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

Authors: Philippe Tillet, H. T. Kung, and David Cox

Presented by: Vaibhav Gurunathan

Programming GPUs is Hard

- CuBLAS / cuDNN – fast but limited to a fixed set of operations
- Domain-Specific Languages – flexible but often slow
- Hand-written micro kernels – high performance but labor-intensive and non-portable
- Other high-level languages – simplify programming but lack full support for tile-level operations and optimizations

Triton's Vision & Approach

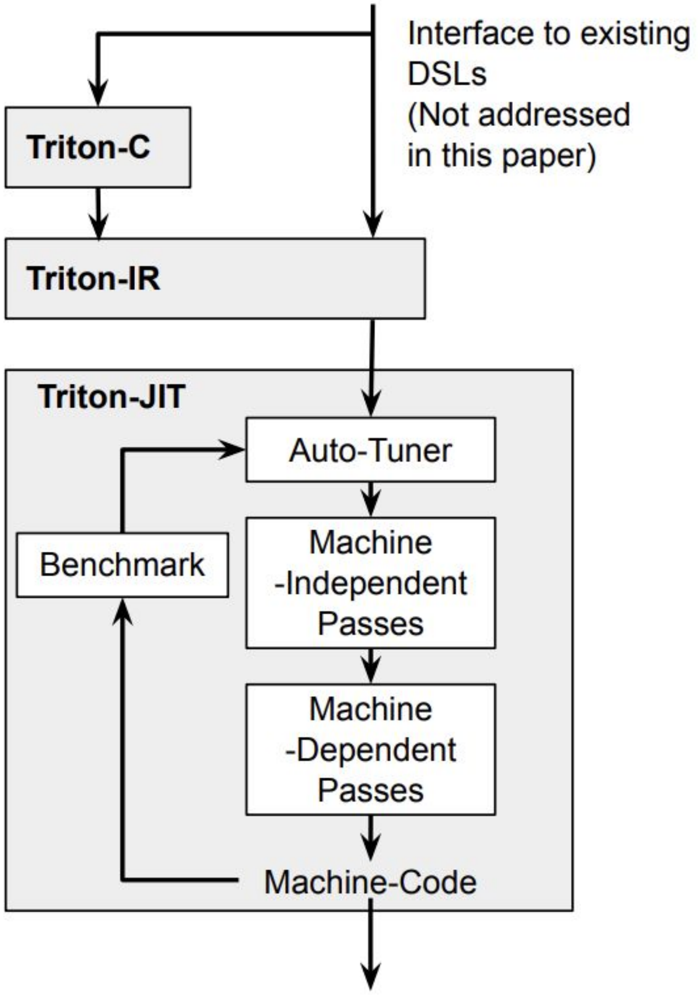
- Tile-based programming - Operate on tiles not individual elements
- Efficient - Compiler-managed optimizations
- Simple - C-like syntax, NumPy-style semantics

Why Tiles?

- Simplifies programming
- Intuitive abstraction
- Performance gains
  - Memory coalescing, cache management, and specialized hardware utilization
- Easy parameter tuning

Main Contributions

1. Triton-C
2. Triton-IR
   a. Triton Intermediate Representation
3. Triton-JIT
   a. Just In Time Compiler



Triton-C

Purpose: Stable Interface for Kernels

- CUDA like syntax
  - C-style loops, functions
- Numpy like semantics
  - Broadcasting, reshaping, and array math
- Single Program Multiple Data
  - Kernels written as a single-threaded program

$C = A \times B^T$

```
// Tile shapes are parametric and can be optimized
// by compilation backends
const tunable int TM = {16, 32, 64, 128}
const tunable int TN = {16, 32, 64, 128}
const tunable int TK = {8, 16}
// C = A * B.T
kernel void matmul_nt(float* a, float* b,float* c,
                      int M, in N, int K)
{
 // 1D tile of indices
 int rm[TM] = get_global_range(0);
 int rn[TN] = get_global_range(1);
 int rk[TK] = 0 ... TK;
 // 2D tile of accumulators
 float C[TM, TN] = 0;
 // 2D tile of pointers
 float* pa[TM, TK] = a + rm[:,newaxis] + rk * M;
 float* pb[TN, TK] = b + rn[:,newaxis] + rk * K;
 for(int k = K; k >= 0; k -= TK){
   bool check_k[TK] = rk < k;
   bool check_a[TM, TK] = (rm < M)[:,newaxis] && check_k;
   bool check_b[TN, TK] = (rn < N)[:,newaxis] && check_k;
   // load tile operands
   float A[TM, TK] = check_a ? *pa : 0;
   float B[TN, TK] = check_b ? *pb : 0;
   // accumulate
   C += dot(A, trans(B));
   // update pointers
   pa = pa + TK*M;
   pb = pb + TK*N;
 }
 // write-back accumulators
 float* pc[TM, TN] = c + rm[:,newaxis] + rn * M;
 bool check_c[TM,TN] = (rm < M)[:, newaxis] && (rn < N);
 @check_c *pc = C;
}
```

```
// Tile shapes are parametric and can be optimized
// by compilation backends
const tunable int TM = {16, 32, 64, 128}
const tunable int TN = {16, 32, 64, 128}
const tunable int TK = {8, 16}
// C = A * B.T
kernel void matmul_nt(float* a, float* b,float* c,
                      int M, in N, int K)
{
 // 1D tile of indices
 int rm[TM] = get_global_range(0);
 int rn[TN] = get_global_range(1);
 int rk[TK] = 0 ... TK;
 // 2D tile of accumulators
 float C[TM, TN] = 0;
 // 2D tile of pointers
 float* pa[TM, TK] = a + rm[:,newaxis] + rk * M;
 float* pb[TN, TK] = b + rn[:,newaxis] + rk * K;
 for(int k = K; k >= 0; k -= TK){
   bool check_k[TK] = rk < k;
   bool check_a[TM, TK] = (rm < M)[:,newaxis] && check_k;
   bool check_b[TN, TK] = (rn < N)[:,newaxis] && check_k;
   // load tile operands
   float A[TM, TK] = check_a ? *pa : 0;
   float B[TN, TK] = check_b ? *pb : 0;
   // accumulate
   C += dot(A, trans(B));
   // update pointers
   pa = pa + TK*M;
   pb = pb + TK*N;
 }
 // write-back accumulators
 float* pc[TM, TN] = c + rm[:,newaxis] + rn * M;
 bool check_c[TM,TN] = (rm < M)[:, newaxis] && (rn < N);
 @check_c *pc = C;
}
```

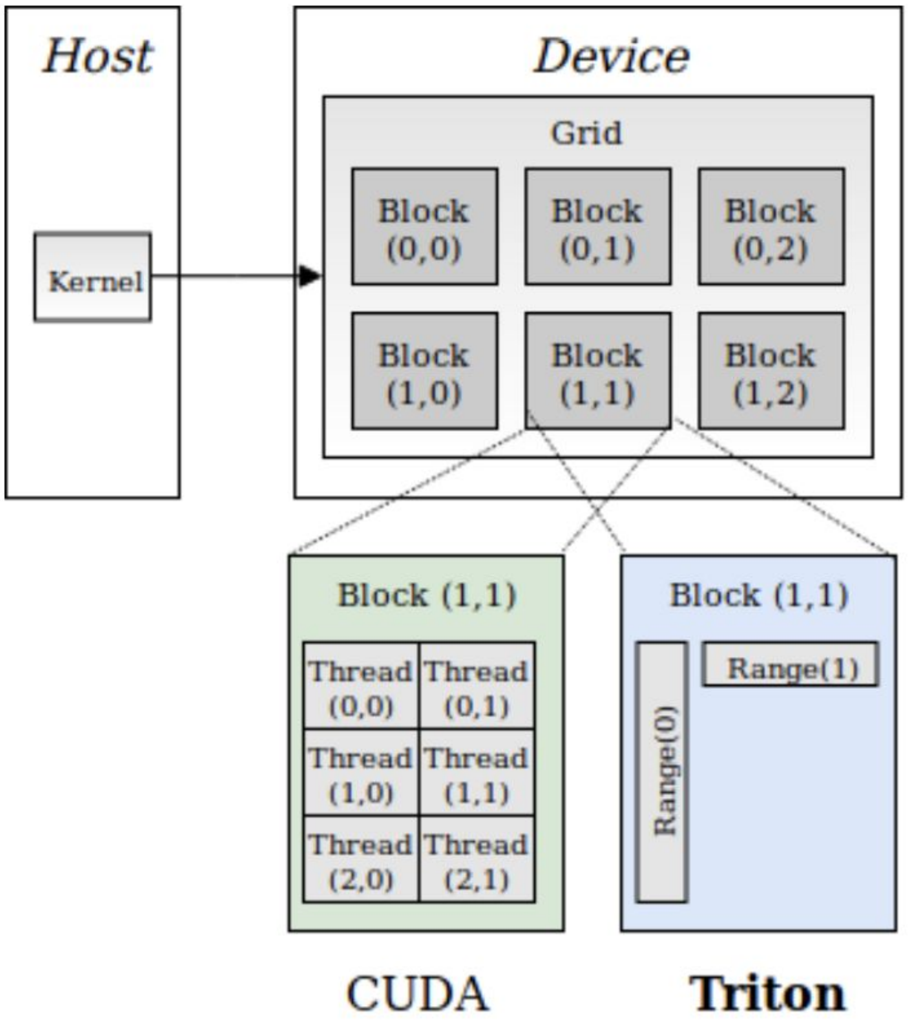
Tile-based syntax

$C = A \times B^T$

```
// Tile sizes (tunable constants)
constexpr int TM[] = {16, 32, 64, 128};
constexpr int TN[] = {16, 32, 64, 128};
constexpr int TK[] = {8, 16};

// Kernel: C = A * B^T
__global__ void matmul_nt(const float* __restrict__ A,
                          const float* __restrict__ B,
                          float* __restrict__ C,
                          int M, int N, int K)
{
    // 1D thread indices for the tile
    int tidm = threadIdx.x + blockIdx.x * blockDim.x;
    int tidn = threadIdx.y + blockIdx.y * blockDim.y;

    // Accumulator for the tile
    float c_val = 0.0f;

    // Loop over K dimension in tiles
    for (int k0 = 0; k0 < K; k0 += TK[0]) {
        // Tile width may exceed remaining K
        int tile_k = min(TK[0], K - k0);

        // Load a tile of A and B
        for (int k = 0; k < tile_k; ++k) {
            float a_val = (tidm < M) ? A[tidm * K + (k0 + k)] : 0.0f;
            float b_val = (tidn < N) ? B[tidn * K + (k0 + k)] : 0.0f;
            c_val += a_val * b_val; // accumulate dot product
        }
    }

    // Write back to C with bounds check
    if (tidm < M && tidn < N) {
        C[tidm * N + tidn] = c_val;
    }
}
```

```
// Tile shapes are parametric and can be optimized
// by compilation backends
const tunable int TM = {16, 32, 64, 128}
const tunable int TN = {16, 32, 64, 128}
const tunable int TK = {8, 16}
// C = A * B.T
kernel void matmul_nt(float* a, float* b, float* c,
                      int M, in N, int K)
{
 // 1D tile of indices
 int rm[TM] = get_global_range(0);
 int rn[TN] = get_global_range(1);
 int rk[TK] = 0 ... TK;
 // 2D tile of accumulators
 float C[TM, TN] = 0;
 // 2D tile of pointers
 float* pa[TM, TK] = a + rm[:,newaxis] + rk * M;
 float* pb[TN, TK] = b + rn[:,newaxis] + rk * K;
 for(int k = K; k >= 0; k -= TK){
   bool check_k[TK] = rk < k;
   bool check_a[TM, TK] = (rm < M)[:,newaxis] && check_k;
   bool check_b[TN, TK] = (rn < N)[:,newaxis] && check_k;
   // load tile operands
   float A[TM, TK] = check_a ? *pa : 0;
   float B[TN, TK] = check_b ? *pb : 0;
   // accumulate
   C += dot(A, trans(B));
   // update pointers
   pa = pa + TK*M;
   pb = pb + TK*N;
 }
 // write-back accumulators
 float* pc[TM, TN] = c + rm[:,newaxis] + rn * M;
 bool check_c[TM,TN] = (rm < M)[:, newaxis] && (rn < N);
 @check_c *pc = C;
}
```

```cuda
// Kernel: C = A * B^T without tiling
__global__ void matmul_nt_nontiled(const float* __restrict__ A,
                                   const float* __restrict__ B,
                                   float* __restrict__ C,
                                   int M, int N, int K)
{
    // Compute row and column indices for this thread
    int row = blockIdx.x * blockDim.x + threadIdx.x;
    int col = blockIdx.y * blockDim.y + threadIdx.y;

    if (row < M && col < N) {
        float sum = 0.0f;
        for (int k = 0; k < K; ++k) {
            float a_val = A[row * K + k];
            float b_val = B[col * K + k]; // B^T access
            sum += a_val * b_val;
        }
        C[row * N + col] = sum;
    }
}
```


Host
Kernel
Device
Grid
Block (0,0)
Block (0,1)
Block (0,2)
Block (1,0)
Block (1,1)
Block (1,2)
Block (1,1)
Thread (0,0)
Thread (0,1)
Thread (1,0)
Thread (1,1)
Thread (2,0)
Thread (2,1)
Block (1,1)
Range(0)
Range(1)
CUDA
Triton

```cpp
// Tile sizes (tunable constants)
constexpr int TM[] = {16, 32, 64, 128};
constexpr int TN[] = {16, 32, 64, 128};
constexpr int TK[] = {8, 16};

// Kernel: C = A * B^T
__global__ void matmul_nt(const float* __restrict__ A,
                          const float* __restrict__ B,
                          float* __restrict__ C,
                          int M, int N, int K)
{
    // 1D thread indices for the tile
    int tidm = threadIdx.x + blockIdx.x * blockDim.x;
    int tidn = threadIdx.y + blockIdx.y * blockDim.y;

    // Accumulator for the tile
    float c_val = 0.0f;

    // Loop over K dimension in tiles
    for (int k0 = 0; k0 < K; k0 += TK[0]) {
        // Tile width may exceed remaining K
        int tile_k = min(TK[0], K - k0);

        // Load a tile of A and B
        for (int k = 0; k < tile_k; ++k) {
            float a_val = (tidm < M) ? A[tidm * K + (k0 + k)] : 0.0f;
            float b_val = (tidn < N) ? B[tidn * K + (k0 + k)] : 0.0f;
            c_val += a_val * b_val; // accumulate dot product
        }
    }

    // Write back to C with bounds check
    if (tidm < M && tidn < N) {
        C[tidm * N + tidn] = c_val;
    }
}
```

```
// Tile shapes are parametric and can be optimized
// by compilation backends
const tunable int TM = {16, 32, 64, 128}
const tunable int TN = {16, 32, 64, 128}
const tunable int TK = {8, 16}
// C = A * B.T
kernel void matmul_nt(float* a, float* b, float* c,
                      int M, in N, int K)
{
 // 1D tile of indices
 int rm[TM] = get_global_range(0);
 int rn[TN] = get_global_range(1);
 int rk[TK] = 0 ... TK;
 // 2D tile of accumulators
 float C[TM, TN] = 0;
 // 2D tile of pointers
 float* pa[TM, TK] = a + rm[:,newaxis] + rk * M;
 float* pb[TN, TK] = b + rn[:,newaxis] + rk * K;
 for(int k = K; k >= 0; k -= TK){
   bool check_k[TK] = rk < k;
   bool check_a[TM, TK] = (rm < M)[:,newaxis] && check_k;
   bool check_b[TN, TK] = (rn < N)[:,newaxis] && check_k;
   // load tile operands
   float A[TM, TK] = check_a ? *pa : 0;
   float B[TN, TK] = check_b ? *pb : 0;
   // accumulate
   C += dot(A, trans(B));
   // update pointers
   pa = pa + TK*M;
   pb = pb + TK*N;
 }
 // write-back accumulators
 float* pc[TM, TN] = c + rm[:,newaxis] + rn * M;
 bool check_c[TM,TN] = (rm < M)[:, newaxis] && (rn < N);
 @check_c *pc = C;
}
```

Triton-IR

- Purpose:
  - Tile-level program analysis, transformation, and optimization
  - Enables data-flow analysis, predicated execution, and code generation for GPUs
- LLVM-based IR
  - Extensions for data-flow and control-flow

Structure

1. Modules - basic compilation units for Triton
    a. Composed of functions, global variables, constants, etc
2. Functions - name, return type, optional arguments
3. Basic blocks - straight line code sequences

Support for Data-flow Analysis

1. Types
    a. Multi-dimensional tiles
    b. Tiles cannot change size
2. Instructions
    a. Retiling instructions
    b. Specialized arithmetic instructions

Control-Flow Analysis

Problem: Divergence within tiles

Solution: Predicated Static Single Assignment form

- Essentially just says which elements are active
- “cmpp” and “psi” instructions

```
;pt[i,j], pf[i,j] = (true, false) if x[i,j] < 5
;pt[i,j], pf[i,j] = (false, true) if x[i,j] >= 5
%pt, %pf = icmpp slt %x, 5
@%pt %x1 = add %y, 1
@%pf %x2 = sub %y, 1
; merge values from different predicates
%x = psi i32<8,8> [%pt, %x1], [%pf, %x2]
%z = mul i32<8,8> %x, 2
```

JIT Compiler

Goal: Simplify and compile Triton-IR programs into efficient machine code

1. Machine independent passes
2. Machine dependent passes
3. Auto-tuning engine

Machine Independent Passes

1. Prefetching
    a. Memory operations in loops create latency
    b. Compiler detects loops and adds prefetching code
2. Tile-level Peephole Optimizations
    a. Compiler technique that tried to optimize small sequences of instruction

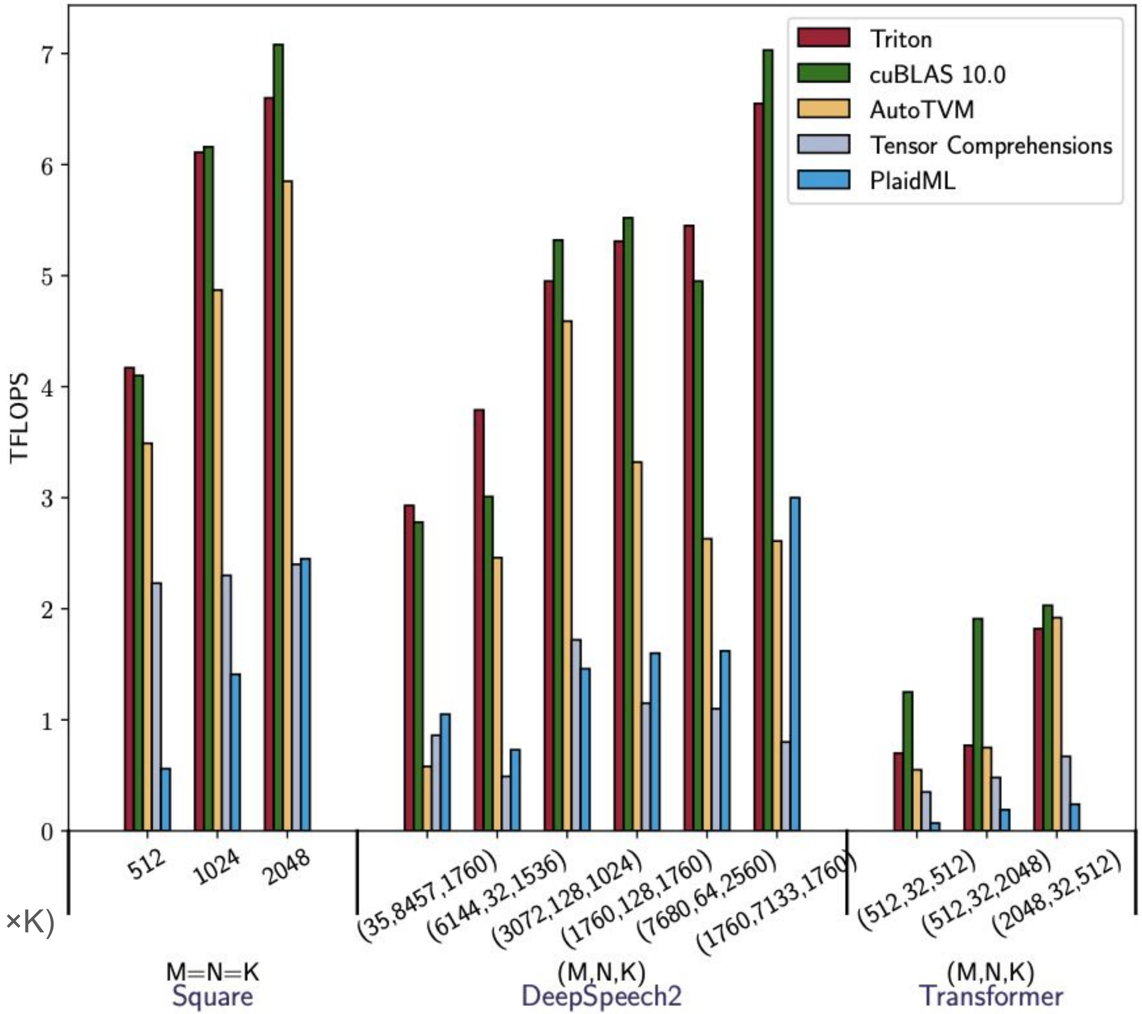
Machine Dependent Passes

1. Hierarchical tiling
   a. Decompose tiles into micro-tiles to better utilize compute/memory
   b. Automatically enumerate and optimize configurations
2. Memory coalescing
   a. Orders threads to better access memory from DRAM
3. Shared memory allocation
   a. Liveness calculations and allocations
4. Shared memory synchronization
   a. Inserts barriers for reads/writes to ensure correctness

Autotuner

- Can directly extract optimization search spaces from the IR
- Concatenates meta-parameters associated with previous optimizations
- This paper only evaluates hierarchical tiling

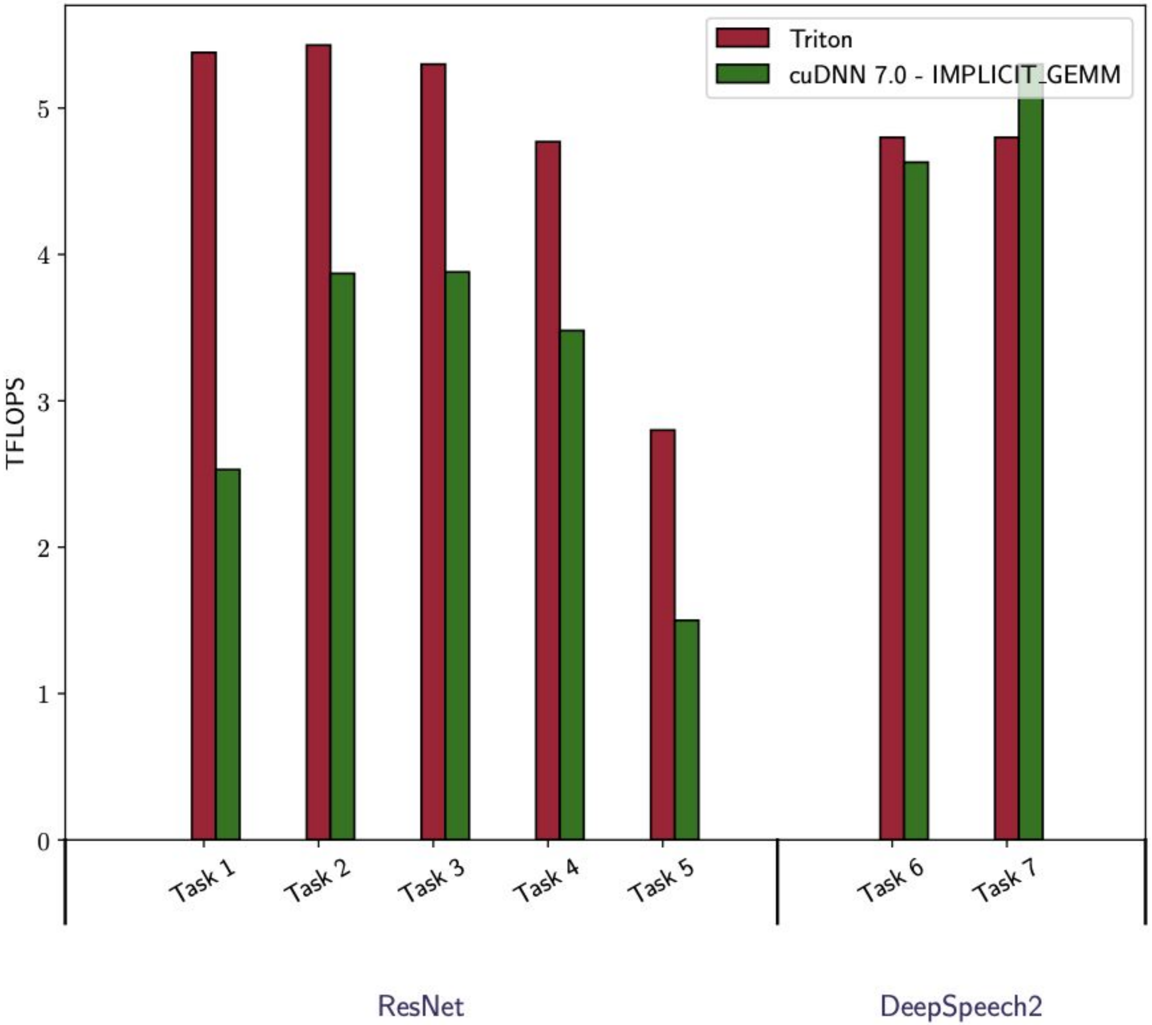
Results



$A = D \times W^T$

$(D \in \mathbf{R}^{(M \times K)}, W \in \mathbf{R}^{(N \times K)})$

Results


Triton
cuDNN 7.0 - IMPLICIT_GEMM
TFLOPS
0
1
2
3
4
5
Task 1
Task 2
Task 3
Task 4
Task 5
Task 6
Task 7
ResNet
DeepSpeech2

Summary

- Introduces Triton
- Split into Triton-C, Triton-IR, Triton JIT Compiler
- Performance on par with vendor libraries