TMModel: Modeling Texture Memory and Mobile GPU Performance to Accelerate DNN Computations

Jiexiong Guan[*]
University of Thessaly
Volos, Greece
William & Mary
Williamsburg, VA, USA
jguan@uth.gr

Zhenqing Hu
William & Mary
Williamsburg, VA, USA
zhu05@wm.edu

Christos D. Antonopoulos
University of Thessaly
Volos, Greece
cda@uth.gr

Nikolaos Bellas
University of Thessaly
Volos, Greece
nbellas@uth.gr

Spyros Lalis
University of Thessaly
Volos, Greece
lalis@uth.gr

Evgenia Smirni
William & Mary
Williamsburg, VA, USA
esmirni@cs.wm.edu

Gang Zhou
William & Mary
Williamsburg, VA, USA
gzhou@wm.edu

Gagan Agrawal
University of Georgia
Athens, GA, USA
gagrawal@uga.edu

Bin Ren
William & Mary
Williamsburg, VA, USA
bren@wm.edu

Abstract

The demand for Deep Neural Network (DNN) execution (including both inference and training) on mobile system-on-a-chip (SoCs) has surged, driven by factors like the need for real-time latency, privacy, and reducing vendors' costs. Mainstream mobile GPUs (e.g., Qualcomm Adreno GPUs) usually have a 2.5D L1 texture cache that offers throughput superior to that of on-chip memory. However, to date, there is limited understanding of the performance features of such a 2.5D cache, which limits the optimization potential. This paper introduces `TMModel`, a framework with three components: 1) a set of micro-benchmarks and a novel performance assessment methodology to characterize a non-well-documented architecture with 2D memory, 2) a complete analytical performance model configurable for different data access pattern(s), tiling size(s), and other GPU execution parameters for a given operator (and associated size and shape), and 3) a compilation framework incorporating this model and generating optimized code with low overhead. `TMModel` is validated both on a set of DNN kernels and for training complete models on mobile GPU, and compared against both popular mobile DNN frameworks and another GPU performance model. Evaluation results demonstrate that `TMModel` outperforms all baselines, achieving $1.48 - 3.61\times$ speedup on individual kernels and $1.83 - 66.1\times$ speedup for end-to-end on-device training with only $0.25\% - 18.5\%$ the tuning cost of the baselines.

CCS Concepts

• **Computing methodologies** → **Neural networks**; **Parallel computing methodologies**; • **General and reference** → **Performance**; • **Human-centered computing** → *Ubiquitous and mobile computing systems and tools*; • **Software and its engineering** → *Source code generation*.

Keywords

Texture Memory, Performance Modeling, Architecture Profiling, Automatic Code Generation, On-device Training

ACM Reference Format:
Jiexiong Guan, Zhenqing Hu, Christos D. Antonopoulos, Nikolaos Bellas, Spyros Lalis, Evgenia Smirni, Gang Zhou, Gagan Agrawal, and Bin Ren. 2025. TMModel: Modeling Texture Memory and Mobile GPU Performance to Accelerate DNN Computations. In *2025 International Conference on Supercomputing (ICS '25), June 08–11, 2025, Salt Lake City, UT, USA*. ACM, New York, NY, USA, 16 pages.
https://doi.org/10.1145/3721145.3725774

[*]Ph.D. candidate at William & Mary. This work was completed while the author was a researcher at University of Thessaly.

1 Introduction

Mobile system-on-a-chip (SoC) has become an indispensable platform for executing powerful compute- and memory-intensive applications. Prominent among these are deep learning (DL) neural network execution tasks [10, 30] that support applications related to speech, image, and text recognition, or even Generative AI. Execution on a mobile device itself, as compared to the mobile device serving as a front-end to cloud-based servers, has several advantages like service continuity or lower latency of applications with no or limited internet availability, and better privacy, among others. However, execution on mobile devices is challenging given the constantly increasing size of DL models and the limited resources of mobile devices.

Performance models, especially those related to memory/-cache, have long played a crucial role in guiding performance tuning of applications [8, 24, 69, 77, 78, 81]. Modeling on-chip cache behaviors has been extensively explored in both sequential [14, 25] and parallel processors [3, 4, 12, 32, 33, 50, 53, 62, 63, 66]. A prevalent approach involves extracting memory traces and calculating the *data reuse distance* metric. In developing performance models for mobile processors, there are significant new considerations. For example, the latest Adreno and Mali GPUs are equipped with specialized L1 texture cache, offering notable performance advantages over other on-chip memory types (e.g., local and constant memory [39]). The existing performance modeling approaches, including the notion of data reuse distance, do not directly apply to the texture cache, which is designed for two-dimensional spatial locality. An additional complication is that the texture representation (i.e., the storage mapping in memory) is patented and obscure to developers.

Based on this motivation, this paper proposes `TMModel`, the first performance model and associated compiler designed to address the following question: *"how to analytically choose between different implementations of a single NN operator for a given input and output shape – specifically, making selections considering data access pattern(s), tile sizes, and other GPU execution parameters – to obtain the best performance"*. This performance model needs to capture (i) the spatial locality of a single thread with texture memory, (ii) the cross-thread spatial and temporal locality, and (iii) the interplay with other GPU factors like occupancy.

The design and implementation of `TMModel` comprises several steps. First, we develop a set of micro-benchmarks to demystify the black-box micro-architecture and introduce novel approaches to characterize performance. One of the key ideas we introduce is *cross-block stride*, designed to capture spatial locality in 2D memory. The insights from micro-benchmarks lead to a performance model, integrating the factors listed above into one unified cost expression. Furthermore, the performance model is incorporated into a compiler that can generate efficient code efficiently (i.e., without high performance tuning costs that many other systems require).

Overall, this paper makes the following contributions:

- We introduce a series of micro-benchmarks to understand several aspects of the target architecture – including the impact of spatial and temporal locality of the texture memory; the work-group size, warp shape, and the occupancy, on the performance. The methodology and the set of micro-benchmarks is designed to be applicable to other architectures as well.
- We propose an automatic way to predict the latency for a given data access pattern on 2D texture memory space.
- We introduce a full performance model, capturing the impacts of memory and occupancy across threads in a warp, warps in a work group, and the full kernel.
- We develop a complete system prototype for code optimization for both kernels and full DL model training on a mobile architecture.

`TMModel` is extensively evaluated on both individual kernels and end-to-end on-device DNN training by comparing with four state-of-the-art mainstream product-level mobile DNN frameworks and their associated performance modeling/tuning methods (`MNN` [30], `CLML` [1], `TFLite` [2], and `TVM` [9]) and a more general-purpose GPU performance modeling method (PPT-GPU [3]). The evaluation results demonstrate that `TMModel` achieves an average speedup of $1.48 - 3.61\times$ on individual kernels and a speedup of $1.83 - 66.1\times$ for end-to-end DNN on-device training. `TMModel` achieves similar performance to the brute-force search (with an average variation of 1%), with only 0.25% - 18.5% tuning cost vs the baseline frameworks. `TMModel` is also characterized by high prediction accuracy and good portability.

2 Background

This section provides an overview of mobile GPU architectures, starting with details of the OpenCL programming model, which is a popular way to program these GPUs. To provide additional context for the entire paper, this section also compares OpenCL with CUDA terminology.

OpenCL Programming Model. OpenCL, like CUDA, employs the SIMT (Single Instruction Multiple Threads) execution model, where a kernel specifies the computation logic for one thread. The data index space associated with this execution is divided into *work groups* (i.e., *thread blocks* in CUDA) based on the specified *work-group size*. Each work group is assigned to a *shader core* (equivalent to a *streaming multiprocessor* in CUDA). A work group consists of multiple *work items*, which are software *threads* mapped to hardware threads for execution on ALUs within the shader core. The fundamental execution unit in a shader core is the *warp*, a group of hardware threads that execute the same instruction simultaneously. Consequently, the work items within a work group are partitioned into warps for execution. The launch order of warps or work groups is not defined, so programs should not assume any specific execution sequence across them [55].

Mobile GPU Architecture and 2D Texture Cache. Fig. 1 shows a high-level overview [1] of the mainstream Adreno mobile GPU architecture by Qualcomm [55]. Different generations of Adreno GPUs or other mobile GPUs (e.g. Mali series) may have different implementations with respect to parameters such as the number of shader cores (#SP) or the capacity of different caches, and other factors [5, 39, 55].

2D texture memory. System memory supports two types of data storage: *buffer* and *image*. The former stores data in 1-dimension continuously as in CPU memory and reads and writes data via L2 cache. The latter allows data storage as images in 1-/2-/3-dimensional entities called *image objects*, with each data element (called *pixel*) capable of storing four channels (RGBA). Usually, mobile GPUs accelerate data reads of *image objects* by a special two-dimensional or 2D L1 cache hardware called *texture cache*[2]. Although the read-only L1 texture cache is small (usually 1KB) and shared among all threads on a shader core, it offers much higher reading throughput than other memory (e.g., 2× over L2 and 7× over main memory [39, 55]). Exploiting 2D data locality turns out to be critical for performance optimization on mobile GPUs.

Differences between Mobile and Desktop GPUs. Mobile and desktop GPUs differ significantly in their memory hierarchy and memory access speed. On desktop GPUs, the access latency of shared memory (or L1 cache) is 1.47× to 6.87× faster than that of the texture cache – as observed across multiple generations of GPU microarchitectures [18, 35, 41, 44, 51, 70]. In contrast, on mobile GPUs, the texture cache offers twice the throughput of shared memory [39].

On Nvidia GPUs, achieving high memory bandwidth primarily depends on coalescing accesses and minimizing bank conflicts from shared memory. In contrast, mobile GPUs benefit more from vectorized memory access and coalesced accesses are not effective [55]. To further explain the difference, coalesced accesses involve threads in a warp collectively accessing contiguous, aligned addresses, whereas vectorized access allows a single thread to load/store multiple consecutive elements in one operation [55, 71]. Moreover, efficiently leveraging the on-chip texture L1 cache, with its limited 1KB capacity, is crucial for high performance.[3]

[1]Graphics-only hardware modules are omitted because this paper focuses on general-purpose computing on mobile GPUs.

[2]Also referred to as 2.5D as each point in the 2D space is a vector.

[3]In comparison, shared memory/L1 size on latest server GPUs is 256 KB [52].

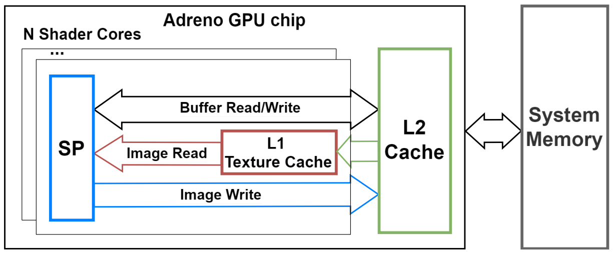

Figure 1: High-level overview of Adreno GPUs.

3 Demystify a System with 2D Cache

Understanding texture cache in-depth is necessary in order to build an analytical performance model for applications on a system using such a cache. However, this is challenging because of two key design features (*texture representation* and *cache organization*) of mobile texture cache [15, 22, 26] are patented and obscure to end-users [55]. Texture representation refers to the two-dimensional texture objects' data storage in memory. This further includes the *data unit* (e.g., 2D data block), their *organization* (e.g., hierarchical blocked storage), and the *data storage order* in memory within and among the data units (e.g., row, column, zigzag, or Hilbert order). Cache organization defines the data read between the 2D texture cache and the main memory including data mapping, cache line replacement/eviction, and other features. Previous methods [44, 70] focused solely on using 1D image data (`Image1D`) to analyze the texture cache organization, leaving the analysis of texture representation, 2D data locality, and their impact on performance unclear. With `Image2D` outperforming `Image1D` by a factor of 2 in terms of throughput, it is clearly important to focus on the former. Our work introduces a general empirical approach for studying the relationship between data access patterns and the resulting memory performance. This is accomplished in two steps: considering texture representation and 2D data locality for a single thread (Section 3.1), and extending this study by considering GPU's SIMT parallel programming (Section 3.2).

3.1 Single-Thread Performance with 2D Cache

This empirical study leverages a set of micro-benchmarks involving random accesses to unveil the relationship between data access patterns and 2D texture object memory latency for a single thread. As noted above, the goal is to achieve this without necessarily knowing the concrete texture representation and 2D cache organization. The outcome of this study is a *machine learning regression model* that can predict the expected memory latency for any given data access pattern. Figure 2 illustrates the overview of this study which consists of four main steps outlined below. Constrained by

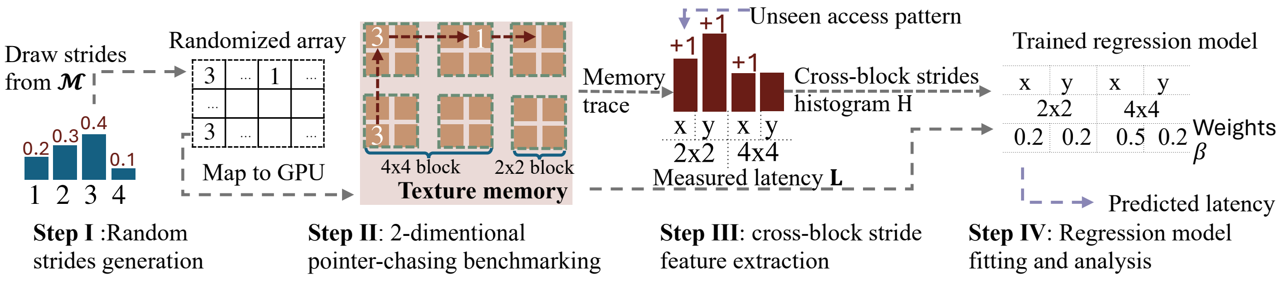
Draw strides from $\mathcal{M}$
Randomized array
Map to GPU
4x4 block
2x2 block
Texture memory
Memory trace
Unseen access pattern
Cross-block strides histogram H
Measured latency L
Trained regression model
Weights β
Predicted latency
Step I :Random strides generation
Step II: 2-dimentional pointer-chasing benchmarking
Step III: cross-block stride feature extraction
Step IV: Regression model fitting and analysis

Figure 2: Overview of the empirical study for single-thread data access pattern and 2D texture cache performance. **Step I** generates a random stride list of (1, 2, 3, 4) with a possibility of (0.2, 0.3, 0.4, 0.1) in a multinomial distribution. **Step II** conducts a 2D pointer-chasing benchmarking using random strides and generates a memory trace. **Step III** extracts cross-block strides and creates a histogram as a regression model feature. This example assumes 2 block shapes, 2×2 and 4×4, counts cross-block strides along both x- and y-directions for both cases, and creates a cross-block strides histogram w/ four bars. **Step IV** runs multiple repetitions of the first three steps and trains a regression model w/ multiple cross-block stride histograms and the corresponding measured latency, which can predict the access latency for arbitrary data access patterns later.

space, please find a complete algorithm from our anonymous Supplementary Material (Sec. I) online [4].

Step I: *Offline multinomial distribution-based random strides generation.* Inspired by pointer-chasing benchmarking [65], this step generates random access indices offline, thus eliminating the overhead of online index generation and memory trace collection, and mitigating their impact on benchmarking accuracy. Unlike existing efforts [39, 44, 65, 70] that generate 1D uniform strides, `TMModel` is novel in generating 2D random strides (along horizontal and vertical directions) to study the 2D texture object access. This step takes a list of possible strides (S) as input and constructs a multinomial distribution $\mathcal{M}$ for S. Each benchmark execution assumes a different multinomial distribution $\mathcal{M}$ to ensure randomness.

Step II: *On-device 2D Random-strided pointer-chasing benchmarking.* After generating random strides, `TMModel` employs a newly designed micro-benchmark kernel to measure memory latency values for a set of random strides. This micro-benchmark kernel runs in a pointer-chasing style, i.e., fetches a pixel and uses its value as 2D strides for subsequent access.

Step III: *Cross-block stride feature extraction.* In a 2D texture cache, data is typically organized in 2D blocks enabling data locality along both the width and height dimensions [15]. Due to this blocked data organization, data elements that look continuous from the logical (image) view may be stored far from each other in the physical memory if they belong to different data blocks. We can assume that accessing elements across blocks results in longer latency than accessing data within each block. Based on this assumption, we can build a machine-learning model to predict the relationship between data access patterns and access latency. We introduce the term **`cross-block stride`** for this machine learning model. It is defined as a function of the shape and size of the data block, and is *the stride that goes across distinct data blocks (under the assumed shape and size of the block).* Our benchmarking collects the cross-block strides for various assumed data block shapes/sizes and the execution latency for each run. This information is used as the input feature of our subsequent machine-learning model.

Step IV: *ML regression model fitting and analysis.* Steps I-III are repeated multiple times to construct the training data (H, L), where H is the histogram of cross-block strides for assumed data block shapes/sizes and L is the profiled latency for each run. The collected training data are fed into an ML regression model based on the least squares method. For a given computation kernel and its associated data access pattern, we can first calculate the histogram of cross-stride accesses, and then use the model to predict the access latency.

3.2 Extension to Parallel Execution

As stated previously, multiple threads share the 2D cache. This section extends the previous benchmarking to a set of threads at both the warp and work-group levels.

3.2.1 Intra-warp spatial locality. This section studies the impact of threads within each warp on spatial data locality. *Empirical study settings.* This experiment uses a single warp with varying numbers of threads – ranging from 1 to the maximum number supported (e.g., 64 on the mobile Adreno GPU). Each thread reads different data sections in a streaming manner. The entire data footprint is larger than the last level (L2) cache to rule out the impact of temporal data locality. This experiment is conducted on 4 kernels with representative data access patterns (as shown in Fig. 3). From the cache line reuse perspective, these kernels represent 4 cases with increasing reuse distances. Specifically, they are: ❶ `Column` major coalesced access: each thread reads data in column-major fashion and the threads collectively consume

[4] https://drive.google.com/file/d/1UYV8sxYRVa1EhSad3qp6U8Od-YWb9kiq

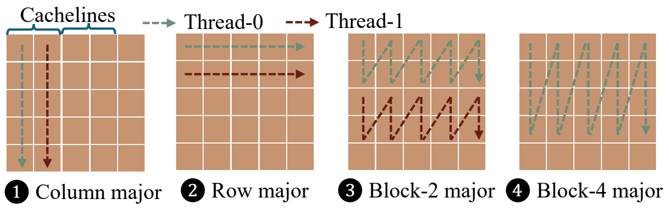

Figure 3: 4 different access patterns: with increasing cache line reuse distance from 0 to 4. In this example, each yellow block is a pixel, and two horizontal pixels occupy a cache line. Each thread consumes one pixel at a time.

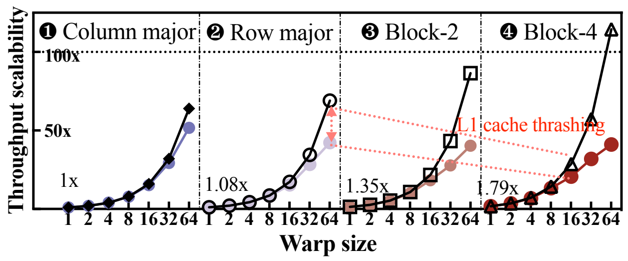

Figure 4: Scalability of each access pattern: restricted by reuse-distance. Black Line: perfect scalability for comparison. All results are normalized by 1 thread of `Column` major. For 1 thread, `Row` major, `Block-2`, and `Block-4` outperform `Column` major by 1.08×, 1.35×, and 1.79×, respectively.

a full cache line immediately (cache line reuse distance of each thread is treated as 0), ❷ `Row` major: each thread reads each row or each cache line sequentially (cache line reuse distance being 1), ❸ `Block-2` major: each thread reads data in a 2-row block manner (cache line reuse distance being 2), and ❹ `Block-4` major: each thread reads data in a 4-row block manner (cache line reuse distance being 4). Each kernel runs 10 times and average memory throughput is measured.

Results Analysis. Figure 4 shows the benchmarking results, where the x-axis corresponds to the number of threads in the warp, and the y-axis shows the measured memory throughput compared with *perfect scalability* (shown by black lines, calculated by $1\_thread\_throughput$×*#threads*). These results unveil two key findings: 1) With respect to single-thread throughput, `Block` access patterns (e.g., ❸ and ❹) achieve better performance because they are more consistent with the 2D texture memory block representation, i.e., result in fewer cross-block strides. 2) Regarding scalability, the performance of `Block` access patterns (e.g., ❸ and ❹) deteriorates quicker as the thread count increases. This is because `Block` results in larger cache line reuse distances, requiring more simultaneously active cache lines within a specific time window. It increases memory bandwidth utilization with a small number of threads as there are more concurrent memory requests. However, this leads to severe cache thrashing and contention when the number of threads increases.

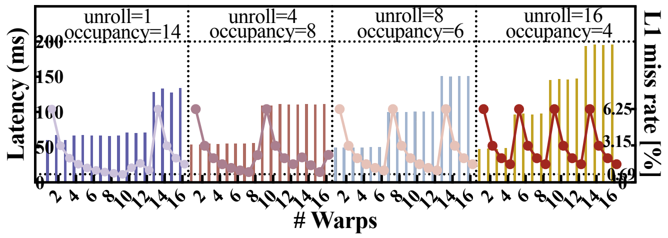

Figure 5: Intra-work-group temporal locality: Lines represent L1 miss rate using the right y-axis; Bars represent execution latency using the left y-axis.

We can observe that `Block` access patterns (e.g., ❸ and ❹) are friendly for low warp size scenarios; data access patterns with smaller cache line reuse distances (e.g., ❶ and ❷) have the potential to achieve better performance as the thread count grows. A likely consequence is that for larger input shapes – where it is more likely that more threads are deployed – performance will be better with data access patterns that scale to a larger number of threads (e.g., ❶ and ❷). Conversely, with smaller input shapes, where the code is likely to use few threads, an access pattern with higher single-thread performance (e.g., ❸ and ❹) is preferred.

Next, transitioning to a coarser-grained perspective, we study temporal data locality among threads (at intra-warp, inter-warp/intra-work-group, and inter-work-group levels). Due to limited space, we only describe experiments related to intra-work-group studies. Others are captured in our aforementioned online Supplementary Material (Sec. II).

3.2.2 *Intra-work-group temporal locality and occupancy.*

This study aims to confirm that temporal data locality among warps within each work group can also help improve performance. Additionally, by analyzing execution latency results, this study identifies critical factors for mobile GPU occupancy modeling with varying work-group sizes.

Empirical study settings. This experiment uses a single work group, varying its size from 1 warp (64 threads) to 16 warps (1024 threads). All threads read the same data in a row-major pattern, exceeding the L2 cache capacity in all cases. An expensive function (e.g., log) is applied to the data read, with `loop unrolling` used to vary register pressure. Increased register usage reduces the number of concurrently executed warps, known as `occupancy` [64]. Unroll factors from 1 to 16 are tested, with occupancy ranging from 14 to 4. This experiment was repeated with other data access patterns, but each yielded similar conclusions.

Results Analysis. Fig. 5 shows the experimental results, where x-axis shows the work-group size in warps with all 4 different unrolling factors, the right y-axis shows the L1 texture cache miss rate, and the left y-axis shows the execution latency. For each unrolling case, the L1 miss rate decreases linearly when the work-group size is smaller than the occupancy, as all threads read the same data. When the work-group size exceeds the occupancy limit, the L1 miss rate exhibits an intriguing pattern: it initially increases, then drops to its lowest point, forming a recurring wave-like curve. We define *occupancy group* as a set of active warps that collectively fit within the available hardware resources (e.g., registers, and execution units) of a GPU's streaming multiprocessor (SM). This *chronic wave* behavior is observed for occupancy group. This indicates that warps within the same work group but in different occupancy groups cannot share data in the L1 cache. Additionally, even though the L1 cache miss rates are the same for different numbers of work groups, the latency values increase linearly with occupancy, resulting in a stepped latency curve. This leads to a conclusion that *warps in the same occupancy group execute concurrently, while those in different groups operate serially.*

Moreover, comparing execution latency among all unrolling cases, particularly between *unroll* = 1 and *unroll* = 16, with identical numbers of thread warps, reveals a trade-off between single-warp performance and occupancy. Both factors together influence overall performance. While unrolling uses more registers and improves performance per thread, it can reduce overall kernel performance by decreasing occupancy and increasing context switching overhead as the work-group size increases.

We now summarize the various observations. First, intra-work-group temporal locality exists, offering more opportunities for using `Block` access patterns (e.g., ❸ and ❹). Second, kernels with different input/output shapes/sizes may prefer different sizes of work groups due to occupancy considerations. For smaller data sizes, kernels favor smaller work groups with unrolling for improved single-thread performance. For larger ones, kernels prefer larger work groups without unrolling to maximize occupancy.

4 Performance model

Based on the benchmarking studies of the mobile GPU architecture introduced above, we developed an analytical model for performance prediction. The goal of this model is to enable (memory-related) performance optimizations and/or selection of better versions of code.

Fig. 6 summarizes the basic idea of this model and Table 1 summarizes all its parameters. The static information input to the model, which is independent of the program being executed, includes cache capacity (number of lines **C**) and the number of streaming processors (**SPs**). Parameters specific to the kernel but independent of the implementation include operator types (**OP**) and output shapes (**O**). The parameters that can vary with implementation include the data access pattern ($\mathcal{A}$), the work-group size $\mathcal{G} = b_x \times b_y \times b_z$ (x, y, z are three dimensions of the work group), and the thread-level tiling size

Table 1: `TMModel` **parameters.** Var.: variable, Con.: constant.

Var.	$\mathcal{A}$, **H**, s	Access pattern, cross-block strides histogram, reuse-distance
	$\mathcal{G}$, $\mathcal{W}$	Work-group size, warp shape
	$\mathcal{T}$	Tiling/unrolling
Con.	**OP**, **O**	Operator type, output shape
	SPs, **R**	#Shader cores, #Registers per core
	C, β	#Cache lines, LR modeled texture represent

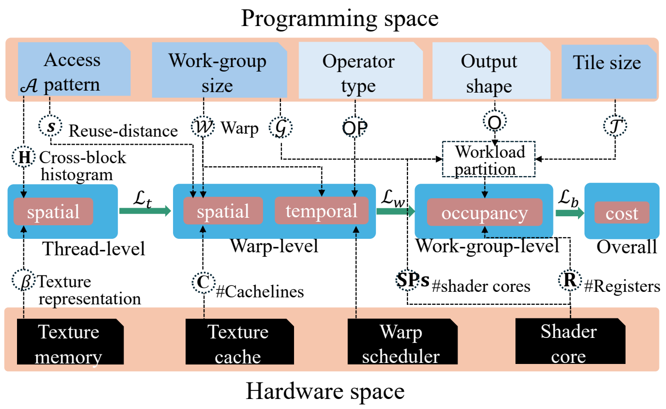

Figure 6: Performance model overview. The model is built from left to right by taking corresponding parameters.

($\mathcal{T}$) which is equivalent to the `unrolling factors` mentioned in Section 3.2 [5]. Specifically, each warp ($\mathcal{W}$) contains a fixed number of threads (64 on Adreno GPUs), with associated work-group size and shape $\mathcal{G}$. This performance model enumerates all possible combinations of implementation-dependent factors, calculates the execution cost (`cost`) for each of them, and chooses the combination with the minimum cost. As we will describe in the next section, this performance model, though broader in its applicability, has already been integrated with a compiler that generates code parameterized with these parameters. Thus, by choosing parameters, the model enables optimized code for each kernel.

More specifically, the performance model consists of four components: 1) at the *thread level*, it models the texture memory latency for varied access patterns – it utilizes the results of the `cross-block stride` study from Section 3.1; 2) at the *warp level*, it models how texture memory latency changes for different access patterns as threads count grows – this component utilizes the `scalability` study from Section 3.2.1; 3) at the *work-group level*, it models the impact of occupancy and thread workload partitioning on parallelism within each work group and across work groups based on the study from Section 3.2.2; and 4) the *final level*, where the data access latency (first 2 components) and parallelism model (third level) are integrated to predict the execution cost of a given kernel. Please note: computation-related aspects (e.g.,

[5]Our implementations empirically unroll all the tiling loops, hence tiling size equals the unrolling size.

arithmetic intensity) are not considered in this model since our goal is to choose between different implementations of one operation – these implementations are expected to be identical with respect to these parameters.

Thread-level. At the thread-level, the cross-block stride information obtained earlier is summarized as a histogram $\mathbf{H}$ capturing the frequency of strides across various block sizes. Our performance model takes as input the histogram and applies the fitted linear regression model introduced in Section 3.1 to predict texture memory access latency $\mathcal{L}_t(\mathbf{H})$. Specifically, the machine learning model described in the previous section results in a vector β of the same length, and each value is a weight that is to be multiplied by the corresponding element in the access pattern vector $\mathbf{H}$ (i.e. a dot product is performed). Thus, the predicted thread-level latency is given by:

$$\mathcal{L}_t(\mathbf{H}) = \langle \beta, \mathbf{H} \rangle. \tag{1}$$

Warp-level. The warp-level step maps the thread-level data access latency $\mathcal{L}_t$ computed above to the warp-level data access latency $\mathcal{L}_w$. This calculation should consider different *warp shapes*, marked as $\mathcal{W} = (w_x, w_y)$, and *warp size*, marked as $|\mathcal{W}| = \min(w_x \times w_y, 64)$. The w_x and w_y are configurable and subject to the work group size [61]. A detailed description of warp shape and thread mapping is in our online Supplementary Material (Sec. III). If there is no resource contention, $\mathcal{L}_w = \mathcal{L}_t$. However, as discussed in Section 3.2.1, the warp-level performance is restricted by two key factors: the texture cache capacity (i.e., the number of cache lines, denoted as C), and intra-thread reuse distance (s). The impact of the latter is dependent on the data access pattern (as shown in Fig. 4). When $s \times |\mathcal{W}| > C$, cache thrashing/contention occurs, thus the memory latency will increase. The effect of cache thrashing on increased latency is denoted by D. Then, the predicted texture memory access latency is:

$$\mathcal{L}_w(\mathcal{W}, \mathbf{H}, s) = \mathcal{L}_t(\mathbf{H}) \times D^{\lceil \frac{(|\mathcal{W}| \times s - C)}{C} \rceil}. \tag{2}$$

Intuitively, if $|\mathcal{W}| \times s \leq C$, $\lceil \frac{(|\mathcal{W}| \times s - C)}{C} \rceil$ equals to zero and cache thrashing decay rate is set to one (i.e. there is no cache thrashing). However, when the value is higher, the exponential term captures the slowdown observed. In further explaining this, if we take log on both sides, we get $\log \mathcal{L}_w - \log \mathcal{L}_t = \lceil \frac{(|\mathcal{W}| \times s - C)}{C} \rceil \log D$. The expression is derived by collecting data access latency values with different warp shape and applying the least squares method to solve for $\log D$. The value of D remains constant for each mobile GPU and does not vary per application, ensuring stability across different workloads.

For operators with a single input (e.g., `SoftMax`) the above formula is sufficient to calculate the warp-level texture memory access latency[6]. For operators with more than one input (e.g., `MatMul`), the memory access latency for each input tensor needs to be accumulated, as the contention for cache may be across different input tensors. Assuming we have i inputs, the general formula is:

$$\mathcal{L}_w(\mathcal{W}, \mathbf{H}, s) = \sum_i \mathcal{L}_t(\mathbf{H}_i) \times D^{\lceil \frac{(\sum_i \mathcal{W}_i \times s_i - C)}{C} \rceil}. \tag{3}$$

Work-group-level. As discussed, the work-group-level component of our cost model considers two key factors affecting execution cost: *occupancy* and *workload partitioning*.

Occupancy refers to the maximum number of warps that can be concurrently executed by each stream processor [64], as also discussed in Section 3.2.2. For example, each work-group on an Adreno GPU logically consists of up to 1024 threads, which is equivalent to 16 warps. Each work-group is exclusively assigned to one shader core, where registers are allocated for each thread for private usage. Hence, occupancy may vary depending on the register usage of each thread and overall register resources. Exceeding the occupancy over what can ensure sufficient register resources results in serial execution. The occupancy can be calculated as:

$$\text{Occupancy} = \mathbf{R}/\text{Register Footprint } (\mathcal{T}), \tag{4}$$

where Register Footprint depends on the computation kernel and thread-level tiling size ($\mathcal{T}$), a factor that is independent of the input size. Elaborating on this, as also discussed in Section 3.2.2, *unrolling* (or tile size) affects the register usage of each thread. Increasing the tile size leads to higher register usage per thread, which improves individual thread performance but decreases occupancy. The register footprint can be profiled offline with negligible overhead [54].

Workload partition refers to partitioning of the entire computation to each thread (in most implementations it is based on the output shape). We calculate the number of work groups (**WGs**) as follows:

$$\mathbf{WGs} = \prod_j \left\lceil \frac{\mathbf{O}_j}{\mathcal{T}_j \cdot \mathcal{G}_j} \right\rceil. \tag{5}$$

Here, j is the index of the output dimension, $\mathbf{O}_j$ is the corresponding dimension size, $\mathcal{T}_j$ and $\mathcal{G}_j$ are the thread-level tile size and work-group size on the jth dimension, respectively.

As demonstrated in Sec. 3.2.2, the concurrency capability at the warp level is determined by both the occupancy and the number of streaming processors. To model this, we calculate the total number of *occupancy group* across all streaming processors, denoted as $\mathcal{L}_g$:

$$\mathcal{L}_g(\mathcal{T}, \mathcal{G}) = \left\lceil \frac{\mathbf{WGs}}{\mathbf{SPs} \times \text{occupancy}} \right\rceil. \tag{6}$$

Integration. The overall performance model for a kernel is expressed as $\mathcal{L} = \mathcal{L}_w \cdot \mathcal{L}_g$, encompassing both warp-level and work-group-level factors. Specifically, we define it as:

$$\sum_i \mathcal{L}_t(\mathbf{H}_i) \times D^{\lceil \frac{(\sum_i \mathcal{W}_i \times s_i - C)}{C} \rceil} \cdot \left\lceil \frac{\prod_j \lceil \frac{\mathbf{O}_j}{\mathcal{T}_j \cdot \mathcal{G}_j} \rceil}{\mathbf{SPs} \times \text{occupancy}} \right\rceil. \tag{7}$$

[6]2D texture memory is read-only, so output write latency is not considered.

Applicability and extensibility. `TMModel` is broadly applicable across diverse scenarios. Given its lightweight design and analytical modeling approach, we discuss its potential extensions to two other cases: dynamic shapes and irregular workloads.

Dynamic shapes. `TMModel` explicitly accounts for input shapes, enabling it to handle dynamic input variables effectively. Owing to its analytical formulation and low overhead, `TMModel` offers a greater advantage over existing methods such as auto-tuning (e.g., Table 5). It can adapt to dynamic input shapes by efficiently recomputing predictions at runtime.

Irregular workloads. Our current work focuses on optimizing dense and regular kernels, which are predominant in modern DNNs. Extending precise occupancy modeling to irregular workloads is part of our future work. This is a fundamentally challenging task due to data-dependent access patterns (e.g., sparse matrices) and additional complexity from unpredictable thread block scheduling. Notably, vendor documentation [55] states: "For Adreno GPUs, developers cannot assume the launch order of workgroups or warps in SPs."

5 Model-based Code Optimization

We implement the cost model as a performance modeling module and integrate it with a state-of-the-art DNN compilation and acceleration framework (DNNFusion [47]) for mobile. Similar to other DNN compilation frameworks [9, 21], DNNFusion leverages a tensor-based DSL [48] and compute/schedule separation principle as Halide [57] to generate multi-versions of kernel codes in OpenCL. Halide [57] is a domain-specific language for image and tensor computations whose core design separates algorithm specification (what to compute) from scheduling (how to compute). We follow a similar design principle, allowing developers to explore diverse code transformations—such as loop tiling, unrolling, and memory layout changes—without modifying the underlying computation logic. To support data access patterns previously discussed, we extend DNNFusion by adding necessary schedules, data storage, and assisted code templates. We also extend DNNFusion to support backward propagation, thus supporting on-device DNN training.

The cost model can help identify the optimal combination of *data access pattern* (and its corresponding memory allocation), *output tile size* (or loop unrolling factor), and *thread configuration* (including warp shape and work-group shape). The first two are *implementation factors* and will be used by our compiler to generate the corresponding schedule and kernel code. The third is a *configuration factor* and will be passed to the generated kernel code as a parameter.

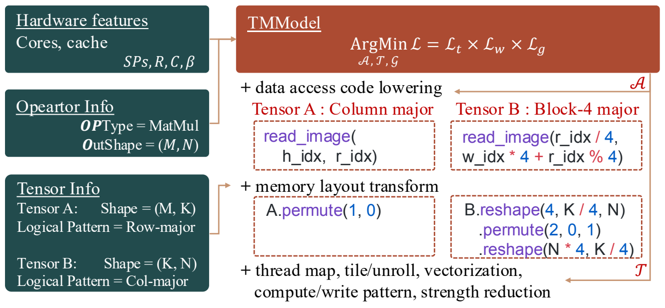
Hardware features
Cores, cache
SPs, R, C, β
Opeartor Info
$\boldsymbol{OP}$Type = MatMul
$\boldsymbol{O}$utShape = (M, N)
Tensor Info
Tensor A: Shape = (M, K)
Logical Pattern = Row-major
Tensor B: Shape = (K, N)
Logical Pattern = Col-major
TMModel
$\underset{\mathcal{A},\mathcal{T},\mathcal{G}}{\text{ArgMin}}\, \mathcal{L} = \mathcal{L}_t \times \mathcal{L}_w \times \mathcal{L}_g$
+ data access code lowering
$\mathcal{A}$
Tensor A : Column major
Tensor B : Block-4 major
read_image(h_idx, r_idx)
read_image(r_idx / 4, w_idx * 4 + r_idx % 4)
+ memory layout transform
A.permute(1, 0)
B.reshape(4, K / 4, N) .permute(2, 0, 1) .reshape(N * 4, K / 4)
+ thread map, tile/unroll, vectorization, compute/write pattern, strength reduction
$\mathcal{T}$

Figure 7: Compiler workflow. A `MatMul` example that generates two data access patterns: `Column` and `Block-4` major. Both data access pattern and memory layout transformation are needed. Other optimizations happen subsequently.

```
__constant sampler_t sampler =
    CLK_NORMALIZED_COORDS_FALSE | CLK_ADDRESS_CLAMP |
     CLK_FILTER_NEAREST;

__kernel void matmul(
  __read_only image2d_t A,
  __read_only image2d_t B,
  __write_only image2d_t C) {
// thread mapping
  const int w_idx = get_global_id(0);
  const int h_idx = get_global_id(1);

  float4 c = 0;
// REDUCE_SIZE is determined by the tile/unroll schedule
  for(short r_idx=0; r_idx<{{REDUCE_SIZE}}; r_idx++) {
    float4 a = read_imagef(
      A, sampler, (int2)(h_idx, r_idx));
    float4 b = read_imagef(
      B, sampler, (int2)(r_idx/4, w_idx*4+r_idx%4));

// Compute pattern is determined by vectorize schedule
    c = mad(a.x, b, c);
    c = mad(a.y, b, c);
    c = mad(a.w, b, c);
    c = mad(a.z, b, c);
  }
// OUT_IDX is determined by the write pattern
// schedule, which defines the output tensor layout
// optimized for downstream consumers.
  write_imagef(
    C, (int2)({{OUT_IDX(w_idx, h_idx)}}), c);
}
```

Listing 1: MatMul Kernel Example. This kernel is generated after applying the data access pattern schedule. Highlighted placeholders (e.g., `REDUCE_SIZE`, `OUT_IDX`) are determined by subsequent schedules such as tiling, unrolling, and write pattern.

It is worth noting that although we have deployed `TMModel` for code generation and optimization in DNNFusion, it equally applies to both other DNN compilation frameworks like TVM [9] for code generation and library-based DNN acceleration frameworks like MNN [30] for kernel selection on mobile GPUs with 2D memory.

Figure 7 shows an indicative workflow of using `TMModel` to optimize `MatMul`. This example also shows more details about the schedule and code lowering for several data access patterns. These schedules are also supported by Halide [57], specifically involving the design of appropriate *domain orders* (and loop transformations) described in Section 3.2 of [57]. The correctness of these schedules is guaranteed because the change in data access pattern and data storage order still maintains the original logical data access order. The schedule describing the data access pattern is decided before all other schedules (e.g., *tiling* and *unrolling*) and supports individual kernel and fused kernel in DNNFusion [47]. Listing 1 presents the corresponding lowered OpenCL code, illustrating the impact of different access patterns on the generated kernel. This example omits subsequent schedules (e.g., tiling) as they are natively supported by Halide.

6 Evaluation

This section assesses the efficacy of TMModel by comparing it with *four* state-of-the-art mainstream product-level mobile DNN acceleration frameworks (and their associated performance modeling/tuning methods): MNN [30], CLML [1], TFLite [2], and TVM [9]. We also compare against a more general-purpose GPU performance modeling method (PPT-GPU [3]). Both individual kernels and end-to-end on-device DNN training are used as target workloads. More specifically, the *evaluation objectives* are as follows: 1) demonstrating that TMModel can lead to better kernel performance, outperforming all baseline methods (Section 6.2); 2) showing the efficacy of TMModel in end-to-end full on-device DNN training (Section 6.3); 3) performing an *ablation study*, i.e., understanding the effectiveness of different components of our cost model, including verifying the model's prediction accuracy at each level (Section 6.4); 4) verifying the cost of TMModel-based code optimization by comparing with other baselines, validating the portability and generality of TMModel by showing its performance on different mobile GPUs, and evaluating its inference speed against other inference frameworks (Section 6.5).

6.1 Experiment Settings

Baselines. Table 2 summarizes they key aspects of five baselines used in our evaluation. Particularly, MNN, CLML, and TFLite explore different work-group sizes using brute-force search and select the best one. The work-group size (G) configuration is independent of kernel implementations, but it is important for achieving optimal performance. TVM's auto-tuning approach predicts kernel latency by an ML-based cost model and searches optimization factors (e.g., tile sizes) by generating code and profiling. TFLite performs heuristic kernel selection based on input shapes. TMModel is also compared with a general GPGPU memory modeling approach (originally for NVIDIA GPUs), PPT-GPU [3] that calculates

Table 2: Baselines characterization. A: data access pattern; T: thread tile; W: Warp shape; G: Work-group size.

Framework	MNN	CLML	TVM	TFLITE	PPT-GPU	TMModel
Perf. model	searching	searching	ML-Based	searching	analytical	analytical
Opt. target	G	G	T, G	T, G	A	A, T, W, G
m-GPU train	Yes	Yes	No	No	No	Yes

cache miss rates given the memory trace/data access order. Cache miss rate is directly related to the warp shape (temporal locality), so PPT-GPU implicitly searches warp shape. Compared to TMModel, PPT-GPU does not target 2D texture memory, it requires extensive knowledge of GPU hardware parameters (e.g., the cache associativity), and relies on memory trace to calculate reuse distance. On a mobile GPU hardware parameters are not documented, and no tracing tools are available to extract address information. To overcome this, we follow the process outlined in earlier work [62] to generate memory traces, in addition to using the information on associativity collected in our experiments [59, 70].

Workloads. TMModel is evaluated on representative individual DNN operators and end-to-end DNN on-device training: *Individual kernels/operators.* The evaluated kernels include Matrix Multiplication (MatMul), 2D Convolution with strides of 1 (Conv2Ds1) and 2 (Conv2Ds2), 2D DeConvolution with strides of 1 (DeConv2Ds1) and 2 DeConv2Ds2, 3D Convolution (Conv3D), 2D Depthwidth Convolution (DwConv2D), SoftMax, and LayerNorm. Their typical application DNNs and ranges of input sizes for our evaluation are summarized in the left half of Table 3. To optimize these kernels, TMModel explores the best access patterns among five pre-defined ones (as shown in Fig. 3) independently for each input: Column-major, Row-major, and three Block-major ones with reuse distance of 2, 4, and 8. Furthermore, varying tiling sizes are considered for each pattern.

On-device training. Training of four representative DNN models is also used for TMModel's evaluation: we use three classic Convolutional Neural Networks with different major kernels, i.e., MobileNet [60] with DwConv2D, ResNet-18 [23] with Conv2D, and Fsrcnn [16] with DeConv2D, and a popular vision transformer Vit [17] with MatMul. We use on-device training for our *full application* evaluation because: 1) it becomes increasingly important for up/downstream applications such as federated learning [42]; 2) it is more challenging than inference – specifically, the forward-backward computational graph has more shape variations and more complex data dependencies; 3) its execution on mobile GPUs remains an open challenge [13, 73]. Our evaluation focuses on the kernel optimization aspects of on-device training without resorting to algorithm optimizations (e.g. the ones that may sacrifice numerical precision [36] or memory space [37]) – these optimizations are orthogonal to TMModel.

Table 3: Individual kernel performance improvements over baselines. Numbers outside and within parentheses denote the *average* and *peak* (tagged w/ ↑) speedup over baseline, respectively. '-' means this framework does not support this kernel on mobile GPU. The third column specifies the input size range of each input: `MatMul` has two inputs, each with each dimension size ranging 2^4-2^{12} (double every time). For `Conv`, two sets are for input matrix and kernel size of weight matrix, respectively.

Kernels	Typical applications	Shape range	Avg. speedup over MNN	CLML	TVM	TFLite	PPT-GPU
MatMul	Transformers	2^{4-12}	2.57 (3.73 ↑)	6.04 (9.17 ↑)	5.34 (11.9 ↑)	1.20 (3.22 ↑)	1.47 (2.78 ↑)
Conv2Ds1	CNNs	$2^{4-9}, 3-7$	1.35 (1.57 ↑)	2.06 (2.55 ↑)	1.18 (1.80 ↑)	1.12 (1.53 ↑)	1.07 (1.38 ↑)
Conv2Ds2	Downsample	$2^{4-9}, 3-7$	1.19 (1.82 ↑)	1.42 (2.49 ↑)	1.42 (4.36 ↑)	1.20 (1.63 ↑)	1.16 (1.81 ↑)
DeConv2Ds1	Conv backward	$2^{4-9}, 3-7$	2.16 (2.41 ↑)	2.32 (2.70 ↑)	1.71 (4.13 ↑)	1.45 (3.49 ↑)	1.79 (2.41 ↑)
DeConv2Ds2	Upsample	$2^{4-9}, 3-7$	2.18 (2.42 ↑)	6.57 (7.45 ↑)	3.96 (8.60 ↑)	1.20 (2.53 ↑)	1.27 (2.42 ↑)
Conv3D	Action recognition	$2^{4-9}, 3-7$	–	–	6.62 (8.67 ↑)	–	4.47 (8.67 ↑)
DwConv2D	CNN	$2^{4-9}, 3-7$	2.49 (3.41 ↑)	1.62 (4.11 ↑)	2.10 (4.85 ↑)	1.76 (3.24 ↑)	2.13 (3.41 ↑)
SoftMax	Transformers	2^{4-12}	4.85 (8.88 ↑)	3.62 (12.5 ↑)	2.32 (4.36 ↑)	1.31 (8.83 ↑)	2.28 (8.88 ↑)
LayerNorm	Transformers	2^{4-12}	3.43 (4.34 ↑)	–	7.77 (18.2 ↑)	3.64 (14.1 ↑)	2.53 (4.34 ↑)
Geometric Mean			2.31 (3.08↑)	2.86 (4.82↑)	2.91 (6.06↑)	1.48 (3.61↑)	1.83 (3.30↑)

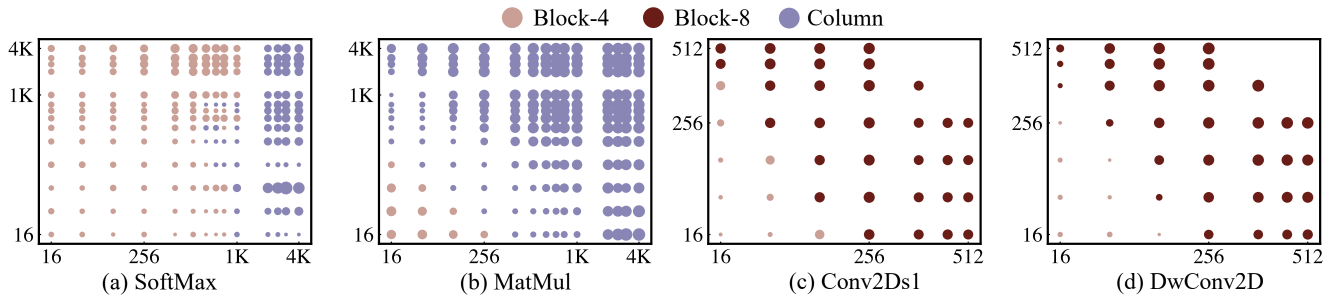

Figure 8: Output shape impacts kernel selection. Circle colors represent different implementations. Circle sizes represent the speed improvements over the base implementation which is a vanilla version without tiling or data access pattern optimizations, but with fine-tuned work-group size. The x- and y-axis denote the height and width of output geometry.

Evaluation environment. Evaluations are carried out on the Qualcomm Snapdragon 8 Gen 2 SoC (`SND Gen2`) [56] with a Qualcomm Adreno 740 GPU, and version 3.0 of the OpenCL driver. Additionally, MediaTek D1100 [43] SoC with Mali G77 GPU is used to demonstrate portability. For individual DNN kernels, each experiment was run 10 times, with the 'warm-up' run excluded, and the average execution time of other runs is reported. The confidence interval is omitted for readability, as the variations were low. For the on-device training experiments, a consistent configuration is applied for each framework and test model: the batch size is set to one, and Stochastic Gradient Descent (SGD) is used to update model parameters once every 32 batches. This approach is commonly used [58, 68] to save memory space while maintaining the same training effectiveness as a standard batch size of 32. The reported latency is the average execution time for one batch, calculated over 32 batches. To maintain training accuracy, floating-point 32 is used for on-device training evaluation (as well as for individual kernel evaluation).

6.2 Overall Performance

Table 3 summarizes the generated code speedup enabled by `TMModel` over all five baselines. The numbers outside and inside parentheses denote the average and peak speedup respectively of `TMModel` over the corresponding baseline across all input sizes for each kernel. For example, for `MatMul`, `TMModel` achieves an average speedup of 2.57× and a peak speedup of 3.73× over `MNN` across all input sizes (size of each dimension ranges from 2^4 to 2^{12}). For all individual kernels, the geometric means of the average speedups of `TMModel` over `MNN`, `CLML`, `TVM`, `TFLite`, and `PPT-GPU` are 2.31×, 2.86×, 2.91×, 1.48×, and 1.83×, respectively, and the geometric means of the peak speedups are 3.08×, 4.82×, 6.06×, 3.61×, and 3.30×, respectively, demonstrating `TMModel`'s efficacy.

`TMModel` outperforms other frameworks for two main reasons: First, it systematically considers multiple optimization factors (data access patterns, thread-level tiling, and warp and work-group level thread mapping) simultaneously, while others mainly focus on one or two of them (see Table 2). For

Table 4: On-device training performance comparison.

Models	Major Kernels	Latency (ms)		
		MNN	DNNF	TMModel
MobileNetV2 [60]	DwConv2D	1916	41	29
ResNet-18 [23]	Conv2D	88	66	48
Fsrcnn [16]	DeConv2D	921	57	32
ViT [17]	MatMul	2105	1224	575

example, MNN only supports work-group fine-tuning without exploring kernel implementation options. Second, and more importantly, TMModel takes into account the 2D texture memory (and cache) and couples it with thread mapping and occupancy considerations. Examining results from other frameworks – although PPT-GPU models cache behavior, it does not take texture representation, occupancy, or output shape into account for memory access and thread mapping optimizations. TFLite selects different implementations for specific input/output shapes, so it performs the best among all baselines. However, it also misses to consider texture representation and occupancy in a nuanced manner. CLML searches only a limited set of candidates and misses many optimization opportunities. Finally, TVM lacks specialized templates for many kernels on mobile GPUs and fallbacks to using a generic template.

Output shapes impact kernel configuration selection. An advantage of TMModel is the consideration of output shape. Fig. 8 verifies this by showing the distribution of best-performing implementations for four representative kernels - SoftMax, MatMul, Conv2Ds1, and DwConv2D[7]. For example, for SoftMax and MatMul, with small input shapes and low parallelism, Block-major access is preferable to achieve low single-thread data access latency ($\mathcal{L}_t$). However, as the input size increases, using Block-major (with large s) and more threads (larger $\mathcal{W}$) would cause cache thrashing, resulting in degraded performance (increased $\mathcal{L}_w$ thus $\mathcal{L}_g$). For DwConv2D and Conv2D, Block-major access still results in ideal performance as their input sizes grow because their input of the *Weight* matrix has high data reuse (and temporal locality), effectively reducing cache thrashing as thread count grows. For readability, Fig. 8 only shows the selection of data access patterns, omitting the tiling factor determination.

6.3 End-to-End DNN On-Device Training

Among all five baselines, MNN is the state-of-the-art framework to support end-to-end DNN on-device training on mobile GPUs. Table 4 compares TMModel with MNN on four representative DNN models with varied major kernels. TMModel achieves 1.83 – 66.1× speedup over MNN for two main reasons: First, TMModel is built on DNNFusion [47] (with the latest memory optimization [49]), which has more advanced operator fusion and memory operation elimination than MNN. Second, TMModel further optimizes DNNFusion with the newly designed 2D texture-aware analytic model to generate more efficient operators (or kernels), with a nuanced selection of data access patterns, tiling, and other GPU configuration parameters. Table 4 also compares TMModel with DNNFusion, showing 1.38 – 2.53× speedup on these four DNN models.

[7]Other kernels show similar trends.

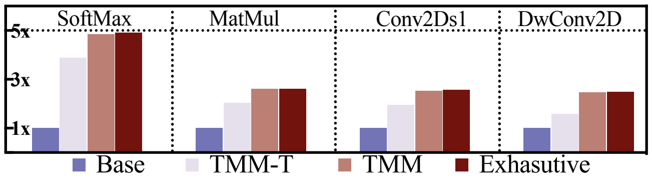

Figure 9: Performance breakdown with varied model outputs. TMM refers to TMModel and TMM-T uses predicted tiling factor only for code opt. Base is a vanilla version as Fig 8 while Exhaustive refers to a fully optimized version with exhaustive search. y-axis is the speedup over Base.

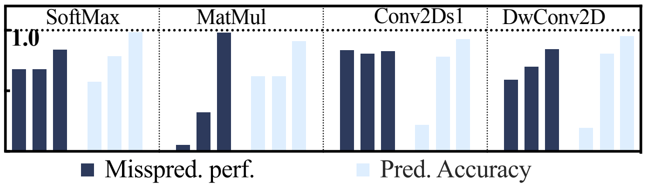

Figure 10: Prediction accuracy and average latency for all miss-prediction cases with each level of TMModel for four kernels. The bars of each group from left to right represent thread-/warp-/integrated-level of TMModel.

6.4 Performance Understanding

To further understand TMModel's performance, this section reports an ablation study on the four representative kernels we previously used for the detailed evaluation.

Breakdown of performance improvement with varied model outputs. Fig. 9 shows a performance breakdown with varied model outputs: Base is a vanilla version w/o tiling or data access pattern optimizations, but w/ fine-tuned work-group size. TMM-T uses predicted tiling sizes from TMModel only for code optimizations. TMM uses both predicted tiling and data access patterns from TMModel. Exhaustive shows the upper bound achieved by exhaustively exploring the search space (A, T, W, G). This study shows that both the access pattern and tiling are essential factors for performance tuning on mobile GPUs and TMModel can achieve performance close (within around 1%) to that of exhaustive search.

Model prediction accuracy study. We focus on 1) the prediction accuracy of TMModel, and 2) what happens when TMModel produces inaccurate configurations. Fig. 10 summarizes the prediction accuracy of each level of TMModel

Table 5: Tuning latency comparison. The tuning latency is the total time for finding the best configurations for `Conv2Ds1` with all input sizes in Table 3.

Framework	MNN	CLML	TVM	TFLite	PPT-GPU	`TMModel`
Tuning Lat. (s)	73.4	10.4	234	552	771	1.92

for these four kernels[8], and the average latency of the miss-predicted cases by normalizing it to the latency of the best configuration with exhaustive search. These prediction accuracy results indicate that as more and more factors in `TMModel` are considered, its prediction accuracy improves, achieving 94.1% accuracy on average for these four operators when the entire performance model is used, demonstrating `TMModel`'s efficacy, and the necessity of each of its components. The miss-prediction performance demonstrates that for certain cases, even if `TMModel` produces a wrong prediction, the predicted configuration achieves performance similar (within 83.8% – 98.4%) to the correct one.

6.5 Other Experiments

Tuning Latency. Table 5 compares the total tuning latency of `TMModel` and all five baselines. We identify the best configurations for `Conv2Ds1` with all input sizes (shown earlier in Table 3). Compared with all baselines, `TMModel` accelerates the tuning stage by 5.42 – 402×. `MNN`, `CLML`, and `TFLite` are slower because of the exhaustive search and execution of the code candidates. `TVM`'s latency is also high because it iteratively profiles generated code on the mobile device and trains ML models. Although `TMModel` also needs a benchmarking stage with a regression model (as described in Section 3), this is performed once for each architecture, rather than for each kernel (like `TVM`). `PPT-GPU`, even though being an analytical performance model, is slow because it calculates the reuse distance with an expensive analysis of memory traces [46].

Portability Study. Table 6 compares the on-device training latency of `MNN` and `TMModel` on another GPU architecture, Mali G77, with one CNN (`MobileNetV2`) and one Transformer (`ViT`). Other DNN models show similar trends. The Mali GPU family differs significantly from Adreno GPUs – it has a smaller warp size (32 threads), a larger texture size (8KB), more shader cores (9), a distinct texture representation, and features separate on-chip texture and data caches. With these critical differences, targeting this device demonstrates the portability of `TMModel`. To maximize the utilization of both data paths, we empirically employ a mixed-use approach where tensor inputs and outputs are handled with `Image2D` objects, while model weights are stored in `Buffer` objects. The evaluation results show that `TMModel` outperforms `MNN` by 68.8× and 3.09× on `MobileNetV2` and `ViT`, respectively.

Table 6: Portability study

Models	Latency (ms)	
	MNN	`TMModel`
MobileNetV2	4060	59
ViT	5987	1936

Table 7: Inference performance comparison. All frameworks use FP16 precision and run on a Snapdragon 8 Gen 2 platform with an Adreno 740 GPU.

Models	Latency (ms)		
	MNN	DNNF	`TMModel`
ViT	533	277	88
ResNet-18	10.9	10.1	8.6

Inference Latency. `TMModel` is a general-purpose performance model applicable to both inference and training workloads. To evaluate its effectiveness for inference, Table 7 presents latency results on ViT and ResNet-18. `TMModel` achieves significant improvements, outperforming `MNN` by 6.06× on `ViT` and 1.27× on `ResNet-18`, and surpassing `DNNF` by 3.15× and 1.17×, respectively.

7 Related Work

Texture memory and micro-benchmarks. Texture cache was originally developed for faster rendering on graphic processors [15, 22, 26]. There have been several micro-benchmarking studies to understand the performance of systems with such a cache. The work by Wong et al. [70] and Mei et al. [44] only used 1D data (`Image1D`) to dissect texture cache organization – generalizability to Image2D was left unclear. Romou [39] profiled cache hierarchy on mobile SoCs, however, the 2D spacial locality of texture memory was not studied. Our work distinguishes itself by studying texture representation and its impact on both spatial and temporal locality.

Cache and locality analysis for 1D memory. Reuse distance [14, 31] is a well-known metric to model cache behavior, and is studied based on an ordered memory access trace. Nugteren et al. [50] extends reuse distance to GPU cache sets by modeling GPU's parallel execution paradigm and predicting cache miss rates. PPT-GPU-Mem [3], on the other hand, leverages a binary instrumentation tool NVBIT [63] to provide a reliable memory access trace to calculate reuse distance. Applying reuse distance to mobile platforms with texture cache is not straightforward, because no trace tools are available and the texture data representation is patented.

Cost models and tensor computing. Models of performance execution are designed for numerous objectives. Roofline model [69] identifies the bottleneck of hardware resources. Gables [24] retargets the Roofline model to model each accelerator on a SoC, apportions work concurrently

[8]A more detailed study is presented in our online Supplementary Material, Sec. IV.B, specifically, Fig.13 due to limited space.

among different accelerators, and calculates a SoC performance upper bound. Li et al. [38] extract features from the architecture and the program to train a generalizable cost model. MAESTRO [34] models DNN accelerators and estimates execution latency for a given the DNN model. Inspired by TVM [9] and Ansor [78], several works employ machine learning methods to predict inference latency and auto-tune DNN kernels [75, 77, 81]. TSM2 [8] designs a cost model to optimize tiling parameters for matrix multiplications with tall-and-skinny inputs. Babalad et al. [6] train machine learning models to select tile sizes and loop orders for CPUs. Jang et al. [29] optimize memory type selection by analyzing data access patterns in existing code. In contrast, our work focuses on generating kernels with efficient data access patterns tailored to the tensor shape and leveraging the 2D spatial locality of texture memory.

On-device training. There is a significant amount of work on DNN training on desktop GPUs [7, 11, 19, 74, 79], or other processors [27, 28, 72] – however, the challenges associated with distinctive features and resource limitations of mobile chips are not handled there. On the other hand, previous research on mobile devices has focused on optimizing on-device training performance through sparsification [76] and quantization [40, 73, 80]. Unlike these approaches, our methods do not compromise model accuracy. Additionally, a separate line of research primarily targets the orthogonal direction of memory consumption reduction [20, 40, 45, 67, 68].

8 Conclusion and Future Work

This paper presented a new performance modeling and optimization approach called `TMModel` for mobile GPUs that takes 2D texture memory into account. `TMModel` consists of three advances: 1) a micro-benchmarking and machine learning-based approach on top of a new concept (`cross-block stride`) to understand 2D spatial locality and analyze the impact of factors like warp size/shape, occupancy, and work-group size; 2) a comprehensive analytic model for choosing factors like data access pattern(s) and tile size(s) for a given kernel (and associated size/shapes), and 3) a complete system prototype that supports code optimization for both kernels and full DNN on-device training. `TMModel` is extensively evaluated by comparing with four state-of-the-art mobile DNN frameworks and a general-purpose GPU performance model, achieving up to 3.61× speedup on individual kernels and 66.1× speedup for end-to-end DNN on-device training with as low as 0.25% tuning cost.

Based on `TMModel`, our future work includes: 1) designing more advanced data layouts, and access patterns, 2) global layout selection, transformation, and memory elimination for DNN training, and 3) supporting irregular kernels (e.g., on sparse matrices or tensors).

Acknowledgments

The authors would like to thank the anonymous reviewers for their insightful and detailed comments. Their constructive suggestions significantly improved the quality of this paper. This work was supported in part by the U.S. National Science Foundation (NSF) under awards CCF-2047516 (CAREER), CCF-2146873, CNS-2230944, OAC-2403088, CCF-2428108, CNS-2341378, OAC-2333899, CCF-2333895, and CCF-2334273. It was also supported in part by the Horizon Europe research and innovation programme of the European Union under grant agreement No. 101092912 (project MLSysOps). Evgenia Smirni is partially supported by NSF under grant No. 2402942 and by the Commonwealth Cyber Initiative (CCI) under grant No. HC-3Q24-047. Any opinions, findings, or errors expressed in this material are those of the authors and do not necessarily reflect the views of the NSF, Horizon Europe, or CCI. The authors also acknowledge William & Mary Research Computing for providing the computational resources used in this study.

References

[1] 2022. Adreno OpenCL Machine Learning SDK v3.0. https://developer.qualcomm.com/downloads/adreno-opencl-machine-learning-sdk-v30.

[2] Martin 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 *OSDI 2016*. USENIX Association, USA, 265–283.

[3] Yehia Arafa, Abdel-Hameed Badawy, Gopinath Chennupati, Atanu Barai, Nandakishore Santhi, and Stephan Eidenbenz. 2020. Fast, accurate, and scalable memory modeling of GPGPUs using reuse profiles. In *Proceedings of the 34th ACM International Conference on Supercomputing*. 1–12.

[4] Yehia Arafa, Abdel-Hameed Badawy, Ammar ElWazir, Atanu Barai, Ali Eker, Gopinath Chennupati, Nandakishore Santhi, and Stephan Eidenbenz. 2021. Hybrid, scalable, trace-driven performance modeling of GPGPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.

[5] ARM. 2024. Arm Immortalis and Mali GPU OpenCL Developer Guide. (2024).

[6] Shilpa Babalad, Shirish Shevade, Matthew Jacob Thazhuthaveetil, and R Govindarajan. 2024. Tile Size and Loop Order Selection using Machine Learning for Multi-/Many-Core Architectures. In *Proceedings of the 38th ACM International Conference on Supercomputing*. 388–399.

[7] Youhui Bai, Cheng Li, Quan Zhou, Jun Yi, Ping Gong, Feng Yan, Ruichuan Chen, and Yinlong Xu. 2021. Gradient compression supercharged high-performance data parallel dnn training. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 359–375.

[8] Jieyang Chen, Nan Xiong, Xin Liang, Dingwen Tao, Sihuan Li, Kaiming Ouyang, Kai Zhao, Nathan DeBardeleben, Qiang Guan, and Zizhong Chen. 2019. TSM2: optimizing tall-and-skinny matrix-matrix multiplication on GPUs. In *Proceedings of the ACM International Conference on Supercomputing*. 106–116.

[9] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.

[10] Yu-Hui Chen, Raman Sarokin, Juhyun Lee, Jiuqiang Tang, Chuo-Ling Chang, Andrei Kulik, and Matthias Grundmann. 2023. Speed is all you need: On-device acceleration of large diffusion models via gpu-aware optimizations. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 4650–4654.

[11] Zhaodong Chen, Andrew Kerr, Richard Cai, Jack Kosaian, Haicheng Wu, Yufei Ding, and Yuan Xie. 2024. EVT: Accelerating Deep Learning Training with Epilogue Visitor Tree. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 301–316.

[12] Thanh Tuan Dao, Jungwon Kim, Sangmin Seo, Bernhard Egger, and Jaejin Lee. 2014. A performance model for GPUs with caches. *IEEE Transactions on Parallel and Distributed Systems* 26, 7 (2014), 1800–1813.

[13] Anish Das, Young D Kwon, Jagmohan Chauhan, and Cecilia Mascolo. 2022. Enabling on-device smartphone GPU based training: Lessons learned. In *2022 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*. IEEE, 533–538.

[14] Chen Ding and Yutao Zhong. 2003. Predicting whole-program locality through reuse distance analysis. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*. 245–257.

[15] Michael Doggett. 2012. Texture caches. *IEEE Micro* 32, 3 (2012), 136–141.

[16] Chao Dong, Chen Change Loy, and Xiaoou Tang. 2016. Accelerating the super-resolution convolutional neural network. In *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part II 14*. Springer, 391–407.

[17] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. 2020. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929* (2020).

[18] Minquan Fang, Jianbin Fang, Weimin Zhang, Haifang Zhou, Jianxing Liao, and Yuangang Wang. 2018. Benchmarking the GPU memory at the warp level. *Parallel Comput.* 71 (2018), 23–41.

[19] Trevor Gale, Deepak Narayanan, Cliff Young, and Matei Zaharia. 2023. Megablocks: Efficient sparse training with mixture-of-experts. *Proceedings of Machine Learning and Systems* 5 (2023), 288–304.

[20] In Gim and JeongGil Ko. 2022. Memory-efficient DNN training on mobile devices. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*. 464–476.

[21] Google. 2023. Tensorflow XLA. https://www.tensorflow.org/xla.

[22] Ziyad S Hakura and Anoop Gupta. 1997. The design and analysis of a cache architecture for texture mapping. In *Proceedings of the 24th annual international symposium on Computer architecture*. 108–120.

[23] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[24] 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)*. IEEE, 317–330.

[25] Mark D Hill and Alan Jay Smith. 1989. Evaluating associativity in CPU caches. *IEEE Trans. Comput.* 38, 12 (1989), 1612–1630.

[26] Homan Igehy, Matthew Eldridge, and Kekoa Proudfoot. 1998. Prefetching in a texture cache architecture. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*. 133–ff.

[27] Vahid Janfaza, Shantanu Mandal, Farabi Mahmud, and Abdullah Muzahid. 2023. ADA-GP: Accelerating DNN Training By Adaptive Gradient Prediction. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 1092–1105.

[28] Vahid Janfaza, Kevin Weston, Moein Razavi, Shantanu Mandal, Farabi Mahmud, Alex Hilty, and Abdullah Muzahid. 2023. Mercury: Accelerating dnn training by exploiting input similarity. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 638–650.

[29] Byunghyun Jang, Dana Schaa, Perhaad Mistry, and David Kaeli. 2010. Exploiting memory access patterns to improve memory performance in data-parallel architectures. *IEEE Transactions on Parallel and Distributed Systems* 22, 1 (2010), 105–118.

[30] Xiaotang Jiang, Huan Wang, Yiliu Chen, Ziqi Wu, Lichuan Wang, Bin Zou, Yafeng Yang, Zongyang Cui, Yu Cai, Tianhang Yu, Chengfei Lv, and Zhihua Wu. 2020. MNN: A universal and efficient inference engine. *Proceedings of Machine Learning and Systems* 2 (2020), 1–13.

[31] Yunlian Jiang, Eddy Z Zhang, Kai Tian, and Xipeng Shen. 2010. Is reuse distance applicable to data locality analysis on chip multiprocessors?. In *Compiler Construction: 19th International Conference, CC 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings 19*. Springer, 264–282.

[32] Mahmoud Khairy, Zhesheng Shen, Tor M Aamodt, and Timothy G Rogers. 2020. Accel-Sim: An extensible simulation framework for validated GPU modeling. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 473–486.

[33] Mohsen Kiani and Amir Rajabzadeh. 2018. Efficient cache performance modeling in GPUs using reuse distance analysis. *ACM Transactions on Architecture and Code Optimization (TACO)* 15, 4 (2018), 1–24.

[34] Hyoukjun Kwon, Prasanth Chatarasi, Michael Pellauer, Angshuman Parashar, Vivek Sarkar, and Tushar Krishna. 2019. Understanding reuse, performance, and hardware cost of dnn dataflow: A data-centric approach. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 754–768.

[35] Raphael Landaverde, Tiansheng Zhang, Ayse K Coskun, and Martin Herbordt. 2014. An investigation of unified memory access performance in cuda. In *2014 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–6.

[36] Andrew Lavin and Scott Gray. 2016. Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4013–4021.

[37] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

[38] Lingda Li, Thomas Flynn, and Adolfy Hoisie. 2024. Learning Generalizable Program and Architecture Representations for Performance Modeling. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.

[39] Rendong Liang, Ting Cao, Jicheng Wen, Manni Wang, Yang Wang, Jianhua Zou, and Yunxin Liu. 2022. Romou: Rapidly generate high-performance tensor kernels for mobile gpus. In *Proceedings of the 28th Annual International Conference on Mobile Computing And Networking*. 487–500.

[40] Ji Lin, Ligeng Zhu, Wei-Ming Chen, Wei-Chen Wang, Chuang Gan, and Song Han. 2022. On-device training under 256kb memory. *Advances in Neural Information Processing Systems* 35 (2022), 22941–22954.

[41] Weile Luo, Ruibo Fan, Zeyu Li, Dayou Du, Qiang Wang, and Xiaowen Chu. 2024. Benchmarking and dissecting the nvidia hopper gpu architecture. *arXiv preprint arXiv:2402.13499* (2024).

[42] H. Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. 2017. Communication-Efficient Learning of Deep Networks from Decentralized Data. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics (AISTATS)*, Vol. 54. 1273–1282.

[43] MediaTek. 2023. MediaTek Dimensity 1100. https://www.mediatek.com/products/tablets/mediatek-dimensity-1100.

[44] Xinxin Mei and Xiaowen Chu. 2016. Dissecting GPU memory hierarchy through microbenchmarking. *IEEE Transactions on Parallel and Distributed Systems* 28, 1 (2016), 72–86.

[45] Ji Joong Moon, Parichay Kapoor, Ji Hoon Lee, Myung Joo Ham, and Hyun Suk Lee. 2022. NNTrainer: Light-weight on-device training framework. *arXiv preprint arXiv:2206.04688* (2022).

[46] Qingpeng Niu, James Dinan, Qingda Lu, and Ponnuswamy Sadayappan. 2012. PARDA: A fast parallel reuse distance analysis algorithm. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. IEEE, 1284–1294.

[47] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. 2021. DNNFusion: accelerating deep neural networks execution with advanced operator fusion. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 883–898.

[48] Wei Niu, Zhengang Li, Xiaolong Ma, Peiyan Dong, Gang Zhou, Xuehai Qian, Xue Lin, Yanzhi Wang, and Bin Ren. 2021. Grim: A general, real-time deep learning inference framework for mobile devices based on fine-grained structured weight sparsity. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 44, 10 (2021), 6224–6239.

[49] Wei Niu, Md Musfiqur Rahman Sanim, Zhihao Shu, Jiexiong Guan, Xipeng Shen, Miao Yin, Gagan Agrawal, and Bin Ren. 2024. SmartMem: Layout Transformation Elimination and Adaptation for Efficient DNN Execution on Mobile. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 916–931.

[50] Cedric Nugteren, Gert-Jan Van den Braak, Henk Corporaal, and Henri Bal. 2014. A detailed GPU cache model based on reuse distance theory. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 37–48.

[51] Nvidia. 2024. CUDA C++ Programming Guide. (2024).

[52] Nvidia. 2024. Hopper Tuning Guide. (2024).

[53] Reena Panda, Xinnian Zheng, Jiajun Wang, Andreas Gerstlauer, and Lizy K John. 2017. Statistical pattern based modeling of GPU memory access streams. In *Proceedings of the 54th Annual Design Automation Conference 2017*. 1–6.

[54] Qualcomm. 2016. Snapdragon Profiler. https://developer.qualcomm.com/software/snapdragon-profiler.

[55] Qualcomm. 2023. Qualcomm Snapdragon Mobile Platform OpenCL General Programming and Optimization. (2023).

[56] Qualcomm. 2023. Snapdragon Gen2. https://en.wikipedia.org/wiki/List_of_Qualcomm_Snapdragon_systems_on_chips.

[57] 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 *PLDI 2013* (Seattle, Washington, USA). Association for Computing Machinery, New York, NY, USA, 519–530.

[58] Joseph Redmon. 2013–2016. Darknet: Open Source Neural Networks in C. http://pjreddie.com/darknet/.

[59] Rafael H Saavedra and Alan Jay Smith. 1995. Measuring cache and TLB performance and their effect on benchmark runtimes. *IEEE Trans. Comput.* 44, 10 (1995), 1223–1235.

[60] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4510–4520.

[61] Yuki Sugimoto, Fumihiko Ino, and Kenichi Hagihara. 2014. Improving cache locality for GPU-based volume rendering. *Parallel Comput.* 40, 5-6 (2014), 59–69.

[62] Tao Tang, Xuejun Yang, and Yisong Lin. 2011. Cache miss analysis for gpu programs based on stack distance profile. In *2011 31st International Conference on Distributed Computing Systems*. IEEE, 623–634.

[63] Oreste Villa, Mark Stephenson, David Nellans, and Stephen W Keckler. 2019. Nvbit: A dynamic binary instrumentation framework for nvidia gpus. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 372–383.

[64] Vasily Volkov. 2016. *Understanding latency hiding on GPUs*. University of California, Berkeley.

[65] Vasily Volkov and James W Demmel. 2008. Benchmarking GPUs to tune dense linear algebra. In *SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE, 1–11.

[66] Lu Wang, Magnus Jahre, Almutaz Adileho, and Lieven Eeckhout. 2020. MDM: The GPU memory divergence model. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1009–1021.

[67] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. 2018. Superneurons: Dynamic GPU memory management for training deep neural networks. In *Proceedings of the 23rd ACM SIGPLAN symposium on principles and practice of parallel programming*. 41–53.

[68] Qipeng Wang, Mengwei Xu, Chao Jin, Xinran Dong, Jinliang Yuan, Xin Jin, Gang Huang, Yunxin Liu, and Xuanzhe Liu. 2022. Melon: Breaking the memory wall for resource-efficient on-device machine learning. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*. 450–463.

[69] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (2009), 65–76.

[70] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. 2010. Demystifying GPU microarchitecture through microbenchmarking. In *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*. IEEE, 235–246.

[71] Bo Wu, Zhijia Zhao, Eddy Zheng Zhang, Yunlian Jiang, and Xipeng Shen. 2013. Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on gpu. *ACM SIGPLAN Notices* 48, 8 (2013), 57–68.

[72] Zhen Xie, Murali Emani, Xiaodong Yu, Dingwen Tao, Xin He, Pengfei Su, Keren Zhou, and Venkatram Vishwanath. 2024. Centimani: Enabling Fast {AI} Accelerator Selection for {DNN} Training with a Novel Performance Predictor. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. 1203–1221.

[73] Daliang Xu, Mengwei Xu, Qipeng Wang, Shangguang Wang, Yun Ma, Kang Huang, Gang Huang, Xin Jin, and Xuanzhe Liu. 2022. Mandheling: Mixed-precision on-device dnn training with dsp offloading. In *Proceedings of the 28th Annual International Conference on Mobile Computing And Networking*. 214–227.

[74] Lang Xu, Quentin Anthony, Qinghua Zhou, Nawras Alnaasan, Radha Gulhane, Aamir Shafi, Hari Subramoni, and Dhabaleswar K DK Panda. 2024. Accelerating Large Language Model Training with Hybrid GPU-based Compression. In *2024 IEEE 24th International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 196–205.

[75] Yufan Xu, Qiwei Yuan, Erik Curtis Barton, Rui Li, P Sadayappan, and Aravind Sukumaran-Rajam. 2022. Effective Performance Modeling and Domain-Specific Compiler Optimization of CNNs for GPUs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. 252–264.

[76] Geng Yuan, Xiaolong Ma, Wei Niu, Zhengang Li, Zhenglun Kong, Ning Liu, Yifan Gong, Zheng Zhan, Chaoyang He, Qing Jin, et al. 2021. Mest: Accurate and fast memory-economic sparse training framework on the edge. *Advances in Neural Information Processing Systems* 34 (2021), 20838–20850.

[77] Li Lyna Zhang, Shihao Han, Jianyu Wei, Ningxin Zheng, Ting Cao, Yuqing Yang, and Yunxin Liu. 2021. Nn-meter: Towards accurate latency prediction of deep-learning model inference on diverse edge devices. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*. 81–93.

[78] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. 2020. Ansor: Generating {High-Performance} tensor programs for deep learning. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*. 863–879.

[79] Zhen Zheng, Xuanda Yang, Pengzhan Zhao, Guoping Long, Kai Zhu, Feiwen Zhu, Wenyi Zhao, Xiaoyong Liu, Jun Yang, Jidong Zhai, Shuaiwen Leon Song, and Wei Lin. 2022. AStitch: enabling a new multi-dimensional optimization space for memory-intensive ML training and inference on modern SIMT architectures. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 359–373.

[80] Qihua Zhou, Song Guo, Zhihao Qu, Jingcai Guo, Zhenda Xu, Jiewei Zhang, Tao Guo, Boyuan Luo, and Jingren Zhou. 2021. Octo:{INT8} training with loss-aware compensation and backward quantization for tiny on-device learning. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 177–191.

[81] Hongyu Zhu, Ruofan Wu, Yijia Diao, Shanbin Ke, Haoyu Li, Chen Zhang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Wei Cui, Fan Yang, Mao Yang, Lidong Zhou, Asaf Cidon, and Gennady Pekhimenko. 2022. {ROLLER}: Fast and efficient tensor compilation for deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 233–248.