ISSN 2186-7437

NII Shonan Meeting Report

No. 249

Fast, furious, and yet correct tensor processing

Albert Cohen
Yukiyoshi Kameyama
Sven-Bodo Scholz
Oleg Kiselyov

May 18 - 22, 2026

National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-Ku, Tokyo, Japan

Background and Introduction

Tensor processing is an advanced form of Basic Linear Algebra (BLAS), which is *the* foundation of numeric programming in general – such as computational fluid dynamics and various simulations, from tsunami to epidemics. Tensor processing is also the foundation of Machine Learning, in particular, Large Language (Transformer) Models – and Neural Networks and Probabilistic Programming. The highest performance is crucial. The most challenging aspects of tensor processing are the number of dimensions, the sizes, the variety of storage/access formats, and the performance requirements.

To accomplish operations on huge (in the number of dimensions and especially in size) tensors in reasonable time, tensor processing often has to rely on special hardware, from GPU to Tensor Processing Units (TPU), Neural Processing Units (NPU), and to dedicated computer architectures such as Groq.

Industry trends and state of the art

Although the dedicated tensor processing hardware can deliver great performance, programming it is just as great a challenge. The first public, and the most well-known framework for tensor processing is TensorFlow, whose programming model, building explicit computational graphs, is quite different from conventional programming. There have been many attempts since to bridge the gap, of which one should mention PyTorch, which allowed writing programs in a subset of Python in more or less idiomatic style, and then 'compile' them to GPU(s) or other backends.

The great variety, and idiosyncrasy of tensor processors urgently called for some level of abstraction – hence the emergence and proliferation of *intermediates*: intermediate layers, DSL, or intermediate representations (IR). The examples include JAX (a framework for tensor transformation and compilation for GPU, which now underlies PyTorch), Futhark (array language for GPU programming) [2], Pallas (an elaboration of JAX mainly targeting TPUs); MLIR (or, a less lower level virtual machine, bring JAX and ordinary compiler optimizations into the same framework) [4] even has 'IR' in the name.

Main Problem: Little Trust, Few Guarantees

The intermediate DSL and IR undoubtedly help: that is why they are widely used. Yet they have not eliminated the need to *keep writing and re-writing* lowest-level (e.g., CUDA) kernels, *by hand*. One of the main reasons is the lack of trust: that the tensor processor code compiled/transformed from a higher-level description is really correct, and, especially, is fast.

One sees the urgent need in *correctness by construction*. Correctness has many aspects:

- at the very least, the generated tensor processing code should be well-formed and well-typed: its preparation and (up)loading to the processor should assuredly be error-free;
- shape correctness: all array accesses must be surely within bounds; dimensions of tensors to add, reduce, etc. must match;

- handling of sparse tensors and ragged shapes;
- correctly dealing with padding (appropriately masking padded data, etc.) and relating masking;
- reasoning about the accuracy of tensor contractions on floating point representations, and about the correctness of tensor algebraic transformations such as factorization and singular value decomposition;
- reasoning with and ensuring alignment;
- assuring race-freedom;
- assuring constraints on kernel scheduling and memory transfers;
- user-friendliness: although often overlooked, user-friendliness is crucial; if programmers cannot understand what the compiler is warning about, they would give up;
- formal, provable, mechanized correctness. Although one can hardly expect a compiler automatically prove that the generated code meets the specification, at the very least it should help in such a proof by reporting the used assertions and assumptions (which could then be used in an offline, mechanized or paper proof);
- some performance guarantees, such as assuring the lack of contention, of bank conflicts, etc.

Approaching Key Problems

The problem of correctness, broadly understood, is urgent. There are several approaches under development, such as:

- rank-polymorphism [5];
- tensor comprehensions and their transformations [6];
- biproducts [3];
- patterns of array programming discovered in APL;
- exterior algebra: a general approach to tensor manipulation [1];
- shape calculi and size types [2].

Although all these approaches have shown promise, they are still under development – in isolated communities. The goal of the seminar was to bring the developers together to talk about common problems and found insights, to stimulate further development and perhaps convergence.

Overview of the Meeting

As planned, the workshop participants consisted of representatives of several communities: advanced users of tensor processing frameworks; developers of many kinds tensor processing frameworks, DSLs and IR; algebraists; advanced users and developers of array languages – in short, practitioners, computer scientists and mathematicians.

We intended the meeting to primarily consist of discussion and cross-fertilization rather than standard conference-style presentations. We identified a few themes (above), and clustered presentations and discussions around them. We also identified more themes and challenges during the seminar.

List of Participants

- Albert Cohen, Google DeepMind (organizer)
- Yukiyoshi Kameyama, University of Tsukuba (organizer)
- Sven-Bodo Scholz, Radboud University (organizer)
- Oleg Kiselyov, Tohoku University (organizer)
- Pascal Costanza, Independent Researcher
- Martin Elsman, University of Copenhagen
- Jeremy Gibbons, University of Oxford
- Vinod Grover, NVIDIA
- Fritz Henglein, DIKU, University of Copenhagen
- Gabriele Keller, Utrecht University
- Thomas Koehler, CNRS
- Geoffrey Mainland, Drexel University
- Jose Nuno Oliveira, University of Minho and INESC TEC
- Adam Paszke, Google DeepMind
- Marc Pouzet, Ecole Normale Supérieure
- Ponnuswamy (Saday) Sadayappan, University of Utah
- Amir Shaikhha, University of Edinburgh
- Artjoms Sinkarovs, University of Southampton
- Tarmo Uustalu, Reykjavik University / Tallinn University of Technology
- Edward Valeev, Virginia Tech
- Ana-Lucia Varbanescu, University of Twente

Meeting Schedule

May 18 (Monday)

Theme: introductions; hardware rules!

- Self-introductions (5-7 mins)
- Ponnuswamy Sadayappan (Saday) (first tutorial)
- Challenge discussion

May 19 (Tuesday)

Theme: ML compute graphs, dynamic shapes, asynchrony; algebra, algorithms

- Vinod Grover (second tutorial)
- Adam Paszke
- Edward Valeev (third tutorial)
- Martin Elsman

May 20 (Wednesday)

Theme: performance analysis and engineering; excursion

- Ana-Lucia Varbanescu (fourth tutorial)
- Albert Cohen
- Discussion 1: Beyond Libraries
- Informal challenge discussions (during and after excursion and dinner)

May 21 (Thursday)

Theme: compiler construction; autodiff; correct tensor computing

- Gabriele Keller
- Thomas Koehler
- Amir Shaikhha
- Fritz Henglein
- Artjoms Sinkarovs
- Pascal Costanza
- Jose Nuno Oliveira
- Marc Pouzet
- Discussion 2

May 22 (Friday)

Theme: final discussions; conclusions

- Peter Braam
- Albert Cohen (numerics)
- Closing discussion: Take-away and next steps

The abstracts are given in the order of the presentations, in the schedule

Domain-Specific Optimization for Tensor Computing Tutorial

Ponnuswamy (Saday) Sadayappan, University of Utah

A high level overview of software factors affecting the performance of tensor computations, and a survey of program transformations addressing performance anomalies.

Shapes and Fast and Furious Asynchrony

Vinod Grover, NVIDIA

A survey and discussion of shape analysis and verification techniques for concurrent, distributed and heterogeneous implementations of tensor algebra.

Safe Management of Memory Spaces, Illustrated on Flash Attention

Adam Paszke, Google DeepMind

Exploration of escape hatch mechanisms to orchestrate asynchronous computations and memory management on hardware accelerators. Illustrated on multiple flash attention variants and the JAX/Pallas framework.

Tensor Algebra Tutorial

Edward Valeev, Virginia Tech

Tensors, as natural representation of multivariate functions and their calculus, are the foundation of physical simulation based on classical and quantum fields. The curse of dimensionality limits the number of degrees of freedom that can be coupled at once, and pushing towards exact solution typically runs into the curse. The countercurse is structure, and real data and operators are almost always structured. I will survey the resulting "sparsity zoo" — element and block sparsity, rank sparsity, and their compositions — with quantitative illustrations whose densities span many orders of magnitude. Two families of structured representations often occur: tensor networks proper, in which cursed tensors (quantum states and their properties) are factorized (e.g. matrix product states / tensor trains, tree tensor networks, matrix product operators, etc), and cumulant or perturbative expansions, in which a sum of tensor networks encodes difference of a cursed tensor relative to a simple reference — the coupled-cluster ansatz being the canonical example.

Structure only pays off if it can be computed with, so the last part of the talk turns to software. Tensor algebra is now a foundational component that must be usable from the arenas where applications are actually composed, and must remain portable and fast across CPUs and accelerators — requirements in tension with the domain-specific nature of most sparsity-exploiting remedies. I

will illustrate with two tools from my group: TiledArray, a distributed-memory block-sparse tensor framework in which contraction is an asynchronously scheduled SUMMA over futures to tiles, and SeQuant, a symbolic tensor algebra engine that closes the loop from operator algebra down to executable numerics. I will close with the open problems — the richer operation basis of tensors-of-tensors, automatic discovery of exploitable structure, and error control.

Gate Fusion is Map Fusion

Martin Elsman, University of Copenhagen (Joint work with Troels Henriksen)

Most efficient state-vector quantum simulation frameworks are imperative. They work by having circuit gates operate on the global state vector in sequence and with each gate operation accessing and updating, in parallel, all (or large subsets of) the elements of the state vector. The precise access and update patterns used by a particular gate operation depend on which qubits the individual gate operates on.

Imperative implementations of state-vector simulators, however, often lack a more declarative specification, which may hinder reasoning and optimisations. For instance, correctness is often argued for using reasoning that involves bit-operations on state-vector indexes, which make it difficult for compilers to perform high-level index-optimisations.

In this work, we demonstrate how gate operations can be understood as maps over index-transformed state-vectors. We demonstrate correctness of the approach and implement a library for gate-operations in the data-parallel programming language Futhark. We further demonstrate that Futhark's fusion-engine is sufficiently powerful that it will ensure that consecutive gate operations on identical qubits are fused using map-map fusion. Moreover, we demonstrate that, using Futhark's uniqueness type system, state vectors may be updated in place. We evaluate the approach by comparing it with the state-of-the art state-vector simulators qsim and QuEST.

The Performance Impact of Memory Layouts: A Benchmarking Approach

Ana-Lucia Varbanescu, University of Twente

Data layouts can significantly impact application performance. However, choosing the best-performing layout for a given application, especially one using complex data structures, remains a challenge for different types of systems and applications. How the data-layout performance depends on factors such as the data access pattern and machine parameters is not sufficiently understood.

We systematically study the performance impact of data layouts. To this end, we start from a given array of (complex) data structures and (1) we generate all possible data layouts for the input array via structure splitting and data member reordering, (2) we generate and execute the kernels with each layout, and (3) we collect performance data for many array sizes. Through this exhaustive testing, we empirically quantify the impact of different layouts and observe patterns that can help develop data layout selection guidelines.

To demonstrate our approach, we implement two basic data access patterns (sequential and random), and run comprehensive bechmarking campaigns for both simple and complex data structures. Our results indicate that data layouts do have a significant impact on runtime performance when the input is sufficiently large. We identify several causes for the large performance gaps (up to 67%) between layouts and, based on these observations, suggest empirical guidelines for (semi-)automated data layout refactoring. However, we acknowledge that many of the observed performance gaps require additional investigation to pinpoint the actual causes behind their performance behaviour.

This talk is based on a work done together with Jolly Chen and Axel Naumann, and published at ICPE'25 and EuroPar'26.

Beyond Libraries

Albert Cohen, Google DeepMind

Argument for the compiler-based production of tensor algebra libraries whose interfaces reside at the "tile" level rather than the end-user application level. Motivated by hardware needs, and made possible by advanced in domain-specific compilation, ML-based program synthesis and large-scale autotuning.

Bridging High-Performance Computing and Verified Compilation: Research Challenges

Gabriele Keller, Utrecht University

The story of the Accelerate embedded domain-specific language. Safe and efficient high-performance computing in Haskell.

OptiTrust: Tool and Challenges

Thomas Koehler, CNRS

Safe interactive performance engineering of tensor computations: a separation logic perspective with semantics- and proof-preserving transformations.

Einstein Meets Codd: Tensor Algebra through the Lens of Relational Algebra

Amir Shaikhha, University of Edinburgh

In this talk, I will show synergies between tensor processing systems and relational databases. First, I will show that the problems of fusion and algebraic optimization have been solved using similar techniques across two communities. I will also show cases that databases are doing a much better job. Then, I will show how we can leverage techniques at the boundary of communities to solve database, machine learning, graph, and program analysis problems faster with different degrees of correctness.

Automatic Differentiation via Adjoint Affine Interpretation

Fritz Henglein, DIKU, University of Copenhagen

We show that automatic differentiation can be performed by (adjoint) affine interpretation of a program, using general differentiation rules for constant, linear and bilinear functions as well as for their sequential and parallel compositions. Bilinear functions include tensor, Schur, dot/inner products and other bilinear operators.

Linear operators can be represented by a combinatory domain-specific language (DSL) where representation of tensors using a symbolic tensor product operator is central for compact representation of high-dimensional linear operators. They support efficient computation of adjoints, which retains data parallelism. Adjoint affine interpretation combines producing the (total) derivative of a function at an input point as a linear operator and computing its adjoint. For scalar functions, this computes the gradient; it implements backpropagation.

Initial empirical results indicate that employing the DSL straightforwardly is not (yet?) competitive compared to mature high-level high-performance functional AD frameworks such as Accelerate+AD, FutharkAD, JAX/XLA, NablaSD, SaC with AD. We speculate that the combination of low-rank symbolic tensor representations and algebraic linear operator optimizations may usefully complement high-performance matrix, tensor network, sparse tensor and code (VJP, JVP) representations of linear operators; even more speculatively, it might be a useful technology for quasi-Newtonian optimization algorithms.

Correctness Meets Performance: From Agda to Futhark

Artjoms Sinkarovs, University of Southampton

We demonstrate a technique for developing high performance applications with strong correctness guarantees. Using a theorem prover, we derive a high-level specification of the application that includes correctness invariants of our choice. After that, within the same theorem prover, we implement an extraction of the specified application into a high-performance language of our choice. Concretely, we are using Agda to specify a framework for automatic differentiation (reverse mode) that is focused on index-safe tensors. This framework comes with an optimiser for tensor expressions and the ability to translate these expressions into Futhark. We specify a canonical convolutional neural network within the proposed framework, compute the derivatives needed for the training phase and then demonstrate that the generated code approaches the performance of TensorFlow code when running on a GPU.

GraphBLAS in Julia with Nonblocking Execution via Multi-stage Programming

Pascal Costanza, Independent Researcher

The GraphBLAS are a collection of sparse matrix operations targeted at graph algorithms. From the beginning, the GraphBLAS were designed for 'nonblocking execution' i.e., calls to GraphBLAS methods return as soon as the

arguments to the methods are validated and define a DAG of GraphBLAS operations. This lets GraphBLAS implementations fuse functions, elide unneeded objects, exploit parallelism, and so on. GraphBLAS implementations exist that utilise nonblocking execution but with limited scope. In this presentation, I describe our work-in-progress to implement GraphBLAS with support for aggressive nonblocking execution based on multi-stage programming in Julia. The work is inspired by Oleg Kiselyov et al.'s work on stream fusion, and adds mechanisms for algorithm selection based on expected container sizes and caching of compiled code based on method signatures.

From Categories to HPC Code — a Path for Correctness?

Jose Nuno Oliveira, University of Minho and INESC TEC

For the past decade, I have been interested in a categorial approach to linear algebra (LA) promoting it to a kind of ´lingua franca' for computing. This research line should encompass the "unification" of LA with relation algebra (= algebraic logic) and functional programming, under the aforementioned categorial umbrella.

I also have a long standing interest in parallel programming, dating back to my PhD in the Manchester Dataflow Machine group, in the remote 1980s. More recent discussions at WG2.1 meetings made me realize that such a categorial approach could be useful in designing "correct by construction" HPC code.

In fact, categorial LA offers matrix processing in a type-safe way that is rich, expressive and diagrammatic. In particular, the notion of a biproduct captures blocked LA constructively and generalizes vectorization in a way that can be used to express HPC code transformations. In particular, a Haskell library has been implemented to handle matrices in such a type-safe manner.

In this proposed talk I would like to present the overall strategy and describe recent work by some of my master's students who are looking at OpenBLAS routines from a reverse engineering perspective, attempting to map them to high-level biproduct-structured LA specifications. The idea is not only to certify such routines, but also to investigate possible, more polymorphic (also more parallel?) generalizations.

This work is still in progress and quite exploratory. Will it be possible to machine generate type-safe, correct-by-construction HPC code from such high-level specifications? Are there biproducts that we are still unaware of, capable of specifying new 'smart' matrix multiplication strategies? Can IA help in this regard?

Type Safety for Sizes without Hindrance

Marc Pouzet, Ecole Normale Supérieure

Many programming languages provide ways to define systems that are parameterized by a value that is ultimately known at compile-time. A simpler but common situation is when those parameters are integer sizes. Various examples exist, e.g., in synchronous circuits (a n-ary adder), in array-based computations based on map/fold iterators (n is then the size of an array size), train tracking models. This parametricity can be dealt with various techniques, for example,

with macro expansion or better, meta-programming techniques. How to ensure that the use of those sizes is correct, e.g., if n is the size of an array, that there is no out-of-the-bound access, that all elements of an array are defined, that a static recursion on n terminates?

This problem, in its general form, can be expressed as a dependent type system. In the general case, this may demand to the program to write a proof that two types are equivalent (that is, that to size expressions are equal) or limit the expressiveness of sizes to get a decidable decision algorithm, or accept that some correct program will be rejected. Several more limited but more ergonomic solutions have been proposed. E.g., Futhark which implements an ML-type system forces conversions to be explicit. The back-up solution is to throw an exception. What if not exception is possible like in a language for real-time programming like Scade? Colaco, Pauget et Pouzet have proposed an ML-type system where sizes are multi-variate polynomials. Type equality is done modulo equality of those polynomials. Yet, inequality constraints are unavoidable, simply because one has to check that integer size are representable with finite precision arithmetic.

In this talk, I will expose an alternative approach to the use of a general dependent type system and the explicit write of a proof. It applied to a DSL that target real-time applications where the size of all data must be statically know and statically allocated. The point-of-view is to replace resolution by computation: the typing phase check types and builds a constraint on sizes that is gathered in the type of a function parameterized by a size. This constraint is evaluated when it is closed. This approach can be combined with the use of a symbolic solver, e.g., z3, to check that size constraints can be satisfied. I will illustrate this system on the description of a synchronous model of a simple CPU from the Nisan ans Shoken book "computer elements". It is implemented in the new type system of Zelus of 2026.

State Time Geometry — A Model of Infrastructure and Parallel Processing

Peter Braam

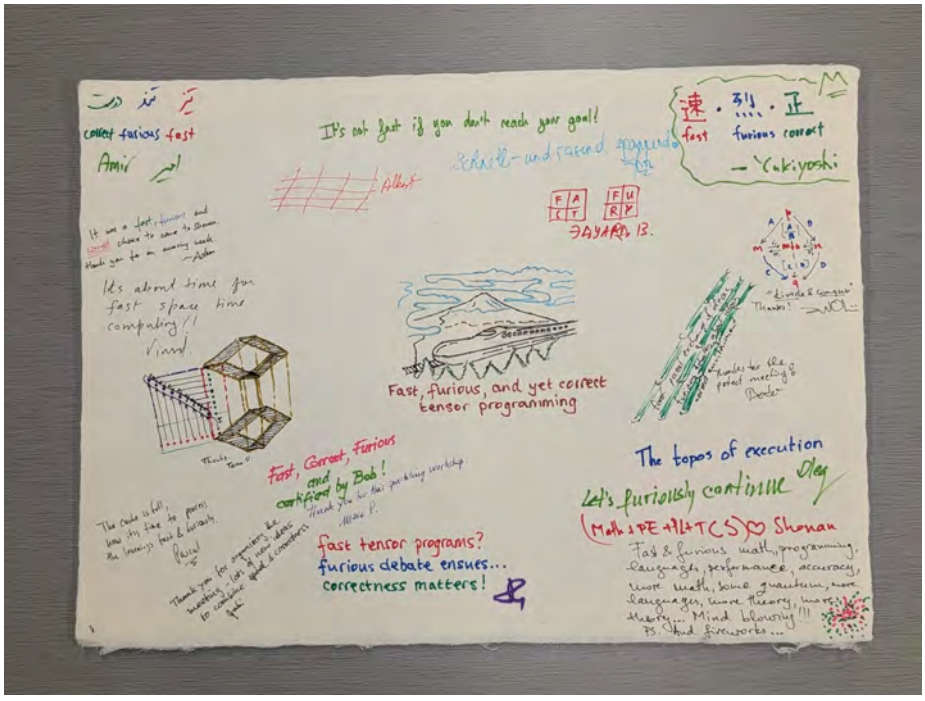
State-Time Geometry provides an intuitive and formal model of execution of parallel programs on infrastructure. Time is explicitly present and anchors the model in physics. State Time Graphs are nearly trivial to understand, their spatial components can describe computer architecture or abstract frameworks, e.g. a language runtime and the time dimension is familiar from execution. Familiar computer science notions – such as layering, composition, abstraction, failure, and recovery – acquire precise meanings. We will illustrate this reasoning power by substantially re-framing Kubernetes. Mathematics takes us further: the stochastic theory of rough paths delivers mechanisms to locate causal relationships affecting performance. Optimal transport theory, applied to data movement across the computational infrastructure, becomes a general mechanism to deliver a gradient across infrastructure, data and executions to improve performance. We illustrate these nascent concepts with a new approach to data processing for the SKA radio telescope.

Correct Numerics for Fast and Furious Compilers

Albert Cohen, Google DeepMind

Early thoughts about making domain-specific compilers numerical-precision-aware by lifting classical forward- and backward-error analyses of numerical expressions to the level of tensor algebraic intermediate representations.

Group Photo and Yosegaki



Summary of Discussions

The first two days of the seminar focused on tutorial-style presentations, while collecting discussion topics and organizing these into dominant themes and challenges. The group then voted to rank the challenges. We organized the next two and half days, interleaving rank-ordered discussions with presentations of research work.

Challenges for Plenary Discussion

The "Generate-and-verify" paradigm for correct and furious resource management (a.k.a. program synthesis)

- Formalization of hardware accelerator ISAs (operational or axiomatic) for translation validation and program synthesis.
- How to interact with hardware architects to define and refine those formalizations.
- Memory concurrency models (e.g. TileIR memory model, PTX, Triton).
- Hardware topology (memory and thread hierarchy, concurrency).
- Heterogeneous asynchronous programming: pipelining APIs and constructs (including MPMD), as well as fused collective communications.
- Static/type-based approaches.
- Separating correct transformations and semantics from optimization search.
- With or without considering numerical issues
- On which tensor IR?
- Proof-carrying approaches?
- Separation logic, e.g. OptiTrust?
- What resource management?
- In particular, what about expressing memory space management—including async/double-buffering (back-pressure)—using OptiTrust separation logic, or SAC IR?
- Use case: memory space management for flash attention, including TMEM on the latest Nvidia GPUs, assuming algebraic rewrites are known (monoidal divide-and-conquer).
- A potential application for effect systems? Or multi-party session types?
- What about adoption (programmers don't want to write proofs)?

Fast, furious yet "numerically correct" tensor computing

- Define a set of benchmarks with high-level specs (see FPBench for numerical accuracy).
- What is the user interface and compiler contract?
- What are the useful workflows and how much are they application-dependent?
- Numerics and beyond numerics: the cost of determinism (e.g. Martin Elsman's example with cuCollection hashmaps).
- Floating point emulation (Osaki schemes, bf16/fp8 for scientific computing).
- More broadly, exploring tradeoffs between numerical correctness and how furious is furious.

Shape and representation of tensors

- We need a taxonomy of shape inference, broadcast, partial evaluation, JIT compilation approaches in different languages and systems.
- Also, these questions should be considered in relation multi-stage programming and quasi-quotation (`‘tensor tensor`).
- What about layouts, HW-dependent, and the different conventions coupled with scheduling and asynchronous data movement?
- Are existing zero-cost abstractions sufficient/suitable for sparse linear algebra libraries, e.g. uniqueness types for in-place SparseLU fill-ins?
- Size types and their practical applications.
- What about the expressiveness of existing layout expression syntax?
- Discussion about what reshape entails—e.g. in-place or not in-place—and whether it should exist as a single primitive or not.
- Explore the tight coupling between sparse computations and data layout, the decomposition of the computation (indexing, splitting/stripmining, co-iteration), and the sparse representation itself; and the relation with container-iterator patterns.
- Explore ragged tensors and heterogeneous-shape nested tensors.

Performance engineering productivity and support

- Errors: examples, classes, anecdotes – per-application-domain, such as Patrik Jansson's type system for tensor notation(s).
- How can we improve the productivity of performance engineering? Are intuitive geometry visualizations effective? Do we need HCI (AgentCI?) experts to co-design tools?

- What do we need from the hardware for performance/efficiency *and* performance debugging? Is there a co-design space for these?
- Race checking for hardware accelerators, complex memory spaces/hierarchies, and scaling to multi-device/multi-node systems.
- How can we design an effective cooperation between human (insight) and compiler (automation)? And as a bonus, what about the role of AI?
- How can we prototype optimization ideas at all abstraction levels with low engineering effort?
- All of this while keeping in mind the view of application experts and system builders; for example here is the check list from the hyperscaler industry:
  - peak performance is non-negotiable;
  - portability is not the goal;
  - maximize productivity for performance engineers;
  - automate boilerplate—we should not stop at assembly or even C++ level;
  - never attempt to solve hard optimization problems (profitability and performance modeling, algorithmic complexity and robustness), unless completely infeasible otherwise (e.g. register allocation);
  - the compiler must be "predictable";
  - math/computations and scheduling intertwined;
  - every compiler decision must be overridable by the user.

Beyond libraries: deeper interfaces, composable performance

- Library construction vs. compiler construction.
- What is a library? seeing languages (and DSLs, MLIR dialects) as libraries, considering libraries of optimizations (AnyDSL, MimIR work).
- What is the interplay between user-control, auto-tuning, optimized libraries, synthesis and composability of existing software?
- Scheduling, fusion language design, calculus, semantics, and generalization of these (tile-level/BLAS2.5 interfaces, see the work of Jeremy Siek).
- Reignite the effort on Telescoping Languages and supercombinators.
- Can/should we blur the lines between software programming and hardware circuit synthesis? Seeing hardware as materialized code.
- There is a CISC revival: indexable register files vs. dedicated registers in the (systolic) ALU flow. What about programmability? What is the algorithmic impact at the tensor level?
- As far as performance and portability are concerned, is there a common formalization, abstraction, representation, for ML researchers and quantum chemists alike (and more domains)? Should we aim for one?

Other Discussion Themes

Many challenges and themes arose during the live discussion. We did not have time to cover each one of these properly into a dedicated session or focus group. We list some of these below and leave them as encouragement for future work by the community.

- Interactions between Automatic Differentiation (AD) and optimizations for tensor algebra.
  - Reverse mode AD for optimized (e.g. kernel-level) tensor algebra emerged from the scientific computing practice, including projects relying on the Enzyme framework written in C++ or Julia. Yet the question remains partly unanswered whether such an approach makes sense, given that the primal is typically optimized to avoid overlapping (conflicts) on writes but not on reads, and the co-tangent reverses the roles of reads and writes.
  - A few participants expressed interest in AD for streaming programs, in the context of reactive programming for control automation or machine learning. There is little literature on the topic, and both theory and practice need to be explored.
- Algebraic rewrites: is there a suitable mathematical and compiler framework to generalize flash attention and similar optimizations?
  - One possible approach would be index-free algebraic rewrites, as a representation to make the optimization space easier to navigate.
  - One could also capture the problem as the memoization of intermediate computation steps, and infer the optimization space from it (this is related to the incremental computation approach of the 70s).
- More broadly, the community shares a common pain to navigate the huge space of tensor IRs and compilers (particularly for AI).
  - How to answer such questions as "what framework, mathematical or software, should my new system derive from?"
  - How do different IRs interoperate? What is the formal semantics of their interaction?

Debates

Define "fast and furious" From the start of the seminar, there has been a general agreement that the focus should be on reaching "near-top performance for a given HW platform". Correctness being a requirement on top of this. How correctness is ensured may be considered secondary from an application perspective, but the need for strong correctness guarantees is paramount, as soon as we are dealing with "near-top performance".

Define "correct" Throughout the seminar, multiple formalizations of correctness emerged and were discussed. But there was generally an agreement on what correctness refers to, aside from numerical issues discussed below.

Define "numerically correct" This remains a sensitive topic. The participants agreed that the issue crosses other domains, on the application side (science or machine learning in particular). But there was little concrete progress on what it takes to automate numerical correctness, by construction/design, through a mechanized proof, or by means of formal verification. The problem (numerical correctness combined with automation) should probably be studied as a focused seminar in its own right.

Define "library" The notion of a "tile-level" interface complementing the usual application-level is recognized as an important one by performance engineers and compiler experts alike. How to achieve this, what level of automation and specialization it should take, and how to deploy extended libraries with tile-level interfaces remain open questions.

Define "single-source" The notion is well defined in a classical design and compilation flow. But what if autotuning and feedback-directed optimization comes into the picture? The question raises open reproducibility and correctness issues.

Resource management The question of the efficient management of memory spaces is important. With concurrency and asynchronous communication/computation being a particularly sensitive point of contention where performance and correctness interact. Kernel programming for flash attention is a very representative and important source of such issues.

Agentic AI We also discussed the impact on programming language abstractions on agentic AI, and reciprocally. And more broadly, how generative and agentic AI impact software engineering, performance engineering. The main interest of the participants being set on DSLs for tensor computing, but the question is of course much wider.

Research Directions

The previous section includes summaries of some of the promising approaches and open problems for each one of the major challenges.

The seminar participants agreed to collaborate in three joint directions of research:

1. experiments to compare the expressiveness, performance, correctness guarantees, and limitations of the different approaches and their adoption; with a focus on sparse and dense matrix multiplication and on the family of attention kernels;

2. explore the feasibility of a unified software framework for the performance engineering of higher-order tensor expressions, leveraging the experience of TiledArrays and TAPP, and considering different classes of hardware;

3. tile-level interfaces for kernel libraries, including application-driven specialization and synthesis.

Recommend Readings

Polyhedral compilation

- Cédric Bastoul, Contributions to High-Level Program Optimization (habilitation, 2012) https://icps.u-strasbg.fr/people/bastoul/public_html/research/papers/Bastoul_HDR.pdf
- Sven Verdoolaege, Presburger Formulas and Polyhedral Compilation (tutorial, 2021) https://libisl.sourceforge.io/tutorial.pdf
- Sven Verdoolaege, Polyhedral Compilation and the Integer Set Library (Trends in Arithmetic Theories, 2022) https://www.google.com/url?sa=t&source=web&rct=j&opi=89978449&url=http://www.cs.ox.ac.uk/people/christoph.haase/home/trends-in-arithmetic-theories/pdf/verdoolaege-trends-22.pdf

ML for superoptimization

- Adapting AlphaEvolve to Optimize Fully Homomorphic Encryption on TPUs https://arxiv.org/pdf/2605.14718

Mechanically proven correct HPC/ML

- Correctness Meets Performance: From Agda to Futhark https://dl.acm.org/doi/10.1145/3747524
- OptiTrust presentation: https://github.com/charguer/optitrust https://www.chargueraud.org/softs/optitrust

Automatic analyses of numerical correctness

- Daisy https://malyzajko.github.io/papers/tacas18_daisy_toolpaper.pdf
- Automated backward error analysis https://dl.acm.org/doi/10.1145/2814270.2814317

Artificial Intelligence

- Brian Cantwell Smith "The Promise Of Artificial Intelligence: Reckoning and Judgment" MIT Press, 2019 https://mitpress.mit.edu/9780262043045/the-promise-of-artificial-intelligence/

References

[1] John M. Browne. Grassmann Algebra: Exploring applications of extended vector algebra with mathematica. https://web.archive.org/web/20090219180241/http://grassmannalgebra.info/grassmannalgebra/book/index.htm, 2007.

[2] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. Futhark: purely functional gpu-programming with nested parallelism and in-place array updates. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 556–571. ACM, 2017.

[3] Hugo Daniel Macedo and José Nuno Oliveira. Typing linear algebra: A biproduct-oriented approach. *Sci. Comput. Program.*, 78(11):2160–2191, 2013.

[4] Multi-level IR compiler framework. https://mlir.llvm.org/.

[5] Sven-Bodo Scholz. Why rank-polymorphism matters. In Thomas Noll and Ira Justus Fesefeldt, editors, *KPS 2023: 22. Kolloquium Programmiersprachen und Grundlagen der Programmierung*, volume AIB-2023-03 of *Aachener Informatik-Berichte*, 2023. https://doi.org/10.18154/RWTH-2023-10034.

[6] Sven-Bodo Scholz and Artjoms Sinkarovs. Tensor comprehensions in SaC. In Jurriën Stutterheim and Wei-Ngan Chin, editors, *IFL '19: Implementation and Application of Functional Languages, Singapore, September 25-27, 2019*, pages 15:1–15:13. ACM, 2019.