Enhancing the Programmability and Performance Portability of GPU Tensor Operations

Arya Mazaheri[1], Johannes Schulte[1], Matthew W. Moskewicz[2], Felix Wolf[1] and Ali Jannesari[3]

[1] Technische Universität Darmstadt, Germany
[2] Deepscale Inc., Mountain View, CA, USA
[3] Iowa State University, Ames, Iowa, USA
{mazaheri,wolf}@cs.tu-darmstadt.de, j_schulte@outlook.com,
moskewcz@deepscale.ai, jannesari@iastate.edu

Abstract. Deep-learning models with convolutional networks are widely used for many artificial-intelligence tasks, thanks to the increasing adoption of high-throughput GPUs, even in mobile phones. CUDA and OpenCL are the two largely used programming interfaces for accessing the computing power of GPUs. However, attaining code portability has always been a challenge, until the introduction of the Vulkan API. Still, performance portability is not necessarily provided. In this paper, we investigate the unique characteristics of CUDA, OpenCL, and Vulkan kernels and propose a method for abstracting away syntactic differences. Such abstraction creates a single-source kernel which we use for generating code for each GPU programming interface. In addition, we expose auto-tuning parameters to further enhance performance portability. We implemented a selection of convolution operations, covering the core operations needed for deploying three common image-processing neural networks, and tuned them for NVIDIA, AMD, and ARM Mali GPUs. Our experiments show that we can generate deep-learning kernels with minimal effort for new platforms and achieve reasonable performance. Specifically, our Vulkan backend is able to provide competitive performance compared to vendor deep-learning libraries.

Keywords: GPU · deep learning · performance portability.

1 Introduction

Differences across GPU architectures and programming interfaces, such as CUDA and OpenCL, make the efficient execution of tensor operations, the constituents of convolutional neural networks (CNN), a challenging task. While CUDA works only on NVIDIA devices, the latter has been designed with portability in mind to run on any OpenCL compatible device. Nonetheless, performance is not necessarily portable [4]. Furthermore, some vendors, such as NVIDIA, are reluctant to fully support OpenCL as they see it as a rival to their own standard. This becomes even worse on a number of mobile GPUs for which there is no official support.

The Khronos group released a new programming API called Vulkan [19] along with an intermediate language named SPIR-V [18] to address the portability of GPU programs. Vulkan is inherently a low-level graphics and compute API, much closer to the behavior of the hardware, and claims to be cross-platform yet efficient on modern GPUs. Unlike others, Vulkan is supported by all major mobile and desktop GPUs. This single feature makes Vulkan a more attractive programming interface compared with OpenCL, not to mention its unique low-level optimizations. However, such worthwhile features come at a price, as it requires significantly higher programming effort. Particularly for newcomers, rewriting their code with Vulkan is a cumbersome task.

CNN inference frameworks such as TVM [1], PlaidML [7], and Tensor Comprehensions [20], which provide support for coding new tensor operations, have been optimized in many ways that allow a more efficient use of the underlying hardware. Important steps in that regard were the use of compiler techniques [7, 20] as well as device-specialized kernels written in shader assembly instead of high-level programming languages [2] or platform-independent intermediate representations. However, implementations that work on multiple platforms are often optimized for certain architectures or vendors. This reduces the portability and performance predictability of CNN execution on server-/desktop-grade GPUs and mobile GPUs alike.

In this paper, we conduct a comparative analysis of CUDA, OpenCL, and Vulkan, which we call *target APIs* in the rest of the paper. We then use the outcome to extend Boda [13], a CNN inference framework, and propose an abstraction layer that enables GPU tensor code generation using any of the target APIs. Equipped with meta-programming and auto-tuning, our code generator can create multiple implementations and select the best performing version on a given GPU. Therefore, we enhance programmability and provide better performance portability with code auto-tuning. Our experiments show that our approach eases the overall burden of targeting NVIDIA, AMD and Mali GPUs while achieving modest performance. We also achieve competitive performance using Vulkan in comparison to existing deep-learning vendor libraries. In some cases, our method achieved higher speedups, by up to $1.46\times$ and $2.28\times$ relative to cuDNN and AMD's MIOpen libraries. In essence, this paper makes the following major contributions:

- Programmability comparison of CUDA, OpenCL, and Vulkan code
- CUDA, OpenCL, and Vulkan code generation using an abstract single-source approach, which reduces the required programming effort by up to 98%
- Acceleration of convolution layers using Vulkan's new features such as kernel batching
- Performance portability analysis of our code generator for each of the three programming interfaces on latest architectures, including mobile GPUs

In the remainder of the paper, we first provide a comparative code analysis for the target APIs. Then, in Section 3 our code generation method will be introduced, followed by an evaluation in Section 4. A concise review of related works is presented in Section 5. Finally, we conclude the paper in Section 6.

2 Comparison of CUDA, OpenCL, and Vulkan

CUDA and OpenCL share a range of core concepts, such as the platform, memory, execution, and programming model. Furthermore, their syntax and built-in functions are fairly similar to each other. Thus, it is relatively straightforward to convert a CUDA to an OpenCL program, and vice versa [5,9]. On the other hand, Vulkan does not fully conform to CUDA and OpenCL standards, as it is geared both towards general-purpose computation and graphics while being portable and efficient. Various OpenCL offline compilers exist for converting C code to an intermediate language, from which later platform-specific assembly code can be easily generated. In contrast, Vulkan is able to target different platforms using a single input code and SPIR-V, a new platform-independent intermediate representation for defining shaders and compute kernels. Currently, SPIR-V code can be generated from HLSL, GLSL and C with OpenCL.

Vulkan has been designed from scratch with asynchronous multi-threading support [10, 16]. Moreover, each Vulkan-capable device exposes one or more queues that can also process work asynchronously to each other. Each queue carries a set of *commands* and acts as a gateway to the execution engine of a device. These commands can represent many actions, from data transfer and compute-shader execution to draw commands. Each command specifies the requested action along with input/output data. The information about the available actions and the corresponding data is encapsulated in a so-called *pipeline*. This pipeline is then bound to a *command buffer*, which represents a sequence of commands that should be sent in batches to the GPU. These buffers are created prior to execution and, to save in time, can be submitted to a queue for execution as many times as required. Creating command buffers is a time-consuming task. Therefore, the host code often employs multiple threads, working asynchronously, to construct command buffers in parallel. Once finished, a thread may submit these command buffers to a queue for execution. Right after the submission, the commands within a command buffer execute without any interruption in order or out of order—depending on the ordering constraints.

Despite these conceptual design differences, we prepared a mapping for the key concepts within each API in terms of memory regions and execution models in Table 1. The table shows that the memory hierarchy abstractions of the three interfaces are quite similar. Figure 1 illustrates the kernel execution space of the target APIs in more detail. Each point in the space is occupied by a thread/work-item/invocation. Each item is an execution instance of the kernel, with multiple of them combined into a thread block or group. The whole execution space is called grid or NDRange. Note that this mapping only covers the concepts shared among these APIs and does not fully cover the features of Vulkan.

		CUDA	OpenCL	Vulkan (SPIR-V)
Memory region		Global mem.	Global mem.	CrossWorkGroup, Uniform
		Constant mem.	Constant mem.	UniformConstant
		Texture mem.	Constant mem.	PushConstant
		Shared mem.	Local mem.	Workgroup
		Registers	Private memory	Private memory
Execution model		Thread	Work-item	Invocation
		Thread block	Work-group	Workgroup
		Grid	NDRange	NDRange

Table 1: A comparison of the terminology used in CUDA, OpenCL, and Vulkan

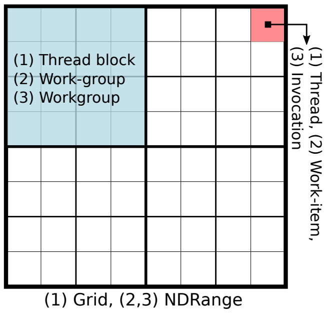
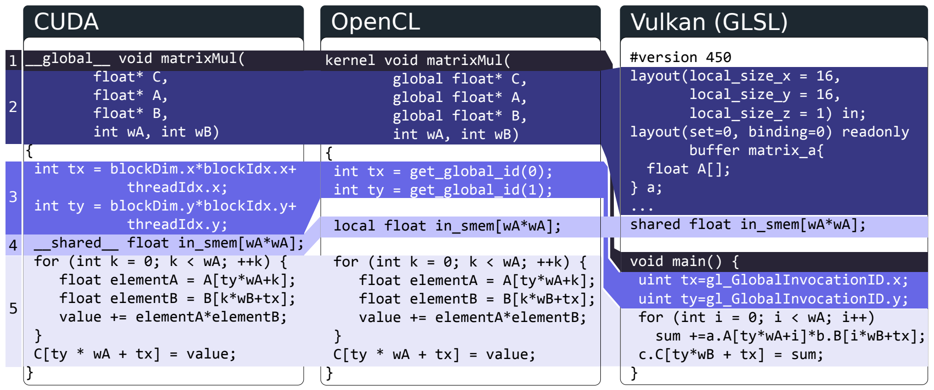


Fig. 1: Kernel execution space for (1) CUDA, (2) OpenCL, and (3) Vulkan

Further comparison shows that Vulkan is more explicit in nature rather than depending on hidden heuristics in the driver. Vulkan provides a more fine-grained control over the GPU on a much lower level. This enables programmers to enhance performance across many platforms. Even though such privilege comes with an extra programming effort, this feature can immensely increase the overall performance. Operations such as resource tracking, synchronization, memory allocation, and work submission internals benefit from being exposed to the user, which makes the application behavior more predictable and easier to control. Similarly, unnecessary background tasks such as error checking, hazard tracking, state validation, and shader compilation are removed from the runtime and instead can be done in the development phase, resulting in lower driver overhead and less CPU usage [10] compared with other APIs.

Particularly, synchronization mechanisms require the developer to be explicit about the semantics of the application but in return save a significant amount of overhead. While other APIs tend to insert implicit synchronization primitives between invocations and constructs, such as kernel executions and buffer reads, Vulkan is by default asynchronous. All synchronization between kernels or buffer I/O must be added explicitly to their respective command buffer via built-in synchronization primitives, including fences, barriers, semaphores, and events. Therefore, if no synchronization is required, we can strictly avoid the overhead of such operations.

Another difference is how Vulkan allocates memory, both on the host and the device. While CUDA and OpenCL often provide a single device buffer type and primitive functions for copying data between the host and device buffers, Vulkan puts the programmer in full control of memory management, including buffer creation, buffer type selection, memory allocation, and buffer binding. Furthermore, by making an explicit distinction between host-transparent device buffers and device-local buffers, we can implement explicit staging buffers or decide if they are not necessary—either because the amount of I/O to the buffer is negligible or because host memory and device memory are actually shared, as it is the case on many mobile platforms. Such explicit querying and handling of



```
__global__ void matrixMul(
        float* C,
        float* A,
        float* B,
        int wA, int wB)
{
 int tx = blockDim.x*blockIdx.x+
             threadIdx.x;
 int ty = blockDim.y*blockIdx.y+
             threadIdx.y;
 __shared__ float in_smem[wA*wA];
 for (int k = 0; k < wA; ++k) {
    float elementA = A[ty*wA+k];
    float elementB = B[k*wB+tx];
    value += elementA*elementB;
 }
 C[ty * wA + tx] = value;
}
```

```
kernel void matrixMul(
        global float* C,
        global float* A,
        global float* B,
        int wA, int wB)
{
 int tx = get_global_id(0);
 int ty = get_global_id(1);

 local float in_smem[wA*wA];

 for (int k = 0; k < wA; ++k) {
    float elementA = A[ty*wA+k];
    float elementB = B[k*wB+tx];
    value += elementA*elementB;
 }
 C[ty * wA + tx] = value;
}
```

```
#version 450
layout(local_size_x = 16,
       local_size_y = 16,
       local_size_z = 1) in;
layout(set=0, binding=0) readonly
       buffer matrix_a{
  float A[];
} a;
...
shared float in_smem[wA*wA];

void main() {
 uint tx=gl_GlobalInvocationID.x;
 uint ty=gl_GlobalInvocationID.y;
 for (int i = 0; i < wA; i++)
   sum +=a.A[ty*wA+i]*b.B[i*wB+tx];
 c.C[ty*wB + tx] = sum;
}
```

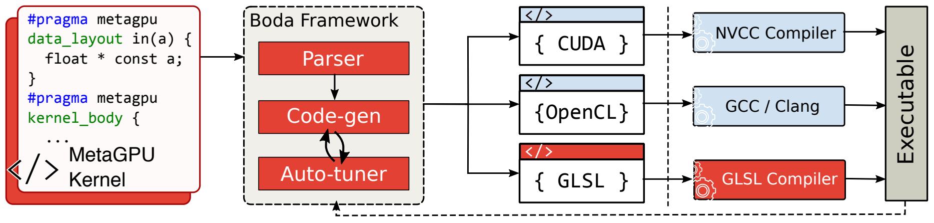
Fig. 2: A SGEMM kernel implemented with CUDA, OpenCL, and Vulkan (GLSL). Numbers on the left denote: (1) function declaration, (2) kernel arguments and data layout, (3) API-specific keywords, (4) shared-memory allocation.

the underlying hardware can reduce unnecessary work and utilize the hardware more efficiently.

Programming conventions. In contrast to other APIs, Vulkan has its own programming conventions. Therefore, code similarities might not seem obvious at the first glance. Figure 2 shows a naïve matrix-multiplication kernel implemented using each programming interface. For Vulkan, we chose GLSL as our kernel language because of its better compatibility. We trimmed off some parts of the code for brevity. Regions with the same color and number share the same functionality. Syntactically, GLSL is similar to OpenCL and CUDA. However, GLSL is more restricted in certain ways, which requires rewriting some parts of the code. The biggest three differences are:

– Arguments to a kernel are not declared in the function header. Instead, they are declared in the global scope as so-called bindings, which can then be set with Vulkan. The compiler expects the entry function for the kernel to take no arguments. However, accessing the arguments within the kernel is the same as in other APIs.
– Workgroup dimensions have to be defined in the kernel and not in the host code. Each workgroup contains many work items or compute-shader invocations.
– GLSL does not provide explicit support for pointer objects. Instead, all pointers are represented as arrays of undefined length.
– Shared-memory objects are not declared within the kernel body. Instead, they are defined in the bindings.

Due to the conceptual discrepancies between Vulkan and the other APIs, the host code of Vulkan is radically different. For example, we can create a simple buffer in CUDA (`cudaMalloc`) or OpenCL (`clCreateBuffer`) with a single line of code. To create the same buffer in Vulkan, we have to: (1) create a buffer object,



```
#pragma metagpu
data_layout in(a) {
  float * const a;
}
#pragma metagpu
kernel_body {
  ...
```

Fig. 3: The workflow behind our approach. Highlighted (dark) boxes denote the extensions in the Boda framework.

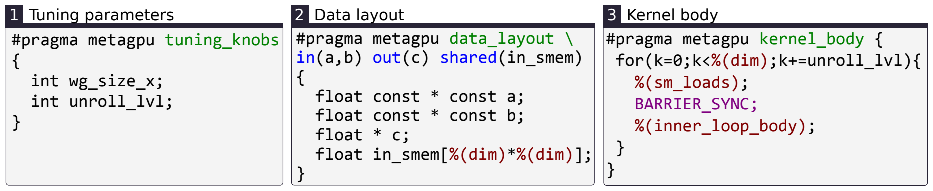
(2) get the memory requirements for that object, (3) decide which memory heap to use, (4) allocate memory on the selected heap, and (5) bind the buffer object to the allocated memory. This requires more than 40 lines of code. Clearly, host code programming in Vulkan is proportionally more complex, which stems from its explicit nature. Such code verbosity not only increases the programming effort but also makes the code more error-prone.

3 Code Generation

To generate tensor kernels, we use Boda [13] as the main engine. Boda is a CNN inference framework that uses template meta-programming to generate efficient GPU tensor code. Relying only on meta-programming made Boda a lightweight framework with minimal external software dependencies. The major required software packages comprise a C++ compiler, Python for building Boda itself, and a compatible GPU backend compiler, such as NVCC, Clang with OpenCL enabled, or GLSL to compile GPU tensor codes. We extended this framework by adding new components to provide Vulkan backend support as well as a kernel-abstraction layer to generate GPU code for each target API. Figure 3 depicts a high-level overview of our method. In the following, we will explain the key components.

MetaGPU abstraction layer. Considering the code discrepancies among the target APIs (see Figure 2), we propose MetaGPU, a compatibility layer over our target APIs. It abstracts away the syntactic differences for the basic GPU programming concepts shared by our target APIs. We did not want to invent a new language because it creates additional learning overhead for programmers. Instead, we keep the coding convention very similar to CUDA and OpenCL and simply ask the user to separate the code into three regions using `#pragma` directives, similar to OpenMP. Figure 4 shows a MetaGPU code sample.

1. *Tuning parameters*: The first region defines tuning parameters. We can either access them in the kernel code or in the host program.
2. *Data layout*: The kernel arguments and required memories which need to be allocated in the shared memory are defined within this region. Additionally,



```
#pragma metagpu tuning_knobs
{
  int wg_size_x;
  int unroll_lvl;
}
```

```
#pragma metagpu data_layout \
in(a,b) out(c) shared(in_smem)
{
  float const * const a;
  float const * const b;
  float * c;
  float in_smem[%(dim)*%(dim)];
}
```

```
#pragma metagpu kernel_body {
 for(k=0;k<%(dim);k+=unroll_lvl){
   %(sm_loads);
   BARRIER_SYNC;
   %(inner_loop_body);
 }
}
```

Fig. 4: A trivial sample of MetaGPU code.

Table 2: The list of pre-defined keywords in the kernel body alongside their corresponding value within each target API.

	CUDA	OpenCL	Vulkan
`GLOB_ID_1D`	`blockDim.x*blockIdx.x+threadIdx.x`	`get_global_id(0)`	`gl_GlobalInvocationID.x`
`LOC_ID_1D`	`threadIdx.x`	`get_local_id(0)`	`gl_LocalInvocationID.x`
`GRP_ID_1D`	`blockIdx.x`	`get_group_id(0)`	`gl_WorkGroupID.x`
`LOC_SZ_1D`	`blockDim.x`	`get_local_size(0)`	`gl_WorkGroupSize.x`
`BARRIER_SYNC`	`__syncthreads()`	`barrier(CLK_LOCAL_MEM_FENCE)`	`barrier()`

the scope of each argument should be defined with any of `in`, `out` or `shared` keywords.

3. *Kernel body*: As the name suggests, this region contains the actual kernel logic. A subtle difference is that using pointers is not allowed. Furthermore, the user has to use pre-defined keywords for accessing the GPU threads, workgroups and synchronization barriers. Table 2 shows the list of keywords and their corresponding string in each target API. MetaGPU also supports template meta-programming to generate adaptive code. Template placeholders are defined by `%(placeholder_name)%` and, using Boda, the user can populate them with C instructions or any desired string. Such a feature can help dynamically generate code and unroll loops to further improve performance.

Code generation. We first parse the input MetaGPU code and extract the three regions. The tuning parameters can later be used for auto-tuning. Then, the data layout of the kernel is parsed to find out the kernel arguments for CUDA/OpenCL code and the bindings for Vulkan GLSL code. Based on the target programming interface, we can then generate the kernel by generating corresponding argument declarations and merging them with the kernel body. All those template placeholders and abstract keywords will be replaced by their values as well.

We also added Vulkan support to Boda by creating a new backend to support host programming. All the required buffers, synchronizations, and timings will be handled by the Vulkan backend within Boda. Therefore, the end user does not have to write any host code using Vulkan. Since the programming effort of Vulkan is very high, this feature will greatly enhance programmer productivity. Furthermore, we use the kernel batching feature in Vulkan and submit up to eight compute shaders at once to the GPU. We believe that this simple optimization will greatly reduce the kernel-invocation overhead.

Auto-tuning. Tensor operations have a wide range of possible input sizes and parameters. It is generally difficult, even with meta-programming, to write code that runs well across more than a limited range of input sizes. Such tuning parameters might control thread blocking, memory access patterns, or load/-store/compute vector widths. Thus, the auto-tuner automatically searches the tuning space to find the right values for the given tuning knobs in the MetaGPU code and even across different implementation variants. This is an important step towards higher performance portability.

The key feature of our autotuning method is automatic per-platform variant selection and automated sweeping over tuning parameters. Currently, we apply a simple brute-force search over a fixed set of configurations, combined with a heuristic parameter selection method, to reduce the search space to a tractable size.

4 Experimental Results

To evaluate the programmability and performance portability of our approach, we selected a range of convolution operations and generated the corresponding GPU code for each of the target APIs. We extracted 43 unique convolutions from AlexNet, Network-in-Network, and the InceptionV1 networks, which have (1) a batch size of five, and (2) more than $1e8$ FLOPS. The rationale behind this selection is that we wanted these convolutions to model a streaming deployment scenario with high computational load but some latency tolerance. The exact specifications for each of these 43 convolutions can be found in Table 3.

For the sake of precision, we measured the execution times using GPU timers. Furthermore, to counter run-to-run variation, we executed each kernel five times and reported the average of the runtimes we obtained. Because Vulkan GPU timers were not supported on our mobile platform, we had to use its CPU timers instead. All the average speedups reported across the convolutions are computed using the geometric mean. Our evaluation artifacts, including source code and instructions on how to rerun the experiments, are available on Figshare [11].

Experimental setup. We chose NVIDIA GTX 1080 Ti and AMD Radeon RX 580, two recent desktop GPUs. We also used a mobile platform based on the Hikey 960 development kit, which contains an ARM Mali-G71 MP8 GPU. Table 4 summarizes the configuration details of the target platforms.

Programmability analysis. Our method offers performance portability while easing the burden of rewriting the program for each API. However, to quantitatively evaluate the programming effort required to generate efficient deep-learning kernels, we propose a metric based on total lines of code. Inspired by Memeti et al. [12], we use cloc to determine the lines of MetaGPU code $LOC_{MetaGPU}$ and the total unique lines of code $LOC_{TotalUniqueLines}$ needed to be written for our target APIs to provide code portability. We then define the programming effort as follows.

Table 3: KSZ, S, OC and B are the kernel size, stride, number of output channels, and batch size of each convolution operation. *in* and *out* are the sizes of input and output, specified as $y{\times}x{\times}chan$; FLOPs is the per-operation FLOP count.

KSZ	S	OC	B	*in*	*out*	FLOPs
5	1	32	5	28×28×16	28×28×32	1.00352e+08
5	1	64	5	14×14×32	14×14×64	1.00352e+08
1	1	256	5	7×7×832	7×7×256	1.04366e+08
1	1	112	5	14×14×512	14×14×112	1.12394e+08
1	1	128	5	14×14×512	14×14×128	1.28451e+08
1	1	64	5	28×28×256	28×28×64	1.28451e+08
1	1	64	5	56×56×64	56×56×64	1.28451e+08
1	1	128	5	14×14×528	14×14×128	1.32465e+08
1	1	144	5	14×14×512	14×14×144	1.44507e+08
1	1	96	5	28×28×192	28×28×96	1.44507e+08
1	1	384	5	7×7×832	7×7×384	1.56549e+08
1	1	160	5	14×14×512	14×14×160	1.60563e+08
1	1	160	5	14×14×528	14×14×160	1.65581e+08
1	1	4096	5	1×1×4096	1×1×4096	1.67772e+08
1	1	192	5	14×14×480	14×14×192	1.80634e+08
5	1	128	5	14×14×32	14×14×128	2.00704e+08
3	1	320	5	7×7×160	7×7×320	2.25792e+08
1	1	384	5	13×13×384	13×13×384	2.49201e+08
1	1	128	5	28×28×256	28×28×128	2.56901e+08
1	1	256	5	14×14×528	14×14×256	2.64929e+08
1	1	96	5	54×54×96	54×54×96	2.68739e+08
3	1	384	5	7×7×192	7×7×384	3.2514e+08
3	1	208	5	14×14×96	14×14×208	3.52236e+08
1	1	1000	5	6×6×1024	6×6×1000	3.6864e+08
1	1	1024	5	6×6×1024	6×6×1024	3.77487e+08
6	1	4096	5	6×6×256	1×1×4096	3.77487e+08
3	1	224	5	14×14×112	14×14×224	4.42552e+08
1	1	256	5	27×27×256	27×27×256	4.77757e+08
3	1	256	5	14×14×128	14×14×256	5.78028e+08
5	1	96	5	28×28×32	28×28×96	6.02112e+08
3	1	288	5	14×14×144	14×14×288	7.31566e+08
3	1	128	5	28×28×96	28×28×128	8.67041e+08
3	1	320	5	14×14×160	14×14×320	9.03168e+08
11	4	96	5	224×224×3	54×54×96	1.01617e+09
11	4	96	5	227×227×3	55×55×96	1.05415e+09
7	2	64	5	224×224×3	112×112×64	1.18014e+09
3	1	1024	5	6×6×384	6×6×1024	1.27402e+09
3	1	256	5	13×13×384	13×13×256	1.4952e+09
3	1	384	5	13×13×256	13×13×384	1.4952e+09
3	1	192	5	28×28×128	28×28×192	1.73408e+09
3	1	384	5	13×13×384	13×13×384	2.24281e+09
3	1	192	5	56×56×64	56×56×192	3.46817e+09
5	1	256	5	27×27×96	27×27×256	4.47898e+09

Table 4: Experimental setup.

	Nvidia GTX 1080Ti	AMD RX 580	ARM Mali G71 MP8
OS	Ubuntu 16.04 64-bit		Android 7.0
CPU	Intel Xeon Gold 6126, 12Core @ 2.6GHz		4 Cortex A73 + 4 Cortex A53
Host Memory	64 GB		3GB LPDDR4 SDRAM
GPU Memory	11GB GDDR5X	8GB GDDR5	-
Driver	Linux Display Driver 410.66	AMDGPU-PRO Driver 17.40	Native driver
CUDA	CUDA 10.0	-	-
OpenCL	OpenCL 1.2	OpenCL 2.0	OpenCL 2.0
Vulkan SDK	Vulkan 1.1.97	Vulkan 1.1.97	Vulkan 1.1.97

Table 5: Lines-of-code comparison for different convolution implementations alongside computed effort metric.

	$LOC_{MetaGPU}$	LOC_{CUDA}	LOC_{OpenCL}	LOC_{Vulkan}	$LOC_{TotalUniqueLines}$	Effort
Direct convolution	113	562	631	1137	2330	4.84
Tiled convolution	115	548	618	1119	2285	5.03
GEMM convolution	89	1103	1172	1666	3941	2.25
1x1 convolution	160	1190	1259	1761	4210	3.80

$$\text{Effort}[\%] = (LOC_{MetaGPU}/LOC_{TotalUniqueLines}) \times 100 \tag{1}$$

In most CNN frameworks, including Boda, multiple convolution variants exist, each specialized for a specific case. For instance, Boda provides direct, tiled, GEMM, and 1×1 convolution variants. We counted the LOCs for each variant and target API. The results are shown in Table 5. For a fair programming effort analysis, we used total unique lines between all the target APIs. The results indicate that using our method requires on average 4% of the total effort needed to implement the code with all of the target APIs.

Performance portability analysis. We now present per-convolution-operation runtime results across hardware targets and programming interfaces to illustrate the performance portability of our method. We sorted the operations by FLOP count, a reasonable proxy for the difficulty of the operations.

A runtime comparison of CUDA, OpenCL, and Vulkan on our benchmark set of operations is given in Figure 5. All runtimes are for running each operation using the best function generated by our method for that operation, selected by auto-tuning. The implementations are the same and only the backend API is different. We also added cuDNN runtimes as the baseline to show the performance of our method relative to the highly-tuned vendor CNN library. The results clearly show that our Vulkan backend often yields lower runtime in comparison to the other two, and closer to cuDNN's performance. We believe that this is owed to kernel batching and the optimizations provided by Vulkan. Note that we are slower especially in cases with 3×3 kernel sizes, where cuDNN is using Winograd convolution, which we have not yet implemented. On average, Vulkan outperformed CUDA and OpenCL kernels by a factor of 1.54 and 1.86,

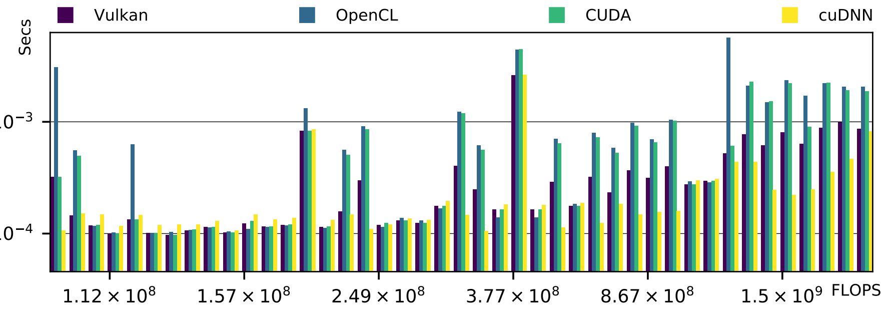


Fig. 5: The runtime comparison of kernels generated by our method and cuDNN vendor library on Nvidia GTX 1080 Ti.

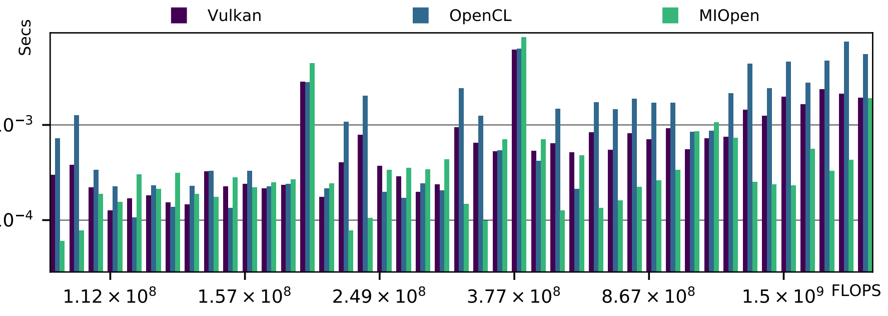


Fig. 6: The runtime comparison of kernels generated by our method and the MIOpen vendor library on AMD Radeon RX 580.

respectively. Although cuDNN was able to operate 1.38× faster than Vulkan, we noticed that in some cases, Vulkan can be up to 1.46× faster than cuDNN.

Figure 6 compares the runtimes of our benchmark using OpenCL and Vulkan on the AMD GPU. We also show MIOpen runtimes as the baseline to show the performance of our method relative to the optimized AMD CNN library. Again, we notice that Vulkan outperforms OpenCL by a factor of 1.51 on average. Presumably benefiting from Winograd convolutions and a highly-optimized MIOpenGEMM, MIOpen performs better than our Vulkan implementation for 25 out of 43 operations. For the 18 remaining operations, however our Vulkan version runs up to 2.28× faster than MIOpen.

Together, Figures 5 and 6 illustrate that we were able to achieve competitive performance compared to the vendor libraries on two different platforms. This observation confirms that our method achieves good performance portability. To further validate the effect of auto-tuning on performance portability, we executed the Vulkan code generated by our backend with and without auto-tuning. The final results after selecting the right variant and tuning parameters are shown in Figure 7. Note that runtimes are reported using CPU timers, because Vulkan GPU timestamps are not supported on Mali G71. Auto-tuning requires much less

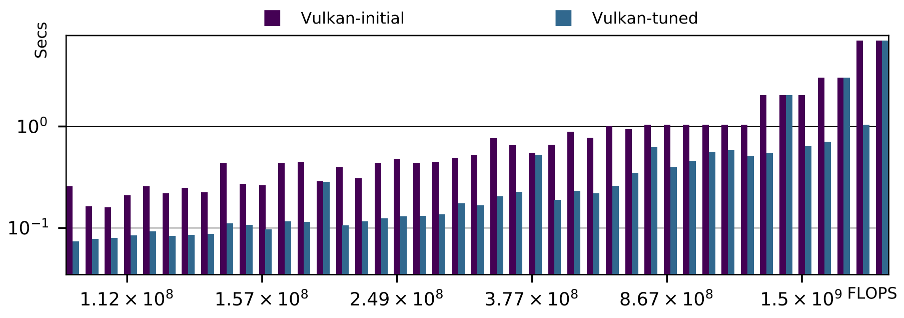


Fig. 7: Vulkan performance with and without auto-tuning on Mali G71.

effort than manual tuning and improves performance significantly—on average by a factor of 3.11.

5 Related Work

With the increasing popularity of GPUs, several authors compared CUDA and OpenCL programming models [3–5, 8, 9, 12, 15, 17], but none of them studied Vulkan. Karimi et al. [8] and Fang et al. [5] compared CUDA with OpenCL, focusing on their performance on conventional desktop GPUs. Du et al. [4] were among the first who studied OpenCL performance portability and showed that performance is not necessarily portable across different architectures. In contrast to these studies, we carried out our experiments on recent architectures and included mobile GPUs to augment the performance portability analysis. Kim et al. [9] proposed a one-to-one translation mechanism for converting CUDA to OpenCL kernels, but they do not employ any meta-programming and code generation to achieve higher efficiency as we do. To the best of our knowledge, VComputeBench [10] is the only work which investigates Vulkan from the compute perspective and proposes it as a viable cross-platform GPGPU programming model. However, the authors concentrated more on creating a benchmark suite and did not provide a method for code translation and enhancing performance portability.

The amount of work published on the portable execution of CNNs as well as the use of Vulkan in this context is very limited. In recent years, a number of tensor compilers and frameworks, such as PlaidML [7], Tensor Comprehensions [20], TVM [1], DeepMon [6], and Boda [13, 14] have been introduced to address the portability issue of deep-learning frameworks using code generation and compiler optimizations. However, none of them are able to generate code for our target APIs using a single-source approach for the kernel definition. PlaidML and Tensor Comprehension do not support Vulkan at all. TVM and DeepMon are able to generate Vulkan code, but they require different input code for each programming model, demanding extra programming effort to introduce new tensor operations. Boda, on the other hand, has a compatibility layer on top of OpenCL and CUDA. Its approach is based on writing lowest-common-denominator code that is compatible between the two and uses macro definitions to abstract away syntactic differences. However, because of its larger code divergence such an approach is definitely not extendable to include Vulkan as well.

6 Conclusion and Outlook

This paper presents a comparative analysis of the GPU programming interfaces CUDA, OpenCL, and Vulkan. We let this comparison guide us in developing a method for generating tensor GPU kernels coded in any of those APIs from a single source that abstracts away the syntactic differences between these APIs. We implemented our approach in a state-of-the-art CNN inference framework called Boda and analyzed the programmability and performance portability of the generated kernels. Based on our experiments, our method reduces the programming effort by 98% when code portability between different APIs is demanded. Furthermore, we showed that Vulkan offers better performance compared with other APIs on our convolution benchmarks and sometimes performs better than CNN vendor libraries.

Acknowledgment. This research has been supported by the Klaus Tschira Foundation, the Hessian LOEWE initiative within the Software-Factory 4.0 project, and the German Research Foundation (DFG) through the Program Performance Engineering for Scientific Software.

References

1. Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E.Q., Shen, H., Cowan, M., Wang, L., Hu, Y., Ceze, L., Guestrin, C., Krishnamurthy, A.: Tvm: An automated end-to-end optimizing compiler for deep learning. In: 13th USENIX Symposium on Operating Systems Design and Implementation. pp. 578–594. OSDI'18 (2018)
2. Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., Shelhamer, E.: cuDNN: Efficient primitives for deep learning. arXiv preprint arXiv:1410.0759 (2014)
3. Da Silva, H.C., Pisani, F., Borin, E.: A comparative study of sycl, OpenCL, and OpenMP. In: Proc. of International Symposium on Computer Architecture and High-Performance Computing Workshops. pp. 61–66. SBAC-PADW'16, IEEE (2016)
4. Du, P., Weber, R., Luszczek, P., Tomov, S., Peterson, G., Dongarra, J.: From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. Parallel Computing **38**(8), 391–407 (2012)
5. Fang, J., Varbanescu, A.L., Sips, H.: A comprehensive performance comparison of CUDA and OpenCL. In: Proc. of International Conference on Parallel Processing (ICPP). pp. 216–225. IEEE (2011)

6. Huynh, L.N., Lee, Y., Balan, R.K.: Deepmon: Mobile gpu-based deep learning framework for continuous vision applications. In: Proc. of 15th Annual International Conference on Mobile Systems, Applications, and Services. pp. 82–95. MobiSys'17, ACM (2017)
7. Intel: PlaidML (2019), https://www.intel.ai/plaidml
8. Karimi, K., Dickson, N.G., Hamze, F.: A performance comparison of CUDA and OpenCL. arXiv preprint arXiv:1005.2581 (2010)
9. Kim, J., Dao, T.T., Jung, J., Joo, J., Lee, J.: Bridging OpenCL and CUDA: a comparative analysis and translation. In: Proc. of International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–12. SC'15, ACM (2015)
10. Mammeri, N., Juurlink, B.: VComputeBench: A Vulkan benchmark suite for GPGPU on mobile and embedded GPUs. In: Proc. of International Symposium on Workload Characterization. pp. 25–35. IISWC'18, IEEE (2018)
11. Mazaheri, A., Schulte, J., Moskewicz, M., Wolf, F., Jannesari, A.: Artifact Evaluation (2019), https://doi.org/10.6084/m9.figshare.8490146
12. Memeti, S., Li, L., Pllana, S., Kołodziej, J., Kessler, C.: Benchmarking OpenCL, OpenACC, OpenMP, and CUDA: programming productivity, performance, and energy consumption. In: Proc. of Workshop on Adaptive Resource Management and Scheduling for Cloud Computing. pp. 1–6. ACM (2017)
13. Moskewicz, M.W., Jannesari, A., Keutzer, K.: A metaprogramming and autotuning framework for deploying deep learning applications. arXiv preprint arXiv:1611.06945 (2016)
14. Moskewicz, M.W., Jannesari, A., Keutzer, K.: Boda: A holistic approach for implementing neural network computations. In: Proc. of International Conference on Computing Frontiers. pp. 53–62. CF'17, ACM (2017)
15. Oliveira, R.S., Rocha, B.M., Amorim, R.M., Campos, F.O., Meira, W., Toledo, E.M.a., dos Santos, R.W.: Comparing CUDA, OpenCL and OpenGL implementations of the cardiac monodomain equations. In: Proc. of 9th International Conference on Parallel Processing and Applied Mathematics. pp. 111–120. PPAM'11, Springer-Verlag (2011)
16. Sampson, A.: Let's fix OpenGL. In: Leibniz International Proceedings in Informatics. LIPIcs '17, vol. 71. Schloss Dagstuhl, Leibniz-Zentrum füer Informatik (2017)
17. Su, C.L., Chen, P.Y., Lan, C.C., Huang, L.S., Wu, K.H.: Overview and comparison of OpenCL and CUDA technology for GPGPU. In: Proc. of Asia Pacific Conference on Circuits and Systems. pp. 448–451. APCCAS'12, IEEE (2012)
18. The Khronos Group: Khronos SPIR-V registry (2019), https://www.khronos.org/registry/spir-v
19. The Khronos Group: Khronos Vulkan registry (2019), https://www.khronos.org/registry/vulkan
20. Vasilache, N., Zinenko, O., Theodoridis, T., Goyal, P., DeVito, Z., Moses, W.S., Verdoolaege, S., Adams, A., Cohen, A.: Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. arXiv preprint arXiv:1802.04730 (2018)