MEMA Runtime Framework: Minimizing External Memory Accesses for TinyML on Microcontrollers

Andrew Sabot*
asabot@g.harvard.edu
Harvard Universtiy
Cambridge, MA, USA

Vikas Natesh*
vnatesh@g.harvard.edu
Harvard Universtiy
Cambridge, MA, USA

H.T. Kung
kung@harvard.edu
Harvard Universtiy
Cambridge, MA, USA

Wei-Te Ting
weiteting@g.harvard.edu
Harvard Universtiy
Cambridge, MA, USA

ABSTRACT

We present the MEMA framework for the easy and quick derivation of efficient inference runtimes that minimize external memory accesses for matrix multiplication on TinyML systems. The framework accounts for hardware resource constraints and problem sizes in analytically determining optimized schedules and kernels that minimize memory accesses. MEMA provides a solution to a well-known problem in the current practice, that is, optimal schedules tend to be found only through a time consuming and heuristic search of a large scheduling space. We compare the performance of runtimes derived from MEMA to existing state-of-the-art libraries on ARM-based TinyML systems. For example, for neural network benchmarks on the ARM Cortex-M4, we achieve up to a 1.8x speedup and 44% energy reduction over CMSIS-NN.

KEYWORDS

tinyML, matrix multiplication, arithmetic intensity, outer product, memory access, computation scheduling, neural networks

1 INTRODUCTION

Small Internet of things (IoT) devices have become increasingly common, and used in a growing number of fields including healthcare and consumer products[26]. As these devices grow more popular, the amount of data collected increases, driving the demand for computation and machine learning (ML), or TinyML [20], on these systems. However, due to size and power constraints, these devices are heavily limited in their available memory, bandwidth, and computation power. In addition, TinyML devices may be deployed as a distributed system (e.g., for AR/VR workloads [12]) with limited communication bandwidth. Thus, there is a need to support efficient computation for machine learning applications on a wide variety of system configurations for such devices.

Machine learning models involve operations such as convolutional layers, fully connected layers, ReLU, max pooling, and batchnorm. Computations in convolutional and fully connected layers of convolutional neural networks (CNNs) and attention layers of transformers may be implemented as matrix multiplications (MMs) (see, e.g., [21, 28]). As a result, MM makes up a substantial amount of the runtime. When MMs do not fit entirely in the local memories of computing hardware (e.g., registers and caches on CPUs), additional data transfers to and from external memory are needed. This additional IO increases energy consumption and latency.

TinyML hardware characteristics and capabilities can vary significantly, so optimal choices of kernels and schedules may differ for the same problem across multiple devices. Consequently, developing a runtime framework that can find optimal schedules for the wide variety of TinyML systems is key for efficient inference.

To characterize the capabilities of system architectures, roofline plots, (see, e.g., [29]) are commonly used. The slope of the slanted line is the memory IO bandwidth and the horizontal line is the peak computation throughput. The horizontal position of the ridge point represents the minimum *arithmetic intensity* (arithmetic operations performed per IO operation) required to achieve the peak computation throughput offered by the hardware. The optimal schedule for minimizing memory accesses is a function of the roofline plot. For example, some microcontroller units (MCUs), e.g., the ARM Cortex-M4, have relatively high memory bandwidths compared to their computation power [15]. For these systems, the ridge point is associated with a smaller arithmetic intensity.

In this paper, we present the MEMA runtime framework for producing efficient inference runtimes for TinyML that minimize external memory accesses. The framework analyzes the hardware and computational problem to derive IO-efficient runtime schedules. We use the roofline model to reason about the MEMA approach, focusing on tiny devices with small local memories. For the three architectures evaluated in this paper (ARM Cortex-M4, M7, and A72), we present their roofline plots in Figure 2. The MEMA framework schedules computations to increase arithmetic intensity for two reasons: 1) increasing computation throughput when left of the ridge point and 2) decreasing the required IO bandwidth, and thereby energy for memory IO when right of the ridge point.

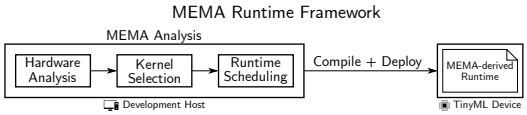


Figure 1: Overview of MEMA runtime framework.

The approach of selecting the appropriate library and kernels based on hardware characteristics is a common practice on CPUs [25]. However, these techniques have not yet been applied to MCUs. We cannot directly transfer decisions in the CPU domain to the MCU domain since the analytical arguments are very different due to disparities in the hardware (e.g., differences in the memory hierarchy and available instructions). As a result, in this paper, we introduce and validate the backbone analysis for MCUs.

Section 2 provides background on matrix multiplication and introduces related works. Section 3 overviews the main objectives of the MEMA runtime framework and describes how streaming as a scheduling technique maximizes data reuse in local memory

*Both authors contributed equally to this research.

tinyML Research Symposium'23, March 2023, San Jose, CA
2023.

so derived runtimes can operate on TinyML devices with severely constrained memory footprint. Section 4 breaks down the general MEMA analysis process for a tiled MM. Section 5 gives an example of how to use the analysis from Section 4 to derive a MEMA schedule on real hardware. Section 6 describes the methodology and benchmarks used in our evaluation of MEMA. Section 7 demonstrates the performance of MEMA runtimes compared to the state-of-the-art libraries on several TinyML devices.

This paper makes the following contributions:

- MEMA runtime framework that reduces total memory IO requirements and hides IO times in computation times for generated runtimes. To our knowledge, MEMA is the first such framework aiming to ease the challenge of designing such schedules for TinyML systems.
- MEMA formalizes the streaming framework to support the proper shaping and allocation of input and result tiles (A, B, and C) in local memory on MCUs with highly constrained memory systems (Section 4)
- MEMA analytically derives optimal tiling for multiple bit widths without the need for searching, as shown in Figure 6.
- Empirical results demonstrating MEMA runtime performance on real hardware (Section 7).

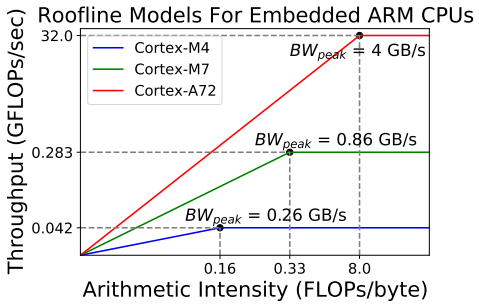


Figure 2: Roofline plots for ARM Cortex-M4, M7, and A72. The ridge point of each device is denoted as a black dot. Systems with a ridge point further to the right are bandwidth-bound, indicating that a higher arithmetic intensity will increase throughput. Conversely, systems with a ridge point further to the left have large memory bandwidths relative to computation throughput.

2 BACKGROUND AND RELATED WORKS

2.1 Partitioning a Matrix Multiplication into Computation Blocks

For MMs where A, B, and C do not all fit in local memory, we must partition, or tile, the MM into smaller $m \times k \times n$ computation blocks so the inputs to each block fit in local memory (Figure 3a).

Algorithm 1 describes a block-partitioned MM where, for each value of t_3 in the outermost loop, an outer-product-based MM between two submatrices ($M \times k$ and $k \times N$ matrices) is performed. We denote this scheme as $M \rightarrow N \rightarrow K$ to reflect the order of the nested loops. For this outer product MM, the scheme performs M-first block computations (t_1 in the innermost loop) as opposed to N-first block computations. There are **six schemes** for scheduling

Algorithm 1: M-first MM using $m \times k \times n$ blocks.

```
// loops on M/m × K/k × N/n computation blocks
for t3 = 0 → K,   t3+ = k do
  for t2 = 0 → N,   t2+ = n do
    for t1 = 0 → M,   t1+ = m do
      // loops for a single computation block
      for k* = t3 → t3 + k,   k*++ do
        for j = t2 → t2 + n,   j++ do
          for i = t1 → t1 + m,   i++ do
            C[i+t1·m][j+t2·n] ⟵ C[i+t1·m][j+t2·n] +
              A[i+t1·m][k*+t3·k] · B[k*+t3·k][j+t2·n];
```

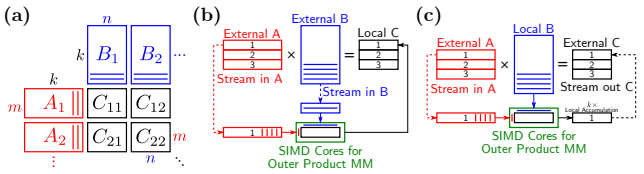


Figure 3: (a) MM for each value of t_3 in Algorithm 1 between $M \times k$ and $k \times N$ matrices using $m \times k \times n$ computation blocks. A_i and B_i are tiles of A and B, with lines indicating column and row vectors, respectively. C_{ij} are tiles of C. (b) We can hold rows of C stationary in the local memory and stream in rows of A and B. (c) By streaming computed rows of C to external memory, we can hide the IO time within the compute time. When k is sufficiently large, the computation time for each row of C exceeds the time to write the row back to external memory. In contrast to (b), this scheme need only hold one row of C as intermediary results. (b) and (c) are both examples of schedules that can be generated by MEMA.

computation blocks: $M \rightarrow N \rightarrow K$, $N \rightarrow M \rightarrow K$, $M \rightarrow K \rightarrow N$, $N \rightarrow K \rightarrow M$, $K \rightarrow M \rightarrow N$, and $K \rightarrow N \rightarrow M$.

2.2 Stationary and Streamed Data

For each MM computation, A, B, and C tiles must be loaded from external memory. When data is kept stationary in local memory, we are able to reduce the number of tiles fetched from external memory. We designate data kept in local memory for multiple computations as stationary (e.g., B matrix in Figure 3b). Data fetched from external memory and used once before being evicted (e.g., A matrix in Figure 3b) is considered "streamed". Accumulated results may be streamed out to external memory (e.g., computed rows of C in Figure 3b). Streaming techniques have been used in prior works such as [14], but our novelty lies in using streaming when automating runtime schedule design.

Using MEMA we can decrease the total amount of data streamed to/from external memory. When the computation time is larger than the streaming time for streamed data items, we are able to hide the streaming time (Figure 3) and decrease the required local memory size. The IO to write back a portion of C can be overlapped with computation of the next portion of C. We assume that there is sufficient bandwidth to also stream in matrix A from external memory (as is the case of the Cortex-M4, with a ridge point further to the left in Figure 2c). The streaming method works similarly when A, B, or C is kept stationary. For all three methods, smaller

tiles of C are accumulated in-place to reduce data movement. If external bandwidth is insufficient, compression techniques, such as [31] can be applied to the data being streamed in.

2.3 Related Works

CMSIS-NN [18] is a set of kernels for common neural network operations, focusing on performance in throughput and latency, and minimizing memory footprints of neural networks on ARM Cortex-M processors. By using inner products and fixing the loop order, CMSIS-NN is not able to maximize data reuse.

In addition to CMSIS-NN, recent frameworks such as MCUNet [20] perform loop tiling and unrolling for neural network layer operations such as convolution. MCUNet only considers the problem size and available local memory when choosing tile sizes and does not explicitly minimize memory bandwidth usage. In contrast, our work uses both local memory and roofline characteristics of the specific device to derive efficient tile sizes. In addition, we leverage outer products and loop reordering to increase data reuse, reducing latency and energy consumption (Figure 6).

ARM provides two libraries for MM on ARM Cortex-A devices: ARM Compute Library for machine learning (ARMCL [6]) and ARM Performance Library (ARMPL [5]). While ARMPL and ARMCL use outer product-based methods for MM, their computation schedule and tiling strategy is based on Goto's algorithm [13], which does not minimize IO bandwidth usage. By minimizing external accesses, we are able to outperform ARMCL and ARMPL (see Figure 7). GOTO's algorithm [13] is a classical algorithm based on data streaming for high-performance MM on CPUs, underlying OpenBLAS [30] and Intel MKL [2]. CUTLASS [4] is an open-source C++ CUDA BLAS library for Nvidia GPUs. CUTLASS is similar to our work in using outer product formulations for MM.

TensorFlow Lite Micro (TFLM) [10] is an ML inference framework for deep learning on embedded systems. Hardware vendors can contribute their own kernels to TFLM, allowing programmers to deploy ML models to many architectures. Currently, TFLM uses the CMSIS-NN library when benchmarking embedded hardware platforms [7]. In this paper, we demonstrate our kernel improves energy usage, bandwidth usage, and computation throughput over the CMSIS-NN kernel for TFLM.

3 MEMA FRAMEWORK

The MEMA runtime framework combines hardware characteristics, problem size information (e.g., input matrix sizes for MM), IO analysis, and scheduling to produce an efficient inference runtime. In this section, we introduce the MEMA framework objectives. We also describe how MEMA leverages streaming to maximize data reuse and overlap compute and IO times when partitioning MM computations with inputs that do not fit in local memory.

TinyML devices are constrained in memory sizes and speeds due to form factor and limited power budgets. The MEMA framework addresses these challenges through its objectives: reducing the number of external memory accesses and hiding IO time within compute time via streaming that efficiently utilizes available local memory. By reducing memory accesses, the framework may reduce energy consumption of machine learning inference tasks on TinyML devices. This allows MEMA to overlap IO with compute and derive runtimes that are not bottlenecked by IO.

3.1 Overview of the MEMA Framework

Given an MM $C = C + A \times B$, where A is $M \times K$ and B is $K \times N$, the MEMA framework can automatically derive optimal schedules in minimizing external memory accesses, subject to local memory size, using techniques that maximize data reuse: (1) tile shaping, (2) matrix operand streaming and (3) loop order selection.

The framework first uses information about the target hardware, such as register count, local memory size, and external memory bandwidth to determine the optimal tile sizes (Section 4.1). Tile sizes are automatically derived to maximize the arithmetic intensity of each tile multiplication (with the goal of increasing computation throughput and reducing energy consumption for IO). Based on the tile sizes, MEMA can select kernels tailored to the target platform from existing libraries or generate optimized outer product kernels. Then, by accounting for potential tile sizes and MM problem size, a schedule that minimizes external IO (Section 4.4) and decreases the required IO bandwidth is selected. The kernels and schedules are then compiled into a MEMA runtime and deployed to the TinyML device, as depicted in Figure 1.

4 MEMA ANALYSIS FOR RUNTIME DERIVATION

In this section we cover the MEMA analysis for MM on two hardware platforms: a single core MCU and a multi-core IoT device. Our analysis is limited to MM operations, but the framework may be extended to other multi-loop reduction operations with static loop bounds such as direct convolution and tensor contractions. The MEMA analysis starts by analytically deriving a tile size that maximizes arithmetic intensity for a given hardware. Then using the derived tiling, we select a loop order which maximizes data reuse between successive tile computations. We show that the external memory IO associated with streaming is determined by both tile size and MM problem size. Meanwhile, the MM problem size alone determines the IO for the stationary data. MEMA chooses the loop order with the lowest combined streaming and stationary IO.

4.1 Deriving Tile Sizes for MCUs

Consider an outer product shown in Figure 3a between a $m \times 1$ vector of A and $1 \times n$ vector of B to produce a stationary $m \times n$ tile of C. For simplicity of discussion, we only count multiplications, excluding additions. To maximize reuse, we maximize the ratio of computation (mn multiplications) to IO ($m+n$ input values) i.e., the arithmetic intensity $\frac{mn}{m+n}$. This ratio is maximized when we choose a square tile $m = n = t$. Assume we have an MCU with 36 registers of local memory available for data reuse. All outer product inputs ($m + n + mn$ values) must fit in local memory, i.e., $2t + t^2 \leq 36$. Solving for t shows the optimal tile size of C for this MCU is 5×5.

The selected C tile in this example will act as an intermediate local store to support the input streaming with computation when the scheme of Figure 3c is used. While we derive tile dimensions via a simple arithmetic intensity argument here, other tiling techniques may be used [8, 16] with MEMA.

4.2 Deriving Tile Sizes for Multicore IoT Devices

Unlike MCUs, IoT devices such as [1, 11, 24] contain multiple low-power RISCV or ARM cores as well as multiple levels of memory. Since we have to tile at multiple memory levels, the number of possible schedules is much larger. For example, on a device with 3 memory levels, we have the choice of 6 MM loop orders at each level for a total of $6^3 = 216$ possible loop orderings. In addition, tiles at each memory level must be properly sized according to the available local memory, bandwidth, and number of cores.

Instead of searching a large space of schedules for the optimal tiling as in [9, 19], we use CAKE [17], a multi-core matrix multiplication tiling and scheduling algorithm that utilizes **constant-bandwidth (CB) blocks** in computation partitioning and block scheduling. A CB block is a block of computation that, when computed from within a local memory, the required off-chip bandwidth is constant, even when utilizing additional cores. CAKE controls the arithmetic intensity of CB blocks by adjusting the CB block shape (i.e., aspect ratios) and size according to available off-chip DRAM memory bandwidth, number of cores, and available local memory. Using CAKE tiling we can increase the use of available computing power without requiring a comparable increase in off-chip memory bandwidth. CAKE does require more local memory when increasing the number of cores, but local memory size is often sufficient in comparison to off-chip bandwidth.

Suppose we want to grow the number of cores by a factor p. CAKE reshapes the computation block from $m \times k \times n$ (shown in Figure 3) to the shape $pm \times k \times pn$. CAKE also holds the large C tile, with dimensions $pm \times pn$, stationary in on-chip memory while streaming in the smaller $pm \times k$ and $k \times pn$ tiles of A and B, respectively. Let $p \cdot f$ be the peak FLOPs of the system where f is the single core peak. To compute the CB block in local memory, CAKE performs $pm \cdot k \cdot pn$ MAC operations in time $T = \frac{pm \cdot k \cdot pn}{p \cdot f}$ using $IO = pmk + pnk$ off-chip memory accesses. The required off-chip bandwidth is then:

$$BW_{off-chip} = \frac{IO}{T} = \frac{p \cdot k \cdot (m+n)}{pm \cdot k \cdot pn} \cdot p \cdot f = \frac{m+n}{mn} \cdot f$$

Note that CAKE's bandwidth usage is constant regardless of the number of cores because the p factors cancel out. Given CAKE's CB block shape, we can directly solve for the optimal tile sizes by choosing m, k, and n such that A, B, and C tiles fit in local memory (LM), i.e., $pmk + kpn + p^2mn \leq LM$. Here, LM is on-chip memory shared by all the cores, each of which computes a single $m \times k \times n$ sub-block at a time. Without loss of generality, we may continue to tile the $m \times k \times n$ sub-block according to the available registers or local memory private to each core (see Section 7).

4.3 Loop Order for Inter-Block Computations

Computing computation blocks with different loop orders may result in different total external memory accesses for the same MM. CAKE does not reorder the loops to minimize external memory accesses, instead it only uses a K-first scheduling of blocks (K-dimension as the inner loop, keeping partial results stationary). Total IO varies since loop order determines the streaming pattern. For skewed matrix shapes, a partial result stationary schedule may not minimize off-chip memory accesses. MEMA improves upon this by selecting the loop order that minimizes total external memory accesses, even if the order does not keep partial results stationary.

For a partitioned MM between an $M \times K$ matrix A and a $K \times N$ matrix B (described in Section 2.1) the total IO for a given loop order can be computed as the sum of streaming and stationary IO:

$$(\#\text{ of blocks}) \cdot (IO_{streaming} \text{ per block}) + IO_{stationary}$$

For an N-first schedule (stationary A tiles), our total IO is:

$$IO_{N\text{-first}} = \frac{MKN}{mkn}(2mn + nk) + \frac{M}{m} \cdot \frac{K}{k}(mk) = MKN\left(\frac{1}{m} + \frac{2}{k}\right) + MK$$

For an M-first schedule (stationary B tiles), our total IO is:

$$IO_{M\text{-first}} = \frac{MKN}{mkn}(mk + 2mn) + \frac{K}{k} \cdot \frac{N}{n}(kn) = MKN\left(\frac{2}{k} + \frac{1}{n}\right) + KN$$

For an K-first schedule (stationary C tiles), our total IO is:

$$IO_{K\text{-first}} = \frac{MKN}{mkn}(mk + kn) + \frac{M}{m} \cdot \frac{N}{n}(2mn) = MKN\left(\frac{1}{m} + \frac{1}{n}\right) + 2MN$$

The 2 factor from $2mn$ reflects reading *and* writing partial C tiles.

4.4 Choosing Loop Ordering Based on Tile Dimensions and Input Matrix Sizes

Using the IO equations in Section 4.3, MEMA selects a schedule which minimizes IO. Tile size determines the number of memory accesses for streaming. The size of A and B determines the memory accesses for stationary data. A loop order that minimizes external memory accesses may be selected by calculating the memory accesses for each of the 6 possible loop orders given in Section 2.1. The rest of this section gives two examples of MM loop order selection.

Consider an MM where $M = K = N$ computed with non-square tiles ($n > k = m$). The B and C tiles are larger and thus would require more IO to stream. In an N-first loop order, an A tile is kept stationary while B and C tiles are streamed. However, we can reduce IO by using an M-first or K-first order and streaming A tiles. In this example, the M-first and K-first orders are equivalent as all other dimensions are equal.

Now consider an MM between A and B with $M > N$ and $M > K$. Suppose the MM is computed using square tiles ($n = m = k$). Computing the MM using the N-first requires more total external memory accesses than the M-first schedule. In comparison, the M-first order has less IO since B tiles can be reused $\frac{M}{m}$ times, which is more than other directions because $M > N$ and $M > K$. The streaming term of the IO function is independent of loop order when tile sizes are equal.

5 MEMA DERIVATION FOR REAL HARDWARE

5.1 Evaluation Hardware Characteristics

In this and subsequent sections (Section 6 and Section 7) we evaluate MEMA's performance against state-of-the-art libraries for MM on multiple MCUs representative of popular embedded platforms: Arduino Nano 33 BLE (ARMv7E-M Cortex-M4), STM32F767ZI Nucleo (ARMv7E-M Cortex-M7), and Raspberry Pi 4 Model B (ARMv8-A Cortex-A72). MEMA is implemented using C++ and compiler intrinsics on each respective platform (open-sourced at https://github.com/vnatesh/MEMA-MM). The Cortex-M4 and M7 have very limited local memory: only 32 floating-point registers and 16 registers for DSP-accelerated fixed-point arithmetic. The Cortex-M4 has a

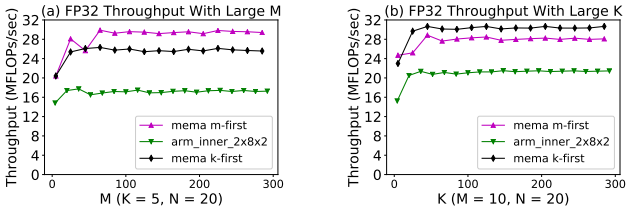


Figure 4: MEMA performance on Cortex-M4 for various MM problems. By scheduling to minimize IO, MEMA maximizes throughput for matrices with skewed shapes. (a) M-first schedule outperforms K-first when $K = 5$, as predicted by MEMA's analysis (Section 5.2). Similarly, (b) shows the K-first schedule outperforms M-first for $K > 10$. The MEMA schedule outperforms the ARM library (`arm_inner_2x8x2`).

256KB external memory (SRAM) and 1MB of flash memory for storing additional data and code. The Cortex-M7 has roughly double the SRAM and flash of the M4. Meanwhile, the Cortex-A72 has more compute and memory resources than the Cortex-M4 and M7 but is still limited by DRAM bandwidth, local memory size, and local memory bandwidth relative to computation power (see Figure 2).

5.2 MEMA Schedule Derived on Cortex-M4

In this section, we apply MEMA's analysis to an ARM Cortex-M4 in minimizing memory accesses for an MM problem, given M, K, N. After determining an optimal tile size based on the device's local memory size and hierarchy (Section 4.1), we derive conditions for when to use any of the six computation block schedules (Section 4.3). Consider, e.g., the choice between M and K-first schedules. To minimize IO, we should use the M-first schedule only when $IO_{M\text{-first}} \leq IO_{K\text{-first}}$ and $IO_{M\text{-first}} \leq IO_{N\text{-first}}$. Using the equations in Section 4.3, we obtain the following inequalities as a condition for choosing the M-first schedule:

$$MKN\left(\frac{2}{k}+\frac{1}{n}\right)+KN \leq MKN\left(\frac{1}{m}+\frac{1}{n}\right)+2MN$$

$$MKN\left(\frac{2}{k}+\frac{1}{n}\right)+KN \leq MKN\left(\frac{2}{k}+\frac{1}{m}\right)+MK$$

$$\implies K \leq \left(\frac{2M}{1+M(\frac{2}{k}-\frac{1}{m})}\right),\quad N \leq \left(\frac{M}{1+M(\frac{1}{n}-\frac{1}{m})}\right)$$

Since the Cortex-M4 has 36 registers available for local data reuse, suppose we use a square 5×5 tile for stationary data as derived in Section 4.1 i.e., $m = k = n = 5$. Then, the analysis above suggests we should use the M-first schedule when K is roughly ≤ 10 and $M \geq N$, since in this case the above two inequalities hold. We confirm this choice empirically in Figure 4a which shows the M-first schedule outperforming K-first on the Cortex-M4 when $K = 5$. Similarly, Figure 4b shows that as K increases beyond 10 with M and N fixed, the K-first schedule outperforms M-first. We may decide between N and K-first schedules via $IO_{N\text{-first}} \leq IO_{K\text{-first}}$, yielding inequalities analogous to those above.

6 NEURAL NETWORK BENCHMARKS AND ENERGY MEASUREMENT METHODOLOGY

Neural Network Benchmarks. We measure computation throughput (FLOPs/sec), external memory IO (bytes), and energy usage (joules) during MMs between weight and data matrices for the layers of neural network models from various benchmarks. On the Cortex-M4 and M7, we use the MLPerf Tiny Benchmark [7], composed of models for tasks such as keyword spotting, visual wake words, and image recognition. Using matrices from the benchmark, we extract dimensions for weight and data matrices in each layer after applying *im2col* [27]. For the ARM Cortex-A72, we use transformer model matrices provided by the Deep Learning Matrix Collection (DMLC) [23]. Packing overhead is expensive for the matrix sizes evaluated, so our kernels on the M4 and M7 are packing-free.

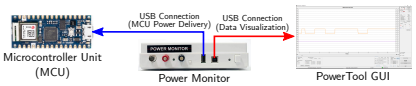


Figure 5: Power monitor setup for collecting data on a USB-connected MCU. The monitor reports real time voltage and current via the PowerTool GUI [22].

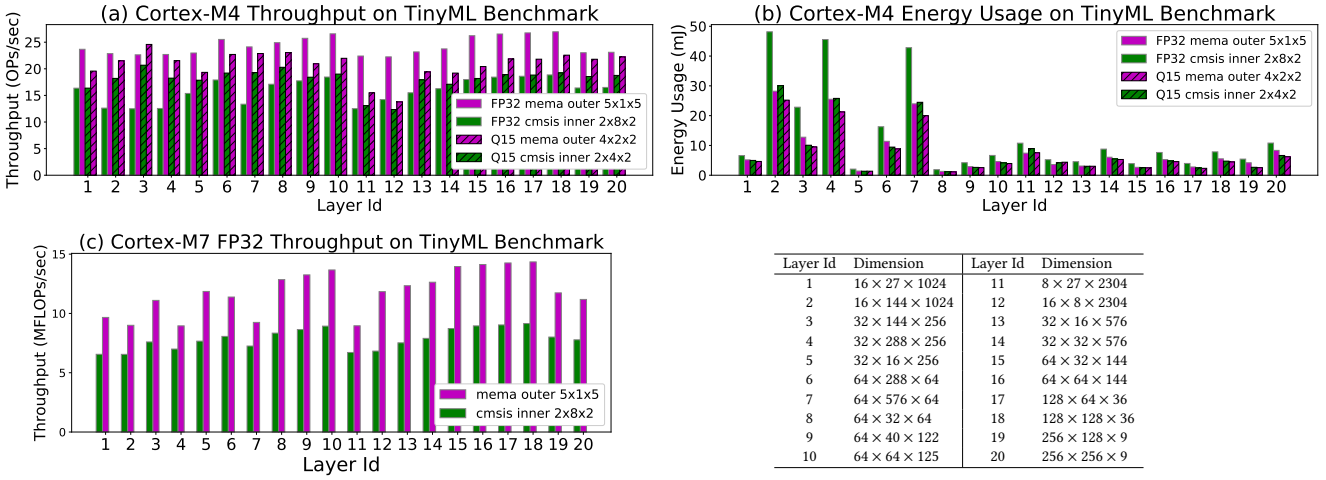
Energy Measurement Setup. To measure energy consumption, we follow the MLPerf Tiny benchmark guidelines [7]. We use a Monsoon Solutions Low-Voltage Power Monitor [22], which has a current resolution of 50 μA and voltage resolution of 125 μV. The power monitor samples voltage and current every 200 μs and reports the data via the PowerTool GUI on a host desktop. MCUs are connected to the power monitor USB channel with the USB passthrough feature enabled to allow communication with the Power Tool GUI on the host desktop (Figure 5). The MCUs operate at 5V, which is within the USB channel's voltage range of 2.1 V to 5.4 V. The MCUs consume between 16 and 22 mA of current, for a margin of error of ~2%. We report energy consumption for the Arduino Nano 33 BLE (ARMv7E-M Cortex-M4), as it is representative of the other devices. MM energy measurements are averaged over hundreds of trials.

7 RESULTS AND EVALUATION

7.1 DLMC Benchmark on ARM Cortex-A72

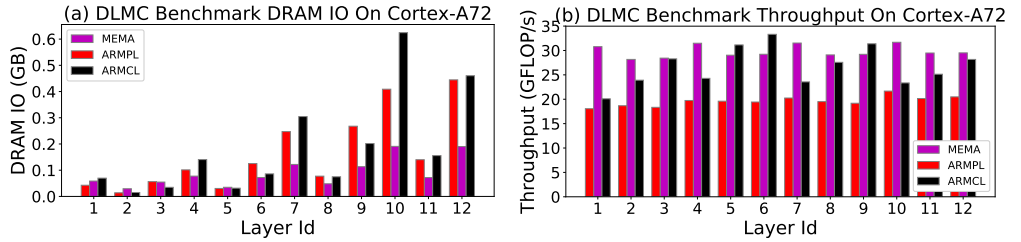
We perform MM on transformer model matrices from the DLMC Benchmark using MEMA, ARMCL, and ARMPL on the ARM Cortex-A72 CPU and collect performance data using *perf* [3]. DRAM accesses are monitored via the ARM PMU event counter (L2 cache refills from DRAM). The Cortex-A72 contains four cores ($p = 4$), a shared L2 cache, and private L1 caches on each core. We tile the MM into CB blocks as shown in Section 4.2, where each core computes a $m \times k \times n$ sub-block at a time with $m = k = n$ and $pmk + kpn + p^2mn \leq L2$. We further partition the $m \times k \times n$ sub-block into individual $8 \times k \times 12$ outer products that can be computed from the 128 32-bit registers available on each core. The $8 \times k \times 12$ shape was chosen by adapting the MCU analysis of Section 4.1 to a CPU single core.

ARMCL attains peak throughput on some MM problems (Figure 7b), but may not schedule efficiently. ARMPL and ARMCL tile MM computations according to the classical Goto's algorithm [13], which requires more DRAM bandwidth. In contrast, MEMA attains high throughput for all input shapes by minimizing IO (Figure 7a) via runtime scheduling and CAKE tiling in local memory.



Layer Id	Dimension	Layer Id	Dimension
1	$16 \times 27 \times 1024$	11	$8 \times 27 \times 2304$
2	$16 \times 144 \times 1024$	12	$16 \times 8 \times 2304$
3	$32 \times 144 \times 256$	13	$32 \times 16 \times 576$
4	$32 \times 288 \times 256$	14	$32 \times 32 \times 576$
5	$32 \times 16 \times 256$	15	$64 \times 32 \times 144$
6	$64 \times 288 \times 64$	16	$64 \times 64 \times 144$
7	$64 \times 576 \times 64$	17	$128 \times 64 \times 36$
8	$64 \times 32 \times 64$	18	$128 \times 128 \times 36$
9	$64 \times 40 \times 122$	19	$256 \times 128 \times 9$
10	$64 \times 64 \times 125$	20	$256 \times 256 \times 9$

Figure 6: Throughput and energy usage for MEMA and ARM CMSIS-NN on the ARM Cortex-M4 and M7 performing MM with matrices from the TinyML benchmark [7]. Each bar group represents the MM for a layer. Input matrix dimensions for each layer are shown in the bottom-right table. For FP32 MM, MEMA attains up to 1.8x the throughput and 44% less energy than CMSIS-NN. For Q15 arithmetic on the MCU DSPs, only 12 registers are available, limiting tiling options. For Q15 MM, MEMA achieves up to 1.2x the throughput and 20% less energy than CMSIS-DSP.



Layer Id	Dimension ($M \times K \times N$)
1	$512 \times 2048 \times 256$
2	$512 \times 512 \times 256$
3	$2048 \times 512 \times 256$
4	$512 \times 2048 \times 512$
5	$512 \times 512 \times 512$
6	$2048 \times 512 \times 512$
7	$512 \times 2048 \times 1024$
8	$512 \times 512 \times 1024$
9	$2048 \times 512 \times 1024$
10	$512 \times 2048 \times 2048$
11	$512 \times 512 \times 2048$
12	$2048 \times 512 \times 2048$

Figure 7: Throughput and external memory IO for MEMA, ARMPL, and ARMCL on the ARM Cortex-A72 when multiplying transformer model matrices from the DLMC Benchmark [23]. Input matrix dimensions for each layer are shown in the right-hand table. By minimizing external memory accesses (a), MEMA generally achieves peak computation throughput (b).

7.2 MLPerf Tiny Benchmark on ARM Cortex-M4 and M7

We compare MEMA to ARM CMSIS-DSP on the Cortex-M4 and M7 when performing MM between matrices from the MLPerf Tiny benchmark [7]. CMSIS uses inner product kernels for tiling MM computations such as GEMM-based convolutions. For instance, FP32 MM computations are tiled as inner products between 2×8 row tiles of A and 8×2 column tiles of B to yield 2×2 tiles of C (`cmsis inner 2x8x2` in Figure 6). In contrast, MEMA tiles MM with outer products between 5×1 column vectors of A and 1×5 row vectors of B to produce 5×5 tiles of C held in registers (`mema outer 5x1x5` in Figure 6). MEMA outperforms the ARM kernel by up to 1.8× while reducing energy usage by up to 44%.

For saturating fixed-point 16-bit arithmetic (Q15) on the Cortex-M4/7 DSPs, a small local memory (twelve 32-bit registers) limits kernel choices to a few shapes. Moreover, pairs of values in the input matrices are stored as 32-bit words, requiring pairs of 16-bit values to be unpacked before dual-MAC SIMD computations occur. This bit packing stalls the computation by several cycles, resulting in a roofline ridge point that is far to the left. Despite the severely compute-bound nature of Q15 computations, MEMA improves throughput over CMSIS by reducing external IO.

CMSIS Q15 MM computations are tiled as inner products between 2×4 tiles of A and 4×2 tiles of B to produce a 2×2 tile of C (`cmsis inner 2x4x2` in Figure 6). MEMA tiles the MM between 4×2 tiles of A and 2×2 tiles of B to produce 4×2 stationary tiles of C (`mema outer 4x2x2` in Figure 6). MEMA and CMSIS use the same SIMD instructions to execute multiple MAC operations per cycle, but MEMA's tiling schedule reduces the number of times B must be streamed from external memory ($\frac{M}{4}$ times for MEMA vs $\frac{M}{2}$ times for CMSIS), shown by the analysis in Section 4.4). Consequently, MEMA attains 20% higher computation throughput than CMSIS while using up to 23% less energy.

8 CONCLUSION

We propose the MEMA framework to generate inference runtimes that minimize external memory accesses for TinyML on microcontrollers (MCUs). By decreasing external memory accesses, MEMA-generated schedules can increase computation throughput and reduce energy consumption.

MEMA can adapt to various TinyML MCUs with different roofline ridge points by selecting data streaming, tile shaping, and loop ordering schemes suited to the hardware (Figure 3b and c). For example, for neural network benchmarks on the Cortex-M4, the MEMA-generated runtime schedule achieves up to a 1.8x speedup and 44% less energy than state-of-the-art CMSIS-NN library (Figure 6). This work demonstrates that simple runtime frameworks, such as MEMA, for TinyML MCUs can significantly reduce memory accesses and energy consumption.

9 ACKNOWLEDGEMENTS

This work was supported in part by the Air Force Research Laboratory under award numbers FA8750-18-1-0112 and FA8750-22-1-0500, and Meta Platforms Technologies under award number A51540.

REFERENCES

[1] [n. d.]. ARM® Cortex®-A72 MPCore Processor Technical Reference Manual. https://developer.arm.com/documentation/100095/0003/?lang=en
[2] [n. d.]. *Intel oneAPI Math Kernel Library.*
[3] [n. d.]. Perf Wiki. https://perf.wiki.kernel.org/index.php/Main_Page.
[4] 2020. CUTLASS: Fast Linear Algebra in CUDA C++. https://developer.nvidia.com/blog/cutlass-linear-algebra-cuda/.
[5] Arm Limited 2021. *Arm Performance Libraries Reference Guide.* Arm Limited. https://developer.arm.com/documentation/101004/latest/
[6] Arm Limited 2022. *Arm Compute Library Reference Guide.* Arm Limited. https://arm-software.github.io/ComputeLibrary/latest/
[7] Colby Banbury, Vijay Janapa Reddi, Peter Torelli, Jeremy Holleman, Nat Jeffries, Csaba Kiraly, Pietro Montino, David Kanter, Sebastian Ahmed, Danilo Pau, Urmish Thakker, Antonio Torrini, Peter Warden, Jay Cordaro, Giuseppe Di Guglielmo, Javier Duarte, Stephen Gibellini, Videet Parekh, Honson Tran, Nhan Tran, Niu Wenxu, and Xu Xuesong. 2021. MLPerf Tiny Benchmark. *arXiv:2106.07597 [cs]* (Aug. 2021). arXiv:2106.07597 [cs]
[8] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. 2008. Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model. In *Compiler Construction (Lecture Notes in Computer Science)*, Laurie Hendren (Ed.). Springer, Berlin, Heidelberg, 132–146. https://doi.org/10.1007/978-3-540-78791-4_9
[9] Alessio Burrello, Angelo Garofalo, Nazareno Bruschi, Giuseppe Tagliavini, Davide Rossi, and Francesco Conti. 2021. DORY: Automatic End-to-End Deployment of Real-World DNNs on Low-Cost IoT MCUs. *IEEE Trans. Comput.* 70, 8 (2021), 1253–1268. https://doi.org/10.1109/TC.2021.3066883
[10] Robert David, Jared Duke, Advait Jain, Vijay Janapa Reddi, Nat Jeffries, Jian Li, Nick Kreeger, Ian Nappier, Meghna Natraj, Tiezhen Wang, Pete Warden, and Rocky Rhodes. 2021. TensorFlow Lite Micro: Embedded Machine Learning for TinyML Systems. *Proceedings of Machine Learning and Systems* 3 (March 2021), 800–811.
[11] Eric Flamand, Davide Rossi, Francesco Conti, Igor Loi, Antonio Pullini, Florent Rotenberg, and Luca Benini. 2018. GAP-8: A RISC-V SoC for AI at the Edge of the IoT. In *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 1–4. https://doi.org/10.1109/ASAP.2018.8445101
[12] Jorge Gomez, Saavan Patel, Syed Shakib Sarwar, Ziyun Li, Raffaele Capoccia, Zhao Wang, Reid Pinkham, Andrew Berkovich, Tsung-Hsun Tsai, Barbara De Salvo, et al. 2022. Distributed On-Sensor Compute System for AR/VR Devices: A Semi-Analytical Simulation Framework for Power Estimation. *arXiv preprint arXiv:2203.07474* (2022).
[13] Kazushige Goto and Robert A. van de Geijn. 2008. Anatomy of High-Performance Matrix Multiplication. *ACM Trans. Math. Softw.*, Article 12 (2008), 25 pages.
[14] Kazushige Goto and Robert A. van de Geijn. 2008. Anatomy of High-Performance Matrix Multiplication. *ACM Trans. Math. Software* 34, 3 (May 2008), 12:1–12:25. https://doi.org/10.1145/1356052.1356053
[15] Mark Hill and Vijay Janapa Reddi. 2019. Gables: A Roofline Model for Mobile SoCs. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 317–330. https://doi.org/10.1109/HPCA.2019.00047
[16] H. T. Kung, Vikas Natesh, and Andrew Sabot. 2021. CAKE: Matrix Multiplication Using Constant-Bandwidth Blocks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/3458817.3476166
[17] H. T. Kung, Vikas Natesh, and Andrew Sabot. 2021. CAKE: Matrix Multiplication Using Constant-Bandwidth Blocks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 85, 14 pages. https://doi.org/10.1145/3458817.3476166
[18] Liangzhen Lai, Naveen Suda, and Vikas Chandra. 2018. CMSIS-NN: Efficient Neural Network Kernels for Arm Cortex-M CPUs. *arXiv:1801.06601 [cs]* (Jan. 2018). arXiv:1801.06601 [cs]
[19] Rui Li, Aravind Sukumaran-Rajam, Richard Veras, Tze Meng Low, Fabrice Rastello, Atanas Rountev, and P. Sadayappan. 2019. Analytical Cache Modeling and Tilesize Optimization for Tensor Contractions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 74, 13 pages. https://doi.org/10.1145/3295500.3356218
[20] Ji Lin, Wei-Ming Chen, Yujun Lin, John Cohn, Chuang Gan, and Song Han. 2020. MCUNet: Tiny Deep Learning on IoT Devices. In *Proceedings of the 34th International Conference on Neural Information Processing Systems (NIPS'20)*. Curran Associates Inc., Red Hook, NY, USA, 11711–11722.
[21] Bradley McDanel, Sai Qian Zhang, HT Kung, and Xin Dong. 2019. Full-stack optimization for accelerating CNNs using powers-of-two weights with FPGA validation. In *Proceedings of the ACM International Conference on Supercomputing*. 449–460.
[22] Monsoon Solutions, Inc. [n. d.]. Low Voltage Power Monitor Documentation. https://www.msoon.com/lvpm-product-documentation.
[23] Google Research. 2020. Deep Learning Matrix Collection. https://github.com/google-research/google-research/tree/master/sgk.
[24] Davide Rossi, Francesco Conti, Andrea Marongiu, Antonio Pullini, Igor Loi, Michael Gautschi, Giuseppe Tagliavini, Alessandro Capotondi, Philippe Flatresse, and Luca Benini. 2015. PULP: A parallel ultra low power platform for next generation IoT applications. In *2015 IEEE Hot Chips 27 Symposium (HCS)*. IEEE Computer Society, 1–39.
[25] Tyler M. Smith and Robert A. van de Geijn. 2019. The MOMMS Family of Matrix Multiplication Algorithms. *arXiv:1904.05717 [cs]* (April 2019). arXiv:1904.05717 [cs]
[26] Stanislava Soro. 2021. TinyML for Ubiquitous Edge AI. https://doi.org/10.48550/ARXIV.2102.01255
[27] Aravind Vasudevan, Andrew Anderson, and David Gregg. 2017. Parallel Multi Channel convolution using General Matrix Multiplication. In *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 19–24. https://doi.org/10.1109/ASAP.2017.7995254
[28] Pete Warden. 2015. Why GEMM Is at the Heart of Deep Learning.
[29] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM* 52, 4 (2009), 65–76. https://doi.org/10.1145/1498765.1498785
[30] Zhang Xianyi, Wang Qian, and Zaheer Chothia. 2012. Openblas. *URL: http://xianyi.github.io/OpenBLAS* (2012), 88.
[31] Vinson Young, Sanjay Kariyappa, and Moinuddin K. Qureshi. 2019. Enabling Transparent Memory-Compression for Commodity Memory Systems. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 570–581. https://doi.org/10.1109/HPCA.2019.00010