Tile-based Programming Models for AI

Keren Zhou
kzhou6@gmu.edu

GEORGE MASON UNIVERSITY OpenAI

Evolution of AI

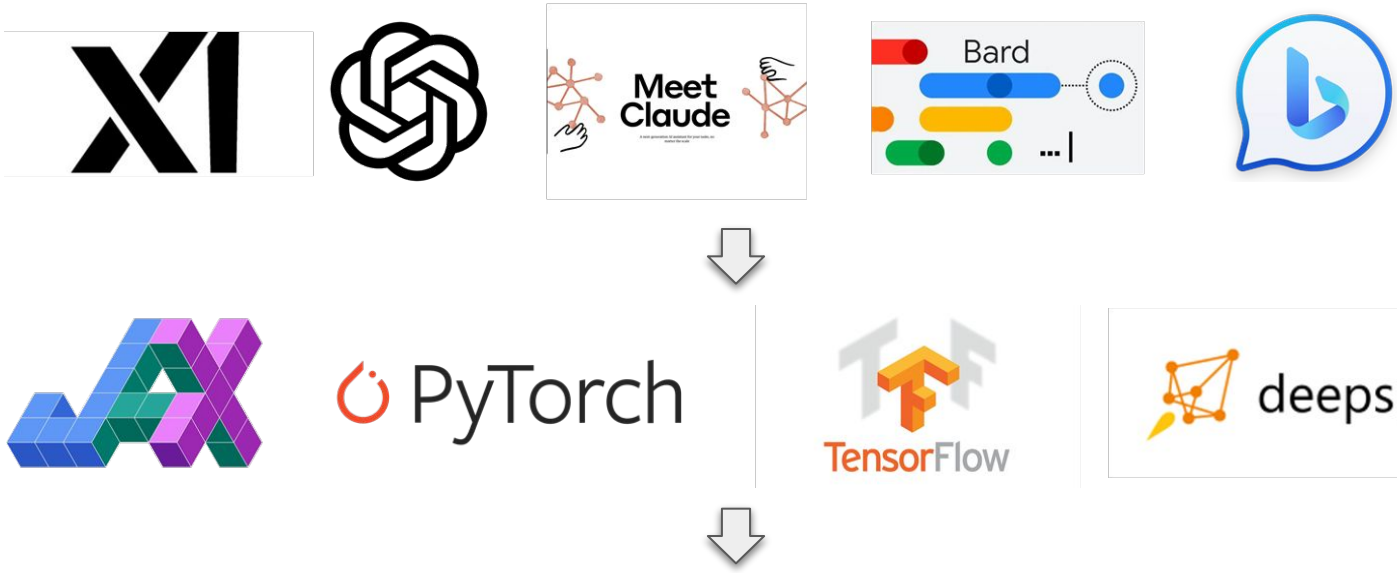
Grok

Claude

Midjourney

runway

AI Software Stack



Hot LLM Topics

- Agent
  - Tool, Planner, Memory, Environment, Multi-Agent, …
- Reasoning
  - Chain-of-Thoughts, ReAct, Reflexion, Self-consistency, …
- Reinforcement Learning
  - Tool-augmented, Instruction Tuning, Human Feedback, …
- Systems
  - KV Cache, Prefill, Decoding, Pipelining, Tensor Parallelism, Batching, …
- Kernels
  - MXFP, Quantization, MoE, Normalization, Batch GEMM, Linear/Flash/Paged/Attention, …

We're going to focus on programming languages that empowers systems and kernels

Outline

- Parallel Systems
- Tile-Based Programming
- Triton
- Triton-Puzzles

Parallel Systems

Overview

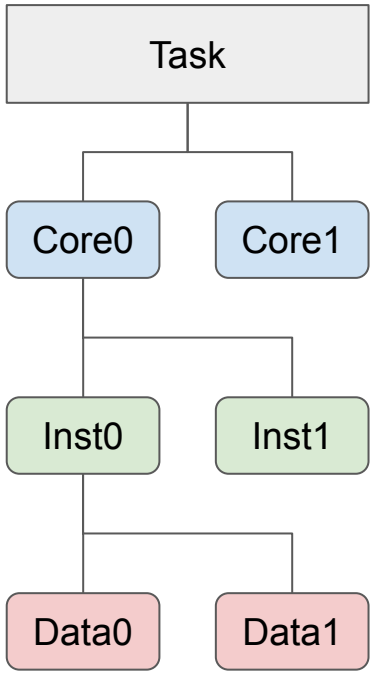
- Multi-core CPUs
- GPUs
- Accelerators

CPU Architectures

- x86 (Intel, AMD)
- ARM (Advanced RISC Machine – common in mobile and embedded)
- RISC-V (Open-source RISC architecture, rising in adoption)
- PowerPC (IBM, used in older Macs, embedded systems)
- MIPS (Used in routers, embedded systems)

Parallelisms on CPUs

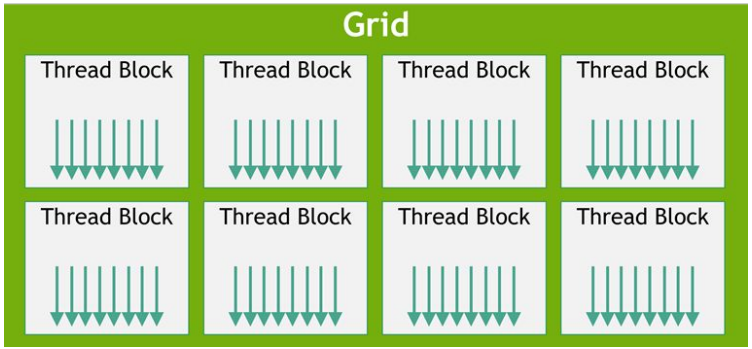
- Thread parallelism
- Instruction parallelism
- SIMD parallelism



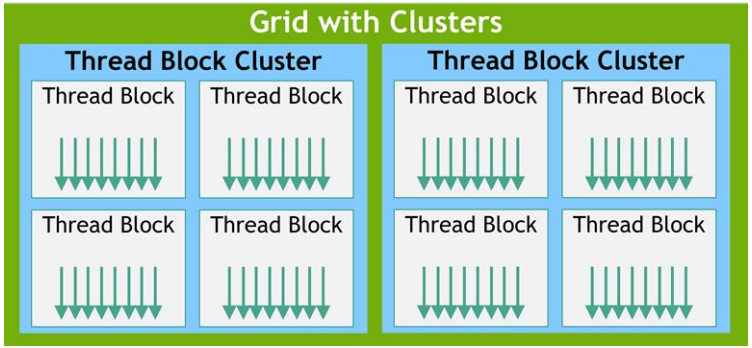
GPUs

- Grid (kernel) parallelism
- Thread block parallelism
- Thread block cluster parallelism
- Warp parallelism
- Thread parallelism
- Instruction parallelism
- SIMD parallelism

NVIDIA GPUs



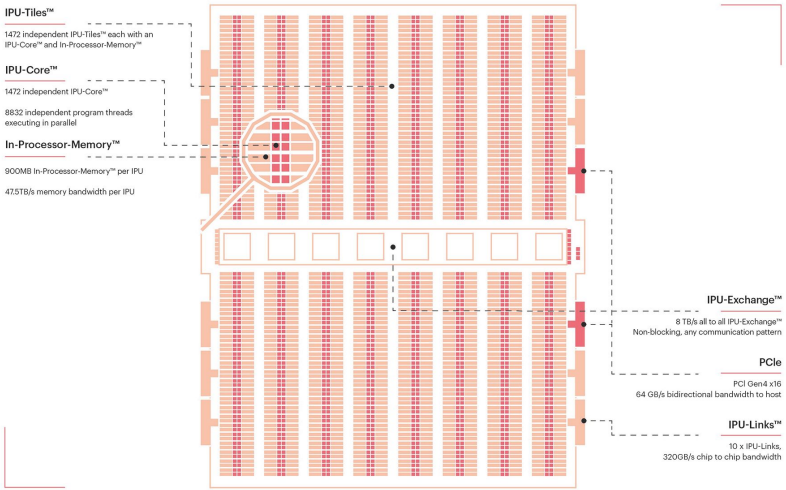
Before Hopper



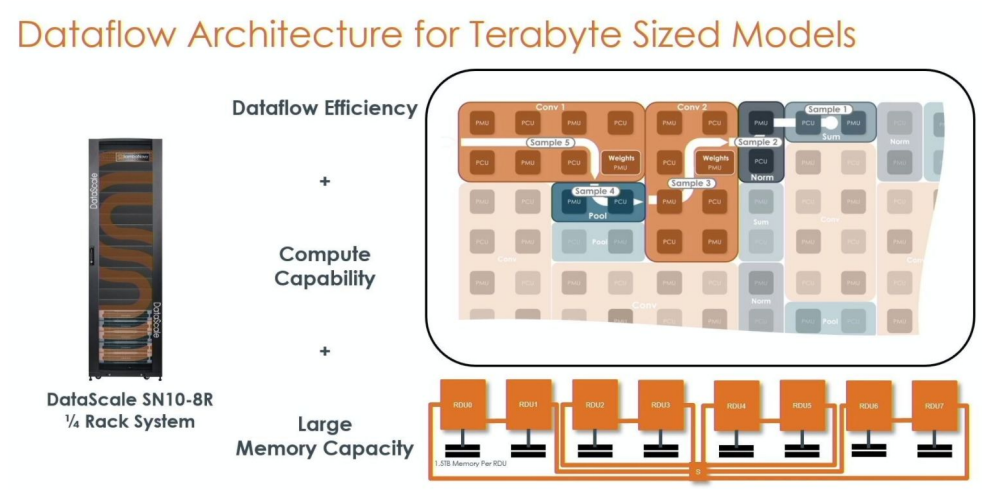
Since Hopper

Accelerators

Different accelerators have different abstractions



Graphcore



Sambanova

Tile-Based Programming

Thread-Based Programming

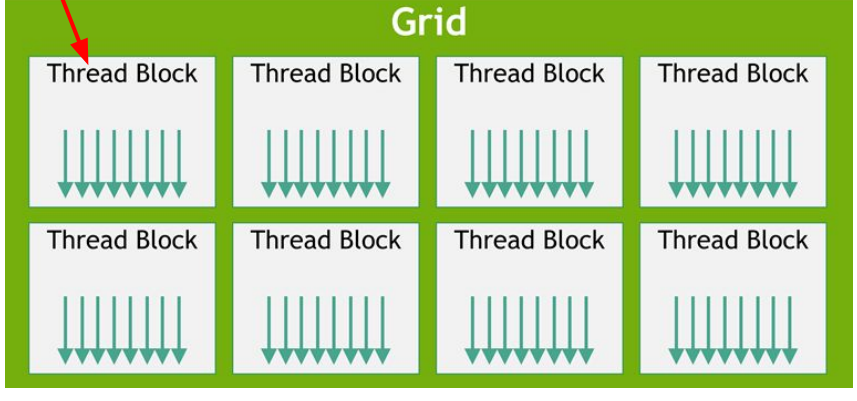
- Users specify the behavior of each thread
- The number of blocks and threads within each block is controlled on the host side

Behavior of Each Thread

vecAdd

```
__global__ void vectorAdd(const float *A, const float *B, float *C, int numElements) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < numElements) {
        C[i] = A[i] + B[i];
    }
}
```

snappify.com

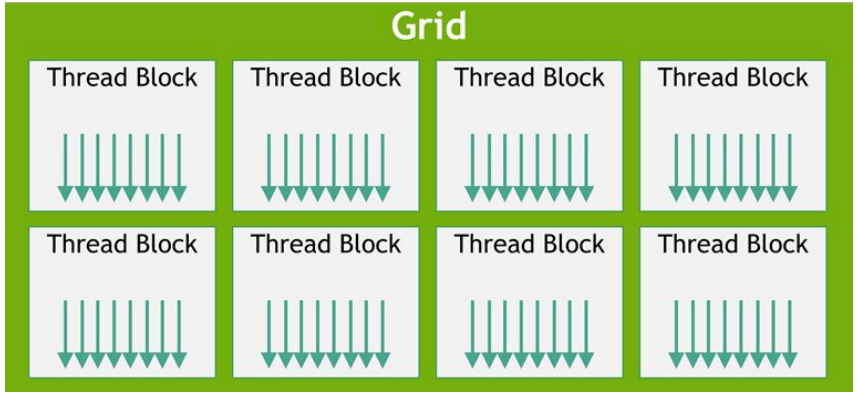


Number of Blocks and Threads

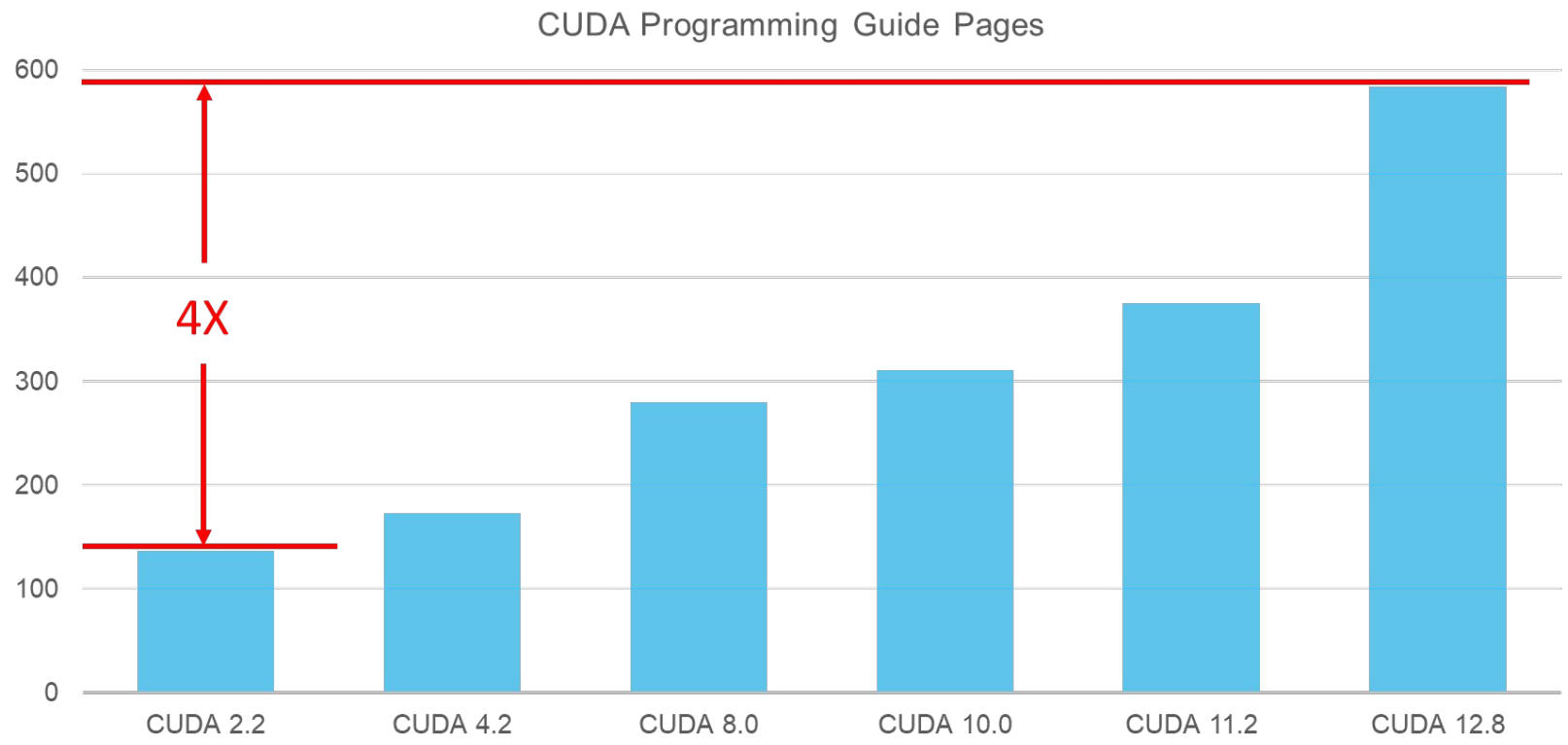
vecAdd

```
int threadsPerBlock = 256;
int blocksPerGrid =(numElements + threadsPerBlock - 1) / threadsPerBlock;
vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, numElements);
```

snappify.com



CUDA Programming Guide



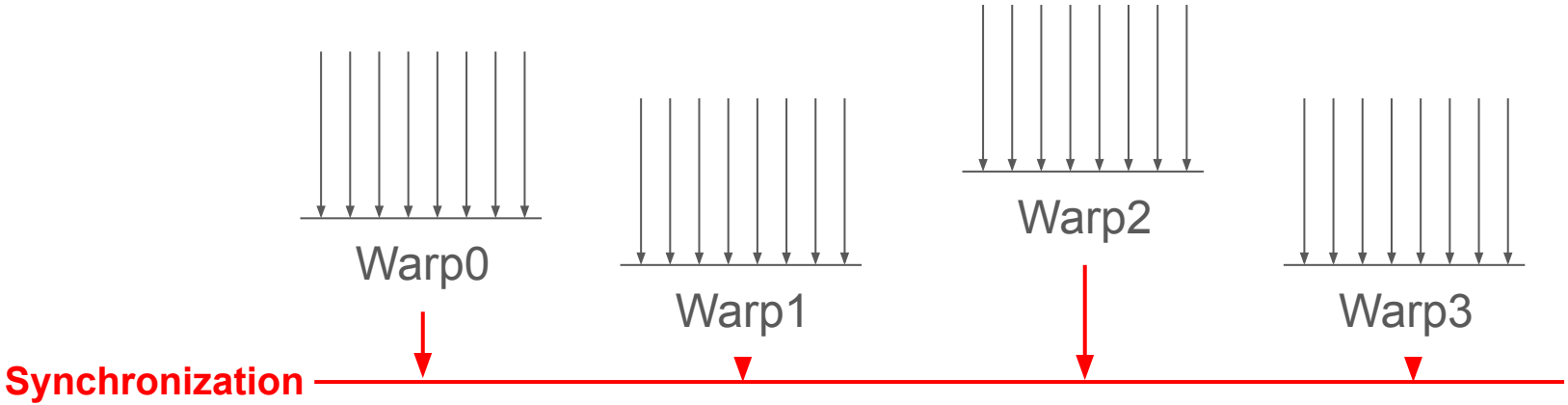
The “Problems” with Full CUDA Specification

- Unnecessary to get high performance in common scenarios
- Increasingly complex compute units/memory hierarchy
- Sophisticated user interfaces for tensor core instructions
- Hidden/unspecified behaviors

How many of you are familiar with “Programmatic Dependent Launch”?

Motivation

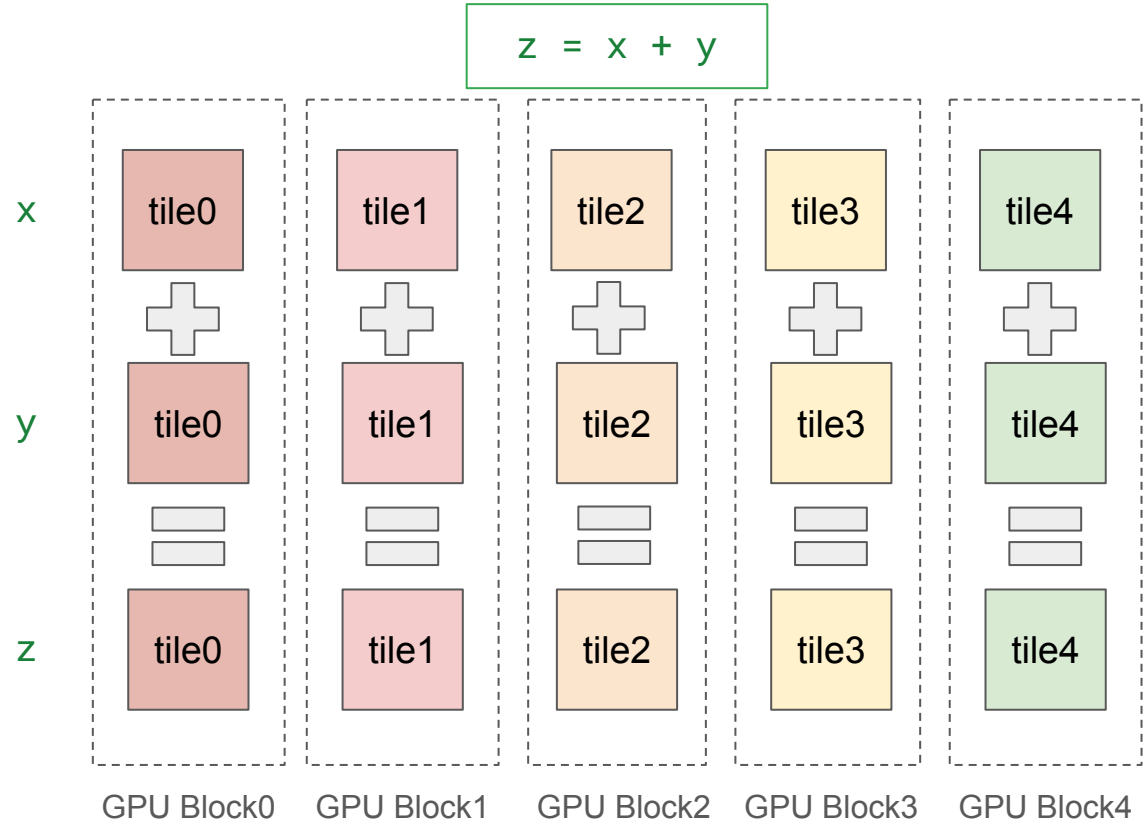
- Threads within a warp are executed in lock steps
  - This is the behavior of many simulators
  - Though not accurate with “independent thread scheduling”
- Warps within a thread block access the same programmable shared memory
- We can coarsen the execution unit as using warps or thread blocks



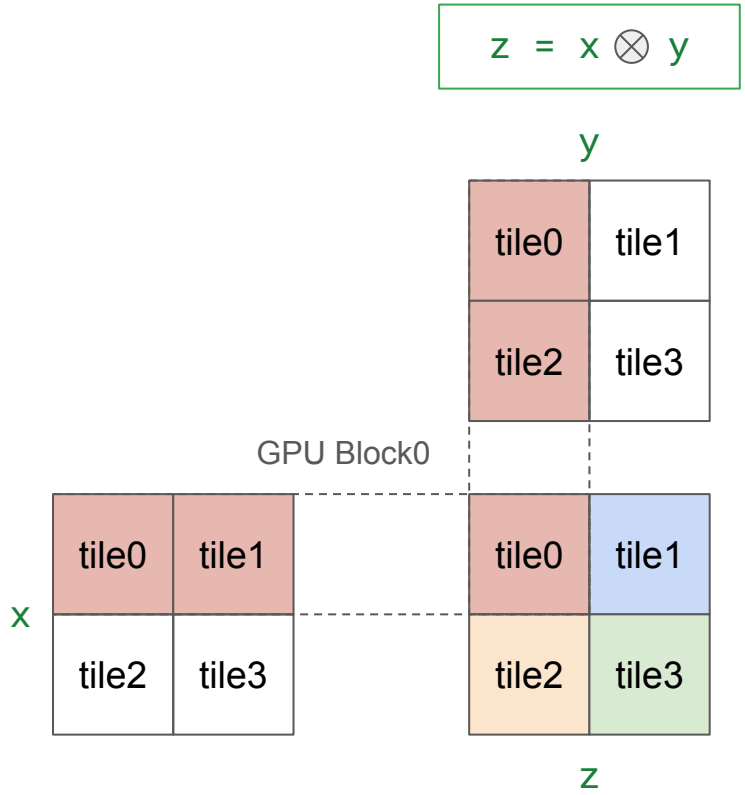
Core Concepts - Tiling/Blocking

- A tile is a block of data elements such as a submatrix or subarray processed collectively by a couple of threads
- In the context of GPU, tiles are utilized to load chunks of data into shared memory or registers
    - Coalesced global memory accesses
    - Reduced bank conflicts
    - Matching tensor core layouts
    - Promoting cache utilizing

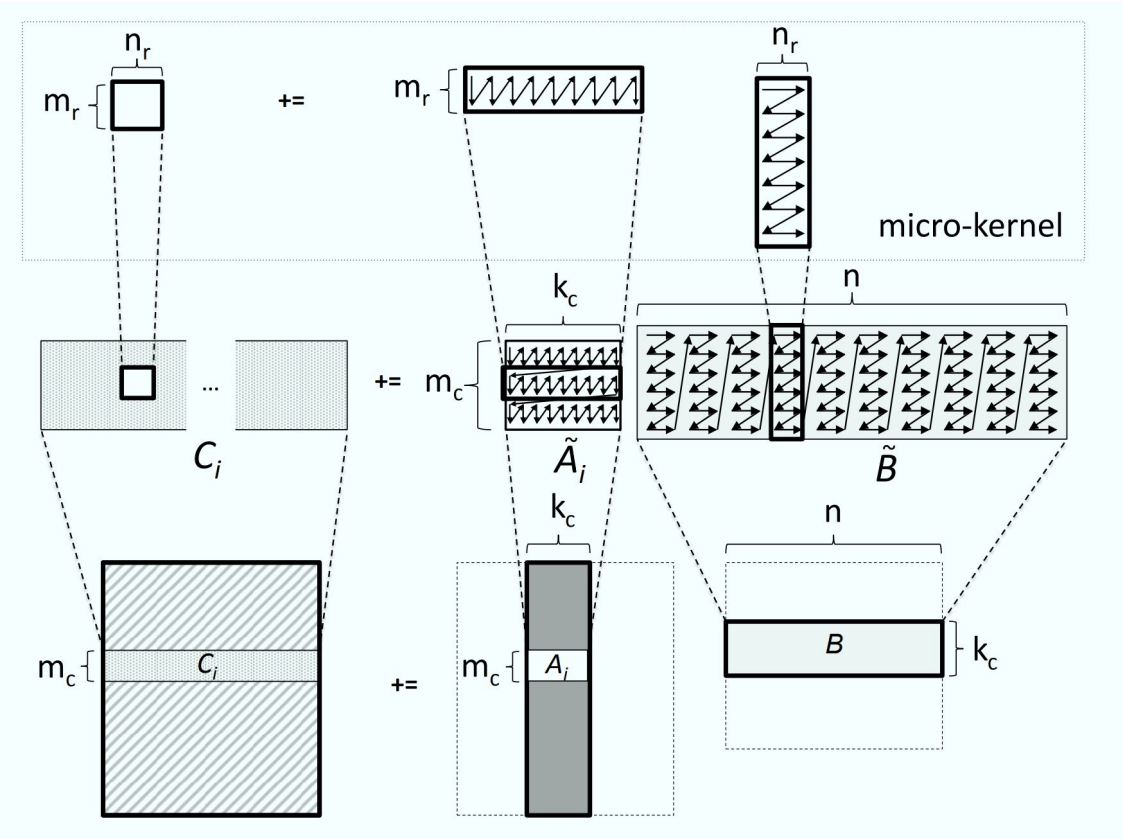
Example - Vector Addition

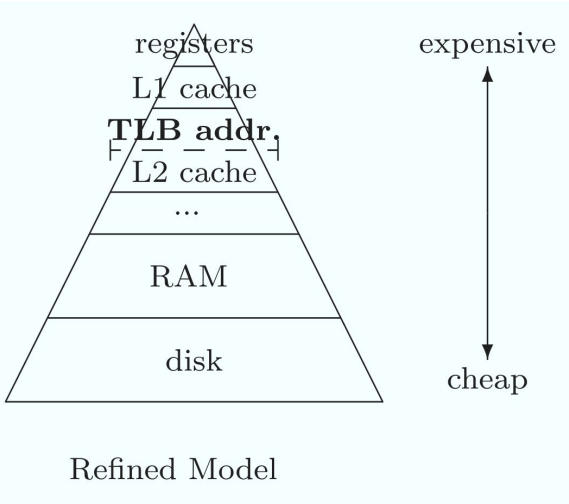


Example - Matrix Multiplication



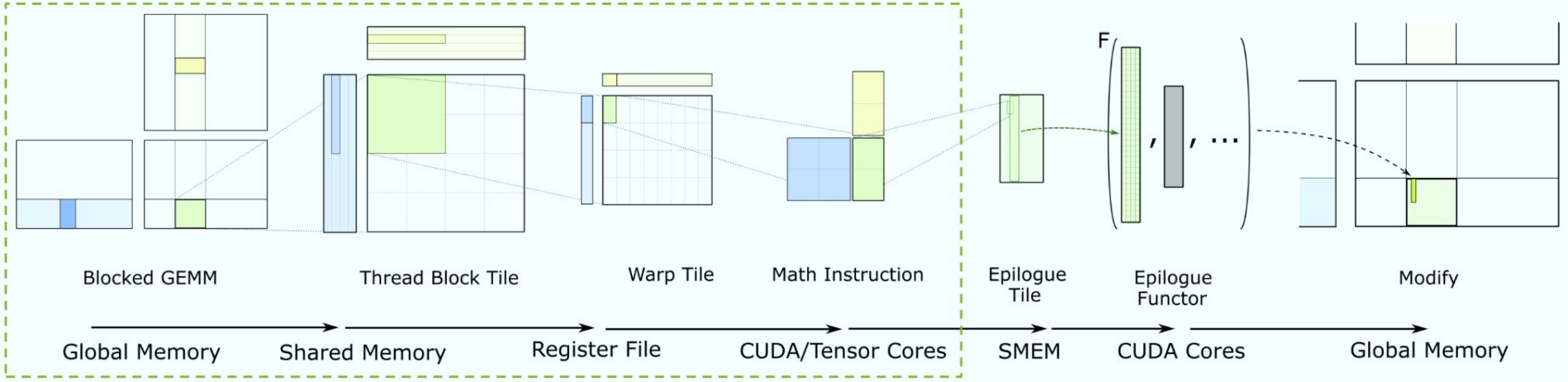
Sophisticated Tiling on CPUs





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.

Sophisticated Tiling on GPUs



Tiled, hierarchical model: reuse data in Shared Memory and in Registers

Automated Transformation - Ideas

Matmul

```
# Multiply A[M][K] * B[K][N] = C[M][N]
for i in range(M):
    for j in range(N):
        C[i][j] = 0
        for k in range(K):
            C[i][j] += A[i][k] * B[k][j]
```

Matmul

```
# Assume tile size is TILE
for i0 in range(0, M, TILE):
    for j0 in range(0, N, TILE):
        for k0 in range(0, K, TILE):
            for i in range(i0, min(i0 + TILE, M)):
                for j in range(j0, min(j0 + TILE, N)):
                    for k in range(k0, min(k0 + TILE, K)):
                        C[i][j] += A[i][k] * B[k][j]
```

Automated Transformation - Pros & Cons

✅ Pros

- Developers write simple code; the compiler or transformation tool handles the complex optimization

⚠️ Cons

- Requires sophisticated compilers (e.g., polyhedral frameworks). Not always available in all toolchains
- Customized code (e.g., fusion) may not be automatically optimized

Tile-based Programming Models

- Programmers explicitly divide the work into tiles and write tiled code using APIs or frameworks
  - Gain fine-grained control
  - Write more flexible and sophisticated kernels
  - Allows for a relatively simple autotuner

Tile-based Programming Models

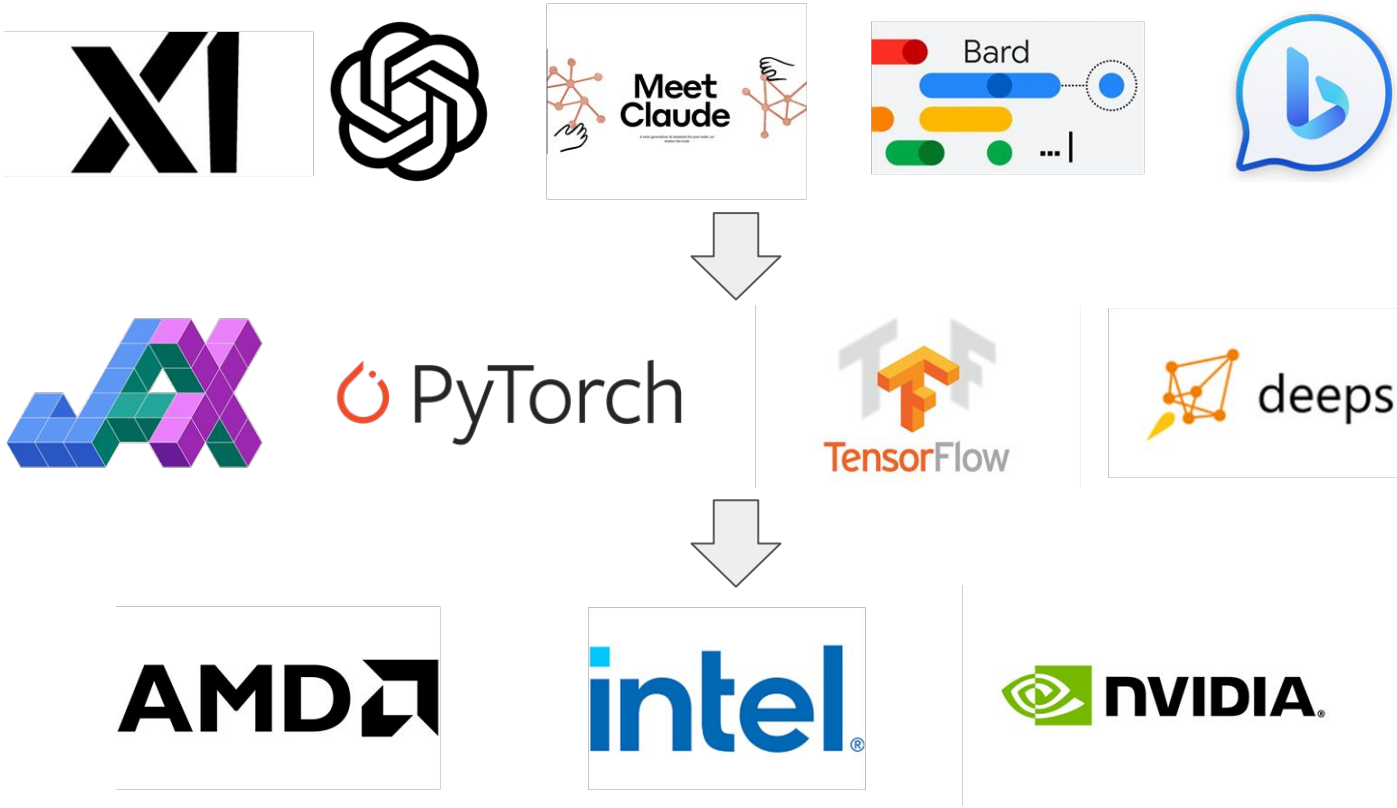
- cuTile
  - NVIDIA GPUs
- NKI
  - AWS Trainium and Inferentia
- Triton
  - AMD, NVIDIA, Intel GPUs, as well as some accelerators
- ThunderKittens, TileLang
  - Research prototypes

Triton

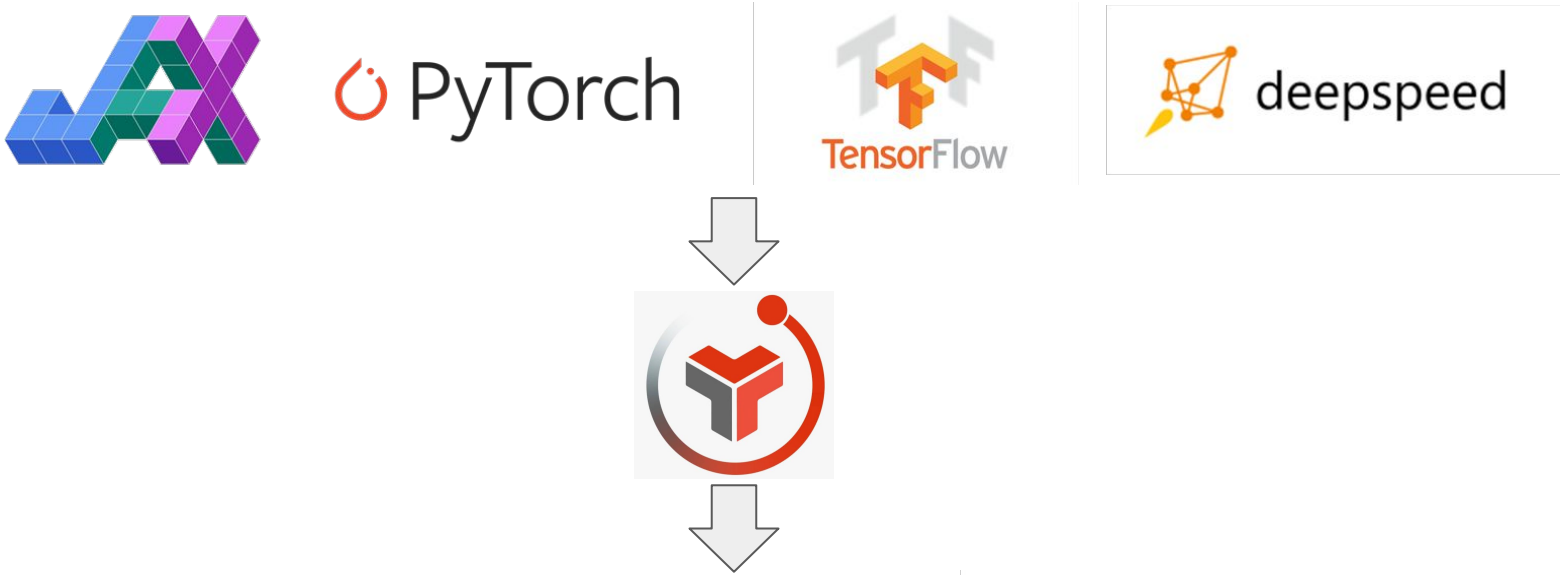
Survey

- How many of you have heard of Triton before?
- How many of you know that Triton starts from a graduate student project at Harvard?

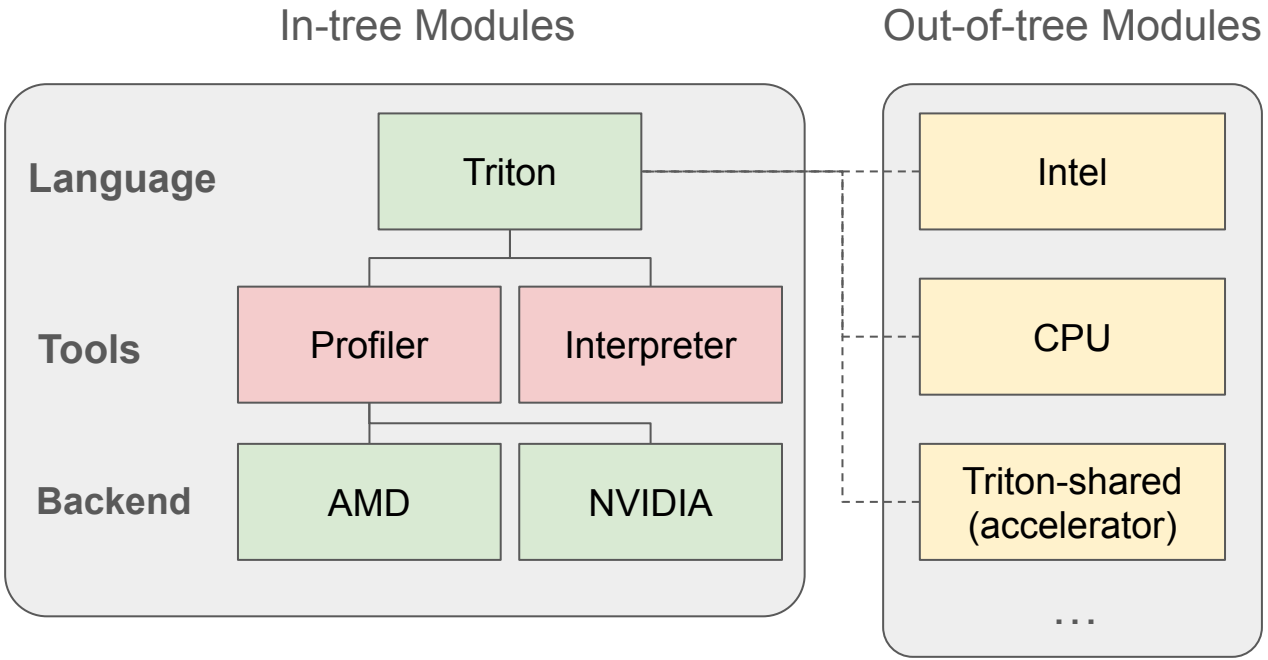
AI System Software Stack



Why Triton?



Triton Modules



Triton Language

- Python-like language designed for high flexibility and performance in deep learning applications
  - Support tensor interface similar to PyTorch
  - Uses Python-like syntax
- Compared to CUDA/ROCm, Triton simplifies GPU programming
  - Only requiring knowledge that a kernel is divided into multiple blocks (Triton programs)
  - Most underlying details are handled by the compiler

Triton vs CUDA

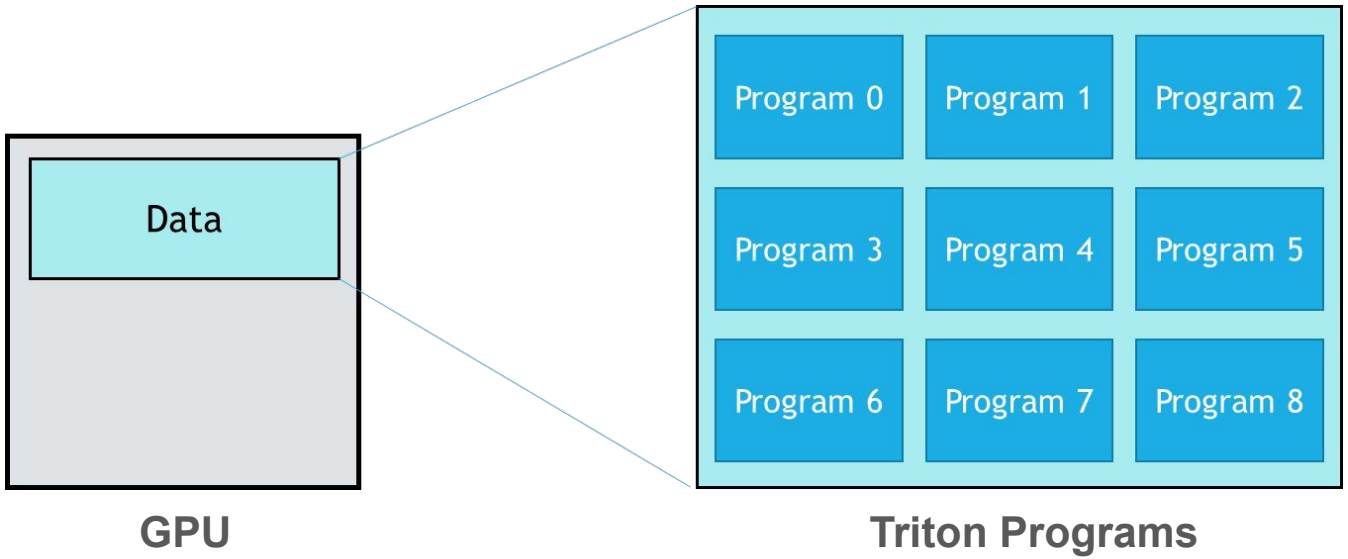
	CUDA	Triton
Memory	Global/Shared/Local	Automatic
Parallelism	Threads/Blocks/Warps	Mostly Blocks*
Tensor Core	Manual	Automatic
Vectorization	.8/.16/.32/.64/.128	Automatic
Async SIMT	Support	Limited
Device Function	Support	Support

Using Triton, you only need to know that a program is divided into multiple blocks

*In Triton 3.3, we added warp specialization so technically not all warps issue the same code

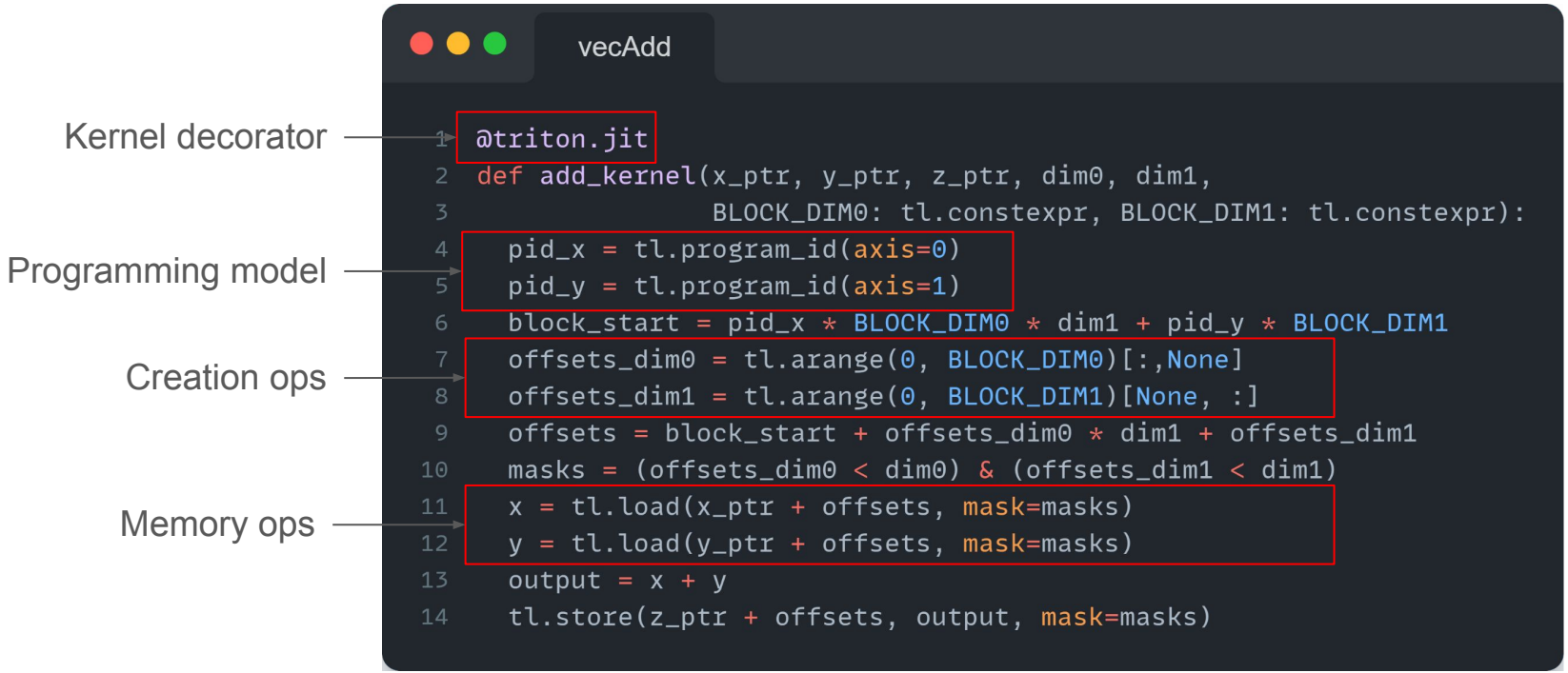
SPMD Model

- Each program executes the “same” code
  - Only program ids are different



A Simple Triton Program - Kernel Code

```
z: dim0 x dim1 = x: dim0 x dim1 + y: dim0 x dim1
```



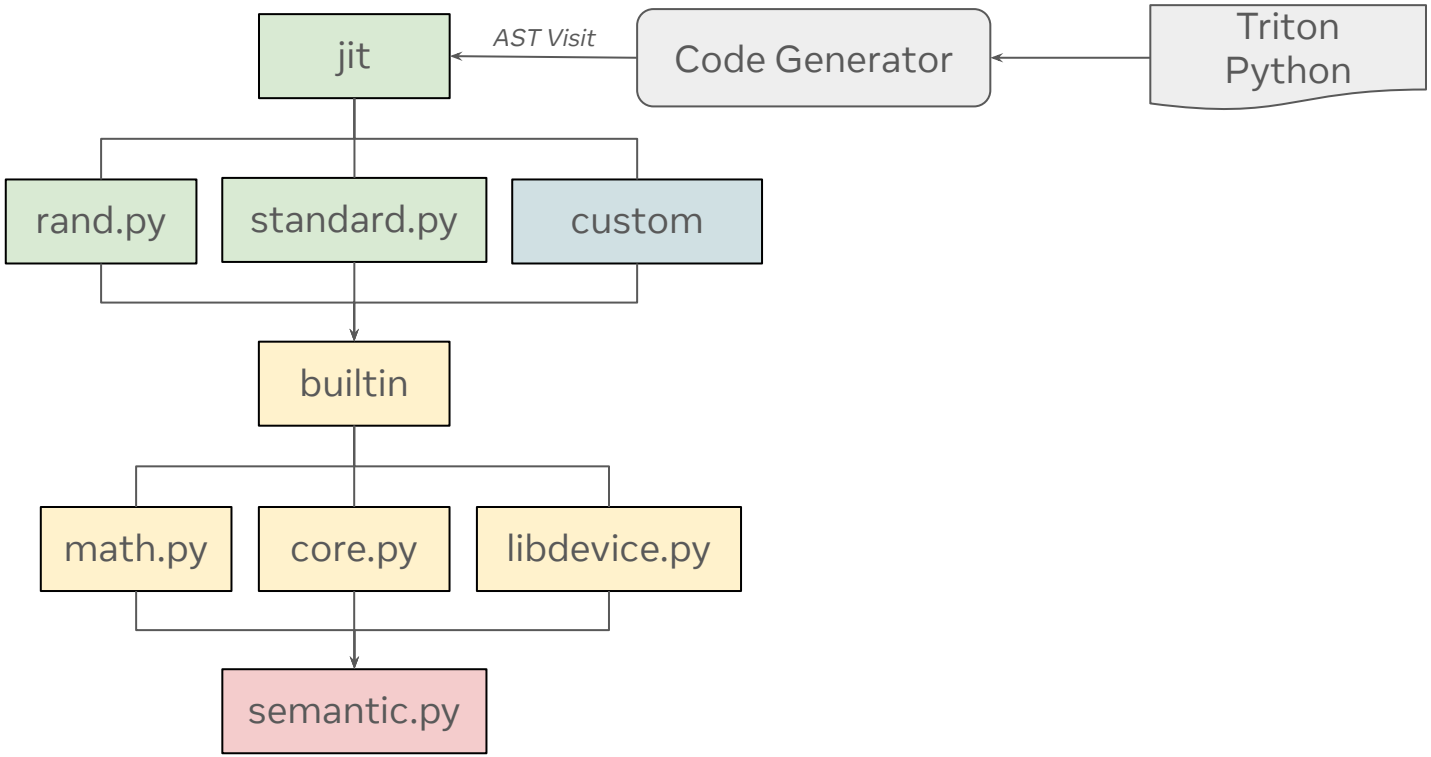
A Simple Triton Program - Kernel Launch

```
z: dim0 x dim1 = x: dim0 x dim1 + y: dim0 x dim1
```

Host Code

```
1  x = torch.randn((4096,1), device="cuda")
2  y = torch.randn((4096,1), device="cuda")
3  z = torch.empty((4096,1), device="cuda")
4  programs = 4096 // 128
5  add_kernel[(programs,)](x, y, z, 4096, 1, 128, 1)
```

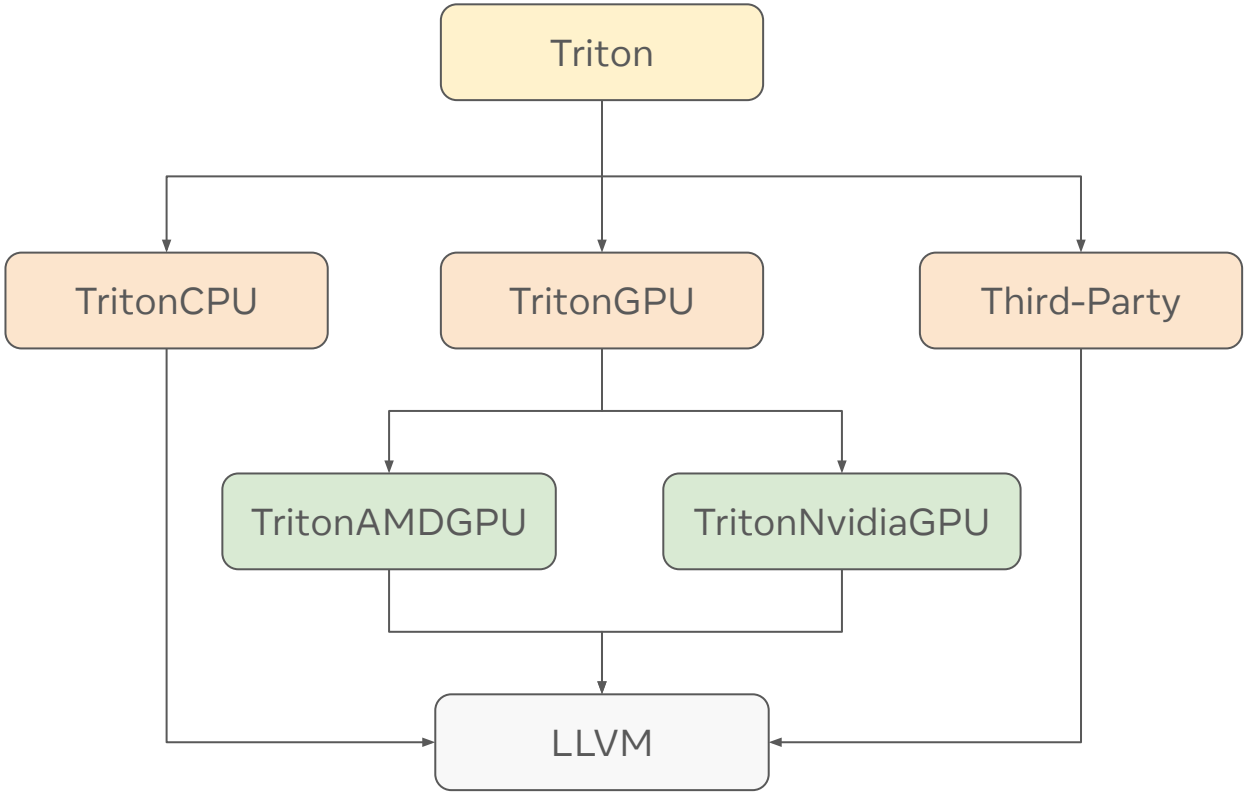
Triton Frontend



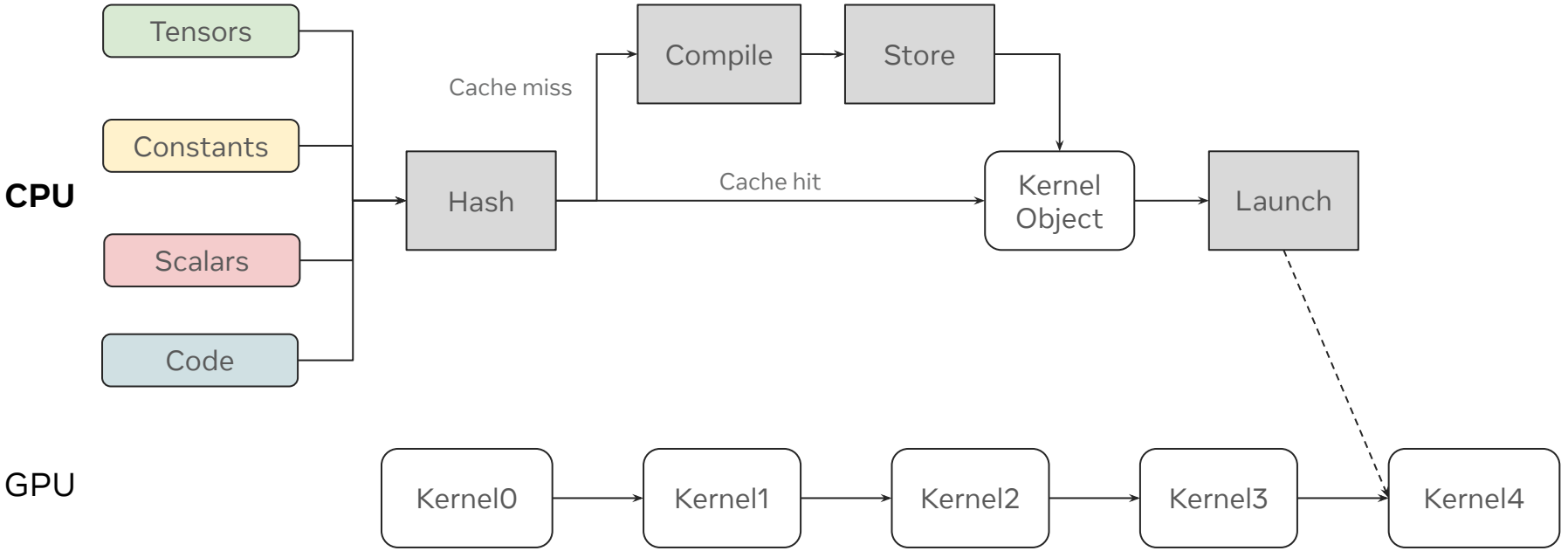
MLIR: Multi-Level IR Compiler Framework

- Building reusable and extensible compiler infrastructure
  - Address software fragmentation
  - Improve compilation for heterogeneous hardware
  - Reduce the cost of building domain specific compilers
- Key concept: Dialects
  - Each dialect is given a unique namespace that is prefixed to each defined attribute/operation/type
    - For example, the *Affine* dialect defines the namespace: affine
  - MLIR allows for multiple dialects exist in the same IR
  - A dialect can be converted to another under conversion rules

Triton Backends



Step-by-Step Compilation: JIT Compilation



Step-by-Step Compilation: TritonIR (TTIR)

```
tt.func                                                    Mangled function name
@matmul_kernel__Pfp32_Pfp32_Pfp32_i32_i32_i32_i32_i32_i32_i32_i32_i32__12c64_13c64_14c64_15c8
(%arg0: !tt.ptr<f32> {tt.divisibility = 16 : i32}, …) {
    %cst = arith.constant dense<true> : tensor<64x64xi1>
    %c64 = arith.constant 64 : i32
    %c0 = arith.constant 0 : i32
    %0 = tt.get_program_id x : i32
    %1 = arith.addi %arg3, %c63_i32 : i32
    %2 = arith.divsi %1, %c64_i32 : i32
    %3 = arith.addi %arg4, %c63_i32 : i32
    %4 = arith.divsi %3, %c64_i32 : i32
    %5 = arith.muli %4, %c8_i32 : i32
    %6 = arith.divsi %0, %5 : i32
    %7 = arith.muli %6, %c8_i32 : i32
```

Tensor

Scalar

Triton operation

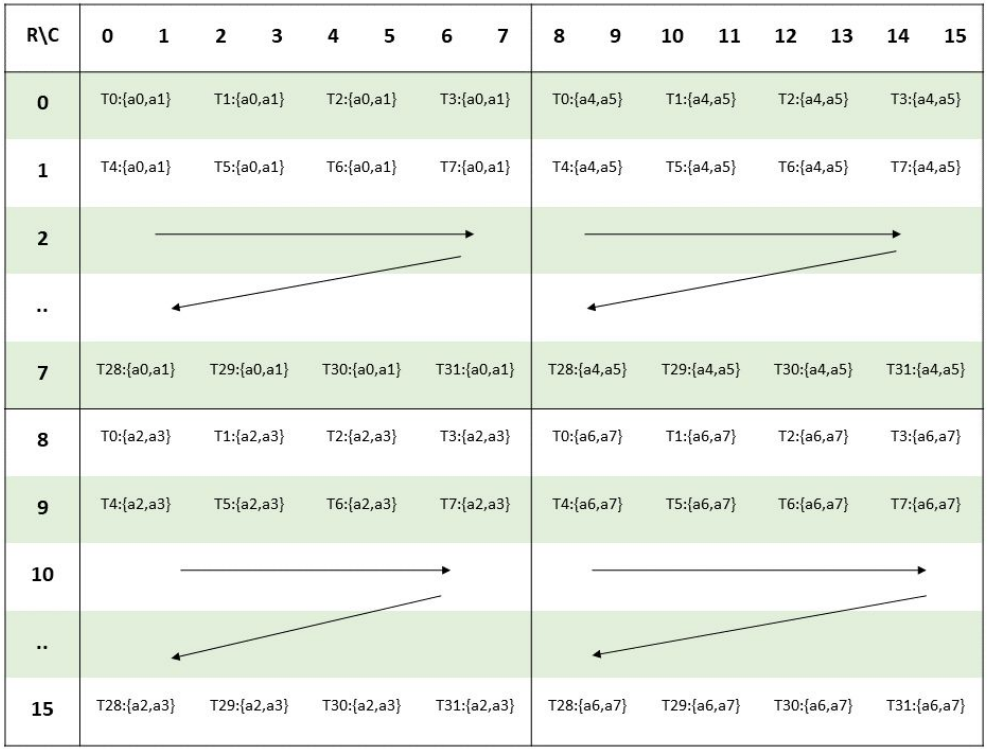
Arith operation

Step-by-Step Compilation: TritonGPU IR (TTGIR)

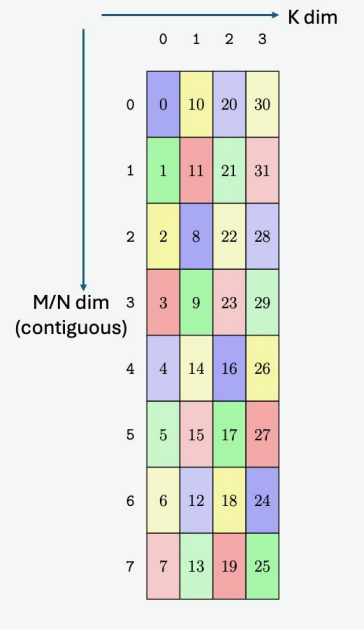
Specialized “layouts”

```
#blocked = #ttg.blocked<{sizePerThread = [8, 1], threadsPerWarp = [8, 4], warpsPerCTA = [1, 4], order = [0, 1]}>
#blocked1 = #ttg.blocked<{sizePerThread = [1, 8], threadsPerWarp = [4, 8], warpsPerCTA = [4, 1], order = [1, 0]}>
#mma = #ttg.nvidia_mma<{versionMajor = 2, versionMinor = 0, warpsPerCTA = [4, 1], instrShape = [16, 8]}>
module attributes {"ttg.num-ctas" = 1 : i32, "ttg.num-warps" = 4 : i32} {
// CHECK-LABEL: tt.func @load_two_users
  tt.func @load_two_users(%arg0: !tt.ptr<f16> {tt.divisibility = 16 : i32}, %arg1: !tt.ptr<f16> {tt.divisibility = 16 : i32})
-> (tensor<128x16xf32, #mma>, tensor<128x64xf32, #mma>) {
    %cst = arith.constant dense<0> : tensor<1x16xi32, #blocked>
    %cst_0 = arith.constant dense<0> : tensor<128x1xi32, #blocked1>
    %c0_i64 = arith.constant 0 : i64
    %c0_i32 = arith.constant 0 : i32
    %cst_1 = arith.constant dense<0.000000e+00> : tensor<128x16xf32, #mma>
    %cst_2 = arith.constant dense<0.000000e+00> : tensor<128x64xf32, #mma>
```

Example Layouts - MMA



Example Layouts - tcgen05



	Column X				Column X+1				Column X+2				Column X+3			
Lane	[0:7]	[8:15]	[16:23]	[24:31]	[0:7]	[8:15]	[16:23]	[24:31]	[0:7]	[8:15]	[16:23]	[24:31]	[0:7]	[8:15]	[16:23]	[24:31]
0	M0 SF0	M0 SF1	M0 SF0	M0 SF1	M32 SF0	M32 SF1	M32 SF0	M32 SF1	M64 SF0	M64 SF1	M64 SF0	M64 SF1	M96 SF0	M96 SF1	M96 SF0	M96 SF1
1	M1 SF0	M1 SF1	M1 SF0	M1 SF1	M33 SF0	M33 SF1	M33 SF0	M33 SF1	M65 SF0	M65 SF1	M65 SF0	M65 S1	M97 SF0	M97 SF1	M97 SF0	M97 SF1
2	M2 SF0	M2 SF1	M2 SF0	M2 SF1	M34 SF0	M34 SF1	M34 SF0	M34 SF1	M66 SF0	M66 SF1	M66 SF0	M66 SF1	M98 SF0	M98 SF1	M98 SF0	M98 SF1
3	M3 SF0	M3 SF1	M3 SF0	M3 SF1	M35 SF0	M35 SF1	M35 SF0	M35 SF1	M67 SF0	M67 SF1	M67 SF0	M67 SF1	M99 SF0	M99 SF1	M99 SF0	M99 SF1
4	M4 SF0	M4 SF1	M4 SF0	M4 SF1	M36 SF0	M36 SF1	M36 SF0	M36 SF1	M68 SF0	M68 SF1	M68 SF0	M68 SF1	M100 SF0	M100 SF1	M100 SF0	M100 SF1
5	M5 SF0	M5 SF1	M5 SF0	M5 SF1	M37 SF0	M37 SF1	M37 SF0	M37 SF1	M69 SF0	M69 SF1	M69 SF0	M69 SF1	M101 SF0	M101 SF1	M101 SF0	M101 SF1
6	M6 SF0	M6 SF1	M6 SF0	M6 SF1	M38 SF0	M38 SF1	M38 SF0	M38 SF1	M70 SF0	M70 SF1	M70 SF0	M70 SF1	M102 SF0	M102 SF1	M102 SF0	M102 SF1
7	M7 SF0	M7 SF1	M7 SF0	M7 SF1	M39 SF0	M39 SF1	M39 SF0	M39 SF1	M71 SF0	M71 SF1	M71 SF0	M71 SF1	M103 SF0	M103 SF1	M103 SF0	M103 SF1
8	M8 SF0	M8 SF1	M8 SF0	M8 SF1	M40 SF0	M40 SF1	M40 SF0	M40 SF1	M72 SF0	M72 SF1	M72 SF0	M72 SF1	M104 SF0	M104 SF1	M104 SF0	M104 SF1
9	M9 SF0	M9 SF1	M9 SF0	M9 SF1	M41 SF0	M41 SF1	M41 SF0	M41 SF1	M73 SF0	M73 SF1	M73 SF0	M73 SF1	M105 SF0	M105 SF1	M105 SF0	M105 SF1
10	M10 SF0	M10 SF1	M10 SF0	M10 SF1	M42 SF0	M42 SF1	M42 SF0	M42 SF	M74 SF0	M74 SF1	M74 SF0	M74 SF1	M106 SF0	M106 SF1	M106 SF0	M106 SF1
11	M11 SF0	M11 SF1	M11 SF0	M11 SF1	M43 SF0	M43 SF1	M43 SF0	M43 SF1	M75 SF0	M75 SF1	M75 SF0	M75 SF1	M107 SF0	M107 SF1	M107 SF0	M107 SF1
12	M12 SF0	M12 SF1	M12 SF0	M12 SF1	M44 SF0	M44 SF1	M44 SF0	M44 SF1	M76 SF0	M76 SF1	M76 SF0	M76 SF1	M108 SF0	M108 SF1	M108 SF0	M108 SF1
13	M13 SF0	M13 SF1	M13 SF0	M13 SF1	M45 SF0	M45 SF1	M45 SF0	M45 SF1	M77 SF0	M77 SF1	M77 SF0	M77 SF1	M109 SF0	M109 SF1	M109 SF0	M109 SF1
14	M14 SF0	M14 SF1	M14 SF0	M14 SF1	M46 SF0	M46 SF1	M46 SF0	M46 SF1	M78 SF0	M78 SF1	M78 SF0	M78 SF1	M110 SF0	M110 SF1	M110 SF0	M110 SF1
15	M15 SF0	M15 SF1	M15 SF0	M15 SF1	M47 SF0	M47 SF1	M47 SF0	M47 SF1	M79 SF0	M79 SF1	M79 SF0	M79 SF1	M111 SF0	M111 SF1	M111 SF0	M111 SF1
16	M16 SF0	M16 SF1	M16 SF0	M16 SF1	M48 SF0	M48 SF1	M48 SF0	M48 SF1	M80 SF0	M80 SF1	M80 SF0	M80 SF1	M112 SF0	M112 SF1	M112 SF0	M112 SF1
17	M17 SF0	M17 SF1	M17 SF0	M17 SF1	M49 SF0	M49 SF1	M49 SF0	M49 SF1	M81 SF0	M81 SF1	M81 SF0	M81 SF1	M113 SF0	M113 SF1	M113 SF0	M113 SF1
18	M18 SF0	M18 SF1	M18 SF0	M18 SF1	M50 SF0	M50 SF1	M50 SF0	M50 SF1	M82 SF0	M82 SF1	M82 SF0	M82 SF1	M114 SF0	M114 SF1	M114 SF0	M114 SF1
19	M19 SF0	M19 SF1	M19 SF0	M19 SF1	M51 SF0	M51 SF1	M51 SF0	M51 SF1	M83 SF0	M83 SF1	M83 SF0	M83 SF1	M115 SF0	M115 SF1	M115 SF0	M115 SF1
20	M20 SF0	M20 SF1	M20 SF0	M20 SF1	M52 SF0	M52 SF1	M52 SF0	M52 SF1	M84 SF0	M84 SF1	M84 SF0	M84 SF1	M116 SF0	M116 SF1	M116 SF0	M116 SF1
21	M21 SF0	M21 SF1	M21 SF0	M21 SF1	M53 SF0	M53 SF1	M53 SF0	M53 SF1	M85 SF0	M85 SF1	M85 SF0	M85 SF1	M117 SF0	M117 SF1	M117 SF0	M117 SF1
22	M22 SF0	M22 SF1	M22 SF0	M22 SF1	M54 SF0	M54 SF1	M54 SF0	M54 SF1	M86 SF0	M86 SF1	M86 SF0	M86 SF1	M118 SF0	M118 SF1	M118 SF0	M118 SF1
23	M23 SF0	M23 SF1	M23 SF0	M23 SF1	M55 SF0	M55 SF1	M55 SF0	M55 SF1	M87 SF0	M87 SF1	M87 SF0	M87 SF1	M119 SF0	M119 SF1	M119 SF0	M119 SF1
24	M24 SF0	M24 SF1	M24 SF0	M24 SF1	M56 SF0	M56 SF1	M56 SF0	M56 SF1	M88 SF0	M88 SF1	M88 SF0	M88 SF1	M120 SF0	M120 SF1	M120 SF0	M120 SF1
25	M25 SF0	M25 SF1	M25 SF0	M25 SF1	M57 SF0	M57 SF1	M57 SF0	M57 SF1	M89 SF0	M89 SF1	M89 SF0	M89 SF1	M121 SF0	M121 SF1	M121 SF0	M121 SF1
26	M26 SF0	M26 SF1	M26 SF0	M26 SF1	M58 SF0	M58 SF1	M58 SF0	M58 SF1	M90 SF0	M90 SF1	M90 SF0	M90 SF1	M122 SF0	M122 SF1	M122 SF0	M122 SF1
27	M27 SF0	M27 SF1	M27 SF0	M27 SF1	M59 SF0	M59 SF1	M59 SF0	M59 SF1	M81 SF0	M81 SF1	M81 SF0	M81 SF1	M123 SF0	M123 SF1	M123 SF0	M123 SF1
28	M28 SF0	M28 SF1	M28 SF0	M28 SF1	M60 SF0	M60 SF1	M60 SF0	M60 SF1	M82 SF0	M82 SF1	M82 SF0	M82 SF1	M124 SF0	M124 SF1	M124 SF0	M124 SF1
29	M29 SF0	M29 SF1	M29 SF0	M29 SF1	M61 SF0	M61 SF1	M61 SF0	M61 SF1	M93 SF0	M93 SF1	M93 SF0	M93 SF1	M125 SF0	M125 SF1	M125 SF0	M125 SF1
30	M30 SF0	M30 SF1	M30 SF0	M30 SF1	M62 SF0	M62 SF1	M62 SF0	M62 SF1	M94 SF0	M94 SF1	M94 SF0	M94 SF1	M126 SF0	M126 SF1	M126 SF0	M126 SF1
31	M31 SF0	M31 SF1	M31 SF0	M31 SF1	M63 SF0	M63 SF1	M63 SF0	M63 SF1	M95 SF0	M95 SF1	M95 SF0	M95 SF1	M127 SF0	M127 SF1	M127 SF0	M127 SF1
	0	1	2	3												

SFA ID = 00 | SFA ID = 10

Key Transformation Passes

- Remove layout conversion
  - Rewrite the `ConvertLayoutOps` to reduce the number of operations and to prefer favorable layouts like `BlockedEncodingAttr` layout for "expensive" loads and stores (good for coalescing) and `NvidiaMmaEncodingAttr` otherwise (good for tensor ops)
- Accelerate matmul
  - Optimize the input/output layout of `dot` instructions to make them compatible hardware accelerators
- Automatic warp specialization
  - Analyze the loops in the kernel and attempt to create a partition schedule so that different warps handles different code regions
- Pipeline
  - Apply software pipelining to loops in the module based on number of stages. This may convert some load into asynchronous loads, and multi-buffer the data.

More info can be found in:
triton/Dialect/TritonGPU/Transforms/Passes.td

State-of-the-art GEMM Performance

Numbers represents fp8 TFLOPS on Blackwell GB200

```
1690.079 3659.446 ROOT
├─ 1598.967 429.774 cublas [M=8192, N=8192, K=512]
 |   └─ nan 429.774 nvjet_qqhsq_256x256_128x4_2x1_2cta_v_bz_TNT
├─ 1491.274 460.810 matmul_kernel [M=8192, N=8192, K=512]
├─ 1778.448 386.401 matmul_kernel_persistent [M=8192, N=8192, K=512]
├─ 1887.893 364.001 matmul_kernel_descriptor_persistent [M=8192, N=8192, K=512]
├─ 2178.427 315.455 matmul_kernel_descriptor_persistent_ws [M=8192, N=8192, K=512]
├─ 1972.040 348.469 matmul_kernel_tma_persistent [M=8192, N=8192, K=512]
├─ 2359.637 291.229 matmul_kernel_tma_persistent_ws [M=8192, N=8192, K=512]
```

FLOPS Time

Recent Advances in Tile-Based Programming Models

Paper	Conf	Target Problems
Rammer: Enabling Holistic Deep Learning Compiler Optimizations with rTasks	OSDI'20	Intra- and inter operator scheduling
Roller: Fast and Efficient Tensor Compilation for Deep Learning	OSDI'22	Tile-code generation with performance modeling
Welder: Scheduling Deep Learning Memory Access via Tile-graph	OSDI'23	Tile scheduling
Cocktailer: Analyzing and Optimizing Dynamic Control Flow in Deep Learning	OSDI'23	uTask scheduling
TensorIR: An Abstraction for Automatic Tensorized Program Optimization	ASPLOS'23	Block abstraction
Hidet: Task-mapping programming paradigm for deep learning tensor programs	ASPLOS'23	Tile with layouts
Graphene: An ir for optimized tensor computations on gpus	ASPLOS'23	Tile with layouts
Task-Based Tensor Computations on Modern GPUs	PLDI'25	Tile with warp specialization

Triton-Puzzles

Triton Puzzles

- A set of questions for you to learn Triton from scratch
- You will start with trivial examples and build your way up to real algorithms like Flash Attention and Quantized neural networks
- These puzzles do not need to run on the GPU since they use the Triton interpreter

Visualization

- One area that learners have trouble with is memory loading and storage which is critical for speed on low-level devices
- Triton-Viz is a visualizer that illustrates load/store and other triton operations using a user-friendly GUI
- It also provides simple a statistic summary about operations done by the triton kernel

Get Started

- https://github.com/srush/Triton-Puzzles

Learning Resources

- rkinas/triton-resources: A curated list of resources for learning and exploring Triton, OpenAI's programming language for writing efficient GPU code.
- gpu-mode/lectures: Material for gpu-mode lectures
- https://discord.gg/gpumode