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

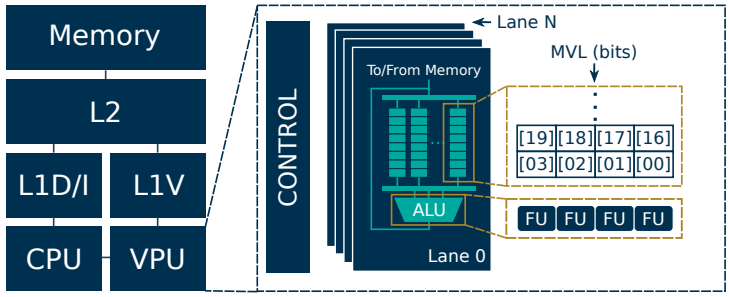

Figure 1.1: Block diagram of a system with a VPUs

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.