FAST AND EFFICIENT TERNARY CONVOLUTIONAL LAYERS

Felix Möller, Daniel Nezamabadi, Rudy Peterson, Luca Tagliavini

Department of Computer Science
ETH Zürich, Switzerland

ABSTRACT

Neural Networks, and especially Convolutional Neural Networks (CNNs), have become the backbone of advances in Computer Vision over the past decade. While CNNs exhibit remarkable performance on a wide variety of Computer Vision tasks, they require a large amount of memory and compute resources. To counteract this and be able to deploy CNNs on edge devices, researchers have proposed using Binary Neural Networks (BNNs) and Ternary Neural Networks (TNNs), new forms of Neural Networks which represent weights as binary and ternary values respectively, thereby saving large amounts of storage and computation.

In this report, we go one step further and present a highly optimized version of a Ternary Convolutional Layer. We optimize the algorithm top-down, starting off by changing the data order and merging subfunctions, and continue our optimizations by blocking, unrolling and eventually vectorizing the code where possible. To assess the quality of our improvements, we perform multiple benchmarks, on which we achieve a speedup of up to 29x.

1. INTRODUCTION

Motivation. The use of Convolutional Layers has enabled breakthroughs in many subfields of Computer Vision, such as Image Classification [1], Semantic Segmentation [2] and Image Generation [3]. Even the popular Transformer architecture [4] has successfully been modified to incorporate convolutions [5].

With this wide presence of Convolutional Layers in modern Neural Network architectures, it becomes clear how crucial it is to optimize them. This is especially necessary because most CNNs used today contain tens or even hundreds of layers and consist of billions of parameters, resulting in large memory and compute requirements, which cannot always be fulfilled. Nonetheless, their optimization is non-trivial. For example, Convolutional Layers need to perform well for different input and kernel shapes, padding and stride, each of which may have different effects on the performance. Another challenge of optimizing the Ternary Convolution Algorithm is indexing, as the algorithm deals with tensors of up to seven dimensions, which change size and shape multiple times throughout the algorithm. Implementing and developing a highly optimized version thus becomes error-prone, slowing down the iteration speed.

Contribution. In this report, we optimize the Ternary Convolution Algorithm proposed by Zhu et al. [6]. For our implementation, the main objective is to improve the runtime and performance of the algorithm on a single thread on one CPU. We employ classical optimization techniques such as blocking, loop unrolling and SIMD vector intrinsics using AVX2 and AVX512. The optimized version we propose is as fast or faster than Zhu et al.'s implementations on all input sizes, exhibiting the most speedup for larger input sizes and a large amount of channels, which was up to 29x faster than the original.

Related Work. TNNs [7] were first introduced by Alemdar et al., aiming to make the previously introduced BNNs [8] more sparse and energy-efficient. Zhu et al.'s algorithm, which we optimized, focuses on improving TNN performance on general purpose platforms whereas previous works, such as Chen et al.'s [9], have focused on improving the accuracy of TNNs.

2. BACKGROUND ON THE ALGORITHM/APPLICATION

In this section, we give a brief overview of a Ternary Convolutional Layer, including a cost analysis, by splitting the algorithm into four steps: Ternarize, Image to Row (im2row), General Matrix Multiply (GEMM) and Parametric Rectified Linear Unit (PReLU).

We define our cost measure as $\mathbb{C} = (W_{int}, W_{float})$, a tuple containing the number of integer operations and the number of floating point operations. To compute the amount of memory movement in bytes, we consider only compulsory misses and count writes twice.

We use the following parameters in our analysis: C (number of channels), B (batch size), IH (input height), IW (input width), KN (kernel number), KH (kernel height), KW (kernel width), P (padding size), and S (stride size).

From these, we derive the following additional parameters:

$$\begin{aligned}
PH &= IH + 2 \cdot P && \text{packed height} \\
PW &= IW + 2 \cdot P && \text{packed width} \\
OH &= \frac{PH - KH}{S} + 1 && \text{output height} \\
OW &= \frac{PW - KW}{S} + 1 && \text{output width} \\
M &= B \cdot OH \cdot OW && \text{GEMM rows} \\
K &= KH \cdot KW \cdot \lceil C/64 \rceil && \text{GEMM channels}
\end{aligned}$$

Ternarize. As a first step, each float in the input is quantized into two-bit ternary values based on some threshold, which is stored into two 64-bit integers, one for each bit of a ternary value. One such pair of integers contains the quantization of 64 floats across the channel dimension.

By considering each line in the code, we determine that

$$\begin{aligned}
W_{int} &= (.5 + .25 \cdot 2) \cdot B \cdot IH \cdot IW \cdot C \\
W_{float} &= (1 + .5) \cdot B \cdot IH \cdot IW \cdot C
\end{aligned}$$

We note that we ignore constant number of operations, as their number is small and shared across all versions. Additionally, since the number of operations is data-dependent, we weigh the work done in different branches accordingly.

im2row. im2row takes the output of ternarize, which is now ternarized and packed, and copies it into a shape such that the convolution can be done using GEMM. We visualize this in fig. 1.

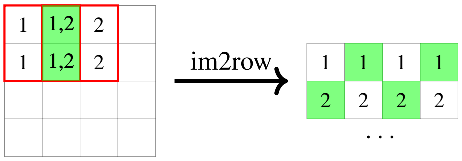


Fig. 1: Visualization of im2row. The red rectangles correspond to the 2×2 kernel sliding across the input with stride 1. Data elements that are copied multiple times are highlighted in green.

GEMM. Since the values we operate on are not floats, but two 64-bit integers representing ternary values instead, we need to adapt the way we do matrix-matrix multiplication. In particular, we will need to use bit-wise operations like AND, XOR, and Population Count (`popcnt`), where the latter counts the number of ones in an integer, which we count as three integer operations. Thus, we count $W_{float} = 0$, and $W_{int} = (5 + 2 \cdot 3) \cdot M \cdot KN \cdot K$.

For more details on this modification of GEMM, we refer the reader to the original paper [6].

PReLU. Given some $\alpha \in \mathbb{R}^+$, PReLU is defined as

$$\text{PReLU}(x) = \begin{cases} x & \text{if } x \geq 0, \\ \alpha x & \text{if } x < 0, \end{cases}$$

and is applied to every element of the output of GEMM.

Hence, $W_{int} = 0$, and since PReLU is data-dependent, we count $W_{float} = (1 + .5) \cdot B \cdot OH \cdot OW \cdot KN$.

3. OPTIMIZATIONS PERFORMED

In this section, we analyze the baseline implementation, introduce a Tensor class for code correctness, and evaluate optimizations including inlining, data order changes, function merging, vectorization, and automatic code generation with autotuning for blocking and unrolling.

3.1. Baseline Implementation

The original code [10] utilizes C++'s `std::vector` to implement each step in section 2 separately. Additionally, it assumes that the data is provided in the NCHW instead of NHWC format, i.e., the channel dimension comes before the height and width dimensions.

We immediately make the following observations:

Data Order. Recall from section 2 that packing is done across the channel dimension. If the data is provided in the NCHW format, it is unlikely that we will be able to exploit spatial locality for any non-trivial values of H and W. Furthermore, as H and W may be powers of two, the working set may be two-power strided, which is like having a smaller cache, making it even harder to benefit from locality.

Locality. We unnecessarily iterate over some data multiple times, since steps like PReLU and GEMM, or ternarize and im2row can be merged. This makes the implementation prone to be memory-bound, which we confirmed for im2row via profiling with Intel VTune.

These observations motivate the first optimizations we undertake, namely changing the data order from NCHW to NHWC, and merging as many steps as possible. This will involve many non-trivial indexing operations which are easy to get wrong, motivating the implementation of a custom Tensor class that provides safe getters/setters.

3.2. Tensor Class

We define Tensors with a fixed number of dimensions in the following way:

```
class Tensor3D {
  T *data;
  const size_t dim1;
  const size_t dim2;
  const size_t dim3; };
```

This data structure has multiple advantages: Firstly, it bundles the data pointer and the associated sizes together, which significantly improves the readability of function signatures. Additionally, it allows for the implementation of safe getters/setters which do bound checks before accessing the data, decreasing the chance of undefined behavior and other hard-to-find bugs.

Optimizing Getters/Setter. However, safe getters/setters do not come for free, as they incur the overhead of a function call and bound checks. One option to combat this is to add the `inline` keyword to the getters/setters and disable bound checks via macros. While this seems to work most of the time, we decided that we did not want to leave this optimization to the compiler. Instead, we added macros that abstract away the index computation, but do not incur any overhead.

We tested two versions of these macros: one with redundant computation, and another one where we manually apply Common Subexpression Elimination (CSE), something that compilers are usually good at. To evaluate these versions, we use the same machine as in section 4 on two benchmarks, where either the number of channels, or the height and width is increased. We used these kinds of experiments to guide our optimizations. For a more thorough evaluation, we refer the reader to section 4.

By replacing our getters/setters with "unsafe" macros, we achieve a speedup of up to 2.5x. Not manually applying CSE seems to be slightly better in some cases, but the difference could simply be in the margin of measurement error. These results make sense as getters/setters are used in the most inner loops, making the overhead of function calls and bound checking clear. Additionally, the runtime similarity between the different macro versions demonstrates that compilers tend to do CSE well.

3.3. Inlining ternarize, im2row, GEMM and PReLU

The steps described in section 2 are implemented as separate functions in different files, meaning that there is most likely no optimization happening across functions. To enable more optimizations, we moved the function definitions to headers and added the `inline` keyword. Surprisingly, this version turned out to be around 60% slower. We do not think that this is due to increased pressure on the instruction cache, as each function is called only once. It may be possible that inlining the functions lead to code that cannot be analyzed as well by the compiler. This could be because the analysis scope is larger, or because inline functions are less common in these situations, leading compiler developers to spend more time optimizing non-inline functions instead.

3.4. Changing Data Order

As motivated in section 2, we change the data order of the input from NCHW to NHWC. Note that this change only affects ternarize, as it is the only function that uses the input. Overall, the NHWC implementation with tensor macros is 3-4 times faster than the NCHW implementation with tensor macros, particularly for increasing input sizes. This makes sense, since, as mentioned before, changing the order improves spatial locality and removes the power-of-two stride in ternarize, resulting in better cache utilization.

3.5. Merging im2row

As im2row only copies data, it is by definition memory-bound and hence a perfect candidate to be merged with either ternarize or GEMM. In this section, we explore both of these possibilities.

Indirect Convolution. To merge GEMM with im2row, we implemented the Indirect Convolution [11] algorithm. Instead of reshaping the output of ternarize by copying all its contents, we can build an *indirection buffer* which contains pointers to the data that would have been copied.

Later, we can use this buffer in place of the activation tensor in GEMM after updating the code accordingly. This mainly involves having many more loops in GEMM to properly index the correct values in the output of ternarize, by using the *indirection buffer*. We implemented two variants: one with less indirection, which thus leads to a simpler implementation of GEMM, and one with maximum indirection, which uses as little memory as possible.

Both versions performed similarly. Performance was slightly better than baseline on tiny input sizes, while it significantly degraded as the input size got larger. We attribute this to the loss of spatial locality on the activation tensor. We are trading the relatively small amount of time required to reshape the activation tensor with having locality on that same tensor. Our results clearly show that for big sizes, having locality is much more important.

We note that, Indirect Convolution could also benefit from hard-coding the *indirection buffer* with precomputed offsets based on the input size, thus completely removing the need for the construction of the buffer. We didn't explore this path as it wouldn't have improved the locality in any way.

Ternarize + im2row. Algorithm 1 in the original paper merges ternarize and im2row by continuously checking whether enough of the input has been ternarized such that a step of im2row can be done. However, we do not consider this approach to be optimal. Firstly, this approach maintains two data structures: the ternarized version of the input, and the final output of im2row, leading to unnecessary copy operations and cache usage. Secondly, Algorithm 1 linearly traverses the input, meaning that by the time it is ready to do a step of im2row, some of the ternarized data has most likely been evicted from the cache.

Instead, we implement the merging operation by traversing the input in the same way im2row does. Compared to Algorithm 1, this approach should have better locality. Another benefit of this approach is that we do not maintain unnecessary data structures, nor do we unnecessarily ternarize the data if the stride is larger than the kernel size.

Overall, merging ternarize and im2row did not seem to improve runtime. Considering that, according to profiling, ternarize is not memory-bound, together with the fact that im2row, which is memory-bound, only comprises a minor fraction of the overall runtime, this is not too surprising.

We also compared different ways of implementing the copy. In particular, we investigated whether there is a difference in using a loop versus `memcpy`. As it turns out, there does not seem to be a significant difference between these options. This could be due to us not copying enough data to be able to measure any benefits from using `memcpy`.

Unrolling the loop does not seem to improve performance. This is not too surprising, since inter-loop dependencies can only be found while packing along the channel dimension. It is also possible that `gcc` automatically unrolls loops, something that we have observed more generally when checking the assembly code.

3.6. Optimizing GEMM + PReLU

We also merge GEMM and PReLU, as this change was straightforward: We only needed to move the float comparison and multiplication from PReLU into the end of the middle loop of GEMM. This only gave about 1.24 times speedup over the original on the GEMM and PReLU part of the algorithm. This is likely because the time PReLU takes is negligible when compared to that of GEMM. One advantage of this optimization is reduced memory consumption. By computing PReLU in place, we can avoid re-allocating the output tensor, thus saving $8 \cdot B \cdot OH \cdot OW \cdot KN$ bytes.

Blocking + Unrolling. We performed blocking on the M, KN, and K loops of GEMM. Each of these required a clean-up loop after the blocked loop. Because we performed blocking on each loop whose number of iterations was not in general a multiple of the chosen block size, we needed several combinations of blocked and clean-up loops. For instance, inside each blocked $M \times KN$ loop, we need a cleanup loop for KN. And after each blocked $M \times KN$ loop, we need a clean-up loop for M that contains a blocked loop for KN followed by a clean-up loop for KN. Initially, we chose the block size to roughly align with the considerations described in [12], but later moved on to using autotuning to find the best block size, as described in section 3.8, where we also summarize the results.

3.7. Vectorization

Ternarize + im2row. Merging ternarize and im2row led to speedups of up to 16x and 14x for AVX512 and AVX2 respectively, compared to our best non-vectorized implementation. The reason for this is twofold: Firstly, vector instructions are able to solve the major bottleneck in ternarize, which according to Intel VTune is branch misprediction. Secondly, AVX512 has some very convenient intrinsics for the kinds of operations needed in ternarize.

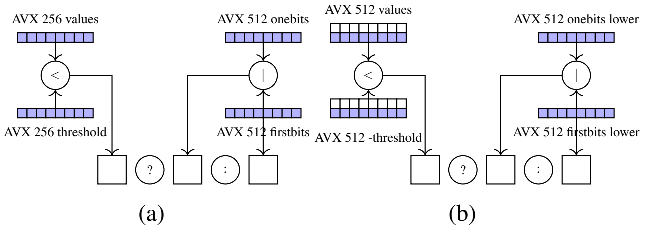


Fig. 2: Comparison of AVX512 and AVX2 ternarize + im2row inner loops. (a) Combined AVX512 and AVX2 ternarize + im2row inner loop. (b) Full AVX512 ternarize + im2row inner loop for the lower bits. The lower blue elements are used here.

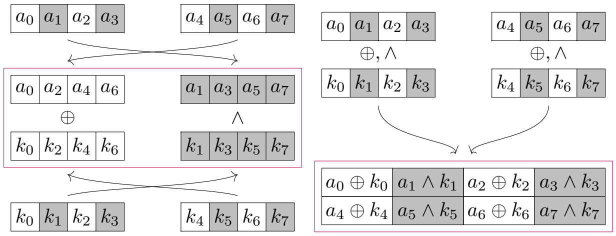
It makes sense that branch misprediction could be a major bottleneck in ternarize, given the two data-dependent branches present in its innermost loop. In AVX2 we can deal with this by computing the values of both branches, followed by a blending operation, which merges the results based on a mask. In AVX512, it is not even necessary to compute the values of both branches, as intrinsics are available which selectively compute values.

Furthermore, ternarize is special in that it deals with both single precision floats, which are 32-bits wide, and 64-bit integers. Thus, we always have a mismatch between how many of each we can fit in a vector. When implementing the AVX512 version, we started by using an AVX2 vector to load the floats (see fig. 2a). Then, to implement a pure AVX512 version, we unrolled the computation by 2, thus being able to fully fill a vector with single precision floats. This final version is shown in fig. 2b.

GEMM + PReLU. We only investigated vectorizing the GEMM part, as the code for PReLU is executed only once every K iterations of the innermost loop of the merged algorithm, meaning that its impact on performance is most likely negligible. On a high level, GEMM can be split into two steps: (a) computing some `p1` and `p2` using a combination of $\oplus$ and $\wedge$ and some data reordering, (b) doing the `popcnt` of $p2$ and $p1 \wedge p2$. We explored optimizations for both (a) and (b), which are outlined in the following paragraphs.

Less unpacking. For step (a), the naïve implementation is shown in fig. 3a. An even more naïve version using `gather` was avoided altogether due to its very low throughput.

The Intel Intrinsics Guideline [13] specifies an inverse throughput of 1 and $\frac{1}{2}$ cycles/op for unpacking and $\oplus / \wedge$ operations respectively on AVX512. For AVX2 we have a similar difference between these operations, with inverse throughputs of $\frac{1}{2}$ (1 for Skylake) and $\frac{1}{3}$ cycles/op for unpacking and other operations respectively. Therefore, we reduced the number of unpacks from 4 to 2, at the expense of some unnecessary computations, as depicted in fig. 3.



(a) Straightforward version with 4 unpacks and 2 computations.

(b) Variant with 2 unpacks and 4 computations.

Fig. 3: Visualization of two variants for the vectorized K loop in GEMM. The result is highlighted in purple.

When comparing between 3a and 3b, the latter shows good speedup on AVX512, up to 3 times faster. It is also consistently better on AVX2, reaching a peak speedup of 5% over variant 3a. Therefore, this variant was used throughout all vectorized implementations.

Popcnt placement. AVX2 doesn't provide native support for a `popcnt` operation. This forced us to resort to `libpopcnt` [14], which can perform this operation on an arbitrarily long byte-array. Thus, we can perform `popcnt` either on a single AVX vector every iteration or just once on a bigger accumulation vector. By doing a single call outside the K loop, we can get up to 90% speedup with AVX2, compared to using the non-vectorized `popcnt` inside the loop. This is expected, as `libpopcnt` implements a faster algorithm [15] using AVX2 vector instructions. We tried the same approach on AVX512, where a native `popcnt` instruction is available, but this just resulted in slowdowns. We believe this is due to `libpopcnt` also using the same AVX512 instruction, thus only adding the extra overhead of a function call.

Finally, we also experimented with unrolling, unrolled cleanup loops, and their effect on the previously detailed optimizations. More about these experiments can be found in the next section.

3.8. Code Generation + Autotuning

Besides experimenting with various kinds of unrolling, as hinted in the previous section, we also wanted to investigate blocking sizes for loops over M and N. We used automated approaches to tweak the various parameters involved. While autotuning alone suffices to explore the blocking sizes for the M and N loops, the other experiments required code generation.

We developed a modular code generator capable of outputting the innermost kernel for GEMM in SSA style. After implementing the straightforward version, we also encoded several AVX2 and AVX512 variants. The generator is modular in that it can reuse generation techniques from other variants to generate code for a new variant[1].

Our code generation approach involves an exhaustive search over all parameter values. While this is less efficient compared to a guided search, it was simpler to implement and successful nonetheless. The main findings from this search are briefly summarized in the following paragraphs.

Blocking of M and N. The blocking factor for M and N does not significantly influence performance. We believe this is due to the TNN GEMM algorithm not being memory bound, unlike typical GEMM, as the innermost loop performs many more operations.

Unrolling of K. Unrolling the K loop did help slightly, but too much unrolling would hinder the performance. We ended up using unrolling factors of 4, 2 and 1 for the straightforward, AVX2 and AVX512 versions respectively. Having the AVX2 unrolling factor be twice that of AVX512 makes sense, as the vector sizes are halved. We believe the reason why these values work well is that they are the largest factors that still allow the program to enter the unrolled loop in most circumstances. Using bigger values would mean that we never enter the unrolled loop, as the value of K would be too small.

Cleanup loop unrolling. Unrolling the cleanup loop allowed us to improve performance even with small values of K, when vectorized loops would not be used. We used an unrolling factor of 2 for both AVX2 and AVX512, in order to (almost certainly) enter the loop. Using a bigger unrolling factor, such as 4 or 8 would be pointless, as the vectorized loop for AVX2 or AVX512 respectively would have been used instead.

4. EXPERIMENTAL RESULTS

In this section, we present an account of the results obtained throughout all major optimization previously presented.

Experimental setup. All our benchmarks were run on an `AMD Ryzen 7 PRO 7840U`, as it was the only CPU we had with AVX512. This CPU is clocked at 3.3GHz and has 512KB, 8MB, 16MB for the L1, L2, L3 caches, respectively. Measurements were taken with Turbo Boost disabled, and we used `taskset` to pin the execution to a single CPU core. For compilers, we tried `gcc`, `clang` and `icc`. We didn't measure any significant performance difference between the three, so throughout this section we'll be using `gcc v12.2.0`. We experimented with various compiler flags, such as `-ffast-math`, `-funroll-loops`, `-fno-strict-aliasing` but none yielded surprising results, besides some making the baseline slightly faster. This makes sense as the computation is mainly integer-based,

[1] For example, to generate the (unrolled) cleanup loop for the AVX variants, we leverage the code generator's capability to output straightforward code.

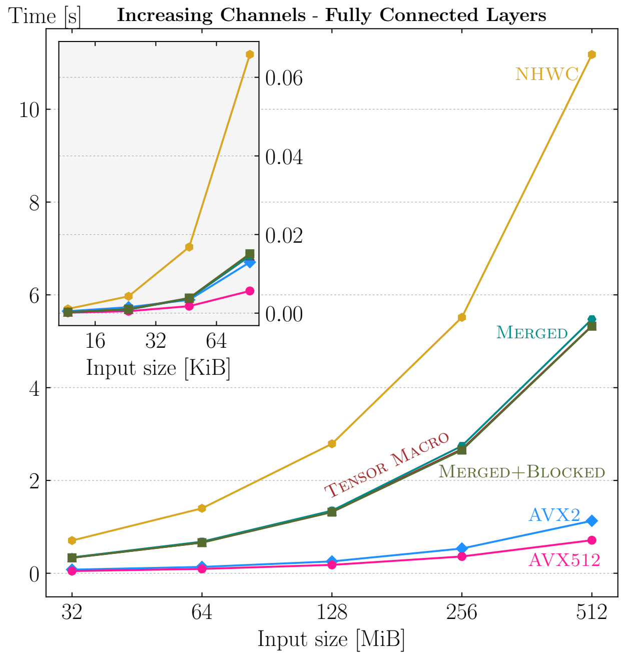


Fig. 4: Runtime of all major optimizations while increasing the number of channels. The main plot shows convolution with a typical 3 × 3 kernel size and padding of 1, while the smaller plot depicts Fully Connected (FC) layers. Both have a stride of 1.

which the compiler can already optimize well with the standard `-O3` optimization level. At the end, we settled on `-O3`, `-march=native`, `-fno-tree-vectorize` and flags to enable AVX512 support as our optimization flags. We disable compiler vectorization to avoid unexpected vectorization, which would have made our analysis harder. The benchmarking parameters have been selected to push the algorithm to its limits, as well as capture a more realistic use of a TNN convolution layer.

Our plots show the runtime or performance of the entire convolution algorithm. We avoid showing a roofline plot, as it is not particularly insightful. We do not reach the machine's peak performance (as defined in the lecture), but we are not memory-bound either, which we confirmed using Intel VTune. We believe there are multiple reasons behind this: Inherent data dependencies (especially in GEMM), instruction mix (e.g., `popcnt` has four times lower throughput than AND or XOR), pipeline overheads like branch mispredictions (which we addressed with AVX), and inefficiencies in the AVX variants (see section 3.7).

Increasing Channels and FC layers. Figure 4 shows the runtime of the various optimized versions when increasing the number of channels. The plot also includes a subplot where we increase channels while simulating a FC layer, which was also done in the original paper [6]. In contrast to other visualizations, these plots do not include Original, the original unoptimized baseline, which would perform so poorly that it would harm the readability of the plot. Instead, we postpone this comparison to later.

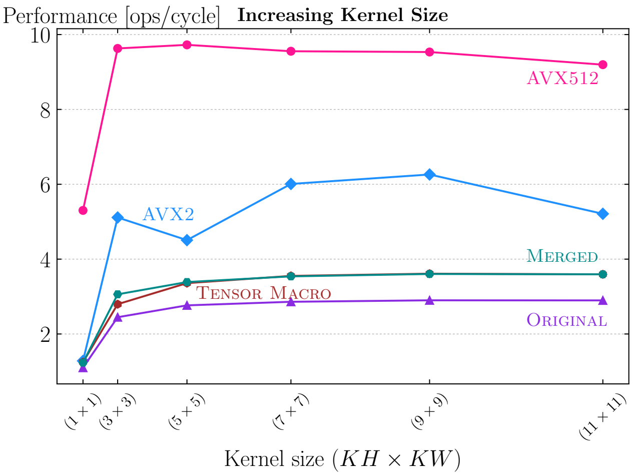


Fig. 5: Performance of all optimizations, including Original, while sweeping through kernel sizes. Input size and KN are fixed at $(2, 56, 56, 512)$ and 256 respectively. Padding is increased as necessary.

As previously discussed, the runtime of Merged, which merges ternarize with im2row, and GEMM with PReLU, and Merged+Blocked aligns with that of Tensor Macro, which uses the NHWC data format and macros instead of getters and setters. While there are no performance improvements, these optimizations do save memory, improve locality and shrink the scope for further optimizations. In the larger plot, AVX2 achieves up to 4.7x speedup compared to Tensor Macro, but it doesn't perform that well in the smaller plot, obtaining a minor 16.5% runtime improvement. This is because, in **FC** layers, ternarize finishes quickly and most of the time is spent inside GEMM, which is harder to optimize for AVX2 since there's no native `popcnt`. In the **Increasing Channels** scenario, ternarize is the bottleneck and the AVX2 optimizations help immensely.

Overall, the maximum achieved speedups compared to NHWC are 15.7x and 11.7x for the **Increasing Channels** and **FC** layers scenarios, respectively. When comparing against the Original baseline, we obtain speedups of 29.3x and 3.2x for **Increasing Channels** and **FC**, respectively. Again, the smaller number in the **FC** case is expected, as ternarize does not impact the runtime as much.

Increasing Kernel Size. Figure 5 aims at investigating the performance impact of increasing kernel sizes. While sizes of 3×3 and 5×5 are most-commonly used, we wanted to push the algorithm to its limit. Thus, we experimented with sizes up to 11×11. Similarly to **FC** layers, the runtime for this computation is going to be dominated by GEMM, as ternarize uses as little as 3% of the runtime for bigger

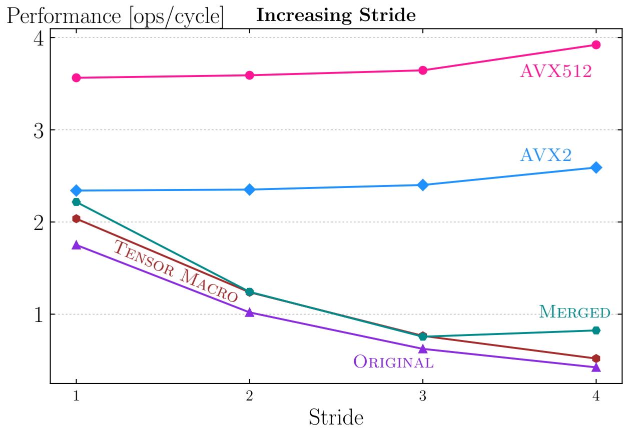


Fig. 6: Performance of all optimizations while increasing the stride. Input and kernel size are fixed at $(2, 224, 224, 80)$ and $(80, 3, 3)$ respectively. Padding is increased as necessary.

kernels.

Starting from the bottom, we can see that just changing the data order of the input improves the performance by about 24%, because it improves the data locality in ternarize. This effect is going to be very limited, however, as ternarize takes up a small fraction of the total runtime. In fact, in this scenario, the bottleneck is GEMM. While MERGED doesn't improve performance, it still has the previously outlined benefits, while saving up to 13% of memory.

AVX2 achieves a speedup of up to 2.2x compared to ORIGINAL, mainly due to the improvements in GEMM. This speed up is quite good considering that AVX2 doesn't have native support for `popcnt`. Because of this and the same issues affecting AVX512, outlined in the next paragraph, we cannot achieve more performance.

AVX512 is able to obtain a speedup of up to 3.45x compared to MERGED, as opposed to the theoretical speedup of 8x. More performance could not be achieved because, as previously discussed in section 3.7, all vectorized implementations of this algorithm have to either use many unpack instructions or perform unnecessary computations. Furthermore, there is no way to avoid at least two unpack operations per iteration, which add latency to the execution. This ultimately leads to cycles wasted either moving bits across AVX registers or computing useless data.

Overall, in this scenario, the maximum speedup achieved is 3.9x compared to the Original.

Increasing Stride. Figure 6 shows how increasing the stride impacts performance. All implementations, except the vectorized ones, struggle more and more as the stride increases, mainly because of slowdowns in ternarize, due to two factors: (a) worsened locality (b) unnecessary computations. Near the end of the plot, we notice an increase in performance for the variants that include the merged ternarize + im2row. This makes sense since, as stated in section 3.5, the merged version of ternarize avoids ternarizing values which are not going to be used due to the stride. This improvement is lacking in ORIGINAL and TENSOR MACRO, hence the lower performance for stride ≥ 4. Further strengthening this claim is the same behavior registered in the performance of the AVX versions, which are based on the MERGED variant. Both vectorized versions exhibit good performance, with a speedup of up to 9.3x and 6x for AVX512 and AVX2, respectively. Compared to the non-vectorized, but otherwise identical, MERGED, AVX2 manages to get a 3.2x speedup, which is close to the theoretical maximum. As usual for ternarize, AVX512 cannot reach its theoretical maximum, but exhibits performance closer to the AVX2 version.

5. CONCLUSIONS

In this project, we optimized the base implementation of a Ternary Convolutional Layer as presented in [6]. We divided the Ternary Convolution Algorithm into four subparts (Ternarize, im2row, GEMM and PReLU) and started optimizing top-down.

As a first step, we changed the channel order, which lead to significant speedup. This was followed by merging im2row into ternarize instead of GEMM (indirect convolution). While the latter performed worse due to the introduction of indirection, the former did not lead to significant speedups. A similar phenomenon was observed when merging GEMM and PReLU.

For ternarize+im2row, we experimented using `memcpy` instead of a loop, without much success. Blocking and unrolling GEMM + PReLU was more successful, however its speedup was negligible.

Finally, we vectorized both functions using AVX512 and AVX2. We also used code generation and auto-tuning to test multiple different versions of vectorizing GEMM + PReLU. As expected, this yielded the majority of our overall speedup, most notably the masked vector operations we used in ternarize+im2row.

Our best version achieves a speedup of up to 29x compared to the original implementation, demonstrating the success of the optimizations that were performed.

Due to the nature of this course project, our optimizations were limited to single-threaded CPU computing. Given that many neural networks run on graphics processing units (GPUs), we believe that future work should aim at making our optimized code GPU-compatible, before applying further GPU-specific optimizations.

6. CONTRIBUTIONS OF TEAM MEMBERS (MANDATORY)

Felix. Worked on GEMM+PReLU. Implemented vectorization for AVX512, implemented blocking and loop-unrolling. Merged all optimization steps together.
Daniel. Implemented and evaluated various tensor types. Implemented different versions of merging ternarize+im2row. Used autotuning to verify blocking parameters based on [12] for GEMM+PReLU.
Rudy. Unrolled and vectorized ternarize+im2row. Merged GEMM+PReLU. Worked on blocking for GEMM+PreLU. Performance plots.
Luca. Implemented and evaluated the Indirect Convolution algorithm. Vectorized GEMM for AVX2. Implemented code generation for vectorized GEMM, including all AVX2 / AVX512 variants.

References

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

[2] Olaf Ronneberger, Philipp Fischer, and Thomas Brox, "U-net: Convolutional networks for biomedical image segmentation," in *Medical image computing and computer-assisted intervention–MICCAI 2015: 18th international conference, Munich, Germany, October 5-9, 2015, proceedings, part III 18*. Springer, 2015, pp. 234–241.

[3] Alec Radford, Luke Metz, and Soumith Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," *arXiv preprint arXiv:1511.06434*, 2015.

[4] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[5] Haiping Wu, Bin Xiao, Noel Codella, Mengchen Liu, Xiyang Dai, Lu Yuan, and Lei Zhang, "Cvt: Introducing convolutions to vision transformers," in *Proceedings of the IEEE/CVF international conference on computer vision*, 2021, pp. 22–31.

[6] Shien Zhu, Luan H. K. Duong, and Weichen Liu, "Tab : unified and optimized ternary, binary and mixed-precision neural network inference on the edge," 2022.

[7] Hande Alemdar, Vincent Leroy, Adrien Prost-Boucle, and Frédéric Pétrot, "Ternary neural networks for resource-efficient ai applications," in *2017 international joint conference on neural networks (IJCNN)*. IEEE, 2017, pp. 2547–2554.

[8] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *European conference on computer vision*. Springer, 2016, pp. 525–542.

[9] Peng Chen, Bohan Zhuang, and Chunhua Shen, "Fatnn: Fast and accurate ternary neural networks," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2021, pp. 5219–5228.

[10] Shaun Zhu, "Github: Asl_tab," 2024, Last Accessed: 2024-06-17.

[11] Marat Dukhan, "The indirect convolution algorithm," *CoRR*, vol. abs/1907.02129, 2019.

[12] K. Yotov, Xiaoming Li, Gang Ren, M.J.S. Garzaran, D. Padua, K. Pingali, and P. Stodghill, "Is search really necessary to generate high-performance blas?," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 358–386, 2005.

[13] "Intel® intrinsics guide," `http://web.archive.org/web/20240616141553/https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html`, Accessed: 2024-06-16.

[14] Kim Walisch, "Github: libpopcnt," 2016, Last Accessed: 2024-06-18.

[15] Wojciech Mula, Nathan Kurz, and Daniel Lemire, "Faster population counts using AVX2 instructions," *CoRR*, vol. abs/1611.07612, 2016.