Technical Review on PyTorch 2.0 and Triton

Keren Zhou
George Mason University
kzhou6@gmu.edu

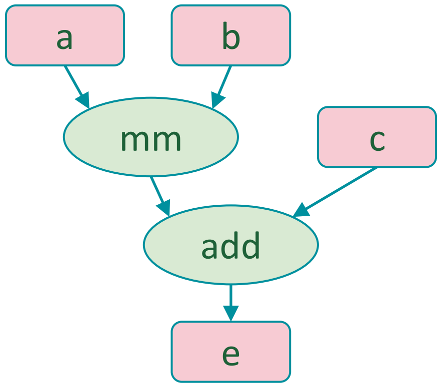
GEORGE MASON UNIVERSITY

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



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

Transform DNNs to Low Level Code

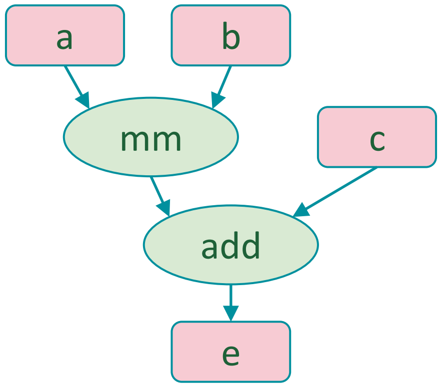
```
__global__
void mm(float *a, float *b,
float *c) {
    float *a_tile;
    float *b_tile;
    …
}
```

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

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




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

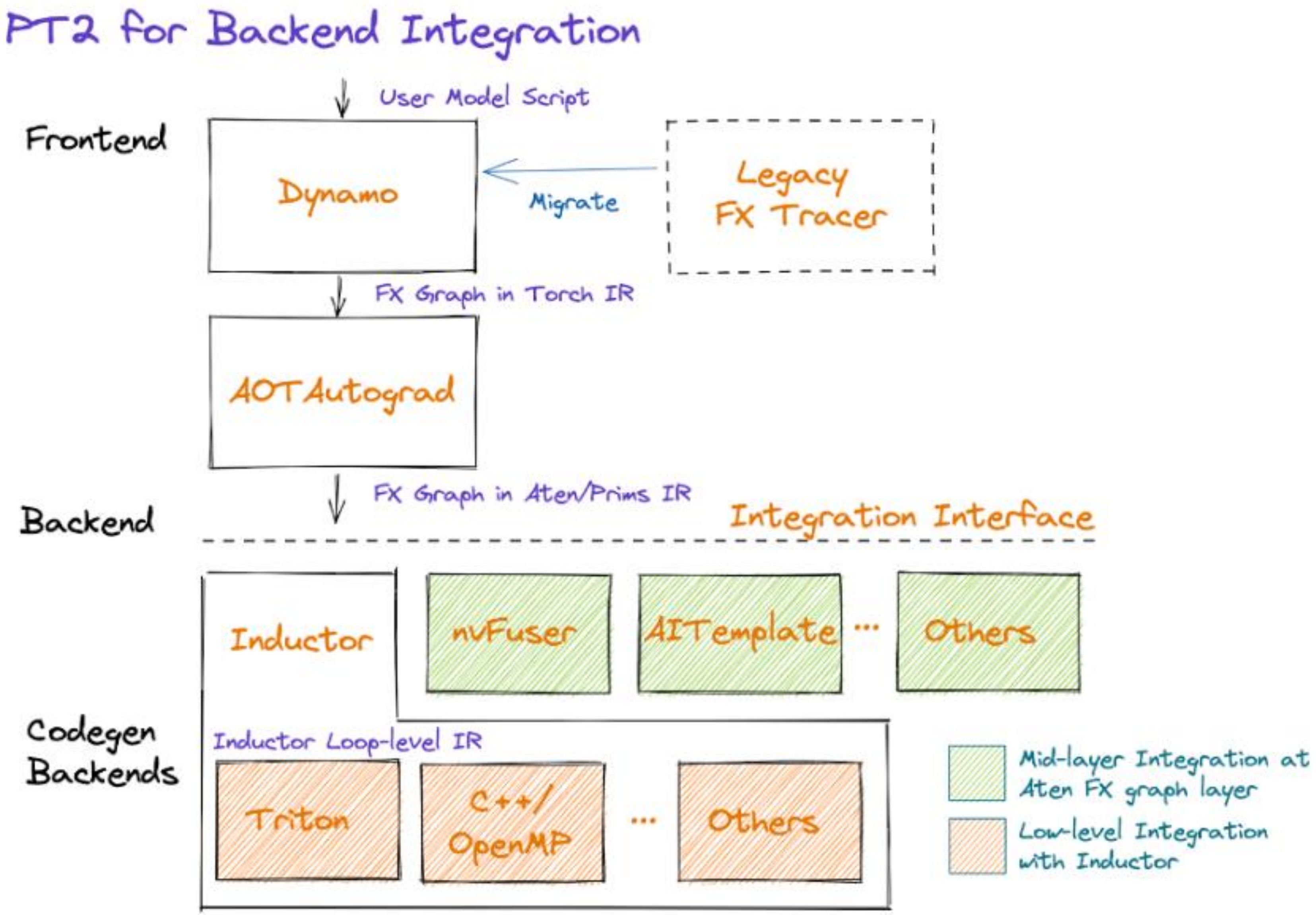
Model	**Graph**	**Kernel**	Device
• **PyTorch**	• **XLA/HLO**	• **CUDA**	• **GPU**
• **TensorFlow**	• **TVM/Relay**	• **HIP**	• **CPU**
• **JAX**	• **PyTorch/fx**	• **OpenCL**	• **FPGA**

PYTORCH 2.0

Features

- TorchDynamo
  - Captures PyTorch programs safely using Python Frame Evaluation Hooks
- AOTAutograd
  - Generating ahead-of-time backward traces
- PrimTorch
  - Canonicalizes ~2000+ PyTorch operators down to a closed set of ~250 primitive operators
- TorchInductor
  - Deep learning compiler that generates fast code for multiple accelerators and backends
  - For NVIDIA and AMD GPUs, it uses OpenAI Triton as a key building block

Overview



PT2 for Backend Integration
User Model Script
Frontend
Dynamo
Migrate
Legacy FX Tracer
FX Graph in Torch IR
AOTAutograd
FX Graph in Aten/Prims IR
Backend
Integration Interface
Inductor
nvFuser
AITemplate
...
Others
Codegen Backends
Inductor Loop-level IR
Triton
C++/ OpenMP
...
Others
Mid-layer Integration at Aten FX graph layer
Low-level Integration with Inductor

Graph Tracers Prior to PyTorch 2.0

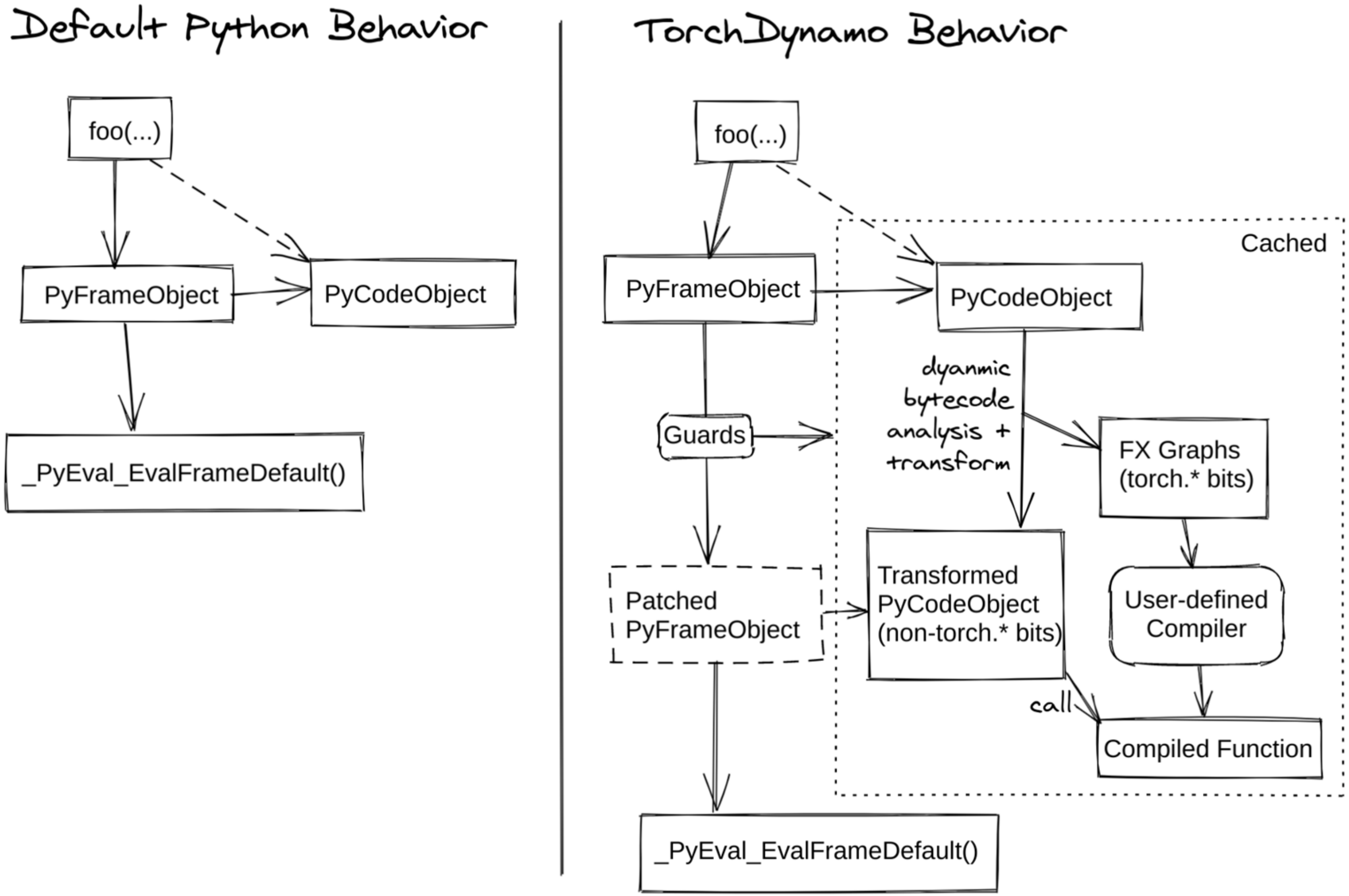
- torch.jit.trace
  - Tracing at C++ level
    - Does not capture any control flow done in Python
- torch.jit.script
  - Static Python AST analysis (i.e., visit_<syntax_name>)
    - An unimplemented component of Python makes the entire program unfit for capture
- Lazy tensors
  - Hashing the graph to avoid recompilation
    - Recompilation if any part of the graph is changed
- torch.fx.symbolic_trace
  - Tracing at python level using proxy objects
    - Silently incorrect results due to random functions and global variables

PEP 523 - Adding a frame evaluation API to CPython

- Expand CPython's C API to allow a per-interpreter function to handle the evaluation of frame
  - seval_frame = _PyEval_EvalFrameDefault by default

```
typedef struct {
    ...
    _PyFrameEvalFunction eval_frame;
} PyInterpreterState;


PyObject *
PyEval_EvalFrameEx(PyFrameObject *frame, int throwflag)
{
    PyThreadState *tstate = PyThreadState_GET();
    return tstate->interp->eval_frame(frame, throwflag);
}
```

TorchDynamo



TorchInductor

- The default “user-defined” compiler
  - Implemented in Python
- Decomposition
  - Log2 -> log * log2_scale
- Lowering
  - Use Python functions to define the bodies of loops
- Scheduling
  - Determine which kernels should be fused to achieve the best performance
- Code generation
  - GPU
    - IR->Triton Python code
  - CPU
    - IR->OpenMP/C++

Usage

- torch.compile
  - model=None
    - required
  - fullgraph=False
  - dynamic=False
  - backend='inductor'
  - mode=None
    - reduce-overhead
    - max-autotune
  - options=None
  - disable=False

- Function

```
compiled_module = torch.compile(module, …)
```

- Decorator

```
@torch.compile(fullgraph=True)
def foo(x):
    return torch.sin(x) + torch.cos(x)
```

Example

```
import torch._dynamo
import torch

def f(x):
    return torch.sin(x)**2 + torch.cos(x)**2

x = torch.ones(256, requires_grad=True, device='cuda')
y = torch.ones_like(x)

torch._dynamo.reset()
compiled_f = torch.compile(f)
out = torch.nn.functional.mse_loss(compiled_f(x),
y).backward()
```

PyTorch Code -> Prims IR -> Triton Code -> Machine Code

Example - Prims IR

```
class GraphModule(torch.nn.Module):
    def forward(self, primals_1: f32[256]):
        # File: /home/keren/code/test.py:7, code: return torch.sin(x)**2 + torch.cos(x)**2
        sin: f32[256] = torch.ops.aten.sin.default(primals_1)
        pow_1: f32[256] = torch.ops.aten.pow.Tensor_Scalar(sin, 2)
        cos: f32[256] = torch.ops.aten.cos.default(primals_1)
        pow_2: f32[256] = torch.ops.aten.pow.Tensor_Scalar(cos, 2)
        add: f32[256] = torch.ops.aten.add.Tensor(pow_1, pow_2);  pow_1 = pow_2 = None
        return [add, sin, primals_1, cos]
```

Example - Triton Code

```
@pointwise(size_hints=[256], filename=__file__, meta={'signature': {0: '*fp32', 1: '*fp32', 2: 'i32'}, 'device': 0,
'constants': {}, 'mutated_arg_names': [], 'configs': [instance_descriptor(divisible_by_16=(0, 1, 2),
equal_to_1=())]})
@triton.jit
def triton_(in_ptr0, out_ptr0, xnumel, XBLOCK : tl.constexpr):
    xnumel = 256
    xoffset = tl.program_id(0) * XBLOCK
    xindex = xoffset + tl.arange(0, XBLOCK)[:]
    xmask = xindex < xnumel
    x0 = xindex
    tmp0 = tl.load(in_ptr0 + (x0), xmask)
    tmp1 = tl.sin(tmp0)
    tmp2 = tmp1 * tmp1
    tmp3 = tl.cos(tmp0)
    tmp4 = tmp3 * tmp3
    tmp5 = tmp2 + tmp4
    tl.store(out_ptr0 + (x0 + tl.zeros([XBLOCK], tl.int32)), tmp5, xmask)
```

Benefits

- Robustness
  - Capture a single graph for most models
  - Fallback to partial graphs is needed
- Speed
  - ~1.5x faster than the eager mode

Geometric mean speedup

Compiler	torchbench	huggingface	timm_models
eager	1.00x	1.00x	1.00x
aot_eager	1.00x	1.00x	1.00x
inductor	1.59x	1.59x	1.41x
inductor_no_cudagraphs	1.30x	1.51x	1.39x

TRITON

Handwritten Low Level Code VS Automated Generation

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

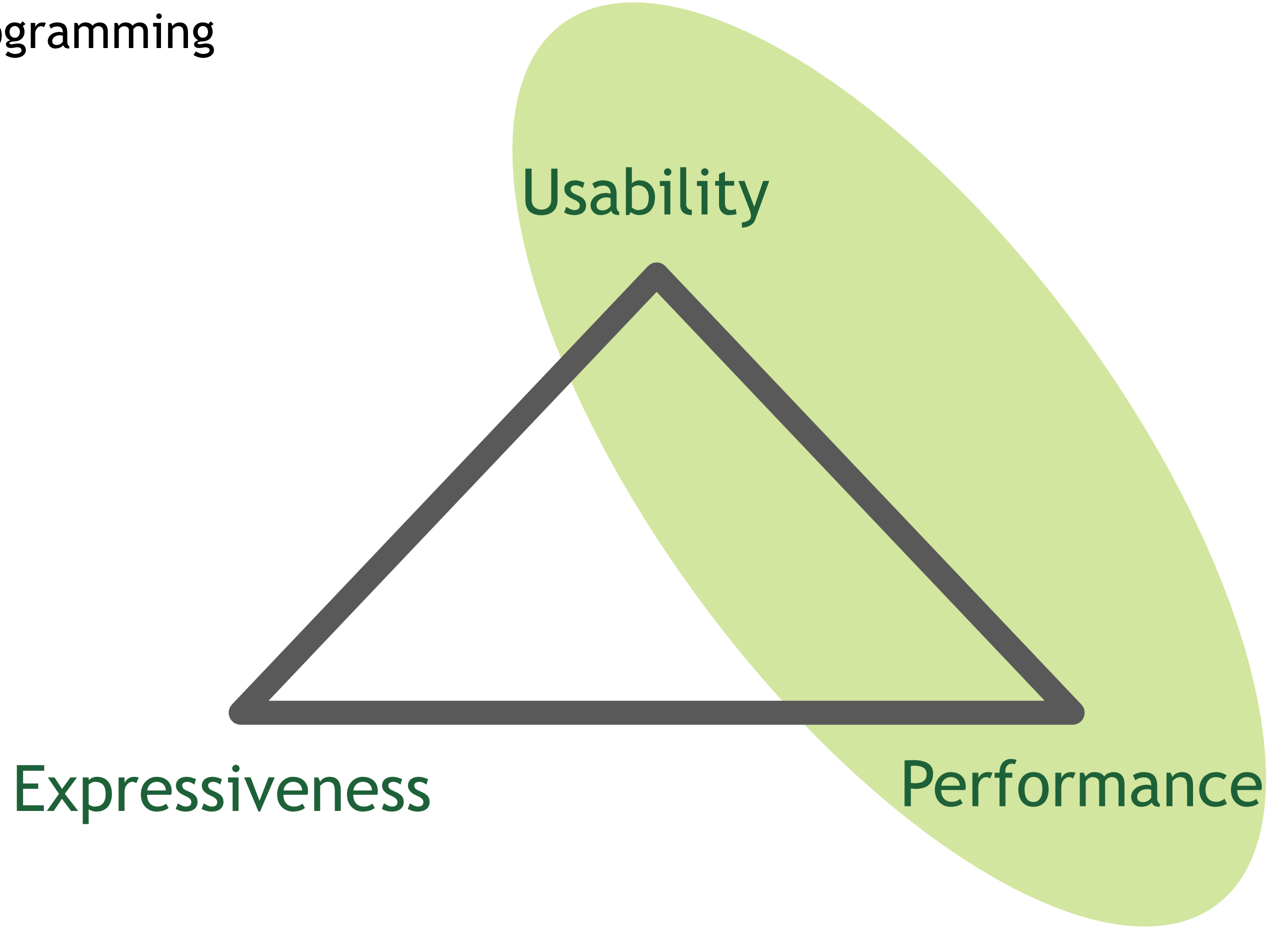
- High flexibility
  - Build upon existing operators
  - No need to recompile
  - …
- Low performance
  - Not fine-tuned for specific shapes
  - Intermediate memory movement
  - …

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

The Programming Language Design Triangle

- Triton focuses on usability and performance
  - The language features supported by triton is a subset of Python
    - No dict
    - No meta-programming
    - No slicing
    - No indexing
    - …



CUDA 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

```
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

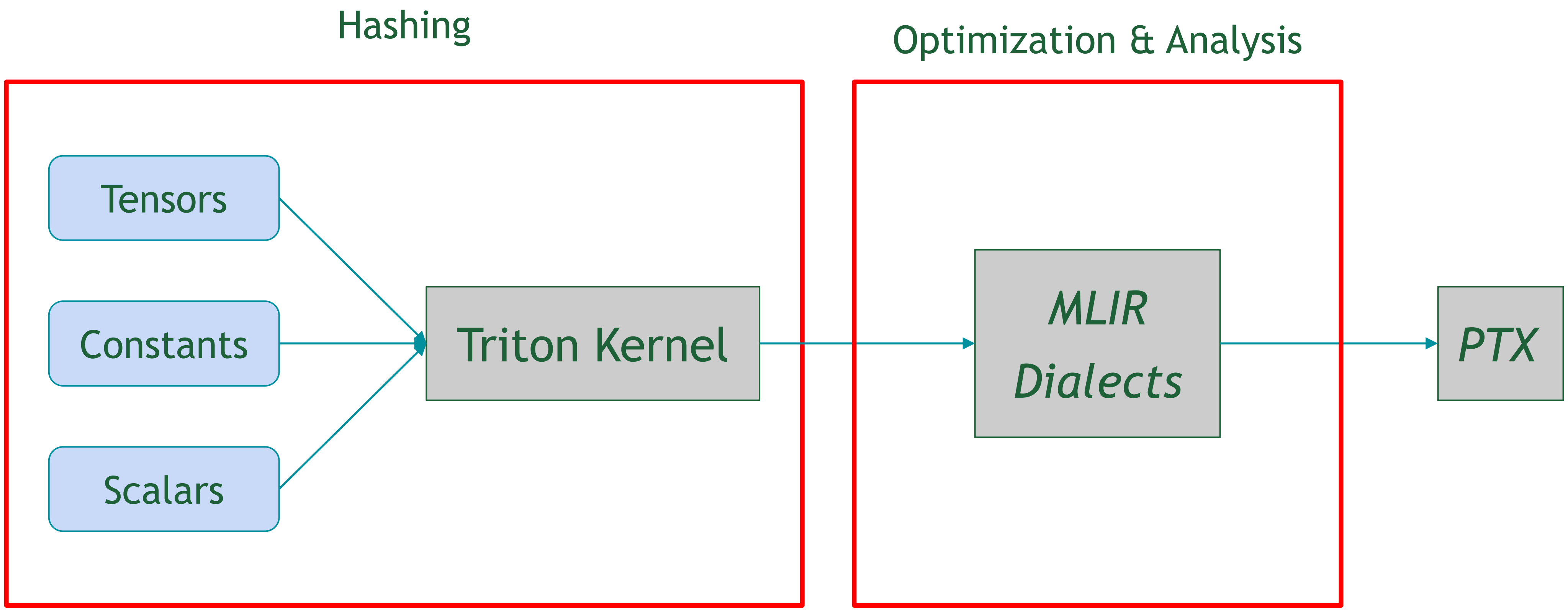
```
import triton.language as tl
import triton

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

Vector Addition (Autotune)

- ➔ 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

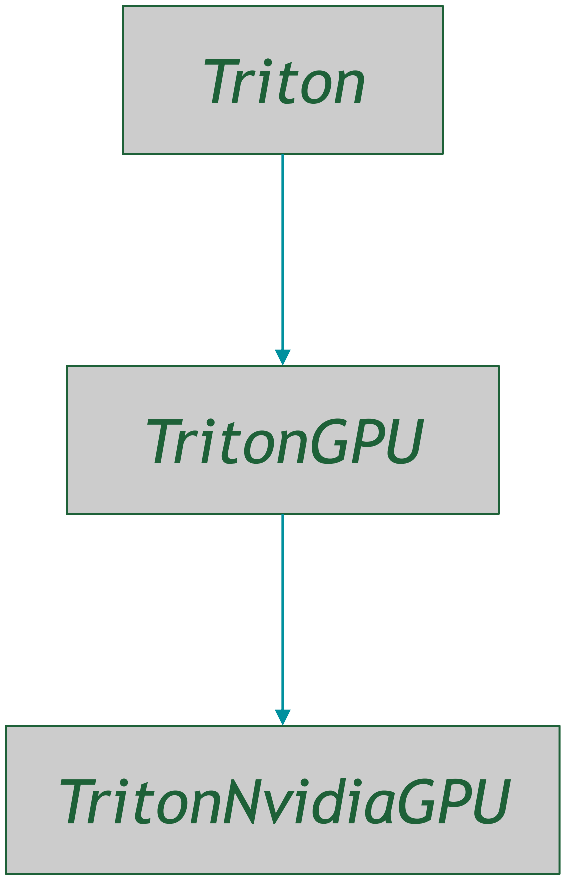
```
@triton.autotune(configs=
    [triton.Config('TILE': 128),
     triton.Config('TILE': 256)]
@triton.jit
def _add(z_ptr, x_ptr, y_ptr, N):
    # same as torch.arange
    offsets = tl.arange(0, TILE)
    offsets += tl.program_id(0)*TILE
    # create TILE pointers to X, Y, Z
    x_ptrs = x_ptr + offsets
    y_ptrs = y_ptr + offsets
    z_ptrs = z_ptr + offsets
    # load TILE 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 TILE elements of X, Y, Z
    tl.store(z_ptrs, z, mask=offset<N)

N = 1024
x = torch.randn(N, device='cuda')
y = torch.randn(N, device='cuda')
z = torch.randn(N, device='cuda')
grid = lambda args: (triton.cdiv(N, args["TILE"]), )
_add[grid](z, x, y, N)
```

Triton JIT-Compilation Workflow



Optimization Passes

- MLIR general optimizations
  - CSE, DCE, Inlining, …
- TritonGPU specific optimizations
  - Pipeline
  - Prefetch
  - Matmul accelerate
  - Coalesce
  - Remove layout
- TritonNVIDIAGPU specific optimizations
  - TMA Materialization
  - TMA Multicast
  - Async Dot
  - Warp Specialization



Layout Encoding in TritonGPU

- A specification that maps data distribution to threads to better utilize the underlying hardware
  - Suppose we have a 2x2 tensor and 8 threads
    - Layout(0, 0) = {0, 4}
    - Layout(0, 1) = {1, 5}
    - Layout(1, 0) = {2, 6}
    - Layout(1, 1) = {3, 7}
  - It means that
    - data(0, 0) is stored on thread 0 and thread 4
    - data(0, 1) is stored on thread 1 and thread 5
    - data(1, 0) is stored on thread 2 and thread 6
    - data(1, 1) is stored on thread 3 and thread 7

Blocked Layout

- The most basic layout in Triton
- Assign a default layout initially
- Optimize the layout based on global memory load/store ops
- A 2d blocked layout example
  - sizePerThread = {2, 2}
  - threadsPerWarp = {8, 4}
  - warpsPerCTA = {1, 2}
  - CTAsPerCGA = {1, 1}
  - order = {1, 0}
    - Row major

Shared Layout

- Specify how data is stored on shared memory
  - Use 2D-swizzling or padding to avoid bank conflicts
- Triton doesn’t manage shared memory explicitly
  - Shared memory is only used when involving data exchange across threads
    - Convert from one layout to another

Dot Operand Layout

- mma.m16n8k16
  - A [m,k] X B [k,n] + C [m, n] = D [m, n]

R\C	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	T0:{a0,a1}		T1:{a0,a1}		T2:{a0,a1}		T3:{a0,a1}		T0:{a4,a5}		T1:{a4,a5}		T2:{a4,a5}		T3:{a4,a5}	
1	T4:{a0,a1}		T5:{a0,a1}		T6:{a0,a1}		T7:{a0,a1}		T4:{a4,a5}		T5:{a4,a5}		T6:{a4,a5}		T7:{a4,a5}	
2																
..																
7	T28:{a0,a1}		T29:{a0,a1}		T30:{a0,a1}		T31:{a0,a1}		T28:{a4,a5}		T29:{a4,a5}		T30:{a4,a5}		T31:{a4,a5}	
8	T0:{a2,a3}		T1:{a2,a3}		T2:{a2,a3}		T3:{a2,a3}		T0:{a6,a7}		T1:{a6,a7}		T2:{a6,a7}		T3:{a6,a7}	
9	T4:{a2,a3}		T5:{a2,a3}		T6:{a2,a3}		T7:{a2,a3}		T4:{a6,a7}		T5:{a6,a7}		T6:{a6,a7}		T7:{a6,a7}	
10																
..																
15	T28:{a2,a3}		T29:{a2,a3}		T30:{a2,a3}		T31:{a2,a3}		T28:{a6,a7}		T29:{a6,a7}		T30:{a6,a7}		T31:{a6,a7}	

%laneid:{fragments}

A: fp16

Row\Col	0	1	2	..	7
0 1	$T0:\begin{Bmatrix}b0\\b1\end{Bmatrix}$	$T4:\begin{Bmatrix}b0\\b1\end{Bmatrix}$			$T28:\begin{Bmatrix}b0\\b1\end{Bmatrix}$
6 7	$T3:\begin{Bmatrix}b0\\b1\end{Bmatrix}$	$T7:\begin{Bmatrix}b0\\b1\end{Bmatrix}$			$T31:\begin{Bmatrix}b0\\b1\end{Bmatrix}$
8 9	$T0:\begin{Bmatrix}b2\\b3\end{Bmatrix}$	$T4:\begin{Bmatrix}b2\\b3\end{Bmatrix}$			$T28:\begin{Bmatrix}b2\\b3\end{Bmatrix}$
14 15	$T3:\begin{Bmatrix}b2\\b3\end{Bmatrix}$	$T7:\begin{Bmatrix}b2\\b3\end{Bmatrix}$			$T31:\begin{Bmatrix}b2\\b3\end{Bmatrix}$

%laneid:{fragments}

B: fp16

MMA Layout

- mma.m16n8k16
  - A [m,k] X B [k,n] + C [m, n] = D [m, n]

R\C	0 1	2 3	4 5	6 7	8 9	10 11	12 13	14 15
0	T0:{a0,a1}	T1:{a0,a1}	T2:{a0,a1}	T3:{a0,a1}	T0:{a4,a5}	T1:{a4,a5}	T2:{a4,a5}	T3:{a4,a5}
1	T4:{a0,a1}	T5:{a0,a1}	T6:{a0,a1}	T7:{a0,a1}	T4:{a4,a5}	T5:{a4,a5}	T6:{a4,a5}	T7:{a4,a5}
2								
..								
7	T28:{a0,a1}	T29:{a0,a1}	T30:{a0,a1}	T31:{a0,a1}	T28:{a4,a5}	T29:{a4,a5}	T30:{a4,a5}	T31:{a4,a5}
8	T0:{a2,a3}	T1:{a2,a3}	T2:{a2,a3}	T3:{a2,a3}	T0:{a6,a7}	T1:{a6,a7}	T2:{a6,a7}	T3:{a6,a7}
9	T4:{a2,a3}	T5:{a2,a3}	T6:{a2,a3}	T7:{a2,a3}	T4:{a6,a7}	T5:{a6,a7}	T6:{a6,a7}	T7:{a6,a7}
10								
..								
15	T28:{a2,a3}	T29:{a2,a3}	T30:{a2,a3}	T31:{a2,a3}	T28:{a6,a7}	T29:{a6,a7}	T30:{a6,a7}	T31:{a6,a7}

%laneid:{fragments}

A: fp16

Row\Col	0 1	2 3	4 5	6 7
0	T0: {c0, c1}	T1: {c0, c1}	T2: {c0, c1}	T3: {c0, c1}
1	T4: {c0, c1}	T5: {c0, c1}	T6: {c0, c1}	T7: {c0, c1}
2				
..				
7	T28: {c0, c1}	T29: {c0, c1}	T30: {c0, c1}	T31: {c0, c1}
8	T0:{c2, c3}	T1:{c2, c3}	T2:{c2, c3}	T3:{c2, c3}
9	T4:{c2, c3}	T5:{c2, c3}	T6:{c2, c3}	T7:{c2, c3}
10				
..				
15	T28:{c2, c3}	T29:{c2, c3}	T30:{c2, c3}	T31:{c2, c3}

%laneid:{fragments}

C or D: fp16

Analysis Passes

- Shared memory
  - Alias
  - Liveness
  - Barrier
- Pointer alignment
  - Axisinfo
- Call graph
  - “noinline” functions

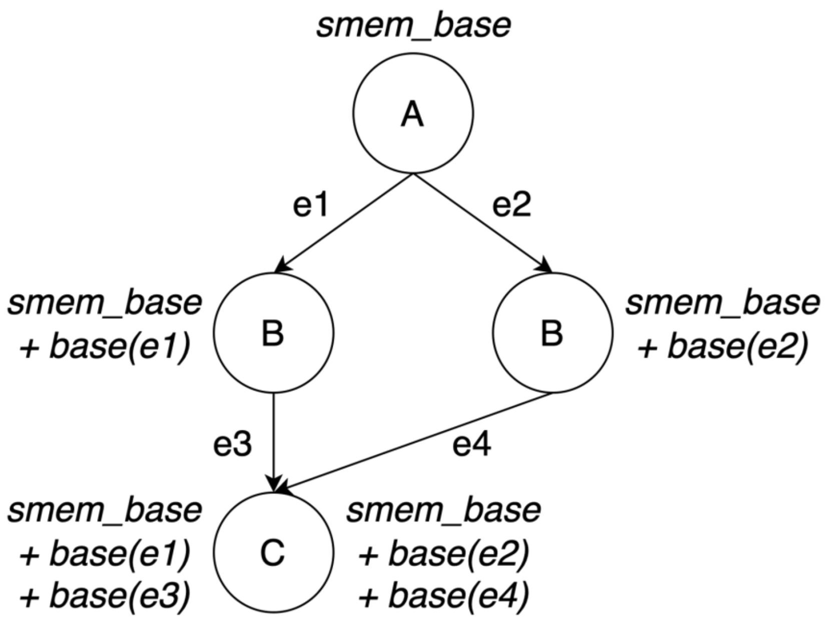


Figure 2

Ecosystem

deepspeed
tinygrad
kernl.ai

Runtime
Debugger
Profiler

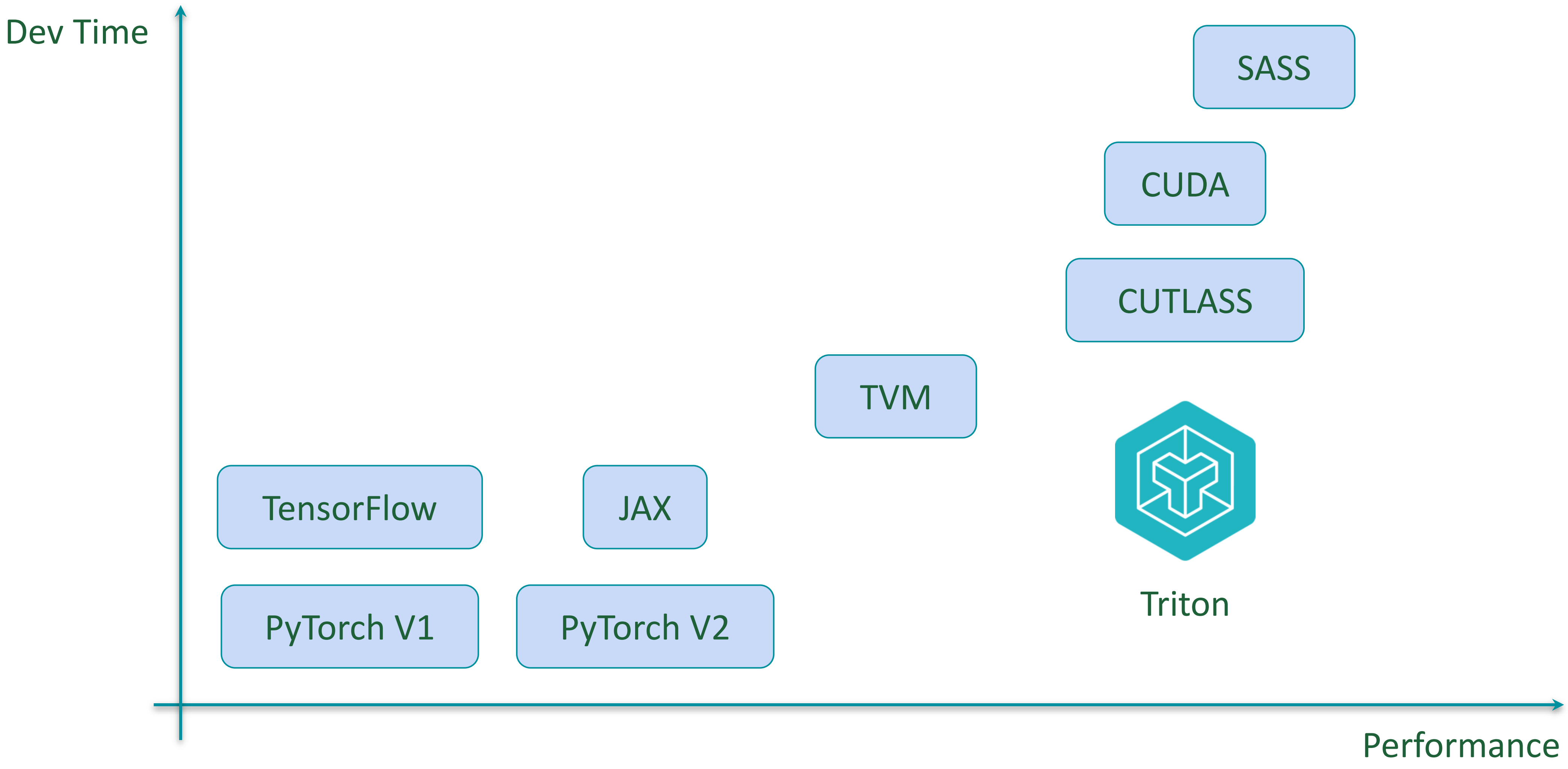
PyTorch

Language

OpenXLA IREE

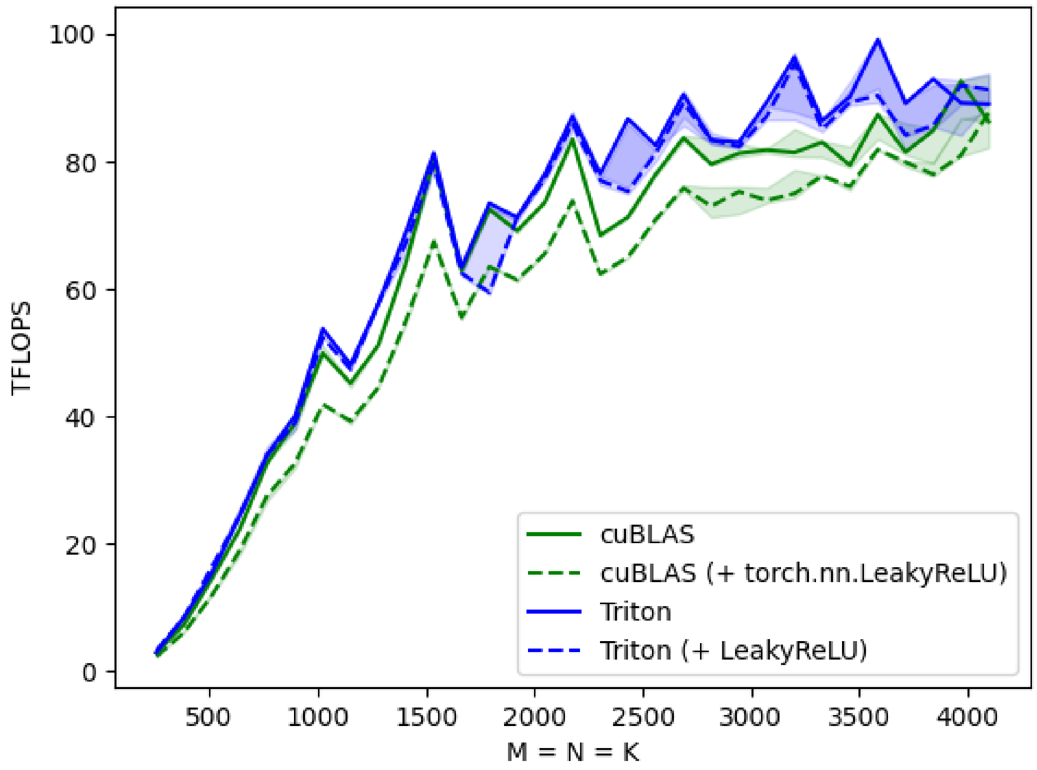
Backends

Dev Time VS Performance

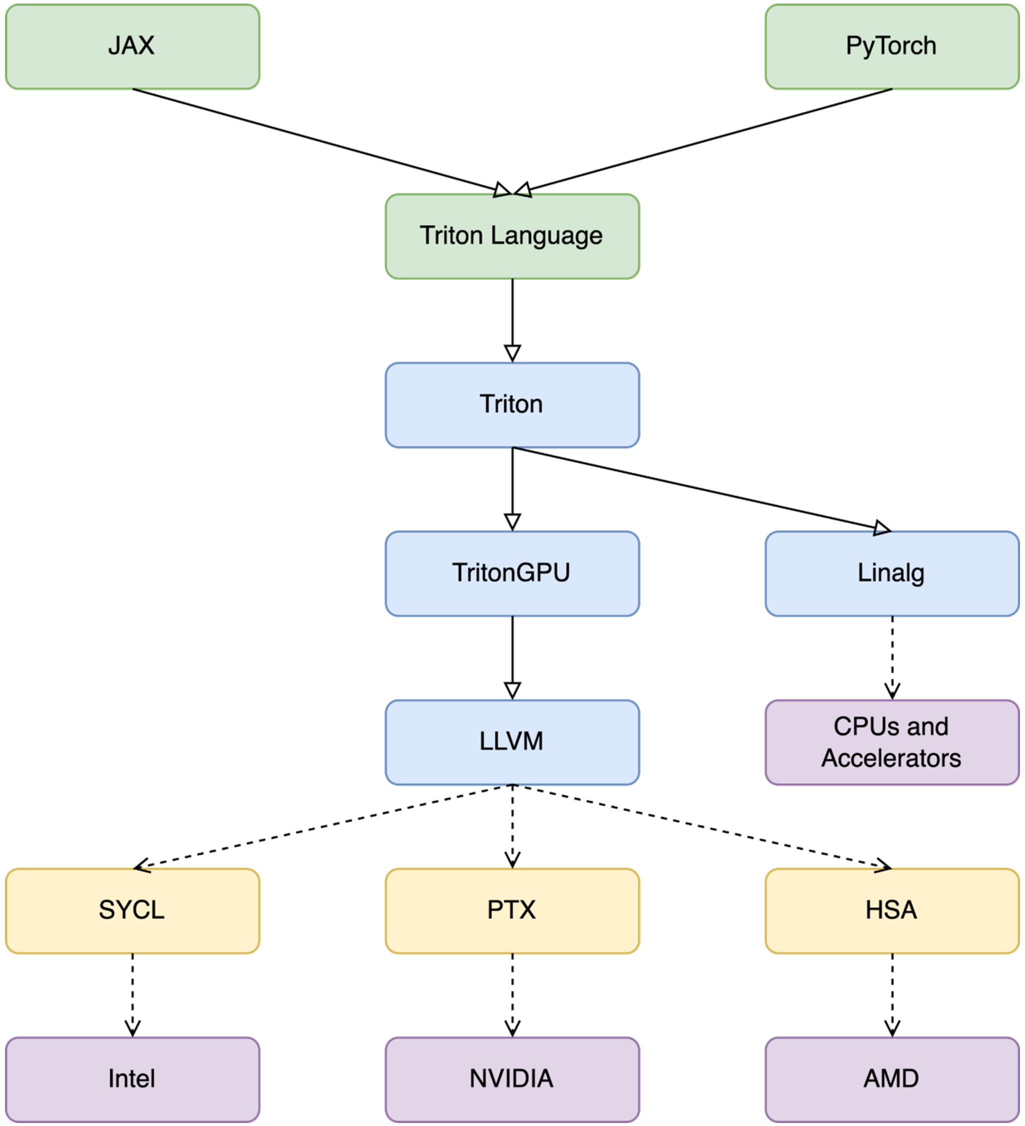


Triton Performance

- 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/cublas



Backend Status



Q & A