Automatic Parallelization for Heterogeneous Computing

Rodrigo Caetano Rocha

Prospective PhD supervisor - Dr. Zheng Wang

Abstract

1 Introduction

During the last several years, modern computers are increasingly parallel and heterogeneous [24, 26]. With parallel architectures being widely spread, parallel programming must extend beyond its traditional realm of scientific applications [19]. The ultimate goal of a programmer in a modern computing environment is to write scalable applications that can take complete advantage of all available resources, since parallelization is an important step that can have serious impact on performance and power consumption [8]. However, parallel programming is known to be difficult and error prone, imposing several challenges to the programmer [8, 22, 23]. Thus, the difficulty of developing parallel software is a key obstacle for an average programmer to exploit the considerable computational power that different multiprocessors offer [24].

In order to automatically leverage all available resources from parallel architectures, automatic parallelization is an optimization that produces parallel code that is semantically equivalent to a sequential code received as input. However, automatic parallelization is a difficult problem since the parallel code must be correct and also perform efficiently on the target machine [39]. Despite intense research interest in the area, it has failed to deliver outside niche domains [33, 37]. Improving the performance of programs for multiprocessors presents a challenge because even when thread-level parallelism exists, it is difficult for a compiler to analyze and identify all true dependencies among potential parallel threads [14].

Furthermore, although this area have been widely studied, there is still much to be explored regarding automatic parallelization that also optimize for energy consumption, embedded systems [9] or other emergent computing architectures [3], such as massively parallel processors including graphic processing units (GPUs), many integrated cores (MICs), and others.

Considering that most programs are executed indefinitely many more times than they are compiled, compilers can afford performing expensive optimization during compilation

time. Previous work suggest different approaches for automatic parallelization based on machine learning techniques [33, 37] and also evolutionary computing [34, 35, 38, 39].

We propose to investigate optimistic approaches, based on machine learning and evolutionary computing, for automatic parallelization. The optimistic approach allows to overcome limitations from traditional approaches based only on static analysis, taking further advantages from the available heterogeneous architecture, regarding parallelism and power consumption.

2 Related work

In this section we present an overview of several work related to automatic parallelization of sequential code, where we discuss several parallelization approaches for both multi-core and heterogeneous systems.

2.1 Automatic parallelism based on static analysis

Numerous early work make use of static analysis techniques in order to parallelize affine loops [4, 5, 17, 20, 24, 27]. However, automatic parallelization based on static analysis is challenging and limited in the type of code that they can parallelize [6, 7, 15]. In order to identify parallel regions of code they must use complex interprocedural analyses to prove that the code has no data dependencies for all possible inputs. Typical code targeted by these compilers consists of nested loops with affine array accesses written in a language with limited aliasing. Large systems written in modern languages usually contain multiple modules and memory aliasing, which makes them not amenable to automatic parallelization. Furthermore, code whose memory access patterns are indeterminable at compile time due to dependence on program inputs can be impossible for these compilers to parallelize [6].

2.2 Thread-level speculative parallelism

In contrast to automatic parallelization that aims at generating a parallel code, thread-level speculation (TLS) approaches performs sequential code parallelization during runtime in a speculative manner. It consists of selecting regions of code to execute in parallel, usually relying on specilized hardware to detect dependence violations and re-execute the conflicting sections such that sequential execution semantics is preserved [7, 14, 40]. Although TLS overcomes limitations intrinsic with conservative compile-time automatic parallelizing tools by extracting parallel threads optimistically and only ensuring absence of data dependence violations at runtime, TSL usually requires a specialized hardware and The overheads associated with maintaining speculative state is a significant barrier for its usage [41].

2.3 Machine learning based approach

In addition to parallelism detection, automatic parallelization mechanisms must also determine whether it is profitable to parallelize a loop and how the loop should be scheduled if a parallel execution is profitable. Recent work [33, 36, 37] have been using ma-

chine learning techniques for integrating automatic portable mapping with automatic parallelism discovery. Prediction mechanisms are applied to each parallel loop candidate, deciding if and how the parallel mapping should be performed.

2.4 Automatic parallelism based on evolutionary computing

Evolutionary computing has previously been applied to automatic parallelization and also several other optimizations in compilers [31, 34, 35, 38, 39]. Previous work on evolutionary parallelization makes no attempt to analyse the original code, instead it attempts to directly performs both loops and instructions parallelization. Evolutionary based approaches have an optimistic parallelization strategy, experimentally validating the best individuals. Although this approach is not limited by dependence analysis techniques, the evolutionary approach produces a probabilistic result, only with statistical guarantees. The correctness of the resulting parallel code is determined by the fitness function, which compares the output produced by the original sequential code and the output produced by each parallel code generated.

2.5 Automatic parallelization for heterogeneous systems

Although most studies focus on automatic parallelization for homogenous shared-memory architectures (CPU only), there have been some recent work [1, 3, 13, 18] toward parallelization for heterogeneous architectures (usually with GPUs). The usual approach consists in detecting affine loops which can be parallelized and executed more efficiently on the GPU rather than on the CPU. There have been work more focused on both low-level transformations (e.g. Leung et al. [18] extend the Java virtual machine for parallelizing Java bytecodes for GPUs) and high-level transformations (e.g. Baskaran et al. [2, 3] propose a source-to-source transformation, using a polyhedral compiler model, also parallelizing for GPUs).

2.6 Performance versus accuracy trade-off

There have been numerous work regarding the trade-off between performance and accuracy in different areas. In contrast with traditional approaches, several work discuss probabilistic and approximate architectures in order to design hardwares with improved performance regarding power consumption, execution time, physical space and architectural simplicity [16, 21, 28, 29]. Some recent work also propose numerous accuracy-aware code transformations with improved performance, regarding execution time and resource consumptions, while producing small accuracy losses [25, 32, 42].

3 Research Themes

It is widely known that detecting parallelism with static compilers in large, general programs is challenging [7, 15]. In order to overcome the limitations of automatic parallelization based on static analysis for general multiprocessors, we intend to explore optimistic approaches that aim at parallelizing aggressively rather than conserva-

tively [7, 38, 39], such as evolutionary approaches aided by profiling and machine learning techniques.

FIX:Breakdown your approach to research themes. Make it clear what problem each theme tries to solve and what's the output of each theme.

3.1 Theme 1: An optimization model

In order to overcome the limitations of automatic parallelization based on static analysis, we intend to define an optimization model for parallelization based on discrete optimization. A discrete optimization model allows for using several integer programming techniques, such as evolutionary computing, simulated annealing, artificial neural networks and others. While this approach is not limited by dependence analysis mechanisms, several challenges need to be addressed.

During the optimization process, each intermediate solution is evaluated by means of profiling techniques. Efficiently profiling each individual is important for reducing the latency involved in the overall parallelization process using the evolutionary approach. Some of the strategies for efficiently profiling include sampling or estimating the performance [10, 11, 12, 25, 32, 42].

3.2 Theme 2: Parallelization for heterogeneous systems and energy consumption

Although automatic parallelization have been widely studied, there is still much to be explored regarding energy consumption, embedded [9] or emergent computing architectures [1, 3, 13, 18], such as massively parallel processors including graphic processing units (GPUs) or many integrated cores (MICs).

Several new challenges are added when parallelizing for heterogeneous systems. In addition to identifying parallelizable loops, it must also decide, for each loop, if the loop parallelization will be profitable, and if so, where it should be executed, i.e., whether to execute it on the CPU, the GPU, or any other available parallel processor [18, 33, 37]. In a heterogeneous system, besides the raw performance of each processor, synchronization, data communication or even energy consumption must also be considered when evaluating how the parallel mapping should be performed.

3.3 Theme 3: Solving dependence violations during runtime

By modeling the automatic parallelization as an optimization problem, defining a parallelization search space in an optimistic manner, both the evolutionary and the machine learning based approaches will produce probabilistic solutions.

If a probabilistic solution is unacceptable, we can use speculative techniques for detecting dependence violations during runtime for re-execution of conflicting sections providing precise solutions. This dependence monitoring mechanism can be implemented entirely in software, similar to Softspec [6], allowing this technique to be used in general parallel hardware. Although this mechanism guarantee correctness, it adds some overhead to the overall execution, which needs to be properly balanced for performance.

It is also possible to integrate techniques from dynamic parallelization, such as those used by the *hybrid analysis* [30]. *Hybrid analysis* combines static and runtime analysis of memory references into a single framework. In the context of heterogeneous systems, Govindarajan and Anantpur [13] describe an algorithm which is able to compute memory dependencies at runtime, where dependence computation and actual computation are pipelined on a GPU.

4 Research plan

Year 1 I will perform an extensive literature review in the area, identifying weaknesses and strengths of current approaches for automatic parallelization.

Year 2

Year 3

5 Expected results

In this research, we expect to improve the state-of-the-art in energy-aware automatic parallelization for heterogeneous systems, overcoming some of the limitations in current dependence analysis mechanisms. The results produced by this research should be published in the main journals and conferences of the area.

FIX:We need a conclusion section and timetable here

References

[1] M. Amini, B. Creusillet, S. Even, R. Keryell, O. Goubier, S. Guelton, J. O. McMahon, F.-X. Pasquier, G. Péan, P. Villalon, and Others. Par4all: From convex array regions to heterogeneous computing. In *IMPACT/HiPEAC*, 2012.

[2] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. A compiler framework for optimization of affine loop nests for gpgpus. In *ICS*, pages 225–234. ACM, 2008.

[3] M. M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic c-to-cuda code generation for affine programs. In *CC/ETAPS*, pages 244–263. Springer-Verlag, 2010.

[4] B. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, B. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford. Polaris: The next generation in parallelizing compilers. In *LCPC*, pages 10–1. Springer-Verlag, 1994.

[5] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *PLDI*, pages 101–113. ACM, 2008.

[6] D. Bruening, S. Devabhaktuni, and S. Amarasinghe. Softspec: Software-based speculative parallelism. In *FDDO*. ACM, 1998.

[7] M. Chen and K. Olukotun. The jrpm system for dynamically parallelizing java programs. In *ISCA*, pages 434–445. IEEE Comp. Soc., 2003.

[8] J. Cockx, B. Vanhoof, R. Stahl, and P. David. System and method for automatic parallelization of sequential code, 2010. US Patent 7,797,691.

[9] D. Cordes, P. Marwedel, and A. Mallik. Automatic parallelization of embedded software using hierarchical task graphs and integer linear programming. In *CODES+ISSS*, pages 267–276. IEEE/ACM, 2010.

[10] T. Fahringer. Estimating and optimizing performance for parallel programs. *Computer*, 28(11):47–56, 1995.

[11] T. Fahringer. *Automatic Performance Prediction of Parallel Programs*. Springer Publishing Company, Incorporated, 1st edition, 2011.

[12] T. Fahringer, A. Pozgaj, J. Luitz, and H. Moritsch. Evaluation of P3T+: a performance estimator for distributed and parallel applications. In *IPDPS*, pages 229–234. IEEE, 2000.

[13] R. Govindarajan and J. Anantpur. Runtime dependence computation and execution of loops on heterogeneous systems. In *CGO*, pages 1–10. IEEE/ACM, 2013.

[14] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. *SIGOPS Oper. Syst. Rev.*, 32(5):58–69, 1998.

[15] K. Kennedy and J. R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc., 2002.

[16] C. M. Kirsch and H. Payer. Incorrect systems: It's not the problem, it's the solution. In *DAC*, pages 913–917. ACM, 2012.

[17] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *POPL*, pages 207–218. ACM, 1981.

[18] A. Leung, O. Lhoták, and G. Lashari. Automatic parallelization for graphics processing units. In *PPPJ*, pages 91–100. ACM, 2009.

[19] J. Li, X. Ma, K. Singh, M. Schulz, B. de Supinski, and S. McKee. Machine learning based online performance prediction for runtime parallelization and task scheduling. In *ISPASS*, pages 89–100. IEEE, 2009.

[20] A. W. Lim and M. S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *POPL*, pages 201–214. ACM, 1997.

[21] A. Lingamneni, K. K. Muntimadugu, C. Enz, R. M. Karp, K. V. Palem, and C. Piguet. Algorithmic methodologies for ultra-efficient inexact architectures for sustaining technology scaling. In *CF*, pages 3–12. ACM, 2012.

[22] M. McCool, J. Reinders, and A. Robison. *Structured parallel programming: patterns for efficient computation*. Elsevier, 2012.

[23] M. D. McCool. Structured parallel programming with deterministic patterns. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Parallelism*, HotPar'10, pages 5–5, Berkeley, CA, USA, 2010. USENIX Association.

[24] S. Misailovic, D. Kim, and M. Rinard. Parallelizing sequential programs with statistical accuracy tests. *ACM Trans. Embed. Comput. Syst.*, 12(2):88:1–88:26, 2013.

[25] S. Misailovic, D. Roy, and M. Rinard. Probabilistically accurate program transformations. In *Static Analysis*, volume 6887 of *Lecture Notes in Computer Science*, pages 316–333. Springer, 2011.

[26] S. Mohanty and M. Cole. Autotuning wavefront abstractions for heterogeneous architectures. In *WAMCA*, pages 42–47. SBC, 2012.

[27] D. A. Padua, R. Eigenmann, J. Hoeflinger, P. Petersen, P. Tu, S. Weatherford, and K. Faigin. Polaris: A new-generation parallelizing compiler for MPPs. Technical report, In CSRD Rept. No. 1306. Univ. of Illinois at Urbana-Champaign, 1993.

[28] K. Palem and A. Lingamneni. What to do about the end of moore's law, probably! In *DAC*, pages 924–929. ACM, 2012.

[29] K. V. Palem, L. N. Chakrapani, Z. M. Kedem, A. Lingamneni, and K. K. Muntimadugu. Sustaining moore's law in embedded computing through probabilistic and approximate design: Retrospects and prospects. In *CASES*, pages 1–10. ACM, 2009.

[30] S. Rus, L. Rauchwerger, and J. Hoeflinger. Hybrid analysis: Static & dynamic memory reference analysis. *International Journal of Parallel Programming*, 31(4):251–283, 2003.

[31] E. Schulte, J. Dorn, S. Harding, S. Forrest, and W. Weimer. Post-compiler software optimization for reducing energy. In *ASPLOS*, pages 639–652. ACM, 2014.

[32] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *ESECP/FSE*, pages 124–134. ACM SIGSOFT, 2011.

[33] G. Tournavitis, Z. Wang, B. Franke, and M. F. P. O'Boyle. Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. In *PLDI*, pages 177–187. ACM SIGPLAN, 2009.

[34] P. Walsh and C. Ryan. Automatic conversion of programs from serial to parallel using genetic programming - the paragen system. In *PARCO*, pages 415–422, 1995.

[35] P. Walsh and C. Ryan. Paragen: A novel technique for the autoparallelisation of sequential programs using gp. In *Proceedings of the 1st Annual Conference on Genetic Programming*, pages 406–409, Cambridge, MA, USA, 1996. MIT Press.

[36] Z. Wang and M. F. O'Boyle. Mapping parallelism to multi-cores: A machine learning based approach. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 75–84, New York, NY, USA, 2009. ACM.

[37] Z. Wang, G. Tournavitis, B. Franke, and M. F. P. O'boyle. Integrating profile-driven parallelism detection and machine-learning-based mapping. *ACM Trans. Archit. Code Optim.*, 11(1):2:1–2:26, 2014.

[38] K. P. Williams and S. A. Williams. Genetic compilers: A new technique for automatic parallelisation. *ESPPE*, pages 27–30, 1996.

[39] K. P. Williams and S. A. Williams. Two evolutionary representations for automatic parallelization. In *GECCO*, volume 2, pages 1429–1436, Orlando, Florida, USA, 1999. Morgan Kaufmann.

[40] P. Wu, A. Kejariwal, and C. Caşcaval. Compiler-driven dependence profiling to guide program parallelization. In *LCPC*, volume 5335 of *Lecture Notes in Computer Science*, pages 232–248. Springer Berlin Heidelberg, 2008.

[41] P. Yiapanis, D. Rosas-Ham, G. Brown, and M. Luján. Optimizing software runtime systems for speculative parallelization. *ACM Trans. Archit. Code Optim.*, 9(4):39:1–39:27, 2013.

[42] Z. A. Zhu, S. Misailovic, J. A. Kelner, and M. Rinard. Randomized accuracy-aware program transformations for efficient approximate computations. In *POPL*, pages 441–454. ACM, 2012.