An innovative compilation tool-chain for embedded multi-core architectures

M. Torquati*[1], M. Vanneschi[1], M. Amini[2], S. Guelton[2], R. Keryell[2], V. Lanore[2], F.-X. Pasquier[2], M. Barreteau[3], R. Barrère[3], T. Petrisor[3], E. Lenormand[3], C. Cantini[4], and F. De Stefani[4]

[1]*Computer Science Department, University of Pisa, Italy*
[2]*HPC Project, Paris Area, France*
[3]*Thales Research & Technology, Paris Area, France*
[4]*SELEX Sistemi Integrati, Rome, Italy*

Abstract

In this paper, we propose a compilation tool-chain supporting the effective exploitation of multi-core architectures offering hundreds of cores. The tool-chain leverages on both the application requirements and the platform-specific features to provide developers with a powerful parallel-programming environment able to generate efficient parallel code.

The design of parallel applications follows a semi-automatic approach enabling the programmer to transfer to back-end tools platform-specific code generation and optimization, thus making possible to avoid the clobbering of code with non-portable and complex directives. The programmer can graphically parallelize the application (mainly data-streaming ones) for the target platform using Thales' SPEAR Design Environment. The resulting parallelization is generated under the form of an Intermediate Representation, which is then passed to the back-end tools (HPC Project's PAR4ALL) that generates efficient target code.

We present the results obtained parallelizing a small subset of the RT-STAP radar algorithm and the Chirp filtering algorithm on standard multi-core and on nVidia GPUs.

1 Introduction

During the last few years, computer hardware manufacturers have moved decisively to multi-core chips. Instead of increasing clock frequency and boosting Instruction Level Parallelism (ILP) for a single chip, processor engineers are adding more and more cores in a single chip in order to increase the performance per watt ratio.

This trend also embraced embedded systems. In fact, the increasing processing demands placed by embedded applications together with the need to reduce power consumption, have opened the route to the wide adoption of high-density multi-core architectures also in embedded systems.

Multi-core embedded architectures have shown the possibility to obtain a good balance between high-performance and low power requirements, but most applications do not exploit fully the potential of these architectures. This situation is in part due to the limited availability of parallel software, compiler technology and development tools, and in part to the fact that multi-core programming is still perceived as a niche of the high-performance-computing area, hence reserved to high specialized software engineers.

The scarcity of good high-level programming tools and environments, forces application developers to rewrite the sequential programs into parallel software taking into account all the low level

*Corresponding Author e-mail: `torquati@di.unipi.it` features and peculiarities of the underlying platforms. This makes developing efficient parallel applications extremely difficult and time consuming, and most of all, the resulting code is difficult to maintain and to port on next generation architectures.

A useful parallel development environment should offer a well defined programming models and easy-to-use development tools. The programming model provides a higher abstract level allowing to control the execution, communications and synchronization of concurrent entities, whereas the development tools help developers to write, debug and deploy the parallel applications.

In this paper, we propose a compilation tool-chain, devised within the SMECY project [1], supporting the effective exploitation of multi-core architectures. The tool-chain leverages on both the application requirements and the platform-specific features to provide developers with a powerful parallel-programming environment able to generate efficient parallel code without significant effort for the programmer. The tool-chain proposed is well-suited for applications that compute continuous streams of data, as for example, the ones in digital signal processing. The programmer, graphically decomposes the application for the target platform using Thales' Spear Design Environment (SpearDE) [2], which offers a graphical interface for describing computational kernels interacting in a data-flow application. Parallelism is explicitly exposed to the programmer, who has to decide a suitable decomposition of the application into modules as well as a mapping of each module onto elements of a graphical model of the target platform. This way, the programmer is able to transfer to back-end tools all information needed in order to produce efficient target code in terms of performance and power consumption. The resulting parallelization is generated under the form of an Intermediate Representation (IR). This design choice enhances tool-chain portability giving the possibility to clearly decouple front-end and back-end tools. In the proposed tool-chain, the back-end tool is HPC Project's Par4All [3]. Starting from the IR generated by SpearDE, Par4All is able to generate efficient target code both in terms of performance and power consumption for a set of multi- and many-cores platforms.

The rest of the paper is organized as follows: in Sect. 2 we summarize tool-chain motivations and design principles. A brief description of SpearDE and Par4All is given in Sect. 3. The IR is described in Sect. 4. Sect. 5 defines the applications selected to test the tool-chain, whose results are reported in Sect. 6. Finally, we report related work in Sect. 7 and summarize the contributions of the paper in the Conclusions section.

2 Tool-chain motivations and design principles

When designing a parallel application, both its requirements and the main characteristics of the target platform have to be taken into account in a synergistic way. As a consequence, a parallel programming tool-chain have to take as input parameters both the application and the high-level platform models and then it should be able to determine all the relevant transformations and mapping decisions on the concrete platform minimizing user intervention.

Two distinct but related needs exist: on the one hand, increasingly efficient mechanisms supporting correct concurrent execution and shared data structures accesses have to be developed; on the other hand, there is a need for higher level programming environments capable of hiding the difficulties related to the correct and efficient use of memory hierarchy and low level code generation.

In order to obtain a good balance between performance and portability, an Intermediate Representation (IR) has been introduced between front-end and back-end tools as depicted in Fig. 1. IR allows to increase code portability enabling also intermediate code inspection and editing in order to simplify both debugging and low-level code optimizations.

Even if both SpearDE and Par4All are able to parallelize data-streaming applications as stand-alone tools, combining them takes benefit from their complementarity.

SpearDE is able to model both the application and the architecture at high level enabling to focus on domain-specific semantics (i.e. signal processing in our case) and not on its implementation level (e.g. C language and low level communications). Thus parallelization is simplified because the users master the objects to be parallelized and rely on their experience to graphically

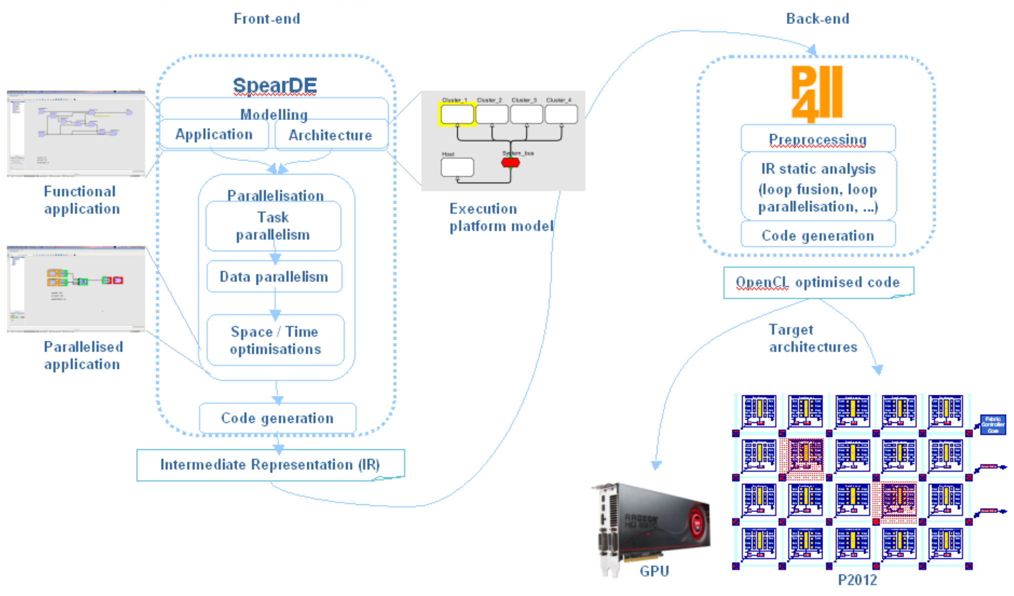

Figure 1: High-level scheme of the proposed tool-chain.

test their parallelization choices.

Par4All is very efficient for low-level optimizations and code generation purposes, especially when it collects coding rules (e.g. no pointers) and hints (e.g. allocation of functions onto resources) that simplify its work to apply parallelization operations.

Hence the user exploits the high level modeling to experiment various parallelization schemes performing design space exploration; the chosen ones are then translated into compilation directives (pragmas) and inserted into the IR according to coding rules that perfectly suit for the back-end tool. Then this IR is interpreted by the back-end parallelizing compiler to generate an efficient code on the chosen target.

3 SpearDE and Par4All brief description

SpearDE [2] is a graphical model-based design environment which helps the user to implement his data-streaming applications onto parallel architectures thanks to a user-oriented parallelization process. It enables to capture (or to import) both the functional application (graph whose tasks iterate on kernels) and the target parallel architecture (topology between processing, memory and communication units).

The parallelization process (see the left-hand side of Fig. 1) consists in the following stages: allocating computation tasks to hardware resources (task parallelism); partitioning arrays on the multiple selected resources (data parallelism); performing space and time optimizations such as overlapping of computation and communication tasks.

From the underlying parallelization results (that point out an optimized memory addressing and a parallelized scheduling), SpearDE is able to produce a glue code that calls kernels (potentially library operations). Hence only kernels have to be maintained; the remaining glue code (that is error-prone and tedious to write) is taken into account by the tool itself. It enables the user to concentrate on the different parallelization schemes to assess on the architecture sizing (this is not feasible by hand in a reasonable time).

Par4All is an automatic parallelizing compiler. Automatic parallelization has been a major failure from the past decades because of the too high expectation involved and automatic parallelization of any program is an intractable issue. Since parallel programming needs some kind of program organization and cleaning, we believe that sequential programming with some principles

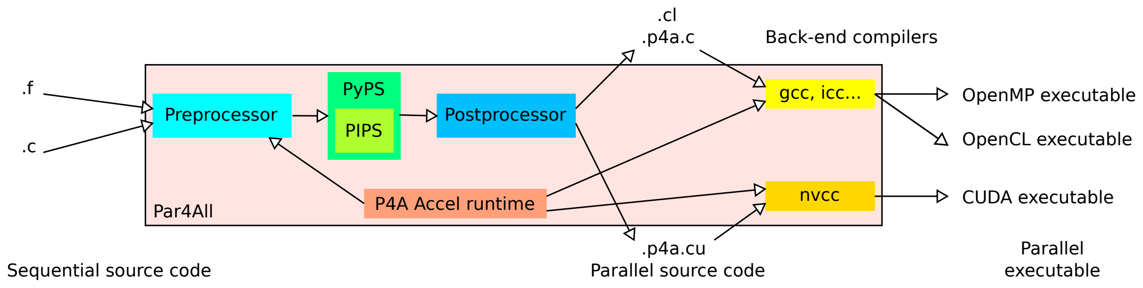

Figure 2: PAR4ALL conceptual architecture.

can be a tractable issue. Explicit parallel programming using a parallel language with parallel constructs often require reorganizing the code. But this work can be done with a sequential language too and lead to a program that can be more easily parallelizable. This is the aim of the PAR4ALL project: parallelizing well written sequential programs. An example of coding rules is for example to use more C99 variable-length array instead of using a lot of pointers and array linearizations.

PAR4ALL uses PIPS [4] as its parallelizer engine. This is a compiler framework dedicated to source-to-source translation that is interesting to be more target-independent by relying on the vendor software for the back-ends. PIPS is one of the projects that introduced polyhedral model-based compilation but with a focus on handling complete programs instead of optimizing only small pieces of code. Coupled with the PAR4ALL Accel runtime that provides the adaptation layer for different accelerators (CUDA, OpenCL, SCMP, Ter@pix [5]), PAR4ALL can be used to program different kinds of accelerators without needing to dig too much into architectural details inside PAR4ALL.

The global architecture is described in Fig. 2. PAR4ALL and its coding rules can be downloaded from the PAR4ALL site [3].

4 SME-C intermediate representation

The concept of a unique intermediate representation is quite interesting since it means having only n translators to interconnect n different tools with this intermediate representation, instead of having $n(n-1)$ direct translators from each tool to the other ones. Unfortunately it is difficult to design such an intermediate representation, not because of syntactic difficulties, but because of the different semantics each tool encompasses, each one in its very domain.

Instead of designing a brand new universal domain specific language, we have adopted a pragmatic approach with the C language[1] seen as an executable semantics with a sequential memory equivalence. This allows at least to describe the basic sequential parts of the applications that can be compiled, executed, tested, debugged without any specific tool.

From this, we adopted the `#pragma`tic view of OpenMP by adding some pragma to express that some parts of the program can be executed in parallel without impairing the sequential semantics. Since some aspects such as the mapping of some functions to specific hardware accelerators or the definition of streaming loops with some well defined pipeline execution are needed, we have complemented the OpenMP pragma with a few other SME-C pragma, such as:

```
#pragma smecy map(hardware[, unit])
  some_function_call(...);
```

To cope with specific hardware constraints in embedded systems such as data size, saturated arithmetic, different memory spaces, etc. we rely on the TR 18037 Embedded C standard extension. To describe different processes communicating together in an asynchronous way, we do not have anymore a sequential equivalence and then do not use pragma to express this. So we

[1]This explains the name SME-C for this intermediate representation for the SMECY project.

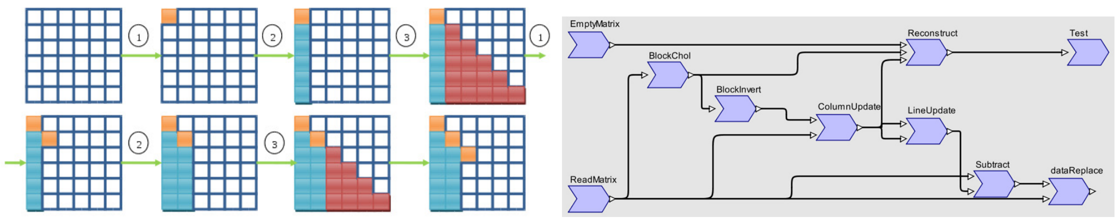

Figure 3: Steps of the right-looking block Cholesky factorization (left). SpearDE's application graph of the algorithm (right).

use a simple communication, synchronization and threading API. Since we target embedded systems with a light efficient implementation, we rely on the standard APIs from the MultiCore Association: MCAPI (communication), MRAPI (synchronization) and MTAPI (threading) [6].

At the higher level for global architecture and process organization, we rely on meta-data described in an XML file.

With this pragmatic approach embracing many different programming models, it is possible to represent all the programming models from the SMECY project and quite more. Thus we may overrun the objectives and this is why we rely on principled programming here too to keep the output programming style clean enough to preserve the ability to interconnect the tools.

5 Application scenarios

The proposed tool-chain has been designed mainly for data-streaming applications as those in digital signal processing. In this section we briefly describe two representative applications of this domain: a subset of the RT-STAP radar algorithm, and the Chirp filtering algorithm.

5.1 RT-STAP and Cholesky decomposition

Real Time Space Time Adaptive Processing (RT-STAP) computation is a good candidate to evaluate tool-chains. A detailed description of the whole STAP algorithm can be found in [7]. Here we give just a brief overview of the basic phases of this algorithm pointing out the main performance problems. The STAP algorithm can be schematized in four different phases: 1) a pre-processing phase which prepares the raw data coming from the antenna; 2) a Doppler processing phase, which eliminates ground clutter from the received echo signals; 3) the filtering phase, which consists of two main parts: i) the calculation of a set of covariance matrices from the received input samples; ii) and the calculation of adaptive filtering weights through the resolution of a set of linear equations; 4) a post-processing phase, in which the weights are applied to the echo signals.

The filtering phase is the most critical and challenging for real-time computation. In the filtering phase, the resolution of a set of linear systems is the most computationally demanding task. In this paper we consider only the parallelization of the resolution of the linear system using the Cholesky algorithm.

As sketched in Fig. 3 left-hand side, the implementation of the block Cholesky algorithm (right-looking method) follows three steps: 1) compute the Cholesky factorization of one diagonal block; 2) update the remaining blocks in the corresponding column of this diagonal block; 3) once the updated column block has become available update the remaining blocks under the diagonal on the corresponding row. Note that, while the diagonal blocks need to be computed in a sequential order, the column blocks can be updated in parallel. Similarly, once one column has been updated, the elements in the corresponding rows can equally be updated in parallel.

The application graph for the Cholesky algorithm in SpearDE is given in Fig. 3 right-hand side. In SpearDE a node in the graph will execute a so-called Elementary Task (ET) into one or several static affine nested loops. The ET represents a basic operation or a computation pattern

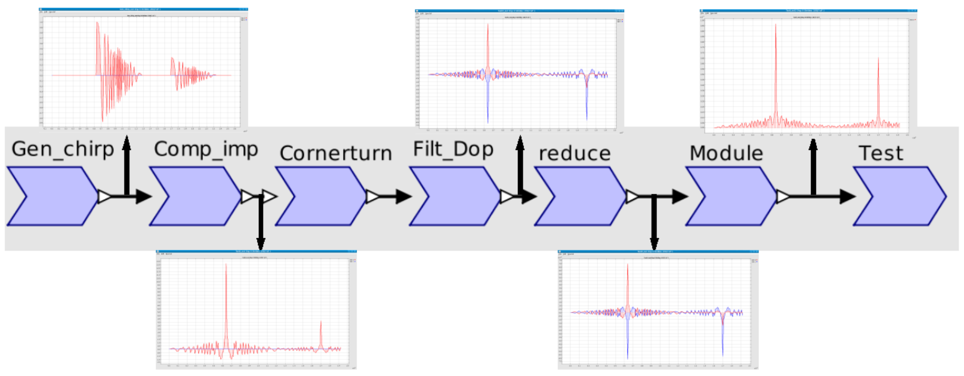

Figure 4: Chirp application with related signals

(e.g. matrix multiplication, convolution, etc.) that does not need (since already optimized) or cannot be parallelized. It is executed on a possibly smaller amount of data from the input signal. The nodes EmptyMatrix, Reconstruct and Test serve only for easier data handling, display and result comparison with reference output file.

5.2 Chirp filter

The Chirp application is a simplified radar signal processing chain, whose goal is to detect radar echoes from signal samples provided in the form of a multi-dimensional (3D in this example) array. Despite its simplicity, it is representative of the front-end filtering phase that involves the largest part of computation power in a majority of radars.

If we ignore the first (`Gen_chirp`) and last (`Test`) tasks that respectively simulate radar inputs (a multidimensional array of range gates, pulses and bursts) and compare results with correct ones, the others match usual radar operations:

- `Comp_imp`: the Pulse Compression detects (among the received - generated here - signals) any waveforms as potential echos; such a waveform on a range gate yields an energy peak. The underlying operation is a convolution one.
- `Cornerturn`: a corner-turn is a communication task in the sense that it produces no new data: it reorganizes data of the previous output array in such a way that the next task will read them in another order (as expected). Hence dimensions are permuted to efficiently feed the following task.
- `Filt_Dop`: the Doppler Filtering consists in transforming pulses from the time to the frequency domains (via FFT). It aims at identifying targets whose signal moves in a regular way and will generate a peak whenever the frequency matches.
- `reduce` deals with the amplitude: it multiplies the real and imaginary parts by a given parameter.
- `Module` simply computes the module of the resulting signal.

Generated signals for nodes in the application graph are depicted in Fig. 4. These provide help in functional debugging/testing.

6 Experiments

In order to validate our tool-chain, we considered two kinds of many-cores architectures: $x86$+GPU and the STMicroelectronics's P2012 platform[2]. The P2012 architecture comprises a number of

[2]The P2012 platform is one of the two reference architectures of the SMECY project.

decoupled STxP70 processors clusters interconnected by a Network on Chip with specific hardware for management of efficient communications. At time of writing, is available from ST Microelectronics only a functional simulator on an $x86$ platform able to simulate real processor instructions.

There is also an OpenCL compiler and runtime available for the P2012 architecture but, to be efficient, the OpenCL code has to be written in very different way from classical GPU. Whereas OpenCL code for GPU must have many work-items per work-group to be less sensitive to the global memory latency, OpenCL for P2012 only accept 1 work-item per work-group, as possible in the OpenCL standard.

Since our tool-chain cannot yet generate this kind of very specific OpenCL code, we validate the tool-chain on standard $x86$ multi-core and on nVidia GPU.

6.1 Parallelization schemes

Ways of parallelizing these applications are relatively restricted if we consider $x86$+GPU or P2012 clusters: sections to parallelize are allocated to accelerators. It can be refined when targeting P2012 because one can consider different processors that are included into any cluster and not only clusters as a whole. One can change the blocksize to influence the number of iterations. For a given input matrix of size 128x128 samples, we can try different block sizes according to the data parallelism potential of the target platform: i) a block size of 16x16 should take benefit of the P2012 platform because each cluster includes 8 cores that could perform 8 iterations each (for the `LineUpdate` iteration space of Cholesky for instance); ii) respectively a block size of 2x2 should suit for GPUs because it maximises the number of iterations (64x64) to be executed.

An example of an automatically generated IR by SpearDE in the first case considering the Chirp code is given below. SpearDE generates an (executable) OpenMP-based code that includes:

- OpenMP directives that mention parallel tasks (with the related parallel dimensions):
```
#pragma omp parallel for private (i__ApplicationModel_F_F_0)
for (i__ApplicationModel_F_F_0 = 0; i__ApplicationModel_F_F_0 < 32;
     i__ApplicationModel_F_F_0++) {
     // Call to the kernel convol_CF
     convol_CF(..., IN1[i__ApplicationModel_F_F_0],
                       ..., OUT1[i__ApplicationModel_F_F_0]); }
```
  and sections (for each processor running in parallel):
```
#pragma omp parallel sections
{ // Section for GPU #0
  #pragma omp section
  {  int i__ApplicationModel_F_0;
      for (i__ApplicationModel_F_0 = 0; i__ApplicationModel_F_0 < 4;
           i__ApplicationModel_F_0++) {
      ... /* calls to functions */ ...
```
- allocation pragmas that indicate on which hardware resource each task is executed:
```
#pragma smecy map(GPU, 0)  arg(3, [32][256], in)  arg(5, [32][193], out)
fusion__ApplicationModel_F_F_Comp_imp(256, 64,
  (Cplfloat(*)[256])&UG.SegB[0].S2.S3._ApplicationModel_F_X__out[0][0],
  ptr_global->val_chirp_inv,
  (Cplfloat(*)[193])&UG.SegB[0].S2.S4._ApplicationModel_F_Comp_imp_out[0][0]);
```
- communication pragmas that detail source and sink memories for each array to be transferred:
```
#pragma smecy communication src(x86,0,RamInt) dst(GPU,0,RamInt)
// Communication from x86.RamInt to GPU.RamInt
memcpy(&UG.SegB[0].S2.S3._ApplicationModel_F_X__out[0][0],
```

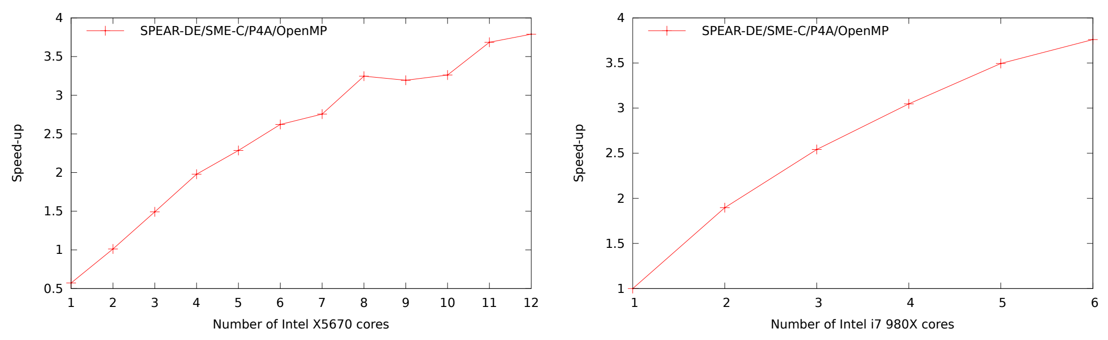

Figure 5: Speedup of the parallel Chirp filter (left-hand side) applied to an input of $256 \times 32 \times 8$ complex float elements, and of the Cholesky decomposition (right-hand side) for a matrix of 1024×1024 complex values.

```
&UG.SegA.S1._ApplicationModel_Gen_chirp_out[0+i__ApplicationModel_F_0][0][0],
32*256*sizeof(Cplfloat));
```

Note that this memory copy statement has been added to ensure a runnable code; Par4All could ignore it because it is able to infer this communication. But it appears more efficient for any back-end to directly collect these informations.

6.2 Performances on $x86$+GPU

In Fig. 5 are sketched the performances obtained running the parallel Chirp filter (left-hand side) on the 12 cores of 2 Intel X5670 at 2.93 GHz (64-bit Linux/Ubuntu 10.04, GCC 4.4.3), and the Cholesky algorithm (right-hand side) on an Intel i7 980X according to different core numbers. The speedup of the two applications has been obtained running the OpenMP code generated by Par4All from the SME-C IR produced by SpearDE.

For Cholesky, the generated code expresses enough parallelism with enough data to be efficient on a small number of processors. Since in the final P2012 target platform the processors are significantly less powerful than the cores of these high-end $x86$ processors, we can expect better efficiency.

On the Chirp application, the parallel execution is less efficient than for Cholesky even if in presence of higher potential parallelism, because some loops with few iterations are parallelized by Par4All. The performance gain obtained by the parallelization of these loops, is not enough to compensate the overhead introduced in the OpenMP runtime (mainly as thread start-up time). Furthermore, there are some parts of the program that are not yet parallelized and optimized by Par4All, so we expect to obtain better speedup in future version.

A direct translation from SME-C code to CUDA code, for the nVidia C2050 GPU (SDK 4.1), is not very efficient. We obtain a modest $1.3\times$ speedup over the sequential time for the Chirp application, going from 7 ms of the SME-C sequential time on the Intel X5670, to 5.4 ms of the CUDA code parallel execution. This is because there is not enough parallelism usable for a high-end GPU and we need to add a phase in Par4All to avoid generating tiny kernels. On the contrary, we obtained a promising $20\times$ speedup for the Chirp filter and a $15\times$ speedup for the Cholesky decomposition algorithm (going from 8440 ms to 560 ms) on an nVidia GTX 480, by directly generating OpenCL 1.1 code. We are currently investigating how performance can be improved for the translation from SME-C code to CUDA code on standard GPUs.

7 Related work

A number of different programming tools and frameworks have already been developed, focusing on high-level design principles and code portability both in parallel and distributed architectures

[8, 9, 10, 11, 12]. Also many APIs and environments, such as OpenMP [13], and BlueBee [14], rely on sequential program annotations either manual or semi-automatic (e.g. via profiling tools).

The automatic approaches rely heavily on the hypothesis that coding rules are strictly observed. To help the programmer, some define also their own programming languages/extensions: such as C in MAPS [15], Idea in Gedae [16], StreamIt [17]. The motivation behind these efforts is on one hand to reuse legacy code and also to raise the degree of expressiveness of the input code. However, despite their many differences, a common goal unites them: making parallel programming for current and future many-core architectures accessible to everyone.

Our proposal fits into this trend and differentiates itself by aiming to put together four main features still lacking from the state-of-the-art today: 1) graphical high-level application design; 2) high-level programming model and platform model; 3) high-level intermediate representation (C code + annotations); 4) efficient code generation on a variety of heterogeneous platforms. Our proposed tool-chain provides an explicit parallel model of computation and a high-level intermediate representation which ensure portability of the parallel code onto different architectures.

From the application standpoint, we argue that the effort of doing an almost from-scratch representation can avoid the clobbering of code with non-portable and complex directives. It also lessens the burden of abiding coding rules, which is usually not feasible with legacy code anyway.

8 Future work and conclusions

In this paper we proposed a compilation tool-chain allowing a high degree of freedom in the definition of the parallel computation for the application programmer. The tool-chain leverages on the SME-C Intermediate Representation (IR) of the parallel program generated by the front-end tool SpearDE allowing to develop efficiently regular parallel data-flow applications. The generated SME-C IR is already directly executable in a very efficient way with a standard OpenMP compiler. The IR can also be given as input to other back-end tools such as Par4All which generates efficient target code for a variety of platforms. The SME-C representation is still an on-going open project that evolves according to the needs of the SMECY [1] project.

The level of abstraction exposed to the applications programmer by the tool-chain proposed can be further raised by introducing the possibility to specify predefined patterns of computations as those used in many parallel skeleton frameworks [18]. This study is planned as a future work.

Currently, we are working in Par4All on developing the tiling and software pipelining to do global-local memory exchange and computation overlapping in P2012 to get good performance. At the same time, we plan to use array region analysis with a new liveness analysis to automatically extract task parallelism to target a P2012 multi-cluster with different tasks and use some polyhedral decomposition of big data parallel kernels to several P2012 clusters [19].

Acknowledgements

The research leading to these results has received funding from the ARTEMIS Joint Undertaking under grant agreement number 100230 and from the national programmes / funding authorities for the SMECY project (www.smecy.eu).

References

[1] SMECY Project, “SMECY — smart multi-core embedded systems.” 2012. [Online]. Available: www.smecy.eu

[2] E. Lenormand and G. Edelin, “An industrial perspective: a pragmatic high-end signal processing design environment at thales,” in *In proceedings of the Workshop on Systems, Architectures, Modeling and Simulation SAMOS*, 2003, pp. 52–57.

[3] HPC Project, "Par4All — single source, multiple targets," 2012. [Online]. Available: par4all.org

[4] M. Amini, C. Ancourt, F. Coelho, B. Creusillet, S. Guelton, F. Irigoin, P. Jouvelot, R. Keryell, and P. Villalon, "PIPS is not (just) polyhedral software," in *First International Workshop on Polyhedral Compilation Techniques (IMPACT 2011)*, Apr. 2011.

[5] P. Bonnot, F. Lemonnier, G. Edelin, G. Gaillat, O. Ruch, and P. Gauget, "Definition and simd implementation of a multi-processing architecture approach on fpga," in *DATE08*, 2008.

[6] MCA, "The Multicore Association: Industry standards to solve multicore challenges." [Online]. Available: http://www.multicore-association.org

[7] R. Klemm, *Space-time adaptive processing: principles and applications*, ser. IEE radar, sonar, navigation, and avionics. Institution of Electrical Engineers, 1998.

[8] M. Vanneschi, "The programming model of ASSIST, an environment for parallel and distributed portable applications," *Parallel Computing*, vol. 28, no. 12, pp. 1709–1732, Dec. 2002.

[9] M. Aldinucci, M. Danelutto, and P. Dazzi, "Muskel: an expandable skeleton environment," *Scalable Computing: Practice and Experience*, vol. 8, no. 4, pp. 325–341, Dec. 2007.

[10] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[11] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati, "Fastflow: high-level and efficient streaming on multi-core," in *Programming Multi-core and Many-core Computing Systems*, ser. Parallel and Distributed Computing, S. Pllana and F. Xhafa, Eds. Wiley, 2012, ch. 13.

[12] J. Reinders, *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*, ser. O'Reilly Series. O'Reilly, 2007.

[13] B. Chapman, G. Jost, and R. v. d. Pas, *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[14] BlueeBee, "BlueBee multi-core technologies." [Online]. Available: http://www.bluebee-tech.com

[15] J. Ceng, J. Castrillon, W. Sheng, H. Scharwächter, R. Leupers, G. Ascheid, H. Meyr, T. Isshiki, and H. Kunieda, "MAPS: An integrated framework for MPSoC application parallelization," in *DAC 2008*, Anaheim, California, USA, 8 – 13 June 2008.

[16] GEDAE. [Online]. Available: http://www.gedae.com/

[17] W. Thies, M. Karczmarek, and S. P. Amarasinghe, "StreamIt: A language for streaming applications," in *Proc. of the 11th Intl. Conference on Compiler Construction (CC)*, London, UK, 2002, pp. 179–196.

[18] H. González-Vélez and M. Leyton, "A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers," *Softw. Pract. Exper.*, vol. 40, pp. 1135–1160, November 2010.

[19] C. Ancourt, F. Coelho, F. Irigoin, and R. Keryell, "A linear algebra framework for static high performance fortran code distribution," *Scientific Programming*, vol. 6, no. 1, pp. 3–27, 1997.