Curso 2012/13
CIENCIAS Y TECNOLOGÍAS/11
I.S.B.N.: 978-84-15910-69-5

RUYMÁN REYES CASTRO

Directive-based Approach to Heterogeneous Computing

Director
FRANCISCO DE SANDE GONZÁLEZ

SOPORTES AUDIOVISUALES E INFORMÁTICOS
Serie Tesis Doctorales

Write a paper promising salvation, make it a 'structured' something or a 'virtual' something, or 'abstract', 'distributed' or 'higher-order' or 'applicative' and you can almost be certain of having started a new cult.

Edsger W. Dijkstra

To all those who told me, "You won't be able to".
Including myself.

Acknowledgements

Several people deserve to be mentioned, however I believe it only correct to begin by thanking my family. My parents have always encouraged me to follow my heart, never allowing me to give up when the going got tough. My mother would always tell me, "Do not close the door without walking through it first."; and so here I find myself, metaphorically walking through another door once again. I would also like to thank Yurena. Her patience was stretched to its limits during those sunny weekends that had to be spent working from home. Thankfully, sunny weekends will be spent outside from now on.

Next, I would like to thank my advisor, F. de Sande. This work would not have been possible without his support. He is very encouraging and convinced me to do an end-of-course project in 2006, before convincing me to undertake this Ph.D. research in 2008. Fortunately, there are no more projects to undertake, or grades to achieve. I would also like to thank F. Almeida and V. Blanco for their support and encouragement over the years. Finally, I must mention E. Quintana who placed his trust in me and allowed me to participate on his projects, despite the fact that Linear Algebra is not one of my strong points.

I would now like to give thanks to several friends who have acoompanied me on this journey and throughout my life. These are people who have shared my pain and happiness, and who reminded me to relax and also take things easy when necessary. So thank you U. Oramas, A. Rubio, M. Rodriguez, M Magdalena, M. Santana and C. Gonzalez, amongst others. Your advice shall continue to be used over the months and years to come.

I wish to thanks to all the folks from SAII, both past and present, but in particular I. Lopez, J. Fumero, and J.L. Grillo, all of whom encouraged me to continue my work. Their motivation and hard work never failed to astonish me and I hope that one day I will be able to help them as much as they were able to help me.

In recent years many people have made my stay abroad a very enjoyable experience: A. M. Huertas from The Center for Numerical Methods in Engineering (CIMNE) - who taught me to work with facts; A. Remon, F. Igual and T. Peña from Universitat Jaume I of Castellon; Davor Davidovic from the Rudjer Boskovic Institute; Ovidiu Tomescu from the Polytechnique University of Bucarest; J. Rius from Universitat de Lleida; R. Filgueira from University of Edinburgh; and C. Inglish and M. Bull from Edinburgh Parallel Computing Centre (EPCC).

Finally, I would like to thank the musicians who livened up all the hours that were spent producing this thesis, specifically Joe Satriani, Daft Punk and P. Tchaikovsky.

Although this thesis has not been directly funded through any scholarship program, the research was made possible thanks to my participation in the following projects: Spanish MEC (Plan Nacional de I+D+i, contract TIN2008-06570-C04-03), Canary Islands Government (ACIISI, contract SolSubC200801000285), TEXT Project (FP7-261580), HPC-EUROPA2 (project number: 228398).

Contents

1 Introduction **1**
- 1.1 Main Contributions . . . 7
- 1.2 Thesis Organisation . . . 8

2 Background and Related Work **9**
- 2.1 Classification of Programming Models . . . 10
  - 2.1.1 Traditional Shared Memory Multicore . . . 11
  - 2.1.2 Distributed-memory Based Systems . . . 14
  - 2.1.3 GPU Devices . . . 20
  - 2.1.4 Directive-based Languages for Accelerators . . . 34
  - 2.1.5 Multi-target Programming Languages . . . 39
  - 2.1.6 Final Remarks . . . 41
- 2.2 Compiler Support for PM . . . 42
  - 2.2.1 GCC . . . 43
  - 2.2.2 Open64 . . . 44
  - 2.2.3 LLVM . . . 47
  - 2.2.4 ROSE . . . 50
  - 2.2.5 Cetus . . . 52
  - 2.2.6 Mercurium . . . 53
  - 2.2.7 Final Remarks . . . 56
- 2.3 Runtime Support . . . 57
  - 2.3.1 StarPU . . . 57
  - 2.3.2 GOMP . . . 58
  - 2.3.3 GMAC . . . 60
  - 2.3.4 Final Remarks . . . 61

3 Yet Another Compiler Framework **63**
- 3.1 Design Considerations . . . 64
  - 3.1.1 Filter . . . 64
  - 3.1.2 Mutator . . . 66
- 3.2 Internal Representation . . . 67
  - 3.2.1 Manipulating the IR . . . 70
- 3.3 Symbol Table . . . 70

3.3.1 Scope Information . . . 72
3.3.2 Computing the Size of Elements . . . 75
3.4 The FRONTEND . . . 76
3.4.1 Defining a New Language . . . 78
3.5 The MIDDLEEND . . . 79
3.5.1 Data Dependency Analysis . . . 79
3.5.2 Loop Analysis . . . 81
3.5.3 Loop Optimizations . . . 82
3.5.4 The *Outliner* . . . 87
3.6 The BACKEND . . . 88
3.6.1 The Template Subsystem . . . 89
3.6.2 The DOT Back end . . . 90
3.6.3 The Writer Classes . . . 90
3.6.4 The CUDA Back end . . . 92
3.6.5 The OPENCL Back end . . . 93
3.7 Final Remarks . . . 93

4 The `Frangollo` Runtime **95**
4.1 The `Frangollo` Platform Model . . . 95
4.1.1 Components . . . 96
4.1.2 Execution . . . 96
4.1.3 Parallelism . . . 96
4.1.4 Memory Model . . . 96
4.1.5 Memory coherence . . . 97
4.1.6 Composing operations . . . 99
4.1.7 Applying the `Frangollo` Platform Model . . . 102
4.2 Software Architecture . . . 102
4.2.1 Abstract Layer . . . 102
4.2.2 Device Layer . . . 107
4.2.3 Interface Layer . . . 109
4.2.4 Overall Usage Workflow . . . 112
4.3 Final Remarks . . . 114

5 Directive-based Code Generation **115**
5.1 Extending `La Laguna C (llc)` to Support Hybrid MPI+OpenMP Programming 115
5.2 Generating CUDA . . . 117
5.3 Intermezzo: `La Laguna Computing Language (llcl)` . . . 119
5.3.1 Implementing `llcl` . . . 121
5.4 `Accelerator ULL (accULL)` . . . 121
5.4.1 OpenACC Programming Interface . . . 123
5.4.2 `c2frangollo` Compiler Driver . . . 126
5.4.3 Interfacing OpenACC in `Frangollo` . . . 127
5.4.4 Putting It All Together . . . 127

6 Performance results **129**
6.1 Experimental Platforms . . . 130
6.2 Other Compilers . . . 131
6.3 OpenMP Source Code Repository . . . 131
6.3.1 Mandelbrot Set Computation . . . 132
6.3.2 Molecular Dynamics . . . 136
6.3.3 Jacobi . . . 144
6.3.4 LU Reduction . . . 147
6.3.5 Matrix Multiplication . . . 150
6.3.6 Development Effort Analysis (LLC-CUDA) . . . 155
6.4 Rodinia Benchmark Suite . . . 157
6.4.1 SRAD . . . 158
6.4.2 LU Decomposition . . . 158
6.4.3 HotSpot . . . 162
6.4.4 PathFinder . . . 163
6.4.5 Needleman-Wunsch (NW) . . . 164

7 Conclusions and Future Work **171**
7.1 Directive-based Programming . . . 171
7.2 Programming Tools . . . 172
7.3 Development Productivity . . . 172
7.4 Future Work . . . 173

Contributions **175**

Bibliography **177**

List of Figures **189**

List of Tables **192**

List of Listings **194**
*

CHAPTER 1

Introduction

Computational science has become the third pillar of the scientific enterprise, a peer alongside theory and physical experiment.

U.S.A President Information Technology Committee report, June 2005 [140]

Computational Science, and particularly High Performance Computing (HPC), are the pillars on which recent advances in the fields of science and engineering can, and in fact, have been built. A wide range of fields have made use of these technologies to model and investigate natural processes that range from the movement of large astral bodies right down to the behaviour of atoms and molecules. Weather forecasting provides a clear example of their application: it uses the output from HPC models and broadcasts it daily to millions of viewers as part of the daily weather report. This modelling and investigation process has been made possible thanks to improved performance in scientific applications. One of the most significant benefits of HPC simulation is that it greatly reduces engineering costs: designers and manufacturers no longer need to build a 3-dimensional models of the devices, they can now use HPC to create models that simulate motor performance, aerodynamics, or even the behaviour of cars involved in a crash. The overall result is that the time-to-market ratio for products is greatly reduced and productivity is increased, both of which directly affect profits.

Even the social sciences have found a uses for HPC, despite it being a field which has rarely been associated with these technologies. Social scientists have used HPC to produce simulations of population growth and emergency evacuation procedures and look set to continue finding more uses for it in the future.

Both governments and their industrial partners have become aware of the opportunities presented by Computational Science and HPC [41], and have funded several research programs over the years. Major industrial partners (Boeing, Airbus, Rolls Royce, BMW, . . .) are long-term users of HPC services. Recently, new funding schemes have emerged for small and medium-sized enterprises so that they can also take advantage of the technological advances offered by HPC. Good examples include the NDEMC [99] in the U.S (proposed by the White House), or Supercomputing Scotland [58].

High Performance Computing centres, which were previously associated with universities, have paved their way to becoming semi-autonomous entities supporting the computing part of science and engineering applications. In Germany, for example, several centres such as HLRS [70] offer computing services to both industry and academia alike.

The performance of the systems in these centres has evolved over the years. Normally, performance evolution - measured using the LinPACK [46] benchmark - can be found by referring to the TOP500 list [47], as shown in Figure 1.1.

Figure 1.2 and 1.3 show the number of cores per socket and the number of accelerators per system. At the beginning of the XXI century most of the systems were only made built using (large) set of processors connected through high-speed networks, but this situation has been changing since 2007.

The long announced end of Moore's Law [97, 78], which prevented increases in the performance of CPUs, meant vendors were now forced to change the architectures of CPUs in order to continually increase their performance. Multi-core and Many-core processors have been broadly adopted in the systems appearing in the TOP500 list and several different forms of accelerators are starting to appear in the most advanced systems. To maintain this performance, future systems shall have to be built using many-core processors and accelerators [24, 15, 137].

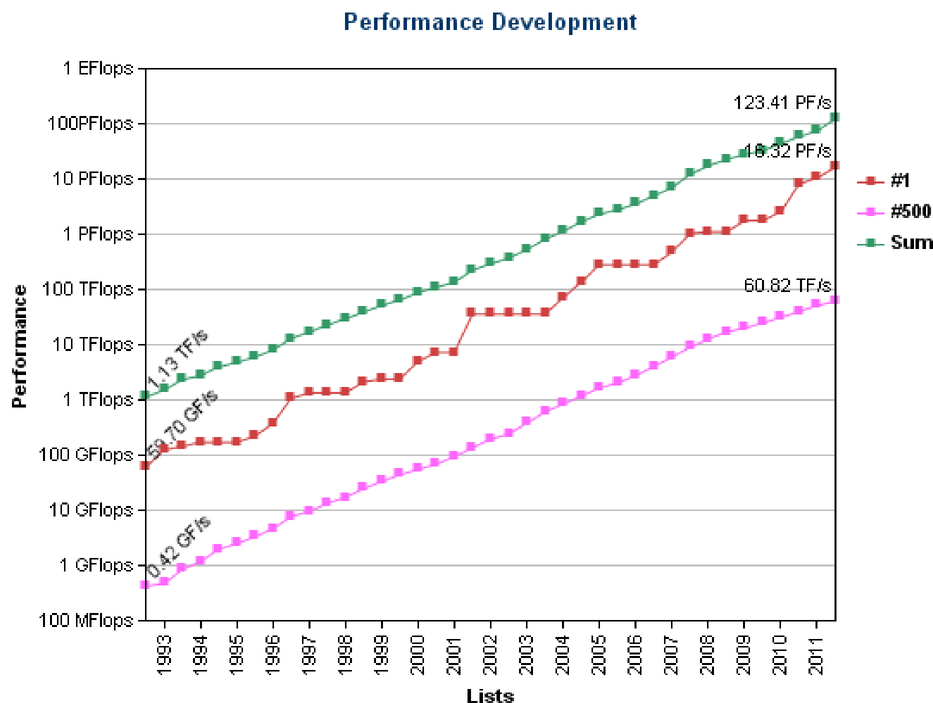


Figure 1.1: Evolution of the overall performance of HPC systems since 1993

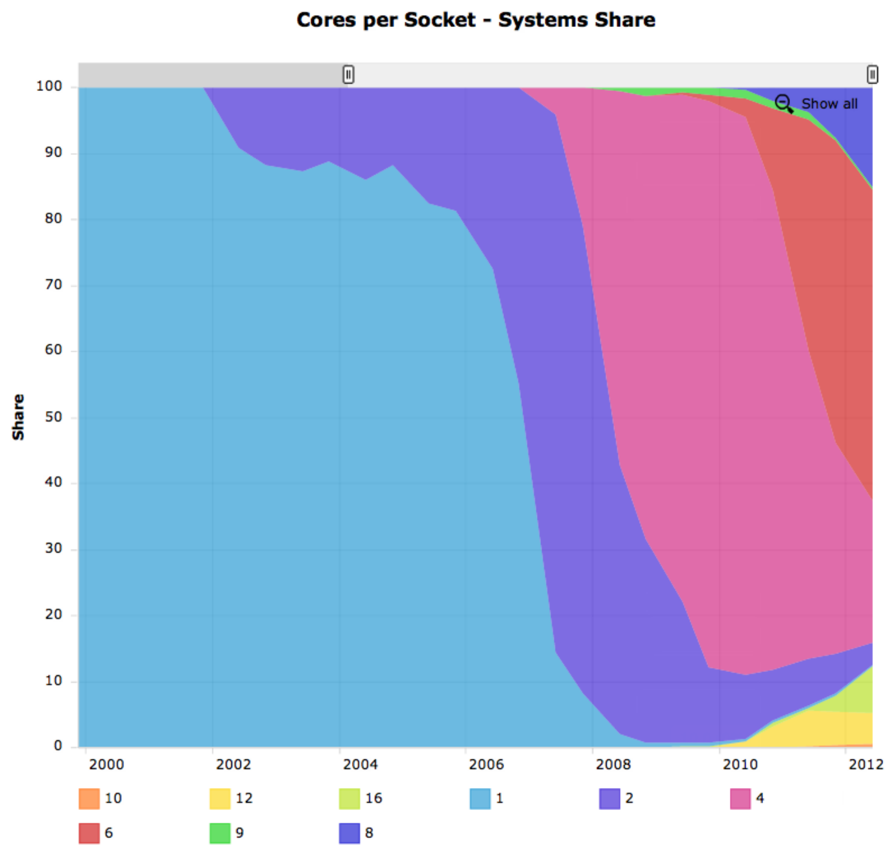


Figure 1.2: Evolution of the number of cores per socket per system since 2000

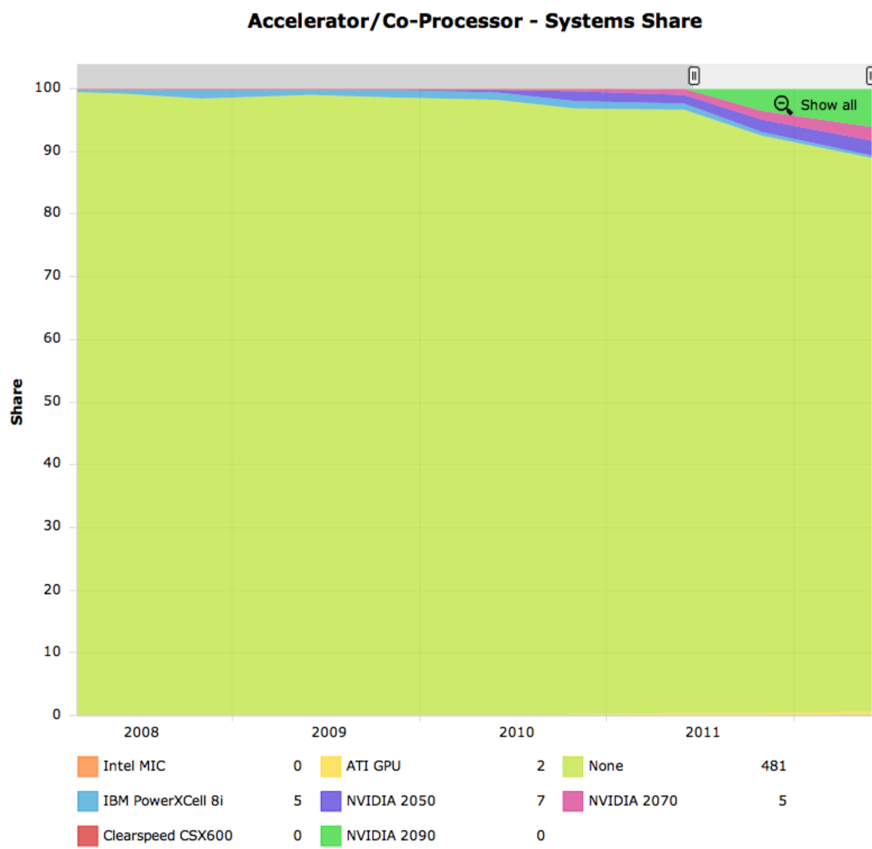


Figure 1.3: Evolution of the number of systems with accelerators since 2007

To get a clear idea of how scientists and engineers are solving current challenges in these systems, and to understand what the existing applications of this technology are, we should take a closer look at the reports being produced by the aforementioned HPC centres. The 2011 report by the UK National Supercomputing Service (HECToR) [139] highlights an important fact - more than 62% of their time was allocated to Environmental Science and Chemistry (see Figure 1.4). This figure reveals that the most common applications in this facility are as follows:

VASP [79], 17% of the total number of jobs; CP2K [25] - just under 7% of jobs; Unified Model (UM) [138] - just over 7%; and GROMACS [23] - just over 4.%. VASP, CP2K and UM have been written in Fortran and MPI, whereas GROMACS has been ported to C++ and MPI.

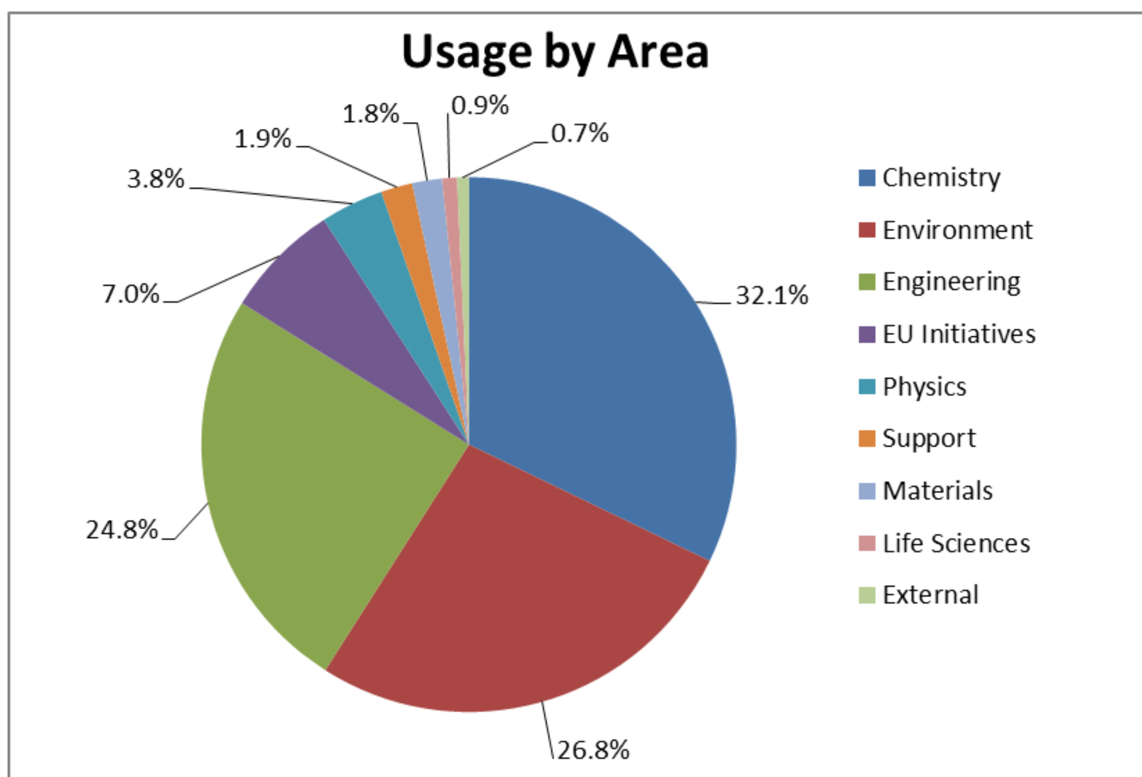


Figure 1.4: Usage of HECToR facilities by area of expertise

In the 2011, the Swiss National Computing Centre surveyed its users [134] and found that that 65% of them develop applications but nearly of 70% of these applications are being developed in Fortran, and more than 80% use MPI; only 40% of the codes used OpenMP.

This data reveals an important fact: the vast majority of the software running in HPC systems have been implemented using Fortran and MPI.

Fortran has existed since the dawn of computer science, and the MPI standard [95] has been available from the early 90s, far before multi- and many-core processors. Despite the appearance of new languages [89] and libraries (see Background and Related Work in Section 2), none of them are used on these top applications. Notably exceptions are low-level accelerator languages and libraries, such as CUDA, but mainly due to the fact that no Fortran alternative is available for writing the kernel code for accelerator devices (*yet*).

Traditionally, any performance improvements have been due to an increase in hardware performance. Nevertheless, increasingly complex hardware is forcing software engineering to participate in the effort of performance improvement. In order to achieve *performance portability*, the computer science community will have to provide additional assistance to both scientists and engineering experts who traditionally work with these applications.

Usually a scientist or engineer would only need to recompile their Fortran code with the compiler for his/her new architecture to take advantage of its performance. Compiler tools have become critical in this process [56, 94] as they enable the transparent handling of the increasingly complex processor instruction set. Auto-vectorization [101] provides a good example of how the effort in the compiler-side have enabled programmers to transparently take advantage of advanced units inside the processors [115]. Unfortunately compiler technology alone is not enough to take advantage of new architectures, especially following the irruption of multi-core and accelerators.

A scientist or engineer who is interested in taking advantage of the new multi-core architectures could try using OpenMP [43, 108] to parallelize the most demanding parts of the application. This API is widely available and there is a lot of supporting documentation available. However, due to the fact that most of the large shared memory machines have non-uniform access to memory banks, the process of scaling OpenMP applications to use a high number of threads is more than a matter of simply adding directives; it is important to also be familiar with the underlying SMP architecture.

In addition, scaling the hardware beyond dozens of SMP processors is still a challenge. Although we can foresee the imminent arrival of processors with eighty or one hundred cores inside, the programmer still needs to use different technologies to obtain the number of processors required to achieve Peta- or Exa-scale computing.

This is where MPI makes its appearance. With a notable amount of effort, this scientist or engineer will be able to write an MPI implementation of their code. Writing applications using an MPI API is not straightforward. Developers have to use low-level calls that hinder legibility [65].

One of the main drawbacks of MPI is that the development of parallel applications is highly time consuming as major code modifications are generally required. In other words, parallelizing a sequential application in MPI requires a considerable amount of effort and expertise. In a sense, we could say that MPI represents the assembler language of parallel computing: you can obtain the best performance but the cost is a significant amount of development.

Some (lucky) scientists or engineers may have access to the expertise of a HPC centre. For these scientists, this problem is easily overcome as the highly skilled developers working in the HPC centre can assist them by producing efficient implementations of their algorithms. However, the (silent) majority of scientists and engineers do not have access to such expertise.

In our opinion, the lack of general purpose high-level parallel languages presents a major drawback that limits the spread of High Performance and Parallel computing. There is a division between the users who have the needs of HPC techniques (scientist and engineers) and the experts who are capable of implementing efficient parallel codes for HPC systems (for example, those in HPC centres).

In general, most users do not have the skills to exploit the tools involved in the development of parallel applications. For this reason, any effort aimed at simplifying higher-level programming languages, and thus bridging the gap between the users and the tools, is welcomed.

A non-HPC expert developer may, given enough time, find that s/he is capable of producing an efficient implementation for their algorithm for a particular architecture. Nevertheless, their time would be better spent focusing on designing new techniques or algorithms.

Several studies have investigated whether there is any value in the time and effort spent writing low-level code [62, 14]. These studies have raised the concept of *programming productivity*, making this already familiar concept in the field of Software Engineering a focal point of HPC [60].

We claim that *using directive-based programming allows application developers to take advantage of heterogeneous architectures with a low-entry development effort and without increasing the maintenance cost of existing applications*. The performance of existing applications can be sustained by taking advantage of novel and emerging architectures, whilst new programming paradigms cope with the requirements in availability and stability required in these environments.

Our main objective is not to obtain maximum performance for a particular set of algorithms - that will be achieved eventually by a new generation of specific algorithms and/or libraries. Instead, our aim is to help users to benefit from new architectures using the existing codebase, while new libraries and tools are being developed.

To accomplish this task, compiler technologies must be taken to the next level; using a small amount of additional information in the form of directives, we have to extract the information that is required to transform the code into an efficient implementation for a heterogeneous sytem. However, most of the information is not available at compile time, therefore runtime support libraries are also needed.

1.1 Main Contributions

The main contributions of this thesis can be outlined as follows:

- An in-depth review of state-of-the-art compiler and runtime technologies.
- A compiler platform suitable for fast-prototyping, research and educational purposes.
- A runtime capable of running transparently on several execution platforms.
- Analysis and comparisons of different directive-based programming models for GPUs.

1.2 Thesis Organisation

In the HPC Group at La Laguna University [72] we have been working for more than ten years in the field of *Directive-based programming for High Performance Computing* [51, 52, 126, 49, 54, 50]. Since the very beginning, the goal of our research has always been to bridge the gap between HPC users and the tools they use; in other words, our aim is to provide the scientific community with the high-level programming tools they require to facilitate the coding effort [87].

In 2008, the Ph.D. dissertation of Mr. A.J. Dorta [48] marked a milestone in our research. As a result of his work, we were able to: define the La Laguna C language (`llc`); present its compiler (`llCoMP`); provide a representative collection of benchmarks that are suitable for implementing with our language; and demonstrate sufficient computational experience to support our conclusions.

To implement `llc` we designed and implemented a source-to-source compiler: `llCoMP`. The original `llCoMP` was implemented using traditional compiler creation toolkits, such as `bison` or `flex` [85]. The original `llCoMP` is still available on request but it is no longer being maintained.

Based on our experience with `llCoMP` we decided that if we were going to support new architectures, as we intended, then more flexible programming tools would be required. This dissertation opens with a comprehensive overview and bibliography of existing programming models and tools in Chapter 2. This is then followed by a detailed description of the specific tools that we have designed. In Chapter 3 we describe `YaCF`, a compiler framework aimed at facilitating the writing of source-to-source (StS) translations. In Chapter 4 we describe `Frangollo`, a runtime to leverage the complexity of heterogeneous platforms. Although their descriptions might be overwhelming at first glance, they provide detailed insights into the tools that allow a developer to implement transformations, or to extend the platform so it includes support for new architectures. Our description of the `Frangollo` Platform Model (Section 4.1), which should be of particular interest to many readers, attempts to formalize the underlying concepts on which our offloading paradigm is based. In Chapter 5 we describe the results of our research on *directive-based programming*. They are described in chronological order in an attempt to illustrate how we have arrived at our conclusions (Chapter 7). To support our conclusions, we have ported several codes using our directive approaches. We have provided performance results together with comments and consideration in Chapter 6. Several publications have been produced as a result of the work that has been carried out for this thesis; these publications are listed after Chapter 7.

CHAPTER 2

Background and Related Work

In this Chapter we present several different works that have both inspired and guided our work during the initial planning stages. These works have also made significant contributions to the field of HPC in general.

We start the Chapter with a general review of the programming models that have been used for reference purposes. In Section 2.1, we provide details of the classification system being used in our work, and in the corresponding paragraphs, we present a description of different programming models.

It is worth noting that several tools are required in order to implement a programming model. A software library may be required in order to implement low-level details of the programming model, such as communications, memory transfer, data distribution, etc. The library requires a well defined Application Programming Interface (API), capable of expressing all of the library features across a simple and orthogonal interface. Sometimes, even a well-defined API does not suffice and it fails to provide the end user with the simplicity that s/he requires.

In this case, a higher-level layer including compiler support is also required in order to leverage the coding effort. This is the case of OpenMP; although a low-level interface exists, it is not designed to the typical user, instead s/he is expected to use OpenMP through the high-level compiler directives. Sometimes, even compiler support is not enough, and other tools are required to write efficient code for a particular architecture. It is worth noting the importance of profiler, tracing and debugging tools that are available for some programming models. The maturity of a programming model can be measured according to the available tools. The more help the developer can access to implement their algorithm, the more mature a programming model is considered to be. For these reasons, we have decided to include in our bibliography review works that are related to programming tools.

In Section 2.2 our attention is focused on the complier tools that enable most of the programming models previously described. We examine their key characteristics before presenting an overview and final remarks. In Section 2.3 our focus is on exploring the different available runtimes for both multi-core and heterogeneous machines. We conclude the section with our remarks together with a brief discussion on the reasons for implementing our own runtime.

2.1 Classification of Programming Models

There are several different classifications for Programming Models (PM) in the bibliography. One example, that of *Skillicorn et al* [133], describes the characteristics of a good programming model: it has to be easy to program; it has to provide a software development methodology; it has to be architecture independent; it has to be easy to understand; and it has to provide some guarantees regarding performance. Based on these characteristics, programming models are classified into different categories depending on which part of the model is explicit or implicit. For example, some very high-level models are capable of completely abstracting the parallelism and the developer only has to specify what has to be done but not the way it has to be done. Other models may require the developer to indicate where the parallelism is, but the way it is decomposed is implicit. However, other models are completely explicit, and everything (parallelism, data mapping, communication and synchronization) has to be instructed by the user.

The classification described in [34] differentiates between *fragmented* and *non-fragmented* programming models.

In the *fragmented* programming models, algorithms are expressed task by task, decomposing data structures and control flow into several sets. The best candidate in this category is MPI. MPI requires the user to be aware of the number of processors that will be running the algorithm in order to properly distribute data across all of them and properly implement explicit communications.

On the other hand, on *non-fragmented* models, algorithms are implemented in a global scheme, assuming that a sole processor will execute all the code except for those sections of code where the parallel execution is explicitly indicated. OpenMP is the most frequently used language of this category. It introduces parallelism using explicit directives such as `for` or `sections`.

From our point of view these classifications do not suffice when we are describing languages for heterogeneous platforms. Whether a PM is *fragmented* or not is not sufficient enough to determine if it is suitable for a heterogeneous system. Although intuitively the *fragmented* models will be more suitable for architectures with separate memory spaces, some *non-fragmented* PM are implemented in these architectures as well.

The rest of this section is divided according to the `target architecture(s)` of the programming model. The `target architecture(s)` is defined as the main platform on which the programming model has been focused. Post-design transformations or implementations of the model are not taken into account. For example, OpenMP was initially designed to support shared-memory systems, but there are some OpenMP implementations that use the same approach for other platforms. Where applicable we provide the classification of the model in [34] form.

2.1.1 Traditional Shared Memory Multicore

Traditional processors have evolved into complex multi-core architectures, forcing programmers to consider multithreading and parallel programming in these environments to improve the usage of the memory hierarchy. Nowadays processors usually feature eight or sixteen cores, and might come in two-way socket boards, presenting a total of 32 cores to the defenceless programmers. In addition, traditional distributed-memory machines require hybrid paradigms to obtain scalable performance [91].

The most successful programming model for traditional multicore architectures is OpenMP. However, other alternatives also exists, such as low-level *pthread* [32] or programming libraries such as Thread Building Blocks (TBB) [143].

2.1.1.1 OpenMP

OpenMP [108] is a programming language based on compiler directives, library functions and environment variables used to declare parallelism in `C`, `C++` and `Fortran`. It is the result of the joint effort of several vendors, such as Intel, HP, IBM, Cray or Sun Microsystems, alongside important feedback from academia. OpenMP has been widely adopted in several projects and production codes [37], and plenty of work has been devoted to port this PM into different architectures, from the SMP NUMA system [105, 27] to distributed memory clusters [49], and even accelerators [26, 33].

OpenMP is a `non-fragmented` PM. The common usage pattern of OpenMP consist off annotating parallel regions inside a sequential code using directives representing common parallel patterns, such as loop, sections or tasks. This enables users to maintain only one sequential code whilst still being able to run it in parallel if appropriate compiler support is enabled. This is one of the key factors of OpenMP as it greatly increases programmer productivity.

OpenMP compilers and runtimes implement parallelism using sets of running threads. Programs developed under this model (called `fork-join`) start with a sole execution thread (`master thread`) and when a `parallel` directive is encountered, execution is split so the parallel region will be run by a team of threads. As long as the semantical end of the region is reached, threads join the master thread again, and this continues the execution sequentially (see Figure 2.1).

OpenMP can also be used in a `fragmented` way as it features a high level API with functions devoted to thread management (thread identification, current number of threads . . .). This enables advanced developers to manually distribute data across threads. Although this method of programming is not common, it provides programmers with the additional flexibility required to enable them to implement more complex patterns than that supported by the worksharing constructs.

The major drawback of OpenMP is its dependence on shared-memory machines as this severely limits its scalability. A way to deal with this limitation is to use OpenMP in combination with MPI. This enables programmers to take advantage of shared-memory parallelism for intra-node computations as MPI can be used to distribute data across computation nodes.

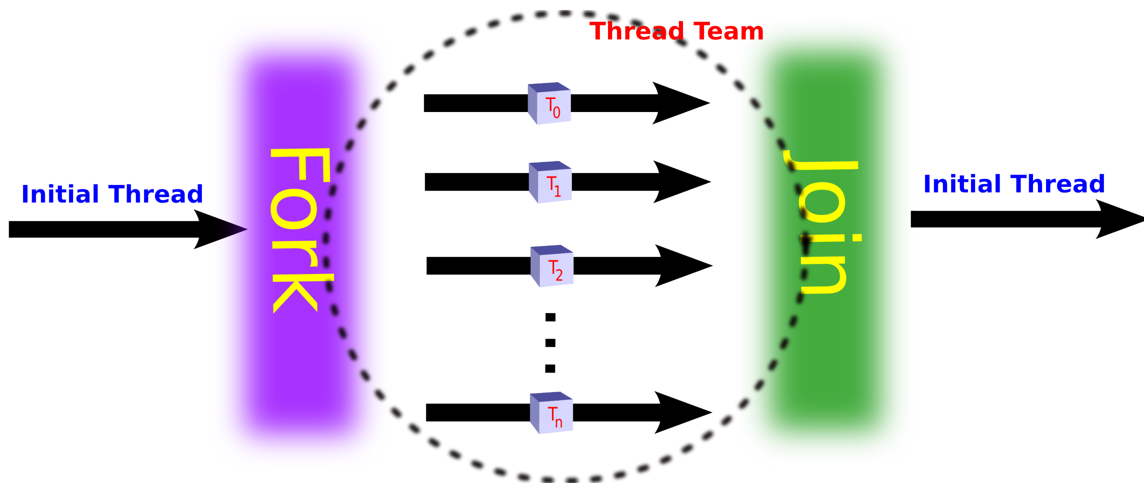


Figure 2.1: Fork join model

```
pi_omp = 0.0f;
#pragma omp parallel for private(i,local) reduction (+: pi_omp)
    for (i = 0; i < N; i++) {
        local = (i + 0.5)*w;
        pi_omp = pi_omp + 4.0/(1.0 + local*local);
    }
pi_omp *= w;
```

Listing 2.1: Implementation of the π computation using OpenMP

Listing 2.1 shows the main loop of the *pi* computation algorithm using OpenMP. Code is annotated using the `#pragma omp parallel for` (line 2). This portion of the code will be parallelized by the compiler using the memory options from the `private` and `reduction` clauses.

Initialization of `pi_omp` is performed sequentially while the loop is executed by all threads. Notice how the `pi_omp` is marked as a reduction var (`reduction` clause in line 5). The compiler will create a private copy of the variable in the local storage, and, after the loop has finished, it will add up all of these local copies into a single value in shared-memory that will be available to all threads.

2.1.1.2 SMPSs

SMPSs [112] is an instance of the StarSs [19] framework tailored for shared-memory multiprocessors. It combines a language with a much reduced number of OpenMP-like pragmas, a source-to-source compiler, and a runtime system to leverage task-level parallelism in sequential codes.

In SMPSs, the programmer employs pragmas to annotate certain routines (functions) appearing in the code as tasks, indicating the directionality of their operands (input, output or input/output) by means of clauses. The runtime exploits task-level parallelism by decomposing the code (transformed by the source-to-source compiler) into a number of tasks during the execution, dynamically identifying dependencies among these, and issuing *ready tasks* (those with all dependencies satisfied) for their execution in the cores of the system.

In order to parallelize this code with SMPSs, the programmer employs the `#pragma css task` directive to mark which functions will become tasks during the execution of the code. The associated clauses `input`, (`output`,) and `inout` specify the directionality of the function arguments, which help the runtime to capture all data dependencies among tasks.

SMPSs is part of StarSs. StarSs is an active project that targets multiple hardware platforms (Grids; multi-core architectures and shared-memory multiprocessors; platforms with multiple hardware accelerators: GPUs, Cell B.E., Clearspeed boards; heterogeneous systems, etc.) with distinct implementations of the framework. One particularly appealing version of this programming model is MPI/SMPSs, which provides specific support for MPI applications. In this particular version it is possible to embed calls to MPI primitives as SMPSs tasks, so that communications can be overlapped with computation transparently to the programmer. To indicate that a particular task comprises of a communication (e.g. a task that invokes a BLACS/MPI primitive), in MPI/SMPSs the developer employs the `device` clause, with the `comm_thread` option, (i.e. `#pragma css task target device (comm_thread)`). A separate thread devoted to communication is created dynamically by the runtime and those dependencies required to enforce the correct communication order are automatically added. Depending on the target platform, the SMPSs runtime may also configure the priority of the communication thread dynamically in order to improve performance.

```
pi_mpi = 0.0;
for (i = MPI_NAME; i < N; i += MPI_NUMPROCESSORS) {
    local = (i + 0.5) * w;
    pi_mpi += 4.0/(1.0 + local*local);
}
MPI_Allreduce(&pi_mpi, &gpi_mpi, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
```

Listing 2.2: Implementation of the π computation using MPI

2.1.2 Distributed-memory Based Systems

In distributed-memory based systems there are a set of processors with their own independent local memories. Processors communicate using an interconnection network. A correct map between the partitions of the data into the different processors, favouring locality and reducing communications is critical to achieve performance. The quality of the interconnection network and its topology is also important for improving the scalability of the codes. The most common programming model for these target architectures is message passing, whose major exponent is MPI, which is described in Section 2.1.2.1. Unfortunately, MPI is a completely explicit programming model which requires a significant amount of effort on the part of the programmer.

2.1.2.1 MPI

MPI (*Message Passing Interface* [95]) is most likely the most common programming interface in the world of HPC. It is based on the `fragmented model`. Several different instances of the program are executed simultaneously, which then communicate to each other via the MPI interface. Different programming paradigms are supported (one-to-one, one-to-many and all-to-all)

Within the scientific community MPI has become a very successful standard this is due to several factors: it is well defined, there are several different open-source implementations available, and there are also highly optimized versions available from mainstream vendors.

The MPI interface supports both `C` and `Fortran`, and there are interfaces for many other languages.

The main problem of the standard is the relatively high complexity of the model. Programmers are required to invest a lot of energy to port an existing code to MPI. The low-level approach to communication might be misleading and generate bad programming [65].

Listing 2.2 shows the MPI version of the implementation of the π algorithm shown in Listing 2.1.

A number of ranks (processors) will be launched when executing the program with the appropriate MPI wrapper (traditionally `mpirun` but this may vary across different implementations).

In order to fully benefit from the PM, the data to compute must be distributed to each rank. In this case, each processor will compute a subset of the total number of iterations and compute a partial value of pi. Next, using the MPI call `MPI_Allreduce`, all the processors will communicate their local value of pi and store the global sum.

Notice that it is no longer possible to execute this code without using MPI. Although it is still possible to understand the sequential algorithm for the MPI code in this motivational example, this may not be the case for more complex examples.

2.1.2.2 llc

As an alternative to MPI and OpenMP, our research group has designed `llc` [52] to exploit the best features of both approaches. `llc` shares the simplicity of OpenMP: users can start from a sequential code and parallelize it incrementally using OpenMP and/or `llc` directives and clauses. The code annotated with parallel directives is compiled by `llCoMP`, the `llc` compiler-translator, which produces an efficient and portable MPI parallel source code that is valid for both shared and distributed memory architectures. An additional advantage of `llc` is that all the OpenMP directives and clauses are recognized by `llCoMP`; what this means is that we have three versions in the same code - sequential, OpenMP and `llc`/MPI - and we only need to choose the proper compiler to obtain the appropriate binary.

In the past different directives have been designed in `llc` to support parallel constructs, such as `forall`, `sections` and `pipelines` [54, 49]. In previous studies [54] we have investigated the implementation of *Task Queues* in `llc` using well known problems from different fields. Many of these codes can be found in the OpenMP Source Code Repository [53].

The *OTOSP* (*One Thread is One Set of Processors*) model is the distributed memory computational model underlying the `llc` language. It is essential to know *OTOSP* in order to understand the `llc` implementation.

OTOSP is a distributed memory computational model in which the memory locations are private to each processor. One of its key concepts is that of the *processor set*. At the beginning of the program (and also in its sequential parts), every available processor in the system belongs to the same unique set. The processor sets follow a fork-join model of computation: the sets divide (fork) into subsets as a consequence of the execution of a *parallel construct*, and they join back together at the end of the execution of the construct. At any point in the code, all the processors belonging to the same set replicate the same computation, that is, they behave as a single execution thread.

When different processor (sub-) sets join into a single set at the end of a *parallel construct*, *partner processors* exchange the contents of the memory areas they have modified inside the *parallel construct*. The replication of computations performed by processors in the same set, together with the communication of modified memory areas at the end of the *parallel construct*, are the mechanisms used in *OTOSP* to guarantee a coherent image of the memory.

The simplicity of the *OTOSP* model greatly facilitates its implementation in distributed memory systems. Among other features, the model allows for an elegant implementation of nested parallelism. For example of this, see the recursive Quicksort implementation in the examples section at [73]. The key reference on the model is [52].

Having studied a wide set of parallel applications, we can conclude that the majority of them can be classified according to the parallelization paradigm used [40]. Moreover, the parallel code itself (data distribution, communications, etc.) is quite similar in applications following the same paradigm. `llCoMP` takes advantage of this feature to generate code. We have named these portions of reusable code *patterns*, and `llCoMP` uses two kinds of *patterns* in order to process the `llc` parallel constructs.

Static patterns are the *pure* parallel code and they do not depend directly on the application, but rather on the parallelization paradigm specified using the `llc` and/or OpenMP constructs. These codes are implemented in `llCoMP` using the target language (i.e. MPI) and they encode operations such as initialization of the parallel environment, resources distribution, data communications, load balancing, etc. The compiler adapts the *static patterns* to a specific translation using special tags in the pattern that the compiler fills with information coming from the source code directives.

To facilitate the maintenance of the compiler code, static patterns have been split into several files, each file implementing a specific stage of the entire pattern: initialization, execution, communication and/or finalization. In addition to the general case, `llCoMP` implements specialized code for some common situations; when it detects any of these situations, it uses the optimized code. For this reason, each of the stages can be supported through different files. The good performance delivered for the general case can be improved if an optimization is detected. As a result, each paradigm and its static pattern is implemented by several text files.

The static patterns need some extra code to work. This additional code handles the operations on data structures needed to build the translation, i.e. management of buffers used during communications. This code is sequential and specific to each application and therefore cannot be embedded in the static patterns. `llCoMP` uses *dynamic patterns* to produce this complementary code. The dynamic pattern code is generated by the compiler during the compilation process and stored in temporary files. The static patterns use special marks to indicate to the compiler the right position for inserting each temporary file to produce the target code. The dynamic patterns are carefully tuned to optimize parallel performance. For instance, data packing/unpacking greatly reduces communications.

It is through a combination of both static and dynamic patterns that `llCoMP` produces the target code. The compilation process is carried out by `llCoMP` without user intervention. All the required information is gathered from the parallel constructs in the source code. The compiler then selects the best combination of static and dynamics patterns for this code, including any available optimization, to build the target code.

The code in Listing 2.3 shows the calculation of π in `llc`.

When compiled by `llCoMP`, the loop (line 5) iterations are distributed among the processors. The clause `private` in line 3 is kept only for compatibility with OpenMP, since all the storages are private in the OTOSP computing model. The OpenMP clause `reduction` indicates that all the local values of variable `pi` have to be added at the end of the loop. This operation implies a collective communication among all processors in the OTOSP group and the updating of the variable with the result of the reduction operation. Since type analysis has not been included in `llCoMP`, the type of reduction variable has to be specified in line 4.

Notice that an OpenMP code can be adapted to `llc` with very few changes. In fact, all the OpenMP 2.5 directives and clauses are recognized by `llCoMP` and as such, we have three versions in the same code: sequential, OpenMP and `llc`/MPI. We need only choose the proper compiler to obtain the corresponding binary.

```
1 h = 1.0 / N;
2 pi = 0.0;
3 #pragma omp parallel for private(t) reduction (+: pi)
4 #pragma llc reduction_type (double)
5 for (i = 0; i < N; i++) {
6   x = (i + 0.5) * h;
7   pi = pi + 4.0 / (1.0 + t * t);
8 }
9 pi *= w;
```

Listing 2.3: Implementation of the π computation using `llc`

```
1  #pragma omp parallel for private(ptr, temp, k, j)
2  for (i=0; i<Blks->size1; i++) {
3    ptr = Blks->ptr[i];
4    temp = 0.0;
5    k = index1_coordinate(ptr); // First element in i-th row
6    for (j=0; j<elements_in_vector_coordinate(Blks, i); j++) {
7      temp += value_coordinate(ptr) * x[index2_coordinate(ptr)*incx];
8      inc_coordinate(ptr);
9    }
10   #pragma llc nc_result(&y[k*incy], 1, y)
11   y[k*incy] += alpha * temp;
12 }
```

Listing 2.4: A parallelization of the USMV operation

A more thorough example is shown in Listing 2.4. The code shows the parallelization using `llc` of the main loop of the *sparse matrix-vector product* operation $y = y + \alpha Ax$, where x and y are both vectors and A is a sparse matrix (this is known as operation USMV in the Level-2 sparse BLAS). Matrix elements are stored using a rowwise coordinate format, but we also store pointers to the first element on each row in vector `ptr`. In the code, each iteration of the external loop in line 2 performs a dot product between a row of the sparse matrix and vector `x`, producing one element of the solution vector `y`. The code uses three C macros (`index1_coordinate(ptr)`, `index2_coordinate(ptr)` and `value_coordinate(ptr)`) in order to access the row index, column index and value of an element of the sparse matrix pointed by `ptr`. A fourth macro, namely `inc_coordinate`, moves the pointer to the next element in the same row. Values `incx` and `incy` allow the code to access vectors `x` and `y` with strides different from 1.

A direct parallelization of the code can be obtained by taking into account that different dot products are fully independent. Therefore, a `parallel for` directive is used in line 1 to indicate that the set of processors executing the loop in line 2 has to fork to execute the loop. The `llc` specific directive `nc_result` in line 11 indicates to the compiler that the value of the *y[k*incy]* element has to be "annotated".

2.1.2.3 Partition Global Address Space (PGAS)

These languages use a memory model where there is a shared-memory among all the processors, but a portion is partitioned across the different processor. Each processor will have a portion of this global memory. The model is traditionally used in distributed memory systems.

PGAS languages are an improvement over the message passing libraries, providing abstractions to create shared data structures and communicate information across different program instances [148].

2.1.2.4 UPC

Unified Paralel C (`UPC`) [57] is a PGAS language. `UPC` is an extension of the C language that presents the developer with a common shared-memory space, partitioned across the different processors. Variables can be read and written by several threads, but each variable has only one physical associated processor.

The language defines the physical association between shared data items and UPC threads. This association, called affinity in UPC terms, indicates that a particular thread "owns" a particular data item. From the implementation point of view, affinity translates into storing data into the physical memory of the CPU where the UPC thread is running.

Scalar data has affinity with thread 0 by default, whereas for shared arrays the language allows three different affinity policies:

- `cyclic (per element)` - successive elements of the array have affinity with successive threads.
- `blocked-cyclic (user-defined)` - the array is divided into user-defined size blocks and the blocks are cyclically distributed among threads.
- `blocked (run-time)` - each thread has affinity to one contiguous part of the array. The size of the contiguous part is determined in such a way so that the array is "evenly" distributed among threads.

A new loop instruction is added to the traditional C iteration statements, the `upc_forall`. Iterations from the loop are assigned to different execution threads, following an affinity expression. UPC uses the SPMD (Single Program Multiple Data), where the parallelism is set when the program is executed, and typically each logical thread runs over one physical processor.

One of the main features of this language is the possibility of using pointers to the shared-memory space. Pointers can be declared as private (i.e. local to an execution thread) or shared. Pointers can be associated with global or private variables.

```
void main(void) {
    float local_pi = 0.0 ;
    int i ;
    l = upc_all_lock_alloc();
    upc_forall (i=0;i<N ; i++; i)
    local_pi += (float) f ((. 5 + i) / (N)) ;
    local_pi *= (float) (4.0 / N) ;
    upc_lock(l) ;
    pi += local_pi ;
    upc_unlock(l) ;
    upc_barrier;
    if (MYTHREAD==0) printf ("PI= %f\n" , pi) ;
    if (MYTHREAD==0) upc_lock_free (l) ;
}
```

Listing 2.5: Implementation of the π computation using UPC

The motivational π implementation in UPC is shown in Listing 2.5. Threads call collectively to the function `upc_all_lock_alloc()` to create lock `l`.

Each thread performs a partial add in its private variable, `local_pi`, and then a reduction is performed. A lock is set again in variable `l` to avoid race conditions on the `pi` variable. Finally, a global barrier is used to synchronize the overall execution.

The variable `MYTHREAD` is defined internally by the language and its value is set to the thread ID. The `THREADS` variable is set at compile time, in situations where the number of threads is known, or it can be set at runtime.

Porting existing codes to UPC might require a significant ammount of development effort. Data structures have to be redesigned to map to the UPC view of memory. The developer needs to be aware of situations with race conditions.

2.1.2.5 X10

`X10` [136] is a modern object-oriented programming language in the PGAS family.

Its fundamental goal is to enable scalable, high-performance programming for high-performance environments. `X10` is heavily based on object-oriented programming ideas, primarily to take advantage of their flexibility and ease-of-use.

`X10` takes advantage of several years of research on how to adapt OO languages to the context of HPC. It supports user-defined struct types, multi-dimensional arrays and IEEE standard floating point arithmetic.

Its syntax resembles Java, but it introduces ways to deal with concurrency, distribution and locality within an integrated type system. It is not strictly a PGAS language, as it extends the traditional PGAS definition with the concept of asynchronism (thus defining the APGAS programming model).

The main concept behind X10 is the place. A place is an abstraction for a computational context with a locally synchronous view of the shared-memory. Computations in X10 run over a large collection of places. Each place hosts some data and runs one or more activities (extremely lightweight threads of execution).

It is implemented using a source-to-source compiler which can generate C++ or Java sources. The generated source code perform calls to the X10 runtime.

Listing 2.6 shows the implementation of the Monte Carlo method to compute π. X10 programs follow the structure of Java programs. In this example, after handling the input parameters a parallel loop is spawned (`finish for`). Inside this loop the assignment expression to `inCircle` is marked as `async`. The *countAt* spawn some threads, each of which generates random points and returns the total number that fell inside the circle. Notice that the reduction operation is marked as `atomic` inside the *countAt* to ensure that the proper result is computed when combining the output of several threads.

2.1.2.6 Chapel

`Chapel` [34] is an emerging parallel programming language whose design and development is being led by Cray Inc. Chapel is being developed as an open-source effort with contributions from academia, industry, and scientific computing centres. Chapel emerged from the entry of Cray in the DARPA-led High Productivity Computing Systems program (HPCS).

Chapel is designed to improve the productivity of users of high-end users while also serving as a portable parallel programming model that can be used on commodity clusters or multi-core desktop systems. The purpose of Chapel is to improve the programmability of large-scale parallel programs without loosing performance.

Chapel supports a multi-thread execution model using high-level abstractions for data parallelism, task parallelism, concurrency and nested parallelism.

The `locale` type enables users to instruct the programming model about the placement of data and tasks on a target architecture, potentially improving its locality. Chapel supports reusing code through object-oriented design, type inference and some features for generic programming

To exploit parallelism, the user is presented with high-level abstractions. The compiler and runtime will be in charge of translating those high-level abstractions into real, low-level tasks.

Chapel supports data parallelism using a language construct known as a domain - a named, first-class set of indices that is used to define the size and shape of arrays and to support parallel iteration. `forall` statements can be used to iterate over aforementioned domains, allowing parallel operations over data transparently.

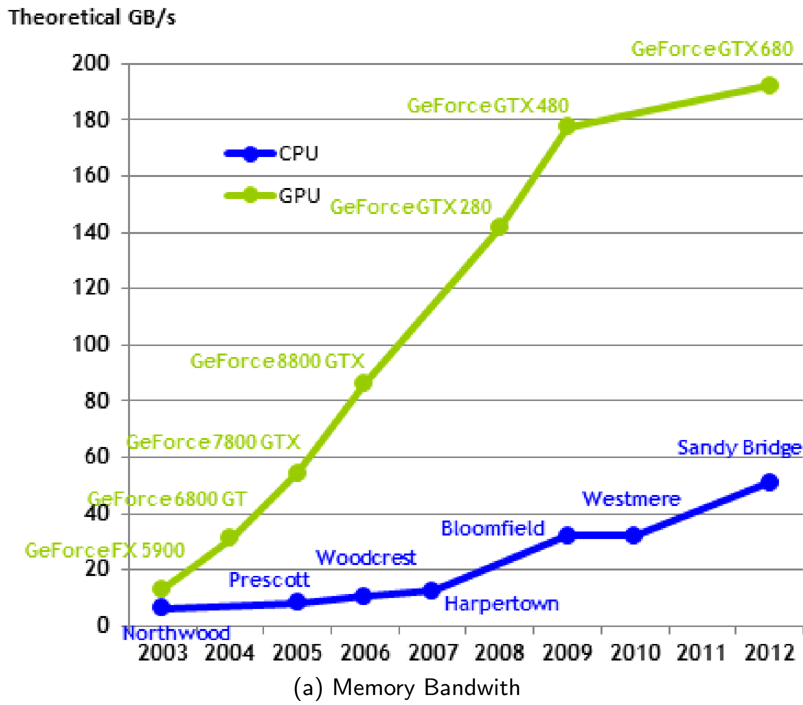
`Chapel` is based mainly on `High-Performance Fortran` (HPF) [74], ZPL [35] and several Cray extensions for `Fortran` and `C`. Listing 2.7 shows part of a Jacobi implementation in `Chapel`.

2.1.3 GPU Devices

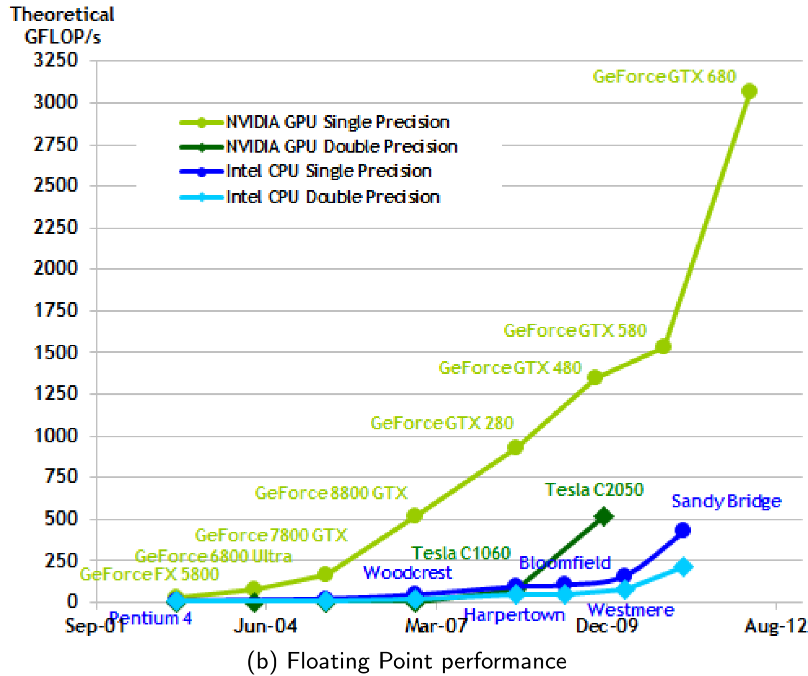
GPU devices, initially tailored to increase the performance of 3D graphics on classical hardware devices, have evolved into highly parallel platforms with an impressive (theoretical) computational power. Figure 2.2 shows the evolution of CPU hardware devices in comparison to GPU processors.

```
public class MontePiCluster {
      public static def countAtP(pId: Int, threads: Int, n: Long) {
      var count: Long = 0;
      finish for (var j: Int = 1; j<= threads; j++) {
          val jj = j;
          async {
              val r = new Random(jj*Place.MAX_PLACES + pId);
              val rand = () => r.nextDouble();
              val jCount = countPoints(n, rand);
              atomic count += jCount;
          }
      }
      return count;
  }
     public static def countPoints(n:Long, rand:()=>Double) {
      var inCircle: Long = 0;
      for (var j: Long=1; j<=n; j++) {
          val x = rand();
          val y = rand();
          if (x*x +y*y <= 1.0) inCircle++;
      }
      return inCircle;
  }
    public static def main(args: Array[String](1)) {
    val N = args.size() > 0 ? Long.parse(args(0)) : 1000000L;
    val places = args.size() > 1 ? Int.parse(args(1)) : Place.MAX_PLACES;
    val tPerP = args.size() > 2 ? Int.parse(args(2)) : 4;
    val nPerT = N/(places * tPerP);
    val inCircle = new Array[Long](1..places);
    finish for(var k: Int = 1; k<=places; k++) {
        val kk = k;
        val pk = Place.place(k-1);
        async inCircle(kk) = at(pk) countAtP(kk, tPerP, nPerT);
    }
    var totalInCircle: Long = 0;
    for(var k: Int =1; k<=places; k++)
        totalInCircle += inCircle(k);
    val pi = (4.0*totalInCircle)/N;
    Console.OUT.println("The value of pi is " + pi);
  }
}
```

Listing 2.6: Example of X10 source code



(a) Memory Bandwith



(b) Floating Point performance

Figure 2.2: Comparison of CPU and GPU devices

```
def main() {
    const ProblemSpace = [ 1..n,1..n] ,BigDomain = [ 0..n+1,0..n+1];
    var X ,XNew : [BigDomain] real = 0.0;
    X[n+1, 1..n] = 1.0 ;
    var iteration = 0 , delta : real;
    const north = (-1,0), south = (1,0),
          east = (0,1), west = (0,-1);
    do {
       forall ij in ProblemSpace do
          XNew (ij) = (X(ij+north) + X(ij+south) +
                X(ij+east) + X(ij+west)) / 4.0;
       delta = max reduce abs(XNew[Problem5pace] -X[Problem5pace]);
       X[ProblemSpace] = XNew[ProblemSpace];
       iteration += 1 ;
    } while (delta > epsilon);
}
```

Listing 2.7: Jacobi algorithm implementation in Chapel

Table 2.1: CUDA Architecture comparison

GPU	G80	G92a	GT200	GF100	GF104	GK104	GK110
Compute Capability	1.0	1.1	1.3	2.0	2.1	3.0	3.5
Reference board	8800GTX	9800GT	C1060	C2050	GTX560	GTX670	N/A
Year	Feb 2007	Feb 2008	Aug 2008	Jun 2010	Feb 2012	Feb 2012	-
Transistors (billion)	$<$ 1.0	$<$ 1.0	1.4	3.0	3.0	3.6	-
CUDA Cores	128	112	240	512	-	192*8	192*15
DP FP cap. (FMA ops/clock)	None	None	30	256	-	-	-
SP FP cap. (MAD ops/clock)	128	112	240	512	-	-	-
SFUs/SM	2	2	2	4	4	32	32
Warp Schedulers	1	1	1	2	-	4	4
Shared Memory per SM	16	16	16	48/16	48/16	48/32/16	48/32/16
L1 Cache per SM	16	16	16	48 or 16	-	-	-
L2 Cache	No	No	No	768 Kb	768 Kb	1536Kb	1536Kb
Load/Store address width	32 bit	32 bit	32 bit	64 bit	-	-	-
Clock rate	513Mhz	600Mhz	1600Mhz	3000Mhz	-	-	-
Memory Bandwith	60GB/s	57.6GB/s	102.4GB/s	144GB/s	-	192.2GB/s	-

The source of this huge performance gap between CPU and GPU is that GPU are devices designed for graphics, which are compute-intensive, highly parallel computations. This involves several compute-intensive and highly parallel computations. To improve the performance of these operations, more and more of the chip's surface was devoted to data processing without bothering with data caching , multitasking or flow control.

The arrival of the first CUDA-enabled device started a race to increase the performance and programmability of these devices. Table 2.1, shows a comparison of different CUDA-capable devices.

The second generation of CUDA began with the GT200 processors, and the widely-known Tesla C1060 boards. This generation defined the compute capability 1.3, with double precision floating point operations and an increased number of streaming processor cores (128 to 240). The number of registers and the efficiency of the memory access were improved.

In devices with compute capability below 1.3, memory coalescence had to be resolved manually by the programmer [129]. This required a huge amount of coding and design in order to take advantage of the processing power of the device. However, all the effort was in vain since it was useless on 1.3 or later devices. Coalescence problems on these devices were solved by hardware.

`Fermi` architecture appeared in 2010 (CUDA compute capability 2.0). NVIDIA applied a new approach during the design and implementation phases. Architectural differences are evident, with improved double precision performance, ECC support for professional environments, increased frequency and more cores, amongst other changes.

The main advantage of Fermi architecture is the use of a true cache hierarchy. Some applications adapt seamlessly to the previous memory hierarchy, however, there are memory-bound applications that are not able to take advantage of local storage, leading to poor performance in CUDA.

To avoid this, Fermi presents a configurable L2 cache, which is transparent to the developer. The only parameter that needs to be manually specified is the cache size, which can be either 16Kb or 48Kb. If the programmer wants to use shared-memory instead of cache, s/he can specify a shared-memory of 48Kb, forcing the cache to be 16Kb. However, if the programmer wants to use the transparent cache, s/he will specify 16Kb of shared-memory.

During the first quarter of 2012, NVIDIA announced a new architecture for its GPU cards, codenamed Kepler[103]. Theoretical performance of this architecture is up to 1 TFLOPs, while maintaining low consumption it has three times the performance per watt of the Fermi cards.

The GK110 architecture is set to replace the Tesla cards, and contains 15 multiprocessors (SMX) and six 64-bit memory controllers.

The new Kepler boards support the CUDA 3.5 compute capability, which increases the number of warps per multiprocessor from 48 to 64, and the number of threads to 2048. The new SMX multiprocessors feature more registers, thus some limits have been increased, for example, the maximum kernel grid size in the X-dimension will be $2^{32}-1$ instead of $2^{16}-1$.

One of most interesting features of this new board is the possibility of dynamically launching new kernels inside the GPU. The new architecture allows any kernel to launch another kernel from within, creating the necessary streams, events and manage dependencies needed to process additional work without the need for host CPU interaction.

This architectural innovation makes it easier for developers to create and optimize recursive and data-dependent execution patterns, and allows more of a program to be run directly on the GPU. The system CPU can then be freed up for additional tasks, or the system could be configured with a less powerful CPU to carry out the same workload.

This new feature is possible because of the new way in which the GPU interacts with the host CPU - via the Grid Management Unit (GMU). Figure 2.3 compares the Fermi interactions with the host with the new GMU-based workflow. This new workflow also allows several different independent grids to be launched and executed on the board, and to execute them simultaneously providing they have the required resources.

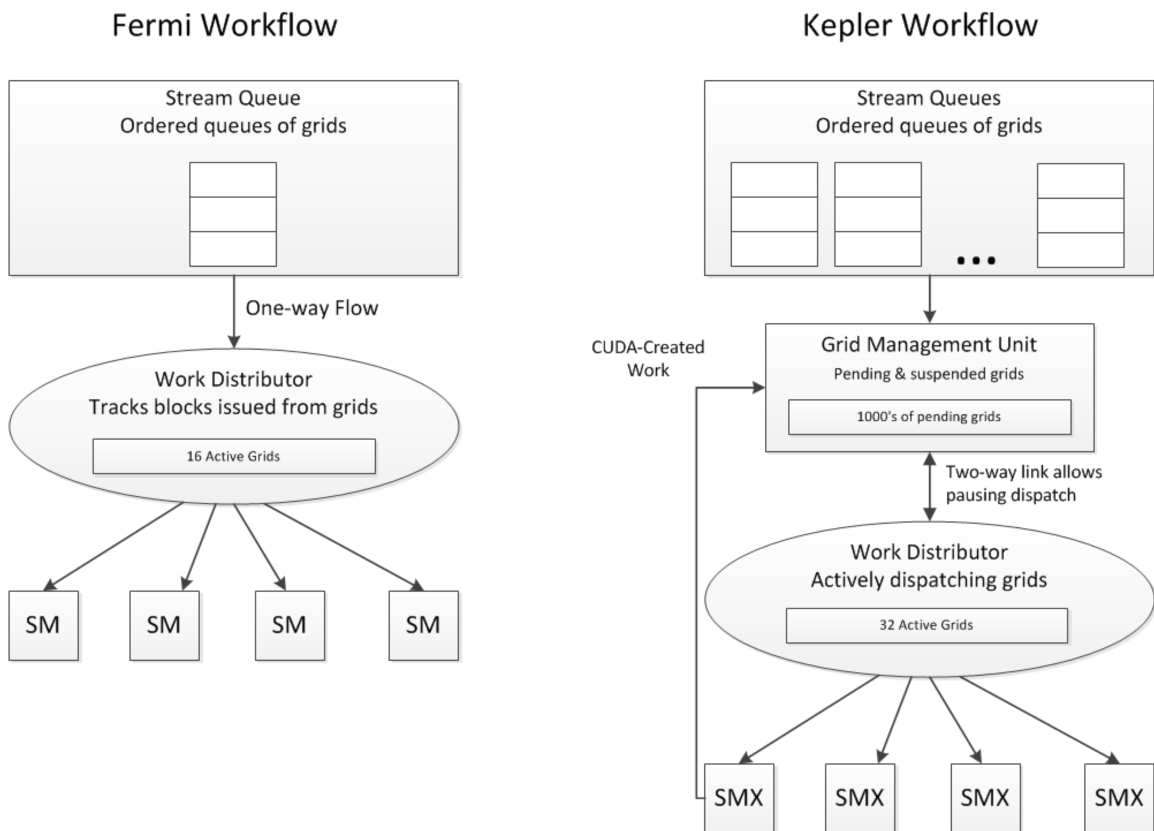


Figure 2.3: Comparison of the old Work Distributor used in Fermi with the new Grid Management Unit used in the new Kepler architectures

2.1.3.1 GPGPU : General Purpose GPU programming

The first attempt to use GPU processors for non-graphical applications was carried out in 2003 [59], although it was not until February 2007, with the appearance in the market of the first CUDA devices, that GPGPU became widespread. Before CUDA, the programmability of GPUs was a major drawback, and the lack of a general-purpose language posed major problems for developers who wanted to implement non-graphical algorithms [88].

As the architecture of GPU devices greatly differ from traditional CPUs, traditional approaches did not worked well when it came to tackling the massive parallelism inherent to the architecture.

2.1.3.2 CUDA Programming Model

NVIDIA attempted to improve the programmability by creating a high-level programming model for its devices - the CUDA Programming Model. This model was designed to overcome the challenge of developing applications that are able to transparently scale its parallelism to leverage the increasing number of processor cores. This is particularly critical in the GPU devices, as their evolution is quite impressive (see Table 2.1). Developers required a programming scheme capable of adapting to new architectures transparently.

The CUDA programming model works around three key abstractions: a hierarchy of thread groups, shared memories and barrier synchronization. They are exposed to the programmer as language extensions (shown in Section 2.1.3.2).

These abstractions provide fine-grained data parallelism and thread parallelism, nested within coarse-grained data parallelism and task parallelism. They guide the programmer to partition the problem into coarse sub-problems that can be solved independently in parallel by blocks of threads, and to then divide each sub-problem into finer pieces that can be solved cooperatively in parallel by all threads within the block.

Figure 2.4 shows how a CUDA program is executed on the CUDA architecture. The CUDA architecture is built around an array of Streaming Multiprocessors (SMs). Whenever a CUDA program launches a kernel, the blocks of the grid are distributed to multiprocessors with available execution capacity. The threads of a thread block execute concurrently on one multiprocessor. Providing the required resources are available multiple thread blocks can execute concurrently on one multiprocessor. Whenever a thread block terminates, remaining blocks are executed.

Multiprocesors are designed to execute several threads concurrently following the SIMT (Single-Instruction, Multiple-Thread) model. In this model, each multiprocessor executes threads normally in groups of 32 (might vary across architectures) called *warp*. The blocks assigned to the multiprocessor are partitioned into warps, with each warp containing threads of consecutive, increasing thread IDs. Although each thread of the warp can branch and execute a different instruction, maximum performance is achieved when the whole warp executes the same instruction. Otherwise the warp executes each divergent path sequentially disabling threads not required for the branch.

CUDA Runtime API

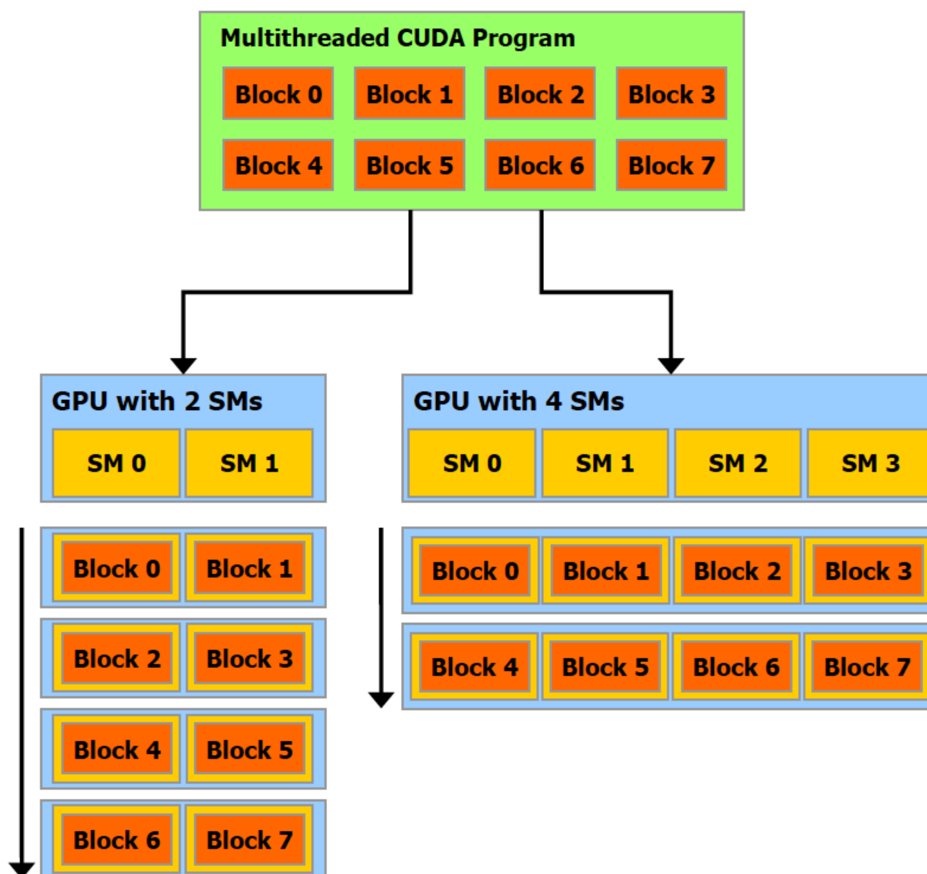


Figure 2.4: CUDA programs, partitioned into blocks of threads, can run on different GPU architectures without being modified. Depending on the available number of resources, more or less blocks may run in parallel

The most common way to develop software for CUDA devices is to use the CUDA Runtime API. It is a relatively high-level API that exposes the basic operations as C functions, and extends the C sintax with a specific statement to execute kernels defining the parameters of the grid (i.e. blocks and threads). An example of a vector addition of size N is shown in Listing 2.8.

```
1  // Kernel definition
2  __global__ void kernel_1(float * A, float * B, float * C) {
3    int i = threadIdx.x;
4    C[i] = A[i] + B[i]
5  }
6
7  int main() {
8    ...
9    // Kernel invocation with N threads
10   kernel_1<<<1, N>>>(a, b, c)
11   ...
12 }
```

Listing 2.8: Example of CUDA Runtime code

Kernels are defined using the `__global__` declaration specifier, and the number of CUDA threads that execute the kernel for a given call is specified using the execution configuration (`<<<...>>>`) syntax. Each thread executing the kernel is given a unique thread ID accesible through the threadIdx variable. threadIdx is a 3-component vector so that threads can be identified using up to three thread indexes, forming up to a three-dimensional thread block; the aim of this is to facilitate working with data structures like vectors, matrices or cubes. The maximum number of threads per block will vary depending on the hardware on which the device is executed. Blocks are also organized in one-, two- or three-dimensional grids of thread blocks. The number of thread blocks is determined by the size of the data being processed. There is also a hardware limit which will depend on the version of hardware being used. Figure 2.5 shows an example of two-dimensional partitioning of threads and blocks.

Thread blocks are required to execute independently, and to be run in parallel or sequentially (allowing them to be independently scheduled across multiprocessors). Threads within a block can cooperate by sharing data and by synchronizing their execution using intrinsic barrier functions.

Figure 2.6 shows the memory layout of a CUDA device. Each thread can access its own private registers and local memory, but threads within the same block can also use the shared-memory which is also inside the SM so it is faster to read. In addition, all threads can access the Global Memory (GM), but as it is outside the SM, access is several orders of magnitude slower than for the in-chip memories. If the developer requires fast read-only memory, s/he can use the Constant- or Texture- memory as well. Global, Constant and Texture memory can be accessed directly from the Host using the API. These memories have to be allocated from the Host so they can be accessed from the device.

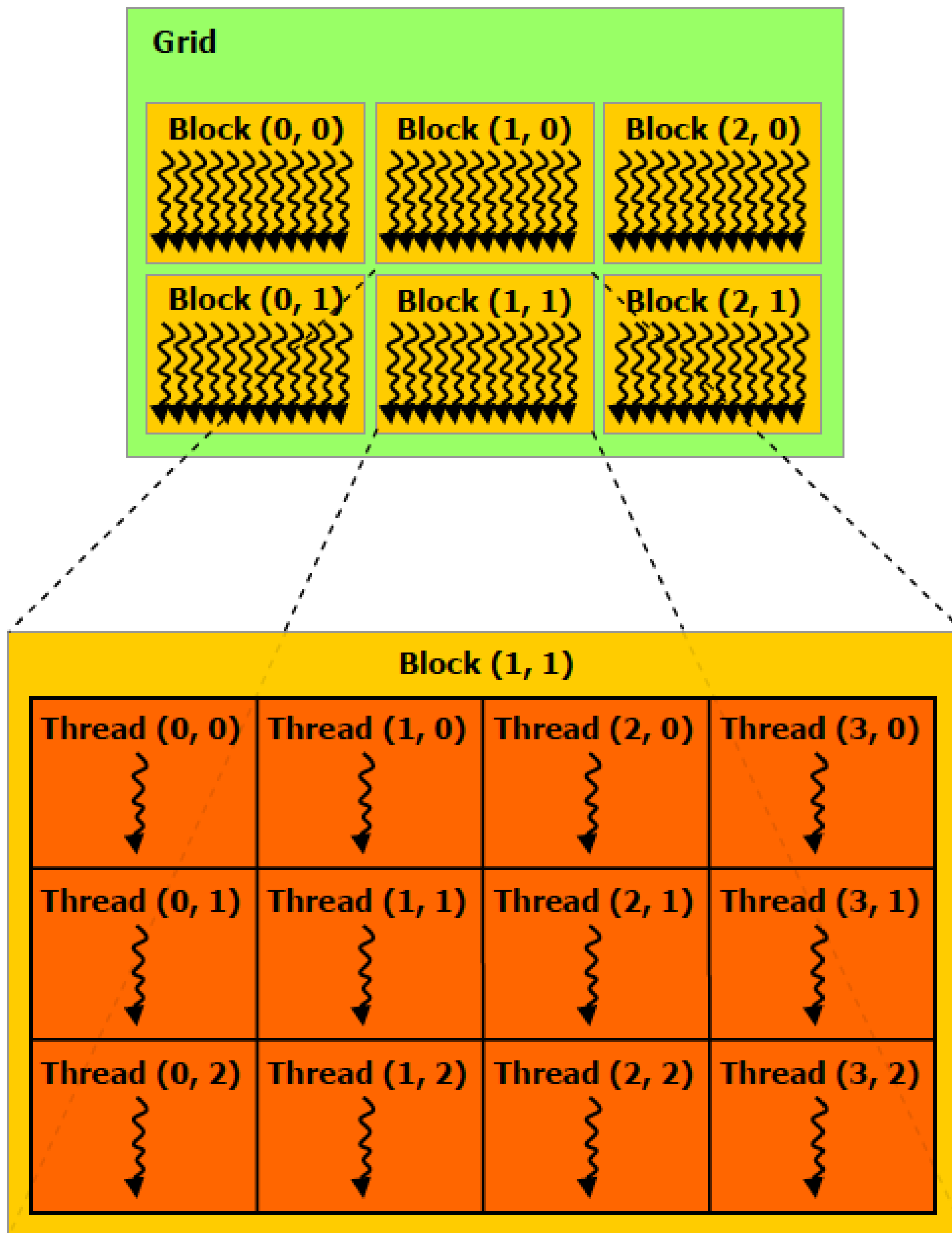


Figure 2.5: Example of a two-dimensional kernel grid

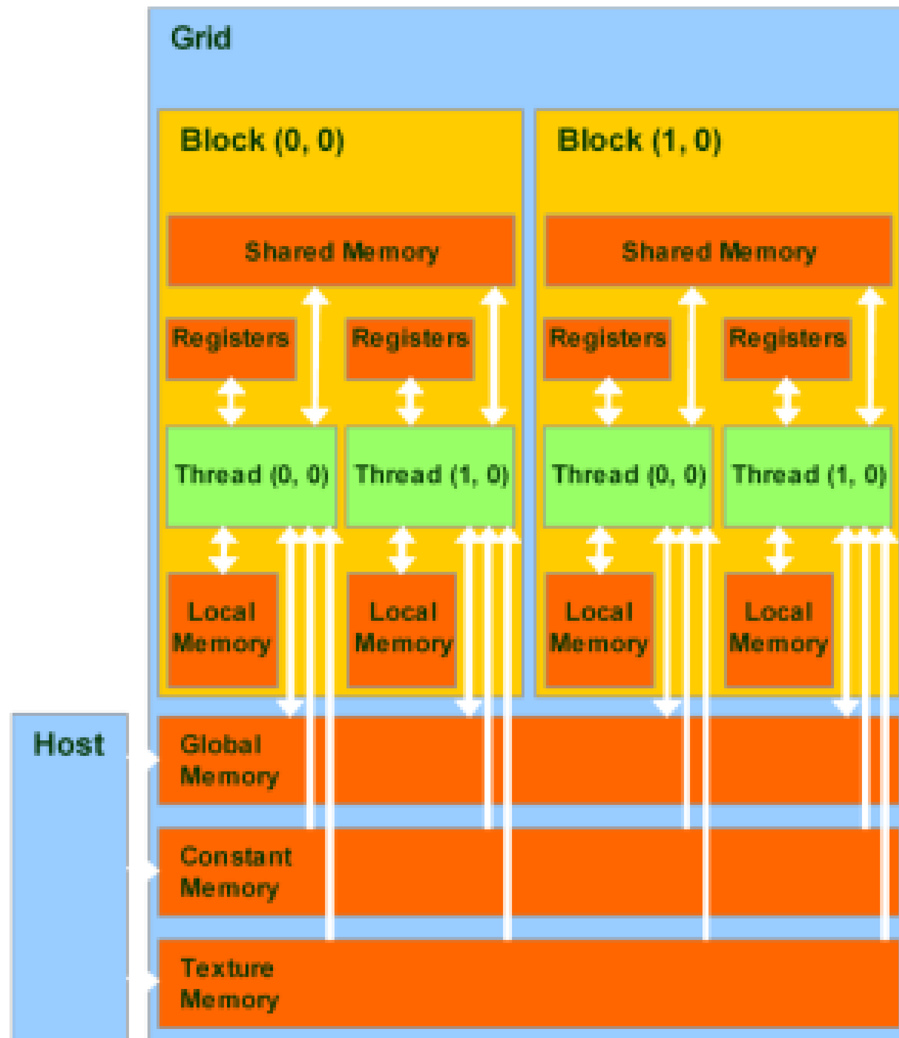


Figure 2.6: Memory hierarchy of the CUDA architecture

Interaction between the Host (CPU) and the GPU is performed using memory transfers (to Global-, Constant- or Texture-memories) and by invoking kernels. Figure 2.7 shows how a traditional C+CUDA code is run. The sequential parts of the code are executed on the CPU, and when a kernel is invoked, the parallel program is run on the GPU. Note that kernel executions are asynchronous to the Host, thus, sequential code can be run in the CPU simultaneously with the kernel execution. An API call to a barrier allows the synchronization of the Host and the device. Depending on the API call used, memory transfers are either asynchronous with respect to the Host CPU or not. One of the critical bottlenecks of CUDA programs is transferring the memory inside or outside of the device as since GPUs are traditionally connected to the host through a slow bus; this reduces the available bandwidth between Host and GPU devices and must be taken into account when working with these devices.

2.1.3.3 OpenCL

OpenCL [76] is an open standard for general purpose parallel programming across many-core architectures. OpenCL supports a wide range of applications, and serves as an efficient, close-to-the-metal programming interface. It aims to be the foundation layer of a parallel computing ecosystem of platform-independent tools, middleware and applications.

OpenCL consists of an API for coordinating parallel computation across heterogeneous processors and a cross-platform programming language with a well-specified computation environment.

The target audience for OpenCL are expert programmers with an interest in writing portable but efficient code. Four models (Platform, Memory, Execution and Programming) define the OpenCL standard.

Platform Model

The Platform model (Figure 2.8) consists of a host connected to one or more OpenCL devices. An OpenCL device is divided into one or more compute units (CU). CU are further divided into one or more processing elements (PEs). Computations on a device occur within the processing elements.

OpenCL applications runs on the host platform, and are able to submit commands from the host to execute on the PEs. The PEs execute a stream of instructions in SIMD or SPMD fashions, depending on the configuration.

OpenCL supports a wide variety of devices with different capabilities under a single platform. To be able to identify the resources available at the time of execution, the user can send a query to the platform to extract information regarding the device capabilities.

Execution Model

Execution of an OpenCL program is split into two parts: kernels executing on devices and a program executing in the Host. The host program defines the context for the kernels and manages their execution.

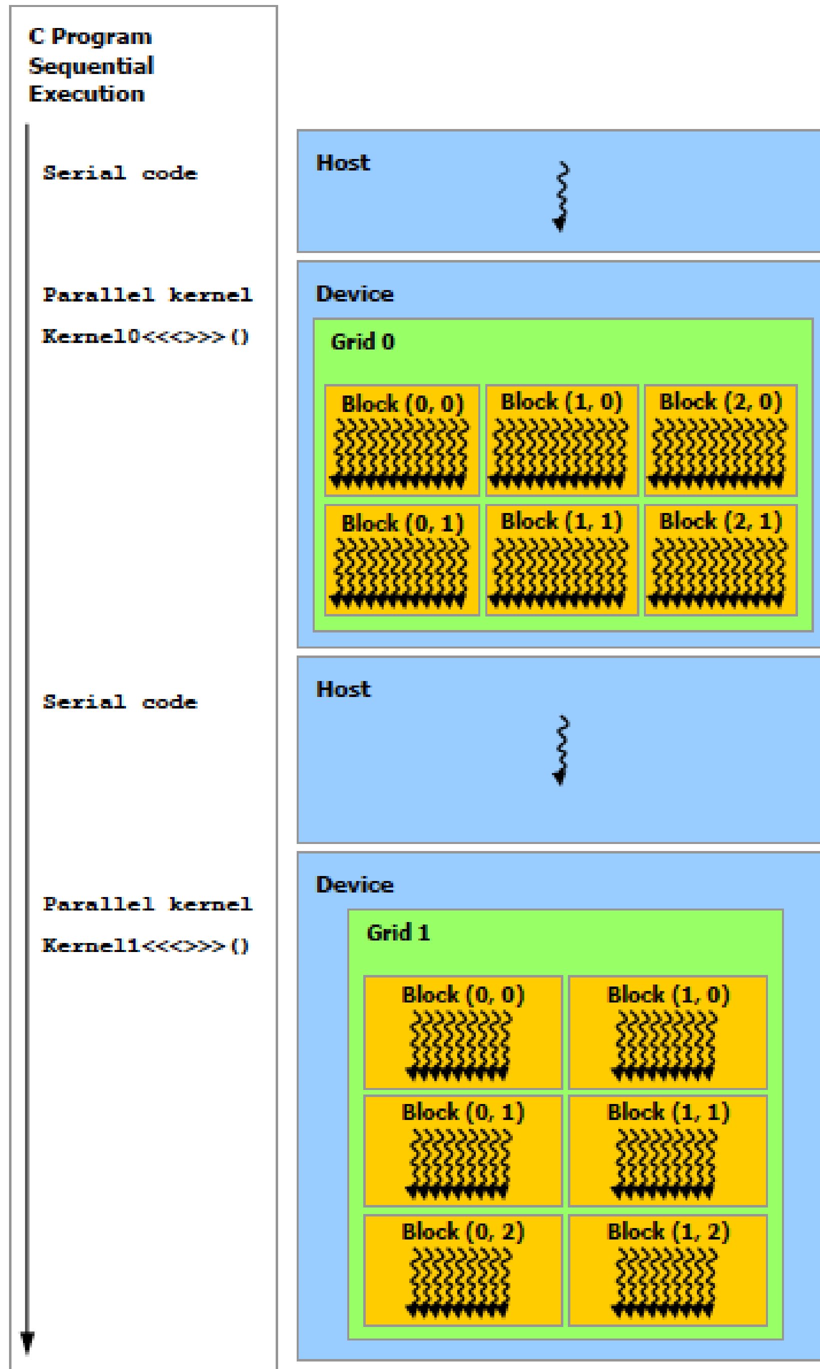


Figure 2.7: Execution of a GPU program from the Host

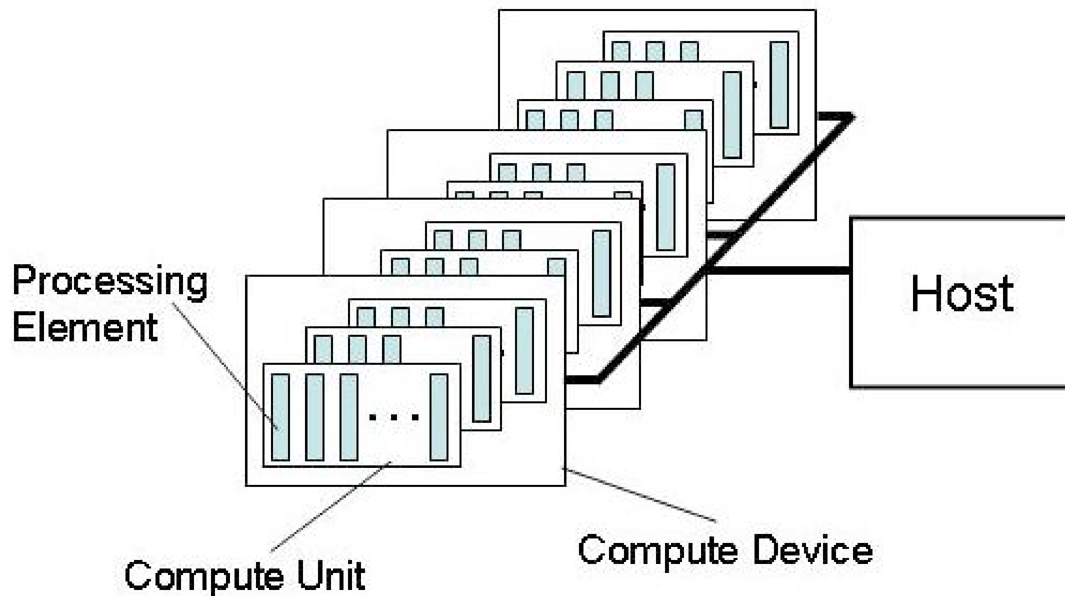


Figure 2.8: OpenCL Platform model

The critical idea behind the OpenCL execution model is the way the kernels are executed. Whenever a kernel is submitted for execution by the Host, an index space is defined. An instance of the kernel executes for each point in this index space, which provides a global ID for the work-item. Each work-item executes the same code, but the specific execution pathway through the code and the data operated upon can vary per work-item.

Work-items are organized into work-groups. Work-groups provide a more coarse-grained decomposition of the index space, and are assigned a unique work-group ID with the same dimensionality as the index space used for the work-items. A single work-item can be uniquely identified by its global ID or by a combination of its local ID and work-group ID. The work-items in a given work-group execute concurrently on the PE of a single CU.

OpenCL supports an N-dimensional index space (NDRange) where N is one, two or three. Each dimension can start at an offset. Each work-item's global and local ID are N-dimensional tuples. Figure 2.9 shows how the work-item, work groups and NDRanges are related to each other.

OpenCL supports the usage of native kernels alongside with OpenCL kernels (compiled with the platform compiler), allowing the use of pointers to native functions when submitting kernels to execute.

OpenCL kernels have access to four distinct memory regions:

- Global Memory: Accessible to all work-items in all work-groups, potentially cache. Initialized and accesible by the Host.

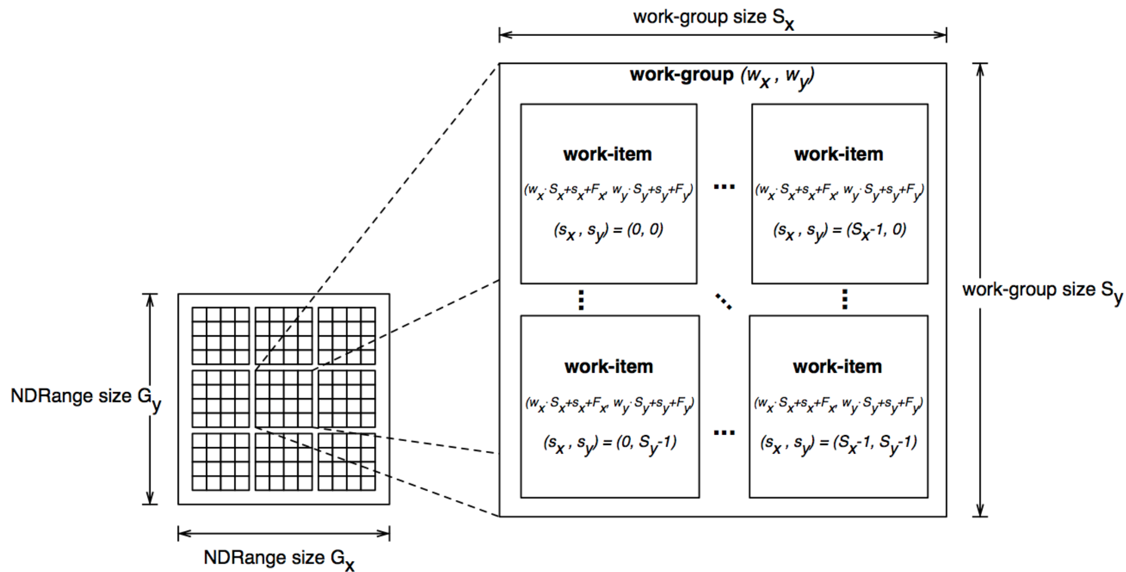


Figure 2.9: OpenCL Execution model

- Constant Memory: Region of global memory that remains constant during kernel execution. Initialized and accesible by the Host.
- Local Memory: Memory region local to a work-group. Can be used to implement memory shared by all work-items in that work-group.
  Is allocated from the Host, but it is not accesible.
- Private Memory: A region of memory private to a work-item, not accesible or allocatable from the Host.

The application uses the OpenCL API to allocate global memory and to enqueue memory operations over those memory objects. Apart from traditional copy data operations, it is possibe to map or unmap regions of a memory object to the host address space.

2.1.4 Directive-based Languages for Accelerators

CUDA and OpenCL provide reasonable abstractions to the underlying complexity of the hardware. In addition, OpenCL facilitates portability across multiple architectures. However, both of them face the same problem, which is that both the programmability and performance portability are limited.

It is necessary to have at least two separate versions of the code (one for the Host and another for the device), which decreases the maintainability of the code. Tunning also presents developers with difficulties, as different GPUs have different characteristics which need different tunning parameters, for example, optimal kernel grid sizes are different accross architectures [128].

Several attempts to leverage this situation have been made, and in this section we focus on those we consider to be of interest to our work.

2.1.4.1 OpenMPC - OpenMP Extended for CUDA

OpenMPC[82] is an extension of the OpenMP programming model designed and implemented by Seyong Lee and Rudolf Eigenmann from Purdue University. It includes several optimizations that deal with the architectural differences between traditional shared-memory systems and stream architectures.

As a front-end programming model, OpenMPC provides programmers with abstractions of the complex CUDA programming model and high-level controls over various optimizations and CUDA-related parameters. The authors have developed a fully automatic compilation and user-assisted tuning system supporting OpenMPC. In addition to a range of compiler transformations and optimizations, the system includes tuning capabilities for generating, pruning, and navigating the search space of compilation variants. OpenMPC is based on the Cetus compiler infrastructure [84].

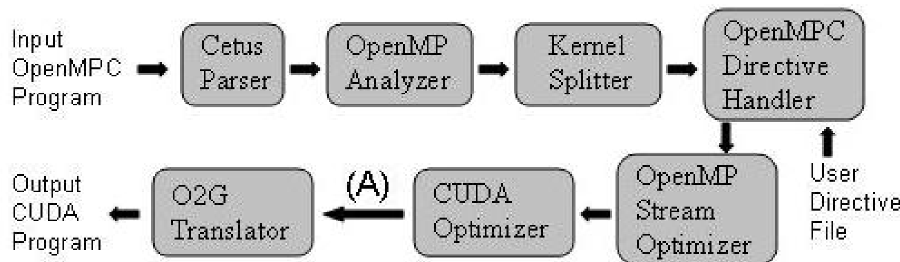


Figure 2.10: OpenMPC workflow

The baseline translation consist of two steps: interpreting OpenMP semantics under the CUDA programming model and identifying `kernel regions` (code sections to be executed on a GPU); and transforming eligible kernel regions into CUDA kernel functions, inserting the necessary memory transfer code.

Eligible kernel regions are found inside the OpenMP parallel regions, whereas `omp for` and `omp sections` constructs are used to partition the work among threads on the GPU. Once determined, kernel regions are extracted from the original source and replaced with calls to the new CUDA kernel functions. To partition the work, each iteration of the *omp for* loops and each section of *omp sections* are assigned to a thread, and the remaining code sections in a kernel region are executed redundantly by all participating threads. To decide how threads are associated with iterations, the source-to-source translator calculates the maximum partition size among parallel work contained in the kernel region. By default, the maximum partition size becomes the total number of thread blocks and the thread block size determine the mapping of threads onto SMs. Command line options or user directives can be used to tune these values, and the source-to-source translator will perform the appropriate tiling transformations to fit the work partition in the mapping of threads.

Mapping data from CPU to GPU is performed using OpenMP data sharing rules, but because the CUDA memory model allows several specialized memory spaces (such as constant or texture memory), some compiler transformations are implemented in order to fully exploit these on-chip memories. Interprocedural data flow analysis is used to identify redundant memory transfers from the GPU to the CPU.

The extensions to the OpenMP come in the form of directives and API calls. The OpenMPC optimization system uses these directives to pass information generated by various analysis passes to the actual OpenMP-to-CUDA translator. Several directives can be used to tune the performance of the codes.

Figure 2.10 shows the overall flow of the compilation. The *Cetus Parser* reads the input OpenMPC program and generates an internal representation. The *OpenMP Analyzer* recognizes standard OpenMP directives and analyzes the program to find all OpenMP data sharing clauses. The *Kernel Splitter* divides parallel regions at each synchronization point to enforce synchronization semantics under the CUDA programming model. The *OpenMPC-directive Handler* annotates each *kernel region* with a directive to assign a unique ID that can be used to reference it later from a configuration file. The *OpenMP Stream Optimizer* is used to transform traditional CPU-oriented OpenMP programs into GPU-optimized ones. Finally the *CUDA Optimizer* performs CUDA-specific optimizations.

The final *O2G Translator* performs the actual code transformations according to the directives provided either by a user or through the optimization passes.

2.1.4.2 hiCUDA

hiCUDA[68] (for high-level CUDA) provides the developer with a set of `pragmas` that map standard CUDA operations. Kernels are automatically extracted from the original source file, and iterations are distributed across threads and blocks according to loop partitioning clauses. It is possible to use shared-memory within kernels using a specific directive. One single source file can be used for sequential and GPU versions of the code. Calls to memory management routines are replaced by pragmas and the user does not need to keep track of device pointers.

The project provides a prototype compiler on top on the Open64 compiler infrastructure that is capable of translating hiCUDA programs to equivalent CUDA programs and the authors claim that the execution time of their generated code is within 2% of that of the hand-written CUDA version [67].

The hiCUDA driver parses the original source using the GNU-3 frontend (to which hiCUDA directives has been added) and then operates on Open64 IR (WHIRL) to replace the pragmas, extract the loops and inject the appropriate CUDA runtime calls. Finally, it uses the C code generator, with the extended CUDA syntax, to generate the target code.

A sketch of a Matrix Product code in hiCUDA is shown in Listing 2.9. Global arrays have to be allocated on GPU using the appropriate clause. In this case, the compiler was not able to properly detect the dimensions of the matrix, thus requiring the `shape` directive to indicate how the data is distributed in memory. The usage of shared-memory is exposed through the `shared` directive.

```
#pragma hicuda shape a [N] [N]
#pragma hicuda shape b [N] [N]
#pragma hicuda shape c [N] [N]
#pragma hicuda global alloc a [*] [*]
#pragma hicuda global alloc b [*] [*] copyin
#pragma hicuda global alloc c [*] [*] copyin

#pragma hicuda kernel mxm tblock(N/16,N/16) thread(16,16)
 #pragma hicuda loop_partition over_tblock over_thread
 for (i = 0; i < N; i++) {
 #pragma hicuda loop_partition over_tblock over_thread
  for (j = 0; j < N; j++) {
      double sum = 0.0;
      for (k = 0; k < N; k++) {
        sum += b[i+k*N] * c[k+j*N];
      }
      a[i+j*N] = sum;
    }
  }
#pragma hicuda kernel_end
#pragma hicuda global copyout a [*] [*]
#pragma hicuda global free a b c
```

Listing 2.9: Sketch of MxM in hiCUDA

Grid dimensions, specified by tblock and thread clauses, were selected so that each thread only performs one iteration. Varying tblock and thread allows the users to fine-tune the kernel configuration. To obtain maximum performance in different GPU architectures, these values have to be determined by hand. hiCUDA does a great job of leveraging loop partitioning from the user and it offers several scheduling possibilities for loop iterations.

As a result of the directives being an almost direct translation of the CUDA programming model, the user is forced to know more details about the underlying platform than in other approaches. This also makes it difficult to port hiCUDA to different accelerators, although this was probably never one of the intended functions. The result of compiling with the hiCUDA driver is not a binary file but a directory with a single CUDA source together with some required headers. This source has to be compiled with NVIDIA tools to generate the final binary file.

2.1.4.3 PGI Accelerator Model

The PGI Accelerator model [146] proposes a set of directives, resembling those used in OpenMP, that help compilers to generate GPU kernels. Most of the directives in the model are optional and they are used to improve performance. The only directive that is mandatory is the acc region which indicates a region containing loops with kernels. Data-flow, alias and array region analysis are used to determine which data need to be allocated on the accelerator and copied to and from the Host. Optional directives allow developers to provide additional information concerning variable directionality.

```
#pragma acc data copyin(b[0:N*N],c[0:N*N]) copy(a[0:N*N])
{
  #pragma acc region
  {
  #pragma acc for independent parallel
  for (j = 0; j < N; j++) {
  #pragma acc for independent parallel
    for (i = 0; i < N; i++) {
      double sum = 0.0;
      for (k = 0; k < N; k++) {
        sum += b[i+k*N] * c[k+j*N];
      }
      a[i+j*N] = sum;
    }
  }
  }
}
```

Listing 2.10: Sketch of `MxM` in PGI Accelerator Model

The PGI compiler maps loop parallelism to the hardware architecture using a Planner module [146] that uses information from other analysis passes present in the compiler. Strip-mining is heavily used to accomplish loop mapping and the user can use optional directives to force these loop transformations.

A naive Matrix Multiplication (`MxM`) implementation is shown in Listing 2.10. Data transfers are declared using copy directives. Array regions to be transferred are declared using a syntax that closely resembles the Fortran 90 array indexes.

The philosophy behind PGI seems to be *better safe than sorry*. Whenever the compiler cannot determine whether a loop can be run in parallel or not, it warns the programmer and generates a sequential version of the code. Programmers can force the compiler to generate parallel versions of the code using the optional `independent` clause.

At the time of compilation, PGI generates a single binary file that contains the CUDA kernel. Kernel parameters, like block grid and threads, are computed at compile time. If requested, the PGI compiler is able to show occupancy, block grid and thread configuration at compile time, amongst other information. It is also possible to show detailed profiling information after execution, this enables the user to have a general idea about the quality of the implementation.

It is worth noting that the PGI Accelerator model is available in both Fortran and C PGI compilers. In both cases, the compiler generates a single binary containing GPU and CPU versions of the code, thus, this binary can run on platforms without GPU. Although it is possible to keep the intermediate kernel files, they are not meant to be read by humans.

2.1.4.4 OpenACC

The OpenACC Application Program Interface [106] describes a collection of compiler directives to specify loops and regions of code in standard C, C++ and Fortran to be offloaded from a host CPU to an attached accelerator, which provides portability across operating systems, host CPUs and accelerators.

The directives and programming model defined in this standard allow programmers to create high-level host and accelerator programs without the need to explicitly initialize the accelerator, manage data or program transfers between the Host and accelerator, or initiate accelerator startup and shutdown.

All of these details are implicit in the programming model and are managed by the OpenACC API-enabled compilers and runtimes. The programming model allows the programmer to augment information available to the compilers, including specification of data local to an accelerator, guidance on mapping of loops onto an accelerator, and similar performance-related details. A more detailed explanation of the OpenACC API is available in Section 5.4.1.

2.1.5 Multi-target Programming Languages

Some languages have been designed from scratch to support heterogeneous architectures, thus effectively supporting multiple target platforms at the same time. We describe here the OmpSs [31] programming model, which represents a major step on extending the OpenMP programming model to heterogeneous architectures [18]. It has been successully used to port several different scientific applications.

2.1.5.1 OmpSs

The OmpSs [31] is a programming model aimed at extending OpenMP with support for asynchronous parallelism and heterogeneous platforms. It has been designed from scratch in order to integrate features of the different languages of the StarSs family into a single PM.

The OmpSs environment is built on top of the Mercurium compiler and Nanos++ runtime system.

Main data dependency synchronization

To enable asynchronous parallelism, the OpenMP task construct is extended with the input, output and inout clauses. This allows developers to specify the directionality of argument for each task, thus, enhancing the knowledge of which data the task is waiting for.

Directionality clauses may contain any `lvalue`. Tasks with an input clause with a given `lvalue` will not be able to run as long as a previously created task with and output clause containing the same `lvalue` has finished its execution. An inout clause is considered as having the same `lvalue` in both input and output clauses.

When a task is created, its dependencies are matched against the existing tasks, creating a task dependency graph at runtime.

Tasks are scheduled for execution as soon as all their predecessor in the graph have finished or at creation if they have no predecessors.

```
void foo (int *a, int *b) {
  for (i = 1; i < N; i++) {
    #pragma omp task input(a[i-1]) inout(a[i]) output(b[i])
      propagate(&a[i-1],&a[i],&b[i]);
    #pragma omp task input(b[i-1]) inout(b[i])
      correct(&b[i-1],&b[i]);
  }
}
```

Listing 2.11: Example of OmpSs directives

Language extensions

Listing 2.11 shows an example using the input and inout directionality clauses.

This code generates a task graph as the loop unfolds. As the program runs through the loop it produces the graph which is shown in Figure 2.11.

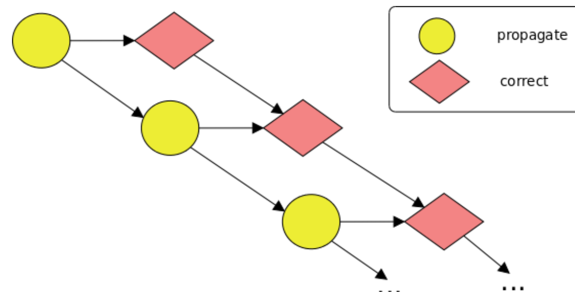


Figure 2.11: Task dependency graph generated by the OmpSs runtime

The use of the data dependency clauses allows the execution of tasks from the multiple iterations at the same time. It is possible to extend the `lvalue` given in the clause by using array sections or shaping expressions. Array sections allow the developer to refer to multiple elements of an array (or pointer data) in single expression. There are two forms of array sections:

- $[lower : upper]$ In this case all elements in the range of lower to upper (both included) are referenced. If no lower is specified it is assumed to be 0. If the section is applied to an array and upper is omitted it is assumed to be the last element of the array.
- $[lower; size]$ In this case all elements in the range of lower to lower+size (both included) are referenced.

The OpenMP `taskwait` construct is also extended with the on clause, enabling it to wait only for the tasks that produce some data in the same way as an input clause.

The task construct also supports the concurrent clause. The concurrent clause is a special version of the inout clause where the dependencies are computed with respect to input, output and inout, but not with respect to other concurrent clauses. As it relaxes the synchronization between tasks the programmer must ensure that either the task can executed concurrently or that additional synchronization is used.

Support for heterogeneous platforms

The syntax of the construct is `#pragma omp target [clauses]` , immediately followed by the task construct.

The valid clauses for the target construct are the following:

- **device**: Allows programmers to instruct the runtime where the task should run
- **copy_**{**in,out,inout,deps**}: Which data should be copied inside the device
- copy_out - It specifies that a set of shared data may be needed to be transferred from the device after the associated code is executed.
- **implements**: Indicates that the code of the task is an alternative implementation for the target device.

Several different target devices are available, for example, `SMP` for shared-memory machines, `CUDA` for CUDA GPU devices or OpenCL devices.

2.1.6 Final Remarks

We have presented a wide variety of languages, and classified them according to the main platform they are meant to be run on. Most of the programming models are based on a complete set with language, libraries and execution environment, such as Chapel or UPC. This represents a notable improvement for programmers and consequently for their productivity levels as they expose very high-level constructions and operations which greatly facilitates the creation of efficient parallel programs. However, porting existing codes requires a major rewrite of existing codes for these new environments.

Others are implemented just as an API, such as MPI, CUDA or OpenCL. Although this facilitates the programming effort of porting existing applications, exposing an API to programmers exposes them to the challenge of facing a complex, low-level set of code that does not provide the necessary abstractions to leverage the knowledge of the underlying platform.

Other languages, like OpenMP, OpenMPC, OmpSs, `llc` or OpenACC are based on directives. Directive-based programming languages allows developers to *incrementally parallelize sequential codes while maintaining a low development effort*. Directives can be added or removed by the user without significant alteration of the source code. The compiler can extract additional semantics from these directives and properly migrate the sequential code to a different set of platforms or technologies. This is critical to fill the gap between new programming paradigms and the huge scientific codebases that already exist.

One significant drawback of these programming models is that it may be possible that the existing sequential code does not implement an algorithm suitable for being parallelized. In this case, other algorithms have to be explored and a directive-based approach might not provide the best tool for the problem.

2.2 Compiler Support for Programming Models

Compiler support is required to implement most of the higher-level programming models that we have mentioned. The design and implementation of a compiler is not a trivial task, it requires tremendous amounts of work and significant amounts of patience to deal with developing quirks and hints in production codes. Some of the aforementioned programming models rely on the support of a commercial or an experimental compiler which is described in this section.

The extension of languages through the usage of directives requires a broad knowledge of compiler technologies and techniques. The previous work used ad-hoc compilers to extend a language with a set of features. Given the speed at which the HPC field is advancing, and the amount of new languages that are being developed, a significant amount of development and effort would be required to design ad-hoc compilers to cover specific languages or to extend features, and we would be unable to focus our attention on adding new features or investigating new research guidelines.

For example, with the irruption of GPU in the HPC arena [100], we decided to study different possible annotation schemes or language extensions to facilitate the automatic or directed generation of GPU code. Our previous `llCoMP` compiler was unsuitable for this task as it was designed ad-hoc to extend the OpenMP language with a particular set of features. With the experience adquired from working on `llCoMP` , and having comprehensively reviewed the bibliography, we detected the four key characteristics that we require in a research compiler in order for it to meet our needs:

- A flexible parser: We wanted to explore several different annotation schemes, language extensions and idioms, thus, a flexible front end where this modifications could be done quickly was our priority. As our work was for experimental purposes, we did not intent to parse commercial codes; our priority was not stability (i.e. speed and memory usage were not constraints).

- Portability: It should be possible to use the compiler on several different platforms, from laptops to clusters. In addition, different users, such as students or collaborators, should be able to use it without having to invest too much time and energy in learning how to use it. The compiler should be easily movable from one machine to another and it has to be written in a common and portable language.

- Debuggability: The user needs to be able to run the StS process step by step or be able to show what each phase is doing at any given point. One of the potential uses of this compilation framework is to teach compiler technology, thus it is desirable to be able to review or stop any process of the translation, so the user can easily see what is going on. As such, the inclusion of a graphical visualization of the Abstract Syntax Tree (AST) at any given moment would be a plus.

- Simplicity: Powerful but simple Object Oriented approach.

Before embracing ourselves with the titanic task of building our own compiler and run-time infrastructure, we evaluated different options and extensively reviewed the works in the bibliography.

The most relevant compiler infrastructures are detailed in the following paragraphs, and at the end of this section we conclude by providing a table comparing each of the aforementioned key features.

2.2.1 GCC

The GNU compiler collection (GCC) [66] is a compiler developed by the GNU project that supports various programming languages. It has been ported to a wide variety of processor architectures and is deployed in several different machines and platforms. It supports C, C++, Objective-C, Fortran, Java, Ada, Go and many others. Several tools are required for its construction (e.g. Perl, Flex, Bison, GMP, MPC ...), and some optimization passes are only enabled if external libraries are available. It can be extended with plugins, which can operate on the intermediate representation (GIMPLE).

Each of the language compilers is a separate program that inputs source code and outputs machine code. All have a common internal structure: a per-language front end parsers the source code in that language and produces an AST.

These are converted to the middle-end representation, which is gradually lowered towards its final, low-level form. GCC is mainly written in C.

2.2.1.1 General Usage/Workflow

GNU is organised into several passes. Language front end is invoked only once in order to parse the entire input, and may use different intermediate representations and language-specific tree codes. After finishing the parsing, the front end must translate the representation used to a representation understood by the language independent portions of the compiler. The C compiler calls the Gimplifier, while the Fortran translates its internal representation to GENERIC and then following this it is then lowered to GIMPLE. The conversion from GENERIC to GIMPLE is called *Gimplification*. After the code is in the language-independent IR, it is possible to call to different passes that are handled by the pass-manager. This package is in charge of running all of the individual passes in the correct order. Passes create several different parallel structures (like the control flow graph or the handlers for the OpenMP expansion). After the tree-optimization phases are completed, the code is transformed to RTL so register optimizations can be applied.

2.2.1.2 Intermediate Representation

GCC uses three different IR: GENERIC, GIMPLE and RTL. Front-end modules generate code into GENERIC, a High Level internal representation, which is then translated to a more manageable IR called GIMPLE.

GIMPLE is a family of IR based on the tree data structure. Two levels of this IR are implemented: High-Level GIMPLE - produced by the middle end when lowering the GENERIC language that is targeted by all the language front ends; and Low-Level GIMPLE - obtained by linearizing all the high-level control flow structures of high-level GIMPLE.

GIMPLE is then lowered to RTL (Register Transfer Language), which is an assembler language for an abstract machine with infinite registers. It represents low-level features (registers, memory addressing, bitfield operations, compare-and-branch ...), and it is commonly represented in a LISP-like form.

2.2.1.3 Unparsing

GCC does not offer unparsing features by itself, as it is meant to be a full source-to-binary compiler. Undoing the parsing could be possible at GENERIC level, but several semantic details are lost after translating to GIMPLE. It may be possible to obtain a C-like representation of GIMPLE by using the `-fdump-tree-gimple` flag, which is useful for debugging purposes. However, a more complete method is required for implementing a source-to-source translation system.

2.2.2 Open64

Open64 [36] is an open-source multi-platform compiler, derived from the SGI compilers. It was released as GPL in 2000, after which the University of Delaware took care of the project.

2.2.2.1 General Usage/Workflow

The compilation flow of Open64 is shown in Figure 2.12. Each step of the compilation flow is followed by a lowering of the IR (see Section 2.2.2.2). Open64 is a full source-to-binary compiler, thus, the end-point of the flow is assembler code suitable for generating the final binary.

The C and C++ front ends are based on GNU technology, while the Fortran one is the SGI Pro64 Fortran front end. Both provide a very high level IR for the input program units, stored as `.B` files. The VHO operates on this file (VHL level) to generate the phases that follow.

The LNO (Loop Nest Optimisation) module features several transformations to improve code performance by taking advantage of the Data Cache, e.g. transforming loop to work on sub-matrices that fit in the cache (Cache Blocking), loop interchange, etc. LNO also simplifies the expressions to facilitate the tasks of the following steps, generates SIMD code and vector intrinsics and also leverages OpenMP directives into intermediate code.

A driver controls the execution of the compiler, deciding which modules to load and executes the compilation plan. The driver is responsible for invoking all steps, and managing the modules input flags. Communication between modules is performed using intermediate temporary files.

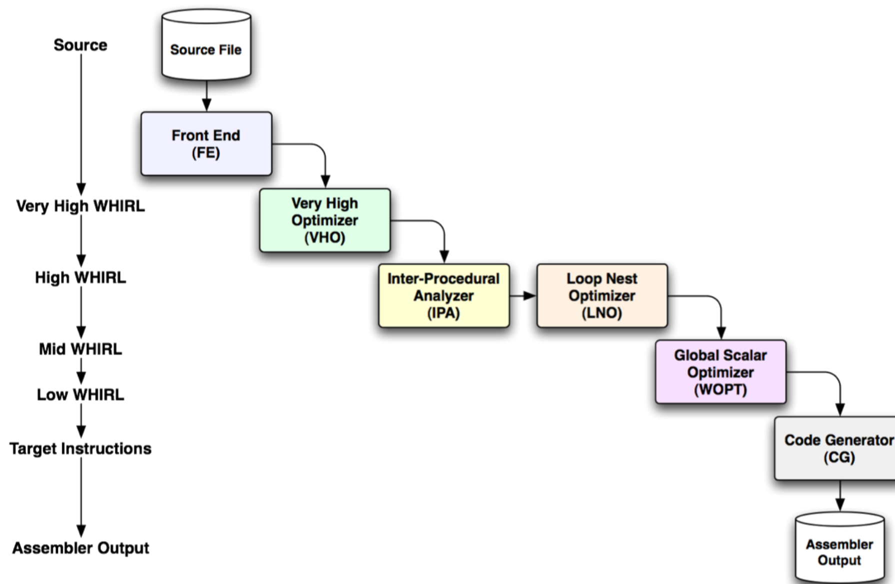


Figure 2.12: Compilation Flow of Open64

2.2.2.2 Intermediate Representation

The IR is called WHIRL, and it is composed of 5 levels (from the closest to the original to the closest to the binary): Very High (VH), High (H), Mid (M), Low (L), and Very Low (VL) level. The front end translate the original file into WHIRL which is then passed to the back end. Each optimisation is designed to work on a particular level of WHIRL, and WHIRL *lowerers* are called to translate WHIRL from the current level to the next lower level. Finally, the code generator translates the lowest level of WHIRL to its own internal representation that matches the target instruction. Because lowering is done only gradually, each lowering step is simpler and easier, which decreases the overall complexity of the translation. A description of the lowering actions for each level is shown in Figure 2.13. A WHIRL file generated by the front end consist of WHIRL instructions and WHIRL symbol tables. The instructions contain references to the symbol table. WHIRL instructions are linked in tree form.

2.2.2.3 Unparsing

Very High level WHIRL can be translated back to C and Fortran source code using the appropriate Open64 tools, but almost no optimisation is performed at this level. It is possible to export the intermediate representation to a file, or to output different stages of the compilation.

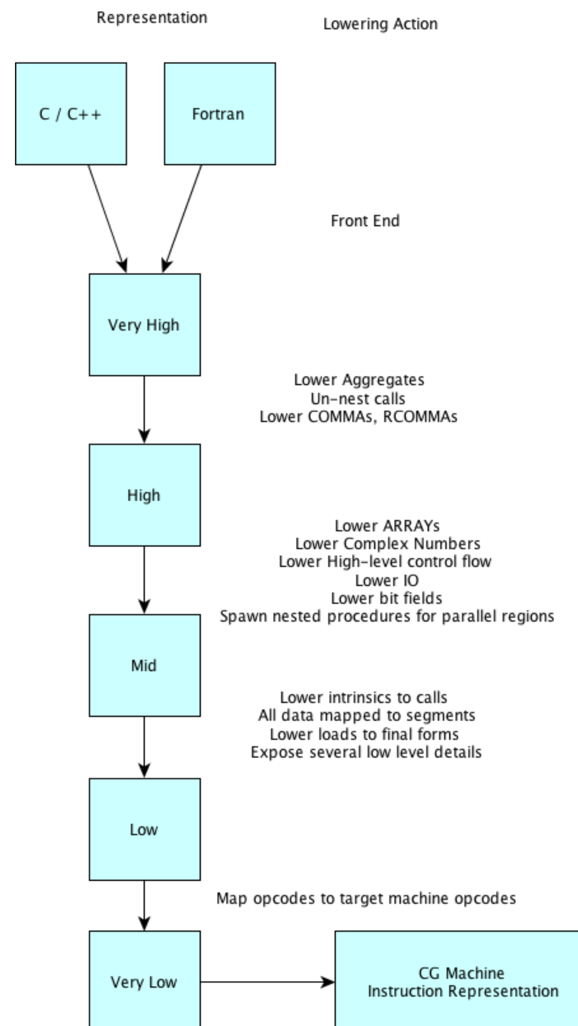


Figure 2.13: Levels of WHIRL and Lowering actions

2.2.2.4 Querying and AST Traversal

The class *WN_TREE_ITER_base* implements a STL-compatible iterator for WHIRL containers (*WN_TREE_CONTAINER*). It is possible to define different traversal orders (pre-order or post-order iteration). Comparison between two trees of WHIRL containers is possible through the overloaded operators. To apply a particular operation to the WHIRL tree, the user can call the function *WN_TREE_WALK*, which receives a WHIRL node, an operation and an instance of *WN_TREE_ITER* to indicate how to traverse the tree. The operation is executed for each WHIRL node. Listing 2.12 shows an example of calling a Tree Traversal to implement a simple node counting operation. Querying for a node or a particular set of node can be implemented using Operations or by implementing a derived class from *WN_TREE_ITER_base*.

```
// function object example: count the number of whirl nodes
struct WN_count {
  INT num_nodes;
  WN_count() : num_nodes(0){}
  void operator()(WN*) {
    ++num_nodes;
  }
};

int main () {
 ...
 WN_TREE_walk_pre_order (wn, WN_count());
 ..
}
```

Listing 2.12: Open64 Tree Traversal example

2.2.3 LLVM

LLVM [81] (Low Level Virtual Machine) is a compiler infrastructure written in C++. It supports several different kinds of optimizations for programs written in arbitrary programming languages. Several different front ends have been created which take advantage of the language-agnostic design, such as Objective-C, Fortran, Ada, etc.

To achieve this language-agnostic design, it is necessary to have a common intermediate representation in which code optimisations can be applied without the need for specific-language optimisations. The IR of LLVM is a Static Single Assignment (SSA) form with a simple, language-independent type system that exposes the primitives commonly used to implement high-level language features.

LLVM is compatible with standard makefiles and can use GCC as a C and C++ parser. Object files compiled with LLVM can be linked with object files built with gcc using the LLVM linker. Notice that LLVM object files contain LLVM IR/bytecode, not machine code.

2.2.3.1 General Usage / Workflow

Figure 2.14 illustrates the compilation worfklow of a typical LLVM driver.

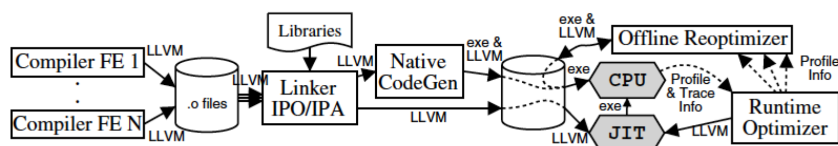


Figure 2.14: LLVM Workflow

After collecting the command line options, which instruct the compiler driver about the passes that it should run, the driver reads the configuration files for each pass - this will vary depending on the kind of input files pointed by the user. These configuration files can be provided by the user or by the tools. Each phase that is going to be executed can result in the invocation of one or more actions. An action is either a whole program or a function in a dynamically linked shared library. In this step, the driver determines the sequence of actions that must be executed. Actions will always be executed in a deterministic order. The actions required to support the original request from the user are executed sequentially and deterministically. All actions result in either the invocation of a whole program to perform the action, or the loading of a dynamically linkable shared library and invocation of a standard interface function within that library.

The compiler driver (`llvmc`) splits every compilation task into the following five distinct phases:

- **Preprocessing**: This phase can be invoked for those languages supporting preprocessing.
- **Translation**: Converts the source language input into the IR.
- **Optimization**: All optimizations are performed on the IR, according to the options provided.
- **Linking**: The inputs are combined to form a complete program.

2.2.3.2 Intermediate Representation

The LLVM code representation describes a program using an abstract RISC-like instruction set but with key high-level information that enables effective code analysis. This includes type information, explicit control flow graphs, and an explicit dataflow representation [42]. The main features of the LLVM IR are:

- A low-level, language-independent type system that can be used to implement data types and operations from high-level languages.
- Instructions for performing type conversions and low-level address arithmetic while preserving type information.
- Two low-level exception-handling instructions for implementing language specific exception semantics.

The LLVM code representation is designed to also be used as a human readable assembly language representation, facilitating development and debugging.

LLVM programs are composed of modules, each of which is a translation unit of the input programs. Each module consists of functions, global variables and symbol table entries. Modules can be merged together by the LLVM linker. Listing 2.13 shows an example of the `Hello World` module.

```
; Declare the string constant as a global constant.
@.str = private unnamed_addr constant [13 x i8] c"hello world\0A\00"

; External declaration of the puts function
declare i32 @puts(i8* nocapture) nounwind

; Definition of main function
define i32 @main() { ; i32()*
  ; Convert [13 x i8]* to i8 *...
  %cast210 = getelementptr [13 x i8]* @.str, i64 0, i64 0

  ; Call puts function to write out the string to stdout.
  call i32 @puts(i8* %cast210)
  ret i32 0
}

; Named metadata
!1 = metadata !{i32 42}
!foo = !{!1, null}
```

Listing 2.13: LLVM Code Representation

2.2.3.3 Unparsing

LLVM has been designed as a full source-to-binary compiler. Although it is possible to extract the different levels of Intermediate Representation to Text Files, it is not possible to recover the original language from these representations as it is completely agnostic to the original source language.

Theoretically speaking, it would be possible to use the IR to recreate a new source and port it to another language by creating an IR-to-language converter. However, as far as we are aware such a tool does not exist.

2.2.3.4 Querying and IR Traversal

It is possible to traverse the internal representation of LLVM using different iterator classes available in the framework. The following is a list of the available iterators:

- `Module::iterator` Iterates through the functions in the module (source file)
- `Function::iterator` Iterates through basic blocks in the module
- `BasicBlock::iterator` Iterates through instructions in a block

These iterators can be extended to create new ones. The code motion is implemented with operations on the nodes (`EraseFromParent`, `RemoveFromParent`, etc). Data dependency, Call Graph and Alias information, among others, can be printed to a Dot graph then to an external file.

2.2.4 ROSE

ROSE [86] is an open source compiler framework designed to facilitate building programs for applying source code transformations. It is primarly tailored to design and implement static analysis tools, source code transformations, loop optimizations, performance analysis and even static security checks of source codes. It has support for C, C++, Fortran and OpenMP.

As usual, ROSE uses a three layer approach (front end, middle end and back end). The result of running a ROSE driver is the generation of a new source code.

ROSE uses the front end of the Edison Design Group (EDG) for C and C++, whereas Open Source Fortran is used to generate the AST from Fortran sources.

Although the license of the front ends is not free, it is possible to redistribute it with the framework. The resulting IR of this front end is translated into a AST. This AST preservers most of the original source information (i.e. comments, preprocessor directives, original line numbers and so on), making it possible to completely *unparse* the AST into a source file.

2.2.4.1 General Usage / Workflow

Figure 2.15 illustrates the different compiler phases of the ROSE compiler framework. After parsing the source, ROSE converts the IR from the EDG front end into the AST. This AST is further processed by several transformation and optimization tools. Developers can add new tools or implement new transformations extending the class hierarchy.

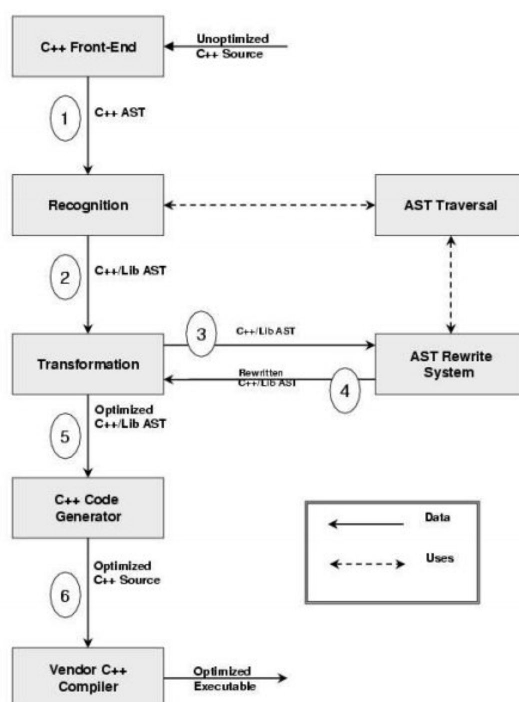


Figure 2.15: Usual Workflow of a ROSE driver

```
1  // Basic ROSE translator
2  #include "rose.h"
3  int main(int argc, char **argv) {
4      // Build the AST used by ROSE
5      SgProject* sp = frontend(argc, argv);
6      // Run internal consistency test on AST
7      AstTests::runAllTests(sp);
8      // Generate source code from AST and call the vendor's compiler
9      return backend(sp);
10 }
```

Listing 2.14: ROSE driver example

Listing 2.14 shows an example driver routine. Line 5 invokes the front end, which returns the AST of the original program. This AST is *unparsed* to rebuild the original source.

2.2.4.2 Intermediate Representation

The internal representation of ROSE is SAGE III (derived from SAGE++, [28]). SAGE III is automatically generated with a tool included within ROSE. When a code is parsed by the front end, a connection code translate EDG IR into SAGE III. This enables to distribute EDG binary files along ROSE while respecting licensing concerns.

2.2.4.3 Unparsing

As mentioned before, ROSE outputs a source file. It attempts to produce an output file as close to the original as possible, however, some differences may arise. Most of these differences come from the fact that there is more than one way to write the same expression in the original language. For reasons that will be described in the following section of this document, it is worth highlighting the following differences: (1) Variable declarations are normalized to separated declarations, (2) Normalization of member access from a pointer, and (3) Array indexing is represented through pointer arithmetic.

2.2.4.4 Querying

One of the critical features of a StS tool is the ability to search for particular nodes or subtrees. In ROSE, this is called `querying`. ROSE offers several querying features for nodes and/or subtrees. Common queries are already predefined (for example, looking for a particular variable declaration), and a developer can implement their own queries implementing the *NodeQuery* interface.

2.2.4.5 AST Traversal

ROSE aids the library writer by providing a traversal mechanism. This mechanism visits all the nodes of the AST in a predefined order. It can be used to compute attributes or to perform an analysis of the code.

```
class MyVisitor:
public AstSimpleProcessing {
  protected :
  void virtual visit (SgNode* node) ;
}

MyVisitor::visit (SgNode* node) {
  cout << node->get_class_name() << endl;
}
```

Listing 2.15: ROSE Traversal example

Based on a fixed traversal order, the framework provides inherited attributes to pass information down the AST (top-down processing) and synthesized attributes for passing information up to the AST (bottom-up processing). Inherited attributes can be used to propagate context information along the edges of the AST, whereas synthesized attributes can be used to compute values based on the information in the subtree. One function for computing inherited attributes and one function for computing synthesized attributes must be implemented when attributes are used.

Different interfaces are provided which will allow either one, both, or no attributes to be used; in the latter case it is a simple traversal with a visit method called at each node. AST Traversal is offered through the *AST*Processing* classes. An example is shown in Listing 2.15.

2.2.5 Cetus

Cetus [44] is a StS framework designed to implement code transformations. The current version supports ANSI C via ANTLR 2 [110], and it is implemented in Java. Cetus derives from the original POLARIS [109] compiler framework, although the Cetus project attempts to be more general than the previous one.

2.2.5.1 General Usage / Workflow

As usual, Cetus is divided into three different layers; the front end, the middle end (with the IR) and the back-end. Figure 2.16 illustrate the Cetus architecture.

Compiler writers usually only need to extend the *Driver* class to implement any kind of transformations, as shown in Listing 2.16

2.2.5.2 Intermediate Representation

In Cetus, the IR follows a hierarchical statement structure, directly reflecting the block structure of a program. A class hierarchy is used to implement the IR. Although this block structure and the subsequent implementation of the *Unparsing* and *Transversal* classes make it difficult to implement languages other than C, some additional abstract classes are provided to ease this task. A C++ front end is in progress, and there are plans to implement Java and Fortran90 back ends.

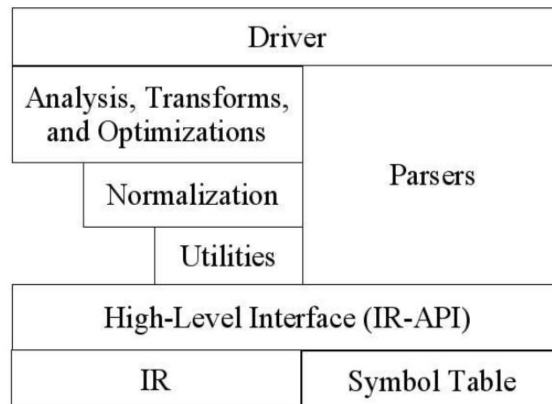


Figure 2.16: Cetus architecture

The IR has been designed to be easily understood by users. The contents of a source file are stored in `traslation units` and `procedures` represent individual functions. Procedures include a list of simple or compound statements, representing the program control flow in a hierarchical manner. Each node of the IR can be annotated with comments or directives.

2.2.5.3 Querying and Traversal

Iterators following the Java Programming style are available. Listing 2.17 shows an example of a DepthFirst iteration. Other iterators, such as Bread-first or Flat are available. The `next(c)` method returns the next object of the class c.

Other traversals apart from basic syntactic order are available. For example, it is possible to instantiate a caller traversal, which iterates across the calltree of a program.

Querying is implemented by means of iterators. The *instanceof* Java operator can be used to check if a particular node of the IR is an instance of a class.

2.2.6 Mercurium

Mercurium [61] is a StS compilation infrastructure designed to implement the StarSs programming model [111], although it has been extended as well.

Mercurium is composed by a set of plugins written in C++ that are automatically loaded by the compiler following instructions from a compilation configuration file. High-level transformations at IR level are implemented similarly to those available in Cetus and ROSE.

2.2.6.1 General Usage / Workflow

Figure 2.17 shows the Mercurium workflow. The compiler receives input in the form the source code written in C, C++ or Fortran and processes it through the front end, to generate a high-level interface. Later, a set of transformations are applied. Some parts of the original code may be outlined to an external file, enabling other back ends to process these external parts.

```
public class MyDriver extends Driver {
  public void run(String[] args) {
    parseCommandLine(args);
    parseFiles();
    if (getOptionValue("parse-only") != null) {
      System.err.println("parsing finished and parse-only option set");
      Tools.exit(0);
    }
    runPasses();
    PrintTools.printlnStatus("Printing...", 1);
    try {
      program.print();
    } catch (IOException e) {
      System.err.println("could not write output files: " + e);
      Tools.exit(1);
    }
  }
  public static void main(String[] args) {
    (new MyDriver()).run(args);
  }
}
```

Listing 2.16: Java code to create a driver using the Cetus API

```
/* Iterate depth-first over program which is instance of Program */
DepthFirstIterator dfs_iter = new DepthFirstIterator(program);

while (dfs_iter.hasNext()) {
  Object o = dfs_iter.next();
  if (o instanceof Loop) {
    System.out.print("Found instance of Loop");
  }
}
```

Listing 2.17: Java code to iterate through the Cetus IR and print a message whenever a `for` loop is traversed

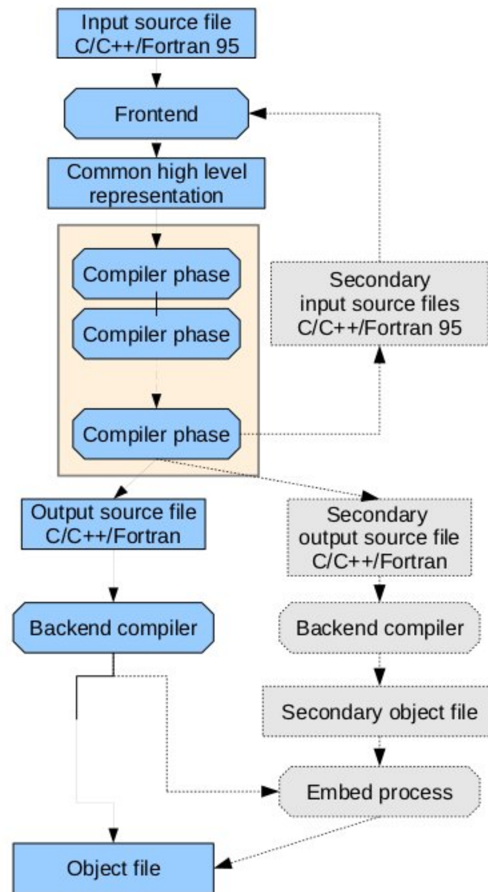


Figure 2.17: Mercurium workflow

As in Cetus (see Section 2.2.5) Mercurium also features a component that is responsible for high-level transformations. These code transformations are aimed to code optimization. Different loop transformations are available to the programmer through the `#pragma hlt` directive. Listing 2.18 shows an example of the directive to perform *loop-unrolling*.

Listing 2.19 shows an example of *loop collapse* in Mercurium: given a perfect nest of regular loops, loop collapse creates a single loop that iterates over the n-dimensional iteration space.

Several other loop transformations (blocking, interchange, fusion, distribution, etc.) are also available in mercurium through the `#pragma hlt` directive.

2.2.6.2 Intermediate Representation

The IR of Mercurium is based on an augmented abstract syntax tree. A class hierarchy of nodes is exposed to the developer and different operators allowing them to manipulate the trees are available.

New code is created using plain source using stream operators.

```
#pragma hlt unroll factor(24)
for (i = 0; i < 100; i++) {
  a[i] = i + 1;
}
```

Listing 2.18: *Loop unrolling* in Mercurium

```
#pragma hlt collapse
for (i = 0; i < 100; i++)
  for (j = 0; j < 200; j++)
    for (k = 0; k < 300; k++)
      a[i][j][k] = i * j * k;
```

Listing 2.19: *Loop collapse* in Mercurium

2.2.6.3 Querying and AST Traversal

Although possible, it is not necessary to work direclty with `TL::AST_t`, as several wrappers (named `LangConstruct`) are available.

AST Traversal can be achieved through the usage of predicate classes, which assert that some boolean properties on the tree nodes are matched. Pre-created predicates are available to walk only specific node (for example, FunctionDefinitio::predicate).

2.2.6.4 Unparsing

Mercurium is capable of unparsing AST nodes using an overloaded stream operator. Redirecting any pointer to an IR subtree recreates its original C (or C++) code.

2.2.7 Final Remarks

Table 2.2 shows an overview of the different characteristics of the compiler frameworks that we have studied in this Section.

LLVM, Open64 and GCC are powerful compilers. The internet is full of information about them and new information is being added regularly. When we started our research back in 2008, we found that finding information on these compilers was far more difficult than it is today. However, despite all of the available information, writing a compiler pass for any of them still remains a challenge due to the enormity of their codebases.

In these three cases, transformations have to be implemented in the Internal Representation, which means writing transformations at low level. This is great for implementing classic compiler optimisations, but it may give rise to explaining loop level transformations (like loop tiling) to students. In the case of Open64 or GCC, working with the front end is not an easy task. Programmers are expected to work after the front end has parsed the code into the IR, so adding new features to a language requires in-depth knowledge of the particular front end. Depending on how the new features modify the original language the generation of WHIRL (Open64) or GIMPLE (GCC) have to be modified as well.

Table 2.2: Final comparison of compilers

Feature	**Open64**	**GCC**	**ROSE**	**Cetus**	**Mercurium**	**LLVM**
Flexible Parser	No	No	No	Some	Some	No
Portability	Some	Some	Some	Some	Some	Some
Step-by-step	No	No	Some	Some	No	No
StS Transform.	No	No	Yes	Yes	Some	No
Recover orig. file	No	No	Yes	Yes	Some	No
Documentation	Some	Some	Yes	Some	No	Yes

ROSE is a great tool for developers wanting to write code analysis tools, or even simple source-to-source translation. The High-Level intermediate representation facilitates code motion and it is possible to print the status of the tree, verify its correctness and insert or remove children to any node without too much effort. However, the front end is a black-box which the user has little or no access to (none if using Fortran).

The Cetus project was in its early stages when we first carried out this survey, and it was not very stable. However, their ideas about a flexible and easy accessible IR were interesting, and our design was inspired by this work.

Mercurium was not completely public and accessible four years ago. Even today, not enough documentation is available, and playing around with the front end is not an easy task.

As none of the available tools completely satisfied our needs, we decided to write our own compiler translator. We did not aim to produce a high-quality commercial compiler, but an easy-going research tool capable of performing code transformations with little development and without excessive bootstrapping time (i.e. time from not knowing anything from the compiler architecture to being able to do useful work).

The pycparser project [22] featured a nearly complete C parser written in Python. The availability and maturity of this project greatly influenced our decision to tackle the laborious task of implementing an entire C front end.

2.3 Runtime Support

Despite different breakthroughs over recent years, there are several issues that still have to be resolved, many of which are more easily resolved when the code is executed. A classical example is dynamic memoy management, which is delegated to runtime execution.

Runtime systems provide higher-level software layers with convenient abstractions. This permits the design of portable algorithms without having to deal with low-level concerns, providing support for both data management and scheduling together.

2.3.1 StarPU

StarPU [16] is a software tool aiming to allow programmers to exploit the computing power of the available CPUs and GPUs, while relieving them of the need to specially adapt their programs to the target machine and processing units.

The runtime is responsible for scheduling tasks on heterogeneous CPU/GPU machines. A set of language C extensions is available to leverage the development effort.

StarPU implements a task-based programming model. Applications submit computational tasks, with CPU and/or GPU implementations, and StarPU schedules these tasks and associated data transfers on available CPUs and GPUs. The data that a task manipulates is automatically transferred among accelerators and the main memory, so that programmers are freed from the scheduling issues and technical details associated with these transfers.

Although most of the effort devoted to designing the runtime has been invested in the task-scheduling algorithms, it is possible for scheduling experts to implement their own custom scheduling policies through an extensible API.

2.3.1.1 Codelet and Tasks

One of the StarPU's primary data structures is the codelet. A codelet describes a computational kernel that may be implemented on multiple architectures such as a CPU, a CUDA device or a Cell vectorial unit.

Another important data structure is the task. Executing a StarPU task consists in applying a codelet on a data set, on one of the architectures on which the codelet is implemented. A task thus describes the codelet that it uses, but also which data is accessed, and how the data is accessed during the computation (read and/or write). StarPU tasks are asynchronous: submitting a task to StarPU is a non-blocking operation. The task structure can also specify a callback function that is called once StarPU has properly executed the task. It also contains optional fields that the application may use to give hints to the scheduler (such as priority levels).

By default, task dependencies are inferred from data dependency (sequential coherence) by StarPU, although the programmer can enforce them by hand.

The effort of handwritting memory transfers between processing units is leveraged from the programmer to the runtime, this is possible because the runtime handles transfers alongside the task execution.

2.3.2 GOMP

GOMP is the implementation of OpenMP for the GNU Compiler Collection. The GNU compiler will translate the OpenMP directives into calls to the GOMP runtime (plus additional code motion to accomodate the calls). GOMP is then in charge not only of implementing the typical OpenMP API routines, such as getting the number of threads or the thread ID, but it also implements low-level operations for the high-level constructs. Figure 2.18 illustrates this process.

As an illustrative example, the single construct from OpenMP as shown in Listing 2.20 is implemented as shown in Listing 2.21.

The library itself is a wrapper around POSIX threads, with various system-specific performance tweaks.

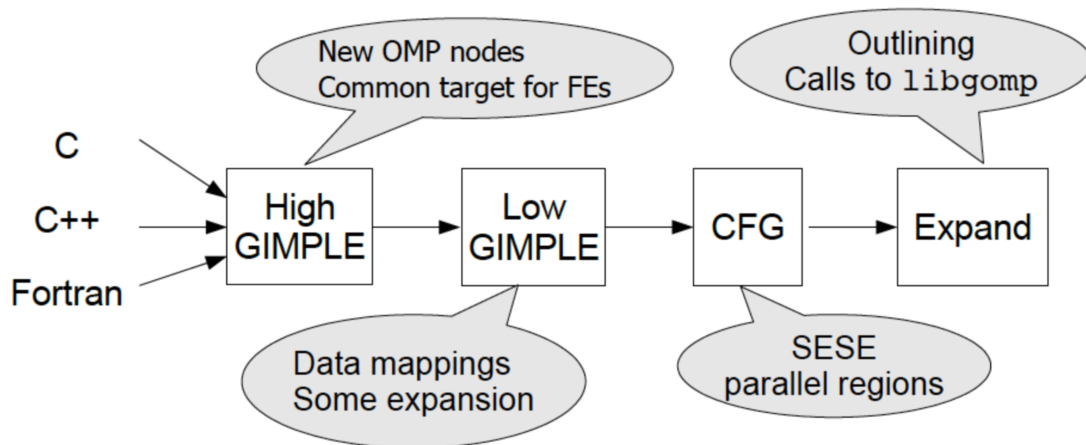


Figure 2.18: The GNU compiler toolkit translates the OpenMP nodes into calls to GOMP

```
1 #pragma omp single
2 {
3   body;
4 }
```

Listing 2.20: Example of the usage of the `single` construct of `OpenMP`

```
1 if (GOMP_single_start()) {
2   body;
3 }
4 GOMP_barrier()
```

Listing 2.21: Code generated by GCC for the `single` construct, calling the GOMP ABI

2.3.3 GMAC

GMAC [64] is a runtime that implements the Asymmetric Distributed Shared Memory (ADSM) model. It supports NVIDIA GPUs and Cell processors on GNU/Linux based systems. This model maintains a shared logical memory space for CPUs to access objects in the accelerator's physical memory, but not the contrary. When a piece of code is set to be run in the accelerator, its associate data is moved to the device. This model removes the need to explicitly make a request for memory on the separated set of memory spaces. ADSM enables also make it possible for variables used by accelerators to be passed as parameters to the corresponding system calls that read or write data for I/O devices. This enables DMA transfers directly between I/O and the accelerator, if it is supported by the hardware.

GMAC builds a shared address space between the CPUs and the accelerator. When the user allocates memory in the accelerator using the appropriate GMAC call, the runtime also allocates system memory over the same range of virtual memory address (i.e. using `mmap` system call on POSIX based systems). This creates two identical memory address ranges, one in the accelerator memory and the other in the system memory, thus a single pointer can be used to both devices. This method of overlapping the virtual memory spaces could not work on all platforms if the range of addresses returned by the accelerator API is not a valid one in the Host, or it is already used. In this case, the user has to use the GMAC API to associate them by hand, using `admSafe` to retrieve the accelerator address from the CPU pointer each time it needs to use it. Three different coherence protocols are defined from the CPU perspective, which handles the data transfers: (1) Batch-update, (2) Lazy-update and (3) Rolling-update.

In (1), whenever a kernel is invoked, all shared objects shared between the Host and the accelerator are invalidated so they have to be transferred from the CPU to the device. When the kernel finishes, all objects are transferred back from the accelerator to the Host. Method (2) attempts to detect the CPU modifications to registered address using the CPU hardware memory protection mechanism (`mprotect` system call) to trigger a page fault exception. This exception is then captured by the runtime which makes the required memory transfers to keep the coherence. Method (3) uses the same memory protection mechanism as (2) but for the purpose of detecting access into blocks of the shared data object entirely. Only the affected blocks of the shared data object are transferred.

2.3.4 Final Remarks

Our research on these different runtimes has shown us that this landscape is even more scarce than the compilers. Available runtimes tend to be huge, complicated pieces of software, focusing on thread scheduling rather than heterogeneity or programmability. We decided that it was worth implementing a runtime from scratch. This new runtime will be *based on the notion of code offloading instead of thread scheduling*. Our intention was not to take advantage of the CPU using threads, but to take advantage of *any kind of accelerator that could be connected to a traditional multi-core machine*. However, one of the basic concepts that we kept in mind when designing our runtime was the ability to easily integrate the runtime with those used in other programming models, such as SMPSs or MPI. Easy integration would enable us to combine the accelerators with existing codes, thereby enhancing its performance without the need for major code modifications.

CHAPTER 3

Yet Another Compiler Framework

`Yet Another Compiler Framework (YaCF)` has been designed to ease the burden on compiler writers. Its components are independent from each other and can be used in full source to source drivers or in small test transformations. Several subclasses, modules and packages have been included within `YaCF` to solve particular problems within StS code translations.

Components have been grouped together into three packages: FRONTEND, MIDDLEEND and BACKEND, through which the Internal Representation (IR) of `YaCF` is used. Details of each package can be found in Sections 3.4, 3.5 and 3.6. Figure 3.1 illustrates the overall StS transformation process.

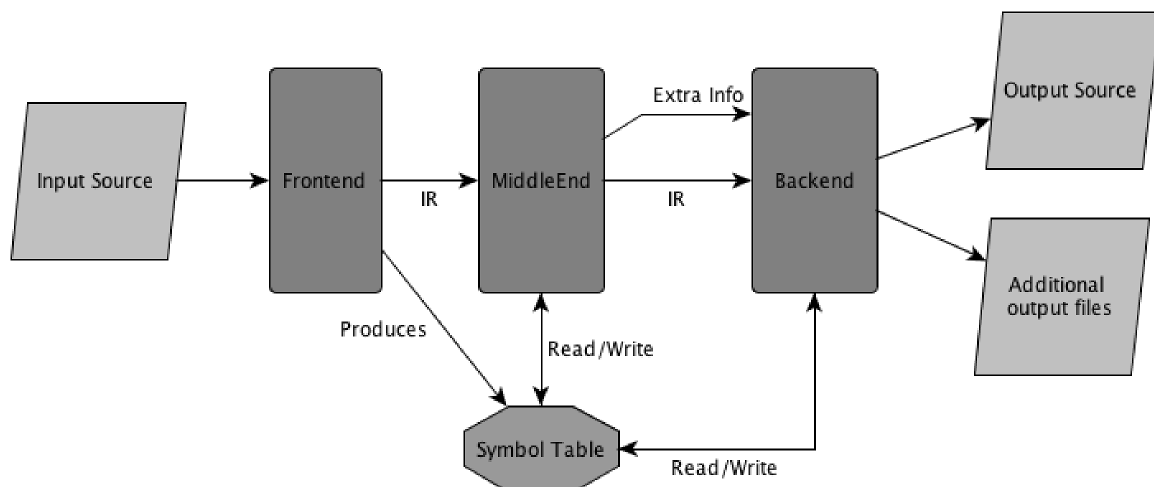


Figure 3.1: Overall translation workflow executed by a typical `YaCF` driver

As the aim of YaCF is to develop StS transformations, the IR chosen is an augmented syntax tree. Details of the IR are provided in Section 3.2. Part of the information used to augment the AST is the Symbol Table (ST), as described in Section 3.3. Outside these packages, a bin directory contains Python driver scripts to perform particular tasks, such as implementing code transformations. The majority of the work in the YaCF Frontend and the IR is derived from the pycparser project [22].

3.1 Design Considerations and Basic Concepts

The typical use case for a YaCF developer is to implement a StS transformation. A YaCF developer would normally use YaCF to implement a StS transformation. This is the most common application of the tool. Code transformations are usually implemented at IR level which means the source will have had to have been parsed first. The user would then like to recover the original file after the transformation, so once modified, the file has to be unparsed (or re-written) once again into the input language.

In YaCF a driver is the script that orchestrates a code transformation. From the driver, the parser is called, the IR is generated and it is passed to the next modules for processing. Usually drivers end up calling a Writer to unparse the final AST into the original input language. However, this is not mandatory such as when drivers are used to gather statistics from the source code.

The code translation can be split into two separate steps: (1) Searching for a particular pattern or idiom in the code and (2) Applying the desired transformation on the nodes matching the criteria. Within YaCF these two tasks are implemented in two class hierarchies: The *Filter* (1) and the *Mutator* (2).

A *Filter* is an implementation of the generic Visitor Pattern [92], which traverses the IR looking for matching nodes. The Visitor Pattern design provides a way of separating an algorithm from the object structure on which it operates. A practical result of this separation is the ability to add new operations to existing object structures without modifying those structures. A *Mutator*, or *Transformer*, is a class that contains a *Filter*; it is designed to apply the contents of a specified function to each node matching the *Filter*.

The entire YaCF framework is built using these two basic concepts. Complex transformations are composed by several nested *Filters* and *Mutators*. Usually, a *Runner* class is used to group together several transformations required to accomplish a major code transformation. *Runner* classes contain source storage facilities and code templates, and they are used to prepare the environment before (and after) calling a set of *Mutators*.

3.1.1 Filter

All *Filters* are derived from the *GenericFilterVisitor* class. This class, derived from the original *Visitor* class from the pycparser project [22] implements several top-down traversals of the IR. A *ReverseVisitor* that traverses the tree from bottom-up is also available. However, this visitor requires the parent link to be set, thus, it is not possible to traverse raw AST nodes. IR levels are discussed later in Section 3.2.

```
class ExampleFilter(GenericFilterVisitor):
    """ Returns the first node matching the example node
    """
    def __init__(self):
        def condition(node):
            if type(node) == c99_ast.Decl:
                return True
            return False
        super(ExampleFilter, self).__init__(condition_func = condition)
```

Listing 3.1: A simple implementation of a *Filter* that will iterate throught all the declarations of a given subtree

To implement a new *Filter*, it is necessary to extend the *GenericFilterVisitor* class, as shown in Listing 3.1. The constructor of this *Filter* has to call the constructor of the parent (call to *super* in the Listing). The *condition_func* determines the matching condition for the *Filter*. The parameter of this condition function is always the current node being visited, and it must return `True` if the node matches the criteria or `False` otherwise. Any other returning value is considered an error and will raise an exception. The condition function in Listing 3.1 checks that the type of the current node is a declaration. Additional information about nodes and subtree types is available in Section 3.4.

More complex *Filters* can be written taking advantage of the *Visitor* pattern. Overriding the *visit* method for a particular kind of node forces all traversals to execute the contents of the method. Listing 3.2 shows an example of this situation. Suppose we want a *Filter* to match all declarations inside a function called `foo`. We can extend the code from Listing 3.1 with an additional method *visit_FuncDef* (see Listing 3.2). The method is called each time a node of that kind is visited. If the node is the one we are looking for, we set the variable to `True`. When visiting declaration nodes, if the variable is `True` we know we are inside the desired function, thus, we mark that this is the desired node.

Notice that: (1) The *visit_FuncDef* method is called before checking the condition and that (2) as the new method overrides any of the defaults, to continue traversing down the AST we have to manually call the *generic_visit* method for each of the attributes of the FuncDef node that we want to visit.

The *GenericFilterVisitor* class implements a variety of methods:

- *apply*: Looks for the first node matching the criteria and returns.
- *iterator*: Iterates over all of the matching nodes preserving the grammatical ordering.
- *fast_iterator*: Iterates over all matching nodes in a deep first search fashion (it is faster but does not guarantee grammatical order).

```
1  class ExampleFilter(GenericFilterVisitor):
2      """ Returns the first node matching the example node
3      """
4      def __init__(self):
5          self._inside_foo = False
6          def condition(node):
7              if type(node) == c99_ast.Decl \
8                  and self._inside_foo = True:
9                   return True
10             return False
11         super(ExampleFilter, self).__init__(condition_func = condition)
12
13     def visit_FuncDef(self,node):
14         if node.name == "foo":
15             self._inside_foo = True
16         self.generic_visit(node.body)
17         self._inside_foo = False
```

Listing 3.2: A more complex example of *Filter* where only those declarations inside a particular function will be traversed

3.1.2 Mutator

Mutators are created by extending the *AbstractMutator* class. The code transformations (*aka* mutations) usually contain a *Filter* that selects which nodes will be transformed. A *mutator-Function* method has to be specified, and it must contain the code to perform the desired transformation.

Mutators modify the IR, but they must ensure its consistence (i.e. update Symbol table, parent links, and so on). Additional information about the Internal Representation is available in Section 3.2.

Note: Recursive Mutators

When implementing a *Mutator*, it is important to take into account the kind of transformation being applied. If the transformation alters the IR in such a way that might match again the condition, it will enter into an infinite loop, matching and applying the same *Mutator* repeatedly.

```python
class ExampleMutator(AbstractMutator):
    """ Apply a mutation
    """
    def filter(self, ast):
        def is_decl:
            if type(node) == c_ast.Decl:
                return True
             return False
        return DeclFilter(ast, condition_func = is_decl)
    def mutatorFunction(self, ast):
        # .... do something here with the matching node
        return ast
```

Listing 3.3: Example of a *Mutator* that will apply a transformation to all declarations within a subtree

3.2 Internal Representation

`YaCF` has been designed as a StS translation tool, thus, it does not offer functionality to generate low-level code. The IR is based on an annotated (sometimes called augmented) high-level tree-layered AST, which maintains a close relation to the original source, while facilitating the work of the developer of code transformations. Developers only need to work with tree-like structures resembling the structure of the original code, rather than focusing on low-level intermediate code. Each node of the IR denotes an element of the original language. For example, in the C front end, the `if` statement or a function definition are nodes of the tree structure. Figure 3.2 shows an example of a subtree. Figure 3.3 shows the IR structure corresponding to a more complex C code that computes an aproximation for the constant π.

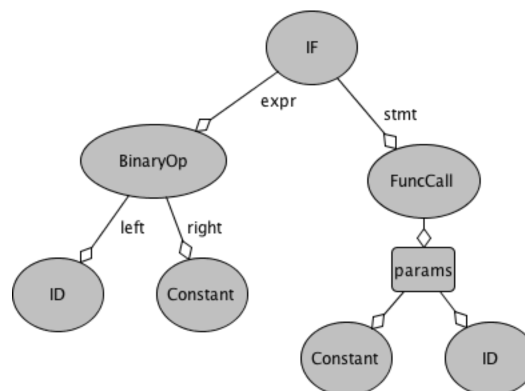


Figure 3.2: IR for the statement `if(a>1) printf("%d",a)`

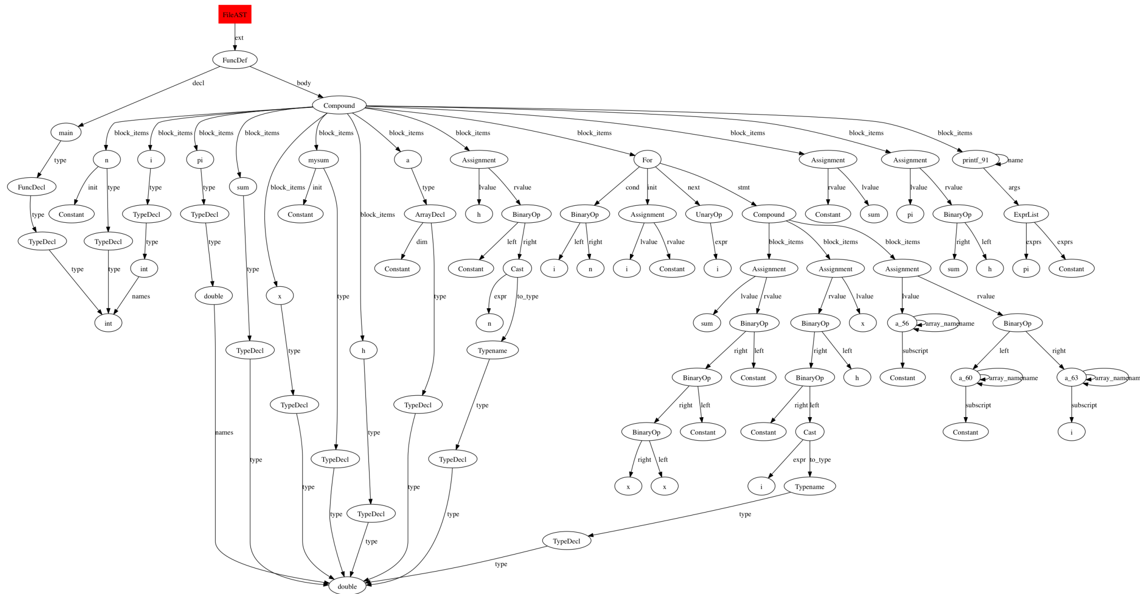


Figure 3.3: IR generated for the main loop of the π computation example shown in Figure 2.1

The IR syntax is abstract, which means that it does not represent every detail appearing in the real syntax. For instance, grouping parenthesis are implicit in the tree structure, and a syntactic construct such as an `if-condition-then expression` is denoted by a single node with two branches. Each node of the IR is a Python class. Nodes of the IR contain three different kind of attributes: simple attributes (any Python class), a child node (a reference to a son) or a list of children nodes (when an attribute of the node may contain several values). *Filters* (see Section 3.1.1) use children and list of children nodes to traverse the tree. Other elements of the Framework rely on simple attributes to work (for example, a *Mutator* can alter the `name` attribute of a node). Information might be added to each node after a translation or an analysis is run, a process referred to as annotation. Traditionally, transformations used the *setattr* and *getattr* functions to set and get information from the IR. Recent versions of `YaCF` feature a new `annotate` dictionary to store the new attributes using a more homogeneous interface.

All nodes of the IR inherit from the *IRNode* class, and contain the following base attributes and methods:

- `parent`: Provides a reference to the direct parent node, if any.
- `coord`: Coordinates of the equivalent lexeme for the node in the file (i.e. line number).
- *show()*: Prints the node in a human-readable form (but not in the original language).
- *children()*: Returns all descendant nodes from the current ones.
- *getRootNode()*: Traverses the parent links up to the top node.
- *getContainerAttribute()*: Returns the attribute of the parent node (if any) linking with the current node.

Depending on the information available on the IR, we distinguish the following incremental levels (i.e. states):

- **IR-1** or **AST**: This is the result of *parsing* a source code with any of the implemented front ends. It contains only basic parsing information. The parent attribute is not set. Neither the ST nor the Writer are available at this level. A description of the FRONTEND package which provides details on the creation of the AST for a particular language, is available in Section 3.4. Here we only describe the overall IR structure.
- **IR-2**: This is the result of processing a **IR-1** subtree with the *AstToIR* class. After the *annotation* process, a ST is available for the entire subtree, it is possible to re-print any node of the IR in the original language, and the parent link is properly set. At this level, two additional attributes (`sequence` and `depth`) are available.
- **IR-3**: The back ends might need to add further information to the nodes in the AST. To represent this fact, we will refer to the IR-3 level whenever we are describing a process in which a `YaCF` component adds information to the `IR-2`. If a transformation requires the information added by another one, we specify that using the notation `IR-3.name` where `name` is the name of the component which augments the IR. For example, a transformation requiring the information from the CUDA back end will require the `IR-3.CUDA`.

3.2.0.1 The AstToIR class

In order to transform the IR-1 into a functional IR-2, it is necessary to use the *AstToIR* transformer class. The *AstToIR* class has been implemented following the *Flyweight* pattern [63]. This forces subsequents calls to the *apply* method of this class with the same node to reuse precomputed information (particularly the Symbol Table (ST)). The root node of the tree (`FileAST`) is used to maintain a cache of the currently available IR. The *AstToIR* class is not generic to all front ends and some of them might have their own implementation, that will always be derived from the original. The *AstToIR* class performs the following tasks:

- Replaces the *str* method with a call to the appropriate writer/unparser
- Connects the parent link. Using a deep first search strategy, the parent link in each node of the IR is connected. The parent link of the root node (usually a `FileAST` node) is set to `None`.
- Creates the Symbol Table: an object of the ST class is created (see Section 3.3) and the class *SymbolTableBuilder* is used to initialize the ST and update the information of the IR.

To apply the aforementioned transformations, the user has to call the *annotate* method. Using the *getSymbolTable* method of the same class it is possible to retrieve the ST that has been created during the annotation process.

```
1 it = InsertTool(subtree = new_subtree)
2 it.apply(node1, 'attribute', position = "begin")
3 it.apply(node2, 'attribute', position = "after")
```

Listing 3.4: Inserting a subtree inside the main IR

```
1 ReplaceTool(new_node=new_subtree, old_node = old).apply(old.parent, 'attr')
```

Listing 3.5: Replace a subtree inside the main IR

3.2.1 Manipulating the IR

It is possible to manipulate the IR using standard Python tools, like `setattr` or `getattr`. However, for convenience, a set of tools is available in the TOOLS.TREE package. These tools work on the IR-2 level, but they do not require information from the ST as they only work within the tree.

Three operations to manipulate the IR are implemented: *Insert*, *Replace* and *Remove*. Figure 3.4 shows the insertion of a new subtree into the IR. Tools to manipulate the IR work on two stages: Firstly they build an instance of the tool and then they apply it to a subtree. This enables YaCF to reuse part of the process when the same operation is applied several times to the same subtree.

InsertTool performs insertions on the subtree. Its constructor receives a subtree to be inserted. The *apply* method receives the node that will be the parent of the subtree and the attribute where it is going to be inserted. If the attribute where the node will be inserted is a list of nodes, it is possible to specify a position inside the list, choosing one of the following values: `begin`, `end`, `after` and `before`. The values `after` and `before` allow users to insert the node immediately after or before an existing node on the list, which is specified by the parameter `prev`. A code example for this operation is shown in Listing 3.4.

In a similar fashion to the *InsertTool*, a *ReplaceTool* and a *RemoveTool* are also available. Figures 3.4, 3.5 and 3.6 show the effect of these operations on the IR. Listing 3.5 shows an example code to perform a replacement of a subtree within the main IR.

3.3 Symbol Table

The YaCF ST is designed to be independent of the parsing process. It can be created or updated at any stage of the StS translation. The ST is implemented as an extension of a Python dictionary. Usual ST operations, such as *lookUp* and *addSymbol*, are implemented in this class.

The creation of the ST for an IR is performed through the builder class *SymbolTableBuilder*, which is a *Visitor* that creates the IR by traversing the tree in grammatical order.

The easiest way to create a ST is to instantiate an object of the *SymbolTable* class and then invoke the *SymbolTableBuilder* to initialize it, as shown in Listing 3.6.

Each element of the ST is stored as a `Symbol` object, and holds the following information:

- `name`: Name of the Symbol.

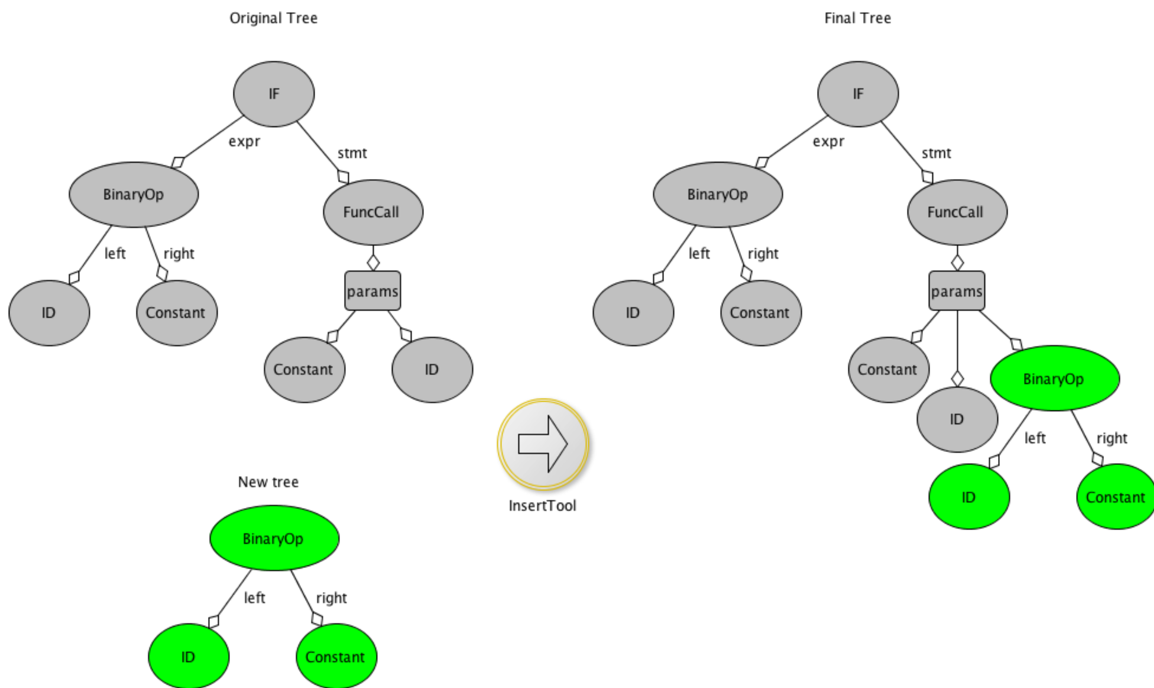


Figure 3.4: Insertion of a new parameter in a function call

```
from Frontend.SymbolTable import SymbolTable, SymbolTableBuilder
st = SymbolTable()
tsv = SymbolTableBuilder(symbol_table = st)
tsv.visit(some_ast)
```

Listing 3.6: Initialization of a *SymbolTable*

- `node`: Reference to the original node in the AST.
- `type`: Type of the node, reference in the IR.
- `scope`: Scope information (see Section 3.3.1).
- `btype`: Basic type of the ID (for example, in C, if the identifier is an integer).
- `sizeExpression`: Expression to determine the size of the variable.
- `extra`: A dictionary holding optional information for different compiler stages.

`Symbol` objects can be printed to a string for debugging purposes. They can also be compared using two simple rules: (1) Two *Symbol* instances are equivalent iff they have the same name and the same scope, and (2) *Symbol* instance A is greater than *Symbol* instance B if and only if `scope`(A) is greater than `scope`(B).

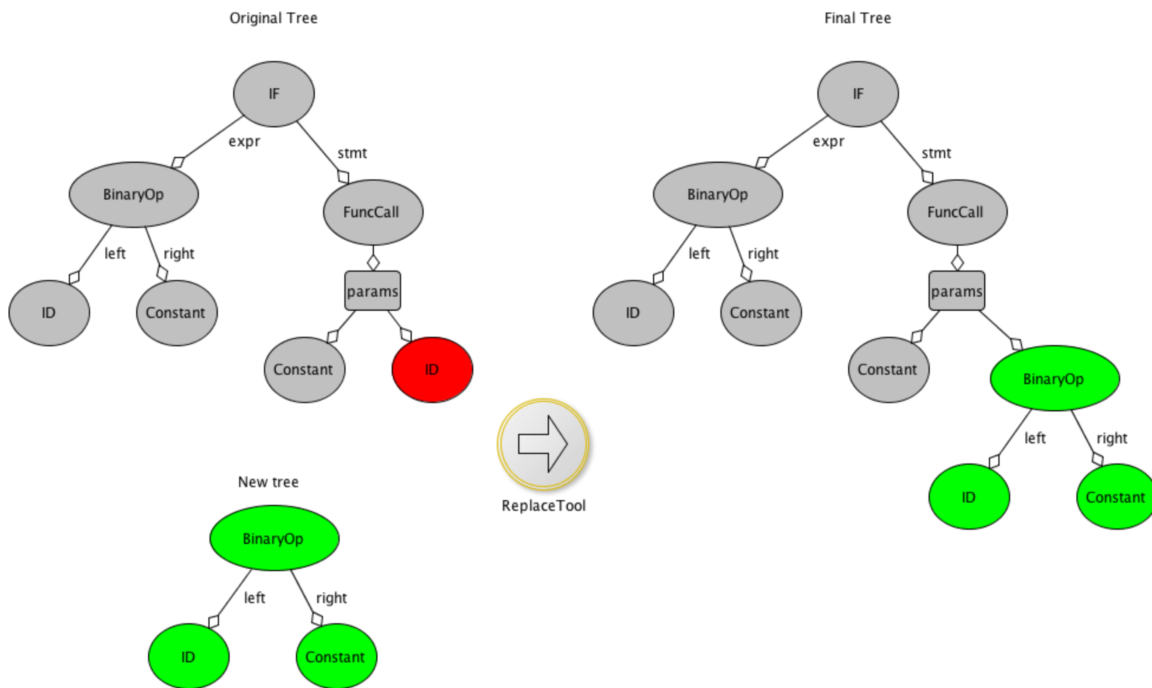


Figure 3.5: Replace operation of a parameter in a function call

3.3.1 Scope Information

As the ST can be created at any stage of the translation process, and it might even be created for a subtree disconnected from the main IR of the code; the coordinate of the lexeme relative to the IRNode is not enough to locate an identifier in the ST. To replace the line number information, each node of the AST is decorated with two additional attributes: the sequence number and the depth. The sequence number is assigned sequentially to each node of the IR in grammatical order, starting from zero. The depth value indicates the number of grammatical nested scopes preceding the declaration. For this reason, each symbol has a `Scope` attribute, which describes the proper scope of the declaration. Algorithm 3.1 is used to insert an element in the *SymbolTable*, whereas the Algorithm 3.2 is used as the lookup for symbols.

Listing 3.8 shows a C code with several name conflicts. Identifier `i` is declared several times (lines 3, 6 and 12). Printing the ST corresponding to the code in Listing 3.8 produces the output shown in Listing 3.7.

Notice how each declaration of `i` has a separate entry in the ST (lines 20, 22 and 27 in Listing 3.8). Level 0 of the ST contains general language declarations and implicit types, like GNU built-in types, or basic language types.

Algorithm 3.1 Insertion of a symbol in the Symbol Table

function ADDSYMBOL($decl, depth$)
  $sizeExpression \leftarrow buildSizeExpression(decl)$
  $nsymbol \leftarrow Symbol(decl, depth, sizeExpression)$
  for all symbol in $st[depth]$ **do**
    if $nsymbol = symbol$ **then return**
    end if
  end for
  $st[depth].insort(nsymbol)$
end function

Algorithm 3.2 Looks for the declaration of an identifier in the Symbol Table

function LOOKUP(id)
  $depth_{act} \leftarrow min(id.depth, len(st))$
  while $depth_{act} \geq 0$ **do**
    for all $symbolinst[depth_{act}]$ **do**
      if $id.sequence \in symbol.scope$ **then**
        if $symbol.name = id.name$ **then return** $symbol$
        end if
      end if
    end for
  end whilereturn Identifier Not Found
end function

```
 Symbol table
==========================
 Level : 0

[{float: float, (0, None)}
{int: int, (0, None)}
{char: char, (0, None)}
{double: double, (0, None)}
{FILE: FILE, (0, None)}
{bool: bool, (0, None)}

 Level : 1

[{foo: char, (1, 73)}
{func: int, (32, 73)}
{main: int, (67, 73)}]

 Level : 2

[{i: int, (7, 33)}
{j: int, (11, 33)}
{i: int, (39, 68)}]

 Level : 3

[{foo: char, (16, 32)}
{i: int, (21, 32)}]
```

Listing 3.7: Content of the ST after analyzing the code in Listing 3.8

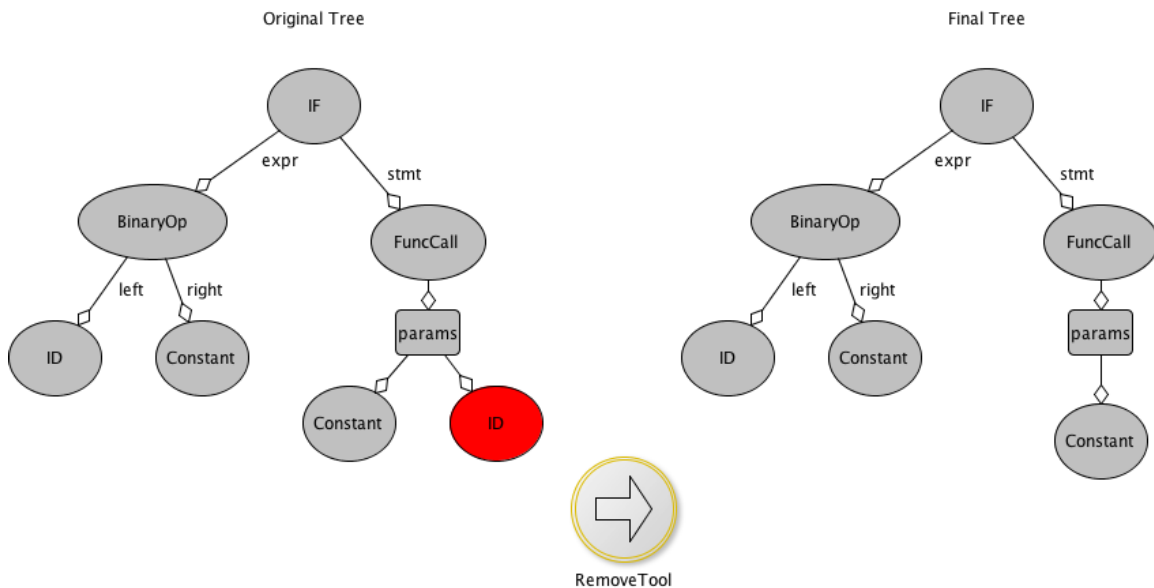


Figure 3.6: Remove operation of a parameter in a function call

These declarations are introduced by the builder before visiting the tree. The declarations of the code start in level 1, with the declaration of *foo* as a char. The numbers enclosed in parenthesis represent the start and end of the scope of the corresponding declaration. In the case of *foo*, the declaration is valid between sequence numbers 1 (beginning of the file) and 73 (end of the file). Notice that sequence number are not related in any way with line numbers in the source code. Function declarations appear on level 1 and their scope covers the declaration itself and continues right throught to the end of the file. Basic type (`btype`) of a function declaration represents its type.

Scope depth information suffices to differentiate the declarations in lines 2 and 5 in Listing 3.8, however, to distinguish the declarations in lines 2 and 12, it is necessary to take into account their sequence numbers.

3.3.2 Computing the Size of Elements

In some situations it is necessary to have access to the size of a particular element in the ST. This information is usually machine dependant (integers or doubles do not always have the same size), thus, it is not possible to have that information to hand when transforming the source. However, it is possible to extract an expression that computes the size of the element in terms of the original code, and, when compiled with the machine-dependant compiler, will generate the real size of the element.

```
char * foo;
int func () {
  int i,j;
  {
    char * foo;
    int i;
    printf("%d",i);
  }
}

int main() {
  int i;
  foo = NULL;
  func()
  printf("%d",i);
  if (foo) printf("%s", foo);
}
```

Listing 3.8: C code example with nested declaration scopes

When building the ST using the *SymbolTableBuilder*, information about the number of elements and the basic type of the declarations is stored in attributes. This information is used to create an expression that, when evaluated, will compute the total size of the declaration in memory. Notice that it is not possible to know information about pointers at translation time, but it is possible to compute that information at runtime. For example, if an structure contains a `void` pointer, the computed expression will not reflect the real size of the structure. However, the computed expression is sufficient to estimate the size of structures or arrays.

3.4 The Frontend

The FRONTEND package is based on the `pycparser` Python module. `pycparser` [22] is a C lexical and syntax analyzer completely written in Python. We have derived most of the structure of the FRONTEND from `pycparser`.

The `YaCF` FRONTEND package contains several components:

- C99, OpenMP, OpenACC and GNU syntax analyzers.
- Abstract Syntax Tree (AST).
- Symbol Table.
- Internal Representation: an extension of the AST with additional information. This information is used in source code transformations.

Figure 3.7 shows the `YaCF` class hierarchy. Class *PLYParser* is the base class of the `YaCF` FRONTEND.

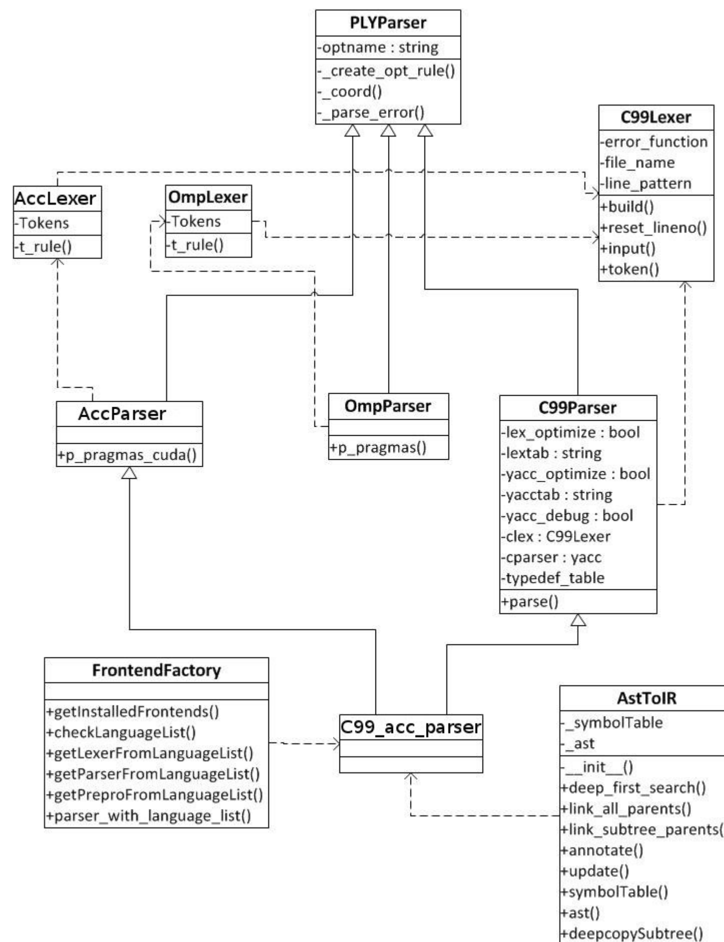


Figure 3.7: FRONTEND package class hierarchy

PLYParser controls some syntax errors and holds general information about parsers. Each parser in YaCF must inherit from *PLYParser*. Lexical analyzers for differente languages inherit from *C99Lexer*. The design allows the developer to extend each lexer with specific rules. In a similar fashion, all parsers inherit from class *C99Parser*.

Parsers are implemented following a Factory Pattern, enabling the developer to combine different extensions of the C language into one new parser. The bottom section of Figure 3.7 shows the *AstToIR* class of the YaCF parser created by the *FrontendFactory* class combining C99 and OpenACC parsers.

```
ArrayDecl: [type*, dim*]
ArrayRef: [name*, subscript*]
Assignment: [op, lvalue*, rvalue*]
BinaryOp: [op, left*, right*]
Break: []
Case: [expr*, stmt*]
Cast: [to_type*, expr*]
Compound: [block_items**]
```

Listing 3.9: Part of the C99 AST configuration file

```
def p_clause_2(self, p):
   """ clause : REDUCTION LPAREN reduction_operator COLON identifier_list RPAREN """
   p[0] = [omp_ast.OmpClause(type = p[3] , name = 'REDUCTION', identifiers = p[5], coord =
       self._coord(p.lineno(1)))]

def p_clause_3(self, p):
   """ clause : NOWAIT """
   p[0] = [omp_ast.OmpClause(type = str(p[1]) , name = str(p[1]).upper(), identifiers =
       None, coord = self._coord(p.lineno(1)))]
```

Listing 3.10: Extract from the OpenMP parser of `YaCF` showing the interpretation of the `reduction` and `nowait` clauses

3.4.1 Defining a New Language

Languages are defined in the FRONTEND package by creating a Python module containing a file named `languagename_ast.cfg` which lists the nodes of the language. All the AST nodes in `YaCF` are configured in this file enabling the compiler to create the AST with the correct information. Listing 3.9 shows an example of this file. In line 1, an array declaration node (`ArrayDecl`) is specified. The list of attributes is specified as a list in the same line. For example, an array declaration node has two attributes: the `type` and the `dimension` of the array. This information will be filled by the right parser using syntax-directed translation. The attributes with an asterisk indicate to the compiler that the node has a child node, while two asterisk indicate a sequence of child nodes. The `Compound` node in line 8 represents a list of blocks in the source code.

From the configuration file, `YaCF` generates a set of AST Python classes. Each class represents a node in the AST. With this information the parser can fill in all the information. Listing 3.10 shows part of the OpenMP parser. Each grammar production features a method and each method creates the appropriate AST node. For example, the OpenMP `reduction` clause is parsed in line 2 while an AST node with the reduction type and the identifiers is created in line 3.

```
S1 a = b + c
S2 if (a > 10) goto L1
S3   d = b * e
S4   e = d + 1
S5 L1: d = e / 2
```

Listing 3.11: Control and data dependency example extracted from [98]

```
  a[3] = a[5] * a[i];
  x = h * ((double) i - 0.5);
  sum += 4.0 / (1.0 + x * x);
```

Listing 3.12: Example of a SESE block statement with variable dependencies

3.5 The MiddleEnd

The input of the package MiddleEnd is an IR and the output is another IR with the same, or higher, level. The packages in MiddleEnd are commonly used as for intermediate processing to optimise or prepare codes so the Backend packages can proceed. Some packages used for analysis of codes can also be found in the MiddleEnd.

3.5.1 Data Dependency Analysis

Some situations might require an analysis of the data dependency in the code. For these situations, YaCF features a (basic) Data Dependency Analysis tool. Dependency analysis produces execution-order constraints between statements.

Listing 3.11 shows a block statement with a typical expression statement. If statement S_1 precedes S_2 in their given execution order, we write $S_1 \lhd S_2$ (Notation from [98]). A dependence between two statements in a program is a relation that constrains their execution order. A control dependence is a constraint that arises from the control flow of the program, such as S_2 with S_3 and S_4 in Listing 3.11. These dependences are written as $S_1 \delta^c S_2$. A data dependence is a constraint that arises from the flow of data between statements, such as S_3 and S_4 in Listing 3.11. If we reorder these statements, the result could be incorrect.

Data dependencies can be classified into four types:

1. If $S_1 \lhd S_2$ and the former sets a value that the latter uses, we call this a flow (or true) dependence, and it is written as $S_1 \delta^f S_2$.

2. If $S_1 \lhd S_2$, S_1 uses a particular variable's value and S_2 sets it, then we have an antidependence (written $S_1 \delta^a S_2$).

3. If $S_1 \lhd S_2$, and both of them set the same variable variable then we have an output dependence (written $S_1 \delta^o S_2$).

4. If $S_1 \lhd S_2$, and both of them read a variable then we have an input dependence (written $S_1 \delta^i S_2$).

It is possible to extend the definition of data dependencies from the statements to the variables themselves: let V_1 and V_2 be variables declared in a program; and let there be S an statment of that program containing an expression involving both V_1 and V_2. Expressions can either read variables or write them. If statement S modififes or updates the value of variable V_2 by solving an expression in which V_1 is involved, we can state that V_2 depends on V_1 ($V_1 \delta^f V_2$). Variable dependency, like statement dependency, is transitive, i.e. if V_2 depends on V_1 and V_3 depends on V_2, then V_3 depends also on V_1:

$V_1 \delta^f V_2 \wedge V_2 \delta^f V_3 \Longrightarrow V_1 \delta^f V_3$

A Data Dependency Analysis module is available within YaCF (DataAnalysis.LlcScope). It is capable of generating a dependency graph of the variables used in a SESE block of statements.

The class *DGraph* implements the dependency graph using a dictionary containing instances of *Dnode*. *Dnode* associates a node of the ST with a list of predecessors and a list of successors. A predecessor of a variable is any variable to which the current variable has a dependency, i.e. if the current variable is V_2, it predecessor list will contain all V_1 satisfying $V_1 \delta V_2$. A successor of a variable is any variable which the current variable creates a dependency, i.e. if the current variable is V_1, the list of sucessors contains all V_2 satisfying $V_1 \delta V_2$.

The *DGraphBuilder* populates a *DGraph* data structure. Each occurrence of an ID creates an instance of *DNode* and adds it to the current *DGraph* instance if it has not already been inserted. When the *DGraphBuilder* visit an assignment expression, the predecessor and successor lists of the nodes involved in the assignment are updated. There is no interprocedural analysis support currently available in YaCF . This forces the analysis to assume that all variables passed through a function call are read and written. It also creates dependencies among them. The algorithm used to populate each *DGraph* instance is shown in Algorithm 3.3.

Algorithm 3.3 Analysis of an Assignment node

function visit_assignment($node, visited_nodes$)
  $rvalue \leftarrow visit(node.rvalue)$
  $lvalue \leftarrow visit(node.lvalue)$
  for all r in $rvalue$ **do**
    for all l in $lvalue$ **do**
      $succ(l) \leftarrow succ(l) + l$
      $pred(r) \leftarrow pred(r) + s$
    end for
  end for
end function

3.5.1.1 Data Dependency Graph

It is possible to print the dependency graph of the variables inside a block statement whenever it is required by using the *DataDependencyTool*.

Each node of the graph represents a variable used in the block statement. A dependency between V_1 and V_2 is represented by an arrow from the node of V_1 to the node of V_2. If a cycle appears in the graph (i.e. $V_1 \delta^f V_2$ and $V_2 \delta^f V_1$) it means that the variable is being reused, for example, in a reduction.

Figure 3.8 shows the variable dependency graph generated for the code in Listing 3.12. Variable `sum` has a cycle over itself in the graph representing the reduction operation. The value of `sum` is computed based on the values of the X variable, which in turn is computed based on values from `h` and the loop variable `i`. A blue color on a node indicates that the variable is read-only, whereas the red color implies that the variable is both read and read-only. A green color (not shown in Figure 3.8) is used when the variable is only written.

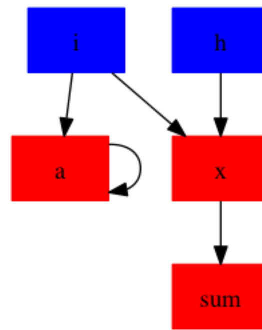


Figure 3.8: Variable dependency graph for the code in Listing 3.12

3.5.1.2 Checking dependencies

Instances of class *DGraph* contain a method *checkDependency*. This method accepts two instances of class *Dnode* as parameters and checks if the first one has a dependency with the second one (i.e. checks if the latter is in the predecessor set of the former or in any of the predecessor sets of any of its predecessors). Transitive dependencies are checked traversing the successor and predecessor lists.

3.5.2 Loop Analysis

A major part of the optimization effort in any compiler is usually devoted to loops. `YaCF` provides developers with a tool to perform loop analysis (*ParametrizeLoopTool*).

Applying *ParametrizeLoopTool* to a loop node of the IR (i.e. a `For`) node will create a dictionary containing information about the loop. Some restrictions apply. Current implementation limits the parameters captured from a given loop to canonical loop or canonical loop nests. The most relevant parameters extracted by *ParametrizeLoopTool* are:

- `loop_variable`: Variable to iterate.
- `iteration_expression`: Expression to define the next loop step.
- `init_value`: Starting value for the iteration variable.

```c
for (int i = 0; i < N; i++)
  a[i] = b[i];
```

Listing 3.13: A canonical C loop

- `first_iteration`: Value for the first iteration (expression).
- `last_iteration`: Value for the last iteration (expression).
- `loop_stride`: Distance between two iterations (expression).
- `number_of_iterations`: Total number of iterations (expression).

This information is stored in IR format (i.e. subtree expressions). For example, Table 3.1 details the information extracted by the *ParametrizeLoopTool* from the loop in Listing 3.13.

Parameter	**Value**
Loop variable	i
Stride	1
Condition Node	$i \leq N$
Last iteration (It_{last})	$N - 1$
Number of iterations	$(It_{last} - It_{first})/1$
Iterator	$i+ = 1$

Table 3.1: Information extracted from the loop in Listing 3.13

Notice that some of the parameters extracted are not constant values, but expressions written in the IR language. These expressions will then be rewritten in the original language by a Writer. The final value will be computed at execution time.

3.5.3 Loop Optimizations

Due to the nature of StS compilers, loop optimizations play a dominant role in code optimization. `YaCF` implements several loop optimizations as interchange, unswitch, unroll, or tiling. These transformations can be reffered to by different names in the literature, and many of them are also implemented in other StS compilers such as Cetus (Section 2.2.5) or Mercurium (Section 2.2.6).

The optimizations available in `YaCF` have been implemented following the *Mutator* and *Visitor* software patterns [63] in the MIDDLEEND.LOOP.MUTATOR directory. This directory contains the key *Mutators* responsible for carrying out processing on the intermediate code IR-2 (Section 3.2). For example, the module LOOPTILING.PY contains the *LoopTilingMutator* which is responsible for implementing a rectangular loop tiling on the AST type supplied, i.e. a tiling with constant tile sizes (see Section 3.5.3.4 for details).

These *Mutator* make extensive use of tools such as *ParametrizeLoopTool* for handling loops, or those available in the TOOLS.TREE package such as *ReplaceTool*.

```
1 double a[N][M], b[N];
2
3 for (int i = 0; i < N; i++)
4   for (int j = 0; i < M; j++)
5     a[i][j] = a[i][j] * b[j];
```

Listing 3.14: Loop nest before aplying *loop interchange*

```
1 double a[N][M], b[N];
2
3 for (int j = 0; i < M; j++)
4   for (int i = 0; i < N; i++)
5     a[i][j] = a[i][j] * b[j];
```

Listing 3.15: The result of applying *loop interchange* to the loop nest in Listing 3.14

In YaCF the programmer is responsible for verifying the safety of certain optimizations which are not always applicable. That is, there is no guarantee that some transformations such as loop tiling, retain the original semantics of the source code after being applied. Ensuring program correctness is the responsability of the YaCF user.

3.5.3.1 Loop Common

The package Common.py in the MiddleEnd.Loop directory contains a number of *Mutators* and *Filters* common to many optimizations and drivers, the *LoopFilter* is one of them. This *Filter* searches for a *For* node in the entire AST.

The default behavior of *LoopFilter* is to iterate over the AST to return all the encountered *For* nodes. The *Filter* can be parametrized with an *identifier* parameter to discriminate on the loop index variable, specifying the loop or loops it will search for.

3.5.3.2 Loop Interchange

This transformation is the process of exchanging the order of two perfectly nested loops. One major purpose of *loop interchange* is to improve the cache performance for accessing array elements. It is not always safe to exchange the iteration variables due to dependencies between statements for the order in which they must execute. To determine whether a compiler can safely interchange loops a dependence analysis must first be carried out.

In the basic example shown in Listings 3.14 and 3.15 the effect of the transformation can be observed.

In YaCF, this effect is the default behaviour of the *LoopInterchangeMutator* when providing it a subtree of the AST containing the outermost loop. Nevertheless, *LoopInterchangeMutator* enables more useful applications such as swapping two nested loops when there are other loops between in a perfect nesting. This feature is useful, particularly when the number of loop nests is greater than two, for implementing some versions or *loop tiling* as we will see in Section 3.5.3.4.

```c
double a[N];

for (int i = 0; i < N; i++)
  a[i] = 0;
```

Listing 3.16: An example of a simple C canonical loop *zeroing* an array

```c
double a[N];

for (int _i_ = 0; i < N; i += B)
  for (int i = _i_; i < min(_i_ + B, N); i++)
    a[i] = 0;
```

Listing 3.17: Loop in Listing 3.16 after aplying strip-mining with strip size B

3.5.3.3 Strip-mining

Strip-mining, also known as *loop sectioning*, is a loop-transformation technique for enabling SIMD-encodings of loops, as well as providing a means of improving memory performance. By fragmenting a large loop into smaller segments or strips, this technique transforms the loop structure in two ways:

- It increases the temporal and spatial locality in the data cache if the data are reusable in different passes of an algorithm.
- It reduces the number of iterations of the loop by a factor of the length of each "strip", or number of operations being performed per SIMD operation.

Strip-mining is equivalent to a rectangular tiling (see Section 3.5.3.4) being applied to simple loops. The transformation has been successfully applied to code optimization in vectorial computers [80, 12, 13].

Listing 3.17 shows the effect of applying the *LoopStripMiningMutator* to the loop shown in Listing 3.16.

In Listing 3.17 we observe (line 3) that a new loop has been introduced enclosing the original loop. The new loop runs over the original iteration space in blocks of size B. Its index variable sets the start and end of the new inner loop (line 4), which now iterates over each block. As a result of applying *strip-mining*, the iterations will execute in consecutive blocks of size B indexed by `_i_`. The limit `min(_i_ + B, N)` ensures that the new code does not run extra iterations.

As all dependencies of a program are lexicographically positive, *strip-mining* is always safe to apply [147].

```
1 double a[N][N], b[N][N], c[N][N];
2
3 for (int i = 1; i < N; i++)
4   for (int j = 1; j < N; j++)
5     for (int k = 1; k < N; k++)
6       c[i][j] = c[i][j] + a[i][k] * b[k][j];
```

Listing 3.18: Square matrices product, example of a perfect loop nest

```
1 double a[N][N], b[N][N], c[N][N];
2
3 for (int _i_ = 1; _i_ < N; _i_+= B)
4 for (int i = _i_; i < min(N, _i_ + B); i++)
5   for (int _j_ = 1; _j_ < N; _j_+= B)
6   for (int j = _j_; j < min(N, _j_ + B); j++)
7     for (int _k_ = 1; _k_ < N; _k_+= B)
8     for (int k = _k_; k < min(N, _k_ + B); k++)
9       c[i][j] = c[i][j] + a[i][k] * b[k][j];
```

Listing 3.19: Square matrices product with its loops incorrectly ordered after applying strip-mining

3.5.3.4 Loop Tiling

Loop tiling, also known as *loop blocking* was promoted by Francois Irigoin and Michael Wolfe at the end of the 80s [144, 145]. It is one of the most important iteration reordering loop optimizations. From *loop tiling* it is possible to extract beneficial properties for both parallel machines and for multiple level cache monoprocessors exposing space locality [147]. *Loop tiling* includes those loop otimizations which reorder its iteration space. Some examples of these are *loop interchange*, *loop skewing*, or *strip-mining* among others. All these transformations change the order in which iterations are executed, while preserving the order of the statements into each iteration.

YaCF implements square or rectangular tiling [147]. It is named rectangular from the shape of the blocks that run the iteration space when *loop tiling* is applied to a two-dimensional space (two nested loops). Formally, the name is kept for higher dimensions as the geometric properties of the rectangles are also preserved.

In the following paragraphs we briefly describe the algorithm that YaCF features for automatic rectangular tiling (for further reading, see also [131]).

Listing 3.18 shows a series of three perfectly nested loops. It corresponds to a square matrices multiplication code, and provides a good example of how to obtain the same benefits as those mentioned with *strip-mining* in Section 3.5.3.3, but now applied to nested loops.

The first optimization used by loop tiling is *strip-mining*. By applying *strip-mining* to each of the loops in Listing 3.18 (lines 3, 4 and 5) three new loops are created as shown in Listing 3.19 (lines 3, 5 and 7) and the bounds of the original loops (lines 4, 6, and 8) are changed as described in Section 3.5.3.3 .

```
double a[N][N], b[N][N], c[N][N];

for (int _i_ = 1; _i_ < N; _i_+= B)
for (int _j_ = 1; _j_ < N; _j_+= B)
for (int _k_ = 1; _k_ < N; _k_+= B)
  for (int i = _i_; i < min(N, _i_+B); i++)
    for (int j = _j_; j < min(N, _j_+B); j++)
      for (int k = _k_; k < min(N, _k_+B); k++)
        c[i][j] = c[i][j] + a[i][k] * b[k][j];
```

Listing 3.20: The effect of applying square *loop tiling* with tile size B to the code in Listing 3.18

So, *strip-mining* itself can produce a program for the matrix product using 6 loops. Nevertheless, the order in these loops is not correct for our purposes, as it does not correspond to a matrix product algorithm with a blocked iteration space. *Strip-mining* itself does not split a nested loop iteration space into strips.

The second transformation we are going to apply is *loop interchange*. To obtain a correct code, we need to move the new loops generated by `strip-mining` outwards, and move the original loops inwards.

This is achieved with *loop interchange* and the resulting code is shown in Listing 3.20.

Note that it is safe to apply *loop interchange* for the matrix product code, but it is not for many other useful programs including some partial differential equation methods.

Algorithm 3.4 Automatic *loop tiling* in `YaCF`

function $\text{LoopTilingMutator}(ast, indexes, sizes)$
  for $i = 0$ to $length(indexes) - 2$ **do**
    $loop \leftarrow LoopFilter(ast, indexes[i])$
    $LoopStripMiningMutator(loop, sizes[i])$
    $LoopInterchangeMutator(loop, indexes[i], indexes[i+1])$
  end for
  $loop \leftarrow LoopFilter(ast, indexes[-1])$
  $LoopStripMiningMutator(loop, sizes[-1])$
  for $i = length(indexes) - 2$ to 0 **do**
    $LoopInterchangeMutator(ast, indexes[i+1], indexes[i])$
  end for
end function

Algorithm 3.4 describes the implementation of tiling by `YaCF`. The algorithm receives the AST subtree corresponding to the outermost loop, an index list of the loops to be processed, and the block sizes to apply when processing each loop in the former list. It is worth noticing that it is common for sizes to be different to each loop.

```
1 ...
2 for (int i = 0; i < N ; i++)
3     if (b[i] != 0)
4        a[i] = b[i] * c[i]
5     else
6        break;
7 ...
```

Listing 3.21: An example of a loop body with a `break` statement

The reader may observe that it is possible to obtain a similar code to that shown in Listing 3.20 by repeatedly applying *strip-mining* and *loop-interchange* over each nested loop in the code shown in Listing 3.18. Repeatedly applying *strip-mining* and *loop-interchange* corresponds to the first `for` loop and the last *LoopStripMiningMutator* in Algorithm 3.4.

Now, the only thing that remains to be done is reorder the series of inner loops to restore their original order. To obtain the final code (Listing 3.20) it is enough to apply several *LoopInterchangeMutator* transformations to the loop list inversely. That process is performed by the second loop in Algorithm 3.4.

3.5.4 The Outliner

To facilitate certain transformations or specific analysis phases, it is easier to extract a piece of code from a code block to an external function, a process which is called Outlining. Although this might be seen as a trivial task, it is critical to take into account several side effects that might occur while applying this code motion technique. For example, in C, any variable used inside a block whose value is modified has to be passed by reference. This ensures that the value modified in the function is the original one and not a copy residing in the stack.

It is also important to take into account potential breaks in the code. For example, suppose that we want to extract the body of the loop in Listing 3.21 to an external function (i.e. each iteration of the loop will call the outlined function). This will generate the code in 3.22. However, this code is not correct as the break statement is not inside a `for` loop. The compiler has to analyse the loop and provide a proper replacement to keep the original code working. In this case, we have decided to replace the `break` statements by `return` statements with a known value, and to check inside the loop body for that return value.

More complex situations can arise when labels and jump statements (i.e. `GOTO` statements) are involved. For the sake of simplicity, in the current section we will assume that the code segment that we want to outline is Single entry - Single exit block (SESE).

```
void outlined_function(int * a, int * b, int * c, int i)
{
   if (b[i] != 0)
      a[i] = b[i] * c[i];
   else
      break;
}
...
for (int i = 0; i < N ; i++)
   outlined_function(a,b,c,i);
```

Listing 3.22: Incorrect extraction of the loop body. The `break` statement is no longer syntactically correct.

The class *OutlinerMutator* from the package MIDDLEEND.FUNCTIONS implements a function outliner. This *Mutator* creates a new function, called the outlined function. The body of the outlined function will be a copy the block statement that we want to outline. It also replaces the original (passed by reference) block in the IR by a function call to the previously created outlined function. The *Mutator* returns a reference to this new call inside the original tree, and the method *get_outlined_function* can be used to retrieve the outlined function. The *Mutator* has been designed so the outlined function can be written onto an external file, and compiled independently. The outlined function can be injected on the original AST, or not, depending on the instance parameters.

It is also possible to retrieve only the function definition, or the type declarations required to define the variables used inside the function body. Constructor-time parameters enable developers to instruct the *Mutator* to not replace the original code and just extract the outlined function, or to change the name of the outlined function.

To avoid replacing all occurrences of the variables in the outlined function by references to the parameters, the *Mutator* takes advantage of the scope nesting to create new local replacements of the parameter variables. However, this changes the default C pass-by-reference parameter passing scheme to a copy-and-restore, so this transformation can produce side-effects in some situations, particularly when using threads. Instead of using this transformation, it is possible to use a traditional replacement of all occurences of R/W parameters by references by changing a parameter in the constructor. The final decision is left up to the `YaCF` user.

3.6 The Backend

The BACKEND package contains transformations whose destination is a programming language. These transformations usually operate over the IR and then use a Writer to generate a final code.

All subpackages under this package follow the same structure. The subpackage is named after the destination target or platform (for example, a back end named Cuda will generate CUDA code). If the back end is meant to be run alone (it is not part of another), it has to contain a *Runner* class that specifies how to create the destination code. Also, inside this subpackage, there must be a FILTER directory containing the *Filters* used, a MUTATORS directory with the *Mutators* used, and a WRITERS directory if any particular Writer is required. A TEST directory may appear also, containing testing scripts for each back end implementation. Documentation of the back end has to be available on the `init` file of the package. If the back-end uses any template file, it must be stored under the TEMPLATE directory.

A set of common classes for *Visitors*, *Filters* and *Transformers* are stored in the COMMON sub-package. These common classes have been already described in Section 3.1. Commonly used Filters are defined in the GENERICVISTIORS module whereas common used *Mutators* are defined in the ASTSUPPORT module.

3.6.1 The Template Subsystem

When working in StS transformations, there are situations where the back-end writer has to fill a pattern written in the destination language with parameters of the current code.

Using the IR manipulation tools, the process will require several insertions and replacements of nodes, together with manually created instances of nodes to recreate the tree. If the parameters of the library change at any point, or we want to implement additional operations, it would be necessary to re-write most of the code as the new IR will be different.

To facilitate the manipulation of the IR, it is possible to create a template. A template is a Python string with placeholders for variables, indicated by `${...}$`. The template engine (Mako [93] in the current version of `YaCF`) will replace these placeholders with the values of the variables at runtime, and generate a new string with the information. This string can be parsed with the *parse_snippet* method of the *Mutator* class, generating a new IR.

An example of a template is shown in Listing 3.23. Placeholders can also be used to insert raw Python code, that will be evaluated when parsing the template (see Line 4). Python functions used inside templates can be declared on external modules (Line 2 for the import, Line 7 for the usage). Some commonly used functions are available on the FUNCTIONS module of the TEMPLATEENGINE directory. Template comments (i.e. code that will not be evaluated by the template engine and that will not appear on the destination code) can be specified using double hash (`##`) before the text (Line 6). More complex control structures, like if/else or `for` statements (Line 5) are available. These loop control structures enable back-end writers to express complex code patterns in the source code in a clear and readable way.

To parse a template, the user has to specify all the variables of the template as it is shown in Listing 3.24.

The `name` and `show` parameters are used for debugging purposes. In the event of an error parsing the template, an exception is raised. If the show parameter is set to `True`, the template is also printed to the standard output.

The code generated by the template must be syntactically correct thus, attention is required in situations where the snippet might not form a valid C code.

```
1   <%
2       from Backends.Common.TemplateEngine.Functions import decl_of_id
3   %>
4   int ${functionName} (${','.join(loop_parameters['inside_vars'])}) {
5            ## This function is just a test
6            %for elem in loop_parameters['inside_vars']:
7                ${decl_of_id(elem)}
8            %endfor
9
10         }
```

Listing 3.23: Example of template mixing C and Python code with the template tags

```
1   self.parse_snippet(template_code, {'reduction_vars' : reduction_vars,
2                     'shared_vars' : shared_vars}, name = 'Retrieve', show = False)
```

Listing 3.24: Calling the *parse_snippet* from the *Mutator* to generate the AST of the code after filling the template

3.6.2 The DOT Back end

To facilitate understanding the IR representation, and to alleviate the development effort when creating source transformations, it is necessary to provide developers with tools to represent the IR at any point. A DOT back end has been implemented to facilitate this task.

DOT [132] is a plain text graph description language. The language allow to describe graphs that both humans and computer programs can use.

The DOT back end contains a *Visitor* which creates a representation of the graph in the DOT language. Nodes in the DOT graph are nodes of the DOT language. Arcs in the DOT graph represent the relation between IR nodes (which node contains which one).

This back-end is used by the *DOTDebugTool* to create snapshots of the translation process and debug the internal IR. However, it can be used standalone to print the results of a StS translation to a file, or as part of any other tool. Listing 3.25 shows part of the output of the DOT commands used to generate the Figure 3.3.

3.6.3 The Writer Classes

A Writer is a class implementing a *Visitor* pattern which traverses the IR generating a source code. This source code could be the original, like in a C-to-C translator, or a different one. Writers can be applied to any AST subtree, although some implementations of the Writer might require features only available on augmented IR.

As is typical in other classes following the Visitor pattern, each method *visits* an element of the AST/IR, and performs an action.

```
digraph G {
FileAST__DOT__0[label = "FileAST"];
FileAST__DOT__0 [shape=box, color=red, style=filled];
FuncDef__DOT__1[label = "FuncDef"];
FileAST__DOT__0-> FuncDef__DOT__1[label = "ext"];
Compound__DOT__2[label = "Compound"];
FuncDef__DOT__1-> Compound__DOT__2[label = "body"];
i_3[label = "i"];
Compound__DOT__2-> i_3[label = "block_items"];
TypeDecl__DOT__4[label = "TypeDecl"];
i_3-> TypeDecl__DOT__4[label = "type"];
int_5[label = "int"];
TypeDecl__DOT__4-> int_5[label = "type"];
int_5-> int[label = "names"];
n_7[label = "n"];
Compound__DOT__2-> n_7[label = "block_items"];
Constant__DOT__8[label = "Constant"];
n_7-> Constant__DOT__8[label = "init"];
TypeDecl__DOT__9[label = "TypeDecl"];
n_7-> TypeDecl__DOT__9[label = "type"];
TypeDecl__DOT__9-> int[label = "type"];
pi_11[label = "pi"];
Compound__DOT__2-> pi_11[label = "block_items"];
TypeDecl__DOT__12[label = "TypeDecl"];
pi_11-> TypeDecl__DOT__12[label = "type"];
double_13[label = "double"];
TypeDecl__DOT__12-> double_13[label = "type"];
double_13-> double[label = "names"];
sum_15[label = "sum"];
Compound__DOT__2-> sum_15[label = "block_items"];
TypeDecl__DOT__16[label = "TypeDecl"];
sum_15-> TypeDecl__DOT__16[label = "type"];
TypeDecl__DOT__16-> double[label = "type"];
...
```

Listing 3.25: Example of the *DOT* Back end output

3.6.3.1 OffsetWriter

When unparsing codes (i.e. recovering the original source from the IR), it is interesting to recover not only the source code but some part of the original indenting, or at least make a best-effort attempt to generate readable code. To accomplish this task, a basic class from where writers can be inherited is available. *OffsetWriter* offers basic features to write strings to a file descriptor (standard output by default) providing an offset value. When writing the output, offset number of spaces will be prepended to the string.

3.6.3.2 C99Writer

C99Writer unparses a C99 [96] IR back to C99 code. Some considerations have to be taken into account:

- Comments are not restored as they are not in the IR.
- Original parenthesis are lost during parsing, Writer has its own rules to apply parenthesis to expressions.
- It does not handle `pragma` statements.

3.6.3.3 OmpWriter

OmpWriter unparses a C99 IR with OpenMP [108] annotations back to the original. The considerations are the same as those mentioned above, except that any OpenMP pragma is printed conforming 3.0 revision of the standard ([107]).

3.6.4 The CUDA Back end

The CUDA module features a set of *Mutators* capable of generating CUDA code from OpenMP sources. See Section 5.2 for details. In the following paragraphs we focus on the components that are currently supported: The *Kernelize* and the *Platform* classes. In order to extract and write a CUDA kernel from a Loop first the Loop parameters need to be extracted using the *ParametrizeLoopTool* as described in Section 3.5.2.

3.6.4.1 Platform

The variables used in the loop have to be classified to decide its location in the device memory. Table 3.2 shows how input variables are transformed.

The class *Platform* also performs several analysis over the kernel code to facilitate optimization on further phases. Three parameters are extracted from the analysis: **Number of Flops**, **Number of Memory Accesses** and **Divergence factor**.

Kernel Parameter	**Value**
Kernel Parameters	All variables used in the kernel except from register variables
Reduction Variables	Variables in the reduction list of the loop parameters
Registers	All variables declared inside the kernel
Code	Loop body

Table 3.2: Placement of variables according to loop information

3.6.4.2 Kernelize

The *Kernelize* receives the parameters of the kernel (as returned by `Platform`) and writes the final device kernel. Writing the kernel makes use of the Template Subsystem. Templates are filled with the parameters. Some convenience functions are implemented on the template layer to convert Symbols into different forms, such as declarations, parameters or pointers. These handler functions, stored in BACKENDS.COMMON.TEMPLATEENGINE.FUNCTIONS work on Symbol nodes and are able to make basic representation transformations on the template itself. In addition to writing the kernel, *Kernelize* also calls the *inlineCalledFunctions* to ensure that the functions called from the kernel are inlined and ensures that the only functions called but not defined are native functions of the device (such as `sqrt` or `exp`). A configuration parameter enables this method to replace the precise version of these functions by a less precise, but faster, implementation on the device hardware. *Kernelize* uses separate kernels for different situations, depending on the kind of kernel (1D, 2D or 3D kernels) or if there are reductions involved.

3.6.5 The OpenCL Back end

In a similar fashion to the CUDA Backend, the OpenCL back end implements the generation of OpenCL code. The structure of the back end is the same. The *Platform* is just a placeholder to call the CUDA version as the parameters required are the same. The **Kernelize** implements OpenCL specific semantics using a different set of Templates for each different class of kernel.

3.7 Final Remarks

In this Chapter we have provided a detailed description of our StS tool: `YaCF`. The `YaCF` tool has enabled us to fast-prototype languages, techniques and optimizations with very little development effort on our part, and it has provided us with a steady learning curve. `YaCF` is not a production-ready compiler. Transformations are not entirely safe and might generate incorrect code. However, the tool suffices for a controlled research environment in which the developer is focused on the features offered, rather than on its completeness.

CHAPTER 4

The Frangollo Runtime

A significant amount of programming is required to handle the complexity of heterogeneous platforms. Most of this effort is devoted to re-implementing several patterns such as initializing devices, managing memory transfers, or loading the kernel code. Storing the instantiation of those patterns into a library which can then be accessed by developers can greatly facilitate the development effort.

Our implementation of such a pattern-storage library is called `Frangollo`. `Frangollo` acts as an interface between the user (or an intelligent enough compiler) and an abstract accelerator model. The library has a programmer-friendly interface for the most common usage patterns of accelerators, and enables platform-independent code to be written. The platform on which the model will be instantiated can be decided on at runtime, once a check has been carried out to identify available devices.

Our aim has not been to completely reproduce a virtual machine for the accelerator. Our intention is simply to provide an abstract layer across different accelerator platforms. It is the responsibility of the programmer (or, once again, an intelligent enough compiler) to provide the source of the kernel. `Frangollo` supports different implementations for the same kernel, and will choose the appropriate kernel source file for the destination platform.

4.1 The `Frangollo` Platform Model

The `Frangollo` Platform Model is inspired by the OpenCL Programming Model. Instead of focusing on the devices and their low-level details, the `Frangollo` Platform Model offers a generic approach to the concept of an accelerator, and includes some additional rules for handling memory coherence between the Host and the device(s). Our intention with this platform model is to facilitate the understanding of the runtime and to provide a set of rules to coherently handle host+accelerator programs, rather to present a new programming paradigm for these architectures. From this point on, we shall refer to any program which conforms to this model as a `Frangollo` program.

4.1.1 Components

We can define an accelerator as a *special-purpose co-processor attached to a Host CPU onto which the Host can offload data and compute kernels for the purpose of performing compute-intensive calculations*. *Offloading a code* implies *running a function of the main program into an accelerator then returning its results to the Host*.

A *Platform* is a set of one or more accelerator devices connected to a Host.

Programming languages reference data in the source code using variables. From now on, we will refer indistinctly to data and variables.

4.1.2 Execution

Programs running on the *Platform* may contain portions of code set to be offloaded to the accelerator.

Usually, accelerators do not share the same instruction set as the Host CPU. The code to be offloaded needs to be written in the native language of the accelerator. Separate source codes are required for the Host CPU and the offloaded code.

In addition, data is not shared across devices on the platform. The offloaded code and the data required to execute it define a *Context*. Once a host program instantiates a *Context* it is possible to register data on it. While the *Context* is active, data is available for the offloaded code (kernel). *Context*s may be nested. When a *Context* is created inside another *Context* they share information about the current *Platform*. The inner *Context* has access to its kernels and registered data. It can also access the kernels and registered data of any ancestors. The ancestors of the new *Context* have no access to the inner kernel or registered data. Only one active *Context* is allowed at any given instant. Different host threads may or may not share the active *Context*, however, this will be dependent on the implementation.

4.1.3 Parallelism

We can define a `thread` of an accelerator as the *the smallest sequence of programmed instructions that can be managed independently by an accelerator*. Accelerators support fine-grain parallelism using several threads concurrently.

An accelerator has many processing units. Each processing unit is capable of executing a number of threads in parallel. Processing units run in parallel, so different groups of threads can be executed simultaneously in different processing units.

It is possible to synchronize threads inside a group, but not all accelerators support synchronization across different processing units.

Hosts may have several accelerator devices attached. Synchronization across separate devices inside a `Platform` is not supported by hardware. The Host is responsible of managing device coordination and orchestrating the kernel execution.

4.1.4 Memory Model

In a similar fashion to the OpenCL memory model (see Chapter 2) we identify four types of memories:

- Global memory.
- Local memory.
- Private memory.
- Constant memory.

Global memory is available to all threads in the device but at low speed. Local memory is available to all threads in a group, but cannot be allocated nor accessed from the Host, only from the device. Each thread has its own private memory.

4.1.5 Memory coherence

Host and devices inside a platform do not share address spaces. In order to copy data from the Host to a device or from a device to another device, a memory transfer through a slow bus is required.

To execute the offloaded code both the Host and the device(s) need to share the data. It is not possible to copy the entire memory of the Host to the device due to time and space constrains. Any data that could be referenced from an offloaded region needs to be registered. *Registered data is available in the device when the offloaded code is executed on it* (**Condition 1**). The Host can transfer the data at any moment using any method it has available as long as it satisfies this condition.

Data registered in a *Context* may have consistency constrains. Data whose initial value is required inside the kernel has an INPUT constrain. If the data is required in the Host after the *Context* is finished it has an OUTPUT constrain. If data has both requirements it has an INOUT constraint. This also applies to variables registered inside nested *Context*s.

Data registered on a Context needs to be allocated on all devices of the Platform (**Condition 2**). It also has to be freed from all devices of the platform after the *Context* ceases its existence.

We can map the correlation between host and device variables in a graph where the nodes represent the data and the edges represents consistency relation (i.e. if one is updated the other has to be updated as well). A non-directed correlation graph (NCG) represents the overall relation among all the variables in the platform. If a variable in a device is modified, all related variables need to be updated as well.

According to the *Context* constraints and **Condition 1** we can define at least two situations in which operations have to be executed in order to keep the coherence of the data across all the variables: Kernel execution and *Context* creation/destruction.

The operations required to keep the coherence in both situations can be mapped in a `Directed Correlation Graph` (DCG). The nodes of the DCG represents data (variables) that have been allocated to a particular level of the device from the Platform. Nodes are named according to where the data allocated using the following convention:

- H nodes represent memory allocated on the Host.
- D nodes represent memory allocated on a device.
- D_g represents memory allocated on the global memory of the device.

- D_l represents memory allocated on the local memory of the device.
- D_p represents memory allocated on the private memory of the device.

By default, unless stated otherwise, when we use D we are referring to D_g. To distinguish two different set of data, a letter may also be used to name the node. For example, A_H is the Host data named A, or just variable A in the Host. A_{D1} will refer to variable A in device 1. Again, for simplicity, A_H can be written just as A.

We distinguish two different types of DCG. A pre-DCG is formed by the operations required to keep the coherence before a kernel execution or a *Context* creation/destruction. The arcs of the pre-DCG represent the operations required to keep the coherence of the data across two nodes. [1]

Given two nodes, H and D, an arc from H to D ($H \rightarrow D$) represents an operation which takes the current value from H and writes its value on D. This is called a *copy_in* operation to accommodate the nomenclature to the common concepts used in accelerators. It is possible to *copy_in* to another device $D1 \rightarrow D2$ or from global memory to private memory $D_g \rightarrow D_p$. An arc such as $H \rightarrow D_p$ is not valid as the Host cannot access private memory directly. However, it is possible to chain arcs $H \rightarrow D_g, D_g \rightarrow D_p$.

A post-DCG is formed by the operations required to keep coherence after kernel execution or *Context* creation/destruction. In this case arcs from D to H ($D \rightarrow H$) are called *copy_out* operations.

If we do not take into account the constraints restrictions of the *Context*, the post-DCG would be the transpose of the pre-DCG. For each `copy_in` ($H \rightarrow D$) in the pre-DCG, the post-DCG would feature an opposite `copy_out` ($D \rightarrow H$) operation. However, the existence of constraints on the *Context* breaks the symmetry of the pre- and post-DCGs because some arcs may be unnecessary.

Figure 4.1 shows a simple action in which a registered host variable is also allocated in the device.

Figure 4.1: Basic pre-DCG representing a variable in the device that needs to be updated with data from the Host

[1]Using three different graphs to represent what basically is a set of memory copying operations might be seen as useless. However these abstractions are needed when we extend the concept of operations in the following paragraphs.

A more complex example involving two nested *Contexts* is detailed in Figure 4.2. This time, the Host instantiates a *Context* and registers the variable A with INOUT directionality. This updates the correlation graph with two additional nodes, A which represents the data in the Host and A_D which represents the data in the device (to satisfy **Condition 2**, *variables have to exist on all devices of the platform*). Then the Host instantiates a new nested *Context* and a new variable is registered with OUTPUT constraints. This updates the correlation graph by adding two additional nodes, B and B_D. Before executing the kernel, the pre-DCG is formed with the required operations to satisfy **Condition 1** (*data has to be available for kernel execution*). In this state only variables related to A need to be updated (B has OUTPUT constraints only), thus the graph contains $A_D \rightarrow A$. After kernel execution the post-DCG graph is empty. At this point, no operations are required to satisfy conditions or constraints. When the Host destroys the *Context*, its memory needs to be updated with the information on the device to satisfy the OUTPUT constraint. Then the post-DCG contains $B_D \rightarrow B$. Since the *Context* is destroyed, the correlation graph no longer shows B or B_D.

Next, a kernel using the variable A is executed. This time data is already available on the device, so both pre- and post-DCG are empty. Finally, the Host destroys the remaining *Context*, so A needs to be de-allocated. To satisfy constraints, the post-DCG shows an operation from the device to the Host ($A_D \rightarrow A_D$).

It is possible for the Host CPU to modify the data after it has been registered. This situation forces the update of the pre-DCG , but does not imply an immediate transfer as long as the system guarantees **Condition 1**.

4.1.6 Composing operations

It is possible to elaborate more complex operations for transferring data across variables in the model. In the Message Passing Model (MPI, Chapter 2) several collective operations are defined, such as scatter, gather, reduction or broadcast.

We can define similar operations in our memory model. Given nodes H , D_1 and D_2 , a scatter operation from H to D_1 and D_2 will partition the variable in H into two subsets, H_1 and H_2. Each subset will be transferred to each device (adding the arcs $H_1 \rightarrow D_1$ $H_2 \rightarrow D_2$) The opposite of this scatter operation is a gather operation: from the data in nodes D_1 and D_2 of the graph we compose the value of the data H.

A reduction operation is similar to the gather, but instead of composing H with consecutive portions of D_1 and D_2, the value of H will be the result of repeatedly applying an operation $f(H, x) = H \star x$ with $x = D_1, D_2$.

Finally, for convenience, we define a trivial broadcast operation, where $H \rightarrow D1$ and $H \rightarrow D2$.

To summarize all operations, Figure 4.3 shows how a variable is reduced on the Host after expanding its value on the accelerator's private memory.

To expand the value to the accelerator private memory, the kernel must create a private variable for each thread and copy the value from the global memory. After finishing the execution, the kernel will copy the value of the private variable to the global memory.

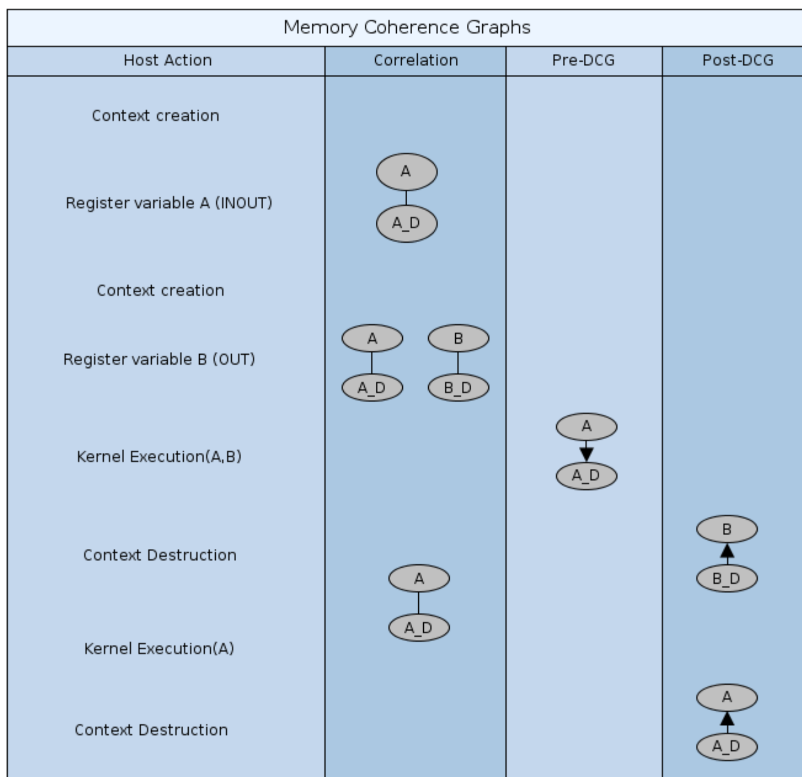


Figure 4.2: Evolution of the three graphs (correlation, pre- and post- graphs) across an example program execution on a platform with a Host and an attached device

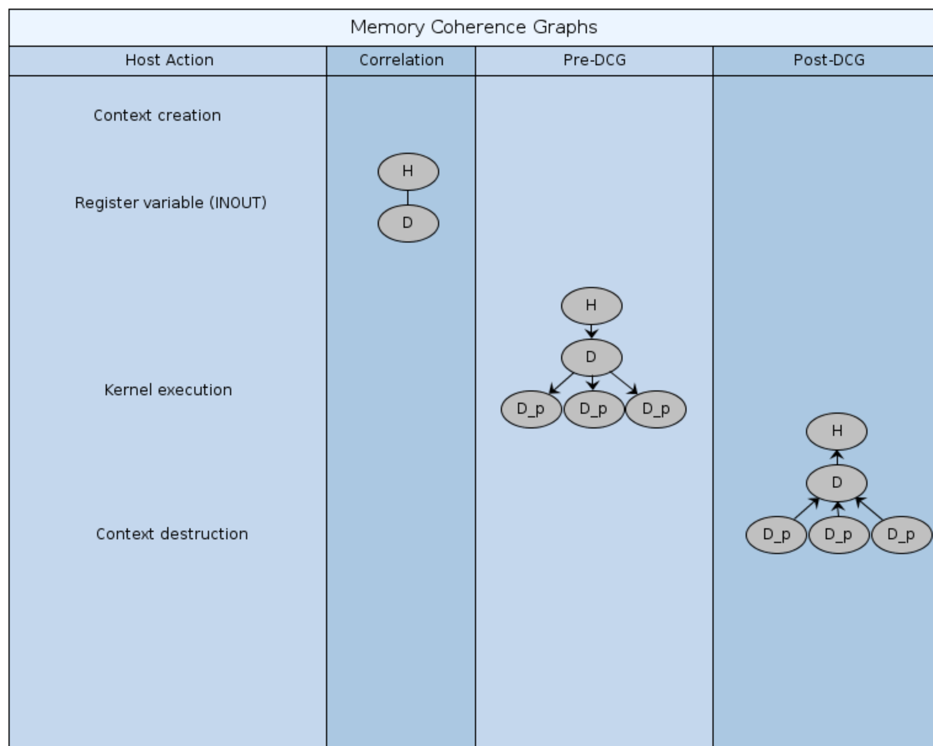


Figure 4.3: Evolution of the three graphs (correlation, pre- and post- graphs) across a Host with a device attached performing a reduction on its private memory

4.1.7 Applying the Frangollo Platform Model

The abstract model could be instantiated on a wide variety of devices. FPGAs can be used to offload parts of the code, but they require the data to be transferred into the board in order to operate. Other special-purpose devices, such as SSL accelerators, can also fit the pattern.

The most common accelerator nowadays is the GPU. The internal GPU architecture has been described in Chapter 2. The parallelism model of the GPU fits the platform model nicely. GPU kernels are sent to the device in assembly. Kernels run on a grid formed by a set of blocks of threads. GPU threads run in parallel in groups called blocks. Blocks are decomposed into warps and executed in the processing units. In CUDA, it is possible to perform synchronization inside a block, but not across blocks. Depending on the hardware this may be possible on OpenCL. This may be possible on OpenCL, but it will depend on the hardware being used.

The execution model is simple: prior to the offload of a code to the GPU, the *Context* is created. Both in CUDA and OpenCL this involves device discovery and initialization. Several devices might be attached to the same Host, and both PMs support handling separate devices. Devices can communicate using memory transfer operations.

Basic *copy_in* and *copy_out* operations can be implemented using the low level memory transfer operations of the device.

It is also possible to instantiate the model on the CPU itself. Using the OpenCL platform it is possible to transparently use a GPU-like interface to use the CPU's vector units. In this case the global memory could be the Host memory, as the accelerator shares memory space with the Host. Despite the fact that invoking a memory transfer might not be necessary, transfers can be seen as synchronization points between the Host and the device. Implementing a copy_in or copy_out on a Host's memory may be as easy as performing an empty operation. Composite operations, such as reductions, can be performed using temporary storage and then copied to the original positions.

4.2 Software Architecture

Frangollo uses a three layer approach to implement its platform model. The general structure of Frangollo is shown in Figure 4.4.

The outermost layer is presented to the user (interface-layer), whereas the innermost one (device-layer) encapsulates device-specific code. The intermediate layer contains high-level and logical operations with a generic representation of the devices. This layer implements most of the logic of the Frangollo platform model. For example, it is possible to copy a variable into another one without bothering with device-implementation details.

4.2.1 Abstract Layer

The Abstract Layer implements the Frangollo Platform model.

When the runtime is loaded, it creates a singleton instance of the *System*. The constructor of this class performs the following actions: it discovers the available platforms and devices; it creates the appropriate data structures to handle them; and it also looks for environment variables that might affect runtime behaviour.

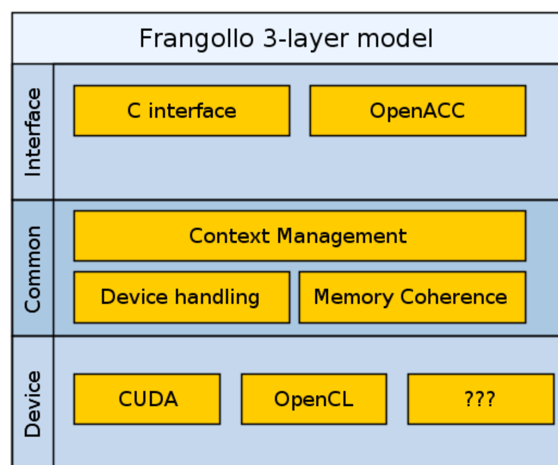


Figure 4.4: Overview of Frangollo layers

4.2.1.1 Context

The class *Context* implements the model Context. An important attribute of this class is the NCG, the relation between variables in the Host and the device. The NCG is implemented using an STL associative container, ensuring that search operations complete in $O(log(n))$. Each element of the associative container is a pair. The first element (key) is the address of the data in the Host. The second element is a list of pointers to structures managing the variables in other devices. These structures are instances of derivates of the abstract *Var* class. The *Context* class features methods to look for variable correspondence or ensure synchronization between the Host and the devices. Handling of nested *Contexts* is implemented through scope subsystem to reduce overhead. Only one instance of the *Context* is active at any given time. An instance of class *Context* contains a property called scope, implemented as an STL map with pairs of level and variable.

If a new instance is requested while another is running, it returns a pointer to the existing one and adds a new level to the scope property. All the registered variables are also added to this level of the scope map.

When the nested Context is finished, either calling the internal method decreaseScope or the class destructor, all the variables registered on it are removed. When variables are removed, its constraints are checked and enforced, thus the rules for nested *Contexts* are preserved.

Some other relevant methods of the class *Context*:

- setName and getName: Sets or retrieves the name of the *Context*.

- `addDevice`: Adds a device to the *Context*.
- `removeVar(Var *)`: Completely removes a variable from the *Context*. This forces checking its constraints, thus, might trigger `copy_in` or `copy_out` operations.

4.2.1.2 Variables

Var class defines the basic operations shared by all device implementations:

- `allocate` / `deallocate` : Allocates or deallocates the variable.
- `copy_in(Var *)` : Copies the value of the variable passed as parameter to the current variable.
- `copy_out(Var *)` : Copies the value of the current variable to the variable passed as parameter.

This class also defines the properties shared by all variables, such as their size, which device they are allocated to, their constraints and their shape. The shape of the variable allows the runtime to distinguish scalars from arrays. Non-contiguous arrays are also supported, allowing the runtime to use specific functions to improve performance. An important internal property of *Var* is the `status`. Any instance of *Var* is in one of the following status:

- `NULL` : Variable has not been allocated.
- `EMPTY` : Variable has been allocated but no data has been transferred or written.
- `COPY_IN` : Variable is currently being copied in.
- `COPY_OUT` : Variable is currently being copied out.
- `READY` : Variable is ready to be used.
- `INUSE` : Variable is currently beign used by a kernel.
- `FREED` : Variable has been freed.

Figure 4.5 shows how the status of a variable changes during execution. Variables can progress directly from `EMPTY` to `READY` directly if they have only `OUTPUT` constraints.

Note that, although it is possible for variables to progress from `EMPTY` to `COPYOUT`, this might indicate an error in the program. Program correctness is the responsibility of the programmer, and the runtime cannot check if the constraints or the copy operations are correct with respect to the non-offloaded part of the code.

The abstract layer relies on the device layer for implementation details. The logic of the model is implemented in a platform-independent fashion. An example of this platform-independent programming style is shown in Listing 4.1. This code receives a variable from the Host, looks for a correspondent variable in the device and updates it. If the variable does not exist, it is created. Notice that we ask for the current device to return a pointer to a new variable, but the programmer does not need to know which platform s/he is working on.

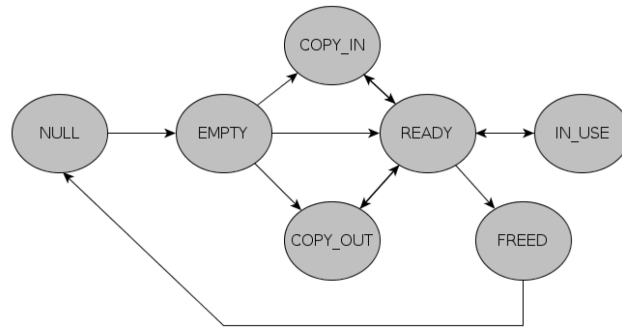


Figure 4.5: Transition state diagram of a *Var* instance

```
1  void Context::updateHostVar(HostVar * hostVar) {
2     /* 1. Find equivalent device Var */
3     CorrMap::iterator iter;
4     Var * d_a = NULL;
5     if (iter == _currentMap.end()) {
6        /* 2. Create device var if does not exist */
7        d_a = this->getDevice()->createVar();
8        d_a->setSize(hostVar->getSize());
9        d_a->alloc();
10    } else { d_a = iter->second; }
11    /* 3. Copy contents of hostVar to device Var if */
12    d_a->copy_in(hostVar);
13 }
```

Listing 4.1: Platform-independent update of a host variable

4.2.1.3 Devices

The *Device* class represents a hardware device. Generic operations of devices are defined in this abstract class. The most relevant of these operations are:

- *kernel_launch(Context, Name, param_list, dimensions)*: A kernel with the given name defined in the given *Context* is executed on this device with the given parameter list. The overall dimensions of the kernel represent the total number of threads required to execute the kernel. Dimensions are composed of three components (x, y, z), defining whether if the kernel is one-dimensional, two-dimensional or three-dimensional.

- *kernelPreload*: Allows the device to prepare a kernel for subsequent execution. This includes: loading the assembly, compiling the code and preparing the internal control data structures. This operation allows to reduce the time of kernel execution particularly when the same kernel is invoked several times.

- *createVar*: Returns a pointer to a variable in the device.

- *barrier*: The device will return the control from this method once it has finished all its operations.

- *estimateThreads(kInfo)*: Estimates a suitable number of threads to use in the kernel using the given kernel information (`kInfo`).

Estimating the number of threads for a particular kernel is one of the key capabilities of the runtime. An incorrect estimation of the number of threads for the device might hinder performance, particularly when using GPU devices.

When a kernel is added to a *Context* using *kernelPreload*, information about it characteristics is stored into the *Device* class. This information can be updated at any moment.

The method *kernel_launch*, which previously invokes the low-level kernel launch, attempts to estimate the number of threads using the method *estimateThreads*. Derived classes can replace this method with a more accurate device-specific implementation.

The default device-independent method is based on the notion that the amount of time required to perform a read or write operation to/from the global memory of a device is C multiplied by the time required to perform a floating-point operation. Although this might be imprecise (the time required for a floating-point operation greatly varies from device to device and different operations might consume more or less time), this notion enable us to understand if the memory or the floating-point operations are the critical bottleneck of the code.

From this notion we can compute the *Intensity* of a kernel as shown in Equation 4.1:

$$Int = C \times (N_{flop}/T_{access}) \tag{4.1}$$

The constant C is called the *Granularity Factor*(GF). Each kernel has an associated *intensity*. Devices can modify the value of GF according to its characteristics. An environment variable (`FRANGOLLO_GRANULARITY_FACTOR`) is also available to users. The *Intensity* information it is used to make a first estimation of the number of threads per block.

Theoretically, a larger number of threads benefits parallel computation. However, in practise a larger number of simultaneous threads accessing random positions in memory (*non-coalescenced memory access*) decreases performance.

Using the *Intensity* information is possible to detect if the kernel contains a large number of floating-point operations with respect to the number of memory access and whether it would benefit from a larger number of threads. However, if the kernel features a large number of memory access over the number of floating-point operations, it might equally benefit from a lower number of threads. Therefore Equation 4.2 shows how the high-level estimator computes an estimated number of threads based on the *Intensity*.

$$Th_{est} = max(Th_{max} * Int, Th_{max}) \tag{4.2}$$

4.2.2 Device Layer

The components of this layer are device-specific and deal with low-level operations on the hardware.

Figure 4.6 shows the class diagram of the CUDA component. The OpenCL component class hierarchy features a similar design.

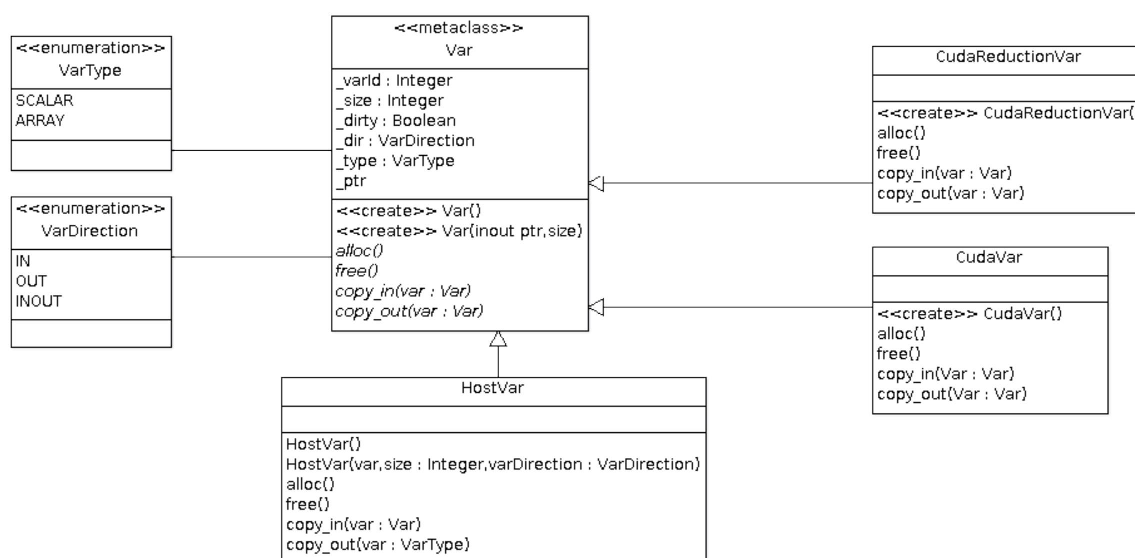


Figure 4.6: UML Diagram of the most relevant classes within the Frangollo runtime

As an example of how this abstract layer has been designed, Listing 4.2 features the implementation of the copy_in method of the class *OCLVar*. Listing 4.2 features the implementation of the copy_in method of the class OCLVar, providing an example of how this abstract layer has been designed.

```
1  void OCLVar::copy_in (Var * var)
2  {
3      Var::pre_copy_in (var);
4      cl_int status;
5      cl_mem *p = (cl_mem *) this->getPtr ();
6      status = clEnqueueWriteBuffer (*_cmdQueue, *p,
7                  BLOCKING_TRANSFERS?CL_TRUE:CL_FALSE, 0, var->getSize (),
8                  var->getPtr (), 0, NULL, NULL);
9      if (status != CL_SUCCESS || *p == NULL)
10     {
11         STOP("clEnqueueWriteBuffer failed with a variable of size %d\n", this->getSize())
           ;
12     }
13     Var::post_copy_in (var);
14 }
```

Listing 4.2: copy_in method from OCLVar

4.2.2.1 CUDA Component

The CUDA component implements the device-specific details of the abstract layer. It is implemented using the CUDA Device API rather than using the most common CUDA Runtime API in an attempt to exclusively use standard C/C++ code. *CudaDevice* class, derived from *Device*, represents a GPU device with CUDA support. It implements the abstract methods plus the following device-specific methods:

- `loadPtx(name)`: Loads a PTX assembly file with the given name and creates a kernel function structure.
- `computeOccupancy(numRegs,nThreads,shMem)`: Computes the occupancy achieved in the current CUDA architecture for a kernel with the given characteristics. Equations to compute the occupancy have been extracted [102].
- `maximizeOccupancy(numRegs,nThreads,shMem)`: Using the given parameters to feed the previously defined function, it attempts to find a suitable number of threads to maximize the occupancy rate.

The value from the generic thread estimator is used to feed the initial value of the *maximizeOccupancy*, whose pseudo code is shown in 4.1.

4.2.2.2 OpenCL component

The OpenCL component is implemented using the C interface of the language, and it features a similar structure to the CUDA component.

Algorithm 4.1 Pseudo-code of *maximizeOccupancy*

function MAXIMIZEOCCUPANCY($Th_{est}, Reg_{used}, Shmem_{used}$)
  for $i = (Th_{est}/Th_{Warp}) \rightarrow (Th_{max}/Th_{Warp})$ **do**
    $tmp \leftarrow computeOccupancy(i, Reg_{used}, Shmem_{used})$
    $optimal \leftarrow max(tmp, optimal)$
    if $optimal > MAX_{OCCUPANCY}$ **then**
      break
    end if
  end for
  return $optimal$
end function

OpenCL also supports the usage of `map` and `unmap` operations instead of traditional memory transfers. This enables the device to directly access to memory regions of the Host CPU. Two situations are detected by the component at runtime, which triggers this optimization: (1) whether the accelerator is a CPU, for example, when using the Intel OpenCL platform or (2) whether the accelerator is a GPU but it has support for pinned memory. Users can disable this behaviour using the appropriate flag in the configure script.

4.2.2.3 Reductions in CUDA and OpenCL

Both CUDA and OpenCL components implement support for reduction variables as defined in the Frangollo Platform Model. When a variable is registered it is possible to mark it as a reduction variable. When a kernel performing the reduction is invoked, the parameter list contains a flag indicating the reduction operation. Currently supported operations are: Addition, Subtraction and Multiplication. Other operations can easily be implemented if required.

Reduction variables assume that the kernel stores per-thread copies of the reduction in a global memory array. The `copy_in` of a reduction var is just an empty method, but the `copy_out` method calls an optimised reduction kernel to reduce all the per-thread private variables into the final scalar result. In OpenCL two separate kernels are available - one for GPU and the other for CPU - each of which is optimised for one architecture or the other. The rules of the Frangollo platform model ensure that the value of the reduction var will be available when requested. The reduction kernel is executed asynchronously from the CPU.

4.2.3 Interface Layer

The interface layer provides functions and subroutines that interacts with the rest of the library infrastructure. Components of this layer are exposed to the user, or can be used as an interface to other languages.

4.2.3.1 C interface

All functions from this interface are prepended with `FRG__`. A basic example creating a `Context`, registering a variable and launching a kernel is shown in 4.3

```
int regionId = FRG__createContext("random", list);
FRG__setActiveContext(regionId);
FRG__registerVar((&a), regionId, SIZE, OPTIONS);
  ...
FRG__launch_kernel (...)
  ...
```

Listing 4.3: Example of calls to the Frangollo interface layer.

The most relevant functions are:

- registerVar: Registers a new host variable in the runtime
- launchKernel: Launches a kernel in the current *Context*.
- createContext: Creates a new *Context*.
- destroyContext: Finishes (destroy) a *Context*.
- setActiveContext: Sets the current active *Context*.
- getActiveContext: Returns the ID of the current active *Context*.
- barrier: Synchronizes Host and device(s).
- wait: Waits for an specific registered variable to be committed to the Host.
- printStats: Throws device-oriented stats to a file descriptor (*stdout* if no other specified).

4.2.3.2 C++ Layer

Developers can interact with the runtime directly from C++. Including header files for the key common components gives access to the abstract interface and to native pointers to the device variables.

It is possible to use compiler-defined macros to write specific code, for example, to differentiate situations where the underlying platform is only CUDA and as such a specific piece of code is used instead of the generic layer. This enables developers to override default device operations and write their own specific operations.

Listing 4.4 shows an example of code interacting directly with Frangollo in C++ (i.e. the interface layer is not used).

The developer gets the list of available devices from the System singleton and prints the available devices to the standard output. Using the HAVE_LIBOCL the developer knows if the OpenCL framework has been enabled when configuring the library, and can use code that is specific to the OpenCL environment. In this case, the developer is checking if the device is an OpenCL device, and prints the hardware information to the screen.

Notice that no runtime initialization or destruction is required. The runtime automatically loads when the program executes and it does the clean-up when finished.

```
1  #include "Var.h"
2  #include "System.h"
3  #include "Region.h"
4  #include "Device.h"
5  #if HAVE_LIBOCL
6  #include <CL/cl.h>
7  #include "OCLDevice.h"
8  #endif
9  using namespace std;
10 int main () {
11     int size = 10;
12     /**** Extract system information */
13     std::vector<common::Device *> devList = common::system.getDeviceList();
14     std::vector<common::Device *>::iterator it = devList.begin();
15     std::cout << " Available Frangollo Devices: " << std::endl;
16     for (; it != devList.end(); it++)
17         std::cout << " --> " << (*it)->getDeviceString() << std::endl;
18 #if HAVE_LIBOCL
19     std::cout << " OpenCL Platform enabled " << std::endl;
20     it = devList.begin();
21     for (; it != devList.end(); it++)
22       if ((*it)->getDeviceString() == ocl::OCL_STRING) {
23           std::cout << "OpenCL device use the following HW component " << std::endl;
24           std::cout << "--> " << (*it)->getHWDeviceString() << std::endl;
25       }
26 #elif HAVE_LIBCUDA
27     std::cout << " CUDA specific code " << std::endl;
28 #else
29     std::cout << " Unknown platform " << std::endl;
30 #endif
31 }
```

Listing 4.4: Calling Frangollo from C++

4.2.4 Overall Usage Workflow

A typical Frangollo execution workflow is shown in Figure 4.7 to illustrate how the different components of Frangollo are glued together: green boxes represent actions from the main program; yellow boxes contain operations performed by the runtime. When the program starts the global system singleton is created, devices are discovered and environment variables are loaded. At some point the main program will create a *Context* and instructs the runtime to pre-load a series of kernels that will be executed inside. Kernels can be added at any time but the sooner they are ready the less performance impact their load will have. When the main program registers a variable it updates its correspondence map (implementing the Frangollo Platform NCG). It detects that there is no equivalent variable in the device so a CudaVar instance A is created. Then a new *Context* is created from the main program. As there is an existing *Context*, a new scope level is increased. The new registered variable B will be allocated in the device, and associated with the new scope. When the main program issues a kernel launch with the variables A and B as parameters, variable A is copied in. In fact, the variable may have already been copied in. The current implementation of the OpenCL component asynchronously copies the variables at the moment they are registered so they are ready when the kernel is called. After the kernel finishes its execution the program releases the nested *Context*, thus the runtime internally decreases the scope level, which triggers an update of the B copy in the Host's memory. As the scope is no longer active and the value of the variable is updated in the Host we can deallocate B. The program now invokes a kernel with the variable A. Variable A is already on the device (and no update has been performed in the Host), thus no additional transfer is required. Finally the *Context* is destroyed, the value of A updated in the Host and the program finishes. When the program finishes, the global singleton is destroyed.

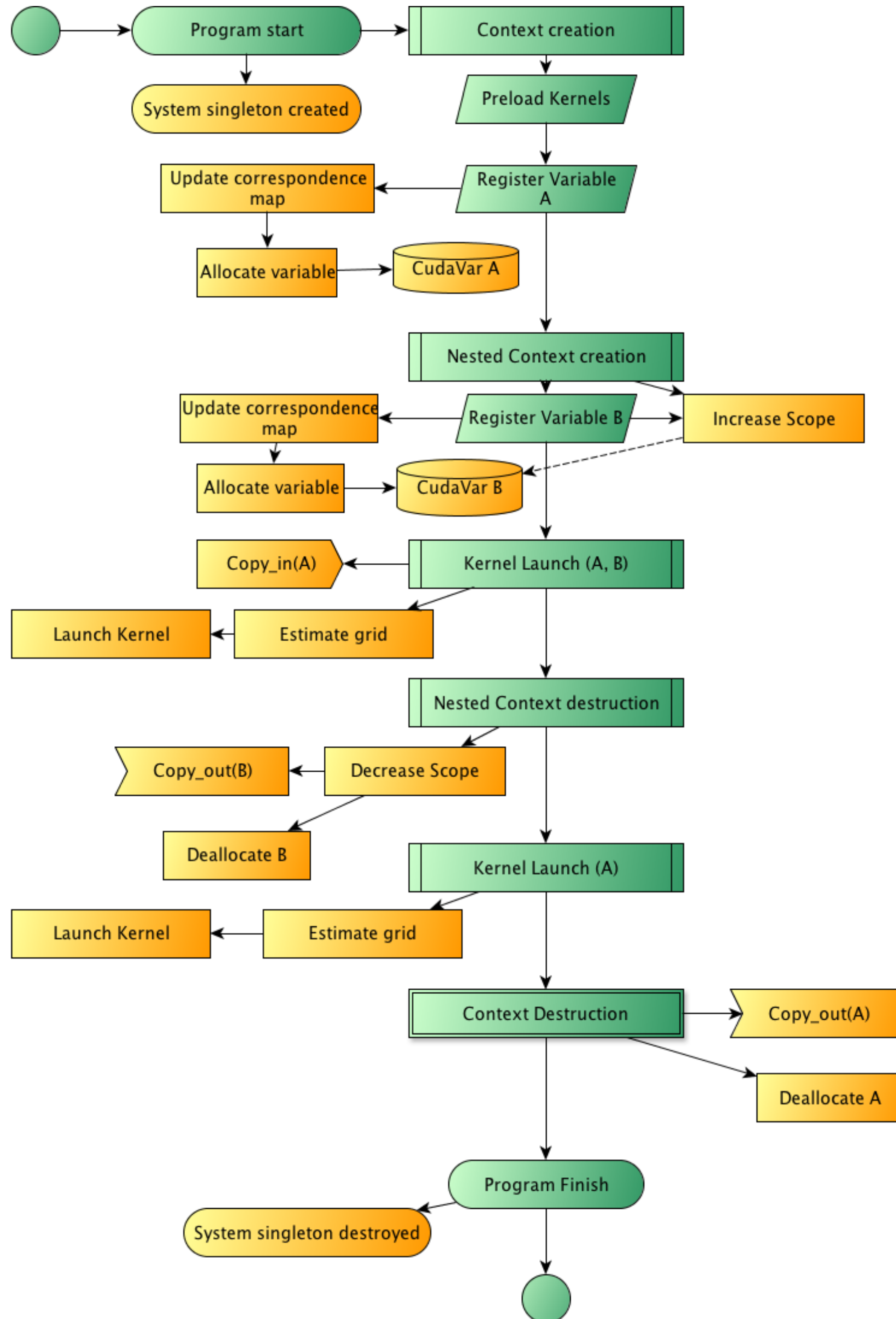


Figure 4.7: Execution workflow of the Frangollo runtime through the same example as that shown in Figure 4.2

4.3 Final Remarks

This Chapter described both the Frangollo Platform Model and the Frangollo runtime, which implements the model. The Frangollo Platform Model represents a set of rules to define the synchronization points between the Host and the device, together with a global explanation of an accelerator-based platform based in code-offloading. Although the current Frangollo runtime implementation does not support handling multiple devices simultaneously, the model allow us to easily support this situation. The Frangollo runtime is capable of handling several different environments whilst displaying the same programming interface to the developer, no matter which underlying platform is present. The only platform-specific information required by Frangollo is the kernel, and that can be extracted from the original code either by the user or by our YaCF driver.

CHAPTER 5

Directive-based Code Generation

In this Chapter we focus our attention on the different approaches to directive-based programming for heterogeneous systems that we have explored during our research. Sections are ordered chronologically and contain references to associated refereed publications. Section 5.4 of this Chapter is devoted to our most recent approach, `accULL`. We describe in detail how OpenACC directives are translated to runtime calls for this particular case.

5.1 Extending `llc` to Support Hybrid MPI+OpenMP Programming

In [121] we made an initial attempt to tackle the problem of automatic code generation for hybrid architectures from high-level standard languages, focusing on multi-core processors.

Starting from an `llc` [52, 54] source code, we can generate hybrid MPI/OpenMP code that is ready to be compiled in a hierarchical architecture using standard tools. An important feature worth highlighting is that the generation process is quite simple and the code generated is as efficient as ad-hoc implementations. Although there are different kinds of *parallel constructs* implemented in `llc` we focused on the loop-oriented constructs.

The OTOSP computational model (see Chapter 2) guarantees that each processor owns at least a part of the main dataset. We are therefore able to add a new level of parallelism by simply re-parallelizing the annotated loop. The tags in the static patterns allow the injection of this second parallel level to generate hybrid code. In this nested level, we use a shared memory approach to parallelization, using pure OpenMP code for the `llc`-annotated loop. By extending the original `llCoMP` compiler, we can generate the required OpenMP code to be injected from the source code. Listing 5.1 shows how the local partition of work for each processor is parallelized using OpenMP.

For the distribution between MPI and OpenMP threads, we chose to use one MPI process per computation node, and shared memory threads between the cores of the node. According to the classification in [114], we use a *hybrid masteronly* scheme. This allows us to take advantage of the shared memory in the cores, in particular of their shared caches, thus increasing the performance of the loops.

```
...
case LLC_BLOCK:
        {
          int llc_nP;
          register int llc_i;
          int llc_nT_save;
          int llc_grp_save;
          llc_var_reg_data llc_nc_result_list_save;

          llc_nP = LLC_NUMPROCESSORS;
          llc_grp_save = LLC_NAME;
          LLC_NUMPROCESSORS = 1;
          LLC_NAME = 0;
        /* HYB: #pragma omp parallel for ... */
        #pragma omp parallel for default(shared) private(i, x, llc_i) reduction(+:sum)
          for(i = (0) + llc_F[llc_grp_save]; i < (0) + llc_F[llc_grp_save] + LLC_PROCSGRP(
               llc_grp_save); i++) {
             {
              x = h * ((double) i - 0.5);
             sum += f (x);
             };
          }
          LLC_NAME = llc_grp_save;
          LLC_NUMPROCESSORS = llc_nP;
        }
...
```

Listing 5.1: Snippet from the code generated by llCoMP-HYB

Since our target systems in this work are multi-node clusters, the most efficient approach is to use one processor to communicate over the internode network, given a single processor's capability to use the entire network [113].

llCoMP generated code handles the MPI communications and synchronization of data between nodes. This code would not be enhanced by using more than one CPU as it mainly consists of synchronization routines.

Remarks

The possibility of taking advantage of emerging hardware architectures easily and without the need of modifying the original code was appealing. Extending llc to support hierarchical architectures such as multi-core clusters was successful and improved the performance of the pre-existing codes. However while implementing this approach we realised that the original design of llCoMP was not suitable for more complex hardware architectures, such as GPUs, where more complex StS transformations were necessary.

5.2 Generating CUDA Code from `llc` Sources

Our first approach to the generation of CUDA kernels was based on `llc` sources [122, 116, 117]. We implemented a `YaCF` driver capable of reading OpenMP + `llc` sources and generate CUDA runtime code. The new driver was called `llCoMP` as in the original `llc` compiler. Our aim was to replace the original `llCoMP` compiler with a driver written in `YaCF` .

Although `llc` supports the most common parallel patterns - *forall*, *sections*, *pipelines* and *task queues* - [54, 49], the current version of the CUDA `llCoMP` driver only supports parallel loops (*forall*). The current version of `llCoMP` support new extensions to the `llc` language that are oriented towards loop optimizations.

Since all OpenMP directives and clauses are recognized by `llCoMP`, we can produce different binaries (sequential or parallel) from a single source code depending on the compiler selected to translate the target code produced by `llCoMP`. Figure 5.1 illustrates this process.

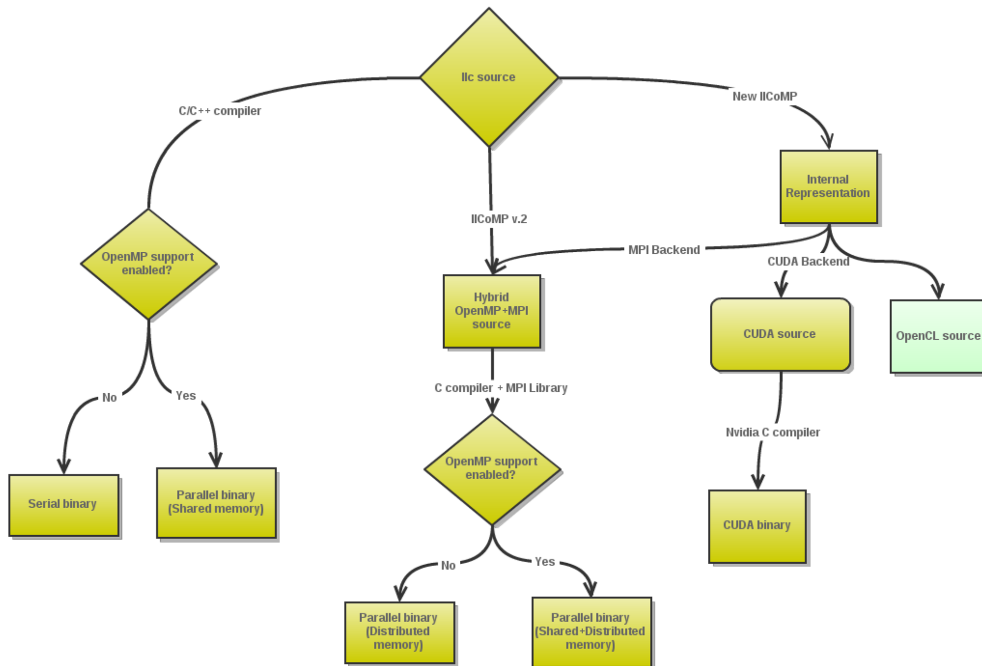


Figure 5.1: `llc` translation options

The `YaCF` drivers uses a set of *Mutator* classes to implement the StS transformation.

```
#pragma omp target device(cuda) copy_in(w) copy(pi_llc)
#pragma omp parallel for private(i, local) reduction (+: pi_llc)
   for (i = 0; i < N; i++)
     {
   local = (i + 0.5) * w;
   pi_llc = pi_llc + 4.0 / (1.0 + local * local);
     }
```

Listing 5.2: OpenMP implementation of π computation using the `target` extension.

This first approach to code generation did not use the runtime but instead directly generated CUDA runtime API code. The driver created a new instance of the *CUDATransformer* class. When applying the instance to the IR of the input file it ran a set of *Mutator*s to look for OpenMP directives. The OpenMP `for` constructs were converted into kernels using a primitive `Kernelize` implementation. No compute intensity information was extracted. Memory transfers relied entirely on the information provided by the user through the extended syntax proposed in [17].

Listing 5.2 shows an example code showcasing the use of these extensions. Due to the limitations of the compiler driver, scalars also needed to be annotated in order to properly outline the function. These limitations have been lifted in new versions (particularly within the `accULL` project).

After the transformations are applied, the IR is no longer an OpenMP code but a CUDA equivalent.

Current implemented patterns are:

- Device initialization: Implements low-level tasks, such as the device identification or the environment set up.
- Compute kernel invocation: The code to be parallelized is extracted from the `llc` source, and the compiler injects it in its corresponding place.
- Reduction operation: Reductions are common parallel operations. A specific template has been implemented in order to use an optimized reduction operation for each device.
- Environment cleaning.

The code in Listing 5.3, shows an example of a CUDA kernel template. `llCoMP` injects the optimized code into the specified placeholders.

```
(${kernel_parameters}) { ${loop_vars[0].declaration} ${loop_vars[0]} =
${kernel_thread_distribution};
 %for var in omp_clause.private_vars:
    ${var.declaration} ${var};
 %endfor
 if (${loop.cond}) ${loop.stmt}; }
```

Listing 5.3: llCoMP template for a CUDA kernel

Remarks

The experience of porting OpenMP/llc sources to CUDA was successful from the point of view of leveraging the development effort on the user side. Nevertheless, porting a shared-memory oriented PM such as OpenMP to a programming model oriented towards a different architecture was not completely satisfying from the point of view of performance. In OpenMP the memory is assumed to be coherent, i.e. all threads share the same view of the memory. If we limit the transformation of the source code to the `parallel for` directive we can translate nearly 1:1 of the semantics of both PMs. More complex situations are, however, not easy to translate. For example, situations involving `parallel` directives spawning a set of threads and then using the `omp_get_thread_num` to manually distribute the load across threads. In addition, OpenMP lacked syntax to handle the separate address spaces. Despite using the syntax of [17] for this purpose we realised that porting the semantics of OpenMP to generate the code of the accelerator would require violating the definition of OpenMP which would cause the syntax to be misleading to the users. To avoid this confusion and have more flexibility to implement an efficient offloading method, we decided to move on to a completely different syntax.

5.3 Intermezzo: llcl

In order to offer the user a better programming experience when using accelerator platforms, we decided to extend our original `llc` language to improve the support for heterogeneous systems. A preliminary version of this language was shown in [118], but has since evolved to become a completely new language called `llcl`. It attempt to mimics OpenCL semantics with OpenMP-like annotations, avoiding a plethora of low-level API calls.

In [17], Ayguadé *et al.* proposed a set of extensions to OpenMP in order to handle heterogeneous platforms. The original aim of the `copy_in|out` clauses is to identify which variables are required inside a particular task. These clauses might be used together with the `in out` clauses to identify data dependencies. Additionally, the `omp target device(...)` construct was used to specify which device should run the tasks which follow. Several tasks with different `target device` clauses might exist, these represent different implementations of the same tasks for particular devices. This enables users to have several different implementations on the same source code.

In `llcl` we do not focus on task parallelism, thus, we do not annotate code on a task-level basis only. The regions of code that are suitable for extraction and for running on a heterogeneous platform are marked. These regions of code (named `Context` following the OpenCL semantics) may contain one or several pieces of parallel code.

Variables used within a Context have to be registered, either as in or out variables, using the copy_in|out respectively. A name can be assigned to a Context for later reference.

Our aim is to maintain as much of the original source code as is possible and keep modifications to a minimum, taking into account that annotations have to be added in order to guide compiler optimization. The only annotations that will be inside the code are those related to data movement and parallelism definition. Device-specific information and particular optimizations will be defined in an external XML file.

The XML platform description file contains platform-dependant information. Giving this situation, it is possible to modify this file without altering the original source code. Several platforms can be defined and named in the same file. When executing the compiler driver the user can specify a target platform; if a target is not specified the first platform in the file will be used. Platform definitions specify context-related optimizations and targets of automated code generation. An example of a platform description file is shown in 5.4, whereas an example of llcl source is shown in Listing 5.5.

```
1  <xml>
2  <platform name="default">
3   <region name="compute">
4     <element name="compute_1" class="loop">
5       <mutator name="Loop.LoopInterchange"/>
6        <target device="cuda"/>
7         <target device="opencl"/>
8        </element>
9   </region>
10 </platform>
11 </xml>
```

Listing 5.4: A platform description file

```
1  double * a, * b, * c;
2  ...
3  #pragma llc context name("mxm") copy_in(a[n * l],b[l * m],c[m * n], l, m, n) copy_out(a[n *
       l])
4  {
5   int i, j, k;
6   #pragma llc for shared(a, b, c, l, m, n) private(i, j, k)
7   for (i = 0; i < l; i++)
8    for (j = 0; j < n; j++) {
9      a[i+j*l] = 0.0;
10     for (k = 0; k < m; k++)
11       a[i+j*l] = a[i+j*l] + b[i+k*l] * c[k+j*m];
12    }
13 }
14 ..
```

Listing 5.5: Matrix product in llcl

5.3.1 Implementing llcl

An instance of the Frangollo *Context* class is created for each llcl context clause. The Context name is set according to the name clause. This Context will last until the end of the enclosed block is reached. The copy_in and copy_out clauses are valid llcl clauses, similar to those used in the OpenMP extension. Their semantic is preserved, and they are used to specify variable constraints to the Frangollo runtime. Variables not passed on any of these clauses are assumed to be host-only variables. The size of the variable declaration is used by default. Users can specify a size expression in the Context definition.

The generation of SIMD kernels is performed when the omp for construct is replaced by its llc counterpart. llcl for takes the same clauses as omp for, and also supports an optional name clause to specify a particular kernel name.

The information extracted by the YaCF driver is used by the CUDABACKEND Kernelize (see 3.6.4). The Kernelize module will use information from the XML platform description file to apply platform-specific transformations. Figure 5.2 depicts an schema of the translation process performed by the YaCF driver.

Remarks

llcl was appealing from the point of view of exploring the directive-based approach of code generation. Referees from different conferences suggested to us that we should not add more languages to the (already divergent) landscape of directives for code generation. Lacking support from vendors or relevant research institutions it was hard to prove that our approach was appropriate. When we were about to release the first llcl implementation, the OpenACC standard was announced at the 2011 Super-Computing conference. We strongly believe that a standard in this field will greatly benefit the adoption of the approach, thus, we welcomed the announcement and adapted our research and work so that it would meet the standard. It is worth noting that the similarities between llcl programming model and OpenACC are notable, and to some extent, OpenACC can be seen as a subset of what we had intended for llcl.

5.4 accULL

As mentioned previously, the irruption of the OpenACC API swiftly moved us towards implementing this emerging standard [125, 123, 124]. With our previous work in llcl and the flexibility of our tools (YaCF and Frangollo) we managed to implement the first version of the standard in only four weeks.

We designed a YaCF driver (c2frangollo) capable of translating C code annotated with OpenACC directives and runtime calls to a Frangollo program capable of running on different platforms. The YaCF driver also automatically extracted OpenCL and CUDA kernels from the original source. The Frangollo C interface offers the low-level capabilities to implement OpenACC whereas the OpenACC runtime calls have been implemented directly on C++.

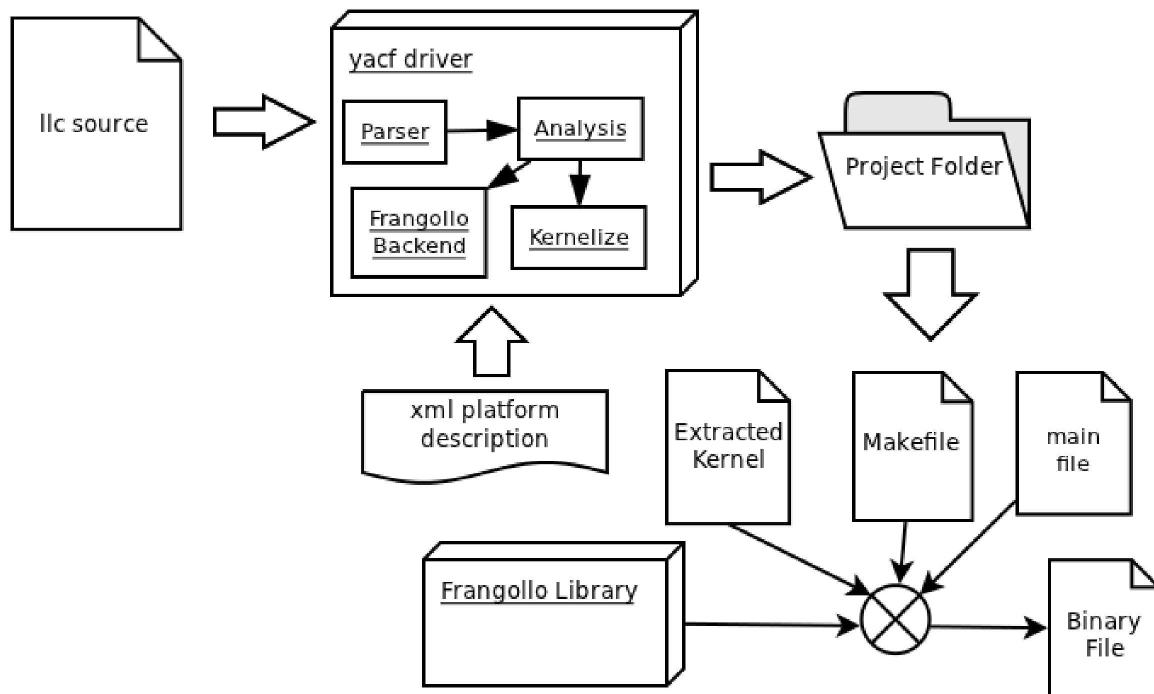


Figure 5.2: Compilation flow of an `llcl` source. The `YaCF` driver parses the file, and with the assistance of the XML platform description, applies the requested optimizations and extracts the kernel(s). The output is a project directory, containing the main file with `Frangollo` calls and with parallel constructs replaced by the equivalent `Frangollo` kernel launch statements. A separate file for each outlined kernel and a Makefile are also generated by the driver. When compiling against the `Frangollo` library, a binary file capable of running CUDA kernels is generated. Further optimizations can be applied to the outlined kernels without modifying the original file

The purposed behind the creation of our compiler was not to match the performance nor the stability of commercial compilers, instead we hoped that it could be used to explore the capabilities of the directive-based approach to heterogeneous computing. The flexibility of our tools enable us to explore new functionalities or platforms for OpenACC in a short period of time, and to provide recommendations to other implementors.

5.4.1 OpenACC Programming Interface

The OpenACC API offers developers a set of directives that provide simple hints to the compiler, helping it to identify areas of code that are suitable for being run on a hardware accelerator. The result is that developers are freed from the task of writing specific device code details. With the information provided by the developer annotations, data movement between accelerator and host memories and data caching are managed by the compiler or runtime.

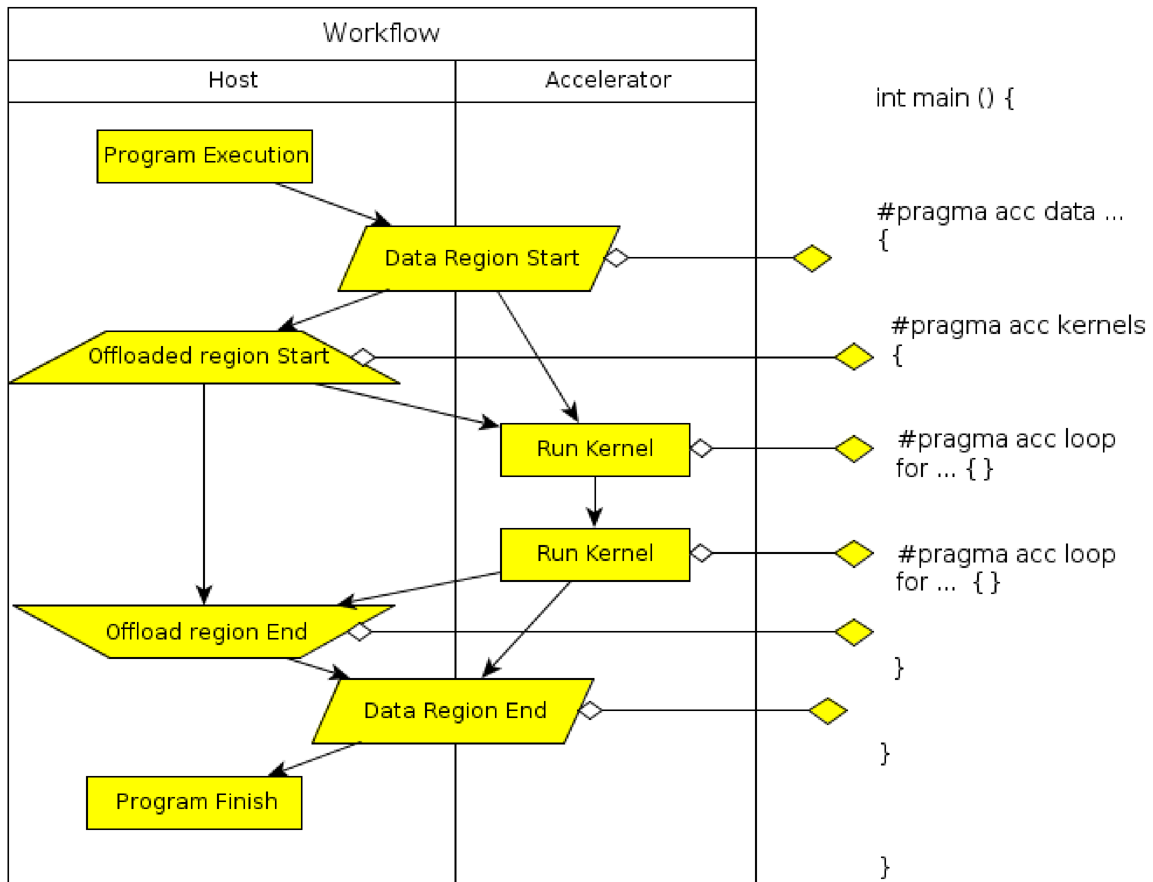


Figure 5.3: Execution workflow of a program using the OpenACC API

The execution model targeted by OpenACC is host-directed execution with an attached accelerator device, such as a GPU. Computationally intensive regions of the code are offloaded to the accelerator device. The devices execute *parallel regions* which usually contain work-sharing loops, or *kernel regions* which mostly contain one or more loops that are executed as kernels in the devices. Figure 5.3 illustrates this execution model.

Up to three different levels of parallelism are available on accelerators. Fine-grain parallelism is handled by multiple threads within a single execution unit. Coarse-grain parallelism is handled by running groups of multiple threads in different execution units. SIMD-like operations are also available on accelerators inside each thread, or by combining a set of threads. These three levels have to be properly mapped to the hardware in order to extract the maximum performance from the accelerator; directives have been provided to allow this. The programmer also has to take into account that synchronization at some levels is not possible.

The memory is modeled on the assumption that there are separate address spaces between the Host and the accelerator. This means that in order to compute anything inside the device, a memory transfer has to be performed. The developer has to be aware of this situation when writing OpenACC, and cannot ignore the fact that memory bandwidth and size limits may impede the automatic offloading of code segments.

The OpenACC API relies on directives implemented using the pragma mechanism to identify different region types inside the code. Three main directives can be identified: `data` to mark data regions inside the code; `kernels` to define groups of loop nests to be offloaded to the accelerator; and `parallel` which allows for more fine-grain control over the code to be offloaded.

Both `parallel` and `kernels` constructs may contain `loop` constructs. This construct precedes a loop or a loop nest and gives indications to the compiler on how the loop should be parallelized. It is possible to specify the number of threads (workers), group of threads (gangs) and the width of SIMD operations (vector), thus forcing the compiler to adapt the subsequent structured block and/or loop to the programmer's specifications (see Figure 5.4).

Data clauses may be used to reduce memory transfers to/from the device. Variables can be marked as `copy_in, copy_out, copy` or `create`. The implementation will be responsible for deciding whether these clauses are required or not. Compiler and/or runtime might decide to add additional variables to ensure correctness.

Asynchronous operations are supported via the `async` clause. Directives without the `async` clause have an implicit barrier at the end of their scope. The `async` accepts an optional scalar integer expression. This clause is used in combination with the `wait` directive to wait for a particular event to finish.

OpenACC also enables users to use native pointers to handle legacy code or library calls. The clause `deviceptr` indicates that the marked variable is a native device pointer (and not a host variable). The `host data` construct makes the address of the device available in the host, as shown in Listing 5.6. The code uses the runtime call `acc_get_device_type` to check if the device is NVIDIA CUDA. If it is a CUDA device, it uses the `host_data` construct to indicate that all the references to the variables `h_a`,`h_B` and `h_C` have to be replaced by its correspondent pointers in the device.

Any reader interested in detailed information regarding the language should check the detailed reference available on [106].

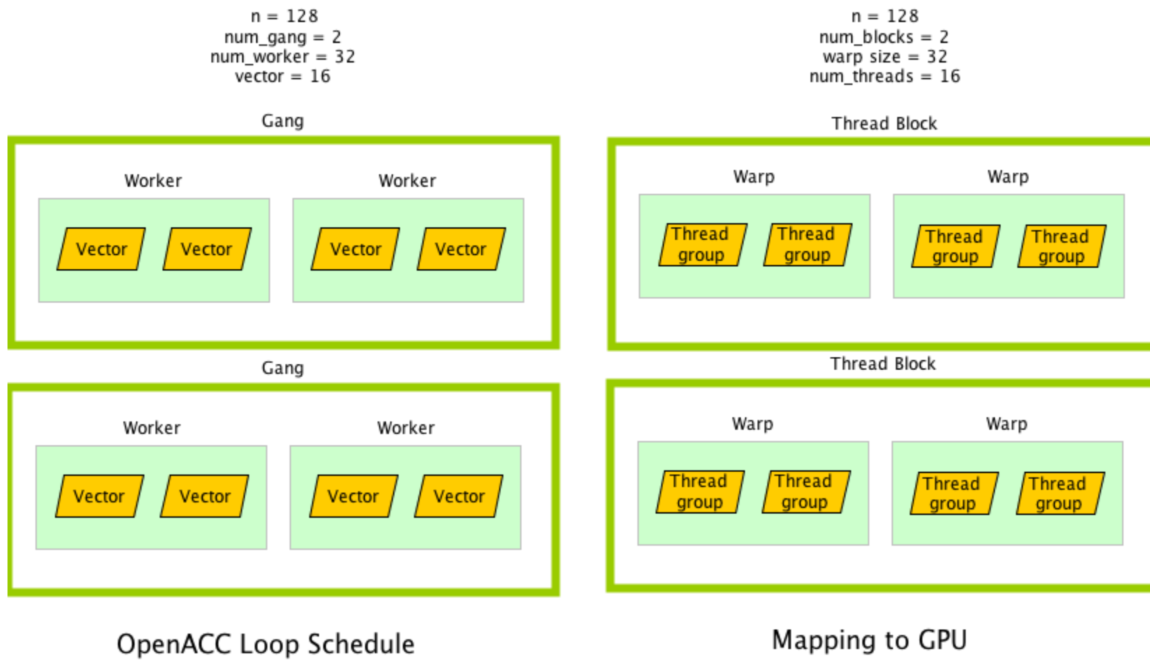


Figure 5.4: Relationship between OpenACC concepts and elements in the GPU architecture

```
...
#pragma acc data copy(h_A[n2], h_B[n2], h_C[n2]) {
 if (acc_get_device_type() == acc_device_nvidia) {
     #pragma acc host_data use_device(h_A, h_B, h_C)
     {
     /* Performs operation using cublas */
     status = cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, N, N, N, &alpha, h_A, N, h_B, N,
          &beta, h_C, N);
     if (status != CUBLAS_STATUS_SUCCESS) {
          fprintf (stderr, "!!!! kernel execution error.\n");
          return EXIT_FAILURE;
     }
    }
 }
}
...
```

Listing 5.6: Code example showcasing the usage of the `host_data` directive

5.4.2 `c2frangollo` Compiler Driver

The StS translation injects a set of `Frangollo` calls within the serial code. The compiler driver orchestrates the overall StS process. Apart from indicating the input file and setting the output directory, users can provide the usual set of parameters, such as the path to include files or additional macro definitions. The driver creates the translation environment and parses the input file, obtaining the IR-1 and the symbol table of the code. The driver then creates an instance of the *Frangollo.Runner* class and applies it to the obtained IR (providing the symbol table). The *apply* method then creates a set of *Mutators* that will look for OpenACC directives and generating the equivalent `Frangollo` API calls.

The first *Mutator* applied is *FM_AccKernel*. This class looks for potential kernel regions (annotated with `loop`) and extracts the necessary information to feed the `CUDA` and `OpenCL` back ends that will extract the Kernel from the source to a separate file for each kernel. This class also applies high-level optimizations, such as `loop invariant` or `loop skewing` to the kernel subtree before passing the information to the `Kernelize` module. A new class *AccParametrizeLoop* has been implemented, extending the original *ParametrizeLoop* to augment the loop information with the information extracted from the clauses (such as reduction operations and parameter directionality). It also extracts the original sequential code to a routine using the *Outliner* module so the sequential version is still available in the code. Retaining the original code in an external routine also allows us to implement the OpenACC `if` clause to enable conditional usage of the offloaded region.

The *FM_AccKernel* also creates a call to the `FRG_launchKernel` function. The parameter list is generated with all the parameters required to launch the kernel, independently from their type or where they are used. Information is added to each parameter regarding if it is read or not in the kernel. This information is validated according to the analysis of the compiler. Expressions extracted from the *Kernelize* module are passed to the runtime call that will use them to compute the intensity at runtime. Using expressions rather than compile-time computed-values enables the runtime to better approximate a value for the kernel grid execution.

The IR of the original subtree is modified as a result of applying this *Mutator* and calls are injected to the subtree. Kernel sources are returned by the *Mutator* and are included in the project source by the *Runner* class.

The *FM_AccData Mutators* look for any directive that could define a context, such as `kernel`, `parallel` or `data` and generating a `Frangollo` `FRG_createContext` call. The variables marked in the `copy` clauses are registered using `FRG_registerVar`. Copy clauses prepended with the `present` are also supported through a parameter to the `FRG_registerVar`. Minor modifications were required to support this feature, this is because internally the runtime uses the pointer address to detect if the variable was present or not. The end of the context-defining directives generate a `FRG_destroyContext` call.

Finally, *FM_AccHostData* replaces `host` directives with calls to `FRG_getDevicePtr` enabling support to use native pointers, and *FM_AccUpdate* replaces `update` clauses with calls to `FRG_update`.

5.4.3 Interfacing OpenACC in `Frangollo`

A new interface has been added to the `Frangollo` runtime. This interface adds support for OpenACC runtime calls together with some convenience functions required to facilitate the transformation.

A user wanting to use OpenACC runtime calls only needs to include the `openacc.h` file from the `Frangollo` distribution. The file will include all the required headers and definitions from the `Frangollo` runtime.

Constants `acc_device_host` and `acc_device_gpu` have been defined to differentiate whether the code runs on the Host or in the GPU using the `acc_device_type`. Also `acc_device_nvidia` and `acc_device_opencl` are defined so users can identify which platform is in use.

The OpenACC API call `acc_init()` creates a *Context* with a random name. `acc_shutdown()` simply destroys the current *Context*.

According to the OpenACC reference the aforementioned calls *may be used to allocate/deallocate memory on the accelerator device. Pointers assigned from this function may be used in deviceptr clauses to tell the compiler that the pointer target is resident on the accelerator*. To satisfy the definition while maintaining support for both CUDA and OpenCLplatforms `acc_malloc()` and `acc_free()` return a pointer to a pointer of the platform type. The `deviceptr` clauses can receive this double pointer [1].

5.4.4 Putting It All Together

The `YaCF` driver supports most of the syntactic constructs in the OpenACC 1.0 specification, but some of them are silently ignored. In addition, although some operations inside the runtime are handled asynchronously, support for the `async` OpenACC clause has not been implemented in the CUDA component yet, although it is available for OpenCL. Table 5.1 compares the compliance of our `accULL` with other OpenACC implementations.

From the description it is relatively straightforward to port a program written in OpenACC to a `Frangollo` program. The copy clauses can be used to define the constraints of the variable. The `data`, `kernels` or `parallel` construct forces the creation of a new `Frangollo` *Context* instance. Subsequent calls to these directives generate nested scopes through the new *Context* creation. Variables marked using the `declare` directive can be registered in the *Context*.

A distribution of the stable `accULL` version is available for download in [71].

[1] Current `deviceptr` implementation is buggy and thus it has not been marked as implemented in the summary.

Table 5.1: Compliance with the OpenACC 1.0 standard (directives)

Construct	Supported by	Notes
kernels	PGI, HMPP, accULL	(accULL) Kernels for OpenCL and CUDA are generated for each loop inside the scope
loop	PGI, HMPP, accULL	-
kernels loop	PGI, HMPP, accULL	-
parallel	PGI, HMPP	-
update	Implemented	(accULL) Mixing host and device clauses in the same device does not work, they must be in separate directives
copy, copyin, copyout, ...	PGI, HMPP, accULL	Runtime handles memory transfers dynamically
pcopy, pcopyin, pcopyout ,...	PGI, HMPP, accULL	Runtime handles memory transfers when required dynamically
async	PGI	(accULL) Only for OpenCL
deviceptr clause	PGI	-
host	accULL	Our framework generates the right code, but we still have to solve portability issues between OpenCL and CUDA
name	Not in standard (accULL only)	Optional clause to name a particular acc region or loop and refer it from an external optimization file at compile time.
private, firstprivate	PGI (private only), HMPP (private only), accULL	-
collapse	accULL	Generates a 2D (or 3D) kernel
gang, worker, vector, independent	PGI, HMPP, accULL	Different levels of support, check Section 6.3.5.1

Table 5.2: Compliance with the OpenACC 1.0 standard (API)

API Call	Supported by	Notes
acc_init	PGI, HMPP, accULL	Initialises runtime
acc_set_device	PGI, HMPP, accULL(no effect)	accULL uses the environment variable
acc_get_device	PGI, HMPP, accULL	-

CHAPTER 6

Performance results

In this Chapter we present experimental results to support our research. Section 6.1 details the hardware and software characteristics of the platforms which have been used. Other compilers used with support for directive-based programming are detailed in 6.2. To corroborate the suitability of our approaches we have explored different source codes from two different benchmark suites. Given our limited time and resources it was unrealistic to try to validate our thesis using real commercial and production codes. However, this was not a significant setback as our intention is not to build fully operative tools but to research them, thus, designing and implementing prototypes fulfills our goal. Nevertheless, we believe that the codes we have selected do actually cover most of the characteristics that real applications feature, thus enabling us to make realistic assumtions about the the performance and productivy of real codes.

In Section 6.3 we present the results from three different approaches, each using codes from the OpenMP source code repository (`OmpSCR`). The `OmpSCR` [53] features a set of codes implemented in OpenMP that we have ported to our different directive-based approaches. This set of codes represents situations or algorithms that appear in production codes, but they are implemented in a relatively straightforward way, thus facilitating the task of implementing compiler translations.

During the final stages of our research, whilst working with `accULL`, we realised that the compiler was mature enough to tackle more complex problems. In addition, splitting the complexity of the programming model into a compiler and a runtime greatly facilitated the task of supporting more complex codes. Situations that are traditionally difficult to solve at compile time, such as interprocedural analysis, are relatively simple to solve at runtime. At the same time, issues that are hard to solve at runtime, like the extraction of kernels for devices, can be solved at compile time. These reasons encouraged us to go that step further and to increase the complexity of the codes we were working on. In Section 6.4 we present the computational results for a selection of codes from the Rodinia Benchmark Suite [38, 39]. As OpenACC is a standard the task of porting this selection of codes from the Rodinia Benchmark Suite to OpenACC enabled us to compare the performance of different commercial implementations. We also evaluated other directive-based approaches, such as hiCUDA and the PGI Accelerator Model and compared their programmability and performance with OpenACC.

6.1 Experimental Platforms

In this section we present a brief description of the different platforms on which research was carried out. (For details on GPU architecture and capabilities, see Chapter 2.)
Tajinaste is a distributed-memory cluster based on AMD Opteron processors. It has seven IBM Blades LS21 and an IBM Blade LS20 installed on an IBM Bladecenter H system. The LS20 blade acts as a front-end node, whereas the LS21 are the compute nodes. The LS21 nodes have a two-way motherboard with AMD Quad Core processors, making a total of eight cores per node. The LS21 nodes are connected through an Infiniband 4x network. The MPI implementation used when running the experiments was OpenMPI 1.2, and the compiler used was GCC 4.1

Verode has four LS22 compute nodes featuring two-way motherboards with AMD Quad Core processors. Blades are connected to each other using a Gigabit Ethernet network with a dedicated switch for message passing. Two of the four LS22 nodes are connected to Tesla C1060 cards. OpenMPI 1.2 and GCC 4.1 were used in the experiments performed in this machine.

Ilion is a desktop computer with an Intel(R) Core(TM)2 Quad CPU Q6600 at 2.40GHz. It has 4MB of L3 Cache. The machine features a GeForce 9800 GT with 500MB of RAM. The OpenMPI implementation used was OpenMPI 1.4 whereas the compiler was GCC 4.2.

Peco is a cluster of four nodes interconnected using an Infiniband QDR network. Each node contains two Intel Xeon 5520 (Nehalem) QuadCore processors running at 1.27 GHz, with 24 GB of DDR2 RAM memory. Attached to the PCIExpress 2.0 bus of each node, there is a NVIDIA C1060 GPU with 4 GB of DDR3 RAM memory. *Peco* features a typical cluster infrastructure. Nowadays clusters are composed by multi-core processors and GPU devices, thus it is possible to take advantage of OpenACC on these platforms. CPU time in *Peco* has been generously donated by the *Universitat Jaume I* of Castellon.

Zape is a single node system using an AMD 9550 (Phenom) QuadCore at 2.2GHz connected to an NVIDIA GTX480 (Fermi) GPU. CPU time has been generously donated by the *Universitat Jaume I* of Castellon.

Garoe Desktop computer with an Intel(R) Core(TM) i7 930 processor (2.80 GHz), with 1MB of L2 cache, 8MB of L3 cache, shared by the four cores. The machine has 4 GB RAM and two GPU devices are attached, a Tesla C11060 (240 MP, 3 GB) and a Tesla C2050 (448 MP, 4 GB). With *Garoe* we mimic the typical scenario of an OpenACC developer: a moderately experienced user interested in improving the performance a scientific code can purchase a new GPU card and plug in it into his/her desktop computer. It is a relatively cheap platform as opposed to a multi-node cluster and could achieve a combined peak theoretical performance of 478.36 GFLOPs in double performance (77.76 GFLOPs from Tesla C1060 + 345.6 GFLOPs from Tesla C2050 + 55 GFLOPs from main processor). This kind of user might have some insight into programming and even GPU computing, but s/he is not an expert. Starting with his/her own serial code and using an OpenACC compliant compiler, this user will take advantage of the GPUs without investing excessive time in low-level programming.

Homero is a laptop computer with one Intel(R) Core(TM) i3 CPU M 350, using Hyper-threading to enable four virtual processors, 3GB RAM, and an integrated NVIDIA Optimus graphic card. *Homero* represents a typical modern medium-end laptop computer. It uses reduced versions of desktop GPUs that support GPGPU computing.

We have included a laptop in our evaluation because we want to explore other scenarios, not just HPC, and identify whether they too can benefit from heterogeneous computing tools.

Drago is a shared memory system, with 4 Intel(R) Xeon(R) E7 4850 CPU, with 2.50MB L2 cache and 24MB L3 cache (for all its 10 cores). 6GB of memory are available per core. *Drago* showcases an alternative platform that might take advantage of OpenCL. Nowadays shared memory machines feature several CPUs with many cores on each. These cores also contain vector processing units that require particular compiler support (or a deep understanding of these technologies) to unleash their potential. There are implementations of OpenCL, such as the Intel OpenCL SDK or the AMD APP SDK, targeting these shared memory machines. Implementing algorithms in OpenCL requires effort, but it allows a better mapping of hardware resources and improves thread scheduling. Using CPU-targeted OpenCL platforms along with OpenACC represents an interesting alternative to traditional OpenMP programming that we will explore in different examples with the `ACC` approach.

6.2 Other Compilers Implementing Directive-based Approaches

Some codes have been ported to other directive-based programming models: hiCUDA and PGI Accelerator Model. For details about these alternative PMs see Section 2.1.4. For the experiments we detail in this Chapter we used hiCUDA version 0.9, and we used version 12.2 of the PGI Compiler to explore the PGI Accelerator Model.

When other compilers released OpenACC implementations we decided to compare our `accULL` OpenACC implementation against theirs. All experiments comparing the performance of OpenACC of the different implementations use version 12.5 of the PGI compiler and the CAPS HMPP compiler version 3.2.3.

Important note: The version of the `accULL` compiler used in the experiments to compare the performance of older accelerator models was an earlier version, one of the early internal releases. However, the version of `accULL` used for the OpenACC comparison is version 0.1: release date - October 2012.

6.3 OpenMP Source Code Repository

The motivation behind the creation of the OpenMP Source Code Repository (`OmpSCR`) [53] was to provide the OpenMP community with an infrastructure which would enable the evaluation of performance across different platforms and compilers. By sharing the OpenMP implementation of an heterogeneous set of codes, the intention was to provide other researchers with a tool that could be used to communicate their results faster and more efficiently. All `OmpSCR` applications are implemented conform to standard programming languages (ISO C99 and FORTRAN) and are relatively simple (around 5000 lines of codes at most).

We have selected the following codes from this repository: Molecular Dynamics (MD), Mandelbrot Set Computation (Mandelbrot), LU reduction (LUred), the solution of a finite difference discretization of Helmholtz equation using a Jacobi iterative method (Jacobi) and a simple Matrix Multiplication (MxM).

In this section we describe the algorithm for each of the codes, this is accompanied by a sketch of the OpenMP implementation. Following this, we then study particular details of the different directive-based approaches being explored. To facilitate the reading of the section have followed the convention detailed below that is used to identify the approach for each particular analysis:

- ACC: The study has been carried out using accULL (Section 5.4).
- LLC-CUDA: The study has been carried out using the OpenMP extensions (Section 5.2).
- LLC-HYB: The study has been carried out using llCoMP with MPI/OpenMP support (Section 5.1).

6.3.1 Mandelbrot Set Computation

The Mandelbrot Set [90] is the convergence domain of the complex series defined by $Z_n = {Z_{n-1}}^2 + C$. The area of the set is an open question in Mathematics. We have selected an algorithm that estimates the set area using a Monte Carlo method, which uses the number of points of the plane to compute as its input. Listing 6.1 shows the main loop of the OpenMP implementation. This loop computes the number of points that are outside the set, and then it computes the number of internal points, the area of the set and the error margin. Each iteration correspond to a check for a point in the plane.

```
1  numoutside = 0;
2  #pragma omp parallel for default(none) \
3            reduction(+:numoutside)     \
4              private(i,j,ztemp,myid,nlocal, \
5              ilower,z,iupper),shared(nt,c)
6  for(i = 0; i < npoints; i++) {
7    z.creal = c[i].creal;
8    z.cimag = c[i].cimag;
9    for (j = 0; j < MAXITER; j++) {
10     ztemp = (z.creal*z.creal)-(z.cimag*z.cimag)+c[i].creal;
11     z.cimag = z.creal * z.cimag * 2 + c[i].cimag;
12     z.creal = ztemp;
13     if (z.creal*z.creal+z.cimag*z.cimag > THRESOLD) {
14       numoutside++;
15       break;
16     }
17   } /* for j */
18 } /* for i */
19 numinside = npoints - numoutside;
20 /* Calculate area and error */
21 area = 2.0 * 2.5 * 1.125 * numinside / npoints;
22 error = area / sqrt(npoints);
```

Listing 6.1: OpenMP implementation of the Mandelbrot set computation

The `break` statement inside the loop causes that some iterations may affect some iterations making them require more time than others, thus, creating an potential unbalance on the OpenMP implementation.

6.3.1.1 Hybrid MPI/OpenMP implementation with `llCoMP` (`LLC-HYB`)

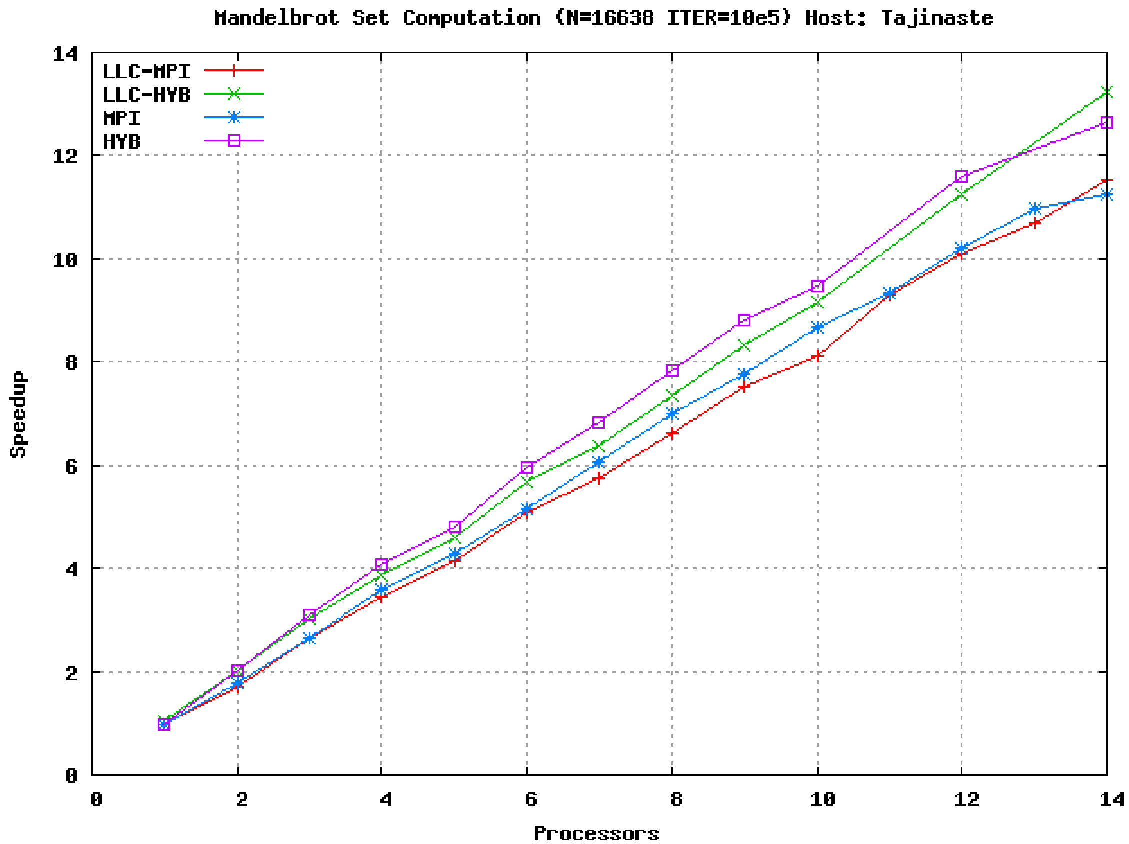


Figure 6.1: Speed-up of the Mandelbrot set computation on *Tajinaste*. `HYB` represents an ad-hoc hybrid implementation, `LLC-MPI` stands for pure MPI code generated by `llCoMP`, `LLC-HYB` correspond to the hybrid code generated by our compiler and finally `MPI` stands for a handwritten MPI implementation

Figure 6.1 shows the results for four different implementations of the Mandelbrot implementation running on *Tajinaste*. In every case the code iterates over 16368 points in the complex plane. This algorithm is usually taken as an academic example of a situation with an intense load imbalance: each complex point analyzed requires a large-varying number of iterations of the main loop to converge. In this situation, Figure 6.1 reflects the benefit of the *LLC-HYB* approach in relation to the *LLC-MPI* case. The parallel execution time improvement is 17% percent for 14 processors. The `llc` versions produce results comparable to their *ad-hoc* counterparts while retaining a lower amount of coding effort.

```
1  #pragma omp target device(cuda) copy_in(c)
2  #pragma omp parallel for reduction(+:numoutside) private(i,j,ztemp,z) shared(nt,c)
3  {
4    numoutside = 0;
5    for(i = 0; i < npoints; i++) {
6      z.creal = c[i].creal;
7      z.cimag = c[i].cimag;
8      for (j = 0; j < MAXITER; j++) {
9        ztemp = (z.creal * z.creal) - (z.cimag * z.cimag) + c[i].creal;
10       z.cimag = z.creal * z.cimag * 2 + c[i].cimag;
11       z.creal = ztemp;
12       if (z.creal * z.creal + z.cimag * z.cimag > THRESOLD) {
13         numoutside++;
14         break;
15       }
16     } /* for j */
17   } /* for i */
18 }
```

Listing 6.2: Using the OpenMP extensions to port a code to CUDA

6.3.1.2 Extending OpenMP to support heterogeneous architectures (LLC-CUDA)

Listing 6.2 illustrates how the extensions for heterogeneous architectures [17] are used in a real code. In Line 1 we specify the target device for the parallel loop in Line 5. When llCoMP translates to CUDA, it looks for parallel regions preceded by an omp target directive (line 1) whose device is CUDA. Once this situation is detected, the compiler inserts the memory transfer pattern into the code and encapsulates the body of any parallel loop into a CUDA kernel. Finally, the patterns for data gathering and resources deallocation are also inserted.

The CUDA back end in llCoMP uses a specialized kernel to perform reduction operations. The kernel implemented in the compiler [69] uses interleaved addressing and makes the first add during data fetching from global memory. This improvement is achieved by using the device to perfor the reduction, thus minimizing the size of the transfer between Host and device.

6.3.1.3 Analysis of the kernel grid configuration (LLC-CUDA)

An important issue that has a large impact on the performance of CUDA programs is the number of thread per block, particularly in the presence of irregular computations. Figure 6.2 shows the execution time of the extended OpenMP implementation in three different architectures.

It is important to note that increasing multiprocessor occupancy may be a better way to get performance from a given board rather than using a higher number of threads. Although it seems counterintuitive, in some CUDA compute versions, the best performance for a problem size may be achieved with a lower number of threads per block. With a lower number of threads, a larger number of blocks can be allocated into the same multiprocessor, thus more blocks can run concurrently on different multiprocessors.

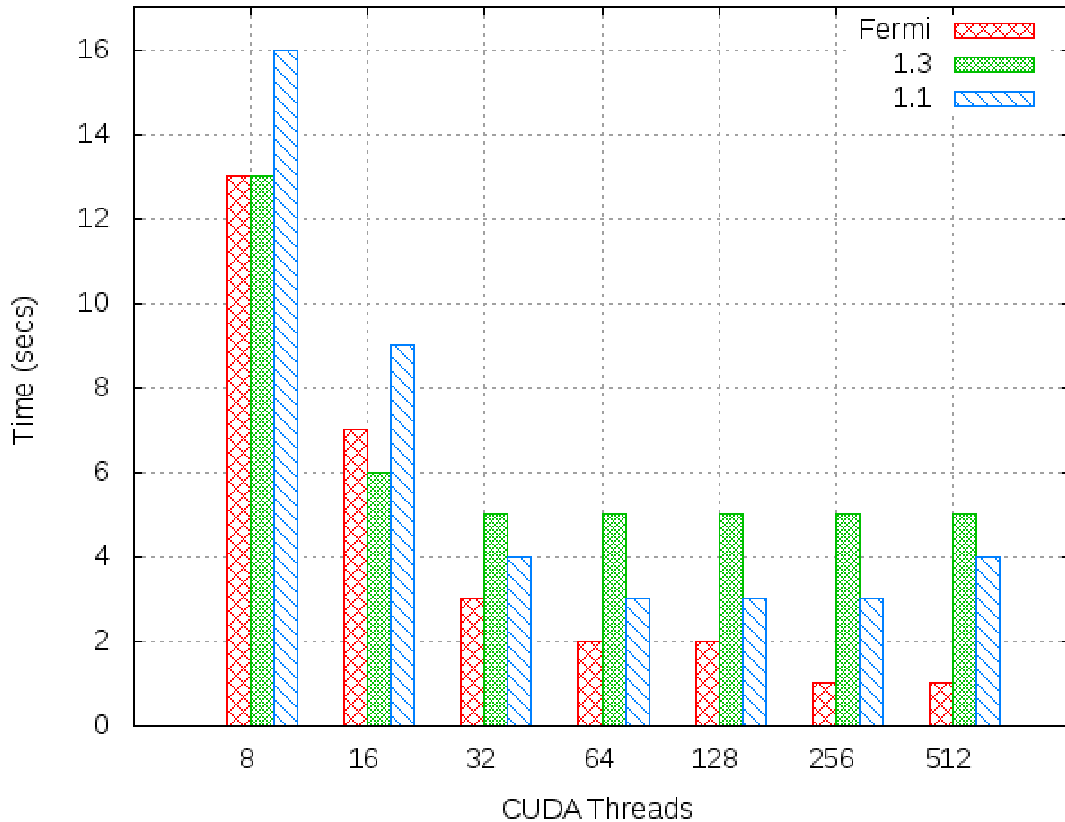


Figure 6.2: Time execution comparison of different numbers of threads in three architectures, using the Mandelbrot set computation implemented in `llc` in *Garoe*. An incorrect choice in the kernel configuration may hinder performance. Each architecture has its own optimal kernel launch configuration

6.3.1.4 The OpenACC implementation of Mandelbrot (`ACC`)

To illustrate the differences with previous approaches Listing 6.3 shows the OpenACC implementation of the Mandelbrot code. When creating the parameters for the kernel launch, `YaCF` indicates to the runtime that the `numoutside` parameter requires a reduction operation and expands the scalar variable to a vector. This vector stores a private copy of the variable in each thread. Later, both CUDA and OpenCL components of the runtime, using a separated and optimized kernel, perform the reduction operation. The reduction operation is not performed during kernel execution, but later on when the variable is transferred back to the device, or if the variable were required by another kernel.

Figure 6.3 shows the execution time of the Mandelbrot code and the effect of varying the number of threads across different implementations.

```
1  #pragma acc kernels loop reduction(+:numoutside) private(i,j) copyin(npoints, c[0:npoints])
       copy(numoutside)
2    for(i = 0; i < npoints; i++) {
3      z.creal = c[i].creal; z.cimag = c[i].cimag;
4      for (j = 0; j < MAXITER; j++) {
5        ....
6        if (z is outside set) {
7          numoutside++;
8          break;
9        }
10     } /* for j */
11   } /* for i */
```

Listing 6.3: The Mandelbrot set computation in OpenACC

6.3.2 Molecular Dynamics

The Molecular Dynamics (MD) code is an implementation of the velocity Verlet algorithm [135] for Molecular Dynamics simulations. It employs an iterative numerical procedure to obtain an approximate solution whose accuracy is determined by the time step of the simulation.

The pseudo code shown in Listing 6.4 illustrates the global structure of the MD implementation. After an initial forces computation, the algorithm performs two basic operations on each simulation step: *compute* (C) and *update* (U). C operation consists of several nested loops computing the forces for each position. An external loop iterates over all particles computing their forces in the current simulation step. This requires the distance among all other particles to be computed and hence access to the position matrix; the total potential and kinetic energy of the system is computed, requiring access to the velocity matrix. In terms of the data access pattern, the code is highly un-coalesced, requiring several non-contiguous loads to compute each particle. In addition, it features several costly double precision operations ($sqrt$, sin and cos) which traditionally perform badly on GPU devices. The U operation is simply a `for` loop that runs over the particles, updating their positions, velocities and accelerations.

6.3.2.1 Evaluation of performance with Hybrid MPI/OpenMP `llCoMP` (`LLC-HYB`)

Number of cores	MPI	Hybrid MPI+OpenMP
1	0.984	0.950
2	1.967	1.903
3	2.948	2.849
4	3.929	3.776
6	5.813	5.600
8	7.637	7.316
9	7.558	8.533
12	9.715	11.303

Table 6.1: Speedup for the MD algorithm in Tajinaste

```
int main(...) {
  ...
  // Initial energy calculation
  compute(position, velocity, mass, force, &potential, &kinetic);
  ...
  // (S) Simulation
  for (i = 0; i < NSTEPS; i++) {
    compute(position, velocity, mass, force, &potential, &kinetic);
    printf(..., potential, kinetic);
    update (position, velocity, mass, force, &potential, &kinetic);
  }
  ...
}
void compute(...) {
    // (C) Compute forces
    for (...) {
    }
}
void update(...) {
    // (U) Update velocity/position
    for (...)
     for (...) {
        ...
    }
}
```

Listing 6.4: Sketch of the MD simulation

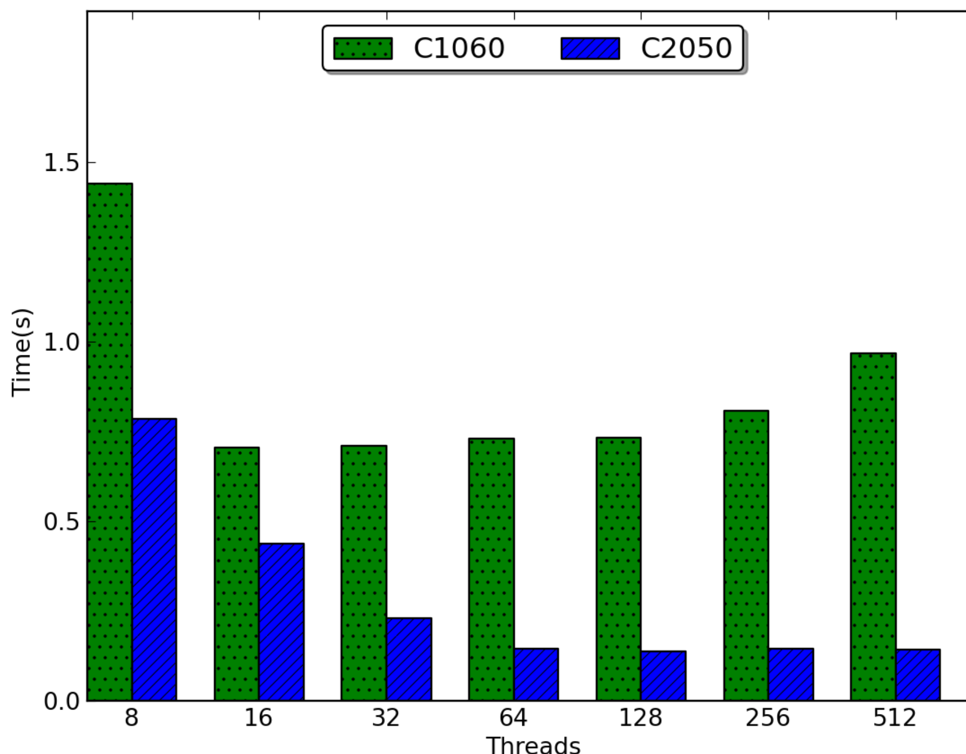


Figure 6.3: Execution time of the implementation greatly varies in terms of the number of threads, using $N = 32768$ points. In addition, the optimal number of threads varies from Tesla C1060 to Tesla C2050. This clearly reflects the significance of a proper estimation of the kernel launch configuration

Table 6.1 shows the speedup over sequential achieved by the MD implementation with the hybrid `llc` compiler. We limited the number of codes to 12 to guarantee exclusivity on the usage of the nodes. Although the hybrid approach does not produce substancial benefits with a lower number of nodes, when using a higher number of nodes we observe a performance increase. This make us believe that the hybrid approach would provide better performance using a larger number of fat nodes.

6.3.2.2 Mixing OpenMP and CUDA with the OpenMP to CUDA translator (`LLC-CUDA`)

This implementation of the MD algorithm enables us to study the best combination of GPU/CPU to target the parallel code. Let us denote the CPU by C and GPU by G. We measured four different versions of the code:

CC: both routines in the CPU (pure OpenMP code).
GG: both routines in the GPU (pure CUDA code).

```
void compute(int np, int nd, double *box, vnd_t *pos, ...) {
  double x, d, pot, kin;
  int i, j, k;
  vnd_t rij;

  pot = kin = 0.0;
  #pragma omp parallel for default(shared)
              private(i, j, k, rij, d) reduction(+ : pot, kin)
    for (i = 0; i < np; i++) { /* Pot. energy and forces */
      for (j = 0; j < nd; j++)
        f[i][j] = 0.0;
      for (j = 0; j < np; j++) {
        if (i != j) {
          d = dist(nd, box, pos[i], pos[j], rij);
          pot = pot + 0.5 * v(d);
          for (k = 0; k < nd; k++) {
            f[i][k] = f[i][k] - rij[k] * dv(d) /d;
          }
        }
      }
      kin = kin + dotr8(nd, vel[i], vel[i]); /* kin. energy */
    }
    kin = kin * 0.5 * mass;
    *pot_p = pot;
    *kin_p = kin;
  }
```

Listing 6.5: Main compute loop of the MD code simulation in OpenMP

GC: *C* in the GPU and *U* in the CPU.

CG: *C* in the CPU and *U* in the GPU.

Figure 6.4 shows the speedup obtained for three different problem sizes (number of particles). The best performance is obtained when both routines are placed in the GPU. For the hybrid OpenMP/CUDA codes, the best choice is to allocate the coarser grain routine to the GPU. The pure OpenMP version of the code does not scale up when the problem size is increased due to memory constrains.

6.3.2.3 Effects of Loop Optimization Techniques Across Different Architectures (LLC-CUDA)

A wide variety of loop optimizations can be applied to the MD implementation. Using our OpenMP extensions we can apply different optimizations.

First, loop unrolling can be applied to the loop in Line 16 of the *compute* function. Loop unrolling seems to improve performance on some algorithms [83], increasing register usage and avoiding branch instructions which tend to degrade performance in CUDA.

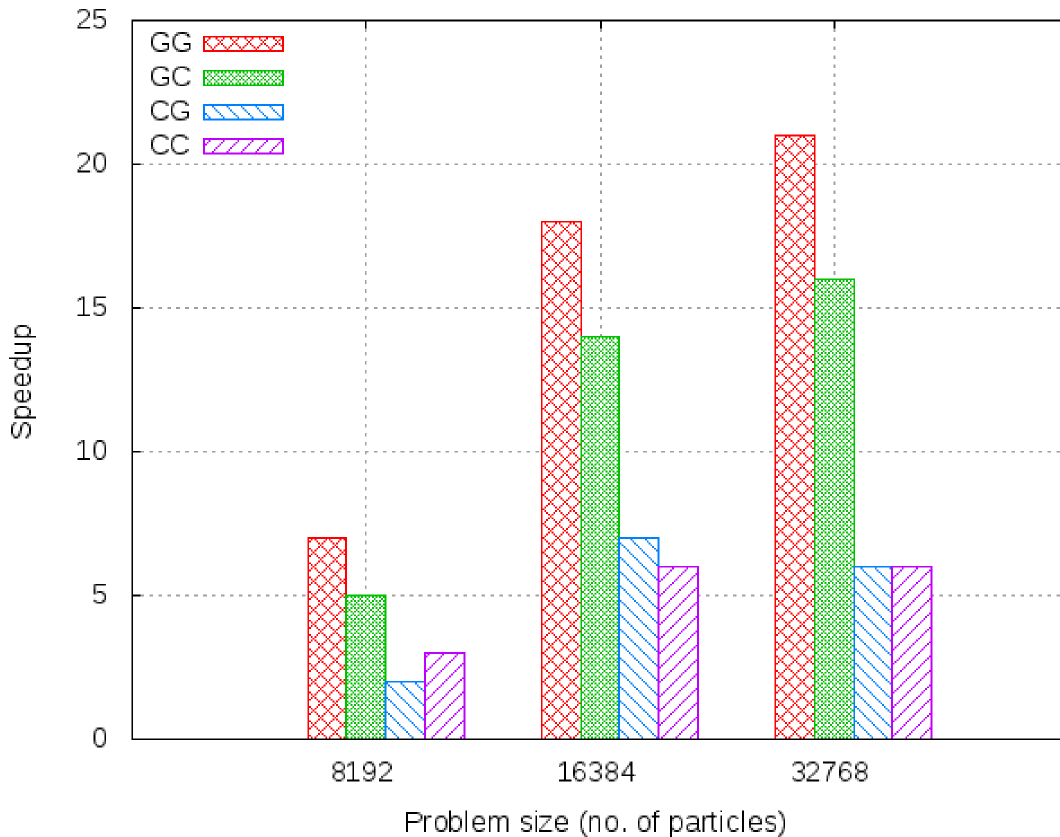


Figure 6.4: Speedup of the MD simulation code for different parallelization strategies, using Tesla C1060 and eight OpenMP threads in *Garoe*, one on each processor core

Applying loop collapse to the `update` function is also a possibility. The `llCoMP` to CUDA compiler converts iterations from the two nested loops into threads for a two dimensional grid. Despite the fact that each thread only computes one iteration, instead of computing all the iterations of the inner loop when the collapse is not enabled, we manage to reduce the number of memory accesses per thread. This reduces the possibility of bank conflicts and increases the number of cache hits.

The results for both optimization techniques are shown in Figure 6.5. The best kernel launch configurations were chosen for each architecture. In 1.1, the best fit for loop collapsing was 4x4 threads, maximizing multiprocessor occupancy. In 1.3, neither loop unrolling nor loop collapsing were useful, due to excessive register usage. 2.0 architecture revision benefits from both unrolling and collapsing, but the gains are negligible.

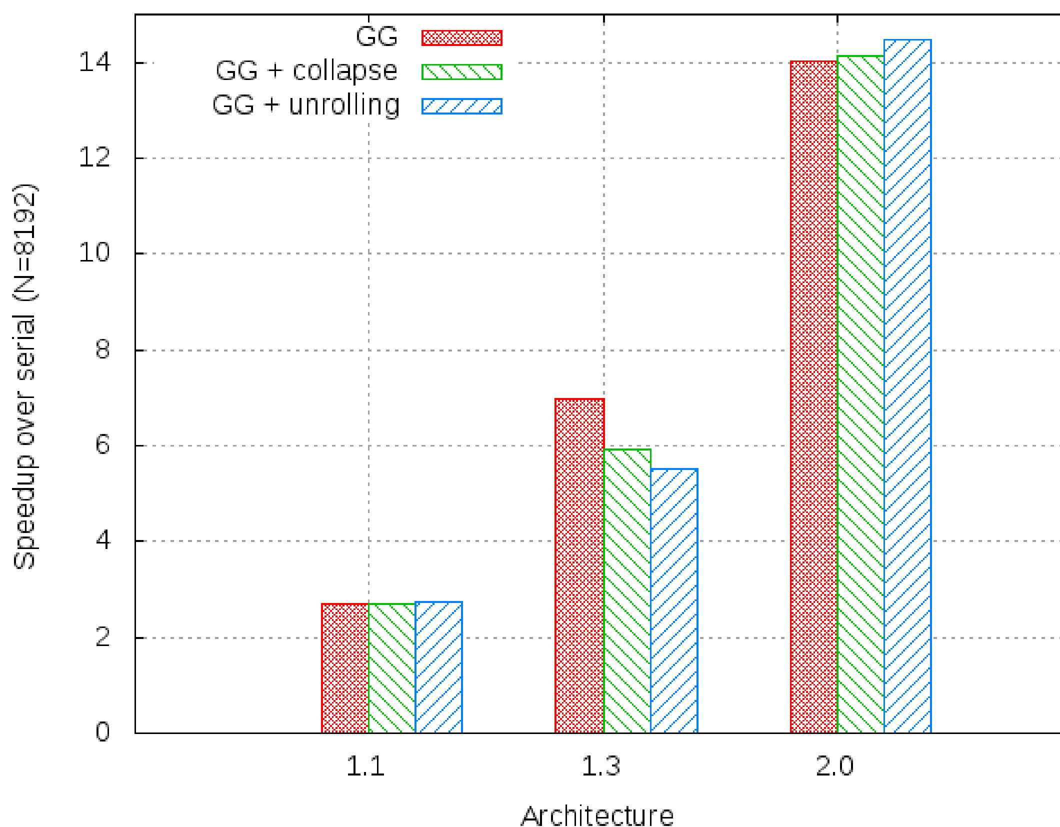


Figure 6.5: Performance comparison of baseline, unrolling and collapsing in the evaluated platforms using *Garoe*

6.3.2.4 Advantages of the Runtime Approach: Reduction of Memory Transfers (ACC)

A naive porting MD using OpenACC directives would consist of into adding the `kernels loop` construct to the top of the outermost loops in both routines (before *compute* and *update* in Listing 6.4), and writing the appropriate copy clauses to indicate variable directionality related to the loop.

In this case, our compiler would extract the kernel from the loops and inject the appropriate runtime calls. It is up to the runtime when the memory transfers are performed as long as it satisfies Condition 1 (see Section 4.1). Transfer time between Host and GPU could represent a significant percentage of the total time. Developers should take into account that the outstanding performance achieved by accelerator devices can be easily hidden by an excessive memory transfer time. We highly recommend using profiling tools to detect bottlenecks.

Version	Time transfer in	Time transfer out	Kernel Time	Total time	% Speedup
Naive Approach	$< 0.02s$	$0.0127834s$	$5.69122s$	$5.791388s$	-
Using a data clause	$< 0.02s$	$0.0121639s$	$5.63023s$	$5.729317s$	1%
Splitting *C* loops	$< 0.02s$	$0.0120155s$	$3.87633s$	4.046456	30.1%

Table 6.2: Time per phase and speedup for each incremental optimization over the naive implementation, as measured in *Homero* using the Intel OpenCL SDK over the CPU. In this situation, using the `data` clause does not represent an important performance benefit, due to the fact that (1) Frangollo OpenCL implementation uses the native pointer whenever possible and (2) the Intel OpenCL features lower initialization time than GPU approaches

The nested loop within the *update* function can be further optimized by using the OpenACC `collapse` clause. This clause instructs the compiler to generate a N-dimensional kernel, where N is the parameter of the clause. The developer is responsible for ensuring that both loops can be executed in parallel. As stated in previous examples, a two-dimensional nested loop may improve memory coalescence, and benefit memory-bound kernels.

OpenACC features a `data` directive that enables a data region to be created in which the information of the indicated variables is transferred into the GPU, and back to the Host at the end of the data region. `accULL` creates a context at this point, and the directionality information provided through the copy clauses is used to register the variables in the runtime. In this case, we precede the `simulation` loop with the aforementioned `data` construct, indicating that the parameters `force`, `position`, `velocity` and `acceleration` can be transferred into the device at this point. From now on, all references to these variables inside a kernel will not require a memory transfer from the Host, as they are all already stored on the device. When entering kernels inside `compute` and `update` functions, the runtime will not create a new context, instead it creates a new scope level within the existent context. Using this mechanism we ensure that variables and directionality of higher scopes are preserved. However, new variables might be added to these nested constructs. For details on the scoping rules see Section 4.1. In the MD code example, we require the variables `pot` and `kin` to be transferred in/out between iterations in order to show the appropriate information to the user. However, as both `pot` and `kin` are registered within an inner scope, whenever these inner scopes are exhausted, variables are transferred back from the device to the Host.

`accULL` enables users to perform incremental parallelization over GPU devices with minor effort. Traditional GPU performance tools can be used with the resulting codes. For example, in the MD code, the NVIDIA profiler [104] shows that more than 80% of the time is devoted to the `compute` kernel. As stated before, this kernel is highly compute-intensive as it features un-coalesced memory accesses and costly non-parallel floating-point operations. One possible solution is to split this kernel into several smaller ones thereby increasing coalescence. This could be considered counterintuitive in traditional CPU programming (where processor features large caches), but in GPUs it is a good idea. In order to rewrite this kernel into smaller ones, a CUDA developer would have to make a considerable effort as s/he would be forced to write additional kernel calls, memory transfers, etc. In OpenACC, the programmer only needs to split the sequential code and put the appropriate directives on the new loops: the compiler will extract the required kernels.

6.3.2.5 OpenACC Beyond GPUs (ACC)

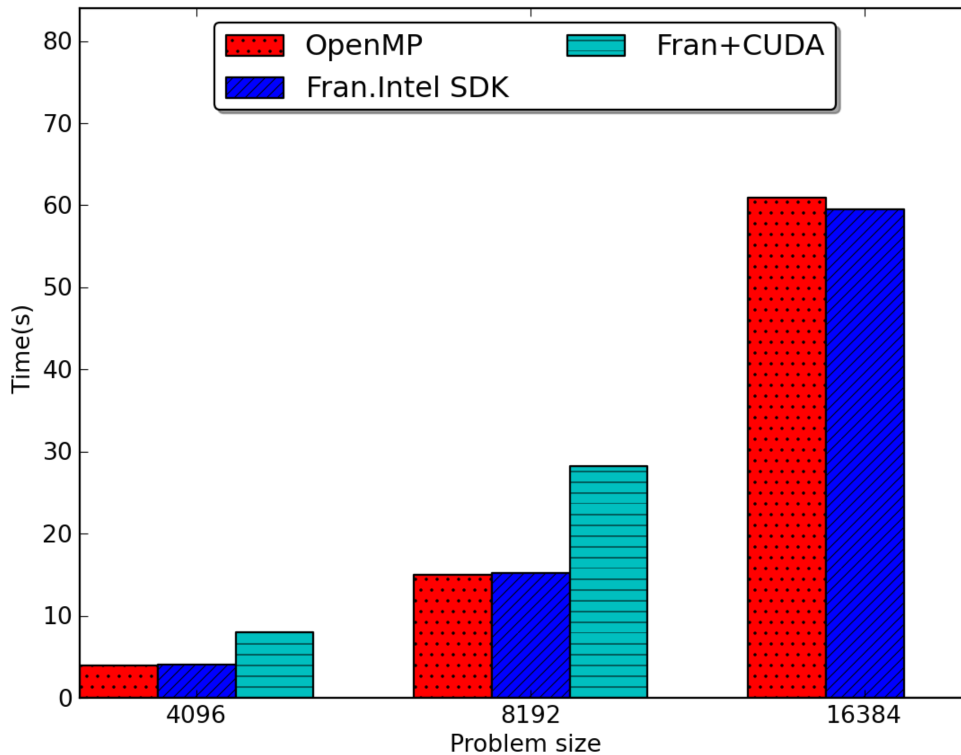


Figure 6.6: Performance comparison of the best OpenACC implementation vs. OpenMP in *Homero* for three different problem sizes. The laptop GPU memory was not large enough to handle the size of the biggest problem and the laptop crashed

accULL provides the means to execute our codes on different platforms outside of typical HPC architectures. Performance figures for the low-end system *Homero* are shown in Figure 6.6. In contrast, Figure 6.7 shows the benefit of using an OpenCL implementation using an high-end shared memory multiprocessor (*Drago*) with no GPU device attached.

Tables 6.2 and 6.3 show detailed timing information for transfers, kernel and total time obtained using Frangollo's internal tracing module. In both Tables, the problem size was 4096 particles and 20 iteration steps were performed. Results were validated against the sequential implementation. Users can turn this tracing feature when building the runtime and produce these statistics through an internal Frangollo call.

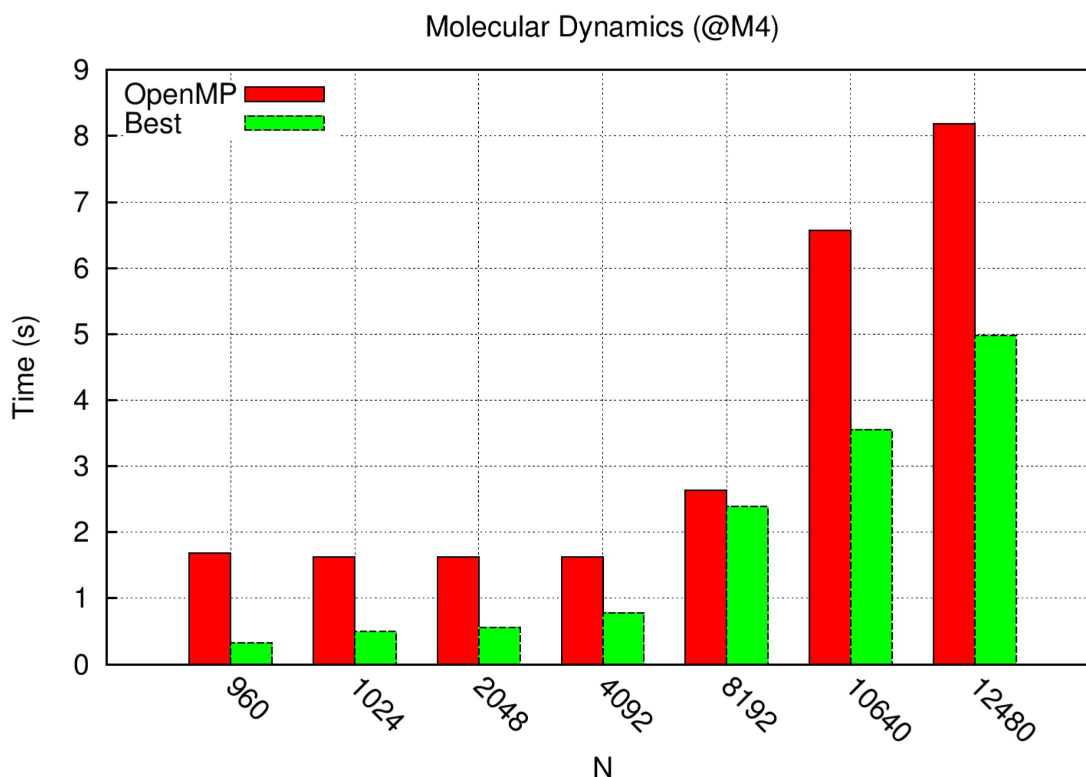


Figure 6.7: Time execution comparison for the MD OpenACC implementation against OpenMP in *Drago*

6.3.3 Jacobi

Jacobi is a program for solving a finite difference discretization of the Helmholtz equation $(d_2/dx_2)\,u + (d_2/dy_2)\,u - \alpha u = f$ using the Jacobi iterative method. This code features a typical *stencil kernel* [127]. *Stencil kernels* are a class of iterative kernels which update array elements according to a fixed pattern, called a stencil. These kind of codes are commonly found in all kinds of computer simulations or, as in this case, are used to solve partial differential equations. The OpenMP implementation consists of two parallel regions with one parallel loop each. Loops are parallelized using the default `static` scheduling. The main computational loop is shown in Listing 6.6.

6.3.3.1 Reducing Memory Transfers from/to the GPU (`LLC-CUDA`)

The code in Listing 6.7 shows the OpenMP extensions in action. In Line 3, we specify the target device for the parallel loops in Lines 6 and 10. In order to use the CUDA device, the programmer need only specify to specify the `target` directive. The `copy_in` and `copy_out` clauses in the directive at Line 3 state the memory positions to be transferred to and from the device. Figure 6.8

```
while (k <= maxit && error > tol) {
   error = 0.0;
   /* copy new solution into old */
#pragma omp parallel for private(i)
   for (j=0; j<m; j++)
     for (i=0; i<n; i++)
       uold[i + m*j] = u[i + m*j];
   /* compute stencil, residual and update */
#pragma omp parallel for reduction(+:error) private(i,resid)
   for (j=1; j<m-1; j++)
     for (i=1; i<n-1; i++){
       resid =( ax * (uold[i-1 + m*j] + uold[i+1 + m*j])
               + ay * (uold[i + m*(j-1)] + uold[i + m*(j+1)])
               + b * uold[i + m*j] - f[i + m*j]
              ) / b;
       /* update solution */
       u[i + m*j] = uold[i + m*j] - omega * resid;
       /* accumulate residual error */
       error =error + resid*resid;

     }
   /* error check */
   k++;
   error = sqrt(error) /(n*m);
 } /* while */
```

Listing 6.6: Code of the OpenMP implementation of the Jacobi method

```
while ((k < maxit) && (error > tol)) {
  error = 0.0;
  #pragma omp target device (cuda) copy_in(uold, f, u) copy_out(u)
  #pragma omp parallel shared(uold, u, ...) private(i, j, resid)
  {
    #pragma omp for
    for (i = 0; i < m; i++)
      for (j = 0; j < n; j++)
        uold[i][j] = u[i][j];
    #pragma omp for reduction(+:error)
    for (i = 0; i < (m - 2); i++) {
      for (j = 0; j < (n - 2); j++) {
        resid = ...
         ...
        error += resid * resid;
      }
    }
  }
  k++;
  error = sqrt(error) / (double) (n * m);
}
```

Listing 6.7: Iterative loop in Jacobi showing the usage of the OpenMP extensions

Version	Time transfer in	Time transfer out	Kernel time	Total time	% Speedup
Naive Approach	$0.02524s$	$0.016229s$	$1.03017s$	$3.747910s$	-
Using a data clause	$0.01133s$	$0.016193s$	$1.02849s$	$1.433504s$	61%
Splitting *C* loops	$> 0.01s$	$0.016176s$	$0.23832s$	$0.434439s$	88.4%

Table 6.3: Time per phase and speedup for each incremental optimization over the naive implementation, as measured in *Peco* using the NVIDIA CUDA platform over the GPU. The cost of the CUDA calls, context initialization and memory transfers were noticeable in this case, thus using the `data` clause improved performance

measures the performance gain when using this language feature by comparing a pure OpenMP implementation (8 threads, one per core) with CUDA code generated by `llCoMP` specifying (label CUDA v2) the memory transfers with these clauses and without them (label CUDA v1) using the Tesla C1060 from *Garoe*.

In our translation strategy, at the end of each parallel region we synchronize Host and device memories. Inside a parallel region we assume that memory locations allocated in the Host remain unchanged. The programmer should use the OpenMP `flush` construct to synchronize the Host and device for the case where access to variables computed in the device in a previous parallel loop is needed inside the parallel region. The insertion of the `flush` construct is not required in the case of function calls because they are automatically translated into device code using *inlining*.

When the `llCoMP` CUDA back end finds the `collapse` clause, it generates a 2D kernel. The `x` coordinate represents the first loop and the `y` coordinate represents iterations of the second loop. This implementation then produces lighter CUDA threads, reducing memory access conflicts and increasing granularity. However, a correct kernel launch configuration has to be chosen in order to increase performance, as we can see in Figure 6.9.

Listing 6.7 highlights the reasons for us dropping this syntax in favour of a different set of directives for GPU code. The information required to compute the information in the `parallel` region is annotated using the `copy_in` and `copy_out` in the `target` directive at Line 3. The variables `u`, `f` and `uold` remain unchanged during the execution of the `while` loop in Line 1. However, in each iteration of the `while` loop we transfer in and out the variables as indicated by the `target` directive. A possible solution for this could be to move the `parallel` construct before the `while` loop as this would reduce memory transfers and increase the performance. However, this breaks OpenMP semantics. According to the OpenMP ver. 3.1 reference:

> When a thread encounters a parallel construct, a team of threads is created to execute the parallel region (see Section 2.4.1 on page 36 for more information about how the number of threads in the team is determined, including the evaluation of the if and num_threads clauses). The thread that encountered the parallel construct becomes the master thread of the new team, with a thread number of zero for the duration of the new parallel region. All threads in the new team, including the master thread, execute the region. Once the team is created, the number of threads in the team remains constant for the duration of that parallel region.

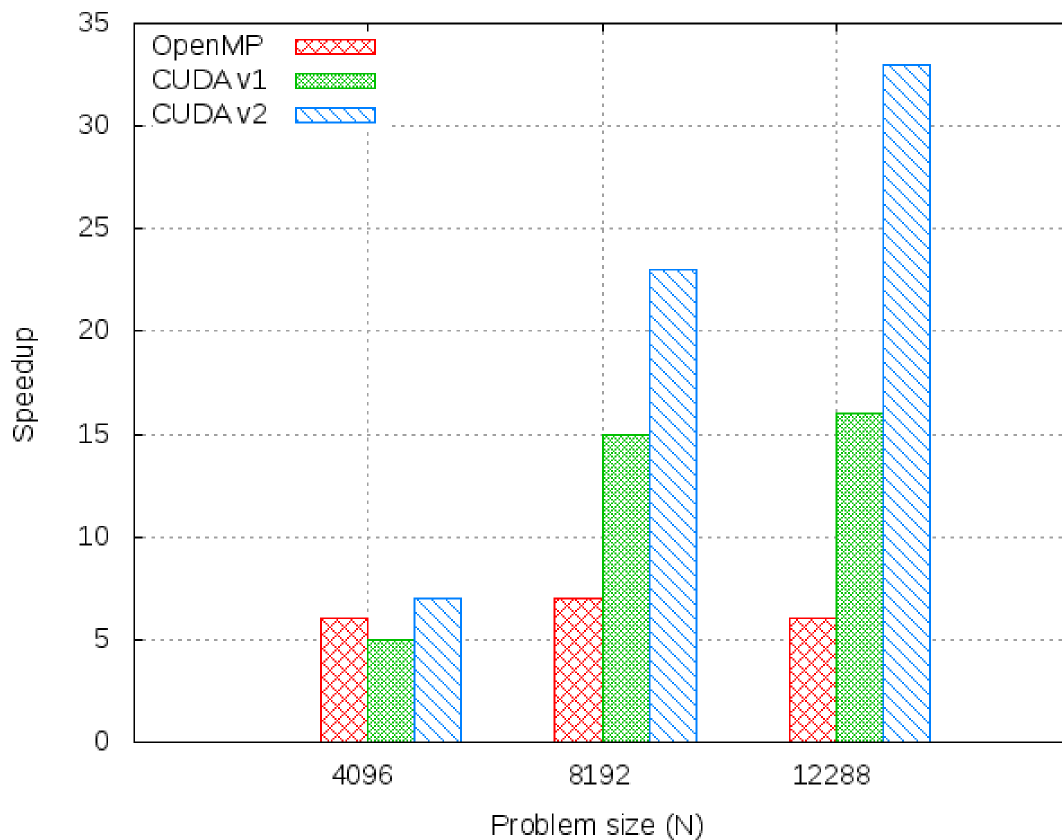


Figure 6.8: Speedup of the Jacobi code for different problem sizes, using the Tesla C1060 of *Garoe*

In this case, the `while` loop is **not parallelizable** (each iteration depends on the next one), so we cannot execute it in parallel on different threads.

A directive-based approach for code offloading must have the ability to separate **data regions** from **compute/parallel regions** to provide users with the possibility of optimizing the memory transfers.

6.3.4 LU Reduction

The LU reduction algorithm, involving LU decomposition, is typically used in HPC as a benchmarking algorithm. LU reduction presents a special parallelized version of a LU decomposition [141].

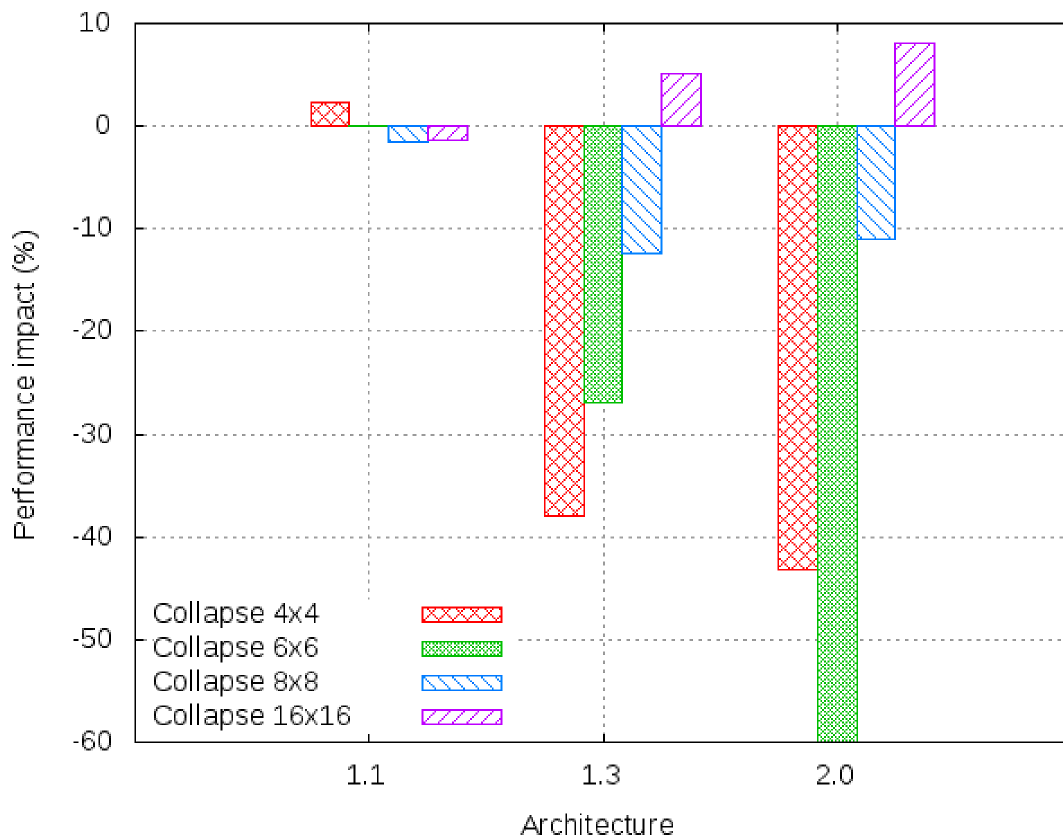


Figure 6.9: Imapct on performance of the `collapse` clause in the Jacobi code using the Tesla C1060 of *Garoe*

6.3.4.1 Impact of Kernel Optimizations in different CUDA architectures (LLC-CUDA)

With this simple source code, whose llc implementation is shown in Listing 6.8, we aim to demonstrate the increased productivity of using llc. From this source code, llCoMP is capable of producing different translations, like MPI, Hybrid MPI-OpenMP or CUDA, without modifications.

In order to use the CUDA device, the programmer needs only to specify the target directive (Line 1). The compiler can use this information to optimize the code.

```
1  #pragma omp target device(cuda) copy_in(M2,L) copy_out(M2)
2  #pragma omp parallel shared(M2, L, size) private(i, j, k)
3  for(k=0; k<size-1; k++) {
4      #pragma omp for
5      for (i=k+1; i<size; i++) {
6          L[i][k] = M2[i][k] / M2[k][k];
7          for (j=k+1; j<size; j++) {
8              M2[i][j] = M2[i][j] - L[i][k]*M2[k][j];
9             }
10         }
11     }
```

Listing 6.8: LU reduction in llc

The llc version significantly reduces the number of lines required for the implementation of the algorithm with respect to both native CUDA and MPI. The sequential version of this code contains 56 lines. The OpenMP implementation only adds one Line (the omp parallel for). The llc version of the code includes some modifications to the OpenMP version, intended to avoid unnecessary memory transfers, and thus contains 59 lines of code. Note that the llc code version is compatible with the OpenMP implementation. The CUDA source produced by the current llCoMP version has 154 lines of code, including error checking, CUDA memory manipulation, etc.

Our methodology leads to a significant reduction in the coding effort because developers can focus on algorithms and high-level implementation rather than working on low-level code, with CUDA pointers, memory transfers or kernel parameterization.

The kernel generated by our compiler from the annotated loop is shown in Listing 6.9. It is exposed here as an example of the code generated by llCoMP. This kernel is obviously memory-bound, since the main task of each thread is to read from or write to memory. On CUDA, using local memory is a common technique for optimizing performance of memory-bound kernels. With minor modifications to the code produced by llCoMP, we can use local storage to improve the performance of the kernel. Using local storage for the L matrix (renamed L_{cu} during the StS translation) avoids consequent references to global memory thus reducing latency in memory access. Each thread only accesses one shared memory position, avoiding bank conflicts.

As it is shown in Figure 6.10, using local storage in older architectures yields a significant performance boost. However, the same technique used in the NVIDIA Fermi architecture only provides a minimal benefit. The new cache hierarchy used to cache global memory access renders the manual implementation of this local storage unnecessary.

```
1  __global__ void CM_0_loopKernel0(double *M2_cu, double *L_cu, int size, int k)
2  {
3     int i = (blockIdx.x * blockDim.x) + threadIdx.x + (k + 1);
4     int j;
5     if ((i < size)) {
6        L_cu[(k * 4096) + i] = M2_cu[(k * 4096) + i] / M2_cu[(k * 4096) + k];
7         for (j = k + 1; j < size; j++) {
8            M2_cu[(j * 4096) + i] = M2_cu[(j * 4096) + i] -
9             (L_cu[(k * 4096) + i] * M2_cu[(j * 4096) + k]);
10        }
11    }
12 }
```

Listing 6.9: Kernel generated automatically by `llCoMP` for the LU reduction

```
1  __global__ void CM_0_loopKernel0(double *M2_cu, double *L_cu, int size, int k)
2  {
3     int i = (blockIdx.x * blockDim.x) + threadIdx.x + (k + 1);
4     int j;
5     double __shared__ cache[CUDA_NUM_THREADS];
6     if ((i < size)) {
7        cache[threadIdx.x] = M2_cu[(k * 4096) + i] / M2_cu[(k * 4096) + k];
8         for (j = k + 1; j < size; j++) {
9          M2_cu[(j * 4096) + i] = M2_cu[(j * 4096) + i] -
10            (cache[threadIdx.x] * M2_cu[(j * 4096) + k]);
11       }
12        L_cu[(k * 4096) + i] = cache[threadIdx.x];
13    }
14 }
```

Listing 6.10: Kernel modified to take advantage of the thread block shared memory

6.3.5 Matrix Multiplication

Matrix multiplication (MxM) is a basic kernel frequently used to demonstrate the peak performance of GPU computing. In this section, we focus on a blocked matrix multiplication algorithm, similar to that used in the well known BLAS routines [45]. The recommended way to implement a matrix product in a source code is to use a BLAS implementation tuned for the machine. The OpenACC implementation of MxM is shown in Listing 6.11.

Figure 6.10: Performance impact of local storage usage using three different GPU architectures in *Ilion* for the GPU card with arch 1.1, the Tesla C1060 of *Garoe* for the architecture 1.3 and the Tesla C2050 for the architecture 2.0

6.3.5.1 Optimizing Loop Nests

Listing 6.11 shows a possible OpenACC implementation of the MxM code, in which we have chosen to use an external `kernels` construct. This construct creates a data region and sets the variables required inside and/or outside the region. Inside the `kernels` construct, we define two loops that will be translated into GPU kernels. The first loop (Line 4) deals with matrix initialization. The `collapse` clause in Line 3 informs to the compiler driver that the loop is suitable to be extracted as a 2D kernel. We use the loop nest at Line 14 to generate a kernel with the inner loop.

The `collapse` clause is not implemented either in the current versions of the PGI or CAPS HMPP compilers. We thus had to choose a slightly different implementation for them, where the loop at Line 16 is also annotated with a `loop` directive. Both j-based and i-based loops feature a gang and `independent` to force the extraction of a 2D kernel for the GPU. Listing 6.11 includes both spellings in order to illustrate both syntaxes (lines 3 and 13), both of them supported by `accULL`. Figure 6.11 shows the floating-point performance of the different implementations.

```
#pragma acc kernels pcopy(a[0:n*l]) pcopyin(b[0:l*m],c[0:m*n]...
{
#pragma acc loop private(i, j) collapse(2)
for (i = 0; i < l; (i++))
 for (j = 0; j < n; (j++))
  a[(i * l) + j] = 0.0;
 /* Iterate over blocks */
for (ii = 0; ii < l; ii += tile_size)
 for (jj = 0; jj < n; jj += tile_size)
  for (kk = 0; kk < m; kk += tile_size)
   {
   /* Iterate inside a block */
   #pragma acc loop private(j) gang independent
   for (j = jj; j < min (n, jj + tile_size); (j++))
     #pragma acc loop private(i) worker independent
    for (i = ii; i < min (l, ii + tile_size); (i++))
     for (k = kk; k < min (m, kk + tile_size); (k++))
            a[(i * l) + j] += (b[(i * l) + k] * c[(k * m) + j]);
   }
}
```

Listing 6.11: Sketch of MxM in OpenACC

6.3.5.2 Kernel Grid Configuration in OpenACC

One of the most important aspect of CUDA tunning involves the selection of an appropriate thread and kernel block.

OpenACC provides the `gang` and `worker` clauses to make the manual tuning of kernel dimensions possible. However, how these clauses map to each level of the GPU architecture is not constant across different implementations of the standard.

Figure 6.12 shows the effect that varying the number of `gang`, `worker` and `vector` has on the overall performance, and how this effect varies from one compiler implementation to another. The implementation of the aforementioned clauses in each compiler is not exactly the same.

In the CAPS HMPP implementation being used, support for the `gang`/`worker` clauses enables a strip-mining transformation of the loop where the number of gangs is used as the number of blocks, and the number of workers as the number of threads. By default, the CAPS HMPP compiler swaps loops assuming that they are written in the usual C style (where the outermost loop iterates over the rows and the innermost iterates the columns). For this particular case - where the outermost loop iterates over the columns and the innermost over the rows - swapping the loops is not useful and degrades performance. To avoid this loop interchanging, we swapped the gangs and workers clauses (i.e. the `worker` clause will be in Line 13 and the `gang` clause will be in Line 15 in Listing 6.11). The default values for gangs and workers are 256 and 32 respectively, as no automatic detection is performed. Each thread inside the block will access non-contiguous memory positions, with a stride of 32 for the rows and a stride of 256 for the columns.

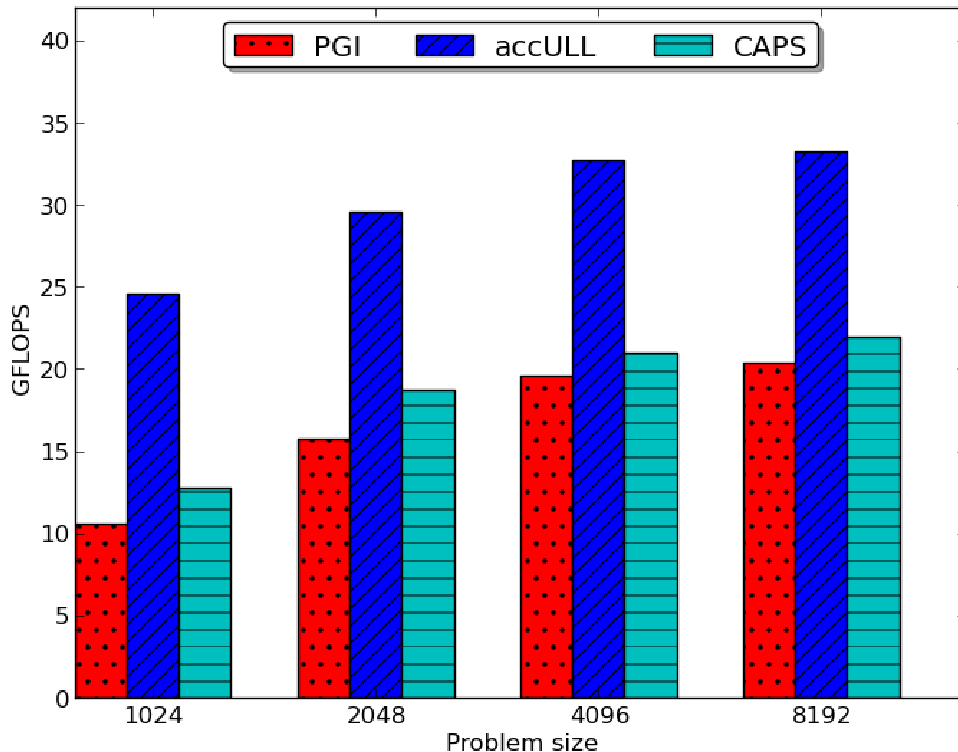


Figure 6.11: Performance comparison of three OpenACC implementations using the Tesla C2050 of *Garoe* . PGI version 12.6, CAPS HMPP version 2.3.3 and the release 0.1 of `accULL` have been used

In `accULL`, we do not use strip-mining to generate the kernel. The kernel we run assumes that each thread in each block will execute a single iteration, and the runtime will adjust the number of threads and blocks dynamically so as to execute the appropriate number of iterations. This imposes an upper limit on iterations since the maximum number of threads and blocks depends on the GPU architecture. Figure 6.12 shows the performance achieved when varying the number of threads. The `gang` and `workers` clauses are not implemented in `accULL`, we used the environment variable to manually set a number of threads, thus, forcing the runtime to compute the appropriate number of blocks and mimic the block/thread configuration of the other compilers.

To maximize performance, the `Frangollo` runtime library favors cache over shared memory if the architecture supports it (as NVIDIA Fermi and future Kepler cards do).

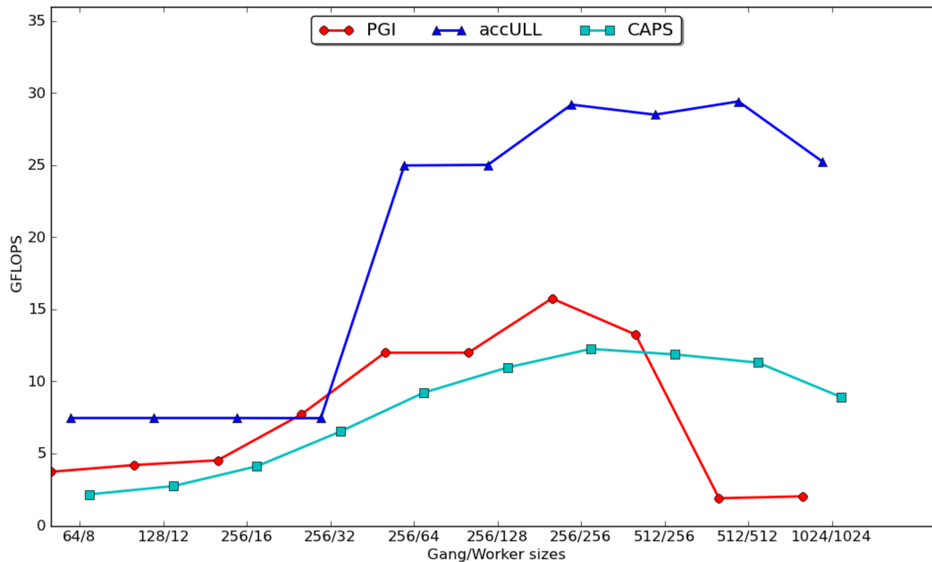


Figure 6.12: Effect of varying the values of the `gang`, `worker` and `vector` clauses using the Tesla C2050 of *Garoe*

The PGI compiler optimizes loop nests in-depth by using an advanced planner, which are described in detail in [146]. These loop optimizations require the loops to be fully parallelizable (i.e. each iteration is completely independent from the rest); thus, if the compiler is not able to ensure this condition, it will not generate the GPU kernel. Users can use the `independent` clause to force the compiler to generate the kernel. This clause instructs the compiler not to check dependencies and assume that iterations are independent. Iterations from the loop in Line 13 of Listing 6.11 correspond to the x dimension of the kernel, whereas the iterations in Line 15 are distributed in the y dimension. The innermost k-loop is unrolled.

The PGI compiler supports different combinations of `worker`, `gang` and `vector` in different nested loops. Each `acc loop` directive may contain combinations of `worker`, `gang` and `vector` clauses.

When encountering a loop nest (like that in the MxM code), iterations for each loop are spread across each dimension in a similar manner to `accULL`. Depending on the memory access pattern of the loop nest body, loops might be interchanged. The PGI compiler output informs the user that a two-dimensional kernel is created with each loop in a different dimension. Information about how these loops transformations are performed is shown during the compilation. Performance figures shown in Figure 6.12 were obtained by equally distributing the gangs and vectors across each dimension (for example, if gangs were 256, then in PGI we used 16 for the gangs in each dimension).

It is advisable to use the PGI information command line option to show detailed information on the CUDA code generation. This information enable users to improve their parallelization by solving the performance bottlenecks flagged up by the compiler. Sometimes the PGI compiler does not create the GPU kernel, but the compilation finishes properly. When we displayed the information at compile time, we realized that the code was not parallelized at all due to false dependency detection among iterations. We believe that it is important to provide developers with not only a proper set of directives, but also with profile and debugging tools that make developing easier.

6.3.5.3 Loop Invariant Optimization

The `YaCF` loop optimization module uses data dependency analysis to enable different optimizations. In the MxM implementation, the index expression for the array access inside the loop `a[i * L +j]` is independent from the innermost loop. As shown in Listing 6.12, the loop analysis module can detect this situation and replaces the array access with a private variable. When running on GPU devices, this variable is mapped to a register, thus, greatly leveraging the number of memory accesses and increasing performance. Figure 6.13 shows the impact in the kernel performance of this loop invariant optimization.

```
13 tmp = a[i * L + j];
14 for (k = kk; k < min(m, kk+tile_size); k++)
15   tmp += (b[i * L + k] * c[k * m + j]);
16 a[i * L + j] = tmp;
```

Listing 6.12: Extracting the array access

In Figure 6.14 we present performance figures using the Intel OpenCL implementation in *Drago*, and compare them with its OpenMP counterpart. In certain instances, depending on the size of the problem, it appears that OpenCL outperforms the OpenMP implementation; this leads us to believe that it is worth exploring the use of `Frangollo` with CPU implementations.

6.3.6 Development Effort Analysis (`LLC-CUDA`)

In order to substantiate our claims that our methodology decreases the amount of effort that needs to be dedicated to development, in this section we provide an analysis using sloccount [142]. This includes both the original `llc` code and the target CUDA code produced by `llCoMP` which is applied to all the algorithms presented in the previous sections. A study of this kind which looks at the OpenACC directive-based approach would produce similar results.

With `sloccount` we used the basic COCOMO [30] model in organic mode to estimate the code cost. This model estimates effort and schedule, including design, code, test, and documentation time for the code which is being analysed. COCOMO involves a couple of constants that represents the development effort - F which represents the effort factor, and E - which represents the complexity of the source code as it grows in line number.

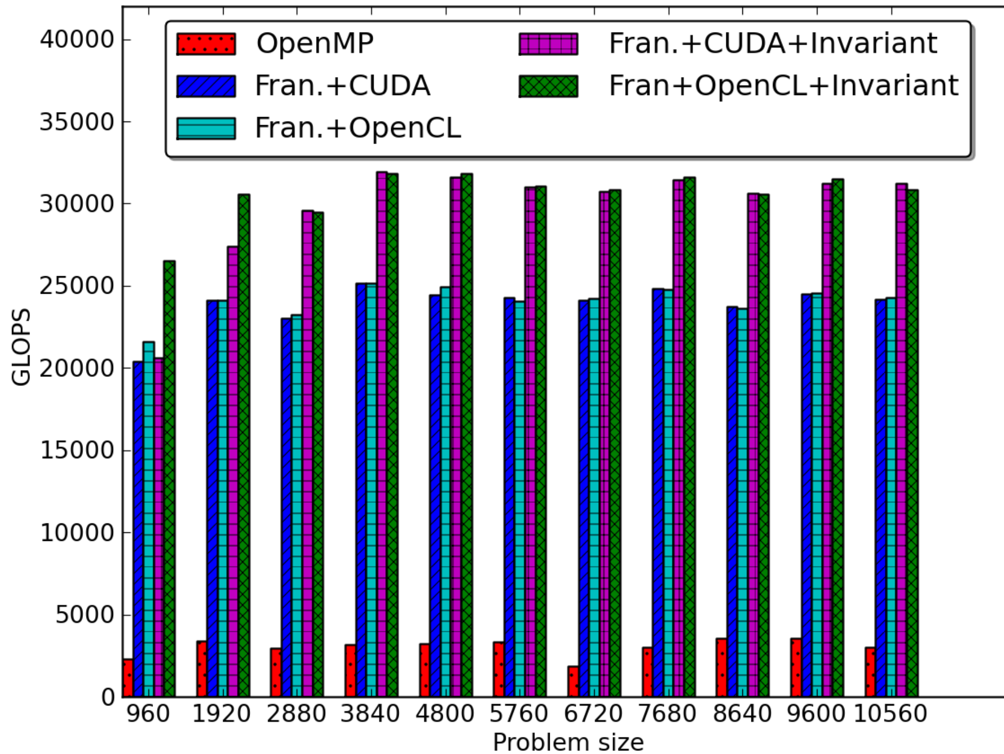


Figure 6.13: Floating point performance for MxM in *Peco*. We compare OpenMP, Frangollo with CUDA/OpenCL and an improved version of the kernel using loop invariant optimization for `a[i*L+j]`

It is not easy to fix the values for E and F to obtain precise metrics, and it requires a lot of experimentation which is outside the remit of our work. The effort required to develop a CUDA code is clearly higher than that corresponding to a `llc` code. We have estimated a relationship between these efforts in which E is equal to 1.05 for both cases, and F is equal 2.4 for `llc` code, and 4.8 for CUDA code.

The choice of different values for E and F is not a matter of correctness for each environment, but a matter of making correct estimates of coding efforts. Although the values themselves may be subject to debate, there is no question that the difference in effort between the environments is evident

The results shown in Table 6.4 confirm that for different metrics the development costs using our methodology are clearly lower.

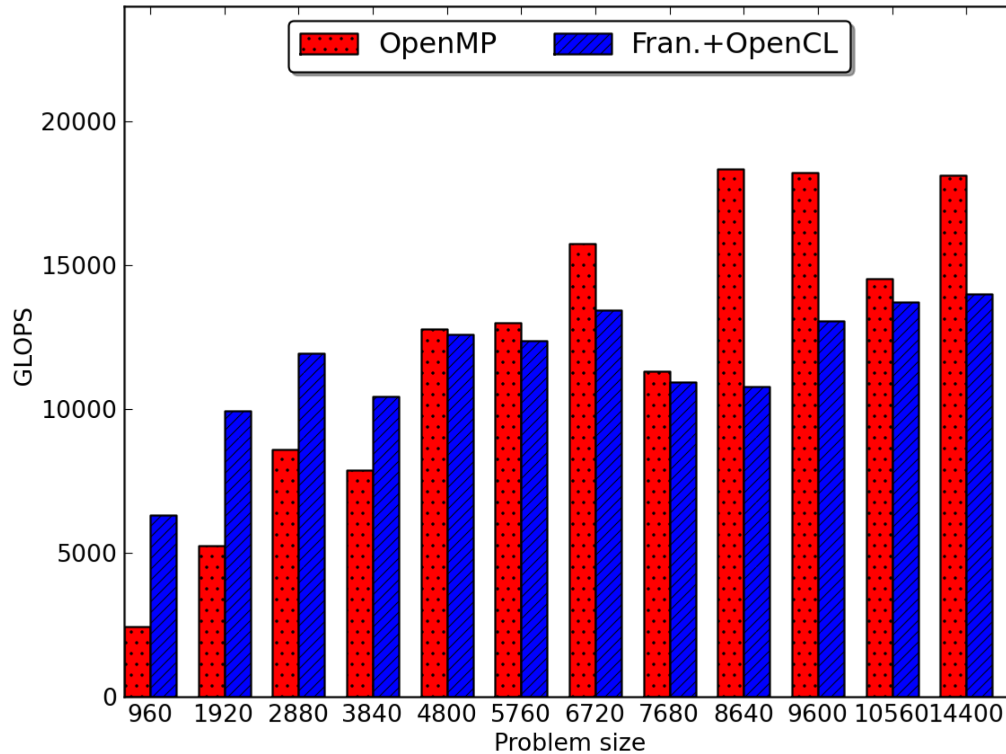


Figure 6.14: Comparison between OpenMP gcc implementation and `Frangollo`+OpenCL in *Drago* platform using the full system (40 cores) for MxM

6.4 Rodinia Benchmark Suite

The Rodinia Benchmark Suite [38, 39] is composed of a set of workloads designed for heterogeneous computing platforms, including both CPUs and devices such as GPUs and FPGAs. Several different kind of compute-heavy applications from different fields are included, alongside classical algorithms like LU decomposition and graph traversal. The Rodinia benchmarks are currently implemented in OpenMP, CUDA and OpenCL.

	LU		Mandelbrot		Jacobi		MD	
Code version	llc	CUDA	llc	CUDA	llc	CUDA	llc	CUDA
# Lines	86	157	145	222	127	260	161	304
Person/Year	0.02	0.06	0.03	0.08	0.02	0.10	0.03	0.11
Person/Month	0.18	0.69	0.32	0.99	0.27	1.17	0.35	1.37
Schedule/Year	0.11	0.18	0.13	0.21	0.13	0.22	0.14	0.24
Schedule/Month	1.31	2.17	1.61	2.49	1.53	2.65	1.68	2.82

Table 6.4: `sloccount` data for cases of study.

6.4.1 SRAD

Speckle Reducing Anisotropic Diffusion (`SRAD`) is a diffusion algorithm based on partial differential equations that is used for removing the speckles in an image without sacrificing important image features. `SRAD` is widely used in ultrasonic and radar imaging applications. The program's inputs are ultrasound images and the value of each point in the computation domain is dependent on its four neighbours. Figures 6.15 and 6.16 show the performance of the `accULL` implementation against the native implementation in *Garoe* and *Drago* respectively. The OpenACC implementation of the code is shown in Listing 6.13. Following the same pattern we have seen in both Jacobi and MD implementations (see Section 6.3), the data region is created outside the iteration loop, whereas kernels are created for the inner loops. This reduces the transfers between the Host and the device as all the actual computation is performed inside the device, and only the final result is computed at the end. The `independent` clause is used in the loops to force PGI and other compilers to generate CUDA kernels for these loops.

6.4.2 LU Decomposition

The LU decomposition has many row-wise and column-wise interdependencies and requires significant optimization to yield good parallel performance. A performance comparison of the OpenACC implementation with the PGI Accelerator Model, the hiCUDA, the original OpenMP and the native CUDA code is shown in 6.17. For these problem instances, the OpenMP (4 threads) version provides the best results relative to the native CUDA implementation. Experiments with a smaller problem size indicate that the speedup achieved by accelerating the computation on the GPU does not compensate the time required for the memory transfers.

The complex row-wise and column-wise interdependencies of the original algorithm are difficult for the compiler to identify, and the resulting kernel is not as efficient as it could be. According to the PGI compile-time information, some references to the matrix are cached, which improves performance. The `accULL` kernel does not use shared memory at the moment, however the `Frangollo` runtime favours cache over shared memory for the kernel in an attempt to compensate for this fact. For small problem sizes no benefit is obtained from offloading the code to GPU, although performance increases as problem sizes increase.

```
#pragma acc data copy(c[0:size_I], J[0:size_I]) copyin(dN[0:size_I],...)
for (iter = 0; iter < niter; iter++) {
        sum = 0;
        sum2 = 0;
        #pragma acc kernels loop private (i) independent
        for (i = r1; i <= r2; i++) {
            #pragma acc loop private (j, tmp) reduction(+:sum, sum2) independent
            for (j = c1; j <= c2; j++) {
                tmp = J[i * cols + j];
                sum += tmp; sum2 += tmp * tmp;
            }
        }
        meanROI = sum / size_R;
        varROI = (sum2 / size_R) - meanROI * meanROI;
        q0sqr = varROI / (meanROI * meanROI);
    #pragma acc kernels loop private(i) independent
    for (int i = 0; i < rows; i++) {
    #pragma acc loop private(j, k, Jc, G2, L, num, den, qsqr) independent
    for (int j = 0; j < cols; j++) {
        k = i * cols + j;
        Jc = J[k];
        // directional derivates
        dN[k] = J[iN[i] * cols + j] - Jc;
        ...
        qsqr = num / (den * den);
        // diffusion coefficent (equ 33)
        den = (qsqr - q0sqr) / (q0sqr * (1 + q0sqr));
        c[k] = 1.0 / (1.0 + den);
        // saturate diffusion coefficent
        if (c[k] < 0) {
            c[k] = 0;
        }
        else if (c[k] > 1) {
            c[k] = 1;
        }
    }
}
...
```

Listing 6.13: Main loop of the SRAD implementation in OpenACC.

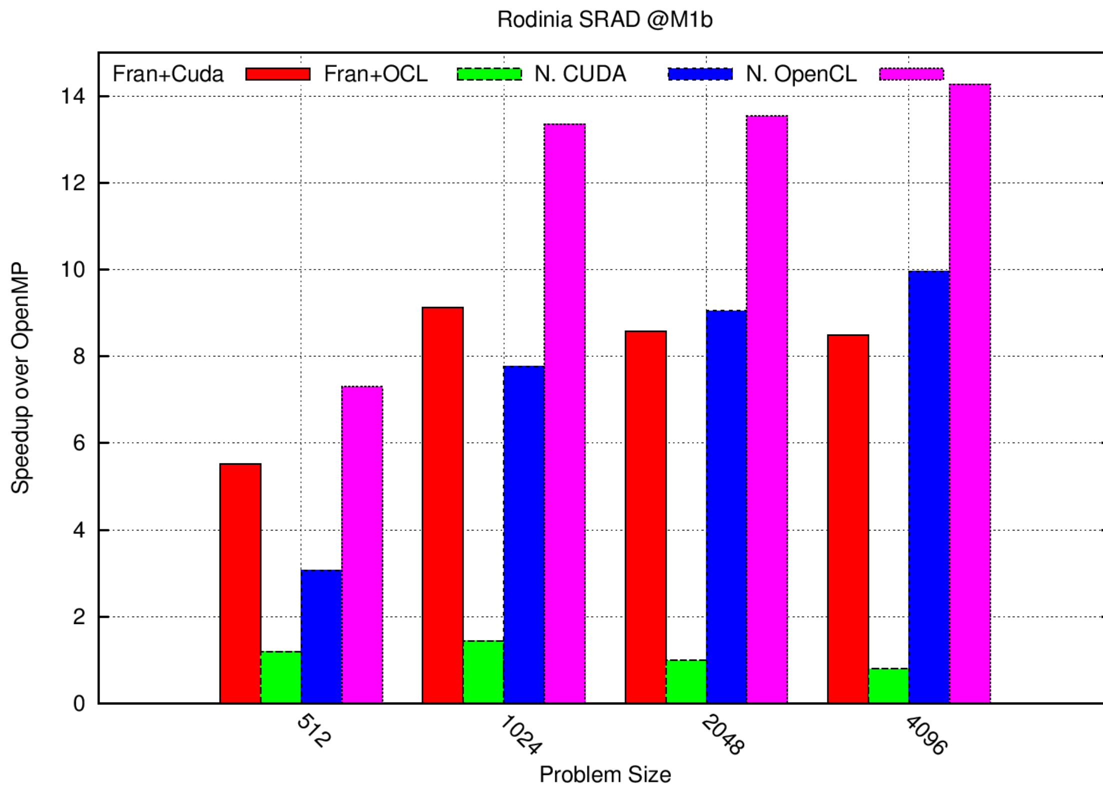


Figure 6.15: Performance comparison of accULL versus native implementation, showing the speedup against OpenMP using the Telsa C2050 of *Garoe*

Listing 6.14 shows the OpenACC implementation of the LU decomposition. Notice that macros are used to differentiate when the code is compiled with the CAPS HMPP implementation (`__HMPP`) or with any other implementation. One of the main pitfalls of the current standard is that some features of the language are not implemented homogeneously across all compiler implementations. In this case, the CAPS HMPP implementation achieves better performance when using a single `kernels` region with two `loop` clauses. On the other hand, the `accULL` and the PGI compilers perform better when using a `data` region and two separate `kernels` regions. Performance figures comparing the performance of different OpenACC implementations are shown in Figure 6.18.

```
#ifndef __HMPP
    #pragma acc data copy(a[0:size*size])
#else
    #pragma acc kernels copy(a[0:size*size])
#endif
     for (i=0; i <size; i++){
#ifndef __HMPP
        #pragma acc kernels loop private(j) independent
#else
        #pragma acc loop private(j) independent gang(64) worker(64)
#endif
         for (j=i; j <size; j++){
             float sum=a[i*size+j] * (-1.0);
             int k;
             for (k=0; k<i; k++) sum += a[i*size+k]*a[k*size+j];
             a[i*size+j]=sum * (-1.0);
         }
#ifndef __HMPP
        #pragma acc kernels loop private(j) independent
#else
        #pragma acc loop private(j) independent gang(64) worker(64)
#endif
         for (j=i+1;j<size; j++){
             float sum=a[j*size+i] * (-1.0);
             int k;
             for (k=0; k<i; k++) sum += a[j*size+k]*a[k*size+i];
             a[j*size+i]=sum * (-1.0)/a[i*size+i];
         }
     }
}
```

Listing 6.14: Main loop of the LUD implementation in OpenACC

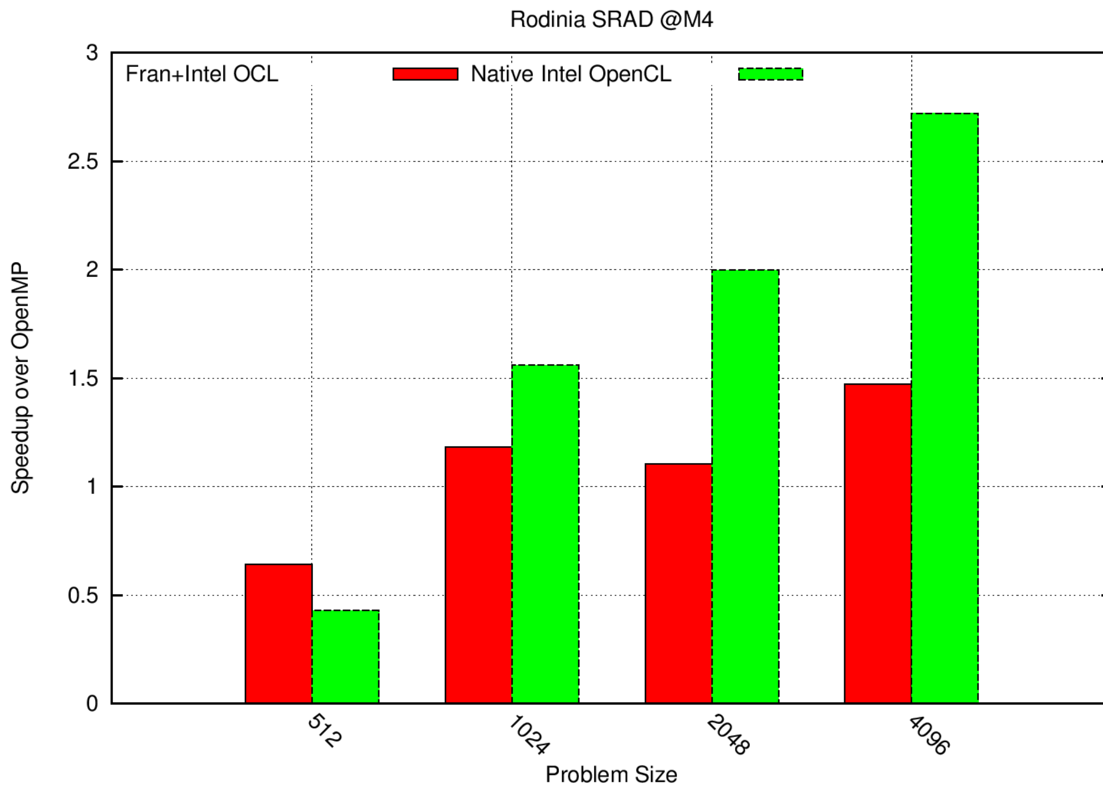


Figure 6.16: Performance comparison of SRAD using *Drago* comparing accULL with the native implementation, showing the speedup against OpenMP GCC implementation

6.4.3 HotSpot

The Rodinia Benchmark Suite includes the 2D transient thermal simulation kernel of Hot Spot (HS), which iteratively solves a series of differential equations for block temperatures. The program inputs are power and initial temperatures. Each output cell in the grid represents the average temperature value of the corresponding area of the chip. The main routine of HS contains two nested loops that run for a predefined number of iterations. The first loop computes the actual temperature of each position inside the chip, while the second one just updates the data with the information computed from the current iteration.

In OpenACC it is possible to use the `kernels` directive, which defines a data region containing a set of loops that will be executed on the accelerator device. Loops are annotated using the `loop` directive. The compiler can then create a unique data region where the information is copied to the device before the iteration steps are performed, and then transferred back to the Host when finished.

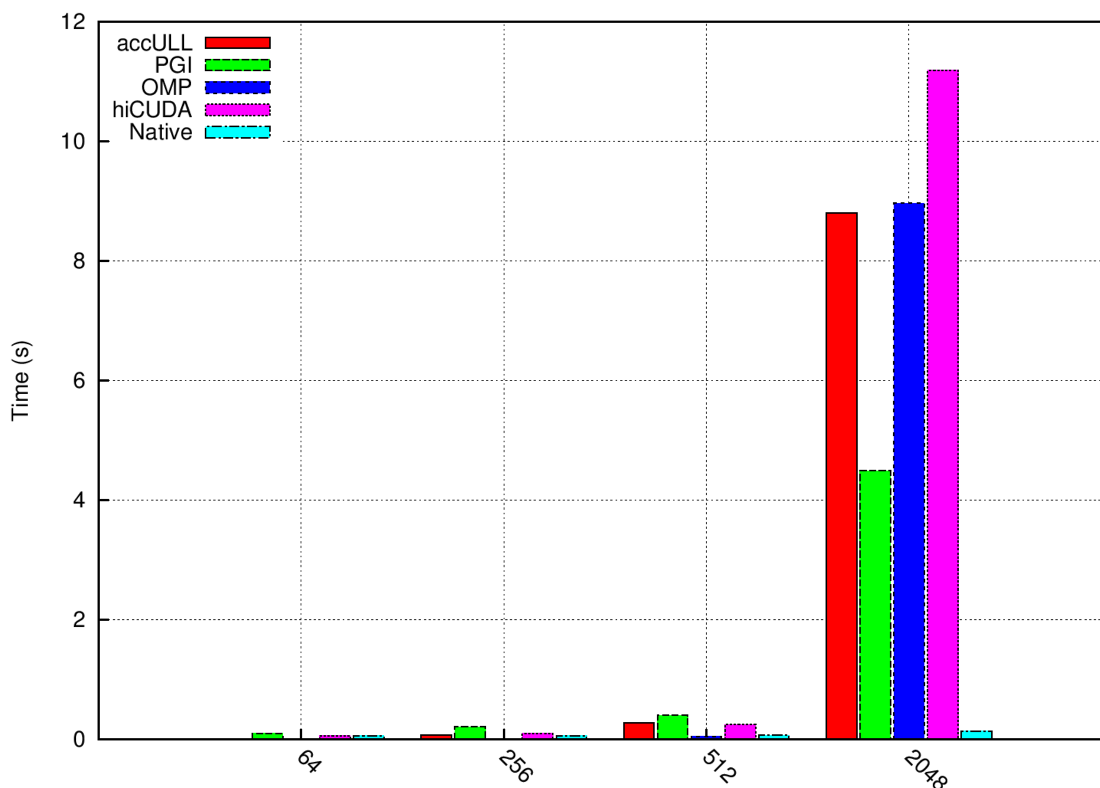


Figure 6.17: Comparison of execution time using `hiCUDA`, `PGI Accelerator Model`, OpenMP and the `accULL` OpenACC implementation of the `LUD` using the Tesla C2050 of *Garoe*

In the `accULL` implementation it is not necessary to inline the subroutine. Replacing the `kernels` directive with a `data` directive and then using the `kernels` inside the subroutine is enough for the runtime to track the usage of the host variables and to handle their device counterparts properly. A sketch of this approach is shown in Listing 6.15.

Although in this case it is easy to inline the routine, there might be other scenarios where the usage of `data` directives that are lexically distant from the point where the kernel is used can be beneficial from a productivity standpoint. As expected, the native CUDA version delivers the best results for all problem sizes. Figure 6.19 shows the performance relative to CUDA for each implementation.

6.4.4 PathFinder

PathFinder (`PF`) uses dynamic programming to find a path on a 2D grid from the bottom row to the top row with the smallest accumulated weights, and where each step of the path moves straight ahead or diagonally ahead. It iterates row by row, with each node picking a neighbouring node in the previous row that has the smallest accumulated weight and adding its own weight to the sum.

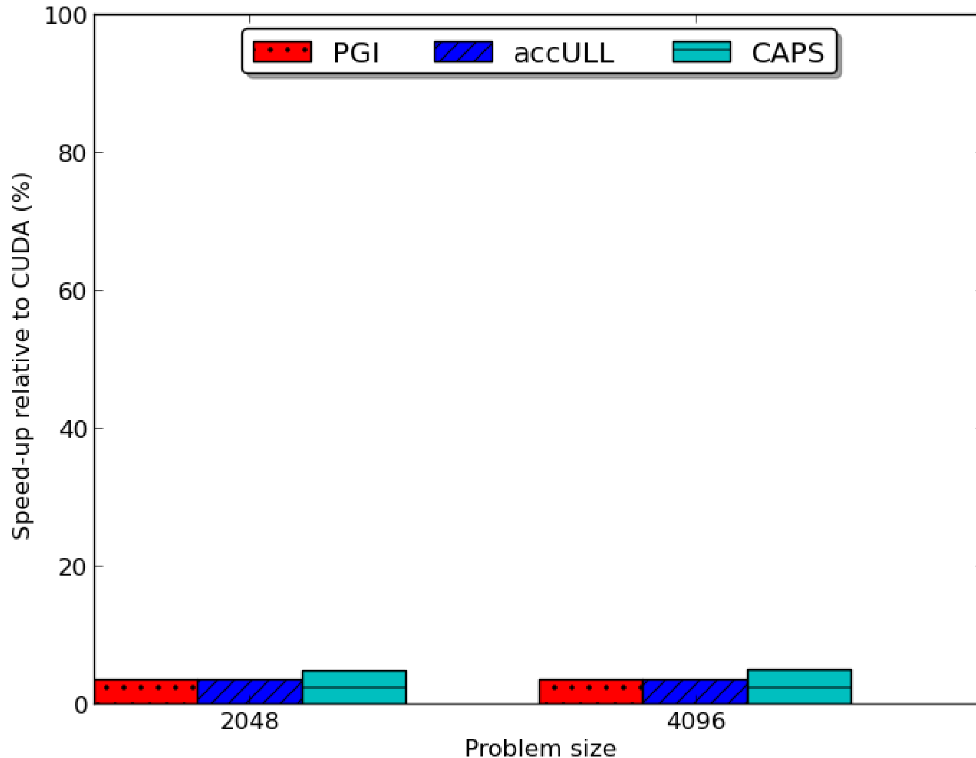


Figure 6.18: Percentage of the time of the native CUDA implementation with three different OpenACC compilers of the LUD using the Tesla C2050 of *Garoe*

Performance figures for PF are shown in Figure 6.20. The usage of the `independent` directive allowed us to force the GPU code generation for both CAPS HMPP and PGI compilers.

6.4.5 Needleman-Wunsch (NW)

Needleman-Wunsch (NW) is a nonlinear global optimization method for DNA sequence alignments. The potential pairs of sequences are organized in a 2D matrix. In the first step, the algorithm fills the matrix from top left to bottom right, step-by-step. The optimum alignment is the pathway through the array with maximum score, where the score is the value of the maximum weighted path ending at that cell. Thus, the value of each data element depends on the values of its northwest-, north- and west-adjacent elements. In the second step, the maximum path is traced backwards to deduce the optimal alignment. Properly scheduling iterations within each loop is critical to improved performance.

Performance comparison of accULL against OpenMP in a Tesla C2050 and in the CPU are shown in Figure 6.21 and 6.22. Performance comparison of PGI, hiCUDA, accULL and OpenMP implementations with respect to the native CUDA implementation is shown in Figure 6.23.

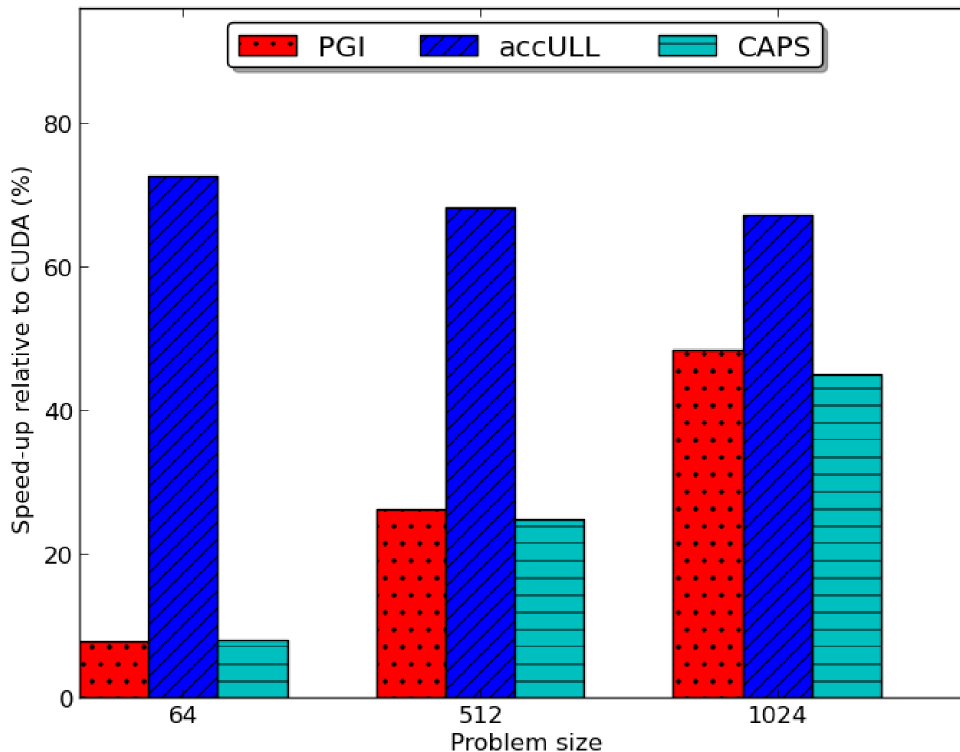


Figure 6.19: Time relative to CUDA for the HS implementation using the Tesla C2050 of *Garoe*

When working with hiCUDA, it is possible to schedule loop iterations with different combinations of blocks and threads. We explored several different thread numbers in order to obtain the maximum performance. Despite NW loops being triangular, hiCUDA was capable of properly scheduling the iterations across threads. The accULL loop scheduling for this particular algorithm is similar to the hiCUDA version, thus performance is comparable.

In this case, the PGI implementation performs the worst. Despite our best efforts to force the compiler to schedule the loops in the GPU, its dependency analysis created sequential GPU kernels. It is advisable to use the PGI information command line option to show detailed information about the CUDA code generation. This information enables users to improve parallelization by solving the performance bottlenecks pointed out by the compiler. In this case, our initial implementation with PGI directives took more than eighty seconds to conclude. When we showed the information at compile time, we realized that the code was not parallelized at all due to detection of false dependency across iterations. We have used this feature, not available on hiCUDA nor in accULL, to improve the performance of all implementations. We believe that it is important to provide developers not only with a proper set of directives, but with profile and debugging tools to make development easier.

```
void do_iteration(double * temp,...) {
 #pragma acc kernels ....
   { /* Compute current temperatures */
     #pragma acc loop ...
     for (r = 0; r < row; r++)
       for (c = 0; c < col; c++)
           ...
     /* Update */
     #pragma acc loop ...
     for (r = 0; r < row; r++)
         for (c = 0; c < col; c++)
              ....
   }
}
void routine(...) {
...
#pragma acc data copy(temp,...)
for (i = 0; i < n_it ; i++)
    do_iteration(temp ...)
}
```

Listing 6.15: Sketch of HS using OpenACC

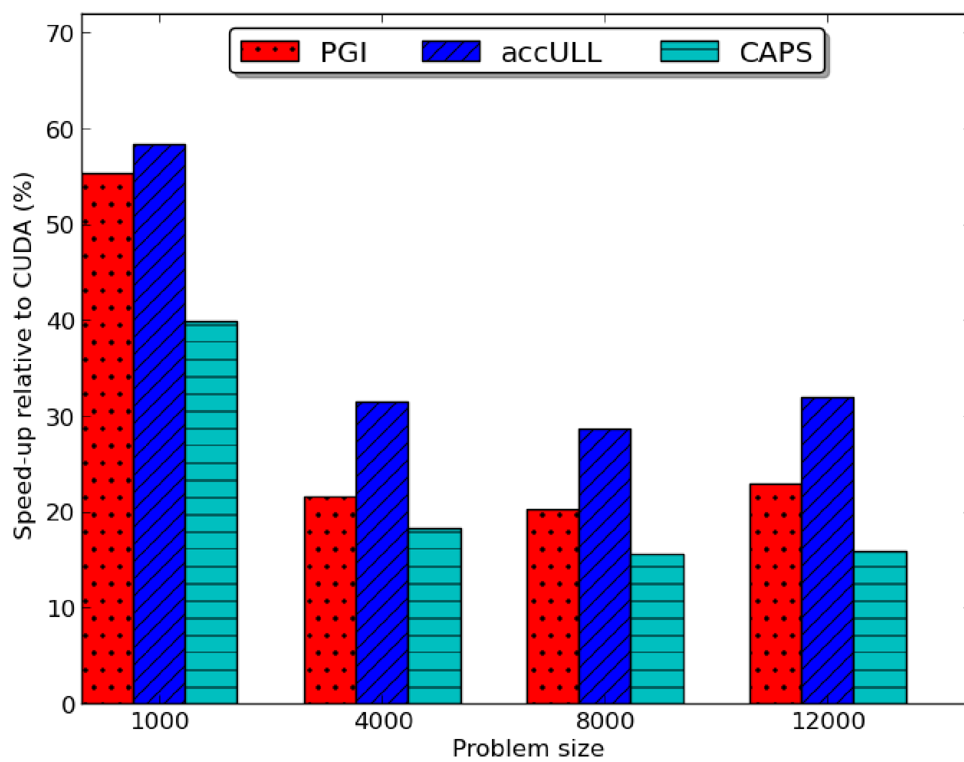


Figure 6.20: Performance of the PF implementation using the Tesla C2050 of *Garoe*

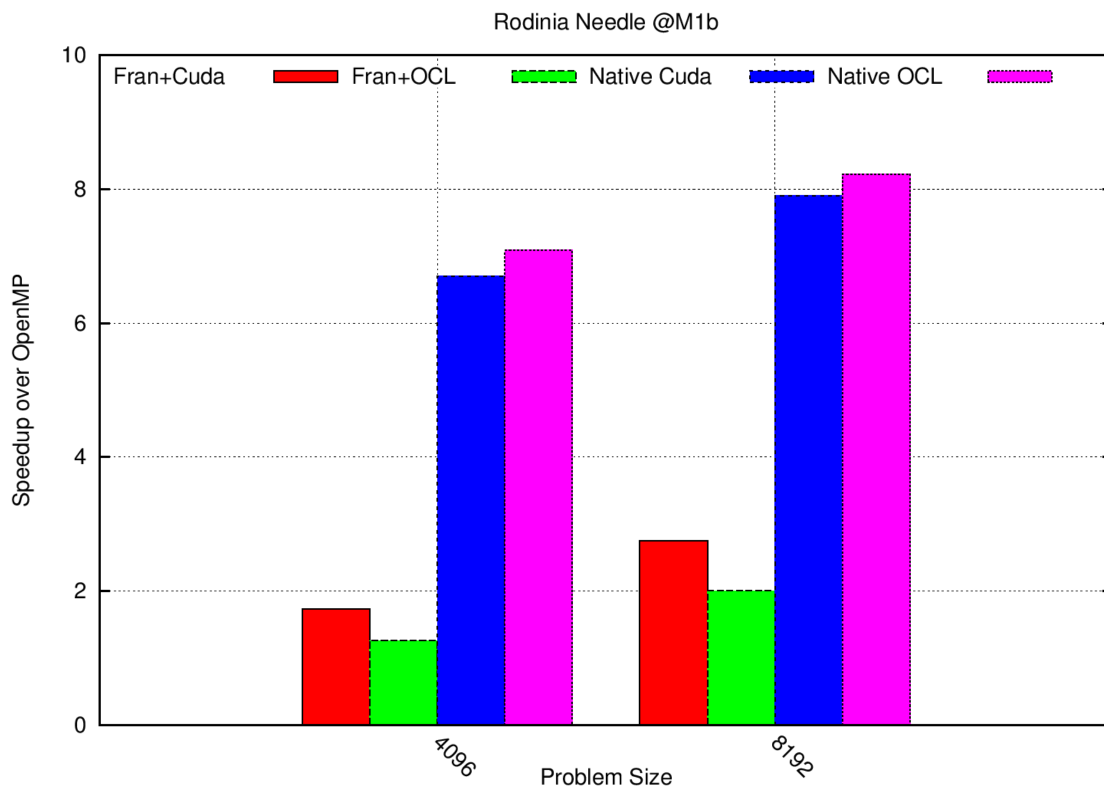


Figure 6.21: Performance comparison of `accULL` using the Tesla C2050 of *Garoe* versus native implementation, showing the speedup against OpenMP

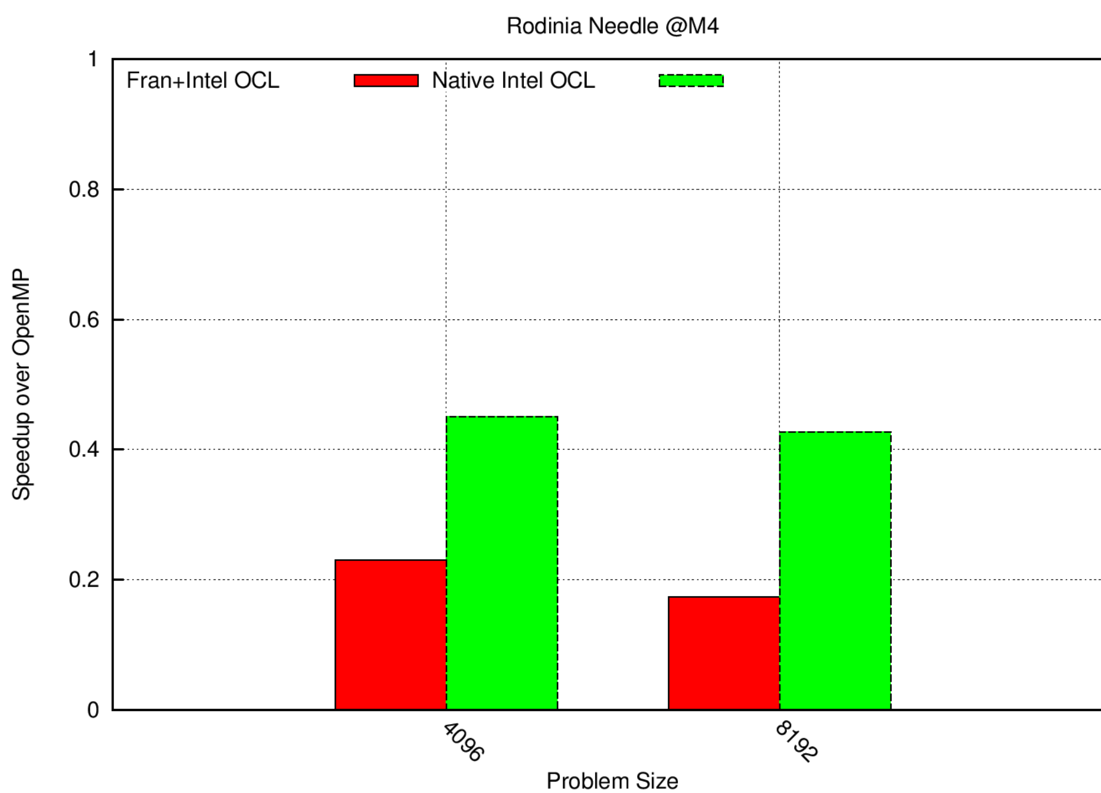


Figure 6.22: Performance comparison of `NW` using *Drago* comparing `accULL` against the provided OpenCL implementation running on the CPU, showing the speedup against OpenMP gcc implementation

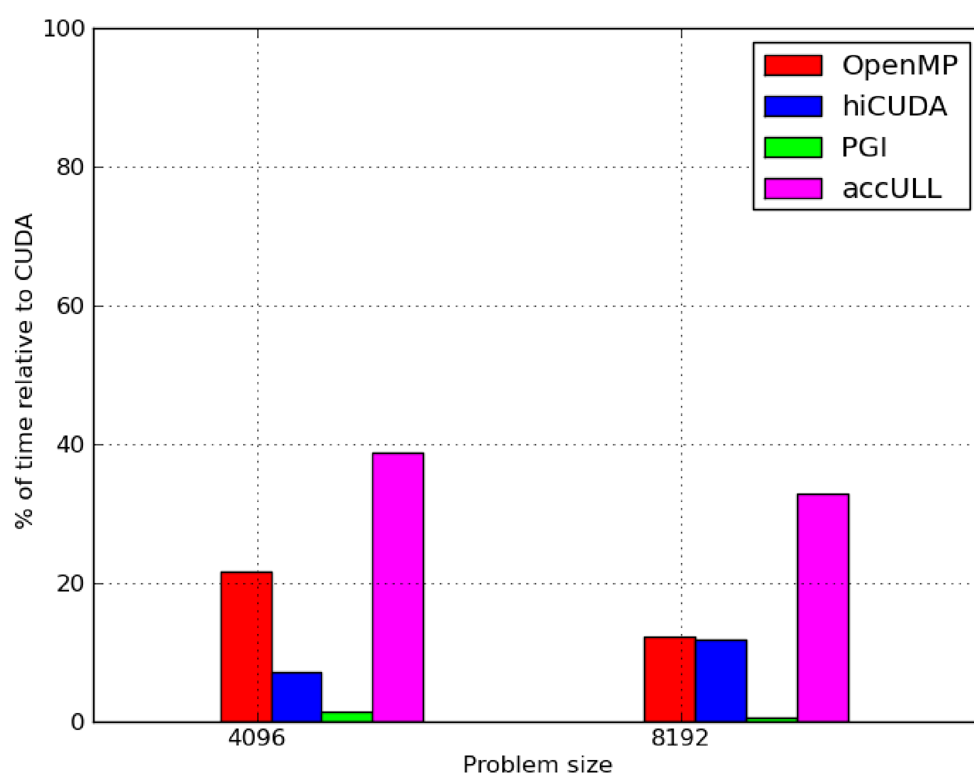


Figure 6.23: Performance comparison of `NW` using the Tesla C2050 of *Garoe*

CHAPTER 7

Conclusions and Future Work

This Chapter presents the main conclusions obtained from this thesis and details the work that still remains to be done in the future.

7.1 Directive-based Programming

The usage of directive-based programming models enables developers to easily migrate codes to new platforms. Through the information gathered by the compiler from the analysis of the annotated code it is possible to generate new code optimized for the architecture. Although this generated code could, theoretically, be as good as a native approach, sometimes this goal is unfeasible. For example, there are situations in which an algorithm is unable to extract the maximum performance because it is different to the one initially implemented in the sequential code. However, using directive-based approaches we can extract sufficient performance from the device, particularly if we compare the development effort required.

From the different directive-based programming model highlighted in the bibliographic review (see Chapter 2) we believe that OpenMP and its extension from the Barcelona Supercomputing Center, OmpSs [55], are going to become ubiquitous in the HPC environment in the near future, particularly when currently existing scientific codes start to be ported to Peta- and Exa-scale environments.

We have experience working with both OmpSs and its predecesor, SMPSs [21]: with the former, it involved porting linear algebra codes to multi-GPU environments [31]; with the latter, it involved porting linear algebra codes to hybrid MPI-SMPSs [91, 20]. Using these directive-based programming models enabled us to provide performance portability while maintaining development efforts at a minimum.

Although the OpenMP standard may one day include a similar system for tracking dependencies across tasks, it will not do so in the near future, so for this reason both programming models will coexist.

Support for accelerators may be added to the OpenMP standard in the future [17, 26, 130, 75]. In the meantime, it is worth exploring the new OpenACC standard for accelerators, whose similarities with different proposals to extend the OpenMP standard (such as the concept of a `data` region) could facilitate porting OpenACC codes to OpenMP in the future.

We have demonstrated the feasibility of porting codes to accelerator using directive-based schemes. Our preliminary work in [116] proposed an approach using minor extensions to OpenMP. This work allowed us to explore optimization strategies for accelerators, presented in [119].

With the irruption of the OpenACC programming language, which presents a similar set of directives to those we were already exploring, more hardware vendors were able to support the model due to the generalization of the accelerator directives. Our work in [125] reveals the strong and weak points of the commercial implementations available at that time, whereas our work in [123] shows how the OpenACC scheme could be used on other platforms.

7.2 Programming Tools

Compiler and runtime tools enable programming models, and particularly those based on directivess, to extract more performance from what is basically a sequential code. Compiler frameworks like [81] and [36] allow experienced developers to implement source-to-source transformations on commercial codes. They also feature a large set of existing optimizations phases that can be used to improve the performance of the final binary after the transformation takes place. However, working with these compilers is not a straightforward process, and the startup time for a developer who is new to them is not negligible. Our tool [120] provides a quick way to validate directive-based approaches, compiler optimisations and runtime interactions.

Having a two layer combination in the form of a compiler and a runtime enabled us to separate compile-time decisions from decisions that can be easily solved in runtime (such as the kind of platform we are dealing with). Other authors, such as [55] and [29] are using a similar two-layer approach with great success.

7.3 Development Productivity

Although in the world of HPC the scientists are usually focused on performance figures, developer productivity figures are also of critical importance. Scientists and engineers should focus on doing their jobs, and not on dealing with low-level codes. However, traditional approaches, particularly in the world of GPU accelerators, force developers to focus on low-level details and fancy optimizations settings. Our view is that all of these low-level, platform-dependent, details have to be removed from the equation. This is where the Programming Model demonstrates its worth. Programming Models should enable the developer to focus on the logic of the code. However, if the Programming Model is not compatible with available codes, there is the risk of loosing too much time in code porting. Directive-based models can fill this gap, and with the help of the appropriate tools (in the form of compiler and/or runtime) can alleviate the task of the developers while they are migrating the code.

7.4 Future Work

The tools designed during this work - YaCF and Frangollo - provide a magnificent opportunity for other researchers. It is easy to extend both the compiler and the runtime with new functionalities or port them both together to new platforms.

The Frangollo Platform Model supports multiple devices intuitively. Unfortunately due to time constraints we were not able to implement that feature on the Frangollo runtime. We envision two different implementations for multiple multiple-device support in the runtime. The first approach will make use of the existing *Device* class to create a virtual device composed by two physical devices. These two physical devices will use a clone operation, as described in Section 4.1, to share the address space for the input variables, and a scatter/gather combination for variables with output constraints. This will require an extention of the OpenACC directives with support to specify regions of data that are accessed by a kernel. It is also possible to re-think the kernel execution as executing tasks, in the same way as OpenMP. Two consecutive and asynchronous kernels can be executed on two separate devices, provided that the output constraints of the first does not collide with the input constraints of the second kernel. This alternative implementation does not require modifications to the OpenACC directives as the runtime could detect the situation where two different kernels with non-colliding parameters can be executed in parallel. This implementation requires manual data partitioning by the user in order to take full advantage of the parallelism. However, the detection of constraints would be easy to implement because the internal status of the variables in the runtime properly track which variables can be used and which ones are being used in a kernel.

Another interesting possibility, derived from the previous concept, is to envision a cluster as a platform with several accelerators. A node could be designated as a host and the rest of the nodes would be accelerators. The hierarchical nature of the Frangollo platform model can be instanced for this platform easily. A similar approach has been used for SnuCL [77]. SnuCL is an OpenCL framework that hides the complexity of an heterogeneous cluster using the OpenCL interface. Since our Frangollo runtime is capable of dealing with OpenCL kernels, it is simply a matter of fully implementing the support for multiple devices should we want to use SnuCL to execute OpenACC codes transparently in cluster environments.

If we now take a look at the compiler, it is clear that there are also plenty of opportunities for further work to be carried out. The YaCF compiler framework is a good tool for research but it is not suitable for production environments. With enough time and resources, it would be interesting to work on porting the OpenACC infrastructure that we have implemented in YaCF to LLVM. In combination with Frangollo, we would be able to create a production-ready OpenACC implementation, while keeping the flexibility of the Frangollo Platform Model to exploit different platforms.

The process of generating kernels in YaCF could be improved using polyhedral frameworks to better exploit the parallelism of the GPU architectures. The implementation of enhanced analysis phases, or a complete variable dependency analysis would be beneficial to enable more aggressive transformations or even the vectorization of some operations (which would benefit Very Long Instruction Word accelerators, such as the AMD Fusion platform). Nevertheless, if the objective is to fulfill the requirements of complex scientific codes, it is worth exploring the LLVM approach as it already integrates some of these optimizations.

Finally, much work could still be done on extending the OpenACC language to support more parallel skeletons, such as tasks or MapReduce, or investigating how to combine OpenACC with OpenMP. Exploring these new directives (or any other) is almost a trivial task using YaCF and Frangollo. Its flexibility and the ease of writing are ideal for these kind of experiments.

Contributions

[1] Reyes, R., and de Sande, F. Automatic code generation for GPUs in llc. *The Journal of Supercomputing 58*, 3 (Mar. 2011), 349–356. 117, 172

[2] Reyes, R., and de Sande, F. Case Studies in automatic GPGPU code generation with llc. In *Proceedings of the 2010 conference on Parallel processing* (Berlin, Heidelberg, 2011), vol. 6586 LNCS of *Euro-Par 2010*, Springer-Verlag, pp. 13–22. 117

[3] Reyes, R., and de Sande, F. Extending llc to support hierarchical clusters. In *Proc. of the Work in Progress special session on the 19th Euromicro International Conference on Parallel, Distributed and Network-Based Computing (PDP 2011)* (Cyprus, Feb 2011), pp. 21–22. 119

[4] Reyes, R., and de Sande, F. Optimize or wait? using llc fast-prototyping tool to evaluate CUDA optimizations. In *Proc. of the 19th Euromicro International Conference on Parallel, Distributed and Network-Based Computing (PDP 2011)* (Cyprus, Feb 2011), pp. 257–261. 172

[5] Reyes, R., and de Sande, F. Optimization strategies in different CUDA architectures using llCoMP. *Microprocessors and Microsystems - Embedded Hardware Design (In Press) 36*, 2 (Mar. 2012), 78–87.

[6] Reyes, R., Dorta, A. J., Almeida, F., and de Sande, F. Automatic hybrid MPI+OpenMP code generation with llc. In *Proc. of the 16th European PVM/MPI Users' Group Meeting* (Espoo, Finland, 2009), M. Ropo, J. Westerholm, and J. Dongarra, Eds., vol. 5759 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 185–195. 115

[7] Reyes, R., Dorta, A. J., Almeida, F., and de Sande, F. Automatic code generation for GPUs in llc. In *Proc. of the 10th International Conference on Computational and Mathematical Methods in Science and Engineering* (Almeria, Andalucia, Spain, 2010), J. Vigo-Aguiar, Ed., vol. III, pp. 804–815. 117

[8] Reyes, R., Fumero, J. J., López, I., and de Sande, F. Estrategias de optimización en diferentes arquitecturas CUDA usando llCoMP. In *Actas XXII Jornadas de Paralelismo (JP2011)* (Sept. 2011), F. Almeida, V. Blanco, C. León, C. Rodriguez, and F. de Sande, Eds., Universidad de La Laguna, pp. 671–676.

[9] Reyes, R., López-Rodríguez, I., Fumero, J. J., and de Sande, F. accULL: An OpenACC implementation with CUDA and OpenCL support. In *Euro-Par 2012 Parallel Processing - 18th International Conference, Euro-Par 2012, Rhodes Island, Greece, August 27-31, 2012. Proceedings* (Rhodes Island, Greece, Aug 2012), C. Kaklamanis, T. S. Papatheodorou, and P. G. Spirakis, Eds., vol. 7484 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 871–882. 121, 172

[10] Reyes, R., López-Rodríguez, I., Fumero, J. J., and de Sande, F. accULL: an user-directed approach to heterogeneous programming. In *Proc. of the 2012 10th IEEE International Symposium on Parallel and Distributed Processing with Applications* (Leganés, Spain, Jul 2012), IEEE Computer Society Press, pp. 654–661. 121

[11] Reyes, R., López-Rodríguez, I., Fumero, J. J., and de Sande, F. An early evaluation of the OpenACC standard. In *Proceedings of the 2012 International Conference on Computational and Mathematical Methods in Science and Engineering* (La Manga - Murcia, Spain, Jul 2012), J. V. A. et al., Ed., vol. 3, pp. 1024–1035. 121, 172

Bibliography

[12] Allen, R., and Kennedy, K. Automatic translation of Fortran programs to vector form. *ACM Trans. Program. Lang. Syst. 9*, 4 (1987), 491–542. 84, 91

[13] Allen, R., and Kennedy, K. Vector register allocation. *IEEE Trans. Computers 41*, 10 (1992), 1290–1317. 84, 91

[14] Anselmo, D., and Ledgard, H. Measuring productivity in the software industry. *Commun. ACM 46*, 11 (Nov. 2003), 121–125. 7, 8

[15] Attig, N., Gibbon, P., and Lippert, T. Trends in supercomputing: The European path to Exascale. *Computer Physics Communications 182*, 9 (2011), 2041 – 2046. 2

[16] Augonnet, C., Thibault, S., and Namyst, R. StarPU: a runtime system for scheduling tasks over accelerator-based multicore machines. Rapport de recherche RR-7240, INRIA, Mar. 2010. 57

[17] Ayguadé, E., and et al. A proposal to extend the OpenMP tasking model for heterogeneous architectures. In *IWOMP'09* (Dresden, Germany, 06/2009 2009), vol. 5568, Springer, Springer, pp. 154–167. 118, 119, 134, 172, 124, 125, 140, 179

[18] Ayguadé, E., and et al. Extending OpenMP to survive the heterogeneous multi-core era. *International Journal of Parallel Programming 38*, 5-6 (2010), 440–459. 39, 43

[19] Badia, R. M. Top down programming methodology and tools with StarSs - enabling scalable programming paradigms. In *Proceedings of the second workshop on Scalable algorithms for large-scale systems* (New York, NY, USA, 2011), ScalA '11, ACM, pp. 19–20. 13, 15

[20] Badia, R. M., Labarta, J., Marjanović, V., Martin, A. F., Mayo, R., Quintana-Ortí, E. S., and Reyes, R. Parallelizing dense matrix factorizations on clusters of multi-core processors using SMPSs. In *Parallel Computing: Proceedings of the International Conference ParCo 2011* (Apr "2012"), E. D'Hollander and D. Padua, Eds., IOS. 171, 179

[21] Badia, R. M., Perez, J. M., Ayguade, E., and Labarta, J. Impact of the memory hierarchy on shared memory architectures in multicore programming models. In *Proceedings of the 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing* (Washington, DC, USA, 2009), PDP '09, IEEE Computer Society, pp. 437–445. 171, 179

[22] Bendersky, E. Pycparse, 2009. http://code.google.com/p/pycparser/ [Online; Last accessed October 2012]. 57, 64, 76, 60, 69, 70, 83

[23] Berendsen, H. GROMACS: A message-passing parallel molecular dynamics implementation. *Computer Physics Communications 91*, 1-3 (Sept. 1995), 43–56. 5, 2

[24] Bergman, K. e. a. ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems. Tech. Rep. TR-2008-13, DARPA, 2008. http://www.cse.nd.edu/Reports/2008/TR-2008-13.pdf [Online; Last accessed October 2012]. 2

[25] Bethune, I., Carter, A., Stratford, K., and Korosoglou, P. CP2K scalable atomistic simulations for the PRACE community. Tech. rep., PRACE, 2012. 5, 2

[26] Beyer, J. C., Stotzer, E. J., Hart, A., and de Supinski, B. R. OpenMP for accelerators. In *Proceedings of the 7th international conference on OpenMP in the Petascale era* (Berlin, Heidelberg, 2011), IWOMP'11, Springer-Verlag, pp. 108–121. 11, 172, 13

[27] Bircsak, J., Craig, P., Crowell, R., Cvetanovic, Z., Harris, J., Nelson, C. A., and Offner, C. D. Extending OpenMP for NUMA machines. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)* (Washington, DC, USA, 2000), Supercomputing '00, IEEE Computer Society. 11, 13

[28] Bodin, F., Beckman, P., Gannon, D., Gotwals, J., Narayana, S., Srinivas, S., and Winnicka, B. Sage++: An object-oriented toolkit and class library for building Fortran and C++ restructuring tools. In *In The second annual object-oriented numerics conference (OON-SKI)* (1994), pp. 122–136. 51, 55

[29] Bodin, F., and Bihan, S. Heterogeneous multicore parallel programming for graphics processing units. *Sci. Program. 17*, 4 (Dec. 2009), 325–336. 172, 180

[30] Boehm, B. W., and et al. *Software Cost Estimation with COCOMO II*, 1st ed. Prentice Hall Press, Upper Saddle River, NJ, USA, 2009. 155, 166

[31] Bueno, J., Martinell, L., Duran, A., Farreras, M., Martorell, X., Badia, R. M., Ayguade, E., and Labarta, J. Productive cluster programming with OmpSs. In *Proceedings of the 17th international conference on Parallel processing - Volume Part I* (Berlin, Heidelberg, 2011), Euro-Par'11, Springer-Verlag, pp. 555–566. 39, 171, 179

[32] Butenhof, D. R. *Programming with POSIX threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997. 11, 13

[33] Cabrera, D., Martorell, X., Gaydadjiev, G., Ayguadé, E., and Jiménez-González, D. OpenMP extensions for FPGA accelerators. In *Proceedings of the 9th international conference on Systems, architectures, modeling and simulation* (Piscataway, NJ, USA, 2009), SAMOS'09, IEEE Press, pp. 17–24. 11, 13

[34] Chamberlain, B. L., Callahan, D., and Zima, H. P. Parallel programmability and the Chapel language. *Int. J. High Perform. Comput. Appl. 21*, 3 (2007), 291–312. 10, 20, 12, 13, 24

[35] Chamberlain, B. L., Choi, S.-E., Lewis, E. C., Snyder, L., Weathersby, W. D., and Lin, C. The case for high-level parallel programming in ZPL. *IEEE Comput. Sci. Eng. 5*, 3 (1998), 76–86. 20, 24

[36] Chan, S. C., Gao, G. R., Chapman, B., Linthicum, T., and Dasgupta, A. Open64 compiler infrastructure for emerging multicore/manycore architecture all symposium tutorial. In *IPDPS* (2008), IEEE, p. 1. 44, 172, 180

[37] Chapman, B., Jost, G., and Pas, R. v. d. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007. 11, 13

[38] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S.-h., and Skadron, K. Rodinia: a benchmark suite for heterogeneous computing. In *IEEE International Symposium on Workload Characterization* (Oct. 2009). 129, 157, 136, 166

[39] Che, S., Sheaffer, J. W., Boyer, M., Szafaryn, L. G., Wang, L., and Skadron, K. A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'10)* (Washington, DC, USA, 2010), IISWC '10, IEEE Computer Society, pp. 1–11. 129, 157, 136, 166

[40] Cole, M. *Algorithmic Skeletons: structured management of parallel computation*. Monograms. Pitman/MIT Press, Cambridge, MA, 1989. 16, 18

[41] Council on competitiveness and USC-ISI. Broad study of desktop technical computing end users and HPC. Tech. rep., Council on Competitiveness, US, 2008. 1

[42] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst. 13*, 4 (Oct. 1991), 451–490. 48, 52

[43] Dagum, L., and Menon, R. OpenMP: An industry-standard API for shared-memory programming. *IEEE Comput. Sci. Eng. 5*, 1 (1998), 46–55. 6

[44] Dave, C., Bae, H., Min, S.-J., Lee, S., Eigenmann, R., and Midkiff, S. Cetus: A source-to-source compiler infrastructure for multicores. *Computer 42*, 12 (2009), 36–42. 52, 56

[45] Dongarra, J. J., Cruz, J. D., Hammerling, S., and Duff, I. S. Algorithm 679: A set of level 3 basic linear algebra subprograms: model implementation and test programs. *ACM Trans. Math. Softw. 16* (March 1990), 18–28. 150, 160

[46] Dongarra, J. J., Luszczek, P., and Petitet, A. The LINPACK benchmark: Past, present, and future. *Concurrency and Computation: Practice and Experience 15* (2003), 2003. 2

[47] Dongarra, J. J., Meuer, H. W., and Strohmaier, E. Top500 supercomputer sites, 11th edition. Tech. Rep. UT-CS-98-391, 1998. 2

[48] Dorta, A. J. *Extensión del modelo de OpenMP a memoria distribuida*. PhD thesis, Universidad de La Laguna, La Laguna, December 2008. 8, 9

[49] Dorta, A. J., Badía, J. M., Quintana, E. S., and de Sande, F. Implementing OpenMP for clusters on top of MPI. In *Proc. of the 12th European PVM/MPI Users' Group Meeting* (Sorrento, Italy, September 18–21 2005), vol. 3666 of *LNCS*, Springer-Verlag, pp. 148–155. 8, 11, 15, 117, 9, 13, 17, 123

[50] Dorta, A. J., Badía, J. M., Quintana, E. S., and de Sande, F. Parallelizing dense linear algebra operations with task queues in `llc`. In *Proc. of the 14th European PVM/MPI Users' Group Meeting* (Paris, France, 2007), Lecture Notes in Computer Science, Springer-Verlag. 8, 9

[51] Dorta, A. J., González, J. A., Rodríguez, C., and de Sande, F. Towards structured parallel programming. In *Proc. Fourth European Workshop on OpenMP (EWOMP 2002)* (Rome, Italy, September 2002). 8, 9

[52] Dorta, A. J., González, J. A., Rodriguez, C., and de Sande, F. `llc`: A parallel skeletal language. *Parallel Processing Letters 13*, 3 (September 2003), 437–448. 8, 15, 115, 9, 17, 18, 121

[53] Dorta, A. J., González-Escribano, A., Rodríguez, C., and de Sande, F. The OpenMP source code repository. In *Proc. of the 13th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP 2005)* (Lugano, Switzerland, February 2005), pp. 244–250. 15, 129, 131, 17

[54] Dorta, A. J., López, P., and de Sande, F. Basic skeletons in `llc`. *Parallel Computing 32*, 7–8 (September 2006), 491–506. 8, 15, 115, 117, 9, 17, 121, 123

[55] Duran, A., and et al. OmpSs: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters 21*, 2 (2011), 173–193. 171, 172, 179, 180

[56] Eigenmann, R., and Hoeflinger, J. Parallelizing and vectorizing compilers. Tech. Rep. ECE-HPCLab-99201, Purdue University School of ECE, High-Performance Computing Lab, 2000. 6, 7

[57] El-Ghazawi, T., and Smith, L. UPC: unified parallel C. In *SC'06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing* (New York, NY, USA, 2006), ACM, p. 27. 18, 20

[58] EPCC, and the Scottish Government. Supercomputing Scotland, 2012. http://www.supercomputingscotland.org/ [Online; Last accessed October 2012]. 1, 2

[59] Fan, Z., Qiu, F., Kaufman, A., and Yoakum-Stover, S. GPU cluster for high performance computing. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing* (Washington, DC, USA, 2004), IEEE Computer Society, p. 47. 26, 27

[60] Faulk, S. e. a. Measuring high performance computing productivity. *Int. J. High Perform. Comput. Appl. 18*, 4 (2004), 459–473. 7, 8

[61] Ferrer, R., Duran, A., Martorell, X., and Ayguadé, E. Unrolling loops containing task parallelism. In *Languages and Compilers for Parallel Computing, 22nd International Workshop, LCPC 2009, Newark, DE, USA* (Oct 2009), G. R. Gao, L. L. Pollock, J. Cavazos, and X. Li, Eds., vol. 5898 of *Lecture Notes in Computer Science*, Springer, pp. 416–423. 53, 57

[62] Funk, A., Basili, V., Hochstein, L., and Kepner, J. Application of a development time productivity metric to parallel software development. In *SE-HPCS '05: Proceedings of the second international workshop on Software engineering for high performance computing system applications* (New York, NY, USA, 2005), ACM, pp. 8–12. 7, 8

[63] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, USA, 1994. 69, 82, 75, 89

[64] Gelado, I., Stone, J. E., Cabezas, J., Patel, S., Navarro, N., and Hwu, W.-m. W. An asymmetric distributed shared memory model for heterogeneous parallel systems. *SIGARCH Comput. Archit. News 38* (March 2010), 347–358. 60, 62

[65] Gorlatch, S. Send-Recv considered harmful? Myths and truths about parallel programming. In *Proc. of the 6th International Conference, PaCT 2001* (Novosibirsk, Russia, September 3-7 2001), vol. 2127 of *LNCS*, pp. 243–257. 6, 14, 7, 16

[66] Gough, B. J., and Stallman, R. M. *An Introduction to GCC*. Network Theory Ltd., 2004. 43

[67] Han, T. D., and Abdelrahman, T. S. hiCUDA: a high-level directive-based language for GPU programming. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units* (New York, NY, USA, 2009), GPGPU-2, ACM, pp. 52–61. 36, 40

[68] Han, T. D., and Abdelrahman, T. S. hiCUDA: High-level GPGPU programming. *IEEE Transactions on Parallel and Distributed Systems 22* (2011), 78–90. 36, 40

[69] HARRIS, M. Optimizing parallel reduction in CUDA. Technical report, NVIDIA, 2007. 134, 140

[70] HIGH PERFORMANCE COMPUTING CENTER STUTTGART. HLRS HomePage, 2012. http://www.hlrs.de/ [Online; Last accessed October 2012]. 2

[71] HIGH PERFORMANCE COMPUTING GROUP AT LA LAGUNA UNIVERSITY. accULL Project, 2007. http://cap.pcg.ull.es/accULL [Online; Last accessed October 2012]. 127

[72] HIGH PERFORMANCE COMPUTING GROUP AT LA LAGUNA UNIVERSITY. Group HomePage, 2012. http://gcap.pcg.ull.es [Online; Last accessed October 2012]. 8, 9

[73] HIGH PERFORMANCE COMPUTING GROUP AT LA LAGUNA UNIVERSITY. `llc` Home Page, 2012. http://llc.pcg.ull.es [Online; Last accessed October 2012]. 15, 18

[74] HIGH PERFORMANCE FORTRAN FORUM. High Performance Fortran language specification, version 1.0. Tech. Rep. CRPC-TR92225, Rice University, Center for Research on Parallel Computation, Houston, Tex., USA, 1993. 20, 24

[75] JIN, H., KELLOGG, M., AND MEHROTRA, P. Using compiler directives for accelerating CFD applications on GPUs. In *Proceedings of the 8th international conference on OpenMP in a Heterogeneous World* (Berlin, Heidelberg, 2012), IWOMP'12, Springer-Verlag, pp. 154–168. 172, 179

[76] KHRONOS GROUP. OpenCL the open standard for parallel programming of heterogeneous systems, 2008. http://www.khronos.org/opencl/ [Online; Last accessed October 2012]. 31

[77] KIM, J., SEO, S., LEE, J., NAH, J., JO, G., AND LEE, J. SnuCL: an OpenCL framework for heterogeneous CPU/GPU clusters. In *Proceedings of the 26th ACM international conference on Supercomputing* (New York, NY, USA, 2012), ICS '12, ACM, pp. 341–352. 173

[78] KISH, L. B. End of Moore's law: thermal (noise) death of integration in micro and nano electronics. *Physics Letters A 305*, 3–4 (2002), 144 – 149. 2

[79] KRESSE, G., AND HAFNER, J. *Ab initio* molecular dynamics for liquid metals. *Phys. Rev. B 47*, 1 (Jan. 1993), 558–561. 5, 2

[80] KUCK, D. J., KUHN, R. H., LEASURE, B., AND WOLFE, M. The structure of an advanced vectorizer for pipelined processors. In *Proc. 4th IEEE Int. Computer Software and Applications Conference* (Oct 1980), pp. 709–715. 84, 91

[81] LATTNER, C., AND ADVE, V. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization* (Washington, DC, USA, 2004), CGO '04, IEEE Computer Society, pp. 75–. 47, 172, 51, 180

[82] Lee, S., and Eigenmann, R. OpenMPC: Extended OpenMP programming and tuning for GPUs. In *SC'10: Proceedings of the 2010 ACM/IEEE conference on Supercomputing, Won the Best Student Paper award* (Washington, DC, USA, 2010), SC '10, IEEE Computer Society, pp. 1–11. 35, 38

[83] Lee, S., Min, S.-J., and Eigenmann, R. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming* (New York, NY, USA, 2009), ACM, pp. 101–110. 139, 147

[84] Lee, S.-i., Johnson, T. A., and Eigenmann, R. Cetus - an extensible compiler infrastructure for source-to-source transformation. In *Languages and Compilers for Parallel Computing, 16th Intl. Workshop, College Station, TX, USA, Revised Papers, volume 2958 of LNCS* (2003), pp. 539–553. 35, 38

[85] Levine, J., and John, L. *Flex & Bison*, 1st ed. O'Reilly Media, Inc., 2009. 8, 9

[86] Liao, C., Quinlan, D. J., Panas, T., and de Supinski, B. R. A ROSE-Based OpenMP 3.0 research compiler supporting multiple runtime libraries. In *IWOMP* (2010), pp. 15–28. 50, 54

[87] López, P., Dorta, A. J., Mediavilla, E., and de Sande, F. Generation of microlensing magnification patterns with high performance computing techniques. In *Applied Parallel Computing. State of the Art in Scientific Computing. Proceedings of the 8th International Workshop, PARA 2006, Umeä, Sweden, June 18-21, 2006* (Umeä, Sweden, 2007), vol. 4699 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 351–360. 8, 9

[88] Luebke, D., Harris, M., Krüger, J., Purcell, T., Govindaraju, N., Buck, I., Woolley, C., and Lefohn, A. GPGPU: general purpose computation on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Course Notes* (New York, NY, USA, 2004), ACM, p. 33. 26, 27

[89] Lusk, E., and Yelick, K. Languages for high-productivity computing: The DARPA HPCS language project. *Parallel Processing Letters 17*, 1 (2007), 89–102. 6, 7

[90] Mandelbrot, B. B. *Form, chance, and dimension. Translation of Les objets fractals: Partition*. W. H. Freeman, San Francisco, 1977. 132, 138

[91] Marjanović, V., Labarta, J., Ayguadé, E., and Valero, M. Overlapping communication and computation by using a hybrid MPI/SMPSs approach. In *Proceedings of the 24th ACM International Conference on Supercomputing* (New York, NY, USA, 2010), ICS '10, ACM, pp. 5–16. 11, 171, 13, 179

[92] Martin, R. C. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003. 64, 70

[93] McNeil, J. *Python 2. 6 Text Processing Beginner's Guide*. Packt Publishing, Limited, 2010. 89, 97

[94] Mehrara, M., Jablin, T., Upton, D., August, D., Hazelwood, K., , and Mahlke, S. Multicore compilation strategies and challenges. *IEEE Signal Processing Magazine 26*, 6 (2009), 55–63. 6, 7

[95] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Tech. Rep. UT-CS-94-230, University of Tennessee, Knoxville, TN, USA, June 1994. 6, 14, 16

[96] Meyers, R. Introducing C99. *C/C++ Users Journal 18*, 10 (Oct. 2000), 49–53. 92, 100

[97] Moore, G. E. Readings in computer architecture. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000, ch. Cramming more components onto integrated circuits, pp. 56–59. 2

[98] Muchnick, S. S. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997. 79, 196, 86, 203

[99] NDEMC. National Digital Engineering and Manufacturing Consortium, 2012. http://ndemc.ncms.org/ [Online; Last accessed October 2012]. 1

[100] Nickolls, J., Buck, I., Garland, M., and Skadron, K. Scalable parallel programming with CUDA. *Queue 6*, 2 (2008), 40–53. 42, 47

[101] Nuzman, D., and Henderson, R. Multi-platform auto-vectorization. In *Proceedings of the International Symposium on Code Generation and Optimization* (Washington, DC, USA, 2006), CGO '06, IEEE Computer Society, pp. 281–294. 6, 7

[102] NVIDIA Corp. CUDA occupancy calculator, 2007. http://goo.gl/gw98l [Online; Last accessed October 2012]. 108, 115

[103] NVIDIA Corp. Kepler compute architecture white paper. Tech. rep., NVIDIA, 2012. 24, 27

[104] NVIDIA Corp. NVIDIA visual profiler, 2012. http://developer.nvidia.com/cuda/nvidia-visual-profiler [Online; Last accessed October 2012]. 142, 149

[105] Olivier, S. L., Porterfield, A. K., Wheeler, K. B., Spiegel, M., and Prins, J. F. OpenMP task scheduling strategies for multicore NUMA systems. *Int. J. High Perform. Comput. Appl. 26*, 2 (May 2012), 110–124. 11, 13

[106] OpenACC. OpenACC directives for accelerators, 2011. http://www.openacc-standard.org [Online; Last accessed October 2012]. 39, 124, 131

[107] OpenMP Architecture Review Board. *OpenMP Application Program Interface v. 3.0*, May 2008. 92, 100

[108] OpenMP Architecture Review Board. OpenMP official web site, 2012. http://www.openmp.org [Online; Last accessed October 2012]. 6, 11, 92, 100

[109] Padua, D. A., and et al. Polaris: A new-generation parallelizing compiler for MPPs. Tech. Rep. CSRD 1306, Univ. of Illinois at Urbana-Champaign, 1993. 52, 56

[110] Parr, T.J., Q. ANTLR: a predicated-LL(k) parser generator. *Software - Practice and Experience 25*, 7 (1995), 789–810. 52, 56

[111] Planas, J., Badia, R. M., Ayguadé, E., and Labarta, J. Hierarchical task-based programming with StarSs. *IJHPCA 23*, 3 (2009), 284–299. 53, 57

[112] Pérez, J. M., Badia, R. M., and Labarta, J. A dependency-aware task-based programming environment for multi-core architectures. In *Proceedings of the 2008 IEEE International Conference on Cluster Computing, 29 September - 1 October 2008, Tsukuba, Japan* (2008), IEEE, pp. 142–151. 13

[113] Rabenseifner, R. Hybrid Parallel Programming on HPC Platforms. In *Proc. of the Fifth European Workshop on OpenMP (EWOMP 2003)* (Aachen, Germany, September 2003), pp. 185–194. 116, 122

[114] Rabenseifner, R., Hager, G., and Jost, G. Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP nodes. In *Proc. of the 17th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP 2009)* (Weimar, Germany, February 2009). 115, 122

[115] Ramanathan, R. Extending the world's most popular processor architecture. Tech. rep., Intel, oct 2006. 6, 7

[116] Reyes, R., and de Sande, F. Automatic code generation for GPUs in llc. *The Journal of Supercomputing 58*, 3 (mar 2011), 349–356. 117, 172, 123, 180

[117] Reyes, R., and de Sande, F. Case Studies in automatic GPGPU code generation with `llc`. In *Proceedings of the 2010 conference on Parallel processing* (Berlin, Heidelberg, 2011), vol. 6586 LNCS of *Euro-Par 2010*, Springer-Verlag, pp. 13–22. 117, 123

[118] Reyes, R., and de Sande, F. Extending `llc` to support hierarchical clusters. In *Proc. of the Work in Progress special session on the 19th Euromicro International Conference on Parallel, Distributed and Network-Based Computing (PDP 2011)* (Cyprus, Feb 2011), pp. 21–22. 119, 125

[119] Reyes, R., and de Sande, F. Optimize or wait? using llc fast-prototyping tool to evaluate CUDA optimizations. In *Proc. of the 19th Euromicro International Conference on Parallel, Distributed and Network-Based Computing (PDP 2011)* (Cyprus, Feb 2011), pp. 257–261. 172, 180

[120] Reyes, R., and de Sande, F. `YACF`: Yet Another Compiler Framework, 2012. http://code.google.com/p/yacf/ [Online; Last accessed October 2012]. 172, 180

[121] Reyes, R., Dorta, A. J., Almeida, F., and de Sande, F. Automatic hybrid MPI+OpenMP code generation with `llc`. In *Proc. of the 16th European PVM/MPI Users' Group Meeting* (Espoo, Finland, 2009), M. Ropo, J. Westerholm, and J. Dongarra, Eds., vol. 5759 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 185–195. 115, 121

[122] Reyes, R., Dorta, A. J., Almeida, F., and de Sande, F. Automatic code generation for GPUs in `llc`. In *Proc. of the 10th International Conference on Computational and Mathematical Methods in Science and Engineering* (Almeria, Andalucia, Spain, 2010), J. Vigo-Aguiar, Ed., vol. III, pp. 804–815. 117, 123

[123] Reyes, R., López-Rodríguez, I., Fumero, J. J., and de Sande, F. accULL: An OpenACC implementation with CUDA and OpenCL support. In *Euro-Par 2012 Parallel Processing - 18th International Conference, Euro-Par 2012, Rhodes Island, Greece, August 27-31, 2012. Proceedings* (Rhodes Island, Greece, Aug 2012), C. Kaklamanis, T. S. Papatheodorou, and P. G. Spirakis, Eds., vol. 7484 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 871–882. 121, 172, 127, 180

[124] Reyes, R., López-Rodríguez, I., Fumero, J. J., and de Sande, F. accULL: an user-directed approach to heterogeneous programming. In *Proc. of the 2012 10th IEEE International Symposium on Parallel and Distributed Processing with Applications* (Leganés, Spain, Jul 2012), IEEE Computer Society Press, pp. 654–661. 121, 127

[125] Reyes, R., López-Rodríguez, I., Fumero, J. J., and de Sande, F. An early evaluation of the OpenACC standard. In *Proceedings of the 2012 International Conference on Computational and Mathematical Methods in Science and Engineering* (La Manga - Murcia, Spain, Jul 2012), J. V. A. et al., Ed., vol. 3, pp. 1024–1035. 121, 172, 127, 180

[126] Rodríguez-Rosa, J., Dorta, A. J., Rodríguez, C., and de Sande, F. Exploiting task and data parallelism. In *Proc. of the Fifth European Workshop on OpenMP (EWOMP 2003)* (Aachen, Germany, September 2003), pp. 107–116. 8, 9

[127] Roth, G., Mellor-Crummey, J., Kennedy, K., and Brickner, R. G. Compiling stencils in High Performance Fortran. In *Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM)* (New York, NY, USA, 1997), Supercomputing '97, ACM, pp. 1–20. 144, 152

[128] Ryoo, S., and et al. Program optimization space pruning for a multithreaded GPU. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization* (New York, NY, USA, 2008), CGO '08, ACM, pp. 195–204. 34, 38

[129] Ryoo, S., Rodrigues, C. I., Baghsorkhi, S. S., Stone, S. S., Kirk, D. B., and Hwu, W.-m. W. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming* (New York, NY, USA, 2008), ACM, pp. 73–82. 24, 26

[130] Sabne, A., Sakdhnagool, P., and Eigenmann, R. Effects of compiler optimizations in OpenMP to CUDA translation. In *Proceedings of the 8th international conference on OpenMP in a Heterogeneous World* (Berlin, Heidelberg, 2012), IWOMP'12, Springer-Verlag, pp. 169–181. 172, 179

[131] Schreiber, R., and Dongarra, J. Automatic blocking of nested loops. Technical report CS-90-108, NASA Ames Research Center, may 1990. 85, 92

[132] Simionato, M. *An Introduction to GraphViz and dot*. O'Reilly Community Press, 2004. 90, 98

[133] Skillicorn, D. B., and Talia, D. Models and languages for parallel computation. *ACM Comput. Surv. 30*, 2 (June 1998), 123–169. 10, 12

[134] Swiss National Supercomputing Centre (CSCS). Annual report, 2011. http://www.cscs.ch/fileadmin/Documents/reports/Annual_Report11.pdf [Online; Last accessed October 2012]. 5, 2

[135] Swope, W. C., Andersen, H. C., Berens, P. H., and Wilson, K. R. A computer simulation method for the calculation of equilibrium constants for the formation of physical clusters of molecules: Application to small water clusters. *Journal of Chemical Physics 76* (1982), 637–649. 136, 142

[136] Takeuchi, M., and et al. Compiling X10 to Java. In *Proceedings of the 2011 ACM SIGPLAN X10 Workshop* (New York, NY, USA, 2011), X10 '11, ACM, pp. 3:1–3:10. 19, 22

[137] Thakur, R., and et al. MPI at Exascale. In *Procceedings of SciDAC 2010* (Jun. 2010). 2

[138] The HadGEM2 Development Team. The HadGEM2 family of Met office unified model climate configurations. *Geoscientific Model Development 4*, 3 (2011), 723–757. 5, 2

[139] UK National Supercomputing Service. HECToR annual report, 2011. http://www.hector.ac.uk/about-us/reports/annual/2011.pdf [Online; Last accessed October 2012]. 5, 2

[140] USA President's Information Technology Advisory Committee. Computational Science: Ensuring America's Competitiveness, 2005. http://www.nitrd.gov/pitac/reports/20050609_computational/computational.pdf [Online; Last accessed October 2012]. 1

[141] Watkins, D. S. *Fundamentals of matrix computations*, second ed. John Wiley and Sons, New York, 2002. 147, 157

[142] Wheeler, D. SLOCcount, 2009. http://www.dwheeler.com/sloccount/ [Online; Last accessed October 2012]. 155, 165

[143] Willhalm, T., and Popovici, N. Putting Intel Threading Building Blocks to work. In *Proceedings of the 1st international workshop on Multicore software engineering* (New York, NY, USA, 2008), IWMSE '08, ACM, pp. 3–4. 11, 13

[144] Wolfe, M. Iteration space tiling for memory hierarchies. In *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing* (Philadelphia, PA, USA, 1989), Society for Industrial and Applied Mathematics, pp. 357–361. 85, 92

[145] Wolfe, M. More iteration space tiling. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing* (New York, NY, USA, 1989), Supercomputing '89, ACM, pp. 655–664. 85, 92

[146] Wolfe, M. Implementing the PGI Accelerator model. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units* (New York, NY, USA, 2010), GPGPU '10, ACM, pp. 43–50. 37, 38, 154, 41, 163

[147] Xue, J. *Loop Tiling for Parallelism*. Kluwer International Series in Engineering and Computer Science. Kluwer Academic, 2000. 84, 85, 92

[148] Yelick, K., and et al. Productivity and performance using partitioned global address space languages. In *PASCO '07: Proceedings of the 2007 international workshop on Parallel symbolic computation* (New York, NY, USA, 2007), ACM, pp. 24–32. 18, 20

List of Figures

1.1 Evolution of the overall performance of HPC systems since 1993 2
1.2 Evolution of the number of cores per socket per system since 2000 3
1.3 Evolution of the number of systems with accelerators since 2007 4
1.4 Usage of HECToR facilities by area of expertise 5

2.1 Fork join model 12
2.2 Comparison of CPU and GPU devices 22
(a) Memory Bandwith 22
(b) Floating-point performamce 22
2.3 Comparison of the old Work Distributor used in Fermi with the new Grid Management Unit used in the new Kepler architectures 25
2.4 CUDA programs, partitioned into blocks of threads, can run on different GPU architectures without being modified. Depending on the available number of resources, more or less blocks may run in parallel 27
2.5 Example of a two-dimensional kernel grid 29
2.6 Memory hierarchy of the CUDA architecture 30
2.7 Execution of a GPU program from the Host 32
2.8 OpenCL Platform model 33
2.9 OpenCL Execution model 34
2.10 OpenMPC workflow 35
2.11 Task dependency graph generated by the OmpSs runtime 40
2.12 Compilation Flow of Open64 45
2.13 Levels of WHIRL and Lowering actions 46
2.14 LLVM Workflow 47
2.15 Usual Workflow of a ROSE driver 50
2.16 Cetus architecture 53
2.17 Mercurium workflow 55
2.18 The GNU compiler toolkit translates the OpenMP nodes into calls to GOMP . 59

3.1 Overall translation workflow executed by a typical `YaCF` driver 63
3.2 IR for the statement `if(a>1) printf("%d",a)` 67
3.3 IR generated for the main loop of the π computation example shown in Figure 2.1 68
3.4 Insertion of a new parameter in a function call 71

3.5 Replace operation of a parameter in a function call 72
3.6 Remove operation of a parameter in a function call 75
3.7 FRONTEND package class hierarchy 77
3.8 Variable dependency graph for the code in Listing 3.12 81

4.1 Basic pre-DCG representing a variable in the device that needs to be updated with data from the Host 98
4.2 Evolution of the three graphs (correlation, pre- and post- graphs) across an example program execution on a platform with a Host and an attached device . 100
4.3 Evolution of the three graphs (correlation, pre- and post- graphs) across a Host with a device attached performing a reduction on its private memory 101
4.4 Overview of `Frangollo` layers 103
4.5 Transition state diagram of a *Var* instance 105
4.6 UML Diagram of the most relevant classes within the `Frangollo` runtime . . . 107
4.7 Execution workflow of the `Frangollo` runtime through the same example as that shown in Figure 4.2 113

5.1 `llc` translation options 117
5.2 Compilation flow of an `llcl` source. 122
5.3 Execution workflow of a program using the OpenACC API 123
5.4 Relationship between OpenACC concepts and elements in the GPU architecture 125

6.1 Speed-up of the Mandelbrot set computation on *Tajinaste* 133
6.3 Execution time of the implementation greatly varies in terms of the number of threads, using $N = 32768$ points. In addition, the optimal number of threads varies from Tesla C1060 to Tesla C2050. This clearly reflects the significance of a proper estimation of the kernel launch configuration 138
6.4 Speedup of the MD simulation code for different parallelization strategies, using Tesla C1060 and eight OpenMP threads in *Garoe*, one on each processor core . 140
6.5 Performance comparison of baseline, unrolling and collapsing in the evaluated platforms using *Garoe* 141
6.6 Performance comparison of the best OpenACC implementation vs. OpenMP in *Homero* for three different problem sizes. The laptop GPU memory was not large enough to handle the size of the biggest problem and the laptop crashed 143
6.7 Time execution comparison for the MD OpenACC implementation against OpenMP in *Drago* 144
6.8 Speedup of the Jacobi code for different problem sizes, using the Tesla C1060 of *Garoe* 147
6.9 Imapct on performance of the `collapse` clause in the Jacobi code using the Tesla C1060 of *Garoe* 148
6.10 Performance impact of local storage usage using three different GPU architectures in *Ilion* for the GPU card with arch 1.1, the Tesla C1060 of *Garoe* for the architecture 1.3 and the Tesla C2050 for the architecture 2.0 151

6.11 Performance comparison of three OpenACC implementations using the Tesla C2050 of *Garoe* . PGI version 12.6, CAPS HMPP version 2.3.3 and the release 0.1 of `accULL` have been used . . . 153
6.12 Effect of varying the values of the `gang`, `worker` and `vector` clauses using the Tesla C2050 of *Garoe* . . . 154
6.13 Floating point performance for MxM in *Peco*. We compare OpenMP, Frangollo with CUDA/OpenCL and an improved version of the kernel using loop invariant optimization for `a[i*L+j]` . . . 156
6.14 Comparison between OpenMP gcc implementation and `Frangollo`+OpenCL in *Drago* platform using the full system (40 cores) for MxM . . . 157
6.15 Performance comparison of `accULL` versus native implementation, showing the speedup against OpenMP using the Telsa C2050 of *Garoe* . . . 160
6.16 Performance comparison of `SRAD` using *Drago* comparing `accULL` with the native implementation, showing the speedup against OpenMP GCC implementation . 162
6.17 Comparison of execution time using `hiCUDA`, `PGI Accelerator Model`, OpenMP and the `accULL` OpenACC implementation of the `LUD` using the Tesla C2050 of *Garoe* . . . 163
6.18 Percentage of the time of the native CUDA implementation with three different OpenACC compilers of the `LUD` using the Tesla C2050 of *Garoe* . . . 164
6.19 Time relative to CUDA for the `HS` implementation using the Tesla C2050 of *Garoe* 165
6.20 Performance of the `PF` implementation using the Tesla C2050 of *Garoe* . . . 167
6.21 Performance comparison of `accULL` using the Tesla C2050 of *Garoe* versus native implementation, showing the speedup against OpenMP . . . 168
6.22 Performance comparison of `NW` using *Drago* comparing `accULL` against the provided OpenCL implementation running on the CPU, showing the speedup against OpenMP gcc implementation . . . 169
6.23 Performance comparison of `NW` using the Tesla C2050 of *Garoe* . . . 170

List of Tables

2.1 CUDA Architecture comparison 23
2.2 Final comparison of compilers 57

3.1 Information extracted from the loop in Listing 3.13 82
3.2 Placement of variables according to loop information 93

5.1 Compliance with the OpenACC 1.0 standard (directives) 128
5.2 Compliance with the OpenACC 1.0 standard (API) 128

6.1 Speedup for the MD algorithm in Tajinaste 136
6.2 Time per phase and speedup for each optimization for MD using OpenACC in the CPU 142
6.3 Time per phase and speedup for each optimization for MD using OpenACC in the GPU 146
6.4 `sloccount` data for cases of study. 158

List of Listings

2.1 Implementation of the π computation using OpenMP . . . 12
2.2 Implementation of the π computation using MPI . . . 14
2.3 Implementation of the π computation using `llc` . . . 17
2.4 A parallelization of the USMV operation . . . 17
2.5 Implementation of the π computation using `UPC` . . . 19
2.6 Example of `X10` source code . . . 21
2.7 Jacobi algorithm implementation in `Chapel` . . . 23
2.8 Example of CUDA Runtime code . . . 28
2.9 Sketch of `MxM` in hiCUDA . . . 37
2.10 Sketch of `MxM` in PGI Accelerator Model . . . 38
2.11 Example of OmpSs directives . . . 40
2.12 Open64 Tree Traversal example . . . 47
2.13 LLVM Code Representation . . . 49
2.14 ROSE driver example . . . 51
2.15 ROSE Traversal example . . . 52
2.16 Java code to create a driver using the Cetus API . . . 54
2.17 Java code to iterate through the Cetus IR and print a message whenever a `for` loop is traversed . . . 54
2.18 *Loop unrolling* in Mercurium . . . 56
2.19 *Loop collapse* in Mercurium . . . 56
2.20 Example of the usage of the `single` construct of `OpenMP` . . . 59
2.21 Code generated by GCC for the `single` construct, calling the GOMP ABI . . . 59
3.1 A simple implementation of a *Filter* that will iterate throught all the declarations of a given subtree . . . 65
3.2 A more complex example of *Filter* where only those declarations inside a particular function will be traversed . . . 66
3.3 Example of a *Mutator* that will apply a transformation to all declarations within a subtree . . . 67
3.4 Inserting a subtree inside the main IR . . . 70
3.5 Replace a subtree inside the main IR . . . 70
3.6 Initialization of a *SymbolTable* . . . 71
3.7 Content of the ST after analyzing the code in Listing 3.8 . . . 74
3.8 C code example with nested declaration scopes . . . 76

3.9 Part of the C99 AST configuration file 78
3.10 Extract from the OpenMP parser of YaCF showing the interpretation of the reduction and nowait clauses 78
3.11 Control and data dependency example extracted from [98] 79
3.12 Example of a SESE block statement with variable dependencies 79
3.13 A canonical C loop 82
3.14 Loop nest before aplying *loop interchange* 83
3.15 The result of applying *loop interchange* to the loop nest in Listing 3.14 83
3.16 An example of a simple C canonical loop *zeroing* an array 84
3.17 Loop in Listing 3.16 after aplying strip-mining with strip size B 84
3.18 Square matrices product, example of a perfect loop nest 85
3.19 Square matrices product with its loops incorrectly ordered after applying strip-mining 85
3.20 The effect of applying square *loop tiling* with tile size B to the code in Listing 3.18 86
3.21 An example of a loop body with a break statement 87
3.22 Incorrect extraction of the loop body. The break statement is no longer syntactically correct. 88
3.23 Example of template mixing C and Python code with the template tags 90
3.24 Calling the *parse_snippet* from the *Mutator* to generate the AST of the code after filling the template 90
3.25 Example of the *DOT* Back end output 91
4.1 Platform-independent update of a host variable 105
4.2 copy_in method from OCLVar 108
4.3 Example of calls to the Frangollo interface layer. 110
4.4 Calling Frangollo from C++ 111
5.1 Snippet from the code generated by llCoMP-HYB 116
5.2 OpenMP implementation of π computation using the target extension. 118
5.3 llCoMP template for a CUDA kernel 119
5.4 A platform description file 120
5.5 Matrix product in llcl 120
5.6 Code example showcasing the usage of the host_data directive 125
6.1 OpenMP implementation of the Mandelbrot set computation 132
6.2 Using the OpenMP extensions to port a code to CUDA 134
6.3 The Mandelbrot set computation in OpenACC 136
6.4 Sketch of the MD simulation 137
6.5 Main compute loop of the MD code simulation in OpenMP 139
6.6 Code of the OpenMP implementation of the Jacobi method 145
6.7 Iterative loop in Jacobi showing the usage of the OpenMP extensions 145
6.8 LU reduction in llc 149
6.9 Kernel generated automatically by llCoMP for the LU reduction 150
6.10 Kernel modified to take advantage of the thread block shared memory 150
6.11 Sketch of MxM in OpenACC 152
6.13 Main loop of the SRAD implementation in OpenACC. 159
6.14 Main loop of the LUD implementation in OpenACC 161