UNIVERSITY OF PISA

Department of Computer Science

Ph.D. Thesis

FSCL

Homogeneous programming, scheduling and execution on heterogeneous platforms

Candidate

Gabriele Cocco

Supervisor

Dott. Antonio Cisternino

December 2014

Acknowledgements

I wish to thank my parents, who have always believed in me, accepting and understanding every single choice I made along the road of life.
I thank my supervisor Antonio Cisternino, for his suggestions, his efforts and his valuable support throughout the years of my Ph.D. research.
Finally, I thank Davide Morelli and Andrea Canciani, who collaborated with me to build and validate the scheduling strategy, a critical part of the whole research.

Contents

List of Figures

Abstract

The last few years has seen activity towards programming models, languages and frameworks to address the increasingly wide range and broad availability of heterogeneous computing resources through raised programming abstraction and portability across different platforms.

The effort spent in simplifying parallel programming across heterogeneous platforms is often outweighed by the need for low-level control over computation setup and execution and by performance opportunities that are missed due to the overhead introduced by the additional abstraction. Moreover, despite the ability to port parallel code across devices, each device is generally characterised by a restricted set of computations that it can execute outperforming the other devices in the system. The problem is therefore to schedule computations on increasingly popular multi-device heterogeneous platforms, helping to choose the best device among the available ones each time a computation has to execute.

Our Ph.D. research investigates the possibilities to address the problem of programming and execution abstraction on heterogeneous platforms while helping to dynamically and transparently exploit the computing power of such platforms in a device-aware fashion.

The Thesis in structured in four parts. In the first part we present today's situation in heterogeneous systems, we identify the major issues induced by spreading heterogeneity and we discuss the implications. We analyse the most recent and relevant works in high-level programming, scheduling and execution on heterogeneous platforms, which represent the starting point of our Ph.D. research. The part is concluded with the formalization of the Thesis proposal.

In the second part we present the research conducted. We define the context of our work and we discuss the approaches adopted to reach the goals that our proposal entails. The presentation is organized following the timeline and the structure of our research and the inter-dependencies between the goals.

In the third part we validate the results obtained against the state of the art in heterogeneous programming and execution. We evaluate the benefits of our work and we identify the principal limitations.

In the fourth part we summarise the work conducted and the relevant outcomes. We briefly propose again the statement of the Thesis and we match it against the results obtained. Finally, we present the implications of our work, the possible refinements and the potential future research.

Part I

Foundation of the Ph.D. Thesis

Chapter 1

Introduction

In the last few years computing has become increasingly heterogeneous, offering a broad portfolio of different parallel devices, ranging from CPUs, through GPUs to APUs[1] and coprocessors. Most of the recent CPUs available on the market contain two parallel processing resources, which are a multicore CPU and a GPU. Given this on-die heterogeneity and the set of recent technologies for multi-GPU (e.g. SLI) nowadays desktop and laptop computers possibly expose one or more multicore CPUs and multiple GPUs. Intel recently released a coprocessor for cluster nodes and desktop systems, called "Xeon Phi", which is characterized by hybrid architecture and execution model.

Researches on manifacturing and optimizing hardware for heterogeneous platforms has been very active, as demonstrated by the effort spent by the majority of the silicon industries in this new kind of processing resources [3, 45, 55].

Not only has the variety of devices increased, but it has spread towards systems traditionally characterised by single computing resources, such as netbooks and mobile phones, which nowadays are equipped with multicore CPUs and GPUs. Heterogeneity on mobile devices is a hot trend in the mobile market [47].

Given the spread of heterogeneous solutions across different systems, from mobile phones to desktop PCs, and the constantly dropping price of hardware components, heterogeneous configurations are increasingly popular and affordable for a broad range of users. This means that computing power is not only becoming highly heterogeneous but it is also becoming available to users that are unaware of the underlying parallel computing power and of how to exploit it. With the term "users", we refer to either developers with no parallel

[1] APU is the term used by some processors brands to refer to a CPU and a GPU integrated onto the same die

programming skills or software that is not designed to dynamically leverage multiple heterogeneous devices.

The major problem that comes from the unawareness of platform heterogeneity is the underutilization of computing power. We consider underutilization of heterogeneous computing power a consequence of two principal aspects: the difficulty of programming across different, parallel devices and the complexity in understanding, exploiting and adapting to a highly dynamic set of resources.

To enhance portability, software tends to target the CPU, which is a computing resource available in the majority of systems. Similarly, generic programmers tend to be more productive in developing software that run sequentially on CPUs, because of the lack of parallel programming skills or because of the effort required to guarantee the software be able to fallback and adapt to different machine configurations. This is particularly true when the set of target platforms is extremely varied in terms of configurations, ranging from CPU-only systems up to machines equipped with multiple, different GPUs and coprocessors. When effort in exploiting multiple devices is spent, software tends to adopt a device-unaware usage policy, where the available resources are treated as an homogeneous set of computing nodes. Provided with an homogeneous programming layer, such a policy is easier to apply than one which takes into account the specific characteristics of each device in the set.

On mobile devices, the unawareness of the available resources and the lack of a strategy to get the most out of each of them can increase the power/battery consumption. In the last few years, research has demonstrated that CPUs and GPUs are described by two very different power consumption profiles. Whereas GPUs are described by an overall higher consumption, they tend to be more energy-saving in terms of performance per-watt [2, 12, 41].

In cloud computing, an inproper use of the available resources can lead to massive overspending. While load-balancing across the resources represents a good improvement [8, 36], extending the traditional notion of cloud computing to provide a cloud-based access model that is heterogeneity-aware can play a key role to boost performance and energy efficience [17].

It is important to consider that the spread of heterogeneous platforms is not only widening the set of programmers that work on such platforms, but it also affects the type of computations possibly run on them. While research and industrial parallel computing centers are likely to accelerate specific, complex and long-lasting algorithms, programmers of today's heterogeneous platforms often develop and execute lightweight and more generic computations. Therefore, approaches that leverage the high completion times of the running computations to cover the scheduling overhead may turn out to be too ex-

pensive when applied to more generic and lightweight programs. Similarly, approaches based on task-subdivision of computations may be unfit to handle the generality of algorithms that run on today's heterogeneous platforms.

To address the issues related to spreading heterogeneity and the dynamicity of platform configurations, research should focus on two different but interrelated aspects.

The first aspect is the complexity of programming across different parallel devices, especially considered the broad audience of programmers working on heterogeneous systems. To widen the range of programmers able to exploit the set of parallel resources populating heterogeneous platforms, an homogeneous and abstract programming layer should be provided. Whereas abstraction and expressiveness can make it easier for generic programmers to develop across devices, flexibility must be also taken into account to guarantee the largest range of algorithms to fit the programming model.

The second aspect is the difficulty to detect and characterize the set of devices that populate heterogeneous platforms in order to use them in a device-aware fashion. To get the most out of such platforms, the runtime support of an homogeneous programming layer must be able to transparently and dynamically analyse both the code to execute and the resources available in the running system. Coupling this information plays a key role in improving resource utilization and dynamic adaptation to different system configurations.

Given the wide variety of computations, possibly lightweight, that can be scheduled on today's heterogeneous systems, a pressing challenge of code analysis and scheduling is to be efficient, in order to guarantee that the overhead doesn't outweigh the time saved by running computations on the devices with the highest performance.

Finally, to enhance portability across systems and programmers, an abstract programming, scheduling and execution layer must be ready-to-go, which means to be able to transparently discover all the information needed about the running platform for self-configuration and deployment.

The two aformentioned challenges stress the need for research on raising abstraction over parallel programming and execution on heterogeneous platforms, hiding the complexity of the underlying system to the programmers while providing dynamic device- and computation-aware scheduling and execution to leverage the whole set of available computing resources.

Research towards high-level programming and execution on heterogeneous platforms should be particularly inspired by a technical report from the University of Berkeley [7], which states:

> The struggle is delivering performance while raising the level of

> abstraction. Going too low may achieve performance, but at the cost of exacerbating the software productivity problem, which is already a major hurdle for the information technology industry. Going too high can reduce productivity as well, for the programmer is then forced to waste time trying to overcome the abstraction to achieve performance.

Depite the years passed since this report was published, balancing abstraction and performance in parallel programming and execution is still an open problem. Actually, this challenge is renovated and underlined by the today's widespread heterogeneity of parallel computing power. Whereas performance is still a pressing challenge, the need of higher programming and execution abstraction is stressed by the increasing variety of available hardware resources and by the generality of the programs running on them.

Chapter 2

Programming, scheduling and execution on heterogeneous platforms

Recenty effort has been spent to address the increasing heterogeneity of the computing platforms. In this chapter we present the state of the art of this research from both the perspective of programming abstraction and the one of scheduling and execution support.

The chapter is organized in four sections. In the first section we present and discuss OpenCL, which is the today's standard-de-facto for parallel programming across different resources. In the second section we show and discuss the most recent and relevant efforts to increase the abstraction over parallel programming on heterogeneous systems. In the third section we present the relevant works that try to address the problem of efficient scheduling and execution on multi-device platforms. Finally, in the fourth section we summarize the state of the art, we discuss the major flaws and the open challenges, and we formulate our Ph.D. Thesis proposal.

2.1 OpenCL

Originally developed as a private API by Apple and published in 2009, OpenCL is an open specification for writing and executing programs for heterogenous platforms.

In the last couple of years, thanks to the huge contribution from the programming community and to the many companies that decided to support it on their own devices, OpenCL has become one of most popular solutions for heterogeneous programming.

2.1.1 OpenCL execution and memory model

From the OpenCL point of view a system is made of one or more *devices* grouped into *platforms*. Each platform generally corresponds to a specific manifacturer (e.g. AMD, Intel) with its own OpenCL-to-executable compiler and runtime support (called *client driver*). Whereas multiple devices can populate a system, the OpenCL specification stands that there is one only *host*, running on the CPU, whose task is to coordinate the execution of computations on the devices (figure 2.1).

The OpenCL specification describes a device abstracted from the particular hardware configuration. An OpenCL device is made of one or more *compute units* (CUs) which are further divided into one or more *processing elements* (PEs). On a GPU, compute units map to independent cores that execute in parallel multiple instructions on multiple data, while processing elements map to the arithmetic and logic units in each core. On a CPU, both compute units and processing elements generally correspond to CPU cores.

An OpenCL application runs on the host according to the native model of the host platform. The OpenCL application submits commands to run computations on the processing elements of a device. The processing elements within a compute unit execute a single stream of instructions as SIMD units (Single Instruction Multiple Data) or as SPMD (Single Program Multiple Data) units.

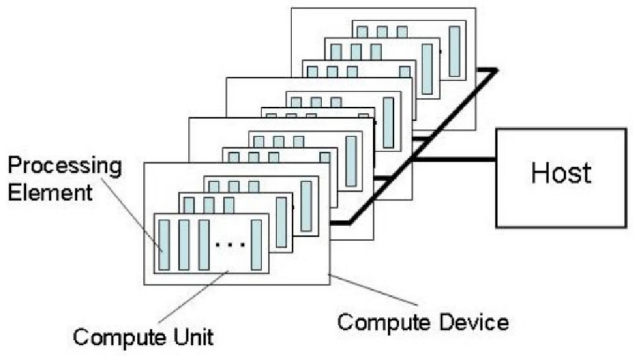

Figure 2.1: Abstract view of host and devices in OpenCL

The distinction between host and devices is not only conceptual but affects the typical code structure of OpenCL programs. A program is in fact made of two distinct parts: the *kernel code* and the *host code*. The kernel code implements the parallel computation (kernel) to run on the device, while the host code handles resource allocation, computation setup, scheduling and

synchronisation with the execution of the kernel on the device.

OpenCL defines an index space for parallel kernel execution. An instance of the kernel executes for each point in this index space. This kernel instance is called *work-item* and is identified by its point in the index space, which provides a *global ID* for the work-item. Each work-item executes the same code but the specific execution path through the code and the data used can vary per work-item.

Work-items are organized into work-groups. The work-groups represent a coarse-grained decomposition of the index space. Work-groups are assigned a unique *work-group ID* with the same dimensionality as the index space used for the work-items. Work-items are assigned a unique *local ID* within a work-group so that a single work-item can be uniquely identified by its global ID or by a combination of its local ID and work-group ID. The set of work-items in a given work-group execute concurrently on the processing elements of a single compute unit (figure 2.2).

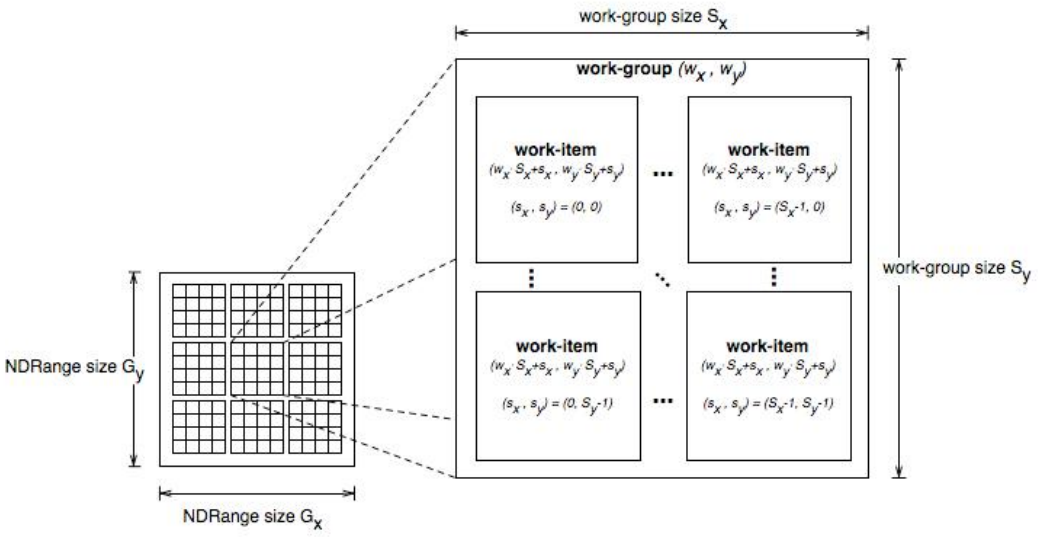

Figure 2.2: OpenCL index space showing work-items and their relative global, local, work-group IDs

OpenCL context resources

The host defines a context for the execution of the kernels, which includes the following resources:

Devices A collection of OpenCL devices to be used by the host for the execution of kernels;

Kernels The set of parallel computations to run

Program Objects The program source and executable that implement the kernels;

Memory Objects A set of memory objects visible to the host and to the devices.

Program objects contain compiled kernels and other information essential for the execution on the device. To create a program object, the host uses specific OpenCL functions, submitting the sources of the kernels forming the program and obtaining an opaque pointer to the executable code. OpenCL kernels are therefore compiled into executables at (host) runtime.

Memory objects contain data that can be accessed and possibly modified by the instances of a kernel. These objects can be of two types: *buffer objects*, and *image objects*. A buffer object stores a one-dimensional collection of elements whereas an image object is used to store a two or three-dimensional texture, frame-buffer or image.

Elements of a buffer object can be scalar data (e.g. int, float), vector data (e.g. float4, int8), or user-defined structures. An image object instead represents a buffer that can be used as a texture or a frame-buffer, whose elements are selected from a list of predefined image formats.

The context and all its resources are created and manipulated by the host using the OpenCL API.

The host also creates a data structure called a *command-queue* to coordinate the execution of kernels on the devices. The host submits commands into the command-queue which are then scheduled onto the devices within the context. These commands include:

Kernel execution commands Execute a kernel on the processing elements of a device;

Memory commands Transfer data to, from, or between memory objects, or map/unmap memory objects to/from the host address space;

Synchronization commands Constrain the order of execution of commands.

Commands scheduled on the queue execute asynchronously between the host and the device and can run relative to each other in an *in-order* or *out-of-order* mode. Programmers can choose a specific relative ordering when creating the command-queue.

Kernel execution and memory commands submitted to a queue generate event objects, which are used to control the execution between commands and to synchronize the host with the device.

OpenCL memory model

Work-items executing a kernel have access to four distinct memory regions, as illustrated in figure 2.3.

Global memory This memory region allows access from all work-items in all work-groups. Work-items can read from or write to any element of memory objects placed in global memory;

Constant Memory A region of global memory that remains constant during the execution of a kernel. The host allocates and initializes memory objects placed into constant memory. Work-items have read-only access rights to the elements of these objects;

Local Memory A memory region local to a work-group. This memory region can be used to share data among the work-items in a work-group;

Private Memory A region of memory private to a work-item. Data placed in one work-item private memory is not visible to the other work-items.

The application running on the host uses the OpenCL API to create memory objects in global/constant memory and to enqueue memory commands that operate on these objects.

Since the host is defined outside of OpenCL, the memory hierarchies of the host and the device are generally separated and independent from each other. Interactions between host and device memory occur by either explicitly copying data or by mapping/unmapping regions of a memory object. To copy data explicitly, the host enqueues commands to transfer data between the memory object and host memory. The mapping/unmapping functions instead allow the host to map a region from the memory object into its address space. Once a region from the memory object has been mapped, the host can read or write to this region. The changes to the content of the mapped region possibly performed by the host are visible to the device after the region is unmapped.

When a memory object is created, some *memory-flags* can be specified to determine the placement in the memory and the access to the object. For example, *AllocHostPtr* flag specifies that the application wants the OpenCL implementation to allocate memory from host accessible memory, while *UseHostPtr* flag indicates that the application wants to use memory referenced by a user-provided pointer as the storage bits for the memory object.

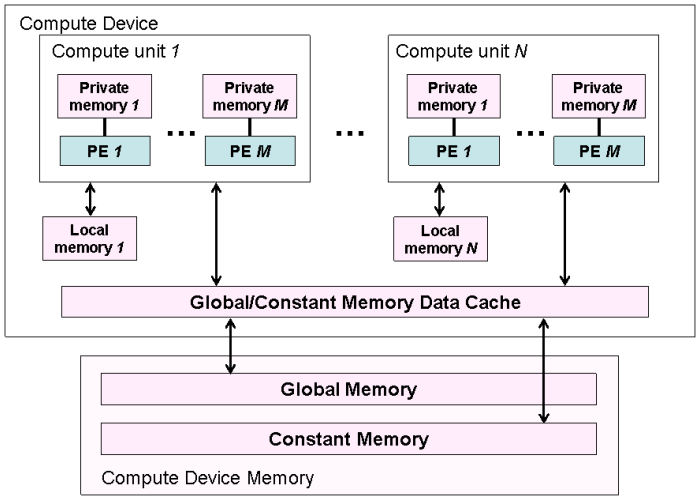

Figure 2.3: OpenCL conceptual organization of memory regions

Synchronization between commands and processing elements

There are two domains of synchronization in OpenCL:

- Work-items in a single work-group
- Commands enqueued to command-queues in a single context

Synchronization between work-items in a single work-group is achieved using a work-group *barrier* in the kernel code. All the work-items in a work-group must execute the barrier before any are allowed to continue. OpenCL doesn't provide any mechanism of synchronisation between work-groups.

The synchronization points between commands in command-queues are:

Command-queue barrier A command-queue barrier ensures that all previously queued commands have finished execution and any resulting updates to memory objects are visible to subsequently enqueued commands before they begin execution. This barrier can only be used to synchronize commands in a single command-queue;

Waiting on an event All the OpenCL API functions that enqueue commands return an event that identifies the command and the memory objects involved. A subsequent command waiting on that event is guaranteed that updates to those memory objects are visible before the command begins execution.

2.1.2 A summary on OpenCL programming

OpenCL represents the today's standard for programming across heterogeneous devices, thanks to which parallel algorithms can be developed once to be then executed on a wide variety of different devices. Despite the portability and the homogeneity of coding, OpenCL shows various drawbacks and limitations.

With regards to programming, OpenCL coding is low-level and requires non-trivial parallel programming skills to guarantee the correctness of kernel execution. In addition, host-side and device-side models are quite different from each other, from both the language object-model and the concurrency paradigm points of view.

OpenCL kernels are written in a subset of C99 with some restrictions and extensions. For example, parameters of kernels that are pointers must come along with a *global*, *constant* or *local* qualifier. Pointers-to-pointers are not supported, which means that vector elements can be only of primitive or struct types. Kernel functions must return *void* and the library functions defined in the C99 standard headers are not supported[1]. On the other hand, host-side is programmed according to the native model of the host platform. Whereas C (without restrictions) is the specification language of the OpenCL API, various wrappers have been built over the latest years to support the execution of kernels from within different languages, like C# and Python.

Some data-types, like vector types (e.g. *float4*, *int8*, *half16*) and image-specific types (e.g. *image2d_t*) are available only on the kernel-side. On the host-side, programmers must use different data-types, such as regular pointers to primitive types (e.g. *float**, *int**, *half**) in place of pointers to vector types (*float4**, *int8**, *half16**) and the opaque *cl_mem* to hold image data.

The concurrency model is also different between the host and the device. On the device, individual processing elements within a group synchronize with each other in a shared-memory fashion. On the other hand, synchronisation between the host and the device and between individual commands submitted by the host is based on events and explicit messages (commands), which resembles the message-passing paradigm.

From the productivity point of view, host-side coding is mostly boilerplate. The steps to code in the host are in fact the same, independently from the kernel executed. These steps include loading kernel sources, compiling them, instantiating a device, creating buffers, setting up kernel arguments, schedul-

[1]This refers to OpenCL Specification 1.2. The new OpenCL specification 2.0 removes some of these restrictions

ing the kernel and waiting for its completion. Nevertheless, host-side coding requires time and effort to guarantee the correctness of execution, especially in allocating/initializing buffers and setting up kernel arguments. Whenever a programming mistake is introduced in the host-side (e.g. because of incorrect buffer size with consequent out-of-bounds accesses), kernel execution may fail or interrupt obscurely. Programmers are therefore asked to spend time and energy in coding *how* a computation has to be executed instead of strictly focusing on *what* the computation has to do (i.e. the kernel).

Despite the recent availability of debugging and profiling tools, debugging OpenCL computations is still difficult and error-prone. OpenCL compilers/-client drivers are characterized by quite basic, and sometime obscure, error reporting, which is definitely not as advanced as in compilers like GCC and Clang-LLVM. Recently, real-time debugging of kernels executing on a device became possible, but programmers must use specific, third-party tools. Productivity in spotting and fixing coding mistakes as well as in detecting runtime issues is therefore very limited.

From the perspective of execution, OpenCL doesn't provide any help to optimise code for specific types of devices nor to understand which of the available devices should be selected to execute a particular kernel with high performance.

Even though kernels can run across different types of devices, to obtain the highest performance device-specific optimisations must be applied. For example, vector data-types can speed up computations on CPUs, while on most GPUs[2] they can be counterproductive. The memory access pattern is another aspect that can be deeply affected by optimisations for different types of devices. GPUs tend to perform better under interleaved accesses (i.e. successive work-items access successive elements of a buffer) while performance on CPUs in higher when a single work-item accesses a contiguous stride of memory, improving cache-usage and exploiting prefetching. To run efficient code across devices, OpenCL programmers must therefore be aware of the specific characteristics of the available devices and code different kernels, each optimised for a specific device.

[2]Especially GPUs with an instruction set not based on Very Long Instruction Word(s) (VLIW)

2.2 High-level programming languages and libraries

From the early ages of general purpose programming on GPU (GPGPU), various researches have been focusing on the definition of an abstraction layer to simplify parallel programming on heterogeneous platforms.

Nowadays, CUDA and OpenCL are the two major solutions for programming on GPUs. Even though OpenCL obtained a wider popularity thanks to its industry standard specification and to the broad support, OpenCL and CUDA are very similar to each other, from both the abstract execution model, the memory model, and the programming-style points of view. For this reason, most of the relevant research of the last few years has leveraged OpenCL or CUDA as the "ground layer" over which to build a more abstract and easy way to express and execute parallel computations.

The first attempts to make parallel programming easier and more confortable are represented by language bindings for CUDA and OpenCL APIs [1,15,46,75]. Despite the ability that these solutions provide to program CUDA and OpenCL host-side using languages different from C/C++ (e.g. Java, C#), they do not introduce any significant abstraction over the low-level APIs.

Given the popularity of the LLVM compiler, and especially of its intermediate representation (IR), some recent efforts have been spent on extending the LLVM backend to map the IR to the specific intermediate or target representation of GPU executables [21,44]. LLVM has been employed to transparently extend the set of target device types of CUDA programs, traditionally GPUs only, to CPUs and to all the architectures supported by LLVM. While leveraging LLVM IR can enhance architecture-specific optimisations, deliver cross-device execution transparency and, more generally, hide the heterogeneity of the platform to the frontend/users [73], from the programming abstraction perspective there have been no significant improvements. On top of these solutions, the developers continue to use the OpenCL/CUDA C/C++ API but relying on LLVM to produce the intermediate representation between code and brand-specific instruction sets. Nonetheless, being an open-source compiler, LLVM (and LLVM frontends like CLANG) may be succesfully employed to define more abstract programming models for heterogeneous platforms.

C++ AMP [30] is an open specification for implementing data-parallelism which represents a more interesting research from the point of view of programming abstraction. C++ AMP exposes reshaped programming constructs/patterns (e.g. *parallel_for_each*), lambdas and custom data-types (e.g. *array_view*) to express data-parallelism on collections. C++ AMP supports both GPU ex-

ecution[3] and CPU vector (SSE) processing. Lambda expressions represent a powerful and productive construct to express parallel code, especially in case of recurrent, succinct computations.

While leveraging on language constructs that C++ programmers are comfortable with, such as arrays and lambdas, the API also introduces additional language features for composition/dispatching that require a certain effort to be understood and used correctly. Native data-types (e.g. *array*) come along with brand new types (e.g. *array_view*) that are specifically used in place of native types to drive a certain execution/memory behaviour (e.g. data-transfer on-request). The definition of custom types in order to drive the runtime behaviour, such as enabling a particular optimization, can be confusing and difficult to assimilate. We consider *pragma* directives, gcc-like attributes and .NET custom attributes examples of programming features that are more independent from the content of the computation and, for this reason, more suitable to express meta-information.

SkePU [22] is one of the most popular libraries for data-parallelism based on skeletons. Skeleton programming is an approach that shows several advantages from both the coding and the execution support points of view. Skeletons are intuitively mapped to specific, well-known processing behaviours, are parametric (i.e. customizable) and easy to compose with each other. This makes skeleton programming easy to learn and to apply productively. The abstraction provided by skeletons makes them also the perfect candidates for transparent target-specific optimisations. Finally, given the intrinsic restrictions on the code used to parametrize skeletons, such as on the input and output type, the correctness of skeleton-based parallel programs is easy to validate. The principal drawback of SkePU and, more generally, of skeleton programming, is the restricted flexibility. If a skeleton library doesn't provide any opaque "container" to define computations that go beyond the provided skeletons, the set of algorithms that can be effectively expressed using the model exposed by the library is very narrow.

Muesli [14] is another popular skeleton-based library. Muesli supports and seemlessly integrates both data and stream paralellism. In particular, a Muesli program is a stream-based computation, where each "node" of the stream can express data-parallelism. Muesli also introduces the concept of *sequential computations* inside parallel programs, especially to distribute and collect stream data. Despite the effort to enhance flexibility by joining stream parallelism, data-parallelism and sequential functions, Muesli designs new functions and data-types (e.g. distributed arrays and matrices) instead of leveraging native

[3]On November 12, 2013 the HSA Foundation [3] announced a C++ AMP compiler that outputs to OpenCL

and possibly expressive language constructs.

FlumeJava [11] is a Java library for programming and executing data-parallel pipelines. The target of FlumeJava is to extend the flexibility of the *map-reduce* pattern by enabling multiple instances of such pattern to act as stages of a pipeline. The programming model is centered around a set of parallel collections, each of which exposes a number of operations that can be performed on its instances. Single operations on collections can be composed in a chain to form a pipeline. The collection functions provided suffer from the same flexibility constraints of most skeleton-only solutions, leading to difficulties to express out-of-scheme computations. In addition, similarly to Muesli and C++ AMP, the programming model and language are based on a set of additional, custom collection types and functions (e.g. *PCollection*, *parallelDo*, *groupByKey*), which yields a loose integration within the host language.

SkelCL [64] represents an effort to raise abstraction over OpenCL programming through skeleton definition and composition in C++. Differently from the other skeleton-based approaches considered, SkelCL embraces predefined function composition to express dependencies between computations and functional arguments to express higher-order skeletons (i.e. skeletons that compose other skeletons). Whereas pre-existing language features are employed to compose skeletons, brand new types are exposed to represent collections of data to process, similarly to other skeleton libraries. From the programming productivity point of view, a major flaw in the design of the SkelCL library consists in the way programmers specify the operators (arguments) to instantiate the skeletons. In fact, operators are expressed as quoted lambdas (strings). Since compiler syntax and type checking doesn't apply to the content of strings, programming mistakes are detected only at runtime, when the OpenCL code is generated and executed.

Intel Threading Building Blocks (TBB) [57] is a library that provides a solution for enabling parallelism in C++ code in a way that is easily accessible to developers writing loop- and task-based applications. Similarly to C++ AMP, Intel TBB exposes a set of new language features to express parallel iterations (e.g. *parallel_for*, *parallel_invoke*) and a set of data-types, called "concurrent containers" to parallel processing data. The main advantage of TBB is that, differently from other pattern- or skeleton-based libraries, it provides programmers a way to define computations that do not fit the scheme of any particular parallel pattern. This is accomplished introducing a complimentary model based on explicit flow-graph creation. The developers that need more flexibility than the one delivered by the high-level, pattern-based model can leverage flow-graph node instantiation and inter-connection to "shape" a parallel computation. The principal flaw in the design of these multiple ab-

straction levels is the profound difference between the respective programming and object models (loop-like constructs versus flow-graph node/edge objects), which may incur doubling the effort to get comfortable using the library and make it difficult to compose and coordinate parallel computations coded at different levels.

Similarly to TBB, Marrow [48] allows working at multiple levels of abstraction. Marrow is a skeleton framework for the orchestration of OpenCL computations which enables programmers to combine and nest parallel computations. The developers can use and compose the set of provided skeletons or define custom kernels and combine them with other computations in the parallel program. The most interesting aspect of Marrow, that distinguishes it from TBB and from many other patterns/skeletons solutions, is the ability for computations expressed at different abstraction levels to be composed with each other. The principal limitation of this framework is instead the same of TBB, which is the inhomogeneity of the models exposed by different programming levels. The high-level interface is in fact based on library functions (skeletons) and their compositions, while the low-level requires custom kernels in OpenCL, escaping the context of the framework, and to load the source files containing the kernels into wrappers (called *KernelWrappers*) in order to compose custom kernels with other computations.

Microsoft Accelerator [70] is a framework that exposes data parallelism to program GPUs on .NET, leveraging the abstraction provided by the Virtual Machine to hide the complexity of GPU programming and execution mentation by compiling the data-parallel operations on the fly to optimized GPU pixel shader code and API calls. While the performance of Accelerator programs is closer to the one of to hand-written pixel shaders, the programming model lacks of integration with the host language, exposing additional data types (e.g. parallel arrays) and custom reduction and transformation operations that limit productivity.

Dandelion [58] is probabily the most inspiring framework for our research. In Dandelion, parallel programs are expressed using LINQ [49] operators. Being a pre-existent and very popular model to work on collections, programmers should find in LINQ a simple and familiar programming context. LINQ operators act as a set of composable skeletons that can be parameterized. Starting from LINQ expressions, Dandelion is able to build a flow graph and to schedule the graph nodes for execution on multiple devices, handling data-transfer and synchronization transparently to the programmer. From our perspective, the major limitation of this programming and execution layer is the choice of LINQ to act as an embedded Domain Specific Language (DSL) to express parallelism. LINQ has been primarily designed to enhance and simplify querying and

updating data for arbitrary data stores. With the set of operators provided, such as *select*, *where* and *groupby*, LINQ strictly resembles the SQL language. Whereas intuitively mapped to some specific types of parallel patterns (e.g. map-reduce), these relational operators may be difficult to use to express more general parallel computations, such a *parallel prefix*. In fact, according to the authors, Dandelion provides a library of parallel patterns that can be joined with the relational operators. While this library extends the flexibility of the programming model, mixing relational-operators and patterns-functions undermines the homogeneousness of the model.

In [67] F# metaprogramming features are extensively discussed and applyed to demonstrate the ability to access specific programming domains from within F# in an expressive and extensible way. The F# quotations infrastructure is presented and applied to express high-level parallel computations in terms of point-wise operations, reduction operations and transformation operations of vector types. Quotations act as an unobtrusive construct to access the (typed) Abstract Syntax Tree of high-level programs, unleashing the possibility to provide custom interpretation of arbitrary F# expressions at runtime. For execution, the interpreter of the high-level programs relies on Microsoft Accelerator [70]. In other terms, the runtime support accesses the high-level language constructs via quotations and provides their semantic in terms of Accelerator API calls.

Herlan [33] is a Scheme-based function language to express and execute parallel programs on GPUs. In Harlan, GPU computations are expressed using declarative constructs, raising abstraction over low-level, error-prone coding in CUDA or OpenCL. In addition, programming abstraction allows the compiler to enhance runtime efficiency, transparently optimising data movement and overlapping CPU and GPU computations. Whereas functional programming unleashes high abstraction and expressive composition, Harlan currently exposes only a very limited set of high-level parallel skeletons/pre-defined functions, such as *reduce* and *scan*, forcing programmers to explicitly code kernels even for very common patterns. In addition, Harlan does its best to run the kernel on the GPU, automatically determining the thread-space configuration. While this choice provides additional abstraction, it limits the flexibility of kernel execution and seems difficult to adapt to kernels where the number of threads spawn is hard to deduce from the code or from the size of the processed data[4].

Aparapi [5] is a framework for integrated OpenCL programming in Java. Unlike regular OpenCL bindings, Aparapi enable programmers to code OpenCL

[4]A common case in *reduction*-like kernels, which are generally executed iteratively on the GPU, spawning each time a number of threads that is half the input size

kernels as Java methods, porting OpenCL types and programming constructs to the Java language. In addition, Aparapi automatizes the OpenCL host-side, allocating OpenCL resources, generating the kernel executable and coordinating the execution on the device. Aparapi relies on running in a Virtual Execution Environment (VEE) and on the introspection capabilities offered by Virtual Machines (VMs) to raise abstraction over OpenCL kernel- and host-side programming. We think that VMs play a key-role in defining high-level, abstract programming models and APIs, especially for the ability to dynamically inspect code and to drive the execution behaviour on the basis of specific information extracted from the running program. Nonetheless, from the kernel-side perspective, OpenCL kernels expressed as Java methods strictly map to (a subset of) their C/C++ counterparts, without allowing higher-level Java contructs or data-types. For example vector types (e.g. float4), tuples or other custom types cannot to be used, while support for custom structs is very limited[5]. Given the possibilities offered by the virtual environment to map and perform custom interpretation of code constructs and types, more abstract features may be introduced in kernel coding without narrowing the flexibility of OpenCL. In addition, besides the ability to write kernels as Java methods exploiting type checking, host-side code generation and other facilities, Aparapi doesn't provide any more abstract way to express parallel computations. This means that even very simple and common patterns for which parallel implementations have been long studied (e.g. map, reduce, sort) must be hand-coded whenever needed[6].

Alea GPU [19] if a .NET development environment for programming and execution on GPUs, which supports .NET languages, including C#, F# and VB. In Alea GPU, the programmer develops parallel computations as high-level (e.g. F#) functions, relying on the underlying framework to generate CUDA code which is then transparently executed on the target device. The Alea GPU F# frontend employs quotations, an interesting and nearly-unique language feature, to introspect the AST of parallel computations and transparently provide their execution. While the set of solutions to simplify parallel programming and execution on GPU looks promising, Alea GPU introduces some additional language constructs and data types that limit its full integration in the host language, such as new vector containers (e.g. $deviceptr<float>$ in place of the pre-existing array of float). In addition, while host-side coding (i.e. the program that schedules and coordinates the execution of computations) certainly requires less effort then in OpenCL/CUDA, we see many points

[5]Structs and vector types are fully supported by the OpenCL specification

[6]The latest news about Aparapi report high-order functions to express such patterns will be available soon with the release of Java 8

where additional abstraction can be introduced, for example by automating the allocation of vector arguments. Finally, F# provides a set of native functions working on collections that can be employed as skeletons to enhance productivity in expressing common parallel computations, making parallel programming and execution fully transparent. Nontheless Alea GPU, at least at the time of writing, neither supports nor takes into account these functions.

Collection processing plays a major part in Java 8 Streams API, which lets programmer express sophisticated data processing queries as pipelines of collection operators [65]. In Java 8, the programmer can use and chain the available set of collection functions (e.g. *map*, *reduce*, *scan*) and run the entire processing in a stream-based fashion on the CPU. The main limitation of Java 8 stream processing is the same that describe the majority of skeleton/patterns solutions, which is the inability to express computations that escape the strict set of available patterns. In addition, Java 8 Streams only support stream parallelism and does not leverage any parallel device apart from multicore CPUs.

In [26] a new class hierarchy on top of Java 8 Stream is proposed to support OpenCL execution and to extend stream processing by allowing user-defined functions to be expressed and composed with predifined operators. From one side, the research seems proposing a guideline for implementing parallel operators more than exposing a model to parallel programmers. In other terms the user of the model has neither the visibility of the OpenCL layer nor the ability to implement a new operator without explicitly generating the OpenCL C implementation of the operator itself. Code generation is indeed discussed only for the single operator available in the proposed framework at the time of writing (i.e. array map), which suggests that ad-hoc solutions must be applied to define new operators and "connect" them to OpenCL. This leads to assume that the programmers who define new operators are unlikely to be the same programmers who use the operators for coding parallel programs.

Nessos Streams [28] is an F# project inspired to Java 8 Streams that provides F# programmers with a continuation-passing-style composition of collection processing. Like Java 8 Streams, stream parallelism is the only choice and no support for execution on GPUs, FPGAs or different accelerators is provided. Unlike Java 8 Streams, composing Nessos Streams with user-defined computations is possible, even though parallel execution of custom computations must be explicitely defined by the programmer.

2.3 Scheduling frameworks for heterogeneous platforms

Scheduling on multi-device systems has been a fervent research area in the last decade. In this section we present and discuss the most recent works in this field, particularly focusing on scheduling approaches for CPU-GPU heterogeneous systems.

In SkePU [22], the parallel program interpreter supports multiple backends (OpenMP, OpenCL, CUDA) and multi-device execution based on equal partitioning of the input data. Specific characteristics of the available computing devices are therefore not considered. An interesting aspect of the SkePU runtime is instead the tunable context-aware implementation selection. Depending on the properties of the context where a skeleton is applied (e.g. input size), different, optimised skeleton implementations for different execution units are automatically selected.

We consider automatic selection of the best "variant" for a computation to be important in heterogeneous systems and complimentary to the device-awareness in scheduling. In fact, whether the programmers specifiy a particular device for execution or leave the system free to choose it, optimised implementation for the device where the computation will run should be used. Nonetheless, for certain algorithms no optimisations can make a device outperform another one, which is why we consider automatic variant selection *complimentary* to best-device selection for a given computation.

In [66] an intermediate infrastructure between the client driver and the OpenCL programmer is proposed to enable multi-device cooperative execution of kernels. OpenCL programmers are asked to define a *divide function* associated to a kernel in order to declare the partitioning scheme of the kernel itself. The divide function is parametric on a *partitioning factor* which encodes the granularity of the partition. The runtime uses a granularity adaptation strategy to explore the partitioning factor range in a two-phase fashion. In a first phase, the engine produces and saves statistics about execution time for each partitioning factor. In a second phase it uses and updates these statistics to choose the appropriate partitioning factor. The main limitation of this approach, besides the need for the users to explicitely define a partitioning function, is the device-unawareness. An instance of the kernel runs on each available device and operates on a different chunk of the input. The system doesn't support device-selection for kernel execution on the basis of the particular algorithm to run or on the peculiar characteristics of the available devices.

Many OpenCL programs, such as FFT and LU factorisation, are made of multiple and potentially iterative kernels. In such cases the scheduling options

are mainly two: either to execute successive kernels sequentially, partitioning each of them on the available resources, or to schedule each kernel on the estimated best device, possibly computing different kernels in parallel. Since the infrastructure proposed in [66] and, more generally, the device-unaware partitioning approaches only considers the first option, scheduling may deliver sub-optimal results whenever the speedup obtained from executing different kernels on different devices is higher than the one resulting from per-kernel cooperative execution. This is particularly true for programs made of kernels differently "shaped" from each other, that is where different kernels are best-scheduled on different devices. Furthermore, partitioning single kernels by distributing work items between multiple resources requires synchronization and communications between them, which can incur significant runtime overhead.

FluidiCL [53] proposes a similar, device-unaware scheduling approach for CPU-GPU systems, based on partitioning individual OpenCL kernels for concurent execution on multiple devices. The partitioning scheme is applied to the OpenCL working space, part of which is assigned to the CPU and part to the GPU. In this way, CPU and GPU are assigned each a subset of the global work-groups to execute. This approach suffers from the aformentioned limitations characterizing every device-unaware scheduling approach.

In [72] a mechanism to predict the performance of computations on single-ISA heterogeneous systems is proposed. Prediction is based on collecting runtime information about instruction- and memory-level parallelism using the CPI stack. Using this information, the approach estimates the relative speedup or slowdown obtained by scheduling the profiled computation on different computing resources. The principal limitation of this apporoach is the inapplicability to heterogeneous systems where the devices have instruction sets different from each other. Multi-ISA heterogeneity is far more likely a popular scenario than single-ISA, especially in CPU-GPU systems. A second limitation consists in the difficulty to employ the Performance Impact Estimation (PIE), which is the "metric" used for scheduling, in situations where different computations are composed each other. In these scenarios, the scheduling policy should take into account the impact on data-transfer due to scheduling computations interacting with each other on different devices. The problem that arises is how to combine PIE with quantities expressed in different units, such as the cost of data-transfer, possibly expressed in time units.

In [73] an algorithmic feature-based classification approach is proposed to determine the *shape* of computations and to consequently deduce the class (CPU or GPU) which a computation belongs to. To reduce the overhead of feature extraction, some information (static features) are collected when a

kernel is compiled while others (runtime features) are computed at kernel execution time. The main problem of this approach is the inability to capture information that comes from considering the program structure and the input or work-item domain size together. For example, estimating the number of cache misses requires features, such as the set of memory access strides, that can only be statically precomputed but for which the evaluation must be delayed at runtime, when the input size is known. In addition, this classification approach leverages the estimation of a *speedup* quantity to determine whether to schedule a computation on the CPU or on the GPU. Whereas the authors state that completion time is not a reliable information to use, completion time is nonetheless an objective metric that allows adapting to highly dynamic systems, such as systems with multiple CPUs and no GPUs or with both integrated and discrete GPUs. It has been shown [18] that integrated GPUs are beneficial to speeding up computations characterised by peculiar combinations of memory and computation boundaries. Since integrated and discrete GPUs are often identical from the perspective of the architecture and of the runtime model with the only exception of the memory subsystem, it may be very difficult to determine a combination of static and runtime features to describe, in terms of the *speedup* quantity, this particular kind of integrated devices. Moreover, a binary CPU/GPU classification like the one proposed can't take into account the potential overhead of copying data whenever two successive kernels are scheduled on different devices. Such an overhead, which may outweigh the benefit of scheduling each kernel on it's own best device, can instead be taken into account in case of completion-time-based classification. In fact, once the cross-device data-copy overhead is computed and the best (lowest) completion time of two kernels predicted, it's trivial to compare the sum of the two with the completion time obtained in scheduling both the kernels on the same device saving the copying overhead. Finally, this feature-based classification approach leverages Support Vector Machines (SVMs), with a Gaussian Kernel to predict whether an algorithm would run faster on the GPU or on the CPU. An important limitation of this classification method is that it is very hard, if not impossible, to interpret the model created by an SVM, which has to be accepted or rejected only looking at the prediction results.

These considerations are supported by [42], where an interesting set of approaches to scheduling and partitioning dataflow graphs is discussed and evaluated, showing that traditional greedy per-kernel policies such as data-locality preservation are insufficient to accurately determine an optimal/sub-optimal scheduling strategy. This work underlines the importance of taking into account cross-device data-transfers to implement an accurate scheduling

policy for a wide range of workflows.

In [37] a case study is discussed for estimating the completion time of workflows based on algorithmic features extraction and on the exploration of a set of estimation functions (regression models) to detect the ones that maximize prediction accuracy. Whereas completion time is indisputably the most flexible metric to employ in a scheduling policy, since it can be naturally composed with other aspects that may influence the scheduling decision (e.g. data-transfer latency), the work proposed exclusively focuses on a restricted set of sequential tasks, drawn from the UCI Machine Learning Repository, whose features are already available as parts of the information contained in the dataset. We claim that in a situation where heterogeneous platforms are becoming a standard for the execution of arbitrary computations, genericity of algorithms and automatic feature extraction are two of the most relevant research aspects for successful scheduling strategies.

In [10] a machine-learning approach is proposed for dynamic task scheduling and load-balancing of batch programs on heterogeneous platforms. To tune scheduling policy dynamically, the approach relies on periodic execution of a *pilot program* to estimate the throughput of a batch of tasks before scheduling the whole batch. The result of successive executions of the pilot programs are collected, stored and used to improve the scheduling quality over time. The application of the approach considered is very limited, since it is specifically designed for, and validated against, batch processing of financial option pricing programs. Nevertheless, the authors particularly stress a point that we consider fundamental, which is the efficiency of the scheduling policy. In particular, the approach takes also into account the overhead introduced by the pilot program and tries to balance this overhead with the potential speedup that can be achieved with the information provided when running the pilot.

Chapter 3

Conclusions

Despite the range of research on raising abstraction over OpenCL and, more generally, over heterogeneous platform programming and execution, OpenCL is still the most popular and widespread solution.

We see two main contributing factors. The first is the complexity of many high-level programming models and of the set of language constructs exposed to express parallelism. In some cases, the programming model provided is implemented through a brand-new set of control-flow constructs, data-types and functions that programmers have to learn from scratch. In other terms, the integration of such models into the host language is very limited. Since the complexity of the model is close to the one that characterizes OpenCL, the developers tend to prefer OpenCL low-level programming, which is as least more popular, supported and consolidated.

In other cases, the integration in the host language is not particularly limited, but it's inexpressive. Expressive models expose concrete constructs that can be intuitivly mapped to concepts of the abstract domain in which programmers are supposed to work. Some solutions tend to employ pre-existing constructs of the host language that are instead difficult to map to the typical abstract concepts of parallelism, making the application of such solutions hard and counter-intuitive.

The second factor that limits the spread of high-level programming solutions is the lack of flexibility. Many parallel programming languages and libraries, especially those based on skeletons and programming patterns, focus exclusively on simplifying coding by making it as abstract as possible. If from one side this increases productivity in developing common, structured parallel programs by composing building blocks, from the other it narrows the set of algorithms that can be expressed. Whenever a computation lies outside the boundaries defined by the set of available patterns, programmers have to switch to a different programming solution.

From the scheduling point of view, device-unaware approaches that consider the available computing resources as an homogeneous set of nodes hardly fit today's heterogeneity. Whereas these approaches can deliver good performance in the case of task-based applications and provide load balancing, assuming task-partitioning of computations and exploiting devices without considering their specific characteristics may incur in poor performance for generic algorithms on many heterogeneous systems. Some parallel programs are not easily partitioned into computation units; in many cases, the explicit programmers' intervention to define task subdivision may be required. In addition, in case of programs made of two (or more) interrelated computations, device-unaware approaches generally try to balance the load on the various devices, without considering whether scheduling both the computations on the same device delivers higher performance thanks to the *particular* suitability of the device to accelerate the *specific* computations to run. For example, programs apparently massively parallel may be faster on a CPU if they contain a sensible number of divergent branches, which is a well-known cause of performance degradation on GPUs.

Recently, the market has been characterized by the availablity of an increasing variety of different devices and of cross-device interconnection solutions, such APUs from AMD, programmable on-chip GPUs with CPU-GPU ring bus from Intel, OpenCL-enabled FPGAs from Altera, Multi-GPU SLI interconnection and the Intel Xeon Phi coprocessor. In terms of hardware characteristics, some devices are subtly different from each other but are possibly characterized by quite distinct subsets of computations that each can execute outperforming the others [18]. Given this, an analytical characterization of devices is difficult to achieve, other than being inflexible to the dynamicity of systems configurations.

We see in machine-learning the ideal candidate to cope with this dynamicity, as well as with the impossibility to assume certain structures of the computations to run and with the difficulty to capture the differences between devices on the basis of their hardware specification.

The major limitation of the machine-learning approaches for heterogeneous CPU-GPU systems considered is the metric used for scheduling and the complexity of the method employed. While estimating the value of quantities like speedup [73] and performance impact [72] may produce reliable results in some cases, such quantities hardly capture a variety of subtle causes of differences in performance between integrated and discrete GPUs. In addition, these quantities are not suitable to be used together with other "concrete" information, such as data-transfer or computation-launching overhead, in order to refine the scheduling policy to improve overall performance.

3.1 Thesis proposal

Heterogeneity is on track to be the dominant aspect of parallel computing in the years to come. The increasing popularity of heterogeneous multi-device systems, from desktop computers to mobile phones, gives rise to the problem of giving access to such a computing power to a wide community of programmers and to a nearly unbounded set of programs.

The state of the art in raising abstraction over heterogeneous programming show various limitations, such as loose integration in the host language, lack of flexibility to express complex computations and inhomogeneity within the programming model.

Most of the recent research in scheduling on heterogeneous multi-device systems addresses the problem of load-balancing in a device-unaware fashion, instead of focusing on device-aware classification of parallel computations. We think that being able to specialize a device for the execution of the specific computations for which it is the "best" (e.g. the most efficient) device in the running system plays a key role in exploiting the increasing heterogeneity of today's platforms.

Our Proposal consists in investigating the today's challenges and open problems in programming, scheduling and execution on heterogeneous platforms. In particular, we research the possibilities of leveraging the abstraction and the introspection capabilities of Virtual Execution Environments and of languages running on Virtual Machines to design a high-level, homogeneous parallel programming layer and to transparently collect code and platform information in order to support efficient, dynamic and device-aware scheduling and execution of parallel programs.

Part II

FSCL: a framework for high-level parallel programming and execution

Chapter 4

Overview of the research

The state of the art in programming, scheduling and execution on heterogeneous platforms discussed in chapter 2 constitutes the starting point of our Ph.D. research.

We consider the integration of parallel models into pre-existing, popular languages a key strategy to ease heterogeneous programming, cutting the learning curve and allowing the programmers to feel comfortable with moving from traditional, sequential programming to parallel computing. For this reason, we start investigating the most suitable language to host an abstract and expressive parallel model. In chapter 5 we introduce F#, which is the language we choose to serve as host. We present the most relevant aspects and features of F#, particularly focusing on those suitable to be reused to express parallel programming constructs and their composition.

Once choosing the host language, we start designing an abstract and flexible model to express parallel computations from within F#. We discuss the details of this model, called *FSCL kernel language*, in chapter 6.

Whereas F# is the host language for our model, OpenCL is used as the ground layer for scheduling and execution. We leverage OpenCL to build executables starting from high-level F# programs and to interact with the device driver for the execution and coordination of parallel computations.

Given F# as a source language and OpenCL as the target layer, we research a way to map our high-level programming model to OpenCL. In chapter 7 we present the *FSCL compiler*, which is the result of our efforts to define a strategy to trasparently map high-level parallel programming constructs to OpenCL kernel- and host-side.

Finally, given a model for high-level parallelism and a way to translate the constructs of this model into OpenCL, we investigate a scheduling approach that leverages the differences among the devices in a platform to dynamically execute computations on each one's most efficient device. As a side research,

we also invest our efforts in performing efficient code analysis and device characterization in order to maintain the performance of the abstraction layer comparable to low-level OpenCL execution. The result is discussed in chapter 8, where we present the the *FSCL runtime*, and in chapter 9, dedicated to the scheduling approach for multi-device heterogeneous systems.

As a whole, the result of our research on a model to raise abstraction over programming, scheduling and execution for heterogeneous platforms consists in the *FSCL framework*. From a high-level view, the framework is composed of three main components: a programming model, a compiler and a runtime. Each of these components addresses a subset of the challenges on which our research is based.

From the perspective of addressing the problems of abstraction, flexibility and efficiency of the programming and execution layer, the framework can be described in terms of four major features, listed below.

Abstract and expressive programming model From the programming point of view, our research introduces a model that brings OpenCL programming into F# and raises abstraction over regular low-level OpenCL coding, exploiting pre-existing language constructs and coupling abstraction and expressiveness to enhance productivity while preserving fine-grain control on coding whenever required.

Automatic per-device optimisation To obtain the highest performance, OpenCL requires coding different versions of a kernel, each optimised for a particular type of device. Starting from the abstract programming model exposed, the FSCL framework is able to transparently generate optimised OpenCL code depending on the particular device chosen for execution.

Transparent and efficient device-aware scheduling One of the most pressing problems induced by today's heterogeneity is to detect the set of computations that a specific device can execute outperforming the other devices in a platform. As a result of our research, the framework provides a transparent device performance estimation and a scheduling policy to help programmers to exploit the heterogeneity of multi-device systems, selecting the best device for each computation to run.

Transparent and efficient parallel execution In OpenCL, the developers have to spend noticeable efforts in host-side coding to setup and coordinate the execution of kernels. The FSCL framework automates the entire process, from resource allocation to scheduling and coordinating kernels, allowing the programmers to focus exclusively on defining the *content* of parallel computations.

One problem that can be introduced by a too abstract object model is the lack of flexibility, which makes it difficult or impossible to express algorithms that do not fit the restricted set of built-in patterns. Similarly, limiting the flexibility of a parallel execution model results in a loose control over resource allocation and usage whenever such a fine-grained control is required. Therefore, our research concentrates on providing a good balance between abstraction and flexibility for all the major features of the framework, possibly introducing multiple abstraction levels at which programmers can see and exploit a feature.

Chapter 5

F#: a flexible multi-paradigm language

In this chapter we present F#, which is the host language of our parallel programming model as well as the language used to develop the entire FSCL framework. First of all, we explain what F# is and why we select it as the host language for our model, showing the advantages to use a functional/object-oriented language to expose an abstract, expressive and flexible parallel programming API. Then, we introduce some of the most widely used F# constructs to allow a deeper comprehension of the object-model exposed by the FSCL framework, of the code-analysis approach for scheduling and of the examples proposed throughout this Thesis.

F# is an open-source, multi-paradigm programming language built on top of the Common Language Runtime (CLR). It shares a core language with *OCaml* [62], which in turn comes from the *ML* family of programming languages. F# also shares some features with *Haskell* [51], which is a purely functional programming language.

The main difference between F# and many other functional languages is that it completely integrates imperative and object-oriented paradigms inside the functional one. Programming with F# tends to be more object-oriented than in other functional languages. Programming also tends to be more flexible. F# embraces techniques such as dynamic loading, dynamic typing, and reflection, and it adds high-level and expressive programming features like quotations and active patterns. Furthermore, F# bridges the gap between static and dynamic typing and between compiled and interpreted languages, combining the programming styles typical of dynamic languages with the performance and robustness of a compiled language.

F# allows writing type-safe functional programs in a succinct and efficient

way, while combining the advantages of typed functional programming with a well-supported runtime system. For these reasons, in the last few years F# has become a very popular, cross-platform first class language.

Since the object-model that FSCL exposes to code parallel computation is heavily based on functions and function composition (section 6.2), one of the most important reasons why we choose F# resides in its functional nature. In F# functions are *first-class objects*: functions can be passed as parameters, returned from a subroutine and assigned to variables.

Given that functions are the basic building-blocks of F#, expressive syntax constructs are provided to work with functions. As an example, F# defines an operator for function composition, which is the operator (≫). Given two functions f and g, it is straightforward to express the *function composition* of f and g (i.e. to build a function that represents the composition of f and g):

```
let f el = // ...
let g el = // ...
// Build the composition of f and g
let composition = f >> g
// Apply the composition
let result = composition(input)
```

Function composition is a concept that can be intuitively mapped to parallel programming, for example to model the pipeline paradigm in stream parallelism (i.e. the *n-th* stage of a pipeline applies a function to the result produced by the *(n-1)-th* stage). In pure data parallelism, function composition can express a dependency or data-flow between two computing nodes.

F# *collection functions* represent another set of F# built-in constructs that can be expressively reused to define and compose parallel computations. Native F# functions like *Array.map*, *Array.scan*, *Array.reduce* directly map to some of the most popular parallel skeletons.

The choice of F# to host the our parallel programming model is also driven by a powerful language construct called *quotations*. Thanks to quotations, it is possible to retrieve the *AST* (*Abstract Syntax Tree*) of an arbitrary expression at runtime. Once obtained, the AST can be visited, analyzed, transformed and evaluated.

Another reason for choosing F# is that it runs on *.NET* and *Mono*, which are both widely used and well consolidated software frameworks. Programs written in .NET/Mono execute in a software runtime environment called *Common Language Runtime* (*CLR*). The *CLR* provides the appearance of an application virtual machine, providing benefits such as isolation, standardization, and portability. Moreover, the CLR implements many important services such

as security, memory management, and exception handling. Not only F# runs on .NET/Mono, but it is also fully integrated with Visual Studio and Xamarin Studio, which are two of the most advanced integrated development environments. Thanks to this integration, F# programmers can benefit of a consolidated set of tools and facilities, such as code completion, intellisense, visual debugging, code refactoring and visualization of type errors at coding time. Finally, F# is open-source and fully supported across Windows, OSX and Linux.

5.1 Language constructs

In this section we give an overview of programming in F#. This is not intended to be an exhaustive guide to F# programming but only an introduction to the subset of F# constructs used in our parallel programming model and in the examples proposed throughout this Thesis. For more details on the F# programming language, refer to the bibliography [69].

5.1.1 Variables and functions

The main keyword of F# is *let*: it is used to define variables, functions, and procedures. This keyword simply binds a value to a name. The left hand of a let binding can be a "simple" variable but also a function. In the following listing the value of the expression *2 + 3* is bound to the variable *v* and a function *f* is defined. This function takes an integer argument and multiplies it by 2.

```
let v = 2 + 3
let f it = it * 2
```

Since functions in F# are first-class citizens, even the definition of a function can be considered as a binding. In fact the function *f* can be also defined as follows:

```
let f = (fun a -> a * 2)
```

This way to define the function *f* emphasizes the fact that *f* can be considered a variable bound to a functional value, whose type is *function from integer to integer* (indicated with the notation $int \rightarrow int$). Furthermore, it allows to introduce another important feature of F#, which is *lambda expressions*.

The syntax of a lambda expression is the following:

```
fun arg_1 arg_2 arg_N -> body
```

A lambda expression can be considered the definition of an anonymous function. To point out the equivalence of functions and any other kind of values, a lambda expression can also be viewed as an instance or value of some function type $T \rightarrow U$. As like as any other value, lambda expressions can be passed as parameters, used as return values and more.

F# provides two styles to pass arguments to a function. The first is known as *curried form*, which is the most common approach to specify the arguments of a function in functional programming. In the curried form, function parameters and function call arguments are separated by whitespaces.

The second style is called *tupled form*, which is the most popular form in object-oriented and imperative programming languages. In tupled form, parameters and arguments are comma-separated. In the following listing the curried form is used to declare *foo*, while *bar* is declared using the tupled form.

```
let foo a b c d = (a * b) + (c * d)
let bar (a,b,c,d) = (a * b) + (c * d)

let foo_result = foo 2 4 5 8
let bar_result = bar (2,4,5,8)
```

From the F# language definition point of view, tupled and curried functions are not different from each other. In fact, a tupled function can be seen as a curried function with one only parameter of a particular *tuple type*. For example, the function *bar* in the previous example can be seen as a function that takes an argument of type "tuple of four integers" and returns an integer.

Currying is a concept that goes beyond F# functions definition. In mathematics and computer science, currying refers to the technique of transforming a function that takes multiple arguments into a chain of functions each of which takes one only argument. F# does exactly this: when the programmer defines a curried function with N arguments F# represents it as a chain of N one-argument functions.

Consider the following curried function:

```
let foo a b c = (a * b) + c
```

To exemplify how F# represents this function we re-write it using lambda expressions, allowing to view it as a chain of one-argument functions:

```
let foo = (fun a -> (fun b -> (fun c -> (a * b) + c)))
```

This rewriting changes neither the semantic of the function nor the way it can be applied.

Currying is tighly related to the concept of *partial application*, which is a powerful mechanism widely employed in our research. To understand partial application, we consider again the curried function:

```
let foo a b c = (a * b) + c
```

Since *foo* is represented as a chain of one-argument functions, we can apply it to one, two or three arguments. In other terms, all the following expressions are valid:

```
let a = foo 2
let b = foo 2 3
let c = foo 2 3 4
```

Anyway, only the last application returns the expected value $(2 * 3) + 4$. The other expressions constitute a *partial application* of *foo*, since only a subset of the three arguments has been provided. Once partially applied, a function can be "finished to be applied" by supplying the remaining arguments to the result of the partial application, as shown in the following listing:

```
// Foo has three arguments, partial application
let partial = foo 2
// Finish to evaluate foo
let result = partial 3 4
```

Among the other benefits, partial application allows to naturally and clearly express functions and function calls (applications) composition. As already said, F#, functions can be composed using the operator $\gg$, which is defined as follows:

```
(>>): ('a -> 'b) -> ('b -> 'c) -> 'a -> 'c
```

Given two functions f and g, their composition is expressed as:

```
f >> g
```

As like as in many other languages, function calls can be chained using the well-known syntax:

```
g(f(argument))
```

F# also introduces a set of *pipe operators* that allows to chain function calls in a left-to-right fashion, as illustrated below.

```
argument |> f |> g
```

Sometimes a function takes the result of another function as well as some other input values. For example, consider a function f that performs a mapping on an input array and a function g that takes the resulting array and multiplies each element for a specific scalar value.

```
let f (input:int[]) =
   let output = Array.zeroCreate<int> (input.Length)
   // Compute
   ...
   // Return
   output

let g (input:int[]) (m:int) =
   Array.map (fun a -> a * m) input
```

The two functions can be composed as discussed so far:

```
let multiplier = 3

let composition = f >> g
// Regular call
let result = composition someArray multiplier
// Pipelining
let result = (someArray, multiplier) ||> composition
```

Alternatively, we can partially apply g to the multiplier and compose the result of partial application with f, as follows:

```
let multiplier = 3
// Compose f with the partial application of g
let composition = f >> (g multiplier)
// Regular call
let result = composition someArray
// Pipelining
let result = someArray |> composition
```

Partial application is extremely common when functions require a set of input values that are known at different times, and plays an important role in the entire parallel programming language proposed in our research.

Collections and collection functions

The F# language introduces three major collection types: *Array*, along with multi-dimensional versions *Array2D* and *Array3D*, *List* and *Seq* (Sequence). Arrays are traditional index-based collections of values, each of which can be read and set. Lists are linked chains of values, each of which can be only read. As like as arrays, lists can be accessed by index[1]. Finally, sequences are generic enumerable sets of values.

For each of these types, F# exposes a set of functions to enable programmers to work on collections using the functional paradigm. In the following list we describe some of the collections functions exposed by the *Array* type:

Array.map f a Builds a new array whose elements are the results of applying f to each of the elements of a.

Array.map2 f a b Builds a new array whose elements are the results of applying f to the corresponding elements of the two arrays pairwise.

Array.filter f a Returns a new array containing only the elements of the input collections a for which the given predicate f returns true.

Array.reverse a Reverses the input array a.

Array.sortBy f a Sorts the elements of the array a into a new array, using the given projection f for comparison.

[1]Accessing lists by index incur an higher cost if compared to accessing arrays, since accessing the i-th element of a list requires to access the previous i-1 elements

Array.reduce f a Applies *f* to each element of the array *a*, threading an accumulator through the computation. If the length of *a* is *n*, then reduce computes *f (... (f (f a[0] a[1]) a[2]) ... a[n-1])*.

Collection functions can be composed to perform non-trivial computations on collections. The following sample shows how to combine collection functions to compute the histogram of an image. In the first step, the luminance is extracted from RGB components using the *map* function. In the second step, the luminance values are grouped by key through the *Seq.groupBy* function, where the key is the index of the bucket where to store the value. The size of each bucket is then computed and, finally, the resulting collection if converted to array (since Seq.groupBy returns an instance of type *Seq*).

```
let image = // an array of pixels

let histogram =
   image |>
   // Get Luminance from RGB
   Array.map(fun p -> (0.2126 * p.R + 0.7152 * p.G + 0.0722 * p
     .B)) |>
   // Classify into N buckets
   Array.groupBy(fun l -> (int) (l * (float)N)) |>
   // For each group compute the size
   Array.map(fun key elems -> Seq.length element)
```

Given their flexibility and the ease-of-use, F# collection functions are very popular in the F# programming community.

5.1.2 Data-types

F# defines some data types that are extremely unseful and productive. The first is the *tuple type*. The following example illustrates a tuple containing three values. The first an integer, the second is a float and the third is a string.

```
// The type of tup is (int * float * string)
let tup = (20, 2.0, "hi")
```

The type of a tuple is represented with the notation $type_1 * type_2 * ...type_n$.

In F#, programmers usually can't change the value bound to a variable. In other terms, variables are *immutable*. There are two ways to define variables whose value can change during their lifetime. The first is by marking a variable

as *mutable*, which allows new values to be assigned to it using the operator "←":

```
let mutable v = 0
v <- 10
v <- 142
```

Another common mechanism to implement a mutable state is known as *reference cells* (or *ref cells*). From the programmer's perspective, ref cells play much the same role as pointers in imperative programming languages.

Ref cells can be defined by using the keyword "*ref*". The value hold in a ref cell can be accessed using the *de-reference operator* "*!*" and can be changed using the *assignment operator* "*:=*", as shown in the following example.

```
let cell = ref "hi"    //reference cell containing strings
cell := "hi there"     //now cell contains "hi there"
let s = !cell          //s = "hi there"
```

Another frequently used construct provided by F# is the *discriminated union*. A discriminated union represents a finite, well-defined set of choices. Discriminated unions often constitute basic blocks to build up more complicated data-structures including linked lists and trees. The following listing illustrates the syntax reserved to discriminated unions:

```
type NameOfTheType =
 | Case1
 | Case2 of dataType1
 | Case3 of dataType2
 ...

//Declare variables of type NameOfTheType
let v1 = Case1
let v2 = Case2(value of type dataType1)
...
```

It can be said that discriminated unions do not appear much different from enum values. This construct is instead more powerful, since it can be parametric (generic type) and can be defined recursively.

For example, the following union recursively models a binary-tree data-structure, where each node contains an integer and a string:

```
type Tree =
  | Leaf of (int * string)
  | Node of (int * string) * (tree * tree)
```

A particular example of discriminated unions is the *optional type*, whose instances represent *optional values*. The instances of an optional type either store a value or don't store anything. The following example shows the definition of the optional type (built-in in the F# core) and its usage.

```
//'a is a generic type parameter
type Option<'T> =
    | Some of 'T
    | None

let v1 = None
let v2 = Some(2)            //Type is Option<int>
let v3 = Some("hi there")   //Type is Option<string>
...
```

5.1.3 Quotations

F# quotations is a feature that allows retrieval of AST of an arbitrary F# expression at runtime. In particular, thanks to F# quotations the programmer can analyze the tree representation of the code and give it an arbitrary interpretation. The interpretation of a quoted expression is also called *evaluation*.

By wrapping an expression into "<@" and "@>" the programmer "forces" F# not to evaluate the expression but to build and return its AST.

For example, in the following case:

```
let sum = <@ 2 + 3 @>
```

F# doesn't evaluate the expression 2+3 (assigning 5 to sum). Instead, *sum* is assigned to an instance of the class *Quotations.Expr<int>*, whose (printed) value is:

```
Call (None, Int32 op_Addition[Int32,Int32,Int32](Int32, Int32),
      [Value (2), Value (3)])
```

The class *Quotations.Expr* represents the generic type of AST nodes. The actual kind of the expression contained in a node can be discovered using pattern-matching. The F# namespace *FSharp.Quotations* defines a set of patterns to check the specific kind of expression (if the expression encodes a method call, a variable reference, the creation of a new object, a for/while loop, etc.).

The quotations mechanism comes with a feature, called "splicing", that allows the composition of quotations. Thanks to the splicing operator "*%*", a complex quoted expression can be built starting from quoted sub-expressions. The following listing shows how a quoted expression can created either by defining it in one go or by composing independent sub-expressions:

```
//The expression (2 + 3).ToString() is defined at one go
let sumToString = <@ (2 + 3).ToString() @>

//First define the expressions ''2'' and ''3''
let two = <@ 2 @>
let three = <@ 3 @>
//Then compose them to form a sum
let sum = <@ %two + %three @>
//Refer sum and call ToString
let sumToString = <@ %sum.ToString() @>
```

Quotations are one of the major reasons why we choose F# to host our high-level parallel programming model and to build the entire programming, scheduling and execution layer. In fact, thanks to the ability to unobtrusively obtain the abstract representation of code at runtime, we can transparently analyse the code of high-level programs, perform optimizations and semantic controls and produce a low-level representation of the program to schedule and execute.

5.2 Conclusions

In this chapter we presented F#, an expressive multi-paradigm, open-source and cross-platform language. We discussed the advantages that make F# the most appropriate language to host our model for programming heterogeneous platforms. The most relevant advantage resides in the set of constructs it provides, which can be naturally mapped to some of the most common concepts of parallel programming, such as composition of parallel computations and parallel skeletons. Moreover, F# comes with a nearly unique feature called *quotations*, which allows the retrieval of the AST of arbitrary expressions at

runtime. This means that it is possible to get "for free" the abstract representation of the body of a function or of a class member. Thanks to this feature, code analysis, optimizations and transformations can be performed transparently, without the need for external programs, such as code parsers, precompilers or rewriting tools.

Chapter 6

FSCL kernel language

The FSCL Kernel Language represents the result of our research in integrating OpenCL parallel programming in a modern and powerful language, allowing the expression of parallel applications in a flexible, safe and expressive way. From the FSCL framework point of view, the kernel language is the layer that exposes data-types, operators, functions and, globally, an object-model, to write parallel programs from within F#. In this chapter we present the FSCL programming model, we discuss its integration into F# and we summarize the most relevant language features on which the model relies.

6.1 Introduction and main approach

In all the languages, APIs and libraries for parallel programming, one of the most important aspects to consider is the balance between abstraction and flexibility. An abstract and expressive language exposes an intuitive model and a set of programming constructs that naturally shape the domain the language is designed for. This makes the language easy to learn and intuitive to use. The major drawback of the close fitting to a specific domain is the lack of flexibility, which makes the language unfit to express computations that cross the boundaries of the domain. To the contrary, low-level languages offer a more generic model and often a broader set of data-types and functions, which turn into a longer learning-curve and a higher level of programming difficulty. Nonetheless, the flexibility of such languages makes them suitable to express many computations that are difficult or impossible to code at high abstraction levels.

As discussed in section 2.2, many high-level parallel programming libraries [14, 22, 64] based on skeletons and patterns are characterized by a strong inflexibility. Whenever the structure of a computation can't fit the provided

patterns, the computation has to be coded relying on a lower-level approach, such as OpenCL or native threading. A question that arises when developing an application made of multiple, parallel computations, is how to connect the computations coded with the high-level library to the ones that require a low-level, more flexible model. Since in many cases interoperability with computations developed beyond the library boundaries is not supported, programmers are forced to fallback to the low-level approach to code the *entire* parallel application.

To increase the flexibility while preserving an overall high level of abstraction, some parallel libraries provide multiple levels of abstraction [48, 57]. Unfortunately, these libraries are characterized by an inhomogeneity between the models exposed at different levels. For example, Intel TBB high-level programming is based on custom constructs/functions (i.e. parallel_do, parallel_for), while low-level model is based on explicit flow-graph creation and interconnection. This inhomogeneity tends to incur a longer learning-curve and make it difficult to fully understand and control compositionality of computations expressed at different abstraction levels.

The FSCL programming layer focuses on the problem of balancing abstraction and flexibility by offering multiple levels of programming, each of which characterized by a particular combination of abstraction and flexibility. Unlike other solutions, the various levels of abstractions are based on the same functional object-model. Computations coded at different levels are therefore fully composable with each other in a natural way.

Another aspect to consider when integrating a new model in a pre-existing language is the set of native constructs that can be reused, without introducing new language features but providing a different interpretation for the existing ones. In our research we focus on the identification of those language constructs that are expressive for parallel programming, which means they can be easily and naturally mapped to some of the most important concepts of parallelism, such as skeletons, stream execution and parallel data-flow. Whereas reusing contructs saves programmer from learning a new set of functions and types, reusing *expressive* constructs also allows to feel comfortable to program in the new domain.

From the perspective of data-types, *array* and *sequences* are extremely popular collection types, widely used in regular, sequential programming. F# exposes a set of native functions to work on collection types, such as *Array.map*, *Array.reduce* and *Array.scan*. Besides being popular in the F# programmer community, these functions are also very expressive for parallel programming, since they directly map to some of the most common data-parallel skeletons. In addition, as discussed in section 5.1.1, F# exposes a set of function composition

operators that can be used to chain functions and function calls inline. As like as collection functions, these operators represent built-in constructs that map naturally to some fundamental concepts of parallel programming, such as stream pipelines and dependencies in data-flow graphs.

In designing a parallel programming model to be integrated into F#, we leverage this set of native types and functions, enabling programmer to write parallel applications (re)using most of the language constructs they are already accustomed to.

6.2 Kernel language programming and object model

The FSCL kernel language programming model is based on three principal abstract concepts: *computing elements*, *computing expressions* and *computing programs*.

Computing element A computing element is a function that represents the unit of execution of a parallel program. A computing element is atomic, in the sense it cannot be divided into sub-units.

Computing expression A computing expression is recursively defined as either a single computing element or a composition of computing expressions.

Computing program A computing program is an user-defined host program which coordinates the execution of one or more computing expressions.

In the rest of this Thesis, we refer to FSCL computing elements also using the term "kernels", because of their analogy to the OpenCL units of parallel computing. Similarly, we refer to a computing expression using the term "kernel expression" or, whenever not confusing, "expression"[1].

The generic structure of an FSCL application is illustrated in figure 6.1. The programming model is based on an FSCL program that execute one or more computing expressions. The concept of *FSCL program* is analogous to the OpenCL concept of host-side, which generally corresponds to the *main* function in a console application or to some other user-defined functions. The main program orchestrates (composes) kernel expressions in an imperative/object-oriented context. Data produced by a kernel expression is possibly bound to variables and passed to successive kernel expressions.

[1]In some contexts, *expression* may refer to an element of the generic domain of F# expressions

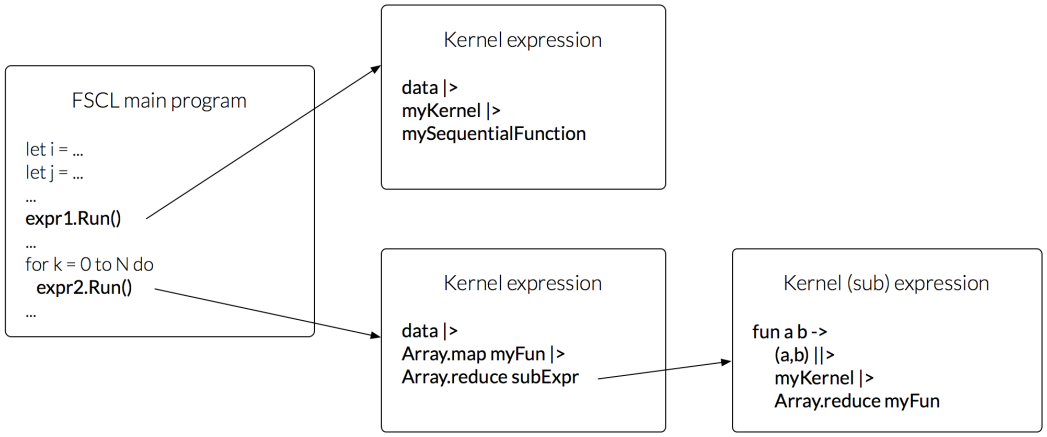

Figure 6.1: Generic structure of an FSCL application

Each computing expression can be a single computing element or a composition of elements. More precisely, by definition 6.2, the arguments of a computing expression can be (sub)expressions in their turn, leading to a finite but possibly unlimited hierarchy of expressions.

Unlike the composition of expressions in the imperative/object-oriented FSCL program, which is user-defined, the composition of elements inside a computing expression is intrinsic in the specific type of composition used (section 6.2.2). In other terms, execution, coordination and appropriate data-passing among multiple FSCL expressions in an FSCL program is on the user, while coordination and data-passing among computing elements in the expression is built-in and implicit in the semantic of the composition.

Inside a computing expression, some of the elements composed effectively represent parallel computations, while others can be user-defined regular (sequential) functions (e.g. *mySequentialFunction* in figure 6.1).

In section 6.2.2 we describe two major ways to apply this model, each of which represents a different combination of abstraction and flexibility in composing kernels.

6.2.1 FSCL computing elements

In the FSCL kernel language object-model computing elements are expressed as particular F# functions. Throughout the rest of this section and except where stated differently, with the term "function" we refer to either F# module functions, static/instance methods or lambdas.

The FSCL programming model exposes three types of computing elements:

custom kernels, *collection kernels* and *sequential functions*.

Custom kernel A custom kernel is an F# function that represents an user-defined computation that one-to-one maps to an OpenCL kernel.

Collection kernel A collection kernel is a native F# function working on collection types (section 5.1.1) used to express a data-parallel skeleton.

Sequential function In FSCL, a sequential function is a native or user-defined function that is neither a custom kernel nor a collection kernel.

Collection kernels are used to express skeletal parallelism in a simple but effective way. The semantic of such functions is intrinsic and they can be used with no changes from regular, sequential F# programs. Custom kernels instead represent parallel computations that go beyond the boundaries of a skeleton or pattern and allow the flexibility of low-level OpenCL kernel programming.

These two classes of functions represent *different levels of abstraction to express parallel computations*. Custom kernels represent the lowest level of abstraction and the highest flexibility, since they preserve the power of OpenCL C programming to express a very wide range of algorithms. Collection kernels instead allow hiding from OpenCL the programmer, raising the abstraction at the price of a limited flexibility.

Functions that do not belong to one of these classes are treated as sequential functions which run on the CPU. The choice to support sequential functions is mainly driven by the will to provide the highest composition capabilities. In many complex computations there are in fact some steps that are inherently sequential. Supporting sequential functions allows the composition of these steps with parallel computations without the need to "slice" a computing expression.

Custom kernels

At a glance, custom kernels are OpenCL kernels coded in F#. Depite some minor syntax differences, every computation that can be coded as an OpenCL C99 kernel can be expressed using an F# function as well.

In listing 6.1 and 6.2 we show, respectively, a vector addition kernel written in OpenCL C99 and in FSCL (F#).

```
_kernel void VectorAdd (_global float* a, _global float* b,
  _global float* c, int length) =
    int gid = get_global_id(0)
    if(gid < length)
      c[gid] = a[gid] + b[gid]
```

Listing 6.1: OpenCL C99 kernel for vector addition

```
[<ReflectedDefinition; Kernel>]
let VectorAdd(a: float32[], b: float32[], c: float32[], wi:
  WorkItemInfo) =
    let gid = wi.GlobalID(0)
    if gid < a.Length then
      c.[gid] <- a.[gid] + b.[gid]
```

Listing 6.2: FSCL custom kernel for vector addition

The major differences between OpenCL C kernels and FSCL custom kernels are the custom attribute *Kernel* associated with the F# function, that allows the FSCL framework to recognize it as a kernel, and the additional function parameter of type *WorkItemInfo*, which holds all the information related to the work-items space, such as the global/local work size and the space rank (section 2.1). It would be possible to avoid this parameter and to encode each work-space-related information using a function as in OpenCL C, preserving syntax similarities. Nevertheless, passing this information using an additional parameter makes it easier to fallback to multithread execution and to enhance debugging capabilities, as discussed in section 8.4. In addition, the presence of such a parameter is used to determine if a lambda function represents a kernel. In listing 6.3 we show the vector addition kernel expressed using a lambda function. Since programmers cannot associate the (static) *Kernel* attribute to the lambda, the FSCL kernel language requires the presence of a *WorkItemInfo* parameter to determine if a lambda represents a custom kernel. Thanks to the introspection capabilities offered by the CLR, the FSCL framework is able to inspect the functions parameters/arguments and to detect whether a lambda represents a kernel, giving programmers an additional construct which is particularly useful and productive to express lightweight computations.

```
fun(a:float32[], b:float32[], c:float32[], wi:WorkItemInfo) ->
    let gid = wi.GlobalID(0)
    if gid < a.Length then
      c.[gid] <- a.[gid] + b.[gid]
```

Listing 6.3: FSCL custom kernel for vector addition expressed as a lambda

The FSCL framework exposes the entire OpenCL kernel language object-model[2][3], allowing programmers to write arbitrarily complex OpenCL kernels without escaping the F# environment. Even though the programming abstraction level is the same of OpenCL, coding kernels as F# functions introduces some advantages, such as type-checking and type-safety, intellisense and code completion. In addition, FSCL introduces some higher-level constructs, listed below, that can be used in coding custom kernels without compromising flexibility:

Intrinsic array length When coding OpenCL C kernels, the length of each array must be explicitely passed as an additional parameter. In FSCL, programmers can code F# kernels without the need to pass the length of each input/output array. Whenever the length of an array is required in the kernel body, programmers can use the well-known *Length* property or *GetLength* method.

Multi-dimensional arrays The OpenCL specification doesn't support multi-dimensional array parameters (i.e. pointer-of-pointers). Working on conceptually multi-dimensional data in OpenCL requires to spent efforts to calculate the proper index to access memory. In FSCL, programmers can instead use native, multi-dimensional arrays (*Array2D*, *Array3D*).

Ref variables Reference cells can be employed to pass information to/from the kernel without using arrays. Conceptually a ref cell represents the storage for a value, which maps expressively to singleton arrays used in OpenCL whenever the result of a kernel is a single value (i.e. an array containing one only element).

Structs and tuples Structs and tuples containing primitive or structured fields can be passed to F# kernels and can be created and used inside the kernel body[4]. Tuples are particularly expressive as a shorthand rep-

[2]In particular, the OpenCL specification 1.2

[3]Currently only the part of the OpenCL API relative to images is not supported

[4]OpenCL supports structures containing primitive or struct types

resentation for very popular data types to process, such as 2D/3D points in space and key-value pairs.

Return types The OpenCL specification demands that kernel functions return void. FSCL removes this restriction and allows F# kernels to return any value. As discussed in the rest of this chapter, allowing custom kernels to return values plays a key role in compositionality.

Generic kernels FSCL allows the definition and use of generic, custom F# kernels. For example, programmers can write a parallel matrix multiplication once, based on generic element types supporting addition and multiplication operators. Once defined, such a kernel can be executed on matrices containing integer, float and other numeric elements.

Since all these high-level constructs and facilities are optional[5], they do not restrict the flexibility of custom kernels and can instead improve programmers' productivity and shorten the kernel development process.

Among all these high-level features built on top of the OpenCL specification, the most relevant is the ability for a kernel to return a value, since this feature makes it possible for custom kernels to be composed with each other and with other computing elements, such as sequential functions and collection kernels.

Collection kernels

In section 5.1.1 we introduced some of the built-in F# functions to work on collections of data. As already said, these functions are particularly popular in the F# community, mainly due to their ability to express non-trivial collection processing in a way that is succinct and easy to compose.

Besides being built-in and easy to use, they come with an intrinsic and potentially parallel behaviour. For example, the *map* function specifies that a particular user-defined operation is applied elementwise to the input collection, but it doesn't specify that the operation is executed sequentially, from the first to the last element. The same reasoning holds for many other collection functions, like *sort* and *filter*, for which parallel implementations have been long studied. Some restrictions must instead apply to specific collection functions, like *reduce* and *scan*, when their semantic is parallel. For example, parallel reduce and scan force the operator to be commutative and associative.

FSCL leverages these popular collection functions to build a higher level of programming abstraction over custom kernel coding. Programmers can use

[5]The developers can still write kernels that return void or explicitely pass the length of an array as an additional parameter

such functions to define parallel, skeletal computations without any syntax change with respect to regular F# programming on collections. The FSCL compiler and runtime are able to transparently discover and identify each collection function used in a computing expression and to generate and execute the corresponding parallel OpenCL implementation.

Sequential functions

As already discussed, a function that is not recognized as a custom kernel or as a collection function is executed in the traditional way (i.e. sequentially on the CPU).

While allowing sequential computations to be part of parallel programs may sound counter-intuitive, there are various reasons why we choose to allow the developers to use and compose them with custom and collection kernels.

The first reason is that some computations are inherently sequential [7,16]. Stream generation in stream-based algorithms, computing the optimal vector in a positive linear program, Huffman coding, are all examples of problems for which it is hard to define a parallel implementation or to gain a speedup from parallel execution. Despite their nature, such inherently sequential computations are heavily used together with parallel ones to solve larger problems. For example, Huffman coding is a fundamental step of the JPEG encoding/decoding pipeline, in which other steps (e.g. Discrete Cosine Transform) are suitable for massive parallelisation. By allowing sequential functions to be composed with custom and collection kernels, we give a chance to express larger computations from within the FSCL framework, especially those where parallel and sequential execution is interleaved.

A second reason to support sequential functions in kernel expressions is that once inside quotations (section 5.1.3), the framework is able to inspect their code and to acquire information otherwise difficult or impossible to retrieve. Such information, like the way parameters are accessed, can be used to deeply optimise the execution of the entire computing expression (section 8.2.2).

6.2.2 FSCL computing expressions

As discussed in the previous section, the FSCL kernel language exposes three types of computing elements. To express richer and more complex computations, elements can be composed with each other. Computing elements composition represents a structured way to define how a set of elements are executed and how data flow among each other.

Composition of computing elements in FSCL is very similar to composing functions in regular F# programming. In particular, given two F# functions

f and g, there are two major ways to compose them:

- Function composition: f and g are composed according to the classic definition of functional composition (i.e. $g(f(x))$ or, equivalently in F#, $x \mid> f \mid> g$);
- High-order function: g can be an high-order function taking f as (part of) the input (i.e. $g(f, x)$ or, equivalently in F#, $x \mid> g(f)$).

These two forms of composition of functions strictly resemble the ones exposed by the FSCL programming model to compose computing elements or, more precisely, computing (sub)expressions.

Function composition Given four computing expressions e_1, e_2, e_3 and e_4, the following expressions are valid computing expressions resulting from function composition:

- $e_1 \mid> e_2$ and $e_2(e_1)$
- $(e_1, e_2) \mid\mid> e_3$ and $e_3(e_1, e_2)$
- $(e_1, e_2, e_3) \mid\mid\mid> e_4$ and $e_4(e_1, e_2, e_3)$

Collection composition Given an FSCL computing expression e_1 and a collection function f (e.g. *Array.map*), $f(e_1)$ is an FSCL computing expression. The resulting expression represents the application of the pattern defined by the collection function, where the operator (i.e. the functional parameter of the collection function) is a computing expression[6].

Please note that, formally, ">>" is the only F# operator for function composition, while the pipeline operators like "|>" are used to apply a function or to chain function calls, which means the input must be defined. Anyway, given an input of the appropriate type, function composition and pipeline operators can be transformed one to each other. In addition, the FSCL object model supports the F# function composition operator ">>" as like as the set of built-in pipeline operators. In other terms, given two computing expressions $e1$ and $e2$ of the appropriate types, both the following expressions are recognized as valid FSCL computing expressions.

[6]For simplicity, we do not consider the difference between a collection composition operator containing calls to kernels and one with only calls to sequential functions (which is translated to an OpenCL kernel). A more precise definition of collection composition is reported in appendix A

```
// Function composition of e1 and e2
let result = data |> (e1 >> e2)
// Pipe e1 and e2
let result = data |> e1 |> e2
```

For these reasons, from now on we refer to the result of applying a pipeline operator with the term "function composition".

It is important to underline that collection functions in FSCL have a twofold purpose. From one side, they represent single units of parallel execution (i.e. computing elements). From the other, they are used to compose other computing elements or, more generally, expressions. The specific behaviour of a collection function depends on the context. If the operator of a collection function doesn't call any custom/collection kernel, the function is treated as a collection kernel and mapped to an OpenCL kernel. If instead the operator calls one or composes multiple kernels, the collection function acts as a higher-order composer, similarly to high-order skeletons [14, 22].

To clarify how collection functions are treated depending on the context and how function and collection composition can be used together, we walk through a real-world example: obtaining the average luminance from an image. In F# the algorithm can be expressed using a composition of two collection functions, as illustrated in the following listing.

```
let image = // an array of pixels (struct with R, G, B fields)

let histogram =
   image |>
   // Get Luminance from RGB
   Array.map(fun p -> (0.2126 * p.R + 0.7152 * p.G + 0.0722 * p
     .B)) |>
   // Get the average
   Array.average
```

Listing 6.4: F# program to compute the average luminance of an image

The exact same code can be used to (parallel) compute the average luminance of an image in FSCL (listing 6.5). In particular, *Array.map* and *Array.reduce* are considered collection kernels, for which the corresponding parallel implementation can be generated. The only differences from the regular F# program are the quotation marks delimiting the expression and the call to the *Run()* method to trigger the OpenCL-based parallel execution (chapter 8).

```
let image = // an array of pixels (struct with R, G, B fields)

let histogram =
 <@ image |>
    // Get Luminance from RGB
    Array.map(fun p -> (0.2126 * p.R + 0.7152 * p.G + 0.0722 *
      p.B)) |>
    // Get the average
    Array.average
 @>.Run()
```

Listing 6.5: FSCL program to compute the average luminance of an image

If instead of processing a single image we want to process a set of images, the resulting FSCL program is the one illustrated in listing 6.6. The computing expression *Array.map |> Array.average* is now "wrapped" in another collection function (another *Array.map*). The innermost map is considered a kernel that processes in parallel the set of image pixels, while the outermost map is treated as a collection composition, which defines how the operator expression is executed and applied to the input collection of images.

```
let images = // an array of images

let histogram =
 <@ images |>
    // Process each image
    Array.map(fun image ->
            image |>
            // Get Luminance from RGB
            Array.map (fun p -> (0.2126 * p.R + 0.7152 * p.G +
              0.0722 * p.B)) |>
            // Get the average
            Array.average)
 @>.Run()
```

Listing 6.6: FSCL program to compute the average luminance of an array of images

In the example the operator expression contains a function composition of collection kernels. Nonetheless, since custom and collection kernels are both F# functions, custom kernel can be used in place of collection kernels to express computations that are difficult to fit in specific collection functions. In both the cases (collection or custom kernels), a collection function whose operator contains calls to kernels is treated as a collection composition.

For the formalization of the distinction between collection kernels and collection compositions see appendix A, which contains the definitions of the elements of the kernel language.

Array.map is probably the most relevant collection function among the ones available to compose (sub)expressions. In a pure data-parallel model this collection function represents the *map* skeleton, where each worker is in its turn a data-parallel computation. In the example 6.6, each worker is a data-flow of two nodes: the first performs a data-parallel map on the pixels of the input image, while the second performs a data-parallel reduction. In a stream-based execution, *Array.map* can instead represent the *farm* skeleton.

Even if one of the most expressive, *Array.map* is not the only collection function that can be intuitively associated to a specific coordination of nested parallel computations. For example, *Array.filter* is a built-in *F#* collection function used to filter out the elements of the an input collection that do not satisfy a particular condition. When the operator is an FSCL computing expression containing kernel calls, this collection function can naturally express the execution of a parallel computation that can result in a failure for certain items of the input data. In chapter 10 we present additional examples of function and collection composition.

6.2.3 Abstraction and flexibility in FSCL composition

According to the FSCL programming model, FSCL programs are user-defined host programs that executes one or more computing expressions. Each computing expression "executes" a single computing element (collection/custom kernel, sequential function) or a set of elements/sub-expressions according to the specific composition used.

While this model is fully supported by the FSCL framework, there are two major "views" that can be built over it, each of which has a particular combination of abstraction and flexibility. These views represent the boundaries of the composition model.

Pure function/collection composition Computing elements are composed in a single expression using function and collection composition. The expression determines the shape of execution and the data flow between computing elements. The FSCL main program executes this expression;

Impervative-style composition Each computing element forms a distinct kernel expression. The FSCL program executes the set of expressions. This means that the imperative/object-oriented user-defined program determines the execution and the data flow among the elements.

Whereas collection kernels and custom kernels represent two different levels of abstraction to express a parallel computation, these two views represent *different levels of abstractions to express parallel composition.*

Function and collection composition determines the highest level of abstraction. Programmers compose computing elements in a single expression and the FSCL main program executes it (figure 6.2), coordinating kernels and passing data among them in a way that is intrinsic and trasparent to the user.

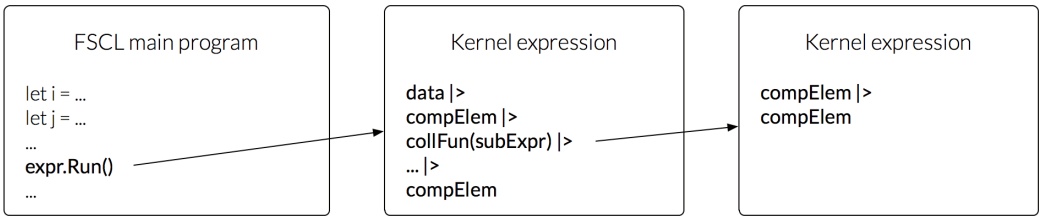

Figure 6.2: FSCL application employing pure functional composition

Imperative-style composition represents instead the highest level of flexibility, leading to a code that closely resembles OpenCL host-side programs. Programmers execute the computing elements independently from each other, taking care of synchronization and data-passing (figure 6.3).

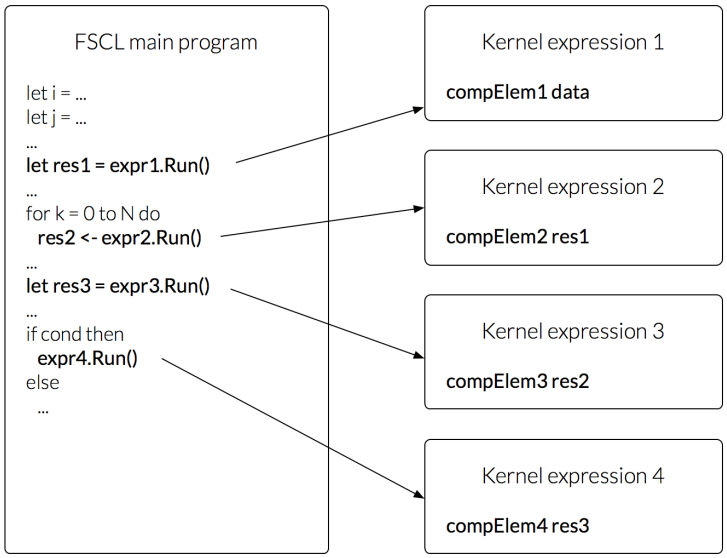

Figure 6.3: FSCL application employing pure imperative composition

The FSCL program illustrated in listing 6.6 is an example of pure functional composition. The entire algorithm is encoded in a single computing expression, which is executed by the programmer using the *Run* method.

In listing 6.7 we show an equivalent FSCL program that employ pure imperative-style composition. As illustrated, each computing expression contains a single kernel and is executed independently from the others. Data-passing among the kernels is determined by the programmer, storing the result of an expression and passing it to the successive expression. In addition, the collection function *Array.map* used to apply the map-average kernels to each input image is outside the context of the quotations. Given this, the function is executed in the traditional fashion, leveraging the CLR (i.e. it is not treated as a collection composition).

```
let images = // an array of images

let histogram =
   // Process each image
   images |>
   Array.map(fun image ->
             // Get avg luminance
             let lumaData =
              <@
                image |>
                Array.map (fun p -> (0.2126 * p.R + 0.7152 * p.G
                   + 0.0722 * p.B))
              @>.Run()
             // Now compute average
              let avg =
              <@
                lumaData |>
                Array.average
              @>
             avg)
 @>.Run()
```

Listing 6.7: FSCL program to compute the average luminance of an array of images (imperative-style composition)

6.2.4 Notes on the execution model and on data constraints

When considering an FSCL computing expression, there are two aspects that concern its execution: how single computing elements are executed and how

each composition function is handled. We briefly discuss the execution model of computing expressions in this section since it leads to a restriction on data usage. We consider the execution model in more details in chapter 8.

Parallel computing elements (custom kernels and collection kernels) are translated to OpenCL kernels and run according to the OpenCL execution model (section 2.1).

The runtime behaviour of function and collection compositions can be instead manifold. A function composition of kernels can be interpreted as a stream-based pipeline or as a simple data-flow where kernels are executed one after the other. For example, in listing 6.6, the set of *images* can be considered as an unique block of data to process or as a stream of images[7]. Similarly, each image processed in the sub-expression can be viewed as an atomic block of data or as a stream of pixels. In case of pure data-parallelism, each step of a function composition executes after the previous ones complete. In case of stream-based execution, multiple steps can run concurrently on different elements of the stream (e.g. on different images, on different pixels).

Similarly, a collection function like *Array.map*, when used to compose sub-expressions, can map to multiple execution behaviours. The simplest behaviour corresponds to sequential execution, which means that the sub-expression forming the operator is applied sequentially and linearly to the set of elements of the input data. In case of parallel execution, as already discussed, such a function can act as a stream-parallel *farm* or as an higher-order data-parallel *map*.

Another aspect of function and collection composition is whether the execution is performed on the host or on the device. Traditionally, OpenCL doesn't support nesting kernels. For this reason, collection compositions cannot map to OpenCL kernels which executes other kernels, but must instead run on the host-side. In this perspective, single computing elements express device-side parallelism while function and collection composition express host-side parallelism (i.e. parallel execution of (parallel) kernels). With the recent release of the specification 2.0, OpenCL introduces the chance for a kernel to trigger the execution of other kernels independently from the host. This means that collection compositions may effectively map to OpenCL kernels that coordinate the execution of the (sub)kernels forming the collection operator.

From the programmers' point of view, the execution model of collection and function composition should not be a concern as long as it is completely transparent. For the purpose of our research, we adopt a pure data-parallel model, with multithread execution of function and collection composition, as

[7]This is particularly true if instead of an array of images we process a sequence (*seq*) of images, since F# sequences are lazy and therefore naturally mapped to a stream of elements

discussed in chapter 8. We discuss the chance to extend this model to support streaming and OpenCL execution of collection compositions in chapter 14.

The FSCL runtime, which is the component of the framework that handles scheduling and execution of computing expression, guarantee per-expression race-freedom. Given the concurrent execution of multiple sub-expressions, the FSCL runtime restricts the computing elements composed in an expression to be side-effect free. Kernels and sequential functions can access data declared outside their scope, but accesses must be read-only. This means that the only way to pass data between computing elements is through each element's return value. Given this restriction, concurrent sub-expressions coordinated by multiple threads cannot lead to race conditions.

In case of expressions containing single, custom kernels (figure 6.3), the runtime imposes a relaxed condition. The single kernel forming the computing expression can have side effects and, in particular, can write to arrays declared and accessible outside the expression. In fact, the runtime guarantees per-expression race-freedom, while race-freedom across-expressions must be guaranteed by the programmer who writes the FSCL program. Since in the specific case considered the expression is made of a single custom kernel, the absence of per-expression race-conditions is guaranteed even if the kernel has side-effects. The choice to introduce a relaxed condition for single, custom kernels, is driven by the will to allow programmers coming from OpenCL C, where kernels can read and write buffer parameters, to develop and execute FSCL kernels in the traditional OpenCL-style.

6.2.5 Dynamic metadata

In OpenCL, programmers associate information with kernels and kernel parameters using modifiers and gcc-style attributes. Examples of such information are the memory space used to allocate the buffer for a particular parameter, the access rights to a buffer or the work size requested by a kernel.

In listing 6.8 we illustrate the usage of modifiers and attributes in an OpenCL C kernel (bold font).

```
__attribute__((reqd_work_group_size(64, 0, 0)))__
_kernel VectorAdd(_constant float* a,
                  _constant float* b,
                  _global float c) {
   int gid = get_global_id(0);
   c[gid] = a[gid] + b[gid];
}
```

Listing 6.8: OpenCL C99 kernel with attributes

Information associated with kernels and kernel parameters through attributes and modifiers are related to the implementation of the code more than to its semantic. In other terms, this information represents a way to drive the compilation and the OpenCL runtime behaviour. For example, switching from the parameter modifier *_constant* to *_global* doesn't affect the body of the kernel[8]. Instead, it is information that the OpenCL driver uses to allocate and to give access to the buffer matching the parameter. Similarly, the *reqd_work_group_size* attribute associated to a kernel doesn't affect the computation behaviour, but it's instead a way to constrain the work-size when the kernel is executed.

Since this information does not change the semantic of a kernel but instead affect OpenCL compilation and execution, FSCL exposes them via the custom attributes mechanism. Listing 6.9 shows the same attributes used in listing 6.8 but applied to an FSCL kernel.

```
[<ReflectedDefinition>]
[<WorkGroupSize(64,0,0)>]
let VectorAdd([<AddressSpace(AddressSpace.Constant)>]
              float32[] a,
              [<AddressSpace(AddressSpace.Constant)>]
              float32[] b,
              [<AddressSpace(AddressSpace.Global)>]
              float32[] c,
              wi: WorkItemInfo) =
   let gid = wi.GlobalID(0)
   c.[gid] <- a.[gid] + b.[gid]
```

Listing 6.9: FSCL kernel with static metadata

Metadata implemented as custom attributes are forced to be statically defined and associated with a kernel, a parameter or to the return type. This implies the need to change or duplicate the definition of an FSCL kernel whenever the value of a metadata must change. This is particularly problematic in testing environments, where different combinations of address-space and memory-flags for specific buffers might be subjects to several changes. In multi-device systems, the same kernel may obtain the highest, overall performance by using different buffer memory-flags for different devices[9]. Static definition of metadata also prevents from using them to associate information with collection kernels. In fact, unlike custom kernels, collection kernels lever-

[8]The only exception is the *_local* modifier, which implies that the buffer is allocated per-group and shared among the work-items in the group

[9]In many cases, CPUs and discrete GPUs obtains respective highest performance under a different set of flags for the buffers used

age native F# collection functions, whose definition can't be manipulated.

For these reasons, our reasearch leads to the definition of a more flexible approach to associate meta-information to kernels, called *dynamic metadata*.

Definition 1. [Dynamic metadata] Given m an abstract meta-information, $p_1, p_2, \ .. \ p_n$ its properties of type $t_1, t_2, \ , \ .. \ t_n$, a dynamic metadata is defined as a tuple formed by:

- A CLR custom attribute (object inheriting from *System.Attribute*) of type A with properties of type $t_1, t_2, \ .. \ t_n$;
- A function, called metadata-function, of type $f : t_1 * t_2 * \ .. \ t_n * t_{wrap} \rightarrow t_{wrap}$, where:
  - t_{wrap} is an arbitrary type
  - Given an instance el of type t and n arguments $a_1, a_2, \ .. \ a_n$, $f(a_1, a_2, \ .. \ a_n, el) = el$. In other terms, the partial application of f to $a_1, a_2, \ .. \ a_n$ results in the identity function.

Custom attributes, which represent the "static representation" of dynamic metadata, are used to associate information to a kernel definition or to a kernel parameter/return type at coding-time. When the programmer wants to associate the same information dynamically (i.e. at kernel compilation/execution time) the corresponding metadata-function can be used to "wrap" a kernel call or a kernel argument.

In the rest of this Thesis, when referring to FSCL dynamic metadata, we use the term "*metadata type*" to indicate the type of the CLR custom attribute or the metadata-function. When no distinction has to be made beween the static and the dynamic representations of dynamic metadata, we use the term "*metadata value*" to refer to either an instance of the CLR custom attribute or to a call to the corresponding metadata-function. Finally, with the generic term "metadata" we refer to an arbitrary meta-information associated to a kernel.

In listing 6.10 we illustrate how the same metadata used in listing 6.9 can be associated to kernels and parameters dynamically. The *WorkGroupSize* meta-information has three integer properties, which are the work size required for each dimension of the work-items space. In listing 6.9 a custom attribute is used to associate that information to the kernel definition. Instead, in listing 6.10 we wrap a call to the kernel in the *WorkGroupSize* metadata-function, which takes an argument for each property of the custom attribute. In case of metadata associated to a parameter, the metadata-function wraps the corresponding argument.

```
let arr1, arr2, arr3 = ...
WorkGroupSize(64, 0, 0,
              VectorAdd((AddressSpace(AddressSpace.Constant),
                         arr1),
                        (AddressSpace(AddressSpace.Constant),
                         arr2),
                        (AddressSpace(AddressSpace.Global),
                         arr3)))
```

Listing 6.10: FSCL kernel with dynamic metadata

Thanks to dynamic metadata, programmers can define kernels once and compile/execute them multiple times with different sets of meta-information associated.

Metadata are not only used to represent predefined OpenCL attributes and modifiers. In fact, since the FSCL framework places various layers on top of OpenCL, additional metadata that do not match any of the OpenCL built-in attribute/modifier are provided. These additional metadata are used by the FSCL framework layers, such as the FSCL compiler and the runtime.

One of the most relevant additional metadata is *DeviceType*, which is used by the FSCL compiler to generate optimized OpenCL source code for a specific type of device (GPUs, CPUs, Accelerators, etc.). Since OpenCL doesn't provide any automatic code optimisation, this is an information without any match in the OpenCL world. Instead, as described in chapter 7, the FSCL compiler is able to exploit this meta-information to output device-specific OpenCL code for collection kernels.

Even though they do not match any OpenCL built-in attribute or modifier, these additional information are exposed using the same dynamic metadata infrastructure. This allows to pass meta-information to the set of framework layers and to the OpenCL compiler/driver in a simple and homogeneous way.

Restrictions on dynamic metadata

Programmers are allowed to extended the set of dynamic metadata that can be associated to kernels and to kernel parameters and return types. Nonetheless, there are some restrictions that applies to both built-in and user-defined metadata.

The first restriction is the uniqueness of target type: if a metadata can be associated to a kernel, it can't be associated to a parameter or to a return type. Similarly, metadata for kernel parameters cannot be used for kernels or return types and metadata for return types cannot be used for kernels or parameters.

We formalize this restriction, which is particularly important for both the FSCL compiler and the runtime (section 7.4.2).

Definition 2. [Uniqueness of dynamic metadata target type] Given a dynamic metadata m, the set of target types $T = \{Kernel,\ Parameter,\ ReturnType\}$ and two targets t_1, t_2, we indicate with $type(t_1) \in T$ ($type(t_2) \in T$) the type of t_1 (t_2). Uniqueness of dynamic metadata target type states that m can be associated to both t_1 and t_2 if and only if $type(t_1) = type(t_2)$

The second restriction demands at most one instance of a particular dynamic metadata for a specific target. In other terms, it is not possible to use two dynamic metadata of the same type for the same target. For example, it is not possible to specify the metadata *DeviceType* twice for a specific kernel.

Definition 3. [Dynamic metadata disjointness by type and target] Given two dynamic metadata values m_1 and m_2, with M_1 the type of m_1 and M_2 the type of m_2, and a target t instance of $T \in \{Kernel,\ Parameter,\ ReturnType\}$, if both m_1 and m_2 are associated to t then $M_1 \neq M_2$

6.3 Conclusions

In this chapter we presented the FSCL kernel language, which is the programming and object-model exposed to express parallel computations and to compose them.

The language is based on two levels of abstraction. At the higher level, programmers employ collection functions and function composition. Since this involves using and composing built-in F# functions without any change from regular, sequential F# coding, programming at this level has the benefit of a complete transparency from the underlying OpenCL layer and a zero-length learning curve.

The flexibility issues of this programming level may come from two different sides. From the first, a computing element can't be expressed using a collection function. In such a case, the developer can code a custom kernel and plug it into the composition without the need to change the other elements, which can continue to be expressed at the highest level of abstraction.

From the second, function and collection composition are not flexible enough to express the coordination of computations that form the parallel program. In this case, the computing expression can be divided into multiple parts (expressions), executed by the FSCL program independently from each other, relying on the program itself to coordinate the execution of the various parts.

The approach to FSCL programming can be summarised in two statements:

- Use F# collection functions to implement parallel computations as long as they are flexible enough to express the computation to execute. If a computation cannot be expressed using a collection function, use a custom kernel;
- Compose computations using collection composition and function composition as long as these constructs are flexible enough to express the logic of execution and the dependencies between computations. If different parts of the parallel program are not feasible to be composed using collection and function composition, place them in separated expressions and coordinate them explicitely in the host program.

The main advantage of the model proposed is the ability to compose computing elements at different levels of abstraction (collection kernels and custom kernels) and to mix function/collection and imperative-style composition. Thanks to this, the developer can code at the most appropriate level with fine-grained control, without the need to port the entire application to the low-level when only part of it doesn't fit the higher one.

Chapter 7

FSCL Compiler

In chapter 6 we presented the FSCL kernel language, which is the programming model exposed by FSCL to express parallelism in a flexible and abstract way. FSCL programmers leverage native collection functions and custom kernels to develop parallel computations, which can be composed with each other through function and collection composition. A single kernel or a composition of multiple kernels is referred as *computing* (or kernel) *expression*.

In this chapter we introduce the result of our research towards a strategy to map the high-level object-model exposed by the framework to the low-level OpenCL model. This mapping is performed by the FSCL compiler, which is the component of the framework resposible to process computing expressions in order to generate:

1. A graph representing the dependencies/coordination of computing elements in the expression, called *Kernel Flow Graph* (*KFG*);

2. The OpenCL source code for each custom/collection kernel in the expression.

7.1 FSCL Compiler structure

The FSCL compiler is a source-to-source compiler whose main purpose is to transform FSCL kernels into OpenCL kernel sources. This purpose is accomplished progressively through a *compilation pipeline*. One after the other, the steps of this pipeline generate the target (OpenCL) representation of FSCL kernels.

Since the compilation process is thought to be fully customizable and extensible (section 7.5), one of the core components of the compiler is the *compiler configuration infrastructure* (figure 7.1). This component is responsible

for taking a set of steps that should process, organize and validate them and finally instantiate the compilation pipeline.

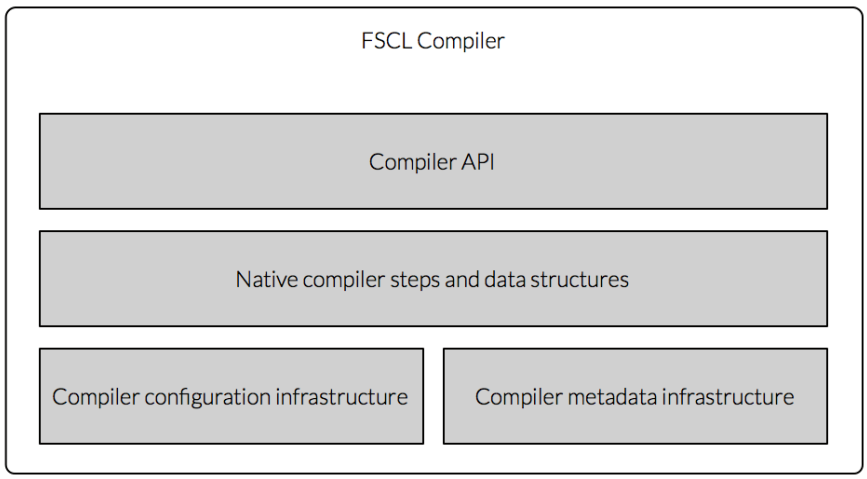

Figure 7.1: Structure of the FSCL compiler

On top of the configuration infrastructure the compiler exposes a set of *native compilation steps and data-structures*. These components represent the default structure and behaviour of the compilation process or, in other terms, the default instance of the compilation pipeline. Without any specific user-defined settings, this is the pipeline that executes whenever an FSCL computing expression is compiled.

As discussed in section 6.2.5, the FSCL kernel language allows programmers to associate meta-information to kernels and to its paramters and return type. Some of this information can be inspected and used by the compiler pipeline to drive its own behaviour. Therefore a third, important component of the FSCL compiler is the *compiler metadata infrastructure*, which is responsible to retrieve and analyze kernel metadata.

Finally, the compiler project exposes an API to configure, extend and use the compiler. The FSCL runtime (chapter 8) interacts with the compiler through this API to compile the computing expression before being futher processed and executed.

In the rest of this chapter we firstly describe the compilation process and components from an abstract point of view. Then we present and discuss the details of the *native compilation pipeline*. Finally, we briefly talk about compilation extension and customisation.

7.2 Abstract compilation process and components

From an abstract point of view, the compiler pipeline is a function that takes an object and returns an object. There are three components involved in the implementation of this function: *steps*, *processors* and *type-handlers*.

7.2.1 Steps and processors

The compiler pipeline is made by a set of steps (or stages), each of which is made of a set of processors. As the name suggests, a step is a sequential, independent processing action of the compiler pipeline. For example parsing, preprocessing, AST transformation and code generation are steps. To perform its tasks a step requires a set of processors, each of which accomplished a subset of the tasks assigned to the step. For example the code generation step emits the target code of a kernel. Each processor of this step generates the code for a particular AST node. The transformation step manipulates the AST to produce an abstract representation that is closer to the OpenCL language. Each processor of this step transforms a particular language construct.

The steps of the pipeline are executed sequentially, from the first to the last one. In other terms, the steps *orchestration* is sequential. While steps are orchestrated in a fixed way, the orchestration of the processors inside a step depends on the particular step considered. Some steps may test their processors against the input and select the first processor that is able to process it. Other steps may execute the entire set of processors in the same way the pipeline executes steps.

The orchestration establishes the model used to run the set of processors of a step or the set of steps of a pipeline (e.g. all the processors, the first suitable, all until one fails), but it doesn't specify the *order* in which the steps/processors are considered. The specific order of execution of steps/processors is determined from the *dependencies* that each step/processor declares. To express inter-processor and inter-step dependencies FSCL employs custom attributes. Each step and processor of the compiler pipeline is characterized by the following set of information needed to determine the execution order.

ID A globally unique step/processor identifier.

Step ID For a processor, the globally unique identifier of the step it belongs to.

After The step/processor dependencies, represented by a list of IDs of steps/processors that must be executed before the current one.

Before A list of IDs of steps/processors that must be executed after the current one.

In most of the cases the dependency list provided by *After* is enough to order processors/steps and the *Before* property is left unspecified. Nevertheless, this information is needed to guarantee the highest flexibility in extending and configuring the compiler (section 7.5).

Given the set of inter-dependencies metadata, the configuration infrastructure is able to build a pipeline where the execution order of steps and processors respect all the inter-step and inter-processor dependencies. In case of dependencies too strict to be respected (e.g. cyclic dependencies), the infrastructure reports an error at pipeline-build time.

Figure 7.2 shows an example of ordering a set of steps. From the partial order determined from the Before/After properties of each step, an absolute order is built. Note that $C \rightarrow A \rightarrow B \rightarrow D$ is another valid absolute order.

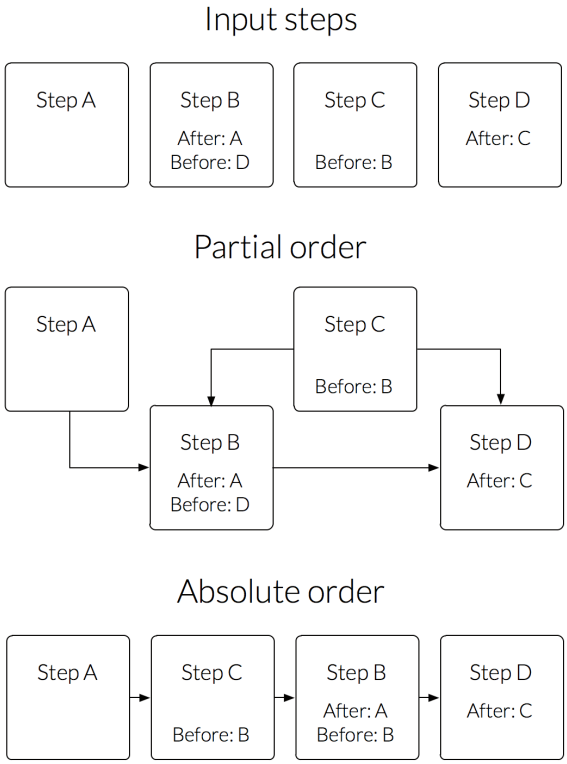

Figure 7.2: Example of partial and absolute ordering of steps

7.2.2 Type-handlers

In a compilation process, the set of types employed in the high-level model generally differs from the set of low-level data-types. In the specific case of F#-to-OpenCL compilation, the set of F# types is wider than the set of valid OpenCL C99 types. In addition, OpenCL only allows to extend the set of types involved in a kernel with custom structs, while F# gives more freedom to declare custom types.

We can identify two sets of types involved in the compilation process: *source types* and *target types*. In FSCL-to-OpenCL compilation, the source types are all the types that the programmers can use in coding FSCL computations. During the compilation, many of these types are mapped to different types to obtain a type-system that is closer to the OpenCL one. For example, since OpenCL doesnt support multi-dimensional arrays, each multi-dimensional array is replaced with the corresponding one-dimensional version, manipulating indexing expressions to guarantee the correctness of accesses. F# records and ref cells are another example of source types that are transformed during compilation. Records are replaced with structs and ref cells with singleton arrays. At the end of this transformation, the set of types employed in the representation of the code is restricted to a subset of the original one, which is what we call *target types*. Type-handlers come into play to produce the target OpenCL code for this restricted set of types. In other terms, a type-handler is an entity responsible for declaring a set of valid target types and producing the string representation for each of them. For this reason, type-handlers are mainly used in the latest compilation steps (codegen), where the target code is generated.

In table 7.1 we show the matching between source types, target types and OpenCL code representation for some valid source types. The matching is established by the set of type-handlers involved in the compilation process. In the table, we specifically show the source-target matching established by the set of native type-handlers (i.e. type-handlers of the native compilation pipeline). Compiler extensions allow developers to define additional type-handlers or to replace the existing ones.

As the table reports, some types are identical across their source, target and OpenCL code representation (e.g. *int*). Other data-types, like *float32* have only a different OpenCL string representation. Multi-dimensional array types (e.g. *int[,]*) are replaced with the corresponding one-dimensional array types. The target type for a reference cell of type *T ref* is the array type *T[]*. Finally, complex source types like tuples and F# structures are replaced with target struct types.

Source type	*Target type*	*OpenCL code*
int	int	"int"
float32	float32	"float"
float	float	"double"
T[]	T[]	"T*"
T[,]	T[]	"T*"
T ref	T[]	"T*"

T * U	`type tuple_TU =`
	`  struct`
	`  val mutable fst: T`
	`  val mutable snd: U`
	`  end`
		``` ``struct tuple_TU {``
		`  T fst;`
		`  U snd;`
		` }'' ```

Table 7.1: A set of FSCL source types with matching OpenCL target types and code representation

7.3 Coordination of steps, processors and type-handlers

The core of the FSCL compiler is an infrastructure that exposes basic, abstract data-types for steps, processors and type-handlers and a mechanism to load, configure and organize these components into a pipeline. The compiler core expects a set of concrete components to be defined by inheriting from the appropriate pre-existing abstract types (e.g. *IStep*, *IProcessor*, *ITypeHandler*) and to be submitted to the pipeline builder. The builder validates the inter-steps and inter-processors dependencies, instantiates each steps with its set of processors and provides each step/processor the set of type-handlers for type-checking and target-type code generation. The result of the pipeline building process is an object that exposes a *Compile* method. When invoked, this method triggers the sequential execution of the steps in the appropriate order and returns the value produced by the last one.

In the next section, we discuss the native compilation pipeline, which is made of a set of concrete components that implement FSCL-to-OpenCL mapping. As already said, this is a specific instance of the unlimited set of pipelines that can be built over the FSCL compiler core. In section 7.4.3 we briefly discuss the possibilities that the FSCL compiler offers to extend/replace the set of native steps thanks to the abstraction and the generality of its core.

7.4 Native compilation process and components

The FSCL compiler comes with a native set of steps, processors, type-handlers and metadata which are used to build the default (native) compilation pipeline. Whenever programmers do not specify a different configuration, this is the compilation pipeline executed to process FSCL computing expressions.

The input of the native compilation pipeline is a *quoted computing expression*, that is a composition of collection/custom kernels and sequential functions wrapped into an F# quotation. As discussed in section 5.1.3, F# quotations allow us to unobtrusively obtain the AST of the quoted expression, which can be consequently analysed and transformed.

The output of the native compilation is a data-structure, called *Computing Expression Module* (*CEM*), which contains all the information produced during the compilation process, including the graph that represents the composition of the elements in the expression and the OpenCL source code for each kernel. We describe this data-structure in section 7.4.2.

In this section we firstly describe the process of parsing a computing expression to build the *Kernel Flow Graph*, which is the data-structure used to represent the composition of computing elements. Then, we introduce the concepts of *kernel equivalence* and *metadata equivalence*, which are used to identify a particular kernel and to characterize the result of its compilation. This is particularly important to enhance the efficiency of the whole FSCL framework and, especially, to avoid the compilation of already-compiled kernels. Finally, we briefly describe the set of native steps and the native data-structures used to represent the output of the compilation of a kernel and the output of the entire compiler pipeline.

For simplicity, in this section we use the terms "compilation" and "compiler" to refer to the native compilation process and to the compiler built with the native set of components. As discussed in the next section, a part of the pipeline (more precisely, the first step) is applied *computing-expression-wise*, which means that the steps are executed only once for the whole input expression, while the rest of the pipeline is applied *kernel-wise*, that is independently to each kernel in the expression. With the term "kernel-compilation (pipeline)" we refer to the set of consecutive steps in the pipeline that globally perform FSCL-to-OpenCL compilation of single FSCL kernels.

7.4.1 Expressions parsing, Kernel Flow Graph and Computing Expression Module

When an FSCL computing expression is submitted for compilation, the first step performed by the native compiler pipeline is to build the *Kernel Flow*

Graph (*KFG*) in order to represent the composition of computing elements inside the expression.

In line with the programming model presented in chapter 6, the KFG can contain four types of nodes:

- Kernel nodes, which represent calls to collection and custom kernels;
- Sequential nodes, which represent calls to functions executed sequentially on the host;
- Collection composition nodes, which encapsulate the composition of sub-expressions using high-order collection functions;
- Data nodes, which represent the input (arguments) of the expression.

In the KFG there is no node representing function composition, since it is implicitly encoded in the dependencies between the nodes of the graph.

In figure 7.3 we show the KFG created for a computing expression that encodes the *K-Means* clustering algorithm [40]. The input of the computation is the set of points to cluster and the initial set of cluster centers. It is important to note that while the set of points is the input of the first kernel in the expression (i.e. *Array.groupBy*), the cluster centers are passed to the (utility) function *nearestCenter*. The compiler is able to recognize references to global variables/properties[1] from within computing expressions and to generate the appropriate KFG data nodes. The OpenCL source of elements referencing global data is also properly generated to guarantee data is passed correctly from the top-level node to the specific, nested elements where the reference is found. In other terms, the programming model and the compiler are able to handle *closures* and *partial-application* (section 5.1.1).

Once processed by *Array.groupBy*, data is passed to the *Array.map* node. Since the operator of this collection functions is a composition of kernels, the function is treated as a collection composition. Therefore, a *macro node* is created to represent it, while for the sub-expression forming its operator a subgraph (sub-KFG) of two nodes is created. The input/output of the macro node is properly connected to the input/output of the sub-graph. The first element in the sub-expression is a collection kernel (*Array.reduce*), while the second and last element is a sequential function that finalizes the calculation of the cluster centroid. Once built, the KFG is passed to the successive steps of the compilation pipeline. Currently, the compiler doesn't perform any global

[1] With "global variables/properties" we indicate variables/properties declared outside the context of the quotation and, more generally, data that can be referenced from multiple kernels/functions

analysis or transformation on the graph, which is in fact an immutable data-structure. Nevertheless, the KFG and the information produced for each kernel during compilation may be employed to fuse together or to optimize some nodes in the future. We discuss potential KFG transformations techniques in section 14.1.

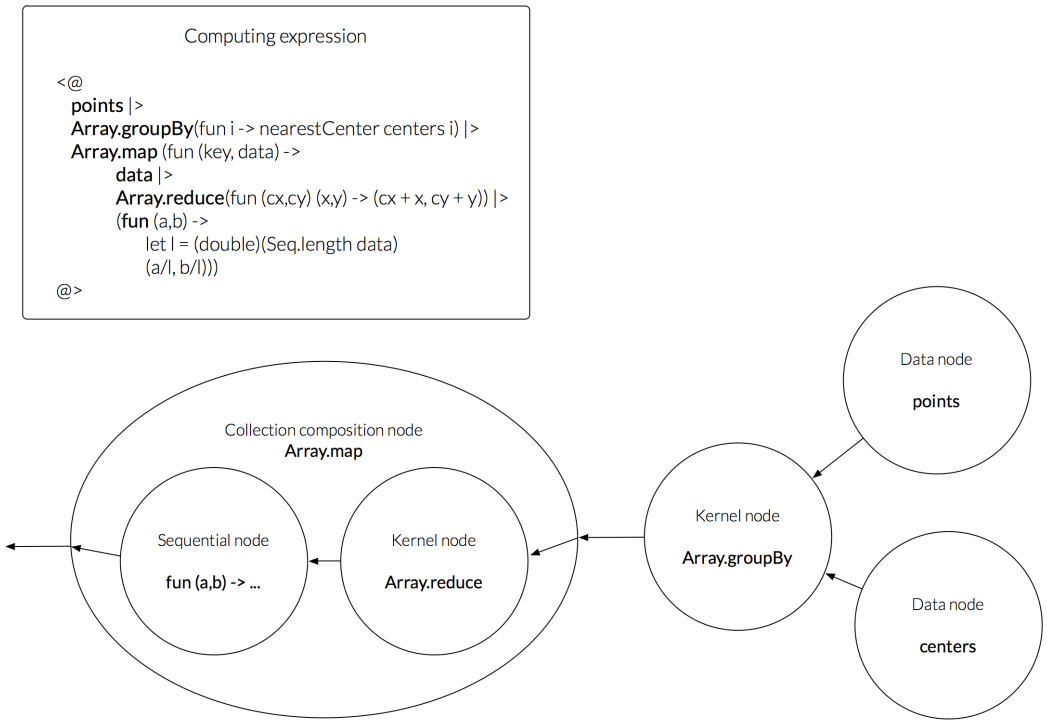

Figure 7.3: Example of the KFG of a computing expression

As already briefly discussed, the first step of the native compilation pipeline (i.e. the computing expression parser) instantiates a data-structure called *Computing Expression Module* (*CEM*), which contains the KFG along with other relevant information. The rest of the pipeline is applied *kernel-wise*, which means that each step is applied repeatedly and independently to each kernel node in the KFG (figure 7.4). Sequential functions are not compiled, since they do not need any processing to be properly executed on the CLR. At the end of the pipeline the CEM contains, in addition to the KFG, the output of the compilation of each kernel node (including the OpenCL source code).

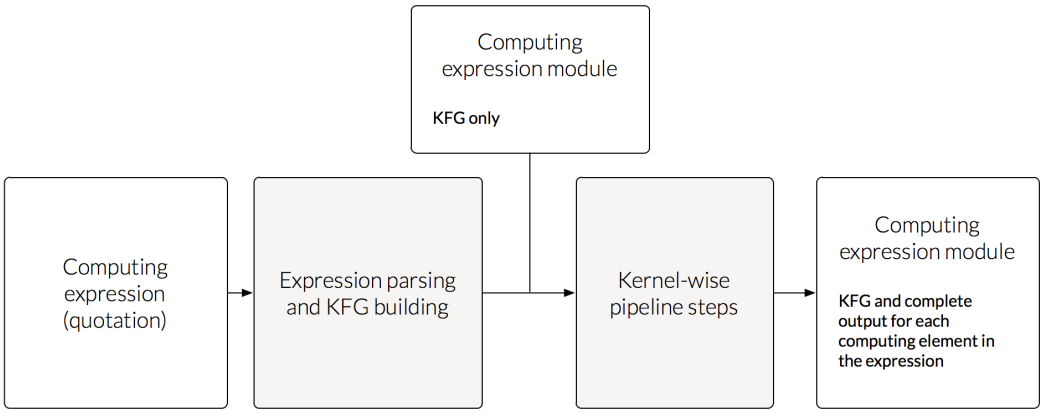

Figure 7.4: Computing Expression Module instantiation and filling

7.4.2 Kernel compilation, kernel equivalence and Kernel Module

Once the KFG has been built, the compiler processes the set of kernel nodes independently from each other. The compiler is a *cross-expression stateful compiler*, which means it maintains a state across successive compilations of computing expressions. In particular, the compiler keeps a cache with the kernels compiled so far. When a kernel node is processed, the compiler firstly checks if the cache contains an *equivalent* kernel. In such a case, the kernel is not processed by the kernel-compilation pipeline but cloned from the cache, saving compilation overhead (figure 7.5).

For this process to work, a criteria of *equivalence* of kernels must be defined. In the rest of this section we formalize this definition and we present the data-structure used to store the information resulting from the compilation of single FSCL kernels.

Equivalence of explicit kernels, collection kernels and lambdas

In the CLR, F# module functions and static/instance methods can be identified with an object of type *MethodInfo*. Since instances of such type can be tested against equality and given that FSCL custom kernels are (user-defined) F# functions, the *MethodInfo* object represents the ideal kernel identifier.

In case of collection kernels, such as *Array.map*, the output of the compilation process is not only affected by the particular collection function, but also by the operator applied to the elements of the collection. For this reason, collection functions are identified by a tuple, where the first item is the *Method-*

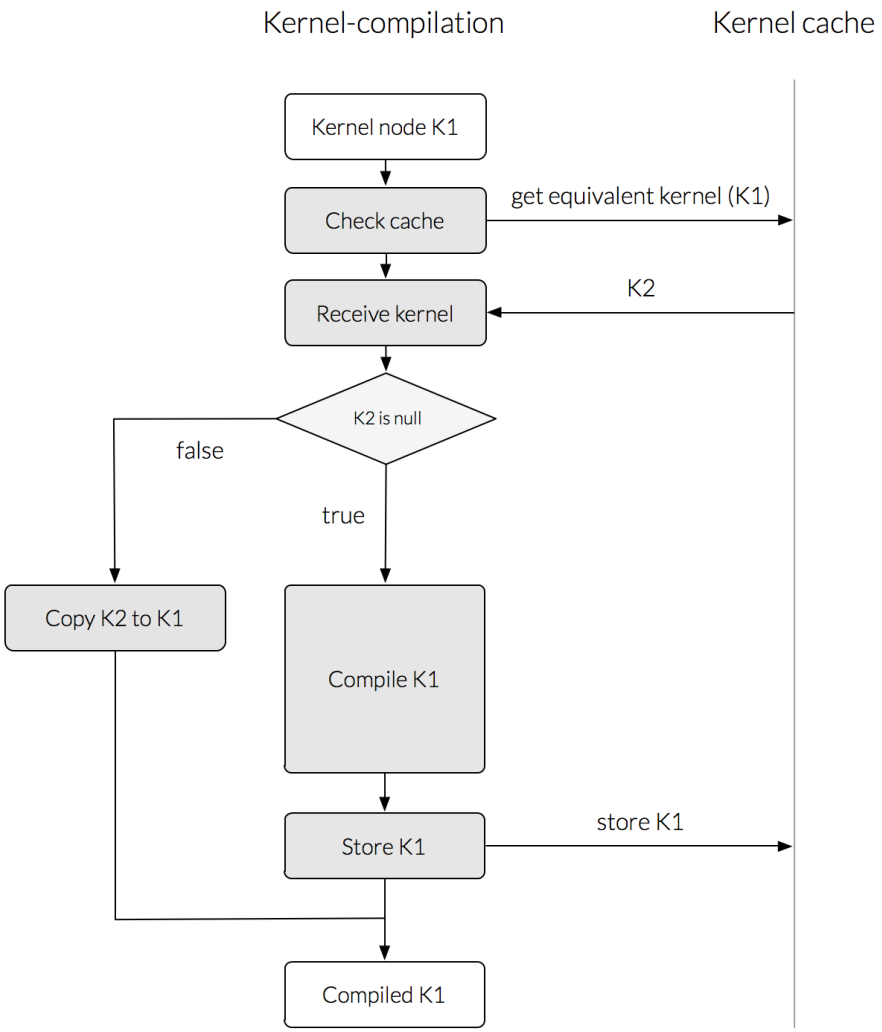

Figure 7.5: Kernel compilation and caching

Info associate to the collection function, while the second is the *MethodInfo* of the operator. While *Array.map* accepts as argument the operator to apply to each element of the input, some other collection functions, like *Array.reverse*, do not expect any. In such a case, the kernel can be uniquely identified by the *MethodInfo* of the collection function.

In FSCL, programmers can also use lambda functions to express kernels and collection functions operators. Even though each lambda object expose an *Invoke* method whose *MethodInfo* may be used to uniquely identify it, the type and the container assembly of lambda functions are dynamically generated, which makes testing for equality fail.

For this reason, in case of lambdas we apply a structural equivalence approach. Given two lambda functions, they are considered *structurally equivalent under alpha conversion* (or *alpha renaming*) if the respective ASTs have the same structure and each matching pair of nodes are equal up to a coherent variable renaming.

To formalize the concept of structural equivalence of ASTs, let's define *node (or expression) equality* the equality definition provided by F# for instances of the F# *Expr* type. Given two nodes, we say that they are equal if and only if they hold the F# expression equality. We use the "=" operator to express the equality of nodes.

Given a node n and two variables var_1 and var_2 of the same type, we denote with $n_{[var_2/var_1]}$ the node obtained by replacing each reference to var_1 with a reference to var_2.

We can now introduce the concept of *structural equivalence under alpha conversion* for AST nodes. We use the "$\equiv_{struct}$" symbol to express this equivalence.

Definition 4. [Node structural equivalence under alpha conversion] Given two AST nodes n_1 and n_2, they are structurally equivalent under alpha conversion if and only if one of the following conditions is satisfied.

- $n_1 = Let(var_1, val_1, body_1)$, $n_2 = Let(var_2, val_2, body_2)$, var_1 and var_2 are of the same type, $val_1 \equiv_{struct} val_2$ and $body_{2[var_1/var_2]} \equiv_{struct} body_1$
- $n_1 = For(var_1, st_1, en_1, incr_1, body_1)$, $n_2 = For(var_2, st_2, en_2, incr_2, body_2)$, var_1 and var_2 are of the same type, $st_1 \equiv_{struct} st_2$, $en_1 \equiv_{struct} en_2$, $incr_1 \equiv_{struct} incr_2$ and $body_{2[var_1/var_2]} \equiv_{struct} body_1$
- $n_1 = n_2$

The definition considers *let-bindings* and *for-loops* independently from the rest of the AST node types, since in the FSCL kernels these constructs are the only two that can introduce a new variable in the context.

Once defined structural equivalence for arbitrary nodes, we can define the equivalence of ASTs.

Definition 5. [AST structural equivalence under alpha conversion] Given two ASTs with n_1 and n_2 the respective root nodes, they are structurally equivalent under alpha conversion if only if $n_1 \equiv_{struct} n_2$ (definition 4)

Structural equivalence of ASTs is more powerful then code string equality, since it holds also when renaming the set of variables declared. For example, the following two lambdas holds structural equivalence, even though their string representations are different from each other.

```
fun a b c ->
  let mutable d = a * 2
  for i = 0 to b do
    d <- d + i
  c * d + 2

fun e f g ->
  let mutable h = e * 2
  for j = 0 to f do
    h <- h + j
  g * h + 2
```

Listing 7.1: Equivalent lambdas under alpha conversion

In addition, testing structural equivalence is more efficient than comparing the string representation of two ASTs, especially for verbose kernels.

To summarize what has been discussed in this section, we generalize the concept of equivalence of kernels. We firstly introduce the concept of equivalence of functions, for which we use the symbol $\equiv_{fun}$.

Definition 6. [Function equivalence] Given two module functions, instance/static methods or lambdas f_1 and f_2, they are considered equivalent if and only if one of the following conditions is satisfied.

- f_1 and f_2 are modules functions or instance/static methods, M_1/M_2 is the *MethodInfo* associated to f_1/f_2 and $M_1 = M_2$
- f_1 and f_2 are lambdas and $f_1 \equiv_{struct} f_2$

We can now formalize the concept of kernel equivalence. We use the symbol $\equiv_{ker}$ to denote equivalence of kernels.

Definition 7. [Kernel equivalence] Given two kernels k_1 and k_2, from the compilation point of view they are considered equivalent if and only if one of the following conditions is satisfied.

- k_1 and k_2 are custom kernels or lambdas and $k_1 \equiv_{fun} k_2$
- k_1 and k_2 are collection kernels, $F_1 = \{f1_1, \; ..f1_n\}$ and $F_2 = \{f2_1, \; ..f2_n\}$ are the respective sets of operators (arguments) to apply, $k_1 \equiv_{fun} k_2$ and $\forall_{i=1 \; .. \; n} f1_i \equiv_{fun} f2_i$

Equivalence of compiler metadata

In the previous subsection we discussed the concept of equivalence of kernels. Two equivalent kernels represent identical input computations for the kernel-compilation pipeline.

As introduced in section 6.2.5, programmers can use dynamic metadata to drive the compilation output and, more specifically, the steps and processsors behaviour. For example, the output of the compilation of a collection function is affected by the *DeviceType* metadata. If the device type specified is GPU, GPU-optimised code is produced. Otherwise, CPU-optimised code is generated. Another item of metadata that affects the compilation output is *AddressSpace*, which is used to declare the memory space used to allocate the buffer for a particular parameter.

Therefore, whereas for kernel equivalence it is sufficient to describe when two kernels are considered the same input, the concept of metadata equivalence must be introduced to define when the compilation process produces the exact same output.

Even if CLR structural equality may be employed to compare metadata, the concept of equivalence of two dynamic metadata is not based on the metadata themeselves, but on the steps or processors whose behaviour is affected by the metadata. In fact, given a metadata type whose domain is made of a set of values, a step or processor may behave differently for some combinations of values while behaving in the same way for other combinations. For example, the collection kernels parser, which is part of the native compiler components, produces diffent results depending on whether the value of the *DeviceType* metadata is "*Cpu*" or not. The code generated when *DeviceType* is "*Gpu*", "*Accelerator*" and "*Other*" is the same. This means that, from the parser point of view, two *DeviceType* metadata values are equivalent when they are both "*Cpu*" or both different from "*Cpu*". Another step may use the same metadata but produce different results under different conditions.

Given that the equivalence of metadata, in terms of the output produced by the kernel-compilation, is determined by the steps and processors in the

compilation pipeline, it is up to each step and processor in the pipeline to declare:

- The types of metadata that can affect the step/processor behaviour
- For each metadata type, a *comparer* used to establish when two metadata values are equivalent, in terms of whether they lead to the same step/processor behaviour

We now introduce the formal concept of equivalence of sets of metadata values, which is used to establish the equivalence of the outputs of the kernel-compilation pipeline for a given kernel under different sets of associated metadata values. By definition 1, a dynamic metadata is made of a (static) CRL custom attribute and of a metadata-function to associate the metadata to a specific target at runtime. In this section we consider only the static representation of metadata. Metadata equivalence can be easily extended to metadata-functions by transforming the set of metadata-functions associated with a kernel (call) to the set of corresponding instances of CLR custom attributes. The compiler actually applies this transformation in order to reduce the twofold representation of dynamic metadata values to uniform representation based on CLR custom attributes.

Let's define a *metadata comparer* for a metadata type M a function that given two values of M returns true if they are equivalent and false otherwise.

$$metadata\ comparer:\ M * M \rightarrow bool$$

The metadata equivalence for a set of given comparers can be formalized ad follows.

Definition 8. [Metadata equivalence] Let M be a metadata type (type inheriting from the built-in *Attribute* type) and $MC = \{mc_1, mc_2,\ ..mc_n\}$ a given set of metadata comparers for M. For each pair of metadata values m_1, m_2 where $typeof(m_1) = typeof(m_2) = M$, m_1 is equivalent to m_2 under the set MC ($m_1 \equiv_{MC} m_2$) if and only if:

$$\forall mc \in MC : mc(m_1, m_2)$$

Given a compilation pipeline and the set M of metadata types declared to be used by the pipeline steps and processors, the set of metadata values specified by the programmer may not match M. In fact, programmers may employ metadata that are not used by the compiler (e.g. they may be used by the runtime) or may not specify a value for one or more compiler metadata.

For this reason, we define the concept of *complete set of metadata values*, which represents a transformation of the input set of metadata values to perfectly match the set of metadata types declared to be used by the compiler components.

Definition 9. [Complete set of metadata values] Given a set of metadata types $M = \{m_1, m_2, \; .. \; m_n\}$ and a set of metadata values $MV = \{mv_1, mv_2, \; .. \; mv_m\}$, we indicate with $default(m)$ the default value of a metadata type m. We define complete set of metadata values for M under the set of comparers MV the set $MV_{[M]}$ where:

- $\forall m \in M : \exists mv \in MV : typeof(mv) = m \rightarrow mv \in MV_{[M]}$
- $\forall m \in M : \nexists mv \in MV : typeof(mv) = m \rightarrow default(m) \in MV_{[M]}$

The construction of the complete set of metadata values filters the input set of metadata values from each elements whose type is not declared to be used by any component of the pipeline. In addition, it adds to the set the default value of each metadata type used by the pipeline that is left unspecified by the programmer.

Given a pipeline with a set M of metadata types that the steps/processors declare, we can partition M into three subsets KM, RM, PM, which are respectively the set of metadata that can be associated to a kernel, to a kernel return type and to a kernel parameter. We call this separation *per-target-type metadata partition*.

Given the restriction imposed by definition 2, we know that if a metadata type M belongs to one of this three partitions it can't belong the other two.

Given a kernel K, $Pars$ its parameters and MV the set of metadata values, we can partition MV into three subsets:

- KMV: the set of metadata values associated to the kernel;
- RMV: the set of metadata values associated to the kernel return type;
- PMV: the set of metadata values associated to the kernel parameters. $PMV = \{pmv_1, pmv_2, \; .. \; pmv_n\}$, where $n = |Pars|$ and pmv_i $(i = 1 \; .. \; n)$ is the set of metadata values for the i_{th} parameter.

We call this separation *per-target metadata-value partition*.

With the definitions provided so far, we can now describe when two sets of input metadata values are equivalent to each other. As in definition 9, we denote with $typeof(m)$ the type of a metadata value m and with $MV_{[M]}$ the complete set obtained by matching a set of metadata values MV with a set of

metadata types M. Given a set of comparers MC, we denote with $MC(M)$ the subset made of all the comparers for the metadata type M. Finally, given the per-target partition $\{KMV, RMV, PMV\}$ of the metadata values specified for a kernel K, we denote with $PMV(p)$ the set of metadata values for the parameter p, for each parameter p of K.

Definition 10. [Equivalence of sets of metadata values] Let P be a pipeline, M the set of metadata types used by its components and MC the set of metadata comparers. Let $\{KM, RM, PM\}$ be the per-target-type metadata partition of M.

Let K be a kernel and $Pars$ its set of parameters.

Given MV_1, MV_2 two sets of metadata values, let $\{KMV_1, RMV_1, PMV_1\}$ and $\{KMV_2, RMV_2, PMV_2\}$ the respective per-target metadata-value partitions of $MV_{1[M]}$ and $MV_{2[M]}$ (complete sets of metadata values).

We say that MV_1 is equivalent to MV_2 for the pipeline P ($MV_1 \equiv_P MV_2$) if and only if all the following conditions are satisfied:

- $\forall m_1 \in KMV_1, m_2 \in KMV_2 : M = typeof(m_1) = typeof(m_2) \rightarrow m_1 \equiv_{MC(M)} m_2$
- $\forall m_1 \in RMV_1, m_2 \in RMV_2 : M = typeof(m_1) = typeof(m_2) \rightarrow m_1 \equiv_{MC(M)} m_2$
- $\forall p \in Pars : \forall m_1 \in PMV_1(p), m_2 \in PMV_2(p) : M = typeof(m_1) = typeof(m_2) \rightarrow m_1 \equiv_{MC(M)} m_2$

When the FSCL compiler is instantiated, it collects the set of metadata types exposed by the steps and processors in the pipeline, together with the related comparers. For each kernel node of the KFG, metadata-values associated to the kernel, to the parameters and to the return type are extracted and collected into separated sets. Then, each individual set is matched (definition 9) with the corresponding set of metadata types collected at compiler-instantiation time. If a value of a metadata type is not in this set, the value is discarded. If instead the value of a metadata type is missing, a default value is created. The resulting complete set of metadata values is stored with its target for later use, as explained in the next section.

Compilation invariance and Kernel Module

In the previous sections we formalized the concept of *kernel equivalence* and of *metadata equivalence*. Kernel equivalence defines when two kernel nodes represent the same input of the kernel-compilation pipeline. Metadata equivalence

instead defines when, given two sets of metadata values, the kernel-compilation pipeline behaves in the same way regardless which set is used.

In this section we merge the two concepts to define when the compilation is *invariant* to a kernel.

Definition 11. [Kernel compilation invariance] Given a kernel-compilation pipeline P and, two input kernels k_1 and k_2, the respective sets MV_1 and MV_2 of metadata-values, we say that P is invariant to the transformation of input $(k_1, MV_1) \leftrightarrow (k_2, MV_2)$ if and only if the following conditions are satisfied:

- $k_1 \equiv_{ker} k_2$
- $MV_1 \equiv_P MV_2$

This definition is particularly important because it states that the kernel-compilation part of the native pipeline *is a function of the input kernel and of its set of metadata values*. In other terms, given two FSCL kernels, we only need to obtain the respective MethodInfo objects (or the AST of the body in case of lambdas) and the sets of metadata values associated to them to know whether the compilation of two kernels will produce the exact same output.

This information is extracted by the kernel parsing step, which is the second step of the native compiler pipeline, and used to instantiate a *Kernel Module* (*KM*) data-structure for each kernel node. Once instantiated, this data-structure is filled and modified by the rest of the kernel-compilation pipeline steps. The only immutable information stored in a Kernel Module is the identifier of the kernel (MethodInfo or AST) and the set of metadata values, which together identify the module (section 7.4.3).

The other information, progressively built during the compilation, is summarized in the list below:

Kernel data The set of information about the kernel, such as the list of parameters, the return type and the AST of its body.

Functions data The set of information about utility functions called by the kernel.

Directives The list of directives that need to be inserted when generating the target code. An example is *#pragma OPENCL EXTENSION cl khr fp64: enable*, which is a directive required to enable double precision in OpenCL.

Global types The set of custom types (e.g. structs, tuples) used in the kernel or in the utility functions.

Code The OpenCL code generated for kernel, functions, global types definitions and directives.

Kernel and function data contain additional, structured information, such as the set of original and generated (e.g. explicit size for arrays) parameters. In addition, for each parameter the compiler inspects the AST to determine the way it is accessed (read, write, read and write) and stores this information together with the parameter itself. This information coming from AST analysis is exploited by the runtime to optimise buffer allocation and reuse during execution (chapter 8).

From the definition 11, we can derive the following definition of *equivalence of Kernel Modules*, which is the property used to check if the kernel cache contains an already-compiled equivalent kernel.

Definition 12. [Kernel Module equivalence] Given a kernel compilation pipeline P and two input kernels k_1 and k_2, let km_1 and km_2 the Kernel Modules resulting from respectively parsing k_1 and k_2. Let kid_1, kid_2, MV_1 and MV_2 the identifiers (MethodInfo or AST node) of the two kernels and the sets of metadata values contained in the respective Kernel Modules. We say that km_1 is equivalent to km_2 for the pipeline P ($km_1 \equiv_P km_2$) if and only if P is invariant to the transformation $(kid_1, MV_1) \rightarrow (kid_2, MV_2)$

7.4.3 Native compiler components

The set of steps that form the native compiler pipeline can be partitioned into three groups: *computing-expression-wise*, *kernel-wise* and *function-wise*. This subdivision reflects the structure of an arbitrary computing expression. A computing expression contains a composition of elements, which can be custom/collection kernels or sequential functions. In its turn, a kernel/sequential function can call one or more utility functions.

The computing expression parsing step is the first step of the native compilation pipeline. As already described, the purpose of this step is to build the KFG to represent the composition of computing elements in the expression. After the KFG has been built, the kernel-wise steps are applied to each node of the KFG that represents a custom/collection kernel in order to generate the corresponding OpenCL source representation.

The first kernel-wise step, which is the second of the entire pipeline, extracts the identifier of the kernel and the associated metadata values to inspect the cache for equivalent kernels in order to save compilation time. If no equivalent kernel is found, the compilation proceeds. As like as part of the set of native steps is applied independently to each kernel node, a subset of the kernel-wise steps is applied independently to the kernel function and to each utility

function called by the kernel or by another utility function. This subset is referred to as the set of *function-wise* steps.

In figure 7.6 we illustrate the set of native steps and we identify the three groups of steps discussed.

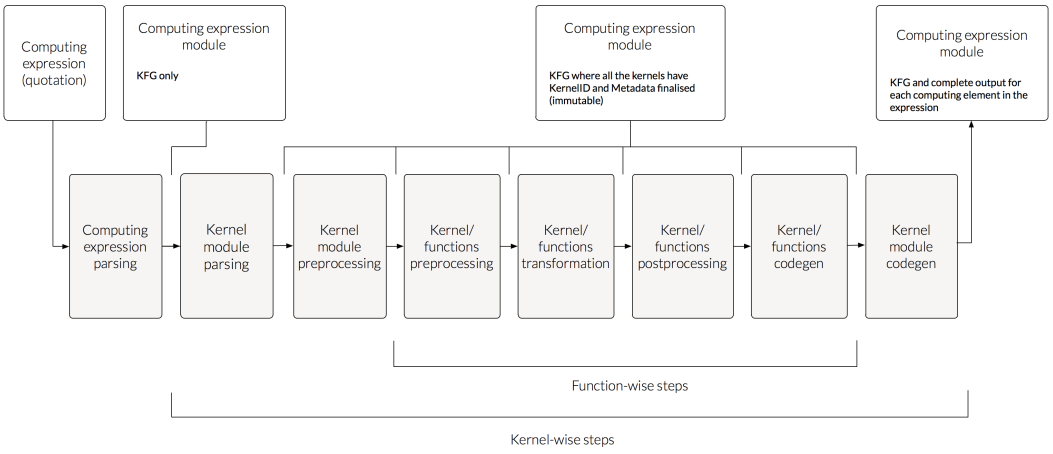

Figure 7.6: Steps of the native compiler pipeline in the order or execution

In the following list we briefly describe each step of the native pipeline.

Computing expression parsing (computing-expression-wise) As already told, this is the step of the pipeline that parses the input expression and instantiates the KFG.

Kernel module parsing (kernel-wise) this is the step of the kernel-pipeline that parses an FSCL kernel and instantiates the Kernel Module data-structure with the least set of information needed to identify it.

Kernel module preprocessing (kernel-wise) This step analyses the body of the parsed kernel to extract all the information that are shared between the kernel and the utility functions called by the kernel. The first action of the step is to spot all the calls to reflected functions from within the kernel body. For each of these functions a new data-structure to hold the function private information is created and stored in the Kernel Module. Then, the body of the kernel and the one of each utility function is inspected to detect the set of user-defined data structures used. This information is stored and used to generate an OpenCL type definition for each custom data-structure. After this step, the Kernel Module contains

a "raw" placeholder for all the data that are shared module-wise by the kernel and its utility functions

Kernel and functions preprocessing (function-wise) For the kernel and for each function, the step performs some actions on the function header to make it closer to the target OpenCL representation. For example, additional parameters are generated to hold the size of array arguments, while return type and return expressions are collected, generating an additional parameter to hold the returned value.

Kernel and functions transformation (function-wise) This is the most complex step of the pipeline, whose purpose is to transform the AST to obtain an abstract representation of the code that is closer to OpenCL. Multi-dimensional arrays are replaced with the respective one-dimensional versions, appropriately manipulating the indices used to access them. Reference cells are replaced with sigleton arrays and optional/tuple values are replaced with references to the properly generated struct types.

Kernel and functions postprocessing (function-wise) Once the AST has been transformed to a normalized version, closer to the OpenCL representation, a final analysis of the tree is performed to collect statistics and other code information.

Kernel and functions codegen (function-wise) This step traverses the normalized AST of the kernel and of each utility function to generate the corresponding OpenCL source code. At the end of this step, each computing element in the module is associated to a string containing its OpenCL code.

Kernel module codegen (kernel-wise) The OpenCL source code of the kernel and of each utility function is assembled together and with directives and user-defined data-types definitions. The output is a valid OpenCL source representing the whole module, which can be passed to an OpenCL-to-binary compiler to generate the kernel executable. The source code is stored in the Kernel Module, which is then returned as the result of the pipeline.

7.5 Compiler configuration and extensibility

As introduced at the beginning of this chapter, the FSCL compiler is built to be completely customizable and extensible. Even if a detailed description of the compiler extensibility and customization infrastructure goes beyond the

scope of this Thesis, we dedicate this section to introduce the customization mechanism for two main reasons. The first reason is to underline the overall capabilities of the compiler project. Since the entire set of steps, processors and type-handlers can be replaced and extended, the FSCL compiler should be viewed as a *dynamic compilation infrastructure* more than just an F# to OpenCL source-to-source compiler. In fact, with the appropriate set of pipeline components, the project can implement any object-model transformation. The second reason is to consider the compiler from the perspective of configuration and deployment. From the beginning of this Thesis, we pointed out the need for an abstract and flexible heterogeneous programming and execution framework to fulfil the requirements of a very wide target of users. From the programming point of view, this implies providing a model that can be used by both "sequential" programmers, leveraging on the functional composition of well-known patterns, and by more experienced programmers capable of understanding and using custom kernels and imperative-style composition to express more complex algorithms. From the framework *usage* point of view, abstraction maps to the simplicity of deployment, while flexibility maps to the ability for the (experienced) users to extend and configure the framework to fit each one's specific needs.

The FSCL compiler configuration infrastructure is based on automatic and transparent self-configuration whenever the compiler runs on a platform for the first time. This includes retrieving essential platform information and producing and storing configuration files for later use. Users with no particular needs are completely unaware of this setup: the compiler is able to load, organize and use the set of native components without requiring any user input.

Whenever a particular, different configuration is required, the compiler gives the chance to define custom steps, processors and type-handlers and to plug them into the compilation pipeline. New components can be added to an empty compilation pipeline or to a pipeline already populated with the set of native components. In the first case the user defines a new compilation pipeline from scratch, while in the second he extends the native one.

Thanks to the information about inter-steps and inter-processors dependencies that can be retrieved via introspection/reflection, the configuration infrastructure is able to detect the order of execution of custom components, potentially merging them with the native ones. In particular, information *(Before, After)* is associated to each step and processor to declare arbitrary dependencies between user-defined components (for which the user has the control on the source code) and pre-existing native or custom components that have been already deployed, which the user has no control.

7.6 Conclusions

In this chapter we presented and discussed the FSCL compiler, which is the part of the FSCL framework responsible to build an abstract and structured representation of the composition of elements in a computing expression and to generate the OpenCL source, along with other information, for each kernel.

Since the compiler is generally invoked everytime a computing expression is executed, compilation efficiency is required in order to reduce the overhead at runtime (chapter 8). For this reason, the compiler maintains a cache of already-compiled kernels. The information stored are used to prevent full-compilation of a kernel when an equivalent version is found in cache.

To balance efficiency and flexibility of compilation behaviour, we introduced the concept of *equivalence* of kernels. The equivalence of kernels depends on both the input kernels and on the steps forming the compiler pipeline. In fact, the same input kernel may result in two different outputs depending on the meta-information associated to the kernel and used by a compiler step (or more) to drive its own behaviour. Once the equivalence of kernels and meta-data has been defined, the compiler can succesfully determine if the cache contains a kernel that is equivalent to the one being processed. If so, the definition of equivalence establishes that the cached data exactly match the output expected for the kernel. Therefore, the rest of kernel- and function-wise steps can be skipped and the content of the output Kernel Module for the kernel being processed can be cloned from the cache.

The compiler is designed to require zero-configuration. Once instantiated through its parameterless constructor, the compiler is able to process computing expressions using the built-in set of steps, processors and type-handlers. At the same time, the compiler offers fine-grained configuration capabilities. Programmers can extend or replace the native compilation pipeline with user-defined compiler components, which can ported across systems and loaded automatically whenever the compiler is instantiated.

Chapter 8

FSCL Runtime

Built on top of the compiler, the FSCL runtime is the part of the framework that handles OpenCL-to-executable compilation, scheduling and execution of FSCL computing expressions. In this chapter we present this part of the framework, discussing the results of our research in raising abstraction over traditional OpenCL host-side coding while keeping the runtime efficiency close to the one that characterize low-level OpenCL programs.

A very important part of our research consists in investigating a flexible and transparent strategy to exploit the heterogeneity of multi-device platforms through an adaptive, device-aware scheduling approach. The FSCL runtime scheduling engine, which represents the result of this research as well as a fundamental component of the framework, is discussed in chapter 9.

8.1 FSCL Runtime structure

From the perspective of the internal structure, the FSCL compiler and the runtime are similar to each other. The first similarity is that both the compiler and the runtime are based on a configurable and extensible pipeline of steps.

Whereas the set of steps of the compiler globally build the KFG representation of a computing expression and generate the OpenCL source for each kernel element (figure 7.4), the steps of the (native) runtime pipeline schedule each kernel on a device, produce the executable code from kernel sources and finally coordinate the execution on the OpenCL devices. The output produced by the runtime pipeline is the result of executing the computing expression (figure 8.1).

Since the abstract view of the runtime pipeline is based on the same concepts of steps and processors already described in section 7.2.1, in this chapter we discuss only the *native* components and behaviour of the runtime. The con-

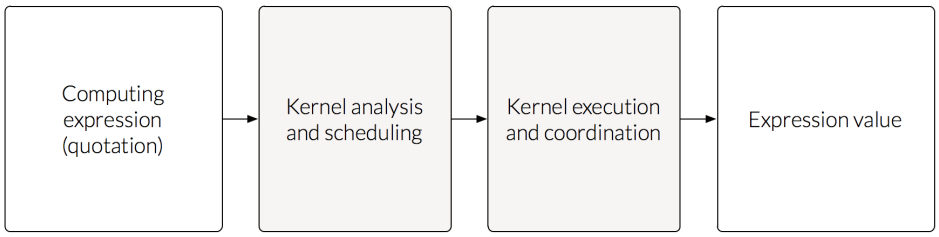

Figure 8.1: Steps of the native runtime pipeline in the order or execution

figuration and the extensibility features discussed in section 7.5 for the FSCL compiler apply with no relevant changes to the FSCL runtime.

In terms of framework components, the FSCL runtime is structured as shown in figure 8.2. Similarly to the compiler, the entire runtime framework is based on a configuration infrastructure and on a set of metadata. In the FSCL compiler, metadata are used to drive the compilation output. Runtime metadata are instead used to drive scheduling and execution.

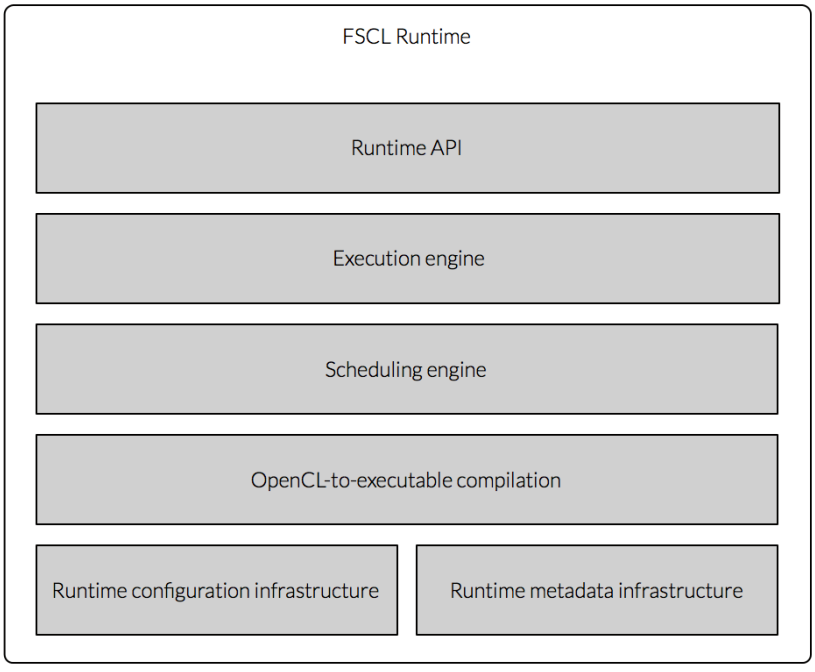

Figure 8.2: Structure of the FSCL runtime

On top of the configuration and metadata infrastructures there are three major components, which represent the native steps of the runtime.

OpenCL-to-executable compilation This is the step that interacts with the appropriate OpenCL compiler/client driver to obtain the executable code from OpenCL sources.

Scheduling engine The scheduling engine is the component responsible to determine on which available device to schedule each kernel in the input computing expression. As already said, the importance of this step leads us to dedicate an entire, separate chapter to the scheduling system (chapter 9).

Execution engine This is the step that handles the execution of each kernel on its target device, creating and managing all the OpenCL resources required and coordinating the flow of data among the computing elements in the expression. We discuss the relevant aspects of the execution engine in section 8.2.

On top of all the components, the FSCL runtime exposes an API. Whereas the compiler API allows the programmers to trigger the *compilation* process and to configure and extend the compiler pipeline, the runtime API allows to trigger the *execution* of a computing expression and to configure and extend the runtime.

It is important to note that the input of the runtime is a quoted computing expression (figure 8.1) and not the Computing Expression Module datastructure produced by the FSCL compiler. This is due to the fact that the FSCL runtime is not a component that "runs after" the compiler but that instead includes (or uses) the compiler.

In fact, even though the runtime and the compiler are two separated parts of the FSCL framework, the compiler is thought to be transparent to the developers of FSCL programs.

The typical flow of actions involving the programmer, the runtime and the compiler is illustrated in figure 8.3. The programmer asks the runtime to run a (quoted) FSCL computing expression. Under the hood, the runtime interacts with the FSCL compiler to generate the Computing Expression Module, which contains the Kernel Flow Graph and the OpenCL source of each kernel. Each kernel node in the KFG is then analyzed to determine the device where to schedule and execute it. When a device has been chosen, the runtime interacts with the appropriate OpenCL compiler to produce the kernel executable for the specific device. When the executable of each kernel has been produced, the runtime executes the expression, interpreting the composition operators

(function/collection composition) and running the kernels on each one's target device. The result of the whole computing expression is eventually returned to the programmer.

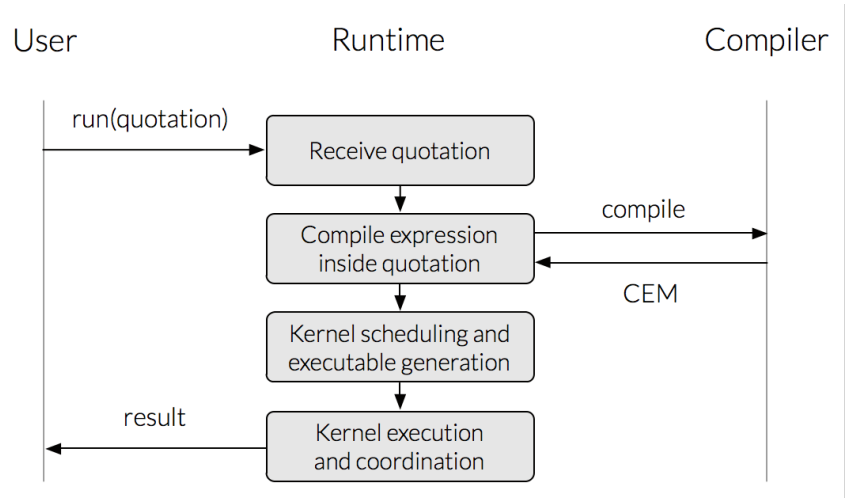

Figure 8.3: Interactions between programmer, runtime and compiler

The whole process, from FSCL-to-OpenCL compilation to scheduling and execution of kernels and composition operators, is completely transparent to the programmer. From the programmer's perspective, parallel execution of a computing expression is therefore not different from *evaluating* an F# expression from both the syntax and the semantic point of view.

Evaluating an expression is performed by calling the *Eval* method on the quotation object[1]. The result is the value of quoted expression.

```
let comp = <@
           input |>
           Array.map(fun i -> i * 2) |>
           Array.reduce (+)
         @>

// Evaluate the quotation
let result = comp.Eval()
```

Similarly, FSCL computing expressions are executed by calling the *Run* method on the quotation object. The result is, again, the value of the quoted

[1]The *Eval* extension method is part of the F# powerpack project. Recently, F# provided a native API to evaluate expressions

expression. From an abstract point of view, the major difference is that whereas *Eval* evaluates a computation in the traditional, sequential fashion, leveraging on the CLR, *Run* leverages OpenCL parallel execution[2].

```
let comp = <@
           input |>
           Array.map(fun i -> i * 2) |>
           Array.reduce (+)
         @>

// Run in parallel
let result = comp.Run()
```

Compared to OpenCL host-side coding to create, execute and coordinate kernels, FSCL host-side automatizes the whole process, reducing the programmer's effort to a single method call.

Investigating an expressive way to reduce the efforts spent in traditionally time-consuming host-side coding is one of the targets of our research. Thanks to the one-method API to run computing expressions and to the similarity between regular evaluation of an F# expression and parallel execution of an FSCL computing expression, programmers should find it easy and intuitive to develop and run FSCL programs.

8.2 Computing expression execution, caching and data management

In chapter 9 we present and discuss the scheduling strategy applied to each kernel in the input computing expression to determine the best device for execution. After scheduling each kernel on a device, the runtime handles the execution of the whole computing expression, that is of each node of the KFG.

As anticipated in section 6.2.4, kernels are executed on each one's target device according to the OpenCL model. All the required resources are created (e.g. context, command-queue, buffers), kernel arguments are prepared and finally the execution of the OpenCL kernel on the device is performed.

Data nodes are evaluated using the native F# quotation-evaluation feature. Similarly, sequential functions are executed on the host (CPU) leveraging the CLR and on the reflection API to pass the arguments and to apply the function.

[2]This is true if the computing expression contains collection kernels, sequential functions and collection/function compositions. In case of custom kernels, the "evaluation" using the *Run* method leads to the correct result, while the traditional *Eval* method would interpret (execute) the custom kernel in the wrong way

The remaining nodes to consider are the ones that represent compositions of elements. The FSCL runtime runs function and collection compositions by executing the operators concurrently whenever feasible. In other terms, different threads are spawned to execute independent sub-expressions (or the same sub-expression applied to different input data).

F# exposes three native operators for function composition, depending on the number of parameters of the functions considered. We can formally define the execution strategy of function composition as follows.

Definition 13. [Function composition execution] Given four computing expressions e_1, e_2, e_3 and e_4:

- $e_1 \mid> e_2$ is a computing expression that executes e_2 after e_1 completes (no concurrency).
- $(e_1, e_2) \mid\mid> e_3$ is a computing expression that executes e_3 after both e_1 and e_2 complete. The expressions e_1 and e_2 execute concurrently.
- $(e_1, e_2, e_3) \mid\mid\mid> e_4$ is a computing expression that executes e_4 after e_1, e_2 and e_3 complete. The expressions e_1, e_2 and e_3 execute concurrently.

Currently, the runtime supports multithread execution of two collection composition functions, which are *Array.map* and *Array.filter*. Even though these two functions alone already unleash high flexibility in expression composition, we plan to support additional functions in the future. The runtime behaviour of *Array.map* and *Array.filter* can be defined as follows.

Definition 14. [Array.map composition execution] Given a computing expression k of type $T \rightarrow U$ and an input array inp of type $T[]$,
$(Array.map\ k\ inp)$ is a computing expression that executes k concurrently on the elements of inp. If out is the output of the $Array.map$ execution, then $\forall_{i \in \{0,1,\ ..\ lenght(a)\}} out[i] = k(inp[i])$.

Definition 15. [Array.filter composition execution] Given a computing expression k of type $T \rightarrow bool$ and an input array inp of type $T[]$,
$(Array.filter\ k\ inp)$ is a computing expression that executes k concurrently on the elements of inp. Foreach $i \in \{0, 1,\ ..\ lenght(inp)\}$, let $ko = k(inp[i]) \in \{true, false\}$. If out is the output of the $Array.filter$, then $ko = true \iff el \in out$. In other terms, the output array contains only the elements of the input for which the k returns $true$. The relative order of the elements in the input collection is preserved.

In section 7.4.1 we illustrated the KFG for K-Means, a very popular algorithm for cluster analysis. For convenience, we replicate the KFG of the algorithm in figure 8.4.

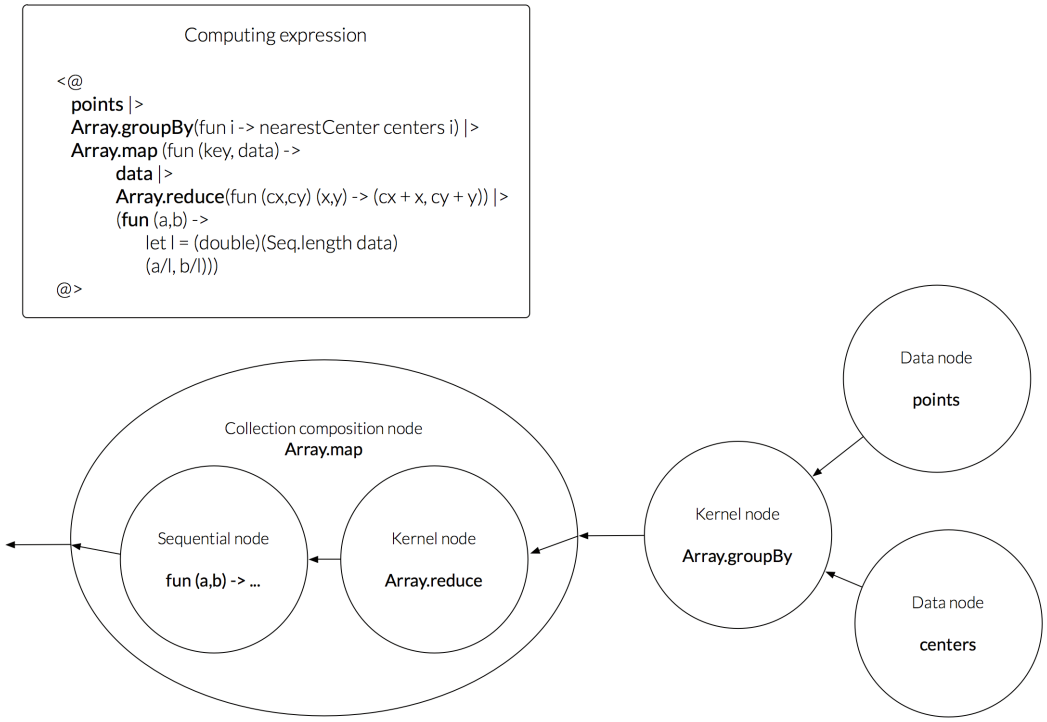

Figure 8.4: Kernel Flow Graph for the K-Means computing expression

In figure 8.5 we illustrate the steps performed by the FSCL runtime to execute K-Means. After *Array.groupBy* has been executed as a kernel on a device, the *Array.map* composition spawns a thread for each group generated by *Array.groupBy* (only two threads are considered in the illustration). Each thread executes the sub-expression that represents the collection operator using a different group as input. If concurrent instances of this sub-expression require to access the same device to run a kernel, kernel executions are enqueued onto the target device one after the other.

It is important to underline that the runtime *virtually* spawns a thread for each input element when executing a collection composition function. The current implementation of the runtime in fact uses a thread pool, which is more efficient than spawning a thread for each item of collections that contain thousands or millions of elements.

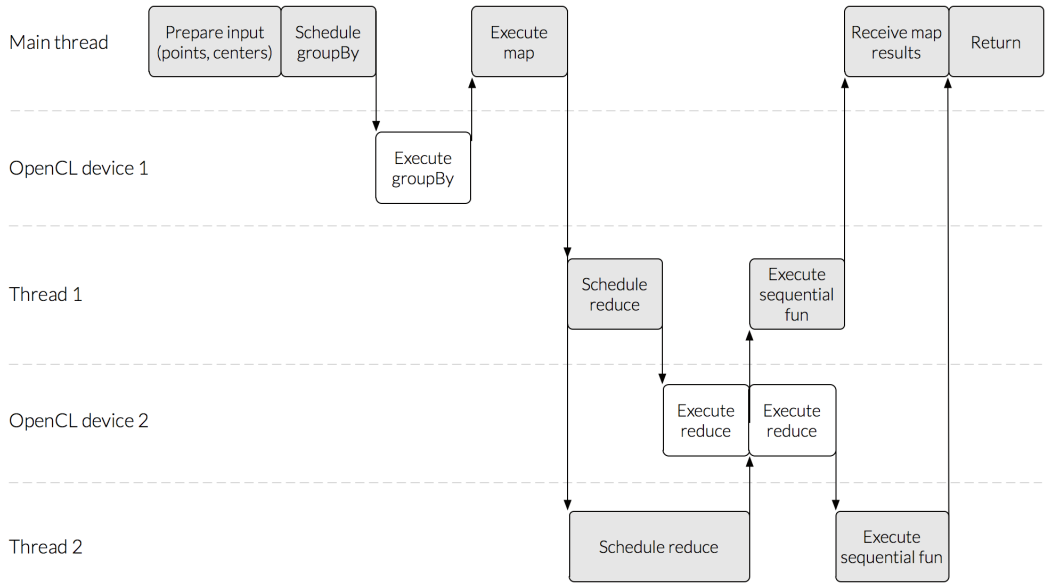

Figure 8.5: Timeline and interactions with OpenCL devices in executing K-Means

8.2.1 Device and kernel resource caching

To effectively run a kernel on an OpenCL device, three relevant data-structures must be allocated and properly initialized. The first is the *OpenCL context*, which is the environment where work-items executes. It includes references to buffers and command queues. The second is the *OpenCL device* pointer, which represents a specific OpenCL device among the ones available in the running system. The last one is the *OpenCL command-queue*, used to submit commands to a device, including transferring data and scheduling kernels for execution.

To reduce the overhead at kernel-execution time, the runtime allocates and initializes these information only once, the first time a device is chosen to execute a kernel. Once created, they are stored in a in-memory data-structure for successive usage (figure 8.6).

As like as for device-specific data, the runtime keeps a cache of compiled (binary) kernels. As shown in figure 8.7, the cache is conceptually structured hierarchically. The top level of the hierarchy stores kernels with different *OpenCL source code*. The level is made of a list of FSCL Kernel Modules, each of which includes all the information obtained when the FSCL kernel was

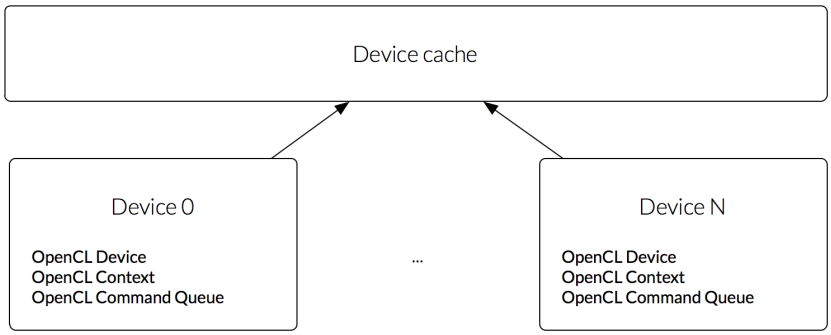

Figure 8.6: Device information stored by the runtime

compiled to OpenCL[3]. Since by definition of Kernel Module equivalence, two equivalent FSCL Kernels Modules have, among the other information, identical OpenCL sources, the set of FSCL Kernels Modules stored are *inequivalent* to each other. In other terms, if P is the compilation pipeline applied to kernels, for each pair of stored Kernel Modules k_1, k_2, $k_1 \not\equiv_P k_2$ (definition 12). For each FSCL Kernel Module the runtime stores the data relative to each OpenCL device where the kernel has been executed so far. These data include the executable code for the specific device (*OpenCL Program* and *OpenCL Kernel* objects). In other terms, the second level of the hierarchy stores kernel executables (for the same OpenCL source) with different binary code.

8.2.2 Data management

Since data processed by OpenCL kernels is stored into buffers, to execute a kernel the runtime must create an OpenCL buffer for each vector (array) parameter. In this section we use the term "vector" to refer to a collection of data that abstracts for the specific implementation (array, OpenCL buffer).

Allocating, initializing and transfering data across buffers has a cost that can overweights the cost of kernel execution. In order to reduce data-transfer and copy the runtime introduces a management layer that handles buffer creation, reusing and disposal.

Given an FSCL computing expression, we can identify two main types of vectors that flow through it: vectors that have both *managed* (i.e. array) and *unmanaged* (i.e. OpenCL buffer) *representations* (or *variants*) and vectors

[3]Since the FSCL compiler already stores the output of FSCL-to-OpenCL kernel compilation, the top level of the runtime cache hierarchy is actually made of pointers (references) to the matching elements in the compiler cache in order to avoid data duplication

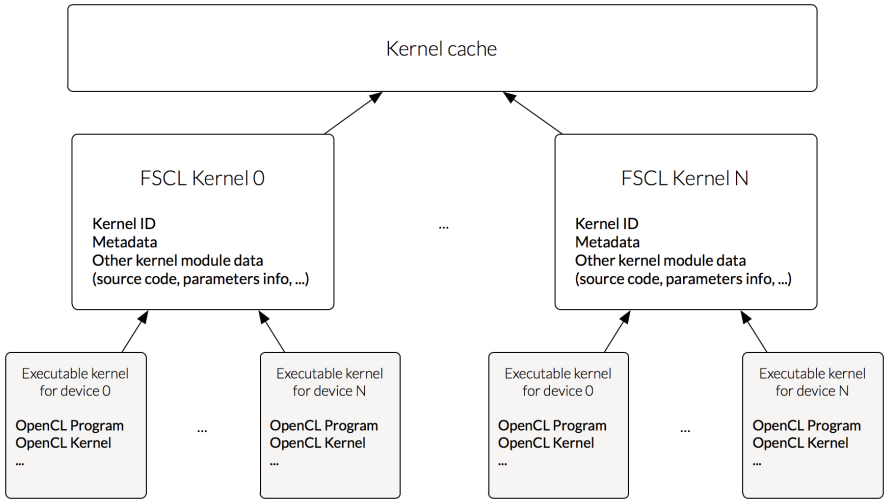

Figure 8.7: Kernel information stored by the runtime

that only need an unmanaged representation. A vector that requires both the variants is a vector that can be accessed from within the managed code (i.e. code executed on the host leveraging on the CLR), such as inside sequential functions and outside computing expressions. If a vector is instead accessible exclusively by kernels, no managed variant is needed, since OpenCL kernels work exclusively on (unmanaged) buffers.

We show some examples of data flowing through computing expressions to allow a deeper comprehension of which data require the managed variant and which data only need the unmanaged variant.

The first example (listing 8.1) only considers a function composition of custom and collection kernels. We use a bold font to highlight vector data accessed by the various computing elements. The vectors a, b and c represent the input of expression and inherently have a managed representation (array) created by the programmer. Note that the vector a is used by both the *Array.reverse* kernel and by *myCustomKernel*, while b is used by both *Array.map2* and by the utility function *utilityCall* of the *Array.map* function. Other vector data that flow through the computing expression are the results returned by *Array.reverse*, *Array.map2* and *Array.map*, which do not need a managed variant, since none of the intermediate outputs can be referred in the managed environment. An exception is the last kernel in the composition (*myCustomKernel*), whose output buffer is bound to the managed variable *result*. Since the content of the output produced by this kernel can be accessed from within the managed environment (in particular from within the host program)

both the managed and the unmanaged variants are needed.

```
let a = Array.zeroCreate<float32> 2.0f
let b = Array.zeroCreate<float32> 3.0f
let c = Array.zeroCreate<float32> 4.0f

let result =
 <@  (a |> Array.reverse, b) ||>
     Array.map2 myFunCall |>
     Array.map (fun el -> utilityCall el b) |>
     myCustomKernel a c
 @>.Run()
```

Listing 8.1: Example of function composition with corresponding vector data

In the case of computing expressions containing sequential functions and collection compositions, the distinction between data that require the managed variant and data requiring only the unmanaged variant is more complicated, as illustrated in listing 8.2. In the example, *Array.map* is used to coordinate a sub-expression made of a kernel call and sequential lambda. Since the operator binds each set (buffer) produced by *Array.groupBy* to a variable (i.e. *data*) the buffer requires a managed variant. This is particularly true since a sequential lambda is accessing *data*. Differently from *data*, the variable *key* is not considered, since it is a scalar value. The result of the execution of *myKernel* is another vector that needs a managed representation, since it is bound to the variable *output* referenced from within the sequential lambda.

```
let a = Array.zeroCreate<float32> 2.0f

let result =
 <@  a |>
     Array.groupBy (fun a -> a % 5.0f) |>
     // A collection composition
     Array.map (fun (key, data) ->
               // A sub-kernel
               myKernel data |>
               // A sequential lambda (function)
               fun output ->
                  data.Length - output.Length)
 @>.Run()
```

Listing 8.2: Example of function and collection composition with corresponding vector data

The examples proposed show the set of vectors that need a managed representation is the set of vectors that are bound to a name inside or outside the computing expression. We say "bound to a name" instead of "bound to a variable" since data allocated outside a computing expression may be associated to an object field or property and not necessarily to a local variable. In other terms, a managed variant is needed whenever vector data can be accessed outside kernels.

It is important to note that if a vector needs a managed variant then there is a way to access it from within managed code. The opposite statement is instead generally false, as demonstrated in listing 8.3. The example is identical to 8.2 except for the sequential function that doesn't access the vector bound to *data*. In the reshaped example, the vector bound to *data* doesn't require a managed variant, since no managed code is accessing its content. The execution may be performed by properly passing each unamanged buffer (group) produced by *Array.groupBy* to *myKernel* without allocating the corresponding managed array. The same reasoning holds for the buffer bound to *output*, which is not accessed by the sequential lambda.

```
let a = Array.zeroCreate<float32> 2.0f

let result =
   <@
       a |>
       Array.groupBy (fun a -> a % 5.0f) |>
       // A collection composition
       Array.map (fun (key, data) ->
                  // A sub-kernel
                  myKernel data |>
                  // A lambda not accessing data
                  fun output ->
                     ())
    @>.Run()
```

Listing 8.3: Example of function and collection composition with vector data that do not need managed variant

The FSCL runtime *estimates* the need for a vector to have a managed representation by checking whether the vector *can* be referred from within the managed environment. Even though this check is unprecise when sequential functions and collection composition are employed in a computing expression, it is more efficient than inspecting the AST of all the sequential functions involved to determine if a vector is effectively accessed from within code running on the CLR.

The runtime handles vectors that require a managed variant and vectors that do not require it using two different types of buffers, respectively called *managed buffers* and *unmanaged buffers*. Whereas an unmanaged buffer only contains a reference to an OpenCL buffer, managed buffers store the association between the managed (array) and unmanaged (OpenCL buffer) variants of vector data. The reason to handle the two kinds of vectors separately is driven by the fact that when a vector is bound to a name, it is likely to be referred (by name) multiple times, like in the example 8.1.

The runtime uses the managed representations to "compare" buffers[4], that is to check if an FSCL kernel is accessing a buffer already created. More precisely, the runtime checks if the argument of a kernel is a reference to an array that corresponds to the managed representation of a pre-existing buffer. If so, conceptually the unmanged variant (i.e. the OpenCL buffer) could be passed to the new kernel instead of creating and initializing a new one.

We say "conceptually" because to reuse the OpenCL buffer associated to the managed array some conditions must hold. The first requirement is that the buffer context must match the context of execution of the kernel whose argument is being processed. Since a different context is created for each OpenCL device, this means that the device for which the stored buffer was created must be the same device where the current kernel is scheduled[5]. The second requirement is that the memory flags possibly specified for the argument in order to drive OpenCL buffer allocation must be compatible with the ones associated to the stored buffer (section 8.3). For example, if the stored buffer has been created using the *AllocHostPtr* OpenCL flag while the argument being processed comes with the *CopyHostPtr* flag (to be used when creating the buffer for it), an incompatibility is found. If flags and context are compatible, the pre-existing OpenCL buffer is used. Otherwise, a new OpenCL buffer must be created.

Given these conditions, the FSCL runtime organizes managed buffers into an hierarchical data-structure, as illustrated in figure 8.8. The top-level indexing is given by the managed representations of the buffers (i.e. arrays). OpenCL buffers corresponding to a particular array are additionally grouped by context.

In figure 8.9 we illustrate the process of retrieving/creating a new managed buffer by checking the managed buffer cache and validating the context and the flags associated to the buffer.

A question that arises for managed buffers is how to keep them synchro-

[4]Array comparison is performed by reference

[5]We discuss potential improvements based on global analysis of the computing expression in section 14.1

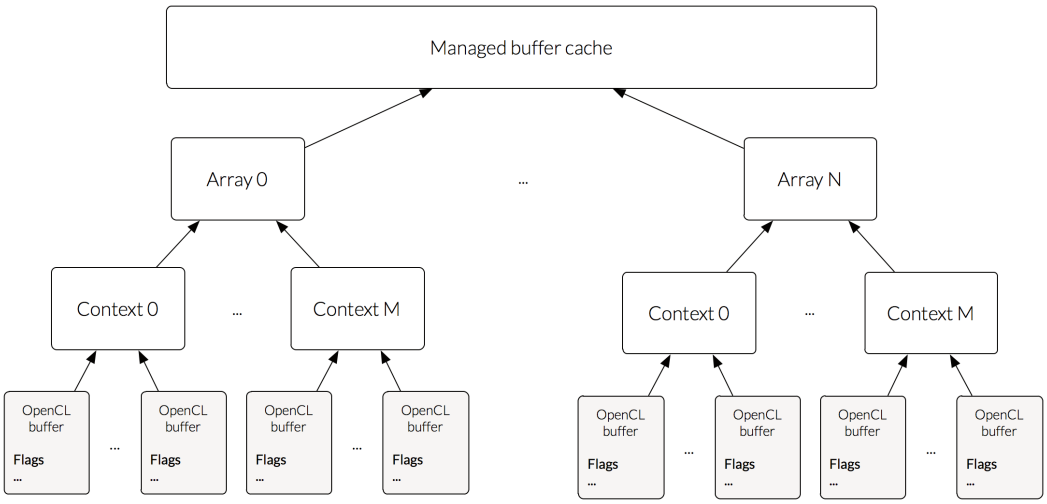

Figure 8.8: Structure of the managed buffers cache

nised during the computing expression execution. Thanks to the restriction for computing elements to be side-effect free, synchronisation is easy to accomplish.

In fact, the only buffers on which a computing element can write are the vectors created inside the element body, which are possibly returned and used by successive computing elements. Whenever a computing element creates a buffer that possibly requires a managed variant, a new array is created, which means the buffer will not match any pre-existing cached buffer (the address of the new array is not equal to the address of any array representing the managed variant of a cached buffer).

Given this, when a compatible pre-existing buffer is found in cache, the buffer can be used without the need of synchronisation. In fact, the buffer either correponds to an array declared outside the quotation containing the computing expression, or it corresponds to a buffer returned from a kernel and then bound to a managed variable (e.g. because it is used by a sequential function). In the first case, the buffer is read-only for all the elements in the expression. In the second, the only entity capable of writing the buffer is the computing element that created and returned it, which has been already executed.

For the same reasons, if the cached buffer is found incompatible because of the memory-flags or the context, the buffer can be copied without the need to keep the stored buffer and its copy synchronised with each other.

For what regards the synchronisation between the managed and the un-

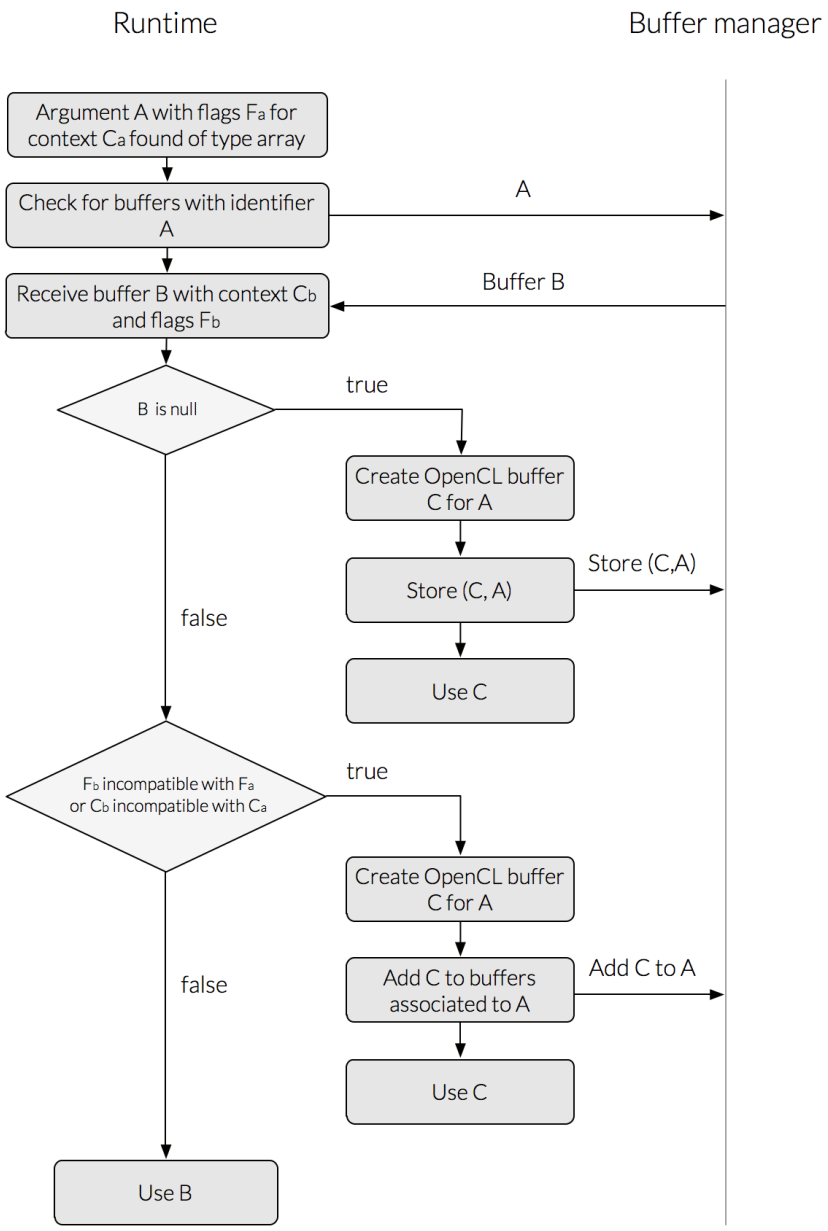

Figure 8.9: Process of creation of a managed buffer

managed variants of the same vector, data-transfer is required only two cases. The first is when an unmanaged buffer is firstly bound to a name, that is when the managed environment must access an unmanaged buffer for the first time. This happens whenever an unmanaged buffer is returned to the programmer or it is accessed by a sequential function or by the operator of a collection composition. In this case the unmanaged buffer is *promoted* to managed and the content of the array is read from the OpenCL buffer. Note that neither the array nor the OpenCL buffer can be modified at this point. The second situation where cross-variants data-transfer is needed is when an OpenCL buffer is required for the argument of a kernel that is a pre-existing array. If the cache check fails, a new managed buffer is created and the OpenCL buffer is initialised with the content of the array. Again, since this is the argument of a kernel, the kernel can't change it and the array cannot be modified as well (at least until the computing expression ends).

Since unmanaged buffers do not match any data accessible from within the managed environment, they are stored and reused in a different way respect to managed buffers. As already discussed, an OpenCL buffer can be reused only if the OpenCL context for which it is required is the same context used to create it. Therefore, unmanaged buffers are organized per-context. The set of buffers in the same context is stored in a list ordered by buffer size, as illustrated in figure 8.10. Buffer selection is performed using a first-fit approach.

When a new unmanaged buffer is required, the runtime inspects the list of pre-existing buffers with the appropriate context and picks the smallest buffer, in the order, whose size is greater or equal than the required one. The flags compatibility check is performed as for managed buffers. Once selected, the buffer information is updated considering the required buffer size to ensure data transfers and accesses are handled appropriately[6].

Unamaged buffers are locked during the execution of a kernel to avoid race-conditions due to multiple concurrent kernels that try to reuse the same pre-existing unmanaged buffer. In addition, since unmanaged buffers cannot be accessed by the managed environment, no synchronisation between variants is needed.

Other than managing buffers in order to reuse them, the runtime applies some optimisations to data allocation and transfer, which emulate the common principles of a good OpenCL programmer. For example, in case the target device is a CPU, buffers are created with some OpenCL memory-flags that

[6]The size of the buffer re-used must be conceptually updated to the size of the buffer required for the specific computation to prevent runtime errors due to a buffer size different from the one expected

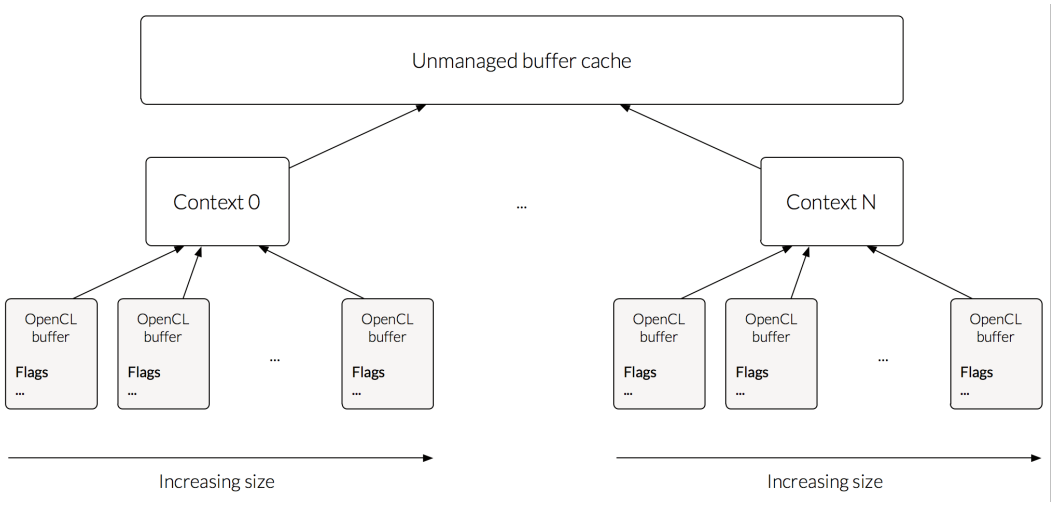

Figure 8.10: Structure of the unmanaged buffers cache

ensure no copy is performed[7][8].

Buffers that are created and returned from a kernel are allocated with *ReadWrite* flags, even if the kernel only writes them. This choice is driven by the consideration that if a buffer is created, written and returned from a kernel, it is likely that successive kernels will read it. Allocating the buffer with *WriteOnly* flags would incur data-copy whenever a kernel tried to read it (cause of memory-flags incompatibility).

All the optimisations applied by the runtime are local to a kernel. In other terms, no global analysis of data usage across the computing elements is performed. For example, a returned buffer is *always* allocated using *ReadWrite* memory flags, even if no successive kernels access it. We discuss possible improvements in chapter 14.

8.3 Scheduling and execution control via meta-data

The highest level of abstraction in coding the host side is given by completely automatic execution. To execute a parallel computation, the programmers

[7] *UseHostPtr* flag is used, passing the pointer to the already-allocated content of the buffer

[8] On discrete GPUs, copying data often results in higher performance, since data is placed and accessed in the GPU memory, which is faster than accessing host memory across the PCI-express bus

have only to call the method *Run* on a quoted expression and wait for the result.

The FSCL runtime exposes an additional level of abstraction for kernel execution, which allows control over scheduling, buffer allocation and data transfer, at the price of a lower transparency.

In section 6.2.5 and 7.4.2 we presented the concept of dynamic metadata and we discussed how they can be employed to drive kernel compilation. The same mechanism of metadata is also used by the runtime to enable control over kernel execution.

For what regards buffer allocation and data-transfer, there are three major aspects that the programmers can control. The first is the set of OpenCL flags used when creating a buffer, which can be set using the *MemoryFlags* attribute or the homonymous metadata-function. The second aspect is the way data is transferred between arrays and OpenCL buffers, controlled using the *BufferRead/WriteMode* metadata. Data transfer can happen calling the OpenCL *EnqueueRead/WriteBuffer* function or the *EnqueueMapBuffer* function, which maps the buffer into the host memory space.

Finally, programmers can force transferring data to and from buffers, overriding the default behaviour based on the results of access analysis [9] and on the context-dependent optimisations discussed in the previous section.

To override automatic scheduling, programmers can force the device where a kernel must be executed, using the *Device* metadata to declare the platform/device to use. If this metadata is associated to a kernel, the runtime skips kernel analysis/scheduling and executes the kernel on the device specified.

8.4 Multithread execution

The main purpose of the FSCL runtime is to handle scheduling and execution of FSCL kernels on OpenCL devices. Nevertheless, the runtime gives also the chance to run a kernel or an entire computing expression in *multithread* mode.

[9] Access analysis is performed by the *Function and kernel postprocessing* step of the native compilation pipeline (7.4.3)

To run an expression using the F# threading API, the developers have only to pass *RunningModel.Multithread* as the (optional) argument of the *Run* method, as shown in the following listing:

```
let comp =
   <@
     input |>
     Array.map(fun i -> i * 2) |>
     Array.reduce (+)
   @>

// Run using OpenCL
let result = comp.Run()
// Run using multithreading
let result = comp.Run(RunningModel.Multithread)
```

In case of multithread execution of a kernel, the runtime launches a number of threads equal to the number of CPU cores, each of which executes a subset of the work-groups. If the global size of the domain is Gs and the size of each group is Ls, then $Ng = \lceil Gs/Ls \rceil$ is the number of groups[10]. If C is the number of CPU cores, each thread executes a set of approximatively Ng/C groups. Work-groups within the set are executed in order of work-group ID. Work-items inside a group are executed in order of local ID.

Kernel arguments to which the *Local* metadata is associated, which corresponds to OpenCL *local buffers* (section 2.1), are handled appropriately by using a different array for each work-group executed. Similarly, OpenCL barriers are mapped to CLR barriers to guarantee the semantic of the computation in terms of synchronization points. For what concerns the rest of the kernel code, the FSCL kernel language provides a CLR-based implementation of the built-in OpenCL kernel-side functions [11]. This means that multithread execution can be handled correctly without any change to the code of the kernels in the computing expression.

There are two relevant reasons why we choose to support multithread execution:

Debugging capabilities Despite the integrated or third-party solutions for debugging and profiling, debugging OpenCL applications is often difficult and forces the programmers to learn and to get comfortable with very specific programs and tools. Thanks to multithread execution, it is

[10]For simplicity we consider a linear work-item domain, but the approach handles 2D/3D domains as well

[11]Math functions like *pown*, *mad*, *atan* and memory functions like *vload* and *vstore*

possible to test kernels and computing expression using traditional tools like *gdb* and integrated visual debuggers (e.g. visual debugging in Visual Studio or Xamarin Studio).

Fallbacking to multithreading Multithread execution not only represents a choice for the programmers, but it is also the execution mode automatically selected whenever the running system doesn't expose any OpenCL device[12]. Automatic fallback to multithread execution enhances the ubiquitousness of computing, allowing to port computations across a wide variety of systems without compromising the execution capabilities.

8.5 Conclusions

In this chapter we presented the FSCL runtime, which is the component of the FSCL framework responsible for raising abstraction over kernels and, more generally, computing expressions execution. From the programmers' point of view, executing a computing expression consists in a single method invocation on the quoted expression. The runtime transparently handles the interaction with the FSCL compiler, the scheduling on the set of available devices and the execution and coordination of the computing elements in the expression.

Despite the high abstraction delivered through an host-side API made of a single method call (compared to regular OpenCL host-side coding), the runtime implements a set of caching and optimisation strategies that allows to limit the execution overhead. In addition, the FSCL runtime exposes a set of metadata to control placement, initialization and data-transfer of buffers as well as to override the choices of the scheduling policy. Whereas custom kernels represent the low-level layer of kernel programming, this set of metadata represent the low-level layer of host-side programming.

Multithread execution is supported in place of OpenCL-based execution without any change to the computing expressions and to the custom/collection kernels they contain. This execution mode comes in handy to inspect and validate the runtime behaviour of computing expressions without the need to rely on a third-party OpenCL debugger. Since the runtime automatically fallbacks to multithreading when no OpenCL devices are found in the running platform, multithread execution represents also a way to enhance the ubiquitousness of FSCL programs.

[12]This may be due to driver malfunctioning or to the lack of devices supporting OpenCL

Chapter 9

Runtime scheduling engine

One of the most relevant aspects of our research is to investigate a way to help the programmer to exploit the heterogeneity of today's parallel platforms in a device-aware fashion. In this chapter we present and discuss the result of this reasearch, which is implemented in the scheduling engine of the FSCL runtime.

We can summarize the most relevant challenges of a scheduling policy as follows:

Flexibility and transparency A scheduling policy should require little to no user input. The scheduling infrastructure must be able to transparently and dynamically extract all the needed information from the set of devices populating the running system. If code analysis is required, the infrastructure should be trasparent also in retrieving information from the code to execute. The scheduling policy may be characterized by a different accuracy on different platforms and for different computations. Nonetheless, it should be able to execute without failures in any case, possibly misestimating certain device characteristics or ignoring certain parts of a computation in code analysis.

Efficiency at runtime Given the possibly generic and lightweight nature of the algorithms running on heterogeneous platforms, for a given computation a device may show a completion time that is only few milliseconds lower than the completion time on the other devices. Nevertheless, running the computation multiple times can turn a couple of milliseconds into a significant amount of time saved. Since the input may change across successive executions, the scheduling policy may need to be reapplied each time. If the time needed to select the best device for execution is higher than the per-execution time saved by running on such

device, the advantage of a device-aware scheduling would turn into a performance bottleneck. For this reason, one of the goals of the scheduling strategy is to be efficient.

9.1 General approach and strategy

Given a computation, it is possible to identify some aspects (or *features*) of its code that *imply* a certain runtime behaviour in terms of control flow and usage of hardware resources. Two of the most popular runtime aspects on which many researches have been conducted are branching behaviour and cache usage [50,52,54]. Some runtime aspects, such as the number of instructions and the amount of cache misses/evictions, have a deep impact on the completion time of a computation. Given these considerations, the basic idea on which our scheduling approach relies is to find the way certain code features are correlated to the completion time.

In the last few years, many analytical models have been proposed to predict the performance of specific device types, using code features and device characteristics [35, 43]. The major limitation of analytical models is that they often require to explicitly specify some code information or some parameters of the target architecture, which are difficult to retrieve or to estimate automatically. In addition, analytical models are generally unreliable in case one or more parameters of the model are missing or sensitive to errors in the estimation of their values. This makes analytical models difficult to employ in highly-dynamic systems, where the architecture parameters may be subject to changes on the basis of the specific platform where the model is evaluated. This is particularly true if the model is integrated in a library/framework that aims to hiding from the users the specific machine configuration.

Simulation- and profiling-based approaches enable a higher abstraction over the details of the running platform. These approaches generally leverage collecting runtime information (e.g. the trace of execution) for a surrogate of the program, obtained by sampling the input or building a restricted, lightweight version of the program code. A model is then applied to estimate the performances of the real program starting from the collected data [20, 74]. The main problem of these approaches is that runtime profiling and simulation may incur a significant overhead.

Machine-learning shows various advantages over analytical and simulation-based approaches. At first, machine-learning strategies for performance prediction and scheduling are naturally transparent to the programmers and highly adaptive. Secondly, machine-learning can capture subtle interactions between the software and the underlying system that are not expressed in analytical

models for sake of simplicity or because these interaction are actually unkown. Machine-learning approaches are also easy to refine through the extension of the training/feature set. Finally, machine-learning can introduce a substantial overhead in training the system, but applying the model built is generally efficient. This allows us to isolate most of the overhead of performance prediction at deploy time, strongly reducing the impact at runtime.

The last decade has seen much research towards statistical approaches to predict computing performance. In [76] random forests are used to build a performance prediction model, which achieves not satisfying R^2 values, indicating that a consistent portion of the information is not modeled properly. In [39] K-Nearest Neighbor is employed to predict completion time on the basis of code and input similarities, while in [37] Sparse Polynomial Regression is applied to a set of automatically selected features for completion time estimation. We think that linear regression should be favoured, because it is an easier approach, it offers an understandable model and it is suitable to progressive refinements. As stated in [37], some non-linear aspects may be impossible to model using linear methods. Nonetheless, linear regression is a feasible approach if the relation between the dependent variable and the explanatory variables (e.g. completion time and code features) is linear, which shifts the problem to meaningful feature selection. For example, whereas the size of the input matrices (n) has not a linear relation with the completion time in matrix multiplication (n^3 for sequential implementations), the completion time is linear on the number of operations/memory accesses. The problem is therefore to select the appropriate features to consider in order to predict the completion time.

The issue that arises with the heterogeneity of today's platforms is that the completion times on different devices may have different relations with specific sets of features. Some features that have a well-known impact on the CPU, such as the number of cache misses, may have less-to-no impact on the completion time of GPUs. Similarly, GPUs specific execution and memory models promote memory access patterns to optimise the usage of hardware resource (e.g. coalescing, reducing bank-conflicts) that negatively affect or do not significantly affect the completion time on CPUs. Therefore, the major issue is to determine which features should be used to reliably predict the completion time on a variety of different devices. In case of analytical approaches, a different model for each specific device or device type must be defined and applied. In case of simulation and profiling, the overhead introduced may not scale with the number of devices in the platform.

In order to combine flexibility and efficiency, we decide to adopt a machine-learning strategy that merges the analytical and the profiling approaches. The

machine-learning strategy that we employ starts from the definition of a process to perform code analysis and to extract a set of relevant features from arbitrary computations (kernels), leveraging the F# quotations mechanism. Once defined, the process is applied to a set of predefined kernels (*training samples*), which are also executed on each available device to determine the corresponding completion time. Training sample execution can be viewed as a way to *profile* the platform, in the sense of collecting runtime information on how the platform reacts to the scheduling of different computations. Given the set of features extracted from the training samples and the completion time on each device, we apply a regression algorithm in order to *build* a model for each device. Each model correlates the set of features to the completion time on a specific device.

For each kernel to execute, we apply the models built through regression to the set of features extracted from the kernel in order to estimate the completion time on each device. The scheduling policy finally selects the device with the lowest expected completion time.

In the rest of this chapter we discuss the details of our scheduling approach. We start presenting the process to extract features from arbitrary computations. Then, we introduce the regression model used to profile the platform in order to build a set of device models. Finally, we discuss how these two building blocks are joined in the FSCL runtime scheduling engine.

9.2 Code analysis and feature extraction

The main problem in code analysis for feature extraction is that for most computations the value of a feature depends on the input.

Like in [73], we aim at reducing the runtime scheduling overhead. Instead of separating features into a static and a dynamic set, which makes it difficult to capture information depending on both the program structure and the specific input, we statically [1] *precompute* features, analysing the Abstract Syntax Tree (AST) of the kernel and building a *finalizer* for each feature, which completes the evaluation of the feature value as soon as the input is known.

To formally define the concept of *feature finalizer*, we consider a given kernel k and we indicate with $body(k)$ the root node of its AST and with $pars(k) = [\ p_1,\ p_2,\ ..\ p_n\ ]$ the list of kernel parameters. Let $val_{f,\ k,\ a_1,\ ..\ a_n}$ be the value of a feature f for a kernel k given a set of actual arguments, one for each parameter. The feature finalizer fz for a feature f and a kernel k consists

[1] With "statically" we mean at kernel compilation time, that is the first time a particular kernel is seen

in a function that takes the same arguments of k and returns an estimation of $val_{f,\ k,\ a_1,\ ..\ a_n}$.

Definition 16. [Feature finalizer] Given a kernel k, $pars(k) = [p_1,\ p_2,\ ..\ p_n]$ its parameters and $[t_1,\ t_2\ ..\ t_n]$ the parameter types. Let f be a feature, tf its type and $val_{f,k,\ a_1,\ ..\ a_n}$ its value for k given a set of actual arguments.

A feature finalizer is a function fz, where:

$$fz : t_1 * t_2 * ..t_n \rightarrow tf \text{ such that } fz(a_1, a_2,\ ..\ a_N) \cong val_{f,k,\ a_1,\ ..\ a_n}$$

In the F# object model a feature finalizer is an anonymous function (lambda) that encapsulates the code to compute the feature value that cannot be statically evaluated because of a dependency on the kernel input.

Feature finalizers are computed the first time a kernel is seen and stored in a data-structure for future use, as illustrated in figure 9.1.

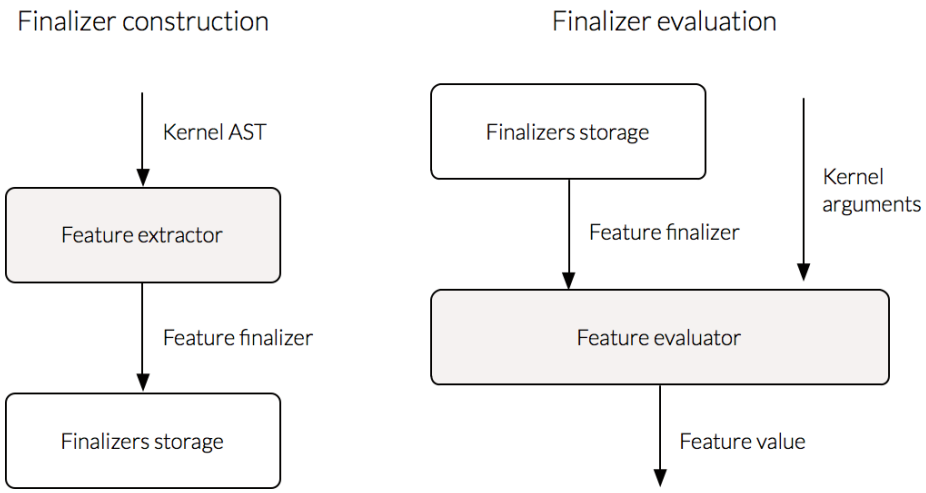

Figure 9.1: Finalizer construction and evaluation

Once built, a feature finalizer can be applied to multiple, different sets of kernel arguments to retrieve the matching feature value. Thanks to the static analysis of the kernel AST and the construction of a lambda that generally contains very lightweight code[2], the overhead of completing feature evalutation at kernel execution time, which corresponds to applying the lambda to the kernel arguments, is mostly irrelevant.

[2]For features counting particular constructs in a kernel, such as the number of memory accesses, the finalizer code contains only few arithmetic operations

Building a feature finalizer for a particular kernel consists in mapping the kernel to a lambda function, preserving the set of parameters but replacing the body. Figure 9.2 shows this mapping between a matrix multiplication kernel and a feature that counts the number of accesses to the elements of the input arrays.

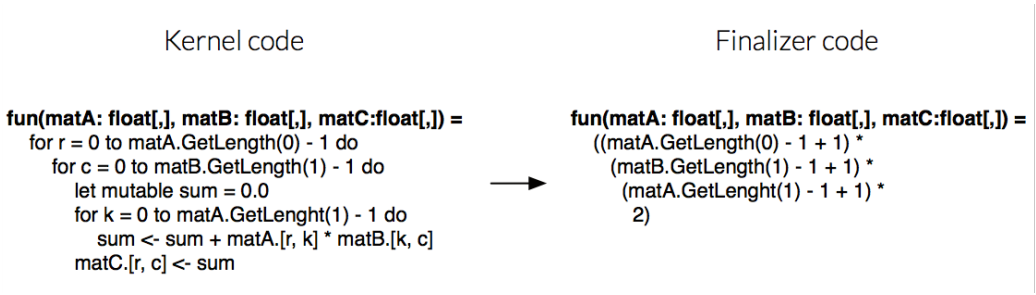

Figure 9.2: Kernel to finalizer mapping for a feature that counts memory reads

9.2.1 Feature finalizer building

In this section we illustrate the process of mapping the AST of a kernel to the AST of the finalizer for a particular feature. For simplicity, we only consider features that estimate the occurrence of certain events at runtime (e.g. the number of memory accesses) through counting the occurrence of specific language constructs in the kernel body. For this type of features, the finalizer building process is parametric on the set of constructs that contribute to the feature value. The process can be specialized and adapted to extract features that have no one-to-one relation with the occurrence of specific language constructs, such as the estimated number of cache misses. We discuss an example of specialized finalizers in section 9.2.2.

Throughout this section, we use the terms "mapping", "processing" and "generating" to refer to the creation of a finalizer AST node from a kernel AST node. In addition, we often say that an action (e.g. sum, subtraction, ceiling) is performed on the values of the feature computed for the children of a particular node. To be formally correct we should say that, given the results of processing the children of a particular node, a new node representing a particular action (e.g. a node representing a call to the *sum* function) is created and the results are set as its children. Finally, unless otherwise stated, we use the terms "AST node", "AST subtree" (rooted in a node) and "expression" (F# representation of an AST node/subtree) interchangeably.

The process of building a feature finalizer for a specific feature and a given kernel is based on three functions, respectively called *map*, *smap* and *dmap*.

The function *map* is responsible to define the mapping between any node n_{kernel} of the kernel AST and the corresponding node $n_{finalizer}$ of the finalizer AST, which expresses the value of the feature for the substree rooted in n_{kernel}. To perform this mapping, the function relies on *smap* and *dmap* as shown in the equation 9.1. The *map* function acts as a proxy: if the node to map is succesfully processed by *smap*, the result of this processing is returned as it is. Otherwise, the *dmap* function is applied to the input node.

$$map(n) = \begin{cases} v & \text{if } smap(n) = Some(v) \\ dmap(n) & \text{otherwise} \end{cases} \tag{9.1}$$

The function *smap* maps only "interesting" nodes of the kernel AST. In other terms, *smap* is responsible to define the value of the feature for the set of AST nodes that are interesting for the feature considered. For example, in case of counting memory read accesses, the interesting nodes are those representing the access to an array element. If applied to a non-interesting node, *smap* returns the special value *None*.

$$smap(n) = \begin{cases} Some(v), & \text{if } n \text{ is interesting} \\ & \text{where } v \text{ is the AST node that contains} \\ & \text{the feature value for the tree rooted in } n \\ None & \text{otherwise} \end{cases} \tag{9.2}$$

Finally, the function *dmap* is responsible to define the mapping of non-interesting nodes, possibly composing the values of the feature computed for the children of a node in order to express the feature value for the node itself. This function represents the generic part of the process. In other terms *dmap* is invariant to the specific feature considered, which only affects the definition of *smap*.

Given a node n, the behaviour of *dmap* is manyfold, depending on the specific type of node considered.

Variable binding nodes. A node in the form $\langle let\ x\ =\ y\ in\ body \rangle$ is mapped to a finalizer node $\langle n_y + n_{body} \rangle$[3], where n_y is the feature value for y and n_{body} the feature value for *body*.

Variable reference nodes and constant value nodes. Var references and constant values are mapped to $\langle 0 \rangle$ (i.e. to a node holding the value 0).

[3]The operator (+) stands for an AST node encapsulating a call to the "sum" function

Conditional nodes. Conditional nodes are handled returning the average of the feature value computed in the if-branch and the one computed in the else-branch. In other terms, a node is in the form $\langle if\ c\ then\ a\ else\ b \rangle$ is mapped to a finalizer node $\langle n_c + (0.5 * n_a + 0.5 * n_b) \rangle$, where n_a, n_b and n_c are, respectively, the feature value in a, b and c.

For loop nodes. A node in the form $\langle for\ i\ in\ a\ ..\ s\ ..\ b\ do\ c \rangle$[4], where s is the increment of i in each iteration, is mapped to the finalizer node $\langle n_a + n_b + n_s + \lfloor (b-a+1)/s \rfloor * n_c \rangle$, where n_a, n_b, n_c and n_s are, respectively, the feature value in a (the initial value of the iteration variable), b (the final value of the iteration variable), c (the loop body) and s (the update of the iteration variable). The expression $\lfloor (b - a + 1)/s \rfloor * n_c$ means that the feature value for a loop node is obtained multiplying the feature value for the body of the loop by the trip count of the loop (e.g. the numer of accesses performed globally by a loop is equal to the number of accesses in the loop body multiplied by the loop trip count).

While loops. The finalizer builder is able to estimate the trip count for a limited by heavily used variety of while loops. The first constraint is that the guard of the loop must be in the form $\langle var\ op_c\ expr \rangle$, where var is a variable and op_c a comparison operator. The second is that the variable var must be updated only once and in the form $var \leftarrow var\ op_a\ expr$, where op_a is an arithmetic operator. Finally, $expr$ must not depend on mutable or iteration variables.

For example, a node in the form $\langle while\ v\ <\ a\ do\ b \rangle$, where v is declared as $\langle let\ v\ =\ c \rangle$ and updated in the form $\langle v \leftarrow v * d \rangle$ is mapped to the finalizer node $\langle \lfloor log_d(a - c) \rfloor * n_b \rangle$ where n_b is the feature value for the loop body b. While loops that do not respect this contraint are mapped to $\langle 0 \rangle$ (i.e. they do not contribute to the feature value).

Nodes that represent a call to a kernel utility function. A node in the form $\langle utilityFun(args) \rangle$ is mapped to a node representing a call to the finalizer built for $utilityFunction$. This means that when a call to an utility function is found, the entire finalizer building process is recursively applied to it, which results in a finalizer associated to the utility function. In other terms, if $fz_{utility}$ is the finalizer for the function $utilityFun$, then a node in the form $\langle utilityFun(args) \rangle$ is mapped to $\langle fz_{utility}(args) \rangle$.

Other non-interesting nodes. Non-interesting nodes that are not considered in the preceding cases are mapped to the sum of the feature values

[4]This syntax includes also F# loops in the form $\langle for\ i\ =\ a\ to\ b\ do\ c \rangle$, where the increment of the variable is implicitly set to 1

computed for their children. For example, a node representing a cast (e.g. $\langle (float)expr \rangle$) is mapped to $\langle n_{expr} \rangle$, which is the feature value for the child *expr*. A call to a function that is not an utility function (e.g. $\langle Math.Pow(expr_1, expr_2) \rangle$ is mapped to the sum of children's feature values $\langle n_{expr_1} + n_{expr_2} \rangle$).

We summarize the *dmap* function in the equation 9.3[5].

$$dmap(n) = \begin{cases} dmap_{let}(val, b) & \text{n = Let(var, val, b)} \\ 0 & \text{n = VarRef(var) or} \\ & \text{n = Value(val)} \\ dmap_{seq}(fst, snd) & \text{n = Sequential(fst, snd)} \\ dmap_{if}(cnd, ifb, elb) & \text{n = IfThenElse(cnd, ifb, elb)} \\ dmap_{for}(st, sp, en, b) & \text{n = For(var, st, sp, en, b)} \\ dmap_{while}(st, en, up, b) & \text{n = While(var} < \text{en, b) and} \\ & var \text{ value before the loop is } st \text{ and} \\ & var \text{ is updated in } b \text{ as } var \leftarrow var * up \\ dmap_{util}(args) & \text{n = Call(uf, args) and } uf \text{ is an} \\ & \text{utility function} \\ dmap_{oth}(ch) & \text{otherwise, where } ch \text{ are the} \\ & \text{children of } n \end{cases} \tag{9.3}$$

[5]For simplicity we express only the mapping for while loops where the variable is multiplied by an expression in each iteration, but a similar mapping can be expressed in case of sum, substraction or division

$$
\begin{aligned}
dmap_{let}(val, b) &= map(val) + map(b) \qquad (9.4)\\
dmap_{seq}(fst, snd) &= map(fst) + map(snd)\\
dmap_{if}(cnd, ifb, elb) &= map(cnd) + 0.5 * (map(ifb) + map(elb))\\
dmap_{for}(st, sp, en, b) &= map(st) + map(en) + map(sp) + \\
&\quad \left\lfloor \frac{en - st + 1}{sp} \right\rfloor * map(b)\\
dmap_{while}(st, en, up, b) &= \lfloor log_{up}(en - st) \rfloor * map(b)\\
dmap_{util}(args, fz) &= \sum_{a \in args} map(a) + fz(args)\\
dmap_{other}(children) &= \sum_{c \in children} map(c)
\end{aligned}
$$

Note that *dmap* calls *map* on the children of the input node in order to guarantee that every interesting node in the AST is processed by *smap*.

Similarly, the function *smap* can rely on *map* to compute the feature values for the children of an interesting node. For example, if the set of interesting nodes is made of calls (applications) of arithmetic operations, the function *smap* described in the equation 9.2 can be specialized as follows.

$$
smap_{arith}(n) = \begin{cases}
1 + map(a) + map(b) & \text{if } n = a + b \text{ or}\\
& n = a - b \text{ or}\\
& n = a * b \text{ or}\\
& n = a/b \text{ or}\\
& n = a\%b\\
None & \text{otherwise}
\end{cases}
$$

Given the definition of *map*, *smap* and *dmap*, the process of building a feature finalizer for a feature f and a kernel k can be described as a function

that returns a lambda with the same parameters of k and with the body generated by applying *map* to the root of k, using a specialization of the *smap* function according to the specific feature to extract.

Definition 17. [Feature finalizer building process] Given a kernel k, be $root(k)$ the root of the AST of k and $pars(k)$ the set of parameters of the kernel. We indicate with $fun\langle P\rangle \rightarrow \langle B\rangle$ a lambda with a set of parameters P and a body that is represented by the AST stored in B.

The process of building a finalizer for a kernel k and a feature f, given $smap_f$ the function to map interesting nodes for f is described by a function $fzbuild$ defined as:

$$fzbuild(smap_f, k) = fun\ \langle pars(k)\rangle \rightarrow \langle map_{[smap_f]}(root(k))\rangle$$

were $map_{[smap_f]}$ is the fuction defined in equation 9.1 where the call to *smap* is replaced by a call to $smap_f$.

Finalizer AST validation

When considering the behaviour of a finalizer, in terms on the nodes that populate its AST, there are some restrictions that should be applied for sake of correctness and efficiency. For example, the finalizer should not contain references to local or iteration variables declared in the kernel, since none of those variables are part of the finalizer AST. More generally, we define *valid finalizer nodes* the set of AST nodes that can appear in a finalizer AST.

Definition 18. [Valid finalizer node] Given an AST node n, we say that n is a valid finalize node if and only if it expresses one of the following constructs.

1. Constant value;
2. Reference to a parameter;
3. Call to a function or access to a property related to the length/rank of an array parameter, where the arguments are valid nodes;
4. Call to an arithmetic/OpenCL built-in function, where the arguments are valid nodes;
5. Call to a function related to the work-item domain (e.g. global-size, work-group ID), where the arguments are valid nodes;
6. Call to the feature finalizer relative to a kernel utility function where the arguments are valid nodes.

We assume that the function *smap* always produces valid finalizer nodes. Assuming that both *dmap* and *smap* produce valid nodes, the function *map* produces valid nodes as well, since it simply forwards the result of either *smap* or *dmap* depending on whether the input node is interesting or not. The only function that can generate invalid nodes is therefore *dmap*. To determine the cases where this function can produce invalid nodes, we consider the equation 9.3. For each case we apply induction, assuming that the result of applying the function to the children of a node returns a set of valid nodes and verifying whether the composition of these nodes results is a valid node.

In case of a let-binding, a variable reference, a constant value or a branch, which are the first four cases of equation 9.3, the nodes built represent arithmetic operations applied the results of processing the children of the input node (case 4 of definition 18). This means that if processing the children produces valid nodes, the node built as the combination of them is valid.

The result of processing a node that matches the last case of equation 9.3 (other non-interesting nodes) is valid, since it represents the sum of the nodes generated for the children of the input.

The only cases in which *dmap* can generate invalid nodes are therefore when the input node is a *for* loop, a *while* loop or a call to an utility function. This is because in these particular cases some nodes of the kernel AST become part of the finalizer AST "as they are", without being processed by the *dmap* function. In case of a call to an utility function, the nodes of the kernel that become part of the finalizer are the arguments of the call. In case of a loop, the nodes preserved in the mapping are the expressions of the boundaries of the iteration variable.

In the sample 9.1 we illustrate a kernel for which the finalizer building process to extract the number of memory reads generates an AST containing an invalid node.

```
let myKernel(a:float[], b:float[], c:float[]) =
   let count = a.Length
   let endIdx = count - 1
    for i = 0 to endIdx do
      c.[i] <- a.[i] + b.[i]
```

Listing 9.1: Memory read count feature depending on local variable

According to the rules defined in thre equation 9.3, the finalizer generated is the one shown in listing 9.2. The finalizer body contains a reference to the kernel local variable *endIdx*, violating the rules expressed in the definition 18. Note that other than being an invalid node, the local variable reference introduces a free variable in the function context, which prevents the function from being evaluated[6].

```
fun (a:float[], b:float[], c:float[]) ->
    // (a.Length) feature value
    0 +
    // (For loop) feature value
    (endIdx + 1) *
    // (Var set) feature value: 0 left hand, 2 right hand
    (0 + 1 + 1)
```

Listing 9.2: Memory read count finalizer

Since local variables are extensively used in most of the kernels, we define a *variable unfolding* procedure to allow feature values to depend on kernel local variables, lifting references to such variables when generating the finalizer AST in order to obtain a valid finalizer.

To implement this procedure, while traversing the kernel AST we keep track of the immutable local variables and of their values using a stack σ. When the function *dmap* generates finalizer nodes containing kernel nodes (i.e. in case of processing *for*/*while* loops or utility function calls), the variable unfolding procedure recursively processes these kernel nodes, replacing each variable reference with the corresponding value in the stack.

The process doesn't support unfolding of mutable variables, since tracking the updates of a mutable variable is generally unpredictable. Therefore, if the variable considered for unfolding is mutable, the unfolding process is interrupted and the kernel node is mapped to $\langle 0 \rangle$ (i.e. the node doesn't contribute to the feature).

In the example 9.1, when processing the *for* loop node the variable unfolding procedure is applied to the nodes *endIdx* (the final value of the iteration variable) and to *0* (the initial value of the iteration variable). The local variable *endIdx* is replaced with its value in σ, which is $count - 1$. Since this expression contains another local variable reference (i.e. *count*), the process is recursively applied to replace *count* with its value, which is $a.Length$. The expression resulting from variable unfolding is therefore $a.Length - 1$, which doesn't contain any local variable reference.

[6]Trying to evaluate a quotation containing the function definition triggers an error, reporting that the free variable is not defined in the translation context

Given the application of variable unfolding, the finalizer AST becomes the one shown in the sample 9.3, which unlike the one reported in the sample 9.2 contains no free variables and can be evaluated.

```
fun (a:float[], b:float[], c:float[]) ->
    // (a.Length) feature value
    0 +
    // (For loop) feature value
    (a.Length - 1 + 1) *
    // (Var set) feature value: 0 for left hand, 2 for right
      hand
    (0 + 1 + 1)
```

Listing 9.3: Memory read count finalizer with variable unfolding

After variable unfolding has been applied, the *dmap* function checks that the resulting expression is valid according to the definition 18. In the sample 9.4 we show a case where the validation fails. When processing the *for* loop, the *dmap* function firstly applies variable unfolding to $endIdx$ and to 0, respectively obtaining the expressions $myCustomFun(a.Length) - 1$ and 0. Then, the resulting expressions are validated. Whereas 0 is a valid node (first case of the definition 18), $myCustomFun(a.Length) - 1$ is invalid since it contains a call to a function that doesn't fall in any of the four classes of function calls considered in the definition 18. The entire loop node is therefore mapped to $\langle 0 \rangle$, which means that its contribution to the feature value is ignored.

```
let myKernel(a:float[], b:float[], c:float[]) =
   let endIdx = myCustomFun(a.Length)
    for i = 0 to endIdx do
      c.[i] <- a.[i] + b.[i]
```

Listing 9.4: Memory read count that depends on custom function

We choose to map unhandled nodes to 0 to enhance the robustness of the scheduling policy and, more generally, of the runtime. Whereas this choice can lead to significant errors in feature values and consequently to poor scheduling quality, it allows to support the execution of a wider set of computations and in particular of kernels for which precise feature extraction may fail. The runtime can nonetheless be configured to interrupt feature extraction if one or more kernel nodes cannot be mapped to finalizer nodes. In this case, the framework randomly selects a device for kernel execution.

Once a valid finalizer AST has been produced, we perform a final processing to further reduce the overhead of evaluating the finalizer at runtime, which

consists in folding constants and pruning the subtrees that do not contain any reference to the arguments. Each of these subtrees is evaluated and replaced by a node containing the evaluation result (i.e. a constant). The result of applying constant folding and pruning to the sample 9.3 is illustrated in listing 9.5.

```
fun (a:float[], b:float[], c:float[]) ->
    // 0+ removed, -1+1 folded, 0+1+1 pruned
    a.Length * 2
```

Listing 9.5: Memory read count finalizer after constant folding

9.2.2 Cache miss estimation

So far we illustrated the finalizer building process for *program-constructs-based* features, which means for features that estimate the occurrence of certain constructs at runtime. For the first class of features the finalizer code is an arithmetic expression with a dependency on the kernel actual arguments, like the one shown in listing 9.2. The evaluation of the finalizer is therefore extremely efficient. In addition, estimating the occurrence of constructs is a process that can be parametrized on the particular interesting AST nodes, which allows to describe the kernel-to-finalizer mapping in terms of a generic part (i.e. the *dmap* function) and a part specific for the construct, or set of constructs, which directly contribute to the feature value (i.e. the *smap* function).

The process can be adapted to extract also *resource-usage-based* features, such as the estimated number of cache misses or the bandwidth required by a kernel, which derive from considering the code structure together with the characteristics of the available devices (e.g. the cache size, the core clock, the memory bandwidth). In the this section we describe the process of extracting a feature that represents the estimation of the number of cache misses during kernel execution. This feature is used in the validation part of this Thesis (chapter 11).

To estimate the number of cache misses, we extract them memory access strides and the trip count of loops containing memory accesses. Since we exclusively consider cache misses generated by accessing vector arguments, throughout this section we use the terms “memory address/access” and “vector index/access” interchangeably.

Like in the previous section, we underline the fact that everytime we apply actions/operations on AST nodes, such as sum, multiplication, absolute value and ceiling, we do not intend to express their execution but the creation of AST nodes that represent such actions/operations. For example, when we

write $n + m$, where n and m are AST nodes, the operator $+$ must be intended as a shorthand representation of a node that encodes a call to the function "sum", whith n and m the node children.

In estimating the number of cache misses, we make the following assumptions.

- The threads (work-items) access each vector with an inter-work-item constant offset. For example, if the first work-item accesses the first element of a vector and the second work-item accesses the fourth element, the third work-item accesses the seventh (i.e. constant 3-elements offset). More formally, given m a memory access in the kernel, let $0 \leq t \leq gsize$ be the index of a work-item, $gsize$ the total number of work-items and $idx(m, t)$ the index of the vector accessed by the work-item with index t. The assumption can be expressed as follows:

$$\forall_{t \in 1..gsize-2} : idx(m, t) - idx(m, t-1) == idx(m, t+1) - idx(m, t)$$

- Work-items are processed in row-major order. In case of a 2D work-item space, the work-item with index (row, col) executes after the item $(row, col - 1)$ and the item $(row + 1, 0)$ after the item $(row, n_{cols} - 1)$, where n_{cols} is the width of the work-items space.

- If the address of a memory access is function of a loop iteration variable, the offset between addresses generated at consecutive iterations of the loop is constant. For example, if in the first iteration a vector is accessed at index 0 and in the second iteration at index 10, in the third iteration the index accessed is 20. We formalize this assumption considering a memory access m in the kernel that depends on an iteration variable v. We indicate with v_i the value associated to the variable in the i_{th} iteration of the loop and with $idx(m, v_i)$ the index accessed in the i_{th} iteration. If $itcount$ is the total number of iterations, the assumption can be expressed similarly to the assumption of inter-work-item constant offset.

$$\forall_{i \in 1..itcount-2} : idx(m, v_i) - idx(m, v_{i-1}) == idx(m, v_{i+1}) - idx(m, v_i)$$

- For each vector, we consider only the "deepest" access in the loops hierarchy of the kernel. In terms of the AST, we consider exclusively the access to the vector that has the highest number of *for* or *while* loops as ancestors. From an abstract point of view, this means that we consider the access that is likely to be occur more frequently. In case of multiple accesses to the same vector at the same level (i.e. contained in the same loop), we take into account only the first access. In the rest of this section we refer to this access as the *key access*.

- The trip count of a loop doesn't depend on any of the iteration variables of the outer loops. The example 9.2.2 illustrates the case of a nested loop that has a dependency on the iteration variable i of the parent loop. In particular, the initial value of the iteration variable j of the nested loop depends on i.

```
for i = starti to endi do
   for j = i + 1 to endj do
      ...
```

- Each vector has its own cache. As a consequence, if an access *acc* to a vector causes eviction, the evicted lines exclusively contains data belonging to the same vector accessed in *acc*. In other terms, each vector evicts its own cache lines.

- The CPU cores and work-items running on them shares the entire cache. This is an aproximation, since the specification of the CPU used reports that half cache is shared by groups of 2 cores (of the 4 available).

- We do not consider the associativity of the cache. In the platform used to evaluate the scheduling strategy (chapter 11), the CPU (L2) cache has a size of 2MB, with cache lines of 64 bytes. From the point of view of cache miss estimation, a cache address is logically divided into and offset, which is stored in the 6 least significant bits (64B line), and an index, which is stored in the bits 6-21 (15 bits to address 32k lines).

- No part of the vector is present in the cache before the first access and cache prefetching is disabled.

We estimate the number of cache misses generated at kernel runtime independently for each vector accessed, as we illustrate in the rest of this section.

For a given vector we consider the hierarchy of loops in the kernel, as shown in the sample 9.6, starting from the innermost loop that contains an access to the vector and bubbling up to the outermost.

```
let kernel (myArray:float32[], ws:WorkItemInfo) =
   ...
   for i = starti to endi do
      ...
      for j = startj to endj do
         ...
         for k = startk to endk do
            myArray.[ws.GlobalID(0)]
```

Listing 9.6: Hierarchy of loops

The hierarchy of loops in the kernel body are then wrapped in n additional loops, where n is the work-space size (from 1D up to 3D) and every call to the *GlobalID* OpenCL function[7] is replaced with a reference to the iteration variable of the corresponding loop, as shown in the example 9.7. With this manipulation, which agrees with the second assumption (order of execution of work-items), the execution of consecutive work-items is treated as a loop like all the other loops in the kernel. This allows to simplify and uniform the estimation of cache misses caused by the entire work-items space (i.e. the whole execution of the parallel program).

```
let kernel (myArray:float32[], ws:WorkItemInfo) =
   ...
   for workItemXIndx = 0 to ws.GlobalSize(0) - 1 do

      for i = starti to endi do
         ...
         for j = startj to endj do
            ...
            for k = startk to endk do
               myArray.[workItemXIndx]
```

Listing 9.7: Additional loops for the work-items space

[7]At kernel execution time, this function returns the identifier of the current work-item for a given dimension. In case of 3D work-item space, the first thread id is $0, 0, 0$ (x, y and z), the second $1, 0, 0$, and so on.

As already said, we define *key access* the deepest access to the vector. This is the only access considered to estimate the number of cache misses. In the sample 9.8, the key access is $myArray.[i, j, k]$, since it is the first access to the vector that occurs in the deepest loop.

```
let kernel (myArray:float32[], ws:WorkItemInfo) =
   for workItemXIndx = 0 to ws.GlobalSize(0) - 1 do

      for i = starti to endi do
         myArray.[i]

         for j = startj to endj do
            myArray.[i*j]

            for k = startk to endk do
               // Key access
               myArray.[i*j*k]
               ...
               myArray.[i*j*k+1]
```

Listing 9.8: Key access in an hierarchy of loops

In the rest of the section, we indicate with $ka(v)$ the expression (AST subtree) of the key access to a vector v. This expression can be extracted performing an initial kernel AST traversal.

To describe how loops are analysed and which information is bubbled up, we consider a kernel containing n nested loops, including the additional loops for the work-items space, and we indicate with $lv_1, lv_2, \ldots lv_n$ the iteration variables of the loops, from the outermost to the innermost. For each loop we build two expressions respectively encoding the following information. Once built, these expressions are passed to the parent loop.

- The estimated number of faults in executing the subtree rooted in the loop;
- Whether or not the loop can be entirely executed without causing eviction.

For simplicity, but without lack of generality, we assume that each loop iterates at least twice. This allows to present the process of building an estimator for the number of cache misses without considering surrounding conditions, enhancing readability and comprehension. These condition are nonetheless handled by the framework. In particular, if the trip count of a loop in the hierarchy is zero, then no access to the vector is performed and the estimated

number of cache misses is 0. In case of a loop iterating once, the number of cache misses estimated for the loop body if forwarded as it is to the parent.

We use the key access index expression to estimate the stride between accesses performed at consecutive iterations of the loop. In analyzing the i_{th} loop, with $1 \leq i \leq n$, let $init_i$ and $step_i$ be the initial expression associated to the iteration variable and the expression that encodes the update to the variable in each iteration. The expression $\langle init_i + step_i \rangle$ represents the value of the iteration variable in the second iteration. We obtain the expressions of the index generated in the first and second iterations by replacing each reference to the iteration variable lv_i respectively with $\langle init_i \rangle$ and $\langle init_i + step_i \rangle$. The references to the iteration variables of the others loops in the hierarchy are set to the respective initial values.

To formalize the construction of this expression for a given vector, we consider an expression n and we indicate with $def(n)$ the expression that results from processing n, replacing each and every reference to the iteration variables of the loops in the hierarchy with the corresponding initial expression. We also indicate with $n[e/v]$ an expression that results from replacing each reference to the variable v with the expression e. If T is the type of the elements contained in the vector, we indicate with $size(T)$ the size in bytes of each element.

The equation 9.5 represents the process of construction of an AST that represents the access stride in bytes between consecutive iterations of the i_th loop.

$$stride_i(v) = |def(ka(v))[init_i + step_i/lv_i] - def(ka(v))[init_i/lv_i]| * size(T) \tag{9.5}$$

In addition to the stride, we build the expression representing the trip count of the loop, formalized in the equation 9.6, where $init_i$, end_i and $step_i$ are respectively the initial/final value (expressions) of the iteration variable and the expression encoding the increment.

$$trip_i = \left\lfloor \frac{end_i - init_i + 1}{step_i} \right\rfloor \tag{9.6}$$

How the stride and the trip count expressions are employed depends on whether the loop is the innermost loop or an outer loop.

Innermost loop (n_{th} loop). For the innermost loop, we use the stride and the trip count to build an expression that represents the estimation of the number of cache misses in executing the loop. If we indicate with $lsize$ the size of a cache line, the expression $max(1, \lfloor lsize/stride_n(v) \rfloor)$

represents the number of iterations that fit a cache line. For example, if the cache line is 64B and the stride is 16B, the cache line accessed changes every 4 iterations. The *max* function is used to express that even when the stride is greater than the line size (i.e. the ratio is 0) at least one iteration fits a cache line (in particular, in this case a different cache line is accessed in each iteration).

We divide the trip count by the number of loops that fit a cache line to obtain the estimated number of misses in executing the loop, as defined in equation 9.7.

$$faults_n(v) = \left\lceil \frac{trip_n}{max\left(1, \left\lfloor \frac{lsize}{stride_n(v)} \right\rfloor\right)} \right\rceil \tag{9.7}$$

The stride and the trip count are also used to build an expression that encodes whether the execution of the loop causes the eviction of some lines. In particular, let $least_1(stride)$ be the position (starting from 0) of the least significant bit in the stride that is set to 1[8]. If $csize$ and $lsize$ are respectively the size of the cache and the of the cache line, then $csize/lsize$ is the number of lines in the cache and $l_{addr} = log_2(csize)$ and $l_{index} = log_2(csize/lsize)$ are respectively the length (size in bits) of a cache address and of its *index* part.

It is important to note that if the least significant 1 in the stride is the n_{th} bit, then all the addresses generated with this stride have the the same least significant $n-1$ bits, since the $n-1$ least significant bits of the stride are 0. This means that if the least significant 1 falls in the *index* part of a cache address, some bits of the index are always the same for each and every address generated, as shown in figure 9.3. Consequently, the number of lines eligible to store data is lower than the number of lines in the cache.

For example, if the stride is 512 the least significant 1 is stored in the bit 8. In case of a 2MB cache with lines of 64 bytes, the index is stored in the bits 6-20. Since the least significant 1 is in position 8, two bits of the index (bit 6 and 7) have the same value for all the addresses generated and only the 13 most significant bits contribute to determine the cache line. Consequently, only $2^{13} = 8K$ lines are eligible to store data, out of the $2^{15} = 32K$ lines in the cache.

[8]Again, since ka is an expression, we indicate with $least_1()$ the expression that contains a call to a function to obtain the least significant 1

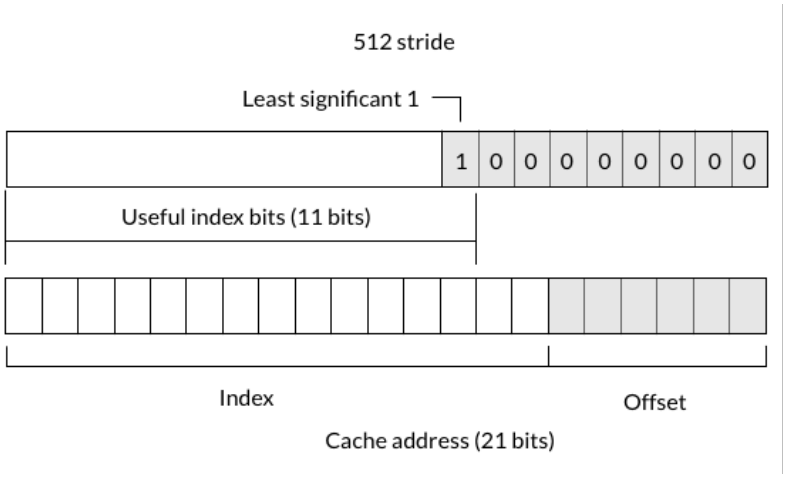

Figure 9.3: Impact of memory access stride on the index part of a cache address

From the figure we deduce that the number of bits of the *index* part of a cache address that contribute to determine the cache line is $l_{addr} - least_1(stride)$. Obviously, regardless the position of the least significant 1 bit, no more than l_{index} bits are required to address a cache line. This means that the actual number of useful bits to address a cache line is $min(l_{addr} - least_1(stride), l_{index})$. From this expression we obtain the number of lines available to store the vector data, which is $2^{min(l_{addr} - least_1(stride), l_{index})}$.

If the number of useful lines is greater than the loop trip count, the cache can store all the data accessed during the execution of the loop. In other terms, the loop can be entirely executed without causing eviction. Otherwise, at a certain iteration the memory access will cause the replacement of a line containing previously-loaded data (equation 9.8).

$$
\begin{aligned}
evict_n(v) = & \quad 2^{min(l_{addr} - least_1(stride), l_{index})} < trip_n \quad (9.8)\\
= & \quad 2^{min(log_2(csize) - least_1(stride), log_2(csize/lsize))} < trip_n\\
= & \quad min(2^{log_2(csize) - least_1(stride)}, 2^{log_2(csize/lsize)}) < trip_n\\
= & \quad min\left(\frac{csize}{2^{least_1(stride)}}, \frac{csize}{lsize}\right) < trip_n\\
= & \quad \frac{csize}{max(2^{least_1(stride)}, lsize)} < trip_n
\end{aligned}
$$

Outer loop (i_{th} loop with $i < n$). In case of an outer loop we consider the child (i.e. outermost nested) loop. If the child loop reports cache eviction (i.e. $evict_{i+1}(v)$ is true), we estimate a number of misses for the outer loop equal to the number of misses reported by the child multiplied by the trip count $trip_i(v)$ of the loop. In other terms, we estimate that in each iteration of the outer loop the inner loop causes the same number of faults.

If the child loop reports no eviction, we consider the stride between consecutive accesses happening at consecutive iterations of the outer loop (equation 9.5, using the index of the outer loop). If the stride in bytes is smaller than the cache line size, it means that consecutive iterations of the outer loop reuse data loaded in the previous iterations. Therefore, we estimate the number of faults for the outer loop as the product of the number of faults reported by child by the number of times an iteration of the outer loop cannot reuse data loaded in the previous iterations. Given the stride in bytes $stride_i(v)$ and the line size $lsize$, $\lfloor lsize/stride_i(v) \rfloor$ represents the number of iterations that can reuse previously-loaded data. Given the trip count $trip_i$, the number of times an iteration cannot reuse data loaded in the previous iterations can be expressed as $\lceil trip/\lfloor lsize/strd \rfloor \rceil$. We multiply this expression by the number of faults in the child loop to express the number of faults of in the parent loop.

In the equation 9.9 we formalize how the informations coming from the analysis of the nested loop are combined to express the number of faults for the outer loop. The *if-otherwise* in the equation represents an *if then else* AST node, built and bubbled up along the loop hierarchy.

$$faults_i(v) = \begin{cases} trip_i * faults_{i+i}(v) & \text{if } evicts_{i+i}(v) \\ \left\lceil \frac{trip_i}{\left\lfloor \frac{lsize}{stride_i(v)} \right\rfloor} \right\rceil * faults_{i+i}(v) & \text{otherwise} \end{cases} \tag{9.9}$$

For what regards cache eviction, if the executing the child loop causes eviction we assume that executing the outer loop causes eviction as well. If the child loop reports no eviction, then we compare the outer loop trip count with $\lfloor lsize/stride_i(v) \rfloor$, which is the number of iterations of the outer loop that can reuse previously-loaded data. If the first is smaller

than the second, than the outer loop can execute entirely without causing eviction (i.e. the data loaded by all the outer loop iterations fit the cache). In this case, the parent loop reports no eviction (equation 9.10).

$$evict_i(v) = evicts_{i+i}(v) \;||\; trip_i(v) > \left\lfloor \frac{lsize}{stride_i(v)} \right\rfloor \tag{9.10}$$

The equations 9.5 to 9.10 globally define how the kernel AST nodes representing loops are mapped to finalizer nodes. All the nodes that do no represent loops are only considered to check if any of the children contains loops. In other terms, they do no map to finalizer nodes. Note that, according to 9.9, if $faults_0(v)$ (0 stands for the index, in the hierarchy of loops, of the outermost loop) is an expression that holds the estimated number of misses for the entire kernel. Given this, similarly to the definition 17, we can express the process of building the cache miss finalizer as stated in the definition 19.

Definition 19. [Cache miss finalizer building process] Given a kernel k, let v be a vector parameter, $ka(v)$ be the key access to v and 0 be the index of the outermost loop in the hierarchy of loops containing $ka(v)$.

The process of building a finalizer that estimates the number of cache misses in accessing v during the execution of k is described by a function *cachebuild* defined as:

$$cachebuild(k) = fun\ \langle pars(k) \rangle \rightarrow \langle faults_0(v) \rangle$$

Variable unfolding, AST validation and pruning introduced in section 9.2.1 for the extraction of a generic feature are applied with no changes in the construction of the estimated cache misses finalizer, in order to guarantee that the tree encoding the number of cache misses for the entire kernel execution exclusively depend on the set of kernel arguments contain only valid nodes.

Example of cache miss estimation

We conclude this section illustrating ad example of cache miss estimation. In particular, we consider a kernel that sums the elements along each column of an NxN matrix (listing 9.9). To execute the kernel, N work-items are launched, each of which operates on the column corresponding to its own index.

```
let kernel (input:float[,], output:float[], ws:WorkItemInfo) =
   let threadIdx = ws.GlobalID(0)
   let mutable accum = 0.0f
   for r = 0 to input.GetLength(0) - 1 do
      accum <- accum + mat.[r, threadIdx]
   output.[threadIdx] <- accum
```

Listing 9.9: Row sum kernel

In listing 9.10 we show the hierarchy of loops, including the additional loop to iterate on the work-items space.

```
for threadIdx = 0 to ws.GlobalSize(0) - 1 do
   for r = 0 to input.GetLength(0) - 1 do
      mat.[r, threadIdx]
```

Listing 9.10: Key access and loops in the row sum kernel

To compute the estimated number of cache misses, we consider a 512x512 elements matrix where each element is a 4 byte floating point or integer value. The cache size is 2MB with cache lines of 64 bytes. Consequently, the length of a cache address is 21 bits, with 6 bits of offset and 15 bits of index.

When the first work-item starts, in each iteration a slice (first 16 elements) of a matrix row is loaded into the cache, as illustrated in figure 9.4.

Note that the index of the key access is $[r, threadIdx]$, which corresponds to a flat 1D index[9] $(r * input.GetLength(1)) + threadIdx$.

Considering the innermost loop and the key access index, we apply the equation 9.5 to obtain the access stride between consecutive iterations, which is $(((r+1)*input.GetLength(1))-(r*input.GetLength(1)))*4 = (1K-512)*4 = 2K$ bytes.

The equation 9.6 is applied to determine the trip count, which is 512.

By applying the equation 9.7 we obtain the number of misses generated during the execution of the loop, which is $512/max(1, 64/2K) = 512/1 = 512$. This means that the work-item generates a cache miss in each iteration of the loop.

Finally, we use the equation 9.8 to verify whether the loop causes eviction. Since the least significant 1 bit of the $2K$ stride is in position 11, the equation returns $2M/max(2^{11}, 64) = 2M/2K = 1K$. Since $1K$ is greater than the trip count (512), the loop doesn't cause eviction. This means that when the work-item ends, the set of data loaded during the overall execution of the loop is

[9]Even though the matrix is 2D, the layout in memory is flat row-major.

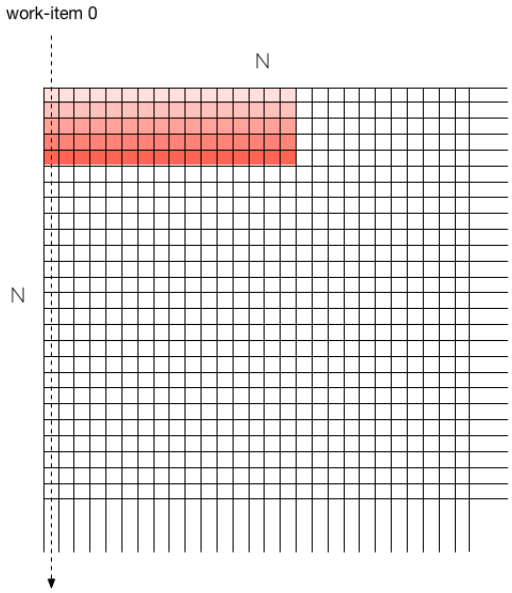

Figure 9.4: Data loaded into cache when scanning the first column of a matrix

stored in cache.

Moving to the outer loop, we obtain the stride by setting r to 0 in the expression of the key access index $(r * input.GetLength(1)) + threadIdx$ and evaluating the result for $threadIdx = 0$ and for $threadIdx = 1$. The result is $(threadIdx + 1 - threadIdx) * 4 = 4$ bytes. In fact, at each iteration of the innermost loop, consecutive work-items access consecutive elements of a row. The trip count of the loop is 512, which is equal to the number of columns and to the number of spawn work-items.

Since the inner loop doesn't cause eviction, we apply the second case of the equation 9.9, resulting in an estimated number of misses for the outer loop equal to $(512/(64/4)) * 512 = 4K$ cache misses, which represents the total number of cache misses since the loop is the outermost in the hierarchy. Note that the term $64/4 = 16$ suggests that cache eviction takes place once every 16 work-items. This situation is illustrated in figure 9.5. As shown, since 16 elements (64 bytes) of each row are loaded into the cache by the execution of the first work-item, successive work-items find those lines already in cache.

To conclude, we consider what happens if the size of the input matrix is 1024x1024. In this case the stride of the innermost loops is $4K$, the trip count is 1024 and the number of cache misses is 1024 as well.

Since the least significant 1 bit of the $4K$ stride is in position 12, when applying the equation 9.8 we obtain $2M/max(2^{12}, 64) = 2M/4K = 512$. Since

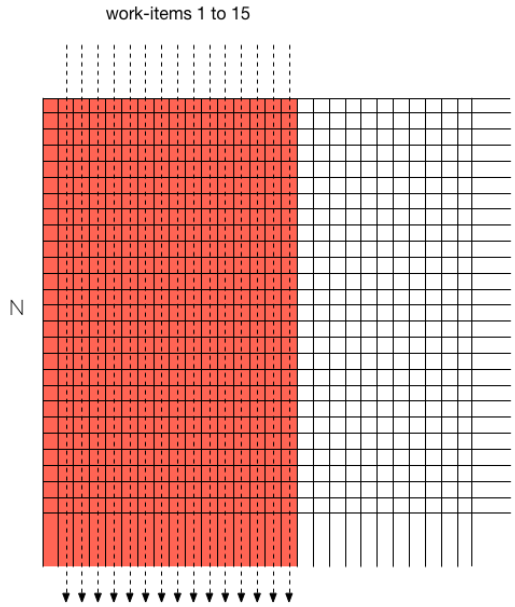

Figure 9.5: Cache lines reused when scanning the columns 1-15 of a matrix

this value is smaller than the trip count (1024), the loop causes eviction. In particular, only 512 consecutive iterations can run without causing eviction. This situation is illustrated in figure 9.6. As shown, successive work-items do not find the data in cache, which entails that the same number of cache misses is generated for each work-item executed.

The total number of cache misses for the kernel is calculated applying the first case of the equation 9.9, resulting in $1024 * 1024 = 1M$ misses.

9.3 Prediction model, profiling and regression

Over the years, linear modeling has been largely employed to describe or estimate the dependency of certain phenomena from a set of known variables in most of the scientific research fields [9, 23, 38, 59, 63]. The pourpose of linear regression is to model the relationship between a *dependent variable*, also called *measured variable* or *regressand*, and a set of *independent variables*, also known as *explanatory variables* or *regressors*.

In the completion-time-prediction model described in this chapter, the dependent variable is the completion time on a device, while the explanatory variables are the set of chosen features. The output of linear regression is a *parameter vector* (also known as *set of regression coefficients*) which correlates

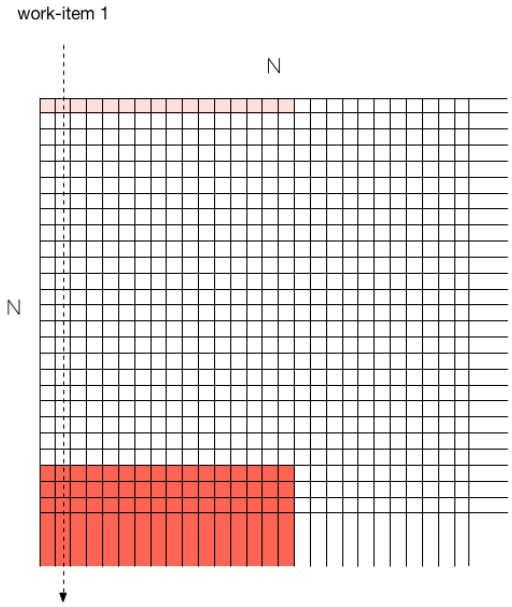

Figure 9.6: Cache eviction when scanning the first column of a matrix

the features to the completion time. Conceptually, the parameter vector determines the impact of each feature on the time required for a computation to complete.

Linear regression is a powerful modeling tool which assumes a linear relationship between the dependent variable and the explanatory variables. This does not imply that only linear behaviors can be explained: as long as the dependent variables expose the same kind of non-linearity that characterizes the dependent variable, linear regression represents a suitable method. As already briefly discussed, for many algorithms the completion time is non-linear with respect to the size of the input, but can be linear with respect to certain features. For example, matrix multiplication completion time is non-linear with respect to the size of the input but is linear with respect to the number of arithmetic operations. Using this number as an explanatory variable, linear regression is able to build a model capable of expressing completion time as the dependent variable.

Linear regression is defined by formula 9.11

$$\mathbf{y} = \mathbf{X}\beta + \epsilon \tag{9.11}$$

Where:

- $\mathbf{y}$ is the dependent variable
- $\mathbf{X}$ is the matrix of the explanatory variables
- β is the vector that contains the regression coefficients
- ϵ is the error term (fitting residuals)

Ordinary linear regression is sensitive to outliers in the dependent variable [6]. Completion time is likely to contain outliers, mainly caused by certain sporadic effects that are not considered in the model or to the instability of the system where the measurement is performed. For example, many algorithms show outliers for few specific combinations of cache and input size that cause particularly aggressive cache eviction. Outliers can be introduced also by heavy task in the system that steals computing resources from the running tests or by driver instability. In ordinary linear regression, outliers can have a severe, negative impact on the quality of the model built. For this reason, our prediction model employs a robust regression method instead of ordinary regression [32] [25].

As described in the previous section, FSCL can efficiently extract and evaluate kernel features, leveraging a mixed static and runtime approach. Despite the generality of the feature-extraction process, not every feature can be used in the prediction model. The first restriction for a feature to be employed in the prediction model is to "count" the occurrences of a certain aspect, measuring or predicting the usage of a particular computing resource. In addition, valid features are characterized by non negativity, null empty set (i.e. an empty program must have all features equal to zero) and countable additivity (i.e. if a feature has values x_1 and x_2 for two programs p_1 and p_2, then if p_3 represents serial execution of p_1 and p_2, the feature value for p_3 must be equal to $x_1 + x_2$).

An example of valid feature is the *number of instructions* (or memory accesses), because it *counts* an aspect of the program, it can't hold negative values, an empty program has no instructions and sequentially running two programs causes the execution of a number of instructions equal to the sum of the respective instructions count.

9.3.1 Completion-time prediction model

To describe our completion-time prediction model, we firstly need to introduce some definitions of the entities involved in building the model.

Definition 20. [Program] A program is a particular FSCL kernel

Definition 21. [Program case (or instance)] A program case is a pair made of a program and a set of actual arguments for its parameters. A case represents a particular execution of a specific program, detemined by the arguments supplied.

Definition 22. [Benchmark case (or instance)] A benchmark case is a program case associated with its completion time on a device

Definition 23. [Benchmark (or Training Sample)] A benchmark is a program where each case is benchmark case. A benchmark represents a program for which we assess the completion time for all its cases.

Definition 24. [Target case (or instance)] A program case for which we want to predict the completion time

Benchmarks are used to build a linear model for each available OpenCL device in the running system. Each model correlates a set of features to the completion time on a specific device. For each device a linear model is separately built starting from the following data:

- A set of benchmarks: for each benchmark we consider several cases by varying the input size. Each benchmark case is executed on the device to obtain the corresponding completion time
- A set of features: each feature captures a certain aspect of a program. The chosen features are extracted from each benchmark case.
- A matrix $\mathbf{A}$ where each column is a feature and each row is a benchmark case.
- A vector $\mathbf{t}$ with the completion times of the benchmark cases on the considered device, in the same order of the cases in $\mathbf{A}$

After having defined a set of branchmarks and features, a matrix $\mathbf{A}$ and a vector $\mathbf{t}$, we build the matrix of the explanatory variables X (definition 9.11) from $\mathbf{A}$.

Linear regression assumes homoscedasticity (i.e. constant variance) in the error terms. In particular, the error on a feature should not be correlated to the completion time. In our model we instead expect the errors on features to be affected by such a correlation. To deal with this issue, we employ the Weighted Least Squares method, which is a generalization of ordinary least squares that relaxes this assumption by normalizing the equations using the variance of the completion time. For our model, we set the dependent variable

$\mathbf{y}$ as the componentwise normalization of $\mathbf{t}$ by its standard deviation (i.e. $y[i] = t[i]/\sigma(t[i])$, so that all of the samples can be assumed to be identically distributed. Consistently, we also normalize each row of $\mathbf{A}$ in the same way.

Finally, we add to the resulting matrix a unary column, normalized like $\mathbf{t}$, which constitutes the intercept of the linear regression. Conceptually, the intercept represents the time needed to start any computation, independently from the specific benchmark case being measured.

$$\mathbf{y} = \begin{pmatrix} t_1/\sigma(t_1) \\ t_2/\sigma(t_2) \\ \vdots \\ t_m/\sigma(t_m) \end{pmatrix} \tag{9.12}$$

$$\mathbf{X} = \begin{pmatrix} 1/\sigma(t_1) & a_{1,1}/\sigma(t_1) & a_{1,2}/\sigma(t_1) & \cdots & a_{1,n}/\sigma(t_1) \\ 1/\sigma(t_2) & a_{2,1}/\sigma(t_2) & a_{2,2}/\sigma(t_2) & \cdots & a_{2,n}/\sigma(t_2) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1/\sigma(t_m) & a_{m,1}/\sigma(t_m) & a_{m,2}/\sigma(t_m) & \cdots & a_{m,n}/\sigma(t_m) \end{pmatrix} \tag{9.13}$$

We can now apply $\mathbf{y}$ (as defined in the equation 9.12) and $\mathbf{X}$ (as defined in the equation 9.13) to the equation 9.11, and solve it using linear regression. The results of linear regression are:

- β: the set of regression coefficients, which describe the completion time on the considered device in terms of a linear combination of the features.
- ϵ: the fitting residuals. Analyzing the residuals allows to assess the quality of the fitting. Residulas of a good fit not only should be small but also should show no correlation to the features and to the completion time.

Completion time is an unreliable measure, subject to unpredictable errors due to the presence of running processes and tasks in the system during the execution of the measured program. Since we expect the presence of outliers in the data, as already said we employ a robust regression method, which uses the Iteratively Reweighted Least Squares algorithm [34] to identify and discard the outliers.

Once β has been calculated, we can use formula 9.14 to predict the completion time of a target program (case) on the device considered. In this formula, $\mathbf{x}$ is a vector that contains the same features used in $\mathbf{X}$ but calculated on a target program.

$$t = \mathbf{x}\beta \tag{9.14}$$

Robust linear regression is performed independently for each OpenCL device available, building $\mathbf{y}$ in equation 9.12 starting from the completion times of the benchmark cases on the device.

9.4 The FSCL runtime scheduling engine

The scheduling engine couples the feature extraction approach and the prediction model to select the best-device for each kernel submitted to the FSCL runtime for execution.

At deploy time (i.e. when the framework is installed on a new machine) the engine profiles the system using a set of training samples. For each case of a training sample, the engine extracts the relevant features and executes the case on each available device to get the corresponding completion time.

With the set of features and the completion times computed for each case of each training sample, the engine builds a matrix on which rebust linear regression is applied.

The features extracted from the training samples, the vector of completion times and the regression coefficients for each device, are stored in a file for futher usage. Everytime an FSCL program is loaded for execution, the scheduling engine transparently loads those information from the file into an in-memory data-structure.

The first time a particular kernel is seen, the engine transparently builds the finalizers for the relevant features and stores them in the kernel cache (section 8.2.1).

At kernel execution time, the call arguments are used to evaluate the finalizers, which are combined with the model (i.e. regression coefficients) of each device to determine the corresponding completion time.

Finally, the engine selects the device with the lowest estimated completion time.

The engine can adapt to dynamic changes to the device set and to the set of features in the running platform. If a profiled device is not available at kernel execution time (e.g. the device has been unplugged or it is malfunctioning), the successive device in (ascending) order of completion time is chosen. If a new device is discovered, the runtime profiles it and enriches the information stored in the data file with the completion times of the training samples on the new device. Whenever a new feature is added[10], the engine computes it for each case of each training sample and finally re-runs linear regression. In

[10]By customizing/extending the scheduling policy with the introduction of a new feature

case of dropping a feature, linear regression is re-run on the matrix that results from dropping the column corresponding to that feature.

9.5 Conclusions

In this chapter we discussed our approach to schedule FSCL kernels in a device-aware fashion on multi-device heterogeneous systems. The approach is based on three main points:

1. A process to extract features from kernels;
2. A set of training samples;
3. A robust linear regression algorithm that, for each device, builds a model that correlates the features to the completion time on the device.

We leverage the F# quotations mechanism to perform code analysis for feature extraction. Thanks to the F# API to create and evaluate expressions, we define a process for efficient feature extraction that is partially applied at kernel-compile-time and partially at kernel-execution-time. For a given kernel, features are precomputed when the kernel is compiled (i.e. the first time the kernel is seen), delaying to kernel-execution-time only a small amount of processing.

System profiling and model construction via linear regression are performed transparently at framework deploy-time and, possibly, whenever the device set or the feature set change.

Currently, the scheduling policy is locally greedy. The best device is chosen for each kernel to execute in a contextless fashion, ignoring how kernels interact with each other in the flow graph. This means, for example, that the engine doesn't take into account the time spent in transferring data across devices when two successive kernels are respectively scheduled on different devices. We discuss an extension the current scheduling strategy in order to consider data-transfer overhead in section 14.1.

Part III

Validation

Introduction

In part II we presented the FSCL framework, which synthesizes the research in raising abstraction over OpenCL programming, scheduling and execution.

In this part we validate our results against the targets discussed in part I.

Separated validation is discussed for each of the goals of our research. In chapter 10 we validate the FSCL programming model. In particular, we take into account a set of real-world algorithms and we show how they can be expressed with the FSCL kernel language. We validate the combination of abstraction and flexibility provided by the multiple levels of abstraction, showing that programmers can code at the lower-level only the parts of an application that do not fit the available collection functions, while keeping the rest of the program at the highest level of abstraction.

In chapter 11 we validate the model proposed to predict the completion time on each available device on the executing platform. We present and discuss the prediction errors in both estimating the completion time for a specific device and in selecting the best-device for a given computation. We also assess the average speedup obtained by applying the scheduling policy, considering the overhead of feature extraction and device selection.

Finally, in chapter 12 we focus on the efficiency of the abstraction layer. We validate the efficiency of the scheduling engine, especially considering the overhead introduced by features evaluation, and the efficiency of the whole runtime. We compare completion times of FSCL computations with equivalent programs written in OpenCL C99 and in other frameworks. We show that the overhead introduced by the runtime layer on the low-level OpenCL execution is mostly irrelevant.

Chapter 10

Validation of the programming model

In this chapter we validate the FSCL programming model against a set of popular algorithms, taking into account both abstraction and flexibility. We show that the language can be employed succesfully to express a wide variety of computations, possibly coding different parts of a computation at different levels of abstraction.

For each algorithm, we show the FSCL code and we underline its relevant aspects. We also discuss the way the same algorithm would be expressed using different high-level programming models and languages (section 2.2). In particular, we consider the equivalent implementations in Aparapi [5], which represents today's framework most similar to FSCL, and in Dandelion [58], which is one of the most recent and relevant works on abstract parallel programming models based on pre-existing language features to represent patterns/skeletons.

10.1 Black-Scholes

BlackScholes is a popular algorithm that implements a formula derived from the homonymous model developed for theoretical estimation of the price of European-style options. The Black-Scholes model provides a partial differential equation for the evolution of an option price under certain assumptions [71]. For European options, the closed-form solution for this equation is the following.

Given:

- S, the current option price
- X, the strike price

- T, the time to expiration
- R, the continuously compounded risk free interest rate
- V, the implied volatility for the underlying stock
- V_{call}, the price for an option call
- V_{put}, the price for an option put
- cnd, the cumulative normal distribution function

The price of option call/put is estimated as follows:

$$V_{call} = S * cnd(d1)X * e^{RT} * cnd(d2)$$

$$V_{put} = X * e^{RT} * cnd(-d2) - S * cnd(-d1)$$

$$d1 = \frac{log(\frac{S}{X}) + (r + \frac{v^2}{2})}{v}\sqrt{T}$$

$$d2 = \frac{log(\frac{S}{X}) + (r - \frac{v^2}{2})}{v}\sqrt{T}$$

The parallel implementation is shown in listing 10.1. We define an utility function *cnd* to calculate the cumulative normal distribution and a function *blackScholes* to calculate the sample of call and put price from a given sample of stock price, strike price, time to expiration, volatility and interest rate. At line 30 we populate an array of samples to pass to the computation. Since the computation is performed independently on each input sample, we use the *Array.map* function to express the kernel. Each work-item of the resulting OpenCL kernel applies *blackScholes* to a different input sample.

```
1  [<ReflectedDefinition>]
2  let cnd(d:float) =
3      let A1 = 0.31938153
4      let A2 = -0.356563782
5      let A3 = 1.781477937
6      let A4 = -1.821255978
7      let A5 = 1.330274429
8      let RSQRT2PI = 0.39894228040143267793994605993438
9      let K = 1.0 / (1.0 + 0.2316419 * (fabs(d)))
10
11     let nd = RSQRT2PI * Math.Exp(-0.5 * d * d) * (K * (A1 + K * (A2 + K * (A3 + K * (A4 + K * A5))
         )))
12     if d > 0.0 then
13         1.0 - nd;
14     else
15       nd
16
17 [<ReflectedDefinition>]
18 let blackScholes R V (S,X,T) =
19     let sqrtT = Math.Sqrt(T)
20     let d1 = (Math.Log(S / X) + (R + 0.5 * V * V) * T) / (V * sqrtT)
21     let d2 = d1 - V * sqrtT
22     let cndD1 = cnd(d1)
23     let cndD2 = cnd(d2)
24     let expRT = Math.Exp(-R * T)
25     (S * cndD1 - X * expRT * cndD2,
26      X * expRT * (1.0 - cndD2) - S * (1.0 - cndD1))
27
28 // Setup data
29 let rnd = new Random()
30 let hSXT = Array.create 1024 (
31             rnd.NextDouble() * (25.0) + 5.0,
32             rnd.NextDouble() * (99.0) + 1.0,
33             rnd.NextDouble() * (9.75) + 0.25)
34 let R, V = 0.02, 0.30
35
36 // Run Blackscholes
37 let result =
38  <@
39    hSXT |>
40     Array.map(blackScholes R V)
41  @>.Run()
```

Listing 10.1: Blackscholes algorithm in FSCL

The FSCL code for parallel Black-Scholes is entirely expressed using the highest level of abstraction offered by the FSCL programming model. In particular, since we exclusively use collection functions, the underlying OpenCL layer is not visible. In addition, the code perfectly matches a possible regular, sequential implementation of the Black-Scholes algorithm in F#. The FSCL framework "parallelizes" the execution by interpreting the *Array.map* function as a kernel, generating the corresponding OpenCL kernel and running the kernel on one of the available OpenCL devices in the system.

In Aparapi, Black-Scholes can be similarly implemented, but it requires the explicit definition of a kernel to perform a map on the input array. The implementation in Dandelion can instead leverage the *Select* LINQ function to act as a map on the input collection. The programmers have only to take

care of using the appropriate custom data-types provided, instead of working with regular arrays.

10.2 K-Means

K-Means is a popular algorithm for vector quantization, widely used for cluster analysis in data mining. K-Means partitions an input set of observations into a set of clusters such that each observation belongs to the cluster with the nearest mean, resulting in a subdivision of the observations space into Voronoi cells.

In listing 10.2 we perform geometric K-Means, where an observation is a point in the cartesian space. The algorithm starts from a set of k initial centers randomly chosen from the set of input points. In our example, $k = 3$ and the initial centers are the first k input points (line 16). At line 24 the points are grouped into clusters on the basis of each one's nearest center. We use the F# *Array.groupBy* collection function, which is executed as an OpenCL kernel leveraging the *nearestCenter* utility function to compute the nearest center for a each given point. The output of *Array.groupBy* is an array of pairs *(key, items)*, where *items* is in its turn an array containing the points for which *key* is the selected center. At line 25 we process each group to calculate the new center as the geometric mean of the points in the group. We use a higher-order collection function, *Array.map*, to process the set of groups. For each group, we execute a reduction (line 27) to sum the points component-wise (i.e. x and y values) and we divide the resulting values by the size of the group. Since the process applied to each group is a computing expression, where *Array.reduce* is executed as a kernel and the dividing function as a sequential function, the high-order *Array.map* is considered a collection composition. The runtime therefore spawns a set of threads to operate (i.e. calculate the new geometric mean) concurrently on each group.

```
1  [<ReflectedDefinition>]
2  let nearestCenter (centers: (float * float)[]) (point: float * float) =
3      let mutable minIndex = 0
4      let mutable minValue = Double.MaxValue
5      let x,y = point
6      for curIndex = 0 to centers.Length - 1 do
7          let cx,cy = centers.[curIndex]
8          let curValue = Math.Sqrt(Math.Pow(x - cx, 2.0) + Math.Pow(y - cy, 2.0))
9          if curIndex = 0 || minValue > curValue then
10             minValue <- curValue
11             minIndex <- curIndex
12     minIndex
13
14 // Prepare data
15 let rnd = new Random()
16 let k = 3
17 let points = Array.init 1024 (fun i -> (rnd.NextDouble() * 5.0, rnd.NextDouble() * 3.0))
18 let centers = Array.init k (fun i -> points.[i])
19
20 // Run kmeans
21 let kmeans =
22  <@
23     points |>
24     Array.groupBy(fun i -> nearestCenter centers i) |>
25     Array.map (fun (key, data) ->
26                 data |>
27                 Array.reduce(fun (cx,cy) (x,y) -> (cx + x, cy + y)) |>
28                 (fun (a,b) -> (a/(double)(Seq.length data), b/(double)(Seq.length data))))
29  @>.Run()
```

Listing 10.2: KMeans algorithm in FSCL

Like Black-Scholes, the K-Means algorithm can be entirely coded at the highest level of abstraction offered by the FSCL kernel language and the underlying OpenCL layer is not visible to the programmers. This is particularly due to the ability to use collection functions of arbitrary order to coordinate sub-expressions.

Since Aparapi doesn't expose any skeleton to work on collections, the K-Means implementation requires an explicit kernel for *Array.groupBy* and *Array.reduce*. Grouping values on GPUs is particularly tricky, since OpenCL/CUDA do not support multi-dimensional arrays. *Efficient* grouping on GPU is even more complicated and generally requires to define/use a high performance sorting algorithm optimised for this type of devices [60]. In addition to explicitly coding each collection function as a kernel, Aparapi programmers must independently execute each kernel and coordinate them, since no form of high-level composition and coordination of kernels is provided.

K-Means is the example considered to illustrate the programming model of Dandelion in the presentation paper [58]. With Dandelion, this algorithm can be easily expressed as a composition of LINQ operators. It is not clear which restrictions are imposed on the data-types processed, since the paper doesn't mention them and the code of the project is not available at the time of writing. A natural representation of 2D points is a tuple of two float or

integer values. Since FSCL is able to properly map tuples to OpenCL valid data-types, programmers can freely use them to define and execute computing expressions. Dandelion may instead require different solutions to work with non-primitive types (e.g. structs).

It is important to note that the listing 10.2 reports a single pass of the K-Means algorithms, which is usually iterated multiple times to converge to a stable set of centroids. Programmers can iterate the K-Means pass without escaping the quotation context, as illustrated in listing 10.3. Iterative execution on progressively refined sets of centroids can be performed using a third-order collection function, which is *Array.fold* (line 5). The input array at line 4 is used to trigger the execution of the fold function 100 times. Each iteration executes in a state bound to the *currCenters* variable, which contains the current set of centroids. The sub-expression executed at each iteration corresponds to the single pass considered in listing 10.2. The result of this sub-expression (new set of centroids) encodes the state for the successive iteration. With the implementation provided, the K-Means pass can be viewed as an user-defined state-transition function. Given a state (centroids) as a set of points, the sub-expression computes the next state (new set of centroids).

```
1  // Run kmeans iteratively
2  let kmeans =
3   <@
4      [| 1 .. 100 |] |>
5      Array.fold(fun currCenters _ ->
6              points |>
7              Array.groupBy(fun i -> nearestCenter currCenters i) |>
8              Array.map (fun (key, data) ->
9                          data |>
10                         Array.reduce(fun (cx,cy) (x,y) -> (cx + x, cy + y)) |>
11                         (fun (a,b) -> (a/(double)(Seq.length data), b/(double)(Seq.length data))))
                             ) centers
12  @>.Run()
```

Listing 10.3: Iterations in K-Means

Iterative K-Means can be expressed in Dandelion using an high-order LINQ aggregate operator. To the contrary, Aparapi would demand an additional effort in host-side coding to coordinate the iterative execution of the K-Means step.

Host-side execution contexts, like F# quoted computing expressions in FSCL, chains of LINQ operators in Dandelion and Java kernels in Arparapi, do not only represent the boundaries of the abstraction provided to compose computations but also the boundaries of manipulations, optimisations and, more generaly, the scope of the computations interpreter (runtime). In other terms, the execution support of the constructs provided is generally limited to a single context. Therefore, when a context is split into multiple contexts,

not only the host-side abstraction decreases but also the performance is negatively affected. In case of FSCL, multiple quotations/computing expression are needed whenever function/collection composition is not flexible enough to express the particular orchestration of kernels required. Since each quotation implies a "setup" time to obtain the AST object, to parse the quotation and to analyze its content, splitting a quotation may incur a significant overhead. Differently from FSCL, Aparapi introduces a different context for each kernel, since no concept of composition of kernels is provided. Given that iterative execution of kernels is a very popular scheme for problems that cannot be solved in a single pass, Aparapi introduces some features to maintain a state across successive execution of kernels in order to limit the impact on performance. A chance to maintain a state across kernels is given by explicit buffer handling, which overcomes the impact of transferring back (i.e. device to host) data that are not accessed by the host but are instead the input of successive kernel instances. Despite the possibility to improve performance thanks to the control over data allocation and transfer, when enabling explicit buffer handling, the developers have to manually read and write buffers, which reduces the abstraction and considerably limits the programmers' productivity. Another possibility to iterate a kernel efficiently in Aparapi is to wrap it into the *iterate* function, which nonetheless doesn't provide any way for a kernel to update the state. In both the cases, efficient iterative or multi-kernel execution is possible only leveraging special constructs that do not integrate seamlessly with the rest of the API. To the contrary, FSCL allows to embed host-side coordination of kernels into the quotation context, enabling optimisations and efficient execution while leveraging an homogeneous set of contructs.

10.3 Tiled matrix multiplication

So far we considered examples of FSCL programs based exclusively on collection functions to express kernels and to compose them. In this section we show an advanced algorithm for matrix multiplication based on tiling. The algorithm is particularly efficient on GPUs since it leverages device local memory (section 2.1) to reduce the number of accesses to global buffers.

Tiled matrix multiplication requires the definition of a *MatMul* custom kernel, illustrated in listing 10.4. The work-item domain is two-dimensional. Given two input matrices $matA$ and $matB$, each 2D work-group of size $w * h$ multiplies a stride of h rows of $matA$ by a stride of w columns of $matB$. In the specific example, the size of each group is set to $16 * 16$ (line 38)[1].

[1]This implies that input matrices must be of a size multiple of 16

Each group horizontally scans $matA$ at blocks of $16 * 16$ elements (line 18). The matrix $matB$ is instead scanned vertically using the same block size (line 29). At each step, a block of $matA$ and one of $matB$ are loaded from global to local memory (lines 22 and 23). Each thread than performs the dot product of a row of the first block by a column of the second, updating the cross-blocks accumulator $Csub$ (line 26). When all the blocks of $matA$ and $matB$ have been scanned, each thread in a group can write (the final value of) the element of the output matrix $matC$ to the global buffer (line 31).

```
1  [<ReflectedDefinition; Kernel>]
2  let MatMul(matA: float32[,], matB: float32[,], matC: float32[,], wi: WorkItemInfo) =
3      let bx, by = wi.GroupID(0), wi.GroupID(1)
4      let tx, ty = wi.LocalID(0), wi.LocalID(1)
5      let block_sz = wi.WorkSize(0)
6      let matAWidth = matA.GetLength(1)
7      let matBWidth = matB.GetLength(1)
8
9      // Index of the first/last block of A processed
10     let aBegin = block_sz * by
11     let aEnd = aBegin + matAWidth - 1
12     // Index of the first block of B processed
13     let bBegin = block_sz * bx
14
15     let mutable b = bBegin
16     let mutable Csub = 0.0f
17     // Loop over all the sub-matrices of A and B
18     for a in aBegin .. block_sz .. aEnd do
19         let As = local(Array2D.zeroCreate<float32> block_sz block_sz)
20         let Bs = local(Array2D.zeroCreate<float32> block_sz block_sz)
21         // Load the matrices from global memory
22         As.[ty, tx] <- matA.[ty, a + tx]
23         Bs.[ty, tx] <- matB.[b + ty, tx]
24         wi.Barrier(CLK_LOCAL_MEM_FENCE)
25         // Multiply the two matrices together
26         for k = 0 to block_sz - 1 do
27             Csub <- Csub + (As.[ty, k] * Bs.[k, tx])
28         wi.Barrier(CLK_LOCAL_MEM_FENCE)
29         b <- b + block_sz
30     // Write the block sub-matrix to device memory
31     matC.[block_sz * by + ty, block_sz * bx + tx] <- Csub
32
33 // Prepare data
34 let rnd = new Random()
35 let a = Array2D.init 1024 1024 (fun i -> (float32)(rnd.NextDouble()))
36 let b = Array2D.init 1024 1024 (fun i -> (float32)(rnd.NextDouble()))
37 let c = Array2D.zeroCreate<float32> 1024 1024
38 let workSize = WorkSize([| 1024L; 1024L |], [| 16L; 16L |])
39
40 // Run matrix multiplication
41 <@ MatMul(a, b, c, workSize) @>.Run()
```

Listing 10.4: Tiled matrix multiplication

Differently from the previous examples, tiled matrix multiplication requires the definition of a custom kernel. Since the developers operate at the lowest level of abstraction, they have visibility of the underlying OpenCL layer (i.e. work-size, local memory, barriers).

Nonetheless, the FSCL kernel language provides some facilities to make cus-

tom kernel programming easier and more productive if compared to OpenCL C kernel coding. For example, programmers can use the *GetLength* method to retrieve the size of the input/output matrices, saving from explicitly passing additional arguments to the kernel. In addition, matrices can be represented with the native *Array2D* type, which allows to separately address rows and columns. This saves from complex index calculations on flat arrays that conceptually represent multi-dimensional data and simplifies iterating through and accessing specific elements of the matrices.

It's important to note that the custom kernel considered has side effects. In fact, the matrix *matC* is written by the kernel and its content is accessible outside the context of the quoted expression. This is an exeption to the rule provided by the FSCL kernel language (section 6.2.4) for computing expressions containing a single custom kernel, introduced to simplify FSCL programming for OpenCL C programmers. Nonetheless, it is possible to rewrite the kernel to remove side-effects, as illustrated in listing 10.5. Side-effects removal is performed by moving the allocation of *matC* into the kernel body (line 3) and by using it as the kernel return value.

```
1  [<ReflectedDefinition; Kernel>]
2  let MatMul(matA: float32[,], matB: float32[,], wi: WorkItemInfo) =
3      let matC = Array.zeroCreate<float32> (matA.GetLength(0)) (matB.GetLength(1))
4      // Per-work-item code
5      ...
6      // Per-work-item code end, return
7      matC
8
9  // Run matrix multiplication
10 let c = <@ MatMul(a, b, workSize) @>.Run()
```

Listing 10.5: Tiled matrix multiplication with no side-effects

Since FSCL matrix multiplication is expressed through a custom kernel, the resulting code is similar to the one obtained implementing the algorithm in Aparapi. Nonetheless, FSCL enhances productivity in coding matrix multiplication by allowing programmers to use multi-dimensional arrays, keeping the code compact and clean.

It's not clear whether and how Dandelion provides support for user-defined computations, beyond the built-in LINQ operators and library functions. In the paper [58], the authors claim the availability of an *Accelerated* annotation that can be used to mark functions that already have an existing "high-performance" (low-level) implementation. The developer can use that annotation in the form *Accelerated(dev, dll, op)* on a .NET function to override the default cross-compilation. Through this annotation, the programmer tells the system that the particular .NET function considered can be replaced by the function *op* in the DLL *dll* to execute on the device *dev*. This makes

custom kernel invocation similar to the regular .NET PInvoke of functions in unmanaged libraries, which leads us to assume that user-defined kernels must be developed outside the Dandelion environment (e.g. in CUDA).

10.4 Average image complexity

One of the most important aspects of the FSCL programming model is the ability to compose computing elements developed at different levels of abstraction. In this section we consider an algorithm that exemplifies this kind of compositionality. The problem that the algorithm solves consists in computing the average complexity of an image, in terms of the number of "details" it contains. Considering this definition of complexity, a picture of a blue, clear sky or of a sand beach is characterized by a very low complexity. To the contrary, a metropolitan landscape or a foliage are highly complex.

To accomplish the task of computing the average complexity of an image, we apply a Sobel filter to the black-and-white version of the input image [61]. The Sobel filter detects the edges in the input image, producing an image (matrix) where the elements belonging to borders are bright (i.e. float values close to 1) while the elements belonging to "flat" zones are dark (i.e. float values close to 0). We then elaborate the output of Sobel filtering by counting the number of pixels belonging to borders.

In listing 10.6 we illustrate the FSCL code of the algorithm. The first step consists in converting the input image to black-and-white (line 30)[2], producing a float 2D array where each element is in $[0, 1]$. We use the *Array2D.map* collection function to process each pixel independently from the others. The resulting 2D array is then processed by the parallel implementation of Sobel filtering, which requires the definition of a custom kernel *SobelFilter2D* (line 2). To compute the complexity once Sobel has been applied, we leverage the *Array2D.averageBy* collection function, whose operator maps each element of the input array to the corresponding value used in average calculation (line 34). Since we want to count the pixels that belong to borders (which have a bright color), we map to 1 each element whose value is above a certain threshold (0.8 in the example) and to 0 all the others. In the example, *Array2D.map*, *SobelFilter2D* and *Array2D.averageBy* are mapped to OpenCL kernels and executed on each one's most efficient device.

[2]Sobel filtering could be also applied separately to each channel of an RGB image

```
1  [<ReflectedDefinition, Kernel>]
2  let SobelFilter2D (wi: WorkItemInfo) (inIm: float32[,]) =
3     // Create output image using FSCL kernel return capability
4     let outIm = Array2D.zeroCreate<float32> (inIm.GetLength(0) - 2) (inIm.GetLength(1) - 2)
5
6     // Work-item computation
7      let x = wi.GlobalID(0)
8      let y = wi.GlobalID(1)
9      let width = outIm.GetLength(1)
10     let height = outIm.GetLength(0)
11     let mutable Gx = 0.0f
12     let mutable Gy = Gx
13
14     if x < width && y < height then
15         // Read each texel component and calculate the filtered value using neighbouring texel
             components
16         Gx <- inIm.[y, x] + 2.0f * inIm.[y, x + 1] + inIm.[y, x + 2] - inIm.[y + 2, x] - 2.0f *
             inIm.[y + 2, x + 1] - inIm.[y + 2, x + 2]
17         Gy <- inIm.[y, x] - inIm.[y, x + 2] + 2.0f * inIm.[y + 1, x] - 2.0f * inIm.[y + 1, x + 2]
             + inIm.[y + 2, x] - inIm.[y + 2, x + 2]
18         outIm.[y, x] <- Math.Sqrt(Gx * Gx + Gy * Gy)/2.0f
19     // Return
20     outIm
21
22 // Prepare data
23 let image = // Load image into an Array2D<float4> instance
24 let threshold = 0.8f
25
26 // Run avg complexity
27 let avgNoise =
28  <@  image |>
29      // To black-and-white
30      Array2D.map(fun p -> (0.2126f * (float32)p.x + 0.7152f * (float32)p.y + 0.0722f * (float32)p.
          z)) |>
31      // Sobel
32      SobelFilter2D wi |>
33      // Count pixels over the white threshold
34      Array2D.averageBy (fun it ->
35                           if it > threshold then
36                             1.0f
37                           else
38                             0.0f)
39 @>.Run()
```

Listing 10.6: Average image complexity calculation in FSCL

Even though the three-steps execution is easy to understand, we may optimize the sample by merging the *Array2D.map* and the Sobel kernel. In particular, the Sobel kernel may perform color-to-bw conversion for each pixel considered right before calculating the gradients Gx and Gy[3].

It is important to underline the importance for a custom kernel to return values, introduced by the FSCL kernel language on top of classic OpenCL kernel programming. Thanks to this, the Sobel-filtering custom kernel can be composed with other computing elements and the entire algorithm can be express in a single computing expression.

As like as the other algorithms, average image complexity in Aparapi de-

[3]This reasoning applies to regular, sequential programming as well

mands to the programmers to explicitly define each kernel involved. In the specific case, this means creating a function for *Array2D.map*, one for *SobelFilter2D* and one for *Array2D.averageBy*. In Dandelion, Sobel filtering is difficult to express leveraging the provided set of LINQ operators. Therefore programmers should escape the Dandelion environment to code it in OpenCL/CUDA and bind them to the managed environment via PInvoke (if supported), as discussed in section 10.3. Thanks to the homogeneous representation of collection and custom kernels as (particular) F# functions, in FSCL it is instead possible to compose high-level collection kernels and lower-level custom kernels using the same function composition operators that F# programmers employ in traditional, sequential programming.

10.5 Conclusions

In this chapter we demonstrated the capabilities of the FSCL programming model presented and discussed in chapter 6. We walked through some non-trivial examples of real-world parallel algorithms that can be expressed in FSCL properly combining kernels, sequential functions and their composition.

The most relevant aspect of the model resides in the ability to combine computing elements defined at different levels of abstraction, that is custom and collection kernels. Some algorithms, like K-Means and Black-Scholes, can be entirely developed on the highest level of abstraction, without the need for the programmer to know anything about OpenCL programming. Since the FSCL code that expresses the algorithms is close to its sequential counterpart, if not the same, parallel execution can be obtained with few to no efforts, without requiring particular parallel programming skills. Whenever required, programmers can express more complex computations through custom kernels, which deliver the same, high flexibility of OpenCL kernel programming but in a safer environment.

In FSCL, collection functions can be used not only to express single, parallel computations on the elements of an input collection but also to compose sub-expressions. High-order collection functions, like *Array.map* in K-Means, increase the flexibility of the abstract programming level without the need to escape the expression environment. Without higher-order collection function, the computing expression used to code algorithms like K-Means should be divided into multiple expressions to be composed, executed and coordinated by the programmer.

Finally, FSCL employs very popular F# programming constructs and techniques like closures and partial application. When writing a computing expression, programmers can partially apply kernels or utility functions (e.g.

nearestCenter center, in the K-Means example, is a partial application of the *nearestCenter* function) and refer to data declared outside the quotation environment (e.g. the variable *threshold* in the *Array.averageBy* operator of the average image complexity sample). The FSCL compiler and runtime are able to detect these constructs and techniques and to generate and execute OpenCL kernels where data is correctly and reliably passed among kernels and utility functions to reach the scope where data is effectively used.

Chapter 11

Validation of the prediction model for device-selection

In chapter 9 we presented the FSCL runtime scheduling engine. The engine is characterized by a strategy to extract relevant features from the code of arbitrary kernels and by a prediction model, based on linear regression, to correlate the features to the completion time on each available device in the running system.

In this chapter we validate the prediction model discussed in section 9.3. In order to validate the model, we firstly define a set of relevant features to extract and a set of programs that form the training set. Then, we extract the chosen features from each training sample case. Finally, we build a device model (i.e. set of regression coefficients) independently for each device in the running system. Given a device d, we run each sample case on d to obtain the corresponding completion time. With the set of feature values and completion times for all the sample cases, we create the matrix of the explanatory variables $\mathbf{X}$ and a dependent vector $\mathbf{y}$ (9.3), which contains the completion time of the cases on the specific device. We finally apply linear regression to $\mathbf{X}$ and $\mathbf{y}$ to obtain the device model. While feature extraction is performed once, linear regression is repeated for each device in the system.

Once the set of device models has been built, we use it to predict the completion time of a set of testing programs. Each test is also executed to measure its actual completion time. Finally, we calculate the accuracy in selecting the best device for each test in terms of the ratio between the completion time of the estimated fastest device and the measured lowest completion time across the set of devices.

11.1 System setup, training samples and features

To validate the prediction model we setup a heterogeneous system equipped with an APU and a discrete GPU. The APU is a chipset that includes a CPU and a GPU on-die. In the rest of this chapter, with "D-GPU" we indicate the discrete GPU and with "I-GPU" the on-die GPU.

- AMD Fusion A10-5800K (CPU with an integrated AMD HD 7660D GPU)
- AMD HD 7970 (discrete GPU)
- 4GB DDR3 Ram - 1333 Mhz
- Windows 7 64 bit

11.1.1 Training set

In the definition of the training set, we focus on including computations that stress only specific features and device characteristics with a minimal effect on the others. Since this set of samples constitutes the "basis" used to predict the completion time of other computations, we want to start from a minimal set of samples (i.e. small basis) capable of describing a possibly wide set of other algorithms and progressively refine it by introducing additional samples.

After some early experiments, we decide to use the following three algorithms to form the training set used to build the device models. All the algorithms work on single precision floating point values (32 bit).

Vector addition This kernel performs an element-wise sum of two vectors, where each work-item sums the elements matching its own global index. Given the very short execution time and the extremely lightweight nature of the computation, this sample allows to focus on the data-transfer time and on the contribution of the work-space size to the completion time. We execute the kernel varying the input size from 1MB up to 128MB with 1MB step.

Matrix multiplication naive A matrix multiplication kernel where each work-item in a 2D work-space multiplies a row of the first matrix by a column of the second. The first matrix is accessed with a cache-friendly pattern, while the second is accessed with a matrix-width stride that may incur many cache misses. For this reason, matrix multiplication allows

to capture the impact of cache misses on completion time. We run this kernel starting from 64x64 elements matrices up to 2048x2048, with a 64-elements step.

Logistic map A logistic map performed on each element on an input vector filled with random floating point values. Since each work-item performs one only memory access and many arithmetic operations, this sample captures the impact of floating point operations on the completion time. We execute the kernel varying the input size from 1MB up to 128MB with 1MB step and the number of iterations per-work-item from 1000 to 10000.

For each training sample we produce 30 cases characterized by different input sizes. Each case is then run 100 times to get the average completion time and the standard deviation. The total training time, including running the samples, is approximatively two hours.

11.1.2 Features

The selection of the features to extract is related to the aspects stressed by the set of training samples. We consider memory accesses and operations (arithmetic, logic and trascendental) two of the most relevant features to use in order to characterize the completion time. While a given operation takes a fixed, similar amount of time to complete, memory accesses have a different impact depending on whether the data acessed is in cache or not. For this reason, instead of considering the amount of memory accesses we estimate the number of cache misses. We estimate this number by detecting the size of the data cache and cache line on a specific device[1] and by computing the strides of memory accesses in the sample. A first approximation introduced is considering each array separately as if each array was stored in a separate, private data cache. At kernel-compilation time we precompute the access strides to each (global) array and we count the number of accesses with a certain stride by assessing the total trip-count of the loops containing the memory access operations. A second approximation is introduced by considering the largest stride among the memory accesses in a loop, as if the block of memory between the lowest and the highest addresses was entirely used. At kernel-execution time we complete the evaluation of the strides and the relative trip count. This information is then coupled with the cacheline and the size of the data cache to estimate the number of cache misses.

[1]This information can be retrieved through the OpenCL device-query API or running micro-benchmarks

Given a certain number of operations and memory accesses per-work-item, the completion time should be affected by the total number of work-items launched. Therefore, we add the work-space size to the set of the computed features.

Number of operations executed per thread This feature represents the computation as if all of the work-items could perform operations in a pure parallel fashion. In terms of the generic special map function described by the equation 9.2 and the feature finalizer process described in 17 (section 9.2.1), the number of operations can be described as follows.

$$fzbuild_{ops}(k) = fzbuild(smap_{ops}, k)$$

where $smap_{ops}$ is defined as:

$$smap_{ops}(n) = \begin{cases} 1 + dmap(a) + dmap(b) & n = a + b \text{ or} \\ & n = a - b \text{ or} \\ & n = a * b \text{ or} \\ & n = a/b \text{ or} \\ & n = a\%b \text{ or} \\ & n = a|||b \text{ or} \\ & n = a\&\&\&b \text{ or} \\ & n = a \wedge \wedge \wedge b \text{ or} \\ & n = a <<< b \text{ or} \\ & n = a >>> b \text{ or} \\ & n = a||b \text{ or} \\ & n = a\&\&b \\ None & \text{otherwise} \end{cases}$$

Global work-items This feature corresponds to the amount of work-items spawn to execute a computation. Given a kernel, let *params* be its set of parameters and $ws \in params$ the parameter holding the information about the work-item space (section 6.2.1). The finalizer for this feature is expressed as follows:

$$fzbuild_{ws}(k) = fun\ \langle params(k) \rangle \rightarrow \langle ws.GlobalSize(0) * ws.GlobalSize(1) * ws.GlobalSize(2) \rangle$$

In case of 1D work-items space, $ws.GlobalSize(1)$ and $ws.GlobalSize(2)$ are both 1. In case of 2D work-item space, $ws.GlobalSize(2)$ is 1.

Total number of operations executed This feature represents the computation as if all of the operations had the same cost and were executed in a sequential order. This number is obtained as the multiplication of the total number of work-items by the operations executed by each item.

$$fzbuild_{tot-ops}(k) = fun\ \langle params(k) \rangle\ \rightarrow \langle fzbuild_{work}(k) * fzbuild_{ops}(k) \rangle$$

Cache misses This feature captures the time spent waiting for memory accesses that do not hit the cache. The formal definition of this features is given is section 9.2.2 (definition 19). All the training samples and the benchmarks used to evaluate the quality of completion time prediction (section 11.2.2) respect the assumptions and the constraints imposed by the process to build the cache miss estimator.

Kernel-launch overhead The cost of starting a kernel. Since the value of this feature is always 1 (intercept), linear regression charges it with the part of the completion time that it is not able to properly describe in terms of the other features (non-linear terms).

This set of features should be sufficient to investigate the costs of launching a kernel, the amount of time spent performing arithmetic/logic/trascendental operations and the overhead of memory accesses. Several other factors may contribute to the completion time, but we want to start from the coarsest model and progressively refine it as for the set of training samples.

11.2 Fitting residuals, completion time prediction and best-device prediction accuracy

With the setup described in the preceding section we compute the values of the features and the completion times on the available devices for each training sample case.

11.2.1 Fitting residuals

Figure 11.1 shows the residuals of the fitting for both CPU, discrete GPU and integrated GPU. Since we normalize $\mathbf{X}$ by $\mathbf{t}$ (section 9.3), the residuals are shown as relative values. The analysis of the residuals reveals the following:

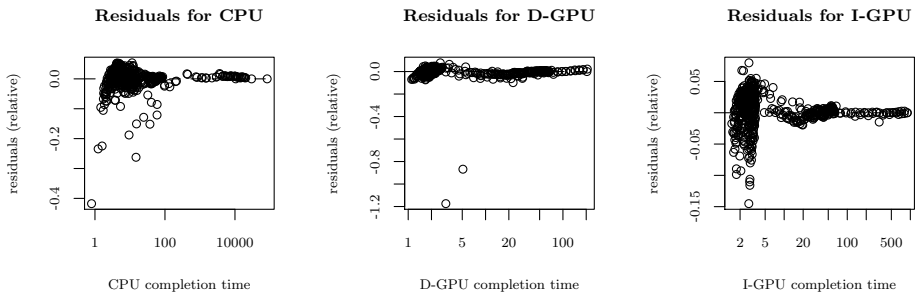

Figure 11.1: Fitting residuals

- Most of the residuals lie near 0, which in general indicates a good fit.
- Residuals are larger for completion times close to zero. This is expected, because cases that have a very short completion time are more likely to be affected by measurement errors induced by the influence of external and unpredictable effects, such as the presence of other processes running in the system. This also suggests that we should expect a relevant prediction error for programs that complete in a short time.

11.2.2 Completion time prediction

To evaluate the error in predicting the completion time of computations using the models built for the devices in the running platform, we define a set of test samples.

Sum of matrix rows This kernel sums the rows of a two-dimensional matrix, producing a vector whose size is equal to the matrix width. Each work-item performs a reduction along a column. As for all the matrix-based samples, we run this kernel starting from 64x64 elements matrices up to 2048x2048, with a 64-elements step.

Sum of matrix columns This kernel sums the columns of a two-dimensional matrix, which results in a vector matching the matrix height. Each work-item performs a reduction along a row.

Matrix multiplication advanced (tiled) A more complex matrix multiplication algorithm which employs local memory to prevent multiple accesses to the same elements of the global memory. The algorithm is the same used to validate the programming model in section 10.3.

Sobel filtering A Sobel 3x3 filtering algorithm, identical to the one shown in section 10.4.

Convolution filtering A generalization of Sobel filtering, with a generic input filter varying in size from 3x3 to 19x19 elements.

Matrix transpose naive Matrix transpose perfomed by making each work-item to transpose a single element of the matrix.

Matrix transpose advanced Matrix transpose that leverages local memory in order to make successive work-items to read and write successive matrix elements, enabling coalescing and reducing channel/bank conflicts.

In the set of test samples we also include the training samples used to build the device models. Using a set of samples to predict themeselves provides, in addition to the set of residuals, an insight on the quality of the fitting.

In figures 11.2 and 11.3 we compare the measured completion time and the estimated completion time for each test sample by varying the input size. The left column reports the measured completion time on CPU, discrete GPU and integrated GPU, while the right column shows the predicted completion time on the same devices. To be easily compared with each other, both the completion times are expressed in the same logarithmic scale.

The first three samples illustrated in figures 11.2 and 11.3 are the training samples used to build the device models. As expected, the predicted completion times of these programs is very close to the measured completion times.

The overall behaviour of most of the samples shows a simple relation between the input size and the completion time. Two noticeable exceptions are *MatMulNaive* and *SumRows*, whose non-monotonicity is well above measurement noise. A possible explanation for the evident spikes in the graphs is the eviction of many cache lines when accessing memory with a stride of 4KB, 6KB and 8KB. These spikes are correctly estimated thanks to the the cache-miss-estimation feature. The same behaviour is correctly predicted in *SumRows*, even though it does not belong to the set of training samples. This is consistent with our expectation: the aforementioned spikes are indeed caused by a particularly aggressive cache eviction.

The prediction of CPU completion times is generally more accurate than the prediction of GPUs. This is mainly due to the fact that the cache-miss-estimation feature models the cost of accessing memory from the CPU with sufficient accuracy, while it doesn't properly fit the GPUs, where the cost of memory accesses depends on the access pattern in a different way. To accurately predict the completion time on GPUs, additional information must

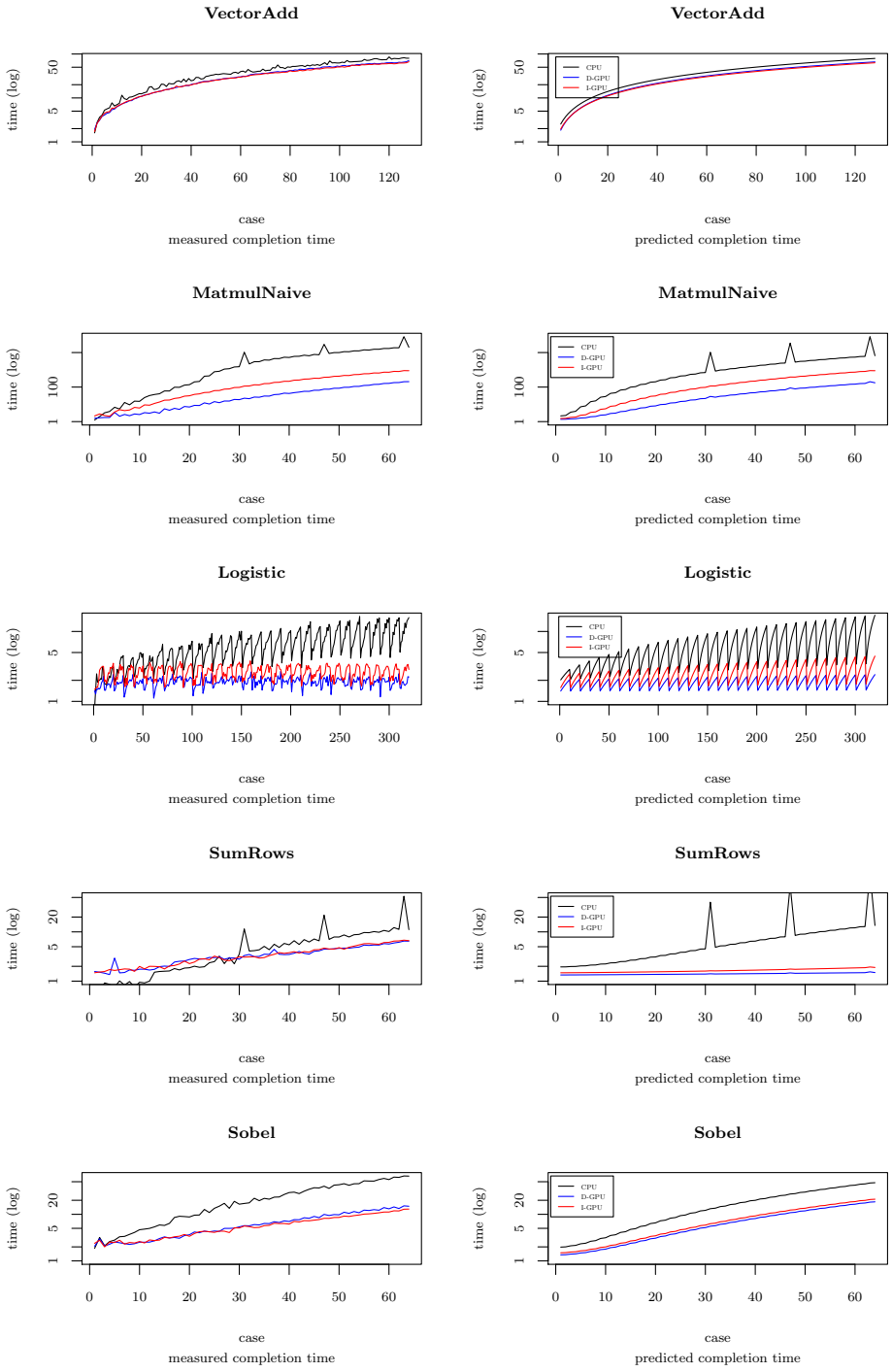

Figure 11.2: Measured and predicted completion time

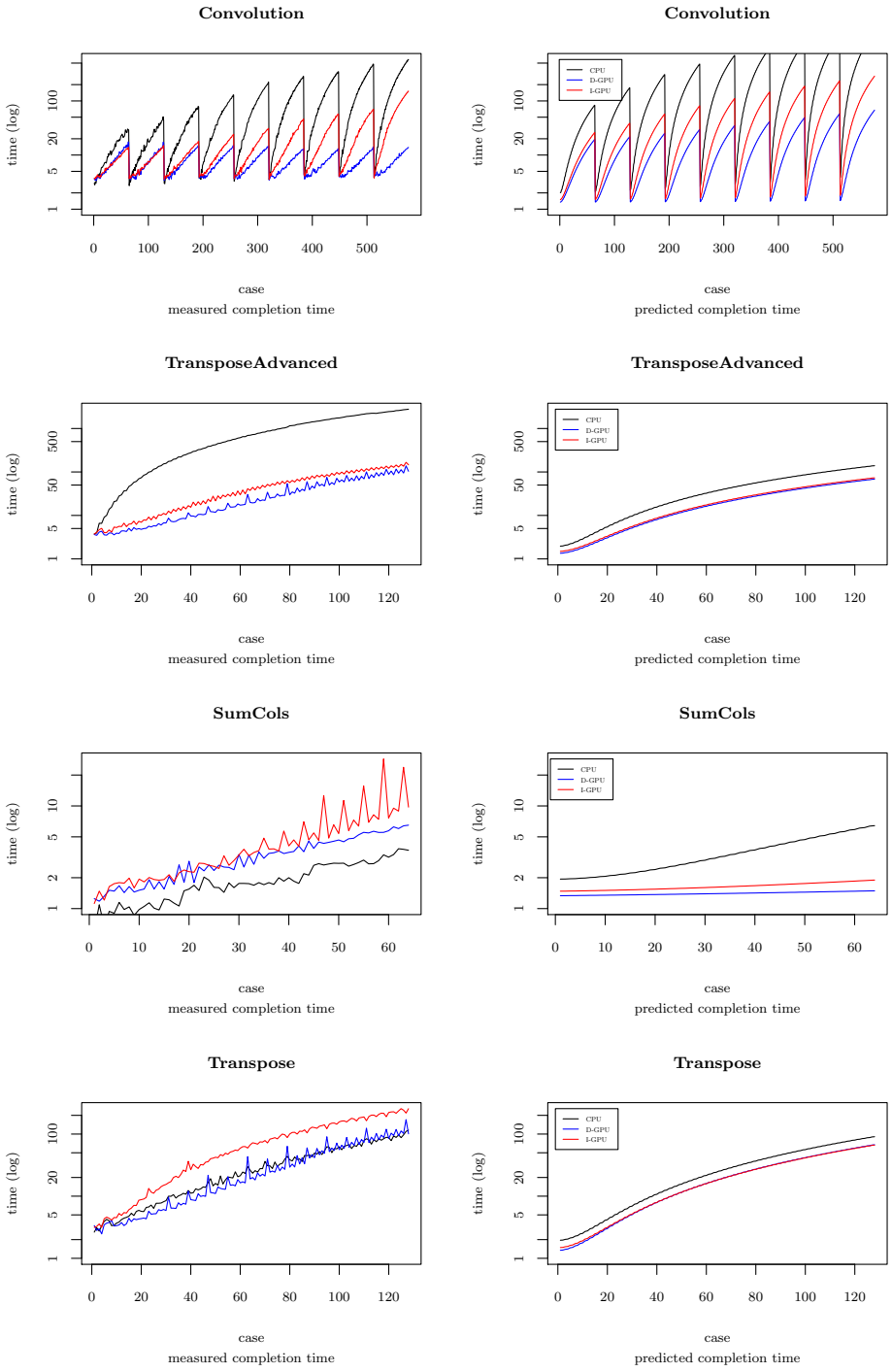

Figure 11.3: Measured and predicted completion time

be retrieved and analysed, such as coalescing in reading and writing memory, ALU fetch ratio[2] and channel/bank conflicts. In addition, information about the usage of LDS (Local Data Share) memory are needed to improve the prediction of *MatMulAdvanced* and *TransposeAdvanced*.

11.2.3 Impact of the feature set on the completion time prediction accuracy

We illustrate how the selection of the features influences the accurancy of the completion time prediction. In particular we assess how the quality of the prediction is affected by considering an increasing number of representative features.

In figures 11.4 and 11.5 we show the impact of feature selection on the error of completion time prediction for two of the samples in the testing set. In the first figure we consider the *Logistic map* sample. The top-left graph reports the measured completion times on the set of available devices by varying the input size. The rightmost figure in the first row (predicted completion time 1) illustrates the completion times predicted using one only feature, which is the total number of operations[3]. Conceptually, the result is the completion time predicted as if the computation was sequential. Symmetrically, the leftmost figure in the second row illustrates the completion times predicted using only the number of operations per-thread (work-item), as if the computation was fully parallel. Finally, the rightmost figure in the second row reports the prediction using the entire set of features except the cache-miss estimation. The set of graphs demonstrates the dependency of the prediction accuracy from the set of features chosen. In particular, by joining different, meaningful features, each one stressing a particular aspect of the computation, we are able to closely estimate the measured completion time.

In figure 11.5 we consider another sample, which is *Matrix multiplication naive*. Again, the top-left graph reports the measured completion times by varying the input size. The rightmost figure in the first row illustrates the prediction resulting from using only the number of operations per-thread. The bottom-left figure instead shows the completion time predicted exclusively leveraging the cache-miss estimation feature. As expected, only the spikes correponding to the input sizes that cause particularly frequent cache misses are modeled. The rightmost figure in the second row reports the prediction using both the features, which "combines" the respective contributions to the completion time.

[2]The occupancy of GPU ALUs during the time a memory request is served

[3]Number of operations per work-item multiplied by the number of items

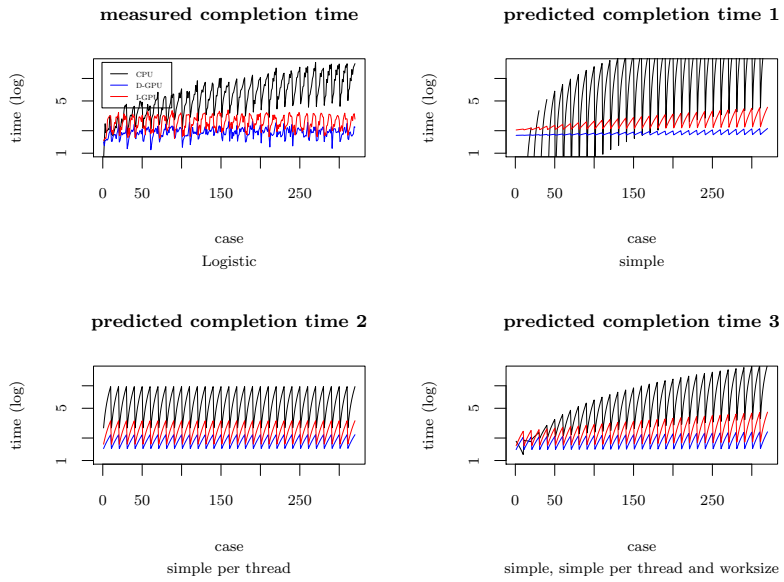

Figure 11.4: Impact of features on prediction error for Logistic map

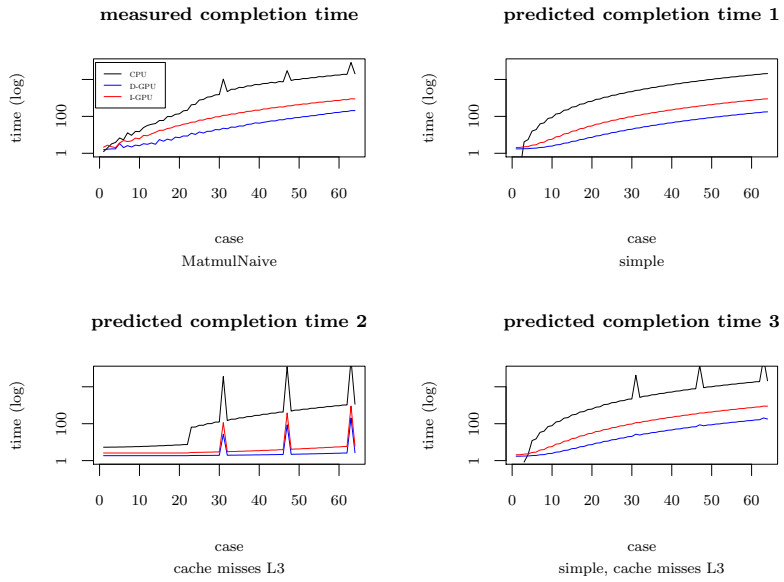

Figure 11.5: Impact of features on prediction error for Matrix multiplication

11.2.4 Best-device prediction

The quality of completion-time prediction is only partially related to the quality of best-device guessing. A very precise prediction of the measured completion times leads to a reliable best-device guess, but errors that may affect the completion time prediction do not necessarily imply a specific error in guessing the most efficient device. It is sufficient to consider a linear model that overestimates the completion time by a constant factor independently from the input size and the device. In such a case, the quality of the completion time estimation is low, but the one of best-device guessing may be instead very high. For example, we may estimate a completion time of 10 and 20 seconds for two different devices, while the respective measured completion time is 20 and 30 seconds. While this implies an error in completion time prediction, the device predicted to be the most efficient is the first one, which corresponds to the real best device between the two considered.

For this reason, we evaluate the accuracy of best-device prediction. Figure 11.6 shows the frequency of the relative prediction accuracy, measured as the ratio between the completion time of the device predicted by our algorithm and the actual optimal device. We also show the geometric mean of the relative accuracy (red line). A geometric mean equal to 1 corresponds to situations where the algorithm always predicts the correct device. When the geometric mean is near to 1 the algorithm does not always predict the best device across the input sizes, but the performance degradation is low, hence the error is neglectable. Higher values of the geometric mean correspond to situations where the difference between the predicted device and the best device is relevant. For example, a geometric mean close to 2 means that the predicted device usually takes twice the time than the best device.

The programs in the basis (*VectorAdd*, *MatMulNaive* and *Logistic*) show an ideal behaviour, with a geometric mean very close to 1, or exactly 1. *MatmulAdvanced* and *Convolution* also show an ideal prediction error. *SumRows*, *Sobel* and *TransposeAdvanced* have a very low geometric mean, with the vast majority of the predictions being accurate. *Transpose* is plotted on a different x scale, because it is the only algorithm for which we obtain prediction errors larger than 2. The geometric mean for this test sample is therefore high (close to 2), even though most of the predictions were accurate (the frequency of the bin relative to 2 is considerably higher than the others combined together).

11.2.5 Interpretation of the regression coefficients

Each linear regression coefficient can be easily interpreted as the time needed to perform one unit of the corresponding feature. As explained in section 9.3

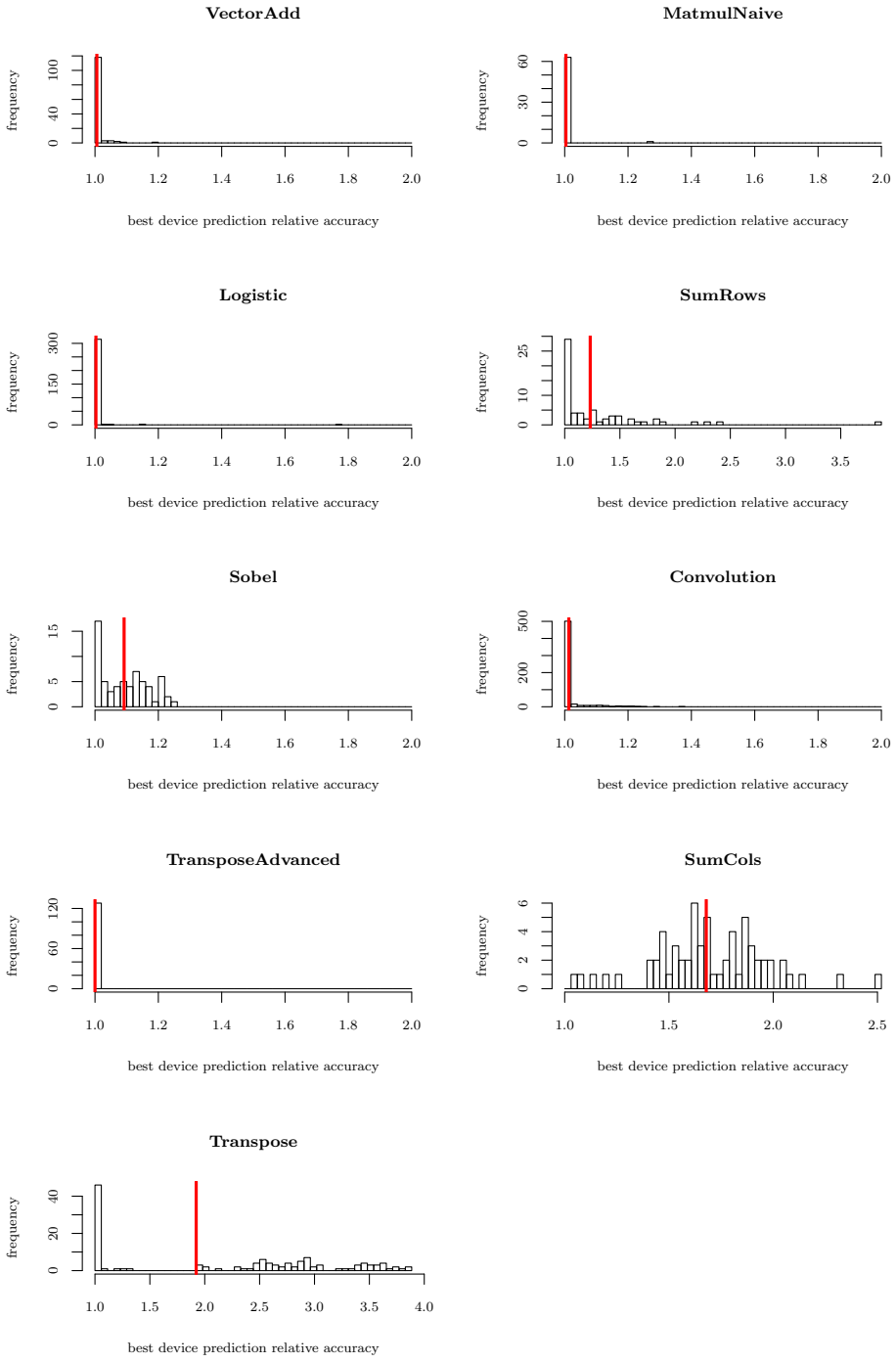

Figure 11.6: Best device prediction relative accuracy

a *valid* feature "counts" the occurrences of a certain phenomenon, while the corresponding coefficient quantifies the time needed for each occurrence. The linear regression applied to the training samples creates a linear model for each device, with a coefficient for each feature. In our experiments, the coefficient for the *total number of instructions executed* feature for the CPU is $0.24e-9$. If we consider this coefficient, the value of the corresponding feature and the completion time on the CPU, we can estimate the number of operations per second of the CPU. The particular value of the regression coefficient for the total numer of instructions leads to an estimated $4.2e9$ operations per second, which means $2.1e9$ operations per second on each of the two CPU cores. This number is close to the declared operating frequency of the CPU, between 3.8 and 4.2 GHz. A similar evaluation can be done for the GPUs.

Given that we count high-level instructions that may not have a one-to-one match with low level executable code, we expected a larger estimation error. Moreover, not all of the low-level instructions require exactly a single clock cycle and many other relevant behaviours are not modeled, such as superscalarity and the effect of branches. Nonetheless, the coefficients and their corresponding physical values are surprisingly close. This shows that our method can accomplish a twofold pourpose: predict the completion time of computations running on heterogeneous platforms and estimate the characteristics of the available devices.

11.2.6 Conclusions

In this chapter we evaluated the best-device prediction model presented in chapter 9 and employed by the FSCL runtime for scheduling kernels in a device-aware fashion.

We built a set of training samples, trying to include programs that stress different characteristics of the various devices. We also defined a set of features that capture the impact of the degree of parallelism and of the operations and memory accesses on the completion time. We applied the regression model described in section 9.3 on these data to obtain a set of regression coefficients for each device in the running platform.

The analysis of the residuals demonstrated a good fitting, with most of the errors concentrated on computations with a very short completion time, which are indeed particularly sensitive to noise. A very short completion time is the result frequently obtained when running lightweight computations working on small datasets. In such cases, all the devices show a very similar completion time, dominated by overhead of the abstraction layer, as discussed in chapter 12. Therefore, errors in predicting the best device should not turn into a sensible loss of performance.

After evaluating the fitting, we assessed the quality of completion time prediction by defining a set of test samples and evaluating the prediction error on the basis of the measured completion times. In most of the cases we are able to reliably predict the completion time on the various devices. In particular, the prediction of the completion time on the CPU is very close to the measured value. Thanks to the cache-miss-estimation feature, the model is able to predict the combinations of code and input size that lead to aggressive cache eviction.

During the validation of the prediction model we also tried to apply the SVM method (including the kernel) proposed in [73] to our data, without being able to replicate the best device prediction accuracy stated in the article. In contrast to SVM methods with kernel tricks, a simple approach like linear regression allows us to analyse the regression coefficients, which can be easily interpreted, and makes it possible to verify and manipulate the features and the training samples used in order to improve accuracy.

In addition to completion-time prediction we also evaluated the quality of best-device guessing. The best-device guessing probability shows a geometry mean close to 1 in most of the cases, which corresponds to a good guess of the most efficient device. In some other cases the mean is higher, suggesting that we should investigate refining the set of features to describe certain completion-time-related aspects that our model is not taking into account, such as conflicts and coalescing in accessing memory.

Chapter 12

Validation of compilation, scheduling and execution efficiency

The efficiency of an abstraction framework for parallel programming and execution is key to for its success. This is particularly true if the framework targets a broad audience of users, from people working on text analysis and manipulation, through developers of images processors to programmers involved in scientific high-performance computing. We see in the lack of comparable performance one of the prominent reasons behind the difficulties for high-level programming frameworks to gain a popularity similar to the one that characterizes OpenCL.

In this chapter we evaluate the efficiency of FSCL, using the same setup used to validate the accuracy of completion time prediction and best-device selection in chapter 11 (three devices, five features). In the first section we analyze the impact of the various strategies proposed in this Thesis in order to increase the efficiency of the abstraction framework and we assess the speedup produced by progressively enabling different optimisations. In the second section we evaluate the performance of FSCL against Aparapi [5] and OpenCL. Finally, in the last section, we evaluate the benefit of applying the device-aware scheduling algorithm proposed in our work, comparing the resulting average completion time to the one obtained with random device selection.

In both FSCL and OpenCL, the "core" of the program execution is a kernel executable running on a device. In OpenCL, the kernel is coded by the programmer in C, while in FSCL the kernel source is generated from collection functions or custom kernels. In all the cases, there are no differences in terms of executable code between an user-defined OpenCL kernel and an FSCL-generated kernel. Therefore, selecting a GPU, a CPU or a coprocessor may

affects the overall completion time but doesn't affect the overhead introduced by FSCL over "raw" OpenCL programs. In other terms, since we are interested in assessing the overhead introduced by FSCL over OpenCL, the particular device chosen for execution is irrelevant. The device selected for execution is also irrelevant in assessing the impact of the FSCL optimisation strategies, such as feature finalisers 9.2 and kernel equivalence 7.4.2, which are applied on high-level programs in a way that is the same regardless the device where the program will run. For these reasons, we choose the 4-core CPU to execute the samples used to evaluate the impact of optimisations and to compare the efficiency of FSCL respect of OpenCL.

12.1 Impact of optimisations on performances

Throughout our research we focused on a set of models and approaches to guarantee the efficiency of kernel compilation, scheduling and execution. In this section we validate the set of choices that we took when designing the language model, the compilation process and the runtime behaviour, which can be summarized as follows:

Feature extraction and evaluation through finalizers In section 9.2 we presented a model to efficiently extract information from arbitrary ASTs. The model is based on precomputing as much information as possible at kernel-compilation time (i.e. the first time the kernel is seen), delivering the final evaluation at kernel-execution time, that is when the actual arguments of the kernel call are provided.

Kernel-compilation invariance and kernel equivalence Given the definition of kernel and metadata equivalence (section 7.4.2), the FSCL compiler can detect when a kernel has been already compiled, which means that an equivalent version has been already seen, processed and stored internally. When an equivalent version of a kernel to process is found, the stored information (i.e. the content of the Kernel Module data-structure) can be used, stopping the kernel-compilation pipeline ahead of time.

OpenCL device and kernel binaries caching When a kernel is scheduled on a device, the FSCL runtime must allocate some OpenCL resources and generate the executable code for the specific device from OpenCL source code. To reduce the overhead, the runtime creates device-specific resources only the first time the device is used, storing them locally for future use. Similarly, as discussed in section 8.2, executable code is

generated only the first time the particular combination of kernel and device is seen.

In figure 12.1, 12.2, 12.3 and 12.4 we show the impact of enabling progressive optimisations for a heterogeneous subset of the training samples used to validate the scheduling approach (chapter 11). Since the optimisations are always applied in the same way independently from the specific device used, we use a single device (the multicore CPU) of the three available to assess the impact of progressive optimisations.

For each algorithm, we compute the average completion time for different input sizes and under a different subset of optimisations. The most relevant performance impact is given by the feature finalizer building process, that isolates most of the analysis overhead at kernel-compilation time. Traversing, building and evaluating expressions has such an impact on perfomances that, if applied each time a kernel is executed, would determine almost the entire computation completion time. Once applied the two-step feature-evaluation model, the efficiency of the framework sensibly increases.

The second, relevant optimisation applied is reusing the output of already-compiled kernels (*kernel-equiv check* in the figures). Since generating the OpenCL source from FSCL kernels involves traversing and manipulating ASTs (F# expressions), in addition to deeply interacting with the reflection/introspection services provided by the CLR, the compiler pipeline has a relevant impact on performances.

The third and fourth optimisations consist in reusing device-specific OpenCL resources already allocated and the set of cached executables of kernels scheduled on a device. While further reducing the completion time, the impact on performance is orders of magnitude lower than the effect of the first two optimisations considered.

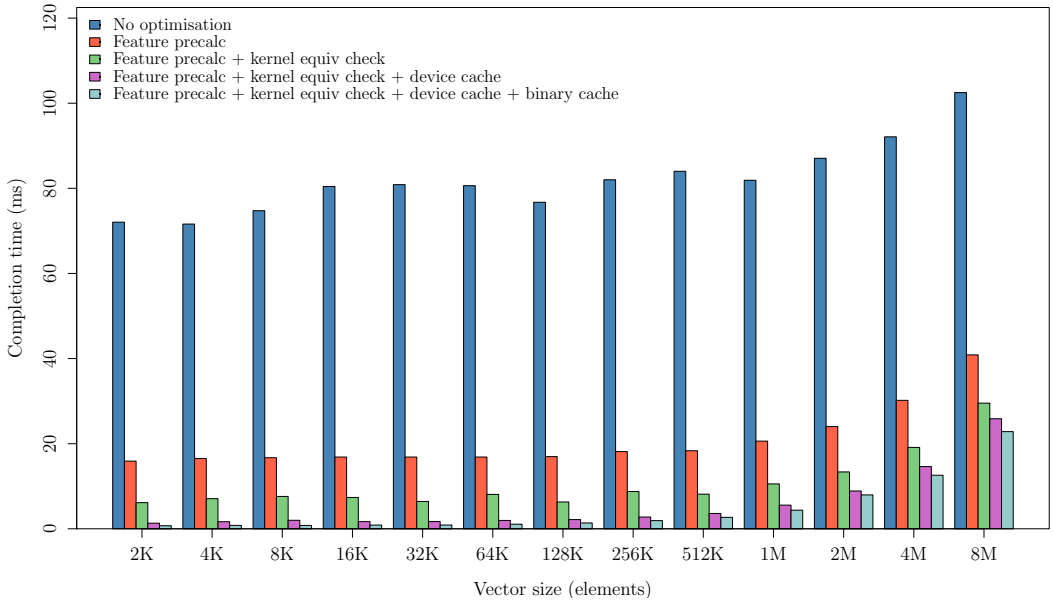

Figure 12.1: Impact of successive optimisations on Vector addition

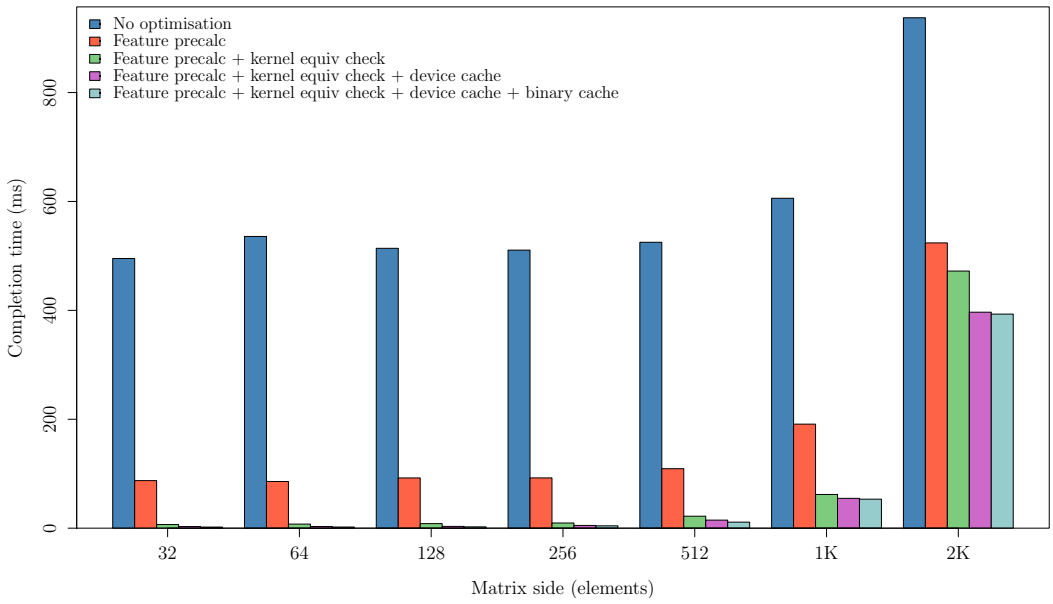

Figure 12.2: Impact of successive optimisations on Matrix multiplication tiled

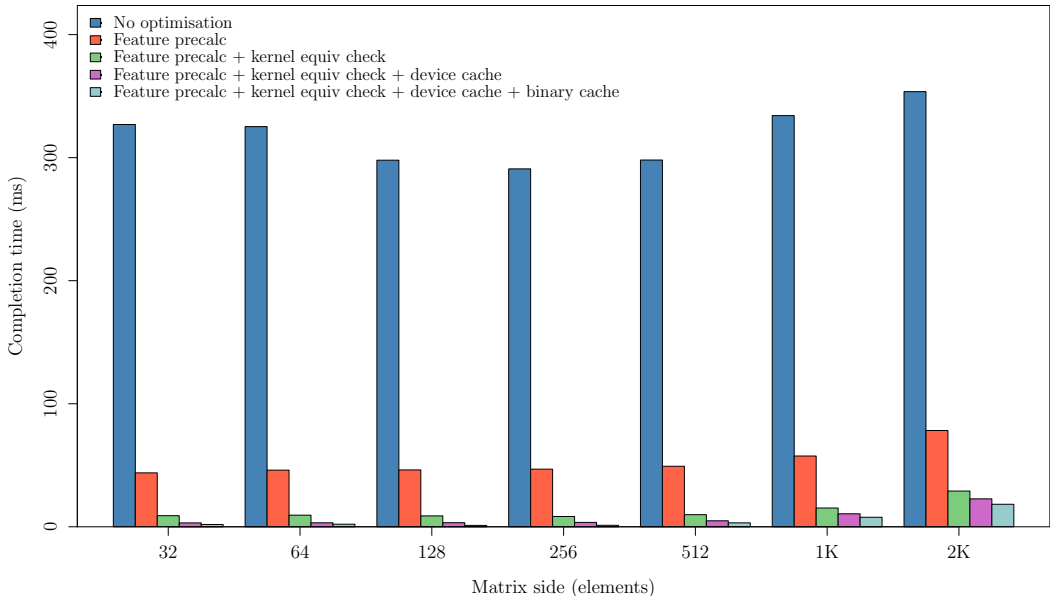

Figure 12.3: Impact of successive optimisations on Convolution

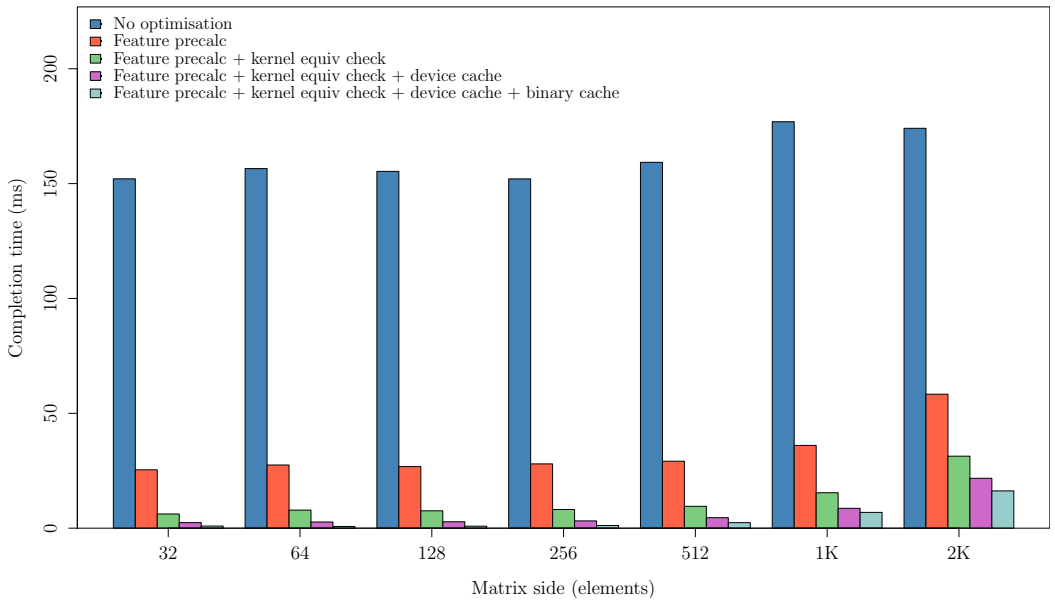

Figure 12.4: Impact of successive optimisations on Matrix transpose

12.2 FSCL versus Aparapi and OpenCL

In this section we consider the optimised version of the FSCL framework and we assess its efficiency compared to other heterogeneous programming solutions. For each algorithm of the four considered in the previous section we take into account the equivalent OpenCL C program, which represents the lowest bounds in completion time or, symmetrically, the highest possible performance. We also evaluate an equivalent program developed in Aparapi [5], which represents the today's framework most similar to FSCL from both the programming and the execution model perspectives. For each of the three solutions considered (including FSCL), we execute each algorithm 100 times and we take the average completion.

As like as in assessing the speedup obtained by enabling different FSCL optimisations, we use a single device (the multicore CPU) of the three available to validate the performance of FSCL against Aparapi and OpenCL.

Figures 12.5, 12.6, 12.7 and 12.8 show the comparison between the completion times of Aparapi, FSCL and OpenCL programs. Aparapi and FSCL are characterized by similar completion times, especially for very small input sizes, where the impact of working on a virtual execution environment is particularly evident. For the same reason, the advantage of low-level OpenCL execution is noticeable. As the input size grows, the FSCL execution shows relevant performance speedup, placing itself between Aparapi and OpenCL. It's important to note that, whereas the generation of OpenCL code and the steps implemented to execute kernels may be similar in FSCL and Aparapi, FSCL also performs feature-extraction and device selection.

By increasing the input size, the gap between the completion time of FSCL and the one of OpenCL becomes narrow, especially in matrix multiplication, whose execution time is orders of magnitude higher than the overhead introduced by the abstraction framework.

12.3 Impact of feature evaluation and device selection on average completion time

Reducing the absolute overhead over regular OpenCL C programming and execution is not the only goal of the framework efficiency. As discussed in the introduction of chapter 9 and in section 9.2, transparent scheduling is meaningful as long as the processing required to select the best device doesn't overweight the computation time saved. In particular, the scheduling overhead plus the completion time on the selected device should be lower than the completion time expected by randomly picking a device. We verify this

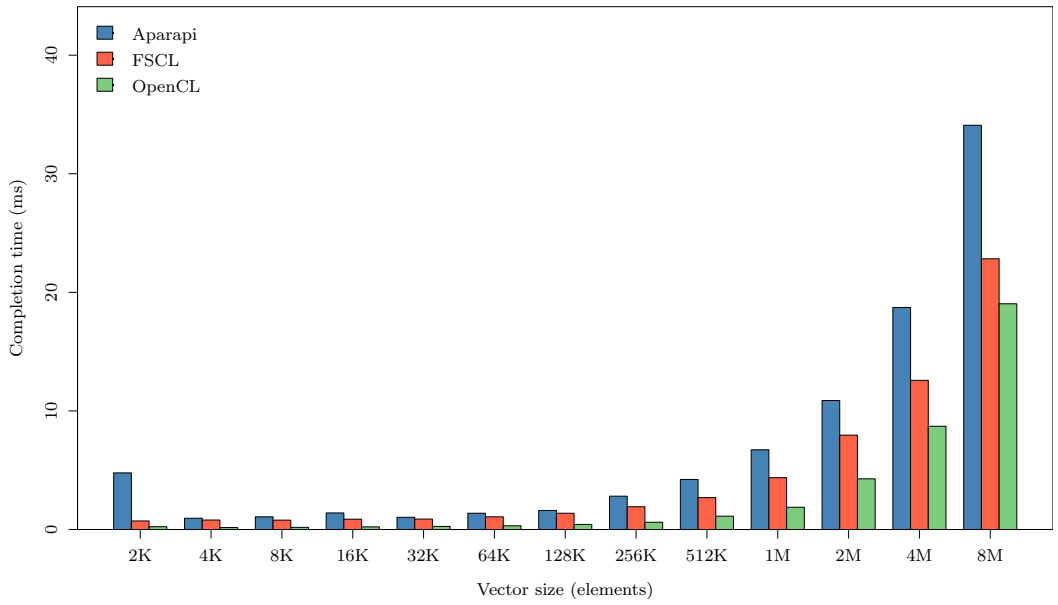

Figure 12.5: Aparapi vs FSCL vs OpenCL for Vector addition

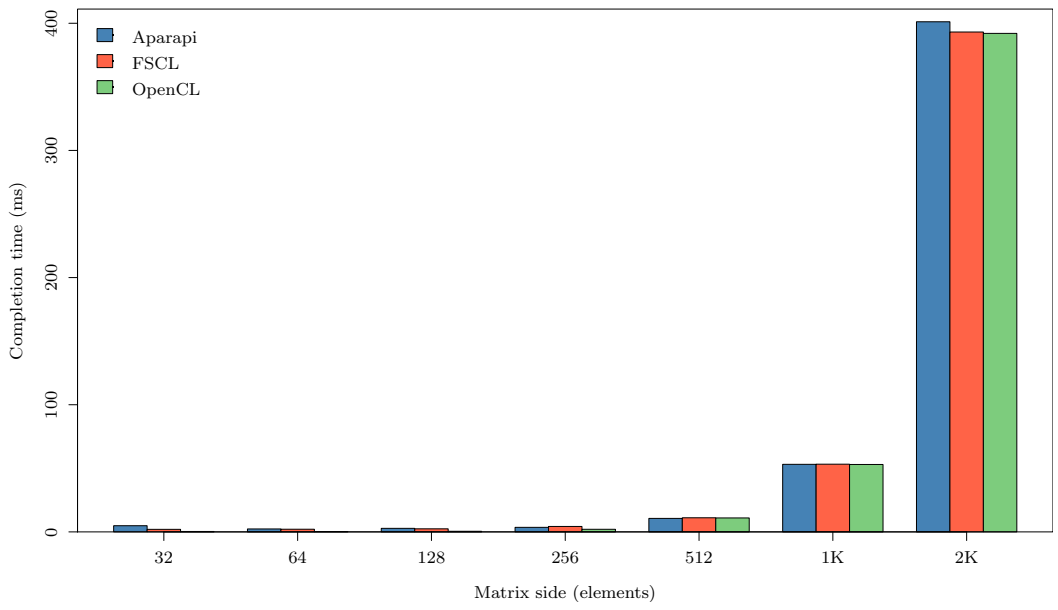

Figure 12.6: Aparapi vs FSCL vs OpenCL for Matrix multiplication tiled

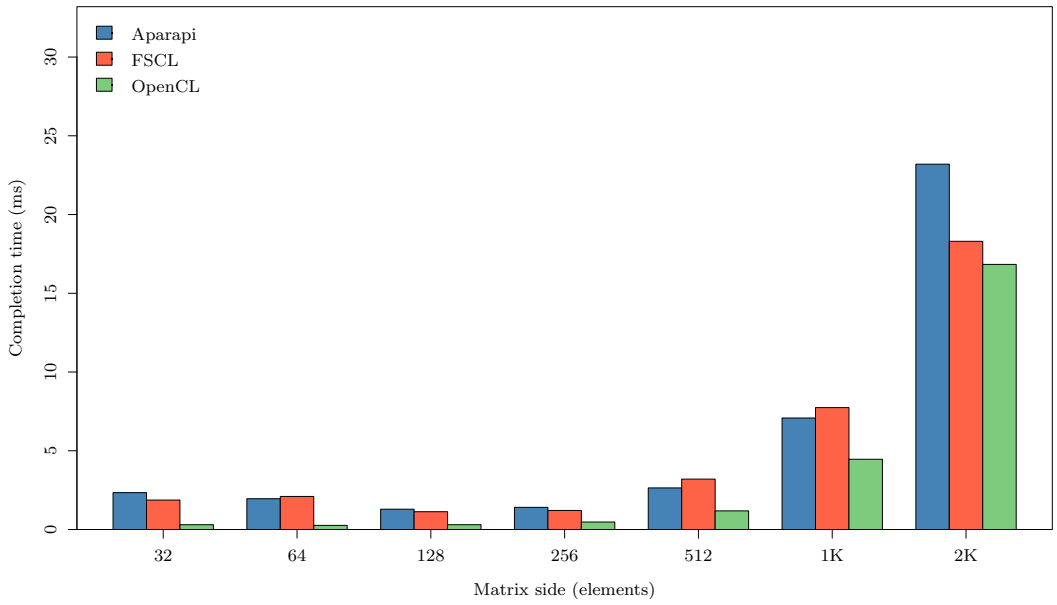

Figure 12.7: Aparapi vs FSCL vs OpenCL for Convolution

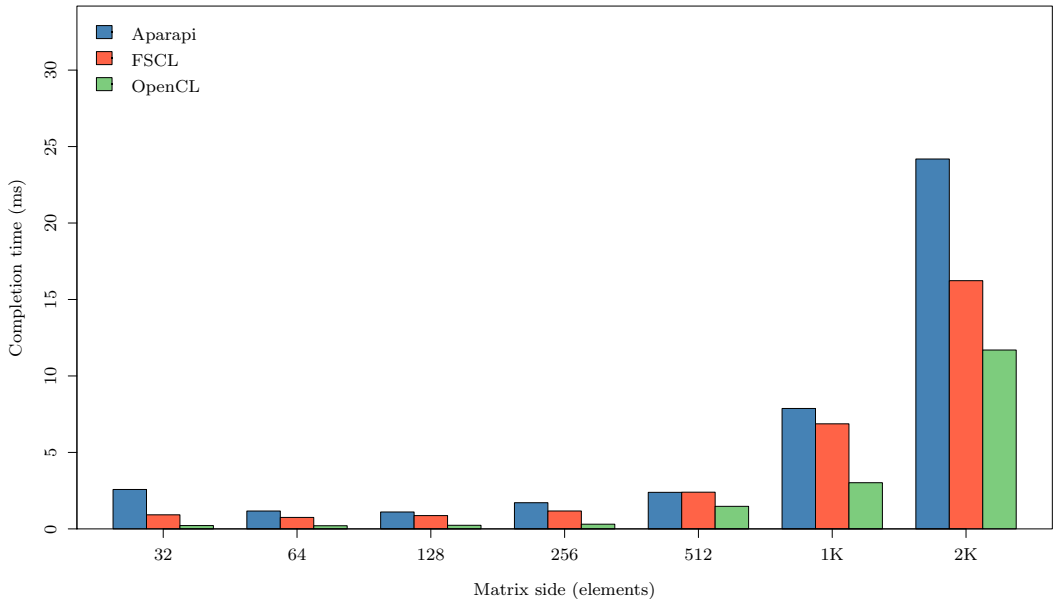

Figure 12.8: Aparapi vs FSCL vs OpenCL for Matrix transpose

assertion for the set of algorithms considered so far, assessing the speedup of best-device selection over random selection. If t_{best} is the completion time on the best device, t_{avg} is the average completion time on the set of available devices and $t_{schedule}$ is the overhead introduced by device selection[1], we can define *best-device-speedup* the ratio between t_{avg} and $t_{best} + t_{schedule}$:

$$\textit{Best-device-speedup} = \frac{t_{avg}}{t_{best} + t_{schedule}}$$

When this ratio is lower than 1, the scheduling overhead outweighs the time saved by executing on the device with the lowest completion time. When equal to or higher than 1, the overhead of code analysis and device selection is balanced by the performance obtained running on the chosen device. The higher the speedup the more convenient is to apply the best-device scheduling strategy.

In figure 12.9 we show the best-device-speedup for the algorithms considered. As illustrated, applying the device-aware scheduling strategy leads to a relevant speedup in most of the cases, even though a small overhead is introduced to analyze kernel code and to apply the set of device-specific models to the extracted features. The benefit of proper device-selection is embarrassingly high in case of matrix multiplication, where the performance on the best device (discrete GPU) is orders of magnitude higher than the one obtained in the worst case scenario (CPU). To the contrary, the completion time of vector addition is mostly uniform across the devices. For this reason, device-aware scheduling can't improve the performance obtained with random device selection. Nevertheless, the slowdown caused by the scheduling overhead is very low (about 10%).

[1]This time includes the time needed to evaluate the feature finalizers

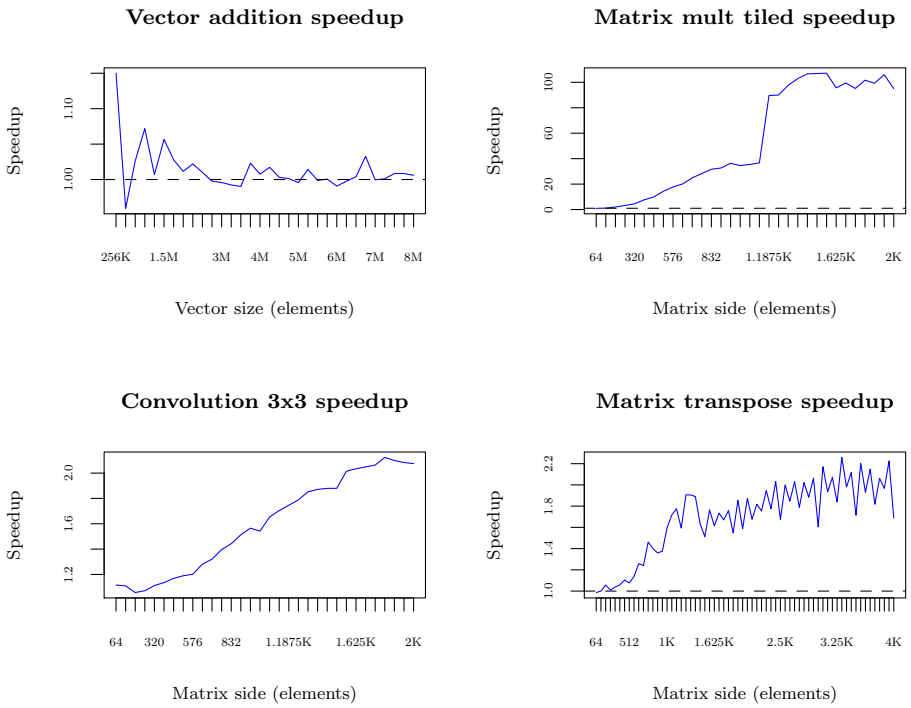

Figure 12.9: Speedup of best-device over random device selection

12.3.1 Single versus hybrid execution for multi-kernel programs

To assess the overall benefit of automatic best-device selection in running an FSCL program, we develop and execute Newton's method for the approximation of matrix inverse [31], which is a procedure suitable to be expressed in its parallel form by composing multiple kernels.

Starting from an initial seed (matrix) X_0, Newton's method iteratively converges to the inverse of a given matrix A according to the following equations:

$$X_0 = A^T * \frac{1}{\|A\|_1 * \|A\|_\infty} \tag{12.1}$$

$$X_{k+1} = 2X_k - X_k A X_k \tag{12.2}$$

We precompute the norms and we assign to a variable α the scalar multiplier for A^T defined in the equation 12.1.

$$\alpha = \frac{1}{\|A\|_1 * \|A\|_\infty}$$

The FSCL parallel algorithm to perform k iterations of the Newton's method is shown in the listing 12.1. To iteratively execute the method we use the *Array.fold* collection function (line 5). At each iteration, given the current state X, we multiply X by 2 (line 7) and we perform two matrix multiplications (line 8). The results of these two computations constitute the input of the *Array.map2* function (line 11), which subtracts the second from the first according to the equation 12.2. The initial seed (state) is computed by transposing the input matrix A (line 13) and multiplying the result by α.

The kernels involved in the computation are *MatMult* (executed twice), *MatTransp*, *Array2D.map* and *Array2D.map2*. In particular, two kernels are generated for *Array2D.map2*. The first is generated for the computation at line 7 and the second for the computation at line 14[2] The function *Array.fold* is instead executed on the host, since it is an higher-order collection function.

[2]The two computations are not structurally equivalent, since the functional arguments are inequivalent lambdas.

```
1  // Run kmeans iteratively
2  let inverse =
3   <@
4      [| 1 .. k |] |>
5      Array.fold(fun it X _ ->
6                  (
7                      (Array2D.map(fun i -> i * 2.0)),
8                       (MatMult ws A X |>
9                        MatMult ws X)
10                 ) ||>
11                 Array2D.map2(fun i j -> i - j))
12               (A |>
13                MatTransp ws |>
14                Array2D.map(fun i -> i * multip))
15   @>.Run()
```

Listing 12.1: Newton's method for matrix inverse approximation

We set the variable k, which represents the number of Newton's method iterations, to 10 and we consider a range of matrix sizes from 64x64 to 1024x1024 elements. For each input size, execution is performed 100 times and the average completion time is finally computed.

In figure 12.10 we show the completion times of the kernels composing the FSCL program, each executed independently. As illustrated, matrix multiplication is the only kernel where the discrete GPU is the device with the lowest overall completion time. In all the other cases, the CPU is generally the most efficient device. The intergated GPU isn't the favorable device for any kernel in the program.

Since the discrete GPU is not the best device for all the kernels in the FSCL program, it is meaningful to investigate whether hybrid execution allows to lower the completion time of single-device execution.

To evaluate the benefit of hybrid execution, we consider four different scheduling configurations:

- All the kernels on the CPU;
- All the kernels on the Integrated GPU;
- All the kernels on the Discrete GPU;
- Kernels automatically scheduled each on its estimated best device.

In case of hybrid execution, we assess that the scheduling strategy always selects the discrete GPU for matrix multiplication and matrix transpose in-

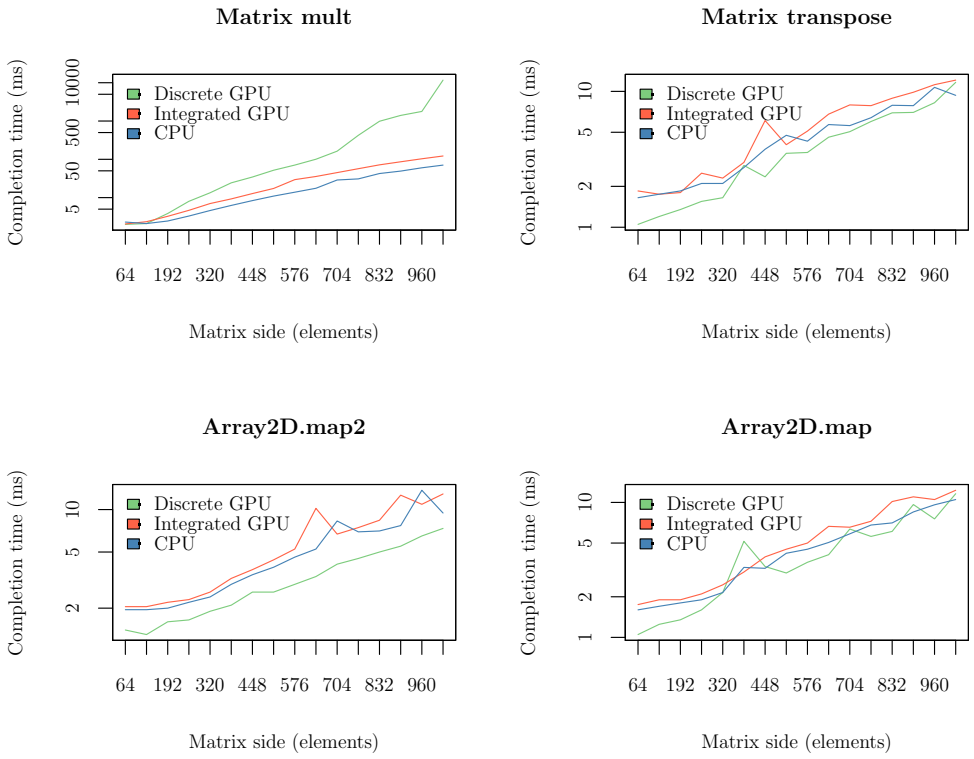

Figure 12.10: Completion times for each kernel in the Newton's method

dependently from the input size, while the CPU is always selected to execute *Array2D.map* and *Array2D.map2*.

In figure 12.11 we compare the completion times of the four configurations in logarithmic scale. As shown, the completion times of hybrid and discrete-GPU-only execution are close to each other, mainly due to the impact of matrix multiplication, whose completion time is about two orders of magnitude higher than the completion times of the other algorithms. Nevertheless, in the range of sizes 192-832 elements, hybrid execution performance outweighs best single-device execution performance (i.e. single-execution on the discrete GPU). The performance gap is underlined in figure 12.12, which shows the completion times of discrete GPU and hybrid execution only.

For very small input sizes hybrid execution has no impact on performance, since the completion times of all the devices are very similar to each other. Processing big matrices (896-1024 elements) represents another case where hybrid execution has negative or no impact on performance. This is mostly due to the effect of cross-device data-transfer, which outweighs the time saved scheduling each algorithm on the best device, suggesting that both execution time and data-transfer overhead should be taken into account in the scheduling policy to improve overall performance of multi-kernel/multi-device FSCL programs. We discuss the possibility to extend the scheduling strategy to consider the impact of data transfer in chapter 13.

Even though not constant, the time required to evaluate a set features has a low variance across the algorithms. This time mainly depends on the size of the finalizer AST, which in its turn depends on the length and complexity of the kernel code. Considering our samples, the time ranges from 0.14 milliseconds for samples with very small ASTs (vector addition, transpose) to 0.5 milliseconds for samples with a quite complex code structure (matrix multiplication and convolution). Half a millisecond is an overhead that is generally order of magnitudes lower than the time saved scheduling on the faster device, even for quite small input sizes. Further improvements may be obtained evaluating multiple features using a single finalizer, which would reduce the computation load, and compiling quotations into IL when building the feature finalizers.

12.4 Conclusions

In this chapter we evaluated the efficiency of FSCL, focusing on the impact on performances of the optimisations applied by the set of framework layers. As expected, the two most relevant performance improvements are the feature-finalizer construction and the kernel equivalence check. The first improvement

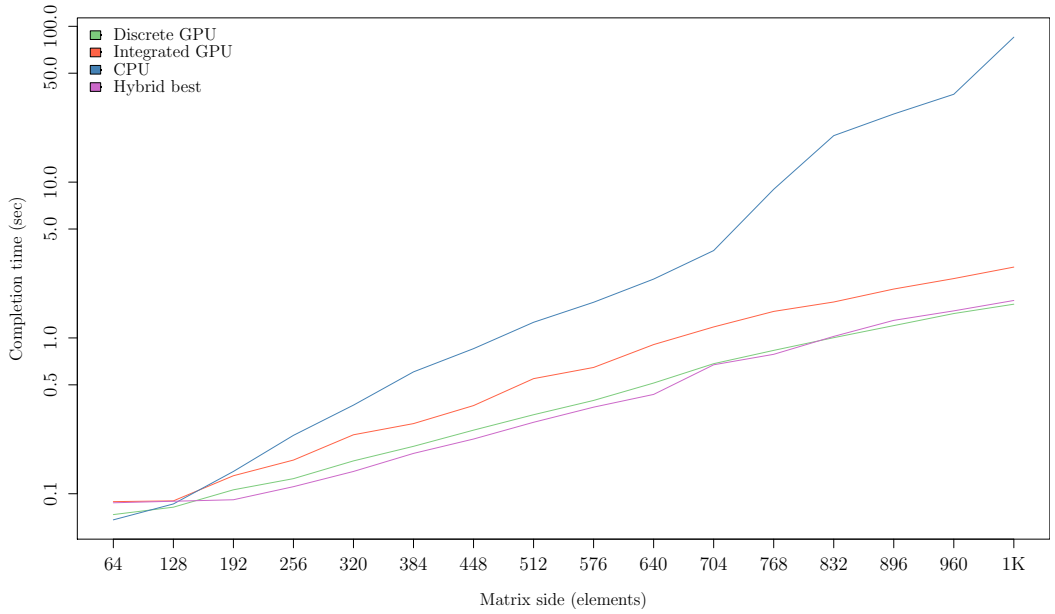

Figure 12.11: Completion times of single device execution and hybrid execution, logarithmic scale

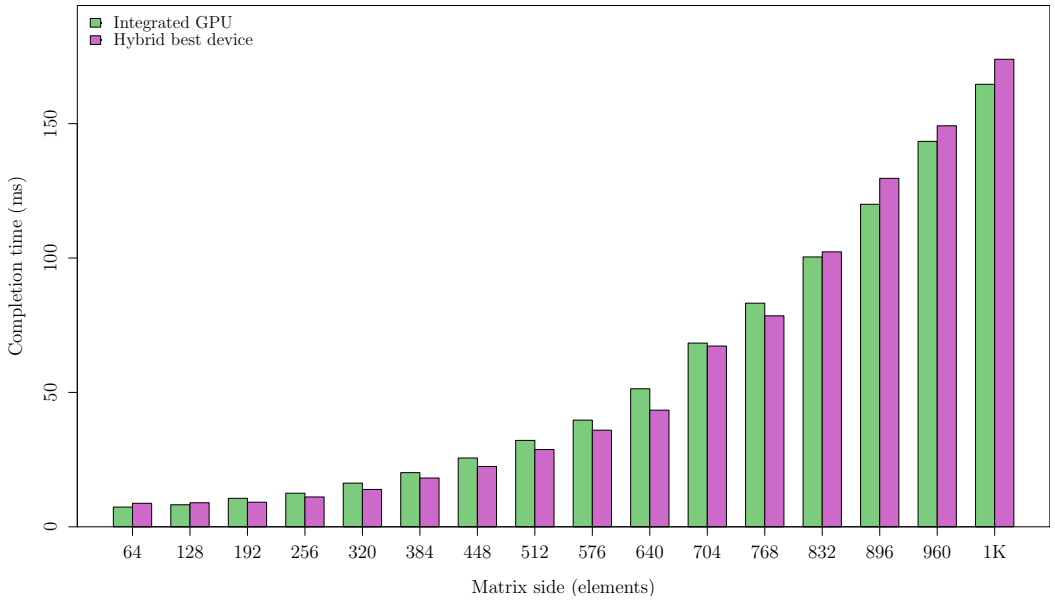

Figure 12.12: Completion times of discrete-GPU-only versus hybrid execution

allows to isolate most of the code-analysis at kernel-compilation time, strongly reducing the overhead when the kernel is executed. The second optimisation saves from re-generating the OpenCL sources for already-compiled kernels, interrupting the compilation pipeline ahead of time.

Given the optimisations introduced, the FSCL framework can transparently execute kernels with performances close to the ones delivered by low-level OpenCL programs. This is especially true for non-trivial algorithms, for which the computing time on the device is particularly high.

Other than verifying the suitability of the high-level programming and execution framework to be used in place of low-level OpenCL without compromising performances, we assessed the efficacy of the device-aware scheduling approach in terms of the speedup provided over random device selection. The results show that in most of the cases, paying the cost of feature extraction and device-guessing results in a sensible performance improvement. The only scenario where the scheduling engine represents a slowdown is when the completion times on the available devices are close to each other. Despite the negative impact on performances, our validation assessed that the slowdown introduced is mostly irrelevant, primarly thanks to the efficiency of feature-finalizers evaluation.

Part IV

Conclusions

Chapter 13

Research, challenges and results

In this Thesis we presented our research towards raising abstraction over programming and execution on heterogeneous platforms. In particular, we focused on two major challenges, which are the complexity of programming across different parallel devices and the difficulty to characterize the set of available devices in order to schedule parallel computations in a device- and computation-aware fashion.

Throughout our research, we tried to balance *abstraction* and *flexibility*. From the programming perspective, abstraction enhances productivity and extends the audience towards developers with few to no parallel programming skills. Flexibility allows instead to widen the range of computations that can be expressed with the provided model, increasing the applicability of the language to accelerate algorithms coming from an heterogeneous set of scientific fields.

From the perspective of scheduling computations on multi-device systems, abstraction allows to hide the details of the running platform to the programmers, enhancing the ubiquitousness of parallel programs and simplifying dynamic exploitation of the available computing power. Flexibility gives the chance to the users to tune or override the decisions of the scheduling policy.

Finally, from the perspective of execution and coordination, abstraction simplifies the composition of computing elements and data-passing among them, allowing the developers to focus on defining *what* the parallel computations have to do instead of spending time coding *how* they should be executed. Flexibility comes into play to enable control over data allocation, initialization and data passing to optimise the execution and to enhance runtime performance.

A third aspect taken into account, especially in the programming model, is *expressiveness*. An expressive programming interface helps the developers to feel comfortable with the concrete language features provided because intuitive to map to the corresponding abstract concepts.

Finally, a struggling challenge is to embrace abstraction, flexibility and expressiveness while delivering efficiency. For this reason, each step of our research was permeated by strategies to preserve the high performances of low-level programming layers and of user-defined scheduling and execution approaches.

We defined a parallel programming model seamlessly integrated in F#. The model is based on a set of parallel computing units and a set of operators/functions to compose and coordinate them in a way that strictly resembles regular F# functional processing on collections. The model exposes two different combinations of abstraction and flexibility in both defining and composing computations.

The developers can code parallel computations and their compositions leveraging on built-in F# collection functions, with no differences from regular, sequential collection processing and no visibility of the underlying OpenCL layer. Collection functions are expressive, succinct and well-known all across the F# community.

To improve the flexibility of skeleton-based solutions, enabling a wider range of algorithms to be expressed, our programming model allows to replace one or more collection functions with custom kernels, which directly map to OpenCL kernels. Custom kernels break the abstraction over the underlying OpenCL layer but, at the same time, improves productivity of low-level OpenCL programming thanks to additional, high-level constructs (e.g. tuples, multi-dimensional arrays, return types) and to type checking and other services characterizing Virtual Machine languages.

To improve the flexibility of composing parallel computations, the model gives the chance to "break" a composition of computations into multiple expressions, each of which can be executed separately in the imperative/object-oriented host-side program. The programmers are responsible for ensuring the correct order of execution and the appropriate data passing, with a fine-grained control over composition and coordination at the price of a lower abstraction.

To map the elements of our programming model to the underlying OpenCL layer we developed a source-to-source compiler, capable of analysing the composition of computations in the input expression and to generate the OpenCL source code for each kernel that the expression contains.

Thanks to the F# quotations mechanism, exposing an expression to the compilation process is unobtrusive, only requiring to put the expression within the quotation delimiters in order to obtain its Abstract Syntax Tree.

We defined an equivalence model for computations and user-defined annotations (metadata) in order to reduce the overhead at runtime. The equivalence

model is employed to determine when two inputs produce the same compilation output and, consequently, to avoid full-compilation of already-compiled computations.

To execute a parallel program, we investigated an abstract execution layer with performances close to the low-level OpenCL host-side programming. The result is a runtime framework that cuts off host-side coding, including resource allocation, kernel compilation, device discovery and data-transfer, allowing programmers to focus exclusively on coding computations while delivering fully automatic execution and coordination.

Flexibility is achieved through the same annotation mechanism exposed to control compilation. At runtime, annotations enable the developers to control the memory-placement of buffers, the data-transfer criteria and the scheduling policy applied to parallel computations.

While delivering much higher abstraction over parallel execution, we investigated various approaches to preserve the runtime performance. We defined a strategy to cache compiled kernels and device-specific OpenCL resources. We also proposed a model to create and reuse buffers depending on whether they can be accessed or not from the managed environment and from multiple computations. Finally, despite the transparency of the execution layer, the runtime framework emulates good-programming practices in data-allocation and transfer, depending on the particular usage of data and on the characteristics of the device chosen for execution.

For scheduling kernels, we proposed a self-contained, adaptive strategy capable of dynamically discovering the platform configuration and of self-training to build a completion-time prediction model for the devices populating the platform.

We defined a model for efficient feature-extraction from ASTs that precomputes features at kernel-compilation time, strongly reducing the overhead at kernel-execution time.

We built a prediction model based on robust linear regression in order to correlate code features and completion time for arbitrary OpenCL devices.

We finally joined these two models to create a scheduling algorithm that generates a completion-time prediction model for each device in the running platform at deploy-time, using an extensible set of training samples. At runtime, the engine transparently extracts the relevant features from each computation to run and estimates the performance on each available device, eventually selecting the one with the lowest completion time.

All the researches conducted and the models and strategies proposed have been implemented in the FSCL framework, which is an open-source cross-

platform project hosted on GitHub [27].

Chapter 14

Limitations, refinements and future works

We see many potential future developments of our research in raising abstraction over heterogeneous programming, scheduling and execution. In this chapter we examine some of the most relevant from our perspective. In the first section we summarize the principal limitations of the results obtained and we discuss potential refinements and improvements. In the second section we propose some interesting concurrent researches that can draw ispiration from our work and leverage the results of our research.

14.1 Research refinements

The FSCL programming model delivers a combination of abstraction and flexibility that allows to express a wide variety of parallel computations, selecting the best-fitting abstraction level from time to time. On top of OpenCL specification 1.2, the model introduces high-order collection functions and function composition operators to express host-side (multithread) parallelism in executing kernels. Despite the high expressiveness of collection functions and function composition, the FSCL programming model is constrained by the "target" OpenCL model.

In OpenCL, the host-side is the only entity responsible for scheduling and coordinating kernels on the available devices. In particular, kernels cannot schedule other kernels and wait for their result. The relation between the host and the set of devices in OpenCL strictly resembles the one between the scheduler (also known as *master*) and the workers (*slaves*) in the *farm* skeleton.

Among the influences on the FSCL programming and execution model generated by this host-device interaction scheme, the two most evident are the

inability for a custom FSCL kernel to call (execute) other kernels and the need to execute function and collection composition on the host-side.

With the recent release of version 2.0 of the specification, OpenCL introduced a new feature called *kernel-side command-queue*, that allows a kernel to schedule other kernels independently from the host. This new feature has an impact on the OpenCL layer that may allow even more expressive high-level models to be mapped to OpenCL. For example, since kernels can schedule the execution of other kernels, the FSCL language could be extended to allow custom kernels to use (call) collection functions, such as *Array.map* and *Array.groupBy*. The compiler could be consequently instrumented to detect such calls, to generate the source for each kernel and finally to "replace" each call with the OpenCL code needed to schedule the corresponding kernel. This extension would spread the usage of collection functions across all the levels of the FSCL programming and execution model. Depending on the case and with complete freedom, the compiler/runtime may in fact interpret a certain collection function as an host-side coordination of kernels or as an OpenCL kernel. This interpretation may be also driven by particular optimisations (e.g. interpret a collection function as a kernel to run on the GPU in order to avoid stealing CPU resources from another concurrent kernel executing on the CPU).

Moreover, this extension would make the FSCL kernel language completely symmetrical from the programmer's perspective. Currently, operators of collection functions can execute custom kernels (i.e. collection composition), but custom kernels cannot execute collection functions. Similarly, the operator of a collection function can contain a call to another collection function, but custom kernels cannot contain calls to other custom kernels. In other terms, collection functions can be nested with no restrictions, while strong restrictions apply to custom kernels. OpenCL 2.0 kernel-side command queues may play a key role in removing these limitations that break the symmetry of the language components.

F# provides a nearly-unique feature called *type providers* [13, 68]. Type providers provide types, properties, and methods at runtime, eliminating barriers to working with diverse information sources. In our work, type providers may be succesfully employed to allow to use and compose already defined OpenCL C kernels in the FSCL environment, exposing them as F# functions (i.e. the opposite process of FSCL compilation) or as a data structure that contains all the informations needed, such parameters types, to execute the kernel from within the FSCL Runtime.

For what regards the data-types used by computing expressions, it would

be interesting to investigate how F# *seq*, which is a lazy collection type, can be employed to enhance expressiveness and flexibility. In particular, lazy collections could be used to model a data-stream in stream-parallelism. Lazy collection may therefore represent a way for the programmer to specify that a computing expression, or part of it, should execute on a stream of data.

Other than a set of constructs to define and compose parallel computations, the FSCL kernel language defines a metadata infrastructure to associate meta-information to the elements of a computing expression in order to drive compilation and execution (section 6.2.5).

Metadata are tighly coupled with the definition of *kernel equivalence* and, consequently, with the one-time-compilation strategy that allows to reuse the OpenCL-source generated for kernels, strongly reducing the overhead of FSCL-to-OpenCL compilation. For this reason, the FSCL framework allows to associate metadata exclusively to single computing elements. Nontheless, metadata may be succesfully employed to associate information to function composition operators and to collection composition functions as well. For example, metadata could be used to define the runtime behaviour of a collection composition function (e.g. *Array.map*), such as whether to run it multithreading or sequentially.

As discussed in section 6.2.4, for the purpose of our research we provided a multithread implementation of collection compositions. As previously mentioned, the release of OpenCL 2.0 enables kernels to schedule other kernels independently from the host, unleashing the possibility to map to OpenCL also collection and function compositions, acting as kernels executing other (sub)kernels. Since this would translate both composition functions and collection/custom kernels to OpenCL sources, extending the metadata infrastructure to cover both kernels and the constructs used to compose them is even more important to guarantee the homogeneusness of metadata application.

Since metadata can affect the compilation output, our programming and compilation model formalizes the definition of *Kernel Module equivalence* (definition 12) to determine when two FSCL kernels are mapped to the same OpenCL kernel. Extending the metadata infrastructure to function and collection composition requires to introduce a similar definition for collection functions and function operators used to compose sub-expressions. It is also required to investigate how the inequivalence between two compositions affects the equivalence of the kernels composed. An interesting example is given by a computing expression with a pattern *Array.map* |> *Array.reduce*. Metadata could be associated to the function composition operator "|>" to drive the compiler to merge the *map* and the *reduce* kernels. In such a case, the two kernels would be mapped to a single OpenCL kernel source. It is therefore im-

portant to define how the metadata associated to *map* and *reduce* are threated and whether it is possible to reuse pre-exiting, independent compilation outputs for *map* and *reduce*.

Merging kernels in a metadata- or content-dependent way entails the mutability of the Kernel Call Graph generated by the compiler parsing step. Currently, this graph is forced to be immutable to be able to stop the compilation pipeline as soon as the graph has been built and the OpenCL source code has been generated for each kernel node. Introducing the chance of reshaping the KCG in a later stage of the pipeline would force to execute, at least, all the stages that preceed it, potentially increasing the compilation overhead. Nonetheless, we see many transparent optimisations possibly applied to the KCG and to its content, such as utility-functions inlining, optimising data-passing and fusing kernels and/or sequential functions together. A potential refinement of our research consists in investigating the set of relevant optimisations that can be applied to the KCG built by the FSCL compiler and on a proper balance between KCG mutability and compilation/execution efficiency.

Given the simplicity of the model, the chance to give an intuitive meaning to the regression coefficients and the general prediction error obtained, the scheduling approach proposed seems extremely promising. Nevertheless, the GPU completion time of some algorithms is hard to be reliably predicted with the samples and the set of features used. It is therefore important to investigate further on indentifing a set of features that can closely describe the completion time on GPUs. Among the aspects that can influence the runtime behaviour of GPUs we see the chance of cohalesching memory accesses, the degree of potential latency-hiding[1] and the estimated number of channel/bank conflicts in accessing global/local memory.

To design GPU-specific features it is possible to exploit the results of various researches on analytical modelling GPUs.

The feature-extraction strategy we developed is flexible enough to inspect several aspects of a program with a single AST traversal. The main limitation, well-known and long-studied in the area of static code analysis, is given by the presence of branch nodes and while-loops. We are able to estimate the trip count of while-loops for some popular but restriced cases. For conditional nodes we assign equal probability to the if- and the else-branch. While assigning equal probability to the branches of a condition may have only a limited effect on the prediction of the CPU completion time, GPUs completion time is generally

[1]Latency-hiding is a GPU runtime aspect that leverages one-cycle context-switch to cover the latency of expensive operations, such as accessing global memory

deeply affected, especially when branches are divergent[2]. In fact, divergent branches are a well-known cause of performance degradation on GPUs, since when encountered, the runtime executes the if- and the else-path one after the other, properly masking the output.

A potential refinement of our feature-extraction model should focus on introducing a form of branch-prediction and extending the set of loops for which it is possible to compute the trip count.

The FSCL runtime applies a greedy scheduling strategy. Each kernel in a computing expression is scheduled on the best device of the platform, without considering where the other kernels are scheduled or how scheduling affects data transfer and copy. In addition, the FSCL runtime currently creates a different set of OpenCL resources, including contexts, for each device in the running system.

For global, optimal scheduling there are many cross-kernel aspects that must be taken into account when instantiating OpenCL resources and selecting a device for execution.

For example, if two successive kernels are respectively best-scheduled on different devices, there are mainly three choices that the runtime can take. The first option is to schedule the kernels on the each one's best device and pay the overhead of data copy. The second is to create an unique context for the two kernels, allowing data (buffers) to be shared at the price of a possibly higher overhead to allocate and access them. Finally, sub-optimal local scheduling can be applied to one of the kernels in order to guarantee that both the kernels execute on the same device (and context).

In case of concurrent execution of two, or more, kernels (e.g. e_1 and e_2 in $e_1, e_2 \mid> e_3$), a problem arises when the two kernels are best-scheduled on the same device. In this case, the device runs one OpenCL kernel after the other. The runtime may therefore choose to perform sub-optimal scheduling for one of the two kernels in order to parallelize the execution on multiple, different devices. Memory and CPU cores contention are another aspect to consider to improve overall performances [29]. In case of concurrent execution on the CPU and on an integrated GPU, the memory system accessed is shared among the kernels. From a global perspective, scheduling kernels on a discrete GPU, if available, when some other kernels run on the CPU may contribute to speedup the computation, since CPUs and discrete GPUs have separated memory systems. Similarly, whereas the CPU seems to be the best device for certain algorithms, we should consider than the CPU cores are used to execute the whole host system, including the compiler, the code to setup OpenCL re-

[2]A divergent branch is a branch where different work-items in a single group/wavefront follow different paths

sources and to interact with the OpenCL client driver, as well as the threads launched to coordinate sub-expressions (i.e. collection and function composition). The sharing of CPU computing power between the various components of the framework and the kernels/threads is therefore another point that may be considered when scheduling kernels.

It's important to note that since our scheduling approach is entirely based on completion-time, serveral of these refinements can be easily applied. For example, the decision of whether to schedule two successive kernels on each one's best device or on the same device can check if the completion time in case of sub-optimal scheduling is lower than the best-scheduled completion time plus the time to copy data[3]. This is possible thanks to the fact that completion time is a metric. Completion times can therefore be coherently summed, subtracted and compared.

14.2 Concurrent researches

Nowadays, cloud computing is becoming extremely popular, stimulating researches on scheduling strategies for tasks/agents and on device-aware exploitation of possibly heterogeneous nodes [17, 45]. Whereas FSCL custom kernels represent a one-to-one mapping with OpenCL kernels, collection functions used to express kernels and to compose them completely hide the underlying execution layer. A potential future research may start from the abstraction provided by F# collection functions and focus on a strategy to map F# collection programming to cloud computing in order to schedule and execute collection functions as parallel cloud agents.

The completion time prediction model that we defined in our research can be considered as a function that, given an input computation, tells how convenient it is to execute the computation on a particular device.

A question that may arise is under which conditions this function is invertible. Conceptually, the inverted function tells, given the completion time on a device, the shape (i.e. the features) of the computation that has such a completion time.

We see at least two interesting researches that can start from focusing on the invertibility of the function that describes our prediction model. The first concerns deducing the behaviour/structure of a black-box program from its completion-time.

[3]Cross-device data copy overhead can be easily obtained with appropriate micro-benchmarks

The second research, which is possibly a specialization of the first one, regards tooling systems that can help the developers to shape and optimize their code. In the last couple of years, many efforts have been spent on techniques to instrument GPU code and to provide the programmers of a reliable set of tools to analyze, profile and optimize heterogeneous, parallel programs [4, 24, 56]. A function that correlates a specific completion time to a set of program features may be succesfully employed to drive static and dynamic optimisations. This is particularly true considering the expressiveness of regression coefficients (section 11.2.5), each of which can be interpreted as the impact that a specific feature has on the completion time. Therefore, it would be interesting to investigate on a way to suggest or to automatically apply code optimisations on the basis of such coefficients, in order to lower the completion time or another target metric, such as energy consumption.

In the model proposed for scheduling, completion time is the quantity to predict in order to decide where to run a kernel. Nevertheless, the model is enough generic to allow to use any other meaningful metric in place of completion time, such as energy consumption. Concurrent researches may therefore employ the prediction and scheduling model resulting from our research and apply it to predict the energy consumption of parallel computations, possibly scheduling each computation on the most energy-efficient device.

Not only the scheduling model is flexible enough to be used to schedule computations on the basis of some other metrics different from completion time, but the FSCL runtime configuration infrastructure allows to employ multiple scheduling models and policies. The developer can choose a different policy for each computing expression to execute and/or develop custom schedulers and plug them into the framework. Future researches may investigate the possibilities offered by dynamic, hybrid scheduling policies, focusing on the potential applications and on the required degree of scheduling-control (i.e. per-computing-expression versus per-kernel policy). We find in cloud computing a particularly oustanding area where hybrid scheduling policies may provide noticeable benefits. From the cloud users point of view, the cost of cloud computing is generally expressed on a time basis: the longer it takes for a computation to complete the higher the price paid. To the contrary, service providers account the cost of processing in the cloud on a power-consumption basis. Future works may inspect this dicotomy and investigate on a way to apply different scheduling policies in a context-dependent fashion. For example, computations could be characterized by specific priorities, possibly deducing the priority from the context where a compution is declared (e.g. library, namespace) or from the frequency of usage. A completion-time-based scheduling policy would consequently apply to high-priority computations, while low-

priority tasks would be assigned to each one's most power-efficient device/node.

Appendices

Appendix A

Definitions of the elements of the kernel language

Definition 25. [FSCL parallel computing expression] Given an expression e, we say that e is a parallel sub-expression ($parsub(e)$) if and only if one of the following conditions is satisfied:

- e is a custom or collection kernel
- e is a function composition with arguments $e_1,\ e_2,\ ..\ e_n$ such that $\exists e' \in \{e_1,\ e_2,\ ..\ e_n\}(m) : parsub(e')$
- e is a collection composition with operators $e_1,\ e_2,\ ..\ e_n$ such that $\exists e' \in \{e_1,\ e_2,\ ..\ e_n\}(m) : parsub(e')$
- e is a computing expression wrapper in the form $fun\ pars\ ->\ body$ and $parsub(body)$

Definition 26. [FSCL custom kernel] A custom kernel is either a module function or instance/static method marked with the *Kernel* attribute or a lambda containing a parameter of type *WorkItemInfo*. Given a function/method-/lambda m, let $attrs(m)$ be the set of custom attributes associated to m. Given an attribute a, let $typeof(a)$ be the type of a. Given a lambda l, let $pars(l)$ be the set of parameters and $typeof(p)$ the type of a parameter p. An expression e is a kernel if and only if one of the following conditions is satisfied:

- e is a function/method reference, m is the *MethodInfo* of the function/method called and $\exists a' \in attrs(m) : typeof(a') = Kernel$
- e is a lambda and $\exists p' \in pars(e) : typeof(p') = WorkItemInfo$

Definition 27. [FSCL collection kernel] Given a collection function f with operators $e_1,\ e_2,\ ..\ e_n$, we say that f is a collection kernel if and only if $\forall e \in \{e_1,\ e_2,\ ..\ e_n\}$: $not\ parsub(e)$ (no operator is a parallel computing expression)

Definition 28. [FSCL sequential function] Given a function f, we say that f is a sequential function if it is neither a custom kernel nor a collection kernel

Definition 29. [FSCL function composition] Given four computing expressions e_1, e_2, e_3 and e_4, the following expressions are valid computing expressions resulting from function composition:

- $e_1 \mid> e_2$ and $e_2(e_1)$
- $(e_1, e_2) \mid\mid> e_3$ and $e_3(e_1, e_2)$
- $(e_1, e_2, e_3) \mid\mid\mid> e_4$ and $e_4(e_1, e_2, e_3)$

Definition 30. [FSCL collection composition] Given a collection function f with operators $e_1,\ e_2,\ ..\ e_n$, we say that f is a collection composition if and only if $\exists e \in \{e_1,\ e_2,\ ..\ e_n\}$: $parsub(e)$ (at least one operator is a parallel computing expression)

Definition 31. [FSCL computing expression wrapper] Given an expression e, we say that e is a computing expression wrapper ($wrapper(e)$) if and only if e is a lambda in the form $fun\ pars\ ->\ body$ where $body$ is a computing expression.

Definition 32. [FSCL computing element] A computing element is either a collection kernel, a custom kernel or a sequential function

Definition 33. [FSCL computing expression] A computing expression is either the application of a computing element (kernel/function call) or the application of a composition (function/collection composition)

Definition 34. [FSCL computing program] A computing program is an user-defined function that executes one or more computing expressions. If e is a computing expression, then $<@\ e\ @>.Run()$ represents the execution of e.

Appendix B

Definitions of equivalence of kernels and metadata

B.1 Equivalence of metadata

Definition (Dynamic metadata)**.** Given m an abstract meta-information, $p_1, p_2, \ .. \ p_n$ its properties of type $t_1, t_2, , \ .. \ t_n$, a dynamic metadata is defined as a tuple formed by:

- A CLR custom attribute (object inheriting from *System.Attribute*) of type A with properties of type $t_1, t_2, \ .. \ t_n$;
- A function, called metadata-function, of type $f : t_1 * t_2 * \ .. \ t_n * t_{wrap} \rightarrow t_{wrap}$, where:
  - t_{wrap} is an arbitrary type
  - Given an instance el of type t and n arguments $a_1, a_2, \ .. \ a_n$, $f(a_1, a_2, \ .. \ a_n, el) = el$. In other terms, the partial application of f to $a_1, a_2, \ .. \ a_n$ results in the identity function.

Definition (Uniqueness of dynamic metadata target type)**.** Given a dynamic metadata m, the set of target types $T = \{Kernel, \ Parameter, \ ReturnType\}$ and two targets t_1, t_2, we indicate with $type(t_1) \in T$ ($type(t_2) \in T$) the type of t_1 (t_2). Uniqueness of dynamic metadata target type states that m can be associated to both t_1 and t_2 if and only if $type(t_1) = type(t_2)$

Definition (Dynamic metadata disjointness by type and target)**.** Given two dynamic metadata values m_1 and m_2, with M_1 the type of m_1 and M_2 the type of m_2, and a target t instance of $T \in \{Kernel, \ Parameter, \ ReturnType\}$, if both m_1 and m_2 are associated to t then $M_1 \neq M_2$

Definition (Metadata equivalence)**.** Let M be a metadata type (type inheriting from the built-in *Attribute* type) and $MC = \{mc_1, mc_2, \ ..mc_n\}$ a given set of metadata comparers for M. For each pair of metadata values m_1, m_2 where $typeof(m_1) = typeof(m_2) = M$, m_1 is equivalent to m_2 under the set MC ($m_1 \equiv_{MC} m_2$) if and only if:

$$\forall mc \in MC : mc(m_1, m_2)$$

Definition (Complete set of metadata values)**.** Given a set of metadata types $M = \{m_1, m_2, \ .. \ m_n\}$ and a set of metadata values $MV = \{mv_1, mv_2, \ .. \ mv_m\}$, we indicate with $default(m)$ the default value of a metadata type m. We define complete set of metadata values for M under the set of comparers MV the set $MV_{[M]}$ where:

- $\forall m \in M : \exists mv \in MV : typeof(mv) = m \rightarrow mv \in MV_{[M]}$
- $\forall m \in M : \nexists mv \in MV : typeof(mv) = m \rightarrow default(m) \in MV_{[M]}$

Definition (Equivalence of sets of metadata values)**.** Let P be a pipeline, M the set of metadata types used by its components and MC the set of metadata comparers. Let $\{KM, RM, PM\}$ be the per-target-type metadata partition of M.

Let K be a kernel and $Pars$ its set of parameters.

Given MV_1, MV_2 two sets of metadata values, let $\{KMV_1, RMV_1, PMV_1\}$ and $\{KMV_2, RMV_2, PMV_2\}$ the respective per-target metadata-value partitions of $MV_{1[M]}$ and $MV_{2[M]}$ (complete sets of metadata values).

We say that MV_1 is equivalent to MV_2 for the pipeline P ($MV_1 \equiv_P MV_2$) if and only if all the following conditions are satisfied:

- $\forall m_1 \in KMV_1, m_2 \in KMV_2 : M = typeof(m_1) = typeof(m_2) \rightarrow m_1 \equiv_{MC(M)} m_2$
- $\forall m_1 \in RMV_1, m_2 \in RMV_2 : M = typeof(m_1) = typeof(m_2) \rightarrow m_1 \equiv_{MC(M)} m_2$
- $\forall p \in Pars : \forall m_1 \in PMV_1(p), m_2 \in PMV_2(p) : M = typeof(m_1) = typeof(m_2) \rightarrow m_1 \equiv_{MC(M)} m_2$

B.2 Equivalence of kernels

Definition (Node structural equivalence under alpha conversion)**.** Given two AST nodes n_1 and n_2, they are structurally equivalent under alpha conversion if and only if one of the following conditions is satisfied.

- $n_1 = Let(var_1, val_1, body_1)$, $n_2 = Let(var_2, val_2, body_2)$, var_1 and var_2 are of the same type, $val_1 \equiv_{struct} val_2$ and $body_{2[var_1/var_2]} \equiv_{struct} body_1$
- $n_1 = For(var_1, st_1, en_1, incr_1, body_1)$, $n_2 = For(var_2, st_2, en_2, incr_2, body_2)$, var_1 and var_2 are of the same type, $st_1 \equiv_{struct} st_2$, $en_1 \equiv_{struct} en_2$, $incr_1 \equiv_{struct} incr_2$ and $body_{2[var_1/var_2]} \equiv_{struct} body_1$
- $n_1 = n_2$

Definition (AST structural equivalence under alpha conversion)**.** Given two ASTs with n_1 and n_2 the respective root nodes, they are structurally equivalent under alpha conversion if only if $n_1 \equiv_{struct} n_2$.

Definition (Function equivalence)**.** Given two module functions, instance/static methods or lambdas f_1 and f_2, they are considered equivalent if and only if one of the following conditions is satisfied.

- f_1 and f_2 are modules functions or istance/static methods, M_1/M_2 is the *MethodInfo* associated to f_1/f_2 and $M_1 = M_2$
- f_1 and f_2 are lambdas and $f_1 \equiv_{struct} f_2$

Definition (Kernel equivalence)**.** Given two kernels k_1 and k_2, from the compilation point of view they are considered equivalent if and only if one of the following conditions is satisfied.

- k_1 and k_2 are custom kernels or lambdas and $k_1 \equiv_{fun} k_2$
- k_1 and k_2 are collection kernels, $F_1 = \{f1_1, ..f1_n\}$ and $F_2 = \{f2_1, ..f2_n\}$ are the respective sets of operators (arguments) to apply, $k_1 \equiv_{fun} k_2$ and $\forall_{i=1\ ..\ n} f1_i \equiv_{fun} f2_i$

B.3 Equivalence of Kernel Modules

Definition (Kernel compilation invariance)**.** Given a kernel-compilation pipeline P and, two input kernels k_1 and k_2, the respective sets MV_1 and MV_2 of metadata-values, we say that P is invariant to the transformation of input $(k_1, MV_1) \leftrightarrow (k_2, MV_2)$ if and only if the following conditions are satisfied:

- $k_1 \equiv_{ker} k_2$
- $MV_1 \equiv_P MV_2$

Definition (Kernel Module equivalence)**.** Given a kernel compilation pipeline P and two input kernels k_1 and k_2, let km_1 and km_2 the Kernel Modules resulting from respectively parsing k_1 and k_2. Let kid_1, kid_2, MV_1 and MV_2 the identifiers (MethodInfo or AST node) of the two kernels and the sets of metadata values contained in the respective Kernel Modules. We say that km_1 is equivalent to km_2 for the pipeline P ($km_1 \equiv_P km_2$) if and only if P is invariant to the transformation $(kid_1, MV_1) \rightarrow (kid_2, MV_2)$

Appendix C

Source code of samples used in language validation

C.1 Black-Scholes

C.1.1 FSCL

```
[<ReflectedDefinition>]
let cnd(d:float) =
    let A1 = 0.31938153
    let A2 = -0.356563782
    let A3 = 1.781477937
    let A4 = -1.821255978
    let A5 = 1.330274429
    let RSQRT2PI = 0.39894228040143267793994605993438
    let K = 1.0 / (1.0 + 0.2316419 * (fabs(d)))

    let nd = RSQRT2PI * Math.Exp(-0.5 * d * d) * (K * (A1 + K * (A2 + K * (A3 + K * (A4 + K * A5))
      )))
    if d > 0.0 then
        1.0 - nd;
    else
      nd

[<ReflectedDefinition>]
let blackScholes R V item =
    let S,X,T = item
    let sqrtT = Math.Sqrt(T)
    let d1 = (Math.Log(S / X) + (R + 0.5 * V * V) * T) / (V * sqrtT)
    let d2 = d1 - V * sqrtT
    let cndD1 = cnd(d1)
    let cndD2 = cnd(d2)
    let expRT = Math.Exp(-R * T)
    (S * cndD1 - X * expRT * cndD2,
     X * expRT * (1.0 - cndD2) - S * (1.0 - cndD1))

// Alloc and init data
let rnd = new Random()
let hSXT = Array.create 1024
            rnd.NextDouble() * (25.0) + 5.0,
            rnd.NextDouble() * (99.0) + 1.0,
            rnd.NextDouble() * (9.75) + 0.25)
```

```
let R, V = 0.02, 0.30

// Run BlackScholes
let result =
 <@
   hSXT |>
    Array.map(blackScholes R V)
 @>.Run()
```

Listing C.1: Black-Scholes in FSCL

C.1.2 Aparapi

```
// Avoid using custom struct, seems not supported or unstable
public static class BlackScholesKernel extends Kernel {
    final int size;
    final float[] s;
    final float[] x;
    final float[] t;
    final float[] result;

    final float R = 0.02f;
    final float V = 0.30f;

    public BlackScholesKernel(float[] hs, float[] hx, float[] ht) {
        s = hs;
        x = hx;
        t = ht;
        size = hs.length;
        result = new float[size * 2];
    }

    float cnd(float d) {
        float A1 = 0.31938153f;
        float A2 = -0.356563782f;
        float A3 = 1.781477937f;
        float A4 = -1.821255978f;
        float A5 = 1.330274429f;
        float RSQRT2PI = 0.39894228040143267793994605993438f;
        float K = 1.0f / (1.0f + 0.2316419f * (abs(d)));

        float nd = RSQRT2PI * exp(-0.5f * d * d) * (K * (A1 + K * (A2 + K * (A3 + K * (A4 + K * A5
          )))));
        if (d > 0.0f)
            nd = 1.0f - nd;
        return nd;
    }

    void blackScholes(float R, float V, float S, float X, float T, float[] outData, int idx) {
        float sqrtT = sqrt(T);
        float d1 = (log(S / X) + (R + 0.5f * V * V) * T) / (V * sqrtT);
        float d2 = d1 - V * sqrtT;
        float cndD1 = cnd(d1);
        float cndD2 = cnd(d2);
        float expRT = exp(-R * T);
        outData[idx] = S * cndD1 - X * expRT * cndD2;
        outData[idx + 1] = X * expRT * (1.0f - cndD2) - S * (1.0f - cndD1);
    }

    @Override public void run() {
        int gid = getGlobalId(0);
        blackScholes(R, V, s[gid], x[gid], t[gid], result[gid]);
```

```
    }

    public float[] getResults() {
        return result;
    }
}

public static void runBlackScholes() {
    int size = 1024;

    // Prepare data
    float[] s = new float[size];
    float[] x = new float[size];
    float[] t = new float[size];
    for (int i = 0; i < size; i++) {
        s[i] = (float) (Math.random() * 25.0 + 5.0);
        x[i] = (float) (Math.random() * 99.0 + 1.0);
        t[i] = (float) (Math.random() * 9.75 + 0.25);
    }

    // Compute black scholes
    BlackScholesKernel k1 = new BlackScholesKernel(s, x, t);
    k1.execute(Range.create(Device.firstGPU(), size, 128));
    float[] results = k1.getResults();
    k1.dispose();
}
```

Listing C.2: Black-Scholes in Aparapi

C.1.3 Dandelion

```
// We assume Dandelion support custom structs
struct SXTRV {
    public float s;
    public float x;
    public float t;

    public SXT(float hs, float hx, float ht)
    {
        s = hs;
        x = hx;
        t = ht;
    }
}

float cnd(float d) {
    float A1 = 0.31938153f;
    float A2 = -0.356563782f;
    float A3 = 1.781477937f;
    float A4 = -1.821255978f;
    float A5 = 1.330274429f;
    float RSQRT2PI = 0.39894228040143267793994605993438f;
    float K = 1.0f / (1.0f + 0.2316419f * ((d < 0? -d : d)));

    float nd = RSQRT2PI * (float)Math.Exp(-0.5f * d * d) * (K * (A1 + K * (A2 + K * (A3 + K * (A4
       + K * A5)))));
    if(d > 0.0f)
        return 1.0f - nd;
    return nd;
}

Vector blackScholes(SXT item, float R, float V) {
```

```
    float S = item.s;
    float X = item.x;
    float T = item.t;
    float sqrtT = (float)Math.Sqrt(T);
    float d1 = (float)(Math.Log(S / X) + (R + 0.5f * V * V) * T) / (V * sqrtT);
    float d2 = d1 - V * sqrtT;
    float cndD1 = cnd(d1);
    float cndD2 = cnd(d2);
    float expRT = (float)Math.Exp(-R * T);
    return new Vector(S * cndD1 - X * expRT * cndD2,
     X * expRT * (1.0f - cndD2) - S * (1.0f - cndD1));
}

IQueryable<Vector> blackScholesKernel(IQueryable<SXT> data, float R, float V) {
    return data.Select(item => blackScholes(item, R, V));
}

void runBlackScholes() {
    int size = 1024;
    Random rnd = new Random();

    // Prepare data
    SXT hSXT = new SXT[size];
    for(int i = 0; i < size; i++) {
        hSXTRV[i] = new SXT(
            (float)rnd.NextDouble() * 25.0f + 5.0,
            (float)rnd.NextDouble() * 99.0f + 1.0f,
            (float)rnd.NextDouble() * 9.75f + 0.25f);
    }
    float R = 0.02f;
    float V = 0.30f;

    // Run BlackScholes
    var result = blackScholesKernel(hSXT.AsDandelion(), R, V);
 }
```

Listing C.3: Black-Scholes in Dandelion

C.2 K-Means

C.2.1 FSCL

```
[<ReflectedDefinition>]
let nearestCenter (centers: (float * float)[]) (point: float * float) =
    let mutable minIndex = 0
    let mutable minValue = Double.MaxValue
    let x,y = point
    for curIndex = 0 to centers.Length - 1 do
        let cx,cy = centers.[curIndex]
        let curValue = Math.Sqrt(Math.Pow(x - cx, 2.0) + Math.Pow(y - cy, 2.0))
        if curIndex = 0 || minValue > curValue then
            minValue <- curValue
            minIndex <- curIndex
    minIndex

// Prepare data
let rnd = new Random()
let k = 3
let points = Array.init 1024 (fun i -> (rnd.NextDouble() * 5.0, rnd.NextDouble() * 3.0))
let centers = Array.init k (fun i -> points.[i])
```

```
// Run kmeans
let kmeans =
 <@
    points |>
    Array.groupBy(fun i -> nearestCenter centers i) |>
    Array.map (fun (key, data) ->
                data |>
                Array.reduce(fun (cx,cy) (x,y) -> (cx + x, cy + y)) |>
                (fun (a,b) -> (a/(double)(Seq.length data), b/(double)(Seq.length data))))
 @>.Run()
```

Listing C.4: K-Means in FSCL

C.2.2 Aparapi

```
public static final class Point {
    public float x;
    public float y;
}

public static class NearestCenterKernel extends Kernel {
    final Point[] points;
    final Point[] centers;
    final int[] association;
    final int size;

    public NearestCenterKernel(Point[] p, Point[] c) {
        points = p;
        centers = c;
        association = new int[p.length];
        size = c.length;
    }

    public int nearestCenter (Point[] centers, Point p, int size) {
        int minIndex = -1;
        float minValue = 0;
        for(int curIndex = 0; curIndex < size; curIndex++) {
            Point c = centers[curIndex];
            float curValue = (float)sqrt(pow(p.x - c.x, 2.0) + pow(p.y - c.y, 2.0));
            if(minIndex < 0 || minValue > curValue) {
                minValue = curValue;
                minIndex = curIndex;
            }
        }
        return minIndex;
    }

    @Override public void run() {
        int i = getGlobalId(0);
        int cIndex = nearestCenter(centers, points[i], size);
        association[i] = cIndex;
    }

    public int[] getAssociations() {
        return association;
    }
}

public static class GroupByCenterKernel extends Kernel {
    final Point[] points;
    final int[] association;
```

```
    final Point[] sortPoints;
    final int[] count;

    public GroupByCenterKernel(Point[] p, int[] a) {
        points = p;
        association = a;
        count = new int[p.length];
        sortPoints = new Point[p.length];
    }

    @Override public void run() {
        int i = getGlobalId(0);
        int n = getGlobalSize(0);

        // Key is the association
        int a = association[i];

        int pos = 0;
        int found = 0;
        int j = 0;
        while(found == 0 && j < n)
        {
            float curA = association[j];
            if((a > curA) || ((a == curA) && (j < i))) {
                found = 1;
            }
            else {
                pos++;
            }
            j++;
        }
        sortPoints[pos].x = points[i].x;
        sortPoints[pos].y = points[i].y;
        // Incr set count
        atomicAdd(count, a, 1);
    }

    public int[] getCounts() {
        return count;
    }
    public Point[] getSortPoints() {
        return sortPoints;
    }
}

public static class CentroidKernel extends Kernel {
    final Point[] sortPoints;
    final int[] count;
    final Point[] centroids;

    public CentroidKernel(Point[] s, int[] c, int totalCount) {
        count = c;
        sortPoints = s;
        centroids = new Point[totalCount];
    }

    @Override public void run() {
        // Each thread perform linear reduction (avg) for a group
        int i = getGlobalId(0);
        int startIdx = 0;
        for(int j = 0; j < i; j++)
            startIdx += count[j];
        int endIdx = startIdx + count[i] - 1;

        float newX = 0.0f;
        float newY = 0.0f;
        for(int j = startIdx; j <= endIdx; j++) {
            newX += sortPoints[j].x;
            newY += sortPoints[j].y;
```

```
        }
        centroids[i].x = newX/(float)count[i];
        centroids[i].y = newY/(float)count[i];
    }

    public Point[] getCentroid() {
        return centroids;
    }
}

public static void runKMeans() {
    int size = 1024;
    int cCount = 3;

    // Prepare data
    final Point[] points = new Point[size];
    final Point[] centers = new Point[cCount];
    for (int i = 0; i < size; i++) {
        points[i] = new Point();
        points[i].x = (float) (Math.random() * 100);
        points[i].y = (float) (Math.random() * 100);
    }
    for (int i = 0; i < cCount; i++) {
        centers[i] = points[i];
    }

    // Compute nearest centers
    NearestCenterKernel k1 = new NearestCenterKernel(points, centers);
    k1.execute(Range.create(Device.firstGPU(), size, 128));

    // Now group by center
    GroupByCenterKernel k2 = new GroupByCenterKernel(points, k1.getAssociations());
    k2.execute(Range.create(Device.firstGPU(), size, 128));

    // Sequentially compute total count
    int totalCount = 0;
    for(int i = 0; i < k2.getCounts().length; i++)
        totalCount += k2.getCounts()[i];

    // Finally, recompute centroid per-group
    CentroidKernel k3 = new CentroidKernel(k2.getSortPoints(), k2.getCounts(), totalCount);
    k3.execute(Range.create(Device.firstGPU(), totalCount, 1));

    k1.dispose();
    k2.dispose();
    k3.dispose();
}
```

Listing C.5: K-Means in Aparapi

C.2.3 Dandelion

```
int NearestCenter(Vector vector, IEnumerable<Vector> centers) {
    int minIndex = 0;
    double minValue = Double.MaxValue;
    int curIndex = 0;
    foreach (Vector center in centers) {
        double curValue = (center - vector).Norm2();
        if (minValue > curValue) {
            minValue = curValue;
            minIndex = curIndex;
        }
```

```
        curIndex++;
    }
    return minIndex;
}

IQueryable<Vector> OneStep(IQueryable<Vector> vectors, IQueryable<Vector> centers) {
    return vectors.GroupBy(v => NearestCenter(v, centers)).Select(g => g.Aggregate((x, y) => x+y)/
      g.Count());
}

void runKMeans() {
    Random rnd = new Random();
    int k = 3;

    // Prepare data
    Vector[] points = new Vector[1024];
    Vector[] centers = new Vector[3];
    for(int i = 0 ; i < points.Length; i++) {
        points[i] = new Vector(rnd.NextDouble() * 5.0, rnd.NextDouble() * 3.0);
    }
    for(int i = 0 ; i < centers.Length; i++) {
        centers[i] = points[i];
    }

    // Run computation
    OneStep(points.AsDandelion(), centers.AsDandelion());
}
```

Listing C.6: K-Means in Dandelion

C.3 Tiled matrix multiplication

C.3.1 FSCL

```
[<ReflectedDefinition; Kernel>]
let MatMul(matA: float32[,], matB: float32[,], matC: float32[,], wi: WorkItemInfo) =
    let bx = wi.GroupID(0)
    let by = wi.GroupID(1)

    let tx = wi.LocalID(0)
    let ty = wi.LocalID(1)

    let block_size = wi.WorkSize(0)
    let matAWidth = matA.GetLength(1)
    let matBWidth = matB.GetLength(1)

    let aBegin = block_size * by
    let aEnd = aBegin + matAWidth - 1
    let bBegin = block_size * bx

    let mutable b = bBegin
    let mutable Csub = 0.0f

    let As = local(Array2D.zeroCreate<float32> block_size block_size)
    let Bs = local(Array2D.zeroCreate<float32> block_size block_size)

    for a in aBegin .. block_size .. aEnd do
        As.[ty, tx] <- matA.[ty, a + tx]
        Bs.[ty, tx] <- matB.[b + ty, tx]
```

```
        wi.Barrier(CLK_LOCAL_MEM_FENCE)

        // Multiply the two matrices together
        for k = 0 to block_size - 1 do
            Csub <- Csub + (As.[ty, k] * Bs.[k, tx])
        wi.Barrier(CLK_LOCAL_MEM_FENCE)

        b <- b + block_size

    matC.[block_size * by + ty, block_size * bx + tx] <- Csub

// Prepare data
let rnd = new Random()
let a = Array2D.init 1024 1024 (fun i -> (float32)(rnd.NextDouble()))
let b = Array2D.init 1024 1024 (fun i -> (float32)(rnd.NextDouble()))
let c = Array2D.zeroCreate<float32> 1024 1024
let workSize = WorkSize([| 1024L; 1024L |], [| 16L; 16L |])

// Run matrix multiplication
<@
    MatMul(a, b, c, workSize)
@>.Run()
```

Listing C.7: Tiled matrix multiplication in FSCL

C.3.2 Aparapi

```
public void MatMul() {
   final int size = 1024;
    final int BLOCK_SIZE = 16;

    // Prepare data
    final float[] matA = new float[size * size];
    final float[] matB = new float[size * size];
    final float[] matC = new float[size * size];

    final float[] As_$local$ = new float[BLOCK_SIZE * BLOCK_SIZE];
    final float[] Bs_$local$ = new float[BLOCK_SIZE * BLOCK_SIZE];

    for (int i = 0; i < size * size; i++) {
        matA[i] = (float) (Math.random());
        matB[i] = (float) (Math.random());
    }

    // Define the kernel
    Kernel kernel = new Kernel(){
        @Override public void run() {
            int bx = getGroupId(0);
            int by = getGroupId(1);

            int tx = getLocalId(0);
            int ty = getLocalId(1);

            int aBegin = size * BLOCK_SIZE * by;
            int aEnd   = aBegin + size - 1;

            int aStep  = BLOCK_SIZE;
            int bBegin = BLOCK_SIZE * bx;
            int bStep  = BLOCK_SIZE * size;

            int a = aBegin;
            int b = bBegin;
```

```
            float Csub = 0.0f;

            while(a <= aEnd) {
                As_$local$[ty * BLOCK_SIZE + tx] = matA[a + (size * ty) + tx];
                Bs_$local$[ty * BLOCK_SIZE + tx] = matB[b + (size * ty) + tx];
                localBarrier();

                for(int k = 0; k < BLOCK_SIZE; k++) {
                    Csub = Csub + (As_$local$[ty * BLOCK_SIZE + k] * Bs_$local$[k * BLOCK_SIZE +
                      tx]);
                }
                localBarrier();

                b = b + bStep;
                a = a + aStep;
            }

            int c = (size * BLOCK_SIZE * by) + (BLOCK_SIZE * bx);
            matC[c + (size * ty) + tx] = Csub;
        }
    };

    // Run kernel on the first GPU device
    kernel.execute(Range.create2D(Device.firstGPU(), size, size, 16, 16));
    kernel.dispose();
}
```

Listing C.8: Tiled matrix multiplication in Aparapi

C.3.3 Dandelion

No Dandelion library is available yet to test. Since Dandelion seems to lack support for user-defined kernels, tiled matrix multiplication may result impossible to express.

C.4 Average image complexity

C.4.1 FSCL

```
[<ReflectedDefinition, Kernel>]
let SobelFilter2D (wi: WorkItemInfo) (inIm: float32[,]) =
    // Create output image using FSCL kernel return capability
    let outIm = Array2D.zeroCreate<float32> (inIm.GetLength(0) - 2) (inIm.GetLength(1) - 2)

    // Work-item computation
    let x = wi.GlobalID(0)
    let y = wi.GlobalID(1)
    let width = outIm.GetLength(1)
    let height = outIm.GetLength(0)
    let mutable Gx = 0.0f
    let mutable Gy = Gx

    if x < width && y < height then
        // Read each texel component and calculate the filtered value using neighbouring texel
          components
        Gx <- inIm.[y, x] + 2.0f * inIm.[y, x + 1] + inIm.[y, x + 2] -
```

```
                inIm.[y + 2, x] - 2.0f * inIm.[y + 2, x + 1] - inIm.[y + 2, x + 2]
            Gy <- inIm.[y, x] - inIm.[y, x + 2] + 2.0f * inIm.[y + 1, x] -
                2.0f * inIm.[y + 1, x + 2] + inIm.[y + 2, x] - inIm.[y + 2, x + 2]
            outIm.[y, x] <- Math.Sqrt(Gx * Gx + Gy * Gy)/2.0f
        // Return
        outIm

// Prepare data
let image = // Load image into an Array2D<float4> instance
let threshold = 0.8f

// Run noise calculation
let avgComplex =
    <@
        image |>
        // To black-and-white
        Array2D.map(fun p -> (0.2126f * (float32)p.x + 0.7152f * (float32)p.y + 0.0722f * (float32
           )p.z)) |>
        // Sobel
        SobelFilter2D wi |>
        // Count pixels over the white threshold
        Array2D.averageBy (fun it ->
                                if it > threshold then
                                    1.0f
                                else
                                    0.0f)
    @>.Run()
```

Listing C.9: Average image complexity in FSCL

C.4.2 Aparapi

```
public static final class RGBA4Pixel {
    public float r;
    public float g;
    public float b;
    public float a;
}

public static class SobelBWFilterKernel extends Kernel {
    final RGBA4Pixel[] input;
    final float[] output;
    final int width;
    final int height;

    public SobelBWFilterKernel(RGBA4Pixel[] i, int w, int h) {
        input = i;
        output = new float[(w - 2) * (h - 2)];
        width = w;
        height = h;
    }

    float toBw(RGBA4Pixel p) {
        return 0.2126f * p.r + 0.7152f * p.g + 0.0722f * p.b;
    }

    @Override public void run() {
        int x = getGlobalId(0);
        int y = getGlobalId(1);
        int outWidth = getGlobalSize(0);
        int outHeight = getGlobalSize(1);
```

```
        float Gx = 0.0f;
        float Gy = Gx;

        if(x < outWidth && y < outHeight) {
            Gx = toBw(input[(y * outHeight) + x]) + 2.0f * toBw(input[(y * outHeight) + x + 1]) +
               toBw(input[(y * outHeight) + x + 2]) -
            toBw(input[((y + 2) * outHeight) + x]) - 2.0f * toBw(input[((y + 2) * outHeight) + x +
                1]) - toBw(input[((y + 2) * outHeight) + (x + 2)]);

            Gy = toBw(input[(y * outHeight) + x]) - toBw(input[(y * outHeight) + x + 2]) + 2.0f *
               toBw(input[((y + 1) * outHeight) + x]) -
            2.0f * toBw(input[((y + 1) * outHeight) + x + 2]) + toBw(input[((y + 2) * outHeight) +
                x]) - toBw(input[((y + 2) * outHeight) + x + 2]);

            output[y * outHeight + x] = sqrt(Gx * Gx + Gy * Gy)/2.0f;
        }
    }

    public float[] getResults() {
        return output;
    }
}

public static class ReduceKernel extends Kernel {
    final float[] input;
    final float[] output;
    final float threshold = 0.8f;
    final int slicePerThread;

    public ReduceKernel(float[] i, int slice) {
        input = i;
        output = new float[i.length / slice];
        slicePerThread = slice;
    }

    @Override public void run() {
        // Serial reduction, each thread reduces an input slice
        int gid = getGlobalId(0);
        if(gid * slicePerThread < input.length) {
            float accum = 0.0f;
            for(int i = gid * slicePerThread; i < min((gid + 1) * slicePerThread, input.length); i
               ++) {
                accum += input[i] >= threshold? 1.0f : 0.0f;
            }
            output[gid] = accum;
        }
    }

    public float[] getResults() {
        return output;
    }
}

public static void runAvgComplex() {
    int size = 1026;
    int reduceSlice = 128;

    final RGBA4Pixel[] image = // Load image into an RGBA4Pixel[] instance

    // Sobel filter
    SobelBWFilterKernel k1 = new SobelBWFilterKernel(image, size, size);
    k1.execute(Range.create2D(Device.firstGPU(), size, size, 128, 128));

    // Now do reduce to (partially) count border pixels
    ReduceKernel k2 = new ReduceKernel(k1.getResults(), reduceSlice);
    k2.execute(Range.create(Device.firstGPU(), k1.getResults().length / reduceSlice, 1));

    // Finish reduction
    float[] partialRed = k2.getResults();
```

```
    float count = 0.0f;
    for(int i = 0; i < partialRed.length; i++)
        count += partialRed[i];

    // count is the avg complexity
    // ...

    // Dispose
    k1.dispose();
    k2.dispose();
}
```

Listing C.10: Average image complexity in Aparapi

C.4.3 Dandelion

No Dandelion library is available yet to test. Since Dandelion seems to lack support for user-defined kernels, average image complexity via Sobel filtering may result impossible to express in an optimal form like in FSCL or Aparapi. Sobel algorithm could be instead expressed through a combination of *Take*, *Skip* and other filtering operators, which nonetheless causes a computation partitioning (including scheduling and execution) that is far from any common implementation parallel Sobel filtering.

Bibliography

[1] Cloo (OpenCL Object Oriented) project, 2010.

[2] Yuki Abe, Hiroshi Sasaki, Martin Peres, Koji Inoue, Kazuaki Murakami, and Shinpei Kato. Power and Performance Analysis of GPU-Accelerated Systems. In *Presented as part of the 2012 Workshop on Power-Aware Computing and Systems*, Hollywood, CA, 2012. USENIX.

[3] AMD. HSA Foundation Overview, 2011.

[4] AMD. AMD CodeXL webpage, 2013.

[5] AMD. Aparapi framework homepage, 2014.

[6] Francis J. Anscombe. Graphs in statistical analysis. *The American Statistician*, 27(1):17–21, 1973.

[7] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, Katherine A. Yelick, Meetings Jim Demmel, William Plishker, John Shalf, Samuel Williams, and Katherine Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical report, TECHNICAL REPORT, UC BERKELEY, 2006.

[8] Andreas Berl, Erol Gelenbe, Marco Di Girolamo, Giovanni Giuliani, Hermann De Meer, Minh Quan Dang, and Kostas Pentikousis. Energy-Efficient Cloud Computing. *Comput. J.*, 53(7):1045–1051, September 2010.

[9] Angela Blanco-Fernndez, Ana Colubi, Marta Garca-Brzana, and Manuel Montenegro. A Linear Regression Model for Interval-Valued Response Based on Set Arithmetic. In Rudolf Kruse, Michael R. Berthold, Christian Moewes, Mara ngeles Gil, Przemysaw Grzegorzewski, and Olgierd

Hryniewicz, editors, *Synergies of Soft Computing and Statistics for Intelligent Data Analysis*, number 190 in Advances in Intelligent Systems and Computing, pages 105–113. Springer Berlin Heidelberg, January 2013.

[10] M. Bogdanski, P.R. Lewis, T. Becker, and Xin Yao. Improving Scheduling Techniques in Heterogeneous Systems with Dynamic, On-Line Optimisations. In *2011 International Conference on Complex, Intelligent and Software Intensive Systems (CISIS)*, pages 496–501, June 2011.

[11] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. FlumeJava: Easy, Efficient Data-parallel Pipelines. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 363–375, New York, NY, USA, 2010. ACM.

[12] Kwang-Ting Cheng and Yi-Chu Wang. Using mobile GPU for general-purpose computing; a case study of face recognition on smartphones. In *2011 International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, pages 1–4, April 2011.

[13] David Christiansen. Dependent type providers. pages 25–34. ACM, 2013.

[14] Philipp Ciechanowicz, Michael Poldner, and Herbert Kuchen. The Mnster Skeleton Library Muesli: A comprehensive overview. ERCIS Working Paper 7, Westflsche Wilhelms-Universitt Mnster (WWU) - European Research Center for Information Systems (ERCIS), 2009.

[15] Codeplex. OpenCL.NET project on Codeplex, 2010.

[16] Stephen A. Cook. A taxonomy of problems with fast parallel algorithms. *Information and Control*, 64(13):2–22, January 1985.

[17] Steve Crago, Kyle Dunn, Patrick Eads, Lorin Hochstein, Dong-In Kang, Mikyung Kang, Devendra Modium, Karandeep Singh, Jinwoo Suh, and John Paul Walters. Heterogeneous Cloud Computing. In *Proceedings of the 2011 IEEE International Conference on Cluster Computing*, CLUSTER '11, pages 378–385, Washington, DC, USA, 2011. IEEE Computer Society.

[18] Mayank Daga. On the Efficacy of a Fused CPU+GPU Processor (or APU) for Parallel Computing. 2011.

[19] Daniel Egloff. Taming GPU Threads with F# and Alea GPU, 2014.

[20] Pablo de Oliveira Castro, Eric Petit, Asma Farjallah, and William Jalby. Adaptive sampling for performance characterization of application kernels. *Concurrency and Computation: Practice and Experience*, 25(17):2345–2362, 2013.

[21] Jiun-Hung Ding, Wei-Chung Hsu, Bai-Cheng Jeng, Shih-Hao Hung, and Yeh-Ching Chung. HSAemu: A Full System Emulator for HSA Platforms. In *Proceedings of the 2014 International Conference on Hardware/Software Codesign and System Synthesis*, CODES '14, pages 26:1–26:10, New York, NY, USA, 2014. ACM.

[22] Johan Enmyren and Christoph W. Kessler. SkePU: A Multi-backend Skeleton Programming Library for multi-GPU Systems. In *Proceedings of the Fourth International Workshop on High-level Parallel Programming and Applications*, HLPP '10, pages 5–14, New York, NY, USA, 2010. ACM.

[23] F. Farahnakian, P. Liljeberg, and J. Plosila. LiRCUP: Linear Regression Based CPU Usage Prediction Algorithm for Live Migration of Virtual Machines in Data Centers. In *2013 39th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, pages 357–364, September 2013.

[24] Naila Farooqui, Karsten Schwan, and Sudhakar Yalamanchili. Efficient Instrumentation of GPGPU Applications Using Information Flow Analysis and Symbolic Execution. In *Proceedings of Workshop on General Purpose Processing Using GPUs*, GPGPU-7, pages 19:19–19:27, New York, NY, USA, 2014. ACM.

[25] John Fox. *Applied regression analysis, linear models, and related methods.* Sage Publications, Inc, 1997.

[26] Juan Jos Fumero, Michel Steuwer, and Christophe Dubach. A Composable Array Function Interface for Heterogeneous Computing in Java. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY'14, pages 44:44–44:49, New York, NY, USA, 2014. ACM.

[27] Gabriele Cocco. FSCL framework website, 2014.

[28] Gian Ntzik. F# Streams, 2014.

[29] Chris Gregg, Jeff Brantley, and Kim Hazelwood. Contention-Aware Scheduling of Parallel Code for Heterogeneous Systems. In *2nd USENIX Workshop on Hot Topics in Parallelism*, HotPar, Berkeley, CA, June 2010.

[30] Kate Gregory and Ade Miller. *C++ AMP: Accelerated Massive Parallelism with Microsoft Visual C++*. "O'Reilly Media, Inc.", September 2012.

[31] C. Guo and N. Higham. A SchurNewton Method for the Matrix \lowercase{\boldmathp}th Root and its Inverse. *SIAM Journal on Matrix Analysis and Applications*, 28(3):788–804, January 2006.

[32] David C. Hoaglin, Frederick Mosteller, and John W. Tukey. *Exploring data tables, trends, and shapes*, volume 101. John Wiley & Sons, 2011.

[33] Eric Holk, William Byrd, Nilesh Mahajan, Jeremiah Willcock, Arun Chauhan, and Andrew Lumsdaine. *BYRD Declarative Parallel Programming for GPUs*. 2011.

[34] Paul W. Holland and Roy E. Welsch. Robust regression using iteratively reweighted least-squares. *Communications in Statistics-Theory and Methods*, 6(9):813–827, 1977.

[35] Sunpyo Hong and Hyesoon Kim. An Analytical Model for a GPU Architecture with Memory-level and Thread-level Parallelism Awareness. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 152–163, New York, NY, USA, 2009. ACM.

[36] Ching-Hsien Hsu, Shih-Chang Chen, Chih-Chun Lee, Hsi-Ya Chang, Kuan-Chou Lai, Kuan-Ching Li, and Chunming Rong. Energy-Aware Task Consolidation Technique for Cloud Computing. In *2011 IEEE Third International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 115–121, November 2011.

[37] Ling Huang, Jinzhu Jia, Bin Yu, Byung-Gon Chun, Petros Maniatis, and Mayur Naik. Predicting Execution Time of Computer Programs Using Sparse Polynomial Regression. In John D. Lafferty, Christopher K. I. Williams, John Shawe-Taylor, Richard S. Zemel, and Aron Culotta, editors, *NIPS*, pages 883–891. Curran Associates, Inc., 2010.

[38] Takashi Isobe, Eric D. Feigelson, Michael G. Akritas, and Gutti Jogesh Babu. Linear regression in astronomy. *The Astrophysical Journal*, 364:104–113, November 1990.

[39] Michael A. Iverson, Fusun Ozguner, and Lee C. Potter. Statistical prediction of task execution times through analytic benchmarking for scheduling in a heterogeneous environment. In *Heterogeneous Computing Workshop, 1999.(HCW'99) Proceedings. Eighth*, pages 99–111. IEEE, 1999.

[40] Anil K. Jain. Data Clustering: 50 Years Beyond K-means. *Pattern Recogn. Lett.*, 31(8):651–666, June 2010.

[41] Y. Jiao, H. Lin, P. Balaji, and W. Feng. Power and Performance Characterization of Computational Kernels on the GPU. In *Green Computing and Communications (GreenCom), 2010 IEEE/ACM Int'l Conference on Int'l Conference on Cyber, Physical and Social Computing (CPSCom)*, pages 221–228, December 2010.

[42] Adam Eversole Jon Currey and Christopher J. Rossbach. Scheduling Dataflow Execution Across Multiple Accelerators. 2014.

[43] P.J. Joseph, K. Vaswani, and Matthew J. Thazhuthaveetil. A Predictive Performance Model for Superscalar Processors. In *39th Annual IEEE/ACM International Symposium on Microarchitecture, 2006. MICRO-39*, pages 161–170, December 2006.

[44] Justin Holewinski. PTX Back-End: GPU Programming with LLVM, November 2011.

[45] Ketan Paranjape, Steve Hebert, and Bob Masson. Heterogeneous Computing in the Cloud: Crunching Big Data and Democratizing HPC Access for the Life Sciences. Technical report, 2014.

[46] Andreas Klckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, and Ahmed Fasih. PyCUDA and PyOpenCL: A Scripting-based Approach to GPU Run-time Code Generation. *Parallel Comput.*, 38(3):157–174, March 2012.

[47] MarketsandMarkets. Heterogeneous Mobile Processing & Computing Market by Component (processor, GPU, DSP, connectivity), Technology Node (45nm-5nm), Application (Consumer, Tele-communication, Automotive, MDA, Medical), & Geography Forecast & Analysis to 2014 2020. Technical report, 2014.

[48] Ricardo Marques, Herv Paulino, Fernando Alexandre, and Pedro D. Medeiros. Algorithmic Skeleton Framework for the Orchestration of GPU Computations. In Felix Wolf, Bernd Mohr, and Dieter an Mey, editors,

Euro-Par 2013 Parallel Processing, number 8097 in Lecture Notes in Computer Science, pages 874–885. Springer Berlin Heidelberg, January 2013.

[49] Erik Meijer, Brian Beckman, and Gavin Bierman. LINQ: Reconciling Object, Relations and XML in the .NET Framework. pages 706–706. ACM, 2006.

[50] Cedric Nugteren, Gert-Jan van den Braak, Henk Corporaal, and Henri Bal. A Detailed GPU Cache Model Based on Reuse Distance Theory. *hgpu.org*, January 2014.

[51] Bryan O'Sullivan, John Goerzen, and Don Stewart. *Real World Haskell*. O'Reilly Media, Inc., 1st edition, 2008.

[52] C. Ozturk and R. Sendag. An analysis of hard to predict branches. In *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*, pages 213–222, March 2010.

[53] P. Pandit and R. Govindarajan. Fluidic kernels: Cooperative execution of opencl programs on multiple heterogeneous devices. 2014.

[54] C.D. Panirwala. *Exploring Correlation for Indirect Branch Prediction*. North Carolina State University, 2012.

[55] Pankaj Singh. FUSION APU & TRENDS/ CHALLENGES IN FUTURE SoC DESIGN, 2011.

[56] Paulius Micikevicius. GPU Performance Analysis and Optimisation, 2012.

[57] James Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007.

[58] Christopher J. Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. Dandelion: A Compiler and Runtime for Heterogeneous Systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 49–68, New York, NY, USA, 2013. ACM.

[59] Paul E. Roundy and William M. Frank. Applications of a Multiple Linear Regression Model to the Analysis of Relationships between Eastward- and Westward-Moving Intraseasonal Modes. *Journal of the Atmospheric Sciences*, 61(24):3041–3048, December 2004.

[60] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, Daehyun Kim, and Pradeep Dubey. Fast Sort on CPUs and GPUs: A Case for Bandwidth Oblivious SIMD Sort. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 351–362, New York, NY, USA, 2010. ACM.

[61] G. T. Shrivakshan and Dr C. Chandrasekar. *A Comparison of various Edge Detection Techniques used in Image Processing.* 1986.

[62] Joshua B. Smith. *Practical OCaml (Practical).* Apress, Berkely, CA, USA, 2006.

[63] Swetha P. T. Srinivasan and Umesh Bellur. Novel Power and Completion Time Models for Virtualized Environments. *arXiv:1411.3201 [cs]*, November 2014. arXiv: 1411.3201.

[64] Michel Steuwer, Philipp Kegel, and Sergei Gorlatch. SkelCL - A Portable Skeleton Library for High-Level GPU Programming. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, volume 0, pages 1176–1182, Los Alamitos, CA, USA, 2011. IEEE Computer Society.

[65] Xueyuan Su, Garret Swart, Brian Goetz, Brian Oliver, and Paul Sandoz. Changing Engines in Midstream: A Java Stream Computational Model for Big Data Processing. *PVLDB*, 7(13):1343–1354, 2014.

[66] Alexandre Denis Raymond Namyst Marie-Christine Counilh Sylvain Henry, Denis Barthou. SOCL: An OpenCL Implementation with Automatic Multi-Device Adaptation Support. 2013.

[67] Don Syme. Leveraging .NET Meta-programming Components from F#: Integrated Queries and Interoperable Heterogeneous Execution. In *Proceedings of the 2006 Workshop on ML*, ML '06, pages 43–54, New York, NY, USA, 2006. ACM.

[68] Don Syme, Keith Battocchi, Kenji Takeda, Donna Malayeri, Jomo Fisher, Jack Hu, Tao Liu, Brian McNamara, Daniel Quirk, Matteo Taveggia, Wonseok Chae, Uladzimir Matsveyeu, and Tomas Petricek. F#3.0 - Strongly-Typed Language Support for Internet-Scale Information Sources. Technical Report MSR-TR-2012-101, Microsoft Research, September 2012.

[69] Don Syme, Adam Granicz, and Antonio Cisternino. *Expert F# 2.0.* Apress, Berkely, CA, USA, 1st edition, 2010.

[70] David Tarditi, Sidd Puri, and Jose Oglesby. Accelerator: Using Data Parallelism to Program GPUs for General-purpose Uses. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 325–335, New York, NY, USA, 2006. ACM.

[71] Ioan Trenca, Maria Miruna POCHEA, and Angela Maria FILIP. Options evaluation - Black-Scholes model vs. binomial options pricing model. *Finante - provocarile viitorului (Finance - Challenges of the Future)*, 1(12):137–146, 2010.

[72] Kenzo Van Craeynest, Aamer Jaleel, Lieven Eeckhout, Paolo Narvaez, and Joel Emer. Scheduling Heterogeneous Multi-cores Through Performance Impact Estimation (PIE). In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 213–224, Washington, DC, USA, 2012. IEEE Computer Society.

[73] Yuan Wen. Smart Multi-Task Scheduling for OpenCL Programs on CPU/GPU Heterogeneous Platforms. 2014.

[74] J.R. Wernsing and G. Stitt. A scalable performance prediction heuristic for implementation planning on heterogeneous systems. In *2010 8th IEEE Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia)*, pages 71–80, October 2010.

[75] Yonghong Yan, Max Grossman, and Vivek Sarkar. JCUDA: A Programmer-Friendly Interface for Accelerating Java Programs with CUDA. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par '09, pages 887–899, Berlin, Heidelberg, 2009. Springer-Verlag.

[76] Ying Zhang, Yue Hu, Bin Li, and Lu Peng. Performance and power analysis of ATI GPU: A statistical approach. In *Networking, Architecture and Storage (NAS), 2011 6th IEEE International Conference on*, pages 149–158. IEEE, 2011.