2015 IEEE Computer Society Annual Symposium on VLSI

Communication-aware Parallelization Strategies for High Performance Applications

(Invited Paper)

Imran Ashraf and Koen Bertels
Computer Engineering Lab,
Delft University of Technology, The Netherlands

Nader Khammassi and Jean-Christophe Le Lann
Lab-STICC UMR CNRS 6285,
ENSTA Bretagne, France

***Abstract*—With the advent of multicore processor architectures and the existence of a huge legacy code base, the need for efficient and scalable parallelizing compilers is growing. Where multi-core processors were seen as the way forward to address the known challenges such as the memory, power and ILP wall, efficient parallelization to make use of the multiple cores, is still an open issue. In this paper, we present two complementary tools, MCROF and XPU which provide an alternative development path to parallelise applications and that address the challenges of identifying potential parallelism and exploiting it in a different way. The MCROF tool provides a detailed profile of the data flowing inside an application and the XPU programming paradigm provides an intuitive and simple interface to express parallelism as well as the necessary runtime support. We demonstrate through two different use cases that better performance up to $4\times$ can be achieved than available commercial compilers.**

***Keywords*—Data-communication profiling, program parallelization, Multicore, Parallel Programming**

I. Introduction

The number of transistors per chip is growing due to technology scaling and increasing the clock rate of processors is becoming technologically less viable [1]. The current trend is therefore to integrate a growing number of processing cores on chip, forcing parallelizing compilers to mature rapidly and to provide efficient code for the multi-core processors. Most parallelizing compilers focus on loop parallelization as most of the execution time is spent in loops. However, scalable parallelism is in many cases not realizable because memory accesses and interprocessor communication are the bottlenecks. Recent research makes it clear that memory accesses and data transfers account for the majority of the power consumption [2] [3] and thus need to be addressed and handled more explicitly in order to achieve (power) efficient performance. This paper presents a tool chain that parallelizes an application based the data flowing inside an application and how this helps in mapping (manually) the algorithm on the architecture using intuitive parallel constructs. We present in detail a use case, Canny Edge Detection, as well as the performance numbers for a second application, fluid animate.

The rest of the paper is organized as follows. Section II discusses some of the available parallelizing compiler frameworks. Section III presents the results of the Canny Edge Detection algorithm when using two state-of-the-art commercial parallelizing compilers. Section IV introduces the tools used in this paper and section V presents the proposed parallelization methodology and achieved results.

II. Related Work

Though static-analysis tools [4] can also track data-communication, a large number of tools [5], [6] utilize dynamic-analysis to collect producer-consumer relationship at runtime. These tools have high run-time overhead, which limits their use for realistic workloads affecting the quality of the generated information. Furthermore, the provided information lacks necessary dynamic details and is not linked to the source code, making it hard to utilize this information. A number of automatic parallelization compilers exist such as Par4All [7], Cetus [8], Parallware [9], Polaris [10] or PolyCC/PLUTO [11] which are source to source compilers that can produce parallel code after analyzing the sequential code using different parallelization techniques. The Intel ICC [12] is a popular compiler which provides an automatic parallelization feature which allows both instruction-level and thread-level parallelization of sequential regions of the input code.

Automatic parallelization of sequential code has had limited success [13]. Great advances have been made in automatic parallelism extraction at the instruction level, however, in order to exploit efficiently modern multicore platforms, compilers need to capture parallelism also at thread-level which is a challenging task. Pareon by Vector Fabrics [14] is an example of a tool, which assists the programmer and guides the parallelization process instead of performing automatic code parallelization.

III. Parallelization using existing Commercial Compilers

In this section, we present the parallelization of the Canny application by using two commercial compilers; we refer to them as CC1 and CC2 in the rest of the discussion. CC1 can be used both in automatic and semi-automatic way, whereas, CC2 uses only a semi-automatic approach. We attempted to use PolyCC/PluTo compiler [11] which uses polyhedral analysis to tile and parallelize loops in sequential programs. However, the compiler suggested significant manual modification of the code in order to make it processable by the compiler. For instance, the compiler requires removing all affine expressions from the inner loop to be able to process it. So the parallelization process is no longer transparent.

978-1-4799-8719-1/15 $31.00 © 2015 IEEE
DOI 10.1109/ISVLSI.2015.89

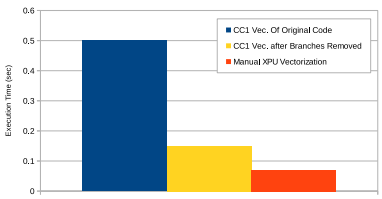

Fig. 1. Performance comparison of automatically vectorized code using CC1 and manually vectorized code.

a) CC1*:* CC1 allows automatic parallelization of sequential program at thread-level using OpenMP and at instruction-level through vectorization using SSE/AVX intrinsics. In order to parallelize a sequential program, the compiler searches for loops which do not expose cross-iteration dependence and are good candidate for parallel execution. A data flow analysis is performed to ensure correct and safe parallel execution. The compiler then uses OpenMP to specify the parallelism.
Default Automatic Mode: is a one shot fully automatic parallelization mode in which the program is parallelized using compile-time static analysis.
Run-time Controlled Mode: collects the run-time information (profiling data) and uses it to guide the program parallelization.
Guided Parallelization Mode: in which compiler can be used to perform run-time analysis to generate an advisory reports, suggesting ways (often code modifications) to the programmer to parallelize loops.

b) Parallelization of Canny Application using CC1*:* For the canny application, the default automatic mode and run-time controlled mode did not show any significant speedup over the original sequential application, hence, we used the guided parallelization mode. By using the advisory reports, the loops in the `gaussian_smooth` were successfully parallelized, however, many other loops could not be parallelized due to false data-flow dependences. Listing 1 illustrates an example where a small manual modification made the loop parallelizable. We made several manual modifications such as nested loop fusion and iterator duplication to help the compiler. The resulting parallel program achieved better speedups (up to $4\times$ for 16 threads on a platform with two Intel Xeon E5-2670 processors at 2.6 GHz).

CC1 is capable of performing instruction-level parallelism by vectorization. In this regard, we performed three experiments to evaluate the vectorization quality in each case:

1) In order to evaluate the vectorization quality, we measured the achieved performance of one of the two loops of the `gaussian_smooth` function. This loop was reported as auto-vectorized by the compiler.
2) In the original code, this loop contains control branches (`if`) to handle loop bounds, which limit vectorization efficiency. We removed these branches.
3) Finally, we vectorized the code manually using SIMD intrinsics (SSE4.2).

The results of these three versions are shown in Figure 1. It can be seen that the manual vectorization achieves significantly better performances than the automatic vectorization performed by the compiler even after vectorization-friendly code transformations.

c) CC2*:* CC2 aims at the parallelization of sequential C/C++ code for ARM Cortex A9 and x86 as target systems. To use CC2, an application is compiled with a C99 compliant compiler to perform instrumentation of the application. The instrumented application is then executed on the model of the target architecture to generate the profile of the application. The generated profile can be visualized in a GUI to select the loops which have high execution contribution as the candidates for parallelization. Furthermore, the dependencies are also reported in a GUI.

After selecting the number of cores in the architecture, CC2 predicts the achievable speedup at the loop-level as well as the application-level. Based on the selected parallelization, refactoring steps are presented by the tool to be applied on the sequential code by the programmer to parallelize it. CC2 suggests parallelizations in terms of a low level threading API and OpenMP pragmas.

```
1
2 // Loop carried dependency on pos in orignal sequential loop
3 for (int pos=0, r=0; r<rows; r++, pos++)
4     for (int c=0; c<cols; c++, pos++)
5         image[pos] = x;
6
7 // equivalent parallelizable loop
8 for (int pos=0; pos<rows*cols; pos++)
      image[pos] = x;
```

Listing 1. CC1 fails to resolve an easily removable data dependencies which prevents parallelization.

d) Parallelization of Canny Application by CC2*:* For the loops in `gaussian_smooth`, CC2 detected that there are no loop dependencies and suggested an OpenMP pragma. For a 4-core system, loop speedup of $4\times$ was predicted, whereas the achieved speedup is only $2.58\times$. Similarly, a $2.2\times$ application speedup was predicted, however, the actual application speedup is $1.64\times$.

For the loop in `magnitude_xy`, CC2 detected `r` as an induction variable, however, for `pos` variable, synchronization was suggested. This synchronization resulted in slowdown, instead of speedup. By changing the code, as already discussed in Listing 1, programmer can avoid this synchronization. Hence, a loop speedup of $2.84\times$ and an application speedup of $1.7\times$ was achieved. Finally, for the third loop in `non_max_supp`, CC2 predicted no speedup, hence, it was not parallelized.

e) Lessons Learned: It can be summarized from the discussion in this section that Parallel compilers, such as CC1, may significantly improve programmer's productivity by automatic code parallelization, however, such compilers suffers from inherent difficulties. Hence, manual code analysis and modifications are required.

CC2 performs run-time analysis to detect dependencies and suggest parallelization. However, there are cases in which even simple loop carried dependencies are not resolved automatically. Therefore, these dependencies are either resolved manually or synchronization is suggested. This synchronization can result in loop (and hence application) slow-down, instead of speedup, requiring manual inspection and modification of application by the programmer. Furthermore, the information is provided among loop iterations, not across loop

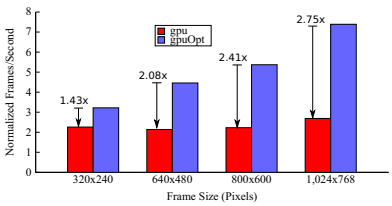

Fig. 2. Normalized Frames per seconds achieved by the GPU and data-communication optimized GPU implementation.

nests or functions in the application to exploit coarse-grained application parallelization and data-flow optimizations.

This outlines the need for tools which analyze the programs dynamically to provide data-flow information to highlight real data-dependences. This information should be provided at various granularity levels to extract and express various forms of parallelization available in the application.

IV. Tools Description

This section briefly introduces the tools involved in this discussion, namely MCROF and XPU.

MCPROF [15] is a runtime memory and communication profiler which generates detailed application profile in terms of memory access patterns and data-communication at function and loop-level granularity. It is based on Intel Pin Dynamic Binary Instrumentation (DBI) framework [16]. MCROF performs instruction-level instrumentation to track memory reads and writes by each instruction. Furthermore, routine-level instrumentation is utilized to keep track of the currently executing function. These are tracked by maintaining a call-stack of the functions executing in the application. To track the dynamic allocations in the application, image-level instrumentation is utilized to selectively instrument library images for memory (re)allocation/free routines.

The tracked memory reads/writes are associated with parts of the source code depending upon the selected granularity i.e. functions, loops or other marked regions in the source code. In this way, a producer-consumer relationship is established between functions/loop/objects in the source code and reported in the form of a communication graph. This production-consumption relation is expressed by edges in the graph where the color of the graph shows the intensity of the communication. Furthermore, the generated graph also shows percentage of the dynamically executed instructions and the number of calls to each function. In this way, communication as well as the computation intensive parts of the application are shown in the generated graph.

The information generated by MCROF has been utilized earlier [15] to perform data-communication aware partitioning of sequential applications on a heterogeneous architecture. Kanade-Lucas-Tomasi Feature Tracker (KLT) [17] application was mapped to a GPU-based platform. After applying the optimizations based on MCROF to the initial GPU implementation (gpu), a data-communication optimized version of the GPU implementation (gpu_{opt}) was obtained. Figure 2 shows the Frames Per Seconds (fps) achieved by both the implementations normalized to the CPU implementation. Increasing the frame size, results in an increase in the amount of computation performed on the GPU. Increased computation results in better utilization of the available resources of the GPU, resulting in higher speedup as can be observed from this figure. On the other hand, increasing the frame size also increases the amount of frame data transferred to the GPU for processing and getting results back. This data-communication has been optimized in the case of gpu_{opt} based on the information provided by MCROF. Hence, gpu_{opt} implementation achieves up-to $2.75\times$ higher speedup as compared to gpu implementation where this communication is not optimized.

XPU [18] [19] is a structured parallel programming framework which aims to easily express and exploit parallelism. XPU allows the expression of different types of parallelism at different levels of granularity. Supported parallelism types includes data parallelism [20], task parallelism [19] and pipeline parallelism [21]. These different parallelism types can be composed hierarchically in the same application [22].

XPU utilizes C++ meta-programming techniques [23] [24] exploiting the potential of the standard C++ language and thus does not require any particular tool except standard C++ compiler. XPU provides a friendly and light weight programming interface which enables the programmer to design parallel applications or parallelize sequential applications with minimal code changes without any significant alteration.

XPU is mainly composed of a set of recurrent parallel execution patterns which specifies execution configuration of a group of tasks. In order to promote reuse of sequential legacy code, these tasks are designed to encapsulate different pieces of code including functions, class methods or lambda expressions. XPU's execution patterns handle transparently many issues such as communication, synchronization and task scheduling. An intelligent run-time system exploits the information which is extracted transparently from both the available hardware resources and perform the resource allocation through cache-aware and load-balanced task scheduling [20], [21].

V. MCROF-XPU Approach

The XPU parallel programming model is designed for easing explicit parallelism expression and therefore requires locating the hot-spots in the program and extracting parallelism by analyzing task-dependencies (producer-consumer relationships). Usually this analysis is performed manually by reading and analyzing the code and profiling the application, which is a time-consuming and error-prone task. MCROF can automate this analysis phase and provides a clear picture of the program with all the required information including task-dependencies, compute-intensive and communication-intensive hot-spots in the application. Hence, MCROF and XPU can form a parallelization tool-chain which offers a much smoother transition from the sequential-code to the parallel-code. The MCROF-XPU parallelization methodology follows the traditional parallelization steps which can be summarized as follows:

1) **Profiling sequential code:** to locate hot-spots.
2) **Extracting parallelism and granularity adjustment:** analyzing data and task dependencies and decomposing tasks to extract more parallelism, if available, at finer grain.
3) **Expressing parallelism**: using the parallel programming API.

4) **Executing the parallel code efficiently:** parallel programming libraries are often based on a run-time which is responsible of efficient scheduling to optimize execution.

A. *Profiling The Sequential Application using* MCROF

Figure 3 shows the MCROF output graph of Canny, where nodes represent functions. Each node contains the function name, the percentage of dynamically executed instructions by this function with respect to the whole application, as well as the total number of calls to this function. For instance, `gaussian_smooth` is executed once and it is a computationally intensive function as it covers 80% of the instructions executed by the whole application.

In addition, the inter-function data-flow is represented by edges. Furthermore, the statically and dynamically allocated objects involved in each flow are also detected by MCROF and represented by rectangles containing object names and allocation sizes. The communication intensity is quantitatively shown by the number of bytes on each edge and also illustrated by the color the edges from red (highest) to green (lowest). In this way, programmer is able to visualize the computation- and communication-intensive parts of an application, in a single graph, without manual source code inspection.

To extract parallelism, coarse-grained functions should be decomposed to extract potential finer-grain parallelism, we refer to this decomposition step as granularity adjustment.

B. *Granularity Adjustment*

As seen in Figure 3, `gaussian_smooth` has 80% execution contribution in the application. In order to extract parallelism inside this function, we use the MCROF feature of splitting functions at loop level granularity. Figure 4 shows the simplified output graph generated by MCROF where the loops in `gaussian_smooth` are split as separate nodes represented by `gaussian_smooth1` and `gaussian_smooth2`. A similar split of `derivative_x_y` is also shown as `derivative_x_y1` and `derivative_x_y2`.

As depicted in the Figure 4, the `gaussian_smooth` function exposes dependencies between its two loop nests, thus they cannot be executed concurrently. However, each of these loops can be parallelized individually since they operate independently on columns and rows of the image. On the other hand, the two parts of the `derivative_x_y` do not expose any dependency and therefore can be executed in parallel, since these loops are using a common read-only input while producing separate outputs. A similar analysis is performed on other functions to extract the available parallelism.

C. *Parallelization Using* XPU

After analyzing the data-flow dependencies and extracting parallelism, we use the XPU framework to express the parallelism. In this regard, XPU skeletons are utilized to exploit various forms of parallelism available in the application.

1) Thread-level parallelism: Thread-level data parallelism can be achieved by parallelizing sequential loops. In XPU, this is expressed by using `parallel_for` pattern. For instance, Listing 2 shows how the `gaussian_smooth` is parallelized using the XPU parallel loop skeleton. The task

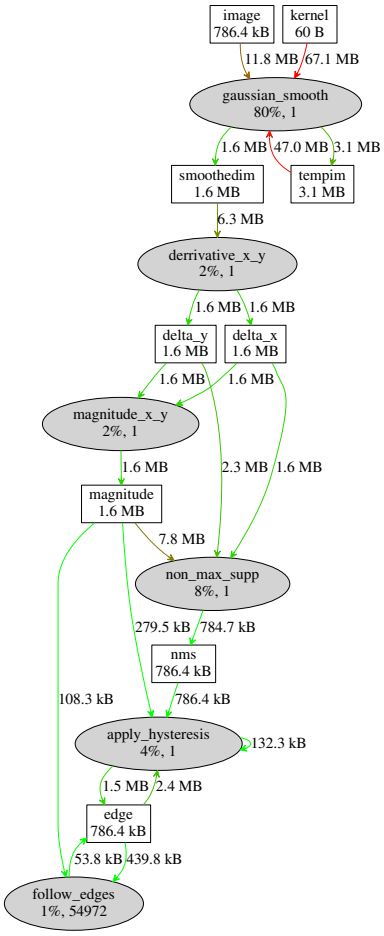

Fig. 3. Task dependency graph of Canny application generated by MCROF.

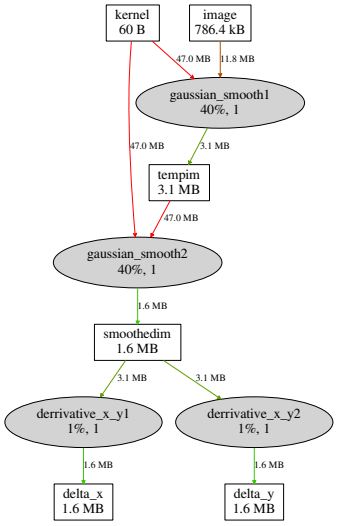

Fig. 4. MCROF profile at loop-level granularity showing independence of the two loop nests inside the *derivative_x_y* function.

```
1 int gaussian_smooth1(char *inImg, float *kernel, char *
      outImg, /* more args */)
2 {   /* process an image row */ }
3 int main(){
4     // task definition
5     xpu::task gauss_task(gaussian_smooth1, image, kernel,
          tempim, /* more args */);
6     // parallel loop construction
7     xpu::parallel_for parallel_gauss(0,size,1,&gauss_task);
8     // parallel loop execution
9     parallel_gaussian.run();
10 }
```

Listing 2. XPU parallel for loop construction and execution.

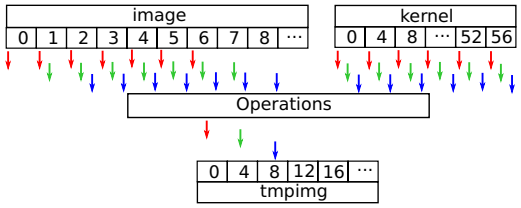

Fig. 5. Fine-grain access-pattern of *gaussian_smooth*1 loop reading *image*, *kernel* and writing *tmpimg* objects. Accesses in same iteration are represented by the same color.

which processes the data elements is constructed and named `parallel_gaussian`. We note that data partitioning and tasks scheduling are handled transparently by the XPU run-time to ensure dynamic forward-scalability and execution-efficiency across different platforms.

2) Instruction-level parallelism: Beside loop parallelization, vectorization can act as a great performance multiplier by allowing SIMD operations on the pixels of the image. XPU provides transparent vectorization through built-in vectorized types such as `vec4f` or `vec8s`, which are implemented using x86 SSE intrinsics. For instance, using `vec4f` allows vectorization of standard operations on single precision `float` and other composite operations which are not available natively in SSE instruction-sets, such as trigonometric functions. Similarly, by using the `vec8s` type, the programmer can operate implicitly on 4 floats simultaneously.

Understanding the data-access pattern is one of the major challenging task in the vectorization process especially in the case of irregular or non-contiguous memory-accesses (load and stores). Tracking data-accesses across loops iterations can be a time-consuming and error-prone task. To address this issue, MCROF can generate a graphical view of the data-access pattern of both input and output data in a user-delimited region of the code, particularly loops. Figure 5 shows the data-access patterns in the first three iterations of the gaussian loop. Each

```
1 xpu::vec4f v1,v2,v3,acc,k1,k2,k3,k4;
2 for (int i=begin; i<end; i++) {
3     v1 = &in[i];   v2 = &in[i+4]; //load inputs
4     v3 = &in[i+8]; v4 = &in[i+12];
5     // vectorized operations :
6     acc = k1*v1 + k2*v2 + k3*v3 + k4*v4;
7     sum = (k1+k2+k3+k4).sum();
8     acc /= sum;
9     out[i-(kernel_size/2)] = acc.sum();  //store output
10 }
```

Listing 3. Code Vectorization using XPU.

```
1 // tasks definition
2 xpu::task dx_t(derrivative_x, smoothed_img, dx);
3 xpu::task dy_t(derrivative_y, smoothed_img, dx);
4 // parallel execution
5 xpu::parallel(&dx_t, &dy_t)->run();
```

Listing 4. Task-level parallelism in *derivative_xy* expressed in XPU.

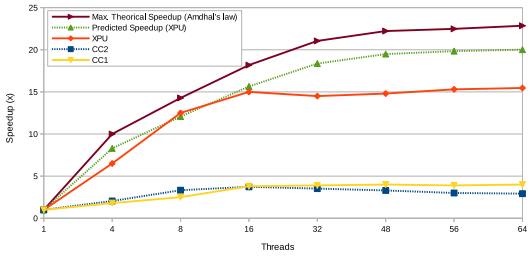

Fig. 6. Achieved speedups over the sequential execution for the Canny application when parallelized by various approaches.

color corresponds to an iteration. In each iteration, we used the XPU vectorized type `vec4f` to load the required inputs from both `image` and `kernel` at the indicated position. Four multiplications and the sum is then performed simultaneously. This allowed us to achieve a significant speedup over both the sequential code and the automatically vectorized code generated by the compiler. Listing 3 shows an example of vectorized code using XPU.

3) Task Parallelism: As depicted in the MCROF output in Figure 4, `derivative_x` and `derivative_y` do not expose any producer-consumer dependencies and thus can be executed as parallel tasks. This can be easily specified using XPU as shown in Listing 4.

Figure 6 shows the speedup achieved for the Canny application parallelized by MCROF-XPU methodology, CC1 and CC2. The theoretical speedup based on Amdahl's law and estimated speedup while considering SIMD support is also shown. It can be seen that a speedup of 15× is achieved for 16 cores on an `Intel Xeon E5-2670`, which is about 4× higher than what achieved by CC1 and Pareon. Similar performances are achieved on 64 cores platform with four `AMD Opteron 6274` processors.

Apart from the Canny application, we have also parallelized *fluidanimate* application which is a part of the PARSEC Benchmark [25]. This benchmark includes three versions of the application: a serial version, a parallel version which uses POSIX Thread API and another parallel version which uses Intel Thread Building Blocks. Another parallel version using XPU has been developed in [18] [20]. We developed a new version using the MCROF-XPU methodology. In the original XPU-based *fluidanimate* version, all the five processing stages were parallelized. However, the MCROF analysis report have shown that some of these stages are not hot-spots which are not worth parallelization. Parallelizing these stages introduces a communication overhead which affects the overall performance. The granularity adjustment using MCROF showed that splitting the `computeForces` processing stage in three sub-stages clearly isolate the computationally intensive regions which should be parallelized.

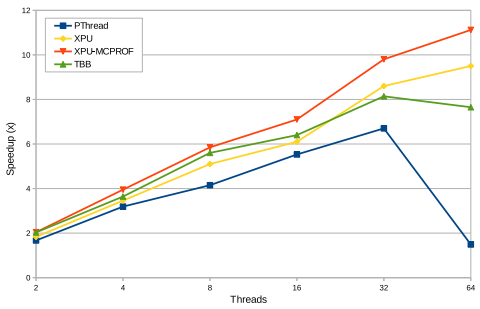

Fig. 7. Performance comparison of the PThread, TBB, XPU only and XPU-MCROF versions of the PARSEC Fluidanimate application.

The XPU `parallel_for` skeleton has been used to express the thread-level data parallelism exposed by different processing stages. The later skeleton provides a scalable data partitioning and uses a cache-aware scheduling policy which promote data reuse and improve spatial and temporal data locality. This scheduling techniques appeared to be particularly beneficial in this study case since each fluid cell is processed on the processor core. Furthermore, we replaced the fluid cells arrays by XPU *vec3f* arrays to take advantage of SSE vectorization. In the reference sequential code, these fluid cells are expressed using regular float arrays.

Figure 7 shows the achieved performance by the four versions. We observe that using MCROF profiling information improved the performances of the original XPU-only version. We note that the performances of the PThread version suffers from significant degradation when more than two processors (32 threads) on the AMD platform (64 cores/4 processors) are used, our investigation have shown that this degradation is caused by the use of barriers which results to an expensive many-to-many communication which affect the performances especially when the four processors are used. The XPU version uses a synchronization mechanism that follows a less-expensive one-to-many communication pattern.

VI. Conclusion

With the emergence of multicore processor architectures, we can no longer avoid parallelizing applications and making parallelizing compilers more efficient is an important objective. In this paper, we have presented the integrated use of two tools which are very complementary in their functionality. As data is in many cases the bottleneck blocking scalable use of multiple computing cores, the need for a detailed analysis of the data flowing through the application (and thus the hardware resources). MCROF provides such a detailed profile which can then be used by XPU, a parallel middle layer and programming approach which provides minimally invasive code changes to express and exploit in a natural way the available parallelism in an application. Not only does the combined approach provide better performance it also reduces substantially the overall time needed to parallelize sequential applications. Future research will allow to fully integrate the approach such that automatic code changes can be introduced.

Acknowledgment

This research is supported by Artemis EMC2 Project (grant 621429), Artemis Almarvi Project (grant 621439) and Artemis CRAFTERS Project (grant 295371). The authors would like to thank Valery Kritchallo for useful discussions.

References

[1] M. Horowitz and W. Dally, "How scaling will change processor architecture," in *ISSCC*, 2004, pp. 132–133 Vol.1.

[2] S. Borkar, "Exascale computing - a fact or a fiction?" in *IPDPS*, 2013.

[3] R. Nair, "Active memory cube: A processing-in-memory approach to power efficiency in exascale systems," in *WoNDP*, 2014.

[4] M. D. Ernst, "Static and dynamic analysis: Synergy and duality," in *WODA*, Portland, Oregon, 2003, pp. 24–27.

[5] W. Heirman *et al.*, "A communication profiler to optimize embedded resource usage," *ProRISC*, 2009.

[6] S. Ostadzadeh, "Quantitative application data flow charac. for heterogeneous multicore architectures," Ph.D. dissertation, TU Delft, Dec 2012.

[7] M. Amini *et al.*, "Par4All: From Convex Array Regions to Heterogeneous Computing," Jan 2012.

[8] C. Dave *et al.*, "Cetus: A Source-to-Source Compiler Infrastructure for Multicores," *Computer*, Dec. 2009.

[9] Appentra, "Parallware," http://www.appentra.com/products/parallware/.

[10] W. Blume *et al.*, "Polaris: Improving the Effectiveness of Parallelizing Compilers," ser. LCPC, London, UK, UK, 1995.

[11] U. Bondhugula *et al.*, "PLuTo: A Practical and Fully Automatic Polyhedral Parallelizer and Locality Optimizer," The Ohio State University, Tech. Rep., Oct 2007.

[12] Intel, "Automatic Parallelization with Intel Compilers," https://software.intel.com/en-us/articles/automatic-parallelization-with-intel-compilers.

[13] J. Shen and M. Lipasti, *Modern Processor Design: Fundamentals of Superscalar Processors*. McGraw-Hill Higher Education, 2002.

[14] Vector Fabrics, "Pareon profile." URL: {http://www.vectorfabrics.com}

[15] I. Ashraf *et al.*, "MCProf: Memory and Communication Profiler," Delft University of Technology, Tech. Rep., November 2014.

[16] C. Luk and et al., "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *PLDI*, 2005, pp. 190–200.

[17] B. D. Lucas and T. Kanade, "An Iterative Image Registration Technique with an Application to Stereo Vision," 1981, pp. 674–679.

[18] N. Khammassi, "High-Level Structured Programming Models For Explicit and Automatic Parallelization on Multicore Architectures." Ph.D. dissertation, ENSTA Bretagne, Lab-STICC, Brest, France, 2014.

[19] N. Khammassi *et al.*, "MHPM: Multi-Scale Hybrid Programming Model: A Flexible Parallelization Methodology," ser. HPCC, Washington, DC, USA, 2012, pp. 71–80.

[20] N. Khammassi et al., "Design and implementation of a cache hierarchy-aware task scheduling for parallel loops on multicore architectures," in *PDCTA*, Sydney, Australia, 2014.

[21] N. Khammassi and J.-C. Le Lann, "A high-level programming model to ease pipeline parallelism expression on shared memory multicore architectures," ser. HPC, San Diego, CA, USA, 2014.

[22] Khammassi, N. and Le Lann, J.C., "Tackling Real-Time Signal Processing Applications on Shared Memory Multicore Architectures Using XPU," ser. ERTS, Toulouse, France, Feb 2014.

[23] J. Koskinen, "Metaprogramming in C++." URL: www.cs.tut.fi/~kk/webstuff/MetaprogrammingCpp.pdf

[24] H. Singh, "Introspective C++," Ph.D. dissertation, Virginia Polytechnic Institute, Virginia, VA, USA, 2004.

[25] C. Bienia *et al.*, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *PACT*, 2008.