TVM on Arm A64FX processor

Srinivas Eswar Mitesh Kothari

December 7, 2020

Abstract

This project studied how one can use a high level compiler for efficient kernel construction on a novel architecture. We implement a multilayer perceptron on an Arm A64FX based machine and optimize the kernels involved using the TVM compiler. We describe the techniques used achieve 10× speedup over a naive implementation of the neural network. These techniques enable us to come close to 50% of `NumPy` performance without careful parameter tuning and assembly level optimizations. We briefly discuss the ways this study can be extended to more complicated networks and kernels.

1 Introduction

Machine Learning [MRC+20], and neural networks in particular, is transforming multiple industries and is the killer application consuming a lot of compute cycles. The computationally demanding nature of neural network training has led to a surge in specialized hardware and software systems to address this need.

In this project we will be working with two major components: a deep learning compiler (TVM [CMJ+18]) and the ARM A64FX based HPC Cluster (Octavius [Gal20]). TVM is an end-to-end optimizing compiler for deep learning. Current deep learning frameworks generate a computational graph for every network and attempt to map it to various operators exposed by the hardware. TVM exposes both graph-level and operator-level optimizations to support a wide variety of hardware backends. In particular, TVM has support for ARM based targets [Apa18]. The Octavius cluster [Gal20] is the latest member of Georgia Tech's Rogues Gallery [YRC+19]. It provides access to the Fujitsu designed A64FX processor which powers the world's most powerful supercomputer Fugaku [Sat20].

1.1 Problem Statment

Run and optimize a deep learning application on the A64FX architecture using the TVM compiler.

1.2 Outcomes

We implemented a Multilayer Perceptron (MLP) [FHT01], a simple feed forward neural network, using TVM and tuned its performance on Octavius. The main outcomes of this project are listed below.

- Implement the MLP using the Tensor Expression interface language in TVM.
- Optimize the bottleneck, matrix-matrix multiplication (DGEMM [LHKK79]), using transformations from the class.
- Optimize the network to achieve 10× speedup over the naive implementation generated by TVM and within 50% of `NumPy`.

2 Background

2.1 MLP - Multilayer Perceptron

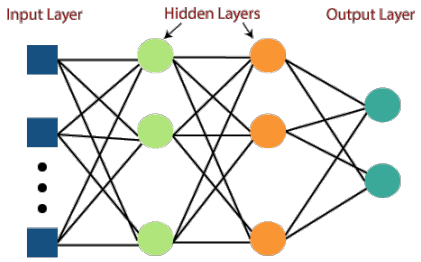

Figure 1: Multilayer Perceptron

In our study we will focus on implementing the MLP Neural network. It is a simple neural network consisting of multiple fully connected layers with threshold activation function. For a better understanding we can take the scenario of classifying images of handwritten digits from the MNIST dataset as one of the ten digits. As shown in as shown in Fig. 1, for the MLP we could use, 3 fully connected layers where the output of the two hidden layers is followed by a ReLU activation. Finally the output of the last layer would be fed into a softmax function to generate the probability vector of image being classified as one of the digits.

2.2 TVM [CMJ+18]

TVM is a deep learning compiler stack for various architectures aimed at compiling and building highly optimized executables of deep learning models for target architectures. The modules in TVM are shown in Fig. 2. The primary modules of focus in our approach are Runtime, Target , Te(Tensor Expression) and Topi. Runtime module contains methods for loading and executing compiled artefacts on different architectures. Target module is responsible for generating target specific code translation. Te module contains the fundamental building blocks for building expressions using Tensors in TVM. TOPI - TVM Operator Inventory contains numpy-like abstractions/methods for various Te arithmetic and Neural network operations.

2.3 A64FX processor

The A64FX [Sat20] is a Fujitsu designed ARM microprocessor of Aarch64 with ARMv8.2 SVE support. It has 48 cores (+4 assistant cores) , 3MiB L1 Data cache (64KiB/core) , 3MiB L1 Instruction cache (64KiB/core) and 32 MiB L2 cache (8 x 4). With 256 bytes cache lines , 512

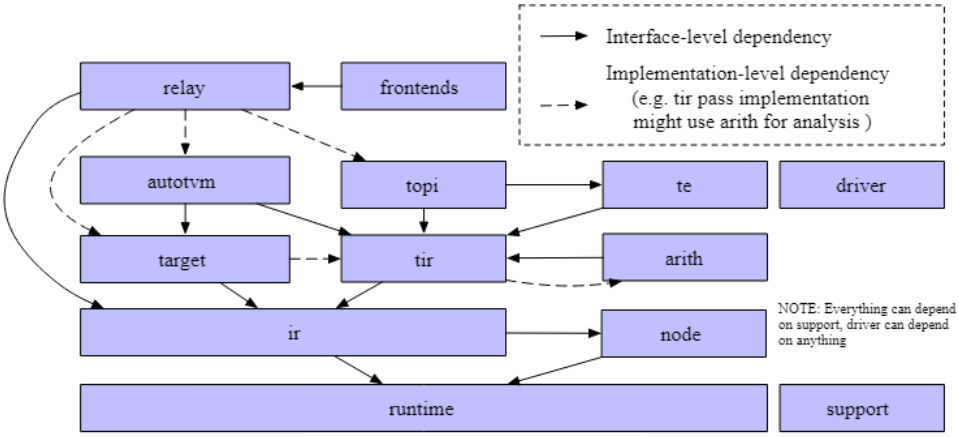

Figure 2: TVM Modules

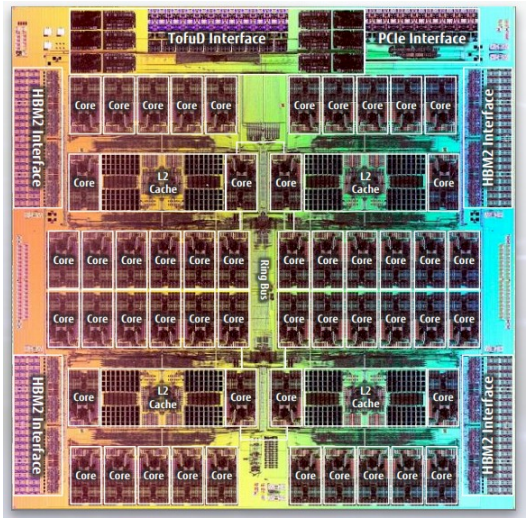

Figure 3: A64FX Architecture

vector widths and 32 GB HBM providing 1 TB/s bandwidth, it can provide high parallelism and excellent performance. The architecture is described in detail in Fig. 3. Additionally, it supports 4 operand FMA (D = C + A*B) instead of the standard (C += A*B) , greatly improving performance for GEMM like operations. Consequently, it also powers the Fukagu supercomputer (Top500 leader!) and Octavius cluster in the Rogues gallery.

3 Setup

For our application we had to build TVM and other dependencies manually from source, as they were not available directly on the Octavius cluster. We have specified our exact environment for perform our experiments and hence, our experiment results can be replicated with the configuration specified below.

- System Info: Octavius Cluster

- Clang - v9.0.1 (C/C++ Compiler)
- CMake - v3.19.0
- Micro-architecture - aarch64
- LLVM - v12.0.0 (Target - aarch64-unknown-linux-gnu , Host-cpu - a64fx)
- TVM - v0.8.dev0

4 Application

4.1 Tensor Expression (TE) module

To understand how MLP is implemented we need to first understand the basics of how expressions are represented in TVM. Using the Tensor Expression module we can build expressions. These expressions can be layer evaluated by providing the appropriate data. The following code snippets provides an insight into how we can express and execute a simple operation such as vector addition.

```
A = te.placeholder((n,), name='a')
B = te.placeholder((n,), name='b')
C = te.compute(A.shape, lambda i: A[i] + B[i], name='c')
```

This would just define the operation for vector addition, similar to a function. TVM also provides the functionality to see the Low level IR for the above expression. It is done using the following code snippet.

```
s = te.create_schedule(C.op)
tvm.lower(s, [A, B, C], simple_mode=True)
```

The output is as follows:

```
produce c {
  for (i, 0, 100) {
    c[i] = (a[i] + b[i])
  }
}
```

Finally we can actually execute the expression defined above by building it and performing the operation using some data as in the following code snippet.

```
mod = tvm.build(s, [A, B, C])
a = tvm.nd.array(np.ones(2))
b = tvm.nd.array(np.ones(2))
c = tvm.nd.array(np.zeros(2))

mod(a, b, c)
```

4.2 MLP - TE Implementation

Now we can build more complex expressions using the TE statements such as building the expression for the MLP Neural network. The exact code snippet for Representing a MLP using Te module in TVM is as follows. The actual execution using data is similar as shown above.

```
A  = te.placeholder((128, 1, 28, 28))
W1 = te.placeholder((128,28*28))
b1 = te.placeholder((128,))

W2 = te.placeholder((64,128))
b2 = te.placeholder((64,))

W3 = te.placeholder((10,64))
b3 = te.placeholder((10,))

A1 = topi.nn.flatten(A)
A2 = topi.nn.relu( topi.nn.dense(A1,W1,b1) )
A3 = topi.nn.relu( topi.nn.dense(A2,W2,b2) )
A4 = topi.nn.dense(A3,W3,b3)
A5 = topi.nn.softmax(A4)
```

The key operation is the `topi.nn.dense(A1, W1, b1)` layer. Here **A1** is the data/activation entering the layer which interacts with the weight matrix **W1** and bias **b1** in the following manner.

$$\mathbf{C} = \mathbf{A1}\mathbf{W1}^{\mathsf{T}} + \mathbf{b1}^{T}$$

The bias vector has the same number of the elements as the number of rows pf **W1** and is added to each row of $\mathbf{A1}\mathbf{W1}^{\mathsf{T}}$. The `topi.nn.relu` function is an elementwise function which operates on **C** to zero-out all the negative entries to produce **A2**.

5 Optimizing DGEMM

The main computation kernel in the MLP is DGEMM. In order to investigate this operation we consider the DGEMM kernel independent of the MLP. The naive implementation of DGEMM is the default form of `topi.nn.dense` which is shown below.

```
k = te.reduce_axis((0, K), "k")
A = te.placeholder((M, K), name="A")
B = te.placeholder((K, N), name="B")
C = te.compute((M, N), lambda x, y: te.sum(A[x, k] * B[k, y], axis=k), name="C")
```

Code Listing 1: Naive DGEMM

Listing 1 shows the computation of $\mathbf{C} = \mathbf{AB}$ where $\mathbf{C} \in \mathbb{R}^{M\times N}$, $\mathbf{A} \in \mathbb{R}^{M\times K}$, and $\mathbf{B} \in \mathbb{R}^{M\times N}$. TVM then generates the IR as shown below.

```
primfn(A_1: handle, B_1: handle, C_1: handle) -> ()
attr = {"global_symbol": "main", "tir.noalias": True}
buffers = {C: Buffer(C_2: Pointer(float32), float32, [1024, 1024], []),
           A: Buffer(A_2: Pointer(float32), float32, [1024, 1024], []),
           B: Buffer(B_2: Pointer(float32), float32, [1024, 1024], [])}
buffer_map = {A_1: A, B_1: B, C_1: C} {
for (x: int32, 0, 1024) {
 for (y: int32, 0, 1024) {
  C_2[((x*1024) + y)] = 0f32
  for (k: int32, 0, 1024) {
   C_2[((x*1024) + y)] = ((float32*)C_2[((x*1024) + y)]
   + ((float32*)A_2[((x*1024) + k)]
```

```
    * (float32*)B_2[((k*1024) + y)]))
    }
   }
  }
 }
}
```

Code Listing 2: Naive DGEMM IR

5.1 Optimizations

As seen from Listing 2, TVM generates the simple 3 `for` loops version of matrix multiply which is not performant on modern architectures [GG08]. We shall now employ known optimization techniques for DGEMM [WY20, vdGH16] to improve upon this IR. The final TVM schedule and generated IR is shown in Listing 3 and Listing 4 respectively. We shall point out individual optimizations in this final IR.

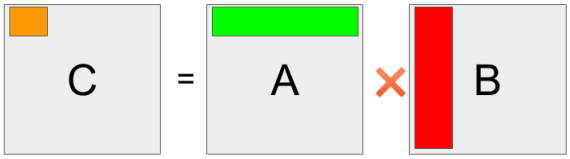

Figure 4: Blocked matrix multiply

Blocking : The first optimization attempted was blocking. Fig. 4 shows the computations and traversals needed for a simple matrix multiply. **A** is accessed in a row-wise manner while **B** is accessed column-wise. Blocking will aid memory accesses by repeatedly hitting all the entries of **A**,**B**, and **C** if we choose the blocks small enough to fit in cache. The blocked version of the IR with blocking factor 32 is shown next. Blocking is achieved in TVM by utilizing the `s_par[C].tile` command where the variable `s` holds the current schedule of the operation. All 3 loops `x`, `y`, and `k` are blocked.

Loop Permutation : The naive blocking scheme will benefit further by moving the contiguous memory accesses to the inner most position. In `Python`, all arrays are row-major therefore we should move the `y` loops to the innermost position which will benefit accesses to matrices **A** and **C**. **B** is still accessed in a column-wise manner and must be handled separately. The final loop order is given by the following schedule `s_par[C].reorder(xo, yo, ko, xi, ki, yi)`.

Vectorization : The A64FX has large vector widths and so we vectorize the innermost loop to make use of the vector FMA units. This is achieved by the TVM command `s_par[CC].vectorize(yc)` in conjunction with `s_par[CC].unroll(ki)`.

```
bnx = 32
bny = 32
packedB = te.compute((N / bny, K, bny), lambda x, y, z: B[y, x * bny + z], name="
                                                packedB")
C_pack = te.compute(
    (M, N),
```

```
    lambda x, y: te.sum(A[x, k] * packedB[y // bny, k, tvm.tir.indexmod(y, bny)],
                                                    axis=k),
    name="C",
)
s_par = te.create_schedule(C_pack.op)

CC = s_par.cache_write(C_pack, "global")

xo, yo, xi, yi = s_par[C_pack].tile(C_pack.op.axis[0], C_pack.op.axis[1], bnx, bny
                                                )

s_par[CC].compute_at(s_par[C_pack], yo)

xc, yc = s_par[CC].op.axis

(k,) = s_par[CC].op.reduce_axis
ko, ki = s_par[CC].split(k, factor=4)
s_par[CC].reorder(ko, xc, ki, yc)
s_par[CC].unroll(ki)
s_par[CC].vectorize(yc)

s_par[C_pack].parallel(xo)

x, y, z = s_par[packedB].op.axis
s_par[packedB].vectorize(z)
s_par[packedB].parallel(x)
```

Code Listing 3: Optimized DGEMM

Array Packing : Now we deal with the fact that we are not moving along the contiguous dimension of **B**. In order to achieve this we need to repack **B** in column major order. Rather than simply packing as $K \times N$ matrix in column-major order we consider the blocking factor, say B_y, and reshape **B** as $\frac{K}{B_y} \times N \times B_y$ tensor. Here every block used in the outer (`xo`,`yo`) iteration will be as a continuous slice of the tensor. This is seen as the intermediate buffer `packedB`.

Write Caches : The write back to the blocks of **C** are not sequential. TVM allows us access to a "cache array" to hold in temporary values and only write back to global **C** when the blocks are ready. This is achieved via the the TVM primitive `s_par.cache_write`.

Parallelization : Finally we can parallelize execution for all the different write caches via the use of `s_par[C].parallel(xo)`. Since all the write blocks are independent of each other and we only read from **A** and **B** this operation is safe to parallelize.

```
primfn(A_1: handle, B_1: handle, C_1: handle) -> ()
attr = {"global_symbol": "main", "tir.noalias": True}
buffers = {C: Buffer(C_2: Pointer(float32), float32, [1024, 1024], []),
           A: Buffer(A_2: Pointer(float32), float32, [1024, 1024], []),
           B: Buffer(B_2: Pointer(float32), float32, [1024, 1024], [])}
buffer_map = {A_1: A, B_1: B, C_1: C} {
attr [packedB: Pointer(float32)] "storage_scope" = "global";
allocate(packedB, float32x32, [32768]) {
for (x: int32, 0, 32) "parallel" {
```

```
 for (y: int32, 0, 1024) {
  # Vectorized initialization of packedB
  packedB[ramp(((x*32768) + (y*32)), 1, 32)] = (float32x32*)B_2[ramp(((y*1024) + (
                                                x*32)), 1, 32)]
 }
}
for (x.outer: int32, 0, 32) "parallel" {
 attr [C.global: Pointer(float32)] "storage_scope" = "global";
 allocate(C.global, float32, [1024]);
 for (y.outer: int32, 0, 32) {
  for (x.c.init: int32, 0, 32) {
   C.global[ramp((x.c.init*32), 1, 32)] = broadcast(0f32, 32)
  }
  for (k.outer: int32, 0, 256) {
   for (x.c: int32, 0, 32) {
    # Write Cache 1
    C.global[ramp((x.c*32), 1, 32)] = ((float32x32*)C.global[ramp((x.c*32), 1, 32)
                                                ] +
    (broadcast((float32*)A_2[(((x.outer*32768) + (x.c*1024)) + (k.outer*4))], 32)
    *(float32x32*)packedB[ramp(((y.outer*32768) + (k.outer*128)), 1, 32)]))
    # Write Cache 2
    C.global[ramp((x.c*32), 1, 32)] = ((float32x32*)C.global[ramp((x.c*32), 1, 32)
                                                ] +
    (broadcast((float32*)A_2[((((x.outer*32768) + (x.c*1024)) + (k.outer*4)) + 1)]
                                                , 32)
    *(float32x32*)packedB[ramp((((y.outer*32768) + (k.outer*128)) + 32), 1, 32)]))
    # Write Cache 3
    C.global[ramp((x.c*32), 1, 32)] = ((float32x32*)C.global[ramp((x.c*32), 1, 32)
                                                ] +
    (broadcast((float32*)A_2[((((x.outer*32768) + (x.c*1024)) + (k.outer*4)) + 2)]
                                                , 32)
    *(float32x32*)packedB[ramp((((y.outer*32768) + (k.outer*128)) + 64), 1, 32)]))
    # Write Cache 4
    C.global[ramp((x.c*32), 1, 32)] = ((float32x32*)C.global[ramp((x.c*32), 1, 32)
                                                ] +
    (broadcast((float32*)A_2[((((x.outer*32768) + (x.c*1024)) + (k.outer*4)) + 3)]
                                                , 32)
    *(float32x32*)packedB[ramp((((y.outer*32768) + (k.outer*128)) + 96), 1, 32)]))
   }
  }
  for (x.inner: int32, 0, 32) {
   for (y.inner: int32, 0, 32) {
    # Write back in order
    C_2[((((x.outer*32768) + (x.inner*1024)) + (y.outer*32)) + y.inner)] =
    (float32*)C.global[((x.inner*32) + y.inner)]
   }
  }
 }
}
}
}
```

Code Listing 4: Optimized DGEMM IR

5.2 DGEMM Results

We can now look at the timing results for these optimizations. The results for two different matrix sizes are show in Tables 1 and 2. Each of timings are the average of 100 runs of the experiments in case of Table 1 and 10 for Table 2 for the slower runs without vectorization and parallelization. The effects of the optimizations are shown after they are applied successively. The matrix sizes involved are small due to the inordinate amount of time TVM takes to generate code for larger iteration sizes. Generating code for matrix dimensions 4096 did not finish in over 8 hours and thus we limit our experiment to these smaller sizes. Each of the input matrices for Table 1 occupies 4 MiB of memory to store and 16 MiB for those in Table 2. Thus Table 1 computations are completely in L2 cache whereas Table 2 does spill a bit out of cache.

Table 1: Results for $(M, K, N) = (1024,1024,1024)$ with blocking factors (32,32,4)

Optimization	Time	Speed Up
Baseline	5.785	-
Blocking	5.789	1
Loop Permutation	6.028	0.96
Vectorization	1.202	4.813
Array Packing	0.764	7.572
Write Cache	0.584	9.906
Parallelization	0.032	180.781
NumPy	0.139	41.619

From Table 1 we can see that blocking and loop permutation have little to no effect on the performance relative to baseline. This is because all matrices fit in cache and blocking should not affect performance. We do see a slight slow down after loop permutation but we attribute this to just measurement noise. Vectorization offers the first big jump in performance adding close to 10× speedup when compared to the Baseline after packing **B**. Utilizing all the cores also provides a healthy 20× speedup over the vectorized code.

Table 2: Results for $(M, K, N) = (2048,2048,2048)$ with blocking factors (32,32,4)

Optimization	Time	Speed Up
Baseline	176.996	-
Blocking	188.445	0.939
Loop Permutation	176.95	1
Vectorization	10.97	16.135
Array Packing	5.173	34.215
Write Cache	4.393	40.29
Parallelization	0.162	1092.568
NumPy	0.079	2240.456

The results for the larger run are mostly similar as seen from Table 2. Again vectorization and parallelization provide the largest boost to performance with 40× and 25× speedups respectively. In this case the input data is out of cache with all 3 matrices (and the write cache) occupying a total of ≈ 50 MiB of memory but the A64FX has an incredible 1 TB/s DRAM bandwidth which makes this transfer almost negligible. We do see the effect of loop permutation aiding cache performance in this case. In both cases the TVM generate code is comparable to that from `NumPy` coming close

to 50% of it for the larger inputs.

5.3 MLP Integration

Finally, we would like to integrate these optimizations into MLP network which we created in Section 4. One thing to notice is that in the fully connected layer the matrix multiplication operation is slightly different. We perform $\mathbf{A} = \mathbf{D}\mathbf{W}^{\mathsf{T}}$, where $\mathbf{W}$ is the weights matrix of the fully connected layer and $\mathbf{D}$ is the data input to the layer. The Tensor Expression for this layer can be expressed as shown below.

```
matmul = te.compute(
 (batch, out_dim),
 lambda i, j: te.sum(data[i, k].astype(out_dtype) * weight[j, k].astype(out_dtype)
                                                  , axis=k),
 name="T_dense",
 tag="dense",
)
```

Code Listing 5: Dense Layer Tensor Expression

The crucial information to notice here is that we access both arrays `data` and `weights` in row-major order and thus can avoid the packing optimizations. We modified this layer of the MLP to utilize all the other optimizations discussed except array packing. We look at the runtime for inferring a batch of 128 images in Table 3. The timings are consistent with that from Section 5.1

Table 3: MLP run

Optimization	Time	Speed Up
Baseline	0.070448	-
Vectorization	0.006842	10.296
Parallelization	0.006837	10.304
NumPy	0.003386	20.806

and we are able to get within 50% of `NumPy` performance with standard loop transformations.

6 Discussion and Conclusions

The key learning from this project was that the classic loop transformations studied in class is competitive with highly optimized DGEMM kernel found in `NumPy`. TVM provides a nice abstraction to employ these transformations directly but we still had to implement them by hand. Integrating these into the automatic code generator of TVM to recognize a DGEMM operation and perform these transformations on its own would be an interesting extension to this work. Another slight drawback in this project was the unability to utilize all the hardware features present on the A64FX. Currently, we were only able to utilize the large vector units only. Another extension would be to use the high bandwidth DRAM effectively but that would involve optimizing the `tvm.build` process which is outside the scope of this work. Some possible extensions would be using AutoTVM for better/optimal parameters for our application and performing other optimizations using Tir module which contains lower level and target specific optimizations.

To summarise, this project implemented a multilayer perceptron on an Arm A64FX based machine and achieved 10× speedup over a naive implementation of the neural network. These techniques enable us to come close to 50% of `NumPy`.

References

[Apa18] Apache. Tvm docs. `https://tvm.apache.org/docs/index.html`, 2018.

[CMJ+18] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Tvm: An automated end-to-end optimizing compiler for deep learning, 2018.

[FHT01] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The elements of statistical learning*. Springer series in statistics New York, 2001.

[Gal20] Rogues Gallery. Octavius cluster. `http://crnch.gatech.edu/rg-arm-hpc`, 2020.

[GG08] Kazushige Goto and Robert A van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software (TOMS)*, 34(3):1–25, 2008.

[LHKK79] Chuck L Lawson, Richard J. Hanson, David R Kincaid, and Fred T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Transactions on Mathematical Software (TOMS)*, 5(3):308–323, 1979.

[MRC+20] Peter Mattson, Vijay Janapa Reddi, Christine Cheng, Cody Coleman, Greg Diamos, David Kanter, Paulius Micikevicius, David Patterson, Guenther Schmuelling, Hanlin Tang, et al. Mlperf: An industry standard benchmark suite for machine learning performance. *IEEE Micro*, 40(2):8–16, 2020.

[Sat20] Mitsuhisa Sato. The supercomputer "fugaku" and arm-sve enabled a64fx processor for energy-efficiency and sustained application performance. In *2020 19th International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 1–5. IEEE, 2020.

[vdGH16] Robert van de Geijn and Jianyu Huang. How to optimize gemm. `https://github.com/flame/how-to-optimize-gemm`, 2016.

[WY20] Jian Weng and Ruofei Yu. How to optimize gemm on cpu. `https://tvm.apache.org/docs/tutorials/optimize/opt_gemm.html#sphx-glr-tutorials-optimize-opt-gemm-py`, 2020.

[YRC+19] Jeffrey S Young, Jason Riedy, Thomas M Conte, Vivek Sarkar, Prasanth Chatarasi, and Sriseshan Srikanth. Experimental insights from the rogues gallery. In *2019 IEEE International Conference on Rebooting Computing (ICRC)*, pages 1–8. IEEE, 2019.