Generator-Based Custom SoC Design For Numerical Data Analysis

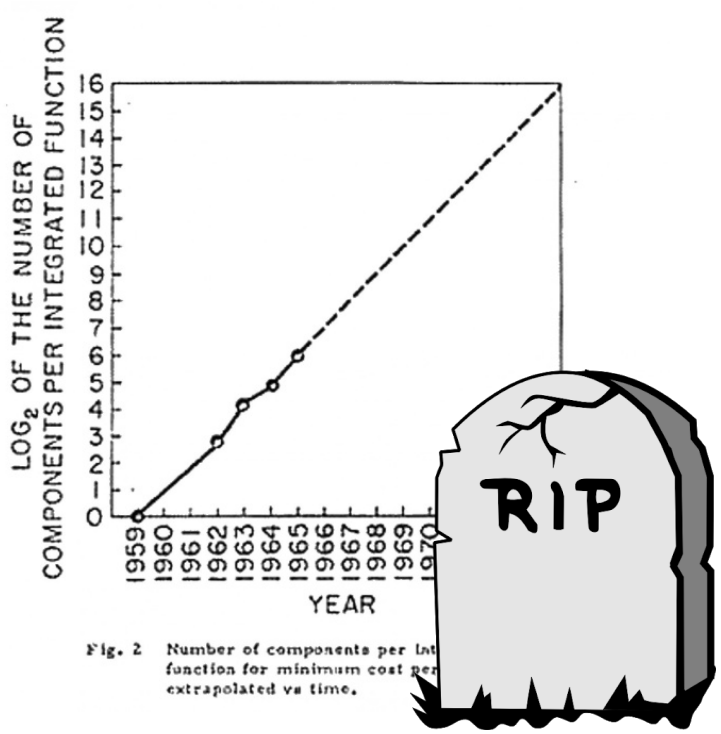
Alon Amid

Berkeley
UNIVERSITY OF CALIFORNIA

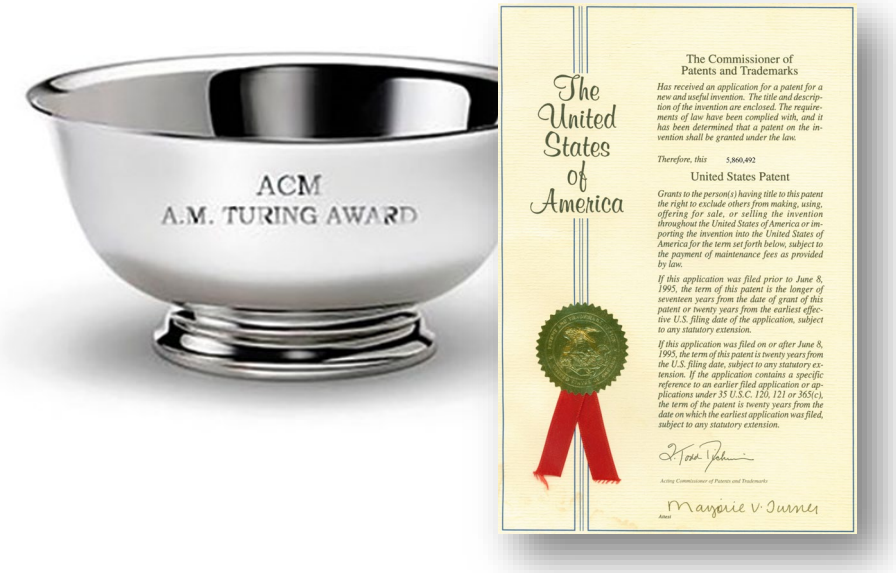
Berkeley
Architecture
Research

ADEPT

Motivation

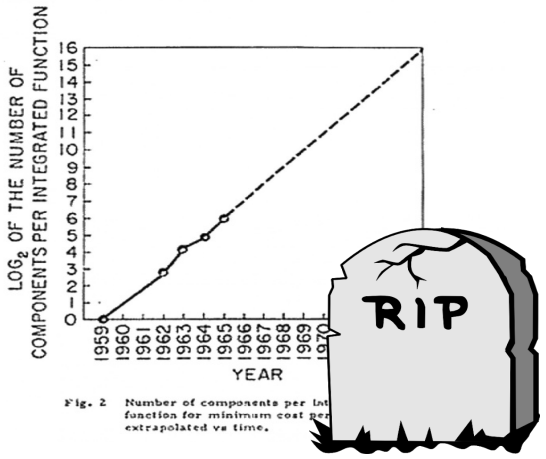


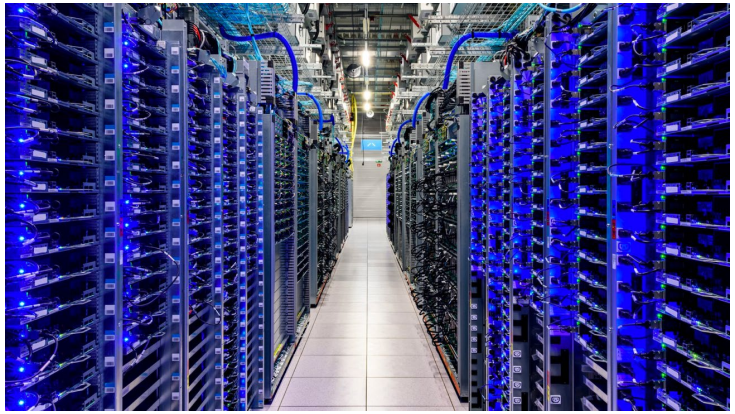
Moore's Law is dead



My Solution

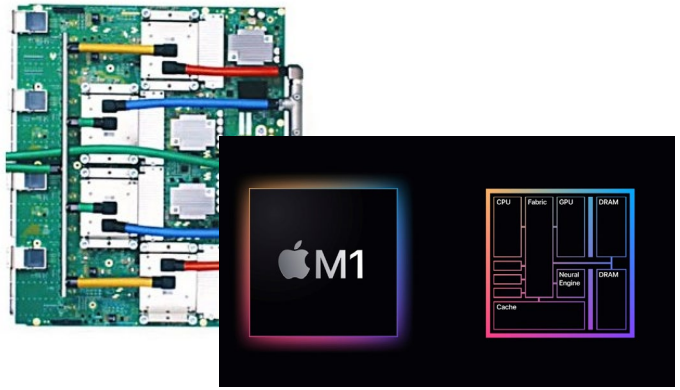
Motivation





Distribute
(Use more computers!)
Increase Scale

Many other solutions:
New devices,
Quantum computing,
….



Heterogenous/Custom Hardware
(Use more specialized computers!)
Increase Efficiency

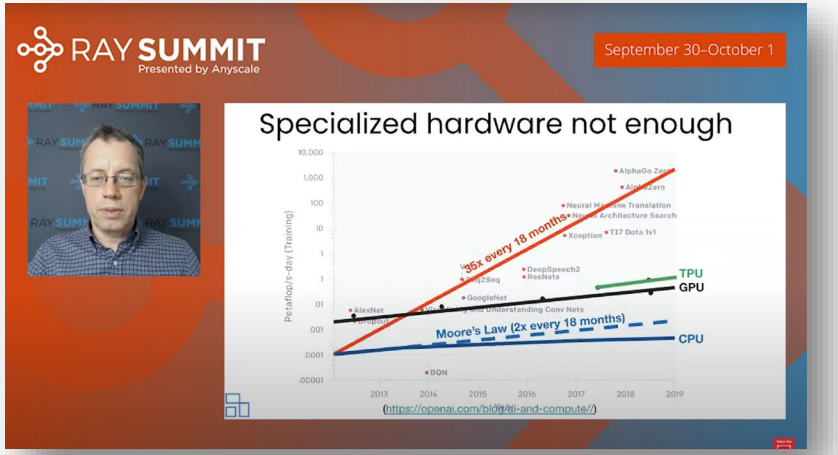
amplab riselab UC Berkeley

ASPIRE ADEPT

We Need All!

Good:

Distributed

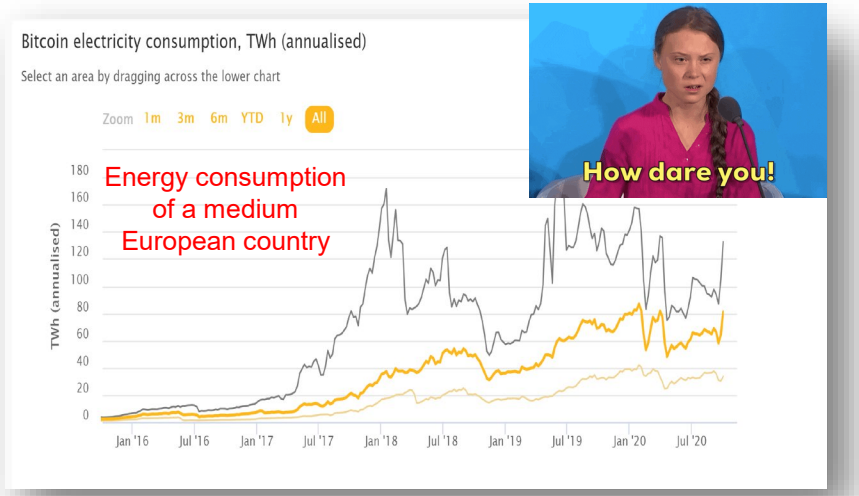


Unlimited Parallel Compute+Memory

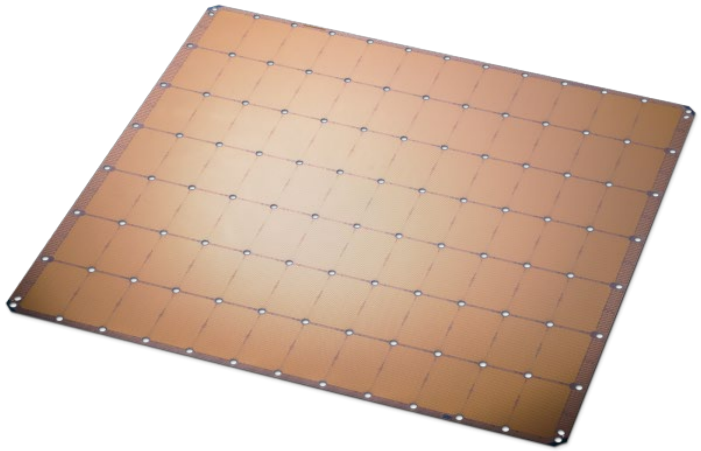
Specialized/Custom

Energy Efficiency + On-Chip Bandwidth

Bad:



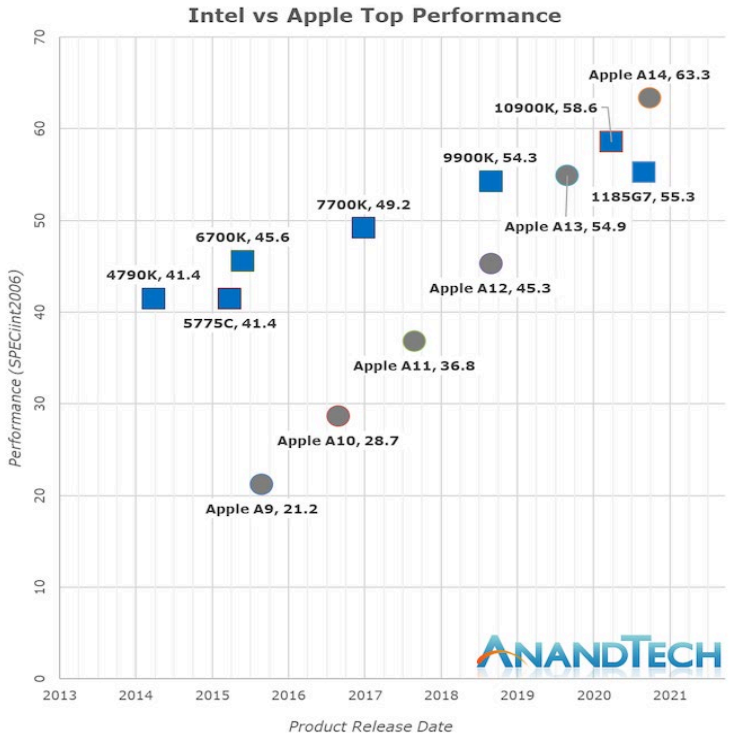
Unlimited Energy Cost



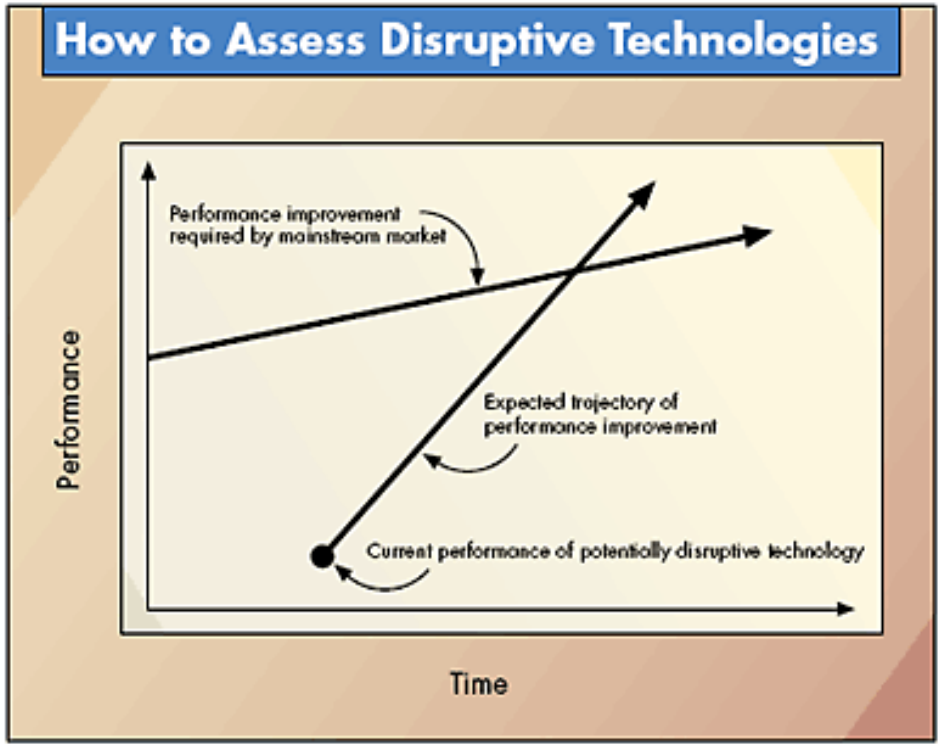
Limited Area / Off-Chip Bandwidth

The Integrated SoC

- The custom SoC increasingly adopted in “general purpose” computing
  - Behavior of a disruptive technology, as characterized by “The Innovator’s Dilemma”



https://www.anandtech.com/show/16226/apple-silicon-m1-a14-deep-dive/4



Innovator’s Dilemma, Clayton Christensen

*Idea discussed separately by Tim Millet (VP, Platform Architecture, Apple) and James Allworth (https://jamesallworth.medium.com/intels-disruption-is-now-complete-d4fa771f0f2c)

This Talk

- How to build custom SoCs ?
  - Open-source generators
  - Design flows
  - Chipyard
- Customizing an SoC for numerical data analysis applications
  - SoC customization
  - Flexibility: From deep learning to traditional linear algebra
  - Software stack
  - Hardware/Software co-design

Trends in Open Source Hardware

- Organization/Specifications: RISC-V, CHIPS Alliance, OpenHW
- Community: LowRISC, FOSSi
- Academia: PULP Platform, OpenPiton, ESP
- Government: DARPA POSH
- Industry: WD SWERVE, NVIDIA NVDLA
- Tools: Verilator, Yosys, OpenRoad
- Fabrication: Skywater 130nm

RISC-V

CHIPS ALLIANCE

FOSSi Foundation

DARPA launches POSH project for open source hardware blocks

Jul 26, 2018 — by Eric Brown — 1151 views

Please share:

DARPA

DARPA announced the first grants for its $1.5 billion Electronic Resurgence Initiative for accelerating chip development. More than $35 million went to a "Posh Open Source Hardware" project for developing and verifying hardware IP.

Western Digital's RISC-V "SweRV" Core Design Released For Free

14 Comments

+ Add A Comment

by Anton Shilov on February 15, 2019 11:30 AM EST

Posted in Storage CPUs SSDs Western Digital RISC-V

OpenHW Group Created and Announces CORE-V Family of Open-source Cores for Use in ...ne Production SoCs

...Executive Director of the RISC-V Foundation, leads ...oration, ecosystem development and open-source

GROUP OPENHW PROVEN PROCESSOR IP

OpenHW Group →

Jun 06, 2019, 04:00 ET

MICROPROCESSOR report

Insightful Analysis of Processor Technologies

Nvidia Shares Its Deep Learning

Xavier Neural-Network Accelerator Now Available as Open Source

March 26, 2018

PDF Version

By Mike Demler

VERILATOR

Building An Open Source RISC-V System

Have you heard of this Free and Open RISC-V thing? It should be so easy to build real systems now

Cool! I want to build an Open-Source custom RISC-V SoC. What do I need to do?

I think I heard of some stuff from Berkeley (Rocketchip? Chisel?), also OpenPiton, and PULP

Building An Open Source RISC-V System

- Processor core IP
- Supporting system IP (memory system, peripherals, buses, etc.)
- Integrate custom blocks
- Write appropriate software
- Verify using bare-metal simulation
- Validate full-system
- Physical design
- Test environment
- Fabrication

Hardware Generators

Instead of writing Verilog instances

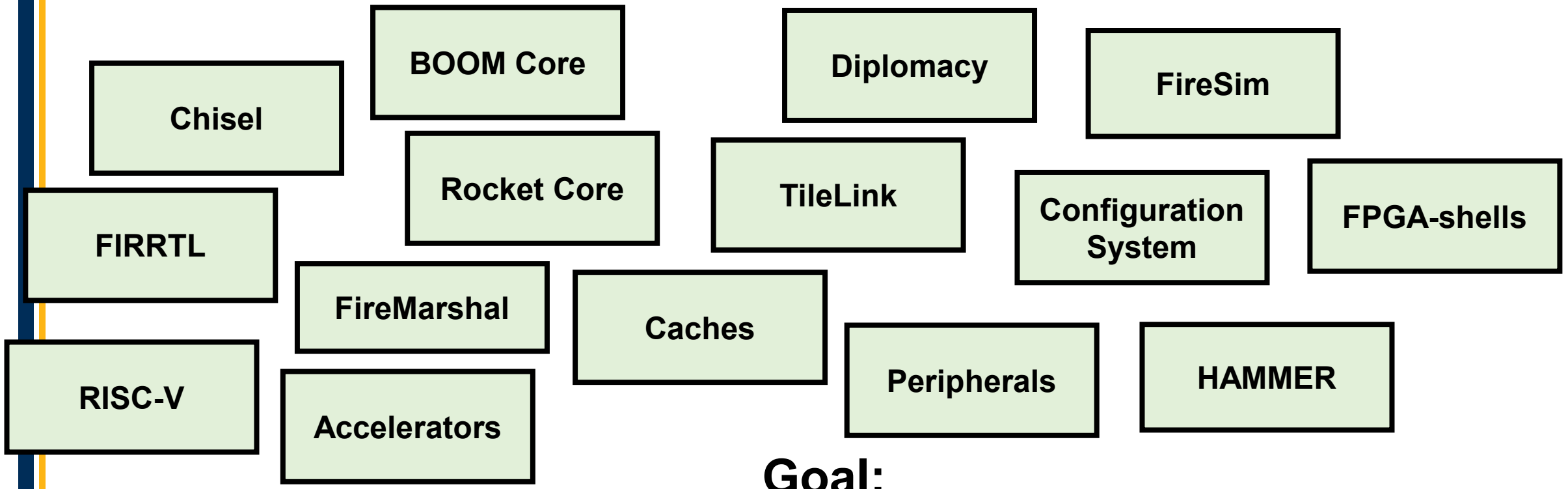
```
module MeshPE
  #(parameter INPUT_BITWIDTH, OUTPUT_BITWIDTH)
  (
    input                                clock,
    input                                reset,
    input signed [OUTPUT_BITWIDTH-1:0]       in_a,
    input signed [OUTPUT_BITWIDTH-1:0]       in_b,
    input                                in_control_dataflow,
    input                                in_valid,
    output reg signed [OUTPUT_BITWIDTH-1:0] out_a,
    output reg signed [OUTPUT_BITWIDTH-1:0] out_c,
    output reg signed [OUTPUT_BITWIDTH-1:0] out_b,
    output reg                           out_control_dataflow,
    output reg                           out_valid
  );

always @(posedge clock) begin
      if (reset)
       begin
        out_control_dataflow <= 1'b0;
        out_a <= {OUTPUT_BITWIDTH{1'b0}};
        out_valid <= 1'b0;
       end
      else
       begin
        out_control_dataflow <= in_control_dataflow;
        out_a <= in_a;
        out_valid <= in_valid;
       end
   end
```

Write a program that generates Verilog

```
class PE[T <: Data](inputType: T, outputType: T,
                    accType: T, df: Dataflow.Value,
                    latency: Int,
                    max_simultaneous_matmuls: Int)
                   (implicit ev: Arithmetic[T]) extends Module {

  val io = IO(new Bundle {
    val in_a = Input(inputType)
    val in_b = Input(outputType)
    val in_d = Input(outputType)
    val out_a = Output(inputType)
    val out_b = Output(outputType)
    val out_c = Output(outputType)

    val in_control = Input(new PEControl(accType))
    val out_control = Output(new PEControl(accType))

    val in_id = Input(UInt(log2Up(max_simultaneous_matmuls).W))
    val out_id = Output(UInt(log2Up(max_simultaneous_matmuls).W))
})

  val cType = if (df == Dataflow.WS) inputType else accType

  val a  = ShiftRegister(io.in_a, latency)
  val b  = ShiftRegister(io.in_b, latency)

  io.out_a := a
  io.out_control.dataflow := dataflow
  io.out_control.propagate := prop
  io.out_control.shift := shift
```


Building An Open Source RISC-V System

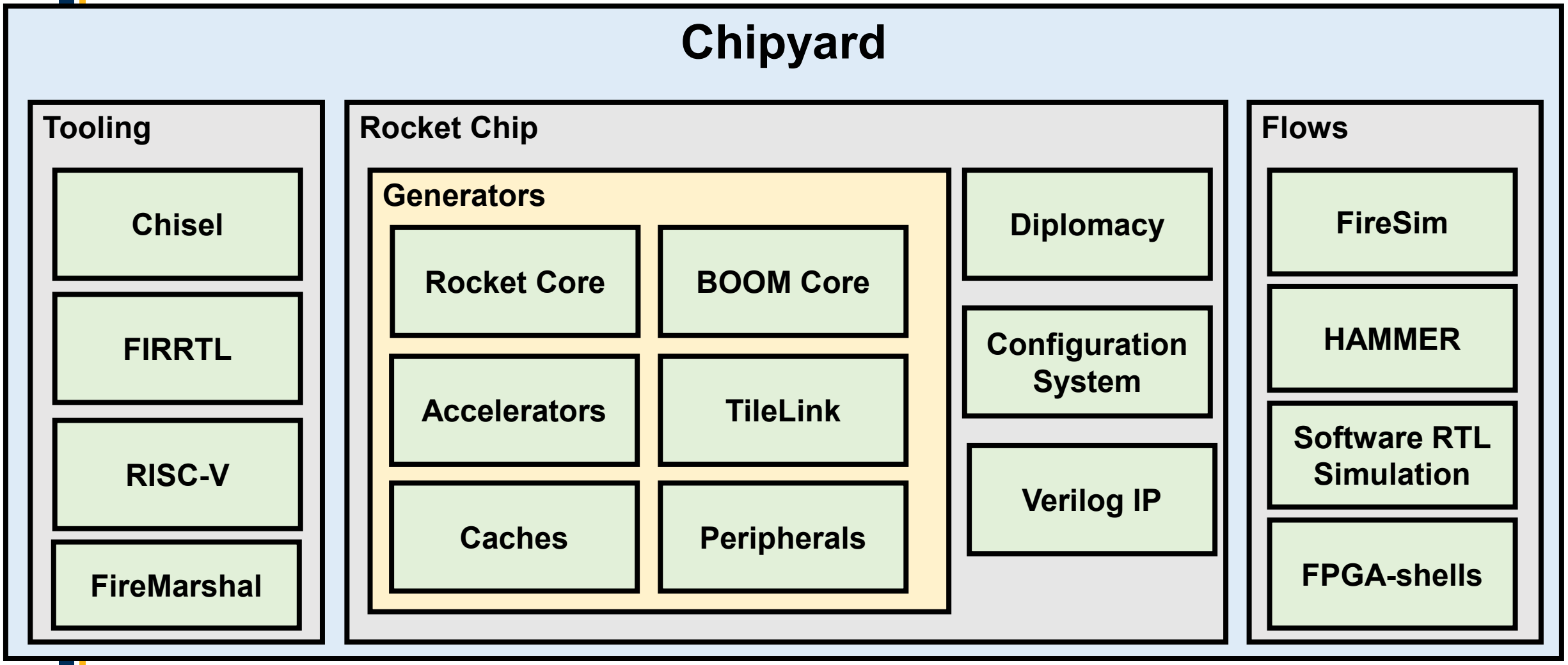
A lot of RISC-V & generator-related open source hardware projects out there



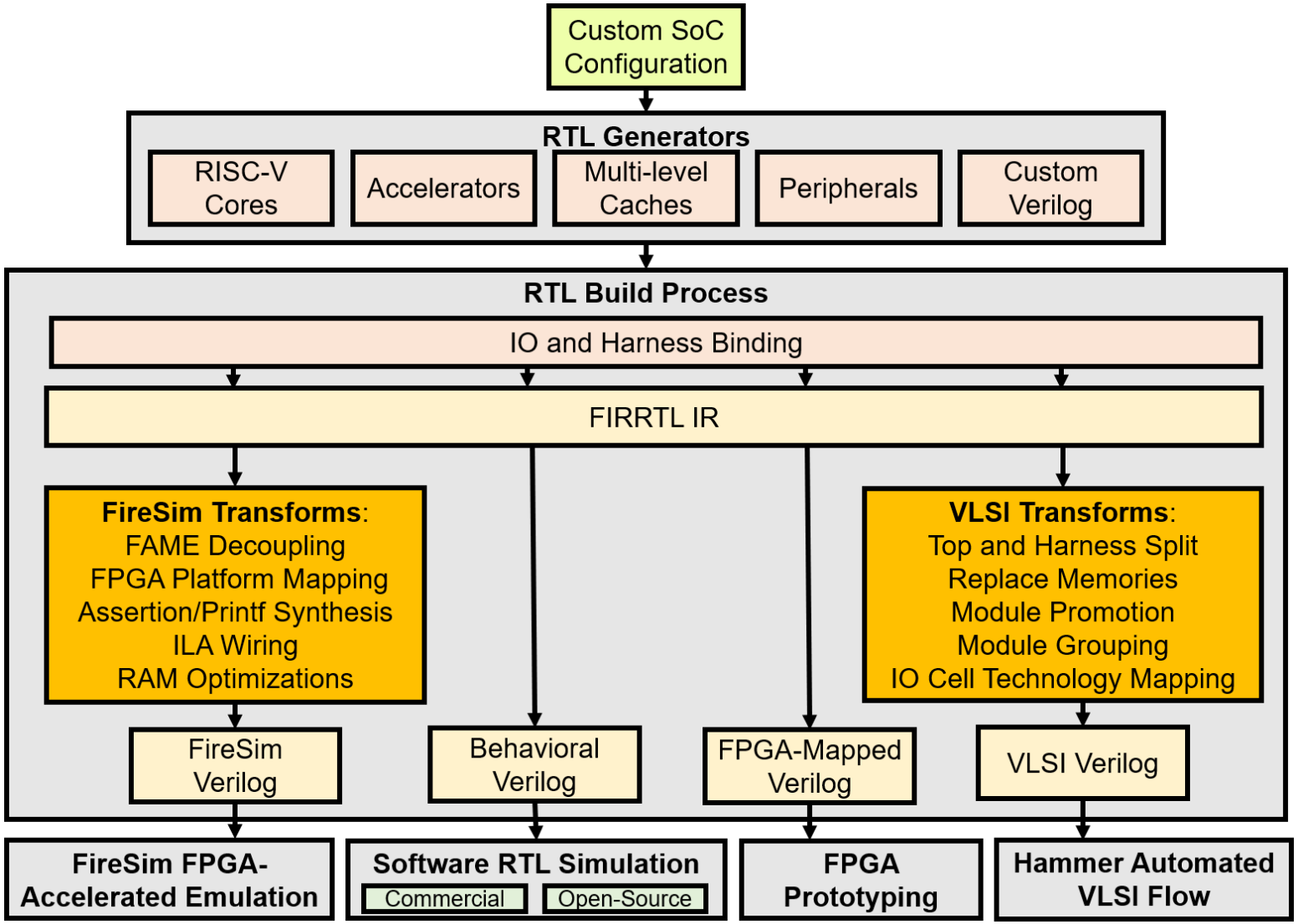
Goal:

Make it easy for small teams to
design, integrate, simulate, and **tape-out** a custom SoC

Chipyard

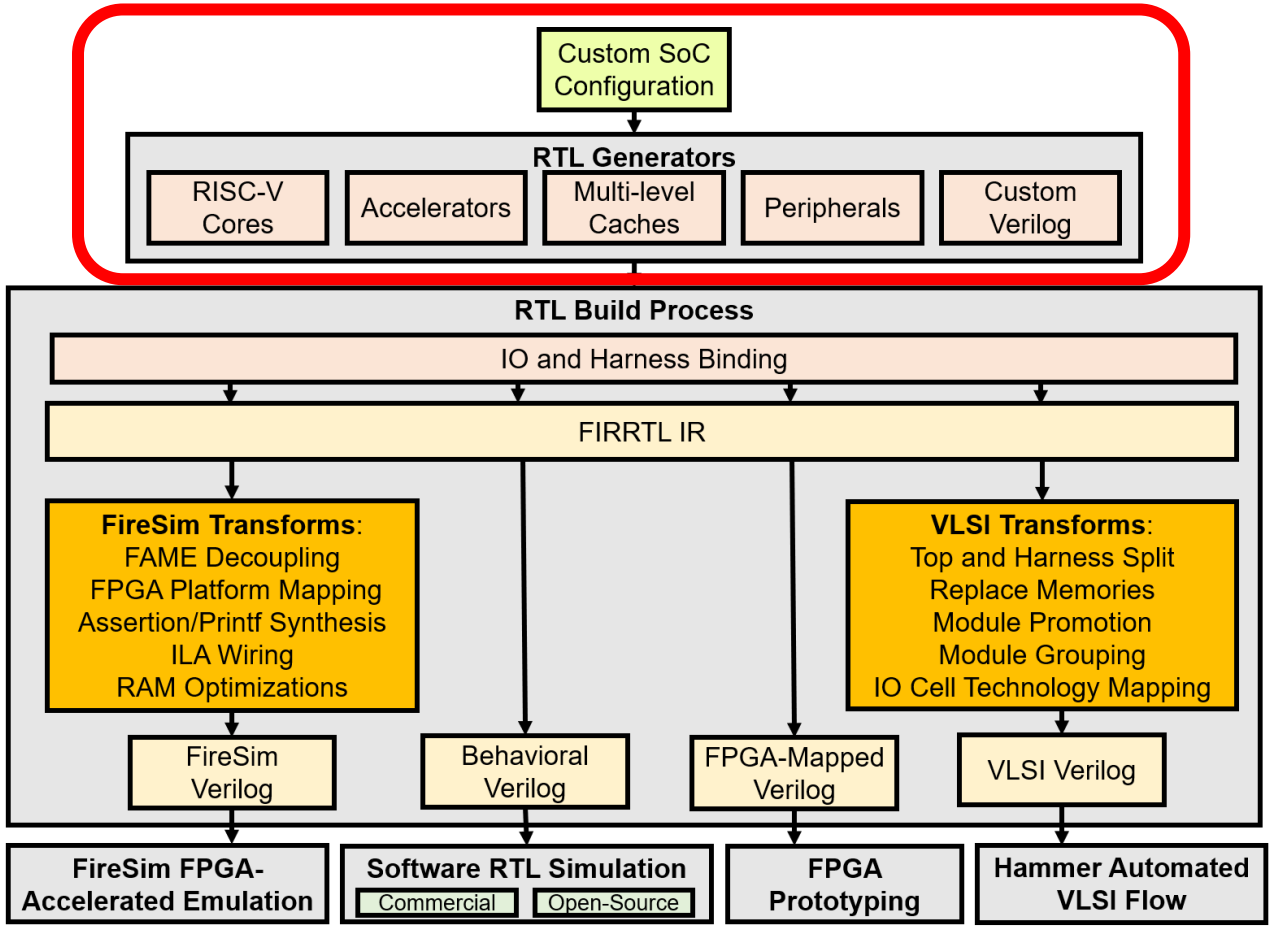

Chipyard
Tooling
Chisel
FIRRTL
RISC-V
FireMarshal
Rocket Chip
Generators
Rocket Core
BOOM Core
Accelerators
TileLink
Caches
Peripherals
Diplomacy
Configuration System
Verilog IP
Flows
FireSim
HAMMER
Software RTL Simulation
FPGA-shells

How is this integrated? Generators!



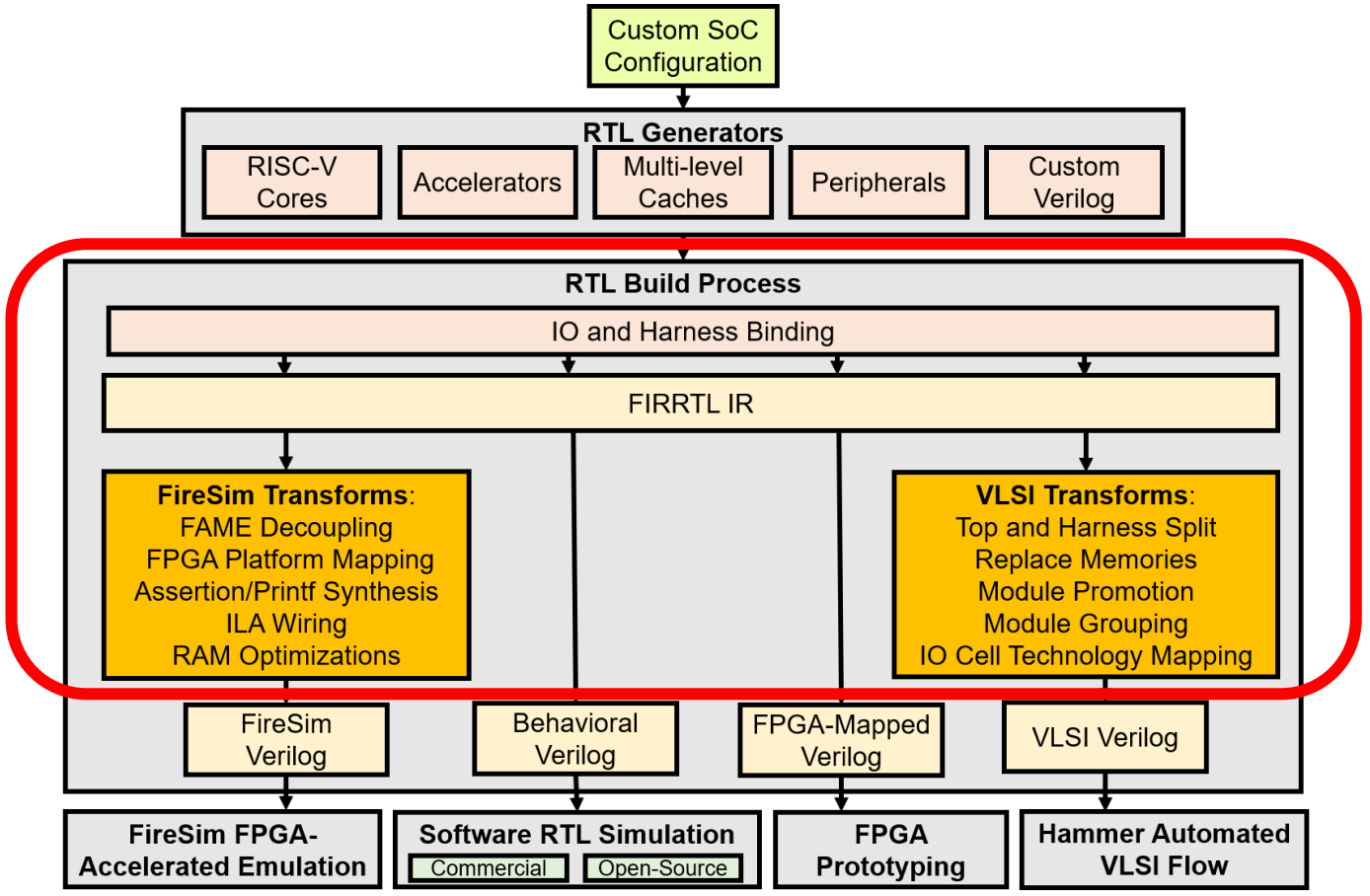
How is this integrated? Generators!

- Everything starts from a generator configuration
- Generators written in Chisel
- Generator SoC basic component libraries (enable integration)
  - Rocket Chip
  - Diplomacy
- Higher level generator libraries: BOOM, Inclusive Cache, SiFive Blocks, Accel.
- Generators can integrate third-party Verilog instance IP
- Generators lead from IP to design flows



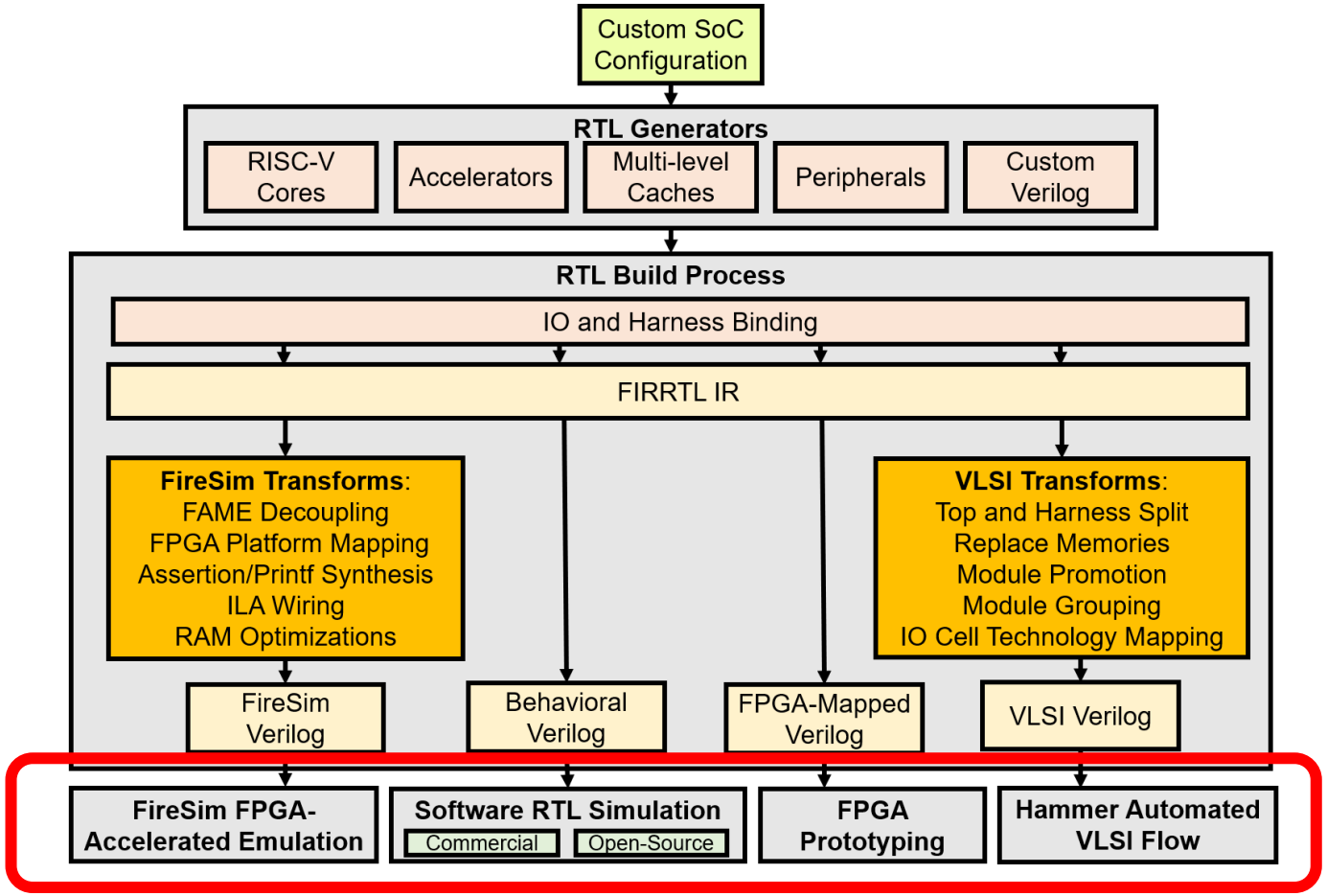
How is this integrated? Generators!

- Elaboration and Transformation
- Internals: FIRRTL – IR enables automated manipulation of the hardware description
- Externals: I/O and Harness Binders – pluggable interface functions enable automated targeting of different external interface requirements



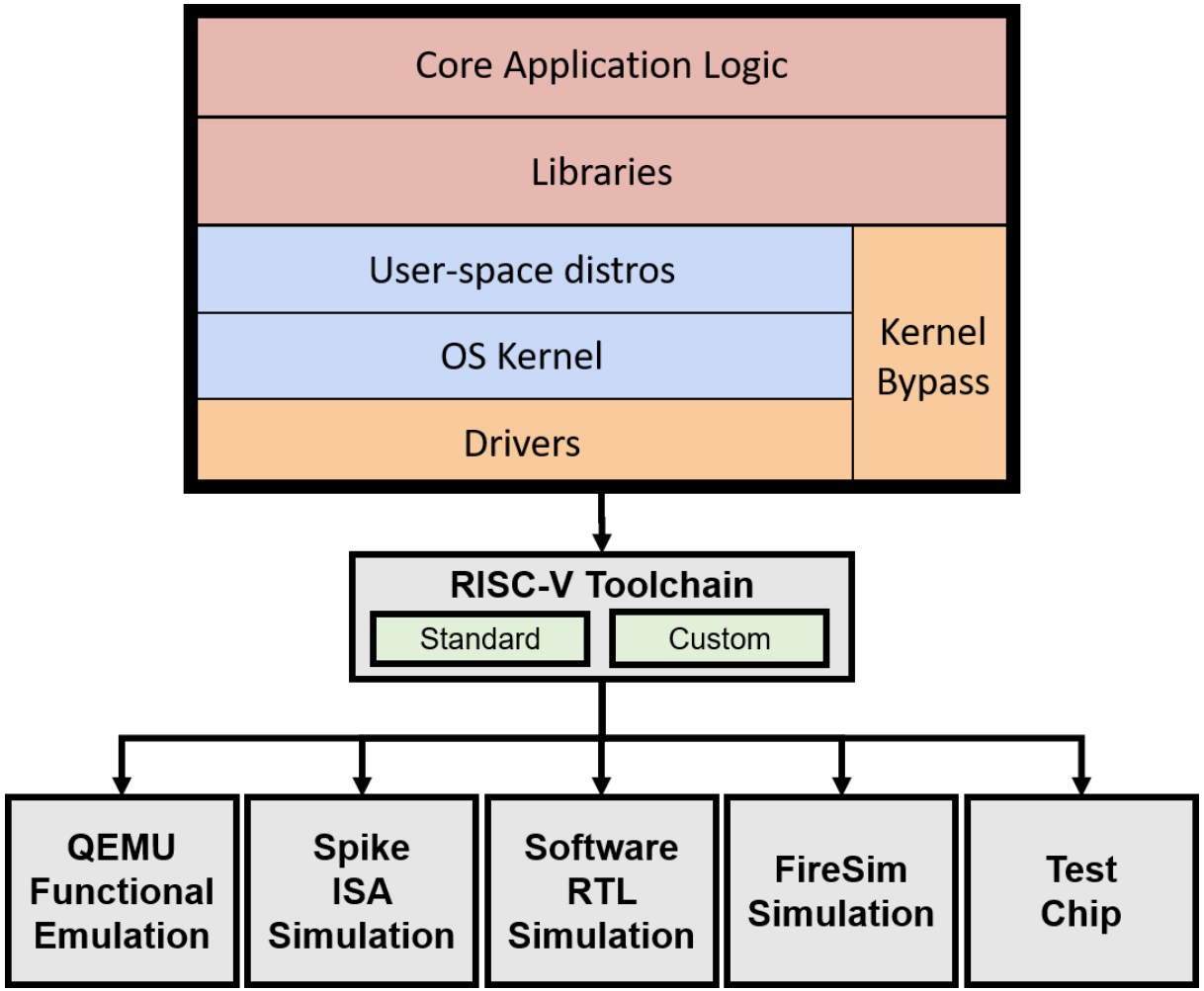
How is this integrated? Generators!

- Design flows
  - Software RTL Simulation
  - FPGA-Accelerated Emulation
  - FPGA Prototyping
  - VLSI Implementation
- Makefile based automation of transition between design flows
- Flow-specific collateral generation (harnesses, drivers, configuration and constraint files, etc.)



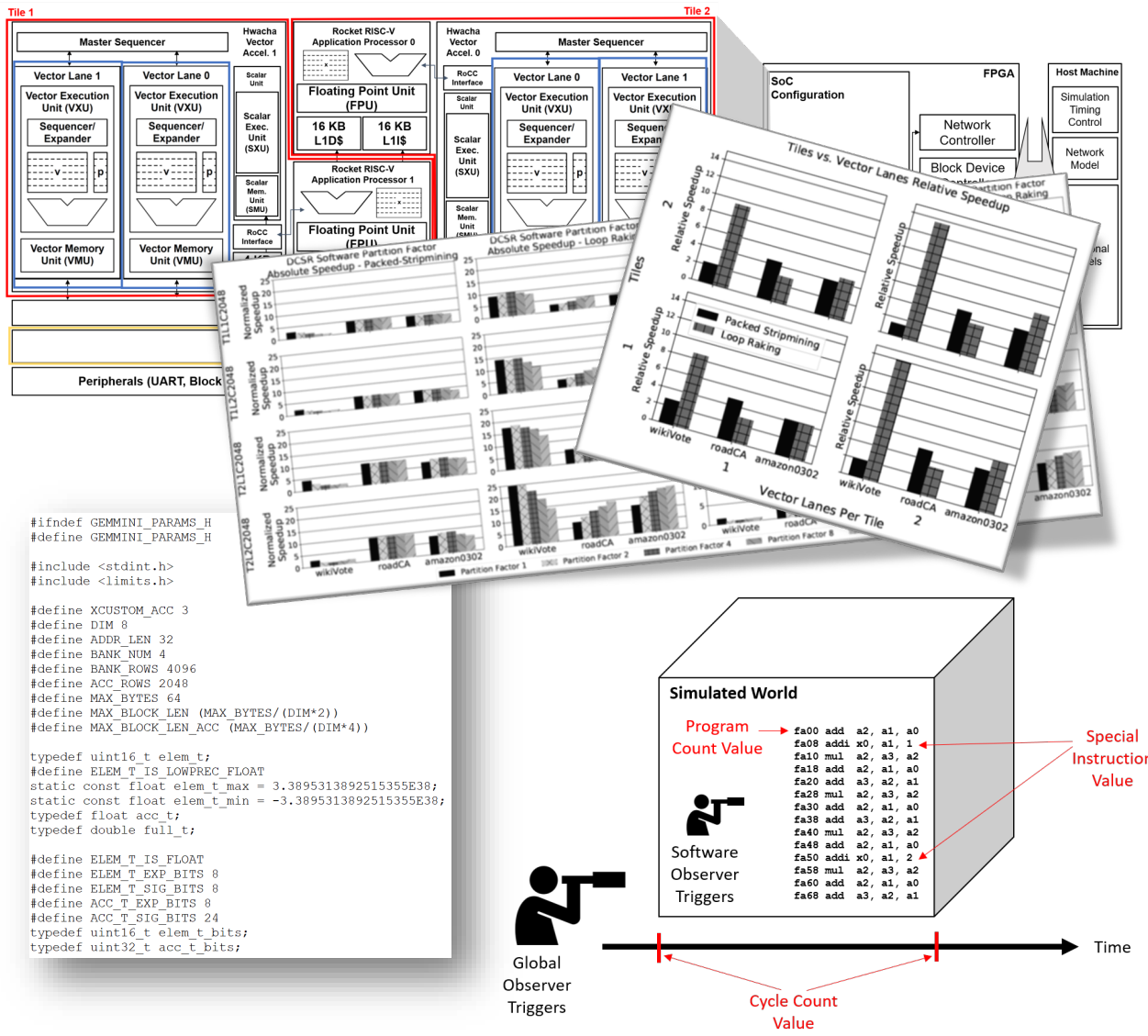

Software

- Hardware alone is not enough
- Custom SoCs require custom software
- Different platforms require different firmware
- Chipyard codifies custom software handling
  - Toolchains
  - Reproducible software generation and management flows using FireMarshal



HW/SW Co-Design

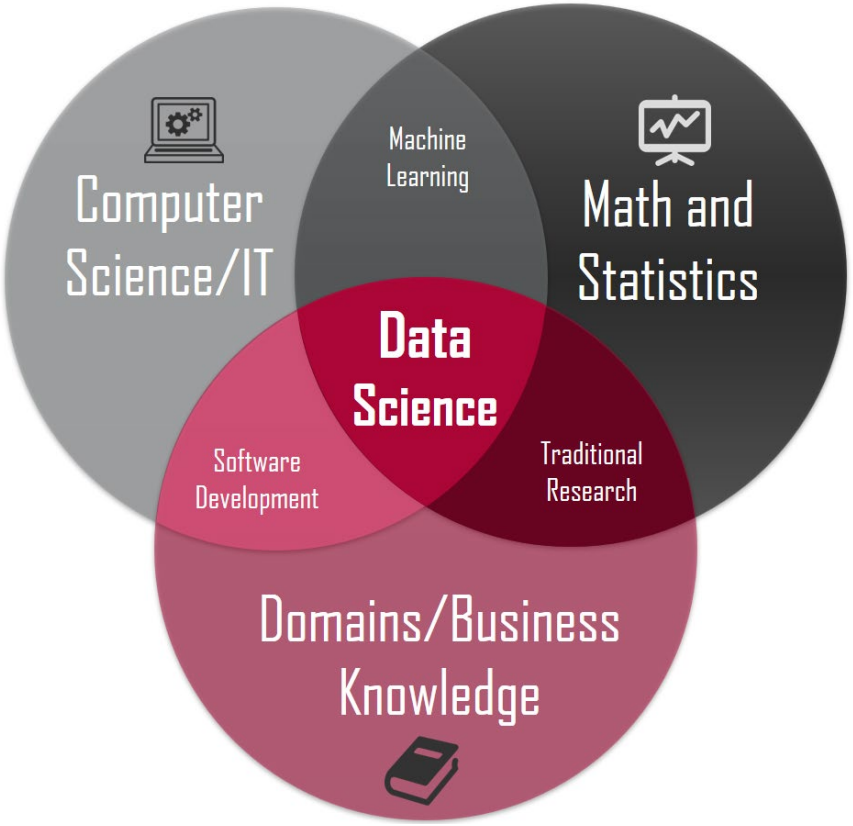
- Chipyard + FireSim enable new levels of HW/SW co-design
- Full-system design space exploration
  - Multi-core, multi-accelerator, multi-threaded SoC configurations
  - Full software stacks. Pre-silicon Linux, SPECInt with reference inputs
- Profiling and performance tuning
  - Auto-generated header files
  - Out-of-band performance counters
  - Hardware logging levels (triggers)



Customizing an SoC for Numerical Data Analysis Workloads

Numerical Data Analysis

- Sensors everywhere are generating data
  - Logs (both cloud and edge)
  - Cyber-physical sensors (gyro, microphones, cameras, LIDAR, RADAR, temperature, GPS)
- Data Science as an emerging paradigm, more than just DNNs:
  - Linear models and regressions
  - Dimensionality reduction
  - Data-mining / unsupervised Learning
  - Graph Analysis
  - Deep learning
- **Lots of dense linear algebra**



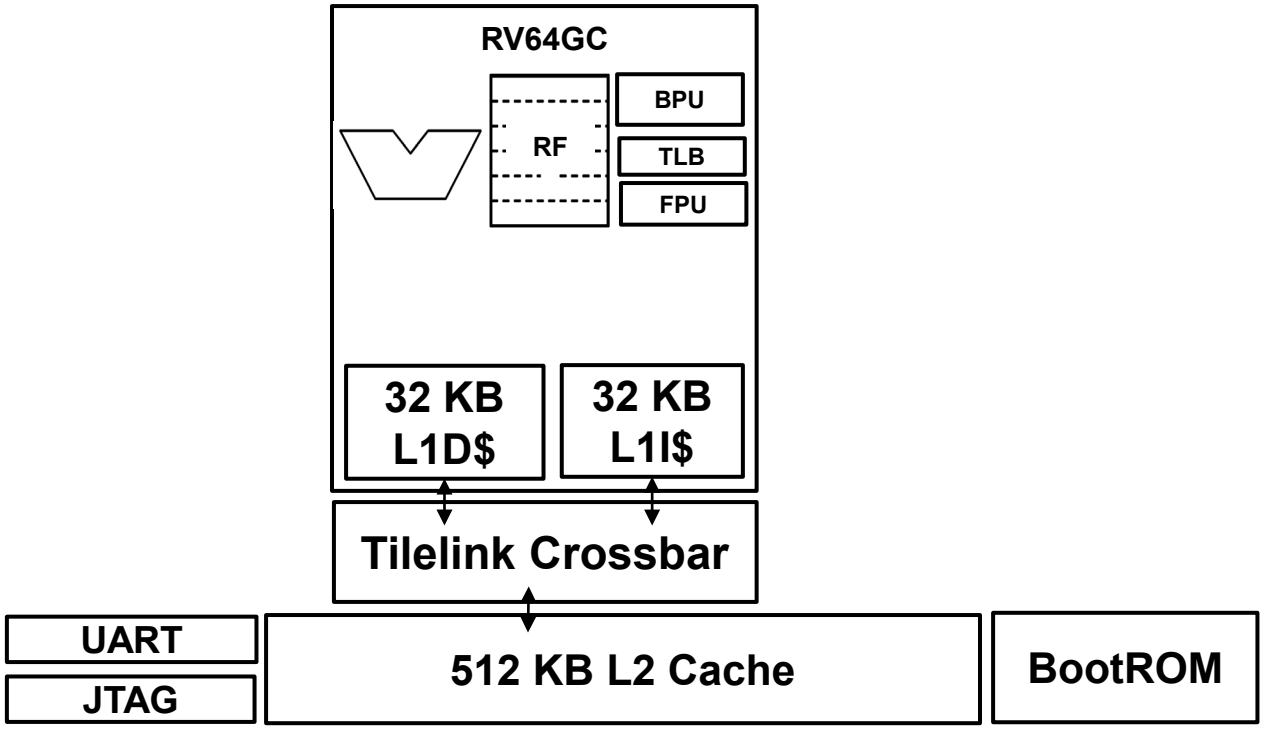
Conway's Van Diagram [1]

[1] https://towardsdatascience.com/introduction-to-statistics-e9d72d818745

SoC for Data Analysis Workloads

- Customize an SoC for numerical data analysis - start with a basic core and memory system

```
class DataSoC extends Config(
  new freechips.rocketchip.subsystem.
            WithInclusiveCache(nBanks=4)   ++
  new chipyard.config.AbstractConfig)
```

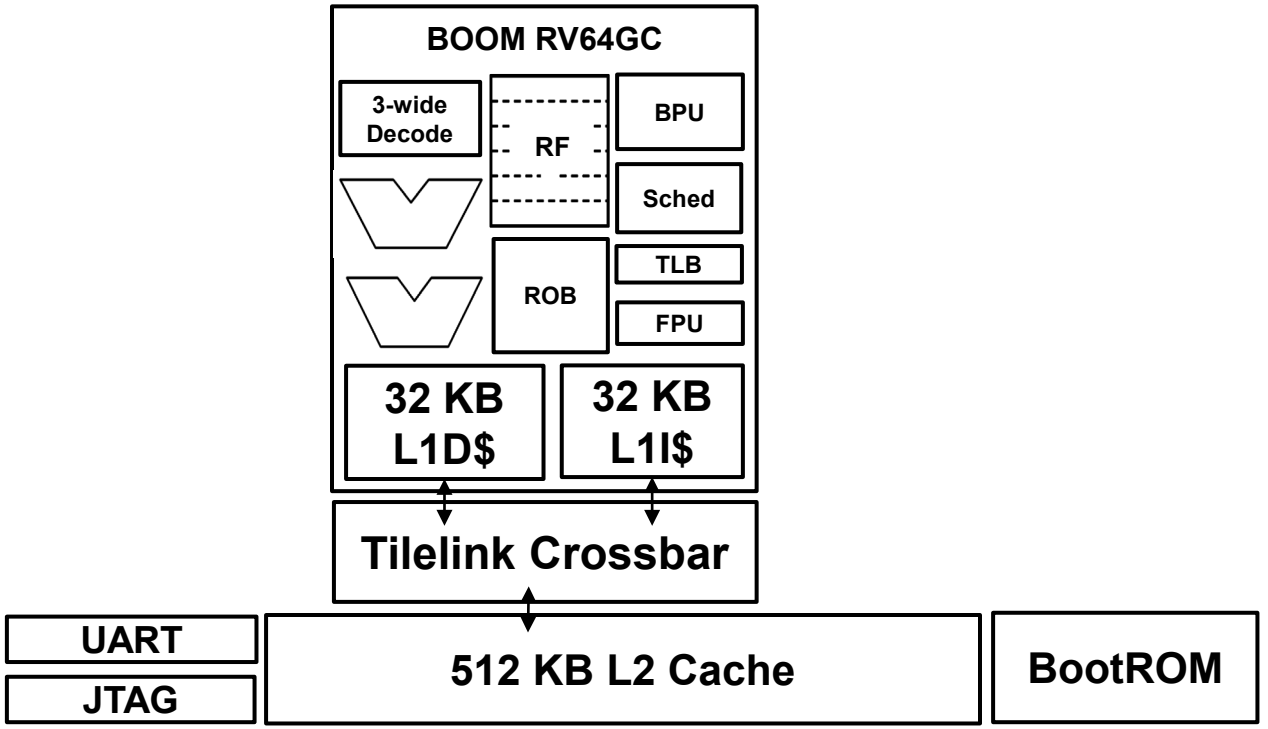



SoC for Data Analysis Workloads

Compute-intensive workloads require a high performance CPU

- 3-wide BOOM configuration, 12-stage out-of-order core

```
class DataSoC extends Config(
  new boom.common.WithNLargeBooms(1)        ++
  new freechips.rocketchip.subsystem.
            WithInclusiveCache(nBanks=4)    ++
  new chipyard.config.AbstractConfig)
```

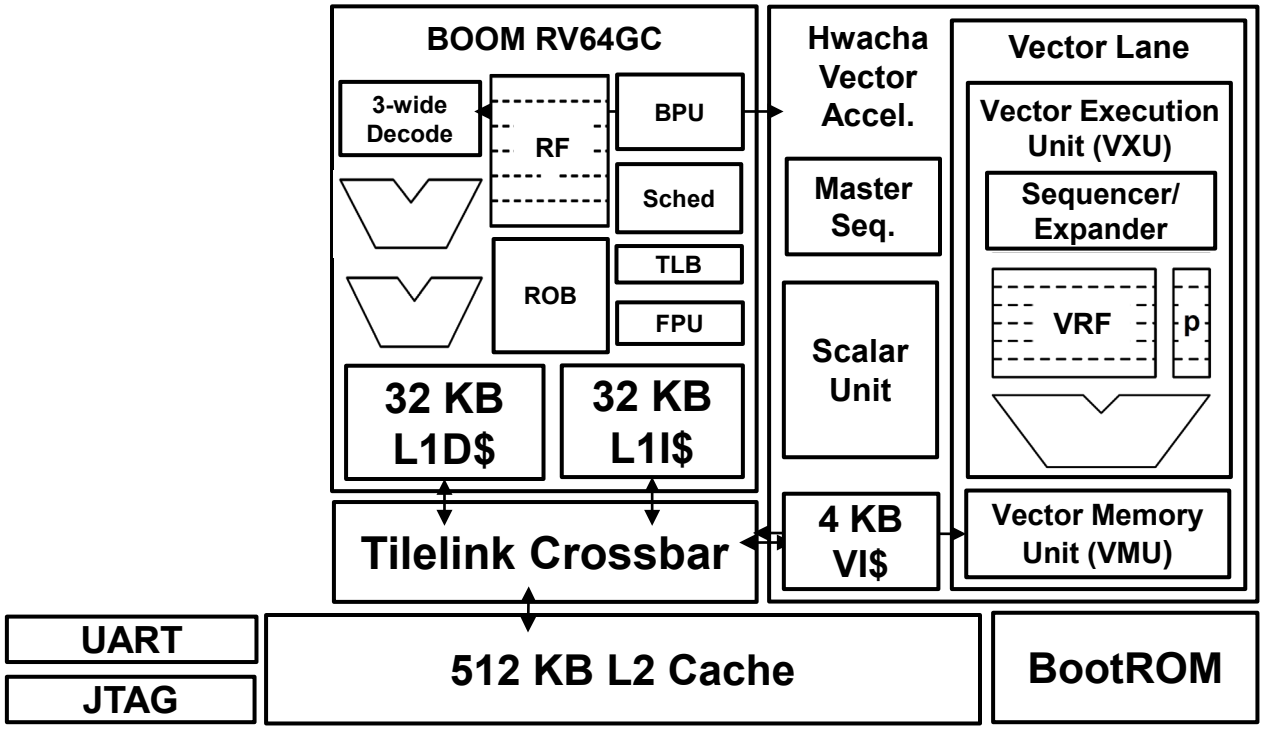


SoC for Data Analysis Workloads

Need to process element-wise vector operations

- Add a data-parallel vector unit
- Hwacha – **temporal** vector-fetch architecture

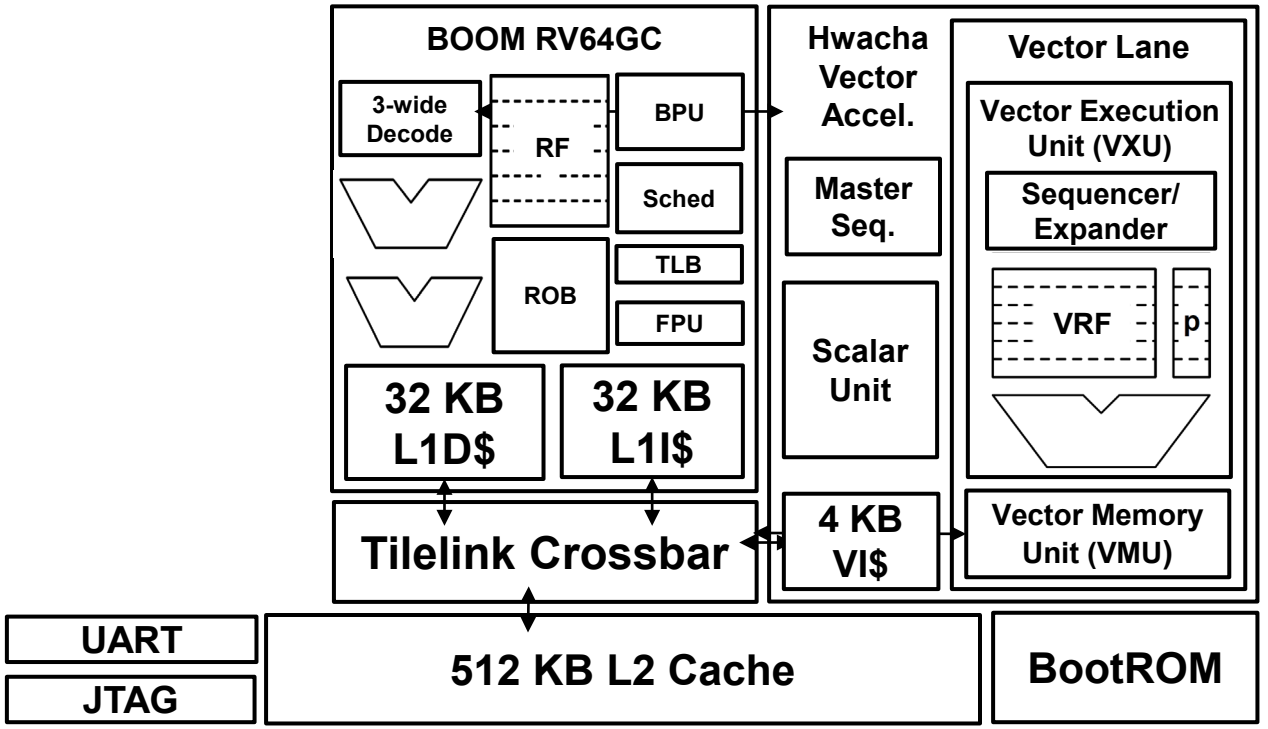
```
class DataSoC extends Config(
  new hwacha.DefaultHwachaConfig              ++
  new boom.common.WithNLargeBooms(1)          ++
  new freechips.rocketchip.subsystem.
            WithInclusiveCache(nBanks=4)      ++
  new chipyard.config.AbstractConfig)
```



SoC for Data Analysis Workloads

What about matrix operations?

```
class DataSoC extends Config(
  new hwacha.DefaultHwachaConfig              ++
  new boom.common.WithNLargeBooms(1)          ++
  new freechips.rocketchip.subsystem.
            WithInclusiveCache(nBanks=4)      ++
  new chipyard.config.AbstractConfig)
```



Customization

Customize (transitive verb) - Modify (something) to suit a particular individual or task.

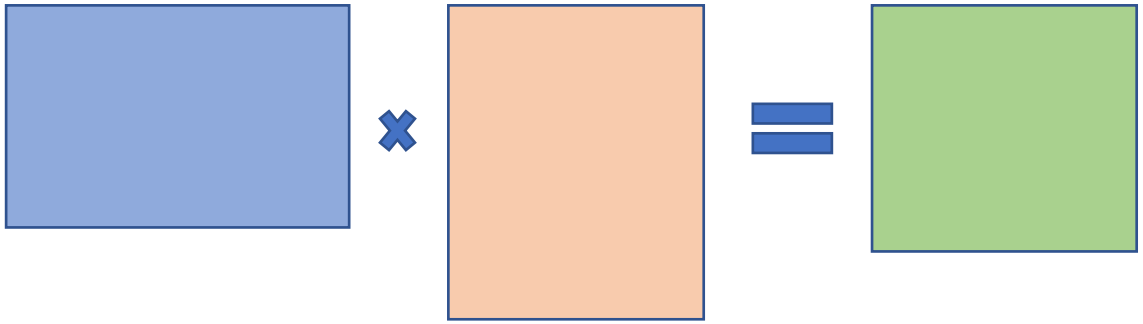
Oxford Dictionary

My interpretation:

- Don’t build from scratch
- Re-use existing system blocks
- For example, re-use a DNN accelerator as a matrix engine


DNN Accelerators

- Primary compute operations:
  - GEMM
  - GEMV
  - CONV
- Fused operations:
  - Pooling
  - Activation function (ReLU / Sigmoid)
- Data re-use
  - Scratchpad
  - Accumulators
- Numerics: 1-bit, Int8, Int16, FP16, Bfloat16, FP32



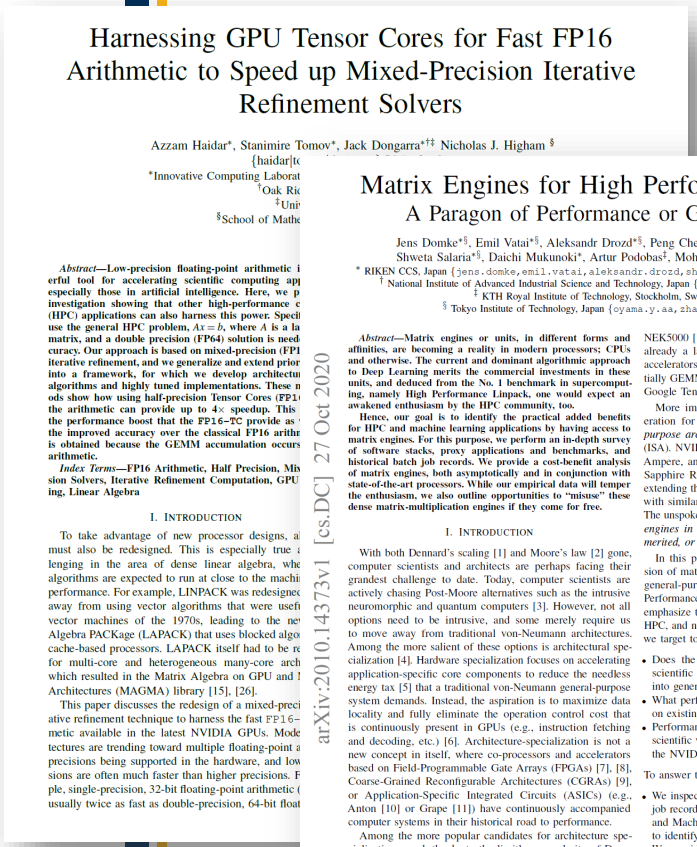
DNN Accelerators

- Primary compute operations:
  - GEMM
  - GEMV
  - CONV
- Fused operations:
  - Pooling
  - Activation function (ReLU / Sigmoid)
- Data re-use
  - Scratchpad
  - Accumulators
- Numerics: 1-bit, Int8, Int16, FP16, Bfloat16, FP32

Common dense linear algebra operations.
Not restricted to just DNNs!

Secondary-Use of DNN Accelerators

- This has been going on in the HPC community for a long time



Harnessing GPU Tensor Cores for Fast FP16 Arithmetic to Speed up Mixed-Precision Iterative Refinement Solvers

Azzam Haidar*, Stanimire Tomov*, Jack Dongarra*†‡ Nicholas J. Higham §

*Innovative Computing Laborat
†Oak Ri
‡Uni
§School of Math

Abstract—Low-precision floating-point arithmetic i
erful tool for accelerating scientific computing ap
especially those in artificial intelligence. Here, we p
investigation showing that other high-performance c
(HPC) applications can also harness this power. Speci
use the general HPC problem, $Ax = b$, where A is a la
matrix, and a double precision (FP64) solution is need
curacy. Our approach is based on mixed-precision (FP1
iterative refinement, and we generalize and extend prior
into a framework, for which we develop architectu
algorithms and highly tuned implementations. These n
ods show how using half-precision Tensor Cores (FP1
the arithmetic can provide up to 4× speedup. This
the performance boost that the FP16-TC provide as
the improved accuracy over the classical FP16 arithm
is obtained because the GEMM accumulation occurs
arithmetic.

Index Terms—FP16 Arithmetic, Half Precision, Mix
sion Solvers, Iterative Refinement Computation, GPU
ing, Linear Algebra

I. Introduction

To take advantage of new processor designs, a
must also be redesigned. This is especially true
lenging in the area of dense linear algebra, whe
algorithms are expected to run at close to the machi
performance. For example, LINPACK was redesigne
away from using vector algorithms that were usefu
vector machines of the 1970s, leading to the ne
Algebra PACKage (LAPACK) that uses blocked algo
cache-based processors. LAPACK itself had to be re
for multi-core and heterogeneous many-core arch
which resulted in the Matrix Algebra on GPU and
Architectures (MAGMA) library [15], [26].

This paper discusses the redesign of a mixed-preci
ative refinement technique to harness the fast FP16-
metic available in the latest NVIDIA GPUs. Mode
tectures are trending toward multiple floating-point a
precisions being supported in the hardware, and low
sions are often much faster than higher precisions. F
ple, single-precision, 32-bit floating-point arithmetic
usually twice as fast as double-precision, 64-bit float

Matrix Engines for High Performance Computing:
A Paragon of Performance or Grasping at Straws?

Jens Domke*§, Emil Vatai*§, Aleksandr Drozd*§, Peng Chen†, Yosuke Oyama§, Lingqi Zhang§, Shweta Salaria*§, Daichi Mukunoki*, Artur Podobas‡, Mohamed Wahib†*, Satoshi Matsuoka*§

* RIKEN CCS, Japan {jens.domke,emil.vatai,aleksandr.drozd,shweta.salaria,daichi.mukunoki}@riken.jp
† National Institute of Advanced Industrial Science and Technology, Japan {mohamed.attia,chin.hou}@aist.go.jp
‡ KTH Royal Institute of Technology, Stockholm, Sweden podobas@kth.se
§ Tokyo Institute of Technology, Japan {oyama.y.aa,zhang.l.ai}@m.titech.ac.jp

Abstract—Matrix engines or units, in different forms and affinities, are becoming a reality in modern processors; CPUs and otherwise. The current and dominant algorithmic approach to Deep Learning merits the commercial investments in these units, and deduced from the No. 1 benchmark in supercomputing, namely High Performance Linpack, one would expect an awakened enthusiasm by the HPC community, too.

Hence, our goal is to identify the practical added benefits for HPC and machine learning applications by having access to matrix engines. For this purpose, we perform an in-depth survey of software stacks, proxy applications and benchmarks, and historical batch job records. We provide a cost-benefit analysis of matrix engines, both asymptotically and in conjunction with state-of-the-art processors. While our empirical data will temper the enthusiasm, we also outline opportunities to "misuse" these dense matrix-multiplication engines if they come for free.

I. Introduction

With both Dennard's scaling [1] and Moore's law [2] gone, computer scientists and architects are perhaps facing their grandest challenge to date. Today, computer scientists are actively chasing Post-Moore alternatives such as the intrusive neuromorphic and quantum computers [3]. However, not all options need to be intrusive, and some merely require us to move away from traditional von-Neumann architectures. Among the more salient of these options is architectural specialization [4]. Hardware specialization focuses on accelerating application-specific core components to reduce the needless energy tax [5] that a traditional von-Neumann general-purpose system demands. Instead, the aspiration is to maximize data locality and fully eliminate the operation control cost that is continuously present in GPUs (e.g., instruction fetching and decoding, etc.) [6]. Architecture-specialization is not a new concept in itself, where co-processors and accelerators based on Field-Programmable Gate Arrays (FPGAs) [7], [8], Coarse-Grained Reconfigurable Architectures (CGRAs) [9], or Application-Specific Integrated Circuits (ASICs) (e.g., Anton [10] or Grape [11]) have continuously accompanied computer systems in their historical road to performance.

Among the more popular candidates for architecture specialization, much thanks to the limitless popularity of Deep-Learning [12], is to target General Matrix Multiplication (GEMM). Targeting GEMM is perhaps not entirely unmotivated: GEMM is often *claimed* to be the core compute-intensive component in many scientific applications spanning multiple domains, such as Computational Fluid Dynamics (e.g., NEK5000 [13]) or Deep Learning (DL) [14]. Today, there are already a large bulk of *application-specific* (primarily DL) accelerators that are based around systolic arrays [15] (essentially GEMM engines), such as Huawei's Ascend 910 [16] and Google Tensor Processing Units (TPUs) [17].

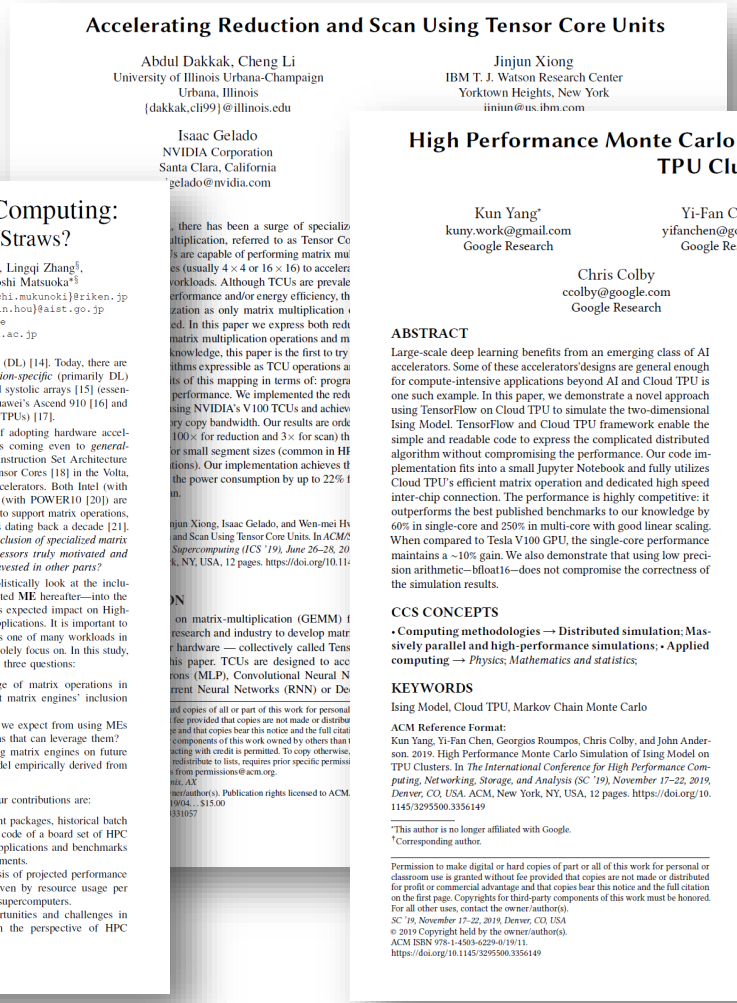
More importantly, the trend of adopting hardware acceleration for GEMM operations is coming even to *general-purpose architectures* and their Instruction Set Architecture (ISA). NVIDIA introduced the Tensor Cores [18] in the Volta, Ampere, and Turing series of accelerators. Both Intel (with Sapphire Rapids [19]) and IBM (with POWER10 [20]) are extending their SIMD-capabilities to support matrix operations, with similar proposals by authors dating back a decade [21]. The unspoken question is: *Is the inclusion of specialized matrix engines in general-purpose processors truly motivated and merited, or is the silicon better invested in other parts?*

In this paper, we aspire to holistically look at the inclusion of matrix engines—abbreviated **ME** hereafter—into the general-purpose processor and its expected impact on High-Performance Computing (HPC) applications. It is important to emphasize that we consider DL as one of many workloads in HPC, and not the application we solely focus on. In this study, we target to answer the following three questions:

- Does the occurrence and usage of matrix operations in scientific workloads truly merit matrix engines' inclusion into general-purpose ISAs?
- What performance benefits can we expect from using MEs on existing scientific applications that can leverage them?
- Performance projection of using matrix engines on future scientific workloads using a model empirically derived from the NVIDIA V100 GPUs.

To answer the above questions, our contributions are:

- We inspect software management packages, historical batch job records, profiles, and source code of a board set of HPC and Machine Learning proxy applications and benchmarks to identify dense matrix requirements.
- We provide a cost-benefit analysis of projected performance gains from matrix engines, driven by resource usage per domain in different production supercomputers.
- A detailed discussion of opportunities and challenges in adopting matrix engines, from the perspective of HPC workloads.



Accelerating Reduction and Scan Using Tensor Core Units

Abdul Dakkak, Cheng Li
University of Illinois Urbana-Champaign
Urbana, Illinois
{dakkak,cli99}@illinois.edu

Jinjun Xiong
IBM T. J. Watson Research Center
Yorktown Heights, New York
jinjun@us.ibm.com

Isaac Gelado
NVIDIA Corporation
Santa Clara, California
gelado@nvidia.com

High Performance Monte Carlo Simulation of Ising Model on TPU Clusters

Kun Yang*
kuny.work@gmail.com
Google Research

Yi-Fan Chen†
yifanchen@google.com
Google Research

Georgios Roumpos
roumposg@google.com
Google Research

Chris Colby
ccolby@google.com
Google Research

John Anderson
janders@google.com
Google Research

ABSTRACT

Large-scale deep learning benefits from an emerging class of AI accelerators. Some of these accelerators'designs are general enough for compute-intensive applications beyond AI and Cloud TPU is one such example. In this paper, we demonstrate a novel approach using TensorFlow on Cloud TPU to simulate the two-dimensional Ising Model. TensorFlow and Cloud TPU framework enable the simple and readable code to express the complicated distributed algorithm without compromising the performance. Our code implementation fits into a small Jupyter Notebook and fully utilizes Cloud TPU's efficient matrix operation and dedicated high speed inter-chip connection. The performance is highly competitive: it outperforms the best published benchmarks to our knowledge by 60% in single-core and 250% in multi-core with good linear scaling. When compared to Tesla V100 GPU, the single-core performance maintains a ∼10% gain. We also demonstrate that using low precision arithmetic—bfloat16—does not compromise the correctness of the simulation results.

CCS CONCEPTS

• Computing methodologies → Distributed simulation; Massively parallel and high-performance simulations; • Applied computing → *Physics*; *Mathematics and statistics*;

KEYWORDS

Ising Model, Cloud TPU, Markov Chain Monte Carlo

ACM Reference Format:
Kun Yang, Yi-Fan Chen, Georgios Roumpos, Chris Colby, and John Anderson. 2019. High Performance Monte Carlo Simulation of Ising Model on TPU Clusters. In *The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '19), November 17–22, 2019, Denver, CO, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3295500.3356149

*This author is no longer affiliated with Google.
†Corresponding author.

ACM ISBN 978-1-4503-6229-0/19/11.
https://doi.org/10.1145/3295500.3356149

1 INTRODUCTION

The Ising model [13], which considers short-range interactions between spin variables on the sites of a d-dimensional lattice, plays an important role in statistical physics as a prototyping system to study the universal behavior of critical phenomena. Many significant breakthroughs in statistical physics are attributed to the study of the model from either its computational or its theoretical perspective. It is well known that the Ising model has no phase transition in one dimension; however, it undergoes a second-order phase transition between an ordered and a disordered phase in two dimensions or more [5, 18]. The critical temperature T_c at which this phase transition occurs on a two-dimensional square lattice was analytically solved by Lars Onsager [18], but it is still an open problem in three or more dimensions. Computer simulation offers a powerful alternative to study such systems and determine critical temperatures, thanks to the development of finite scaling theory [3] and availability of increasing computational power. This approach ushered in a plethora of interdisciplinary applications outside of physics, including bioinformatics [2], economics [21] and operations research [11, 23].

Large-scale simulation of systems such as Ising model requires a large amount of high performance computing resources, which are usually available in multi-core computing architectures based on distributed shared memory, or distributed clusters (a.k.a data-centers) with homogeneous or heterogeneous nodes commonly seen in private or commercial clouds. Benefiting from the explosion of machine learning, especially deep learning, commercial clouds provide not only CPUs and GPUs, but also specialized chips such as FPGAs and other in-house processors. The Tensor Processing Unit ("Cloud TPU" or "TPU" for short)—an AI application-specific integrated circuit (ASIC) developed by Google for neural network machine learning—has received much attention in the machine learning community [15, 16]. Its latest release, Cloud TPU v3, offers 420×10^{12} floating-point operations per second (FLOPS) and 128GB of high bandwidth memory (HBM)[1]. Multiple units are connected to form a "POD" (Cloud TPU v3 Pod) through a dedicated high speed 2-D toroidal mesh network, allowing up to 100+ peta-FLOPS and 32TB of HBM[1] to be accessed by the application with very low latency and in lockstep. TPU is programmable via software frontends such as TensorFlow [1] or PyTorch [20], and can be deployed both for training huge deep neural networks and for performing low-latency

[1] cloud.google.com/tpu/

High Accuracy Matrix Computations on Neural Engines: A Study of QR Factorization and its Applications

Shaoshuai Zhang, Elaheh Baharlouei, Panruo Wu
Department of Computer Science
University of Houston
{szhang36,ebaharlouei,pwu7}@uh.edu

and its Applications. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '20), June 23–26, 2020, Stockholm, Sweden.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3369583.3392685

1 Introduction

Driven by the need to train large scale deep neural networks, there's been a tidal wave of the specialized low precision matrix matrix multiplication units. Among them are TensorCore from NVIDIA on its Volta and Turing architecture, Google's Tensor Processing Unit (TPU)[1], and Intel's latest FPGA, CPU, and Nervana Neural Processors. These neural engines are usually characterized by the support of lower precision arithmetic (such as 16-bit floating point format), and extremely efficient matrix-matrix multiplication. For example, NVIDIA V100 boasts up to 120 "deep learning" TeraFLOPS (120×10^{12} floating point operation per second) [32], which is half precision matrix multiplication accumulated in single precision. Google's TPU v3 claims 420 TeraFLOPS, also in doing half precision matrix-matrix multiplication. In contrast, V100 single precision peak performance is 14 TeraFLOPS, and double precision is 7TeraFLOPS. Having these neural engines greatly speeds up applications that primarily spend time in low precision matrix-matrix multiplication, and also results in much higher energy efficiency.

However, outside the neural networks, the effective use of such neural engines is only emerging. Two challenges must be addressed in using neural engines: 1) how to expose enough locality and parallelism to enable neural engines to run at high speed? 2) how to mitigate the loss of accuracy of using the limited half precision format? In this paper, we present an effective use of NVIDIA TensorCore units to QR factorize matrix and its applications in solving linear least square problems, orthogonalization, and low rank approximation. Least square problem and its many variants are prevalent in science, engineering, and statistical machine learning; for instance, non-linear least square problems are probably the largest source of all non-linear optimization problems. As such, QR factorization and its applications form a core component of any linear algebra packages such as LAPACK [1] which have been downloaded millions of times, and supported by all major processor vendors.

Thus, we set to answer the following question: is it profitable to use neural engines to accelerate common linear algebra operations reliably and accurately? We use QR factorization to demonstrate that the answer is yes, but new algorithms are needed to satisfy the data locality and parallelism that neural engines need to run at full speed and to compensate for the loss of accuracy and stability.

We consider the contributions of this paper to be:

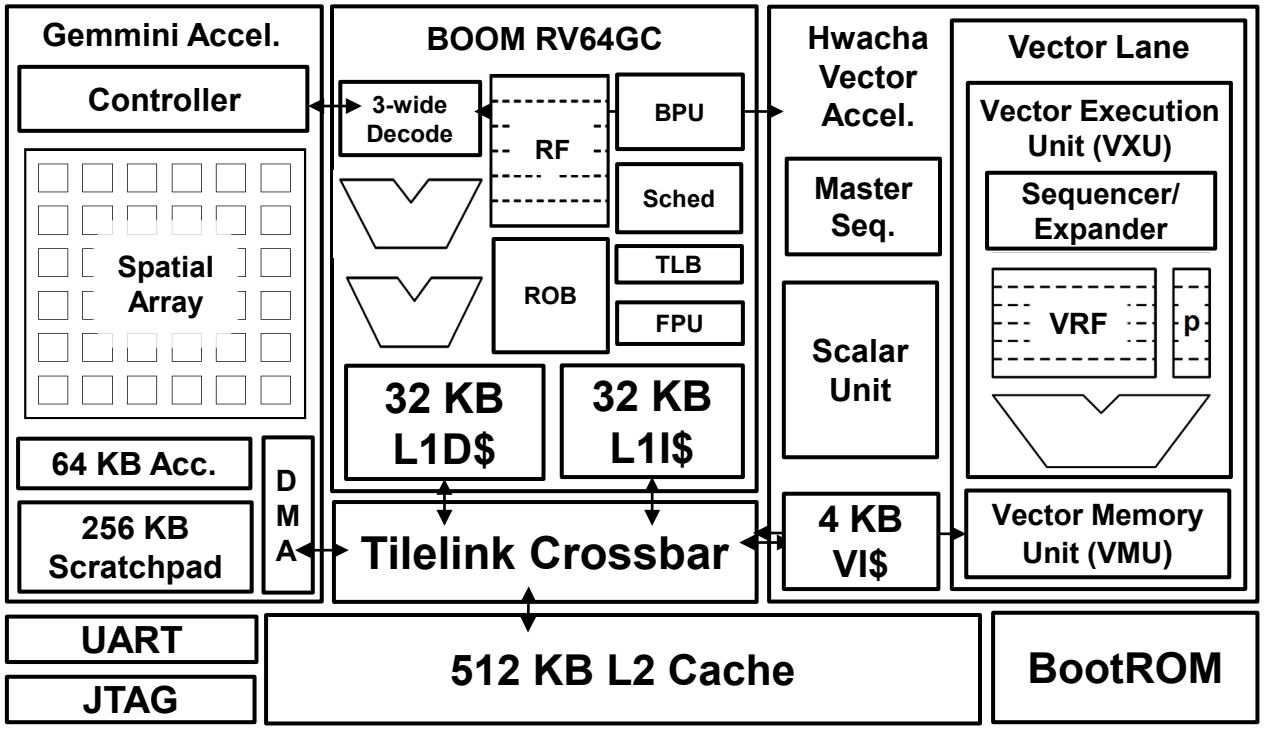
[1] https://cloud.google.com/tpu/

SoC for Data Analysis Workloads

Need to handle matrix operations, so add a deep learning accelerator with a spatial matrix multiplication unit

- Gemmini – Spatial-array DNN accelerator

```
class DataSoC extends Config(
  new gemmini.DefaultGemminiConfig
        (gemmini.GemminiFPConfigs.
                  BF16DefaultConfig)          ++
  new hwacha.DefaultHwachaConfig              ++
  new boom.common.WithNLargeBooms(1)          ++
  new freechips.rocketchip.subsystem.
             WithInclusiveCache(nBanks=4)     ++
  new chipyard.config.AbstractConfig)
```



Customize DNN Accelerator

```
val defaultConfig =
    GemminiArrayConfig[SInt, Float, Float](
      opcodes = OpcodeSet.custom3,
      tileRows = 1,
      tileColumns = 1,
      meshRows = 16,
      meshColumns = 16,

      ...

      dataflow = Dataflow.BOTH,

      inputType = SInt(8.W),
      outputType = SInt(20.W),
      accType = SInt(32.W),

      acc_read_full_width = true,
      acc_read_small_width = false,

      pe_latency = 0,
  )
```

```
val defaultFPConfig =
    GemminiArrayConfig[Float, Float, Float](
      opcodes = OpcodeSet.custom3,
      tileRows = 1,
      tileColumns = 1,
      meshRows = 8,
      meshColumns = 8,

      ...

      dataflow = Dataflow.WS,

      inputType = Float(8, 8),
      outputType = Float(8, 8),
      accType = Float(8, 24),

      acc_read_full_width = true,
      acc_read_small_width = true,

      pe_latency = 2,
   )
```


What about software?

"Because many tasks on mobile phones are run on specialized processors, Apple has hundreds of programmers who work to ensure the compatibility of Apps across iPhone generations." [1]

> *"Software Is The Hardest Word. Popular AI applications and frameworks are built on Nvidia CUDA. Accelerator vendors must port these applications to their chips.*
> *Most don't offer full compatibility.*
> *Thus, customer applications often fail to compile at first. Even after compiling, performance may not be optimized"* [2]

[1] Neil C. Thompson and Svenja Spanuth "The Decline of Computers as a General Purpose Technology: Why Deep Learning and the End of Moore's Law are Fragmenting Computing"
[2] Linley Gwennap, "Application-Specific Accelerators Extend Moore's Law", Keynote, Linley Fall Processor Conference 2020.

Accel. Integration into DNN Stack

DNN Model: ResNet | MobileNet | BERT | DLRM | YOLO

DNN/ML Framework: TensorFlow | PyTorch | Caffe | MxNet

DNN Graph Formats: ONNX | NNEF

DNN Graph Compilers: ONNX-Runtime | XLA | TVM | Glow | nGraph

Optimized Libraries: Gemmini SDK | MKL-DNN | cuDNN | ATLAS | OpenBLAS | BLIS | NVBLAS | MKL | Accelerate

Low-level Language Runtimes: CUDA | C / C++ | OpenCL | OpenMP | Posix Threads

Operating System: Linux | Windows | Android | RTOS | MacOS/iOS | Bare-metal

Hardware: CPU | GPU | TPU/NPU | FPGA | DSP


Gemmini SDK

```
val defaultFPConfig =
    GemminiArrayConfig[Float, Float, Float](
      opcodes = OpcodeSet.custom3,
      tileRows = 1,
      tileColumns = 1,
      meshRows = 8,
      meshColumns = 8,

      ...

      dataflow = Dataflow.WS,

      inputType = Float(8, 8),
      outputType = Float(8, 8),
      accType = Float(8, 24),

      acc_read_full_width = true,
      acc_read_small_width = true,

      pe_latency = 2,
   )
```

```
#ifndef GEMMINI_PARAMS_H
#define GEMMINI_PARAMS_H

#define XCUSTOM_ACC 3
#define DIM 8
#define ADDR_LEN 32
#define BANK_NUM 4
#define BANK_ROWS 4096
#define ACC_ROWS 2048
#define MAX_BYTES 64
#define MAX_BLOCK_LEN (MAX_BYTES/(DIM*2))
#define MAX_BLOCK_LEN_ACC (MAX_BYTES/(DIM*4))

typedef uint16_t elem_t;
#define ELEM_T_IS_LOWPREC_FLOAT
typedef float acc_t;

#define ELEM_T_IS_FLOAT
#define ELEM_T_EXP_BITS 8
#define ELEM_T_SIG_BITS 8
#define ACC_T_EXP_BITS 8
#define ACC_T_SIG_BITS 24
typedef uint16_t elem_t_bits;
typedef uint32_t acc_t_bits;

#define HAS_MVIN_SCALE
typedef float scale_t;
typedef uint32_t scale_t_bits;

#define ACC_READ_SMALL_WIDTH
#define ACC_READ_FULL_WIDTH

...
```

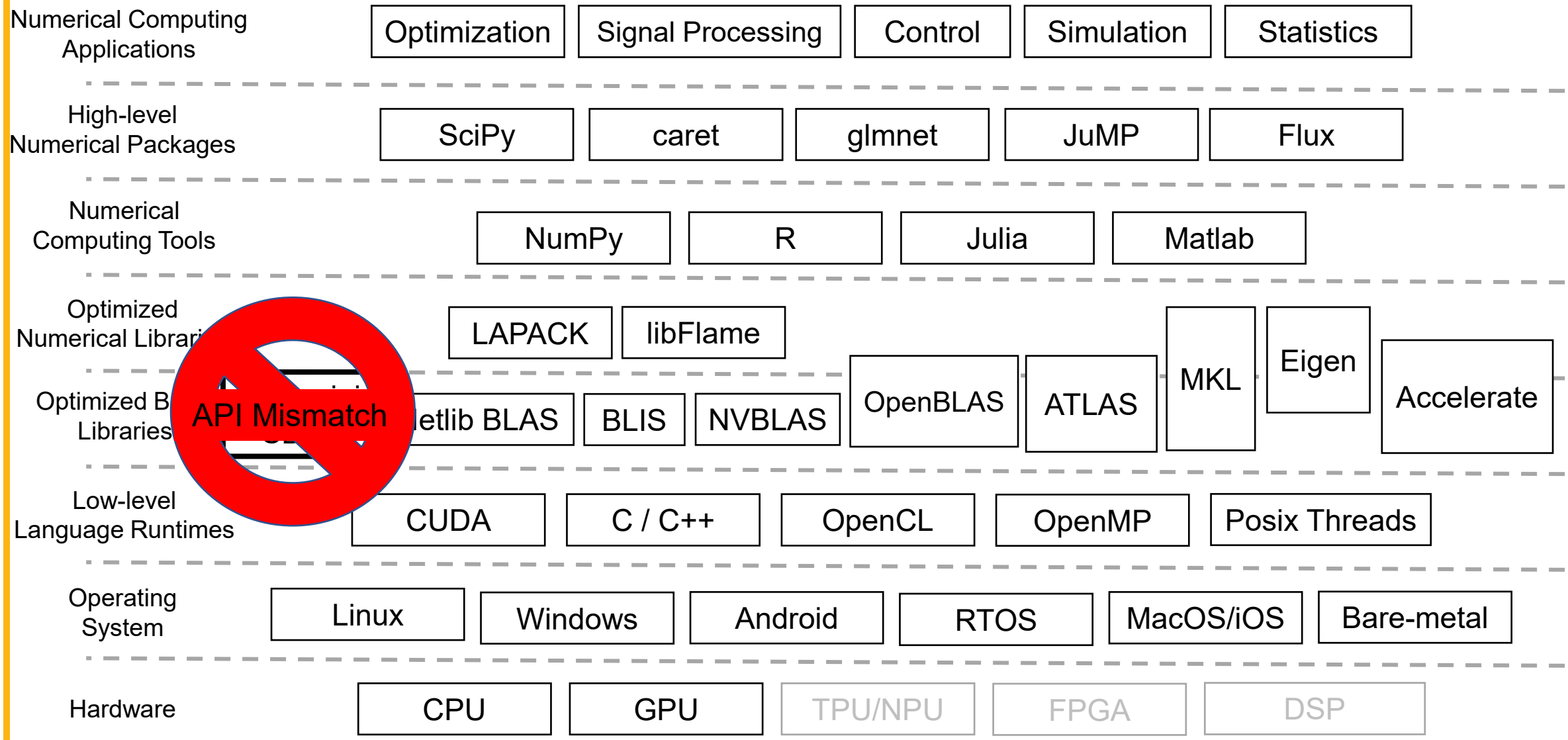
Numerical Computing Stack

Numerical Computing Applications	Optimization	Signal Processing	Control	Simulation	Statistics					
High-level Numerical Packages	SciPy	caret	glmnet	JuMP	Flux					
Numerical Computing Tools	NumPy	R	Julia	Matlab						
Optimized Numerical Libraries	LAPACK	libFlame								
Optimized Basic Libraries	Netlib BLAS	BLIS	NVBLAS	OpenBLAS	ATLAS	MKL	Eigen	Accelerate		
Low-level Language Runtimes	CUDA	C / C++	OpenCL	OpenMP	Posix Threads					
Operating System	Linux	Windows	Android	RTOS	MacOS/iOS	Bare-metal				
Hardware	CPU	GPU	TPU/NPU	FPGA	DSP					

Numerical Computing Stack

Numerical Computing Applications	Optimization	Signal Processing	Control	Simulation	Statistics						
High-level Numerical Packages	SciPy	caret	glmnet	JuMP	Flux						
Numerical Computing Tools	NumPy	R	Julia	Matlab							
Optimized Numerical Libraries	LAPACK	libFlame									
Optimized Basic Libraries	Gemmini SDK?	Netlib BLAS	BLIS	NVBLAS	OpenBLAS	ATLAS	MKL	Eigen	Accelerate		
Low-level Language Runtimes	CUDA	C / C++	OpenCL	OpenMP	Posix Threads						
Operating System	Linux	Windows	Android	RTOS	MacOS/iOS	Bare-metal					
Hardware	CPU	GPU	TPU/NPU	FPGA	DSP						

Numerical Computing Stack




Numerical Computing Stack

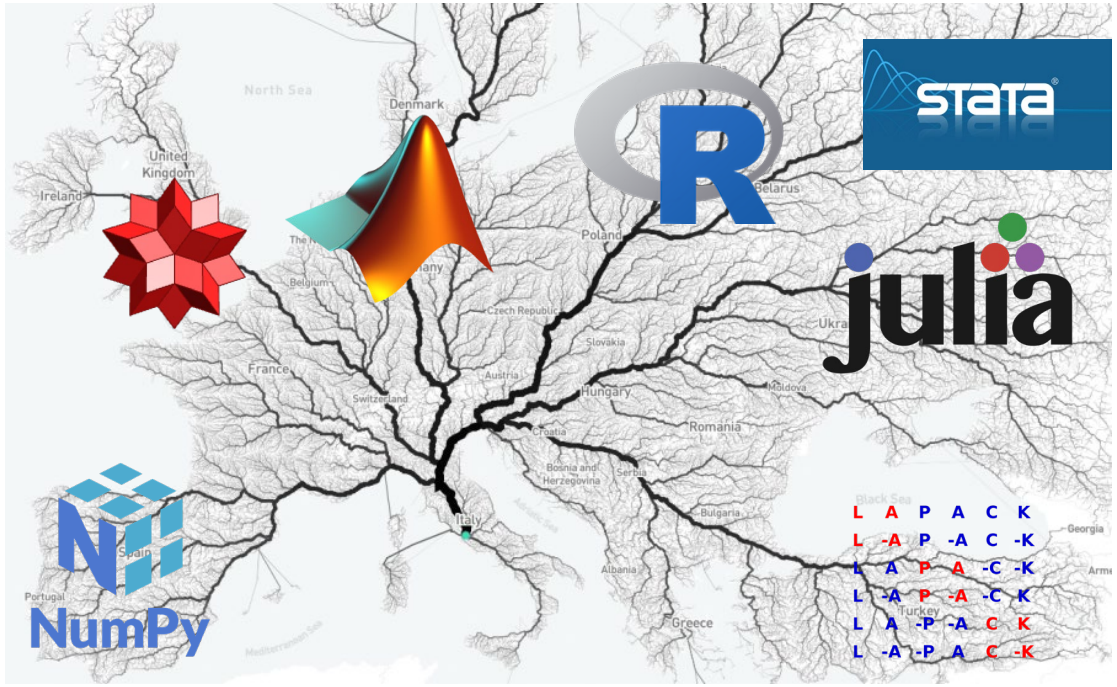
Numerical Computing Applications	Optimization	Signal Processing	Control	Simulation	Statistics	
High-level Numerical Packages	SciPy	caret	glmnet	JuMP	Flux	
Numerical Computing Tools	NumPy	R	Julia	Matlab		
Optimized Numerical Libraries	LAPACK	libFlame				
Optimized Basic Libraries	Netlib BLAS	BLIS	NVBLAS	OpenBLAS	ATLAS	MKL
	Gemmini SDK					
Low-level Language Runtimes	CUDA	C / C++	OpenCL	OpenMP	Posix Threads	
Operating System	Linux	Windows	Android	RTOS	MacOS/iOS	Bare-metal
Hardware	CPU	GPU	TPU/NPU	FPGA	DSP	

Eigen

Accelerate

BLAS

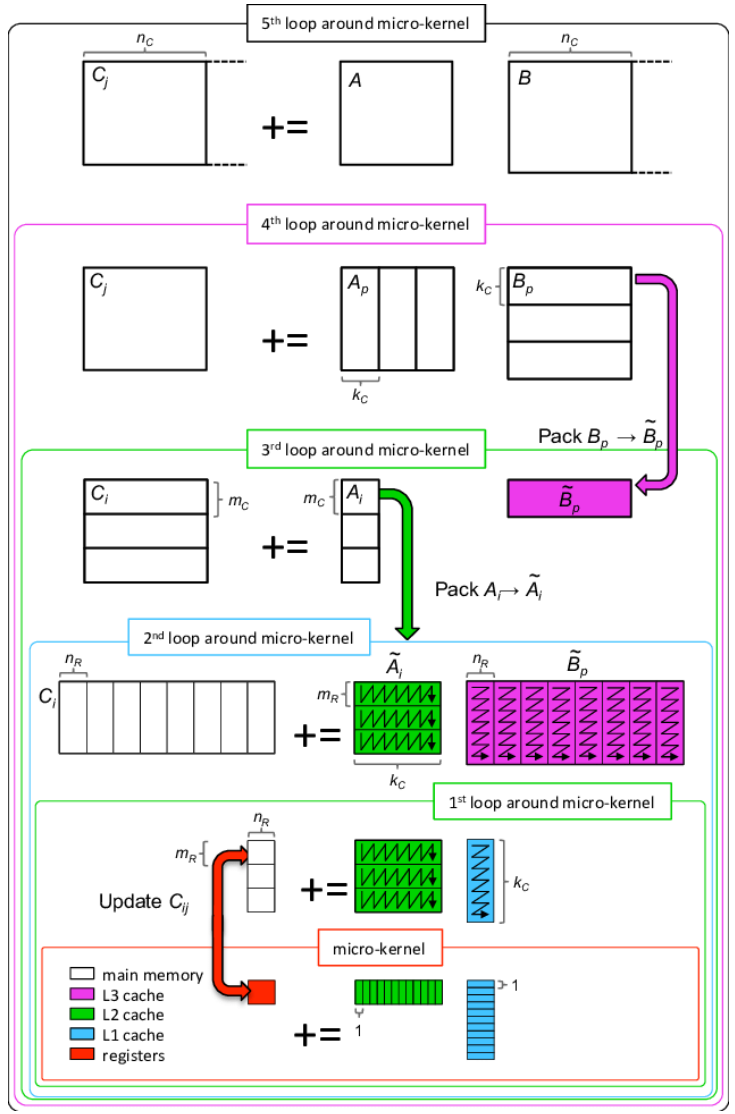
- "All roads lead to ~~Rome~~ BLAS"
  - BLAS-1 – vector operations
  - BLAS-2 – matrix-vector operations
  - BLAS-3 – matrix-matrix operations
- Widely-adopted API (together with LAPACK):
  - ABI compatibility
- Accepted Nomenclature (**XYYZZZ_**):
  - X – datatype
  - YY – matrix type
  - ZZZ – computation type
- Self-documenting decomposition for high-level numerical algorithms



https://www.openculture.com/2018/05/an-interactive-map-shows-just-how-many-roads-actually-lead-to-rome.html

BLIS [1]

- Based on the Goto/BLIS algorithm [2]
  - Like OpenBLAS/GotoBLAS
  - Streaming into L1 rather than keeping data in L1
  - Packing into block-panel structure
- Portable, Template-based, Open-source
  - Architecture-specific code encapsulated in microkernels
  - Three compute micro-kernels: GEMM, TRSM, GEMMTRSM
  - Other kernels can be overrides for specific architectures
- Generates a complete optimized BLAS API implementation



[1] Field G. Van-Zee, and Robert A. van de Geijn “BLIS: A Framework for Rapidly Instantiating BLAS Functionality”, *ACM Transactions on Mathematical Software (TOMS)* 41.3 (2015): 1-33.
[2] Goto, Kazushige, and Robert A. van de Geijn. "Anatomy of high-performance matrix multiplication." *ACM Transactions on Mathematical Software (TOMS)* 34.3 (2008): 1-25.


Will it work out-of-the-box? (No)

Numerical Computing Applications	Optimization	Signal Processing	Control	Simulation	Statistics	
High-level Numerical Packages	SciPy	caret	glmnet	JuMP	Flux	
Numerical Computing Tools	NumPy	R	Julia	Matlab		
Optimized Numerical Libraries	LAPACK	libFlame				
Optimized Basic Libraries	Netlib BLAS	BLIS	NVBLAS	OpenBLAS	ATLAS	MKL / Eigen / Accelerate
	Gemmini SDK					
Low-level Language Runtimes	CUDA	C / C++	OpenCL	OpenMP	Posix Threads	
Operating System	Linux	Windows	Android	RTOS	MacOS/iOS	Bare-metal
Hardware	CPU	GPU	TPU/NPU	FPGA	DSP	

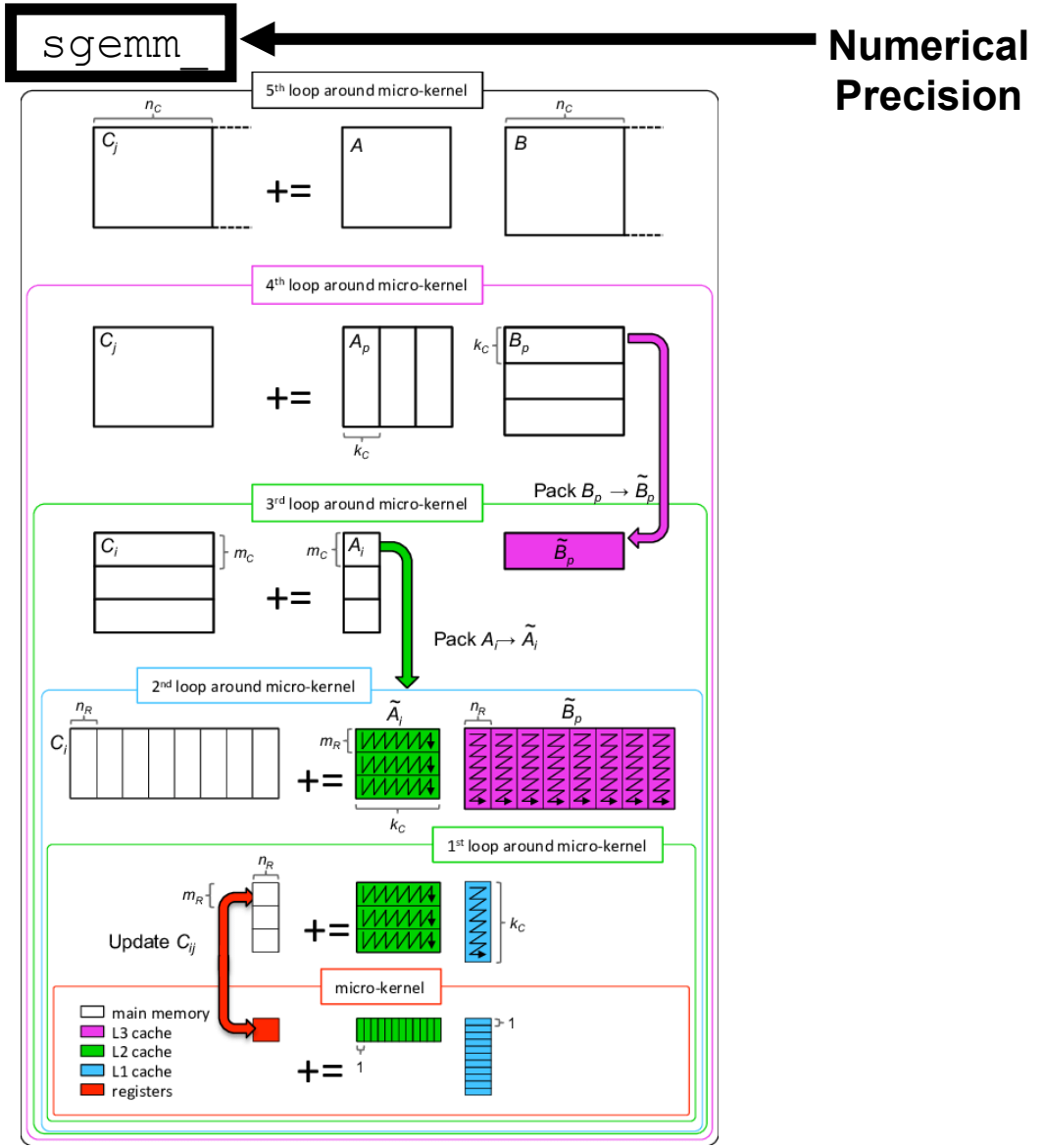
Accelerator Numerics

- BLAS is only floating point
- Accelerator numerics
  - Edge DNN accelerators likely to have Int8, FP16 or BF16
- Would low precision work for general-purpose workloads?
  - Should be sufficient for basic statistics (depending on data precision)
  - Probably shouldn't use for weather/nuclear simulations
  - Numerical stability

	Matrix Multiplication Accelerator Numerics						
	Int4	Int8	Int16	fp16	bf16	fp32	tf32[1]
NVIDIA Volta TensorCore	✓	✓		✓			
NVIDIA Ampere TensorCore	✓	✓	✓	✓	✓	✓	✓
Google TPUv1		✓					
Google TPUv2					✓		
Google TPUv3					✓		
Intel AMX		✓			✓		
AWS Inferentia		✓		✓	✓		
AWS Trainium							
Qualcomm Hexagon [2]		✓					
Huawei Da Vinci [3]		✓		✓			
MediaTek APU 3.0		✓	✓	✓			
NVIDIA DLA [4]		✓	✓	✓			
Samsung NPU [5]		✓					
Tesla NPU		✓					

BLAS Numerics

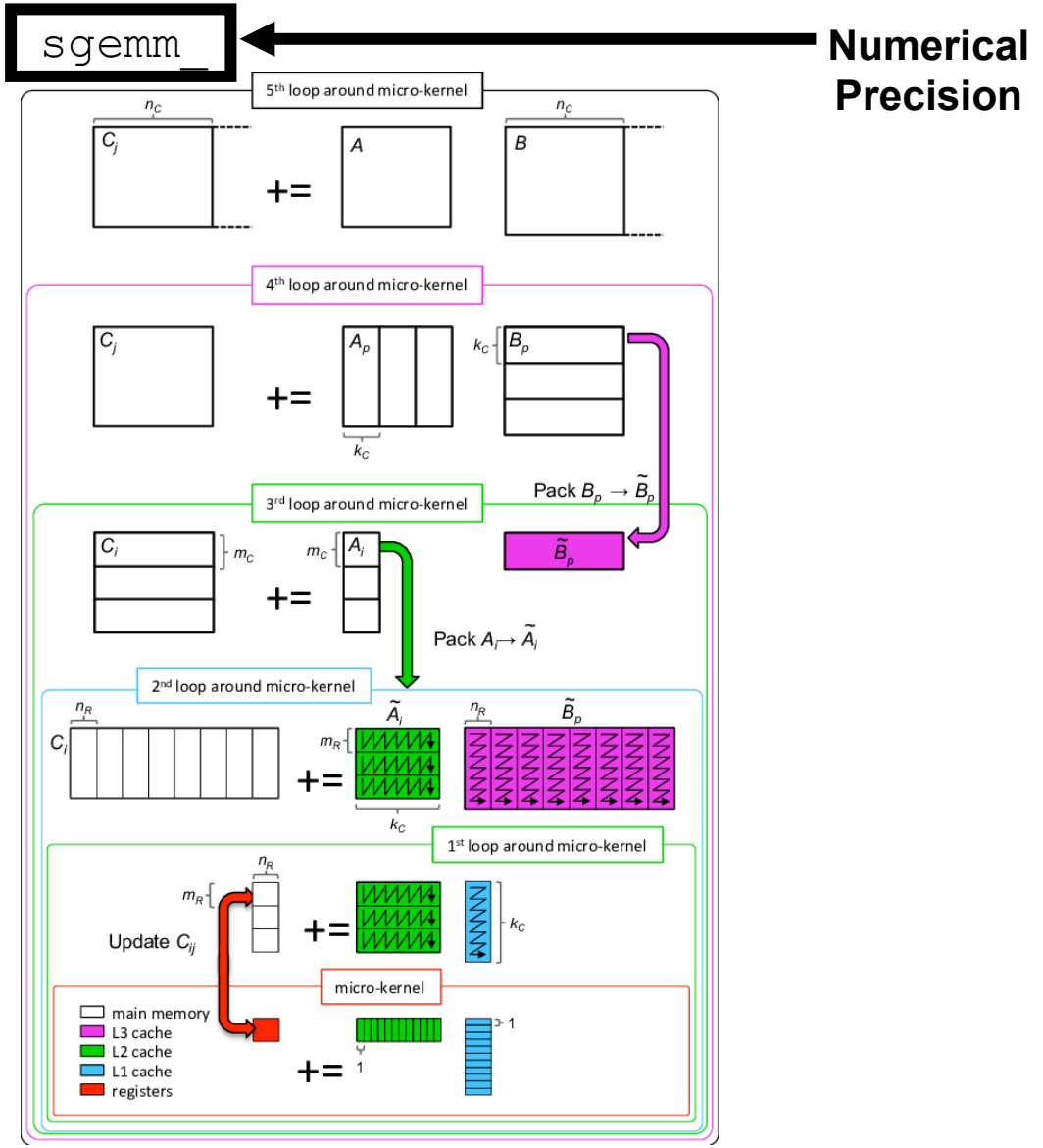
- BLAS does not have a standard low-precision API
- How do current HPC applications deal with low precision?
  - End-to-end mixed precision algorithms ("under the hood") [1]
  - Application-level static analysis and explicit replacement (Precimonious [2])
- We would like to integrate at the BLAS level for transparent integration with higher-level apps (NumPy, etc.)



[1] Azzam Haidar et al. "Harnessing GPU tensor cores for fast FP16 arithmetic to speed up mixed-precision iterative refinement solvers
[2] C. Rubio-Gonzalez et al. "Precimonious: Tuning assistant for floating-point precision"

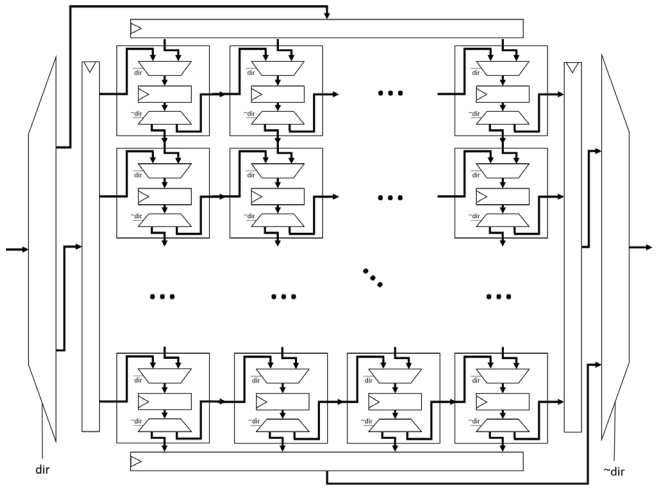
BLAS Numerics

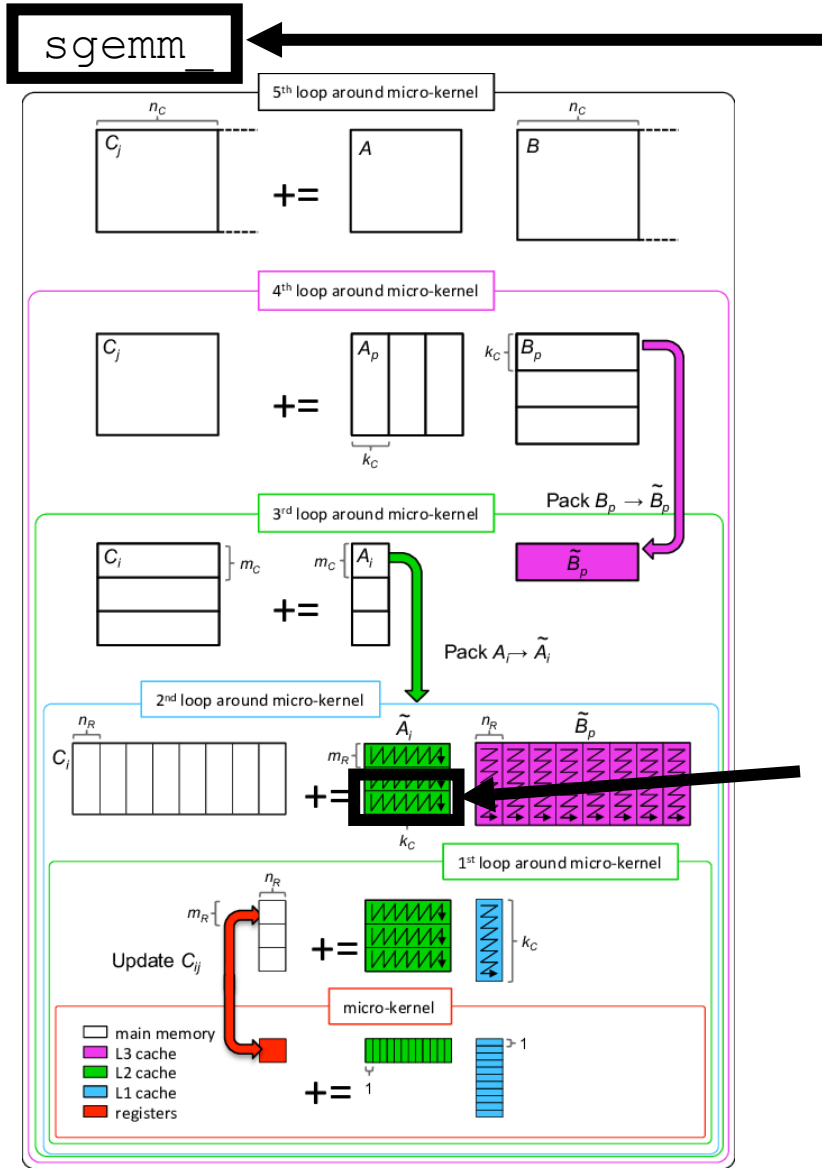
- `RELAXED_NUMERICS` environment variable
  - Coarse-grain control
  - Not all applications actually need single-precision (depends on source data precision)
  - Maintain a single-precision API, but perform GEMM computation in BF16 if `RELAXED_NUMERICS` environment is true
  - Automatically fallback on FP32 in vector unit
- Enables transparent integration with deep legacy software stack



BLAS Data Layout

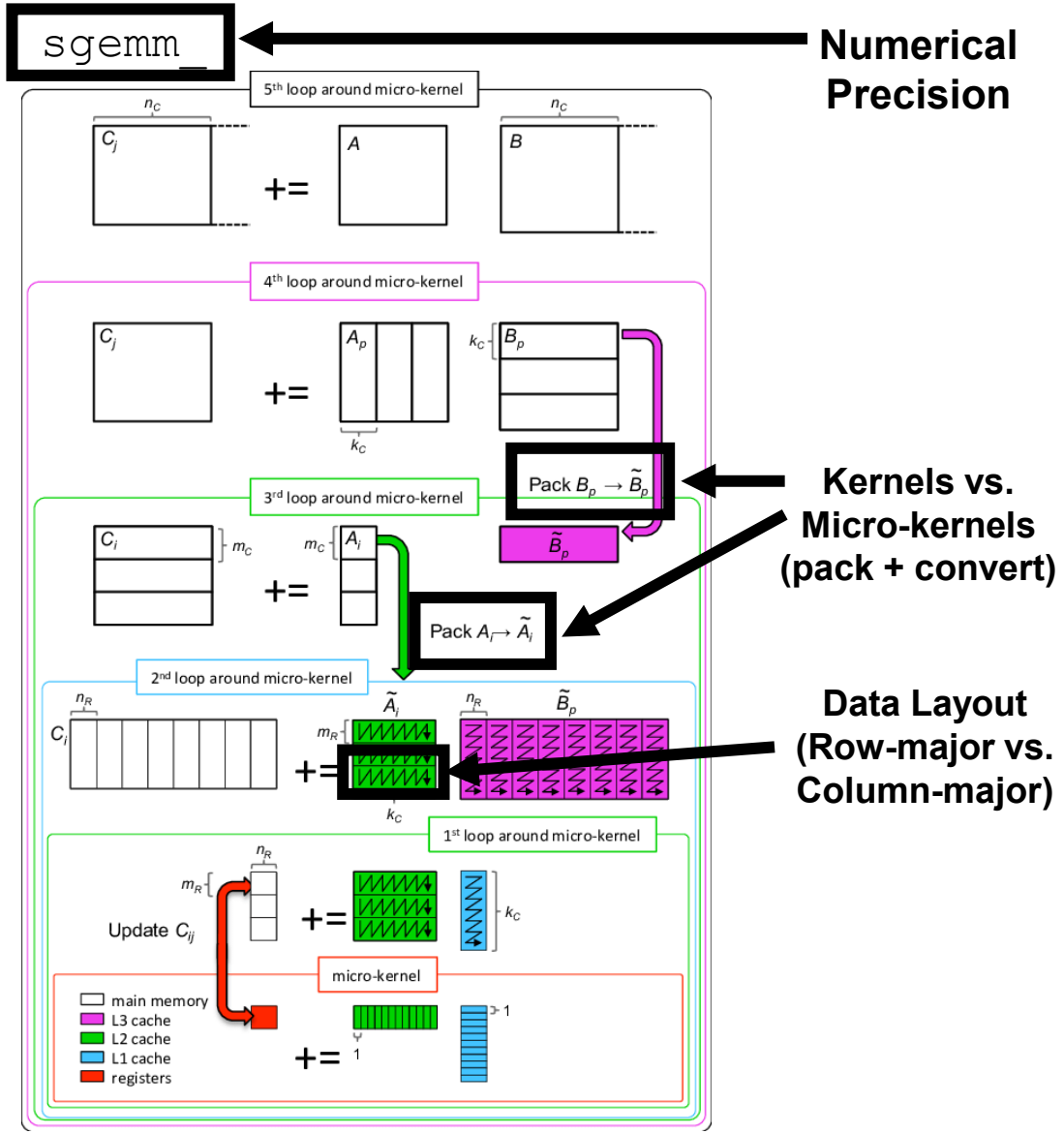
- Data layout: row-major vs column-major
  - Gemmini assumes row-major
  - Transposed computation in BLAS
- Hardware Transposer in Gemmini
  - Was already there for OS dataflow
  - Low-cost (1% compared to compute array area). Just need to expose to software





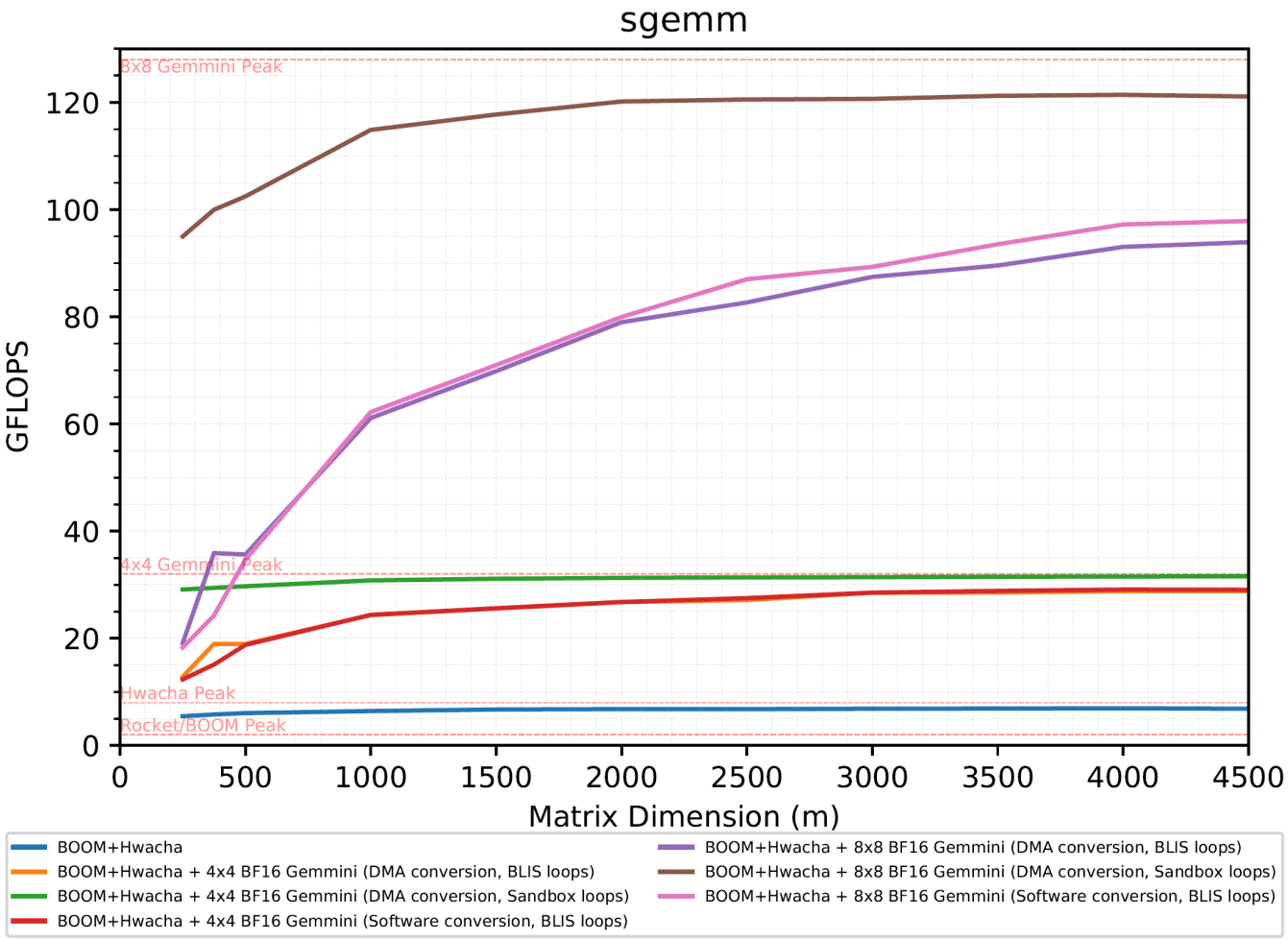
BLIS

- Kernels vs. Micro-kernels
  - GEMM
  - TRSM
  - TRMM, SYMM, SYRK, etc.
- Micro-kernel: good and bad
  - Good: Generalized for multiple BLAS-3 kernels
  - Bad: Assumes fixed-size hardware support (not good for variable length vectors or zero-padded matrix units with hardware sequencers)
- Kernel:
  - Good: Optimized end-to-end
  - Bad: Development time for each new uarch



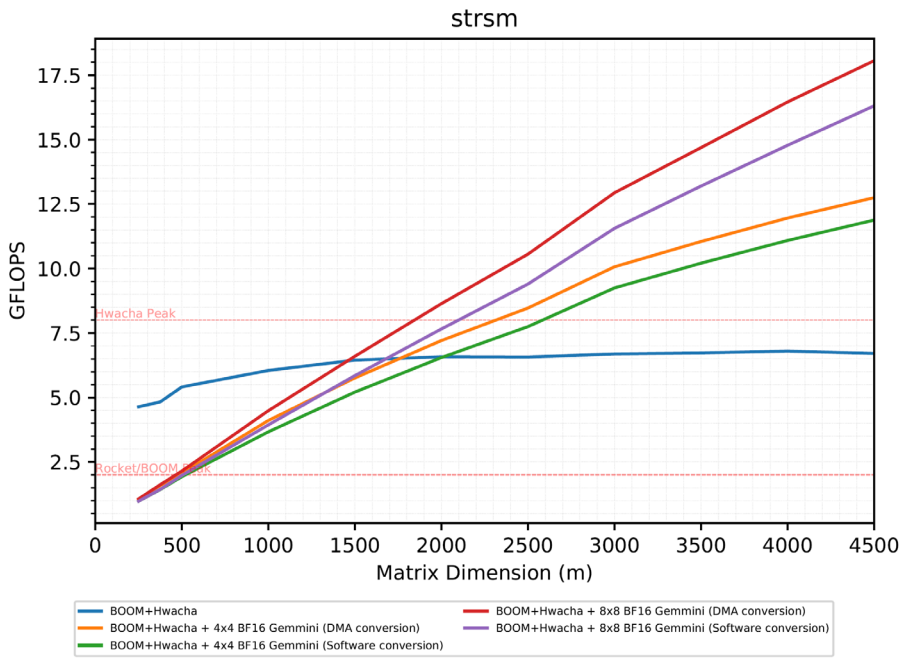

BLIS

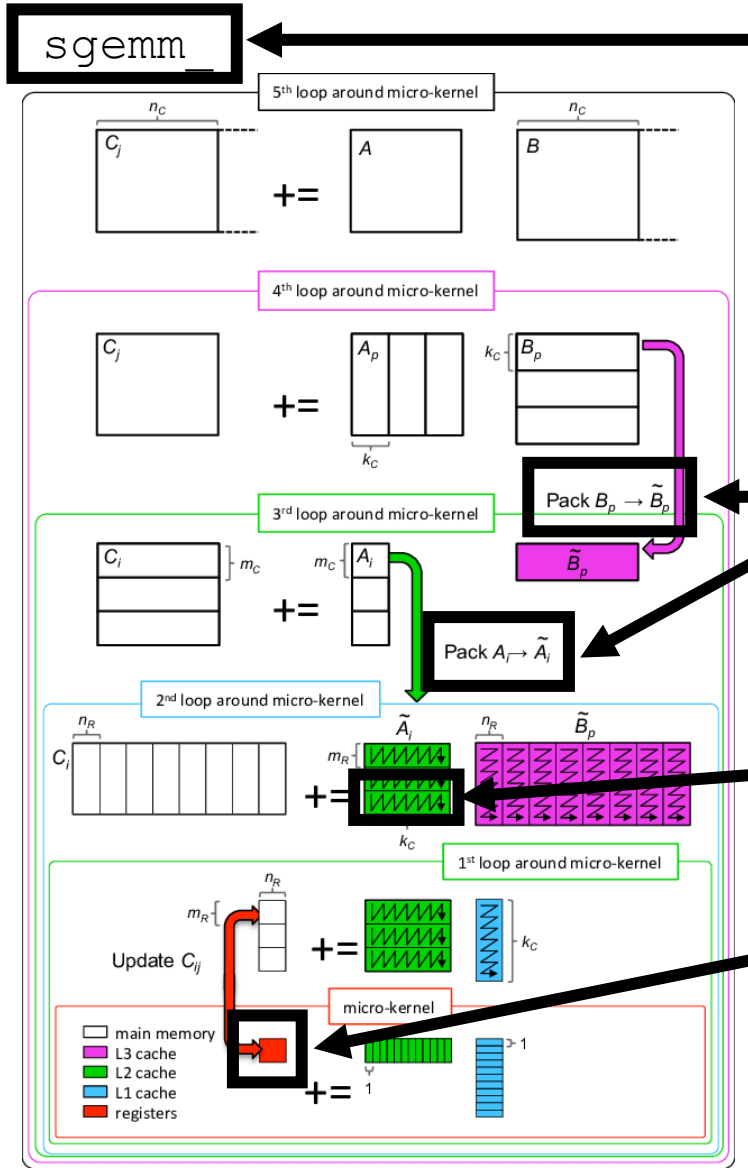
- On-the-fly conversion vs. L2 pack+convert
  - TPU vs. Intel AMX
  - DMA bandwidth vs. vector unit conversion bandwidth and fencing
  - Hardware controller flow continuity and Gemmini latency-hiding
- Zero padding
  - Hardware-padding in kernel
  - Software-padding for micro-kernel, due to fixed ukernel size



BLIS

- And more….
  - Small Matrices
  - BLAS-3 - Register blocking size
  - TRSM





Numerical Precision

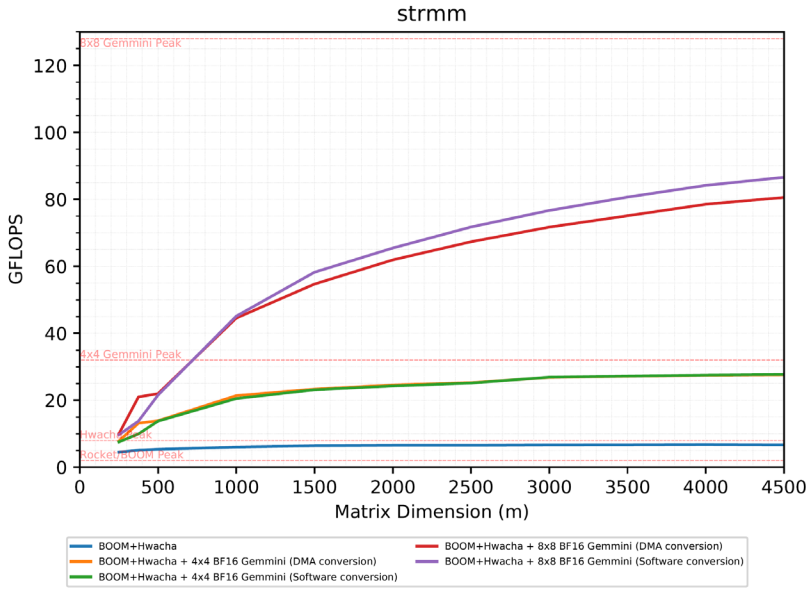
Kernels vs. Micro-kernels (pack + convert)

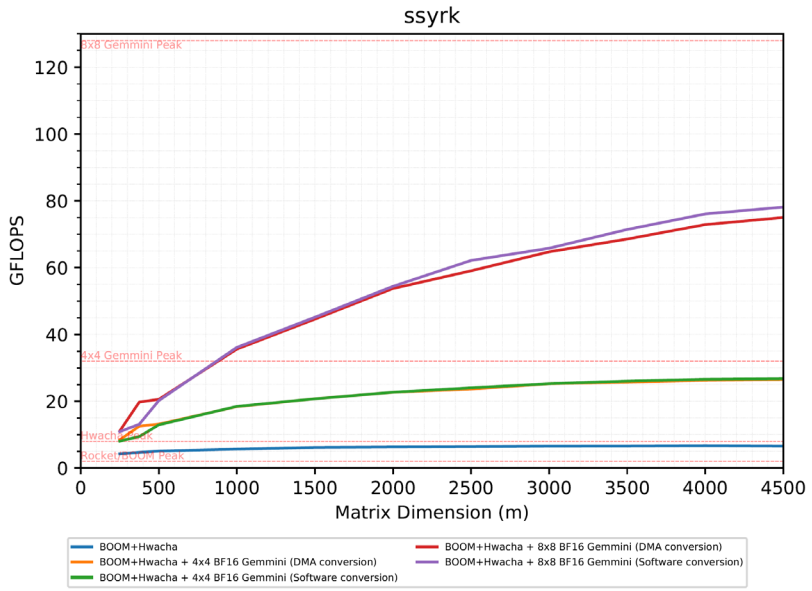
Data Layout (Row-major vs. Column-major)

Matrix Size & Large Register Blocking

BLAS-3 Performance

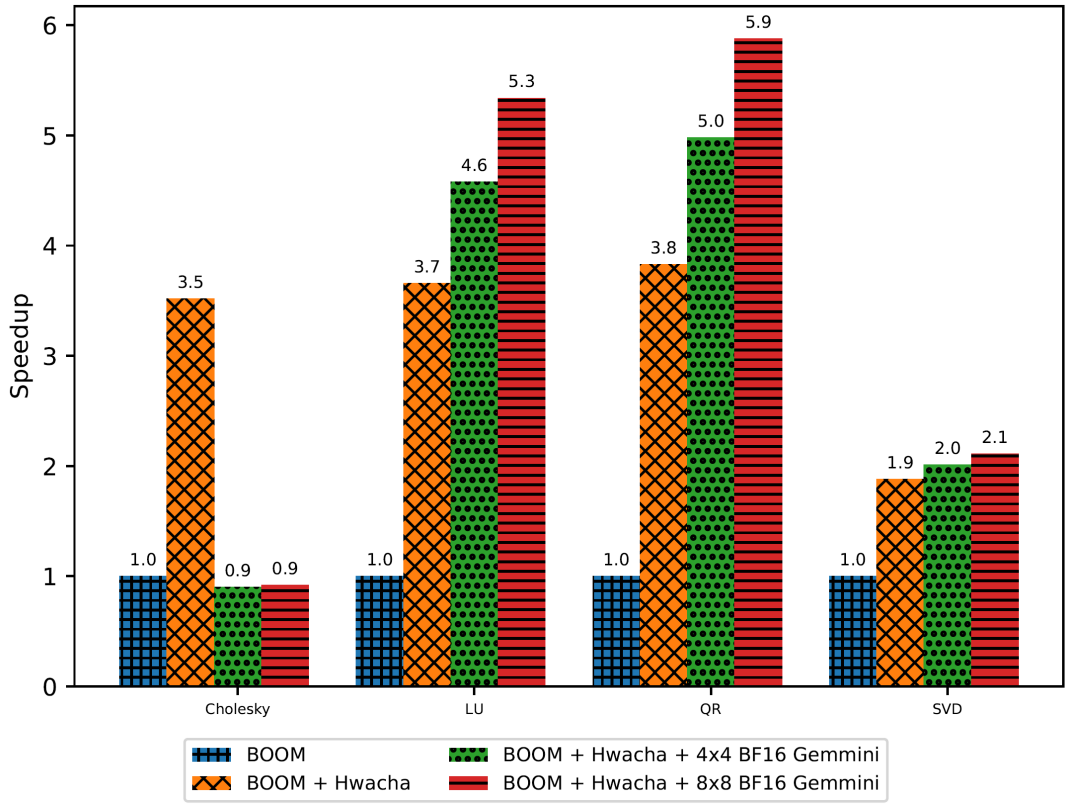
- GEMM: 95-98% Utilization
- TRMM/SYRK/SYMM
  - Micro-kernel-based implementations
  - 60%-70% utilization on 8x8 Gemmini
  - 80%-90% utilization on 4x4 Gemmini
- Need large matrices for good utilization
  - >1000 for GEMM
- Residual norm ~ $10^{-5} - 10^{-7}$





Matrix Decompositions

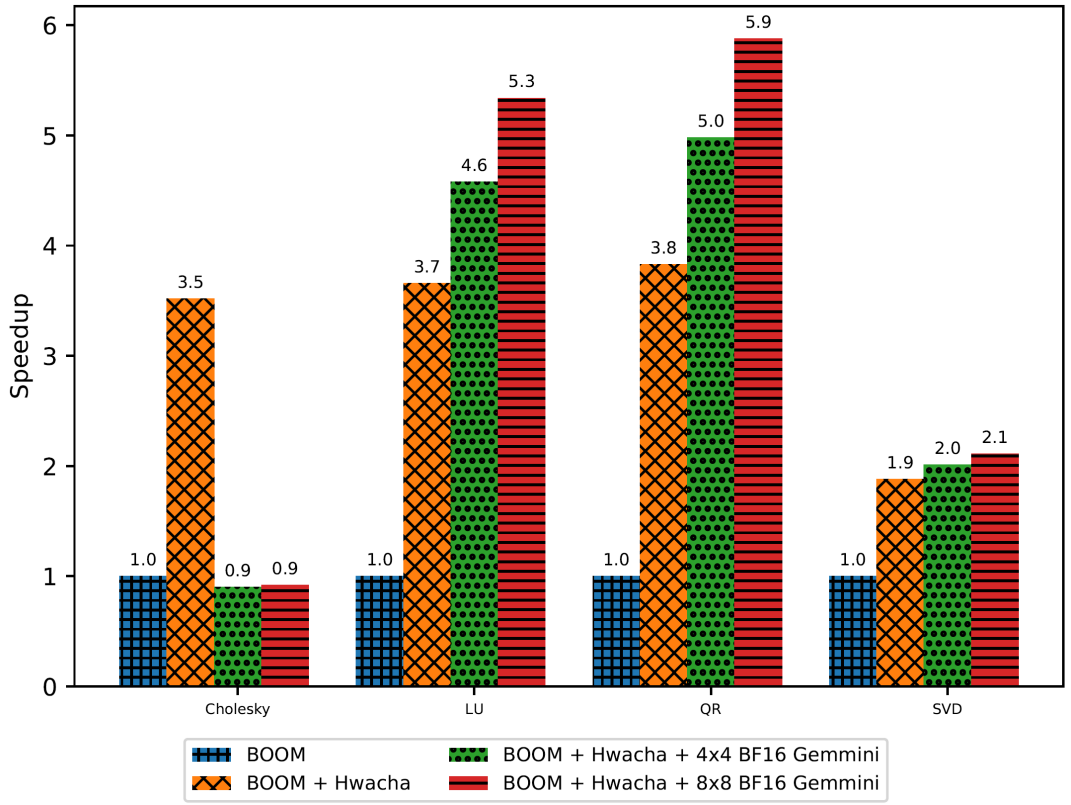
- Matrix decompositions as core linear algebra kernels
  - LU and Cholesky decompositions for linear system solve
  - QR, SVD for least squares solutions and low-rank approximation
- Diminishing returns with matrix unit compared to vector unit
  - Amdahl's Law
  - 1.9x-3.8x speedup using vector unit over scalar processor
  - 1.06x-1.3x speedup using 4x4 Gemmini over vector unit
  - 1.05x – 1.18 speedup using 8x8 Gemmini over 4x4 Gemmini



Matrix decompositions on 1600x1600 square matrix

Matrix Decompositions

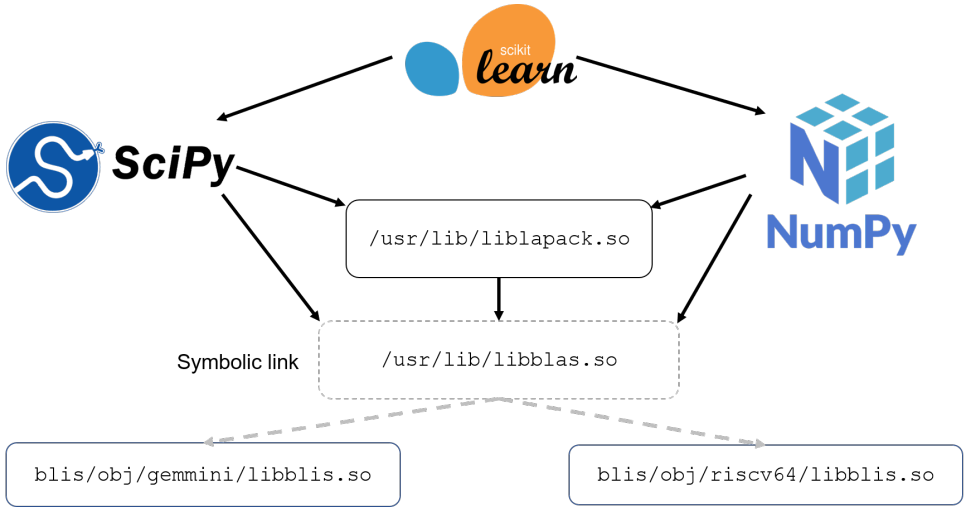
- SVD – bidiagonalization limited to BLAS-2
  - ~50% of the operation count
  - Limited speedup to ~2x (Amdahl's law)
- Cholesky – slowdown with Gemmini
  - Micro-kernels in BLIS
  - Recursive LAPACK algorithms

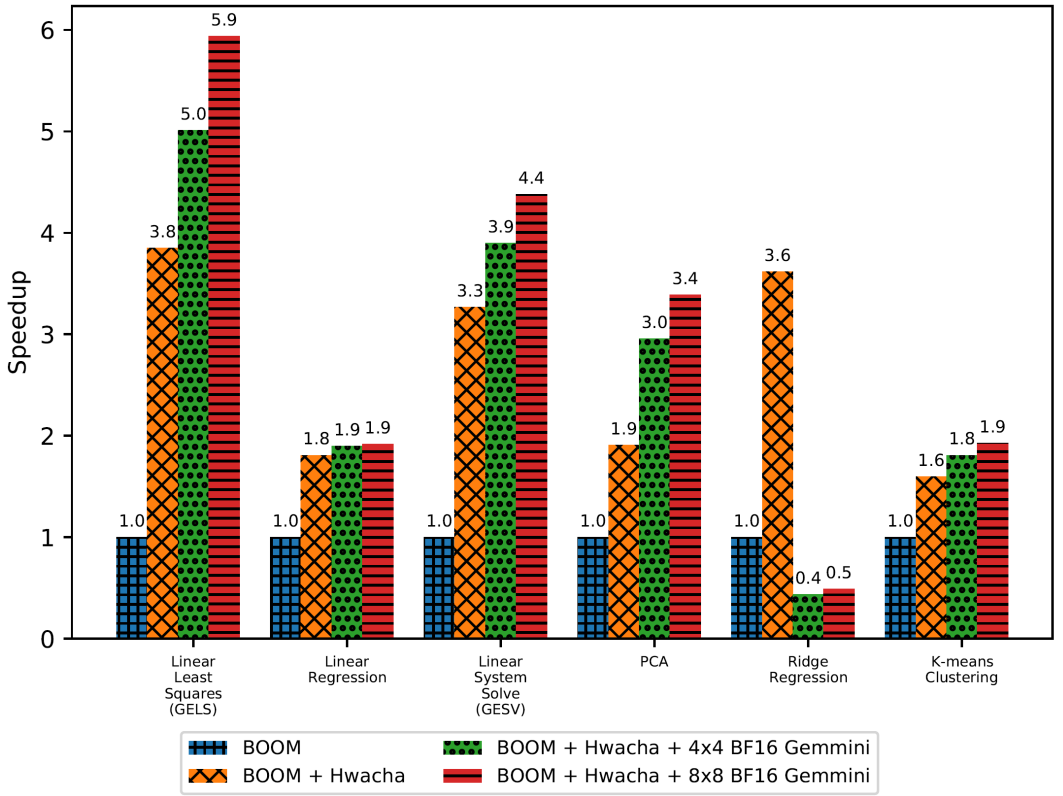


Matrix decompositions on 1600x1600 square matrix

Python Apps

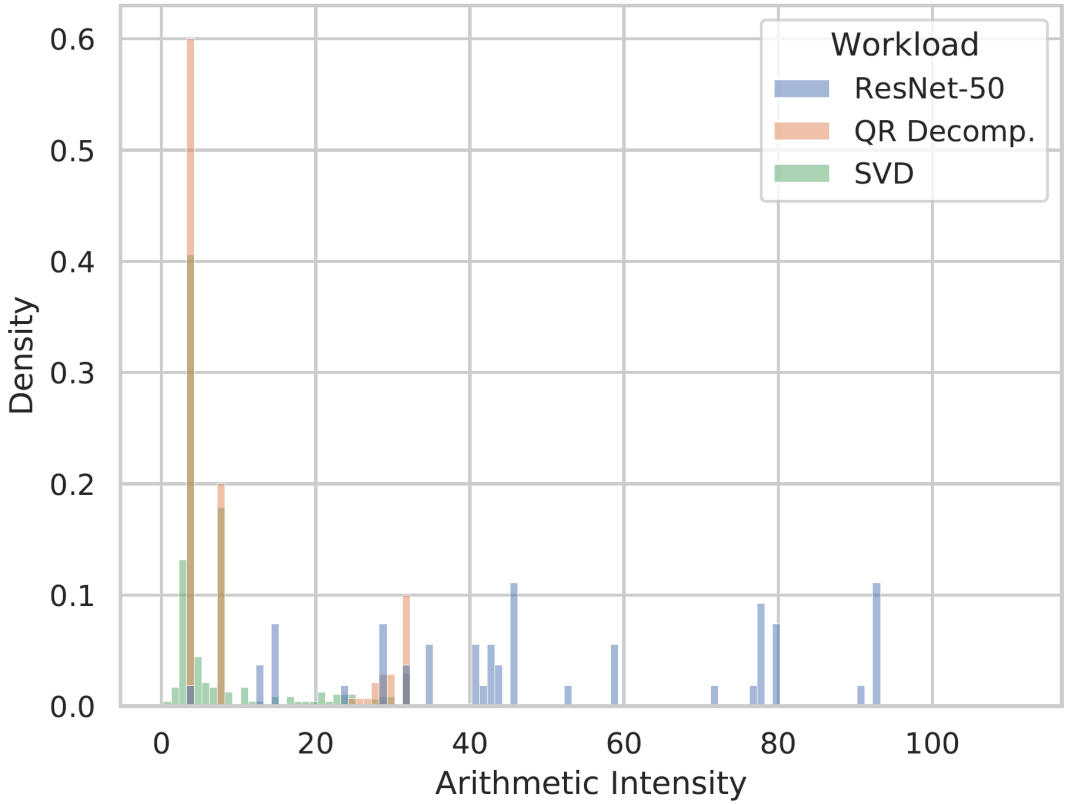
- SciPy and Scikit-Learn
  - Full-stack applications
  - Data-scientist perspective
- Speedups similar to matrix decompositions
- PCA
  - Randomized SVD => higher speedup than SVD.
- Linear Models
  - Ridge slowdown
  - Scikit-learn LinearRegression vs. LAPACK GELS least squares





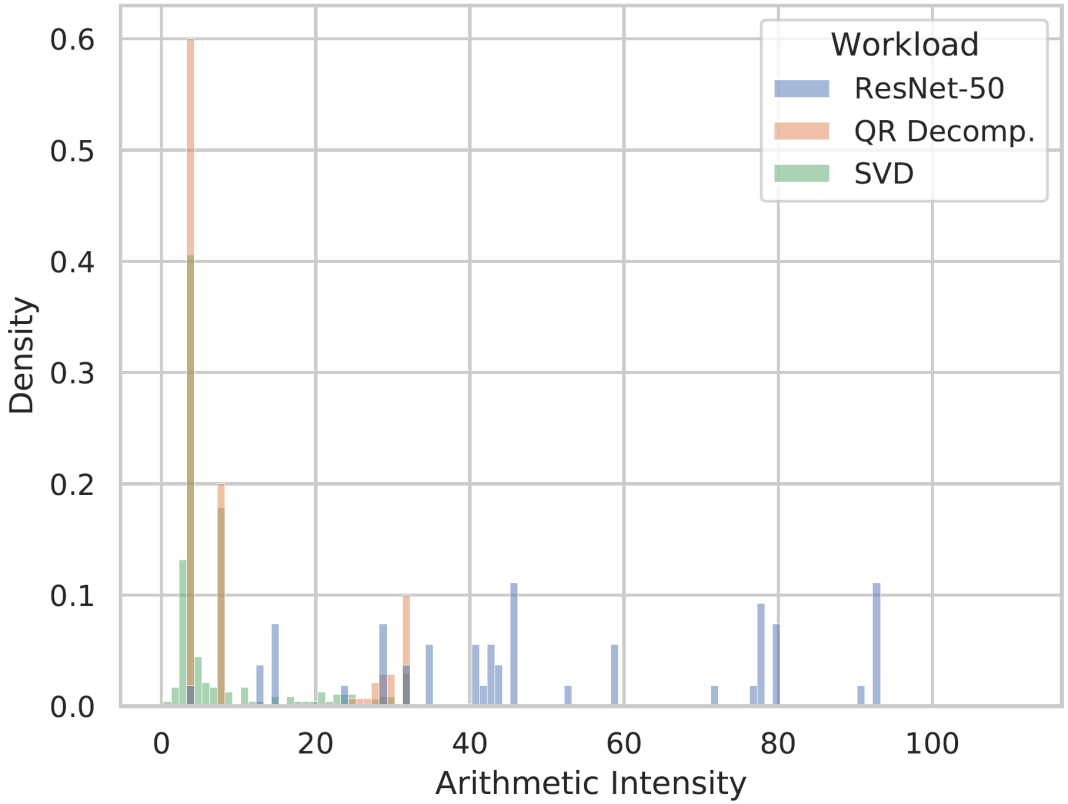
But…..

- Are DNN accelerators actually a good fit for general numerical data analysis matrix operations?
- The arithmetic intensity of BLAS-3 operations in general numerical data analysis kernels is much lower than DNN models
  - More smaller matrices
  - More rectangular matrices



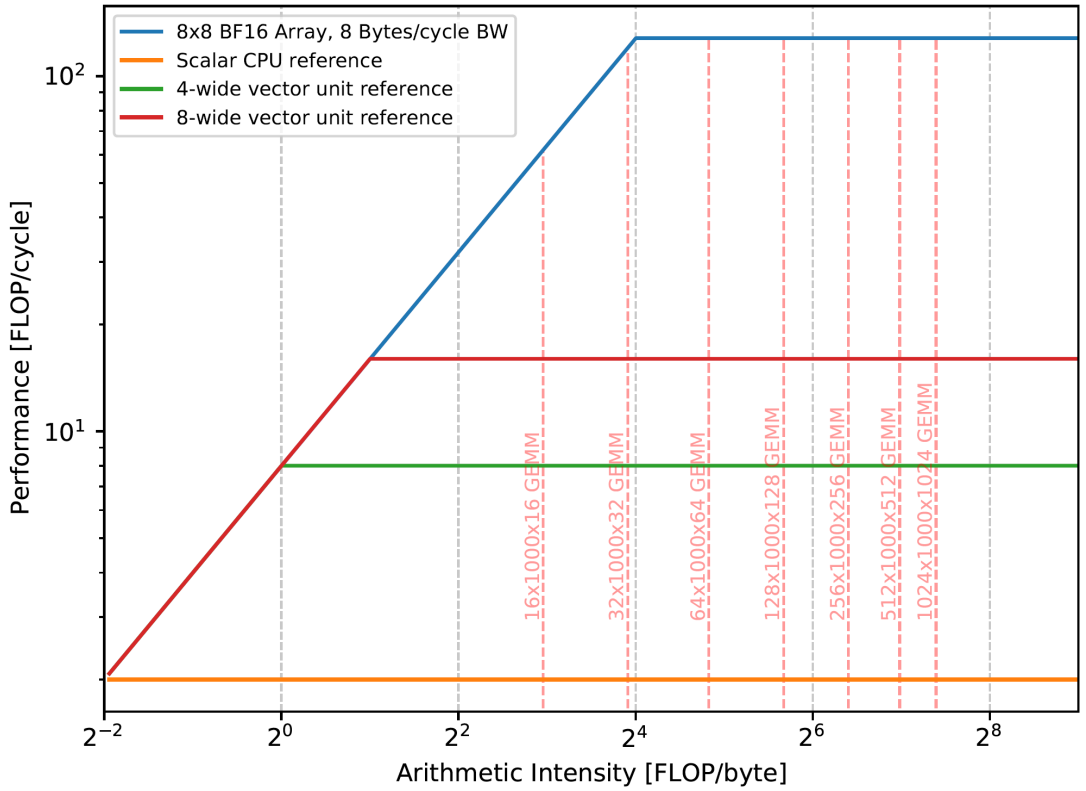

But…..

- Example ResNet-50 matrix shapes (batch size 1):
  - m=784, n=512, k=256
  - m=784, n=256, k=512
- Example QR decomp. matrix shapes (block size 32):
  - m=7096, n=305, k=32
  - m=305, n=32, k=7096
  - m=7192, n=7192, k=32
  - m=7192, n=32, k=7192
  - m=7192, n=32, k=32



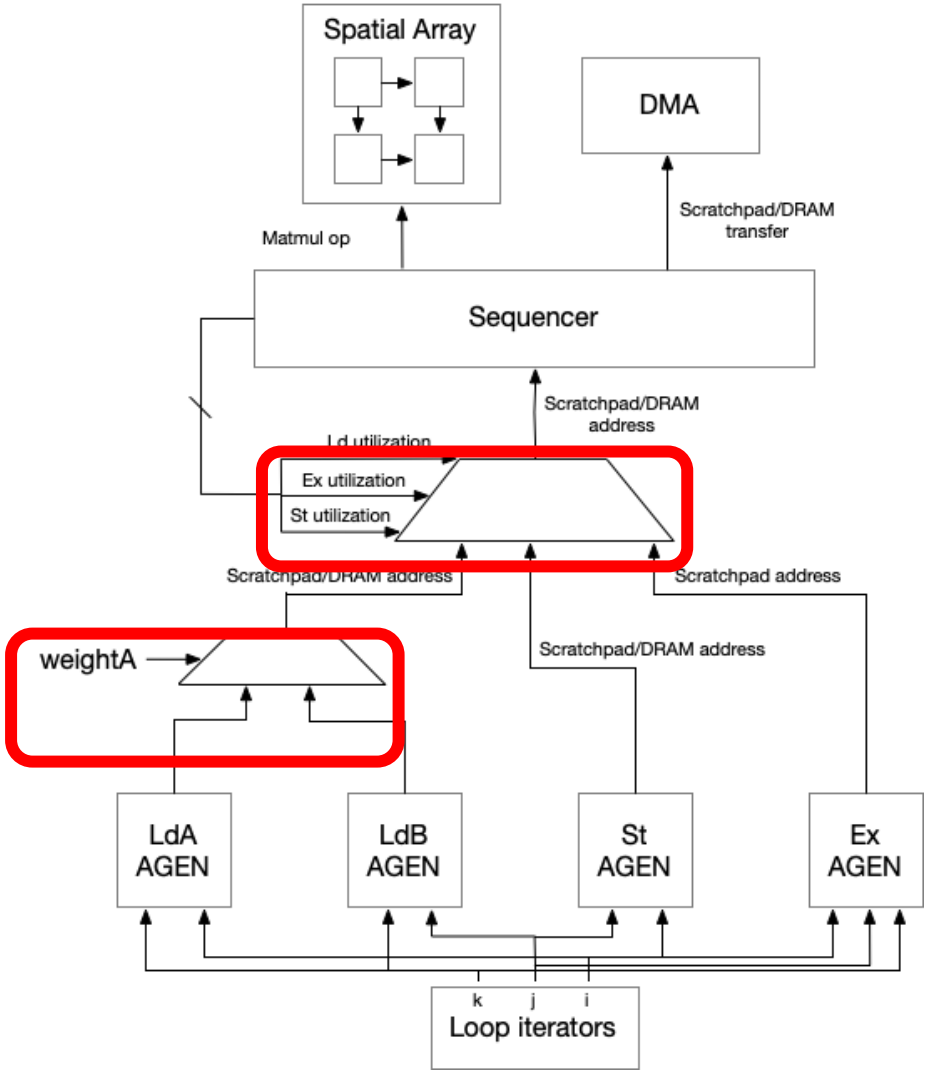
But.....

- While relatively low arithmetic intensity can *easily* saturate a typical 1D vector unit, many low arithmetic-intensity shapes becomes memory bound when using a DNN accelerator such as Gemmini.
- Scheduling becomes important
  - Static scheduling
  - Dynamic scheduling
- Hardware scheduling using accelerator controller



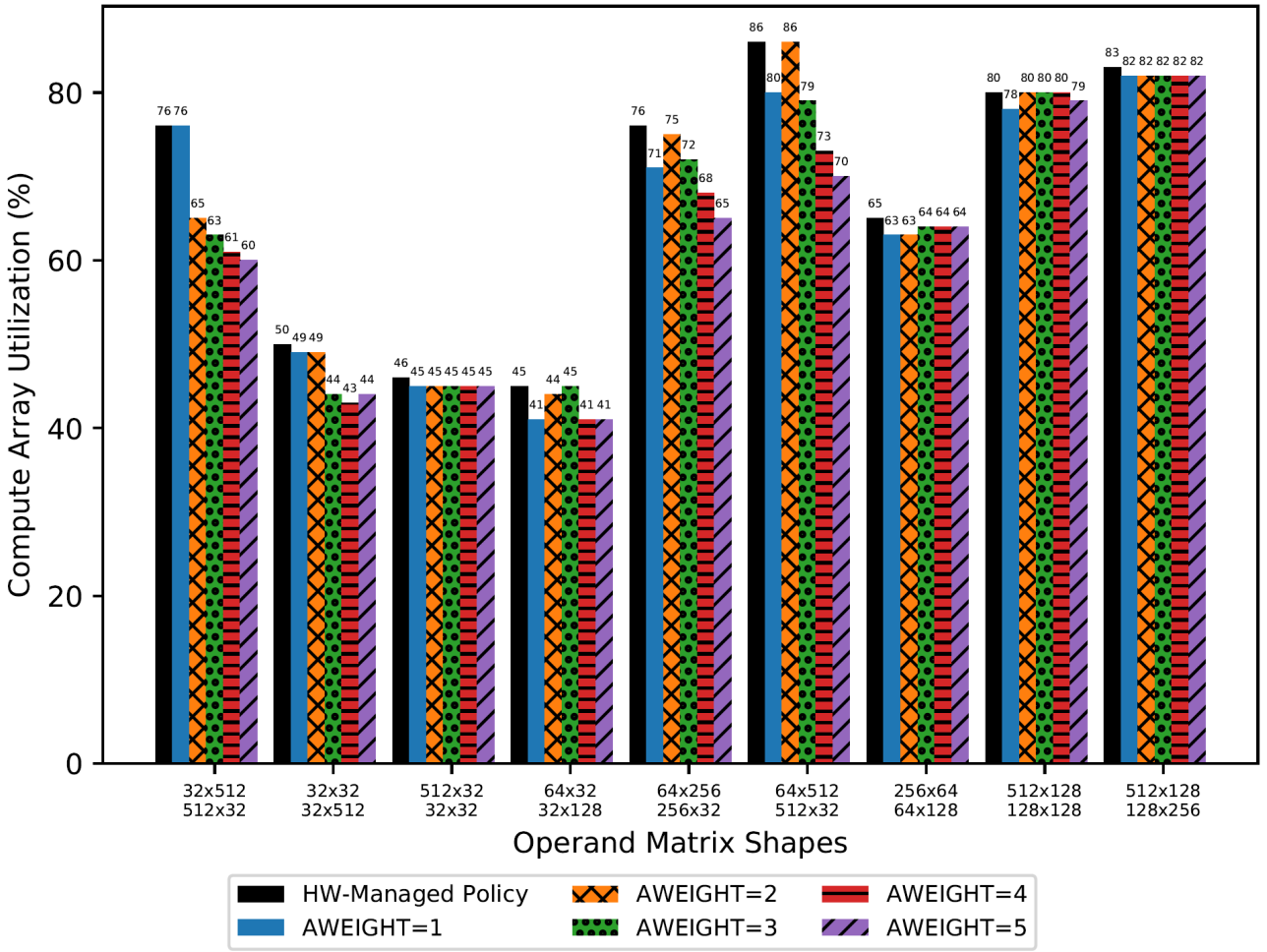
Gemmini Hardware Controller

- Fine-grained instructions (RISC) vs. coarse-grained instructions (CISC)
- Finite-state machine implementation
  - Hardware-managed scheduling
  - Hardware-managed operation dispatch
  - Hardware-managed double buffering
- Scheduling resource allocation managed through software-controller and feedback-controlled arbiters.
- Can we improve the FSM to better handle small, rectangular matrices?



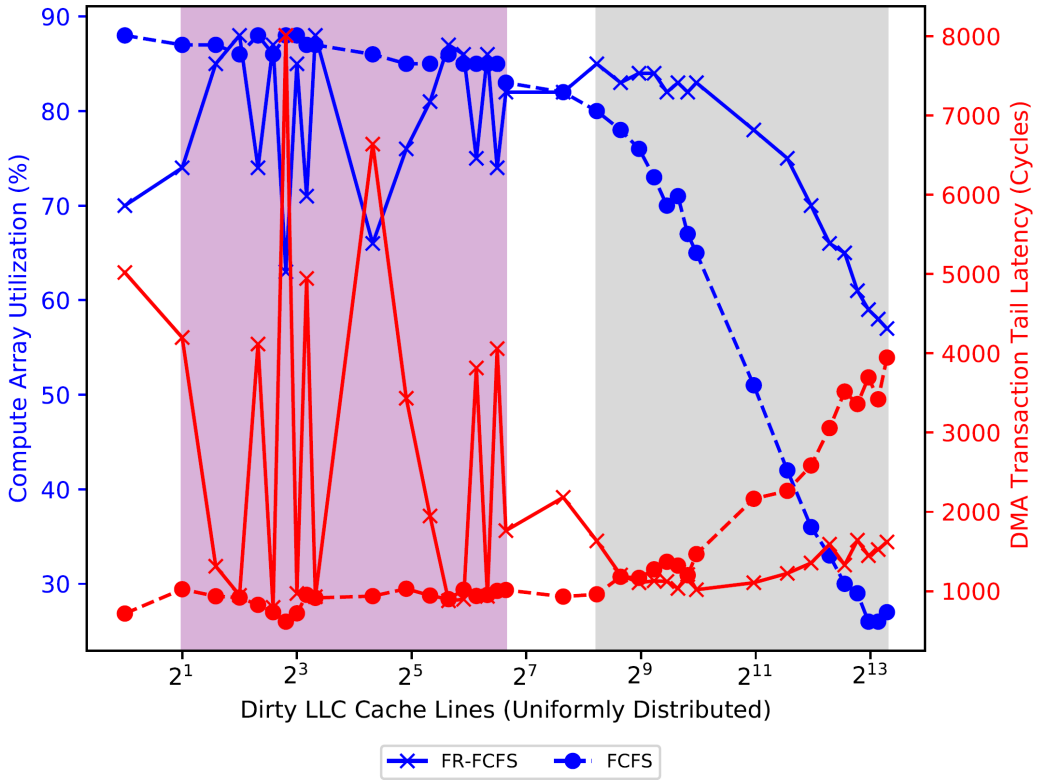
Static Scheduling

- Scheduling of memory load operations (A, B operands) in memory-bound workloads
- Managed in software (Programmable AWEIGHT)
  - Coarse-grained
  - Domain-knowledge
- Managed in hardware (adaptive policy)
  - Based on FSM iterator values
  - 2 muxes and 2 comparators
- Simple hardware policy is sufficient and better



Dynamic Scheduling

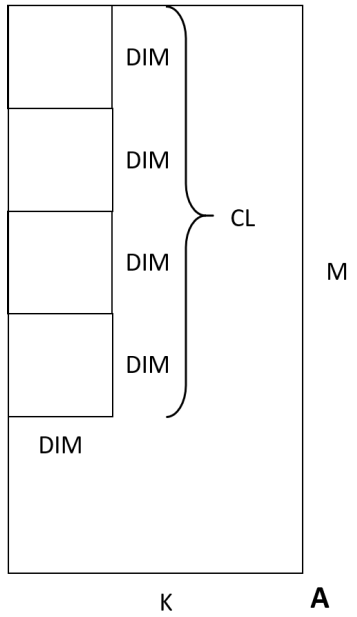
- Variable memory tail-latency
  - Caches
  - DRAM scheduler
  - Fabric
- Double-buffering => in-order execution
  - Decoupled access-execute
  - Double-buffering hides variable latency
- What if the matrix is too small to be double-buffered?
  - Out-of-order execution
  - Micro-threads

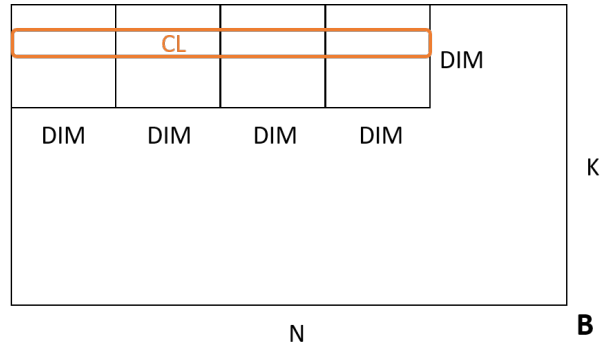


32x1000 times 1000x32 matrix multiplication
With different starting cache state

Dynamic Scheduling

- OoO in accelerator controller
  - Dependencies within the static schedule on a single cache line
  - Load issuing can remain in-order, only execute/store OoO
- Commutativity of accumulation => hardware-controlled micro-threads
  - Allocation of reservation station resources between micro-threads
- Results demonstrate tolerance to tail latency at the beginning of execution, but not at the end of execution
  - Only 2%-10% overall utilization drop due to tail latency, as opposed to 10%-30%

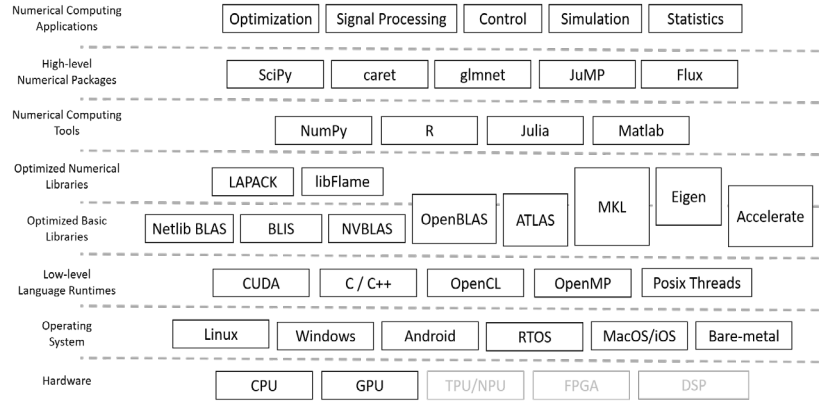


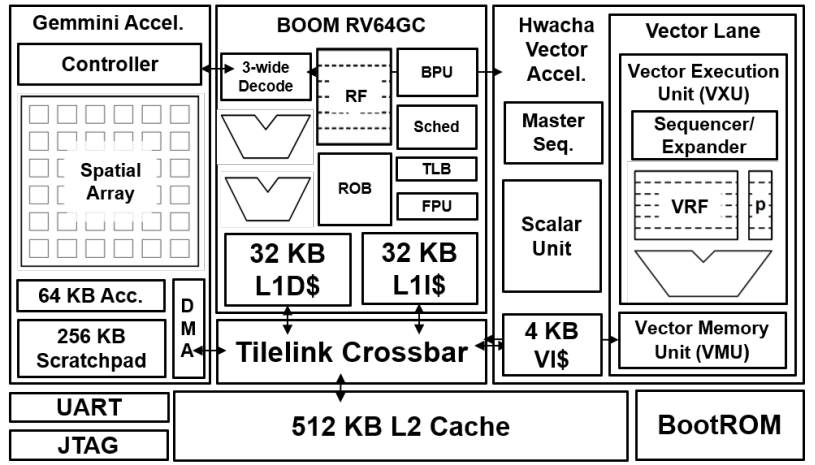


Conclusion (Technical Portion)

Summary

- Custom SoC Design
  - Open-source & generator-based IP
  - Chipyard
  - Multi-flow framework
  - HW/SW co-design
- Customization for Numerical Data Analysis
  - SoC with 1D + 2D data-parallel accelerators
  - Secondary-use of DNN accelerator
  - Software mapping
  - Customization based for small, rectangular matrices for numerical data analysis algorithms





Education and Open Source

- The “non-research” aspects that consumed 90% of time
- Open Source Academic Artifacts
  - Longevity of an academic software artifact beyond the paper deadline
  - User support
  - Documentation
- Education
  - Gemmini in class
  - Chipyard in classes
  - Enabling cross-class collaboration without excessive pre-requisites