Automatic Parallelization for Heterogeneous Computing

Rodrigo Caetano Rocha

Prospective PhD supervisor - Dr. Zheng Wang

Abstract

1 Introduction

FIX:ZW: The introduction seems to be too long. I think you should make sure you get to the point to describe your approach on the first paper

During the last several years, modern computers are increasingly parallel and heterogeneous [28, 30]. With parallel architectures being widely spread, parallel programming must extend beyond its traditional realm of scientific applications [22]. 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 [11]. However, parallel programming is known to be difficult and error prone, imposing several challenges to the programmer, such as race conditions, deadlocks, or even non-determinism [11, 25, 26]. 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 [28].

Although manual parallelization by expert programmers is known to usually results in the most efficient parallel implementation, it is a costly and time-consuming approach, besides requiring specialized knowledge about parallelism and the target architecture. As hardware parallelism increases in scale with each generation and programming costs increase, automated parallelizing technology becomes extremely attractive with the potential to greatly reduce the costs while still profiting from the parallel architecture [39, 44].

Automatic parallelization is an optimization that produces parallel code that is semantically equivalent to a sequential code received as input. This is a difficult problem since the parallel code must be correct and also perform efficiently on the target machine [46]. Despite intense research interest in the area, it has failed to deliver outside niche domains [39, 44]. 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 between potential parallel threads [17].

The traditional approach had been to concentrate on restructuring computationally intensive loops using dependence analysis to determine which loops can be parallelized. This analytical approach has only proved successful with a small and restricted-set of programs [46]. Traditional static parallelism detection techniques are not effective in finding parallelism due to limitations on the level of information that can be confidently acquired during static analysis [39, 44]. Another limitation in traditional static parallelism relates to the combination of automatic parallelism discovery and portable mapping. Given that the number and type of processors of a parallel system is likely to change from one generation to the next, finding the right mapping for an application may have to be repeated many times throughout an application's lifetime, hence, making automatic approaches attractive [39, 44].

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 [12] 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 [39, 44] and also evolutionary computing [40, 41, 45, 46]. However, extensive research into automatic parallelization over the years has lead to the development of a large body of heuristic and analytical techniques which may be included into such modern approaches [46].

FIX:This section should answer four questions: 1. Why bother? (What problems the academic/industry will have if this problem is not solved? 2. Why it has been solved yet? 3. How will you solve this problem? (This is missing) 4. How much better the world will be if the proposed approach works? (Summarize your impact.)

FIX:ZW: I think this section should be shorten and focus answering the previous four questions. Try to use terms that can be understood by a layman.

2 Related work

FIX:One paragraph per topic–not more.

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 [6, 7, 20, 23, 28, 32]. However, automatic parallelization based on static analysis is challenging and limited in the type of code that they can parallelize [8, 10, 18]. 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 [8].

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 [10, 17, 47]. 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 [48].

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 [39, 43, 44] have been using machine 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 [36, 40, 41, 45, 46]. 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, 16, 21] toward parallelization for heterogeneous architectures, usually GPUs.

Leung et al. [21] extend the Java virtual machine to detect affine loops, in plain Java bytecodes, which can be parallelized and executed more quickly on the GPU rather than on the CPU. They present a cost model that balances the higher raw performance of the available GPU against the cost of transferring input and output data between main memory and GPU memory. There are several obstacles that need to be overcome when parallelizing Java bytecode language, such as unstructured control flow, the lack of multi-dimensional arrays, the precise exception semantics, and the proliferation of indirect references.

Baskaran et al. [2, 3] propose a framework for automatic parallelization and optimization of affine loops on GPUs. A polyhedral compiler model of data dependence is used to perform program transformation for efficient data access from GPU global memory. Their framework performs source-to-source transformations, generating CUDA from C code, optimized for efficient data access, including tiling strategies. They use the PLuTo [7] polyhedral parallel tiling infrastructure and the polyhedral code generator called CLooG [4], which transforms a polyhedral representation of a program and affine scheduling constraints into final loop code.

Parallelizing techniques has several limitations, one such limitation is regarding indirect array accesses. Govindarajan and Anantpur [16] present an algorithm for executing on GPU, loops with cross iteration dependencies due to indirect memory accesses. The algorithm described is able to compute memory dependencies at runtime, where dependence computation and actual computation are performed in pipeline.

2.6 Performance versus accuracy trade-off

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 [19, 24, 33, 34]. In contrast with traditional approaches, these probabilistic and approximate architectures does not attempt to correct the errors introduced by components which are susceptible to perturbations, allowing for errors to arise with small probabilities. For example, Palem et al. demonstrate a XOR gate with a 3× reduction in energy consumption for a 2.3% decrease in the probability of correctness.

Some recent work also propose numerous code transformations aiming at the trade-off between performance and accuracy. For certain types of applications, accuracy-aware program transformations may generate error-tolerant code that may perform better than the original code [49]. The main transformations are: (*i*) *substitution transformations* replace parts of a program with code that computes approximations of the output computed by the original parts but with less computational overhead; (*ii*) *sampling transformations* produces a transformed code that performs the same computation as the original code but only on a subset of the elements obtained by some sampling policy; (*iii*) *loop perforation* is another type of program transformation which transforms a loop into a new loop where only a subset of the original loop iterations is performed. Loop perforation has been studied by several work showing a reduction in execution time and resource consumptions while producing small accuracy losses [29, 37].

3 Research Themes

FIX:ZW: Try to summarize your approach in the first paragraph of this section

It is widely known that detecting parallelism with static compilers in large, general programs is challenging [10, 18]. In order to overcome the limitations of automatic parallelization based on static analysis for general multiprocessors, we intend to explore optimistic and evolutionary approaches aided by profiling and machine learning techniques. We aim at approaches where it is possible to parallelize aggressively rather than conservatively [10, 45, 46]. In this section we suggest some possible research directions and their respective challenges.

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 An evolutionary approach

FIX:I think this goes into too much details. Bear in mind the reviewer is unlikely to be an expert in compilers or machine learning. Try to explain your ideas without using jargons.

Automatic parallelization using evolutionary computing has an optimistic approach, where the code is parallelized more aggressively. Each individual of a generated population can then be scored based on profiling techniques. While this approach is not limited by dependence analysis mechanisms, several challenges need to be addressed, including: (*i*) define a genetic representation for the source code, the parallel transformations and optimizations; (*ii*) define a fitness function that score each individual considering the trade-off between performance and accuracy; (*iii*) measure, with statistical confidence, the accuracy of the parallelized source code; (*iv*) efficiently profile each individual.

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 the program corresponding to each individual include: simulating or actually executing the program; executing a transformed version of the individual's program after performing sampling or loop perforations [29, 37, 49]; estimating the performance without actually executing the program [13, 14, 15].

Besides the traditional *Darwinian evolution model* for randomly creating the initial population, mutations and crossovers, it is also possible to use the *learnable evolution model* [9, 27], where the individuals are *genetically engineered*. In the *learnable evolution model*, individuals can be created guided by static analysis, profiling, or machine learning.

3.2 A machine learning based approach

In addition to being used for mapping parallelism to multi-core processors [39, 43, 44], machine learning can also be used for parallelizing a sequential code. Similar to the evolutionary approach, once we have optimistically identified all once the optimization and parallelization search space has been identified in an optimistic manner, by analyzing the sequential code, machine learning techniques, such as ANN [31, 42], can be used for automatically selecting the best parallelization.

Several work propose solutions using neural networks for a variety optimization problems in general [31, 38, 42], which can be applied either using supervised or unsupervised learning [5].

3.3 Solving dependence violations during runtime

By modeling the automatic parallelization problem 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 [8], 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* [35]. *Hybrid analysis* combines static and runtime analysis of memory references into a single framework. In the context of heterogeneous systems, Govindarajan and Anantpur [16] describe an algorithm which is able to compute memory dependencies at runtime, where dependence computation and actual computation are pipelined on a GPU.

3.4 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 [12] or emergent computing architectures [1, 3, 16, 21], 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 [21, 39, 44]. 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.

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] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT*, pages 7–16. IEEE, 2004.

[5] Y. Bengio, P. Lamblin, D. Popovici, H. Larochelle, U. D. Montréal, and M. Québec. Greedy layer-wise training of deep networks. In *NIPS*. MIT Press, 2007.

[6] 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.

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

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

[9] G. Cervone, R. S. Michalski, K. K. Kaufman, and L. A. Panait. Combining machine learning with evolutionary computation: Recent results on LEM. In *IWML*, pages 41–58, 2000.

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

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

[12] 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.

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

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

[15] 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.

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

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

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

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

[20] 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.

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

[22] 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.

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

[24] 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.

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

[26] 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.

[27] R. S. Michalski. Learnable evolution model: Evolutionary processes guided by machine learning. *Machine Learning*, 38(1-2):9–40, 2000.

[28] 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.

[29] 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.

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

[31] M. Ohlsson, C. Peterson, and B. Söderberg. Neural networks for optimization problems with inequality constraints: The knapsack problem. *Neural Comput.*, 5(2):331–339, 1993.

[32] 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.

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

[34] 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.

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

[36] 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.

[37] 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.

[38] K. A. Smith. Neural networks for combinatorial optimization: A review of more than a decade of research. *INFORMS Journal on Computing*, 11(1):15–34, 1999.

[39] 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.

[40] 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.

[41] 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.

[42] B. Wang, H. Dong, and Z. He. A chaotic annealing neural network with gain sharpening and its application to the 0/1 knapsack problem. *Neural Processing Letters*, 9(3):243–247, 1999.

[43] 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.

[44] 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.

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

[46] 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.

[47] 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.

[48] 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.

[49] 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.