"OpenMP" automatic parallelization tools: An Empirical comparative evaluation

Emna KALLEL, Yassine AOUDNI and Mohamed ABID

**Sfax University, National school of Engineers of Sfax,
3038,Tunisia,**

Abstract

Today, multi-core design has become the trend of enhancing the processor's performance, and most industries have been considering multi-core as the future of development. Thus, a programmer or a compiler explicitly parallelizes the software, which is the key to enhance the performance on multi-core design. Nevertheless, currently, needs an in-depth knowledge of both software and hardware design to develop parallel applications. Automatic parallelization is one of the approaches aiming at a better and easier use of parallel computers. In recent years, several research auto-parallelization tools appeared. However, the automatic parallelization is yet to become a widely adopted industrial practice. This paper presents an empirical comparison between three research tools, namely CETUS, PLUTO and GASPARD. Indeed, we discuss the success of these tools to automatically generate OpenMP parallel codes from serial C codes and compare them using known benchmark C workloads and some evaluation metrics.

Keywords: *Multi-core, automatic parallelization, OpenMP, metrics, parallel benchmarks.*

1. Introduction

While classical parallel machines serve a relatively small user community, multi-cores aim to capture a mass market, which targets user-oriented, high-productivity programming tools. Furthermore, multi-cores are replacing complex superscalar processors, the parallelism of which was unquestionably exploited by the compiler and underlying architecture [6]. However, while developing a parallel application, multiple challenges, ranging from the extraction of the application parallelism to the management of the communications between processors, are to be overcome. Dating back to more than 15 years, the automatic parallelization approaches have appeared as one of solutions that aimed at a better and easier use of parallel computers. It consists in taking a program written for a sequential computer (which has only one processor) and to adapt it to a parallel computer. The condition for executing loop iterations in parallel is that the variable usage in the loop does not result in loop-carried dependences. So, the auto-parallelizing compilers try to analyze codes and eliminate these dependences to automatically generate parallel codes. Building an auto-parallelizing compiler has proven to be a very difficult task when producing efficient code in all cases for a wide variety of applications.

Multiple research auto-parallelization projects have been developed to address some of these problems. *For example,* ROSE [9] is an open source compiler. It builds source-to-source program transformation and analysis tools for large-scale FORTRAN, C, C++, and OpenMP. CETUS [6] is a source-to-source C compiler written in Java and maintained at Purdue University. CETUS provides automatic parallelization, and many other applications have emerged. CETUS 1.2.1 release provides OpenMP to-CUDA package. PAR4ALL [10] is an open-source environment to do source-to-source transformations on C and FORTRAN programs for parallelizing, optimizing, reverse engineering, etc. on various targets, from multicore system-on-chip with some accelerators up to high performance computers and GPU. Also, there are other compilers which can automatically transform serial C/C++ codes to parallel C/C++ codes or to parallel programs by using OpenMP directives [15] or CUDA [5]. In recent years, several commercial compilers that parallelize sequential code also appeared (eg. Intel's[19], IBM's[20] compilers).

Despite all those efforts, the automatic parallelization is yet to become a widely adopted industrial practice. The

most important reason is the quality of the generated parallel code, resulting in inefficient hardware utilization and low performance gain.

This paper studies the above outlined problems through an empirical comparison between three available research tools, namely CETUS, PLUTO and GASPARD. The objective of our work is to have an experiment on some auto-parallelization tools, to let us know if these tools are really available, or if they are actually effective when they are used to enhance the performance. Some parallel benchmarks are used through these compilers. To compare their performance some parallel program performance metrics are adopted.

The rest of this paper is organized as follows. Section 2 presents automatic parallelization techniques and some related work. Section 3 describes the analysis model proposed for the experimentation, while section 4 presents the obtained results. Finally conclusions and future work are presented in section 5.

2. Automatic parallelization techniques

Several works in parallel programming area aim to automatically parallelize a sequential code through many analyses programmes. On an academic scale, we find several auto-parallelization tools. In this section, we separated these auto-parallel tools into two types, one is source to source and the other is model to source parallelizer.

2.1 Source-to-Source auto-parallelizer

Many compute-intensive applications often spend most of their execution time in nested loops. The Polyhedral Model provides a powerful abstraction to reason about transformations on such loop nests by viewing a dynamic instance of each statement as an integer point in a well-defined space called the statement's polyhedron [7]. Also, the polyhedral model allows representing easily regular calculations. He allows specifying, examining, analyzing, transforming, and parallelizing affine equations and so generating regular parallel architecture. It can treat programs with static control structure and affine references/loop-bounds. Also, codes with non-affine array access functions or code with dynamic control can be handled, but only with conservative hypothesis on some dependences. Automatic parallelization efforts in the polyhedral model broadly fall into two classes: (1) scheduling/allocation-based, and (2) partitioning based. Griebl [18] (to some extent) fall into the former class, while Lim/Liao's approach [14] falls into the second class. Griebl [18] presents an integrated framework for optimizing locality and parallelism with space and time tiling, by treating tiling as a post-processing step after a schedule is found. Lim and Liao [14] proposed a framework that identifies outer parallel loops (communication-free space partitions) and permutable loops (time partitions) to maximize the degree of parallelism and minimize the order of synchronization. They employ the same machinery for blocking. Several solutions equivalent in terms of the criterion they optimize for result from their algorithm, and these significantly differ in performance.

No metric is provided to differentiate between these solutions as maximally independent solutions are sought, without using any cost function. As shown through this work, without a cost function, solutions obtained even for simple input may be unsatisfactory with respect to communication cost, locality, and target code complexity. The proposed approach in [7] is closer to the latter class of partitioning-based approaches. However, it is the first to explicitly model tiling in a polyhedral transformation framework, thereby enabling the effective extraction of coarse-grained parallelism along with data locality optimization. At the same time, codes which cannot be tiled or only partially tiled are all handled, and traditional transformations are captured. In fact, it presents an automatic polyhedral source-to-source transformation framework PLUTO that can optimize regular programs for parallelism and locality simultaneously. This framework has been implemented into a tool to automatically generate OpenMP parallel code from C program sections.

However, the front end used in PLUTO accepts only a very little nested loops set. An important limitation of polyhedral parallelizer compilers: there isn't enough information during the compilation to generate a parallel code. For example when the parallelism of the application depends on input data, the compiler is not able to parallelise the program. This problem can be resolved by the taking advantage of the compilation directives which can indicate to the compiler how to decompose the parts of a sequential program for its parallel executing. Automatic parallelization by directives insertion is the set of code transformations at the code source level where the parallelism is expressed by directives given to the compiler. These directives allow the programmer a separation of the preoccupations correction and efficiency [1]: they are a good means to optimize the programmes without questioning the correctness of them. Directives of compilation are used to facilitate the extraction of the parallelism as well as to help in the placement of the calculations and the data on processors. This

methodology is used for example in HPF [4]. In this work, the programmer supplies the directives of partitioning and placement to obtain the placement of the calculations on processors. More than the directives of placement, we distinguish another directive type: the directives of scheduling for shared memory architectures (such as the parallel directive of OpenMP [15]). For example, the CETUS tool [6] provides an infrastructure for research on multicore compiler optimizations that emphasizes automatic parallelization. The CETUS compiler translates OpenMP directives into text strings and stores them in the Intermediate Representation (IR). Then, the OpenMP parser analyzes these text strings to convert them into Annotation, which is a CETUS' map data structure that contains processed OpenMP directive information. For example, "#pragma omp parallel private(a, b)" in the input OpenMP program is transformed into a single text string, which is converted into Annotation map data structure whose (key, values) mappings are (parallel, true) and (private, {a, b}). Compiler analysis passes can easily look up this Annotation map data structure to query the necessary OpenMP information, which is also stored in the CETUS IR. The compiler infrastructure, which targets C programs, supports source-to-source transformations, is user oriented and easy to handle, and provides the most important parallelization passes. HiCUDA [8], a high-level directive-based language for CUDA programming. It allows programmers to perform these tedious tasks in a simpler manner, and directly to the sequential code. Nonetheless, it supports the same programming paradigm already familiar to CUDA programmers. Han [8] prototyped a source-to-source compiler that translates a hiCUDA program to a CUDA program. HiCUDA presents the programmer with a computation model and a data model. The computation model allows the programmer to identify code regions that are intended to be executed on the GPU and to specify how they are to be executed in parallel. The data model allows programmers to allocate and de-allocate memory on the GPU and to move data back and forth between the host memory and the GPU memory. The hiCUDA directives are specified using the pragma mechanism provided by the C and C++ standards. The hiCUDA compiler, despite its flexibility, demand to programmer to write correct and optimal programme to obtain preferment results. Lin and Chen [3] introduce a novel compiler based approach for GPGPU programming by providing a high degree of data parallelism. Compiler directives are used to label code fragments that are to be executed on the GPU. The proposed GPGPU compiler, Guru, converts the labelled code fragments into ISO-compliant C code that contains appropriate OpenGL and Cg APIs. A native C compiler can then be used to compile it into the executable code for GPU. Guru is implemented based on the Open64 compiler infrastructure.

However, for these parallelizer compilers by directives insertion, the process of parallel programming remains ineffective because of absence of the inevitable specific details of programming. The model-based automatic parallelization environments reduce a lot the complexity of development of the parallel programs by their graphic parallel programs modelling in a high level abstraction.

2.2 Model-to-Source auto-parallelizer

The usage of models for the design of multi-processor systems is on its own a great improvement over current practice because it provides a higher abstraction level that especially helps for component reuse and parallel coding. The graphical representation also facilitates the global vision of complex systems and of interactions between the parts of the system. One methodology for the extraction of the parallelism is to partition the original program to diverse independents tasks. It can be realized by using tasks graphs as in ParDT [17]. ParDT is a graphical model-driven development tool suite which supports not only modelling of parallel programs on high abstraction level but the translation of the constructed models into source code skeletons according to the specific runtime environments and libraries. The process of translation, which involves the parsing of graphical models and the generation of source code skeletons aimed at different parallel platforms, are explained in detail. ParDT is implemented based on Eclipse and compatible with its plug-in architecture. The tool suite manages to help programmers relieve the burden of building parallel applications. One important challenge that arises in multicore systems is the ability to dynamically adapt a running application to target architecture in the face of changes in resource availability (e.g., number of cores, available memory or bandwidth). In GASPARD [2], the compilation is a sequence of small and maintainable transformations that allows passing gradually from a high-level description into models closer in abstraction to the final model, which is then converted into code. The specification of the system is done exclusively via UML MARTE [11] models. From the MPSoC model GASPARD provides several transformation chains. As output of a transformation chain, the user expects compilable code which can be used in already available tools. The GASPARD environment permits to select a target into which the SoC should be transformed. The most obvious target is a synthesizable hardware description and application code compilable for this particular hardware. Each chain is a sequence of several model transformations separated by

metamodels and finished by a code generation. It is notable that the majority of auto-parallelization frameworks, like CETUS, use the sequential C code as input and openMP as output. But, despite they have the same objective, they use different transformation techniques. In this paper, we propose an experiment model to evaluate three research auto-parallelization tools, namely PLUTO, CETUS and GASPARD. These tools have the same main goal: automatically generating parallel code with openMP.

3. Measurement model

Experimental Software evaluation is important to discover how some techniques perform, discover its limitations and understand how to improve them [12]. Indeed, in evaluating a system, we need to identify a set of performance metrics that provide adequate information to understand the behavior of the system. Metrics which capture the processor characteristics in terms of the clock speed (MHz), the instruction execution speed (MIPS), the floating point performance (MFLOPS), and the execution time for standard benchmarks (SPEC) have been widely used in modeling uniprocessor performance. A nice property of a uniprocessor system is that given the hardware specifications it is fairly straightforward to predict the performance for any application to be run on the system. However, in a parallel system the hardware specification (which quantifies the available compute power) may never be a true indicator of the performance delivered by the system. This is due to the growth of overheads in the parallel system either because of the application characteristics or certain architectural limitations. So, metrics for parallel system performance evaluation should quantify this gap between available and delivered compute power.
When we wish to evaluate some techniques or process, it is necessary to follow some measurement models that provide the mechanisms to conduct this evaluation. Some mechanisms for defining measurable goals have appeared in the literature like the Goal/Question/Metric Paradigm (GQM) [13] that we choose for defining our measurement model. The GQM paradigm is a mechanism for defining and evaluating a set of operational goals using measurements [13]. A measurement model is defined into three levels: conceptual (goal), operational (question), and quantitative (metric).

In this work, we use the Goal Question Metric (GQM) paradigm for defining our measurement model.
Goal: Evaluate the auto-parallelization tools: CETUS, PLUTO and GASPARD.

Questions:
- What is the parallelization effort of auto-parallelization tools?
- How much performance gain is achieved by parallelizing a given application over a sequential implementation?
- What is its ability to increase performance as number of processors increases?

Metrics to be considered in order to find out answers to these questions are defined as follow:

3.1 Cost/Effort

Cost C reflects the sum of the time that each PE (processor element) spends solving the problem:

C= Parallel runtime x the no. of PEs used (1)

If p=1: The cost of solving a problem on a single PE is the execution time of the fastest known sequential algorithm.

3.2 Speedup

Speedup is a widely used metric for quantifying improvements in parallel system performance as the number of processors is increased. Speedup is defined as the ratio of the time taken by an application of fixed size to execute on one processor to the time taken for executing the same on processors. However, ideal behavior is not achieved because while executing a parallel algorithm, the processors cannot devote 100% of their time to the computations of the algorithm. For Example, part of the time required by the processors to compute the sum of n numbers is spent idling (and communicating in real systems). **Speedup**, S, is the ratio of the time taken to solve a problem on a single
PE to the time required to solve the same problem on a parallel computer with *p* identical PEs.

$$S=Ts/Tp \qquad (2)$$

3.3 Efficiency

So, Efficiency is a measure of the fraction of time for which a processor is usefully employed.
Efficiency E is the ratio of *Speedup S* to the number of PEs (p):

$$E=S/p \qquad (3)$$

In an ideal parallel system efficiency is equal to one. This means that all processor resources are spent on the task. But, rarely the case because of the overhead associated with coordinating processes. Also some parts of a program (such as I/O) might not parallelized. So, in practice, efficiency is between zero and one.

4. Performance evaluation

The experiments were run on a quad-core Intel Core 2 Quad Q6600 CPU clocked at 2.4 GHz (1066 MHz FSB), with a 32 KB L1 D cache, 8MB of L2 cache (4MB shared per core pair), and 2 GB of DDR2-667 RAM, running Linux kernel version 2.6.22 (x86-64). The used compiler is ICC10.0 [16], which was also used to compile the C codes transformed by the three systems.

Table 1. Benchmarks list for evaluating CETUS, PLUTO and GASPARD systems.

Benchmark	Description
MM (Matrix Multiply)	Computes the product of two matrices.
FT (Fast Fourier Transform: NAS parallel benchmarks)	FT contains the computational kernel of a 3-D fast Fourier Transform (FFT)-based spectral method.
LU (Lower-Upper symmetric Gauss-Seidel: NAS parallel benchmarks)	Solve a synthetic system using three different algorithms involving block tridiagonal, scalar pentadiagonal and symmetric successive over-relaxation (SSOR) solver kernels
MVT (Matrix Vector Transpose)	The MVT kernel is a sequence of two matrix vector transposes. It is found within an outer convergence loop with the Biconjugate gradient algorithm.

The NAS Parallel Benchmark (NPB) suite [23] consists of five kernel benchmarks and three pseudo-applications from the field of computational fluid dynamics. The NPB presents an excellent resource for this study, in that it provides multiple language implementations of each benchmark. Table 1 lists the four used parallel benchmarks code for performance evaluation.

- **Runtime and memory usage**

One aspect of evaluating a compiler infrastructure is its efficiency in terms of runtime and memory usage when dealing with realistic applications. Runtime consists mainly of parallelization time, of which the dependence analyzer consumes a major portion. Experiments (table 2) show that PLUTO has the best performance than GASPARD and CETUS for the benchmarks running on the same system. Indeed, CETUS and GASPARD are written in Java which is slower and requires more memory than C or C++. Also, memory usage is driven primarily by the complexity of loops analyzed for dependence testing, in terms of their nesting levels and the total number of array accesses they contain. These factors contributed to CETUS and GASPARD taking a noticeably longer time to process its input than, for instance, the PLUTO compiler.

Table 2. Statistics on loops parallelization with PLUTO, CETUS and GASPARD

	Runtime (sec)	Memory usage (Mbytes)
PLUTO	10	70
CETUS	18	110
GASPARD	25	140

- **parallelization quality**

The performance evaluation of auto-parallelization infrastructure is based on several criteria like the ability to achieve auto-parallelization and to detect parallel and nested loops in the sequential code. In this work, the GASPARD system is tested with only the Matrix Multiply benchmark due to many problems. First, only the Multiply Matrix example is available. Second, in the available GASPARD version, it is not permuted to model our proper parallel application. These preliminary obstacles show the difficulty founded in using a graphical system. But these systems offer an efficient solution to solve the automatic parallelisation problems. They reduce a lot the complexity of development of the parallel programs by their graphic parallel programs modelling in a high level abstraction. Effectively, as shown in Figure 1 GASPARD performs close to or better than PLUTO on MM benchmark, and better than PLUTO in terms of achieve automatic parallelization. Figure 2 show that GASPARD performance can attain to 80% in terms of parallel loops and Nested loop detection, and to 70% in terms of analysis dependence.

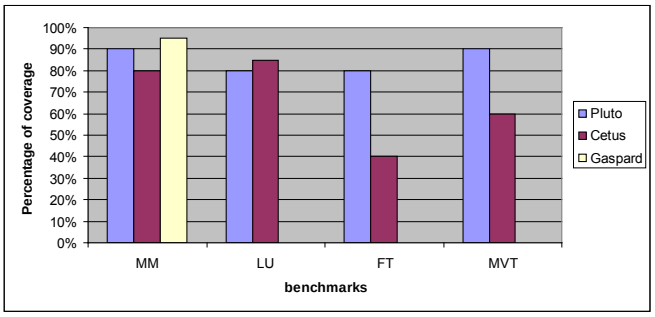

Fig 1. Performance comparison in terms of ability to achieve automatic parallelisation between the different systems

In the case of FT and MVT, CETUS performs poorly than PLUTO. In fact, CETUS is not able to achieve an automatic parallelization. In fact, the codes generated by CETUS need a manual correction to optimally obtain parallel codes. On the other hand, in the LU's case, CETUS performs close to or better than PLUTO. Thus, PLUTO successfully performs the majority of benchmarks by using "–tile" option in the framework execution. In fact, the PLUTO compiler uses the tiling technique which is a key transformation in optimizing for parallelism and data locality.

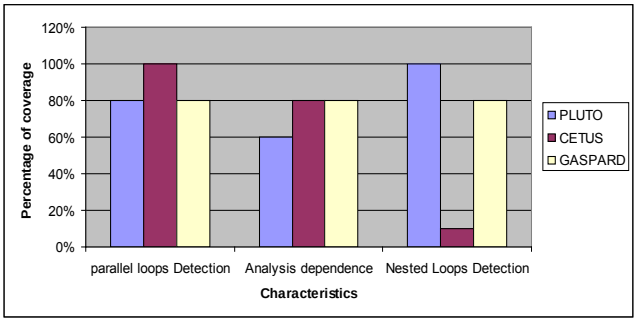

Fig 2. Performance comparison between PLUTO, CETUS and GASPARD systems**Error! Bookmark not defined.**

As shown in figure 2, PLUTO has a high ability to detect nested loops in the contrary of CETUS which poorly detect such loops. On the other hand, the tests show that CETUS detect parallel loops with 100 percent. Thus, CETUS is powerful in terms of parallel loops detection. But, PLUTO and GASPARD have the same efficiency to detect parallel loops. Moreover, GASPARD and CETUS are better than PLUTO in the terms of dependence analysis. In fact, CETUS enables automatic parallelization by using dependence analysis with the dated Banerjee-Wolfe inequalities, array and scalar privatization, reduction-variable recognition, and induction-variable replacement. These are the techniques found to be most important for automatically parallelizing compilers.

In GASPARD' case, the designer creates a model of the application with all the information needed for the implementation, that is: without ambiguity and with all the details concerning the realization of a parallel application. This is the main advantage of the model-based systems.

- **Cost, speedup and efficiency**

This section presents a quantitative comparison between the three selected tools to better evaluate their performance and parallelization quality. Indeed, the performance of a compiler is usually measured in terms of the execution efficiency of compiled code [21]. The results presented in this section are based on metrics described in section 3. The experiments are performed using OpenMP standard functions that calculate the wallclock time between two points in the program. For example, OMP GET WTIME(OpenMP) function is intrusive in the code to evaluate the speedup. CPU time does not include the overhead for parallelization.

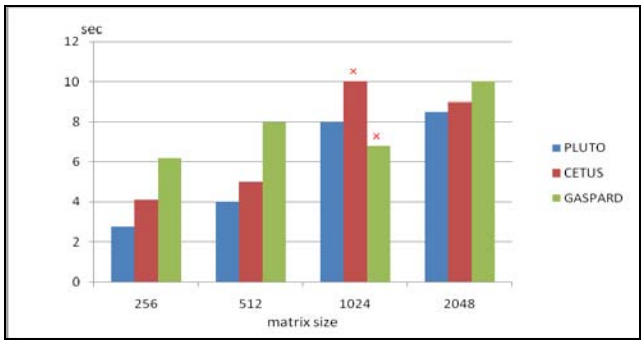

Fig 3. Execution time for parallel Matrix Multiplication

Figure 3 shows the execution time for parallel Matrix Multiplication (size from 256 to 2048) generated by the three tools. We notice that, comparing to PLUTO, CETUS and GASPARD has the higher execution time in all matrix size except in n=1024. Indeed, when we execute the codes transformed by GASPARD and CETUS, we got a run-time error. But, we could only transform successfully by using PLUTO. This is can be explained by insufficient of memory. Indeed, a computer can run out of memory when it is running multiple programs at once or even when running just one or two memory-intensive programs. Running out of available memory causes an error because the computer cannot continue running all of the programs until free memory is available. Execution time is not a serious issue for modestly-sized programs, but it can be a problem for

large benchmarks like LU. Figure 4 shows that when increasing the number of cores (multiplying by 2), the Cost C decrease executing the LU application in the case of both CETUS and PLUTO. So, a super-speedup is achieved.

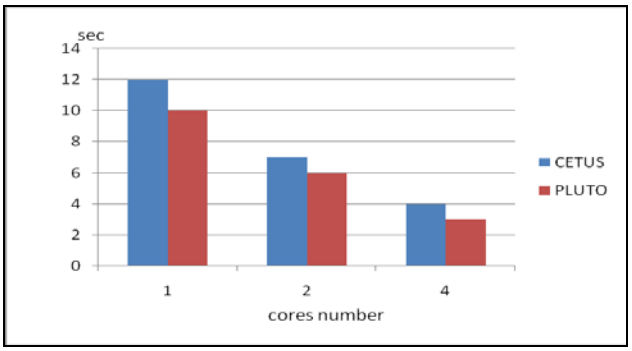

Fig 4. Cost/Effort C running LU application by varying the cores number

Table 3 summarizes the performance of transformed codes using the three tools. The results show a good speedup attained by different cores numbers. Indeed, in the case of MM (n=1024), the three tools have a comparable speedup on dual core design. But, on multi-core design (4 cores), PLUTO has a higher speedup than CETUS and GASPARD. In the case of FT application, PLUTO and CETUS have a comparable speedup on both dual and multi core designs. While, in the case of MVT and LU applications, PLUTO is better than CETUS on both dual and multi core designs.
Figure 5 shows the efficiency of the code generated by the selected tools in the case of multi-core design (4 cores). We notice, firstly, that CETUS performs close to or better than PLUTO on FT application. Secondly, comparing to CETUS and GASPARD, a very high efficiency percentage is attained (97,5%) using PLUTO in the case of MM application. Also, PLUTO is more efficient than CETUS executing LU and MVT application. The final comparison list is presented in table 4.

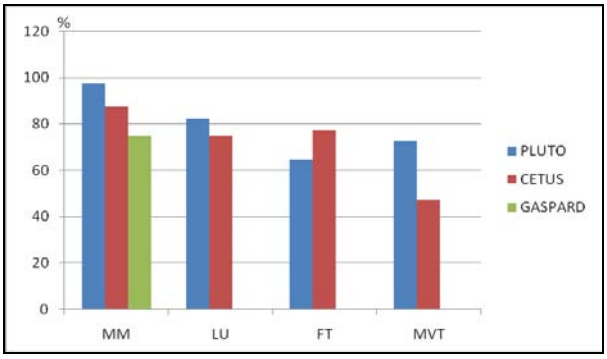

Fig 5. Efficiency percentage

Table 3: speedup comparison

Bench-mark	dual core speedup (2 cores)			Multi-core speedup (4 cores)		
	Pluto	Cetus	Gaspard	Pluto	Cetus	Gaspard
MM	1.9×	1.5×	1.1×	3.9×	3.5×	3×
LU	1.8×	1.5×	-	3.3×	3×	-
FT	1.7×	2.1×	-	2.6×	3.1×	-
MVT	1.1×	0.8×	-	2.9×	1.9×	-

Table 4: comparison list (the smaller is better)

	PLUTO	CETUS	GASPARD
Runtime	1	2	3
Memory usage	1	2	3
Dependency analysis	2	1	1
Parallel loops Detection	2	1	3
Nested loops Detection	1	2	3
Cost/effort	1	2	3
Speed up	1	2	3
Efficiency	1	2	3

From the experimental results in this paper, we find that only PLUTO can transform all benchmarks codes successfully. We conclude also, that CETUS is more efficient than GASPARD. But it gives us, in some cases error results; sometimes we could not transform successfully using CETUS. So, the perfect auto-parallelizing research compiler is yet to be produced. However, there are some cases where auto-parallelization is perfectly suited.

4. Conclusions

Parallel programming is not easy to programmers. So, some frameworks support the auto-parallelization helping them to easily transform sequential codes to parallel codes. In this paper, we presented a comparison between three auto-parallelization tools using a set of criteria. The goal of this assessment was to evaluate the current state of the available automatic parallelization tools when intended to be used by

software designers to develop parallel applications. The comparative study showed that PLUTO is more efficient than the other tools; it gave optimal results for all benchmarks. The advantage of CETUS was his efficiency in terms of dependency analysis and parallel loops detection. But, it gave us error results in detecting and parallelizing nested loops. GASPARD showed the limit of the model-to-source parallelizer comparing to source-to-source parallelizer. It was not flexible and applicable for all benchmarks. But, it gave tolerable results for MM workload.

One common limit of those auto-parallelization tools is the generation of parallel openMP code which depends on the OpenMP API, compiler and OS run time support to realize task partition. However, such support is rarely available in an embedded context where OS is not always present [22]. For future work, we will propose an automatic accelerator generation flow that integrates PLUTO and adapts an application targeting the general purpose processor to an embedded environment. The choice of PLUTO is based on the empirical comparative study presented in this paper.

References

[1] P. Gerner. La sémantique des directives au compilateur: application au parallélisme de données. PhD thesis, Université Louis Pasteur, 2002.

[2] Eric Piel, Philippe Marquet, and Jean-Luc Dekeyser *Model Transformations for the Compilation of Multi-processor Systems-on-Chip*, GTTSE 2007, LNCS 5235, pp. 459–473, 2008. Springer-Verlag Berlin Heidelberg 2008

[3] *Compiler support for general-purpose computation on GPUs*, Yu-Te Lin · Peng-Sheng Chen Springer Science+Business Media, LLC 2008.

[4] B. Chapman, H. Zima, and P. Mehrotra. Extending HPF for advanced data-parallel applications. Parallel & Distributed Technology: Systems & Applications, Jan 1994.

[5]. NVIDIA (2008) NVIDIA CUDA homepage. Website. Online available at http://developer.nvidia.com/object/cuda.html

[6] Chirag Dave, Mansang Bae, Seung-jai Min,Seyong Lee, Rudolf Eignmann, Samuel Midkiff "CETUS: A source-to-Source Compiler Infrastructure for Multicores," computer, PLDI'08, vol.42,no.12,pp .36-42, Dec.2009.

[7] Uday Bondhugula, Albert Hartono, J. Ramanujam, P. Sadayappan "A Practical Automatic Polyhedral Parallelizer and Locality Optimizer", June 7–13, 2008, Tucson, Arizona, USA.

[8] Tianyi David Han, Tarek S. Abdelrahman "hiCUDA: A High-level Directive-based Language for GPU Programming" GPGPU '09 March 8, 2009, Washington, D.C. USA

[9] ROSE, http://www.rosecompiler.org/

[10] PAR4ALL, http://www.par4all.org/

[11] ProMarte partners: UML Profile for MARTE, Beta 1 (August 2007), http://www.omg.org/cgi-bin/doc?ptc/2007-08 04.

[12] V. R. Basili. *The role of experimentation in software engineering: past, current, and future*. In ICSE '96: Proceedings of the 18th international conference on Software engineering, pages 442–449. IEEE Computer Society, 1996.

[13] V. R. Basili, G. Caldiera, and H. D. Rombach. *The goal question metric approach*. Encyclopedia of Software Engineering, 1:528–532, 1995.

[14] A. Lim, S. Liao, and M. Lam. Blocking and array contraction across arbitrarily nested loops using affine partitioning. In ACM SIGPLAN PPoPP, pages 103–112, 2001.

[15] OpenMP Architecture Review Board. OpenMP Application Program Interface, version 3.0. OpenMP Architecture Review Board, May 2008.

[16] http://software.intel.com/en-us/c-compilers, 2013

[17] Xu Zhen , Sun Jizhou , Yu Ce, Wu Huabei , Meng Xiaojing , Tang Shanjiang. A Visual Model-Driven Rapid Development Tool suite for Parallel Applications 2009 World Congress on Computer Science and Information Engineering.

[18] M. Griebl. *Automatic Parallelization of Loop* Programs for Distributed Memory Architectures. University of Passau, 2004. Habilitation thesis.

[19] http://software.intel.com/en-us/intel-parallel-studio-xe, 2013.

[20]http://www.ibm.com/developerworks/wikis/display/hpccentral/IBM+Parallel+Environment+Developer+Edition, 2013.

[21] CHING, W.-M., NELSON, Rick, and SHI, Nungjane. An empirical study of the performance of the APL370 compiler. ACM, 1989.

[22] G. Tian, G Hammami, O. "Performance measurements of synchronization mechanisms on 16PE NOC based multi-core with dedicated synchronization and data NOC". In: International Conference on Electronics, Circuits, and Systems (ICECS'09), 2009, pp. 988 – 991.

[23] NAS Parallel Benchmarks - http://www.nas.nasa.gov/Software/NPB/, 2013

First Author Emna KALLEL: received his Dipl.-Ing. degree in 2006 from the National School of Engineers of Sfax (ENIS) Tunisia, where he is currently working toward the Ph.D.degree with research focused on automatic code generation for

embedded systems. She has been with Enis School, as Research Assistant.
Here further research interests include parallel computing, parallel architectures, automatic parallelisation, Rapid prototyping and video processing and object-oriented methods for hardware generation.

Second Author Yassine Aoudni: Studied Electrical Engineering and Computer software Engineering at the National School of Engineers of Sfax (ENIS) Tunisia. He received Dipl.-Ing.from the National School of Engineers of Sfax in 2002 and Dr.-Ing. from the University of South Brittany, France in 2010. From 2008 till 2011, he worked as a Research Assistant at National School of Engineers of Sfax (ENIS) Tunisia conducting research in FPGA prototyping Since 2011 he is Assistant Professor at the National School of Engineers of Sfax. He acts as a member of several technical program committees, as a reviewer for different journals. His research interests include joint source FPGA prototyping, signal processing, system high level design, parser design, information theory, as well as multiprocessor architecture

Third Author Mohamed Abid: He received Dipl.-Ing. from the National School of Engineers of Sfax (ENIS) in 1985 and the phddegree from the National Institution of applied Science, Toulouse, France. In 2000, he received his doctorale degree in Electrical and Computer Engineering at National Engineering School of Tunis. He is currently Professor at the Electrical Department of ENIS. Since 2006, he has been on the Head of the research laboratory «Computer Embedded System » CES-ENIS. He is responsible for research projects in the area of automatic signal and image processing, wireless networks and information systems. He has been on the Head of Federator Research Project since 2009. He has authored or co-authored more than 120 international conference papers, and he has written more than 20 technical contributions to various international standardization projects. He is a member of the Scientific and Program Committees of several international conferences and workshops.
He is the Co-coordinator of several Nationals and Internationals projects with universities and industries like DGRSRT, CNRS, INRIA, CMCU, training for research, PNM, Tempra, etc.