Romou: Rapidly Generate High-Performance Tensor Kernels for Mobile GPUs

Rendong Liang*
Microsoft Research
University of California, Irvine
rendongl@uci.edu

Ting Cao
Microsoft Research
ting.cao@microsoft.com

Jicheng Wen
Microsoft STCA
jicheng.wen@microsoft.com

Manni Wang*
Microsoft Research
Xi'an Jiao Tong University
manny.w.m.n@stu.xjtu.edu.cn

Yang Wang*
Microsoft Research
t-yangwa@microsoft.com

Jianhua Zou
Xi'an Jiao Tong University
jhzou@sei.xjtu.edu.cn

Yunxin Liu†
Institute for AI Industry Research
(AIR), Tsinghua University
liuyunxin@air.tsinghua.edu.cn

Abstract

Mobile GPU, as a ubiquitous and powerful accelerator, plays an important role in accelerating on-device DNN (Deep Neural Network) inference. The frequent-upgrade and diversity of mobile GPUs require automatic kernel generation to empower fast DNN deployment. However, current generated kernels have poor performance.

The goal of this paper is to rapidly generate high-performance kernels for diverse mobile GPUs. The major challenges are (1) it is unclear about what is the optimal kernel due to the lack of hardware knowledge; (2) how to rapidly generate it from a large space of candidates. For the first challenge, we propose a cross-platform profiling tool, the first to disclose and quantify mobile GPU architecture. The result demystifies the hardware bottleneck, and also directs the solution for the second challenge by exposing the unique high-performance hardware feature, identifying inefficient kernels against hardware constraints, and specifying performance bound for kernels.

Directed by that, we propose a mobile-GPU-specific kernel compiler *Romou*. It supports the unique hardware feature in kernel implementation, and prunes inefficient ones against hardware resources. Romou can thus rapidly generate high-performance kernels. Compared to the state-of-the-art generated kernels, it achieves up-to 14.7× speedup on average for convolution. Up-to 99% search space is pruned. The performance is even up-to 1.2× faster on average than the state-of-the-art hand-optimized implementation.

*Work is done during internship at Microsoft Research.
†Corresponding author.

ACM MobiCom '22, October 24–28, 2022, Sydney, NSW, Australia

ACM ISBN 978-1-4503-9181-8/22/10…$15.00
https://doi.org/10.1145/3495243.3517020

CCS Concepts

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

Keywords

Mobile GPU, Automatic Code Generation, Architecture Profiling, Deep Neural Networks

ACM Reference Format:
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 *The 28th Annual International Conference on Mobile Computing and Networking (ACM MobiCom '22), October 24–28, 2022, Sydney, NSW, Australia.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3495243.3517020

1 Introduction

Mobile applications provide vast deployment scenarios for DNNs. On-device DNN inference is gaining prevalence compared to the server counterpart, due to its specific advantages in privacy protection, network resilience, and quick response.

Mobile GPUs are powerful, ubiquitous, and accessible accelerators. For example, the theoretical performance of Adreno 640 GPU on Google Pixel4 is 890 GFLOPS, 6× of its CPU performance. Almost every phone is equipped with a GPU and all these GPUs can be interfaced with standardized APIs. Dedicated DNN accelerators, on the other hand, are only available on a small fraction of devices and lack unified APIs. Therefore, mobile GPUs play a more and more critical role in accelerating on-device inference.

However, current DNN deployment on mobile GPUs requires huge human effort. Since different GPUs or tensor sizes prefer different kernel implementation, current general practice, such as TFLite [29] and Mace [35], is to manually optimize kernels for different settings. Unfortunately, this is not scalable considering the many GPU versions in the market and the frequent upgrade [10].

Therefore, automatic tensor kernel generation is essential for DNN deployment on mobile GPUs. Unexpectedly, although the state-of-the-art kernel generator [7] can achieve >90% peak performance on server GPUs, the performance is very poor on mobile GPUs, e.g., <10% peak performance on Adreno 640 GPU. Besides, automatic kernel generation searches for a good kernel implementation from a huge implementation space for a tensor computation. Current searching takes hours for one DNN model. This is infeasible considering the thermal throttling on mobile devices.

The goal of our work is to *rapidly generate high-performance tensor kernels on diverse mobile GPUs*. The major challenges are: (1) it is unclear what is the optimal implementation and performance bound; and (2) how to rapidly generate implementations approaching the bound from a large space of possible implementations.

The first challenge has been a long-term mystery for mobile DNN developers. Even the state-of-the-art hand-written kernels (by Mace) can only reach about 26% peak performance. None of works before can answer the questions because: (1) mobile GPUs are all black-box due to frequent upgrade, vendor customization and intellectual property protection; (2) profiling tools for mobile GPUs only report limited performance events, not adequate to expose bottlenecks; (3) kernel compilers for mobile GPUs are closed source and immaturely implemented. No low-level instructions (e.g., CUDA PTX) or binary disassembling tool (e.g., nvdisasm) [37] are provided.

To tackle this challenge, we propose a cross-platform profiling tool named *ArchProbe*, which is the first one to expose and quantify mobile GPU architecture, including register usage, the whole cache hierarchy, hardware parallelism, etc. The design overcomes specific profiling difficulties on mobile GPUs, such as uncertain compiler behavior and no fine-grained timing. It minimally assumes that the profiled device has a general SIMT (Single Instruction Multiple Threads) architecture, which allows profiling on a variety of GPUs conforming to the standard APIs.

The profiling demystifies the hardware bottlenecks that limit inference performance: *the limited number of registers* and *cache bandwidth*. It also reveals the solution for the second challenge in three aspects. (1) It exposes the unique high-performance feature, *texture cache*, on mobile GPUs, which should be used in kernels for best performance. The high-performance feature on server GPUs, i.e., local/shared memory, however, has inferior performance on mobile GPUs. The fundamental difference of server and mobile GPUs is because mobile chip vendors prefer to optimize texture access for more generalized use cases under limited cost and chip area. (2) It identifies inefficient implementations over/under-utilizing hardware resources that should be excluded to accelerate kernel generation. (3) It specifies the hardware performance bound that generated kernels can approach.

Related works on code generation [1, 7, 48, 49] lack hardware profiling and overlook the difference between mobile and server GPUs. They do not support texture cache and include all inefficient implementations in the search space, leading to poor performance.

Empowered by the profiling, we propose a mobile-GPU-specific DNN kernel compiler named *Romou*. It features two techniques that can rapidly generate high-performance kernels: (1) *the texture cache support* across the compiler stack including the user interface, compiler IR (Intermediate Representation) transformation, and kernel generation. To overcome the challenge of costly address calculation,

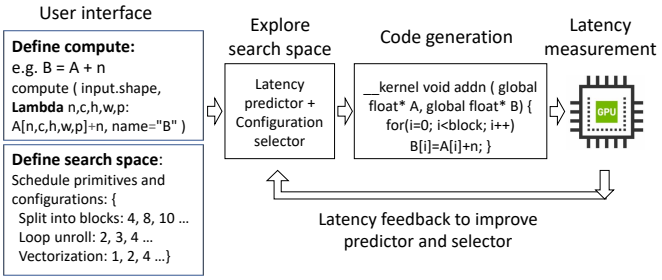

Figure 1: Flow of automatic code generation.

Romou also introduces *address-calculation elimination* as a compiling pass; (2) *hardware-aware search space pruning* that greatly prunes inefficient implementations against hardware constraints. To avoid repetitive pruning setting for each kernel, Romou integrates the pruning in the compiling process that can apply to all kernels, and abstract the hardware details from users.

For evaluation, two representative Adreno and one Mali GPUs are picked, since Adreno and Mali take 79% of the mobile GPU market [6]. Compared to the state-of-the-art DNN compiler i.e., the TVM mobile GPU backend [33], Romou achieves substantial speedup. For example, on Adreno 640, the average speedup is 14.7× for 1×1 convolution, 8.6× for 3×3 convolution, 8.1× for depth-wise convolution, and 8.1× for fully connected operator. This speedup is due to Romou's utilization of efficient hardware features on mobile GPUs. Depending on the operator hyperparameters and search space definition, Romou can reduce the search space by >90% and converge to the global optima in much reduced time.

Compared to the state-of-the-art hand-tuned kernels of Mace, the average performance improvement is 24% for 1×1, 23% for 3×3, 49% for depth-wise convolution, and 120% for fully connected. This speedup is because Romou can automatically customize implementations for each hyperparameter setting. Although theoretically hand-tuned kernels could achieve similar or better performance than Romou, it is too costly to tune the kernel by hand for each hyperparameter setting.

To sum up, the contributions of the paper include:

- Propose a profiling tool ArchProbe[1] which is the first to expose and quantify the architecture of mobile GPUs.
- Identify the high-performance hardware feature and performance bottleneck on mobile GPUs.
- Propose kernel compiler Romou for mobile GPUs which supports the use of high-performance hardware feature and prunes inefficient implementations.
- Implement Romou and achieve significant speedup compared to other auto-generated and hand-optimized kernels in much reduced searching cost.

2 Background and Motivation

2.1 Poor performance of generated kernels

Search space of kernel generation Fig. 1 shows the general flow of automatic kernel generation. DNN kernel generators normally separate the definition of tensor compute and possible implementations [41] in the user interface. They provide tensor domain-specific

[1] Open sourced code link: https://github.com/microsoft/ArchProbe

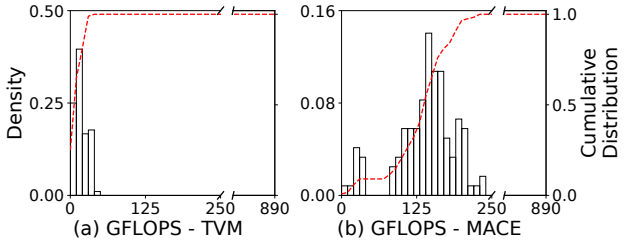

Figure 2: Performance distribution of DNN operators by (a) TVM and (b) MACE on Adreno 640.

languages that users can specify the compute expression. Implementations are described by *Schedule* which is a set of code transformations from the expression to concrete kernels. These transformations are specified by *schedule primitives*, such as block partitioning and loop unrolling. These primitives and corresponding configurations define a space of possible kernel implementations.

The space is then explored for the low-latency implementation. During the exploration process, a latency predictor is trained by measured data to avoid future measurements. The predictor is periodically updated as more configurations are measured to improve accuracy. Probabilistic methods such as simulated annealing is used to select promising configurations, jointly with the latency predictor, to quickly converge to good configurations.

Poor performance of generated kernels Fig. 2(a) shows the performance distribution of generated kernels by the state-of-the-art kernel generator—TVM (use TVM default settings for mobile GPUs [33]). The kernels are for operators such as convolution and fully connected in popular DNN models, including MobileNet, SqueezeNet, ResNet, Inception and BERT. The results illustrate that although TVM can achieve good performance on server GPUs, the kernel performance on mobile GPUs is poor, $<$10% of the theoretical peak 890 GFLOPS on the Adreno GPU. Most kernels are around 10 GFLOPS. Also, the whole tuning takes several hours to complete for one model. This tuning needs to be done for every model on every mobile GPU for the optimal performance.

2.2 Low performance of hand-written kernels

To understand the performance bottleneck of generated kernels, we first analyze hand-written kernels. Albeit faster than generated ones, the performance is still much less than the peak performance.

Performance comparison of different frameworks To find the state-of-the-art hand-written DNN kernels, we evaluate the mobile GPU backends of widely-used frameworks for on-device inference: TFLite from Google, Mace from Xiaomi, MNN [23] from Alibaba, and ncnn [45] from Tencent. Both OpenCL and Vulkan implementations are included for comparison (OpenGL is ignored due to inferior performance [30]). The convolution algorithms currently used for mobile GPUs are direct convolution and Winograd [28]. Other algorithms such as im2col+GEMM and implicit GEMM [9] are not suitable for the resource-constrained mobile GPUs. Inference frameworks can be set to use Winograd or not. We show the results with higher performance.

The comparison results in Fig. 3a tell that (1) Mace runs the fastest among these popular on-device frameworks; (2) OpenCL backends run faster than the Vulkan ones. Hence, for Mace with OpenCL, we

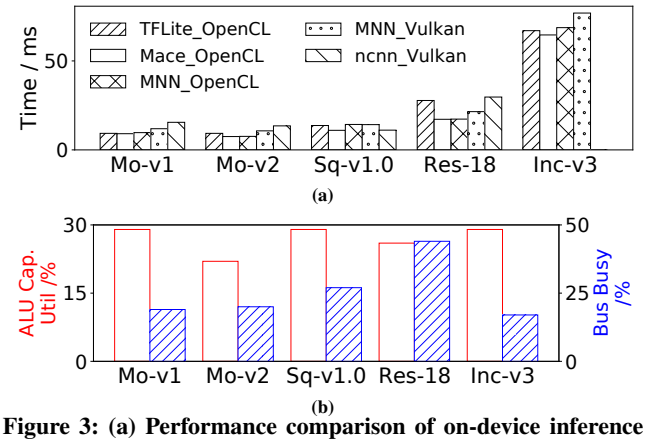

Figure 3: (a) Performance comparison of on-device inference frameworks (missing InceptionV3 with ncnn due to the lack of support). Mace with OpenCL runs faster. (b) % of ALU capacity utilization (blank bar, left axis) and % of time that GPU to global memory bus is busy (slash bar, right axis) by Mace. The GPU is Adreno 640. "Mo/Sq/Res/Inc" is short for MobileNet/SqueezeNet/ResNet/Inception respectively.

further evaluate its kernel performance distribution in Fig. 2(b) for the same operators in Fig. 2(a) as comparison. The highest kernel performance in Mace is 232 GFLOPS. Most of the kernels are around 130 to 170 GFLOPS. Albeit higher than generated ones, *hand-written kernels also run much slower than the peak performance* ($<$26% on Adreno 640).

Limited profiling from available tool We then analyze the performance issue by the available tool for Adreno GPU—Snapdragon profiler [40]. From the results of limited events supported as shown in Fig. 3b, we find that *both the ALU capacity and the GPU bus to global memory are underutilized*. The % of ALU capacity means the average % of all ALUs utilized in each cycle. That means, $>$70% ALUs are wasted for DNN inference. Papers before [14, 20] simply attribute this issue to limited memory bandwidth. However, this cannot be true because if so, the GPU bus to global memory should be highly busy rather than idle.

In summary, current DNN inference performance on mobile GPUs is very poor. To understand the bottleneck requires more in-depth profiling beyond available profilers. This motivates us to develop more comprehensive profilers.

2.3 Background: programming paradigm

There are various terminologies from different GPU APIs, although the meanings are similar. To avoid confusion, this section introduces the concepts and terminology of OpenCL used in this paper, since OpenCL shows higher inference performance on mobile GPUs. Note particularly the differences between OpenCL and CUDA.

Execution model GPU programming uses the typical SIMT (Single Instruction Multiple Threads) execution model. Users only need to write the *kernel* for one thread and specify a *work group* size. Then, the OpenCL and GPU scheduler are in charge of running the kernel parallelly on the GPU. OpenCL first decomposes the data index space into work groups (i.e., *blocks* in CUDA) according to the work group size. Take a 1×1 convolution as an example in Fig. 4,

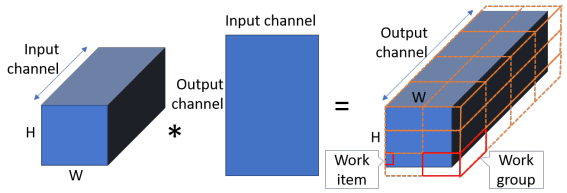

Figure 4: Work group and work item for a 1×1 convolution.

the index space is the output tensor (the space needs to be padded if not divisible by the work group size). Each work group is assigned to run on a GPU core called a *shader core* (or a *shader processor* or *stream multiprocessor*).

A work group is composed of *work items*. A work item is a software thread and mapped to a hardware thread to run (i.e., an ALU in a shader core). A *warp* is a group of hardware threads that run the same instruction simultaneously. It is the basic execution unit in a core. Thus, the work items of a work group will be partitioned to run in warps.

Memory model To potentially speed up data accesses, there are four types of memory regions / address spaces in OpenCL programming: *global memory* and *constant memory* shared by all work groups, *local memory* shared by the work items of a work group (i.e., *shared memory* in CUDA), *private memory* for each thread. How to support these regions in hardware depends on the GPU manufacturers. For example, on the Adreno GPU, local, constant, and private memory are implemented on the GPU chip [39]. However, on the Mali GPU, the regions are all in system DRAM [3].

The data in global memory can be stored in *image* or *buffer* types. The buffer is similar as the CPU buffer which stores data continuously. The image type in OpenCL, similar as other image formats supported by other GPU APIs, is designed to speed up graphics rendering. An image object can be declared as one, two or three dimensions. Each element of an image object has to be in four channels i.e., RGBA. To use image type requires data layout aligned with this format.

Normally GPUs have special hardware support for image, called *texture cache or memory* [12]. This storage is designed for rapid random accesses, and thus separated from normal cache designed for continuous accesses.

3 Performance Bottleneck Demystification

To explore the bottleneck and design optimal kernels for various black-box GPUs, we develop a cross-platform micro-benchmark toolkit called *ArchProbe*, to disclose and quantify performance-vital hardware features.

3.1 Hardware feature quantification

Challenges Accurate quantification relies on *predictable hardware behaviour* of micro-benchmark kernels and *high-resolution timing*. However, it is much harder for mobile GPUs to achieve the two requirements than server GPUs, mainly due to two challenges.

Firstly, the language compilers are closed source. Mobile GPUs can only be programmed in high-level languages such as OpenCL, with no public low-level instructions. What the compiler finally generates is unknown. Therefore, ArchProbe is designed carefully to avoid compiler optimizations that could lead to unexpected behaviour. Besides, without low-level instructions, the information that could be easily read from the assembly such as the number of registers has to be profiled.

Secondly, there is no user-level accurate timing. Rather than the support of cycle-level and flexible timing on server GPUs, current mobile GPUs do not expose interface for accurate and fine-grain timing. The timing function cannot be inserted inside a kernel or give accurate cycles. ArchProbe has to be designed, such as using the law of large numbers, to reduce the timing error.

Due to these challenges, none of mobile GPU profiling before [2, 5, 22] can go deeply into the architecture, nor explain the performance bottleneck. By comparison, ArchProbe discloses the following key architecture features. We will next introduce the rationale to pick these features and the ArchProbe design idea. (Unless specifically stated, this section will use Adreno 640 result as an example.)

- The maximum number of registers for each work item
- The size, cacheline size, and bandwidth of memory hierarchy including all levels of unified and texture caches, local, constant and global memory
- The number of threads in a warp
- The number of ALUs in a shader core

The number of registers Register usage is one of the most performance-critical design for a kernel. Given that registers are the fastest data storage, increasing register usage can reduce data access cost and improve performance. However, registers are also used for context switching of concurrent warps to hide execution stalls. Therefore, increasing register usage could also reduce concurrent warps and hurt performance. Besides, overuse of registers in a kernel will cause register spill to memory and greatly reduce performance.

Therefore, it is essential to detect the maximum available registers and the sharing mechanism. Our method is to gradually increase the number of used registers in a kernel and work items. The inflection points of kernel latency will show these parameters. The code is shown in Listing 1. It generates kernels for varying register usage. The code is carefully designed to avoid potential compiler optimizations (Line 22) as well as unrelated costs, such as extra memory accesses (Line 20).

The profiled results of Adreno 640 in Fig. 5 show two conclusions. (1) *The detected 32bit-register file size is 384×181.* The latency increases dramatically after the kernel uses 181 registers. (2) *The registers are shared among concurrent work items.* When the number of work items is no more than 384, the latency lines overlap each other, which means all the work items can run concurrently. As the number increases, take 768 as an example (orange line), all the work items can run concurrently only when the register usage is less than half of 181 registers in the kernel. Otherwise, the latency is doubled. That means the 768 work items have to run in two sequential 384-sized groups, constrained by the available registers. The same conclusion also applies for 512 and 1024 work items.

Memory hierarchy The measured parameters of the memory hierarchy can direct data layout and placement to leverage the locality and bandwidth.

To profile cache parameters, our technique is to extend the classical pointer-chase method [37]. By using image and buffer types (refer to Sec. 2.3), we can measure the texture and unified cache

```
for (nWorkItem = 1; nWorkItem<maxLogicalThread; nWorkItem+=step)
  for (nReg = 0; nReg < threshold; nReg++)
    runKernel(reg_count, (nWorkItems,1,1)/*work group size*/,
              1/*total work groups*/, nReg, clEventTimer);

/* Generate kernel codes for different nReg */
for (int i = 0; i < nReg; ++i){
  reg_declare += format("float reg_data", i, " = ", i, ";\n");
  reg_comp += format("reg_data", i, " *= reg_data",
                     i==0? nReg-1: i-1, ";\n");
  save_to_mem += format("out_buf[", i, " * i] = reg_data",
                        i, ";\n");}

auto src = format(R"(
__kernel void reg_count(__global float* out_buf) {
  )", reg_declare, R"(
  int i = 0;
  for (; i < N; ++i) { /*run N times to reduce timing error*/
  )", reg_comp, R"( }
  i = i >> 31; /* make output buffer index a variable */
  /*save results to memory in case of dead code elimination*/
  )", save_to_mem, R"( } )");
```

Listing 1: The code to detect the number of registers.

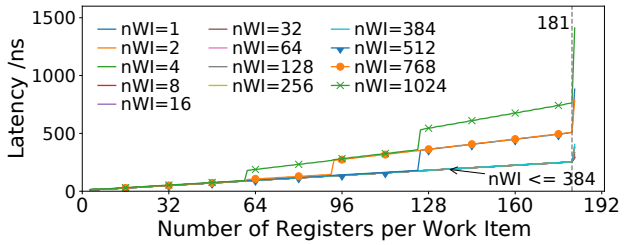

Figure 5: The latency response to the increase of registers and work items on Adreno 640 ("nWI" is the number of work items). It shows a 384×181 shared 32b-register pool.

accordingly. The pointer-chase algorithm is to traverse an array of which an element is initialized as the index of the consecutively-accessed element. As the size of the array and the stride of two consecutive accesses increase, the cache parameters such as access latency and cache size, can be detected from the latency inflection point. Listing 2 shows the algorithm for the image type.

```
/*Array initialization. Buffer type is needed in the CPU side.*/
int* idx_buf = mapImageToBuffer(src_image);
for (size_t i = 0; i < dataRange; i++)
  idx_buf[i] = (i + stride) % dataRange;
src_img = unmapImage(idx_buf);

/* Work group size (1, 1, 1), one work group */
__kernel void image_cache(__read_only image1d_t src,__global int* dst)
{ int idx = 0;
  for (int i = 0; i < N; ++i)
    idx = read_imagei(src, SAMPLER, idx).x;
  *dst = idx; }
```

Listing 2: The code to profile cache hierarchy (image type).

Fig. 6 shows the average accessed latency for a truncated sample (128 to 384×4 B) of the whole profiled data range. The parameters on Adreno 640 exposed from the figure are: (1) *L1 texture cache size is 256×4 B*. The data access latency increases greatly when the size of the array is > 256; (2) *The cacheline size of L1 texture cache is 32 B*, since the latency lines after stride=8 are overlapped (red and purple). The flat latency after 264 also illustrates the different

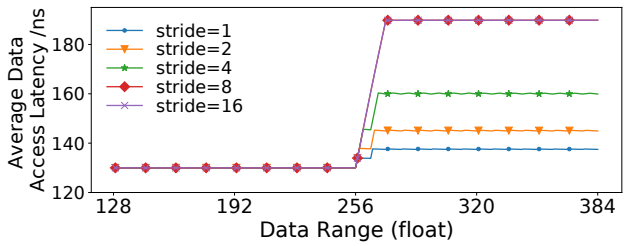

Figure 6: The average access latency of different data range and stride for image type on Adreno 640. It shows 1 KB L1 texture cache and 32 B cacheline size.

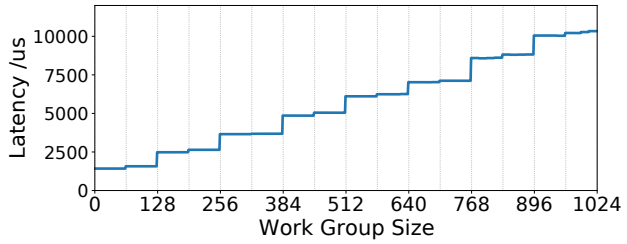

Figure 7: The latency response to the increasing work items shows 64 or 128 warp size on Adreno 640.

behaviour of texture cache from normal cache (normal cache would have increasing latency between 264 and 384).

For cache bandwidth, the ArchProbe kernel is designed to make work items sequentially access the data range of a cache for many iterations. The bandwidth is then calculated by dividing latency into the size of total accessed data.

Warp size A warp is the basic execution unit of a core. The selection of work group size has to consider the warp size to utilize all the simultaneous threads. To detect it, our idea is to run enough work groups (e.g., 1024) to potentially saturate all the ALUs, and then gradually increase the work items in each work group. The latency response can thus show the warp size. We utilize the cycle-consuming division operation as the kernel body to avoid latency hiding by warp switch.

The measured latency in Fig. 7 demonstrates that on Adreno 640 *the warp size is 64 or 128*. When the work group size is between 1 to 64, the latency keeps the same although the total work items increase. Between 65 to 128, there is a slight latency increase, but much smaller than >128. Therefore, depending on the total work items of a task, the work group size is preferred to be a multiple of 64 or 128 to achieve full GPU utilization.

```
/*Pick group size (64,6,2) as an example, one work group*/
__kernel void warp_size (__global int* output) {
  __local int local_counter;
  local_counter = 0;
  barrier(CLK_LOCAL_MEM_FENCE); /*sync all work items*/
  int i = atomic_inc(&local_counter);
  output[globalID0 + globalID1*globalSize0 +
         globalID2*globalSize0*globalSize1] = i;}
```

Listing 3: Detect logical and physical thread mapping.

The number of ALUs To detect how many threads can run parallelly, our design is to let every work item atomically increase a shared counter and then store the counter value in the entry of the work-item ID in the result array. The array will save the running

Warp0	ItemID	(0,0,0)	(1,0,0)	...	(63,0,0)	(0,1,0)	...	(63,1,0)
	Output	0	4	...	192	195	...	388
Warp1	ItemID	(0,2,0)	(1,2,0)	...	(63,2,0)	(0,3,0)	...	(63,3,0)
	Output	1	3	...	188	191	...	378
Warp2	ItemID	(0,4,0)	(1,4,0)	...	(63,4,0)	(0,5,0)	...	(63,5,0)
	Output	2	5	...	190	193	...	380

Figure 8: The running order illustrates 384 ALUs on a core of Adreno 640.

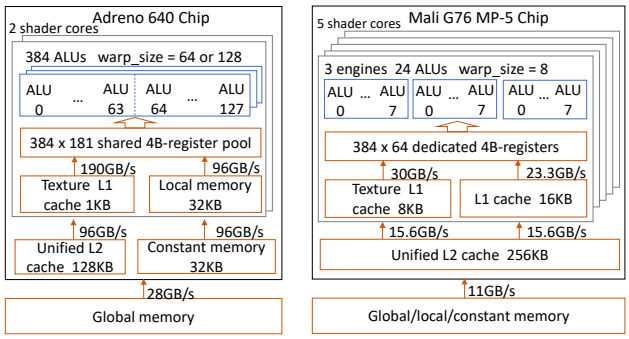

Figure 9: Measured architecture of (a) Adreno 640 and (b) Mali G76 MP-5 (MP-5 means five cores).

order of the work items, and show the number of ALUs. The code is in Listing 3.

The running order of the work items in Fig. 8 shows that they are assigned to three parallel warps and executed consecutively with an interval of the warp size. Therefore, there are totally 384 ALUs on a core of Adreno 640 (warp 0 ends in 388 rather than 383 due to the start of another two warps).

Overall profiled architecture The overall results are shown in Fig. 9 for the two dominant mobile GPU series: Adreno and Mali. There are two outstanding conclusions from the figure. (1) Adreno and Mali have quite different architecture. For example, Mali has dedicated registers for each work item, while Adreno has a shared pool. The warp of Mali is much more fine-grained than Adreno. The cache and memory bandwidth of Adreno 640 is much superior to Mali G76. (2) Mobile GPUs have distinct features compared to server GPUs. A critical one is that for server GPUs, local/shared memory is a fast on-chip storage with similar bandwidth as L1 cache. However, on mobile GPUs, L1 texture cache is much faster than local memory. We find that using local memory for convolution only achieves half of the performance using texture cache.

These conclusions justify the necessity of ArchProbe profiling for each GPU. The results can direct the design of kernel generation.

3.2 Performance bottleneck analysis

Leveraging the profiled results and classical roofline model [46], we can identify the performance bottleneck on mobile GPUs: *the limited number of registers and cache bandwidth.* The number of registers determines the data reuse rate i.e., how frequently data is loaded from cache, while cache bandwidth decides how fast data is loaded from cache. The two limitations make ALUs wait for the data and cannot be fully utilized on mobile GPUs.

Take Adreno 640 as an example to elaborate the limitations. As shown in Fig. 5, the highest GPU performance is achieved when the size of the work group is 1024 and the number of used registers is <60. In this setting, since the registers are enough to support warp switching, all the 1024 work items can run concurrently in the same latency as only 1 work item, i.e., 1024× performance. When the number of used registers is more than 60, due to no enough registers for warp switching, the latency is doubled and performance is halved. Similarly, on Mali G76, the performance peaks at 240 work-group size (albeit only 120 ALUs) with 64 registers for each item.

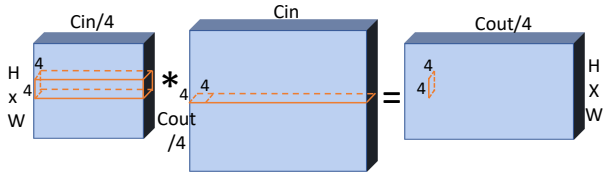

Figure 10: An example implementation for 1×1 convolution with coarsening size 4×4 (orange) in 2D image.

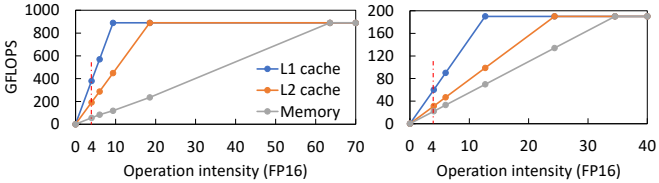

Figure 11: Roofline models for Adreno 640 and Mali G76.

This register limitation greatly constrains the data reuse (i.e., operation intensity in the roofline model). Take an implementation for 1×1 convolution in Fig. 10 as an example. To utilize the fast texture cache, the tensor layout has to be packed into four channels to align with image format. The packing is normally done on the convolution channel dimension i.e., Cin and Cout in the figure. Based on this layout and the register limitation at max GPU performance, the kernel coarsening can be 4×4 (i.e., a work item computes a 4×4 basic block). In this case, a loop iteration in the kernel conducts a (4,4)×(4,4)->(4,4) sub-matrix multiplication. The number of registers used is 48 (=3×4×4). The operation intensity is 4 (=4×4×4×2/(4×4+4×4)).

The roofline models in Fig. 11 can show the performance bound at various operation intensity. When the operation intensity is 4 (marked by the red line), the performance is bounded by data accessing, and much below the computation bandwidth. Even assuming all data could be held in L1 cache, the performance bound is only 380 GFLOPS. There might be opportunities to increase kernel coarsening size, e.g., to 4×6, to increase operation intensity and raise the bound. However, this requires careful tuning since except for the sub-matrix multiplication, other operations in the kernel also compete for registers.

The analysis above demystifies the performance gap of DNN inference from the theoretical GPU peak performance. It clearly directs hardware designers that to improve DNN inference performance on mobile GPUs, increasing registers or cache bandwidth is the key rather than increasing ALUs.

3.3 Implications for optimal kernel generation

The profiling results direct the high-performance kernel generation in three aspects.

Firstly, the results expose the fast hardware feature i.e., texture cache that kernel generation should utilize for best performance. Due

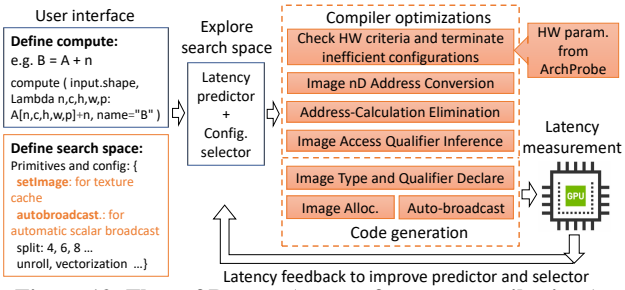

Figure 12: Flow of Romou (orange for new contributions).

to the lack of hardware analysis, current kernel generators [7, 48, 49] fail to utilize texture cache, resulting in poor performance.

Secondly, the results identify inefficient implementations against hardware constraints that can be excluded during kernel generation. For example, implementations that overuse the available registers can be excluded directly. By comparison, current kernel generators search for the efficient implementation in a huge space filled with inefficient implementations. This potentially leads to three major issues. (1) Global optima cannot be found. The latency distribution of a large inefficient space can be hard to learn by the latency predictor (refer to Fig. 1). The space exploration directed by the predictor may converge at local optima rather than the global one. (2) Long time is taken before converging to the optima. With a large search space, the possibility to find optima by the probabilistic explorer is diluted (evaluation in Sec. 5.3). (3) On mobile GPUs, long-time searching can cause performance slowdown due to thermal throttling, which misleads the searching result.

Thirdly, the results can specify the performance bound of mobile GPUs that kernels can approach. As Sec. 5.3 will show, the performance of Romou's generated kernels is approaching the bound.

4 Kernel Compiler for Mobile GPUs

Directed by the profiling, we propose high-performance kernel compiler *Romou* for mobile GPUs. It empowers the kernel to use the unique hardware features and excludes the inefficient implementations from the searching. Romou can thus rapidly generate optimal kernels for mobile GPUs.

Support unique features Texture cache is the fastest data storage. To empower kernels to use it requires support across the compiler stack, including the schedule user interface, compiler IR transformation, and kernel generation. Fig. 12 shows the flow of Romou. In the schedule interface, Romou introduces new primitive: `setImage` for texture cache, since image is the data type provided by the programming API to use texture cache.

However, the challenge of using image type is that it requires additional address conversion from general linear addressing to the nD-4channel addressing of image. The address calculation is costly that can even offset the gain by using image. To solve that, Romou introduces a compiler optimization pass Address-Calculation Elimination (ACE) to greatly reduce the calculations.

Another hardware feature is the automatic scalar broadcast supported in recent scalar-based mobile GPUs. Relative to the vector-based mobile GPUs, automatic scalar broadcast allows kernels to conduct scalar-vector computation directly without explicit scalar-vector conversion beforehand. Romou introduces another new primitive `autobroadcast` to utilize this feature.

Exclude inefficient implementations With the profiled hardware parameter, Romou can exclude implementations that over/under-use the hardware resources such as caches and registers. A straightforward way is to properly set the primitive configurations in the user interface, such as the split size shown in Fig. 12. However, the challenge is that the configurations have to be tuned for every kernel, and it unnecessarily exposes hardware parameters to users.

To this end, Romou introduces a pruning pass during compiling to exclude inefficient implementations. The advantages of this method are that (1) the pruning can be abstracted from users and applied to all kernel generation; (2) more inefficient implementations can be pruned since compiler IR provides more information than user interface. For example, the compiler can easily count the total register usage of a kernel from the variable allocation nodes in the AST (Abstract Syntax Tree).

The pruning process is that after a candidate implementation is selected by the space explorer, Romou first checks if the implementation violates the pre-defined hardware criteria. If so, the following processes, including code generation and profiling, are terminated. Romou then goes back to the space explorer to select the next implementation. For implementations conforming to the hardware criteria, Romou conducts the following IR transformations to generate kernels and run on the hardware. This method can greatly reduce the searching cost (see Sec. 5.3).

4.1 Compiling support for new primitives

End-to-end kernel generation Besides Fig. 12, Listing 4 uses a simple tensor-scalar compute as an example, to show the end-to-end process from user compute definition to a generated OpenCL kernel.

With the `setImage` primitive, each tensor can be set to use image type or not (Line 4, 12) in the configuration. If image is used, three more passes will be done in the IR level as shown in Fig. 12. (1) Address conversion. The compiler has to map the linear address to multi-dimensional address (e.g., 2D) of an image type. (2) Common address-calculation elimination (ACE). Address conversion introduces costly calculation such as division. Romou reduces the cost by eliminating common sub-expressions of address calculation. (3) Access qualifier inference. An access qualifier, such as `read_only` (Line 31), has to be specified for an image-typed argument of a kernel. The qualifier is to direct data placement optimizations. Romou employs a compiler pass to infer read or write access for each tensor to decide the qualifier.

After IR passes, the related codes for image type is generated, including the declarations of image type and access qualifier (Line 31, 32), as well as image operations such as `read/write_imagef` (Line 33). Image declaration and allocation run on the CPU (Line24-29). The kernel runs on the GPU (Line31-36).

If the primitive `autobroadcast` is set (Line 13), no explicit scalar-vector conversion will be generated in the kernel. For example in Line 38, the scalar `n` and the 4-element vector from `A` can be added directly. This applies to the recent scalar-based mobile GPUs with hardware scalar broadcast, such as the Bifrost and Valhall architecture of Mali [4]. If `autobroadcast` is not set, an instruction `vstore4(float4(n,n,n,n),vec_n)` will be generated before Line 38 to create a vector of `n` first. Then, the two vectors can be calculated. This applies to the traditional vector-based mobile GPUs. Without this knowledge, current compilers generate explicit vector

```
# Operator developer interface
def addConstant(cfg, input:Tensor, n:int)->Tensor:
  B = compute(input.shape, lambda n,c,h,w,p:A[n,c,h,w,p]+n, name="B")
  ↪  # p is packing, p=4 for image type
  cfg.define_knob('set_input_image',[True,False])
  cfg.define_knob('autobroadcast',[True,False])
  ... # Define other configurations
  return B

def scheduleAddConstant(cfg, B:Tensor):
  input, = B.op.input_tensors
  s = create_scheule(B.op)
  if cfg['set_input_image'].val == True: s[B].setImage(input)
  if cfg['autobroadcast'].val == True: s[B].autobroadcast(B)
  ... # Define Other primitives
  return s

# Generated IR
# For each element in B
B[@ir._2d_coord(linearIndex,@ir.imageWidth(B),dtype=int32)]
  = ((imgwfloat32*)A[@ir._2d_coord(linearIndex,
  @ir.imageWidth(A),dtype=int32)]+n)}}

# Generated OpenCL kernel
cl_image_format fmt={CL_RGBA, CL_HALF_FLOAT};
# w*c/p is image width, h is image height
cl_image_desc desc={CL_MEM_OBJECT_IMAGE2D, w*c/p, h};
cl_mem A=clCreateImage(context, CL_MEM_READ, &fmt, &desc);
cl_mem B=clCreateImage(context, CL_MEM_WRITE, &fmt, &desc);
__constant sampler_t sampler = TEXTURE_CONFIG;

__kernel void addConstant(__read_only image2d_t A,
                    __write_only image2d_t B) {
  write_imagef(B,(int2)(linearIndex%(p*get_image_width(B))/p,
    linearIndex/(p*get_image_width(B))), read_imagef(A,
    sampler, (int2)(linearIndex%(p*get_image_width(A))/p,
    linearIndex/(p*get_image_width(A)))) + n);}
```

Listing 4: An end-to-end code generation example for a tensor-scalar (B=A+n) calculation. The IR and kernel snippets are for one implementation in the search space.

conversion for all mobile GPUs, leading to poor performance. As we will show in Sec. 5.3, using `autobroadcast` can result in 4.9× speedup on Adreno 640.

Address calculation elimination Compared to the simple linear addressing for buffer type, the challenge brought by image type is the big cost of address calculation. As introduced in Sec. 2.3, image type uses the multi-dimension and RGBA four-channel format (1D image is not used since its addressing range is normally not enough for a DNN tensor). If `setImage` is true, the tensor layout has to be transformed to follow this format. The data address also has to be calculated accordingly from the linear index (Line 35-38).

Unlike server GPUs, the image address calculation is very costly on mobile GPUs. It can even offset the performance gain from using image type. The big cost is due to two reasons: (1) division and modular is time-consuming; (2) some language compilers such as the OpenCL compiler for mobile GPUs are unable to optimize kernel code sufficiently [26].

Therefore, Romou adds a compiler pass to conduct address calculation elimination, as shown in Algorithm 1, to generate kernels without repeated calculation. While traversing the AST of a kernel, the `load` and `store` nodes in the `for` loop are processed to rewrite common address-calculation expressions as new variables (Line 23). After all the expressions are processed, the declaration nodes of these new variables are added to the AST (Line 19).

For each address calculation expression, the algorithm simplifies it, and scans all its sub-expressions in the order of the longest to the shortest (Line 2) so that the longest sub-expression can be matched first. If a sub-expression has appeared before, it is replaced by the given variable (Line 5). Otherwise, this sub-expression and a new variable are inserted in the expression-variable map (Line 9).

Algorithm 1 Common address calculation elimination

Input: AST (Abstract Syntax Tree) of a kernel
Output: AST with common address calculation eliminated

```
1:  function RewriteCommSubExpr(node)
2:      reversely add subExpr in node.addrExpr to exprList
3:      for subExpr ∈ exprList do
4:          if exprVarMap[subExpr] then
5:              replace(node.addrExpr,exprVarMap[subExpr])
6:              return
7:          end if
8:      end for
9:      exprVarMap.insert(node.addrExpr, newVar)
10:     replace(node.addrExpr,newVar)
11: end function
12: function TraverseAST(node)
13:     if node.type==forNode then
14:         enterForNode ← enterForNode+1
15:         TraverseAST(node)
16:         enterForNode ← enterForNode-1
17:         for subExpr ∈ exprVarMap do
18:             ▷ exprVarMap is the <expression,variable> map.
19:             add the declaration node for exprVarMap[subExpr]
20:             exprVarMap.delete(subExpr)
21:         end for
22:     else if enterForNode and (node.type==loadNode or storeNode) then
23:         RewriteCommSubExpr(node)
24:     end if
25:     TraverseAST(node.next)
26: end function
```

```
for (int i; i < coarsening_size - 1; i++) {
  write_imagef(B, int2((linearIndex+i*2*p)%
    (p*get_image_width(B))/p, (linearIndex+i*2*p)/
    (p*get_image_width(B))), B_local+i*2);
  write_imagef(B, int2((linearIndex+(i*2+1)*p)%
    (p*get_image_width(B))/p, (linearIndex+(i*2+1)*p)/
    (p*get_image_width(B))), B_local+(i*2+1)); };
```

```
const int comm1=linearIndex/(p*get_image_width(B));
const int comm2=linearIndex%(p*get_image_width(B))/p;
for (int i; i < coarsening_size - 1; i++) {
  write_imagef(B, int2(comm2+i*2, comm1),B_local+i*2);
  write_imagef(B, int2(comm2+i*2+1, comm1),B_local+i*2+1);}
```

Listing 5: The generated kernel (write B_local to B) before and after common address calculation elimination.

Listing 5 uses an example to compare the generated kernel before and after Algorithm 1. After the elimination, the common address calculation is conducted only once during variable initialization. This technique can greatly improve performance by 6× on Adreno GPUs (see Sec. 5.3).

4.2 Hardware-aware search space pruning

Romou leverages the profiled hardware parameters to prune inefficient kernel implementations in the search space. With the hardware parameters, a direct thought is to formulate an analytical latency prediction model so that the online training cost for the latency predictor can be eliminated. We followed this thought at first. However, we

Table 1: Space pruning criteria

HW feature	kernel exclusion criteria
L1 cache	Access more data in one loop iteration than L1 cache
register	Overuse available registers for the work group size
buffer	Use buffer type when L1 cache is for texture only
local memory	Use local memory when work group size $> \alpha \cdot$ ALUs
warp	Work group size $<$ warp size
data access width	Use inefficient data access width

find that the analytical model is reasonably accurate for handwritten kernels, but not for compiler-generated ones. The reason is that the automatically generated kernels can invoke some unpredictable cost.

To this end, Romou inputs the hardware parameters and the kernel's hardware utilization to the predefined criteria, to prune the kernel implementations that over/under-use the hardware. The kernel's hardware utilization is calculated during compiling by visiting the nodes of the kernel AST. Table 1 lists the major criteria. We will next explain each of them.

For L1 cache, since all the work items in a work group run concurrently on a shader core, it is important to make sure the loaded data of all the work items in one loop iteration in the fast L1 cache. By visiting the memory access nodes of the kernel AST, we can get the size of loaded data of one work item in one loop iteration. By multiplying it with the work group size read from attribute nodes, Romou can calculate the total loaded data. If it is larger than L1 cache size, this implementation will be abandoned. Similarly, for register, the pruning pass traverses all the allocation nodes of the AST, to check whether the total register usage of the kernel is over the available ones for the work group size (refer to Fig. 5).

For local memory, since it is slower than cache and the synchronization is costly on mobile GPUs, the use of it is excluded if the number of work items (i.e., work group size) is enough for the number of ALUs (α is an experimental threshold. 3 is set in this paper). Or, Romou will split the reduction axis and use local memory to increase parallelism. Since a warp is the basic execution unit in a core, Romou excludes the ones use smaller work group size than warp size. For vector length, the pruning pass checks whether the data loading of the implementation is in the vector width that achieves the best cache bandwidth (the width is four for mobile GPUs).

How much the searching cost can be reduced from pruning depends on the operator hyper-parameters, the primitives and configurations set by users, and the smart walk of the search space explorer. For example, it is easier to find good implementations for factorable tensor size e.g., 256×256 than prime numbers e.g., 17×17. In the example analyzed in Sec. 5.3, our method reduces the space to 0.6% of the default space, and also avoids converging at local optima.

5 Implementation and Evaluation

5.1 Implementation

ArchProbe is implemented as an independent tool. It can be compiled and run directly on a mobile phone to output hardware parameters. It is implemented from scratch in C++ and employs template-based kernel generation method to flexibly generate OpenCL kernels with different configurations (refer to Listing 1). The interface with OpenCL is from Khronos OpenCL SDK [24]. The evaluation results are recorded in a JSON file, as the input for Romou space pruning.

Romou is implemented based on TVM, which is the state-of-the-art DNN compiler. To use it also facilitates Romou to be consolidated under one coherent code generation effort in the whole DNN community. The user interface is the same as TVM. Users can define the computation and schedules. The generated kernels can then be compiled and run on a mobile phone. The optimization passes of Romou is developed on TVM's Tensor IR (TIR). The code generation is implemented in TVM's OpenCL backend.

5.2 Experimental methodology

For experimental hardware, we select three widely-used mobile GPUs in the market: Adreno 630, Adreno 640, and Mali G76 shown in Table 2. The profiled hardware parameters for Adreno 640 and Mali G76 have been shown in Fig. 9. The parameters for Adreno 630 are similar as 640, except that on 630, the number of ALUs is 256 per core; the register file is 256×181 accordingly; the bandwidth of L1 cache is 145 GB/s, and the bandwidth of the other on-chip storage (including local memory, unified L2 cache, and constant memory) is 79 GB/s.

For comparison of current DNN inference backends for mobile GPUs, the frameworks and versions used are: TFLite r2.1, Mace v0.13.0-62-g63feaf5, MNN v1.1.0, ncnn 20191113 tag, and TVM 0.8.dev0. TFLite, Mace and MNN are compiled for Android with android-ndk-r18b, ncnn with android-ndk-r19c, and TVM with android-ndk-r22. The reported latency is the arithmetic mean of 20 runs (first run is excluded). The latency for code generation is the searched result. Mace and MNN use FP32 for computation, and FP16 for storage. TFLite uses FP16 for both computation and storage. ncnn uses FP16 for packing and computation.

Performance evaluation is conducted for all the convolution operators of four selected CNN models. These operators dominate the inference latency. The four models are: SqueezeNetV1.0 [21] and MobileNetV1 [18] as light-weight models for mobile applications, ResNet-18 [17] and InceptionV3 [43] as large models. All use default input size and NN structure. We also evaluate 2 fully-connected (FC) operators from BERT [11]. The input shapes are [(1,768),(768, 3072)] and [(1,3072),(3072,768)]. We pick two major operators for Bert because its whole model inference is not supported by Mace or TVM on mobile GPUs. There are totally 96 unique configurations among these operators, with different types (convolution, depth-wise convolution, and FC) and hyper-parameters (i.e., input and filter size, the number of channels and stride). Out of them, 45 are 1×1 convolution, 25 are 3×3 convolution, 9 are 3×3 depth-wise convolution, 2 are FC, and 15 other (e.g., 7×7 and 3×1) convolution.

The two comparison baselines are the mobile GPU backends of Mace and TVM, as the state-of-the-art hand-optimized and automatic-generated kernels respectively. Mace is set to use all the default options (e.g., NHWC input format and no tuning) except for enabling Winograd algorithm for 3×3 convolution since it greatly improves performance than direct convolution. For TVM, we use its default search space, search method (XGBoost), and search threshold (1000 steps) for mobile GPUs [33].

5.3 Results

This section will show performance improvement by Romou for various operators, as well as speedup breakdown for compiler optimizations and reduced kernel searching cost.

Table 2: Experimental mobile GPU specs

Mobile GPU	Adreno 630	Adreno 640	Mali G76
Phone	Google Pixel 3XL	Google Pixel 4XL	Vivo X30
SoC	Snapdragon 845	Snapdragon 855	Exynos 980
the number of cores	2	2	5
the number of total ALUs	512	768	120
Frequency	710 MHz	585 MHz	800 MHz
Computation bandwidth	720 GFLOPS	890 GFLOPS	190 GFLOPS
DRAM bandwidth	26 GB/s	28 GB/s	11 GB/s

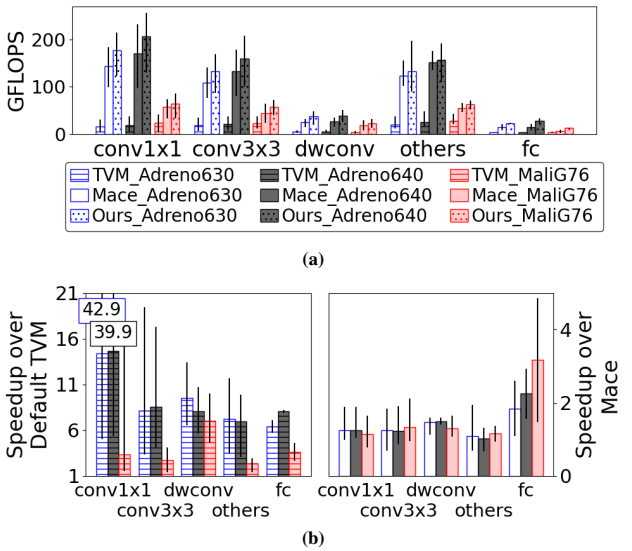

Figure 13: The average, max, and min (a) performance and (b) corresponding speedup for all the evaluated operators by Romou compared to TVM and Mace.

Speedup over generated kernels Fig. 13 shows the GFLOPS and the corresponding speedup by Romou for all our evaluated operators grouped into five categories. Height of the bar is the average speedup, while the black line shows the speedup range of all the operators in a category (not standard deviation). Compared to TVM on Adreno (Fig. 13b left), the [average, max, min] speedup by Romou for each category are [14.7, 39.9, 5.4] for 1×1, [8.6, 17.3, 4.1] for 3×3, [8.1, 10.7, 5.7] for depth-wise, [7.0, 9.8, 3.1] for other convolutions, and [8.1, 8.23, 7.87] for fully connected operators on Adreno 640 and similarly on Adreno 630. Romou can achieve higher GFLOPS (Fig. 13a) and speedup for 1×1 convolution. This is due to the higher operation intensity of 1×1 than 3×3 Winograd and depth-wise convolution, as papers [28, 34, 44] have discussed before. Winograd algorithm improves performance by reducing the number of multiplication but at the cost of extra data accesses for tensor transformation. Depth-wise removes the data reuse of input channels since each filter only applies on one input channel. Lower operation intensity restrains the peak GFLOPS and speedup (refer to the roofline model in Fig. 11).

Compared to TVM on Mali, the speedup is [3.3, 18.1, 1.6] for 1×1, [2.7, 4.1, 1.4] for 3×3, [7.1, 10.0, 4.7] for depth-wise, [2.4, 3.0, 1.4] for other convolution, and [3.6, 4.6, 2.7] for fully connected operator. Overall, the performance and speedup by Romou on Adreno is higher than Mali mainly due to two reasons. (1) The potential gain from using texture cache on Adreno is much higher. On Mali, texture and normal cache has similar bandwidth (30 vs 23.3 GB/s), while on Adreno, texture cache has twice bandwidth than normal cache (190 vs 96 GB/s). (2) TVM has tuned kernels for Mali GPUs [33] and there could be special adaptions. However, clearly these adaptions are not portable to Adreno GPUs.

Speedup over hand-optimized kernels Even compared to the state-of-the-art hand-optimized kernels in Mace, Romou can mostly achieve obvious performance improvement shown in Fig. 13b right axis. On Adreno 640, the average performance improvement is 24% for 1×1, 23% for 3×3, 49% for depth-wise, 3% for other convolution, and 2.2× speedup for fully connected operator. Adreno 630 and Mali G76 achieve similar improvement. The improvement is mainly because Romou can rapidly search more promising configurations to generate optimal kernels. There is little space for further improvement due to the register and cache bottlenecks on mobile GPUs discussed in Sec. 3.2.

The outstanding speedups of Romou for FC operators compared to Mace is because the hand-written kernels of Mace does not split the reduction axis of the tensor. For the matrix-vector multiplication of FC, the number of work items is thus not enough to fill the ALUs. Romou splits the reduction axis and by utilizing local memory, one work item can aggregate the results. The method can increase the number of work items and improve the ALU utilization as well as performance. It again shows that it is hard for hand-written kernels to optimize each case, while the kernel generation can.

Out of the 96 unique operators, around 8 operators generated by Romou runs slower than Mace. This is an issue of current code-generation technique compared to hand-written kernels. These slow operators normally have a size of a prime number. For example, the worst case by Romou is 33% slower than Mace for a 7×1 convolution with output size height=width=17. To utilize GPU parallelism requires different levels of factorization on the output size, such as work group partition and thread coarsening. For a prime number, the misalignment with factorization causes the compiler to generate an `if_else` for each data read instruction to check the boundary. Unlike server GPUs, `if_else` on mobile GPUs is very costly. In comparison, hand-written kernels can flexibly avoid some boundary checking such as utilizing hardware boundary check for image type. How to avoid boundary check by a compiler will be a future work.

Speedup for DNN models Besides the five categories, in Fig. 14, we can clearly see the performance improvement for each convolution operator in the execution order for the four picked models. With several exceptions, Romou can achieve obvious improvement for all the operators of the models. Operators in ResNet-18 gains large improvement on Mali because most of them (19 out of 20) are Conv3×3 and Conv1×1. Conv3×3 with stride 2 contributes the most speedup. The last several operators of InceptionV3 on Adreno also achieves higher speedup than others. These operators use a small input shape with width and height below 16, which is more cache-friendly. Therefore, as shown in Fig. 15, considering the total time of the models, Romou achieves 6.6× to 9.1× speedup on Adreno, and 2.3× to 2.8× speedup on Mali compared to TVM. Compared to Mace, Romou shows up-to 28% performance improvement on Adreno 630, 21% on Adreno 640, and 37% on Mali.

We also compared Romou with TFLite mobile GPU backend [14]. TFLite is optimized for DNN inference on mobile GPUs, and designed as template-based code generation. Albeit not as flexible as

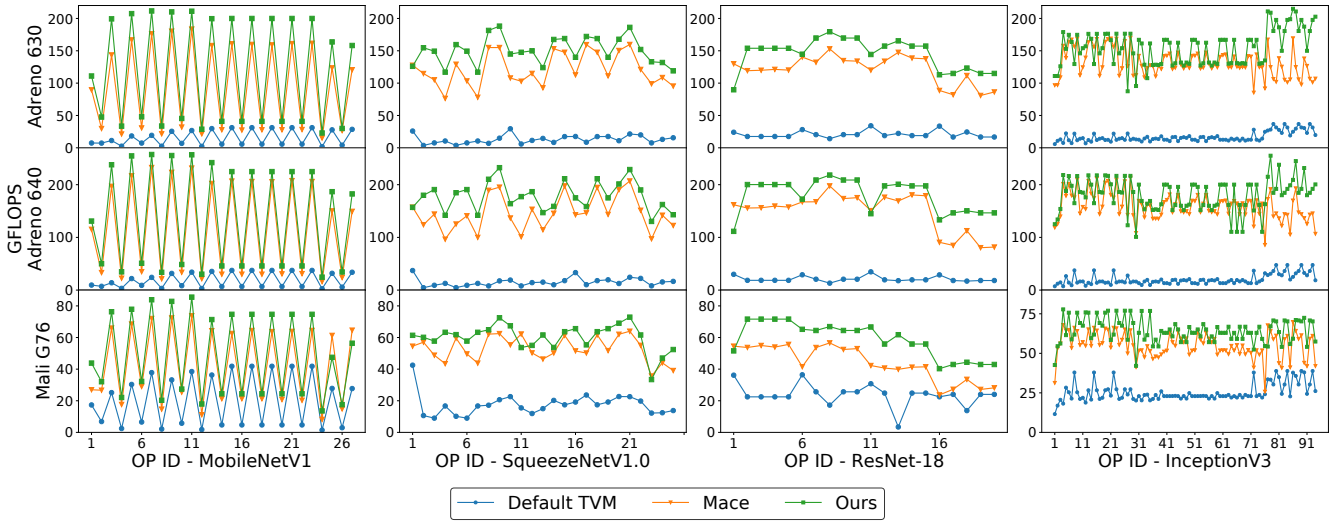

Figure 14: Operator performance comparison in the order of operator execution (x axis) for (a) MobileNetV1, (b) SqueezeNetV1.0, (c) ResNet-18, and (d) InceptionV3.

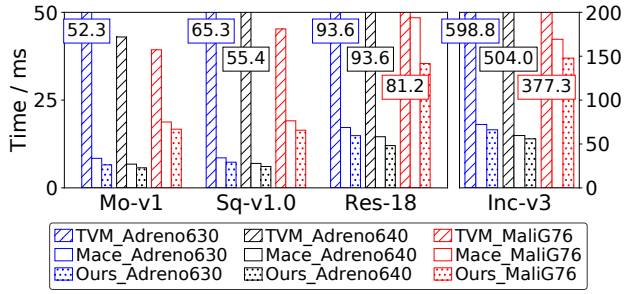

Figure 15: The sum of operator latencies for each model. Romou achieves up-to 9× speedup and 37% improvement compared to TVM and Mace respectively (text box marks TVM time).

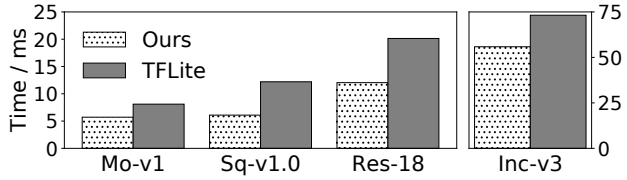

Figure 16: Latency sum of all the operators for each model on Adreno640. Romou achieves up-to 2× speedup compared to TFLite mobile GPU backend.

Romou, the simple code generation of TFLite can customize the kernel to some extent for different hyperparameters. Fig. 16 shows that Romou can achieve up-to 2× speedup.

Speedup breakdown To understand the speedup from each new primitive, Fig. 17 illustrates the speedup breakdown compared to TVM for four example convolution settings. On Adreno, `setImage` contributes more speedup than `autobroadcast`, while on Mali, it is the opposite. The reason is that on Adreno, empowering texture cache can gain more benefits since it has twice bandwidth than normal cache. By comparison, texture cache on Mali has similar performance as normal cache. `autobroadcast` can greatly improve

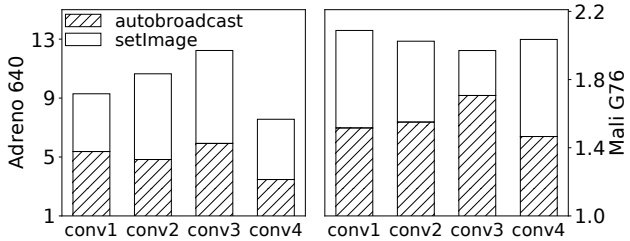

Figure 17: Speedup breakdown for each primitive of Romou compared to TVM for four example 1×1 convolution. The size [H,W,Cin,Cout] for each convolution: [64,64,512,256], [18,18,128,768], [56,56,128,128], and [28,28,256,256].

performance (49% on Adreno and 78% on Mali of the total) by reducing register usage and avoiding the register spill (refer to Fig. 5).

Searching cost comparison Fig. 18 compares the searching cost and the performance of generated kernels with and without Romou pruning for two example kernels. The two examples are for the two issues caused by large search space mentioned in Sec. 4. By pruning, Romou manages to generate faster kernels in shorter search time and fewer device runs. In Fig. 18a, though the achieved kernel performance without pruning can be the same as pruning, it costs 2.8× searching time. The search space size is reduced from 102060 to 1057 by pruning. In Fig. 18b, the kernel performance without pruning converges at local optima 105 GFLOPS in 3× searching cost than Romou which achieves 132 GFLOPS. The search space size is reduced from 1498176 to 9450.

The ALU capacity utilization in the results is similar as Fig. 3b. This is because the utilization is bottlenecked by the hardware (register and cache bandwidth), as identified in Sec. 3.2. Romou can generate kernels that approach the hardware limitation.

6 Discussion

Reliability of ArchProbe The hardware parameters profiled by ArchProbe have been cross validated with different resources, including: (1) the common parameters reported by other profiling

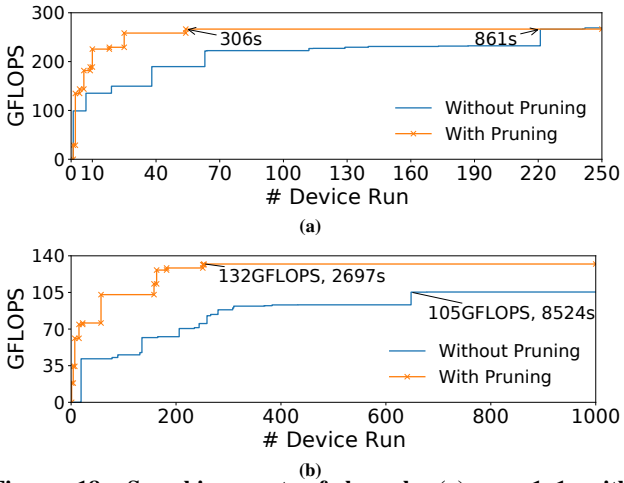

Figure 18: Searching cost of kernels (a) conv1x1 with [H,W,Cout,Cin] = [28,28,256,256]; and (b) conv3x3 with stride=2 [H,W,Cout,Cin] = [36,36,384,288].

tools [2, 5, 22], e.g., the computation and memory bandwidth; (2) the public parameters reported by vendors, e.g., the warp size of Mali GPU; (3) the related parameters reported by ArchProbe itself. For example, the profiled number of ALUs multiplied with GPU frequency can cross validate the profiled computation bandwidth.

Coverage of ArchProbe The coverage is restrained by the SIMT assumption and OpenCL implementation. Apple's mobile GPU is not supported since it uses Metal API. Other than that, GPUs supporting OpenCL could all be profiled by ArchProbe, e.g., NVIDIA's, AMD's, and PowerVR's. This paper focuses on the mainstream mobile GPUs i.e., Adreno and Mali GPUs. The results on other GPUs have not been cross validated yet. This can be the effort from us in the future or the community as ArchProbe is open sourced.

Portability of Romou Romou is built on a DNN compiler TVM, which provides a domain-specific language for tensor operations. Similarly, Romou could be integrated to other DNN compilers by extending compiler optimization passes. Romou will be hard to be integrated into general programming language compilers such as GCC or LLVM with no direct support for tensors.

Applicability to different models Romou generates optimal kernels for common DNN operators, e.g., convolution, matrix multiplication, and FC (i.e., matrix-vector multiplication). Current DNN models are mostly composed of these operators, and thus can be benefited by Romou. For example, RNN is mainly composed of FC operators; RL is composed of FC or convolution operators; Transformer is composed of matrix multiplication operators.

Applicability to different SoCs Although there are many different SoCs used by different phones, they contain common reusable IPs or embedded processors. For example, the SoCs made by Hisilicon and Mediatek use Mali GPUs. The profiled results for these GPUs should not be impacted by different SoC manufacturers.

7 Related work

Instead of low-level computation kernel optimization, some other works [16, 19, 20, 25, 27, 36, 47] aim to coordinate mobile GPUs with other resources to improve performance under different scenarios. μLayer [25] implements intra- and inter-operator parallelism between the CPU and GPU on a mobile SoC. Heimdall [47] solves the mobile GPU contention issue caued by multiple concurrent tasks by breaking down the DNN models into smaller units. DeepMon [20] utilizes mobile GPUs for continuous vision applications. These works are orthogonal to our work. They can leverage our generated kernels in their systems.

Automatic search and code generation To reduce human effort of kernel optimization, many works [1, 7, 8, 13, 15, 31, 32, 38, 42, 48, 49] propose automatic search and code generation to find the best kernel implementation from a massive search space. TVM [7] provides schedule primitives and template specification APIs to let users declare a search space, and uses machine learning methods to search for the best generated code from this space. Its following work Ansor [48] generates the schedule specifications automatically to further reduce human workload. CLTune [38] is an auto-tuner specifically for OpenCL kernels which also automatically search for the optimal configurations such as workgroup size, tile size and loop unrolling in a user-defined space. MPFFT [32] is an auto-tuning FFT library for OpenCL GPUs.

The resulting kernel performance of these works depends on the search space definition. However, none of these works solve the issue of proper space definition for a new hardware which can cover the optimal configurations and exclude the inefficient ones. For example, based on the best practice of server GPU programming, TVM provides primitives to define the search space for server GPUs and gain comparable performance as manual-written kernels. However, since there is no knowledge for mobile GPUs, the TVM-defined space lacks the optimal configurations for mobile GPUs, and the searching is stuck in billions of inefficient configurations [7]. Our work can extract the critical hardware features and define a largely-pruned space for mobile GPUs to find optimal kernels within limited steps.

There are much more works for DNN inference on server GPUs. As we have stated, mobile and server GPUs have quite different hardware design. The technologies work well for server GPUs generally cannot be applied to mobile GPUs. For example, shared memory and CPU-GPU data copying are widely used for server GPUs. However, these should be avoided on mobile GPUs for better performance. We use the TVM mobile GPU backend as the baseline since it has been adapted to mobile GPUs already. As Sec. 3 stated, the micro-benchmarks for server GPUs [37] cannot be directly used on mobile GPUs. This is because mobile GPUs do not support accurate timing or low-level instructions required by the server-GPU benchmarks. We thus do not discuss these works here.

8 Conclusion

Directed by hardware profiling, this paper empowers the rapid generation of optimal DNN kernels on diverse mobile GPUs. The profiling also exposes hardware design implications for potentially higher inference performance. The performance bottleneck on current mobile GPUs is the limited number of registers. To avoid wasting ALUs, hardware designers could scale up the register file size or L1 cache bandwidth. Besides, for Adreno GPUs, more fine-grained warp could help to eliminate idle threads for small input sizes.

References

[1] Byung Hoon Ahn, Prannoy Pilligundla, Amir Yazdanbakhsh, and Hadi Esmaeilzadeh. 2020. Chameleon: Adaptive Code Optimization for Expedited Deep Neural Network Compilation. (2020). https://openreview.net/forum?id=rygG4AVFvH

[2] Fitsum Assamnew Andargie and Jonathan Rose. 2015. Performance characterization of mobile GP-GPUs. In *AFRICON*. IEEE, 1–6.

[3] ARM. 2020. Arm Mali Bifrost and Valhall OpenCL. (2020).

[4] ARM. 2020. *The Bifrost Shader Core*. https://developer.arm.com/solutions/graphics-and-gaming/developer-guides/learn-the-basics/the-bifrost-shader-core/the-bifrost-shader-core

[5] Krishnaraj Bhat. 2020. *A tool which profiles OpenCL devices to find their peak capacities*. https://github.com/krrishnarraj/clpeak

[6] Businesswire. 2019. *Strategy Analytics: Q2 2019 Smartphone and Tablet GPU Market Share: Apple Gains Share as Arm Falters*. https://www.businesswire.com/news/home/20191118005549/en/

[7] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 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.

[8] Chris Cummins, Pavlos Petoumenos, Michel Steuwer, and Hugh Leather. 2015. Autotuning OpenCL workgroup size for stencil patterns. *arXiv preprint arXiv:1511.02490* (2015).

[9] CUTLASS. 2020. *CUTLASS Convolution*. https://github.com/NVIDIA/cutlass/blob/master/media/docs/implicit_gemm_convolution.md

[10] DeviceAtlas. 2019. *The most used smartphone GPU - 2019*. https://deviceatlas.com/blog/most-used-smartphone-gpu

[11] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv preprint arXiv:1810.04805* (2018).

[12] Michael Doggett. 2012. Texture Caches. *IEEE Micro* 32, 3 (2012), 136–141. https://doi.org/10.1109/MM.2012.44

[13] Thomas L. Falch and Anne C. Elster. 2015. Machine Learning Based Auto-Tuning for Enhanced OpenCL Performance Portability. In *IPDPS Workshops*. IEEE Computer Society, 1231–1240.

[14] Google. 2019. *TensorFlow Lite: Deploy machine learning models on mobile and IoT devices*. https://www.tensorflow.org/lite

[15] Dominik Grewe and Anton Lokhmotov. 2011. Automatically generating and tuning GPU code for sparse matrix-vector multiplication from a high-level representation. In *GPGPU*. ACM, 12.

[16] Seungyeop Han, Haichen Shen, Matthai Philipose, Sharad Agarwal, Alec Wolman, and Arvind Krishnamurthy. 2016. MCDNN: An Approximation-Based Execution Framework for Deep Stream Processing Under Resource Constraints. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services* (Singapore, Singapore) *(MobiSys '16)*. ACM, New York, NY, USA, 123–136.

[17] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. [n.d.]. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*. 770–778. https://doi.org/10.1109/CVPR.2016.90

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

[19] Loc Nguyen Huynh, Rajesh Krishna Balan, and Youngki Lee. 2016. DeepSense: A GPU-based Deep Convolutional Neural Network Framework on Commodity Mobile Devices. In *Proceedings of the 2016 Workshop on Wearable Systems and Applications* (Singapore, Singapore) *(WearSys '16)*. ACM, New York, NY, USA, 25–30.

[20] Loc N. Huynh, Youngki Lee, and Rajesh Krishna Balan. 2017. DeepMon: Mobile GPU-based Deep Learning Framework for Continuous Vision Applications. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services* (Niagara Falls, New York, USA) *(MobiSys '17)*. ACM, New York, NY, USA, 82–95.

[21] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size. *CoRR* abs/1602.07360 (2016). http://arxiv.org/abs/1602.07360

[22] Shiqi Jiang, Lihao Ran, Ting Cao, Yusen Xu, and Yunxin Liu. 2020. Profiling and optimizing deep learning inference on mobile GPUs. In *APSys*. ACM, 75–81.

[23] Xiaotang Jiang, Huan Wang, Yiliu Chen, Ziqi Wu, Lichuan Wang, Bin Zou, Yafeng Yang, Zongyang Cui, Yu Cai, Tianhang Yu, Chengfei Lyu, and Zhihua Wu. 2020. MNN: A Universal and Efficient Inference Engine. In *MLSys*. mlsys.org.

[24] Khronos. 2020. *OpenCL-SDK*. https://github.com/KhronosGroup/OpenCL-SDK/

[25] Youngsok Kim, Joonsung Kim, Dongju Chae, Daehyun Kim, and Jangwoo Kim. 2019. μLayer: Low Latency On-Device Inference Using Cooperative Single-Layer Acceleration and Processor-Friendly Quantization. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 45, 15 pages. https://doi.org/10.1145/3302424.3303950

[26] Kazuhiko Komatsu, Katsuto Sato, Yusuke Arai, Kentaro Koyama, Hiroyuki Takizawa, and Hiroaki Kobayashi. 2010. Evaluating performance and portability of OpenCL programs. In *The fifth international workshop on automatic performance tuning*, Vol. 66. 1.

[27] Nicholas D. Lane, Sourav Bhattacharya, Petko Georgiev, Claudio Forlivesi, Lei Jiao, Lorena Qendro, and Fahim Kawsar. 2016. DeepX: A Software Accelerator for Low-power Deep Learning Inference on Mobile Devices. In *Proceedings of the 15th International Conference on Information Processing in Sensor Networks* (Vienna, Austria) *(IPSN '16)*. IEEE Press, Piscataway, NJ, USA, Article 23, 12 pages.

[28] 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.

[29] Juhyun Lee, Nikolay Chirkov, Ekaterina Ignasheva, Yury Pisarchyk, Mogan Shieh, Fabio Riccardi, Raman Sarokin, Andrei Kulik, and Matthias Grundmann. 2019. On-Device Neural Net Inference with Mobile GPUs. (2019). arXiv:1907.01989 http://arxiv.org/abs/1907.01989

[30] Juhyun Lee and Raman Sarokin. 2020. *Even Faster Mobile GPU Inference with OpenCL*. https://blog.tensorflow.org/2020/08/faster-mobile-gpu-inference-with-opencl.html

[31] Yinan Li, Jack J. Dongarra, and Stanimire Tomov. 2009. A Note on Auto-tuning GEMM for GPUs. In *Computational Science - ICCS 2009, 9th International Conference, Baton Rouge, LA, USA, May 25-27, 2009, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 5544)*, Gabrielle Allen, Jaroslaw Nabrzyski, Edward Seidel, G. Dick van Albada, Jack J. Dongarra, and Peter M. A. Sloot (Eds.). Springer, 884–892. https://doi.org/10.1007/978-3-642-01970-8_89

[32] Yan Li, Yunquan Zhang, Yiqung Liu, Guoping Long, and Haipeng Jia. 2013. MPFFT: An Auto-Tuning FFT Library for OpenCL GPUs. *J. Comput. Sci. Technol.* 28, 1 (2013), 90–105.

[33] Eddie Yan Lianmin Zheng. 2021. *Auto-tuning a Convolutional Network for Mobile GPU*. https://tvm.apache.org/docs/tutorials/autotvm/tune_relay_mobile_gpu.html

[34] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. 2018. ShuffleNet V2: Practical Guidelines for Efficient CNN Architecture Design. In *The European Conference on Computer Vision (ECCV)*.

[35] MACE. 2020. https://github.com/XiaoMi/mace.

[36] Akhil Mathur, Nicholas D. Lane, Sourav Bhattacharya, Aidan Boran, Claudio Forlivesi, and Fahim Kawsar. 2017. DeepEye: Resource Efficient Local Execution of Multiple Deep Vision Models Using Wearable Commodity Hardware. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services* (Niagara Falls, New York, USA) *(MobiSys '17)*. ACM, New York, NY, USA, 68–81.

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

[38] Cedric Nugteren and Valeriu Codreanu. 2015. CLTune: A Generic Auto-Tuner for OpenCL Kernels. In *MCSoC*. IEEE Computer Society, 195–202.

[39] Qualcomm. 2017. Qualcomm Snapdragon Mobile Platform OpenCL General Programming and Optimization. (2017).

[40] Qualcomm. 2021. *TensorFlow Lite: Deploy machine learning models on mobile and IoT devices*. https://developer.qualcomm.com/software/snapdragon-profiler

[41] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman P. Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 519–530. https://doi.org/10.1145/2491956.2462176

[42] Mingcong Song, Yang Hu, Huixiang Chen, and Tao Li. 2017. Towards pervasive and user satisfactory cnn across gpu microarchitectures. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 1–12.

[43] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. 2016. Rethinking the Inception Architecture for Computer Vision. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition,*. http://arxiv.org/abs/1512.00567

[44] Xiaohu Tang, Shihao Han, Li Lyna Zhang, Ting Cao, and Yunxin Liu. 2021. To Bridge Neural Network Design and Real-World Performance: A Behaviour Study for Neural Networks. In *MLSys*. mlsys.org.

[45] Tencent. 2018. *Tencent ncnn deep learning framework*. https://github.com/Tencent/ncnn

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

[47] Juheon Yi and Youngki Lee. 2020. Heimdall: Mobile GPU Coordination Platform for Augmented Reality Applications. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking* (London, United Kingdom) *(MobiCom '20)*. Association for Computing Machinery, New York, NY, USA, Article 35, 14 pages. https://doi.org/10.1145/3372224.3419192

[48] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. 2020. Ansor: Generating high-performance tensor programs for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 863–879.

[49] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. 2020. Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 859–873.