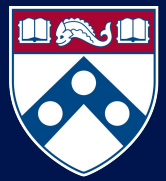
An Analysis of Triton VS Cuda

Ian Lamont, Paul Loh, Tarunyaa Sivakumar, Kidus Seyoum



Penn Engineering
University of Pennsylvania

Introduction

Project Overview

- Comparing Triton and CUDA
- Implementing Conv2D

Key Questions:

- How does Triton achieve CUDA performance
- What are differences in implementation and optimization

Methodology

Compare performance of similar algorithms in Triton and CUDA, Convolutional Neural Network (ConvNN) in particular.

- Implement ConvNN in both CUDA and Triton.
- Benchmark performance.
- Analyze generated PTX to understand optimization differences.

July 28, 2021

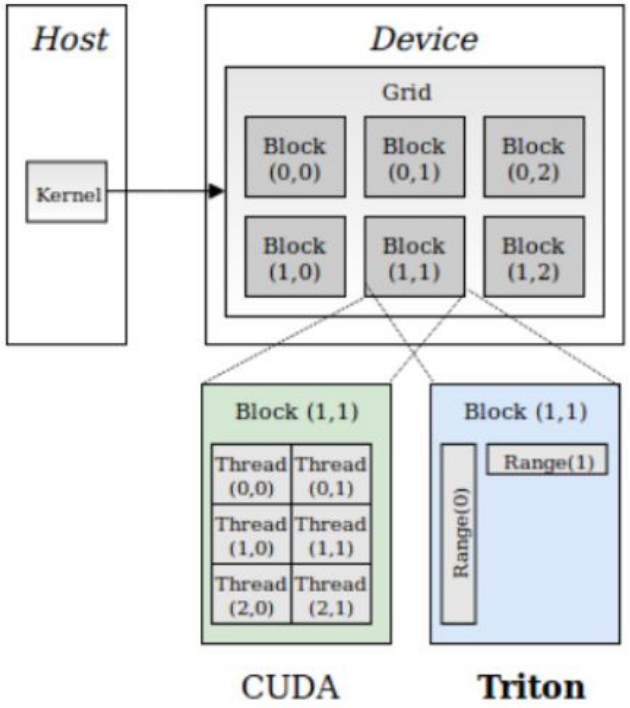
Introducing Triton: Open-source GPU programming for neural networks

View code ↗ Read documentation ↗

Triton Motivation

- Cuda is hard to write
  - Memory management
  - Thread level parallelism
- Triton does this for you

- Single-threaded "tile" level programming



Triton

- Language system built around Triton-IR
- Tile-Level analysis
- Auto-Tuning

- Python -> Triton-IR -> PTX

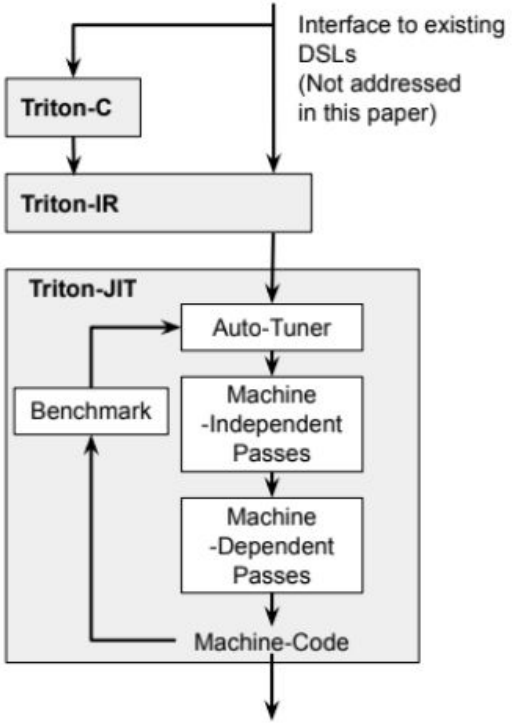


Figure 2. Overview of Triton

Triton Jit

The goal of Triton-JIT is to simplify and compile Triton-IR programs

- hierarchical tiling
- memory coalescing,
- Shared memory allocation
- shared memory synchronization

	CUDA	Triton
Memory Coalescing	Manual	Automatic
Shared Memory Management	Manual	Automatic
Scheduling (Withing SMs)	Manual	Automatic
Scheduling (Across SMs)	Manual	Manual

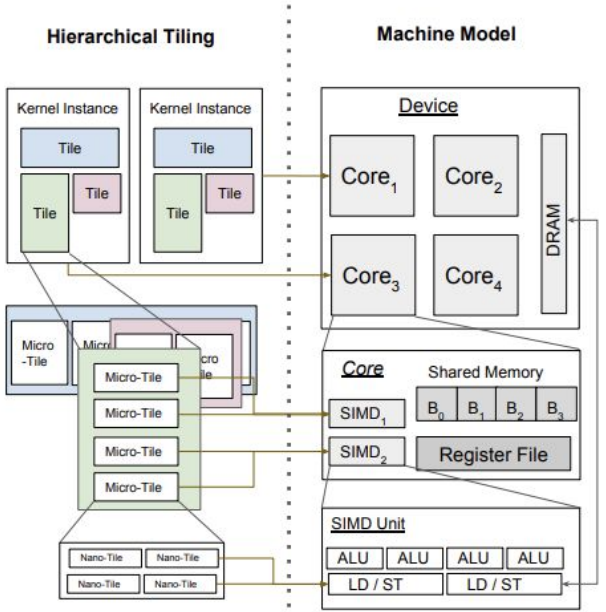
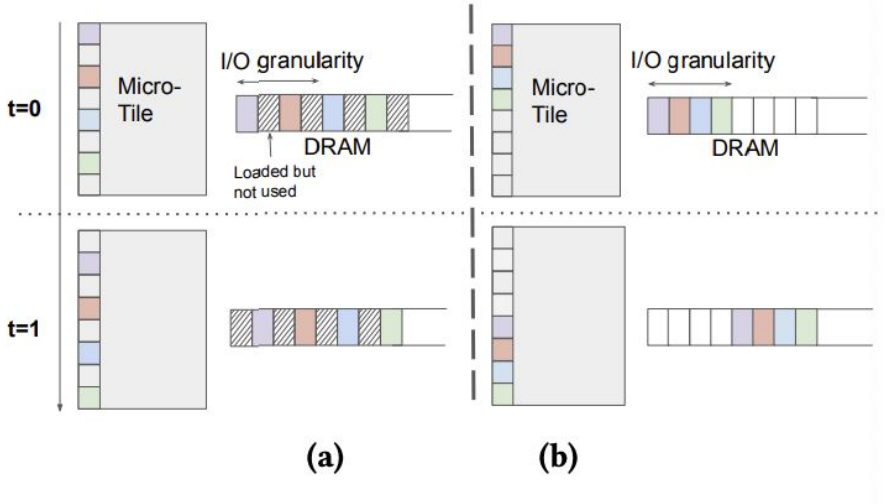


Figure 5. Hierarchical Tiling in the Triton-IR Machine Model

Penn Engineering

Memory Coalescing

- Triton-IR programs are single-threaded and auto parallelized
- Compiler Back-end orders threads internally within micro-tile



Shared Memory optimizations

- For tile level ops with high AI. eg. dot.
- When/where should a tile be stashed in this space?
- Linear-time storage allocation
- Mem synch by inserting barriers, preventing RAW/WAR hazards

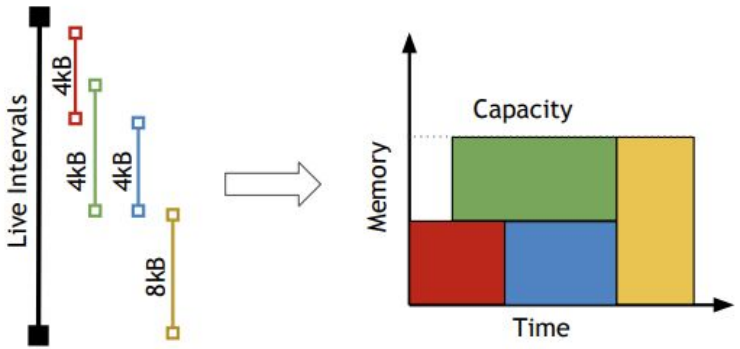


Figure 7. Shared Memory Allocation

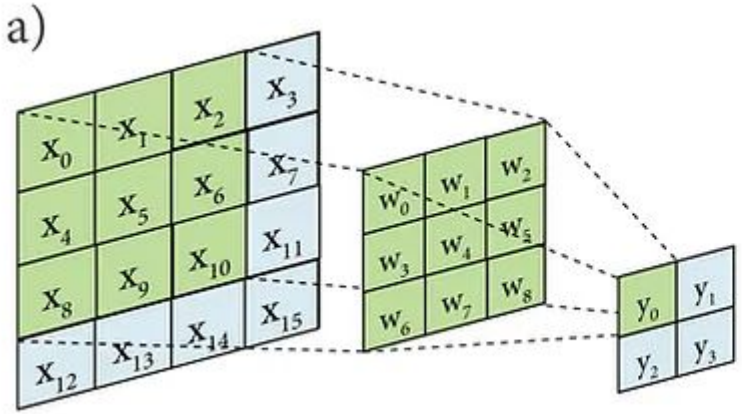
Pythonic / Torch-like vector operations

- Higher-level language compared to CUDA
- Low-level semantics such as memory management, coalescing, synchronization hidden from user

```
# If it is out of bounds, set it to 0.
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)
# We accumulate along the K dimension.
accumulator = tl.dot(a, b, accumulator)
```

2D Convolution Explained

- Kernel Size: The dimensions of the kernel (e.g., 3×3, 5×5).
- Stride: The step size for moving the kernel.
- Padding: Adding extra rows/columns to the input to control output size.
- Output Dimensions: Depend on the input size, kernel size, stride, and padding.



CUDA benchmark

```
// Each thread responsible for calculating element of output
const unsigned output_x = blockIdx.x * blockDim.x + threadIdx.x;
const unsigned output_y = blockIdx.y * blockDim.y + threadIdx.y;

// Ensure thread is within output bounds
if (output_x < outputWidth && output_y < outputHeight) {
    float temp = 0.0f;

    // Calculate starting position (to add padding)
    const unsigned input_x_start = output_x * stride;
    const unsigned input_y_start = output_y * stride;

    // Perform convolution (sliding window)
    for (int i = 0; i < kernelHeight; i++) {
        for (int j = 0; j < kernelWidth; j++) {
            int input_x = input_x_start + j;
            int input_y = input_y_start + i;
            temp += input[input_y * inputWidth + input_x] * kernel[i * kernelWidth + j];
        }
    }

    output[output_y * outputWidth + output_x] = temp;
```

cuDNN benchmark

```
// cuDNN describes data with a generic n-D tensor descriptor defined with the following parameters
// - a number of dimensions from 3 to 8
// - a data type
// - an integer array defining the size of each dimension
// - an integer array defining the stride of each dimension

// Create tensor descriptors
cudnnTensorDescriptor_t inputDesc, outputDesc;
cudnnFilterDescriptor_t kernelDesc;
cudnnConvolutionDescriptor_t convDesc;

// Input tensor descriptor
cudnnCreateTensorDescriptor(&inputDesc);
cudnnSetTensor4dDescriptor(inputDesc, CUDNN_TENSOR_NCHW, CUDNN_DATA_FLOAT, 1, 1, inputHeight, inputWidth);
// NCHW stands for Batch-Channel-Height-Width layout - how its laid out in memory

// Kernel descriptor
cudnnCreateFilterDescriptor(&kernelDesc);
cudnnSetFilter4dDescriptor(kernelDesc, CUDNN_DATA_FLOAT, CUDNN_TENSOR_NCHW, 1, 1, kernelHeight, kernelWidth); // K=1, C=1

// Output tensor descriptor
cudnnCreateTensorDescriptor(&outputDesc);
cudnnSetTensor4dDescriptor(outputDesc, CUDNN_TENSOR_NCHW, CUDNN_DATA_FLOAT, 1, 1, outputHeight, outputWidth);

// Convolution descriptor
cudnnCreateConvolutionDescriptor(&convDesc);
cudnnSetConvolution2dDescriptor(convDesc,
                                0, 0,         // Padding: height, width
                                stride, stride, // Stride: height, width
                                1, 1,         // Dilation: height, width
                                CUDNN_CONVOLUTION, CUDNN_DATA_FLOAT);

// Chooses the fastest available algorithm based on hardware capabilities
int returnedAlgoCount;
cudnnConvolutionFwdAlgoPerf_t algoPerf;
cudnnGetConvolutionForwardAlgorithm_v7(cudnn, inputDesc, kernelDesc, convDesc, outputDesc,
    1, &returnedAlgoCount, &algoPerf);
cudnnConvolutionFwdAlgo_t algo = algoPerf.algo;

// Allocating a workspace - this is a temporary memory buffer on the GPU used as a scratch memory for storing intermediate results
size_t workspaceSize = 0;
cudnnGetConvolutionForwardWorkspaceSize(cudnn, inputDesc, kernelDesc, convDesc, outputDesc, algo, &workspaceSize);
void* workspace = nullptr;
if (workspaceSize > 0) {
    cudaMalloc(&workspace, workspaceSize);
}
```

- Descriptors: Define the shape, layout, and data type of tensors (input, output, kernel) and convolution parameters (e.g., stride, padding).
- Workspace: Temporary GPU memory used for intermediate results, enabling faster execution of advanced convolution algorithms.
- Convolution Algorithm: cuDNN selects the fastest algorithm (e.g., GEMM, FFT, Winograd) based on the descriptors and hardware.
- Memory Management: GPU memory is allocated for tensors and workspace, with data copied between CPU and GPU as needed.

Triton (Python) benchmark

```python
# Calculate the output coordinates for this block
bh = tl.program_id(0) * BLOCK_H
bw = tl.program_id(1) * BLOCK_W

# Create a range for the block
h_idxs = bh + tl.arange(0, BLOCK_H)
w_idxs = bw + tl.arange(0, BLOCK_W)

# Mask to ensure we don't go out of bounds for the output
h_output_mask = h_idxs < (H - KH + 1)
w_output_mask = w_idxs < (W - KW + 1)

# Initialize the output block
out = tl.zeros((BLOCK_H, BLOCK_W), dtype=tl.float32)
```

Single-Threaded "**tiles**"
Concise ~20 LOC

```python
# Iterate over each element in the kernel
for khw in range(KH * KW, num_stages = 3):
    kh = khw // KW
    kw = khw % KW
    # Calculate the pointers for the input
    h_in = h_idxs + kh
    w_in = w_idxs + kw

    # Mask to ensure valid input indices
    valid_h = h_in < H
    valid_w = w_in < W
    # we want to make sure that we are producing a neccessary output,
    # and that we are not out of bounds for the input
    # mask is of shape (BLOCK_H, BLOCK_W)
    # we want to broadcast everything to be of shape
    # (1, BLOCK_W) or (BLOCK_H, 1)
    mask = h_output_mask[:, None] & w_output_mask[None, :] & \
           valid_h[:, None] & valid_w[None, :]

    # Load input and kernel values
    input_val = tl.load(input_ptr + h_in[:, None] * W + w_in[None, :], \
                        mask=mask)
    kernel_val = tl.load(kernel_ptr + kh * KW + kw)

    # Accumulate convolution result (here its a tensor * a scalar)
    out += input_val * kernel_val

# Write the output block
tl.store(output_ptr + h_idxs[:, None] * (W - KW + 1) + w_idxs[None, :], \
         out, mask=h_output_mask[:, None] & w_output_mask[None, :])
```

CUDA PTX

- Very simple translation to PTX
- Lots of jumps/branches
- Works on 1 value at a time

```
$L__BB0_5:
    ld.global.f32   %f14, [%rd23+-8];
    ld.global.f32   %f15, [%rd24+-8];
    fma.rn.f32  %f16, %f15, %f14, %f35;
    ld.global.f32   %f17, [%rd23+-4];
    ld.global.f32   %f18, [%rd24+-4];
    fma.rn.f32  %f19, %f18, %f17, %f16;
    ld.global.f32   %f20, [%rd23];
    ld.global.f32   %f21, [%rd24];
    fma.rn.f32  %f22, %f21, %f20, %f19;
    ld.global.f32   %f23, [%rd23+4];
    ld.global.f32   %f24, [%rd24+4];
    fma.rn.f32  %f35, %f24, %f23, %f22;
    add.s32     %r39, %r39, 4;
    add.s64     %rd24, %rd24, 16;
    add.s64     %rd23, %rd23, 16;
    add.s32     %r38, %r38, -4;
    setp.ne.s32     %p8, %r38, 0;
    @%p8 bra    $L__BB0_5;
```

```
.visible .entry _Z11convolutionPKfPfS0_iiiiiii(
    .param .u64 _Z11convolutionPKfPfS0_iiiiiii_param_0,
    .param .u64 _Z11convolutionPKfPfS0_iiiiiii_param_1,
    .param .u64 _Z11convolutionPKfPfS0_iiiiiii_param_2,
    .param .u32 _Z11convolutionPKfPfS0_iiiiiii_param_3,
    .param .u32 _Z11convolutionPKfPfS0_iiiiiii_param_4,
    .param .u32 _Z11convolutionPKfPfS0_iiiiiii_param_5,
    .param .u32 _Z11convolutionPKfPfS0_iiiiiii_param_6,
    .param .u32 _Z11convolutionPKfPfS0_iiiiiii_param_7,
    .param .u32 _Z11convolutionPKfPfS0_iiiiiii_param_8,
    .param .u32 _Z11convolutionPKfPfS0_iiiiiii_param_9
)
{
    .reg .pred  %p<13>;
    .reg .f32   %f<37>;
    .reg .b32   %r<40>;
    .reg .b64   %rd<25>;


    ld.param.u64    %rd14, [_Z11convolutionPKfPfS0_iiiiiii_param_0];
    ld.param.u64    %rd13, [_Z11convolutionPKfPfS0_iiiiiii_param_1];
    ld.param.u64    %rd15, [_Z11convolutionPKfPfS0_iiiiiii_param_2];
    ld.param.u32    %r17, [_Z11convolutionPKfPfS0_iiiiiii_param_3];
    ld.param.u32    %r18, [_Z11convolutionPKfPfS0_iiiiiii_param_5];
    ld.param.u32    %r19, [_Z11convolutionPKfPfS0_iiiiiii_param_6];
    ld.param.u32    %r20, [_Z11convolutionPKfPfS0_iiiiiii_param_7];
    ld.param.u32    %r21, [_Z11convolutionPKfPfS0_iiiiiii_param_8];
    ld.param.u32    %r22, [_Z11convolutionPKfPfS0_iiiiiii_param_9];
    cvta.to.global.u64  %rd1, %rd15;
    cvta.to.global.u64  %rd2, %rd14;
    mov.u32     %r23, %ntid.x;
    mov.u32     %r24, %ctaid.x;
    mov.u32     %r25, %tid.x;
    mad.lo.s32  %r1, %r24, %r23, %r25;
    mov.u32     %r26, %ntid.y;
    mov.u32     %r27, %ctaid.y;
    mov.u32     %r28, %tid.y;
    mad.lo.s32  %r2, %r27, %r26, %r28;
    setp.ge.u32     %p1, %r1, %r21;
    setp.ge.u32     %p2, %r2, %r22;
    or.pred     %p3, %p1, %p2;
    @%p3 bra    $L__BB0_12;
```

Triton PTX

- Heavy use of predication
- Operates on up to 8 values per thread
- 128 cooperative threads (16 warps) per cluster (tile/block)
  - High coalescing (8 fully saturated cache line requests)

```
$L__BB0_2:
    add.s64     %rd16, %rd1, %rd15;
    mul.wide.s32    %rd17, %r77, 4;
    add.s64     %rd18, %rd1, %rd17;
    mul.wide.s32    %rd19, %r78, 4;
    add.s64     %rd20, %rd1, %rd19;
    mul.wide.s32    %rd21, %r79, 4;
    add.s64     %rd22, %rd1, %rd21;
    mul.wide.s32    %rd23, %r80, 4;
    add.s64     %rd24, %rd1, %rd23;
    mul.wide.s32    %rd25, %r81, 4;
    add.s64     %rd26, %rd1, %rd25;
    mul.wide.s32    %rd27, %r82, 4;
    add.s64     %rd28, %rd1, %rd27;
    .loc    1 80 60
    mul.wide.s32    %rd29, %r74, 4;
    add.s64     %rd4, %rd14, %rd29;
    add.s64     %rd5, %rd16, %rd29;
    add.s64     %rd6, %rd18, %rd29;
    add.s64     %rd7, %rd20, %rd29;
    add.s64     %rd8, %rd22, %rd29;
    add.s64     %rd9, %rd24, %rd29;
    add.s64     %rd10, %rd26, %rd29;
    add.s64     %rd11, %rd28, %rd29;
    .loc    1 80 28
    mov.u32 %r54, 0x0;
    @%p19 ld.global.b32 { %r54 }, [ %rd4 + 0 ];
    mov.u32 %r55, 0x0;
    @%p20 ld.global.b32 { %r55 }, [ %rd5 + 0 ];
    mov.u32 %r56, 0x0;
    @%p21 ld.global.b32 { %r56 }, [ %rd6 + 0 ];
    mov.u32 %r57, 0x0;
    @%p22 ld.global.b32 { %r57 }, [ %rd7 + 0 ];
    mov.u32 %r58, 0x0;
    @%p23 ld.global.b32 { %r58 }, [ %rd8 + 0 ];
    mov.u32 %r59, 0x0;
    @%p24 ld.global.b32 { %r59 }, [ %rd9 + 0 ];
    mov.u32 %r60, 0x0;
    @%p25 ld.global.b32 { %r60 }, [ %rd10 + 0 ];
    mov.u32 %r61, 0x0;
    @%p26 ld.global.b32 { %r61 }, [ %rd11 + 0 ];
```

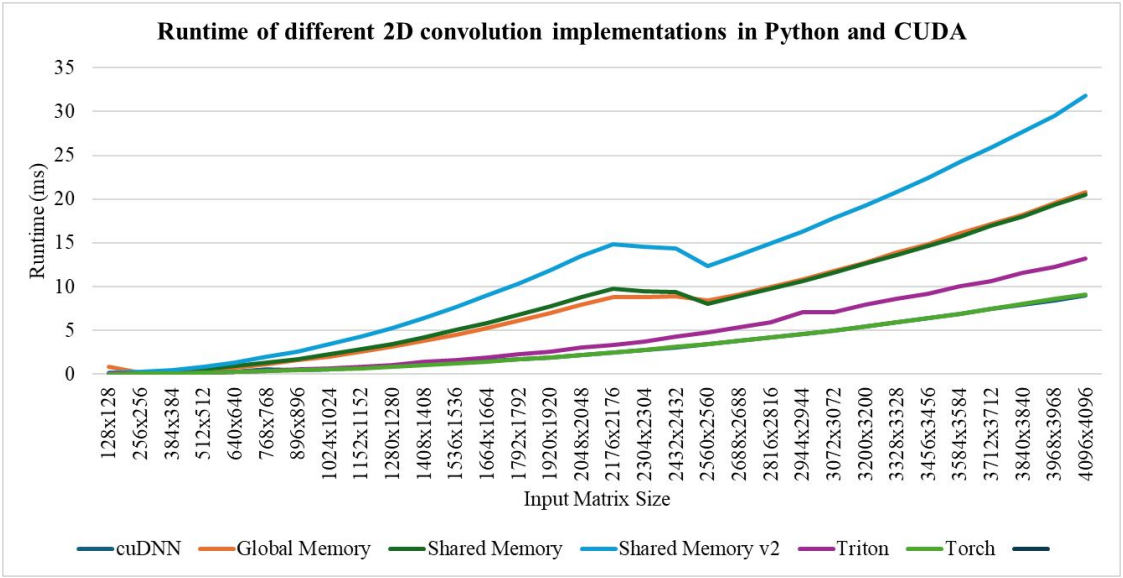
```
.visible .entry conv2d_kernel_no_stride(
    .param .u64 conv2d_kernel_no_stride_param_0,
    .param .u64 conv2d_kernel_no_stride_param_1,
    .param .u64 conv2d_kernel_no_stride_param_2,
    .param .u32 conv2d_kernel_no_stride_param_3,
    .param .u32 conv2d_kernel_no_stride_param_4,
    .param .u32 conv2d_kernel_no_stride_param_5,
    .param .u32 conv2d_kernel_no_stride_param_6
)
.maxntid 128, 1, 1
{
    .reg .pred  %p<54>;
    .reg .b32   %r<110>;
    .reg .f32   %f<42>;
    .reg .b64   %rd<58>;
    .loc    1 30 0
$L__func_begin0:
    .loc    1 30 0

    ld.param.u32    %r35, [conv2d_kernel_no_stride_param_6];
    ld.param.u32    %r34, [conv2d_kernel_no_stride_param_4];
    ld.param.u32    %r33, [conv2d_kernel_no_stride_param_3];
    ld.param.u64    %rd3, [conv2d_kernel_no_stride_param_2];
$L__tmp0:
    .loc    1 47 23
    mov.u32 %r36, %ctaid.x;
    .loc    1 47 28
    shl.b32     %r46, %r36, 5;
    .loc    1 48 23
    mov.u32 %r37, %ctaid.y;
    .loc    1 48 28
    shl.b32     %r1, %r37, 5;
    ld.param.u32    %r47, [conv2d_kernel_no_stride_param_5];
```

Penn Engineering

Experimental Setup

- GPU: All code was run locally on a laptop NVIDIA RTX 4060
- CPU: Intel Xeon (host processor for data preparation and non-GPU operations)
- Triton packages were downloaded using WSL
- Python 3.12.3
  - Dependencies: Triton, PyTorch
- C++ with CUDA extensions
  - GPU Libraries: CUDA toolkit, cuDNN
  - Compiler: nvcc, Clang
- Benchmarks: Triton @triton.testing.perf_report framework was used for performance benchmarking and comparison with PyTorch's native implementations.

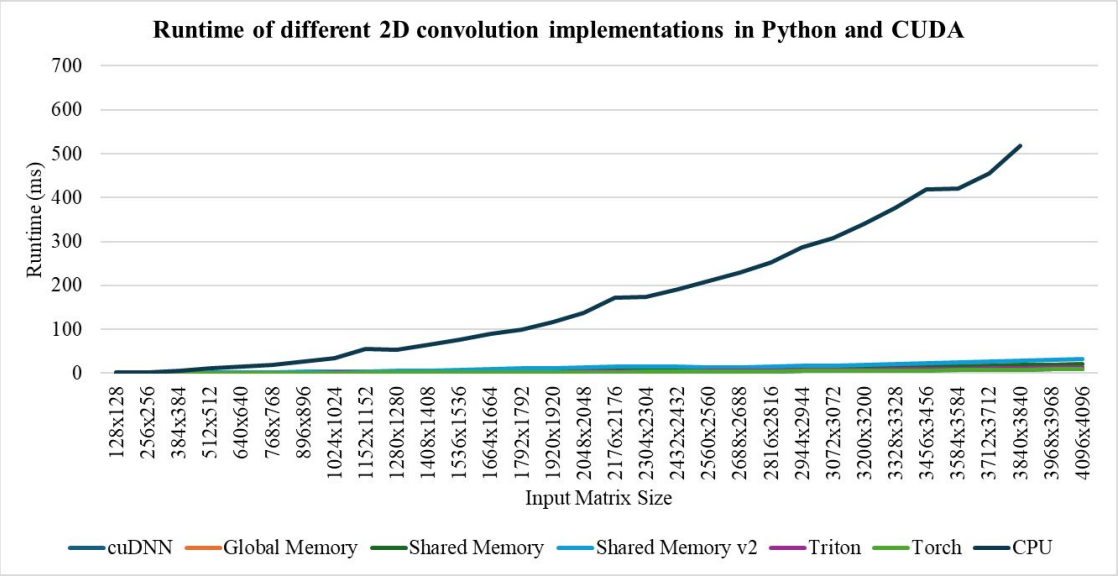
Runtime of Algorithms



The graph shows that cuDNN and torch is the fastest due to its highly optimized implementation. Shared Memory (Strategy 1) outperforms Global Memory and Shared Memory (Strategy 2), showcasing efficient memory usage. Triton achieves similar performance to Strategy . Overall, cuDNN dominates, while Shared Memory (1) and Triton offer competitive alternatives.

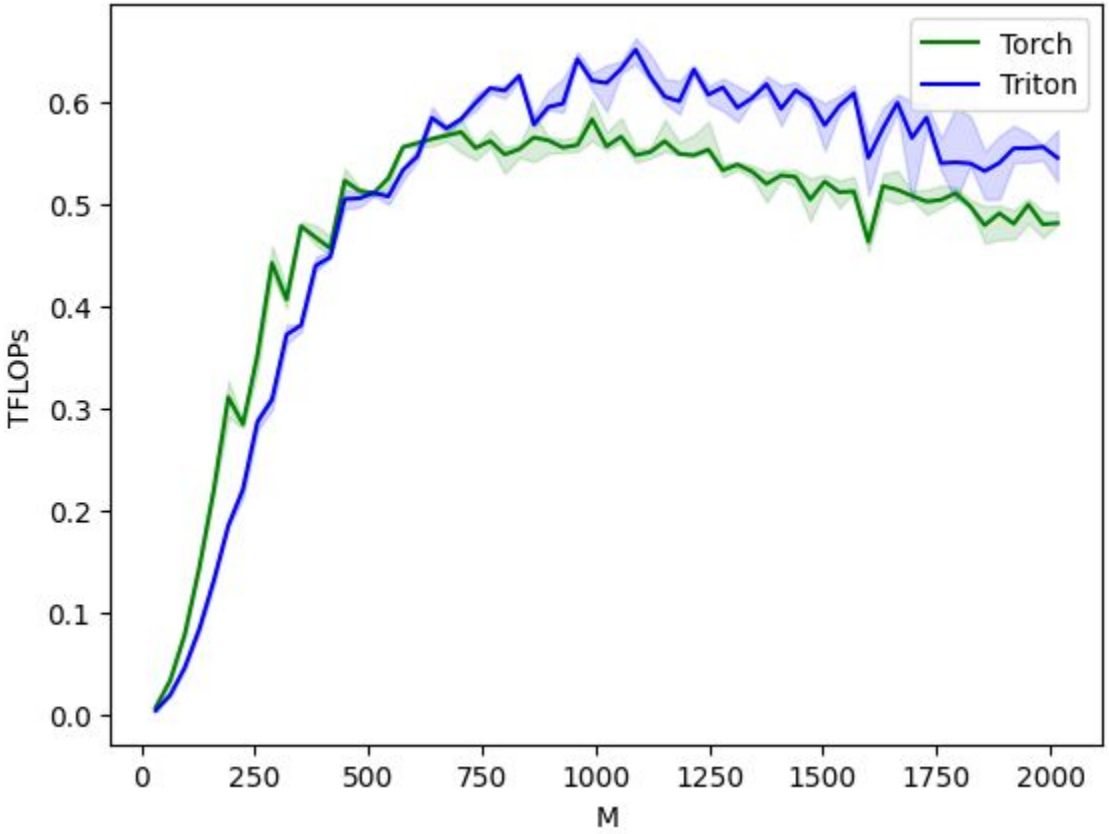
Note that this graph was edited such that the timing data was only for the kernel launch so different implementations are now directly comparable.

Runtime of Algorithms



Obviously, the CPU performs way worse.

(Our) Triton vs Torch



Optimising CUDA code

3 versions were implemented

1. **Global Memory (Coalesced Access):**

Direct implementation using only global memory, optimized for coalesced memory access.

2. **Shared Memory (Strategy 1):**

The shared memory tile matches the output tile size.
Multiple steps are used to load input tiles into shared memory.

3. **Shared Memory (Strategy 2):**

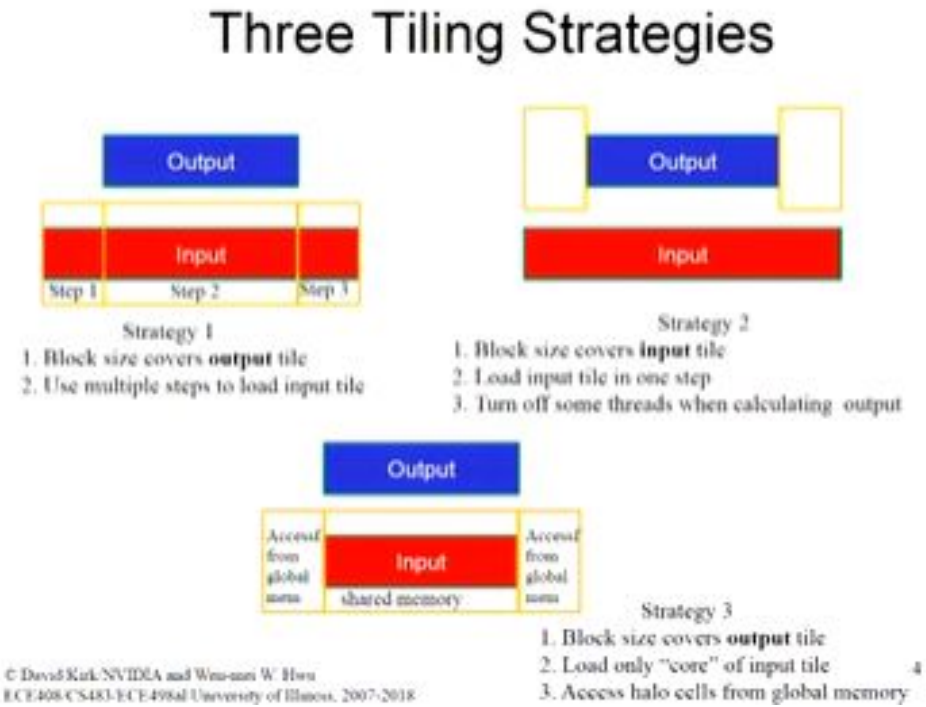
The shared memory tile size matches the input tile size, leading to some idle threads during computation.
Input tiles are loaded into shared memory in a single step.

4. **cuDNN Convolution:**

NVIDIA's optimized cuDNN library implementation for convolution.

CUDA Shared Memory

- Strategy 1: Focuses on output tiles; multiple steps load input tiles, optimizing output writes but increasing input memory accesses.
- Strategy 2: Focuses on input tiles; loads the entire input tile in one step, but some threads may remain idle during output computation.
- Strategy 3: Balances shared memory usage; only loads the core of input tiles while accessing halo (boundary) cells directly from global memory.



Penn Engineering

CUDA Shared Memory

Strategy 1: Block Size Matches Output Tile Size

- Tile Size and Memory Layout:
  - The shared memory tile is sized larger than the block output tile by the mask size for halo regions.
  - Threads load data into shared memory in two steps because the shared memory area exceeds the block size.
- Execution:
  - Each thread loads at most two elements from global memory into shared memory.
  - The convolution is computed after all threads synchronize (__syncthreads()).
- Advantages:
  - Allows a compact mapping of threads to output elements.
  - Each thread is responsible for a single output, ensuring a balanced workload.
- Disadvantages:
  - Requires two load steps from global memory, adding complexity and potential inefficiency.
  - More shared memory usage compared to simpler strategies.

```
__shared__ float inputTile[w_y][w_x];

// Each thread calculates one element of output tile

// dest is wrt output shared memory tile
// dest: 1D index within shared memory output tile for current thread
int dest = threadIdx.y * TILE_WIDTH + threadIdx.x;
// destY and destX are the row and column of shared memory - convert 1D index into 2D row and col coords
int destY = dest / w_x;
int destX = dest % w_x;

// src is wrt global memory
// srcY and srcX are indexes to fetch data from input image
// Need to offset since we are also using values from neighbouring input tiles
int srcY = blockIdx.y * TILE_WIDTH + destY - Mask_radius_x;
int srcX = blockIdx.x * TILE_WIDTH + destX - Mask_radius_y;
// Make into 1D index to access within input
int src = srcY * inputWidth + srcX;

if (srcY >= 0 && srcY < inputHeight && srcX >= 0 && srcX < inputWidth) {
    inputTile[destY][destX] = input[src];
} else {
    inputTile[destY][destX] = 0.0f;
}

// When shared memory tile is larger than number of threads in the block, threads iterate to load
// all necessary data
dest = threadIdx.y * TILE_WIDTH + threadIdx.x + (TILE_WIDTH * TILE_WIDTH);
destY = dest / w_x;
destX = dest % w_x;
srcY = blockIdx.y * TILE_WIDTH + destY - Mask_radius_x;      You, 46 minutes ago • multiple shared mem im
srcX = blockIdx.x * TILE_WIDTH + destX - Mask_radius_y;
src = srcY * inputWidth + srcX;
if (destY < w_y && destX < w_x)
{
    if (srcY >= 0 && srcY < inputHeight && srcX >= 0 && srcX < inputWidth) {
        inputTile[destY][destX] = input[src];
    } else {
        inputTile[destY][destX] = 0.0;
    }
}
__syncthreads();

float temp = 0.0;
for (int y = 0; y < Mask_width; y++) {
    for (int x = 0; x < Mask_width; x++) {
        temp += inputTile[threadIdx.y + y][threadIdx.x + x] * kernel[y * Mask_width + x];
    }
}
int outputY = blockIdx.y * TILE_WIDTH + threadIdx.y;
int outputX = blockIdx.x * TILE_WIDTH + threadIdx.x;
if (outputY < inputHeight && outputX < inputWidth) {
    output[(outputY * inputWidth + outputX)] = temp;
}
```

Penn Engineering

CUDA Shared Memory v2

Strategy 2: Block Size Matches Input Tile Size

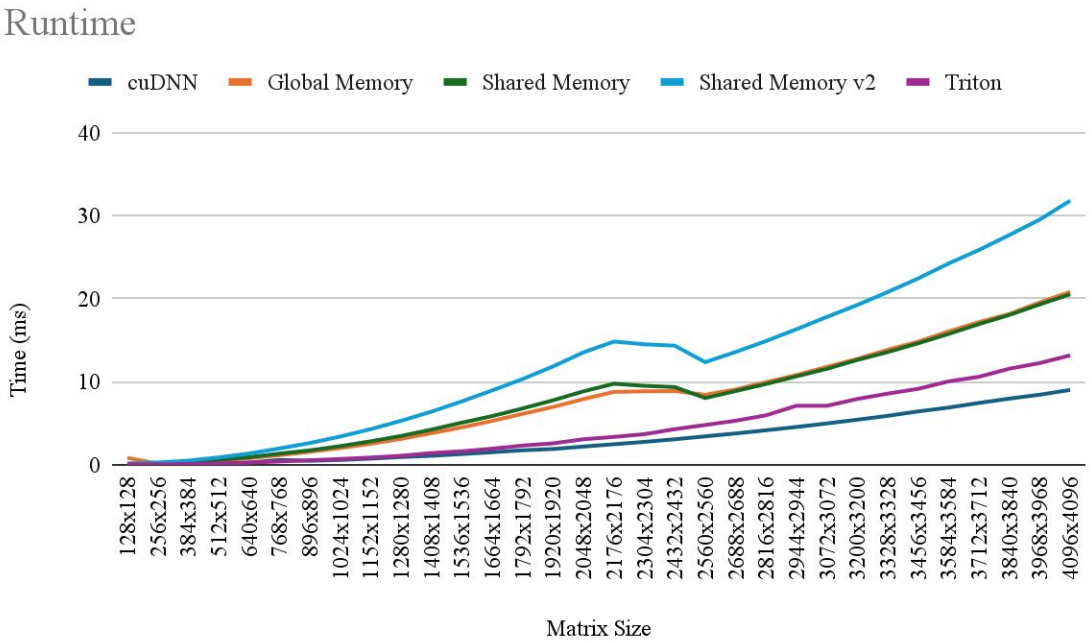
- Tile Size and Memory Layout:
  - The block size matches the entire shared memory input tile, including halo regions.
  - Threads load data from global memory to shared memory in a single step.
- Execution:
  - Threads calculate outputs within the central region of the shared memory tile (TILE_WIDTH × TILE_WIDTH).
  - Threads at the borders are "turned off" during output calculation to avoid invalid writes.
- Advantages:
  - Simpler memory loading: shared memory is populated in one step.
  - Avoids redundant data movement compared to Strategy 1.
- Disadvantages:
  - Some threads remain idle during computation, leading to underutilization.
  - Thread-block size is larger, potentially leading to fewer concurrent blocks on the GPU.

```
__global__ void convolution_shared_v2(const float *input, float *output, const float *kernel,
    int inputWidth, int inputHeight,
    int kernelWidth, int kernelHeight) {

        int outputX = blockIdx.x * TILE_WIDTH + threadIdx.x;
        int outputY = blockIdx.y * TILE_WIDTH + threadIdx.y;
        int srcY = outputY - Mask_radius_y;
        int srcX = outputX - Mask_radius_x;
        int src = srcY * inputWidth + srcX;

        __shared__ float inputTile[w_y][w_x];

        if (srcY >= 0 && srcY < inputHeight && srcX >= 0 && srcX < inputWidth) {
            inputTile[threadIdx.y][threadIdx.x] = input[src];
        } else {
            inputTile[threadIdx.y][threadIdx.x] = 0.0f;
        }
        __syncthreads();

        // Some threads won't write any outputs, only need to calculate w_x, w_y elements
        float temp = 0.0f;
        if (threadIdx.y < TILE_WIDTH && threadIdx.x < TILE_WIDTH) {
            for (int i = 0; i < Mask_width; i++) {
                for (int j = 0; j < Mask_width; j++) {
                    temp += inputTile[i + threadIdx.y][j + threadIdx.x] * kernel[i * Mask_width + j];
                }
            }
            output[outputY * inputWidth + outputX] = temp;
        }


    }
```

Comparing CUDA Implementations

- GPU - Shared Memory refers to Strategy 1
- GPU - Shared Memory v2 refers to Strategy 2

As expected, cuDNN has the lowest runtime due to its highly optimized implementation tailored for Nvidia GPUs. The shared memory Strategy 1 implementation performs well, as it effectively utilizes shared memory to minimize redundant global memory access while maintaining high thread utilization. In contrast, Strategy 2 performs worse than the version with just global memory coalescing, likely because many threads remain idle during output computation, leading to underutilization of resources. Additionally, the larger thread block size in Strategy 2 may reduce the number of concurrent blocks, further impacting performance.



Penn Engineering

Notes on Shared Memory Implementation

- The **input matrix requires padding** to account for the halo regions defined by the kernel's size. Note that padding wasn't used in the results of the previous section.
- Each thread block is designed to calculate an output tile of size TILE_WIDTH × TILE_WIDTH.
- To compute these outputs, the shared memory tile must include the halo regions, making its size (TILE_WIDTH + Mask_width - 1) × (TILE_WIDTH + Mask_width - 1).
- Why two loadings are needed in Strategy 1:
  - Since the shared memory area is larger than TILE_WIDTHxTILE_WIDTH and assuming that it is smaller than 2xTILE_WIDTHxTILE_WIDTH,
  - Then each thread should move at most two elements from global memory to shared memory
- For some reason, TILE_SIZE = 16 was the one that worked at the end.

Most difficult parts of the project

- Understanding how Triton works
  - Reading the original Triton paper
  - Programming in the new "tile" paradigm (extremely hard to shift perspectives, whole new way of thinking)
- Understanding optimizations in PTX
- Figuring out the shared memory implementation of the 2D convolution
  - Took forever to understand that it wasn't working because the implementation needed padding
  - Had to play around with the TILE_SIZE and input matrix sizes
- Setting up the virtual environment (WSL) to run Triton

Github Repositories

- Cuda code
- Triton code

References

- Original Triton paper
- Triton docs
  - Triton examples
- OpenAI
- Nvidia PTX docs