ISSN 2186-7437

NII Shonan Meeting Report

No. 181

Programming Language Support for Emerging Memory Technologies

Jeremy Gibbons
Oleg Kiselyov
Peter Braam

May 7–10, 2024

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

Background and Introduction

History and summary goal

Optimal placement and movement of data between storage, memory and processing components has been the main driver for higher performance and lower energy consumption of all software. Attaining (or even approaching) optimality requires an accurate model. A flat homogeneous address space abstraction (the von Neumann model), presented by hardware and by the hardware abstraction layers of the OS/runtime, is becoming harder and harder to maintain and is less and less accurate as a model.

What new abstractions should compiler writers and programmers use instead? In the emerging landscape of new memory devices, with radically different processing elements in systems spanning from super-computers to IoT devices, how can we save users from being overwhelmed with the diversity and complexity of memory architectures? Conversely, how can we give programmers and compiler writers some idea of the memory architecture and some degree of control over data placement and movement?

Industry trends and state of the art

A revolution in the electronics and computer industry has started to deliver hybrid, widely diverse kinds of memory, such as off-package persistent (e.g. Intel's Optane Memory [6]) and many forms of volatile RAM, on-package RAM (e.g. High Bandwidth Memory [1]), and logic-integrated on-chip persistent RAM [9]. The kinds of memory becoming available show orders of magnitude of diversity in capacity, latency and throughput. Some kinds are already used as a cache generally controlled by hardware, others as part of the address spaces for traditional processors, while other memory is seen as a long term storage device. The ways to access memory are also diverse, from load/store instructions to bus programming to co-processors (as Google's Tensor Processing Units [2] and GPUs).

Limited support using hardware instructions, operating systems and low level libraries has been introduced, for example for cache coherency control, selective cache flush mechanisms [7] and NUMA addressability. The high-performance computing (HPC) community is familiar with analysing performance, and creating generally non-portable ad-hoc optimizations using fragile pragmas and qualifiers with poor diagnostics.

Compilers for mainstream high-level languages like Java, Swift, Golang, Scala, Python, Ruby, Javascript, OCaml and Haskell provide the user with the illusion of a flat address space, which fits well with the calculi and abstract machines that underlie programming language research. In practice, such languages rarely deliver good performance out of the box; they have widely varying features to optimize instruction scheduling, but much less developed memory management, and inspire many duplicated efforts between different compilers [8]. A key disconnect between the programming languages, computer architecture, and high-performance computing communities may be a root cause.

Domain Specific Languages (DSLs) have delivered promising results, and in some cases (e.g. Halide and Tensorflow) include automatic and guided analysis and interfaces for data placement and movement. But the high-level specifica-

tions of such DSL's (including memory specifications) are too model-specific, and so are hard to extend to different domains and to generalize.

Approaching key problems

The major opportunities offered by the new technologies are difficult to exploit with present-day programming languages:

- persistent-memory transactions (as introduced by Intel) have non-compositional lock management properties;
- large core counts make cache coherency electronically challenging;
- tier management of memory; placement and movement policies are in their infancy;
- portable support for persistent and non-persistent memory semantics in libraries appears elusive, even for structures as simple as a doubly-linked list. Leveraging persistent memory for programs that resume after interruptions is currently beyond reach.

To approach these problems, first we need control, or discipline: defining types to describe computer architectures, and providing methods to relate these with program data structures. Beyond control, we need a systematic approach to automatic optimization, possibly addressed as an optimization problem for an automated solver or machine learning, as done in some DSL's and in physical micro-electronics layout design systems. The control should be at a high level, with diagnostics e.g. for wrongly situated data.

Opportunities are also many, mostly unexplored. For example, one could demonstrate that unmodified NumPy code gains performance when Python observes data access patterns in its garbage collector and leverages them for data placement in memory tiers. New compiler developments like MLIR [5] may allow some control of memory placement and layout in high level languages. The fully understood performance model of the HPCG benchmark [4] may also provide many hints.

The Difficult Road to the Seminar

The seminar was originally scheduled to take place in the Fall of 2020. However, the COVID19 pandemic made that impossible. After several attempts to reschedule, the seminar finally took place in May 2024.

One of the many things that happened in the intervening period is that the Intel Optane memory [6], which was one of the main motivations for the seminar, has not attained the hoped-for commercial success and was withdrawn from the market. That is not to say that the problems the seminar was meant to address have become moot. First of all, as several participants of the seminar have emphasized, the present commercial failure of Optane memory is due to business issues (misguided marketing, fear of vendor lock-in, dearth of initial applications, failure to achieve the planned price, etc.). The technology itself is sound and may well make a come-back.

Mainly, memory diversity did not go away. The continued evolution and speciation of GPU, TPU, Neural Processing Unit (NPU) and non-conventional architectures such as Groq has made data layout and data access problems even more varied and difficult, and the need to reflect them in some way in programs even more pressing.

As a preparation for the Shonan seminar, we have conducted, in December 2022, a small-scale meeting hosted by Green Computing Center at Waseda University (supported by MEXT TGU Program "Waseda University ICT Robot Project" and Advanced Multicore Processor Research Insititute), organized by Professor Hironori Kasahara. Most participants of that meeting later attended the Shonan seminar.

Planning

We invited participants representing three groups of people: specialists in memory technologies, leaders in specific domains in which sophisticated memory management is being used, and computer scientists with a solid foundation in the structure of languages and compilers.

In organizing the seminar, we followed some of Shriram Krishnamurthi's helpful advice [3]. In particular, we invested as much effort as we could in advance on choosing topics, soliciting talk proposals and requests, and collating into coherent structure. We sent out a preliminary survey to determine general focus topics, from which we distilled the following list:

- formal models of emerging non-uniform memory architectures
- language features for separating memory layout from algorithmic content
- physical properties of emerging memory, with implication for cost models
- scheduling, autotuning, mathematical cost models
- metaprogramming, program generation, partial evaluation
- programming with persistent memory, transactions
- novel compute architectures (GPUs, TPUs...)

We then distributed a second survey based on those results, asking for more specific talk proposals and requests. We also asked everyone to give a short (2 to 5 minute) introduction to their interests and expectations.

We intended that the meeting should primarily consist of discussion and cross-fertilisation rather than standard conference-style presentations. We identified a few themes, and clustered presentations and discussions around them. Presentations of old work, or of other people's work, were very definitely fine if topically in scope.

Discussion Framework

Satnam Singh has suggested, and participants agreed on the following framework for presentation and discussion:

- When discussing a programming language system, who is the target: application programmer, compiler developer, run-time system, OS, firmware?
- Are we discussing something that an application may use, or is it something that an application must know about? (The answer may vary for different classes of applications).
- Which exact level memory we are talking about: "just memory" (flat address space) or a particular memory within a particular hierarchy?

Overview of Talks

A Case for Programmer Aware Accelerated Memory Access

Gary Grider, Los Alamos National Laboratory (online)

There are many applications, like sparse embeddings in recommender systems and sparse database joins, that have very irregular memory access characteristics. These applications are horribly underserved by the current application trends that are concentrating on dense memory access. Los Alamos has multi-physics simulations that exhibit irregular/sparse memory access characteristics making it hard to buy suitable hardware and difficult to gain high throughput on these important applications. Recently, much effort has been expended in attempting to address this irregular memory access through deep co-design between applications and prototype hardware with promising results. These efforts show that how you consume the acceleration is as important as the accelerations themselves which implies that software, programming models/systems, and programming languages will need to play a role in exploiting irregular memory access acceleration, beyond simple scatter/gather. This talk will cover the work that has been done and point towards potential programming techniques for helping these important and underserved applications via exposing parallelism in the memory access to the programmer.

Deep Codesign of Memory Systems in the Post-Exascale Computing Era

Jeff Vetter, Oak Ridge National Laboratory

The US Department of Energy has just deployed its first Exascale system at Oak Ridge National Laboratory. Now is an appropriate time to revisit our Exascale predictions from over a decade ago and think about post-Exascale. We are now seeing a Cambrian explosion of new technologies during this "golden age of architectures", making codesign of architectures with software and applications more critical than ever. In this talk, I will revisit the Exascale trajectory, survey post-Exascale technologies, and discuss their implications for system design and software. I will discuss the evolution of memory systems over time and the introduction of various non-volatile memories for HPC. In our NVL-C work, we developed a transactional-based compiler to allow users to access multiple NVM heap segments consistently with transactions, logging, and several optimizations to improve performance.

Pallas: A Multi-Platform High-Productivity Language for Accelerator Kernels

Adam Paszke, Google DeepMind

Compute accelerators are the workhorses of modern scientific computing and machine learning workloads. But, their ever increasing performance also comes at a cost of increasing micro-architectural complexity. Worse, it happens at

a speed that makes it hard for both compilers and low-level kernel authors to keep up. At the same time, the increased complexity makes it even harder for a wider audience to author high-performance software, leaving them almost entirely reliant on high-level libraries and compilers.

In this talk I plan to introduce Pallas: a domain specific language embedded in Python and built on top of JAX. Pallas is highly inspired by the recent development and success of the Triton language and compiler, and aims to present users with a high-productivity programming environment that is a minimal extension over native JAX. For example, kernels can be implemented using the familiar JAX-NumPy language, while a single line of code can be sufficient to interface the kernel with a larger JAX program. Uniquely, Pallas kernels support a subset of JAX program transformations, making it possible to derive a number of interesting operators from a single implementation. Finally, based on our experiments, Pallas can be leveraged for high-performance code generation not only for GPUs, but also for other accelerator architectures such as Google's TPUs.

Memory System Trends and Opportunities

Mattan Erez, University of Texas at Austin

Memory capacity and performance are critical for attaining high performance and system utilization. Memory technology, however, offers challenging tradeoffs between capacity, bandwidth, and cost, leading to complex and heterogeneous memory systems. The trends are toward more distributed and tiered memory systems arising from advances in the technology of packaging (e.g., HBM in-package memory), memory cells (e.g., non-volatile memories), signaling (e.g., high-speed and optical links), and architectures (e.g., the Compute eXpress Link). I will explain the basics of modern memory design, focusing on recent trends relating to DRAM, but also memory in general. I will then discuss other technological trends and constraints that are driving memory system architecture and leading toward tiered memories. Finally, I will discuss implications to software, including applications and system software that must manage the increasing heterogeneity.

Compiler Optimization Challenges for Matrix/Tensor Computations

Ponnuswamy Sadayappan, University of Utah

Production compilers today are extremely effective in lowering high-level programs to very compact machine code, i.e., they are very effective in minimizing the number of executed instructions in the compiled code. However, the dominant cost (both in terms of energy and execution time) on all computer systems today is not that of executing arithmetic/logic operations but of the movement of data, between processors of a parallel system and through the memory hierarchy at each processor. Despite significant research advances in compiler optimization for affine computations, such as the powerful polyhedral model for dependence analysis and loop transformation, it remains extremely challenging for any compiler today to generate optimized code (for either multicore CPUs

or GPUs) that achieves performance comparable to manually developed vendor libraries or autotuning optimizers. At the heart of the problem is the massive space of possible combinations of loop transforms like tiling and fusion, and the challenge of devising accurate cost models and efficient search/optimization strategies driven by the cost models. While the challenges are already stiff even for dense matrix/tensor computations, sparsity makes the performance optimization problem much harder. Finally, compiler optimization of programs for distributed-memory platforms, such as the spate of recently developed spatial accelerators for machine learning, is much more challenging than for shared-memory architectures like multicore CPUs and GPUs. Co-design of architectural mechanisms and data access abstractions driven by key computational patterns/paradigms could alleviate compiler optimization challenges.

Optimizing an Array Language in the Presence of Nested Parallelism

Cosmin Oancea, University of Copenhagen

This presentation highlights a compilation technique for nested-parallel applications that builds on the classical flattening transformation, but can adapt to hardware and dataset characteristics. Our solution uses the degree of utilized parallelism as the driver for generating a multitude of code versions, which systematically cover all mappings of the application's regular nested parallelism to the levels of parallelism exposed by the hardware. These code versions are then combined into one program by guarding them with predicates, whose threshold values are automatically tuned to hardware and dataset characteristics. The autotuning procedure is customized to the proposed code transformation, and is demonstrated to provide reliable and efficient results if a certain monotonicity property holds.

Rank Polymorphism and Memory Layouts

Sven-Bodo Scholz, Radboud University

Rank polymorphism enables algorithms to be specified in a way that allows arrays to be of statically unknown rank (dimensionality) and shape. As this presentation demonstrates, this capability does not only widen the applicability of operations, but it also enables a control of traversal schemata through array shapes rather than through a fixed choice in code. At several examples, we demonstrate how a fixed layout of higher dimensional arrays in memory facilitates shapes to guide various forms of program optimisations.

Democratizing Data Science by Leveraging Structure

Amir Shaikhha, University of Edinburgh

Modern data science pipelines employ a variety of workloads, including tensor algebra, graph processing algorithms, and relational query processing. This results in using a set of loosely coupled data processing frameworks that move

the data across the analytics pipeline, leading to unnecessary resource and energy consumption. This talk shows a compilation-based approach to move the computation closer to the data. This is achieved by designing (domain-specific) languages that leverage the structure of data with algebraic optimizations. We show that our proposed approach significantly outperforms the state-of-the-art frameworks for a wide range of applications, including database query processing and tensor processing.

Ribbit: Data Layout of Algebraic Data Types

Gabriel Radanne, INRIA Lyon

Initially present only in functional languages such as OCaml and Haskell, Algebraic Data Types (ADTs) have now become pervasive in mainstream languages, providing nice data abstractions and an elegant way to express functions through pattern matching. Unfortunately, ADTs remain seldom used in low-level programming. One reason is that their increased convenience comes at the cost of abstracting away the exact memory layout of values. Even Rust, which tries to optimize data layout, severely limits control over memory representation.

Ribbit presents a new approach to specify the data layout of rich data types based on a dual view: a source type, providing a high-level description available in the rest of the code, along with a memory type, providing full control over the memory layout. This dual view allows for better reasoning about memory layout, both for correctness, with dedicated validity criteria linking the two views, and for optimizations that manipulate the memory view. Ribbit provides optimized compilation of any code over ADTs for arbitrary mangled memory representation to a Destination-Passing-Style intermediate representation with explicit memory allocations and full support for recursive types. In this talk, we showcase some interesting usage examples, and part of the compilation process of Ribbit.

EverParse: Verified Secure Binary Data Parsing for All

Tahina Ramananandro, Microsoft Research

Software security exploits often begin with an attacker providing an unexpected input to a program, causing it to misbehave in a way that allows the attacker to gain access to a critical system. Such attacks are mostly due to binary data parsers and validators generally being written by hand in unverified languages.

To prevent such attacks, we have been designing EverParse, a framework to produce data validators formally verified for memory safety and functional correctness with respect to the data format, along with a choice of security properties such as unique binary representation for signature-based cryptographic authentication, constant-time memory accesses to avoid time-based side channels during parsing, and absence of double-fetches for concurrent settings, where users want high-performance parsing from a potentially attacker-controlled input memory region without fully copying its contents into a temporary buffer beforehand. We have leveraged such security guarantees by using EverParse in verified F* applications for various network protocols, including TLS, QUIC, ASN.1 X.509, etc.

To make verified parsing accessible to a wider range of users, we presented EverParse3D, a fully automatic generator producing efficient, verified data validators from data-dependent format descriptions via partial evaluation with zero user proof effort. EverParse3D has been in use in the Windows kernel since 2021, as part of the Microsoft Hyper-V network virtualization stack to filter malformed packets away from hosts and guests alike.

We also discussed avenues for automatic generation of data format specifications, with our ongoing approach, 3DGen, leveraging AI to produce data format specifications from IETF RFC standard documents, with a feedback loop including both AI-based and symbolic test case generation.

Further discussions investigated the use of verified parsing and serialization techniques to ensure safety, functional correctness, and security of high-performance memory layout.

EverParse is open-source and available on GitHub: `https://github.com/project-everest/everparse`

Persistent Lock-Free Data Structures for Non-Volatile Memory

Erez Petrank, Technion CS

In this talk I will discuss the design of lock-free (concurrent) data structures adequate for non-volatile RAM. I will shortly review the challenges in building software for non-volatile memory, and definitions for correctness of algorithms in this domain. I will then review constructions of persistent queues and sets, mention general transformation for building persistent concurrent data structures and discuss the basic techniques behind all.

Pulse, and Verification for GPUs

Guido Martinez, Microsoft

I will present Pulse, a language designed for verified and efficient imperative programming, based on dependent types and separation logic. I will show some previous work about verifying task-parallel programs in Pulse, including parallel versions of sorting algorithms, and how we reason about parallelism and asynchrony conveniently within the language. Finally, I will show some rough work in progress for verifying GPU kernels in Pulse.

Under: A Squiggol for Lenses and Conjugations

Juuso Haavisto, University of Oxford

"Under" is a dyadic operator in array programming languages that utilizes rank polymorphic function inverses. The BQN array language implements structural and computational cases of Under. In this work we showcase how the the lens properties of structural Under can be used to model parallel programming on GPUs.

The "Forward" Iteration or How to Iterate on Space with Operators on Time

Marc Pouzet, École normale supérieure

The Lustre and Scade languages allow to write equations on infinite sequences. A sequence defines a time evolving value. We speak here of iteration over time because the functions apply to successive elements of a sequence. Moreover, Lustre and Scade are called "data-flow" languages because a variable is defined by a single definition (equation), the order of computations being defined only by data dependencies.

The SISAL language (Stream and Iteration in a Single Assigment Language), born at the same time as Lustre and independently, also adopted the "data-flow" point of view but targeting array-based applications. In SISAL, an array can also be interpreted as a sequence that is finite. The 'for' loop of SISAL corresponds to an iteration in space which follows the order of the successive elements of an array.

What is the link between Luster and SISAL? In this presentation, I I will show recent work bringing together these two approaches and which leads to the introduction of a new iteration construct for a language such as Lustre and Scade. This construction, called "forward", allows to iterate a sequential function over sequences by applying it to an array. It can be used to write classical examples of linear algebra (e.g., sum arrays, tensor product, matrix/matrix product, Choleski). More broadly, it can be use to abstract a sequence of computations as a combinational one by grouping them into a single reaction. I will show a functional semantics of a Lustre kernel extended with this construct and give examples using the ZRun interpreter: `https://github.com/marcpouzet/zrun/tree/work`.

This work is close to the old idea of "temporal refinement" studied by Caspi and Mikac, for Lustre (thesis by Mikac, article FMICS'05); and the reactive and clock domains, by Mandel, Pasteur and Pouzet, for ReactiveML (Pasteur's thesis, article SCP'15).

This is a joint work with Jean-Louis Colano and Baptiste Pauget (Scade Core team, ANSYS Toulouse).

Data-Parallel Flattening by Expansion and Size-Dependent Types

Martin Elsman, University of Copenhagen

In this talk, we address two problems related to programming massively data-parallel algorithms. The first problem concerns the difficulty of expressing irregular nested parallel algorithms in a way that materialises into practical efficient code on GPUs. The second problem concerns the difficulty that programmers often have with respect to expressing size constraints on array data and ensuring that such constraints are satisfied.

For addressing the first problem, we suggest a method that applies to certain classes of irregular nested parallelism and that captures a solution to the problem using programmer-level design patterns (i.e., higher-order functions) for hiding flattening details (e.g., flag vectors).

For addressing the second problem, we suggest a mechanism for so-called size-dependent types, which allows programmers to express constraints on array sizes and which supports a high degree of implicit static checking of array sizes.

Both techniques are developed in the context of the Futhark compiler, a strict data-parallel pure functional programming language and compiler aimed at executing programs efficiently on massively-parallel hardware (e.g., GPUs).

Rank polymorphic Parallel Array Computations in Accelerate

Gabrielle Keller, Utrecht University

Accelerate is a Haskell EDSL for parallel array computations. Parallel computations are expressed via higher order parallel operations, such as maps and reductions, which can be parametrised with sequential computations. The parallel operations are rank polymorphic, in the sense that they can be applied to arrays of any dimensionality. In addition to arrays of primitive and tuple types, the language also supports arrays of (non-recursive) user defined data types. Through the use of type families, GADTs and generics, these are mapped automatically to efficient, machine-friendly representations.

Memory on Groq's AI Chips

Satnam Singh, Groq

This presentation gave an overview of the special purpose hardware made by Groq for accelerating machine learning inference. The Groq architecture makes essential use of a highly distributed SRAM-based memory architecture to help implement low latency high throughput large language models (LLMs) like Llama3 70B and the various Mixtral models. High performance is achieved by keeping a central data structure in these LLMs, the KV-cache (a key-value cache) entirely resident in SRAM, distributed across many chips in nodes and multiple racks. A fundamental aspect of the Groq architecture is its deterministic behaviour, and distributed SRAM is a key enabling component. This allows for predictable performance and makes it easier to compose large systems spanning multiple racks with high performance.

List of Participants

- Jeremy Gibbons, Oxford University (co-organizer)
- Oleg Kiselyov, Tohoku University (co-organizer)
- Martin Elsman, University of Copenhagen
- Mattan Erez, University of Texas at Austin
- Juuso Haavisto, University of Oxford
- Gabriele Keller, Utrecht University
- Guido Martinez, Microsoft
- Cosmin Oancea, University of Copenhagen
- Adam Paszke, Google DeepMind
- Erez Petrank, Technion CS
- Marc Pouzet, École normale supérieure
- Gabriel Radanne, INRIA Lyon
- Tahina Ramananandro, Microsoft Research
- Tiark Rompf, Purdue University
- Andreas Rossberg, Independent
- Ponnuswamy (Saday) Sadayappan, University of Utah
- Sven-Bodo Scholz, Radboud University
- Amir Shaikhha, University of Edinburgh
- Mary Sheeran, Chalmers University of Technology
- Satnam Singh, Groq
- George Stelle, Los Alamos National Laboratory
- Jeffrey Vetter, Oak Ridge National Laboratory

Meeting Schedule

May 7 (Tuesday)

Theme: introductions, emerging technology

- Self-introductions (2-5 mins)
- Guest lecture: Gary Grider (online)
- Jeffrey Vetter
- Adam Paszke
- Mattan Erez

May 8 (Wednesday)

Theme: arrays/tensors, memory layout

- Satnam Singh: on the focus of the seminar
- Mattan Erez (continued)
- Ponnuswamy Sadayappan
- Cosmin Oancea
- Sven-Bodo Scholz
- Amir Shaikhha
- Gabriel Radanne
- Tahina Ramananananandra

May 9 (Thursday)

Theme: asynchrony/concurrency, excursion

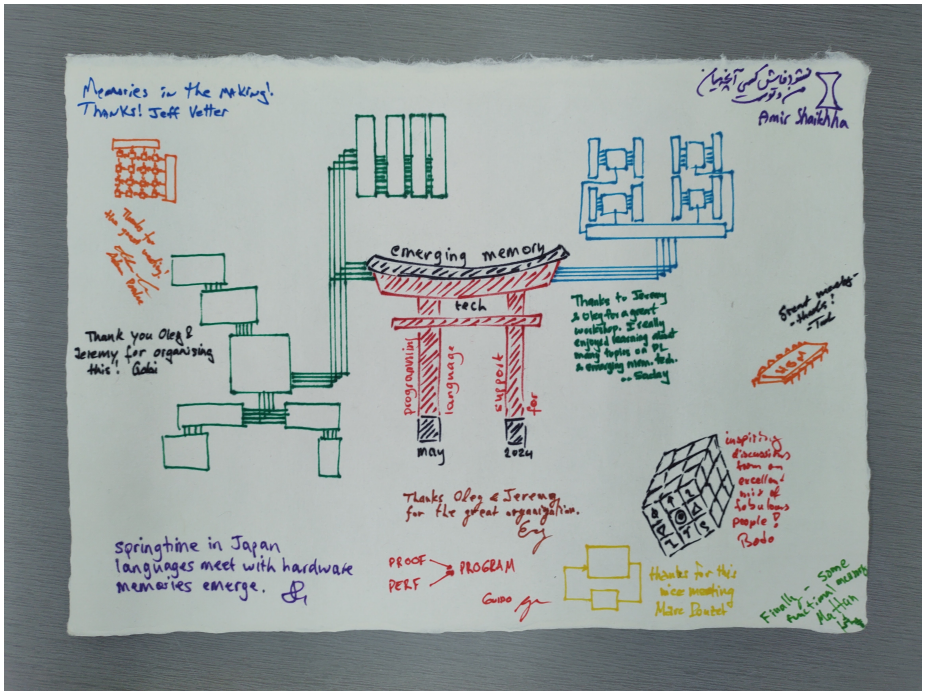
- Erez Petrank
- Guido Martinez
- Juuso Haavisto
- Marc Pouzet
- Excursion and banquet

May 10 (Friday)

Theme: arrays/tensors, Conclusions

- Martin Elsman
- Gabriele Keller
- Satnam Singh
- Closing discussion

Group photo and yosegaki



Summary of Discussions

Mostly as a conclusion of the extensive and extremely detailed talk by Mattan Erez, here is what a programming language could/should provide to efficiently use modern memory:

- specifying memory allocation and placement;
- expressing (temporal) locality;
- exposing concurrency and parallelism.

The programmer must also be aware that modern hardware does a lot behind the scenes (scheduling memory accesses, balancing memory channels, prefetching) which the programmer should not attempt to micro-manage. Cache oblivious algorithms are a poster example of dealing with memory "at the right level".

Identified Issues and Future Directions

A discussion is productive when there are more questions than answers, some say. Here are some of the questions left for future seminars.

- "I would like to hear about the verification aspect of the problems (e.g., writing fully verified code for tensor/array programming systems)."
- "I'd be particularly interested to hear more about the PL techniques useful for modeling asynchrony."
- "I would be particularly interested to hear about work on expressing and controlling data layout and perhaps location. This seems to be linked to the problem of how to optimise networks of tensor contractions. I guess the optimisation has to be with respect to a system description, including the arrangement of memory."
- Language integration and metaprogramming for data layout abstraction
- Similarities/differences in optimizing array/tensor computations in functional versus imperative languages
- Ways to analyse/express access patterns in a scalable yet still useful way
- Future memories and interconnect
- Fine-grained asynchrony for tolerating memory latencies
- Behavior/micro-architecture of TPUS and compute-in-memory architectures (what do they do, and how fast)
- Experiences with different intermediate representations for describing computations on large data

Recommend Reading

Streams:

- John Feo, David C. Cann, and Rodney R. Oldehoeft. A report on the Sisal language project. *Journal of Parallel and Distributed Computing*, 10(4):349–366, 1990.
- Marc Pouzet. Lucid Synchrone. `https://www.di.ens.fr/~pouzet/lucid-synchrone/`, 2006.
- Scott Kovach, Praneeth Kolichala, Tiancheng Gu, and Fredrik Kjolstad. Indexed streams: A formal intermediate representation for fused contraction programs. *PACMPL*, 7(PLDI), 2023.

Flattening:

- Guy E. Blelloch. NESL: A nested data-parallel language. Technical Report CMU-CS-92-103, Carnegie Mellon University, 1992. `https://www.cs.cmu.edu/~guyb/papers/Nesl2.0.pdf`.
- Gabriele Keller, Manuel M.T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Ben Lippmeier. Regular, shape-polymorphic, parallel arrays in Haskell. In *International Conference on Functional Programming (ICFP)*, page 261–272. Association for Computing Machinery, 2010.
- Lars Bergstrom, Matthew Fluet, Mike Rainey, John Reppy, Stephen Rosen, and Adam Shaw. Data-only flattening for nested data parallelism. In *Principles and Practice of Parallel Programming (PPoPP)*, page 81–92. Association for Computing Machinery, 2013.

Rank polymorphism:

- Justin Slepak, Olin Shivers, and Panagiotis Manolios. The semantics of rank polymorphism. cs.PL 1907.00509, arXiv, 2019. `https://arxiv.org/abs/1907.00509`.
- 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`.

Linear memory access descriptors:

- Yunheung Paek, Jay Hoeflinger, and David Padua. Efficient and precise array access analysis. *TOPLAS*, 24(1):65–109, 2002.
- Philip Munksgaard, Troels Henriksen, Ponnuswamy Sadayappan, and Cosmin E. Oancea. Memory optimizations in an array language. In Felix Wolf, Sameer Shende, Candace Culhane, Sadaf R. Alam, and Heike Jagode, editors, *International Conference for High Performance Computing, Networking, Storage and Analysis (SC22)*, pages 31:1–31:15. IEEE, 2022.

GPUs:

- Ethel Bardsley and Alastair F. Donaldson. Warps and atomics: Beyond barrier synchronization in the verification of GPU kernels. In Julia M. Badger and Kristin Yvonne Rozier, editors, *NASA Formal Methods*, pages 230–245. Springer International Publishing, 2014.
- Neil Henning. Vulkan subgroup tutorial. `https://www.khronos.org/blog/vulkan-subgroup-tutorial`, March 2018.
- Stephen Jones. How CUDA programming works. `https://www.youtube.com/watch?v=QQceTDjA4f4`, May 2022.
- NVidia. Parallel Thread Execution ISA Version 8.5. `https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#programming-model`, May 2024.

Non-volatile memory:

- Joy Arulraj, Andrew Pavlo, and Subramanya R. Dulloor. Let's talk about storage and recovery methods for non-volatile memory database systems. In *International Conference on Management of Data (SIGMOD)*, page 707–722. Association for Computing Machinery, 2015.
- Adrian Colyer. The Morning Paper: Let's talk about storage and recovery methods for non-volatile memory database systems. `https://blog.acolyer.org/2016/09/29/lets-talk-about-storage-and-recovery-methods-for-non-volatile-memory-database-systems/`, September 2016.

CXL:

- AMD. Introduction to CXL. `https://www.youtube.com/watch?v=k0m3N8JIZZ8`, January 2023.
- Mark Orthodoxou. Rambus: Transforming the data center with CXL. `https://www.youtube.com/watch?v=u9Ca2fBtP18`, August 2022.

References

[1] JC-42.2. High-bandwidth memory (HBM3) DRAM. `https://www.jedec.org/committees/jc-42`, January 2023.

[2] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, et al. Indatacenter performance analysis of a tensor processing unit. In *Proc. 44th International Symposium on Computer Architecture (ISCA)*, June 2017.

[3] Shriram Krishnamurthi. Organizing a workshop. `https://cs.brown.edu/~sk/Memos/Organizing-a-Workshop/`, July 2019.

[4] Vladimir Marjanovic, José Gracia, and Colin W. Glass. Performance modeling of the HPCG benchmark. In Stephen A. Jarvis, Steven A. Wright, and Simon D. Hammond, editors, *High Performance Computing Systems: Performance Modeling, Benchmarking, and Simulation*, volume 8966 of *Lecture Notes in Computer Science*, pages 172–192. Springer, 2014.

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

[6] Intel Optane technology. https://web.archive.org/web/20170106102302/https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html, January 6, 2017.

[7] Andy Rudoff. Persistent memory programming. *USENIX ;login:*, 42(2):34–40, 2017. https://www.usenix.org/system/files/login/articles/login_summer17_07_rudoff.pdf.

[8] Tatiana Shpeisman and Chris Lattner. MLIR: Multi-level intermediate representation compiler infrastructure. https://llvm.org/devmtg/2019-04/slides/Keynote-ShpeismanLattner-MLIR.pdf, 2019.

[9] Intel demonstrates STT-MRAM for L4 cache. https://www.tomshardware.com/news/intel-demonstrates-stt-mram-for-l4-cache, December 2019.