2023 IEEE 35th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD) | 979-8-3503-0548-7/23/$31.00 ©2023 IEEE | DOI: 10.1109/SBAC-PAD59825.2023.00013

Exploiting the Potential of Flexible Processing Units

Mateo Vázquez, Muhammad Waqar Azhar, Pedro Trancoso
Department of Computer Science and Engineering
Chalmers University of Technology, Gothenburg, Sweden
{mateo.vazquezmaceiras, waqarm, ppedro}@chalmers.se

***Abstract*—In order to meet the increased computational demands and stricter power constraints of modern applications, architectures have evolved to include domain-specific accelerators. In order to design efficient accelerators, three main challenges need to be addressed: compute, memory, and control. Moreover, since SoCs usually contain multiple accelerators, selecting the right one for each task also become crucial. This becomes specially relevant in Flexible Processing Units (xPUs), processing units that provide multiple functionalities with the same hardware. While it is possible to use shared support components for all functionalities, this will lead to sub-optimal performance. In this work, we take one example of such xPU, and analyze the aspects which have not yet been fully addressed, showing that there is more potential to be exploited. By understanding the required memory patterns, we can achieve up to 72% speedup gains compared to using the memory support optimized for a different functionality. Furthermore, we propose an in-depth analysis of the different functionalities provided by the xPU. We then leverage the insights obtained from this analysis by providing a mechanism that selects the right functionality, maximizing hardware utilization.**

***Index Terms*—Flexible Processing Unit, Vector Unit, Systolic Array, GEMM, DNN, Scientific Computing**

I. Introduction

In recent years, the landscape of computer architecture has been characterized, among others, by diminishing returns from technology scaling and power density limitations. Additionally, application requirements are increasing faster than before [1]. To bridge that gap, processors are shifting from homogeneous multi-cores composed of general-purpose CPUs to heterogeneous System-on-Chip (SoC) designs. These SoCs integrate one or more Domain-Specific Accelerators (DSAs) coupled with the host CPU(s) [2]. This way, systems can offer the required performance while fulfilling the power budget. Consequently, multiple DSAs have been proposed, both in academia and in industry, targeting different application domains, such as Deep Neural Networks (DNNs) [3]–[9], graphs [10]–[12] and bioinformatics [13], [14].

When it comes to designing accelerators, one needs to address three main challenges: (1) compute, (2) memory, and (3) control. First, the compute units need to be efficiently implemented. Moreover, they need to be accompanied by an efficient data flow, that maximizes data reuse within the computational parts of the system. Second, the memory system needs to feed data at the required rate to the compute units, enabling maximum performance and utilization. To achieve this, the memory system must be designed around the data patterns required by the compute units and their dataflow. Third, there is a need for an efficient control of compute and memory to achieve maximum performance and efficiency. Furthermore, the system needs to provide a way to select the best matching accelerator to use. Otherwise, performance may be penalized due to inefficiencies, such as considerable resource under-utilization. Selecting the best matching accelerator can be trivial in cases where the differences between accelerators are clear. However, this becomes increasingly difficult if different accelerators can efficiently compute the same application or core kernel within the application. In this case, the point when one accelerator starts to outperform the other may not be clear. Thus, a more in-depth analysis is required. One example of this situation is the General Matrix Multiplication (GEMM). This kernel, key in multiple compute-intensive applications, such as Deep Neural Networks (DNN), can be efficiently computed by either Systolic Arrays (SAs) [7], [15] or Vector Processing Units (VPUs) [16]–[19]. While SAs are typically a better option, as they are explicitly designed to compute GEMM, performance can be worse than expected due to under-utilization of their compute units [20].

Nowadays, there are multiple systems, both in industry and academia, that combine SAs and VPUs in different ways: separate SA and VPU units such as in Google's TPU (v2 onwards) [21], or combined SA and VPU such as in the Vector-Systolic Architecture (VSA) [22]. VSA does not present two different architectures but rather a single heterogeneous architecture that can behave as both a VPU and an SA. To do so, the available resources in a given baseline VPU are reused to implement a SA with minimal hardware overhead. This way, it is not just a VPU or a Matrix Multiply Unit (MMU) (a name typically given to architectural components based on an SA for GEMM computation). Instead, it can be defined as a Flexible Processing Unit (xPU), which uses the same hardware to offer different functionalities. However, while [22] has addressed the computational part and extended the baseline VPU's control logic, less emphasis has been placed on both the memory system and the decision-making.

In this work, we will analyze these two missing points to further exploit the potential of such a xPU. In particular, this

This work was partially supported by the eProcessor project funded by the European High-Performance Computing Joint Undertaking (JU), grant agreement No 956702. The JU receives support from the EU H2020 research and innovation program and Spain, Sweden, Greece, Italy, France, and Germany. This work was also partially supported by the VEDLIoT project, which received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 957197 and the Swedish Foundation for Strategic Research (contract number CHI19-0048) under the PRIDE project.

2643-3001/23/$31.00 ©2023 IEEE
DOI 10.1109/SBAC-PAD59825.2023.00013

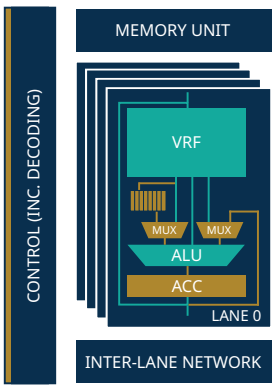


Fig. 1. Block diagram for the VSA xPU. The elements added to the VPU baseline are highlighted in golden color.

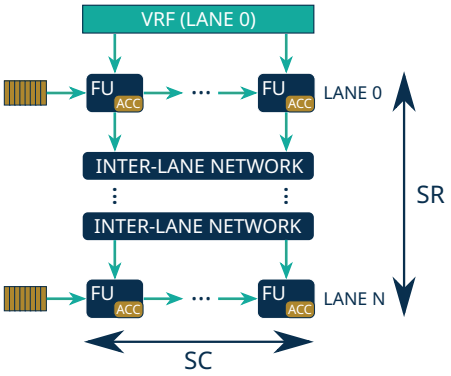


Fig. 2. Behaviour of the SA

work presents the following contributions:

- A set of custom memory access instructions to support SA-like patterns in VPU memory systems.
- A partitioning schema analysis for improved utilization of vector and systolic functionalities in the VSA xPU.
- A quantitative performance analysis of the enhanced VSA for DNN inference models and scientific applications.

We have prototyped and deployed both the VPU baseline and the VSA xPU into an FPGA using HLS. We have observed speedup increases of up to 72% thanks to the new memory instructions. In addition, we have proposed an analysis methodology to evaluate utilization and used the gained insights to maximize this parameter.

II. Background and Motivation

In this section we discuss the VSA xPU, how it merges a SA and a VPU into one hybrid architecture, how to compute GEMM using either SA or VPU, and what could be improved.

A. A Hybrid Vector-Systolic Architecture

In recent years, SAs and VPUs have been implemented together in different systems, to leverage the advantages of both architectures. Moreover, as more of these systems were being designed, the overlapping degree between both architectures got closer and closer. This has been taken to the extreme in [22], where it was shown that starting from a baseline VPU, it is possible to create a unit that can behave as a SA for GEMM with minimum hardware overhead, as shown in Figure 1. This is a Flexible Processing Unit, or xPU, a processing unit that provides different functionalities using the same hardware. This way, this novel xPU architecture can behave both as a SA (xPU_{SA}), or as a VPU (xPU_{VPU}). Following [22], here we also focus on an output stationary implementation of the SA.

Nowadays, VPUs provide parallelism in three different ways [17]–[19]: (1) by working with a Vector Register File (VRF) instead of with scalar registers, (2) by leveraging the full datapath width using packed SIMD, and (3) by instantiating multiple lanes inside a VPU. xPU_{SA}, leverages all these parallelism levels: (1) the vector register is used to emulate the input streams, (2) packed SIMD determines the column parallelism, and (3) multiple lanes are used to emulate the multiple rows of the SA. The values are initially stored in the VRF. Rows of the first matrix are stored in the VRF slice of the corresponding lane. Columns of the second matrix are stored in the VRF slice of the first lane. This way, this xPU can remap the functional units available on the baseline VPU as shown in Figure 2. This results in a SA with sizes SR $\times$ SC, being SR the number of rows and SC the number of columns.

In order to choose between xPU_{SA} and xPU_{VPU}, [22] proposed to extend the existing RISC-V vector extension [23], with an instruction that forces the VPU to run as a SA. This instruction can be represented by the mnemonics `vsa` for integer and `vfsa` for floating point data types.

B. GEMM

GEMM, which stands for General Matrix Multiply, is a fundamental linear algebra operation used in a wide range of scientific and data analytic applications, as well as in the field of Deep Learning (DL). In this last field, GEMM is the core kernel of DNN models. Example applications that use it are Finite Element Solvers (FES), such as [24]. GEMM can also be used to compute 2D convolution [25], one of the core kernels in image processing. Moreover, this has been integrated with DL, and nowadays the 2D convolution is the core kernel of Convolutional Neural Networks (CNNs). However, CNNs are not the only type of DNNs that use GEMM, as it is also used in transformers [26], and it can compute the dense or fully connected layers.

While GEMM is defined in BLAS as $C[M,N] = \alpha \times A[M,K] \times B[K,N] + \beta \times C[M,N]$ [27], we focus on the matrix operations. Thus, in this work GEMM is computed as $C[M,N] = A[M,K] \times B[K,N] + C[M,N]$.

1) GEMM with xPU_{SA}: In this xPU, the initial idea is to use xPU_{SA} for this kernel, as this is the purpose of adding the SA. Therefore, GEMM can be computed with xPU_{SA} using the custom `vsa/vfsa` instruction as shown in Algorithm 1.

2) GEMM with xPU_{VPU}: GEMM is a highly vectorizable kernel, and thus, it is suited to be efficiently offloaded to a vector processor. Thus, it can also be computed by xPU_{VPU}, as it retains all the functionality of the baseline VPU. A

Algorithm 1 Computing GEMM with xPU_{SA}

```
1: SET_DATA_WIDTH(D)
2: v_idx_a = GEN_IDX_A()
3: v_idx_b = GEN_IDX_B()
4: v_idx_c = GEN_IDX_C()
5: for all i ∈ {1, ..., M/SR} do
6:     v_a = LOAD_IDX(v_a, v_idx_a)
7:     for all j ∈ {1, ..., N/SC} do
8:         v_b = LOAD_IDX(v_b, v_idx_b)
9:         v_c = LOAD_IDX(v_c, v_idx_c)
10:        v_c = vsa/vfsa(v_a, v_b, v_c)
11:        STORE(v_c)
12:    end for
13: end for
```

basic vectorized algorithm for computing GEMM is shown in Algorithm 2 (`.vs` indicates a vector-scalar instruction).

C. Motivation

While the original VSA paper efficiently merges both architectures with minimal hardware overhead and extends the control to decode and handle the new instruction, this is not enough to exploit the maximum performance from an xPU, as described before.

1) Memory: An efficient use of new architecture requires a way to efficiently feed it with input data. To this end, xPU_{SA} requires different data patterns compared to xPU_{VPU}, and thus it is needed to adapt to them. The original paper proposes two options: (1) assuming there is a scratchpad that can provide the correct patterns, or (2) using indexed memory accesses. While the former would need extra hardware support, the latter is inefficient. Out of the three memory access modes supported in vector ISAs such as RISC-V (unitary, strided, and indexed) [23], indexed memory accesses are the least efficient ones. Instead of using a defined regular pattern, they take as extra input a vector register containing the offsets to the corresponding base address. Thus, a way to support the new patterns required by xPU_{SA} should be researched.

2) Mode selection: In the VSA xPU, the problem is no longer choosing between two different accelerators. Rather, it implies choosing between two different modes in the same accelerator. However, the decision-making problem is still present. This is especially relevant in the case of GEMM, as both xPU_{SA} and xPU_{VPU} can compute this kernel. While the first intuition would be to compute it with xPU_{SA}, the original paper showed that this intuition is wrong in some cases. Therefore, we should understand under which conditions each mode is better.

III. Memory Analysis

In the VSA xPU, both xPU_{SA} and xPU_{VPU} use the same memory system: a VRF connected to L1 or directly to L2. So xPU_{SA} still uses the memory support designed for xPU_{VPU}. Thus, there is potential for improvement in this area.

Algorithm 2 Computing GEMM with xPU_{VPU}

```
1: for all i ∈ {1, ..., M} do
2:     v_c = LOAD_ROW(C, M)
3:     for all j ∈ {1, ..., K} do
4:         a = A[ i×K + j ]
5:         v_b = LOAD_ROW(B, K)
6:         v_c = vfmacc.vs(a, v_b, v_c)
7:     end for
8:     STORE(v_c)
9: end for
```

A. Maximizing VRF utilization

The first point to notice is that, with the values of matrix B being stored only in the VRF slice of the first lane, the slices of the other lanes are not being utilized. Moreover, to balance the dataflow, the slices keeping the rows cannot be fully utilized either. In this approach, the first lane acts as the data source, while the last lane acts as the sink.

One way to solve this would be to continue storing the columns in the following lanes. When all the elements of matrix B in the first lane have been used, the second lane starts behaving as a data source. After getting to the last lane, the elements will continue to the first one, acting now as a sink instead of the last one. This process will continue until lane $N-1$ behaves as the weight source and lane $N-2$ does so as a sink. This way, vector register utilization can be maximized.

B. Data pattern description

With a new functionality, new memory access patterns appear. In particular, the element placement inside the VRF has to exhibit specific patterns, required by xPU_{SA} so that the right element arrives at the right functional unit at the right time. These patterns, which differ from the ones that VPUs are designed for, are shown in Figure 3, and are described as:

1) For matrix A, in each lane, the corresponding vector register slice shall contain the corresponding row of A, up to P, which is the maximum amount of elements that can be pipelined for the execution of one instruction.
2) For matrix B, all the elements shall be placed first in the first lane, and then continue filling the lanes in order until the last lane is full. The amount of elements of the same column that fit in the vector register slice of one lane determines the maximum P.
3) For the output matrix, each lane will contain the corresponding row of the tile.

C. Data pattern implementation

To achieve these patterns without memory support, the only option is to use vector-indexed memory accesses. We exclude rearranging the data in memory, which would incur in time and memory overheads, and having extra hardware support to do this rearrangement. To use indexed memory accesses, the corresponding index vectors need to be generated. Pseudocode for generating the said indices for matrices A, B and C is shown in Algorithms 3, 4 and 5 respectively. From these

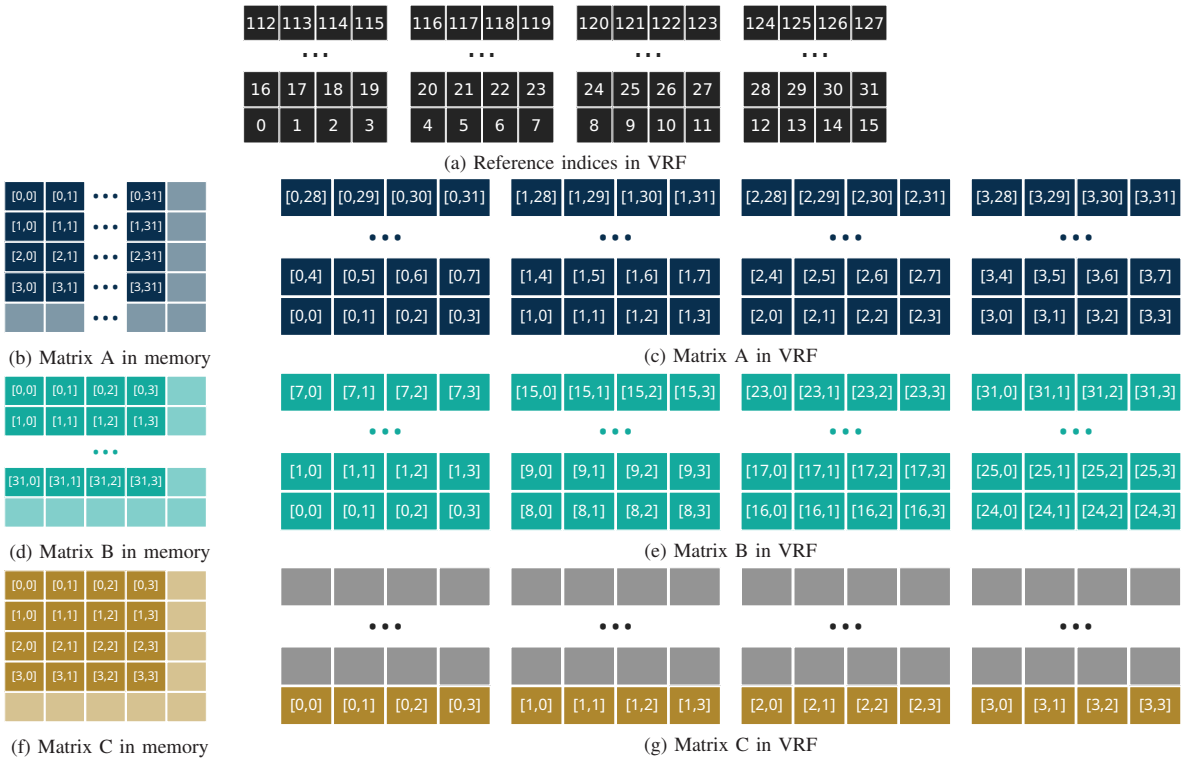


Fig. 3. Memory and VRF patterns for a VPU with 4 lanes, 4 elements per lane and MVL that fits 128 elements

algorithms, we can observe that said patterns are not trivial to vectorize for a traditional VPU architecture. Finally, only one different index pattern per matrix is needed. This is because, once a pattern is described as an index vector, given different base addresses, the generated indexes can extract the same pattern from different starting positions, thus not being needed to regenerate it every time.

While it is possible to use the xPU_{SA} with indexed memory accesses, this type of access is the least efficient among the available ones. Moreover, in this case, they fail to adequately capture locality due to the write order. However, understanding the previously mentioned requirements, it is possible to realize that there is indeed spatial locality to be leveraged. Regarding matrix A, it is clear that we are loading SR rows, each of them to a different lane, with a unitary stride. To leverage this locality, the approach would be to load first all the elements from the first row in the first lane, then the ones from the second row in the second lane, etc., via independent instructions. Moving on to matrix B, here all the values are loaded first into the first lane and, once it is full, it moves to the following one. Comparing with A, it is possible to see a trend where the VSA xPU would benefit if values could be accessed on a lane-by-lane basis. Focusing on the specific lane, the different columns of matrix B are loaded on a row-by-row basis, i.e., it takes packs of SC elements of each row, corresponding to the columns to be streamed. This pattern could be done by reading SC elements, and then jumping with a stride of N, to fetch the column elements of the next row. However, this would imply the need for another instruction for expressing that specific increase pattern. It would require a combination of a unitary stride with a non-unitary stride every SC elements. But if we go back to the original remapping, we can see that SC is determined by packed SIMD. This means we are loading as many elements as the datapath width allows. Having a datapath with W-bit width and working with D-bit wide data, we would be loading $W \times \text{SC} = D$ bits of column elements per row. Therefore, instead of loading independent elements, we could load them packed and then stride to the next row. This approach would thus require only a custom instruction that accesses the VRF on a lane-by-lane basis. This same idea could be applied to loading and storing the output tile matrix, accessing each row of it as packed data, and then moving to the next one. No custom instruction for accessing only specific lanes is needed in this case.

Therefore, to efficiently use the SA functionality available in the xPU, we propose to add a variation of the existing memory instructions that support memory accesses as the already available ones, but that interacts with only one lane at a time. These new instructions can replace the costly indexed memory accesses, and better leverage the spatial locality. Data comes into the VPU from the memory hierarchy the same way as in regular instructions, but then it is directed to a single lane instead of being distributed across all lanes. Therefore, the main issue that could arise from this new approach is that the performance would suffer due to a lower level of parallelism, specifically due to a bottleneck in the VRF. However, this is

Algorithm 3 Vector index generation for loading rows of matrix A

```
for all k ∈ {1, ..., P} do
    for all i ∈ {1, ..., SR} do
        for all j ∈ {1, ..., SC} do
            vd[k · SR · SC + i · SC + j] = k · SR + i · K + j
        end for
    end for
end for
```

Algorithm 4 Vector index generation for loading columns of matrix B

```
for all k ∈ {1, ..., P} do
    for all i ∈ {1, ..., SR} do
        for all j ∈ {1, ..., SC} do
            vd[k · SR · SC + i · SC + j] = (k + i · (#elements/#units)) · N + j
        end for
    end for
end for
```

Algorithm 5 Vector index generation for loading/storing an output matrix tile

```
for all i ∈ {1, ..., SR} do
    for all j ∈ {1, ..., SC} do
        vd[i · SC + j] = i · N + j
    end for
end for
```

not so troublesome as it may seem, as VRF can be sliced not only across lanes, but also within them [19], thus making memory ports that can be accessed in parallel not be a problem. Thus, the remaining concern would be the width of the path to the VRF, which may need to be widened in order to support the degree of parallelism within each lane.

Regarding the memory access extension, we add the new set of instructions with the same format as other already defined memory instructions, as shown in Figure 4 for load instructions. Here, we change the *OPCODE* field, adding one opcode for lane loads and another for lane stores. The main difference between these new instructions and the old ones is that they only interact with one lane, instead of with all. To do so, it is necessary to encode this lane selection functionality in the instruction. For that, we propose to use the *nf* field, which in the original instructions is used to enable segmented memory operations. This way, we can select up to 8 different lanes. Besides this, all the other functionality is the same. With this instruction, we modify the original xPU_{SA} implementation of GEMM shown in Algorithm 1 to the one presented in Algorithm 6. While the number of times the data width is set increases, this has little effect on the overall performance, as it just sets a control status registers. In addition, the *von Neumann* bottleneck is lessened in VPUs, as one instruction operates over multiple data [17].

Algorithm 6 GEMM using custom `vsa/vfsa` instruction with per lane memory access

```
for all i ∈ {1, ..., M/SR} do
    SET_DATA_WIDTH(D)
    for all r ∈ {1, ..., SR} do
        v_a = LOAD_LANE(v_a, r)
    end for
    SET_DATA_WIDTH(W)
    for all j ∈ {1, ..., N/SC} do
        for all r ∈ {1, ..., SR} do
            v_b = LOAD_LANE_STRIDE(v_b, r)
        end for
        v_c = LOAD_STRIDE(v_c)
        SET_DATA_WIDTH(D)
        v_c = vsa/vfsa(v_a, v_b, v_c)
        SET_DATA_WIDTH(W)
        STORE(v_c)
    end for
end for
```

While the new memory instructions can be generated by a compiler, we developed a library with an implementation of GEMM using these instructions.

IV. Mode Selection

In order to make the right decision selecting between xPU_{SA} and xPU_{VPU}, we need to understand how they handle GEMM differently. While the main difference between them is the utilization [22], we need to better understand the causes of under-utilization, and how it affects performance.

Comparing the algorithms for computing GEMM with VPU and SA functionalities (Algorithms 2 and 6 respectively), it can be seen that they partition the problem in different ways. The VPU algorithm computes GEMM on a row-by-row order, each iteration of the outer loop computing a matrix-vector multiplication. Contrary, the algorithm using xPU_{SA} does so by tiling the output matrix in tiles with shape SR $\times$ SC. Therefore, it is necessary to have a way to understand the performance differences, and see how changes in matrix sizes affect the utilization of the available resources. This can be used to enable optimized partitioning at runtime knowing the matrix sizes M, N, and K, as defined in Section II-B. Here, the best partitioning schema is selected, thus leveraging the heterogeneity offered by the xPU. To do such an analysis, the first step is to find the smallest matrices that can be computed by both architectures at full utilization. Here, utilization is measured at the instruction level. To find such matrices, we look for the Least Common Minimum (LCM) of each pair of matrices, as shown in Equations 1, 2 and 3 for sizes M, N, and K respectively. Starting from those sizes, then we do sweeps across the three dimensions, decreasing the sizes by their corresponding Greater Common Divisor (GCD) at each step. The result of this is a 3D array of values, one for each (M, N, K) set of matrix sizes. This is done for both VPU and SA functionalities. Then these volumes are divided, resulting

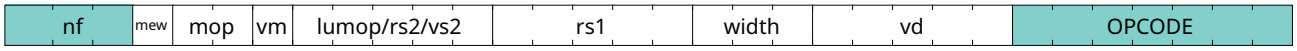


Fig. 4. Format of the proposed set of memory instructions. Highlighted are the elements that changed compared to the original RISC-V vector memory instructions.

in a single 3D volume containing the relative speedup for each (M, N, K) set.

$$\text{LCM}(M_{VPU}, M_{SA}) = \text{LCM}(1, \text{SR}) = \text{SR} \tag{1}$$

$$\text{LCM}(N_{VPU}, N_{SA}) = \text{LCM}\left(\frac{\text{MVL}}{D}, \text{SC}\right) = \frac{MVL}{D} \tag{2}$$

$$\text{LCM}(K_{VPU}, K_{SA}) = \text{LCM}\left(1, \frac{\text{MVL}}{W \cdot L}\right) = \frac{\text{MVL}}{W \cdot L} \tag{3}$$

Having this 3D array of relative speedups, the next step is to find the border, i.e., the points where the better approach changes. If the tests are done running an operating system, the lack of determinism will make it so that this border is not clear, as the tested matrix sizes are quite small. Thus, in order to analyze the border, we propose to apply an error function that indicates the performance loss for each (M, N, K) set. Then, we look for the points that minimize this performance loss. For that, we apply it over the dimension N and sweep through the other two. This way, we can recreate such a border. The reason for choosing this dimension is that the VPU instruction is not affected by variations in K, while the number of points across M is quite small compared to N, and thus would offer worse results. Along N, VPU performance will be comparatively better for higher values, where it is closer to full utilization. Considering values < 1 to show better VPU performance, we propose the error function shown in Equation 4, where r_i represents the ith element in the selected row across dimension N, j represents the potential border element that is being evaluated, and n represents the number of elements in a row.

$$f(r) = \sum_{i=0}^{i<j} (1 - r_i) + \sum_{i=j}^{i<n} (r_i - 1) \tag{4}$$

Knowing for which matrix sizes each of the functionalities provides better performance, we can partition the problem according to this information. With this, we can develop a GEMM implementation with increased utilization, and include it in a library, so that the user does not need to perform this utilization analysis. Note that the results of this analysis depend on the specific hardware implementation, and thus such a library needs to be optimized for the specific system. An example of this evaluation is shown in Section VI-B.

V. Experimental Setup

In order to analyze the different architectures discussed in this paper, we have implemented them using HLS and evaluated the design on a ZCU102 development board, which contains a Zynq UltraScale+ MPSoC. Table I contains the main specifications of the board. The main application runs on the ARM hardcores, while the computation of GEMM is offloaded to the FPGA fabric, acting as a coprocessor accelerator. Due to the overhead of offloading to the FPGA, we have moved the instruction generation to the fabric. On the software side, the board was running a Linux kernel, generated with *Petalinux*. For measuring power, we have leveraged the */sys/class/hwmon* interface that provides access to the measurements from the different power rails taken by different INA 226 integrated circuits (ICs) from TI. We get the actual power consumption of the device by running a power measuring program in parallel that reads the ICs and computes the current power with a fixed sampling rate. Based on the instant power and on the sampling rate, we compute the energy consumption. Although this application runs in parallel, it does not affect the performance of running benchmarks. As for the specific implementations, we have done them programming in C/C++ and then generating the RTL using HLS.

TABLE I
ULTRA96V1 SPECIFICATIONS

Processing System	
CPU	Quad-Core Cortex-A53
Frequency	1.2GHz
L1d Cache	32 kB
L1i Cache	32 kB
L2 Cache	1 MB
Programmable Logic	
LUTs	274 080
Flip-flops	548160
Distributed RAM	8.8 Mb
Block RAM (total)	32.1 Mb
DSP	2 520
Memory	
RAM	4GB LPDDR4, 2666 MHz

With this setup, we implemented both the baseline VPU and the original VSA xPU presented in [22]. First, we validated our FPGA results against the ones presented in [22]. Then, we analyzed the original VPU baseline. We observed that the original simulator [28] performs the vector-scalar operation by means of broadcasting the scalar to a vector register. In this paper, our VPU baseline is an actual vector-scalar implementation, where the input corresponding to the scalar register is fixed for the execution of the whole instruction.

To represent the different configurations in the design space analysis, we modified the number of lanes and the total number of units. We chose array sizes of 2×2, 4×4, and 8×8, as bigger sizes would lead to configurations uncommon in current VPUs, even for narrow datatypes. As for the VRF, we evaluated different Maximum Vector Length (MVL) configurations. MVL defines the maximum number of bits that fit in a vector register at a given time. As the original xPU was designed targeting long-vector architectures, our configurations range from 2048 bits (ARM's SVE biggest supported MVL [29]) up to 16384 bits ([17], [19], [28]). This needs to be seen in light of the

TABLE II
SUMMARY OF APPLICATIONS USED

Application/Benchmark	Domain	M	N	K
ResNet18 [30]	DL	64-512	1-16384	147-4608
AlexNet [31]	DL	1-384	169-4096	363-9216
DeepBench Device [32]	DL	64-5124	1-1500	128-2048
Linpack [33]	SC	2-8190	2-128	2-129
Low Order FES [34]	SC	8	32	16

datatype width. For example, experiments with 32b data and MVL of 16384b are equivalent to experiments with narrower data and shorter MVL (e.g., 8b data and MVL of 4096b). In our case, we will be showing the results of 32b data.

A. Workload

We define each GEMM computation as $C[M,N] = A[M,K] \times B[K,N] + C[M,N]$. To compute it, we have implemented Algorithms 1, 2 and 6 at the instruction level, and programming in a vector-length agnostic fashion. We have selected applications from the fields of Deep Learning (DL) and Scientific Computing (SC). For the former, we have selected two well-known image recognition models: ResNet18 [30] and AlexNet [31]. We have also used the DeepBench benchmark, which is part of the Coral-2 benchmark suite [32]. As for applications in the scientific domain, we have tested the Linpack benchmark [33] and finite element solvers as presented in [34]. Table II summarizes the applications used, and shows the ranges of their matrix sizes.

The methodology used for the evaluation of the xPU is to focus on the different GEMM calls of each application. This is a valid approach as GEMM is the most compute-intensive kernel across all the evaluated applications. Therefore, one intermediate step in this evaluation is to determine the input sizes for each GEMM, either from the corresponding papers or from executing the applications. For AlexNet and ResNet18, we got them from the Darknet framework [35], after the transformation with the *im2col()* function, taken from the Caffe framework [36].

VI. EXPERIMENTAL RESULTS

A. Custom Memory Accesses

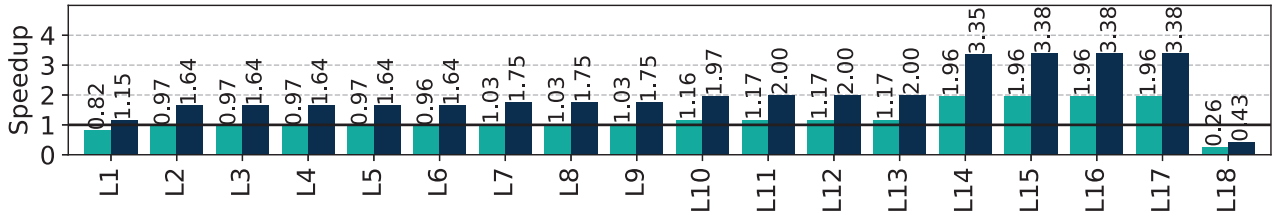
We start by analyzing ResNet18, but the conclusions extracted for the individual GEMM calls also apply to the other applications. The first step is to analyze the performance for all layers of ResNet18, as shown in Figure 5. This figure represents the execution time speedup for running xPU_{SA}, compared to the VPU baseline. It shows two sets of bars: the first one uses indexed memory accesses, as in the original paper (Algorithm 1), while the second one uses our proposed memory access (Algorithm 6). Both sets include the improved vector-scalar baseline and the increased VRF utilization proposed in Section III. These two improvements cancel each other in terms of performance. The specific configuration for this experiment consisted of an MVL of 16384 bits and 16 functional units organized as a 4×4 SA. As can be observed, the initial xPU implementation using indexed memory accesses struggles to provide speedup, which is only significant in layers 14-17. This is due to the indexed accesses not leveraging the locality. However, by adding our custom lane-by-lane memory accesses, xPU_{SA} achieves a speedup of up to 3.38x compared to the VPU baseline. This is an increase of up to 72% compared to xPU_{SA} with the VPU memory support.

Next, we show in Figure 5b the design space analysis for xPU_{SA} for different configurations, changing both the MVL and the number of functional units. Here the data shown is the result of calculating the speedup across all GEMM calls together, adding all the execution times and then computing the speedup. The first point to notice is that, for greater SA sizes, xPU_{SA} offers more speedup. However, when using indexed memory addresses, the speedup decreases as the MVL increases. This is contrary to what is shown in [22], and it is due to using actual vector-scalar operations instead of performing them by means of broadcasting. While broadcasting time depends on the MVL, pure scalars take the same time to load independently of it. It is not that increasing the MVL makes xPU_{SA} perform slower, but the VPU baseline leverages it better. However, when using our custom memory instructions, xPU_{SA} can leverage the increase in MVL. This is due to the lane-by-lane instructions being able to appropriately leverage the spatial locality present in the SA-like patterns. One point to note is that a similar conclusion to this can be extracted from the energy consumption data. The corresponding energy measurements are shown in Figure 5c.

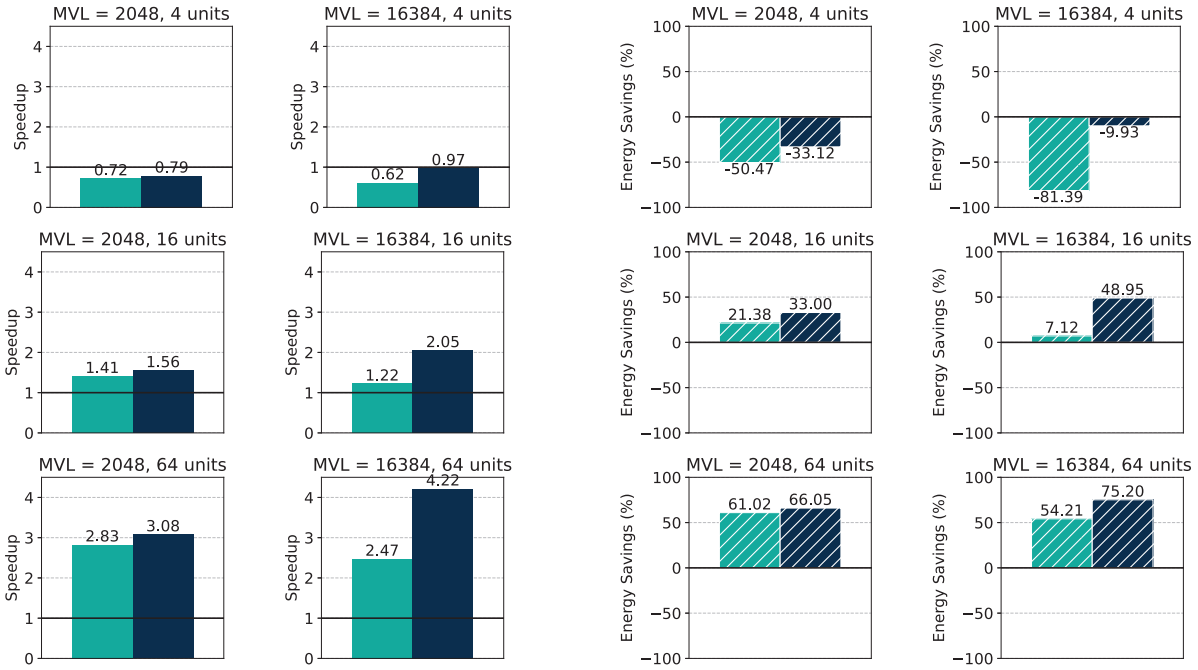
B. Leveraging Heterogeneity

As discussed in Section IV, algorithms for $\text{xPU}_{\text{SA}}$and xPU_{VPU} partition the problem in different ways. By understanding the implications of these different approaches, we can leverage the heterogeneity of the different matrix sizes with the heterogeneity of our hybrid architecture. Therefore, now we will analyze how this affects the xPU by applying the analysis methodology presented in Section IV. Important to remember is that the results presented here apply only to the specific configuration tested. This analysis would have to be repeated for different configurations. Like in the layer analysis, the configuration for this experiment consisted on a MVL of 16384 bits and 16 functional units organized as a 4×4 SA.

The results of the analysis can be seen in Figure 6. In this figure, each subfigure represents the sweep across dimensions M (Y axis) and N (X axis) for a given K. Therefore, they are slices of the 3D array obtained after performing all the sweeps. Here we only show five values of K: the one with minimum utilization for xPU_{SA}, and milestones corresponding to loading the columns up to filling different lanes. Each point thus represents the relative difference between computing GEMM for a given (M,N,K) with xPU_{SA} or xPU_{VPU}. As we can see, for $K = 128$, i.e., when the deepest pipeline available for this configuration, xPU_{SA} can outperform xPU_{VPU} as long as the vertical tiling does not leave less than 3 rows per tile (Figure 6e). If not, the penalty for padding with new rows will not be recovered. Note that, for matrices where $M >$ SR, this only applies to the edges, as the rest of the



(a) Layer-by-layer speedup. Configuration of 4×4 units and MVL = 16384 bits.



(b) Total speedup for different xPU configurations.

(c) Total energy savings for different xPU configurations.

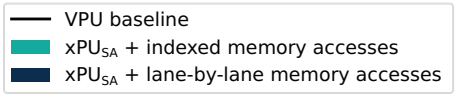


Fig. 5. ResNet18 results including the support of the new memory instructions.

rows can be tiled in groups of SR rows. Across the other tiling dimension, the closer the matrix can be partitioned in groups of MVL/D elements, the better it will be for xPU$_{VPU}$, as it will get closer to full utilization. Here, xPU$_{SA}$ also pads new columns, but the penalty is smaller than the one paid by the VPU. If K is reduced, the benefits of running xPU$_{SA}$ are incrementally reduced, until reaching the state seen in Figure 6a, where it cannot outperform xPU$_{VPU}$. This is due to the fact that initialization and synchronization penalties are not being balanced out.

In a more general way, this means that the VRF needs to be able to feed sufficient data to xPU$_{SA}$. The functional units will not be sufficiently fed if K or the VRF is small. Therefore, when implementing this xPU, it is important to keep a good compute memory balance. When designing xPU, units that offer different functionalities in the same hardware, it is important to consider that all the parameters affect all the functionalities. This makes the design process more sensitive, as changes that are beneficial to one architecture can be detrimental to the other.

With this data, we get the border with Equation 4, and divide the problem based on it, running xPU$_{SA}$ or xPU$_{VPU}$ according to which one is better for each situation.

Figure 7a is the updated version of Figure 5, in which a third set of bars has been added. This new set of bars represents the combination of xPU$_{SA}$ and xPU$_{VPU}$ with optimized functionality selection. This approach aims to minimize underutilization. As it can be seen, most of the layers remain the same, as the initial configuration turned out to be the best one for the corresponding matrix sizes, and thus the improved

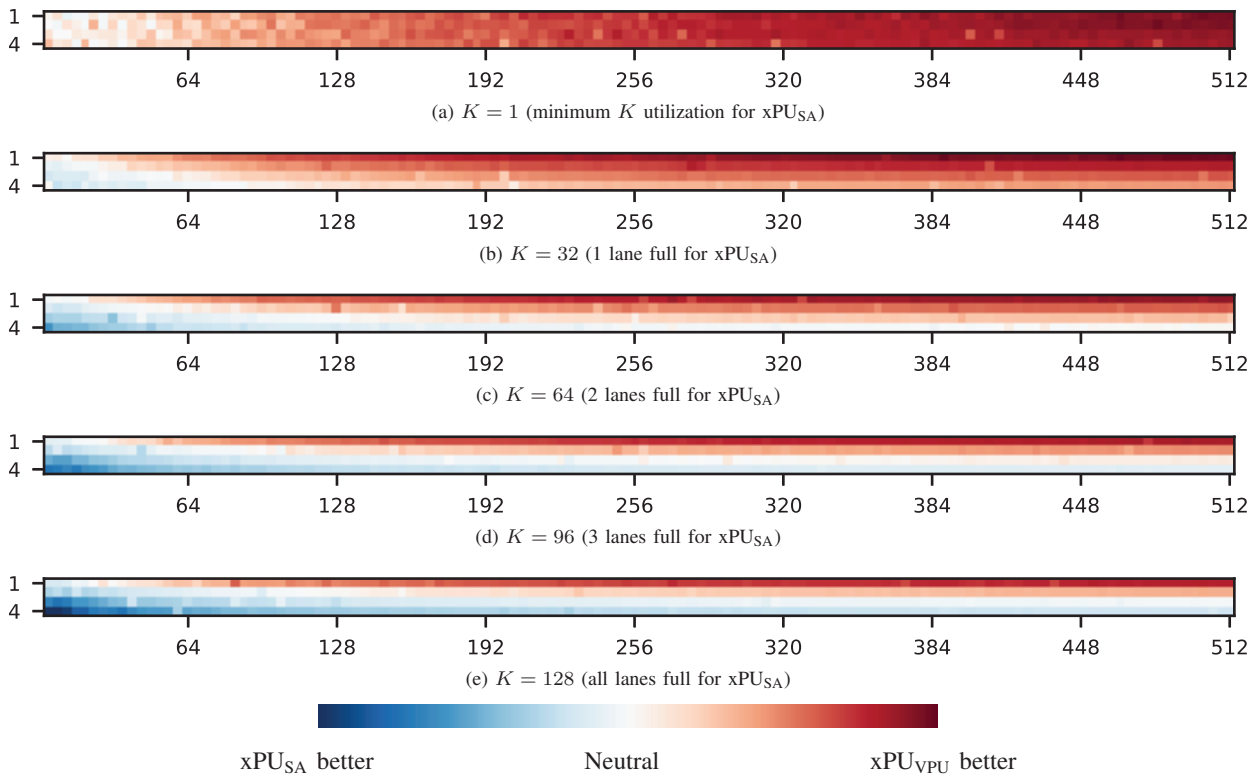


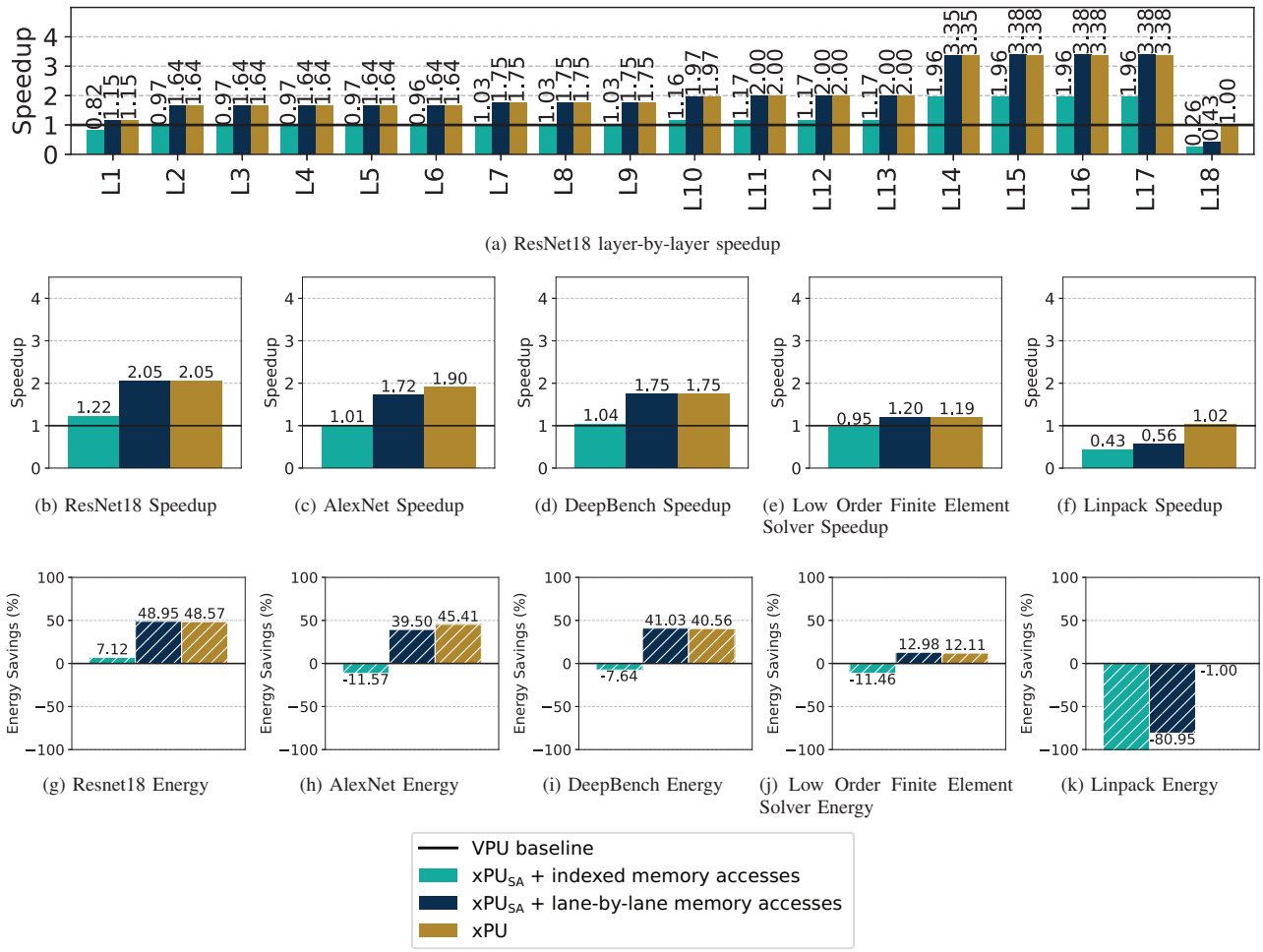
Fig. 6. Tradeoffs between xPU$_{VPU}$ and xPU$_{SA}$ for different pipeline depths. The X-axis represents the different values of N and the Y-axis the different values of M. Configuration of 4×4 units and MVL = 16384 bits.

implementation still computed these layers exclusively using xPU$_{SA}$. However, for the last layer, where xPU$_{SA}$ was being heavily underutilized, the new program can detect it and compute it more efficiently with xPU$_{VPU}$. For each GEMM call, this improved implementation provides the best partitioning, maximizing the utilization of the available resources using xPU$_{SA}$ or xPU$_{VPU}$ accordingly. Moreover, as can be seen in Figure 7a, this hybrid approach does not incur any relevant penalty. The only overhead present is partitioning the problem at runtime, but it is negligible relative to the matrix multiplication itself. Figure 7a shows the updated results for the whole network. As only one layer, the least time-consuming one showed benefits, no relevant changes can be seen compared to the xPU$_{SA}$ implementation. The same conclusions apply to energy consumption.

Besides ResNet18, we have also tested other applications, as described in Section V-A. For AlexNet we can see that performance increase is achievable using the custom load memory accesses (Figure 7c). Moreover, here we can see how the utilization analysis can provide observable benefits. DeepBench offers results similar to ResNet18 (Figure 7d). Regarding scientific applications, for the Low Order Finite Element Solver, we observe only minimal improvement (Figure 7e), as its matrix sizes are small, i.e., $(M, N, K) = (8, 32, 16)$, problem size for which GEMM is compute bound. The small performance increase is due to using fewer instructions, and a slightly better VRF utilization. As for Linpack, most of the GEMM calls have small values of K. Therefore xPU$_{SA}$'s pipeline suffers, and cannot achieve good performance compared to xPU$_{VPU}$ in most operations. However, as seen in Figure 7f, our optimized implementation can detect this and avoid any performance degradation.

VII. Related Work

VPUs and SAs have their corresponding advantages and drawbacks and, to be able to leverage the advantages of both, several works have combined them in different degrees. One of the first architectures to do this is Google's TPUv2. In this case, they had to extend the SA-based architecture with a VPU to efficiently support batch normalization [21]. While this combination happened by necessity, to support a specific kernel, other architectures are intentionally combining both. One such example is the MEEP platform, which includes the Vector and Systolic Accelerator Tiles [37]. Each of these tiles provides one VPU and two different SAs. In this case, VPU and SAs share the issue unit with a scalar core and also share the memory interface. IBM went one step further in its latest Power10 processor [38]. In it, the VPUs share their VRF with the SAs. Finally, VSA difussed the barriers between VPUs and SAs to the point of reusing the arithmetic units of a VPU to implement a SA [22]. This can be called a Flexible Processing Unit, or xPU, a processing unit that uses the same hardware to offer different functionalities. However, that work presented a



(a) ResNet18 layer-by-layer speedup

(b) ResNet18 Speedup (c) AlexNet Speedup (d) DeepBench Speedup (e) Low Order Finite Element Solver Speedup (f) Linpack Speedup

(g) Resnet18 Energy (h) AlexNet Energy (i) DeepBench Energy (j) Low Order Finite Element Solver Energy (k) Linpack Energy

Fig. 7. Speedup and energy savings for tested applications. Configuration of 4×4 units and MVL = 16384 bits.

heterogeneous architecture, the available memory functionality to fetch the data was reused. This functionality was designed for a VPU and does not fit the SA without major penalties. In addition, while VSA was evaluated running as either a VPU or an SA for different GEMM calls, an in-depth analysis to understand under which conditions each of the functionalities is better was missing.

Another example of an xPU is SIMD^2 [39]. In this work, the authors observe that there are algorithms that share key characteristics with GEMM. In particular, they find several matrix applications with semiring-like structures equivalent to GEMM. Thus, they leverage this fact to present an extended SA that handles all those different matrix operations.

A different approach would be the case of TCUDB [40]. In this paper, the authors present an approach to efficiently compute databases using NVIDIA's Tensor Core Units (TCUs) [41]. Therefore, while not conceived as such, it could be said that TCUs have become a xPU *a posteriori*.

In summary, processing units that provide different functionalities are being proposed. For them, designers need to be aware that each aspect of the unit needs to be able to efficiently support both functionalities to exploit all its potential.

VIII. Conclusions

In this work, we have shown that, when designing processing units with different functionalities, all the aspects need to be considered for all the functionalities in order to achieve the best performance. To illustrate this we have selected VSA, a xPU that provides both SA and VPU functionalities. By analyzing in detail the memory patterns, we have been able to achieve speedups of up to 4.22x compared to a VPU baseline. This means an increase of up to 72% compared to only focusing on the computational aspect of the architecture. Moreover, we have presented a detailed analysis procedure that allows us to understand under which conditions each of the functionalities offers better performance. By integrating the insights from this analysis, we have obtained a software implementation that minimizes hardware under-utilization by using both functionalities combined.

References

[1] OpenAI, *Ai and compute*, May 2018. [Online]. Available: https://openai.com/blog/ai-and-compute/ (visited on 02/16/2023).

[2] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," *Communications of the ACM*, vol. 62, no. 2, pp. 48–60, 2019.

[3] S. Han, X. Liu, H. Mao, *et al.*, "EIE: Efficient inference engine on compressed deep neural network," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 243–254, 2016.

[4] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 367–379, 2016.

[5] B. Moons, R. Uytterhoeven, W. Dehaene, and M. Verhelst, "Envision: A 0.26-to-10TOPS/W subword-parallel dynamic-voltage-accuracy-frequency-scalable convolutional neural network processor in 28nm FDSOI," in *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, IEEE, 2017, pp. 246–247.

[6] A. Parashar, M. Rhu, A. Mukkara, *et al.*, "SCNN: An accelerator for compressed-sparse convolutional neural networks," *ACM SIGARCH computer architecture news*, vol. 45, no. 2, pp. 27–40, 2017.

[7] N. P. Jouppi, C. Young, N. Patil, *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th annual international symposium on computer architecture*, 2017, pp. 1–12.

[8] J.-W. Jang, S. Lee, D. Kim, *et al.*, "Sparsity-aware and re-configurable NPU architecture for samsung flagship mobile soc," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, 2021, pp. 15–28.

[9] Y. Tortorella, L. Bertaccini, D. Rossi, L. Benini, and F. Conti, "RedMulE: A compact FP16 matrix-multiplication accelerator for adaptive deep learning on risc-v-based ultra-low-power socs," in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2022, pp. 1099–1102.

[10] X. Chen, T. Huang, S. Xu, T. Bourgeat, C. Chung, and A. Arvind, "Flexminer: A pattern-aware accelerator for graph pattern mining," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, 2021, pp. 581–594.

[11] G. Dai, Z. Zhu, T. Fu, *et al.*, "Dimmining: Pruning-efficient and parallel graph mining on near-memory-computing," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 130–145.

[12] N. Talati, H. Ye, Y. Yang, *et al.*, "Ndminer: Accelerating graph pattern mining using near data processing," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 146–159.

[13] D. Fujiki, S. Wu, N. Ozog, *et al.*, "Seedex: A genome sequencing accelerator for optimal alignments in sub-minimal space," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2020, pp. 937–950.

[14] D. S. Cali, K. Kanellopoulos, J. Lindegger, *et al.*, "Segram: A universal hardware accelerator for genomic sequence-to-graph and sequence-to-sequence mapping," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 638–655.

[15] H.-T. Kung, "Why systolic architectures?" *Computer*, vol. 15, no. 01, pp. 37–46, 1982.

[16] R. M. Russell, "The cray-1 computer system," *Communications of the ACM*, vol. 21, no. 1, pp. 63–72, 1978.

[17] M. Cavalcante, F. Schuiki, F. Zaruba, M. Schaffner, and L. Benini, "Ara: A 1-GHz+ scalable and energy-efficient RISC-V vector processor with multiprecision floating-point support in 22-nm FD-SOI," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 2, pp. 530–543, 2019.

[18] M. Perotti, M. Cavalcante, N. Wistoff, R. Andri, L. Cavigelli, and L. Benini, "A "new ara" for vector computing: An open source highly efficient RISC-V V 1.0 vector processor design," in *2022 IEEE 33rd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, IEEE, 2022, pp. 43–51.

[19] F. Minervini, O. Palomar, O. Unsal, *et al.*, "Vitruvius+: An area-efficient RISC-V decoupled vector coprocessor for high performance computing applications," *ACM Trans. Archit. Code Optim.*, Dec. 2022, Just Accepted, ISSN: 1544-3566. DOI: 10.1145/3575861.

[20] A. Boroumand, S. Ghose, B. Akin, *et al.*, "Google neural network models for edge devices: Analyzing and mitigating machine learning inference bottlenecks," in *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, IEEE, 2021, pp. 159–172.

[21] N. P. Jouppi, D. H. Yoon, G. Kurian, *et al.*, "A domain-specific supercomputer for training deep neural networks," *Communications of the ACM*, vol. 63, no. 7, pp. 67–78, Jun. 2020.

[22] M. Vázquez Maceiras, M. W. Azhar, and P. Trancoso, "VSA: A hybrid vector-systolic architecture," in *2022 IEEE 40th International Conference on Computer Design (ICCD)*, IEEE, 2022, pp. 368–376.

[23] RISC-V, *RISC-V V vector extension*, 2023. [Online]. Available: https://github.com/riscv/riscv-v-spec.

[24] M. W. Scroggs, I. A. Baratta, C. N. Richardson, and G. N. Wells, "Basix: A runtime finite element basis evaluation library," *Journal of Open Source Software*, vol. 7, no. 73, p. 3982, 2022.

[25] K. Chellapilla, S. Puri, and P. Simard, "High performance convolutional neural networks for document processing," in *Tenth international workshop on frontiers in handwriting recognition*, Suvisoft, 2006.

[26] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[27] *BLAS (basic linear algebra subprograms)*. [Online]. Available: https://www.netlib.org/blas/ (visited on 11/14/2022).

[28] C. Ramírez, C. A. Hernández, O. Palomar, O. Unsal, M. A. Ramírez, and A. Cristal, "A RISC-V simulator and benchmark suite for designing and evaluating vector architectures," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 17, no. 4, pp. 1–30, 2020.

[29] N. Stephens, S. Biles, M. Boettcher, *et al.*, "The ARM scalable vector extension," *IEEE micro*, vol. 37, no. 2, pp. 26–39, 2017.

[30] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[31] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, 2012.

[32] *Coral-2 benchmarks*, 2023. [Online]. Available: https://asc.llnl.gov/coral-2-benchmarks.

[33] J. J. Dongarra, P. Luszczek, and A. Petitet, "The linpack benchmark: Past, present and future," *Concurrency and Computation: practice and experience*, vol. 15, no. 9, pp. 803–820, 2003.

[34] T. Yamaguchi, K. Fujita, T. Ichimura, *et al.*, "Low-order finite element solver with small matrix-matrix multiplication accelerated by ai-specific hardware for crustal deformation computation," in *Proceedings of the Platform for Advanced Scientific Computing Conference*, 2020, pp. 1–11.

[35] J. Redmon, *Darknet: Open source neural networks in c*, http://pjreddie.com/darknet/, 2013–2016.

[36] Y. Jia, E. Shelhamer, J. Donahue, *et al.*, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*, 2014, pp. 675–678.

[37] A. Fell, D. J. Mazure, T. C. Garcia, *et al.*, "The marenostrum experimental exascale platform (MEEP)," *Supercomputing Frontiers and Innovations*, vol. 8, no. 1, pp. 62–81, 2021.

[38] W. J. Starke, B. W. Thompto, J. A. Stuecheli, and J. E. Moreira, "IBM's POWER10 processor," *IEEE Micro*, vol. 41, no. 2, pp. 7–14, 2021.

[39] Y. Zhang, P.-A. Tsai, and H.-W. Tseng, "SIMD2: A generalized matrix instruction set for accelerating tensor computation beyond gemm," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ser. ISCA '22, New York, New York: Association for Computing Machinery, 2022, pp. 552–566, ISBN: 9781450386104.

[40] Y.-C. Hu, Y. Li, and H.-W. Tseng, "TCUDB: Accelerating database with tensor processors," in *Proceedings of the 2022 International Conference on Management of Data*, 2022, pp. 1360–1374.

[41] NVIDIA, *NVIDIA tesla V100 GPU architecture*, WP-08608-001_v1.1, 2017.