Universitat Politècnica de Catalunya (UPC) - BarcelonaTech

Facultat d'Informàtica de Barcelona (FIB)

Treball Final de Grau (TFG)

2012-2013 | Spring Term

Computer Architecture Department (DAC)

Bachelor Degree in Informatics Engineering (Computing)

Domain Specific Languages for High Performance Computing

A Framework for Heterogeneous Architectures

Alejandro Fernández Suárez
(alejandro.fernandez@bsc.es)

Director: **Eduard Ayguadé Parra** (eduard@ac.upc.edu)
Co-director: **Vicenç Beltran Querol** (vbeltran@bsc.es)

In the hallway of the university, Dominique asked a colleague: “I’ve always wondered why, in general, people talk using always the same general cumbersome vocabulary instead of trying to be concise and specific”.

“Hmm” retorted his surprised interlocutor, “well, I suppose everybody likes keeping things simple, like using the first words that come to their minds and everybody knows”.

“Yeah, that’s it!”, exalted Dominique, “Simple, short, concise and specific so everybody gets your point!”.

This puzzled the interlocutor, but both walked into class with a happy smile...

Acknowledgments

I would like to thank my advisors Vicenç Beltran and Eduard Ayguadé for giving me the opportunity to work at BSC and be involved in very exciting projects. Thanks for believing in me for this project, listening to my ideas and all the helpful comments and advice about my work. I am also grateful to Tomasz Patejko for all his help, comments and deep thinking conversations.

The early days of this project were about getting to know the potential of LMS for our purpose. I would like to thank Arvind Sujeeth and Tiark Rompf for their detailed explanations about LMS and Delite when I was trying to understand the whole system.

Thanks also to Florentino Sainz for his work on the OpenCL support for the Mercurium compiler and all the programming models department people that make our code able to run. In general, thanks to all my colleagues at BSC for being part of a great and enjoyable work environment.

Special thanks to my lovely girl Eva, for her support in everything I do and her enthusiasm in reading and commenting my documents. I really enjoy your questions about practically everything, they help me to better understand everything I write about.

Last but not least, I want to thank my friends and family for their support and providing a healthy counterbalance to my work and research.

Domain Specific Languages for High Performance Computing

Alejandro Fernández Suárez

Computer Architecture Department (DAC)
Universitat Politècnica de Catalunya (UPC)
2013

ABSTRACT

High Performance Computing (HPC) relies completely on complex parallel, heterogeneus architectures and distributed systems which are hard and error-prone to exploit, even for HPC specialists. Further and further knowledge on runtime systems, dependency tracking, memory transaction optimization and many other techniques are a must-have requirement to produce high quality software capable of exploiting every single bit of power an HPC system has to offer. On the other hand, domain experts like geologists or biologists are usually not technology-aware enough to produce the best software for these complex systems. Nowadays, the only way to successfully exploit an HPC system requires that computer and domain experts work closely towards producing applications to solve domain problems. Domain experts have the knowledge on the domain algorithms, while computer experts know how to efficiently map these algorithms on HPC systems.

This project proposes a framework that eases most of the processes related to the production of Domain Specific Languages (DSLs) that run on top of accelerator-based heretogeneus architectures. By using DSLs, domain experts can develop their applications using their own high level language, focusing only on their hard-enough issues. Meanwhile, computer experts stay only improving the implementation of these DSLs to make the most out of an HPC platform. This way, we keep each expert focused as much as possible on its natural domain of expertise.

Keywords: DSL, OmpSs, Scala, OpenCL, Compilers, Parallelism.

Lenguajes de Dominio Específico para Computación de Alto Rendimiento

Alejandro Fernández Suárez

Departamento de Arquitectura de Computadores (DAC)
Universitat Politècnica de Catalunya (UPC)
2013

RESUMEN

La computación de alto rendimiento (HPC) se basa en la utilización de sistemas distribuidos heterogéneos difíciles de explotar, incluso por expertos en el área de HPC. Para producir software HPC capaz de explotar toda la potencia de un supercomputador, cada vez se requiere más conocimiento en técnicas avanzadas de programación tales como análisis de dependencias u optimización de transacciones de memoria. Por otra parte, expertos de dominio tales como geólogos o biólogos no estan tan familiarizados con las tecnologías actuales como para producir software para arquitecturas modernas. Por este motivo, la única manera de explotar supercomputadores actuales es hacer que los expertos de dominio trabajen muy conjuntamente con programadores expertos para producir sus aplicaciones.

Este proyecto propone un sistema que agiliza muchos de los procesos asociados al desarrollo de DSLs para HPC. Mediante el uso de DSLs, los expertos de dominio pueden desarrollar sus aplicaciones utilizando su propio lenguaje de alto nivel, centrándose solamente en sus ya complicados problemas. Mientras tanto, los programadores expertos focalizan sus esfuerzos en mejorar y optimizar la implementación de estos DSLs de modo que aprovechen al máximo los recursos que ofrezca cada sistema. De este modo, cada experto se mantiene en su dominio natural, maximizando así su productividad.

Palabras clave: DSL, OmpSs, Scala, OpenCL, Compilador, Paralelismo.

Llenguatges de Domini Específic per a Computació d'Alt Rendimient

Alejandro Fernández Suárez

Departament d'Arquitectura de Computadors (DAC)
Universitat Politècnica de Catalunya (UPC)
2013

RESUM

La computació d'alt rendiment (HPC) es basa en la utilització de sistemes distribuïts heterogenis difícils d'explotar, inclús per experts en l'àrea de HPC. Per a produïr software HPC capaç d'explotar tota la potència d'un supercomputador, cada cop es requereix més coneixement en tècniques avançades de programació tals com anàlisi de dependències o optimització de transaccions de memòria. D'altra banda, experts de domini tals com geòlegs o biòlegs no estan tan familiaritzats amb les tecnologies actuals com per a produïr software per a arquitectures modernes. Per aquest motiu, l'única manera d'explotar supercomputadors actuals es fer que els experts de domini treballin molt conjuntament amb programadors experts per tal de produïr les seves aplicacions.

Aquest projecte proposa un sistema que agilitza molts dels processos associats al desenvolupament de DSLs per a HPC. Mitjançant DSLs, els experts de domini poder desenvolupar les seves aplicacions fent servir el seu propi llenguatge d'alt nivell, concentrant-se en els seus prou complicats problemes. Mentres tant, els programadors experts focalitzen els seus esforços en millorar i optimitzar la implementació d'aquests DSLs de manera que aprofitin al màxim els recursos que ofereix cada sistema. D'aquesta manera, cada expert es manté al seu domini natural, maximitzant així la seva productivitat.

Paraules clau: DSL, OmpSs, Scala, OpenCL, Compilador, Paral·lelisme.

Table of Contents

List of Figures

List of Code Samples

Chapter 1

Introduction

The evolution of computer architectures is nowadays focused on multicore technology. Every single computer, even mobile phones have more than one central processing unit. Therefore, multicore technology is everywhere today, and we require parallel programming techniques to use them efficiently. Thus, parallel programming does no longer just involve expert parallel programmers, but even also mobile application and web developers, operative system programmers, and many more, specially scientists working in supercomputing environments. However, computer architectures are becoming more and more heterogeneous and complex, with more and more different parallel programming models and techniques, which makes it difficult even for experts to keep pace. So, how do we keep productivity and performance at the same time? Application developers need to keep producing their applications, but they also need them to be scalable, fast and efficient in modern architectures. Domain Specific Languages (DSLs) are in fact the most popular approach for such purpose.

DSLs are known for being small languages designed to only solve problems from a particular domain. First, due to their tight coupling with a specific domain, they allow domain experts to intuitively express their applications. However, the performance and scalability of these languages are proportional to the effort required to develop them. Thus, it only pays off to implement a high performance tool-chain (compiler and execution runtime) for a DSL if it is going to be really widely used. Simple Query Language (SQL) is a good example. SQL is massively used to query and update relational databases, so spending lot of effort to develop and implement a powerful framework to support it is justified. On the other hand, there are hundreds of useful DSLs with a (relatively) small number of users that do not justify the efforts required to optimize it. Another problem of programming languages in general and DSLs in particular is that they often need to be mixed together to solve large problems, which is usually a hard and error-prone process that significantly reduces the benefits and applicability of DSLs.

1.1 Motivation

A framework that eases the development of DSLs that run on HPC systems would justify the effort required to develop a custom tool-chain for each of them. Starting by composing and reusing this high performance framework to produce multiple unrelated

DSLs, the total development costs gets more than amortized. Moreover, providing a framework to develop DSLs for HPC, we will be able to use multiple DSLs to develop complex applications, because the current development of applications that use several languages is usually a painful experience.

Isolating DSL designers (which are the ones that will create DSLs using our system) from burdens such as low level device management and explicit parallel programming will ease the development of new DSLs, leaving to framework developers activities that only expert machine-level programmers can successfully perform. To begin with, high level DSLs built upon this framework will empower domain experts with productive tools for their domain. At the same time, these tools would be implemented on our system to take advantage of underlying heterogeneus architectures through Open Computing Language (OpenCL) and/or runtime systems. In a ideal situation, domain experts should only access abstractions, which are expressions of their DSL. These abstractions will be provided by language designers, experts in both programming language design and heterogeneous computing. Additionally, this isolated design will in turn would allow further optimizations of generated code. As a result, productivity is increased from both sides: Domain experts, now able to just focus on implementations of algorithms specific for their domains, and language/compiler designers that can safely and blissfully ignore information not related to language and compiler design. Figure 1.1 depicts the approach towards high performance DSLs.

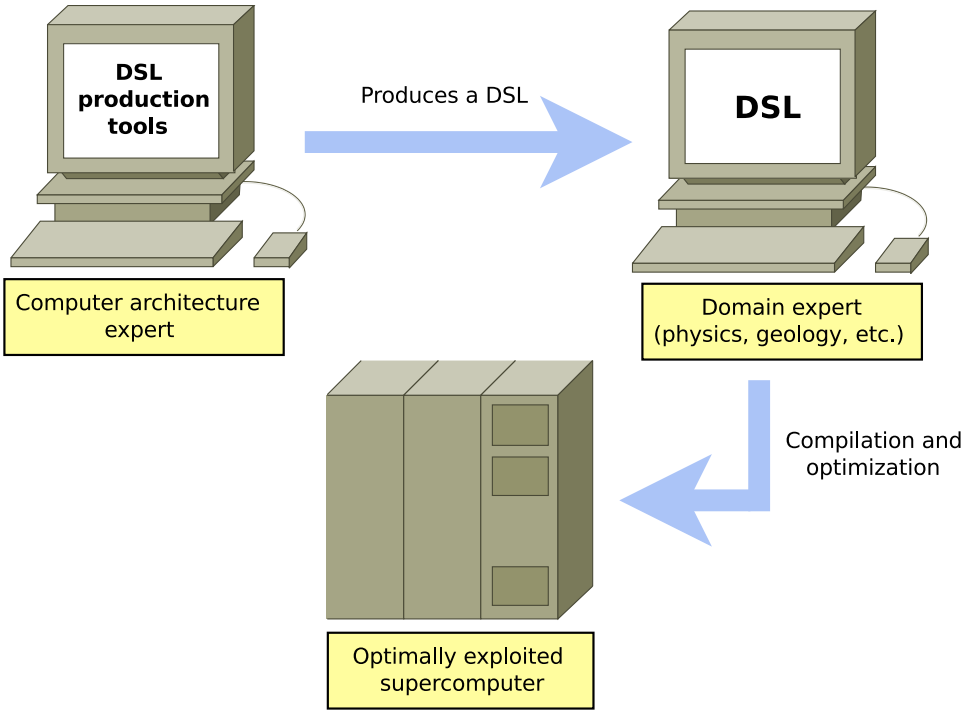

Fig. 1.1: Isolated modular approach to DSLs for HPC

1.2 Objectives

This project presents Data Flow SuperScalar Language (DFSSL), a language that leverages the performance and portability of kernels developed in OpenCL C, replacing the cumbersome OpenCL + OmpSs API by a high level programming model. This language, DFSSL, is implemented as an embedded compiler with Scala and the Lightweight Modular Staging (LMS) library. The LMS library will be overviewed in Section 2.7 as related work and explained in detail in Section 3.4 as an important tool for this project. Once DFSSL is ready, high level DSLs can be implemented using Scala + LMS or any other approach, they just need to be translated to DFSSL to benefit from all of its features. Afterwards, the DFSSL code will be translated to OpenMP Superscalar (OmpSs) (a variant of OpenMP for heterogeneus architectures) in order to use the Nanox [9] runtime from Barcelona Supercomputing Center (BSC). Figure 1.2 represents this toolchain and all the steps that take place in the compilation process of a DSL built on top of DFSSL. Please note that high level DSLs targetting DFSSL can be implemented used any convenient approach, but this project will present the tools we used to build DFSSL detailedly, so any other DSL can be built in the same way.

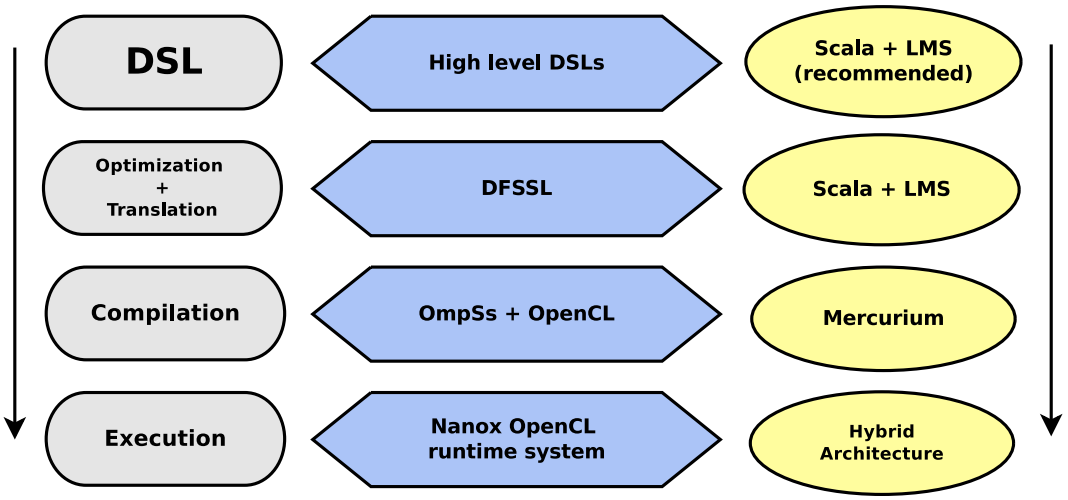

Fig. 1.2: DSL production toolchain for HPC environments

Given the purpose of our framework, the main requirements of DFSSL are:

- **Composability.** We want DFSSL to interoperate with already existing libraries so we save as much effort as possible on development stages.
- **Efficiency.** Since it has to be the common layer for high level DSLs, it pays off to invest time on improving its performance.
- **Genericity.** It is essential that despite being a language focused in data–flow accelerator–based applications, DFSSL is expressive enough to allow almost any kind of high level DSL be implemented on top of it.

- **Safety.** In DFSSL applications and applications for heterogeneus architectures in general, the program that is run on the host machine interacts with the accelerator program. The language must provide safety mechanisms such as type checking between host and accelerator code or any required system that ensures there are no programming errors when creating or running accelerator programs (kernels).

1.2.1 Detailed objectives

The specific objectives of this project are exactly:

Prototype implementation of DFSSL

This prototype had to be enough to explore the potential of the framework as well as to experiment the possible problems such a project can have, effectively setting a base for further research. With this idea in mind, we designed and implemented DFSSL, focusing on making it best fitted for serving as target language to DSLs for HPC. Thus, DFSSL and the whole framework had to meet the following requirements:

- Comfortable and versatile syntax
- Static checking of source code to minimize programming errors related to host/accelerator interaction
- Support for integrating already existing external libraries
- Moderated complexity, enough to specify all major performance–related details

System validation by means of examples

To test and validate DFSSL, we chose some typical problems and algorithms on the area of HPC. These problems were *N-Bodies* [25], *K-Means* [45] and Cholesky's factorization [26]. These examples are shown later in this memo, in Chapter 6. Please note that even if DFSSL is designed to serve as target language to HPC DSLs, we tested it implementing algorithms for simplicity. However, the language was completely verified because all the translation is performed no matter if the code being translated is a simple algorithm or a whole higher level DSL application translated by another tool.

1.3 Contributions of this TFG

This project presents DFSSL, a high level programming model that eases the development of applications for heterogeneous architectures, focused on serving as a target language for parallel high-level DSLs. DFSSL has been implemented as an embedded compiler in Scala and we explain all the tools we used in detail. This way, any programmer(s) can implement DSLs on top of DFSSL with a modest effort, using just the same approach.

1.4 Document structure

The rest of this document is structured as follows: Chapter 2 gives a briefing over the state of the art on the field of DSLs for HPC. Following on, Chapter 3 then introduces the tools used to develop the project and justifies the choices made. The next chapter, Chapter 4, explains DFSSL and how to use it. Then, Chapter 5 nails down the details on the implementation of DFSSL's most important features. To continue, some results from example applications are shown in Chapter 6. Chapter 7 details the development cost of the project, its initial and real planning, the methodology we followed and its social/economic impact. Chapter 8 concludes with some insight about the whole experience and research results obtained from the work done, overviewing some of the future work we can start now. The document ends with two Appendixes A and B. Appendix A contains instructions on how to deploy and install DFSSL, getting it ready for use. Finally, Appendix B shows the code generated by the DFSSL applications explained in Chapter 6.

CHAPTER 2

Related Work and State of the Art

There exist different DSLs for all sort of applications (including non-performance critical environments), but none of them has been implemented by translation into a language like the one we present in this project. This chapter will overview the state of the art on the main topics regarding DSLs in general and tools for providing high performance DSLs.

2.1 DSLs boosting productivity

DSLs, also called small languages [15], are widely used in computer science [22] where performance and productivity are prior to generality and they are also a subject of intensive research [49]. On one hand, their implementation and design are suitable for a particular domain, so they are very expressive as languages for solving problems from this domain. On the other hand, due to their simplicity and size they are easy to learn and use. DSLs can be found in many domains of computer science, such as AWK [14] for text processing and Bison [21] for parser generation. Further examples are VHDL and Verilog, which are widely used for electronic circuitry design. Agda [36] and Coq [16] are used for theorem proving.

DSLs have also proved to be successful outside computer science. For example, R [19] is used for statistical computation and data analysis. More examples of successful DSLs are Matlab [32] and Mathematica [50] for numerical computing.

With advent of multi/many-core architectures, many emerging programming languages implement parallelism and concurrency as one of their paradigms. DSLs are no different. One of such languages is OptiML [47], a parallel DSL for solving problems from the area of machine learning. OptiML was implemented using Delite (see 2.7.1), which is a library built on top of LMS (see 2.7) and the Scala Virtualized compiler [13, 34].

2.2 Embedding domain-specific languages

Different methods allow DSL developers to embed DSLs in general-purpose languages. DSL embedding has the main advantage of saving time and effort by deep reuse of the host language compiler infrastructure.

One of the first languages that introduced a mean for embedding DSLs is Lisp [46]. Lisp offers a powerful macro system that allows programmers to extend the syntax of the core language. Macros are expanded at preprocessing time into regular Lisp code, then they are further compiled by the compiler. This makes Lisp macros good fit for DSLs. DSL syntax can be defined in terms of Lisp syntax by extending it using this macro system. DSL constructs can be implemented as macros and composed the same way macros are composed.

Lisp macros are an approach to static meta-programming and code generation. A similar idea was implemented in Template Haskell [44]. Even though Haskell does not have macro system like Lisp, the same compile-time meta-programming was achieved by several extensions in the language, such as quasi-quotations [44]. This facility implements a set of functionalities such as fresh name generation or error reporting, needed for compile-time code generation.

So far, only compile-time code generation has been discussed. One of the platforms that allow runtime code generation is MetaOCaml [41, 48]. This language is a set of extensions of the OCaml language that allow dynamic type-checked code generation. The language is based on a concept called multi-stage programming [48]. The idea is that some computation can be divided into stages, depending on information availability. In MetaOCaml, the programmer is equipped with language constructs that allow explicit specification of the part of the application that should be staged.

2.3 Parallel computing libraries

The first approach (and the most direct and straightforward) to develop a high performance DSL is to provide a collection of functions and types defined on a library of an existing language. Then, this library becomes be the actual DSL produced and its implementation usually relies on parallel computing libraries that provide parallel operations. However, implementing DSLs as libraries has some it has some limitations. These limitations include difficulty to optimize applications written in this DSL or supporting an expressive syntax. Such problems occur mainly because the compiler is used just as a nonprogrammable black box that turns our code into something we can run. The following sections will give some examples of parallel computing libraries upon which parallel DSLs could be implemented.

2.3.1 DFScala

DFScala [23] is a Scala [39] library for expressing data-flow parallelism. Libraries written in Scala can offer very complete and expressive interfaces, as if the library syntax was supported by the Scala core language. In contrast with our system, we want to offer support for accelerators and any runtime system, not necessarily the Java Virtual Machine (JVM), which is the case of DFScala.

2.3.2 Generic Parallel Collection Framework

The Generic Parallel Collection Framework [42] of Scala is an implementation of all the standard data structures of Scala whose operations are executed in parallel. This is possible due to the pure functional design of immutable data structures. In the same way as DFScala (see section 2.3.1), using these libraries to implement DSLs poses a restriction on the runtime platform and some other desirable features, whereas our framework will be fully customizable in all these aspects.

2.3.3 CLPP and CUDPP

CLPP [1] and CUDPP [4] are C++ [2] libraries that implementent a collection of parallel operations in OpenCL [11, 35] and CUDA [3], repectively. As all parallel computing libraries, it can be used to provide efficient DSLs. However, this method is very limited when it comes to provide convenient syntax, because DSLs implemented as libraries that call CLPP and/or CUDPP internally can just offer C++ syntax.

2.4 DSLs by extending compilers

An alternative way to produce DSLs without writing a whole compiler is modifying and/or extending an already existing compiler to fit our current needs. By using plugins, implicit operations can be added to an already existing language. Depending of the compiler support for extensions, even some operations syntax elements can be completely redefined, thus changing the core behaviour of the language. Although this approach is effective for producing complete DSLs, modifying an existing compiler is not necessarily a straightforward task and in many cases. The only complete DSL implemented this way for HPC environments is Liszt.

2.4.1 Liszt

Liszt [6, 20] is a DSL for mesh-based partial differential equations solvers. Liszt comes with its own runtime systems that optimize all the operations performed on geometry. The main disadvantage of Liszt against DSLs implemented with our system is that Liszt components and optimizations are completely ad-hoc, not reusable for by any other similar DSL and not composable with other libraries. In contrast, our framework will allow optimizations and components of the implementation of DSLs to be reused flawlessly among different DSLs.

2.5 Leveraging HPC programming for heterogeneus architectures with OmpSs

OmpSs is a parallel task-based programming model for C/C++ applications. It extends OpenMP `task` pragmas with constructs for specifying data dependencies between tasks and the devices they will run in. It consists of source-to-source translator, Mercurium, and runtime library, Nanox++. Mercurium translates C/C++ code with pragmas delimiting regions or functions that will be executed in parallel to C/C++ code with calls to Nanox++ API. When an application is executed, Nanox++ handles the efficient parallel execution of tasks. It is worth noting, that the OmpSs program do not require any call to the OpenCL API, the Mercurium compiler will generate all the required boilerplate code to setup the kernel, while the Nanox++ runtime will perform the required data transfers from the host to the accelerator and viceversa and run the kernel.

```
1 const unsigned N = 100;
2 #pragma omp task in([N*100] bufin) out([N*100] bufout)
3 #pragma omp device target(opencl) ndrange(1, N*100, N)
4 parKernel(bufin, bufout);
```

Code 2.1: Call to OpenCL kernel annotated with an OmpSs task pragma

Code 2.1 shows how OmpSs pragmas are used to run the kernel `parKernel` in parallel on an OpenCL device. First, the programmer specifies dependencies through direction statements (line 1): `in`, `out`, and `inout`. These statements tell the OmpSs compiler (Mercurium) whether a task reads data, writes to it or both, as well as the input size in case of pointers. In line 2, a target device is specified through a `device target` statement. Also in line 2, the programmer specifies the OpenCL NDRange (offset, global size and local size, repectively) through a `ndrange` statement. The information about the data dependencies of a task and target architecture is used by the Nanox++ runtime to build dependency graphs, handle data transfers between CPU and GPU, and schedule tasks for parallel execution.

2.6 DSLs for stencil computations

Finite-difference stencil computations are very common in numerical modeling and they exhibit high degree of data parallelism and regular structure. There exist DSLs and frameworks for stencil computations and they usually exploit the stencil's data access pattern by generating CUDA/OpenCL code, replacing loops by kernels that run on the accelerators.

One of these frameworks is Orio [30], an extensible framework for transformation and autotuning of codes written in different source and target languages, including transformations from a number of simple languages (e.g., a restricted subset of C) to C, Fortran, and CUDA targets. The tool generates many tuned versions of the same operation using different optimization parameters, and performs an empirical search for selecting the best among multiple optimized code variants.

Another similar DSL + framework for stencil computations is Physis [31]. Physis is a compiler-based programming framework that automatically translates user-written structured grid code into scalable parallel implementation code for GPU-equipped clusters. The language features declarative constructs that allow the user to express stencil computations in a portable and implicitly parallel manner. Then, the framework translates the user-written code into actual implementation code in CUDA for GPU acceleration and MPI for node-level parallelization with automatic optimizations, such as computation and communication overlapping.

2.7 Lightweight Modular Staging

The LMS library [7, 43] is a Scala library composed by a set of functions and type definitions that allow the programmer to define stages [48] throughout the translation process of a language. Each stage represents a translation of the application code in which the system applies the optimizations corresponding the application's domain. The last stage is the application execution with a specific input, so all the data is defined and the program computes its results. Please note that LMS is a library written in Scala, so a DSL application is compiled together with the DSL implementation, giving place to an executable code generator. Then, once the code generator is run, the DSL application gets translated. LMS allows a modular quick development of all the part composing a DSL: Representation (language user interface), expression (representation semantics or language implementation) and code generation, where the application is finally translated to the next stage. The LMS library requires an experimental version of the Scala compiler called Scala Virtualized Compiler [13, 34]. This compiler allows to redefine all the syntax elements of the language, thus LMS is able to change integrally the Scala default semantic for its purpose. LMS will be explained in proper detail in Chapter 3, in section 3.4.

2.7.1 Delite

Delite [5, 17] is a Scala library built on top of LMS with the purpose of adding parallelism concepts to the LMS DSL implementation constructs. The Delite framework allows to define the semantics of DSL operations in terms of parallel constructs that run on accelerators or centralized multiprocessors by generating CUDA, MPI and C++ code. One of the most popular DSLs implemented with Delite is OptiML [47], a DSLs for machine learning applications.

Even though the purpose of Delite is similar to ours, its support for accelerators is limited to the use of parallel predefined operations. Our system, in contrast, supports any kind of accelerator function written by the DSL implementer. Finally, Delite's runtime system handles all the predefined operations on at a data flow level. In our case, we will use OmpSS [10], which runs on top of Nanox [9] to get dynamic scheduling for accelerators and the host side application in addition to regular data flow related optimizations.

CHAPTER 3

Tools

This chapter introduces the main tools used for developing the language this project presents, DFSSL. Firstly, we needed a language to program accelerators and express parallelism. Among the choices are OpenCL [11, 35] and CUDA [3], which are today's most common languages for accelerators (GPUs, basically). For the purpose of this project, CUDA is too focused on NVIDIA GPUs, so, in order to target any accelerator platform, OpenCL was chosen. Then, we decided to provide DFSSL as an embedded DSL, so we required a host language. This language was required to be virtualizable, meaning every piece of syntax is implemented through functions that can be redefined, extended and combined.

Scala fitted all these requirements. One of the strongest points of Scala is that provides a way to naturally create libraries that look and feel like they are supported natively. Moreover, by using Scala, we could use the LMS library to quickly and efficiently implement our language as a DSL embedded in Scala. DFSSL required some support to analyze OpenCL kernel files to provide some of its safety mechanisms. The Low Level Virtual Machine (LLVM) and Clang served that purpose. Finally, in order to use a runtime system to automate the scheduling of application tasks, we decided that DFSSL was going to target (be translated to) C++ OmpSs with OpenCL. The rest of this chapter briefly describes each of the previously mentioned tools.

3.1 OpenCL

OpenCL [11, 35] is considered nowadays as the open standard for parallel programming of heterogeneus architecture. OpenCL applications are composed of two major parts: Host and accelerator side. The former is the application in charge of issuing work to accelerators, manage input/output and expressing concurrency. The latter runs on the accelerators producing results.

3.1.1 A language for heterogeneus systems

OpenCL requires the programmer to explicitly manage devices (acelerators) and platforms, take command over each data transfer made between the host and the devices and specify the control flow of execution queues for each device. Let us take a closer look to these aspects:

Host application A regular C or C++ application containing calls to the OpenCL API. It contains all the code related to device initialization, management of execution queues and data transfers.

Kernels Functions loaded into computing resources such as CPUs, GPUs or any device supported by the OpenCL platforms installed on a particular system. These functions are written in a language called OpenCL C, which is basically a C dialect with vector types and many restrictions such as lack of recursivity and pointers

The following concepts related to OpenCL devices need to be known in order to understand how they operate:

Single Instruction Multiple Thread (SIMT) . This concept refers to the way instructions are executed in the device. The same code is executed in parallel by a different thread, and each thread executes the code with different data.

Work items are the smallest execution entity. Each time a kernel is executed, lots of work-items (a number specified by the programmer) are spawned, each one executing the same code. Each work-item has an ID, which is accessible from the kernel, and which is used to distinguish the data to be processed by each work-item.

Work groups exist to allow communication and cooperation between work items. They reflect how work items are layered (an N-dimensional grid of work groups, with N being 1, 2 or 3). Work-groups also have an unique ID that is accessible from the kernel.

ND-Ranges are the next organization level, specifying how work groups are organized (again, an N-dimensional grid of work groups). ND-Ranges are defined by a local and a global size. The local size corresponds to the number of work items inside a work group and the global size is the actual number of work groups.

The typical workflow of an OpenCL application is:

1. Get available platforms and select one or more.
2. Get platforms' devices and select one or more.

3. Compile each kernel for each device it will execute on.

4. For each device we are using inside each platform, create a command queue and start submitting work to the this queue.

```
#include <iostream>
#include <cstring>
#include <CL/cl.h>
#define BUF_SIZE_LOCAL  1024
#define BUF_SIZE_GLOBAL BUF_SIZE_LOCAL*5*1024
const char kernel_init[] = "__kernel void init (__global float *out) {\
      const uint index = get_global_id(0); \
      out[index] = 1.0f;}";
const char kernel_add[] = " \
__kernel void add (__global float *inout, __global float *in) { \
      const uint index = get_global_id(0); \
      inout[index] = inout[index] + in[index];}";
int main(int argc, char** argv) {
      using namespace std;
      cl_platform_id platform;
      cl_device_id devices[2];
      clGetPlatformIDs(1, &platform, NULL);
      clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 2, devices, NULL);
      cl_context_properties properties[] = { CL_CONTEXT_PLATFORM,
            (cl_context_properties)platform, 0 };
      cl_context context = clCreateContext(properties, 2, devices,
            NULL, NULL, NULL);
      cl_command_queue cq1 = clCreateCommandQueue(context, devices[0],
            0, NULL);
      cl_command_queue cq2 = clCreateCommandQueue(context, devices[1],
            0, NULL);
      size_t local = BUF_SIZE_LOCAL;
      size_t global = BUF_SIZE_GLOBAL;
      size_t bs = BUF_SIZE_GLOBAL*sizeof(float);
      float buf1[BUF_SIZE_GLOBAL];
      float buf2[BUF_SIZE_GLOBAL];
      cl_mem device_buf1 = clCreateBuffer(context, CL_MEM_READ_WRITE,
            bs, NULL, NULL);
      cl_mem device_buf2 = clCreateBuffer(context, CL_MEM_READ_WRITE,
            bs, NULL, NULL);
      size_t initsrcsize = strlen(kernel_init);
      const char *initsrcptr[] = {kernel_init};
      size_t addsrcsize = strlen(kernel_add);
```

```
39        const char *addsrcptr[] = {kernel_add};
40        cl_program initprog = clCreateProgramWithSource(context, 1,
41             initsrcptr, &initsrcsize, NULL);
42        clBuildProgram(initprog, 0, NULL, "", NULL, NULL);
43        cl_program addprog = clCreateProgramWithSource(context, 1,
44             addsrcptr, &addsrcsize, NULL);
45        clBuildProgram(addprog, 0, NULL, "", NULL, NULL);
46        cl_kernel initK = clCreateKernel(initprog, "init", NULL);
47        cl_kernel addK = clCreateKernel(addprog, "add", NULL);
48        clSetKernelArg(initK, 0, bs, &device_buf1);
49        clEnqueueNDRangeKernel(cq1, initK, 1, NULL, &local, &global,
50             0, NULL, NULL);
51        clSetKernelArg(initK, 0, bs, &device_buf2);
52        clEnqueueNDRangeKernel(cq2, initK, 1, NULL, &local, &global,
53             0, NULL, NULL);
54        clEnqueueReadBuffer(cq2, device_buf2, CL_FALSE, 0, bs,
55             buf2, 0, NULL, NULL);
56        clFinish(cq1); clFinish(cq2);
57        clSetKernelArg(addK, 0, bs, &device_buf1);
58        clSetKernelArg(addK, 1, bs, &device_buf2);
59        clEnqueueWriteBuffer(cq1, device_buf2, CL_FALSE, 0, bs,
60             buf2, 0, NULL, NULL);
61        clEnqueueNDRangeKernel(cq1, addK, 1, NULL, &local, &global,
62             0, NULL, NULL);
63        clEnqueueReadBuffer(cq1, device_buf1, CL_FALSE, 0, bs,
64             buf1, 0, NULL, NULL);
65        clFinish(cq1);
66        return 0;
67  }
```

Code 3.1: Simple OpenCL Example

Queues are a way to submit commands and enable the host to use all devices at the same time. Code 3.1 shows an example of an OpenCL application that does the following:

1. Initialize buffers 1 and 2 on two different GPUs
2. Adds them together on GPU 1

Please note there are two different queues, one for each device. The initialization is done in parallel one each GPU, but the addition has to wait for both initializations to finish. Then the addition is performed on one of the GPUs. This tiny example illustrates some of the details a programmer has to take into account to produce a relatively simple application. Subsection 3.1.2 describes these concerns in more detail.

3.1.2 Drawbacks of OpenCL

Despite being powerful and flexible, OpenCL is in many cases too cumbersome. Its power and flexibility come at a price. The main aspects that make OpenCL powerful and thus harder to use are:

Explicit device selection including managing different OpenCL platforms on the same machine and all devices inside each platform. For example, the programmer needs to write code to check each device's properties in order to determine which is the best one to run a particular kernel.

Explicit data transfers between host and devices. These transfers need also to be manually managed, which means each time data has to go back and forth from devices to the host, the programmer needs to write functions that keep different views of the same data (one for each device used) in order to get good performance.

Explicit flow control through queues and events. A queue is associated to a particular device (and its context), so each device inside a machine has its own associated set of queues that can signal events each time an action is finished, commands can wait until a set of events are done before starting and many other synchronization mechanisms. The programmer is then also required to handle all this concurrency in order to effectively use all of a machine's computing resources. In general, for machines with multiple devices, some kind of user-provided load balancing is necessary to achieve the best performance.

Such amount of code independent from the algorithms being implemented and the application design itself can be astounding, even for computer science specialists. The fact is that there are no large applications written in OpenCL nowadays, due precisely to these highly complex aspects involving production of HPC software with OpenCL.

The tools and methodologies arising from this project will separate these concepts and technicalities from the parallel algorithm design itself. Later in this document, Chapter 4 will show a DSL that allows to just express the most high level concepts of a data flow application. Then, the implementation of this language will automatically determine all data dependencies, thus freeing the programmer from technical duties such as data transfers, flow control or explicit resource management. Outside the scope of this project, as future work, all this information will be then used by a runtime system to execute the application in an even more optimal way.

3.2 OmpSs (OpenMP SuperScalar)

OmpSs [10] is a programming model developed at BSC based on precompiler directives. Its objective is to extend OpenMP with new directives to support asynchronous parallelism and heterogeneity. However, it can also be understood as new directives extending other accelerator based Application Programming Interface (API) like CUDA or OpenCL. OmpSs is built on top of the Mercurium [8] source-to-source compiler and the Nanox [9] runtime system.

3.2.1 OmpSs features used in this project

OmpSs provides a lot of features, however, for the moment, this project will just use parallel loops, host tasks and accelerator tasks.

Parallel loops

Taken directly from OpenMP, a regular `for` loop can be annotated so the runtime can run it in parallel flawlessly if all iterations are independent. The code snippet 3.2 show an example of annotated parallel loop.

```
#pragma omp parallel for
for (unsigned i = 0; i < N; i++) a[i] += b[i];
```

Code 3.2: Annotating parallel loops

Host side tasks

Tasks are functions run asynchronously at runtime that are automatically scheduled to run in the host application. A given function can be declared and used as a task by annotating it specifying the directions of its parameters, like shown in code snippet 3.3.

```
#pragma omp task inout(a) output(b)
void sum(int &a, int &b) { a += b; }
```

Code 3.3: OmpSs host side task

Accelerator side tasks (kernels)

In order to run OpenCL kernels in OmpSs and get them scheduled together with host side tasks and other kernels as well, we need to annotate the kernel functions with the `device` construct. Code snippet 3.4 shows an example of an annotated OpenCL kernel.

```
unsigned N = 1024;
#pragma omp device(opencl) ndrange(1, N*10, N)
#pragma omp task inout([N*10] a) output([N*10] b)
kernel void sum(global int *a, global int *b) {
    unsigned id = get_global_id(0);
    a[id] += b[id];
}
```

Code 3.4: OpenCL kernel for OmpSs

3.2.2 The OpenCL example revisited

In order to illustrate the potential of OmpSs + OpenCL over raw OpenCL, Code 3.5 shows the equivalent code of the OpenCL application example shown previously in Code 3.1.

```
const unsigned LOCAL = 1024;
const unsigned GLOBAL = LOCAL*5*1024;

#pragma omp device(opencl) ndrange(1, GLOBAL, LOCAL)
#pragma omp task output([GLOBAL] a)
kernel void init(global int *a) {
    unsigned id = get_global_id(0);
    a[id] = 1;
}

#pragma omp device(opencl) ndrange(1, GLOBAL, LOCAL)
#pragma omp task inout([GLOBAL] a) output([GLOBAL] b)
kernel void add(global int *a, global int *b) {
    unsigned id = get_global_id(0);
    a[id] += b[id];
}

int main() {
    float buf1[GLOBAL], buf2[GLOBAL];
```

```
    init(buf1);
    init(buf2);
    sum(buf1, buf2);
    #pragma omp taskwait
    return 0;
}
```

Code 3.5: OmpSs version of the OpenCL example

Note how we do not need to specify any OpenCL programs, queue intermediate synchronizations or platforms, just the tasks directions, the target device type and the waiting point (the `taskwait` statement). Everything is automatically managed at runtime by Nanox++, giving place to a much efficient and convenient approach to heterogeneous architecture programming.

3.3 Scala

Scala [37] is a statically typed, multi-paradigm programming language with type inference. It supports both functional [1] and object-oriented programming paradigms. It also supports concurrency with the actor model. Scala has a lot of mechanisms that allow the programmer to write libraries that are used like they were built-in features provided by the language itself. Thus being attractive for embedding DSLs.

3.3.1 Objects and classes

Scala provides classes, objects and traits. These language constructs are used to implement abstract data types. Objects are created through class instantiation. Scala provides singleton objects – they act as group of static functions (they operate on no particular implicit instance). If a class and an object share the same name they are called companion classes/objects. Companion objects and classes have access to each other's private members.

Scala allows declaring values, methods and types as abstract. Abstract type members are used to build generic components. This feature will be explained in more detail at Section 3.3.5.

Abstract classes cannot be instantiated and its abstract members need to be defined in their subclasses. This is achieved through the `extends` keyword. Subclasses inherit non-private members of its superclasses. In Scala, multiple inheritance is not allowed, meaning classes cannot inherit from multiple superclasses.

[1] Functions are first-class citizens

3.3.2 Traits and mixin composition

Traits in Scala are the basic unit of code reuse. Traits are declared in a similar way as regular classes with the exception that the programmer uses keyword `trait` instead of keyword `class`.

Scala classes can mix in several traits with the keywords `extends` and `with`. By mixing in several traits into classes Scala provides stackable composition. The order in which traits are mixed in matters. Calls to methods on the superclass or mixed in traits through the `super` call, are determined by linearization rules. Hierarchies of superclasses and mixed in traits are ordered in a linear way by the Scala compiler. Whenever `super` is invoked, the next method from the formed chain is called.

3.3.3 Generics

Scala supports type parametrization by introducing the concept of generic classes. It allows the programmer to define types that can operate in a type-safe manner on values without relying on their types. Examples of definition and use of generic classes can be seen on Code 3.6 and Code 3.7.

```
1 class TypePrinter[T:Manifest] {
2       def print() = manifest[T].toString
3 }
```

Code 3.6: Declaration of a generic class `TypePrinter`

```
1 val i = new TypePrinter[Int]
2 i.print
3 val d = new TypePrinter[Double]
4 d.print
```

Code 3.7: Usage example of instances of a parametrized class `TypePrinter[_]`

On Code 3.6, a generic class `TypePrinter[_]` is declared. The implementation of this class was prepared for some, unspecified type `T` (line 1). Generics implement parametric polymorphism, meaning the implementation and interface of generics `TypePrinter[_]` is not coupled with any concrete type such as `Int` or `String`. This coupling is achieved when a generic type is instantiated and a type is specified.

Scala's `Manifest` is used here as implicit value (line 1). Manifests are also generic, preserving type information, so they can be used at runtime. Such wrapper for type information is needed, because the JVM implements type erasure: Type information is erased by the compiler and becomes unavailable at runtime for compatibility with previous versions of Java that did not support generics.

The generic class `TypePrinter[_]` contains only one member: A method called `print` that returns the name of the type as a `String`. This is achieved through a call to its generic function `manifest[_]` in line 2.

Code 3.7 shows how generic classes can be used. In line 1 the instance of the class `TypePrinter[_]` is parametrized with type `Int`. So, by parameterizing the generic class with a concrete type, such as `Int`, another type is constructed, namely `TypePrinter[Int]`. `TypePrinter[_]` is a type constructor.

When method `print` is called in line 2, it returns the string "`Int`". A similar situation takes place for the code in lines 4-5, but, in this case, the output is string "`Double`".

Please note that in this Section we are only scratching the surface of the features generics can implement. They also have a wide variety of purposes such as implementing higher-kinded types [33] and Concept Patterns [40][24].

3.3.4 Pattern matching

One of the functional features that Scala implements is pattern matching. Scala allows the programmer to match values of any type with a match-first policy. However, Scala, as an object-oriented language, extends this concept for objects. This is achieved with a special kind of classes called case classes.

Let us look at the example Code 3.8:

```
1 def findRoom(n: Int): String = n match {
2       case 103 => "Lab"
3       case 105 => "Dean"
4       case 104 => "Secretary"
5       case _ => "Empty"
6 }
```

Code 3.8: Pattern matching of values of type `Int`

Function `findRoom` (line 1 on Code 3.8) takes a parameter `n` of type `Int`. This parameter is pattern-matched against several cases (lines 2-5). If `n` is equal to one of these values, the corresponding string is returned. Line 5 is the default pattern, which, if reached, is always executed.

As previously mentioned, Scala allows the programmer to pattern-match case classes. The definition of a case class is shown on Code 3.9.

```
case class Rectangle(x: Int, y: Int, w: Int, h: Int)
```

Code 3.9: Simple case class declaration

The only difference between case classes and regular Scala classes is that case classes come by default with a constructor with the same name as the case class. Regular scala classes cannot be pattern-matched.

Code 3.10 shows how case classes are pattern-matched.

```
1 def assertRectangle(r: Rectangle): String = r match {
2       case Rectangle(0, 0, _, _) => // When rectange is at (0,0)...
3       // More cases...
4       case Rectangle(_, _, w, h) => // Use w and h to...
5 }
```

Code 3.10: Pattern matching on case classes

Code 3.10 illustrates how case classes can be pattern-matched against values of their members (line 2). Values of case class members can be bound and used (line 4).

Case classes act like regular Scala classes in terms of class hierarchies. This means that an instance of a superclass can be matched against instances of case classes implemented as subclasses derived from a given type.

3.3.5 Abstract type members

Subsection 3.3.1 shows how types can be abstract members of a class. This is another way of building abstractions in Scala (next to type parametrization, described briefly in Section 3.3.3).

Abstract type members, as it was the case with generics and type parametrization, allow the programmer to abstract interfaces over implementations.

The source code on Code 3.11 and 3.12 implement the same functionality as the one on Code 3.6 and 3.7: They print out the names of types `Int` and `Double`.

```
1 trait TypePrinter {
2       type T
3       implicit val m: Manifest[T]
4       def print() = m.toString
5 }
```

Code 3.11: Definition of a trait `TypePrinter` with abstract type member `T`

```
1 class Integrals extends TypePrinter {
2       type T = Int
3       val m: Manifest[T] = implicitly
```

```
4  }
5  class Doubles extends TypePrinter {
6        type T = Double
7        val m: Manifest[T] = implicitly
8  }
9  val i = new Integrals
10 i.print
11 val s = new Strings
12 s.print
```

Code 3.12: Implementation of abstract type members

Firstly, a trait is defined so that it contains an abstract type member. We can see this in line 2 on Code 3.11. This type member is not defined, it will be defined in classes that mix in the `TypePrinter` trait.

In line 4 an abstract value `m` of type `Manifest[T]` is declared. This value is defined in classes that mix in the trait. As it was the case with examples in subsection 3.3.3, we want to avoid type erasure by using manifests.

The type member `T` is defined in classes `Integrals` and `Doubles` (lines 2 and 8 on Code 3.12). Calls to the method `print` in lines 14 and 17 give the same results as calls on Code 3.7.

Abstract type members can be used to build more powerful abstractions apart from what is just needed for this project. See [38] for more details.

3.3.6 TypeTags replacing Manifest

The previous sections have shown some examples on how Scala Manifests, which are directly mapped to Java Manifests, are used to work around type erasure. However, Manfiest is currently deprecated as runtime type information mechanism, and is being replaced by a feature call TypeTag, which operates directly with types as the compiler understands them, not as raw strings. Nevertheless, Manifests are used for this project because one of the main libraries used, LMS, makes heavy use of it on its current version.

3.4 Lightweight Modular Staging

LMS [7, 43] is a Scala library for dynamic code generation. The main idea behind LMS, staging [27], is based on the observation that some computations can be lifted to the stages in which they will be performed less frequently or more information is available. The staging approach, although introduced initially as a set of compiler transformations, can be thought of as a method for embeddeding domain-specific languages [18].

LMS is just a library, so a DSL application is compiled together with the DSL implementation, giving place to an executable code generator. Then, once the code generator is run, the DSL application gets translated. Building DSLs with LMS can be broken down into the following phases:

1. Defining interface of the DSL
2. Implementing operations performed by DSL
3. Implementing code generator for the DSL to target language.

We will use the simple Vector DSL[2] as an example for explaining the steps for building DSLs. The DSL provides a generic type `VT[_]` and a set of operations such as vector addition or dot product. For brevity this section will focus on addition of two vector and multiplication of vector by a scalar.

Interface of the DSL

Firstly, the interface of the DSL must be defined:

```
1  trait VectorOps extends Base {
2       class VT[0]
3       def infix_+[0:Manifest](x: Rep[VT[0]], y: Rep[VT[0]]) =
4            vector_plus(x,y)
5       def infix_*[0:Manifest](x: Rep[VT[0]], y: Rep[0]) =
6            vector_tms_scalar(x,y)
7       def vector_plus[0:Manifest](x: Rep[VT[0]], y: Rep[VT[0]]):
8            Rep[VT[0]]
9       def vector_tms_scalar[T:Manifest](x: Rep[VT[0]], y: Rep[0]):
10           Rep[VT[0]]
11 }
```

Code 3.13: Interface of the Vector DSL

[2]Building Vector DSL is an assignment in the CS 442 course at Stanford University: *High Productivity and Performance with Domain Specific Languages in Scala,* given in Spring 2011 at Stanford University. Detailed information on the assignment can be found at http://www.stanford.edu/class/cs442/

The `VectorOps` trait on Code 3.13 defines the interface of our DSL, which consists of types and operations the programmer will use in the application. The basic type of the language is `VT[_]` represented here by an empty Scala class (line 2). In lines 4 and 5, operations for adding two vectors (`infix_+`) and multiplying vector elements by a scalar (`infix_*`) are defined. The bodies of these methods consist on calls to abstract methods `vector_plus` (line 4) and `vector_tms_scalar` (line 5). These abstract methods will be implemented in a later phase of the process. They bridge operations defined by the interface with their actual implementations.

The arguments taken by method `infix_+` are of type `Rep[VT]`, not `VT`. `Rep[+T]` is a higher-kinded type that represents a staged computation. For example, type `Rep[Int]` indicates that the staged computation will result in type `Int` in the next stage[43]. `VectorOps` is a trait mixed in with `Base`. `Base` is a part of LMS library.

Implementing DSL operations

Code 3.14 shows the exact implementation of the Vector DSL for this example. The implementation forms an expression tree[43].

```
1  trait VectorOpsExp extends VectorOps with Expressions {
2       case class VtPlus[O:Manifest](x: Exp[VT[O]], y: Exp[VT[O]])
3            extends Def[VT[O]]
4       case class VtTmsScalar[O:Manifest](x: Exp[VT[O]], y: Exp[O])
5            extends Def[VT[O]]
6       case class AXPY[O:Manifest](a: Exp[O], x: Exp[VT[O]], y:
7            Exp[VT[O]]) extends Def[VT[O]]
8       def vector_plus[O:Manifest](x: Exp[VT[O]], y: Exp[VT[O]]) =
9            (x,y) match {
10                case (x, Def(VtTmsScalar(y1,a))) => AXPY(a, y1, x)
11                case (Def(VtTmsScalar(x1,a)), y) => AXPY(a, x1, y)
12                case _ => VtPlus(x, y)
13      }
14      def vector_times_scalar[T:Manifest](x: Exp[VT[O]], y: Exp[O]) =
15           VtTmsScalar(x, y)
16 }
```

Code 3.14: Implementation of the Vector DSL

The implementation is defined in trait `VectorOpsExp` (line 1). It implements the interface of our DSL, `VectorOps`. The trait `Expressions` provides an infrastructure for implementing an Intermediate Representation (IR) of our DSL as expression tree. The tree contains IR nodes of our DSL.

The IR nodes are implemented as case classes (lines 2, 3 and 4). These classes take arguments of type `Exp[_]`. This LMS type constructor represents constants and sym-

bols. Each case class extends a parametrized type constructor `Def[_]`. This LMS type constructor represents definitions. Symbols are bound to definitions. The design choice of LMS is that each composite construct (such as case class `VtPlus`) refers to its parameters through symbols. Symbols are globally numbered so no redundant code occurs in the target application. This mechanisms also allow for some optimizations that will not be discussed here, such as common subexpression elimination.

Lines 6-12 show the body of methods `vector_plus` and `vector_times_scalar`. On Code 3.13 these methods were declared abstract. This is the way an implementation is bound to its interface. However, this binding is loose, meaning interface and implementation of a DSL are implemented as separate traits. The DSL implementer is able to provide several implementations for the same interface.

In lines 2-4, definitions are implemented as Scala case classes. They can be pattern-matched. This in turn allows the implementation of optimizations such as rewriting (lines 6-10). Here the IR node `AXPY`[3] is emitted if any of the arguments passed to the `+` operator is the result of a vector-scalar multiplication (lines 7 and 8). In other case (line 9) we emit a regular addition IR node.

DSL code generator to target language

Code 3.15 shows the code generator to C++ of the Vector DSL. The code generator is implemented as a trait that extends LMS' trait `CGen` (line 1). Trait `CGen` extends the core code generation traits with code generators for regular C code, so we do not have to redefine regular assignment or looping, for example. The core code generation traits provide an infrastructure for dynamic code generation such as method `quote`, which obtainins the symbol corresponding to a given particular definition.

```
1  trait VectorCppGen extends CGen {
2        val IR: VectorOpsExp
3        import IR._
4        override def emitNode(sym: Sym[Any], rhs: Def[Any]): Unit =
5              rhs match {
6              case VtPlus(x, y) => emitValDef(sym,
7                    quote(x) + " + " + quote(y))
8              case VtTmsScalar(x, y) => emitValDef(sym,
9                    quote(x) + " * " + quote(y))
10             case AXPY(a, x, y) => emitValDef(sym,
11                   quote(x) + ".axpy(" + quote(y) + ", " + quote(a) + ")")
12             case _ => super.emitNode(sym, rhs)
13       }
```

[3] $aX + Y$, where a is a scalar and X and Y are vectors. The multiplication and the addition can be performed on the same loop.

```
14 }
```

Code 3.15: C++ code generator for Vector DSL

In lines 2 and 3, there is the information about the intermediate representation and the implementation of the DSL interface. The `import` statement in line 3 injects the types of IR nodes into the scope of the code generator so it can refer to them through pattern matching (body of method `emitNode` in lines 6-9).

While LMS is traversing an expression tree of our DSL, each IR node is pattern-matched (lines 6-9). If the definition is matched (left-hand side of a `case` statement in line 6), C code is generated for a given IR node (right-hand side of the statement). If no match is found, the symbol and definition of the IR node are passed forwarded to the next mixed-in trait (or superclass) on the chain.

An example application

Once we have all the parts of a DSL (representation, implementation and code generation), our DSL is ready to translate applications into the target language. For example, Code 3.16 would be a valid application for the Vector DSL.

```
1  object VectorDSLApp {
2       trait SimpleApp {
3            val v = Vector(1, 2, 3)
4            val w = Vector(2, 5, 2)
5
6            val scaledv = 5 * v
7            val sum = v + w
8            val sum2 = scaledv + w
9
10           show(sum * sum2)
11      }
12 }
```

Code 3.16: Vector DSL example application

For simplicity, we will not show all the details of the operations not described in this section (like the `Vector(...)` constructor or the `show` operation. For a complete documentation about LMS, see the referenced material and its official website [7]. By the way, the code emitted after running the generator of the application is shown in Code 3.17.

```
#include <iostream>
#include "vectorDataStruct.h"
int main() {
      Vector x0(1, 2, 3);
      Vector x1(2, 5, 2);
      Vector x2 = x0+x1;
      Vector x3 = x0.axpy(5, x1); // Pattern recognized and optimized
      Vector x4 = x2*x3;
      for (size_t i = 0; i < x4.len(); ++i) std:cout << x4[i] << " ";
      std::cout << std::endl;
}
```

Code 3.17: C++ code generated for the Vector DSL example application

3.5 Scala-Virtualized

LMS allows programmers to stage any kind of computation resulting on an object of a particular type. However, the vanilla Scala compiler does not provide us with staging of control flow constructs such as `if-else` statements or loops.

The functionality of staging control flow is achieved through an extended Scala compiler called Scala-Virtualized[34]. The control flow constructs are compiled down into regular method calls. So, Scala statement `if(a) b then c` is compiled into function call `__ifThenElse(a, b, c)`. Statement `return a` is compiled into `__return(a)`. LMS requires all these features and thus DSLs need to be compiled with Scala-Virtualized. More examples and implementation details can be found in [34] and [43].

3.6 LLVM and Clang

LLVM[29] is a compiler for C-based languages that provides an infrastructure for program analysis, transformations and optimizations. LLVM implements different compilation phases. These phases can work upon program units of different granularities: module, function and basic block.

One of the key features of LLVM is that it compiles program code to bytecode that can be further translated into a human-readable intermediate representation. LLVM bytecode can be subject of a program transformation but the results can be easily inspected by analyzing this IR. The IR of LLVM is a virtual instruction set, that is fully typed, register-based, and independent from the source language and target architecture.

Clang[28] is a frontend for LLVM for C-based languages. It shares the same design goals as LLVM: Providing library-based infrastructure for indexing, static source analysis and

source-to-source transformation tools. Thanks to its extensible design, standalone tools that operate on program's abstract syntax tree can be developed. As a code generator for LLVM, Clang can be used for writing tools that operate on LLVM's IR.

For this project, Clang and LLVM are used to create a custom tools that allows the implementation of DFSSL to integrate OpenCL kernel files with the host application. Then type checking is performed at staging time, as well as many other security checks regarding input and output specification of parameters. These details will be further explained in Chapter 5.

CHAPTER 4

Data Flow SuperScalar Language

This chapter will explain DFSSL, the language we developed in this project. DFSSL is a language for abstracting the low-level details of data flow accelerator-based application development. It also provides safety elements regarding the interoperation between host and accelerator code. Tasks such as managing execution queues and setting kernel arguments explicitly, are hidden behind syntax of the language and managed by its compiler and the runtime architecture it targets. This in turn allows language designers to treat DFSSL as an intermediate layer upon which higher-level DSLs can be implemented.

DFSSL provides the following main features:

- Declaring NDRanges that represent `N`-dimensional workspaces for kernels.
- Binding and using containers of kernels (a regular .cl file containing kernel functions in OpenCL C)
- Declaring and initializing data buffers
- Executing kernels
- Specify tasks that should run asynchronously on the host side
- Use external libraries when the target platform is known

The complete DFSSL manual and user guide come with the main distribution of its compiler (see Appendix A for further details about how to install DFSSL and use all of its features including the input/output system or advanced math support). This chapter will focus on explaining the most important features of DFSSL to justify its value and existence.

4.1 NDRanges

DFSSL has one basic parallelism source on the accelerator side. Just like in regular OpenCL, NDRanges are the reffer to the way a kernel is run at the accelerator. Separating this concept from the kernel execution itself allows us to reason about the kind

of traversal a particular kernel will perform over the data. Though outside the scope of this project, having NDRanges as a standalone entity would allow our compiler to reason and optimize how kernels are run by fusing different kernels together, for example. Code 4.1 shows a usage example of NDRange expressions.

```
// Local size: 64, Global size: 64 * 32
val rx = Range1D(64, 32)
val rxy = Range2D(rx, rx)
val rxyz = Range3D(rx, Range1D(128, 8), rx)
```

Code 4.1: Example of use NDRange expressions

NDRanges are stored as a regular Scala value. Then a two-dimensional range can be expressed in terms of an already present one-dimensional range or a new range declared in-place. LMS will take care of reusing NDRange objects if they are equivalent even if they are declared twice.

4.2 Kernels and Kernel Containers

A kernel container represents a file that contains kernels written in OpenCL C. In Code 4.2 we can see how kernel containers are used in DFSSL.

```
val kc = KernelContainer("/path/to/kernels.cl")
```

Code 4.2: Example of use of KernelContainer expression

Once a kernel container is declared, kernels can be retrieved from it using the `Kernel` operations. For example, given the declaration of `kc` in Code 4.2, we would use it as shown in Code 4.3. The second list of arguments shows the direction of each argument inside the kernel body.

```
// This retrieves the 3-parameter "add" kernel
// from kc, namely myAdd
val myAdd = Kernel(kc, "add")(In, In, Out)
// Put on mySub the 2-parameter kernel named
// "sub" inside kc
val mySub = Kernel(kc, "sub")(InOut, In)
```

Code 4.3: Example of use of Kernel expression

The implementation of DFSSL ensures that all kernel calls match their corresponding prototypes on the containers, so any mismatch on the kernel name or argument types will be signaled by the DFSSL implementation.

4.2.1 Optional default NDRange when creating kernels

A kernel needs an NDRange to be specified when it is run. If the user wants to specify a particular NDRange traversal when creating the kernel, the kernel can be run just referring to the "default" ndrange without requiring any more details. This default range can be overridden at any particular call site, so it is always safe to specify a default traversal when there is a clear good pattern for a particular kernel. An example of declaring a default ndrange for a kernel is shown in Code 4.4.

```
val mySub = Kernel(kc, "sub")(InOut, In) withDefaultRange Range1D(8, 16)
```

Code 4.4: Specifying a default NDRange for a Kernel

4.3 Data Buffers

Data buffers represent our well-known plain data containers. Thanks to the expressive power of Scala, buffers provide type inference when initialized with immediate values. A buffer can be just allocated, by specifying its size and element type. Code 4.5 shows how to use data buffers.

```
// Allocate space for 4096 floating point values
val buf1 = Buffer.fill[Float](4096)
// Buffer of 4 integer values
val buf2 = Buffer(5, 8, 21, -3)
```

Code 4.5: Data buffer initialization

Just like regular C arrays, buffer elements can be randomly accessed and/or update in constant time like shown in Code 4.6.

```
val buf1 = Buffer.fill[Float](4096)
buf1(4) = 4.32f
buf1(7) = buf(2) + 6.23f
```

Code 4.6: Data buffer random access

4.3.1 Applying functions to all buffer elements

In Scala, one of the most common operations over a list is `map`. This operation applies a particular function to a list and returns the new list resulting from it. In the case of buffers, the `map` function does not return a new buffer, but updates the contents of it with the result of applying a given function to all of its elements. A simple example of mapping a function to a buffer would be to initialize all elements with random values[1], like shown in Code 4.7.

```
val myData = Buffer.alloc[Float](1024)
myData map (_ => rand)
// myData now contains 1024 random floating point values
```

Code 4.7: Updating a buffer with a function

4.4 Executing kernels

The features explained in sections 4.1, 4.2 and 4.3 converge to result in actual kernel executions. To run a kernel, we just cal it specifying the actual argument like we would expect from a regular function like shown in Code 4.8. The only additional details that distinguishes a kernel from a raw function is the need of specifying the NDRange to be used by the runtime scheduler.

```
1  val range1 = Range1D(4, 1)
2  val input1 = Buffer(4, 1, 2, 5)
3  val input2 = Buffer(2, 6, -2, 9)
4  val out = Buffer.fill[Int](4)
5  val kc = KernelContainer("kernels.cl")
6  val addk = Kernel(kc, "add")(In, In, Out)
7  addk(input1, input2, out) using range1
8
9  // Implementation of procedure "show" not shown for simplicity
10 show(out)
```

Code 4.8: DFSSL simple application

As we can see, the whole application is more concise and clear by just the fact that all platform selection and automatable execution details are hidden by the compiler and the runtime.

[1] rand is a built-in function of DFSSL

In case a default range was specified when the kernel was created, we can just run the kernel specifying the default range is to be used as Code 4.9 shows.

```
val range1 = Range1D(4, 1)
val addk = Kernel(kc, "add")(In, In, Out) withDefaultRange range1
addk(input1, input2, out) using default
```

Code 4.9: DFSSL simple application with default range

4.5 The OpenCL example revisited again

In Chapter 3 we showed a program (see Code 3.1) implementing a vector addition that initialized both vectors on different GPUs. We then saw how OmpSs refines the approach by relying on a runtime system to schedule tasks asynchronously in Code 3.5. To illustrate how DFSSL can produce complex and even more neat applications, Code 4.10 shows the equivalent program to Code 3.1 written in DFSSL.

```
1  val local = 1024
2  val global = local*5*1024
3  val range = Range1D(local, global)
4  val buf1 = Buffer.fill[Float](global)
5  val buf2 = Buffer.fill[Float](global)
6  val kc = KernelContainer("kernels.cl")
7  val kinit = Kernel(kc, "init")(Out)
8  val kadd = Kernel(kc, "add")(InOut, In)
9
10 kinit(range)(buf1)
11 kinit(range)(buf2)
12 kadd(range)(buf1, buf2)
```

Code 4.10: Simple OpenCL example written in DFSSL

The first thing to note is that now the kernels are on a separate file (which is also possible in OmpSs). Therefore, file `kernels.cl` contains the OpenCL kernels without any pragma specification. In DFSSL, the programmer needs not to be aware of any low level detail while the implementation is allowed to reason about parallelism and NDRanges. Note that we do not mention any devices, platforms, queues nor memory transactions. These decisions are to be taken by the Nanox++ runtime system. Let us comapre between our first OpenCL application (Code 3.1) and its DFSSL version Code 4.10:

- The first three lines correspond to lines 27 and 28 in OpenCL, where the size of the ranges is specified.

- Buffer declarations on lines 5 and 6 correspond to all the buffer related code from lines 30 to 35 in OpenCL.
- Line 8 is the equivalent to all the code from lines 40 to 45 where the programs are created and compiled.
- Lines 10 and 11 create kernels just like lines 46 and 47 do on the OpenCL code.
- The last three lines are the kernel calls present from lines 48 to 65 in OpenCL. Note how argument specification and buffer transfers are all implicit on the DFSSL side.

4.6 Host asynchronous tasks

The host application main goal is to submit work to the accelerators. However, there are many cases in which some kind of work has also to be done at the host side. In general, the most frequent places where the host has to perform some tasks apart from submitting work to accelerators are:

- Performing independent computations while the accelerators are busy.
- Submitting tasks to be done after the accelerators finish (very important, if the results of kernels are used outside tasks, it is not guaranteed that they will be already available).
- Implement algorithms that require recursivity or are not good for accelerators for any reason.

DFSSL tasks work exactly like OmpSs tasks and they look very similar. An example is shown in Code 4.11.

```
Task(centroids, sumPoints, nPoints)(Out(4 * K), In(4 * K), In(N)) {
    (0 until K) foreach { i =>
      (0 until 3) foreach { j =>
          centroids(4 * i + j) = sumPoints(4 * i + j) / nPoints(i)
      }
    }
}
```

Code 4.11: DFSSL Host tasks

The first group of arguments of for the `Task` construct are the actual parameters this task binds. The second group of argument specifies the directions of these arguments in the task implementation and their size (in case on non-buffer arguments, we can just provide `In`, `Out` or `InOut` like in kernels.

4.7 Using external libraries

DFSSL is a language for imlpementing DSLs in an easy and convenient way. Therefore, usage of already existing libraries can be extremely useful when it comes to provide certain functionalities. Common examples of external libraries used in C/C++ HPC applications are FFT, BLAS or LAPACK. Libraries written for C or C++ are fully compatible with DFSSL applications. A basic example of using external libraries is shown in Code 4.12.

```
1  // External includes
2  include("atlas/clapack", true)
3  lib("lapack_atlas")
4
5  // Defining external functions
6  def spotrf = LibFun("clapack_spotrf")
7
8  // Defining External Symbols
9  def ColMajor = LibSym[Int]("CblasColMajor")
10 def Lower = LibSym[Int]("CblasLower")
11
12 val A = Buffer.alloc[Float](N * N)
13 spotrf(ColMajor, Lower, N, A, N)
```

Code 4.12: DFSSL external library support

Let us discuss the code snippet above line by line:

- Line 2 tells the code generator to include `atlas/clapack.h` to the generated code. The second argument is set to true because this header is written for C and it need to be wrapped with a `extern 'C'` block.
- Line 3 tells the linker to add `liblapack_atlas` to the symbol resolution stage.
- Line 6 defines a function that is guaranteed to be available at runtime.
- Lines 9 and 10 define symbols that are guaranteed to be available at runtime.
- Line 13 uses all the previously defined external symbols and the buffer created in line 12.

4.8 Kernel usage safety mechanisms

Kernel files written for DFSSL have no special difference with respect to kernel files written for regular OpenCL applications, but DFSSL adds some security features at compile time. In regular OpenCL, the kernels are compiled at runtime, so any compilation error or simple type mismatches will produce failed executions. Such errors are difficult to debug and require a considerable effort to be detected and fixed. DFSSL helps the programmer by compiling the kernel files before the code generator is produced, so all compilation errors are shown long before running anything. Moreover, when a kernel is called from the DFSSL application, the implementation checks whether all the real parameters type-match with the corresponding kernel declaration. For the rest of this section, we will use the file shown in Code 4.13, called `kernels.cl`.

```
kernel void CoolKernel(global float* a, global float* b){
    unsigned int i = get_global_id(0);
    a[i] = 1.0f;
    b[i] = 2.5f;
}
```

Code 4.13: Simple OpenCL C kernel file

Apart from compilation errors, the DFSSL kernel usage checker tests por three kind of programming mistakes:

4.8.1 Undefined kernel error

This error is triggered when the application is trying to create a kernel whose corresponding name in the container is K and no kernel function exists inside the container named K. For example, let us say there is a typographic error on the name when creating a kernel, like shown in Code 4.14.

```
1 val myCoolContainer = KernelContainer("kernels.cl")
2 val myCoolKernel = Kernel(myCoolContainer, "wrong")(InOut, In)
```

Code 4.14: Trying to create an undefined kernel

Whenever this error occurs, the DFSSL code generator will trigger error message shown in Code 4.15:

```
Error in file Code5p7.scala, line 2: No kernel named "wrong" was found
```

Code 4.15: Trying to create an undefined kernel

4.8.2 Kernel parameter list mismatch error

This is the error issued by the type checker when it finds differences between the types of the formal parameters of a given kernel implementation and the actual parameters used when calling the kernel. This kind of error is triggered when the number of arguments does not match between call and implementation sides or when their types are wrong, like shown in Code 4.16.

```
1 val myInoutBuffer = Buffer.alloc[Float](2*16)
2 val myInoutBuffer = Buffer.alloc[Int](2*16)
3 val myCoolContainer = KernelContainer("kernels.cl")
4 val myCoolKernel = Kernel(myCoolContainer, "CoolKernel")(InOut, In)
5 val myRange = Range1D(2, 16)
6 myCoolKernel(myInoutBuffer, MyInBuffer) using myRange
```

Code 4.16: Sample program that triggers a type mismatch error

In this case, the code generator will then trigger a detailed message, like error message shown in Code 4.17:

```
Error in file Code5p10.scala, line 6:
Type mismatch at kernel named "CoolKernel"
Call site actual parameters are: (float*, i32*)
Kernel file formal parameters are: (float*, float*)
```

Code 4.17: Kernel type mismatch error

4.8.3 Kernel called with no ND-Range

As explained in section 4.2, kernels require the user to specify either a default ND-Range when creating a kernel or a specific ND-Range each time the kernel is run. If the user tries to run a kernel K with its default range and no default range has been specified for K when it has been created, like shown in Code 4.18 the DFSSL compiler will trigger an error.

```
1 val myInoutBuffer = Buffer.alloc[Float](2*16)
2 val myInoutBuffer = Buffer.alloc[Float](2*16)
3 val myCoolContainer = KernelContainer("kernels.cl")
4 val myCoolKernel = Kernel(myCoolContainer, "CoolKernel")(InOut, In)
5 myCoolKernel(myInoutBuffer, MyInBuffer) using default
```

Code 4.18: Sample program that triggers a no default range error

When such a situation happens, the code generator will show an explicit error like the one shown in Code 4.19.

```
Error in file Code5p12.scala, line 5: Called kernel "CoolKernel"
using the default range when no default range was specified for it.
This only available if a default ndrange is specified when the
kernel is created using the withDefaultRange construct.
```

Code 4.19: Error shown when no default range is specified

Chapter 5

Implementation

The DFSSL compiler was built on top of a set of tools and libraries. This chapter will show how each of the key features of the language was implemented.

5.1 Kernel usage safety mechanism

As explained in section 4.8, the DFSSL compiler checks whether kernel calls are going to be successful at runtime. When a kernel container is declared, the implementation calls an LLVM-based tool written in C++ ad-hoc for the DFSSL compiler. This tool, given a file containing kernel functions, it returns their names and parameter lists. This output is captured by the compiler and stored in a symbol map that will be associated to this particular kernel container.

```
1  val sMap = Map[String, List[(String, String)]]()
2  // Executing parser and storing its output
3  val parsed = ("smapgettertool " + kernelfile).!!
4  val st = new StringTokenizer(parsed)
5  while (st.hasMoreTokens()) {
6      if (st.nextToken().equals("\\K")) {
7        val kname = st.nextToken()
8        val nargs = st.nextToken()
9        val plist = MutableList[(String, String)]()
10       for (i <- 1 to nargs.toInt) {
11           plist += Tuple2(st.nextToken(), st.nextToken())
12       }
13       sMap += (kname -> plist.toList)
14     }
15 }
```

Code 5.1: Initialization of the kernel file symbol map

In Code 5.1 we can see how the symbol map is initailized for a given kernel container. Lines 3 uses the Scala process library to call an external program and capture its output. The rest of the code snippet is a simple parsing loop where each kernel starts with the

`\K` escape sequence. The following subsections will show how the compiler uses this symbol map to detect and signal all the errors shown in section 4.8.

5.1.1 Undefined kernel error

When a kernel is created, the container in which is implemented is required as a mandatory parameter. The symbol map of this kernel container is accessed whenever a kernel is created to check if the name specified by the user is one of the kernel names in the symbol map. If the kernel is not found in the given container, the error is shown and the compilation stops.

5.1.2 Kernel parameter list mismatch error

To detect errors in parameter lists like incorrect number of arguments or type mismatch between one or more arguments, the compiler checks all kernel calls. Whenever a kernel is run, the actual parameter list at call site is retrieved and compared against the formal parameter list in the symbol map. If any mismatch occurs, the error is shown and the compilation halts.

5.1.3 Kernel called with no ND-Range

If the user tries to run a kernel with its default range when such kernel has no default range, the compiler stops and shows the error. Detecting this error does not require the symbol map, just checking whether a kernel has a default range when the user tries to use such range.

5.2 Dynamic task and kernel scheduling

By requiring the user to specify the directions (input, output or both) of each argument when creating a kernel or a task, the compiler is able to just forward this information as OmpSs pragmas when it generates code. When the OmpSs generated code is compiled, Mercurium transforms these pragmas into calls to the Nanox runtime library, which dynamy schedules workload among accelerators and host. In conclusion, DFSSL just forwards the information from the top-level application to the runtime.

5.3 Code generation for OmpSs

Code generation to OmpSs has been achieved by defining a code generator using the LMS library. This section will show how the DFSSL compiler maps its high-level constructs to C++ OmpSs code.

5.3.1 Buffer function mapping

The buffer mapping syntax shown in subsection 4.3.1 is translated as a parallel for given there can not be inter-iteration dependencies. Therefore, the code shown in Code 5.2 will be translated to the code shown in Code 5.3.

```
1 val buf = Buffer(5, 7, 3, 2, 73)
2 buf map {_ + 10}
```

Code 5.2: Mapping a simple function to all the elements of a buffer

```
1 int x0 = {5, 7, 3, 2, 73};
2 #pragma omp parallel for
3 for (size_t i = 0; i < 5; i++) {
4     int t = x0[i];
5     x0[i] = t + 10;
6 }
```

Code 5.3: Parallel loop generated for a buffer mapping

5.3.2 Generation of header files for kernel functions

In some cases, all the information related to how a kernel is scheduled and run (input/output, NDRanges...) may be available at code generation time. Using this chance, the DFSSL compiler creates a headerfile with the declaration of such kernels and inserts the pragma annotations directly. This way, the generated application code includes this headerfile and calls the kernel directly without specifying the pragmas, which are already taken from the declaration. Code 5.4 shows an example of an automatically generated header file for a kernel whose input sizes and NDRanges are known at compile time.

```
// Header file automatically generated by DFSSL
#ifndef _KMEANS_HEADER_
#define _KMEANS_HEADER_
#pragma omp task in(x0, x1, [4096] x2, [12] x3) out([1024] x4)
#pragma omp device target(opencl) ndrange(1, 257*4, 4)
kernel void asgn_closest(int x0, int x1, float* x2, float* x3, int* x4);
#endif // _KMEANS_HEADER_
```

Code 5.4: DFSSL-generated header file

5.4 External library support

To support external libraries, the DFSSL compiler provides constructions that just forward the symbol names and the method calls to the generated code. The headers to be included are placed together with the fixed includes on the generated code and the libraries are used by the linker as usual when the final binary is to be generated. For example, the application shown in Code 5.5 will be translate to the C++ OmpSs code in Code 5.6.

```
include("utility")
def swap = LibFun("swap")
val b = Buffer(6, 5, 4, 3, 2, 1)
swap(b(1), b(5))
```

Code 5.5: Using an externally defined function

```
// Rest of DFSSL includes omitted
#include <utility>
int main(int argc, char *argv[]) {
    using namespace std;
    int b[] = {6, 5, 4, 3, 2, 1};
    swap(b[1], b[5]);
}
```

Code 5.6: Generated code for external function usage

CHAPTER 6

Results

In order to test the whole DFSSL implementation, we have coded different application examples. Even though the main goal of DFSSL is serving as target language for higher level DSLs, we have tested the code generator directly with short applications for simplicity. This chapter will show the DFSSL code corresponding to each application we tested. Moreover, the OmpSs + OpenCL code generated for each application is shown in Appendix B.

6.1 N-Bodies physical simulation

The N-Bodies physical simulation we implemented is a basic variant which consists on simulating the interaction of N spherical bodies the mass of each applies a force to all the others, causing its position and velocity to vary at each time step. The application simulates a finite number of time steps. Let us take a look at the DFSSL application, shown in Code 6.1.

```
1  def WARP_SIZE = 32
2  def N_ITERATIONS = 3600 * 24 // One day in earth, must be an even number
3  def TIME_STEP = 1.0f // In seconds
4  def usage(app: Rep[String]) = { stdout << "Usage: "; stdout << app;
5      stdout << " <input> [output]\n" }
6  def checkArgs = if (argc < 2) {
7      usage(argv(0))
8      exit(1)
9  }
10 // Returns the number of bodies and creates the buffers
11 def getInput(inpath: Rep[String]) = {
12     val in = FileIn(inpath)
13     val n = in.get[Int]
14     val masses = Buffer.alloc[Float](n)
15     val positions = Buffer.alloc[Float](n * 4)
16     val velocities = Buffer.alloc[Float](n * 4)
17     (0 until n) foreach { i =>
```

```
18          in >> masses(i)
19          val j = 4 * i
20          in >> positions(j)
21          in >> positions(j + 1)
22          in >> positions(j + 2)
23          in >> velocities(j)
24          in >> velocities(j + 1)
25          in >> velocities(j + 2)
26          masses(i) = rand(5, 48)
27      }
28      (n, masses, positions, velocities)
29  }
30  def showOutput(n: Rep[Int], masses: Rep[Buffer[Float]],
31      positions: Rep[Buffer[Float]], velocities: Rep[Buffer[Float]]) = {
32      Task(n, masses, positions, velocities)
33          (In, In(n), In(4 * n), In(4 * n)) {
34          (0 until n) foreach { i =>
35              stdout << masses(i); stdout << " "
36              val j = 4 * i
37              stdout << positions(j); stdout << " "
38              stdout << positions(j + 1); stdout << " "
39              stdout << positions(j + 2);
40              stdout << " "
41              stdout << velocities(j); stdout << " "
42              stdout << velocities(j + 1); stdout << " "
43              stdout << velocities(j + 2)
44              stdout << endl
45          }
46      }
47  }
48  def createSwappingBuffers(n: Rep[Int]) = {
49      val positions2 = Buffer.alloc[Float](n * 4)
50      val velocities2 = Buffer.alloc[Float](n * 4)
51      (positions2, velocities2)
52  }
53  def createNBodiesKernel(n: Rep[Int]) = {
54      val kc = KernelContainer("nbody.cl")
55      Kernel(kc, "nbodyStep")
56          (In, In, In(n), In(4 * n), Out(4 * n), In(4 * n), Out(4 * n))
57          withDefaultRange Range1D(WARP_SIZE, n / WARP_SIZE + 1)
58  }
59  def nbodiesrun() = {
60      checkArgs
```

```
61      // Getting input (number of bodies, buffer of masses, ...)
62      val (n, masses, positions1, velocities1) = getInput(argv(1))
63      // Creating two additional buffers to swap in/out on each iteration
64      val (positions2, velocities2) = createSwappingBuffers(n)
65      val nBodyStep = createNBodiesKernel(n)
66      (0 until N_ITERATIONS) foreach { i =>
67          if (i % 2 == 0) nBodyStep(TIME_STEP, n, masses, positions1,
68              positions2, velocities1, velocities2) using default
69          else nBodyStep(TIME_STEP, n, masses, positions2, positions1,
70              velocities2, velocities1) using default
71      }
72      showOutput(n, masses, positions1, velocities1)
73  }
```

Code 6.1: N-Bodies problem solved in DFSSL

The first thing we notice is the input/output system of DFSSL. This system has not been shown in the corresponding chapter because of its similarity with the standard C++ IO system. Note that when the application is reading the input in function `getInput` in line 11, no tasks or kernels have been issued to run yet, so the code can be directly run by the host. However, the `showOutput` in line 30 wraps its actions inside a task because it uses the results obtained from the execution of the kernels at lines 67 and 69.

Each time step requires the results from the previous time step, so the application uses double buffering to update the result from frame to frame. Function `createSwapping-Buffers` in line 48 creates the additional swapping buffers.

Finally, in line 53, function `createNBodiesKernel` creates a kernel container with the kernel file and returns a kernel with a default range associated. This kernel is called on the main application loop (line 66) swapping the corresponding buffers. When the loop ends, the output is shown in line 72.

6.2 K-Means Clustering algorithm

There is a wide variety of algorithms for clustering data sets. The problem consists in grouping N-dimensional points into K exact clusters where K is known. The objective is to reduce the global entropy of the clustering, meaning we want to minimize the sum of the distance of each point to its corresponding cluster centroid. The K-means algorithm is basically a convergence loop where the global entropy is guaranteed to decrease at each step. Each step is composed of two basic actions, the first one is determining the closest centroid to each data point. Then the next action is to recompute the centroid of each cluster. Afterwards, the algorithm proceeds with the next iteration with the new centroids until the result converges or a fixed number of iterations is run. The algorithm implemented in DFSSL is shown in Code 6.2.

```
def WARP_SIZE = 32
def N_ITERATIONS = 5000
def RANDOM_SEED = 1
def K = 3
def N = 1024
def initData = {
    val data = Buffer.alloc[Float](4 * N)
    val centroids = Buffer.alloc[Float](4 * K)
    data map (_ => rand)
    centroids map (_ => rand)
    (data, centroids)
}
def kernelFileContainer = KernelContainer("kmeans.cl")
// Creates the assign kernel and allocs the output buffer it needs
def initAssignKernel = {
    val kAssignClosest = Kernel(kernelFileContainer, "assign_closest")
        (In, In, In(4 * N), In(4 * K), Out(N))
        withDefaultRange Range1D(WARP_SIZE, N / WARP_SIZE + 1)
    val assignations = Buffer.alloc[Int](N)
    (kAssignClosest, assignations)
}
def showOutput(centroids: Rep[Buffer[Float]]) = {
    def sp = stdout << " "
    Task(centroids)(In(4 * K)) {
        (0 until K) foreach { it =>
            val i = 4 * it
            stdout << "Centroid "; stdout << it; stdout << ": "
            stdout << centroids(i); sp
            stdout << centroids(i + 1); sp
            stdout << centroids(i + 2)
            stdout << endl
        }
    }
}
def computeCentroids(data: Rep[Buffer[Float]],
    assignations: Rep[Buffer[Int]], centroids: Rep[Buffer[Float]]) {
    val sumPoints = Buffer.alloc[Float](4 * K);
    val nPoints = Buffer.alloc[Int](K);
    sumPoints map { _ => 0 }
    nPoints map { _ => 0 }
    (0 until N) foreach { i =>
        val cc = assignations(i) // Current cluster
        val cc4 = cc * 4
```

```
44          val i4 = i * 4
45          // Counting and adding points of each cluster
46          Task(sumPoints(cc4), sumPoints(cc4 + 1), sumPoints(cc4 + 2),
47              nPoints(cc), data(i4), data(i4 + 1), data(i4 + 2))
48              (InOut, InOut, InOut, InOut, In, In, In) {
49                  sumPoints(cc4) = sumPoints(cc4) + data(i4)
50                  sumPoints(cc4 + 1) = sumPoints(cc4 + 1) + data(i4 + 1)
51                  sumPoints(cc4 + 2) = sumPoints(cc4 + 2) + data(i4 + 2)
52                  nPoints(cc) = nPoints(cc) + 1
53          }
54      }
55      // Computing centroids
56      Task(centroids, sumPoints, nPoints)(Out(4 * K), In(4 * K), In(N)) {
57          (0 until K) foreach { i =>
58              (0 until 3) foreach { j =>
59                  centroids(4 * i + j) = sumPoints(4 * i + j) / nPoints(i)
60              }
61          }
62      }
63  }
64  def kmeansrun() = {
65      set_random_seed(RANDOM_SEED)
66      val (data, centroids) = initData
67      val (kAssignClosest, assignations) = initAssignKernel
68      (0 until N_ITERATIONS) foreach { i =>
69          kAssignClosest(N, K, data, centroids, assignations) using default
70          computeCentroids(data, assignations, centroids)
71      }
72      showOutput(centroids)
73  }
```

Code 6.2: K-Means clustering algorithm in DFSSL

The loop of the application just calls 2 important functions. To show an example of hybrid host + accelerator side computing, the `kAssignClosest` function is implemented as a OpenCL kernel, while the `computeCentroids` procedure is written as a DFSSL task that will run on the host side.

The data set is initialized with random values in the `initData` function at line 6. Then, function `initAssignKernel` creates the corresponding kernel and the buffer that will store the output at each iteration.

Finally, function `showOutput` shows the final centroids obtained for the K clusters. Note again that since this function requires the result from the tasks and kernels run at the main loop, its code is scheduled as a task specifying the directions of each argument.

6.3 Cholesky decomposition of a matrix

The Cholesky decomposition of a matrix is the equivalent for matrices of what the square root is for scalars. More specifically, the Cholesky decomposition of a matrix A is a matrix C such that $CC^T = A$. In our example, we have implemented a computationally interesting variant called the Cholesky-Banachiewicz algoritm. The memory access pattern of this algorithm makes it very suitable for data reuse procedures. Therefore, using a LAPACK implementation for C, we implement this algorithm at a block level. The good thing about using LAPACK is that we can express the operations in terms of submatrices (or tiles) of the original matrix to get better performance and reuse their implementation. For this particular example, shown in Code 6.3, no accelerators nor tasks have been used. We just want this example to show how to define and use external libraries inside DFSSL.

```
1  // External includes
2  def includeLAPACK = {
3      include("atlas/cblas", true)
4      include("atlas/clapack", true)
5      lib("lapack_atlas")
6      lib("blas")
7  }
8  // Defining external functions
9  def spotrf = LibFun("clapack_spotrf")
10 def strsm = LibFun("cblas_strsm")
11 def sgemm = LibFun("cblas_sgemm")
12 def ssyrk = LibFun("cblas_ssyrk")
13 // Defining External Symbols
14 def ColMajor = LibSym[Int]("CblasColMajor")
15 def Lower = LibSym[Int]("CblasLower")
16 def Right = LibSym[Int]("CblasRight")
17 def Trans = LibSym[Int]("CblasTrans")
18 def NoTrans = LibSym[Int]("CblasNoTrans")
19 def NonUnit = LibSym[Int]("CblasNonUnit")
20 def RAND_MAX = LibSym[Int]("RAND_MAX")
21 def BS = 3; // Block size
22 def NB = 2; // Number of blocks
23 def N = BS * NB; // Total number of elements in a row
24 def show(A: Rep[Buffer[Float]]) = (0 until N) foreach { i =>
25     (0 until N) foreach { j => stdout << A(i * N + j); stdout << " " }
26     stdout << endl
27 }
28 def initRandMatrix = {
29     val A = Buffer.alloc[Float](N * N)
```

```
    A map { _ => val f: Rep[Float] = rand; f / RAND_MAX }
    // Zeroing lower triangle
    (0 until N) foreach { i =>
        (0 until i) foreach { j => A(i * N + j) = 0.f }
    }
    // Making A positive definite
    (0 until N) foreach { i => A(i * N + i) += N }
    A
}
def cholesky(A: Rep[Buffer[Float]]) = {
    // Pointer to the element (i,j) on the matrix A
    def AREF(i: Rep[Int], j: Rep[Int]) = A + i * N * BS + j * BS
    (0 until NB) foreach { k =>
        spotrf(ColMajor, Lower, BS, AREF(k, k), N)
        // Divide all the blocks on the same row by the
        // Cholesky factorization of the diagonal element
        (k + 1 until NB) foreach { i =>
            strsm(ColMajor, Right, Lower, Trans,
                NonUnit, BS, BS, 1.0, AREF(k, k), N, AREF(k, i), N)
        }
        // Updating trailing matrix
        (k + 1 until NB) foreach { i =>
            (k + 1 until i) foreach { j =>
                sgemm(ColMajor, Trans, NoTrans, BS, BS,
                    BS, -1.0f, AREF(k, i), N, AREF(k, j),
                    N, 1.0f, AREF(j, i), N)
            }
            ssyrk(ColMajor, Lower, NoTrans, BS, BS,
                -1.0f, AREF(k, i), N, 1.0f, AREF(i, i), N)
        }
    }
}
def Choleskyrun() = {
    includeLAPACK
    val A = initRandMatrix
    stdout << "Matrix A:\n"
    show(A)
    stdout << endl
    cholesky(A)
    stdout << "Cholesky Factorization of A:\n"
    show(A)
}
```

Code 6.3: Cholesky decomposition of a matrix in DFSSL

The function `includeLAPACK` includes the necessary headers for the compiler and the libraries for the linker. Then, from lines 9 to 20 we define the symbols our application will use from these libraries or are assumed to be available at global scope.

The initialization and output is very similar to the other examples, only that in the output there is no need for tasks because no code runs concurrently in this particular example.

CHAPTER 7

Impact, Plan and Cost Analyses

In this chapter, the most formal aspects of this project's management are explained. The first main section goes through the professional context and the social impact of the project. Then, the next section details the planning we followed along the almost four months of development. Finally, the last section shows the overall cost of the project and justifies its amortization in the middle term.

7.1 Context

Since this project consists on developing a system featuring an intermediate language that eases the production of DSLs, its context is closely bound to scientific programming for HPC systems. More specifically, the context of the project are research environments that use supercomputers to obtains scientific results. Given that the tools we will provide are aimed at improving research processes, its impact in the long term can be very diverse. Such a framework could be used to improve many different development processes, ranging from an improvement on industrial, medical or environmental analysis processes to an increase of productivity in high performance software. In general, any research group from the area of HPC can offer the same services reducing costs, maximizing productivity and efficiency by using the tools of this project. The following subsections describe the kind of users and professionals this project is intended to involve in.

7.1.1 Final users

The final users of our DSL production system are computer architecture experts that know how to translate any language expressing computational constructs to efficient code for supercomputers. Since our system is completely modular, any improvement on the implementation on the system is ready to be used by all the implemented DSLs. This way, computer architecture experts can work on improving the compilation pipeline without interrupting domain experts already working with their DSLs.

7.1.2 Indirect users

Domain experts themselves are the ones making the most out of our system, given that they get DSLs that ease their everyday work, letting them focus on their scientific problems rather in than the programming problems. As the previous section stated, these indirect users will be able to work uninterruptedly in their applications while computer architecture experts keep on improving the generated for their DSLs.

7.2 Planning

The development is composed of 9 different and clear stages. Each stage corresponds with a task. The resources used to develop the project are independent from any task in particular, so we will not make any concrete reference to which resources are used on each task. These resources, as stated in previous chapters, are just a basic programming set, an internet connection and our Redmine server. The rest of this section details these nine tasks and its approximate duration in business days. The initial planning was also done in business days, because then we have the advantage that if any problem occurs, we have the weekend margins to correct and fix it so we stay on schedule as much as possible.

7.2.1 Previous study of available tools - 10 days

In this first stage we studied the different alternatives we had to implement DFSSL, our intermediate language for developing DSLs. The options were very different and ranged from extensible and/or virtualizable languages to syntax transformation tools. By the end of this task, we had a clear decision over the tool set we were going to use for our purpose (see Chapter 3).

7.2.2 Interface design - 8 days

Once we decided which tools were the settlement for our project, we knew which kind of interface we could offer. Therefore, we designed the syntax and its semantics paying special attention to the level of detail we exposed to the users of DFSSL to make it as easy as possible to use.

7.2.3 Interface validation - 2 days

We spent a couple of days in trying to formalize problems and examples in terms of DFSSL. This way, we made sure the syntax of the language is sufficient and adequate. Another advantage of this early testing of the interface was that design errors were spotted and handled before implementing anything.

7.2.4 Semantic core - 15 days

This stage focused on providing the semantics implementation for the interface we designed. In more detail, the semantic core is the software component where we implement how applications written in DFSSL are expressed in terms of an unambiguous and unique tree representation.

7.2.5 Basic optimizations - 5 days

This task consisted in providing basic optimizations like automatic parallelization of safe sections and more. Chapter 5 explains these optimizations in further detail.

7.2.6 Code generation - 10 days

The code generation layer is where we defined how DFSSL is translated to the language we are able to compile and run on machines like plain OpenCL or OmpSs with OpenCL to make use of the Nanox runtime system [9].

7.2.7 Static checking - 5 days

The static checking system provided by the tools we use is enough to type check the host application, but not across the code run on the host and the accelerator respective sides. We dedicated five days to implementing a LLVM-based tool that the DFSSL compiler uses to see if all kernel calls from the host code match with their actual implementations on the OpenCL C code run by the accelerators.

7.2.8 Verification - 15 days

Once we had a finished stable implementation of DFSSL, we tested it along three weeks in order to verify the system complies to the requirements stated as our objectives. This stage is where we implemented full real running examples to test and validate the whole system.

7.2.9 Documentation - 15 days

Finally, the last three weeks of the project were dedicated to writing this document, its presentation and revising it over and over before the final delivery.

7.3 Action plan

This section will analyze the dependencies between the tasks just mentioned along the previous section and try to estimate a minimum/maximum completion time for the whole project.

7.3.1 Critical path analysis and gaps

In theory, the project had a clear parallelization chance at the implementation tasks. This is because each block of functionalities is independent from the other. Figure 7.1 shows the critical path analysis with its corresponding start/end limits for each task.

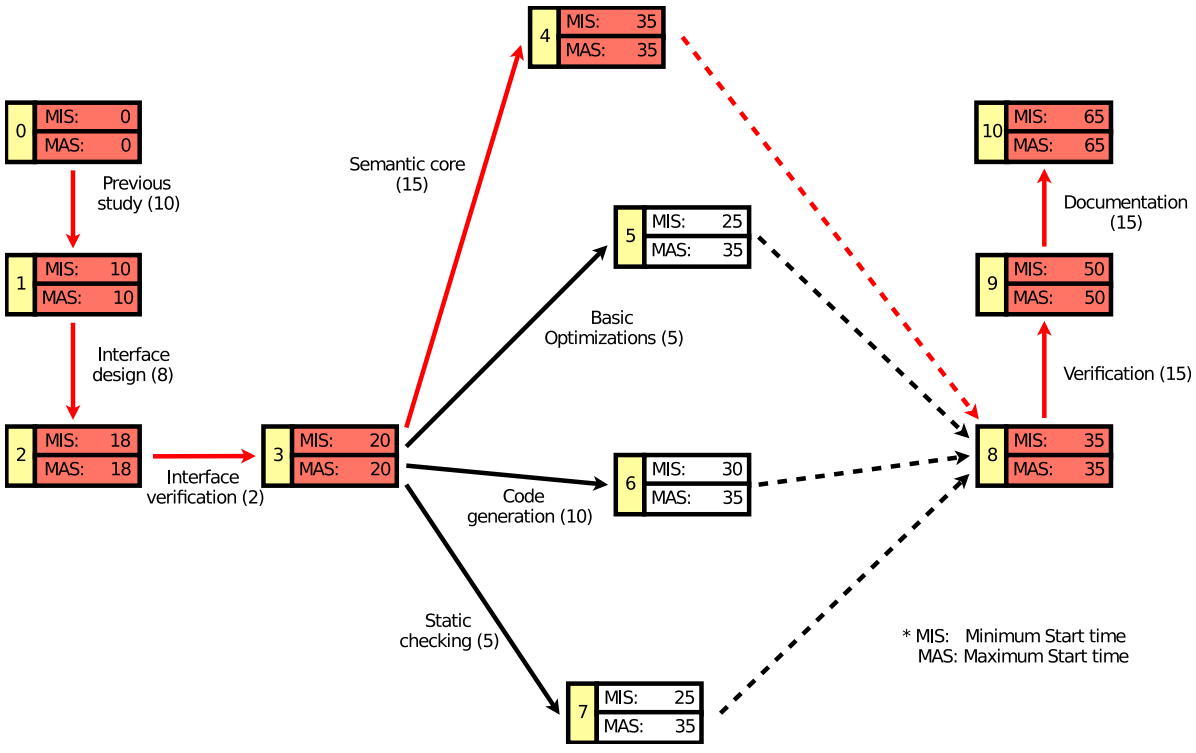

Fig. 7.1: Critical path analysis

As we can see, the only non-critical tasks of the project were:

- Basic optimizations. Gap $= 35 - 20 - 5 = 10$ days

- Code generation. Gap $= 35 - 20 - 10 = 5$ days
- Static checking. Gap $= 35 - 20 - 5 = 10$ days

7.3.2 Time schedule

The parallelization chances analyzed on the previous section allow the project to be finished in 65 business days. However, there was only one student in charge of the development, so all tasks were done sequentially, one after the other and only one at a time. Given this scenario, the resulting time schedule is shown in Figure 7.2 as a Gantt chart, where we can see the project finished on the last Friday of May. This allowed us to have the whole month of June to make the final preparations for the delivery.

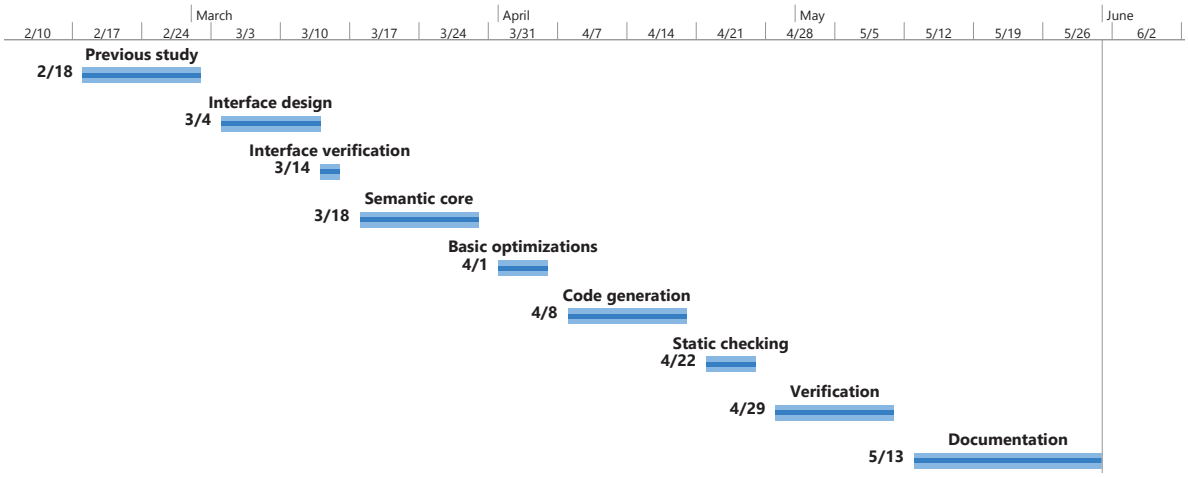

Fig. 7.2: Gantt diagram

7.4 Alternative plans in case of possible deviations

Many of the tasks of this project allow for an alternative action plan in case of possible schedule deviations. These alternative plans simplify some of the non-critical subtasks to ensure the project finishes on time. Given that there is only one student in charge of the project development, any delay on any task would suppose a delay on the total project duration. However, this section will show alternative plans for the most error-prone tasks. Those tasks not having an alternative plan in this section are sufficiently studied and they should not present any unexpected problems.

7.4.1 Interface design

In case we had problems when determining the exact interface in terms of the level of detail it exposes, it was preferable to choose the more detailed options. This will avoid possible expressivity lacks when in comes to use DFSSL. In case we later realized we had excessive and unnecessary details, we could remove them.

7.4.2 Basic optimizations

Even though basic optimizations add value to DFSSL, they are not an absolute priority for a stable version release. Therefore, in case of deviations or problems on other more important tasks like the semantic core or the code generation, this task can be postponed or even omitted until a sufficient time window is available.

7.4.3 Static checking

This task was not as dispensable as the optimizations one. However, in case of deviations we could also postpone or omit the static checking implementation to ensure the project keeps on schedule.

7.4.4 Code generation

In the ideal case, we wanted to experiment with different code generators (see Section 7.2.6). However, our priority was to have at least a correct code generator that fits out purpose in order to get a first stable version of DFSSL. In case we had problems with this task, it was preferable to finish as soon as possible the easiest alternative.

7.4.5 Verification

If any of the programs and tests we prepared for verifying the system presents important problems, it can be omitted or replaced by a similar example. Nevertheless, we had to take into account that all the parts of the system needed testing and verification. Therefore, we had to be sure that whenever we replaced or omitted any example, the functionalities tested by it were tested somewhere else.

7.5 Methodology

This final section shows the details on our methodology and work flows to coordinate management and development.

7.5.1 Development

All the software this project is composed by was developed by Alejandro, the student presenting this project. Scala IDE was used for editing Scala code and Simple Build Tool (SBT) [12] for building the DFSSL code generator and manage its dependencies.

7.5.2 Directive feedback

The student met the directors of the project weekly to ensure the project is progressing adequately and following the schedule. Since the code, calendar and planning was stored in our shared Redmine server, the directors had access to the GIT source code repository, so they could check out all the changes/additions the student made in the software at any time.

Chapter 8

Conclusions and Future Work

The solution presented in this project is an effort towards adding a layer of abstraction upon the complexity and heterogeneity underlying today's modern supercomputers. Even if the language we implemented, DFSSL, resembles very much to OmpSs specially in host-side task syntax, we have provided a compilation layer that is able to reason and optimize at NDRange level. Therefore, with the DFSSL compilation layer, we have set up the base for optimizing the host/accelerator interaction in the future. However, we could not provide stable versions of kernel fusion algorithms or GPU model code specialization because of the limited time we had to present this project. Nevertheless, the version of DFSSL we present here provides basic optimizations by just the fact of relying on LMS to schedule the code generation, as well as security mechanisms to ensure the host/accelerator interaction is as safe as possible. The fact that DFSSL is embedded in Scala gives it yet another good trait by introducing type inference and a quite expressive/clean syntax compared to other languages serving similar purposes like raw OpenCL or CUDA.

The main goal of DFSSL is to become the foundation to build high-level DSLs that can run on heterogeneous architectures. Following this idea, we are currently working on domain specific languages for solving convection-diffusion-reaction (CDR) equations. This work is being carried out in cooperation with experts from fields of computational fluid dynamics and computational biomechanics. These DSLs will model mathematical vocabulary commonly used in the area of partial differential equations, whereas the language backend will emit DFSSL code. We also plan to extend DFSSL to support the concept of message passing to better support the development of distributed applications. It is worth noting that this work is done for the HPC field, so we are also planning an exhaustive performance evaluation of our approach to validate its feasibility.

Appendix A

Installation

As part of the DFSSL compiler package, we include the corresponding instructions to install it, test it, and run examples. This appendix will explain how we designed the build system and a step-by-step guide for those who wish to install and test the language in a Linux-based OS. The project is not yet published anywhere, but we will assume it is already published and stored on a folder we will call `/home/you/dfssl`.

A.1 Get the Simple Build Tool

SBT is a powerful build system easy configurable by just writing Scala scripts. The LMS library uses this tool to build and install itself on any system and make it available to any other project being built by SBT. For instructions on how to get and install SBT, go to [12].

A.2 Getting and installing LMS

The following steps will download and install LMS assuming you have SBT already installed on your system and available in your path by directly referring to it with the `sbt` command.

1. Clone the LMS GIT repository: `https://github.com/TiarkRompf/virtualization-lms-core.git`.
2. Get inside the LMS folder.
3. Switch to a branch called `develop`.
4. Compile and install LMS to the SBT library folder by running `sbt publish-local`.

Now you should have LMS ready to be used by any project compiled with SBT.

A.3 Installing LLVM and the DFSSL analysis tool

DFSSL uses a LLVM-based tool to implement the safety mechanisms related to the usage of kernel functions. The specific version of LLVM required by DFSSL is 3.1 and newer versions will not work due to a structural redesign of the LLVM framework from version 3.2 onwards, so please be sure of getting this exact version of LLVM. To get LLVM ready, we recommend following these steps:

1. Get the LLVM source code:
   http://llvm.org/releases/3.1/llvm-3.1.src.tar.gz.
2. Get the CLANG source code and place it under `llvm/tools`:
   http://llvm.org/releases/3.1/clang-3.1.src.tar.gz.
3. Get the Compiler-RT source code and place it under `llvm/projects`:
   http://llvm.org/releases/3.1/compiler-rt-3.1.src.tar.gz.
4. Copy the folder `stagingtime/ompss-pragma-parser` from DFSSL to the folder called `llvm/tools/clang/examples` and edit the `CMakeLists.txt` there so it just contains this line: `add_subdirectory(ompss-pragma-parser)`. This will add the DFSSL analysis tool to the build tree of your LLVM project.
5. Create a folder named `build` inside the LLVM directory.
6. Inside the `build` directory you just created, run `cmake ..` to generate the make-files and then run `make` to compile LLVM and the DFSSL analysis tool.
7. Once the compilation finishes, you will have the binary `ompss-pragma-parser` under `llvm/build/bin`. Be sure to put it on your path.

Upon completing these steps, you will have the LLVM tool required by DFSSL. We could package the binary directly inside our distribution, but it is larger than 700 MB, so we decided it was better to give the source and let users compile it.

A.4 Define the DFSSL path and run examples

The last thing remaining to install DFSSL is to define an environment variable called `DFSSL_DIR` to your `/home/you/dfssl`. This environment variable is used by the DFSSL compiler to access the files it requires to run.

Now, to test your installation, generate code for the *K-Means* algorithm by running `make kmeans` under your DFSSL directory. Alternatively, you can just run in `sbt run` and SBT will prompt you to select the application you want to run among all the available.

Appendix B

OmpSs code generated by DFSSL

This appendix contains the exact code obtained from running the DFSSL compiler on each of the applications shown in Chapter 6.

B.1 N-Bodies

In this application, the input is given from a file, so the NDRange of the kernel used cannot be known at compile time. Therefore, as shown in Code B.1, the pragmas are specified each time the kernel is called.

```
1  /*****************************************
2    Emitting C++ OmpSS BSC Generated Code
3
4    Compile command for a file foo.cpp:
5    mcxx foo.cpp -o foo --ompss
6  *****************************************/
7  #include <iostream>
8  #include <cstring>
9  #include <cstdlib>
10 #include <cmath>
11 #include <fstream>
12 #include <boost/scoped_array.hpp>
13
14 int main(int argc, char** argv) {
15   using namespace std;
16   cout.setf(ios::fixed);
17   cout.precision(3);
18   const int x0 = argc;
19   const bool x1 = x0 < 2;
20   const std::string x2 = argv[0];
21   if (x1) {
22     cout << "Usage: ";
23     cout << x2;
```

```
    cout << " <input> [output]\n";
    exit(1);
  } else {
  }
  const std::string x10 = argv[1];
  ifstream x11(x10);
  int x12;
  x11 >> x12;
  boost::scoped_array<float> x89(new float[x12]);
  float* x13 = x89.get();
  const int x14 = x12 * 4;
  boost::scoped_array<float> x90(new float[x14]);
  float* x15 = x90.get();
  boost::scoped_array<float> x91(new float[x14]);
  float* x16 = x91.get();
  for(int x18=0; x18 < x12; x18++) {
    const float x19 = x13[x18];
    x11 >> x13[x18];
    const int x21 = 4 * x18;
    const float x22 = x15[x21];
    x11 >> x15[x21];
    const int x24 = x21 + 1;
    const float x25 = x15[x24];
    x11 >> x15[x24];
    const int x27 = x21 + 2;
    const float x28 = x15[x27];
    x11 >> x15[x27];
    const float x30 = x16[x21];
    x11 >> x16[x21];
    const float x32 = x16[x24];
    x11 >> x16[x24];
    const float x34 = x16[x27];
    x11 >> x16[x27];
    const int x36 = rand() % (48-5) + 5;
    float x37 = (float)x36;
    x13[x18] = x37;
  }
  boost::scoped_array<float> x92(new float[x14]);
  float* x41 = x92.get();
  boost::scoped_array<float> x93(new float[x14]);
  float* x42 = x93.get();
  const int x44 = 4 * x12;
  const int x45 = x12 / 4;
```

```
67    const int x46 = x45 + 1;
68    for(int x49=0; x49 < 86400; x49++) {
69      const int x50 = x49 % 2;
70      const bool x51 = x50 == 0;
71      if (x51) {
72        #pragma omp task in(x12, [x12] x13, [x44] x15, [x44] x16)
73                         out([x44] x41, [x44] x42)
74        #pragma omp device target(opencl) ndrange(1, x46*4, 4)
75        nbodyStep(1.0000000000f, x12, x13, x15, x41, x16, x42);
76
77      } else {
78        #pragma omp task in(x12, [x12] x13, [x44] x41, [x44] x42)
79                         out([x44] x15, [x44] x16)
80        #pragma omp device target(opencl) ndrange(1, x46*4, 4)
81        nbodyStep(1.0000000000f, x12, x13, x41, x15, x42, x16);
82      }
83    }
84    #pragma omp task in(x12, [x12] x13, [x44] x15, [x44] x16)
85    {
86      for(int x59=0; x59 < x12; x59++) {
87        const float x60 = x13[x59];
88        cout << x60;
89        cout << " ";
90        const int x63 = 4 * x59;
91        const float x64 = x15[x63];
92        cout << x64;
93        cout << " ";
94        const int x67 = x63 + 1;
95        const float x68 = x15[x67];
96        cout << x68;
97        cout << " ";
98        const int x71 = x63 + 2;
99        const float x72 = x15[x71];
100       cout << x72;
101       cout << " ";
102       const float x75 = x16[x63];
103       cout << x75;
104       cout << " ";
105       const float x78 = x16[x67];
106       cout << x78;
107       cout << " ";
108       const float x81 = x16[x71];
109       cout << x81;
```

```
110       cout << "\n";
111     }
112   }
113   return 0;
114 }
115 /*****************************************
116   End of C++ OmpSS BSC Generated Code
117 *****************************************/
```

Code B.1: Generated code for the N-Bodies application

B.2 K-Means

In contrast with the N-Bodies application, we implemented K-Means so the size of the problem is known at compile time. Hence, as we can see in Code B.2, the DFSSL compiler can create a headerfile (`kmeans.h`) with the pragma annotations, then include it and call the kernel directly. The content of the generated headerfile was shown previously in Code 5.4.

```
1  /*****************************************
2    Emitting C++ OmpSS BSC Generated Code
3
4    Compile command for a file foo.cpp:
5    mcxx foo.cpp -o foo --ompss
6  *****************************************/
7  #include <iostream>
8  #include <cstring>
9  #include <cstdlib>
10 #include <cmath>
11 #include <fstream>
12 #include <boost/scoped_array.hpp>
13 #include "kmeans.h"
14
15 int main(int argc, char** argv) {
16   using namespace std;
17   cout.setf(ios::fixed);
18   cout.precision(3);
19   srand(1);
20   boost::scoped_array<float> x98(new float[4096]);
21   float* x1 = x98.get();
22   boost::scoped_array<float> x99(new float[12]);
23   float* x2 = x99.get();
```

```
  #pragma omp parallel for
  for (size_t i = 0; i < 4096; ++i) {
    const float x3 = x1[i];
    const int x4 = rand();
    x1[i] = x4;
  }
  #pragma omp parallel for
  for (size_t i = 0; i < 12; ++i) {
    const float x7 = x2[i];
    const int x8 = rand();
    x2[i] = x8;
  }
  boost::scoped_array<int> x100(new int[1024]);
  int* x13 = x100.get();
  for(int x15=0; x15 < 5000; x15++) {
    assign_closest(1024, 3, x1, x2, x13);
    boost::scoped_array<float> x101(new float[12]);
    float* x17 = x101.get();
    boost::scoped_array<int> x102(new int[3]);
    int* x18 = x102.get();
    #pragma omp parallel for
    for (size_t i = 0; i < 12; ++i) {
      const float x19 = x17[i];
      x17[i] = 0;
    }
    #pragma omp parallel for
    for (size_t i = 0; i < 3; ++i) {
      const int x21 = x18[i];
      x18[i] = 0;
    }
    for(int x24=0; x24 < 1024; x24++) {
      const int x25 = x13[x24];
      const int x26 = x25 * 4;
      const float x28 = x17[x26];
      const int x29 = x26 + 1;
      const float x30 = x17[x29];
      const int x31 = x26 + 2;
      const float x32 = x17[x31];
      const int x33 = x18[x25];
      const int x27 = x24 * 4;
      const float x34 = x1[x27];
      const int x35 = x27 + 1;
      const float x36 = x1[x35];
```

```
67        const int x37 = x27 + 2;
68        const float x38 = x1[x37];
69        #pragma omp task in(x1[x27], x1[x35], x1[x37]) inout(x17[x26],
            x17[x29], x17[x31], x18[x25])
70        {
71        const float x39 = x17[x26];
72        const float x40 = x1[x27];
73        const float x41 = x39 + x40;
74        x17[x26] = x41;
75        const float x43 = x17[x29];
76        const float x44 = x1[x35];
77        const float x45 = x43 + x44;
78        x17[x29] = x45;
79        const float x47 = x17[x31];
80        const float x48 = x1[x37];
81        const float x49 = x47 + x48;
82        x17[x31] = x49;
83        const int x51 = x18[x25];
84        const int x52 = x51 + 1;
85        x18[x25] = x52;
86        }
87      }
88      #pragma omp task in([12] x17, [1024] x18) out([12] x2)
89      {
90        for(int x59=0; x59 < 3; x59++) {
91        const int x61 = 4 * x59;
92        for(int x60=0; x60 < 3; x60++) {
93          const int x62 = x61 + x60;
94          const float x63 = x17[x62];
95          const int x64 = x18[x59];
96          float x65 = (float)x64;
97          const float x66 = x63 / x65;
98          x2[x62] = x66;
99        }
100       }
101     }
102   }
103   #pragma omp task in([12] x2)
104   {
105     for(int x76=0; x76 < 3; x76++) {
106       cout << "Centroid ";
107       cout << x76;
108       cout << ": ";
```

```
109         const int x77 = 4 * x76;
110         const float x82 = x2[x77];
111         cout << x82;
112         cout << " ";
113         const int x85 = x77 + 1;
114         const float x86 = x2[x85];
115         cout << x86;
116         cout << " ";
117         const int x89 = x77 + 2;
118         const float x90 = x2[x89];
119         cout << x90;
120         cout << "\n";
121       }
122     }
123     return 0;
124 }
125 /*****************************************
126   End of C++ OmpSS BSC Generated Code
127 *****************************************/
```

Code B.2: Generated code for the K-Means application

B.3 Cholesky decomposition

Our implementation of the Cholesky decomposition runs completely on the host side and it was created to show how to use external target code libraries with DFSSL. Please note how the first comment shown in Code B.3, indicated which additional libraries should be included when we compile the generated application.

```
1  /*****************************************
2    Emitting C++ OmpSS BSC Generated Code
3
4    Compile command for a file foo.cpp:
5    mcxx foo.cpp -o foo --ompss -llapack_atlas -lblas
6  *****************************************/
7  #include <iostream>
8  #include <cstring>
9  #include <cstdlib>
10 #include <cmath>
11 #include <fstream>
12 #include <boost/scoped_array.hpp>
13
```

```
14 extern "C" {
15   #include <atlas/cblas.h>
16 }
17 extern "C" {
18   #include <atlas/clapack.h>
19 }
20
21 int main(int argc, char** argv) {
22   using namespace std;
23   cout.setf(ios::fixed);
24   cout.precision(3);
25   boost::scoped_array<float> x104(new float[36]);
26   float* x4 = x104.get();
27   float x8 = (float)RAND_MAX;
28   #pragma omp parallel for
29   for (size_t i = 0; i < 36; ++i) {
30     const float x5 = x4[i];
31     const int x6 = rand();
32     float x7 = (float)x6;
33     const float x9 = x7 / x8;
34     x4[i] = x9;
35   }
36   for(int x13=0; x13 < 6; x13++) {
37     const int x16 = x13 * 6;
38     for(int x15=0; x15 < x13; x15++) {
39       const int x17 = x16 + x15;
40       x4[x17] = 0.0000000000f;
41     }
42   }
43   for(int x23=0; x23 < 6; x23++) {
44     const int x24 = x23 * 6;
45     const int x25 = x24 + x23;
46     const float x26 = x4[x25];
47     const float x27 = x26 + 6.0000000000f;
48     x4[x25] = x27;
49   }
50   cout << "Matrix A:\n";
51   for(int x33=0; x33 < 6; x33++) {
52     const int x35 = x33 * 6;
53     for(int x34=0; x34 < 6; x34++) {
54       const int x36 = x35 + x34;
55       const float x37 = x4[x36];
56       cout << x37;
```

```
57        cout << " ";
58      }
59      cout << "\n";
60    }
61    cout << "\n";
62    for(int x47=0; x47 < 2; x47++) {
63      const int x48 = x47 * 6;
64      const int x49 = x48 * 3;
65      float* x50 = x4 + x49;
66      const int x51 = x47 * 3;
67      float* x52 = x50 + x51;
68      clapack_spotrf(CblasColMajor, CblasLower, 3, x52, 6);
69      const int x54 = x47 + 1;
70      for(int x56=x54; x56 < 2; x56++) {
71        float* x57 = x4 + x49;
72        float* x58 = x57 + x51;
73        const int x59 = x56 * 3;
74        float* x60 = x57 + x59;
75        cblas_strsm(CblasColMajor, CblasRight, CblasLower, CblasTrans,
76        CblasNonUnit, 3, 3, 1.0, x58, 6, x60, 6);
77      }
78      for(int x64=x54; x64 < 2; x64++) {
79        const int x68 = x64 * 3;
80        for(int x66=x54; x66 < x64; x66++) {
81          float* x67 = x4 + x49;
82          float* x69 = x67 + x68;
83          const int x70 = x66 * 3;
84          float* x71 = x67 + x70;
85          const int x72 = x66 * 6;
86          const int x73 = x72 * 3;
87          float* x74 = x4 + x73;
88          float* x75 = x74 + x68;
89          cblas_sgemm(CblasColMajor, CblasTrans, CblasNoTrans, 3, 3,
90            3, -1.0000000000f, x69, 6, x71, 6, 1.0000000000f, x75, 6);
91        }
92        float* x79 = x4 + x49;
93        float* x80 = x79 + x68;
94        const int x81 = x64 * 6;
95        const int x82 = x81 * 3;
96        float* x83 = x4 + x82;
97        float* x84 = x83 + x68;
98        cblas_ssyrk(CblasColMajor, CblasLower, CblasNoTrans, 3, 3,
99        -1.0000000000f, x80, 6, 1.0000000000f, x84, 6);
```

```
100     }
101   }
102   cout << "Cholesky Factorization of A:\n";
103   for(int x91=0; x91 < 6; x91++) {
104     const int x93 = x91 * 6;
105     for(int x92=0; x92 < 6; x92++) {
106       const int x94 = x93 + x92;
107       const float x95 = x4[x94];
108       cout << x95;
109       cout << " ";
110     }
111     cout << "\n";
112   }
113   return 0;
114 }
115 /*****************************************
116   End of C++ OmpSS BSC Generated Code
117 *****************************************/
```

Code B.3: Generated code for the Cholesky decomposition application

Bibliography

[1] The CLPP Project Website. URL http://code.google.com/p/cudpp/. 8

[2] The C++ Programming Language Website. URL http://www.cplusplus.com/. 8

[3] CUDA Framework for GPGPU programming of NVIDIA Graphics cards. . URL http://www.nvidia.com/object/cuda_home_new.html. 8, 12

[4] The CUDPP Project Website. . URL http://code.google.com/p/cudpp/. 8

[5] Delite Project. URL http://stanford-ppl.github.com/Delite/index.html. 11

[6] Liszt: A DSL for mesh solvers. URL http://liszt.stanford.edu/. 9

[7] LMS Official website. URL http://scala-lms.github.io/. 10, 24, 27

[8] The mercurium compiler website. URL http://pm.bsc.es/mcxx. 17

[9] NANOX++ Project website. URL https://pm.bsc.es/projects/nanox. 3, 11, 17, 54

[10] The OmpSS Programming Model. URL http://pm.bsc.es/ompss. 11, 17

[11] OpenCL, the standard programming language for heterogeneus parallel systems. URL http://www.khronos.org/opencl/. 8, 12

[12] Simple build tool for scala. URL http://typesafe.com/technology/sbt. 58, 60

[13] Scala Virtualized. URL https://github.com/TiarkRompf/scala-virtualized. 6, 10

[14] A. V. Aho, B. W. Kernighan, and P. J. Weinberger. *The AWK programming language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987. ISBN 0-201-07981-X. 6

[15] J. Bentley. Programming pearls: little languages. *Commun. ACM*, 29(8):711–721, Aug. 1986. ISSN 0001-0782. doi: 10.1145/6424.315691. URL http://doi.acm.org/10.1145/6424.315691. 6

[16] Y. Bertot and P. Castéran. Interactive theorem proving and program development. Technical report, 1997. 6

[17] K. J. Brown, A. K. Sujeeth, H. J. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. A heterogeneous parallel framework for domain-specific languages. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 89–100, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4566-0. doi: 10.1109/PACT.2011.15. URL http://dx.doi.org/10.1109/PACT.2011.15. 11

[18] J. Carette, O. Kiselyov, and C.-c. Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.*, 19(5):509–543, Sept. 2009. ISSN 0956-7968. doi: 10.1017/S0956796809007205. URL http://dx.doi.org/10.1017/S0956796809007205. 24

[19] J. M. Curran. *Introduction to Data Analysis with R for Forensic Scientists*. CRC Press, Boca Raton, FL, 2011. ISBN 9781420088267. 6

[20] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan. Liszt: a domain specific language for building portable mesh-based pde solvers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 9:1–9:12, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0771-0. doi: 10.1145/2063384.2063396. URL http://doi.acm.org/10.1145/2063384.2063396. 9

[21] C. Donnelly and R. Stallman. Bison. the YACC-compatible parser generator, 2004. 6

[22] M. Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 1st edition, 2010. ISBN 0321712943, 9780321712943. 6

[23] D. Goodman, S. Khan, C. Seaton, Y. Guskov, B. Khan, M. Lujan, and I. Watson. DFScala: High level dataflow support for Scala. 2012. 8

[24] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. D. Reis, and A. Lumsdaine. Concepts: linguistic support for generic programming in C++. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 291–310, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-348-4. doi: http://doi.acm.org/10.1145/1167473.1167499. 21

[25] D. Groen, S. P. Zwart, T. Ishiyama, and J. Makino. High performance gravitational n-body simulations on a planet-wide distributed supercomputer. *CoRR*, abs/1101.0605, 2011. 4

[26] N. J. Higham, M. Eprint, and N. J. Higham. Analysis of the cholesky decomposition of a semi-definite matrix. In *in Reliable Numerical Computation*, pages 161–185. University Press, 1990. 4

[27] U. Jørring and W. L. Scherlis. Compilers and staging transformations. In *Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '86, pages 86–96, New York, NY, USA, 1986. ACM. doi: 10.1145/512644.512652. URL http://doi.acm.org/10.1145/512644.512652. 24

[28] C. Lattner. LLVMM and Clang: Next Generation Compiler Technology. *BSDCan 2008: The BSD Conference, Ottawa, Canada*, May 2008. 28

[29] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004. 28

[30] A. Mametjanov, D. Lowell, C.-C. Ma, and B. Norris. Autotuning stencil-based computations on gpus. In *Proceedings of the 2012 IEEE International Conference on Cluster Computing*, CLUSTER '12, pages 266–274, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-0-7695-4807-4. doi: 10.1109/CLUSTER.2012.46. URL http://dx.doi.org/10.1109/CLUSTER.2012.46. 10

[31] N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka. Physis: An Implicitly Parallel Programming Model for Stencil Computations on Large-Scale GPU-Accelerated Supercomputers. 10

[32] MATLAB. *version 7.10.0 (R2010a)*. The MathWorks Inc., Natick, Massachusetts, 2010. 6

[33] A. Moors, F. Piessens, and M. Odersky. Generics of a higher kind. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, OOPSLA '08, pages 423–438, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-215-3. doi: 10.1145/1449764.1449798. URL http://doi.acm.org/10.1145/1449764.1449798. 21

[34] A. Moors, T. Rompf, P. Haller, and M. Odersky. Scala-virtualized. In *Proceedings of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation*, PEPM '12, pages 117–120, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1118-2. doi: 10.1145/2103746.2103769. URL http://doi.acm.org/10.1145/2103746.2103769. 6, 10, 28

[35] A. Munshi, B. R. Gaster, T. G. Mattson, J. Fung, and D. Ginsbur. *OpenCL Programming Guide*. The Khronos Group, 2009. 8, 12

[36] U. Norell. Dependently typed programming in agda. In *In Lecture Notes from the Summer School in Advanced Functional Programming*, 2008. 6

[37] M. Odersky and al. An Overview of the Scala Programming Language. Technical Report IC/2004/64, EPFL, Lausanne, Switzerland, 2004. 19

[38] M. Odersky and M. Zenger. Scalable Component Abstractions. In *Proceedings of OOPSLA 2005*, 2005. 23

[39] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. Artima, 2008. ISBN 9780981531601. URL http://books.google.es/books?id=MFjNhTjeQKkC. 8

[40] B. C. d. S. Oliveira, A. Moors, and M. Odersky. Type Classes as Objects and Implicits. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pages 341–360. ACM, 2010. 21

[41] E. Pašalić, W. Taha, and T. Sheard. Tagless staged interpreters for typed languages. *SIGPLAN Not.*, 37(9):218–229, Sept. 2002. ISSN 0362-1340. doi: 10.1145/583852.581499. URL http://doi.acm.org/10.1145/583852.581499. 7

[42] A. Prokopec, T. Rompf, P. Bagwell, and M. Odersky. On A Generic Parallel Collection Framework. Technical report, 2011. 8

[43] T. Rompf and M. Odersky. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. In *Proceedings of the ninth international conference on Generative programming and component engineering*, GPCE '10, pages 127–136, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0154-1. doi: 10.1145/1868294.1868314. URL http://doi.acm.org/10.1145/1868294.1868314. 10, 24, 25, 28

[44] T. Sheard and S. P. Jones. Template meta-programming for haskell. *SIGPLAN Not.*, 37(12):60–75, Dec. 2002. ISSN 0362-1340. doi: 10.1145/636517.636528. URL http://doi.acm.org/10.1145/636517.636528. 7

[45] M. Shindler, A. Wong, and A. W. Meyerson. Fast and accurate k-means for large datasets. In J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, and K. Weinberger, editors, *Advances in Neural Information Processing Systems 24*, pages 2375–2383. 2011. 4

[46] G. L. Steele, Jr. and R. P. Gabriel. The evolution of lisp. In *The second ACM SIGPLAN conference on History of programming languages*, HOPL-II, pages 231–270, New York, NY, USA, 1993. ACM. ISBN 0-89791-570-4. doi: 10.1145/154766.155373. URL http://doi.acm.org/10.1145/154766.155373. 7

[47] A. K. Sujeeth, H. Lee, K. J. Brown, T. Rompf, H. Chafi, M. Wu, A. R. Atreya, M. Odersky, and K. Olukotun. Optiml: An implicitly parallel domain-specific language for machine learning. In *ICML*, pages 609–616, 2011. 6, 11

[48] W. Taha. A gentle introduction to multi-stage programming. In *Domain-specific Program Generation, LNCS*, pages 30–50. Springer-Verlag, 2004. 7, 10

[49] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, June 2000. ISSN 0362-1340. doi: 10.1145/352029.352035. URL http://doi.acm.org/10.1145/352029.352035. 6

[50] S. Wolfram. *The Mathematica Book*. Wolfram Research Inc., 2000. 6

Index

Glossary

Domain expert Expert in any particular field of science that requires HPC software for research and/or production. iv, 1, 2, 52, 53

DSL embedding reffers to a DSL implementation technique consisting of providing DSLs as libraries which exploit the syntax of their host general purpose language or a subset thereof, while adding domain-specific language elements (data types, routines, methods, macros etc.). 3

GPU (Graphics Processing Unit) A massively parallel processor most commonly used for computer graphics. 12, 13, 15, 78

Heterogeneus architecture An heterogeneus computer architecture is composed by different kinds of processors (usually CPUs and GPUs) that can run at the same time. iv, 2, 3, 12

Kernel A function that runs on an accelerator such as a GPU. 3, 13, 14, 16, 37, 54

Scalability is the ability of a system, network, or process to handle a growing amount of work in a capable way or its ability to be enlarged to accommodate that growth. 1

Source-to-source compiler is a type of compiler that takes the source code of a programming language as its input and outputs the source code into another programming language. 17

Stencil kernel is a class of iterative kernel which updates array elements according to some fixed pattern, called stencil. They are most commonly found in computer simulations, e.g. for computational fluid dynamics in engineering applications. 10

Acronyms

API Application Programming Interface. 17

BSC Barcelona Supercomputing Center. 3, 17

DFSSL Data Flow SuperScalar Language. 3–5, 12, 29–31, 34–38, 40–44, 46, 48, 49, 53, 54, 57–61

DSL Domain Specific Language. iv, 1–12, 16, 19, 24–28, 30, 36, 44, 52, 53

HPC High Performance Computing. iv, 1, 2, 4, 5, 8, 16, 36, 52, 78

IR Intermediate Representation. 25–29

JVM Java Virtual Machine. 8, 20

LLVM Low Level Virtual Machine. 12, 28, 29, 40

LMS Lightweight Modular Staging. 3, 6, 10–12, 23–28, 31

OmpSs OpenMP Superscalar. 3, 12, 17, 18, 35, 54

OpenCL Open Computing Language. 2, 3, 12, 13, 15–18, 29–31, 34, 35, 37, 48, 54

SBT Simple Build Tool. 58

SIMT Single Instruction Multiple Thread. 13

SQL Simple Query Language. 1