Report from Dagstuhl Seminar 18111

Loop Optimization

Edited by

Sebastian Hack[1], Paul H. J. Kelly[2], and Christian Lengauer[3]

1 **Universität des Saarlandes, DE,** hack@cs.uni-saarland.de
2 **Imperial College London, UK,** p.kelly@imperial.ac.uk
3 **Universität Passau, DE,** christian.lengauer@uni-passau.de

Abstract

This report documents the programme of Dagstuhl Seminar 18111 "Loop Optimization". The seminar brought together experts from three areas: (1) model-based loop optimization, chiefly, in the polyhedron model, (2) rewriting and program transformation, and (3) metaprogramming and symbolic evaluation. Its aim was to review the 20+ years of progress since the Dagstuhl Seminar 9616 "Loop Parallelization" in 1996 and identify the challenges that remain.

Seminar March 11–16, 2018 – https://www.dagstuhl.de/18111

2012 ACM Subject Classification Theory of computation → Semantics and reasoning, Computing methodologies → Symbolic and algebraic manipulation, Computing methodologies → Parallel computing methodologies, Software and its engineering → Software notations and tools, Software and its engineering → Compilers

Keywords and phrases Autotuning, dependence analysis, just-in-time (JIT), loop parallelization, parallel programming, polyhedron model

Digital Object Identifier 10.4230/DagRep.8.3.39

1 Executive Summary

Sebastian Hack
Paul H. J. Kelly
Christian Lengauer

Motivation

Loop optimization is at the heart of effective program optimization – even if the source language is too abstract to contain loop constructs explicitly as, e.g., in a functional style or a domain-specific language. Loops provide a major opportunity to improve the performance of a program because they represent compactly a large volume of accessed data and executed instructions. Because the clock frequency of processors fails to continue to grow (end of Dennard scaling), the only way in which the execution of programs can be accelerated is by increasing their throughput with a compiler: by increasing parallelism and improving data locality. This puts loop optimization in the center of performance optimization.

Context

The quick and easy way to optimize a loop nest, still frequently used in practice, is by restructuring the source program, e.g., by permuting, tiling or skewing the loop nest. Beside

being laborious and error-prone, this approach favors modifications that can be easily recognized and carried out, but which need not be the most suitable choice. A much better approach is to search automatically for optimization options in a mathematical model of the iteration space, in which all options are equally detectable and the quality of each option can be assessed precisely.

Recently, the polyhedral compilation community has produced a set of robust and powerful libraries that contain a variety of algorithms for the manipulation of Presburger sets, including all standard polyhedral compilation techniques. They can be incorporated in a program analysis to make other compiler optimizations more precise and powerful, like optimizers and code generators for domain-specific languages, or aggressive optimizers for high-performance computing.

Polyhedral loop optimization relies on strict constraints on the structure of the loop nest and may incur a computationally complex program analysis, based on integer linear programming. The optimization problems become much simpler when information at load or run time is available, i.e., the optimization is done just-in-time. Also, the search for the best optimization can be supported by other techniques, e.g., auto-tuning, machine learning or genetic algorithms. While these techniques are all fully automatic, engineering of software with robust performance characteristics requires programmers to have some level of explicit control over the data distribution and communication costs. However, manually optimized code is far too complicated to maintain. Thus, a major research area concerns the design of tools that allow developers to guide or direct analysis (e.g., via dependence summaries or domain-specific code generation) and optimization (e.g., via directives, sketches and abstractions for schedules and data partitioning).

Goal

The goal of this seminar was to generate a new synergy in loop optimization research by bringing together representatives of the major different schools of thought in this field. The key unifying idea is to formulate loop optimization as a mathematical problem, by characterizing the optimization space and objectives with respect to a suitable model.

One school is focused on reasoning about scheduling and parallelization using a geometric, "polyhedral", model of iteration spaces which supports powerful tools for measuring parallelism, locality and communication – but which is quite limited in its applicability.

Another major school treats program optimization as program synthesis, for example by equational rewriting, generating a potentially large space of variants which can be pruned with respect to properties like load balance and locality. This approach has flourished in certain application domains, but also suffers from problems with generalization.

A third family of loop optimization approaches tackles program optimization through program generation and symbolic evaluation. Generative approaches, such as explicit staging, support programmers in taking explicit control over implementation details at a high level of abstraction.

The seminar explored the interplay of these various loop optimization techniques and fostered the communication in the wide-ranging research community of model-based loop optimization. Participants represented the various loop optimization approaches but also application domains in high-performance computing.

Conclusions

The seminar succeeded in making the participants aware of common goals and relations between different approaches. Consensus emerged on the potential and importance of tensor contractions and tensor comprehensions as an intermediate representation. There was also some excitement in connecting the classical dependence-based optimization with newly emerging ideas in deriving parallel algorithms from sequentially-dependent code automatically. Guided automatic search and inference turned out to be a dominant theme. Another important insight was that the optimization criteria currently in use are often too coarse-grained and do not deliver satisfactory performance. More precise hardware models are needed to guide optimization. This will require a closer collaboration with the performance modeling and engineering community.

It was agreed that publications and collaborations fueled by the seminar will acknowledge Schloss Dagstuhl.

2 Table of Contents

3 Overview of Talks

30-min talk slots covered the programme until Thursday mid-afternoon; four keynote presentations took up two slots each. The latter part of the seminar was devoted to the planning of future collaborations. A list of talks follows in the order in which they were presented.

3.1 On the Design of Intermediate Representations for Loop Nest Optimization (Keynote)

Albert Cohen (ENS – Paris, FR)

Joint work of Ulysse Beaugnon, Albert Cohen, Tobias Grosser, Jacques Pienaar, Chandan Reddy, Vivek Sarkar, Jun Shirako, Nicolas Vasilache, Sven Verdoolaege, Oleksandr Zinenko, Jie Zhao

The associated slides focus on advanced affine scheduling heuristics and attempt to derive some lessons from the experience of customizing a rather generic framework (polyhedral compilation) to a specific purpose (harnessing the multi-level parallelism and memory hierarchy of modern multi-processors).

3.2 Beyond the Polyhedral Model (Keynote)

Paul Feautrier (ENS – Paris, FR)

The polyhedral model is a powerful tool for program analysis, verification, optimization and parallelization. However, its applicability is restricted to regular programs with only scalar and arrays as the only data structures, affine subscripts, and counted loops with affine bounds as the only control constructs. As it now stands, the model is especially applicable to linear algebra and signal processing algorithms. Many extensions to the model were proposed since its inception, as for instance using enabling transformations or detection of static control parts: SCoPs. I feel that the time has come for a more drastic approach: the creation and exploration of new models, either more powerful than the polyhedral model or directed at other families of algorithms.

I will first review early attempts at the creation of other models, as for instance the polynomial model, the flowchart model, or models based on the theory of formal languages. A promising research direction is the use of approximate methods, applying concepts borrowed from the abstract interpretation theory. The flowchart model is a first attempt at combining both types of tools.

Proof assistants like Coq may help in the construction of models. However, Coq is not a solver, and hence is not suited for the analysis of existing programs (the "dusty deck" problem). It may best be used for the construction of frameworks guaranteeing bug-free programming.

3.3 Static Instruction Scheduling for High Performance on Limited Hardware

Alexandra Jimborean (Uppsala University – Uppsala, SE)

Joint work of Kim-Anh Tran, Trevor E. Carlson, Konstantinos Koukos, Magnus Själander, Vasileios Spiliopoulos, Stefanos Kaxiras, Alexandra Jimborean

Complex out-of-order (OoO) processors have been designed to overcome the restrictions of outstanding long-latency misses at the cost of increased energy consumption. Simple, limited OoO processors are a compromise in terms of energy consumption and performance, as they have fewer hardware resources to tolerate the penalties of long-latency loads. In the worst case, these loads may stall the processor entirely.

We present Clairvoyance, a compiler-based technique that generates code able to hide memory latency and better utilize simple OoO processors. By clustering loads found across basic block boundaries, Clairvoyance overlaps the outstanding latencies to increase memory-level parallelism. We show that these simple OoO processors, equipped with the appropriate compiler support, can effectively hide long-latency loads and achieve performance improvements for memory-bound applications. To this end, Clairvoyance tackles (i) statically unknown dependencies, (ii) insufficient independent instructions, and (iii) register pressure.

Clairvoyance achieves a geomean execution time improvement of 14% for memory-bound applications, on top of standard O3 optimizations, while maintaining compute-bound applications' high performance.

3.4 FPGAs vs. GPUs: How to Beat the Beast

Frank Hannig (Friedrich-Alexander University Erlangen-Nürnberg – Erlangen, DE)

Joint work of Frank Hannig, Richard Membarth, M. Akif özkan, Oliver Reiche, Moritz Schmid, Jürgen Teich

Graphics processing units (GPUs) and field-programmable gate arrays (FPGAs) are often employed as accelerators for computationally intensive applications. Both architectures have an abundant number of computational resources in common, albeit of different type and granularity. However, when it comes to programming, FPGAs and GPUs are greatly different. To bridge this sort of programmability gap, domain-specific languages (DSLs) are a promising solution, since they separate algorithm development from parallelization and low-level implementation details on an actual target architecture. Thus, DSLs offer high flexibility among heterogeneous hardware targets, such as CPUs and GPUs. With the recent rise of high-level synthesis (HLS) tools, such as Xilinx Vivado HLS and Intel/Altera OpenCL, FPGAs become tame. Particularly in the domain of image processing, applications often come with stringent requirements regarding performance, energy efficiency, and power, for which FPGAs have been proven to be among the most suitable architectures.

We present the Hipacc framework, a DSL and source-to-source compiler for image processing. We show that domain knowledge can be captured to generate tailored implementations for C-based high-level software from a common high-level DSL description targeting FPGAs. Our approach includes FPGA-specific memory architectures for handling point and local operators, as well as several high-level transformations. We evaluate our approach by comparing the resulting hardware accelerators to GPU implementations, generated from the same DSL source code.

3.5 Structured Parallel Programming: Code Generation by Rewriting Algorithmic Skeletons

Michel Steuwer (University of Glasgow – Glasgow, GB)

Joint work of Lift team

I will argue that we should structure our parallel programs using higher-level primitives which encode higher-level semantics explicitly. I will draw on similarities to "structured programming" which introduced concepts such as 'while' and 'for' loops as an answer to the software crisis in the late 1960s. The arguments made by Dahl, Dijkstra, and Hoare are still valid today but we need to revisit them in the context of parallelism.

I will discuss our work on the Lift project (http://www.lift-project.org) which introduces a set of data-parallel high-level primitives, called algorithmic skeletons, which are used to express programs in an abstract purely functional way. A rich exploration process optimizes these programs by rewriting the high-level program into low-level programs which encode implementations and optimization decisions explicitly. I will present encouraging performance result and sketch our ongoing research.

3.6 Rewriting with an Index-Based Intermediate Representation

Charisee Chiw (University of Chicago – Chicago, US)

Joint work of Charisee Chiw, John Reppy

The EIN representation is a hybrid design that embeds expression trees into a normalized SSA representation. The representation can concisely and clearly express the indexing of different arguments involved in a large computation as a single term. Invariant terms can be moved in and out of loops with simple pattern matching, rewriting, and analysis. In the future, we want to examine the entire computation at an earlier stage of compilation. Then we can develop a more advanced approach to loop optimization.

We designed an intermediate representation (IR), EIN, for tensor math. This design preserves the useful properties of the SSA representation, while providing flexibility in the specification of tensor and tensor-field operations. The key property of this design is that it allows reference to indices in the body of EIN expressions, while also providing a compact representation of the nested iteration that is implicit in the definition of tensors and tensor fields. A single EIN term consists of a tensor or field variable binding and indices. The index binding and ordering describes the shape and sampling of each argument. EIN supports standard linear algebra operations on tensors, such as addition, subtraction, the dot product, as well as other tensor operations such as the double-dot (colon) product, the outer product, and trace. One can think of the index space as defining an n-deep loop nest over the index variables, where the EIN expression is the loop body that defines scalar components of the result.

By examining the indices in EIN terms, we can do some modest loop invariant code motion. Each summation operator represents a loop nest that will be unrolled later in the compilation. Thus, moving operations outside the summation can avoid subsequent code duplication. This transformation is essentially loop-invariant code hoisting for the special case

of summations. When there are nested summations, our method applies additional analysis to see whether the summation can be converted into the product of independent summations. We identify loop invariants by looking at the indices. We are currently wondering whether we can expand on this simple idea to improve code generation. If we take a step back and look at the mathematical structure of the larger computation, we can leverage this knowledge into efficient code.

3.7 Synthesis of Modular Parallelism for Nested Loops

Victor Nicolet (University of Toronto – Toronto, CA)

Joint work of Victor Nicolet, Azadeh Farzan

Parallelizing loops is notoriously difficult when there are true dataflow dependencies that forbid parallelization using the wealth of sound code transformation techniques studied over the years. In previous work, we tackled the problem of finding divide-and-conquer parallel implementations of sequential loops. The idea was to first lift the loop by discovering and adding new computation of information that is redundant in the sequential program, but necessary for an efficient divide-and-conquer parallelization. This approach was specific to a class of loops that traverse a linear iteration space and compute a function of a sequence, and does not generalize to nested loops over multidimensional data. We propose a modular approach to analyze these by treating each loop nest separately. First, we encapsulate the inner loop to abstract its effect in the outer loop. Then, we explain how parallelizing the outer loop, that uses only this encapsulation, can give us a parallel implementation of the initial loop nest. I will talk about how this lets us leverage our existing automatic parallelization solution for single loops to one for more sophisticated loops.

3.8 Multidimensional Scheduling in the Polyhedral Model

Louis-Noël Pouchet (Colorado State University – Fort Collins, US)

Compositions of loop transformations are represented in the polyhedral compilation framework using scheduling functions, represented as integer matrices. To find a good schedule, two approaches can be employed: (a) computing each row of the scheduling matrix one at a time, typically solving one integer linear program (ILP) for each row, as done for example by the Pluto algorithm; and (b) computing all rows at once, using a single but usually more complex ILP, as done in the Ponos tool.

We will focus on one-shot multidimensional scheduling, where a single ILP is formulated to find the entire scheduling matrix. We will present the generic space of legal schedules implemented in Ponos, and show how to quickly design scheduling objectives, e.g., for specific parallelization or data locality patterns. We will present an interactive interface to the Ponos tool, to facilitate the design of new ILP-based multidimensional scheduling techniques.

3.9 Iterative Schedule Optimization for Parallelization in the Polyhedron Model

Stefan Ganser (University of Passau – Passau, DE)

Joint work of Stefan Ganser, Armin Größlinger, Norbert Siegmund, Sven Apel, Christian Lengauer

The polyhedron model is a powerful model to identify and apply systematically loop transformations that improve data locality (e.g., via tiling) and enable parallelization. In the polyhedron model, a loop transformation is, essentially, represented as an affine function. Well-established algorithms for the discovery of promising transformations are based on performance models. These algorithms have the drawback of not being easily adaptable to the characteristics of a specific program or target hardware. An iterative search for promising loop transformations is more easily adaptable and can help to learn better models. We present an iterative optimization method in the polyhedron model that targets tiling and parallelization. The method enables either a sampling of the search space of legal loop transformations at random or a more directed search via a genetic algorithm. For the latter, we propose a set of novel, tailored reproduction operators. We evaluate our approach against existing iterative and model-driven optimization strategies. We compare the convergence rate of our genetic algorithm to that of random exploration. Our approach of iterative optimization outperforms existing optimization techniques in that it finds loop transformations that yield significantly higher performance. If well configured, then random exploration turns out to be very effective and reduces the need for a genetic algorithm.

3.10 The Polyhedral Model Beyond Static Compilation, Affine Functions and Loops

Philippe Clauss (University of Strasbourg – Strasbourg, FR)

The polyhedral model has been proven to be a powerful framework for automatic analysis & transformation of loops. However, it suffers from strong limitations since it is mostly limited to "Fortran like" loops and linear transformations. We show that its scope and efficiency can be extended either by extending its mathematical objects to polynomials and algebraic expressions, or thanks to its use at run time. Run-time (speculative) polyhedral compilation opens new challenging opportunities for handling more general (non-linear) loops or handling non-loop programs that have a looping behavior, while algebraic expressions provide greater effectiveness and a larger scope of loop transformations. We illustrate the presentation with the speculative polyhedral optimization framework Apollo, and with the Trahrhe expressions ("Ehrhart" read backwards), which are the inverse of ranking Ehrhart polynomials. Some uses of Trahrhe expressions are presented: collapsing of non-rectangular loops and algebraic tiling.

3.11 Efficient Online Tuning of Accelerator Mapping Decisions

Philip Pfaffe (KIT – Karlsruhe, DE)

Automatic parallelization is a key component in state-of-the-art industry-grade compilers as well as research. With polyhedral optimization, even parallelization for heterogeneous platforms is realizable within a well-structured framework. Nevertheless, achieving optimal or even near-optimal performance with automatic transformations is hard, courtesy of a multitude of degrees of freedom inherent to platform specific optimization and parallelization. Fortunately, autotuning is an already established technique to deal with optimizing performance in the presence of high-dimensional search spaces. In this positional talk, we will examine the benefits that online-autotuning can offer to heterogeneous parallelization, and discuss promising future directions in tightly coupling autotuners with parallelizing compilers.

3.12 Loop Execution Time Modeling

Julian Hammer (Friedrich-Alexander University Erlangen-Nürnberg – Erlangen, DE)

The modeling of loop execution time allows us to evaluate optimization potentials prior to run time. If found in closed form, they may even directly yield optimization parameters, derived from hardware and code properties. We will present the layer condition cache model and ongoing efforts in instruction-level out-of-order execution time prediction. We believe that the integration of such models in compilers and auto-tuning tools will largely reduce – or even eradicate – test-runs, decrease guessing and support informed choices during build time.

3.13 Compiling Tensor Algebra for Finite-Element Computations

Lawrence Mitchell (Imperial College – London, GB)

Joint work of Lawrence Mitchell, Thomas Gibson, David A. Ham, Miklós Homolya, Paul H. J. Kelly, Fabio Luporini, Tianjiao Sun

At the core of a PDE, any library that uses finite elements is a large tensor contraction. Providing a low flop count, highly efficient implementation of this contraction is either devolved to the computational scientist (and then a general-purpose compiler), or else to a domain-specific compiler (and thence to a general-purpose one).

I will talk about the domain-specific compiler, and the optimisation passes, that we use in the Firedrake project (www.firedrakeproject.org), that deliver low algorithmic complexity algorithms on a class of finite elements that exhibit kronecker product structure.

I will then cover some open questions and future research directions, in particular how to extend the code transformation pipeline to incorporate operations on tensors that are not easily expressible as scalar indexed expressions: of particular interest is to widen the applicability to include tensor inverse and determinant calculations.

3.14 Automated Cross-Element Vectorization in Firedrake

Tianjiao Sun (Imperial College – London, GB)

Joint work of Tianjiao Sun, Lawrence Mitchell, David A. Ham, Paul H. J. Kelly

Firedrake is a domain-specific language embedded in Python for numerical solution of partial differential equations (PDEs) using the finite-element method. Firedrake provides the users with a high-level interface to express the problems in a high-level mathematical language while generating efficient low-level code. The internal intermediate representations in this code generation pipeline offer performance optimization opportunities at different levels of abstraction. We present one of the latest developments in Firedrake which enables automated vectorization across elements on unstructured meshes for typical finite-element assembly kernels, so as to address the problem of better performance and hardware utilization on SIMD architectures.

Modern CPUs increasingly rely on SIMD instructions to achieve higher throughput and better energy efficiency. It is therefore important to vectorize sequences of computations in order to sufficiently utilize the hardware today and in the future. This requires the instructions to operate on groups of data that are multiples of the width of the vector lane (e.g., 4 doubles, 8 floats on AVX2 instructions). Finite-element computations usually require the assembly of vectors and matrices which represents differential forms on the domain. This process consists of applying a local assembly kernel to each element, and increment the global data structure with the local contribution. Typical local assembly kernels suffer from issues that often preclude efficient vectorization. These include complicated loop structure, poor data access patterns, and loop trip counts that are not multiples of the vector width. General-purpose compilers often perform poorly in generating efficient, vectorized code for such kernels.

We present a generic and portable solution in Firedrake based on cross-element vectorization. Although vector-expanding the assembly kernel is conceptually clear, it is only enabled by applying a chain of complicated loop transformations. Loo.py is a Python package that defines array-style computations in the integer-polyhedral model and supports a rich family of transformations that operate on this model. In Firedrake, we adapt the form compiler, TSFC, to generate Loo.py kernels for local assembly operations, and systematically generate data gathering and scattering operations across the mesh in PyOP2. Firedrake drives loop transformations using Loo.py from this high-level interface to generate efficient code vectorized across a group of elements which fully utilizes the vector lane. This tool chain automates the tedious and error-prone process of data layout transformation, loop unrolling and loop interchange, while being transparent to the users.

We will present experimental results performed on multiple kernels and meshes. We achieve speedups consistent with the vector architecture available compared to baseline which vectorizes inside the local assembly kernels. The global assembly computations reach tens of percent of hardware peak arithmetic performance.

3.15 Automated Loop Generation for High-Performance Finite Differences (and Beyond)

Fabio Luporini (Imperial College – London, GB)

Joint work of Fabio Luporini, Charles Yount, Mathias Louboutin, Navjot Kukreja, Philipp Witte, Tim Burger, Michael Lange, Felix Herrmann, Gerard Gorman

We present the architecture and performance of Devito, a system to express numerical kernels in high-level mathematical notation. We focus, in particular, on the generation of highly optimized operators for seismic inversion. These involve solving partial differential equations (via finite differences) as well as other non-trivial mathematical operations (e.g., sparse points interpolation). The codes that need to be generated by the Devito compiler are therefore quite complex, including arbitrary, non-perfect nests of regular or irregular loops. We discuss the design of the compiler and the performance of the generated code for production-level seismic operators, showing roofline models for two Intel architectures (Skylake, KNL).

3.16 Implementations of Loop Constructs

Shigeru Chiba (University of Tokyo – Tokyo, JP)

This presentation compares and discusses several implementation techniques of programming language constructs for loops. Programmers need appropriate abstraction for describing their loops. Compiler developers also need it to retrieve optimization hints related to data dependency and other kinds of memory access patterns seen in the loop. Such abstraction will be provided as (built-in) domain-specific data types and operators for programmers. A question is how to implement these data types and operators to deliver good performance. This presentation shows an overview of several techniques including simple object-oriented libraries (also known as shallow embedding), C++ template meta programming, external DSLs developed from scratch, pragmas, deep embedding, and deep reification. Benefits and drawbacks of those techniques are mentioned. An important metric here is the implementation costs of abstraction (the data types and operators) since providing abstraction designed for a smaller application domain will be feasible if its implementation cost is not expensive.

3.17 Loop Iterations – Aligned and/or Pipelined?

Ayal Zaks (Intel and Technion – Haifa, IL)

There are two distinct and complementary transformations that can be applied to the iterations of a loop: aligning them to achieve data-level parallelism, also known as vectorization, and pipelining them according to an iteration initiation interval to achieve instruction-level parallelism at fine grain, and double-buffering to achieve memory-level parallelism at coarse grain. Vectorization, also related to loop coarsening, can handle uniform branches and certain

dependencies. Pipelining can handle dependencies but not branches. Both are applied to countable loops, or loops whose trip count is known a few iterations ahead of time.

Aligning iterations is supported by data-parallel heterogeneous programming models such as OpenCL's ND-range, facilitating both SIMD execution and dynamic load balancing across massively parallel GPUs. Pipelining and double-buffering, however, stitches all iterations together and leads to the static allocation of all iterations on one device. As a result, there is a mismatch between data-parallel models and deeply pipelined devices, such as FPGAs, which we seek to resolve.

3.18 Parallelizing Dependent Computations

Madanlal Musuvathi (Microsoft Research – Redmond, US)

Joint work of Madanlal Musuvathi, Mike Barnett, Saeed Maleki, Todd Mytkowicz, Yufei Ding, Daniel Lupei, Charith Mendis, Mathias Peters, Veselin Raychev

Parallelization is often synonymous with identifying independent subcomputations. On the other hand, it is well known that certain dependent computations, such as summing all elements in an array, can be parallelized by using the associativity of the operations involved. I will present our recent work on generalizing this insight to mechanically parallelize computations that appear inherently sequential.

The basic idea is to treat dependences as symbolic unknowns and use techniques inspired by program analysis and symbolic execution to execute dependent computations in parallel. Applications include large-scale stream processing and machine learning.

3.19 Communication-Optimal Loop Tilings (Keynote)

James Demmel (University of California – Berkeley, US)

Joint work of James Demmel, Michael Christ, Grace Dinh, Nicholas Knight, Alex Rusciano, Thomas Scanlon, Katherine Yelick

It is well-known that, given a two-level memory hierarchy with a fast memory of size M, the optimal loop tiling for classical $O(n^3)$ matrix multiplication, $C = A * B$, that minimizes data movement (communication) between fast and slow memory, tiles A, B and C into square tiles of size $\Theta(M^{1/2})$, and achieves a communication lower bound of $\Omega(n^3/M^{1/2})$. We extend this result as follows: Given any perfectly nested set of loops, with any number of arrays accessed in the innermost loop, each of which may have any number of subscripts, where each subscript may be an arbitrary affine function of the loop indices (e.g., $A(i, i-j, i+2*j-3*k+4, \dots)$), we present algorithms for computing (1) a constant e so that $\Omega(\#\text{loop\_iterations}/M^e)$ is a communication lower bound, and (2) an optimal loop tiling that achieves this lower bound. The lower bound assumes any execution order in which the loop bodies are not interleaved, and also that array entries cannot be allocated, used arbitrarily often, and then freed/discarded, without requiring any memory traffic. The proof depends on a recent discrete extension of the well-known Brascamp-Lieb inequality by Terry Tao, Michael Christ and others.

The optimal tiling makes two assumptions:

1. The loop bounds are large enough to fit the optimal tile. When this is not the case (e.g., think of matrix-vector multiply as a special case of matrix-matrix multiply) then it is possible to extend our results to get tighter lower bounds. We illustrate this with Convolutional Neural Nets (CNNs), expressible as seven nested loops, and provide a loop reordering that lowers the communication cost by a greater factor than possible for matrix multiply.
2. Data dependencies between loop iterations permit reordering. In the case of uniform-dependence algorithms, where data dependencies are represented by a finite set of constant distance vectors, it is possible to test whether data dependencies permit reordering. Generalizing the Brascamp-Lieb inequality to derive tighter lower bounds in the presence of dependencies is an open problem.

Time permitting, we can describe extensions of these results to memory hierarchies with multiple levels, and to distributed memory.

3.20 Effective Performance Modeling: A Grand Challenge for Loop Transformations in Compilers

P. Sadayappan (Ohio-State University – Columbus, US)

Joint work of P. Sadayappan, Changwan Hong, Aravind Sukumaran-Rajam, Sriram Krishnamoorthy, Louis-Noël Pouchet, Fabrice Rastello

A fundamental challenge for loop optimization is effective performance modeling. Existing loop optimizers in compilers generally use highly simplified performance models that do not necessarily correlate very well with actual realized performance on the target platform. This is particularly true of polyhedral loop optimization, where linear objective functions enable elegant ILP-based solutions. While transformations based on simplified models may improve performance over a naive baseline version, achieving performance comparable to hand-optimized code or code generated by domain-specific compilers is extremely challenging.

We describe an approach to performance modeling for GPU kernels that used abstract emulation of a small number of thread-blocks of the kernel. Key hardware resources like global memory, shared memory, functional units, etc. are modeled using two parameters: latency and gap (the inverse of throughput). Sensitivity analysis with respect to resource latency/gap parameters is used to predict the bottleneck resource for a given kernel's execution. Bottleneck analysis is in turn used for performance optimization. The approach hold promise in assisting manual code optimization, as well as automated model-driven auto tuning for performance enhancement.

3.21 Polyhedral Expression Propagation

Johannes Doerfert (Saarland University – Saarbrücken, DE)

Joint work of Johannes Doerfert, Shrey Sharma, Sebastian Hack

Polyhedral techniques have proven to be powerful for various optimizations, from automatic parallelization to accelerator programming. At their core, these techniques compute accurate dependences among statement instances in order to apply complex program transformations. Such transformations comprise memory layout or program order modifications by optimizing memory access functions or scheduling functions. However, these approaches treat statements as opaque entities and do not consider changing the structure of the contained expressions or the memory accesses involved.

We present a technique that statically propagates expressions in order to avoid communicating their result via memory. While orthogonal to other polyhedral optimizations, this transformation can be used to enable them. Applied separately, expression propagation can increase parallelism, eliminate temporary arrays, create independent computations and improve cache utilization. It is especially useful for streaming codes that involve temporary arrays and scalar variables.

For multiple image processing pipelines, we achieve portable speedups of up to 21.3x as well as a significant memory reduction compared to a naive parallel implementation. In 6 out of 7 cases, expression propagation outperforms a state-of-the-art polyhedral optimization especially designed for this kind of programs by a factor of up to 2.03x.

3.22 The isl Scheduler

Sven Verdoolaege (Facebook – Paris, FR)

isl is a library for manipulating integer sets and relations bounded by affine constraints, such as those that occur in polyhedral compilation. Next to several generic operations on such objects, isl also supports some operations tailored to polyhedral compilation, including a row-by-row scheduler.

After a general overview of isl focusing on the representation of fundamental concepts, some details are presented about the isl scheduler, including the representation and meaning of the input and the output, the available algorithms and their use of ILP solvers, as well as some issues that have been encountered in practice.

3.23 PolyJIT: Polyhedral Optimization Just in Time

Andreas Simbürger (University of Passau – Passau, DE)

Joint work of Andreas Simbürger, Sven Apel, Armin Größlinger, Christian Lengauer

While polyhedral optimization appeared in mainstream compilers during the past decade, its profitability in scenarios outside its classic domain of linear-algebra programs has remained in question. Recent implementations, such as the LLVM plugin Polly, produce promising speedups, but the re- striction to affine loop programs with control flow known at compile time continues to be a limiting factor. PolyJIT combines polyhedral optimization with multi-versioning at run time, at which one has access to knowledge enabling polyhedral optimization, which is not available at compile time. By means of a fully-fledged implementation of a light-weight just-in-time (JIT) compiler and a series of experiments on a selection of real-world and bench- mark programs, we demonstrate that the consideration of run-time knowledge helps in tackling compile-time violations of affinity and, consequently, offers new opportunities of optimization at run time.

3.24 Organizing Computation for High Performance Graphics & Visual Computing (Keynote)

Jonathan Ragan-Kelley (University of California – Berkeley, US)

Future visual computing applications – from photorealistic real-time rendering, to 4D light field cameras, to pervasive sensing and computer vision – demand orders of magnitude more computation than we currently have. From data centers to mobile devices, performance and energy scaling is limited by locality (the distance over which data has to move, e.g., from nearby caches, far away main memory, or across networks) and parallelism. Because of this, I argue that we should think of the performance and efficiency of an application as determined not just by the algorithm and the hardware on which it runs, but critically also by the organization of computations and data. For algorithms with the same complexity – even the exact same set of arithmetic operations and data – executing on the same hardware, the order and granularity of execution and placement of data can easily change performance by an order of magnitude because of locality and parallelism. To extract the full potential of our machines, we must treat the organization of computation as a first class concern while working across all levels from algorithms and data structures, to compilers, to hardware.

I will present facets of this philosophy in systems I have built for visual computing applications from image processing and vision, to 3D rendering, simulation, optimization, and 3D printing. I will show that, for data-parallel pipelines common in graphics, imaging, and other data-intensive applications, the organization of computations and data for a given algorithm is constrained by a fundamental tension between parallelism, locality, and redundant computation of shared values. I will focus particularly on the Halide language and compiler for image processing, which explicitly separates what computations define an algorithm from the choices of organization which determine parallelism, locality, memory footprint, and synchronization. I will show how this approach can enable much simpler

programs to deliver performance often many times faster than the best prior hand-tuned C, assembly, and CUDA implementations, while scaling across radically different architectures, from ARM cores, to massively parallel GPUs, to FPGAs and custom ASICs.

3.25 AnyDSL: A Partial Evaluation System for Programming High-Performance Libraries

Roland Leißa (Saarland University – Saarbrücken, DE)

Joint work of Roland Leißa, Klaas Boesche, Sebastian Hack, Richard Membarth, Arsène Pérard-Gayot, Philipp Slusallek, André Müller, Bertil Schmidt

Nowadays, the computing landscape is becoming increasingly heterogeneous and this trend is currently showing no signs of turning around. In particular, hardware becomes more and more specialized and exhibits different forms of parallelism. For performance-critical codes it is indispensable to address hardware-specific peculiarities. Because of the halting problem, however, it is unrealistic to assume that a program implemented in a general-purpose programming language can be fully automatically compiled to such specialized hardware while still delivering peak performance.

We present AnyDSL. This framework allows to embed a domain-specific language (DSL) into a host language. On the one hand, a DSL offers the application developer a convenient interface; on the other hand, a DSL can serve to specify domain-specific optimizations and effectively map DSL constructs to various architectures. In order to implement a DSL, one usually has to write or modify a compiler. With AnyDSL, though, the DSL constructs are directly implemented in the host language while a partial evaluator removes any abstractions that are required in the implementation of the DSL.

3.26 Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions

Nicolas Vasilache (Facebook – New York City, US)

Deep learning models with convolutional and recurrent networks are now ubiquitous and analyze massive amounts of audio, image, video, text and graph data, with applications in automatic translation, speech-to-text, scene understanding, ranking user preferences, ad placement, etc. Competing frameworks for building these networks such as TensorFlow, Chainer, CNTK, Torch/PyTorch, Caffe1/2, MXNet and Theano, explore different tradeoffs between usability and expressiveness, research or production orientation and supported hardware. They operate on a DAG of computational operators, wrapping high-performance libraries such as CUDNN (for NVIDIA GPUs) or NNPACK (for various CPUs), and automate memory allocation, synchronization, distribution. Custom operators are needed where the computation does not fit existing high-performance library calls, usually at a high engineering cost. This is frequently required when new operators are invented by researchers: such operators suffer a severe performance penalty, which limits the pace of innovation.

Furthermore, even if there is an existing runtime call these frameworks can use, it often does not offer optimal performance for a user's particular network architecture and dataset, missing optimizations between operators as well as optimizations that can be done knowing the size and shape of data. Our contributions include (1) a language close to the mathematics of deep learning called Tensor Comprehensions, (2) a polyhedral just-in-time compiler to convert a mathematical description of a deep learning DAG into a CUDA kernel with delegated memory management and synchronization, also providing optimizations such as operator fusion and specialization for specific sizes, (3) a compilation cache populated by an autotuner.

3.27 Loop Synthesis for Basic Linear Algebra Computations with Structured Matrices

Daniele G. Spampinato (Carnegie Mellon University – Pittsburgh, US)

Joint work of Markus Püschel

I describe the loop synthesis process in LGen, a research compiler designed for the generation of explicitly vectorized C code for small-scale basic linear algebra computations where input and output matrices may have a structure, such as lower triangular or symmetric. The input computation is expressed mathematically and the structures of the matrices on which it computes are described using a polyhedral notation. These structures, and the semantics of the operations contained in the computation, are used by LGen to produce a SCoP representation of the computation's iteration space. The resulting SCoPs are finally processed by an adapted version of the CLooG generator capable of synthesizing a mathematical formulation of the input computation where loops are expressed in terms of summations.

3.28 A Systematic Approach to High-Performance Generalized Matrix Multiplication Kernels

Richard Veras (Louisiana State University – Baton Rouge, US)

Joint work of Richard Veras, Tyler M. Smith, Tze Meng Low, Franz Franchetti, Robert van de Geijn

A large body of problems arising from linear algebra and big data can be represented by a generalization of the matrix-matrix multiplication operation. These domains are performance-critical and obtaining the level of performance seen in traditional dense matrix-matrix multiplication is a sought-after goal by the community. This is difficult because current high-performance matrix multiplication kernels are tuned for their target architecture and are written by hand in assembly code by expert programmers. Unfortunately, this approach is not sustainable for producing the large span of kernels that we are interest in. Therefore, our solution involves capturing the expert's knowledge and automating the application of this knowledge in the form of kernel code generator. The result is generalized matrix-matrix multiplication kernels that perform as well as an expert implementation.

3.29 Reasoning about Program Properties using Polyhedral Analysis

Sriram Krishnamoorthy (Pacific Northwest National Laboratory – Richland, US)

Joint work of Sriram Krishnamoorthy, Wenlei Bao, Sanket Tavarageri, Louis-Noël Pouchet, Fabrice Rastello, P. Sadayappan

Similarly to other program analysis techniques, polyhedral analysis can be used to reason about and check program properties. I will present a few use cases: detecting soft memory errors, checking program transformations, and modeling caching behavior. In addition, I will argue the need for robust and optimized tool chains to enable broader use of loop optimization techniques.

3.30 Using #pragmas to Direct Polly Transformations

Michael Kruse (ENS – Paris, FR)

Polly today decides automatically which loop transformations it applies, using isl's rescheduling algorithm. This might not always be the optimal transformation, so users may want to decide themselves which transformations lead to the most performant code. We are proposing to use #pragmas in the source code which enforce a specific transformation. These pragmas can either be inserted directly by the programmer, or, inserted by an autotuner framework which explores the search space of promising optimizations.

3.31 Polyhedral Optimizations toward Performance Portability

Jun Shirako (Georgia Institute of Technology – Atlanta, US)

Performance portability, the ability to enable sufficient performance across multiple hardware platforms, is getting more important in the era of extreme-scale heterogeneous computing. Compiler optimizations can play a key role in achieving this goal – i.e., enabling users to write simple and platform-independent programs and compilers to handle performance-oriented optimizations and customizations for the target system. We introduce a series of attempts to address this challenge based on the polyhedral model: (1) integration of polyhedral and AST-based transformations; (2) optimizations of explicitly parallel programs; (3) two-level parallelization for GPU accelerators; and (4) integration of data-layout and loop transformations.

Participants

- Cédric Bastoul
University of Strasbourg, FR
- Barbara M. Chapman
Stony Brook University, US
- Shigeru Chiba
University of Tokyo, JP
- Charisee Chiw
University of Chicago, US
- Philippe Clauss
University of Strasbourg, FR
- Albert Cohen
ENS – Paris, FR
- James W. Demmel
University of California – Berkeley, US
- Johannes Doerfert
Universität des Saarlandes, DE
- Andi Drebes
University of Manchester, GB
- Paul Feautrier
ENS – Paris, FR
- Stefan Ganser
Universität Passau, DE
- Armin Größlinger
Universität Passau, DE
- Tobias Grosser
ETH Zürich, CH
- Sebastian Hack
Universität des Saarlandes, DE
- Julian Hammer
Universität Erlangen-Nürnberg, DE
- Frank Hannig
Universität Erlangen-Nürnberg, DE
- Alexandra Jimborean
Uppsala University, SE
- Paul H. J. Kelly
Imperial College London, GB
- Sriram Krishnamoorthy
Pacific Northwest National Lab. – Richland, US
- Michael Kruse
ENS – Paris, FR
- Roland Leißa
Universität des Saarlandes, DE
- Christian Lengauer
Universität Passau, DE
- Fabio Luporini
Imperial College London, GB
- Benoit Meister
Reservoir Labs, Inc. – New York, US
- Lawrence Mitchell
Imperial College London, GB
- Madan Musuvathi
Microsoft Research – Redmond, US
- Victor Nicolet
University of Toronto, CA
- Philip Pfaffe
KIT – Karlsruher Institut für Technologie, DE
- Antoniu Pop
University of Manchester, GB
- Louis-Noël Pouchet
Colorado State University – Fort Collins, US
- Jonathan Ragan-Kelley
University of California – Berkeley, US
- P. (Saday) Sadayappan
Ohio State University – Columbus, US
- Jun Shirako
Georgia Institute of Technology – Atlanta, US
- Andreas Simbürger
Universität Passau, DE
- Daniele G. Spampinato
Carnegie Mellon University – Pittsburgh, US
- Michel Steuwer
University of Glasgow, GB
- Tianjiao Sun
Imperial College London, GB
- Nicolas Vasilache
Facebook – New York, US
- Richard M. Veras
Louisiana State Univ. – Baton Rouge, US
- Sven Verdoolaege
Facebook – Paris, FR
- Ayal Zaks
Technion – Haifa, IL