HNMTP CONV: OPTIMIZE CONVOLUTION ALGORITHM FOR SINGLE-IMAGE CONVOLUTION NEURAL NETWORK INFERENCE ON MOBILE GPUS

A PREPRINT

Zhuoran Ji
Department of Computer Science
The University of Hong Kong
Hong Kong, China
jizr@hku.hk

September 9, 2019

ABSTRACT

Convolution neural networks are widely used for mobile applications. However, GPU convolution algorithms are designed for mini-batch neural network training, the single-image convolution neural network inference algorithm on mobile GPUs is not well-studied. After discussing the usage difference and examining the existing convolution algorithms, we proposed the HNTMP convolution algorithm. The HNTMP convolution algorithm achieves $14.6\times$ speedup than the most popular *im2col* convolution algorithm, and $2.1\times$ speedup than the fastest existing convolution algorithm (direct convolution) as far as we know.

***K*eywords** Convolution Neural Network · Mobile GPU · Edge Computing Platforms

1 Introduction

This preprint is a technical report rather than a paper and the introduction is kept short. In this report, we addressed the challenges and opportunities of single-image convolution neural network inference on mobile GPUs. We discussed the popular existing GPU convolution algorithms and proposed the HNTMP (**H**NTMP does **N**ot **M**ap **T**hreads to **P**ixels) convolution algorithm.

2 Challenges and Opportunities of Single-Image Inference on Mobile GPUs

Even though both of them deal with convolution neural networks, single-image convolution neural network inference on mobile GPUs is quite different from mini-batch convolution neural network training on high-end dedicated GPUs, if they are not entirely different stories. The differences mainly come from three aspects: the disparity of input images numbers, the gap between mobile GPUs and dedicated GPUs, and the different engineering considerations.

2.1 Single-Image Reduces Threads-Level Parallelism

The most critical difference or challenge of single-image inference is that only one image is fed into the convolution neural network. For a single image, the insufficiency of data parallelism prevents us from using as many threads as mini-batch training. There are two ways for GPUs to hide the latency: thread-level parallelism (TLP) and instruction-level parallelism (ILP). The insufficiency of threads reduce the thread-level parallelism significantly and may lead to low GPU utilization. For single-image inference, we need to take better advantage of instruction-level parallelism.

The instruction-level parallelism is to issue long latency independent instruction in pipeline. The compilers are responsible for scheduling the instructions and rearrange to achieve high instruction-level parallelism. It also fusions

the memory instructions and arithmetic instructions to hide the latency. For example, consider a GPU kernel which accumulates four floating-point numbers stored in the global memory (CODE 1). It is compiled and executed on a virtual GPU, whose global memory access latency is two cycles, and floating-point addition latency is one cycle. We also assume the global memory access is synchronous instruction. Thus the compiler needs to insert NOP instruction if the to be loaded value is needed by the next instruction but has not been available yet. It should be noted that global memory access in real GPUs is usually asynchronous instruction and costs hundreds of cycles, and our assumption is to simplify the illustration.

The most trivial assembly code without instruction scheduling is shown in CODE 2. The global memory load issued at tick 2 is not available at tick 3. As the next instruction needs the value, a NOP instruction is inserted at tick 3 to guarantee the floating-point number addition instruction reads the correct value from the register `r2`. There is no instruction-level parallelism to hide the latency, and the total execution time is 13 cycles.

On the other hand, the optimal assembly code is shown in CODE 3. At tick 3, instead of inserting a NOP instruction, the global memory load instruction of m1 is brought to this tick to keep the GPU busy. As the loading of `m1` does not depend on the result of the global memory load of `m0`, these two instructions can be issued independently. With instruction-level parallelism, the total execution time is reduced to 9 cycles, and the GPU utilization is 100%. However, the `m1` should be loaded into another register (`r3`), as the value of `r2` will be used by following instructions.

```
a = 0
a += memory[0]
a += memory[1]
a += memory[2]
a += memory[3]
```

CODE 1: GPU Kernel Code

```
1: SET r1, 0
2: LOAD r2, m0
3: NOP
4: ADD r1, r1, r2
5: LOAD r2, m1
6: NOP
7: ADD r1, r1, r2
8: LOAD r2, m2
9: NOP
10: ADD r1, r1, r2
11: LOAD r2, m3
12: NOP
13: ADD r1, r1, r2
```

CODE 2: No Instruction Scheduling

```
1: SET r1, 0
2: LOAD r2, m0
3: LOAD r3, m1
4: ADD r1, r1, r2
5: LOAD r2, m2
6: ADD r1, r1, r3
7: LOAD r3, m3
8: ADD r1, r1, r2
9: ADD r1, r1, r3
```

CODE 3: With Instruction Scheduling

However, instruction scheduling is not come for free, and there are mainly two constraints. The first constraint is introduced by memory barriers. There is a common optimization method for GPU kernel that all threads in a workgroup load data from the global memory to the shared memory collaboratively. As the thread needs to use the data loaded by other threads, before reading the shared memory, a shared memory barrier is needed to guarantee every value has been written into the shared memory. The compiler cannot schedule instructions across memory barriers, which limits the feasibility of instructions rearrangement and the instruction-level parallelism. Even if the GPU kernel code is rearranged manually, the pipeline is still not possible. In OpenCL, the shared memory barriers are applied to all shared memory of the workgroup. Therefore, even if we know that a group of arithmetic instructions only depends on one of the shared memory (denoted as S1) and another group of arithmetic instructions depends on another shared memory (denoted as S2), this relation cannot be expressed by high-level GPU programming languages. In the expression of OpenCL, the latter group of arithmetic instructions depends on both S1 and S2. On GPUs that guarantee the order of the shared memory operations, it is possible to load values from the global memory into the shared memory parallelly with assembly code. However, it is incredibly costly to write and optimize the assembly codes for every convolution setting.

Another constraint is the additional usage of registers. As the value loaded from the global memory needs to be stored at different registers, pipelining the memory loading will significantly increase the number of registers used by a thread. As GPUs usually do not support dynamic register allocation, all used registers need to be reserved in the lifetime of the threads. More registers usage means fewer warps can be assigned for each compute unit, which may reduce the thread-level parallelism. However, as discussed above, each compute unit only has few warps for single-image inference, and the register usage is not a critical problem in this case.

To know more about latency hiding on GPUs, Volkov provided several high-quality materials [1, 2].

2.2 Memory Bandwidth and Energy Consumption

Another challenge of mobile GPUs comes from hardware limitations. Both mobile GPUs and integrated GPUs use LPDDR4 or DDR4 as off-chip global memory, whose bandwidth is far less than GDDR6 and HBM2. The memory

bandwidth of dual-channel LPDDR4 is about 30 GB/s, while the bandwidth of GDDR6 and HBM2 is about 600GB/s and 1TB/s, respectively. Even worse, the limited memory bandwidth of mobile GPUs is shared by CPUs and other processors of the SoC, leading to even lower real memory bandwidth. It is much more easily for the global memory access to become the bottleneck on mobile GPUs.

Additionally, the off-chip memory access consumes tens of times energy compared with on-chip cache access and hundreds of times the energy compared with floating-point addition and multiply. Even though energy consumption becomes to draw attention in the deep learning areas, it is the last consideration when designing convolution algorithms for GPUs that are powered by mains electricity. However, edge computing platforms are usually battery-powered. The energy consumption determines the battery lifetime.

Therefore, when trading off between global memory access with other operations, global memory access is more undesirable on mobile GPUs than dedicated GPUs.

2.3 Tuning is Possible for Inference

Last but not least, the engineering choice of inference is usually different from training. For deep neural network training, various convolution neural network architecture and combination of convolution setting are examined. In such a scenario, an algorithm that provides stable and acceptable performance for any convolution setting is preferred. It frees neural network developers from tuning the GPU kernels for every convolution setting. Thus neural network developers can focus on the performance of the neural networks. However, for inference, the architecture and settings of convolution neural network are fixed. Therefore, the goals become to optimize for short inference time and low energy consumption. It is worth to adopt the convolution algorithm that achieves the fast speed even if gert efforts need to be pard for tuning.

3 Existing GPU Convolution Algorithms

This section reviews several popular GPU convolution algorithms, which are unrolling-based convolution(*im2col* and *libdnn*), Winograd convolution, and direct convolution. The FFT based convolution is not discussed, which performs well only with large convolution kernel size, but the state-of-the-art convolution neural networks mainly use small kernels.

3.1 Unrolling-based Convolution Algorithms

Unrolling-based convolution is the most popular GPU convolution algorithm. The key idea is to convert the sliding-window convolution, which is hard to optimize, into well-studied matrix multiplication. The local regions of input images are repeated and unrolled into columns of a matrix, and the convolution kernels are unrolled into rows of another matrix. The unrolling procedure of the input images is named *im2col*. Each pixel of the output image is the dot product of the corresponding column of the input matrix and row of the kernel matrix. Thus the whole output images are the product of the input matrix and the kernel matrix, and there are many efficient and highly-optimized libraries for matrix multiplication(GEMM), such as cuBLAS and clBLAS. I found a great blog about this progress [3].

As the BLAS libraries provide GEMM as a standalone function, most implementation of the unrolling-based convolution separates the *im2col* and matrix multiplication into two GPU kernels. The *im2col* kernel first loads input images from global memory and writes the unrolled input matrix back to global memory, and the unrolled input matrix is then loaded from global memory by GEMM. The size of the unrolled input matrix is `kernel_size` times of the input images, which incurs significant global memory bandwidth overhead for edge computing platform due to the unnecessary global memory read and write.

Another unrolling-based convolution implementation named *libdnn* eliminates the global memory bandwidth overhead by combining the *im2col* and GEMM into a single GPU kernel. Instead of reading the unrolled input matrix, each workgroup of *libdnn* loads original input images and unrolls it into a tile of the unrolled input matrix that processed by this workgroup just-in-time. The tile of the unrolled input block is multiplied with the corresponding tile of the kernel matrix and then discarded after usage. As the unrolled input matrix is constructed and used by the same kernel, it does not need to be stored in, read from, and written back to the global memory. The code is available at [4] and there is also a technical report [5].

It should be noticed that even though *libdnn* eliminates the storage and bandwidth overhead incurred by the unrolled input matrix, the performance of *libdnn* is not always better, especially for convolution neural network training. First, the state-of-the-art dedicated GPUs have up to 1 TB/s global memory bandwidth. Together with the thread-level parallelism, the read and write of the unrolled input matrix only cost negligible time compared with the matrix multiplication. Also,

as each tile of the unrolled input matrix is used by different workgroups, many workgroups need to unroll the same tile. The unrolling operation involves complex index calculation and irregular global memory access, which are unfriendly to GPUs.

3.2 Winogard Convolution Algorithm

The Winogard convolution algorithm [6] was proposed in 1980, but it has not been widely used by convolution neural network until 2015. It divides the input images into small tiles and transforms both the tiled input images and convolution kernels into a group of small matrices. Each matrix of the transformed input matrices group is multiplied with the corresponding matrix of the transformed kernel matrices group. As the size of the transformed matrices is much smaller than the original matrices, the total arithmetic numbers are reduced. For example, if the kernel size is $R \times R$ and the tile size is $M \times M$, the standard algorithm uses M^2R^2 multiplications while the Winogard convolution only needs $(M + R - 1)^2 \times R^2$ multiplications for each tile.

However, the number of addition and constant multiplications required by the Winogard transforms increases quadratically with the kernel size. Thus for large convolution kernels, the memory access and the arithmetic complexity of the transformation will overwhelm the benefits of multiplication reduction. Also, the transforms introduce substantial global memory access, while global memory is expensive on mobile GPUs due to the bandwidth limitation and energy consumption.

3.3 Direct Convolution Algorithm

Direct convolution is the convolution algorithm that follows the definition of convolution [7]. The convolution kernels slide along the input images, and the dot product between the convolution kernel and the slid image region is accumulated to the corresponding output image. From the aspect of memory shared usage, the direct convolution needs the least shared memory as it caches the input image rather than the unrolled input matrix. Meanwhile, as direct convolution does not need to do the complex index and memory address calculation, the scalar arithmetic operands of direct convolution are much less than the unrolled-based convolution.

However, direct convolution usually needs great efforts to tune the GPU kernels to get a reasonable speed, let alone the optimal speed. compared with the *libdnn*, whose critical hyperparameters are only the size of the workgroup and workload of each workgroup, direct convolution has all these hyperparameters in addition with much more other hyperparameters. For example, the number of the output channels dealt by each workgroup affects how many times the input images are read and the number of the warps. Meanwhile, whether the filter should be loaded collectively and cached in the shared memory is always a hard choice, especially for single-image inference, whose latency cannot be hidden by TLP. If not, then each thread needs to use kernel size times global memory load operands, and all threads within a workgroup load the same data at the same time. The most serious problem is that the compiler may try to pipeline the kernel loading as it is global memory operands. A great number of registers are needed, which is not necessary if convolution kernels are stored in the shared memory. Also, even with the L2 cache, the overhead caused by issuing the global memory still quite substantial. On the other hand, if the filter is loaded collectively, as shown in Algorithm 1, then a shared memory fence needs to be put after every convolution kernel load. The instructions between such two fence are all arithmetic instructions, so the compiler cannot fusion the memory instructions and arithmetic instruction to hide the latency, which significantly hurts the ILP.

Usually, for neural network training, which has enough TLP, caching the filter will benefit. On the other hand, for single-image neural network inference, which mainly relies on ILP as discussed above, not caching the filter is better.

4 Methodology

In this report, we proposed a convolution algorithm specialized for single-image convolution neural network inference on mobile GPUs, named HNTMP convolution[1]. It is based on the direct convolution algorithm but eliminates the tradeoff between caching the filter or not. The key idea is to map threads to output channels and iterate along pixel, instead of mapping threads to pixels and iterate along output channels. The rough idea is shown in Algorithm 2.

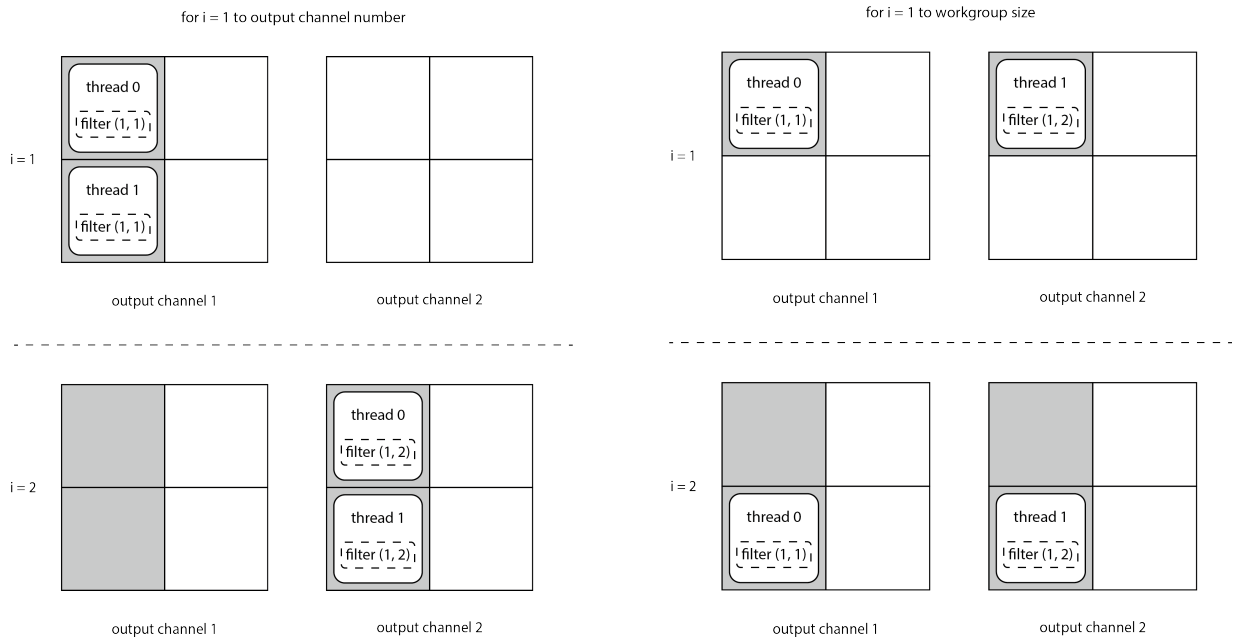
Same with the original direct convolution algorithm, all threads of a workgroup first load the input image tile from the global memory into the shared memory collaboratively. In this stage, each thread is mapped to a pixel. After that, each thread is mapped to an output channel and computes the whole output image tile of this output channel. As all pixels of an output channel is calculated with the same convolution filter, each thread loads and only needs to load one convolution filter in its whole lifetime. In contrast, for the original direct convolution algorithm, each thread needs to

[1]The code is available at: https://github.com/jizhuoran/sj_convolution

Algorithm 1 Direct Convolution Algorithm

```
1:  function CONVOLUTION(input, filter, output)
2:      __local float in_img_cache[TILE_WIDTH][TILE_WIDTH]
3:      __private float out_img_cache[OUTPUT_CHANNELS] = .0
4:      for in_channel ← 1 to input_channels do
5:
6:          in_cache[local_id.y][local_id.x] = input[in_channel][global_id.y][global_id.x]
7:          BARRIER(LOCAL_MEM)
8:
9:          for out_channel ← 1 to OUTPUT_CHANNELS do
10:
11:             filter_cache[local_id] = filter[in_channel][out_channel][local_id]
12:             BARRIER(LOCAL_MEM)
13:
14:             for r ← 1 to FILTER_WIDTH do
15:                 for c ← 1 to FILTER_WIDTH do
16:                     out_reg[out_channel] += in_cache[local_id.y+r][local_id.x+c] * filter_cache[r*FILTER_WIDTH+c]
17:                 end for
18:             end for
19:         end for
20:
21:     end for
22:
23:     for out_channel ← 1 to OUTPUT_CHANNELS do
24:         output[bucket_id][global_id.y][global_id.x] = out_reg[bucket_id]
25:     end for
26: end function
```

load `workgroup_size` convolution filters to compute the same number of output channels. The difference is illustrated in Figure 1.



(a) Direct Convolution (b) HNMTP Convolution

Figure 1: Difference of Direct Convolution and HNMTP Convolution. $filter(i, j)$ refers to the convolution filter of input channel i and output channel j. The gray square indicates this pixel has been calculated.

To further reduce the registers used for storing the convolution filter, a thread will load one weight of the filter corresponding to its assigned output channels each time. This weight is multiplied with the whole input image tile, and the products are accumulated into the output image tile.

Algorithm 2 HNMTP Convolution Algorithm

```
1:  GPU Related Variables
2:      local_id.#, the #id of the thread in it workgroup
3:      group_id.#, the #id of the workgroup
4:      LOCAL_DIM_#, the shape of the workgroup at dim#
5:      TILE_SIZE_#, LOCAL_DIM_# + 2 * HALF_FILTER_WIDTH
6:      C, R, S, K, the input channel, filter width, filter height, output channel
7:  end GPU Related Variables
8:
9:  function CONVOLUTION(input, filter, output)
10:
11:     __local float in_img_cache[TILE_SIZE_Y][TILE_SIZE_X]
12:     __private float out_img_cache[DIM] = .0
13:
14:     offset_x ← group_id.x * LOCAL_DIM_X
15:     offset_y ← group_id.y * LOCAL_DIM_Y
16:
17:     for in_channel ← 1 to input_channels do
18:
19:         for i ← 1 to ⌈TILE_SIZE / LOCAL_DIM⌉ do
20:             dest ← i * DIM + local_id
21:             destY ← dest / TILE_SIZE_X
22:             destX ← dest % TILE_SIZE_Y
23:             srcY ← offset_y + destY - HALF_FILTER_WIDTH
24:             srcX ← offset_x + destX - HALF_FILTER_WIDTH
25:             in_img_cache[destY][destX] = input[in_channel][srcY][srcX]          ▷ ignore boundary check
26:         end for
27:         BARRIER(LOCAL_MEM)
28:
29:         for r ← 1 to FILTER_WIDTH do
30:             for s ← 1 to FILTER_WIDTH do
31:                 filter_reg ← filter[i][r][s][local_id]   ▷ the filter is reorganized by [C][R][S][K] for coalescing read
32:                 for wy ← 1 to LOCAL_DIM_Y do
33:                     for wx ← 1 to LOCAL_DIM_X do
34:                         out_reg[wy][wx] += filter_reg * in_img_cache[wy + r][wx + s]
35:                     end for
36:                 end for
37:             end for
38:         end for
39:
40:     end for
41:
                                          ▷ transpose the output images for coalescing write may improve the speed
42:     for wy ← 1 to LOCAL_DIM_Y do
43:         for wx ← 1 to LOCAL_DIM_X do
44:             output[LOCAL_DIM * global_id.z + local_id][offset_y+wy][offset_x+wx] = out_reg[wy][wx]
45:         end for
46:     end for
47: end function
```

HNMTP convolution inherits all advantages of direct convolution, which is regarded as the most suitable convolution algorithm for single-image convolution algorithm for mobile GPUs. Meanwhile, our convolution algorithm solves the dilemma of whether or not to cache the filter. HNMTP convolution uses the same number of global memory access and registers with the caching schema without introducing shared memory barrier for convolution filter loading. Compared

with the caching schema, our convolution algorithm avoids shared memory barriers. There are `kernel_size` global memory reads and `kernel_size` times `workgroup_size` floating-point arithmetic instruction between the shared memory barrier for input image tile. The compiler has enough instruction scheduling feasibility to fusion the global memory operands and arithmetic operands to hide the latency. Even only with few warps, the ILP keeps the GPU busy and therefore achieves high GPU utilization.

On the other hand, different from the non-caching schema, HNMTP convolution does not introduce duplicated global memory instructions nor waste registers to store the same values. To pipeline N computing groups, HNMTP convolution only needs N register to store the filters, while the non-caching schema needs N times `kernel_size` registers. In other words, for the same number of registers, N times computing groups can be pipelined to hide the latency compared with the non-caching schema. Also, even with L2 cache, which relieves the burden of global memory bandwidth, the duplicated global memory read instructions of the non-caching schema still incurs instruction issuing overhead and L2 cache access latency.

5 Experiments

We ran speed experiments with ResNet, which is the most popular and state-of-the-art convolution neural networks. The first convolution layer, whose convolution filter is 7×7, is ignored in our experiments. All no-1x1 convolution layers in Reset have 3×3 convolution filters except the first one. There are only three kinds of no-1x1 convolution layers of ResNet and the difference between different ResNet is only the number of these convolution layers, which is summarized in Table 1.

Table 1: Convolution layers of ResNet

Layer	C × K	H × W	ResNet 18	ResNet 34	ResNet 50	ResNet 101	ResNet 152
conv2.x	64 × 64	56 × 56	2 × 2	2 × 3	1 × 3	1 × 3	1 × 3
conv3.x	128 × 128	28 × 28	2 × 2	2 × 4	1 × 4	1 × 4	1 × 8
conv4.x	256 × 256	14 × 14	2 × 2	2 × 6	1 × 6	1 × 23	1 × 36
conv5.x	512 × 512	7 × 7	2 × 2	2 × 4	1 × 3	1 × 3	1 × 3

The experiments are conducted on three typical platforms, which are mobile GPUs in Soc, integrated GPUs and dedicated GPUs. The detailed model and configurations are shown in Table 2.

Table 2: Experiment Devices Configuration

Model	Global Memory Type	Memory Bandwidth	CU	ALUs / CU	Total ALUs
AMD Radeon™ VII	HBM2	1024 GB/s	60	64	3840
AMD Radeon™ Vega 8	DDR4 single-channel	25 GB/s	8	64	512
Arm Mali-G76 MP10	LPDDR4 dual-channel	33.3 GB/s	10	24	240

5.1 Result

Figure 2 shows the execution times of convolution layers on different kinds of GPUs. It is quite surprising that the HNMTP convolution algorithm surpasses all other convolution algorithms in all convolution layers on all platforms. The thread-level parallelism is too insufficient when there is only one image, so convolution algorithms that provide higher instruction-level parallelism perform better.

As the *libdnn* convolution algorithm eliminates the global memory access overhead of the unrolled input matrix, it overperforms the *im2col* one on both integrated GPUs and mobile GPUs, whose memory bandwidth is quite limited. However, on dedicated GPUs, *im2col* is more than two times slower. It confirms our knowledge that most deep learning frameworks use *im2col* as the GPU convolution algorithm.

It seems quite strange that Winograd convolution performs worse on Radeno VII and Vega 8. The reason is that we serialize the group of matrix multiplications which can be combined into a single GPU kernel to take advantage of thread-level parallelism. However, it is safe to assume that the execution time ratio of Winograd convolution to *im2col* convolution should be similar to that of Mali-G76. Actually, the theoretical speed of Winogard convolution (F(2, 3×3)) to the *im2col* convolution algorithm is about $2.6\times$, which is still slower than direct convolution, let alone the HNMTP convolution.

In most of the experiments, the direct convolution algorithm is slightly better than the *libdnn* convolution algorithm. It should be noted that the convolution filters are not cached in shared memory in our implementation. These two

algorithms have around the same number of global memory accesses, but the direct convolution has higher instruction-level parallelism and less index calculation than the *libdnn*. However, the direct convolution needs substantial efforts to tune the GPU kernels. It is worth to pay efforts to tune the direct convolution GPU kernels for the performance improvement, but can it be better?

Of course, HNMTP convolution is much better than the direct convolution in terms of both the speed and tuning efforts. Especially on mobile GPUs and integrated GPUs, HNMTP convolution reduces the execution time by up to 50% and 46.6% compared with the direct convolution, respectively. On the least powerful platform, the Mali-G76 mobile GPUs, HNMTP convolution achieves speedup by $14.6\times$ of unrolling-based convolution (*im2col*), $3.1\times$ of unrolling-based convolution (*libdnn*), $10.7\times$ of Winograd convolution and $2.1\times$ of direct convolution. HNMTP convolution also reduces energy consumption as it has fewer global memory access. Meanwhile, tuning the HNMTP convolution GPU kernels is easier than that of the direction convolution ones, as the prior has fewer hyperparameters.

5.2 Profile

Meanwhile, we also profiled the most common convolution layer of ResNet, which has 256 input channels and output channels and each channel is a 14×14 images. The profile was conducted by codeXL on Vega 8 with Windows 10, as it is the only profiler that works. The result is shown in Table 3. It should be noted that the sgemm of Winogard convolution is executed 16 times.

Table 3: Profile Results

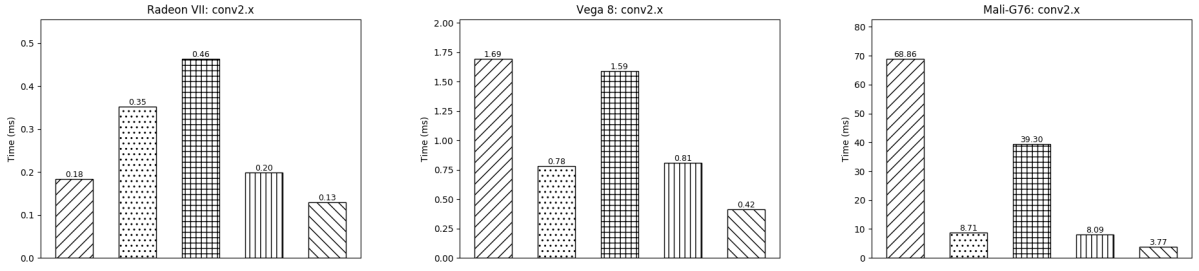
Algorithm	Kernel(s)	ThreadNum	LDS	GPRs(V+S)	VALU(%)	GMS(R+W)	FlatVMem
im2col Conv	im2col	196*128	0	15+17	94.4	102+893	18
	sgemm	32*256	4224	35+25	92.65	10182+98	1156
libdnn	conv	8*256	4480	75+26	86.54	9342+98	2320
Winograd Conv	TransFromS	512*64	1408	50+19	80.99	102+513	19
	sgemm (1 in 16)	16*256	4224	35+25	95.73	1065+32	98
	TransToD	128*256	0	38+19	100	517+98	18
Direct Conv	conv	32*64	640	118+27	98.53	9627+98	18704
HNMTP Conv	conv	32*64	640	103+31	97.32	9331+98	4944

LDS is the size of the shared memory of each workgroup in bytes, GPRs(V+S) is number of vector registers and scalar registers of each thread, VALU(%) is the percentage of the vector ALU utilization, GMS(R+W) is the total global memory read and write in kilobytes, and FlatMem is the number of FLAT instructions of each thread.

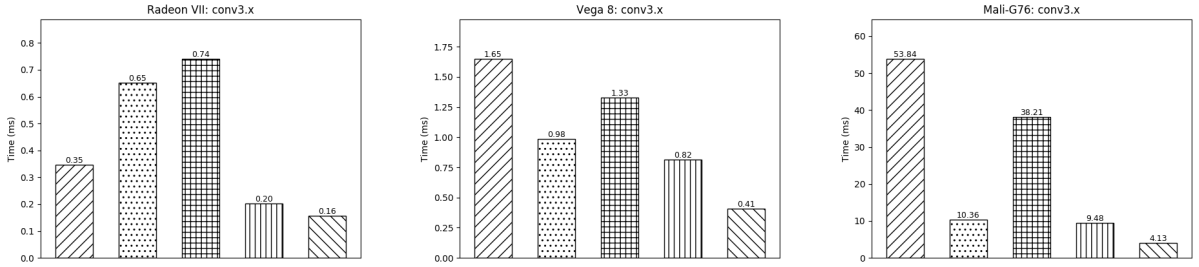
The the number of FLAT instructions of HNMTP convolution is much less the that of the direct convolution, as each thread of HNMTP convolution only loads its corresponding convolution filter. The usage of vector registers somehow reflects the parallelism. High vector registers usage usually indicates high ILP but low TLP, while low vector registers usage may suggest high TLP but low ILP. However, it still depends on the register usage before instruction pipeline and the number of warps. In this profile result, even both direct convolution and HNMTP convolution use about the same registers, the direct convolution has lower instruction-level parallelism due to its redundant storage of convolution filters.

6 Conclusion and Future Work

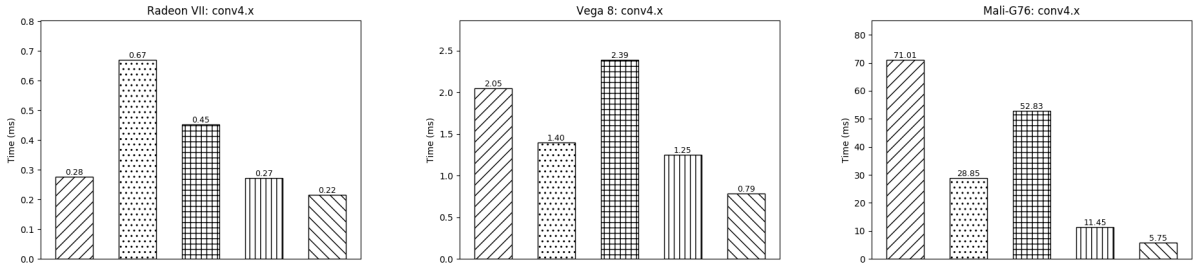
In this report presents the fastest convolution algorithm for single-image convolution neutral network inference on mobile GPUs. In the future, we are going to supports more tuning options, such as workload per threads and output coalescing write.



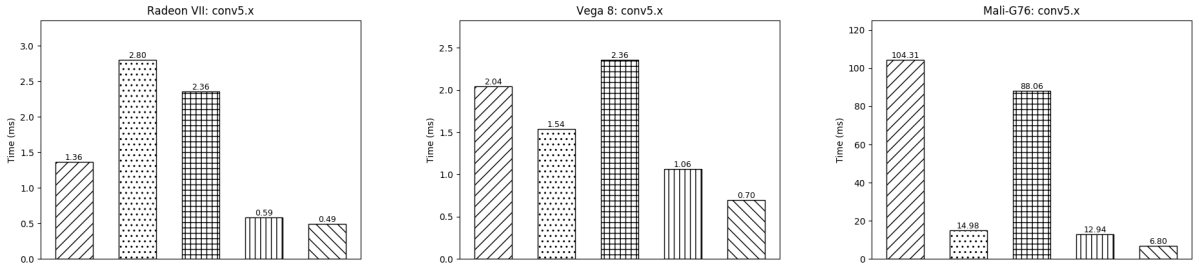
(a) conv2.x



(b) conv3.x



(c) conv4.x



(d) conv5.x

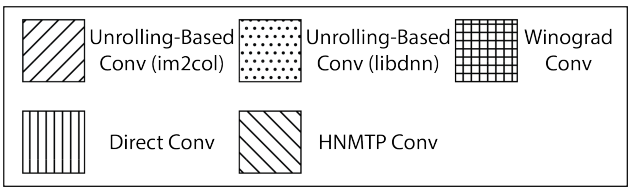


Figure 2

References

[1] Vasily Volkov. Better performance at lower occupancy. In *Proceedings of the GPU technology conference, GTC*, volume 10, page 16. San Jose, CA, 2010.

[2] Vasily Volkov. *Understanding latency hiding on gpus*. PhD thesis, UC Berkeley, 2016.

[3] Xu Han. Implement convolution in cnn. *http://xuhan.me/2016/09/09/conv/*, 2016.

[4] Tschopp. Opencl caffe. `https://github.com/BVLC/caffe/tree/opencl`, 2018.

[5] Fabian Tschopp, Julien NP Martel, Srinivas C Turaga, Matthew Cook, and Jan Funke. Efficient convolutional neural networks for pixelwise classification on heterogeneous hardware systems. In *2016 IEEE 13th International Symposium on Biomedical Imaging (ISBI)*, pages 1225–1228. IEEE, 2016.

[6] Andrew Lavin and Scott Gray. Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4013–4021, 2016.

[7] Alex Krizhevsky. cuda-convnet: High-performance c++/cuda implementation of convolutional neural networks. *Source code available at https://github. com/akrizhevsky/cuda-convnet2 [March, 2017]*, 7, 2012.