Automatic Parallelization for Heterogeneous Computing

Rodrigo Caetano Rocha

Prospective Ph.D. supervisor - Dr. Zheng Wang

Abstract

1 Introdução

Durante os últimos anos, os computadores modernos estão cada vez mais paralelos e heterogêneos [24, 26]. Com arquiteturas paralelas sendo amplamente difundido, a programação paralela deve estender-se para além de sua esfera tradicional de aplicações científicas [19]. O objetivo final de um programador em um ambiente de computação moderno é escrever aplicações escaláveis que podem aproveitar por completo de todos os recursos disponíveis, uma vez que paralelização é um aspecto importante que pode produzir grande impacto sobre o desempenho e consumo de energia [8]. No entanto, programação paralela é conhecida por ser difícil e sujeita a erros, impondo vários desafios ao programador [8, 22, 23]. Assim, a dificuldade de desenvolvimento de software paralelo é um dos principais obstáculos para um programador médio explorar consideravelmente o poder computacional que diferentes multiprocessadores oferecem [24].

A fim de aproveitar de forma transparente todos os recursos disponíveis por arquiteturas paralelas, paralelização automática é uma otimização que produz código paralelo que é semanticamente equivalente a um código seqüencial recebido como entrada. No entanto, paralelização automática é um problema difícil uma vez que o código paralelo deve ser correto e também executar de forma eficiente na máquina alvo [39]. Apesar do interesse de pesquisa intenso na área, ainda não somos capazes resolver fora de alguns nichos de domínios [33, 37]. Melhorar o desempenho dos programas para multiprocessadores apresenta um desafio porque mesmo quando existe um nível de paralelismo, é difícil para um compilador analisar e identificar todas as dependências reais entre os potenciais *threads* paralelas [14].

Além disso, embora esta área tenha sido amplamente estudado, ainda há muito para ser explorado em relação a paralelização automática que também otimiza para consumo de energia, sistemas embarcados [9] ou outras arquiteturas emergentes de computação [3], tais como processadores massivamente paralelos, incluindo unidades de processamento gráfico (GPUs), processadores de muitos núcleos integrados (MICs), e outros.

Considerando que a maioria dos programas são executados indefinidamente muito mais vezes do que são compilados, compiladores podem suportar otimizações custosas

durante o processo de compilação. Trabalhos anteriores sugerem diferentes abordagens para paralelização automática baseado em técnicas de aprendizagem de máquina [33, 37] e também em computação evolutiva [34, 35, 38, 39].

Propomos investigar abordagens otimistas, com base no aprendizado de máquina e computação evolutiva, para a paralelização automática, aproveitando ainda das arquiteturas heterogêneas disponíveis, referente a paralelismo e consumo de energia. A abordagem otimista permite superar as limitações das abordagens tradicionais baseadas apenas na análise estática, penalizando precisão com soluções probabilísticas.

2 Trabalhos relacionados

Nesta seção, apresentamos uma visão geral dos vários trabalhos relacionados sobre paralelização automática de código seqüencial, onde discutimos várias abordagens de paralelização para ambos sistemas multi-core e heterogêneos.

2.1 Paralelismo automático baseado em análise estática

Inúmeros trabalhos iníciais fazem uso de técnicas de análise estática a fim de paralelizar laços afins [4, 5, 17, 20, 24, 27]. Contudo, paralelização automática baseada em análise estática é um desafio e limitada ao tipo de código capaz de paralelizar [6, 7, 15]. A fim de identificar regiões paralelas no código, deve-se usar análises complexas interprocedurais para provar que o código não possui dependências de dados para todas as entradas possíveis. Códigos típicos visados por estes compiladores consistem de laços aninhados com acessos afins de arranjos escrito em uma linguagem com aliasing limitado. Grandes sistemas escritos em linguagens modernas geralmente contêm vários módulos e aliasing de memória, o que os tornam não passíveis de paralelização automática. Além disso, código cujo padrão de acesso à memória é indeterminável em tempo de compilação devido às dependências das entradas do programa podem ser impraticável para esses compiladores paralelizarem [6].

2.2 Paralelismo especulativo no nível de threads

Em contraste com a paralelização automático que visa gerar código paralelo, a abordagem de especulação no nível de thread (TLS) realiza paralelização de código seqüencial em tempo de execução de maneira especulativa. Consiste em selecionar regiões de código para executar em paralelo, geralmente contando com hardware especializado para detectar violações de dependência e re-executar seções com conflito de tal forma que a semântica de execução seqüencial é preservada [7, 14, 40]. Embora TLS supera as limitações intrínseca às ferramentas conservadoras de paralelização automática em tempo de compilação, extraindo segmentos paralelos de maneira otimista e garantindo a ausência de violações de dependência dados apenas em tempo de execução. TLS geralmente requer um hardware especializado e os custos associadas com a manutenção do estado especulativo é um obstáculo significativo para a sua utilização [41].

2.3 Abordagens baseadas em aprendizagem de máquina

Além da detecção de paralelismo, mecanismos de paralelização automática devem também determinar se é rentável para paralelizar um laço e como o laço deve ser escalonado se a execução paralela for rentável. Trabalhos recentes [33, 36, 37] têm utilizado técnicas de aprendizagem de máquina para integrar o mapeamento automático e portátil com a descoberta automático de paralelismo. Mecanismos de previsão são aplicados a cada laço candidato à paralelismo, decidindo se e como o mapeamento paralelo deve ser realizado.

2.4 Paralelismo automático baseado em computação evolucionária

Computação evolutiva foi previamente aplicada a paralelização automática e também várias outras otimizações em compiladores [31, 34, 35, 38, 39]. Trabalhos anteriores sobre paralelização evolutiva não faz nenhuma tentativa de analisar o código sequencial original, em vez disso, tenta realizar diretamente a paralelização de laços e instruções. Abordagens evolutivas têm uma estratégia de paralelização otimista, experimentalmente validando os melhores indivíduos. Embora esta abordagem não é limitada por técnicas de análise de dependências, a abordagem evolutiva produz resultados probabilísticos, apenas com garantias estatísticas. A correção do código paralelo resultante é determinada pela função de avaliação, que compara a saída produzida pelo código seqüencial original e a saída produzida por cada código paralelo gerado.

2.5 Paralelização automática para sistemas heterogêneos

Embora a maioria dos estudos focam em paralelização automática para arquiteturas homogéneas de memória compartilhada (somente CPU), tem havido alguns trabalhos recentes em direção a paralelização para arquiteturas heterogêneas (geralmente com GPUs). A abordagem usual consiste em detectar laços afins que podem ser paralelizados e executados de forma mais eficiente na GPU ao invés da CPU. Tem havido trabalhos mais focados em ambas as transformações de baixo nível (por exemplo, Leung et al. [18] estende a máquina virtual do Java para paralelização de bytecodes Java para GPUs) e transformações de alto nível (por exemplo, Baskaran et al. [2, 3] propõem uma transformação entre códigos fonte, utilizando um modelo de compilador poliédrico, também paralelizando para GPUs).

2.6 Desempenho versus precisão

Existem inúmeros trabalhos sobre o equilíbro entre desempenho e precisão em diferentes áreas. Em contraste com as abordagens tradicionais, vários trabalhos discutem arquiteturas probabilísticas e aproximadas, a fim de projetar hardwares com melhor desempenho em matéria de consumo de energia, tempo de execução, espaço físico e simplicidade arquitetural [16, 21, 28, 29]. Alguns trabalhos recentes também propõem numerosas transformações de código visando precisão com melhor desempenho, tempo de execução e o consumo de recursos, enquanto produz pequenas perdas de precisão [25, 32, 42].

3 Temas de pesquisa

É amplamente conhecido que a detecção de paralelismo por compiladores estáticos em programas grandes e gerais é um desafio [7, 15]. A fim de superar as limitações de paralelização automática com base em análise estática para multiprocessadores gerais, pretende-se explorar abordagens otimistas que visam paralelizar de maneira agressiva ao invés de conservadora [7, 38, 39], tais como abordagens evolutivas auxiliadas por técnicas de *profiling* e aprendizagem de máquina.

3.1 Tema 1: Um modelo de otimização

A fim de superar as limitações da paralelização automática baseada em análise estática, temos a intenção de definir um modelo de otimização para paralelização baseado em otimização discreta. Um modelo de otimização discreta permite usar várias técnicas de programação inteira, como a computação evolucionária, *simulated annealing*, *hill climbing* e outros. Enquanto essa abordagem não é limitada por mecanismos de análise de dependências, vários desafios precisam ser abordados.

Durante o processo de otimização, cada solução intermediária é avaliada por meio de técnicas de *profiling*. Avaliar eficientemente cada indivíduo é importante para reduzir a latência envolvida em todo processo de paralelização usando uma abordagem evolutiva. Algumas das estratégias para avaliar eficientemente incluem realizar amostragem ou estimar o desempenho o desempenho do código paralelizado [10, 11, 12, 25, 32, 42].

3.2 Tema 2: Paralelização para sistemas heterogêneos e consumo de energia

Apesar de que paralelização automática têm sido amplamente estudada, ainda há muito a ser explorado em relação ao consumo de energia, arquiteturas de computação embarcada [9] ou emergentes [1, 3, 13, 18], como processadores massivamente paralelos, incluindo unidades de processamento gráfico (GPUs) ou processadores de muitos núcleos integrados (MICs).

Vários novos desafios são adicionados quando paralelizamos para sistemas heterogêneos. Além de identificar laços paralelizáveis, deve-se também decidir, para cada laço, se a paralelização do laço será rentável, e em caso afirmativo, onde ele deve ser executado, ou seja, se será executado na CPU, GPU, ou qualquer outro processador paralelo disponível [18, 33, 37]. Em um sistema heterogêneo, além do desempenho bruto de cada processador, sincronização, comunicação de dados ou até mesmo o consumo de energia também devem ser considerados quando se avalia como o mapeamento paralelo deve ser realizado.

Temos a intenção de estender o modelo de otimização discreta, a fim de suportar paralelização sensível à consumo energéticos, além de integrar com arquiteturas heterogêneas, como MICs e GPUs. O modelo de otimização deve considerar a rentabilidade da paralelização para a arquitetura disponível, resultando em um modelo que engloba mapeamento portátil.

3.3 Tema 3: Solução de violações de dependência durante o tempo de execução

Abordagens otimistas para paralelização automática produzem soluções probabilística. Se uma solução probabilística for inaceitável, podemos usar técnicas especulativas para detectar violações de dependência em tempo de execução para re-executar seções conflitantes, fornecendo soluções precisas. Este mecanismo de controle de dependência pode ser totalmente implementado em software, semelhante ao Softspec [6], permitindo que esta técnica seja utilizada em hardwares paralelos comuns. Embora este mecanismo garante a correção, ele adiciona alguns custos para a execução como um todo, o que precisa ser devidamente equilibrado com questões de desempenho.

Também é possível integrar técnicas de paralelização dinâmica, tais como as utilizadas em análises híbridas [30], que combinam a análise estática com análises em tempo de execução de referências de memória em uma única plataforma. No contexto de sistemas heterogêneos, Govindarajan e Anantpur [13] descrevem um algoritmo que é capaz de calcular as dependências de memória em tempo de execução, onde a computação de dependências e a computação real são executadas diretamente em uma 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. The discrete optimization model (Theme 1) is intended to be developed during the first year.

Year 2 I will extend the optimization model regarding energy consumption. Afterwards, the automatic parallelization will be integrated with heterogeneous architectures (Theme 2). One or two papers are expected during the second year.

Year 3 During the third year, the optimistic automatic parallelization will be extended with solutions for dependence violations during runtime (Theme 3). One or two papers as well as the Ph.D. thesis are expected during the third and final year.

4.1 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.

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.