Thesis for The Degree of Licentiate of Engineering

Extending Vector Processing Units for Enhanced Linear Algebra Performance

Mateo Vázquez Maceiras

Division of Computer Engineering
Department of Computer Science & Engineering
Chalmers University of Technology and Gothenburg University
Gothenburg, Sweden, 2024

Extending Vector Processing Units
for Enhanced Linear Algebra Performance

Mateo Vázquez Maceiras

ISSN 1652-876X
Department of Computer Science & Engineering
Division of Computer Engineering
Chalmers University of Technology and Gothenburg University
Gothenburg, Sweden

This thesis has been prepared using LaTeX.
Printed by Chalmers Reproservice,
Gothenburg, Sweden 2024.

Abstract

Vector Processing Units (VPUs) have made a comeback to the landscape of computer architecture as a response to the diminishing returns from technology scaling and power density limitations. VPUs are used as general-purpose accelerators, offering a trade-off between the flexibility of general-purpose architectures and the performance of hardware accelerators. However, application demands keep growing. Thus, we want to extract even more performance out of VPUs, as well as achieving better area and energy efficiency. To achieve these improvements, one approach is to enhance current VPUs with Instruction Set Architecture (ISA) extensions tailored to specific kernels or applications.

A relevant set of kernels widely used nowadays are linear algebra kernels. These kernels have been used in multiple domains for decades. However, they are at the core of Machine Learning (ML) applications, which is one of the domains with the fastest requirement increase, both in terms of performance and energy. Consequently, there is a high interest in computing these kernels faster and more efficiently. VPUs are a good mapping for these kernels but they do not offer the same performance and efficiency as custom accelerators.

This Thesis presents two different extensions for enhancing linear algebra kernels in VPUs. The first extension enhances VPUs with the functionality of Systolic Arrays (SAs) for more efficient computation of General Matrix-Matrix Multiplication (GEMM). This enhancement is done by remapping the functional units of the VPU from a 1D to a 2D array. In addition, this Thesis also analyzes the implications of this new SA-like functionality, proposing corresponding new memory instructions and an analysis to dynamically select the functionality that maximizes resource utilization. The second extension proposes a memory extension that provides VPUs with index-matching functionalities for sparse linear algebra operations. This extension transforms the index-matching problem into one of hash lookup, and implements this problem in hardware using cache-like techniques. These extensions achieve up to 4.22x and 3.19x speedup respectively.

Keywords:

Vector, SIMD, Linear Algebra, Dense, Sparse, ISA extension

Acknowledgment

I would like to express my gratitude to my supervisor, Prof. Pedro Trancoso, for his continuous guidance and support during these years. I am confident I will continue to benefit from his help and encouragement during the following years of my doctoral journey. Likewise, I would also like to thank my co-supervisor, Dr. Muhammad Waqar Azhar, for the advice and insights shared during these years.

I would like to extend my appreciation to my colleagues and fellow researchers at Chalmers, Fareed, Stravoula, Mo, Alessio, Xu, Sonia, Arne, and others for their support and company. I also include here my collaborators at other institutions, like Alexandre Rodrigues at INESC-ID, Lisbon.

Finally, I would like to give special thanks to my family for their continuous support and unwavering encouragement. Specially, I would like to thank my parents for always being there despite all the difficulties, and to my cousin and friend Timín for sharing with me his passion for hardware design. I would not have been here without any of them.

This work would have not been possible without the research grants from the eProcessor project, which has received funding from the European High-Performance Computing Joint Undertaking Joint Undertaking (JU) under grant agreement No 956702. The JU receives support from the European Union's Horizon 2020 research and innovation programme 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 programme under grant agreement No 957197 and the Swedish Foundation for Strategic Research (contract number CHI19-0048) under the PRIDE project.

Mateo Vázquez Maceiras
Gothenburg, May 2024

List of Publications

Appended publications

This thesis is based on the following publications:

I Mateo Vázquez Maceiras, Muhammad Waqar Azhar, Pedro Trancoso "VSA: A Hybrid Vector-Systolic Architecture"
2022 IEEE 40th International Conference on Computer Design (ICCD).

II Mateo Vázquez Maceiras, Muhammad Waqar Azhar, Pedro Trancoso "Exploiting the Potential of Flexible Processing Units"
2023 IEEE 35th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD).
Best Paper Award.

III Mateo Vázquez Maceiras, Mohammad Ali Maleki, Muhammad Waqar Azhar, Pedro Trancoso "Scalable Hardware Hash for Index-Matching in Vector Architectures"
Submitted to the *2024 International Conference on Parallel Architectures and Compilation Techniques (PACT).*

The papers will be referred to in the thesis using their Roman numerals.

Contents

Chapter 1

Introduction

The landscape of computer architecture in recent years has been characterized, among others, by diminishing returns from technology scaling and by power density limitations. In addition, application requirements are increasing faster than before [1]. In order to bridge that gap, processors are shifting from homogeneous multi-cores composed of general-purpose CPUs to heterogeneous System-on-Chip (SoC) designs [2]. These SoCs integrate one or more domain-specific accelerators coupled with the host CPU(s). Consequently, systems can offer the required performance while still fulfilling the power budget.

However, this approach has drawbacks. Its main drawback is the area overhead required for implementing said accelerators [3]. This overhead implies a cost increase in both development and implementation. This approach may even be totally unfeasible to implement on some devices due to resource limitations. Moreover, the general-purpose functionality of SoCs stills need to efficiently compute tasks that cannot be offloaded to the accelerators. Under such conditions, exploring the use of more general units that can support multiple domains is interesting. One such flexible unit is the Vector Processing Unit (VPU), which can execute vector or SIMD instruction extensions. This unit can be efficiently leveraged by vectorizable applications. In particular, linear algebra kernels are a good match for VPUs [4].

Linear algebra is a key component in multiple applications across different domains. Usage examples are image recognition [5, 6], natural language processing [7], graphs [8–10], databases [11] and finite element solvers [12]. Hence, recent works are still trying to find optimized software implementation for linear algebra kernels [13–15]. However, while VPUs are a good match for these kernels, they are not as efficient as custom accelerators. One example is the case of General Matrix-Matrix Multiplication (GEMM), which can be more efficiently computed with 2D Systolic Arrays (SAs) [16]. Other example are sparse-sparse operations, which rely on costly index-matching operations [17]. Therefore, the goal of this Thesis is to further improve the performance of linear algebra kernels in VPUs from a hardware perspective.

1.1 Background

1.1.1 Vector Processing Units

Vector architectures are a type of computer architecture designed to perform operations on arrays of data. Compared to traditional scalar cores, VPUs operate over sequential registers, known as vector registers. The maximum number of elements that can fit at once in a given vector register is limited by an architectural parameter named Maximum Vector Length (MVL). Each vector instruction works with a Vector Length (VL) stored in a Control Status Register (CSR). In case the target vectors are longer than the MVL, they have to be partitioned via software. However, most vector Instruction Set Architectures (ISAs) support vector agnostic programming. Consequently, it is possible to program a VPU without knowing its MVL, as the vectorized code can adapt itself at runtime [18]. To increase computational capabilities, VPUs implement multiple functional units in parallel. Moreover, as most operations will involve regular patterns, these functional units are grouped

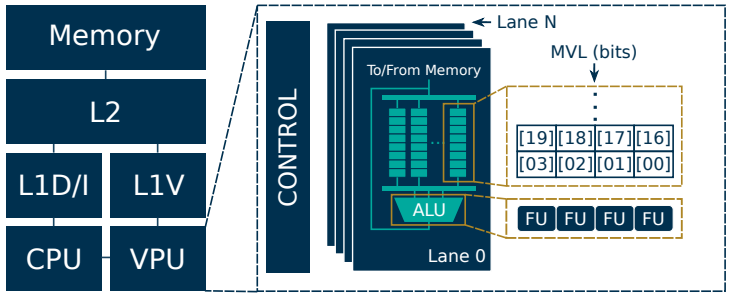


Figure 1.1: Block diagram of a system with a VPUs

into multiple vector lanes. Each lane contains its corresponding functional units and the corresponding slice of the Vector Register File (VRF). This way, it is possible to implement an inexpensive multi-ported vector register by using multiple interleaved banks, with fewer ports per bank [19]. Moreover, inside each lane, it is possible to further interleave banks, reducing ports per bank while still being able to read and write data from multiple registers in parallel [20]. This way, the VRF's banks are interleaved both across lanes and within the lanes. Figure 1.1 shows an example of how a VPU is organized and how it can be integrated within a CPU-based system.

Historically, vector architectures have been implemented targeting the supercomputing domain, with one of the most well-known examples being the CRAY-1 supercomputer from the 70s [21]. However, these architectures began to fade away with the appearance of microprocessor-based systems, which offered more competitive cost-performance off-the-shelf. In recent times, vector support has been added to CPUs, with SIMD extensions like Intel's AVX [22], ARM's NEON [23], ARM's SVE [18] and the RISC-V Vector Extension [24]. In the case of the latter, there have been several projects in academia in recent years, such as Ara [25], Arrow [26], and Vitruvius+ [20].

1.2 Linear Algebra Kernels

Linear algebra kernels form the backbone of numerous computational tasks across various domains, ranging from scientific computing to machine learning. They work with data structures like matrices and vectors. Specially relevant are the Basic Linear Algebra Subroutines (BLAS), a standardized set of optimized routines for common low-level linear algebra operations [27, 28]. Depending on the data they are working with, linear algebra kernels can be classified as dense or sparse. This difference is so relevant that both Colella's and Berkeley's Dwarfs separate dense and sparse linear algebra kernels as two different dwarfs [4, 29]. While the mathematical operations are the same, the nature of the data invites different implementations, which leads to different computing characteristics.

1.2.1 Dense Linear Algebra Kernels

Dense linear algebra kernels leverage the regularity of their data. The uniformity and predictability of this data enable efficient utilization of computational resources by facilitating data access patterns that can be easily optimized for performance. These are highly parallelizable and, when working with matrices, they offer high data reuse. By optimally leveraging this reuse, we can achieve heavily compute-bound implementations [25].

1.2.2 Sparse Linear Algebra Kernels

Contrary to their dense counterparts, sparse linear algebra kernels cannot leverage regularity within their data. The reason is that, in order to save space and avoid computations with zeros, sparse data is usually stored in a compressed data format [30–36]. While using such compressed formats for sparse data leads to more efficient kernels, the regular memory accesses are replaced by three operations: indirection, intersection, and union [37]. Indirection is

used by sparse-dense operations, while sparse-sparse operations use intersection and union. Indirection can be implemented with scatter-gather instructions, which are supported in VPUs with indexed memory accesses [24]. For intersection and union, as both arrays are compressed, finding a corresponding element implies comparing the different indices until a match is found. If no matching element is found, then the corresponding element is a zero. Therefore, linear algebra kernels where both arrays are sparse rely on index matching.

1.3 Problem Statement

1.3.1 Problem 1

Within dense linear algebra, General Matrix-Matrix Multiplication (GEMM) is the most widely used kernel, as well as one of the most computationally intensive kernels. GEMM is a fundamental operation in many applications across different domains, such as ML, big data, and general scientific computing. Moreover, it can be efficiently implemented in a VPU due to being highly vectorizable. Despite this, multiple custom accelerators have been proposed in order to further improve GEMM's performance and efficiency, with SAs being one of the common underlying architectures. Compared to VPUs, SAs provide a more efficient data reuse by feeding the output of the functional units directly to the inputs of the next ones [38, 39]. Consequently, multiple works have presented SoCs that combine both VPUs and SAs [16, 40, 41]. However, these works provide both functionalities by replicating hardware structures, like the Fused Multiply-Accumulate (FMA) units used to perform the computation. This means incurring an area increase. Hence, by avoiding this replication, we could develop more area-efficient implementations. This would especially benefit resource-constrained devices, enabling them to provide both VPU and SA functionalities with little overhead.

Problem Statement 1: Given a baseline VPU, extend it with the functionality of a SA for increasing the performance and efficiency of computing GEMM. Design this extension with area in mind, adding minimal hardware overhead.

Key Insight 1: From a high level of abstraction, VPUs and SAs are a set of functional units with a memory system that feeds them. VPUs arrange these units as a 1D array, while SAs arrange them as a 2D array. We can implement the flow of a SA in a baseline VPU by remapping its functional units to support forwarding data directly to the next set of units, organizing them as a 2D array that an SA would support.

1.3.2 Problem 2

Index-matching is the main problem in sparse-sparse linear algebra operations. Finding the matching element means comparing multiple indices until finding the right one. Sparse linear algebra kernels are memory-bound [42], and this increases the memory bottleneck. One solution to solve this is to move the index-matching operation to memory, only sending to the compute part the actual value. Current state-of-the-art for VPUs already supports this approach, but it uses hardware techniques that rely on data replication and provide poor scaling [17, 43, 44]. Therefore, to efficiently provide the same memory size and parallelism as regular memory architectures for vector architectures, we need to look for memory architectures that provide the same functionality while offering better scalability.

Problem Statement 2: Given a baseline VPU, extend its memory architecture to support index-matching operations. To fit with VPUs, which are general-purpose accelerator architectures, this new memory architecture also needs to support regular memory accesses and be scalable, not being limited to small memories.

Key Insight 2: Index-matching is equivalent to a hash-lookup problem, with the index being the key. This can be implemented in hardware using cache-like techniques, where a hash bucket maps to a cache set.

1.4 Thesis Contributions

In summary, the goal of this Thesis is to contribute to the development of vector architectures, focusing on linear algebra. For that, it proposes two contributions, tackling both dense and sparse linear algebra. This Thesis is a summary of three Papers (referred to with Roman numerals: **I**, **II**, and **III**), included in Appendices A, B, and C, respectively.

The first contribution, presented in Papers **I** and **II**, proposes an extension of existing VPUs that aims to increase their performance of computing the GEMM kernel. To do so, the baseline VPU is provided with the functionality of an SA, which enables data reuse between functional units. Paper **I** describes how to implement this extension with minimal hardware overhead. To pass the data between functional units, we propose to reuse an interconnect network, which is already available in VPUs to support operations like sliding and reduction. Paper **II** analyzes the implications of this new functionality in the memory system of the original VPU, and more in particular in its VRF. First we analyze how the existing memory instructions cannot efficiently load the data for the new SA-like functionality, and propose a variation of the existing instructions that can better support this functionality. The second contribution in Paper **II** is a partitioning schema analysis that enables us to dynamically choose between the vector and systolic functionalities, maximizing utilization. Evaluating this SA-like extension of VPUs shows up to 4.22x speedup over the baseline vector architecture.

The second contribution, presented in Paper **III**, proposes a scratchpad-based memory architecture that provides VPUs with hardware acceleration for index-matching. This memory architecture transforms the index-matching problem into one of hash lookup and maps it to hardware using cache-like structures over a base multi-banked scratchpad. Consequently, it can efficiently leverage key parallelism, resulting in a scalable memory architecture. Moreover, it can behave in two different modes at a given time, supporting both regular memory accesses and index-matching. This novel memory architecture shows up to 3.19x speedup over the state-of-the-art.

Chapter 2

VSA: A Hybrid Vector-Systolic Architecture

As mentioned in Problem 1 (Section 1.3.1), GEMM can be computed both by a VPU and by an SA. SAs are more efficient, while VPUs offer general-purpose capabilities. However, from a high level of abstraction, both are a set of functional units together with a memory system that can feed them. When it comes to the functional units, the difference is that VPUs organize them in a 1D array, while SAs do so in a 2D array. As for the memory system, VPUs use a VRF to feed their units, compared to the buffers used by SA. This raises the question of what would be needed to extend an existing VPU with the dataflow of a SA, and if doing so would be feasible.

The first contribution of this Thesis tackles this question. It does it in two parts: first, we analyze the computational aspect, implementing the dataflow of an SA into an existing VPU. This corresponds to Paper **I**, and is summarized in Section 2.1. Second, we analyze the memory system, and in particular the VRF, with the goal of efficiently utilizing this component. This corresponds to Paper **II**, and is summarized in Section 2.2.

2.1 Compute

To implement the functionality of a SA for GEMM into an existing VPU, there are two main challenges: (1) emulating the dataflow of the SA with the VPU and (2) feeding the functional units according to the requirements of the new dataflow. Moreover, we need to implement it with minimal hardware overhead to make it feasible.

For the first challenge, the VPU needs to support sending data between functional units instead of communicating explicitly through the VRF. One way to do it would be through chaining, where the output of one vector instruction is used directly by the next one [21]. Chaining is done by allocating the computation of different instructions to different sets of functional units and forwarding the data from one set to the next one. However, while this approach helps reducing latency, it has its own drawbacks, such as requiring extended support for data hazards [19] and having multiple sets of functional units. Hence, modern VPUs tend to support this functionality on a limited way. For example, Ara [25] limits chaining to different types of functional units (integer ALU, integer MUL and Floating Point Unit (FPU)) within the same lane, while Vitruvius+ [20] supports only memory-to-arithmetic chaining. Therefore, we want to look for other ways to emulate the dataflow of an SA. From chaining, we realized that using multiple instructions would likely require increased hardware overhead. Thus, we moved on to analyze individual instructions. There are some vector instructions, like sliding and reduction, that require communication between different functional units. To do so, they require specialized hardware support. For example, Vitruvius+ [20] includes a unidirectional data ring to forward data to the functional units in the following lanes. Moreover, this type of support is common in VPUs, as these operations are widely used (for example, reduction is used to calculate the dot product of two vectors). After analyzing this support, the mapping of the systolic functionality onto a VPU is done by mapping the

rows of the SA onto the different lanes of the VPU, emulating the vertical dataflow using the available data ring. As for the horizontal dataflow, it happens within each lane.

For the second challenge, SA has different data patterns compared to most vector operations. Traditionally, VPUs support three types of memory accesses: unitary (accessing consecutive memory addresses), non-unitary strided (accessing memory addresses with a stride, i.e., a jump between accessed addresses), and indexed (accessing memory addresses relative to the base address, given an offset stored in an input vector register). These three memory access types were presented in order of efficiency, which is opposite to the degree of flexibility they provide. An SA has fixed regular patterns, but not ones that can be represented with unitary or strided instructions. Hence, the implementation presented in Paper **I** uses indexed memory accesses, which are less efficient but can generate the right access patterns. This was a temporary solution, as it was one of the major points addressed in Paper **II**.

To interact with this new functionality, the RISC-V vector extension is further extended with two new instructions: `vsa` and `vfsa`. They are, respectively, the integer and float versions of the SA-like instruction.

This new functionality is evaluated over different Deep Neural Networks (DNNs), achieving up to a 2.25x speedup over the baseline vector functionality. Additionally, in order to analyze the full potential of the new functionality, the evaluation includes a set of results based on the supposition that using unitary memory accesses instead of indexed ones would be possible. With this supposition, the maximum performance improvement seen increases to 3.5x. Moreover, this approach also fulfills the criteria of having minimum area overhead, being it only 0.1% over an existing VPU.

Besides the specific speedup values, two conclusions are extracted from the evaluation: (1) the systolic functionality can better leverage higher amounts of hardware resources, and, most importantly, (2) the speedup differences are mostly related to how different matrix sizes fit are partitioned by different functionalities, leading to under-utilization of the vector registers. This second point is the second major point analyzed in Paper **II**.

2.2 Vector Register File

As mentioned in Section 2.1, Paper **I** left two major points to analyze: (1) how to efficiently access memory and (2) how to better utilize the available resources. Both of these problems are affected by a common component: the VRF.

2.2.1 Memory Accesses for SA

For the first point, the problem is that the more efficient types of memory accesses do not support the patterns required by the systolic functionality. Thus, this functionality needs to rely on the less efficient indexed memory access. The reason is that the other types of memory accesses (unitary and strided) cannot describe those patterns. As seen in Paper **I**, having more efficient memory accesses could provide considerable memory improvements, so it is a key point to study. The systolic patterns can be described as a combination of unitary or strided patterns. However, part of the existing strides are not due to the patterns themselves but to how the VRF is accessed by load and store operations. For example, to stream one matrix row horizontally with the SA functionality, it needs to be fully stored in the vector register slice of the corresponding lane. However, when loading or storing data, the accesses to the different VRF slices are interleaved. This constraint is defined by hardware. A similar situation arises when loading the columns to be streamed vertically. This is solved by accessing the VRF in a lane-by-lane fashion. Lane-by-lane accesses enable using the existing and more efficient memory access types for the SA patterns. Therefore, Paper **II** proposes a new set of instructions for the RISC-V vector extension. It duplicates the existing load and store instructions, with the difference that one of the fields is now used to select the target lane.

A new evaluation of the architecture shows up to a 4.22x speedup over the vector baseline, compared to the 2.47x speedup for the same configuration without the new lane-by-lane memory accesses. Moreover, the performance difference between using this new support and not increases with the MVL. This shows that the new instructions can better leverage the locality existing in the systolic patterns.

2.2.2 Utilization Analysis

For the second point, as seen in Paper **I**, the performance gap between vector and systolic functionality is due to their utilization of the VRF. Different functionalities divide the problem in different ways, and thus lead to different levels of utilization. While the VRF is a register file, it has been scaled up to the point of providing similar storage as a cache. However, the VRF is more constrained than a regular cache when it comes to accessing it, making it harder to fully utilize all the available space. To tackle this, Paper **II** proposes a methodological approach to understand how different parameters affect the utilization of the VRF. For a fixed MVL, this depends on two parameters: the functionality used and the sizes of the input matrices. With matrix sizes defined by a (M, N, K) set, the first step is to find the smallest set that pushes both functionalities to their maximum utilization using the Least Common Minimum. Bigger matrices can be partitioned into sub-matrices equal to or smaller than said set. Then, parameter sweeps are performed for M, N and K, reducing their sizes. The result of this process is a discrete 3D volume where each point represents the relative speedup for the different functionalities for a given (M, N, K) set. In this volume, two regions can be identified. Each of the regions represents the points where one functionality outperforms the other. After finding the border between these two regions, it is possible to use this information to partition the problem accordingly, increasing the VRF's utilization. Moreover, as this analysis can be done offline, the runtime overhead is negligible except for really small matrices.

Paper **II** presents an example of the proposed analysis. After performing it and finding the border between the two regions, this information has been added to the GEMM implementation. This enhanced implementation is capable of detecting which of the functionalities will provide better resource utilization at a given time and switching to it accordingly. This enables seeing no performance downgrade due to choosing the worse functionality. Moreover, this is performed with low runtime overhead, being only perceived in really small matrices.

Chapter 3

Scalable Hardware Hash for Index-Matching in Vector Architectures

As mentioned in Section 1.2.2, sparse linear algebra kernels rely on three different operations: indirection, intersection, and union. In VPUs, indirection is directly supported in hardware with scatter-gather operations, but existing ISAs do not provide hardware support for the main component of intersection and union: index-matching. Existing state-of-the-art proposes to implement a small custom memory to accelerate indirection, intersection, and union [17]. It uses a Content Addressable Memory (CAM) to find the corresponding elements, comparing the incoming indices against all the stored ones. Moreover, it uses an Live Value Table (LVT)-based scratchpad as the data array to avoid collision penalties. This approach relies on replication and multi-porting [43, 44]. Due to this reliance, the LVT-based is effective for small memories, but suffers from considerable penalties with increased memory sizes, and especially with an increased number of ports. Nowadays, VPUs and similar architectures like GPUs have larger memories [45, 46]. Moreover, datasets for sparse matrices keep growing [47]. Therefore, we should find a way to efficiently implement larger memories with index-matching support. Outside VPUs, we can look for alternatives among custom accelerators. Many custom accelerators rely on stream-based approaches, which would not fit on current VPUs, but others, like InnerSP [48], use hashing to implement index-matching. Finally, we need to consider that VPUs are general-purpose accelerators. Applications should be able to efficiently use the proposed memory also when not working with sparse data. Thus, we should try to develop a multi-functional memory, not limited to sparse operations.

In summary, Paper **III** tackles the issue of developing a memory architecture that accelerates index-matching for VPUs. This novel memory should provide (1) index-matching functionality, (2) efficient scaling for large memory sizes and number of ports, and (3) flexibility to also support existing memory accesses and multiple datatypes.

To solve this problem we present SH^2. SH^2 builds upon a scratchpad implemented as a multi-banked memory, where each bank contains a portion of the total memory. Each bank has its own index array, which contains the corresponding indices for the values stored in the bank. Each pair of bank and index array behaves like a set associative cache. The index array can contain fewer indices than values fit in the corresponding bank. The banks determine the total memory size, while the index array determines the maximum memory space supported for index-matching. Moreover, at runtime, we can allocate the memory space required for the index-matching functionality. The non-allocated space is used as a regular scratchpad. Consequently, SH^2 can efficiently support regular and index-matching operations at the same time. Hence, SH^2 offers two main modes: Direct Data (DD) and index-matching (IM). The former supports direct accesses to the scratchpad, while the latter offers hardware index-matching support. At any given time, SH^2 can support either of the modes, or both. With this baseline architecture, the index-matching problem is implemented as a hash problem and supported in hardware with cache-like mechanisms. The

main difference compared to a cache is that in index-matching operations a miss means that the corresponding value is a zero.

In addition to the base architecture, SH^2 can reduce collision penalties by leveraging that it is designed to efficiently support multiple datatypes. SH^2 is designed to work with 32-bit wide data. To work with 64-bit data, two adjacent banks are accessed with the same memory address. In the case of the index-matching functionality, the address is provided by the index array after an index hit. This means that only one index array is actually needed, and we have the adjacent index array unused. To reduce collision penalties, when working with wider data, we send two colliding indices to the two adjacent index arrays. Both adjacent index arrays have the same indices stored. If only one of the indices causes a hit, there is only one access to the bank, and thus we do not need to pay the collision penalty. This way, for wider data, we use replication of indices to compare multiple colliding indices in parallel, and exploit index misses to reduce collision penalties.

In addition to the memory architecture, we have proposed an extension of vector ISAs to support the new functionality. With the architecture and the extended ISA, SH^2 shows speedups of up to 3.19x compared to the state-of-the-art, when running more than 1600 real-world matrices. Moreover, even if we try to scale up the state-of-the-art in a more efficient way, SH^2 offers similar performance in less than 7.4% of the area. In addition, the mechanism for reducing collision penalties removes 38.58% of the extra cycles, reducing the CPI for the corresponding index-matching instructions from 1.80 to 1.56.

Chapter 4

Concluding Remarks and Future Work

Vector Processing Units (VPUs) are a good target architecture for linear algebra kernels [4], but application requirements keep increasing [1], so we should strive for higher performance and efficiency. This Thesis aims to show that it is possible to increase the performance of VPUs for these kernels by integrating ideas for custom accelerators. This way, we can get performance closer to custom accelerators while retaining the general-purpose characteristics of VPUs. In this Thesis we have focused on linear algebra kernels, which are widely used in different applications across multiple domains. Papers **I** and **II** show an example of this approach for GEMM, achieving up to 4.22x speedup. Paper **III** presents another example for sparse kernels, in this case achieving up to 3.19x speedup. However, this extension cannot be done sacrificing generality. VPUs are still general-purpose accelerators, and thus area needs to be considered, not based on just the target kernel. There are two approaches for this: minimal overhead, as followed in Papers **I** and **II**, and resource reusability, as proposed in Paper **III**.

For future work, the intuitive approach would be to continue extending VPUs, now for Graph Neural Networks (GNNs). There are already multiple custom accelerator proposals for these applications, both for ASIC and FPGA platforms [49–52], but not yet an extension for VPUs. Hence, this approach is an in-depth specialization for GNNs in VPUs.

An alternative to this bottom-up depth-first approach would be a top-down breadth-first approach where we analyze all applications (or a set of representative ones) and analyze their characteristics. With this analysis, we can classify the different characteristics into a set of groups according to their hardware requirements. Then, for each of these groups, we can design different VPU extensions. Hence, this approach would also lead to a set of vector extensions, but now with a limited number of extensions and covering all applications. By combining all the different extensions in one system, we would have a fully general-purpose system that can also provide specialized hardware acceleration for different applications according to their characteristics.

Bibliography

[1] OpenAI, "Ai and compute," May 2018. [Online]. Available: https://openai.com/blog/ai-and-compute/

[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] T. Nowatzki, V. Gangadhar, K. Sankaralingam, and G. Wright, "Domain specialization is generally unnecessary for accelerators," *IEEE Micro*, vol. 37, no. 3, pp. 40–50, 2017.

[4] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams *et al.*, "The landscape of parallel computing research: A view from berkeley," 2006.

[5] 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.

[6] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *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.

[7] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[8] J. R. Gilbert, S. Reinhardt, and V. B. Shah, "A unified framework for numerical and combinatorial computing," *Computing in Science & Engineering*, vol. 10, no. 2, pp. 20–25, 2008.

[9] R. Yuster and U. Zwick, "Detecting short directed cycles using rectangular matrix multiplication and dynamic programming," in *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, 2004, pp. 254–260.

[10] M. M. Wolf, M. Deveci, J. W. Berry, S. D. Hammond, and S. Rajamanickam, "Fast linear algebra-based triangle counting with kokkoskernels," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2017, pp. 1–7.

[11] G. V. Demirci and C. Aykanat, "Scaling sparse matrix-matrix multiplication in the accumulo database," *Distributed and Parallel Databases*, vol. 38, pp. 31–62, 2020.

[12] 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.

[13] J. Li, F. Wang, T. Araki, and J. Qiu, "Generalized sparse matrix-matrix multiplication for vector engines and graph applications," in *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*. IEEE, 2019, pp. 33–42.

[14] S. R. Gupta, N. Papadopoulou, and M. Pericas, "Accelerating cnn inference on long vector architectures via co-design," in *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2023, pp. 145–155.

[15] V. Le Fèvre and M. Casas, "Efficient execution of spgemm on long vector architectures," in *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing*, 2023, pp. 101–113.

[16] N. P. Jouppi, D. H. Yoon, G. Kurian, S. Li, N. Patil, J. Laudon, C. Young, and D. Patterson, "A domain-specific supercomputer for training deep neural networks," *Communications of the ACM*, vol. 63, no. 7, pp. 67–78, Jun. 2020.

[17] J. Pavón, I. V. Valdivieso, A. Barredo, J. Marimon, M. Moreto, F. Moll, O. Unsal, M. Valero, and A. Cristal, "Via: A smart scratchpad for vector units with application to sparse matrix computations," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 921–934.

[18] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu *et al.*, "The ARM scalable vector extension," *IEEE micro*, vol. 37, no. 2, pp. 26–39, 2017.

[19] K. Asanovic, *Vector microprocessors*. University of California, Berkeley, 1998.

[20] F. Minervini, O. Palomar, O. Unsal, E. Reggiani, J. Quiroga, J. Marimon, C. Rojas, R. Figueras, A. Ruiz, A. Gonzalez, J. Mendoza, I. Vargas, C. Hernandez, J. Cabre, L. Khoirunisya, M. Bouhali, J. Pavon, F. Moll, M. Olivieri, M. Kovac, M. Kovac, L. Dragic, M. Valero, and A. Cristal, "Vitruvius+: An area-efficient RISC-V decoupled vector coprocessor for high performance computing applications," *ACM Trans. Archit. Code Optim.*, dec 2022, just Accepted.

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

[22] Intel, *Intel® Architecture Instruction Set Extensions and Future Features Programming Reference*, Mar. 2020, 319433-038.

[23] ARM, *NEON Programmer's Guide*, 2013, iD071613.

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

[25] 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.

[26] I. A. Assir, M. E. Iskandarani, H. R. A. Sandid, and M. A. Saghir, "Arrow: A risc-v vector accelerator for machine learning inference," *arXiv preprint arXiv:2107.07169*, 2021.

[27] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic linear algebra subprograms for fortran usage," *ACM Transactions on Mathematical Software (TOMS)*, vol. 5, no. 3, pp. 308–323, 1979.

[28] "BLAS (basic linear algebra subprograms)." [Online]. Available: https://www.netlib.org/blas/

[29] P. Colella, "Defining software requirements for scientific computing," 2004.

[30] F. G. Gustavson, "Two fast algorithms for sparse matrices: Multiplication and permuted transposition," *ACM Transactions on Mathematical Software (TOMS)*, vol. 4, no. 3, pp. 250–269, 1978.

[31] L. Buatois, G. Caumon, and B. Lévy, "Concurrent number cruncher: a gpu implementation of a general sparse linear solver," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 24, no. 3, pp. 205–223, 2009.

[32] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks," in *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, 2009, pp. 233–244.

[33] W. Liu and B. Vinter, "Csr5: An efficient storage format for cross-platform sparse matrix-vector multiplication," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, 2015, pp. 339–350.

[34] D. R. Kincaid, T. C. Oppe, and D. M. Young, "Itpackv 2d user's guide," Tech. Rep., 1989.

[35] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey, "Efficient sparse matrix-vector multiplication on x86-based many-core processors," in *Proceedings of the 27th international ACM conference on International conference on supercomputing*, 2013, pp. 273–282.

[36] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop, "A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide simd units," *SIAM Journal on Scientific Computing*, vol. 36, no. 5, pp. C401–C423, 2014.

[37] P. Scheffler, F. Zaruba, F. Schuiki, T. Hoefler, and L. Benini, "Sparse stream semantic registers: A lightweight isa extension accelerating general sparse linear algebra," *arXiv preprint arXiv:2305.05559*, 2023.

[38] H.-T. Kung and S. W. Song, "A systolic 2-d convolution chip," Tech. Rep., 1981.

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

[40] A. Fell, D. J. Mazure, T. C. Garcia, B. Perez, X. Teruel, P. Wilson, and J. D. Davis, "The marenostrum experimental exascale platform (MEEP)," *Supercomputing Frontiers and Innovations*, vol. 8, no. 1, pp. 62–81, 2021.

[41] 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.

[42] Z. Gu, J. Moreira, D. Edelsohn, and A. Azad, "Bandwidth optimized parallel algorithms for sparse matrix-matrix multiplication using propagation blocking," in *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, 2020, pp. 293–303.

[43] S. Dropsho, A. Buyuktosunoglu, R. Balasubramonian, D. H. Albonesi, S. Dwarkadas, G. Semeraro, G. Magklis, and M. L. Scottt, "Integrating adaptive on-chip storage structures for reduced dynamic power," in *Proceedings. International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 2002, pp. 141–152.

[44] A. M. Abdelhadi and G. G. Lemieux, "Modular multi-ported sram-based memories," in *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*, 2014, pp. 35–44.

[45] Meta, "MTIA v1: Meta's first-generation ai inference accelerator," accessed: 2024-02-26. [Online]. Available: https://ai.meta.com/blog/meta-training-inference-accelerator-AI-MTIA/

[46] *CUDA C++ Programming Guide, Release 12.1*, NVIDIA, 2023.

[47] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, pp. 1–25, 2011.

[48] D. Baek, S. Hwang, T. Heo, D. Kim, and J. Huh, "InnerSP: A memory efficient sparse matrix multiplication accelerator with locality-aware inner product processing," in *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2021, pp. 116–128.

[49] M. Yan, L. Deng, X. Hu, L. Liang, Y. Feng, X. Ye, Z. Zhang, D. Fan, and Y. Xie, "HyGCN: A GCN accelerator with hybrid architecture," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 15–29.

[50] M. Yoo, J. Song, J. Lee, N. Kim, Y. Kim, and J. Lee, "SGCN: Exploiting compressed-sparse features in deep graph convolutional network accelerators," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 1–14.

[51] R. Sarkar, S. Abi-Karam, Y. He, L. Sathidevi, and C. Hao, "Flowgnn: A dataflow architecture for real-time workload-agnostic graph neural network inference," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 1099–1112.

[52] P. Chen, P. Manjunath, S. Wijeratne, B. Zhang, and V. Prasanna, "Exploiting on-chip heterogeneity of versal architecture for gnn inference acceleration," in *2023 33rd International Conference on Field-Programmable Logic and Applications (FPL)*, 2023, pp. 219–227.

Appendix A

Paper I

VSA: A Hybrid Vector-Systolic Architecture

Mateo Vázquez Maceiras, Muhammad Waqar Azhar, Pedro Trancoso

2022 IEEE 40th International Conference on Computer Design (ICCD)

VSA: A Hybrid Vector-Systolic Architecture

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

***Abstract*—In order to deliver high performance efficiently, modern processors include dedicated hardware to accelerate different application domains. For example, several recent processors include dedicated Machine Learning (ML) accelerators. However, while adding dedicated hardware improves efficiency compared to general-purpose CPUs, it also requires a larger area, making it unfeasible for smaller devices. Therefore, exploring ways to use the existing hardware for different functionalities becomes desirable in those setups. In this work, we explore the reuse of the components in a Vector Processing Unit (VPU) to offer the functionality of a Systolic Array (SA) for General Matrix Multiplication (GEMM), a kernel extensively used in machine learning, big data, and scientific computing. This hybrid Vector-Systolic Architecture (VSA) can thus support Single Instruction Multiple Data (SIMD) instruction extensions with the VPU functionality and efficiently compute GEMM with the SA functionality. We present an implementation of VSA as a RISC-V co-processor that adds minimal hardware overhead of less than 0.1% compared to a baseline RISC-V implementation with a VPU. In our evaluation using different Deep Neural Network (DNN) models, VSA shows a speedup of up to 3.5x and a reduction in energy consumption of up to 70%.**

***Index Terms*—Machine Learning, DNN, GEMM, SIMD, Vector Unit, Systolic Array.**

I. Introduction

Given the diminishing returns from technology scaling and the power density limitations, processor development is going through a shift from homogeneous multi-cores consisting of general-purpose CPUs to heterogeneous System-on-Chip (SoC) designs, often composed of several domain-specific accelerators coupled with CPU [1]. This approach allows for delivering the required performance within the power budget in an efficient way.

Nevertheless, implementing one accelerator per application domain may be unfeasible for resource-limited devices. In such setups, exploring the use of units that can support different domains is interesting. One such flexible unit is the Vector Processing Unit (VPU), responsible for executing Vector or SIMD instruction extensions.

This work was partially supported by the eProcessor project which has received funding from the European High-Performance Computing Joint Undertaking Joint Undertaking (JU) under grant agreement No 956702. The JU receives support from the European Union's Horizon 2020 research and innovation programme 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 programme under grant agreement No 957197 and the Swedish Foundation for Strategic Research (contract number CHI19-0048) under the PRIDE project

One highly vectorizable set of instructions (kernel) is the General Matrix Multiply (GEMM), a fundamental operation in many applications across different domains, such as Machine Learning (ML), big data, and general scientific computing.

While the VPU can execute matrix operations, several dedicated hardware accelerators have been proposed for GEMM processing [2]–[7]. Some of these GEMM accelerators are based on Systolic Arrays (SA) [8], [9]. SAs are architectures composed of multiple interconnected processing elements (PEs), exploiting the dataflow between the operations. As this reduces the bandwidth requirements, it is beneficial for applications that are memory bound and with a regular memory access pattern such as GEMM, for the PEs are arranged in a 2D layout. SA-based accelerators for GEMM are currently used in several commercial products [10], [11], both in the cloud and at the edge.

The main disadvantage of SAs is that they perform only a limited set of instructions when compared to VPUs. As a consequence, SAs can not be the sole accelerator for applications with diverse demands. For example, while an SA is a very good match for the convolutional layers in a Deep Neural Network (DNN), it can not be used for batch normalization [12]. More flexible processing units such as the VPU are usually used in such situations [13].

To get the advantages of both architectures, there are systems that include both of them, such as IBM's Power10 processors [14] and the Vector and Systolic Accelerator Tiles of the MEEP platform [15]. Nevertheless, the resources required to implement both architectures can be a problem in systems where the area is limited. Thus, one of the architectures may be sacrificed to satisfy the resource constraints.

To avoid leaving any of the architectures out, this paper explores a scheme to reuse the available resources in a VPU to achieve the functionality of a SA. This proposed scheme was motivated by the fact that even though the architectures are different, their main building blocks, from the compute units to the memory system, are equivalent.

To achieve the dual architecture goal, we propose a low overhead novel hybrid Vector-Systolic Architecture, which we denote as VSA. One key characteristic of VSA is that the functionality of the hardware (VPU or SA) can be chosen dynamically at runtime through the use of dedicated instructions. In this work, we show how this can be implemented as a RISC-V ISA extension.

Overall, this work presents the following contributions:

- VSA, a hybrid Vector-Systolic Architecture that can use VPU resources as a SA with minimal hardware overhead.
- An instruction added to the RISC-V Vector extension to provide VSA with software support.
- A quantitative performance analysis of VSA for DNN inference models.

Our evaluation, using well-known DNN models, shows that VSA can be implemented with less than 0.1% hardware overhead and achieves speedups of up to 3.5x with energy savings of up to 70% in the L2 cache due to more efficient memory accesses by the SA when compared to a VPU with the same number of functional units.

II. Background

A. GEMM in DNNs

The GEMM operation is not only the core kernel in DNNs but is also extensively employed in application domains such as scientific computing and data analytics. This work specifically considers inference execution for DNNs, but insights apply to other application domains that employ GEMM.

DNN models are composed of a sequence of multiple layers, and different layer types will be computed differently. Without loss of generality, in this work, we will focus on the popular Convolutional Neural Networks (CNNs) used for image recognition.

The core layers in these networks are convolutional layers, which perform the 2D convolutions of an input feature map with a set of filters. Such operations are commonly implemented as matrix-matrix multiplications [16].

Other layers in these networks are fully connected (FC) layers. In those layers, each input is connected to each node in the FC layer. Matrix-vector multiplications are an efficient way of implementing the FC layer.

Finally, these networks also include pooling layers, which are used to reduce the spatial size of the representation. All elements inside a window are reduced to a single element by either computing the average or maximum value.

B. Vector Processing Unit

Vector architectures are 1D SIMD architectures that operate over sets of data elements loaded into vector registers, i.e., sequential register files, instead of over single-element registers, and thus single instructions operate over whole arrays. To further exploit data-level parallelism, it is possible to implement multiple functional units to compute several elements in parallel [17]. These elements are organized in sets of units known as lanes, as shown in Figure 1. An example of such architecture is VPUs.

Typically, the data used in the vector operations is loaded from the system's cache hierarchy into the VPU's local Vector Register File (VRF) [18]. The maximum number of scalar elements that fit in a vector register is defined by the data width and the Maximum Vector Length (MVL) supported by the VPU architecture. If the vector values are longer than the MVL, the vector has to be partitioned by software, thus increasing the number of instructions.

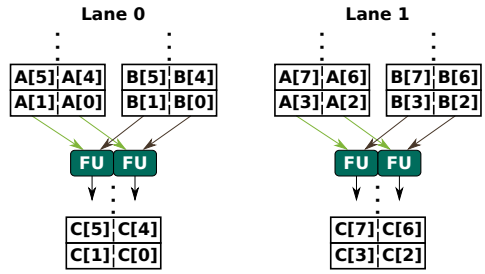


Fig. 1. VPU dataflow for an instruction with two source operands (2 lanes with two units per lane for input datatype)

To support certain instructions (e.g., scatter and gather), VPUs need to include an efficient communication infrastructure between the lanes, such as a data ring [19].

C. Vectorized Acceleration of GEMM

GEMM is a highly vectorizable kernel suitable to be efficiently implemented in a vector accelerator. A vectorized algorithm for computing GEMM with matrices $A[M, K] \times B[K, N] = C[M, N]$ is shown in Algorithm 1 (*.vs* indicates a vector-scalar instruction).

When implementing this algorithm, the programmer needs to be aware that the MVL limits the maximum number of elements a single vector instruction can compute, and implement it in a vector-length agnostic fashion, i.e., that the vectorized code can adapt and scale itself to the MVL of the VPU at runtime.

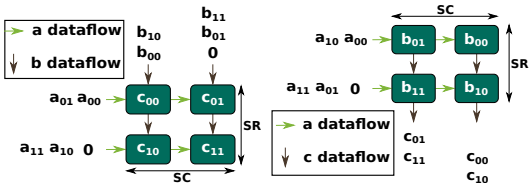
Algorithm 1 Vectorized GEMM

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

D. Systolic Array

Systolic arrays are architectures composed of a set of interconnected processing elements (PEs) [9]. Inputs and outputs happen at the perimeter of the architecture while, inside it, communication is limited to neighbors, with data flowing in a pipelined fashion, leading to a waveform progression. SAs are regular structures with regular communication patterns and are thus a good match for applications that exhibit this same regularity.

One example of an application with such characteristics is the GEMM kernel. To implement a SA for this kernel, the PEs are organized in a 2D fashion. Its dimension determines the size of the matrices for the multiplication operation. Figure 2



(a) Output stationary dataflow (b) Input stationary dataflow

Fig. 2. SAs for GEMM ($A \times B = C$)

shows two commonly used dataflow configurations (output and input stationary) of SAs for GEMM with SR rows and SC columns. In terms of the memory system, in most cases, the SA is closely coupled to scratchpads, which act as buffers [20] for the input data.

In the output stationary dataflow, each PE will contain the partial sum, while the first matrix (A) flows horizontally and the second (B) one does so vertically (Figure 2a). This way, a SR $\times$ K stream of matrix A and a K $\times$ SC stream of matrix B will generate a SR $\times$ SC completed output tile.

In the input stationary dataflow (also known as weight stationary in the field of DNNs), one of the input matrices is stored in the SA, one element per PE (Figure 2b). The other input is streamed horizontally, and each element is multiplied by the one stored in the PE. The partial sums (psum) flow vertically and are accumulated with the multiplication result. This way, having a SR $\times$ SC weight matrix tile stored, we can use a SR $\times$ K input stream to generate an output stream of dimension K $\times$ SC. This stream is made of partial sums, which will be later added to other equivalent streams (unless SR $\geq$ K).

When the matrices are larger than the SA, they are divided into smaller matrices called tiles. However, if matrices (or tiles) are smaller than the SA, the operation will still perform, leading to under-utilization.

III. VSA Architecture

As presented in the previous section, VPUs have the advantage of exploiting data parallelism for a range of different instructions, while SAs are very efficient for computing a specific kernel. While conceptually VPUs are quite different from SAs, fundamentally they are both at their core a collection of computational units. Therefore, we propose to merge them into a common unit. Since VPUs are more general architecture, we propose to use one as the starting point, in order to maintain their generality. Consequently, this work proposes VSA, a hybrid Vector-Systolic Architecture that extends a VPU implementation to offer the functionality of a SA with minimal hardware overhead while keeping all the functionality of the VPU.

To offer the functionality of a SA accelerator for GEMM, we will use the existing functional units available in the VPU for executing the Multiply and Accumulate (MAC) operation. The rows of an SA are mapped to lanes, and the horizontal flow seen in Figure 2 can be achieved inside each lane. The vertical flow will then be achieved between lanes, and thus inter-lane communication network is also needed. For the SA, this communication is unidirectional and only between adjacent lanes. Also, this needs to be done in parallel for all the lanes in order not to add a performance bottleneck. Thus, a network architecture such as a data ring would be preferable to a bus.

Given a VPU with L lanes and with a W-bit width datapath, we can remap it onto a SA of SR $\times$ SC units. The number of lanes L determines the number of rows SR in the SA, while the data type width D and datapath width W determines the number of columns SC. For floating point formats, the maximum number of columns is defined in Eq. 1, while for integer it is defined in Eq. 2. The reason for this is that the result of multiplying two integers requires double the input width. Also, depending on the implementation and accumulator size, a smaller SC may be required, as wider accumulators may further limit the maximum amount of columns we can implement. When describing the architecture, we will consider floating point formats, and modifications needed for working with integers will be explicitly stated.

On the memory side, and in order to keep the design simple, we use the VRF to replace the buffers that typically feed the SA. While improvements could be done to the memory system, this matter is not trivial, and we plan to address it in future work.

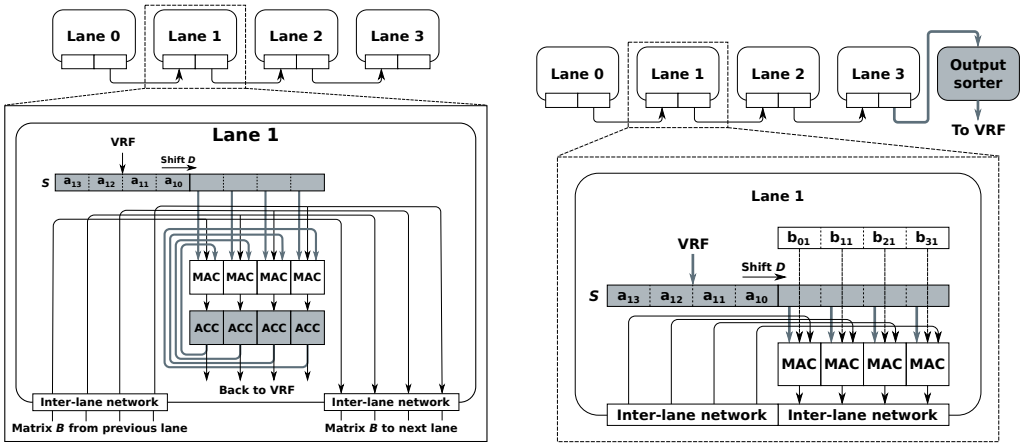
$$\text{SC}_{\text{float}} = \frac{W}{D} \quad (1) \qquad \text{SC}_{\text{int}} = \frac{W}{2D} \quad (2)$$

While the SA executes any GEMM operation, as we focus on applying it to DNNs, we use the DNN terminology and refer to the SA inputs as activations (matrix A) and weights (matrix B).

For VSA, we present two versions, which support different dataflows: VSA-OS and VSA-WS. The implementation of VSA used in the evaluation for this work is VSA-OS, as described later in Section V.

A. Output Stationary VSA

In the output stationary version of VSA (VSA-OS), the data that flows between the lanes are the weights, as shown in Figure 3a. Each lane will contain the accumulators that hold the partial results, which will be fed back into the functional units. These values only need to be sent to the VRF after the tile computation is finished. As for the activations, an option to emulate the horizontal flow would be shifting. We add a register of $2W$ bits (S in Figure 3 and connect half of it to the input of the functional units. The new values are loaded into the other half. To emulate the dataflow, we shift the data D bytes into the half connected to the input of the units. Regarding the weights, we store them in the first lane and make them move through the inter-lane network (Figure 3a). In case an interconnect network is unavailable, or if it would add a non-acceptable delay, it is possible to avoid using it by replicating the weights across all lanes instead of storing them only in the first one. Because vector architectures pay most of the latency per vector load/store, not so much per element



(a) Output stationary dataflow

(b) Input stationary dataflow

Fig. 3. VSA implementation (in grey is the overhead comparing to VPU baseline)

[18], and there is no need to fetch new data from memory, the penalty will not be increased L times.

The extra hardware support needed, as marked in Figure 3a, consists of a $2W$-bit width shift register and W-bit width accumulator per lane.

The output accumulation creates a data dependency that needs to be dealt with when using pipelined fused multiply-accumulate (FMA) units in order to use the full pipeline. To solve this, having a pipeline depth P, the input streams shall be considered as P independent interleaved streams and added together before sending them back to VRF.

Working with integer numbers with VSA-OS would require an accumulator wider than the datapath to avoid values overflowing too easily. Due to this, the accumulation should be done next to the accumulators, with an extra adder, to be able to handle the wider data.

B. Weight Stationary VSA

In the weight stationary version of VSA (VSA-WS), the data that flows between the lanes are the partial accumulations, as shown in Figure 3b. The weights are the values stored next to each functional unit. Each lane is responsible for setting up the weights for the corresponding row. There is no need to add extra registers, as it is possible to use the already existing first register of the weight vector register to store them. The horizontal dataflow is equivalent to the one of the output stationary. If these outputs are partial results, they can be later added with the next set of partial results with a vector add instruction. The extra hardware support needed, as marked in Figure 3b, consists of a $2W$-bit width shift register per lane and an output sorter that sets up the data to be sent back to VRF.

In this case, working with integer numbers would require a way to send partial accumulations wider than the datapath to avoid losing information due to overflow.

IV. Programming VSA

In order to exploit the benefits of VSA at the application level, some software support is required. In this section, we present the software support for the RISC-V implementation being evaluated in this work. While VSA is independent of the processor's ISA, RISC-V has the benefit of offering an open and extensible ISA.

A. RISC-V Extension

In this work, we consider as a baseline a VPU supporting the RISC-V Vector Instruction [21]. For the VSA support, we have decided to extend the ISA by adding a *SA* instruction. The goal was to add support for VSA to have minimal effects over the original system, thus making it easier to integrate and use.

The instruction format, established in [21], is shown in Figure 4. Here, *OPCODE* is the vector opcode, so the instruction is decoded as a vector instruction. *funct6* and *funct3* select the SA instruction and the data-type (integer or float). The data-type width is not included in the instruction. Instead, and following what is done for other operations, it is set up using a different instruction in a control status register. *vs1* and *vs2* contain the values for the horizontal and vertical dataflow respectively, as described in Section III. *vd* is not only the destination register, but it also contains the initial values for the accumulators. This allows us to compute a tile when the MVL is not enough to fit all the required data at once, and it could also be useful to add a matrix to the final result, as is the case of the bias matrix in DNNs.

With this instruction, it is possible to compute the GEMM kernel as described in Algorithm 2. In this algorithm, the loop in Line 1 selects the rows for the corresponding tile, while the one in Line 3 does so for the columns, thus tiling the matrix. Then, each tile is initialized and computed using the new SA instruction. In cases where the output matrix

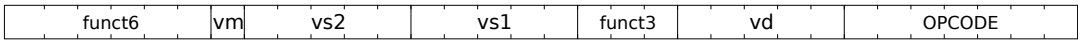


Fig. 4. Instruction format for proposed RISC-V extension

Algorithm 2 GEMM using custom instruction

```
1: for all i ∈ {1, ..., M/SA_R} do
2:     v_r = LOAD_ROW_SET(i)
3:     for all j ∈ {1, ..., N/SA_C} do
4:         v_c = LOAD_COL_SET(i)
5:         v_t = INIT_TILE(i,j)
6:         v_t = SA(v_r, v_c, v_t)
7:     end for
8:     STORE(v_t)
9: end for
```

cannot be perfectly tiled, this algorithm assumes zero padding. With VSA, it is possible to compute the remaining elements using vector instructions, but such further optimization shall be analyzed in future work.

B. Memory operations for VSA

For this instruction to compute the right results, the values need to be in their corresponding position in the vector register, as described in Section III. To load them, three modes are available in the current ISA: unitary strided load, non-unitary strided load, and indexed load. Unitary loads load values in consecutive memory positions starting from a given address, thus having an access increment (or stride) of one. Non-unitary strided loads are equivalent, but with a stride different from one. Indexed loads have an input register that contains, for each element to be read, the corresponding offset with respect to the base address. While the data requirements for the new instruction have specific patterns, none of those patterns are supported by the first two load modes when assuming the matrices are stored in a row-major fashion. As such, without any other changes in the software and hardware, the indexed load is the only available option to load the data for the SA operation.

However, in order to show the full potential of the VSA architecture, in this paper, we also consider the case in which a unitary load can be used. The assumption is that we have a scratchpad into which the runtime packs the data from memory ahead of time into the right pattern required by the SA instruction. This data packing can be done by the runtime as a data pre-fetching phase without incurring any direct latency overhead by overlapping this with other computations.

V. Experimental Setup

In order to analyze our architecture, we have modified the gem5-based VPU simulator [22] presented in [19] to model VSA-O. In the simulated system, the VPU is connected directly to a 1MB L2 cache, shared with a scalar RISC-V core. With this simulator, we have measured the execution time of each GEMM kernel call. We have also used this simulator as the VPU baseline, as our goal was to analyze how much VSA can improve an already existing VPU. Moreover, we have used the generated gem5 stats to estimate power consumption with McPAT [23]. As data type for the GEMM operations, we have used IEEE FP32. To represent the different configurations, we modified the architecture variables (number of lanes and datapath width) accordingly. We have chosen array sizes of 2×2 and 4×4, as bigger sizes are not reasonable in the resource-constrained systems VSA is targeting. As for the MVL, we have evaluated different configurations ranging from 2048 bits (ARM's SVE biggest supported MVL [24]) up to 16384 bits [19]. While we are focusing on small devices and large MVLs could seem unrealistic, they are not unheard of. Moreover, techniques such as vector grouping could be used to solve this problem [21]. Besides, the MVL should be related to the data width. Our experiments with 32b data and MVL of 16384b would be equivalent to experiments with narrower data and shorter MVL (e.g., 8b data and MVL of 4096b).

Over the initial VSA-OS architecture, described in Section III-A, we have implemented two modifications to make it fit better within the original VPU's datapath. First, we removed the accumulators and reused the last pipeline barrier of our pipelined floating point unit to keep the values for one cycle while they are being fed back. Then, instead of shifting the values one by one to simulate the horizontal flow, we decided to broadcast each value to all the units on the lane, which also helped to increase the utilization rate of the SA.

In addition to the gem5 simulation, we have also implemented VSA in RTL and synthesized it in 65nm (STM General Purpose nominal voltage library) for the highest possible frequency (i.e. 333MHz), to measure the area overhead compared to the baseline VPU. This VPU coprocessor contains IEEE-754 compliant FPUs, supporting FP64, FP32, BF16, and CFP8. It also has a data ring as an inter-lane network. We achieved the same maximum frequency for both designs, with the critical path not being affected by the extra VSA hardware support. The synthesis results showed a negligible area overhead of less than 0.1%.

A. Workload

We define each GEMM computation as $[M, K] \times [K, N]$. To compute it, we have implemented Algorithm 1 and 2 by hand coding them using the instructions and data loading as described in Section IV, in a vector-length agnostic fashion.

To evaluate our architecture, we have selected different DNNs models: three well-known image recognition models: AlexNet [25], ResNet18 and ResNet50 [26]; and one model for skin cancer melanoma classification, obtained through the DeepHealth Toolkit [27].

The methodology used for the VSA evaluation is to estimate the performance for the whole model execution based on the

TABLE I
ALEXNET GEMM

	L1	L2	L3	L4	L5	L6	L7	L8
M	96	256	384	384	256	1	1	1
K	363	2400	2304	3456	3456	9216	4096	4096
N	3025	729	169	169	169	4096	4096	1000

performance of the GEMM operations used for each layer of the model. This is a valid approach, as we have measured up to 80% of the time for the complete vectorized application execution being spent on GEMM operations. As such, one intermediary step in this evaluation is determining the input sizes for each GEMM. We obtain these sizes from Darknet [28], after the transformation with the *im2col()* function, taken from the Caffe framework [29], and before calling the GEMM function. To illustrate this, for AlexNet, the array dimensions are shown in Table I.

VI. EXPERIMENTAL RESULTS

We start this evaluation by analyzing the performance for all layers of AlexNet, as shown in Figure 5. This Figure represents the execution time speedup for SA mode when compared to the VPU baseline. Note that we depict two SA mode results, one for the unitary load (SA UNIT) and the other for the index load (SA INDEX), as described in Section IV. For this experiment, we have used a MVL of 16384 bits, which is considered large [19], and a systolic array of 4×4, comparable to other similar implementations [14].

As expected, the last three layers (L6-8), which are FC layers, suffer from a considerable slowdown due to the under-utilization as they are actually a matrix-vector multiplication, as mentioned in Section II. As for the convolutional layers (L1-5), we can see the benefits of using VSA, although that benefit is not the same across all layers. The benefit depends on the different matrix shapes and on how the different configurations can leverage them. In general, the SA configuration shows better performance than the VPU, reaching up to 3.5x speedup over baseline. In order to justify this speedup, we looked more in depth and found out this is because for larger MVLs the vectorized implementation cannot make as good use of it as the SA. For example, in layers L3-5, N, the vectorized dimension is 169, while a MVL of 16384 bits supports up to 512 FP32 values. Thus, for such cases, the VPU mode cannot take full advantage of it, while the SA mode can leverage it better. Another observation is that the largest benefit, as expected, is for the unitary load version, with slightly lesser benefits for the index load version. It is relevant to notice that the index load penalty for Layer-1 is considerable enough that it results in a slowdown of the operation when compared to its execution on the VPU.

The next results depicted in Figure 6 show the design space analysis for the SA when changing the MVL and the number of functional units. In order to highlight the benefit of combining SA and VPU modes in VSA, we present here two sets of bars. The first one depicts the speedup of VSA using only the SA mode, with unitary load (SA UNIT) and with index load (SA INDEX) for all layers of AlexNet. The second one shows the benefit of VSA, where the execution is performed using both VPU and SA modes depending on which one performs better for each layer of the model independently.

We first analyze the results running GEMM SA mode alone. From the results, it is possible to observe that, as the MVL or the number of units increases, the speedup also increases. This means that the SA configuration scales better than the VPU for an increasing amount of resources. However, this is not true for all cases. For a MVL of 2048 bits, we can see how, for the architecture with 16 functional units, the SA does not perform as well as in the smaller case compared to the VPU. This is because a bigger SA has bigger data requirements, and when the MVL does not allow it to fit enough elements, it has more difficulties leveraging the larger number of units.

Next we analyse the complete VSA results (second group of bars in Figure 6). For them, as stated before, we execute the operations of a layer using the VPU or the SA mode, depending on which one achieves the best performance. While we show only the results for the complete execution, the detailed results show that, as expected, VSA can leverage the data reuse offered by the SA for the convolutional layers and use the VPU as such for the FC layers, avoiding the problem of under-utilization. This is known offline, according to each layer, but it can be implemented dynamically based on the matrices' sizes (if $1 \in \{M, N, K\}$), obliviously to the user.

From the results, it is possible to observe that, overall, the hybrid VSA architecture improves the performance of the standalone VPU or SA units. The improvement does not seem to depend on the MVL size and is only more relevant for the configurations with a larger number of functional units. Configurations of 2×2 show improvements of 5-10%, while those with 4×4 units improve 15-25%.

In addition to achieving better performance, energy consumption is also improved at the system level, with the main benefit coming from the memory subsystem, as shown in Figure 7. In this analysis, we focused on the energy savings that VSA can achieve in the L2 cache, as the main benefit of this architecture derives from better memory utilization, and memory is the most expensive part when it comes to the energy budget [30]. The results show potential energy savings for VSA of up to 60% for VSA with unitary loads, while with the use of indexed loads, we observe up to 40% savings.

It is interesting to observe that the index loads have a considerable negative impact on the energy consumption of SA. This correlates with the slowdowns seen in Figure 6. At the same time, these results highlight the benefit of using VSA as with its capability to execute always in the most efficient way, the heavy penalty of the index loads is avoided with the VPU execution, and thus the VSA results shown in the charts always result in energy savings.

In addition to AlexNet, we have evaluated VSA for other DNNs as described in Section V. The results obtained are similar to the ones presented before for AlexNet, as shown in Figure 8. In the case of ResNet18, the potential speedup goes up to 3.5x, with memory energy savings of up to 70%.

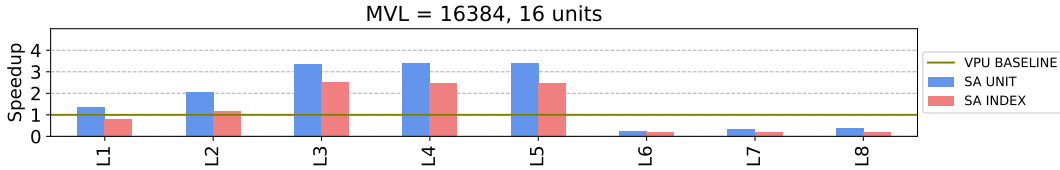


Fig. 5. AlexNet's GEMM calls for all layers

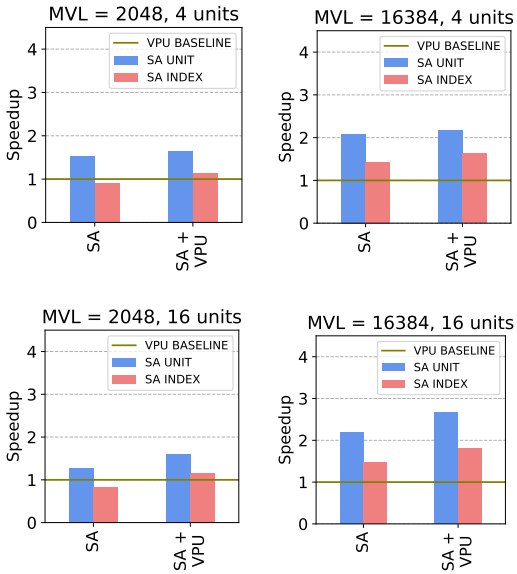


Fig. 6. AlexNet total speedup for different VSA configurations

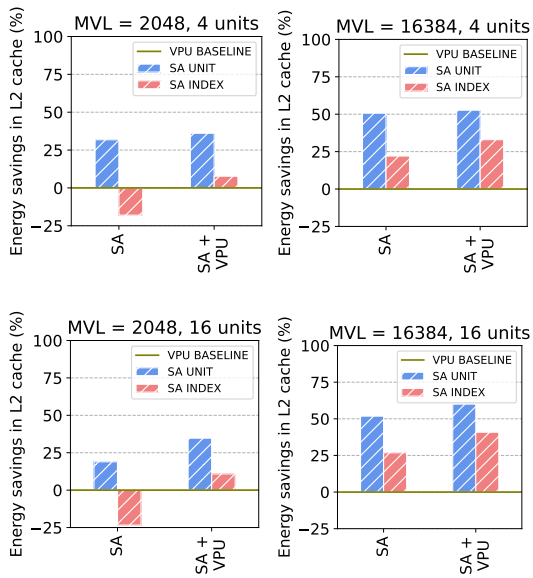


Fig. 7. AlexNet total L2 cache energy consumption for different VSA configurations

VII. Related Work

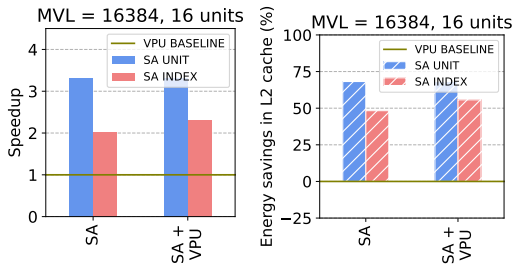
Vector architectures have been traditionally implemented in the supercomputing domain, such as CRAY-1 in the 70s [31]. In recent times, vector support has been added to CPUs, with extensions like Intel's AVX [32] and ARM's NEON [33]. Further support for vector architectures includes ARM's SVE [24] and the RISC-V V extension [21].

Systolic architectures were introduced in the late 1970s. They are used for applications in the fields of signal and image processing (e.g. filtering, convolution and correlation, discrete Fourier transform) and matrix arithmetic (e.g. matrix-vector and matrix-matrix multiplication) [8], [9], [34]. SAs provide modular regular architectures and reduce bandwidth requirements by passing data between neighboring processing units.

Currently, SAs are used in commercial products such as Google's TPU. TPUv1 provides a 256×256 MAC array targeted to inference [10], [35], while later versions provide arrays of 128×128 MACs accompanied by VPUs [13], [36]. Other commercial implementations of SAs are NVIDIA's tensor cores, which consist of 4×4 SAs. [11].

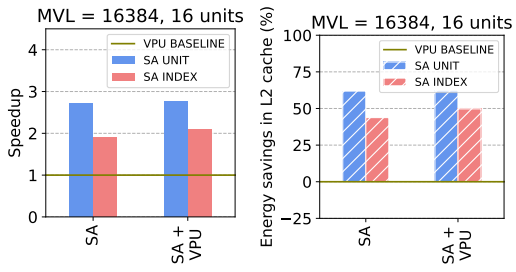
Thanks to their modular architecture, SAs can be designed with the help of generator tools. PolySA [37] provides an end-to-end compilation framework that maps algorithms to SAs using polyhedral models and then generates the design using HLS. Gemmini [20] is a RISC-V-based generator of custom parameterizable SAs for the GEMM kernel.

Acknowledging the benefit of accelerating matrix operations in general, IBM's POWER10 processor includes multiple Matrix Math Accelerators (MMA), arrays of 4x2 array of processing units [14], with software support through Power ISA^{TM} version 3.1 [38]. Instructions are issued to matrix units through a path shared with their corresponding vector unit. Both units in the same slice also share the same vector register file (VRF). Similar to this are the Vector and Systolic Accelerator Tiles of the MEEP platform [15], each of them



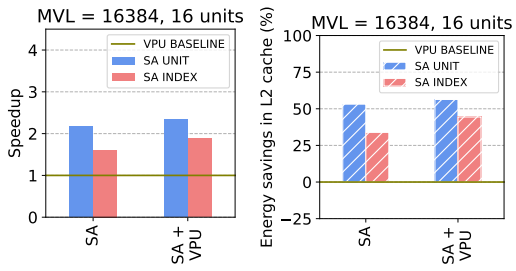
(a) ResNet18 Speedup

(b) ResNet18 Energy



(c) ResNet50 Speedup

(d) ResNet50 Energy



(e) Skin Speedup

(f) Skin Energy

Fig. 8. Speedup and energy for other DNNs

including one VPU and two different SAs.

In summary, both vector architectures and systolic arrays exploit application parallelism. Moreover, to compensate for the lack of flexibility of SAs, several proposals include both architectures together, to the point of sharing their whole memory hierarchy. With VSA, we propose going one step further by adding minimal hardware overhead to existing VPU resources in order to also offer the functionality of a SA.

VIII. Conclusions

In this work, we have presented VSA, a novel hybrid Vector-Systolic Architecture that extends a VPU implementation to offer the functionality of a SA with minimal hardware overhead while keeping all the functionality of the VPU. Using a RISC-V VPU as our starting point, we added a custom instruction to the RISC-V vector extension in order to offer software support. Our evaluation for the execution of different DNN models has shown that VSA can achieve speedups of up to 3.5x in the case of ResNet18, with corresponding energy savings of up to 70% over the VPU baseline. Moreover, we have synthesized the proposed VSA implementation and obtained an area overhead of less than 0.1% without affecting the maximum frequency.

While these results are already promising on their own, we still see the potential for further improvements with VSA. In future work, we will focus on the memory architecture support for VSA, implementing one that fits not only the VPU but also the SA.

Acknowledgment

The authors would like to thank Sonia Rani Gupta and Lars Svensson from the Chalmers University of Technology, as well as the partners from the eProcessor project, for the valuable discussions and contributions to this work.

References

[1] 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.

[2] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *International conference on machine learning*, PMLR, 2015, pp. 1737–1746.

[3] Z. Du, R. Fasthuber, T. Chen, *et al.*, "Shidiannao: Shifting vision processing closer to the sensor," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015, pp. 92–104.

[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] K. Guo, L. Sui, J. Qiu, *et al.*, "Angel-eye: A complete design flow for mapping cnn onto embedded fpga," *IEEE transactions on computer-aided design of integrated circuits and systems*, vol. 37, no. 1, pp. 35–47, 2017.

[6] 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.

[7] J. Fowers, K. Ovtcharov, M. Papamichael, *et al.*, "A configurable cloud-scale dnn processor for real-time ai," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, 2018, pp. 1–14.

[8] H. T. Kung and C. E. Leiserson, "Systolic arrays (for vlsi)," Carnegie-Mellon Univ Pittsburgh PA Dept of Computer Science, Tech. Rep., 1978.

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

[10] 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.

[11] NVIDIA, *Nvidia tesla v100 gpu architecture*, WP-08608-001_v1.1, 2017.

[12] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International conference on machine learning*, PMLR, 2015, pp. 448–456.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] S. Rivoire, R. Schultz, T. Okuda, and C. Kozyrakis, "Vector lane threading," in *2006 International Conference on Parallel Processing (ICPP'06)*, IEEE, 2006, pp. 55–64.

[18] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[19] 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.

[20] H. Genc, S. Kim, A. Amid, *et al.*, "Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration," in *Proceedings of the 58th Annual Design Automation Conference (DAC)*, 2021.

[21] *Risc-v v vector extension*. [Online]. Available: https://github.com/riscv/riscv-v-spec.

[22] C. Ramírez, *Gem5*, 2020. [Online]. Available: https://github.com/RALC88/gem5/tree/develop.

[23] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the 42nd annual ieee/acm international symposium on microarchitecture*, 2009, pp. 469–480.

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

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

[26] 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.

[27] M. Cancilla, L. Canalini, F. Bolelli, *et al.*, "The deephealth toolkit: A unified framework to boost biomedical applications," in *2020 25th International Conference on Pattern Recognition (ICPR)*, IEEE, 2021, pp. 9881–9888.

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

[29] 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.

[30] M. Horowitz, "Computing's energy problem (and what we can do about it)," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, IEEE, 2014, pp. 10–14.

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

[32] Intel, *Intel® architecture instruction set extensions and future features programming reference*, 319433-038, Mar. 2020.

[33] ARM, *Neon programmer's guide*, ID071613, 2013.

[34] H. Kung and S. W. Song, "A systolic 2-d convolution chip," Carnegie-Mellon Univ Pittsburgh PA Dept of Computer Science, Tech. Rep., 1981.

[35] N. P. Jouppi, C. Young, N. Patil, and D. Patterson, "A domain-specific architecture for deep neural networks," *Communications of the ACM*, vol. 61, no. 9, pp. 50–59, 2018.

[36] N. P. Jouppi, D. H. Yoon, M. Ashcraft, *et al.*, "Ten lessons from three generations shaped google's tpuv4i: Industrial product," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, 2021, pp. 1–14.

[37] J. Cong and J. Wang, "Polysa: Polyhedral-based systolic array auto-compilation," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, IEEE, 2018, pp. 1–8.

[38] J. E. Moreira, K. Barton, S. Battle, *et al.*, "A matrix math facility for power isa (tm) processors," *arXiv preprint arXiv:2104.03142*, 2021.

Appendix B

Paper II

Exploiting the Potential of Flexible Processing Units

Mateo Vázquez Maceiras, Muhammad Waqar Azhar, Pedro Trancoso

2023 IEEE 35th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)

Best Paper Award

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.

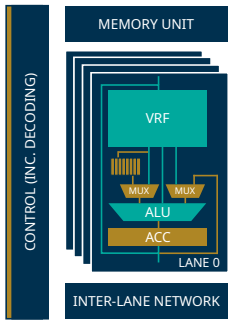


Fig. 1. Block diagram for the VSA xPU. The elements added to the VPU baseline are highlighted in golden color.

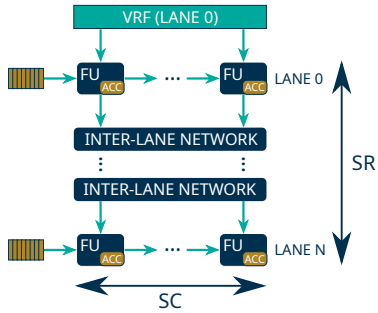


Fig. 2. Behaviour of the SA

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}

```
SET_DATA_WIDTH(D)
v_idx_a = GEN_IDX_A()
v_idx_b = GEN_IDX_B()
v_idx_c = GEN_IDX_C()
for all i ∈ {1, ..., M/SR} do
    v_a = LOAD_IDX(v_a, v_idx_a)
    for all j ∈ {1, ..., N/SC} do
        v_b = LOAD_IDX(v_b, v_idx_b)
        v_c = LOAD_IDX(v_c, v_idx_c)
        v_c = vsa/vfsa(v_a, v_b, v_c)
        STORE(v_c)
    end for
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}

```
for all i ∈ {1, ..., M} do
    v_c = LOAD_ROW(C, M)
    for all j ∈ {1, ..., K} do
        a = A[ i×K + j ]
        v_b = LOAD_ROW(B, K)
        v_c = vfmacc.vs(a, v_b, v_c)
    end for
    STORE(v_c)
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

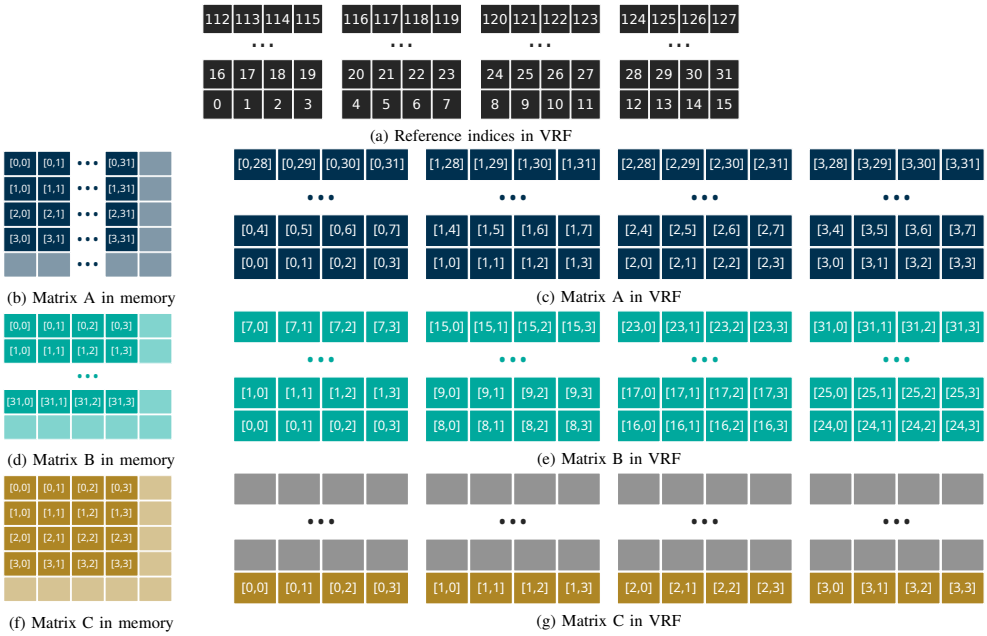


Fig. 3. Memory and VRF patterns for a VPU with 4 lanes, 4 elements per lane and MVL that fits 128 elements

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

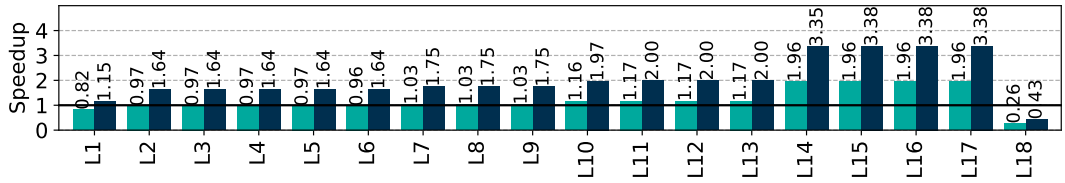
We start by analyzing ResNet18, but the conclusions extracted for the individual GEMM calls also apply to the other applications. The first step is to analyze the performance for all layers of ResNet18, as shown in Figure 5. This figure represents the execution time speedup for running xPU_{SA}, compared 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.

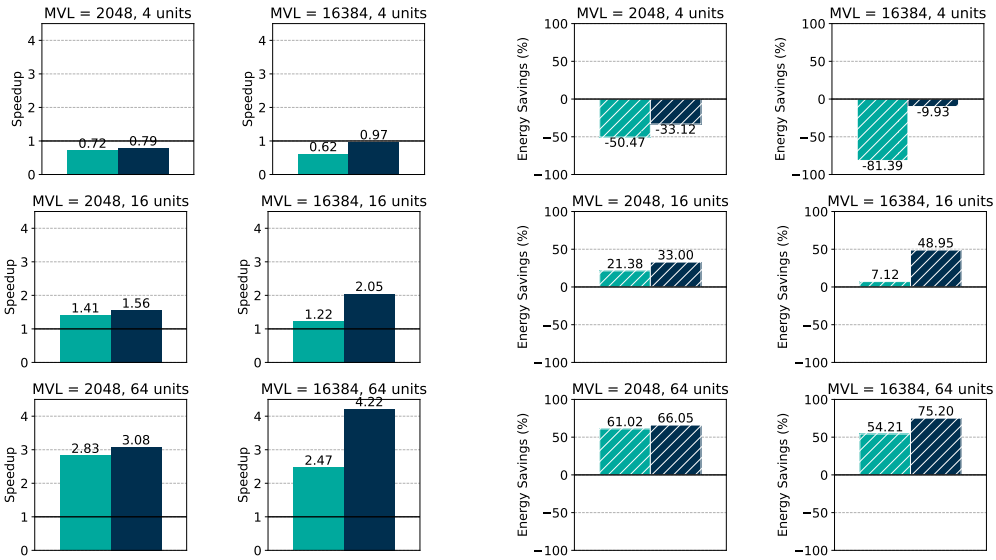
B. Leveraging Heterogeneity

As discussed in Section IV, algorithms for xPU_{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.

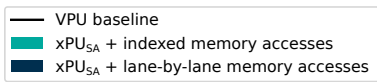


Fig. 5. ResNet18 results including the support of the new memory instructions.

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 under-utilization. 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

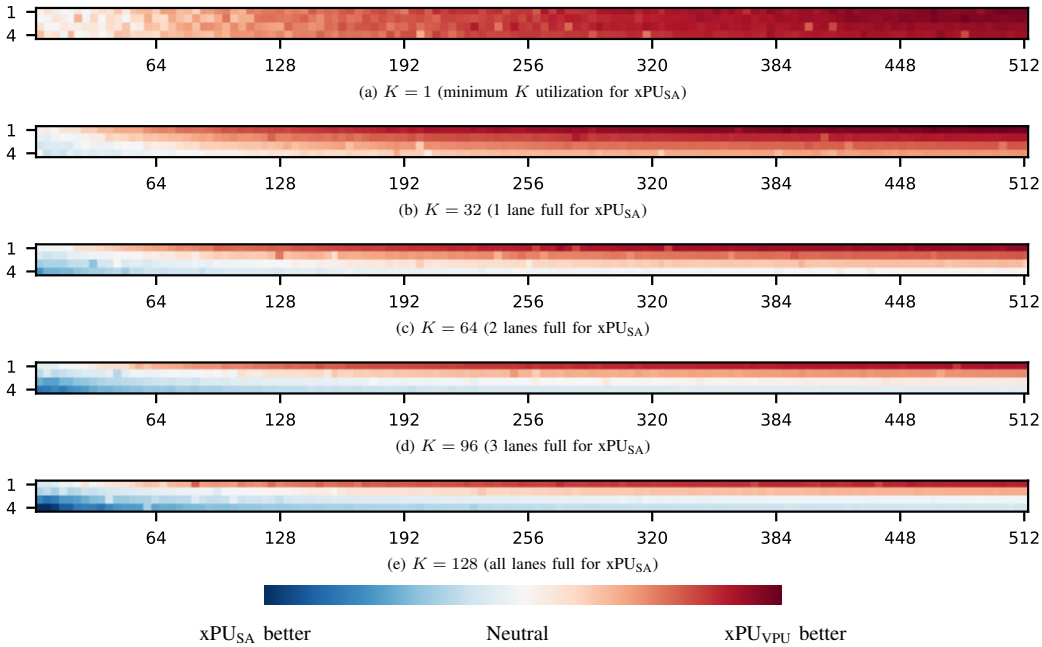


Fig. 6. Tradeoffs between xPU$_{VPU}$ and xPU$_{SA}$ for different pipeline depths. The X-axis represents the different values of N and the Y-axis the different values of M. Configuration of 4×4 units and MVL = 16384 bits.

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

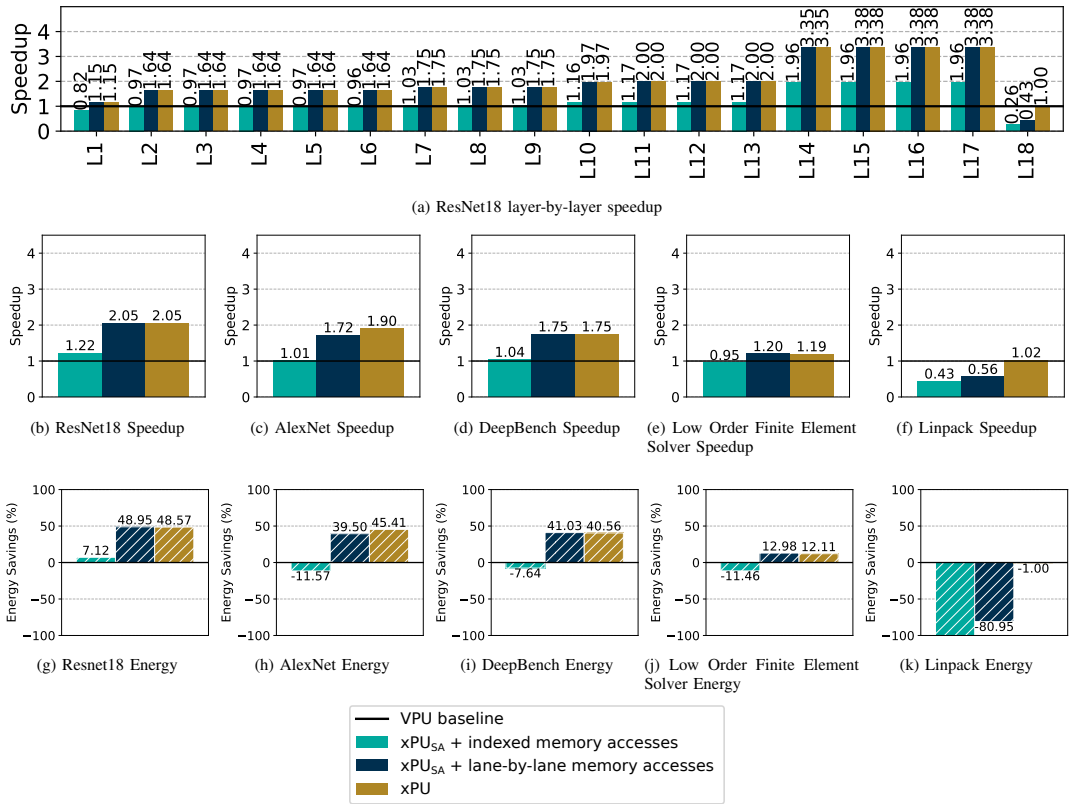


Fig. 7. Speedup and energy savings for tested applications. Configuration of 4×4 units and MVL = 16384 bits.

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 subminimal 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.

Appendix C

Paper III

Scalable Hardware Hash for Index-Matching in Vector Architectures

Mateo Vázquez Maceiras, Mohammad Ali Maleki, Muhammad Waqar Azhar, Pedro Trancoso

Submitted to the *2024 International Conference on Parallel Architectures and Compilation Techniques (PACT)*

Scalable Hardware Hash for Index-Matching in Vector Architectures

ABSTRACT

Sparse linear algebra kernels are widely used in multiple domains like graph applications, Machine Learning and other High Performance Computing applications. While dense kernels operate with regular patterns, sparse kernels rely on irregular indirection, intersection and union operations. Modern architectures, that rely heavily on large scale vector units to deliver the required computational power, suffer the most with these irregular patterns and operations. As such, our goal is to improve their capabilities for computing such kernels. While hardware acceleration for the critical operations (indirection, intersection, and union) has been proposed, these solutions do not scale efficiently.

In this work we propose SH^2, a Scalable Hardware Hash memory architecture for general-purpose vector architectures. SH^2 is based on a multi-banked scratchpad memory, which can be partially used for regular data storage, as well as to perform the index-matching operation with the help of dedicated hardware support. In addition, SH^2 includes mechanisms to mitigate index collisions. The proposed architecture provides better latency while requiring less than 7.4% area compared to existing state-of-the-art. An evaluation using real world matrices and multiple sparse kernels, showed that SH^2 achieves up to 3.19x speedup when compared to the state-of-the-art.

CCS CONCEPTS

• **Computer systems organization → Single instruction, multiple data**.

KEYWORDS

Vector, Scratchpad, Hash, Index-Matching, Sparse Linear Algebra

ACM Reference Format:

. 2024. Scalable Hardware Hash for Index-Matching in Vector Architectures. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques (PACT'24).* ACM, New York, NY, USA, 12 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Sparse linear algebra kernels are key components of many high-performance applications, as evident from its inclusion in the Seven Dwarfs [15] and Berkeley's Dwarfs [7]. Sparse linear algebra is widely used in graph applications, such as graph contraction [26], breadth-first search [26], cycle detection [71], Markov clustering [10, 63], and triangle counting [9, 69]. Sparse linear algebra is also used in databases [19], genome assembly [29], colored intersection [40], sparse deep neural networks [31] and molecular dynamics [38].

Compared to their dense counterpart, sparse kernels work with compressed sparse data, which introduces irregular patterns. Moreover, as the data is compressed to avoid storing and processing zeroes, accesses are more costly as we need to load the metadata to locate the data in addition to the data itself. This results in an increased pressure on the memory system. Depending on the nature of the data and the algebraic operation to be performed, there are three different underlying operations related to accessing the correct data: indirection for sparse-dense, intersection and union for sparse-sparse [62]. While indirection can be directly implemented with gather-scatter instructions, intersection and union are more complex, as they require index matching.

The applications using these sparse kernels are easily parallelizable. Therefore, they target systems that provide support for SIMD instructions, such as CPUs with wide vector units, and also SIMT, like GPUs. Most modern CPUs include SIMD or vector units which are exposed as ISA extensions [5, 36, 61, 66].

Aiming to accelerate sparse applications in vector architectures, VIA [59] proposes to use a small Smart ScratchPad Memory (SSPM) memory next to the vector core, which relies on a Content Addressable Memory (CAM) for index matching. Additionaly, it uses a Live Value Table (LVT)-based memory as its scratchpad to reduce collision penalties [2, 23]. While this approach is effective for small memories, it does not scale well as its area and latency costs explode for bigger configurations as it relies on redundancy and each bank requires multiple read ports. However, proposals for custom accelerators that support hardware-accelerated index matching have included memory systems capable of better scaling. One such example is InnerSP [11], which uses a multi-banked memory to implement a hash table. This hash table performs index-matching by handling indices as keys. It supports parallel hash key lookup and in-place accumulation. Nevertheless, as a part of a custom accelerator, the hash tables only need to handle limited functionality and can be fine-tuned for it. They do not need to efficiently support different memory operations, instruction streams, and datatypes at the same time. All these characteristics are required in a general-purpose architecture. Moreover, InnerSP does not tackle the issue of collisions, with multiple indices trying to access the same hash table at the same time.

Aiming to bridge this gap, and in order to **provide vector architectures with area- and latency-efficient index-matching hardware support** for intersection and union operations, we propose SH^2, a Scalable Hardware Hash memory. The key insight is to develop a memory architecture that combines the functionality of VIA's SSPM and the scalability of InnerSP's Hash Table. In SH^2 we transform the index matching into a hash lookup operation. We propose augmenting a regular scratchpad memory with hardware support for index-matching that is scalable to serve requests from wide SIMD units. In order **to reduce the collision penalty, the index-matching hardware can exploit index misses to offer**

PACT'24, October 13–16, 2024, Long Beach, California, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM

https://doi.org/XXXXXXX.XXXXXXX

increased parallelism, without significantly increasing the complexity and overhead. Finally, **the SH^2 scratchpad space can be simultaneously used directly by the application as a regular scratchpad and for storing the index-matching data**. The space reserved for index-matching can be allocated dynamically at runtime. Without loss of generality, in this work we focus on vector architectures. The same technique could be applied as well to SIMT architectures, i.e., GPUs.

The main contributions of this work are the following:

- A scalable scratchpad memory architecture that is extended to support hardware hashing for intersection and union operations over sparse data.
- An approach that leverages index misses and data widths to handle bank collision penalties.
- A set of instruction extensions to a vector Instruction Set Architectures (ISAs).
- An analysis of the proposed approach, including a study of hash space utilization using real world data.

The evaluation of SH^2 for different sparse kernels using both synthetic and real sparse data shows a speedup of up to 3.19x over existing state-of-the-art solutions. This is achieved with a reduction of at least 7.4% in area. Moreover, leveraging index misses enables a reduction of 38.58% in the penalties caused by collisions.

2 BACKGROUND AND RELATED WORK

2.1 Working with Sparse Linear Algebra

In order to save space and avoid computations with zero-values, sparse data is usually stored in a compressed data format [13, 14, 30, 41, 43, 46, 47]. While using such compressed formats for sparse data leads to more efficient kernels, the regular memory accesses are replaced by three operations: indirection, intersection and union [62].

Indirection occurs when trying to fetch the elements of a sparse array from a dense array. This can be achieved by employing indexed memory accesses, using dense arrays with the indices of the sparse one. This can be implemented using scatter-gather [59].

In both intersection and union operations, both arrays store sparse data. As both arrays are compressed, finding a corresponding element implies comparing the different indices until finding it. If no matching element is found, then it is a zero. Hence, both intersection and union are implemented with index-matching. The distinction between the operations lies in handling zero and non-zero pairs. In intersection, only positions with both arrays having a non-zero remain as a non-zero. In union, as long as one of the arrays contains a non-zero for a given position, the corresponding output will be a non-zero. Examples of intersection and union are element-wise sparse multiplication and sparse addition, respectively.

2.2 Memory Systems for Vector Architectures

In order to accelerate sparse linear algebra kernels, one common approach is to offload them to general-purpose accelerators, like CPUs with vector units or GPUs. As the architectures work with multiple data in parallel, their memory systems need to provide support for parallel read and write operations. While multi-ported memories are costly, multiple interleaved memory banks can be used as cost-effective alternative for implementing parallel memory systems [6]. Therefore, both vector engines and GPUs leverage this kind of memory architecture [1, 48, 49, 67]. Ideally, each access from a SIMD/SIMT operation would be served by a different memory bank [56]. This happens with the unitary-stride memory accesses, widely used in dense linear algebra. When it is not the case, a common approach is to partition the batch of accesses into a number of smaller batches that do not contain bank collisions [1]. Serving these accesses is also not a trivial issue since any SIMD access may be directed to any bank. A crossbars network is usually used to facilitate the bank accesses [49].

To support indirection, intersection and union in vector architectures, VIA [59] proposes a SSPM that provides hardware support for these operations. The SSPM is divided into three parts: (1) an index-tracking array, (2) a valid bitmap, and (3) the data scratchpad. The valid bitmap tracks the stored elements of the dense array during intersection. The index-tracking array stores the indices of all the elements of the sparse array stored in the data array. To perform index-matching, the SSPM behaves like a Content Addressable Memory (CAM). It compares the incoming indices against all the stored ones and checks if there was a successful match through `OR` reduction. While effective, this approach results in considerable hardware overhead in the form of comparators and `OR` gates. Lastly, the SSPM's data array uses the Live Value Table (LVT) technique to emulate a monolithic multi-ported memory. This is done to mitigate the constraints associated with multiple write ports [23]. Instead of having one bank with P write ports and P read ports, it uses P banks with 1 write port and P read ports. As it leverages replication, each bank is of the same size of the memory it models. The LVT keeps track of the bank where the last element for each address was written. On a read, the LVT decides for each of the P outputs, which of the values coming from the P banks is the right one. With this approach, while the memory architecture can handle multiple elements without collisions, it incurs considerable area and latency cost. Thus, this approach becomes impractical for implementing memories capable of handling higher levels of parallelism.

2.3 Hashing for Index-Matching

Among the kernels that rely on index-matching, General Sparse Matrix-Sparse Matrix Multiplication (SpGEMM) stands out as one of the most relevant ones.. Most implementations of SpGEMM build upon Gustavson's algortithm [30], which relies on union to accumulate sparse rows, i.e., union. Performing this accumulation efficiently is one of its major challenges [55]. To address this challenge, various software accumulator strategies have been proposed, including using fully dense rows [25, 30], heaps [8], merge [16, 27, 28, 39], sort [12, 17, 44], and hash maps [4, 20–22, 24, 45, 51–53, 58]. The latter, hash maps, is one of the most commonly used approaches in the state-of-the-art for performing union operations in SpGEMM. Compared to other alternatives, hashing has been shown to have better memory usage [58, 63]. Nevertheless, hash maps suffer from certain inefficiencies. The memory accesses are still irregular, consuming a considerable portion of the execution time [45, 53, 72]. Moreover, with intersection and union being memory-bound operations [62], these overheads further add to the memory bottleneck.

To alleviate this bottleneck, one approach is to use hardware acceleration for hashing [32, 42]. This has two advantages: (1) it

Table 1: Comparison with prior scratchpad/caches for sparse linear algebra

Memory	SIMD	HW-Accelerated Index Matching	Hybrid Sparse-Dense Functionalty	Collision Penalty Reduction	Efficient Scaling	ISA Support	Multiple Datatypes
NVIDIA's Shared Memory [1, 49]	✓	X	X	X	✓	✓	✓
Vortex' High Bandwidth Cache [67]	✓	X	X	X	✓	✓	✓
ASA [72]	X	✓	X	X	✓	✓	X
InnerSP's Hash Table [11]	✓	✓	X	X	✓	X	X
VIA's SSPM [59]	✓	✓	X	✓	X	✓	X
SH^2 (this work)	✓	✓	✓	✓	✓	✓	✓

reduces the bandwidth requirements by doing the lookup near-memory/cache, with the compute unit sending the key and receiving the actual value, and (2) it reduces hash lookup time by checking multiple keys in parallel, aiming for O(1) time complexity. Nowadays, there is great focus on leveraging cache characteristics for hashing [54, 70, 74]. Focusing on sparse linear algebra, one major example of this cache-based hashing approach is ASA [72], which accelerates sparse accumulation in SpGEMM for a scalar core. It uses a private cache, transforming the key lookup into a tag lookup. However, it can only handle one key at once. Looking into custom accelerators, InnerSP's Hash Table [11] goes one step further and supports handling multiple incoming keys in parallel. This architecture uses a multi-banked approach, similar to the ones used by memory architectures for vector architectures and GPUs, and thus could be a good alternative to VIA's LVT when it comes to efficient area and latency scaling. However, as a part of a custom accelerator, the hash tables only need to handle limited functionality and can be fined tuned for it. Working with a general-purpose architecture, they need to efficiently support different memory operations, instruction streams, and datatypes at the same time. Moreover, InnerSP does not tackle the issue of collisions, with multiple indices trying to access the same hash table at the same time.

2.4 Stream Architectures

Besides the hash-based approach, other works have proposed a stream-based approach for enabling efficient index-matching. This approach is common in custom hardware accelerators, such as OuterSPACE [57], ExTensor [33], MatRaptor [65], Gamma [73], and Spaghetti [35]. Looking into ISAs-based architectures, Rao *el al.* [60] propose a stream-centered ISA designed to handle sparse tensors as streams. Going one step further away from specialization, Scheffler *el al.* [62] extend a scalar core with hardware index-matching capabilities by providing architectural registers with streaming capabilities. While streaming is an alternative for index-matching, existing vector architectures are not designed to work with streams in the way index-matching requires.

3 ARCHITECTURE

3.1 Base Architecture

SH^2 builds upon a scratchpad implemented as a multi-banked memory, where each bank contains a portion of the total memory. Each bank (ⓗ) is accompanied by a private index array (ⓒ), which stores

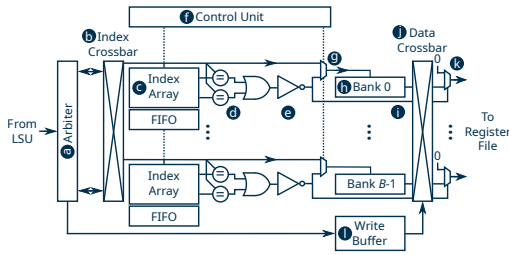


Figure 1: SH^2 Architecture, with $L = 2$.

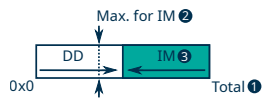


Figure 2: Memory partitioning in SH^2: ① represents the total memory available, ② represents the total memory available for index-matching, and ③ represents the memory allocated for index-matching at a given time.

the indices for the corresponding values, as shown in Figure 1. This pair of bank and index array behaves like a set associative cache with a block size of one value (e.g., 4B when designed to work with FP32). Each position of the index array has a corresponding *valid* bit, that signal if the corresponding index is valid. The size of the index array (②) determines the maximum memory that can be allocated for index-matching, while the total memory available is determined by the bank size (①) (see Figure 2). The allocated memory space for index-matching (③) may be re-sized dynamically at runtime through the exported user-level interface. Hence, ① ≥ ② ≥ ③. The index-matching allocation is handled in hardware through a control unit (Figure 1, ⓕ), which notifies the index arrays about the allocated space. This control unit is also in charge of controlling the multiplexers that bypass the index array to directly access the banks ⓖ. In addition, the control unit contains performance counters that inform about the utilization of the index arrays. With this setup, **SH^2 offers two main modes: Direct Data (DD) and index-matching (IM)**. The former support direct accesses to the

scratchpad, while the latter offers hardware index-matching support for the intersection and union operations. **At any given time, SH^2 can support either of the modes, or both**, as shown in Figure 2. The different memory regions are accessed using different instructions, as explained in Section 4.1. To prevent different modes from accessing the same memory region, the region working in DD-mode starts at address 0x0, while for IM-mode the index array is in charge of providing inverted addresses, i.e., starting at the highest memory address. This inversion is transparent to the user (ⓔ), as in the IM-mode all addresses to the banks are handled by the hardware. The software only needs to provide the indices to match against. In addition, each index array includes a FIFO queue that contains the addresses of inserted elements. These queues facilitate data retrieval from the IM-region using instructions that do not provide indices, without needing to check empty positions. When retrieving data from the IM-region without doing index-matching, each FIFO queue provides the address of the next inserted element. If a particular FIFO queue is empty, this will be treated as a zero, and the corresponding mask bit will be set. This is controlled by the control unit (ⓕ), which simply forwards the corresponding *read* bit signal to the FIFOs. This data retrieval is required as some keys may need pre-processing before going to memory. For instance, when working with matrices stored in CSB format, each value within a block has two indices: row and column. In this case, the software needs to combine these indices into a unique key to properly interact with the index-matching functionality, as only one index per value is supported. Upon data retrieval, these indices have to be restored to properly indicate the position of the element in the original data structure. To read the data without indices, the addresses are read from the FIFO, providing the associated value and resetting the *valid* bit. The status of all the FIFOs is sent to the control unit (ⓕ), which contains a performance indicating whether there is any data left in the IM-region.

For improving energy efficiency, the index arrays can also be bypassed when not being used to perform index-matching operations (Figure 1, ⓖ). When are not accessed, they are clock-gated to reduce energy consumption. Moreover this setup could also enable pipelining of index array and bank accesses. Consequently, the maximum frequency would be increased, and the DD accesses would require only a single cycle at this higher frequency, whereas the IM access would need two cycles.

3.2 Hashing Mechanism

With the described architecture, the memory region allocated for IM-mode accelerates index-matching with the following hashing mechanism: first, the number of buckets (i.e., groups of elements with the same hash value) is determined by the total memory size allocated for index-matching (③) and the number of hash lookups performed in parallel, L. The former can be changed at runtime, while the latter is an architectural parameter. Buckets are evenly distributed across the banks, with elements of the same bucket stored in the same bank. In this manner, indexing to a specific bank is the same as applying the modulo of the number of banks B as the first hash function. Likewise, within each index array, indexing to the specific bucket acts as a second hash function. Here, a parallel L-hash lookup is implemented in the same manner as an L-way set associative cache check. The main difference between a regular cache and this index-matching mechanism is the way misses are handled. In a regular cache, a miss requires fetching the element from higher-level memory. In SH^2, a miss means that the corresponding element is zero, which is the value returned.

With this mapping, the process of performing intersection and union operations begins by loading the data to be matched against into SH^2. This does not have to be the full sparse array, but it can be a subset. This is valid as long as the subset guarantees that the incoming indices can only find a match within that subset. Once the data is preloaded, the indices can start arriving from the Load-Store Unit (LSU). The arbiter (Figure 1, ⓐ) dispatches these indices to their corresponding ports and queues them if collisions occur (see Section 6.3 for details). The indices are then sent to their corresponding index array through a crossbar (ⓑ). At the index arrays (ⓒ), each incoming index is compared in parallel against the indices stored in the corresponding bucket. If a matching index is found, the corresponding address is sent to the bank (ⓗ) to fetch the correct element. Otherwise, a zero is returned, utilizing a *zero* bit wire directly from the comparators (ⓘ), bypassing the bank and resulting in further energy savings. Then, a second crossbar (ⓙ) is used to redirect the values and the *zero* bit back to the Vector Register File (VRF). This crossbar acts symmetrically compared to the address crossbar (ⓑ) and is also controlled by the arbiter. Finally, a multiplexer (ⓚ) selects whether the data stored in the register comes from the bank or is a hard-coded zero, based on the 'zero' bit. Multiplexing after the data crossbar, and not before, (1) minimizes switching in the crossbar and (2) enables to reduce penalties due to collisions, as discussed in Section 3.4. In addition, for the elements that where found to be zero, the corresponding mask vector register should be set/reset (according to the ISA) to enable masking out operations with zeros.

For storing index-value pairs, the process is the same as for reading a value. The difference is that, in case of a miss, the index-matching logic provides the address of an empty position. To provide the address, the index-matching logic behaves like a cache writing a block. If the corresponding index is found, its address is the one provided, so that it can be overwritten with the new value. If not, the index gets assigned an empty address. The address is sent to its corresponding bank (ⓗ) for storing the value. These values are queued in a write buffer (ⓛ), which forwards one of them per cycle and bank, syncing with the index-matching functionality. An exception occurs when there is no matching index and no space left in a bank. In such cases, SH^2 raises a hardware exception, which should be handled in software. For handling this exception, the user or compiler has two options: (1) allocate more space for the IM-region, or (2) modify the problem partitioning, so that the stored sub-array may fit. After handling the exception, the computation is restarted. We do not support the option of sending overflows to memory. This is because the penalty paid during reading would be considerable and not easily hidden, as the computing units will be waiting for new data. This penalty occurs each time there is a miss with the first L elements, regardless of whether the corresponding index is actually located in memory. Instead, the exception will be raised just a few times per problem until the right partitioning is found, and most times in the first blocks [20]. To avoid these exceptions being raised, some works using hash-based approaches use

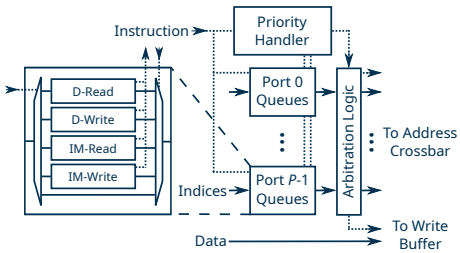


Figure 3: SH^2 Arbiter

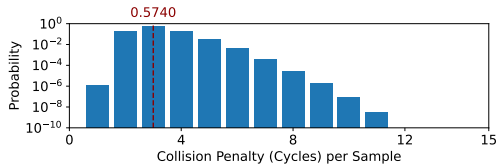


Figure 4: Results of the Monte Carlo simulation for estimating collisions in a memory with 16 banks (1 billion samples)

pre-scanning approaches that estimate the required size of the hash *a priori* [4, 11]. However, we still need to be able to handle potential overflows, as pre-scanning does not guarantee hash fitting.

3.3 Collision Handling

A major issue with this approach, where each bank has a single port and contains a portion of the total memory, is when multiple elements need to access the same bank at the same time. Bank collisions can be addressed by either adding more ports or serializing the accesses. However, both approaches incur penalties, with an increase in area, energy, and latency in the case of adding ports or an increase in energy and latency in the case of serialization.

In SH^2, we propose a multi-queue schema to handle collisions, managed by the arbiter shown in Figure 3. When a new memory instruction arrives, the priority handler inside the arbiter decides whether the incoming addresses/indices can immediately proceed. If they cannot proceed, they are stored in a queue for later processing. There are four available queues, one for each type of operation: two for DD-mode and two for IM-mode, with separate read and write queues for each of the modes. For write operations, the data is forwarded to the write buffer (Figure 1, ⓛ). If one of the queues is full, the arbiter instructs the load-store unit to hold operations of that type until space becomes available. In case of collisions, the arbiter serializes accesses to the banks. After a collision, non-issued addresses/indices are issued from their corresponding queue in the next cycle. An exception to this occurs when the queued operation is a write and there is an incoming read operation to the opposite memory region (DD or IM) arrives. In such cases, reads take priority to minimize latency. If the read operation is of the same type (DD or IM) as the write operation, the incoming read is delayed until all writes are completed to avoid a Read-After-Write (RAW) hazard. This is managed by the priority handler in the arbiter (Figure 3).

3.4 Reducing Collision Penalties

Before presenting how we can reduce collision penalties, it is necessary to discuss the data widths we are targeting. Previous works like VIA [59] are designed for handling 64-bit wide data, common in many scientific applications. Working with 32-bit wide data would lead to memory under-utilization, as only at most half of it could be used. However, we intend to design SH^2 as a flexible memory architecture, developed not exclusively for scientific applications but also capable of efficiently handling various memory access modes and data types. For this purpose, similar to NVIDIA's Shared Memory, we design each of our banks to be 32-bit wide [1, 49, 56]. This approach allows us to efficiently support applications that work with 32-bit, including both floating point and integer types (e.g. deep learning applications). Narrower data-types are supported, but they lead to inefficient memory utilization. In order to work with 64-bit data, two adjacent banks are accessed with the same address, in both DD- and IM-mode. With this setup, for performing index-matching with 64-bit data, only half of the index arrays are needed. In this configuration, one index array generates the address for two adjacent banks, leaving half of the index arrays unused. However, we can leverage this to reduce penalties caused by collisions. When performing write index-matching operations with 64-bit data, we store the index in both adjacent arrays. This way, both arrays keep track of data stored in their corresponding adjacent banks. Later, upon a collision in a read operation, each index array receives and handles a different index in parallel. If both index arrays get a hit, access to the adjacent banks is serialized, incurring the collision penalty. However, if at least one of the indices misses, this missed value can bypass the bank via the *zero* signal (Figure 1, ⓘ), freeing the bank access for the other index. This value will then go through the data crossbar (ⓙ) towards the corresponding multiplexer ⓚ. Past this data crossbar, collisions are no longer an issue. Consequently, we can utilize the multiplexer ⓚ to transmit a zero through the relevant datapath, without having to handle collisions. With this approach, we can leverage sparsity misses to reduce collision penalties. Moreover, this maps well with the nature of the data. Applications working with 32-bit wide data, like the ones in the field of Deep Learning (DL), have densities up to 10% [34], while scientific applications working with 64-bit data have densities reaching orders of magnitude lower than 1% [37]. Therefore, we are supporting collision reduction for the less dense applications, which are the more likely to produce the index misses that we exploit.

With this setup, the question arises: why not use narrower banks to potentially further reduce collision penalties? There are two reasons against this: (1) its penalties and (2) it may not be necessary. Narrowing the banks incurs two penalties: (1a) the increase of the crossbar size (Figure 1, ⓑ and ⓙ) due to accessing more banks, and (1b) the area increase due to lower area efficiency ratio (memory cell area / total area) of the banks. Regarding its necessity, we add as many index arrays per bank (or group of banks) as there are incoming elements, aiming to address the worst-case scenario. However, this scenario is a quite specific case, where the incoming indices have a stride of B. A Monte Carlo simulation with 1 billion

Table 2: Instructions supported to interact with the index-matching functionality

Instruction	Description
vim_ld vd, vs1	Performs an index-matching operation to read the values corresponding to the indices stored in vs1.
vim_st vs1, vs2	Performs an index-matching operation to write the values in vs2, based on the indices stored in vs1.
vim_init rs1, rs2, rs3	Loads rs3 index-value pairs into the IM-region of SH^2. Indices are initially found in consecutive memory positions starting at address rs1 and values starting at address rs2.
vim_evict_simd vs1, vs2	Reads the first inserted value of each bank, storing it in vs2. The corresponding index is stored in vs1, and removed from the FIFO queue.
vim_evict rs1, rs2	Stores the valid elements of the IM-region back in memory, starting at address rs2, with the corresponding indices being stored starting at address rs1. In addition, the FIFO queues are cleared.
vim_rst	Clears the index data from the IM-region, as well as the FIFO queues.
vim_alloc rd, rs1	Allocates rd memory positions for IM-mode, the immediate larger configuration from the rs1 requested positions. This instruction also resets the index array and the FIFO queues.
vim_updv	Updates the *valid* bit according to the *matched* bit.
vim_val rd	Reads from the performance counter in the control unit the number of valid index-value pairs stored in SH^2. The value is stored into rd. This can be used to analyze utilization.
vim_vval vd	Equivalent to **vim_val**, but per-bank. Stores the number of elements of each bank in the corresponding position in vd. Shall be used to analyze data balance across banks.

samples for a memory with 16 banks shows that most of the cases derive into lesser collision penalties (Figure 4), with the highest probability being of 0.5740 for three cycles of penalty. Cases with penalties 2 and 4 cycles are the next most likely, with probabilities slightly lower than 20%. Moreover, while this Monte Carlo simulation uses random data with no regularity, real matrices exhibit some regularity, leading into lower collisions, as will be observed later in Section 6.

In summary, we consider 32-bit wide banks a proper trade-off between flexibility and area, being data width used in many applications. In addition, it fits our needs for reducing collision penalties in the 64-bit wide sparse scientific applications. While narrower widths could be supported, like 16 bits for working with FP16 or BF16, we should take into account the penalties this would incur, and realize that there is a limit to how many collisions we can reduce. Therefore, we decided to continue with 32-bit banks.

4 USING SH^2

4.1 ISA Support

In order to use SH^2, the corresponding vector Instruction Set Architecture (ISA) needs to be extended to support its functionality. While DD-mode operations are already supported in existing vector ISAs with unitary, strided and indexed memory access instructions, IM-mode operations are not. Therefore, to cover the new functionality, we use the instructions described in Table 2. The added instructions are generic, and can be applied to any vector ISA. Instructions **vim_init** and **vim_evict** can be used with formats in which the index can be used immediately as a key, like CSR. These instructions do not require the data going through the VRF For other formats that require key pre-processing (e.g., two indices for a given element), the corresponding instructions **vim_st** and **vim_evict_simd** are used to send the data to the VRF, so that the indices/keys can be pre-/post-processed. In our case, compared to VIA [59], we do not employ hybrid memory-compute instructions. We do so to avoid instruction space explosion, being closer to the RISC paradigm, and also to fit within vector units with parallel memory and arithmetic issue queues, like Vitruvius+ [48].

4.2 Modes and Operations

4.2.1 Index-Matching. For both intersection and union, we use the instructions presented in Section 4.1. For both operations, having the data we want to match against stored in the IM-region, we start by using the **vim_ld** instruction to perform an IM-read operation. To intersect both arrays, we just need to apply the mask generated by that instruction, effectively discarding the unmatched elements. For performing union, we use the **vim_st** instruction to store the data back into the IM-region after performing the required computations. Doing this effectively merges both arrays, completing the union. In addition, it updates the positions shared between both arrays, with no need for further merging. Another advantage of this approach is that there are algorithms like SpGEMM that rely on consecutive accumulations of sparse arrays, i.e., consecutive unions of partial sparse arrays. Writing back to the IM-region enables using it as an accumulator memory, without having to load the data into this region for every union operation. Finally, we use the **vim_evict** or **vim_evict_simd** instructions to retrieve the data from the IM-region. Although this may result in unsorted elements, hash-based approaches already apply random permutations to the data [4]. Hashing is unaffected by data order, so working with unsorted data presents no problem.

4.2.2 Direct Data. As a scratchpad for general purpose vector architectures, SH^2 also needs to behave as a regular storage. This functionality is covered by the DD-mode, where the DD-region of SH^2 behaves like a regular scratchpad, bypassing the index arrays. This way, SH^2 ensures compatibility with existing software implementations. In this mode, it is the responsibility of the user or compiler to ensure the accessed addresses are valid.

Algorithm 1 Vectorized CSR SpAdd ($C = A + B$)

```
1:  Notations:
2:  rows_A : set of all the rows of matrix A
3:  GET_NEXT_SIMD(): load the next SIMD elements from regular
    memory into the vector register file.
4:
5:  vim_alloc rd() rs1(-1)
6:  for all row ∈ rows_A do
7:      vim_rst
8:      vinit_im rs1(B.indices[B_row_start]),
9:               rs2(B.data[B_row_start]),
10:              rs3(B_row_end − B_row_start)
11:     row_len = A_row_end − A_row_start
12:     while row_len > 0 do
13:         v_idx_A = GET_NEXT_SIMD(A.indices)
14:         v_data_A = GET_NEXT_SIMD(A.data)
15:         vim_ld vd(v_data_B), vs1(v_idx_A)
16:         v_data_C = v_data_A + v_data_B
17:         vim_st vs1(v_idx_A), vs2(v_data_C)
18:         row_len = row_len − SIMD
19:     end while
20:     vim_evict rs1(C.indices[end]), rs2(C.data[end])
21: end for
```

4.3 Kernel Example

To exemplify the use of the proposed instructions to compute a sparse linear algebra kernel, Algorithm 1 describes a CSR-based implementation of sparse matrix addition (SpAdd). As the addition operation returns a non-zero as long as one of the inputs is a non-zero, SpAdd is a case of union. Looking at the Algorithm, it starts with line 5 allocating the maximum memory space for the index-matching functionality (we use the maximum space for simplicity). Then, for each row of A, line 7 resets the IM-region and lines 8-10 initialize it with the corresponding row of B. The algorithm then continues by processing the elements from A in batches according to the SIMD length of the architecture. To do so, it (1) reads the indices and data of A (lines 13-14), (2) finds the corresponding indices of B with the index-matching functionality (line 15), (3) performs the element-wise addition in the vector units (line 16), and (4) stores the results back into the IM-region (line 17), merging them with stored matrix B, i.e., completing the union. Finally, upon completing the corresponding row, it is stored back to memory by being evicting the data from the IM-region (line 20).

5 EXPERIMENTAL METHODOLOGY

5.1 Simulation Setup

To simulate the proposed architecture, we implemented a custom time-accurate memory simulator in C++. This simulator contains two main components: (1) the memory models and (2) an index generator. The memory models are time-accurate models of the different memory configurations evaluated, with timing values taken from CACTI [50, 64, 68]. The index generator acts as a functional simulator, executing the kernels and providing the different memory models with the corresponding memory traces. The objective of this work's evaluation was to assess the proposed technique using a large number of real-world input datasets. As such, we needed to develop a lightweight, but still accurate simulation, instead of the more traditional approach using a full-system cycle-accurate setup. This approach allows us to accurately model the execution time for the sparse operations. We run the same algorithms for both SH^2 and VIA. Moreover, VIA's SSPM and SH^2 are interchangeable, working with the same memory traces. Consequently we are able to determine the impact of the proposed technique for applications where those kernels are dominant.

For the specific architecture parameters, we take inspiration from NVIDIA's architecture. The modeled memories have 32 ports (with an equal SIMD parallelism) and a total data array size (Figure 2, ①) of 256kB, as per H100's Unified L1 Data Cache and Shared Memory [3]. In our architecture, the memory size that can work in IM-mode is the same as the total memory (② = ①). In addition, for our tests, we allocate the maximum memory possible for the relevant mode. We undertake this approach to facilitate a direct 1-on-1 comparison with the baseline. Regarding the degree of parallelism for the hash-parallel lookup, we evaluate a configuration of 8 parallel lookups (i.e., $L = 8$), drawing inspiration from ASA [72].

Regarding the baseline, we compare SH^2 to three different configurations of VIA's SSPM: (1) the configuration evaluated in their paper, with 4 ports; (2) a scaled-up version of the original work, using one LVT-based memory with 16 ports to match our configuration for 64-bit data; and (3) a combination of 4 SSPMs with 4 ports each to provide a total of 16 ports, with an arbiter that dispatches requests and a crossbar to forward data to the corresponding SSPM. During the rest of the paper, we will address these configurations as VIA_4, VIA_{16}, and $VIA_{4\times4}$, respectively. All these configurations offer a total of 256kB of architectural memory, i.e., memory seen by the user. In addition, these configurations are designed to work with 64-bit wide data. Note that $VIA_{4\times4}$ is no longer a pure SSPMs implementation: each port can only access the memory positions of its corresponding SSPM, which may lead to collisions.

To measure the area and access latency, we have modeled the memory components using CACTI [50, 64, 68]. For SH^2, we have modeled it as a L-way multi-banked associative cache, with 4B block size. To compensate for the FIFO queues in terms of area, we use a model with a wider tag than needed for the index array. For VIA's SSPM, we first model it as a regular CAM. Then, we determine the overhead related to the index-matching logic by deducting the area and latency associated with the data array. Finally, we model the individual banks of the LVT-based scratchpad and combine them with the corresponding index-matching overhead. Comparing this approach with the original paper [59], our area results are 80% of the target area for the combination with 16kB and 4 ports. This is reasonable, as we are not modeling the LVT that the SSPM uses. Furthermore, by adopting this approach, we are favoring the baseline. Regarding the frequency, the original paper claims 1-cycle access working with a 2GHz processor. With our modeling approach, we have measured an access latency of 0.480ns, which fits within the original configuration. Moreover, when in doubt, we took the decision that would favor the baseline over our proposed memory architecture. For the $VIA_{4\times4}$'s crossbar, we modeled memories ranging from 1MB to 32kB, all of which require the same network configuration. Then, we applied regression to estimate the area and latency overhead of this component. The

obtained R^2 coefficients for area and latency are 0.999 and 0.967 respectively.

5.2 Workloads

To test our approach, we have used both synthetic arrays and real-world matrices from the SuiteSparse Matrix Collection [18]. The synthetic arrays enable us to analyze the behavior of our approach in a more controlled environment, while the matrices from the SuiteSparse Matrix Collection allow to evaluate real-world patterns. The synthetic arrays have densities between 0.01% to 30% and a total amount of 60k elements, including zeroes. The synthetic data is generated with a random uniform distribution. Each element is 32-bit wide. For evaluating 64-bit wide data, we use real-world matrices. We have selected square matrices containing real numbers with a row count greater than 1000, resulting in 1681 matrices evaluated.

The applications used to evaluate our system are sparse kernels that test indirection, intersection, and union, for matrices stored in the CSR format. They are described in Table 3. For the synthetic arrays, to mitigate potential biases or anomalies introduced by the random data, we run each of the synthetic tests 1000 times. For SpMV, we store the dense vector in SH^2 / VIA's SSPM, which means it is limited to 32768 elements (256kB working with 64-bit data). Therefore, for this specific test, we limit our matrices to up to 32768 columns, so that the full dense vector can fit within SH^2/VIA. Regarding our approach to reduce penalties due to index collisions, we evaluate it over SpGEMM. To ensure appropriate matrix sizes for our experiments, our setup operates on a matrix and its transpose. With many of them being symmetric matrices, SpAdd and SpMul will not generate the index misses that our approach exploits. Therefore, we evaluate the collision penalty reduction only over SpGEMM and not over SpAdd or SpMul.

6 EVALUATION

6.1 Implementation Evaluation

Table 4 shows the area and latency for SH^2, as well as for the different baseline configurations. The first value that stands out is the area for VIA$_{16}$. As an LVT-based scratchpad, it relies not only on multi-banking, but also on replication. Each of the banks requires to provide the same memory capacity as the architectural memory (256kB in our case). Therefore, while behaving as a 256kB scratchpad, it actually requires 4MB. Moreover, each of those 16 banks has only one write port but requires 16 read ports, which further increases area consumption per bank [64]. All this translates into a access latency of 3.909ns, limiting its frequency to less than 500MHz. In the case of VIA$_4$, the area required is considerably smaller. Even though each of the banks requires to provide the same amount of memory, there are only 4 of them. In addition, they only require 4 read ports, so the area is considerably smaller. However, SH^2 requires only 7.4% of its area. When it comes to latency, it is also reduced in comparison to VIA$_{16}$, although not to the same extent as the area reduction. VIA$_{4\times4}$ manages to achieve both better area and considerably better latency, while still offering the same number of architectural ports. **SH^2 outperforms the baselines as it eliminates the need for data replication and multiple physical ports acting as a single architectural port**.

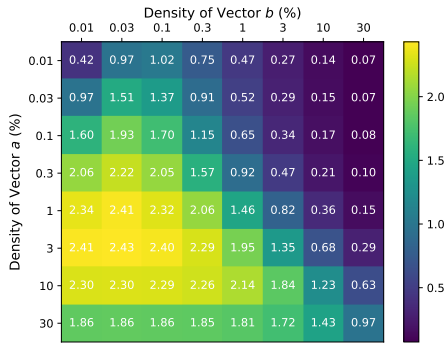


Figure 5: Average extra cycles per instruction due to collisions compared to ideal memory

Consequently, SH^2 achieves 13.4% faster access time compared to VIA$_{4\times4}$ while requiring only 6.4% of its area.

6.2 Kernel Performance Evaluation

6.2.1 Synthetic Data. To evaluate the performance of SH^2 for index-matching, we first use synthetic arrays. We try different combinations of densities for A and B, as illustrated in Figure 5. This figure shows the average penalty caused by index collisions in SH^2. Here, B is the vector stored in SH^2, while the indices used to look for a match belong to A. On the one hand, by increasing the density of the vector stored in SH^2, the penalty incurred starts to increase, reaching a peak at 0.1% density. However, as we keep increasing the elements, the impact of collisions is reduced. This is because we are writing the elements successively with the **`vim_init`**, balancing the write queues from Figure 3. On the other hand, by increasing the density of the vector used to match, we also see that the performance of SH^2 starts to degrade. However, we cannot balance the queues because read instructions need to be computed immediately to minimize the read latency. Therefore, we only start to see improvements at a density of 30%, where some regularity starts to appear in the arrays, and thus fewer collisions take place. To solve this differences across the diagonal, there are two possible (and complementary) options: (1) analyzing the incoming arrays before computing and storing the one with a larger number of non-zero values in SH^2, and (2) utilizing hardware/software implementations that can hide the reading latency. In summary, for the index-matching functionality, the queues should be kept balanced to reduce collision penalties. This balance can be affected by both the data and the implementation, so both should be considered.

6.2.2 Real-World Data.

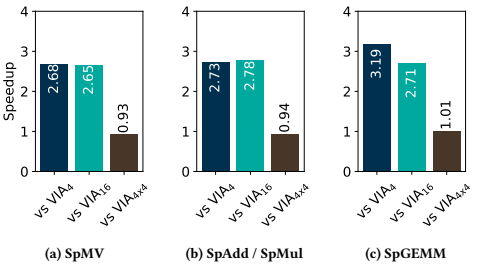
SpMV. Figure 6a shows the average speedup that SH^2 achieves across all real-world matrices for SpMV compared to the three different baselines. In these tests, as the vector stored is fully dense, its access pattern is totally regular, with a stride of 1, and thus there are no collisions while storing it. However, as it needs to be stored only once for the whole computation, this does not greatly

Table 3: Summary of kernels evaluated

Kernel	Sparse Operation	Arrays	Data	Operation
SpMV	Indirection	Sparse-Dense	Real-world matrices (64-bit)	$A \times b$
SpAdd/SpMul	Union/Intersection	Sparse-Sparse	Synthetic arrays (32-bit)	$a + b$; $a \odot b$
			Real-world matrices (64-bit)	$A + A^T$; $A \odot A^T$
SpGEMM	Union	Sparse-Sparse	Real-world matrices (64-bit)	$A \times A^T$

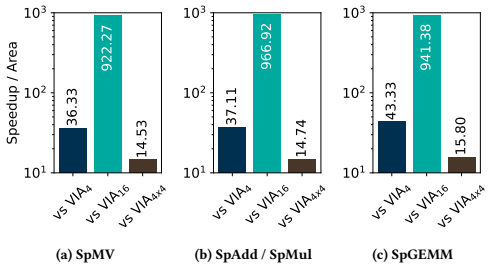
Table 4: Area and maximum frequency for baseline and our work, for 256kB of architectural memory. For SH^2, $L = 8$.

Memory	kB/Bank	Ports/Bank	#Banks	Area (mm^2)	Latency (ns)
VIA$_4$	256	4R/1W	4	9.320	1.637
VIA$_{16}$	256	16R/1W	16	238.698	3.909
VIA$_{4\times4}$	64	4R/1W	16	10.745	1.087
SH^2	8	1(R/W)	32	0.686	0.941



(a) SpMV (b) SpAdd / SpMul (c) SpGEMM

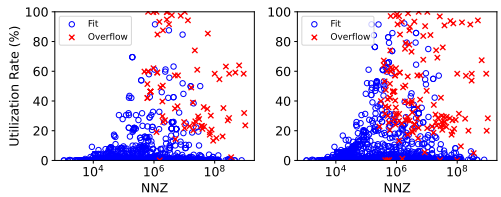
Figure 6: Speedup results with real matrices



(a) SpMV (b) SpAdd / SpMul (c) SpGEMM

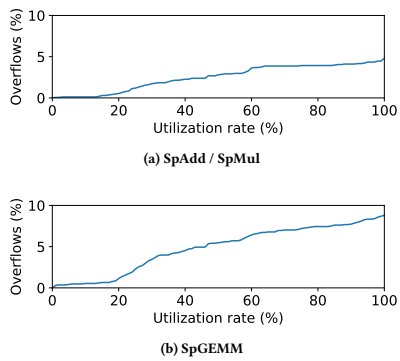
Figure 7: Speedup per area results with real matrices

impact the results, with the indirection directed by the sparse matrix dominating the total time. Compared to VIA$_4$, we see that SH^2 achieves a 2.68x speedup. Looking at the latency (Table 4), we see that the 1.77x difference is not enough to justify that speedup. The



(a) SpAdd / SpMul (b) SpGEMM

Figure 8: SH^2 utilization



(a) SpAdd / SpMul

(b) SpGEMM

Figure 9: Overflow rate analysis

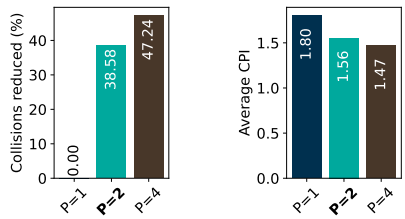
remaining difference is due to SH^2 offering 16 ports, while VIA$_4$ offers only 4, thus causing more port collisions. In the case of VIA$_{16}$ we find the opposite: the speedup (2.65x) is lower than the latency difference (4.15x). This is because the execution on VIA$_{16}$ does not suffer of any collisions. Nevertheless, the cost does not overcome the benefit, and thus SH^2 provides higher performance. Finally, SH^2 is not able to outperform VIA$_{4\times4}$, being 7% slower. This is because the 15.5% latency improvement is not enough to overcome the reduced collisions. However, once we take area into account, SH^2 clearly outperforms VIA$_{4\times4}$, demonstrating a 14.53x speedup per unit of area, as shown in Figure 7a.

SpAdd and SpMul. Computing the same SpAdd and SpMul kernels with real world matrices shows similar speedup results to those

for SpMV, as can be seen from comparing Figures 6a and 7a with Figures 6b and 7b respectively. This is because the major difference between SH^2 and VIA, besides area and latency, is how they deal with collisions. This is for both DD-mode (SpMV) and IM-mode (SpAdd and SpMul). Hence, if the input matrices causing collisions are identical, we can anticipate similar speedups across the tests. Especially, in the case of these kernels, the index comparison is performed in the same order, and led by the same matrix (e.g., A in Algorithm 1), resulting in fewer differences between both kernels.

Besides actual speedups, for the SpAdd and SpMul kernels the interesting insight is how the hash function affects the memory utilization for different matrices. To illustrate that, Figure 8a shows the scattered utilization data of all the matrices tested, ordered in the X-axis by the Number of Non-Zeroes (NNZ). Each of the points on it represents the SpAdd and SpMul tests of a given matrix. Blue points represent matrices whose tests fit successfully in SH^2, and they show the maximum memory utilization achieved (i.e., the ratio between elements stored and the total number of elements that could fit). Red crosses represent tests for which, at some point, a new element could not be inserted, causing an overflow in the corresponding bucket. In that case, the Y coordinate represents the utilization at the moment the overflow took place. As can be seen, most of the matrices require less than 20% of the available memory. This is true not only for matrices with few non-zeros but also for matrices with millions of non-zeros. However, we can see a few matrices with higher utilization rates, reaching close to 90% Regarding the matrices with overflows, most of them fail with utilization rates of 40% or higher. The ones close to 100% represent no problem for the hashing mechanism. Solutions would be to allocate a larger space for IM-mode if possible or further partition the problem so that it fits. This would also apply to VIA's SSPM: overflows will still occur if the total required number of elements exceeds capacity, even if any element can occupy any position. The problem for the hashing mechanism are the matrices that fail at lower utilization rates. However, as seen in Figure 9a, only less than 0.5% of the matrices fail with less than 20% utilization, and less than 2.2% do so with less than 40% utilization. With those low ratios, we shall simply handle them as outliers, which ought to be computed without hardware index-matching support.

SpGEMM. For SpGEMM, the speedup results shown in Figures 6c and 7c compared to the three baselines are still similar to SpMV, SpAdd, and SpMul. However, we can see that, specially for VIA$_4$ and VIA$_{4\times4}$ the speedup is slightly better. This is due to our mechanism for reducing collision penalties, which we will analyze in more detail in Section 6.3. When it comes to the utilization of SH^2, we can see in Figure 8b that the average utilization is higher than in the case of SpAdd and SpMul. This occurs because each output row is formed by an accumulation of multiple input rows, thereby exhibiting a higher density than the previous kernels. This also impacts the amount of bucket overflows, which increase for the same set of matrices, as seen in Figure 9b. However, despite this increase in overflows, only 0.8% of the matrices fail with less than 20% utilization. This means once again that only a few outliers will have to be handled in software, without the index-matching hardware support.



(a) Cycles reduced **(b) Average CPI for reads**

Figure 10: Effects of reducing collision penalties. P represents the number of indices handled in parallel for a same bank. $P = 2$ is the actual configuration used for the tests with real matrices.

6.3 Reducing Collision Penalties

We now analyze in more detail our approach to reducing index collisions for IM-read instructions. As mentioned in Section 5.2, we evaluate it over SpGEMM, whose results in Figures 6c and 7c already include this support. For this kernel, our implemented configuration that checks just two elements in parallel per bank set ($P = 2$) reduces read penalty collisions by 38.58% (Figure 10a). To demonstrate that simply increasing the number of index arrays does not necessarily lead to proportionally higher returns, we also model a setup where we could accommodate 4 indices arriving at the same bank (or set of adjacent banks). By doing so, we would only remove 8.66% more collision penalties than by handling two indices at a time. These results show that just handling one collision per index array gives most of the benefits, being a lower requirement than the one we could have initially extracted from the Monte Carlo analysis in Section 3.3. This happens for two reasons: (1) data in real matrices is not uniformly randomly distributed, with some denser regions that reduce collisions, and (2) our approach can be exploited when at most one index hits, which is less likely to happen with more indices being analyzed in parallel. Compared to a setup without support for collision penalty reduction ($P = 1$), our approach reduces the average CPI from 1.80 to 1.56, as shown in Figure 10b.

7 CONCLUSIONS

In this paper, we introduce SH^2, a Scalable Hardware Hash memory architecture for general-purpose vector architectures. Its main goal is to accelerate the performance of sparse operations, with a special focus on index-matching. For that, SH^2 adds to a regular multi-banked scratchpad memory, the hardware support for hash-lookup. The proposed memory architecture is flexible allowing the scratchpad space to be used for either regular data storage or index-matching. Moreover, SH^2 reduces the hardware complexity compared to state-of-the-art, resulting in speedups of up to 3.19x while requiring at most 7.4% of the area. In addition, SH^2 also supports a mechanism to mitigate the impact of collisions in the index-matching operations, resulting a in reduction of the collision penalty by 38.58% compared to standard multi-banked memories without such support.

REFERENCES

[1] Tor M Aamodt, Wilson Wai Lun Fung, Timothy G Rogers, and Margaret Martonosi. 2018. *General-purpose graphics processor architectures*. Springer.

[2] Ameer MS Abdelhadi and Guy GF Lemieux. 2014. Modular multi-ported SRAM-based memories. In *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*. 35–44.

[3] Michael Andersch, Greg Palmer, Ronny Krashinsky, Nick Stam, Vishal Mehta, Gonzalo Brito, and Sridhar Ramaswamy. [n. d.]. NVIDIA Hopper Architecture In-Depth. https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/

[4] Pham Nguyen Quang Anh, Rui Fan, and Yonggang Wen. 2016. Balanced hashing and efficient gpu sparse general matrix-matrix multiplication. In *Proceedings of the 2016 International Conference on Supercomputing*. 1–12.

[5] ARM. 2013. *NEON Programmer's Guide*. ID071613.

[6] Krste Asanovic. 1998. *Vector microprocessors*. University of California, Berkeley.

[7] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, et al. 2006. The landscape of parallel computing research: A view from berkeley. (2006).

[8] Ariful Azad, Grey Ballard, Aydin Buluc, James Demmel, Laura Grigori, Oded Schwartz, Sivan Toledo, and Samuel Williams. 2016. Exploiting multiple levels of parallelism in sparse matrix-matrix multiplication. *SIAM Journal on Scientific Computing* 38, 6 (2016), C624–C651.

[9] Ariful Azad, Aydin Buluç, and John Gilbert. 2015. Parallel triangle counting and enumeration using matrix algebra. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. IEEE, 804–811.

[10] Ariful Azad, Georgios A Pavlopoulos, Christos A Ouzounis, Nikos C Kyrpides, and Aydin Buluç. 2018. HipMCL: a high-performance parallel implementation of the Markov clustering algorithm for large-scale networks. *Nucleic acids research* 46, 6 (2018), e33–e33.

[11] Daehyeon Baek, Soojin Hwang, Taekyung Heo, Daehoon Kim, and Jaehyuk Huh. 2021. InnerSP: A memory efficient sparse matrix multiplication accelerator with locality-aware inner product processing. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 116–128.

[12] Nathan Bell, Steven Dalton, and Luke N Olson. 2012. Exposing fine-grained parallelism in algebraic multigrid methods. *SIAM Journal on Scientific Computing* 34, 4 (2012), C123–C152.

[13] Luc Buatois, Guillaume Caumon, and Bruno Lévy. 2009. Concurrent number cruncher: a GPU implementation of a general sparse linear solver. *International Journal of Parallel, Emergent and Distributed Systems* 24, 3 (2009), 205–223.

[14] Aydin Buluç, Jeremy T Fineman, Matteo Frigo, John R Gilbert, and Charles E Leiserson. 2009. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*. 233–244.

[15] Phillip Colella. 2004. Defining software requirements for scientific computing. (2004).

[16] Steven Dalton, Sean Baxter, Duane Merrill, Luke Olson, and Michael Garland. 2015. Optimizing sparse matrix operations on gpus using merge path. In *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 407–416.

[17] Steven Dalton, Nathan Bell, Luke Olson, and Michael Garland. 2014. Cusp: Generic Parallel Algorithms for Sparse Matrix and Graph Computations. http://cusplibrary.github.io/ Version 0.5.0.

[18] Timothy A Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 1–25.

[19] Gunduz Vehbi Demirci and Cevdet Aykanat. 2020. Scaling sparse matrix-matrix multiplication in the accumulo database. *Distributed and Parallel Databases* 38 (2020), 31–62.

[20] Julien Demouth. 2012. Sparse matrix-matrix multiplication on the GPU. In *Proceedings of the GPU technology conference*, Vol. 3.

[21] Mehmet Deveci, Christian Trott, and Sivasankaran Rajamanickam. 2017. Performance-portable sparse matrix-matrix multiplication for many-core architectures. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 693–702.

[22] Mehmet Deveci, Christian Trott, and Sivasankaran Rajamanickam. 2018. Multithreaded sparse matrix-matrix multiplication for many-core and GPU architectures. *Parallel Comput.* 78 (2018), 33–46.

[23] Steve Dropsho, Alper Buyuktosunoglu, Rajeev Balasubramonian, David H Albonesi, Sandhya Dwarkadas, Greg Semeraro, Grigorios Magklis, and Michael L Scottt. 2002. Integrating adaptive on-chip storage structures for reduced dynamic power. In *Proceedings. International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 141–152.

[24] Valentin Le Fèvre and Marc Casas. 2023. Optimization of SpGEMM with Risc-V vector instructions. *arXiv preprint arXiv:2303.02471* (2023).

[25] John R Gilbert, Cleve Moler, and Robert Schreiber. 1992. Sparse matrices in MATLAB: Design and implementation. *SIAM journal on matrix analysis and applications* 13, 1 (1992), 333–356.

[26] John R Gilbert, Steve Reinhardt, and Viral B Shah. 2008. A unified framework for numerical and combinatorial computing. *Computing in Science & Engineering* 10, 2 (2008), 20–25.

[27] Felix Gremse, Andreas Hofter, Lars Ole Schwen, Fabian Kiessling, and Uwe Naumann. 2015. GPU-accelerated sparse matrix-matrix multiplication by iterative row merging. *SIAM Journal on Scientific Computing* 37, 1 (2015), C54–C71.

[28] Felix Gremse, Kerstin Kupper, and Uwe Naumann. 2018. Memory-efficient sparse matrix-matrix multiplication by row merging on many-core architectures. *SIAM Journal on Scientific Computing* 40, 4 (2018), C429–C449.

[29] Giulia Guidi, Oguz Selvitopi, Marquita Ellis, Leonid Oliker, Katherine Yelick, and Aydın Buluç. 2021. Parallel string graph construction and transitive reduction for de novo genome assembly. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 517–526.

[30] Fred G Gustavson. 1978. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Transactions on Mathematical Software (TOMS)* 4, 3 (1978), 250–269.

[31] Song Han, Huizi Mao, and William J. Dally. 2016. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. In *Proceedings of the 4th International Conference on Learning Representations (ICLR)*, Yoshua Bengio and Yann LeCun (Eds.).

[32] Timothy Hayes, Oscar Palomar, Osman Unsal, Adrian Cristal, and Mateo Valero. 2012. Vector extensions for decision support dbms acceleration. In *2012 45th annual IEEE/ACM international symposium on microarchitecture*. IEEE, 166–176.

[33] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W Fletcher. 2019. Extensor: An accelerator for sparse tensor algebra. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 319–333.

[34] Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. 2021. Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks. *Journal of Machine Learning Research* 22, 241 (2021), 1–124.

[35] Reza Hojabr, Ali Sedaghati, Amirali Sharifian, Ahmad Khonsari, and Arrvindh Shriraman. 2021. Spaghetti: Streaming accelerators for highly sparse gemm on fpgas. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 84–96.

[36] Intel. 2020. *Intel® Architecture Instruction Set Extensions and Future Features Programming Reference*. 319433-038.

[37] Valentin Isaac-Chassande, Adrian Evans, Yves Durand, and Frédéric Rousseau. 2024. Dedicated Hardware Accelerators for Processing of Sparse Matrices and Vectors: A Survey. *ACM Transactions on Architecture and Code Optimization* (2024).

[38] Satoshi Itoh, Pablo Ordejón, and Richard M Martin. 1995. Order-N tight-binding molecular dynamics on parallel computers. *Computer physics communications* 88, 2-3 (1995), 173–185.

[39] Haonan Ji, Shibo Lu, Kaixi Hou, Hao Wang, Zhou Jin, Weifeng Liu, and Brian Vinter. 2021. Segmented merge: A new primitive for parallel sparse matrix computations. *International Journal of Parallel Programming* 49 (2021), 732–744.

[40] Haim Kaplan, Micha Sharir, and Elad Verbin. 2006. Colored intersection searching via sparse rectangular matrix multiplication. In *Proceedings of the twenty-second annual symposium on Computational geometry*. 52–60.

[41] David R Kincaid, Thomas C Oppe, and David M Young. 1989. *ITPACKV 2D user's guide*. Technical Report.

[42] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, and Parthasarathy Ranganathan. 2013. Meet the walkers: Accelerating index traversals for in-memory databases. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. 468–479.

[43] Moritz Kreutzer, Georg Hager, Gerhard Wellein, Holger Fehske, and Alan R Bishop. 2014. A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide SIMD units. *SIAM Journal on Scientific Computing* 36, 5 (2014), C401–C423.

[44] Jiayu Li, Fugang Wang, Takuya Araki, and Judy Qiu. 2019. Generalized sparse matrix-matrix multiplication for vector engines and graph applications. In *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*. IEEE, 33–42.

[45] Jiawen Liu, Jie Ren, Roberto Gioiosa, Dong Li, and Jiajia Li. 2021. Sparta: High-performance, element-wise sparse tensor contraction on heterogeneous memory. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 318–333.

[46] Weifeng Liu and Brian Vinter. 2015. CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication. In *Proceedings of the 29th ACM on International Conference on Supercomputing*. 339–350.

[47] Xing Liu, Mikhail Smelyanskiy, Edmond Chow, and Pradeep Dubey. 2013. Efficient sparse matrix-vector multiplication on x86-based many-core processors. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*. 273–282.

[48] Francesco Minervini, Oscar Palomar, Osman Unsal, Enrico Reggiani, Josue Quiroga, Joan Marimon, Carlos Rojas, Roger Figueras, Abraham Ruiz, Alberto Gonzalez, Jonnatan Mendoza, Ivan Vargas, Cesar Hernandez, Joan Cabre, Lina

Khoirunisya, Mustapha Bouhali, Julian Pavon, Francesc Moll, Mauro Olivieri, Mario Kovac, Mate Kovac, Leon Dragic, Mateo Valero, and Adrian Cristal. 2022. Vitruvius+: An Area-Efficient RISC-V Decoupled Vector Coprocessor for High Performance Computing Applications. *ACM Trans. Archit. Code Optim.* (dec 2022). https://doi.org/10.1145/3575861 Just Accepted.

[49] Alexander L Minkin, Steven J Heinrich, Rajeshwaran Selvanesan, Charles McCarver, Stewart Glenn Carlton, Ming Y Siu, Yan Yan Tang, and Robert J Stoll. 2012. Cache miss processing using a defer/replay mechanism. US Patent 8,266,383.

[50] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. 2009. CACTI 6.0: A tool to model large caches. *HP laboratories* 27 (2009), 28.

[51] Yusuke Nagasaka, Satoshi Matsuoka, Ariful Azad, and Aydın Buluç. 2018. High-performance sparse matrix-matrix products on Intel KNL and multicore architectures. In *Proceedings of the 47th International Conference on Parallel Processing Companion*. 1–10.

[52] Yusuke Nagasaka, Satoshi Matsuoka, Ariful Azad, and Aydın Buluç. 2019. Performance optimization, modeling and analysis of sparse matrix-matrix products on multi-core and many-core processors. *Parallel Comput.* 90 (2019), 102545.

[53] Yusuke Nagasaka, Akira Nukada, and Satoshi Matsuoka. 2017. High-performance and memory-saving sparse general matrix-matrix multiplication for NVIDIA Pascal GPU. In *2017 46th International Conference on Parallel Processing (ICPP)*. IEEE, 101–110.

[54] Fan Ni, Song Jiang, Hong Jiang, Jian Huang, and Xingbo Wu. 2019. SDC: a software defined cache for efficient data indexing. In *Proceedings of the ACM International Conference on Supercomputing*. 82–93.

[55] Yuyao Niu, Zhengyang Lu, Haonan Ji, Shuhui Song, Zhou Jin, and Weifeng Liu. 2022. TileSpGEMM: a tiled algorithm for parallel sparse general matrix-matrix multiplication on GPUs. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 90–106.

[56] NVIDIA 2023. *CUDA C++ Programming Guide, Release 12.1*. NVIDIA.

[57] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok Kim, David Blaauw, Trevor Mudge, and Ronald Dreslinski. 2018. Outerspace: An outer product based sparse matrix multiplication accelerator. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 724–736.

[58] Mathias Parger, Martin Winter, Daniel Mlakar, and Markus Steinberger. 2020. Speck: Accelerating gpu sparse matrix-matrix multiplication through lightweight analysis. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 362–375.

[59] Julián Pavón, Ivan Vargas Valdivieso, Adrián Barredo, Joan Marimon, Miquel Moreto, Francesc Moll, Osman Unsal, Mateo Valero, and Adrian Cristal. 2021. VIA: A Smart Scratchpad for Vector Units with Application to Sparse Matrix Computations. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 921–934.

[60] Gengyu Rao, Jingji Chen, Jason Yik, and Xuehai Qian. 2022. Sparsecore: stream isa and processor specialization for sparse computation. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 186–199.

[61] RISC-V. 2023. RISC-V V Vector Extension. https://github.com/riscv/riscv-v-spec

[62] Paul Scheffler, Florian Zaruba, Fabian Schuiki, Torsten Hoefler, and Luca Benini. 2023. Sparse Stream Semantic Registers: A Lightweight ISA Extension Accelerating General Sparse Linear Algebra. *arXiv preprint arXiv:2305.05559* (2023).

[63] Oguz Selvitopi, Md Taufique Hussain, Ariful Azad, and Aydın Buluç. 2020. Optimizing high performance markov clustering for pre-exascale architectures. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 116–126.

[64] Premkishore Shivakumar and Norman P Jouppi. 2001. Cacti 3.0: An integrated cache timing, power, and area model. (2001).

[65] Nitish Srivastava, Hanchen Jin, Jie Liu, David Albonesi, and Zhiru Zhang. 2020. Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 766–780.

[66] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanael Premillieu, et al. 2017. The ARM scalable vector extension. *IEEE micro* 37, 2 (2017), 26–39.

[67] Blaise Tine, Krishna Praveen Yalamarthy, Fares Elsabbagh, and Kim Hyesoon. 2021. Vortex: Extending the RISC-V ISA for GPGPU and 3D-graphics. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 754–766.

[68] Steven JE Wilton and Norman P Jouppi. 1996. CACTI: An enhanced cache access and cycle time model. *IEEE Journal of solid-state circuits* 31, 5 (1996), 677–688.

[69] Michael M Wolf, Mehmet Deveci, Jonathan W Berry, Simon D Hammond, and Sivasankaran Rajamanickam. 2017. Fast linear algebra-based triangle counting with kokkoskernels. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.

[70] Chencheng Ye, Yuanchao Xu, Xipeng Shen, Xiaofei Liao, Hai Jin, and Yan Solihin. 2021. Hardware-based address-centric acceleration of key-value store. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 736–748.

[71] Raphael Yuster and Uri Zwick. 2004. Detecting short directed cycles using rectangular matrix multiplication and dynamic programming. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*. 254–260.

[72] Chao Zhang, Maximilian Bremer, Cy Chan, John Shalf, and Xiaochen Guo. 2022. ASA: Accelerating Sparse Accumulation in Column-wise SpGEMM. *ACM Transactions on Architecture and Code Optimization (TACO)* 19, 4 (2022), 1–24.

[73] Guowei Zhang, Nithya Attaluri, Joel S Emer, and Daniel Sanchez. 2021. Gamma: Leveraging Gustavson's algorithm to accelerate sparse matrix multiplication. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 687–701.

[74] Guowei Zhang and Daniel Sanchez. 2019. Leveraging caches to accelerate hash tables and memoization. In *Proceedings of the 52nd annual IEEE/ACM international symposium on microarchitecture*. 440–452.