HERCULES:
High-Performance Real-time Architectures for Low-Power Embedded Systems

Project title:	**High-Performance Real-time Architectures for Low-Power Embedded Systems**
Acronym:	HERCULES
Project ID:	688860
Call identifier:	H2020 - ICT 04-2015 - Customised and low power computing
Project Coordinator:	Prof. Marko Bertogna, University of Modena and Reggio Emilia

D3.1: Report on Programming Model(s) Selection

Document title:	Programming Model(s) Selection
Version:	1.1
Deliverable No.:	D3.1
Lead task beneficiary	ETHZ
Partners Involved:	ETHZ, AB, MM, EVI
Author:	Andrea Marongiu, Björn Forsberg (ETHZ)
Status:	FINAL
Date:	30.06.2016
Nature[1]:	R

[1] **For deliverables: R** = Report; **P** = Prototype; **D** = Demonstrator; **S** = Software/Simulator; **O** = Other
For milestones: O = Operational; **D** = Demonstrator; **S** = Software/Simulator; **ES** = Executive Summary; **P** = Prototype

Dissemination level		
PU	Public	X
PP	Restricted to other programme participants (including the Commission Services – CS & IAB)	
RE	Restricted to a group specified by the consortium (including the IAB)	
CO	Confidential to consortium (including CS & IAB)	

This Project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement: 688860

Document history:

Version	Date	Author	Comments
0.1	2016-05-20	ETHZ	First complete draft
0.3	2016-06-04	ETHZ	Finale complete draft
1.0	2016-06-29	ETHZ	Updated due to reviews
1.1	2016-06-30	UNIMORE	Final revision

Table of contents

GLOSSARY

Item	Description
API	Application Program Interface
COTS	Commercial Off-The-Shelf
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
CV	Computer Vision
DSP	Digital Signal Processor
FPGA	Field-Programmable Gate Array
GPGPU	General-Purpose GPU
GPU	Graphics Processing Unit
HPC	High-Performance Computing
JIT	Just-In-Time
MIMD	Multiple Instruction, Multiple Data
NUMA	Non-Uniform Memory Access
SDK	Software Development Kit
SIMD	Single Instruction, Multiple Data
SIMT	Single Instruction, Multiple Thread
SOC	System on Chip

1. EXECUTIVE SUMMARY

As part of WP3, the objective of this report is that of identifying a suitable programming model (or more than one) capable of addressing the numerous requirements for the fulfilment of the project goals at large. The chosen programming model(s) will constitute the basis for developing the predictability extensions to be explored during the next stages of the project.

2. INTRODUCTION

This deliverable describes the activities carried out during the first six months of the project in task 3.1: “Programming model specification”. The main objective of this task is identifying a suitable programming model (or more than one) capable of addressing several requirements that are central to the goals of the project as a whole: ease of programming, productivity, and effectiveness at extracting the available peak parallel processing bandwidth. Besides that, the selected programming model should lend itself to extensions at various levels (i.e., programming interface, runtime environment, and compiler) for improving the overall predictability of the system.

This deliverable first describes the process of analysing and selecting suitable candidates among the several existing programming paradigms for massively parallel systems. Second, a list of criteria for evaluating the various alternatives is proposed. Besides several technical aspects, the evaluation also took into account other relevant requirements such as the availability of industry-supported standards, the vitality of the user/developer community, the maturity of the available tools, the availability of the latter on relevant hardware platforms for the project, and the capability of expressing application-specific, functional and non-functional requirements.

3. THE LANDSCAPE OF PARALLEL PROGRAMMING MODELS

3.1. Architectural template

One of the central ideas of the HERCULES project is that the proposed solutions and framework for enhanced predictability will not be based on the development of new hardware. Instead, cutting-edge heterogeneous COTS platforms will be selected, composed of a multi-core host system coupled with a multi-/many-core accelerator, like a GPU or GPGPU, DSP cluster, FPGA or many-core module. This System-on-a-Chip (SoC) topology is dominating the embedded market thanks to the impressive performance that can be achieved within a limited power consumption.

The selection of the reference platform(s) has been carried out in task 2.1, in parallel to the identification of the programming model, and there have been interactions between the two tasks to take into account cross-requirements. In WP2 several choices for the hardware platform have been considered, each featuring different kinds of accelerators. Table 1 shows the list of the five platforms that have been initially selected out of a larger list of candidates (cfr. D2.1).

Platform	GPU	DSP	FPGA	other
NVIDIA Tegra Parker	*GPGPU*	-	-	*X*
Xilinx Zynq Ultrascale+	*GPU*	-	*X*	
Renesas RCAR H3	*GPGPU*	-	-	*X*
Texas Instruments TDA2 + EVE	*GPU*	*X*	-	*X*
Infineon Aurix 2	-	*X*	-	*X*

Table 1: List of candidate platforms (from task 2.1)

Task 3.1 has thus based its evaluations on explicitly considering a) architectural traits that are common to the different platforms and b) those that imply a relevant difference from the point of view of the programming model. It is important to mention that, since the beginning of the project, there was consensus among the consortium members that NVIDIA offered platforms that seemed to satisfy most of the requirements identified for the project platform. For this reason, we have included CUDA as a possible candidate for programming model selection in the analysis presented in this deliverable. CUDA is NVIDIA-proprietary, thus it comes in completely closed-source form. Extending CUDA for predictability thus introduces additional difficulties (getting access to the source code of NVIDIA drivers and middleware, or devising a solution that builds upon “black boxes”). However, it also provides the most optimized tools for exploitation of NVIDIA hardware, thus it deserves consideration for the analysis presented here.

The main type of accelerator considered in the project consists of a programmable parallel unit, for which the programming model should expose language features to 1) specify blocks of code to be “offloaded”; 2) identify parallelism within the offloaded blocks; 3) orchestrate data mappings, data layout and data movements.
A key architectural feature that impacts the choice of a programming model resides in the execution model supported by the specific accelerator. Specifically, the implications of single-instruction, multiple-data (SIMD) versus multiple-instruction, multiple-data (MIMD) execution style. That is, if the execution units of the accelerator are capable of executing separate instruction streams (i.e. have separate program- and stack-pointers). If this is the case, they can efficiently execute conditional branches and control-oriented code. It is therefore preferable to select a programming model which supports both the SIMD and MIMD approaches.
When the architecture only supports the SIMD execution model (which is typically the case for GPGPUs), allowing MIMD parallelization from the programming model might be detrimental to performance.

3.2. The identikit of a modern programming model

There are several programming models available on the market today, each with its own advantages and disadvantages. From the technical viewpoint, it is important to identify which features offered by modern programming models are mostly relevant to our evaluation. To this aim, we discuss in this section the basics of parallel and heterogeneous computing, to outline the characteristics that will later be used to derive objective evaluation criteria.
When dealing with parallel systems, the first and most important category of features to be assessed is the support for different ***parallelism patterns***, i.e., different types of parallelism for which the model offers support.
When dealing with heterogeneous systems, a second fundamental category of features to be assessed is ***architectural abstraction***, i.e., the extent to which, and the level of abstraction at which, the model allows specifying computation offloading to accelerators and to deploy computations with high data-locality, following the structure of the core and memory hierarchy.
Finally, considering the explicitly-managed nature of the memory hierarchy inside parallel accelerators, language features to control ***data management*** is also very relevant. This category includes constructs to specify static data placement (including specific data layouts that might impact data access latency/bandwidth) and dynamic data movements.

We further analyze each of these categories in the following.

3.2.1 Parallelism patterns

There are several different forms of parallelism that can be exploited in different programs, and the choice of a programming model greatly influences which forms of parallelism can be easily extracted from the algorithms.

Starting at the application level, the algorithms themselves can have **data parallelism**, where the same computation is applied to distinct elements of some program data structures. This implies that the calculation of one output value does not depend on the calculation on any other value, which in turn means that the calculations themselves can be made in any order, and more importantly, completely in parallel. This form of parallelism is commonly found in operations on *arrays* or *vectors*. A simple example could be the squaring of each element in an input array. Here it is clear that there are no data dependencies between the elements of the vector, as the squaring only requires the current element in the input array. For this reason it does not matter if calculations are done from “left-to-right”, “right-to-left”, in any random order, or, more conveniently, all calculations in parallel.

A program may also include **task-level parallelism**, which identifies parts of the program that are independent from each other and that can be flexibly scheduled (and possibly suspended and migrated) among threads. While with data-level parallelism the same operation is executed on each input element, this limitation is not present in task parallelism. Indeed, with task-level parallelism each task may have a completely different control flow, and operate on completely different data, as long as there exist no dependencies between them. *Tasking* is a convenient programming model for those systems that architecturally support MIMD, and it is very well suited for those applications that exhibit irregular parallelism (e.g., functional loops, recursion, based on the traversal of graphs, trees, etc.).

In practice, both task level and data level parallelism may exist in the same program, and the classification of a program does often not fit into only one of these categories. For this reason, programming models typically do not generally focus on a single of these two levels, but it offers tools to exploit the form of parallelism best

suited for each individual part of the program (unless they are meant for a system that only supports SIMD execution, as is the case for GPGPUs and associated programming models, e.g., CUDA).

Parallelism can also be expressed more abstractly, through **graph-based schemes**. In graph schemes, each vertex expresses one or more operations to be done sequentially on a piece of data, and each edge expresses a data dependency to data produced by a previous vertex. In this way, *streaming* or *pipeline* parallelism can be conveniently expressed through the graph abstraction. A program is built up by combining one or more sequential and parallel regions to best express each form of parallelism. Task parallelism can for example be used to identify vertices, possibly constraining their scheduling via explicit dependencies. Internally to a vertex, additional data parallelism can be expressed by dividing the target data (e.g., matrices, images) in *stripes* or *tiles*, to be assigned to parallel threads.

In addition to these forms of parallelism, additional levels of parallelism can be extracted at lower levels. These include **instruction-level parallelism**, in which the execution of individual instructions can be done in parallel. Instruction-level parallelism can be exploited using HW support for SIMD execution, which is available in some form in most current ISAs. Here, a single instruction takes a vector of data elements on which to perform the operation, thus performing the operation on several elements at once while only executing a single instruction. Some programming models provide constructs to explicitly transform certain loop-based computations to fit the SIMD model. In other cases, these optimizations are done at compiler level based on the support offered by the target architecture and not exposed at the programming model level.

3.2.2 Architectural abstraction

The main role of a programming model is that of providing means to effectively use the available hardware while hiding the complexity of achieving this task. Abstracting architectural details behind high-level constructs is key to improve ease of programming, productivity and code maintainability.

In a heterogeneous system, specialization is the key architectural idea to make better use of die area. Specifically, different execution units are typically better suited at different tasks. For example, the main host CPU is a general-purpose unit, perfect for the execution of control-intensive codes and moderately parallel workloads (considering coarse-grained thread-level parallelism). Many parallel accelerators (with GPGPUs on top of the list) are optimized for data parallelism, as the numerous, simple cores are able to efficiently execute the same stream of instructions over a large amount of data items. In this scenario, the programming model should provide efficient means to express both forms of parallelism, and give the programmer ways of controlling which of the computation units a particular workload is to be executed on. This capability is typically referred to as computation **offloading**.

Since most modern programmable accelerators feature a very large number of cores, a common design choice to avoid architectural bottlenecks is hierarchically grouping such cores in small-medium sized computation *clusters*. Intra-cluster communication leverages low-latency, high-bandwidth interconnects, whereas inter-cluster communication exhibits *non-uniform memory access* (NUMA) effects. Such effects are exacerbated by the fact that memory hierarchies within many-core accelerators are typically based (in part, or *in toto*) on explicitly-managed scratchpad memory, rather than data caches. An effective programming model for computation offloading should then provide features to control the deployment of highly parallel computations in a way that is aware of the **memory and core hierarchy**.

3.2.3 Data management

Following the explicitly-managed nature of most many-core accelerators' internal memory systems and considering that several types of memory are usually available (e.g., *private* to each cluster, *shared* among groups of cores/clusters, *global* to the accelerator, etc.), it becomes fundamental for the programming model to provide constructs for **explicit mapping** of data. Specifically, it is important to have i) data *qualifiers* allowing for static placement of permanent data items into the most suitable memory block for the computation at hand; ii) constructs for explicitly moving data within the memory hierarchy.

It is also important to give the programmer efficient means of expressing **data dependencies**, as the offloading of data to an accelerator means that the data must be made available to the accelerator, either by copying the data to the local memory of the accelerator, or, in case of shared memory, pass pointers to the relevant data structures. By providing efficient tools for expressing data dependencies, the instructions for managing the transfer of data can be managed by the compiler tool chain, which provides an opportunity to optimize the data transfer for predictability. If task parallelism is supported by the accelerator (i.e., the MIMD execution model), it is important that constructs for specifying dependencies between tasks executing inside the accelerator are also provided.

3.3. An overview of the most widespread programming models

In this section we provide an overview of the most widespread programming models, which we have selected as initial candidates for our evaluation.

3.3.1 CUDA

CUDA (Compute Unified Device Architecture), is the programming model provided by NVIDIA for general purpose programming of their GPUs. It is built as an extension to C/C++11, where the CPU (*host*) and GPU (*device*) functions are specified by decorating the function and variable declarations.

The execution model, is centered on a host CPU offloading kernels onto the accelerator device, in this case an NVIDIA GPU. Thus, CUDA provides no native methods for expressing parallelism in the host code, although this can be achieved by combining CUDA with other host-centric models, such as *pthreads* or OpenMP. For the kernels executing on the device, CUDA provides two main abstractions: the *execution abstraction* and the *memory abstraction*.

The execution abstraction layer provided to the programmer is made up of a three-layer division into *grids* of *blocks* of *threads*. These concepts are further described in the following paragraph, beginning from the smallest unit, the thread, and moving upwards.

Threads are the smallest unit of execution in CUDA, and the architecture is referred to as Single Instruction Multiple Threads (SIMT). This is similar to SIMD with the exception that the execution is not done using a single instruction on a vector of data, but a vector of threads each operating on a single data element. On all current CUDA compatible hardware, 32 threads are executed at once in *lock-step*, meaning that the same instruction is executed in each thread. This mimics the behavior of a 32-way SIMD instruction, offering a large amount of data-level parallelism.

Threads can be further divided into blocks, which provides means of data sharing between threads within the block, and isolation from threads in other blocks. The threads of each block have access to a shared memory, which can be used to share data between the threads, or be used as a scratchpad memory. Threads within

the same block can also be synchronized using barrier synchronization. In contrast, threads that belong to different blocks have no means of sharing data via scratchpad memory, nor can they be synchronized using any means provided by the programming model[2].

As already hinted at, the model provides means for executing several blocks at once, and just like the threads are organized into blocks, the blocks are themselves organized into grids.

The individual indexes of the threads are commonly used to decide on which data the thread shall operate. The index itself can be decided at the global- or block-level, as each block contains local indexing, which can be exploited for locality within the computation. Furthermore, the threads can be defined in 1, 2 or 3 dimensions, thus giving native support for working on data structures in multiple dimensions.

The memory abstraction in CUDA consists of three levels; *global*, *shared* and *private* memory. The shared memory can be accessed by every thread in a block. Likewise, each thread may keep local variables in its private memory, which can only be seen from the thread itself. Lastly, the global memory is the memory that is available to all threads executing on the device. The global memory in CUDA refers to memory that is shared between different streaming multiprocessors on the device. Note that host memory is logically separated from the GPU's global memory. In discrete GPUs, the separation is also physical, thus host memory is inaccessible from the device-side kernels. This means that the programmer must ensure that the data is available in the global memory before a kernel is started. For integrated GPUs, the host memory and the global memory are typically implemented in the same shared DRAM.

The CUDA kernels themselves have no means of initiating data transfers from the host-specific memory to the global memory, and thus this must be done from the host code using calls to the CUDA API. There are two ways to ensure that the data is mapped to the device memory. The classical method is to manually copy the data using the *memcpy* functionality in the CUDA API. In more current versions of CUDA however, this can be automatically done via the use of what NVIDIA refers to as Unified Memory. This is an additional abstraction layer which lets the CUDA API and hardware automatically "page in" the data to global memory, thereby letting the programmer refer to the data using the host-side pointers on the device as well.

3.3.2 OpenCL

OpenCL, or the Open Computing Language, is the programming model supported by the Khronos Group[3], with the goal of defining a standard for heterogeneous systems programming. In many ways, OpenCL is similar to CUDA in the abstractions offered to the programmer. There are however large differences as well. As with CUDA, the OpenCL model relies on the execution abstraction and the memory abstraction. While the CUDA abstraction is closely tied to the architecture of the NVIDIA GPUs on which the programs are designed to run, OpenCL is somewhat more general in the abstractions provided, as it is designed to be portable over a broader set of devices.

As with CUDA, the CPU is referred to as the *host*, which offloads kernels to an accelerator, referred to as the *device*. Thus, there is no native way to express parallelism on the host side. The execution abstraction available to OpenCL kernels is divided into three levels, where the highest level is the *compute device,* e.g., an accelerator to which the OpenCL runtime can offload kernels. The compute devices are internally divided into *compute units*, which contain a set of processing elements doing the actual work. The processing elements can be thought of as a SIMT lane.

[2] Synchronizing through the main, global memory is always possible, but it clearly comes at a very high cost.
[3] www.khronos.org

As opposed to CUDA, a main goal of OpenCL is to be portable between different host and accelerator systems. For this reason, OpenCL kernels are generally distributed in source form, and compiled by the runtime for the accelerators present in the system the code is executing on. Currently, OpenCL is programmed in a C dialect (OpenCL C), but support for C++ is in the pipeline.

The memory abstraction in OpenCL is divided into global memory, read-only memory, local memory and private memory. Essentially, the abstractions match the ones found in CUDA, where the local memory in OpenCL corresponds to the shared memory in CUDA. The main difference is the addition of the read-only memory, which is a low-latency memory readable from both the host and device, although it can only be written from the host side.

3.3.3 OpenVX

OpenVX is a graph-based programming model specialized on Computer Vision (CV) workloads. The abstraction provided is given at a very high level which essentially hides the hardware altogether. Instead, the programmer defines graphs of predefined or custom nodes that perform operations common to the CV field.

In OpenVX, all interactions with the hardware are done by the runtime, which is tasked with deciding which operations shall be executed on which computing device, and in which order. This removes the need of hardware knowledge from the end programmer, which can focus on the implementation of the high-level functionality of the program.

Below the abstraction layer given to the end programmer, however, the runtime libraries themselves are very closely related to the hardware. As the programmer cannot make any optimizations, they rely on the implementation of the runtime libraries to be finely tuned for the underlying hardware.

While OpenVX provides a convenient abstraction for writing portable CV-based applications for accelerators, it lacks flexibility, as other types of parallelism are not explicitly supported.

3.3.4 OpenMP

OpenMP has long been the de-facto standard for defining task- and data-level parallelism on shared memory multi-core systems. It is a directive driven model, in which the code is written as a serial sequence of instructions (using the well-known C or C++ language[4]), where directives are used to specify regions that contain data or task level parallelism. This approach is very portable, as the serial code remains the same, while the parallelization of the code is done by the compiler, where it can be optimized for the target architecture. For this reason, the same code can be used to generate parallel code for any platform with an OpenMP compliant compiler. For platforms which do not have compilers which support OpenMP, the OpenMP directives, which are given as preprocessor pragmas, are simply ignored, and a sequential program is generated.

For multi-cores, OpenMP has enabled the programmer to specify task and data level parallelism using different pragmas. Task level parallelism has been defined by enclosing different regions of the program which can be executed in parallel with task directives. This instructs the compiler to fork into different threads to execute the different regions, after which the threads are rejoined. For data level parallelism, a similar approach is used, but the parallelism is expressed in loops. In sequential programs, each element would be calculated in sequence by the loop; with the addition of the parallel loop pragma, the OpenMP compiler will spawn a number of parallel threads, each executing one or more of the loop iterations in parallel.

[4] Fortran is also supported by the standard, but it is not used in the embedded domain and so it is not relevant to our evaluation.

OpenMP has been used to compile code for multi-core systems since the late 1990s. With the introduction of OpenMP 4.0 in 2013, the standard was extended to include directives which enable offloading to accelerator co-processors, or, in OpenMP terminology, offloading computations from the host to a device. The programmer specifies a region to be offloaded by enclosing the region with the target pragma, which tells the compiler that the region is suitable for offloading. To support a general class of accelerators, the target region can then contain any combinations of sequential, task parallel and data parallel regions. If no further instructions are given as part of the directive, the compiler will figure out which data needs to be moved to and from the accelerator, but, for increased control and the possibility of optimizations, this can also be explicitly defined by the programmer. In addition to this, data can be kept on the device between offloaded regions using the target data directive, minimizing the need for redundant transfer of data. Lastly, OpenMP provides the *team* and *distribute* abstractions which can be used to express groups (teams) of threads which have data locality, and to further specify the independence of the members of different teams, i.e., no synchronization between threads in different teams. This mimics the behavior of thread blocks in CUDA.

Depending on the final choice of target architecture, it might be the case that only the data parallel regions are really suitable for offloading (i.e., if the accelerator only supports the SIMT execution model), however, the support for task level parallelism may be fully utilized on the CPU.

3.3.5 OpenACC

OpenACC is, similarly to OpenMP a directive driven model, which presents a similar view of the data dependencies and management of offloads. It was created to provide means for computational offloading to accelerators, which at the time of founding was not available in OpenMP. For this reason, OpenACC is specialized for this kind of offloading, and does not provide any native means for expressing parallelism on the host side. However, this limitation can be overcome by combining OpenACC with other models which provide this functionality.

While there is a large overlap in the offloading functionality between OpenACC and OpenMP, the directives of OpenACC are mostly focused on loop-level parallelism, which is very well-suited for GPU-like accelerators. The mature accelerator support in OpenACC is more or less continuously in the process of being merged into OpenMP 4.x. For this reason, OpenACC can be seen as the fast moving, quick implementation of applications supporting accelerator offloading, while OpenMP is the slower moving, more stable model into which the tried and proved parts of OpenACC is being merged[5].

3.3.6 Cilk+

Cilk+ enables the programmer to express parallel behavior by using a number of Cilk+ specific keywords which extend C++. The principle behind the design of the Cilk language is that the programmer should be responsible for exposing the parallelism, identifying elements that can safely be executed in parallel; it should then be left to the run-time environment, particularly the scheduler, to decide during execution how to actually divide the work between processors. It is because these responsibilities are separated that a Cilk program can run without rewriting on any number of processors, including a single one.

Cilk+ is focused on task-based parallelism, with sophisticated runtime schedulers aimed at achieving the most from the available HW and SW concurrency. Cilk+ also supports the parallel execution of loop iterations, function calls and the vectorization of operations. It also provides means for lock-less reduction of values calculated in parallel.

[5] Source: http://www.slideshare.net/insideHPC/jeff-larkin

The main target of the Cilk+ programming model is SMPs.

3.3.7 TBB

Similar to Cilk+, TBB focuses on task parallelism, where the programmer specifies parallel execution through a graph abstraction and the runtime implements work stealing-based task schedulers. The main roles of the programmer are two: i) to divide the work into sequential functions which can be executed in parallel and to create edges between the nodes which specify the data dependencies; ii) to create, synchronize and destroy such graphs of dependent tasks according to *algorithms*, or *Algorithmic Skeletons*.

There are several specialized kinds of nodes within the TBB model which behave differently when exposed to input, and generate different kinds of output, however, they are all capable of executing user-defined functions. In addition to this, the model provides means for expressing different kinds of work queues, fork-join patterns, locks, and other primitives relevant for parallel workloads.

The approach taken by TBB, where the parallelism in the algorithm is expressed implicitly through the graph, and the runtime is tasked with executing the program in the most efficient way on the available execution units, positions this programming model in a family of solutions for parallel programming aiming to decouple the programming from the particulars of the underlying machine.

Consequently, as with Cilk+, the TBB model is currently focused on SMP.

4. Evaluation

In this Chapter we present the evaluation carried out to select the most promising programming models for the project. We first focus on defining technical evaluation criteria, which allow us to filter out those options that are unsuited to achieve the main technical goals (i.e., efficient exploitation of heterogeneous hardware). Second, we consider a set of non-functional criteria that are all the same fundamental for the evaluation, in that they define the practical usability of the selected solution and the industrial impact that such a solution could have for the project in the longer term. Third, and last, we consider the tools available.

4.1. Technical Evaluation

From the discussion provided in Chapter 3, we have derived the following set of technical evaluation criteria:

- Parallelism patterns
  - Data-level Parallelism
    - Data parallelism: The constructs available within the programming model to express data-level parallelism.
    - Instruction-level parallelism: The constructs available within the programming model to express low-level instruction-level parallelism.
  - Task-level Parallelism
    - Task parallelism: The constructs available within the programming model to express task-level parallelism.
    - Graph parallelism: The constructs available within the programming model to express graph-level parallelism, that is, a set of tasks whose data dependencies can be specified using the graph abstraction.
- Architectural abstraction
  - Offloading: Does the programming model have means of expressing code segments that are suitable for executing on the accelerator?
  - Memory and core hierarchy: Which means does the programming model have for expressing the locality of execution and memory units?
- Data movements
  - Explicit mapping: The constructs available for explicitly placing or moving data within the memory hierarchy.
  - Data dependency: The constructs available for expressing data dependencies between different sequential or parallel regions.

For increased readability, the characteristics of each programming model is presented in Table 2, where the main supporting construct of each evaluation criteria is highlighted.

	Parallelism patterns				Architectural abstraction		Data placement	
	Data level parallelism		Task level parallelism					
	Data parallelism	Instruction-level parallelism	Graph-based parallelism	Task parallelism	Offloading	Memory and core hierarchy	Data dependency	Explicit mapping
OpenMP	parallel for directive	parallel for simd directive	task depend(in|out|inout) directive	task/taskwait directives	target directive	teams/teams distribute directives	Depend (in|out|inout)	map (to|from|tofrom|alloc)
Cilk+	cilk_for, element function	Simd	cilk_spawn/sync	cilk_spawn/sync	*None*	*None*	*None*	*None*
TBB	parallel_for/while/do, etc	*None*	tasks	task::spawn/wait	*None*	*None*	Pipeline, parallel_pipeline	*None*
OpenACC	kernel/parallel	vector, vector_length	*None*	async/wait	acc directive	cache, gang/worker/vector	wait	Data copy/copyin/copyout/...
CUDA	Kernel	*Hidden*	dynamic parallelism	async kernel launching and memcpy	<<<...>>>	blocks/threads, shared memory	stream	cudaMemcpy function
OpenCL	Kernel	*Hidden*	dynamic parallelism	Async kernel launching	clEnqueue*	work group/item	pipe	bufferWrite function
OpenVX	*Hidden (implementation specific)*	*Hidden (implementation specific)*	nodes	*Hidden (implementation specific)*	*Hidden*	*Hidden*	graph	*None*

Table 2 : Evaluation of several programming models based on technical evaluation criteria. Meanings of classifications: Hidden = Handled by runtime or compiler, None = Not supported.

From the results in Table 2, it is clear that there are several differences between the programming models analyzed. Most prominently, the support for *offloading* and *memory management* is very different. In OpenMP and OpenACC, the programmer specifies regions that are suitable for offloading through the use of directives, which is known to increase ease of programming and code maintenance. Likewise, offloading is a central concept in CUDA and OpenCL, with the difference that the programmer explicitly writes kernels which are executed on the accelerator, possibly using different languages/dialects to specify parallelism on the host and on the accelerator. This programming style implies a significantly higher degree of programmer involvement in practical deployment aspects and it is more error-prone and less maintainable. Cilk+ and Intel TBB provide no explicit control over offloading to the end programmer. Since a main feature of the target template platform is the availability of a many-core accelerator, it seems unwise to build upon a baseline programming model which does not provide good abstractions and tools for the use of accelerators.
OpenVX deserves a separate discussion. The main purpose of the OpenVX initiative is that of providing a portable and abstract interface for describing graph-based applications composed of a set of "filters" (computer vision kernels). Thus, an OpenVX framework provides an API to build the graphs and a runtime system to schedule their execution respecting precedence constraints (data dependencies). OpenVX targets the acceleration of computer-vision (CV) kernels, but it is platform-agnostic, for portability. Thus, the implementation of the kernel themselves is platform vendor-specific and internally implements the kernel functionality by leveraging the available accelerators.
Consequently, OpenVX is very well suited for simplifying the development of accelerated CV-based applications, as it completely hides the accelerators from the programmer. However, it is poorly suited for the goals of the HERCULES project, in that i) it lacks generality (only focuses on CV); ii) the hidden nature of all the acceleration/parallelization makes it very difficult, if not impossible, to devise predictability extensions to the model (which is the focus of the work in WP3).
The following subsections will briefly describe the general observations made from Table 2 with respect to the different sets of evaluation criteria.

4.1.1 Parallelism patterns

In general, all programming models that are under evaluation support some form of task-based parallelism, which is good. However, it shall be noted that OpenACC is only capable of managing the offloading of kernels to the accelerator, which makes the task model too unexpressive to support graph-based parallelism in a good way. As comparison, the similarly constructed, but more general OpenMP allows the programmer to specify several sequential and parallel regions within an offloading section, making it far more expressive. This also means that offloaded sections can contain graph-based parallelism through the use of tasks embedded within the target section.
Furthermore, most programming models lack a way to explicitly express instruction-level parallelism, with the exception of Cilk+ and OpenMP.
Lastly, OpenVX is designed to provide an easy way to create workflows from common computer visions building blocks that are provided as part of the programming model. This very abstract way of defining program behavior means that there is no way to express data parallelism, as this is hidden within the runtime provided by the hardware vendor or third party.
Based on this, we give the different programming models scores from 1 to 5, where 5 is the most complete support for different programming models, and 1 is the least amount of support.

Requirement	OpenMP	Cilk+	TBB	OpenACC	CUDA	OpenCL	OpenVX
Data parallelism	5	5	3	3	4	4	1
Task parallelism	5	5	3	3	1	3	4

4.1.2 Architectural abstraction

With respect to the architectural abstraction, the main point of interest is the support for offloading. Here it is clear that only four of the seven programming models under evaluation natively provide such support. These models are OpenMP, OpenACC, CUDA and OpenCL.
With respect to Cilk+ and TBB, offloading support is not present at all, as these models are mainly focused on SMPs. For OpenVX, offloading support *may* be available, as the underlying runtime can be implemented in such a way that some of the nodes are executed on an accelerator. However, and most importantly, this is not transparent or controllable by the end programmer, and therefore OpenVX contains no means for explicitly controlling which parts of a program that should be offloaded.
The second requirement in this section is the amount of control the programmer has over where data is placed within the memory hierarchy. While each offload capable model has a somewhat different approach, with different names and expressions, the overall level of abstraction is the same. The execution is divided into three levels which can be thought of as threads, thread-blocks, and programs. Likewise, the NUMA effects of memory placements can be controlled from all models, as they all provide means for defining data affinity.

Requirement	OpenMP	Cilk+	TBB	OpenACC	CUDA	OpenCL	OpenVX
Offloading	5	0	0	4	4	4	1
Memory and core hierarchy	5	0	0	5	5	5	0

4.1.3 Data placement

All programming models under evaluation except Cilk+ provide native means for expressing data dependencies between different tasks. This is particularly useful for graph-based parallelism patterns, where each node can start executing once the data from the previous node is available. All programming models that were identified to provide explicit offloading capabilities (CUDA, OpenACC, OpenCL, OpenMP), also provide means for explicit data mapping between the host and accelerator.

Requirement	OpenMP	Cilk+	TBB	OpenACC	CUDA	OpenCL	OpenVX
Data dependencies	5	0	2	3	5	5	5
Explicit mapping	5	0	0	5	5	5	0

4.1.4 Final Verdict

Based on Table 2, Figure 1 shows the number of requirements satisfied in each of the sets of evaluation criteria. Here, the requirement fulfilment of four of the programming models stand out. In particular, these four programming models, i.e., OpenMP, OpenACC, CUDA and OpenCL, all provide good architectural abstractions and programmer control over data placement. Because of this, these models will be further evaluated with respect to the non-functional requirements in the next subsection.

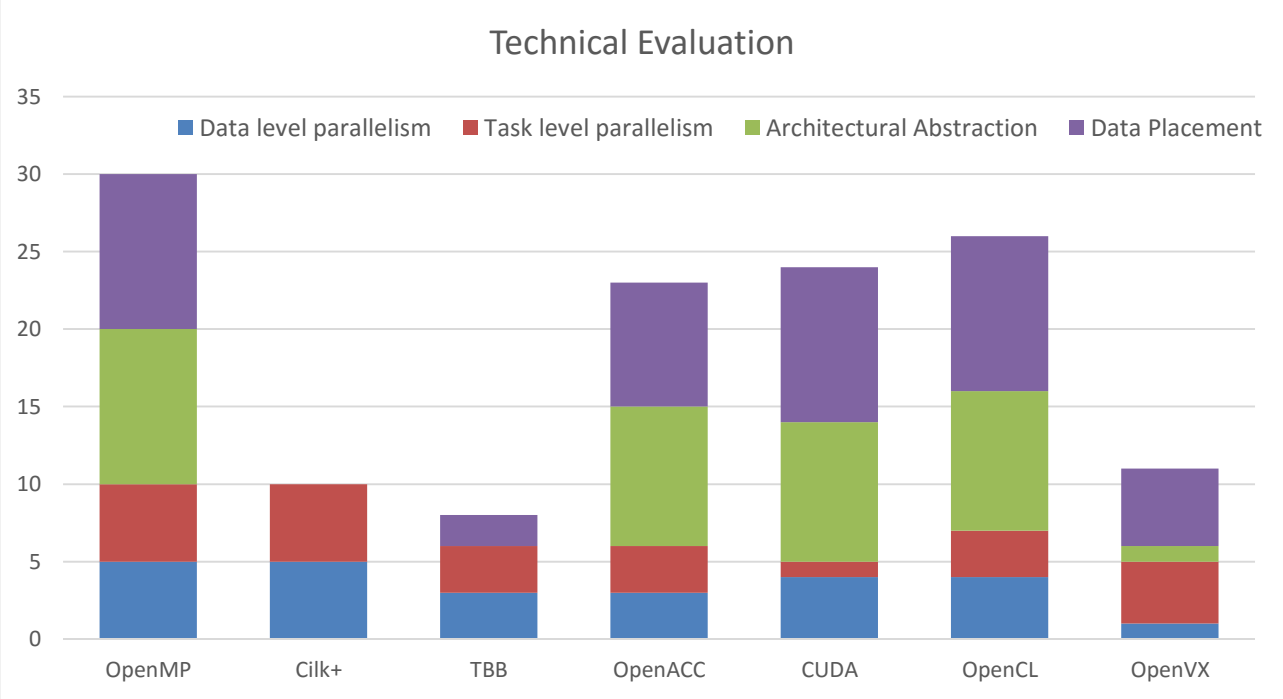

Figure 1: The support offered in the different categories for each programming model.

4.2. Non-functional Evaluation

The technical evaluation focused on the different kinds of parallelism and the constructs available to the programmer for controlling the program behavior. This section instead focuses on the non-functional requirements that must also be fulfilled for the programming model to be suitable for use in the HERCULES project. The non-functional requirements for the programming model for the Hercules project can be summarized as in Table 3:

Item	Description
Industry supported standard	The programming model shall be based on a standard which is supported by industry.
Active user/developer community	The programming model shall have a large community of active users to ease adoption.
Programming simplicity and Ease of Adoption	The programming model shall be simple so that new developers can easily adopt it. It shall also make it easy to adopt the system by permitting reuse of previously written code.
Platform generality and support	The programming model shall be general enough to provide portable code for multiple platforms, and in particular to support the platforms selected as part of WP2.
Application requirements	The programming model shall support the applications outlined as part of WP1.

Table 3: The non-functional requirements.

The selection of parallel programming models in section 3.3 were done in such a way that they reflect the models that are in use today, which are also well supported in industry. Because of this, the two initial

requirements are implicitly fulfilled for each of the four programming models which were selected from the technical evaluation. On top of this, this subsection will make a comparison of the different programming models presented with regard to the remaining three requirements.

4.2.1 Programming simplicity and Ease of Adoption

While it is difficult to quantify the simplicity of a programming model or programming language, we can make some overall observation on the number of things the programmer must explicitly express. In these models, the use of directives or the use of explicitly written kernels is the main difference between OpenMP and OpenACC on the one hand, and CUDA and OpenCL on the other. While in both cases the decision of how to place and access data, how to group threads into independent (i.e., not synchronized) execution blocks and similar considerations are all under the control of the programmer, CUDA or OpenCL provide a much lower-level interface to achieving these goals. Specifically, a programmer writing code for CUDA or OpenCL will have to write a much larger amount of code to explicitly specify operations that are controlled by the compiler in OpenMP and OpenACC. This becomes a large overhead in coding effort while not providing any additional level of control. For example, memory allocation in CUDA is still done via API calls such as *cudaMalloc*, which means that the exact addresses are still out of the programmer's control, yet the programmer is still tasked with performing the actual transfer of data to this address.
It is true that the explicit management of memory has been somewhat relaxed in later versions of CUDA with the introduction of *Unified Memory*. However, Unified Memory relies on software implemented on-demand paging of data to and from the device, and since it is closed-source, it is very difficult to manage from a predictability viewpoint, and a reimplementation of the API does not seem reasonable within the given timeframe. For these reasons, it must be considered fair to conclude that the programming simplicity of the directive-driven models OpenMP and OpenACC is much higher than that of CUDA and OpenCL. On these grounds we give the programming models scores from 1-5, where 5 is minimum-effort, and 1 is maximum-effort (i.e., explicit management of every operation).

The availability of libraries for specific compute domains must also be considered in the analysis, as it severely impacts the productivity of a programmer. For example, while OpenMP has been used for decades in the HPC community, thus producing a vast amount of legacy code, its adoption in the heterogeneous computing is very recent. Consequently, if we focus on GPGPUs, there is a larger amount of legacy code written for OpenCL and CUDA (libraries), which has been available for several years now. In this context, rewriting such libraries (or the parts required) using OpenMP would require a significant effort[6].

As previously, we assign a score of 1-5 in this category, where 1 represents a very small available codebase, and 5 represents a large installed codebase.

Requirement	OpenMP	OpenACC	CUDA	OpenCL
Programming simplicity	5	4	2	1
Availability of large code-base (libraries) for existing accelerators	2	2	5	4

[6] It is important to consider two things. First, DSP-based embedded accelerators such as TI Keystone II provide support for OpenMP, and all the libraries are available as optimized C (or assembly) code. Thus, the problem of the availability of libraries is mostly specific to GPGPUs. Second, an alternative solution would be that of exploring the possibility of linking a CUDA or OpenCL library to a program written in OpenMP or OpenCL.

4.2.2 Platform generality and support

An important requirement for the programming model selected is that it should be used effectively on a large number of potential target platforms.

From this viewpoint, it is clear that both CUDA and OpenACC are closely tied to the GPGPU brand of accelerator offloading. For the case of CUDA, this is not surprising as the programming model was developed by NVIDIA to support general purpose computations on top of their GPUs. In the case of OpenACC, this might not be as clear, since the approach may seem general and similar to OpenMP. However, remember from the technical evaluation that the task level parallelism expressible in OpenACC is limited to the concurrent offloading of several kernels (i.e., task-parallelism is not supported within the accelerator). This is the expected behavior of a system which employs a SIMT semantic, but might not be as relevant for devices where each thread has its own stack and program counters and thus are capable of more irregular patterns. Here, it is clear that the OpenMP support for the target regions is more general, as these offloaded code segments can in themselves contain different forms of parallelism, meaning that it is possible to express a fork-join behavior on the accelerator itself, without resorting to the execution of several kernels from the host side. Likewise, OpenCL supports a large number of accelerators natively.

We give scores of 1-5 for the platform generality, where 1 is specific to a single platform, and 5 is general support for a large number of device types and expressiveness of offloaded code. Furthermore, from WP2, the specific hardware platforms under consideration support a sparse set of programming models, shown below ("**?**" indicates that the information on websites/datasheets is partial/unclear), and provides the score in platform support.

Platform	OpenMP	OpenACC	CUDA	OpenCL
NVIDIA Tegra Parker	**Host**	*No*	**Yes**	*No*
Xilinx Zynq Ultrascale+	**Host**	*No*	*No*	**Yes**
Renesas RCAR H3	**?**	*No*	*No*	**?**
Texas Instruments TDA2 + EVE	**Host**	*No*	*No*	*No*
Infineon Aurix 2	*No*	*No*	*No*	*No*

In many cases, OpenMP is supported on the host processor of the target system, which constitutes a good starting point (OpenMP could be used as a programming *front-end* to wrap another programming model's offload procedure, depending of the availability of tools, discussed later on). The scores for this criterion are assigned based on how many platforms support the programming model in question, again assigned from 1-5. When this is not available, we estimate the effort of developing its support on top of available middleware.

Requirement	OpenMP	OpenACC	CUDA	OpenCL
Generality	5	3	1	5
Support	3	0	1	1

4.2.3 Ease of implementing application requirements

According to D1.1, one of the main application requirements is given by real-time guarantees (deadlines). None of the considered programming models natively provides such support. The second phase of the project will thus be focused on implementing *predictability extensions*. We discuss ease of implementing practically such extensions in Section 4.3 (which mostly depends on the availability of tools). Here, we discuss the extent to which the syntax and semantics of a programming model allow for easily modifying a program for better predictability.

In CUDA and OpenCL, the management of memory transfers, an important potential source of runtime variance, is largely up to the programmer. This means that a CUDA/OpenCL program is typically heavily optimized to achieve better performance. The same applies to the partitioning of parallel units in the program (e.g., loop iterations) into blocks of threads that better match the characteristics of the hardware. This makes it more difficult to implement code transformations aimed at improving predictability, as some of the parallelization strategies already implemented for performance might have to be undone.

In the case of the directive-driven approaches the compiler has a larger amount of freedom for implementing transformations that are advantageous from a predictability viewpoint. The abstract way in which parallelism is specified leaves the possibility for the implementation of compiler passes which can automatically lower the sequential code into parallel regions that are both high-performing and highly predictable – two often conflicting goals that would otherwise place a large workload on the programmer.

We assign scores of 1-5 in this category, where 1 is the highest amount of work required to implement predictable execution models within the compiled binaries, and 5 is a low amount of work.

Requirement	OpenMP	OpenACC	CUDA	OpenCL
Ease of implementing application requirements	4	4	1	2

4.2.4 Final verdict

The results of the individual requirements are put together, and the total score is summed up. The results are presented in Figure 2.

When putting it all together, we see that OpenMP is the overall best choice for programming model, closely followed by OpenCL and OpenACC. However, the scores in the two runner-ups are far more unevenly distributed, i.e., the programming model may fulfil one requirement quite badly, but compensates by excelling at another. In contrast, OpenMP receives high scores on all requirements except *Application requirements*, which is low across the board.

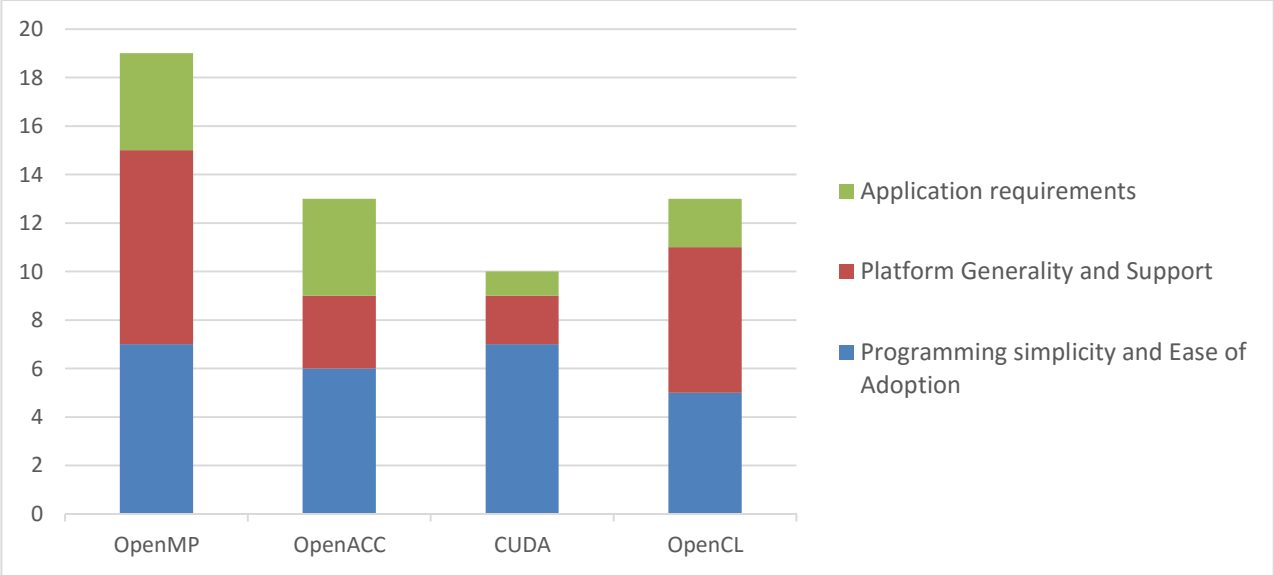

Figure 2: The results of the non-functional evaluation.

4.3. The availability of tools

Predictability extensions can be implemented as extensions to the runtime environments or to the compilation toolchains of a programming model, where the associated framework and SDKs are released in open-source form. When the compiler or runtime library sources are not available, the alternative approach will be that of "wrapping" native framework with additional software layers that implement the desired semantics. For compilation, source-to-source transformations can be applied to modify the input code prior to passing it to the native compilers. For each evaluated programming models, it is important to assess the quality of available tools (SDKs, compilers, runtimes), their maturity and their openness.

It is clear at this point that the directive-based programming models in general, and OpenMP in particular, provide the best solution in terms of ease of programming and ease of implementing predictability extensions. It is also clear that in current platforms OpenMP is mostly supported at the host level, while CUDA and OpenCL are considered for computation offloading to the accelerator.

Given the limited existing support on the targeted platforms, the most general approach for implementing predictability extensions at the compiler level is to use *source-to-source* compilation. This approach reads input from a source language and outputs transformed code in the same language, which allows bypassing the problem of closed-source compilers. Another good fact about *source-to-source* compilation is that it is possible to generate OpenCL/CUDA code from an OpenMP/OpenACC specification, which is very good to combine the ease of use and abstraction of the latter and the native support on the target hardware of the former.

4.3.1 CUDA and OpenCL

For the compilation of CUDA code, there are two main tools available: the closed-source NVIDIA NVCC compiler, which compiles CUDA code into PTX (device-readable machine code) and the Clang 3.8 frontend for the LLVM compiler. LLVM has a backend for producing PTX code, but for compiling to device-native source code, the closed-source assembler from the NVCC tool chain is required. However, it is also possible to ship PTX code, as the CUDA driver is able to JIT-compile PTX to the native ISA of the GPU.

As the NVCC toolchain is closed-source, any approach for compiling CUDA code would have to go through source-to-source compilation or through Clang/LLVM, if additional compiler passes were to be used to implement predictability extensions.

The OpenCL model is more general, and the kernels to be offloaded to the accelerator are generally JIT-compiled for the current hardware. The available tools differ from one architecture to another. There are, however, open-source implementations of OpenCL for certain platforms.

In addition to both of these compiler tool chains, both models also depend heavily on runtime libraries which must be available for the target platform.

4.3.2 OpenMP and OpenACC

The directive-based style of these programming models provides the right level of abstraction to improve productivity and code maintenance and allows for easier development of predictability extensions. However, none of the candidate platforms natively supports OpenMP or OpenACC for computation offloading[7]. For this reason, the main focus of the analysis here has been on evaluating source-to-source (S2S) compilers. We consider six evaluation criteria, which are presented in Table 4.

Requirement ID	Requirement description
Ease of use	The effort required to set up and use the toolchain.
Extensibility	The ease of implementing predictability extensions in the toolchain.
Robustness	The frequency and resilience to crashes or other problems when compiling and executing test applications.
Coverage	The amount of features of the programming model specifications that is actually supported
Documentation	The amount and quality of documentation
Community	The vitality of the supporting community of developers/users and the resources available for support (mailing lists, for a, etc.)

Table 4: The requirements under which the available tools are evaluated.

There are several S2S compilers available for compiling OpenMP 4.0 and OpenACC code into mainly CUDA and OpenCL source code. The following S2S compilers and compiler frameworks were part of the initial evaluation:

- **OMNI (OpenACC) – http://omni-compiler.org**
- **OpenUH (OpenACC) – http://github.com/pumpkin83/OpenUH-OpenACC**
- **OmpSs/Mercurium (OpenMP + StarSs) – http://pm.bsc.es/ompss**
- **ROSE (OpenMP) – http://rosecompiler.org**
- **CETUS (OpenMP) – http://engineering.purdue.edu/Cetus**

The first evaluation is based on publicly available documentation and community mailing lists, while the technical evaluation of ease-of-use, robustness and standards conformance was done by compiling the EPCC OpenACC benchmarks[8] and parts of the GCC OpenMP testsuite[9], respectively. We use the same score range (from 1 to 5) for each of the requirements specified. The results are presented in Figure 3.

[7] As previously mentioned, OpenMP is often supported on the host processor.
[8] https://github.com/EPCCed/epcc-openacc-benchmarks
[9] https://github.com/mickael-guene/gcc/tree/master/libgomp/testsuite

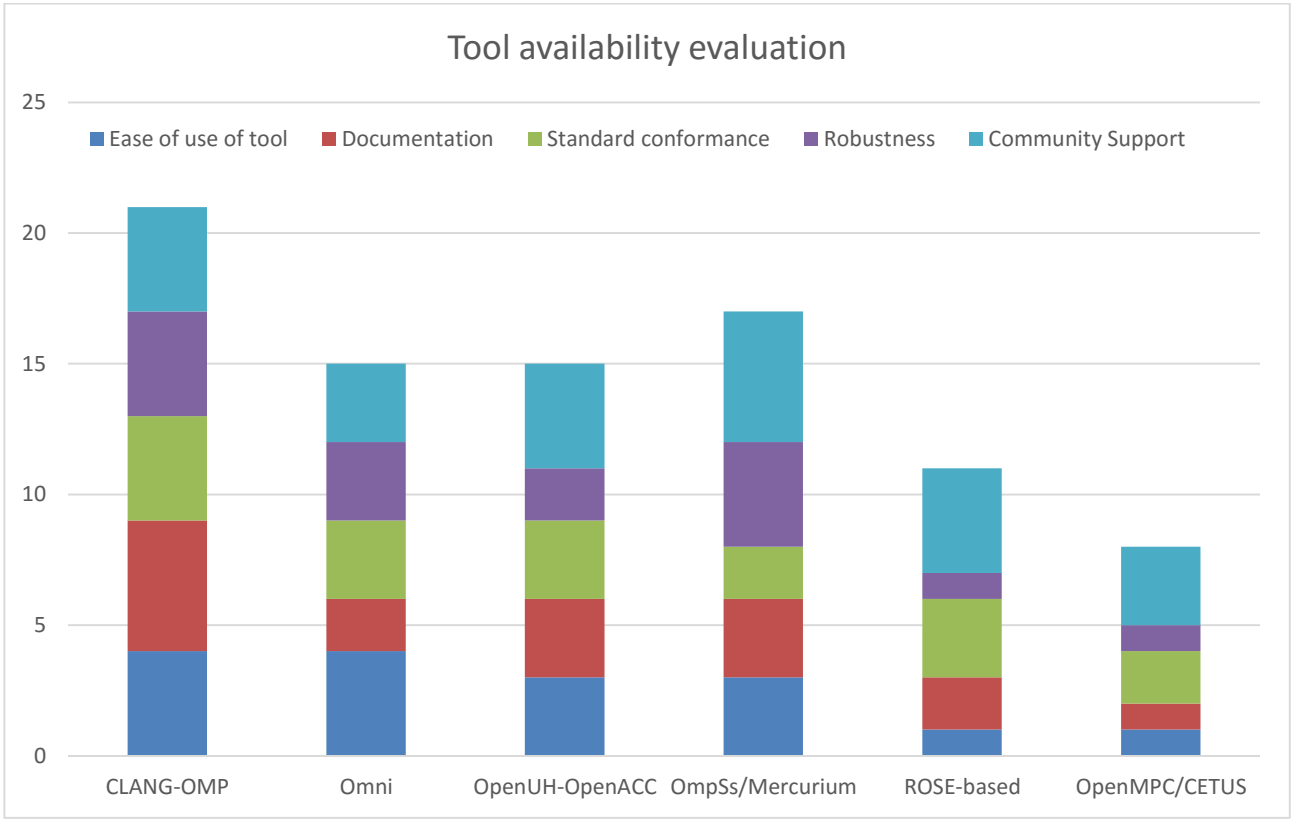

Figure 3: An overview of the different scores given to the compilers available for the directive driven programming models.

Figure 3 also shows the evaluation of the **Clang-OMP** toolchain. This is an OpenMP 4.0 frontend to the LLVM compiler, contributed by, among others, IBM, Intel, AMD, and TI. In addition to providing a good tool for OpenMP compilation, **Clang-OMP** stands out in terms of suitability to the implementation of predictability extensions. These should ideally be implemented at the latest possible compilation stage, to avoid successive optimization to mess-up with their semantics. In our setup, an OpenMP application has been compiled for the NVIDIA platform combining the **Clang-OMP** front-end to the open-source NVPTX back-end for LLVM (provided by NVIDIA). Being LLVM modular in design, large parts of the compiler could be reused when targeting another platform, which would only require to replace the back-end. This is a clear advantage compared to the other S2S compilers, which target a single output language.

In addition, there is one point were **Clang-OMP** really stands out from the others: documentation. While many of the S2S have little or no documentation, the Clang project is extensively documented, both on code level (Doxygen), and design level. In addition to this, Clang has a large community of users and developers. Lastly, the open version management system makes it possible to follow design decisions and code reviews to understand why certain parts of the compiler are implemented as they are.

As a final note, **Clang-OMP** also supports the compilation of CUDA source code, which may provide a good entry point for providing support for legacy code to run on the system. As discussed in Section 4.2, it is not reasonable to expect users to re-implement every piece of legacy code in OpenMP, and thus this could provide an opportunity to natively support CUDA, which is worth exploring.

5. CONCLUSION

This deliverable has outlined the work done during the first six months of the HERCULES project in Task 3.1. In this task, the state of the art for parallel programming models has been surveyed, with the objective of understanding which paradigm constitutes the best starting point to design a solution which combines ease-of-use (programmability, abstraction) with suitable features to design predictability extensions.

Among the various evaluation criteria, particular attention has been put in selecting candidates for which there is a large amount of community support and industrial acceptance (or a standardization process), without neglecting the availability of suitable tools to achieve the technical goals planned for WP3.

We have come to the conclusion that the most promising paradigm is the one based on directives. In particular, **OpenMP4.0** fulfills all the identified requirements more flexibly than OpenACC. In addition, the choice of OpenMP4.0 gives access to mature tools upon which the predictability transformations can be implemented. The most promising tool is **Clang-OMP**, an OpenMP frontend to the widespread LLVM compiler infrastructure.

As NVIDIA is one strong candidate for platform adoption in the HERCULES project, it is important to consider two additional aspects.
First, back-end compilers and runtime systems might not be accessible due to the closed-source policy of NVIDIA. The predictability extensions should be implemented at the latest possible compilation stage to guarantee their effectiveness. **Clang-OMP** provides the best solution in this sense, as LLVM has a PTX (NVIDIA's close-to-assembly intermediate format) backend. Compared to OpenMP-to-CUDA compilation, which requires the generated CUDA program to go through the entire CUDA compilation flow, this approach promises much more efficient implementation of predictability extensions.
Second, NVIDIA provides a lot of libraries for CUDA, where several key computational kernels are optimized for execution on their GPGPUs. Supporting such legacy code when considering OpenMP4.0 as a programming interface becomes fundamental. Also with this respect, **Clang-OMP** constitutes a good solution, as this tools natively supports the compilation of CUDA code, thus making it easier to develop link-time approaches to reuse the existing large legacy codebase.

Note that these conclusions are not limited to NVIDIA platforms, as they do apply to all GPGPU-based heterogeneous systems. For non-NVIDIA products, OpenCL can be considered as a back-end component instead of CUDA. **Clang-OMP** can still be used as an OpenMP frontend to ease application development and to allow for efficient compiler-level predictability extensions, possibly combined to OpenCL backend compilers and runtime systems on the target hardware.