Heterogeneous Computing Architectures
Challenges and Vision
Olivier Terzo, Karim Djemame, Alberto Scionti, Clara Pezuela

Programming and Architecture Models

Publication details
Published online on: 10 Sep 2019

Accessed on: 11 Sep 2023

PLEASE SCROLL DOWN FOR DOCUMENT

Full terms and conditions of use: https://www.routledgehandbooks.com/legal-notices/terms

This Document PDF may be used for research, teaching and private study purposes. Any substantial or systematic reproductions, re-distribution, re-selling, loan or sub-licensing, systematic supply or distribution in any form to anyone is expressly forbidden.

The publisher does not give any warranty express or implied or make any representation that the contents will be complete or accurate or up to date. The publisher shall not be liable for an loss, actions, claims, proceedings, demand or costs or damages whatsoever or howsoever caused arising directly or indirectly in connection with or arising out of the use of this material.
3

Programming and Architecture Models

J. Fumero, C. Kotselidis, F. Zakkak, and M. Papadimitriou
The University of Manchester, Manchester, UK

O. Akrivopoulos, C. Tselios, and N. Kanakis
Spark Works ITC, Derbyshire, UK

K. Doka, I. Konstantinou, I. Mytilinis, and C. Bitsakos
National Technical University of Athens, Athens, Greece

CONTENTS

3.1 Introduction ................................................. 54
3.2 Heterogeneous Programming Models ............................ 56
  3.2.1 Directive-Based Programming Models ....................... 56
    3.2.1.1 OpenACC ........................................ 56
    3.2.1.2 OpenMP 4.0 ..................................... 57
  3.2.2 Explicit Heterogeneous Programming Models ............... 59
    3.2.2.1 CUDA ........................................... 62
  3.2.3 Intermediate Representations for Heterogeneous Programming ................................................. 66
  3.2.4 Hardware Synthesis; FPGA High-Level, Synthesis Tools 68
3.3 Heterogeneous Programming Languages .......................... 69
  3.3.1 Unmanaged Programming Languages ......................... 70
  3.3.2 Managed and Dynamic Programming Languages .......... 70
  3.3.3 Domain Specific Languages ................................ 72
3.4 Heterogeneous Device Selection .................................. 73
  3.4.1 Schedulers .............................................. 73
  3.4.2 Platforms ............................................... 74
  3.4.3 Intelligence ............................................. 76
3.5 Emerging Programming Models and Architectures ............... 78
  3.5.1 Hardware-bound Collaborative Initiatives ................. 78
    3.5.1.1 Khronos Group .................................... 78
    3.5.1.2 OpenCAPI ......................................... 80
  3.5.2 Serverless Frameworks .................................... 81
3.6 Ongoing European Projects ..................................... 83
  3.6.1 ALOHA ............................................... 83
3.1 Introduction

Contemporary computing systems comprise processors of different types and specialized integrated chips. For example, currently, mobile devices contain a Central Processing Unit (CPU) with a set of computing cores, and a Graphics Processing Unit (GPU). Recently, data centers started adding FPGAs into the rack-level and switch-levels to accelerate computation and data transfers while reducing energy consumption.

Figure 3.1 shows an abstract representation of a heterogeneous shared-memory computing system. Heterogeneous shared-memory computing systems are equipped by at least one CPU, typically—at the time of writing—comprising 4 to 8 cores. Each core has its own local cache, and they share a last level of cache (LLC). In addition to the CPU, they are also equipped with a set of other accelerators, normally connected through the PCIe bus. Such accelerators may be GPUs, FPGAs, or other specialized hardware components.

---

1Part of the material presented in this chapter is included in Juan Fumero’s PhD thesis [162] with the permission of the author.
On one hand, GPUs contain hundreds, or even thousands, of computing cores. These cores are considered “light-cores”, since their capabilities are limited compared to those of CPU cores. GPUs organize cores in groups. Each group contains its own cache memory. These groups are called Compute Units (CUs) or Stream Multiprocessors (SM).

FPGAs, on the other hand, are programmable hardware composed of many programmable units and logic blocks. In contrast to CPUs and GPUs, FPGAs do not contain traditional computing cores. Instead, they can be seen as a big programmable core in which logic blocks are wired at runtime to obtain the desired functionality.

Each of these devices, including the CPU, has its own instruction set, memory model and programming models. This combination of different instruction sets, memory models, and programming models renders a system heterogeneous. Ideally, programmers want to make use of as much resources as possible, primarily to increase performance and save energy.

This chapter describes the prevalent programming models and languages used today for programming heterogeneous systems. The rest of this chapter is structured as follows: Section 3.2 discusses the most popular heterogeneous programming models that are currently being used in industry and academia; Section 3.3 shows programming languages that are currently used to develop applications on heterogeneous platforms; Section 3.4 analyses state-of-the-art projects for selecting the most suitable device to run a task within an heterogeneous architecture environment; Section 3.5 describes new and emerging heterogeneous platforms that are currently under research; Section 3.6 shortly
3.2 Heterogeneous Programming Models

This section gives a high-level overview of the most popular and relevant programming models for heterogeneous computing. It first describes the most notable directive-based programming models such as OpenACC and OpenMP. Then, it describes explicit parallel and heterogeneous programming models such as OpenCL and CUDA. Furthermore, it discusses the benefits and the trade-offs of each model.

3.2.1 Directive-Based Programming Models

Directive-based programming is based on the use of annotations in the source code, also known as pragmas. Directives are used to annotate existing sequential source code without the need to alter the original code. This approach enables users to run the program sequentially by simply ignoring the directives, or to use a compatible compiler that is capable of interpreting the directives and possibly create a parallel version of the original sequential code. These new directives essentially inform the compiler about the location of potentially parallel regions of code, how variables are accessed in different parts of the code and how synchronization points should be performed. Directive-based programming models are implemented as a combination of a compiler—often source-to-source [55, 394, 395, 251]—and a runtime system. The compiler is responsible to process the directives and generate appropriate code to either execute code segments in parallel, transfer data, or perform some synchronization. The generated code mainly consists of method invocations to some runtime systems that accompanies the resulting application. These runtime systems are responsible for managing the available resources, transferring the data across different memories, and scheduling the execution of the code [303, 370, 360, 187, 77, 78].

3.2.1.1 OpenACC

Open Multi-Processing (OpenMP) [296] is the dominant standard for programming multi-core systems using the directive-based approach. Following the successful case of OpenMP, the OpenACC [293] (Open Accelerators) standard was proposed at the Super-Computing conference in 2011. This new standard allows programmers to implement applications for running on heterogeneous architectures, such as multi-core CPUs and GPUs, using the directive-based approach.
To use OpenACC, developers annotate their code using the `#pragma` directives in C code. Then, a pre-processor (a computer program that performs some evaluation, substitution, and code replacements before the actual program compilation) translates the user’s directives to runtime calls that identify a code section as a parallel region, and prepares the environment for running on a heterogeneous system.

Listing 1 shows a simple example of a vector addition written in C and annotated with OpenACC directives. Computing kernels are annotated with the `acc kernels` directive. In OpenACC, developers also need to specify which arrays and variables need to be copied to the parallel region (i.e., are read by the kernel) and which need to be copied out (i.e., are written by the kernel). In this example, the arrays `a` and `b` are marked as `copyin` since they are read by the annotated for loop. Since the statement within the `for` loop writes the result of each iteration in the array `c`, the latter is marked as `copyout`. This information is used by the compiler and the runtime system to generate and execute only the necessary copies, to and from the target device (e.g., a GPU). Note that OpenACC requires setting the range of the array that a kernel is accessing. This essentially enables different kernels to access different parts of the same array with the need of synchronization as long as the two parts do not overlap.

Listing 1—Sketch of code that shows a vector addition in OpenACC

```c
#pragma acc kernels copyin(a[0:n],b[0:n]), copyout(c[0:n])
for(i=0; i<n; i++) {
    c[i] = a[i] + b[i];
}
```

3.2.1.2 OpenMP 4.0

Following the success of OpenACC for heterogeneous programming on GPUs, OpenMP also included support for heterogeneous devices since the standard OpenMP 4.0, one of the latest versions of the standard at the time of writing this book [296]. This version of the standard includes a set of new directives to program GPUs as well as multi-core CPUs.

Listing 2 shows an example of how to perform vector addition to target GPUs with OpenMP 4.0. The first observation is that OpenMP for GPU programming is much more verbose than OpenACC. This is because OpenMP is completely explicit about how to use the GPU from the program directive level. OpenMP 4.0 also introduced new terminology for extending the existing OpenMP for multi-cores and shared computing memory systems to be able to express computation for GPUs.

Breaking down Listing 2, line 1 sets a new OpenMP region that targets an accelerator through the `target` directive. An accelerator could be any device, such as a CPU or a GPU. Then, it specifies how the data should be seen by
the host (main OpenMP thread on the CPU). By default, data is owned by
the target device. Therefore, to copy the data from the host to the device,
OpenMP uses the directive `map` with the parameter `to`. On the contrary, to
perform a copy out, we use the `map-from` clause. These two clauses specify
how to move the data from the host to the device and back.

The second `pragma` in Listing 2 expresses how to perform computation
on the target device (line 3). The `target` directive indicates that execution
needs to be relocated (e.g., moved to a GPU). The `teams` directive tells the
compiler that parallel code should be handled by at least one team (group of
thread working together on the target device). The `distribute` directive tells
the compiler that the `i` induction variable in the loop should be distributed
across the teams. The `parallel` directive activates all the threads inside a
team. These threads should run in parallel. The `for` clause tells the compiler
that the work is shared across all the teams. Finally, the `schedule(static, 1)`
tells the scheduler that blocks of threads use contiguous memory, to increase
memory coalescing.

As seen, OpenMP can get very verbose, giving fine control of the execution
to the programmer. Furthermore, programmers have control of how data are
shared, allocated, and transferred, as well as how execution is performed and
scheduled on the GPU.

**Listing 2**—Sketch of code that shows a vector addition in OpenMP 4.5

```c
#pragma omp target data map(to:a[:n],b[:n]) map(from:c[:n])
{
  #pragma omp target teams distribute parallel for 
    schedule(static, 1)
  for(i=0; i<n; i++) {
    c[i] = a[i] + b[i];
  }
}
```

**Considerations:**

Both OpenMP and OpenACC work for C/C++ and Fortran programs, and
they require parallel programming expertise as well as good understanding
about the target architecture to achieve performance. To build heterogeneous
applications with OpenMP and OpenACC, programmers need a good under-
standing of the data flow in their application. They also need to understand
the different barriers (explicit and implicit), how to perform synchronization,
how to share variables, how data are moved and when the data are moved
from CPU to GPU and vice versa, and how execution is mapped to the tar-
target architecture. Furthermore, these two models, although targeting hetero-
genous architectures, and more precisely GPUs, are not equivalent. OpenMP
extended the standard that targets multi-core CPUs, meanwhile OpenACC
was designed from scratch having GPU computation in mind.
Due to the simpler directives of OpenACC, some programmers rewrite OpenMP programs that target multi-core CPUs using the OpenACC directives to port their applications to heterogeneous devices with the least possible effort. However, considering the overheads of the data transfers to and from the target device, along with the limitations that GPU cores impose, blindly porting OpenMP annotated applications targeting multi-core CPUs to OpenACC may not yield performance improvements. On the contrary, it might even end up giving a worse performance than the sequential execution.

### 3.2.2 Explicit Heterogeneous Programming Models

This section describes the most common explicit heterogeneous programming models available at the moment, both from the industry and academia. These programming models are relatively new compared to the directive-based ones, such as OpenMP.

OpenCL (Open Computing Language) [294] is a standard for heterogeneous computing released in 2008 by the Khronos Group. The OpenCL standard is composed of four main parts:

- **Platform model:** it defines the host and devices model in which OpenCL programs are organized.

- **Execution model:** it defines how programs are executed on the target device and how the host coordinates the execution on the target device. It also defines OpenCL kernels as functions to be executed on the devices and how they are synchronized.

- **Programming model:** it defines the data and tasks programming models as well as how synchronization is performed between the accelerator and the host.

- **Memory model:** it defines the OpenCL memory hierarchy as well as the OpenCL memory objects.

The rest of the section explains each of these parts in more detail.

#### Platform Model

Figure 3.2 shows a representation of the OpenCL platform model. OpenCL defines a host (e.g., the main CPU) that is connected to a set of devices within a platform. A host can have multiple devices. Each device (e.g., a GPU) contains a set of compute units (group of execution units). In its turn, each compute unit contains a set of processing elements, that are, in fact, the computing elements in which code will be executed.

The number of computing units (CU) and processing elements (PE) within a device depend on the target device. For instance, if the target device is a CPU, a CPU contains multiple cores. Therefore each core is a CU. In its turn,
each core contains a set of SIMD (Single Instruction Multiple Data) units. Those units are the PE within a CU.

In the case of GPUs, each CU comprises a set of stream multiprocessors (using the NVIDIA GPU terminology) that contains a set of physical cores, instruction schedulers and a set of functional units. Each small core is mapped to a PE, where instructions will be executed.

Execution Model

The execution model is composed of two parts: the host program and the kernel functions.

**Host Program:** It defines the sequence of instructions that orchestrate the parallel execution on the target device. The OpenCL standard provides a set of runtime utilities that facilitate the coordination between the host and the device. A typical OpenCL program is composed of the following operations:

1. Query the OpenCL platforms available.
2. Query all devices within an OpenCL platform that are available.
3. Select a platform and a target device.
4. Create an OpenCL context (an abstract entity in which OpenCL can create commands associated with a device and send those commands to the device).
5. Create a command queue to send requests, such as read and write buffers and launch a kernel on the device.
6. Allocate buffers on the target device.
7. Copy the data from the main host to the target device.
8. Set the arguments to the kernel.
9. Launch a kernel on the device.
10. Obtain the result back through a copy back (read operation) from the device to the host.
Kernel functions: Kernels are the functions that are executed on a target device. OpenCL defines a set of new modifiers and tokens that are used as extensions of C99 functions. Listing 3 shows an example of how to perform vector addition in OpenCL. OpenCL adds the modifiers \textit{kernel} to indicate that this function is the main kernel and \textit{global} to indicate that data is located on the global memory of the target device.

\begin{Verbatim}
Listing 3—Vector addition in OpenCL

\begin{verbatim}
1 kernel vectorAdd(global float*a, global float*b, global float*c) {
2     int idx = get_global_id(0);
3     c[idx] = a[idx] + b[idx];
4 }
\end{verbatim}
\end{Verbatim}

Note that there is no loop that iterates over the data for the code shown. This is due to the fact that OpenCL maps an OpenCL kernel into a N-dimensional index space. Figure 3.3 shows an example of a 2D kernel representation in OpenCL. OpenCL can execute kernels of 1D, 2D, and 3D. Following the example in Figure 3.3, the index space is organized in a 2D block. Each block within a work-group contains a set of work-items (threads). Each work-item indexes data from the input data set using its coordinates \((x, y)\)—if it is a 2D block. The OpenCL execution model provides built-in support for accessing and locating work-items, either using the global identifiers (for instance, the \texttt{get\_global\_id(0)} accesses a work-item in the first dimension), or using the functions to locate a work-item within a work-group (for instance, as shown in the right side of Figure 3.3). One of the key aspects of this organization is that threads in different work-groups cannot share memory or add barriers to wait for other groups of threads in another work-group.
Programming Model

A host program en-queues the data and the function to be executed on the target device. An OpenCL kernel follows the SIMT (Single Instruction Multiple Thread) model, in which each thread maps an item from the input data sets and performs the computation. The host issues the kernel and a bunch of threads to be executed on the target device. Then, this kernel is first submitted to the target device and executed.

If the target device is a GPU, it organizes the threads in blocks of 32 threads (or 64 threads, depending on the vendor and the GPU architecture) called warp or wave-fronts. Those threads are issued to a specific CU on the GPU. Each CU (or SM using the NVIDIA terminology) also contains a set of thread schedulers that issues an instruction per cycle. Those threads are finally located on the GPU physical cores. Threads within the same CU can share memory.

Memory Model

The OpenCL standard also defines the memory hierarchy and memory objects to be used. Figure 3.4 shows a representation of the memory hierarchy in OpenCL. The bottom shows the host memory, the main memory on the CPU. The top of the figure shows a compute device (e.g., a GPU) attached to a host. The target device also contains a region for global memory, in which input/output buffers are copied to/from. All work-groups can read and write into the device global memory. Each work-group contains its own local memory (shared memory using the NVIDIA terminology). This memory is much faster than the global memory but very limited in space. All work-items within a work-group can read and write into its own local memory. In its turn, work-items can access private memory (register files of the target device). Data in private registers is much faster to access than data in local memory. OpenCL programmers have fully control over all of these memory regions.

3.2.2.1 CUDA

Common Unified Device Architecture (CUDA) [118] is an explicit parallel programming framework and a parallel programming language that extends C/C++ programs to run on NVIDIA GPUs. CUDA shares many principles, terminology, and ideas with OpenCL. In fact, OpenCL was primarily inspired from the CUDA programming language. The difference is that CUDA is slightly simpler and more optimized due to the fact that it can only execute on NVIDIA hardware. This section shows an overview of the CUDA architecture and programming model.

CUDA Architecture

Figure 3.5 shows a representation of the CUDA architecture. The bottom of the stack shows a GPU with CUDA support (which are the majority of the
NVIDIA GPUs). Then, a driver at the operating system (OS) level to compile and run CUDA programs, and the CUDA driver, which performs low-level tasks and generates an intermediate representation called PTX (Parallel Thread Execution). The CUDA driver also provides an Applications Programming Interface (API) in which programmers can directly make use for developing applications for GPUs. This API is very low-level and it is very similar, in concept, to the OpenCL API and programming model.

The CUDA architecture also provides a CUDA runtime (CUDART) that facilities programming through language integration, compilation, and linking of user kernels. The CUDA runtime automatically performs tasks such as device initialization, device context initialization, loading the corresponding low-level modules, etc.

CUDA also exposes to developers a set of libraries, such as CUDA Trust for parallel programming, cuBLAS for scientific computing, and cuDNN (CUDA accelerated version for Deep Neural Networks) among many others.

**CUDA Programming Overview**

CUDA programs are executed on systems with GPUs attached via PCI-express to the main CPU. In the CUDA terminology, as well as in OpenCL, a host means the main director where the GPU execution is organized. A device is the target GPU in which a set of kernels is launched and executed for acceleration.
FIGURE 3.5: Overview of the CUDA architecture.

Figure 3.6 shows an overview of a CUDA program. A CUDA application is composed of two parts: the host code, which is the code to be executed on the main CPU, and the device code, which is the function to be executed and accelerated on the GPU. In a similar way to OpenCL, CUDA programs make use of streams to communicate commands between the host and devices. This architecture is practically identical to an OpenCL program’s view explained in the previous section.

The main CPU contains its own physical memory. Besides that, the GPU also contains its own physical memory. The typical workflow for a CUDA application is as follows:

1. Allocate the host buffers.
2. Allocate the device buffers.
3. The CPU issues a data transfer from the host to the device (H→D) to copy the data on the device.
4. The CPU issues to launch a CUDA kernel. It sets the number of CUDA threads.
5. The CPU issues a data transfer from the device to the host (D→H) to copy the results to the main CPU.

CUDA Programming

Given the CUDA architecture, developers can write applications using three different levels of abstraction: the CUDA library, the CUDA runtime API, and the CUDA driver API. Each level of abstraction within the CUDA architecture provides an API for developing GPU applications. Many programmers prefer using libraries, mostly at the beginning of development. Additionally, the CUDA runtime API extends the C99 language (as OpenCL) with new
modifiers and tokens that express parallelism for CUDA. A CUDA driver API provides more control and tuning of CUDA applications within the physical architecture.

Listing 4 shows an example of using CUDA for vector addition. This kernel (CUDA function to be executed on the GPU) is very similar to the vector addition in OpenCL shown in Listing 3. The new language token `__global__` tells the CUDA compiler that the following function corresponds to the GPU code. The code inside the function corresponds to the sequential C code for computing the vector addition for a single thread. CUDA, as well as OpenCL, uses the thread-id to index the data in an index-space range configuration. In this example, the kernel obtains its thread-id by querying the index within a 1D configuration. To query the thread-id, CUDA provides a set of high-level built-ins such as `threadIdx.x`. Then, the kernel uses this thread-id to obtain the data-item at that given location and performs the computation.

```
1. __global__ void addgpu(float *a, float *b, float *c) {
2.     int tid = blockIdx.x;
3.     c[tid] = a[tid] * b[tid];
4. }
```

Listing 5 shows the host-code to execute the vector addition using the CUDA runtime API. It first allocates the data on the GPU by using the function `cudaMalloc`. Then, it copies the data from the host to the device using the `cudaMemcpy` function. Then, the host launches the CUDA kernel using a C extension to indicate to the CUDA compiler to execute a CUDA kernel. The CUDA kernel uses two parameters between the symbols `<<,>>>` to indicate the number of threads to run and the number of blocks to run. CUDA threads, as well as in OpenCL, are organized in blocks. Therefore, the user controls the amount of threads per block and the number of blocks to use. Finally, once the CUDA kernel has been executed, the host issues a copy back from the device to the host to obtain the result (line 12).
The CUDA example just shown corresponds to the use of the CUDA runtime API. CUDART takes responsibility for querying all devices available, initializing them, creating the context in which sends commands, creating the stream (objects to send commands to the GPU), compiling the CUDA kernel to PTX, and running it. GPU and CUDA experts could also use the CUDA driver API, in which all these operations are fully controlled by the programmer. Normally, CUDA driver is also used by optimizing compilers and libraries, in which CUDA code can be auto-generated.

**CUDA or OpenCL?**

CUDA and OpenCL are very similar. Both use the same programming model and the same core concepts. Furthermore, both require deep-level expertise from the programmers’ perspective. A good understanding of the GPU architecture in combination with the programming model are crucial to understand and implement applications with such models.

However, CUDA is much simpler to start with because it simplifies its model to the NVIDIA GPUs. Furthermore, CUDA is said to obtain much higher performance due to the fact that it is specially optimized for NVIDIA GPUs. Additionally, CUDA offers different levels of abstraction, such as high-level libraries, CUDA syntax, and the CUDA driver. This makes CUDA programming more flexible and suitable for many industries and academics from many different fields.

Nonetheless, OpenCL is more portable and can execute code on any OpenCL-compatible device that currently includes CPUs, GPUs from various vendors, and FPGAs (e.g., from Xilinx and Intel Altera).

### 3.2.3 Intermediate Representations for Heterogeneous Programming

An Intermediate Representation (IR) is a low-level language that is used by compilers and tools to optimize the code. Compilers first translate high-level
source code to an intermediate representation and then optimize this IR performing several passes that implement a number of different analyses and optimizations. In the end, the resulting IR is translated into efficient machine code. This section presents a number of IRs specifically designed for heterogeneous architectures.

**HSA**

Heterogeneous System Architecture [158] is a specification developed by the HSA foundation for cross-vendor architectures that specifies the design of a computer platform that integrates CPU and GPUs. This specification assumes a shared memory system that connects all the computing devices. HSA is optimized for reducing the latency of data transfers across devices over the shared memory system.

The HSA foundation also defines an intermediate language named HSAIL (HSA Intermediate Language) [159]. The novelty of this IR is that it includes support for exceptions, virtual functions, and system calls. Therefore, GPUs that support HSAIL enable applications running on them to benefit from these higher-level features.

Taking the ideas developed in the HSA and HSAIL, AMD developed ROCm [160], a heterogeneous computing platform that integrates CPU and GPU management into a single Linux driver. ROCm is directly programmed using C and C++. The novelty of ROCm is that there is a single CPU and GPU compiler that targets both types of devices for C++, OpenCL, and CUDA programs.

**CUDA PTX**

CUDA Parallel Thread Execution (PTX) is an intermediate representation developed by NVIDIA for CUDA programs. CUDA, as shown in the previous section, is a language based on C99 for expressing parallelism. Then, a CUDA compiler (e.g., nvcc), translates the CUDA code to the PTX IR. At runtime, an NVIDIA driver finally compiles the PTX IR into efficient GPU code. CUDA PTX, as well as CUDA, is only available for NVIDIA GPUs.

Listing 6 shows an example of CUDA PTX for the vector addition computation. This IR is similar to an assembly code for GPUs. First, it obtains the parameters that are passed as arguments (ld.param). Then, it obtains the respective addresses from global memory using the cvta.to.global instruction, obtains the thread-id, and sums the elements. The result is finally stored using the st.global instruction.

**SPIR**

The Standard Portable Intermediate Representation (SPIR) [231] is an LLVM IR-based binary intermediate language for compute graphics and compute kernels. SPIR was originally designed by the Khronos Group in order to be
Listing 6—Example of CUDA PTX for a vector addition

1. ld.param.u64 %rd1, [ZShelloPcPi_param_0];
2. ld.param.u64 %rd2, [ZShelloPcPi_param_1];
3. cvta.to.global.u64 %rd3, %rd1;
4. cvta.to.global.u64 %rd4, %rd2;
5. mov.u32 %r1, %tid.x;
6. cvt.u64.u32 %rd5, %r1;
7. mul.wide.u32 %rd6, %r1, 4;
8. add.s64 %rd7, %rd4, %rd6;
9. ld.global.u32 %r2, [%rd7];
10. add.s32 %r3, %r2, %r2;
11. add.s64 %rd8, %rd3, %rd5;
12. st.global.u8 [%rd8], %r3;
13. ret;

used in combination with OpenCL kernels. Instead of directly compiling the source code of OpenCL kernels to machine code, compilers can produce code in the SPIR binary format. The main reason to represent SPIR in binary format is to circumvent possible license issues because of the source code distribution of the compute kernels on different devices. SPIR is currently used by optimizing compilers and libraries to distribute heterogeneous code across multiple heterogeneous architectures. Section 3.5.1.1 provides more details about SPIR.

3.2.4 Hardware Synthesis: FPGA High-Level, Synthesis Tools

High-Level Synthesis (HLS) tools enable the programming of FPGAs in a higher level of abstraction with high-level languages (HLLs). Raising the abstraction level and reducing the long design cycles, makes FPGAs more accessible and easier to adopt in modern computing systems, like data centers. The traditional approach for programming an FPGA was initially limited to hardware description languages (HDLs), such as VHDL and Verilog. However, using HDLs is a tedious process, and it requires high expertise and deep understanding of the underlying hardware. This results also in extensive periods of programming and long design cycles.

Figure 3.7 provides a high-level overview of the workflow in most HLS tools. User starts by providing a design specification written in C/C++ or SystemC along with a number of directives (also called pragmas). These directives allow users to provide hints to HLS compilers; for instance, a program may contain for loops without any dependencies that can be unrolled or pipelined. These directives guide the compiler to map a number of optimizations, such as loop unrolling, loop pipelining, or memory partitioning into the hardware. Then the initial program specification is compiled, and a formal model is produced. The information acquired during the compilation process allows the HLS to
define the type of resources needed (memory blocks, LUTs, buses, etc.), and then to schedule these operations into cycles. When all these processes come to an end, and the design constraints (e.g., timing, power, and area) are met, the RTL generator provides a synthesizable model of the hardware design for the input program. Finally, logic synthesis generates a hardware configuration file which contains a hardware implementation of the initial design specification.

There is a large selection of HLS tools. The most popular and widely adopted ones at the time of writing are the Vivado HLS [31] by Xilinx, and Intel FPGA HLS [28]. Both take as input C/C++ programs and parallel code, such as OpenCL, and produce the corresponding VHDL/Verilog programs along with the machine code in binary form for the FPGA device. Other examples of HLS tools are the LegUp [87], MaxCompiler [29], Bambu HLS [312], Bluespec [26], and Catapult C [27].

3.3 Heterogeneous Programming Languages

This section describes how modern programming languages integrate with the models described previously to enable heterogeneous execution. We classify those languages in (i) unmanaged programming languages, such as C/C++ and Fortran, (ii) managed languages, such as Java, Python, Javascript, and Ruby, and (iii) high-level domain-specific programming languages, such as R.
3.3.1 Unmanaged Programming Languages

Unmanaged programming languages is a term that refers to those languages in which memory is directly handled by the programmers. These languages tend to be lower-level compared to those that automatically manage memory (e.g., Java). C/C++ and Fortran are the most common unmanaged programming languages used for programming heterogeneous architectures such as GPUs, CPUs and FPGAs. In fact, many standards for heterogeneous programming are designed for C and Fortran. Furthermore, the majority of applications developed for High-Performance Computing (HPC), such as physics simulation, particle physics, chemistry, biology, weather forecast, and more recently, big data and machine learning, tend to use low-level and high-performance programming languages.

To program heterogeneous devices from C/C++ and Fortran, developers can use both directive-based, as well as explicit heterogeneous programming models. Both OpenACC and OpenMP include all directives for both languages in the standard definition. At the same time, OpenCL and CUDA extend the C99 standard with new modifiers and tokens in the language to identify parallelism.

The advantage of programming in C or Fortran using any of the standards described in the previous sections is that all rules, syntax, execution models, and memory models are well defined for the target language. On the contrary, the disadvantage of using unmanaged languages is that they are low-level and require higher expertise from the programmers. Higher-level programming languages allow for faster prototyping, and thus quicker development cycles, while requiring less expertise from the programmers, at the cost of reduced control over the execution of the program.

3.3.2 Managed and Dynamic Programming Languages

Managed programming languages is a term that refers to those languages whose memory is automatically managed by a runtime system provided by the language implementation. Some examples are languages like Java and C#, in which objects are allocated in a managed heap area. The implementation of those languages includes a language Virtual Machine (VM) that executes the applications and automatically manages memory, including object de-allocation. This removes the burden of memory management from the programmer, along with the potential accompanying bugs, allowing them to focus on other aspects of their application.

Furthermore, some managed programming languages are un-typed, like Python, Ruby and Javascript. From the programmer’s perspective, this makes application development much easier and faster, allowing programmers to focus on what they want to solve instead of how to efficiently implement it. Those programming languages rely on an efficient language VM, an interpreter and a compiler to execute efficiently the input programs.
Table 3.1 shows a ranking of the most popular programming languages in October 2018 using three different sources: TIOBE Index\textsuperscript{2}, IEEE\textsuperscript{3}, and PYPL\textsuperscript{4}. Each source reference uses a different search criteria for ordering the programming languages by its popularity, such as Github projects, Google searches, and academic papers. What all have in common is that the majority of the languages in the top 10 are managed-programming languages, such as Java, C#, Python, and Javascript.

The question that arises after looking at the data from Table 3.1 is how to use accelerators that follow the programming models listed in the previous section from managed languages. There are currently two main techniques to program GPUs and FPGAs from managed languages:

**Using external libraries:** accelerated-libraries are implemented directly in C/C++ and Fortran. They provide a set of common operations, normally operations for array and matrix computation that are exposed to the managed language through a language interface from the C level. For example, in the case of Java, they can be exposed using the Java Native Interface (JNI), in which operations are implemented using C code and called from the Java side.

**Using a wrapper:** the use of accelerators is directly programmed from the high-level programming languages using the same set of standards API calls. This provides fine-tune control over the applications to be executed on the accelerators, but contradicts with the nature of the managed programming languages since it requires low-level understanding.

\begin{table}
\centering
\begin{tabular}{|c|c|c|c|}
\hline
Language Rank & TIOBE & IEEE & PYPL \\
\hline
1 & Java & Python & Python \\
2 & C & C++ & Java \\
3 & C++ & Java & Javascript \\
4 & Python & C & C# \\
5 & VBasic & C# & PHP \\
6 & C# & PHP & C/C++ \\
7 & PHP & R & R \\
8 & Javascript & Javascript & Objective-C \\
9 & SQL & Go & Swift \\
10 & Swift & Assembly & Matlab \\
\hline
\end{tabular}
\end{table}
Examples of external libraries are ArrayFire [264] and PyTorch\(^5\). ArrayFire is a library for programming parallel CPUs and GPUs in C, C++, Java, and Python. It is designed for matrix computation using CUDA and OpenCL underneath, and it contains numerous highly-optimized-function GPUs for signal processing, statistics, and image processing. PyTorch is a Python library for deep learning on GPUs. It exposes a set of high-level operations that are efficiently implemented in CUDA.

Some examples of wrappers are JOCL [217], for programming OpenCL within Java, and JCUDA [386], for programming CUDA within Java. Both expose a set of native calls to be invoked from Java that matches the OpenCL and CUDA definitions respectively. Kernels are expressed as a Java string that is then passed to the compiled and executed via JNI calls.

**Challenges**

Although high-level and managed-programming languages ease the development process, the higher the abstraction the more difficult it becomes to optimize the code for the target platform. For example, in Java, all arrays are copied from Java’s side to the JNI’s side. This means that JNI makes an extra copy for all arrays, and therefore, the performance of applications will be different compared to those implemented in unmanaged languages like C. Programming heterogeneous devices from managed languages increases the complexity of efficiently managing memory and types. For example, neither OpenCL nor CUDA support objects, while in object-oriented high-level programming languages everything is an object. Therefore, extra effort is required by the runtime system to convert the data from their high-level representation to a low-level representation and vice versa. This process is called *marshalling/unmarshalling* and can be time-consuming depending on the objects at hand.

Furthermore, the semantics of high-level programming languages do not always match those of the heterogeneous programming models. For example, the OpenCL standard does not clarify what happens with runtime exceptions, such as a division by zero. In the case of Java, the JVM is required to throw an *ArithmeticException*, but hardware exceptions are not currently supported in CUDA or OpenCL.

### 3.3.3 Domain Specific Languages

Domain Specific Languages (DSLs) are languages designed for a specific purpose. For example R [205] is a programming language specifically designed for statistics. R, as shown in Table 3.1, is one of the most popular, and has been increasing its popularity during the last three years. Many scientists use R to process big data, provide statistics, and even predict future events using machine learning due the fact that it provides an enormous quantity of external data.

\(^5\)https://pytorch.org
Heterogeneous Device Selection

modules. These external modules are normally written in lower programming languages such as C and Fortran for better performance.

In DSLs, heterogeneous devices can be programmed using external libraries and wrappers, as in managed programming languages. In the case of DSLs, however, the challenges of programming heterogeneous systems are even more complex. DSLs are mainly used because of their simplicity and their specialization in a specific domain. Introducing wrappers raises the complexity of DSLs and breaks their specialization. As a result, programmers are required to mix and understand different programming and architecture models, totally unrelated to their domain.

3.4 Heterogeneous Device Selection

This section describes how developers can select the most suitable device for executing their code. More specifically, it presents and discusses various techniques for device selection such as: offline source code analysis, machine learning models, and profile-guided selection.

3.4.1 Schedulers

In typical computing environments, such as Cloud and HPC datacenters, the infrastructure is being managed by software systems called resource schedulers. Schedulers are responsible for allocating tasks to the underlying infrastructure that can consist of clusters of computing nodes, storage, networking, etc. Resource schedulers typically operate in a master-slave manner, where a central machine (called the “master”) is used to consolidate and manage the amount of different workers (i.e., slaves) by utilizing a centralized resource registry. The workers use different approaches to register themselves with the master, such as heartbeat messages, etc, following the same architecture with typical legacy “batch” schedulers such as Condor [362] and PBS-Torque [196, 89].

Modern schedulers are starting to embrace the heterogeneity of resources encountered in Cloud infrastructures. The scheduling techniques can be categorized in workload partitioning methods (static vs. dynamic), subtask-based scheduling, pipe-lining, and MapReduce-based [276].

Apache Mesos [198] uses a two-level scheduling mechanism where resource offers are made to frameworks (apps like web servers, map-reduce programs, NoSQL databases, etc. that run on top of Mesos). The Mesos master node decides how many resources to offer each framework, while each framework determines the resources it accepts and what application to execute on those resources. Apache YARN [372] is the de-facto scheduler used in modern deployments of the Hadoop ecosystem software. It also utilizes a negotiation framework where resources are being requested and provided upon requests from different applications. Both schedulers offer the possibility to “label”
different execution nodes with user defined-labels, leaving the user to handle heterogeneity.

ORiON [391] and IReS [133] schedulers both can schedule heterogeneous workloads on big data clusters. ORiON can adaptively decide the appropriate framework along with the cluster configuration in terms of software and hardware resources to execute an incoming analytics job. It is based on a combination of a decision-tree-like machine learning process for resource prediction and an integer linear problem formulation for resource optimization. IReS can identify and materialize the optimal sequence of analytics engines required to execute a data-processing pipeline in an abstract execution workflow given by the user by utilizing a dynamic-programming technique. Although both schedulers can work with heterogeneous software systems, the hardware heterogeneity is not exploited.

TetriSched [369] also schedules heterogeneous resources by forming a Mixed Integer Linear Programming problem, nevertheless it is not aware of any task relations in the submitted workload (i.e., it is DAG-oblivious), similar to ORiON.

In [378] the authors present a scheduling methodology for tasks that can be executed on both CPU and GPU devices. They evaluate their methodology with different complex algorithms (i.e., kernels) such as bfs, BlackScholes, Dotproduct, and QuasirandomG, and they show that they can predict and deploy their kernels to the most appropriate engine. Their approach is based on predictive modeling, i.e., on identifying the important kernel features that affect the performance (i.e., execution time). They utilize these features to build a machine learning model based on support vector machines and they use it to predict the execution time upon workload arrival. Nevertheless, they are also DAG-oblivious and they do not explore different performance metrics.

A similar approach is also followed by [377], where a machine learning model is built to predict execution time based on carefully selected features. Their main differentiation compared to [378] is that they study the effect of concurrently executing kernels on the same device (i.e., merging) to identify whether a speedup can be achieved or not.

### 3.4.2 Platforms

Over the last years many distributed and parallel data analytics frameworks have emerged. These systems process in parallel large amounts of data in a batch or streaming manner. In this section, we investigate to what extent the heterogeneity of modern datacenters is exploited to improve application performance.

GFlink [103] is a distributed stream-processing platform that extends Apache Flink[93]. Its key-feature is that, apart from CPU processors, it can also execute tasks on top of GPU accelerators. To achieve that, it builds upon the master-slave architecture of Flink, and introduces processes (GPUMangers) that are responsible for the GPU management. Execution is coordinated
Heterogeneous Device Selection

by a single master which schedules tasks to workers either for CPU or GPU execution. The scheduling scheme is locality-aware and manages to avoid unnecessary communication, while it achieves load-balancing among GPUs. Nevertheless, details on the scheduling algorithm are not provided.

GFlink runs in the Java Virtual Machine (JVM). The standard way of communication between JVM and the GPU devices is to serialize memory objects that live in the JVM heap, and transfer data over the PCIe bus. However, as this process incurs high serialization-deserialization cost, GFlink opts for a more efficient memory management: Users can define custom structures that live off-heap and operate directly on them. This way, object serialization is avoided. Moreover, the master can instruct the GPU devices to cache data and reduce the time spent in copying operations.

In [102] the authors propose EML [129], another system that combines Flink and GPUs to process large datasets. Although very similar to GFlink, EML does not use a separate process for managing GPU execution but modifies the existing Flink TaskManagers to support both execution modes.

Spark-GPU [389] is a CPU-GPU hybrid data analytics system based on Apache Spark[393]. Data in Spark is modeled as RDDs [392] and consumed in a one-element-at-a-time fashion. However, this does not match the massive parallelization a GPU can offer and leads to resource underutilization. Spark-GPU overcomes this issue by extending Spark’s iterator model and introducing GPU-RDD: a new structure which buffers all its data in native memory and can be consumed in a per-element or per-block basis. Moreover, as GFlink, Spark-GPU utilizes native memory instead of the Java heap space in order to avoid the excessive cost of often serialization-deserialization tasks.

For taking advantage of the capabilities of a GPU, isolation should exist and only one application should run on the device at a given time. At the moment when Spark-GPU was developed, resource managers like YARN [372] and Mesos [198] did not offer isolated execution for GPUs. Thus, to fully exploit the potential of the underlying hardware, Spark-GPU comes with its own custom resource manager.

Regarding task scheduling, Spark-GPU extends the Spark-SQL query optimizer and creates a GPU-aware version of it. A rule-based optimizer adaptively schedules user queries to the most beneficial hardware device. The selection of a GPU for a task depends on whether an algorithm fits the GPU execution model.

HeteroSpark [250] is another GPU-accelerated Spark-based architecture. Applications that run on top of HeteroSpark can explicitly choose whether or not a task should be executed on a GPU device.

Contrary to the aforementioned systems, objects are serialized and deserialized on demand. Furthermore, accelerating a Java application with HeteroSpark does not happen in an automated way. It requires the following steps: (i) write a GPU kernel, (ii) develop a wrapper in C that makes use of the Java Native Interface (JNI) in order to create a Java API for the kernel, and (iii) deploy in Spark.
SWAT [184] and SparkCL [339] are two open-source frameworks that are able to accelerate user-defined Spark kernels by using OpenCL. While SWAT supports only GPU acceleration, SparkCL targets a broader range of processing devices, like GPUs, APUs, CPUs, FPGAs and DSPs.

Both systems use the Aparapi [47] framework for translating Java methods to OpenCL kernels and the communication with the accelerators is based on the on-demand serialization-deserialization of Java objects.

A flurry of activity in the development of heterogeneous systems also exists in the realm of machine learning. The well-known Google’s TensorFlow [34] is a system that enables the training of machine learning algorithms over large-scale heterogeneous environments. The devices TensorFlow supports are: CPUs, GPUs, and TPUs (Tensor Processing Units, a unit specialized for machine learning applications). The execution model is based on a dataflow graph, where nodes represent tasks and edges data dependencies. Each task can run on a different kind of processor, and the user can explicitly select the device of preference for each separate task. In case she does not, Tensorflow may employ an automatic placement algorithm. However, the algorithm takes into account only basic considerations (e.g., a stateful operation and its state should be placed in the same device) and is not yet mature enough to take decisions that guarantee optimal performance in large-scale clusters.

Tensors, the logical abstraction of Tensorflow’s data structures, consist of primitive values that can be efficiently interpreted by all supported devices. Thus, concerns about the overhead of serialization-deserialization do not apply in this case.

3.4.3 Intelligence

The approaches presented in this subsection alleviate the burden of manually selecting the most beneficial mapping between application tasks and heterogeneous hardware. Contrarily, they automatically determine the processing element that best fits each application, input, and configuration, making educated decisions on the preferred computing resource based on intelligence derived from code analysis, offline profiling, and machine learning techniques.

Qilin [259] is a heterogeneous programming system that relies on an adaptive mapping technique, which automatically maps computations to heterogeneous devices. Qilin offers a programming API built on top of C/C++, which provides primitives to express parallelizable operations. The Qilin API calls are dynamically translated into native CPU or GPU code though the Qilin compiler. More specifically, the compiler first builds a Directed Acyclic Graph (DAG), where nodes represent computational tasks of the application and edges represent data dependencies. Then, it automatically finds the near-optimal mapping from computations to processing units relying on a per-task linear regression model that provides execution time estimations for the current problem size and system configuration. The regression model of each task is trained on-the-fly, during the task’s first execution under Qilin: The input
Heterogeneous Device Selection

is divided in multiple parts and assigned equally to the CPU and the GPU. Execution-time measurements are fed to curve-fitting techniques to construct linear equations, which are then used as projections for the actual execution time of the task over CPU or GPU.

Hayashi et al. [195] have developed a system for compiling and optimizing Java 8 programs for GPU execution extending IBM’s just-in-time (JIT) compiler. One of the extensions consists in adding the capability of automatic CPU vs. GPU selection. This is achieved by using performance heuristics that rely on supervised machine learning models, i.e., a binary Support Vector Machine (SVM) classifier constructed in an offline manner using as training data measurements of actual program executions over various input datasets. The classifier input dimensions consist of a set of static code features that affect performance, extracted at compile time: loop range of a parallel loop, number of instructions per iteration, number of array accesses and data transfer size. The output is the preferred computing resource (CPU vs. GPU) that optimizes the program performance.

The work in [181] proposes an approach to partitioning data-parallel OpenCL tasks among the available processing units of heterogeneous CPU-GPU systems. Code analysis is used during compilation to extract 13 static code features, which include information such as the amount of int and float operations, the number of memory accesses, the size of data transferred, etc. Principal Component Analysis (PCA) is applied to reduce the dimensionality of the feature space and normalize the data. The normalized, low-dimensional data are then passed through a two-level machine-learning predictor that performs hierarchical classification to determine the optimal partitioning for the corresponding OpenCL program: The first level distinguishes CPU- and GPU-only optimal executions using a binary Support Vector Machine (SVM) classifier, while the second one handles the cases where the best performance is achieved when distributing execution over both GPU and CPU. The latter is performed by classifying programs to nine different categories along the spectrum between CPU- and GPU-only (i.e., 10% CPU-90% GPU, 20% CPU-80% GPU, etc.) using again SVM models. Both models are trained in an offline manner with profiling measurements over various partitioning schemes.

HCl (Heterogeneous Cluster) [227] is a scheduler that maps heterogeneous applications to heterogeneous clusters. HCl represents heterogeneous applications as Directed Acyclic Graphs (DAG), where nodes stand for computations and edges represent data transfer between connected nodes. Taking into account the I/O volume between DAG tasks, the available hardware resources and the runtime estimations of each task on each resource, HCl exhaustively evaluates all possible combinations of task to node mappings and selects the global optimum, i.e., the execution schedule that optimizes the entire task graph rather than each task separately.

DeepTune [119] is an optimization framework that relies on machine learning over raw code. Its goal is to bypass the code feature selection stage involved in the techniques presented so far. Feature selection requires manual work by
domain experts and heavily affects the quality of the resulting machine learning model. One of the demonstrated use cases of DeepTune is the creation of a heuristic to select the optimal execution device (CPU or GPU) for an OpenCL kernel.

The architecture of DeepTune is a machine-learning pipeline: after the source code is automatically rewritten according to a consistent code style, it is transformed into a sequence of integers using a language-specific vocabulary which maps source code tokens to integer indices. Using embeddings that translate each token of the vocabulary to a low-dimensional vector space, the sequence of integers is transformed into a sequence of embedding vectors that capture the semantic relationship between tokens. A Long Short-Term Memory (LSTM) neural network is then used to extract a single, fixed-size vector that characterizes the entire sequence of embedding vectors. During the last stage, the resulting vectors, i.e., the learned representations of the source code is fed to a fully connected, two-layer neural network to make the final optimization prediction: The first layer has a constant number of neurons, while the second layer consists of one neuron per possible heuristic decision.

3.5 Emerging Programming Models and Architectures

This section describes the state-of-the-art and current developments in the areas of heterogeneous programming models and architectures. The overall description is divided into two parts. The first part describes the most common hardware-oriented solutions, where hardware heterogeneity is tackled by deploying low-level software drivers and end-to-end frameworks which unify the different sets of underlying hardware. The second part describes a different approach in which an abstraction layer is generated, mostly using contemporary containerization techniques, that handles all different hardware in a seamless matter. The user code thus remains hardware-agnostic but is executed in a highly efficient manner.

3.5.1 Hardware-bound Collaborative Initiatives

3.5.1.1 Khronos Group

The Khronos Group\(^6\) is probably the leading industry consortium for the creation of open-standard, royalty-free application programming interfaces (APIs) for authorization and hardware-accelerated playback of graphics and dynamic media in general, onto a large variety of devices. The activity of the consortium is mainly towards video, however, all heterogeneous platforms benefit from software and APIs which enhance the abilities of the underlying

\(^6\)https://www.khronos.org
hardware. In addition, the consortium is also interested in parallel computation efficiency thus having several of its solutions addressing related issues.

**Vulkan**

Vulkan\(^7\) is a cross-platform graphics and compute API which provides highly-efficient access to all modern GPU hardware resources. Its prime scope is to offer higher performance in 3D graphics applications, such as video and interactive media, with a more balanced CPU/GPU utilization. More specifically, the API ensures that the GPU only executes shaders while, the CPU executes everything else. In addition to its lower CPU usage, Vulkan is also able to better distribute work among multiple CPU cores, offering a reduced driver overhead, and extensively utilizing batching thus releasing more computational cycles for the CPU. The latest version of the API, Vulkan 1.1 also supports subgroup operations, an important new feature since it enables highly-efficient sharing and manipulation of data between multiple tasks running in parallel on a GPU.

**SPIR**

Standard Portable Intermediate Representation (SPIR)\(^8\) is an intermediate language for parallel compute and graphics, originally developed for use with OpenCL. SPIR has now evolved into a cross-API intermediate language that is fully defined by Khronos with native support for shader and kernel features used by affiliated APIs. The current version, SPIR-V 1.3 was released on March 7th, 2018 to accompany the launch of Vulkan 1.1, and is designed to expand the capabilities of the Vulkan shader intermediate representation by also supporting subgroup operations thus enabling enhanced compiler optimizations. SPIR-V is the first open-standard, cross-API intermediate language for natively representing parallel compute and graphics. It is part of the core specifications of OpenCL 2.1, OpenCL 2.2, and it is supported in an OpenGL 4.6 extension but is no longer using LLVM.

However, Khronos has open-sourced SPIR-V/LLVM conversion tools to enable construction of flexible toolchains that use both intermediate languages. SPIR-V is catalyzing a revolution in the language-compiler ecosystem—it can split the compiler chain across multiple vendors’ products, enabling high-level language front-ends to emit programs in a standardized intermediate form to be ingested by Vulkan, OpenGL, or OpenCL drivers. For hardware vendors, ingesting SPIR-V eliminates the need to build a high-level language source compiler into device drivers, significantly reducing driver complexity, and will enable a broad range of language and framework front-ends to run on diverse hardware architectures.

\(^7\)https://www.khronos.org/vulkan/
\(^8\)https://www.khronos.org/spir/
SYCL

SYCL\(^9\) is a high-level abstraction layer that builds on the underlying concepts, portability, and efficiency of OpenCL, thus allowing code for heterogeneous processors to be written in a “single-source” style using completely standard C++. SYCL single-source programming enables the host and kernel code for an entire application to be contained in the same source file, in a type-safe way, and with the simplicity of a cross-platform asynchronous task graph. SYCL includes templates and generic lambda functions to enable higher-level application software development. SYCL not only introduces the power of single-source modern C++ to the SPIR world, but with its recent 1.2.1 revision 3, it integrates features related to machine learning environment requirements. In addition, Khronos provides an open-source implementation to experiment with and provide necessary feedback\(^10\).

OpenKODE

OpenKODE\(^11\) is a royalty-free, open standard that combines a set of native APIs to increase source portability for rich media and graphics applications. It reduces mobile platform fragmentation by providing a cross-platform API for accessing operating system resources, and a media architecture for portable access to advanced mixed graphics acceleration. OpenKODE also includes the OpenKODE Core API that abstracts operating system resources to minimize source changes during application porting.

3.5.1.2 OpenCAPI

OpenCAPI\(^12\) is an Open Interface Architecture that allows any microprocessor to attach to (i) Coherent user-level accelerators and I/O devices, (ii) Advanced memories accessible via read/write or user-level DMA semantics, (iii) Agnostic-processor architectures. This initiative aims to create an open, high-performance bus interface based on a new bus standard called Open Coherent Accelerator Processor Interface (OpenCAPI) and grow the ecosystem that utilizes this interface. The main drive behind this initiative is the constantly increasing acceleration on computing and advanced memory/storage solutions that have introduced significant system bottlenecks in today’s current open-bus protocols, all of which require a technical solution that is openly available.

---

\(^9\)https://www.khronos.org/sycl/
\(^10\)https://github.com/Xilinx/triSYCL
\(^11\)https://www.khronos.org/openkode/
\(^12\)https://opencapi.org/
3.5.2 Serverless Frameworks

Function-as-a-Service (FaaS) approach has recently emerged and immediately gained a lot of interest in the Cloud services community. Initially, only Cloud providers provided such a functionality but recently several open-source approaches emerged. This trend was also boosted by the rise of container orchestrators simplifying the deployment and execution of such frameworks in a serverless way.

Serverless services enable the developers to compose an application from several multi-language services disengaging different teams from utilizing the same programming language. Moreover, since, every individual Cloud provider is not able to meet all of the requirements of every service composing a platform, it is common to deploy services to multiple Cloud providers according to the needs of each service. This requirement can also be covered by serverless architectures enabling the deployment of a platform in heterogeneous Cloud infrastructures.

OpenFaaS

OpenFaaS [17] is an open-source FaaS framework that utilizes Docker and Kubernetes to host serverless functions. OpenFaaS is able to package any process as a function and execute it anywhere utilizing containerization technology; it supporting many programming languages. Since OpenFaaS is open source, it can be easily extended to meet any specific requirements, for instance to add support for a programming language.

Nuclio

Nuclio [14] is another serverless framework focused on high-performance events and data processing. Nuclio provides a convenient way for the user to define any function for processing data and event streams providing integration with several heterogeneous data sources. Nuclio provides a convenient SDK to write, test, and submit function code without knowledge on the entire Nuclio architecture and source code.

Fn Project

Fn Project [7] is another open-source serverless platform utilizing Docker containers since each function submitted in Fn is executed in a Docker container. Fn provides support for a broad range of programming languages and utilizes a smart load balancer for routing traffic to functions. Through the Fn FDK (Function Development Kit) a user is able to quickly bootstrap functions in any language supported by Docker, defining input source binding models, and testing the submitted functions.
Apache OpenWhisk

Apache OpenWhisk [3] is also an open-source distributed serverless platform that is event-driven since the functions are executed in response to events. OpenWhisk also uses Docker containers to manage the infrastructure and handle scaling. Again, the user is able to define a function block that gets triggered upon the reception of an event providing integration with several sources. OpenWhisk supports several programming languages and provides a CLI for managing several aspects of an OpenWhisk instance.

Kubeless

Kubeless [10] is a Kubernetes native serverless framework designed to be deployed on top of a Kubernetes cluster, leveraging Kubernetes infrastructure management, auto-scaling, routing, monitoring and other primitives. Kubeless uses a Custom Resource Definition to be able to create functions as custom Kubernetes resources. It then runs an in-cluster controller that watches these custom resources and launches runtimes on-demand. The controller dynamically injects the functions code into the runtimes and makes them available over HTTP or via a PubSub mechanism.

Fission

Fission [6] is also a framework for serverless functions on Kubernetes enabling the definition of functions in any language, and it maps them to event triggers abstracting away containers creation and management. Moreover, Fission Workflows enable the orchestration of a set of serverless functions without directly dealing with networking, message queues, etc., automating the process of building complex serverless applications that span many functions.

Funktion

Funktion [8] is an open-source event-driven lambda-style programming model designed for Kubernetes. Funktion supports several event sources and connectors including many network protocols, transports, databases, messaging systems, social networks, Cloud services, and SaaS offerings. Funktion is a serverless approach to event driven microservices and focuses on being Kubernetes- and OpenShift-native rather than a generic serverless framework.

Quebic

Quebic [19] is a FaaS framework for writing serverless functions to run on Kubernetes. Currently quebic supports only Python, Java, and NodeJS. The event-driven messaging mechanism of Quebic enables invocations from an API Gateway to function or inter-functions calls. Quebic also provides automated processes for submitted functions update or downgrade and, apart from functions, Quebic can also host event-driven microservices.
Ongoing European Projects

Riff

Riff [21] is another open-source serverless framework that works in any certified Kubernetes environment. Again Riff is a framework designed for running functions in response to events. Since functions in Riff are packaged as containers, they can be written in a variety of languages and provides integration with several event sources. In Riff when a software function is triggered, Kubernetes, the orchestrator of Riff, spins one container and afterwards kills it off abstracting away those operations from the developers.

Serverless Framework

Serverless Framework [22] is an open-source CLI for building and deploying serverless applications. Serverless enables Infrastructure as Code defining entire serverless applications, utilizing popular serverless technologies like AWS Lambda, with simple configuration files. Serverless framework is Cloud provider agnostic and provides a simple, intuitive CLI experience that makes it easy to develop and deploy applications to public Cloud platforms. Serverless framework supports also several programming languages and provides a robust plug-ins ecosystem and built-in support for application life-cycle management.

3.6 Ongoing European Projects

This section gives a high-level overview of ongoing European projects that perform research on programming models for heterogeneous computing and study heterogeneity programming challenges. Given the fact that in the upcoming era of exascale computing [147], where systems will be capable of a quintillion $10^{18}$ floating-point operations per second (FOPS), all major infrastructure is expected to be vastly heterogeneous and heavily relying on GPUs [244], we have also included the most prominent ongoing projects from the relevant EU call

3.6.1 ALOHA

ALOHA [14] aims to ease the deployment of deep learning (DL) algorithms on edge-nodes. To achieve its goals the ALOHA project will develop a software development tool flow, automating among others the porting of DL tasks to heterogeneous embedded architectures, their optimized mapping, and scheduling. The ALOHA project uses two distinct platforms as test-beds. The first one is a low-power Internet of Things (IoT) platform, while the second is an

14https://www.aloha-h2020.eu/
FPGA-based heterogeneous architecture designed to accelerate Convolutional Neural Networks (CNNs).

3.6.2 E2Data

E2Data\(^{15}\) proposes an end-to-end solution for Big Data deployments that will deliver performance increases while utilizing less Cloud resources without affecting current programming norms (i.e., no code changes in the original source). E2Data will provide a new Big Data paradigm, by combining state-of-the-art software components, in order to achieve maximum resource utilization for heterogeneous Cloud deployments. The evaluation will be conducted on both high-performing x86 and low-power ARM cluster architectures representing realistic execution scenarios of real-world deployments in four resource-demanding applications from the finance, health, green buildings, and security domains.

3.6.3 EPEEC

The European joint Effort toward a Highly Productive Programming Environment for Heterogeneous Exascale Computing\(^{16}\), in short EPEEC, aims to deliver a production-ready parallel programming environment that will ease the development and deployment of applications on the upcoming overwhelmingly-heterogeneous exa-scale supercomputers. The project will advance and integrate state-of-the-art components based on European technology, with the ultimate goal to provide high coding productivity, high performance, and energy awareness.

3.6.4 EXA2PRO

EXA2PRO\(^{17}\) is another project that aims to deliver a programming environment that will enable efficient exploitation of exa-scale systems’ heterogeneity. The EXA2PRO programming environment will support a wide range of scientific applications, provide tools for improving source code quality and integrate tools for data and memory management optimization. Furthermore, it will provide performance monitoring features, as well as fault-tolerance mechanisms.

3.6.5 EXTRA

EXTRA\(^{18}\) project aims to create a new and flexible exploration platform for developing reconfigurable architectures, design tools and HPC applications with runtime reconfiguration built-in from the start.

\(^{15}\)https://e2data.eu
\(^{16}\)https://cordis.europa.eu/project/rcn/215832_en.html
\(^{17}\)https://exa2pro.eu
\(^{18}\)https://www.extrahpc.eu/
3.6.6 LEGaTO

LEGaTO’s goal is to provide a software ecosystem for Made-in-Europe heterogeneous hardware composed of CPUs, GPUs, FPGAs and FPGA-based data-flow engines (DFEs). LEGaTO will leverage task-based programming models, similar to OpenMP, to ultimately achieve a one order of magnitude increase in energy efficiency. Additionally, LEGaTO will explore ways to ensure the resilience of the software stack running on the heterogeneous hardware.

3.6.7 MANGO

MANGO targets achieving extreme resource efficiency in future QoS-sensitive HPC through ambitious cross-boundary architecture exploration for performance/power/predictability (PPP) based on the definition of new-generation high-performance, power-efficient, heterogeneous architectures with native mechanisms for isolation and quality-of-service, and an innovative two-phase passive cooling system. Its disruptive approach will involve many interrelated mechanisms at various architectural levels, including heterogeneous computing cores, memory architectures, interconnects, runtime resource management, power monitoring and cooling, as well as programming models. The system architecture intends to be inherently heterogeneous as an enabler for efficiency and application-based customization, where general-purpose compute nodes (GN) are intertwined with heterogeneous acceleration nodes (HN), linked by an across-boundary homogeneous interconnect. It will provide guarantees for predictability, bandwidth, and latency for the whole HN node infrastructure, allowing dynamic adaptation to applications.

3.6.8 MONT-BLANC

MONT-BLANC is a long-running project, currently in its fourth phase. MONT-BLANC’s aim is to provide solutions for European energy-efficient HPC. During its previous phases, among others, the project created an ARM-based HPC cluster and proposed techniques to address the challenges of massive parallelism, heterogeneous computing, and resiliency.

3.6.9 PHANTOM

PHANTOM aims to enable next-generation heterogeneous, parallel and low-power computing systems, while hiding the complexity of the underlying hardware from the programmer. The PHANTOM system comprises a hardware-agnostic software platform that will offer the means for multi-dimensional

---

19https://legato-project.eu/
20http://www.mango-project.eu
21https://www.montblanc-project.eu/
22http://www.phantom-project.org/
optimization. A multi-objective scheduler decides where in the computing continuum (e.g., Cloud, embedded systems, mobile devices, desktops, data centers), at which cross-layer system level (analog, digital, hybrid analog-digital, software) and on which heterogeneous technology (GPU, FPGA, CPU) to execute each part of an application. Additionally, it orchestrates dynamically the hardware and software components of reconfigurable hardware platforms.

3.6.10 RECIPE

RECIPEd (REliable power and time-ConstraInts-aware Predictive management of heterogeneous Exascale systems) will provide a hierarchical runtime resource-management infrastructure to optimise energy efficiency and minimise the occurrence of thermal hotspots. At the same time it will enforce the time constraints imposed by the applications, and ensure reliability for both time-critical and throughput-oriented computation. Apart from the runtime itself RECIPE’s second work package includes a task specifically focusing on programming models.

3.6.11 TANGO

The scope of TANGO24 is to provide the means for controlling and abstracting underlying heterogeneous hardware architectures, configurations and software systems including heterogeneous clusters, chips and programmable logic devices while developing tools to optimize various dimensions of software design and operations (energy efficiency, performance, data movement and location, cost, time-criticality, security, dependability on target architectures). The key novelty of the project is a reference architecture and its actual implementation that includes the results of the research work into different optimization areas (energy efficiency, performance, data movement and location, cost, time-criticality, security, dependability on target architectures). Moreover, TANGO integrates a programming model with built-in support for various hardware architectures including heterogeneous clusters, heterogeneous chips and programmable logic devices. In addition, TANGO creates a new cross-layer programming approach for heterogeneous parallel hardware architectures featuring automatic code generation including software and hardware modeling. Last but not least, the project provides certain mechanisms which facilitate the control of all aforementioned heterogeneous parallel infrastructures in an open-source toolbox25.

23http://www.recipe-project.eu/
24http://www.tango-project.eu
25http://www.tango-project.eu/content/beta-tango-toolbox-released-open-source
3.6.12 VINEYARD

VINEYARD\textsuperscript{26} aims to develop an integrated platform for energy-efficient heterogeneous data centers based on servers with programmable hardware accelerators. To increase productivity on such platforms, VINEYARD also builds a high-level programming framework for allowing end-users to seamlessly utilize such heterogeneous platforms by using typical data-center programming frameworks (e.g., Storm, Spark, etc.).

3.7 Conclusions

This chapter provided a high-level overview of current programming and architecture models for heterogeneous computing. It first showed an overview of heterogeneous programming models such as OpenACC, CUDA, and OpenCL. Then, it covered programming such systems from managed programming languages such as Java, Python, and R. It also presented the state-of-the-art research and projects regarding device selection for heterogeneous computing. Finally, it showed current initiatives and ongoing European projects working towards improving and handling programming for heterogeneous computing.

\textsuperscript{26}http://vineyard-h2020.eu