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

ACM Reference Format:
Philippe Tillet, H. T. Kung, and David Cox. 2019. Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL '19), June 22, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3315508.3329973

MAPL '19, June 22, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6719-6/19/06…$15.00
https://doi.org/10.1145/3315508.3329973

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.

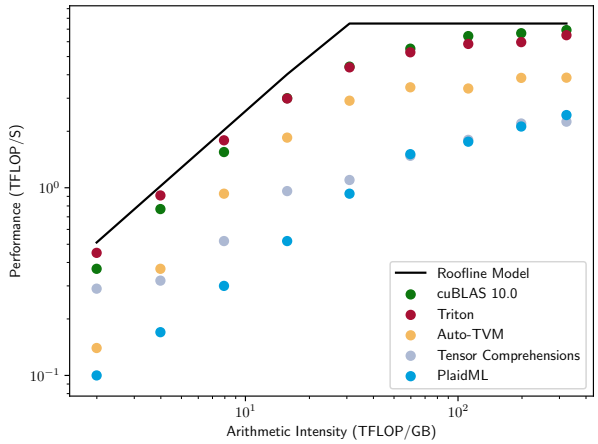


Figure 1. Performance of various implementations of $C = AB^T$ vs. Roofline [46] Model (NVIDIA GeForce GTX1070), where $A \in \mathbb{R}^{1760\times1760}$ and $B \in \mathbb{R}^{N\times1760}$ as N modulates arithmetic intensity.

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[1] intermediate language and compiler for specifying and compiling tile programs into efficient GPU code.

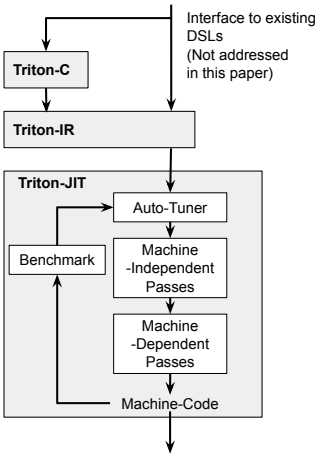


Figure 2. Overview of Triton

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.

[1] http://triton-lang.org

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

```
// 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;
}
```

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).

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. $C = A \times B^T$ in TF, PlaidML, TC and TVM

```
C = tf.matmul(A, tf.transpose(B))      // TF
C[i, j: I, J] = +(A[i, k] * B[j, k]); // PlaidML
C(i, j) +=! A(i, k) * B(j, k)          // TC
tvm.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[35]-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 accomodate 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`).

Listing 3. Grammar extensions for Triton-C. We assume the existence of certain C constructs shown in blue.

```
// Broadcasting semantics
slice          : ':' | 'newaxis'
slice_list     : slice | slice_list ',' slice
slice_expr     : postfix_expr | expr '[' slice_list ']'
// Range initialization
constant_range : expr '...' expr
// Intrinsics
global_range   : 'get_global_range' '(' constant ')'
dot            : 'dot' '('  expr ',' expr ')'
trans          : 'trans' '(' expr ',' expr ')'
intrinsic_expr : global_range | dot | trans
// Predication
predicate_expr : '@' expr
// Tile extensions for abstract declarators
abstract_decl  : abstract_decl | '[' constant_list ']'
// Extensions of C expressions
expr           :  expr | constant_range  |  slice_expr
               | intrinsic_expr
// Extensions of C specifiers
storage_spec : storage_spec | 'kernel'
type_spec    : type_spec | 'tunable'
// Extensions of C statements
statement    : statement | predicate_expr statement
```

Built-in function: 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 Semantics

3.2.1 Tile 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.

3.2.2 Broadcasting Semantics

Tiles in Triton-C are strongly typed in the sense that certain instructions statically require their operands to obey strict shape constraints. For example, a scalar may not be added to an array unless it is first appropriately broadcast. *Broadcasting semantics* [35] provide a set of rules to perform these conversions implicitly (see Listing 4 for an example):

1. *Padding*: the shape of the shortest operand is left-padded with ones until both operands have the same dimensionality.

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.

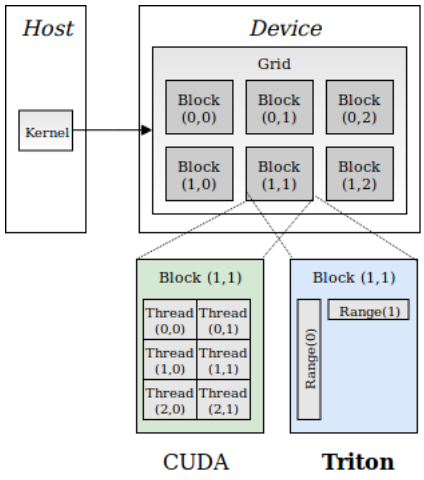


Figure 3. Difference between the CUDA and the Triton programming model

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.

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> %rn;
  %rn_bc = broadcast i32<8, 8> %4;
  ; compute mask
  %pm = icmp slt %rm_bc, %M0;
  %pn = icmp slt %rn_bc, %N0;
  %msk = and %pm, %pn;
  ; compute pointer
  %A0 = splat float*<8, 8> %A;
  %5 = getelementptr %A0, %rm_bc;
  %6 = mul %rn_bc, %M0;
  %pa = getelementptr %5, %6;
  ; compute result
  %a = load %pa;
  %_0 = splat float<8, 8> 0;
  %result = max %float %a, %_0;
  ; write back
  store fp32<8, 8> %pa, %result
}
```

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

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

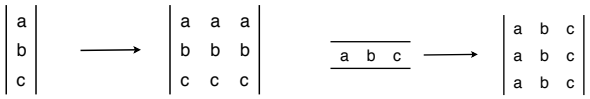
4.2.1 Types

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 parametric 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) $[3 \times 1]$ input **(b)** $[1 \times 3]$ input

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 = 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
```

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 `cmpp` 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:
 %p0 = getelementptr %1, %2
B1:
 %p = phi [%p0,B0], [%p1,B1]
 %x = load %p
 ; increment pointer
 %p1 = getelementptr %p, %3
```

```
B0:
 %p0 = getelementptr %1, %2
 %x0 = load %p0
B1:
 %p = phi [%p0,B0], [%p1,B1]
 %x = phi [%x0,B0], [%x1,B1]
 ; increment pointer
 %p1 = getelementptr %p, %3
 ; prefetching
 %x1 = load %p
```

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).

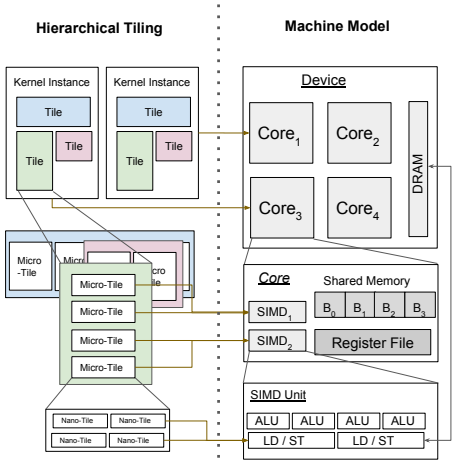


Figure 5. Hierarchical Tiling in the Triton-IR Machine Model

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).

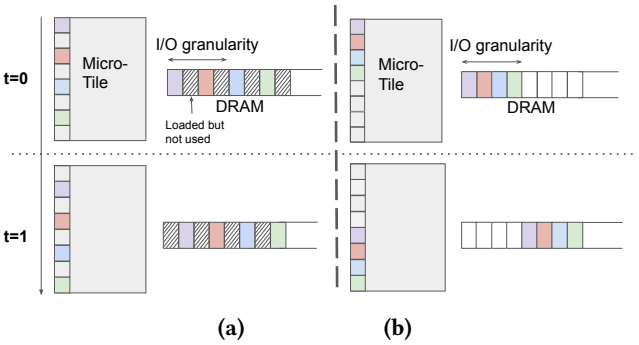


Figure 6. Uncoalesced (a) and coalesced (b) DRAM accesses. Different threads are shown in different colors.

5.2.3 Shared Memory Allocation

Tile-level operations that have high arithmetic intensity (e.g., `dot`) can benefit from temporarily 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].

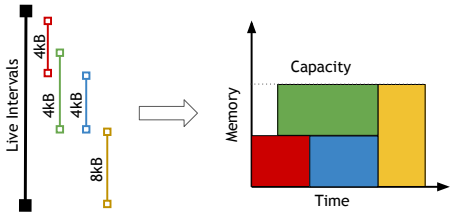


Figure 7. Shared Memory Allocation

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:

$$in_s^{(RAW)} = \bigcup_{p \in \text{pred}(s)} out_p^{(RAW)}$$

$$in_s^{(WAR)} = \bigcup_{p \in \text{pred}(s)} out_p^{(WAR)}$$

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

$$out_s^{(WAR)} = \begin{cases} \emptyset & \textbf{if } in_s^{(WAR)} \cap write(s) \neq \emptyset \text{ (barrier)} \\ in_s^{(WAR)} \cup read(s) & \textbf{otherwise} \end{cases}$$

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 [3]) and transformer [44] neural networks; we report their performance in Figure 8.

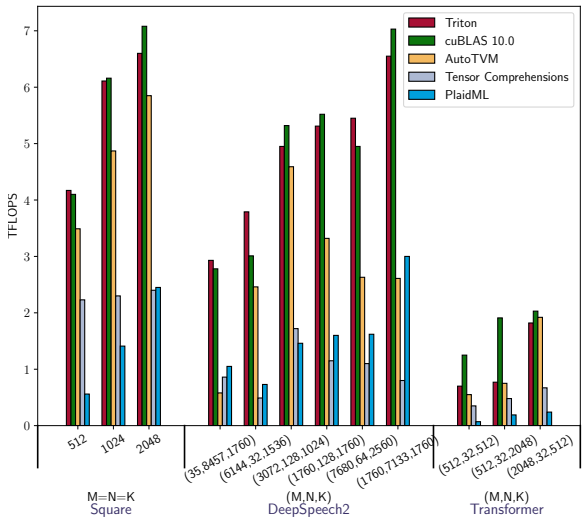
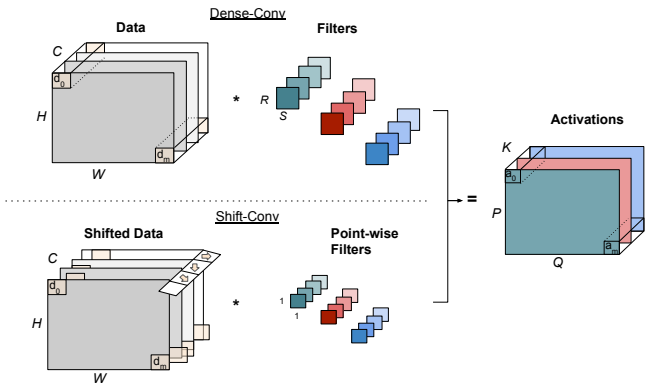


Figure 8. Performance of matrix multiplication

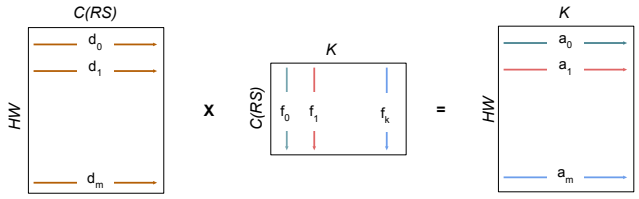
Triton and cuBLAS are generally on par with each other, and achieve more than 90% of the device's peak performance on certain tasks. CuBLAS, however, remains faster than Triton on shallow transformer neural networks thanks to the use of a 3D algorithm [2] which splits deep reductions into independent 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.



(a) A convolutional layer



(b) Equivalent matrix multiplication

Figure 9. Dense and shifted convolutional layers (a) viewed as matrix multiplication (b)

6.2.1 Dense Convolutions

The convolutional layers considered in this subsection are from the Deep Learning literature and shown in Table 1.

As shown in Figure 10, Triton outperforms cuDNN's implementation of IMPLICIT_GEMM for ResNet. This may be due to the fact that cuDNN also maintains better algorithms for 3×3 convolutions (i.e., Winograd [25]), thereby leaving

	H	W	C	B	K	R	S	Application
Task 1	112	112	64	4	128	3	3	ResNet [19]
Task 2	56	56	128	4	256	3	3	ResNet
Task 3	28	28	256	4	512	3	3	ResNet
Task 4	14	14	512	4	512	3	3	ResNet
Task 5	7	7	512	4	512	3	3	ResNet
Task 6	161	700	1	8	64	5	5	DeepSpeech2
Task 7	79	341	32	8	32	5	10	DeepSpeech2

Table 1. Convolution tasks considered in this paper

little engineering resources for optimizing kernels of lesser importance. When fast algorithms are not available (e.g., DeepSpeech2), cuDNN and Triton are on par.

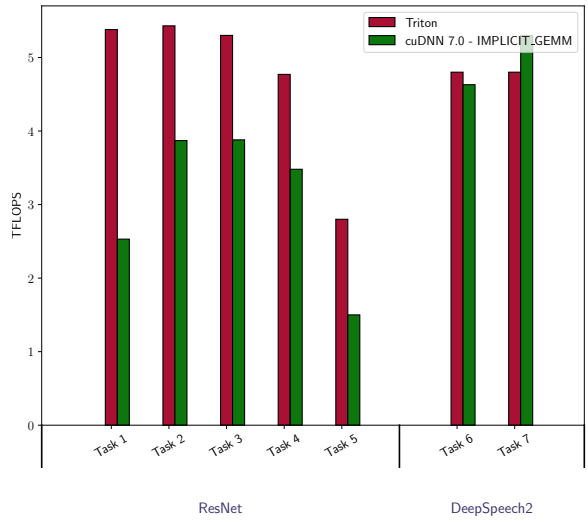


Figure 10. Performance of implicit matrix multiplication

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., 1×1 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.

Listing 8. shift-convolutions in Triton-C

```
const tunable int TM = {16, 32, 64, 128};
const tunable int TN = {16, 32, 64, 128};
const tunable int TK = {8};

__constant__ int* delta = alloc_const int[512];

for(int c = 0; c < C; c++)
  delta[c] = c*H*W + shift_h[c]*W + shift_w[c]


void shift_conv(restrict read_only float *a,
                restrict read_only float *b, float *c,
                int M, int N, int K){
  int rxa[TM] = get_global_range[TM](0);
  int ryb[TN] = get_global_range[TN](1);
  int rka[TK] = 0 ... TK;
  int rkb[TK] = 0 ... TK;
  float C[TM, TN] = 0;
  float* pxa[TM, TK] = a + rxa[:, newaxis];
  float* pb[TN, TK] = b  + ryb[:, newaxis] + rkb*N;
  __constant__ int* pd[TK] = delta + rka;
  for(int k = K; k > 0; k = k - TK){
    int delta[TK] = *pd;
    float *pa[TM, TK] = pxa + delta[newaxis, :];
    float a[TM, TK] = *pa;
    float b[TN, TK] = *pb;
    C = dot(a, trans(b), C);
    pb = pb + TK*N;
    pd = pd + TK;
  }
  int rxc[TM] = get_global_range[TM](0);
  int ryc[TN] = get_global_range[TN](1);
  float* pc[TM, TN] = c + rxc[:, newaxis] + ryc*M;
  bool checkc0[TM] = rxc < M;
  bool checkc1[TN] = ryc < N;
  bool checkc[TM, TN] = checkc0[:, newaxis] && checkc1;
  @checkc *pc = C;
}
```

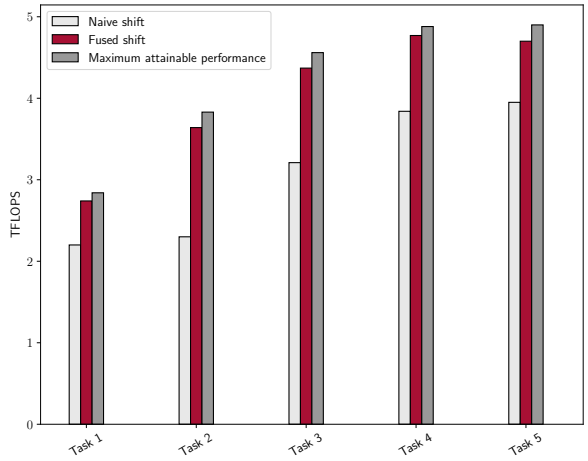


Figure 11. Performance of shifted convolutions in Triton

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

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*.

[2] R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar. 1995. A three-dimensional approach to parallel matrix multiplication. *IBM Journal of Research and Development* 39, 5 (Sep. 1995), 575–582. https://doi.org/10.1147/rd.395.0575

[3] Dario Amodei, Rishita Anubhai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Jingdong Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, Erich Elsen, Jesse Engel, Linxi Fan, Christopher Fougner, Tony Han, Awni Y. Hannun, Billy Jun, Patrick LeGresley, Libby Lin, Sharan Narang, Andrew Y. Ng, Sherjil Ozair, Ryan Prenger, Jonathan Raiman, Sanjeev Satheesh, David Seetapun, Shubho Sengupta, Yi Wang, Zhiqian Wang, Chong Wang, Bo Xiao, Dani Yogatama, Jun Zhan, and Zhenyao Zhu. 2015. Deep Speech 2: End-to-End Speech Recognition in English and Mandarin. *CoRR* abs/1512.02595 (2015). arXiv:1512.02595 http://arxiv.org/abs/1512.02595

[4] M. Auguin and F. Larbey. [n. d.]. Opsila: an advanced SIMD for numerical analysis and signal processing. ([n. d.]), 311–318.

[5] Bin Bao and Chen Ding. 2013. Defensive Loop Tiling for Shared Cache. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO) (CGO '13)*. IEEE Computer Society, Washington, DC, USA, 1–11.

[6] Muthu Manikandan Baskaran, Albert Hartono, Sanket Tavarageri, Thomas Henretty, J. Ramanujam, and P. Sadayappan. 2010. Parameterized Tiling Revisited. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '10)*. ACM, New York, NY, USA, 200–209. https://doi.org/10.1145/1772954.1772983

[7] Matthias Braun, Sebastian Buchwald, Sebastian Hack, Roland Leissa, Christoph Mallon, and Andreas Zwinkau. 2013. Simple and Efficient Construction of Static Single Assignment Form. In *Proceedings of the 22Nd International Conference on Compiler Construction (CC'13)*. 102–122.

[8] Lori Carter, Beth Simon, Brad Calder, Larry Carter, and Ferrante. [n. d.]. Predicated Static Single Assignment. In *Proceedings of the PACT 1999 Conference on Parallel Architectures and Compilation Techniques*.

[9] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *CoRR* abs/1512.01274 (2015).

[10] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q. Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: End-to-End Optimization Stack for Deep Learning. *CoRR* abs/1802.04799 (2018).

[11] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q. Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: End-to-End Optimization Stack for Deep Learning. *CoRR* abs/1802.04799 (2018).

[12] Jack W. Davidson and Sanjay Jinturkar. 1994. Memory Access Coalescing: A Technique for Eliminating Redundant Memory Accesses. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI '94)*. ACM, New York, NY, USA, 186–195. https://doi.org/10.1145/178243.178259

[13] Greg Diamos, Shubho Sengupta, Bryan Catanzaro, Mike Chrzanowski, Adam Coates, Erich Elsen, Jesse Engel, Awni Hannun, and Sanjeev Satheesh. 2016. Persistent RNNs: Stashing Recurrent Weights On-Chip. In *Proceedings of The 33rd International Conference on Machine Learning*.

[14] Venmugil Elango, Norm Rubin, Mahesh Ravishankar, Hariharan Sandanagobalane, and Vinod Grover. 2018. Diesel: DSL for Linear Algebra and Neural Net Computations on GPUs. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL 2018)*. ACM, New York, NY, USA, 42–51. https://doi.org/10.1145/3211346.3211354

[15] Jordan Gergov. 1999. Algorithms for Compile-time Memory Optimization. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '99)*.

[16] Google Inc. 2017. *Tensorflow XLA*. https://www.tensorflow.org/performance/xla/

[17] Scott Gray and Alex Radford ans Diederik P. Kingma. 2017. Gpu kernels for block-sparse weights. *CoRR* abs/1711.09224 (2017).

[18] M. Griebl, C. Lengauer, and S. Wetzel. 1998. Code generation in the polytope model. In *Proceedings. 1998 International Conference on Parallel Architectures and Compilation Techniques (Cat. No.98EX192)*. 106–111. https://doi.org/10.1109/PACT.1998.727179

[19] K. He, X. Zhang, S. Ren, and J. Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778. https://doi.org/10.1109/CVPR.2016.90

[20] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *CoRR* abs/1704.04861 (2017). arXiv:1704.04861 http://arxiv.org/abs/1704.04861

[21] Jianyu Huang and Robert A. van de Geijn. 2016. *BLISlab: A Sandbox for Optimizing GEMM*. FLAME Working Note #80, TR-16-13. The University of Texas at Austin, Department of Computer Science. http://arxiv.org/pdf/1609.00076v1.pdf

[22] Intel AI. 2018. *PlaidML*. https://www.intel.ai/reintroducing-plaidml

[23] Andrew Kerr, Duane Merrill, Julien Demouth, and John Tran. [n. d.]. CUTLASS: Fast Linear Algebra in CUDA C++. ([n. d.]). https://devblogs.nvidia.com/cutlass-linear-algebra-cuda/

[24] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1 (NIPS'12)*.

[25] Andrew Lavin and Scott Gray. 2016. Fast Algorithms for Convolutional Neural Networks. *(CVPR'16)*.

[26] Darryl D. Lin, Sachin S. Talathi, and V. Sreekanth Annapureddy. 2016. Fixed Point Quantization of Deep Convolutional Networks. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48 (ICML'16)*. JMLR.org, 2849–2858. http://dl.acm.org/citation.cfm?id=3045390.3045690

[27] Matt Martineau, Patrick Atkinson, and Simon McIntosh-Smith. 2017. *Benchmarking the NVIDIA V100 GPU and Tensor Cores*. Springer.

[28] Bradley McDanel, Surat Teerapittayanon, and H.T. Kung. 2017. Embedded Binarized Neural Networks. In *Proceedings of the 2017 International Conference on Embedded Wireless Systems and Networks (EWSN ’17)*. Junction Publishing, USA, 168–173. http://dl.acm.org/citation.cfm?id=3108009.3108031

[29] W. M. McKeeman. 1965. Peephole Optimization. *Commun. ACM* 8, 7 (July 1965), 443–444. https://doi.org/10.1145/364995.365000

[30] Sanyam Mehta, Rajat Garg, Nishad Trivedi, and Pen-Chung Yew. 2016. TurboTiling: Leveraging Prefetching to Boost Performance of Tiled Codes. In *Proceedings of the 2016 International Conference on Supercomputing (ICS '16)*. ACM, New York, NY, USA, Article 38, 12 pages.

[31] Sharan Narang, Eric Undersander, and Gregory F. Diamos. 2017. Block-Sparse Recurrent Neural Networks. *CoRR* abs/1711.02782 (2017).

[32] Rajib Nath, Stanimire Tomov, and Jack Dongarra. 2011. Accelerating GPU Kernels for Dense Linear Algebra. In *Proceedings of the 9th International Conference on High Performance Computing for Computational Science (VECPAR'10)*. Springer-Verlag, Berlin, Heidelberg, 83–92.

http://dl.acm.org/citation.cfm?id=1964238.1964250

[33] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. 2008. Scalable Parallel Programming with CUDA. *Queue* 6, 2 (March 2008), 40–53. https://doi.org/10.1145/1365490.1365500

[34] Cedric Nugteren. 2017. CLBlast: A Tuned OpenCL BLAS Library. *CoRR* abs/1705.05249 (2017). arXiv:1705.05249 http://arxiv.org/abs/1705.05249

[35] Travis Oliphant. 2006–. NumPy: A guide to NumPy. USA: Trelgol Publishing. (2006–). http://www.numpy.org/ [Online; accessed <today>].

[36] PyTorch. 2016. . https://github.com/pytorch/pytorch

[37] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*.

[38] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Summer Deng, Roman Dzhabarov, James Hegeman, Roman Levenstein, Bert Maher, Nadathur Satish, Jakob Olesen, Jongsoo Park, Artem Rakhov, and Misha Smelyanskiy. 2018. Glow: Graph Lowering Compiler Techniques for Neural Networks. *CoRR* abs/1805.00907 (2018).

[39] Arthur Stoutchinin and Francois de Ferriere. 2001. Efficient Static Single Assignment Form for Predication. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 34)*. IEEE Computer Society, Washington, DC, USA, 172–181. http://dl.acm.org/citation.cfm?id=563998.564022

[40] Philippe Tillet and David Cox. 2017. Input-aware Auto-tuning of Compute-bound HPC Kernels. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '17)*. ACM.

[41] Didem Unat, Tan Nguyen, Weiqun Zhang, Muhammed Nufail Farooqi, Burak Bastem, George Michelogiannakis, Ann Almgren, editor="Kunkel Julian M. Shalf, John", Pavan Balaji, and Jack Dongarra. 2016". TiDA: High-Level Programming Abstractions for Data Locality Management.

[42] Field G. Van Zee and Robert A. van de Geijn. 2015. BLIS: A Framework for Rapidly Instantiating BLAS Functionality. *ACM Trans. Math. Softw.* 41, 3 (June 2015).

[43] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. *CoRR* abs/1802.04730 (2018).

[44] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. *CoRR* abs/1706.03762 (2017). arXiv:1706.03762 http://arxiv.org/abs/1706.03762

[45] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. 2000. Automated Empirical Optimization of Software and the ATLAS Project. *PARALLEL COMPUTING* 27 (2000), 2001.

[46] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM* 52, 4 (April 2009), 65–76. https://doi.org/10.1145/1498765.1498785

[47] Bichen Wu, Alvin Wan, Xiangyu Yue, Peter H. Jin, Sicheng Zhao, Noah Golmant, Amir Gholaminejad, Joseph Gonzalez, and Kurt Keutzer. 2017. Shift: A Zero FLOP, Zero Parameter Alternative to Spatial Convolutions. *CoRR* abs/1711.08141 (2017).

[48] Jiacheng Zhao, Huimin Cui, Yalin Zhang, Jingling Xue, and Xiaobing Feng. 2018. Revisiting Loop Tiling for Datacenters: Live and Let Live. In *Proceedings of the 2018 International Conference on Supercomputing (ICS '18)*. ACM, New York, NY, USA, 328–340.