AI on Intel

From Tensor Processing Primitives towards Tensor Compilers using upstream MLIR

Alexander Heinecke

Intel Fellow, Intel Parallel Computing Lab

intel®

Disclaimer

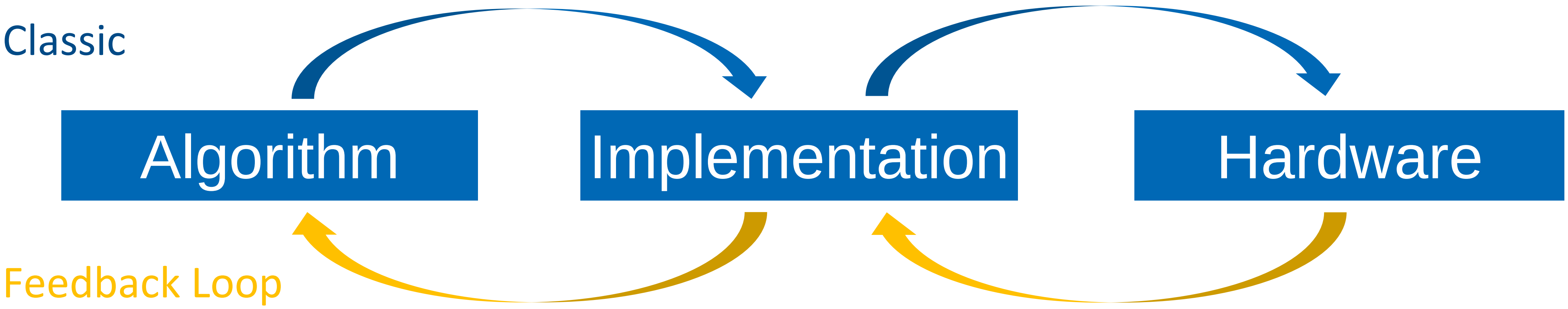
Outline

- Motivation & Parallel Computing Lab Charter
- Tensor Processing Primitives - TPP (micro-kernels for hardware abstraction)
  - CPU
  - GPU ukernel and CUTLASS efforts
- TPP-MLIR for CPU & GPU (a compiler based on standard micro-kernel abstraction)
- Triton-CPU accelerated by TPP
- Summary

Motivation – Parallel Computing Lab Charter

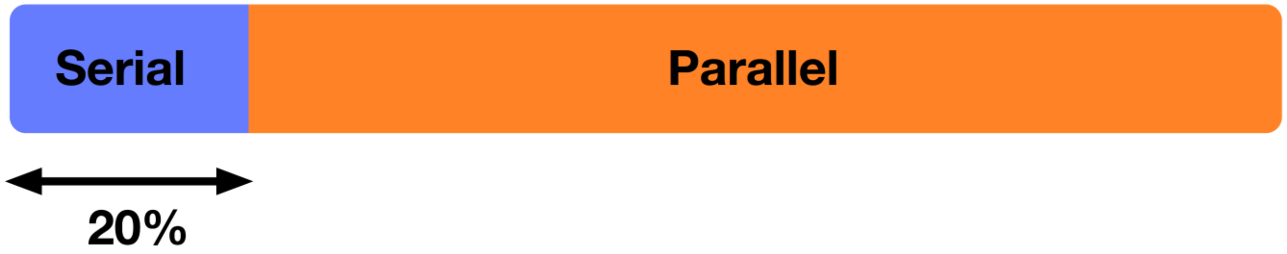
Hardware/Software Co-Design – the next 1000x

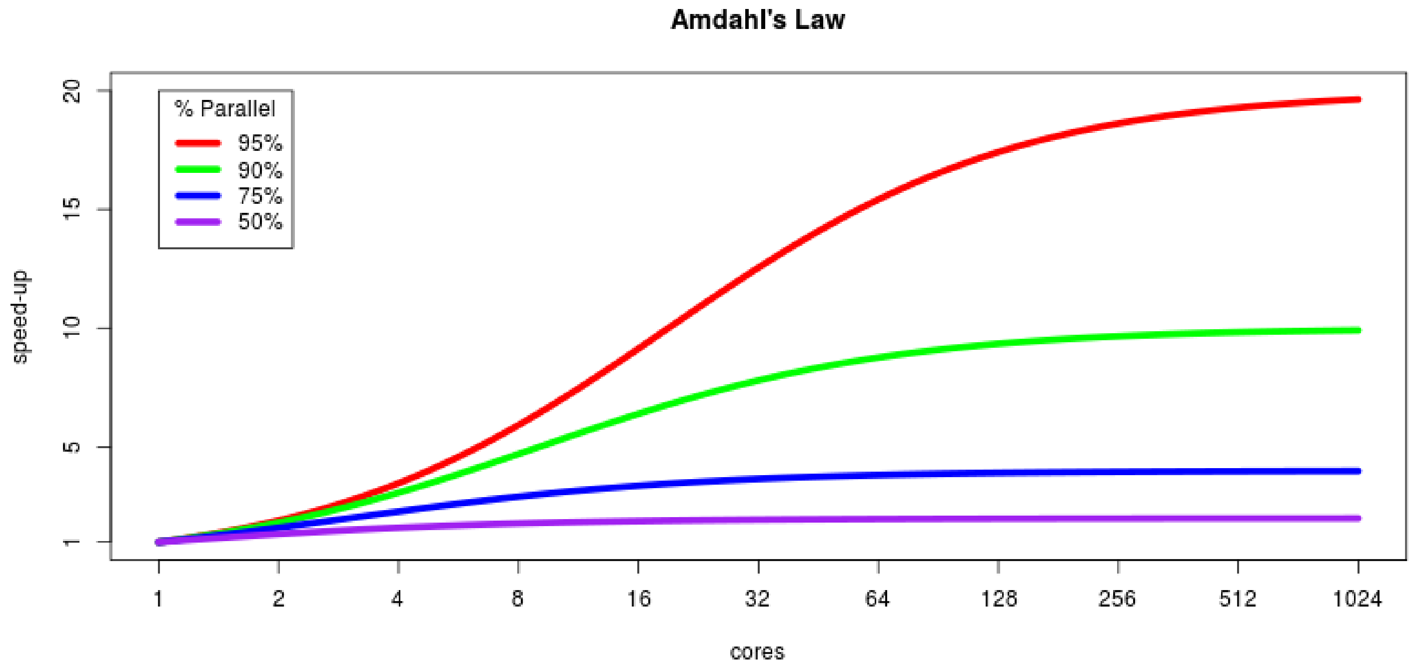
- We are no longer getting higher frequencies
- The only way forward is more cores and even these avenues start to fall off the die size cliff -> wafer-scale, packaging, interconnect
- Architectural Innovation is more important than ever
- Portable, Automated programming, e.g. DSL/JITs



https://cacm.acm.org/magazines/2019/2/234352-a-new-golden-age-for-computer-architecture/fulltext
https://newsroom.intel.com/press-kits/intel-labs-day-2020/

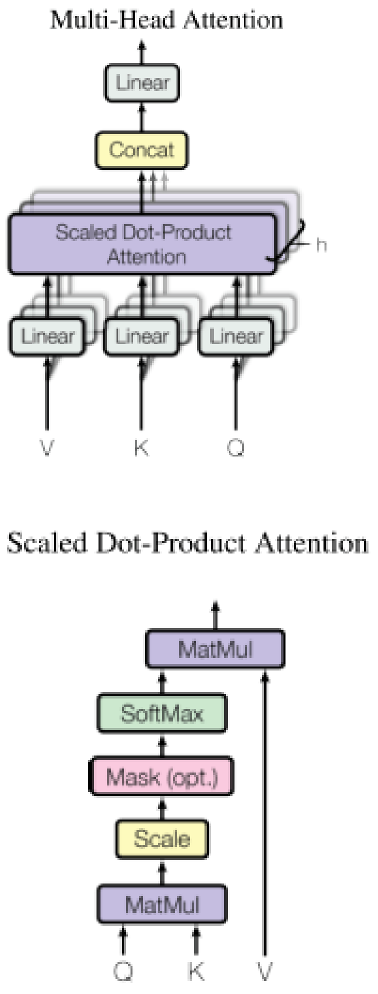
Algorithmic Challenges – Amdahl's Law



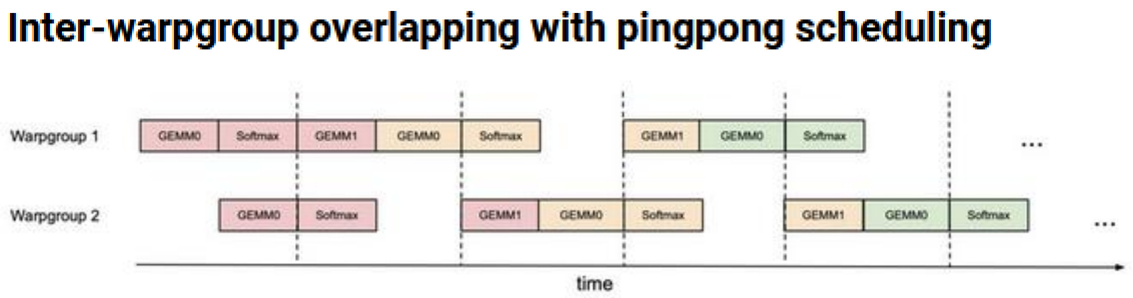


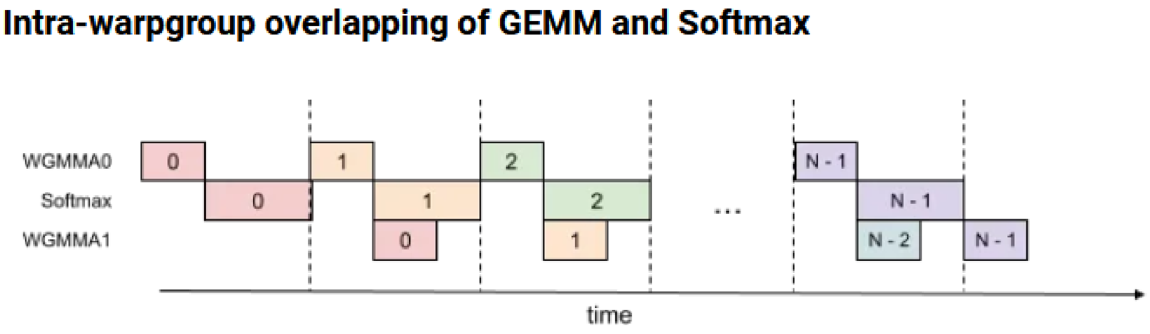
Fused Building Blocks: Attention

- Fuse transposes with GEMM compute
  - Avoid explicit matrix transpose on input
  - Fuse it in BRGEMM kernel (B-trans)
  - If required, perform it on output matrix
- Fuse Scale/Dropout/Softmax/Mask
  - Operation performed on block of output
  - Local to given thread
  - Suitable for tensor-ISA implementation



Pipeline of Operations (DAG Optimizations) for reduced runtime as Linear Part (MatMul) is so fast on modern hardware.
→ Flash Attention





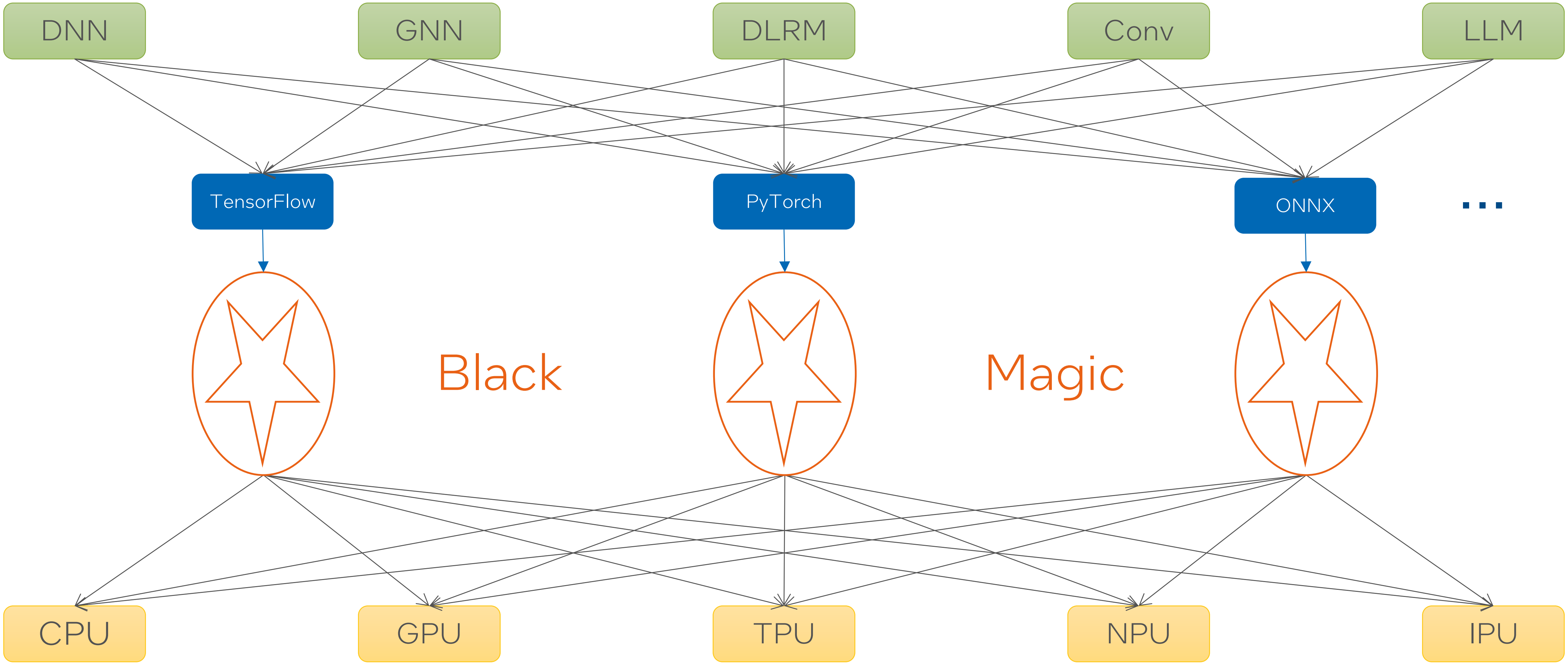
https://tridao.me/blog/2024/flash3/

Tensor Processing Primitives (TPP)

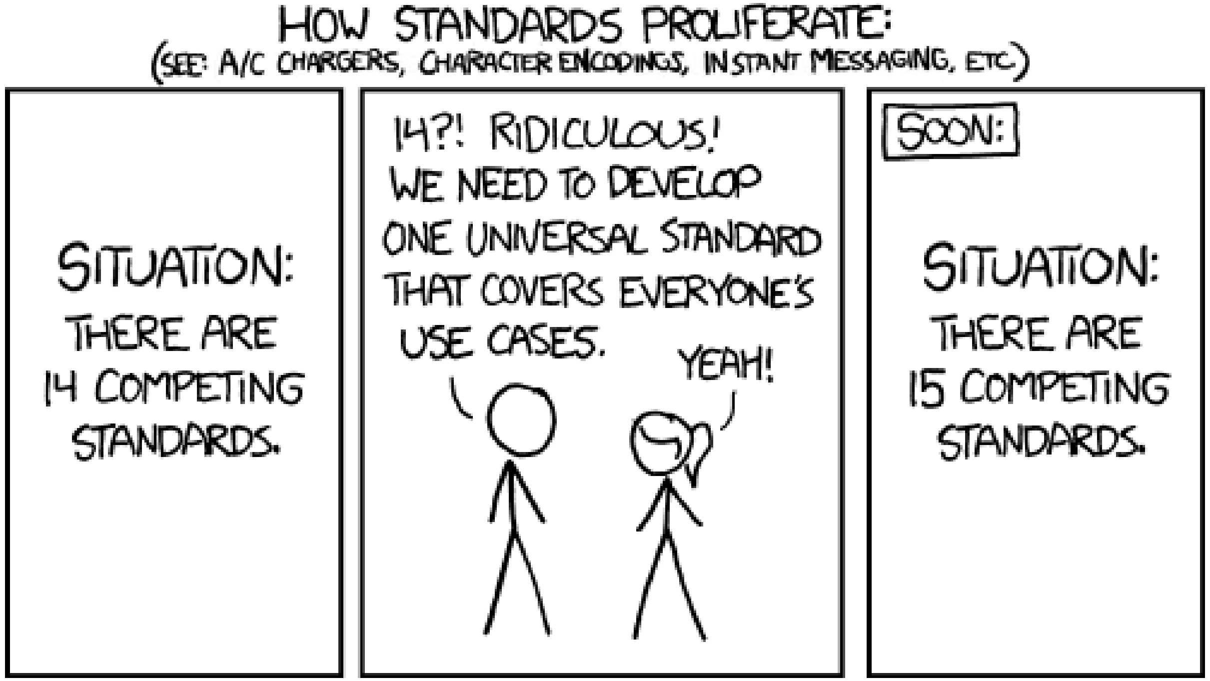
https://arxiv.org/abs/2104.05755

https://github.com/libxsmm/libxsmm

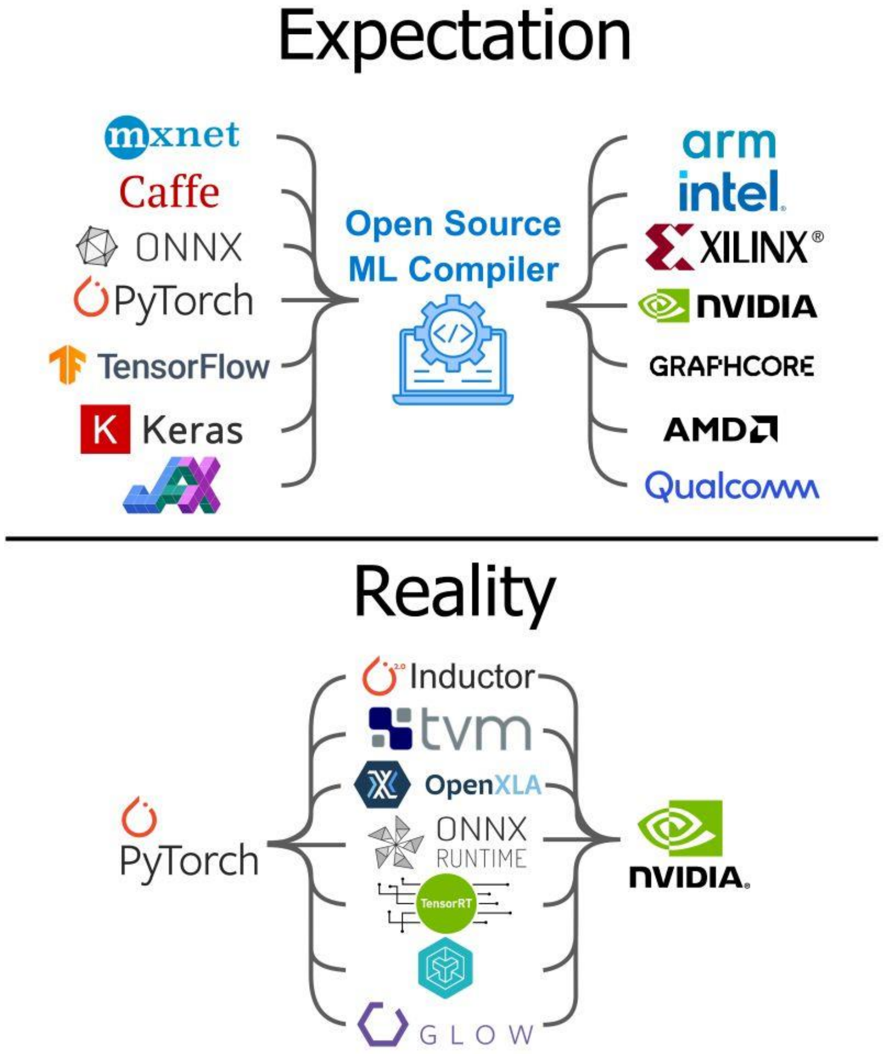
All AI Framework attempt to solve M:N Challenge



Some not so serious Truths



https://xkcd.com/927/

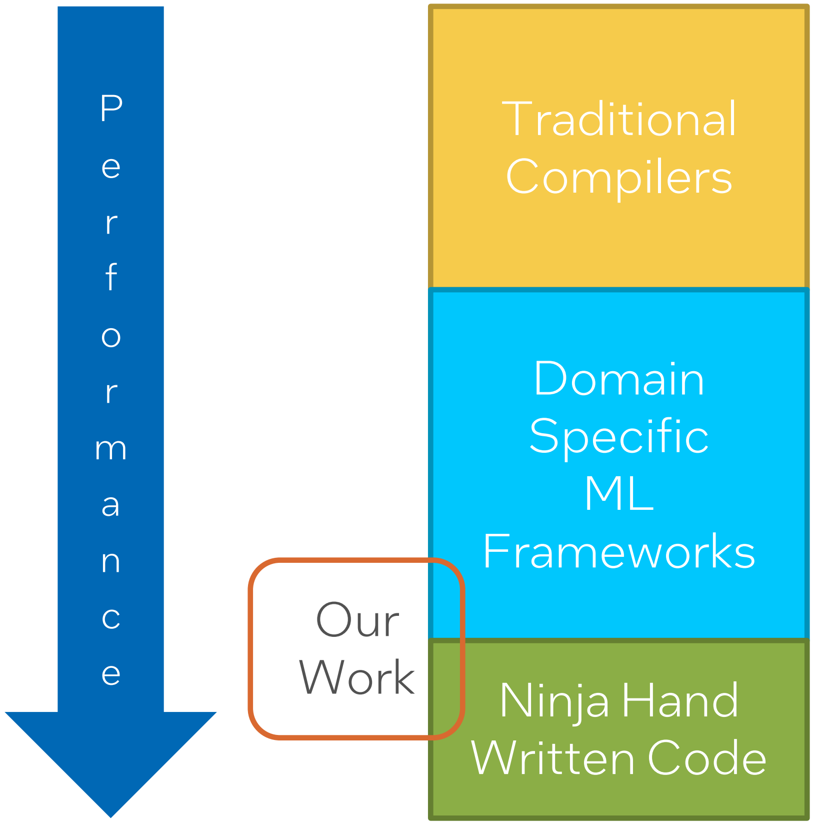


https://www.linkedin.com/posts/matthew-barrett-a49929177_i-think-its-fair-to-say-that-ml-compilation-activity-7185745237049286657-z5_Q/

How to avoid the 15th Standard

- Co-evolve with existing frameworks (ex. PyTorch)
- Collaborate with existing compilers (ex. IREE)
- Promote flexibility & adaptability (ex. cost models)
- Design a common rewrite semantics framework (ex. MLIR-Linalg)

Bridging the Ninja Performance Gap

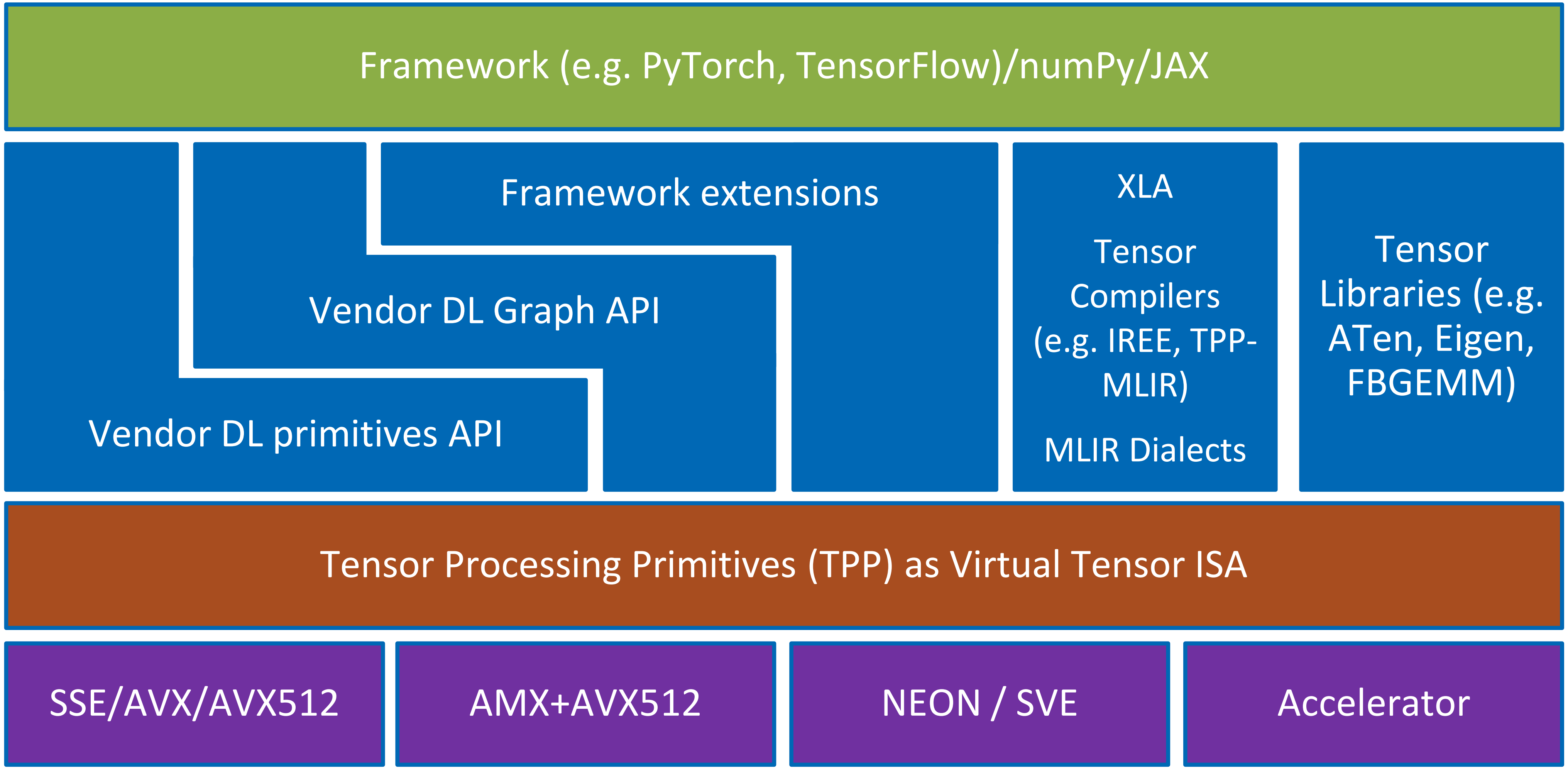


Very good at generic code
Naïve vectorization
Poor hardware utilization
No high-level transformations

Hard-coded pass pipelines
Baked in assumptions
Poor view into micro-architecture

Very efficient hardware utilization
Very hard to generalize
Very hard for large models

Scalable DL software stack



What are Tensor Processing Primitives (TPP)

- Think (BR)GEMM and 2D Operations
- We express every operation in 2D space
  - "Virtual Tensor Instructions": abstraction of AVX?, AMX, Neon, SVE, XPU
  - portable and future proof as SIMD-width can be SW defined
  - Memory-to-memory "instructions" to achieve abstraction from hardware
  - DL ***and*** HPC, everybody who loves Tensors: DL, higher-order FEM, chemistry
- Using Entity (UE) (Human ***or*** Tensorcompiler) can focus on performance in a mostly hardware-agnostic way on:
  - Outer loop schedule
  - (Outer) tensor memory layout
  - (Outer) parallelization
- True Mixed precision by design (in, out, compute)
- Optimal interplay with paradigm shift

Matrix+Vector Programming -> Tensor Programming

GP x86/aarch64
+ vISA TPP unit

CUTLASS
cuTensor

Data Movement and Parallelization ⟷ Well-defined and standardized mapping to ALUs

UE | TPP

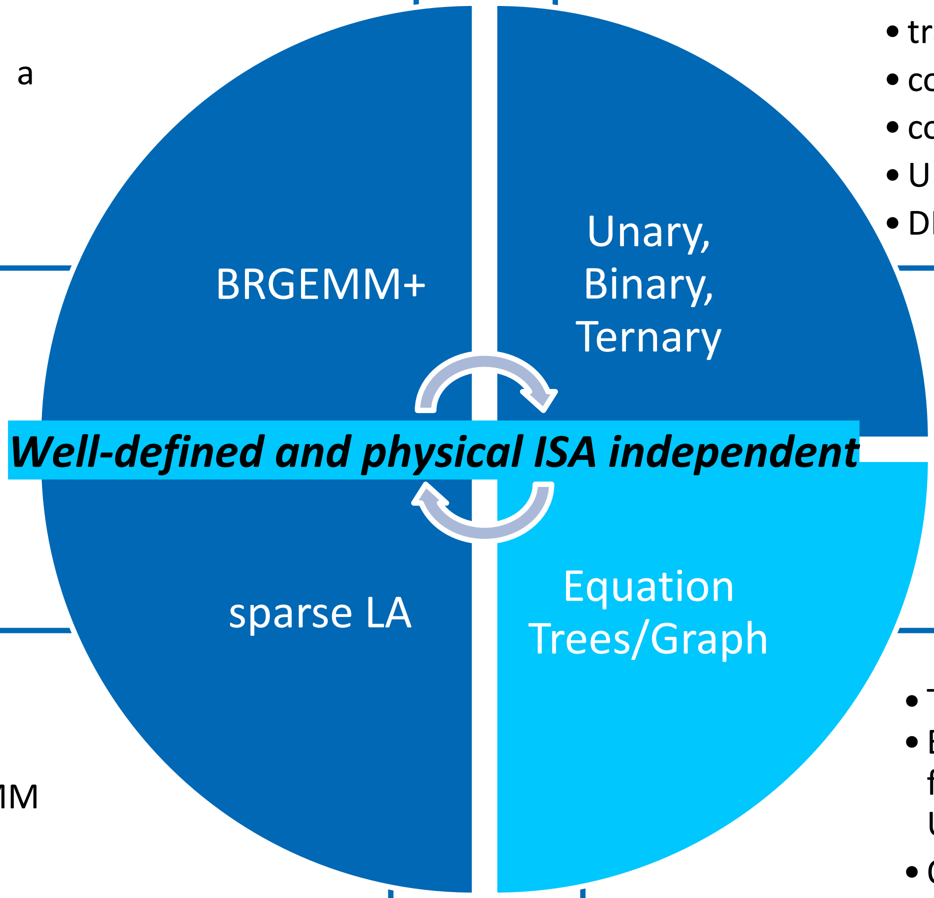
clear separation
of Concerns

TPP Ingredients

https://arxiv.org/abs/2104.05755
https://github.com/libxsmm/libxsmm

- Classic batch-reduce GEMM/MatMul concept
- Spiced up with operations on A, B and a ternary on C
- Allows for reformatting the C matrix

- col/row gather/scatter/replicate
- transform/reorder/transpose
- col/row reduce
- col/row broadcast
- Unary/binary/ternary math incl. blend
- DL specifics (dropout, fast GeLU)



- Sparse LA operations, e.g. sparse GEMM in compact formulation
- Experimental for now

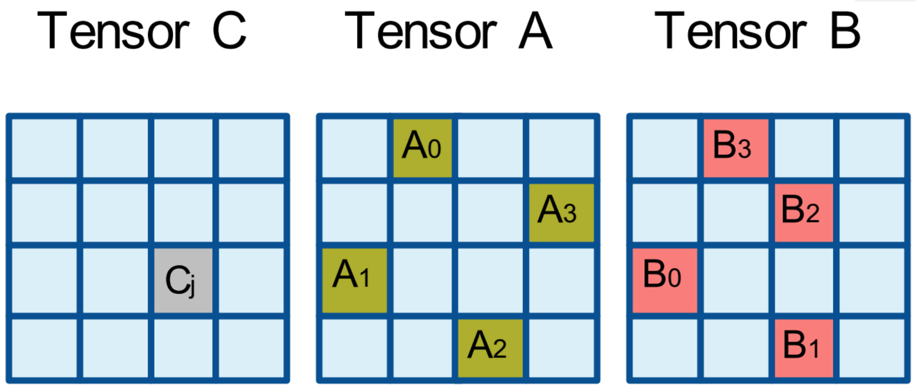
- TPPs are memory-to-memory
- Eqn Trees allow for a post-order traversal fusion of BRGEMM+ and Unary/binary/ternary operations
- Optional, UE can decide if the eqn tree should be used

Anchor stone of TPPs: Ternary BRGEMM for Tensor Contractions

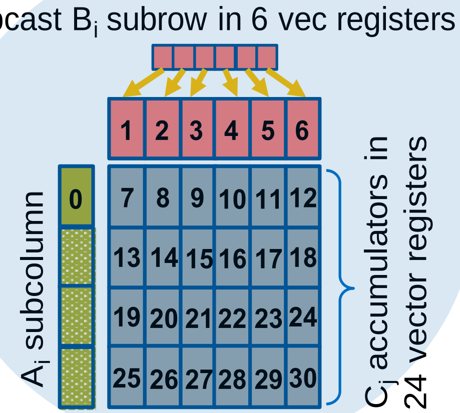
Algorithm 2 The batch-reduce GEMM TPP

Inputs: $A_i^{M \times K}$, $B_i^{K \times N}$ for $i = 0, \ldots, n\text{-}1$, $C^{M \times N}$, $\beta \in \mathbb{R}$
Output: $C = \beta \cdot C + \sum_{i=0}^{n-1} A_i \times B_i$

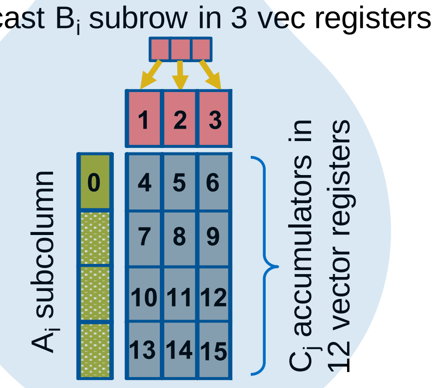
```
for i_n = 0 ... N − 1 with step n_b do
    for i_m = 0 ... M − 1 with step m_b do
        acc_regs ← load_generic m_b × n_b C-subblock_{i_m,i_n}
        for i = 0 ... n − 1 with step 1 do
            for i_k = 0 ... K − 1 with step k_b do
                ▷ Outer product GEMM microkernel
                acc_regs += A_i sub-panel_{i_m,i_k} × B_i sub-panel_{i_k,i_n}
        C-subblock_{i_m,i_n} ← store_generic acc_regs
```



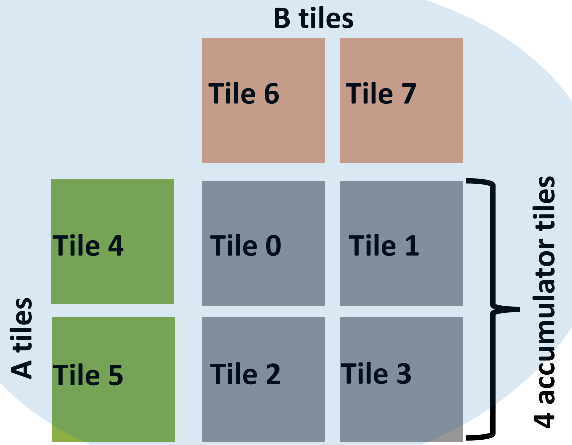
$$C_j = \beta * C_j + \alpha \sum_{i=0}^{N-1} A_i * B_i$$



Microkernel with 32 vector registers (e.g. Intel with avx512, Arm Neoverse)



Microkernel with 16 vector registers (e.g. Intel/AMD with avx2)



Microkernel with 2D register file (e.g. Intel with AMX)

Blueprint of Primitives via TPPs

Most of Developers (Libraries & applications)

Loops around Unary/Binary/Ternary/Equations of TPP

(e.g. tensor tiling, cache blocking, parallelization)

A handful of experts

Unary/Binary/Ternary/Equation TPPs before tensor contraction

Tensor contraction via the ternary BRGEMM TPP

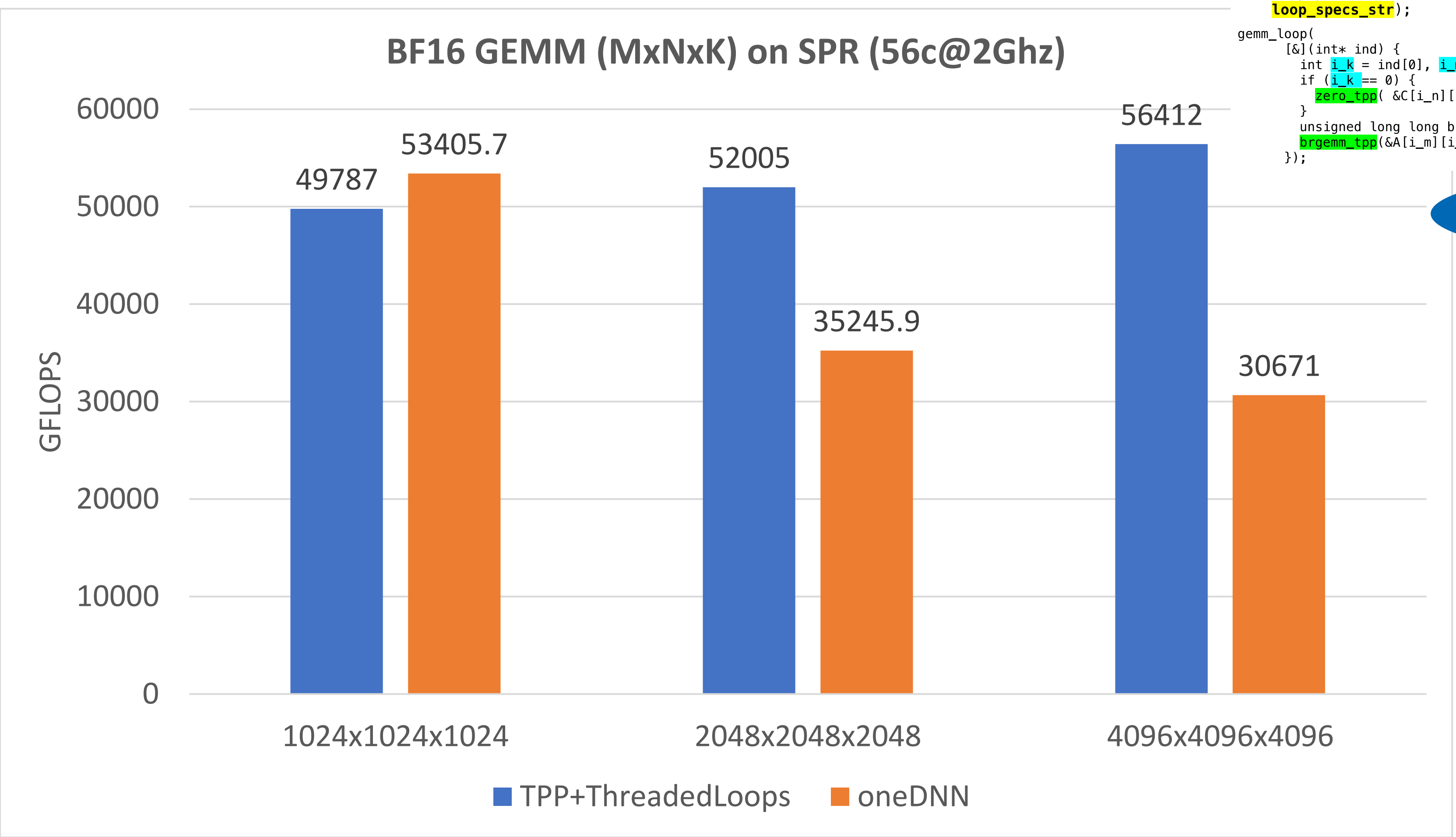
Unary/Binary/Ternary/Equation TPPs after tensor contraction

BF16 Matrix Multiplication on 56c SPR

https://arxiv.org/pdf/2304.12576

- Specific instantiations of loop nest is governed at runtime by a single param (loop_spec_str)
- Trivial auto-tuning on the loop_spec_string – 0 lines of code change in user code
- Same code for all platforms and precisions !
- oneDNN GEMM does *not* support blocked layout for A, thus degraded performance

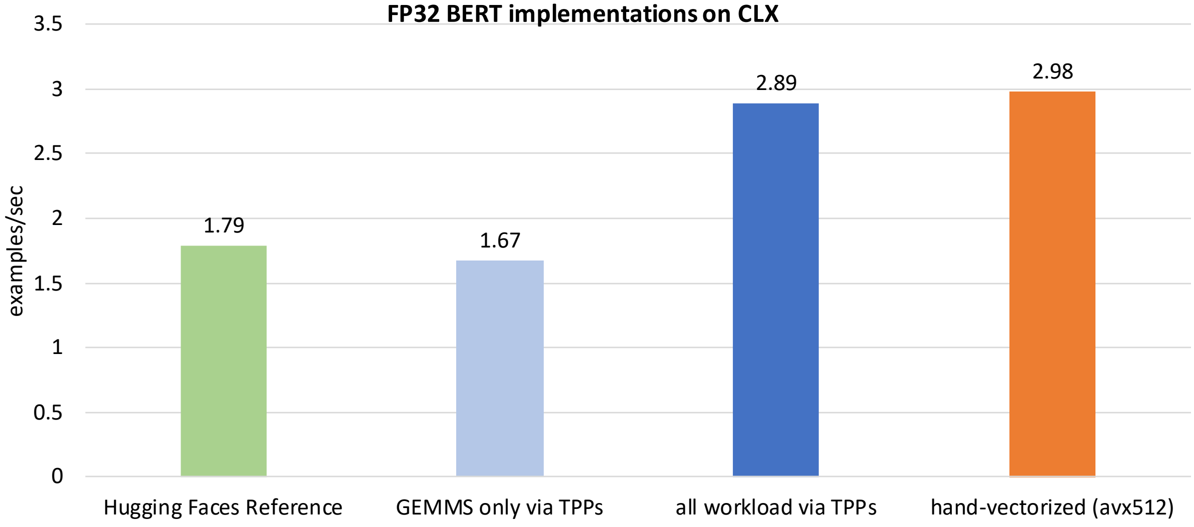
```
auto gemm_loop = ThreadedLoop<3>({
    LoopSpecs{0, Kb, k_step, {l1_k_step, l0_k_step}},   // Logical K loop specs
    LoopSpecs{0, Mb, m_step, {l1_m_step, l0_m_step}},   // Logical M loop specs
    LoopSpecs{0, Nb, n_step, {l1_n_step, l0_n_step}}},  // Logical N loop specs
    loop_specs_str);
gemm_loop(
      [&](int* ind) {
        int i_k = ind[0], i_m = ind[1], i_n = ind[2];
        if (i_k == 0) {
          zero_tpp( &C[i_n][i_m][0][0]);
        }
        unsigned long long brcount = k_step;
        brgemm_tpp(&A[i_m][i_k][0][0], &B[i_n][i_k][0][0], &C[i_n][i_m][0][0], &brcount);
      });
```

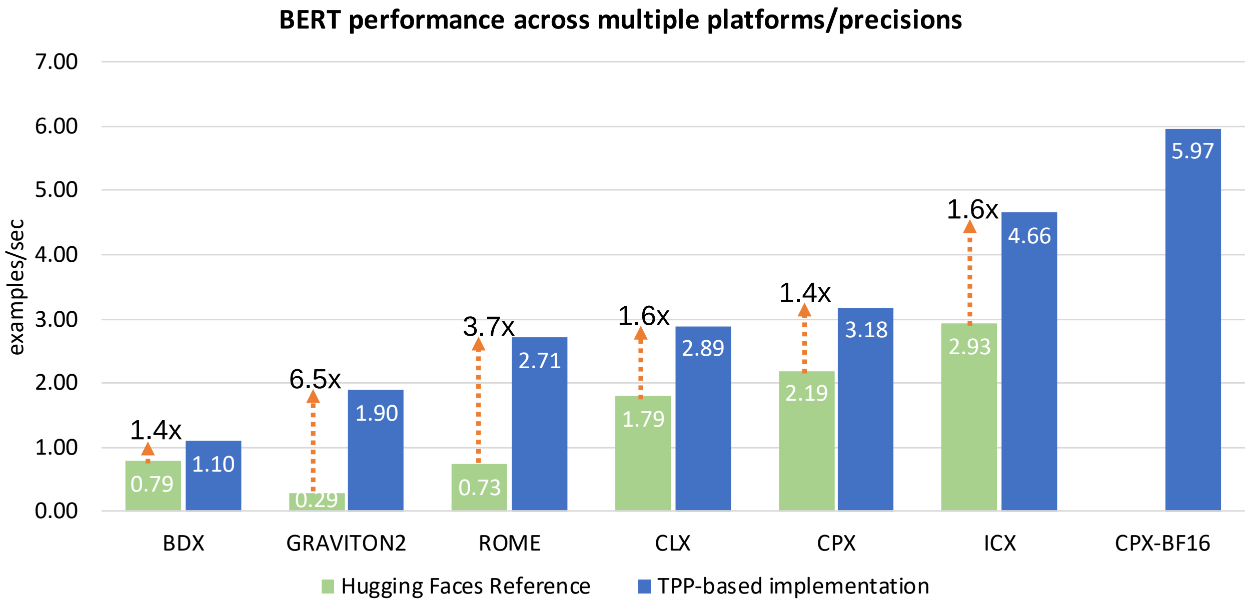


Penguin’s Programming World

- **Logically** describe the loop nest
- **Express the computation** using the logical indices and TPP
- **Resembles** the for the (AI) programmer **familiar CUDA/CUTLASS programming paradigm** on CPU and GPU
- **Exactly the same user code for all platforms and compute precisions**
- This framework **naturally** lends itself to **auto-tuning / AI guide tuning**.
- **Efficient chaining of TPPs** without dealing with Polish Notations and lengthy APIs.

BERT Large Fine-tuning Performance





- TPP based BERT matches the performance of SOTA hand-vectorized and non-portable code
- Outperform Hugging Faces reference implementation up to 6.5x
- Multiple precisions and portable across multiple platforms without code changes

TPP GPU Efforts @Intel

- GPU: CUTLASS for SYCL:
  - https://github.com/codeplaysoftware/cutlass-fork/blob/sycl-develop/examples/sycl/pvc/pvc_bfloat_dpas_gemm_cute.cpp
- GPU: ukernels in oneDNN (can be extended if needed)
  - https://github.com/oneapi-src/oneDNN/tree/main/src/gpu/intel/microkernels
  - SDPA (Scaled Dot Product Attention) using ukernels
    - https://github.com/oneapi-src/oneDNN/tree/main/src/gpu/intel/ocl

TPP-Like efforts outside of Intel

GPU

- Nvidia CUTLASS
  - https://github.com/NVIDIA/cutlass
- OpenAI Triton
  - https://github.com/triton-lang/triton
- AMD Composable Kernels:
  - https://github.com/ROCm/composable_kernel
- ThunderKittens (Stanford)
  - https://github.com/HazyResearch/ThunderKittens

CPU

- ARM Kleidi
  - https://gitlab.arm.com/kleidi/kleidiai

TPP-MLIR

https://arxiv.org/abs/2404.15204v1

https://github.com/plaidml/tpp-mlir

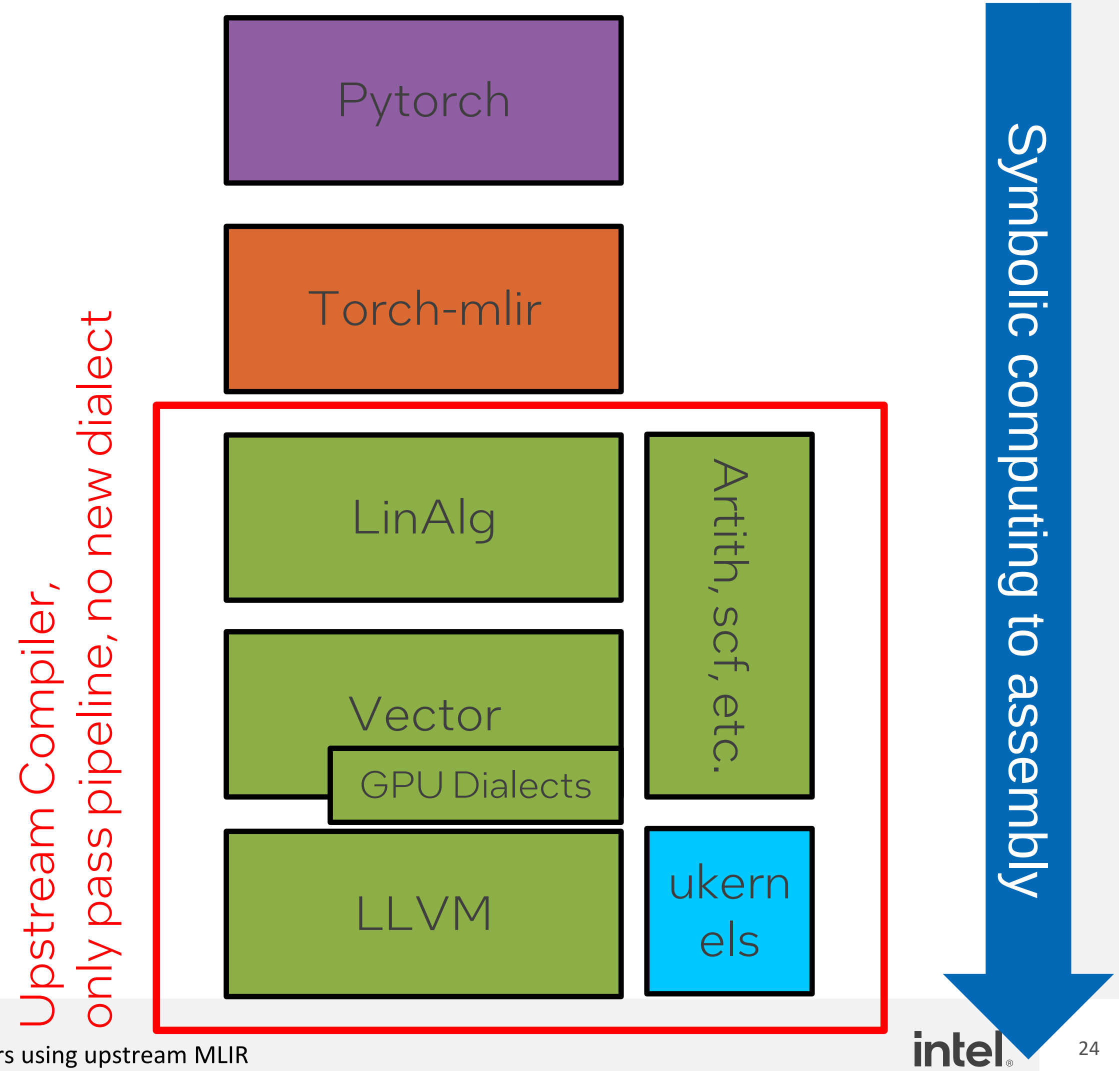
Goal: “clang for AI” in MLIR*

https://github.com/pytorch/pytorch/
https://github.com/llvm/torch-mlir
https://github.com/llvm/llvm-project

Goals

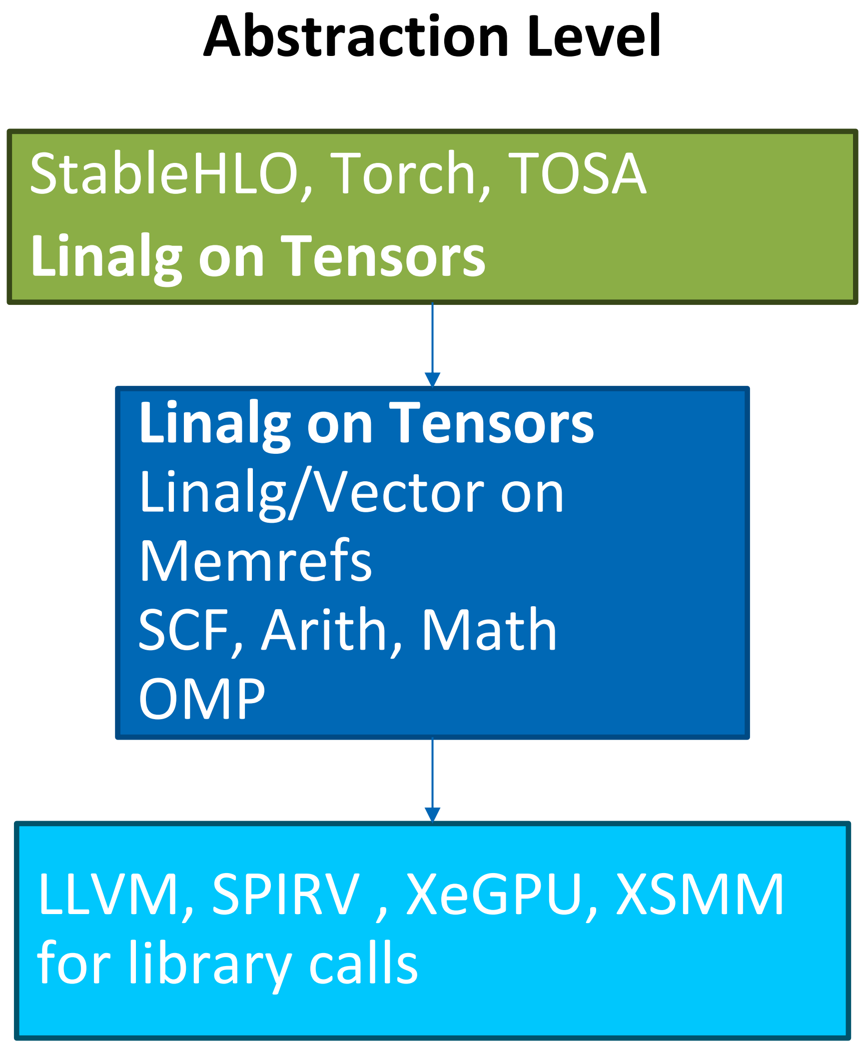
- Standardizing torch->MLIR->hardware lowering by establishing the “beaten path” by an upstream compiler, e.g. llvm incubator project
- Dialects, Passes, Transforms, e.g. stay in MLIR (llvm-project) and ideally the compiler is just the glue-code with the pass pipeline
- Compiler starts with LinAlg as the highest-level dialect
- Focus on x86 for now (it’s everywhere), but run on GPUs as well.

*Multi Level Intermediate Representation

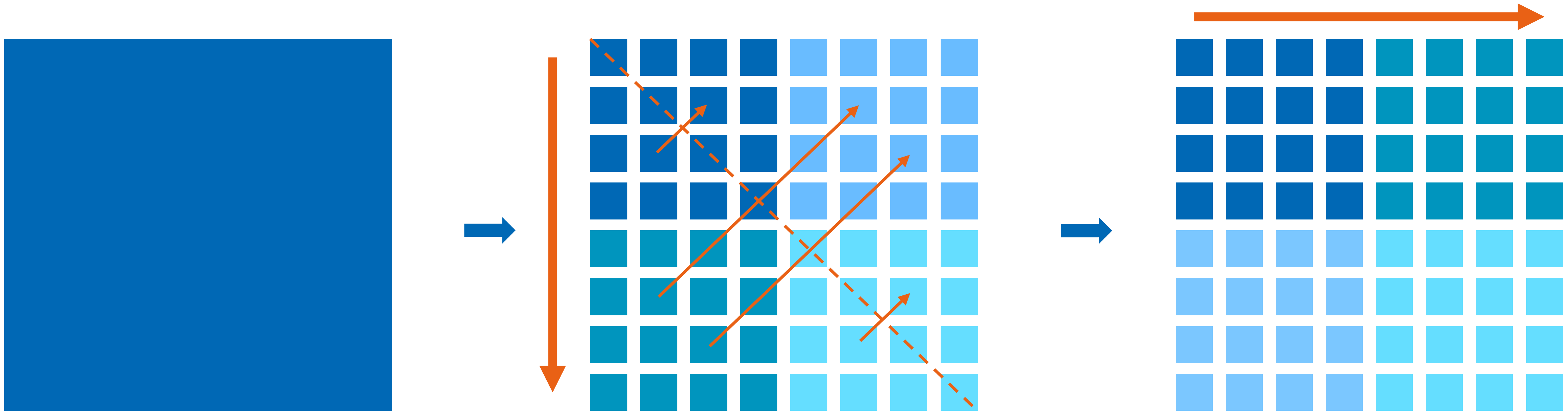


Overview

- Ingress
  - Whole tensor ops
  - Language semantics (graph)
- Transform
  - Graph sharding, placement
  - Tiling, blocking, cache fusing
  - Loop reordering, k-splitting
  - Register level fusing
- Lowering
  - Optimal SIMD/SIMT code
  - Linking, Offloading



Packing shapes



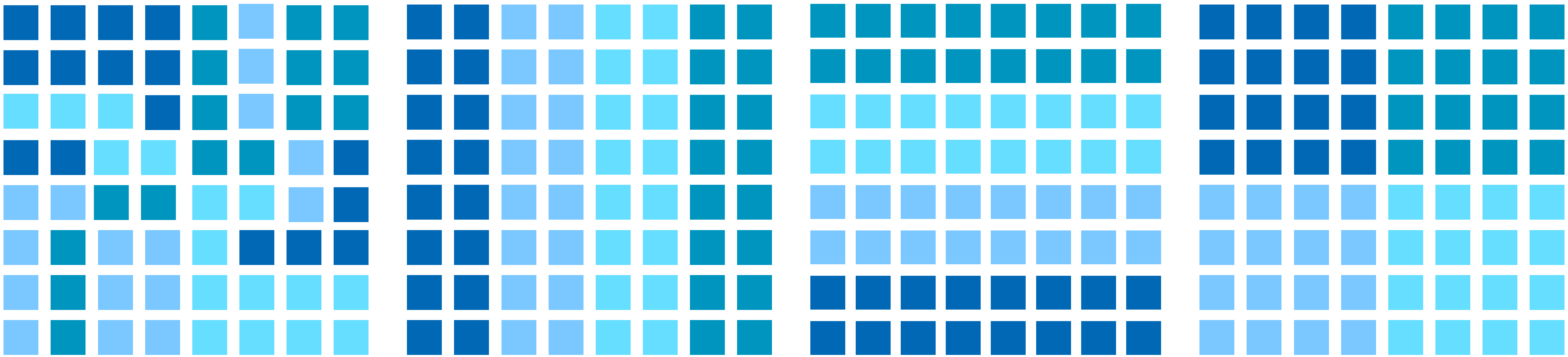
2D shape
M x N

4D shape
(MB x NB) x (mb x nb)

Block-transpose
(NB x MB) x (mb x nb)

B matrix column access becomes row access (cache-friendly), transpose is fast (block copy)
A, B and C are now on the same access pattern
$O(n^2)$ packing cost pays off with $O(n^3)$ GEMM access savings

Parallelisation Strategies



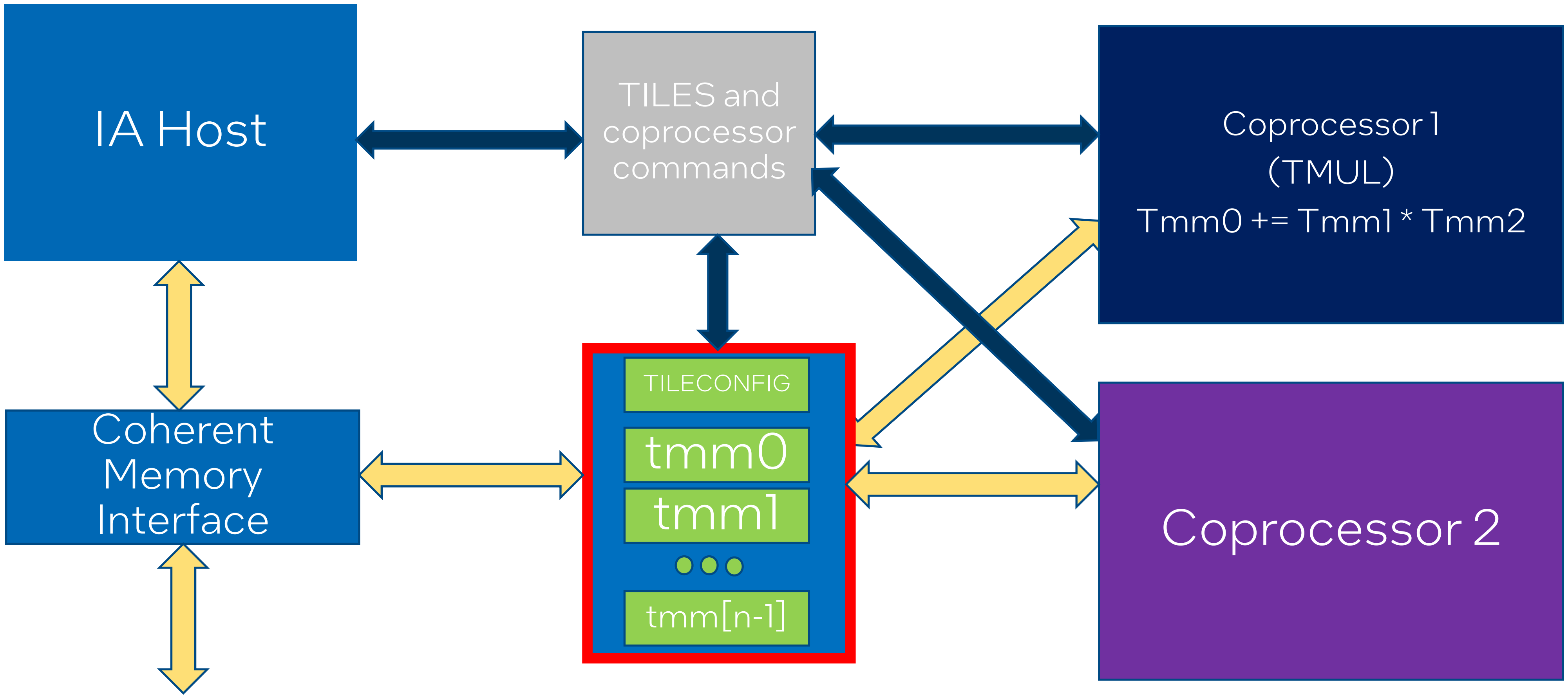
Random
`scf.parallel` has no defined rule

Row / Column
Block tiles by multiple rows or columns
Still not optimal for multi-threaded

2D parallel
Rectangular blocking
Multi-thread aware
Minimizes data moves

Increased cache awareness

Intel® AMX High-Level Architecture



New state to be managed by OS

Commands and status delivered synchronously via TILE/accelerator instructions

Dataflow – accelerators communicate to host through memory

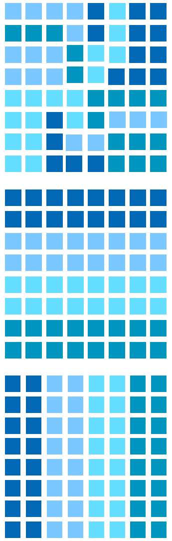
2D Parallel + AMX Tile Config Hoisting

M x N iterations
T threads

T << M x N

```
FORALL (M, N) {
  BRGEMM(A′, B′, C′)
  ADD(C′, bias)
  ReLU(C′)
}
```

Which distribution?



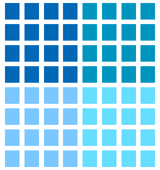
```
FORALL (M, N) {
  AMXSetup(Tm, Tn)
  BRGEMM(A′, B′, C′)
  ADD(C′, bias)
  ReLU(C′)
 AMXReset(Tm, Tn)
}
```

Tile Setup runs on every tile,
even if on the same thread

2D Parallel

```
FORALL (M/m, N/n) {
 FOR (m) {
  FOR (n) {
   AMXSetup(Tm, Tn)
   BRGEMM(A′, B′, C′)
   ADD(C′, bias)
   ReLU(C′)
  AMXReset(Tm, Tn)
  }
 }
}
```

Each thread on
this distribution!



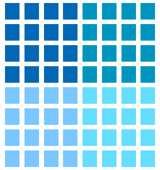
Tile Config per thread

```
FORALL (M/m, N/n) {
AMXSetup(Tm, Tn)
 FOR (m) {
  FOR (n) {
   BRGEMM(A′, B′, C′)
   ADD(C′, bias)
   ReLU(C′)
  }
 }
AMXReset(Tm, Tn)
}
```

One setup
per thread

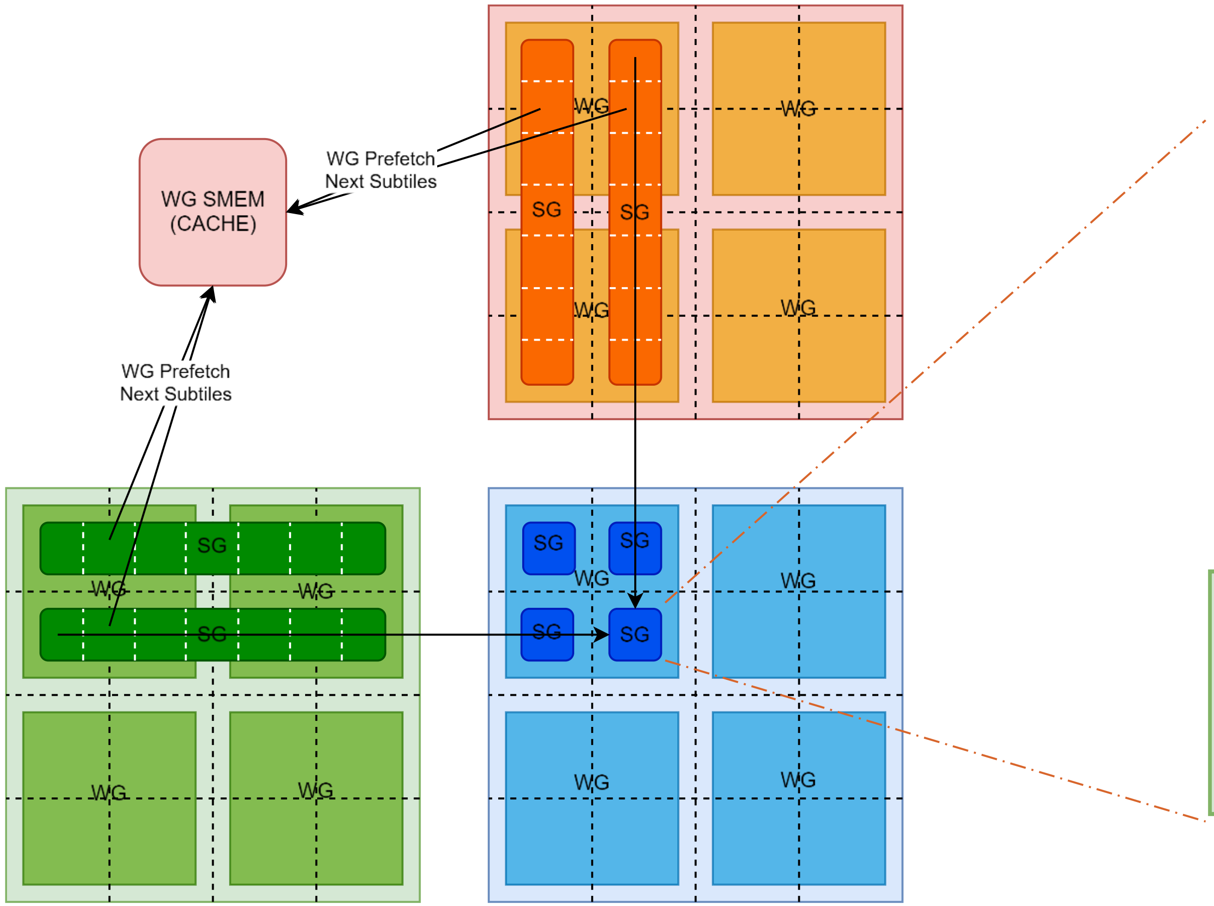
One reset
per thread

Each thread on
this distribution!

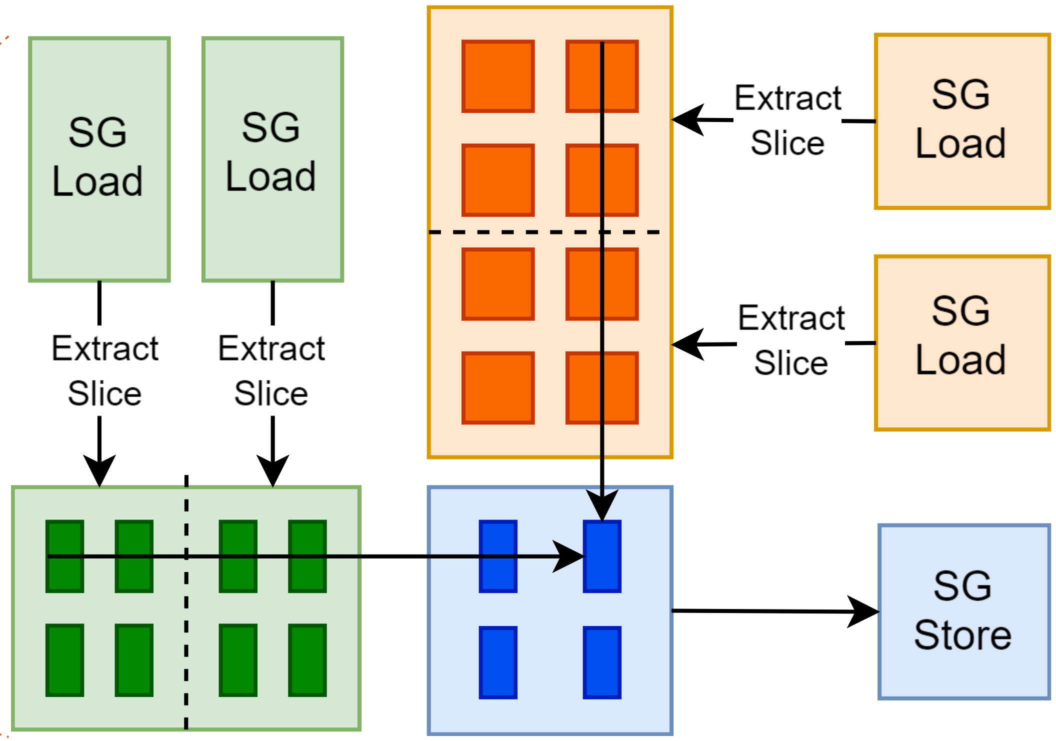


GEMM on Intel Max GPU (Ponte Vecchio / PVC)

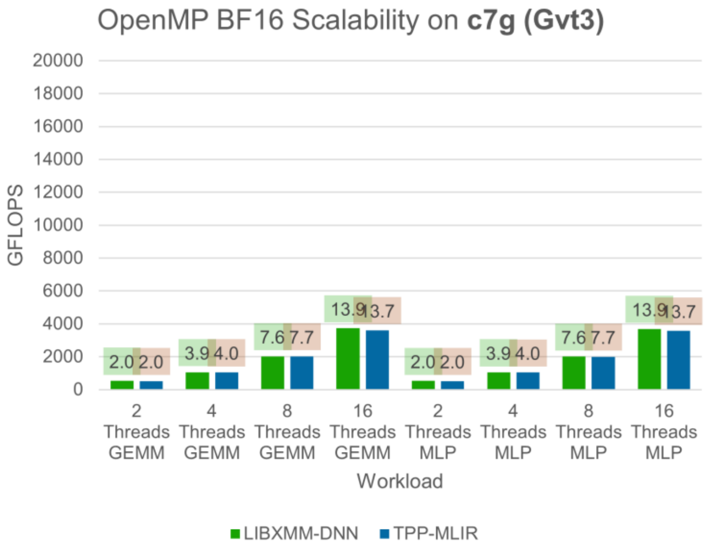
GEMM to GPU Work- and Subgroups

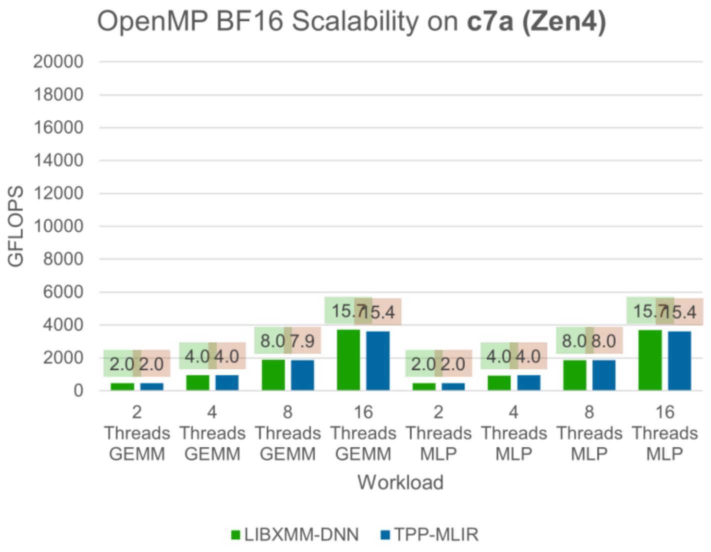


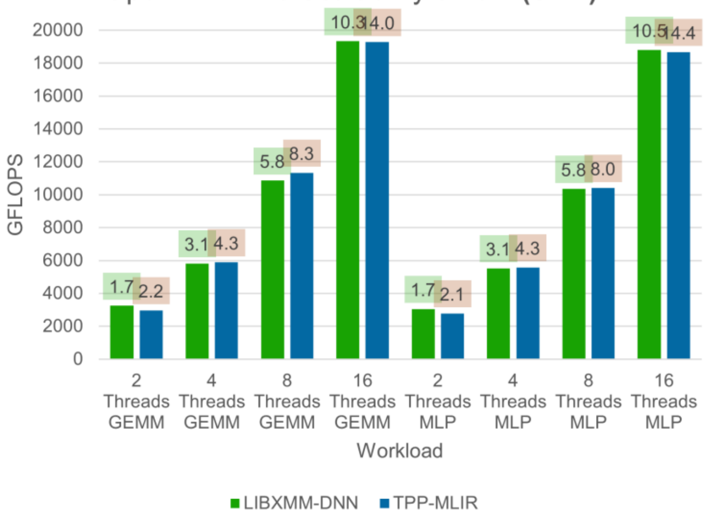
GEMM Subgroup tile as Systolic Array tiles



TPP-MLIR -- Multi-Threaded BF16 (IR gen, pre-packed 4D)

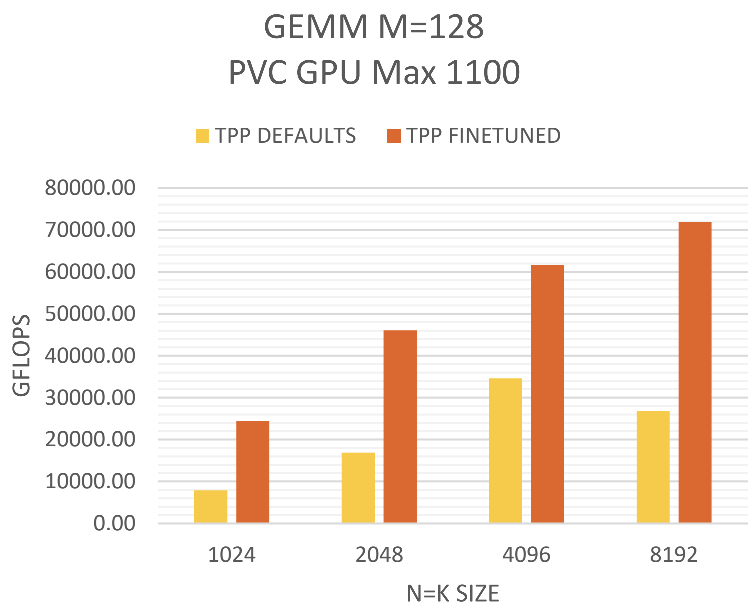


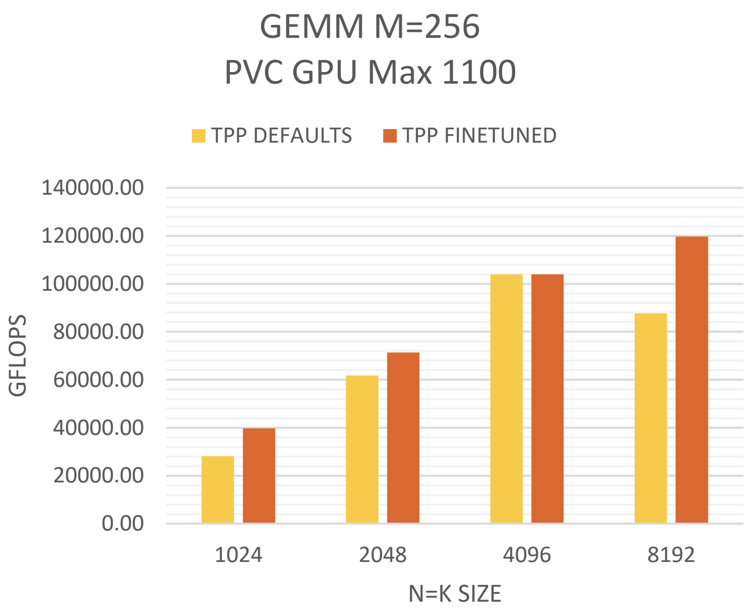


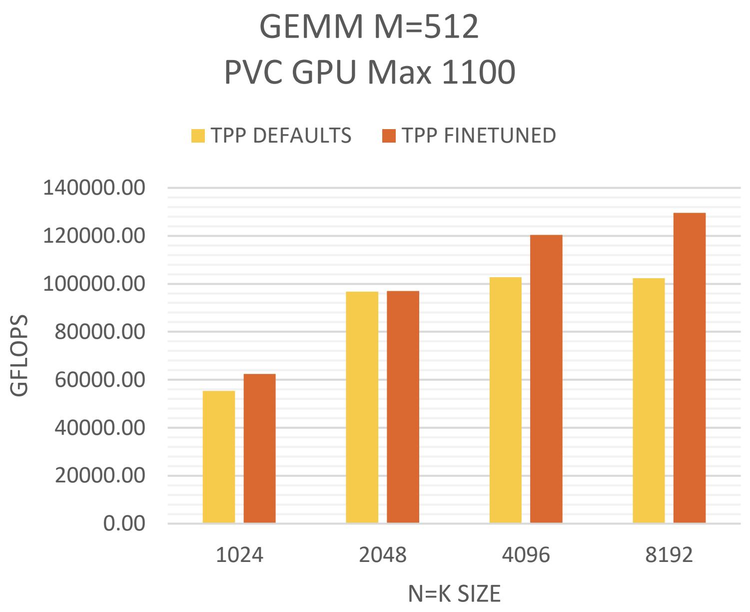


- 2D parallelization using *optimal* blocking depending on the number of threads
- Almost perfect scalability on Zen4, good scalability on Graviton 3
- SPR shows the same final performance as Ninja-Coded applicaitons

TPP-MLIR Intel Max GPU Performance GEMM FP16





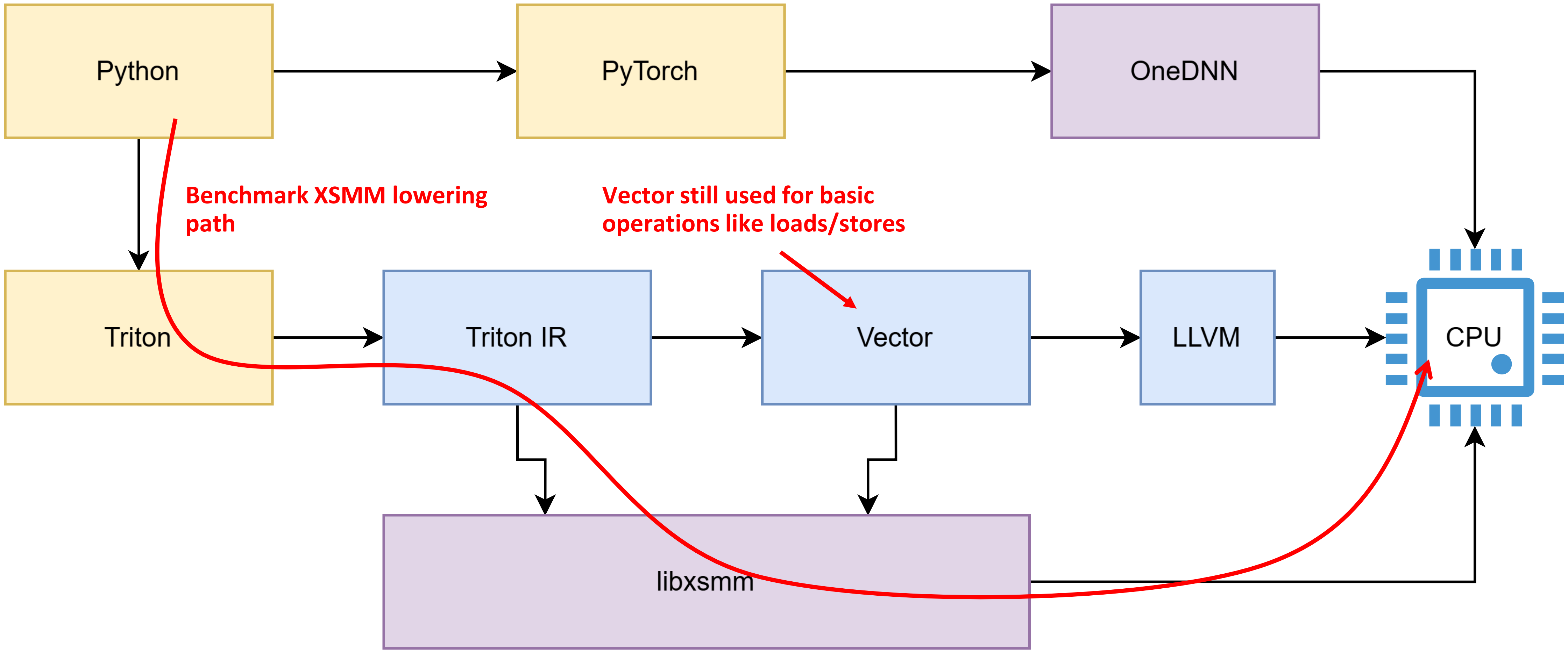


- GEMM kernel tuning parameters
  - Workgroup tile sizes – default: 128x128 – used tuning values: 64, 128, 256
  - Subgroup tile sizes – default: 32x32 – used tuning values: 16, 32, 64
  - Reduction dimension tiling – default: 32 – used tuning values: 16, 32, 64
- Kernel parameter selection is crucial for good performance
  - Requires cost model and heuristics
- Lowering allows for quick GEMM kernel finetuning

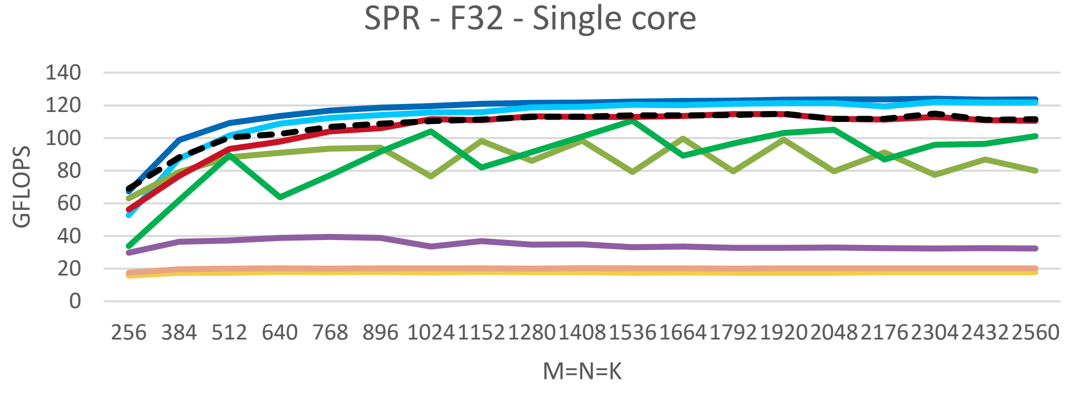
Triton-CPU with TPP

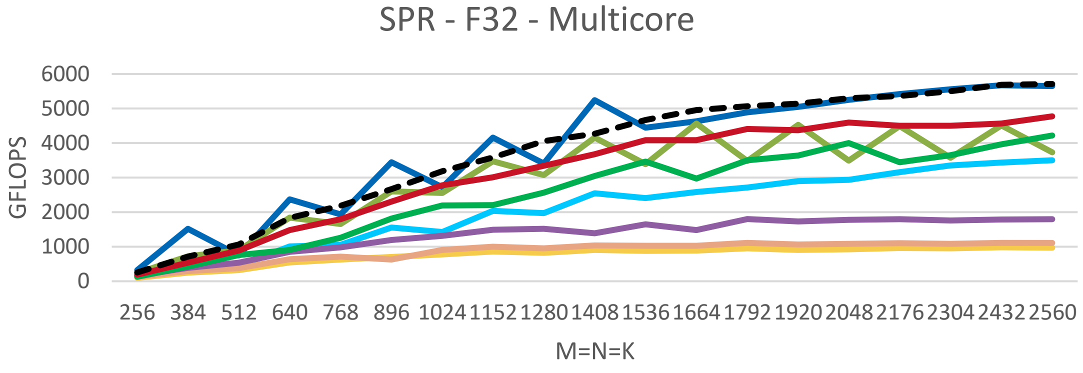
https://github.com/plaidml/triton-cpu

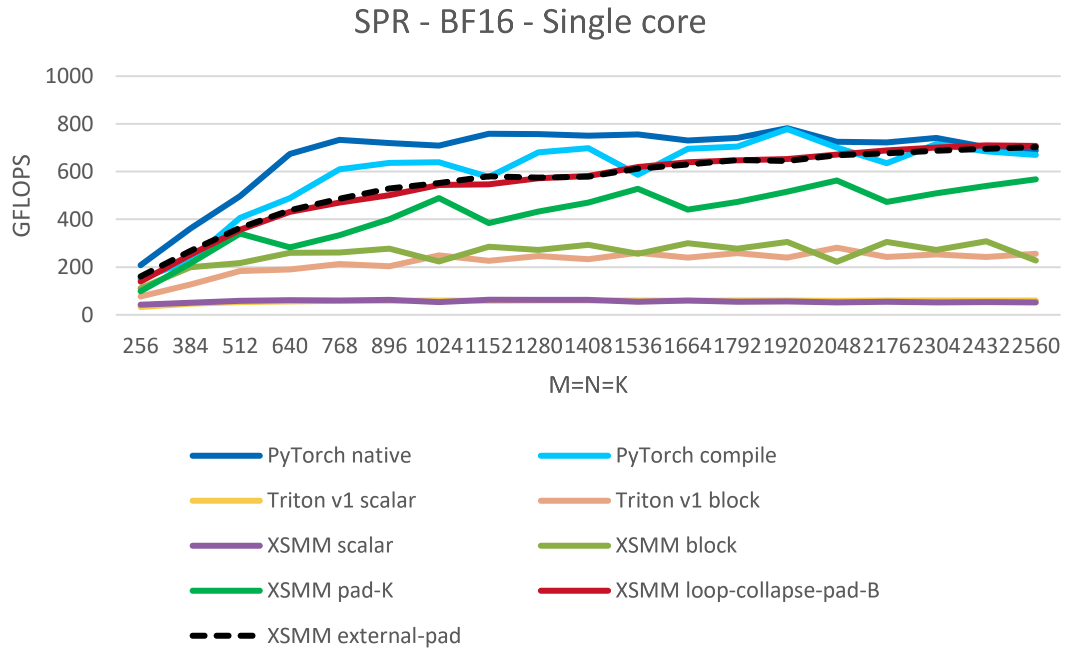
Triton-CPU Pipeline

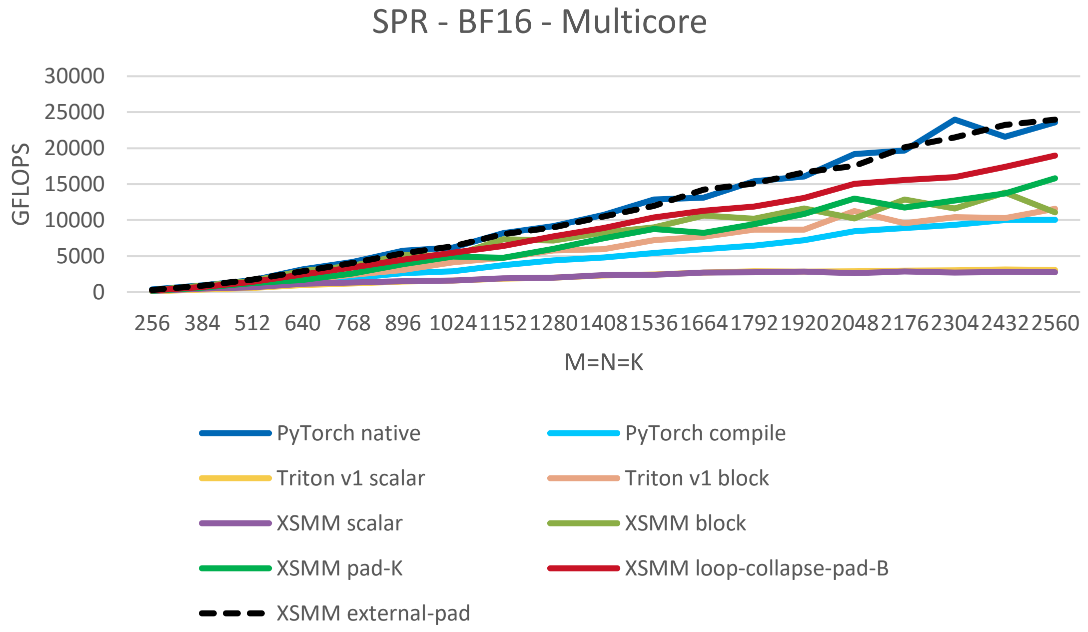


Performance – 5th Gen Xeon









What is needed for high performance Triton-CPU

- Robust and performant infrastructure
  - Efficient representation and/or implementation of basic operations e.g., data transfers
  - User and compiler cooperation e.g., high-performance vectorizer
- Dense memory representation
  - Block pointers essential to map to ukernels in a plug-n-play fashion
- Reduction loop collapsing
  - Reconstruct full K dim from tiling loop
  - Whole GEMM loop as a single BRGEMM kernel: amortize overhead of tile configs in case of AMX, avoid multiple C load & stores, enables effective SW-pipeline opportunities within the ukernel (e.g. to vnni-format weight matrix within the ukernel with minimal overhead)
- Microkernels
  - Feasible path for quick results → for all precisions supported in ukernel (see Triton CPU v1 which is substantially slower for FP32 than BF16)
  - Bridging interface mismatch – vector vs memref – is expensive
- Eliminate the power-of-2 size restrictions in Triton
  - Large power-of-2 leading dimensions cause excessive number of cache conflict misses that plummet performance (cache trashing)
  - Obviates the need for padding (happening *always* now) that is not needed algorithmically and hinders performance (unless the real GEMM dimensions *are* large powers-of-2 where padding is *optimization*)

Conclusions

Conclusions

- Even the high-level abstraction will map directly to TPP without issues on CPU and GPU
- Software and Ease of Use is most challenged with speeding up MatMul in hardware
  - How to make sure the non-linear portion is not holding us back (same for digital and optical)
  - Not covered here: scaling to trillions of parameters requires very large systems (communication/sharding) in all cases
- We should all thrive for an upstream & community owned compiler, ala “clang for AI”
- Notable papers for how to program Intel systems
  - https://arxiv.org/abs/2104.05755
  - https://arxiv.org/pdf/2304.12576
  - https://arxiv.org/abs/2404.15204v1