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

Abstract
The validation and deployment of novel research ideas in the field of Deep Learning is often limited by the availability of efficient compute kernels for certain basic primitives. In particular, operations that cannot leverage existing vendor libraries (e.g., cuBLAS, cuDNN) are at risk of facing poor device utilization unless custom implementations are written by experts – usually at the expense of portability. For this reason, the development of new programming abstractions for specifying custom Deep Learning workloads at a minimal performance cost has become crucial.

We present Triton, a language and compiler centered around the concept of tile, i.e., statically shaped multi-dimensional sub-arrays. Our approach revolves around (1) a C-based language and an LLVM-based intermediate representation (IR) for expressing tensor programs in terms of operations on parametric tile variables and (2) a set of novel tile-level optimization passes for compiling these programs into efficient GPU code. We demonstrate how Triton can be used to build portable implementations of matrix multiplication and convolution kernels on par with hand-tuned vendor libraries (cuBLAS / cuDNN), or for efficiently implementing recent research ideas such as shift convolutions.

CCS Concepts → Computing methodologies → Parallel computing methodologies.

Keywords → compiler; neural networks; GPU

1 Introduction
The recent resurgence of Deep Neural Networks (DNNs) was largely enabled [24] by the widespread availability of programmable, parallel computing devices. In particular, continuous improvements in the performance of many-core architectures (e.g., GPUs) have played a fundamental role, by enabling researchers and engineers to explore an ever-growing variety of increasingly large models, using more and more data. This effort was supported by a collection of vendor libraries (cuBLAS, cuDNN) aimed at bringing the latest hardware innovations to practitioners as quickly as possible. Unfortunately, these libraries only support a restricted set of tensor operations, leaving the implementation of novel primitives to experts [13, 17, 25].

This observation has led to the development of various Domain-Specific Languages (DSLs) for DNNs, based on polyhedral machinery (e.g., Tensor Comprehensions [43]) and/or loop synthesis techniques (e.g., Halide [37], TVM [10] and PlaidML[22]). But while these systems generally perform well for certain classes of problems such as depthwise-separable convolutions (e.g., MobileNet [20]), they are often much slower than vendor libraries in practice (see, e.g., Figure 1), and lack the expressivity necessary to implement structured sparsity patterns [28, 31, 47] that cannot be directly specified using affine array indices in nested loops.
These issues have often been addressed by the use of micro-kernels [11, 21] – i.e., hand-written tile-level intrinsics – but this solution requires a lot of manual labor and lacks portability. And while several high-level programming abstractions for tiling have recently been proposed [23, 41], underlying compiler backends still lack support for tile-level operations and optimizations. To this end we present Triton (Figure 2), an open-source intermediate language and compiler for specifying and compiling tile programs into efficient GPU code.

Figure 2. Overview of Triton

The main contributions of this paper are summarized as follows:

- **Triton-C** (Section 3): A C-like language for expressing tensor programs in terms of parametric tile variables. The purpose of this language is to provide a stable interface for existing DNN transcompilers (e.g., PlaidML, Tensor Comprehensions) and programmers familiar with CUDA. Listing 1 shows the Triton-C source code associated with a simple matrix multiplication task.

- **Triton-IR** (Section 4): An LLVM-based Intermediate Representation (IR) that provides an environment suitable for tile-level program analysis, transformation and optimization. Listing 5 shows the Triton-IR code for a Rectified Linear Unit (ReLU) function. Here Triton-IR programs are constructed directly from Triton-C during parsing, but automatic generation from embedded DSLs or higher-level DNN compilers (e.g., TVM) could also be explored in the future.

- **Triton-JIT** (Section 5): A Just-In-Time (JIT) compiler and code generation backend for compiling Triton-IR programs into efficient LLVM bitcode. This includes (1) a set of tile-level, machine-independent passes aimed at simplifying input compute kernels independently of any compilation target; (2) a set of tile-level machine-dependent passes for generating efficient GPU-ready LLVM-IR; and (3) an auto-tuner that optimize any meta-parameter associated with the above passes.

- **Numerical Experiments** (Section 6): A numerical evaluation of Triton that demonstrates its ability to (1) generate matrix multiplication implementations on par with cuBLAS and up to 3× faster than alternatives DSLs on recurrent and transformer neural networks; (2) re-implement cuDNN’s IMPLICIT_GEMM algorithm for dense convolution without performance loss; and (3) create efficient implementations of novel research ideas such as shift-conv [47] modules.

This paper will be prefaced by a brief analysis of the existing related literature (Section 2) and concluded by a summary and directions of future work (Section 7).

### Listing 1. \( C = A \times B^T \) in Triton-C. Keywords specific to Triton are shown in purple.

```
// 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--)
{
    bool check_k[TK] = rk < k;
    bool check_b[TN, TK] = (rn < N)[.., newaxis] & check_k;
    bool check_a[TM, TK] = (rm < M)[.., newaxis] & check_k;
    // load tile operands
    float A[TM, TK] = check_a ? a + pa : 0;
    float B[TN, TK] = check_b ? 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;
```


2 Related Work

The existence of frameworks [1, 9, 36] and libraries for Deep Learning has been critical to the emergence of novel neural network architectures and algorithms. But despite advances in analytical [5, 48] and empirical [6, 30] heuristics for linear algebra compilers, these software still invariably rely on hand-optimized sub-routines (e.g., cuBLAS and cuDNN). This has led to development of various DSLs and
compilers for DNNs, generally based on one of three distinct approaches:

- **Tensor-level IRs** have been used by XLA [16] and Glow [38] to transform tensor programs into predefined LLVM-IR and CUDA-C operation templates (e.g., tensor contractions, element-wise operations, etc.) using pattern-matching.
- **The polyhedral model** [18] has been used by Tensor Comprehensions (TC) [43] and Diesel [14] to parameterize and automate the compilation of one or many DNN layers into LLVM-IR and CUDA-C programs.
- **Loop synthesizers** have been used by Halide [37] and TVM [10] to transform tensor computations into loop nests that can be manually optimized using user-defined (though possibly parametric [11]) schedules.

By contrast, Triton relies on the addition of **tile-level operations** and optimizations into traditional compilation pipelines. This approach provides (1) more flexibility than XLA and Glow; (2) support for non-affine tensor indices contrary to TC and Diesel; and (3) automatic inference of possible execution schedule that would otherwise have to be specified manually to Halide or TVM. The benefits of Triton come at the cost of increased programming efforts – see Listing 2 for implementations of matrix multiplication in these DSLs.

### Listing 2. C = A × B^T in TF, PlaidML, TC and TVM

```plaintext
C = tf.matmul(A, tf.transpose(B))  // TF
C[i, j] = A[i, k] * B[j, k];       // PlaidML
C[i, j] = A[i, k] * B[j, k];       // TC
C[i, j] = tf.reduce_sum(A[i, k] * B[j, k], axis=k); // TVM
```

### 3 The Triton-C Language

The purpose of Triton-C is to provide a stable frontend for existing (and future) DNN transcompilers, as well as programmers familiar with low-level GPU programming. In this section we describe the CUDA-like syntax of Triton-C (Section 3.1), its Numpy-like semantics (Section 3.2) and its “Single-Program, Multiple-Data” (SPMD) programming model (Section 3.3).

#### 3.1 Syntax

The syntax of Triton-C is based on that of ANSI C (and more specifically CUDA-C), but was modified and extended (see Listing 3) to accommodate the semantics and programming model described in the two next subsections. These changes fall into the following categories:

- **Tile declarations**: We added special syntax for declaring multi-dimensional arrays (e.g., `int tile[16, 16]`) so as to emphasize their semantical difference with nested arrays found in ANSI C (e.g., `int tile[16][16]`). Tile shapes must be constant but can also be made parametric with the `tunable` keyword. One-dimensional integer tiles may be initialized using ellipses (e.g., `int range[8] = 0 ... 8`).

#### 3.2 Semantics

- **Broadcasting**: While common C syntax was retained for element-wise array operations (+, -, &, &, etc.), various built-in functions (`dot, trans, get_global_range`) were added to support tile semantics (Section 3.2.1) and the SPMD programming model.

    **Broadcasting**: N-dimensional tiles can be broadcast along any particular axis using the `newaxis` keyword and usual slicing syntax (e.g., `int broadcast [8, 8] = range[:, newaxis]` for stacking columns). Note that slicing tiles to retrieve scalars or sub-arrays is otherwise forbidden.

- **Predication**: Basic control-flow within tile operations (Section 4.3) is achieved through the use of predicated statements via the `@` prefix.

#### 3.2.1 Tile semantics

- **Broadcasting semantics**: The existence of built-in `tile` types and operations (i.e., tile semantics) in Triton-C offers two main benefits. First, it simplifies the structure of tensor programs by hiding important performance details pertaining to intra-tile memory coalescing [12], cache management [32] and specialized hardware utilization [27]. Second, it opens the door for compilers to perform these optimizations automatically, as discussed in Section 5.
2. Broadcasting: the content of both operands is replicated as many times as needed until their shape is identical; an error is emitted if this cannot be done.

Listing 4. Broadcasting semantics in practice

```
int a[16], b[32, 16], c[16, 1];
// a is first reshaped to [1, 16]
// and then broadcast to [32, 16]
int x_1[32, 16] = a[newaxis, :] + b;
// Same as above but implicitly
int x_2[32, 16] = a + b;
// a is first reshaped to [1, 16]
// a is broadcast to [16, 16]
// c is broadcast to [16, 16]
int y[16, 16] = a + c;
```

3.3 Programming Model

The execution of CUDA [33] code on GPUs is supported by an SPMD [4] programming model in which each kernel is associated with an identifiable thread-block in a so-called launch grid. The Triton programming model is similar, but each kernel is single-threaded – though automatically parallelized – and associated with a set of global ranges that varies from instance to instance (see Figure 3). This approach leads to simpler kernels in which CUDA-like concurrency primitives (shared memory synchronization, inter-thread communication, etc.) are inexistent.

The global ranges associated with a kernel can be queried using the `get_global_range(axis)` built-in function in order to create e.g., tiles of pointers as shown in Listing 1.

![Figure 3. Difference between the CUDA and the Triton programming model](image)

4 The Triton IR

Triton-IR is an LLVM-based Intermediate Representation (IR) whose purpose is to provide an environment suitable for tile-level program analysis, transformation and optimization. In this work, Triton-IR programs are constructed directly from Triton-C during parsing, although they could also be generated directly from higher level DSLs in the future.

Triton-IR and LLVM-IR programs share the same high-level structure (recalled in Section 4.1), but the former also includes a number of extensions necessary for tile-level data-flow (Section 4.2) and control-flow (Section 4.3) analysis. These novel extensions are crucial for carrying out the optimizations outlined in Section 5, and for safely accessing tensors of arbitrary shapes as shown in Section 6.

4.1 Structure

4.1.1 Modules

At the highest level, Triton-IR programs consist of one or multiple basic units of compilation known as `modules`. These modules are compiled independently from one another, and eventually aggregated by a linker whose role is to resolve forward declarations and adequately merge global definitions.

Each module itself is composed of functions, global variables, constants and other miscellaneous symbols (e.g., metadata, function attributes).

4.1.2 Functions

Triton-IR function definitions consist of a return type, a name and a potentially empty arguments list. Additional visibility, alignment and linkage specifiers can be added if desired. Function attributes (such as inlining hints) and parameter attributes (such as read-only, aliasing hints) can also

Listing 5. $A = \max(A, 0)$ in Triton-IR. Note that tile shapes are non-parametric here. In this paper their values are instantiated by the Triton-JIT.

```
define kernel void @relu(float * %A, i32 %M, i32 %N) {
prologue:
%rm = call i32 <8 > get_global_range(0);
%rn = call i32 <8 > get_global_range(1);
// broadcast shapes
%1 = reshape i32 <8, 8> %M;
%M0 = broadcast i32 <8, 8> %1;
%2 = reshape i32 <8, 8> %N;
%N0 = broadcast i32 <8, 8> %2;
// broadcast global ranges
%3 = reshape i32 <8, 1> %rm;
%rm_bc = broadcast i32 <8, 8> %3;
%4 = reshape i32 <1, 8> %rm;
%rm_bc = broadcast i32 <8, 8> %4;
// compute mask
%pm = icmp slt %rm_bc , %M0;
%pn = icmp slt %rn_bc , %N0;
%mask = and %pm , %pn;
// compute pointer
%a = splat float*<8, 8> %A;
%S = getelementptr %A, 0, %rm_bc;
%Sx = mul %S, %rm_bc, %M0;
%spa = getelementptr %S, %N0;
// compute result
%result = max %float %a, %S;
// write back
store fp32 <8 , 8> %pa , % result
}```
be specified, allowing compiler backends to perform more aggressive optimizations by, for instance, making better use of read-only memory caches.

This header is followed by a body composed of a list of basic blocks whose interdependencies form the Control Flow Graph (CFG) of the function.

4.1.3 Basic Blocks

Basic blocks are, by definition, straight-line code sequences that may only contain so-called terminator instructions (i.e., branching, return) at their end.

Triton-IR uses the Static Single Assignment (SSA) form, meaning that each variable in each basic block must be (1) assigned to only once and (2) defined before being used. In so doing, each basic block implicitly defines a Data-Flow Graph (DFG) whose different paths correspond to use-def chains in the program’s SSA representation. This form can be created directly from Abstract Syntax Trees (ASTs) as shown in [7].

4.2 Support for Tile-Level Data-Flow Analysis

4.2.1 Types

Multi-dimensional tiles are at the center of data-flow analysis in Triton-IR and can be declared using syntax similar to vector declarations in LLVM-IR. For example, i32<8, 8> is the type corresponding to 8 × 8 32-bit integer tiles. Note that there is no tunable keyword in Triton-IR, hence parameteric shape values must be resolved before programs are generated. In our case, this is done by Triton-JIT’s auto-tuner (Section 5.3).

4.2.2 Instructions

Triton-IR introduces a set of retiling instructions whose purpose is to support broadcasting semantics as described in Section 3.2.2:

- The reshape instruction creates a tile of the specified shapes using data from its input argument. This is particularly useful to re-interpret variables as higher-dimensional arrays by padding their input shapes with ones in preparation of implicit or explicit broadcasting.
- The broadcast instruction creates a tile of the specified shapes by replicating its input argument as many times as necessary along dimensions of size 1 - as shown in Figure 4.

```
(a) [3 x 1] input  (b) [1 x 3] input

<table>
<thead>
<tr>
<th>a</th>
<th>a</th>
<th>a</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
</tr>
</tbody>
</table>
```

**Figure 4.** The broadcast <3,3> instruction

Usual scalar instructions (cmp, getelementptr, add, load...) were preserved and extended to signify element-wise operations on tile operands. Finally, Triton-IR also exposes specialized arithmetic instructions for transpositions (trans) and matrix multiplications (dot).

4.3 Support for Tile-Level Control-Flow Analysis

One problem that arises from the existence of tile-level operations in Triton-IR is the inexpressibility of divergent control flow within tiles. For example, a program may need to partially guard tile-level loads against memory access violations, but this cannot be achieved using branching since tile elements cannot be accessed individually.

```
Listing 6. Tile-Level Predication in Triton-IR

;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 = cmp bhi %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
```

We propose to solve this issue through the use of the Predicated SSA (PSSA) form [8] and ψ-functions [39]. This requires the addition of two instruction classes (see Listing 6) to Triton-IR:

- The cmp instructions [8] are similar to usual comparison (cmp) instructions, except for the fact that they return two opposite predicates instead of one.
- The psi instruction merges instructions from different streams of predicated instructions.

5 The Triton-JIT compiler

The goal of Triton-JIT is to simplify and compile Triton-IR programs into efficient machine code, via a set of machine-independent (Section 5.1) and machine-dependent (Section 5.2) passes backed by an auto-tuning engine (Section 5.3).

5.1 Machine-Independent Passes

5.1.1 Pre-Fetching

Tile-level memory operation inside loops can be problematic, as they may induce severe latency that cannot be hidden in the absence of enough independent instructions. It is however possible to mitigate this problem in Triton-IR directly by detecting loops and adding adequate prefetching code where necessary (See Listing 7).

```
Listing 7. Automatic pre-fetching

B0:
%z0 = getelementptr %1, %2
%z0 = load %p
B1:
%z1 = phi [%z0, B0], [%z1, B1]
%z = load %p
%z1 = getelementptr %p, %3
%z1 = load %p
```

Arithmetic instructions for transpositions (trans) and matrix multiplications (dot).
5.1.2 Tile-Level Peephole Optimization

The presence of tile-level operations in Triton-IR offers new opportunities for peephole [29] optimizers. For instance, chains of transpositions can be simplified using the identity $X = (X^T)^T$ for any tile $X$. We believe that other algebraic properties related to e.g., diagonal tiles could also be exploited in the future.

5.2 Machine-Dependent Passes

We now present a set of optimization passes for machines that follow the high-level model shown in Figure 5. Specifically, the optimizations performed by Triton-JIT consist of (1) hierarchical tiling, (2) memory coalescing, (3) shared memory allocation and (4) shared memory synchronization.

5.2.1 Hierarchical Tiling

Nested tiling strategies (see Figure 5) aim at decomposing tiles into micro-tiles and eventually nano-tiles in order to fit a machine’s compute capabilities and memory hierarchy as tightly as possible. While this technique is routinely used in auto-tuning frameworks [34, 40], the structure of Triton-IR makes it possible to automatically enumerate and optimize valid nested tiling configurations for any expressible program (and without the need for polyhedral machinery).

![Hierarchical Tiling](image)

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

5.2.2 Memory Coalescing

Memory accesses are said to be coalesced when adjacent threads simultaneously access nearby memory locations. This is important because memory is usually retrieved in large blocks from DRAM.

Because Triton-IR programs are single-threaded and automatically parallelized, our compiler backend is able to order threads internally within each micro-tile so as to avoid uncoalesced memory accesses when possible. This strategy reduces the number of memory transactions necessary to load a tile column (see Figure 6).

![Coalesced DRAM Accesses](image)

Figure 6. Uncoalesced (a) and coalesced (b) DRAM accesses. Different threads are shown in different colors.

5.2.3 Shared Memory Allocation

Tile-level operations that have high arithmetic intensity (e.g., dot) can benefit from temporally storing their operands in fast shared memory. The purpose of the Shared Memory Allocation pass is to determine when and where a tile should be stashed to this space. This can be done, as illustrated in Figure 7, by first calculating the live range of each variable of interest, and then using the linear-time storage allocation algorithm proposed in [15].

![Shared Memory Allocation](image)

Figure 7. Shared Memory Allocation

5.2.4 Shared Memory Synchronization

Reads from and write to shared memory are asynchronous in our machine model. The goal of the Shared Memory Synchronization pass automatically inserts barriers in the generated GPU source code so as to preserve program correctness. This is done by detecting read-after-writes (RAW) and write-after-read(WAR) hazards using forward data-flow analysis with the following data-flow equations:

$$\text{in}^{(RAW)}_s = \bigcup_{p \in \text{pred}(s)} \text{out}^{(RAW)}_p$$

$$\text{in}^{(WAR)}_s = \bigcup_{p \in \text{pred}(s)} \text{out}^{(WAR)}_p$$

$$\text{out}^{(RAW)}_s = \begin{cases} \emptyset & \text{if } \text{in}^{(RAW)}_s \cap \text{read}(s) \neq \emptyset \text{ (barrier)} \\ \text{in}^{(RAW)}_s \cup \text{write}(s) & \text{otherwise} \end{cases}$$

$$\text{out}^{(WAR)}_s = \begin{cases} \emptyset & \text{if } \text{in}^{(WAR)}_s \cap \text{write}(s) \neq \emptyset \text{ (barrier)} \\ \text{in}^{(WAR)}_s \cup \text{read}(s) & \text{otherwise} \end{cases}$$
5.3 Auto-tuner

Traditional auto-tuners [42, 45] typically rely on hand-written parameterized code templates to achieve good performance on pre-defined workloads. By contrast, Triton-JIT can extract optimization spaces directly from Triton-IR programs by simply concatenating meta-parameters associated with each of the above optimization passes.

In this work, only the Hierarchical Tiling pass is considered, leading to no more than 3 tiling parameters per dimension per tile. These parameters are then optimized using an exhaustive search over powers of two between (a) 32 and 128 for tile sizes; (b) 8 and 32 for micro-tile sizes; and (c) 1 and 4 for nano-tile sizes. Better auto-tuning methods could be used in the future.

6 Numerical Experiments

In this section we evaluate the performance Triton on various workloads from the Deep Learning literature. We used an NVIDIA GeForce GTX1070 and compared our system against the most recent vendor libraries (cuBLAS 10.0, cuDNN 7.0) as well as related compiler technology (AutoTVM, TC, PlaidML). When applicable, we auto-tuned these DSLs for each individual problem size following official documentation guidelines.

6.1 Matrix Multiplication

Matrix multiplication tasks of the form: $A = D \times W^T$ ($D \in \mathbb{R}^{M \times K}, W \in \mathbb{R}^{N \times K}$) are at the heart of neural network computations. Here we consider a variety of tasks from recurrent (DeepSpeech2 [5]) and transformer [44] neural networks; we report their performance in Figure 8.

![Figure 8. Performance of matrix multiplication](image)

Triton and cuBLAS are generally on par with each other, and achieve more than 90% of the device’s peak performance on certain tasks. CuBLAS, however, remains faster than Triton on shallow transformer neural networks thanks to the use of a 3D algorithm [2] which splits deep reductions into independent chunks to provide more parallelism when $M$ and $N$ are too small. Otherwise, existing DSLs are 2-3x slower than our solution – except for TVM (< 2x slower) when input shapes are multiples of 32.

6.2 Convolutions

Convolutional Neural Networks (CNNs) are an important class of machine learning models which should be well supported by DSLs and compilers. They are based around convolutional layers (Figure 9a) whose implementation as matrix multiplication (Figure 9b) is necessary to make use of specialized tensor-processing hardware – yet unsupported by existing DSLs. Here we benchmark a Triton re-implementation of cuDNN’s “IMPLICIT_GEMM” algorithm (Section 6.2.1) and provide the first fused kernel available for shifted convolutions (Section 6.2.2). We implemented these routines using look-up tables of pointer increments, as shown in Listing 8.
little engineering resources for optimizing kernels of lesser importance. When fast algorithms are not available (e.g., DeepSpeech2), cuDNN and Triton are on par.

6.2.2 Shift Convolutions

We finally consider an implementation of Task1-5 from Table 1 as shifted convolutions – a novel approach to CNNs (see Figure 9a). We compare our implementation of a fused shift-conv module in Triton (Listing 8) against that of a naive implementation relying on a hand-written shift kernel and a separate call to cuBLAS. We also report the maximum attainable performance when shift is not done (i.e., $1 \times 1$ convolution). As we can see in Figure 11, our Triton implementation is able to almost entirely hide the cost of shifting.

7 Conclusions

In this paper we presented Triton, an open-source language and compiler for expressing and compiling tiled neural network computations into efficient machine code. We showed that the addition of just a few data- and control-flow extensions to LLVM-IR could enable various tile-level optimization passes which jointly lead to performance on-par with vendor libraries. We also proposed Triton-C, a higher-level language in which we were able to concisely implement efficient kernels for novel neural network architectures for CNNs.

Directions of future work includes support for tensor cores, implementation of quantized kernels [26] and integration into higher level DSLs.

8 Acknowledgements

This work was supported by the NVIDIA Graduate Fellowship.
References


