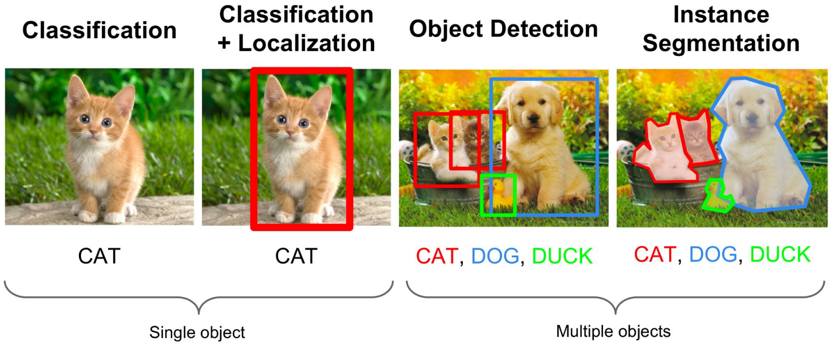
OpenAI

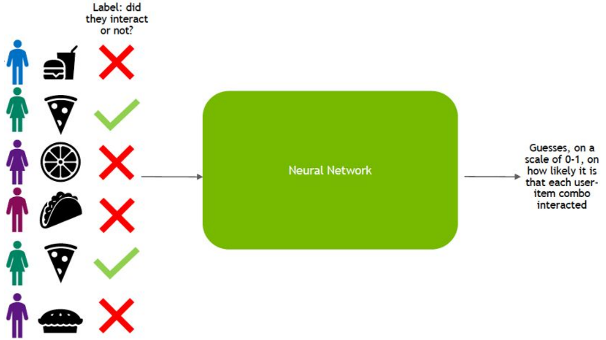
# Towards Agile Development of Efficient Deep Learning Operators

Keren Zhou & Philippe Tillet

# Deep Neural Networks (DNNs)



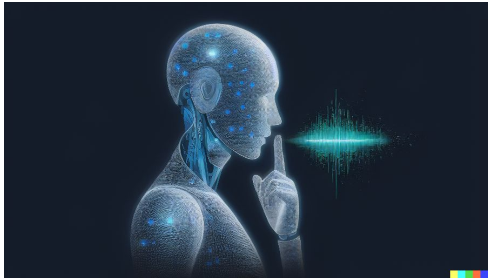

Computer Vision




Recommendation Systems

ChatGPT

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/*

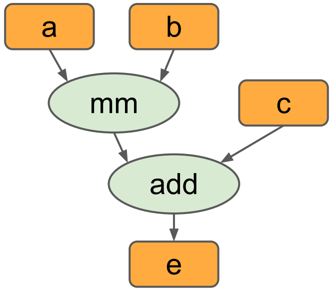
# Transform DNNs to Low Level Code

```
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<br>• TensorFlow<br>• JAX | • XLA/HLO<br>• TVM/Relay<br>• PyTorch/fx | • CUDA<br>• HIP<br>• OpenCL | • GPU<br>• CPU<br>• FPGA |

# Transform DNNs to Low Level Code

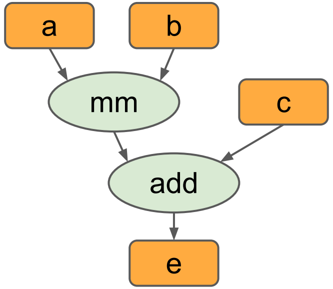
```
a = torch.randn(64, 32)
b = torch.randn(32, 64)
c = torch.randn(64, 64)
d = torch.mm(a, b)
e = c + d
```




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

# Transform DNNs to Low Level Code

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




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

Model

- PyTorch
- TensorFlow
- JAX

Graph

- XLA/HLO
- TVM/Relay
- TorchDynamo

Kernel

- CUDA
- HIP
- OpenCL

Device

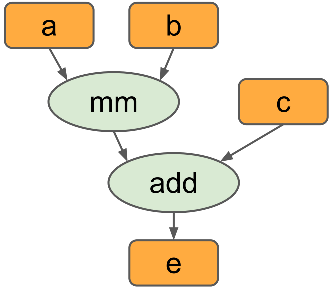
- GPU
- CPU
- FPGA

# Transform DNNs to Low Level Code

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

Model

- **PyTorch**
- **TensorFlow**
- **JAX**



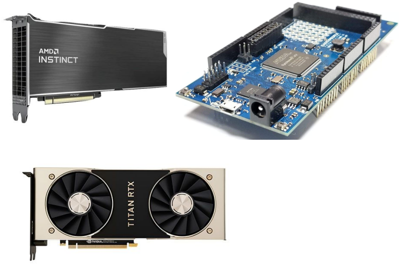

Graph

- **XLA/HLO**
- **TVM/Relay**
- **TorchDynamo**

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

Kernel

- **CUDA**
- **HIP**
- **OpenCL**

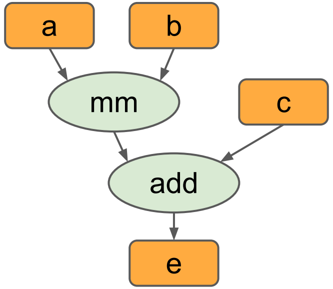



Device

- **GPU**
- **CPU**
- **FPGA**

# Transform DNNs to Low Level Code

```python
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 | TorchDynamo | 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

Thousands of Operators in PyTorch and TensorFlow

# 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 hire 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

**Model**

- PyTorch
- TensorFlow
- JAX

**Graph**

- XLA/HLO
- TVM/Relay
- TorchDynamo

**Kernel**

- CUDA
- HIP
- OpenCL

# Programming Models for DNNs

**Model**

- PyTorch
- TensorFlow
- JAX

**Graph**

- XLA/HLO
- TVM/Relay
- TorchDynamo

**Kernel**

- CUDA
- HIP
- OpenCL
- Triton

# Inefficiencies of PyTorch V1

- A neural network with individual kernels
  - Can be slow
  - Can run out-of-memory
- A neural network with graph compiler (TorchScript)
  - 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 (TorchDynamo)

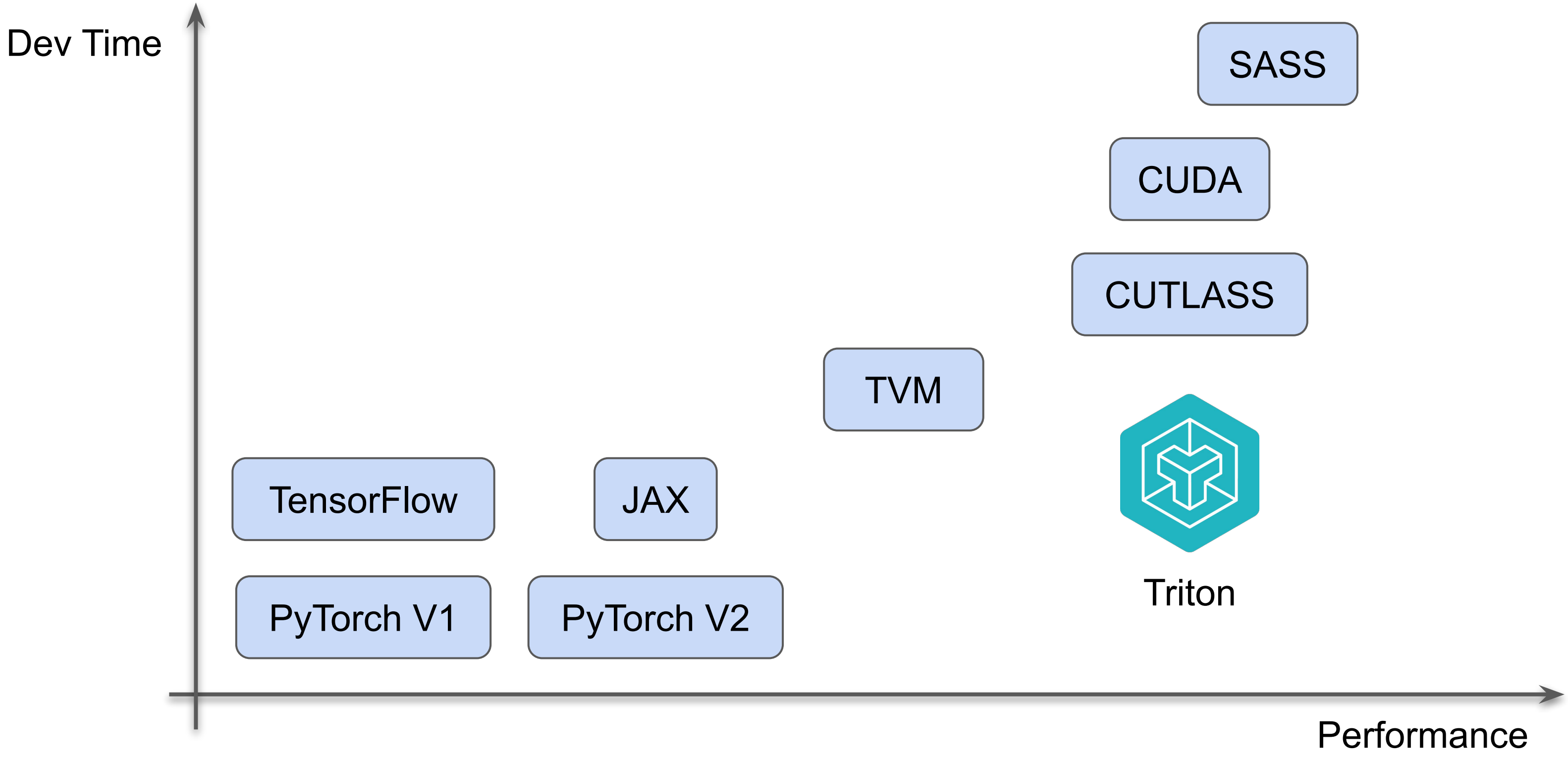
# Triton is Designed to Achieve Both High Flexibility and Performance

- 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 is a Python-Like Language

- PyTorch compatible
  - Inputs can be PyTorch tensors or custom data-structures (e.g., tensors of pointers)
- Python syntax
  - All standard python control flow structure (for/if/while/return) are supported
  - Python code is lowered to Triton IR

# Dev Time *VS* Performance



# Write GPU Kernels Using Triton

# Terminologies

- Parallelism
  - Grid
    - One for each kernel (Pre-Hopper)
  - 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 | Support |

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

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

```python
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

```
@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

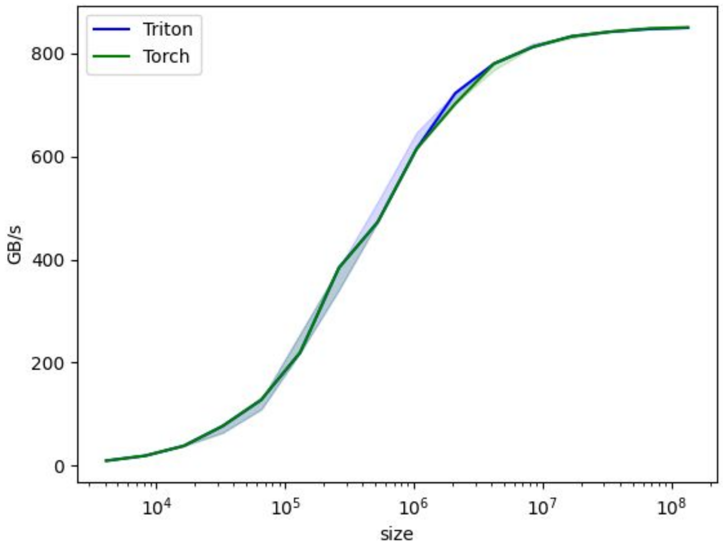
```python
@triton.jit
def _add(z_ptr, x_ptr, y_ptr, N):
    # same as torch.arange
    offsets = tl.arange(0, TILE)
    offsets += tl.program_id(0)*TILE
    # create 128/256 pointers to X, Y, Z
    x_ptrs = x_ptr + offsets
    y_ptrs = y_ptr + offsets
    z_ptrs = z_ptr + offsets
    # load 128/256 elements 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 128/256 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')
```

# Performance of Triton Kernels

# Element-wise Operators

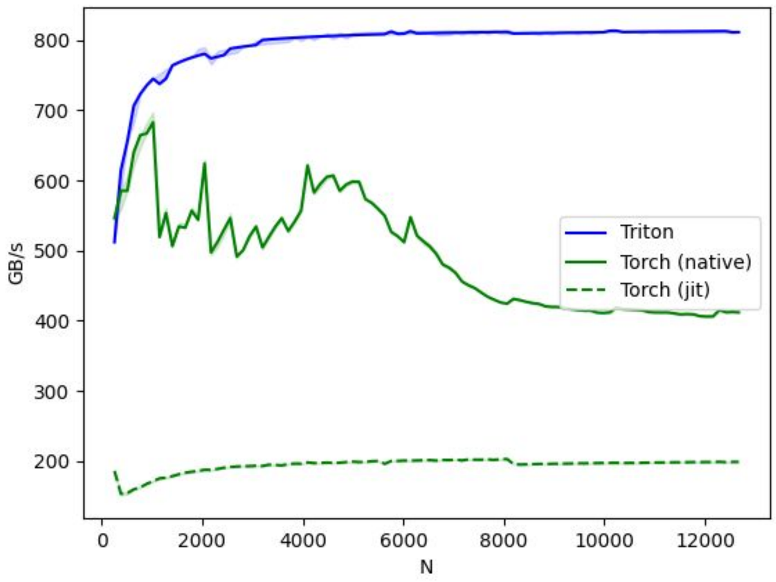
➔ Triton and Torch both achieve peak bandwidth

➔ Researchers can write *fused element-wise operators* easily using Triton



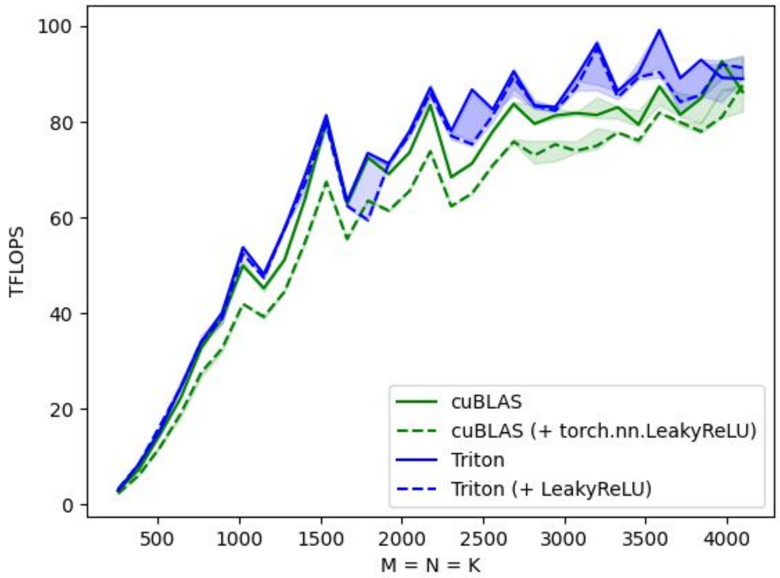
# Fused Softmax

- ➔ 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

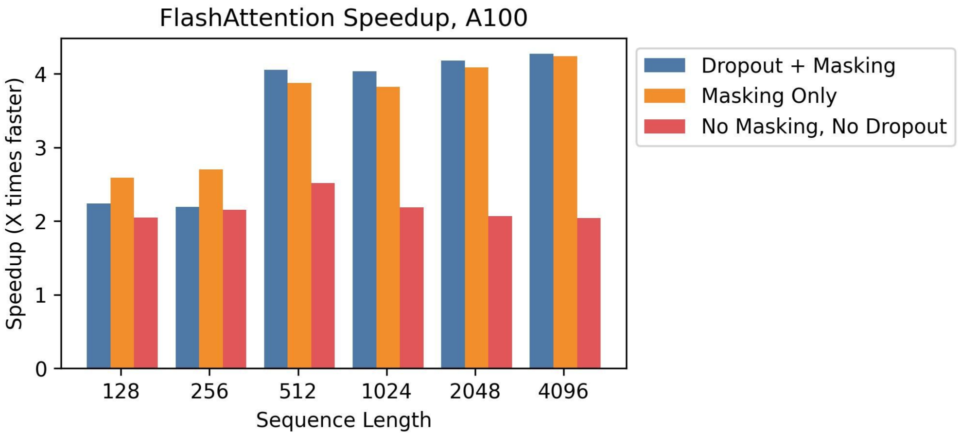
- ➔ It takes <25 lines of code to write a Triton kernel on par with cuBLAS
- ➔ Arbitrary ops can be “fused” before/after the GEMM while the data is still on-chip, leading to large speedups over PyTorch

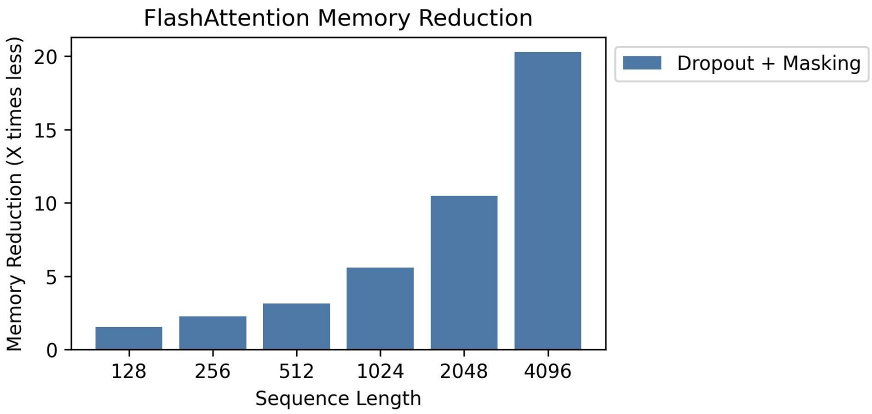


# Fused Attention (Flash Attention)

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

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

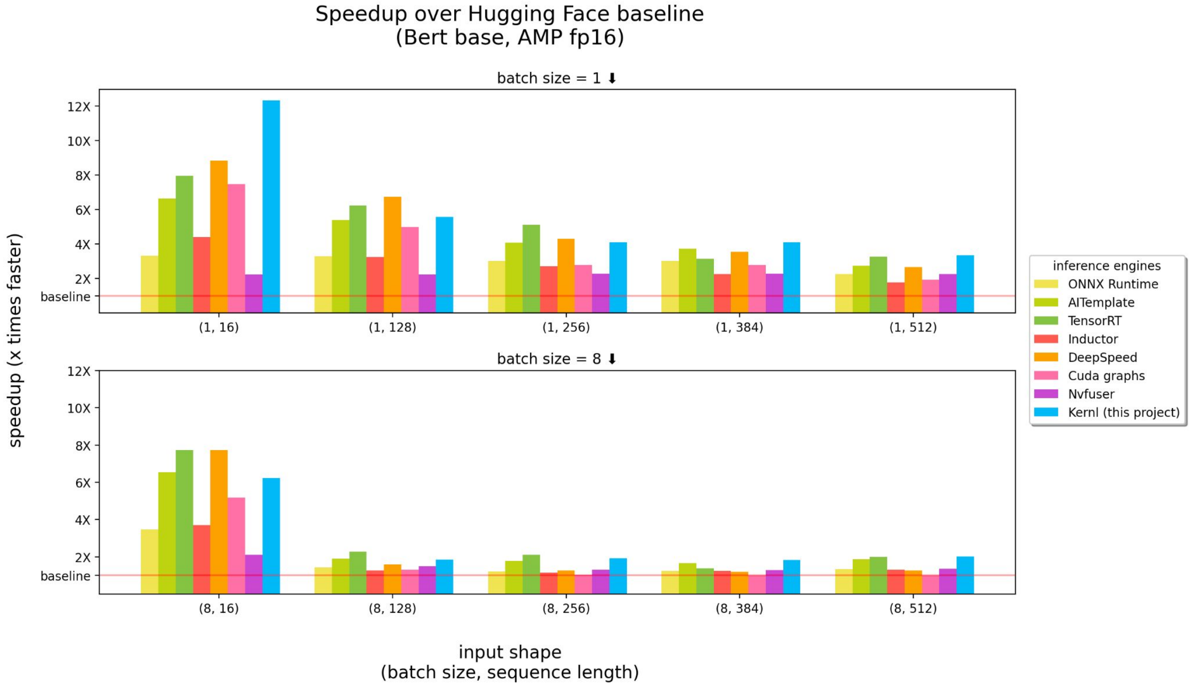



FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness
Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, Christopher Ré
Paper: https://arxiv.org/abs/2205.14135

# Kernl

- ➔ Run PyTorch transformer models several times faster on GPU with a single line of code
- ➔ The first OSS inference engine written in Triton




https://github.com/ELS-RD/kernl/

# Contributing to Triton

# Goals

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

# Ecosystem


deepspeed
tinygrad
kernl.ai
Runtime
Debugger
Profiler
PyTorch
Language
OpenXLA
IREE
Backends

# Debugger Status & Roadmap

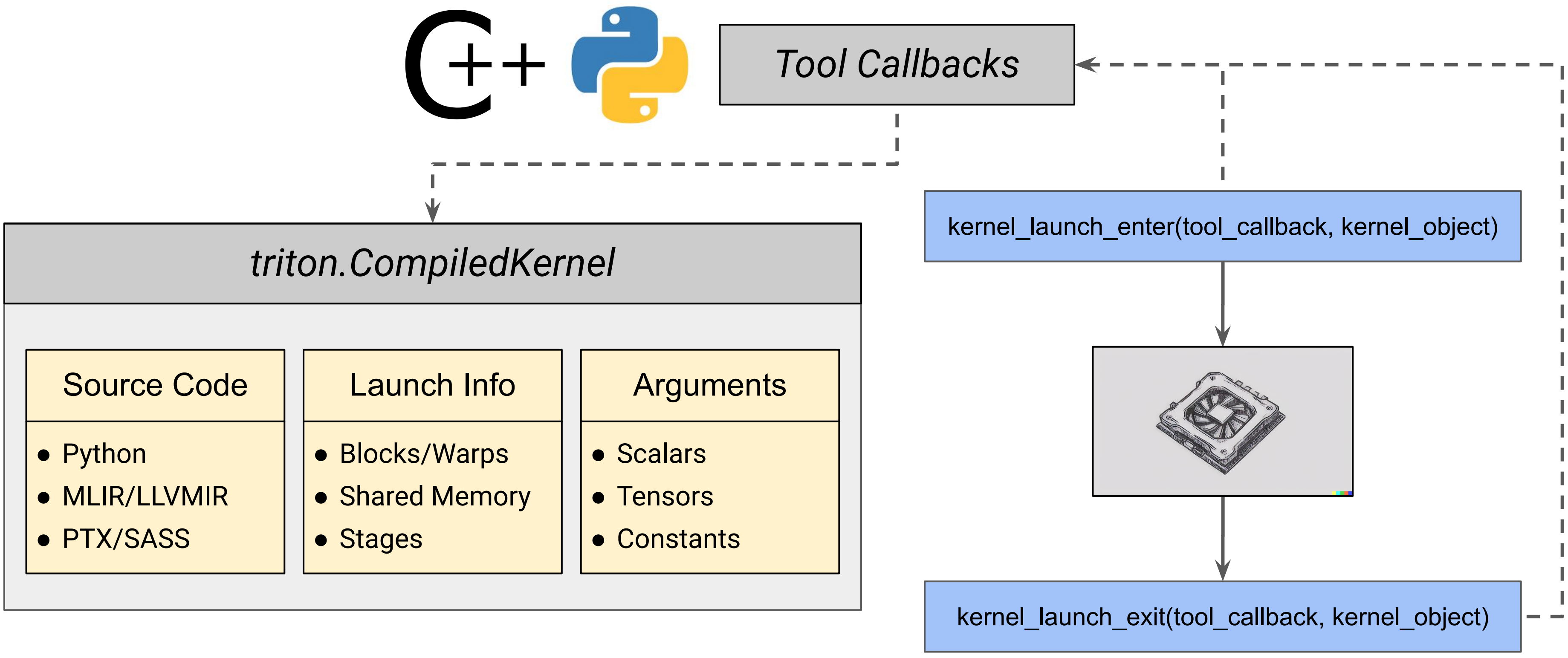
- ➔ Offloading mode (in progress)
  - ◆ Translate from Triton ops to PyTorch ops
    - Facilitate debugging *algorithm/numerical* issue
- ➔ Native mode (proposed)
  - ◆ Assemble relevant line mapping information
    - Attribute out-of-bound memory accesses from SASS to Triton
    - Understand conversions between compiler transformation passes
- ➔ **Call for contributions!**

# Profiler Status & Roadmap

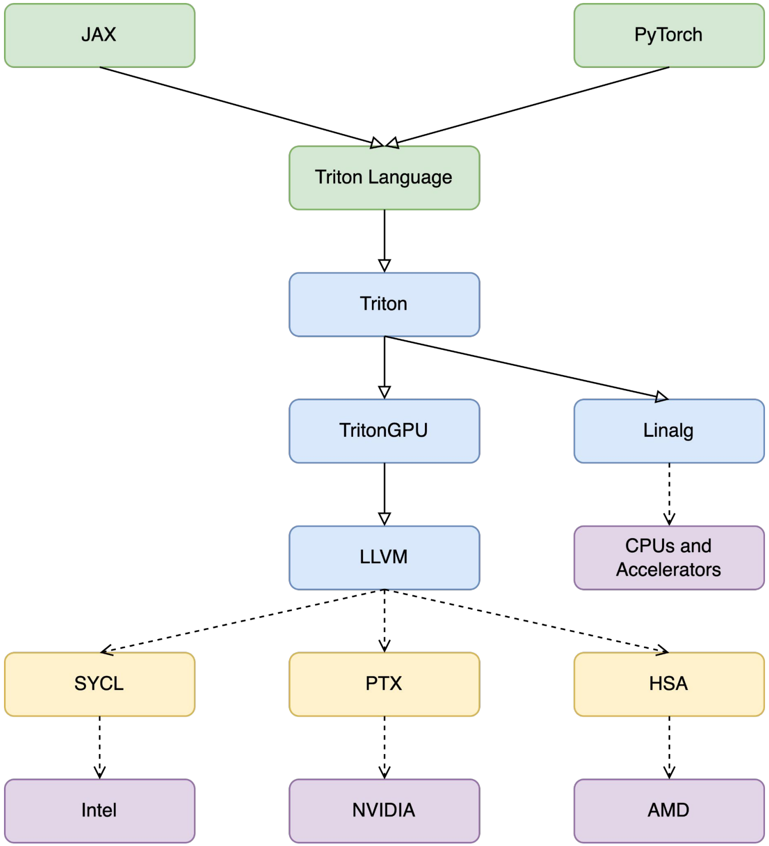
➔ Key objective: Provide low-overhead callbacks and essential kernel information for external tools

- Avoid unnecessary reinvention of existing solutions
  - hpctoolkit/tau/nsight
- Allow tools to instrument at multiple levels
  - Python/TritonIR/TritonGPUIR
- Retain Triton’s focuses on the design and optimization of the language

# Callback Design



# Backend Status



# Takeaways

- ➔ Triton is designed to achieve both high performance and flexibility
- ➔ Triton has been used widely in open source projects
- ➔ Triton supports multiple GPU backends already, with NVIDIA GPUs provide the highest performance

# Additional Topics

- ➔ Triton for HPC?
    - ◆ Rewrite existing algorithms for maintenance and performance
- ➔ Details about Triton GPU backends?
    - ◆ Encoding/alias/membar/layout conversion
- ➔ Refactor Triton APIs to address problems on emerging GPUs?
    - ◆ CTA cluster/warp specialization/tensor slicing
- ➔ Challenges and opportunities of JIT-based code generation?

# Thank You

Visit openai.com for more information.

FOLLOW @OPENAI ON TWITTER