YFlows: Systematic Dataflow Exploration and Code Generation for Efficient Neural Network Inference using SIMD Architectures on CPUs

Cyrus Zhou
University of Chicago
Chicago, Illinois, USA
zhouzk@uchicago.edu

Zack Hassman
University of Chicago
Chicago, Illinois, USA
zhassman@uchicago.edu

Dhirpal Shah
University of Chicago
Chicago, Illinois, USA
dhirpalshah@uchicago.edu

Vaughn Richard
University of Chicago
Chicago, Illinois, USA
vaughnrichard@uchicago.edu

Yanjing Li
University of Chicago
Chicago, Illinois, USA
yanjingl@uchicago.edu

Abstract

We address the challenges associated with deploying neural networks on CPUs, with a particular focus on minimizing inference time while maintaining accuracy. Our novel approach is to use the dataflow (i.e., computation order) of a neural network to explore data reuse opportunities using heuristic-guided analysis and a code generation framework, which enables exploration of various Single Instruction, Multiple Data (SIMD) implementations to achieve optimized neural network execution. Our results demonstrate that the dataflow that keeps outputs in SIMD registers while also maximizing both input and weight reuse consistently yields the best performance for a wide variety of inference workloads, achieving up to 3x speedup for 8-bit neural networks, and up to 4.8x speedup for binary neural networks, respectively, over the optimized implementations of neural networks today.

CCS Concepts: • **Computer systems organization** → **Single instruction, multiple data**; Embedded software; • **Software and its engineering** → **Source code generation**; • **General and reference** → *Performance*.

Keywords: code generation, compiler support, SIMD vectorization, CPU optimization, dataflow, neural network

ACM Reference Format:
Cyrus Zhou, Zack Hassman, Dhirpal Shah, Vaughn Richard, and Yanjing Li. 2024. YFlows: Systematic Dataflow Exploration and Code Generation for Efficient Neural Network Inference using SIMD Architectures on CPUs. In *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction (CC '24), March 2–3, 2024, Edinburgh, United Kingdom.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3640537.3641566

CC '24, March 2–3, 2024, Edinburgh, United Kingdom

ACM ISBN 979-8-4007-0507-6/24/03
https://doi.org/10.1145/3640537.3641566

1 Introduction

In recent years, neural networks have expanded their reach beyond high-performance computing environments, permeating low-end servers and edge devices such as smartphones, IoT devices, and smart sensors [29, 55, 90, 91]. However, the deployment of neural networks on these devices presents various challenges, with inference time being a critical factor [31, 38, 40, 55, 95]. The Single Instruction, Multiple Data (SIMD) capabilities of contemporary CPUs present an opportunity to accelerate neural networks. SIMD allows a single instruction to be executed on multiple data elements concurrently, thereby substantially improving computational throughput and overall performance, and yielding benefits in terms of both energy conservation and efficient utilization of computational resources at the same time [35, 59, 82].

Dataflow refers to the execution order of computational operations of a neural network. It is an important consideration when utilizing SIMD for inference. It determines the reuse opportunities of different variables (e.g., inputs, weights, and outputs), and can therefore guide how to best allocate valuable SIMD register resources to maximize reuse. While dataflows for deep learning accelerators have been extensively explored [12, 15, 32], the majority of previous studies and libraries for CPUs do not consider dataflows [2, 13, 39, 80]. Instead, weight stationary, i.e., keep using the same weight value until all computations requiring this value are done before moving on to computations requiring a different weight value, is widely adopted [20, 42, 68]. The weight stationary dataflow turns out to be suboptimal – we found that by carefully devising dataflows to maximize data

reuse, and co-optimizing with other code optimization techniques (i.e., blocking, operator fusion), inference speed and energy efficiency can be improved significantly.

Unfortunately, compiler support for efficient SIMD code generation is inadequate [6, 28, 56], as evidenced in our experiments on x86 and ARM architectures. Programs written to explicitly utilize SIMD receive no further compiler optimization, such as effectively utilizing unused vector registers [69]. Furthermore, auto-vectorization features in compilers [21, 64] tend to overlook opportunities for vectorizing scalar implementations [6, 28, 50, 77], potentially because of the vast search space as indicated in [103].

The intricacies of SIMD optimization, such as ensuring independence among vector register values, are highlighted in [50, 69]. These challenges are compounded by the reliance on fragile heuristics in current auto-vectorization techniques, as critiqued in [30, 56, 77]. This issue extends to highly optimized frameworks like TVM [13], which depend on compiler backends such as LLVM [57]. While maximizing data reuse has been mentioned as a method to optimize neural network inference programs on CPUs in prior work [68, 79], to the best of our knowledge, there are no systematic approaches. With these challenges, the burden of SIMD optimization predominantly falls on programmers. Hence, there is a pressing need for a systematic approach to maximize the efficiency of SIMD programs for performing neural network inference tasks.

To fill in these blanks, we present the first work that employs the notion of dataflow to systematically explore the full SIMD computation capacities for efficient neural network inference. The major contributions include:

1. We extended the existing dataflows, which typically specify only one type of variable to be reused, to allow all types of variables to be reused. Extended dataflows enable SIMD register resources to be fully utilized, and substantially reduce costs associated with data and instruction movements.
2. We formalized a set of heuristics, based on data movement costs, to optimize three basic, general neural network dataflows (defined in Sec. 2) by maximizing data reuse within each dataflow.
3. We implemented a code generator that automatically uses SIMD instructions to implement various extended and basic dataflows, for any given neural network configuration. This code generator allows quantitative comparison of different dataflows to identify the most efficient implementation.
4. We quantitatively compared our best implementations against state-of-the-art implementations using representative workloads. The results demonstrate significant improvements: our implementations achieve up to a 3.5x speedup for 8-bit neural networks (against TVM [13]), and up to a 4.8x speedup for binary neural networks (against [68]), respectively.

This paper is organized as follows. We discuss the basic dataflows in Sec. 2, and our methods to extend the basic dataflows in Sec. 3 and 4. Experiment setup and results are presented in Sec. 5 and 6, followed by related work and conclusions.

2 Basic Dataflows of Neural Networks

Three major, basic dataflows have been identified in the literature for CPUs[1] [52, 86, 106], as shown in Algorithms 1, 2, and 3 following the semantics of ARM SIMD intrinsics [63], using convolution layers as an example.

2.1 Input Stationary (IS)

IS operates by iterating through the entries in the input tensor. It applies all relevant filters to each input and accumulates the results to the respective entries in the output tensor.

Algorithm 1 IS Dataflow for Convolution Layers.

```
Require: inputs[H], weights[R], outputs[E]
  for h in H do
    input ← vload(&inputs[h]);
    for r in R do
      weight = vload(&weights[r]);
      calculate corresponding input index e from i, r, if the
corresponding output index e falls out of the range then continue;
      outputs[e] += vredsum(vmul(input, weight));
    end for
  end for
```

2.2 Weight Stationary (WS)

WS iterates through weight tensors. For each output entry whose computation depends on the current weight tensor, WS collects each relevant entry from the input tensor for computations and accumulates the result to the corresponding output entries.

Algorithm 2 WS Dataflow for Convolution Layers.

```
Require: inputs[H], weights[R], outputs[E]
  for r in R do
    weight ← vload(&weights[r]);
    for e in E do
      calculate input index i from e, r;
      input = vload(&inputs[i]);
      outputs[e] += vredsum(vmul(input, weight));
    end for
  end for
```

[1] We exclude dataflows that are tailored to specific deep learning accelerator architectures (e.g., *Row-stationary* [15], *No-local-reuse* [5], etc.) as they cannot be applied to CPUs. For example, row-stationary keeps software variables stationary in the rows of processing engines of a 2D systolic array; however, there is no notion of "rows of cores" in CPUs.

2.3 Output Stationary (OS)

OS iterates through the entries in the output tensor. It performs all necessary multiply-accumulate computations to obtain the final result for one output entry before moving on to the next.

Algorithm 3 OS Dataflow for Convolution Layers.

Require: inputs[H], weights[R], outputs[E]
  for e in E **do**
    output = $vmov(\vec{0})$
    for r in R **do**
      calculate input index i from e, r, if i falls out of the range then continue;
      input, weight = $vload$(&inputs[i]), $vload$(&weights[r]);
      output = $vadd(vmul$(input, weight), output);
    end for
    outputs[e] = $vredsum$(output);
  end for

2.4 Memory Layout and Computation Order

Naturally, the computation order under a dataflow follows the sequential memory addresses of the corresponding data elements. We illustrate the memory layout scheme in Fig. 1.

We opt for the NCHW[xc] memory layout for each input/output tensor. In traditional NCHW alignment, tensors are arranged first by the number of images (batch size, N), then channels (C), followed by height (H), and lastly width (W). In NCHW[xc], data are grouped into blocks of size $x \times H \times W$, and we call these blocks *channel blocks*. The channel blocks follow the NCHW layout, while data in each channel block follows the HW[xc] layout, and x is typically chosen so that $x \times element\_width$ is a multiple of the size of the physical vector registers (1-3× in our implementation).

Previous works have demonstrated the effectiveness of this scheme for floating-point, integer, and binary neural networks [39, 68, 87]. We attribute this to two main reasons. First, vectorization in the channel dimension streamlines vector computations, avoiding excessive operations such as shifting, because the number of channels multiplied by data size in a neural network layer is usually a multiple of the size of SIMD registers (or vice versa). Second, NCHW[xc] enables data reuse between successive channel blocks. In NHWC, no element engages in calculations across two spatially successive elements, be it inputs, weights, or outputs, under any dataflow. In contrast, NCHW[xc] allows for the exploration of various dataflows to maximize data reuse (refer to Sec. 3). Note that, for binary networks, NHWC can yield performance comparable to NCHW[xc] since the number of channels in most network architectures is ≤ 512 and a multiple of the vector register size in modern ISAs [39].

To optimize weight data access locality, we adopt the CKRS[xc] memory layout (matching the input/output tensor layout), where C, K, R, S denote #Input Channels, #Output Channels, #rows/filter height, #columns/filter width, respectively, and x for weight tensors is set to the exact same value as the x chosen for the input tensor. Following this layout, output tensors can be written back sequentially regardless of the size of the input/output channel blocks and the dataflow.

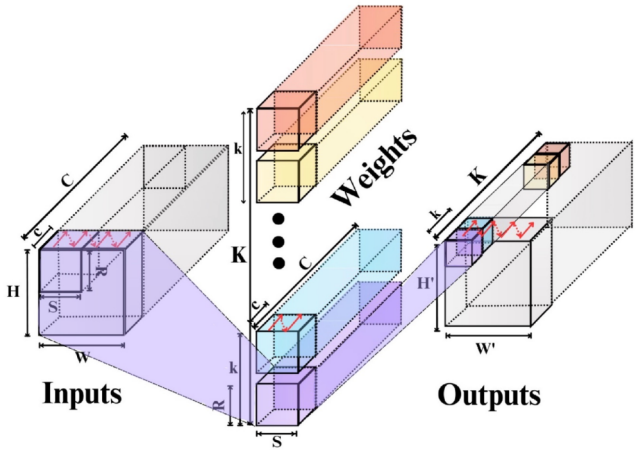


Figure 1. Memory layout of tensors. Red arrows show a subset of data elements following sequential memory addresses. Input channel blocks are traversed first along the output channel dimension. The purple shade covers a single vector variable.

In terms of the compute order across input channel blocks, for better memory locality, we proceed along the output channel dimension before moving on to the next input channel block. In other words, the loop on the input channel dimension is an outer loop of that on the output channel dimension.

2.5 Implementation and Performance of Basic SIMD Dataflows

In software, we declare three *vector variables* to implement any of the three basic dataflows, one for each of the input, weight, and output data types. The size of each vector variable is $x \times element\_width$ (as shown in Fig. 1, shaded in purple), which is a multiple of the vector register size. Also, the total size of all vector variables is less than or equal to the total size of all vector registers. We use the term vector variable in addition to vector registers because physical vector registers in some architectures can be concatenated to form longer vectors. For example, in ARM, vector registers are 128 bits in size, but vector variables can be multiples of 128 bits occupying multiple physical registers.

We compared the three basic dataflows (the experiment setup is outlined in Sec. 5), and the results can be found in Fig. 2. We see that OS consistently outperforms the other dataflows in terms of runtime. With a stride of 1, OS is by median 1.93x and 3.50x faster than IS and WS, respectively. With a stride of 2, OS is, by median, 5.39x and 2.81x faster than IS and WS, respectively. The superior performance of

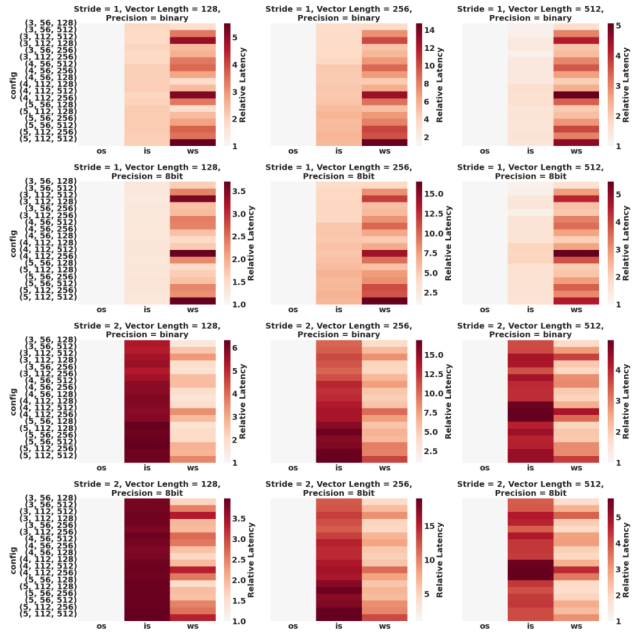


Figure 2. Relative latency of basic dataflows for various convolution layers for Vector Length = $(elem\_width \times c) \in \{128, 256, 512\}$ (mean of 100 runs), normalized to the latency of OS. Configurations on the y-axes are in the format of (weight width/height, input width/height, number of output channels).

OS can be attributed to several factors, including smaller numbers of reduction sum operations, less frequent output tensor data movement, and more regular instruction and memory access patterns.

The implementations of all basic dataflows are only capable of utilizing a limited number of vector registers (precisely $\frac{3\times\text{vector variable size}}{\text{vector register size}}$), leaving all others idle. This is because, as discussed in Sec. 1, compilers today are not able to discover vectorizable code effectively and fully utilize all vector registers automatically. This necessitates the need to extend and optimize the basic dataflows.

3 Extending the Basic Dataflows

We say that a dataflow utilizes the stationarity of some data if it keeps that data close to the compute units — in vector registers in our case — for reuse. A dataflow is σ stationary if it uses σ stationarity, where σ is a predefined type of data (inputs, weights, or outputs). We extend the notion of dataflow by defining two types of stationarities, i.e., *anchoring stationarities* and *auxiliary stationarities*.

Anchoring stationarity is the stationarity that decides the execution order of computations. For example, output stationary dataflows have the outputs as their anchoring data type, so we always complete **all** computations involving an output element before moving on to the next. One dataflow can have at most one *anchoring stationarity*. The most naive implementation of a dataflow is constituted of an anchoring stationarity only, which is equivalent to one of the basic dataflows discussed in Sec. 2. A major limitation of all basic dataflows is that not all vector registers are utilized.

In more optimized implementations, vector registers are fully utilized to minimize data movement costs associated with both anchoring and non-anchoring data types (i.e., *auxiliary data types*). The *auxiliary stationarities* determine which auxiliary data types should be allocated in vector registers. For example, an output-anchored dataflow may be accompanied by weight or input auxiliary stationarity. More than one auxiliary stationarity can accompany an anchoring stationarity.

An important question is to decide how to allocate vector registers to store (or stash) anchoring and auxiliary data types, which is dependent on two factors: (1) the total number of available vector registers, which constraints the overall SIMD capability, and (2) data reuse opportunities, which affects data movement costs, and also bounds the benefits that can be obtained by stashing the corresponding data in vector registers.

4 Optimizing Extended Dataflows

Our methodology for optimizing an extended dataflow follows two steps. First, we analyze reuse opportunities and develop heuristics to maximize data reuse benefits within each basic (i.e., anchoring stationarity only) dataflow to derive the corresponding auxiliary stationarities. Next, we empirically compare different implementations of the extended dataflows by varying vector register allocation schemes using a code generator to determine the best dataflow for performance.

While this methodology can be applied to most layers in neural networks, we focus our discussions on convolution layers, including simple convolutions [58], depthwise convolutions [38, 92], grouped convolutions [51], shuffled grouped convolutions [111], and so on. This is because these layers are common, and their latencies are generally longer compared to other layers [18, 38, 40, 95, 112]. However, our methodologies and techniques can be generalized to other types of layers or operations where data reuse can enhance performance. For instance, in tensor-contraction operations like MatMul, we can calculate multiple output data simultaneously, allowing for the reuse of the same input row with different columns (or vice versa, and can be expressed as input or weight auxiliary stationarity under output-anchored stationarities with multiple anchored elements). In pooling operations, the basic OS dataflow already achieves maximal reuse. As for softmax, dataflow choices are less critical.

The convolution operation is shown in Fig. 3. Notation-wise, we use ih, iw, fh, fw, oh, ow for input height, input width, filter/weight height, filter/weight width, output height, and output width, s for strides, x for the number of

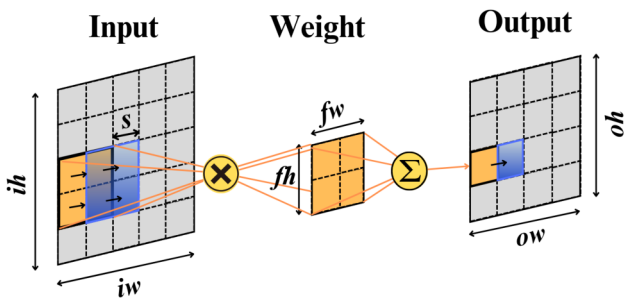


Figure 3. Convolution operations and notations, showing only 1 channel and 1 kernel.

data elements in a vector variable. We also define H, R, E for the sizes of input, filter/weight, and output tensors within a single channel block. Thus, $H = ih \cdot iw \cdot x$, $R = fh \cdot fw \cdot x$, $E = oh \cdot ow$. Below, we describe our approach for optimizing dataflows within the context of the combination of a single input channel block and a single kernel for clarity, and the same approach is applied across all such combinations.

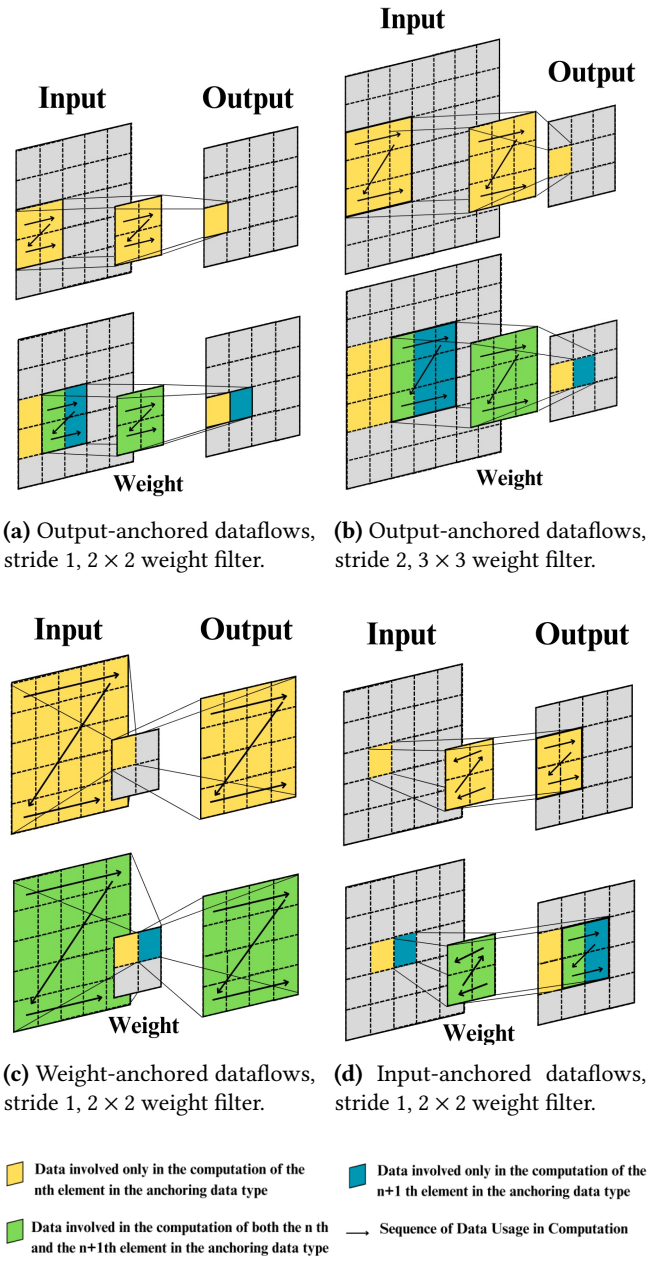
4.1 Maximizing Data Reuse under Each Basic Dataflow

4.1.1 Reuse under Output Stationary Dataflows. Under output-anchored dataflows with the computation sequence following the description in Sec. 2.3, all corresponding weights, totaling R, are reused between the computations for two successive output elements. Additionally, there are $(fw - s) \cdot fh$ reusable input elements involved in the computations for two successive outputs. We demonstrate these reuse opportunities in Fig. 4a.

The reuse scheme of inputs is similar for $s > 1$, as shown in Fig. 4b, differing only by the number of inputs reusable between the computations around two successive outputs.

4.1.2 Reuse under Input Stationary Dataflows. Given the algorithm of the basic input-anchored dataflow (Sec. 2.1), when $s = 1$, all corresponding weights, totaling R, can be reused between the computations around two successive input elements. Outputs (partial sums) under input-anchored dataflows can be reused in a way similar to how inputs are reused under output-anchored dataflows. We demonstrate this reuse scheme in Fig. 4d. Note that we would need to reverse the sequence of the weights (i.e., following the order of the outputs) to enable this reuse scheme (see Fig. 4d).

When $s > 1$, reusing both outputs and weights becomes more difficult. Not all weights are applied to every input. For $s = 2$, the number of weights/outputs associated with the computations around one input can be 1, 2, or 4, as demonstrated in Fig. 5. Consequently, the reuse opportunities become more sparse. Additionally, code structure becomes less regular.



(a) Output-anchored dataflows, stride 1, 2×2 weight filter.

(b) Output-anchored dataflows, stride 2, 3×3 weight filter.

(c) Weight-anchored dataflows, stride 1, 2×2 weight filter.

(d) Input-anchored dataflows, stride 1, 2×2 weight filter.

Figure 4. Reuse opportunities under each anchoring dataflow, showing only one channel and one kernel.

4.1.3 Reuse under Weight Stationary Dataflows. In weight-anchored dataflows (Sec. 2.2), between the computations around two successive weights in an input channel block, all H inputs and E outputs can be reused, as depicted in Fig. 4c.

When using vector registers to stash an input, the input will not be reused in the computation involving each weight when $s > 1$. On the other hand, stashed outputs are guaranteed to be reused with each weight. As stashing outputs also

Table 1. Summary of gains from auxiliary allocation for each operation involving one input channel block and one kernel.

Anc. Dataflow	Aux.	# vector variables for aux.	Stride	Reduction in # mem. reads for each additional vector variable allocated to auxiliary data	Reduction in # mem. writes for each additional vector variable allocated to auxiliary data
OS	Both	$[1, R]$	$[1, fw-1]$	E	0
WS	Input	$[1, H]$	$[1, fw-1]$	R	0
	Output	$[1, E]$	$[1, fw-1]$	R	R
IS	Weight	$[1, R]$	1	H	0
	Weight	$[1, fw]$	$[2, fw-1]$	$\frac{H}{s}$	0
	Weight	$[fw+1, 2 \cdot fw]$	$[2, fw-1]$	$\frac{H}{(fw-s)s}$	0
	Output	$[1, R]$	1	H	H
	Output	$\{1\}$	$[2, fw-1]$	$H + \frac{H}{fw}$	$H + \frac{H}{fw}$
	Output	$\{2\}$	$[2, fw-1]$	$\frac{ih}{fw-s}(H + \frac{H}{fw}) + \frac{ih}{s}(fw-s-1)$	$\frac{ih}{fw-s}(H + \frac{H}{fw}) + \frac{ih}{s}(fw-s-1)$
	Output	$[3, (3+fw-s)]$	$[2, fw-1]$	$(fh-s)(fw-s)\frac{H}{R}$	$(fh-s)(fw-s)\frac{H}{R}$

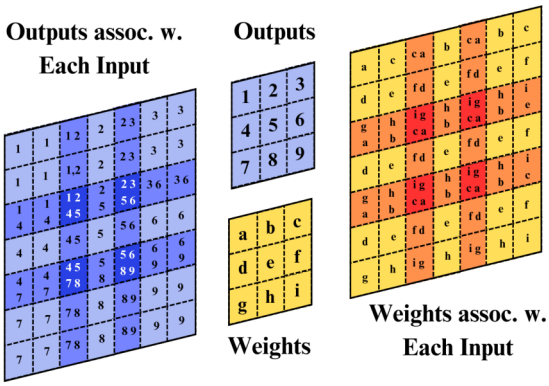


Figure 5. Under input-anchored dataflows: weights and outputs associated with each input when s=2 for each channel. Darker colors in the input indicate more data are associated with that input element.

saves write-related operations and the size of the output tensor is almost always greater than the remaining SIMD vector registers, we will later demonstrate that it is sufficient to only include output auxiliary stationarity under weight-anchored dataflows.

4.1.4 Heuristics to Quantify the Effectiveness of Data Reuse under Each Dataflow.

We use the reduction in the number of memory instructions (both read and write, data size = $x \times elem\_width$) for each input channel as the guiding metric for framing the heuristics for choosing auxiliary stationarities, summarized in Table 1. The baseline configurations correspond to the basic dataflow implementations discussed in Sec. 2, where only (3 × vector variable size ÷ vector register size) vector registers are allocated. For the extended dataflows, we utilize additional vector variables (which are mapped to vector registers) for the auxiliary data types to further reduce data movement costs.

Output-Anchored Dataflows. Independent of the value of s, the numbers of inputs and weights associated with an output element, disregarding edge cases, are always equal to R for each input channel. Thus, every time we stash an input or weight vector variable in one or more vector register(s), the number of memory reads always goes down by the size of the output tensor.

Input-Anchored Dataflows. When $s = 1$, the gains from auxiliary allocation are similar to those under output anchored dataflows. We expect a reduction of H memory reads and H memory writes for every vector variable allocated to stash outputs for each input channel block. For each vector variable allocated for stashing weights, we expect a reduction of H memory reads per input channel block. Note that $H \approx E$ in this case. When $s > 1$, the gains from auxiliary allocation depend on multiple factors, as shown in Table 1.

Weight-Anchored Dataflows. Recall from Sec. 4.1.3 that we iterate through both the whole input and output tensors under weight-anchored dataflows. While we proceed by 1 element on the output tensor, we need to leap forward by s elements on the input tensor and also increment the starting input index (i.e., the first weight starts with the input at index 0, the second weight starts with the input at index 1, and so forth) for the computations associated with each weight element. This implies that each vector variable allocated for inputs saves $R \approx \frac{H}{s^2}$ memory reads, and each vector variable assigned to stash outputs saves R reads and R writes, respectively, per input channel block.
Guided by the heuristics, we derive the following observations for typical convolution layers:

Observation 1: Weight-anchored dataflows will gain the least performance improvement from auxiliary stationarities.

Observation 2: Output-anchored dataflows will likely yield better performance than input-anchored dataflows when both are fully optimized.

Observation 3: Under output-anchored dataflows, prioritizing input auxiliary stationarity and prioritizing weight auxiliary stationarity will yield similar results.

Observation 4: Under input-anchored dataflows, prioritizing output auxiliary stationarity will yield better performance than prioritizing weight auxiliary stationarity.

Observation 5: Under weight-anchored dataflows, prioritizing output auxiliary stationary will yield better performance than prioritizing input auxiliary stationary.

4.2 Extended Dataflow Implementations and Code Generator

Based upon the above observations, we develop a code generator to extend all three basic anchoring dataflows with auxiliary stationarities to further determine vector register allocation schemes, which is done by varying the number of vector registers allocated to each data type. We first allocate a subset of vector registers (the number of these vector registers ranges from 1 to $3n$, where n = vector variable size/vector register size, vector variable size $\in \{128, 256, 512\}$, and vector register size=128 in our implementation) to store the vector variables corresponding to the anchoring data type, then the remaining vector registers to the auxiliary data types.

Algorithm 4 Allocation sequence for inputs under secondary-unrolled output-anchored dataflows. (The same sequence applies for outputs under input-anchored dataflows when $s = 1$.)

```
Initialize the original (i.e., without secondary unrolling) allocation sequence with sequential row-major allocation.
for un in range[1, lcm(all #vector variables per row > stride)) do
    if # vector variables in the current row > stride then
        Rotate stash indices in this row left by stride
    else
        The sequence stays the same
    end if
end for
```

4.2.1 Implementation of Output-Anchored Dataflows.

For each output element under computation, we first determine if the required input and weight elements are already stashed in vector variables. If so, we perform the computation using those stashed data. Otherwise, we load the required data from memory into 2 vector variables of length $x \times element\_width$. Note that the sequence of vector variable usage between every two consecutive outputs is identical for weights but different for inputs. This means that we incur the cost of SIMD data transfer if we assign vector registers in the same way across all unrolled iterations of the weight loop, as the same position on the "window" covering all inputs involved in the computations of an output data would be matched to different inputs in two successive iterations.

To circumvent unnecessary data transfers between vector registers used for auxiliary input stationarity, we implement *secondary unrolling*, performed on the output loop with a magnitude of the least common multiple (lcm) of all numbers of input vector variables per row (in the input tensor) that are greater than s, so that each iteration of the secondary unrolled loop uses vector variables differently: the specific sequence of allocating input vector variables differs between the computations around two successive outputs if the number of input vector variables in that row is greater than s, and remains the same otherwise. Algorithm 4 demonstrates the sequences of vector variable allocations for input auxiliary stationarity across each secondary-unrolled iteration, and Fig. 6 provides a graphical example of secondary loop unrolling.

To further minimize data movements, we directly load vectors of input data to be newly stashed into their corresponding vector variables (thereby overwriting the previous data), instead of using new vector variables.

It is also worth noting that, through our observations, we found it advantageous to accumulate all results in a single vector register (instead of a scalar register) and execute the reduction sum operation only when all computations involving an output element have been completed. Although this approach consumes more vector registers, it ultimately saves costs related to performing a reduction sum operation on a scalar variable upon the completion of each computation.

Algorithm 5 summarizes the implementation of output-anchored dataflows.

Algorithm 5 Implementation of Output-anchored Dataflows

```
Require: inputs[ih · iw · ic], weights[ih · iw · ic · oc], outputs[oh · ow · ic · oc]
Require: numInStash, numWgtStash, x, s
  Prep 1: Initialize a total of numInStash input vector variables by loading data from the input tensor.
  Prep 2: Initialize a total of numWgtStash weight vector variables by loading data from the weight tensor.

  For each combination of one channel block (staring c) and one kernel (k)
  for h in oh by s do
    for w in ow by s do                                   ▷ Secondary Unroll
      Set the anchoring output vector variable to 0⃗
      for r in fh do                                      ▷ Unroll
        for s in fw do                                    ▷ Unroll
          if r · fh + fw < numInStash then
            Use the stashed vector as input
          else if r · fh + (fw − s) < numInStash then
            Overwrite a completely used input stash with the new input by vload(c · ih · iw + h · iw + w) and then use it as input
          elseinput = vload(c · ih · iw + h · iw + w)
          end if
          if r · fh + fw < nunWgtStash then
            Use the stashed vector as weight
          else weight = vload(c·oc·ih·iw+k·fh·fw+r·fw+s)
          end if
          res = vmul(input, weight)
          output = vadd(output, res)
        end for
      end for
      output[k · oh · ow + h · ow + w]+ = vaddv(output)
    end for
  end for
```

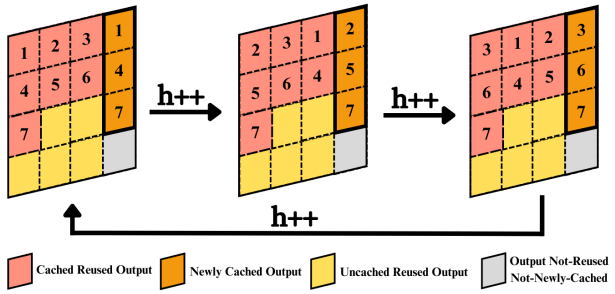


Figure 6. Secondary loop unrolling to bypass vector data transfer, using one channel for demonstration. h refers to the horizontal index of the current stationary input/output, and the numbers on the blocks are indices of the vector registers associated with that particular value.

Algorithm 6 Implementation of Input-anchored Dataflows

Require: $inputs[ih \cdot iw \cdot ic]$, $weights[ih \cdot iw \cdot ic \cdot oc]$, $outputs[oh \cdot ow \cdot ic \cdot oc]$
Require: $numInStash, numWgtStash, x, s$
Prep 1: Initialize a total of $numInStash$ input vector variables by loading data from the input tensor.
Prep 2: Initialize a total of $numWgtStash$ weight vector variables by loading data from the weight tensor.

For each combination of one channel block (staring c) and one kernel (k)
for h in ih **do**
  for w in iw **do** ▷ Secondary Unroll
    $input = vload(c \cdot ih \cdot iw + h \cdot iw + w)$
    for $((h', w'), (r, s))$ in $(assoc\_idx(h, w, c))$ **do**
      ▷ In reverse order
      ▷ Output and weight indices. See Fig. 5
      if $r \cdot fw + s \in stashedWeightsIndices$ **then**
        Use stashed vector as weight
      else $weight = vload(c \cdot oc \cdot ih \cdot iw + k \cdot fh \cdot fw + r \cdot fw + s)$
      end if
      if $h' \cdot iw + w' \in stashedOutputIndices$ **then**
        Use the stashed vector as output
        $res = vmul(input, weight)$
        $output = vadd(res, output)$
        if Last use of this output **then**
          $output[k \cdot oh \cdot ow + h \cdot ow + w] += vaddv(output)$
        end if
      else if $h' \cdot iw + w'$ to be newly stashed **then**
        Use a free vector as output
        $output = vmul(input, weight)$
      else $output[k \cdot oh \cdot ow + h \cdot ow + w] += vaddv(vmul(input, weight))$
      end if
    end for
    $output[k \cdot oh \cdot ow + h \cdot ow + w] += vaddv(output)$
  end for
end for

4.2.2 Implementation of Input-Anchored Dataflows.

Under input-anchored dataflows, we can allocate the remaining vector variables to both weights and outputs. When s is 1, the sequences of vector variable usage between every two consecutive inputs are identical for weight data but different for output data. Similar to the output-anchoring dataflows, this means that we incur the cost of vector data transfer if we consistently use variables in the same sequence. Therefore, again, we perform secondary unrolling on the output loop, following a similar procedure as described in Sec. 4.2.1, but with the sequence of weights in reverse. We write the stashed outputs back to memory when their usage is complete for this row, i.e., when the output is in the first column of the current window of computation. The pseudocode of Input-anchored dataflows is provided in Algorithm 6.

Algorithm 7 Implementation of Weight-anchored Dataflows

Require: $inputs[ih \cdot iw \cdot ic]$, $weights[ih \cdot iw \cdot ic \cdot oc]$, $outputs[oh \cdot ow \cdot ic \cdot oc]$
Require: $numInStash, numOutStashs$
Prep 1: Initialize a total of $numInStash$ input vector variables by loading data from the $input$ tensor.
Prep 2: Initialize a total of $numOutStash$ output vector variables by setting them to 0's.

For each combination of one channel block (staring c) and one kernel (k)
for fi in $fh \cdot fw$ **do** ▷ Unroll to Split the last iteration
  $weight = vload(c \cdot oc \cdot ih \cdot iw + k \cdot fh \cdot fw + fi)$
  for h in oh **do**
    for w in ow **do**
      Calculate ih and iw with $oh, ow, padding, s$
      if $ih \cdot input\_width + iw < numInStash$ **then**
        Use the stashed vector as input
      else $input = vload(c \cdot ih \cdot iw + h \cdot iw + w)$
      end if
      if $h \cdot ow + w < numOutStash$ **then**
        Use the stashed vector as output
        $output = vadd(vmul(input, weight))$
        if $fi == fh \cdot fw - 1$ **then**
          $outputs[k \cdot oh \cdot ow + h \cdot ow + w] += vaddv(output)$
        end if
      else $outputs[k \cdot oh \cdot ow + h \cdot ow + w] += vaddv(vmul(input, weight))$
      end if
    end for
  end for
end for

4.2.3 Implementation of Weight-Anchored Dataflows.

Similar to output- and input-anchored dataflows, we describe a concrete and general method to implement weight-anchored dataflows in Algorithm 7. For input and output auxiliary stationarity under weight-anchored dataflows, we always stash the earliest element that has not been stashed to exploit locality. We perform a loop split on the weight loop on top of unrolling to write stashed outputs back to memory only when their last usage is complete. When $s > 1$, inputs are reused once for every s weights.

4.3 End-to-End Optimization of Memory Layout Sequence

Consistent memory layout alignment across consecutive layers is essential for efficient neural network inference. Any

layout discrepancy entails the need for transformations, leading to additional overhead. To combat this issue, we resort to the commonly adopted dynamic programming approach based on searched results [3, 68, 72]. The algorithm hinges on minimizing layout transformations by using costs obtained from repeated runs of different scheduling schemes on each layer, which minimizes variance. The algorithm leverages these costs to determine optimal layouts that synchronize every two successive layers, thereby reducing the need for layout transformations.

In addition, we search for the optimal blocking schemes in compile time by running the program under each of the possible configurations and comparing their performance.

4.4 Code Generator Implementation

We develop a fully automated code generator that does not require expertise from the user. Given neural network layer configurations (type, tensor dimensions, precision), hardware information (ISA, SIMD vector size, number of vector registers), and the desired anchoring and auxiliary stationarities, the code generator generates C++ inference programs, following Algorithms 5, 6, or 7 to implement the corresponding dataflows using ARM Intrinsics.

5 Experiment Setup

We use 64-bit quad-core ARM systems with Neoverse-N1 CPUs that support the aarch64 architecture to quantitatively evaluate and compare dataflows implemented using our code generator. Our experiments encompass executing convolution layers with various combinations of the following parameters, as well as collecting end-to-end runtime results for neural networks, to facilitate a thorough and comprehensive evaluation and comparison of different dataflows. Each program was executed 100 times to obtain the average/median run time.

- **Input Size:** We focus on larger convolution layers that are time-consuming with input sizes of 56×56 and 112×112.
- **Weight Filter Size:** We use filters of sizes 3×3, 4×4, and 5×5, as these dimensions are most widely employed.
- **Stride:** We use strides of 1 and 2, as these values are the most common.
- **Number of Filters:** We tested with 128, 256, and 512 filters to compare the different dataflows across various numbers of filters.
- **Vector Lengths:** 128, 256, and 512, which are supported by modern ISAs such as ARM [64] and x86 [21, 43].

We use the GCC compiler [1] with the most aggressive optimization flags (`-O3, -march=armv8.2-a+sve2`) to compile all programs. We also experimented with clang/LLVM, upon which the Arm C++ Compiler is built. We observed results and limitations similar to those of GCC.

6 Results and Discussions

We present our experiment results in this section. Although our experiments were conducted for the ARM architecture, we expect that our approach and findings are applicable to other SIMD architectures. One piece of evidence is provided by additional experiments we performed for the x86 architecture, which yielded results consistent with those observed on ARM.

6.1 Validation of Heuristics

We generated programs that implement extended dataflows for various convolution layers in ARM Intrinsics and ran experiments following the setup described in Sec. 5 to validate the heuristics described in Sec. 4.

We primarily present the results for $s = 1$, and summarize the key results observed from experiments conducted using $s = 2$ below. (1) With output-anchored dataflows, the relative gains from weight and input auxiliary stationarities stay constant regardless of whether s is 1 or 2. (2) For weight-anchored dataflows, the improvement of extended dataflows over the basic, anchoring-only dataflow under $s = 2$ is expected to be less than that for $s = 1$. This can be explained by our heuristics in Sec. 4.1.4, (3) Under input-anchored dataflows, as s increases, the difference between the gains from weight and output auxiliary stationarity amplifies – we have empirically observed this behavior. (4) Considering the substantial performance gap of 5.39x between the basic OS and IS dataflows for $s = 2$, it is unlikely that input-anchored extended dataflows could surpass output-anchored extended dataflows in terms of performance.

6.1.1 Comparing Different Anchoring Stationarities.

Finding 1: Weight-anchored dataflows yield the least improvement from auxiliary dataflow optimizations and are consistently the slowest by a large magnitude.

Weight-anchored dataflows, even when fully optimized, significantly underperform in comparison to other anchoring stationarities (Fig. 7). Fully optimized output-anchored dataflow implementation is by median approximately 7.41x faster than their weight-anchored counterpart. However, when comparing the basic dataflows, we observe only a median performance difference of about 5 times between WS and OS, and roughly 2.91 times between WS and IS, given $s = 1$. This escalating disparity is attributed to the different performance enhancements yielded by adding auxiliary stationarities for different anchoring dataflows. The introduction of auxiliary stationarities results in a modest median improvement of around 1.08x for WS, while IS and OS

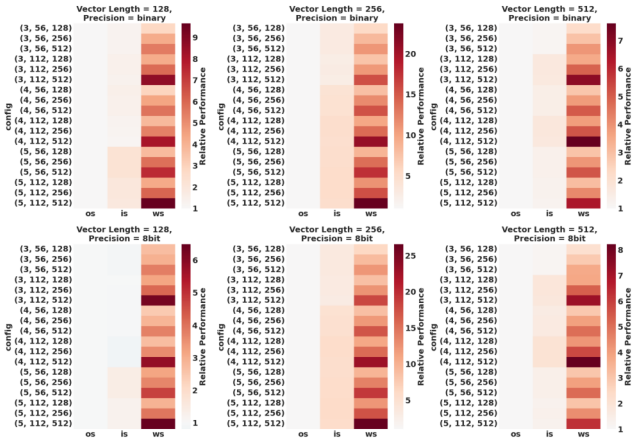


Figure 7. Relative Latency comparing the most optimized extended dataflows, normalized to the performance of OS.

enjoy more substantial median speedups of approximately x1.96 and x1.78 times, respectively. Moreover, we find that adding auxiliary stationarities to the basic WS dataflow can sometimes increase the run time. This is due to a low reuse frequency of the stashed auxiliary data and a more dominant increase in the code size. These results validate **Observation 1** derived from our heuristics.

> **Finding 2:** Output-anchored Dataflows outperform input-anchored Dataflows in the majority of the cases.

While IS seems to gain a larger performance improvement from the addition of auxiliary stationarities, output-anchored dataflows are still superior in terms of performance upon full optimization. For the same convolution layer configuration, optimized output-anchored dataflows are faster than input-anchored dataflows for around 90% of the cases, which validates **Observation 2**. The differences in performance for the remaining cases are small, most likely due to noises.

6.1.2 Findings Related to Auxiliary Stationarity.

Here, we compare different auxiliary stationarity schemes under each anchoring dataflow.

> **Finding 3:** Prioritizing stashing inputs or weights does not significantly impact performance under output-anchored dataflows.

This finding validates **Observation 3**. By comparing the latency of dataflows that prioritize allocation for weight auxiliary stationarity and the ones that prioritize input auxiliary stationarity, we observe that neither allocation scheme is consistently superior to the other, and the differences between the two schemes are small (within 6%).

> **Finding 4:** Allocating vector variables to outputs first improves performance compared to prioritizing allocation for weights under input-anchored dataflows.

By average, prioritizing stashing outputs yields an 8% performance gain, which becomes more evident as we increase the vector length. It follows that **Observation 4** is validated.

> **Finding 5:** Prioritizing output allocation yields only slightly better performance than prioritizing input allocation under weight-anchored dataflows.

We find that under almost all cases, prioritizing output auxiliary stationarity brings a performance gain of up to 3% over prioritizing weight auxiliary stationarity. This validates **Observation 5**; however, the differences between these two schemes are small.

Algorithm 8 Optimized Dataflow: Output Anchored Stationarity with Weight Auxiliary Stationarity

Require: $numVecReg$, $vecVarSize$, $vecRegSize$
$regsPerVar = vecVarSize \,/\, vecRegSize$
$numVarAvailable = numVecReg \,/\, regsPerVar$

$auxVarAvailable = numVarAvailable - 3$

1. Use **output stationary** as the anchoring stationarity
2. Allocate $auxVarAvailable$ vector variables **first to weight and then to input** (if some vector registers are still available).

6.1.3 Optimized Dataflow.

From our analyses and results, we conclude that OS-anchored dataflow with auxiliary weight stationarity is the most optimized dataflow in our study. While there is generally little difference between prioritizing auxiliary WS and prioritizing auxiliary IS, we find the former to yield better code readability and more regular instruction patterns. Algorithm 8 summarizes this dataflow.

6.2 Neural Network Speedup against State-of-the-Art Implementations

Applying end-to-end optimizations discussed in Sec. 4.3, we compare our approach to state-of-the-art baselines.

For INT8 neural networks, we use TVM as one of the baselines. TVM is a highly optimized machine learning compiler stack for efficient neural network deployment across various hardware platforms [13]. We compare the end-to-end inference latency of variants of ResNet [34] (Resnet-18 and Resnet-34) and VGG [93] (VGG-11, VGG-13, and VGG-16) with TVM-autotuned implementations (we use GridSearchTuner as the KernelTuner, which enumerates through the entire search space [27]) and untuned implementations (TVM

default)[2]. We set TVM to target the architecture and SIMD extension to match the physical machines used for our experiments. Across all network architectures and numbers of threads, we observe a ~3x speedup over TVM's implementations, and up to ~14x over its untuned implementation. Moreover, our multi-threaded scheme yields comparable scalability. We also compare the end-to-end results with programs generated by GCC/clang (with the highest level of optimization and auto-vectorization enabled). Ours achieve significant (4x-6x) speedup.

For the evaluation of binary neural networks, we compare the inference latency of our implementations with Cowan et al.'s TVM-based bitserial implementations [23]. Since the code released by Cowan et al. only works for convolution layers on CPUs (while their end-to-end code generation tool targets Raspberry Pi and is not applicable to CPUs), we only perform this comparison for convolution layers. Bitserial implementations, although optimized for low-power consumption, do not offer satisfactory inference speed. Notably, our implementations are over 12x faster for various convolution layers. Based on the end-to-end results reported in their paper (which incorporates additional optimizations through microkernel synthesis) [23], we anticipate that our implementations will still outperform theirs by a large margin (6x or higher) in the end-to-end comparisons. We also compare our implementations of various convolution layers in VGG against those from [68], which already outperforms Openvino, MXNet, and TensorFlow across numerous networks (ResNets, VGGs, DenseNets) and CPUs (Intel, AMD, and ARM), and ours achieve up to 4.8x speedup.

7 Related Work

This section offers an overview of existing techniques for accelerating neural network inference. Our work already employs quantization [17, 22, 33, 36], vectorization [39, 60, 84, 87, 108], tiling/blocking [11, 96], and operator fusion [45, 83, 85]. For operator fusion and quantization, convolution and batch normalization are mathematically fused by preprocessing parameters, and activation and quantization for each output can be performed on the accumulated results before we store the quantized activation back to memory [8, 13]. We compare and contrast our work with other related efforts.

Unroll-and-Jam. Unroll-and-jam reduces memory access costs by reordering instructions without breaking data dependencies [9, 10, 76], which can enhance the performance of convolution and fully-connected layers in DNNs [13, 74, 98]. Our technique bypasses unneeded load instructions previously handled by jamming, and further jamming (performed by GCC in our implementation) can be applied on top of our technique to reduce latency.

[2]We do not use Ansor, another TVM tuner, because Resnet-18, Resnet-34, VGG-11,and VGG-13 became 3.76x, 7.28x, 3.90x, and 3.89x slower when it is used.

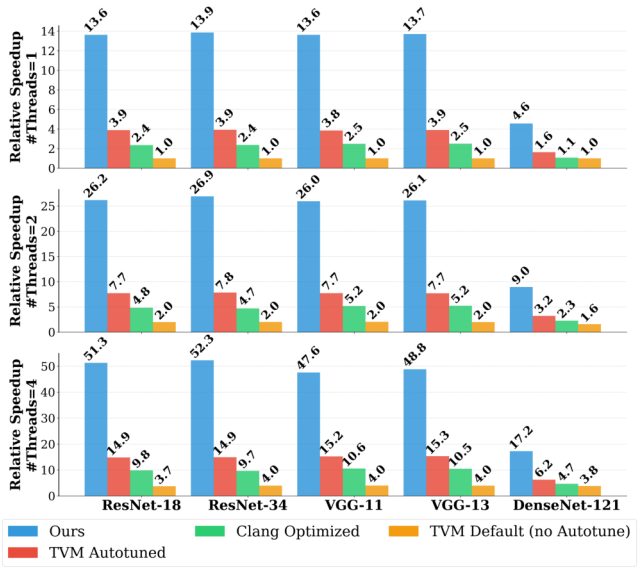


Figure 8. End-to-end relative speedups for INT8 neural networks from our techniques, normalized to TVM's default mode without autotune (Note: TVM's default mode did not work for DenseNet-121, so we used a different tuner called TaskScheduler, and the first tuning trial is used as the baseline).

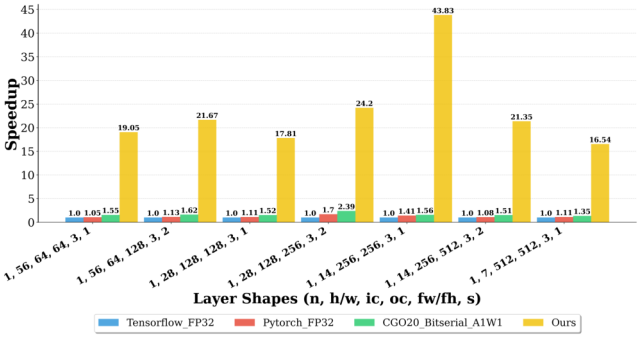


Figure 9. Layer-wise latency comparisons for binary ResNet Workloads between Ours and Cowan et al [23].

Winograd Convolution. Winograd convolution reduces the complexity of convolution operations [4, 67, 78, 101, 104], and there exist various optimizations of its implementation on CPUs [46, 61, 62, 73]. Utilizing a similar concept of reusing data to speed up convolution inference, DREW [102] optimizes Winograd convolution by clustering data and reusing computed results and trades off accuracy and inference performance. In contrast, our method retains accuracy and can be applied to various SIMD architectures. Moreover, while standard Winograd convolutions encounter challenges with quantization [16, 25, 26, 62], our technique does not suffer from this limitation.

Transformer Optimizations. Transformers have revolutionized several areas of machine learning [49, 88, 94, 97, 105, 107]. However, optimizing their performance, particularly on CPUs, remains a significant challenge [24, 44, 47, 99]. Efforts to date include pruning [53, 54, 75, 115], quantization [7, 19, 70, 81], knowledge distillation [14, 48, 66, 100], architecture search [65, 99, 109], GEMM optimizations [24, 37, 47], and hardware-level optimizations [44, 113]. Moreover, while there exist previous works on studying dataflows for transformers on other hardware platforms [71, 89, 110, 114], no dataflow work has been done on CPUs to the best of our knowledge. Our technique is orthogonal to and may be combined with other Transformer optimization techniques such as GEMM optimizations (e.g., [24]).

Intel AMX Extension. Intel's AMX [43] is designed to accelerate tile-based operations on CPUs (e.g., matrix multiplication and accumulation along multiple dimensions), and is only available in high-performance processors like the 4th Generation Xeon Scalable Processors [41]. Our research focuses on prevalent SIMD architectures. It is essential to develop dataflows that maximize data reuse opportunities in AMX to further optimize its performance, and our methodology may be extended for this purpose.

8 Conclusions

In this paper, we present the first approach to systematically explore dataflows to achieve efficient neural network inference using SIMD capabilities. Our experiment results demonstrate significant performance improvements over state-of-the-art implementations. We anticipate that this work will catalyze further investigation of dataflows to reduce inference time on contemporary CPU architectures [3]y.

Acknowledgements

We thank all anonymous reviewers for their constructive feedback and Clark (Yilin) Xu of Brandeis University for his help with graphics creation. This work is partially supported by the Quad Undergraduate Research Fellowship and Metcalf Fellowship Grant at the University of Chicago.

References

[1] [n. d.]. GNU Compiler Collection. https://gcc.gnu.org/. Accessed: 2023-08-28.

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

[3] Byung Hoon Ahn, Jinwon Lee, Jamie Menjay Lin, Hsin-Pai Cheng, Jilei Hou, and Hadi Esmaeilzadeh. 2020. Ordering chaos: Memory-aware scheduling of irregularly wired neural networks for edge devices. *Proceedings of Machine Learning and Systems* 2 (2020), 44–57.

[4] Syed Asad Alam, Andrew Anderson, Barbara Barabasz, and David Gregg. 2022. Winograd convolution for deep neural networks: Efficient point selection. *ACM Transactions on Embedded Computing Systems* 21, 6 (2022), 1–28.

[5] M. Alwani, H. Chen, M. Ferdman, and P. Milder. 2016. Fused-layer CNN accelerators. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–12. https://doi.org/10.1109/MICRO.2016.7783725

[6] Hossein Amiri and Asadollah Shahbahrami. 2020. SIMD programming using Intel vector extensions. *J. Parallel and Distrib. Comput.* 135 (2020), 83–100.

[7] Yelysei Bondarenko, Markus Nagel, and Tijmen Blankevoort. 2021. Understanding and overcoming the challenges of efficient transformer quantization. *arXiv preprint arXiv:2109.12948* (2021).

[8] Xuyi Cai, Ying Wang, and Lei Zhang. 2022. Optimus: An operator fusion framework for deep neural networks. *ACM Transactions on Embedded Computing Systems* 22, 1 (2022), 1–26.

[9] Steve Carr, Chen Ding, and Philip Sweany. 1996. Improving software pipelining with unroll-and-jam. In *Proceedings of HICSS-29: 29th Hawaii International Conference on System Sciences*, Vol. 1. IEEE, 183–192.

[10] Steve Carr and Yiping Guan. 1997. Unroll-and-jam using uniformly generated sets. In *Proceedings of 30th Annual International Symposium on Microarchitecture*. IEEE, 349–357.

[11] K. Chellapilla, S. Puri, and P. Simard. 2006. High Performance Convolutional Neural Networks for Document Processing. In *Tenth International Workshop on Frontiers in Handwriting Recognition*.

[12] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. DianNao: A Small-Footprint High-Throughput Accelerator for Ubiquitous Machine-Learning. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (Salt Lake City, Utah, USA) *(ASPLOS '14)*. Association for Computing Machinery, New York, NY, USA, 269–284. https://doi.org/10.1145/2541940.2541967

[13] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, ..., and Y. Chen. 2018. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.

[14] Xianing Chen, Qiong Cao, Yujie Zhong, Jing Zhang, Shenghua Gao, and Dacheng Tao. 2022. Dearkd: data-efficient early knowledge distillation for vision transformers. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 12052–12062.

[15] Y.-H. Chen, J. Emer, and V. Sze. 2017. Using Dataflow to Optimize Energy Efficiency of Deep Neural Network Accelerators. *IEEE Micro* 37, 3 (2017), 12–21. https://doi.org/10.1109/MM.2017.54

[16] Vladimir Chikin and Vladimir Kryzhanovskiy. 2022. Channel balancing for accurate quantization of winograd convolutions. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 12507–12516.

[17] Jiahao Choi, Mostafa El-Khamy, and Jungwon Lee. 2018. Towards the limit of network quantization. In *International Conference on Learning Representations*.

[18] Francois Chollet. 2017. Xception: Deep Learning with Depthwise Separable Convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 1251–1258.

[3]Our implementation can be found at https://github.com/YLab-UChicago/YFlows

[19] Insoo Chung, Byeongwook Kim, Yoonjung Choi, Se Jung Kwon, Yongkweon Jeon, Baeseong Park, Sangha Kim, and Dongsoo Lee. 2020. Extremely low bit transformer quantization for on-device neural machine translation. *arXiv preprint arXiv:2009.07453* (2020).

[20] Larq Contributors. 2023. Larq: An Open-Source Library for Training Binarized Neural Networks. https://github.com/larq/larq. GitHub repository.

[21] Intel Corporation. 2023. Intel Intrinsics Guide. https://software.intel.com/sites/landingpage/IntrinsicsGuide/. Accessed: 2023-05-19.

[22] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. 2015. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in Neural Information Processing Systems*. 3123–3131.

[23] Meghan Cowan, Thierry Moreau, Tianqi Chen, James Bornholt, and Luis Ceze. 2020. Automatic Generation of High-Performance Quantized Machine Learning Kernels. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization* (San Diego, CA, USA) *(CGO 2020)*. Association for Computing Machinery, New York, NY, USA, 305–316. https://doi.org/10.1145/3368826.3377912

[24] Dave Dice and Alex Kogan. 2021. Optimizing Inference Performance of Transformers on CPUs. arXiv:2102.06621 [cs.CL]

[25] Javier Fernandez-Marques. 2020. *Even Faster Convolutions: Winograd Convolutions meet Integer Quantization and Architecture Search*. https://community.arm.com/arm-research/b/articles/posts/even-faster-convolutions-winograd-convolutions-meet-integer-quantization-and-architecture-search Accessed: [Your Access Date Here].

[26] Javier Fernandez-Marques, Paul Whatmough, Andrew Mundy, and Matthew Mattina. 2020. Searching for winograd-aware quantized networks. *Proceedings of Machine Learning and Systems* 2 (2020), 14–29.

[27] Apache Software Foundation. 2023. tvm.autotvm — tvm 0.14.dev0 documentation. https://tvm.apache.org/docs/reference/api/python/autotvm.html Accessed: 2023-08-31.

[28] Venkatraman Govindaraju, Tony Nowatzki, and Karthikeyan Sankaralingam. 2013. Breaking SIMD shackles with an exposed flexible microarchitecture and the access execute PDG. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*. IEEE, 341–351.

[29] Ramyad Hadidi, Jiashen Cao, Yilun Xie, Bahar Asgari, Tushar Krishna, and Hyesoon Kim. 2019. Characterizing the deployment of deep neural networks on commercial edge devices. In *2019 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 35–48.

[30] Ameer Haj-Ali, Nesreen K Ahmed, Ted Willke, Yakun Sophia Shao, Krste Asanovic, and Ion Stoica. 2020. Neurovectorizer: End-to-end vectorization with deep reinforcement learning. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*. 242–255.

[31] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. 2016. EIE: efficient inference engine on compressed deep neural network. In *Proceedings of the 43rd International Symposium on Computer Architecture*. 243–254.

[32] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. 2016. EIE: efficient inference engine on compressed deep neural network. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16)*. 243–254. https://doi.org/10.1109/ISCA.2016.30

[33] Song Han, Huizi Mao, and William J Dally. 2016. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. In *International Conference on Learning Representations*.

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

[35] John L. Hennessy and David A. Patterson. 2011. *Computer Architecture: A Quantitative Approach*. Elsevier.

[36] Elad Hoffer, Itay Hubara, and Daniel Soudry. 2017. Train longer, generalize better: closing the generalization gap in large batch training of neural networks. In *Advances in Neural Information Processing Systems*. 1731–1741.

[37] Seongmin Hong, Seungjae Moon, Junsoo Kim, Sungjae Lee, Minsub Kim, Dongsoo Lee, and Joo-Young Kim. 2022. DFX: A Low-latency Multi-FPGA Appliance for Accelerating Transformer-based Text Generation. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 616–630.

[38] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).

[39] Y. Hu and et al. 2018. BitFlow: Exploiting Vector Parallelism for Binary Neural Networks on CPU. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 244–253. https://doi.org/10.1109/IPDPS.2018.00034

[40] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. 2016. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size. *arXiv preprint arXiv:1602.07360* (2016).

[41] Intel. 2023. 4th Gen Xeon Scalable Processors. https://www.intel.com/content/www/us/en/products/docs/processors/xeon-accelerated/4th-gen-xeon-scalable-processors.html Accessed: Aug 23, 2023.

[42] Intel. 2023. Intel oneDNN Developer Guide and Reference. https://github.com/oneapi-src/oneDNN/blob/master/src/cpu/ Accessed: 18-05-2023.

[43] Intel. 2023. Intel® Advanced Matrix Extensions Overview. https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/advanced-matrix-extensions/overview.html Accessed: Aug 23, 2023.

[44] Andrei Ivanov, Nikoli Dryden, Tal Ben-Nun, Shigang Li, and Torsten Hoefler. 2021. Data movement is all you need: A case study on optimizing transformers. *Proceedings of Machine Learning and Systems* 3 (2021), 711–732.

[45] Y. Jia, S. Yin, C. He, and T. Zhang. 2018. MLFusion: Multi-Layer Fusion for FPGA-Based CNN Accelerators. In *Proceedings of the 27th International Conference on Field Programmable Logic and Applications (FPL)*.

[46] 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.

[47] Jiazhi Jiang, Jiangsu Du, Dan Huang, Dongsheng Li, Jiang Zheng, and Yutong Lu. 2022. Characterizing and optimizing transformer inference on arm many-core processor. In *Proceedings of the 51st International Conference on Parallel Processing*. 1–11.

[48] Yidi Jiang, Bidisha Sharma, Maulik Madhavi, and Haizhou Li. 2021. Knowledge distillation from bert transformer to speech transformer for intent classification. *arXiv preprint arXiv:2108.02598* (2021).

[49] Salman Khan, Muzammal Naseer, Munawar Hayat, Syed Waqas Zamir, Fahad Shahbaz Khan, and Mubarak Shah. 2022. Transformers in vision: A survey. *ACM computing surveys (CSUR)* 54, 10s (2022), 1–41.

[50] Seonggun Kim and Hwansoo Han. 2012. Efficient SIMD code generation for irregular kernels. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*. 55–64.

[51] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems* 25 (2012).

[52] Hyoukjun Kwon, Prasanth Chatarasi, Michael Pellauer, Angshuman Parashar, Vivek Sarkar, and Tushar Krishna. 2019. Understanding reuse, performance, and hardware cost of dnn dataflow: A data-centric approach. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 754–768.

[53] Woosuk Kwon, Sehoon Kim, Michael W Mahoney, Joseph Hassoun, Kurt Keutzer, and Amir Gholami. 2022. A fast post-training pruning framework for transformers. *Advances in Neural Information Processing Systems* 35 (2022), 24101–24116.

[54] François Lagunas, Ella Charlaix, Victor Sanh, and Alexander M Rush. 2021. Block pruning for faster transformers. *arXiv preprint arXiv:2109.04838* (2021).

[55] Nicholas D Lane, Sourav Bhattacharya, Petko Georgiev, Claudio Forlivesi, and Fahim Kawsar. 2016. Deepx: A software accelerator for low-power deep learning inference on mobile devices. *Proceedings of the 15th International Conference on Information Processing in Sensor Networks* (2016), 1–12.

[56] Samuel Larsen and Saman Amarasinghe. 2000. Exploiting superword level parallelism with multimedia instruction sets. *Acm Sigplan Notices* 35, 5 (2000), 145–156.

[57] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization* (Palo Alto, California) *(CGO '04)*. IEEE Computer Society, USA, 75.

[58] Yann LeCun, Larry Jackel, Leon Bottou, A Brunot, Corinna Cortes, John Denker, Harris Drucker, Isabelle Guyon, UA Muller, Eduard Sackinger, et al. 1995. Comparison of learning algorithms for handwritten digit recognition. In *International conference on artificial neural networks*, Vol. 60. Perth, Australia, 53–60.

[59] Sung-Jin Lee, Sang-Soo Park, and Ki-Seok Chung. 2018. Efficient SIMD implementation for accelerating convolutional neural network. In *Proceedings of the 4th International Conference on Communication and Information Processing*. 174–179.

[60] Sung-Jin Lee, Sang-Soo Park, and Ki-Seok Chung. 2018. Efficient SIMD Implementation for Accelerating Convolutional Neural Network. In *Proceedings of the 4th International Conference on Communication and Information Processing* (Qingdao, China) *(ICCIP '18)*. Association for Computing Machinery, New York, NY, USA, 174–179. https://doi.org/10.1145/3290420.3290444

[61] Dongsheng Li, Dan Huang, Zhiguang Chen, and Yutong Lu. 2021. Optimizing massively parallel winograd convolution on arm processor. In *Proceedings of the 50th International Conference on Parallel Processing*. 1–12.

[62] Guangli Li, Zhen Jia, Xiaobing Feng, and Yida Wang. 2021. Lowino: Towards efficient low-precision winograd convolutions on modern cpus. In *Proceedings of the 50th International Conference on Parallel Processing*. 1–11.

[63] ARM Limited. 2023. Architectures | Instruction sets | Intrinsics. https://developer.arm.com/architectures/instruction-sets/intrinsics/ Accessed: 2023-08-27.

[64] Arm Limited. 2023. NEON Programmer's Guide for Armv8-A. https://developer.arm.com/documentation/100069/0101/. Accessed: 2023-05-19.

[65] Jihao Liu, Xin Huang, Guanglu Song, Hongsheng Li, and Yu Liu. 2022. Uninet: Unified architecture search with convolution, transformer, and mlp. In *European Conference on Computer Vision*. Springer, 33–49.

[66] Ruiping Liu, Kailun Yang, Alina Roitberg, Jiaming Zhang, Kunyu Peng, Huayao Liu, and Rainer Stiefelhagen. 2022. TransKD: Transformer knowledge distillation for efficient semantic segmentation. *arXiv preprint arXiv:2202.13393* (2022).

[67] Xingyu Liu, Jeff Pool, Song Han, and William J Dally. 2018. Efficient sparse-winograd convolutional neural networks. *arXiv preprint arXiv:1802.06367* (2018).

[68] Y. Liu, Y. Wang, R. Yu, M. Li, V. Sharma, and Y. Wang. 2019. Optimizing CNN model inference on CPUs. In *Proc. USENIX Annu. Tech. Conf.* 1025–1040.

[69] Zhengyang Liu, Stefan Mada, and John Regehr. 2023. Minotaur: A SIMD-Oriented Synthesizing Superoptimizer. arXiv:2306.00229 [cs.PL]

[70] Zhenhua Liu, Yunhe Wang, Kai Han, Wei Zhang, Siwei Ma, and Wen Gao. 2021. Post-training quantization for vision transformer. *Advances in Neural Information Processing Systems* 34 (2021), 28092–28103.

[71] Liqiang Lu, Yicheng Jin, Hangrui Bi, Zizhang Luo, Peng Li, Tao Wang, and Yun Liang. 2021. Sanger: A co-design framework for enabling sparse attention using reconfigurable architecture. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 977–991.

[72] Liqiang Lu and Yun Liang. 2018. SpWA: An Efficient Sparse Winograd Convolutional Neural Networks Accelerator on FPGAs. In *Proceedings of the 55th Annual Design Automation Conference* (San Francisco, California) *(DAC '18)*. Association for Computing Machinery, New York, NY, USA, Article 135, 6 pages. https://doi.org/10.1145/3195970.3196120

[73] Partha Maji, Andrew Mundy, Ganesh Dasika, Jesse Beu, Matthew Mattina, and Robert Mullins. 2019. Efficient winograd or cook-toom convolution kernel implementation on widely used mobile cpus. In *2019 2nd Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications (EMC2)*. IEEE, 1–5.

[74] Ankush Mandal. 2017. *Optimizing Convolutions in State-of-the-Art Convolutional Neural Networks on Intel Xeon Phi*. Ph. D. Dissertation. Rice University.

[75] Jiachen Mao, Huanrui Yang, Ang Li, Hai Li, and Yiran Chen. 2021. Tprune: Efficient transformer pruning for mobile devices. *ACM Transactions on Cyber-Physical Systems* 5, 3 (2021), 1–22.

[76] John Mellor-Crummey and John Garvin. 2004. Optimizing sparse matrix–vector product computations using unroll and jam. *The International Journal of High Performance Computing Applications* 18, 2 (2004), 225–236.

[77] Charith Mendis and Saman Amarasinghe. 2018. GoSLP: Globally Optimized Superword Level Parallelism Framework. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 110 (oct 2018), 28 pages. https://doi.org/10.1145/3276480

[78] Lingchuan Meng and John Brothers. 2019. Efficient winograd convolution via integer arithmetic. *arXiv preprint arXiv:1901.01965* (2019).

[79] Sparsh Mittal, Poonam Rajput, and Sreenivas Subramoney. 2021. A survey of deep learning on CPUs: opportunities and co-optimizations. *IEEE Transactions on Neural Networks and Learning Systems* 33, 10 (2021), 5095–5115.

[80] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library.

[81] Gabriele Prato, Ella Charlaix, and Mehdi Rezagholizadeh. 2019. Fully quantized transformer for improved translation. (2019).

[82] Yu Pu, Yifan He, Zhenyu Ye, Sebastian Moreno Londono, Anteneh Alemu Abbo, Richard Kleihorst, and Henk Corporaal. 2011. From Xetal-II to Xetal-Pro: On the road toward an ultralow-energy and high-throughput SIMD processor. *IEEE Transactions on Circuits and Systems for Video Technology* 21, 4 (2011), 472–484.

[83] Y. Qiao, Y. Zhang, J. Wang, T. Tang, and Y. Wang. 2019. Layer Fusion for Memory-Efficient Inference of Convolutional Neural Networks on GPUs. In *International Symposium on Benchmarking, Measuring and Optimizing (Bench)*.

[84] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[85] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler. 2016. vDNN: Virtualized Deep Neural Networks for Scalable, Memory-Efficient Neural Network Design. In *Advances in Neural Information Processing Systems (NIPS)*.

[86] Ananda Samajdar, Yuhao Zhu, Paul Whatmough, Matthew Mattina, and Tushar Krishna. 2018. Scale-sim: Systolic cnn accelerator simulator. *arXiv preprint arXiv:1811.02883* (2018).

[87] Alexandre de Limas Santana, Adrià Armejach, and Marc Casas. 2023. Efficient Direct Convolution Using Long SIMD Instructions. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming* (Montreal, QC, Canada) *(PPoPP '23)*. Association for Computing Machinery, New York, NY, USA, 342–353. https://doi.org/10.1145/3572848.3577435

[88] Fahad Shamshad, Salman Khan, Syed Waqas Zamir, Muhammad Haris Khan, Munawar Hayat, Fahad Shahbaz Khan, and Huazhu Fu. 2023. Transformers in medical imaging: A survey. *Medical Image Analysis* (2023), 102802.

[89] Guan Shen, Jieru Zhao, Quan Chen, Jingwen Leng, Chao Li, and Minyi Guo. 2022. SALO: an efficient spatial accelerator enabling hybrid sparse attention mechanisms for long sequences. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*. 571–576.

[90] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. 2016. Edge Computing: Vision and Challenges. *IEEE Internet of Things Journal* 3, 5 (2016), 637–646. https://doi.org/10.1109/JIOT.2016.2579198

[91] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. 2016. Edge computing: Vision and challenges. *IEEE internet of things journal* 3, 5 (2016), 637–646.

[92] Laurent Sifre and Stéphane Mallat. 2014. Rigid-motion scattering for texture classification. *arXiv preprint arXiv:1403.1687* (2014).

[93] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[94] Cem Subakan, Mirco Ravanelli, Samuele Cornell, Mirko Bronzi, and Jianyuan Zhong. 2021. Attention is all you need in speech separation. In *ICASSP 2021-2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 21–25.

[95] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. 2017. Efficient processing of deep neural networks: A tutorial and survey. *Proc. IEEE* 105, 12 (2017), 2295–2329.

[96] N. Vasilache, J. Johnson, M. Mathieu, S. Chintala, S. Piantino, and Y. LeCun. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. In *International Conference on Learning Representations (ICLR)*.

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

[98] Anand Venkat, Tharindu Rusira, Raj Barik, Mary Hall, and Leonard Truong. 2019. SWIRL: High-performance many-core CPU code generation for deep neural networks. *The International Journal of High Performance Computing Applications* 33, 6 (2019), 1275–1289.

[99] Hanrui Wang, Zhanghao Wu, Zhijian Liu, Han Cai, Ligeng Zhu, Chuang Gan, and Song Han. 2020. Hat: Hardware-aware transformers for efficient natural language processing. *arXiv preprint arXiv:2005.14187* (2020).

[100] Wenhui Wang, Furu Wei, Li Dong, Hangbo Bao, Nan Yang, and Ming Zhou. 2020. Minilm: Deep self-attention distillation for task-agnostic compression of pre-trained transformers. *Advances in Neural Information Processing Systems* 33 (2020), 5776–5788.

[101] Shmuel Winograd. 1980. *Arithmetic complexity of computations*. Vol. 33. Siam.

[102] Ruofan Wu, Feng Zhang, Jiawei Guan, Zhen Zheng, Xiaoyong Du, and Xipeng Shen. 2022. DREW: Efficient Winograd CNN Inference with Deep Reuse. In *Proceedings of the ACM Web Conference 2022* (Virtual Event, Lyon, France) *(WWW '22)*. Association for Computing Machinery, New York, NY, USA, 1807–1816. https://doi.org/10.1145/3485447.3511985

[103] Yao Xiao, Nesreen Ahmed, Mihai Capotă, Guixiang Ma, Theodore L Willke, Shahin Nazarian, and Paul Bogdan. 2022. Structural Code Representation Learning for Auto-Vectorization. (2022).

[104] Da Yan, Wei Wang, and Xiaowen Chu. 2020. Optimizing batched winograd convolution on GPUs. In *Proceedings of the 25th ACM SIGPLAN symposium on principles and practice of parallel programming*. 32–44.

[105] Chenghao Yang, Hongyuan Mei, and Jason Eisner. 2021. Transformer embeddings of irregularly spaced events and their participants. *arXiv preprint arXiv:2201.00044* (2021).

[106] Dingqing Yang, Amin Ghasemazar, Xiaowei Ren, Maximilian Golub, Guy Lemieux, and Mieszko Lis. 2020. Procrustes: a dataflow and accelerator for sparse deep neural network training. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 711–724.

[107] Xi Yang, Jiang Bian, William R Hogan, and Yonghui Wu. 2020. Clinical concept extraction using transformers. *Journal of the American Medical Informatics Association* 27, 12 (2020), 1935–1942.

[108] Xuan Yang, Mingyu Gao, Qiaoyi Liu, Jeff Setter, Jing Pu, Ankita Nayak, Steven Bell, Kaidi Cao, Heonjae Ha, Priyanka Raina, Christos Kozyrakis, and Mark Horowitz. 2020. Interstellar: Using Halide's Scheduling Language to Analyze DNN Accelerators. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. 369–383. https://doi.org/10.1145/3373376.3378514

[109] Yichun Yin, Cheng Chen, Lifeng Shang, Xin Jiang, Xiao Chen, and Qun Liu. 2021. Autotinybert: Automatic hyper-parameter optimization for efficient pre-trained language models. *arXiv preprint arXiv:2107.13686* (2021).

[110] Haoran You, Zhanyi Sun, Huihong Shi, Zhongzhi Yu, Yang Zhao, Yongan Zhang, Chaojian Li, Baopu Li, and Yingyan Lin. 2023. Vitcod: Vision transformer acceleration via dedicated algorithm and accelerator co-design. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 273–286.

[111] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. 2017. ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices. arXiv:1707.01083 [cs.CV]

[112] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. 2018. ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices. *Proceedings of the IEEE conference on computer vision and pattern recognition* (2018), 6848–6856.

[113] Zining Zhang, Yao Chen, Bingsheng He, and Zhenjie Zhang. 2023. NIOT: A Novel Inference Optimization of Transformers on Modern CPUs. In *IEEE Transactions on Parallel and Distributed Systems*, Vol. 34. 1982–1995. Issue 6.

[114] Zhongyu Zhao, Rujian Cao, Ka-Fai Un, Wei-Han Yu, Pui-In Mak, and Rui P Martins. 2022. An fpga-based transformer accelerator using output block stationary dataflow for object recognition applications. *IEEE Transactions on Circuits and Systems II: Express Briefs* 70, 1 (2022), 281–285.

[115] Mingjian Zhu, Yehui Tang, and Kai Han. 2021. Vision transformer pruning. *arXiv preprint arXiv:2104.08500* (2021).

Received 09-NOV-2023; accepted 2023-12-23