Optimizing Half Precision Winograd Convolution on ARM Many-Core Processors

Dedong Xie*
University of Toronto

Zhen Jia
Amazon Web Services

Zili Zhang
Peking University

Xin Jin
Peking University

ABSTRACT

Convolutional Neural Networks (CNNs) are widely used in real world applications, e.g, computer vision. Winograd based convolution is usually applied due to its low computation complexity. For the underling hardware, ARM many-core CPUs, by their price performance, are favored by cloud providers like Amazon Web Services (AWS). However, existing Winograd convolution implementations for ARM architecture are mostly optimized for mobile devices, and usually cannot fully utilize hardware resources of many-core processors. In this paper, we propose HAWC, an optimized half precision floating-point (FP16) Winograd convolution implementation for ARM many-core processors. HAWC employs a series of optimization methods, which are suitable for ARM NEON architecture, and assembles them as an entire solution to improve performance. Our evaluation shows that HAWC achieves on average 10.74× and up to 27.56× speedup on representative convolution layers over state-of-the-art solutions.

CCS CONCEPTS

• **Theory of computation → Massively parallel algorithms**; • **Computing methodologies → Neural networks**; • **Software and its engineering → Just-in-time compilers**.

KEYWORDS

Convolution, Winograd, Parallelization, Vectorization

ACM Reference Format:
Dedong Xie, Zhen Jia, Zili Zhang, and Xin Jin. 2022. Optimizing Half Precision Winograd Convolution on ARM Many-Core Processors. In *13th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '22), August 23–24, 2022, Virtual Event, Singapore.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3546591.3547529

1 INTRODUCTION

Convolutional Neural Networks (CNNs) are widely used in real application scenarios including recommendation and recognition [15, 22, 26]. CNNs can effectively excavate the hidden meaning of the input image through the convolution layers and have become one of the most actively researched and applied models in deep learning.

Cloud providers are always focusing on providing flexible and efficient services to customers. Besides GPUs, CPUs can also run deep learning inference. To achieve high price performance, ARM many-core processors are becoming one of the primary choices. For instance, AWS has released Graviton ARM-based many-core processors, which are designed to have high price performance [11, 27].

From the perspective of algorithms, low precision computation can be used to increase the computation speed. Previous studies have demonstrated that low precision computation can greatly reduce memory footprint and provide faster computations at low or even no accuracy loss for neural networks [2, 18, 30]. On the other hand, Winograd based convolution [29] was recently proposed as a state-of-the-art solution due to its ability to reduce the number of operations. Since then, it has attracted a lot of follow-up studies to optimize it on diverse hardware. A combination of low precision and Winograd can bring further performance improvements.

Half precision computation, a sub-class of low precision computing, is well supported in ARM architecture and there are many open source libraries, e.g, NCNN [25] and MNN [12], which employ half precision (i.e., FP16) Winograd based convolution. However, existing work on ARM is mostly optimized for multi-core mobile CPUs, not ARM many-core processors. In some situations, Winograd based convolution could be slower than direct convolution, even though Winograd convolution has lower computation complexity than direct convolution (§2.1). This is because the increasing number of cores aggravates memory system's pressure and hurts the performance of I/O bound stage in Winograd algorithm. So a multi-core optimized implementation may not scale well on a many-core platform.

We propose HAWC: an efficient Half precision, ARM many-core processor optimized, Winograd Convolution implementation. HAWC consists of three stages: 1) input and kernel transformation, 2) matrix multiplication and 3) output transformation. Among these, input, kernel and output transformations are memory bound, while matrix multiplication is computation bound. We employ a custom data layout to fully vectorize all the computation and maximize data re-use. In order to hide the memory operation latency, we overlap the computation and data transformation across stages. A custom matrix-matrix multiplication kernel (GEMM) is implemented to achieve high resource utilization on ARM many-core CPUs. Finally, we employ a static job scheduler to assign each job at compilation time to reduce runtime overhead and balance the workload on each

*Work done during Dedong's internship at AWS.

APSys '22, August 23–24, 2022, Virtual Event, Singapore

ACM ISBN 978-1-4503-9441-3/22/08…$15.00
https://doi.org/10.1145/3546591.3547529

thread. We combine these optimizations into a coherent system to optimize the performance of Winograd convolution.

Our contributions are summarized as below:

- We propose HAWC, an efficient implementation of FP16 Winograd convolution optimized on ARM many-core processors. We apply various optimizations to achieve high performance.
- We design a custom JIT-compiled matrix multiplication kernel for Winograd convolution to take fully advantage of ARM NEON ISA.
- We perform comprehensive evaluations on Graviton 2 platform. The experimental results show that our implementation achieves acceptable accuracy with on average 10.74× and up to 27.56× speedup on representative convolution layers.

2 BACKGROUND AND MOTIVATION

2.1 Winograd-based Convolution

Convolution is a mathematical function that slides a kernel along the input image to excavate the local meaning of the input image. In each individual step, a sub image and kernel are element-wise multiplied and accumulated to produce the final output value [7]. For direct convolution, computing a size m output with a size r kernel requires $m \times r$ multiplications. Winograd based convolution [29] shows an opportunity that requires only $m + r - 1$ multiplications. A 2D Winograd based convolutions between kernel K of size $r \times r$ and input image I of size $(m + r - 1) \times (m + r - 1)$, generating an output O of size $m \times m$ is denoted as $F(m \times m, r \times r)$ and can be expressed as:

$$O = A^T[(GKG^T) \odot (B^T IB)]A \tag{1}$$

where $\odot$ represents element-wise multiplication and matrices A, G, B are transformation matrices determined by Chinese remainder theorem [14, 29]. With the transformations, the convolution can be done efficiently by taking the common parts of the calculation, then storing the intermediate results and re-using them to reduce the number of multiplications needed. More details of Winograd based convolution can be found in [29].

By dividing the input image into overlapping tiles and applying the overlapping-add (OLA) method [21], all convolutions with kernel of size $r \times r$ can be calculated through the same algorithm of $F(m \times m, r \times r)$. For multi-channel convolutions, the accumulation of output from element-wise multiplications along the channel dimension is equivalent to a matrix multiplication.

The Winograd based convolution can be transformed to a process consisting of three stages: 1) input and kernel transformation, 2) matrix multiplication, and 3) output transformation.

2.2 Half precision Arithmetic on ARM

Previous work [2, 18, 30] has demonstrated that for neural networks, half precision floating point (FP16) computation is stable with no significant accuracy loss and can reduce memory footprint. This motivates us to take advantage of FP16 data format to perform Winograd convolution: FP16 instruction doubles the speed of computation and halves the size of data movement, which benefits both computation bound and I/O bound stages in Winograd algorithm.

Procedure 1: FMLA V1.8H, V2.8H, V3.8H

input : Three vector registers V1, V2, and V3

```
for i ← 0 to 7 do
    V1[i] += V2[i] * V3[i];
end for
```

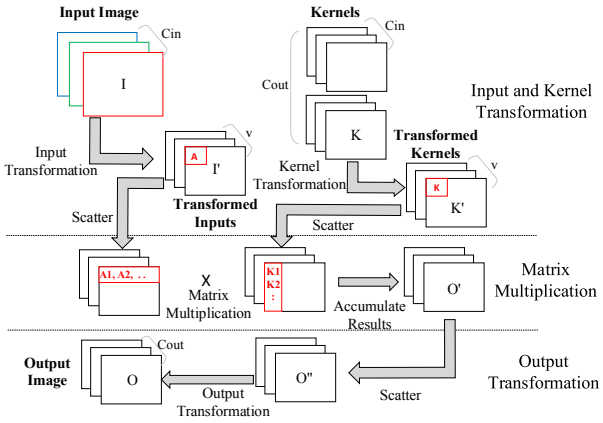

Figure 1: The workflow of our algorithm.

ARM NEON ISA [5] provides the instructions operating on half precision floating point data. On ARM Neoverse-N1 core, a set of 32 128-bits long vector registers named by V0 to V31 is used to achieve SIMD operations on different sizes of data. For FP16 operations, each register can hold 8 lanes of half precision floating point data. Fused multiply-add (FMLA) instruction is also supported in the ISA so the theoretical FLOPS is doubled. A single SIMD instruction's semantics are shown in Procedure 1. Apart from register-wise operations, operations on a single lane or all lanes are supported. The FMLA can be applied to one lane in the third operand to produce scalar by-element multiply-add.

3 DESIGN

This section introduces the overall design of HAWC, which provides optimized FP16 Winograd based convolution implementation for ARM many-core processors. We first describe the data layout (§3.1), which decides how we store data. We then introduce the implementation and optimizations in each stage including input and kernel transformation (§3.2), matrix multiplication (§3.3) and output transformation (§3.4) as illustrated in Figure 1. Finally, we introduce the stage-wise scheduler to achieve efficient parallel processing (§3.5).

3.1 Data Layout

Data layout is crucial for performance as it determines how data is stored, accessed, and used. In this subsection, we present the custom data layout of HAWC, inspired by state-of-the-art implementations on a different platform [10, 31]. The data layout is listed in Table 1 where the elements in brackets represent the corresponding dimensions. The layout is designed for three main requests: apply vectorization, maximize data re-use, and gain low-latency memory access.

Variable	Symbol	Data Layout
input Image	$I_{b,c_{in}}$	$[B][\frac{C_{in}}{v}][D_{in}][H_{in}][W_{in}][v]$
Transformed inputs	$I'_{b,c_{in},t}$	$[\frac{T \cdot B}{Tb}][\frac{C_{in}}{Cb}][T_{size}][Cb][Tb]$
Kernels	$K_{c_{in},c_{out}}$	$[C_{in}][\frac{C_{out}}{v}][D_K][H_K][W_K][v]$
Transformed Kernels	$K'_{c_{in},c_{out}}$	$[\frac{C_{in}}{Cb}][\frac{C_{out}}{Cb'}][T_{size}][Cb][Cb']$
Intermediate Matrices	$O'_{b,c_{out},t}$	$[\frac{T_n \cdot B}{Tb}][\frac{C_{out}}{Cb'}][T_{size}][Tb][Cb']$
Pre-Transform Outputs	$O"_{b,c_{out},t}$	$[B][\frac{C_{out}}{v}][T_{size}][D_T][H_T][W_T][v]$
Output Image	$O_{b,c_{out}}$	$[B][\frac{C_{out}}{v}][D_{out}][H_{out}][W_{out}][v]$

Table 1: Data layout.

To support vector arithmetic, we use a packed data layout that packs $v = 8$ channels of an image into continuous data segments. The selection of 8 is based on the following facts: vector registers of ARM architecture accommodates 8 lanes of FP16 data; modern CNNs are designed to have convolution layers' channel number divisible by 16, and hence the channel number is also divisible by 8. With this technique, a batch of input images with each image of C_{in} channels are stored in a row-major array of size $B \times (C_{in}/v) \times D_{in} \times H_{in} \times W_{in} \times v$. Similarly, the kernels are also stored in a row-major array of size $C_{in} \times (C_{out}/v) \times D_k \times H_k \times W_k \times v$. Furthermore, the same pattern applies to output images.

To increase data re-use and reduce access latency, we break the transformed inputs and transformed kernels into sub-matrices of size $Tb \times Cb$ and $Cb \times Cb'$, respectively. The determination of blocking sizes is explained in §3.3. Such strategy will support the calculation of matrix multiplication to be done in small blocks that can be fully stored in cache and stay in cache as long as possible until all computations they involve are done. This will increase locality and make use of caching to ensure fast data access.

3.2 Input and Kernel Transformation

The first stage of Winograd-based convolution is to apply the transformation on the input image I (Figure 1). The transformation computes the result of $B^T IB$. As described in §2.1, we divide the input image into overlapping tiles and apply the same transformation to each tile. In HAWC, the transform matrix B is generated by Wincnn [13]. During the transformation, we re-use intermediate values to accelerate the computations. Apart from this, the input and output transformations are coded in NEON SIMD intrinsic [6] to apply vector computations.

To further reduce the compilation and execution overhead, we code the desired transformation on the input tiles using c++ templates with respect to variables m and r , so only the transformation codelet of the specific $F(m, r)$ is compiled and executed. Similar process is applied to kernel transformation. For inference tasks, the kernels are pre-transformed to further reduce overall latency.

Another optimization involved in input transformation is the scattering of transformed tiles to sub-matrices of transformed inputs and kernels. Instead of storing the transformed tiles, we scatter the tiles to their position in corresponding sub-matrices as described in the data layout of transformed inputs and transformed kernels. By doing so, each sub-matrix from transformed inputs and transformed kernels will contain the tiles needed for matrix-matrix multiplication in continuous memory space. This arrangement will then reduce the overhead for matrix-matrix multiplication stage as data of sub-matrices is localized.

3.3 Matrix Multiplication

In this stage, we perform batched matrix-matrix multiplication between transformed inputs (I') and transformed kernels (K'). There are total $(m+r-1) \times (m+r-1)$ matrix multiplications. However, the transformed inputs (I') and transformed kernels (K') are usually tall and skinny matrices so that they can not fit into cache. Forcing transformed inputs and kernels into cache will result in poor data locality. In order to increase data re-use and further optimized the matrix multiplication efficient, we performance matrix blocking on the transformed inputs and transformed kernels. The transformed inputs of size $TB \times C_{in} \times T_{size}$ is divided into $\lceil \frac{TB}{Tb} \rceil \cdot \lceil \frac{C_{in}}{Cb} \rceil \cdot T_{size}$ sub-matrices, $U_{i,j}$, each of size $Tb \times Cb$. Similarly, transformed kernels is divided into $\lceil \frac{C_{in}}{Cb} \rceil \cdot \lceil \frac{C_{out}}{Cb'} \rceil \cdot T_{size}$ sub-matrices, $V_{j,k}$, each of size $Cb \times Cb'$. In order to get the correct result, we need to perform matrix multiplications between sub-matrix U and V. For the sub-matrix multiplication loop, we reuse the sub-matrix V and load different Us for each computation. To be specific, after we finish $U_{i,j} \times V_{j,k}$, we will execute $U_{i+1,j} \times V_{j,k}$. By caching the $V_{j,k}$ we eliminate the unnecessary memory accesses.

JIT compiled matrix multiplication kernel. We employ just-in-time (JIT) compile technique to generate assembly code to perform matrix multiplication for different matrix shapes. This design gives us the flexibility to generate code for diverse matrix blocking and register blocking strategies through using different blocking parameters. At the same time, coding at assembly level enables us to tune for the specific hardware to achieve higher performance.

For each single pair of sub-matrix multiplication between transformed inputs and transformed kernels, $U \times V = Y$, we apply register blocking, where the inner loop computation (Procedure 2) is between a block of size $Tb \times 8$ of U and a block of size 8×8 of V. The sizes of these blocks are determined by the need of fitting data into vector registers: 8 lanes of FP16 data in each vector register. We use the notation of registers $U(n), V(n), Y(n)$ as shown in Figure 2 to represent the registers holding data of different matrices. This matrix multiplication is illustrated by Procedure 2. We first load a block of size $Tb \times 8$ from U to Tb vector registers. In addition, we load a vector register with 8 FP16 data in a row from V. Then we loop through all Tb registers to execute FMLA on data from V then accumulate to Y. Figure 2 shows the execution of one FMLA instruction in the procedure. We repeat the previous steps through all the 8 result registers $Y(n)$ until Y is computed. Here we unroll all the computation and memory access loops so that we will have a large number of continues computation instructions or store instructions that can be pipelined so that the latency is hidden.

As for the efficient memory access during the multiplication procedure, we use the method of pre-set offsets with ARM's post-index offset load and store. The accessed memory position for each register can be calculated based on the position by adding constant offsets. Therefore, we are able to use general purpose registers to store the memory offset for the 1st, 3rd, 5th row of the matrix and base registers to store the position of the 1st and 7th row. Through this mechanism, the memory position to load/store can be described by base register, offset register, and shifting factor in a

Procedure 2: Unit Multiplication of Matrices

```
for i ← 0 to 7 do                                   ▷ unrolled
    for j ← 0 to Tb − 1 do                          ▷ unrolled
        if i == 0 then
            Load U(j).8H
        end if
        FMLA Y(j).8H, V(i%4).8H, U(j).H[i]
    end for
    Load V((i+1)%4).8H
end for
for k ← 0 to Tb − 1 do                              ▷ unrolled
    Store Y(k).8H
end for
```

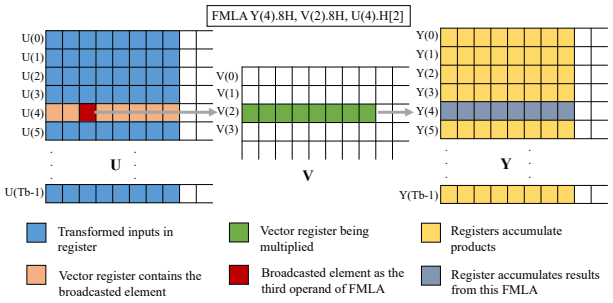

Figure 2: Execution of one FMLA instruction in our JIT compiled matrix multiplication.

single instruction. With the aforementioned computation of offsets integrated in one instruction occupying one clock cycle on ARM processors, we save extra instructions and clock cycles. Compared to calculating the memory location through multiplication, shift, and addition, this will reduce the required clock cycles for each store or load from 3 to 1.

Another ARM-specific instruction adopted by HAWC is the post-index version of loading 1-element structure to register, LD1. This instruction loads from memory and then updates the base register by adding an offset. The use of this instruction greatly saves the computation time when HAWC works on matrices. This is because matrix has continuous data layout with fixed offsets between cells and rows. The load of an entire row or column of n elements can hence be implemented with n post-index LD1, while manually updating base register requires n extra additions.

Blocking size determination. We follow the guidelines described in [10, 16], to determine our blocking parameters. According to the Roofline model [28], we would like to let our implementation have an arithmetic intensity (operations per moved byte) as high as possible to make sure the performance is not bounded by memory I/O. We calculate the arithmetic intensity as follows: 1) The number of floating-point computation needed for each matrix multiplication is $2TbCbCb'$ (computation performed between a matrix of size of $Tb \times Cb$ and a matrix of size $Cb \times Cb'$, generating a matrix of size $Tb \times Cb'$). 2) In this matrix multiplication procedure, we use one V to multiply with all the corresponding Us and accumulate the results to Ys. This procedure can be expressed as $Y = \gamma Y + U \times V$. Only in the first iteration of sub-matrices multiplication, we can set $\gamma = 0$. For all the other cases, we need to set $\gamma = 1$ as we need to accumulate the result. So, the majority of cases is when $\gamma = 1$ and we have the procedure being $Y = Y + U \times V$. Hence, in each pass of the procedure, we need to load U and the corresponding Y to accumulate results. V is cached and can assume the number of loaded bytes is amortized. This takes $TbCb + TbCb'$ loads. In addition, by the end of one pass of the procedure, we have to store Y, yielding $TbCb'$ stores. In total, there are $Tb(2Cb' + Cb)$ FP16 data load/store, which is $2Tb(2Cb' + Cb)$ bytes. 3) Therefore, the arithmetic intensity AI can be calculated by dividing FLOPS by bytes of memory traffic: $AI = \frac{2TbCbCb'}{2Tb(2Cb'+Cb)} = \frac{CbCb'}{2Cb'+Cb}$.

As illustrated in the Roofline model [28], when the application's arithmetic intensity is higher than the compute-to-memory ratio of the platform, the application falls in the ideal situation when the performance is compute-bound instead of memory I/O bound. The Graviton 2 platform's compute-to-memory for FP16 data is 25.6 (5120 GFLOPS with memory bandwidth of 200 GBytes/s [27]). When $Cb = 64, Cb' = 128$, the compute-to-memory ratio is 32, higher than the hardware's compute-to-memory ratio. When $Cb = Cb' = 64$, the compute-to-memory ratio is only 21.33, lower than the threshold, meaning the matrix multiplication will be memory bound. From the previous analysis, we know that the greater Cb and Cb', the higher the arithmetic intensity could achieve. However, the parameters for sub-matrices, Cb, Cb', and Tb should also follow the following restrictions:

- C_{in} should be divisible by Cb and C_{out} should be divisible by Cb'. Otherwise, zero padding is needed and will cost extra unnecessary computations.
- Both Cb and Cb' should be divisible by 8. This is to align data with the 8 lanes of FP16 data in ARM vector registers.
- Tb should be no greater than 14. As Tb corresponds to the number of registers we can use for transformed inputs in JIT-compiled matrix multiplication.
- Matrix U,V, and Y should fit into cache. So, there is no data movement back and forth between cache and memory.

As for the selection of Tb, higher Tb increase the number of FMLA instructions to be pipelined, which could help hide the instruction latency. However, large Tb may also decrease Cb and Cb' as we need fit U, V and Y into cache. Also, when Tb does not divide TB, we need to perform zero padding, increasing the number of operations. Therefore, the determination of Tb cannot be easily calculated. On the other hand, as Tb has a relatively small search space (from 1 to 14), empirical determination is available at low cost of time. For this reason and the fact that layer dimensions are given in advance, we empirically determine the best value of Tb through benchmarking.

Scattering of matrices' product. The challenge of output transformation stage is the time-consuming gathering of tiled data from different matrices. The gathering triggers non-continues data access from different matrices, resulting in poor data locality. Furthermore, this situation gets even worse as tile size increases. Inspired by [10], instead of storing the results from matrix multiplication continuously, we scatter the results (i.e., Y) to the locations in the tiles of

output image. By using scattering at the end of matrix multiplication stage, we reform tiles in memory so that output transformation stage read data continuously.

On x86 platform [10, 17], the best performance of such scattering strategy is achieved by using streaming store that writes to memory without caching. This technique eliminates the overhead caused by scattering store. While there exists streaming store instruction in ARM, the store a pair of registers with non-temporal hint (STNP), ARM A64 ISA [1] only supports general purpose register as its operands. In order to use streaming store, we must move the data from vector register to general purpose registers before each store of pairs of registers. This introduces two extra instructions and triples the number of instructions for a single store. Such move and store is inefficient, requiring even more cycles than it could save compared to normal store. Therefore, HAWC uses normal store with scattering, instead.

3.4 Output Transformation

Apart from the aforementioned optimization of data scattering after matrix multiplication for efficient memory access, HAWC performs the output transformation with the same set of optimizations for input transformations in §3.2.

3.5 Stage-wise Static Parallel Scheduling

To minimize scheduling overhead and achieve high level of parallelism through multi-threading, HAWC uses a static scheduler. This is possible as the parameters of the layer and the number of cores to be used are determined before actually running the network. To further reduce the cost of thread scheduling and management, HAWC optimizes the complexity of the scheduler and minimizes the number of barriers set for synchronization. Inspired by previous work [10, 19], instead of using widely-used OpenMP [4], we implement a simplified scheduler with c++ atomics that applies busy-wait strategy for synchronization. Besides, instead of implementing parallel on loop, we apply multi-threading on stage level. In our implementation, only three barriers are set at the end of each stage. Through this strategy, the number of barriers is minimized, and the latency caused by rapid synchronizations is reduced.

Workload balance on different threads is another challenge in achieving high performance for a parallel system. Through dividing inputs of each stage to tiles or blocks, all data in HAWC is in the form of continuous memory chunks. As all chunks of data are of the same size with unified data layout, we evenly assign the chunks to threads. In the ideal situation, all threads will finish at the same time and hence achieve perfect parallelism.

4 EVALUATION

4.1 Experiments Setup

Instance configuration. We perform the evaluation on Graviton 2 ARM many-core platform. We use an AWS m6g.metal instance, deployed with Ubuntu 18.04 (18.04.6 LTS) and configured with 256 GB 8-channel DDR4 memory with a bandwidth of over 200 GBytes/s [27]. The AWS m6g.metal instance has 64 cores; each runs at a frequency of $2.5GHz$. The SIMD-length is 128 bits, so for FP16 computation with fused multiply add (FMLA) support, the theoretical FLOPS (Floating point Operations Per Second) is 80 G for a single core and 5.12 T for the entire instance.

Layer	C_{in}	C_{out}	Input Size	Kernel Size
VGG 1.2	64	64	$< 224, 224 >$	$< 3, 3 >$
VGG 2.2	128	128	$< 112, 112 >$	$< 3, 3 >$
VGG 3.2	256	256	$< 56, 56 >$	$< 3, 3 >$
VGG 4.2	512	512	$< 28, 28 >$	$< 3, 3 >$
VGG 5.2	512	512	$< 14, 14 >$	$< 3, 3 >$
FusionNet 1.2	64	64	$< 640, 640 >$	$< 3, 3 >$
FusionNet 2.2	128	128	$< 320, 320 >$	$< 3, 3 >$
FusionNet 3.2	256	256	$< 160, 160 >$	$< 3, 3 >$
FusionNet 4.2	512	512	$< 80, 80 >$	$< 3, 3 >$
FusionNet 5.2	1024	1024	$< 40, 40 >$	$< 3, 3 >$

Table 2: Parameters of layers benchmarked.

VGG	Direct	$F(2\times2, 3\times3)$	$F(4\times4, 3\times3)$	$F(6\times6, 3\times3)$	$F(6\times8, 3\times3)$
Max	1.33E-4	2.83E-2	1.54E-2	2.21E+1	4.25E+3
Avg	5.63E-6	5.83E-4	4.19E-4	6.43E-2	2.56E+1

Table 3: Element errors in convolution layers.

Convolution layers and comparisons. We evaluate HAWC on representative convolution layers on prevalent CNN models: VGG[23] and FusionNet[20]. Table 2 summarizes the detailed configurations of the convolution layers used in our experiments. We compare HAWC with two widely used open-source commercial ARM optimized Winograd implementations: MNN [12] and NCNN [25]. We also compare with direct convolution and GEMM based convolution im2col (performing image to column before GEMM).

4.2 Accuracy

Because of Winograd algorithm's numerical instability [14], the result derived from Winograd algorithm is not identical to the direct or GEMM based convolution. In general, the larger the m of $F(m, r)$ is used, the more operations Winograd algorithm can save but at the higher precision loss. In addition, half precision computation, as a result of reduced precision bits, exacerbates the numerical instability. Therefore, it is important to guarantee that the precision loss is acceptable to make sure the method gives meaningful convolution result. We compute the maximum error and average error of all layers in VGG with different Winograd algorithms (i.e., $F(m, r)$).

The errors are calculated by the absolute difference between the result of HAWC (half precision Winograd) and the ground truth calculated by direct convolution with long double typed data. The input images are retrieved from a uniform distribution on $[-0.1, 0.1]$, while the kernels are generated by Xavier initialization [8].

Table 3 shows the max element difference and average element difference. According to previous studies [3, 9], an average error of $E-2$ will not affect the stability of training and the inference can tolerate a magnitude of higher order errors.

From the results, we can see that Winograd $F(2\times2, 3\times3)$, $F(4\times4, 3\times3)$, $F(6\times6, 3\times3)$ fall in the range of $E-2$ or lower in average error. Hence, we can conclude that in HAWC, Winograd $F(2\times2, 3\times3)$, $F(4\times4, 3\times3)$, $F(6\times6, 3\times3)$ are capable to be used without stability issues.

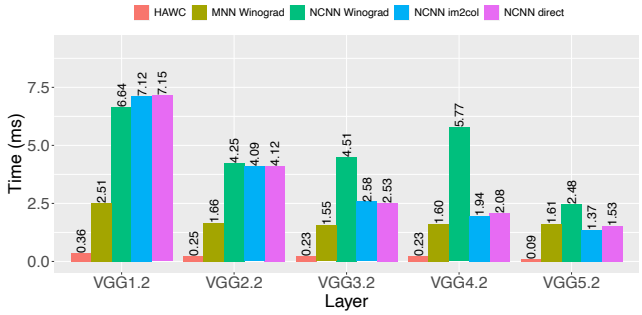

Figure 3: VGG on batch size 1.

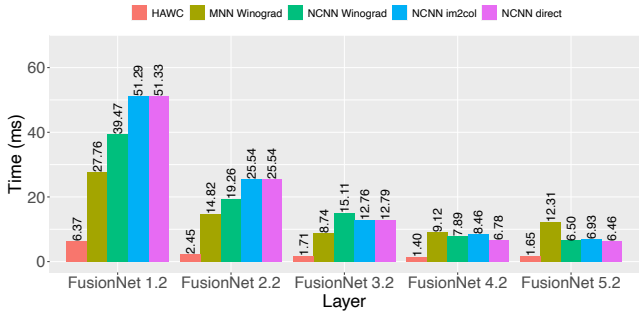

Figure 4: FusionNet on batch size 1.

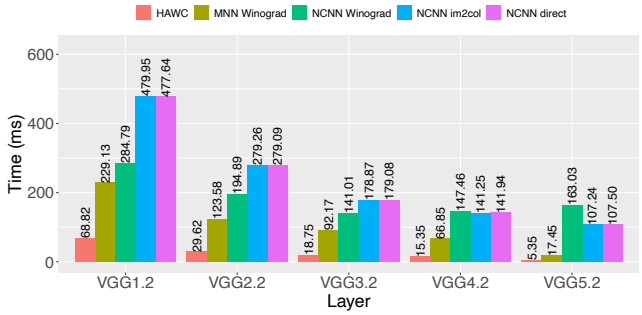

Figure 5: VGG on batch size 64.

4.3 Performance

In this set of experiments, we first investigate HAWC's latency on the representative convolution layers and perform comparisons with the state-of-the-art implementations. Then we evaluate our JIT based matrix multiplication's performance. Finally, we investigate each stage's latency by performing a execution time breakdown.

To evaluate HAWC, we benchmark it on Graviton 2 platform with the convolution layers shown in Table 1. Both MNN [12] and NCNN [25] choose the Winograd algorithms (i.e., which $F(m, r)$ is used) automatically with their own heuristics. We perform further investigation on the algorithm they used and find for all the layers except VGG 5.2, they choose $F(6 \times 6, 3 \times 3)$ as their FP16 Winograd implementations. For VGG 5.2, both use $F(4 \times 4, 3 \times 3)$. We use the same Winograd algorithms as MNN and NCNN. To show Winograd's superiority, we also benchmark direct convolution and GEMM based convolution (im2col), which are provided by NCNN.

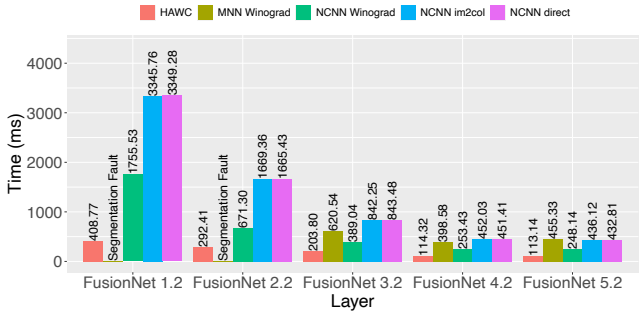

Figure 6: FusionNet on batch size 64.

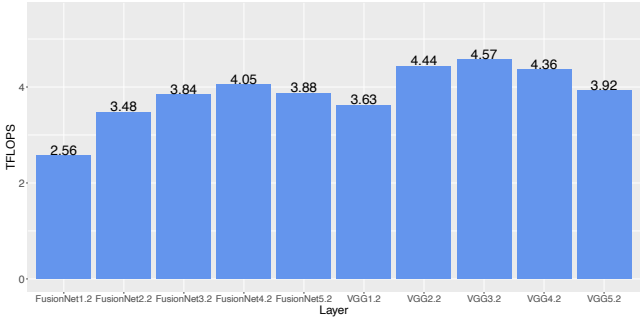

Figure 7: GEMM FLOPS.

Figure 3 shows the latency of single batch VGG layers on Graviton 2 platform. The result shows that HAWC outperforms NCNN by a factor of on average 21.54× and up to 27.56×. For MNN, HAWC achieves 9.04× speedup on average and up to 17.89× speedup. Compared to GEMM based method, i.e., NCNN im2col, HAWC achieves on average 14.20× and at most 19.78×. With respect to direct convolution, HAWC achieves on average 14.68× and at most 19.86×.

FusionNet results are shown in Figure 4. Comparing with NCNN, HAWC achieves on average 6.49× speedup and up to 8.84× speedup. HAWC is on average 5.90× and up to 7.46× faster than MNN. When we perform the comparison between Winograd convolution and direct (GEMM based) convolution, we can see HAWC is better than direct (GEMM based) convolution. For NCNN and MNN, there are situations where Winograd implementations take longer time than direct (GEMM based) counterparts, for instance VGG3.2, VGG 4.2, FusionNet 5.2 etc, even though Winograd has a huge theoretical speedups. Those phenomena also indicate NCNN and MNN Winograd implementations need to be optimized on many-core ARM platform.

We also benchmark multi-batch VGG and FusionNet. The results are shown in Figure 5 and Figure 6. We can find similar phenomena: HAWC outperforms NCNN and MNN by a large factor. For Fusion 1.2 and Fusion 2.2, MNN reports segmentation fault, which indicates the limited multi-batch support in MNN.

GEMM performance. To evaluate our custom GEMM, we calculate FLOPS of the GEMM stage for each layer. Figure 7 shows the FLOPS achieved by our GEMM. Our GEMM can achieve up to 4.57 TFLOPS, which is around 90% of the theoretical FLOPS of Graviton 2 platform (5.12 TFLOPS). For most layers, our GEMM achieves

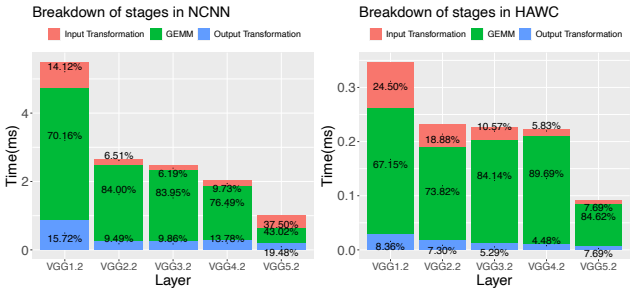

Figure 8: Stage latency breakdown

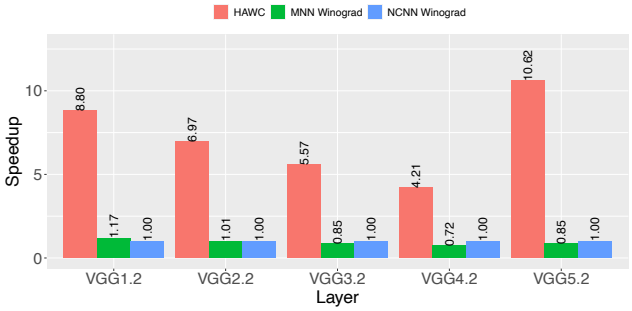

Figure 9: VGG on Graviton 3 relative speedup

above 80% of the theoretical FLOPS. There are some layers that their GEMM stage does not achieve good FLOPS. There are two reasons. First, in matrix multiplication stage, the matrix size is decided by the input and output channel. If the channel number is relatively small, Cb and Cb' are bounded by the channel number. This will result in the matrix multiplication having low arithmetic intensity and yields poor performance since in this situation, performance is bound by I/O [28]. Second, in our matrix blocking implementation, the actual matrix size may not always be dividable by the blocking factor. We perform zero padding for that case. Such padding will result in extra non-necessary computation.

Execution time breakdown. We investigate the execution time of each stage in HAWC and NCNN. MNN applies fused Winograd algorithm, which performs all the transformations and computation on each input partition in a single thread, and there is no clear boundary between stages. Therefore, we do not perform the breakdown of stages for MNN.

Figure 8 shows the execution time breakdown. Compared to NCNN, HAWC's output transformation's proportion is much less than NCNN. This is because we perform the scatter at the end of matrix multiplication stage, which achieves continuous memory access in output transformation. The scatter may increase store instruction latency due to irregular memory access, but it does not hurt much on the overall matrix multiplication performance as shown previously. We also scatter the result at the end of input transform stage so that the matrix multiplication stage can achieve continuous memory access. The input transformation may take more proportion for some layers. The constraint of streaming store in ARM A64 ISA, is the streaming store instruction only supports general-use registers and relies on register-level copy. Such constraint causes two extra instruction cycles. This makes streaming store does not benefit from vector operations. Otherwise, the input transformation may take much less time.

4.4 Case Study: Graviton 3

Amazon released the newly built Graviton 3 instances and opened it to public recently (May, 2022) [24]. The Graviton 3, compared to Graviton 2, provides more features e.g., BF16 support, large SIMD width, and higher performance. We perform the benchmark with VGG layers on an AWS c7g.16xlarge instance (powered by Graviton 3) and compare HAWC with MNN and NCNN FP16 Winograd implementations.

The results are summarized in Figure 9. We calculate the speedups with respect to NCNN. The results show that HAWC achieves on average 7.55× and up to 10.52× speedup compared to the baseline. For some layers, MNN is better than NCNN, and for other layers, NCNN beats MNN. While, at the same time, none of them can achieve similar performance as HAWC.

We remark that HAWC is mostly optimized for ARM Neoverse N1 cores (i.e., Graviton 2 architecture). ARM Neoverse V1 cores (i.e., Graviton 3) are the next generation architecture when we performed this study. However, the ideas and optimizations in HAWC are orthogonal to the underline hardware architecture. HAWC is easy to be extended to the ARM Neoverse V1 cores. The current implementation uses a fixed size of SIMD width, i.e., 128, on Neoverse N1. The SIMD width decides how we design the data layout and how to perform resister blocking in matrix multiplication. So when it comes to Neoverse V1 with a SIMD width of 512, direct migration of HAWC can not fully utilize the hardware resources. This is the reason why HAWC achieves less speedups on Graviton 3 platform compared with NCNN and MNN than that on Graviton 2. To optimize on ARM Neoverse V1 cores, we need to adjust the register blocking size and data layout to utilize the large SIMD width. In our future work, we plan to employ a platform aware mechanism to choose the data layout and register blocking method for HAWC.

5 CONCLUSION AND FUTURE WORK

In this paper, we present HAWC, a system that implements optimized half precision Winograd based convolution for ARM many-core processors. We employ a series of optimizations including customized data layout suited for ARM NEON ISA, cache and register blocking matrix multiplication, static scheduling etc. With these techniques, HAWC can significantly outperform existing implementations by 21.54× on average and up to 27.56× on Graviton 2 platform. We can achieve an up to 12.49× speedup on Graviton 3 platform, even though we did not specifically optimize for it.

In the future, we will extend HAWC to better support Graviton 3, which has wider SIMD width and will affect our cache and register blocking strategies. We will provide a uniform implementation to cover most SIMD width on diverse ARM platforms. Graviton 3 also supports BF16 instead of only FP16, for which we also plan to support it. Besides those, one possible improvement is to implement an auto-tune mechanism to jointly choose the best Winograd algorithm, data layout, and GEMM configurations combination according to layer parameters.

REFERENCES

[1] ARM. 2021. *Arm A64 Instruction Set Architecture.* Retrieved 2022-07-14 from https://developer.arm.com/documentation/ddi0596/2021-12/Base-Instructions/STNP--Store-Pair-of-Registers--with-non-temporal-hint-?lang=en

[2] Zehua Cheng, Weiyang Wang, Yan Pan, and Thomas Lukasiewicz. 2019. Distributed Low Precision Training Without Mixed Precision. https://doi.org/10.48550/ARXIV.1911.07384

[3] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. 2014. Training deep neural networks with low precision multiplications. *arXiv preprint arXiv:1412.7024* (2014).

[4] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE* 5, 1 (1998), 46–55.

[5] ARM Developer. 2013. *NEON Programmer's Guide.* Retrieved 2022-05-06 from https://developer.arm.com/documentation/den0018/a/?lang=en

[6] ARM Developer. 2022. *NEON Intrinsic.* Retrieved 2022-05-06 from https://developer.arm.com/architectures/instruction-sets/intrinsics/#f:@navigationhierarchiessimdisa=[Neon]

[7] Vincent Dumoulin and Francesco Visin. 2016. A guide to convolution arithmetic for deep learning. *arXiv preprint arXiv:1603.07285* (2016).

[8] Xavier Glorot and Yoshua Bengio. 2010. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics (Proceedings of Machine Learning Research, Vol. 9)*, Yee Whye Teh and Mike Titterington (Eds.). PMLR, Chia Laguna Resort, Sardinia, Italy, 249–256. https://proceedings.mlr.press/v9/glorot10a.html

[9] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. Deep Learning with Limited Numerical Precision. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37* (Lille, France) *(ICML'15)*. JMLR.org, 1737–1746.

[10] Zhen Jia, Aleksandar Zlateski, Fredo Durand, and Kai Li. 2018. Optimizing N-dimensional, winograd-based convolution for manycore CPUs. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 109–123.

[11] Qingye Jiang, Young Choon Lee, and Albert Y Zomaya. 2020. The power of ARM64 in public clouds. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. IEEE, 459–468.

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

[13] Andrew Lavin. 2020. *wincnn: A simple python module for computing minimal Winograd convolution algorithms for use with convolutional neural networks.* Retrieved 2022-05-03 from https://github.com/andravin/wincnn

[14] 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.

[15] Yann LeCun and Yoshua Bengio. 1998. *Convolutional Networks for Images, Speech, and Time Series.* MIT Press, Cambridge, MA, USA, 255–258.

[16] Dongsheng Li, Dan Huang, Zhiguang Chen, and Yutong Lu. 2021. Optimizing Massively Parallel Winograd Convolution on ARM Processor. In *50th International Conference on Parallel Processing*. 1–12.

[17] Guangli Li, Zhen Jia, Xiaobing Feng, and Yida Wang. 2021. LoWino: Towards Efficient Low-Precision Winograd Convolutions on Modern CPUs. In *50th International Conference on Parallel Processing*. 1–11.

[18] Jinwook Oh, Sae Kyu Lee, Mingu Kang, Matthew Ziegler, Joel Silberman, Ankur Agrawal, Swagath Venkataramani, Bruce Fleischer, Michael Guillorn, Jungwook Choi, Wei Wang, Silvia Mueller, Shimon Ben-Yehuda, James Bonanno, Nianzheng Cao, Robert Casatuta, Chia-Yu Chen, Matt Cohen, Ophir Erez, Thomas Fox, George Gristede, Howard Haynie, Vicktoria Ivanov, Siyu Koswatta, Shih-Hsien Lo, Martin Lutz, Gary Maier, Alex Mesh, Yevgeny Nustov, Scot Rider, Marcel Schaal, Michael Scheuermann, Xiao Sun, Naigang Wang, Fanchieh Yee, Ching Zhou, Vinay Shah, Brian Curran, Vijayalakshmi Srinivasan, Pong-Fei Lu, Sunil Shukla, Kailash Gopalakrishnan, and Leland Chang. 2020. A 3.0 TFLOPS 0.62V Scalable Processor Core for High Compute Utilization AI Training and Inference. In *2020 IEEE Symposium on VLSI Circuits*. 1–2. https://doi.org/10.1109/VLSICircuits18222.2020.9162917

[19] Markus Püschel, Franz Franchetti, and Yevgen Voronenko. 2011. *Encyclopedia of Parallel Computing*. Springer, Chapter Spiral.

[20] Tran Minh Quan, David Grant Colburn Hildebrand, and Won-Ki Jeong. 2021. Fusionnet: A deep fully residual convolutional neural network for image segmentation in connectomics. *Frontiers in Computer Science* (2021), 34.

[21] Lawrence R Rabiner and Bernard Gold. 1975. Theory and application of digital signal processing. *Englewood Cliffs: Prentice-Hall* (1975).

[22] Joseph Redmon and Ali Farhadi. 2018. YOLOv3: An Incremental Improvement. https://doi.org/10.48550/ARXIV.1804.02767

[23] Karen Simonyan and Andrew Zisserman. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. https://doi.org/10.48550/ARXIV.1409.1556

[24] Sébastien Stormacq. 2022. *New – Amazon EC2 C7g Instances, Powered by AWS Graviton3 Processors.* Retrieved 2022-07-15 from https://aws.amazon.com/blogs/aws/new-amazon-ec2-c7g-instances-powered-by-aws-graviton3-processors/

[25] Tencent. 2022. *ncnn is a high-performance neural network inference computing framework optimized for mobile platforms.* Retrieved 2022-05-05 from https://github.com/Tencent/ncnn

[26] Du Tran, Lubomir Bourdev, Rob Fergus, Lorenzo Torresani, and Manohar Paluri. 2015. Learning spatiotemporal features with 3d convolutional networks. In *Proceedings of the IEEE international conference on computer vision*. 4489–4497.

[27] Jeff Underhill, Arthur Petitpierre, and Sudhir Raman. 2020. *Enable up to 40% better price-performance with AWS Graviton2 based Amazon EC2 instances.* Retrieved 2022-05-06 from https://pages.awscloud.com/rs/112-TZM-766/images/2020_0501-CMP_Slide-Deck.pdf

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

[29] S. Winograd. 1980. *Arithmetic Complexity of Computations.* Society for Industrial and Applied Mathematics. https://books.google.ca/books?id=wANiW8bGQpEC

[30] Rengan Xu, Frank Han, and Quy Ta. 2018. Deep learning at scale on nvidia v100 accelerators. In *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. IEEE, 23–32.

[31] Aleksandar Zlateski, Zhen Jia, Kai Li, and Fredo Durand. 2019. The Anatomy of Efficient FFT and Winograd Convolutions on Modern CPUs. In *Proceedings of the ACM International Conference on Supercomputing* (Phoenix, Arizona) *(ICS '19)*. Association for Computing Machinery, New York, NY, USA, 414–424. https://doi.org/10.1145/3330345.3330382