Efficient Direct Convolution Using Long SIMD Instructions

Alexandre de Limas Santana
Barcelona Supercomputing Center
Barcelona, Catalunya, Spain
Universitat Politècnica de Catalunya
Barcelona, Catalunya, Spain
alexandre.delimassantana@bsc.es

Adrià Armejach
Barcelona Supercomputing Center
Barcelona, Catalunya, Spain
Universitat Politècnica de Catalunya
Barcelona, Catalunya, Spain
adria.armejach@bsc.es

Marc Casas
Barcelona Supercomputing Center
Barcelona, Catalunya, Spain
Universitat Politècnica de Catalunya
Barcelona, Catalunya, Spain
marc.casas@bsc.es

Abstract

This paper demonstrates that state-of-the-art proposals to compute convolutions on architectures with CPUs supporting SIMD instructions deliver poor performance for long SIMD lengths due to frequent cache conflict misses. We first discuss how to adapt the state-of-the-art SIMD direct convolution to architectures using long SIMD instructions and analyze the implications of increasing the SIMD length on the algorithm formulation. Next, we propose two new algorithmic approaches: the *Bounded Direct Convolution (*`BDC`*)*, which adapts the amount of computation exposed to mitigate cache misses, and the *Multi-Block Direct Convolution (*`MBDC`*)*, which redefines the activation memory layout to improve the memory access pattern. We evaluate `BDC`, `MBDC`, the state-of-the-art technique, and a proprietary library on an architecture featuring CPUs with 16,384-bit SIMD registers using ResNet convolutions. Our results show that `BDC` and `MBDC` achieve respective speed-ups of 1.44× and 1.28× compared to the state-of-the-art technique for ResNet-101, and 1.83× and 1.63× compared to the proprietary library.

CCS Concepts: • **Theory of computation → Design and analysis of algorithms**; • **Computer systems organization → Single instruction, multiple data**.

Keywords: high-performance convolutions, software optimization, SIMD architectures, vector architectures

1 Introduction

Convolution kernels are fundamental building blocks of Deep Neural Networks (DNNs). Their highly parallel nature makes them a very appealing option for parallel architectures exploiting Single Instruction Multiple Data (SIMD) parallelism. Processors equipped with CPUs supporting SIMD instructions have become critical components of parallel architectures applied on modern supercomputers [24], which has motivated numerous research efforts focused on accelerating convolutions on architectures featuring SIMD instructions [8, 10, 31]. While this body of work has focused on SIMD instructions operating on 512-bit registers, there is a trend toward SIMD architectures implementing registers larger than 512 bits. Emerging Instruction Set Architectures (ISAs) like the ARM Scalable Vector Extension (SVE) [26] or the RISC V "V" vector extension [6], which support SIMD instructions operating on very large registers, and commercial products featuring 2KB registers like the SX-Aurora processor [30], confirm this trend towards CPUs supporting SIMD instructions operating on large registers.

This paper demonstrates that state-of-the-art approaches to compute convolutions on CPUs supporting SIMD instructions [10] deliver poor performance when operating on long SIMD architectures. We show that the poor performance originates from unnecessary associations of the architecture SIMD length to optimization variables, causing large memory footprints and memory access patterns with poor locality and cache *conflict misses* [13].

We propose two novel algorithms to efficiently run convolution workloads on long SIMD architectures and overcome the two main issues of state-of-the-art approaches. The first algorithm, the *Bounded Direct Convolution* (`BDC`), prevents memory access patterns from triggering a large number of cache misses by throttling down the register blocking optimization while still exposing enough computation to avoid stalling the floating-point functional units. The second algorithm, the *Multi-Block Direct Convolution* (`MBDC`), redefines the tensor's memory layout in favor of regular memory access patterns, eliminating the possibility of cache conflict misses entirely. We use a code generation engine, either a Just-In-Time (JIT) assembler or a collection of statically-tuned routines, to generate code for `BDC` or `MBDC` tailored to the needs of each convolution workload and architecture. This paper makes the following contributions:

- Demonstrates that state-of-the-art approaches to run convolution workloads on SIMD architectures suffer from poor performance in the context of long SIMD architectures.
- Proposes the *Bounded Direct Convolution* (`BDC`) algorithm, which judiciously limits the amount of computation exposed to the hardware in order to reduce data cache conflict misses without stalling functional units.
- Proposes the *Multi-Block Direct Convolution* (`MBDC`) algorithm, which improves the memory access pattern by redefining the tensor memory layout.

The final publication is available at ACM via http://dx.doi.org/10.1145/3572848.3577435

- Evaluates the performance of BDC, MBDC and the state-of-the-art approach on the SX-Aurora [30] processor, an architecture featuring CPUs with 16,384-bit SIMD registers. Our analysis includes the convolution algorithms in vednn [22], a highly-tuned vendor-proprietary library. Our results indicate that BDC and MBDC achieve respective speed-ups of 1.44× and 1.28× against the state-of-the-art, and of 1.83× and 1.63× with respect to vednn on ResNet-101 workloads.

2 The Convolution Primitive

The 2-dimensional convolution, widely used on computer vision models [11, 25, 28], is a function over three rank-4 tensor operands: two tensors describe the source (S) and destination (D) activations, and a final tensor (W) represents the filter weights. Activation tensors (S and D) dimensions represent: the minibatch size (N), the number of input or output feature maps (IC and OC), the activations height (IH and OH) and width (IW and OW). The weights tensor contains both input (IC) and output (OC) feature maps and the perception field height (KH) and width (KW) dimensions. Throughout this paper, we describe these tensor shapes using tuples, following the state-of-the-art convention [1, 15]. We represent S and D as (N, IC, IH, IW) and (N, OC, OH, OW) respectively, and the W tensor as (OC, IC, KH, KW).

The W tensor slides over the S tensor during the forward convolution, creating a series of 3-dimensional intersections of shape (IC, KH, KW) over their shared dimensions. Each intersection generates one output element computed by the summation of element-wise multiplications within the intersected area. This routine sweeps the S tensor spatial domain and then repeats across different OC indices and images within the minibatch, ultimately composing the D tensor data. Additional convolution arguments, stride (C_{str}) and padding (C_{pad}), govern how to move the W tensor across the S spatial space by either skipping activations or considering zero-padding at the edges of the spatial domain.

Training a DNN model requires two other passes: backward data and backward weights. The W tensor slides across the D tensor during the backward data direction and computes the partial derivatives concerning the S tensor operands. The backward data outputs a new tensor S_{diff}, with the same shape as S, and propagates it back to the previous model layer where it is called D_{diff}, the output tensor gradients. During the backward weights pass, the algorithm convolves the D_{diff} and S tensors to compute the weight gradients, or W_{diff}, that are applied to the W tensor to adjust the weights. Although the different directions alter the tensor roles, the changes do not profoundly affect the way computations are carried out. Indeed, the algorithm shares the same structure and optimizations across all directions, with a few exceptions involving large filters and non-unit strides [10], which are not standard on computer vision workloads.

Algorithm 1 The Naive Convolution

Input: S, W, C_{str}, C_{pad}
Output: D

```
1:  for n = 0, N do
2:   for oc = 0, OC do
3:    for ic = 0, IC do
4:     for oh = 0, OH do
5:      for ow = 0, OW do
6:       for kh = 0, KH do
7:        ih = oh * C_str + kh - C_pad
8:        for kw = 0, KW do
9:         iw = ow * C_str + kw - C_pad
10:         D[n,oc,oh,ow] += S[n,ic,ih,iw] * W[oc,ic,kh,kw]
```

Algorithm 1 shows a simple 2-dimensional forward convolution, highlighting seven nested loops that wrap a single Fused Multiply and Accumulate (FMA) operation. The loops and the underlying tensor memory layout may change, but the output remains the same, provided the computation of memory offsets is kept consistent. The backward weights and backward data directions use a similar loop structure and differ in which tensor is the output, *i.e.,* S during the backward data pass and W during the backward weights.

2.1 The High-Performance Convolution

Each convolution output element is independent, fitting the SIMD parallel execution model. The forward, backward data, and backward weights directions contain $N \cdot OC \cdot OH \cdot OW$, $N \cdot IC \cdot IH \cdot IW$ and $OC \cdot IC \cdot KH \cdot KW$ independent output elements respectively, defining the convolution as highly parallel. Indeed, efficient convolution algorithms rely primarily on correctly mapping parallel resources to convolution loops and strategies to increase data reuse at the various memory levels [4, 27]. The convolution can be transformed into GEMM [2, 5, 29] and FFT [19] problems and solved with highly-optimized math kernel libraries. Other techniques to accelerate convolutions involve the use of quantization and reduced precision data-types [3, 17], which reduce the tensor memory footprint at the cost of model accuracy.

2.2 The SIMD Direct Convolution Algorithm

The direct algorithm formulation for convolutions accelerates computations by using optimizations like cache blocking and loop reordering. Previous works propose highly efficient convolution algorithms for CPUs equipped with SIMD ISAs [8, 10, 31], reporting up to 90% of the theoretical peak performance on AVX512 processors for some ResNet [11] layers. These proposals formulate the *SIMD direct convolution algorithm*, avoiding the memory overhead of *im2col* transformations required by FFT- and GEMM-based solutions [5, 19]. Despite minor differences, all SIMD direct convolution variants apply a standard set of optimizations driven by analytical architecture models [12, 18]. These models guide the

Table 1. Architecture analytical model applied to SIMD cpus. Assuming 32-bit float datatypes.

Architecture	N_{vlen}	N_{fma}	L_{fma}	$\mathcal{E}$
Intel Skylake	16	2	5	160
NEC SX-Aurora	512	3	8	12288

definition of optimization variables like the register blocking factor. Modern numeric kernel libraries such as oneDNN [15] and VEDNN [22] incorporate implementations of the SIMD direct convolution algorithm, supporting software tools like Tensorflow [1], Caffe [16], and PyTorch [23]. We later show, in Section 5, that unnecessary associations of the maximum SIMD length to optimization variables undermine the performance of this technique on architectures featuring CPUs with long SIMD registers.

3 Architecture Analytical Model

We motivate our algorithmic design choices using the analytical SIMD machine model employed to guide optimization efforts on high-performance GEMM [18] and convolution [31] kernels. This model considers the following hardware features of SIMD architectures:

SIMD Registers: SIMD instructions consume and produce data from/to SIMD registers, each with a SIMD length of N_{vlen} elements. A total of N_{vregs} logical vector registers are addressable by SIMD instructions.

FMA Units: These units support SIMD Fused Multiply-Add (FMA) instructions with a latency of L_{fma} cycles. The hardware schedules SIMD FMA instructions to the N_{fma} independent FMA processing units. It is possible to pipeline each FMA unit fully by issuing instructions every cycle.

SIMD architectures achieve the theoretical peak performance when fully subscribing all N_{fma} vector FMA units. Sustaining this performance requires exposing several independent computations ($\mathcal{E}$) to the architecture. Formula 1 expresses the relationship between the model architecture variables to avoid register dependency stalls and fully benefit from the CPUs' deep SIMD pipelines.

$$\mathcal{E} \geq N_{vlen} \cdot N_{fma} \cdot L_{fma} \tag{1}$$

Table 1 displays the values of the N_{vlen}, N_{fma}, L_{fma} and $\mathcal{E}$ parameters for the SX-Aurora [30] and the Intel Skylake [9] architectures. As this Table indicates, long SIMD architectures, like the SX-Aurora, require parallel algorithms to expose a significantly larger number of independent FMA computations $\mathcal{E}$ compared to 512-bit architectures like the Intel Skylake. This difference originates not only from the larger vector length (N_{vlen}), but also from the increased number of FMA units (N_{fma}) and FMA latency (L_{fma}).

4 The Direct Convolution on Long SIMD Architectures

This section describes how to apply state-of-the-art techniques [8, 10, 31] to formulate the direct convolution on long SIMD architectures. We use this formulation in Section 5 to reveal shortcomings that manifest on long SIMD architectures. Our proposals, presented in Section 6, improve this formulation by adapting certain algorithmic aspects to longer SIMD lengths. Unless otherwise stated, we discuss the convolution concerning the forward data direction in this Section, providing due comments whenever required. Section 8 evaluates all directions demonstrating that our contributions benefit all of them.

4.1 Using SIMD Instructions and Register Blocking

Using SIMD instructions allows the CPU to exploit data-level parallelism, which corresponds to the N_{vlen} contribution of Formula 1. The register blocking optimization subscribes all FMA units with independent computations to prevent CPU stalls due to data dependencies between instructions. Variables L_{fma} and N_{fma} of Formula 1 account for the number and the latency of SIMD FMA units. We use SIMD instructions to run the computations across the feature map dimensions (OC), following current practice [8, 10, 31]. We apply register blocking to the output tensor spatial dimensions (OW and OH) by factors RB_w and RB_h. Since $\mathcal{E} = RB_w \cdot RB_h \cdot N_{vlen}$ and according to Formula 1, the blocking factors must fulfill Formula 2 under the constraint $RB_w \cdot RB_h < N_{vregs}$, creating $RB_w \cdot RB_h$ independent accumulation chains of N_{vlen} elements.

$$RB_w \cdot RB_h \geq N_{fma} \cdot L_{fma} \tag{2}$$

For the backward data propagation, the output tensor is S_{diff}, and we use SIMD instructions in the loop over the IC dimension and apply register blocking to the IW and IH dimensions. During the backward weights pass, the output tensor is W_{diff}, and we select the largest feature map direction (IC or OC) to vectorize, as both are available. We apply register blocking to the smaller feature map dimension, using a single register blocking factor (RB_c).

4.2 Tensor Memory Layout

State-of-the-art convolution implementations [15] use a tensor memory layout that enables the movement of partial sums to/from registers via unit-stride SIMD load/store instructions. Figure 1 depicts this memory layout, applied to activation tensors by using black arrows to indicate unit stride accesses. The memory layout is constructed by blocking the IC and OC feature map dimensions by the factors IC_b and OC_b, both set to N_{vlen}, and pulling the blocks to the innermost dimension in memory. The following tuples represent the memory layouts we consider for tensors S, D, and W, from outer- to inner-most dimension order: (N, IC/IC_b,

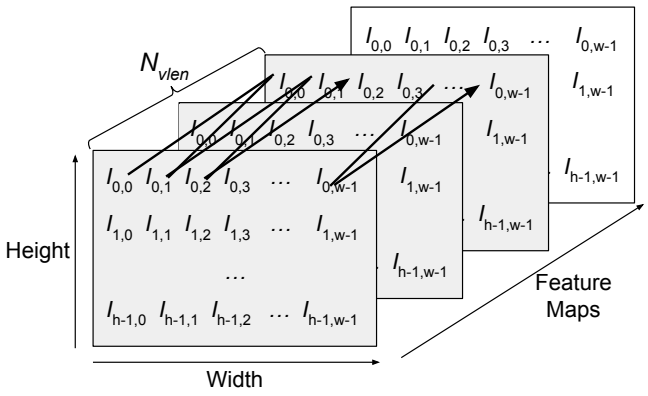

Figure 1. The high-performance activation tensor memory layout used in the SIMD direct convolution. The arrows denote the data elements in contiguous memory positions. Notice that the feature map block interleaves the data for adjacent spatial points.

OH, OW, IC_b), (N, OC/OC_b, OH, OW, OC_b) and (OC/OC_b, IC/IC_b, KH, KW, IC_b, OC_b).

Unprecedented conditions like $OC < N_{vlen} < IC$ or $IC < N_{vlen} < OC$ occur on long SIMD architectures. Typical SIMD architectures typically handle these scenarios using strip mining or zero-padding the smallest operand. However, ISAs supporting long SIMD instructions [6, 26, 30] can handle such cases by dynamically reducing the SIMD length N_{vlen}. To exploit this feature, we use independent and dynamic blocking factors $IC_b = min(IC, N_{vlen})$, and $OC_b = min(OC, N_{vlen})$, to S and D tensors, avoiding padding and strip mining entirely.

4.3 Loop Order and Multithreading

We apply state-of-the-art optimizations [8, 10, 31] to prioritize the cache reuse of the W tensor. We place the loops over the weights spatial dimensions and input feature map block (*i.e.*, KH, KW, and IC_b) inside the loop nest surrounding the vector instructions. The loops over the blocked output spatial domain (*i.e.*, OW/RB_w and OH/RB_h) are next, followed by the blocked feature map loops (*i.e.*, IC/IC_b and OC/OC_b). The last loop iterates over the minibatch elements and can be executed in parallel, where each compute unit processes a subset of images while sharing the weights tensor data from the Last Level Cache (LLC).

We consider different parallelization strategies from the state-of-the-art to parallelize the computations during the backward weights direction [8, 10]. We execute the loop over the smallest feature map dimension in parallel during the backward weights propagation. For instance, when $OC > IC$, the OC loop is vectorized, and the IC loop receives the register blocking optimization and executes in parallel.

Algorithm 2 The direct convolution algorithm for long SIMD architectures

Input: Source Activation Tensor (S), Weights Tensor (W)
Output: Destination Activation Tensor (D)
Architectural variables: N_{vlen}, N_{vregs}

1: $OC_b = min(OC, N_{vlen})$
2: $IC_b = min(IC, N_{vlen})$
3: $RB_w = min(N_{vregs} - 1, OW)$
4: $RB_h = max((N_{vregs} - (1 + RB_w))/OH, 1)$
5: $vl = min(OC, N_{vlen})$ ▷ SIMD Length
6: **for** $n = 0, N$ **do**
7: **for** $oc = 0, OC/OC_b$ **do**
8: **for** $ic = 0, IC/IC_b$ **do**
9: **for** $oh = 0, OH/RB_h$ **do**
10: **for** $ow = 0, OW/RB_w$ **do**
11: **for** $(h, w) = (0 : RB_h, 0 : RB_w)$ **do** ▷ Unrolled
12: $vo_{h,w} = vload(D[n][oc][oh + h][ow + w][0], vl)$
13: **for** $(kh, kw, ic_i) = (0 : KH, 0 : KW, 0 : IC_b)$ **do**
14: $vw = vload(W[oc][ic][kh][kw][ic_i]][0], vl)$
15: **for** $(h, w) = (0 : RB_h, 0 : RB_w)$ **do** ▷ Unrolled
16: $vi_{h,w} = S[n][ic][oh + kh + h][ow + kw + w][ic_i]$
17: $vo_{h,w} = vfma(vo_{h,w}, vi_{h,w}, vw, vl)$
18: **for** $(h, w) = (0 : RB_h, 0 : RB_w)$ **do** ▷ Unrolled
19: $vstore(vo_{h,w}, D[n][oc][oh + h][ow + w][0], vl)$

4.4 The Direct Convolution Algorithm for Long SIMD Architectures

We show the algorithm to compute the forward direct convolution on long SIMD architectures in Algorithm 2. We set the dynamic cache blocking factors in Lines 1-2. Lines 3-4 drive register blocking as Section 4.1 describes. Next, line 5 sets the working SIMD length for all SIMD instructions. The following code lines specify (i) the scheduling loops, which appear in Lines 6-10; and (ii) the micro-kernel, encompassing Lines 11-19. The scheduling loops organize the order of sub-convolutions, while the micro-kernel refers to the compute-intensive region. The latter segment features fully-unrolled loops in Lines 11, 15, and 18, which are generally produced by Just-In-Time (JIT) assemblers [8, 10], or an ensemble of static compilation techniques and hand-tuned code segments [14, 32].

The innermost convolution loops (Line 13) contain a vector load to the shared W operand (Line 14) and a series of scalar loads (Line 16) and SIMD FMA instructions (Line 17). We use a SIMD FMA with one scalar multiplicand ($vi_{h,w}$) in Line 17, which is supported by all major emerging vector ISAs like RISC-V V [6], ARM SVE [26], and NEC SX-Aurora [30]. In this FMA operation, the CPU implicitly broadcasts the scalar element to form a temporary vector operand without using a dedicated vector register.

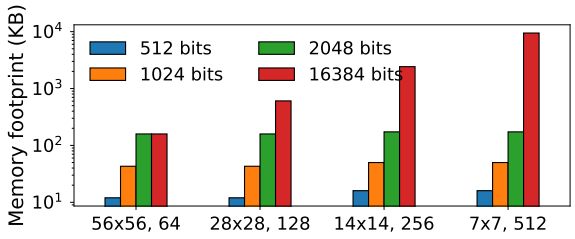

Figure 2. Memory footprint of the SIMD direct convolution algorithm micro-kernel region considering 3x3 convolutions on architectures with different vector lengths.

5 Shortcomings of the Direct Convolution

5.1 Large Micro-Kernel Memory Footprint

The memory footprint of the convolution micro-kernel is driven by the D, W, and S sub-tensors. The W sub-tensor, accessed in Line 14 of Algorithm 2, is reused multiple times since it is applied to all spatial output points (*i.e.*, loops in Lines 9 and 10). Therefore, it should be kept in the cache hierarchy until the next iteration of the loop at Line 8. Constraining the memory footprint of the convolution micro-kernel region to the Last Level Cache (LLC) size is enough to fulfill this criterion and avoid the proliferation of unnecessary off-chip memory accesses in Line 14.

The micro-kernel region loads $OC_b \cdot IC_b \cdot KH \cdot KW$ weight elements, $IC_b \cdot min(RB_h + KH, IH) \cdot min(RB_w + KW, IW)$ source tensor elements, and moves $OC_b \cdot RB_h \cdot RB_w$ elements to/from the destination tensor. Since the parameters IC_b and OC_b are both associated with N_{vlen}, as we explain in Section 4.2, an increase in the architectural vector length brings a quadratic growth on the weights sub-tensor size. This issue mainly affects convolutions with large filters, where the number of iterations in the KH and KW loops further intensifies the problem. Figure 2 depicts the memory footprint of 3x3 convolutions found in VGG [25] and ResNet [11] models, where the memory footprints can reach up to 9 megabytes on architectures with 16384-bit vectors. The x-axis describes convolution layers in terms of the activation spatial size and the number of feature map channels.

5.2 Memory Access Pattern Displays High Cache Miss Rates

The activation memory layout, described in Section 4.2, and register block optimizations define a scalar memory access pattern that leads to frequent L1 data cache conflict misses on CPUs supporting long SIMD instructions. Figure 3 illustrates this memory access pattern across the S tensor during the forward pass. Black arrows indicate the dynamic order of accesses with a stride of N_{vlen}, matching the feature map blocking factor. This pattern also manifests in the D tensor during the backward data and weights directions.

When using long SIMD instructions, a series of memory accesses with stride equal to the SIMD length (N_{vlen}), visit only a fraction of the cache sets before the accumulated offset loops around the cache addressing space. The relationship between the N_{vlen} and the cache line size (N_{cline}), typically following $N_{vlen} = 2^n \cdot N_{cline}$ and $n \geq 0$, results in visits to the same sequence of cache sets, independently of the cache associativity. If a series of such accesses are large enough to loop around the total cache size, the first cache lines might be evicted before they can be reused when accessed on the following convolution loops.

Formula 3 specifies an inequality that indicates when cache conflict misses appear in the direct convolution. The formula depends on (i) the L1 cache size ($L1_{size}$); (ii) the distance between spatial points, constituting the convolution stride parameter (C_{str}) and the activation tensor feature map blocking factor (A_b); and (iii) the number of spatial points visited before reusing previously accessed cache lines (defined by the register blocking optimization as $RB_h \cdot RB_w$). The activation tensor feature map blocking factor A_b is either IC_b or OC_b depending on which tensor the algorithm accesses with scalar instructions.

$$L1_{size} < A_b \cdot RB_h \cdot RB_w \cdot C_{str} \tag{3}$$

An architecture featuring 32KB L1 caches and 16384-bit wide vector registers like SX-Aurora [30] requires a combined register blocking factor ($RB_h \cdot RB_w$) of 24 to avoid data dependency stalls, according to Formula 2 and Table 1 data. Assuming a convolution where C_{str} is 1, and A_b is 16384 (N_{vlen}), it is impossible to meet the inequalities of Formulas 2 and 3 simultaneously due to the unsolvable inequality ($16 > RB_h \cdot RB_w$ and $24 < RB_h \cdot RB_w$). Conflict misses severely undermine the effectiveness of the optimizations described in Sections 4.1 and 4.2, making it impossible to saturate the computation, either because the register file is underused, generating pipeline dependency stalls, or because the SIMD lanes starve waiting on data dependencies from L1. Our evaluation on Section 8 validates this phenomenon in SX-Aurora.

6 Efficient Direct Convolution Using Long SIMD Instructions

This Section describes our algorithmic solutions to efficiently run the direct convolution operation using long SIMD instructions. Section 6.1 describes a method to dynamically reduce the memory footprint of the convolution micro-kernel and workaround the issue discussed in Section 5.1. Sections 6.2 and 6.3 introduce two proposals to reduce the number of cache conflict misses by judiciously limiting the amount of computation exposed to the SIMD units, and adjustments to the activations tensor memory layout, respectively.

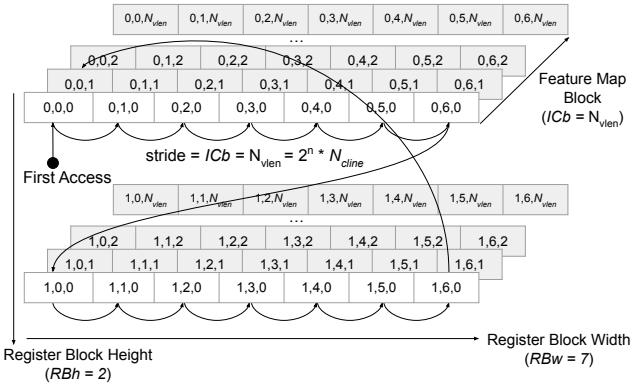

Figure 3. Example of the SIMD direct convolution memory access pattern on the S tensor during the forward pass. This access pattern stresses a small number of cache sets when using long SIMD instructions, which results in conflict misses.

6.1 Dynamically Adapting the Micro-Kernel Memory Footprint

We propose a method to automatically adapt the convolution microkernel's memory footprint to many SIMD architectures, including those with long vectors. Our method combines two strategies: *loop reordering* and *loop resizing*, implemented into a dynamic auto-tuning algorithm.

Loop Reordering: the simplest way to reduce the memory footprint of convolutions is to move the loops over the weight spatial domain (KH and KW) from the micro-kernel to the sub-convolution scheduling region. However, this technique does not apply to 1x1 convolutions and decreases the S tensor L1 cache reuse on regular convolutions, requiring additional passes over the S tensor activations.

Loop Resizing: another strategy is to reduce the number of iterations of the micro-kernel loop over the input feature map block (last loop iterator in Line 13 of Algorithm 2). This loop has its number of iterations unnecessarily tied to the vector length through the IC_b variable. Reducing the number of iterations of this loop from IC_b to N_{cline} reduces the memory footprint by the same rate while still accessing all data brought to the L1 cache. We reflect this change on the weights tensor memory format as well, where we decouple the W tensor IC dimension blocking factor from IC_b and associate it to N_{cline}, altering the W tensor tuple representation from $(OC/OC_b, IC/IC_b, KH, KW, IC_b, OC_b)$ to $(OC/OC_b, IC/N_{cline}, KH, KW, N_{cline}, OC_b)$.

We apply the aforementioned strategies via an algorithm that prioritizes *loop resizing* over *loop reordering*. The algorithm finds the largest W sub-tensor size that fits into the cache, using the tensor sizes and the following architectural parameters as input: (i) the vector length (N_{vlen}), (ii) the cache line size (N_{cline}), and (iii) the LLC storage capacity (LLC_{size}). The algorithm generates the blocking factors kh_i, kw_i, and ic_i for the three loops at Line 13 in Algorithm 2,

Algorithm 3 The Auto-tuning Algorithm

Input: $IH, IW, OH, OW, KH, KW, IC, OC$
Architectural variables: $N_{vlen}, N_{cline}, LLC_{size}$
Output: kh_i, kw_i, ic_i

1: $kh_i = KH$
2: $kw_i = KW$
3: $ic_i = IC$
4: $OC_b = min(OC, N_{vlen})$
5: $nih = min(IH, RB_h + kh_i - 1)$
6: $niw = min(IW, RB_w + kw_i - 1)$
7: $W_{mem} = OC_b \cdot ic_i \cdot kh_i \cdot kw_i$
8: $D_{mem} = OC_b \cdot RB_h \cdot RB_w$
9: $S_{mem} = ic_i \cdot nih \cdot niw$
10: **while** $W_{mem} + O_{mem} + I_{mem} > LLC_{size}$ **do**
11: **if** $ic_i > 2 * N_{cline}$ **then**
12: $ic_i = ic_i/2$
13: **else if** $kh_i > 1$ **then**
14: $kh_i = 1$
15: $ic_i = IC$
16: $nih = min(IH, RB_h)$
17: **else if** $kw_i > 1$ **then**
18: $kw_i = 1$
19: $ic_i = IC$
20: $niw = min(IW, RB_w)$
21: $W_{mem} = OC_b \cdot ic_i \cdot kh_i \cdot kw_i$
22: $D_{mem} = OC_b \cdot RB_h \cdot RB_w$
23: $S_{mem} = ic_i \cdot nih \cdot niw$
return (kh_i, kw_i, ic_i)

deciding their size at runtime based on the convolution problem and the target architecture.

Algorithm 3 displays this procedure. The method initially assumes the highest possible loop values over the weights sub-tensor in the convolution micro-kernel region (Lines 1-3). The memory footprints for the W, S, and D tensors are computed first at Lines 4-9 and later compared against the LLC size at Line 10. The routine applies the *loop resizing* strategy in Lines 11-12, reducing the IC loop size unless this reduction would cause it to be smaller than the cache line size. The *loop reorder* fallback strategy happens at Lines 13-16 and 17-20 by setting the KH and KW loop iteration counts to 1. After these arrangements, the routine recomputes the memory footprint of the micro-kernel (Lines 21-23) and begins another iteration at Line 10.

Algorithm 3 fits into multicore systems with shared caches by multiplying the activation tensors' memory footprint by the number of threads at Lines 8, 9, 22, and 23.

6.2 The Bounded Direct Convolution

The *Bounded Direct Convolution* (BDC) algorithm mitigates cache conflict misses with a less aggressive lower limit for the RB_w and RB_h optimization variables and employs the

method we describe in Section 6.1 to adapt the micro-kernel memory footprint to the cache hierarchy storage.

We relax the inequality of Formula 2 based on the observation that architectures with CPUs supporting long SIMD instructions are load/store machines, thus requiring scalar code for loading the FMA scalar operand and updating the memory offsets for future scalar loads. These scalar instructions are placed between SIMD FMA instructions and create some distance between dependent SIMD FMAs. We also use Formula 3 to set an upper limit to the register blocking factors, preventing the memory access pattern from generating cache conflict misses. We combine these two approaches to formulate a value range for the RB_h and RB_w optimization variables, described in Formula 4, that avoids data dependency stalls and cache conflict misses.

$$N_{fma} \cdot L_{fma}/B_{seq} \leq RB_h \cdot RB_w < L1_{size}/C_{str} \quad (4)$$

Formula 4 introduces a new variable B_{seq}, which is the minimum distance in terms of instructions between subsequent SIMD FMAs within the direct convolution micro-kernel region. Empirical observations indicate that compilers targeting the RISC-V "V" [6] and SX-Aurora [30] ISAs use two scalar instructions in between vector FMA instructions: (i) a scalar load to bring the value to the register and (ii) an addition to update the S tensor memory pointer. Therefore, in this case, the B_{seq} distance is three, as a subsequent SIMD FMA is three instructions ahead. In SX-Aurora, setting B_{seq} to three allows the register blocking factors to be as low as 8, in contrast to the previous minimum value of 24.

6.3 The Multi-Block Direct Convolution

The *Multi-Block Direct Convolution* (MBDC) algorithm redefines the blocking factor that characterizes the tensor memory layout, improving the memory access pattern of the convolution micro-kernel and reducing cache conflict misses.

The S tensor memory access pattern, described in Section 5.2, does not cause conflict misses when $N_{vlen} = N_{cline}$ because both IC_b and OC_b also become equal to N_{cline}, *i.e.,* the stride between accesses to the S tensor in this scenario is just one cache line, meaning that all cache sets are stressed equally. Based on this observation, we propose to block S and D tensors by N_{cline}, instead of N_{vlen}. This change disassociates the architectural vector length from the optimization variables IC_b and OC_b to improve the S tensor scalar access pattern locality.

Our proposal to reduce OC_b and IC_b, from N_{vlen} to N_{cline}, divides the feature map dimensions into smaller blocks scattered through the activation tensor memory layout. With the innermost dimension being smaller than N_{vlen}, it is impractical to move an entire vector register worth of feature map data regarding one spatial point, using unit-stride vector load/stores (*i.e.,* Lines 12 and 19 in Algorithm 2). Therefore, we support our proposed memory layout using coarse granularity gather/scatter instructions when accessing the activation tensors with SIMD instructions on Lines 12 and 19 in Algorithm 2.

Algorithm 4 MBDC (C_{str}=1, C_{pad}=0)

Input: Source Activations S, Weights Tensor W
Output: Output Activations O
Architectural variables: $N_{vlen}, N_{vreg}, N_{cline}$
Optimization variables: kh_i, kw_i, ic_i

1: $OC_b = cline$
2: $IC_b = cline$
3: $RB_w = min(N_{vregs}, OW)$
4: $RB_h = max((N_{vregs} - RB_w)/OH, 1)$
5: $vid = make\_indices(OH, OW, N_{cline})$
6: $vl = min(OC, N_{vlen})$ ▷ Vector Length
7: **for** $n = 0, N$ **do**
8: **for** $oc = 0, OC/OC_b$ **do**
9: **for** $ic = 0, IC/ic_i$ **do**
10: **for** $oh = 0, OH/RB_h$ **do**
11: **for** $ow = 0, OW/RB_w$ **do**
12: **for** $khb = 0, KH/kh_i$ **do**
13: **for** $kwb = 0, KW/kw_i$ **do**
14: **for** $(h, w) = (0 : RB_h, 0 : RB_w)$ **do** ▷ Unrolled
15: $vo_{h,w} = vgather(D[n][oc][oh + h][ow + w][0], vid, vl)$
16: **for** $(kh, kw, i) = (0 : kh_i, 0 : kw_i, 0 : ic_i)$ **do**
17: $vw = vload(W[oc][i/IC_b][kh][kw][i\%IC_b][0], vl)$
18: **for** $(h, w) = (0 : RB_h, 0 : RB_w)$ **do** ▷ Unrolled
19: $vi_{h,w} = S[n][icb][ohb+kh+h][owb+kw+w][i]$
20: $vo_{h,w} = vfma(vo_{h,w}, vi_{h,w}, vw, vl)$
21: **for** $(h, w) = (0 : RB_h, 0 : RB_w)$ **do** ▷ Unrolled
22: $vscatter(vo_{h,w}, D[n][oc][oh + h][ow + w][0], vid, vl)$

Algorithm 4 depicts MBDC. The values of kh_i, kw_i, and ic_i are determined by the dynamic method we describe in Section 6.1. The gather/scatter operations in Lines 15 and 22 replace the vector load/stores involving the D tensor of Algorithm 2. Indices are computed at Line 5 and defined in Equation 5 for 32-bit elements. The feature map blocks with size N_{cline} form a sub-tensor of size $OH \cdot OW \cdot N_{cline}$, leading to a distance between blocks of $i/(N_{cline}/4) \cdot OH \cdot OW$ bytes for 32-bit elements. We make note of the number of iterations for the loops in Lines 9 and 16 which exploit the micro-kernel compute granularity dynamically determined by the auto-tuner.

$$v[i] = mem[(i/(N_{cline}/4) \cdot OH \cdot OW + i\%(N_{cline}/4)) \cdot 4] \quad (5)$$

Memory to register gather operations and register to memory scatters typically incur a significant performance cost. Emerging SIMD ISAs reduce this cost by supporting different kinds of gather/scatters operations combining contiguous

and non-contiguous memory accesses. For example, RISC-V V1.0 [6] supports gather/scatters of element up to 128 bits each, which can be seen as blocks of four contiguous 32-bit words. ARM SVE [26] supports combined gather/scatter operations that reduce the number of individual memory requests by returning two consecutive elements for each index. SX-Aurora [30] supports 2-dimensional vector load/stores, which emulate vector gather/scatters at the granularity of an entire 128-byte cache line. We exploit coarse-grain gather/scatter operations to load/store non-contiguous blocks of N_{cline} contiguous elements to/from SIMD registers and reduce the number of memory requests issued by the CPU.

6.4 Summary

Table 2 depicts a summary of this section and our design contributions, displaying the original Direct Convolution targeting 512-bits SIMD instructions, its adaptation to long SIMD architectures (Section 4), as well as our two new algorithmic contributions: (i) BDC and (ii) MBDC. The *Activation block* column depicts the values each algorithm uses for the IC_b and OC_b variables, which defines the activation tensor memory layout. The *Weight block* column depicts the blocking factors for the weights tensor, which is also responsible for defining the weights tensor memory layout. *Schedule grain* defines the minimum number of iterations for the loop over the *IC* block within the convolution micro-kernel region. The *Register block* column defines the value range for the combined register blocking factors for the different algorithms. This table highlights our proposal's main points: (i) the new register block size constraints on BDC, and (ii) the redefinition of the activation memory layout on MBDC.

6.5 Implementation

We articulate our contributions following the oneDNN [15] library design, where the instantiating of kernels follows a two-step process involving: (i) the initial problem declaration and (ii) the kernel execution. Algorithm 3 receives architectural variables and convolution arguments during the problem declaration, including the tensor shapes, and computes all optimization variables. This information is codified in a data structure and forwarded to a code generator engine, either a JIT assembler or a collection of statically-tuned functions. The code generator returns a function pointer corresponding to the convolution micro-kernel region. Later, the convolution scheduling loops call the micro-kernel function pointer during the kernel execution.

7 Experimental Methodology

Convolution Workloads: We consider the convolution layers used on ResNet [11] models. Table 3 describes these layers in terms of the convolution parameters detailed in Section 2.

Hardware Platform: We run our experimental campaign on the NEC SX-Aurora system [30]. The SX-Aurora processor features SIMD registers with a capacity of 16,384 bits. The SIMD processing unit employs 8-cycles deep pipelines and operates on 64 32-bit elements per cycle. There are three vector unit ports dedicated to computing vector fused multiply-add instructions yielding a maximum throughput of $64 \cdot 3 \cdot 2 = 384$ 32-bit floating-point operations per core each cycle. The SX-Aurora system used in this work houses eight cores running at a frequency of 1.6GHz, configuring a theoretical peak performance of 614 GFLOP/s per core and 4912 GFLOP/s when using all eight vector cores. Each core features two 32KB 2-way set associative caches for instructions and data, respectively, and a 256KB L2 cache 4-way set associative covering both instructions and data. A 2-dimensional mesh Network on Chip (NoC) connects the cores to a shared 16 MB LLC. The LLC 128-byte cache lines are interleaved into 16 memory banks for parallel access when serving unit-stride vector memory instructions and handles both scalar and SIMD requests. The processor features a 48 GB on-chip HBM2 RAM with a total bandwidth of 1.35 TB/s.

Convolution Algorithms: We consider the following approaches to run the convolution workloads:

1. *Direct Convolution for Long SIMD architectures* (DC): This approach applies state-of-the-art optimizations [8, 10, 31] tailored to long SIMD architectures. We describe DC in Section 4.
2. *NEC Proprietary Library* (vednn): This approach is based on vednn [22], a high-performance library featuring optimized convolution kernels for the SX-Aurora, originally created to support TensorFlow-VE [21]. We always use the best performing algorithm in vednn for a given problem, which may be a direct convolution or implicit/explicit GEMM convolution kernels.
3. *Bounded Convolution* (BDC): We describe this algorithm in Section 6.2.
4. *Multi-Block Direct Convolution* (MBDC): We describe this algorithm in Section 6.3.

Software Tools: We generate the DC, BDC, and MBDC convolution kernels via explicit vectorization and compiler vector intrinsic functions of the NEC LLVM compiler v1.16 [20]. The convolution kernels extend oneDNN [15] v1.7.4 taking the form of custom convolution primitives targeting the SX-Aurora VE [30]. We compile vednn using the NEC NCC v3.3.1 proprietary compiler that features automatic code transformations for performance, including loop unrolling and automatic vectorization. We use the OpenMP [7] runtime system to run our multithreaded workloads and fully subscribe to the eight available cores.

8 Evaluation

Figure 4 shows the vednn, DC, BDC and MBDC convolution algorithms performance (y-axis), in GFLOP/s, when subjected

Table 2. Summary of Convolution Algorithms

Algorithm	Activation block	Weight block (*IC*,*OC*)	Schedule grain	Register block (min)	Register block (max)
Direct Convolution [8, 10, 31]	N_{vlen}	(N_{vlen}, N_{vlen})	N_{vlen}	$L_{fma}N_{fma}$	N_{vregs}
Direct Conv. Long SIMD (Sec. 4)	$min(N_{vlen}, C)$	$(N_{vlen}, min(N_{vlen}, OC))$	N_{vlen}	$L_{fma}N_{fma}$	N_{vregs}
BDC (Sec. 6.2)	$min(N_{vlen}, C)$	$(N_{cline}, min(N_{vlen}, OC))$	N_{cline}	$L_{fma}N_{fma}/B_{seq}$	$L1_{size}/(C_{str}N_{vlen})$
MBDC (Sec. 6.3)	N_{cline}	$(N_{cline}, min(N_{vlen}, OC))$	N_{cline}	$L_{fma}N_{fma}$	N_{vregs}

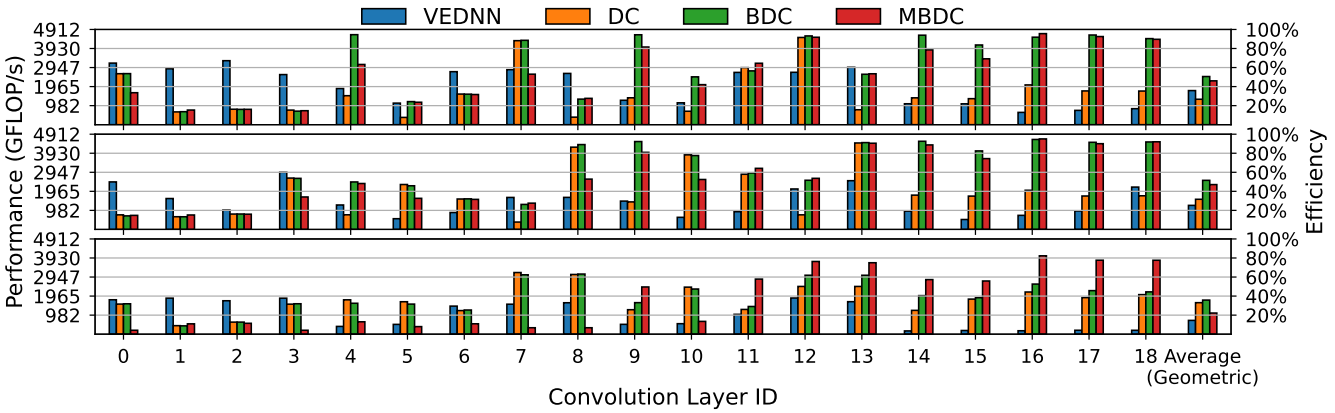

Figure 4. Convolution algorithms performance during the forward data (top), backwards data (middle), and backwards weights (bottom) directions with a minibatch size of 256. These experiments run on the 8 SIMD CPUs described in Section 7.

Table 3. Convolution Layer Parameters

ID	IC	OC	IH/IW	OH/OW	KH/KW	C_{str}	C_{pad}
0	64	256	56	56	1	1	0
1	64	64	56	56	1	1	0
2	64	64	56	56	3	1	1
3	256	64	56	56	1	1	0
4	256	512	56	28	1	2	0
5	256	128	56	28	1	2	0
6	128	128	28	28	3	1	1
7	128	512	28	28	1	1	0
8	512	128	28	28	1	1	0
9	512	1024	28	14	1	2	0
10	512	256	28	14	1	2	0
11	256	256	14	14	3	1	1
12	256	1024	14	14	1	1	0
13	1024	256	14	14	1	1	0
14	1024	2048	14	7	1	2	0
15	1024	512	14	7	1	2	0
16	512	512	7	7	3	1	1
17	512	2048	7	7	1	1	0
18	2048	512	7	7	1	1	0

to the workloads in Table 3 (x-axis). The minibatch size for this figure is 256, following ResNet settings for the ImageNet challenge [11]. We evaluate other minibatch sizes in Section 8.1. Each subplot displays results concerning one propagation direction: (i) Forward Data (top); (ii) Backwards Data (middle); and (iii) Backwards Weights (bottom), referred to as $fwdd$, $bwdd$, and $bwdw$, respectively, throughout this section. Each algorithm executes on all eight cores within the system described in Section 7, and we express their efficiency as the percentage of the system's theoretical peak on the right-hand side y-axis. The rightmost columns aggregate the other columns' performance using a geometric mean.

vednn algorithms rely on vectorizing computations across the spatial domain. Figure 4 showcases vednn achieving greater efficiencies during layers 0-13, up to 65.5% of the peak performance on layer ID 2, compared to the rightmost layers over 7x7 activations. vednn efficiency is also lower on strided convolutions (i.e. $C_{str} > 1$), as shown by layers 4, 5, 9, 10, 14, and 15. Overall, vednn performs best on convolutions over large tensors while accessing spatial data in unit strides.

The DC algorithm performance in SX-Aurora is subject to two factors: (i) how much parallelism is exposed to the SIMD units, and (ii) whether cache conflict misses manifest in the convolution problem. DC vectorizes the computations over one feature map dimension (i.e., OC during $fwdd$, IC for $bwdd$, and IC or OC on $bwdw$) and performs best when this dimension is equal to or larger than N_{vlen} in size. Layers 7 and 12 during $fwdd$ are examples of layers where both sufficient parallelism is exposed and no cache conflicts occur, achieving efficiencies of 88.2% and 91.6% of the theoretical peak. In layers where cache conflicts are predicted by Formula 3 (layers 4,5,8-10,13-18 during $fwdd$ and layers 4,7,9,12,14-18 during $bwdd$), DC obtains average efficiencies of 21.5% and 24.7% during the $fwdd$ and $bwdd$ passes, respectively, as opposed to 35.9% and 39.6% for layers where cache conflicts are not predicted. DC may experience conflict misses on every layer during the $bwdw$ pass but it also vectorizes the computations across the largest dimension, between IC and OC, to improve the SIMD hardware utilization. These traits make the DC average efficiency during $bwdw$ to be 33.0%, higher than the 26.7% and 31% obtained for $fwdd$ and $bwdd$.

The BDC algorithm behaves similarly to DC but reduces the incidence of cache conflict misses by judiciously selecting the register block factor of the convolution inner loops. BDC achieves similar performance regimes to DC on layers where Formula 3 does not predict cache conflict misses. However, BDC offers average speed-ups of 2.95× when compared to DC, during the *fwdd* and *bwdd* passes on layers predicted to experience cache conflict misses. BDC efficiency is higher than DC across all layers, achieving up to 94.4% of the peak performance on layer 16 during the *fwdd*, despite using less aggressive register block sizes. BDC offers a smaller benefit, 8% on average, over DC for the *bwdw* pass because fine-tuning the register block size is not as effective in this direction, as we later show using hardware performance counters.

MBDC is an alternative to BDC that tackles the cache conflict misses issue by adjusting the memory layout. As expected, MBDC and DC have similar performance regimes on layers without cache conflict misses. MBDC offers similar benefits as BDC, yielding average speed-ups of 2.79× over DC during *fwdd* and *bwdd* on layers predicted to experience cache conflict misses. MBDC displays two different performance regimes during *bwdw*: (i) the initial layers, IDs 0-10, with an average efficiency of 9.7%, and (ii) the layers consisting of convolutions over 14x14 and 7x7 activations, IDs 11-18, with average efficiency of 57.7% of the peak performance. MBDC outperforms DC and BDC on the second group of layers, providing average speed-ups of 1.83× and 1.52×, respectively. However, it fails to improve the performance of the first layer group, yielding average slowdowns of 68.0% concerning DC.

MBDC showcases lower performance during the *bwdw* pass because the vector gather/scatter operations are more frequent and some tensor shapes cause accesses to SX-Aurora LLC memory banks to be serialized. The SX-Aurora processor implements low latency memory-to-register operations by interleaving LLC lines on independent memory banks so that unit-stride vector load/stores access consecutive cache lines in parallel. Gather/scatter operations, used in MBDC, can also access the LLC banks in parallel when the offset between the feature map blocks generates a bijective map between cache lines and LLC memory banks, which happens on the late convolutions (i.e., IDs 11-18) over smaller spatial shapes (i.e., 14x14 and 7x7). This scenario changes on layers 0-10, where one LLC bank serves all feature map cache blocks, serializing the data transfer of the different cache blocks requested by the vector gather instruction, inducing high vector load latency.

We use the SX-Aurora hardware performance counters to measure the Misses per Kilo Instruction (MPKI) rates of all algorithms. Our study reveals that BDC and MBDC reduce the MPKI, on average, by 27% and 22%, for the *fwdd* pass, and by 18% and 20% during the *bwdd* pass, respectively, when compared to DC. BDC achieves a similar MPKI as DC considering the *bwdw* direction while MBDC reduces the MPKI by 8% in this setting. The lower MPKI rates originate from the techniques to avoid cache conflict misses and serve as empirical evidence of this problem in long SIMD architectures.

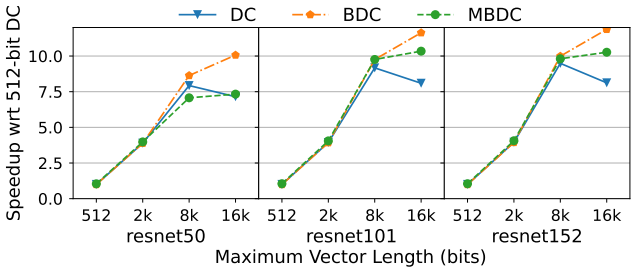

Figure 5. Performance of DC, BDC, and MBDC on different maximum SIMD length settings for ResNet workloads.

8.1 Performance Evaluation on ResNet Models

We evaluate our algorithms under the training workloads of different ResNet models to measure our algorithm's impact at the network level. Indeed, the ResNet models employ the same layers described in Table 3, but each layer appears a different number of times on each model (e.g., layer IDs 11-13 are more frequent in the larger models).

Evaluation with different SIMD lengths: We evaluate our algorithms on a variety of maximum SIMD length settings by limiting the maximum vector length of the SX-Aurora system to 512, 2048, 8196, and 16384 bits. Figure 5 showcases the speed-ups (y-axis) of the three approaches normalized to the state-of-the-art DC convolution with a maximum SIMD length of 512 bits for different vector length settings (x-axis). All three algorithms have similar performance across all models for SIMD lengths smaller than 8192 bits. At a SIMD length of 16384 bits, the BDC algorithm is 1.41×, 1.44×, and 1.46× faster than DC considering the ResNet-50, ResNet-101, and ResNet-152 models respectively. Considering the same scenario, MBDC is 1.28× and 1.26× faster than DC on ResNet-101 and ResNet-152 models. MBDC provides no benefits for ResNet-50 due to the relatively higher frequency of layers that experience serial accesses to SX-Aurora LLC memory banks during the *bwdw* pass.

Performance Evaluation Using Different Minibatch Sizes: Figure 6 depicts the performance (y-axis) of each algorithm considering the execution of the ResNet-101 model, on all training directions, and under different minibatch sizes (x-axis). The BDC algorithm delivers the best performance on all settings, and all three strategies presented in this paper showcase better scaling as problem size increases. vednn is slightly faster than DC on minibatch sizes smaller than 32, and faster than MBDC on minibatch size of 8. However, it fails to scale as problem size increases.

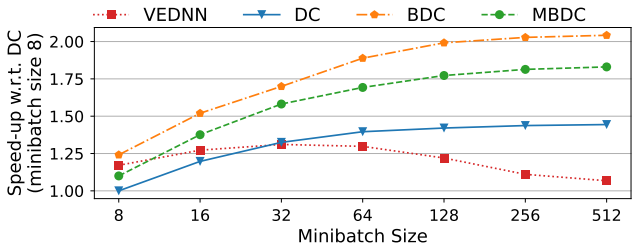

Figure 6. ResNet-101 evaluation on different minibatch sizes.

9 Conclusions

This paper demonstrates that previous approaches to run convolution kernels on SIMD architectures [8, 10, 31] deliver poor performance when applied to processors employing long SIMD instructions. The paper provides theoretical and empirical evidence linking this poor performance primarily to cache conflict misses, originating from the cache blocking (i.e., blocked memory layout) and loop unrolling optimizations. It proposes two approaches, BDC and MBDC, to improve the memory access pattern and mitigate such events. For the ResNet-101 workload, BDC achieves 1.44× and 1.83× speed-ups concerning our DC algorithm, based on the state-of-the-art SIMD direct convolution, and the NEC vednn library. For the same workload, MBDC obtains 1.28× and 1.63× speed-ups compared to DC and vednn.

Acknowledgments

This work receives EuroHPC-JU funding under grant no. 101034126, with support from the Horizon2020 program. Adrià Armejach is a Serra Hunter Fellow and has been partially supported by the Grant IJCI-2017-33945 funded by MCIN/AEI/10.13039/501100011033. Marc Casas has been partially supported by the Grant RYC-2017-23269 funded by MCIN/AEI/10.13039/501100011033 and ESF Investing in your future. This work is supported by the Spanish Ministry of Science and Technology through the PID2019-107255GB project and the Generalitat de Catalunya (contract 2017-SGR-1414).

References

[1] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (2016), pp. 265–283.

[2] Anderson, A., Vasudevan, A., Keane, C., and Gregg, D. Low-memory gemm-based convolution algorithms for deep neural networks. *arXiv preprint arXiv:1709.03395* (2017).

[3] Bhandare, A., Sripathi, V., Karkada, D., Menon, V., Choi, S., Datta, K., and Saletore, V. Efficient 8-bit quantization of transformer neural machine language translation model. *arXiv preprint arXiv:1906.00532* (2019).

[4] Chen, Y.-H., Emer, J., and Sze, V. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. *ACM SIGARCH Computer Architecture News 44*, 3 (2016), 367–379.

[5] Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., and Shelhamer, E. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759* (2014).

[6] Community, R.-V. Risc-v vector extension, 2022. https://github.com/riscv/riscv-v-spec/blob/master/v-spec.adoc.

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

[8] Das, D., Avancha, S., Mudigere, D., Vaidynathan, K., Sridharan, S., Kalamkar, D., Kaul, B., and Dubey, P. Distributed deep learning using synchronous stochastic gradient descent. *arXiv preprint arXiv:1602.06709* (2016).

[9] Doweck, J., Kao, W.-F., Lu, A. K.-y., Mandelblat, J., Rahatekar, A., Rappoport, L., Rotem, E., Yasin, A., and Yoaz, A. Inside 6th-generation intel core: New microarchitecture code-named skylake. *IEEE Micro 37*, 2 (2017), 52–62.

[10] Georganas, E., Avancha, S., Banerjee, K., Kalamkar, D., Henry, G., Pabst, H., and Heinecke, A. Anatomy of high-performance deep learning convolutions on simd architectures. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis* (2018), IEEE, pp. 830–841.

[11] He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2016), pp. 770–778.

[12] Heinecke, A., Henry, G., Hutchinson, M., and Pabst, H. LIBXSMM: Accelerating small matrix multiplications by runtime code generation. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2016), IEEE, pp. 981–991.

[13] Hill, M. D., and Smith, A. J. Evaluating associativity in CPU caches. *IEEE Trans. Computers 38*, 12 (1989), 1612–1630.

[14] Intel. Neon, 2022. https://github.com/NervanaSystems/neon.

[15] Intel. Oneapi deep neural network library, 2022. https://oneapi-src.github.io/oneDNN/.

[16] Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., and Darrell, T. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia* (2014), pp. 675–678.

[17] Kalamkar, D., Mudigere, D., Mellempudi, N., Das, D., Banerjee, K., Avancha, S., Vooturi, D. T., Jammalamadaka, N., Huang, J., Yuen, H., et al. A study of bfloat16 for deep learning training. *arXiv preprint arXiv:1905.12322* (2019).

[18] Low, T. M., Igual, F. D., Smith, T. M., and Quintana-Orti, E. S. Analytical modeling is enough for high-performance blis. *ACM Transactions on Mathematical Software (TOMS) 43*, 2 (2016), 1–18.

[19] Mathieu, M., Henaff, M., and LeCun, Y. Fast training of convolutional networks through ffts. *arXiv preprint arXiv:1312.5851* (2013).

[20] NEC. Nec llvm compiler, 2022. https://github.com/sx-aurora-dev/llvm-project.

[21] NEC. Tensorflow-ve, 2022. https://github.com/sx-aurora-dev/tensorflow.

[22] NEC. Vednn, 2022. https://github.com/sx-aurora-dev/vednn.

[23] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems 32* (2019), 8026–8037.

[24] Sato, M., Ishikawa, Y., Tomita, H., Kodama, Y., Odajima, T., Tsuji, M., Yashiro, H., Aoki, M., Shida, N., Miyoshi, I., Hirai, K., Furuya, A., Asato, A., Morita, K., and Shimizu, T. Co-Design for A64FX Manycore Processor and "Fugaku". SC '20, IEEE Press.

[25] Simonyan, K., and Zisserman, A. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[26] Stephens, N., Biles, S., Boettcher, M., Eapen, J., Eyole, M., Gabrielli, G., Horsnell, M., Magklis, G., Martinez, A., Premillieu, N., et al. The ARM scalable vector extension. *IEEE micro 37*, 2 (2017), 26–39.

[27] Sze, V., Chen, Y.-H., Yang, T.-J., and Emer, J. S. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE 105*, 12 (2017), 2295–2329.

[28] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (2015), pp. 1–9.

[29] Vasudevan, A., Anderson, A., and Gregg, D. Parallel multi channel convolution using general matrix multiplication. In *2017 IEEE 28th international conference on application-specific systems, architectures and processors (ASAP)* (2017), IEEE, pp. 19–24.

[30] Yamada, Y., and Momose, S. Vector engine processor of NEC's brand-new supercomputer SX-Aurora TSUBASA. In *Proceedings of A Symposium on High Performance Chips, Hot Chips* (2018), vol. 30, pp. 19–21.

[31] Zhang, J., Franchetti, F., and Low, T. M. High performance zero-memory overhead direct convolutions. In *International Conference on Machine Learning* (2018), PMLR, pp. 5776–5785.

[32] Zlateski, A., Lee, K., and Seung, H. S. ZNN–A Fast and Scalable Algorithm for Training 3D Convolutional Networks on Multi-core and Many-Core Shared Memory Machines. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2016), IEEE, pp. 801–811.

A Artifact Instructions Appendix

A.1 Abstract

The artifact instructions appendix describes the procedures to obtain and interact with our reproducibility artifact. Readers may use our artifact to replicate the experiments in the SX-Aurora processor, generate performance plots, and validate our performance claims.

A.2 Description

The artifact package consists of the pre-compiled *benchdnn* application, the oneDNN library targeting the SX-Aurora VE hardware, and auxiliary scripts to facilitate interactions with the artifact. The package includes a text guide covering the steps to conduct correctness checks and performance experiments regarding the convolution algorithms presented in this work. Finally, the package contains python notebooks that implement our performance evaluation methodology, allowing users to inspect intermediate results that lead to our final performance claims.

How to obtain the artifact: download the Zenodo package at https://zenodo.org/record/7371471.

Hardware/Software Dependencies: users interested in reproducing our experiments and collecting performance data require access to a system with the following characteristics.

- **Vector Engine**: NEC SX-Aurora TSUBASA 20B
- **Runtime Environment**: Linux aurora4 v3.10
- **Runtime Libraries**: SX-Aurora OpenMP

Users interested in reproducing our performance evaluation method through python notebooks require Python3.7.4, or later and the following packages: (i) numpy 1.17.2, (ii) pandas 1.2.0, and (iii) matplotlib 3.5.2.

Datasets: The experiments use the *benchdnn* benchmark application, which automatically generates the input data for the convolution operand tensors. We use *benchdnn* to execute convolution operations with identical arguments as the convolutional layers found in ResNet models for the ImageNet challenge.

A.3 Experiment Workflow

The experiments have two stages: (i) correctness checks and (ii) performance data collection. The correctness checks take about ten minutes to complete, asserting that our convolution algorithms generate correct results using a reference implementation. The command below, issued at the package root directory, carries out the correctness checks. The script outputs a CSV file to the terminal, with each line representing a test case and containing a *status* field indicating if the test has *passed* or *failed*.

```
experiments/validate.sh
```

The reader can start the performance data collection with the command below, which takes about twenty minutes to complete. The script outputs a CSV file to the terminal, and each line represents an experiment indexed by: (i) a convolution problem id, (ii) a training direction, (iii) an algorithm, and (iv) a minibatch size. Each CSV line also contains the performance in GFLOP/s and the execution time in milliseconds.

```
experiments/performance.sh
```

A.4 Experiment Evaluation

The performance data collection stage outputs a CSV file with performance metrics for each proposed algorithm. Moreover, the reproducibility artifact includes a set of python notebooks that reads the CSV files and evaluates the performance of each algorithm across all problems, assisting readers with data visualization and clustering tools.

A.5 Notes

The README file in the reproducibility artifact root folder contains additional instructions for customizing the experiments, example datasets, and detailed information about the procedures described in this section.