

# Decoupled Triton: A Block-Level Decoupled Language for Writing and Exploring Efficient Machine-Learning Kernels

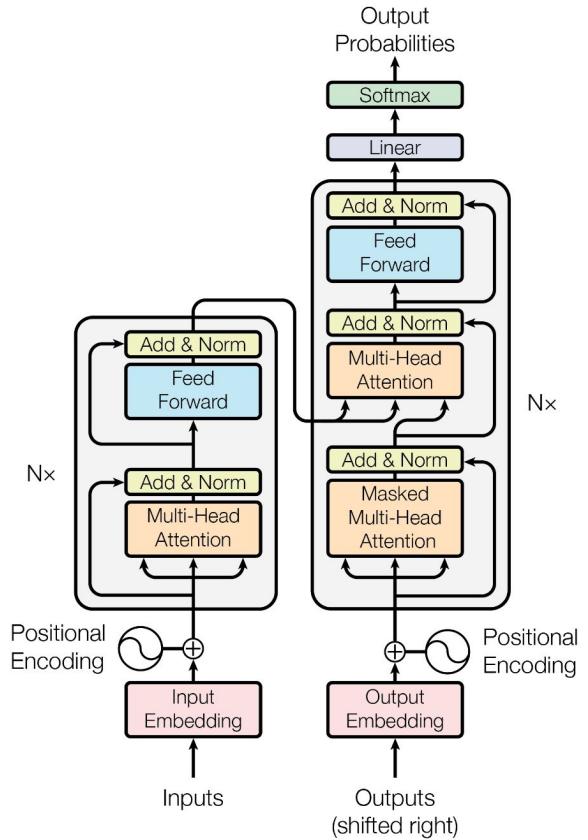
Quinn Pham

# Machine-Learning (ML) Kernels

# Machine-learning kernels

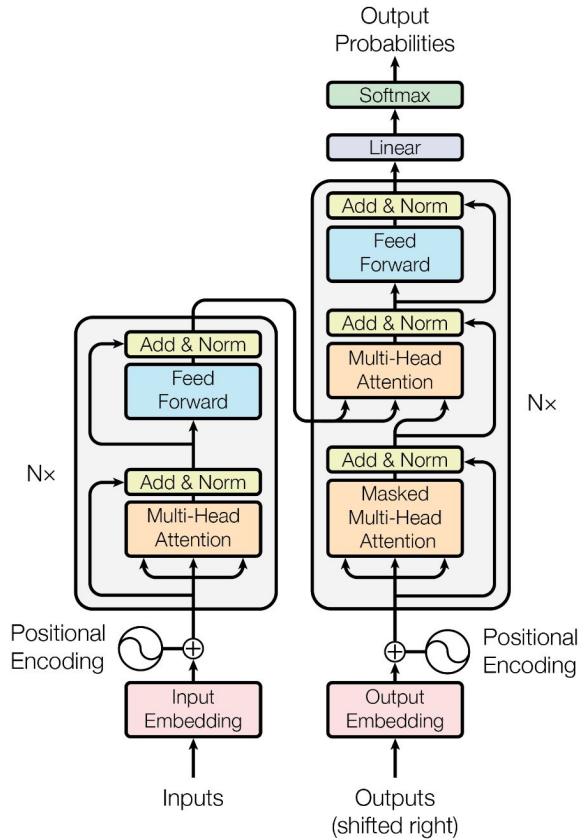
# Machine-learning kernels

- Specialized functions for tensor operations.



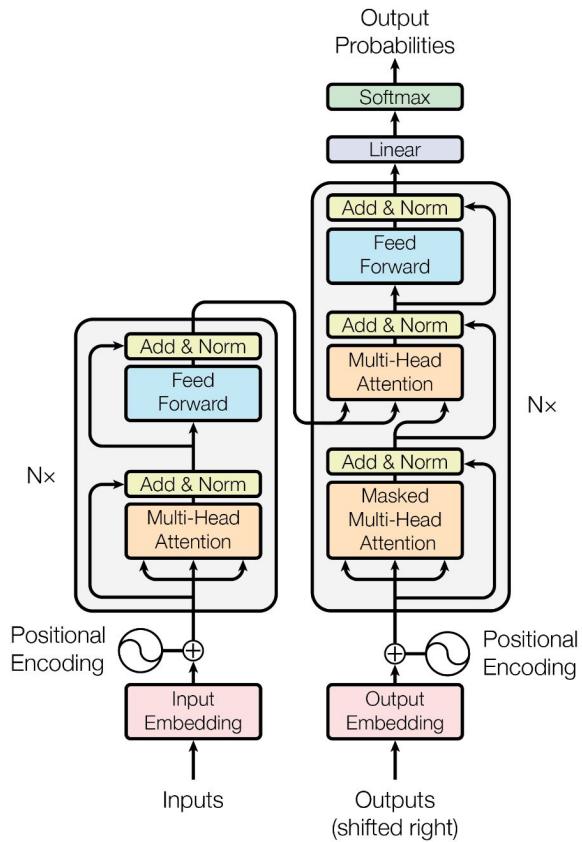
# Machine-learning kernels

- Specialized functions for tensor operations.
  - Matrix multiplication



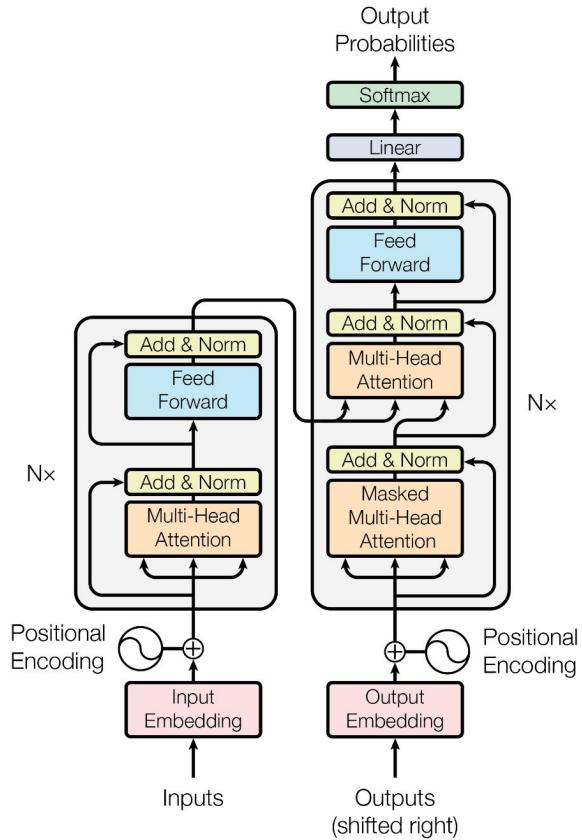
# Machine-learning kernels

- Specialized functions for tensor operations.
  - Matrix multiplication
  - Softmax



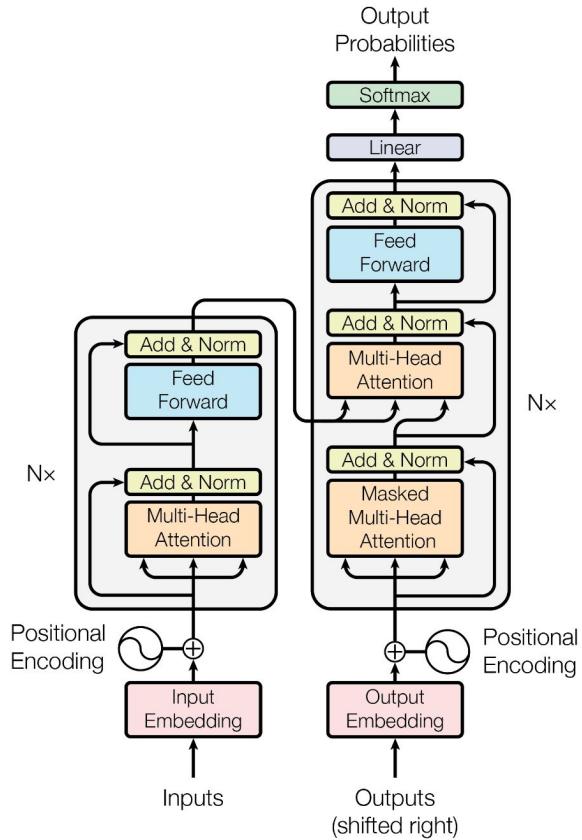
# Machine-learning kernels

- Specialized functions for tensor operations.
  - Matrix multiplication
  - Softmax
  - Attention



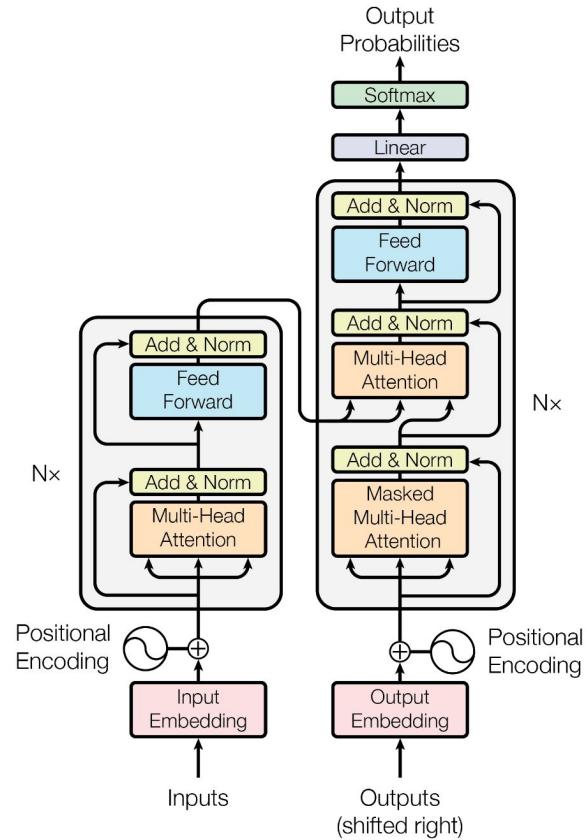
# Machine-learning kernels

- Specialized functions for tensor operations.
  - Matrix multiplication
  - Softmax
  - Attention
  - LayerNorm



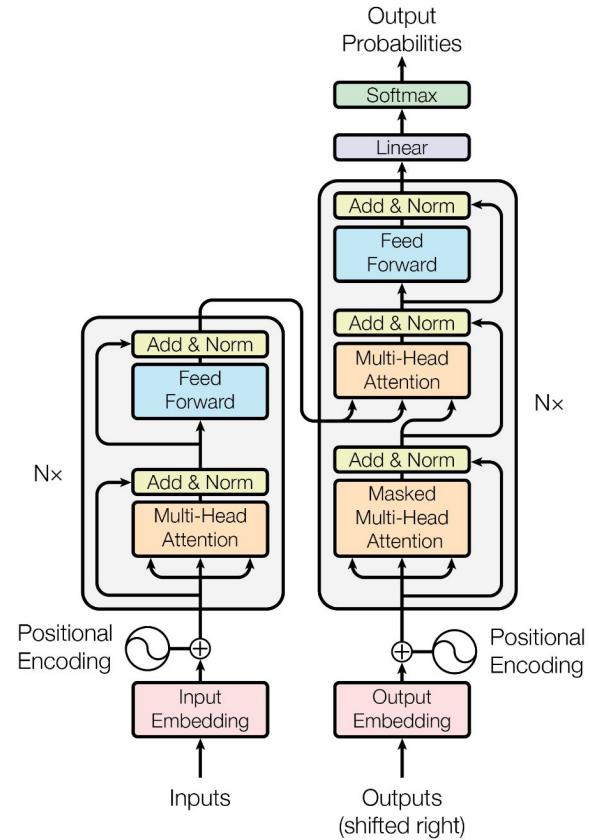
# Machine-learning kernels

- Specialized functions for tensor operations.
  - Matrix multiplication
  - Softmax
  - Attention
  - LayerNorm
  - Fused operations (e.g. matmul + sigmoid)



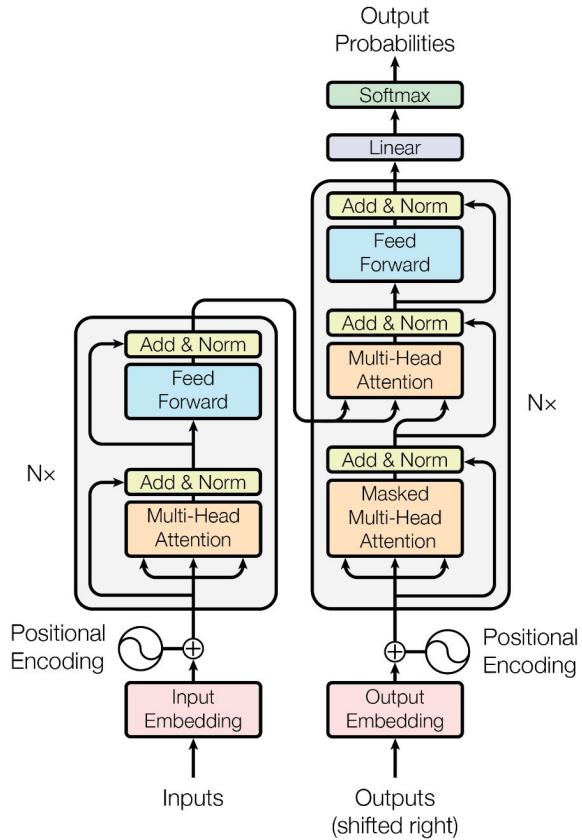
# Machine-learning kernels

- Specialized functions for tensor operations.
  - Matrix multiplication
  - Softmax
  - Attention
  - LayerNorm
  - Fused operations (e.g. matmul + sigmoid)
  - ...



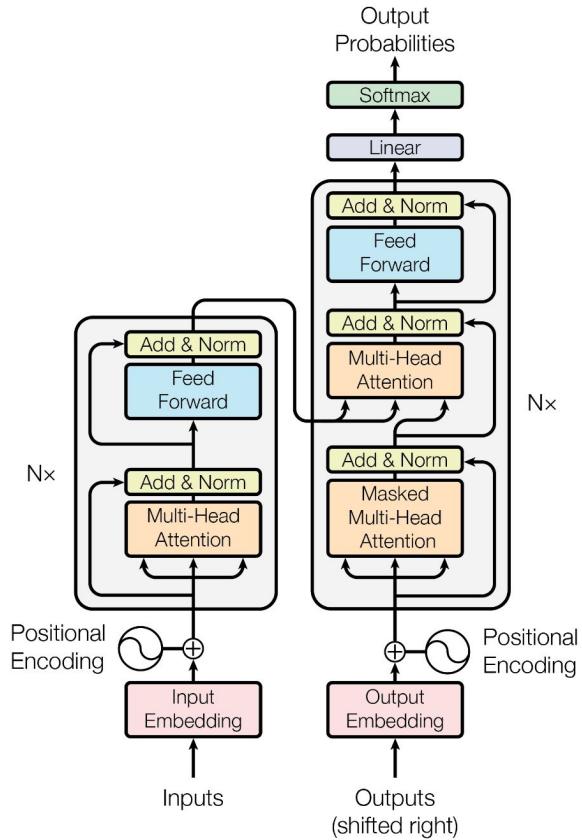
# Machine-learning kernels

- Specialized functions for tensor operations.
  - Matrix multiplication
  - Softmax
  - Attention
  - LayerNorm
  - Fused operations (e.g. matmul + sigmoid)
  - ...
- Building blocks of deep-learning models.



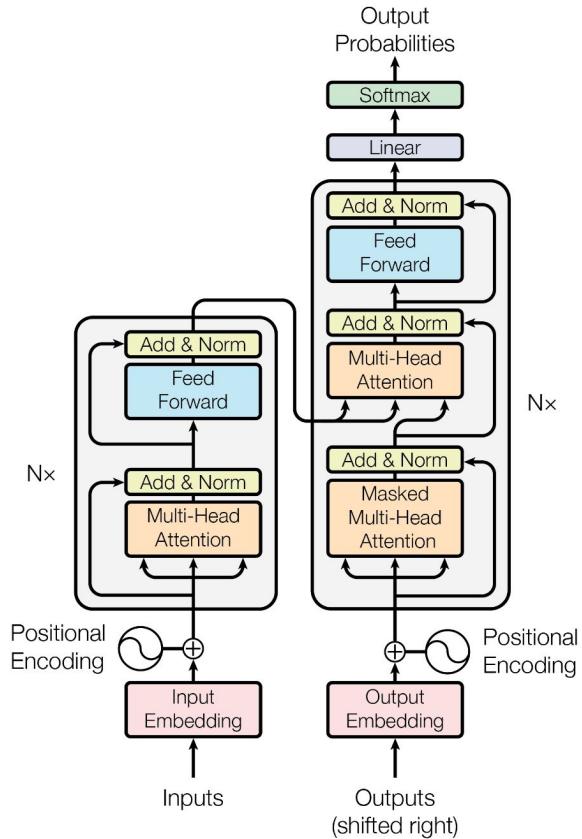
# Machine-learning kernels

- Specialized functions for tensor operations.
  - Matrix multiplication
  - Softmax
  - Attention
  - LayerNorm
  - Fused operations (e.g. matmul + sigmoid)
  - ...
- Building blocks of deep-learning models.
  - Efficient models require efficient kernels



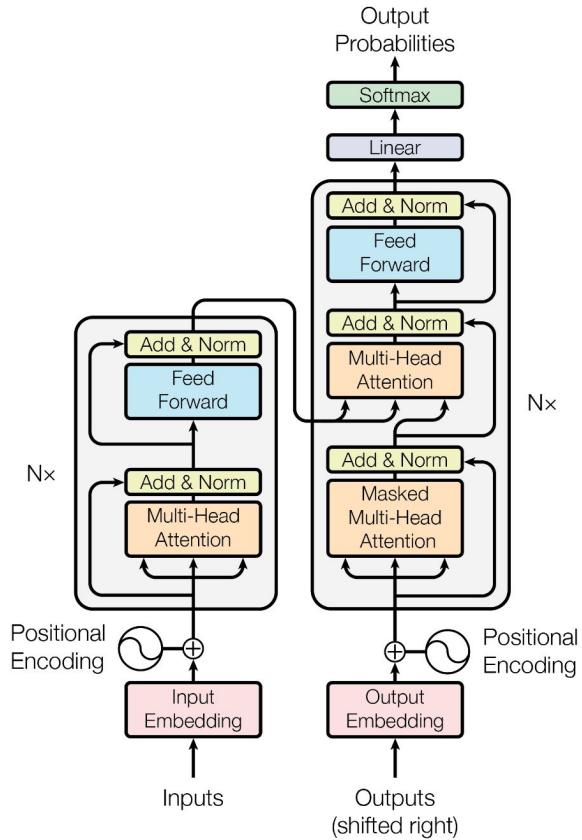
# Machine-learning kernels

- Specialized functions for tensor operations.
  - Matrix multiplication
  - Softmax
  - Attention
  - LayerNorm
  - Fused operations (e.g. matmul + sigmoid)
  - ...
- Building blocks of deep-learning models.
  - Efficient models require efficient kernels
- Executed on parallel hardware (GPUs, TPUs).



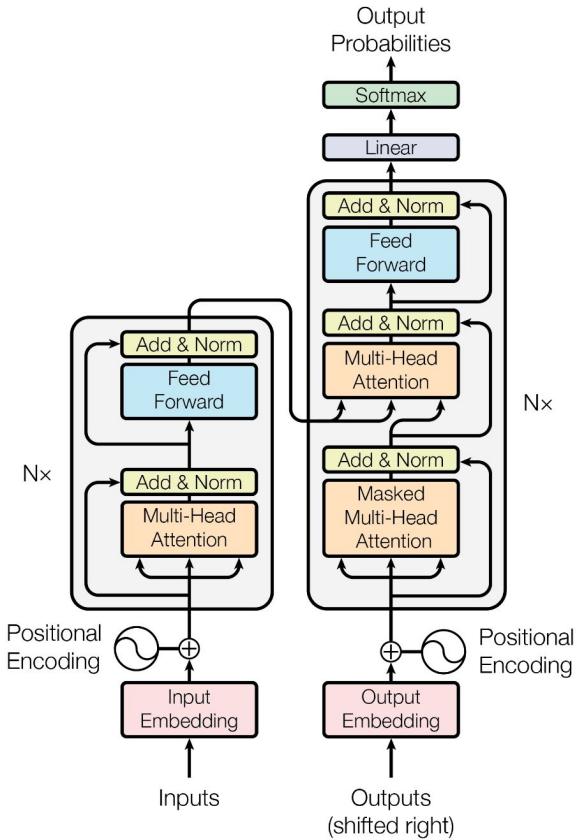
# Machine-learning kernels

- Specialized functions for tensor operations.
  - Matrix multiplication
  - Softmax
  - Attention
  - LayerNorm
  - Fused operations (e.g. matmul + sigmoid)
  - ...
- Building blocks of deep-learning models.
  - Efficient models require efficient kernels
- Executed on parallel hardware (GPUs, TPUs).
- High-performance kernel libraries (cuBLAS, cuDNN).

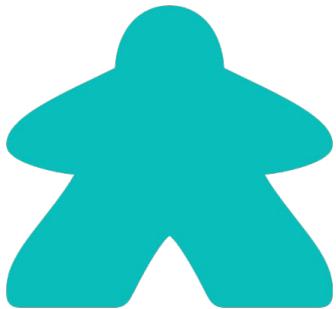


# Machine-learning kernels

- Specialized functions for tensor operations.
  - Matrix multiplication
  - Softmax
  - Attention
  - LayerNorm
  - Fused operations (e.g. matmul + sigmoid)
  - ...
- Building blocks of deep-learning models.
  - Efficient models require efficient kernels
- Executed on parallel hardware (GPUs, TPUs).
- High-performance kernel libraries (cuBLAS, cuDNN).
  - Provide kernels for common tensor operations

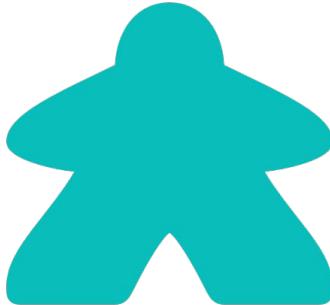


Building the motivation for Decoupled Triton.

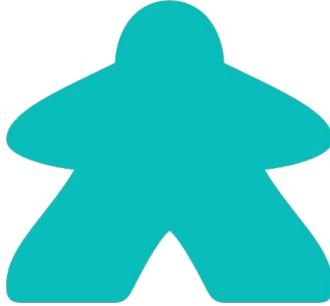


Deep Learning  
Researcher

I created an amazing  
Deep Learning model  
that demonstrates AGI.



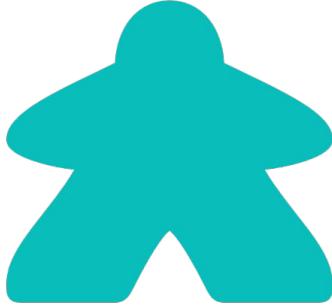
Deep Learning  
Researcher



Deep Learning  
Researcher

I created an amazing  
Deep Learning model  
that demonstrates AGI.

The model needs to  
compute this new  
tensor operation:  
 $C = \text{op}(A, B)$

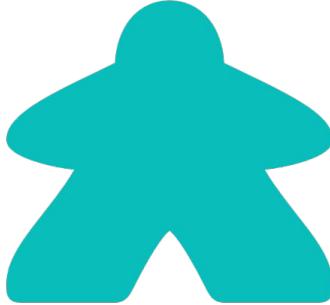


Deep Learning  
Researcher

I created an amazing  
Deep Learning model  
that demonstrates AGI.

op could be:

The model needs to  
compute this new  
tensor operation:  
 $C = \text{op}(A, B)$



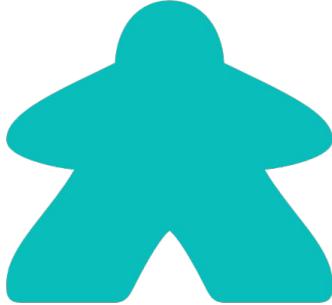
Deep Learning  
Researcher

I created an amazing  
Deep Learning model  
that demonstrates AGI.

The model needs to  
compute this new  
tensor operation:  
 $C = \text{op}(A, B)$

$\text{op}$  could be:

- an entirely new  
operation



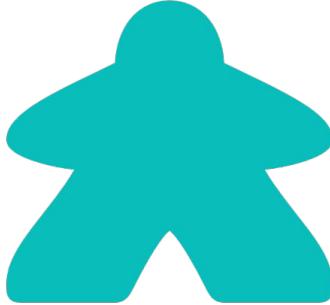
Deep Learning  
Researcher

I created an amazing  
Deep Learning model  
that demonstrates AGI.

The model needs to  
compute this new  
tensor operation:  
 $C = \text{op}(A, B)$

$\text{op}$  could be:

- an entirely new operation
- a new fusion of existing operations



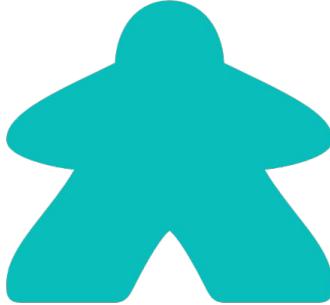
Deep Learning  
Researcher

I created an amazing  
Deep Learning model  
that demonstrates AGI.

The model needs to  
compute this new  
tensor operation:  
 $C = \text{op}(A, B)$

$\text{op}$  could be:

- an entirely new operation
- a new fusion of existing operations
- an existing operation performed with unusual dimension sizes



Deep Learning  
Researcher

I created an amazing  
Deep Learning model  
that demonstrates AGI.

The model needs to  
compute this new  
tensor operation:  
 $C = \text{op}(A, B)$

$\text{op}$  could be:

- an entirely new operation
- a new fusion of existing operations
- an existing operation performed with unusual dimension sizes

An efficient  $\text{op}$  kernel will not be provided by  
high-performance kernel libraries!

# Writing efficient machine-learning kernels



# Writing efficient machine-learning kernels

- Low-abstraction parallel-programming languages  
(CUDA, OpenCL, HIP).



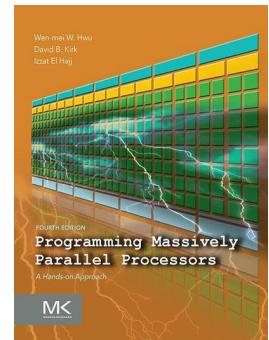
# Writing efficient machine-learning kernels

- **Low-abstraction** parallel-programming languages  
(CUDA, OpenCL, HIP).



# Writing efficient machine-learning kernels

- **Low-abstraction** parallel-programming languages (CUDA, OpenCL, HIP).
  - Require strong understanding of parallel architectures and parallel programming patterns



# Triton

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

Philippe Tillet  
Harvard University  
USA

H. T. Kung  
Harvard University  
USA

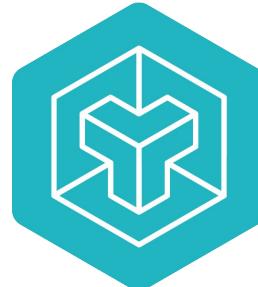
David Cox  
Harvard University, IBM  
USA

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

Philippe Tillet  
Harvard University  
USA

H. T. Kung  
Harvard University  
USA

David Cox  
Harvard University, IBM  
USA



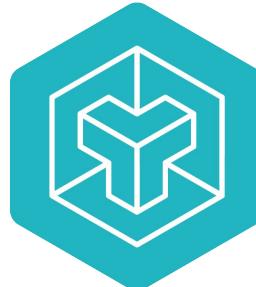
# Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations

Philippe Tillet  
Harvard University  
USA

H. T. Kung  
Harvard University  
USA

David Cox  
Harvard University, IBM  
USA

Maintained by OpenAI



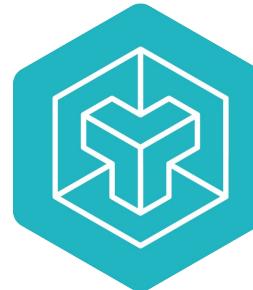
# Triton

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

Philippe Tillet  
Harvard University  
USA

H. T. Kung  
Harvard University  
USA

David Cox  
Harvard University, IBM  
USA



# Triton

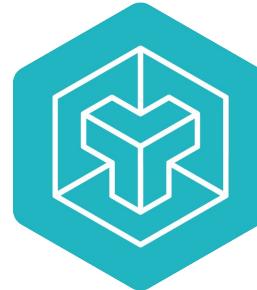
- High-abstraction parallel programming language and compiler.

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

Philippe Tillet  
Harvard University  
USA

H. T. Kung  
Harvard University  
USA

David Cox  
Harvard University, IBM  
USA



# Triton

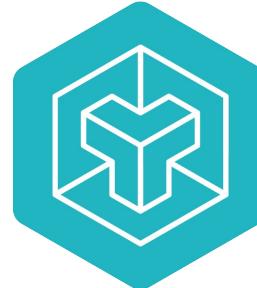
- High-abstraction parallel programming language and compiler.
  - Primarily used to define ML kernels

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

Philippe Tillet  
Harvard University  
USA

H. T. Kung  
Harvard University  
USA

David Cox  
Harvard University, IBM  
USA



# Triton

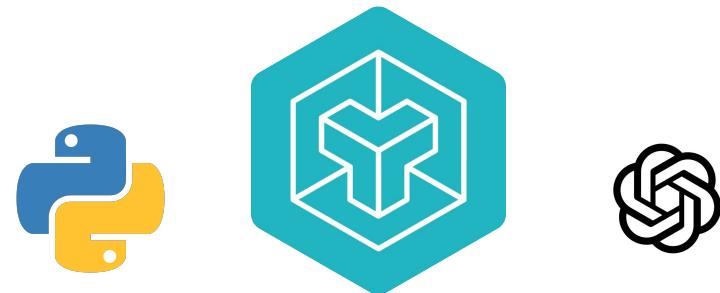
- High-abstraction parallel programming language and compiler.
  - Primarily used to define ML kernels
  - Python-based

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

Philippe Tillet  
Harvard University  
USA

H. T. Kung  
Harvard University  
USA

David Cox  
Harvard University, IBM  
USA



# Triton

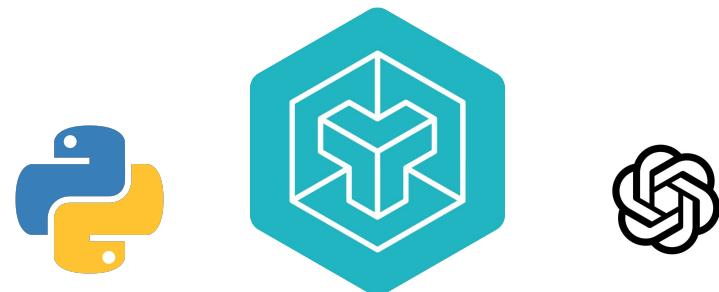
- High-abstraction parallel programming language and compiler.
  - Primarily used to define ML kernels
  - Python-based
  - Supports NVIDIA and AMD GPUs

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

Philippe Tillet  
Harvard University  
USA

H. T. Kung  
Harvard University  
USA

David Cox  
Harvard University, IBM  
USA



# Triton

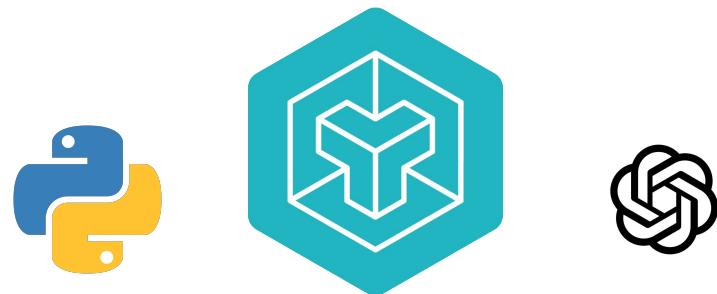
- High-abstraction parallel programming language and compiler.
  - Primarily used to define ML kernels
  - Python-based
  - Supports NVIDIA and AMD GPUs
- Introduced a block-level programming model.

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

Philippe Tillet  
Harvard University  
USA

H. T. Kung  
Harvard University  
USA

David Cox  
Harvard University, IBM  
USA



# Triton

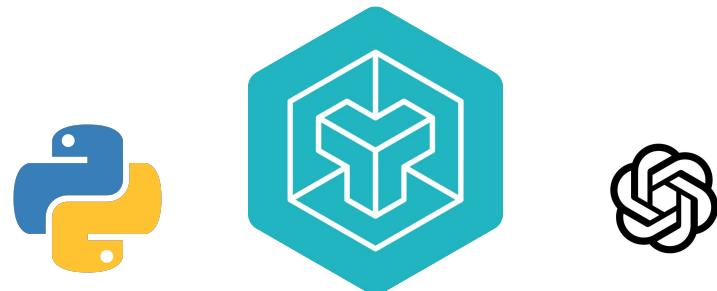
- High-abstraction parallel programming language and compiler.
  - Primarily used to define ML kernels
  - Python-based
  - Supports NVIDIA and AMD GPUs
- Introduced a block-level programming model.
  - Abstracts away many low-level details.

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

Philippe Tillet  
Harvard University  
USA

H. T. Kung  
Harvard University  
USA

David Cox  
Harvard University, IBM  
USA



# Triton

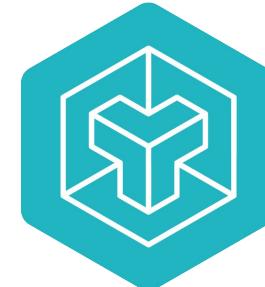
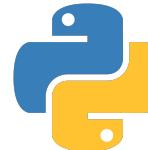
- High-abstraction parallel programming language and compiler.
  - Primarily used to define ML kernels
  - Python-based
  - Supports NVIDIA and AMD GPUs
- Introduced a block-level programming model.
  - Abstracts away many low-level details.
- Default GPU kernel backend for PyTorch 2.0 (TorchInductor).

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

Philippe Tillet  
Harvard University  
USA

H. T. Kung  
Harvard University  
USA

David Cox  
Harvard University, IBM  
USA



# Triton

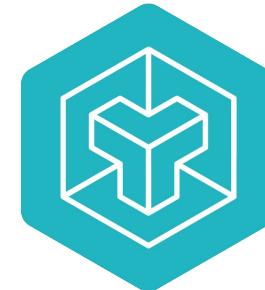
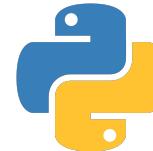
- High-abstraction parallel programming language and compiler.
  - Primarily used to define ML kernels
  - Python-based
  - Supports NVIDIA and AMD GPUs
- Introduced a **block-level programming model**.
  - Abstracts away many low-level details.
- Default GPU kernel backend for PyTorch 2.0 (TorchInductor).

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

Philippe Tillet  
Harvard University  
USA

H. T. Kung  
Harvard University  
USA

David Cox  
Harvard University, IBM  
USA



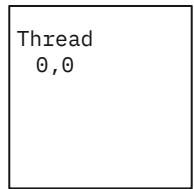
# Matmul without block-level model (CUDA)

# Matmul without block-level model (CUDA)

$$\textcolor{red}{A} @ \textcolor{blue}{B} = \textcolor{violet}{C}$$

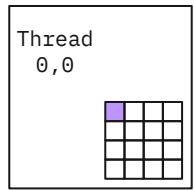
# Matmul without block-level model (CUDA)

A @ B = C



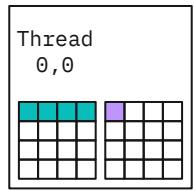
# Matmul without block-level model (CUDA)

A @ B = C



# Matmul without block-level model (CUDA)

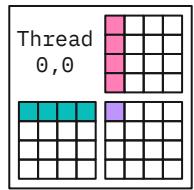
A @ B = C



Thread  
0,0

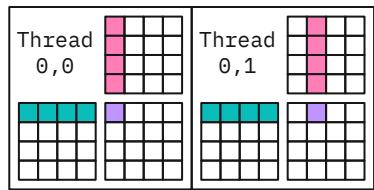
# Matmul without block-level model (CUDA)

A @ B = C



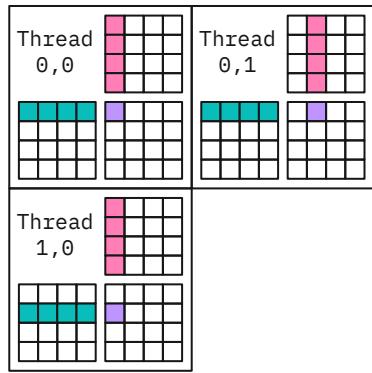
# Matmul without block-level model (CUDA)

A @ B = C



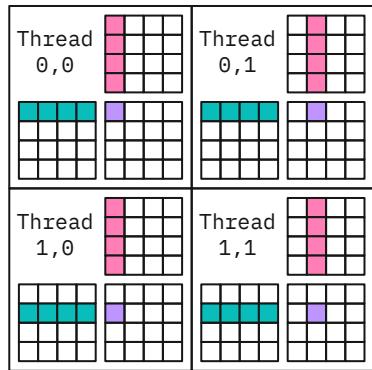
# Matmul without block-level model (CUDA)

A @ B = C



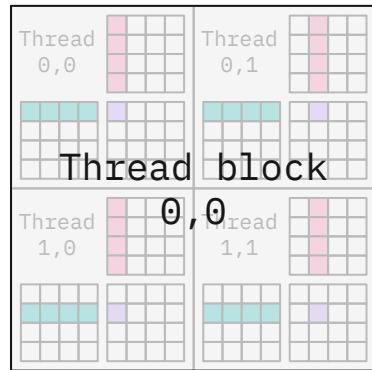
# Matmul without block-level model (CUDA)

A @ B = C



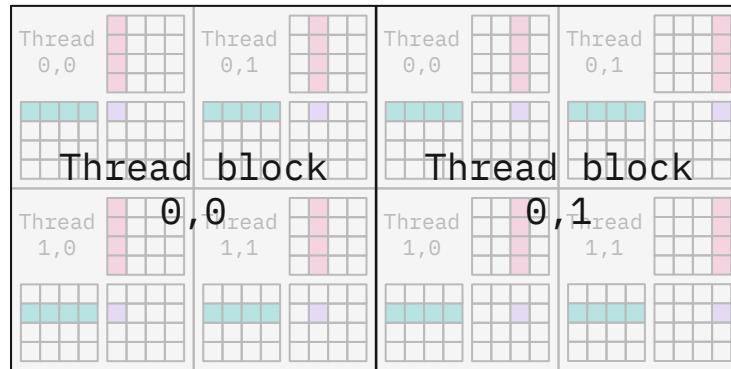
# Matmul without block-level model (CUDA)

A @ B = C



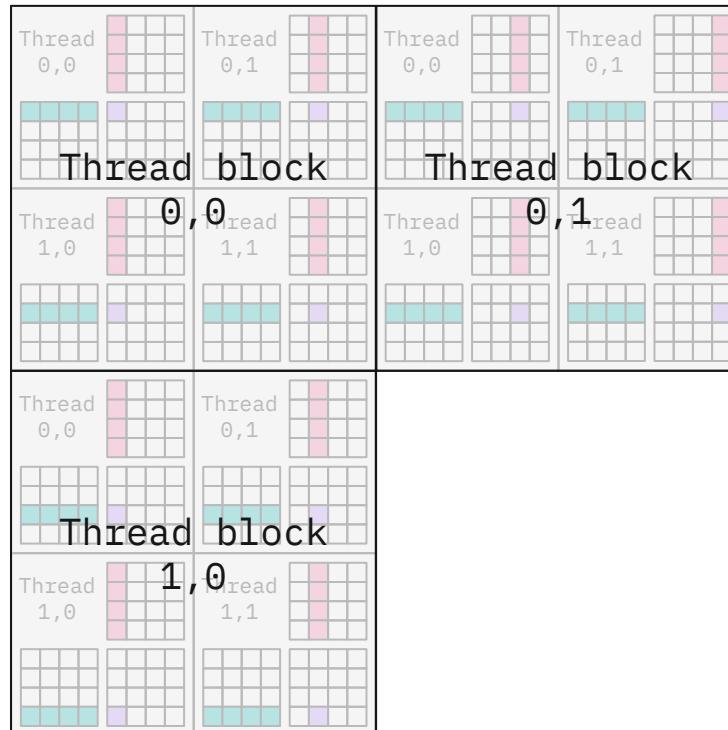
# Matmul without block-level model (CUDA)

$$\textcolor{teal}{A} @ \textcolor{pink}{B} = \textcolor{violet}{C}$$



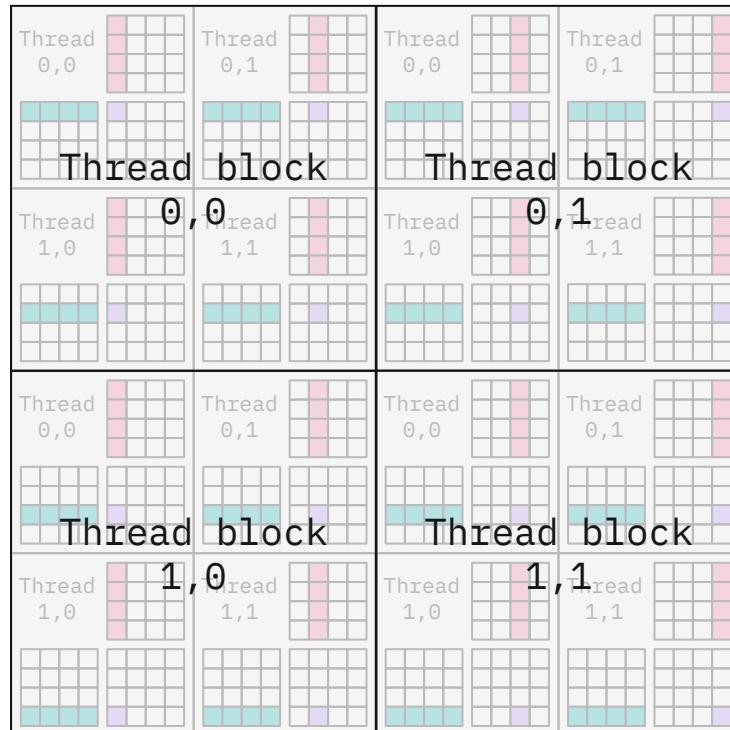
# Matmul without block-level model (CUDA)

$$\mathbf{A} @ \mathbf{B} = \mathbf{C}$$



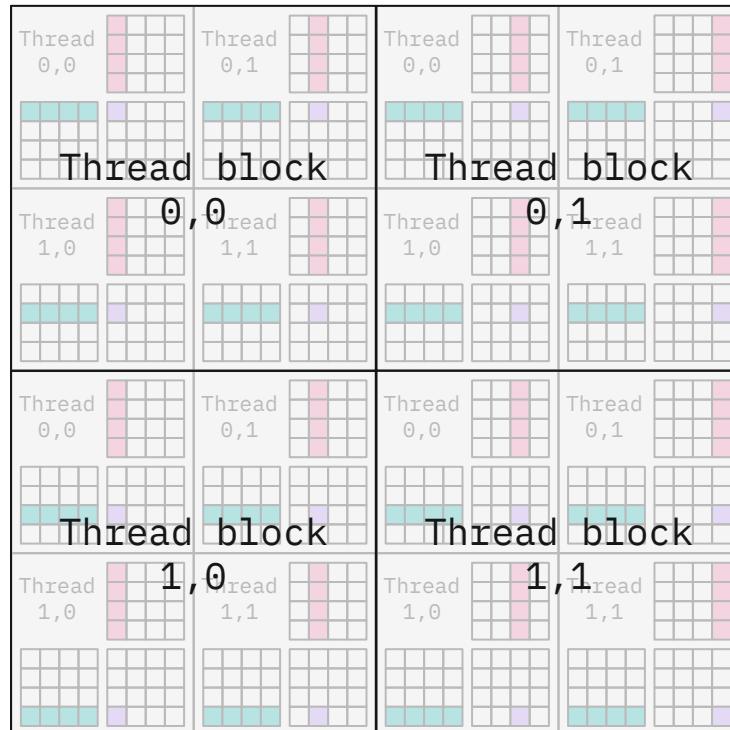
# Matmul without block-level model (CUDA)

$$\mathbf{A} @ \mathbf{B} = \mathbf{C}$$



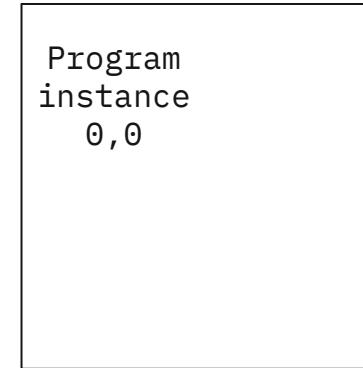
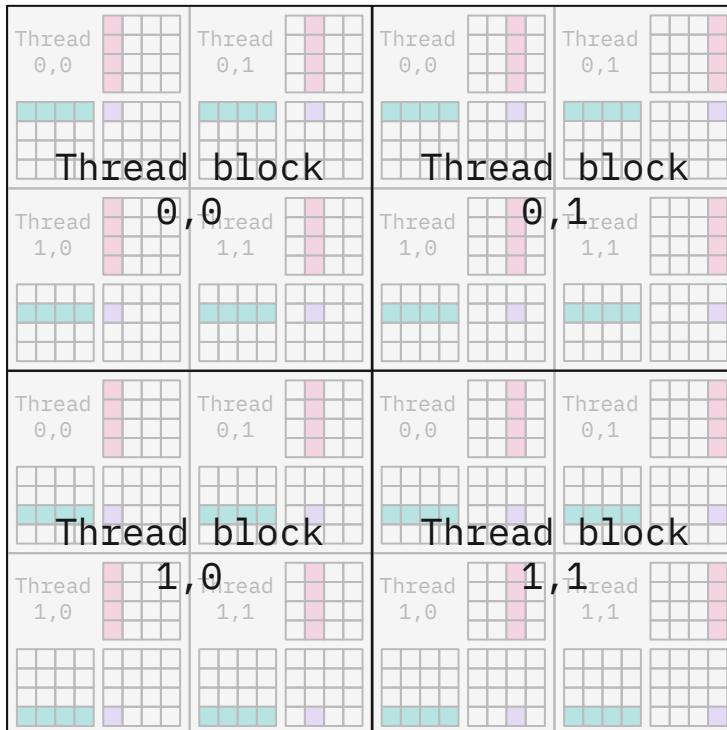
# Matmul with block-level model (Triton)

$$\mathbf{A} @ \mathbf{B} = \mathbf{C}$$



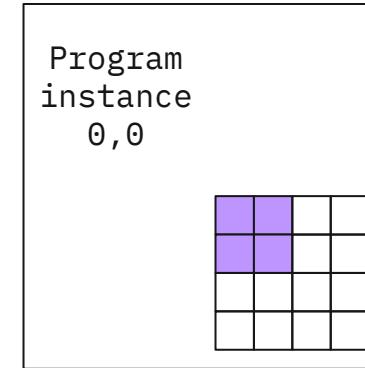
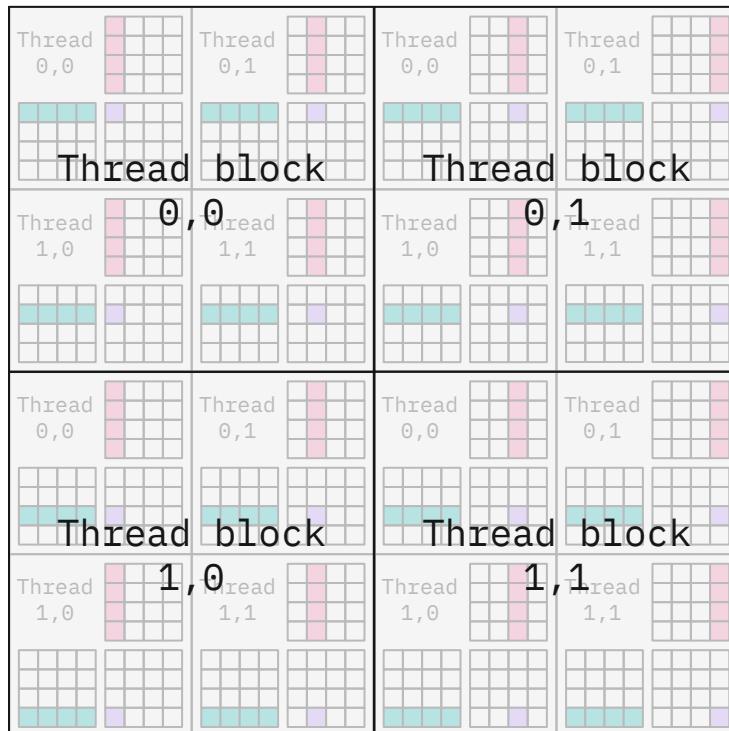
# Matmul with block-level model (Triton)

A @ B = C



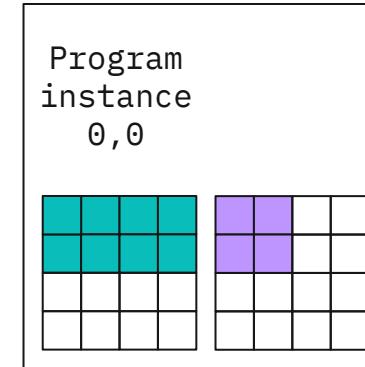
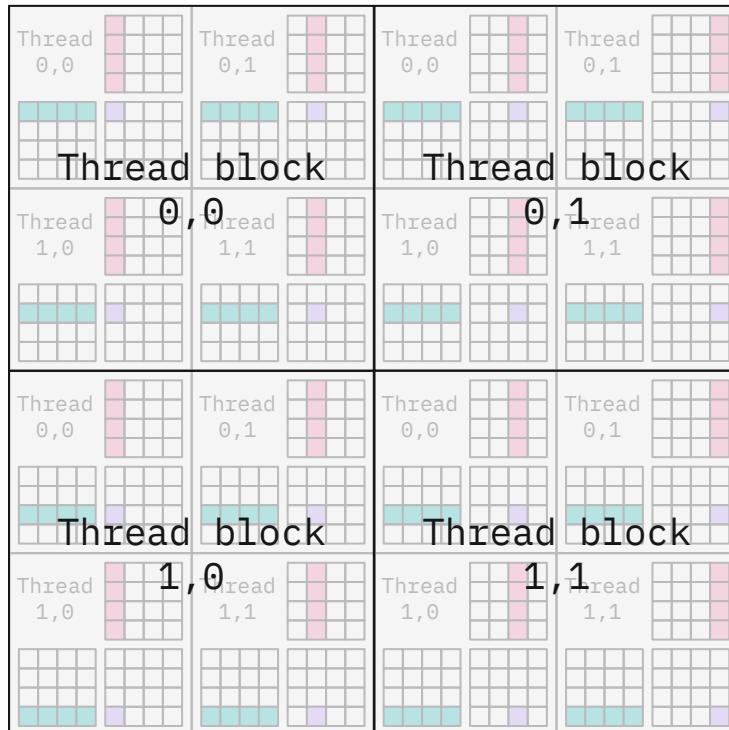
# Matmul with block-level model (Triton)

A @ B = C



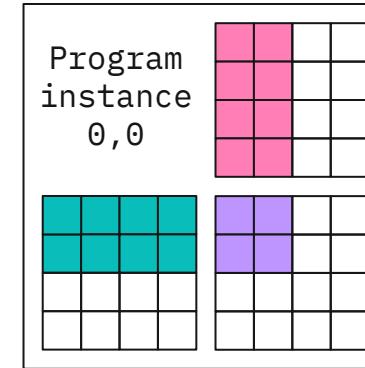
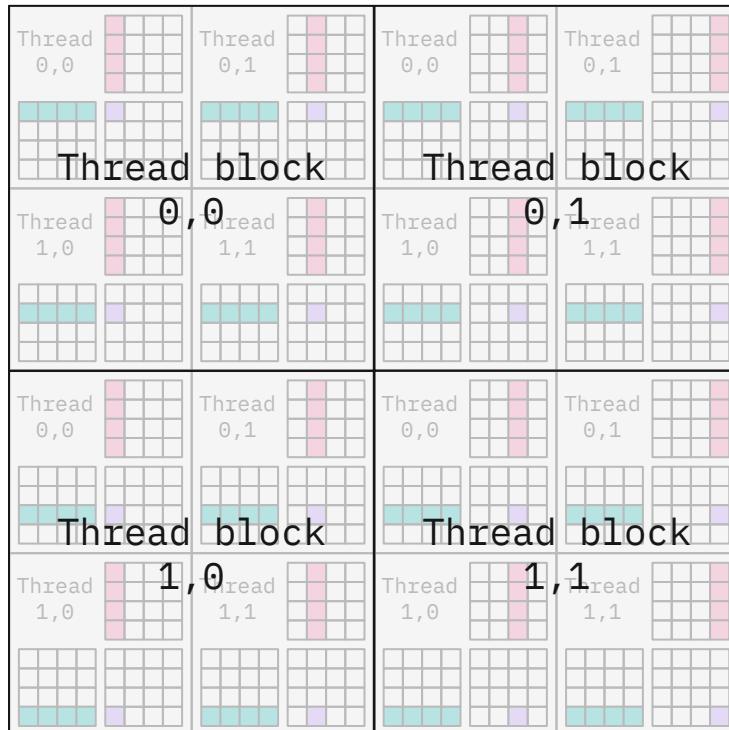
# Matmul with block-level model (Triton)

$$\textcolor{teal}{A} @ \textcolor{pink}{B} = \textcolor{violet}{C}$$



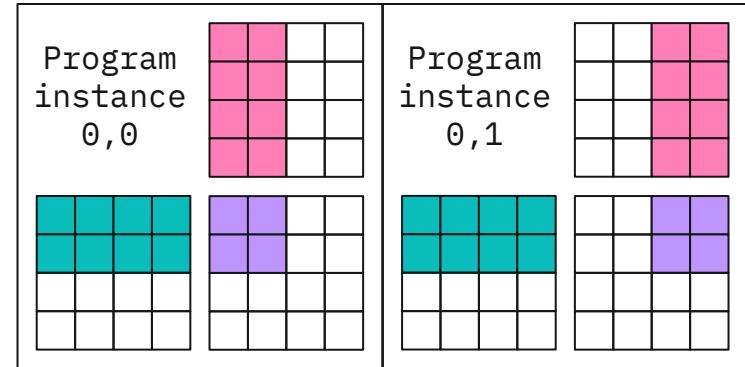
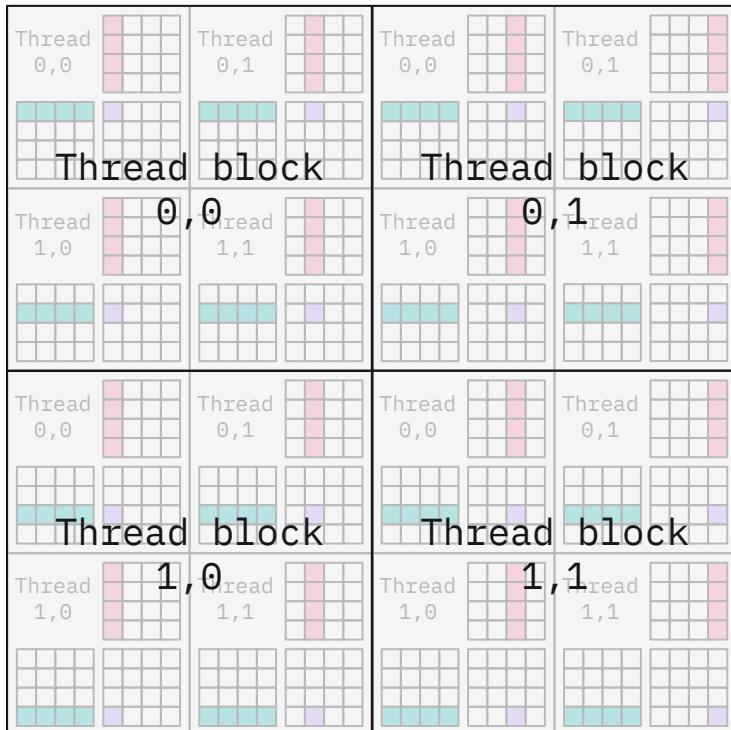
# Matmul with block-level model (Triton)

A @ B = C



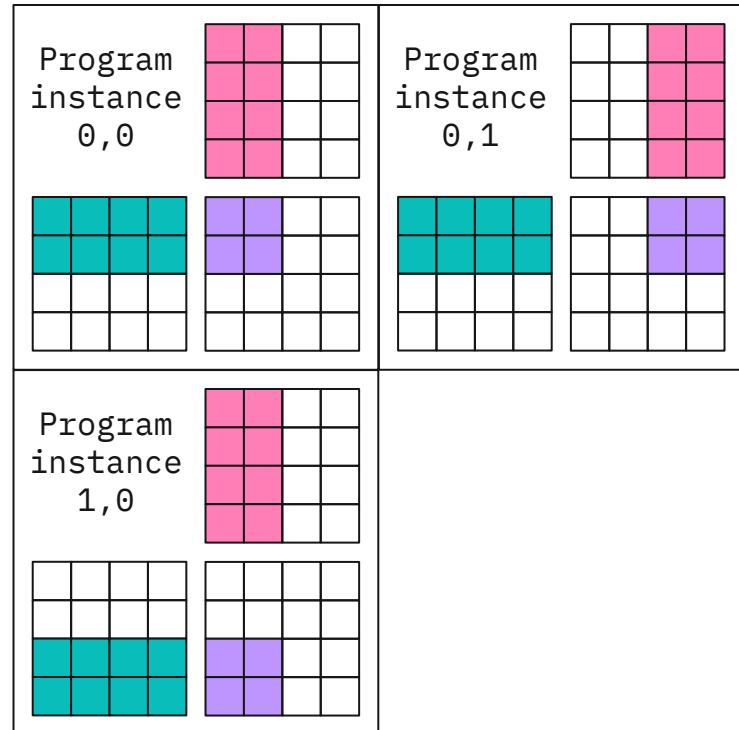
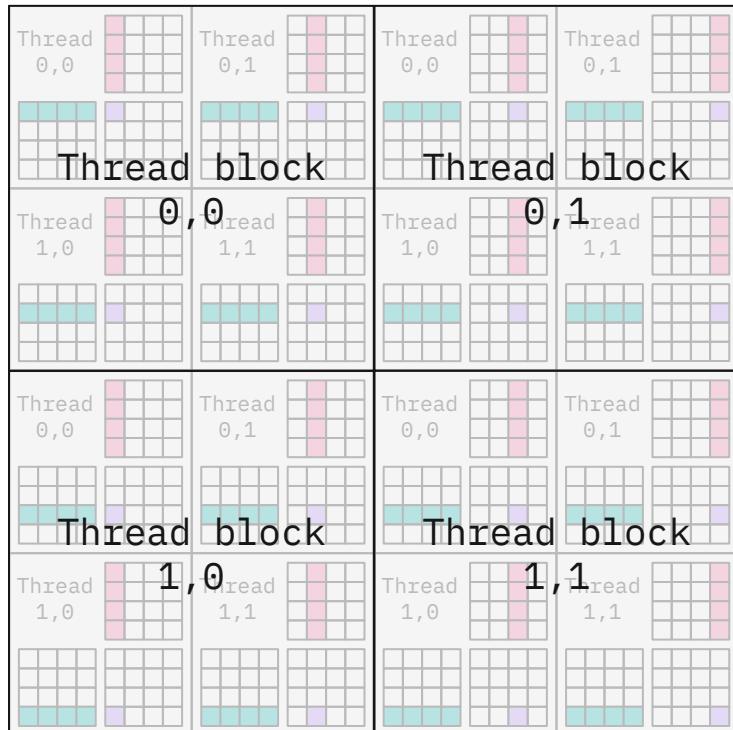
# Matmul with block-level model (Triton)

A @ B = C



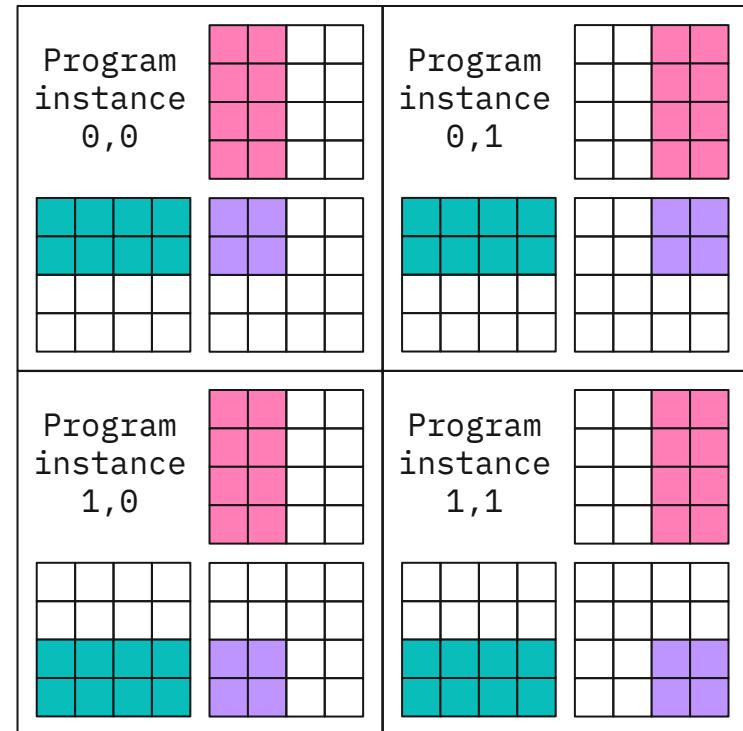
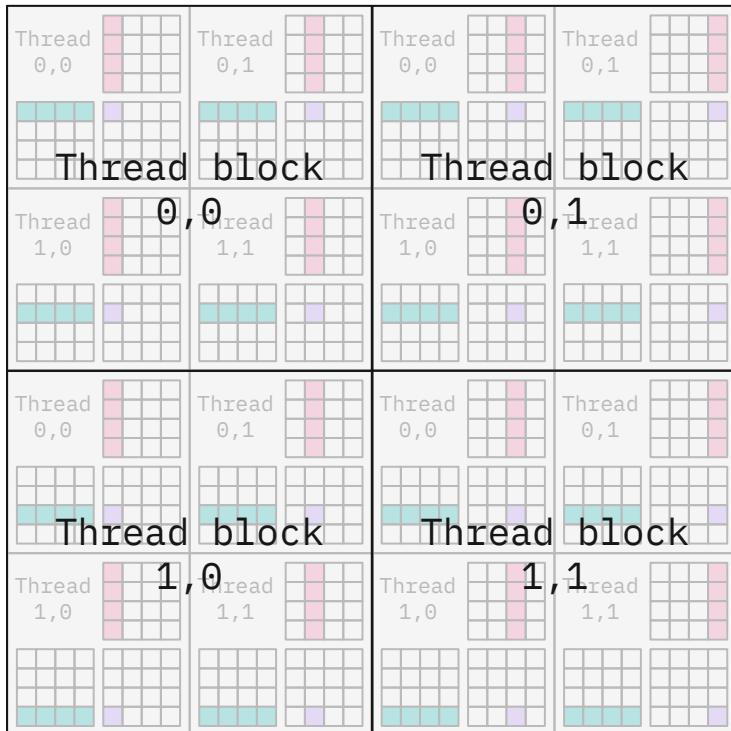
# Matmul with block-level model (Triton)

A @ B = C



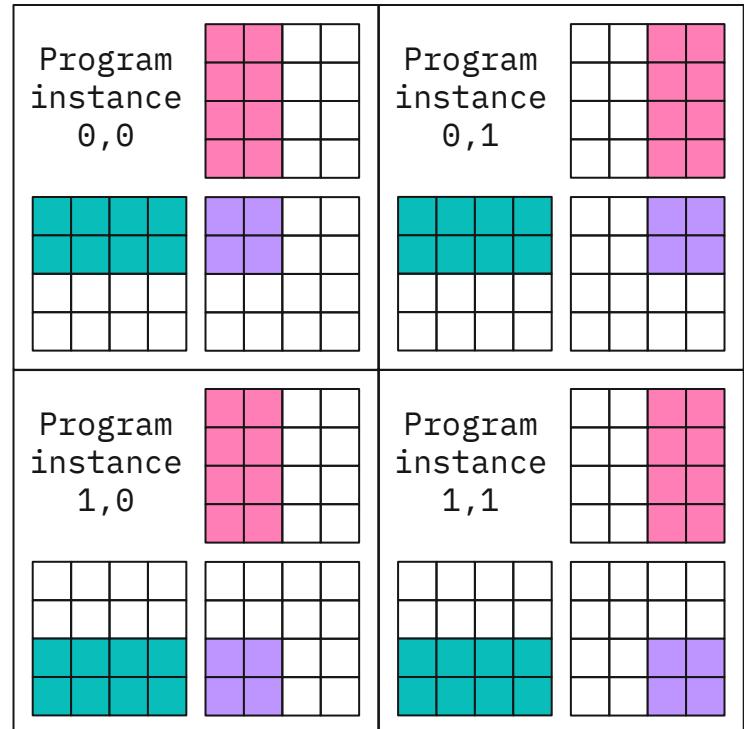
# Matmul with block-level model (Triton)

$$\textcolor{teal}{A} @ \textcolor{pink}{B} = \textcolor{violet}{C}$$



# Block-level programming model

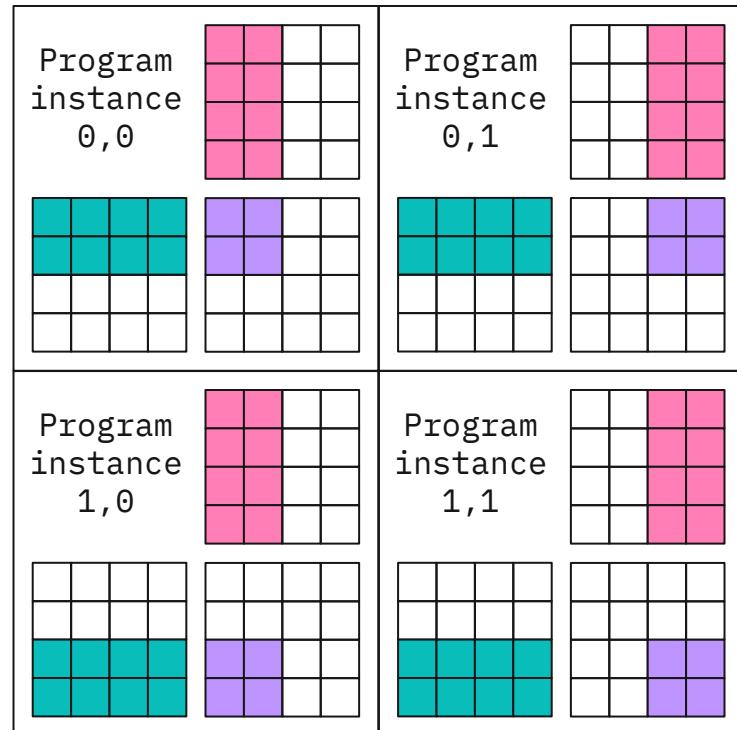
$$\textcolor{teal}{A} @ \textcolor{black}{B} = \textcolor{magenta}{C}$$



# Block-level programming model

$$\textcolor{teal}{A} @ \textcolor{pink}{B} = \textcolor{violet}{C}$$

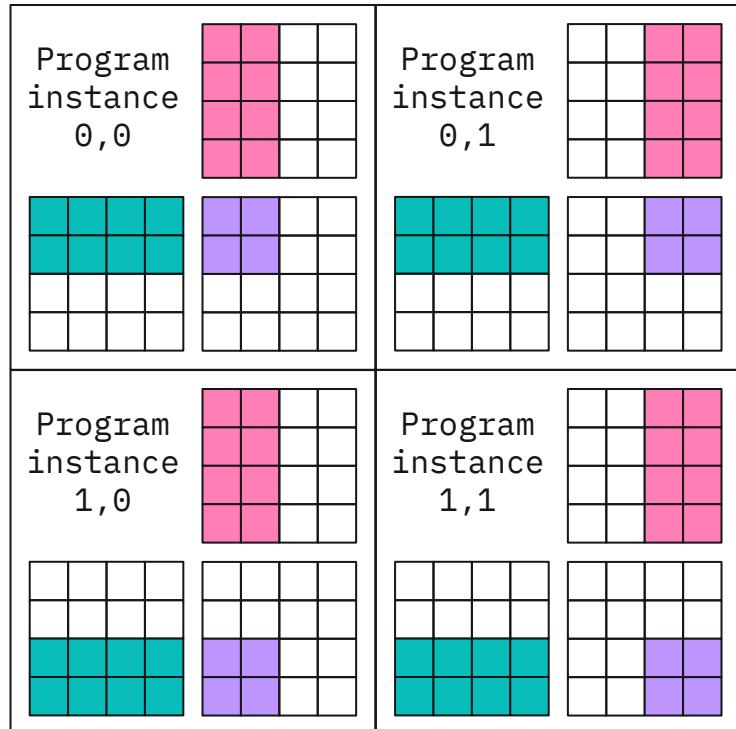
- Programmers define parallel computations using a block-level abstraction.



# Block-level programming model

$$\textcolor{teal}{A} @ \textcolor{pink}{B} = \textcolor{purple}{C}$$

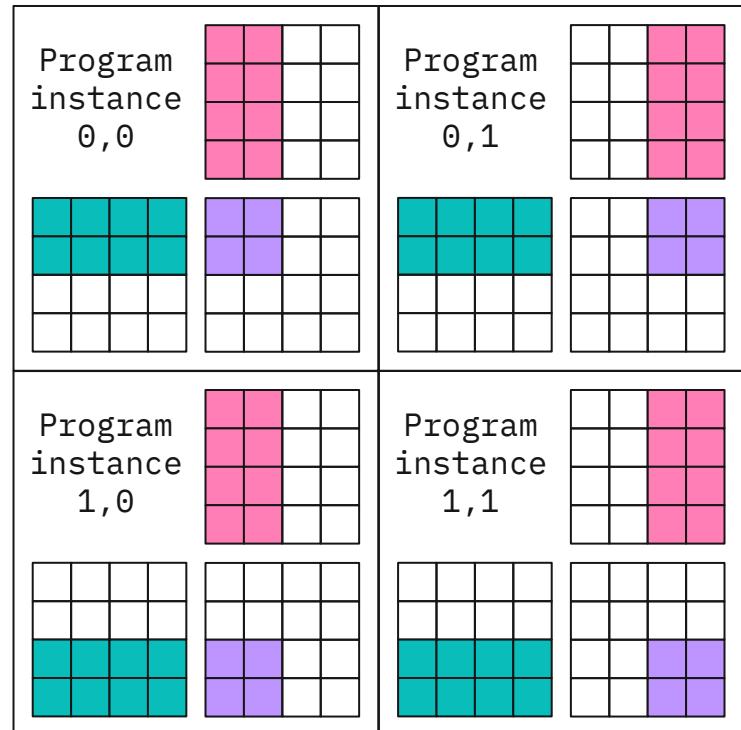
- Programmers define parallel computations using a block-level abstraction.
  - Above the thread-level programming model (CUDA, OpenCL, HIP)



# Block-level programming model

$$\textcolor{teal}{A} @ \textcolor{pink}{B} = \textcolor{violet}{C}$$

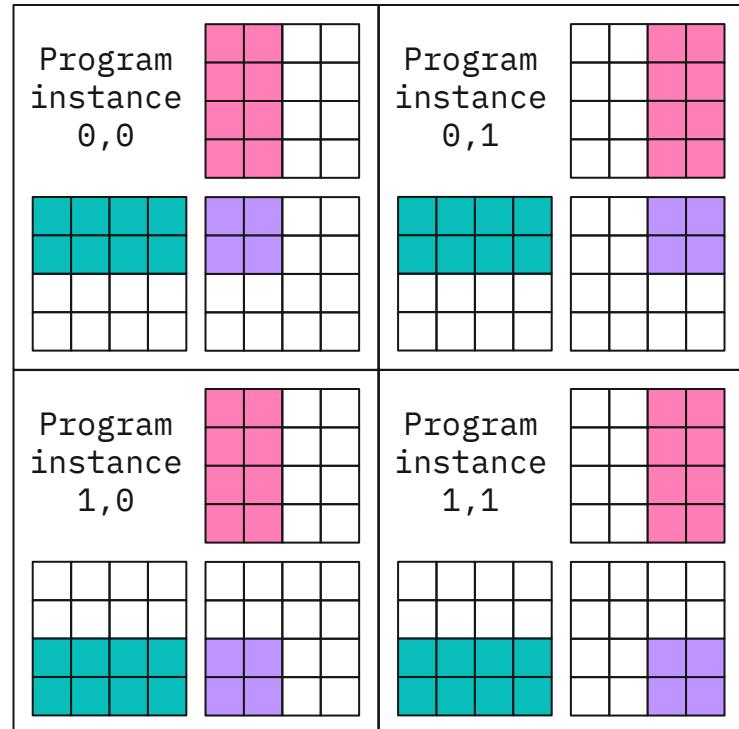
- Programmers define parallel computations using a block-level abstraction.
  - Above the thread-level programming model (CUDA, OpenCL, HIP)
- Partition output tensor into blocks (tiles).



# Block-level programming model

$$\textcolor{teal}{A} @ \textcolor{pink}{B} = \textcolor{purple}{C}$$

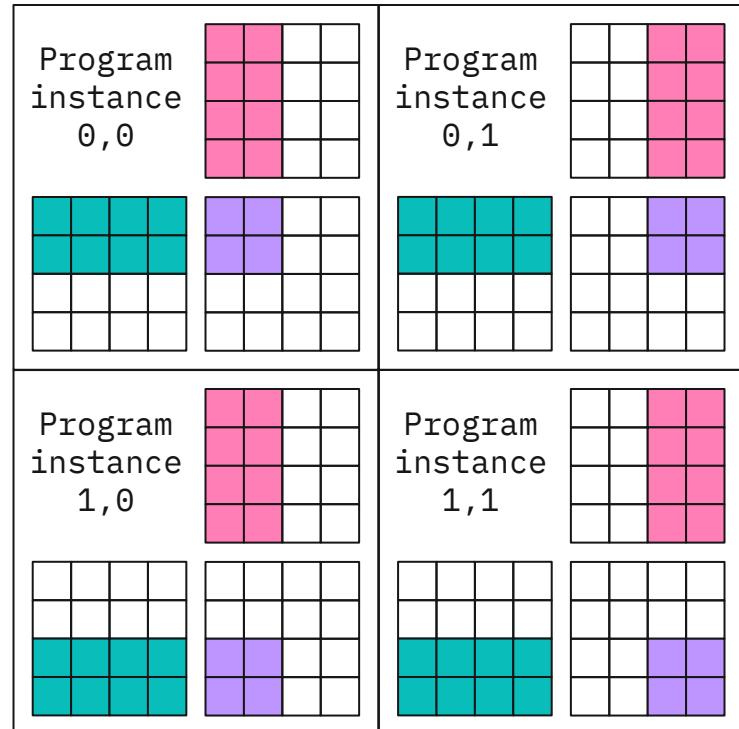
- Programmers define parallel computations using a block-level abstraction.
  - Above the thread-level programming model (CUDA, OpenCL, HIP)
- Partition output tensor into blocks (tiles).
  - Many program instances are launched in parallel from a kernel grid.



# Block-level programming model

$$\textcolor{teal}{A} @ \textcolor{pink}{B} = \textcolor{purple}{C}$$

- Programmers define parallel computations using a block-level abstraction.
  - Above the thread-level programming model (CUDA, OpenCL, HIP)
- Partition output tensor into blocks (tiles).
  - Many program instances are launched in parallel from a kernel grid.
  - Each program computes a different block of the output



# Matmul Triton kernel

# Matmul Triton kernel

```
@triton.jit
def matmul_kernel(a_ptr, b_ptr, c_ptr, M, N, K,
                  stride_am, stride_ak, stride_bk, stride_bn, stride_cm, stride_cn,
                  BLOCK_SIZE_M: tl.constexpr, BLOCK_SIZE_N: tl.constexpr, BLOCK_SIZE_K: tl.constexpr):
    pid_m = tl.program_id(axis=0)
    pid_n = tl.program_id(axis=1)
    offs_am = (pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M))
    offs_bn = (pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N))
    offs_k = tl.arange(0, BLOCK_SIZE_K)
    a_ptrs = a_ptr + (offs_am[:, None] * stride_am + offs_k[None, :] * stride_ak)
    b_ptrs = b_ptr + (offs_k[:, None] * stride_bk + offs_bn[None, :] * stride_bn)
    accumulator = tl.zeros((BLOCK_SIZE_M, BLOCK_SIZE_N), dtype=tl.float32)
    for k in range(0, tl.cdiv(K, BLOCK_SIZE_K)):
        a = tl.load(a_ptrs, mask=offs_k[None, :] < K - k * BLOCK_SIZE_K, other=0.0)
        b = tl.load(b_ptrs, mask=offs_k[:, None] < K - k * BLOCK_SIZE_K, other=0.0)
        accumulator = tl.dot(a, b, accumulator)
        a_ptrs += BLOCK_SIZE_K * stride_ak
        b_ptrs += BLOCK_SIZE_K * stride_bk
    c = accumulator.to(tl.float16)
    offs_cm = pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M)
    offs_cn = pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N)
    c_ptrs = c_ptr + stride_cm * offs_cm[:, None] + stride_cn * offs_cn[None, :]
    c_mask = (offs_cm[:, None] < M) & (offs_cn[None, :] < N)
    tl.store(c_ptrs, c, mask=c_mask)
```

# Matmul Triton kernel

```
@triton.jit
def matmul_kernel(a_ptr, b_ptr, c_ptr, M, N, K,
                  stride_am, stride_ak, stride_bk, stride_bn, stride_cm, stride_cn,
                  BLOCK_SIZE_M: tl.constexpr, BLOCK_SIZE_N: tl.constexpr, BLOCK_SIZE_K: tl.constexpr):
    pid_m = tl.program_id(axis=0)
    pid_n = tl.program_id(axis=1)
    offs_am = (pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M))
    offs_bn = (pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N))
    offs_k = tl.arange(0, BLOCK_SIZE_K)
    a_ptrs = a_ptr + (offs_am[:, None] * stride_am + offs_k[None, :] * stride_ak)
    b_ptrs = b_ptr + (offs_k[:, None] * stride_bk + offs_bn[None, :] * stride_bn)
    accumulator = tl.zeros((BLOCK_SIZE_M, BLOCK_SIZE_N), dtype=tl.float32)
    for k in range(0, tl.cdiv(K, BLOCK_SIZE_K)):
        a = tl.load(a_ptrs, mask=offs_k[None, :] < K - k * BLOCK_SIZE_K, other=0.0)
        b = tl.load(b_ptrs, mask=offs_k[:, None] < K - k * BLOCK_SIZE_K, other=0.0)
        accumulator = tl.dot(a, b, accumulator)
        a_ptrs += BLOCK_SIZE_K * stride_ak
        b_ptrs += BLOCK_SIZE_K * stride_bk
    c = accumulator.to(tl.float16)
    offs_cm = pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M)
    offs_cn = pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N)
    c_ptrs = c_ptr + stride_cm * offs_cm[:, None] + stride_cn * offs_cn[None, :]
    c_mask = (offs_cm[:, None] < M) & (offs_cn[None, :] < N)
    tl.store(c_ptrs, c, mask=c_mask)
```

# Matmul Triton kernel

```
@triton.jit
def matmul_kernel(a_ptr, b_ptr, c_ptr, M, N, K,
                  stride_am, stride_ak, stride_bk, stride_bn, stride_cm, stride_cn,
                  BLOCK_SIZE_M: tl.constexpr, BLOCK_SIZE_N: tl.constexpr, BLOCK_SIZE_K: tl.constexpr):
    pid_m = tl.program_id(axis=0)
    pid_n = tl.program_id(axis=1)
    offs_am = (pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M))
    offs_bn = (pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N))
    offs_k = tl.arange(0, BLOCK_SIZE_K)
    a_ptrs = a_ptr + (offs_am[:, None] * stride_am + offs_k[None, :] * stride_ak)
    b_ptrs = b_ptr + (offs_k[:, None] * stride_bk + offs_bn[None, :] * stride_bn)
    accumulator = tl.zeros((BLOCK_SIZE_M, BLOCK_SIZE_N), dtype=tl.float32)
    for k in range(0, tl.cdiv(K, BLOCK_SIZE_K)):
        a = tl.load(a_ptrs, mask=offs_k[None, :] < K - k * BLOCK_SIZE_K, other=0.0)
        b = tl.load(b_ptrs, mask=offs_k[:, None] < K - k * BLOCK_SIZE_K, other=0.0)
        accumulator = tl.dot(a, b, accumulator)
        a_ptrs += BLOCK_SIZE_K * stride_ak
        b_ptrs += BLOCK_SIZE_K * stride_bk
    c = accumulator.to(tl.float16)
    offs_cm = pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M)
    offs_cn = pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N)
    c_ptrs = c_ptr + stride_cm * offs_cm[:, None] + stride_cn * offs_cn[None, :]
    c_mask = (offs_cm[:, None] < M) & (offs_cn[None, :] < N)
    tl.store(c_ptrs, c, mask=c_mask)
```

# Matmul Triton kernel

```
@triton.jit
def matmul_kernel(a_ptr, b_ptr, c_ptr, M, N, K,
                  stride_am, stride_ak, stride_bk, stride_bn, stride_cm, stride_cn,
                  BLOCK_SIZE_M: tl.constexpr, BLOCK_SIZE_N: tl.constexpr, BLOCK_SIZE_K: tl.constexpr):
    pid_m = tl.program_id(axis=0)
    pid_n = tl.program_id(axis=1)
    offs_am = (pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M))
    offs_bn = (pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N))
    offs_k = tl.arange(0, BLOCK_SIZE_K)
    a_ptrs = a_ptr + (offs_am[:, None] * stride_am + offs_k[None, :] * stride_ak)
    b_ptrs = b_ptr + (offs_k[:, None] * stride_bk + offs_bn[None, :] * stride_bn)
    accumulator = tl.zeros((BLOCK_SIZE_M, BLOCK_SIZE_N), dtype=tl.float32)
    for k in range(0, tl.cdiv(K, BLOCK_SIZE_K)):
        a = tl.load(a_ptrs, mask=offs_k[None, :] < K - k * BLOCK_SIZE_K, other=0.0)
        b = tl.load(b_ptrs, mask=offs_k[:, None] < K - k * BLOCK_SIZE_K, other=0.0)
        accumulator = tl.dot(a, b, accumulator)
        a_ptrs += BLOCK_SIZE_K * stride_ak
        b_ptrs += BLOCK_SIZE_K * stride_bk
    c = accumulator.to(tl.float16)
    offs_cm = pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M)
    offs_cn = pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N)
    c_ptrs = c_ptr + stride_cm * offs_cm[:, None] + stride_cn * offs_cn[None, :]
    c_mask = (offs_cm[:, None] < M) & (offs_cn[None, :] < N)
    tl.store(c_ptrs, c, mask=c_mask)
```

# Matmul Triton kernel

```
@triton.jit
def matmul_kernel(a_ptr, b_ptr, c_ptr, M, N, K,
                  stride_am, stride_ak, stride_bk, stride_bn, stride_cm, stride_cn,
                  BLOCK_SIZE_M: tl.constexpr, BLOCK_SIZE_N: tl.constexpr, BLOCK_SIZE_K: tl.constexpr):
    pid_m = tl.program_id(axis=0)
    pid_n = tl.program_id(axis=1)
    offs_am = (pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M))
    offs_bn = (pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N))
    offs_k = tl.arange(0, BLOCK_SIZE_K)
    a_ptrs = a_ptr + (offs_am[:, None] * stride_am + offs_k[None, :] * stride_ak)
    b_ptrs = b_ptr + (offs_k[:, None] * stride_bk + offs_bn[None, :] * stride_bn)
    accumulator = tl.zeros((BLOCK_SIZE_M, BLOCK_SIZE_N), dtype=tl.float32)
    for k in range(0, tl.cdiv(K, BLOCK_SIZE_K)):
        a = tl.load(a_ptrs, mask=offs_k[None, :] < K - k * BLOCK_SIZE_K, other=0.0)
        b = tl.load(b_ptrs, mask=offs_k[:, None] < K - k * BLOCK_SIZE_K, other=0.0)
        accumulator = tl.dot(a, b, accumulator)
        a_ptrs += BLOCK_SIZE_K * stride_ak
        b_ptrs += BLOCK_SIZE_K * stride_bk
    c = accumulator.to(tl.float16)
    offs_cm = pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M)
    offs_cn = pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N)
    c_ptrs = c_ptr + stride_cm * offs_cm[:, None] + stride_cn * offs_cn[None, :]
    c_mask = (offs_cm[:, None] < M) & (offs_cn[None, :] < N)
    tl.store(c_ptrs, c, mask=c_mask)
```

# Matmul Triton kernel

```
@triton.jit
def matmul_kernel(a_ptr, b_ptr, c_ptr, M, N, K,
                  stride_am, stride_ak, stride_bk, stride_bn, stride_cm, stride_cn,
                  BLOCK_SIZE_M: tl.constexpr, BLOCK_SIZE_N: tl.constexpr, BLOCK_SIZE_K: tl.constexpr):
    pid_m = tl.program_id(axis=0)
    pid_n = tl.program_id(axis=1)
    offs_am = (pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M))
    offs_bn = (pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N))
    offs_k = tl.arange(0, BLOCK_SIZE_K)
    a_ptrs = a_ptr + (offs_am[:, None] * stride_am + offs_k[None, :] * stride_ak)
    b_ptrs = b_ptr + (offs_k[:, None] * stride_bk + offs_bn[None, :] * stride_bn)
    accumulator = tl.zeros((BLOCK_SIZE_M, BLOCK_SIZE_N), dtype=tl.float32)
    for k in range(0, tl.cdiv(K, BLOCK_SIZE_K)):
        a = tl.load(a_ptrs, mask=offs_k[None, :] < K - k * BLOCK_SIZE_K, other=0.0)
        b = tl.load(b_ptrs, mask=offs_k[:, None] < K - k * BLOCK_SIZE_K, other=0.0)
        accumulator = tl.dot(a, b, accumulator)
        a_ptrs += BLOCK_SIZE_K * stride_ak
        b_ptrs += BLOCK_SIZE_K * stride_bk
    c = accumulator.to(tl.float16)
    offs_cm = pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M)
    offs_cn = pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N)
    c_ptrs = c_ptr + stride_cm * offs_cm[:, None] + stride_cn * offs_cn[None, :]
    c_mask = (offs_cm[:, None] < M) & (offs_cn[None, :] < N)
    tl.store(c_ptrs, c, mask=c_mask)
```

# Matmul Triton kernel launch

# Matmul Triton kernel launch

```
def matmul(a, b):
    M, K = a.shape
    K, N = b.shape
    c = torch.empty((M, N), device=a.device, dtype=torch.float16)
    grid = lambda META: (triton.cdiv(M, META['BLOCK_SIZE_M']), triton.cdiv(N, META['BLOCK_SIZE_N']), )
    matmul_kernel[grid](
        a, b, c,
        M, N, K,
        a.stride(0), a.stride(1),
        b.stride(0), b.stride(1),
        c.stride(0), c.stride(1)
    )
    return c
```

# Matmul Triton kernel launch

```
def matmul(a, b):
    M, K = a.shape
    K, N = b.shape
    c = torch.empty((M, N), device=a.device, dtype=torch.float16)
    grid = lambda META: (triton.cdiv(M, META['BLOCK_SIZE_M']), triton.cdiv(N, META['BLOCK_SIZE_N']), )
    matmul_kernel[grid](
        a, b, c,
        M, N, K,
        a.stride(0), a.stride(1),
        b.stride(0), b.stride(1),
        c.stride(0), c.stride(1)
    )
    return c
```

# Matmul Triton kernel launch

```
def matmul(a, b):
    M, K = a.shape
    K, N = b.shape
    c = torch.empty((M, N), device=a.device, dtype=torch.float16)
    grid = lambda META: (triton.cdiv(M, META['BLOCK_SIZE_M']), triton.cdiv(N, META['BLOCK_SIZE_N']), )
    matmul_kernel[grid](
        a, b, c,
        M, N, K,
        a.stride(0), a.stride(1),
        b.stride(0), b.stride(1),
        c.stride(0), c.stride(1)
    )
    return c
```

# Matmul Triton kernel launch

```
def matmul(a, b):
    M, K = a.shape
    K, N = b.shape
    c = torch.empty((M, N), device=a.device, dtype=torch.float16)
    grid = lambda META: (triton.cdiv(M, META['BLOCK_SIZE_M']), triton.cdiv(N, META['BLOCK_SIZE_N']), )
    matmul_kernel[grid](
        a, b, c,
        M, N, K,
        a.stride(0), a.stride(1),
        b.stride(0), b.stride(1),
        c.stride(0), c.stride(1)
    )
    return c
```

What is abstracted away by Triton's block-level programming model?

# Matmul CUDA kernel

# Matmul CUDA kernel

```
template <int BLOCK_SIZE> __global__ void MatrixMulCUDA(float *C, float *A, float *B, int wA, int wB) {
    int bx = blockIdx.x;
    int by = blockIdx.y;
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    int aBegin = wA * BLOCK_SIZE * by;
    int aEnd = aBegin + wA - 1;
    int aStep = BLOCK_SIZE;
    int bBegin = BLOCK_SIZE * bx;
    int bStep = BLOCK_SIZE * wB;
    float Csub = 0;
    for (int a = aBegin, b = bBegin; a <= aEnd; a += aStep, b += bStep) {
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
        As[ty][tx] = A[a + wA * ty + tx];
        Bs[ty][tx] = B[b + wB * ty + tx];
        __syncthreads();
        #pragma unroll
        for (int k = 0; k < BLOCK_SIZE; ++k) {
            Csub += As[ty][k] * Bs[k][tx];
        }
        __syncthreads();
    }
    int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    C[c + wB * ty + tx] = Csub;
}
```

# Matmul CUDA kernel

```
template <int BLOCK_SIZE> __global__ void MatrixMulCUDA(float *C, float *A, float *B, int wA, int wB) {  
    int bx = blockIdx.x;  
    int by = blockIdx.y;  
    int tx = threadIdx.x;  
    int ty = threadIdx.y;  
    int aBegin = wA * BLOCK_SIZE * by;  
    int aEnd = aBegin + wA - 1;  
    int aStep = BLOCK_SIZE;  
    int bBegin = BLOCK_SIZE * bx;  
    int bStep = BLOCK_SIZE * wB;  
    float Csub = 0;  
    for (int a = aBegin, b = bBegin; a <= aEnd; a += aStep, b += bStep) {  
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];  
        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];  
        As[ty][tx] = A[a + wA * ty + tx];  
        Bs[ty][tx] = B[b + wB * ty + tx];  
        __syncthreads();  
        #pragma unroll  
        for (int k = 0; k < BLOCK_SIZE; ++k) {  
            Csub += As[ty][k] * Bs[k][tx];  
        }  
        __syncthreads();  
    }  
    int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;  
    C[c + wB * ty + tx] = Csub;  
}
```

- Thread- & thread-block-level program mapping

# Matmul CUDA kernel

```
template <int BLOCK_SIZE> __global__ void MatrixMulCUDA(float *C, float *A, float *B, int wA, int wB) {
    int bx = blockIdx.x;
    int by = blockIdx.y;
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    int aBegin = wA * BLOCK_SIZE * by;
    int aEnd = aBegin + wA - 1;
    int aStep = BLOCK_SIZE;
    int bBegin = BLOCK_SIZE * bx;
    int bStep = BLOCK_SIZE * wB;
    float Csub = 0;
    for (int a = aBegin, b = bBegin; a <= aEnd; a += aStep, b += bStep) {
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
        As[ty][tx] = A[a + wA * ty + tx];
        Bs[ty][tx] = B[b + wB * ty + tx];
        __syncthreads();
        #pragma unroll
        for (int k = 0; k < BLOCK_SIZE; ++k) {
            Csub += As[ty][k] * Bs[k][tx];
        }
        __syncthreads();
    }
    int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    C[c + wB * ty + tx] = Csub;
}
```

- Thread- & thread-block-level program mapping
- Shared memory

# Matmul CUDA kernel

```
template <int BLOCK_SIZE> __global__ void MatrixMulCUDA(float *C, float *A, float *B, int wA, int wB) {
    int bx = blockIdx.x;
    int by = blockIdx.y;
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    int aBegin = wA * BLOCK_SIZE * by;
    int aEnd = aBegin + wA - 1;
    int aStep = BLOCK_SIZE;
    int bBegin = BLOCK_SIZE * bx;
    int bStep = BLOCK_SIZE * wB;
    float Csub = 0;
    for (int a = aBegin, b = bBegin; a <= aEnd; a += aStep, b += bStep) {
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
        As[ty][tx] = A[a + wA * ty + tx];
        Bs[ty][tx] = B[b + wB * ty + tx];
        __syncthreads();
        #pragma unroll
        for (int k = 0; k < BLOCK_SIZE; ++k) {
            Csub += As[ty][k] * Bs[k][tx];
        }
        __syncthreads();
    }
    int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    C[c + wB * ty + tx] = Csub;
}
```

- Thread- & thread-block-level program mapping
- Shared memory
- Thread coalescence

# Matmul CUDA kernel

```
template <int BLOCK_SIZE> __global__ void MatrixMulCUDA(float *C, float *A, float *B, int wA, int wB) {
    int bx = blockIdx.x;
    int by = blockIdx.y;
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    int aBegin = wA * BLOCK_SIZE * by;
    int aEnd = aBegin + wA - 1;
    int aStep = BLOCK_SIZE;
    int bBegin = BLOCK_SIZE * bx;
    int bStep = BLOCK_SIZE * wB;
    float Csub = 0;
    for (int a = aBegin, b = bBegin; a <= aEnd; a += aStep, b += bStep) {
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
        As[ty][tx] = A[a + wA * ty + tx];
        Bs[ty][tx] = B[b + wB * ty + tx];
        __syncthreads();
        #pragma unroll
        for (int k = 0; k < BLOCK_SIZE; ++k) {
            Csub += As[ty][k] * Bs[k][tx];
        }
        __syncthreads();
    }
    int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    C[c + wB * ty + tx] = Csub;
}
```

- Thread- & thread-block-level program mapping
- Shared memory
- Thread coalescence
- Thread synchronization

# Matmul CUDA kernel

```
template <int BLOCK_SIZE> __global__ void MatrixMulCUDA(float *C, float *A, float *B, int wA, int wB) {  
    int bx = blockIdx.x;  
    int by = blockIdx.y;  
    int tx = threadIdx.x;  
    int ty = threadIdx.y;  
    int aBegin = wA * BLOCK_SIZE * by;  
    int aEnd = aBegin + wA - 1;  
    int aStep = BLOCK_SIZE;  
    int bBegin = BLOCK_SIZE * bx;  
    int bStep = BLOCK_SIZE * wB;  
    float Csub = 0;  
    for (int a = aBegin, b = bBegin; a <= aEnd; a += aStep, b += bStep) {  
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];  
        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];  
        As[ty][tx] = A[a + wA * ty + tx];  
        Bs[ty][tx] = B[b + wB * ty + tx];  
        __syncthreads();  
        #pragma unroll  
        for (int k = 0; k < BLOCK_SIZE; ++k) {  
            Csub += As[ty][k] * Bs[k][tx];  
        }  
        __syncthreads();  
    }  
    int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;  
    C[c + wB * ty + tx] = Csub;  
}
```

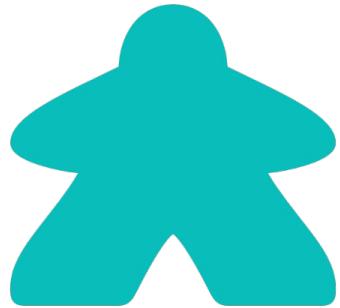
- Thread- & thread-block-level program mapping
- Shared memory
- Thread coalescence
- Thread synchronization
- ...

# Matmul CUDA kernel

```
template <int BLOCK_SIZE> __global__ void MatrixMulCUDA(float *C, float *A, float *B, int wA, int wB) {  
    int bx = blockIdx.x;  
    int by = blockIdx.y;  
    int tx = threadIdx.x;  
    int ty = threadIdx.y;  
    int aBegin = wA * BLOCK_SIZE * by;  
    int aEnd = aBegin + wA - 1;  
    int aStep = BLOCK_SIZE;  
    int bBegin = BLOCK_SIZE * bx;  
    int bStep = BLOCK_SIZE * wB;  
    float Csub = 0;  
    for (int a = aBegin, b = bBegin; a <= aEnd; a += aStep, b += bStep) {  
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];  
        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];  
        As[ty][tx] = A[a + wA * ty + tx];  
        Bs[ty][tx] = B[b + wB * ty + tx];  
        __syncthreads();  
        #pragma unroll  
        for (int k = 0; k < BLOCK_SIZE; ++k) {  
            Csub += As[ty][k] * Bs[k][tx];  
        }  
        __syncthreads();  
    }  
    int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;  
    C[c + wB * ty + tx] = Csub;  
}
```

- Thread- & thread-block-level program mapping
- Shared memory
- Thread coalescence
- Thread synchronization
- ...

Handled by the Triton compiler!



What wrong with Triton?

# Matmul Triton kernel

```
@triton.jit
def matmul_kernel(a_ptr, b_ptr, c_ptr, M, N, K,
                  stride_am, stride_ak, stride_bk, stride_bn, stride_cm, stride_cn,
                  BLOCK_SIZE_M: tl.constexpr, BLOCK_SIZE_N: tl.constexpr, BLOCK_SIZE_K: tl.constexpr):
    pid_m = tl.program_id(axis=0)
    pid_n = tl.program_id(axis=1)
    offs_am = (pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M))
    offs_bn = (pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N))
    offs_k = tl.arange(0, BLOCK_SIZE_K)
    a_ptrs = a_ptr + (offs_am[:, None] * stride_am + offs_k[None, :] * stride_ak)
    b_ptrs = b_ptr + (offs_k[:, None] * stride_bk + offs_bn[None, :] * stride_bn)
    accumulator = tl.zeros((BLOCK_SIZE_M, BLOCK_SIZE_N), dtype=tl.float32)
    for k in range(0, tl.cdiv(K, BLOCK_SIZE_K)):
        a = tl.load(a_ptrs, mask=offs_k[None, :] < K - k * BLOCK_SIZE_K, other=0.0)
        b = tl.load(b_ptrs, mask=offs_k[:, None] < K - k * BLOCK_SIZE_K, other=0.0)
        accumulator = tl.dot(a, b, accumulator)
        a_ptrs += BLOCK_SIZE_K * stride_ak
        b_ptrs += BLOCK_SIZE_K * stride_bk
    c = accumulator.to(tl.float16)
    offs_cm = pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M)
    offs_cn = pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N)
    c_ptrs = c_ptr + stride_cm * offs_cm[:, None] + stride_cn * offs_cn[None, :]
    c_mask = (offs_cm[:, None] < M) & (offs_cn[None, :] < N)
    tl.store(c_ptrs, c, mask=c_mask)
```

# Matmul Triton kernel

```
@triton.jit
def matmul_kernel(a_ptr, b_ptr, c_ptr, M, N, K,
                  stride_am, stride_ak, stride_bk, stride_bn, stride_cm, stride_cn,
                  BLOCK_SIZE_M: tl.constexpr, BLOCK_SIZE_N: tl.constexpr, BLOCK_SIZE_K: tl.constexpr):
    pid_m = tl.program_id(axis=0)
    pid_n = tl.program_id(axis=1)
    offs_am = (pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M))
    offs_bn = (pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N))
    offs_k = tl.arange(0, BLOCK_SIZE_K)
    a_ptrs = a_ptr + (offs_am[:, None] * stride_am + offs_k[None, :] * stride_ak)
    b_ptrs = b_ptr + (offs_k[:, None] * stride_bk + offs_bn[None, :] * stride_bn)
    accumulator = tl.zeros((BLOCK_SIZE_M, BLOCK_SIZE_N), dtype=tl.float32)
    for k in range(0, tl.cdiv(K, BLOCK_SIZE_K)):
        a = tl.load(a_ptrs, mask=offs_k[None, :] < K - k * BLOCK_SIZE_K, other=0.0)
        b = tl.load(b_ptrs, mask=offs_k[:, None] < K - k * BLOCK_SIZE_K, other=0.0)
        accumulator = tl.dot(a, b, accumulator)
        a_ptrs += BLOCK_SIZE_K * stride_ak
        b_ptrs += BLOCK_SIZE_K * stride_bk
    c = accumulator.to(tl.float16)
    offs_cm = pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M)
    offs_cn = pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N)
    c_ptrs = c_ptr + stride_cm * offs_cm[:, None] + stride_cn * offs_cn[None, :]
    c_mask = (offs_cm[:, None] < M) & (offs_cn[None, :] < N)
    tl.store(c_ptrs, c, mask=c_mask)
```

- Pointer arithmetic

# Matmul Triton kernel

```
@triton.jit
def matmul_kernel(a_ptr, b_ptr, c_ptr, M, N, K,
                  stride_am, stride_ak, stride_bk, stride_bn, stride_cm, stride_cn,
                  BLOCK_SIZE_M: tl.constexpr, BLOCK_SIZE_N: tl.constexpr, BLOCK_SIZE_K: tl.constexpr):
    pid_m = tl.program_id(axis=0)
    pid_n = tl.program_id(axis=1)
    offs_am = (pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M))
    offs_bn = (pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N))
    offs_k = tl.arange(0, BLOCK_SIZE_K)
    a_ptrs = a_ptr + (offs_am[:, None] * stride_am + offs_k[None, :] * stride_ak)
    b_ptrs = b_ptr + (offs_k[:, None] * stride_bk + offs_bn[None, :] * stride_bn)
    accumulator = tl.zeros((BLOCK_SIZE_M, BLOCK_SIZE_N), dtype=tl.float32)
    for k in range(0, tl.cdiv(K, BLOCK_SIZE_K)):
        a = tl.load(a_ptrs, mask=offs_k[None, :] < K - k * BLOCK_SIZE_K, other=0.0)
        b = tl.load(b_ptrs, mask=offs_k[:, None] < K - k * BLOCK_SIZE_K, other=0.0)
        accumulator = tl.dot(a, b, accumulator)
        a_ptrs += BLOCK_SIZE_K * stride_ak
        b_ptrs += BLOCK_SIZE_K * stride_bk
    c = accumulator.to(tl.float16)
    offs_cm = pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M)
    offs_cn = pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N)
    c_ptrs = c_ptr + stride_cm * offs_cm[:, None] + stride_cn * offs_cn[None, :]
    c_mask = (offs_cm[:, None] < M) & (offs_cn[None, :] < N)
    tl.store(c_ptrs, c, mask=c_mask)
```

- Pointer arithmetic
- Load/store masking

# Matmul Triton kernel

```
@triton.jit
def matmul_kernel(a_ptr, b_ptr, c_ptr, M, N, K,
                  stride_am, stride_ak, stride_bk, stride_bn, stride_cm, stride_cn,
                  BLOCK_SIZE_M: tl.constexpr, BLOCK_SIZE_N: tl.constexpr, BLOCK_SIZE_K: tl.constexpr):
    pid_m = tl.program_id(axis=0)
    pid_n = tl.program_id(axis=1)
    offs_am = (pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M))
    offs_bn = (pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N))
    offs_k = tl.arange(0, BLOCK_SIZE_K)
    a_ptrs = a_ptr + (offs_am[:, None] * stride_am + offs_k[None, :] * stride_ak)
    b_ptrs = b_ptr + (offs_k[:, None] * stride_bk + offs_bn[None, :] * stride_bn)
    accumulator = tl.zeros((BLOCK_SIZE_M, BLOCK_SIZE_N), dtype=tl.float32)
    for k in range(0, tl.cdiv(K, BLOCK_SIZE_K)):
        a = tl.load(a_ptrs, mask=offs_k[None, :] < K - k * BLOCK_SIZE_K, other=0.0)
        b = tl.load(b_ptrs, mask=offs_k[:, None] < K - k * BLOCK_SIZE_K, other=0.0)
        accumulator = tl.dot(a, b, accumulator)
        a_ptrs += BLOCK_SIZE_K * stride_ak
        b_ptrs += BLOCK_SIZE_K * stride_bk
    c = accumulator.to(tl.float16)
    offs_cm = pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M)
    offs_cn = pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N)
    c_ptrs = c_ptr + stride_cm * offs_cm[:, None] + stride_cn * offs_cn[None, :]
    c_mask = (offs_cm[:, None] < M) & (offs_cn[None, :] < N)
    tl.store(c_ptrs, c, mask=c_mask)
```

- Pointer arithmetic
- Load/store masking
- Hard to find an efficient schedule

# Matmul Triton kernel

```
@triton.jit
def matmul_kernel(a_ptr, b_ptr, c_ptr, M, N, K,
                  stride_am, stride_ak, stride_bk, stride_bn, stride_cm, stride_cn,
                  BLOCK_SIZE_M: tl.constexpr, BLOCK_SIZE_N: tl.constexpr, BLOCK_SIZE_K: tl.constexpr):
    pid_m = tl.program_id(axis=0)
    pid_n = tl.program_id(axis=1)
    offs_am = (pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M))
    offs_bn = (pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N))
    offs_k = tl.arange(0, BLOCK_SIZE_K)
    a_ptrs = a_ptr + (offs_am[:, None] * stride_am + offs_k[None, :] * stride_ak)
    b_ptrs = b_ptr + (offs_k[:, None] * stride_bk + offs_bn[None, :] * stride_bn)
    accumulator = tl.zeros((BLOCK_SIZE_M, BLOCK_SIZE_N), dtype=tl.float32)
    for k in range(0, tl.cdiv(K, BLOCK_SIZE_K)):
        a = tl.load(a_ptrs, mask=offs_k[None, :] < K - k * BLOCK_SIZE_K, other=0.0)
        b = tl.load(b_ptrs, mask=offs_k[:, None] < K - k * BLOCK_SIZE_K, other=0.0)
        accumulator = tl.dot(a, b, accumulator)
        a_ptrs += BLOCK_SIZE_K * stride_ak
        b_ptrs += BLOCK_SIZE_K * stride_bk
    c = accumulator.to(tl.float16)
    offs_cm = pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M)
    offs_cn = pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N)
    c_ptrs = c_ptr + stride_cm * offs_cm[:, None] + stride_cn * offs_cn[None, :]
    c_mask = (offs_cm[:, None] < M) & (offs_cn[None, :] < N)
    tl.store(c_ptrs, c, mask=c_mask)
```

- Pointer arithmetic
- Load/store masking
- Hard to find an efficient **schedule**

# Decoupled Languages

# **Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines**

Jonathan Ragan-Kelley\*    Andrew Adams\*    Sylvain Paris<sup>†</sup>    Marc Levoy<sup>‡</sup>    Saman Amarasinghe\*    Frédo Durand\*

\*MIT CSAIL

<sup>†</sup>Adobe

<sup>‡</sup>Stanford University

# **Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines**

Jonathan Ragan-Kelley

MIT CSAIL

Sylvain Paris

Adobe

Connelly Barnes

Adobe

Frédo Durand

MIT CSAIL

Andrew Adams

MIT CSAIL

Saman Amarasinghe

MIT CSAIL

# Decoupling the algorithm from the schedule

## Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines

Jonathan Ragan-Kelley\* Andrew Adams\* Sylvain Paris<sup>†</sup> Marc Levoy<sup>†</sup> Saman Amarasinghe\* Frédo Durand\*  
\*MIT CSAIL      <sup>†</sup>Adobe      <sup>†</sup>Stanford University

## Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines

Jonathan Ragan-Kelley  
MIT CSAIL

Sylvain Paris  
Adobe

Connelly Barnes  
Adobe

Frédo Durand  
MIT CSAIL

Andrew Adams  
MIT CSAIL

Saman Amarasinghe  
MIT CSAIL

# Decoupling the algorithm from the schedule

- Halide introduced the concept of algorithm and schedule in a programming language.

## Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines

Jonathan Ragan-Kelley\* Andrew Adams\* Sylvain Paris† Marc Levoy‡ Saman Amarasinghe\* Frédo Durand\*  
\*MIT CSAIL †Adobe ‡Stanford University

## Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines

Jonathan Ragan-Kelley  
MIT CSAIL

Sylvain Paris  
Adobe

Connelly Barnes  
Adobe

Frédo Durand  
MIT CSAIL

Andrew Adams  
MIT CSAIL

Saman Amarasinghe  
MIT CSAIL

# Decoupling the algorithm from the schedule

- Halide introduced the concept of algorithm and schedule in a programming language.
  - Algorithm: what to compute

## Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines

Jonathan Ragan-Kelley\* Andrew Adams\* Sylvain Paris† Marc Levoy‡ Saman Amarasinghe\* Frédo Durand\*  
\*MIT CSAIL †Adobe ‡Stanford University

## Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines

Jonathan Ragan-Kelley  
MIT CSAIL

Sylvain Paris  
Adobe

Connelly Barnes  
Adobe

Frédo Durand  
MIT CSAIL

Andrew Adams  
MIT CSAIL

Saman Amarasinghe  
MIT CSAIL

# Decoupling the algorithm from the schedule

- Halide introduced the concept of algorithm and schedule in a programming language.
  - Algorithm: what to compute
  - Schedule: how to compute

## Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines

Jonathan Ragan-Kelley\* Andrew Adams\* Sylvain Paris† Marc Levoy‡ Saman Amarasinghe\* Frédo Durand\*  
\*MIT CSAIL †Adobe ‡Stanford University

## Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines

Jonathan Ragan-Kelley  
MIT CSAIL

Sylvain Paris  
Adobe

Connelly Barnes  
Adobe

Frédo Durand  
MIT CSAIL

Andrew Adams  
MIT CSAIL

Saman Amarasinghe  
MIT CSAIL

# Decoupling the algorithm from the schedule

- Halide introduced the concept of algorithm and schedule in a programming language.
  - Algorithm: what to compute
  - Schedule: how to compute
- A function is defined by the composition of these two independent components.

## Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines

Jonathan Ragan-Kelley\* Andrew Adams\* Sylvain Paris† Marc Levoy‡ Saman Amarasinghe\* Frédo Durand\*  
\*MIT CSAIL †Adobe ‡Stanford University

## Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines

Jonathan Ragan-Kelley  
MIT CSAIL

Sylvain Paris  
Adobe

Connelly Barnes  
Adobe

Frédo Durand  
MIT CSAIL

Andrew Adams  
MIT CSAIL

Saman Amarasinghe  
MIT CSAIL

# Decoupling the algorithm from the schedule

- Halide introduced the concept of algorithm and schedule in a programming language.
  - Algorithm: what to compute
  - Schedule: how to compute
- A function is defined by the composition of these two independent components
- Decoupled Triton applies this concept to block-level machine learning kernels.

## Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines

Jonathan Ragan-Kelley\* Andrew Adams\* Sylvain Paris† Marc Levoy‡ Saman Amarasinghe\* Frédo Durand\*  
\*MIT CSAIL †Adobe ‡Stanford University

## Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines

Jonathan Ragan-Kelley MIT CSAIL	Connelly Barnes Adobe	Andrew Adams MIT CSAIL
Sylvain Paris Adobe	Frédo Durand MIT CSAIL	Saman Amarasinghe MIT CSAIL

# Efficient schedules

## Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines

Jonathan Ragan-Kelley\* Andrew Adams\* Sylvain Paris† Marc Levoy‡ Saman Amarasinghe\* Frédo Durand\*  
\*MIT CSAIL †Adobe ‡Stanford University

## Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines

Jonathan Ragan-Kelley  
MIT CSAIL

Sylvain Paris  
Adobe

Connelly Barnes  
Adobe

Frédo Durand  
MIT CSAIL

Andrew Adams  
MIT CSAIL

Saman Amarasinghe  
MIT CSAIL

# Efficient schedules

- The schedule determines performance factors.

## Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines

Jonathan Ragan-Kelley\* Andrew Adams\* Sylvain Paris† Marc Levoy‡ Saman Amarasinghe\* Frédo Durand\*  
\*MIT CSAIL †Adobe ‡Stanford University

## Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines

Jonathan Ragan-Kelley  
MIT CSAIL

Sylvain Paris  
Adobe

Connelly Barnes  
Adobe

Frédo Durand  
MIT CSAIL

Andrew Adams  
MIT CSAIL

Saman Amarasinghe  
MIT CSAIL

# Efficient schedules

- The schedule determines performance factors.
  - E.g. memory access pattern

## Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines

Jonathan Ragan-Kelley\* Andrew Adams\* Sylvain Paris† Marc Levoy‡ Saman Amarasinghe\* Frédo Durand\*  
\*MIT CSAIL †Adobe ‡Stanford University

## Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines

Jonathan Ragan-Kelley  
MIT CSAIL

Sylvain Paris  
Adobe

Connelly Barnes  
Adobe

Frédo Durand  
MIT CSAIL

Andrew Adams  
MIT CSAIL

Saman Amarasinghe  
MIT CSAIL

# Efficient schedules

- The schedule determines performance factors.
  - E.g. memory access pattern
- The most efficient schedule will depend on:

## Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines

Jonathan Ragan-Kelley\* Andrew Adams\* Sylvain Paris† Marc Levoy‡ Saman Amarasinghe\* Frédo Durand\*  
\*MIT CSAIL †Adobe ‡Stanford University

## Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines

Jonathan Ragan-Kelley  
MIT CSAIL

Sylvain Paris  
Adobe

Connelly Barnes  
Adobe

Frédo Durand  
MIT CSAIL

Andrew Adams  
MIT CSAIL

Saman Amarasinghe  
MIT CSAIL

# Efficient schedules

- The schedule determines performance factors.
  - E.g. memory access pattern
- The most efficient schedule will depend on:
  - Algorithm

## Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines

Jonathan Ragan-Kelley\* Andrew Adams\* Sylvain Paris† Marc Levoy‡ Saman Amarasinghe\* Frédo Durand\*  
\*MIT CSAIL †Adobe ‡Stanford University

## Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines

Jonathan Ragan-Kelley  
MIT CSAIL

Sylvain Paris  
Adobe

Connelly Barnes  
Adobe

Frédo Durand  
MIT CSAIL

Andrew Adams  
MIT CSAIL

Saman Amarasinghe  
MIT CSAIL

# Efficient schedules

- The schedule determines performance factors.
  - E.g. memory access pattern
- The most efficient schedule will depend on:
  - Algorithm
  - Workload size

## Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines

Jonathan Ragan-Kelley\* Andrew Adams\* Sylvain Paris† Marc Levoy‡ Saman Amarasinghe\* Frédo Durand\*  
\*MIT CSAIL †Adobe ‡Stanford University

## Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines

Jonathan Ragan-Kelley  
MIT CSAIL

Sylvain Paris  
Adobe

Connelly Barnes  
Adobe

Frédo Durand  
MIT CSAIL

Andrew Adams  
MIT CSAIL

Saman Amarasinghe  
MIT CSAIL

# Efficient schedules

- The schedule determines performance factors.
  - *E.g.* memory access pattern
- The most efficient schedule will depend on:
  - Algorithm
  - Workload size
  - Hardware architecture

## Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines

Jonathan Ragan-Kelley\* Andrew Adams\* Sylvain Paris† Marc Levoy‡ Saman Amarasinghe\* Frédo Durand\*  
\*MIT CSAIL †Adobe ‡Stanford University

## Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines

Jonathan Ragan-Kelley  
MIT CSAIL

Sylvain Paris  
Adobe

Connelly Barnes  
Adobe

Frédo Durand  
MIT CSAIL

Andrew Adams  
MIT CSAIL

Saman Amarasinghe  
MIT CSAIL

# Decoupled Triton example

# Decoupled Triton example

```
# Declarations
Func mm;
In A, B;
Var x, y;
RVar k;

# Algorithm
mm[x, y] = rdot(A[x, k], B[k, y], k);

# Schedule
mm.tensorize(x:128, k:32, y:128);
mm.block(x:128, y:128);
mm.map(x:x1/8, y, x1);
mm.num_warps(4);
mm.num_stages(3);
mm.compile();
```

# Decoupled Triton example

```
# Declarations
Func mm;
In A, B;
Var x, y;
RVar k;

# Algorithm
mm[x, y] = rdot(A[x, k], B[k, y], k);

# Schedule
mm.tensorize(x:128, k:32, y:128);
mm.block(x:128, y:128);
mm.map(x:x1/8, y, x1);
mm.num_warps(4);
mm.num_stages(3);
mm.compile();
```

# Decoupled Triton example

```
# Declarations
Func mm;
In A, B;
Var x, y;
RVar k;

# Algorithm
mm[x, y] = rdot(A[x, k], B[k, y], k);

# Schedule
mm.tensorize(x:128, k:32, y:128);
mm.block(x:128, y:128);
mm.map(x:x1/8, y, x1);
mm.num_warps(4);
mm.num_stages(3);
mm.compile();
```

# Decoupled Triton example

```
# Declarations
Func mm;
In A, B;
Var x, y;
RVar k;

# Algorithm
mm[x, y] = rdot(A[x, k], B[k, y], k);

# Schedule
mm.tensorize(x:128, k:32, y:128);
mm.block(x:128, y:128);
mm.map(x:x1/8, y, x1);
mm.num_warps(4);
mm.num_stages(3);
mm.compile();
```

# Decoupled Triton example

## # Declarations

```
Func mm;  
In A, B;  
Var x, y;  
RVar k;
```

## # Algorithm

```
mm[x, y] = rdot(A[x, k], B[k, y], k);
```

## # Schedule

```
mm.tensorize(x:128, k:32, y:128);  
mm.block(x:128, y:128);  
mm.map(x:x1/8, y, x1);  
mm.num_warps(4);  
mm.num_stages(3);  
mm.compile();
```

```
@triton.jit  
def mm_kernel(A_ptr, A_x_stride: tl.constexpr, A_k_stride:  
tl.constexpr, B_ptr, B_k_stride: tl.constexpr, B_y_stride:  
tl.constexpr, mm_ptr, mm_x_stride: tl.constexpr, mm_y_stride:  
tl.constexpr, y_SIZE: tl.constexpr, x_SIZE: tl.constexpr, k_SIZE:  
tl.constexpr):  
    y_BLOCK_COUNT = y_SIZE // 128  
    x_BLOCK_COUNT = x_SIZE // 128  
    x_pid = tl.program_id(0).to(tl.int64) // (y_BLOCK_COUNT * 8) %  
(x_BLOCK_COUNT // 8)  
    y_pid = tl.program_id(0).to(tl.int64) // 8 % y_BLOCK_COUNT  
    x1_pid = tl.program_id(0).to(tl.int64) % 8  
    x_pid_ = x_pid * 8 + x1_pid  
    y_block_start = y_pid * 128  
    x_block_start = x_pid_ * 128  
    k_arange = tl.arange(0, 32)  
    y_arange = tl.arange(0, 128)  
    x_arange = tl.arange(0, 128)  
    k_accumulator = tl.zeros((128, 128), tl.float32)  
    for k_iter in range(0, k_SIZE, 32):  
        A = tl.load(A_ptr + (x_block_start + x_arange[:, None]) *  
A_x_stride + (k_iter + k_arange[None, :]) * A_k_stride)  
        B = tl.load(B_ptr + (k_iter + k_arange[:, None]) * B_k_stride +  
(y_block_start + y_arange[None, :]) * B_y_stride)  
        k_accumulator = tl.dot(A, B, k_accumulator)  
        k_reduction_result = k_accumulator  
        mm = k_reduction_result  
        tl.store(mm_ptr + (x_block_start + x_arange[:, None]) * mm_x_stride +  
(y_block_start + y_arange[None, :]) * mm_y_stride, mm)  
  
def mm(B, A, y, x, k):  
    A_x_stride, A_k_stride, = A.stride()  
    B_k_stride, B_y_stride, = B.stride()  
    mm = torch.empty(y, x, dtype=torch.float32, device='cuda')  
    mm_x_stride, mm_y_stride, = mm.stride()  
    mm_grid = (triton.cdiv(x, 128) * triton.cdiv(y, 128)),  
    mm_kernel[mm_grid](A, A_x_stride, A_k_stride, B, B_k_stride,  
B_y_stride, mm, mm_x_stride, mm_y_stride, y, x, k, num_stages=3,  
num_warps=4)  
    return mm
```

# Decoupled Triton example

```
# Declarations
Func mm;
In A, B;
Var x, y;
RVar k;

# Algorithm
mm[x, y] = rdot(A[x, k], B[k, y], k);

# Schedule
mm.tensorize(x:128, k:32, y:128);
mm.block(x:128, y:128);
mm.map(x:x1/8, y, x1);
mm.num_warps(4);
mm.num_stages(3);
mm.compile();
```

More readable!

```
@triton.jit
def mm_kernel(A_ptr, A_x_stride: tl.constexpr, A_k_stride:
tl.constexpr, B_ptr, B_k_stride: tl.constexpr, B_y_stride:
tl.constexpr, mm_ptr, mm_x_stride: tl.constexpr, mm_y_stride:
tl.constexpr, y_SIZE: tl.constexpr, x_SIZE: tl.constexpr, k_SIZE:
tl.constexpr):
    y_BLOCK_COUNT = y_SIZE // 128
    x_BLOCK_COUNT = x_SIZE // 128
    x_pid = tl.program_id(0).to(tl.int64) // (y_BLOCK_COUNT * 8) %
(x_BLOCK_COUNT // 8)
    y_pid = tl.program_id(0).to(tl.int64) // 8 % y_BLOCK_COUNT
    x1_pid = tl.program_id(0).to(tl.int64) % 8
    x_pid_ = x_pid * 8 + x1_pid
    y_block_start = y_pid * 128
    x_block_start = x_pid_ * 128
    k_arange = tl.arange(0, 32)
    y_arange = tl.arange(0, 128)
    x_arange = tl.arange(0, 128)
    k_accumulator = tl.zeros((128, 128), tl.float32)
    for k_iter in range(0, k_SIZE, 32):
        A = tl.load(A_ptr + (x_block_start + x_arange[:, None]) *
A_x_stride + (k_iter + k_arange[None, :]) * A_k_stride)
        B = tl.load(B_ptr + (k_iter + k_arange[:, None]) * B_k_stride +
(y_block_start + y_arange[None, :]) * B_y_stride)
        k_accumulator = tl.dot(A, B, k_accumulator)
        k_reduction_result = k_accumulator
        mm = k_reduction_result
        tl.store(mm_ptr + (x_block_start + x_arange[:, None]) * mm_x_stride +
(y_block_start + y_arange[None, :]) * mm_y_stride, mm)

    def mm(B, A, y, x, k):
        A_x_stride, A_k_stride, = A.stride()
        B_k_stride, B_y_stride, = B.stride()
        mm = torch.empty(y, x, dtype=torch.float32, device='cuda')
        mm_x_stride, mm_y_stride, = mm.stride()
        mm_grid = (triton.cdiv(x, 128) * triton.cdiv(y, 128)),
        mm_kernel[mm_grid](A, A_x_stride, A_k_stride, B, B_k_stride,
B_y_stride, mm, mm_x_stride, mm_y_stride, y, x, k, num_stages=3,
num_warps=4)
        return mm
```

# Decoupled Triton example

```
# Declarations
Func mm;
In A, B;
Var x, y;
RVar k;

# Algorithm
mm[x, y] = rdot(A[x, k], B[k, y], k);

# Schedule
mm.tensorize(x:128, k:32, y:128);
mm.block(x:128, y:128);
mm.map(x:x1/8, y, x1);
mm.num_warps(4);
mm.num_stages(3);
mm.compile();
```

```
@triton.jit
def mm_kernel(A_ptr, A_x_stride: tl.constexpr, A_k_stride:
tl.constexpr, B_ptr, B_k_stride: tl.constexpr, B_y_stride:
tl.constexpr, mm_ptr, mm_x_stride: tl.constexpr, mm_y_stride:
tl.constexpr, y_SIZE: tl.constexpr, x_SIZE: tl.constexpr, k_SIZE:
tl.constexpr):
    y_BLOCK_COUNT = y_SIZE // 128
    x_BLOCK_COUNT = x_SIZE // 128
    x_pid = tl.program_id(0).to(tl.int64) // (y_BLOCK_COUNT * 8) %
(x_BLOCK_COUNT // 8)
    y_pid = tl.program_id(0).to(tl.int64) // 8 % y_BLOCK_COUNT
    x1_pid = tl.program_id(0).to(tl.int64) % 8
    x_pid_ = x_pid * 8 + x1_pid
    y_block_start = y_pid * 128
    x_block_start = x_pid_ * 128
    k_arange = tl.arange(0, 32)
    y_arange = tl.arange(0, 128)
    x_arange = tl.arange(0, 128)
    k_accumulator = tl.zeros((128, 128), tl.float32)
    for k_iter in range(0, k_SIZE, 32):
        A = tl.load(A_ptr + (x_block_start + x_arange[:, None]) *
A_x_stride + (k_iter + k_arange[None, :]) * A_k_stride)
        B = tl.load(B_ptr + (k_iter + k_arange[:, None]) * B_k_stride +
(y_block_start + y_arange[None, :]) * B_y_stride)
        k_accumulator = tl.dot(A, B, k_accumulator)
        k_reduction_result = k_accumulator
        mm = k_reduction_result
        tl.store(mm_ptr + (x_block_start + x_arange[:, None]) * mm_x_stride +
(y_block_start + y_arange[None, :]) * mm_y_stride, mm)

    def mm(B, A, y, x, k):
        A_x_stride, A_k_stride, = A.stride()
        B_k_stride, B_y_stride, = B.stride()
        mm = torch.empty(x, y, dtype=torch.float32, device='cuda')
        mm_x_stride, mm_y_stride, = mm.stride()
        mm_grid = (triton.cdiv(x, 128) * triton.cdiv(y, 128)),
        mm_kernel[mm_grid](A, A_x_stride, A_k_stride, B, B_k_stride,
B_y_stride, mm, mm_x_stride, mm_y_stride, y, x, k, num_stages=3,
num_warps=4)
        return mm
```

# Decoupled Triton example

```
# Declarations
Func mm;
In A, B;
Var x, y;
RVar k;

# Algorithm
mm[x, y] = rdot(A[x, k], B[k, y], k);

# Schedule
mm.tensorize(x:128, k:32, y:128);
mm.block(x:128, y:128);
mm.map(x:x1/8, y, x1);
mm.num_warps(4);
mm.num_stages(3);
mm.compile();
```

Modular algorithm and schedule!

```
@triton.jit
def mm_kernel(A_ptr, A_x_stride: tl.constexpr, A_k_stride:
tl.constexpr, B_ptr, B_k_stride: tl.constexpr, B_y_stride:
tl.constexpr, mm_ptr, mm_x_stride: tl.constexpr, mm_y_stride:
tl.constexpr, y_SIZE: tl.constexpr, x_SIZE: tl.constexpr, k_SIZE:
tl.constexpr):
    y_BLOCK_COUNT = y_SIZE // 128
    x_BLOCK_COUNT = x_SIZE // 128
    x_pid = tl.program_id(0).to(tl.int64) // (y_BLOCK_COUNT * 8) %
(x_BLOCK_COUNT // 8)
    y_pid = tl.program_id(0).to(tl.int64) // 8 % y_BLOCK_COUNT
    x1_pid = tl.program_id(0).to(tl.int64) % 8
    x_pid_ = x_pid * 8 + x1_pid
    y_block_start = y_pid * 128
    x_block_start = x_pid_ * 128
    k_arange = tl.arange(0, 32)
    y_arange = tl.arange(0, 128)
    x_arange = tl.arange(0, 128)
    k_accumulator = tl.zeros((128, 128), tl.float32)
    for k_iter in range(0, k_SIZE, 32):
        A = tl.load(A_ptr + (x_block_start + x_arange[:, None]) *
A_x_stride + (k_iter + k_arange[None, :]) * A_k_stride)
        B = tl.load(B_ptr + (k_iter + k_arange[:, None]) * B_k_stride +
(y_block_start + y_arange[None, :]) * B_y_stride)
        k_accumulator = tl.dot(A, B, k_accumulator)
        k_reduction_result = k_accumulator
        mm = k_reduction_result
        tl.store(mm_ptr + (x_block_start + x_arange[:, None]) * mm_x_stride +
(y_block_start + y_arange[None, :]) * mm_y_stride, mm)

def mm(B, A, y, x, k):
    A_x_stride, A_k_stride, = A.stride()
    B_k_stride, B_y_stride, = B.stride()
    mm = torch.empty(y, x, dtype=torch.float32, device='cuda')
    mm_x_stride, mm_y_stride, = mm.stride()
    mm_grid = (triton.cdiv(x, 128) * triton.cdiv(y, 128)),
    mm_kernel[mm_grid](A, A_x_stride, A_k_stride, B, B_k_stride,
B_y_stride, mm, mm_x_stride, mm_y_stride, y, x, k, num_stages=3,
num_warps=4)
    return mm
```

# Decoupled Triton example

```
# Declarations
Func mm;
In A, B;
Var x, y;
RVar k;

# Algorithm
mm[x, y] = rdot(A[x, k], B[k, y], k);

# Schedule
mm.tensorize(x:128, k:32, y:128);
mm.block(x:128, y:128);
mm.map(x:x1/8, y, x1);
mm.num_warps(4);
mm.num_stages(3);
mm.compile();
```

Modular algorithm and schedule!  
Rapidly iterate and explore  
schedules!

```
@triton.jit
def mm_kernel(A_ptr, A_x_stride: tl.constexpr, A_k_stride:
tl.constexpr, B_ptr, B_k_stride: tl.constexpr, B_y_stride:
tl.constexpr, mm_ptr, mm_x_stride: tl.constexpr, mm_y_stride:
tl.constexpr, y_SIZE: tl.constexpr, x_SIZE: tl.constexpr, k_SIZE:
tl.constexpr):
    y_BLOCK_COUNT = y_SIZE // 128
    x_BLOCK_COUNT = x_SIZE // 128
    x_pid = tl.program_id(0).to(tl.int64) // (y_BLOCK_COUNT * 8) %
(x_BLOCK_COUNT // 8)
    y_pid = tl.program_id(0).to(tl.int64) // 8 % y_BLOCK_COUNT
    x1_pid = tl.program_id(0).to(tl.int64) % 8
    x_pid_ = x_pid * 8 + x1_pid
    y_block_start = y_pid * 128
    x_block_start = x_pid_ * 128
    k_arange = tl.arange(0, 32)
    y_arange = tl.arange(0, 128)
    x_arange = tl.arange(0, 128)
    k_accumulator = tl.zeros((128, 128), tl.float32)
    for k_iter in range(0, k_SIZE, 32):
        A = tl.load(A_ptr + (x_block_start + x_arange[:, None]) *
A_x_stride + (k_iter + k_arange[None, :]) * A_k_stride)
        B = tl.load(B_ptr + (k_iter + k_arange[:, None]) * B_k_stride +
(y_block_start + y_arange[None, :]) * B_y_stride)
        k_accumulator = tl.dot(A, B, k_accumulator)
        k_reduction_result = k_accumulator
        mm = k_reduction_result
        tl.store(mm_ptr + (x_block_start + x_arange[:, None]) * mm_x_stride +
(y_block_start + y_arange[None, :]) * mm_y_stride, mm)

def mm(B, A, y, x, k):
    A_x_stride, A_k_stride, = A.stride()
    B_k_stride, B_y_stride, = B.stride()
    mm = torch.empty(y, x, dtype=torch.float32, device='cuda')
    mm_x_stride, mm_y_stride, = mm.stride()
    mm_grid = (triton.cdiv(x, 128) * triton.cdiv(y, 128)),
    mm_kernel[mm_grid](A, A_x_stride, A_k_stride, B, B_k_stride,
B_y_stride, mm, mm_x_stride, mm_y_stride, y, x, k, num_stages=3,
num_warps=4)
    return mm
```

# Why decouple the algorithm from the schedule?

```
# Declarations
Func mm;
In A, B;
Var x, y;
RVar k;

# Algorithm
mm[x, y] = rdot(A[x, k], B[k, y], k);

# Schedule
mm.tensorize(x:128, k:32, y:128);
mm.block(x:128, y:128);
mm.map(x:x1/8, y, x1);
mm.num_warps(4);
mm.num_stages(3);
mm.compile();
```

# Why decouple the algorithm from the schedule?

```
# Declarations
Func mm;
In A, B;
Var x, y;
RVar k;

# Algorithm
mm[x, y] = rdot(A[x, k], B[k, y], k);

# Schedule
mm.tensorize(x:128, k:32, y:128);
mm.block(x:128, y:128);
mm.map(x:x1/8, y, x1);
mm.num_warps(4);
mm.num_stages(3);
mm.compile();
```

- More readable kernel definitions.

# Why decouple the algorithm from the schedule?

```
# Declarations
Func mm;
In A, B;
Var x, y;
RVar k;

# Algorithm
mm[x, y] = rdot(A[x, k], B[k, y], k);

# Schedule
mm.tensorize(x:128, k:32, y:128);
mm.block(x:128, y:128);
mm.map(x:x1/8, y, x1);
mm.num_warps(4);
mm.num_stages(3);
mm.compile();
```

- More readable kernel definitions.
- Modular algorithm and schedule definition.

# Why decouple the algorithm from the schedule?

```
# Declarations
Func mm;
In A, B;
Var x, y;
RVar k;

# Algorithm
mm[x, y] = rdot(A[x, k], B[k, y], k);

# Schedule
mm.tensorize(x:128, k:32, y:128);
mm.block(x:128, y:128);
mm.map(x:x1/8, y, x1);
mm.num_warps(4);
mm.num_stages(3);
mm.compile();
```

- More readable kernel definitions.
- Modular algorithm and schedule definition.
  - Enables rapid kernel schedule iteration and exploration

# Why decouple the algorithm from the schedule?

```
# Declarations
Func mm;
In A, B;
Var x, y;
RVar k;

# Algorithm
mm[x, y] = rdot(A[x, k], B[k, y], k);

# Schedule
mm.tensorize(x:128, k:32, y:128);
mm.block(x:128, y:128);
mm.map(x:x1/8, y, x1);
mm.num_warps(4);
mm.num_stages(3);
mm.compile();
```

- More readable kernel definitions.
- Modular algorithm and schedule definition.
  - Enables rapid kernel schedule iteration and exploration
- Allows users to explicitly define their own kernel scheduling (user-schedulable-language).

# Why decouple the algorithm from the schedule?

```
# Declarations
Func mm;
In A, B;
Var x, y;
RVar k;

# Algorithm
mm[x, y] = rdot(A[x, k], B[k, y], k);

# Schedule
mm.tensorize(x:128, k:32, y:128);
mm.block(x:128, y:128);
mm.map(x:x1/8, y, x1);
mm.num_warps(4);
mm.num_stages(3);
mm.compile();
```

- More readable kernel definitions.
- Modular algorithm and schedule definition.
  - Enables rapid kernel schedule iteration and exploration
- Allows users to explicitly define their own kernel scheduling (user-schedulable-language).
  - PyTorch is not user-schedulable

# Why decouple the algorithm from the schedule?

```
# Declarations
Func mm;
In A, B;
Var x, y;
RVar k;

# Algorithm
mm[x, y] = rdot(A[x, k], B[k, y], k);

# Schedule
mm.tensorize(x:128, k:32, y:128);
mm.block(x:128, y:128);
mm.map(x:x1/8, y, x1);
mm.num_warps(4);
mm.num_stages(3);
mm.compile();
```

- More readable kernel definitions.
- Modular algorithm and schedule definition.
  - Enables rapid kernel schedule iteration and exploration
- Allows users to explicitly define their own kernel scheduling (user-schedulable-language).
  - PyTorch is not user-schedulable



# Decoupled Triton

# Decoupled Triton

An abstraction layer on top of Triton that decouples  
the algorithm from the schedule.

# Decoupled Triton

# Decoupled Triton

**DT File**

# Decoupled Triton

**DT File**

**DT Compiler**

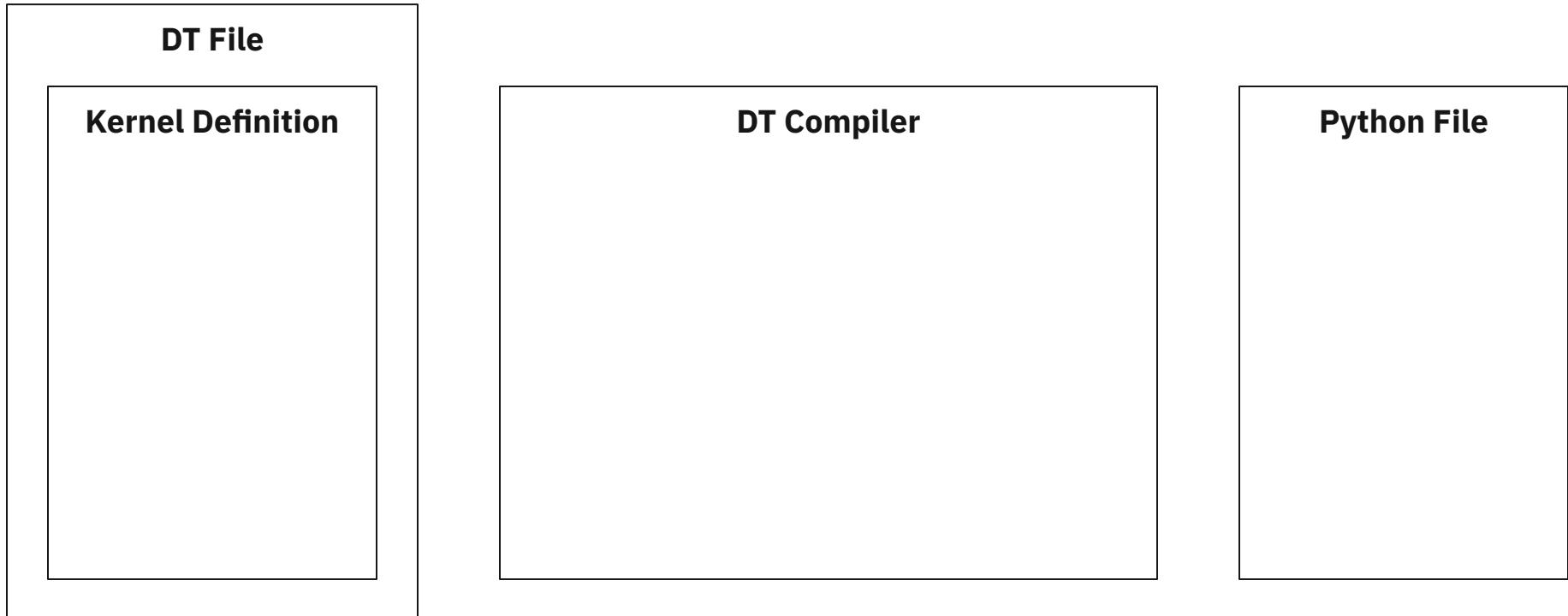
# Decoupled Triton

**DT File**

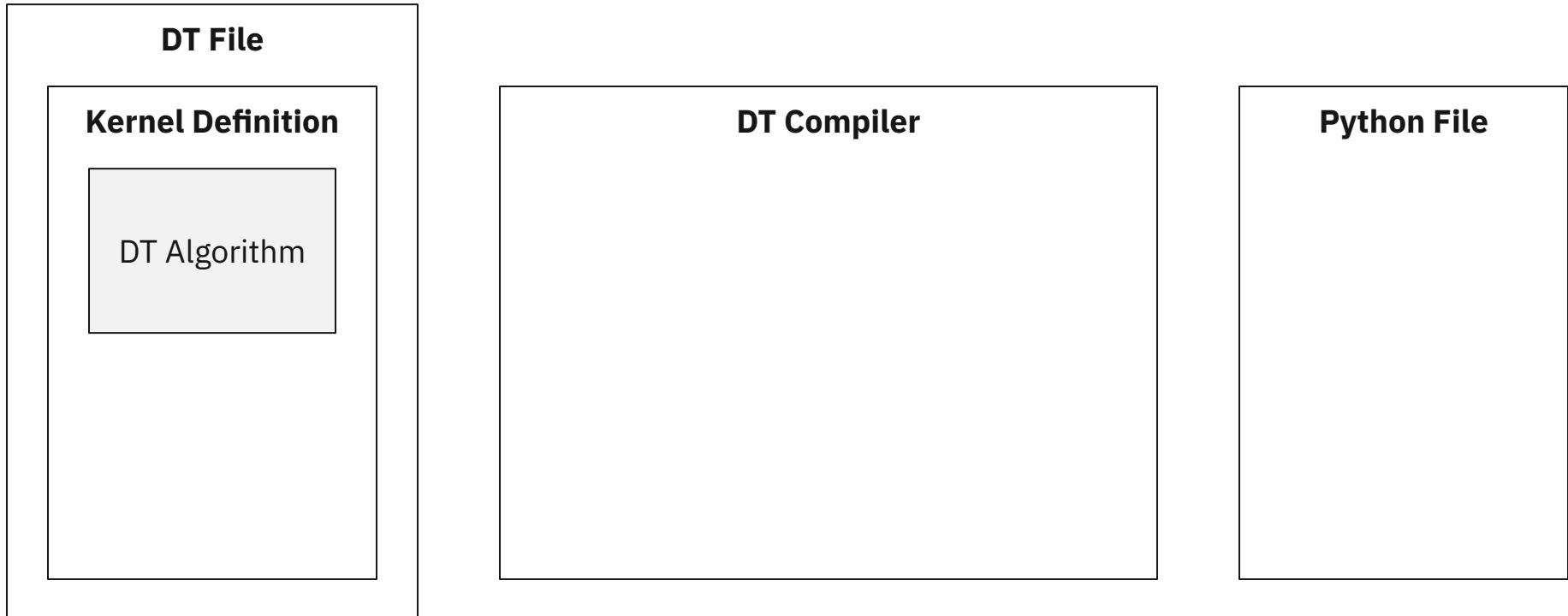
**DT Compiler**

**Python File**

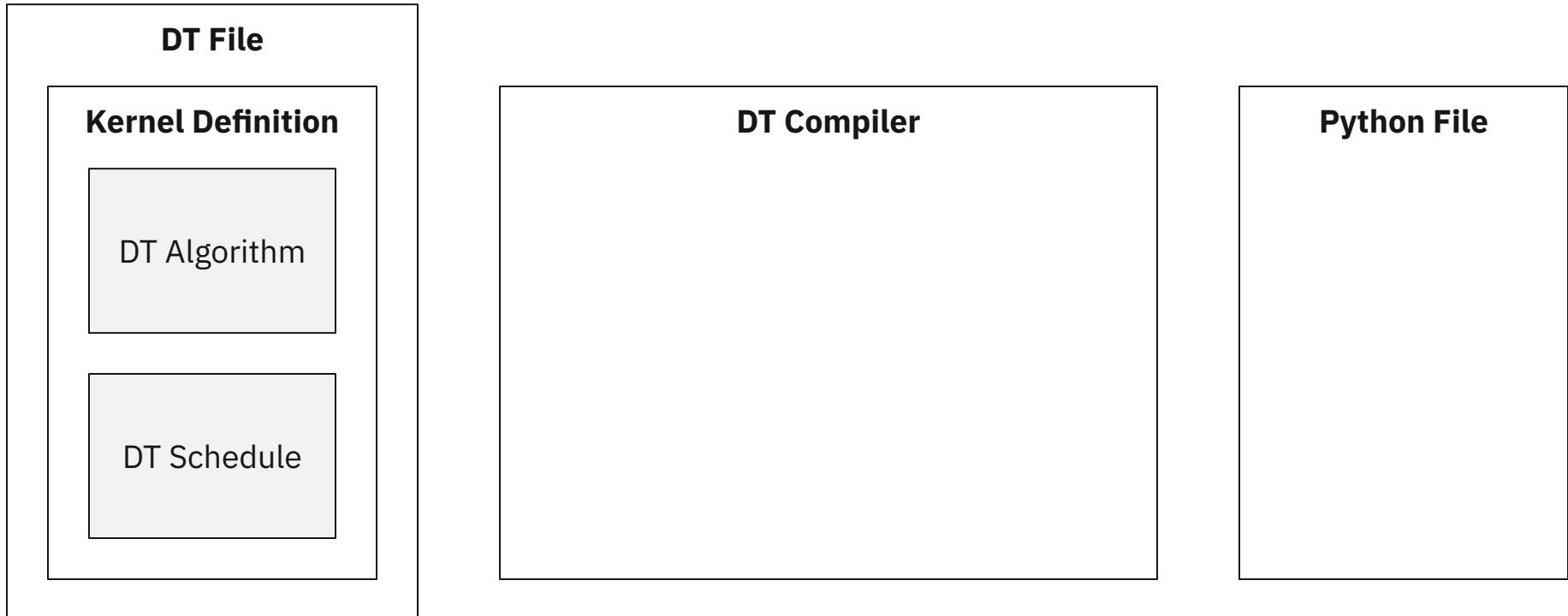
# Decoupled Triton



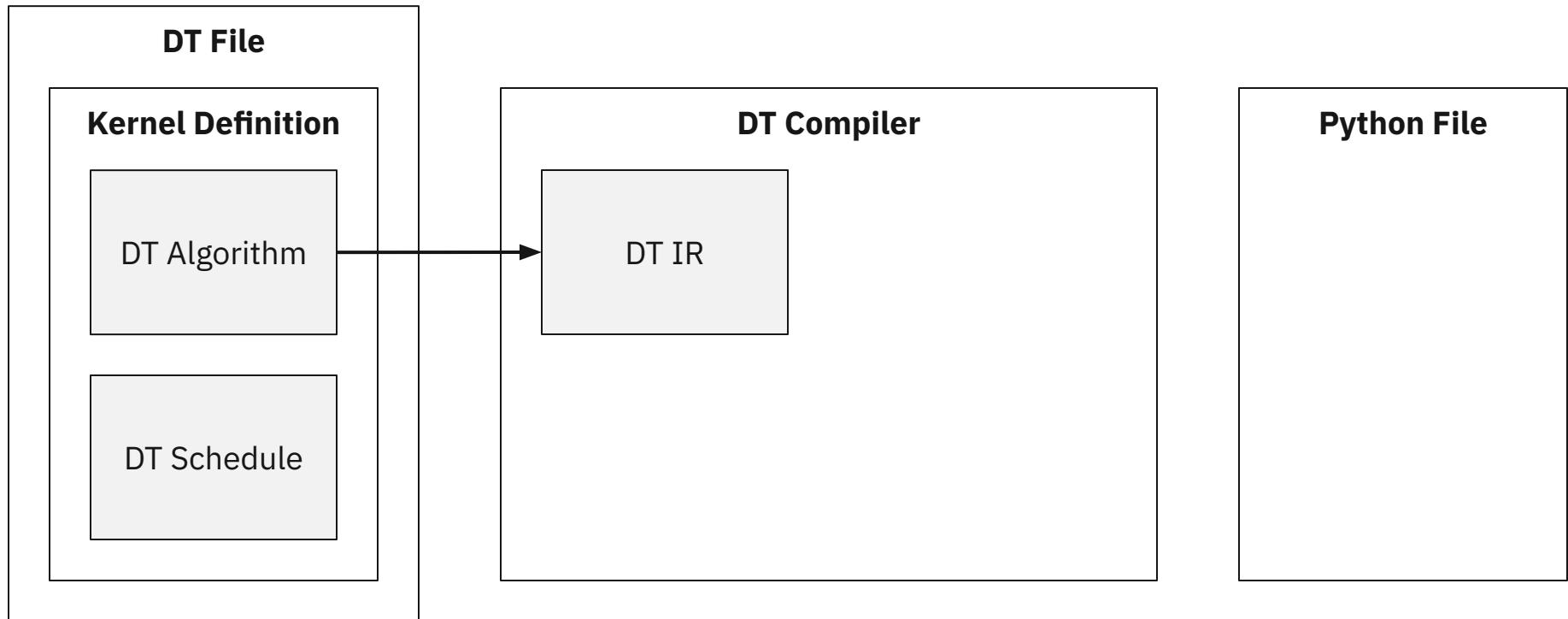
# Decoupled Triton



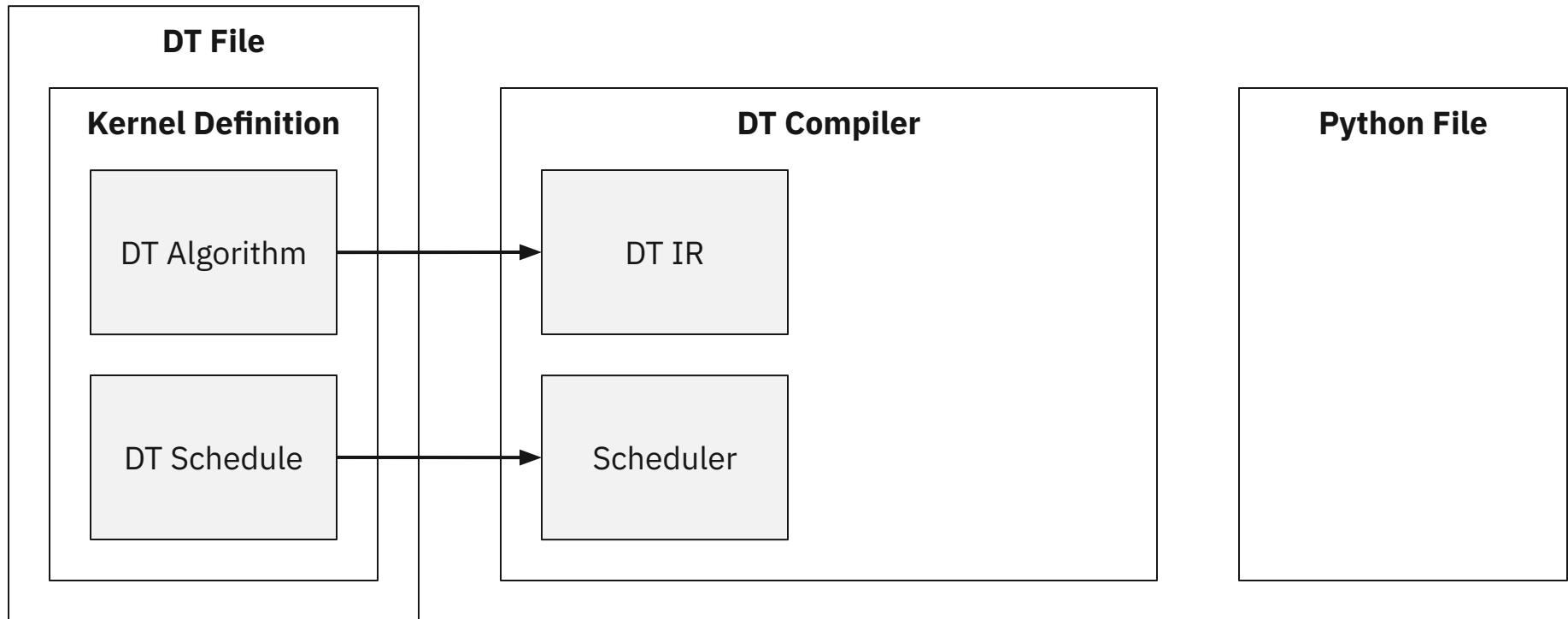
# Decoupled Triton



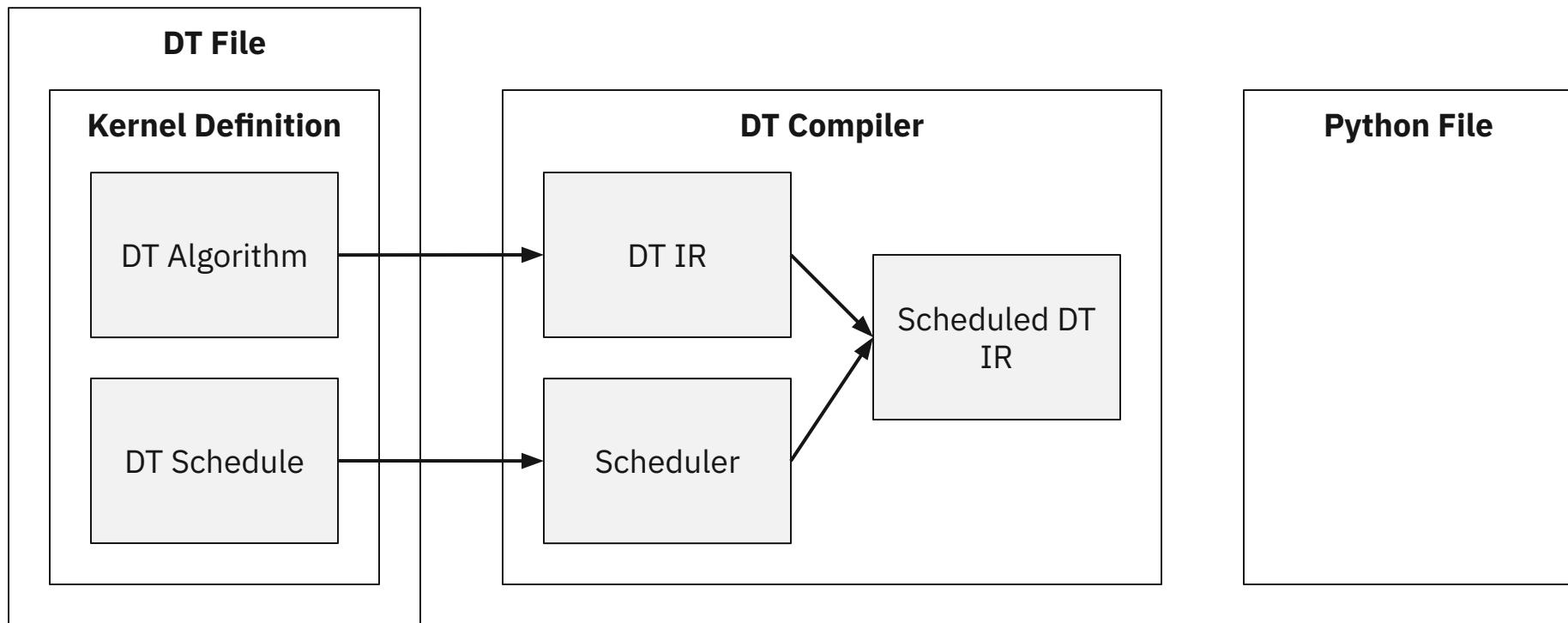
# Decoupled Triton



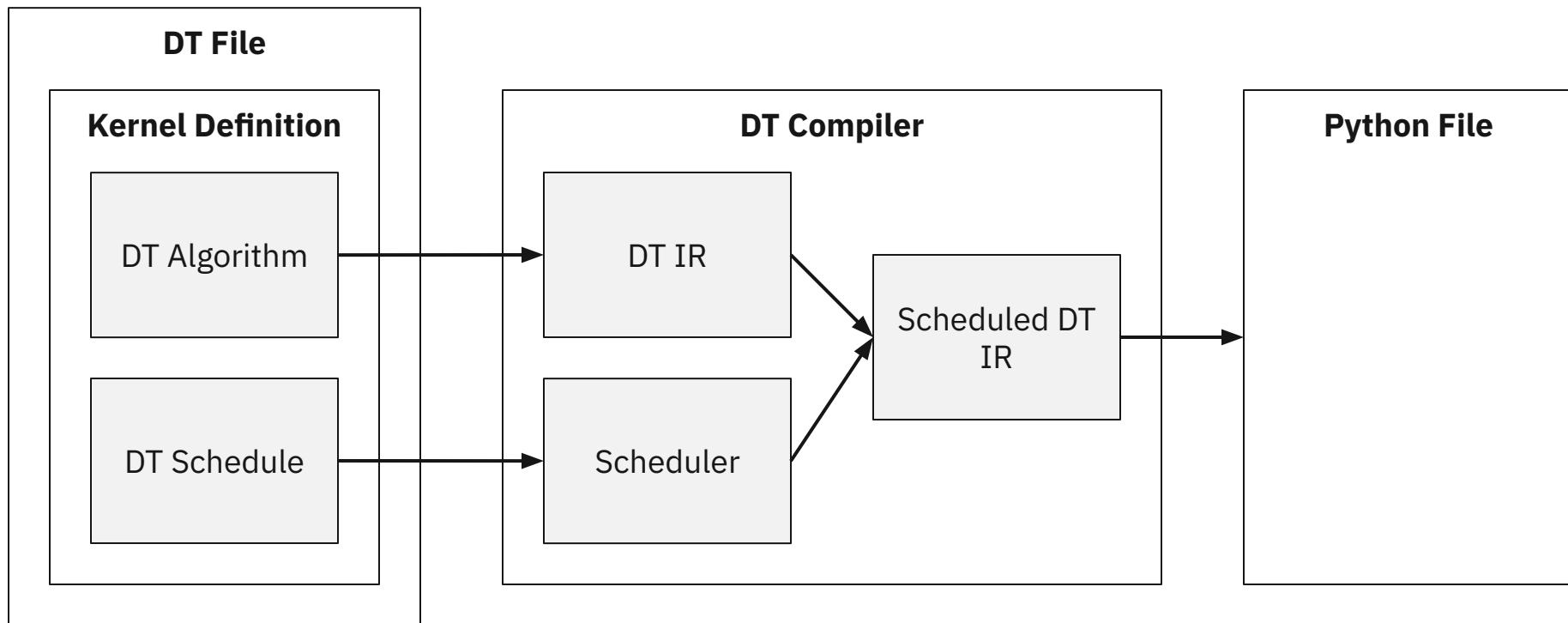
# Decoupled Triton



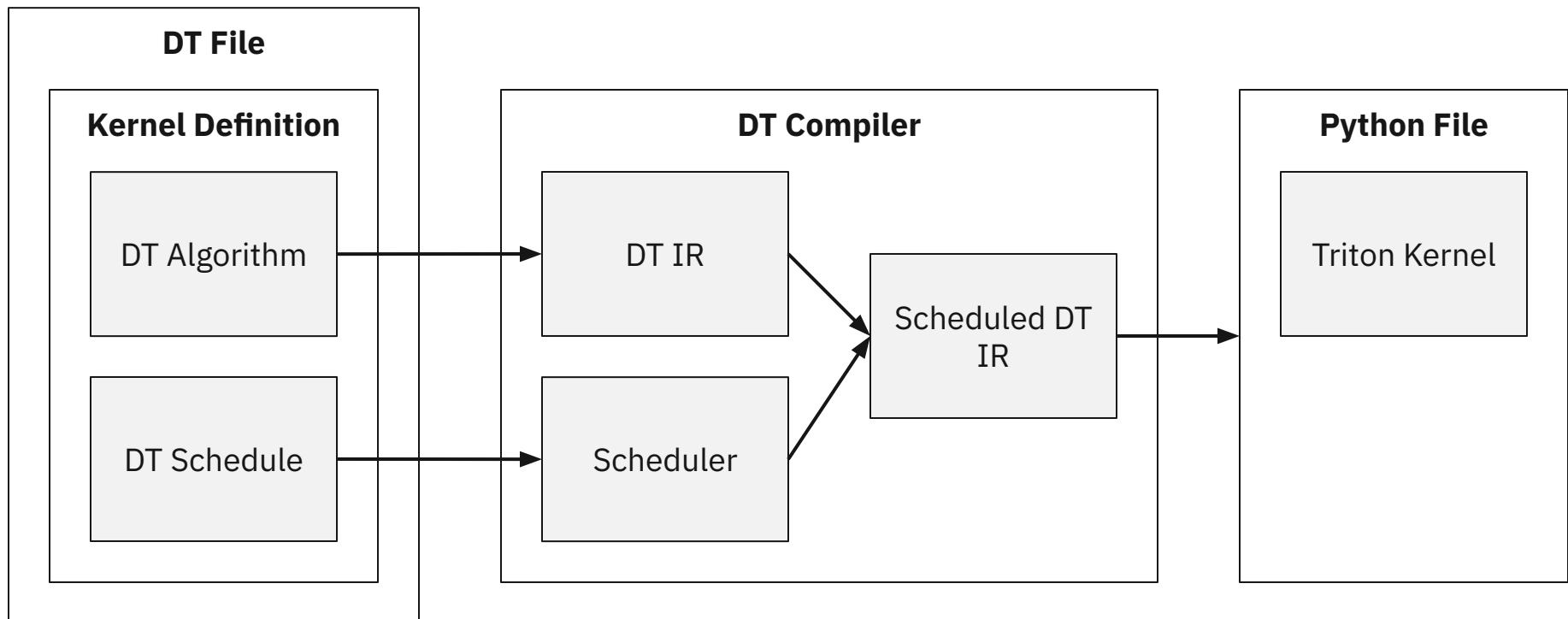
# Decoupled Triton



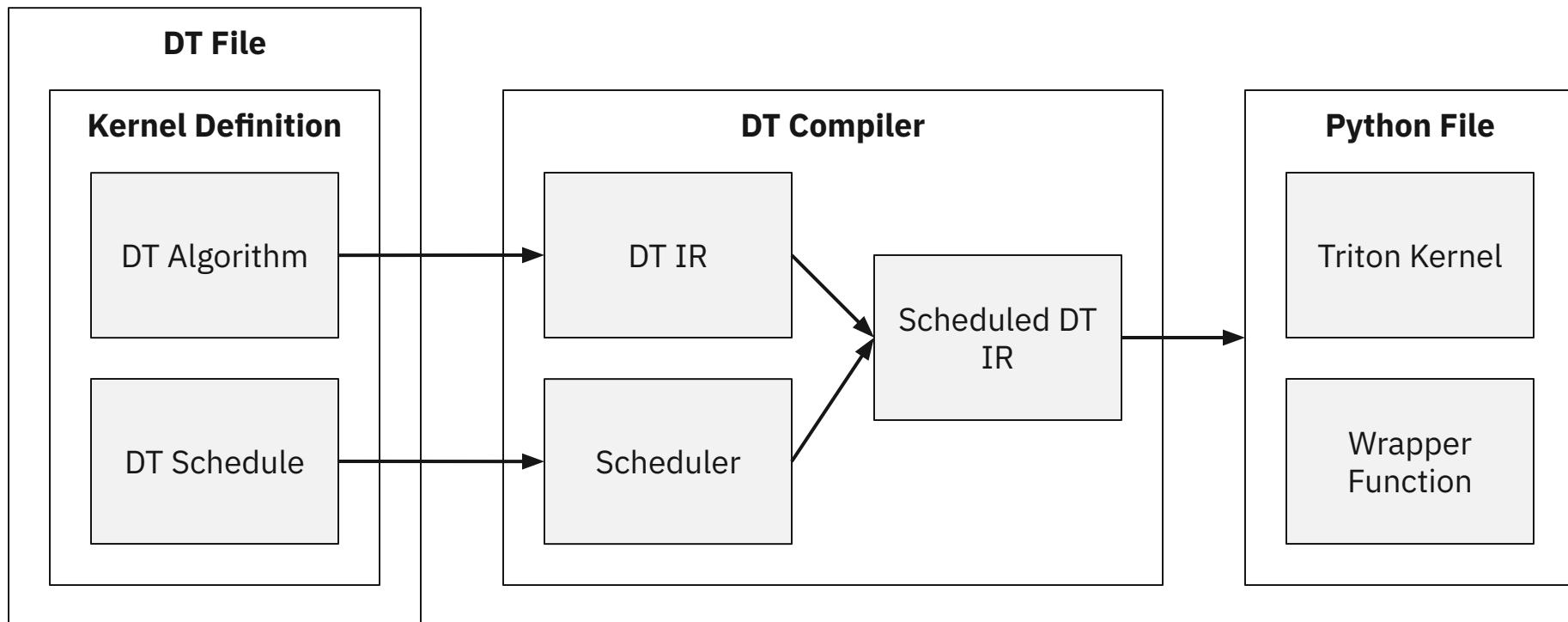
# Decoupled Triton



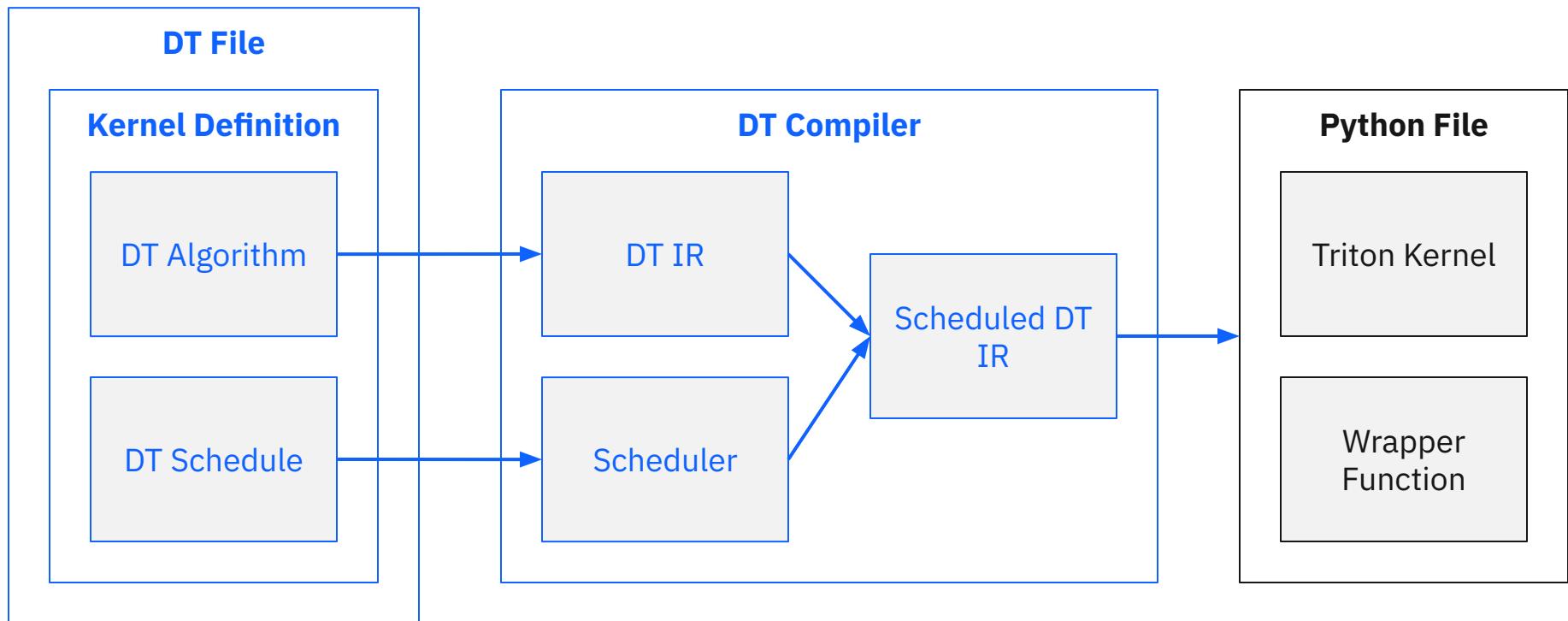
# Decoupled Triton



# Decoupled Triton

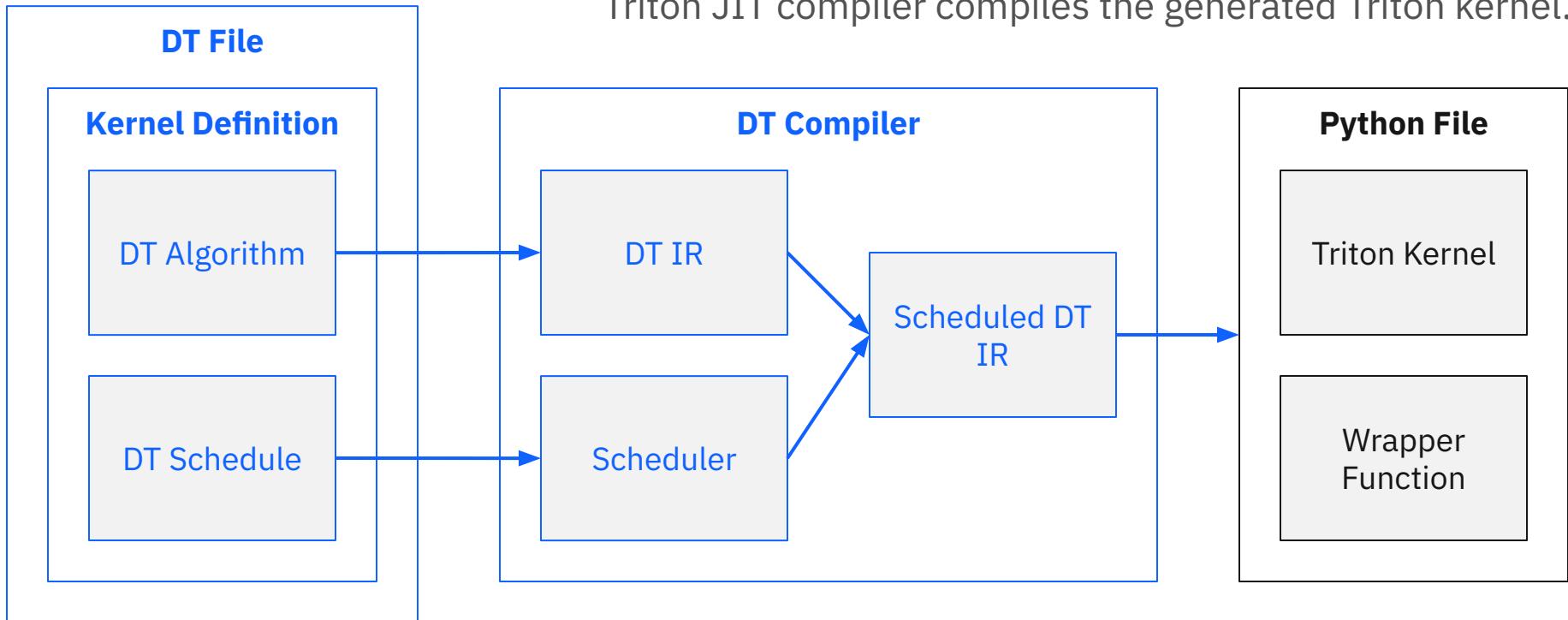


# Decoupled Triton

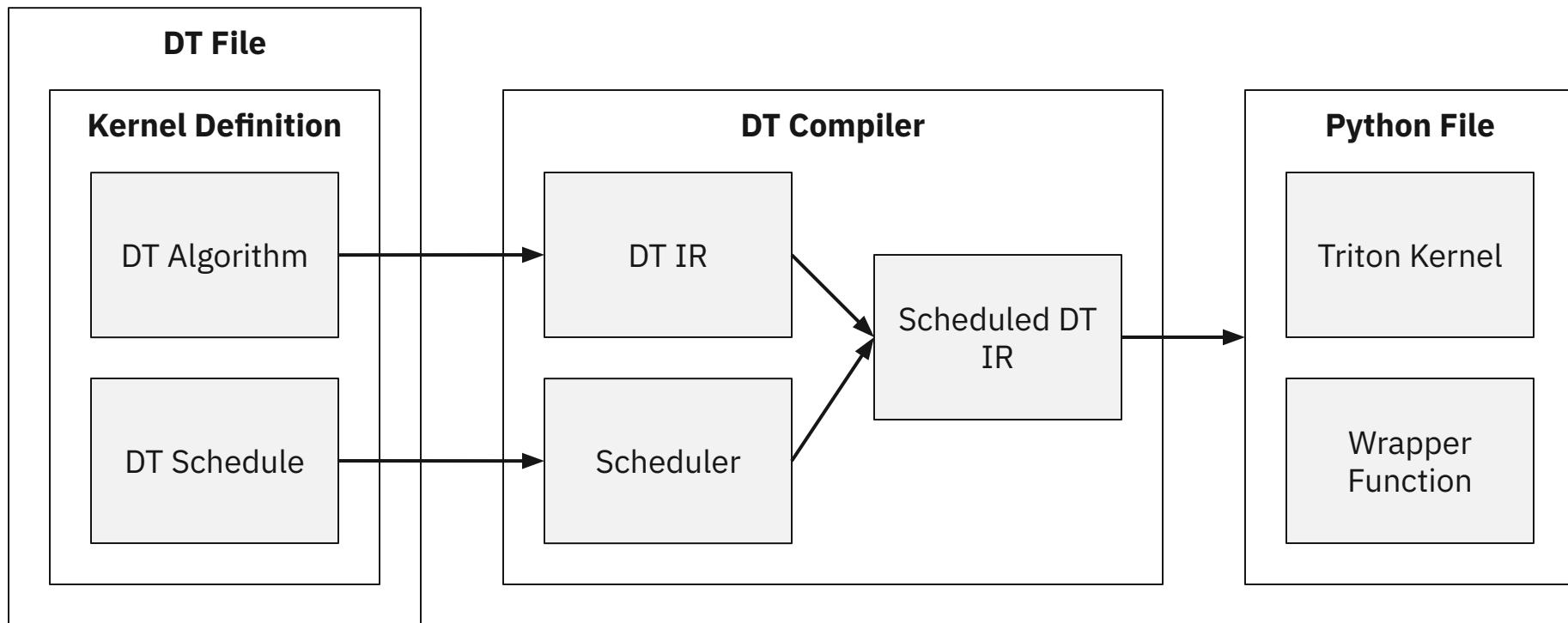


# Decoupled Triton

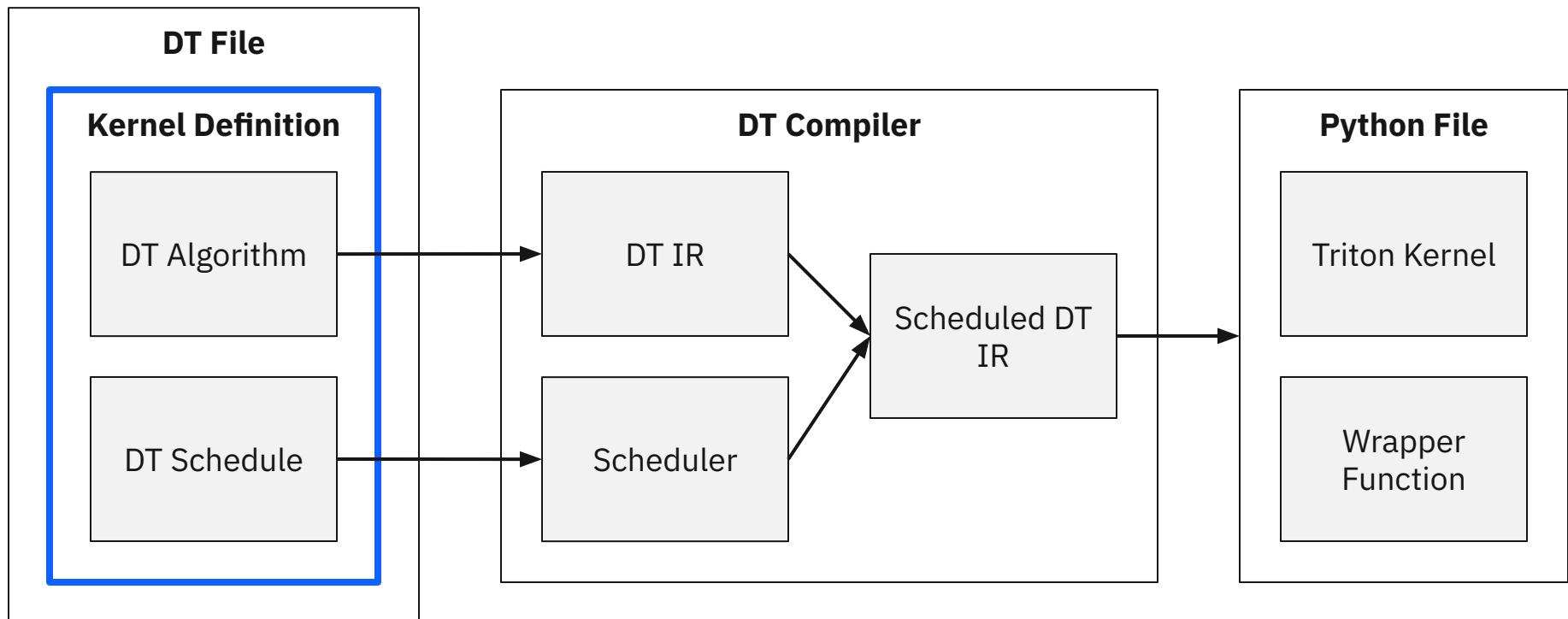
Triton JIT compiler compiles the generated Triton kernel.



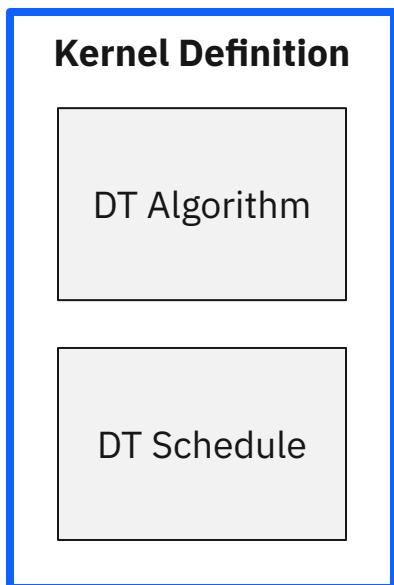
# Decoupled Triton



# Decoupled Triton



# Scaled vector addition kernel definition in DT



# Scaled vector addition kernel definition in DT

## Kernel Definition

DT Algorithm

DT Schedule

```
# Declarations
Func add_out; # Tensor function to be defined
In A, B;      # Input tensors
SIn alpha;    # Input scalar
Var x, y;    # Dimensions labels

# Algorithm
add_out[x, y] = alpha * (A[x, y] + B[x, y]);

# Schedule
add_out.block(x:1, y:256);
add_out.tensorize(y:64);
add_out.compile();
```

# Scaled vector addition kernel definition in DT

## Kernel Definition

DT Algorithm

DT Schedule

```
# Declarations
Func add_out; # Tensor function to be defined
In A, B;      # Input tensors
SIn alpha;    # Input scalar
Var x, y;    # Dimensions labels

# Algorithm
add_out[x, y] = alpha * (A[x, y] + B[x, y]);

# Schedule
add_out.block(x:1, y:256);
add_out.tensorize(y:64);
add_out.compile();
```

# Scaled vector addition kernel definition in DT

## Kernel Definition

DT Algorithm

DT Schedule

```
# Declarations
Func add_out; # Tensor function to be defined
In A, B;      # Input tensors
SIn alpha;    # Input scalar
Var x, y;    # Dimensions labels

# Algorithm
add_out[x, y] = alpha * (A[x, y] + B[x, y]);

# Schedule
add_out.block(x:1, y:256);
add_out.tensorize(y:64);
add_out.compile();
```

# Scaled vector addition kernel definition in DT

## Kernel Definition

DT Algorithm

DT Schedule

```
# Declarations
Func add_out; # Tensor function to be defined
In A, B;      # Input tensors
SIn alpha;    # Input scalar
Var x, y;    # Dimensions labels

# Algorithm
add_out[x, y] = alpha * (A[x, y] + B[x, y]);

# Schedule
add_out.block(x:1, y:256);
add_out.tensorize(y:64);
add_out.compile();
```

# Scaled vector addition kernel definition in DT

## Kernel Definition

DT Algorithm

DT Schedule

```
# Declarations
Func add_out; # Tensor function to be defined
In A, B;      # Input tensors
SIn alpha;    # Input scalar
Var x, y;     # Dimensions labels

# Algorithm
add_out[x, y] = alpha * (A[x, y] + B[x, y]);

# Schedule
add_out.block(x:1, y:256);
add_out.tensorize(y:64);
add_out.compile();
```

# Scaled vector addition kernel definition in DT

## Kernel Definition

DT Algorithm

DT Schedule

```
# Declarations
Func add_out; # Tensor function to be defined
In A, B;      # Input tensors
SIn alpha;    # Input scalar
Var x, y;     # Dimensions labels

# Algorithm
add_out[x, y] = alpha * (A[x, y] + B[x, y]);

# Schedule
add_out.block(x:1, y:256);
add_out.tensorize(y:64);
add_out.compile();
```

# Scaled vector addition kernel definition in DT

## Kernel Definition

DT Algorithm

DT Schedule

```
# Declarations
Func add_out; # Tensor function to be defined
In A, B;      # Input tensors
SIn alpha;    # Input scalar
Var x, y;    # Dimensions labels

# Algorithm
add_out[x, y] = alpha * (A[x, y] + B[x, y]);

# Schedule
add_out.block(x:1, y:256);
add_out.tensorize(y:64);
add_out.compile();
```

# Scaled vector addition kernel definition in DT

## Kernel Definition

DT Algorithm

DT Schedule

```
# Declarations
Func add_out; # Tensor function to be defined
In A, B;      # Input tensors
SIn alpha;    # Input scalar
Var x, y;    # Dimensions labels

# Algorithm
add_out[x, y] = alpha * (A[x, y] + B[x, y]);

# Schedule
add_out.block(x:1, y:256);
add_out.tensorize(y:64);
add_out.compile();
```

# Scaled vector addition kernel definition in DT

## Kernel Definition

DT Algorithm

DT Schedule

```
# Declarations
Func add_out; # Tensor function to be defined
In A, B;      # Input tensors
SIn alpha;    # Input scalar
Var x, y;    # Dimensions labels

# Algorithm
add_out[x, y] = alpha * (A[x, y] + B[x, y]);

# Schedule
add_out.block(x:1, y:256);
add_out.tensorize(y:64);
add_out.compile();
```

# Scaled vector addition kernel definition in DT

## Kernel Definition

DT Algorithm

DT Schedule

```
# Declarations
Func add_out; # Tensor function to be defined
In A, B;      # Input tensors
SIn alpha;    # Input scalar
Var x, y;    # Dimensions labels

# Algorithm
add_out[x, y] = alpha * (A[x, y] + B[x, y]);

# Schedule
add_out.block(x:1, y:256);
add_out.tensorize(y:64);
add_out.compile();
```

# Scaled vector addition kernel definition in DT

## Kernel Definition

DT Algorithm

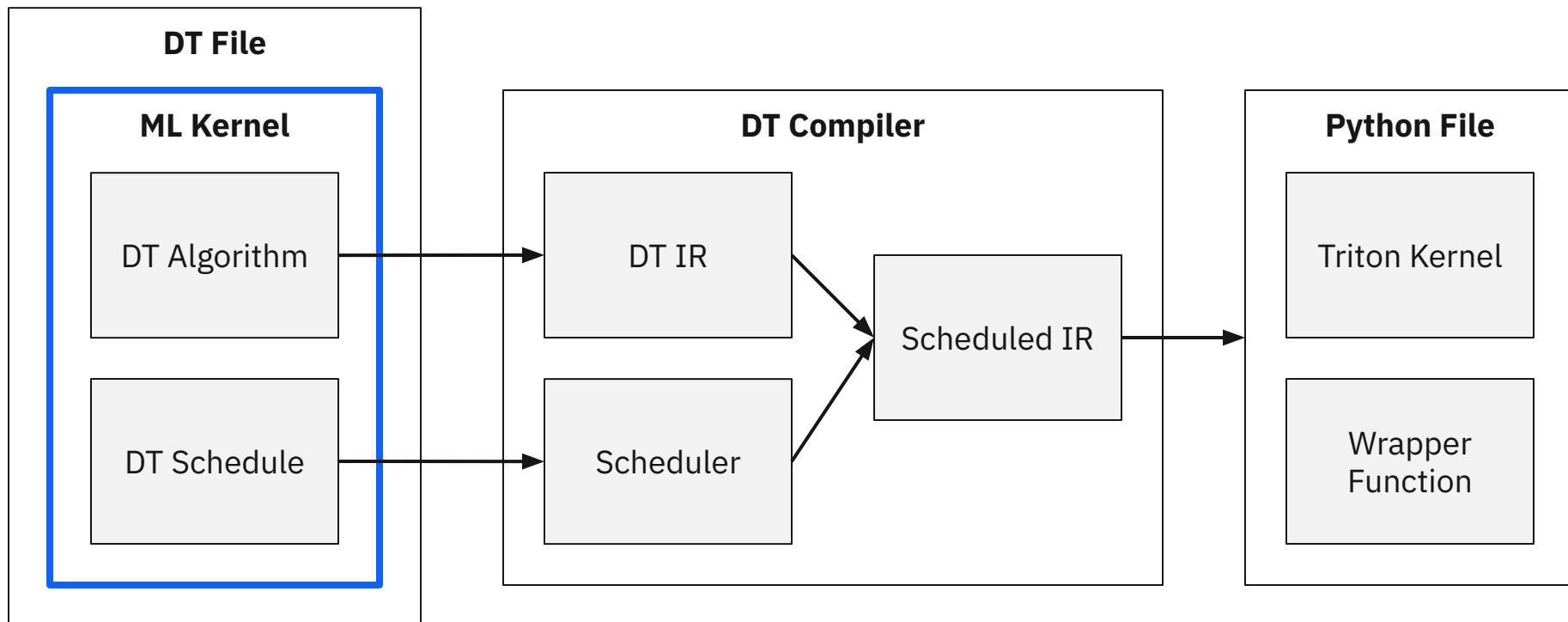
DT Schedule

```
# Declarations
Func add_out; # Tensor function to be defined
In A, B;      # Input tensors
SIn alpha;    # Input scalar
Var x, y;     # Dimensions labels

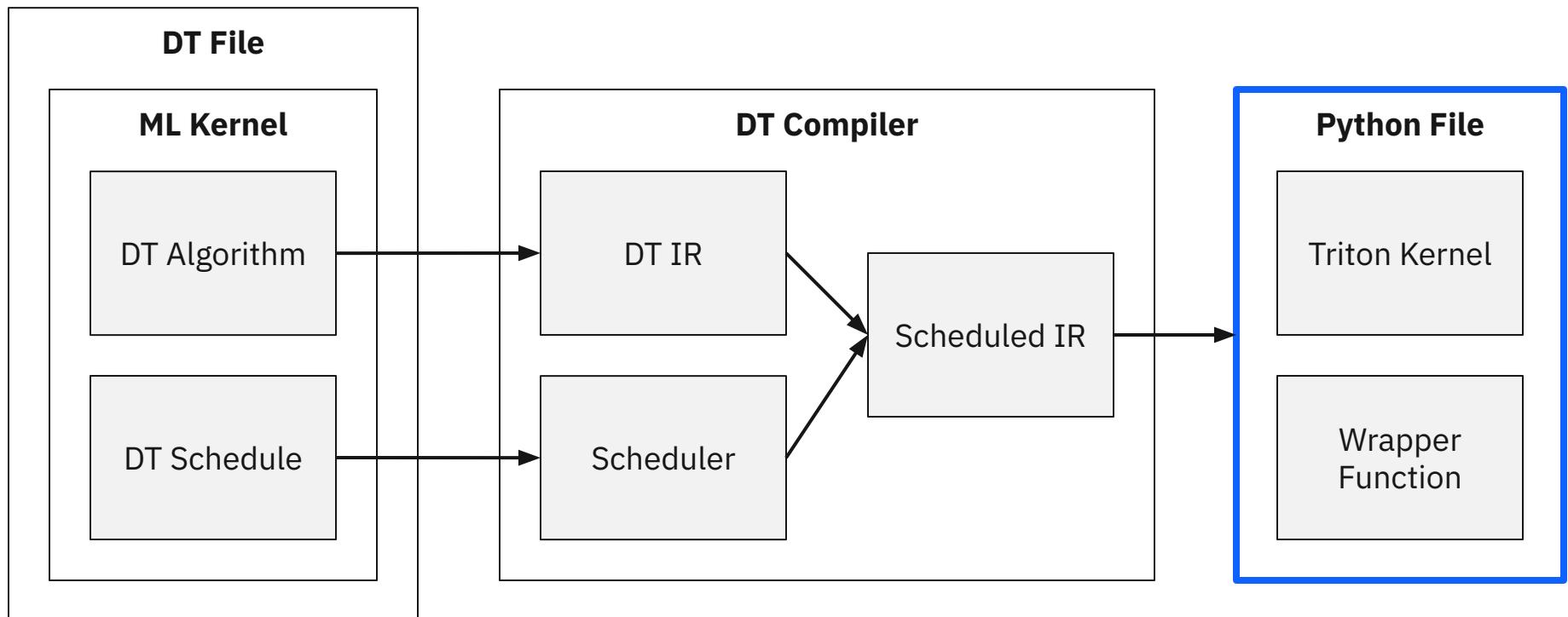
# Algorithm
add_out[x, y] = alpha * (A[x, y] + B[x, y]);

# Schedule
add_out.block(x:1, y:256);
add_out.tensorize(y:64);
add_out.compile();
```

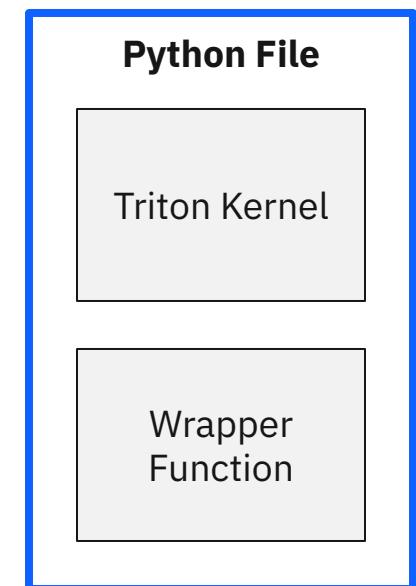
# Decoupled Triton



# Decoupled Triton



# Generated Python file



# Generated Python file

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

@triton.jit
def add_out_kernel(A_ptr, A_x_stride: tl.constexpr, A_y_stride: tl.constexpr, B_ptr,
                    B_x_stride: tl.constexpr, B_y_stride: tl.constexpr, alpha, add_out_ptr,
                    add_out_x_stride: tl.constexpr, add_out_y_stride: tl.constexpr, y_SIZE: tl.constexpr,
                    x_SIZE: tl.constexpr):
    y_BLOCK_COUNT = y_SIZE // 256
    x_BLOCK_COUNT = x_SIZE // 1
    y_pid = tl.program_id(0).to(tl.int64) // x_BLOCK_COUNT % y_BLOCK_COUNT
    x_pid = tl.program_id(0).to(tl.int64) % x_BLOCK_COUNT
    y_block_start = y_pid * 256
    x_block_start = x_pid * 1
    y_arange = tl.arange(0, 64)
    for y_iter in range(0, 256, 64):
        A = tl.load(A_ptr + x_block_start * A_x_stride + (y_block_start + y_iter + y_arange) * A_y_stride)
        B = tl.load(B_ptr + x_block_start * B_x_stride + (y_block_start + y_iter + y_arange) * B_y_stride)
        add_out = alpha * (A + B)
        tl.store(add_out_ptr + x_block_start * add_out_x_stride + (y_block_start + y_iter + y_arange) * add_out_y_stride, add_out)

    def add_out(B, A, alpha, y, x):
        A_x_stride, A_y_stride, = A.stride()
        B_x_stride, B_y_stride, = B.stride()
        add_out = torch.empty(x, y, dtype=torch.float32, device='cuda')
        add_out_x_stride, add_out_y_stride, = add_out.stride()
        add_out_grid = (triton.cdiv(x, 1) * triton.cdiv(y, 256)),
        add_out_kernel[add_out_grid](A, A_x_stride, A_y_stride, B, B_x_stride, B_y_stride, alpha,
                                    add_out, add_out_x_stride, add_out_y_stride, y, x, num_stages=3, num_warps=4)
        return add_out
```

## Python File

Triton Kernel

Wrapper Function

# Generated Python file

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

@triton.jit
def add_out_kernel(A_ptr, A_x_stride: tl.constexpr, A_y_stride: tl.constexpr, B_ptr,
                    B_x_stride: tl.constexpr, B_y_stride: tl.constexpr, alpha, add_out_ptr,
                    add_out_x_stride: tl.constexpr, add_out_y_stride: tl.constexpr, y_SIZE: tl.constexpr,
                    x_SIZE: tl.constexpr):
    y_BLOCK_COUNT = y_SIZE // 256
    x_BLOCK_COUNT = x_SIZE // 1
    y_pid = tl.program_id(0).to(tl.int64) // x_BLOCK_COUNT % y_BLOCK_COUNT
    x_pid = tl.program_id(0).to(tl.int64) % x_BLOCK_COUNT
    y_block_start = y_pid * 256
    x_block_start = x_pid * 1
    y_arange = tl.arange(0, 64)
    for y_iter in range(0, 256, 64):
        A = tl.load(A_ptr + x_block_start * A_x_stride + (y_block_start + y_iter + y_arange) * A_y_stride)
        B = tl.load(B_ptr + x_block_start * B_x_stride + (y_block_start + y_iter + y_arange) * B_y_stride)
        add_out = alpha * (A + B)
        tl.store(add_out_ptr + x_block_start * add_out_x_stride + (y_block_start + y_iter + y_arange) *
                add_out_y_stride, add_out)

def add_out(B, A, alpha, y, x):
    A_x_stride, A_y_stride, = A.stride()
    B_x_stride, B_y_stride, = B.stride()
    add_out = torch.empty(x, y, dtype=torch.float32, device='cuda')
    add_out_x_stride, add_out_y_stride, = add_out.stride()
    add_out_grid = (triton.cdiv(x, 1) * triton.cdiv(y, 256)),
    add_out_kernel[add_out_grid](A, A_x_stride, A_y_stride, B, B_x_stride, B_y_stride, alpha,
                                add_out, add_out_x_stride, add_out_y_stride, y, x, num_stages=3, num_warps=4)
    return add_out
```

## Python File

Triton Kernel

Wrapper Function

# Generated Python file

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

@triton.jit
def add_out_kernel(A_ptr, A_x_stride: tl.constexpr, A_y_stride: tl.constexpr, B_ptr,
                    B_x_stride: tl.constexpr, B_y_stride: tl.constexpr, alpha, add_out_ptr,
                    add_out_x_stride: tl.constexpr, add_out_y_stride: tl.constexpr, y_SIZE: tl.constexpr,
                    x_SIZE: tl.constexpr):
    y_BLOCK_COUNT = y_SIZE // 256
    x_BLOCK_COUNT = x_SIZE // 1
    y_pid = tl.program_id(0).to(tl.int64) // x_BLOCK_COUNT % y_BLOCK_COUNT
    x_pid = tl.program_id(0).to(tl.int64) % x_BLOCK_COUNT
    y_block_start = y_pid * 256
    x_block_start = x_pid * 1
    y_arange = tl.arange(0, 64)
    for y_iter in range(0, 256, 64):
        A = tl.load(A_ptr + x_block_start * A_x_stride + (y_block_start + y_iter + y_arange) * A_y_stride)
        B = tl.load(B_ptr + x_block_start * B_x_stride + (y_block_start + y_iter + y_arange) * B_y_stride)
        add_out = alpha * (A + B)
        tl.store(add_out_ptr + x_block_start * add_out_x_stride + (y_block_start + y_iter + y_arange) * add_out_y_stride, add_out)

def add_out(B, A, alpha, y, x):
    A_x_stride, A_y_stride, = A.stride()
    B_x_stride, B_y_stride, = B.stride()
    add_out = torch.empty(x, y, dtype=torch.float32, device='cuda')
    add_out_x_stride, add_out_y_stride, = add_out.stride()
    add_out_grid = (triton.cdiv(x, 1) * triton.cdiv(y, 256)),
    add_out_kernel[add_out_grid](A, A_x_stride, A_y_stride, B, B_x_stride, B_y_stride, alpha,
                                 add_out, add_out_x_stride, add_out_y_stride, y, x, num_stages=3, num_warps=4)
    return add_out
```

## Python File

Triton Kernel

Wrapper Function

# Generated Python file

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

@triton.jit
def add_out_kernel(A_ptr, A_x_stride: tl.constexpr, A_y_stride: tl.constexpr, B_ptr,
                    B_x_stride: tl.constexpr, B_y_stride: tl.constexpr, alpha, add_out_ptr,
                    add_out_x_stride: tl.constexpr, add_out_y_stride: tl.constexpr, y_SIZE: tl.constexpr,
                    x_SIZE: tl.constexpr):
    y_BLOCK_COUNT = y_SIZE // 256
    x_BLOCK_COUNT = x_SIZE // 1
    y_pid = tl.program_id(0).to(tl.int64) // x_BLOCK_COUNT % y_BLOCK_COUNT
    x_pid = tl.program_id(0).to(tl.int64) % x_BLOCK_COUNT
    y_block_start = y_pid * 256
    x_block_start = x_pid * 1
    y_arange = tl.arange(0, 64)
    for y_iter in range(0, 256, 64):
        A = tl.load(A_ptr + x_block_start * A_x_stride + (y_block_start + y_iter + y_arange) * A_y_stride)
        B = tl.load(B_ptr + x_block_start * B_x_stride + (y_block_start + y_iter + y_arange) * B_y_stride)
        add_out = alpha * (A + B)
        tl.store(add_out_ptr + x_block_start * add_out_x_stride + (y_block_start + y_iter + y_arange) *
add_out_y_stride, add_out)

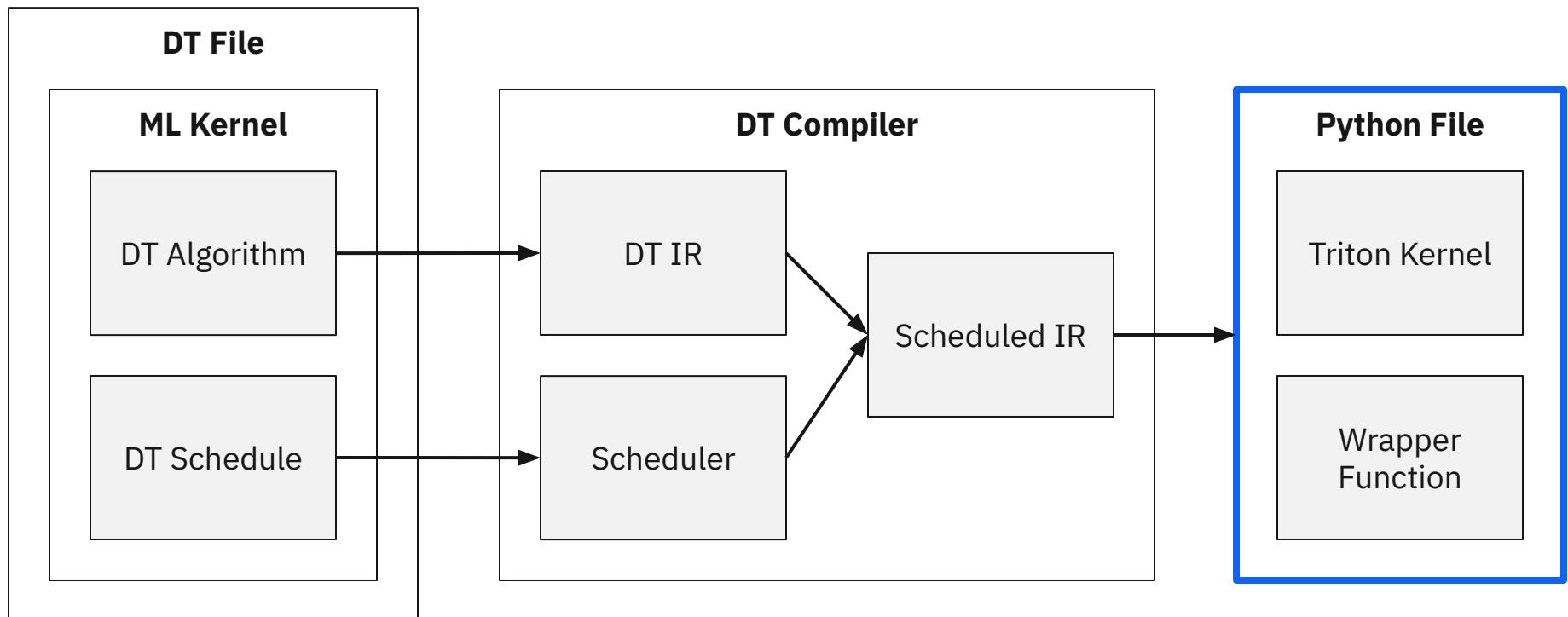
def add_out(B, A, alpha, y, x):
    A_x_stride, A_y_stride, = A.stride()
    B_x_stride, B_y_stride, = B.stride()
    add_out = torch.empty(x, y, dtype=torch.float32, device='cuda')
    add_out_x_stride, add_out_y_stride, = add_out.stride()
    add_out_grid = (triton.cdiv(x, 1) * triton.cdiv(y, 256)),
    add_out_kernel[add_out_grid](A, A_x_stride, A_y_stride, B, B_x_stride, B_y_stride, alpha,
                                add_out, add_out_x_stride, add_out_y_stride, y, x, num_stages=3, num_warps=4)
    return add_out
```

## Python File

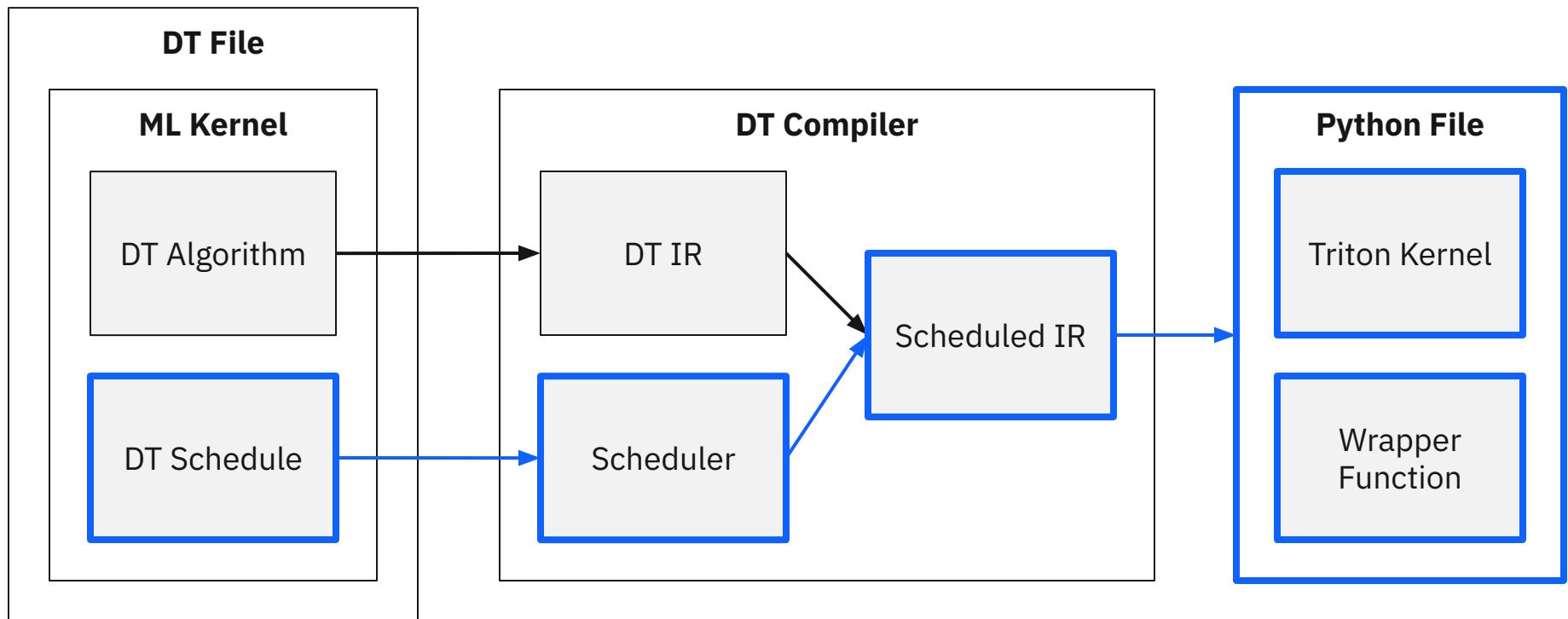
Triton Kernel

Wrapper Function

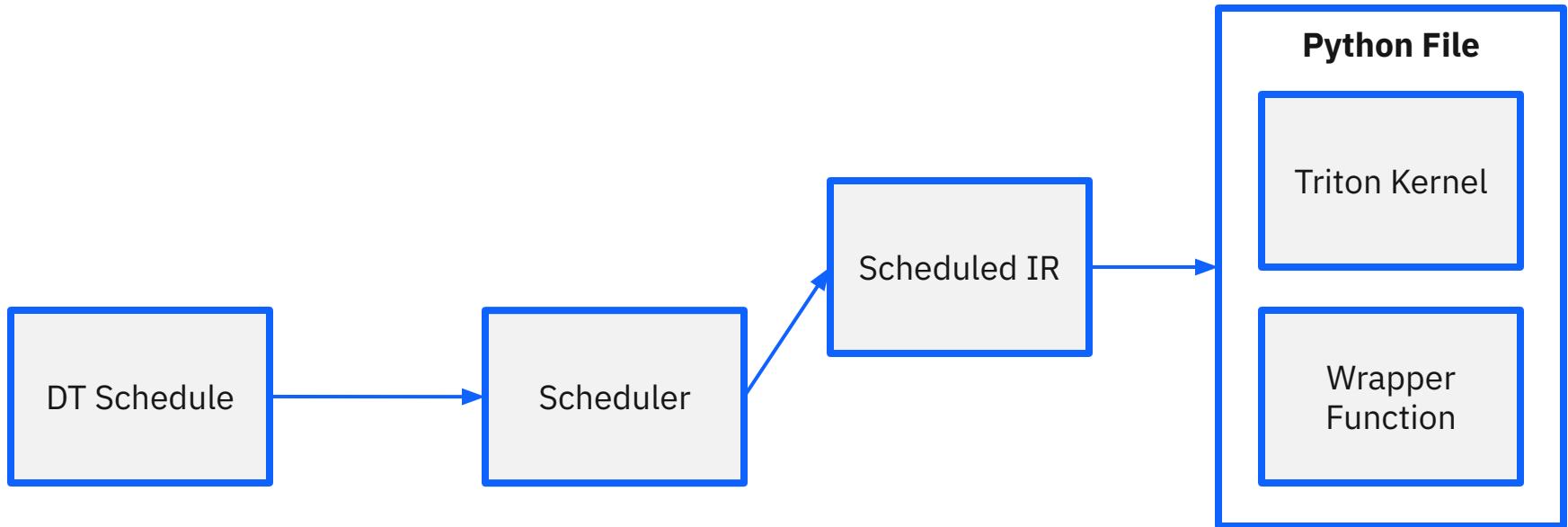
# Decoupled Triton



# Decoupled Triton



# Scheduling



# Scheduling

```
# Schedule  
add_out.block(x:1, y:256);  
add_out.tensorize(y:64);  
add_out.compile();
```



# Scheduling

```
# Schedule  
add_out.block(x:1, y:256);  
add_out.tensorize(y:64);  
add_out.compile();
```



Scheduling primitives transform the kernel's schedule.

# Default schedule

# Default schedule

```
Func add_out;  
In  A, B;  
SIn alpha;  
Var x, y;  
  
add_out[x,y] = alpha * (A[x,y] + B[x,y]);  
  
add_out.compile();
```

# Default schedule

```
Func add_out;
In  A, B;
SIn alpha;
Var x, y;

add_out[x,y] = alpha * (A[x,y] + B[x,y]);

add_out.compile();
```

```
@triton.jit
def add_out_kernel(...):
    for x_iter in range(0, x_SIZE, 1):
        for y_iter in range(0, y_SIZE, 1):
            A = tl.load(A_ptr + x_iter * A_x_stride + y_iter * A_y_stride)
            B = tl.load(B_ptr + x_iter * B_x_stride + y_iter * B_y_stride)
            add_out = alpha * (A + B)
            tl.store(add_out_ptr + x_iter * add_out_x_stride +
                    y_iter * add_out_y_stride, add_out)

def add_out(...):
    ...
    add_out_grid = (1,)
    add_out_kernel[add_out_grid](...)
    ...
```

# Default schedule

```
Func add_out;
In  A, B;
SIn alpha;
Var x, y;

add_out[x,y] = alpha * (A[x,y] + B[x,y]);

add_out.compile();
```

```
@triton.jit
def add_out_kernel(...):
    for x_iter in range(0, x_SIZE, 1):
        for y_iter in range(0, y_SIZE, 1):
            A = tl.load(A_ptr + x_iter * A_x_stride + y_iter * A_y_stride)
            B = tl.load(B_ptr + x_iter * B_x_stride + y_iter * B_y_stride)
            add_out = alpha * (A + B)
            tl.store(add_out_ptr + x_iter * add_out_x_stride +
                    y_iter * add_out_y_stride, add_out)

def add_out(...):
    ...
    add_out_grid = (1,
    add_out_kernel[add_out_grid](...)
    ...
```

# Default schedule

```
Func add_out;
In  A, B;
SIn alpha;
Var x, y;

add_out[x,y] = alpha * (A[x,y] + B[x,y]);

add_out.compile();
```

```
@triton.jit
def add_out_kernel(...):
    for x_iter in range(0, x_SIZE, 1):
        for y_iter in range(0, y_SIZE, 1):
            A = tl.load(A_ptr + x_iter * A_x_stride + y_iter * A_y_stride)
            B = tl.load(B_ptr + x_iter * B_x_stride + y_iter * B_y_stride)
            add_out = alpha * (A + B)
            tl.store(add_out_ptr + x_iter * add_out_x_stride +
                    y_iter * add_out_y_stride, add_out)

def add_out(...):
    ...
    add_out_grid = (1,)
    add_out_kernel[add_out_grid](...)
    ...
```

# Default schedule

```
Func add_out;
In  A, B;
SIn alpha;
Var x, y;

add_out[x,y] = alpha * (A[x,y] + B[x,y]);

add_out.compile();
```

```
@triton.jit
def add_out_kernel(...):
    for x_iter in range(0, x_SIZE, 1):
        for y_iter in range(0, y_SIZE, 1):
            A = tl.load(A_ptr + x_iter * A_x_stride + y_iter * A_y_stride)
            B = tl.load(B_ptr + x_iter * B_x_stride + y_iter * B_y_stride)
            add_out = alpha * (A + B)
            tl.store(add_out_ptr + x_iter * add_out_x_stride +
                    y_iter * add_out_y_stride, add_out)

def add_out(...):
    ...
    add_out_grid = (1,)
    add_out_kernel[add_out_grid](...)
    ...
```

# Default schedule

```
Func add_out;
In  A, B;
SIn alpha;
Var x, y;

add_out[x,y] = alpha * (A[x,y] + B[x,y]);

add_out.compile();
```

```
@triton.jit
def add_out_kernel(...):
    for x_iter in range(0, x_SIZE, 1):
        for y_iter in range(0, y_SIZE, 1):
            A = tl.load(A_ptr + x_iter * A_x_stride + y_iter * A_y_stride)
            B = tl.load(B_ptr + x_iter * B_x_stride + y_iter * B_y_stride)
            add_out = alpha * (A + B)
            tl.store(add_out_ptr + x_iter * add_out_x_stride +
                    y_iter * add_out_y_stride, add_out)

def add_out(...):
    ...
    add_out_grid = (1,)
    add_out_kernel[add_out_grid](...)
    ...
```

# Default schedule

```
Func add_out;
In  A, B;
SIn alpha;
Var x, y;

add_out[x,y] = alpha * (A[x,y] + B[x,y]);

add_out.compile();
```

```
@triton.jit
def add_out_kernel(...):
    for x_iter in range(0, x_SIZE, 1):
        for y_iter in range(0, y_SIZE, 1):
            A = tl.load(A_ptr + x_iter * A_x_stride + y_iter * A_y_stride)
            B = tl.load(B_ptr + x_iter * B_x_stride + y_iter * B_y_stride)
            add_out = alpha * (A + B)
            tl.store(add_out_ptr + x_iter * add_out_x_stride +
                    y_iter * add_out_y_stride, add_out)

def add_out(...):
    ...
    add_out_grid = (1,)
    add_out_kernel[add_out_grid](...)
    ...
```

# Default schedule

```
Func add_out;
In  A, B;
SIn alpha;
Var x, y;

add_out[x,y] = alpha * (A[x,y] + B[x,y]);

add_out.compile();
```

```
@triton.jit
def add_out_kernel(...):
    for x_iter in range(0, x_SIZE, 1):
        for y_iter in range(0, y_SIZE, 1):
            A = tl.load(A_ptr + x_iter * A_x_stride + y_iter * A_y_stride)
            B = tl.load(B_ptr + x_iter * B_x_stride + y_iter * B_y_stride)
            add_out = alpha * (A + B)
            tl.store(add_out_ptr + x_iter * add_out_x_stride +
                    y_iter * add_out_y_stride, add_out)

def add_out(...):
    ...
    add_out_grid = (1,
    add_out_kernel[add_out_grid](...)
    ...
```

# Block

# Block

- Partitions output tensor into blocks.

# Block

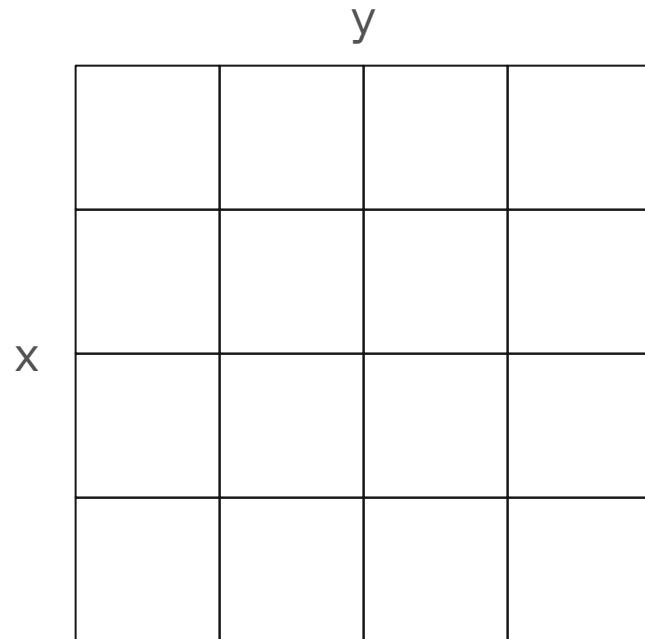
- Partitions output tensor into blocks.
  - Each block is computed by a separate program instance in the kernel launch grid

# Block

- Partitions output tensor into blocks.
  - Each block is computed by a separate program instance in the kernel launch grid


# Block

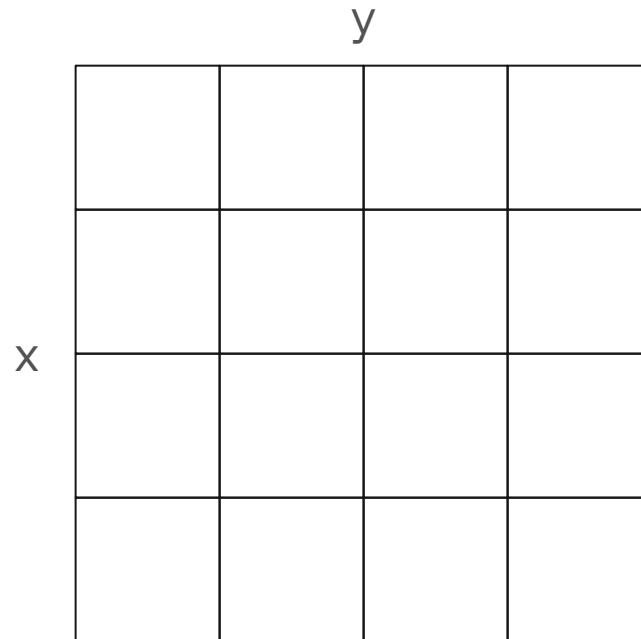
- Partitions output tensor into blocks.
  - Each block is computed by a separate program instance in the kernel launch grid



# Block

- Partitions output tensor into blocks.
  - Each block is computed by a separate program instance in the kernel launch grid

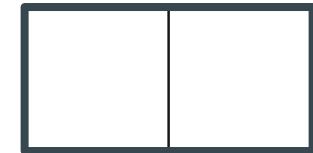
`out.block(x:1, y:2)`



# Block

- Partitions output tensor into blocks.
  - Each block is computed by a separate program instance in the kernel launch grid

```
out.block(x:1, y:2)
```



# Block

- Partitions output tensor into blocks.
  - Each block is computed by a separate program instance in the kernel launch grid

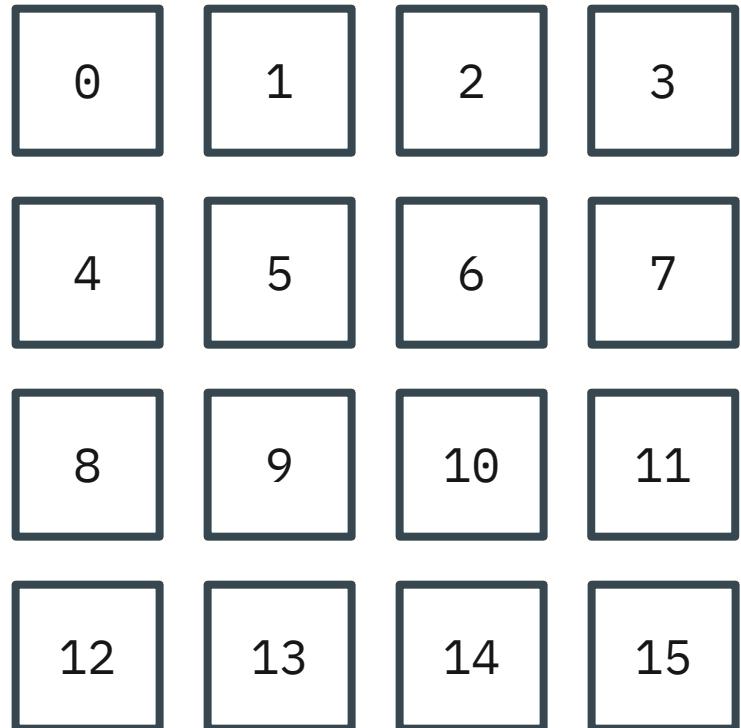
```
out.block(x:1, y:2)
```



# Block

- Partitions output tensor into blocks.
  - Each block is computed by a separate program instance in the kernel launch grid

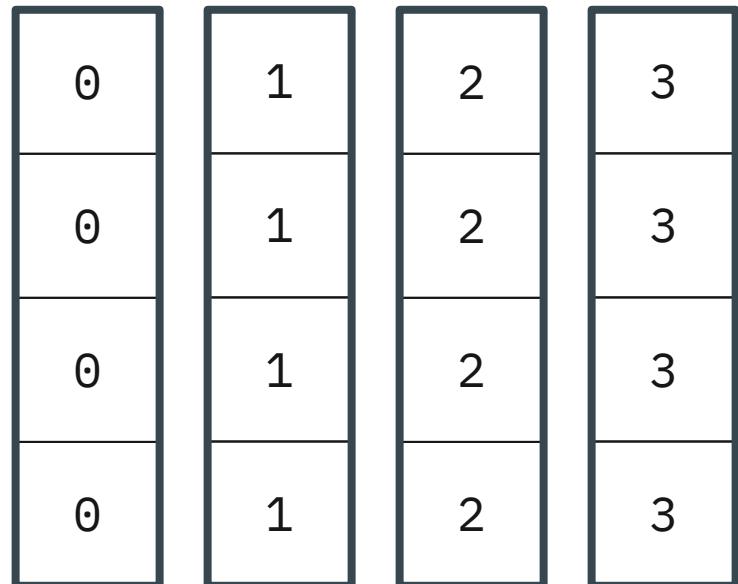
```
out.block(x:1, y:1)
```



# Block

- Partitions output tensor into blocks.
  - Each block is computed by a separate program instance in the kernel launch grid

```
out.block(x:4, y:1)
```



# Block

- Partitions output tensor into blocks.
  - Each block is computed by a separate program instance in the kernel launch grid

```
out.block(x:4, y:4)
```

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

# Block

```
Func add_out;  
In  A, B;  
SIn alpha;  
Var x, y;  
  
add_out[x,y] = alpha * (A[x,y] + B[x,y]);  
  
add_out.block(x:4, y:64);  
add_out.compile();
```

# Block

```
Func add_out;
In  A, B;
SIn alpha;
Var x, y;

add_out[x,y] = alpha * (A[x,y] + B[x,y]);

add_out.block(x:4, y:64);
add_out.compile();
```

# Block

```
Func add_out;  
In  A, B;  
SIn alpha;  
Var x, y;  
  
add_out[x,y] = alpha * (A[x,y] + B[x,y]);  
  
add_out.block(x:4, y:64);  
add_out.compile();
```

```
@triton.jit  
def add_out_kernel(...):  
    y_BLOCK_COUNT = y_SIZE // 64  
    x_BLOCK_COUNT = x_SIZE // 4  
    y_pid = tl.program_id(0).to(tl.int64) //  
            x_BLOCK_COUNT % y_BLOCK_COUNT  
    x_pid = tl.program_id(0).to(tl.int64) % x_BLOCK_COUNT  
    y_block_start = y_pid * 64  
    x_block_start = x_pid * 4  
    for x_iter in range(0, 4, 1):  
        for y_iter in range(0, 64, 1):  
            A = tl.load(A_ptr + (x_block_start + x_iter) * A_x_stride +  
                        (y_block_start + y_iter) * A_y_stride)  
            B = tl.load(B_ptr + (x_block_start + x_iter) * B_x_stride +  
                        (y_block_start + y_iter) * B_y_stride)  
            add_out = alpha * (A + B)  
            tl.store(add_out_ptr + (x_block_start + x_iter) *  
                    add_out_x_stride + (y_block_start + y_iter) *  
                    add_out_y_stride, add_out)  
  
def add_out(...):  
    ...  
    add_out_grid = (triton.cdiv(x, 4) * triton.cdiv(y, 64)),  
    add_out_kernel[add_out_grid](...)  
    ...
```

# Block

```
Func add_out;  
In  A, B;  
SIn alpha;  
Var x, y;  
  
add_out[x,y] = alpha * (A[x,y] + B[x,y]);  
  
add_out.block(x:4, y:64);  
add_out.compile();
```

```
@triton.jit  
def add_out_kernel(...):  
    y_BLOCK_COUNT = y_SIZE // 64  
    x_BLOCK_COUNT = x_SIZE // 4  
    y_pid = tl.program_id(0).to(tl.int64) //  
            x_BLOCK_COUNT % y_BLOCK_COUNT  
    x_pid = tl.program_id(0).to(tl.int64) % x_BLOCK_COUNT  
    y_block_start = y_pid * 64  
    x_block_start = x_pid * 4  
    for x_iter in range(0, 4, 1):  
        for y_iter in range(0, 64, 1):  
            A = tl.load(A_ptr + (x_block_start + x_iter) * A_x_stride +  
                        (y_block_start + y_iter) * A_y_stride)  
            B = tl.load(B_ptr + (x_block_start + x_iter) * B_x_stride +  
                        (y_block_start + y_iter) * B_y_stride)  
            add_out = alpha * (A + B)  
            tl.store(add_out_ptr + (x_block_start + x_iter) *  
                    add_out_x_stride + (y_block_start + y_iter) *  
                    add_out_y_stride, add_out)  
  
    def add_out(...):  
        ...  
        add_out_grid = (triton.cdiv(x, 4) * triton.cdiv(y, 64)),  
        add_out_kernel[add_out_grid](...)  
        ...
```

# Block

```
Func add_out;  
In  A, B;  
SIn alpha;  
Var x, y;  
  
add_out[x,y] = alpha * (A[x,y] + B[x,y]);  
  
add_out.block(x:4, y:64);  
add_out.compile();
```

```
@triton.jit  
def add_out_kernel(...):  
    y_BLOCK_COUNT = y_SIZE // 64  
    x_BLOCK_COUNT = x_SIZE // 4  
    y_pid = tl.program_id(0).to(tl.int64) //  
            x_BLOCK_COUNT % y_BLOCK_COUNT  
    x_pid = tl.program_id(0).to(tl.int64) % x_BLOCK_COUNT  
    y_block_start = y_pid * 64  
    x_block_start = x_pid * 4  
    for x_iter in range(0, 4, 1):  
        for y_iter in range(0, 64, 1):  
            A = tl.load(A_ptr + (x_block_start + x_iter) * A_x_stride +  
                        (y_block_start + y_iter) * A_y_stride)  
            B = tl.load(B_ptr + (x_block_start + x_iter) * B_x_stride +  
                        (y_block_start + y_iter) * B_y_stride)  
            add_out = alpha * (A + B)  
            tl.store(add_out_ptr + (x_block_start + x_iter) *  
                    add_out_x_stride + (y_block_start + y_iter) *  
                    add_out_y_stride, add_out)  
  
def add_out(...):  
    ...  
    add_out_grid = (triton.cdiv(x, 4) * triton.cdiv(y, 64)),  
    add_out_kernel[add_out_grid](...)  
    ...
```

# Block

```
Func add_out;  
In  A, B;  
SIn alpha;  
Var x, y;  
  
add_out[x,y] = alpha * (A[x,y] + B[x,y]);  
  
add_out.block(x:4, y:64);  
add_out.compile();
```

```
@triton.jit  
def add_out_kernel(...):  
    y_BLOCK_COUNT = y_SIZE // 64  
    x_BLOCK_COUNT = x_SIZE // 4  
    y_pid = tl.program_id(0).to(tl.int64) //  
            x_BLOCK_COUNT % y_BLOCK_COUNT  
    x_pid = tl.program_id(0).to(tl.int64) % x_BLOCK_COUNT  
    y_block_start = y_pid * 64  
    x_block_start = x_pid * 4  
    for x_iter in range(0, 4, 1):  
        for y_iter in range(0, 64, 1):  
            A = tl.load(A_ptr + (x_block_start + x_iter) * A_x_stride +  
                        (y_block_start + y_iter) * A_y_stride)  
            B = tl.load(B_ptr + (x_block_start + x_iter) * B_x_stride +  
                        (y_block_start + y_iter) * B_y_stride)  
            add_out = alpha * (A + B)  
            tl.store(add_out_ptr + (x_block_start + x_iter) *  
                    add_out_x_stride + (y_block_start + y_iter) *  
                    add_out_y_stride, add_out)  
  
def add_out(...):  
    ...  
    add_out_grid = (triton.cdiv(x, 4) * triton.cdiv(y, 64)),  
    add_out_kernel[add_out_grid](...)  
    ...
```

# Block

```
Func add_out;
In  A, B;
SIn alpha;
Var x, y;

add_out[x,y] = alpha * (A[x,y] + B[x,y]);

add_out.block(x:4, y:64);
add_out.compile();
```

```
@triton.jit
def add_out_kernel(...):
    y_BLOCK_COUNT = y_SIZE // 64
    x_BLOCK_COUNT = x_SIZE // 4
    y_pid = tl.program_id(0).to(tl.int64) //
                  x_BLOCK_COUNT % y_BLOCK_COUNT
    x_pid = tl.program_id(0).to(tl.int64) % x_BLOCK_COUNT
    y_block_start = y_pid * 64
    x_block_start = x_pid * 4
    for x_iter in range(0, 4, 1):
        for y_iter in range(0, 64, 1):
            A = tl.load(A_ptr + (x_block_start + x_iter) * A_x_stride +
                        (y_block_start + y_iter) * A_y_stride)
            B = tl.load(B_ptr + (x_block_start + x_iter) * B_x_stride +
                        (y_block_start + y_iter) * B_y_stride)
            add_out = alpha * (A + B)
            tl.store(add_out_ptr + (x_block_start + x_iter) *
                    add_out_x_stride + (y_block_start + y_iter) *
                    add_out_y_stride, add_out)

def add_out(...):
    ...
    add_out_grid = (triton.cdiv(x, 4) * triton.cdiv(y, 64)),
    add_out_kernel[add_out_grid](...)
    ...
```

# Block

```
Func add_out;
In  A, B;
SIn alpha;
Var x, y;

add_out[x,y] = alpha * (A[x,y] + B[x,y]);

add_out.block(x:4, y:64);
add_out.compile();
```

```
@triton.jit
def add_out_kernel(...):
    y_BLOCK_COUNT = y_SIZE // 64
    x_BLOCK_COUNT = x_SIZE // 4
    y_pid = tl.program_id(0).to(tl.int64) //
                  x_BLOCK_COUNT % y_BLOCK_COUNT
    x_pid = tl.program_id(0).to(tl.int64) % x_BLOCK_COUNT
    y_block_start = y_pid * 64
    x_block_start = x_pid * 4
    for x_iter in range(0, 4, 1):
        for y_iter in range(0, 64, 1):
            A = tl.load(A_ptr + (x_block_start + x_iter) * A_x_stride +
                         (y_block_start + y_iter) * A_y_stride)
            B = tl.load(B_ptr + (x_block_start + x_iter) * B_x_stride +
                         (y_block_start + y_iter) * B_y_stride)
            add_out = alpha * (A + B)
            tl.store(add_out_ptr + (x_block_start + x_iter) *
                     add_out_x_stride + (y_block_start + y_iter) *
                     add_out_y_stride, add_out)

def add_out(...):
    ...
    add_out_grid = (triton.cdiv(x, 4) * triton.cdiv(y, 64)),
    add_out_kernel[add_out_grid](...)
    ...
```

# Block

```
Func add_out;
In  A, B;
SIn alpha;
Var x, y;

add_out[x,y] = alpha * (A[x,y] + B[x,y]);

add_out.block(x:4, y:64);
add_out.compile();
```

```
@triton.jit
def add_out_kernel(...):
    y_BLOCK_COUNT = y_SIZE // 64
    x_BLOCK_COUNT = x_SIZE // 4
    y_pid = tl.program_id(0).to(tl.int64) //
               x_BLOCK_COUNT % y_BLOCK_COUNT
    x_pid = tl.program_id(0).to(tl.int64) % x_BLOCK_COUNT
    y_block_start = y_pid * 64
    x_block_start = x_pid * 4
    for x_iter in range(0, 4, 1):
        for y_iter in range(0, 64, 1):
            A = tl.load(A_ptr + (x_block_start + x_iter) * A_x_stride +
                         (y_block_start + y_iter) * A_y_stride)
            B = tl.load(B_ptr + (x_block_start + x_iter) * B_x_stride +
                         (y_block_start + y_iter) * B_y_stride)
            add_out = alpha * (A + B)
            tl.store(add_out_ptr + (x_block_start + x_iter) *
                     add_out_x_stride + (y_block_start + y_iter) *
                     add_out_y_stride, add_out)

def add_out(...):
    ...
    add_out_grid = (triton.cdiv(x, 4) * triton.cdiv(y, 64)),
    add_out_kernel[add_out_grid](...)
    ...
```

# Tensorize

# Tensorize

- Partitions output block into tensors.

# Tensorize

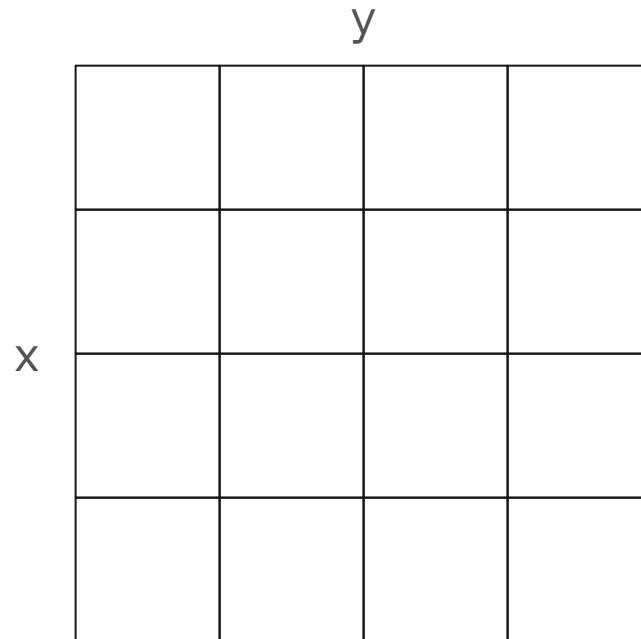
- Partitions output block into tensors.
  - Each tensor is computed by SIMD tensor operations in a sequential loop

# Tensorize

- Partitions output block into tensors.
  - Each tensor is computed by SIMD tensor operations in a sequential loop


# Tensorize

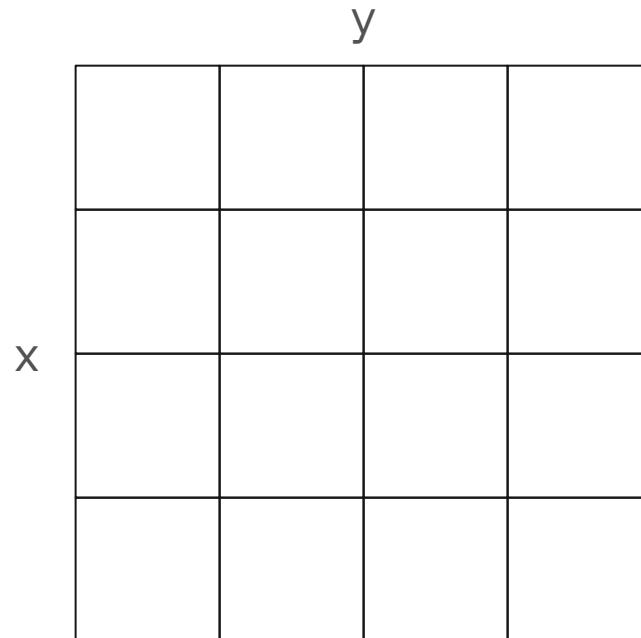
- Partitions output block into tensors.
  - Each tensor is computed by SIMD tensor operations in a sequential loop



# Tensorize

- Partitions output block into tensors.
  - Each tensor is computed by SIMD tensor operations in a sequential loop

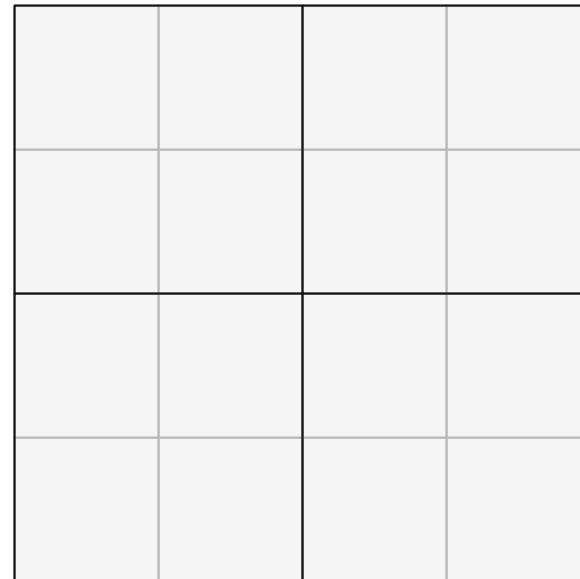
```
out.tensorize(x:2, y:2)
```



# Tensorize

- Partitions output block into tensors.
  - Each tensor is computed by SIMD tensor operations in a sequential loop

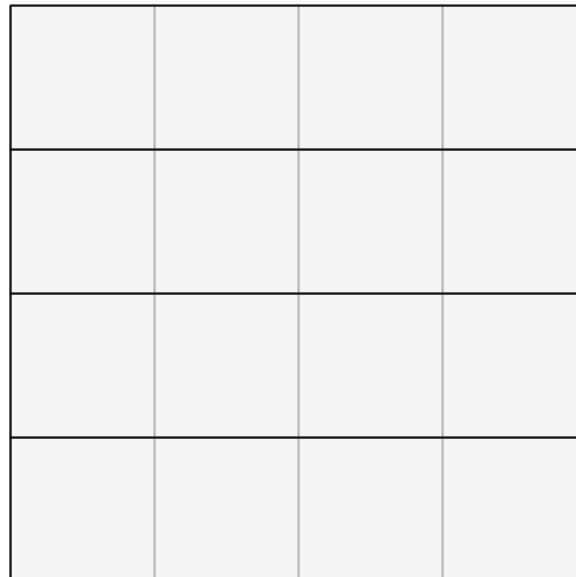
```
out.tensorize(x:2, y:2)
```



# Tensorize

- Partitions output block into tensors.
  - Each tensor is computed by SIMD tensor operations in a sequential loop

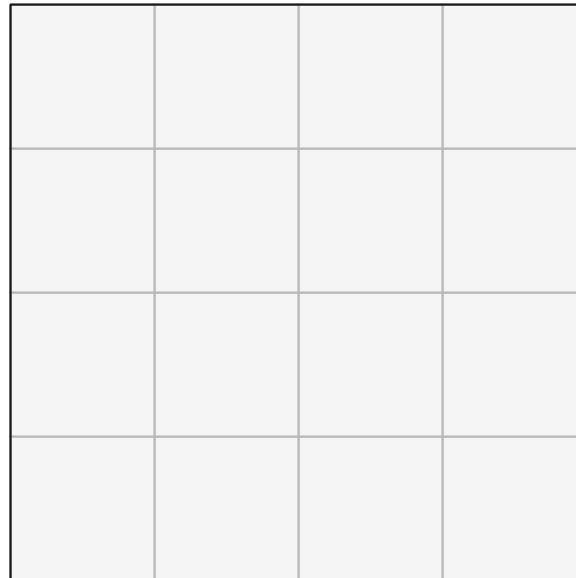
```
out.tensorize(x:1, y:4)
```



# Tensorize

- Partitions output block into tensors.
  - Each tensor is computed by SIMD tensor operations in a sequential loop

```
out.tensorize(x:4, y:4)
```



# Tensorize

- Partitions output block into tensors.
  - Each tensor is computed by SIMD tensor operations in a sequential loop

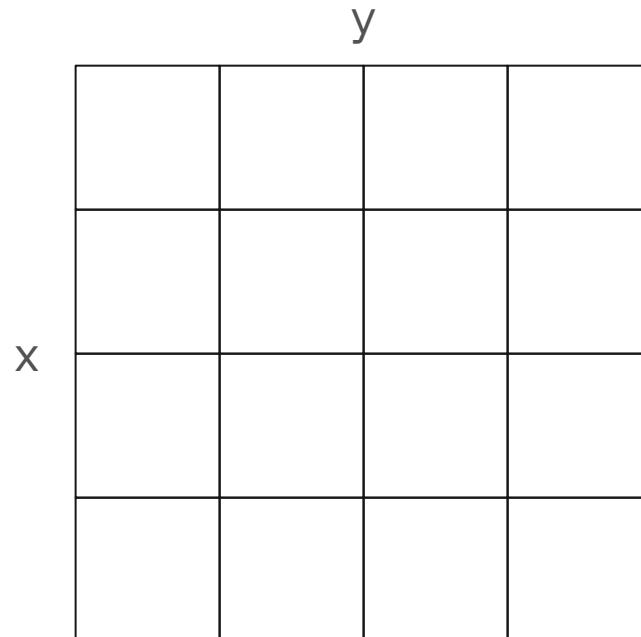
```
out.tensorize(x:1, y:1)
```


# Tensorize

- Partitions output block into tensors.
  - Each tensor is computed by SIMD tensor operations in a sequential loop

```
out.block(x:2, y:4)
```

```
out.tensorize(x:2, y:1)
```



# Tensorize

- Partitions output block into tensors.
  - Each tensor is computed by SIMD tensor operations in a sequential loop

```
out.block(x:2, y:4)
```

```
out.tensorize(x:2, y:1)
```

0	0	0	0
0	0	0	0

1	1	1	1
1	1	1	1

# Tensorize

- Partitions output block into tensors.
  - Each tensor is computed by SIMD tensor operations in a sequential loop

```
out.block(x:2, y:4)
```

```
out.tensorize(x:2, y:1)
```

0	0	0	0
0	0	0	0

1	1	1	1
1	1	1	1

# Tensorize

```
Func add_out;
In  A, B;
SIn alpha;
Var x, y;

add_out[x,y] = alpha * (A[x,y] + B[x,y]);

add_out.block(x:4, y:64);
add_out.tensorize(x:2, y:64);
add_out.compile();
```

# Tensorize

```
Func add_out;
In  A, B;
SIn alpha;
Var x, y;

add_out[x,y] = alpha * (A[x,y] + B[x,y]);

add_out.block(x:4, y:64);
add_out.tensorize(x:2, y:64);
add_out.compile();
```

# Tensorize

```
Func add_out;
In  A, B;
SIn alpha;
Var x, y;

add_out[x,y] = alpha * (A[x,y] + B[x,y]);
add_out.block(x:4, y:64);
add_out.tensorize(x:2, y:64);
add_out.compile();
```

```
@triton.jit
def add_out_kernel(...):
    y_BLOCK_COUNT = y_SIZE // 64
    x_BLOCK_COUNT = x_SIZE // 4
    y_pid = tl.program_id(0).to(tl.int64) //  
        x_BLOCK_COUNT % y_BLOCK_COUNT
    x_pid = tl.program_id(0).to(tl.int64) % x_BLOCK_COUNT
    y_block_start = y_pid * 64
    x_block_start = x_pid * 4
    y_arange = tl.arange(0, 64)
    x_arange = tl.arange(0, 2)
    for x_iter in range(0, 4, 2):
        A = tl.load(A_ptr + (x_block_start + x_iter + x_arange[:, None])  
                    * A_x_stride + (y_block_start + y_arange[None, :])  
                    * A_y_stride)
        B = tl.load(B_ptr + (x_block_start + x_iter + x_arange[:, None])  
                    * B_x_stride + (y_block_start + y_arange[None, :])  
                    * B_y_stride)
        add_out = alpha * (A + B)
        tl.store(add_out_ptr +  
                (x_block_start + x_iter + x_arange[:, None]) *  
                add_out_x_stride + (y_block_start + y_arange[None, :]) *  
                add_out_y_stride, add_out)

def add_out(...):  
    ...  
    add_out_grid = (triton.cdiv(x, 4) * triton.cdiv(y, 64)),  
    add_out_kernel[add_out_grid](...)  
    ...
```

# Tensorize

```
Func add_out;
In  A, B;
SIn alpha;
Var x, y;

add_out[x,y] = alpha * (A[x,y] + B[x,y]);
add_out.block(x:4, y:64);
add_out.tensorize(x:2, y:64);
add_out.compile();
```

```
@triton.jit
def add_out_kernel(...):
    y_BLOCK_COUNT = y_SIZE // 64
    x_BLOCK_COUNT = x_SIZE // 4
    y_pid = tl.program_id(0).to(tl.int64) //  
        x_BLOCK_COUNT % y_BLOCK_COUNT
    x_pid = tl.program_id(0).to(tl.int64) % x_BLOCK_COUNT
    y_block_start = y_pid * 64
    x_block_start = x_pid * 4
    y_arange = tl.arange(0, 64)
    x_arange = tl.arange(0, 2)
    for x_iter in range(0, 4, 2):
        A = tl.load(A_ptr + (x_block_start + x_iter + x_arange[:, None])  
                    * A_x_stride + (y_block_start + y_arange[None, :])  
                    * A_y_stride)
        B = tl.load(B_ptr + (x_block_start + x_iter + x_arange[:, None])  
                    * B_x_stride + (y_block_start + y_arange[None, :])  
                    * B_y_stride)
        add_out = alpha * (A + B)
        tl.store(add_out_ptr +  
                (x_block_start + x_iter + x_arange[:, None]) *  
                add_out_x_stride + (y_block_start + y_arange[None, :]) *  
                add_out_y_stride, add_out)

def add_out(...):
    ...
    add_out_grid = (triton.cdiv(x, 4) * triton.cdiv(y, 64)),
    add_out_kernel[add_out_grid](...)
    ...
```

# Tensorize

```
Func add_out;
In  A, B;
SIn alpha;
Var x, y;

add_out[x,y] = alpha * (A[x,y] + B[x,y]);
add_out.block(x:4, y:64);
add_out.tensorize(x:2, y:64);
add_out.compile();
```

```
@triton.jit
def add_out_kernel(...):
    y_BLOCK_COUNT = y_SIZE // 64
    x_BLOCK_COUNT = x_SIZE // 4
    y_pid = tl.program_id(0).to(tl.int64) //  
        x_BLOCK_COUNT % y_BLOCK_COUNT
    x_pid = tl.program_id(0).to(tl.int64) % x_BLOCK_COUNT
    y_block_start = y_pid * 64
    x_block_start = x_pid * 4
    y_arange = tl.arange(0, 64)
    x_arange = tl.arange(0, 2)
    for x_iter in range(0, 4, 2):
        A = tl.load(A_ptr + (x_block_start + x_iter + x_arange[:, None])  
                    * A_x_stride + (y_block_start + y_arange[None, :])  
                    * A_y_stride)
        B = tl.load(B_ptr + (x_block_start + x_iter + x_arange[:, None])  
                    * B_x_stride + (y_block_start + y_arange[None, :])  
                    * B_y_stride)
        add_out = alpha * (A + B)
        tl.store(add_out_ptr +  
                (x_block_start + x_iter + x_arange[:, None]) *  
                add_out_x_stride + (y_block_start + y_arange[None, :]) *  
                add_out_y_stride, add_out)

def add_out(...):
    ...
    add_out_grid = (triton.cdiv(x, 4) * triton.cdiv(y, 64)),
    add_out_kernel[add_out_grid](...)
    ...
```

# Tensorize

```
Func add_out;
In  A, B;
SIn alpha;
Var x, y;

add_out[x,y] = alpha * (A[x,y] + B[x,y]);
add_out.block(x:4, y:64);
add_out.tensorize(x:2, y:64);
add_out.compile();
```

```
@triton.jit
def add_out_kernel(...):
    y_BLOCK_COUNT = y_SIZE // 64
    x_BLOCK_COUNT = x_SIZE // 4
    y_pid = tl.program_id(0).to(tl.int64) //  
        x_BLOCK_COUNT % y_BLOCK_COUNT
    x_pid = tl.program_id(0).to(tl.int64) % x_BLOCK_COUNT
    y_block_start = y_pid * 64
    x_block_start = x_pid * 4
    y_arange = tl.arange(0, 64)
    x_arange = tl.arange(0, 2)
    for x_iter in range(0, 4, 2):
        A = tl.load(A_ptr + (x_block_start + x_iter + x_arange[:, None])  
            * A_x_stride + (y_block_start + y_arange[None, :])  
            * A_y_stride)
        B = tl.load(B_ptr + (x_block_start + x_iter + x_arange[:, None])  
            * B_x_stride + (y_block_start + y_arange[None, :])  
            * B_y_stride)
        add_out = alpha * (A + B)
        tl.store(add_out_ptr +  
            (x_block_start + x_iter + x_arange[:, None]) *  
            add_out_x_stride + (y_block_start + y_arange[None, :]) *  
            add_out_y_stride, add_out)

def add_out(...):
    ...
    add_out_grid = (triton.cdiv(x, 4) * triton.cdiv(y, 64)),
    add_out_kernel[add_out_grid](...)
    ...
```

# Tensorize

```
Func add_out;
In  A, B;
SIn alpha;
Var x, y;

add_out[x,y] = alpha * (A[x,y] + B[x,y]);
add_out.block(x:4, y:64);
add_out.tensorize(x:2, y:64);
add_out.compile();
```

```
@triton.jit
def add_out_kernel(...):
    y_BLOCK_COUNT = y_SIZE // 64
    x_BLOCK_COUNT = x_SIZE // 4
    y_pid = tl.program_id(0).to(tl.int64) //  
        x_BLOCK_COUNT % y_BLOCK_COUNT
    x_pid = tl.program_id(0).to(tl.int64) % x_BLOCK_COUNT
    y_block_start = y_pid * 64
    x_block_start = x_pid * 4
    y_arange = tl.arange(0, 64)
    x_arange = tl.arange(0, 2)
    for x_iter in range(0, 4, 2):
        A = tl.load(A_ptr + (x_block_start + x_iter + x_arange[:, None])  
            * A_x_stride + (y_block_start + y_arange[None, :])  
            * A_y_stride)
        B = tl.load(B_ptr + (x_block_start + x_iter + x_arange[:, None])  
            * B_x_stride + (y_block_start + y_arange[None, :])  
            * B_y_stride)
        add_out = alpha * (A + B)
        tl.store(add_out_ptr +  
            (x_block_start + x_iter + x_arange[:, None]) *  
            add_out_x_stride + (y_block_start + y_arange[None, :]) *  
            add_out_y_stride, add_out)

def add_out(...):
    ...
    add_out_grid = (triton.cdiv(x, 4) * triton.cdiv(y, 64)),
    add_out_kernel[add_out_grid](...)
    ...
```

# Map

# Map

- Specifies the mapping from program instance in the kernel launch grid to output block.

# Map

- Specifies the mapping from program instance in the kernel launch grid to output block.
  - Requires blocking

# Map

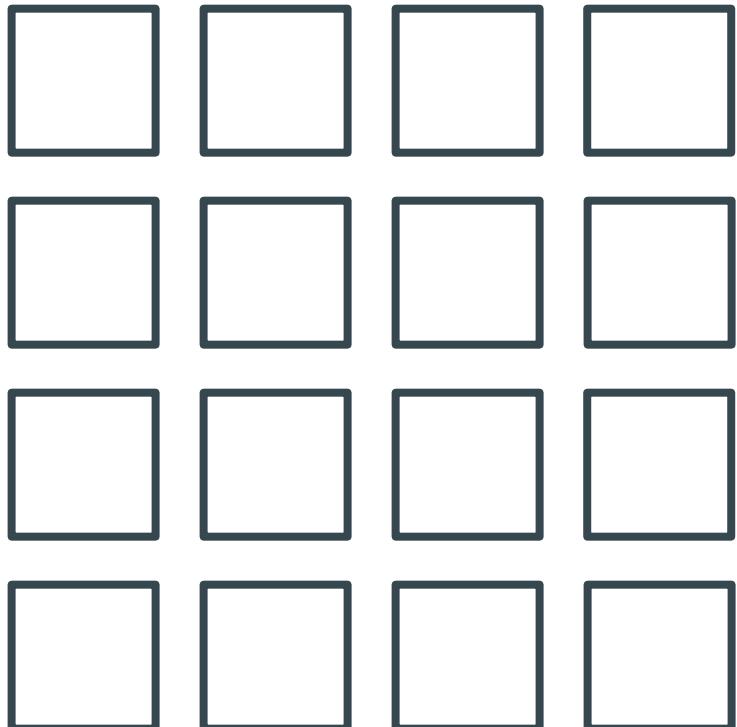
- Specifies the mapping from program instance in the kernel launch grid to output block.
  - Requires blocking

```
out.block(x:..., y:...)
```

# Map

- Specifies the mapping from program instance in the kernel launch grid to output block.
  - Requires blocking

```
out.block(x:..., y:...)
```

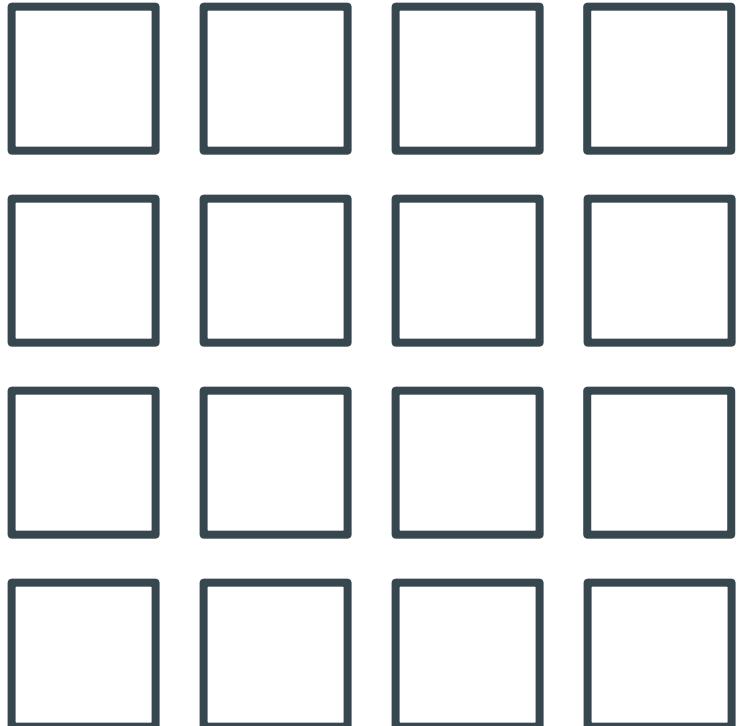


# Map

- Specifies the mapping from program instance in the kernel launch grid to output block.
  - Requires blocking

```
out.block(x:..., y:...)
```

```
out.map(x, y)
```

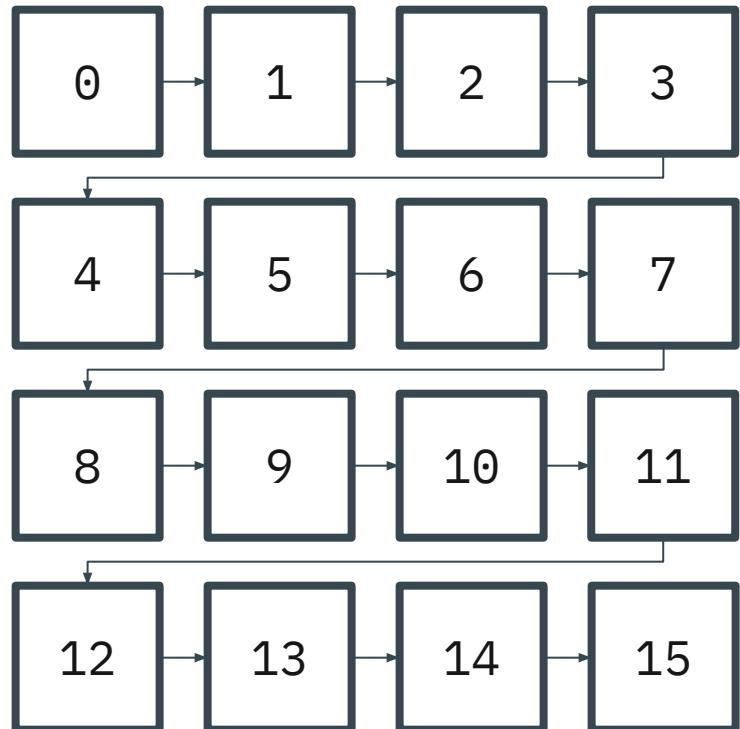


# Map

- Specifies the mapping from program instance in the kernel launch grid to output block.
  - Requires blocking

```
out.block(x:..., y:...)
```

```
out.map(x, y)
```

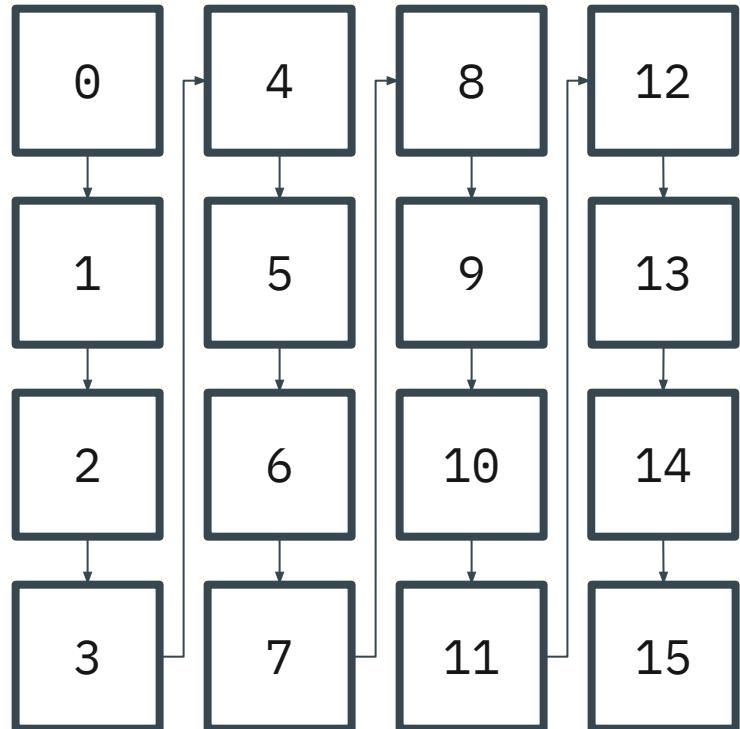


# Map

- Specifies the mapping from program instance in the kernel launch grid to output block.
  - Requires blocking

```
out.block(x:..., y:...)
```

```
out.map(y, x)
```

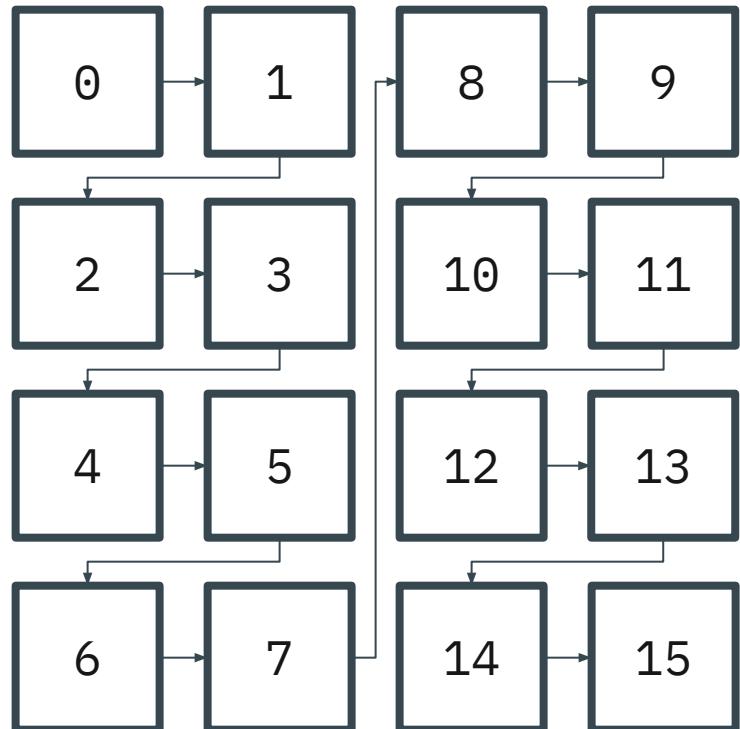


# Map

- Specifies the mapping from program instance in the kernel launch grid to output block.
  - Requires blocking

```
out.block(x:..., y:...)
```

```
out.map(y:yi/2, x, yi)
```

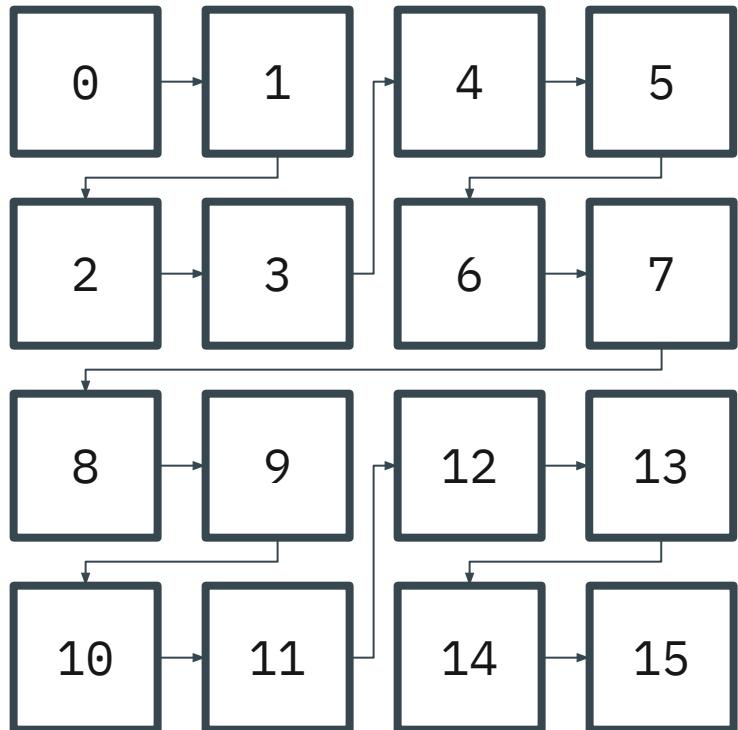


# Map

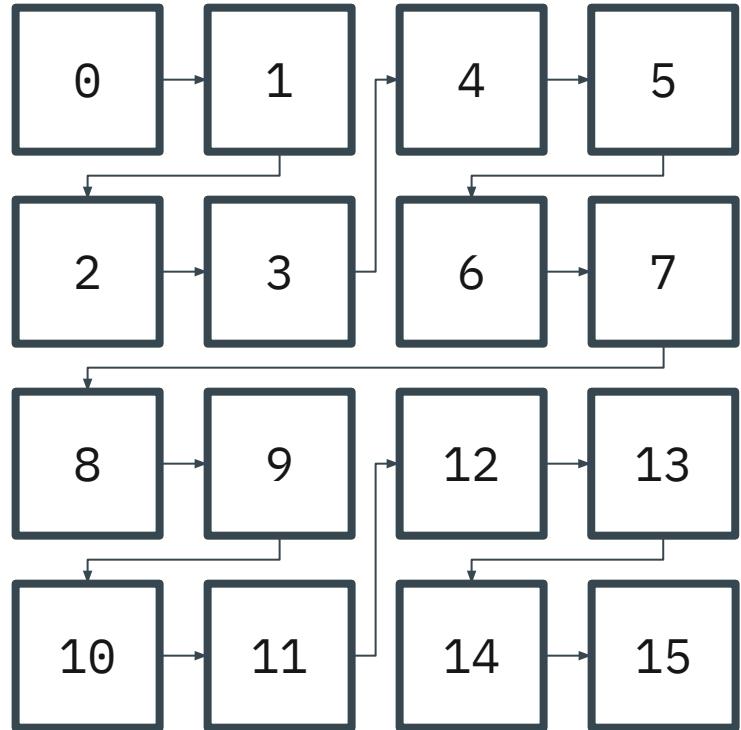
- Specifies the mapping from program instance in the kernel launch grid to output block.
  - Requires blocking

```
out.block(x:..., y:...)
```

```
out.map(x:xi/2, y:yi/2, xi, yi)
```

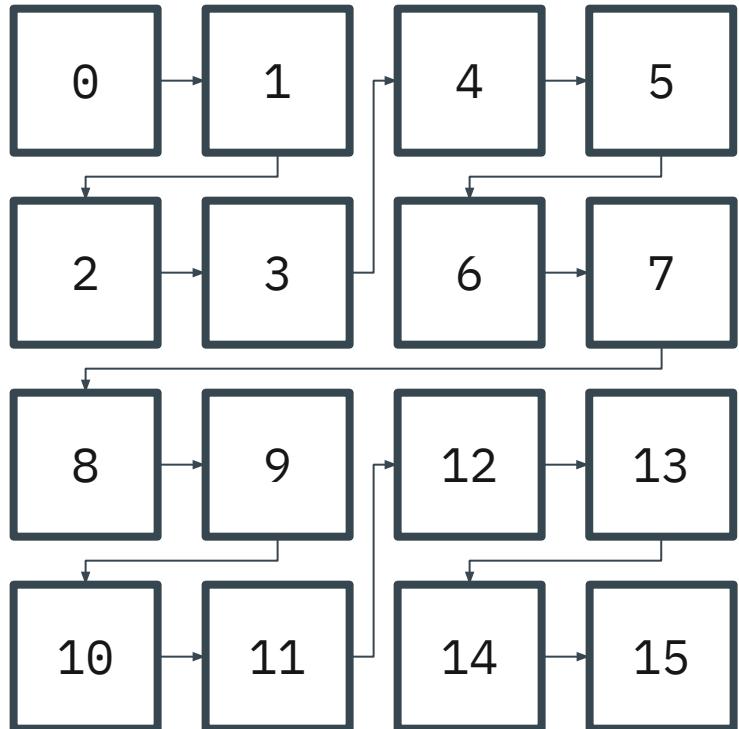


# Why map?



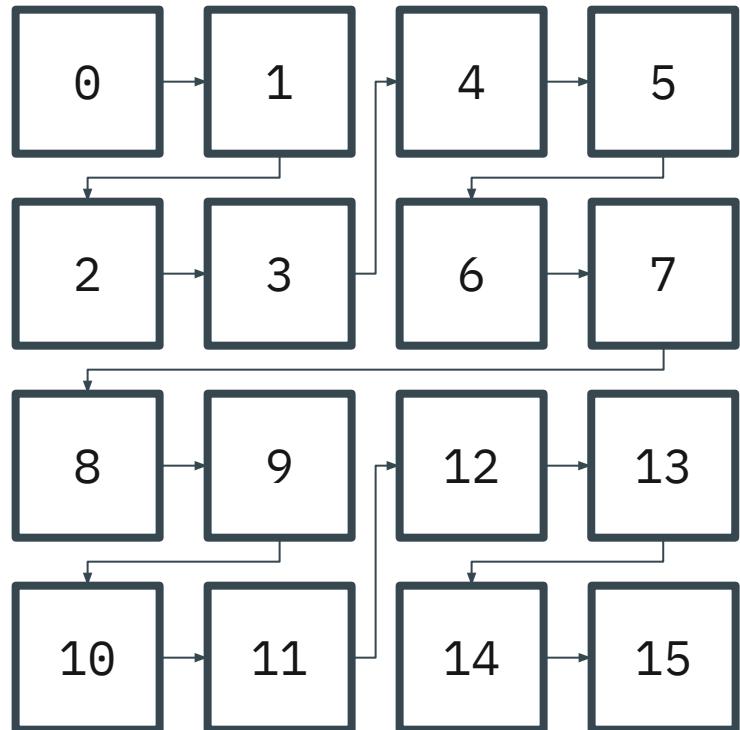
# Why map?

- Hardware resources constrain program instance parallelism.



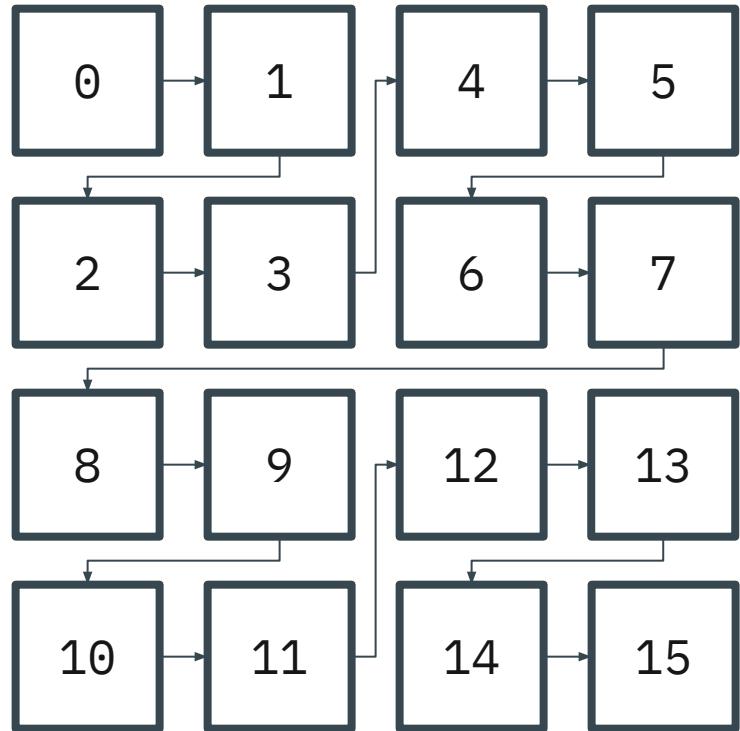
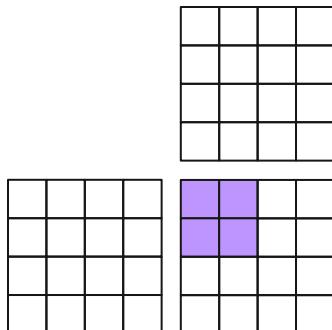
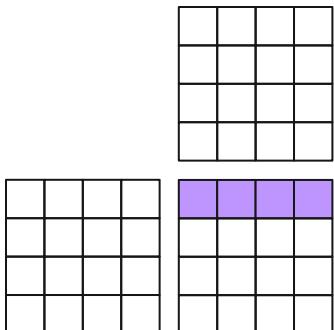
# Why map?

- Hardware resources constrain program instance parallelism.
- Program instances may reuse data loaded into the cache by others.



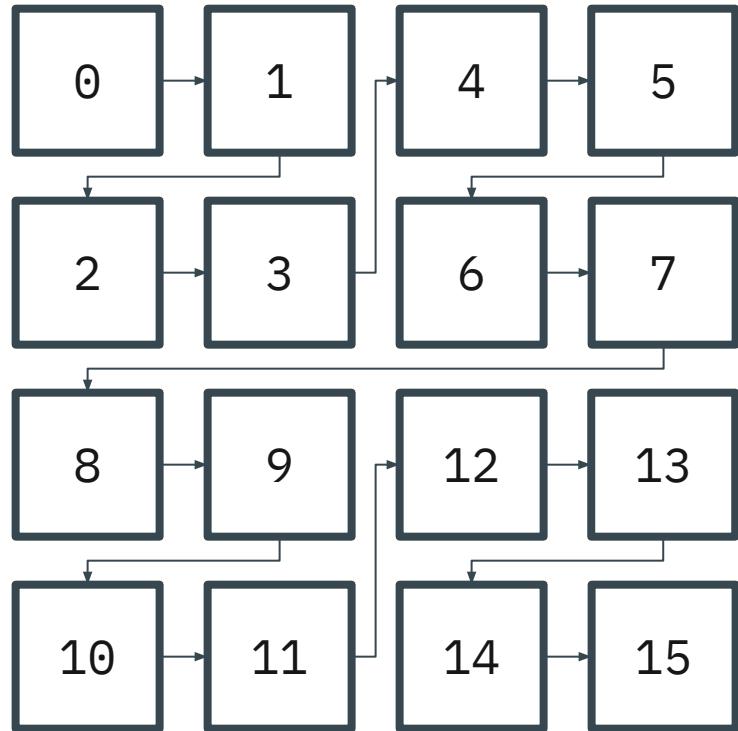
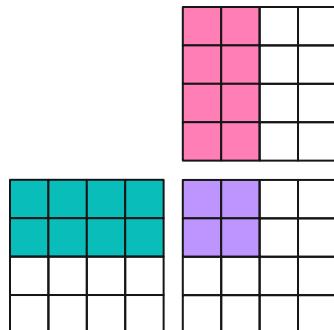
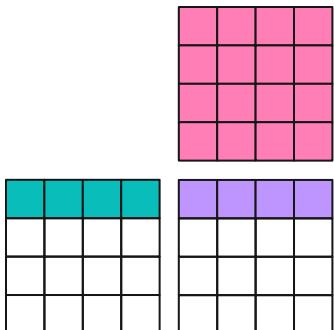
# Why map?

- Hardware resources constrain program instance parallelism.
- Program instances may reuse data loaded into the cache by others.



# Why map?

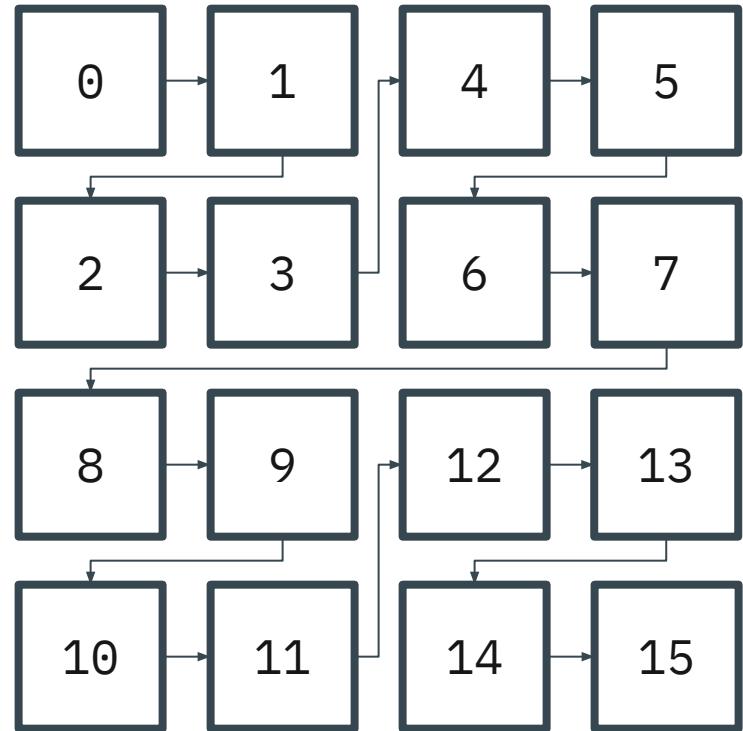
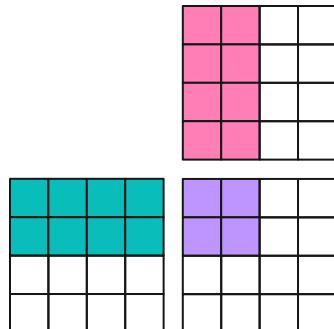
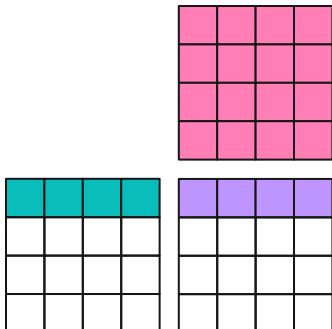
- Hardware resources constrain program instance parallelism.
- Program instances may reuse data loaded into the cache by others.



# Why map?

- Hardware resources constrain program instance parallelism.
- Program instances may reuse data loaded into the cache by others.

Program instance mapping matters!



Fuse at

## Fuse at

- Fuse a target func into a destination func at a particular loop level.

# Fuse at

- Fuse a target func into a destination func at a particular loop level.
  - Same as successive loop fusions

# Fuse at

- Fuse a target func into a destination func at a particular loop level.
  - Same as successive loop fusions

```
Func swish_out, tmp;
In A;
SIn beta;
Var x, y;

tmp[x, y] = sigmoid(beta * A[x, y]);
swish_out[x, y] = A[x, y] * tmp[x, y];

tmp.fuse_at(swish_out, y)
swish_out.compile();
```

# Evaluation

# Research Questions

# Research Questions

1. How **expressive** are DT's scheduling primitives for generating efficient ML kernels?

# Research Questions

1. How **expressive** are DT's scheduling primitives for generating efficient ML kernels?
2. Is finding efficient schedules with DT **feasible** for most ML operations?

# Research Questions

1. How **expressive** are DT's scheduling primitives for generating efficient ML kernels?
2. Is finding efficient schedules with DT **feasible** for most ML operations?
3. Are there **opportunities** for DT to generate ML kernels that outperform expert-written Triton kernels and ML frameworks?

# Experimental Methodology

# Experimental Methodology

DT  
Algorithm

# Experimental Methodology

DT  
Algorithm

```
Func relu_out;  
In A;  
Var x, y;  
  
relu_out[x, y] = maximum(0, A[x, y]);
```

# Experimental Methodology

DT  
Algorithm

# Experimental Methodology

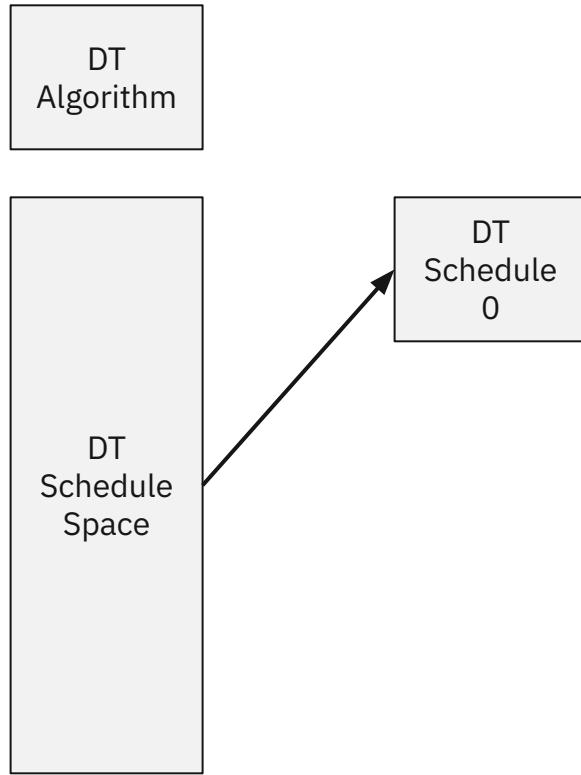


# Experimental Methodology

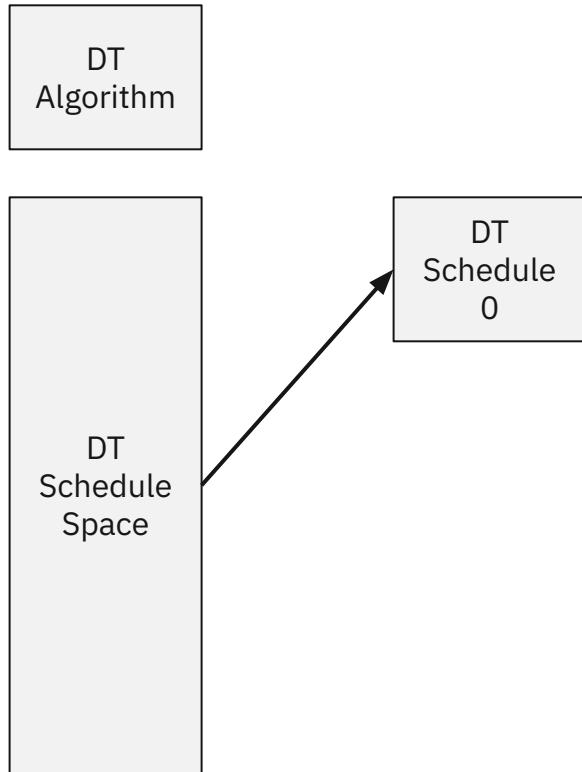


To define a schedule space, a small manual exploration is performed.

# Experimental Methodology

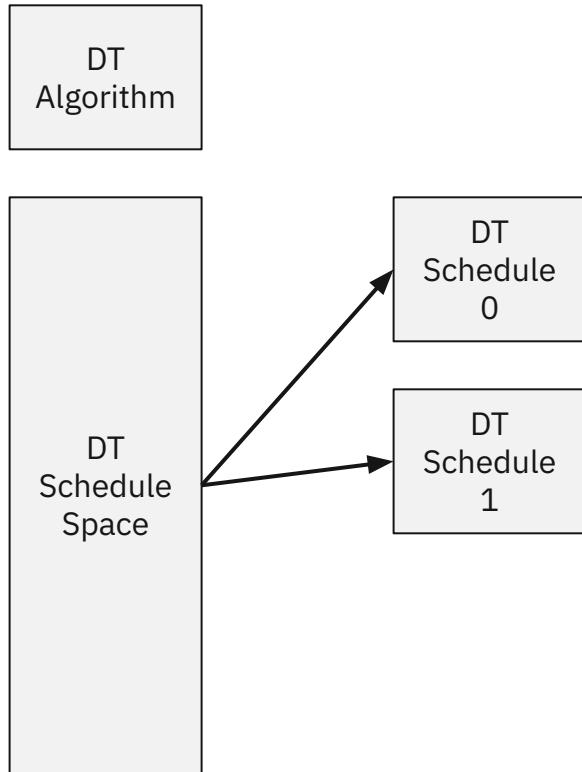


# Experimental Methodology



```
relu_out.block(x:8, y:16);  
relu_out.map(x:xi/4, y, xi);  
relu_out.compile();
```

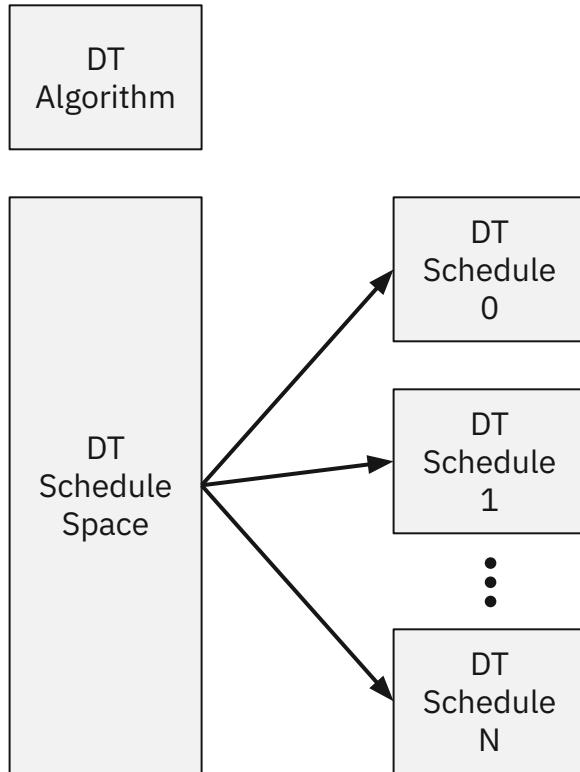
# Experimental Methodology



```
relu_out.block(x:8, y:16);  
relu_out.map(x:xi/4, y, xi);  
relu_out.compile();
```

```
relu_out.block(x:1, y:128);  
relu_out.tensorize(y:128);  
relu_out.map(y, x);  
relu_out.compile();
```

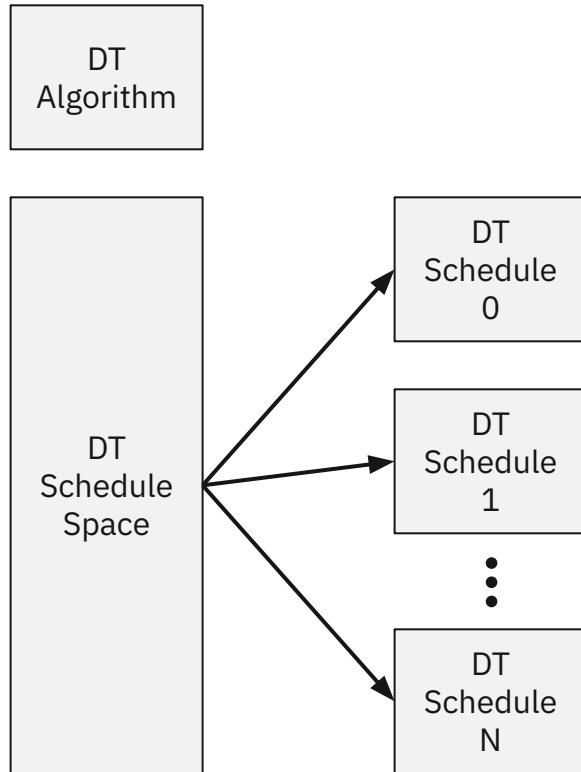
# Experimental Methodology



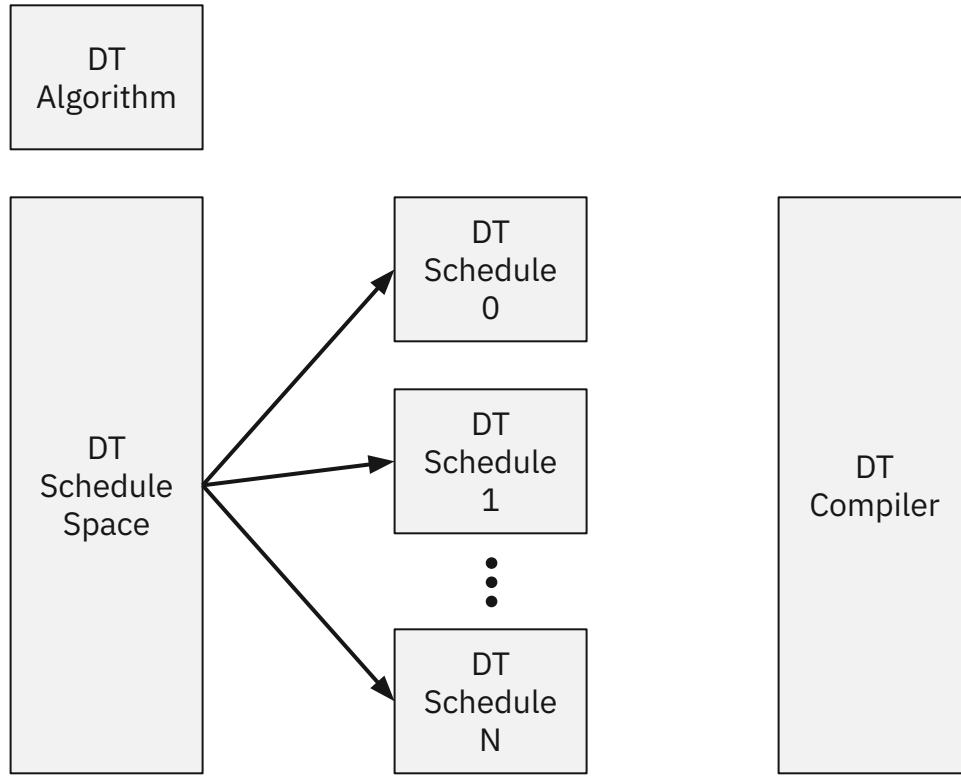
```
relu_out.block(x:8, y:16);  
relu_out.map(x:xi/4, y, xi);  
relu_out.compile();
```

```
relu_out.block(x:1, y:128);  
relu_out.tensorize(y:128);  
relu_out.map(y, x);  
relu_out.compile();
```

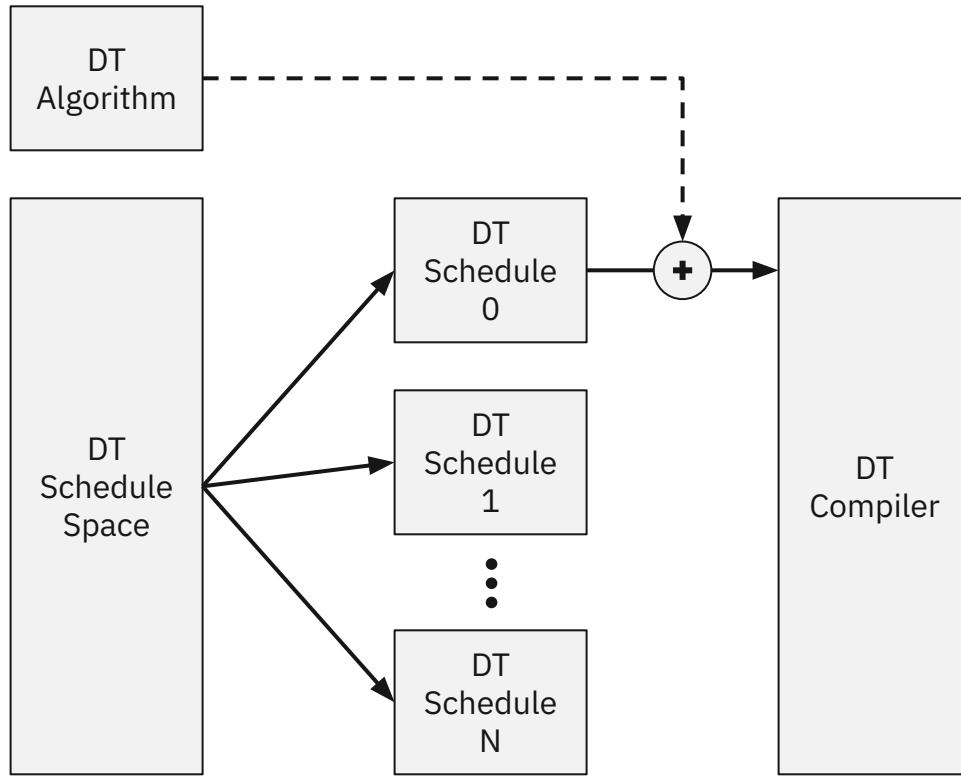
# Experimental Methodology



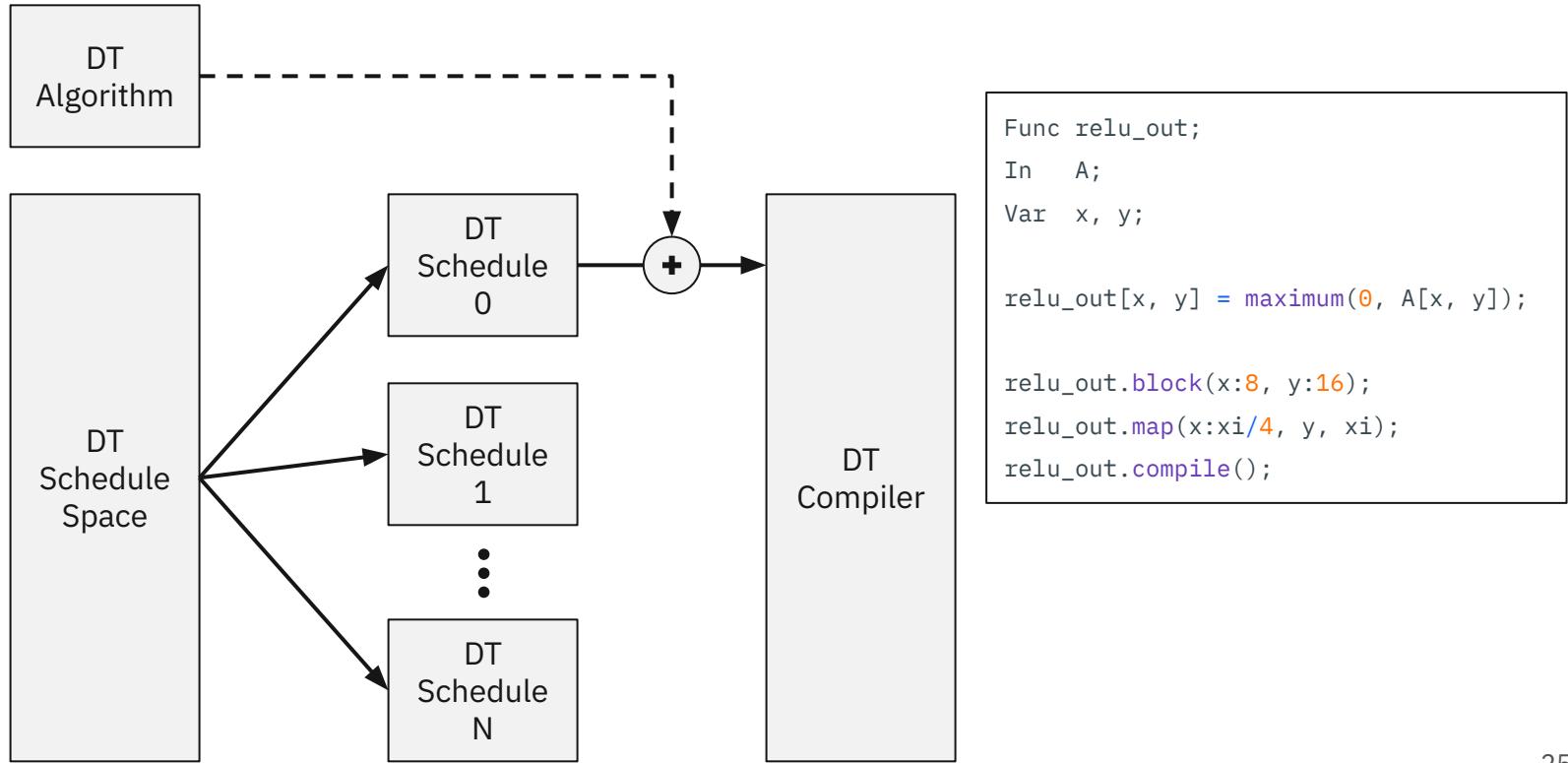
# Experimental Methodology



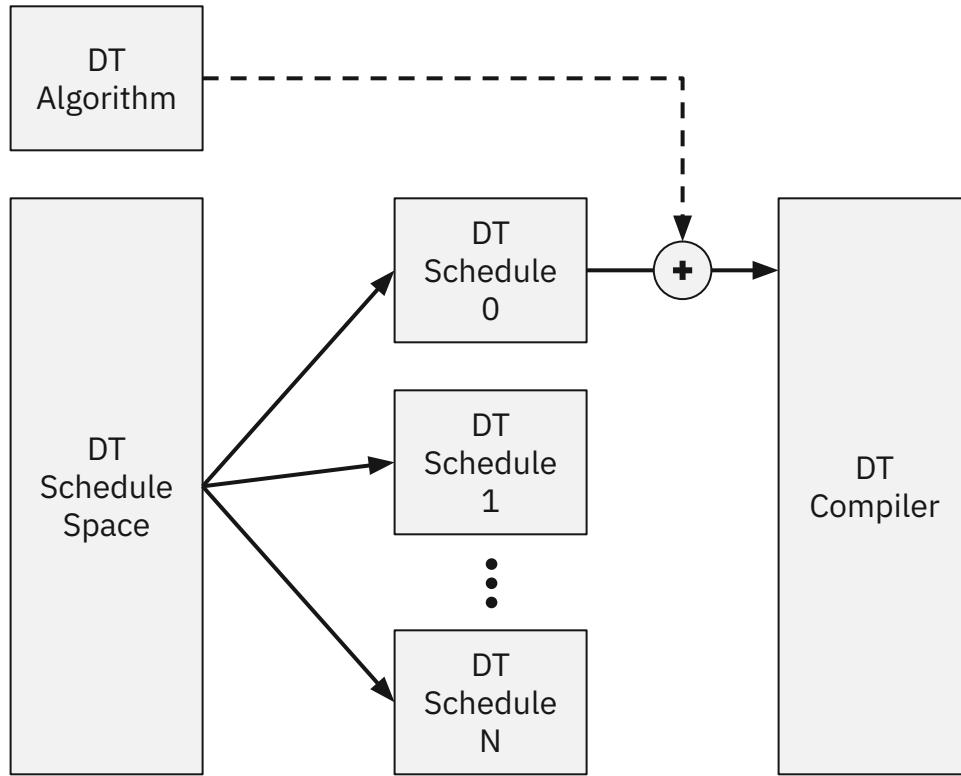
# Experimental Methodology



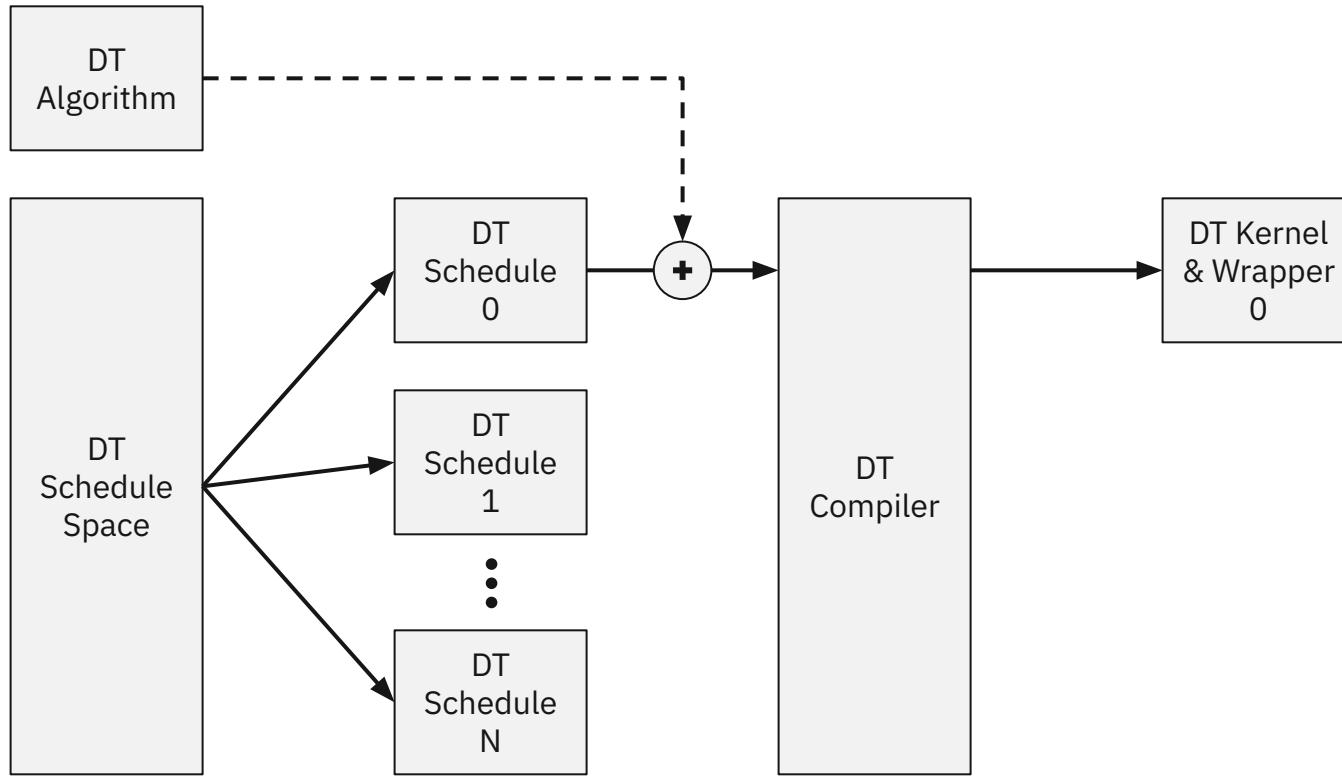
# Experimental Methodology



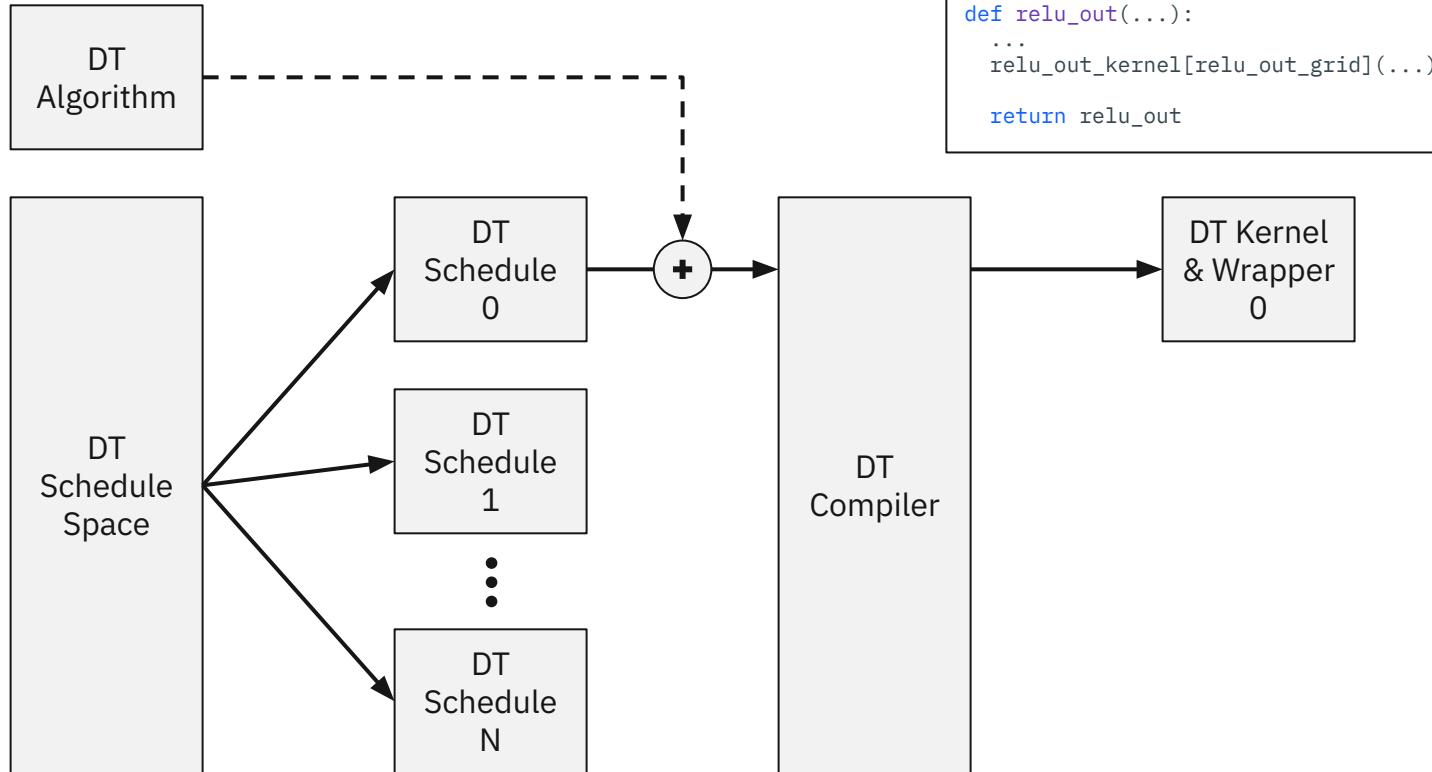
# Experimental Methodology



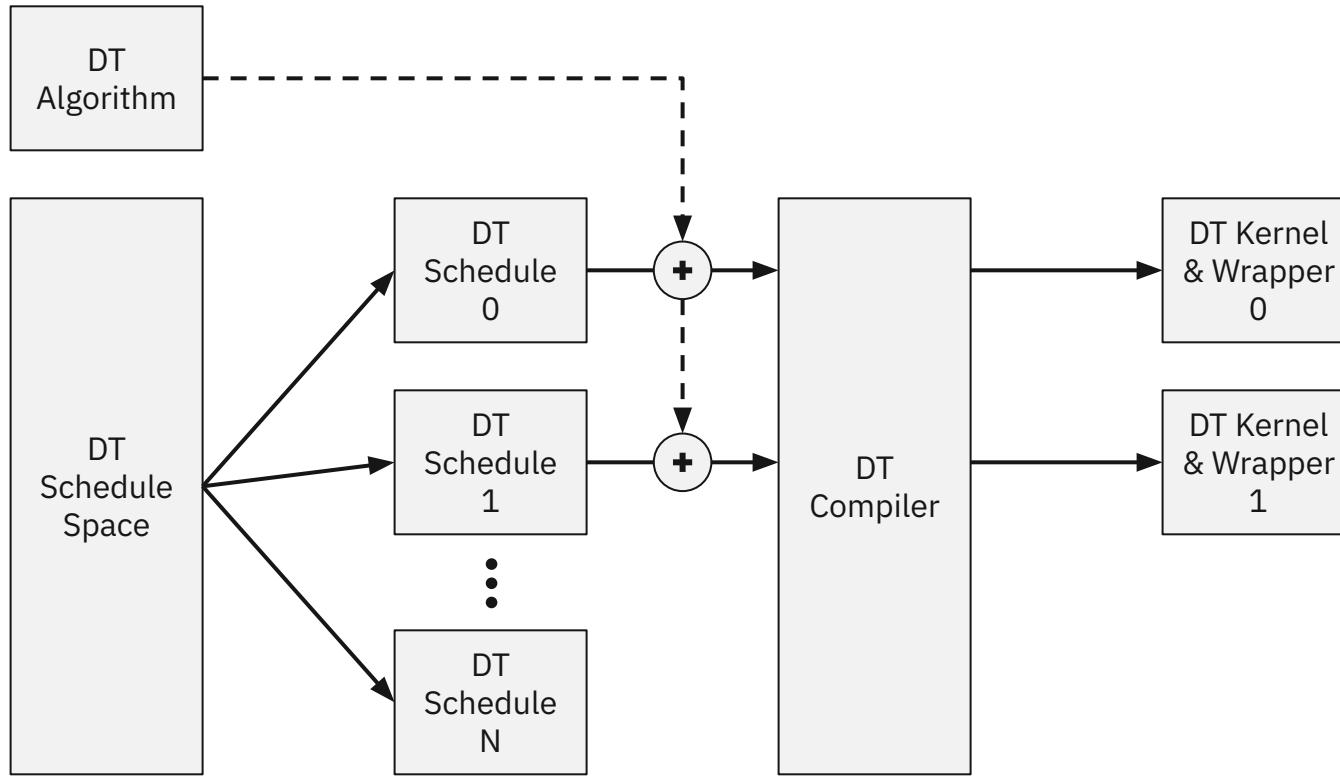
# Experimental Methodology



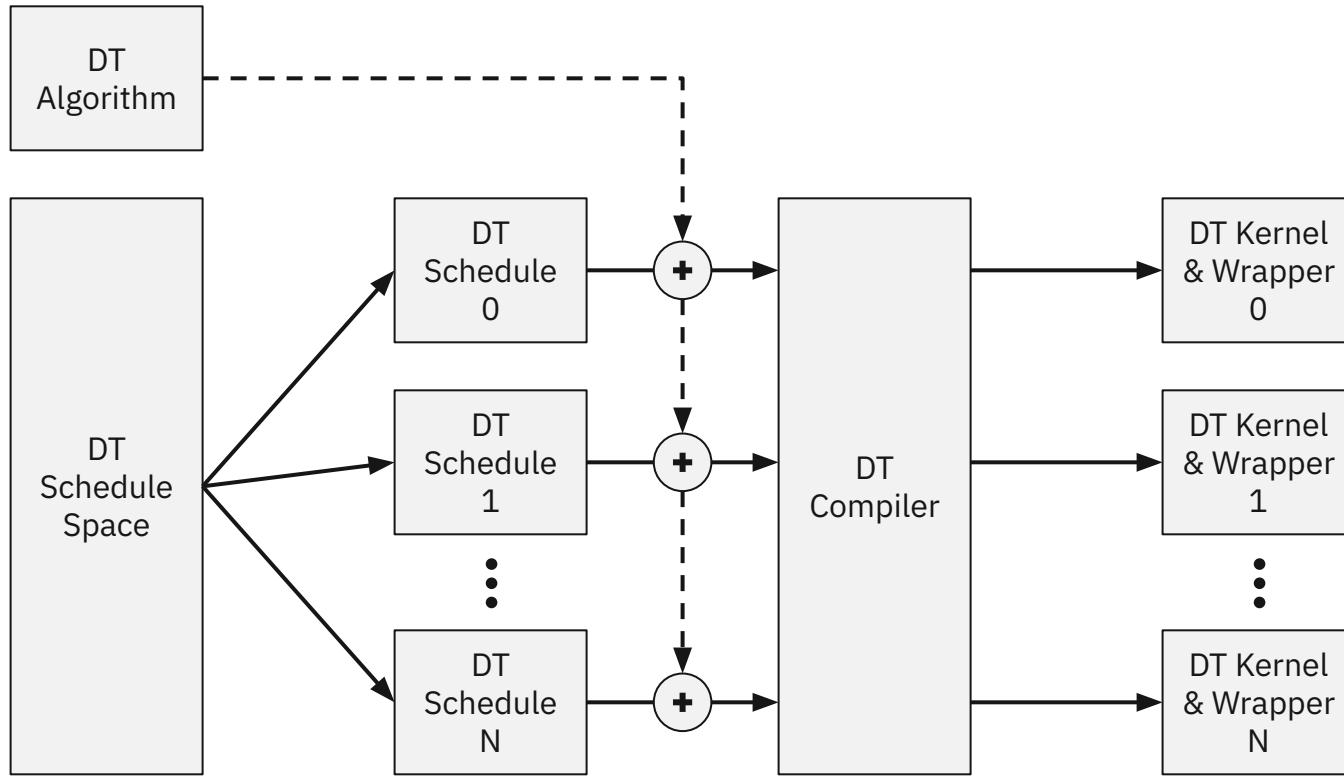
# Experimental Methodology



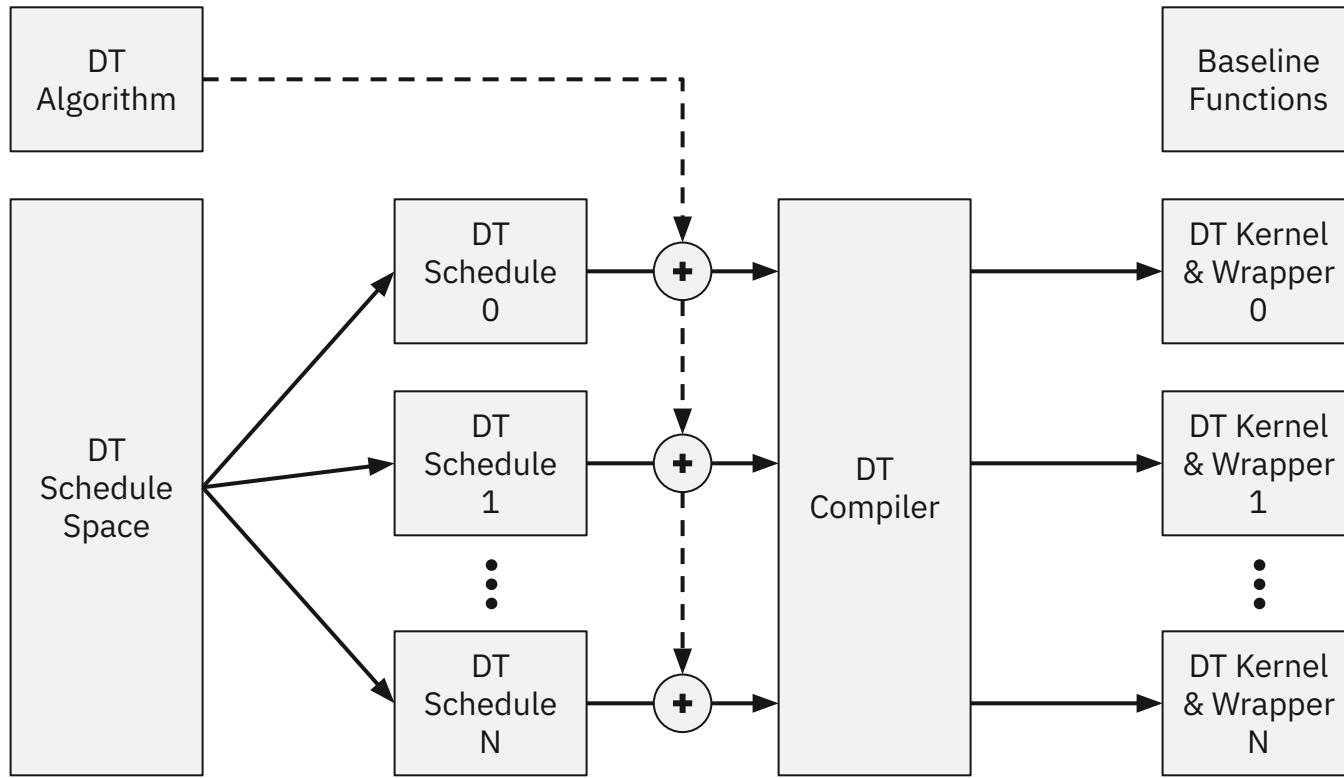
# Experimental Methodology



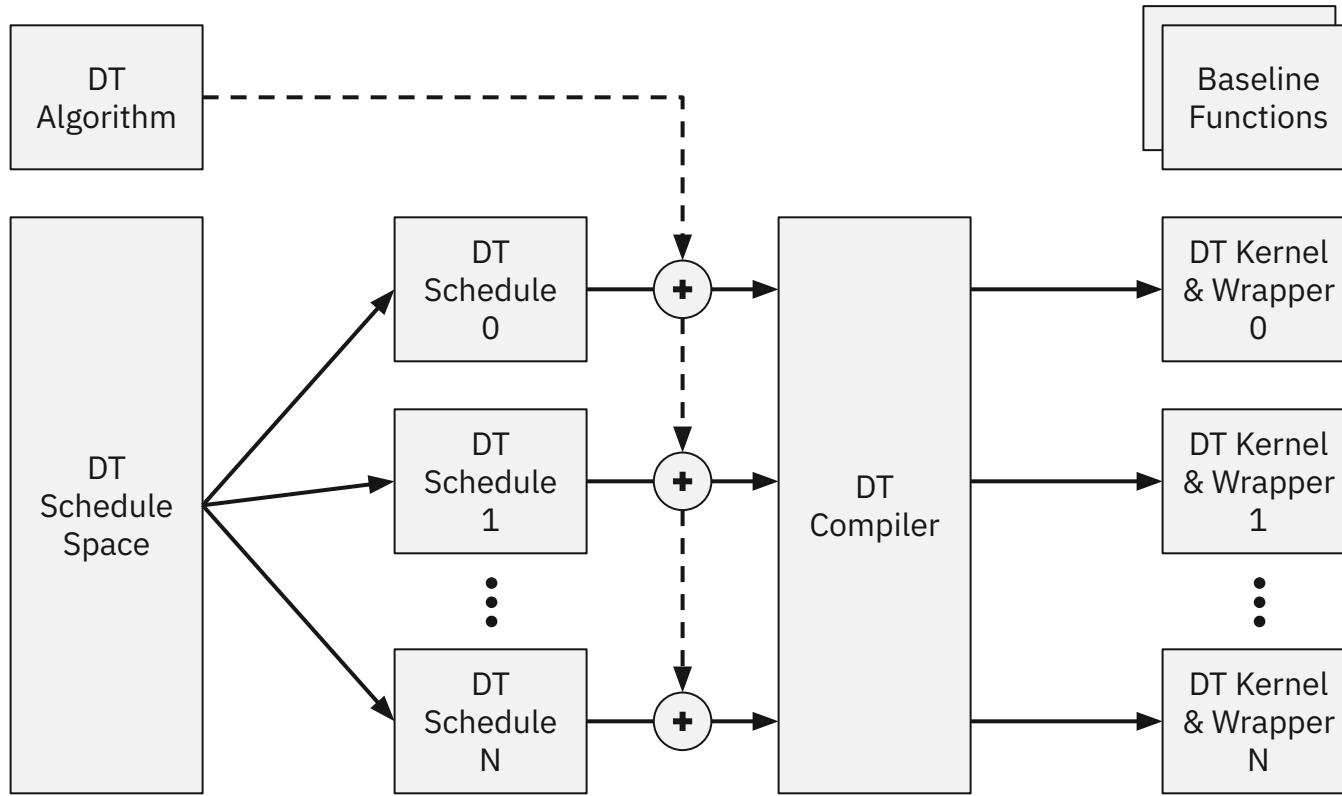
# Experimental Methodology



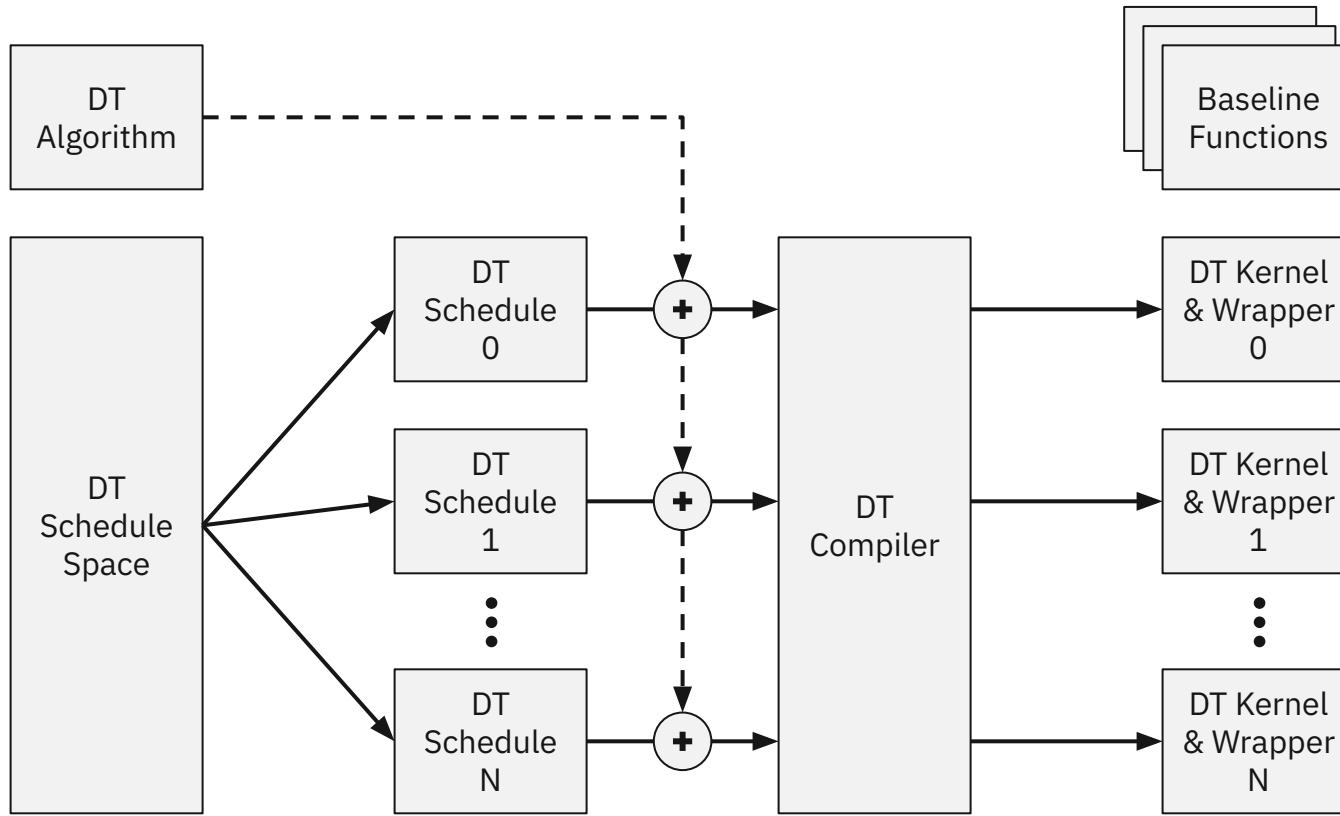
# Experimental Methodology



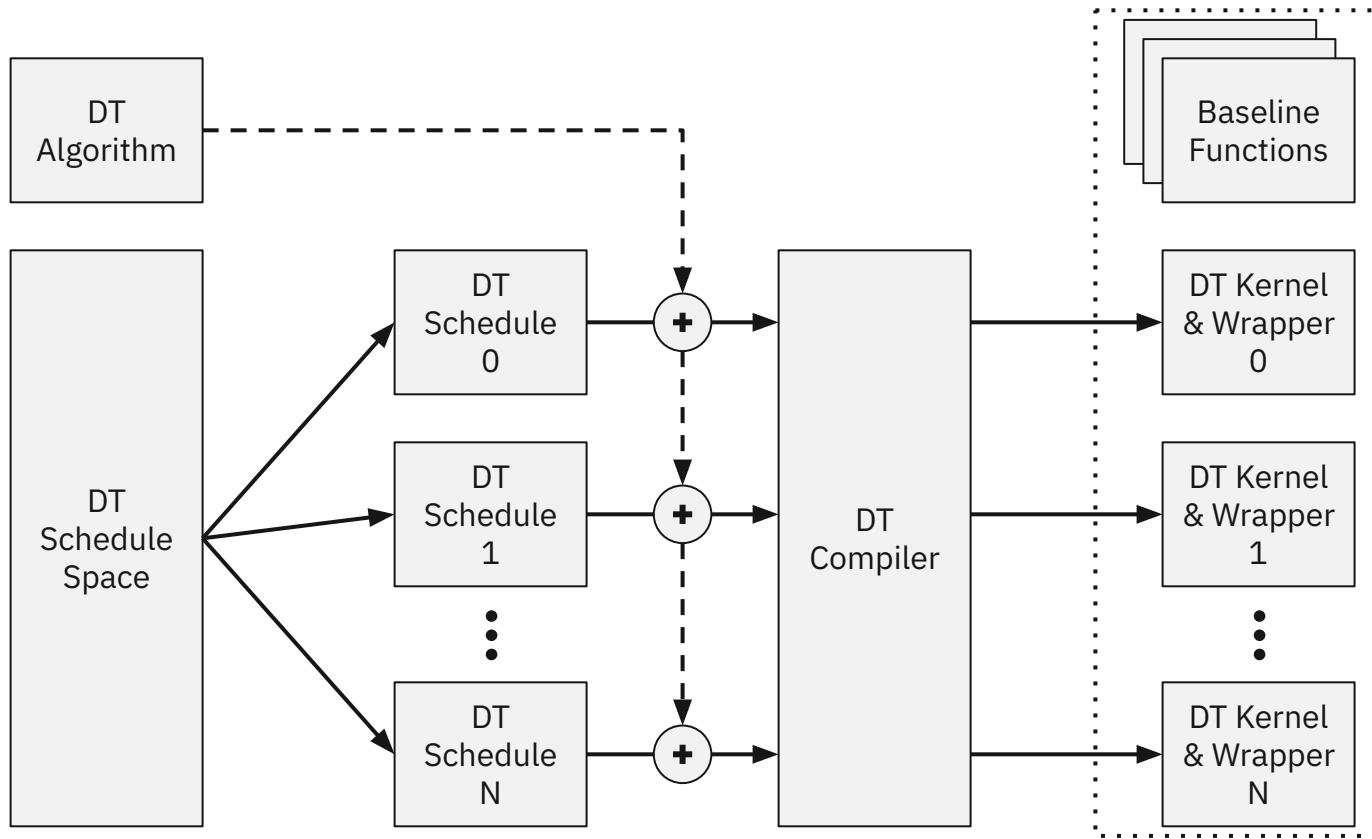
# Experimental Methodology



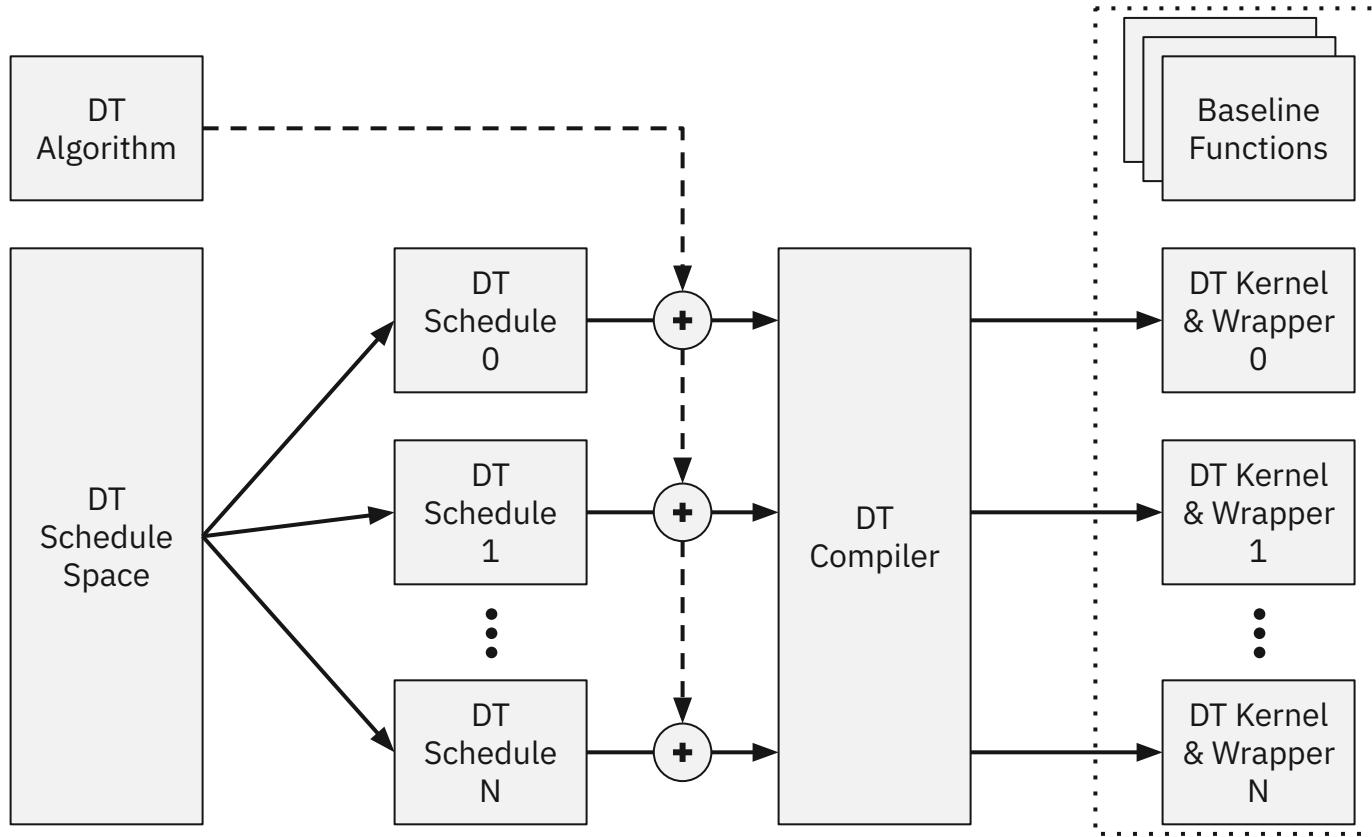
# Experimental Methodology



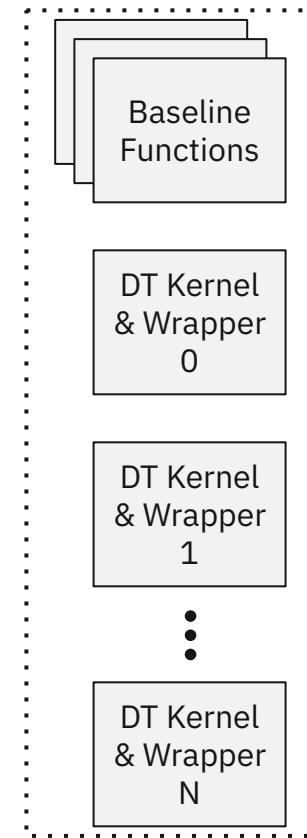
# Experimental Methodology



# Experimental Methodology

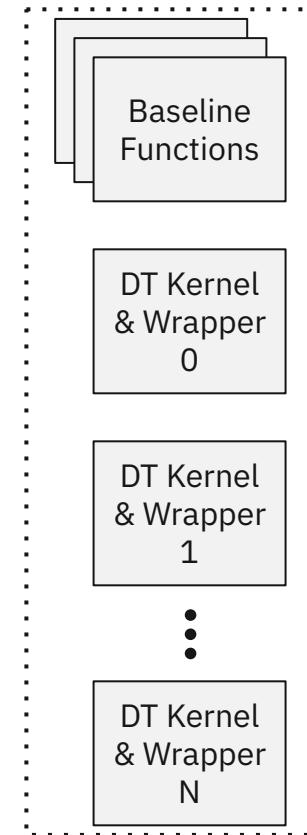


# Experimental Methodology



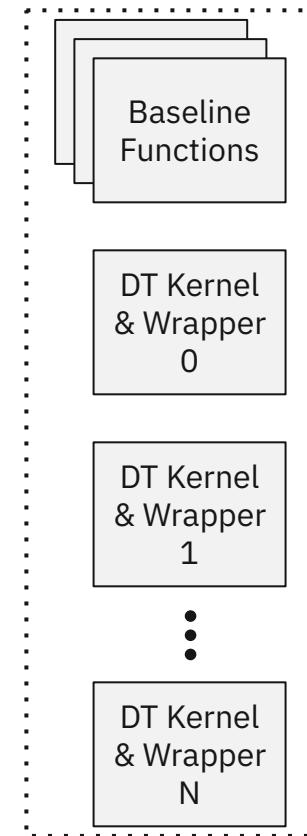
# Experimental Methodology

- Same inputs



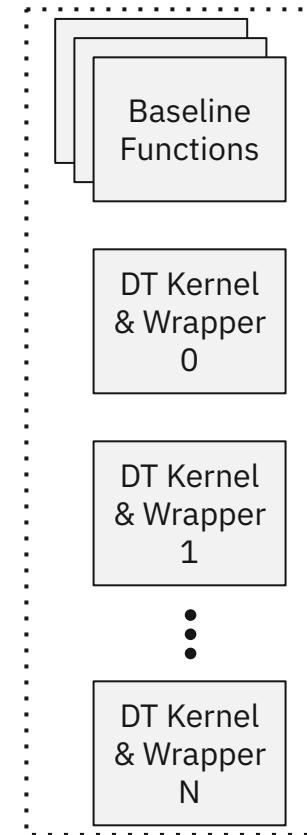
# Experimental Methodology

- Same inputs
- Measure run time



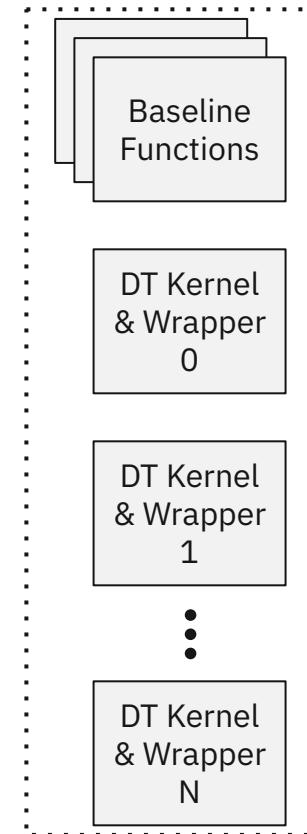
# Experimental Methodology

- Same inputs
- Measure run time
  - `triton.testing.do_bench()`



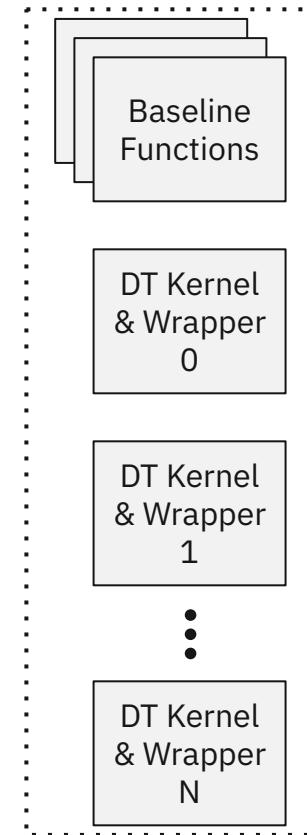
# Experimental Methodology

- Same inputs
- Measure run time
  - `triton.testing.do_bench()`
    - 25 ms warmup



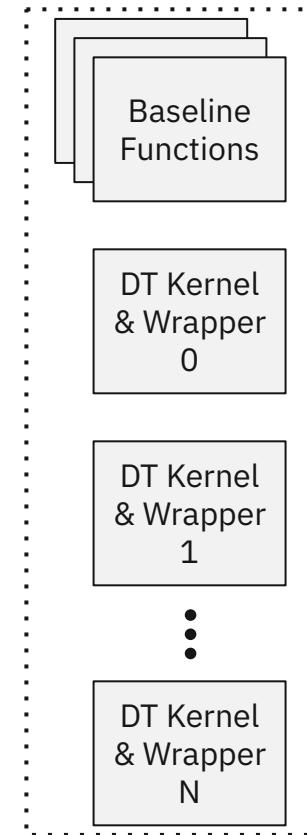
# Experimental Methodology

- Same inputs
- Measure run time
  - `triton.testing.do_bench()`
    - 25 ms warmup
    - 100 ms repetition



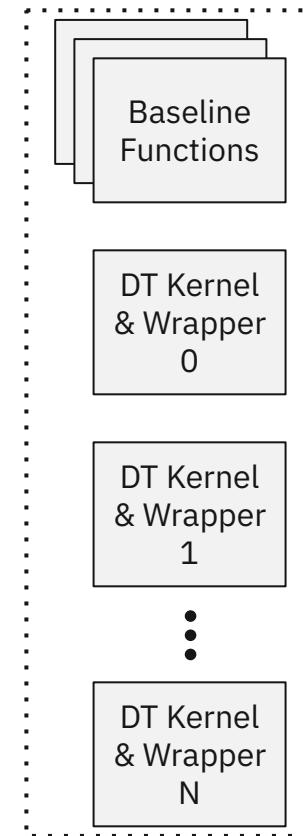
# Experimental Methodology

- Same inputs
- Measure run time
  - `triton.testing.do_bench()`
    - 25 ms warmup
    - 100 ms repetition
  - Mean of 5 iterations



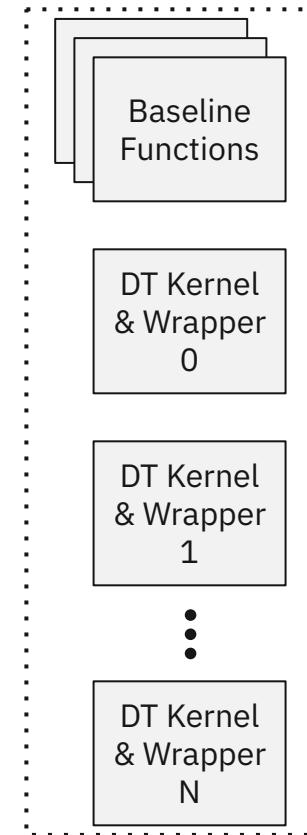
# Experimental Methodology

- Same inputs
- Measure run time
  - `triton.testing.do_bench()`
    - 25 ms warmup
    - 100 ms repetition
  - Mean of 5 iterations
    - Random order within iteration



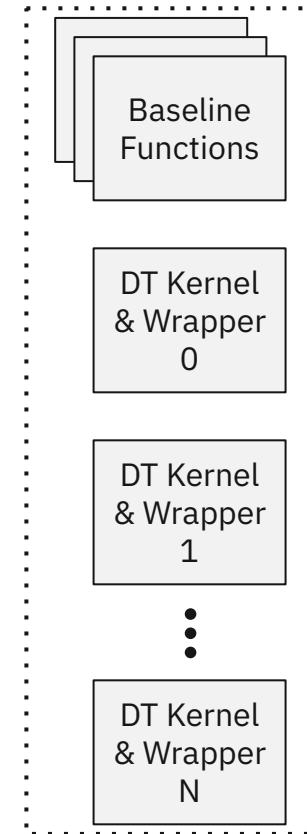
# Experimental Methodology

- Same inputs
- Measure run time
  - `triton.testing.do_bench()`
    - 25 ms warmup
    - 100 ms repetition
  - Mean of 5 iterations
    - Random order within iteration
- Correctness



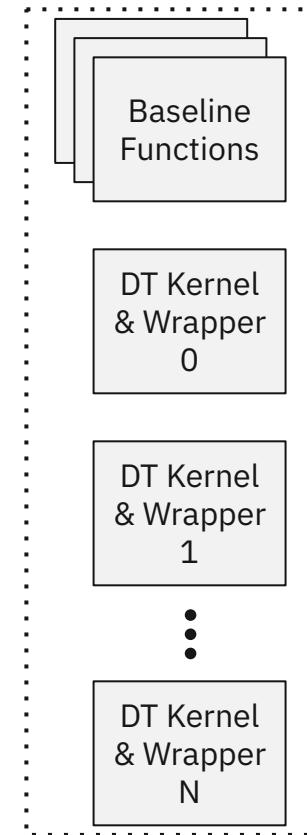
# Experimental Methodology

- Same inputs
- Measure run time
  - `triton.testing.do_bench()`
    - 25 ms warmup
    - 100 ms repetition
  - Mean of 5 iterations
    - Random order within iteration
- Correctness
  - `torch.allclose()`

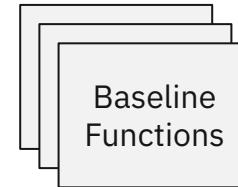


# Experimental Methodology

- Same inputs
- Measure run time
  - `triton.testing.do_bench()`
    - 25 ms warmup
    - 100 ms repetition
  - Mean of 5 iterations
    - Random order within iteration
- Correctness
  - `torch.allclose()`
- NVIDIA RTX 5000 Ada Generation GPU

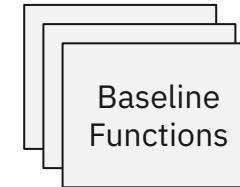


# Baselines



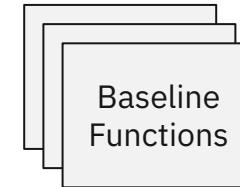
# Baselines

- PyTorch



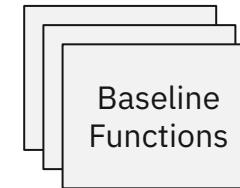
# Baselines

- PyTorch
  - `torch.compile()`



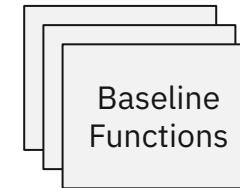
# Baselines

- PyTorch
  - `torch.compile()`
  - `torch.compile(mode="max-autotune")`



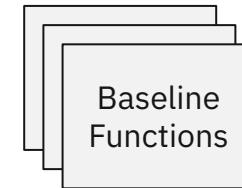
# Baselines

- PyTorch
  - `torch.compile()`
  - `torch.compile(mode="max-autotune")`
- Liger Kernels



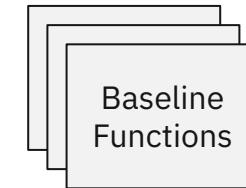
# Baselines

- PyTorch
  - `torch.compile()`
  - `torch.compile(mode="max-autotune")`
- Liger Kernels
  - Open source collection of efficient triton kernels



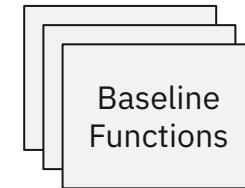
# Baselines

- PyTorch
  - `torch.compile()`
  - `torch.compile(mode="max-autotune")`
- Liger Kernels
  - Open source collection of efficient triton kernels
  - Common LLM operations



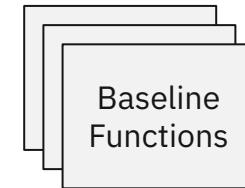
# Baselines

- PyTorch
  - `torch.compile()`
  - `torch.compile(mode="max-autotune")`
- Liger Kernels
  - Open source collection of efficient triton kernels
  - Common LLM operations
- Other Triton Kernel



# Baselines

- PyTorch
  - `torch.compile()`
  - `torch.compile(mode="max-autotune")`
- Liger Kernels
  - Open source collection of efficient triton kernels
  - Common LLM operations
- Other Triton Kernel
  - Expert-written Triton kernel



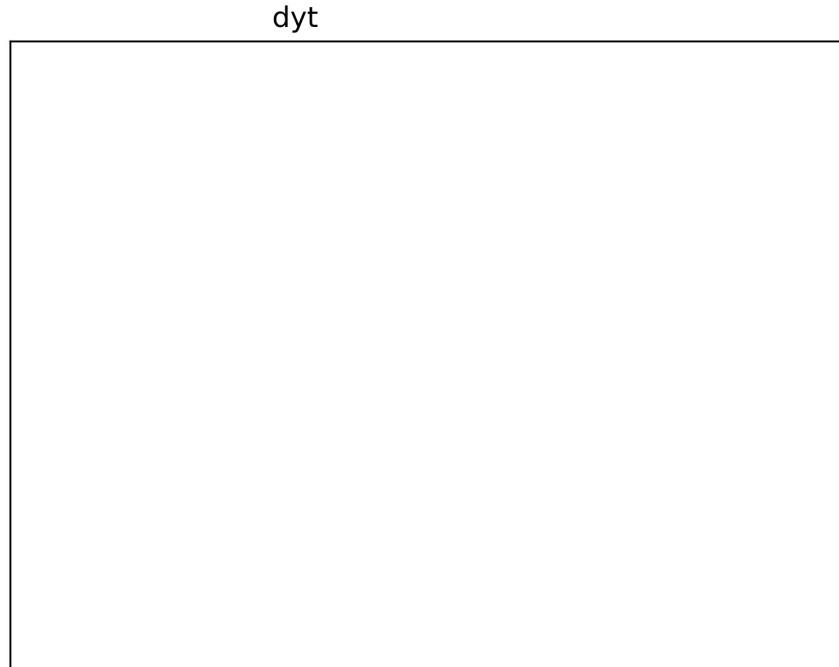
# Results

# Graphs

# Graphs

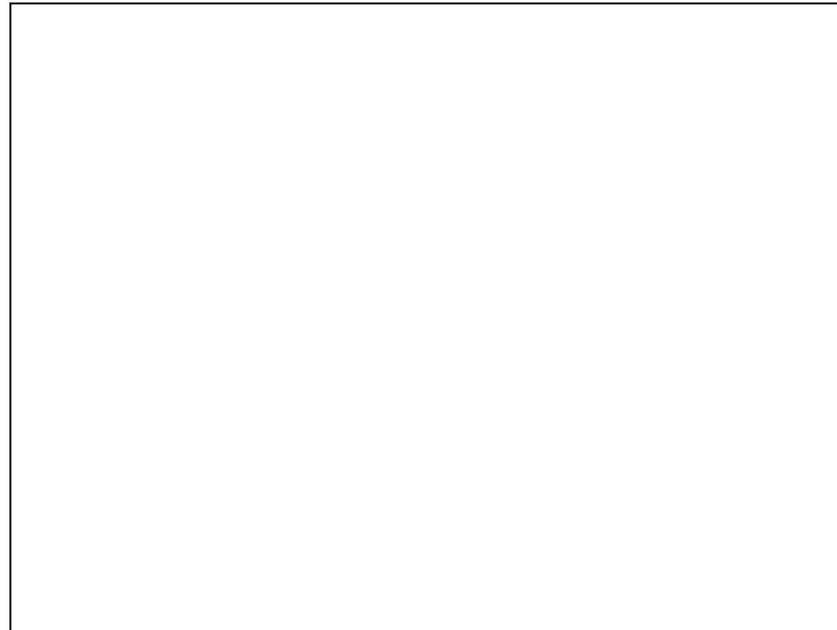


# Graphs

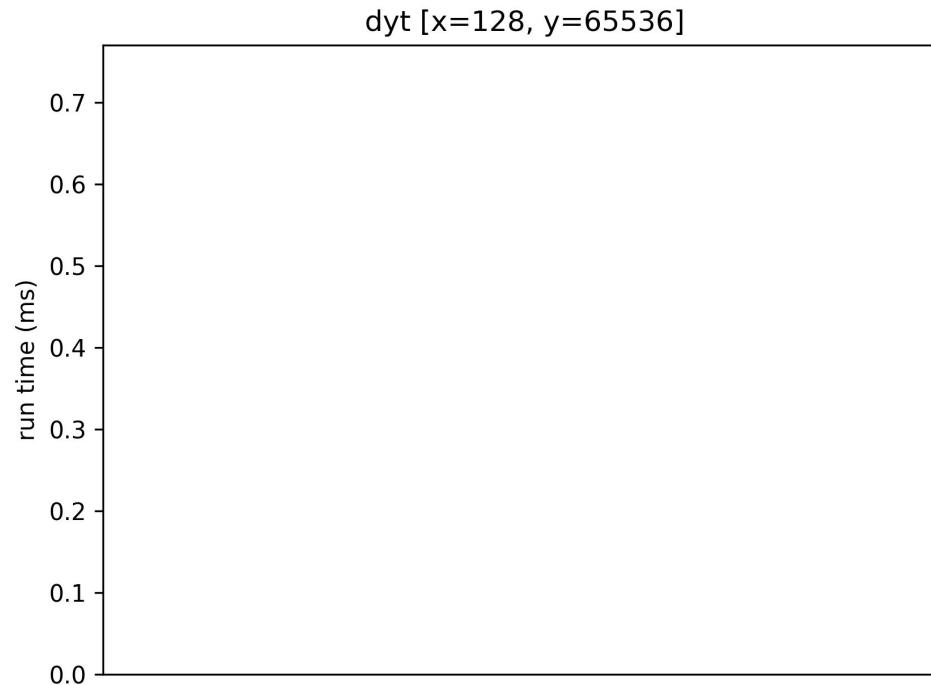


# Graphs

dyt [x=128, y=65536]

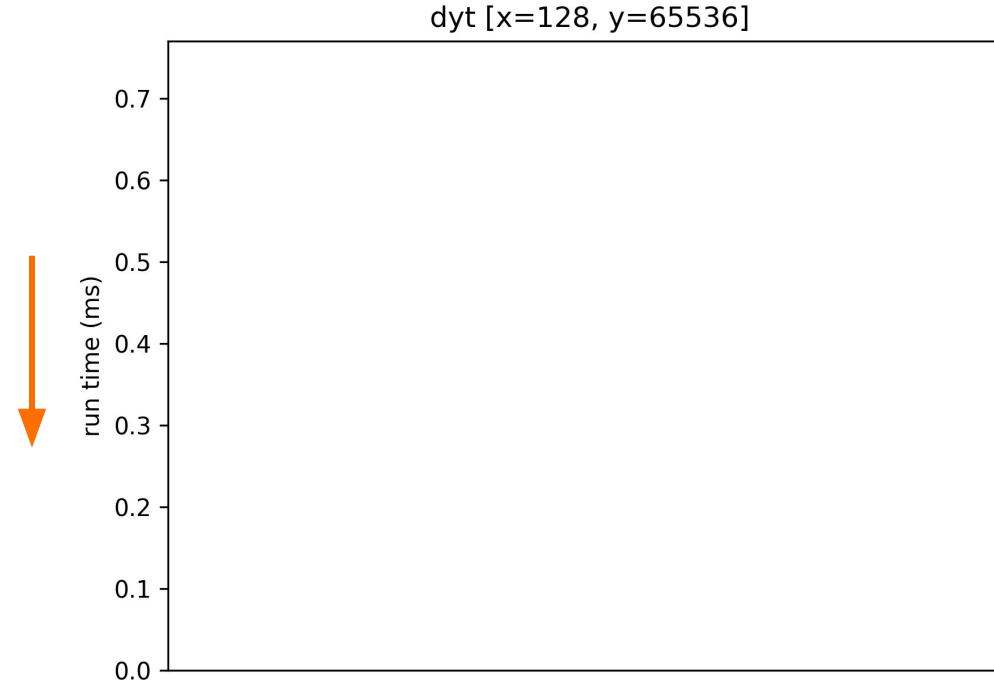


# Graphs

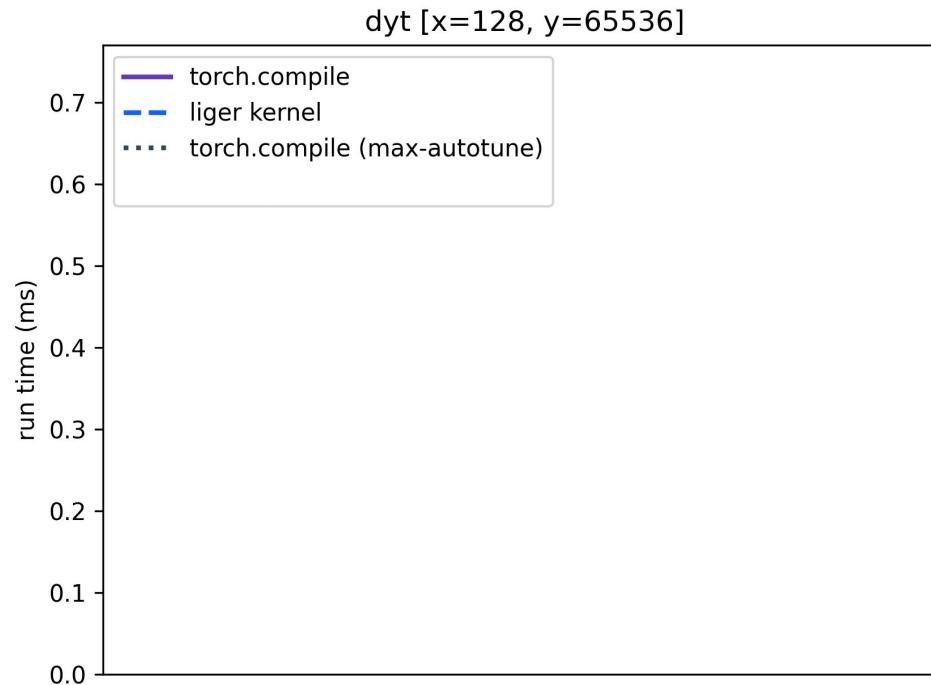


# Graphs

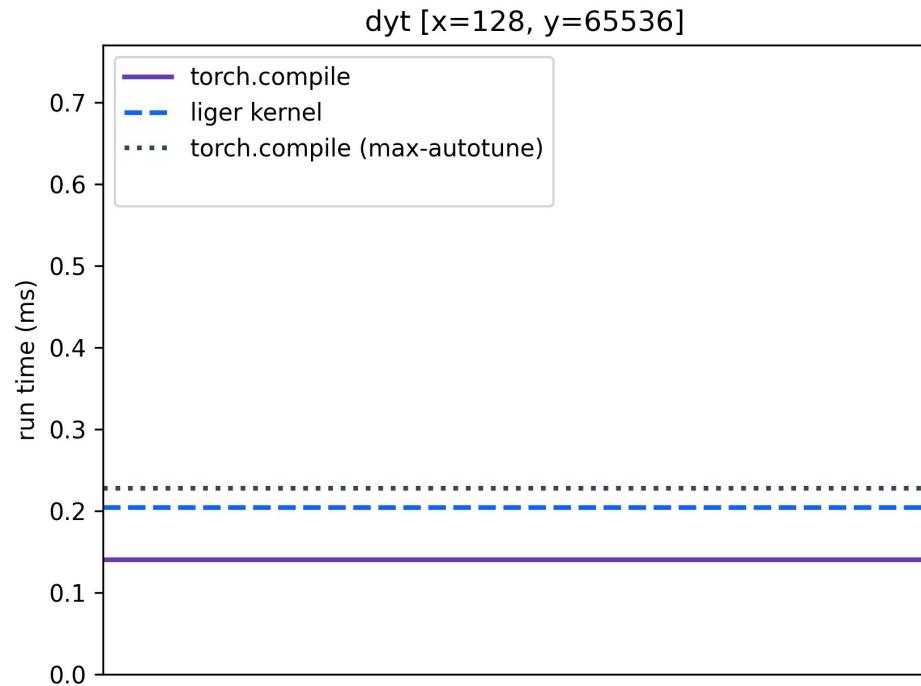
Lower is better!



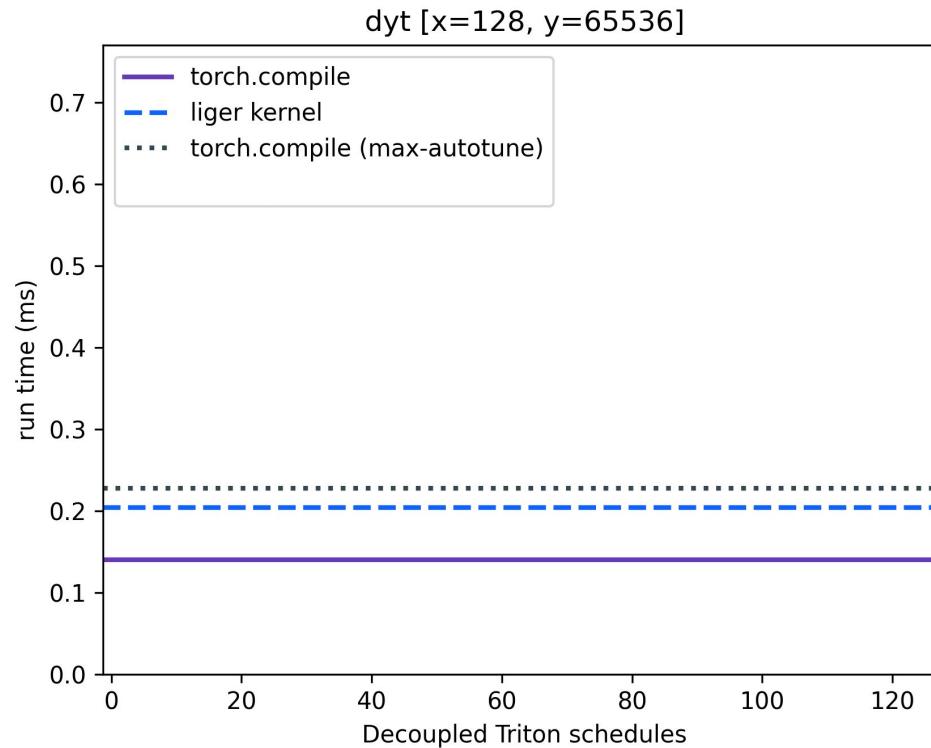
# Graphs



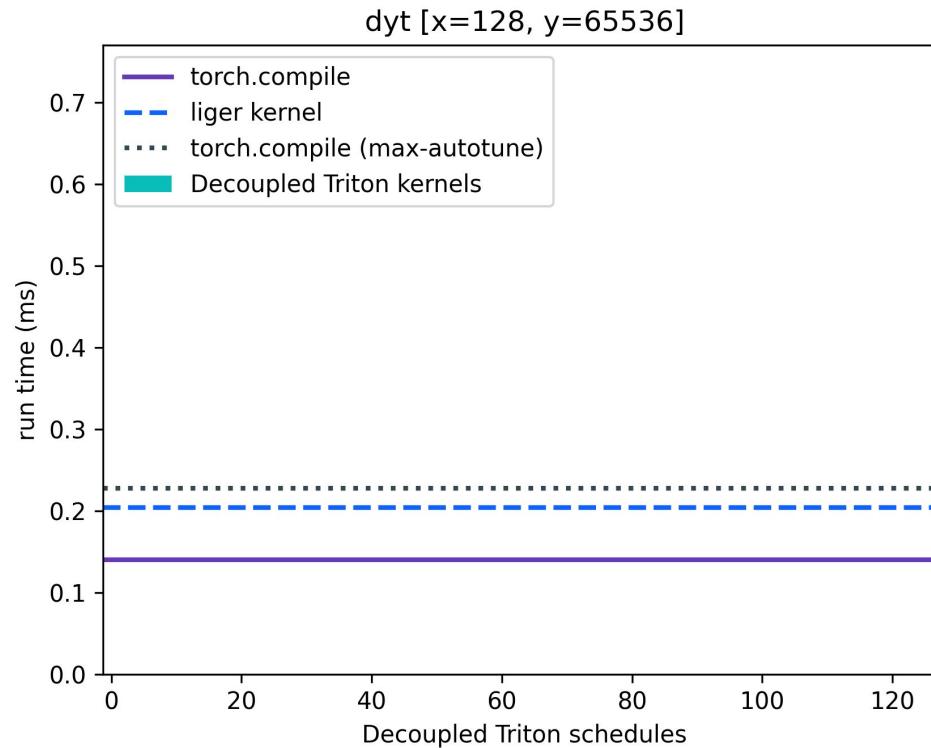
# Graphs



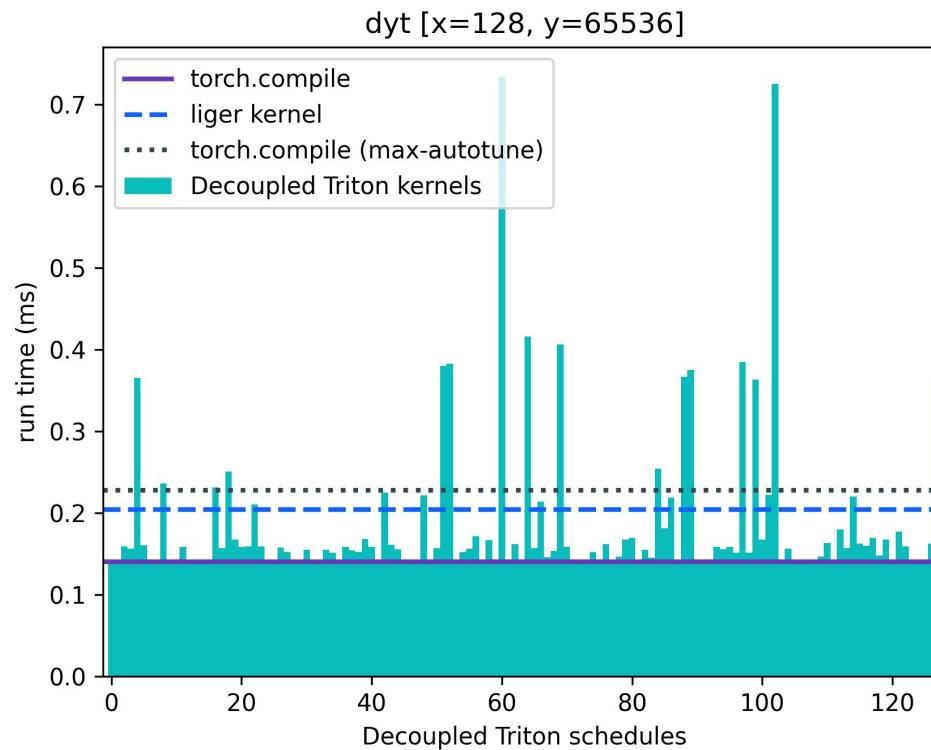
# Graphs



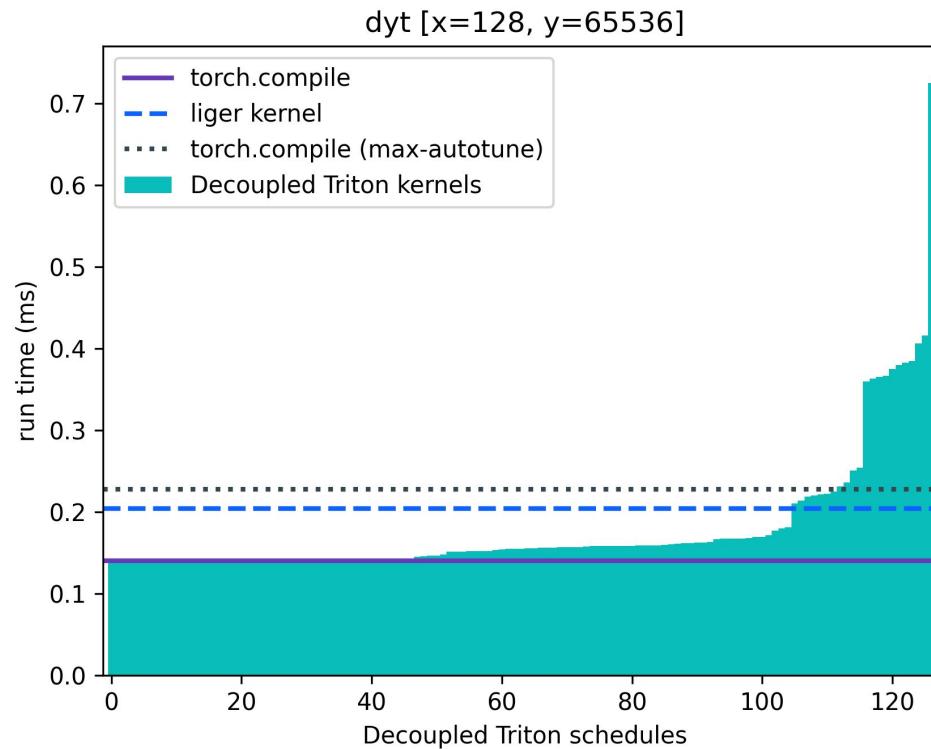
# Graphs



# Graphs

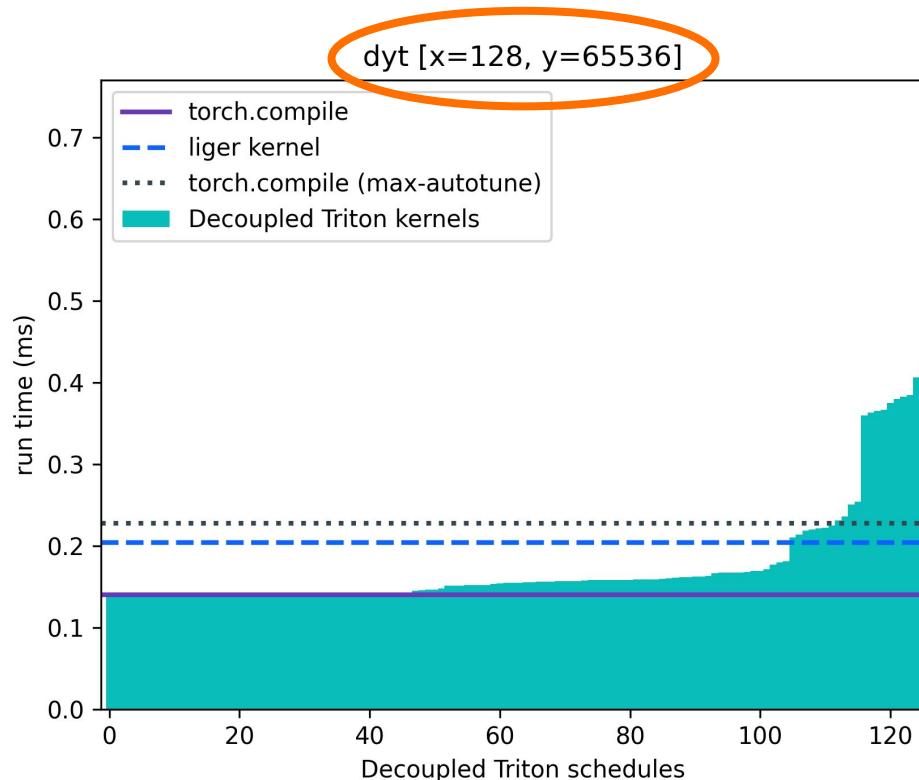


# Graphs



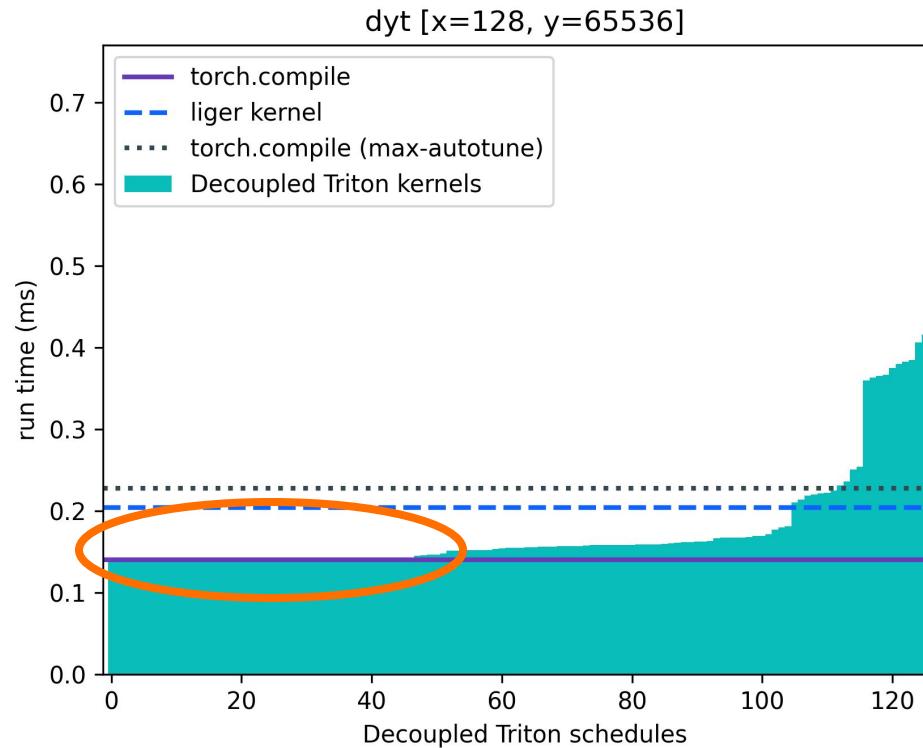
# Graphs

What operation and dimension sizes?

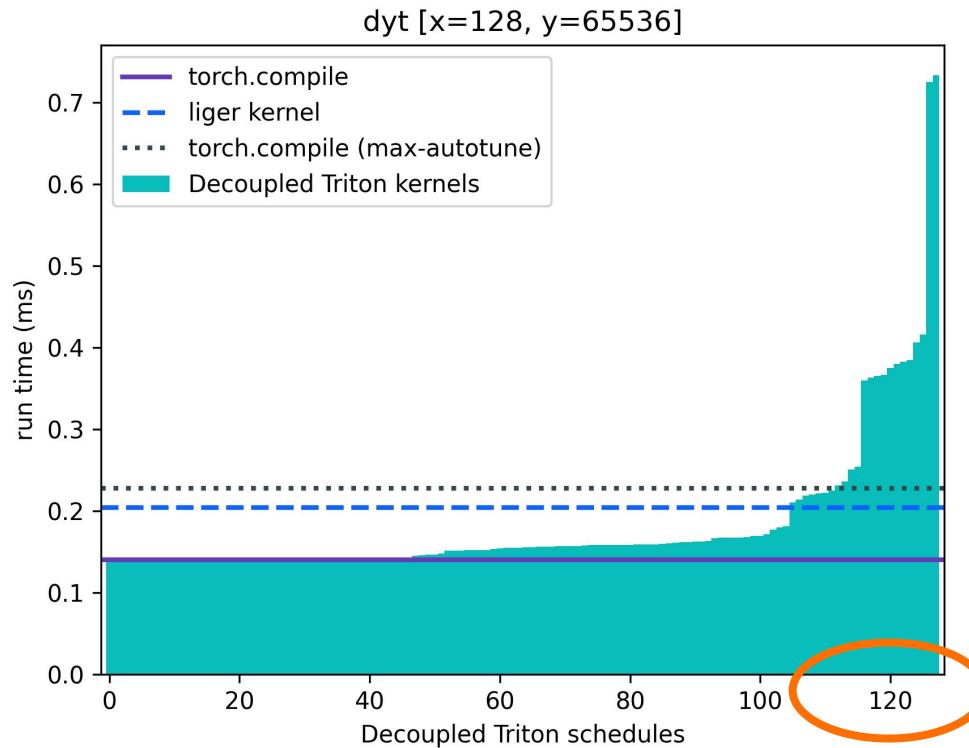


# Graphs

Competitive  
with the  
baselines?



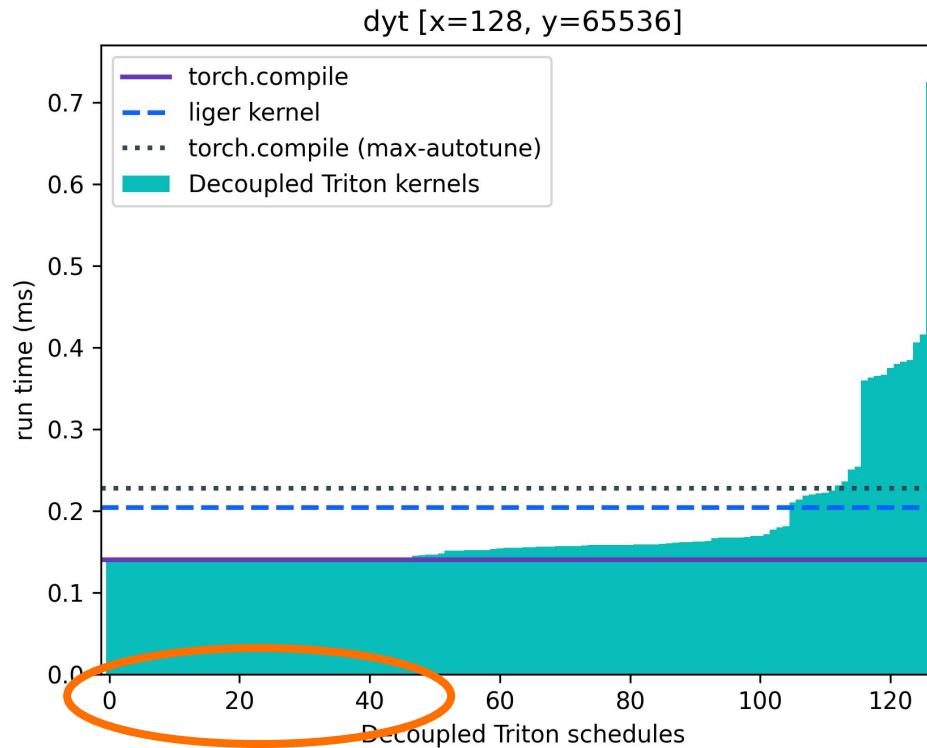
# Graphs



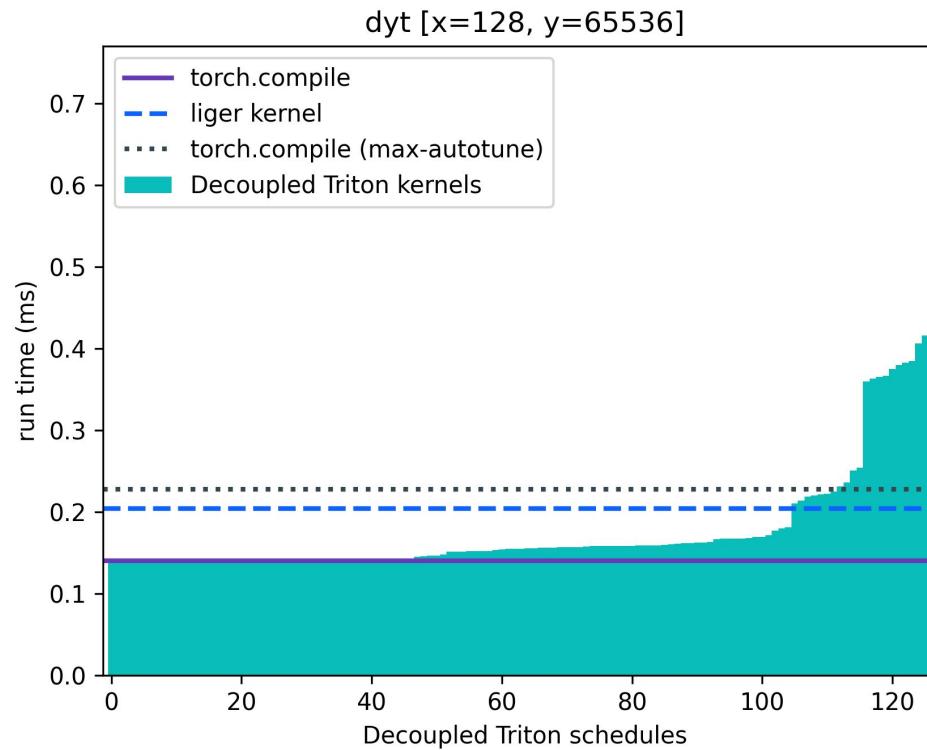
How many  
schedules in  
search space?

# Graphs

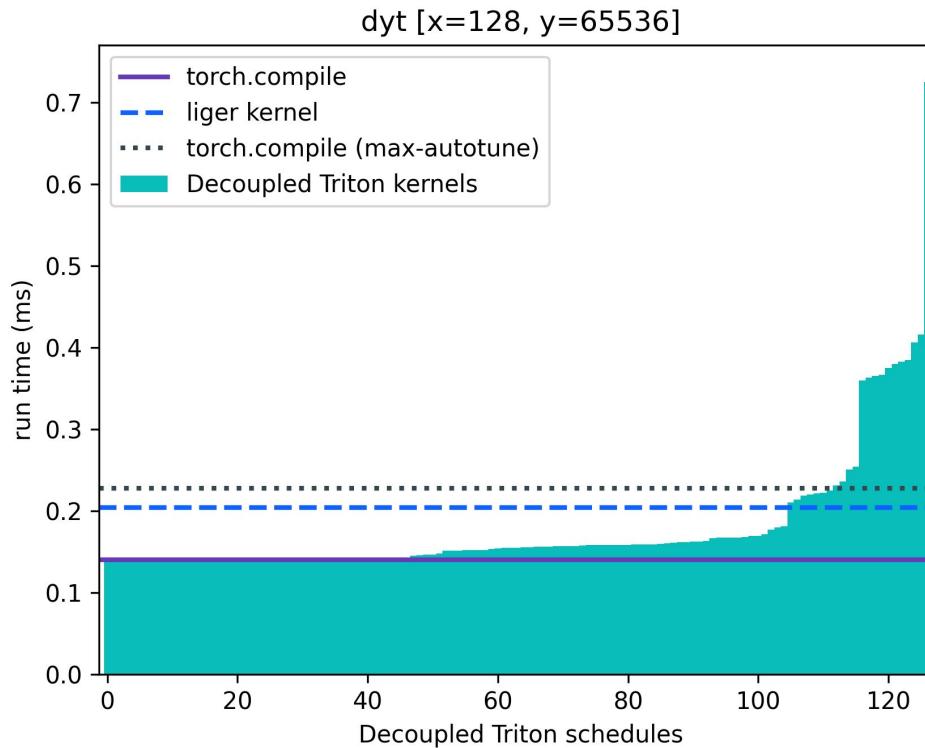
How many schedules are competitive?



# DyT



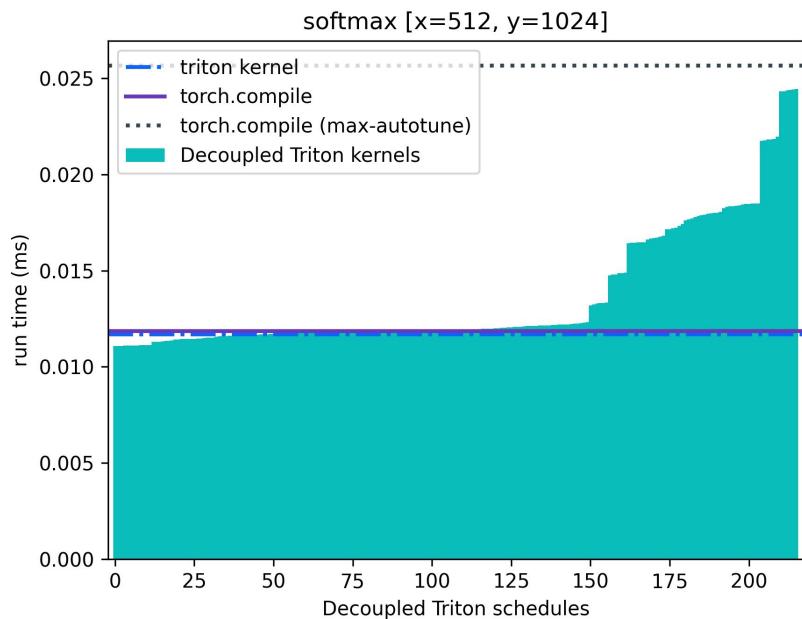
# DyT



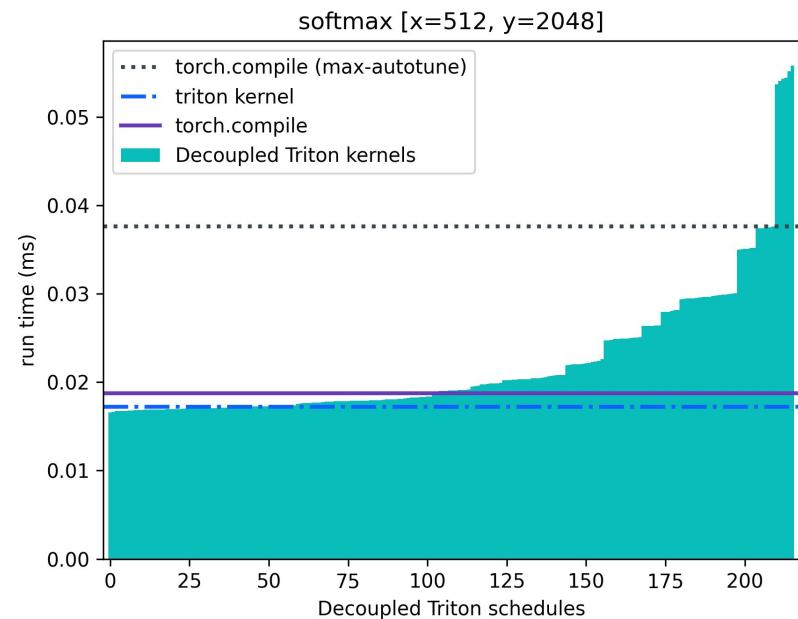
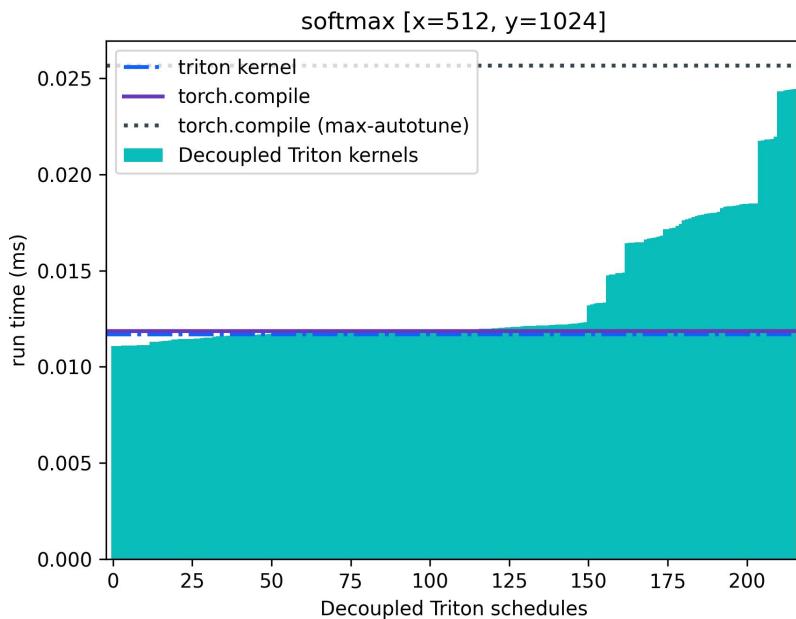
Many liger kernels have a limitations about the innermost dimension size!

# Softmax

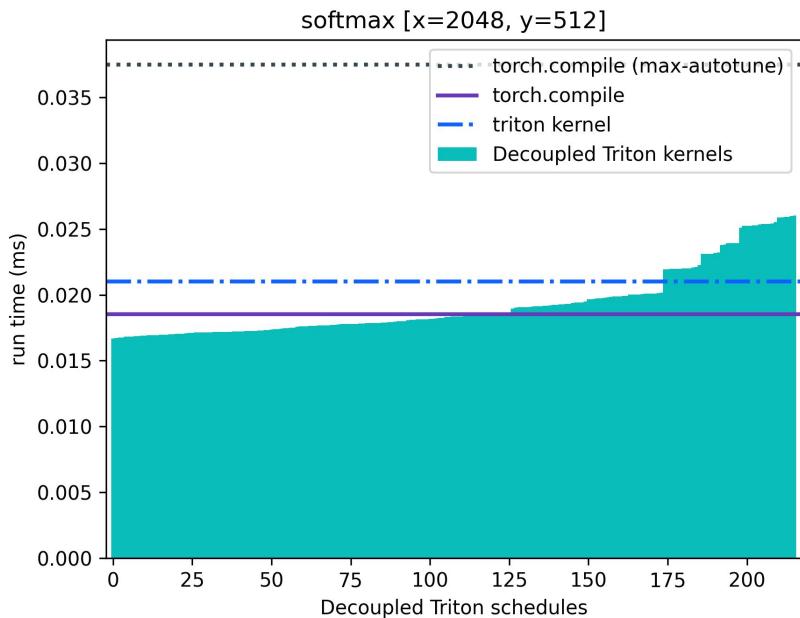
# Softmax



# Softmax



# Softmax

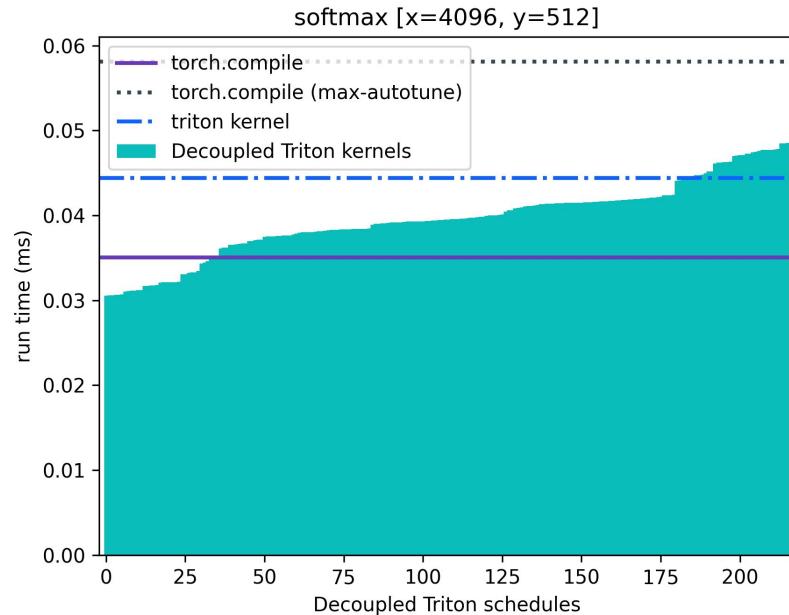


# Softmax

```
Func _softmax, _sum;
In A;
Var x;
RVar y;

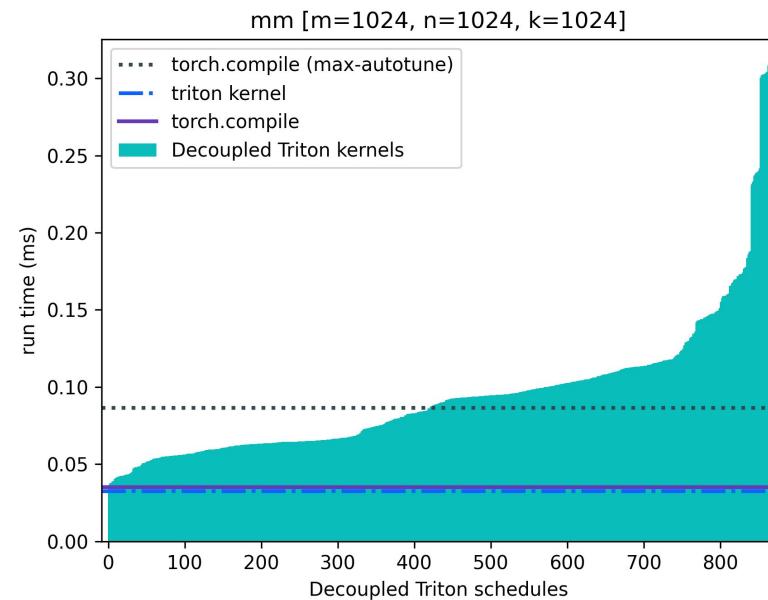
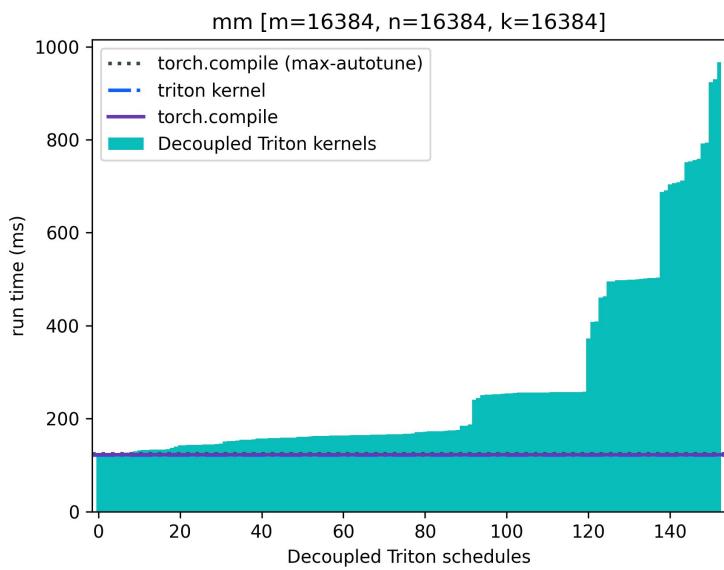
_sum[x]      = rsum(exp(A[x, y]), y);
_softmax[x, y] = exp(A[x, y]) / reshape(_sum[x], x, 1);

_softmax.block(x:4);
_softmax.tensorize(y:512);
_softmax.tensorize(x:0);
_softmax.num_warps(32);
_sum.fuse_at(_softmax, x);
_softmax.compile_to_kernel();
```

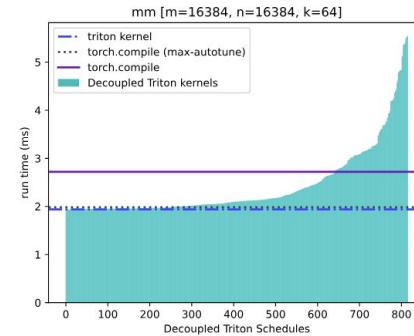
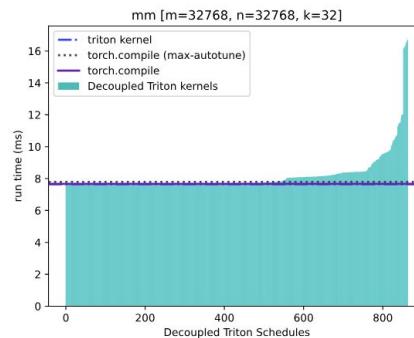
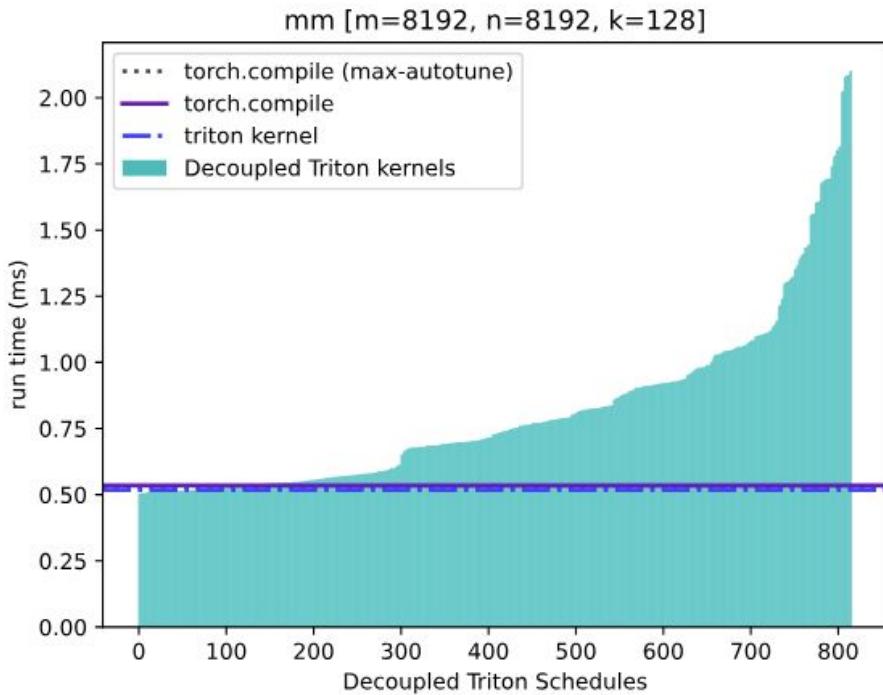


# Matmul

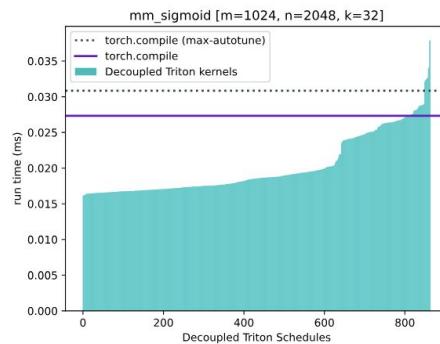
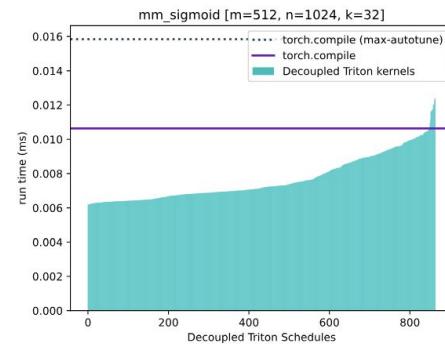
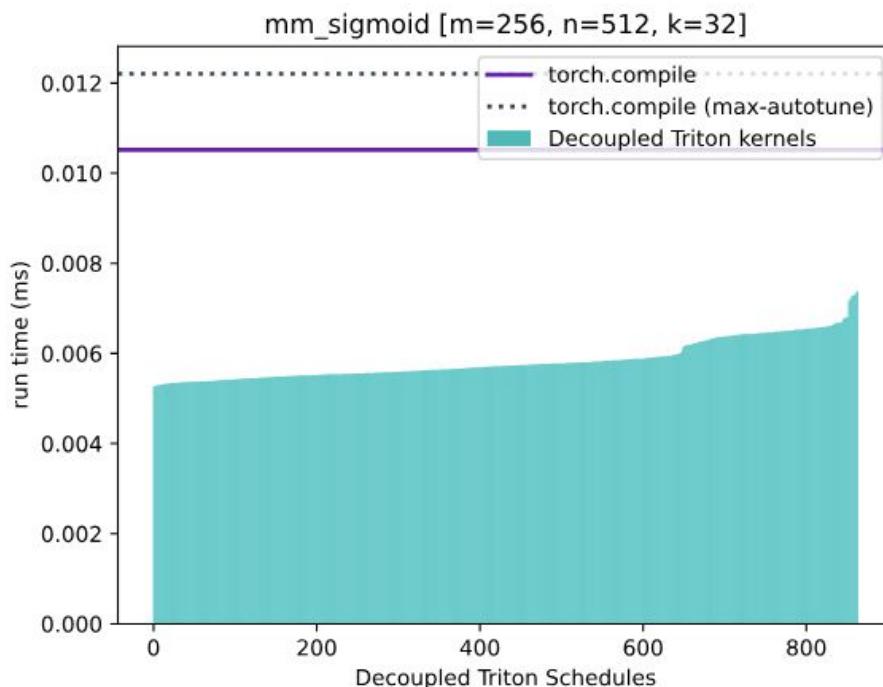
# Matmul



# Matmul



# mmsigmoid



# 2mm (outer-loop fusion)

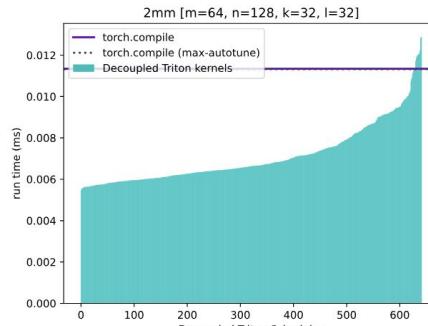
```

Func _2mm, mm;
In A, B, C;
Var m, n;
RVar k, l;

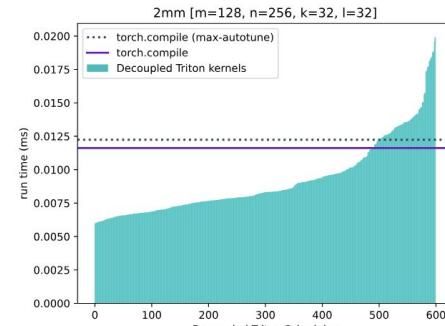
mm[m, 1] = rdot(A[m, k], B[k, 1], k);
_2mm[m, n] = rdot(mm[m, 1], C[l, n], l);

_2mm.block(m:16);
_2mm.tensorize(m:16);
_2mm.tensorize(n:64);
_2mm.tensorize(k:32);
_2mm.tensorize(1:0);
_2mm.num_stages(4);
_2mm.num_warps(4);
mm.fuse_at(_2mm, m);
_2mm.compile_to_kernel();

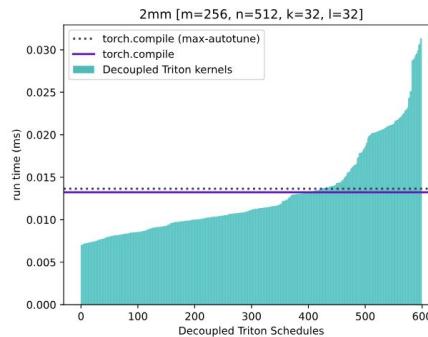
```



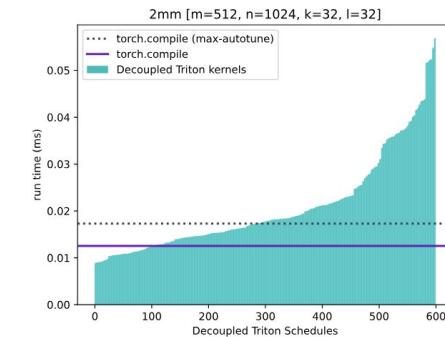
(a) m=64, n=128, k=32, l=32



(b) m=128, n=256, k=32, l=32

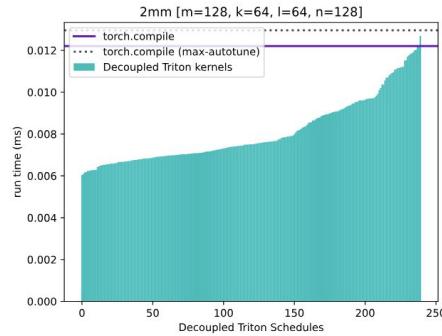


(c) m=256, n=512, k=32, l=32

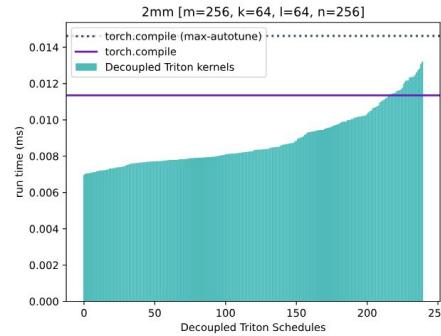


(d) m=512, n=1024, k=32, l=32

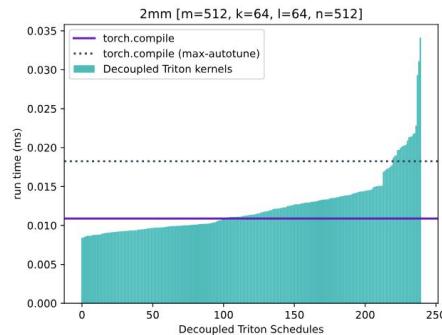
# 2mm (inner-loop fusion)



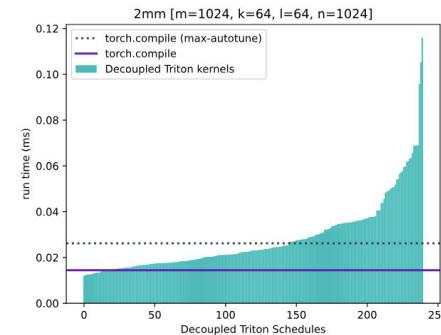
(a)  $m=128, k=64, l=64, n=128$



(b)  $m=256, k=64, l=64, n=256$



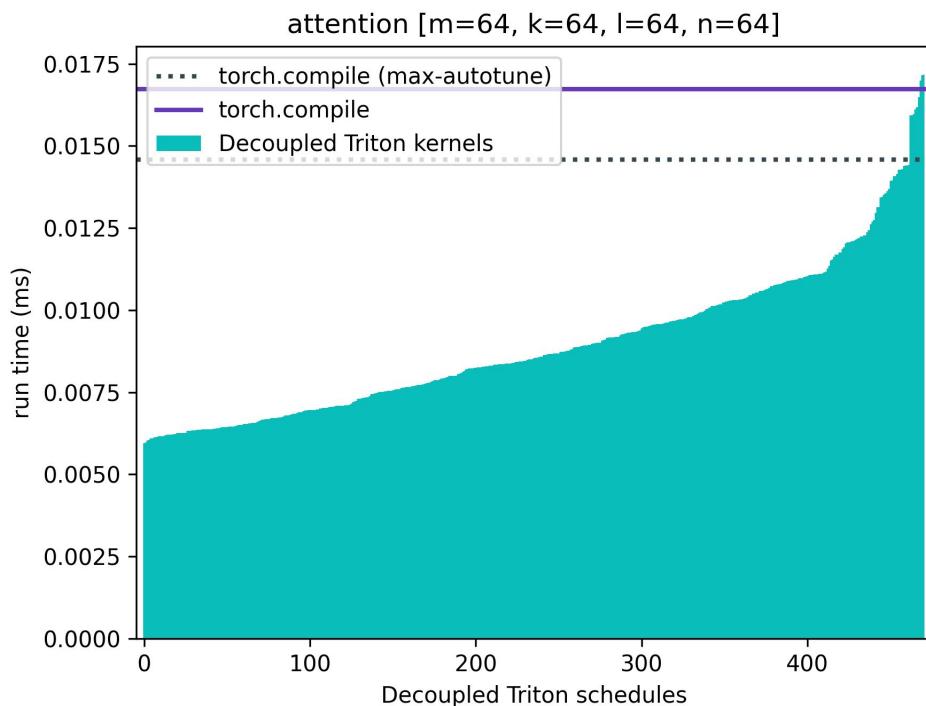
(c)  $m=512, k=64, l=64, n=512$



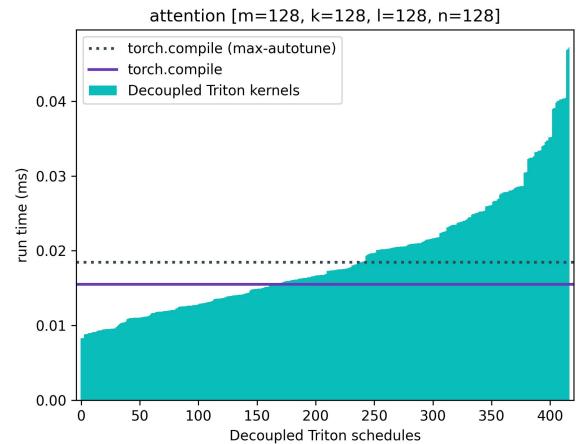
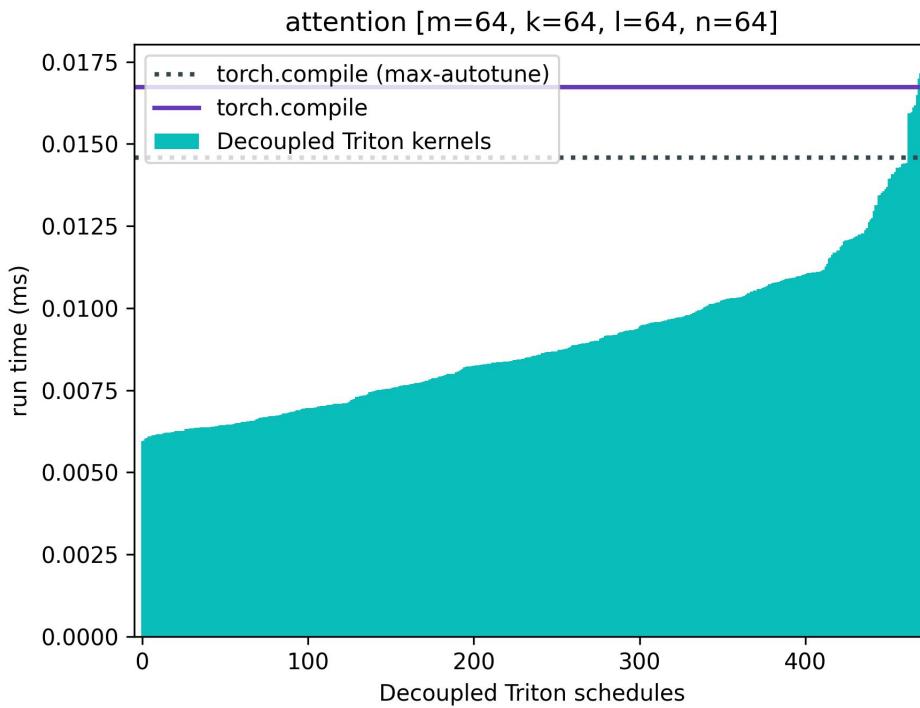
(d)  $m=1024, k=64, l=64, n=1024$

# Attention

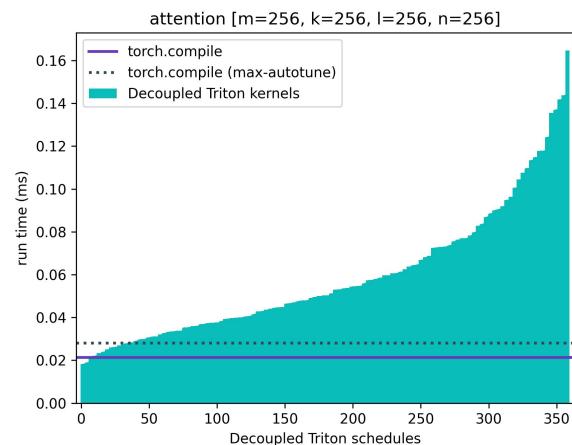
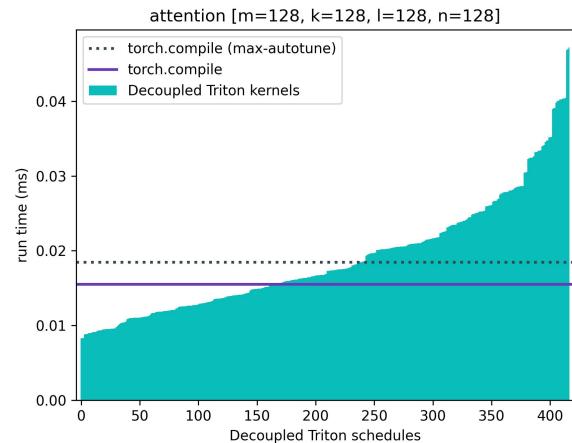
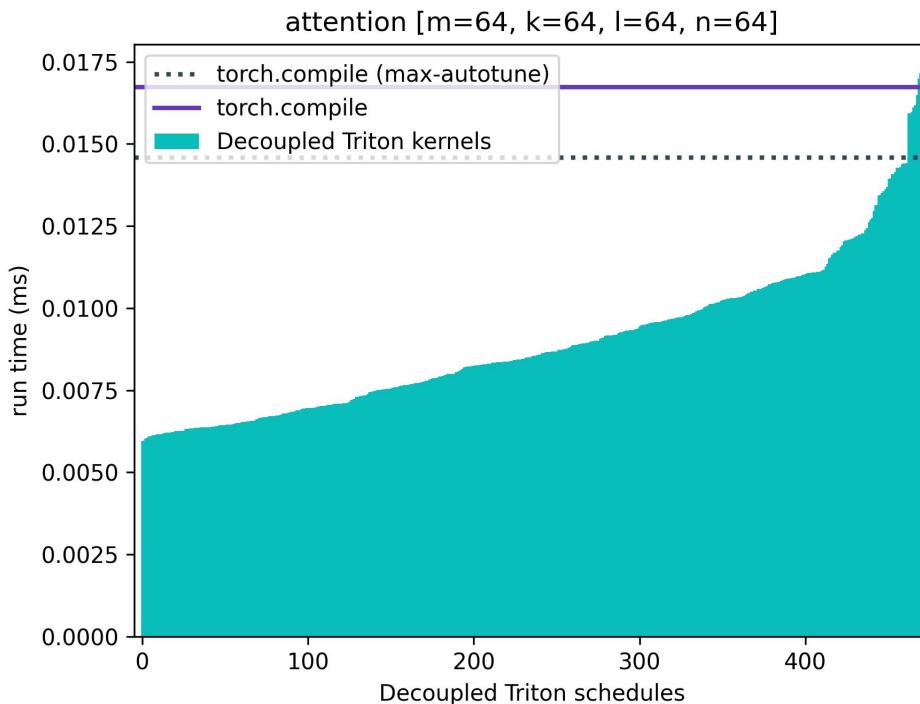
# Attention



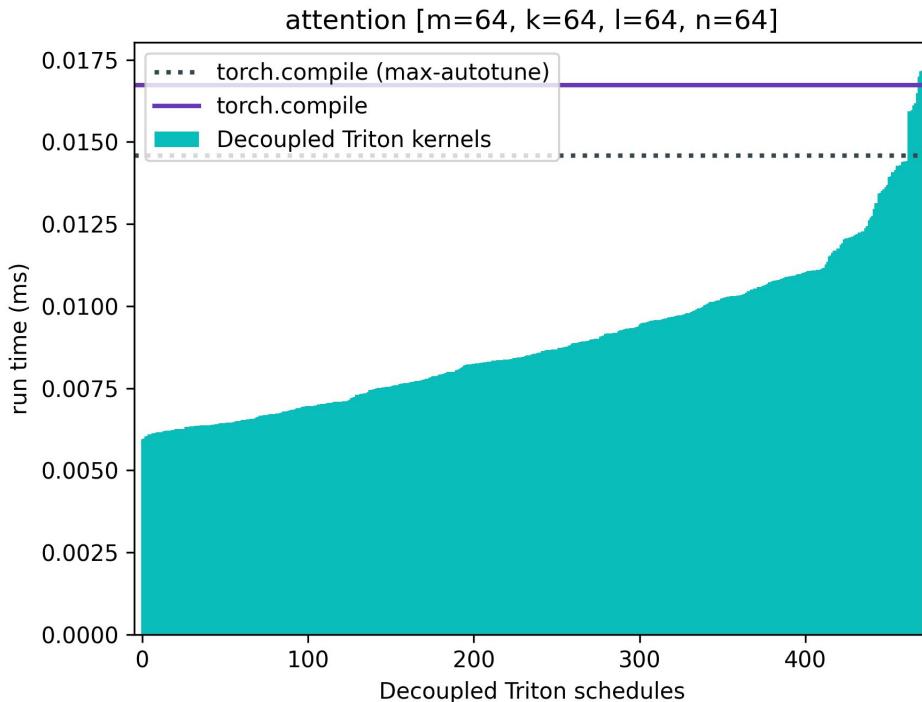
# Attention



# Attention



# Attention



```
Func attention, mm, e, sm, dvsr;
In  A, B, C;
Var m, n;
RVar k, l;

mm[m, 1]      = rdot(A[m, k], B[k, 1], k) / sqrt(len(l));
e[m, 1]        = exp(mm[m, 1]);
dvsr[m]        = rsum(e[m, 1], 1);
sm[m, 1]        = e[m, 1] / reshape(dvsr[m], m, 1);
attention[m, n] = rdot(sm[m, 1], C[1, n], 1);

attention.tensorize(m:16);
attention.block(m:16);
attention.tensorize(n:64);
attention.tensorize(k:16);
attention.tensorize(l:0);
attention.num_stages(8);
attention.num_warps(8);
mm.fuse_at(e, 1);
dvsr.fuse_at(sm, m);
sm.fuse_at(attention, m);
e.fuse_at(attention, m);
attention.compile();
```

# Discussion

# Discussion

- Decoupled Triton can always compete with expert-written Triton kernels and PyTorch.

# Discussion

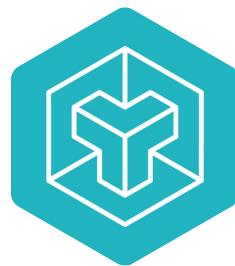
- Decoupled Triton can always compete with expert-written Triton kernels and PyTorch.
- Generally, it is easy to find an efficient schedule for an operation.

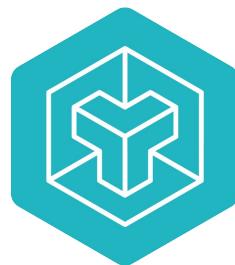
# Discussion

- Decoupled Triton can always compete with expert-written Triton kernels and PyTorch.
- Generally, it is easy to find an efficient schedule for an operation.
- Decoupled Triton can outperform expert-written Triton kernels and PyTorch when the shapes are unusual or when the existing solutions cannot implement efficient fusions.

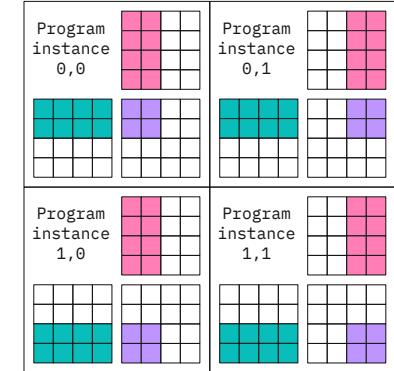
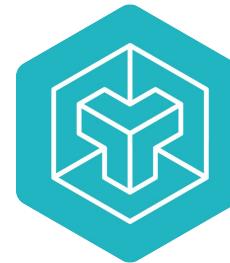
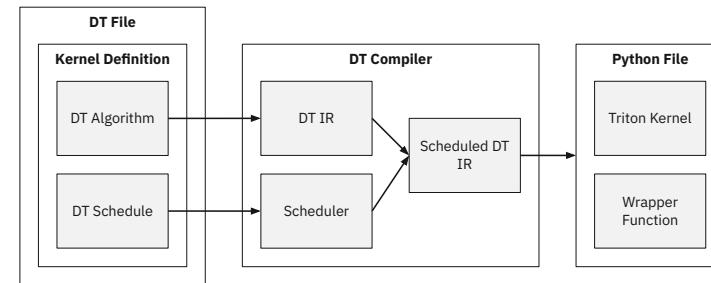


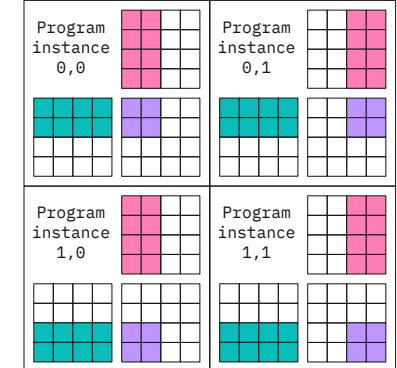
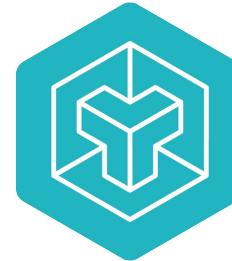
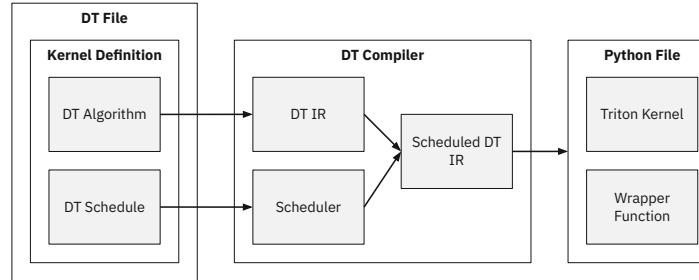






Program instance 0,0	Program instance 0,1	Program instance 1,0	Program instance 1,1





```

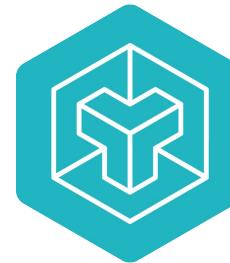
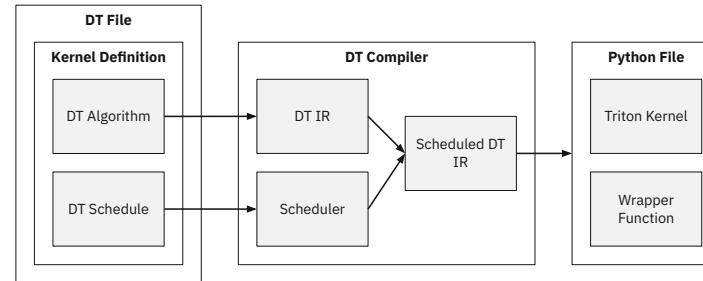
# Declarations
Func mm;
In A, B;
Var x, y;
RVar k;

# Algorithm
mm[x, y] = rdot(A[x, k], B[k, y], k);

# Schedule
mm.tensorize(x:128, k:32, y:128);
mm.block(x:128, y:128);
mm.map(x:x1/8, y, x1);
mm.num_warps(4);
mm.num_stages(3);
mm.compile();

```





0	0	0	0
0	0	0	0

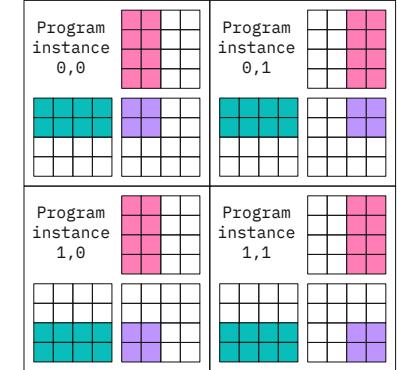
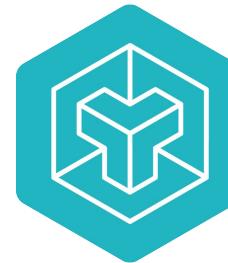
1	1	1	1
1	1	1	1

```
# Declarations
Func mm;
In A, B;
Var x, y;
RVar k;

# Algorithm
mm[x, y] = rdot(A[x, k], B[k, y], k);

# Schedule
mm.tensorize(x:128, k:32, y:128);
mm.block(x:128, y:128);
mm.map(x:x/8, y, x1);
mm.num_warps(4);
mm.num_stages(3);
mm.compile();
```

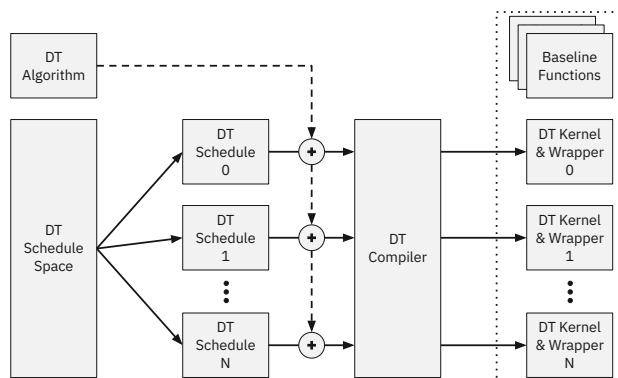
Program instance 0,0		Program instance 0,1	
Program instance 1,0		Program instance 1,1	



```
# Declarations
Func mm;
In A, B;
Var x, y;
RVar k;
```

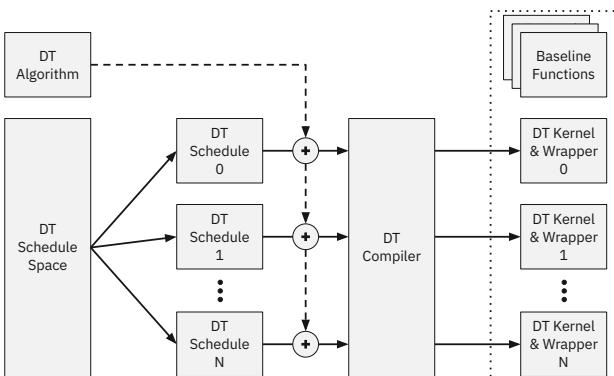
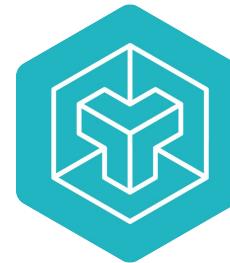
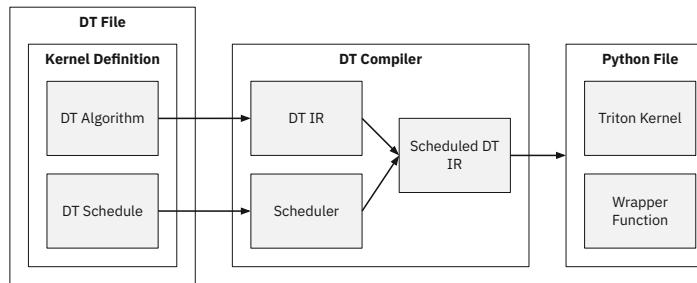
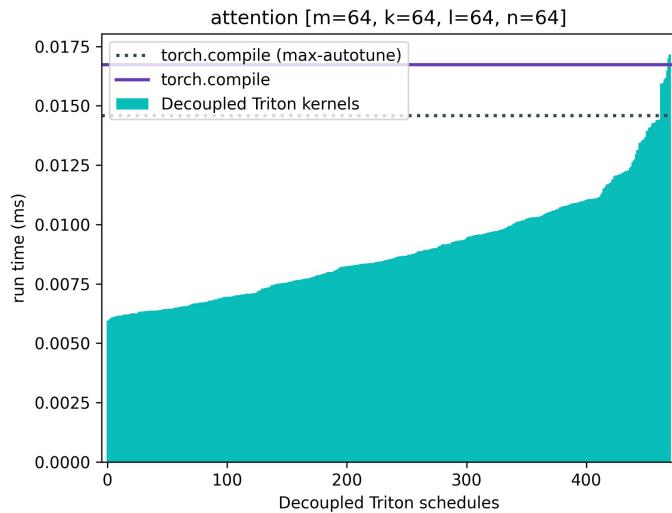
```
# Algorithm
mm[x, y] = rdot(A[x, k], B[k, y], k);
```

```
# Schedule
mm.tensorize(x:128, k:32, y:128);
mm.block(x:128, y:128);
mm.map(x:x1/8, y, x1);
mm.num_warps(4);
mm.num_stages(3);
mm.compile();
```



0	0	0	0
0	0	0	0

1	1	1	1
1	1	1	1



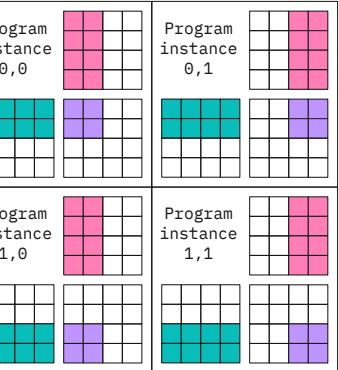
0	0	0	0
0	0	0	0

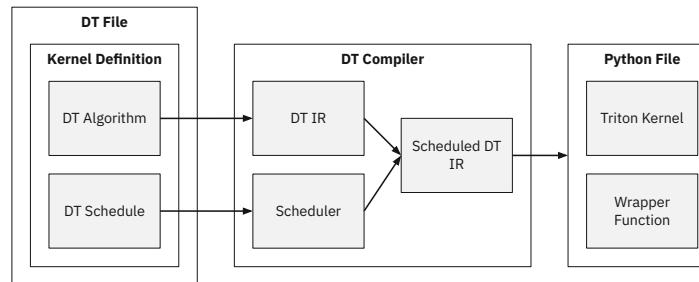
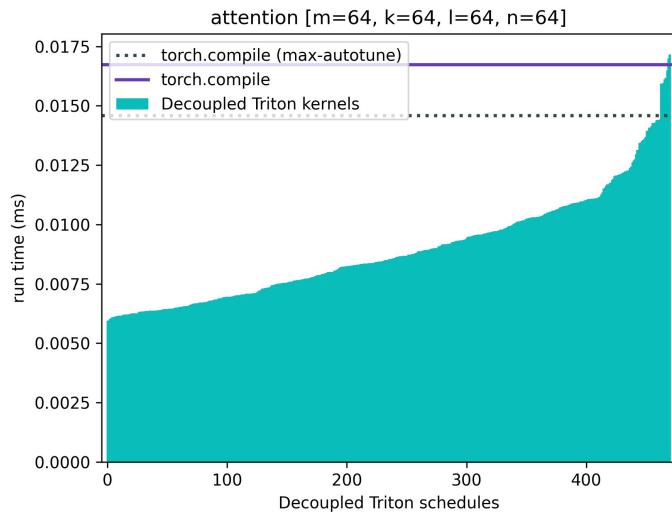
1	1	1	1
1	1	1	1

```
# Declarations
Func mm;
In A, B;
Var x, y;
RVar k;
```

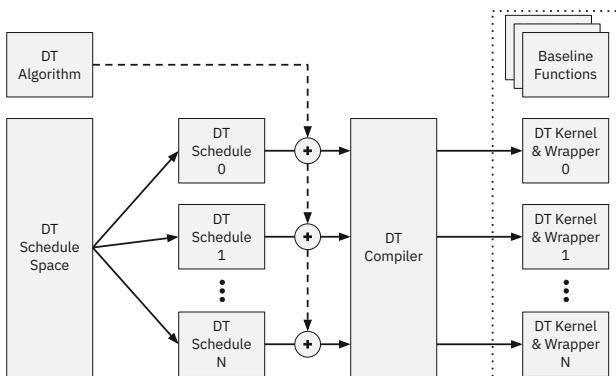
```
# Algorithm
mm[x, y] = rdot(A[x, k], B[k, y], k);
```

```
# Schedule
mm.tensorize(x:128, k:32, y:128);
mm.block(x:128, y:128);
mm.map(x:x1/8, y, x1);
mm.num_warps(4);
mm.num_stages(3);
mm.compile();
```





# Thank you :)



0	0	0	0
0	0	0	0

1	1	1	1
1	1	1	1

```

# Declarations
Func mm;
In A, B;
Var x, y;
RVar k;

# Algorithm
mm[x, y] = rdot(A[x, k], B[k, y], k);

# Schedule
mm.tensorize(x:128, k:32, y:128);
mm.block(x:128, y:128);
mm.map(x:x1/8, y, x1);
mm.num_warps(4);
mm.num_stages(3);
mm.compile();

```