Towards Effective Automatic Parallelization for Multicore Systems

Uday Bondhugula[1] Muthu Baskaran[1] Albert Hartono[1] Sriram Krishnamoorthy[1]
J. Ramanujam[2] Atanas Rountev[1] P. Sadayappan[1]

[1]Dept. of Computer Science and Engineering
The Ohio State University
2015 Neil Ave. Columbus, OH, USA
{bondhugu,baskaran,hartonoa,krishnsr,rountev,saday}@cse.ohio-state.edu

[2]Dept. of Electrical & Computer Engg. and
Center for Computation & Technology
Louisiana State University
jxr@ece.lsu.edu

Abstract

The ubiquity of multicore processors in commodity computing systems has raised a significant programming challenge for their effective use. An attractive but challenging approach is automatic parallelization of sequential codes. Although virtually all production C compilers have automatic shared-memory parallelization capability, it is rarely used in practice by application developers because of limited effectiveness. In this paper we describe our recent efforts towards developing an effective automatic parallelization system that uses a polyhedral model for data dependences and program transformations.

1 Introduction

Current trends in microarchitecture are increasingly towards larger number of processing elements on a single chip. This has led to parallelism and multi-core architectures becoming mainstream. In addition, several specialized parallel architectures (accelerators) like the Cell processor and General-Purpose GPUs have emerged. The difficulty of programming these architectures to effectively tap the potential of multiple on-chip processing units is a significant challenge. Among several approaches to addressing this issue, one that is very promising but simultaneously very challenging is automatic parallelization. This requires no effort on part of the programmer in the process of parallelization and optimization and is therefore very attractive.

Automatic parallelization through compiler analysis and transformation has been a goal from the early days of the Illiac IV. Over the last three decades, although there has been significant progress in compiler techniques towards automatic parallelization, the current state-of-practice leaves much to be desired. Automatic parallelization has been available in commercial compilers for many years. But unlike vectorization technology, which was indeed heavily used in practice by developers of production application codes on vector machines, automatic parallelization across multiple processors has not yet been sufficiently effective to draw much interest from application developers. Intel's production compiler incorporates automatic parallelization and automatic vectorization. But while its automatic vectorization capability is very good, its automatic parallelization is not very effective, as we show with experimental data later in the paper.

Many compute-intensive applications often spend most of their running time in nested loops. This is particularly common in scientific and engineering applications. The polyhedral model provides a powerful abstraction to reason about transformations on such loop nests by viewing a dynamic instance (iteration) of each statement as an integer point in a well-defined space called the statement's *polyhedron*. We are developing an automatic parallelization framework based on some very recent developments [7, 6, 8] showing great promise. The key to our approach is the use of a model that allows us to model transformations on collections of loop nests. In this model, a dynamic instance (iteration) of each statement is viewed as an integer point in a well-defined space called the statement's polyhedron. With such a representation for each statement and a precise characterization of inter or intra-statement dependences, it is possible to reason about the correctness of complex loop transformations in a completely mathematical setting using powerful machinery from linear algebra and integer linear programming. With the conventional abstractions for data dependences used in most optimizing compilers (including gcc and all vendor compilers), it is virtually impossible to perform integrated model-driven optimization using multiple loop transformations

This paper is organized as follows. Section 2 provides an overview of the polyhedral transformation system and presents some experimental results. Section 3 discusses our work on extending the polyhedral transformation framework to generate optimized code for accelerators such as GPGPUs that have software-managed scratchpad memory. In Section 4 we summarize recent work on enabling enhanced concurrency with tiled execution. Section 5 discusses related work and conclusions are presented in Section 6.

2 Polyhedral Transformation Framework

The task of program optimization for parallelism and locality in the polyhedral model may be viewed in terms of three phases: (1) static dependence analysis of the input program, (2) transformations in the polyhedral abstraction, and (3) generation of code for the transformed program. Significant advances were made in the past decade on de-

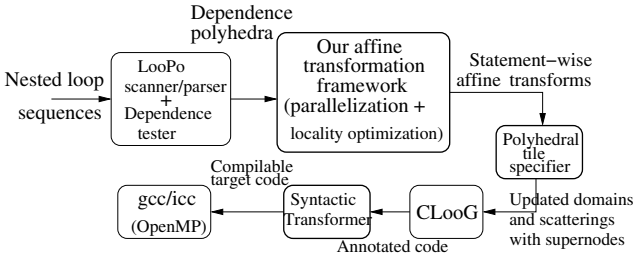

Figure 1. Our source-to-source transformation system prototype

pendence analysis [15, 14, 31] and code generation [23, 20] in the polyhedral model, but the approaches suffered from scalability challenges. Recent advances in dependence analysis and more importantly in code generation [32, 5, 38] have solved many of these problems resulting in the polyhedral techniques being applied to code representative of real applications like the spec2000fp benchmarks. But a significant limitation has been the absence of a scalable and practical approach for automatic transformation for parallelization and locality. Our recent work has sought to address this problem and very promising progress has been made towards developing a compiler framework that enables end-to-end fully automatic parallelization and locality optimization.

We have recently prototyped an end-to-end practical parallelizer and locality optimizer in the polyhedral model. A model-driven approach to tile for both parallelism and locality directly through an affine transformation framework is the central idea. Our approach is thus a departure from scheduling-based approaches in this field [16, 17, 13, 19] as well as partitioning-based approaches [27, 26, 25] (due to incorporation of more concrete optimization criteria), but is built on the same mathematical foundations and machinery. Our system generates tiled code for statement domains of arbitrary dimensionalities under statement-wise affine transformations for data locality optimization as well as shared memory parallel execution; this has not been previously feasible.

Figure 1 shows the components of our prototype system [8] for automatic parallelization. The scanner, parser and dependence tester from the LooPo infrastructure [28] are used. LooPo is a polyhedral source-to-source transformation system that includes implementations of various polyhedral analyses and transformations from the literature. We used PipLib 1.3.3 [29, 14] as the ILP solver and CLooG 0.14.1 (with 64 bits) for code generation.

We [8] have done several initial experiments from the existing system. Significant improvements are obtained over production compilers as well as the state-of-the-art from the research community.

Table 1. Initial results from Automatic Parallelization system: improvements on Intel quad-core (details in [8])

Benchmark	Single core improvement:		Quad-core improvement:	
	`icc`	research compiler	`icc`	research compiler
1-d Jacobi	5.2x	2.1x	20.0x	1.7x
2-d FDTD	3.7x	3.1x	7.4x	2.5x
LU	5.6x	5.7x	14.0x	3.7x
MVT	9.3x	5.5x	13.0x	7.0x

3 GPGPUs and Accelerators

Several parallel architectures such as GPUs and the Cell processor have fast explicitly managed on-chip memories, in addition to slow off-chip memory. They also have very high computational power with multiple levels of parallelism. A significant challenge in programming these architectures is to effectively exploit the parallelism available in the architecture and manage the fast memories to maximize performance.

In recent work, we have developed an approach to effective automatic data management for on-chip memories, including creation of buffers in on-chip (local) memories for holding portions of data accessed in a computational block, automatic determination of array access functions of local buffer references, and generation of code that moves data between slow off-chip memory and fast local memories. We also addressed the problem of mapping computation in regular programs to multi-level parallel architectures using a multi-level tiling approach, and studied the impact of on-chip memory availability on the selection of tile sizes at various levels. Below, we provide results (on an nVIDIA GeForce 8800 GTX GPU) of an experimental study on two kernels, Mpeg4 Motion Estimation (ME) and 1-D Jacobi. For efficient execution of the kernels, multi-level tiling was performed scratchpad memory was managed using our automatic data management framework. Details were reported in a recent paper [4].

Fig. 2 and Fig. 3 illustrate the benefits of efficient data access using scratchpad memory and also exemplify the high speedup achieved in running the kernel in GPU in contrast to the CPU (Intel Core2 Duo processor clocked at 2.13 GHz). The speedup of the implementation utilizing scratchpad memory was 8x for Mpeg4 ME kernel and 10x for Jacobi kernel over that using only GPU DRAM. The speedup over CPU performance was over 100x for the Mpeg4 ME kernel and 15x for the Jacobi kernel.

For various problem sizes, we conducted experiments to analyze the performance of the Mpeg4 ME kernel. The results are shown in Fig. 4. The number of thread blocks was chosen as 32 and the number of threads as 256. The tile sizes were determined by automatically using a model that minimized data movement subject to the scratchpad mem-

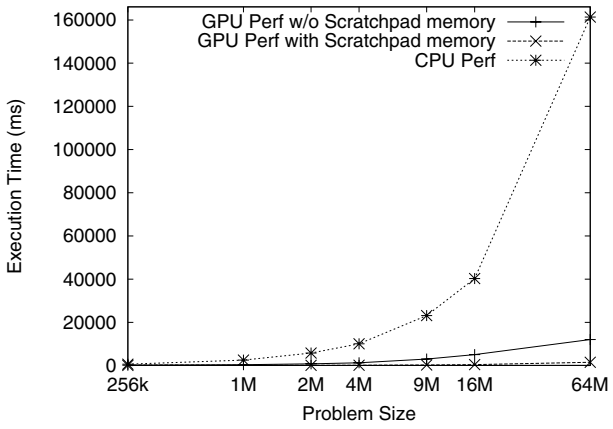

Figure 2. Execution time: Mpeg4 ME

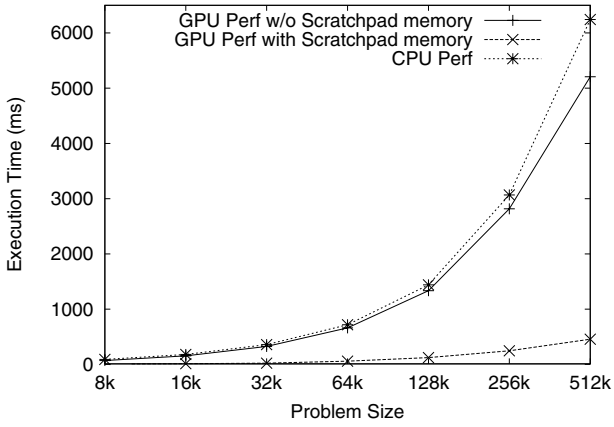

Figure 3. Execution time: 1-D Jacobi

ory constraint.

The tile sizes chosen by the algorithm gave better performance than other tile sizes for various problem sizes, as shown in Fig. 4.

4 Concurrency in Tiled Execution

With few exceptions (e.g. work of Griebl [19]), research on performance optimization with tiling [22, 39, 37, 12, 36, 33, 35, 9, 21, 2] has generally focused on one or the other of the two complementary aspects: (a) data locality optimization [1, 2, 39, 37, 12]; or (b) tile size/shape optimization for parallel execution [36, 33, 9, 21]. Tiling for data locality optimization involves maximization of data reuse, i.e., tiling along directions of the data dependence vectors. But such tiling may result in inter-tile dependences that inhibit concurrent execution of tiles on different processors. We address in an integrated fashion, the issues of tiling for data

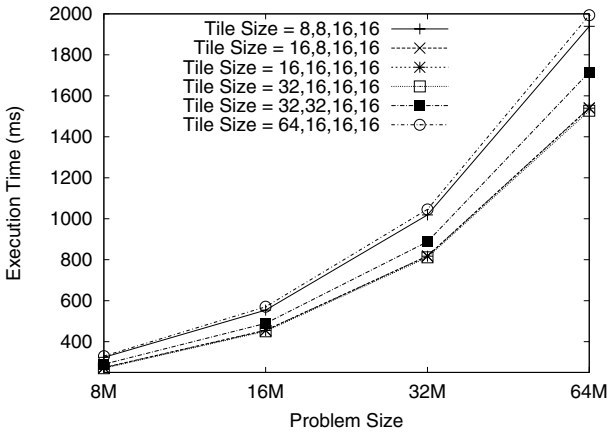

Figure 4. Execution time: Mpeg4 ME for varying tile sizes

locality optimization and load balancing for parallel execution.

Consider the one-dimensional Jacobi code shown in Figure 5. The expanded array version is shown for simplicity of illustration. Tiling for data reuse optimization (e.g. using the approach presented in [1]) results in tiles of shape as shown in Figure 6(a). The horizontal axis corresponds to the spatial dimension, with time along the vertical dimension. Using a sufficiently large tile size along the time dimension facilitates significant data reuse within caches/registers. However, there are inter-tile dependences in the horizontal direction, inhibiting concurrent execution of tiles by different processors. However, if the vertical tile size is reduced to one (i.e., tiling is eliminated along the time dimension), all tiles along the spatial dimension (adjoining the x-axis) can be executed concurrently. Thus there is a trade-off between achieving good data reuse and load balancing of parallel execution.

Instead of the *standard* tiling described above, consider the tiling shown in Figure 6(b). Starting with the tiles formed by the same hyperplanes, an additional triangular region is added to the left of the tile, overlapping with the points at the right end of the neighboring tile. With this tiling, some of the iterations are executed redundantly by two neighboring tiles. While increasing the computation cost, this eliminates the dependence between tiles along the horizontal direction. All processors can start executing in parallel, eliminating the initial processor idling that results with the pipelined parallel execution of tiles in Figure 6(a).

While standard tiling can enhance data locality in this context, overlapped tiling can both improve data locality and eliminate the overhead of pipelined parallelism, at the cost of slightly increased computation time. However, the increased computational cost is independent of tile size. Therefore the fractional computation overhead is inversely proportional to the tile size in the direction of overlapped tiling, and can be made insignificant if a sufficiently large

```
for t = 0 to T-1
 for i = 1 to N-1
    A[t,i] = (A[t-1,i-1] + A[t-1,i] + A[t-1,i+1])/3;
```

Figure 5. Single-statement form of one-dimensional Jacobi code

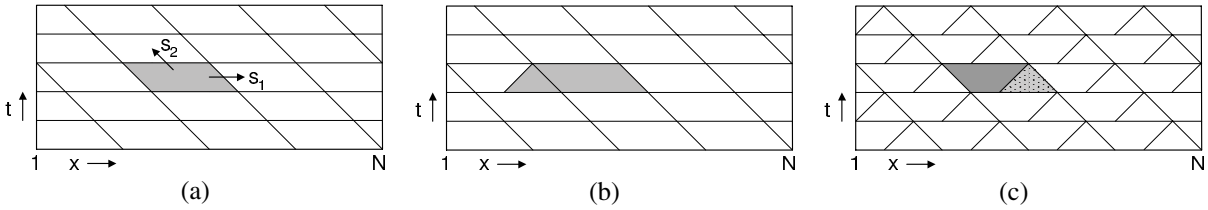

Figure 6. One-dimensional Jacobi code. s_1 and s_2 denote the inter-tile dependences. (a) Standard-tiling (b) Overlapped tiling (c) Split tiling

tile size is chosen along the time dimension.

An alternative approach, shown in Figure 6(c), splits the interior of each tile into two sub-tiles, where the points in only one of the two sub-tiles (shaded) are dependent on points in the neighbor tile, while the points in the other sub-tile are not dependent on any neighboring tile's points, and therefore executable concurrently. With this approach, each standard tile is split into two sub-tiles, and load-balanced concurrent execution is possible as a sequence of two steps: first all non-dependent sub-tiles are concurrently executed and communicate with the neighbor tiles, and then the dependent sub-tiles are all concurrently executed.

Both approaches are novel parallelization schemes exposing greater parallelism in the program. We have demonstrated a unified formulation to derive overlapped and split-tiled versions of tiled loop programs in which concurrent start is inhibited in the tiled space. Both were demonstrated to have improved performance over pipelined execution.

5 Related work

Iteration space tiling [22, 39, 33] is a standard approach for aggregating a set of loop iterations into tiles, with each tile being executed atomically. It is well known that it can improve register reuse, locality and optimize communication. Researchers have considered the problem of selecting tile shape and size to minimize communication, improve locality or minimize finish time [33, 9, 36]. But these studies were restricted to perfectly nested loops with uniform dependences or had other restrictions that limited their applicability to very simple codes. Some specialized works [37, 40] also exist on tiling a restricted class of imperfectly nested loops.

Loop parallelization has been studied extensively. The reader is referred to the survey of Boulet et al.[9] for a detailed summary of older parallelization algorithms which accepted restricted input and/or were based on weaker dependence abstractions than exact polyhedral dependences. Overall, automatic parallelization efforts in the polyhedral model broadly fall in two classes: (1) scheduling/allocation-based, and (2) partitioning-based. The works of Feautrier [16, 17], Darte and Vivien [13], and Griebl [19] (to some extent) fall into the former class, while Lim/Lam's approach falls into the second class. Our approach is closer to the latter class of partitioning-based approaches. In addition to model-based approaches, semi-automatic and search-based transformation frameworks in the polyhedral model also exist [11, 18, 30].

Code generation under multiple affine mappings was first addressed by Kelly et al. [23]. Significant advances were made by Quilleré et al. [32] and more recently by Bastoul et al. and Vasilache et al. [5, 38], resulting in a powerful open-source code generator, CLooG [10]. Our tiled code generation scheme uses Ancourt and Irigoin's classic approach [3] to specify domains with fixed tile sizes and shape information, but combines it with CLooG's support for scattering functions to allow generation of tiled code for multiple domains under transformations obtained from our theoretical framework. Techniques for parametric tiled code generation [34, 24] were recently proposed for single statement domains for which rectangular tiling is valid. Such techniques complement our parallelization framework and we plan to integrate them into our system.

6 Conclusions

We have summarized our recent progress on automatic parallelization for multicore processors, including general-purpose systems as well as accelerators. The polyhedral model for transformation provides a powerful basis for the system, and recent advances have made it feasible to use with non-toy codes. We plan to work with application developers and further advance the infrastructure with the goal of making effective automatic parallelization of real programs feasible in the future.

Acknowledgments We would like to acknowledge Cédric Bastoul and other contributors to the CLooG code

generator and Martin Griebl and team for the LooPo infrastructure. This work was supported in part by NSF through grants 0121676, 0121706, 0403342, 0508245, 0509442, 0509467, and 0541409.

References

[1] N. Ahmed, N. Mateev, and K. Pingali. Synthesizing transformations for locality enhancement of imperfectly nested loops. In *Proc. ACM Intl. Conf. on Supercomputing*, pages 141–152, 2000.

[2] N. Ahmed, N. Mateev, and K. Pingali. Tiling imperfectly-nested loop nests. In *Proc. Supercomputing (SC 2000)*, 2000.

[3] C. Ancourt and F. Irigoin. Scanning polyhedra with do loops. In *PPoPP'91*, pages 39–50, 1991.

[4] M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories. In *PPoPP'08: 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Feb. 2008.

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

[6] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Affine transformations for communication minimal parallelization and locality optimization for arbitrarily-nested loop sequences. Technical Report OSU-CISRC-5/07-TR43, The Ohio State University, May 2007.

[7] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Affine transformations for communication minimal parallelization and locality optimization for arbitrarily-nested loop sequences. In *CC 2008: International Conference on Compiler Construction*, Apr. 2008.

[8] U. Bondhugula, J. Ramanujam, and P. Sadayappan. PLuTo: A practical and fully automatic polyhedral parallelizer and locality optimizer. Technical Report OSU-CISRC-5/07-TR70, The Ohio State University, Oct. 2007.

[9] P. Boulet, A. Darte, T. Risset, and Y. Robert. (Pen)-ultimate tiling? *Integration, the VLSI Journal*, 17(1):33–51, 1994.

[10] CLooG: The Chunky Loop Generator. http://www.cloog.org.

[11] A. Cohen, S. Girbal, D. Parello, M. Sigler, O. Temam, and N. Vasilache. Facilitating the search for compositions of program transformations. In *ACM ICS*, pages 151–160, June 2005.

[12] S. Coleman and K. S. McKinley. Tile size selection using cache organization and data layout. In *Proceedings of PLDI '95*, pages 279–290, 1995.

[13] A. Darte and F. Vivien. Optimal fine and medium grain parallelism detection in polyhedral reduced dependence graphs. *Int. J. Parallel Programming*, 25(6):447–496, Dec. 1997.

[14] P. Feautrier. Parametric integer programming. *Operationnelle/Operations Research*, 22(3):243–268, 1988.

[15] P. Feautrier. Dataflow analysis of array and scalar references. *IJPP*, 20(1):23–53, 1991.

[16] P. Feautrier. Some efficient solutions to the affine scheduling problem: I. one-dimensional time. *IJPP*, 21(5):313–348, 1992.

[17] P. Feautrier. Some efficient solutions to the affine scheduling problem. part II. multidimensional time. *IJPP*, 21(6):389–420, 1992.

[18] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic composition of loop transformations. *IJPP*, 34(3):261–317, June 2006.

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

[20] M. Griebl, C. Lengauer, and S. Wetzel. Code generation in the polytope model. In *IEEE PACT*, pages 106–111, 1998.

[21] K. Hogstedt, L. Carter, and J. Ferrante. Selecting tile shape for minimal execution time. In *Proceedings of SPAA '99*, pages 201–211, 1999.

[22] F. Irigoin and R. Triolet. Supernode partitioning. In *PoPL*, pages 319–329, 1988.

[23] W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. In *FRONTIERS*, page 332, 1995.

[24] D. Kim, L. Renganarayanan, M. Strout, and S. Rajopadhye. Multi-level tiling: 'm' for the price of one. In *SC*, 2007.

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

[26] A. W. Lim, G. I. Cheong, and M. S. Lam. An affine partitioning algorithm to maximize parallelism and minimize communication. In *ACM ICS*, pages 228–237, 1999.

[27] A. W. Lim and M. S. Lam. Maximizing parallelism and minimizing synchronization with affine partitions. *Parallel Computing*, 24(3-4):445–475, 1998. Extended version of PoPL'97 paper.

[28] The LooPo Project - Loop parallelization in the polytope model. http://www.fmi.uni-passau.de/loopo.

[29] PIP: The Parametric Integer Programming Library. http://www.piplib.org.

[30] L.-N. Pouchet, C. Bastoul, A. Cohen, and N. Vasilache. Iterative optimization in the polyhedral model: Part I, one-dimensional time. In *ACM CGO*, Mar. 2007.

[31] W. Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8:102–114, Aug. 1992.

[32] F. Quilleré, S. V. Rajopadhye, and D. Wilde. Generation of efficient nested loops from polyhedra. *IJPP*, 28(5):469–498, 2000.

[33] J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for multicomputers. *JPDC*, 16(2):108–230, 1992.

[34] L. Renganarayana, D. Kim, S. Rajopadhye, and M. M. Strout. Parameterized tiled loops for free. In *PLDI'07*, pages 405–414, 2007.

[35] L. Renganarayana and S. Rajopadhye. A geometric programming framework for optimal multi-level tiling. In *Proceedings of SC '04*, page 18, 2004.

[36] R. Schreiber and J. Dongarra. Automatic blocking of nested loops. Technical report, University of Tennessee, Knoxville, TN, Aug. 1990.

[37] Y. Song and Z. Li. New tiling techniques to improve cache temporal locality. In *Proceedings of PLDI '99*, pages 215–228, 1999.

[38] N. Vasilache, C. Bastoul, and A. Cohen. Polyhedral code generation in the real world. In *CC 2006: International Conference on Compiler Construction*, pages 185–201, Mar. 2006.

[39] M. Wolf and M. S. Lam. A data locality optimizing algorithm. In *PLDI '91*, pages 30–44, 1991.

[40] Q. Yi, K. Kennedy, and V. Adve. Transforming complex loop nests for locality. *J. Supercomput.*, 27(3):219–264, 2004.