

OpenMP originally targets HPC systems, being its main focus to expose features for exploiting performance. There are however several works that push the introduction of OpenMP into other domains in which other non-functional requirements must be fulfilled. For such a purpose, the OpenMP has to be adapted to meet functional safety and time-predictability. Several features and techniques have been proposed targeting these aspects. Following paragraphs describe these proposals.

4.1 Functional safety and correctness

The *functional safety* of the OpenMP specification has been analyzed [10]. This work shows that, although certain features might jeopardize the analyzability of the system, minimal limitations on the available features together with the use of two new directives for enabling full-system analysis even in the existence of third-party libraries, may cover most of the possible sources of non-determinism introduced by the specification.

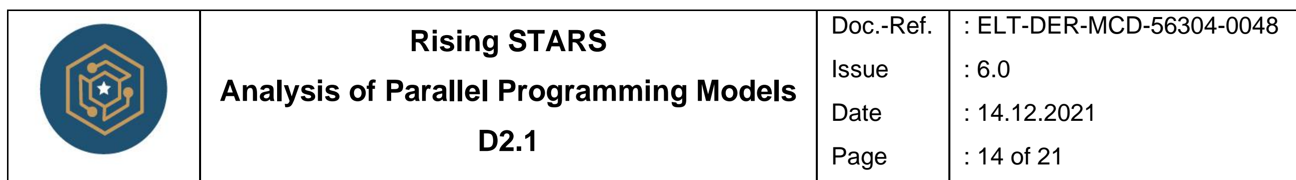
Several *correctness* techniques aiming at delivering fault-free OpenMP systems have been developed. These mainly target dead-locks and data-races. Regarding the former, there are techniques that apply to different programming models, like Sherlock [11] and Chord [12], targeting effectiveness. More interestingly, there is a sound technique for detecting dead-locks in C/Phtreads programs [13] that can be easily applied to OpenMP. Regarding the latter, there are techniques that retrieve data races in specific subsets of OpenMP, like a fixed number of threads [14], or using affine constructs [15]. More general approaches also exist, providing no false negatives [16], or at least providing one race when races are present [17].

Programmability (and productivity) has also been largely tackled in OpenMP. In this regard, different works consider the use of compiler-analysis techniques for relieving the user from defining certain information that can be automatically derived [18] [19] [20]. This works enhance not only the programmability of the model, but also the correctness expectations, because of two reasons: (1) they automatize certain tasks avoid possible human errors, and (2) they include compiler analysis techniques that can be used to check the correctness of the system based on the users definitions.

Resiliency is also an aspect that has been considered in OpenMP. In order to enhance the reliability of the framework, different proposals for including an error model in the specification have been provided [21] [5]. These aim at providing programmers with the tools for recovering the system at certain points where the parallel execution might fail. In this regard, the OpenMP specification includes one mechanism targeting resilience (and also performance), which is *cancellation*. Two directives allow defining points at which the parallel execution can be resumed and the regions to be cancelled (e.g., a parallel region or a taskgroup region).

4.2 Time predictability

Although OpenMP has not been designed for providing timing guarantees, previous works have tackled this aspect. The mainly focus on the OpenMP tasking model because the TDG resembles the Direct Acyclic Graph (DAG) scheduling model used in real-time systems for verifying the timing constraints of the tasks, and tackle both tied [22] and untied tasks [23].



OpenMP lacks however the concept of time. Based on the previous works, different extensions to the OpenMP specification regarding tasks have been proposed [24] in order to consider time:

- *Recurrency*: in real-time systems, tasks are either periodic or sporadic triggered by an event. In this sense, a new clause, named `event`, containing the event that triggers a task has been proposed.
- *Deadlines*: the criticality of a task can be related to the point in time at which the task has to be finished. Several schedulers, like earliest deadline first (EDF) and least laxity (LL), use this information to prioritize the tasks. A new clause, named `deadline`, containing the expression that determines the time instant at which the task must finish has been proposed.
- *Time management* in the runtime: the control loop used in real-time systems to trigger tasks has to be implemented in the OpenMP runtime. An extension derived from this is the concept of *persistent task* [25].

The scheduling decisions are paramount for the time predictability of the system. In this sense, the use of work-conserving schedulers is paramount to avoid incorrect or too pessimistic timing analysis [23] [22]. Work conserving policies can be ensured within OpenMP teams, but there is a limitation when different parallel regions can run in parallel: the specification states that the number of threads in an OpenMP team cannot vary during the life of the parallel region to which it was associated. In this regard, there is a proposal that considers the cooperation between different OpenMP teams (different OpenMP parallel regions) in order to avoid idle cycles in threads from one team when there is work to do in a different team, but works at an OS-thread level and is limited but the aforementioned limitation.

4.3 Energy

Power and so energy management is also an aspect that has been considered in OpenMP, and the importance of power management has already been noted [26]. There is a proposal for extending the OpenMP specification in order to allow addressing the issue of energy consumption and power management [27]. This work provides also the compiler and runtime systems that fulfill these constraints. The extensions proposed include multi-objective optimization goals with a clause that allow providing the goals in terms of execution time, power, energy and quality of service. A different proposal tackles the energy consumption from a cost-per-operation point of view, and defines extensions to model to allow defining the accuracy of the floating point operations [28].

Horizon 2020 R&I programme under the Marie Skłodowska-Curie grant agreement No 873120

5 Suitability of OpenMP on RisingSTARS Adaptive Optics Real-Time Controller use-cases

This section discusses the use of the OpenMP programming model to develop the AO RTC use-case considered within the Rising STARS and described in Deliverable D3.1*" Data Acquisition Solution specifications".*

The AO RTC is composed of three main blocks:

- A Hard real-time controller (HRTC), responsible of computing the commands to be applied to the deformable mirrors to compensate the wave-front aberration, requires to provide guarantees on high throughput, low latency and low jitter.
- A Soft real-time cluster (SRTC), reponsible of configuration, calibration and optimisation of the hard real-time pipeline (supervisor) and for data telemetry and storage (telemetry)
- A Simulator system, providing accurate simulation at real-time frame rates of hardware components in their absence, such as WFS cameras and deformable mirrors.

Among them, the HRTC is the most critical and performance demanding component to guarantee the correct operation of the telescope. As the most demanding part of the HRTC computation is memory bound, the design of the HRTC HW is driven by the ability to leverage the performance of GPU and FPGA based acceleration devices with streaming data from the camera sensors.

Figure 1 presents a block diagram of the main components of the HRTC and the corresponding accelerator device that Rising STARS will consider. Concretely:

- The image captured by the wavefront sensor camera will be processed by the FPGA-based device.
- The processed image will be stored within the memory of GPU-based accelerators from which the DM commands will be computed.
- This commands will be then transformed to actions on the different DM actuators and transferred to the DM.

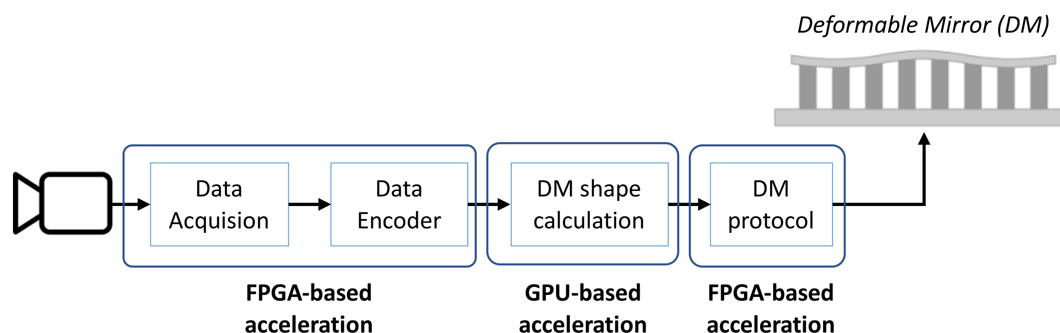


Figure 1. General overview of the accelerator devices considered within the AO RTC.

To achieve deterministic performance, two aspects are of paramount importance: (1) routing efficiently these streaming data from the FPGA to the GPU memory in which the commands of the deformable mirror (DM) will be computed, and (2) minimize the synchronization between the host and the device.

Horizon 2020 R&I programme under the Marie Skłodowska-Curie grant agreement No 873120

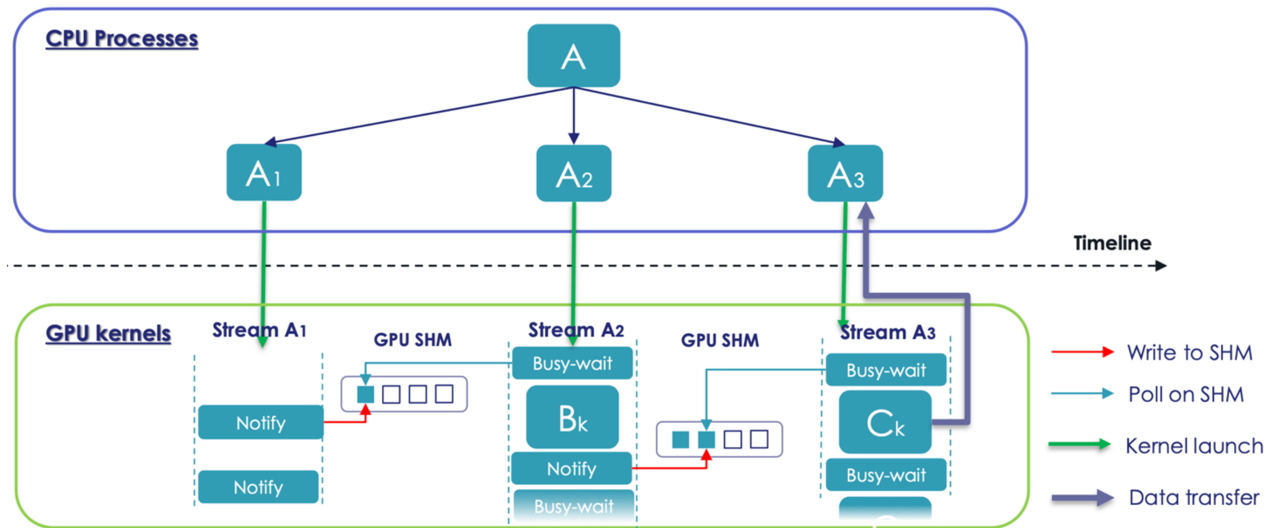


Figure 2. General structure of a general HRTC pipeline.

Figure 2 shows a general schema of a pipeline implemented within the GPU part of the HRTC, composed of a CPU process that spawns three CPU threads, each responsible of: (1) offloading GPU kernels B_k and C_k and (2) implementing a custom synchronization mechanism using CPU-GPU shared memory.

In order to enhance the productivity on the development and execution of such a complex system, the Rising STARS project will leverage the capability of OpenMP to:

1. describe the data-flow of the different computing elements involved within the HRTC,
2. configure the data-transfers and kernels to be executed on the GPU-based accelerator with minimum interaction between the host and the device,
3. include the FPGA-based data transfer mechanisms required to copy the data directly on GPU shared memory to make them available for all GPU computing processes.

```
void A() {
   while(control_loop_active) {
      #pragma omp target depend(out:img) {
         FPGA_kernel_Data_Acquision(…);
      }
      #pragma omp target depend(in:img) {
         FPGA_kernel_Data_Encoder(…);
      }
      #pragma omp taskwait
      #pragma omp target map(to:new_data, from:bk_data) depend(out:bk_deps) {
         GPU_kernel_Bk(…);
      }
      #pragma omp target map(to:bk_data, from:out_data) depend(in:bk_deps) {
         GPU_kernel_Bk(…);
      }
      #pragma omp taskwait
   }
}
```

Figure 3. Simplified OpenMP source coding implementing Figure 1 pipeline.

Figure 3 presents a simplified version of the OpenMP source coding implementing the HRTC pipeline presented in Figure 1 and Figure 2. As shown, the OpenMP syntax allows to describe most of the revelant aspects involving CPU-GPU orchestration (by means of the `target` construct), program data-flow description ((by means of the `map(to/from)` clause) and synchronization dependencies (by means of the `depend` clause).

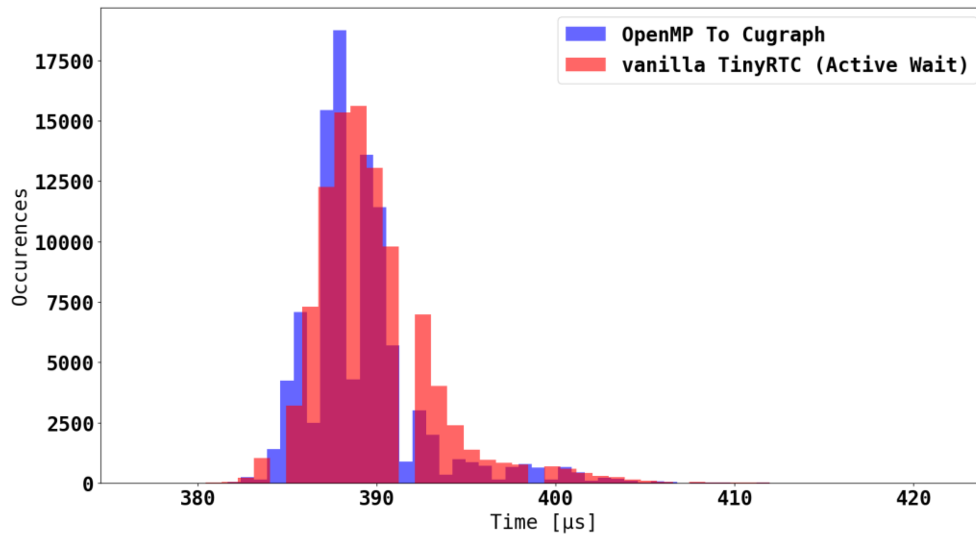


Figure 4. Execution time histogram of a simplified HRTC pipeline.

Figure 4 shows preliminary results in the form of the execution time histogram of a simplified HRTC pipeline composed of 5 stages, implemented using the strategies shown in Figure 2 (labelled as *vanilla TinyRTC*) and Figure 3 (labelled as *OpenMP to Cugraph*).

As shown, the OpenMP implementation provides the same performance as the customised solution (providing even less jitter than the *active_wait* solution), with a simplifier coding.

6 Conclusions

This deliverable analyses different programming models and concludes that OpenMP is the most suitable for the RisingSTARS use-cases, due to its fine-grain parallel capabilities, its support for both host and accelerator execution and data-flow description and data movement. OpenMP also include a nice interoperability with CUDA, OpenCL and FPGA kernels, and has several mechanisms to control the runtime behavior, exploiting the performance capabilities of heterogeneous computing required by Rising STARS use-cases.

Horizon 2020 R&I programme under the Marie Skłodowska-Curie grant agreement No 873120

7 Bibliography

[1] NVIDIA®, "CUDA C++ Programming Guide," June 2020. [Online]. Available: https://docs.nvidia.com/pdf/CUDA_C_Programming_Guide.pdf. [Accessed June 2020].

[2] Khronos OpenCL working group, "The OpenCL™ Specification v3.0.1-provisional," Apr 2020. [Online]. Available: https://www.khronos.org/registry/OpenCL/specs/3.0-unified/pdf/OpenCL_API.pdf. [Accessed June 2020].

[3] Y. Wen, Z. Wang and M. F. O'boyle, "Smart multi-task scheduling for OpenCL programs on CPU/GPU heterogeneous platforms," in *21st International conference on high performance computing (HiPC)*, 2014.

[4] OpenMP ARB, "OpenMP 5.0 Specification," Nov 2018. [Online]. Available: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf. [Accessed June 2020].

[5] M. Wong, M. Klemm, A. Duran, T. Mattson, G. Haab, B. R. de Supinski and A. Churbanov, "Towards an error model for OpenMP," in *International Workshop on OpenMP*, 2010.

[6] Y. Yan, B. M. Chapman and M. Wong, "A Comparison of Heterogeneous and Manycore Programming Models," 2015. [Online]. Available: https://www.hpcwire.com/2015/03/02/a-comparison-of-heterogeneous-and-manycore-programming-models/. [Accessed June 2020].

[7] BSC Performance Tools, "Extrae Documentation release 3.8.0," June 2020. [Online]. Available: https://tools.bsc.es/doc/pdf/extrae.pdf. [Accessed June 2020].

[8] C. Yu, S. Royuela and E. Quiñones, "OpenMP to CUDA graphs: a compiler-based transformation to enhance the programmability of NVIDIA devices," in *23rd International Workshop on Software and Compilers for Embedded Systems*, Sankt Goar, Germany, 2020.

[9] A. Filgueras, E. Gil, D. Jimenez-Gonzalez, C. Alvarez, X. Martorell, J. Langer, J. Noguera and K. Vissers, "OmpSs@ Zynq all-programmable SoC ecosystem," in *ACM/SIGDA international symposium on Field-programmable gate arrays*, 2014.

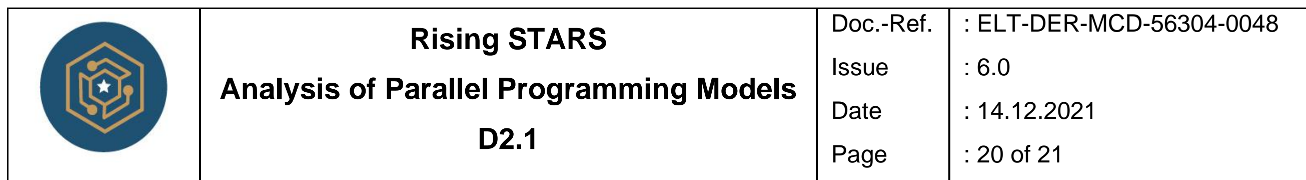
[10] S. Royuela, A. Duran, M. A. Serrano, E. Quiñones and X. Martorell, "A Functional Safety OpenMP* for Critical Real-Time Embedded Systems," in *International Workshop on OpenMP*, 2017.

[11] M. Eslamimehr and J. Palsberg, "Sherlock: scalable deadlock detection for concurrent programs," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014.

[12] M. Naik, C.-S. Park, K. Sen and D. Gay, "Effective static deadlock detection," in *IEEE 31st International Conference on Software Engineering*, 2009.

[13] D. Kroening, D. Poetzl, P. Schrammel and B. Wachter, "Sound static deadlock analysis for C/Pthreads," in *31st IEEE/ACM International Conference on Automated Software Engineering*, 2016.

Horizon 2020 R&I programme under the Marie Skłodowska-Curie grant agreement No 873120



[14] H. Ma, S. R. Diersen, L. Wang, C. Liao, D. Quinlan and Z. Yang, “Symbolic analysis of concurrency errors in OpenMP programs,” in *42nd International Conference on Parallel Processing*, 2013.

[15] V. Basupalli, T. Yuki, S. Rajopadhye, A. Morvan, S. Derrien, P. Quinton and D. Wonnacott, “ompVerify: polyhedral analysis for the OpenMP programmer,” in *International Workshop on OpenMP*, 2011.

[16] Y. Lin, “Static nonconcurrency analysis of OpenMP programs,” in *International Workshop on OpenMP*, 2005.

[17] U. Banerjee, B. Bliss, Z. Ma and P. Petersen, “A theory of data race detection,” in *Workshop on Parallel and distributed systems: testing and debugging*, 2006.

[18] S. Royuela, A. Duran, C. Liao and D. J. Quinlan, “Auto-scoping for OpenMP tasks,” in *International Workshop on OpenMP*, 2012.

[19] S. a. D. A. a. M. X. Royuela, “Compiler automatic discovery of OmpSs task dependencies,” in *International Workshop on Languages and Compilers for Parallel Computing*, 2012.

[20] H. Ma, S. R. Diersen, L. Wang, C. Liao, D. Quinlan and Z. Yang, “Symbolic analysis of concurrency errors in OpenMP programs,” in *42nd International Conference on Parallel Processing*, 2013.

[21] A. Duran, R. Ferrer, J. J. Costa, M. Gonzàlez, X. Martorell, E. Ayguadé and J. Labarta, “A proposal for error handling in OpenMP,” *International Journal of Parallel Programming,* vol. 35, no. 4, pp. 393--416, 2007.

[22] J. Sun, N. Guan, Y. Wang, Q. He and W. Yi, “Real-time scheduling and analysis of openmp task systems with tied tasks,” in *IEEE Real-Time Systems Symposium (RTSS)*, 2017.

[23] M. A. Serrano, A. Melani, R. Vargas, A. Marongiu, M. Bertogna and E. Quinones, “Timing characterization of OpenMP4 tasking model,” in *International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, 2015.

[24] M. A. Serrano, S. Royuela and E. Quiñones, “Towards an OpenMP specification for critical real-time systems,” in *International Workshop on OpenMP*, 2018.

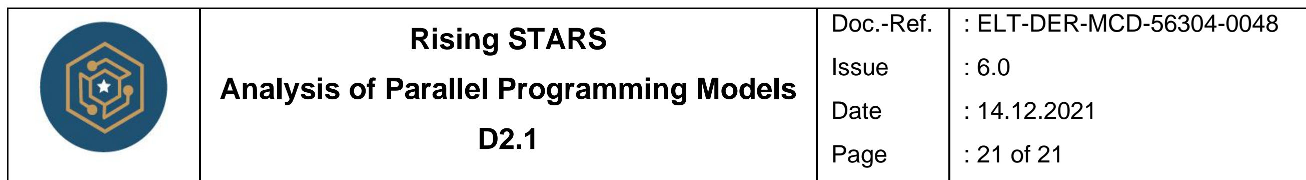
[25] A. Pop and A. Cohen, “A stream-computing extension to OpenMP,” in *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, 2011.

[26] B. Chapman, L. Huang, E. Biscondi, E. Stotzer, A. Shrivastava and A. Gatherer, “Implementing OpenMP on a high performance embedded multicore MPSoC,” in *IEEE International Symposium on Parallel \& Distributed Processing*, 2009.

[27] F. Alessi, P. Thoman, G. Georgakoudis, T. Fahringer and D. S. Nikolopoulos, “Application-level energy awareness for OpenMP,” in *International Workshop on OpenMP*, 2015.

[28] A. Rahimi, A. Marongiu, R. K. Gupta and L. Benini, “A variability-aware OpenMP environment for efficient execution of accuracy-configurable computation on shared-FPU processor

Horizon 2020 R&I programme under the Marie Skłodowska-Curie grant agreement No 873120



clusters," in *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*, 2013.

[29] AMPERE, "D5.1. Reference parallel heterogeneous hardware selection," 2020.