Rethinking Embedded Blocks for Machine Learning Applications

SEYEDRAMIN RASOULINEZHAD, The University of Sydney, Australia
ESTHER ROORDA and STEVE WILTON, The University of British Columbia, Canada
PHILIP H.W. LEONG and DAVID BOLAND, The University of Sydney, Australia

The underlying goal of FPGA architecture research is to devise flexible substrates which implement a wide variety of circuits efficiently. Contemporary FPGA architectures have been optimized to support networking, signal processing and image processing applications through high precision digital signal processing (DSP) blocks. The recent emergence of machine learning has created a new set of demands characterized by: 1) higher computational density and 2) low precision arithmetic requirements. With the goal of exploring this new design space in a methodical manner, we first propose a problem formulation involving computing nested loops over multiply-accumulate (MAC) operations, which covers many basic linear algebra primitives and standard deep neural network (DNN) kernels. A quantitative methodology for deriving efficient coarse-grained compute block architectures from benchmarks is then proposed together with a family of new embedded blocks, called MLBlocks. An MLBlock instance includes several multiply-accumulate units connected via a flexible routing, where each configuration performs a few parallel dot-products in a systolic array fashion. This architecture is parameterized with support for different data movements, reuse and precisions, utilizing a columnar arrangement that is compatible with existing FPGA architectures. On synthetic benchmarks, we demonstrate that for 8-bit arithmetic, MLBlocks offer 6× improved performance over the commercial Xilinx DSP48E2 architecture with smaller area and delay; and for time-multiplexed 16-bit arithmetic, achieves 2× higher performance per area with the same area and frequency.

CCS Concepts: • **Hardware → Reconfigurable logic and FPGAs**; **Reconfigurable logic and FPGAs**; **Programmable interconnect**; • **Computer systems organization → Systolic arrays**; **Architectures**; **Reconfigurable computing**; **Neural networks**; • **Mathematics of computing**;

Additional Key Words and Phrases: FPGA Architectures, Coarse-grained compute blocks, Reconfigurable Architecture, Neural Networks, Digital signal processing

ACM Reference Format:
Seyedramin Rasoulinezhad, Esther Roorda, Steve Wilton, Philip H.W. Leong, and David Boland. 20xx. Rethinking Embedded Blocks for Machine Learning Applications. *J. ACM* 11, 1, Article 111 (January 20xx), 31 pages. https://doi.org/1x.xxxx/xxxxxxx.xxxxxxx

1 INTRODUCTION

Recent advances in deep neural network (DNN) algorithms have driven unprecedented interest in their embedded implementations. New optimizations in DNN architecture, quantization, memory and datapath organization [1], have enabled field-programmable gate array (FPGA) accelerators

Authors' addresses: Seyedramin Rasoulinezhad, seyedramin.rasoulinezhad@sydney.edu.au, The University of Sydney, Sydney, NSW, 2006, Australia; Esther Roorda, estherr@ece.ubc.ca; Steve Wilton, stevew@ece.ubc.ca, The University of British Columbia, Vancouver, BC, V6T1Z4, Canada; Philip H.W. Leong, philip.leong@sydney.edu.au; David Boland, david.boland@sydney.edu.au, The University of Sydney, Sydney, NSW, 2006, Australia.

0004-5411/20xx/1-ART111 $15.00
https://doi.org/1x.xxxx/xxxxxxx.xxxxxxx

to surpass graphics processing unit (GPU) and application-specific integrated circuit (ASIC) implementations in performance [2]. The ultimate goal in FPGA architecture design is to provide a reconfigurable and flexible platform that can implement a wide variety of circuits. Existing commercial FPGA architectures are mature and have evolved from decades of optimization for traditional networking, image processing and signal processing applications. Typically, these circuits use relatively high-precision arithmetic and have been addressed by digital signal processing (DSP) units which support high-precision multiply-accumulation (MAC) operations.

Recent research has shown the efficacy of low-precision fixed point and block floating point [3] arithmetic for the implementation of DNNs in both inference [4] and training tasks [5]. The importance of these solutions for compute-bound DNN applications is likely to grow in importance in the future. This motivates our goals of improving FPGA performance with coarse-grained blocks that provide better support for low precision, since any dedicated course-grained blocks are likely to outcompete any LUT-based solution. However, developing such blocks in a systematic fashion requires: 1) a generalized problem formulation that covers the relevant range of computations, and 2) an efficient mapping to configurable architectures.

In this paper, we develop a tool to create a generalized embedded block (EB) architecture called an MLBLock, constructed from low-precision MAC units and programmable routing. The difficulty in its design lies in ensuring maximum utilisation of the MAC units over a benchmark set with minimal routing overhead. To achieve this, we introduce a methodology which allows all potential loop unrollings to be enumerated and analysed over a range of DNN computations. These computations include standard, depth-wise, dilated and point-wise convolutions, fully-connected layers, and recurrent neural networks (RNNs) such as long short-term memories (LSTMs). Using this analysis, our tool generates an MLBlock that supports a subset of loop unrollings that best satisfy the area/performance trade-offs. This serves as the target architecture to which algorithms can be efficiently mapped. Specifically, our contributions are as follows:

- A methodology in which different algorithms in the form of a benchmark set are mapped to MLBlocks. Algorithm descriptions are of the form of MAC operations within a number of nested loops that is a generalization of convolution computation. Using a set of loop transformations that covers a solution space, projections with different hardware tradeoffs are generated and a specific configuration of an MLBlock which supports all the benchmarks identified.
- A case study demonstrating the application of this methodology, with some restrictions, to automatically find a DSP-like replacement block that demonstrates higher performance and efficiency across a benchmark suite in comparison to existing expert-designed architectures.
- Confirmation that MLBlocks are suitable for implementation in the familiar columnar manner and compatible with existing FPGA architectures.
- Greedy and heuristic approaches to select an implementable projection set from the set of possible projections which achieve a balance between flexibility and performance.
- A parameterized FPGA module generator for MLBlocks which compiles a projection set to Verilog, creating an instance that uses a specified number of MAC units (called MLBlock-*M*). The suggested EB architecture offers a bijection from a projection to an MLBlock configuration.
- A quantitative architectural study of the performance benefits of augmenting the Xilinx FPGA architecture with MLBlocks.

Open source Python tools to implement the techniques described in this manuscript and reproduce our results will be made available upon publication.

The remainder of the paper is organized as follows. In Section 2, we provide the background for this work. In Section 3, we describe our design methodology having an algorithm template that

covers a superset of DNN computations and many basic linear algebra subprograms (BLAS), and techniques to analyse the potential computation of an EB, their costs and methods to find the right trad-off for different implementation candidates. In Section 4, we describe a generalized EB target architecture to which a set of projections can be mapped. Results are presented in Section 5 and conclusions in Section 6.

2 BACKGROUND

2.1 FPGA Architectures and Motivation

After decades of FPGA architecture research, commercial FPGA architectures have evolved to comprise both fine and coarse-grained reconfigurable blocks arranged in a columnar fashion. The most basic units are logic elements (LEs) which are built from look-up tables (LUTs) and additional logic such as adders and flip-flops. For higher area efficiency and speed, commonly-used circuits such as memories, DSP blocks and microprocessors are implemented as coarse-grained EBs. A flexible interconnection network is used to connect LEs and EBs to input/output (IO) blocks which include general IO, memory interfaces, and transceiver blocks. Together these form a programmable system on chip which can implement arbitrary circuits.

Designing the EBs requires making tradeoffs between 1) flexibility to support a wide range of domains and 2) specialization to efficiently support selected applications. This is conventionally done by evaluating their utility over a set of benchmark problems. However, to date the benchmarks for FPGA evaluation have not included low precision, quantized DNN applications. Such applications are dominated by MAC operations which should be mapped to DSPs and LEs [6]. However, DSPs are heavily underutilized for low-precision DNNs. While the Xilinx DSP48E2 is capable of executing a 27×18 multiply and 48-bit accumulate operation, for the low precision case it only delivers two 8×8 multiplies (with shared multiplicand) with 24-bit accumulation. This is roughly a third of its potential since a 27×18-bit multiplier occupies the area of roughly six 9×9-bit ones [7]. As a result, even in a state of the art accelerator, DSPs impose a performance limit [8]. The Intel FPGA architecture has similar limitations.

This inefficiency causes a barrier to higher performance low-precision implementations compared to GPUs, e.g., for INT8 operations, the embedded Jetson Xavier NX GPU [9] offers 21-TOPs which would require a high-end Virtex UltraScale+ FPGA with 6,840 DSPs for comparable performance [10].

2.2 Previous works

Compared to ASICs and GPUs, FPGAs suffer from lower maximum clock frequency and larger area, which subsequently impacts performance. To address these issues, we need to increase the compute capacity. Routing restrictions and efficiency make it impractical to add a sea of additional multipliers or MAC units to FPGA architectures as programmable routing is both expensive and power-hungry. However, it is possible to include dense compute blocks while meeting the IOs and area budget limits. Solutions are categorized into three main tiers, described in the following subsections. The MLBlocks approach differs from all three in that we derive a near-optimal EB by considering the mapping of an arbitrary set of benchmarks of a certain design pattern onto a flexible MLBlock fabric. Considering nested loops as a generalization for DNN implementation was first proposed by Yang et. al. [11], our approach of optimizing designs by considering different tiling patterns over a set of benchmarks for DNNs is a generalization of that described by Zhang et. al. [12].

2.2.1 Enhancing existing DSP blocks. This work attempts to optimize FPGAs for DNNs by providing DSPs with low-precision support. Fracturable multipliers were introduced in Stratix-V [13]. Boutros at al. [14] proposed a bespoke structure for Intel DSPs to support arithmetic down to

INT4. Rasoulinezhad et al. [15] proposed divide-and-conquer and decomposition techniques to break high-precision MAC operations into multi lower-precision dot-products for any DSP block. This work also explored low-precision streaming and weight reuse using DSP interconnections. Low precision is also supported on commercial platforms such as Intel Agilex which offers native support for INT9 and FP16 by DSP blocks [16]. A recent Xilinx DSP architecture, the DSP58, offers a broader range of precisions compared to its predecessor (the DSP48E2), while providing complete backward compatibility [17]. A single instance of DSP58 is capable of MAC operations up to 27×24 bits, wider pre-adder and logic circuits, as well as support for a run-time configurable 3-element dot-product of 9×8-bit signed values. The main issue with this approach is it increases the critical path for extreme low-precision applications, since flexibility requires more multiplexers in the implementation.

Recent work on LE structures by Rasoulinezhad et al. [18] enhanced the support for low-precision multi-operand operations such as dot-product operations. Similar work by Boutros et al. [19] also suggest LE-level modifications which offer better support for low-precision adders and multipliers. More comprehensive details of this research work is presented in [20].

2.2.2 Integrating domain-specific engines. Adding hardened domain-specific processing units is a new commercial trend. The Xilinx Versal architecture uses a coarse-grained gate array (CGRA) of AI-engines, on the same die as the FPGA [21], interconnected via a network on chip. An AI-engine is a simple RISC processor enriched by fixed and floating-point SIMD units accessing dedicated register file, data and program memory, and streaming interconnections. This introduces an additional heterogeneous compute platform and network on chip to manage, and the RISC processor is large in area compared with a DSP block.

Intel Agilex FPGAs utilize chiplet technology for connecting custom circuits from separate dies as a system in package [16]. This enables integration of an embedded ASIC (eASIC) Intel tensor tile architecture with promising results for deep learning benchmarks [22].

A fundamental issue with domain-specific approaches reviewed is they advocate heterogeneous FPGA and CGRA resources as a solution for ML. This does not directly address the shortcomings of current FPGA architectures, instead augmenting the fabric with CGRA resources off to the side. The other issue is resource duplication where the separation of CGRAs from the FPGA resources necessitates duplicating memory, buffer and routing resources on CGRA side.

2.2.3 Designing a new embedded block. Achronix MLP blocks offer flexible multi-precision integer / floating-point dot-product operations [23]. This block also includes dedicated large memories to feed the data via parallel IOs. However, support of different data movements such as windowing is missing, resulting in large IO requirements. This increase the complexity of routing as data movement requires using logic element circuits. MLP72 is capable of computing dot-products of 32, 16, 4 and 2 pairs of inputs respectively with 4-bits, 8-bits, 16-bits and 16-bit floating-point precision. This block also natively supports block floating point arithmetic through dedicated circuitry for exponent addition and subtraction. Achronix Speedster7t chips also offer bus routing and a cascadable, hardened max function at switch boxes. An 8-bit configurable ALU and fracturable look up tables provide efficient support for low-precision arithmetic.

Intel AI-Tensor is another commercial example introduced in the Intel Stratix 10 NX FPGA architecture [24]. Tensor blocks compute three one-input-shared 10-element dot-products. This EB can deliver thirty low-precision MAC operations with high utilization rates for low batched computations using double buffering. As we show, this block is a special case of MLBlocks.

Recently, Arora et al. [25] proposed an output stationary 2D systolic array block to augment an FPGA architecture. The blocks can be composed through a 2D mesh arrangement to create larger systolic arrays. This architecture focuses on only one computation, namely matrix multiplication.

Algorithm 1: Pseudo code for the generalized nested loop model (loop format for each $v \in \mathbb{V}$ is $v \leftarrow V^{init}{:}V^{stride}{:}V^{limit}$)

for $g_0 \leftarrow 0 : G_0^{stride} : G_0^{limit} - 1$
  for $g_1 \leftarrow 0 : G_1^{stride} : G_1^{limit} - 1$
  ...
    for $g_n \leftarrow 0 : G_n^{stride} : G_n^{limit} - 1$

      for $b_0 \leftarrow 0 : B_0^{stride} : B_0^{limit} - 1$
        for $b_1 \leftarrow 0 : B_1^{stride} : B_1^{limit} - 1$
        ...
          for $b_m \leftarrow 0 : B_m^{stride} : B_m^{limit} - 1$

            for $e_0 \leftarrow 0 : E_0^{stride} : E_0^{limit} - 1$
              for $e_1 \leftarrow 0 : E_1^{stride} : E_1^{limit} - 1$
              ...
                for $e_p \leftarrow 0 : E_p^{stride} : E_p^{limit} - 1$

                  for $r_0 \leftarrow 0 : R_0^{stride} : R_0^{limit} - 1$
                    for $r_1 \leftarrow 0 : R_1^{stride} : R_1^{limit} - 1$
                    ...
                      for $r_q \leftarrow 0 : R_q^{stride} : R_q^{limit} - 1$
                        $\mathbf{O}[g_0, g_1, .., b_0, b_1, .., e_0, e_1, ..]$ += $\mathbf{I}[g_0, g_1, .., b_0 \pm r_0, b_1 \pm r_1, ..] \times \mathbf{W}[g_0, g_1, .., e_0, e_1, .., r_0, r_1, ..]$

Unfortunately, many applications do not map well to matrix multiplication units, e.g. 1D data processing and Microsoft Brainwave-like accelerators that uses Matrix-Vector Unit (MVU) as the primitive. As a follow-up, Arora et al. [26] proposed Tensor Slice. This EB is a flexible array of processing elements supporting multiple hand-picked tensor operations with various precisions. Although this is the most similar approach to ours, our work is different as we propose a systematic approach rather than a manual one. The main question that we address is, "What computations should a distributed course-grained FPGA block for low-precision support?"

3 MLBLOCKS DESIGN METHODOLOGY

3.1 Overview

An EB must support all algorithms in a user-specified benchmark suite and we assume all algorithms can be written as a set of nested constant loops. The innermost computation could be any arithmetic operation; however, in this paper, we only consider low-precision multiply-accumulate (MAC) operations with three inputs (**I**, **W**, and **O**) and one output (**O**). A template for our algorithm descriptions is given in Algorithm 1. While straightforward, this is sufficient to describe a broad range of DNN related computations as demonstrated by our chosen benchmark suite in Section 5.

An overview of our approach to determine the optimal EB is given in Figure 1. Assuming a fixed unrolling factor (M), we generate a list of *unrollings* for each algorithm in the benchmark suite. We describe each unrolling instance through a compact representation that we call a *projection*. Each projection has a one-to-one mapping with a hardware EB configuration. We then perform *selection* by analyzing the list of all projections to identify a subset that will cover all algorithms in the given benchmark suite, maximise MAC utilisation, and minimise the implementation cost. We call the resulting subset the *selected projections*. In the generation stage, a tool merges all configurations to produce a Verilog description, namely an MLBlock-M.

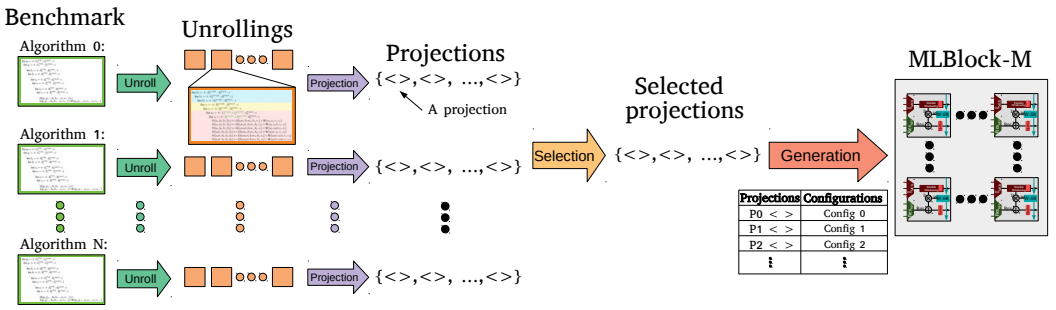


Fig. 1. An overview of MLBlock generation for a given benchmark suite

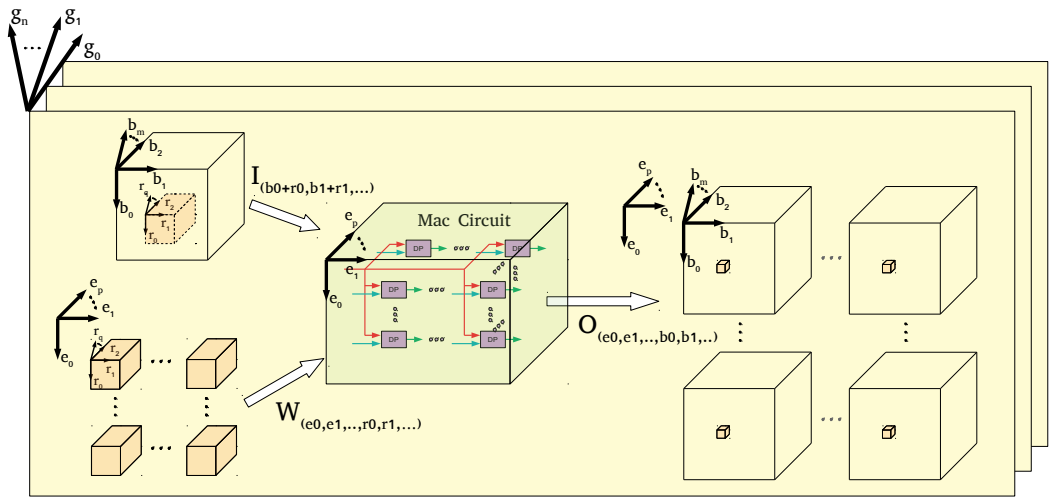


Fig. 2. A visualization for algorithms covered by Algorithm 1.

3.2 Benchmark Algorithm Template

Algorithm 1 is our generalized algorithm template that takes advantage of the fact that in DNN layers and basic linear algebra subroutines (BLAS), many loop variables have similar data access patterns; these result in similar hardware realizations. To characterise these access patterns, we define that loop variables $(r_i, e_i, b_i, g_i)\ \forall i$, *accesses* an input/output if it reads the input/output. Considering MAC operation input and outputs, the loop variables can then be classified into one of four variable group sets ($\{R, E, B, G\}$), the first character being used to identify the variable names according to:

(1) **Reduction (R)**: The loop variables index elements of **I** and **W** to produce a single output in **O**, e.g. a dot-product.
(2) **Expansion (E)**: The loop variables index elements of **W** and **O** to produce multiple values of **O** reusing **I**, e.g. processing different kernels for the same input.
(3) **Batching (B)**: The loop variables index elements of **I** and **O** to generate multiple values of **O**, reusing **W**, e.g. inference over different inputs.
(4) **Grouping (G)**: The loop replicates computations performed in other loops, e.g. depth-wise and grouped convolutions.

Algorithm 2: Pseudo code for a batched standard 2D convolution layer

```
for x ← 0 : X^stride : X^limit − 1                          ▷ This loop variable is of type Batching (b0)
  for y ← 0 : Y^stride : Y^limit − 1                        ▷ This loop variable is of type Batching (b1)
    for b ← 0 : B^stride : B^limit − 1                      ▷ This loop variable is of type Batching (b2)
      for k ← 0 : K^stride : K^limit − 1                    ▷ This loop variable is of type Expansion (e0)
        for f_x ← 0 : F_X^stride : F_X^limit − 1            ▷ This loop variable is of type Reduction (r0)
          for f_y ← 0 : F_y^stride : F_Y^limit − 1          ▷ This loop variable is of type Reduction (r1)
            for c ← 0 : C^stride : C^limit − 1              ▷ This loop variable is of type Reduction (r2)
              O[b, x, y, k] += I[b, x + f_x, y + f_y, c] × W[k, f_x, f_y, c]
```

Mapping different algorithms to this format may require arbitrary numbers of variables in each group, where this is parameterized by *n*,*m*,*p*, and *q* respectively for grouping, batching, expansion, and reduction variable groups in Algorithm 1. Let $\mathbb{V} = R \cup E \cup B \cup G$ denote the set of all loop variables, where each $v \in \mathbb{V}$ iterates according to an initial, stride, and limit value of V^{init}, V^{stride}, and V^{limit}. Without loss of generality, we assume the initial values are all zero.

A visualization of algorithms covered by this generalization of convolutional layer in a DNN is presented in Figure 2. Inputs I and W are convolved to produce output O. Briefly, the number of input groups and their sizes are described by variables from grouping and batching variable groups. Also, the number of convolution kernels for each input group and their sizes are expressed by variables from expansion and reduction variable groups respectively. This model can represent all the common DNN layers such as standard, depth-wise and point-wise convolutions, as well as many BLAS functions that can be used to implement fully connected layers. For instance, the computation of a batched standard 2D convolutional layer, shown in Algorithm 2, can be mapped to this model using seven nested loops where variables b_0, b_1, b_2, e_0, r_0, r_1, r_2, are input height (X), width (Y) and batch size (B), number of filters (K) and their height (F_X), width (F_Y), and depth (C) respectively. In this example, b_0, b_1 and b_2 are the batching variables; e_0 is an expansion variable; r_0, r_1 and r_2 are reduction variables; and there is no grouping variable. Loop variables of the same group define dimensions of the corresponding access pattern.

3.3 Unroll

For each algorithm in the benchmark set, we study the application of the following techniques: 1) tiling via loop splitting, 2) fully/partially unrolling loops, 3) reordering loops, and 4) partitioning the scheduling and compute boundary. This is similar to the approach of Yang et al. [11]. However, we introduce an extra partitioning level which defines the computation to be performed on EBs, accelerator and data scheduler. Our technique restructures the loops so that the EBs can receive input data in parallel from an on-FPGA scheduler using on-chip memory. The on-chip memory receives data from off-chip through a software data scheduler. In this design space, we seek the instances where the EB computation is fully unrolled with unrolling factor M. This parameter defines the parallelism in EBs and is equal to the number of MAC operations in each EB. In our accelerator, all EBs perform the same computation. As detailed later, explicit formulas for utilisation and I/O requirements can be determined, allowing design tradeoffs to be optimized prior to hardware translation.

Algorithm 3 shows one potential unrolling instance of Algorithm 2, where the the variables f_x and b are unrolled. We describe the unrolling by the variables $\hat{F}_X^{\text{unroll}} = 3$ and $\hat{B}^{\text{unroll}} = 2$. To support this unrolling, the computation is partitioned into three shaded areas, assigned to the off-chip data scheduler, scheduler on the FPGA accelerator, and each EB in blue, yellow, and red respectively. The EB computations in red describe a unique, spatially parallel data path that can be translated to

Algorithm 3: Pseudo code of an unrolling instance for a batched standard 2D convolution layer ($M = 6$, $\hat{F}_X^{\text{unroll}} = 3$, $\hat{B}^{\text{unroll}} = 2$). The blue, red, and yellow area defines the computation partitions assigned to data scheduler, accelerator, and EBs, respectively. The computation on EBs can be performed in parallel (see Figure 3a).

for $x \leftarrow 0 : X^{\text{stride}} : X^{\text{limit}} - 1$
  for $y \leftarrow 0 : Y^{\text{stride}} : Y^{\text{limit}} - 1$
    for $k \leftarrow 0 : K^{\text{stride}} : K^{\text{limit}} - 1$
      for $f_y \leftarrow 0 : F_Y^{\text{stride}} : F_Y^{\text{limit}} - 1$
        for $c \leftarrow 0 : C^{\text{stride}} : C^{\text{limit}} - 1$
          for $b \leftarrow 0 : B^{\text{stride}} \times B^{\text{unroll}} : B^{\text{limit}} - 1$
            for $f_x \leftarrow 0 : F_X^{\text{stride}} \times F_X^{\text{unroll}} : F_X^{\text{limit}} - 1$
              $\mathbf{O}[b, x, y, k] \mathrel{+}= \mathbf{I}[b, x{+}f_x, y{+}f_y, c] \times \mathbf{W}[k, f_x, f_y, c]$
              $\mathbf{O}[b, x, y, k] \mathrel{+}= \mathbf{I}[b, x{+}f_x{+}1, y{+}f_y, c] \times \mathbf{W}[k, f_x{+}1, f_y, c]$
              $\mathbf{O}[b, x, y, k] \mathrel{+}= \mathbf{I}[b, x{+}f_x{+}2, y{+}f_y, c] \times \mathbf{W}[k, f_x{+}2, f_y, c]$
              $\mathbf{O}[b{+}1, x, y, k] \mathrel{+}= \mathbf{I}[b{+}1, x{+}f_x, y{+}f_y, c] \times \mathbf{W}[k, f_x, f_y, c]$
              $\mathbf{O}[b{+}1, x, y, k] \mathrel{+}= \mathbf{I}[b{+}1, x{+}f_x{+}1, y{+}f_y, c] \times \mathbf{W}[k, f_x{+}1, f_y, c]$
              $\mathbf{O}[b{+}1, x, y, k] \mathrel{+}= \mathbf{I}[b{+}1, x{+}f_x{+}2, y{+}f_y, c] \times \mathbf{W}[k, f_x{+}2, f_y, c]$

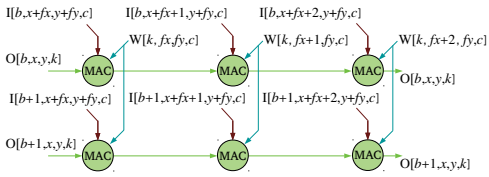
a hardware accelerator. The corresponding computation is depicted in Figure 3a. Note, due to the unrolling, the loop stride size also increases by the same factor.

However, for $M = 6$, there are many other potential unrolling instances. Figure 3 shows five of these for Algorithm 2. Figure 3b unrolls c and b, Figure 3c unrolls k and c, Figure 3d unrolls x and b, and Figure 3e unrolls b and f_x. Note that Figure 3a and Figure 3b share the same MAC arrangement and interconnection scheme; indeed there are other unrolling instances that could also share the same datapath. It follows that the same circuit can be used to support multiple unrolling instances, discussed further in Section 3.4. Alternatively, there is only a minor routing overhead to support Figure 3c and Figure 3e, this can be considered when selecting an optimal subset of desirable projections, discussed further in Section 3.5.

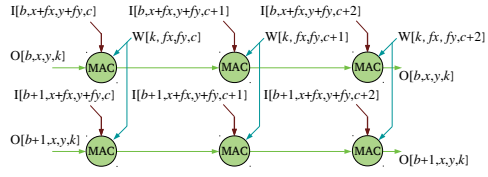
Figure 4 and Figure 5 illustrate how the utilisation rate of these unrolling instances will differ depending on the target algorithm from the benchmark set for depth-wise and point-wise convolution kernels, respectively.

First let us compare how depth-wise convolution with 3×3 kernels can be scheduled based on unrollings of Figure 3a, Figure 3d and Figure 3e; their utilisation is described by Figure 4a, Figure 4b and Figure 4c respectively. A depth-wise convolution multiplies inputs by weights and accumulates over the size of the kernel. Since Figure 3a provides parallel access over the variable f_x (also with access size suitable to 3×3 kernels), 100% utilisation is achieved; the parallelism over b and shared f_x enables two batches to be computed simultaneously. The other rows of the kernel can be computed with separate EBs again with 100% efficiency. However, the unrolling of Figure 3c does not provide parallel access to f_x, meaning accumulation must be done using separate EBs. Furthermore, there is no need to access k in parallel resulting in very low utilisation. A better unrolling with $f_x^{\text{unroll}} = 2$, $B_y^{\text{unroll}} = 3$, as shown in Figure 3e achieves 75% utilisation as there are still unused multipliers due to the 3×3 kernel size. Note, for low-batch inputs, unrolling variable b may decrease utilisation.

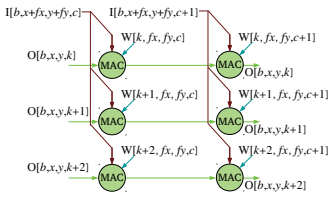
Now let us compare how point-wise convolution tiles with these same unrollings. Point-wise convolution accumulates over channels (c). This maps well to the unrolling of Figure 3c, which will achieve 100% utilisation with an even number of channels; with odd numbers of channels, the final EB will not be fully utilised, as seen in the Figure 5b. The unrolling of Figure 3a achieves poor utilisation as accumulation over channels must be between EBs. Figure 3b performs better as



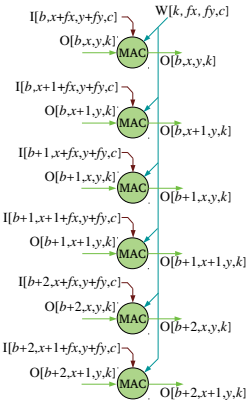
(a) $\hat{F}_X^{unroll} = 3, \hat{B}^{unroll} = 2$



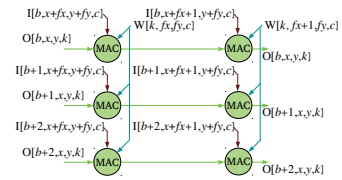
(b) $\hat{C}^{unroll} = 3, \hat{B}^{unroll} = 2$



(c) $\hat{K}^{unroll} = 3, \hat{C}^{unroll} = 2$



(d) $\hat{X}^{unroll} = 2, \hat{B}^{unroll} = 3$



(e) $\hat{B}^{unroll} = 3, \hat{F}_X^{unroll} = 2$

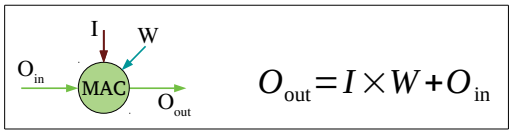
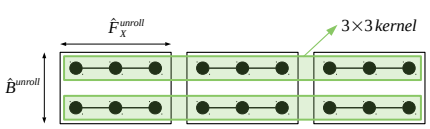
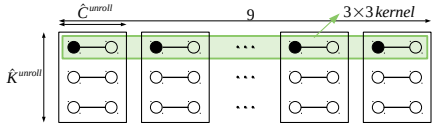


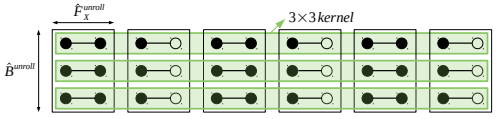
Fig. 3. EB computations for four unrolling instances of Algorithm 2 assuming $M = 6$.



(a) Scheduling using the unrolling in Figure 3a



(b) Scheduling using the unrolling in Figure 3c



(c) Scheduling using the unrolling in Figure 3e

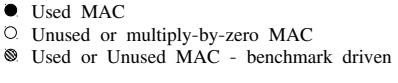


Fig. 4. Scheduling depth-wise convolution kernels on different unrolling instances

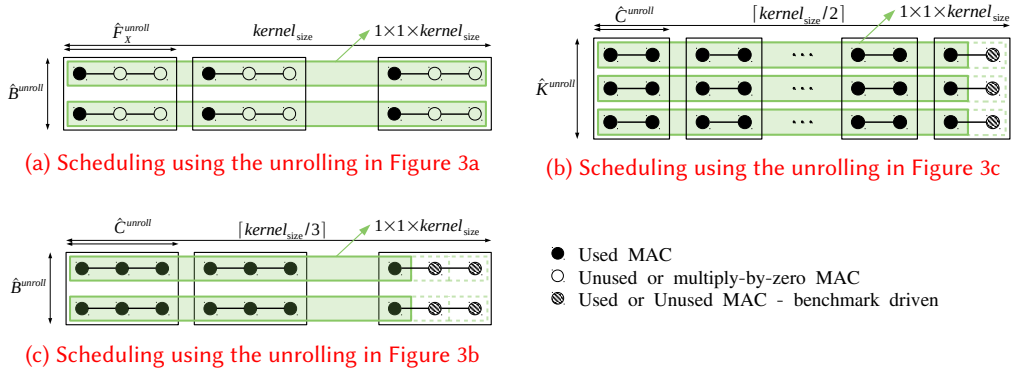


Fig. 5. Scheduling point-wise convolution kernels on different unrolling instances

channels can be accumulated over EBs, but this may result in lower efficiency in the final EB, as shown in Figure 5c.

Given these examples, if our benchmark suite were to only consist of depth-wise convolutions, the EBs to support the unrolling of Figure 3a would be a good solution. Alternatively, if our benchmark suite were to only consist of point-wise convolutions, we would only be interested in developing EBs to support the unrolling of Figure 3c. If the benchmark suite were to consist of both point-wise and depth-wise convolutions, we may wish to develop EBs that can support the unrollings of both Figure 3a and Figure 3c. However, this comes at a hardware cost of additional routing logic. An alternative would be to support unrollings of Figure 3a and Figure 3b. While this would have slightly worse utilisation, supporting either pair would introduce no additional hardware cost, relying on scheduling to ensure correct data patterns.

The rest of this section describes how we navigate these trade-offs. In the unrolling stage, we simply explore all potential unrollings and evaluate their utilisation. In the projection stage, we simplify this by identifying the unique unrollings with best utilisation (e.g. Figure 3a and Figure 3b can be described by the same projection). In the selection stage, we describe how we identify the best projections to achieve high utilisation with minimal hardware cost.

3.4 Projections

In order to uniquely identify the computation of unrolling instances and analyse their performance, we first develop a compact representation that we describe as a *projection*. The foundation for this format is the fact that loop variables belonging to the same variable groups access the IOs similarly for the same computation while in different dimensions. Thus, the computation and IO bandwidth requirements of the EB remain the same.

Specifically, we define unrolling degree for each variable group as the product of all unrollings for the relevant variable group. We then employ a list to distinguish between unique computation projections as follows:

$$< U_{\mathrm{R}}, U_{\mathrm{E}}, U_{\mathrm{B}}, U_{\mathrm{G}} > \tag{1}$$

For instance, recall from Algorithm 2 that the loop variables x, y and b are Batching variables, k is an Expansion variable, f_x, f_y and c are Reduction variables. Since f_x is unrolled by a factor of 3 and b is unrolled by a factor of 2, the relevant projection for Figure 3a and Figure 3b is <3,1,2,1>.

3.4.1 Computation model. The projection encodes the necessary information to realize a unique computation data path. Given a projection, the number of MACs, M is given by (2).

$$M = U_{\mathrm{R}} \times U_{\mathrm{E}} \times U_{\mathrm{B}} \times U_{\mathrm{G}} \tag{2}$$

3.4.2 I/O constraints. Furthermore, the projection defines the required IO bandwidth for each inputs and outputs from the matrices I, W and O. We consider the bandwidth for each input separately as follows:

$$Bandwidth_{\mathrm{I}} = U_{\mathrm{G}} \times U_{\mathrm{B}} \times U_{\mathrm{R}} \times P_{\mathrm{I}} \tag{3}$$

$$Bandwidth_{\mathrm{W}} = U_{\mathrm{G}} \times U_{\mathrm{R}} \times U_{\mathrm{E}} \times P_{\mathrm{W}} \tag{4}$$

$$Bandwidth_{\mathrm{O}} = U_{\mathrm{G}} \times U_{\mathrm{B}} \times U_{\mathrm{E}} \times P_{\mathrm{O}} \tag{5}$$

, where the P parameters represent the data precision for the corresponding IO. This assumes all data must be streamed into the EB in parallel each cycle, without serialization or double buffering. Delivering the data to this dense compute unit relies on the flexible data movements of the FPGA fabric [11]. The total bandwidth is then described by (6). Note, the bandwidth for O is counted twice since it appears in the both inputs and output.

$$Bandwidth = Bandwidth_{\mathrm{I}} + Bandwidth_{\mathrm{W}} + 2 \times Bandwidth_{\mathrm{O}} \tag{6}$$

3.4.3 Windowing. In practice, the required bandwidths limit the unrolling degrees when an EB with a constrained IO budget is desired. However, in the case of neural networks, spatial locality is often exploited to reduce I/O requirements by using windowing to reuse of input data. In our algorithm template, it is feasible only for batching variables where they access an index in combination with a reduction variable, i.e., X and Y where access two different dimensions of input I in combination with F_X and F_X respectively.

In this work, windowing in only one dimension is considered. For higher dimensions, line buffers are required, and we assume that these are implemented outside of the EBs and streamed. We support multiple types of windowing that are common in modern CNNs, as shown in Figure 6. Figure 6a illustrates the simplest form, where the streamed data is delayed by one cycle via a shift register. In the case of Figure 6b, the computation is performed over a larger window, but not all values stored in this window are utilised. Comparing Figure 6a and Figure 6b, the same input bandwidth is required, but the hardware cost to support the latter is higher. Figure 6c shows windowing with a larger stride. In comparison to Figure 6a, the hardware requirements are approximately the same, but the routing is slightly different. Figure 6d shows a combination of a larger window with strided input, where not all values are utilised. Once again, the hardware cost is similar to Figure 6b, but the routing is different.

When enumerating all types of unrolling, it is important also to keep track of the relevant windowing. This will ensure that the final EB architecture can support all the desired forms of windowing. Referring to Figure 6, a window can be described by 1) window length, W_{length} 2) number of samples utilised per window, W_{samples} 3) window stride, W_{stride}. We define W_{buffer} as the number of storage elements required to delay the data between two cascaded MAC operations. This will be used for the implementation of the windowing circuit.

To include the windowing parameters in the projection, we update the initial projection notation in Equation 1 to Equation 7, where $U_{\mathrm{R}} = U_{\mathrm{R}}^{\mathrm{W}} \times U_{\mathrm{R}}^{\mathrm{N}}$, where superscript $^{\mathrm{W}}$ and $^{\mathrm{N}}$ distinguish the unrolling in windowing and non windowing part. This includes all potential forms of windowing.

$$< (U_{\mathrm{R}}^{\mathrm{W}}, W_{\mathrm{buffer}}, W_{\mathrm{stride}}), U_{\mathrm{R}}^{\mathrm{N}}, U_{\mathrm{E}}, U_{\mathrm{B}}, U_{\mathrm{G}} > \tag{7}$$

For example, in the case of Algorithm 3, if windowing is applied, the relevant projection is <(3,1,1),1,1,2,1>; if windowing is not applied, the relevant projection is <(1,-,-),3,1,2,1>. Note, Equations 2, 4, and 5 are still valid. However, due to the optimization for input I, the $Bandwidth_{\mathrm{I}}$ is

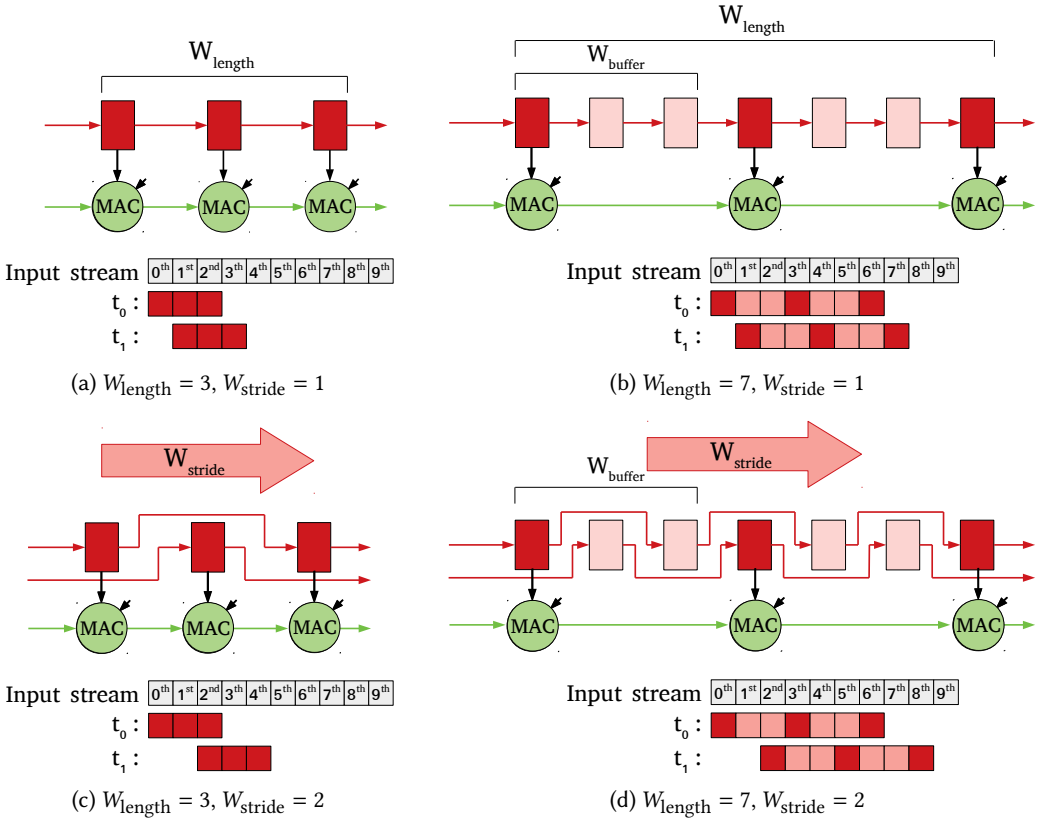


Fig. 6. Few sample of Generalized windowing regarding Algorithm 2 (In all cases $W_{\text{sample}} = 3$)

calculated as follows:

$$Bandwidth_{\text{I}} = \begin{cases} U_{\text{G}} \times U_{\text{B}} \times U_{\text{R}}^{\text{N}} \times W_{\text{stride}} \times P_{\text{I}} & \text{if } (W_{\text{stride}} \leq W_{\text{length}}) \\ U_{\text{G}} \times U_{\text{B}} \times U_{\text{R}}^{\text{N}} \times W_{\text{length}} \times P_{\text{I}} & \text{other wise} \end{cases} \tag{8}$$

Our generalized windowing approach enlarges the design search space but supports new types of computations such as dilated convolutions [27, 28], recently proposed for temporal convolutional networks [29]. Note that windowing is a feature of ASIC/FPGA architectures which is unavailable on GPUs where a convolution-like computation must be promoted to matrix multiplication which requires an expensive data expansion.

3.5 Selection

Each projection eventually maps to a hardware configuration regardless of the EB architecture (Figure 1). Due to the design constraints, performance, and implementation costs, supporting all possible projections in an EB is not necessarily desirable or even feasible. Selection is the process of picking a reasonable subset of possible projections, called selected projections. This should exclude projections that do not meet *design constraints* and find the best tradeoff between projection subset, performance, and implementation costs according to the optimization objective.

Design constraints: This includes hard limits on EB interface and implementation cost. The required Bandwidth for I, W, and O is given by the maximum bandwidth of the corresponding inputs and outputs among the selected projections. Note, that the limit for I, W and O are not necessary identical, and will be determined by the available on-chip memory interfaces. Implementation costs, such as area and clock frequency limit can be used to further limit the projection set.

Objective: To take into account both performance and implementation costs while comparing two different projection subsets, we define compute density as the selection objective as follows:

$$\text{compute density} = M\frac{set_{\text{utilization}}}{set_{\text{area}}} \tag{9}$$

where set_{area} is the area cost over the projection subset (computed by synthesis) and $set_{\text{utilization}}$ is the average utilization rate over the benchmark. The utilization rate for each algorithm is the maximum utilization rate of that algorithm over each of the projections in the projection subset, *i.e.* that corresponding to the best projection for that algorithm. This measures the highest achievable percentage of time EB MACs are performing useful computation. To find the utilization rate of a projection for implementing an algorithm, we iterate all unrolling instances which map to that projection to find the best tiling using that projection for the given algorithm.

Using these metrics, we can analyse the search space, which involves millions of possible projection subsets. We comment that it is not possible to simply enumerate and synthesize all potential designs. Due to the number of possible subsets, we implemented two different selection strategies 1) a fast greedy and 2) a heuristic, called N-Config.

Greedy: This approach incrementally builds a subset of projections by iterating over benchmark cases one by one. In each iteration, it initially finds the best performing projections for a benchmark case, only considering utilization rate, available MACs and IO requirements. It then checks whether any of the best performing projections is already a member of the selected projection. If not, it adds one of them to the selected subset randomly. This approach optimizes the average utilization rate without considering implementation cost. Figure 7a illustrates the process.

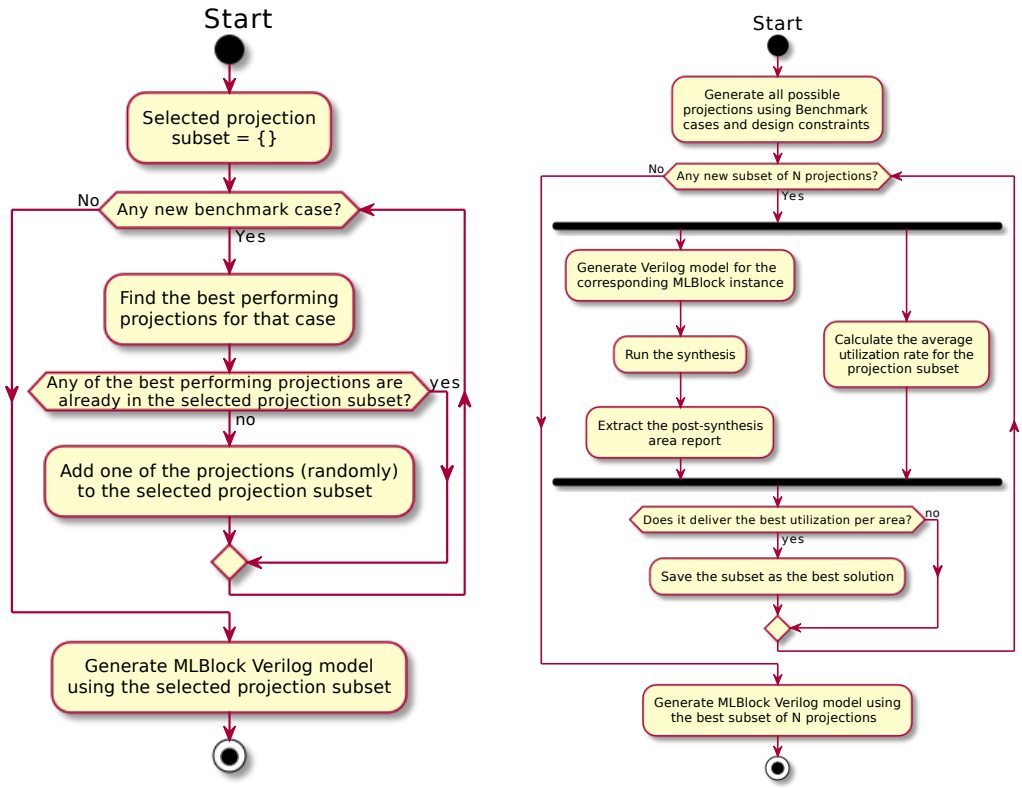
N-Config: This technique, shown in Figure 7b, considers both performance and implementation area. N-Config exhaustively searches for the best solution in the space of T projections considering subsets of N projections at a time. To do so, it generates the Verilog model for the MLBlock instance defined by the selected N configurations, runs synthesis and uses the post synthesis area to optimize for the best compute density (9). This requires $\binom{T}{N}$ calls to the synthesis process and as N is increased, the search quickly becomes intractable. Thus only a relatively small N is used. Increasing T also increases the number of possible solutions, which in turn depends on 1) the unrolling factor M, 2) and the computation diversity among the benchmark cases, e.g. various strides and dilations.

3.6 Generation

The generation step in Figure 1 takes the selected projections, i.e. a set of projections in the form of Equation 7 as inputs, and outputs an EB, where each projection is mapped to a hardware configuration offering the corresponding computation. This requires a parameterized EB architecture based on the projection format. We will suggest an example for such architecture in Section 4.

4 ARCHITECTURE

Generation involves providing the flexible routing (via direct connections and multiplexers) required to connect input streams to the MAC unit and implement the selected projections. The MLBlock architecture is based on a desire to balance two key features: 1) **flexibility** and 2) **modularity**.



(a) Greedy search (b) N-Config search

Fig. 7. Projection selection techniques

Our approach has a precision agnostic routing scheme and allows the integration of multi-precision operations using serial multiplication.

Choosing the dataflow for our systolic implementation is an important high-level design decision. We selected a W-stationary approach because: 1) FPGA memory architectures can efficiently implement streaming and windowing (using BRAMs as line-buffers) for **I** signals, and 2) **O**-stationary requires both **I** and **W** data movements. The chosen approach relaxes the IO requirements of delivering W values every cycle by feeding them in a serial manner and reusing them.

4.1 Parameterized MLBlock-M

An MLBlock-M is defined by the number of MAC units (M), and a set of configurations. By changing M, we explore different MLBlocks. A computation projection is a specific routing instance of MAC units. Figure 8 shows a parameterized datapath for a given computation projection of the form of Equation 7. Parameters $U_{\mathrm{R}}^{\mathrm{W}}$ and $U_{\mathrm{R}}^{\mathrm{N}}$ describe the reduction circuit (dot-product), in which $U_{\mathrm{R}}^{\mathrm{N}}$ groups of $U_{\mathrm{R}}^{\mathrm{W}}$ **I**-**O**-cascaded MAC units are **O**-cascaded. This computation is shown in the purple and red areas. The required windowing values are described by W_{buffer} and W_{stride} which is the same for all MAC units.

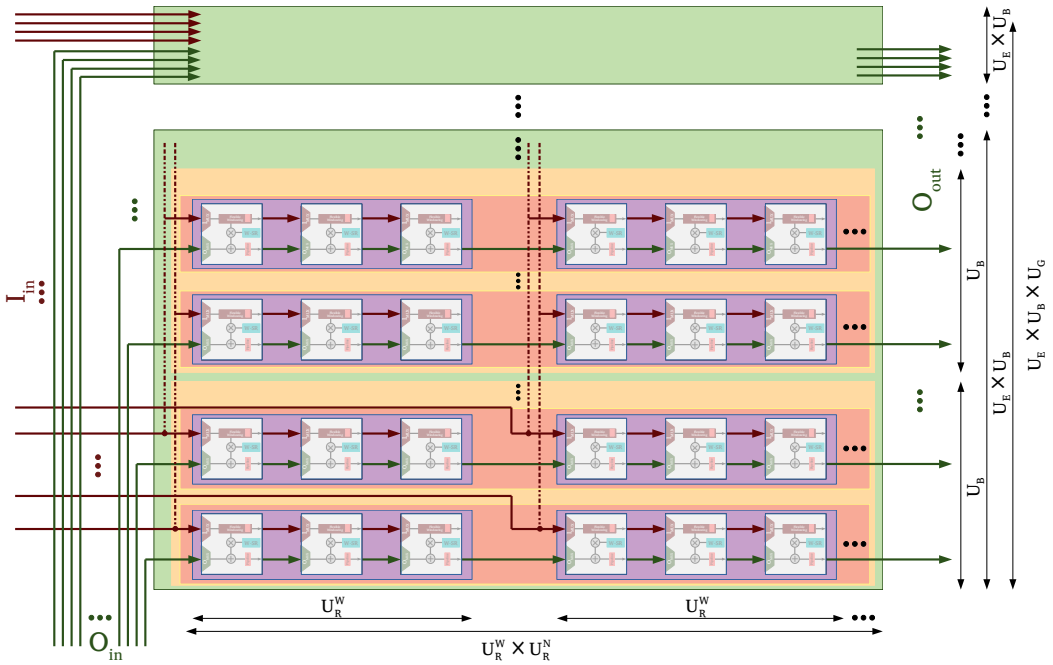


Fig. 8. Parameterized flexible systolic architecture for a configuration expressed by Notation 7

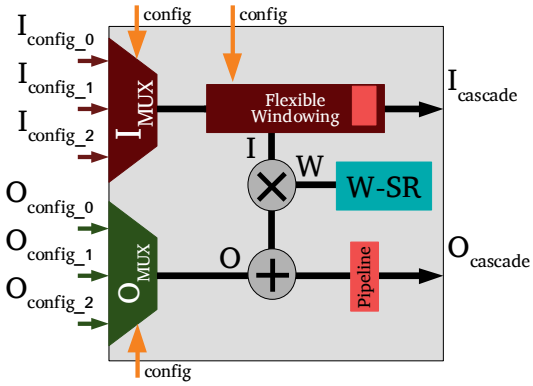
The other three parameters, U_E, U_B, and U_G, define the number of copies of the dot-product circuits along three different dimensions. The inputs and outputs to/from each copy are different except for U_E and U_B directions where **I** and **W** signals are shared by broadcasting along those directions. In our architecture, MAC units have dedicated **W** memories. Thus they do not share the **W** values in the U_B direction (orange areas).

Our MAC unit is illustrated in Figure 9a. It computes $O_{\text{cascade}} = I \times W + O$. I and O signals come from separate multiplexing circuits which are designed to provide different signal sources including other MAC units and block ports per configuration. These allow rearrangement of the MAC units for implementing different projections as configurations. This flexibility is run-time controllable by dedicated mode signals.

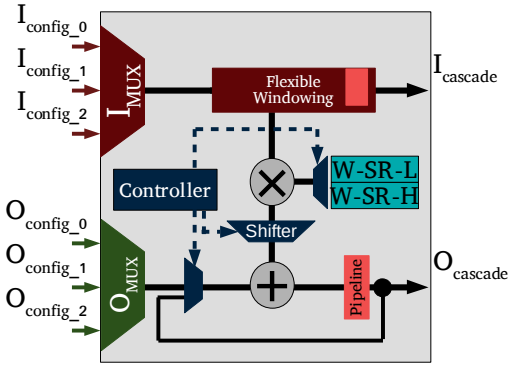
The I signal is also registered in the flexible windowing circuit and this circuit can handle windowing for different W_{stride} and W_{buffer}. Its design supports different W_{buffer} and W_{stride} across all desired configurations. An example of this circuit which supports $W_{\text{stride}} = 1$ or 2 and $W_{\text{buffer}} \leq 3$ is depicted in Figure 9c. This generates I_{cascade}, which should be connected to the next PE where windowing is required. Registering I prevents excessive propagation delay. To store the **W** values, each MAC unit has a shift register. Finally, O_{cascade} carries partial results. To maximize the frequency of a MAC computation, this signal is also registered. The corresponding register for the I_{cascade} signal is augmented with a windowing circuit.

By cascading the O signals of N MAC units, an N-point dot-product computation is implemented. By including the cascade of I signals, the dot product computations with input windowing is achievable. Figure 9d shows the cascading arrangement for $N = 3$.

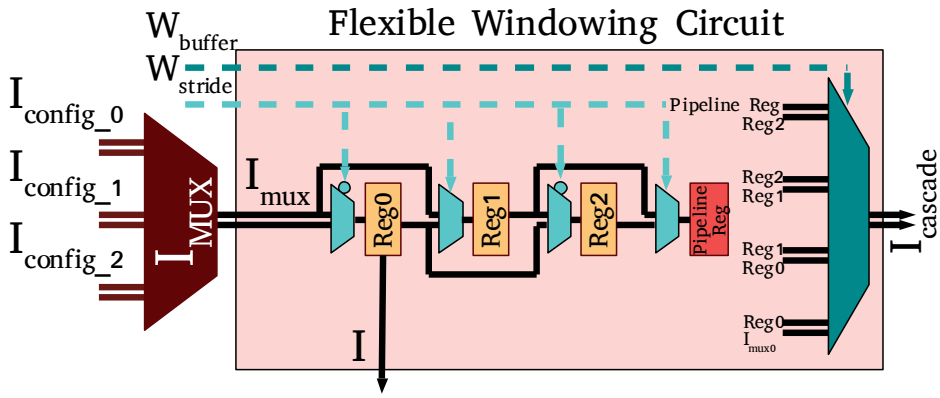
Figure 9b shows the modifications of the MAC unit to support higher-precision computations through serial multiplication. We include a deeper shift register for saving the higher and lower



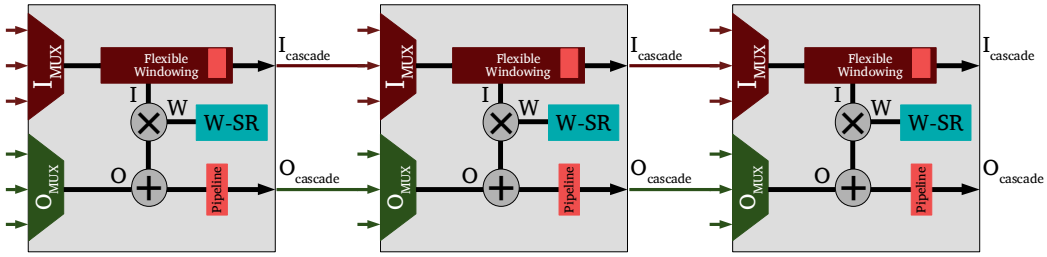
(a) A MAC unit



(b) A serial multiplier-armed MAC unit



(c) Flexible windowing circuit



(d) Cascading MAC units using I_{out} and Res_{out} signals

Fig. 9. A MAC unit and cascading technique

parts of the weight, multiplexing, shifting circuit and an extra multiplexer for reusing the partial result. These circuits require a small sequencer which is compile-time configurable and includes a run-time enable signal.

The operation of MAC units with high precision support is slightly different compared to the baseline version. Considering P_I and P_W as the input precisions, this type of MAC unit can compute

$P_I \times P_W$, $P_I \times 2P_W$, $2P_W \times P_W$, or $2P_W \times 2P_W$ MAC operations in 1, 2, 2, or 4 cycles respectively. This means the MLBlock's computation pipeline delivers new output in 1, 2, 2, or 4 cycles. On the other hand, loading the deeper weight shift register also requires 2× increase in bandwidth or the loading latency.

We should note that this architecture does not practically scale with the number of MAC units and configurations due to implementation costs. As the focus of this work is on FPGA EBs with area approximately the same as a DSP48E2, the number of MACs are limited and only a small number of configurations are required. This also raises the necessity for smart configuration selection. Fortunately, many routing resources are shared between different configurations, i.e, wide partial result signals which are always connected to the previous MAC unit. This leads to significant optimization opportunities for the synthesis process.

4.2 MLBlock-M Optimizations

In addition to the introduced general block designing procedures, we applied some optimizations to make the MLBlocks a practical solution as an FPGA EB. In our models, an MLBlock has a configuration signal mode to change between configurations at run-time. This also controls the block output multiplexers.

To place MLBlocks on FPGA architectures, we suggest a DSP-like columnar placement. Similar to DSP dedicated cascading routing between two adjacent MLBlocks for **O** signals is possible. This offers dot-product computation expansion using multiple MLBlocks. We removed input port for $\mathbf{O}_{\text{in}}$ while providing internal multiplexer to select between the cascaded $\mathbf{O}_{\text{cascade}}$ signal from the previous MLBlock and constant zero. The main reason behind this is managing IO requirements where, in practice, MLBlocks will be instantiated in the cascade mode and on average, these very wide inputs are not used.

Efficient distributed interface to/from memory blocks, the same for DSP, is imaginable as depicted in Figure 10 for MLBlock-12. This enables efficient cascading of the MLBlock computations where partial results are passed to the next MLBlock via dedicated cascade interconnections. It is crucial as using FPGA fabric for high-precision partial result signals pose significant routing challenges.

Initially, we assume a MAC unit comprises 8×8-bit multiplication followed by 32-bit accumulation. Using a serial multiplier configuration, we expand the supported precisions to the three cases of 8×16, 16×8, and 16×16 multiplication followed by 32-bit accumulation.

To define IO constraints, we used the Xilinx FPGA Ultrascale+ architecture as a model. Since MLBlock is a (partial) replacement for DSP blocks, we select constraints similar to a DSP block. Although the memory to DSP block ratio varies among the different part numbers, we assume a $1:1$ ratio of DSP48E2 to BRAM18 blocks, and use the same ratio for MLBlock and BRAM18. Recently, Samajdar et al. [8] showed how dedicated cascade interconnections of BRAM18s could be used to implement an efficient 18-bit streaming FIFO which is matched to DSP requirements. In contrast, MLBlocks require higher bandwidth. As tested using Vivado 2018.3, a 36-bit FIFO implementation by a single BRAM18 and custom FIFO controller circuitry using LEs is achievable working with highest routing fabric clock rate. Thus, we limit the bandwidth for **I** signals to 36-bits. Similar to [8], to stream in the **W** signals, we use URAMs which can provide 8-bit signals per DSP/MLBlock [8] (one 72-bit width URAM per 9 DSPs). Since DSP48E2 have more than 171 input and output data signals, we limit the $\mathbf{O}_{\mathbf{out}}$ signal up to 4×32 bits to maintain the same range number of data signal IOs. Note, we do not consider a limit for input O, as it will either enter the MLBlocks using the cascade routing or it would be constant zero.

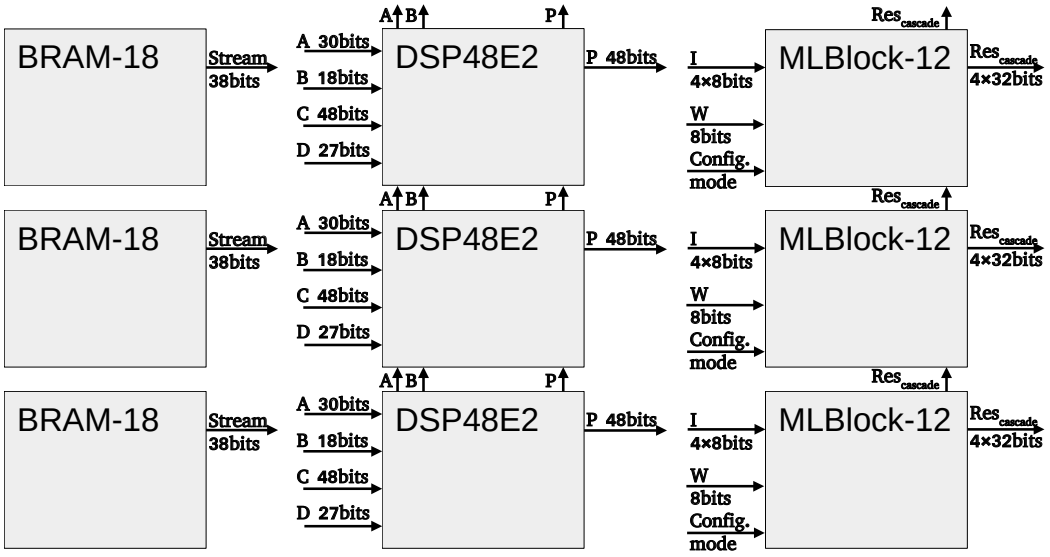


Fig. 10. Columnar placement of BRAMs, DSP48E2, and MLBlock-12

5 EXPERIMENTS AND RESULTS

In this section, we first demonstrate configuration selection. Next, the performance of MLBlock architectures over the Xilinx DSP48E2 for low-precision arithmetic is presented. Then, we integrate the serial multiplication technique with MLBlocks and evaluate performance for both high and low-precision computation. Next, we study MLBlock architectures as an overlay. Afterwards, we study the performance enhancement of replacing DSP48E2 blocks with MLBlock instances for Xilinx Ultrascale+ architecture in details, considering limitation on the number of EBs and the data scheduling overheads. Finally, we compare our MLBlock instances with other solutions for both Xilinx and Intel architectures using well-known CNN benchmarks.

For evaluation, we use the Baidu DeepBench benchmark [30], a collection of computation kernels from three categories of CNNs, RNNs, and GEMMs selected from real image detection, voice recognition, and text processing applications. Both inference and training tasks are represented. Table 1 lists the selected kernels and their loop variable iteration limit and stride. We report the results in three ways: averaged over all benchmark instances, averaged over the instances of each group (GEMM, CNN, RNN), and performance of individual instances (instance names are used individually).

We synthesized MLBlocks using Encounter(R) RTL Compiler RC14.11 targeting STMicro 28nm technology, with 750 MHz as the synthesis clock frequency target. This is the maximum practical frequency for Xilinx DSP48E1 on Virtex-7 using the same technology node [31]. Please note, we use post-synthesis results to report area in um^2 and power in mW directly according to the reports. The DSP48E2 Verilog model was an open source version available through [15].

5.1 Configuration selection methods

Table 2 summarizes results for MLBlock-12 using different design exploration techniques. The greedy approach chooses four configurations leading to higher area while delivering the highest utilization rate. In contrast, the N-Config approach maximises compute density ($\frac{\text{Utilization}}{\text{Area}}$) as shown in column Obj. The N-Config approach requires synthesis to be executed within its search loop and

Table 1. Loop variable iteration limit and stride, (limit, stride), for selected kernels from DeepBench [30]. (GEMMs: [row,column]×[row,column], CNNs: $B[X,Y,C]*K[F_X,F_Y,C]$, RNNs: (Hidden layer size, batch size))

Group	ID	Details	b_0	b_1	b_2	e_0	r_0	r_1	r_2
GEMM	0	[1760,1760]×[1760,128]	(1760,1)	-	-	(128,1)	-	-	(1760,1)
	1	[7860,2560]×[2560,64]	(7860,1)	-	-	(64,1)	-	-	(2560,1)
	2	[2560,2560]×[2560,64]	(2560,1)	-	-	(64,1)	-	-	(2560,1)
	3	[5124,2560]×[2560,9124]	(5124,1)	-	-	(9124,1)	-	-	(2560,1)
	4	[3072,1024]×[1024,128]	(3072,1)	-	-	(128,1)	-	-	(1024,1)
	5	[5124,2048]×[2048,700]	(5124,1)	-	-	(700,1)	-	-	(2048,1)
	6	[35,2048]×[2048,700]	(35,1)	-	-	(700,1)	-	-	(2048,1)
	7	[3072,1024]×[1024,3000]	(3072,1)	-	-	(3000,1)	-	-	(1024,1)
	8	[512,2816]×[2816,6000]	(512,1)	-	-	(6000,1)	-	-	(2816,1)
	9	[7680,2560]×[2560,1]	(7680,1)	-	-	(1,1)	-	-	(2560,1)
	10	[7680,2560]×[2560,2]	(7680,1)	-	-	(2,1)	-	-	(2560,1)
	11	[7680,2560]×[2560,1500]	(7680,1)	-	-	(1500,1)	-	-	(2560,1)
	12	[10752,3584]×[3584,1]	(10752,1)	-	-	(1,1)	-	-	(3584,1)
	13	[5124,2048]×[2048,700]	(5124,1)	-	-	(700,1)	-	-	(2048,1)
	14	[35,2048]×[2048,700]	(35,1)	-	-	(700,1)	-	-	(2048,1)
	15	[3072,1024]×[1024,1500]	(3072,1)	-	-	(1500,1)	-	-	(1024,1)
	16	[7680,2560]×[2560,1]	(7680,1)	-	-	(1,1)	-	-	(2560,1)
	17	[7680,2560]×[2560,1500]	(7680,1)	-	-	(1500,1)	-	-	(2560,1)
	18	[7680,2560]×[2560,1]	(7680,1)	-	-	(1,1)	-	-	(2560,1)
CNN	0	32[700,161,1]*32[5,20,1]	(700,2)	(161,2)	(32,1)	(32,1)	(5,1)	(20,1)	(1,1)
	1	8[54,54,64]*64[3,3,64]	(54,1)	(54,1)	(8,1)	(64,1)	(3,1)	(3,1)	(64,1)
	2	16[224,224,3]*64[3,3,3]	(224,1)	(224,1)	(16,1)	(64,1)	(3,1)	(3,1)	(3,1)
	3	16[7,7,512]*512[3,3,512]	(7,1)	(7,1)	(16,1)	(512,1)	(3,1)	(3,1)	(512,1)
	4	16[28,28,192]*32[5,5,192]	(28,1)	(28,1)	(16,1)	(32,1)	(5,1)	(5,1)	(192,1)
	5	4[341,79,32]*32[5,10,32]	(341,2)	(79,2)	(4,1)	(32,1)	(5,1)	(10,1)	(32,1)
	6	1[224,224,3]*64[7,7,3]	(224,2)	(224,2)	(1,1)	(64,1)	(7,1)	(7,1)	(3,1)
	7	1[56,56,256]*128[1,1,256]	(56,2)	(56,2)	(1,1)	(128,1)	(1,1)	(1,1)	(256,1)
	8	2[7,7,512]*2048[1,1,512]	(7,1)	(7,1)	(2,1)	(2048,1)	(1,1)	(1,1)	(512,1)
	9	1[112,112,64]*64[1,1,64]	(112,1)	(112,1)	(1,1)	(64,1)	(1,1)	(1,1)	(64,1)
	10	1[56,56,256]*128[1,1,256]	(56,2)	(56,2)	(1,1)	(128,1)	(1,1)	(1,1)	(256,1)
	11	1[7,7,512]*2048[1,1,512]	(7,1)	(7,1)	(1,1)	(2048,1)	(1,1)	(1,1)	(512,1)
RNN	0	RNN (1760, 16)	(1760,1)	-	(16,1)	(1760,1)	-	-	(3520,1)
	1	RNN (2560, 32)	(2560,1)	-	(32,1)	(2560,1)	-	-	(5120,1)
	2	LSTM (1024, 128)	(4096,1)	-	(128,1)	(1024,1)	-	-	(2048,1)
	3	GRU (2816, 32)	(8448,1)	-	(32,1)	(2816,1)	-	-	(5632,1)
	4	LSTM (1536, 4)	(6144,1)	-	(4,1)	(1536,1)	-	-	(3072,1)
	5	LSTM (256, 4)	(1024,1)	-	(4,1)	(256,1)	-	-	(512,1)
	6	GRU (2816, 1)	(8448,1)	-	(1,1)	(2816,1)	-	-	(5632,1)
	7	GRU (2560, 2)	(7680,1)	-	(2,1)	(2560,1)	-	-	(5120,1)

is therefore time consuming. However, calculating utilization rate is quick. We comment that as reported in Table 2 the utilization rate plateaus for $N \geq 4$, but at the same time, in general supporting more complex configurations results in a higher implementation cost, so little improvement is

Table 2. Selection methods for MLBlock-12 (results based on average over all benchmark cases)

Method	Utilization	Area†	Obj‡	# Synth.	time
Greedy	88.241	6093	1	1	1–2 mins
1-Config	72.000	5243	0.94	28	3–4 mins
2-Config	86.019	5245	1.13	378	1–2 hours
3-Config	88.192	5634	1.08	3276	≈ a day
4-Config	88.241§	-	-	20475	≈ a week
5-Config	88.241§	-	-	98280	≈ a month

†um^2, ‡normalized objective: $\frac{Utilization}{Area}$

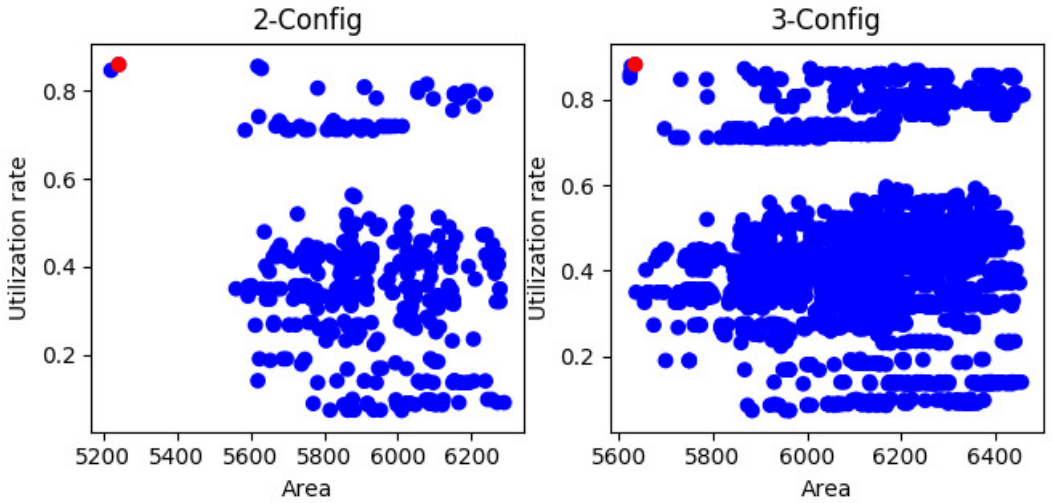


Fig. 11. Search space for MLBlock-12 using 2 and 3-Config technique (results based on average over all benchmark cases). The most efficient architecture is pointed by red (Area: um^2).

expected. The search space of 2/3-Config is also shown in Figure 11. The wide range of utilization rate is due to a mismatch in supported stride or unrolling of the selected projection and benchmarks. The final compute density (utilization/area) can vary by up to two orders of magnitude, highlighting the need for automated design space exploration. Although 2-Config delivers the best performing MLBlock for our benchmark, for the remaining experiments we used the Greedy method for its speed. The resource utilization range is also larger for Greedy since supporting more configurations requires more routing and buffering resources.

5.2 MLBlocks vs. DSP48E2

Using the same IO constraints as DSP48E2, we generate MLBlocks for different numbers of MAC units and results are summarized in Table 3. We normalized the area and power values based on the results for DSP48E2 model, where the absolute numbers are 7208.87 um^2 and 19.516 mW, respectively. From the table, it can be seen that in an MLBlock-M, area and power scale linearly with M. All 8×8 precision configurations have significantly lower power and area than the DSP48E2.

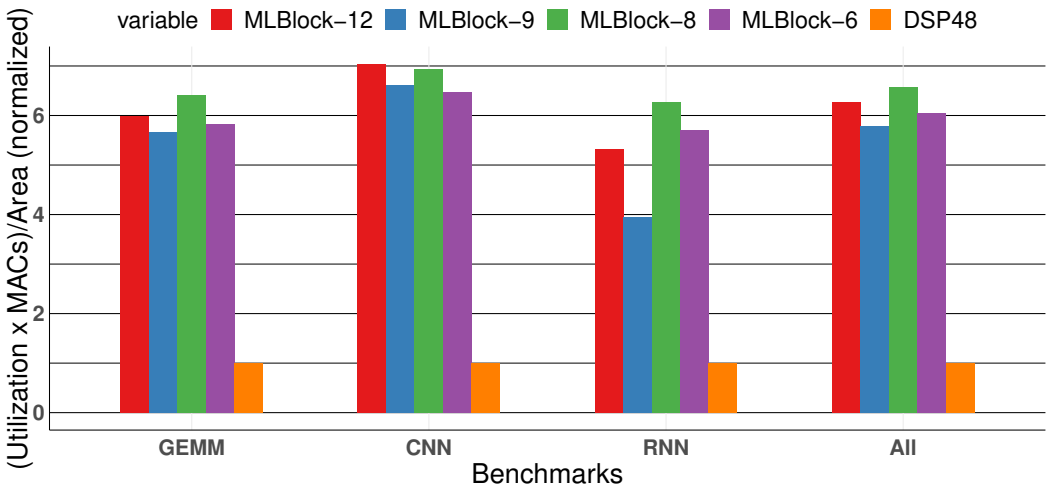


Fig. 12. Compute density of various MLBlocks for low precision computations (results based on average for each benchmark category and over all benchmark cases)

It is also important to note that the MLBlock-M can perform $M/2$ times more MACs per cycle at 8-bit precision.

Compute density (or performance per area) can be calculated as $\frac{\text{Utilization}\times\text{MACs}}{\text{Area}}$. Figure 12 shows this metric for the different benchmark classes. The performance of MLBlocks can be seen to be around 6× higher than DSP48E2 for 8-bit multiply and 32-bit accumulate operations. This is because the MLBlock architecture can provide more MACs without increasing routing. Also, DSP48E2 includes some unneeded circuits such as a 48-bit comparator, wide logic functions, pre-adder, and wider accumulator. It is notable that MLBlock-12 is also 15% smaller than a DSP48E2. To be fair, these extra features are required for other applications not considered in this study and replacing a percentage of DSP48E2s with MLBlocks could achieve a compromise.

Figure 12, also shows how the number of MAC units affects the efficiency. MLBlock-12 is a particularly good choice as 12 it is divisible by 2, 3, 4 and 6, making the mapping of different loop counts more efficient. In general, choosing an M with a diverse set of factors leads to a higher utilization rate. Thus, MLBlock-9 shows a lower performance in general and is particularly bad for the LSTM cases; its excellent performance in CNN benchmarks is due to the fact that 3 is a common window size.

The other aspect of MLBlocks is IO efficiency. Physically this corresponds to providing local connections and run-time configuration flexibility inside the block, instead of relying on FPGA fabric and LE-based multiplexing. For instance, MLBlock-12 requires 24-bits data signals per MACs while this number is 93 for the DSP48E2. In both cases, the computations themselves are still dot-product computations. The key to MLBlock's efficiency lies in avoiding global routing by providing internal broadcasting or using windowing.

5.3 MLBlocks with High-precision support

Adding some additional circuitry to the 8×8 bit multiplier can enable its use as a serial-parallel multiplier to achieve higher precision. We now describe a high precision MLBlock which supports $8/16 \times 8/16 + 32$-bit MAC operations. The effect of this change on area, utilization and power

Table 3. Post-synthesis results for different EB architectures

EB name	Precisions	Utilization	Area*	Power*
DSP48E2	27×18 or two 8×8	≈ 1	1	1
MLBlock-12	8×8	88%	0.85	0.98
MLBlock-9	8×8	86%	0.66	0.81
MLBlock-8	8×8	92%	0.56	0.65
MLBlock-6	8×8	93%	0.46	0.54
MLBlock-12	$(16/8) \times (16/8)$	88%	1.44	1.81
MLBlock-9	$(16/8) \times (16/8)$	86%	1.20	1.57
MLBlock-8	$(16/8) \times (16/8)$	92%	0.96	1.20
MLBlock-6	$(16/8) \times (16/8)$	93%	0.80	1.02

*normalized

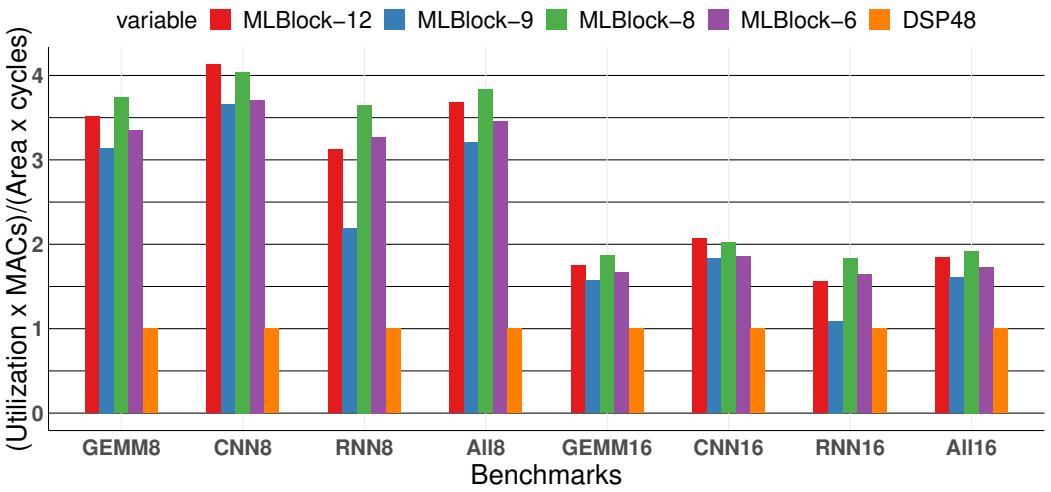


Fig. 13. Compute density of various MLBlocks using serial-parallel multipliers for high/low precision computations (results based on average for each benchmark category and over all benchmark cases)

is shown in the bottom half of Table 3. As expected, the area increases and performance of low-precision arithmetic is reduced to around 3.5× since multiplication becomes a multi-cycle operation. Figure 13 shows the compute density of different high precision MLBlocks. While adding high precision support necessarily decreases the performance advantage and increases area, the overall compute density is still a significant improvement over DSP482E. Indeed, MLBlock-8 is 4% smaller than DSP48E2 while having approximately 3× higher compute density for 8-bit operations and 2× higher compute density for 16-bit operations.

Table 4. Post implementation resource and timing results for different EB architectures as overlays

EB name	Precision	CLBs	Registers	f_{Max} (MHz)	Performance/CLB
DSP48E2	27×18	338	564	154	0.91
MLBlock-12	8×8	324	649	221	7.20
MLBlock-9	8×8	243	650	251	7.99
MLBlock-8	8×8	187	553	222	**8.74**
MLBlock-6	8×8	160	386	223	7.78
MLBlock-12	$(16/8) \times (16/8)$	605	1054	223	3.89
MLBlock-9	$(16/8) \times (16/8)$	419	941	211	3.90
MLBlock-8	$(16/8) \times (16/8)$	383	674	225	**4.32**
MLBlock-6	$(16/8) \times (16/8)$	291	606	197	3.78
8×8 MAC (32-bit acc.)	8×8	19	80	378	19.89

5.4 MLBlocks as overlays

MLBlock models can also be used as soft IP cores which utilize FPGA logic elements. Although these models are not optimized for look-up table based implementations, studying them as overlays brings insight into the MLBlock architecture and its variations.

Table 4 summarizes the required FPGA resources for implementing MLBlock-12, 9, 8, 6 with and without high precision support. We report post-implementation results targeting the Virtex UltraScale+ architecture (part number xcvu5p-flva2104-1-i).

Considering the resource utilization of a single-cycle signed 8×8 multiply and 32-bit accumulation unit, we calculate the resource overhead due to the additional routing for MLBlock-M (without high precision support) by $\frac{\text{CLB}_{\text{MLBlock-}M} - M \times \text{CLB}_{8\times8\text{MAC}}}{M \times \text{CLB}_{8\times8\text{MAC}}}$, where $\text{CLB}_{\text{MLBlock-}M}$ and $\text{CLB}_{8\times8\text{MAC}}$ are the CLB costs for MLBlock-M and the single-cycle signed 8×8 MAC unit, respectively. Using that, MLBlock models (without high precision support) introduce 37% resource overhead on average. This means MLBlocks as EBs encapsulate significant run-time configurable routing resources, it could subsequently reduce routing pressure in deployments where run-time reconfiguration is required.

In addition, using the utilization rates from Table 3, we calculate performance per CLB as $\frac{\text{MACs} \times \text{Utilization} \times F_{Max}}{\text{CLB}}$. MLBlock-8 delivers the highest score, due to 1) high utilization rate over the benchmarks, and 2) an efficient configuration set. This is the same conclusion as for the standard cell synthesis results.

5.5 Performance analysis considering data scheduling

Increasing compute density does not necessarily translate to higher performance. Utilizing the added compute power relies on efficient data scheduling and becomes crucial when the number of EBs are limited and data movement overheads are significant. In MLBlocks, there is a narrow port for streaming in one of the inputs, which means an MLBlock-M requires M cycles to be (re)initialized. The pipeline registers within MAC units causes a number of cycles of delay before the first output appears. This latency may vary between the configurations of an MLBlock.

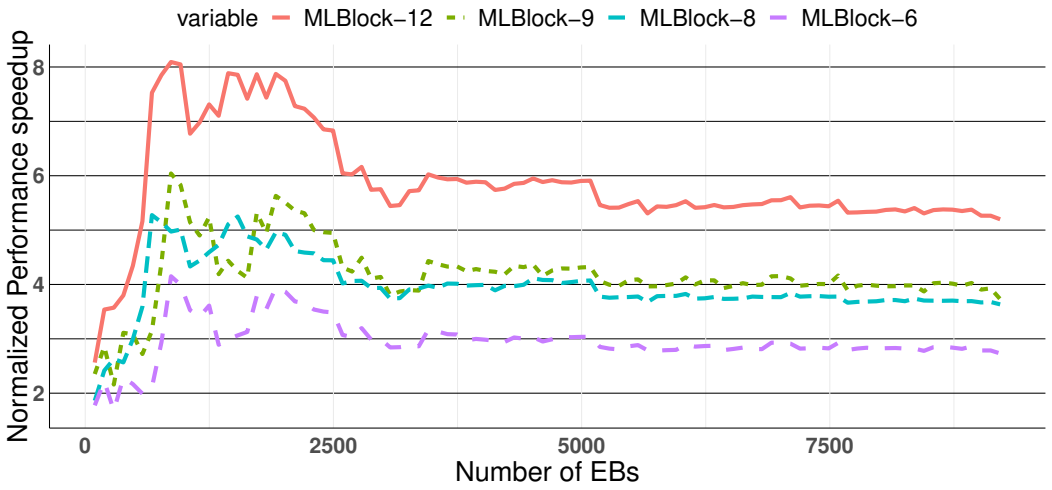


Fig. 14. Performance speedup of various MLBlock instances comparing to DSP48E2 for various budget for number of available EBs with narrowing the EB column's width

To explore the effect of data scheduling, we developed an open source Verilog generator for circuits expressed as nested loops according to Algorithm 1[1]. The generator instantiates the data path (EB instances, memories interconnections and memories) together with a corresponding state machine to implement the algorithm. The design space exploration is constrained by selected data flow, the number of EBs, memory components, and EB interfaces. We select a weight stationary data flow for this experiment. Assuming Ultrascale+ architecture, BRAM and URAM blocks are used to define the memory components. The required clock cycles is the optimization objective as a proxy for performance. This considers both 1) (re)initialization, and 2) the computation pipeline latencies for MLBlocks. We modelled an MLBlock by describing each compute projection as a separate logical EB. The best performing projection represents the MLBlock performance for the given input algorithm.

Number of EBs vs performance: Generally, weight stationary data flow works better when compute units have sufficient memory to buffer the fetched kernel parameters and avoid unnecessary data transfers. Increasing the number of EBs or their embedded memories leads to scheduling schemes with higher data reuse. Figure 14 shows the normalized speedup for different MLBlock instances comparing to the implementations with the same number of the DSP48E2 blocks. We swept the EB budget through the multiples of 96, where 96 is the common number of DSPs in a DSP column for this FPGA family. In this experiment, we again used the DeepBench benchmarks.

We first replace each DSP48E2 block with an MLBlock-M without changing the EB column size. This is possible as MLBlock instances are all smaller than the DSP48E2. Hence, with the same area footprint, MLBlock-12, 9, 8, and 6, are 6, 4.5, 4, 3× denser than the DSP48E2 for low-precision arithmetic.

Generally, the performance gain is expected to be the same as the compute density for an MLBlock-M architecture. However, for small numbers of EBs/DSP48E2s, the MLBlock-M architecture struggles with the high number of costly data transfers. Although MLBlock-M offers M× larger memories suitable for reusing a larger portion of the kernels, it also requires M× longer

[1]A separate paper describing this work has been submitted separately to the FPT'21 ACM TRETS journal-track

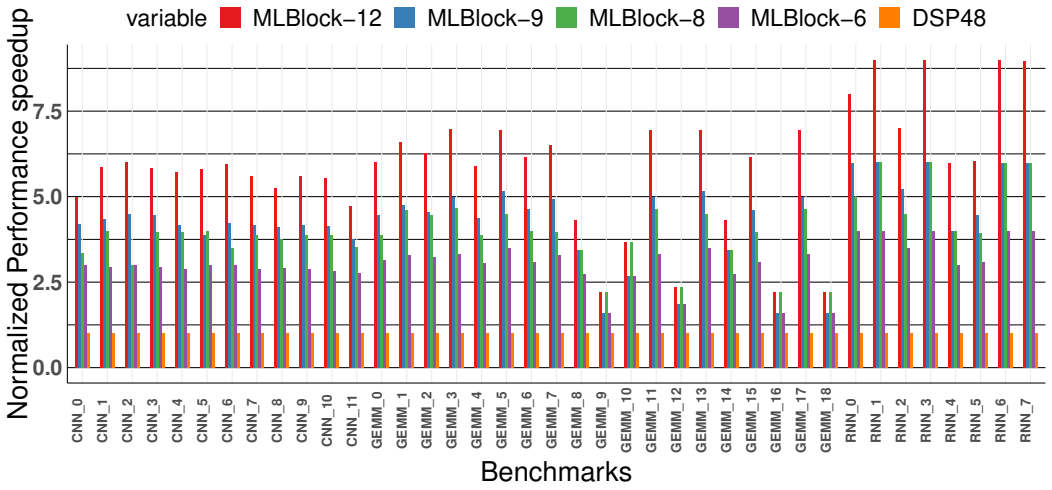


Fig. 15. Performance speedup of various MLBlock instances comparing to DSP48E2 for fixed budget of 4800 EBs

(re)initialization phases compared to their DSP48E2 counterpart. This trade-off restricts the performance speedup of MLBlocks to about 2-3×.

As the number of EBs/DSP48E2s are increased, the performance speedup for both MLBlocks and DSP48E2 columns improve. However, due to the higher memory and compute power density, the MLBlock architecture avoids unnecessary data movements, leading to higher performance. In fact, the growth surpasses the performance speedup expectations because of benchmark instances where the MLBlock memories are sufficient to save and reuse the kernel parameters, while DSP48E2 must incur the overhead of extra data transfers. Eventually both architectures reach the point that weights can be stored on the compute unit memories without excessive data movements and overheads. As can be seen in Figure 14, the performance speedups converge to the compute density ratios.

Figure 15 displays the speedups for each benchmark case, assuming the fixed budget of 4800 EBs. As expected, the average speedups should be consistent with the abovementioned computation density rates. Somewhat surprisingly, it varies among the benchmark groups and their members. For CNNs, the speedups are consistently close to the computation density ratios. However, for GEMMs, this is not always the case as some kernels have very low reuse factors or extremely narrow dimensions, e.g. GEMM-9, 10, 12, 16, 18. In contrast, due to the very large matrix multiplications in RNNs, there are some cases (e.g. RNN-1, 3, 6, 7) in which MLBlocks surpass the nominal speedup rates. This is because of their larger memories to save and reuse weights, which leads to a higher utilization rate while having more MAC units.

If narrowing the EB column's width is allowed, a small reduction in the FPGA footprint is expected. As given in reference [32], the DSP blocks contribute to only 5% of the total FPGA fabric. Using the estimated area ratios, the total FPGA chip area reduction is in the range of 1-2.7%, which slightly enhance the results on Figure 14 without affecting the trends.

It may also be possible to use the saved area to instantiate more MLBlocks in a column. Figure 16 shows the performance comparison where each DSP48E2 column is replaced by an MLBlock column with 122, 145, 171, or 208 instances of MLBlock-12,-9,-8, or -6 respectively. As expected, similar to Figure 14, initially, the number of EBs limits the performance to the range of 2.5-5×. Since each

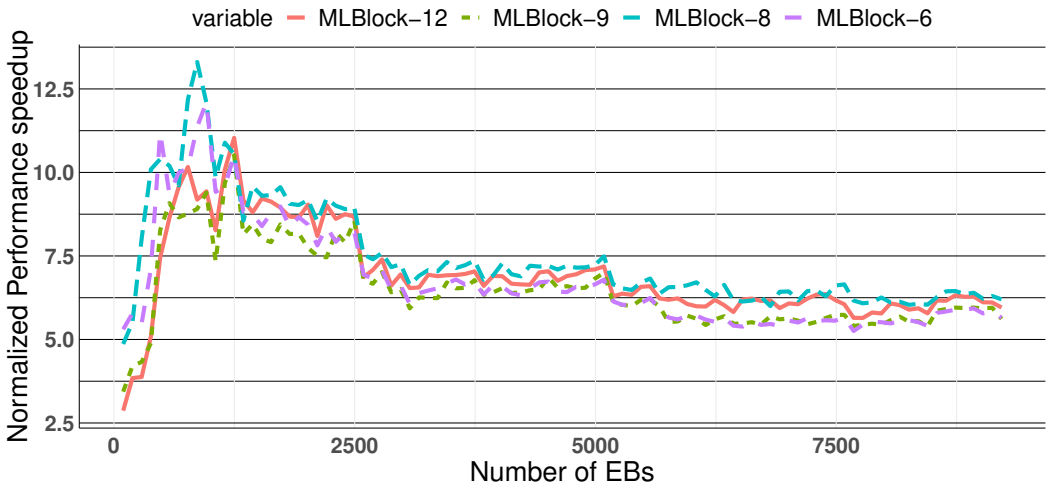


Fig. 16. Performance speedup of various MLBlock instances comparing to DSP48E2 for various budget for number of available EBs without narrowing the EB column's width

column includes more EBs; increasing the number of columns enhances the performance at higher rates. Thus, the same trend with higher performance gains is achieved. Finally, the performance speedups converge to the compute density rates, which calculated by Table 12. Note, in this setting, each MLBlock column may require more than a column of BRAM support as the 1:1 BRAM-MLBlock ratio is not preserved. Consequently, denser memory units are required to achieve such gains.

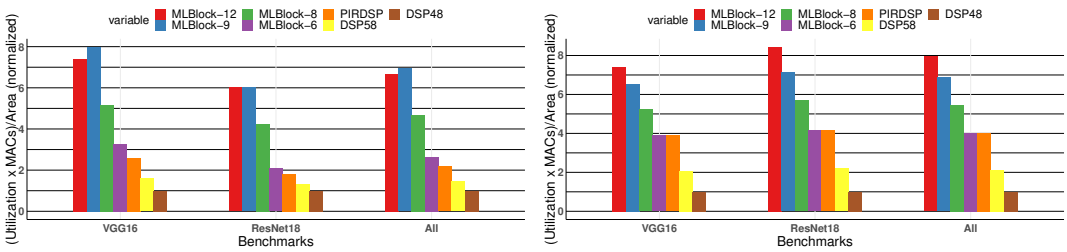
5.6 Comparison with Commercial Devices

5.6.1 Xilinx architecture: To demonstrate the effectiveness of the suggested design methodology, we first compare MLBlocks with other expert-designed alternatives for DSP48E2 from industry (Xilinx DSP58) and academia (PIR-DSP[15]). We use a new benchmark comprising computation kernels of convolution and fully-connected layers for two well known CNNs (VGG16 [33] and ResNet18 [34]) (batch size 2), while keeping the same MLBlocks as for the DeepBench [30] kernels.

The plots in Figure 17 compare the average speedup of VGG16 and ResNet18 kernels using MLBlocks, PIR-DSP, DSP58, and DSP48E2, assuming a fixed budget of 360 (Figure 17a) and 4272 (Figure 17b) EBs. These represent small embedded (Ultra 96v2 [35]) and large, high-end (ZCU111 [36]) FPGA development board specifications respectively.

As expected, similar to Figure 14, MLBlock modes achieve higher speedups for low precision arithmetic in both scenarios. In the embedded scenario, as the benchmark set is heavily weighted with standard 2D convolution layers having 3×3 kernels, MLBlock-9 is a better fit for the computations and delivers the best performance. In contrast, by increasing the size of EBs (high-end scenario), the performance speedups converge to the compute density ratios, and MLBlock-12 achieves the best performance.

PIR-DSP [15] and MLBlock-6 both can do six low-precision (8×8-bit) MACs. Interestingly, one of the MLBlock-6 configurations exactly matches the low precision mode in PIR-DSP (two parallel 3-element dot-products). However, other configurations of MLBlock-6 result in more efficient computation tiling comparing to PIR-DSP. Furthermore, PIR-DSP's commitment to backward compatibility meant retaining extra circuits such as a large pre-adder, comparator, and wide logic circuits. Consequently, it has about 18% larger area than DSP48E2. In our comparison, we dismiss



(a) Considering Xilinx Zynq UltraScale+ MPSoC ZU3EG resources as an embedded FPGA, used in U96v2 [35]

(b) Considering Xilinx Zynq UltraScale+ XCZU28DR resources as a high-end FPGA, used in RFSoC development kit, ZCU111 [36]

Fig. 17. Performance comparison between MLBlocks and expert-designed replacement for Xilinx DSP48E2 from both academia and industry considering both embedded and high-end FPGAs

this area overhead. This gap is larger when the number of EBs are limited, as extra configurations help to boost the performance. By increasing the number of EBs, the speedups converge to compute density rates which are the same for both architectures.

DSP58, with three low-precision MAC operations, shows performance between PIR-DSP and DSP48E2 with 6 and 2 MACs, respectively. Similar to PIR-DSP, it has an area overhead due to backward compatibility with the DSP48E2. Since there is no published information available, we assume that similar DSP/CLB/BRAM ratios for both DSP48E2 and DSP58, and also assume the same area and working frequency for these two DSPs.

5.6.2 Intel architecture: The suggested design methodology can be applied to other FPGA architectures, such as Intel FPGAs. Although it is better to recreate MLBlock instances according to the area and IO constraints of the new architecture, for consistency we use our previously generated instances. We gathered implementation results for MLBlocks, Stratix-10 DSP, an alternative DSP proposed in [14], as well as Tensor Slice [26] all in Table 5. Since the other DSPs support high precision, they are compared with MLBlocks with high-precision support.

The last column of this table presents our comparison metric calculated as $\frac{\#\text{ of MACs}}{\text{Area}\times\text{Delay}}$ for both high and low-precision arithmetic, which considers both performance and implementation costs. MLBlocks (especially MLBlock-8 and 12) delivers the highest score for both high and low-precision. As expected, Tensor Slice, with its high number of MACs, is the next best design. To be fair, this architecture supports floating-point and element-wise operations, which are not included in MLBlocks. The next architecture is that of Boutros et. al. [14]. Although it is a suitable replacement for Intel DSPs, backward compatibility prevents it from offering a dense low-precision compute unit. However, in high-precision mode, its performance (similarly for Stratix-10 DSP) only halves whereas MLBlocks performance is reduced by a factor of 4. Nevertheless, while [14] narrows the gap for high-precision computation, MLBlocks are still obtain the best performance.

Due to the lack of public information, we were not able to compare our work with other commercial solutions such as the Achronix MLB72 [23] or AI-tensor from Intel [24]. We also note that the AI-tensor implementation is different to MLBlocks. For instance, AI-tensor uses a pipelined adder-tree structure whereas MLBlocks are based on systolic arrays. AI-tensor also includes a double buffering system to cover reloading latencies, which is not present in our architecture.

However, aside from these differences, the computation of these two units are special cases of our architecture, which our tool explores, based on a systolic array architecture. For instance, AI-tensor

Table 5. Implementation results comparison between MLBlocks, Intel Stratix-10 DSP and its alternatives

EB name	Precision	# of MACs		Area	Tech size	f_{max}	$\frac{\#\text{ of MACs}}{\text{Area}\times\text{Delay}}$ *	
		8×8	16×16	(um^2)	(nm)		8×8	16×16
Stratix-10 DSP[14]	27×27,18×18	2	1	8404	28	600	1	1
MLBlock-6	8 × 8	6	-	3315	28	750	9.5	-
MLBlock-8	8 × 8	8	-	4026	28	750	10.4	-
MLBlock-9	8 × 8	9	-	4789	28	750	9.9	-
MLBlock-12	8 × 8	12	-	6093	28	750	10.3	-
MLBlock-6	(16/8) × (16/8)	6	6 (4cyc.)	3315	28	750	5.5	2.7
MLBlock-8	(16/8) × (16/8)	8	8 (4cyc.)	4026	28	750	6.1	3.1
MLBlock-9	(16/8) × (16/8)	9	9 (4cyc.)	4789	28	750	5.5	2.7
MLBlock-12	(16/8) × (16/8)	12	12 (4cyc.)	6093	28	750	6.1	3.0
Boutros et al.[14]	27×27,18×18,9×9,4×4	4	2	8810	28	600	1.9	1.9
Tensor Slice[26]	FP16×FP16, 8 × 8	64	-	50032	22	371	3.3	-

*normalized

is able to compute three 10-element dot-products (one input is shared). Focusing only on the computation, we can describe this single computation by a couple of projections: 1) <(1,-,-),10,3,1,1> (one shared 10-element input I and three sets of 10-element weights), and 2) <(1,-,-),10,1,3,1> (10-element shared input W and three batches of 10-element of I). If the number of MAC units is 30, our tool automatically considers these two projections along with other choices such as <(1,-,-),3,10,1> and <(3,1,1),10,1,1>.

6 CONCLUSION

We proposed a novel methodology for designing coarse-grained embedded blocks for machine learning applications. The procedure is based on modeling the computations and providing a mapping to different configurations of a flexible architecture. Using a set of benchmarks applicable to text, voice and image processing applications, together with design constraints from a commercial Xilinx FPGA, different instances of our architecture are generated. Our results show that MLBlocks can provide a 6× improvement in compute density over the Xilinx DSP48E2 in the same technology for 8-bit arithmetic, without increasing port requirements. Our 16-bit configuration, which uses serial multipliers can achieve 2× more computation performance per area compared to the Xilinx DSP48E2. Our approach generalizes earlier work which either focused on the implementation of a single type of DNN, or creating an FPGA/ASIC generator for arbitrary DNNs; rather we discover an efficient embedded block which covers a representative set of algorithms represented by the benchmark set.

The proposed model assumes the algorithm only comprises MAC operations and there are no output data dependencies apart from accumulation. Future work will involve generalization to support other classes of problems such as digital signal processing and stencil computations. We will also work on adding improved input/output, double buffering, and various data stationary types to the current MLBlock architecture to expand the design search space. Similarly implementations on alternative architectures, such as pipelined adder-tree structure of the AI-tensor, could also be

added to the MLBlock framework. Also, a careful comparison of EB for low-precision arithmetic compared to implementations using fined-grained logic elements will be undertaken.

REFERENCES

[1] E. Wang, J. J. Davis, R. Zhao, H. Ng, X. Niu, W. Luk, P. Y. K. Cheung, and G. A. Constantinides, "Deep neural network approximation for custom hardware: Where we've been, where we're going," *ACM Comput. Surv.*, vol. 52, no. 2, pp. 40:1–40:39, 2019. [Online]. Available: https://doi.org/10.1145/3309551

[2] E. Nurvitadhi, G. Venkatesh, J. Sim, D. Marr, R. Huang, J. Ong Gee Hock, Y. T. Liew, K. Srivatsan, D. Moss, S. Subhaschandra, and G. Boudoukh, "Can FPGAs beat gpus in accelerating next-generation deep neural networks?" in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 5–14. [Online]. Available: https://doi.org/10.1145/3020078.3021740

[3] Z. Song, Z. Liu, and D. Wang, "Computation error analysis of block floating point arithmetic oriented convolution neural network accelerator design," in *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, S. A. McIlraith and K. Q. Weinberger, Eds. AAAI Press, 2018, pp. 816–823. [Online]. Available: https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16057

[4] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," *J. Mach. Learn. Res.*, vol. 18, no. 1, p. 6869–6898, Jan. 2017.

[5] G. Yang, T. Zhang, P. Kirichenko, J. Bai, A. G. Wilson, and C. D. Sa, "SWALP : Stochastic weight averaging in low precision training," in *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, ser. Proceedings of Machine Learning Research, K. Chaudhuri and R. Salakhutdinov, Eds., vol. 97. PMLR, 2019, pp. 7015–7024. [Online]. Available: http://proceedings.mlr.press/v97/yang19d.html

[6] M. Langhammer, S. Gribok, and G. Baeckler, "High density pipelined 8bit multiplier systolic arrays for FPGA," in *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 322. [Online]. Available: https://doi.org/10.1145/3373087.3375352

[7] Xilinx, "Deep Learning with INT8 Optimization on Xilinx Devices - WP486 (v1.0.1)," 2017, https://www.xilinx.com/support/documentation/white_papers/wp486-deep-learning-int8.pdf.

[8] A. Samajdar, T. Garg, T. Krishna, and N. Kapre, "Scaling the cascades: Interconnect-aware FPGA implementation of machine learning problems," in *29th International Conference on Field Programmable Logic and Applications, FPL 2019, Barcelona, Spain, September 8-12, 2019*, 2019, pp. 342–349. [Online]. Available: https://doi.org/10.1109/FPL.2019.00061

[9] Nvidia, "DS-10184-001: NVIDIA Jetson Xavier NXSystem-on-Module," 2020.

[10] Xilinx, "XMP103: Product Selection Guide," 2021, https://www.xilinx.com/support/documentation/selection-guides/ultrascale-plus-fpga-product-selection-guide.pdf#VUSP.

[11] X. Yang, M. Gao, Q. Liu, J. Setter, J. Pu, A. Nayak, S. Bell, K. Cao, H. Ha, P. Raina, C. Kozyrakis, and M. Horowitz, "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*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 369–383. [Online]. Available: https://doi.org/10.1145/3373376.3378514

[12] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 161–170. [Online]. Available: https://doi.org/10.1145/2684746.2689060

[13] Intel, "SV51001 Stratix V Device Overview," https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/stratix-v/stx5_51001.pdf.

[14] A. Boutros, S. Yazdanshenas, and V. Betz, "Embracing diversity: Enhanced DSP blocks for low-precision deep learning on FPGAs," in *28th International Conference on Field Programmable Logic and Applications, FPL 2018, Dublin, Ireland, August 27-31, 2018*, 2018, pp. 35–42. [Online]. Available: https://doi.org/10.1109/FPL.2018.00014

[15] S. Rasoulinezhad, H. Zhou, L. Wang, and P. H. W. Leong, "PIR-DSP: an FPGA DSP block architecture for multi-precision deep neural networks," in *27th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2019, San Diego, CA, USA, April 28 - May 1, 2019*. IEEE, 2019, pp. 35–44. [Online]. Available: https://doi.org/10.1109/FCCM.2019.00015

[16] Intel, "Intel Agilex Variable Precision DSP Blocks User Guide," 2019, https://www.intel.com/content/dam/altera-www/global/en=US/pdfs/literature/hb/agilex/ug-ag-dsp.pdf.

[17] Xilinx, "Versal ACAP DSP Engine, Architecture Manual, AM004 (v1.1.2)," 2021, https://www.xilinx.com/support/documentation/architecture-manuals/am004-versal-dsp-engine.pdf.

[18] S. Rasoulinezhad, Siddhartha, H. Zhou, L. Wang, D. Boland, and P. H. W. Leong, "LUXOR: an FPGA logic cell architecture for efficient compressor tree implementations," in *FPGA '20: The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Seaside, CA, USA, February 23-25, 2020*, S. Neuendorffer and L. Shannon, Eds. ACM, 2020, pp. 161–171. [Online]. Available: https://doi.org/10.1145/3373087.3375303

[19] A. Boutros, M. Eldafrawy, S. Yazdanshenas, and V. Betz, "Math doesn't have to be hard: Logic block architectures to enhance low-precision multiply-accumulate on FPGAs," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2019, Seaside, CA, USA, February 24-26, 2019*, K. Bazargan and S. Neuendorffer, Eds. ACM, 2019, pp. 94–103. [Online]. Available: https://doi.org/10.1145/3289602.3293912

[20] M. Eldafrawy, A. Boutros, S. Yazdanshenas, and V. Betz, "FPGA logic block architectures for efficient deep learning inference," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 13, no. 3, Jun. 2020. [Online]. Available: https://doi.org/10.1145/3393668

[21] B. Gaide, D. Gaitonde, C. Ravishankar, and T. Bauer, "Xilinx adaptive compute acceleration platform: Versal^{tm} architecture," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2019, Seaside, CA, USA, February 24-26, 2019*, 2019, pp. 84–93. [Online]. Available: https://doi.org/10.1145/3289602.3293906

[22] E. Nurvitadhi, J. J. Cook, A. K. Mishra, D. Marr, K. Nealis, P. Colangelo, A. C. Ling, D. Capalija, U. Aydonat, S. Y. Shumarayev, and A. Dasu, "In-package domain-specific asics for intel® stratix® 10 FPGAs: A case study of accelerating deep learning using tensortile asic(abstract only)," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2018, Monterey, CA, USA, February 25-27, 2018*, 2018, p. 287. [Online]. Available: https://doi.org/10.1145/3174243.3174966

[23] Achronix - Data Acceleration, "Speedster7t IP Component Library User Guide (UG086)," https://www.achronix.com/sites/default/files/docs/Speedster7t_IP_Component_Library_User_Guide_UG086.pdf.

[24] A. Boutros, E. Nurvitadhi, R. Ma, S. Gribok, Z. Zhao, J. C. Hoe, V. Betz, and M. Langhammer, "Beyond peak performance: Comparing the real performance of ai-optimized fpgas and gpus," in *International Conference on Field-Programmable Technology, (IC)FPT 2020, Maui, HI, USA, December 9-11, 2020*. IEEE, 2020, pp. 10–19. [Online]. Available: https://doi.org/10.1109/ICFPT51103.2020.00011

[25] A. Aror, Z. Wei†, and L. K. John, "Hamamu: Specializing FPGAs for ML Applications by Adding Hard Matrix Multiplier Blocks," in *31th IEEE International Conference on Application-specific Systems, Architectures and Processors, ASAP*, 2020. [Online]. Available: http://lca.ece.utexas.edu/pubs/Hamamu___ASAP2020_Jun9.pdf

[26] A. Arora, S. Mehta, V. Betz, and L. K. John, "Tensor slices to the rescue: Supercharging ML acceleration on fpgas," in *FPGA '21: The 2021 ACM/SIGDA International Symposium on Field Programmable Gate Arrays, Virtual Event, USA, February 28 - March 2, 2021*, L. Shannon and M. Adler, Eds. ACM, 2021, pp. 23–33. [Online]. Available: https://doi.org/10.1145/3431920.3439282

[27] Y. Shi, X. Yao, R. Chen, L. Yuan, N. Xu, and X. Liu, "Image recognition based on multi-scale dilated lightweight network model," in *Proceedings of the 5th International Conference on Multimedia and Image Processing*, ser. ICMIP '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 43–48. [Online]. Available: https://doi.org/10.1145/3381271.3381300

[28] W. Sun, X. Zhou, X. Zhang, and X. He, "A lightweight neural network combining dilated convolution and depthwise separable convolution," in *Cloud Computing, Smart Grid and Innovative Frontiers in Telecommunications*, X. Zhang, G. Liu, M. Qiu, W. Xiang, and T. Huang, Eds. Cham: Springer International Publishing, 2020, pp. 210–220.

[29] M. Carreras, G. Deriu, L. Raffo, L. Benini, and P. Meloni, "Optimizing temporal convolutional network inference on FPGA-based accelerators," *CoRR*, vol. abs/2005.03775, 2020. [Online]. Available: https://arxiv.org/abs/2005.03775

[30] Baidu, "DeepBench," 2016, https://github.com/baidu-research/DeepBench.

[31] Xilinx, "DS183 (v1.28) - Virtex-7 T and XT FPGAs Data Sheet:DC and AC Switching Characteristics," 2019, https://www.xilinx.com/support/documentation/data_sheets/ds183_Virtex_7_Data_Sheet.pdf.

[32] M. Langhammer and B. Pasca, "Floating-point DSP block architecture for fpgas," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, February 22-24, 2015*, G. A. Constantinides and D. Chen, Eds. ACM, 2015, pp. 117–125. [Online]. Available: https://doi.org/10.1145/2684746.2689071

[33] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2015. [Online]. Available: http://arxiv.org/abs/1409.1556

[34] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*. IEEE Computer Society, 2016, pp. 770–778. [Online]. Available: https://doi.org/10.1109/CVPR.2016.90

[35] Avnet, "Ultra96-V2 Single Board Computer Hardware User's, Guide Version 1.3," 2021, https://www.avnet.com/wps/wcm/connect/onesite/b85b9556-0b2a-42b3-ad6a-8dcf3eac1ff9/Ultra96-V2-HW-User-Guide-v1_3.pdf?MOD=AJPERES&CACHEID=ROOTWORKSPACE.Z18_NA5A1I41L0ICD0ABNDMDDG0000-b85b9556-0b2a-42b3-ad6a-8dcf3eac1ff9-nDNP5R3.

[36] Xilinx, "ZCU111 Evaluation Board, User Guide, UG1271 (v1.2)," 2018, https://www.xilinx.com/support/documentation/boards_and_kits/zcu111/ug1271-zcu111-eval-bd.pdf.