Occamy: Elastically Sharing a SIMD Co-processor across Multiple CPU Cores

Zhongcheng Zhang
zhangzhongcheng20s@ict.ac.cn
SKLP, ICT, CAS
UCAS
Zhongguancun Laboratory
Beijing, China

Yan Ou[*]
ouyan4@hisilicon.com
Turing Business Department,
HiSilicon Technologies Company, Ltd
Beijing, China

Ying Liu[*]
liuying2007@ict.ac.cn
SKLP, ICT, CAS
Zhongguancun Laboratory
Beijing, China

Chenxi Wang
wangchenxi@ict.ac.cn
SKLP, ICT, CAS
Beijing, China

Yongbin Zhou
zhouyongbin@hisilicon.com
Turing Business Department,
HiSilicon Technologies Company, Ltd
Beijing, China

Xiaoyu Wang
xy.wang@hisilicon.com
Turing Business Department,
HiSilicon Technologies Company, Ltd
Beijing, China

Yuyang Zhang
zhangyuyang22s@ict.ac.cn
SKLP, ICT, CAS
UCAS
Beijing, China

Yucheng Ouyang
ouyangyucheng22z@ict.ac.cn
SKLP, ICT, CAS
UCAS
Beijing, China

Jiahao Shan
shanjiahao21s@ict.ac.cn
SKLP, ICT, CAS
UCAS
Beijing, China

Ying Wang
wangying2009@ict.ac.cn
SKLP, ICT, CAS
Beijing, China

Jingling Xue
j.xue@unsw.edu.au
UNSW
Sydney, Australia

Huimin Cui
cuihm@ict.ac.cn
SKLP, ICT, CAS
UCAS
Beijing, China

Xiaobing Feng
fxb@ict.ac.cn
SKLP, ICT, CAS
UCAS
Zhongguancun Laboratory
Beijing, China

ABSTRACT

SIMD extensions are widely adopted in multi-core processors to exploit data-level parallelism. However, when co-running workloads on different cores, compute-intensive workloads cannot take advantage of the underutilized SIMD lanes allocated to memory-intensive workloads, reducing the overall performance. This paper proposes Occamy, a SIMD co-processor that can be shared by multiple CPU cores, so that their co-running workloads can spatially share its SIMD lanes. The key idea is to enable elastic spatial sharing by dynamically partitioning all the SIMD lanes across different workloads based on their phase behaviors, so that each workload may execute in variable-length SIMD mode. We also introduce an Occamy compiler to support such variable-length vectorization by analyzing such phase behaviors and generating the vectorized code that works with varying vector lengths. We demonstrate that Occamy can improve SIMD utilization, and consequently, performance over three representative SIMD architectures, with negligible chip area cost.

[*]To whom correspondence should be addressed.

ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9918-0/23/03.
https://doi.org/10.1145/3582016.3582046

CCS CONCEPTS

• **Computer systems organization** → **Architectures**.

KEYWORDS

Architecture, Simd, Auto Vectorization

ACM Reference Format:
Zhongcheng Zhang, Yan Ou, Ying Liu, Chenxi Wang, Yongbin Zhou, Xiaoyu Wang, Yuyang Zhang, Yucheng Ouyang, Jiahao Shan, Ying Wang, Jingling Xue, Huimin Cui, and Xiaobing Feng. 2023. Occamy: Elastically Sharing a SIMD Co-processor across Multiple CPU Cores. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3582016.3582046

1 INTRODUCTION

SIMD extensions are widely adopted in mainstream multi-core processor to exploit data-level parallelism (DLP) [31, 45, 46]. However, when co-running workloads on different cores, some workloads (e.g., compute-intensive ones) cannot take advantage of the underutilized SIMD lanes that are allocated to some other workloads (e.g., memory-intensive ones), resulting underutilized SIMD resources and reduced overall performance.

This paper focuses on exploiting DLP for workloads (with different operational intensities) co-running on multi-core CPUs. For CPU cores with their own private SIMD lanes (Figure 1(a)), there are two factors causing underutilized SIMD resources. First, a workload runs on a core with a fixed number of SIMD lanes even though it actually requires varying vector lengths in its different phases [2, 14, 30, 33]. Second, the SIMD lanes in each core may be underutilized for *memory-intensive workloads* even they exhibit sufficient DLP, since the underlying SIMD pipelines may stall frequently to wait for expensive memory access instructions to be completed.

To improve SIMD utilization, two representative approaches for sharing the SIMD lanes by different cores exist:

- **Temporal-Sharing.** Adopted in Apple Matrix co-processor [17–19] and others [3, 4, 29], temporal sharing is introduced to improve SIMD utilization by allowing a core to use all the SIMD lanes of a SIMD co-processor (Figure 1(b)). However, all the cores will compete for other hardware resources, *e.g.*, vector register file (VRF), leading to reduced SIMD instruction issue rates for the co-running workloads. As a result, the SIMD resources can still be underutilized.
- **Static Spatial-Sharing.** As proposed in [3, 4, 29], one can statically partition all the SIMD lanes available and assign different partitions to different workloads co-running on different cores (Figure 1(c)). However, a partitioning decision, once determined, is fixed at runtime.

In this paper, we improve spatial sharing significantly by making it achievable dynamically or *elastically* for the first time (to the best of our knowledge), as illustrated in Figure 1(d). We propose Occamy, an ARM SVE SIMD co-processor, to support this *elastic spatial sharing* policy for a multi-core processor. The key insight is a hardware and software co-designed *elastic MIMD-SIMD* (EM-SIMD) execution model, which enables partitioning dynamically the SIMD lanes among a set of co-running workloads so that each workload can run in variable-length SIMD mode, by introducing a resource table to dynamically control the number of SIMD lanes allocated to each core. On the hardware side, Occamy designs SIMD lanes as a set of fully homogeneous functional execution units, so that a SIMD data path can be flexibly assembled from a subset of these execution units. Meanwhile, an embedded lane manager serves to

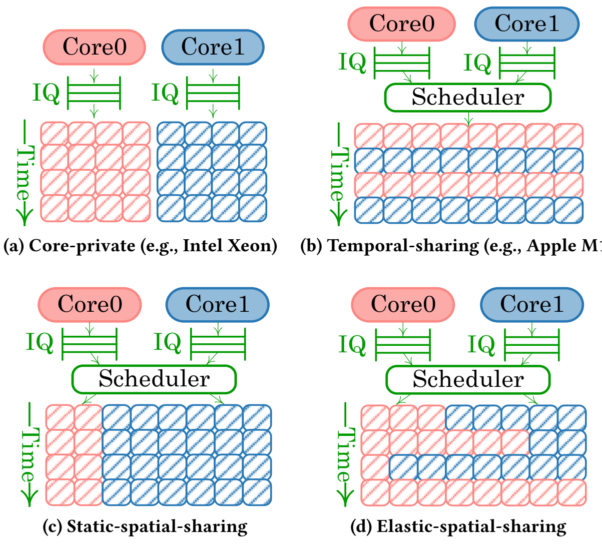

Core0
Core1
IQ
IQ
Time
Scheduler
(a) Core-private (e.g., Intel Xeon)
(b) Temporal-sharing (e.g., Apple M1)
(c) Static-spatial-sharing
(d) Elastic-spatial-sharing

Figure 1: Comparing three existing SIMD architectures with our approach (where IQ stands for Issue Queue).

partition the SIMD lanes for the co-running workloads by making use of the resource table. On the software side, Occamy includes a compiler to support such variable-length vectorization, which analyzes the phase-changing points and the phase behaviors in a workload, performs elastic vectorization, and finally, generates the assembly code that can elastically switch to the number of SIMD lanes needed during program execution.

The paper makes the following major contributions:

- **A Hardware and Software Co-Designed EM-SIMD Execution Model.** We present an EM-SIMD execution model, which enables partitioning a set of SIMD lanes into multiple variable-length lanes across the CPU cores by maintaining a lane partition plan in an on-chip resource table. Our execution model provides a set of ISA extensions, based on which a workload can first find the number of allocated lanes and then execute the variable-length SIMD instructions.
- **A SIMD Co-Processor for Supporting Elastic Spatial-Sharing.** We have designed an on-chip SIMD co-processor shared by multiple CPU cores, with its SIMD lanes being flexibly partitionable among all the cores during program execution. This is achieved by instructing a set of its homogeneous function units to be assembled into SIMD data paths with various widths appropriately, by the software according to the proposed EM-SIMD ISA.
- **A Hardware Lane Partitioning Manager.** We have designed a hardware lane manager for performing lane partitioning for a set of co-running workloads, guided by a vector-length-aware roofline model for estimating the performance of a workload for a given number of SIMD lanes.
- **Compiler Support.** We have developed a compiler for supporting elastic spatial-sharing by generating code that can switch to a new vector length needed at runtime.
- **Evaluation.** We have evaluated Occamy by running 34 workloads extracted from 28 `SPECCPU2017` loops and 14

OpenCV kernels on a simulated two-core processor. In the case of a pair of memory- and compute-intensive workloads, we assign the former to Core0 and the latter to Core1. Occamy (Figure 1(d)) improves SIMD utilization by $1.29-1.45\times$, and consequently, boosts performance by $1.11-1.39\times$ on Core1 while maintaining the same performance on Core0, on average, over three existing SIMD architectures (Figure 1(a)-(c)) with negligible chip area cost.

2 MOTIVATION

We use a two-workload example to motivate why Occamy can achieve the best SIMD utilization and performance among the four two-core SIMD architectures depicted in Figure 1.

Figure 2(a) gives an example of two co-running workloads, where WL#0 includes two memory-intensive loops from SPECCPU2017/654.rom_s with different operational intensities and WL#1 includes one computation-intensive loop from SPECCPU2017/621.wrf_s. Let there be a total of 32 SIMD lanes for both cores with each lane processing 32-bit floating-point data. We simulate all the four SIMD architectures in Figure 1 on a gem5 cycle-level simulated two-core ARM processor, with their configurations described in Section 7, by using the same amount of SIMD resources for fair comparison, *e.g.*, with the vector length set as 32 lanes for each architecture. We run WL#0 on Core0 and WL#1 on Core1. For WL#0, there are two phase-changing points, with the first indicating an increase in the operation intensity (implying that WL#0 can benefit from more lanes) and the second indicating that WL#0 is finished (implying that WL#1 can now use all the lanes). Thus, there are three phases, p1, p2 and p3, in between these two phase-changing points when co-running WL#0 and WL#1.

Occamy can complete WL#0 in nearly the same amount of time as the other three SIMD architectures but WL#1 more efficiently, delivering the best overall performance. Figures 2(b)-(d) illustrate their execution on the three architectures (Figures 1(a)-(c)) and Figure 2(e) illustrates our approach (Figures 1(d)). In Figure 2(b), 32 SIMD lanes are equally divided and used privately by its two cores. Thus, WL#1 cannot have its DLP sufficiently exploited since it cannot use the 16 lanes allocated to but under-utilized by WL#0. The entire execution takes 36606 cycles. In Figure 2(c), temporal-sharing enables WL#1 to fully utilize all the 32 lanes after WL#0 is done, shortening the overall execution time to 25937 cycles. In Figure 2(d), static spatial-sharing allocates 12 lanes to WL#0 and 20 lanes to WL#1. As a result, WL#1 can now enjoy more lanes than in Figure 2(b) but still cannot fully use all the 32 lanes, resulting in an overall execution time siting in between the first two cases: 29293 cycles. Figure 2(e) illustrates our elastic spatial sharing approach, by which we can dynamically adjust lane partitioning at the two phase-changing points to execute p1, p2 and p3, obtaining the fastest execution in 22666 cycles.

Figures 2(b)-(e) plot the SIMD utilization for the four SIMD architectures in Figure 1. In the x-axis, each point represents a set of 1000 consecutive cycles executed. In the y-axis, "#Lanes/Cycle (Avg)" represents the average number of SIMD lanes used in each core over the total allocated to the core. Figure 2(f) lists their performance statistics. For each SIMD architecture, the speedup obtained is given with the core-private SIMD lanes in Figure 1(a) as the baseline. For the SIMD issue rate and SIMD utilization metrics, the former represents the number of SIMD compute instructions issued in each core per cycle while the latter represents the fraction of busy lanes for a period of time (in terms of cycles) considered:

$$SIMD\_util = \frac{\sum_{c=1}^{C} \#busy\_lanes(c)}{\#total\_lanes \times C}$$

where C represents the total number of cycles, $\#busy\_lanes(c)$ the number of busy lanes in all the cores at cycle c, and $\#total\_lanes$ the total number of lanes available.

2.1 Existing SIMD Architectures

Let us examine each existing SIMD architecture in more detail.

- **Core-Private SIMD Lanes (Figure 1(a)).** As shown in Figure 2(b), this traditional SIMD architecture has the lowest SIMD utilization, achieving 60.6% only (Figure 2(f)). The low operational intensity in WL#0 leads to frequent pipeline stalls waiting for memory instructions to be completed.
- **Temporal Sharing (Figure 1(b)).** Temporal-sharing improves SIMD utilization from 60.6% to 84.7% via time-sharing multiplexing, leading to a speedup of $1.41\times$ on Core1 (and the same performance on Core0) over the baseline with core-private SIMD lanes (Figure 2(f)). This is because Core1 can steal the idle cycles of the SIMD co-processor from Core0 and execute its own SIMD instructions using all the SIMD lanes (Figure 2(c)). However, due to the sharing mechanism, both cores compete for hardware resources, such as vector register file (VRF) and store queues, causing the SIMD instruction issue rates to drop significantly for both WL#0 and WL#1. Compared to the baseline, temporal sharing results in reduced SIMD issue rates for both cores, 50% for WL#0 and 30% for WL#1 (calculated as weighed issue rates for the three phases from Figure 2(f)), leaving the SIMD resources still under-utilized. We will return to this temporal sharing policy in Section 7.6.
- **Static Spatial-Sharing (Figure 1(c)).** By allocating 12 lanes to WL#0 and 20 to WL#1, static spatial-sharing improves the SIMD utilization from 60.6% to 75.6% and thus achieve a speedup of by $1.25\times$ on Core1 (while also maintaining the same performance on Core0) over the baseline (Figure 2(f)). Compared to temporal-sharing, static spatial-sharing exhibits the improved SIMD issue rates for the two workloads, 168% for WL#0 and 100% for WL#1, since static spatial-sharing can partition both the SIMD lanes and the VRF entries, leading to much more fine-grained resource sharing (Figure 2(d)). In particular, static spatial-sharing can split a single VRF entry and provide the split parts to distinct workloads (cores) simultaneously, reducing the stalls of the SIMD co-processor incurred before. However, its partitioning decision is static and stays unchanged, leading to under-utilization of the SIMD resources. As shown in Figure 2(d), even if WL#0 has stopped and released its occupied resources, WL#1 still cannot use them, causing 37.5% of the SIMD resources to be wasted. In addition, as WL#0 has two phases, p1 and p2, each phase can be assigned a different number of SIMD lanes to achieve better SIMD utilization.

2.2 Our Approach: Elastic Spatial Sharing

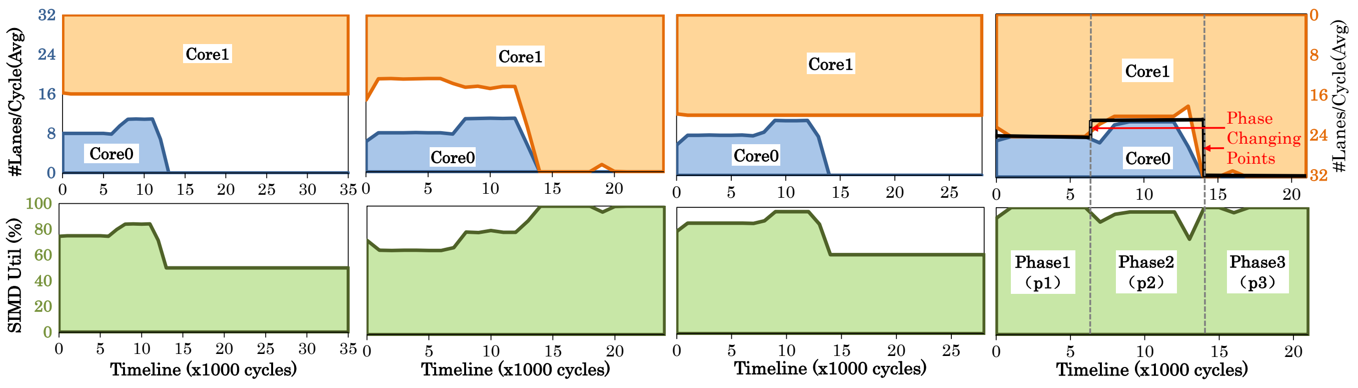
We present a software and hardware co-designed elastic MIMD-SIMD (EM-SIMD) co-processor, Occamy, that supports dynamic

```
Workload #0 (654.rom_s)
//Phase1 (rh3d.f90:1442), the body of i-loop
Ufx[i]=0.5*dndx[i]*(v[i]+v_1[i])²-dmde[i]*(v[i]+v_1[i])*(u[i]+u_1[i])
Ufe[i]=0.5*dndx[i]*(v[i]+v_1[i])*(u[i]+u_1[i])-dmde[i]*(u[i]+u_1[i])²

//Phase2 (rho_eos.f90:1548), the body of i-loop
wrk[i]=(den[i]+1000)*(bulk[i]+0.1*z_r[i])²
Tcof[i]=-(bulkDT[i]*0.1*z_r[i]*den1[i]+den1DT[i]*bulk[i]*(bulk[i]+0.1*z_r[i]))
Scof[i]=-(bulkDS[i]*0.1*z_r[i]*den1[i]+den1DS[i]*bulk[i]*(bulk[i]+0.1*z_r[i]))
```

```
Workload #1 (621.wrf_s/module_mp_wsm.f90:1363)
wi[k]=(ww[k]*dz[k-1]+ww[k-1]*dz[k])/(dz[k-1]+dz[k])     //the body of k-loop
```

(a) Workloads in `SPECCPU2017` **(with workload** `WL#0` **including two memory-intensive loops and** `WL#1` **including one computation-intensive loop)**



(b) Core-private **(c) Temporal-sharing** **(d) Static spatial-sharing** **(e) Elastic spatial-sharing**

Arch	**VL** (#Lanes)		**SIMD Issue Rates** (#Insts/Cycle)					**Times** ($\times 10^5$ Cycles)		**Speedups**		**SIMD Util**
	`WL#0`	`WL#1`	`WL#0.p1`	`WL#0.p2`	`WL#1.p1`	`WL#1.p2`	`WL#1.p3`	`WL#0`	`WL#1`	`WL#0`	`WL#1`	
Private	16	16	0.97	1.25	1.99	2.00	2.00	1.33	3.66	1.00×	1.00×	60.6%
Temporal	32	32	0.48	0.63	0.85	1.00	1.99	1.33	2.59	1.00×	1.41×	84.7%
Static	12	20	1.29	1.67	1.99	2.0	2.00	1.33	2.93	1.00×	1.25×	75.6%
Elastic	12/8	20/24/32	1.94	1.67	1.95	1.97	1.99	1.37	2.27	0.98×	1.62×	96.7%

(f) Performance statistics

Figure 2: Performance of co-running `WL#0` **and** `WL#1` **on the four two-core SIMD architectures (Figure 1): (a) workloads** `WL#0` **and** `WL#1`**; (b)-(e) the SIMD utilization of the four SIMD architectures (with each point in the x-axis representing a set of 1000 consecutive cycles); and (f) the performance statistics (with the SIMD utilization calculated for the entire program execution).**

SIMD lane partitioning at the instruction level (Figure 1(d)). We also introduce a compiler to analyze and pass the phase-changing behaviors of a workload to low-level hardware. Thus, the Occamy hardware can dynamically adjust the SIMD lanes and VRF partitioning for the co-running workloads to improve SIMD utilization and performance.

Let us see how Occamy dynamically adjusts the SIMD lanes (and VRF) for the two cores according to the phase behaviors of `WL#0`, as illustrated in Figure 2(e). The Occamy compiler analyzes the phase-changing behaviors of `WL#0` by estimating the operational intensity (FLOPs/byte) of each phase and divides it into a memory-intensive phase (`WL#0.p1`) and a less memory-intensive phase (`WL#0.p2`). Similarly, `WL#1` represents just one compute-intensive phase. By passing this information to the SIMD hardware, Occamy starts by assigning 8 SIMD lanes to `WL#0.p1` and then dynamically scaling it up to 12 when `WL#0.p2` is executed. When the SIMD hardware detects that `WL#0` has completed, Occamy will assign all the 32 lanes to the co-running workload `WL#1`. Compared to the three existing SIMD architectures (Figures 1(a)-(c)), Occamy achieves the best SIMD utilization (96.7%) and the best speedup on `Core1` (1.62×) while achieving nearly the same performance on `Core0` (0.98×) (Figure 2(f)).

Occamy is capable of partitioning the SIMD lanes (and VRF) among the co-running workloads appropriately. Its compiler passes the inferred phase-changing behaviors of a workload to the SIMD hardware. Its lane manager balances the hardware resources among the co-running workloads intelligently based on a roofline model (Section 5). Compared with the three existing SIMD architectures, Occamy provides the best speedup for `WL#1` (1.62× vs. 1.00×, 1.41× and 1.25×) while maintaining nearly the same performance for `WL#0` (0.98× vs. 1.00×, 1.00× and 1.00×) by using fewer hardware resources, achieving the best performance overall.

For the three existing SIMD architectures (Figures 1(a)-(c)), the root causes of their low hardware utilization and subdued performance improvements are that each SIMD co-processor (1) lacks the ability to dynamically partition the hardware resources among the co-running workloads, and (2) is unaware of program semantics needed for partitioning the hardware resources among the co-running workloads appropriately.

3 THE OCCAMY DESIGN

In this section, we give an overview of Occamy. We describe its EM-SIMD execution model, EM-SIMD ISA (i.e., hardware-software interface), hardware (i.e., SIMD co-processor), and software (i.e., the Occamy compiler).

3.1 Elastic MIMD-SIMD Execution Model

Essentially, EM-SIMD is an MIMD-SIMD hybrid execution model, which encompasses the following key aspects:

Table 1: Dedicated registers defined by EM-SIMD ISA (with the vector length being represented at a granularity of 128 bits, so that `<VL>`=2 indicates a vector length configured to 256 bits).

Name	Description
`<OI>`	Operational Intensity of a Phase
`<decision>`	Suggested (i.e., Requested) Vector Length
`<VL>`	Configured (i.e., Current) Vector Length
`<status>`	Success/Fail for Changing Vector Length
`<AL>`	Number of Free SIMD Lanes Available

Parallelism. From the view of a multi-core processor, EM-SIMD leverages an MIMD execution model, which abstracts the processor as consisting of a number of CPU cores that function asynchronously and independently. From the view of each core, EM-SIMD can exploit DLP via SIMD instructions.

Vector/Scalar Execution. A multi-core processor is abstracted as including a SIMD co-processor physically shared by all the CPU cores. The co-processor consists of a set of SIMD lanes, which can be partitioned and assigned to different cores. Logically, the set of lanes assigned to a core form a SIMD unit (a small-sized co-processor) for the core. Each core issues SIMD and scalar instructions sequentially in program order, executes them on its SIMD and scalar units, respectively, and finally, commits them in program order.

Synchronization. After lane partitioning, each core uses its own SIMD unit to execute SIMD instructions with all its lanes operating in lock-step, without the need of explicit synchronization. Our EM-SIMD model does not provide any synchronization between the SIMD units across the cores, since this, if needed, can be defined by the CPU cores.

3.2 EM-SIMD ISA

To support our EM-SIMD execution model, we describe the EM-SIMD ISA used. This provides a hardware-software interface at the ISA level for the software (the programmer or the compiler) to request the hardware to dynamically adjust the vector length for a CPU core at a granularity of 128 bits (i.e., the minimum supported in ARM SVE). While we describe EM-SIMD ISA as an extension to ARM SVE, the basic principle introduced applies to any ISA that makes programs agnostic to the underlying hardware vector length.

Our ISA extension features a set of instructions reading/writing a set of five dedicated registers (Table 1) via `MRS`/`MSR` in ARM to provide the following functionalities.

Phase-Behaviors. For each workload, the software (programmer or compiler) can delineate each of its phases and describe its behavior by writing its operational intensity (FLOPs/byte) into a dedicated register `<OI>` at its beginning and end.

Lane-Partitioning. A lane-partition plan for dividing the SIMD lanes among the co-running workloads, which is usually done by the hardware based on their phase behaviors maintained in `<OI>`, is stored in a dedicated register `<decision>`.

Vector-Length Reconfiguration. For each workload, the software is allowed to reconfigure its vector length by inserting an instruction writing a new one *vl* into a dedicated register `<VL>`, with a dedicated register `<status>` recording whether the reconfiguration is successful or not. If this is successful, then the ensuing SIMD instructions (in program order) will be executed under *vl*

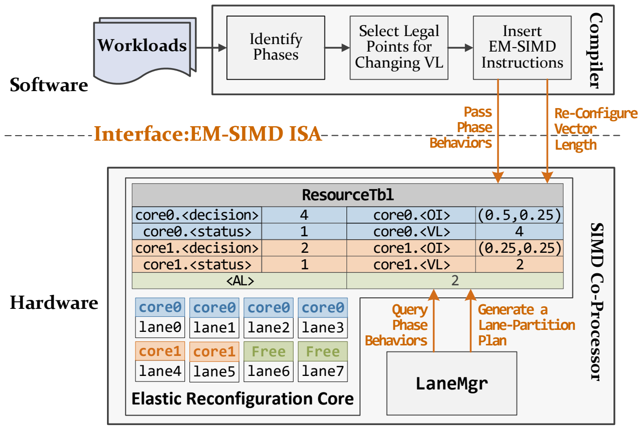


Figure 3: The Occamy design.

until another vector-length reconfiguration succeeds. The Occamy compiler guarantees program correctness for vector-length reconfiguration (Section 6).

3.3 The Hardware and Software Co-Design

To understand the capabilities of the hardware (SIMD co-processor) and software (compiler) in Occamy, Figure 3 sketches the Occamy architecture for a two-core processor, with both interacting via the EM-SIMD ISA as shown.

SIMD Co-Processor. There are two components: an `Elastic Reconfiguration Core` (Section 4) and an on-chip resource manager, `LaneMgr` (Section 5). The `Elastic Reconfiguration Core` consists of a number of homogeneous function units that can be assembled into SIMD data paths of varying widths according to a resource table, `ResourceTbl`, defined by EM-SIMD's five dedicated registers. `LaneMgr` serves to partition elastically all the SIMD lanes across the CPU cores (i.e., generate a *lane-partition plan*) according to the phase behaviors of their co-running workloads whenever a workload enters into or exits from a phase.

Compiler. The Occamy compiler inserts the required EM-SIMD ISA extension instructions into each workload by describing its *phase-behaviors* and requesting *vector-length reconfiguration* (Section 6). To define the behavior of a phase, the compiler will divide a workload into a number of phases (with a loop typically being regarded as a phase) and determine the behavior characteristics of each phase by analyzing its computations and memory accesses. To request for vector-length reconfiguration, the compiler will generate code that queries a new lane-partition plan and requests for the new vector length to be configured at runtime.

Workflow. At the EM-SIMD ISA level, Occamy works in two stages: (1) triggering a new lane partition to generate a set of new vector lengths for the co-running workloads, and (2) reconfiguring the vector length for each CPU core. The Occamy compiler is responsible for determining when these two stages will happen by inserting the required EM-SIMD instructions at some selected program points. Given a set of co-running workloads, the first stage takes place when one workload enters into or exits from one of its phases. For each phase, the Occamy compiler has inserted an EM-SIMD instruction that writes its operational density (currently specified as a pair of values (Section 6.3)) into `<OI>` at its beginning and 0 into `<OI>` at its end. Whenever such an instruction is executed, `LaneMgr` will fetch the phase behaviors of the co-running workloads stored in `<OI>` and then generate a new lane-partition plan in `<decision>`. For each workload, the second stage takes place at a program point when it is legal to switch to the new vector length recommended. At such a point, the workload will execute a set of EM-SIMD instructions that retrieve the recommended vector length from `<decision>` and then request it to be changed by writing it to `<VL>`.

4 OCCAMY: THE MICRO-ARCHITECTURAL SUPPORT

We discuss our micro-architectural support for the EM-SIMD ISA for a two-core processor with Occamy embedded on-chip as shown in Figure 4. In Section 4.1, we look at how EM-SIMD instructions (for reading (writing) the five dedicated registers via `MRS` (`MSR`)) and SVE instructions are transmitted from scalar cores to Occamy. In Section 4.2, we look at three types of data paths in Occamy, the EM-SIMD data path for executing EM-SIMD instructions and the SIMD compute and ld/st data paths for executing SVE instructions, and explain how to reconfigure the widths of the latter *two SIMD data paths*, with respect to our micro-architecture in Figure 5.

4.1 Interaction between Scalar Cores and Occamy

Occamy is tightly coupled with multiple out-of-order scalar (CPU) cores, with instructions transmitted from scalar cores to Occamy for execution. It maintains instruction ordering to deal with data dependency and address overlap hazards.

4.1.1 *Transmitting Instructions.*

All the EM-SIMD and SVE instructions are transmitted from a scalar core to Occamy, enqueued into its `Instruction Pool` (Figure 5) for execution. This happens when each instruction becomes *non-speculative* (*non-squash*) in its scalar pipeline for a scalar score, so that if a pipeline flush occurs in either the scalar core or Occamy, the other pipelines in both will not be affected. In particular, these EM-SIMD and SVE instructions are transmitted after being retired from their scalar pipelines and thus enqueued to Occamy in order. However, there is one exception: the EM-SIMD instruction reading `<decision>` via `MRS` can be speculatively transmitted to Occamy to reduce runtime overhead. The Occamy compiler is responsible for ensuring correctness in this case, as discussed in Section 6.4.

4.1.2 *Maintaining Instruction Ordering.*

Table 2 summarizes how to maintain instruction ordering for a pair of instructions, ⟨`older instruction`, `younger instruction`⟩, by considering a total of nine possibilities for three types of instructions: `Scalar`, `SVE` and `EM-SIMD`. ⟨`Scalar`, `Scalar`⟩ is standard. We will discuss ⟨`SVE`, `SVE`⟩, ⟨`SVE`, `EM-SIMD`⟩ and ⟨`EM-SIMD`, `EM-SIMD`⟩, which are managed by the Occamy hardware, in Section 4.2, and ⟨`EM-SIMD`, `SVE`⟩, which is managed by the Occamy compiler, in Section 6.4. Below we examine the four remaining pairs involving a scalar instruction and an SVE/EM-SIMD instruction, managed by scalar scores.

Table 2: Maintaining instruction ordering.

		Older Instruction		
		Scalar	**SVE**	**EM-SIMD**
Younger Instruction	**Scalar**	Managed by *scalar cores*: standard	Managed by *scalar cores*: delay issuing the younger scalar inst until the SVE/EM-SIMD inst writes back or completes its memory access	
	SVE	Managed by *scalar cores*: delay transmitting the younger inst to Occamy until scalar operands ready or the scalar inst's memory access completed	Managed by the *OCCAMY hardware*: standard	Managed by the Occamy *compiler*: repeatedly writes <VL> until success
	EM-SIMD		Managed by the *OCCAMY hardware*: <VL> changes after the corresp. SIMD pipeline is drained	Managed by the *OCCAMY hardware*: execute the EM-SIMD inst in order

To maintain instruction ordering due to data dependency or address overlap, scalar cores detect data dependency in renaming scalar registers for each instruction and address overlap by using a Memory Ordering Buffer (MOB), which tracks the memory regions within which at least one SVE ld/st instruction has not yet completed, after generating and translating the addresses for all scalar and SVE ld/st instructions.

Let's now consider two cases: (1) ⟨`Scalar`, `SVE`⟩ or ⟨`Scalar`, `EM-SIMD`⟩, and (2) ⟨`SVE`, `Scalar`⟩ or ⟨`EM-SIMD`, `Scalar`⟩. For each pair of instructions in both cases, scalar cores will delay handling the younger one until the scalar operands are ready when ordered due to data dependency or the matched MOB entries are deallocated when ordered due to address overlap. Note that Occamy includes special data paths to write back scalar results and signal the completion of an memory access (as shown in Figure 5). Let us examine one such a pair of instructions for a scalar core. For (1), the scalar core will delay transmitting the younger EM-SIMD/SVE instruction to Occamy, and in case when it has been speculatively transmitted (for an instruction reading `<decision>`), the Occamy compiler will ensure correct execution. For (2), the scalar core will delay issuing the younger scalar instruction, and in case when it has been speculatively issued or hit an MOB entry, its scalar pipeline will flush to maintain the correct ordering.

4.2 Elastic Spatial-Sharing

Figure 5 depicts the Occamy micro-architecture, in which small modifications have been made to the standard out-of-order SIMD pipelines (Section 4.2.1). With these small modifications, Occamy is now capable of flexibly reconfiguring the widths of its SIMD data paths at runtime (Section 4.2.2).

4.2.1 *Modifying OoO SIMD Pipelines.*

We have modified `SIMD Execution Units` and `Register File`, and introduced three new tables, a resource table `ResourceTbl` in `Manager`, and two configuration tables (`ConfigTbls`), with one in `Dispatcher` and one in `Register File`.

Modified Components. We have designed `SIMD Execution Units` as a set of N homogeneous *basic execution units* (`ExeBUs` in Figure 5), with each performing 128-bit SIMD operations (supporting all integer/float-point data types specified in ARMv8-A [36]). The `Register File` adopts a semi-custom physical design, partitioned into N distinct blocks (`RegBlks`), with each consisting of a set of (configured as 160 in our evaluation) 8R4W 128-bit physical vector

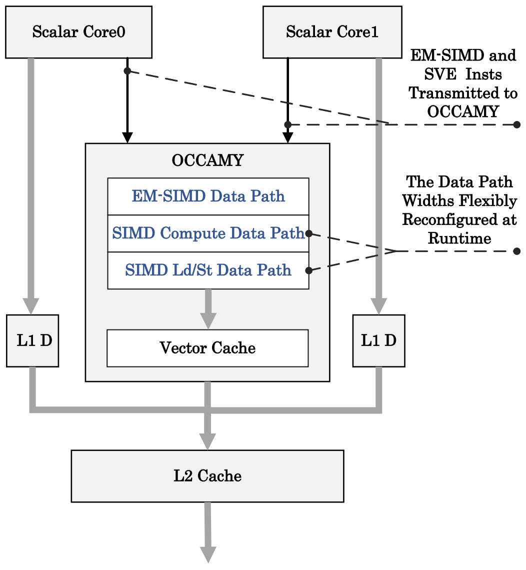


Figure 4: On-chip embedding of OCCAMY.

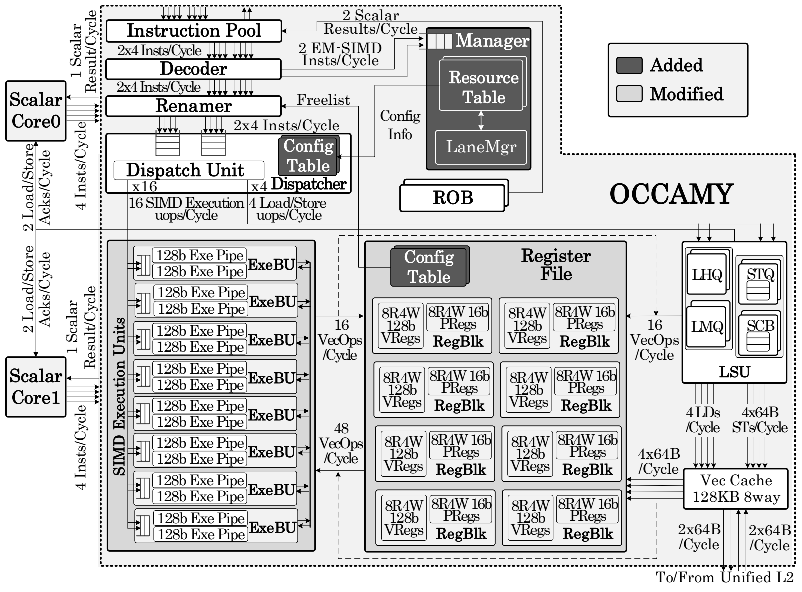


Figure 5: The OCCAMY micro-architecture.

registers and a set of (configured as 64 in our evaluation) 8R4W 16-bit physical predicate registers, denoted VRegs and PRegs, respectively. Each ExeBU is connected to a distinct RegBlk, with both always being assigned to the same scalar core together.

Introduced Tables. Suppose OCCAMY is used to serve C scalar cores. Then ResourceTbl consists of $(4 * C + 1)$ 32-bit registers, including 4 dedicated registers (the first four in Table 1) per core and one dedicated register <AL> shared by all the cores (Figure 3). Each ConfigTbl table consists of N registers, with each using $\lceil \log_2 (C+1) \rceil$ bits to record $(C+1)$ possible states. Each register in Dispatcher.Cfg (ConfigTbl in Dispatcher) records the ownership of an ExeBU, with its states ranging over {free, core0, core1, ...}. Similarly, each register in RegFile.Cfg (ConfigTbl in Register File) records the ownership of a RegBlk.

To scale OCCAMY up, some changes are to be made:

- The three tables introduced, must be enlarged, in order to record the states corresponding to more cores added;
- The data paths and control logic in some components, e.g., instruction pipelines in Instruction Pool/Decoder/ Renamer/ Dispatcher/LSU/ROB, must be expanded; and
- Other function/storage units, e.g., SIMD Execution Units/ Register File/VecCache, can be optionally increased in quantity in order to improve performance.

Increasing the first two types of resources adds little area cost to OCCAMY, e.g., 3% when scaling from 2 to 4 cores.

4.2.2 Reconfiguring SIMD Data Paths. At runtime, OCCAMY can dynamically build SIMD data paths (two per core as shown in Figure 4) with a width of $128 \times l$ bits for core c, once an EM-SIMD instruction writing l to <VL> has been received from core c and successfully executed. This instruction will update the three introduced tables, with the ownership of l ExeBUs set to core c in Dispatcher.Cfg and the ownership of l RegBlks connected to these l ExeBUs also set to core c in RegFile.Cfg. Thus, the ensuing SVE instructions from core c will be executed on the SIMD data paths consisting of these l ExeBUs and l RegBlks until another EM-SIMD instruction writing <VL> from core c has been successfully executed.

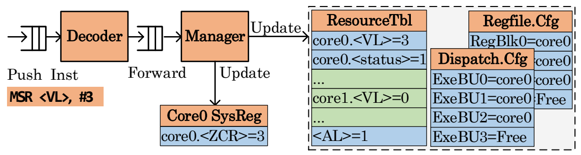
Figures 6(a)-(c) illustrate the execution of EM-SIMD, SVE compute, and SVE ld/st instructions, as explained below.

Executing EM-SIMD Instructions. All EM-SIMD instructions are processed in order in an EM-SIMD data path shared by all the cores (Figure 4). Suppose that OCCAMY has received a vector-length re-configuration instruction MSR <VL>, l from core c. It will update a set of registers atomically if (1) c.<VL> + <AL> $\geq l$ (number of requested lanes is available), and (2) the SIMD pipeline associated with core c is drained:

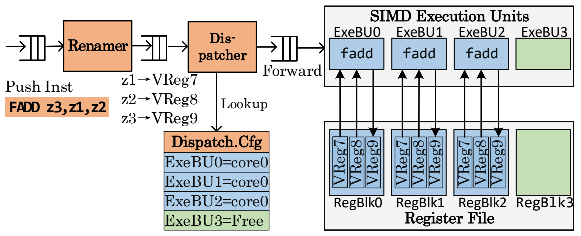
- ResourceTbl: Set <AL> to c.<VL> + <AL> $- l$, c.<VL> to l, and c.<status> to 1.
- Dispatch.Cfg **and** RegFile.Cfg: Free all the ExeBUs and RegBlks previously occupied by core c and assign l free ExeBUs and their associated RegBlks to core c. The data values in these freed RegBlks are not preserved since the OCCAMY compiler ensures that they will not be used.
- **Core** c: Set the SVE control register <ZCR>, which stores the vector length according to the ARM specification, to l.

As for instruction ordering involving SVE and EM-SIMD instructions (Table 1), ⟨EM-SIMD, EM-SIMD⟩ is maintained since both are executed in order, and ⟨SVE, EM-SIMD⟩ is maintained since the SVE instruction is executed under the old vector length and the new one will only come into effect after the SIMD pipeline associated with core c is drained.

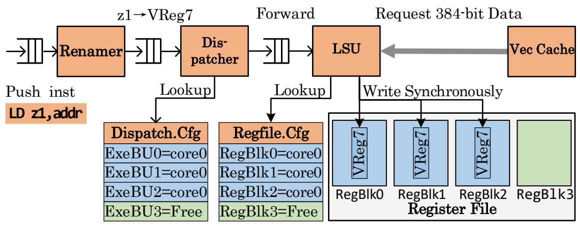
Executing SVE Compute Instructions. When an SVE compute instruction from core c reaches Dispatcher, l identical μops are dispatched with each routed to an ExeBU owned by core c according to Dispatch.Cfg. Each ExeBU fetches the operands from its RegBlk, operates on them, and produces a 128-bit result. The results from the l ExeBUs are written back synchronously into the l physical vector



(a) An EM-SIMD instruction received from `core0`



(b) An SVE compute instruction received from `core0`



(c) An SVE ld/st instruction received from `core0`

Figure 6: Execution of an EM-SIMD/SVE instruction in core 0.

registers residing in the l `RegBlk`s, to ensure atomic instruction execution.

Executing SVE ld/st Instructions. When an SVE ld/st instruction from core c reaches `Dispatcher`, the dispatched μop will be appended with the width of its memory access, i.e., $128 \times l$ bits according to `Dispatch.Cfg`. For an SVE load instruction, the `LSU` will request $128 \times l$ bits data from `VecCache` and store the data synchronously into the l physical vector registers residing in the l `RegBlk`s according to `RegFile.Cfg`. Similarly, an SVE store instruction is handled.

Finally, Occamy proceeds in the standard manner to maintain instruction ordering for ⟨SVE, SVE⟩ (Table 1), where SVE is either an SVE compute or SVE ld/st instruction.

5 OCCAMY: THE SIMD LANE MANAGER

The SIMD lane manager, `LaneMgr` (Figure 5), is implemented in hardware for deciding when and how to re-partition all the SIMD lanes among a set of co-running workloads. `LaneMgr` uses a monitor listening for a lane re-partition, i.e., `MSR` instruction in a workload that writes `<OI>` to indicate a phase-changing point in the workload (with the beginning of a phase set as its operational intensity (in FLOPs/Byte) and the end as 0). On seeing such an instruction, `LaneMgr` will go ahead to find a lane-partition plan. This is done by obtaining the phase behaviors of the co-running workloads from `<OI>`, determining a lane partition plan by using a vector-length-aware roofline model, and finally, recording the plan in `<decision>`.

We need to address two challenges faced in lane partitioning: (1) how to predict the performance of a workload for a given number of `ExeBU`s, and (2) how to design an algorithm to partition all the `ExeBU`s across all the co-running workloads.

We formalize this problem for a C-core processor as follows. Given $\texttt{WL} = \{\texttt{WL}_1, ...\texttt{WL}_M\}$ as M co-running workloads that are currently executing some phases, i.e., vectorized loops (where $M \leq C$) and N `ExeBU`s to be partitioned, we aim to find a *lane partition plan*, $\{vl_1, ..., vl_M\}$, that maximizes the overall performance of `WL`, where vl_i represents the number of `ExeBU`s allocated to $\texttt{WL}_\texttt{i}$, subject to the following constraints:

$$\forall\, i : vl_i > 0 \quad \text{and} \quad \sum_{i=1}^{M} vl_i \leq N \tag{1}$$

We assume that lane partitioning and task scheduling work independently so that we can consider only the workloads currently scheduled to execute. For other policies, it may be more profitable to let both work together. Under commonly used OS policies (e.g., first-come-first-serve), resource contention will not happen when compute-intensive tasks are scheduled to run, since context-switching will trigger a new lane partitioning process. Specifically, the OS will save the contexts (including the five EM-SIMD dedicated registers) when all the pipelines (including those in Occamy) are drained, and restore `<OI>` using `MSR` (when `<OI>`$\neq 0$), to trigger lane partitioning.

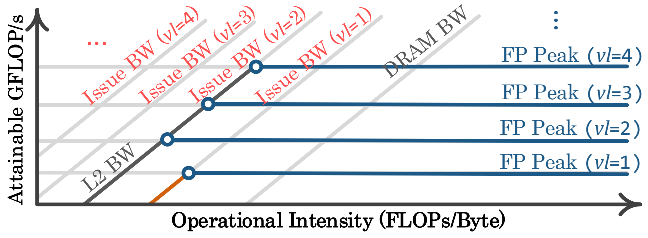
5.1 Vector-Length-Aware Roofline Model

To predict the performance of a workload for a given number of `ExeBU`s, we use a roofline model to represent the architecture-specific performance ceilings. The basic roofline model has been extended by considering a range of hardware issues, e.g., cache hierarchy [21, 44], NUMA [28], software prefetching and locality [40, 41], and instruction- and task-level parallelism [40]. The key challenge is that Occamy supports variable-length SIMD mode, leading to variable peak performance for SIMD compute instructions and variable widths for SIMD load/store instructions. We have extended the basic roofline model to accounts for such variability by modeling the *computation*, *memory-bandwidth*, and *SIMD-issue-bandwidth ceilings*, where the first and last are related to the vector length.

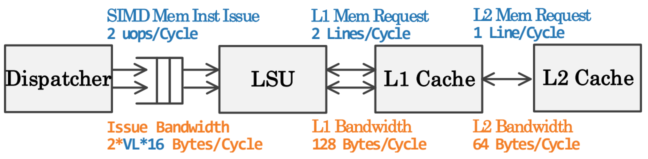
Computation Ceilings. It is simple to compute the variable-vector-length SIMD performance as the performance ceilings of one `ExeBU` multiplied by vl, illustrated by the horizontal lines annotated with `FP peak` ($1 \leq vl \leq 4$) in Figure 7(a) for peak floating-point performance in single precision.

Memory-Bandwidth Ceilings. We leverage the hierarchical roofline model [21] to obtain the bandwidth ceilings of (unified) L2 cache and memory, as shown by the two lines annotated with `L2 BW` and `DRAM BW` in Figure 7(a), respectively.

SIMD-Issue-Bandwidth Ceilings. We have observed that the bottleneck for memory access will shift from memory bandwidth to SIMD issue bandwidth, when the SIMD ld/st data path is not wide enough which happens potentially in variable vector-length SIMD processing. Figure 7(b) depicts an abstract SIMD ld/st data path. In particular, `Dispatcher` can send at most two vector memory μops to `LSU` per cycle, meaning that `LSU` may request at most $2 \times 128 \times vl$ bits ($32 \times vl$ bytes) data from `VecCache` per cycle. When a core



(a) Computation, memory-bandwidth and SIMD-issue-bandwidth ceilings



(b) Abstract vector load/store data path

Figure 7: A Vector-length-aware roofline model.

is assigned with a small number of lanes, the SIMD issue bandwidth (32B/cycle when $vl = 1$) will be smaller than L2 BW (fixed at 64B/cycle), becoming the bottleneck for memory access. So we define the *SIMD issue bandwidth* (B/cycle) of a core as:

$$\text{SIMD-Issue\_BW} = \text{SIMD-issue\_}width \times vl \times 16 \tag{2}$$

where *SIMD-issue_width* gives the number of vector memory μops dispatched per cycle, and 16 represents 128 bits in bytes.

Finally, we have plotted the SIMD-issue bandwidth ceilings in Figure 7(a), as the four lines in light gray for $1 \le vl \le 4$.

5.2 Algorithm

LaneMgr uses a greedy algorithm to partition a set of ExeBUs (i.e., SIMD lanes) for a set of co-running workloads in three steps. First, one ExeBU is assigned to each workload currently executing a phase (with its operational intensity <OI> ≠ 0).

Second, the algorithm iteratively assigns the remaining ExeBUs to these workloads. In each iteration, it sorts the workloads in decreasing order of their net performance gains obtained when each workload receives one extra ExeBU, and then assigns one ExeBU to each workload with a positive gain in that order. The net performance gain of a workload WL (running in the current phase recorded in <OI>) when moving from vl to $vl+1$ ExeBUs is estimated as follows:

$$\text{net\_perf\_gain}_{vl,vl+1} = AP_{vl+1}(\texttt{<OI>}) - AP_{vl}(\texttt{<OI>}) \tag{3}$$

where $AP_l(\texttt{<OI>})$ denotes the performance attainable by using our roofline model that accounts for both the computation and SIMD-issue-bandwidth ceilings for vector length l, denoted FP_peak_l and $SIMD\text{-}issue\_BW_l$, as well as memory ceiling mem_BW (which is specific to a chosen level in memory hierarchy, e.g., *L2 BW* or *DRAM BW* but unrelated to l):

$$\begin{aligned} AP_l(\texttt{<OI>}) = \min\{&FP\_peak_l,\ SIMD\text{-}issue\_BW_l \\ &\times \texttt{<OI>}.issue,\ mem\_BW \times \texttt{<OI>}.mem\} \end{aligned} \tag{4}$$

where <OI>.*issue* and <OI>.*mem* are defined in Section 6.3.

Finally, the algorithm terminates when all the ExeBUs available have been allocated or no workload can have any further performance gain even when given more ExeBUs.

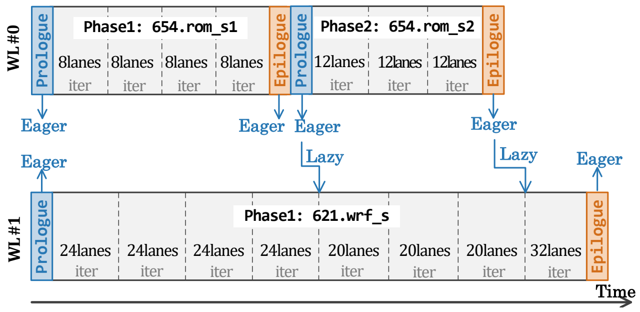


Figure 8: Eager-lazy lane partitioning for Figure 2(a).

Let us look at its fairness. When only compute-intensive workloads are co-running, the SIMD lanes will be divided equally, avoiding hardware contention, since each workload always benefits with more ExeBUs allocated according to Equations (3) and (4). When memory- and compute-intensive workloads are co-running, OCCAMY ensures that each workload receives at least one ExeBU, avoiding "starving out" completely. To improve fairness further, performing lane partitioning and OS scheduling together may be preferred.

6 OCCAMY: THE COMPILER

Given a workload, the OCCAMY compiler inserts EM-SIMD instructions at its selected program points to (1) trigger a lane partition plan to be generated, and (2) ask for a suggested new vector length to be set. In Section 6.1, we introduce an *eager-lazy lane partitioning* approach that performs (1) eagerly but (2) lazily. In Section 6.2, we incorporate this approach into a vectorized loop. In Section 6.3, we explain how to specify the phase behaviors of a workload at its eager lane partition points. In Section 6.4, we describe how to ensure program correctness at the lazy lane partition points of each workload.

6.1 Eage-Lazy Lane Partitioning

The key idea is to prevent a workload from changing the vector length used in the middle of an iteration of a vectorized loop, so that program correctness can be easily maintained. According to our eager-lazy partitioning approach, a workload can trigger a lane partition plan to be generated eagerly when it enters into or exits from a phase (i.e., a loop). However, a co-running workload can only request lazily for the new vector length recommended by this plan to be reconfigured at the the beginning of an iteration of a loop (i.e., one of its phases).

Figure 8 illustrates one possible lane-partitioning scenario for our motivating example in Figure 2. WL#0 contains four eager partition points but only two lazy partition points. The prologue and epilogue of a phase are explained below.

6.2 Lane-Partitioning-Enabled Vectorized Code

Figure 9 gives a vectorized code with EM-SIMD instructions inserted to support our eager-lazy lane partitioning approach.

Code for Eager Partitioning. The eager lane partition points are just before and after a vectorized loop (i.e., a phase), i.e., *Phase Prologue* and *Phase Epilogue*, respectively. In *Phase Prologue*, the MSR <OI>,X1 instruction is inserted to pass the behavior of this phase to LaneMgr to trigger a new lane partition plan to be generated and the

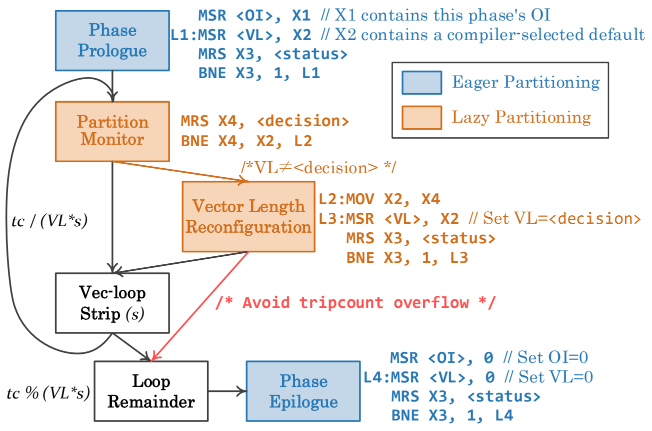


Figure 9: A vectorized loop incorporating eager-lazy lane partitioning for supporting elastic vector lengths, where $X1-X4$ are general registers, tc denotes the (scalar) loop tripcount, and s represents the length of a stripmined loop segment.

three-instruction loop sets the vector length to a compiler-selected default (since it was 0 on arriving at this point). In *Phase Epilogue*, `LaneMgr` is triggered again to release all the lanes occupied by the loop.

Code for Lazy Partitioning. The lazy lane partition points reside at the beginning of each iteration in a vectorized loop, marked as *Partitioning Monitor* and *Vector Length Reconfiguration*. The *Partition Monitor* checks whether a co-running workload has caused a new lane partition plan to be generated by using `MRS X4,<decision>`. If this happens, the *Vector Length Reconfiguration* code will be executed to switch to the new vector length if it is different from the current one. The branch to Loop Remainder is needed so that some extra loop bound check is done to avoid a tripcount overflow.

6.3 Phase Behavior Analysis

To describe the behavior of a phase (loop) delineated by *Phase Prologue* and *Phase Epilogue* (Figure 9), its operational intensity is written into <OI> in *Phase Prologue* and 0 into <OI> in *Phase Epilogue*. The operational intensity (in Flops/Byte) of a vectorized loop is defined in terms of the following two quantities (without considering its trip count):

$$\begin{aligned} \texttt{<OI>}.issue &= \frac{comp}{\sum_{i=1}^{i\le mem} byte^{i}_{type}} \\ \texttt{<OI>}.mem &= \frac{comp}{fp} \end{aligned} \tag{5}$$

where *comp* (*mem*) is the number of SIMD compute (memory-access) instructions in the loop body, $byte^i_{type}$ is the size of the data type used in the i-th memory access instructions (in bytes), and fp is the memory footprint of SIMD memory-access instructions in one iteration (with data reuse considered). In the absence of data reuse, <OI>.*issue* = <OI>.*mem*.

In addition, two optimizations are included to avoid frequent lane re-partitions. First, code hoisting is leveraged to hoist *Phase Prologue* and *Phase Epilogue* from inside a loop to outside to avoid chaining phase-changing points for the same phase. Second, multiple-version code generation [6, 26] is leveraged to handle loops with small trip counts, so that we can choose their non-vectorized code variants at runtime.

Currently, Occamy supports elastic vector lengths only for vectorized loops containing no synchronization inside.

6.4 Program Correctness

We can leverage any existing vectorization algorithm [7, 8, 23, 39, 48] that can produce a vectorized loop for the maximum vector length available. To support lane re-partitioning, the Occamy compiler has also generated extra code for a workload to execute after it has received a new vector length (Figure 9) for (1) re-initializing the SIMD registers containing loop-invariants for the new vector length, (2) completing the partial result for a reduction variable obtained so far so that the partial result can be used to start the next round of reduction under the new vector length. Nothing needs to be done for a SIMD register using fresh values loaded from memory (directly or indirectly) at the beginning of each iteration.

To maintain instruction ordering for ⟨`EM-SIMD`, `SVE`⟩ (Table 1), the compiler encloses an instruction writing `<VL>` with a `while` loop testing whether `<status>` has turned into 1 in order to ensure that the subsequent SVE instructions are executed under the new vector length (Figure 9). In addition, an instruction reading `<decision>` in the *Partitioning Monitor* may read an old value due to speculation, but this will not lead to a program error, since the Occamy compiler ensures program correctness for any vector length set at this point.

7 EVALUATION

We show that Occamy improves the prior art in exploiting DLP by supporting elastic spatial-sharing for SIMD resources.

7.1 Experimental Setups

Workloads. We have created 34 workloads from 28 loops in `SPEC-CPU2017` [10] and 14 kernels in `OpenCV`, as given in Table 3. For `SPECSpeedInt`, we have considered a total of 76 loops that are vectorized by `LLVM13.0/gfortran7.5` and actually executed under the `REF` input. For `SPECSpeedFP`, we have considered a total of 186 such vectorized and executed loops such that each loop resides in a hot function that consumes more than 10% of the execution time of its containing program. For these 262 loops, we have further excluded 234 since they each simply update an array with a single operation or iterate for less than 128 iterations. To evaluate Occamy, the loops in FORTRAN have been rewritten in C++. For `OpenCV`, we have selected 14 memory-intensive and compute-intensive kernels from the `core` and `imgproc` modules.

Our workloads are compiled by the Occamy compiler, with its elastic vectorization implemented in `LLVM 13.0`. The compiler options `−O3` and `−scalable − vectorization = preferred` are applied, targeting `armv8.2 − a + sve`.

Architectures and Configurations. We have modeled a 2-core ARM processor, interfacing with the four SIMD architectures in Figure 1, by using the gem5 cycle-level simulation infrastructure [5]. Table 4 lists their micro-architectural parameters. In `Private`, only core-private SIMD lanes are used as in Figure 1(a), implemented as standard fixed-vector-length SIMD units dedicated to each core. `FTS` and `VLS` are from [4], representing the temporal-sharing and static spatial-sharing alternatives shown in Figures 1(b) and (c).

Table 3: Workloads (where oi_{mem} represents the operational intensity defined in Equation (5) (with data reuse considered)).

SPEC	Phases (oi_{mem})	SPEC	Phases (oi_{mem})	OpenCV	Phases (oi_{mem})
WL1	select_atoms2 (0.25)+ step3d_uv2 (0.09)	WL13	set_vbc2 (0.56)	WL1	fitLine2D (0.92)
WL2	select_atoms1 (0.25) + step3d_uv4 (0.13)	WL14	set_vbc1 (0.56)	WL2	addWeight (0.33) + compare(0.25)
WL3	rhs3d1 (0.13) + select_atoms3(0.25)	WL15	rhs3d5 (0.32)	WL3	rgb2xyz (0.63)
WL4	select_atoms4(0.083) + select_atoms5(0.75)	WL16	wsm51(1)	WL4	calcDist3D (0.875)
WL5	step3d_uv1(0.11)+ rhs3d7(0.17)	WL17	wsm52(1)	WL5	rgb2hsv (1.83)
WL6	rho_eos1 (0.09)+ rho_eos4 (0.16)	WL18	wsm53(0.56)	WL6	accProd (0.17) + dotProd(0.25)
WL7	rho_eos5 (0.08)+ select_atoms3 (0.25)	WL19	rho_eos2 (0.25)	WL7	normL1 (0.5) + normL2(0.25)
WL8	rho_eos2 (0.25)+ rho_eos6(0.06)	WL20	sff2 (0.13)+ sff5(0.21)	WL8	compare (0.25) + accProd (0.17)
WL9	wsm53 (0.56)+ select_atoms5 (0.25)	WL21	sff5 (0.16)+	WL9	blend(0.3) +fitLine3D(0.44)
WL10	rhs3d1 (0.13)+ rho_eos4 (0.16)		rho_eos6 (0.06)	WL10	dotProd (0.25) + addWeight(0.33)
WL11	step2d1 (0.22)+ step2d6 (0.18)	WL22	rho_eos2 (0.08)+	WL11	blend (0.3) + compare(0.25)
WL12	step3d_uv3(0.13)+ step3d_uv1 (0.11)		step3d_uv1 (0.11)	WL12	rgb2ycrcb(0.42) + rgb2gray(0.31)

Occamy is our SIMD architecture supporting elastic spatial-sharing in Figure 1(d).

We have formed 25 pairs of co-running workloads, with 1 **<memory, memory>**, 2 **<compute, compute>**, and 22 **<memory, compute>**. In the last case, we run the memory-intensive workload on Core0 and the compute-intensive workload on Core1. All the averages used are geometric means.

Table 4: Micro-architectural parameters for Figure 1.

Private	FTS	VLS	Occamy
2 Scalar Cores, TaiShan V110-Extensive Architecture [42], 8-Issue ARMv8-A Superscalar Out-of-Order Pipeline, 2GHz			
16 Lanes/Core	32 Lanes for a 2-Core Configuration		
Vector Issue Width - 4 (SIMD Execution Units - 2, ld/st Units - 2)			
Area: 1.263mm^2		Area: 1.265mm^2	
VRF: 10KB/Core	VRF: 20KB for a 2-Core Configuration		
64KB Private L1I/D per Scalar Core - Latency 4 Cycles - Cache Line 64B 128KB Vec Cache - Latency 5 cycles - Cache Line 64B 8MB Shared Unified L2 - Latency 18 Cycles - Cache Line 64B 4 GB 64GB/s Memory			

7.2 Performance

Figure 10 gives the performance for the four SIMD architectures (with Private as the baseline). FTS, VLS, and Occamy outperform Private by 1.20×, 1.11× and 1.39×, respectively, on average, for compute-intensive workloads executed on Core1 while faring similarly as Private for memory-intensive workloads on Core0, since they are capable of assigning more SIMD lanes to compute-intensive workloads. In particular, Occamy is the best performer by adopting a more flexible sharing policy that requires little hardware resources. FTS will consume abundant physical registers to keep separate contexts for each core while VLS adopts a static sharing policy that is not well suited for all the phases.

Figure 11 gives their SIMD utilization. On average, Private is the worst achieving 63.2%. By sharing the SIMD lanes, FTS, VLS, and Occamy have improved it to 72.5%, 70.8%, and 84.2%, respectively, with Occamy as the best.

7.3 Area

We have implemented the key components in the four SIMD architectures, e.g., issue pipe/hiq/lhq/stq in LSU and ResourceTbl/control logic/fifo in Manager in RTL and synthesized them with TMSC 7nm technology by using Synopsys Design Compiler 2019.03 SP1 [38] under the implemented design at TT/0.75V/85°C/2GHz. In addition, the key parameters in our simulator, e.g., memory access and execution pipeline latencies (Table 4) have been set based on this RTL simulation.

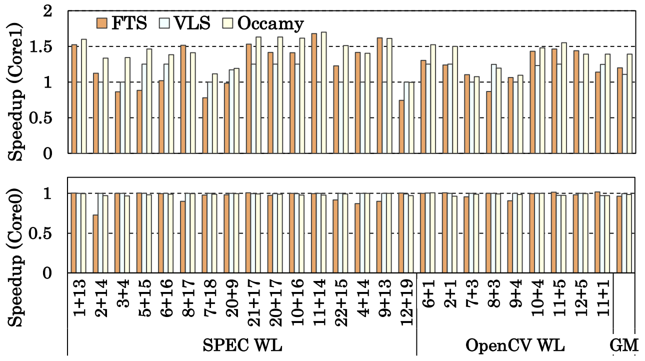


Figure 10: Speedups (with Private as the baseline).

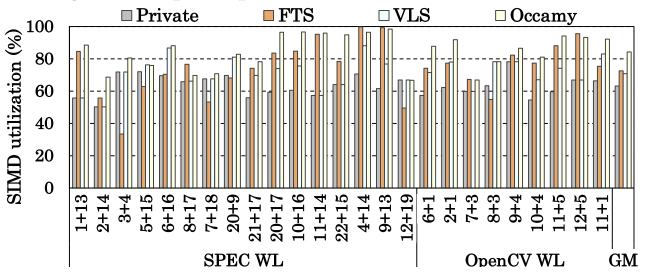


Figure 11: SIMD utilization.

Figure 12 gives an area break down for the four SIMD architectures, indicating that SIMD execution units, ld/st units and register files are the most area-consuming components. Note that Occamy does not introduce notable area costs since the Manager (Figure 5) takes less than 1% of the total area.

In particular, FTS enables temporal sharing by multiplexing but it must maintain separate contexts for each core in register files. As a result, the physical register pressure increases significantly, causing instructions to be blocked frequently in the Renamer waiting for free registers. Figure 13 shows the fraction of cycles with some instructions blocked, indicating that renaming stalls happen in over 70% cycles on FTS on average compared with hardly any on the other three architectures.

7.4 Case Studies

We examine three different categories of memory/compute-intensive co-running workloads, <memory, compute> , <compute, compute> and <memory, memory>. Note that Occamy aims mainly to improve performance for the first category. After that, we look at a case where FTS outperforms Occamy.

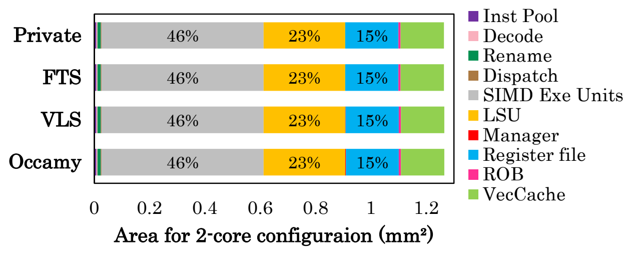


Figure 12: Area breakdown.

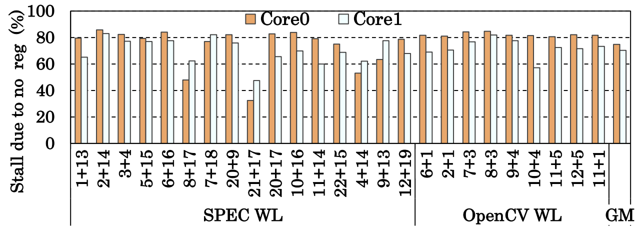


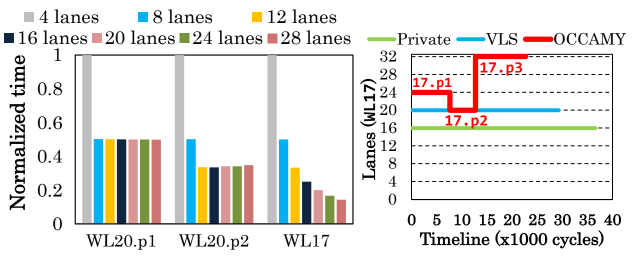
Figure 13: Fraction of cycles waiting for free registers on FTS**.**

Case 1: <memory, compute>. Let us analyze WL20 + W17 in Figure 14, where WL20 has two phases p1 and p2 and WL17 has just one (Table 3). Figure 14(a) gives their execution times with varying SIMD resources. For WL20.p1 (WL20.p2), no performance gain is attained when more than 8 (12) lanes are used. In contrast, WL17 can always benefit with more lanes. This suggests that why Private is ineffective for such a workload since it always assigns 16 lanes to each workload.

In Occamy, LaneMgr allocates just enough lanes to each workload eastically. Specifically, WL20.p1 (WL20.p2) receives 8 (12 lanes) but WL17 benefits with more lanes. As a result, LaneMgr allocates to it all the remaining free lanes, i.e., 24/20/32 to p1/p2/p3, where p3 is the last phase when W17 runs alone (Figure 14(b)). This vector length flexibility leads to improved SIMD issue rates for memory-intensive phases, from 0.96/1.22 on Private to 1.88/1.65 for WL20.p1/p2, as revealed in Figure 14(c). Thus, Occamy has boosted the performance of WL17 by 1.63× over Private, without reducing the performance of WL20 (Figure 10).

VLS statically allocates 12 lanes to WL20, resulting in 4 lanes underutilized in WL20.p1, causing the SIMD issue rate to drop to 1.26 compared with 1.88 on Occamy. Unfortunately, WL17 cannot use these wasted lanes, resulting in a speedup of 1.25× only. By exploiting temporal sharing, FTS can more flexibly adjust the SIMD issue rates in the three phases, resulting in 0.77/0.89/2 for WL17.p1/p2/p3, as the operational intensity of WL20 changes. However, this flexibility comes at a price of causing instructions to be blocked in Renamer waiting for free registers for 82.9/65.5% of the cycles on Core0/Core1, enabling FTS to achieve only a speedup of 1.42× for WL17.

Case 2: <compute, compute>. Let us look at WL9 + W13. Both workloads, when co-running, can fully utilize the SIMD resources on all the four SIMD architectures, but after WL9 finishes, FTS/Occamy allows WL13 to use the SIMD lanes released, improving its performance by the same speedup, 1.61×, over Private (Figure 10). While VLS, which lacks this flexibility, yields no performance improvement (1.0×).



(a) Normalized execution times **(b) Lane partitioning**

Phases		20.p1	20.p2	17.p1	17.p2	17.p3
SIMDIssueRate (insts per cycle)	Private	0.96	1.22	1.95	2	2
	VLS	1.26	1.65	1.95	2	2
	FTS	0.49	0.59	0.77	0.89	2
	Occamy	1.88	1.65	1.91	1.99	1.98
Stall (cycles)	FTS	6365	4443	6614	4604	5820
	Occamy	0	0	0	0	0

(c) SIMD-issue rate and cycles stalled waiting for free registers

Figure 14: Workload WL20 + W17**.**

Table 5: Attainable performance (GFLOPs/s) for WL8.p1**.**

VL	4	8	12	16	20/24/28/32
SIMDIssueBound	**5.3**	**10.7**	**16**	21.3	26.7/32/37.3/42.7
MemBound	16	16	**16**	**16**	**16**
CompBound	8	16	24	32	40/48/56/64
Performance	5.3	10.7	16	16	16

Case 3: <memory, memory>. For WL12 + W19, FTS, VTS and Occamy that enable sharing SIMD lanes perform similarly as Private (Figure 10) as both workloads are memory-bound.

Case 4: FTS Outperforming Occamy. For WL8 + W17, with the first phase of WL8 (i.e., WL8.p1) exhibiting different operational intensities for its SIMD issue and memory bandwidths ($oi_{issue} = 0.17$ and $oi_{mem} = 0.25$) due to data reuse.

Table 5 gives its attainable performance results obtained by Equation (4), which are bounded by instruction issue when $VL < 12$ lanes. Occamy has assigned 12 lanes to WL8.p1, which would be otherwise assigned with 8 lanes by considering only memory bandwidth and computation requirements, achieving 1.41×. These 4 extra lanes, which are underutilized, are traded for issue bandwidth to maintain comparable performance as Private. In contrast, FTS issues all instructions to the full SIMD data path with the 32 lanes, avoiding the under-utilization above. As WL17 receives more SIMD resources, leading to a slightly higher speedup of 1.52×.

7.5 Overhead Analysis

Figure 15 gives the runtime overhead of Occamy, which spends an average of 0.5% of the execution time of a workload to facilitate its EM-SIMD execution. This overhead consists of monitoring lane partitioning decisions and adjusting the vector lengths. The former is negligible (0.3% on average), since instructions reading <decision> are speculatively sent to Occamy (Section 4.1). The latter is 0.2% on average, spent on draining the SIMD pipelines for vector-length re-configuration. This is also negligible since lane partitioning does not happen frequently in vectorized inner loops (Section 6.3).

7.6 Scalability

Figure 16 compares Occamy further with the three existing SIMD architectures (Figure 1(a)-(c)) by using four groups of SPEC workloads (with Private as the baseline). For each of the first three

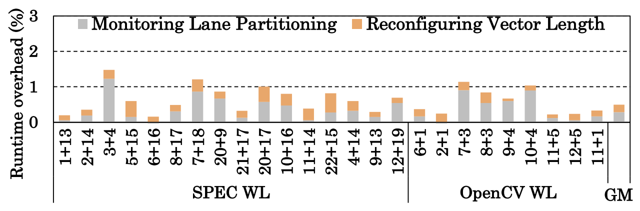


Figure 15: Overhead for supporting elastic spatial-sharing.

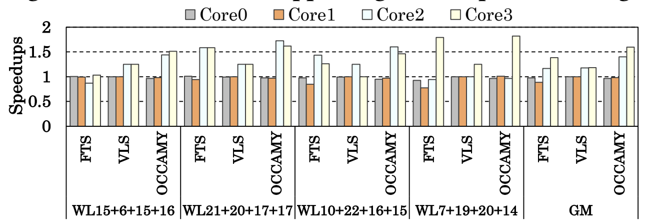


Figure 16: Speedups on 4 cores (with `Private` as the baseline).

groups, we combine two pairs of **<memory, compute>** workloads used in Figure 10 by running the two memory-intensive workloads on `Core0` and `Core1` and the two compute-intensive workloads on `Core2` and `Core3`. We form the last group by running three memory-intensive workloads on `Core0` – `Core2` and one compute-intensive workload on `Core3` with these workloads not paired in Figure 10.

Compared to the two-core case (Figure 10), three observations are in order. (1) Occamy fares similarly as `Private`, `FTS` and `VLS` on `Core0` and `Core1` but delivers better speedups on `Core2` and `Core3`. (2) Occamy exhibits good scalability when moving from 2 to 4 cores. In particular, the solo compute-intensive group, `WL14`, in the last group runs faster than in the 2-core case as expected. (3) For `FTS`, achieving temporal sharing has shifted the bottleneck from SIMD lanes to other hardware resources, which may be expanded at the expense of chip area [38], but this is often impractical. Nevertheless, by maintaining the same number of vector and predicate physical registers per core as in the two-core case (and thus incurring a 33.5% more chip area than the other three architectures), `FTS` still under-performs Occamy.

8 RELATED WORK

Some earlier works have designed vector accelerators that support mixed SIMD and non-SIMD execution models. For example, SIMD has been mixed with MIMD, by decomposing SIMD units into fine-grained computation tiles and allocating them to different tasks [2] or enabling different threads to operate independently in MIMD or lockstep SIMD mode [22, 24, 25]. Our EM-SIMD execution model is similar, but these earlier works achieve vector-length flexibility with hardware innovations while Occamy leverages compiler support to adjust the vector lengths needed. For example, in [2], an instruction forwarding network instead of the fetching logic and I-cache was introduced, enabling instructions to be passed directly among the neighboring cores. Occamy passes the phase behaviors of co-running workloads to the hardware to improve SIMD lane partitioning. Thus, both may be combined in future work. Alternatively, SIMD can be mixed with ILP by reconfiguring SIMD units into VLIW processors [33] or enabling SIMD lanes to be morphed into a feed-forward subgraph accelerator [11]. Furthermore, DLP can also be extracted together with TLP by assembling vector instructions at runtime [15, 16], allowing idle vector lanes to run short-vector or scalar threads [35], and splitting a SIMD group into smaller SIMD sub-groups [30]. In particular, Gobieski et al. [13] exploit the dataflows in a sequence of vector instructions and Dyser et al. [14] achieve SIMD flexibility by using a configurable data path. However, these works focused on individual workloads while Occamy focuses on co-running workloads.

Some recent vector accelerators are designed to interface with multiple scalar cores. Apple's AMX [17–19] and the NEC vector engine [20, 43] embrace temporal sharing in order to hide memory access latency. Beldianu and Ziavras [3, 4, 29] have introduced coarse- and fine-grained temporal sharing policies including a static spatial-sharing policy and found fine-grained temporal sharing to be the most effective. In contrast, Occamy adopts an elastic spatial-sharing policy, capable of balancing performance and area costs.

Vector-length-agnostic programming is now supported by emerging vector ISAs, e.g., the RISC-V vector extension [1], ARM SVE/SVE2 [27, 37], and unlimited vector extension [12]. In addition, some compilers may also generate vectorized code with varying vector lengths [34, 47] for a fixed-width SIMD data path. Unlike these works, Occamy can adjust the vector lengths at runtime by using a hardware-software co-design.

In this paper, Occamy is introduced as a SIMD processor for improving SIMD utilization for co-running workloads with memory- and compute-intensive workloads on a multi-core CPU processor. Thus, our work differs from those on exploiting DLP on GPUs [9, 32]. GPUs offer high throughput by executing threads in a SPMD fashion, with each vector lane shared temporally by several thread warps. Its underlying architecture employs a large register file to achieve this, since otherwise the their vectors lanes will be severely underutilized. For CPUs, however, this solution is impractical due to a huge area cost incurred. Thus, GPUs benefit workloads consisting of batch computations with a massive amount of SPMD parallelism at the thread level. For CPUs, their SIMD co-processors provide low latency by issuing vector instructions directly from their scalar pipelines, benefiting workloads with SIMD parallelism possibly scattered in their different phases at the instruction level. Currently, existing SIMD architectures for CPUs cannot exploit DLP adequately for co-running workloads with both memory- and compute-intensive workloads via temporal sharing [17–19] or static spatial-sharing [3, 4, 29] as evaluated in this paper. In contrast, Occamy is capable of adjusting the SIMD resources dynamically for such workloads, adapting well to their phase behaviors.

Finally, the compiler community has developed a number of SIMD vectorizers [7, 8, 23, 39, 48], which can be leveraged to further improve the effectiveness of the Occamy compiler.

9 CONCLUSION

This paper proposes Occamy, a SIMD co-processor that can be shared by multiple CPU cores, with an elastic spatial sharing policy for its SIMD lanes. The SIMD lanes are dynamically partitioned across the co-running workloads based on their phase behaviors, so that each workload may execute in variable-length SIMD mode. This has been achieved by a hardware-software co-design. Our evaluation shows that Occamy can significantly improve SIMD utilization, and consequently, achieve performance speedups over the existing SIMD architectures, with only negligible chip area cost.

ACKNOWLEDGMENTS

We thank all the reviewers for their valuable comments and suggestions. This work was supported in part by the National Key Research and Development Program of China (2021ZD0110101), the National Natural Science Foundation of China (62232015, 62090024, 62132020, 62202452), and Australian Research Council grant (DP210102409).

REFERENCES

[1] Alon Amid, Krste Asanovic, Allen Baum, Alex Bradbury, Tony Brewer, Chris Celio, Aliaksei Chapyzhenka, Silviu Chiricescu, Ken Dockser, Bob Dreyer, et al. 2020. RISC-V "V" Vector Extension, version 0.9.

[2] Philip Bedoukian, Neil Adit, Edwin Peguero, and Adrian Sampson. 2021. Software-Defined Vector Processing on Manycore Fabrics. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture.* 392–406.

[3] Spiridon F Beldianu and Sotirios G Ziavras. 2013. Multicore-based vector coprocessor sharing for performance and energy gains. *ACM Transactions on Embedded Computing Systems (TECS)* 13, 2 (2013), 1–25.

[4] Spiridon F Beldianu and Sotirios G Ziavras. 2014. Performance-energy optimizations for shared vector accelerators in multicores. *IEEE Trans. Comput.* 64, 3 (2014), 805–817.

[5] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH computer architecture news* 39, 2 (2011), 1–7.

[6] Mark Byler, Michael Wolfe, James Davies, Christopher Huson, and Bruce Leasure. 1987. Multiple Version Loops. In *ICPP.*

[7] Yishen Chen, Charith Mendis, and Saman Amarasinghe. 2022. All You Need is Superword-Level Parallelism: Systematic Control-Flow Vectorization with SLP. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022).* Association for Computing Machinery, New York, NY, USA, 301–315. https://doi.org/10.1145/3519939.3523701

[8] Yishen Chen, Charith Mendis, Michael Carbin, and Saman Amarasinghe. 2021. VeGen: A Vectorizer Generator for SIMD and Beyond. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS 2021).* Association for Computing Machinery, New York, NY, USA, 902–914. https://doi.org/10.1145/3445814.3446692

[9] Jack Choquette, Wishwesh Gandhi, Olivier Giroux, Nick Stam, and Ronny Krashinsky. 2021. NVIDIA A100 tensor core GPU: Performance and innovation. *IEEE Micro* 41, 2 (2021), 29–35.

[10] Standard Performance Evaluation Corporation. 2017. SPEC CPU® 2017. https://www.spec.org/cpu2017/.

[11] Ganesh Dasika, Mark Woh, Sangwon Seo, Nathan Clark, Trevor Mudge, and Scott Mahlke. 2010. Mighty-Morphing Power-SIMD. In *Proceedings of the 2010 International Conference on Compilers, Architectures and Synthesis for Embedded Systems* (Scottsdale, Arizona, USA) *(CASES '10).* Association for Computing Machinery, New York, NY, USA, 67–76. https://doi.org/10.1145/1878921.1878934

[12] Joao Mario Domingos, Nuno Neves, Nuno Roma, and Pedro Tomás. 2021. Unlimited vector extension with data streaming support. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA).* IEEE, 209–222.

[13] Graham Gobieski, Amolak Nagi, Nathan Serafin, Mehmet Meric Isgenc, Nathan Beckmann, and Brandon Lucia. 2019. Manic: A vector-dataflow architecture for ultra-low-power embedded systems. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture.* 670–684.

[14] Venkatraman Govindaraju, Tony Nowatzki, and Karthikeyan Sankaralingam. 2013. Breaking SIMD shackles with an exposed flexible microarchitecture and the access execute PDG. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques.* 341–351. https://doi.org/10.1109/PACT.2013.6618830

[15] Sajith Kalathingal, Caroline Collange, Bharath N. Swamy, and André Seznec. 2016. Dynamic Inter-Thread Vectorization Architecture: Extracting DLP from TLP. In *2016 28th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD).* 18–25. https://doi.org/10.1109/SBAC-PAD.2016.11

[16] Sajith Kalathingal, Sylvain Collange, Bharath N. Swamy, and André Seznec. 2018. DITVA: Dynamic Inter-Thread Vectorization Architecture. *J. Parallel and Distrib. Comput.* 120 (2018), 267–281. https://doi.org/10.1016/j.jpdc.2017.11.006

[17] Aditya Kesiraju, Andrew J. Beaumont-Smith, Deepankar Duggal, and Ran A. Chachick. 2020. Coprocessor with Distributed Register.

[18] Aditya Kesiraju, Andrew J. Beaumont-Smith, Boris S. Alvarez-Heredia, Pradeep Kanapathipillai, Ran A. Chachick, and Srikanth Balasubramanian. 2020. Coprocessors with Bypass Optimization, Variable Grid Architecture, and Fused Vector Operations.

[19] Aditya Kesiraju, Brett S. Feero, Nikhil Gupta, and Viney Gautam. 2020. Coprocessor Operation Bundling.

[20] 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.* IEEE, 685–696.

[21] Tuomas Koskela, Zakhar Matveev, Charlene Yang, Adetokunbo Adedoyin, Roman Belenov, Philippe Thierry, Zhengji Zhao, Rahulkumar Gayatri, Hongzhang Shan, Leonid Oliker, et al. 2018. A novel multi-level integrated roofline model approach for performance characterization. In *International Conference on High Performance Computing.* Springer, 226–245.

[22] Ronny Krashinsky, Christopher Batten, Mark Hampton, Steve Gerding, Brian Pharris, Jared Casper, and Krste Asanovic. 2004. The vector-thread architecture. In *Proceedings. 31st Annual International Symposium on Computer Architecture, 2004.* IEEE, 52–63.

[23] Samuel Larsen and Saman Amarasinghe. 2000. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (Vancouver, British Columbia, Canada) *(PLDI '00).* Association for Computing Machinery, New York, NY, USA, 145–156. https://doi.org/10.1145/349299.349320

[24] Yunsup Lee, Rimas Avizienis, Alex Bishara, Richard Xia, Derek Lockhart, Christopher Batten, and Krste Asanović. 2011. Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators. In *Proceedings of the 38th annual international symposium on Computer architecture.* 129–140.

[25] Yunsup Lee, Rimas Avizienis, Alex Bishara, Richard Xia, Derek Lockhart, Christopher Batten, and Krste Asanović. 2013. Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators. *ACM Transactions on Computer Systems (TOCS)* 31, 3 (2013), 1–38.

[26] Bruce Paul Leung. 1990. *Issues on the design of parallelizing compilers.* Master's thesis. Citeseer.

[27] Arm Limited. 2021. Introduction to SVE2. https://developer.arm.com/documentation/102340/0001/Introducing-SVE2.

[28] Oscar G Lorenzo, Tomás F Pena, José C Cabaleiro, Juan C Pichel, and Francisco F Rivera. 2014. Using an extended Roofline Model to understand data and thread affinities on NUMA systems. *Annals of Multicore and GPU Programming* 1, 1 (2014), 37–48.

[29] Yaojie Lu, Seyedamin Rooholamin, and Sotirios G Ziavras. 2016. Vector coprocessor virtualization for simultaneous multithreading. *ACM Transactions on Embedded Computing Systems (TECS)* 15, 3 (2016), 1–25.

[30] Jiayuan Meng, Jeremy W. Sheaffer, and Kevin Skadron. 2012. Robust SIMD: Dynamically Adapted SIMD Width and Multi-Threading Depth. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium.* 107–118. https://doi.org/10.1109/IPDPS.2012.20

[31] Anant V Nori, Rahul Bera, Shankar Balachandran, Joydeep Rakshit, Om J Omer, Avishaii Abuhatzera, Belliappa Kuttanna, and Sreenivas Subramoney. 2021. REDUCT: Keep it Close, Keep it Cool!: Efficient Scaling of DNN Inference on Multi-core CPUs with Near-Cache Compute. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA).* IEEE, 167–180.

[32] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. 2008. GPU computing. *Proc. IEEE* 96, 5 (2008), 879–899.

[33] Yongjun Park, Jason Jong Kyu Park, Hyunchul Park, and Scott Mahlke. 2012. Libra: Tailoring simd execution using heterogeneous hardware and dynamic configurability. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture.* IEEE, 84–95.

[34] Yongjun Park, Sangwon Seo, Hyunchul Park, Hyoun Kyu Cho, and Scott Mahlke. 2012. SIMD defragmenter: Efficient ILP realization on data-parallel architectures. *ACM SIGPLAN Notices* 47, 4 (2012), 363–374.

[35] Suzanne Rivoire, Rebecca Schultz, Tomofumi Okuda, and Christos Kozyrakis. 2006. Vector lane threading. In *2006 International Conference on Parallel Processing (ICPP'06).* IEEE, 55–64.

[36] Nigel Stephens. 2016. ARMv8-A next-generation vector architecture for HPC. In *2016 IEEE Hot Chips 28 Symposium (HCS).* 1–31. https://doi.org/10.1109/HOTCHIPS.2016.7936203

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

[38] Inc Synopsys. [n. d.]. Synopsys Design Compiler. https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html.

[39] Alexa VanHattum, Rachit Nigam, Vincent T. Lee, James Bornholt, and Adrian Sampson. 2021. Vectorization for Digital Signal Processors via Equality Saturation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS 2021).* Association for Computing Machinery, New York, NY, USA, 874–886. https://doi.org/10.1145/3445814.3446707

[40] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (2009), 65–76.

[41] Samuel Webb Williams. 2008. *Auto-tuning performance on multicore computers*. University of California, Berkeley.

[42] Jing Xia, Chuanning Cheng, Xiping Zhou, Yuxing Hu, and Peter Chun. 2021. Kunpeng 920: The First 7-nm Chiplet-Based 64-Core ARM SoC for Cloud Services. *IEEE Micro* 41, 5 (2021), 67–75. https://doi.org/10.1109/MM.2021.3085578

[43] Yohei Yamada and Shintaro Momose. 2018. Vector engine processor of NEC's brand-new supercomputer SX-Aurora TSUBASA. In *Proceedings of A Symposium on High Performance Chips, Hot Chips*, Vol. 30. 19–21.

[44] Charlene Yang, Yunsong Wang, Thorsten Kurth, Steven Farrell, and Samuel Williams. 2021. Hierarchical roofline performance analysis for deep learning applications. In *Intelligent Computing*. Springer, 473–491.

[45] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. 2019. MArk: Exploiting Cloud Services for Cost-Effective,SLO-Aware Machine Learning Inference Serving. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 1049–1062.

[46] Minjia Zhang, Samyam Rajbhandari, Wenhan Wang, and Yuxiong He. 2018. DeepCPU: Serving RNN-based Deep Learning Models 10x Faster. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 951–965.

[47] Hao Zhou and Jingling Xue. 2016. A Compiler Approach for Exploiting Partial SIMD Parallelism. *ACM Trans. Archit. Code Optim.* 13, 1, Article 11 (mar 2016), 26 pages. https://doi.org/10.1145/2886101

[48] Hao Zhou and Jingling Xue. 2016. Exploiting Mixed SIMD Parallelism by Reducing Data Reorganization Overhead. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization* (Barcelona, Spain) *(CGO '16)*. Association for Computing Machinery, New York, NY, USA, 59–69. https://doi.org/10.1145/2854038.2854054

Received 2022-10-20; accepted 2023-01-19