PhD - References

Unai Sainz de la Maza Gamboa

June 22, 2024

Contents

1 Compiler-based distributed computing techniques (thesis title?)

1.1 Summary

Compute-intensive applications running on clusters of shared-memory computers typically utilize OpenMP and MPI for implementation. These applications are challenging to program, debug, and maintain. Additionally, achieving performance portability is often limited, requiring several program transformations at multiple levels of the software and hardware stack to leverage features like parallelism and granularity. Finally, specific characteristics of the target system, such as network topology and node capacity, must be considered to achieve high-performing code.

Performance portability and user productivity are major concerns for modern compilers, as significant code reorganization and restructuring are often required to adapt applications to the resources of a computer cluster. Consequently, polyhedral auto-transformation frameworks have garnered significant interest in general-purpose compilation due to their capability to identify and implement complex loop transformations that extract high performance from modern architectures. These frameworks automatically identify loop transformations that enhance locality, parallelism, minimize latency, or achieve a combination of these improvements.

In this thesis, we try to improve the applicability and profitability of polyhedral-model-based techniques for distributed computing challenges, such as scheduling and communication optimization. Specifically, we enrich the polyhedral program representation with domain-specific knowledge from distributed computing systems, e.g., network topology or nodes capacity.

1.2 Background

The polyhedral model is a mathematical representation of programs that simplifies both analysis and restructuring. It provides a compact and expressive way to describe parallelization and optimization problems. Unlike operational or syntactic representations, this model is closer to program execution because it operates on individual statement iterations or instances. For each instance, the optimizing algorithm computes a mapping to determine the execution time (time mapping or scheduling) and/or the processor on which it will run (space mapping or placement) [**?**, **?**].

The pioneering work of the polyhedral model is attributed to Karp et al. on systems of uniform recurrence equations [**?**]. With the advancement of polyhedral compilation, integer polyhedra [**?**] and Presburger relations [**?**, **?**] were introduced to enhance expressiveness and flexibility. Polyhedral compilation can either be integrated as a building block into general-purpose compilers, such as Polly in LLVM [**?**] and affine in MLIR [**?**], or serve as

a source-to-source translator like Pluto [?, ?] and PPCG [?]. This flexibility and compatibility enable the construction of a well-defined compilation workflow and significantly extend its application domain.

Polyhedral compilation typically comprises three steps: modeling, transformation, and code generation. In the modeling step, a mathematical representation of the program is constructed using integer polyhedra and Presburger relations to capture the program's computational patterns. The transformation step then applies various optimizations, such as loop transformations, to enhance parallelism and data locality. Finally, the code generation step translates the optimized polyhedral model back into executable code.

The polyhedral model has proven useful for transforming and generating parallel programs from sequential codes featuring affine loop nests [?]. Additionally, it facilitates automatic code generation for distributed-memory platforms [?]. The model's dependence analysis supports code generation by (1) identifying values requiring communication across processes, (2) managing data packing and unpacking, and (3) executing necessary communication operations. Griebl et al. presents the use of the polyhedral model on distributed-memory systems [?], but their approach suffers from significant redundancy in communication operations. Existing methods for calculating communication in distributed memory, such as those discussed in [?, ?, ?, ?, ?], focus on sequences of nested loops with regular (affine) accesses, known as affine loop nests. These methods involve loop transformation, tiling, and parallelization. However, they often overlook characteristics specific to the target distributed platform, such as heterogeneity of nodes, network topology, or node capacities.

More?

1.3 Objectives

ok? Plantilla? Based on the issues outlined in the previous section, the research question to be addressed in this Ph.D. thesis is: ...

- Is it possible to improve the application of polyhedral compilation to distributed systems by means of specific information such as topology?
- Can we improve the techniques used in distributed systems with the help of compilers?

1.4 Methodology

ok? Plantilla? The research methodology employed in this thesis is based on an experimental approach and a software engineering method encompassing four stages: observing existing solutions, proposing improved solutions,

developing the proposed solutions, and measuring and analyzing the outcomes [**?**]. This iterative methodology is repeated to refine the solutions. It parallels the stages of the classical scientific method: proposing a question, formulating a hypothesis, making predictions, and validating the hypothesis.

1. Observe existing solutions: this is an exploratory phase where the literature and state-of-the-art tools related to our research field are studied to identify potential improvements.

2. Propose better solutions: this phase focuses on designing and analyzing improved solutions, aiming to surpass the limitations of previous proposals or enhance existing methods from the literature.

3. Build or develop the solutions: in this phase, we concentrate on constructing a prototype to demonstrate the feasibility of the solution.

4. Measure and analyze the new solutions: finally, the implemented prototypes are empirically evaluated and compared with various alternatives. The goal of this evaluation is to validate the proposed solution and confirm that the problems identified in the initial step have been resolved.

1.5 Work plan

Plantilla?

1.5.1 Part I: Topology-aware distributed data communication optimization?

1.5.2 Part II: ...

1.6 Scientific or social interest

Requisitos: Memoria científico–técnica en formato normalizado del proyecto de tesis doctoral firmado por la persona directora del mismo, y codirectora si la hubiere, y cuyo contenido será el siguiente: título del proyecto, resumen, antecedentes, objetivos, metodología y plan de trabajo, interés científico y/o social y la bibliografía más destacada (máximo 5 páginas). Además de este contenido, dicha memoria deberá recoger expresamente lo siguiente: Si es o no esperable que los objetivos del proyecto de Tesis Doctoral y su defensa se alcancen por la persona solicitante en 3 años de ayuda con dedicación completa.

2 Polyhedral papers

2.1 Topics (taken from IMPACT CFP)

- Program optimization (automatic parallelization, tiling, etc.)
- Code generation.
- Data/communication management on GPUs, accelerators and distributed systems.
- Hardware/high-level synthesis for affine programs.
- Static analysis.
- Program verification.
- Model checking.
- Theoretical foundations of the polyhedral model.
- Extensions of the polyhedral model.
- Scalability and robustness of polyhedral compilation techniques.

2.2 Courses

- Polyhedral seminar (C. Alias): Link
- UCLA CS33/CS31 (LN. Pouchet): Link
- Presburger Formulas and Polyhedral Compilation Tutorial (S. Verdoolaege): Link
- Static Analysis and Optimizing Compilers (C. Alias): Link
- Polyhedral Compilation as a Design Pattern for Compilers (A.Cohen): Link 1 and Link 2

2.3 The Polyhedral Model Is More Widely Applicable Than You Think

[?]

2.4 Modelling linear algebra kernels as polyhedral volume operations

In [?], programs are represented as sequences of operations on indexed volumes characterized by their element-wise dependencies, efficiently implementing compound kernels by partitioning and specializing over index domains.

- They generalize over arrays and define a volume as an aggregate value comprising indexed element values (scalars). Volume is characterized by an index domain, which is a polyhedron of all element indices.
- Volumes permit arbitrary polyhedral index domains and are intended to model the structural properties of data, as well as memory when needed.
- They perform dynamic shape inference of input/output/intermediate volumes. They use affine pattern matching to identify regions of interest (sparse regions in a volume) and specialize over them. They separate computation from memory (operational semantics based on general affine volumes, not strictly hyperrectangular ones).
- The proposed model can be used on the MLIR linalg abstraction to generalize tiling and other partitioning transformations to arbitrary tensor programs.

The main idea is to detect sparsity and other structural properties, and act on them by partitioning computations. State-of-the-art compilers struggle to separate the different regions of the program, which may lead to suboptimal performance. However, their model can conveniently identify all kernel regions, i.e., dense, sparse, constant, and corners. This empowers specializing over these regions, e.g., by using sparse compiler optimizations for sparse regions or offloading them to a sparsity-aware hardware target and mapping dense regions to external library calls.

2.5 ParameTrick: Coefficient Generalization for Faster Polyhedral Scheduling

In [?],

2.6 A practical tile size selection model for affine loop nests

2.7 Maximal Atomic irRedundant Sets: a Usage-based Dataflow Partitioning Algorithm

2.8 A polyhedral compilation framework for loops with dynamic data-dependent bounds

2.9 Generating SIMD Instructions for Cerebras CS-1 using Polyhedral Compilation Techniques

In [**?**], they use polyhedral compilation to generate SIMD instructions for the cerebras cs-1 computing system. More precisely, they use polyhedral operations such as variable compression and fixed-sized box hull.

- SIMD engine that can mimic a rectangular loop nest of depth at most four. For this, is important to represent the set of computation instances as a rectangular domain.
- They rely on *isl* library for manipulating sets of integer tuples described by Presburger formulas (a first order logic formula involving affine expressions, equality and the less-than-or-equal relation).
- Variable compression exploits the equality constraints in the description of a set to obtain a set with the same number of points, but of a lower dimensionality.
- Fixed-size box hull operation examines the constraints of a binary relation to find an overapproximation where the range can be described as a rectangular box with fixed size and an offset that depends on the domain variables and the symbolic constants.
- Target architecture: MPPA (Massively Parallel Processor Array), consisting of a 2-dimensional grid of PEs (processing elements) that communicate with their nearest neighbors in the four cardinal directions. Memory distributed overt the PEs and hardware performs dataflow scheduling. (not much public details).

The main idea is to expose rectangle sets of operations that can be mapped to the advanced SIMD operations of the Cerebras CS-1 architecture.

2.10 Polygeist: Affine C in MLIR

2.11 Automatic multi-dimensional pipelining for high-level synthesis of dataflow accelerators

2.12 Tile size selection of affine programs for GPGPUs using polyhedral cross-compilation

In [?], they use Integer Linear Programming (ILP) constraints and objectives in a cross-compiler fashion to faithfully and effectively mimic the transformations applied in a polyhedral GPU compiler (PPCG) to compute resource-conscious tile sizes for affine programs.

2.13 PolySA: Polyhedral-Based Systolic Array Auto-Compilation

In [?], they present PolySA, the first fully automated compilation framework for generating high performance systolic array architectures on the FPGA that leverages recent advances in high-level synthesis and can generate optimal designs in one hour with performance comparable to state-of-the-art manual designs.

2.14 AutoSA: A Polyhedral Compiler for High-Performance Systolic Arrays on FPGA

In [?], they present AutoSA, an end-to-end compilation framework for generating systolic arrays on FPGA, based on the polyhedral framework, and further incorporates a set of optimizations on different dimensions to boost performance.

2.15 Scalable Polyhedral Compilation, Syntax vs. Semantics: 1–0 in the First Round

In [?], a family of techniques is introduced called offline statement clustering, which integrates transparently into the flow of a state-of-the-art polyhedral compiler and can reduce the scheduling time by a factor of 6 without inducing a significant loss in optimization opportunities.

2.16 Polyhedral Compilation for Racetrack Memories

In [?], they present the first automatic compilation framework that optimizes static loop programs over arrays for linear-latency memories, extending the polyhedral compilation framework Polly to generate code that maximizes accesses to the same or consecutive locations, thereby minimizing the number of shifts.

2.17 Polyhedral Compilation for Multi-dimensional Stream Processing

In [?], they present a method that involves a novel polyhedral schedule transformation called periodic tiling that enables efficient execution of programming languages with unbounded recurrence equations, as well as optimization of existing languages from which this form can be derived.

2.18 An Autotuning Framework for Scalable Execution of Tiled Code via Iterative Polyhedral Compilation

2.19 Employing polyhedral methods to optimize stencils on FPGAs with stencil-specific caches, data reuse, and wide data bursts

In [?], polyhedral methods are used to generate stencil-specific cache-structures of the right sizes on the FPGA and to fill and flush them efficiently with wide bursts during stencil execution. They derive the appropriate directives and code restructurings from stencil codes so that the FPGA compiler generates fast stencil hardware.

- From previous attempts: HLS cannot turn the tiled loop into hardware pipelines (as the loop boundaries and increments are too complex for the HLS to detect patterns). On CPUs or GPUs tiled loops run faster since their data locality exploits the built-in cache hierarchy, however, on FPGAs there are no caches, there is no benefit from tiling alone. In general, the HLS generates from the tiled loop a hardware block (kernel) with connections to an FPGA bus through which the kernel directly talks to the DDR.
- Polyhedral methods are used to understand the access patterns we can statically determine (a) the stencil-specific best cache sizes and (b) preload exactly what is needed and spill what is no longer needed. But the polyhedral method can also (c) help improve the burst behavior. They also (d) inline the cache functionality, i.e., use buffer arrays and load operations within the stencil.

This work uses loop tiling to improve locality, adds buffers and directives to generate inlined cache-like hardware circuits that exploit that locality, and calculates a data-shipment that uses wide bursts to fill these caches, i.e., a polyhedral-based code generator for FPGAs.

2.20 Modelling linear algebra kernels as polyhedral volume operations

2.21 Automated Partitioning of Data-Parallel Kernels using Polyhedral Compilation

2.22 Superloop Scheduling: Loop Optimization via Direct Statement Instance Reordering

In [**?**], they propose a different approach to affine scheduling construction called superloop scheduling. A 7-step loop optimization process:

- Iteration space bounding: limit the number of iterations and replaces the parameters with known values.
- Full loop unrolling: transform the code to a sequence of statement instances.
- Statement instance reordering: transform the unrolled code via direct reasoning and manipulation of the statement instances. Parallel block extraction and internal reordering to enable vectorization. Also, basic block level techniques such as superword level parallelisation [**?**] to enable possibly unprecedented loop vectorization opportunities. Reordering policies should include constraints or mechanisms to favor some regularity when possible.
- Nested loop recognition (NLR): recover loops in a fast and incremental way [**?**] from a trace comprised of tagged vectors of numbers. NLR is able to recover arbitrarily deep and/or complex affine loop nests from their traces.
- Affine scheduling reconstruction: build an affine scheduling expression from the loop structure and the mapping information present in NLR's output. Also, attempt to recover parameters, e.g. through pattern-matching.
- Legality check: verify the scheduling correctness on the original input code using the Candl tool, and may assess its properties such as parallel and vector dimensions.
- Code generation: produce the optimized code that implements the scheduling using, e.g., CLooG [**?**]. The optimization is different than both Feautrier [**?**] (which favors k, i, j version with internal parallelism) and Pluto without tiling [**?**] (which splits S0 and S1 in two loop nests to enable i, k, j order for S1, or uses less CPU-efficient i, j, k).

Two families of techniques for affine scheduling construction:

- Compute the scheduling by solving systems of affine constraints: proposed by Feautrier [?] and has set the ground for many later techniques, the Pluto algorithm designed by Bondhugula et al. to extract outermost parallelism, data locality and tilable loops [?], and a number of variants that differ in the way the affine constraint system is built.
- Build the affine scheduling by composition of basic primitives: proposed by Kelly and Pugh [?] and improved by many authors, e.g., Baghdadi et al. who propose a rich scheduling language for the Tiramisu framework [?].

However, in this work, they present the seed for a new family that builds affine scheduling from finest-grain statement instance reordering and loop reconstruction. Which could have a significant potential for exploiting custom vector instructions.

Important note: presented at IMPACT 2023, a short (working) paper.

2.23 Pipelined Multithreading Generation in a Polyhedral Compiler

2.24 Compiling affine loop nests for distributed-memory parallel architectures

[?]

2.25 Dynamic memory access monitoring based on tagged memory

[?]

2.26 Effective automatic computation placement and data allocation for parallelization of regular programs

[?]

References

[1] Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and Nicolas Vasilache. Iterative optimization in the polyhedral model: Part i, one-dimensional time. In *Fifth International Symposium on Code Generation and Optimization (CGO 2007), 11-14 March 2007, San Jose, California, USA*, pages 144–156. IEEE Computer Society, 2007.

[2] Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and John Cavazos. Iterative optimization in the polyhedral model: part ii, multidimensional time. In Rajiv Gupta and Saman P. Amarasinghe, editors, *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 90–100. ACM, 2008.

[3] Richard M. Karp, Raymond E. Miller, and Shmuel Winograd. The organization of computations for uniform recurrence equations. *J. ACM*, 14(3):563–590, 1967.

[4] Vincent Loechner and Doran K. Wilde. Parameterized polyhedra and their vertices. *Int. J. Parallel Program.*, 25(6):525–549, 1997.

[5] William W. Pugh and David Wonnacott. An exact method for analysis of value-based array data dependences. In Utpal Banerjee, David Gelernter, Alexandru Nicolau, and David A. Padua, editors, *Languages and Compilers for Parallel Computing, 6th International Workshop, Portland, Oregon, USA, August 12-14, 1993, Proceedings*, volume 768 of *Lecture Notes in Computer Science*, pages 546–566. Springer, 1993.

[6] William W. Pugh and David Wonnacott. Static analysis of upper and lower bounds on dependences and parallelism. *ACM Trans. Program. Lang. Syst.*, 16(4):1248–1278, 1994.

[7] Tobias Grosser, Armin Größlinger, and Christian Lengauer. Polly - performing polyhedral optimizations on a low-level intermediate representation. *Parallel Process. Lett.*, 22(4), 2012.

[8] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: A Compiler Infrastructure for the End of Moore's Law, February 2020. arXiv:2002.11054 [cs].

[9] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *International Conference on Compiler Construction (ETAPS CC)*, April 2008.

[10] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral program optimization system. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2008.

[11] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for cuda. *ACM Trans. Archit. Code Optim.*, 9(4):54:1–54:23, January 2013.

[12] Cédric Bastoul. Code generation in the polyhedral model is easier than you think. In *13th International Conference on Parallel Architectures and Compilation Techniques (PACT 2004), 29 September - 3 October 2004, Antibes Juan-les-Pins, France*, pages 7–16. IEEE Computer Society, 2004.

[13] Saman P. Amarasinghe and Monica S. Lam. Communication optimization and code generation for distributed memory machines. *SIGPLAN Not.*, 28(6):126–138, jun 1993.

[14] Michael Classen and Martin Griebl. Automatic code generation for distributed memory architectures in the polytope model. In *20th International Parallel and Distributed Processing Symposium (IPDPS 2006), Proceedings, 25-29 April 2006, Rhodes Island, Greece*. IEEE, 2006.

[15] Ganesh Bikshandi, Jia Guo, Daniel Hoeflinger, Gheorghe Almási, Basilio B. Fraguela, María Jesús Garzarán, David A. Padua, and Christoph von Praun. Programming for parallelism and locality with hierarchically tiled arrays. In Josep Torrellas and Siddhartha Chatterjee, editors, *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2006, New York, New York, USA, March 29-31, 2006*, pages 48–57. ACM, 2006.

[16] Roshan Dathathri, Ravi Teja Mullapudi, and Uday Bondhugula. Compiling affine loop nests for a dynamic scheduling runtime on shared and distributed memory. *ACM Trans. Parallel Comput.*, 3(2):12:1–12:28, 2016.

[17] Uday Bondhugula. Compiling affine loop nests for distributed-memory parallel architectures. In William Gropp and Satoshi Matsuoka, editors, *International Conference for High Performance Computing, Networking, Storage and Analysis, SC'13, Denver, CO, USA - November 17 - 21, 2013*, pages 33:1–33:12. ACM, 2013.

[18] Chandan Reddy and Uday Bondhugula. Effective automatic computation placement and data allocation for parallelization of regular programs. In Arndt Bode, Michael Gerndt, Per Stenström, Lawrence

Rauchwerger, Barton P. Miller, and Martin Schulz, editors, *2014 International Conference on Supercomputing, ICS'14, Muenchen, Germany, June 10-13, 2014*, pages 13–22. ACM, 2014.

[19] Tomofumi Yuki and Sanjay Rajopadhye. Parametrically tiled distributed memory parallelization of polyhedral programs. *Colorado State University, Tech. Rep. CS13-105*, 2013.

[20] W Richards Adrion. Research methodology in software engineering. In *Summary of the Dagstuhl workshop on future directions in software engineering" Ed. Tichy, Habermann, and Prechelt, ACM software engineering notes, SIGSoft*, volume 18, pages 36–37, 1993.

[21] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. The polyhedral model is more widely applicable than you think. In *International Conference on Compiler Construction*, 2010.

[22] Karl FA Friebel, Asif Ali Khan, Lorenzo Chelini, and Jeronimo Castrillon. Modelling linear algebra kernels as polyhedral volume operations, 2022.

[23] Gianpietro Consolaro, Harenome Razanajato, Nelson Lossing, Denis Barthou, Zhen Zhang, Corinne Ancourt, and Cédric Bastoul. Parametrick: Coefficient generalization for faster polyhedral scheduling. In *14th International Workshop on Polyhedral Compilation Techniques (IMPACT 2024, in conjunction with HiPEAC 2024)*, 2024.

[24] Sven Verdoolaege, Manjunath Kudlur, Rob Schreiber, and Harinath Kamepalli. Generating simd instructions for cerebras cs-1 using polyhedral compilation techniques. In *IMPACT 2020-10th International Workshop on Polyhedral Compilation Techniques*, 2020.

[25] Kh. K. Abdelaal and Martin Kong. Tile size selection of affine programs for gpgpus using polyhedral cross-compilation. *Proceedings of the ACM International Conference on Supercomputing*, 2021.

[26] Jason Cong and Jie Wang. Polysa: Polyhedral-based systolic array auto-compilation. *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, 2018.

[27] Jie Wang, Licheng Guo, and Jason Cong. Autosa: A polyhedral compiler for high-performance systolic arrays on fpga. *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2021.

[28] Riyadh Baghdadi and Albert Cohen. Scalable polyhedral compilation, syntax vs. semantics: 1–0 in the first round. In *IMPACT 2020 workshop (associated with HIPEAC 2020)*, 2020. Informal proceedings.

[29] Asif Ali Khan, Hauke Mewes, Tobias Grosser, Torsten Hoefler, and Jeronimo Castrillon. Polyhedral compilation for racetrack memories. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):3968–3980, 2020.

[30] Jakob Leben and George Tzanetakis. Polyhedral compilation for multi-dimensional stream processing. *ACM Transactions on Architecture and Code Optimization (TACO)*, 16(3):1–26, 2019.

[31] Florian Mayer, Julian Brandner, and Michael Philippsen. Employing polyhedral methods to optimize stencils on fpgas with stencil-specific caches, data reuse, and wide data bursts. In *14th International Workshop on Polyhedral Compilation Techniques (IMPACT 2024, in conjunction with HiPEAC 2024)*, 2024.

[32] Cedric Bastoul, Alain Ketterlin, and Vincent Loechner. Superloop scheduling: Loop optimization via direct statement instance reordering. In *IMPACT 2023*, 2023.

[33] Charith Mendis and Saman Amarasinghe. goslp: globally optimized superword level parallelism framework. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–28, October 2018.

[34] Alain Ketterlin and Philippe Clauss. Prediction and trace compression of data access addresses through nested loop recognition. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, pages 94–103, 2008.

[35] Cédric Bastoul. Code generation in the polyhedral model is easier than you think. In *Proceedings. 13th International Conference on Parallel Architecture and Compilation Techniques, 2004. PACT 2004.*, pages 7–16. IEEE, 2004.

[36] Paul Feautrier. Some efficient solutions to the affine scheduling problem. part ii. multidimensional time. *International journal of parallel programming*, 21:389–420, 1992.

[37] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 101–113, 2008.

[38] Wayne Kelly and William Pugh. A framework for unifying reordering transformations, 1998.

[39] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 193–205. IEEE, 2019.

[40] Uday Bondhugula. Compiling affine loop nests for distributed-memory parallel architectures. In William Gropp and Satoshi Matsuoka, editors, *International Conference for High Performance Computing, Networking, Storage and Analysis, SC'13, Denver, CO, USA - November 17 - 21, 2013*, pages 33:1–33:12. ACM, 2013.

[41] Mikhail A. Gorelov and Lev Mukhanov. Dynamic memory access monitoring based on tagged memory. In Christian Fensch, Michael F. P. O'Boyle, André Seznec, and François Bodin, editors, *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques, Edinburgh, United Kingdom, September 7-11, 2013*, page 409. IEEE Computer Society, 2013.

[42] Chandan Reddy and Uday Bondhugula. Effective automatic computation placement and data allocation for parallelization of regular programs. In *Proceedings of the 28th ACM International Conference on Supercomputing*, ICS '14, page 13–22, New York, NY, USA, 2014. Association for Computing Machinery.