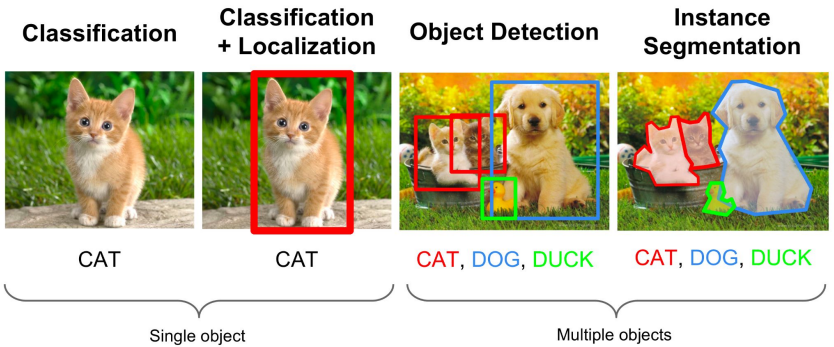
OpenAI

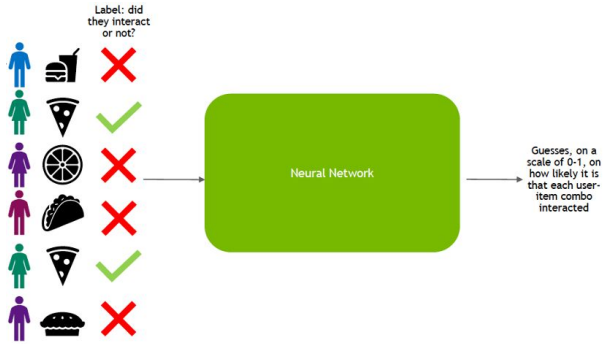
Towards Agile Development of Efficient Deep Learning Operators

Keren Zhou & Philippe Tillet

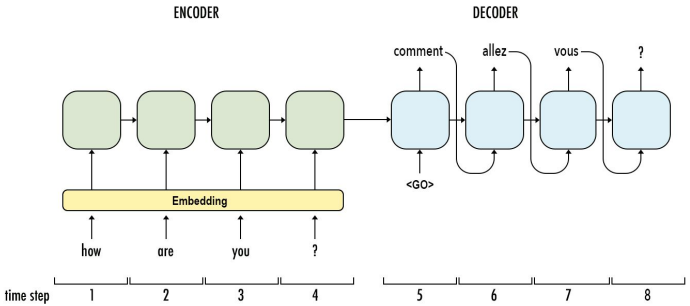
Deep Neural Networks (DNNs)



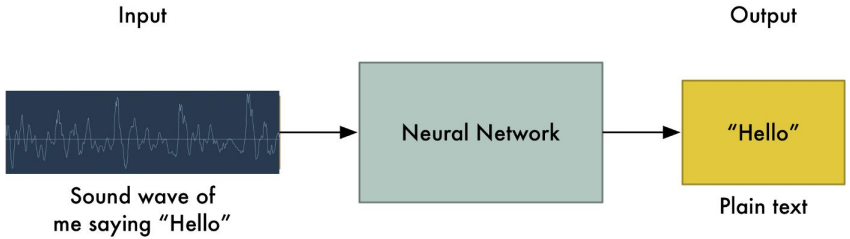
Computer Vision



Recommendation Systems



Natural Language Processing



Speech Recognition

Image sources

https://chaosmail.github.io/deeplearning/2016/10/22/intro-to-deep-learning-for-computer-vision/
https://medium.com/@ageitgey/machine-learning-is-fun-part-6-how-to-do-speech-recognition-with-deep-learning-28293c162f7a
https://towardsdatascience.com/language-translation-with-rnns-d84d43b40571
https://developer.nvidia.com/blog/how-to-build-a-winning-recommendation-system-part-2-deep-learning-for-recommender-systems/

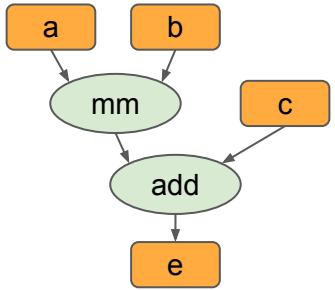
Transform DNNs to Low Level Code

```
a = torch.randn(64, 32)
b = torch.randn(32, 64)
c = torch.randn(64, 64)
d = torch.mm(a, b)
e = c + d
```

Model	Graph	Kernel	Device
• PyTorch	• XLA/HLO	• CUDA	• GPU
• TensorFlow	• TVM/Relay	• HIP	• CPU
• JAX	• PyTorch/fx	• OpenCL	• FPGA

Transform DNNs to Low Level Code

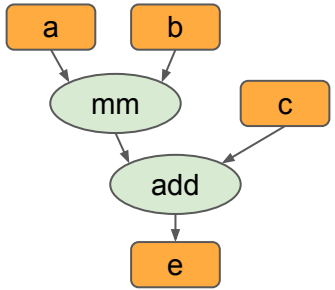
```
a = torch.randn(64, 32)
b = torch.randn(32, 64)
c = torch.randn(64, 64)
d = torch.mm(a, b)
e = c + d
```



Model	Graph	Kernel	Device
• PyTorch	• XLA/HLO	• CUDA	• GPU
• TensorFlow	• TVM/Relay	• HIP	• CPU
• JAX	• PyTorch/fx	• OpenCL	• FPGA

Transform DNNs to Low Level Code

```
a = torch.randn(64, 32)
b = torch.randn(32, 64)
c = torch.randn(64, 64)
d = torch.mm(a, b)
e = c + d
```

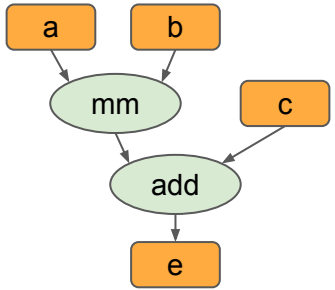



```
__global__
void mm(float *a, float *b,
float *c) {
    float *a_tile;
    float *b_tile;
    ...
}
```

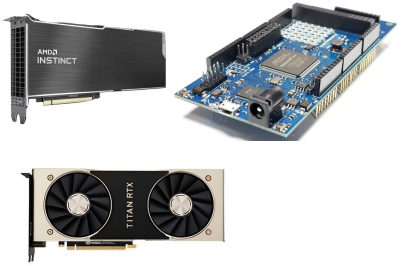
Model	Graph	Kernel	Device
• PyTorch	• XLA/HLO	• CUDA	• GPU
• TensorFlow	• TVM/Relay	• HIP	• CPU
• JAX	• PyTorch/fx	• OpenCL	• FPGA

Transform DNNs to Low Level Code

```
a = torch.randn(64, 32)
b = torch.randn(32, 64)
c = torch.randn(64, 64)
d = torch.mm(a, b)
e = c + d
```



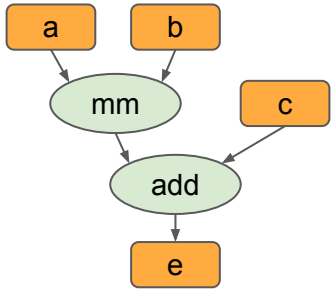

```
__global__
void mm(float *a, float *b,
float *c) {
    float *a_tile;
    float *b_tile;
    ...
}
```



Model	Graph	Kernel	Device
• PyTorch	• XLA/HLO	• CUDA	• GPU
• TensorFlow	• TVM/Relay	• HIP	• CPU
• JAX	• PyTorch/fx	• OpenCL	• FPGA

Transform DNNs to Low Level Code

```
a = torch.randn(64, 32)
b = torch.randn(32, 64)
c = torch.randn(64, 64)
d = torch.mm(a, b)
e = c + d
```




```
__global__
void mm(float *a, float *b,
float *c) {
    float *a_tile;
    float *b_tile;
    ...
}
```

Model	Graph	Kernel	Device
• PyTorch	• XLA/HLO	• CUDA	• GPU
• TensorFlow	• TVM/Relay	• HIP	• CPU
• JAX	• PyTorch/fx	• OpenCL	• FPGA

A Large Number of Tensor Operators

- Linear
  - Fused
    - Attention
    - Bilinear
  - Sparse
    - SDDMM
    - SPMM
- Convolution
  - Depthwise
  - Dilated
  - Transposed
- Pooling
  - Max/Min/Avg
  - Adaptive
- Normalization
  - Batch
  - Layer
- Loss
  - NLL
  - BCE
- Embedding
- Recurrent
  - LSTM
  - GRU

- TensorFlow: > 400 operators
- PyTorch: > 200 operators

Various Data Types

➔ Common tensor data types

- Float64
- Float32
- Float32
- Float16
- BFloat16
- Int64
- Int32
- Int16
- Int8
- Bool

For performance critical kernels:
#Implementations ≈
#Data types ✕ #Kernels

Handwritten Code

- ➔ **Low** flexibility
  - ◆ Fine-tune for every shape/data type/algorithm
  - ◆ Employ assembly instructions
  - ◆ …
- ➔ **High** performance
  - ◆ Apply sophisticated instruction/operator scheduling
  - ◆ Simplify code
  - ◆ …

Handwritten Code is a Pain

- ➔ For the company
  - ◆ Hard to recruit new Machine Learning Engineers
  - ◆ Difficult to maintain libraries
- ➔ For the researchers
  - ◆ A black box
    - ● They want to understand how kernels work
    - ● They want to fast validate new ideas at scale

Python-like Code

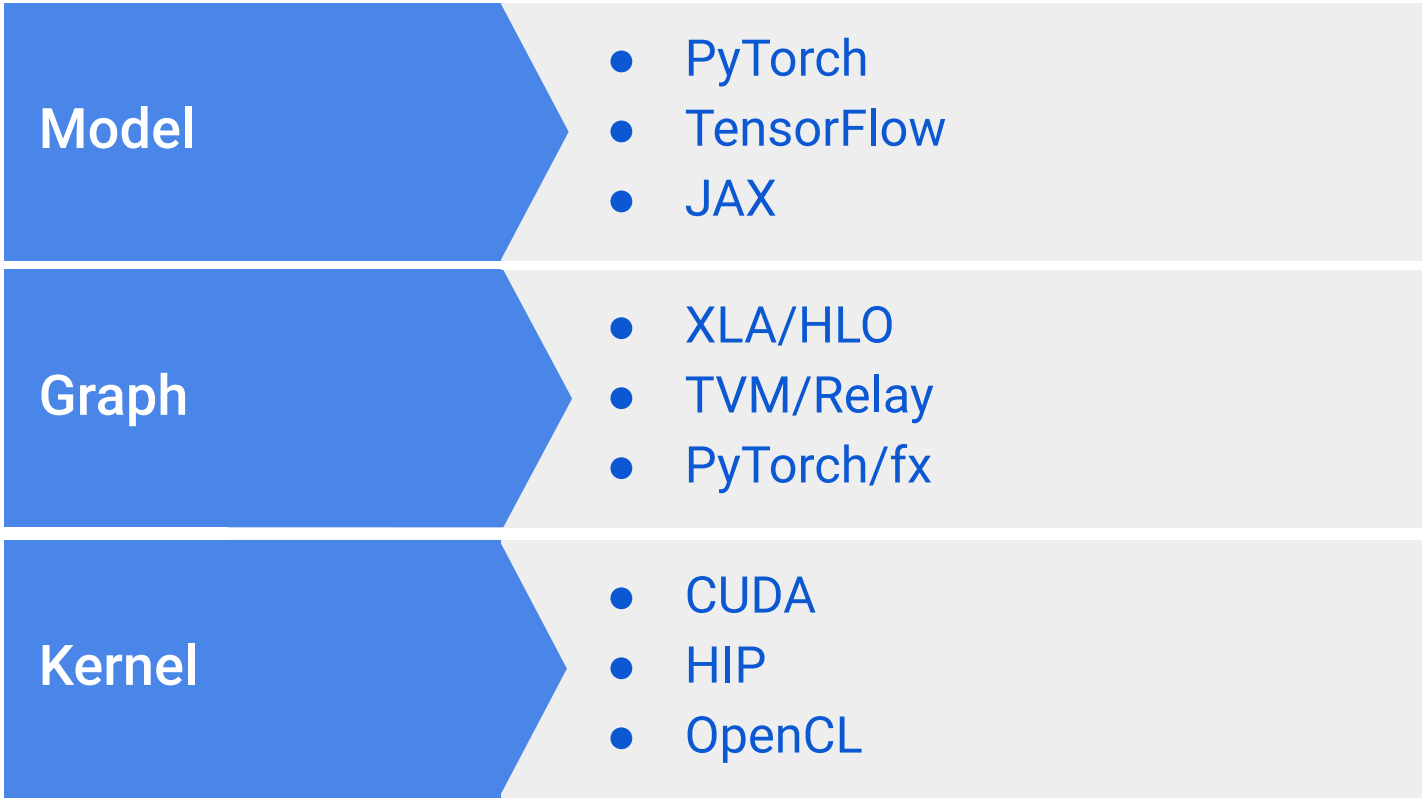
- ➔ **High** flexibility
  - ◆ Build upon existing operators
  - ◆ No need to recompile
  - ◆ …
- ➔ **Low** performance
  - ◆ Not fine-tuned for specific shapes
  - ◆ Intermediate memory movement
  - ◆ …

Can we design a language to achieve both high performance and flexibility?

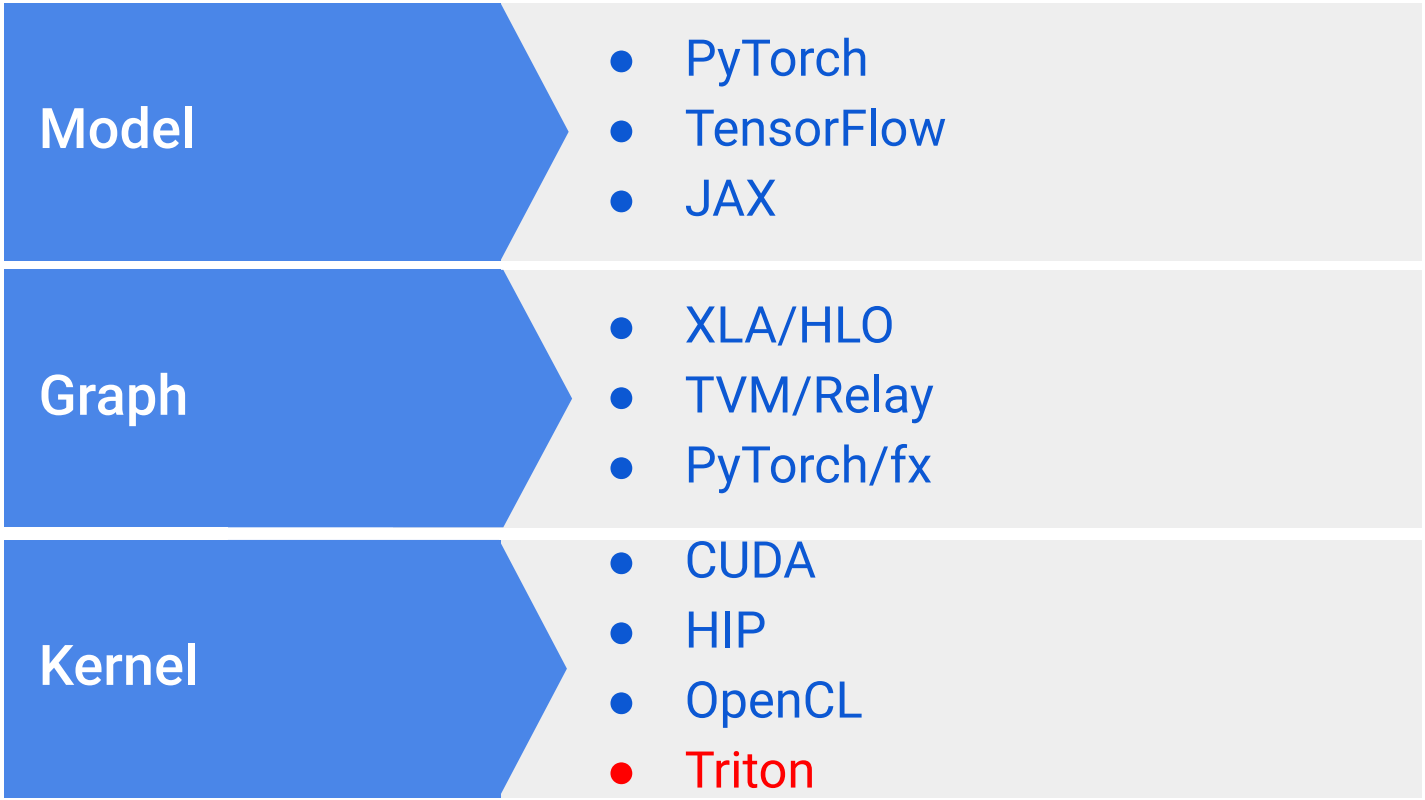
Triton

A Programming Model for the Next Generation Deep Learning Systems

Programming Models for DNNs



Programming Models for DNNs



Inefficiencies of Existing PyTorch V1 Operators

- ➔ Individual kernels
  - ◆ Can be slow
  - ◆ Can run out-of-memory
- ➔ Graph compiler
  - ◆ Don’t support custom data-structures
    - lists/trees of tensors
    - block-sparse tensors
  - ◆ Don’t support custom precision format
  - ◆ Automatic kernel fusion is limited

Solution: Employ Triton -> PyTorch V2

Triton is Designed to Achieve Both High Flexibility and Performance

➔ Flexibility

- A small core set of operations (~40 interface functions and ~20 core functions)
- Can be composed into almost all existing PyTorch operators (TorchInductor)
- SPMD but not SIMT

➔ Performance

- JIT generated kernels
- Handwritten PTX code
- Many passes to combine, simplify, and schedule operations

Triton Design

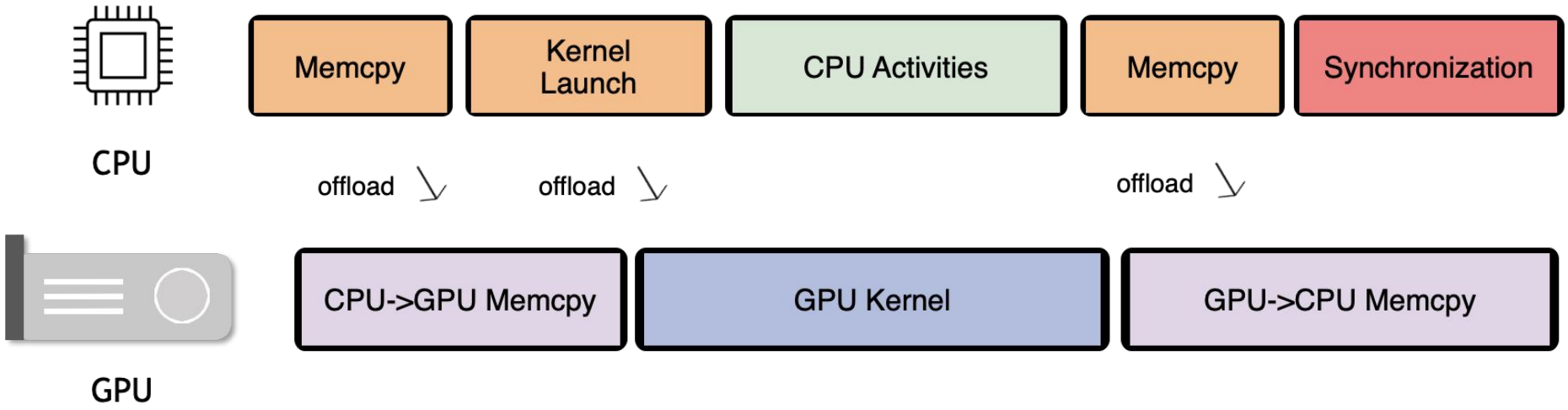
- ➔ PyTorch compatible
  - ◆ Tensors are stored on-chip rather than off-chip
  - ◆ Custom data-structures using tensors of pointers
- ➔ Python syntax
  - ◆ All standard python control flow structure (for/if/while) are supported
  - ◆ Python code is lowered to Triton IR

Write GPU Kernels Using Triton

GPU-accelerated Application Overview

➔ CPU and GPU execute asynchronously

➔ CPU dispatches commands to GPU



Terminologies

- ➔ Parallelism
  - ◆ Grid
    - One for each kernel
  - ◆ Block/Warp/Thread
- ➔ Memory
  - ◆ Global
    - Visible to all threads
  - ◆ Shared
    - Private to each block
  - ◆ Local
    - Private to each thread

CUDA *vs* Triton

	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	Not Available

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

Vector Addition (Single Block)

- Z[:] = X[:] + Y[:]
  - Without boundary check

```
import triton.language as tl
import triton
```

```
N = 1024
x = torch.randn(N, device='cuda')
y = torch.randn(N, device='cuda')
z = torch.randn(N, device='cuda')
```

Vector Addition (Boundary Check)

- Z[:] = X[:] + Y[:]
  - With boundary check
- program_id()
  - Get the block id
- mask
  - if mask[idx] is false, do not load the data at address pointer[idx]
- triton.cdiv(N, 1024)
  - (N − 1)//1024 + 1

```
@triton.jit
def _add(z_ptr, x_ptr, y_ptr, N):
    # same as torch.arange
    offsets = tl.arange(0, 1024)

    # create 1024 pointers to X, Y, Z
    x_ptrs = x_ptr + offsets
    y_ptrs = y_ptr + offsets
    z_ptrs = z_ptr + offsets
    # load 1024 elements of X, Y, Z


    # do computations
    z = x + y
    # write-back 1024 elements of X, Y, Z
```

```
N = 192311
x = torch.randn(N, device='cuda')
y = torch.randn(N, device='cuda')
z = torch.randn(N, device='cuda')
grid = (triton.cdiv(N, 1024), )
_add[grid](z, x, y, N)
```

Vector Addition (Custom Tile Size)

- ➔ Z[:] = X[:] + Y[:]
  - ◆ Each block computes TILE elements
- ➔ @triton.autotune
  - ◆ Select the best config based on the execution time
  - ◆ We don’t want to build complex autotune policies into Triton

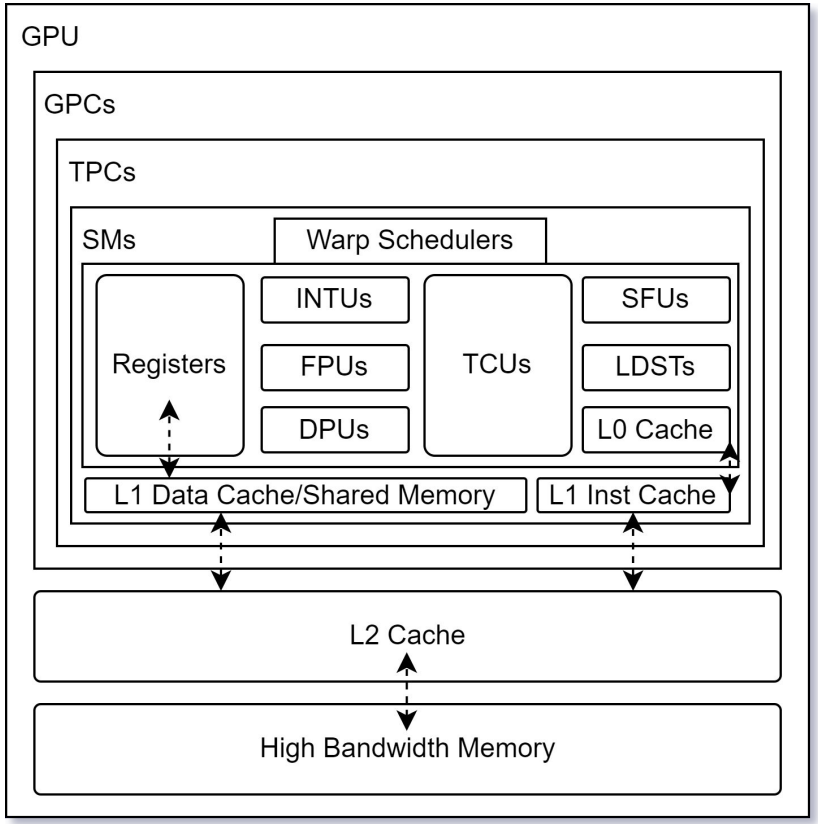
```
@triton.jit
def _add(z_ptr, x_ptr, y_ptr, N):
    # same as torch.arange
    offsets = tl.arange(0, TILE)
    offsets += tl.program_id(0)*TILE
    # create 1024 pointers to X, Y, Z
    x_ptrs = x_ptr + offsets
    y_ptrs = y_ptr + offsets
    z_ptrs = z_ptr + offsets
    # load 1024 elements of X, Y, Z
    x = tl.load(x_ptrs, mask=offset<N)
    y = tl.load(y_ptrs, mask=offset<N)
    # do computations
    z = x + y
    # write-back 1024 elements of X, Y, Z
    tl.store(z_ptrs, z, mask=offset<N)

N = 192311
x = torch.randn(N, device='cuda')
y = torch.randn(N, device='cuda')
z = torch.randn(N, device='cuda')
```

Optimizing GPU Kernels

NVIDIA GA100 Architecture & Programming Challenges

- ➔ Multiple compute units
- ➔ Multiple memory spaces
- ➔ Multiple data types
- ➔ Thread synchronization/divergence
- ➔ Tensor cores



Techniques for Optimizing a GEMM Kernel

- ➔ Vanilla (1-10% fp32 peak)
- ➔ NVIDIA CUDA Programming Guide (30%-50% fp32 peak)
  - ◆ +global memory coalesce
  - ◆ +shared memory

C++/C

- ➔ CUTLASS (80%-90% tf32 peak)
  - ◆ +vectorization
  - ◆ +shared bank conflict reduction
  - ◆ +thread layout autotune
  - ◆ +async shared memory transfer
  - ◆ +multi-stage shared memory

C++ Template & PTX

  - ◆ +tf32 tensor core
- ➔ cuBLAS (~90% tf32 peak)
  - ◆ +register bank conflict reduction
  - ◆ +control code optimization

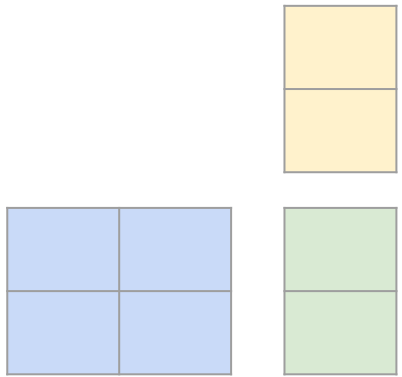
SASS

Difficulty

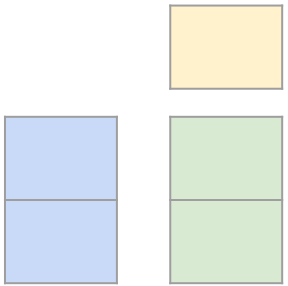
Utilizing Tensor Cores - Layout

➔ For each warp, we must load values into tiles of a specific layout to perform matrix multiplications

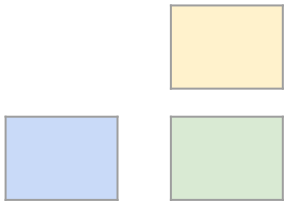
- ◆ Each data type could have multiple layouts
- ◆ Different data types (e.g., fp16 vs fp64) have different layouts



16x8x16 (fp16)



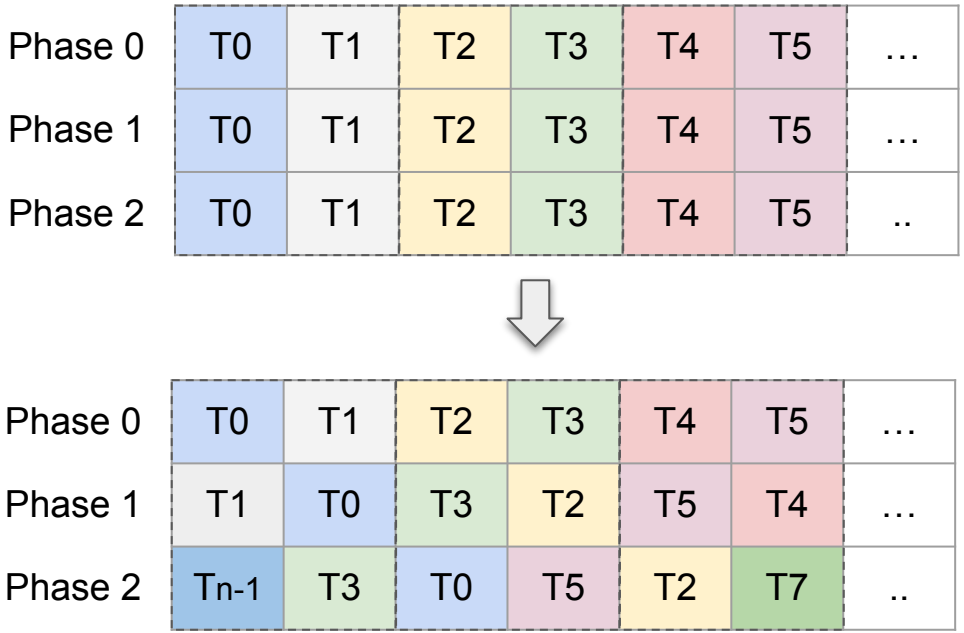
16x8x8 (fp16)



8x8x4 (fp64)

Utilizing Tensor Cores - Memory Swizzling

- ➔ Swizzling tiles (T) when loading from global memory to avoid bank conflicts
- ➔ Simple padding do not work because we need to read multiple tiles on different rows



Utilizing Tensor Cores - ldmatrix & stmatrix

- ➔ Each thread provides a pointer to 128b row of data in Shared Memory
- ➔ A row is broadcast to four threads to match the arrangement of tensor cores

ldmatrix with .num = .x1, r = {d0}								
	Col0	col1	col2	col3	col4	col5	Col6	col7
row0	%laneid = 0 dst=d0		%laneid = 1 dst=d0		%laneid = 2 dst=d0		%laneid = 3 dst=d0	
row1	%laneid = 4 dst=d0		%laneid = 5 dst=d0		%laneid = 6 dst=d0		%laneid = 7 dst=d0	
row2	%laneid = 8 dst=d0		%laneid = 9 dst=d0		%laneid = 10 dst=d0		%laneid = 11 dst=d0	
row3	%laneid = 12 dst=d0		%laneid = 13 dst=d0		%laneid = 14 dst=d0		%laneid = 15 dst=d0	
row4	%laneid = 16 dst=d0		%laneid = 17 dst=d0		%laneid = 18 dst=d0		%laneid = 19 dst=d0	
row5	%laneid = 20 dst=d0		%laneid = 21 dst=d0		%laneid = 22 dst=d0		%laneid = 23 dst=d0	
row6	%laneid = 24 dst=d0		%laneid = 25 dst=d0		%laneid = 26 dst=d0		%laneid = 27 dst=d0	
row7	%laneid = 28 dst=d0		%laneid = 29 dst=d0		%laneid = 30 dst=d0		%laneid = 31 dst=d0	

https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#warp-level-matrix-instructions-ldmatrix

Techniques for Optimizing a GEMM Kernel

- ➔ Vanilla (1-10% fp32 peak)
- ➔ NVIDIA CUDA Programming Guide (30%-50% fp32 peak)
  - ◆ +global memory coalesce
  - ◆ +shared memory

C++/C

- ➔ CUTLASS (80%-90% tf32 peak)
  - ◆ +vectorization
  - ◆ +shared bank conflict reduction
  - ◆ +thread layout autotune
  - ◆ +async shared memory transfer
  - ◆ +multi-stage shared memory

C++ Template & PTX

  - ◆ +tf32 tensor core
- ➔ cuBLAS (~90% tf32 peak)
  - ◆ +register bank conflict reduction
  - ◆ +control code optimization

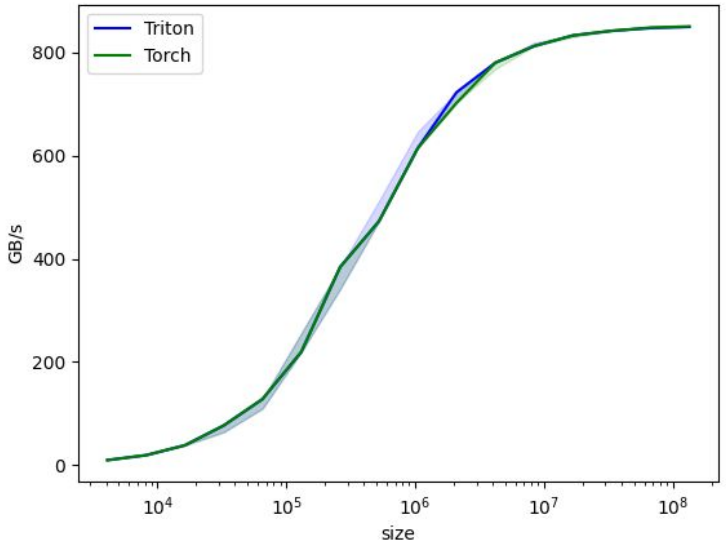
SASS

Triton applies optimizations with minimal annotations required

Difficulty

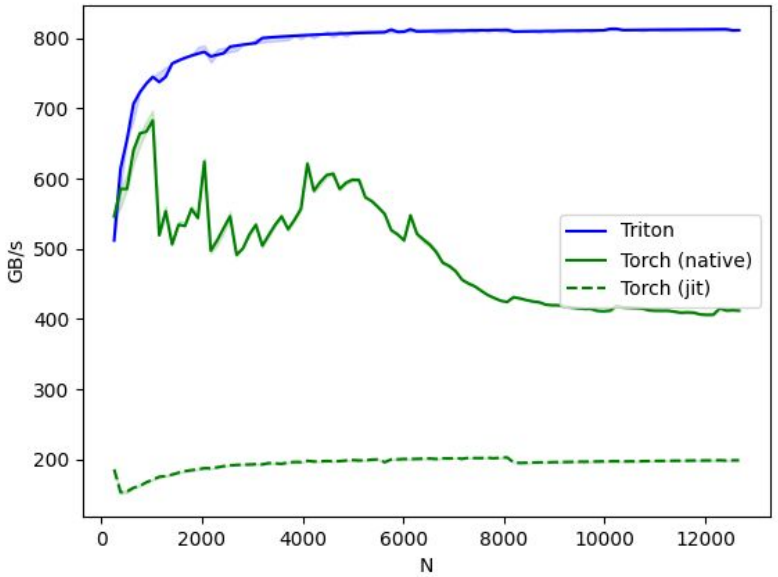
Element-wise Operators

- ➔ Triton and Torch both achieve peak bandwidth
- ➔ Researchers can write *fused element-wise operators* easily using Triton



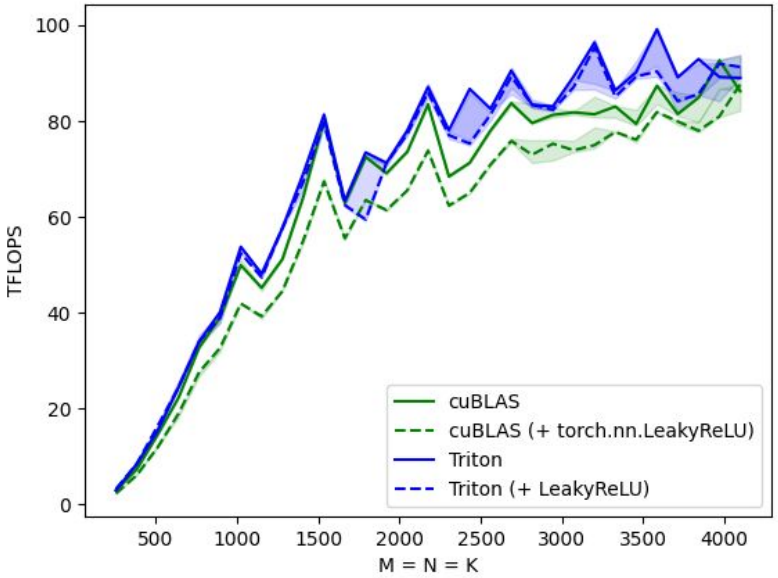
Fused Softmax

- ➔ Triton kernels can keep data on-chip throughout the entire softmax
- ➔ PyTorch JIT could in theory do that but in practice doesn't
- ➔ The native PyTorch op is designed to work for every input shape and is slower in cases where we care



Matrix Multiplication

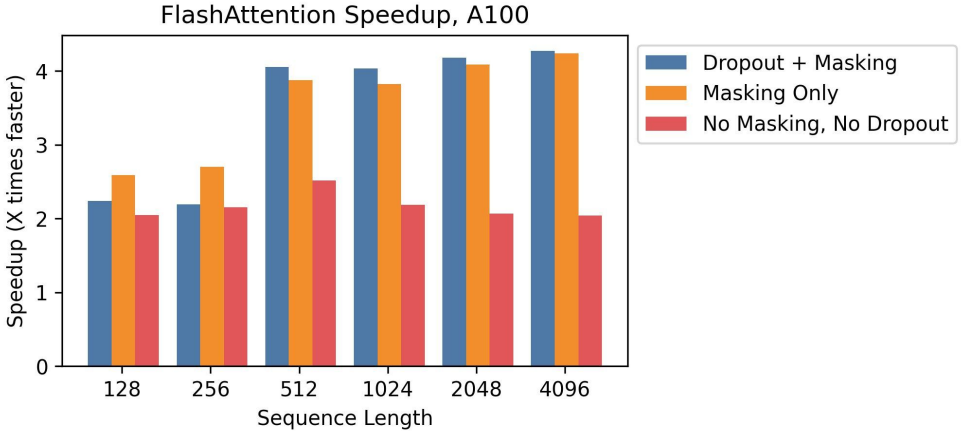
- ➔ It takes <25 lines of code to write a Triton kernel on par with cuBLAS
- ➔ Arbitrary ops can be “fused” before/after the GEMM while the data is still on-chip, leading to large speedups over PyTorch

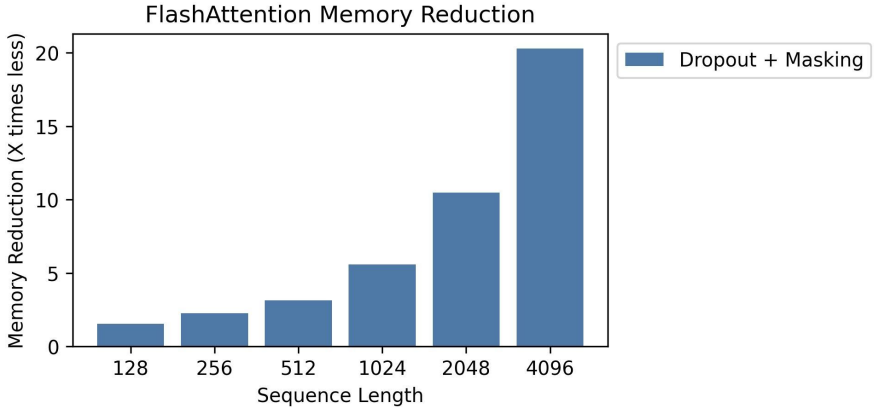


Fused Attention (Flash Attention)

➔ **From the author:** Triton is easier to understand and experiment with than CUDA

➔ Triton forward + backward is slightly slower than CUDA forward + backward



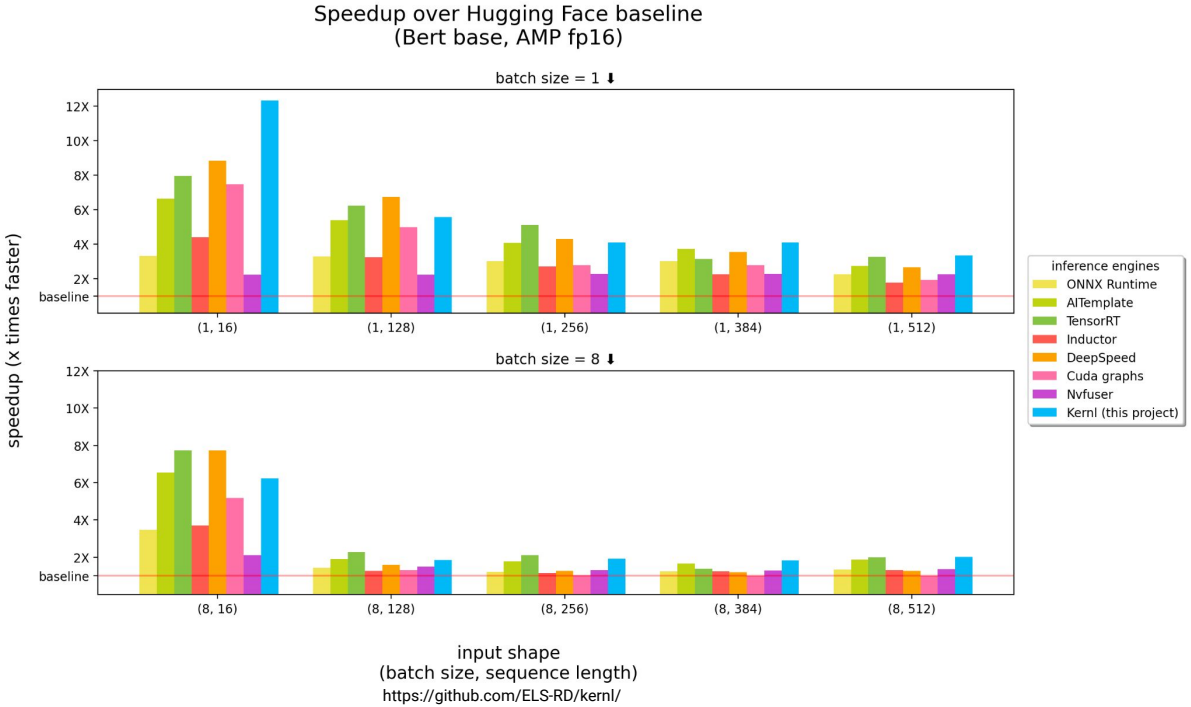


FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness
Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, Christopher Ré
Paper: https://arxiv.org/abs/2205.14135

Kernl

➔ Run PyTorch transformer models several times faster on GPU with a single line of code

➔ The first OSS inference engine written in Triton



New Challenges With Hopper

- Tensor Memory Accelerator (TMA)
  - Transfer large blocks of data between global memory and shared memory
- Distributed Shared Memory
  - Direct communication between shared memory on different SMs
- Thread Block Cluster
  - Cluster -> Grid -> Block -> Warp
- FP8 Data Types and Mode (Transformer Engine)
  - Native FP8 tensor core

Triton-MLIR (Triton V2)

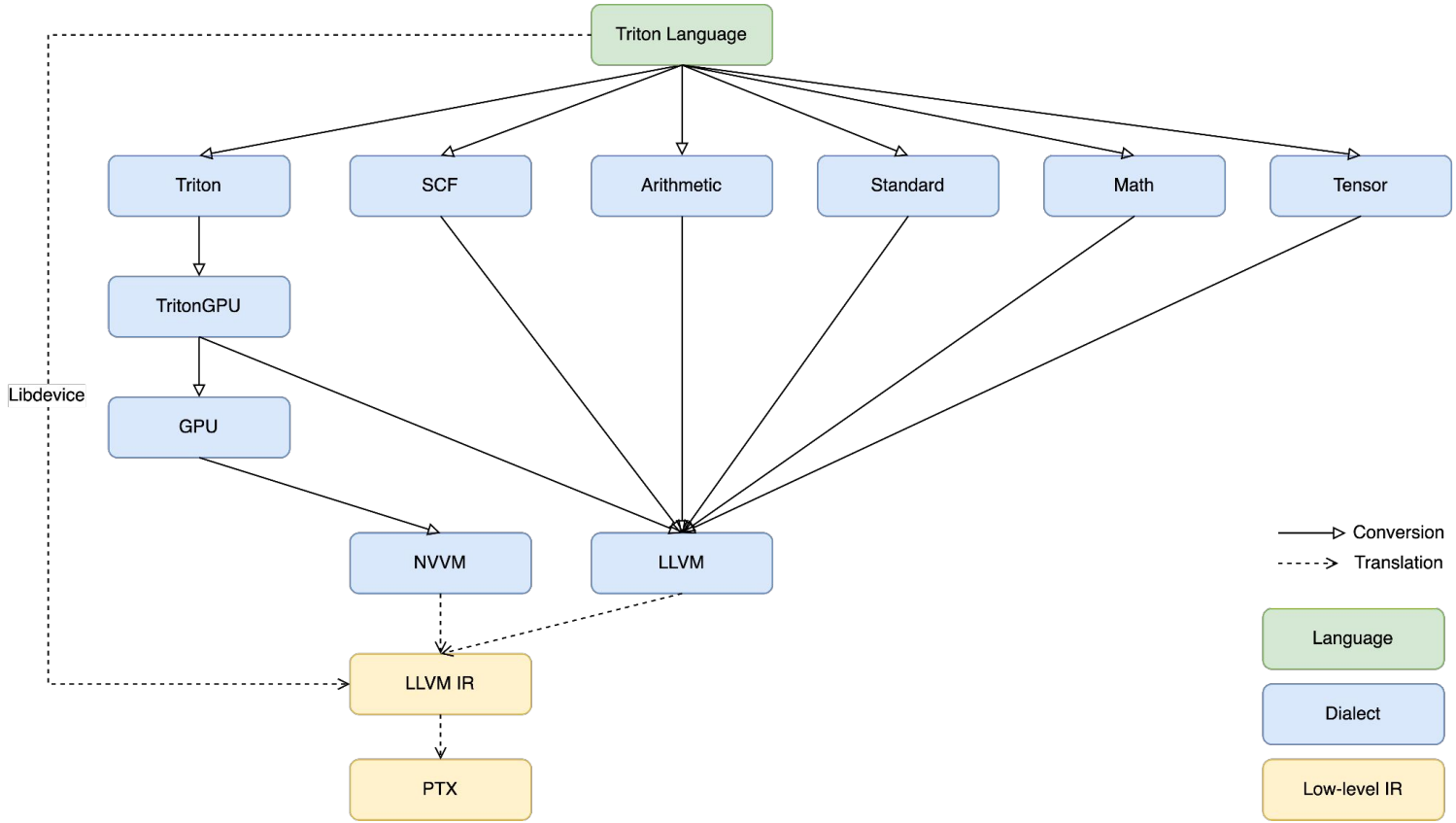
Goals

- ➔ Make Triton more robust
- ➔ Using existing infrastructure to avoid creating new wheels
- ➔ Support more backends

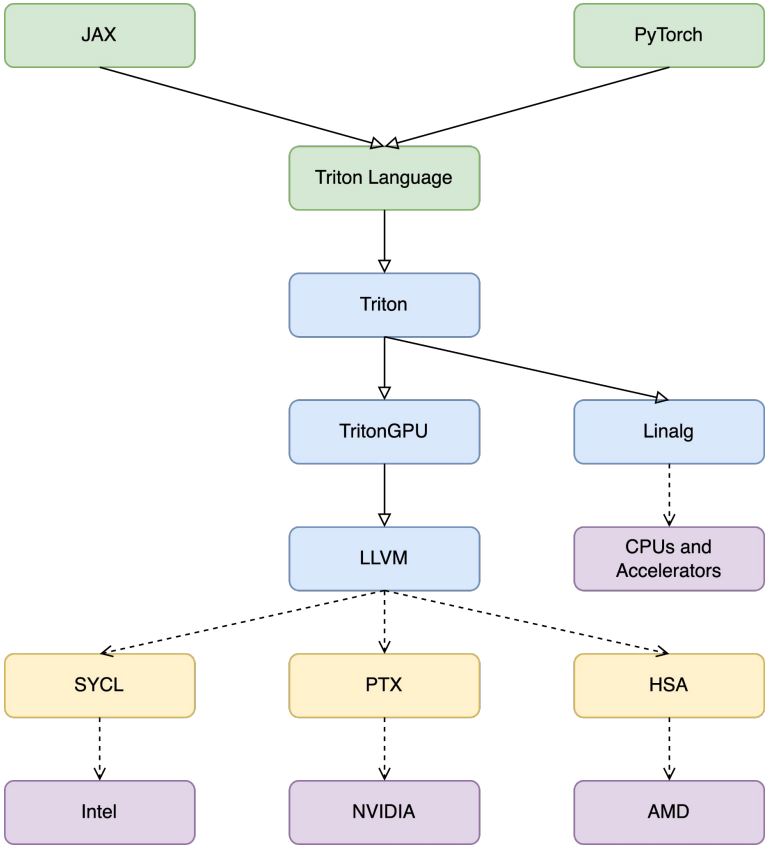
Features

- ➔ MLIR (Multi-level intermediate representation)
  - ◆ Triton dialect
  - ◆ TritonGPU dialect
- ➔ Clean layout concepts
  - ◆ Distributed, Sliced, Blocked, Shared, DotOperand
  - ◆ Adopted by CUTLASS (CuTe)
- ➔ Low overhead runtime
  - ◆ Cache and fetch kernels using efficient signatures
- ➔ Debugging
  - ◆ triton.language.print
- ➔ Profiler interface
  - ◆ Kernel launch hooks
  - ◆ Compilation hooks

Hierarchical Design



Multiple Frontends and Backends (In Progress)



Contributors

Anthropic

Da Yan

Meta

Shintaro Iwasaki

Microsoft

Ian Bearman

NVIDIA

Dongdong Li, Qingyi Liu, Chunwei Yan, Jun Yang, Chenggang Zhao, Ben Zhang, Goostavz Zhu

Takeaways

- ➔ Triton is designed to achieve both high performance and flexibility
- ➔ Triton V2 will be more robust than Triton V1
- ➔ Triton will support more backends other than NVIDIA GPUs soon

Thank You

Visit openai.com for more information.

FOLLOW @OPENAI ON TWITTER