Triton

Philippe Tillet, H. T. Kung, David Cox
(MAPL '19)

Presented by: Krut Patel (CS 598 AIE FA24)

Contents

Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations

Philippe Tillet, H. T. Kung, David Cox (MAPL ‘19)

Background - GPU architecture

- HBM is slow – need to load once to SRAM, do lots of compute on it (FlashAttention)
- Lots of cores (SM) – need to partition and schedule work
    - V100 has new tensor cores, 2-3 orders of magnitude more FLOPs for Matmul

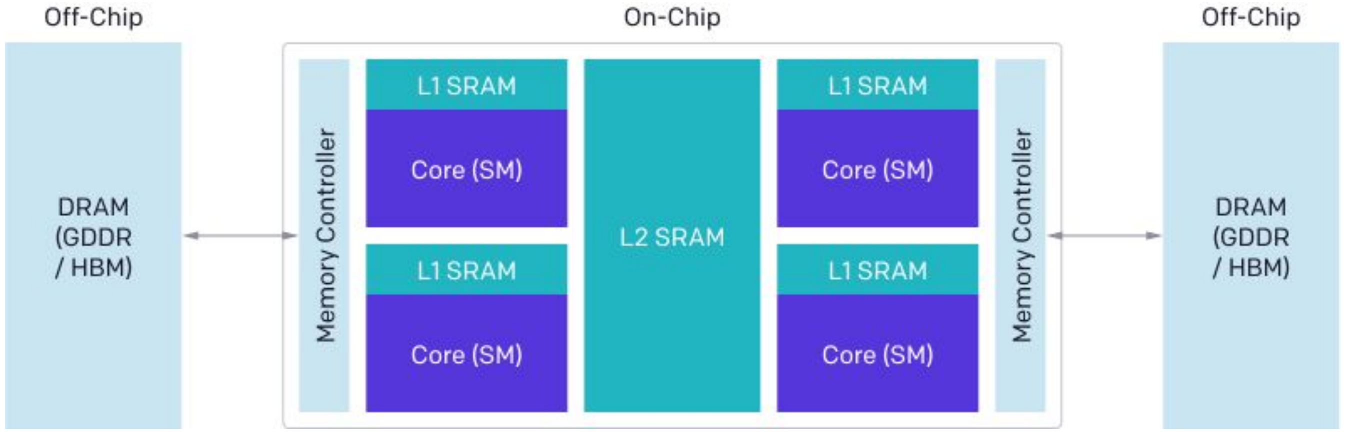
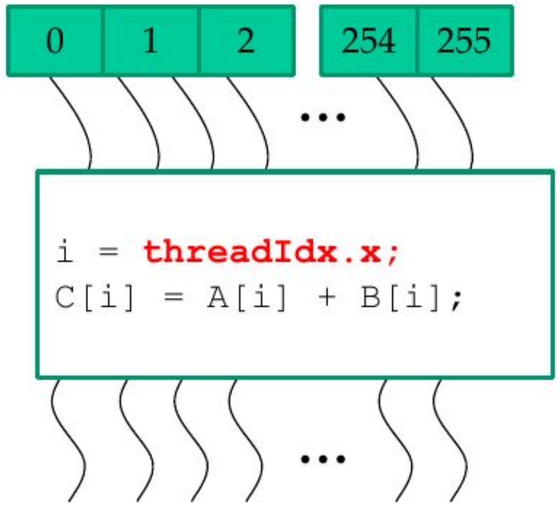


Image Source: https://openai.com/index/triton/

Background – CUDA Programming Model

- Highly parallel hardware
- Each SM runs one "Block" of threads at a time
  - In this example, there are 256 threads in the block
- There are multiple blocks in the grid
- Each thread uses `threadIdx` variable to find out its own position in the grid
- Threads can use shared memory as fast scratchpads (local to a block)



```
i = threadIdx.x;
C[i] = A[i] + B[i];
```

Source: CS 483 slides

Motivation

- Provide a better abstraction than CUDA for programming with GPUs
- But still have some control over execution schedule

	CUDA	**TRITON**	**TVM**
Memory Coalescing	Manual	Automatic	Automatic
Shared Memory Management	Manual	Automatic	Automatic
Scheduling (Within SMs)	Manual	Automatic	Automatic
Scheduling (Across SMs)	Manual	Manual	Automatic
Parallelism	Threads/Blocks/Warps	Mostly Blocks	Automatic

Triton Programming Model

- `tile` is a first-class language construct
- Single Instruction Multiple Thread vs Single Program Multiple Data

```
BLOCK = 512

@jit
def add(X, Y, Z, N):
  tid = threadIdx.x
  bid = blockIdx.x
  # scalar index
  idx = bid * BLOCK + tid
  if id < N:
    # There is no pointer in Numba.
    # Z,X,Y are dense tensors
    Z[idx] = X[idx] + Y[idx]

...
grid = (ceil_div(N, BLOCK),)
block = (BLOCK,)
add[grid, block](x, y, z, x.shape[0])
```

Numba code

```
BLOCK = 512
@jit
def add(X, Y, Z, N):
  pid = program_id(0)
  # block of indices
  idx = pid * BLOCK + arange(BLOCK)
  mask = idx < N
  # Triton uses pointer arithmetics
  # rather than indexing operators
  x = load(X + idx, mask=mask)
  y = load(Y + idx, mask=mask)
  store(Z + idx, x + y, mask=mask)

...
grid = (ceil_div(N, BLOCK),)
# no thread-block
add[grid](x, y, z, x.shape[0])
```

Triton code

Source: https://openai.com/index/triton/

SIMT vs SPMD in more detail

SIMT implies all threads are executing in lock-step.

- This happens in a GPU warp (32 threads)
- Even though they are different threads, they execute same instructions

SPMD just says overall program can be executed on different data

- CUDA Blocks are technically SPMD, since blocks have very limited communication and synchronization options
- Triton kernels are also defined at the CUDA block level, hence SPMD.

Triton-C

- Will skip, mostly irrelevant nowadays.
- Main part was the programming model, which we have discussed previously

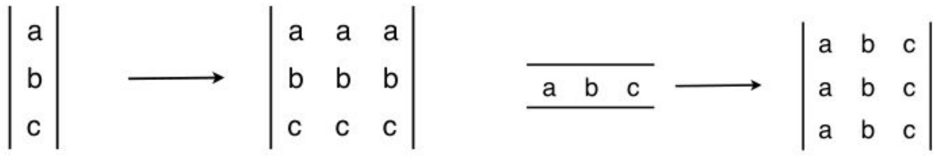
Triton IR

```
define kernel void @relu(float* %A, i32 %M, i32 %N) {
prologue:
 %rm = call i32 <8> get global range (0);
 %rn = call i32 <8> get_global_range (1);
 ; broadcast shapes
 %1 = reshape i32 <8, 8> %M;
 %M0 = broadcast i32 <8, 8> %1;
 ; ... broadcast global ranges
 %3 = reshape i32 <8, 1> %rm;
 %rm bc = broadcast i32 <8, 8> %3;
 ; ... compute mask
 %pm = icmp slt %rm bc , %M0;
 %pn = icmp slt %rn bc , %N0;
 %msk = and %pm , %pn;
 ; compute pointer
 %A0 = splat float*<8, 8> %A;
 %5 = getelementptr %A0 , %rm_bc;
 %6 = mul %rn bc , %M0;
 %pa = getelementptr %5, %6;
 ; compute result
 %a = load %pa;
 % 0 = splat float <8, 8> 0;
 %result = max %float %a, %_0;
 ; write back
 store fp32 <8, 8> %pa , %result}
```

- Modification of LLVM IR, with `tile` data types

  `i32<8,8>, float*<4>`

- Broadcasting and reshape support on tiles

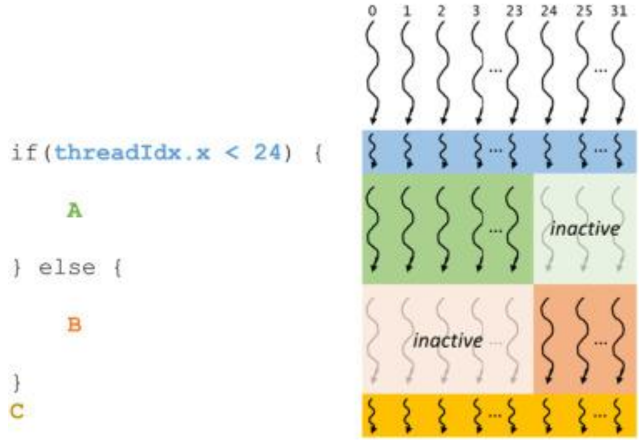


(a) [3 × 1] input **(b)** [1 × 3] input

Figure 4. The `broadcast <3,3>` instruction

Predication

- Recall, GPU follows a SIMT model. What happens during control divergence?
  - Aka, when some threads take if branch and others take else branch?
- GPU will run *both* the branches on *all* the threads. SIMT.
- But the inactive threads will be “masked-off”
- Only occurs at warp-level (32 threads)
  - If entire warp takes one branch, no divergence
- Triton uses predication for its own IR
- See `mask` variable in the `add` kernel
- Some loss of fidelity, as mask might be unnecessary for a warp

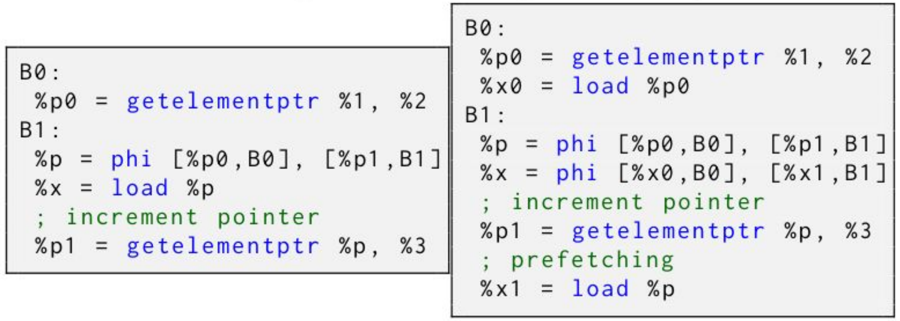


Source: https://www.sciencedirect.com/topics/computer-science/thread-divergence

Optimizations: Machine-Independent

- Prefetching:
  - Problem: Access to memory is very slow. Bad if done inside a loop
  - Solution: Detect loops, load next tile in current iteration
- Peephole optimizations:
  - Usually, look at some sequence of instructions, replace with a better version
  - Triton adds tile-specific algebraic identities

Listing 7. Automatic pre-fetching



```
B0:
 %p0 = getelementptr %1, %2
B1:
 %p = phi [%p0,B0], [%p1,B1]
 %x = load %p
 ; increment pointer
 %p1 = getelementptr %p, %3
```

```
B0:
 %p0 = getelementptr %1, %2
 %x0 = load %p0
B1:
 %p = phi [%p0,B0], [%p1,B1]
 %x = phi [%x0,B0], [%x1,B1]
 ; increment pointer
 %p1 = getelementptr %p, %3
 ; prefetching
 %x1 = load %p
```

Optimizations: Hierarchical Tiling

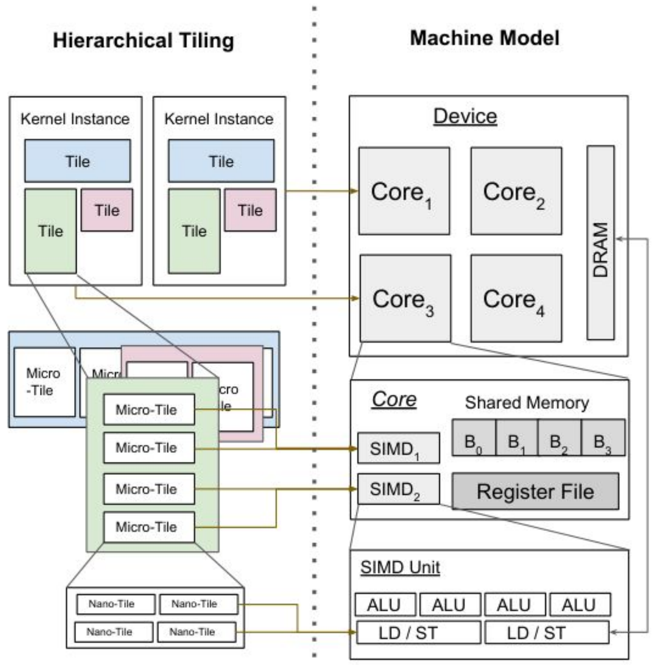
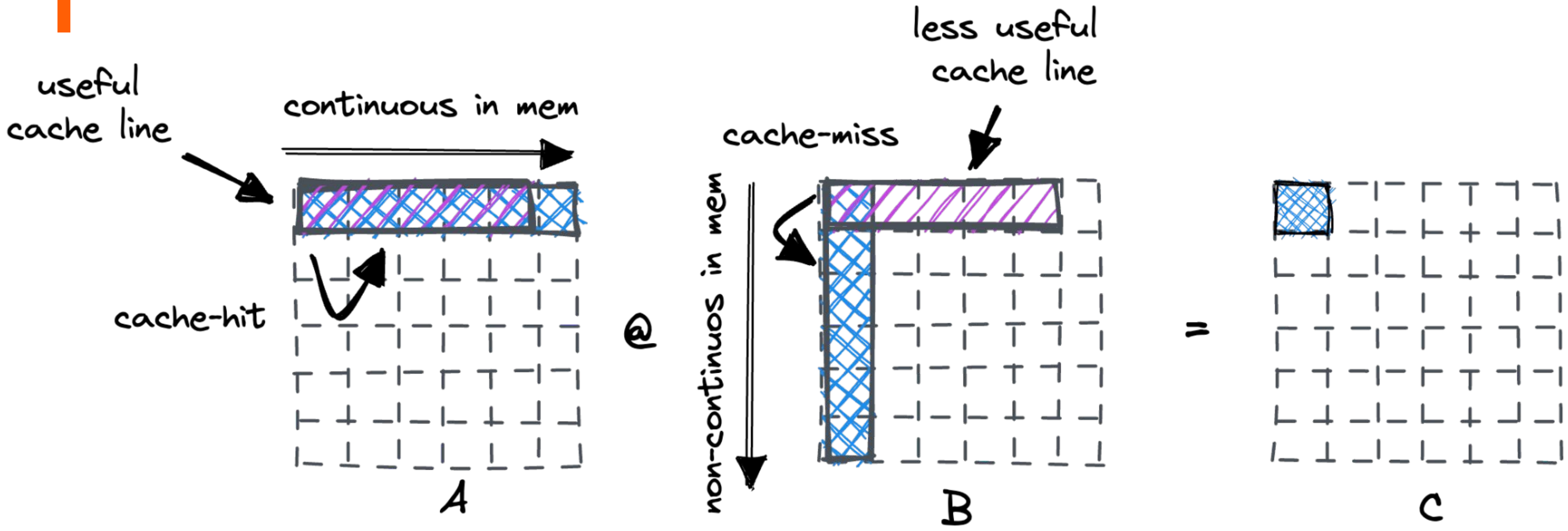


Figure 5. Hierarchical Tiling in the Triton-IR Machine Model

- Compiler needs to optimize at various levels of hardware organization
- Blocks -> Tile
- Warps -> Micro-Tile
- Threads -> Nano-Tile

Optimizations: Memory Coalescing



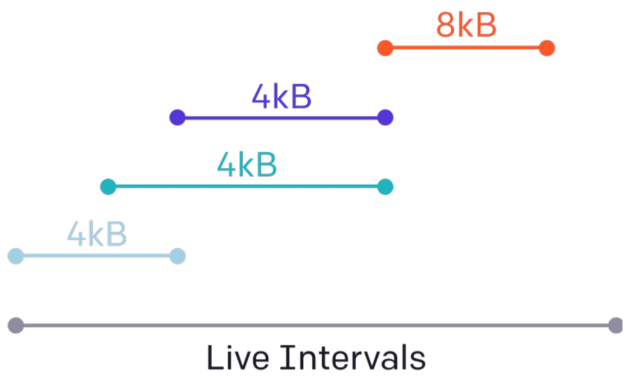
Mental model: Hardware will fetch 512 bytes from HBM on any access

For max perf, we want all threads to access elems from nearby locations

Image Source: https://siboehm.com/articles/22/Fast-MMM-on-CPU

Optimizations: Shared Memory Allocation

- Essential to avoid global memory accesses inside the kernel
- Classic liveness analysis pass on tiles
- If overlap is too high (exceed shared mem limits), need to spill to global mem



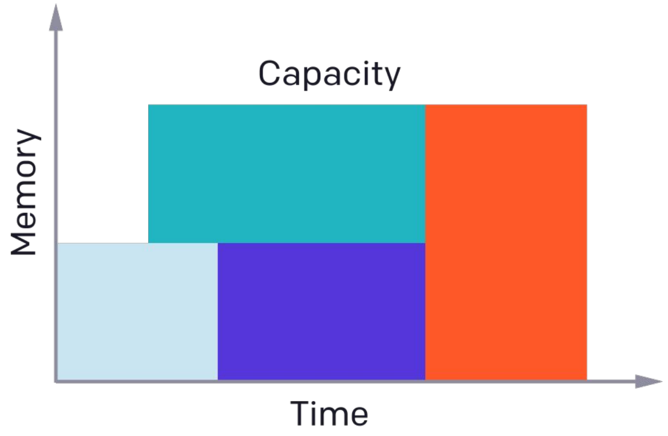


Image Source: https://openai.com/index/triton/

Optimizations: Shared Memory Synchronization

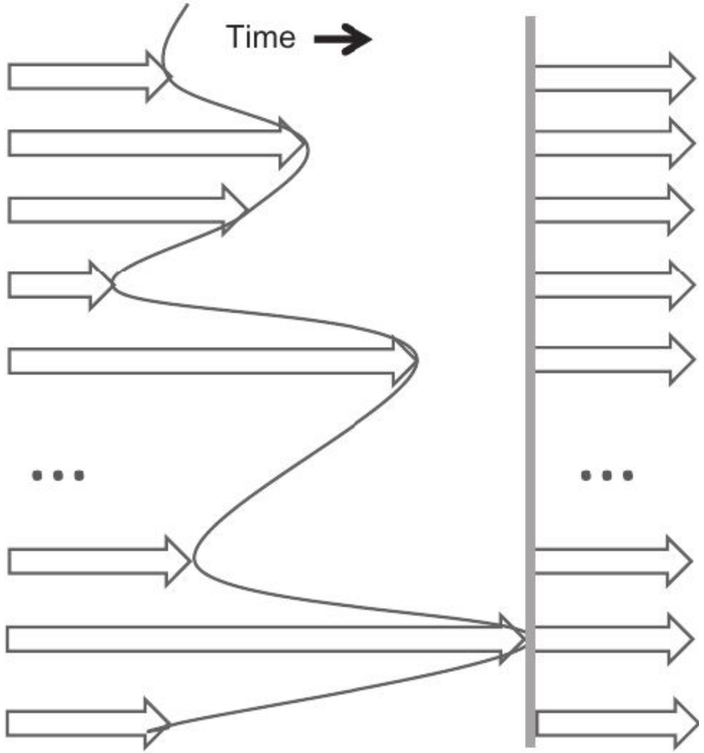


Image Source: Programming Massively Parallel Processors by Kirk and Hwu

- SIMT only at warp level
    - Warps may finish shared mem writes at different times
- During next stage, if thread reads from a location written to by another warp, no guarantee we will see update
- Need a block-level barrier
- Triton performs dataflow analysis to identify Read-after-Write and Write-after-Read deps

Extra: RAW/WAR Dataflow

$$in_s^{(RAW)} = \bigcup_{p \in \text{pred}(s)} out_p^{(RAW)}$$

$$in_s^{(WAR)} = \bigcup_{p \in \text{pred}(s)} out_p^{(WAR)}$$

$$out_s^{(RAW)} = \begin{cases} \emptyset & \textbf{if } in_s^{(RAW)} \cap read(s) \neq \emptyset \text{ (barrier)} \\ in_s^{(RAW)} \cup write(s) & \textbf{otherwise} \end{cases}$$

$$out_s^{(WAR)} = \begin{cases} \emptyset & \textbf{if } in_s^{(WAR)} \cap write(s) \neq \emptyset \text{ (barrier)} \\ in_s^{(WAR)} \cup read(s) & \textbf{otherwise} \end{cases}$$

AutoTuner

- Small component that finds best config based on execution times
- Triton only cares about tile sizes (at normal, micro, and nano levels)
- Much simpler compared to TVM

```
@triton.autotune(configs=[
   triton.Config(kwargs={'BLOCK_SIZE': 128}, num_warps=4),
   triton.Config(kwargs={'BLOCK_SIZE': 1024}, num_warps=8),
 ],
 key=['x_size'] # the two above configs will be evaluated anytime
                # the value of x_size changes
)
@triton.jit
def kernel(x_ptr, x_size, **META):
   BLOCK_SIZE = META['BLOCK_SIZE']
```

Evaluation

- Done on GTX1070
- Almost matches performance of handwritten cuBLAS kernels
  - 90% of peak device performance
- cuBLAS much better for transformers – uses 3D reductions
- A bit lacking: no sparse workloads, which are supposed to be better than TVM

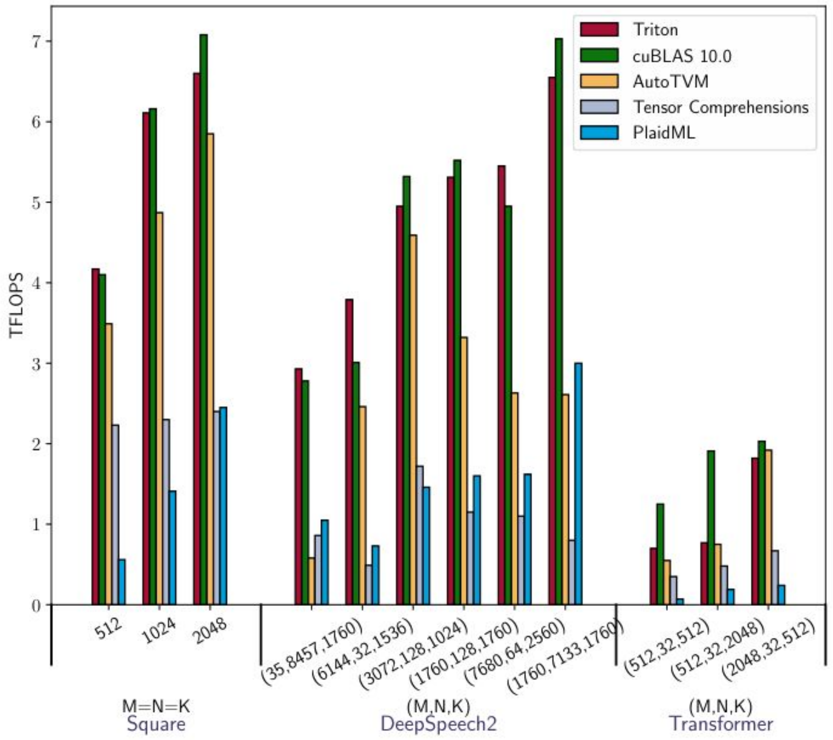


Figure 8. Performance of matrix multiplication

Triton – now

OpenAI adoption, PyTorch, and more

PyTorch 2.0 uses Triton

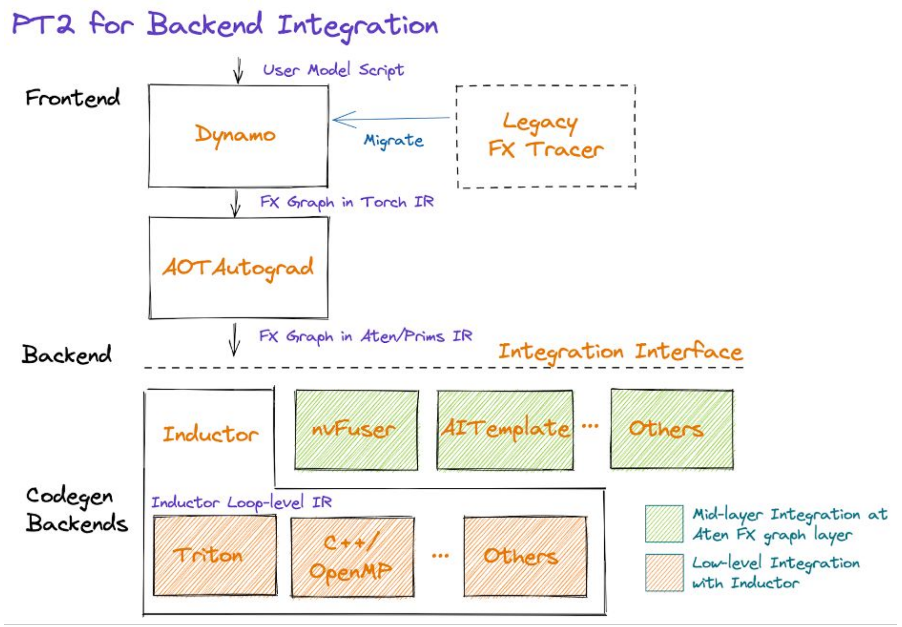


Image Source: https://pytorch.org/get-started/pytorch-2.0/

- Torch compiler traces python code and generates Triton for GPUs
- 86% performance improvement for training on Nvidia's A100 and 26% on CPUs for inference!
- Uses Triton only for GPUs
  - Hints at non-generalizability of Triton to other hardware

Why not TVM?

“On NVIDIA GPUs, we have observed **better performance results from Triton** than TVM on most models. [...] **TVM performance varies greatly depending on autotuning**. I think the strength of TVM is in its many non-GPU execution targets, while Triton is GPU-only.”

-From PyTorch 2.0 compiler dev (source)

- Essentially, TVM has too much to optimize – need to handwrite the schedules, or hope the autotuner finds a good configuration
- Similar argument for polyhedral frameworks – the ILP solvers cannot deal with large problem spaces

Triton backend rewrite

- Common complaint – CUDA focused
- Triton 2.0 rewritten in MLIR
- MLIR Principle: Different IRs are better suited for different opts
- Supports multiple dialects with successive lowering
- CPU backend is still WIP

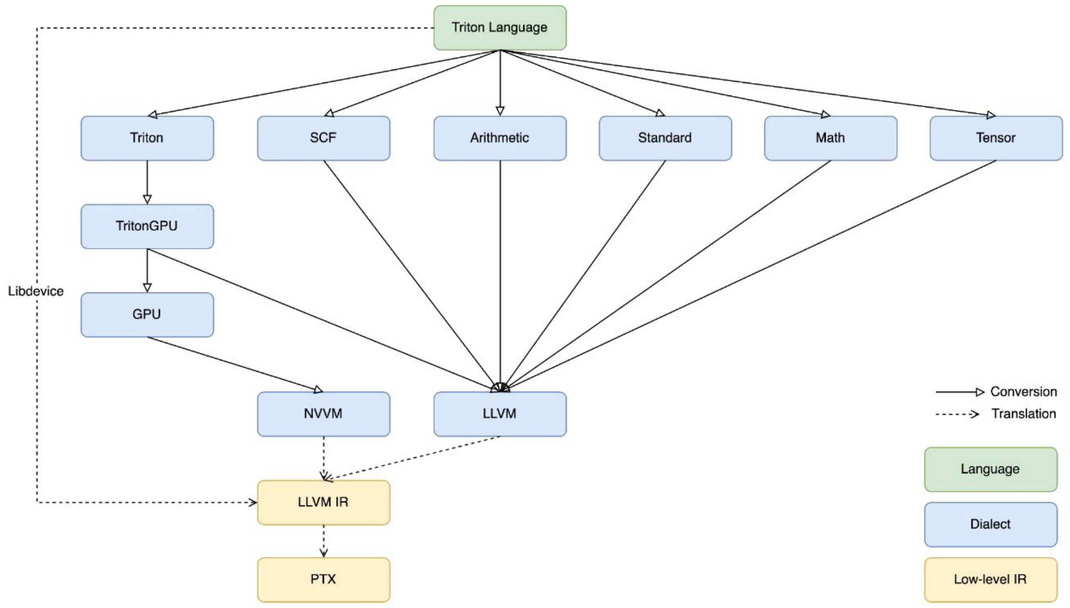


Image Source: https://www.jokeren.tech/slides/triton_next.pdf

Thoughts

Strengths:

1. Easy to use, rapid iteration time
2. Uses standard compiler techniques at tile-level: powerful idea

Weaknesses:

1. Paper’s evaluation was a bit lacking – missing ablation studies on optimizations
2. Very focused on NVIDIA GPUs. Compilers are supposed to make things easier for hardware developers too! (Triton is slowly getting better at this)
3. Syntactic matching when offloading to accelerator intrinsics. Would be nice to find semantic equivalences (3LA, Glenside tackle this)

Future Directions

1. Support for other hardware. Getting CPU codegen working on Triton took ~2 years! TPUs are probably much further away.
2. Need to focus more on memory. Hopper architecture brought many new features.
    - Async compute, overlapping ops
    - Memory layouts and swizzles
3. AutoTuner improvements

Let's Discuss!