Available online at www.prace-ri.eu

Partnership for Advanced Computing in Europe

The State-of-the-Art in Directive-Guided Auto-Tuning for Accelerator and Heterogeneous Many-Core Architectures

Renato Miceli*[a], François Bodin[b]

[a] *Irish Centre for High-End Computing, Dublin, Ireland*
[b] *CAPS Entreprise, Rennes, France*

Abstract

In this whitepaper we discuss the latest achievements in the field of auto-tuning of applications for accelerator and heterogeneous many-core architectures guided by programming directives. We provide both an academic perspective, presenting preliminary results obtained by the EU FP7 AutoTune project, and an industrial point of view, demonstrated by the commercial uptake by a leader in compiler technology and services, CAPS Entreprise.

1. Introduction

Automatic performance tuning, also known in the field as "auto-tuning", is an automatic software procedure for selecting one out of several possible algorithmic implementations of a solution to a computational problem. The objective of auto-tuning is to increase an application's performance by selecting the best implementation for a particular environment, i.e. the adjoined software and hardware. Each of the implementations, called a variant, provides different time and space complexities that may be better suited to one or another logical architecture, physical hardware, runtime system, compilation tool chain or desired execution workload. In this paper we investigate auto-tuning approaches for improving the performance of applications for accelerator and heterogeneous many-core architectures. We focus on auto-tuning guided by programming directives, also known as code annotations. We go over the state-of-the-art in directive-guided auto-tuning for many-core architectures, providing examples coming from the EU FP7 AutoTune project. In the end, we discuss the technology's commercial uptake, using the CAPS compilers as the example of current industrial efforts to bringing auto-tuning into the mainstream software development cycle.

2. Background

2.1 Definition of Auto-Tuning

Auto-tuning is as simple as using software to tune other pieces of software. The objective is to take the task of code tuning – i.e. making an application perform better – out of the hands of programmers, in order to release human time to be invested in more skilled activities. Auto-tuning may be executed by the same software to be tuned, in which case we call a "self-tunable" code; or by an external mechanism – e.g. library, runtime environment, independent application –, which we call an "auto-tuner". In either case, the auto-tuning approach will introduce changes to the algorithmic implementation in order to promote performance gains; some examples of approaches are the use of novel programming languages, automatic code transformation techniques, programming environments, and language extensions, such as programming directives and annotations.

The task of auto-tuning is necessarily an empirical search. The auto-tuner identifies the code regions that may benefit from performance tuning and elects a set of possible implementations that perform the same computation but with different algorithmic complexities; these are the code variants. The auto-tuner's task is therefore to optimize the performance based on its knowledge of the software, the hardware and the data involved, searching for the best performing code variants in its search space. The search is conducted by experimentation, fed back by measurements gathered while executing the application. The

* Corresponding author. E-mail address: renato.miceli@ichec.ie

auto-tuner decides which code variants to experiment next based on expert knowledge, commonly guided by search heuristics using results of previous runs.

Introducing performance gains by automatic mechanisms is becoming an increasingly important field especially with the growing complexity of computer architectures, the user's demand for portable performance across general architectures, and the current development of technologies to support exascale high-performance computing. In exascale, an application's runtime and energy consumption must be specifically tailored to the underlying architecture, in order to avoid the waste of precious resources.

2.2 Programming Directives and Auto-Tuning

Programming directives, also known as annotations or pragmas, are programming language constructs generally employed to specify source code metadata. Directives have a dual use: (i) they may be commands, commonly referring to cross-cutting concerns; or (ii) they may be statements, describing local or global blocks of code, but performing no action. Programmers often insert directives into the code to facilitate the software development cycle – i.e. to insert user-friendly information about the development process – or to provide useful information to the compiler or runtime system, such as to instruct the compiler to use implementation-dependent features or to introduce changes to the runtime system agnostically to the application code.

Currently, directives may carry important information auto-tuners may use to understand the source code structure, the operations carried out, the data and dependencies involved, and to extract the key factors that influence application performance. Searching the set of possible code variants becomes a less complex task with the help of programming directives. Directives have historically been widely adopted in the community of parallel application development, with the advent of programming models that enable the use of multi- and many-core architectures in a seamless and portable way, such as OpenMP, OpenHMPP and OpenACC. Extending the use of programming directives from portable programming towards auto-tuning should therefore face no obstacles in the community.

In this paper we focus on language extensions, specifically programming directives and annotations.

2.3 Auto-Tuning Many-Core Architectures

A many-core architecture is the one that includes a many-core processor. These processors are a specialization of multi-core processors and include several independent CPUs, i.e. the so-called cores. The expression "many-core processor" was coined in opposition to "multi-core processor"; while multi-core processors contain tens of robust cores, many-core processors contain hundreds to thousands of light cores, commonly vector or SIMD/SIMT cores. Besides, whereas multi-core processors are generally connected to the remainder computing system via a CPU socket, many-core units are separate boards generally linked to a motherboard via a PCIe port. Therefore, many-core units act as coprocessors or accelerators; they help the multi-core processor with its calculations by accepting offloaded work and returning computed results, in a master-worker fashion. Some examples of many-core units are GPUs, GPGPUs, FPGAs and Intel Xeon Phi coprocessors.

The massively parallelism on many-core units and the contrast to multi-core processor cores allow applications to benefit from both paradigms depending on the resource usage and the level of parallelism required. It also becomes apparent that balancing resource usage between the main processor and the accelerator is a necessary measure to obtain maximum performance on these heterogeneous many-core architectures. Besides, Auto-tuning applications running on accelerators, as well as balancing the load between the main processors and the accelerators, has become a great source of study and interest, both within the academic and industrial communities.

3. Research Achievements and Results

While not many works tackle directive-guided auto-tuning for accelerator and many-core architectures, subsets of these features are found in several works in the state-of-the-art. Directive-guided auto-tuning, and auto-tuning for many-core accelerators, are the subject of study of the many more pieces of work.

A relevant project tackling directive-guided auto-tuning for accelerator and many-core architectures is the AutoTune project. AutoTune is a European-funded FP7 project that aims at building an extensible auto-tuning framework for tuning application performance and energy consumption. The framework, called the Periscope Tuning Framework (PTF), instruments, compiles and executes the application to gather measurements, which will guide the tuning process. The framework focuses on static tuning, i.e. it identifies tuning recommendations in special application tuning runs, so that the recommendations can then be applied to optimize the code for later production runs.

PTF includes automated performance analysis strategies for various parallel paradigms and programming models. The strategies are based on formalized performance properties specifying typical performance bottlenecks, the metrics required for their

detection, as well as the severity of the bottlenecks.

In the center of PTF are the so-called tuning plugins, which focus on individual tuning aspects. A plugin explores a tuning space; however, as the tuning space is typically quite large, plugins can run performance analysis strategies first and use codified expert knowledge to shrink the tuning space based on the resulting performance properties. The remaining space is searched by generic predefined or plugin-specific search strategies.

The framework is quite extensible to the point that plugins are dynamically loaded, in order to allow plugin to be developed by both the academia and the industry. Able to execute on over 65,000 BlueGene cores, PTF's extensible and scalable infrastructure is currently implemented on top of Periscope, a distributed online performance analysis tool conceived at Technische Universität München (TUM) in Germany.

Within the AutoTune project, the framework and a number of tuning plugins are under development. The project consortium is currently working on a total of 7 plugins, 3 of which tune for heterogeneous many-core architectures. The 3 plugins are the following:

- User-guided tuning plugin: assesses multiple code variants statically defined in the code, exhaustively searching for the variant that yields the shortest execution time;
- Tuning plugin for high-level parallel patterns for GPGPUs: focuses on maximizing the throughput of pipeline patterns for many-core heterogeneous architectures by efficiently exploiting CPU and GPU cores of a targeted architecture, especially the replication factor and the buffer sizes of pipeline stages; and
- Tuning plugin for hybrid many-core HMPP codelets: tunes the performance of a codelet computation, written either using OpenHMPP or OpenACC directives and compiled using the CAPS compiler, by performing source code transformations and assessing and choosing code variants for operations and algorithms used to implement codelets and values to target-specific variables and callbacks.

3.1 AutoTune's Proof of Concept

Currently, PTF only requires that the user inserts programming directives to specify the region where it should tune and the tuning parameters whose values should be experimented. A proof of concept was developed for the user-guided tuning plugin, and the experimented Fortran source code is as below:

```
1  do k=1,20
2     var=k
3     !$MON USERREGION TP name(Test) variable(var) variants(10)
4     tstart=MPI_Wtime()
5     !<user compute code depending on the value of variable 'var'>
6     tend=MPI_Wtime()
7     !$MON END USERREGION
8  enddo
```

Figure 1: Sample use of programing directives for defining a tuning region

In this code, it is possible to see that a tuning region is defined directly in the code by the AutoTune pragmas "USERREGION" and "END USERREGION", which must surround the tuning region. The same "USERREGION" AutoTune pragma defines the tuning parameter the plugin can process: its declaration starts with the keyword "TP", followed by (i) the name of the tuning parameter, (ii) the variable identifier, and (iii) the number of variants.

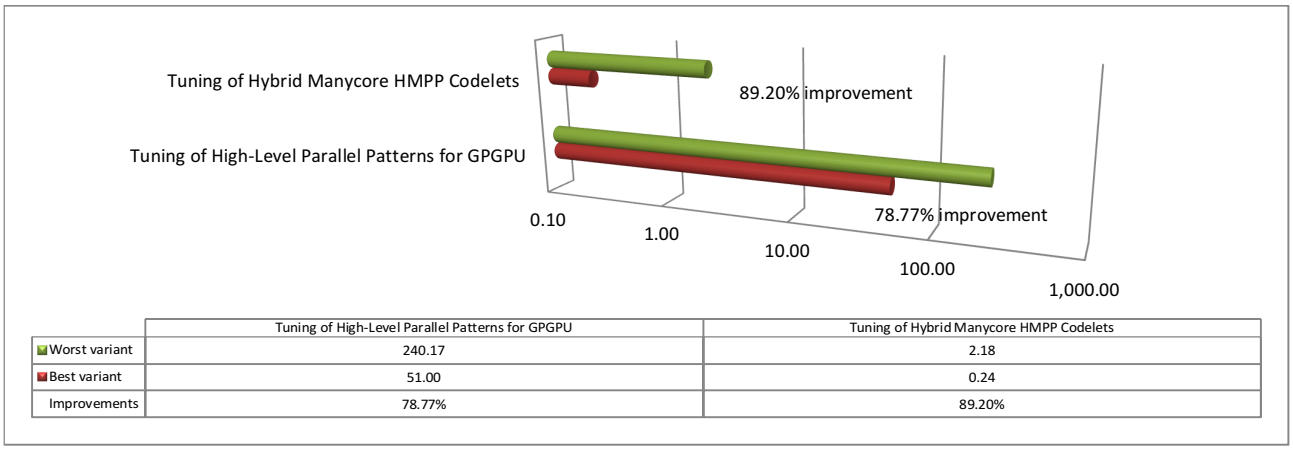
During instrumentation, the PTF instrumenter parses the Fortran source code to extract information from the AutoTune pragmas. The tuning parameters (variable *var* in our example) are provided to the frontend as input. Whenever the region is entered, the monitor assigns a value to the variable to trigger the execution of a certain variant. In our example, PTF sweeps over all variants from 1 to 10.

Low programming overhead and high application portability are possible on PTF and demonstrated in the previous example. Here, the value of variable *var* is initially set to *k*; however, since *var* is a tuning parameter, the monitor overwrites its value at runtime upon entry in the tuning region. On the other side, setting *var* to *k* allows the code to be run without PTF.

During execution, whenever the tuning region is entered, PTF assigns a different value to the tuning parameter and gather statistics that will enable it to reason about the best code variant for that specific region.

3.2 Preliminary Results

Apart from the user-guided tuning plugin, each of the six remaining tuning plugins is being implemented based on tuning techniques and search strategies for a specific tuning aspect. A preliminary study about the quality of these tuning techniques was conducted. It consisted of experimenting each code variant, measuring runtime metrics and traversing the search space, all manually. The objective was to assess the quality of the tuning techniques before investing in actual plugin development, hinting at the effectiveness and efficiency of the tuning techniques, and to show that the amplitude of code variants was broad to permit good improvements to be made to the application. The manual procedure tuned real-world applications and the initial results, all gives in seconds, are as below:



	Tuning of High-Level Parallel Patterns for GPGPU	Tuning of Hybrid Manycore HMPP Codelets
Worst variant	240.17	2.18
Best variant	51.00	0.24
Improvements	78.77%	89.20%

Table 1: Runtime comparison of code variants (in seconds)

It can be seen that all tuning techniques for many-core tuning managed to find optimized code variants. They excelled in the searching task and were able to find variants at least 75% better performing than the worst counterparts.

However, manual tuning consumes human effort. The following table describes the time taken for manual tuning per technique.

Tuning Technique	Time effort for manual tuning
High-level Parallel Patterns for GPGPU	120 man-hours
Hybrid Manycore HMPP Codelets	140 man-hours

Table 2: Human effort spent in manual code tuning

It is possible to see that the tuning techniques for many-core tuning took an average of 130 man-hours to be manually implemented. It could be concluded that the tuning techniques were very promising, since good improvements were manually found. As tuning plugins will be able to search the code variant space and assess a variant's performance automatically, many more variants will be tested in less time; thus, even better code variants can be found, accounting for high improvements relative to the worst variants found. All the tuning will be performed without human investment so developer time can be spent in more mentally-challenging tasks.

4. Commercial Uptake

CAPS Entreprise, a leading provider of many-core technology and services, provides the commercial uptake in the AutoTune project outcomes. CAPS integrated auto-tuning interfaces and features to their CAPS compiler, allowing the source code to be adapted at runtime according to the underlying architecture. The CAPS compiler takes the source code, written using C/C++ or Fortran annotated with OpenHMPP or OpenACC directives, understands special auto-tuning pragmas and generates an executable that captures usage information and runtime metrics. The CAPS auto-tuning driver connects to the executable at runtime to create an optimization space of code transformations and algorithms to explore.

A diagram showing how compiling and auto-tuning using CAPS compilers works is as it follows.

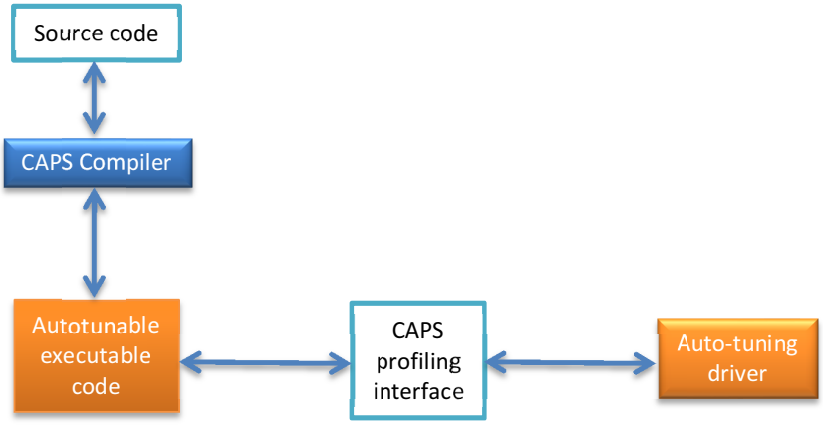


Figure 2: Compilation and auto-tuning workflow using CAPS's tool chain

In the previous diagram, we can see the workflow for both compiling and auto-tuning OpenACC- and OpenHMPP-directed codes using the CAPS compilers. First, the source code written in C/C++ or Fortran is annotated with OpenACC and/or OpenHMPP directives and compiled using the CAPS compiler. The output will be an auto-tunable code, executable by a heterogeneous many-core architecture. Via the CAPS profiling interface, the CAPS auto-tuning driver gathers measurements to guide the search through the tuning space for well-performing code variants for a specific underlying architecture.

The tuning parameters currently supported by the CAPS compiler are the following:

- Implementations of a same kernel or codelet;
- Runtime configurable parameters for kernel and codelet executions (e.g. number of gangs, workers and vectors);
- Automatic code transformations, especially regarding loop transformations (e.g. jam, split, unroll, gridify);
- Implementations of library calls, especially where no one-to-one mapping between libraries with similar features exist.

4.1 Auto-tuning dynamic kernel parameters

A simple example of how auto-tuning techniques work on OpenACC codes follows. Below is a sample OpenACC piece of code, annotated with kernel and memory copy directives.

```
1  #pragma acc parallel, copyin(dst_caps[0:height*width]),
2     copyout(src_caps[0:height*width]), num_gangs(gangs),
3     num_workers(workers), vector_length(32)
4  {
5  #pragma acc loop, gang
6  for (tileY = 0; tileY < tileCountY; tileY++) {
7     for (tileX = 0; tileX < tileCountX; tileX++) {
8        ...
```

Figure 3: Sample OpenACC-annotated kernel code where gangs and workers are tunable parameters

Notice that the declaration of number of gangs and workers is parameterized. The number of gangs and workers is assigned at runtime, on the kernel call by the auto-tuning driver. The parameter space to explore is statically defined in the code as arrays of values, whose index is established by the auto-tuner:

```
1  size_t gangs[] = { 8, 16, 32, 64, 128, 128, 8, 16, 32, 64, 128, 256 };
2  size_t workers[] = { 16, 16, 16, 16, 16, 16, 24, 24, 24, 24, 24, 24 };
3  while (nber_of_iterations < max_iterations) {
4     variant = variantSelectorState("kernel.c:21",
5                                    (sizeof(gangs)/sizeof(size_t))-1);
6     blur(images[ (currentImage + 1) % 2], image_caps, width, height,
7          blockSize, gangs[ variant], workers[ variant] );
8  }
```

Figure 4: Sample code for an OpenACC kernel call showing the exploration space for gangs and workers for the kernel described in Figure 3

Some preliminary testing with combinations of gangs and workers was conducted over the bioinformatics application DNADist on the NVIDIA CARMA, NVIDIA Kepler and Fermi, Intel Xeon Phi and AMD 7970 and Trinity architectures. The x-axis represents a specific configuration pair (#gangs, #workers), while the y-axis denotes the execution time in seconds.

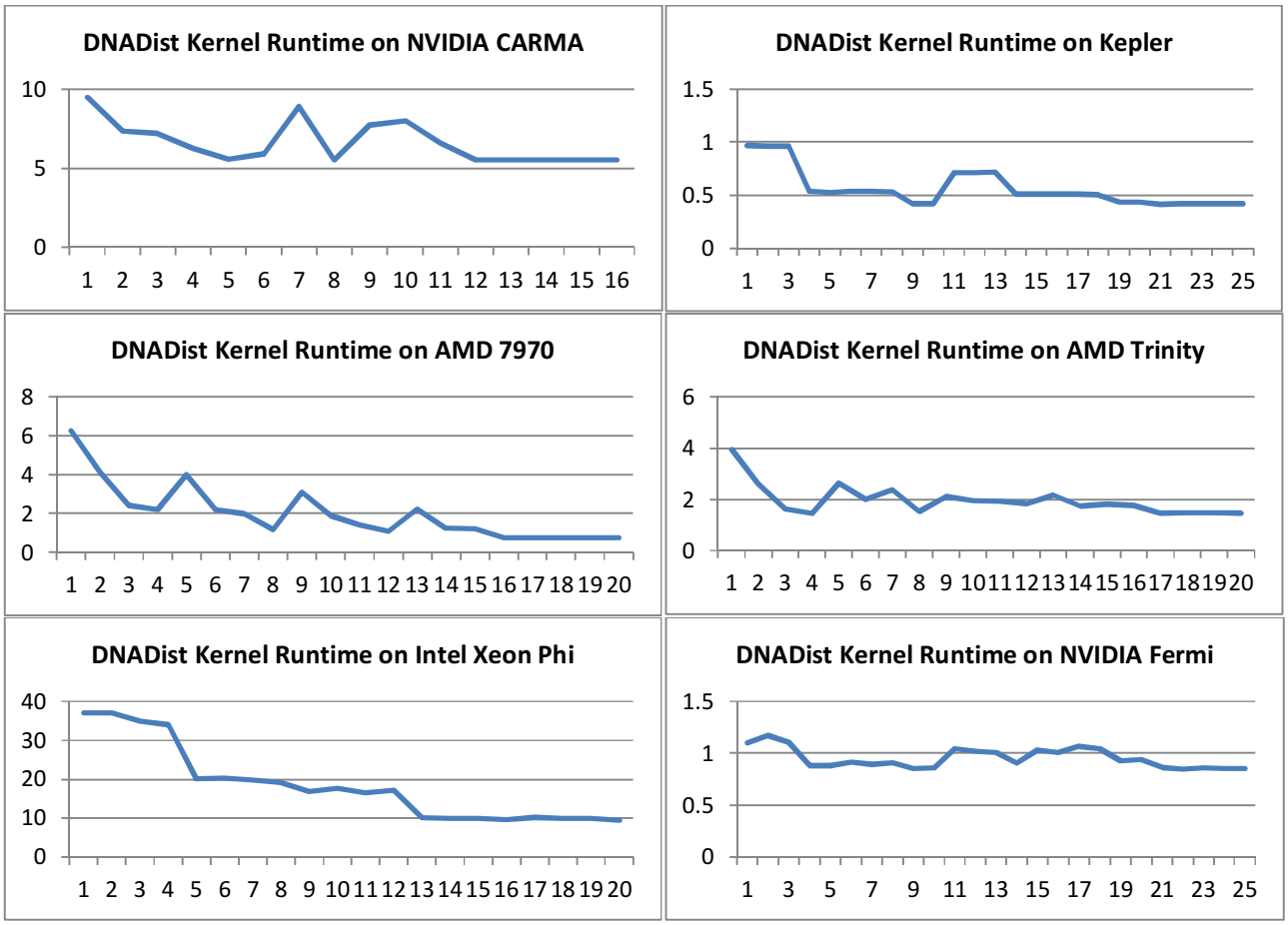


Figure 5: Performance for the bioinformatics application DNADist on six different architectures, depicting the runtime for several configurations of the OpenACC parameters for gang and workers as explored by the CAPS auto-tuning driver

It can be seen that on CARMA, configuration number 8 yields the shortest kernel execution time; this configuration determines 14 gangs and 16 workers per gang. On Kepler, configuration number 10 manages to run the quickest; this configuration establishes 256 gangs and 128 workers per gang. On AMD 7970, 16 is the best configuration, with 64 gangs and 8 workers. AMD Trinity runs DNADist better with configuration 8, rendering 14 gangs and 8 workers per gang. DNADist on Xeon Phi performs better with configuration 16, i.e. 64 gangs and 8 workers, whereas on Fermi configuration 10 is best, with 256 gangs and 128 workers per gang.

4.2 Selecting codelet implementations on runtime

The AutoTune pragmas in OpenHMPP implemented by the CAPS Compilers allows them to dynamically select a codelet based on its execution time experimented on the fly. Consider the two codelets below, which have different implementations for the same algorithm.

```
1  void filterStencil5x5_V1 (const uint32 p_heigh[1],
2     const uint32 p_width[1], const RasterType filter[5][5],
3     const RasterType *p_inRaster, RasterType *p_outRaster) {
4        ...
5  #pragma hmppcg grid blocksize "32x4"
6  #pragma hmppcg unroll 6, jam
7     for (i = stencil; i < heigh - stencil; i++) {
8        ...
```

```
1  void filterStencil5x5_V2 (const uint32 p_heigh[1],
2     const uint32 p_width[1], const RasterType filter[5][5],
3     const RasterType *p_inRaster, RasterType *p_outRaster) {
4        ...
5  #pragma hmppcg grid blocksize "16x8"
6  #pragma hmppcg unroll 4, jam
7     for (i = stencil; i < heigh - stencil; i++) {
8        ...
```

Figure 6: Two different implementations of the 'filterStencil5x5' function; notice only the values for the OpenHMPP directives change

These two code variants, and even more, may be experimented by the auto-tuner driver on runtime before selecting the variant to serve for a specific architecture. The code below illustrates how the list of possible variants may be inserted into the code for the auto-tuner to test, in an AutoTune pragma placed before the codelet callsite.

```
1  #pragma hmpp <convolution> filter5x5 callsite variants( &
2  #pragma hmpp & filterStencil5x5@<convolution>[C], &
3  #pragma hmpp & filterStencil5x5_V1@<convolution>[CUDA], &
4  #pragma hmpp & filterStencil5x5_V2@<convolution>[CUDA]) &
5  #pragma hmpp & selector(filterVariantSelector)
6  filterStencil5x5(&fullHeigh, &width, stencil1, raster1, raster2);
```

Figure 7: Sample code for a HMPP callsite, where the possible variants are declared using OpenHMPP directives and selected at runtime

The list of code variants for a same codelet may include not only variants for many-core devices, but also for x86 and ARM multi-core processors. The auto-tuner will select the variant to apply provided the underlying architecture supports, and depending on experiments run on execution time.

4.3 A DSL-Oriented Approach to Auto-Tuning

It may be said that the code transformations applied to applications for a targeted architecture vary according to that architecture, since the architecture itself contains the hardware and software features that dictate the performance of applications. Nonetheless, the code transformations must currently be applied by software developers, and not by hardware architects. CAPS introduced a high-level approach, driven by programming directives and DSL scripts, to condensate low-level code transformations details in scripts that can be generically written by hardware architects when shipping their devices. The CAPS auto-tuner feeds the application source code to the script, in order to adapt the source code to the underlying architecture (i.e. auto-tune the code) and thereafter benefiting from specific nuances the hardware may display.

The following piece of code illustrates a code region upon where a certain script is activated.

```
1  !$capstune scriptName scriptInput
2    ... code region ...
3  !$capstune end scriptName
```

Figure 8: Sample code for selective script activation using CAPS's 'capstune' directives

A short example is now introduced. Consider the problem of performing a stencil computation. Each architecture contains its level of compute power, amount of RAM and cache, number of registers and so on. These fine-grain low-level details may not be particularly known by all applications developers; hence architecture developers may enclose them in DSL scripts to be selected upon tuning time. The script may decide to test multiple variants, to use external tools, or simply employ a library. It is up to the script writer to define the strategy for tuning the stencil.

5. Conclusions

In this paper we described the current state-of-the-art in auto-tuning guided by directives, for accelerator and heterogeneous many-core architectures. We focused on the EU FP7 AutoTune project, which is already producing preliminary promising results towards auto-tuning pipelined GPGPU applications and OpenHMPP codelets/OpenACC kernels. Then, we showed how the industry is currently approaching auto-tuning, exemplified by CAPS Entreprise. CAPS is integrating auto-tuning approaching into their mainstream product line, the CAPS compilers. Their results demonstrate that auto-tuning is capable of highly increasing an application's performance with little effort from the developer side. Both projects show that auto-tuning is a promising technology that will be very important for portable performance across architectures, hence a procedure that will soon become seamlessly integrated into the development and execution software cycles.

Acknowledgements

This work was supported by the PRACE project funded in part by the EU 7th Framework Programme (FP7/2007-2013) under grant agreement no. RI-283493. The authors would like to thank the FP7 AutoTune project (grant agreement no. 288038) and CAPS Entreprise for their collaboration in various aspects of this work.

References

[1] H. Mantri, S. Banerjee, "Auto Tuning" (available at http://www.slideshare.net/miracle99i/auto-tuning-12240393)
[2] K. Naono, K. Teranishi, J. Cavazos, R. Suda, "Software Automatic Tuning: From Concepts to State-of-the-Art Results", ISBN 978-1-4419-6934-7, Springer (doi: 10.1007/978-1-4419-6935-4)
[3] B. Ylvisaker, S. Hauck, "Probabilistic Auto-Tuning for Architectures with Complex Constraints", Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era (EXADAPT'11), pp. 22-23
[4] P. Ivanenko, "TuningGenie — an autotuning framework for optimization of parallel applications" http://taac.org.ua/files/a2012/proceedings/UA-1-Ivanenko-272.pdf
[5] T. Karcher, C. Schaefer, V. Pankratius, "Auto-Tuning Support for Manycore Applications - Perspectives for Operating Systems and Compilers" (available at http://www.ipd.uni-karlsruhe.de/multicore/research/download/Autotuning-OSCompilers.pdf)
[6] C. Schaefer, V. Pankratius, W. Tichy, "Atune-IL: An Instrumentation Language for Auto-Tuning Parallel Applications"
[7] R. Miceli, G. Civario, A. Sikora, E. César, M. Gerndt, H. Haitof, C. Navarrete, S. Benkner, M. Sandrieser, L. Morin, F. Bodin, "AutoTune: A Plugin-Driven Approach to the Automatic Tuning of Parallel Applications", Proceedings of the 11th International Workshop on the State-of-the-Art in Scientific and Parallel Computing (PARA 2012), pp. 328-342 (Springer LNCS vol. 7782 http://dx.doi.org/10.1007/978-3-642-36803-5_24)
[8] The AutoTune Project website (http://www.autotune-project.eu/)
[9] The CAPS Entreprise website (http://www.caps-entreprise.com/)
[10] R. Miceli, G. Civario, F. Bodin, "AutoTune: Automatic Online Code Tuning", NVIDIA GPU Technology Conference 2012 (GTC 2012). San Jose, USA. May 2012 (available at http://www.autotune-project.eu/sites/default/files/autotune-gtc2012-poster-ichec.pdf)
[11] R. Miceli, "AutoTune: Plugin-based Tuning of Parallel Codes", The 8th International Conference on High-Performance and Embedded Architectures and Compilers (HiPEAC 2013). Berlin, Germany. January 2013
[12] F. Bodin, "Advanced Programming of Many-core Systems Using CAPS OpenACC Compiler", Saudi Arabian High Performance Computing Users' Group Conference (Thuwal 2012). Thuwal, Saudi Arabia. December 2012 (available at http://www.hpcsaudi.com/wp-content/uploads/2012/09/Francois-Bodin.pdf)
[13] F. Bodin, F. Lebeau, "Programming Heterogeneous Many-cores Using Directives", NVIDIA GPU Technology Conference 2012 (GTC 2012). (http://developer.download.nvidia.com/GTC/PDF/GTC2012/PresentationPDF/S0630-Monday-

Programming-Heterogeneous.pdf)

[14] F. Bodin, "Using CAPS Compiler on NVIDIA Kepler and CARMA Systems", The 24th International Conference for High Performance Computing, Networking, Storage and Analysis (SC12). Salt Lake City, USA. November 2012 (available at http://developer.download.nvidia.com/GTC/PDF/GTC2012/PresentationPDF/FrancoisBodin_CAPS_SC12.pdf)

[15] H. Jordan, P. Thoman, J. J. Durillo, S. Pellegrini, P. Gschwandtner, T. Fahringer, H. Moritsch, "A Multi-Objective Auto-Tuning Framework for Parallel Codes", Proceedings of the 24th International Conference for High Performance Computing, Networking, Storage and Analysis (SC12). Salt Lake City, USA. November 2012

[16] D. Mustafa, R. Eigenmann, "Portable section-level tuning of compiler parallelized applications", Proceedings of the 24th International Conference for High Performance Computing, Networking, Storage and Analysis (SC12). Salt Lake City, USA. November 2012

[17] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, J. Cavazos, "Auto-tuning a High-Level Language Targeted to GPU Codes", Proceedings of the Innovative Parallel Computing Conference (InPar 2012), San Jose, USA. May 2012