Partial Instruction Execution on Long SIMD Architectures

Francesc Martinez
Barcelona Supercomputing Center
Barcelona, Catalunya, Spain
francesc.martinez@bsc.es

Adrià Armejach
UPC
Barclona, Catalunya, Spain
Barcelona Supercomputing Center
Barcelona, Catalunya, Spain
adria.armejach@upc.edu

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

Abstract

Highly parallel architectures constitute a viable alternative to handle the Memory Wall as they are able to quickly instantiate a large amount of memory accesses while hiding the memory access latency by overlapping costly memory accesses with computation. these high degree of parallelism is achieved by GPU architectures, with exploit Single Instruction Multiple Threads (SIMT) parallelism, or by CPU architectures implementing emerging Single Instruction Multiple Data (SIMD) Instruction Set Architectures (ISAs) like the ARM Scalable Vector Extension (SVE) or the RISC-V Vector (RVV) extension. While architectures supporting instructions operating on these long SIMD registers improve performance since these instructions encode much program semantics, their long-latencies offer novel optimization opportunities.

This paper proposes the *Partial Instruction Execution (PIE)* approach. PIE leverages the observation that a large portion the data required to run arithmetic vector instructions is ready to be consumed much earlier than the whole vector register. PIE decomposes the monolithic execution of long SIMD or vector instructions, exposing their arithmetic operations to the underlying parallel hardware as soon as operands become available. We run an extensive evaluation campaign considering a variety of parallel workloads to show that that PIE has up to a 6.41% average performance speedup with respect to a state-of-the-art baseline. In addition, PIE achieves a larger average functional unit use (42.46%) than a state-of-the-art vector architecture with chaining (37.12%) and without chaining (10.55%).

Keywords

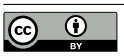
Vector Processing Units (VPUs), Single Instruction Multiple Data (SIMD), Scalable Vector Extension (SVE), RISCV Vector Extension (RISCV-V)

1 Introduction

Emerging workloads in the fields of High Performance Computing, Data Analytics and Artificial Intelligence have increasing memory footprints, while the ratio between memory throughput and compute throughput is decreasing. This issue, known as the *Memory Wall* [24, 45] continues to drive computer architecture research. Multiple efforts are made to increase the performance while avoiding the Memory Wall, improving frequencies [14][5], improving cache designs [37], larger number of cores and more memory channels in a system [32], 3D-stacked memory [13] or adopting different memory technologies like HBM [15], which provide a higher memory bandwidth at the cost of larger latency.

In addition, highly parallel architectures emerge as a viable alternative to mitigate the impact of the Memory Wall as they are able to quickly instantiate a large amount of memory accesses while hiding the memory access latency by overlapping costly memory accesses with computation. This high degree of parallelism achieved by GPU architectures, which exploit Single Instruction Multiple Threads (SIMT) parallelism, or by CPU architectures implementing emerging Single Instruction Multiple Data (SIMD) Instruction Set Architectures (ISAs),such as the ARM Scalable Vector Extension (SVE) [36] or the RISC-V Vector (RVV) extension [40]. These SIMD ISAs support vector length agnostic instructions, which means that their instructions can run on arbitrarily large SIMD registers, like the 16kbit registers of the NEC SX-AURORA [18] [38]. While letting instructions operate on long SIMD (also known as vector) registers bring performance improvements due to the large number of program semantics they encode [43], the long latency of arithmetic instructions with operands stored on large SIMD registers provides novel optimization opportunities. Our work is based on the observation that the monolithic nature of these arithmetic operations forces them to wait until long input SIMD registers are available, which may require a large number of memory requests to be brought from memory. These requests have an inherent latency irregularity caused by cache or Translation Lookaside Buffer (TLB) misses. Importantly, the larger the size of the vector register, the more significant this irregularity becomes, exacerbating the impact of monolithic vector instructions.

This paper proposes the *Partial Instruction Execution (PIE)* approach. PIE leverages the observation that a large portion the data required to run arithmetic vector instructions is ready to be consumed much earlier than the whole vector register. However, these early data are typically left waiting for the completion of all memory requests or arithmetic operations required to fill the whole vector register, which delays the execution of the arithmetic vector instruction. PIE partially executes an arithmetic instructions when a portion of its inputs is ready. Importantly, once this arithmetic instruction is partially executed, PIE has the ability to switch the instruction assigned to a certain functional unit to start the partial execution of a subsequent arithmetic instruction, hiding the latency of the remaining memory requests required by the first instruction. This flexibility improves the utilization rate of the processor functional units, leading to measurable performance gains. PIE is one



This work is licensed under a Creative Commons Attribution 4.0 International License.
PACT '26, Chicago, IL, USA
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2915-7/2026/10
https://doi.org/10.1145/3838684.3846874

of the first hardware approach able to fully break the monolithic nature of long SIMD or vector instructions and expose as early as possible the required arithmetic operations to the underlying parallel hardware. Previous approaches like chaining [31] [26] only partially break this monolithic nature by relaxing dependencies between arithmetic instructions, but they do not allow functional units to switch between partially executed arithmetic operations, neither enable hiding the latency of late memory request serving data required by long SIMD instructions. Also related approach is the decomposition into micro-operations on AMD's Jaguar [30], which while being similar would incur into significant overheads on processors implementing RISC-V Vector extension with long vector registers.

This paper makes the following contributions over the state-of-the-art:

- It reveals that the functional units of vector architectures suffer from low utilization due to the monolithic nature of vector instructions, which require the whole register to be ready to start their execution.
- It proposes the *Partial Instruction Execution (PIE)* hardware mechanism, which supports the partial execution of arithmetic SIMD instructions when a portion of their input is ready. In addition, PIE allows functional units to switch between different partially executed instructions before they are completed, improving workload granularity.
- It evaluates the performance of PIE considering a state-of-the-art vector architecture supporting chaining. The evaluation includes 18 parallel scientific computing and Artificial Intelligence (AI) workloads. Across these workloads PIE achieves a 6.41% average performance speedup and a larger average functional unit use (42.46%) than the state-of-the-art vector architecture with chaining (37.12%) and without chaining (10.55%). Our evaluation also shows that PIE reduces the number of instruction commit stalls caused by non-memory instructions from 16% over a chaining approach, while only a 44% are converted to commit stalls due to memory instructions.

Section 2 introduces the background and motivation of our proposal, Section 3 introduces the Partial Instruction Execution (PIE) mechanism, Section 4 explains the evaluation methodology, Section 5 analyzes the evaluation results, Section 6 summarizes the related work, and Section 7 contains the final the conclusions.

2 Background and motivation

2.1 Vector Processors and Vector Length Agnostic ISAs

Emerging computational workloads from domains like AI [8] or data analytics [23] require an increasingly large computational throughput, which has motivated the advent of largely parallel architectures like GPUs or vector architectures featuring long vector registers, with the NEC SX-AURORA TSUBASA [18, 38] being its most prominent example. In addition, the growing interest in vector architectures [29] has motivates the appearance of Vector Length Agnostic (VLA) ISAs like RVV [40] or SVE [36]. These ISAs support the generation of vector-length agnostic code that does not need to be explicitly adapted to any specific vector register length, increasing the flexibility of hardware architects in designing a wide range of SIMD architectures without requiring modifications to fundamental pieces of the software stack for each specific micro-architecture.

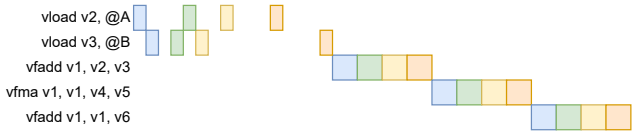


Figure 1: Example instruction execution timeline

2.2 Complex Memory Hierarchies

Current computer architectures feature complex cache hierarchies to reduce the memory access latency. In addition, there is a notable increase in the amount of memory channels, and also the use of systems with multiple memory technologies inside the same system [35]. This increasing complexity is producing systems where the latency for the different memory requests needed for a single instruction can be highly variable.

2.3 Motivation for Partial Instruction Execution

PIE is motivated by the observation that large vector architectures manage vector load instructions generating several memory demand requests, which typically incur highly variable amounts of cycles to be served due to cache or TLB misses. This generates a typical scenario where vector instructions have large portions of their input data available, but are required to wait for the remaining data during a potentially extended period of time. Figure 1 illustrates this scenario with an example where we can observe the impact of the late memory requests corresponding to vector load instructions `vload v2 @A` and `vload v3 @B` in delaying the execution of three vector arithmetic instructions. The example illustrates how these three vector instruction could be partially executed taking the data belonging to the blue and green portions as input.

Figure 2 shows in its y-axis percentage of cycles with all functional units being used (stacks, left) and the average functional Units in use (lines, right) considering 18 parallel workloads and two different baselines: a state-of-the-art vector architectures (denoted as `baseline` in Figure 2) and the same architecture with chaining (`chaining` Figure 2), chaining allows a dependent vector instruction to begin execution as soon as the first result elements from the preceding instruction become available, rather than waiting for the entire vector operation to complete. This overlaps dependent vector operations and effectively forms a pipeline across functional units. We consider from one to eight functional units, each one delivering a Fused Multiply and Add (FMA) instruction per cycle. Section 4 describes in detail the experimental setup. Figure 2 indicates that, while most benchmarks make full use of the system functional units when only one is available, once the number of functional units increases the average functional unit usage decreases. And, while this effect is partially mitigated by chaining, there is still significant under-use of the functional units.

PIE leverages two key observations: functional units are underutilized in many workloads, and a significant fraction of an instruction's input data may become available many cycles before

all of its inputs are ready. PIE exploits these opportunities by allowing vector instructions to execute partially as soon as a portion of their inputs becomes available. This advances computation while the remaining inputs are still being fetched or produced, so that by the time the last input data arrives, much of the instruction may already have been executed, effectively overlapping computation with the latency of late-arriving data. Moreover, partial execution reduces the granularity at which work is assigned to functional units, enabling them to switch between instructions more flexibly and improving utilization when the available computational workload is uneven.

3 Partial instruction execution

Partial Instruction Execution (PIE) aims to improve performance by dividing the execution of vector instructions in segments whose size matches the floating-point throughput (i.e. 16 64-bit floating-point instructions per cycle) and allowing instructions to be issued when at least one segment of their inputs are ready. To enable this mechanism, PIE tracks the validity of each vector register segment, allowing the instruction scheduler to check whether all input registers of a given instruction have at least one segment ready. Section 3.1 describes the hardware components required to support PIE. Section 3.2 indicates the modifications on the instruction execution to keep track of the validity of each vector register segment. Section 3.3 provides an example illustrating the operation of PIE. Section 3.4 explains why PIE incurs negligible overhead on the already available hardware.

3.1 Hardware Components of PIE

This section describes the hardware modifications needed to enable partial instruction execution. PIE introduces changes in the Vector Register File, the Load/Store Queue, the Instruction Scheduler and the Vector Processing Unit.

3.1.1 The Vector Register File (VRF). to enable the fine-grain dependency tracking in terms of segments at instruction segment level PIE augments the Vector Register File with a *Valid Segment Array (VSA)*. This array tracks the validity/readiness of registers with segment granularity. The VSA is SRAM table separated from the register file that does not require any extra read/write port. This is done to avoid increasing pressure on the VRF ports.

3.1.2 The Load Store Queue (LSQ). we modify it to track the number of pending memory requests per register segment rather than per register. These counters are initialized when the individual memory requests to the memory subsystem are generated, and are decremented when the associated memory request is fulfilled. When a counter reaches zero, the VSA is notified so the segment valid bit of the associated register can be updated. The load instruction completes when all pending request counters are set to zero as it would without segment decomposition.

3.1.3 The Instruction Scheduler. is modified to issue vector instructions when at least a common segment for all their input registers is available, rather than waiting for all their input registers to be fully ready. Standard resource availability constraints remain unchanged. For instructions with dependencies to more than one segment within same vector registers, e.g. reductions over the whole vector register, we do not modify the standard instruction scheduling.

3.1.4 The Vector Processing Unit (VPU). is modified to keep track of which instructions are partially executed. PIE adds a small buffer that contains the information of the partially executed instructions. Per each instruction, the buffer contains the instruction identifier, the functional unit assigned to execute it, whether it is currently being executed, and the valid bitmasks for its input and output registers. Input valid bitmasks are initialized when the instruction is issued by the instruction scheduler, and are updated when the values on the VSA are modified. The output ready bits track which segments have finished execution and are written back to the VSA when modified.

These modifications allow PIE to utilize the functional units earlier and with a finer granularity than the standard CPUs supporting SIMD units. As a result, PIE decreases the duration of the execution critical path. Unlike other approaches translating long vector instructions into micro-instructions, PIE does not increase pressure on both the reorder buffer and the register renaming unit.

3.2 Instruction Execution Changes

PIE requires modifications to the standard instruction execution flow to maintain and update segment validity information.

3.2.1 Instruction Issue. for non-memory instructions the instruction scheduler logic (Section 3.1.3) checks the valid bitmasks (Section 3.1.1) of all input vector registers. And if at least a segment is valid across all its input registers, the instruction can be issued. The standard constraints regarding non-vector inputs and resource availability still apply.

3.2.2 Execution of Load Instructions. when a load instruction is issued, the LSQ (Section 3.1.2) initializes the counters that track the number of memory requests pending for each segment. As data from a memory requests arrives to the LSQ, the corresponding counters are reduced. Once a counter reaches zero, the LSQ notifies the VRF so the valid bit for that segment is set to one (Section 3.1.1).

3.2.3 Execution of Arithmetic Instructions. when they are issued, a functional unit is assigned to execute one or more segments of the instruction. After those computations are completed for those segments, the valid bits for the output registers are updated in the VPU's instruction buffer (Section 3.1.4). The output generated by the instruction is written to the VRF, and the corresponding valid bits are set. Once the instruction commits, the instruction scheduler is notified, and the corresponding entry in the VPU instruction buffer is cleared.

3.2.4 VPU. once a functional unit finishes executing a segment, it becomes idle. Whenever there is an idle functional unit, the VPU checks its instruction buffer (Section 3.1.4) for instructions that have segments with all input vector registers available. Out of these available instructions, the VPU starts executing the oldest. If there is no such instruction, and there are available entries on the instruction buffer, the VPU accepts new instructions to be executed from the instruction scheduler.

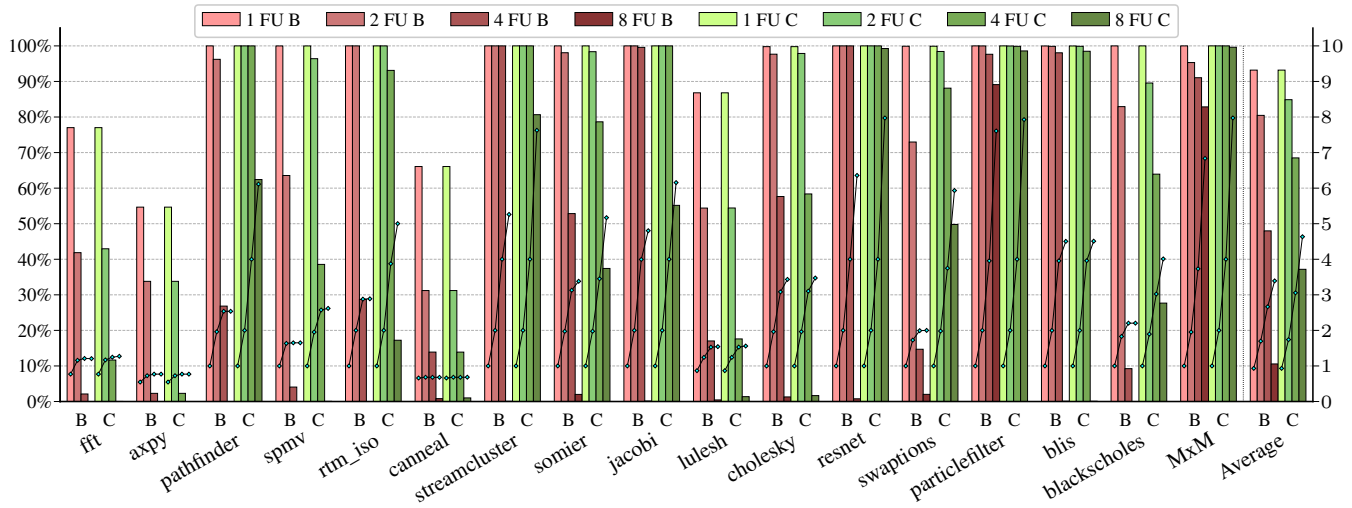


Figure 2: Percentage of cycles with full Functional Units (stacks, left) and average Functional Units in use (lines, right) for baseline (B) and chaining (C) simulations

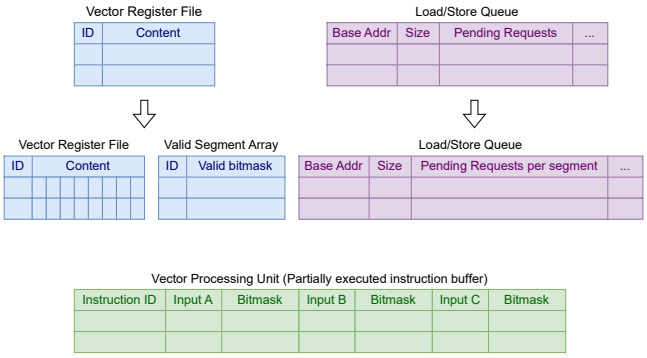


Figure 3: Added hardware diagram.

3.3 Execution example

Figure 4 displays a synthetic example illustrating the behavior of PIE considering a group of dynamic instructions representative of the parallel workloads we consider, which are described in Section 4.2. The group of dynamic instructions is composed by a vector load (`vle`), a vector floating-point add (`vfadd`), and a vector floating-point fused multiply-add (`vfmacc`). The figure shows three basic components: the Load/Store Queue with the number of pending requests for each segment on the left, the Vector Register File with the valid bitmask per each physical register on the center, and the vector processing unit with the instruction buffer for all the instructions that have been issued on the right. We also include the status of the functional units for clarity purposes.

At the beginning of the example (Step 0), the load instruction (`vle`) is issued, and the corresponding memory requests sent to the memory hierarchy over the course of several cycles. The load instruction is fetching the data, and the subsequent arithmetic instructions are waiting for their input dependencies to be resolved, i.e., the `vfadd` is waiting on the data being loaded on register v1, and the `vfmacc` is waiting on the data produced by the `vfadd` on register v4. All other register dependencies have been resolved, i.e., registers v2 and v3 are ready to be consumed, and their valid bits are already set. In this step, the load/store queue indicates that v1 register segment has one memory request pending for each segment. For non-contiguous or unaligned memory accesses, the counters can have values greater than one. In this initial step the functional units and the instruction buffer in the VPU are empty, as `vfadd` and `vfmacc` have not been issued.

On Step 1, data from the memory requests corresponding to the first two segments of register v1 arrive to the LSQ, and are stored on the VRF. The LSQ decrements the pending counters associated with those segments, and as those counters reach zero, notifies the VRF to update the corresponding valid values on the VSA. As the valid bits are set, the instruction scheduler detects that the `vfadd` instruction has segments with all input registers available, and issues the instruction to the VPU.

On Step 2, more data from the vector load instruction continue to arrive and are loaded in the v1 register. The corresponding segment pending counters are decreased, and reaching zero, the LSQ notifies the VRF to update the corresponding VSA valid values. In addition, the functional units finish the execution of the first segment of the `vfadd` instruction, update the corresponding values of the instruction output register v4 on the instruction buffer, and notify the VRF to update its valid values. The data arriving to register v1 allows the VPU to issue again the execution of the vfadd in the functional unit. As the register v4 contains valid values, the instruction scheduler also issues `vfmacc` to the VPU.

Between Steps 2 and 3, further data fetched by the `vle` instructions continues arriving to the LSQ. If at any moment the `vfadd` or `vfmacc` instructions finish processing their valid register segments, the functional unit becomes idle and may be reassigned to execute another instruction.

On Step 3, all remaining memory request from the `vle` instruction arrive, the LSQ sets all its pending counters to 0, and the instruction completes. The LSQ also notifies the VRF to set all the bits corresponding to register v1 to the valid state. With all the input registers of the `vfadd` instruction marked as valid, its last segment

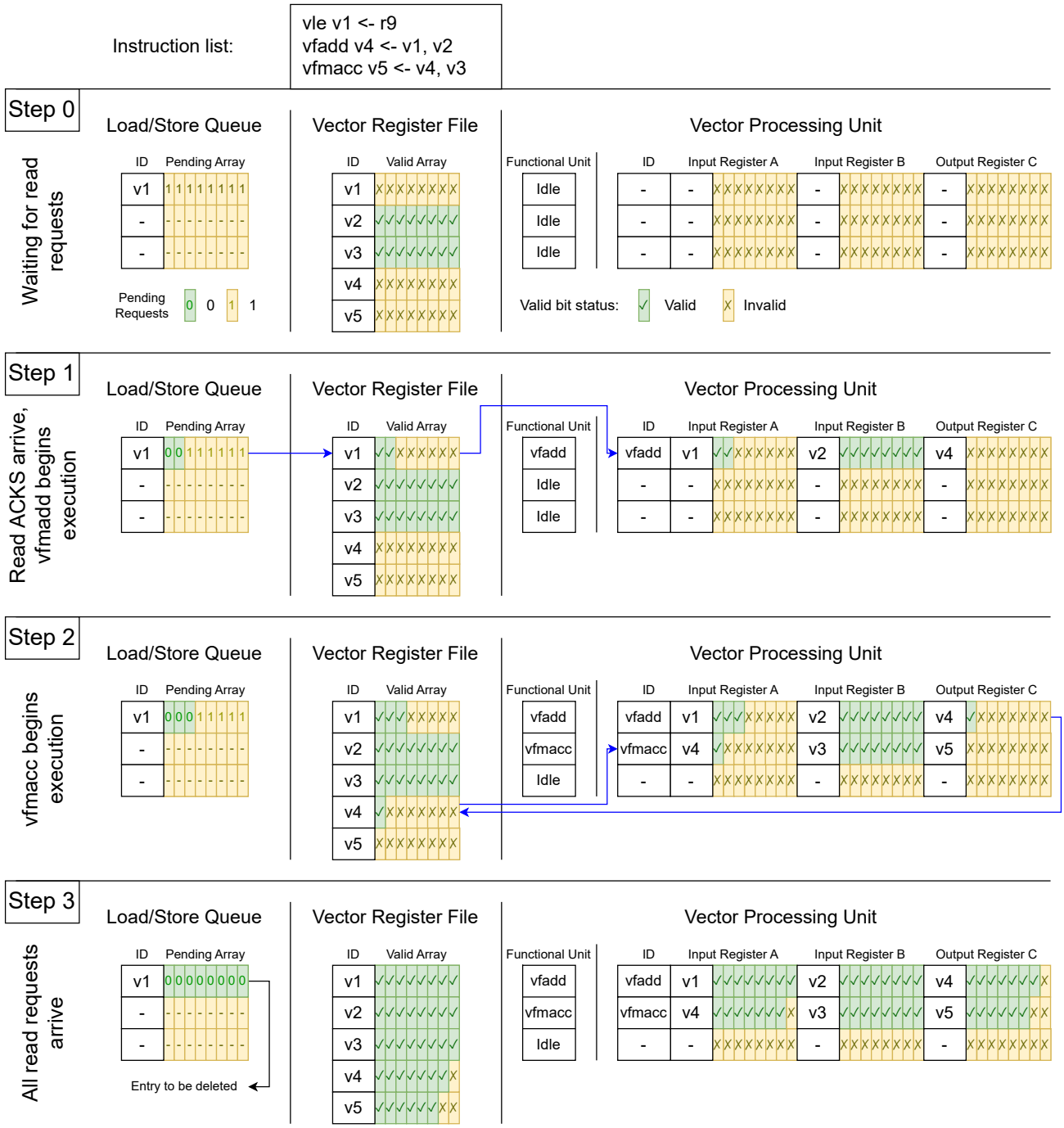


Figure 4: Step by step example.

is issued. Additionally, the `vfmacc` instruction is only missing the last segment of its input register v4, and is generating the second to last segment of register v5, which will be consumed by subsequent instructions.

The example in Figure 4 illustrates the advantage of PIE with respect to a standard vector processor in terms of early start of arithmetic instructions. Under the standard execution model of a vector processor under the standard execution model of a vector processor, which does not happen until Step 3. In contrast, PIE enables the `vfadd` to begin execution on Step 1 and to be nearly complete once its inputs are available.

3.4 Hardware Design Complexity

PIE incurs negligible hardware overhead. The valid bitmask structure does not belong to the register file. Therefore, PIE does not increase the total number of accesses to the vector register file. With a vector register file of 16 kbits, a segment size of 512 bits, and 37 physical registers, the valid mask table requires only 1,184 bits.

The partially executed instruction table inside the VPU contains 16 entries. Each entry contains an active bit, an assigned FU (6 bits), 3 input registers and 1 output registers each with 6 bits ids and 32 bits for the copy of the valid mask, for a total of 2,544 bits.

As we assume a RISC-V V architecture that can issue byte sized gathers for a full vector register, the LSQ needs to store 7 bit counters for each of the 32 segments for each entry to count up to 64 pending memory requests. We modeled a 16 entry LSQ, which brings the total additional space needed in the LSQ to 3,584 bits. The total space needed for the design is 7,314 bits, or 914 bytes.

4 Experimental environment

4.1 Simulation Methodology

Our simulation methodology is based on a trace-driven simulator similar to Champsim [1], [9] in which we model a 4-way out-of-order processor. Instruction traces are extracted on a RISC-V machine configured to run vector registers of 16kbits. The simulator implements a vector unit with a vector register file that manages register dependencies, register renaming, and segmented vector processing units and has been extended to determine whether an instruction can be partially executed following the PIE mechanism described in Section 3.

The Load/Store queue implements memory address disambiguation and store-to-load forwarding in the load-store queue. The memory hierarchy is modeled in detail, and includes port contention, MSHRs, cache associativity, cache allocation policies, cache replacement policies, write-back buffers and access latency. To simulate the main memory we use the MESS memory simulator [7] simulating 6 DDR4-2666 memory modules. Table 2 displays the parameters of the simulated architecture, which is based on contemporary vector architectures like the NEC SX-Aurora TSUBASA [18, 38].

4.2 HPC Workloads

The selected benchmarks come from a wide range of domains: the PARSEC suite [3] (*blackscholes*, *canneal*, *streamcluster*, and *swaptions*), Rodinia [4] (*particlefilter* and *pathfinder*), Basic Linear Algebra Subroutines (BLAS) kernel (*blis*) [42], the *somier* benchmark [28], 5 known scientific kernels (*AXPY*, *cholesky*, *jacobi*, *MxM*, *SpMV*), and five additional scientific workloads (*FFT* [44], *lulesh* [16], *HPCG* [6], *ResNet* [11], and the *RTM* isotropic kernel [47]).

Table 1 provides the arithmetic intensity of each considered workload and the input sets we consider. All codes have been vectorized using hand-tuned intrinsics based on the RISC-V "V" vector extension [40] and compiled with the LLVM [21] clang v17.0 compiler. Codes are Vector-Length Agnostic (VLA), which means they can run on architectures with different vector lengths without recompiling.

4.3 AI workloads

We have also tested PIE with GEMMs and convolution. The code has been generated extending oneDNN [12] with a RISC-V jit assembler similar to Xbyak [33][17]. Our objective being to analyze our approach against currently widespread workloads. Having a more reduced set allows us to better analyze the cases and reasons for performance improvements.

4.4 Evaluated Scenarios

We consider three alternative scenarios: (1) Basic: a scenario with strict register dependencies between instructions, (2) Chaining [31] [26]: an scenario where instructions can be issued if their input registers can be guarantied to be available due to chaining from another functional unit, and (3) Partial: an scenario where instructions can be issued to be partially executed if only a segment of its input registers is available, even if we cannot guarantee that the instruction will not stall while waiting for its inputs. This scenario applies the PIE approach described in Section 3.

4.5 Design space exploration

To evaluate the robustness of PIE we perform a design space exploration experiments to evaluate PIE's sensitivity to different architectural parameters. The goal of this analysis is to ensure that PIE's performance improvements are not limited to a narrow configuration space. Those parameters are last level cache (LLC) capacity, LLC load-to-use latency, Re-Order Buffer (ROB) size, number of physical vector registers and instruction execution latency. The outcome of these experiments is shown on Section 5.3.

4.6 Vector Processing Unit Internal Buffer Experiments

We conducted an experimental campaign to explore which is the optimal size of the internal buffer used by the Vector Processing Unit to store partially executed instructions. We evaluated two buffer organizations: one with a single unified buffer for all functional units inside the vector processing unit, and a second with individual buffers inside each functional unit. The experiments also verify that those buffers do not need to be excessively large for PIE to achieve most of its performance gains. The results of these experiments are shown on Section 5.2

4.7 Evaluation metrics

Our experiments compare the performance of PIE against a baseline that implements instruction chaining. For completeness some results against a basic baseline without chaining will also be provided. In Section 5 in addition of speedup we provide the average number of functional units in use, as well as the percentage of cycles that all available functional units are in use.

A performance analysis that the evaluation of Section 5 shows is based on breakdown of the total execution time in terms of: (1) Commit cycles: cycles in which instructions were committed, (2) Load stalls: cycles in which the head or the ROB is a load instruction waiting for memory; (3) Store stalls: cycles in which the head of the ROB is a store instruction that has not yet issued all its memory requests; and (4) Non-memory stalls: cycles in which the head of

Table 1: Set of evaluated benchmarks, with their domain, inputs used, and arithmetic intensity.

Benchmark	Domain	Computational Pattern	Input	Arith. I. (a)
AXPY	Scientific Kernel	Dense Linear Algebra	524,288 DP elements	0.099
blackscholes	Financial Analysis	Dense Linear Algebra	in_64k	5.611
blis	Scientific Kernel	Dense Linear Algebra	1024 × 1024	3.692
canneal	Engineering	Unstructured Grids	nsw=100, t=300, netlist=2500000.nets, ns=8	1.000
cholesky	Scientific Kernel	Dense Linear Algebra	1536 × 1536	3.689
FFT	Engineering	Fast Fourier Transform	256 × 256, blocksize=8	0.767
HPCG	HPC Benchmark	Sparse Linear Algebra	nx=64, ny=64, nz=64	0.045
jacobi-2d	Engineering	Dense Linear Algebra	n=256, tsteps=2	0.142
lulesh	Hydrodynamics	Unstructured Grids	i=2, s=30	0.897
MxM	Scientific Kernel	Dense Linear Algebra	ix=iy=iz=256	5.564
particlefilter	Medical Imaging	Structured Grids	x=4096, y=4096, z=16, np=2048	144.708
pathfinder	Grid Traversal	Dynamic Programming	1024 × 1024	0.333
ResNet	Image Recognition	Convolutions	mb=1, ic=256, oc=1024, ih=14, oh=14, kh=1	1.565
RTM_iso	Engineering	Finite Difference Method	nit=3, nx=1040, ny=1040, nz=16	0.301
SpMV	Scientific Kernel	Sparse Linear Algebra	bmw7st_1.mtx (141347 × 141347)	0.142
streamcluster	Data Mining	Dense Linear Algebra	k1=10, k2=20, d=128, n=16k, cluster=1000	3.891
somier	Physics Simulation	Dense Linear Algebra	iter=4, N=20	0.229
swaptions	Financial Analysis	MapReduce	ns=2, sm=32768	2.682

a: Arithmetic intensity calculated as number of floating-point operations (FLOPs) divided by number of bytes read (FLOPs/Byte).

Table 2: Architectural parameters of the simulated system.

Processing unit (CPU)	
Issue / Commit width	4
Core frequency	2.4GHz
Re-Order Buffer entries	220
Vector unit (VU)	
Vector Re-Order Buffer entries	64
Vector register length	16384 bits
VPU width (throughput)	8, 16, 32 DP FP/cycle
Number of Functional Units in the VPU	1, 2, 4, 8
Number of physical registers	40
LSQ instructions	16 instructions
LSQ MSHR entries	256 memory requests
Memory subsystem	
L1/L2/LLC capacity	32KB/256KB/1 MB per core
L1/L2/LLC cache line size	64 bytes
L1/L2/LLC load-to-use latency	2/12/50 cycles
L1/L2/LLC associativity	2/4/8
LLC → VU bandwidth	2048 bits/cycle
L1/L2/LLC MSHR entries	16/32/256
L1/L2/LLC prefetcher	NextLine/NextLine/Best-Offset
Main memory	MESS: 6 DDR4-2666 modules

the ROB is a non-memory instruction that has not yet finished its execution.

5 Evaluation

To extract results we analyze the performance with partial vector instruction execution against a baseline with instruction chaining, but in some plots we include a basic baseline simulation without any instruction forwarding. We analyze the results for simulation speedup, the number of cycles in which the system's functional units are fully occupied, and the breakdown of the instructions that block the system from committing instructions.

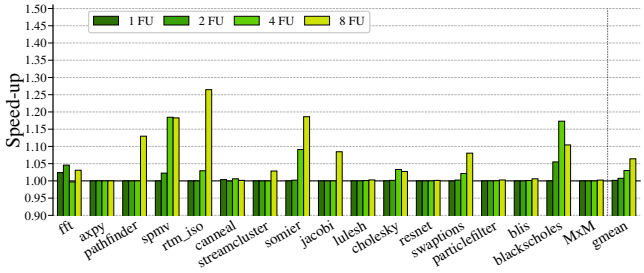


Figure 5: Speedup w.r.t. chaining baseline simulation with a throughput of 16 DP FP/cycle for HPC workloads with the same number of functional units.

5.1 HPC Workloads

In Figure 5 we can see the speedup of the simulations with respect to the chaining baseline with the same number of functional units. We observe that the performance is not altered when using only one functional unit, and it scales up with the number of functional units. The exception is FFT, which exhibits a small slowdown with PIE and four functional units. This slowdown results from increased load instruction latency due to a higher LLC miss ratio, caused by fewer prefetch requests. The LLC uses a Best-Offset Prefetcher (BOP) [25], which disables prefetching when training fails to identify a suitable offset. During training, BOP evaluates candidate offsets by comparing the triggering demand access only against memory requests that are at least 50 cycles old (BOP parameter). This minimum delay favors offsets that allow prefetches to arrive sufficiently early; offsets associated with more recent requests may produce late prefetches, resulting in half-misses and smaller latency reductions. In this configuration, PIE changes the time between memory accesses, interfering with BOP's ability to identify a suitable offset and consequently reducing prefetch activity. Adjusting BOP's minimum delay could mitigate this slowdown, but might

also affect the performance of other benchmarks. FFT is the only benchmark in our evaluation for which PIE changes prefetcher behavior.

Pathfinder, SpMV, RTM_iso, somier, jacobi, swaptions and blackscholes have a clear performance speedup of 1.08× to 1.26× with 8 functional units. Some benchmarks like pathfinder, streamcluster, jacobi, ResNet, particlefilter, and MxM have only performance gains with eight functional units. This lack of speedup can be explained due to the functional unit usage, as we will show later. AXPY, canneal, and lulesh do not have a performance gain, as they are mainly memory limited, and don't benefit from improved functional unit usage. Also noteworthy, the speedup for SpMV is nearly the same for four and eight functional units, this is due to the benchmark barely benefiting from the increase. Finally, for blackscholes, the performance speedup decreases with eight functional units compared to four. This is because chaining significantly improves its functional unit utilization, while the performance improvements from PIE are more modest. PIE reaches the limits imposed by the benchmark's instruction dependencies, and cannot improve on functional unit usage over chaining as much as it does with one to four functional units, as shown in Figure 7.

The geometric means of the speedup achieved by PIE against the a chaining baseline are 0.16%, 0.75%, 3.01% and 6.41% when considering 1, 2, 4, and 8 functional units, respectively. When considering a basic baseline, the improvements of PIE are 0.16%, 3.29%, 18.42% and 43.28%, not shown in the plot.

On Figure 6 we observe the instruction type that is preventing the commit of instructions, with the data scaled to the basic baseline simulation with a single functional unit. The plot contains 12 bars for each benchmark, which correspond to the three considered scenarios (basic (B), chaining (P) and partial (P)) times the four different functional unit counts we consider (1, 2, 4 and 8).

We can observe that AXPY, canneal, and lulesh do not improve with out approach nor do they greatly benefit from an increase in the number of functional units, as the majority of the core is idle is due to memory instructions.

We can also observe that the benchmarks that obtained a speedup on Figure 5 (pathfinder, streamcluster, somier, swaptions and blackscholes) there is a clear reduction in the number of stalls due to non-memory instructions, our main goal.

On most benchmarks we can observe that they are limited by non-memory instructions, but an approach based on improve the functional unit usage does not improve their performance, indicating another bottleneck on the core or in the internal instruction dependencies of the benchmarks.

We can also observe another behavior, in some benchmarks there is a decrease in the stalls due to non-memory instructions, but the total execution time does not improve accordingly. This happens on FFT, SpMV, jacobi and cholesky. Their lack of performance improvement is due to the increase in the number of cycles that the system is stalled waiting for load instructions to finish, indicating that we have moved the system bottleneck from the functional units to the memory subsystem.

The average non-memory stalls change from 85.95%, 44.77%, 29.23% and 25.82% for the basic scenario, to 85.95%, 42.83%, 23.30% and 16.18% for the chaining scenario, and 83.08%, 40.95%, 21.33% and 13.56% for the partial instruction execution scenario. Also, the percentage of memory stalls change from 10.77%, 10.69%, 11.63% and 12.13% for the basic scenario, to 10.77%, 11.19%, 12.25% and 13.40% for the chaining scenario, and 13.45%, 12.52%, 13.12% and 14.54% for the partial instruction execution scenario.

On Figure 7 we can observe the percentage of cycles that all the functional units are occupied (stacks, left), and the average number of functional units occupied during the simulation (lines, right). We can observe that some benchmarks like AXPY, canneal and lulesh have an extremely low functional unit usage and do not show any sign of improvement.

On FFT, AXPY, canneal and lulesh there is very little functional unit usage. This indicates that the performance increase on FFT is clearly due to reducing the time between all the instruction inputs are ready and the time the instruction completes its execution. On the rest of benchmarks where the functional unit utilization metrics improve the performance gain can also be attributed to a better functional unit usage due to a decrease in the instruction granularity.

Some like SpMV, RTM_iso and cholesky have a modest usage reaching at most 25% usage with 8 functional units with no instruction forwarding, but with modest improvements for partial instruction execution on SpMV and RTM_iso.

Finally on pathfinder, streamcluster, somier, jacobi, ResNet, swaptions, particlefilter and blackscholes the percentage of cycles that all the functional units are full does increase significantly providing the basis for their performance increase. In the case of Jacobi, with 1-4 functional units all scenarios keep the functional units in use nearly 100% of the time. With 8 functional units they are no longer the bottleneck, and chaining and PIE improve functional unit utilization. But Figure 6 shows that while non-memory stalls are reduced in more than half, many of those are replaced by load stalls, thus limiting performance gain.

In the case of particlefilter while the functional unit usage improves, the system is already making a very efficient use of the functional units, and the performance increase is very modest. For MxM we can observe that the benchmark does already make a nearly full functional unit usage with chaining and partial, explaining the lack of a performance improvement.

On the average, the percentage of cycles in which all Functional Units are fully utilized goes from 92.44%, 879.94%, 47.90% and 10.55% on the basic scenario, to 92.43%, 84.48%, 68.32% and 37.11% on the chaining scenario, and 92.54%, 85.98%, 71.61% and 42.46% for PIE. Also the average Functional Units in use, on average, goes from 0.92, 1.69, 2.65 and 3.38 on the basic scenario, to 0.92, 1.73, 3.05 and 4.61 on the chaining scenario, and 0.93, 1.74, 3.14 and 4.93 for PIE.

5.2 Partial Internal Buffer Simulations

We run a set of simulations to check which would be the appropriate size for the internal buffer that the VPU must implement to keep the intermediate information.

We run two basic scenarios, one with a shared intermediate buffer, and one where each functional unit has a smaller but private intermediate buffer. All simulations results have been evaluated but we show only the geometric mean of the speedup for the HPC workloads for the different number of functional units and intermediate buffer entries.

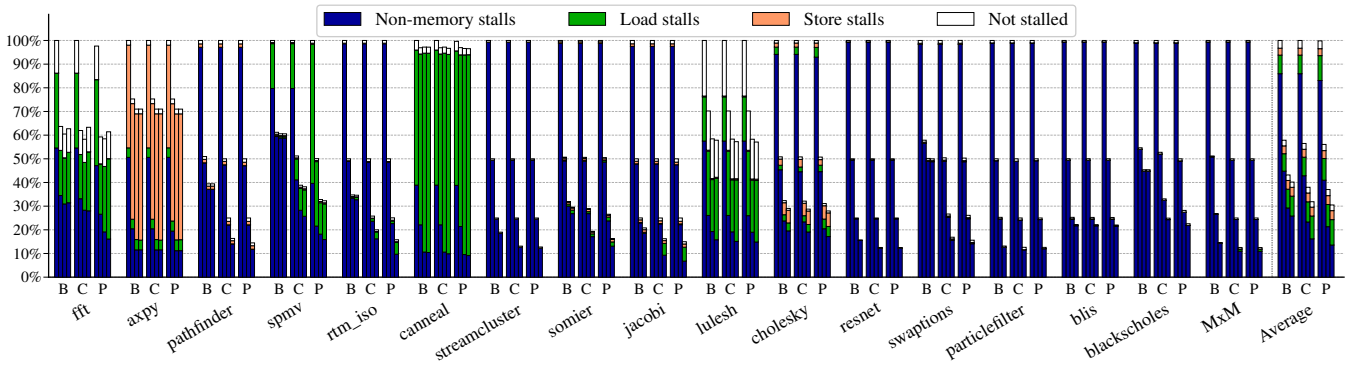


Figure 6: Stall breakdown for basic (B), chaining (C) and partial (P) simulations, each with 1, 2, 4 and 8 Functional Units, normalized to basic baseline with 1 Functional Unit

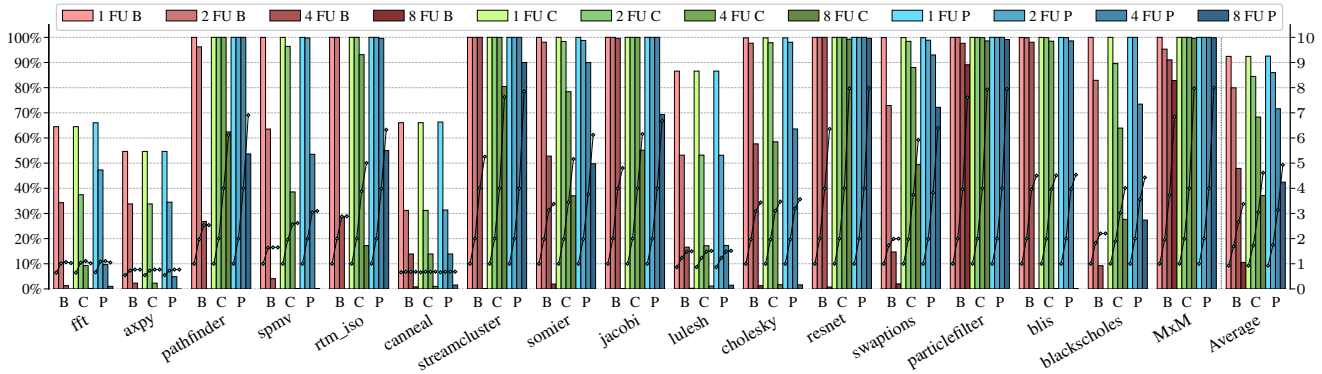


Figure 7: Percentage of cycles with full Functional Units (stacks, left) and average Functional Units in use (lines, right) for basic (B), chaining (C) and partial (P) simulations

Table 3: Shared buffer speedup geometric mean

Buffer Size	1	2	4	8	16	32	unlimited
1 FU	0.988	1.000	1.001	1.001	**1.002**	1.002	1.002
2 FU		0.987	1.002	1.007	**1.007**	1.007	1.007
4 FU			0.999	1.020	1.030	**1.032**	1.032
8 FU				1.020	1.064	**1.065**	1.065

Table 4: Individual buffer speedup geometric mean

Buffer Size	1	2	4	8	16
1 FU	0.989	1.000	1.000	1.001	**1.002**
2 FU	0.987	1.000	1.004	1.007	**1.009**
4 FU	1.000	**1.011**	1.010	1.004	0.999
8 FU	1.021	**1.023**	1.000	0.991	0.990

Table 3 contains the geometric mean speedup for with a shared buffer. We observe that PIE requires at least 4 buffer entries to work efficiently, otherwise, partially executed instructions can fill the buffer and prevent further instructions from executing.

We can also observe that while it takes 32 entries to achieve optimal performance, a configuration with 16 buffer entries achieves most of the performance speedup. The speedup achieved with unlimited buffer entries did not surpass that of a 32 entry buffer.

On Table 4 we have the results of the simulations with a private buffer for each functional unit. We observe that the performance gains over chaining are lower than with a shared buffer. We observe that the optimal speedup is still lower than with a shared buffer, as instruction cannot migrate functional units.

Combining both results we chose a shared buffer with 16 entries as our default configuration as it achieves most of the speedup, while maintaining a small resource footprint.

5.3 Design Space Exploration

Table 5: Instruction latency speedup comparison

Instruction latency	1 FU	2 FU	4 FU	8 FU
normal	1.002	1.007	1.030	1.064
fast	1.002	1.012	1.018	1.027
very fast	1.005	1.019	1.028	1.034

On Table 5 we can observe the results of modifying the instruction latencies. As the instruction latency decreases, the effectiveness of our approach also declines.

A mode detailed analysis reveals that this decrease is not uniform, and is particularly significant in those benchmarks in which we typically achieve a considerable speedup, as the benefits from reducing work granularity are reduced in most cases.

Further design space exploration simulations were conducted for the number of physical vector registers, size of the ReOrder Buffer, and the size of the Last Level Cache. Changing the LLC size to half or two megabytes had little effect (±0.3%). Changing the size of the ROB, had an also limited effect reducing it to only 40 entries resulted in a slowdown of up to 0.7%, and augmenting it to 82 or 100 entries introduced changes to the speedup of ±0.2%. Changing the number of physical vector register to 33 or 41 could reduce the speedup achieved by up to 1%.

5.4 AI Workloads

We also tested PIE with different AI workloads: GEMMs and Convolutions.

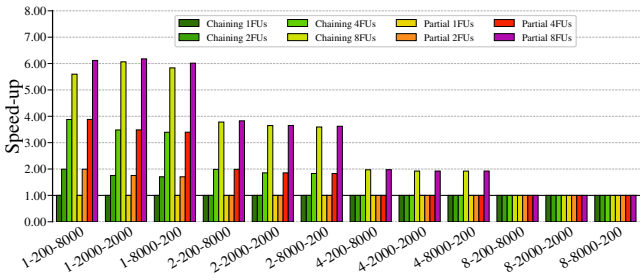


Figure 8: Speedup w.r.t. basic baseline simulations for GEMMs with very low M input parameter.

The results can be observed on Figure 8 labeled with the parameters M-N-K. Matrices with a parameter M=1 show a significant speedup compared to the basic baseline for both PIE and chaining. However, this speedup decreases as the size of the matrices increases, eventually disappearing with a matrix with parameter M=8. The speedup of PIE over chaining goes from 1.09× on the first matrix, and only achieves between 1.01× and 1.03× on the next three examples.

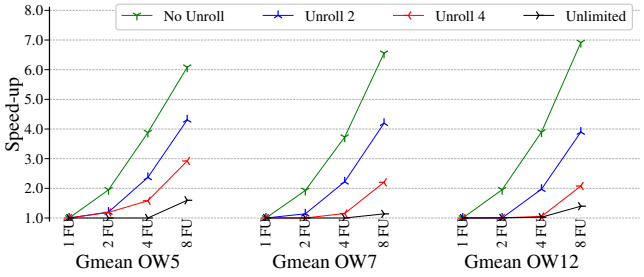


Figure 9: Speedup w.r.t. basic baseline simulations for AI workloads with the same number of Functional Units.

A similar scenario happens with the convolutions, which have a very low speedup in the general case. We observed though, a small improvement on convolutions with low output width parameter, we theorized that the improvement is due to the code not being able to do aggressive loop unrolling.

To validate this hypothesis, we ran a set of input convolutions with the lowest output width configuration, with different versions of the code limiting the loop unrolling to 8, 4, 2 and 1 iterations. Figure 9 shows these results, divided in three different plot sets

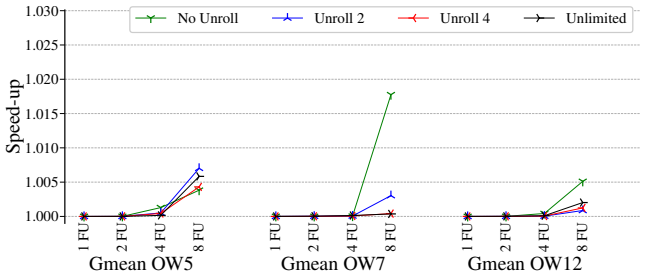


Figure 10: Speedup w.r.t. chaining baseline simulations for AI workloads with the same number of Functional Units.

for PIE over the basic baseline, for the different convolutions types grouped by their output width parameter. Each plot contains the speedup with respect to the basic baseline for 1, 2, 4 and 8 functional units for each loop unrolling configuration.

The results show a consistent pattern for all inputs: with the speedup is higher when loop unrolling is disabled (green) and decreases when the loop unrolling limit increases: blue with 2 loop unrolling limit, red with 4 and black with unlimited. Additionally, the speedup is reduced when moving to inputs with a bigger output width. Figure 10 shows the results compared to the chaining scenario, where PIE's speedup is more limited, specially with lower FU counts, leading to small improvements with 8 FUs.

6 Related work

Our work focused on improving performance on architectures with large SIMD registers by enabling a fine-grained execution model that breaks the monolithic SIMD execution model. Previous work includes the proposal of instruction chaining first developed for the Cray1 [31] machine, or contributions focused on improving the performance of the Register Files like González et al. [27] [10], which introduced a dynamic register renaming technique in which the key idea is to postpone the allocation of physical registers until the write-back stage. They proposed a concept called Virtual-Physical Registers, which are not tied to any physical storage location and instead serve as tags to track data dependencies.

Several studies [2], [20], [34] propose associating a counter with each physical register to track pending read operations, allowing the register to be reused once the counter reaches zero.These aggressive reclamation techniques enable physical register usage to closely align with their actual lifetimes.

On Ramírez et al. [22] they propose AVA (Adaptable Vector Architecture) which implements a flexible Maximum Vector Length, that remains efficient both at short and large MVLs (512bits and 16kbits). Other work on the Vector Register File includes the work of Titopoulous et al. [41] which centers around ultra low-cost processors and introduces Register Dispersion, an ISA compliant proposal that uses a physically smaller VRF, that dynamically caches only the vector registers currently accessed by the application.

Research from Tanaka et al. [39] proposes a vector register sharing mechanism that directly shares vector registers inside the processor using specific circuitry, accelerating the vector processing capabilities. Research from Zhao et al. [46] proposes a novel microarchitecture for instruction sequencing in vector units with short architectural vector lengths, supporting fine-granularity chaining

and multi-issue out-of-order execution. Some research has aimed at solving the same problem in a different way, like Kooli et al. [19] who propose a Computational SRAM solution combining In- and Near-Memory Computing approaches, introduced as an integrated vector processing unit to be used by a scalar processor as an energy-efficient and high performing co-processor.

7 Conclusions

This paper proposes Partial Instruction Execution (PIE), a methodology that leverages the long time that vector register data spend partially available on the Vector register file, and the under-utilization of functional units on architectures with large vector registers. PIE enables the early issue of instructions to the vector processing Unit, reducing the granularity of the SIMD execution model, improving resource utilization, and ultimately reducing total execution time.

PIE has a small footprint of 914 bytes and delivers larger performance than state-of-the-art techniques accelerating vector architectures. PIE offers performance improvements for large vector architectures that remain significant across a wide range of hardware configurations. PIE is particularly effective in scenarios where the memory system is able to provide sufficient data, but the core is unable to make full use of the available functional units due to the monolithic nature of SIMD instructions.

As highly parallel architectures like CPUs with long SIMD units emerge as a viable alternative to handle the Memory Wall, innovative solutions are required in order to leverage this large parallelism. In this context, PIE successfully increases the granularity of SIMD execution and achieves significant performance improvements by exposing computation earlier and improving functional unit utilization.

Acknowledgments

This work has received funding from 'Future of Computing, a Barcelona Supercomputing Center and IBM initiative' (2023). It has been partially supported by the project PID2023-146511NB-I00 funded by the Spanish Ministry of Science, Innovation and Universities MCIU /AEI /10.13039/501100011033 and EU ERDF. A. Armejach is a Serra Hunter Fellow.

References

[1] [n. d.]. ChampSim. https://crc2.ece.tamu.edu/. Accessed: 09-10-2025.

[2] H. Akkary, R. Rajwar, and S.T. Srinivasan. 2003. Checkpoint processing and recovery: towards scalable large instruction window processors. In *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.* 423–434. doi:10.1109/MICRO.2003.1253246

[3] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques* (Toronto, Ontario, Canada) *(PACT '08)*. Association for Computing Machinery, New York, NY, USA, 72–81.

[4] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *IEEE International Symposium on Workload Characterization (IISWC)*. 44–54. doi:10.1109/IISWC.2009.5306797

[5] Su-Jin Cho, Jaewoo Ahn, Hyojin Choi, and Wonyong Sung. 2012. Performance analysis of multi-bank DRAM with increased clock frequency. In *2012 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2477–2480. doi:10.1109/ISCAS.2012.6271802

[6] Jack Dongarra, Michael A Heroux, and Piotr Luszczek. 2016. High-performance conjugate-gradient benchmark: A new metric for ranking high-performance computing systems. *The International Journal of High Performance Computing Applications* 30, 1 (2016), 3–10.

[7] Pouya Esmaili-Dokht, Francesco Sgherzi, Valéria Soldera Girelli, Isaac Boixaderas, Mariana Carmin, Alireza Monemi, Adrià Armejach, Estanislao Mercadal, Germán Llort, Petar Radojković, Miquel Moreto, Judit Giménez, Xavier Martorell, Eduard Ayguadé, Jesus Labarta, Emanuele Confalonieri, Rishabh Dubey, and Jason Adlard. 2024. A Mess of Memory System Benchmarking, Simulation and Application Profiling. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 136–152. doi:10.1109/MICRO61859.2024.00020

[8] Amir Gholami, Zhewei Yao, Sehoon Kim, Coleman Hooper, Michael W. Mahoney, and Kurt Keutzer. 2024. AI and Memory Wall. *IEEE Micro* 44, 3 (2024), 33–39. doi:10.1109/MM.2024.3373763

[9] Nathan Gober, Gino Chacon, Lei Wang, Paul V. Gratz, Daniel A. Jimenez, Elvira Teran, Seth Pugsley, and Jinchun Kim. 2022. The Championship Simulator: Architectural Simulation for Education and Competition. doi:10.48550/ARXIV.2210.14324

[10] A. Gonzalez, J. Gonzalez, and M. Valero. 1998. Virtual-physical registers. In *Proceedings 1998 Fourth International Symposium on High-Performance Computer Architecture*. 175–184. doi:10.1109/HPCA.1998.650557

[11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778.

[12] Intel. 2024. OneAPI Deep Neural Network Library. https://oneapi-src.github.io/oneDNN/.

[13] Philip Jacob, Aamir Zia, Okan Erdogan, Paul Belemjian, Jin-Woo Kim, Michael Chu, R.P. Kraft, J.F. Mcdonald, and Kerry Bernstein. 2009. Mitigating Memory Wall Effects in High-Clock-Rate and Multicore CMOS 3-D Processor Memory Stacks. *Proc. IEEE* 97 (02 2009), 108 – 122. doi:10.1109/JPROC.2008.2007472

[14] Jaehyeon Jang and Moonju Park. 2017. DRAM frequency scaling for energy efficiency based on memory usage. 308–309. doi:10.1109/ICCE.2017.7889331

[15] Hongshin Jun, Jinhee Cho, Kangseol Lee, Ho-Young Son, Kwiwook Kim, Hanho Jin, and Keith Kim. 2017. HBM (High Bandwidth Memory) DRAM Technology and Architecture. In *2017 IEEE International Memory Workshop (IMW)*. 1–4. doi:10.1109/IMW.2017.7939084

[16] Ian Karlin, Abhinav Bhatele, Jeff Keasler, Bradford L. Chamberlain, Jonathan Cohen, Zachary Devito, Riyaz Haque, Dan Laney, Edward Luke, Felix Wang, David Richards, Martin Schulz, and Charles H. Still. 2013. Exploring Traditional and Emerging Parallel Programming Models Using a Proxy Application. In *2013 IEEE 27th International Parallel and Distributed Processing Symposium (IPDPS)*. 919–932. doi:10.1109/IPDPS.2013.115

[17] Kentaro Kawakami, Kouji Kurihara, Masafumi Yamazaki, Takumi Honda, and Naoto Fukumoto. 2022. A Binary Translator to Accelerate Development of Deep Learning Processing Library for AArch64 CPU. *IEICE Transactions on Electronics* 105, 6 (2022), 222–231.

[18] Kazuhiko Komatsu, Shintaro Momose, Yoko Isobe, Osamu Watanabe, Akihiro Musa, Mitsuo Yokokawa, Toshikazu Aoyama, Masayuki Sato, and Hiroaki Kobayashi. 2018. Performance Evaluation of a Vector Supercomputer SX-Aurora TSUBASA. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. 685–696. doi:10.1109/SC.2018.00057

[19] Maha Kooli, Antoine Heraud, Henri-Pierre Charles, Bastien Giraud, Roman Gauchi, Mona Ezzadeen, Kevin Mambu, Valentin Egloff, and Jean-Philippe Noel. 2022. Towards a Truly Integrated Vector Processing Unit for Memory-bound Applications Based on a Cost-competitive Computational SRAM Design Solution. *J. Emerg. Technol. Comput. Syst.* 18, 2, Article 40 (April 2022), 26 pages. doi:10.1145/3485823

[20] Charles Eric LaForest and J. Gregory Steffan. 2010. Efficient multi-ported memories for FPGAs. In *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '10)*. Association for Computing Machinery, New York, NY, USA, 41–50. doi:10.1145/1723112.1723122

[21] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *CGO*. San Jose, CA, USA, 75–88.

[22] Cristobal Ramirez Lazo, Enrico Reggiani, Carlos Rojas Morales, Roger Figueras Bague, Luis A. Villa Vargas, Marco A. Ramirez Salinas, Mateo Valero Cortes, Osman Sabri Unsal, and Adrian Cristal. 2022. Adaptable Register File Organization for Vector Processors. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 786–799. doi:10.1109/hpca53966.2022.00063

[23] Hosein Mohammadi Makrani, Hossein Sayadi, Sai Manoj Pudukotai Dinakarra, Setareh Rafatirad, and Houman Homayoun. 2018. A comprehensive memory analysis of data intensive workloads on server class architecture. In *Proceedings of the International Symposium on Memory Systems* (Alexandria, Virginia, USA) *(MEMSYS '18)*. Association for Computing Machinery, New York, NY, USA, 19–30. doi:10.1145/3240302.3240320

[24] Sally A. McKee. 2004. Reflections on the Memory Wall. In *Proceedings of the 1st Conference on Computing Frontiers* (Ischia, Italy) *(CF '04)*. Association for Computing Machinery, New York, NY, USA, 162.

[25] Pierre Michaud. 2016. Best-offset hardware prefetching. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 469–480.

[26] Francesco Minervini, Oscar Palomar, Osman Unsal, Enrico Reggiani, Josue Quiroga, Joan Marimon, Carlos Rojas, Roger Figueras, Abraham Ruiz, Alberto Gonzalez, Jonnatan Mendoza, Ivan Vargas, César Hernandez, Joan Cabre, Lina Khoirunisya, Mustapha Bouhali, Julian Pavon, Francesc Moll, Mauro Olivieri, Mario Kovac, Mate Kovac, Leon Dragic, Mateo Valero, and Adrian Cristal. 2023. Vitruvius+: An Area-Efficient RISC-V Decoupled Vector Coprocessor for High Performance Computing Applications. *ACM Trans. Archit. Code Optim.* 20, 2, Article 28 (March 2023), 25 pages. doi:10.1145/3575861

[27] T. Monreal, A. Gonzalez, M. Valero, J. Gonzalez, and V. Vinals. 1999. Delaying physical register allocation through virtual-physical registers. In *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*. 186–192. doi:10.1109/MICRO.1999.809456

[28] Cristóbal Ramírez, César Alejandro Hernández, Oscar Palomar, Osman Unsal, Marco Antonio Ramírez, and Adrián Cristal. 2020. A RISC-V Simulator and Benchmark Suite for Designing and Evaluating Vector Architectures. *ACM Trans. Archit. Code Optim.* 17, 4, Article 38 (nov 2020), 30 pages. doi:10.1145/3422667

[29] Alejandro Rico, José A. Joao, Chris Adeniyi-Jones, and Eric Van Hensbergen. 2017. ARM HPC Ecosystem and the Reemergence of Vectors: Invited Paper. In *Proceedings of the Computing Frontiers Conference* (Siena, Italy) *(CF'17)*. Association for Computing Machinery, New York, NY, USA, 329–334. doi:10.1145/3075564.3095086

[30] Jeff Rupley. 2012. "Jaguar" AMD's next generation low power x86 core. 1–20. doi:10.1109/HOTCHIPS.2012.7476479

[31] Richard M. Russell. 1978. The CRAY-1 computer system. *Commun. ACM* 21, 1 (Jan. 1978), 63–72. doi:10.1145/359327.359336

[32] José Carlos Sancho, Michael Lang, and Darren J. Kerbyson. 2010. Analyzing the trade-off between multiple memory controllers and memory channels on multi-core processor performance. In *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*. 1–7. doi:10.1109/IPDPSW.2010.5470812

[33] Mitsunari Shigeo. 2024. Xbyak, A C++ JIT assembler for x86 (IA32), x64 (AMD64, x86-64). https://github.com/herumi/xbyak.

[34] J.E. Smith and G.S. Sohi. 1995. The microarchitecture of superscalar processors. *Proc. IEEE* 83, 12 (1995), 1609–1624. doi:10.1109/5.476078

[35] Avinash Sodani, Roger Gramunt, Jesus Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu. 2016. Knights Landing: Second-Generation Intel Xeon Phi Product. *IEEE Micro* 36 (2016). doi:10.1109/MM.2016.25

[36] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martínez, Nathanael Premillieu, Alastair Reid, Alejandro Rico, and Paul Walker. 2017. The ARM Scalable Vector Extension. *IEEE Micro* 37 (May 2017), 26–39.

[37] Ching-Long Su and Alvin M. Despain. 1995. Cache design trade-offs for power and performance optimization: a case study. In *Proceedings of the 1995 International Symposium on Low Power Design* (Dana Point, California, USA) *(ISLPED '95)*. Association for Computing Machinery, New York, NY, USA, 63–68. doi:10.1145/224081.224093

[38] Keichi Takahashi, Soya Fujimoto, Satoru Nagase, Yoko Isobe, Yoichi Shimomura, Ryusuke Egawa, and Hiroyuki Takizawa. 2023. Performance Evaluation of a Next-Generation SX-Aurora TSUBASA Vector Supercomputer. arXiv:2304.11921 [cs.DC]

[39] Tomoaki Tanaka, Ryosuke Higashi, Hidetaro Tanaka, Takefumi Miyoshi, Yasunori Osana, Jubee Tada, Kiyofumi Tanaka, and Hironori Nakajo. 2022. Shared Vector Register of RISC-V for the Future Hardware Acceleration. In *Proceedings of CARRV 2022*. –. https://carrv.github.io/2022/papers/CARRV2022_paper_10_Tanaka.pdf Presented at CARRV 2022, June 19, 2022, ISCA, NY.

[40] The RISC-V Foundation. 2020. The RISC-V "V" Vector Extension. https://github.com/riscv/riscv-v-spec/releases/download/v1.0/riscv-v-spec-1.0.pdf.

[41] Vasileios Titopoulos, George Alexakis, Kosmas Alexandridis, Chrysostomos Nicopoulos, and Giorgos Dimitrakopoulos. 2025. Register Dispersion: Reducing the Footprint of the Vector Register File in Vector Engines of Low-Cost RISC-V CPUs. arXiv:2503.17333 [cs.AR] https://arxiv.org/abs/2503.17333

[42] Field G. Van Zee and Robert A. van de Geijn. 2015. BLIS: A Framework for Rapidly Instantiating BLAS Functionality. *ACM Trans. Math. Softw.* 41, 3, Article 14 (jun 2015), 33 pages. doi:10.1145/2764454

[43] Pablo Vizcaino, Georgios Ieronymakis, Nikolaos Dimou, Vassilis Papaefstathiou, Jesus Labarta, and Filippo Mantovani. 2023. Short Reasons for Long Vectors in HPC CPUs: A Study Based on RISC-V. In *Proceedings of the SC '23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis* (Denver, CO, USA) *(SC-W '23)*. Association for Computing Machinery, New York, NY, USA, 1543–1549. doi:10.1145/3624062.3624231

[44] Pablo Vizcaino, Filippo Mantovani, and Jesus Labarta. 2021. Accelerating FFT Using NEC SX-Aurora Vector Engine. In *Euro-Par 2021: Parallel Processing Workshops: Euro-Par 2021 International Workshops, Lisbon, Portugal, August 30-31, 2021, Revised Selected Papers* (Lisbon, Portugal). Springer-Verlag, Berlin, Heidelberg, 179–190. doi:10.1007/978-3-031-06156-1_15

[45] Wm. A. Wulf and Sally A. McKee. 1995. Hitting the Memory Wall: Implications of the Obvious. *SIGARCH Comput. Archit. News* 23, 1 (mar 1995), 20-24.

[46] Jerry Zhao, Daniel Grubb, Miles Rusch, Tianrui Wei, Kevin Anderson, Borivoje Nikolic, and Krste Asanovic. 2024. Instruction Scheduling in the Saturn Vector Unit. doi:10.48550/arXiv.2412.00997

[47] Hua-Wei Zhou, Hao Hu, Zhihui Zou, Yukai Wo, and Oong Youn. 2018. Reverse time migration: A prospect of seismic imaging methodology. *Earth-Science Reviews* 179 (2018), 207–227.