Vector-Processing for Mobile Devices: Benchmark and Analysis

Alireza Khadem[†] *arkhadem@umich.edu*
Daichi Fujiki[‡] *dfujiki@keio.jp*
Nishil Talati[†] *talatin@umich.edu*
Scott Mahlke[†§] *mahlke@umich.edu*
Reetuparna Das[†] *reetudas@umich.edu*

[†]*University of Michigan,* [‡]*Keio University,* [§]*Nvidia Research*

2023 IEEE International Symposium on Workload Characterization (IISWC) | 979-8-3503-0317-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/IISWC59245.2023.00036

Abstract

Vector processing has become commonplace in today's CPU microarchitectures. Vector instructions improve performance and energy which is crucial for resource-constraint mobile devices. The research community currently lacks a comprehensive benchmark suite to study the benefits of vector processing for mobile devices. This paper presents Swan—an extensive vector processing benchmark suite for mobile applications. Swan consists of a diverse set of data-parallel workloads from four commonly used mobile applications: operating system, web browser, audio/video messaging application, and PDF rendering engine. Using Swan benchmark suite, we conduct a detailed analysis of the performance, power, and energy consumption of vectorized workloads, and show that: (a) Vectorized kernels increase the pressure on cache hierarchy due to the higher rate of memory requests. (b) Vector processing is more beneficial for workloads with lower precision operations and higher cache hit rates. (c) Limited Instruction-Level Parallelism and strided memory accesses to multi-dimensional data structures prevent vector processing benefits from scaling with more SIMD functional units and wider registers. (d) Despite lower computation throughput than domain-specific accelerators, such as GPU, vector processing outperforms these accelerators for kernels with lower operation counts. Finally, we show five common computation patterns in mobile data-parallel workloads that dominate the execution time.

1. Introduction

Vector processing was first introduced in supercomputers [38, 42, 45] and has become commonplace in today's CPU microarchitectures. Both server-grade and resource-limited mobile CPUs integrate vector pipelines. Vector Instruction Set Architecture (ISA) extensions efficiently encode multiple operations within a single instruction and enjoy fine-grain scalar-vector instruction interleaving [21]. The performance and energy improvement of vector processing is even more significant to the resource-limited mobile processors. This work aims to study the strengths and limitations of vector processing for mobile devices and provides a comprehensive benchmark suite for further research in this area.

Despite the availability of several domain-specific and general-purpose mobile benchmark suits [14, 16, 17, 27, 33, 34, 40, 47], a comprehensive benchmark suite for evaluating vector processing in mobile devices is lacking. This is because prior vector benchmark suites include Chip Multiprocessor (CMP) or GPU applications for vector processing evaluation, where the control-flow divergence limit the power of vector processing. These applications are offloaded to domain-specific accelerators of the mobile SoCs. In addition, these benchmark suites contain desktop and server applications with massive parallelism that are offloaded to the cloud-computing in mobile applications. Therefore, providing a benchmark suite for mobile applications significantly eases the state-of-the-art research in this area.

This paper introduces **Swan**—a benchmark suite for studying vector processing in mobile devices. Swan encompasses vectorized workloads for resource-constraint mobile processing platforms from multimedia processing, graphics, data compression, cryptography, and string utility domains. We assemble a diverse set of optimized mobile applications and conduct an in-depth performance and energy analysis study. Swan benchmark suite is maintained online at https://github.com/arkhadem/Swan.

To create Swan, we extensively study the source code of four widely-used mobile applications: Chromium [5] (web browsing), Android [1] (operating system), WebRTC [10] (audio/video messaging service), and PDFium [7] (PDF rendering engine). We carefully choose 12 libraries that are commonly used among these applications to cover a large set of workloads representative of mobile applications. We select a rich set of 59 data-parallel kernels, each of which executes an independent task. We study the vector instruction set and instruction reduction diversity of these workloads.

Using the Swan benchmark suite, we conduct a detailed study of vector processing for mobile applications, specifically Arm Neon architecture, in terms of performance and energy/power consumption. We show that vector processing is more beneficial for workloads with lower operation precision and higher cache hit rate. In addition, we illustrate the limited performance improvement of compiler auto-vectorization for the scalar implementation since the scalar code is optimized for superscalar execution. Our analysis shows substantial energy savings of vector processing due to significantly lower execution time. On the other hand, vector processing increases chip power consumption in workloads with large working set sizes due to the higher main memory access rates. Studying three different core microarchitectures of a big.LITTLE Arm architecture proves that more vector pipelines are not always beneficial due to the limited vector Instruction Level Parallelism (ILP).

We introduce five common computation patterns of vector implementations: reduction, random memory access, strided memory access, matrix transposition, and vector APIs. We show the importance of these computation patterns in terms

2835-2238/23/$31.00 ©2023 IEEE
DOI 10.1109/IISWC59245.2023.00036

Authorized licensed use limited to: University of Michigan Library. Downloaded on December 20,2024 at 14:35:50 UTC from IEEE Xplore. Restrictions apply.

TABLE 1: PRIOR GENERAL-PURPOSE AND DOMAIN-SPECIFIC BENCHMARK SUITES FOR MOBILE PLATFORMS.

Benchmark Suite	Microarchitecture	Implementation	Application Domain	Metrics	Diversity Analysis	Open Source
EEMBC CoreMark [22]	Single-core	Scalar	Mobile	Perf./Pow.	✗	✓
Geekbench [30]	Single-core	Scalar	Desktop/Server/Mobile	Perf	✓	✗
EEMBC MultiBench [6]	CMP	UNK	Mobile	Perf.	✗	✗
ParVec [14]	Vector (CMP)*	Pthread/AVX/Neon	Desktop/Server	Perf./Pow.	✓	✓
RiVEC [40]	Vector (CMP/GPU)*	RVV	Desktop/Server	Perf.	✓	✓
VectorBench [34]	Vector	AVX/Neon	Desktop/Server	Perf.	✗	✓
VComputeBench [33]	Vector (GPU)*	Vulkan	Mobile	Perf.	✓	✓
MEVBench [17]+	CMP	Pthread/OpenCV	Mobile	Perf.	✗	✓
MLPerf [27]+	CMP/GPU/DSP	VendorSDK/TFLite	Mobile	Perf.	✗	✓
ARBench [16]+	CMP/GPU/DSP	ARCore/OpenGL	Mobile	Perf.	✗	✓
WiBench [47]+	Vector	SSE	Desktop/Mobile	Perf.	✗	✓
Swan (this work)	Vector	Neon	Mobile	Perf./Pow.	✓	✓

+ Domain-Specific Benchmark. *Benchmark redevelops workloads of other benchmark suites (features of the source benchmark).

of the number of workloads exhibiting them, and the average fraction of workload instructions they consume.

Arm Neon limits vector register width to 128 bits and our evaluated mobile SoC contains only two vector execution pipelines. To further study the scalability of vector processing with wider registers and more vector pipelines, we implement a fake Arm Neon library (available on GitHub) and re-develop eight representative workloads with wider vector registers up to 1024 bits using an in-house cycle-accurate simulator. We show that wider vector registers provide substantial speedup for workloads with streaming memory access patterns. When Data-Level Parallelism (DLP) is not evenly divisible by the register width or strided memory accesses are needed to access a multidimensional data structure, vector engine utilization decreases and workloads hardly enjoy wider implementations. Besides, high register pressure limits Instruction Level Parallelism (ILP), which in turn drops the utilization of vector pipelines and prevents performance from scaling with more vector execution pipelines.

Finally, we show the limitations of domain-specific accelerators (GPU and DSP) in terms of data transfer and kernel launch overheads. We compare GPU and vector processing performance for various problem sizes, showing that GPU, despite having higher throughput, only outperforms Neon in workloads with high operation counts.

Swan is the **first** benchmark suite for mobile vector processing. Key contributions of this work are: (a) Swan, an open-source and true vector processing benchmark suite with 59 diverse data-parallel kernels of four commonly-used mobile applications. (b) Performance, power, and energy analysis of mobile vector processing benefits and compiler vectorization limitations. (c) Performance scalability study of vector processing with wider instructions and more vector execution pipelines. (d) Evaluating domain-specific acceleration bottlenecks for fine-grain data-parallel workloads.

2. Background and Motivation

2.1. Vector-Processing

Vector Processing was initially employed in supercomputers [38, 42, 45] to accelerate workloads with massive parallelism [21]. Vector Instruction Set Architecture (ISA) extension reduces the number of instructions by encoding multiple operations, improving performance and energy consumption. Due to the instruction efficiency, general-purpose microprocessors employed Vector ISA extensions for multimedia processing workloads with high DLP [19, 23, 29]. Today, various ISAs provide vector extensions that exploit fine-grain DLP in a broad set of applications.

Vector processing plays a crucial role in mobile devices with limited resources due to performance and energy efficiency. Arm first employed vector processing by Vector Floating-point (VFP) Coprocessor [11]. State-of-the-art Arm architectures integrate vector execution pipelines into mobile processors with Advanced-SIMD (ASIMD) functional units. While modern mobile SoCs incorporate various domain-specific accelerators, vector processing enjoys tight integration with the scalar pipeline, which is especially important for kernels with *fine-grain scalar-vector instruction interleaving*.

2.2. Benchmark Suites

Benchmarks ease the evaluation of instruction sets, compiler optimizations, and architecture bottlenecks. Table 1 shows various general-purpose and domain-specific benchmark suites for mobile processors. We observe that the research community lacks a benchmark suite containing different state-of-the-art vector processing applications for mobile devices. In this section, we describe the design goals of Swan benchmark suite and discuss the opportunities for improvement in prior works within each aspect.

Target Microarchitecture and Implementation. Different benchmark suites such as Embedded Microprocessor Benchmark Consortium (EEMBC) MultiBench [6] and Geekbench [30] focus on the Single-Core scalar performance of mobile processors. Due to the lack of enough parallelism, it is reported that these benchmark suites face difficulty exercising vector engines [30]. EEMBC MultiBench [6] contains Chip Multiprocessor (CMP) workloads. In addition, various benchmark suites are available for GPUs [8, 15, 44]. However, Task-Level Parallelism (TLP) of CMP and GPU workloads is not suitable for vector processing.

Application Domain. Desktop and Server workloads encompass extensive problem sizes and exhibit greater DLP levels than mobile applications. For example, RiVEC [40] and ParVec [14] contain Financial Analysis and Data Mining workloads. Mobile applications offload these workloads to cloud computing due to the huge problem size. In addition, various applications that mobile vector processors executed

Authorized licensed use limited to: University of Michigan Library. Downloaded on December 20,2024 at 14:35:50 UTC from IEEE Xplore. Restrictions apply.

in the past are now offloaded to domain-specific accelerators. For example, VectorBench [34] and ParVec [14] provide video processing workloads that are efficiently offloaded to mobile GPUs. MEVBench [17], MLPerf [27], ARBench [16], and WiBench [47] are domain-specific benchmark suites that contain a single workload, such as Computer Vision, Machine Learning, Augmented/Virtual Reality, and Wireless Signal Processing workloads, respectively.

Metrics, Diversity Analysis, and Open Source. Prior vector benchmark suites [34, 40] lack Power and Energy analysis that is crucial to resource-limited mobile devices. Instruction diversity is yet another important factor for benchmark design to exercise different aspects of the design.

Despite prior works, Swan encompasses *vector* workloads from a wide set of *mobile* applications. Section 6 shows that Swan benchmarks contain different computation patterns and Section 8 studies the inadequacy of domain-specific accelerators for these benchmarks. In Section 5, we study mobile platform-specific performance metrics, auto-vectorization challenges, and instruction diversity of the benchmarks. We release Swan as an open-source benchmark suite on GitHub to ease the study of vector ISA, compiler, and architecture.

3. The Swan Benchmark Suite

3.1. Scope of Benchmark

To create Swan benchmark suite, we target real-world **mobile applications**, analyze their common libraries, and select time-consuming data-parallel kernels. Table 2 shows these libraries and their usage across four mobile applications: (a) *Chromium Project* [5] encompass Chromium and Chromium OS, which contain the source code of Google Chrome browser and Google Chrome OS. (b) *Android Project* [1] contains the source code and scripts for Android operating system. (c) *WebRTC Project* [10] provides Real-Time (voice/text/video) Communication (RTC) APIs for many messaging and audio/video conferencing platforms such as Zoom, Microsoft Teams, Slack, or Google Meet. And (d) *PDFium* [7] is a PDF rendering engine in Google Chrome and Microsoft Edge browsers.

Our code analysis shows 12 common **libraries** between these applications that contain data-parallel kernels and Arm Neon implementations. We profile Chrome execution with GPU acceleration while browsing top visited websites [25] and choose nine libraries that are not offloaded to GPU due to the offloading overheads (Section 8). Table 2 shows the maximum and average execution time of Chrome consumed by these libraries. Swan also encompasses three traditional vector processing application domains, *i.e.*, audio and video codecs and Machine Learning for low-end mobile products that lack GPU or DSP due to area and power constraints. Note that Swan is a benchmark suite for CPU vector processing exploration and does not provide GPU and DSP applications.

We carefully separate and benchmark 59 **data-parallel kernels** from libraries, each of which performs an independent task or algorithm. We prioritize algorithms that are not specific to a library. For example, **`libvpx`** contains many data-parallel kernels for VP8 and VP9 video codecs, but we choose a subset of four kernels that are commonly used in the coding process of different video codecs.

TABLE 2: ACCELERATED LIBRARIES: DOMAIN, USAGE, AND CHROMIUM EXE. TIME.

Library	Domain	Symbol	Chromium	Android	WebRTC	PDFium	Max. (%)	Avg. (%)
`libjpeg-turbo`	Image Processing	`LJ`	✓	✗	✗	✓	6.8	2.4
`libpng`	Image Processing	`LP`	✓	✗	✗	✓	0.8	0.3
`libwebp`	Image Processing	`LW`	✓	✗	✗	✓	7.3	1.7
`Skia`	Graphics	`SK`	✓	✓	✗	✓	8.5	4.6
`WebAudio`	Audio Processing	`WA`	✓	✗	✓	✗	16.3	2.5
`PFFFT`	Audio Processing	`PF`	✓	✓	✓	✗	5.6	1.3
`zlib`	Data Compression	`ZL`	✓	✓	✗	✓	0.4	0.2
`boringssl`	Cryptography	`BS`	✓	✓	✓	✗	0.9	0.6
`Opt. Routines`	String Utilities	`OR`	✓	✓	✓	✓	9.6	1.2
`libopus`	Audio Processing	`LO`	✓	✓	✓	✗	-	-
`libvpx`	Video Processing	`LV`	✓	✓	✓	✗	-	-
`XNNPACK`	Machine Learning	`XP`	✓	✓	✗	✗	-	-

3.2. Workloads

Table 2 shows the 12 libraries of Swan benchmark suite.

Image processing is a primary application for vector processing. **`libjpeg-turbo`**, **`libpng`**, and **`libwebp`** are all image processing libraries used in Chromium and PDFium projects for JPEG, PNG, and WEBP image codecs. **`Skia`** is a graphics library that provides rasterization (*i.e.*, converting paint operations to pixel bitmap) APIs as a graphics engine for other mobile applications. These libraries provide fine-grain APIs that process a block, row, channel, or multiple channels of the image. These APIs: (1) modify the color space (Gray, RGBA, and YCbCR), color code (PNG's true and indexed color), or size (down/upsample or vertical/horizontal convolution) of the image, (2) apply prediction filters (WEBP's DC, TrueMotion, Vertical, and Horizontal predictions) for image (de)compression, or (3) enhance image quality (**`libwebp`**'s Sharp YUV filters). Due to the fine-grain interleaving of scalar and vector APIs, domain-specific acceleration of these image processing libraries imposes a non-negligible cost of kernel launch overhead. In addition, while **`Skia`** exploits GPU for accelerated rasterization, different operations that convert data to GPU-compatible format before the GPU rasterization exploit vector processing locally on the CPU.

Web Audio API provides audio synthesizing for Web Applications. Web Audio applications create a graph of audio processing nodes that define the overall rendering behavior. **`Webaudio`** modules of Chromium and WebRTC similarly operate on an audio frame (2 channels of 128 audio samples) to: (1) implement an audio node's functionality (GainNode that changes audio volume), (2) transfer audio samples between audio node buffers, or (3) merge multiple audio inputs. **`PFFFT`** is a Pretty Fast FFT implementation that provides frequency-domain analysis APIs for **`Webaudio`**. Due to the latency constraints of Web Audio APIs, these libraries are not suited for domain-specific acceleration.

`zlib` provides compression APIs for different libraries and applications, including **`libpng`**. **`zlib`** uses two checksum algorithms (Adler-32 and CRC-32) to detect data corruption,

Authorized licensed use limited to: University of Michigan Library. Downloaded on December 20,2024 at 14:35:50 UTC from IEEE Xplore. Restrictions apply.

which takes considerable execution time. While both stages of compression (LZ77 and Huffman de/encode) are scalar, **zlib** uses vector processing for checksum calculations.

boringssl is Google's fork of OpenSSL, an SSL library that provides both low-level cryptography primitives (AES, DES, ChaCha20, and SHA256) and SSL implementation for secure network communications. **Arm Optimized Routines** contains an optimized version of math, network, and string utilities for Arm ISA-based processors. We target string routines (memcmp, memchr, memcpy, and strlen) as many mobile applications extensively use them.

Due to the fine-grain **zlib**, **boringssl**, and **Arm Optimized Routines** utility APIs that are extensively interleaved with applications' scalar code, domain-specific accelerations are not suited.

Opus is an audio codec for interactive audio applications. **libopus** is an Opus coder whose kernels operate on an audio frame. We benchmark multiple kernels that: (a) apply audio filters (Autoregressive Moving Average (ARMA) and Linear Predictive Coding (LPC) filters), and (b) calculate autocorrelation (pitch and frequency autocorrelations).

libvpx is the reference implementation of VP8 and VP9 video codecs, widely used by content providers like YouTube and Netflix. We benchmark kernels that are common among most video codecs. These kernels operate on an block of pixels to: (a) calculate frequency transforms (forward and inverse Discrete Cosine Transform), (b) compute block similarities (*e.g.*, Sum of Absolute Difference), and (c) quantize pixels for a higher compression rate.

XNNPACK provides optimized Neural Network primitives that are employed in the back-end of Machine Learning frameworks such as TensorFlow Lite and PyTorch. We evaluate four precisions for General Matrix Multiply (GEMM) and Sparse-Dense Matrix Multiplication (SpMM) kernels. Various neural network APIs (such as convolutional and fully-connected layers) use these two kernels.

4. Methodology and Tools

4.1. Workloads and Input Data Size

Swan benchmark suite generates random inputs for workloads with the following sizes: (a) 720x1280 (HD) images for image processing, graphics, and video processing libraries, (b) 1 second of a standard audio stream with a 44.1 kHz sample rate for audio processing libraries, (c) 128 KB data for data compression, cryptography, and string utility libraries, and (d) 156 layers of Convolutional Neural Networks for the machine learning library. When input data values affect the control flow of the workload, we carefully set the configuration and generate inputs based on the source library configurations and testing infrastructures. To amortize the measurement errors, we repeat the measurement for multiple *iterations* until the execution time reaches 1 second. Finally, we ensure the correctness of Swan workloads by comparing the Scalar and Arm Neon implementation outputs.

TABLE 3: QUALCOMM SNAPDRAGON 855 - CORTEX-A76 PRIME CORE BASELINE

Configuration	Detail
Scalar core	2.8GHz, 128 entry ROB, out-of-order 4-way Decode, 8-way Issue, 4-way Commit
Vector engine	2 128-bit Advance SIMD units + crypto and FP16 ext
L1-I cache	64 KiB, 4-way, 4 cycle latency
L1-D cache	64 KiB, 4-way, 4 cycle latency
L2 cache	512 KiB, 8-way, Private, Inclusive, 9 cycle latency
LLC	2 MiB, 8-way, Shared, Inclusive, 31 cycle latency

4.2. Evaluation Environment

We analyze Swan workloads on Snapdragon 855 SoC, which enjoys Arm Big.LITTLE architecture with three different CPU configurations: 1 Prime and 3 Gold high-performance Cortex-A76 cores, and 4 Silver efficient Cortex-A55 cores. We pin the Swan benchmark process in all experiments to the high-performance Prime core. In Section 5.5, we show the sensitivity of vector processing performance to the core microarchitecture. To focus on vector processing performance, we study single-thread implementation to minimize the error due to the multi-threading overhead.

4.3. Measurement and Simulation Tools

We employ *Android NDK r23c* to cross-compile Swan with *Android Clang 12.0.9*. We choose level 3 optimization for all workloads and compile the scalar code with auto-vectorization (*Auto* implementation) to study its benefits for our set of data-parallel kernels. We disable auto-vectorization for the scalar code in the rest of the evaluation (*Scalar* implementation). In addition, Clang replaces pieces of code with certain standard C library functions (such as *memset*) that use Arm Neon acceleration. Therefore, we also disable optimizations of standard C library functions for the *Scalar* implementation. With these compilation flags, we ensure that the *Scalar* implementation does not employ any vector operations. We also compiled the explicitly-vectorized Neon code with auto-vectorization, yet, our evaluation showed negligible auto-vectorization improvement. Therefore, we only study the Neon code without auto-vectorization (*Neon* implementation). In addition, we minimize memory stalls to focus on the vector processing benefits by warming up caches before each *iteration*.

We measure average battery current and voltage while executing each benchmark to calculate the chip's power consumption, including main memory. Energy consumption is calculated as a product of power consumption and execution time. We use *Simpleperf* [20] to profile Performance Monitor Unit (PMU) events [2] of the device for all workloads to study microarchitectural bottlenecks in Section 5.4.

To study the scalability of vector processing with wider registers, we implement eight kernels with fake wider Neon intrinsics. We develop a functional simulator for Neon intrinsics to ensure the correctness of the fake intrinsics. Next, we capture dynamic instruction traces using a *DynamoRIO* [24] client running on a server-class CPU with Armv8.2-A architecture (same as the mobile processor). Finally, we develop a trace-driven simulator based on Ramulator [28] CPU model and simulate instruction traces.

Authorized licensed use limited to: University of Michigan Library. Downloaded on December 20,2024 at 14:35:50 UTC from IEEE Xplore. Restrictions apply.

5. Performance Analysis

We analyze the instruction set, performance, and energy improvement of vectorized Swan workloads. Due to the limited of space, we show the Geometric mean of performance and energy improvements for kernels of each library.

5.1. Instruction Diversity Analysis

Figure 1 (primary Y-axis) shows the instruction distribution of the Arm Neon implementation based on Arm Software Optimization Guide [4]. *S-Integer* + *S-Float* shows the scalar part of the kernels. **PF** requires significant scalar computation for FFT calculation. Hence, this library contains the highest portion of scalar instructions. **LT** linearly operates on an image row that requires the smallest portion of scalar instructions for control flow and address calculation. A higher fraction of *Vector Load and Store* instructions shows memory-intensive libraries with lower compute density. **WA** takes advantage of *Vector APIs* (Section 6.5) that keeps vector elements in memory, requiring a load and a store for each vector operation. Swan contains libraries with different data types (integer and floating-point). Audio processing and machine learning libraries (**WA**, **PF**, **LO**, and **XP**) contain floating-point operations. **XP** contains FP16 implementation of GEMM and SpMM. Swan also provides a set of libraries that exploit cryptography acceleration of Arm Neon architecture, *i.e.*, **ZL** and **BS**. Finally, Vector Miscellaneous instructions are used for vector manipulation, *i.e.*, converting vector register width and vector element type and combining/extracting multiple vector registers. **LO** contains operations with different data types and extensively uses Vector Miscellaneous instructions to manipulate vector registers. In summary, *Swan workloads encompass various applications with diverse instruction sets.*

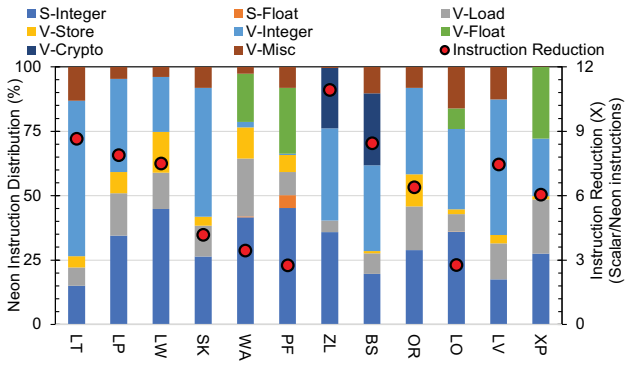

Figure 1: Scalar and Vector instruction breakdown (primary Y-axis) of Arm Neon kernels and total instruction reduction (secondary Y-axis) compared to the Scalar implementation.

Figure 1 (secondary Y-axis) compares the dynamic instruction count of the Scalar and Neon implementations. Assuming a similar code structure, the maximum instruction reduction of vector processing is bounded by the number of Vector Register Elements (*VRE*), calculated by Equation 1.

$$VRE = \frac{Vector\ Register\ Width\ (128\ bits)}{Element\ Data\ Width} \quad (1)$$

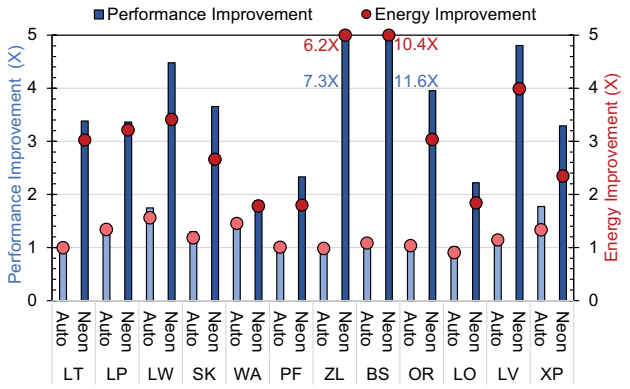

Figure 2: Performance (primary Y-axis) and Energy (secondary Y-axis) improvement of Auto-vectorization (Auto) and Explicit vectorization using Arm Neon intrinsics (Neon) compared to the Scalar implementation.

This equation shows that instruction reduction improves with lower precision data types. Image and video processing libraries (**LT**, **LP**, **LW**, and **LV**) process 8-bit pixel values, encoding more operations in a vector instruction. **ZL** and **BS** exploit cryptography instructions of Arm Neon architecture that reduces dynamic instructions substantially. Therefore, *Vector ISA encodes low-precision operations more efficiently compared to the scalar instruction set.*

5.2. Performance Analysis

Figure 2 (primary Y-axis) shows the performance of auto-vectorization (*Auto*) and vector implementations (*Neon*) normalized to the *Scalar* implementation. *Vector processing, theoretically, improves the performance of the data-parallel portion of the program by VRE.* To explain this claim, we break the instructions of the data-parallel portion of the kernels into three categories: (a) Address calculation and control flow scalar instructions: Data-parallel kernels only need to calculate the base address of each vector memory access, which reduces the address calculations by *VRE*. Moreover, loop trip counts and their required control-flow instructions drop by a factor of *VRE*. (b) Efficient vector loads and stores: A 128-bit vector load/store accesses *VRE* data elements from the memory hierarchy. The scalar kernel requires *VRE*× more cache accesses to load/store the same amount of data. Assuming a high cache hit rate for data-parallel applications, vector loads and stores take *VRE*× fewer cycles. (c) Vector compute instructions encode *VRE* operations. Assuming the same number of scalar and vector functional units with the same compute throughput in the microarchitecture, vector compute instructions provide a speedup factor of *VRE*.

Neon Performance Improvement. **ZL** and **BS** provide higher speedup because these libraries exploit the cryptography acceleration of Arm Neon architecture. Except these libraries, Neon speedup is between 1.9× (**WA**) to 4.8× (**LV**).

According to Amdahl's law, Neon speedup decreases in kernels with a more significant portion of scalar code.

Authorized licensed use limited to: University of Michigan Library. Downloaded on December 20,2024 at 14:35:50 UTC from IEEE Xplore. Restrictions apply.

LO requires pre-processing input and post-processing output data of the data-parallel portion of the code. **PF** frequently calculates the address and value of the FFT coefficients for the data-parallel computation. The Neon implementations of **LO** and **PF** libraries achieve low dynamic instruction reduction (both 2.7×) and speedup (2.2× and 2.3×).

Our discussion showcases that total speedup correlates with *VRE*. **XP** provides a speedup of 3.3×. While the theoretical maximum speedup of 32-bit and 16-bit kernels is 4× and 8×, our evaluation shows that the performance improvement of FP32/INT32 and FP16/INT16 kernels are in the range of [2.0× to 3.0×] and [3.5× to 5.1×], respectively. In addition, audio processing libraries (**WA**, **PF**, and **LO**) contain FP32 operations that drops *Neon* speedup to 1.8×, 2.3×, and 2.2×, respectively. *Vector instructions encode low-precision operations more efficiently, increasing the speedup.*

Image processing libraries (*e.g.*, **LT** and **LP**) require large working set sizes. A large working set drops L1D, L2D, and LLC cache hit rates to 91%, 90%, and 67% for the **LT** library. Therefore, memory accesses take more cycles and vector load latency is close to scalar load latency. In this case, the speedup of vector memory accesses is lower than *VRE*. Hence, although *LT* and *LP* contain low precision operations on 8-bit pixel data and provide high *VRE*, the achieved speedup of the *Neon* implementation of both libraries is 3.3×. *Lower cache hit-rate drops vector processing speedup in data-parallel kernels with large working set size.*

Auto-Vectorization Performance Improvement. It is believed that auto-vectorization requires less effort from programmers than explicit vectorization. However, our analysis shows that *auto-vectorizing the legacy scalar code that is optimized for higher super-scalar performance is not achieving sufficient speedups*. In fact, auto-vectorization requires complex loop transformations [32] to expose the DLP to the compiler. Table 4 shows that among the 59 analyzed data-parallel kernels, auto-vectorization enhances the performance of only 23 kernels. From these 23 kernels, *Auto* marginally outperforms 5 *Neon* implementations only because of higher loop interleaving count.

TABLE 4: *Auto* PERFORMANCE W.R.T *Scalar* AND *Neon*.

Auto vs. *Scalar*	#Kernels	*Auto* vs. *Neon*	#Boosted Kernels
Auto ≈ *Scalar*	34	*Auto* ≈ *Neon*	6
Auto < *Scalar*	2	*Auto* < *Neon*	12
Auto > *Scalar*	23	*Auto* > *Neon*	5

In general, our study shows two main reasons for auto-vectorization failure:

First, *compiler is unable to prove the legality of vectorization, i.e., the safety and correctness of vector transformations.* We provide three examples that fail the legality of vectorization and how *Neon* implementations solve them.

Example 1. Loops shall be countable, *i.e.*, compiler can calculate the number of loop iterations based on the variables. A `for` loop with a `break` statement or a `while` loop with an unknown condition is not countable. *Uncountable loops prevent vectorization in eight data-parallel kernels.* Neon uses `reduction` instructions to detect loop break conditions.

Example 2. Compilers use run-time checks if there is not enough information in compile-time to prove the safety of the vectorization. Code patterns that hinder run-time checks fail vectorization. For example, indirect memory accesses such as `A[B[i]]` prevent *memory aliasing* checks as calculating the boundary of accesses to array `A` requires evaluating all elements of array `B`. This code pattern is used to convert computations to look-up tables and optimize the scalar code for super-scalar performance. *Indirect memory accesses fail compiler vectorization in 8 data-parallel kernels.*

Example 3. Variables are often initialized before the loops and used and modified inside the loop. In this case, compilers add a `PHI` node that selects between the initial values (before the loop) and modified values (within the loop). Therefore, `PHI` nodes generate data dependencies between different iterations of the loop that the vectorizer must appropriately handle. LLVM can recognize and handle the common `PHI` nodes in vectorization. However, complex `PHI` nodes fail compiler vectorization because of data dependencies. For example, *Downsample* kernel of **LT** initializes bias values before the loop and used and modifies biases inside the loop. LLVM auto-vectorizer fails as it cannot safely resolve this data dependency. Neon implementation, however, uses predefined constant bias values. *Loop data dependencies prevent compiler vectorization in 9 kernels of Swan benchmark suite.*

Other legality obstacles, such as *reordering Floating-Point and Memory operations*, *inability to vectorize* `CALL` *instructions and* `switch` *statements*, and *unsafe memory operations* prevent compiler vectorization in 10 kernels.

Second, compiler employs a heuristic cost model that compares the benefits of vectorizing a loop with different Vectorization Factors (*VF*). Then, it chooses the *VF* with the minimum cost-to-width ratio. LLVM cost model, however, suffers from inaccuracies because different code characteristics (*e.g.,* loop trip counts and control flow behavior) and microarchitectural features (*e.g.,* throughput and latency of each instruction) are not known in the compile time. *Inaccurate cost model prevents compiler vectorization in 12 kernels of the Swan benchmark suite.*

5.3. Power and Energy Analysis

While vector processing substantially improves execution time, Figure 3 shows that it increases total chip power consumption, *including the main memory.* This is because vector processing increases DRAM access rate. Since memory accesses consume significant power, an increased DRAM access rate results in higher power consumption.

We measure the number of main memory accesses using LLC misses and the main memory access rate as the number of memory accesses per cycle. Our evaluation shows that the *Neon* memory access rate is 8.8× more than *Scalar* implementation. This is because of the higher LLC miss rate and lower cycles of the *Neon* implementation. Comparing the power consumption of libraries shows that kernels with higher LLC miss rate and memory access rate, such as the image processing and graphics libraries (**LT**, **LP**, **LW**, and **SK**), consume more chip power.

Authorized licensed use limited to: University of Michigan Library. Downloaded on December 20,2024 at 14:35:50 UTC from IEEE Xplore. Restrictions apply.

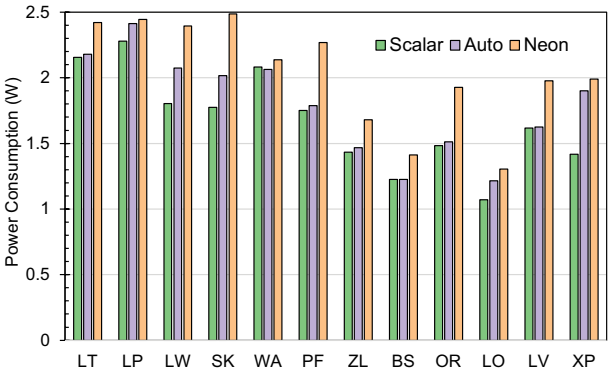

Figure 3: Total chip Power Consumption of libraries (including DRAM power).

Figure 2 (secondary Y-axis) shows the energy consumption of *Auto* and *Neon* normalized to the *Scalar*. Despite the higher power consumption of *Neon* implementations, significantly lower execution time and dynamic instruction count result in high energy savings, specifically in kernels with lower precision (higher *VRE*) and higher cache hit rate, such as **LW** (4.4×), **OR** (3.9×), and **LV** (4.8×).

5.4. Bottleneck Analysis

To study the bottlenecks of vector processing, we follow Intel's top-down microarchitecture analysis [46] by profiling microarchitectural characteristics of data-parallel kernels. Table 5 studies cache (L1D, L2, and LLC) Miss Per Kilo Instructions (MPKI), the portion of cycles stalled by Front-End and Back-End (%), and Instructions Per Cycle (IPC).

Data-parallel kernels enjoy regular control flow. Therefore, front-end stalls are less than 5% in the Neon implementations. In 4 data-parallel kernels, control divergence is handled by *If-Conversion*, where control dependency is converted to data dependency by executing both branches and choosing the final values of a vector using `AND/OR` or `BSL` (Bitwise Select) instructions. Low front-end stalls show that the *front-end modules of the microarchitecture, such as Instruction Cache and TLB, Branch Predictor, and Instruction Fetch and Decode are not a bottleneck for vector processing*.

IPC is lower in Neon implementations due to higher Back-End stalls of data-parallel kernels. We observe that two ASIMD functional units of the Prime core provide enough computation throughput. Therefore, *data-parallel applications with high back-end stalls are mainly bounded by memory stalls.* Vector processing increases the pressure on the cache hierarchy by frequent cache accesses. Hence, all cache levels experience higher MPKI. Scalar and Neon implementations of **XP** manually unroll the loops with a factor of 32 and 8. Thus, a burst of memory accesses is injected into the L1D cache, increasing L1D MPKI of *Scalar* and *Neon* to 4.9 and 20.1. Workloads with larger working set sizes, such as image processing and graphics libraries (**LJ**, **LP**, and **SK**), increase L2 and LLC MPKI.

TABLE 5: MICROARCHITECTURAL CHARACTERISTICS OF EVALUATED LIBRARIES.

Library	L1D MPKI		L2 MPKI		LLC MPKI		Front-End Stalls (%)		Back-End Stalls (%)		IPC	
	S	V	S	V	S	V	S	V	S	V	S	V
LJ	1.7	17.0	0.5	7.4	2.8	27.0	0.2	0.2	14.9	51.9	3.04	1.2
LP	1.2	8.1	0.3	6.5	2.8	17.9	0.2	0.4	11.3	47.2	2.9	1.4
LW	0.2	2.4	0.1	0.0	0.1	0.1	0.1	0.1	11.4	38.4	2.1	1.2
SK	1.2	4.8	0.1	0.4	2.3	9.4	0.1	0.1	10.4	25.2	2.6	2.3
WA	0.1	0.2	0.1	0.1	0.1	0.1	0.2	3.2	1.1	10.3	3.1	2.7
PF	3.1	9.7	1.2	3.3	0.1	0.1	0.4	0.2	22.3	28.2	2.9	2.4
ZL	0.8	5.8	0.1	0.1	0.1	0.1	0.1	0.4	12.1	25.9	3.3	2.1
BS	0.5	1.2	0.1	0.1	0.1	0.1	0.1	0.1	11.0	34.5	2.6	2.2
OR	1.2	6.2	0.1	0.1	0.1	0.1	0.2	0.2	4.3	27.3	3.3	2.2
LO	0.8	2.4	0.1	0.2	0.1	0.1	0.4	0.1	27.3	49.7	2.1	1.6
LV	1.4	7.8	0.3	3.3	1.9	6.5	0.2	0.1	15.9	41.9	2.8	1.6
XP	4.9	20.0	0.5	1.3	0.6	2.0	0.2	0.2	35.7	46.3	2.1	1.5

*Numbers are rounded up to 1 decimal place.
*S and V are Scalar and Vectorized Neon implementations.

5.5. Performance vs. Core Architecture

To further study the sensitivity of vector processing to core microarchitecture, Figure 4 shows the performance (Primary Y-axis) and energy (Secondary Y-axis) improvement of Neon normalized to the Scalar execution. Silver (Cortex-A55) core contains an In-Order pipeline with one 128-bit ASIMD functional unit at 1.8 GHz, while Gold and Prime (Cortex-A76) enjoy two ASIMD functional units of an out-of-order microarchitecture at 2.4 and 2.8 GHz, respectively.

Comparing Silver with Gold and Prime cores shows us that more ASIMD units do not substantially improve performance and energy consumption. In fact, more ASIMD units only benefit when there is enough Instruction-Level Parallelism (ILP). When a vector compute instruction is data dependent on a vector load/store, it is not issued to the ASIMD units until the memory access is finished and the vector load/store is executed. Thus, higher memory stalls of **LT**, **LP**, and **LO** libraries reduce ILP and Neon performance and energy improvement. **XP** implementations manually unroll the loops; therefore, this library increases the ILP, and Gold and Prime cores enjoy more ASIMD functional units and out-of-order processing.

Silver cores require less power consumption due to the in-order pipeline and lower DRAM access rate. Silver and Gold cores consume less power than the Prime core due to lower frequencies. Consequently, Neon achieves higher energy savings for Prime cores in nearly all workloads.

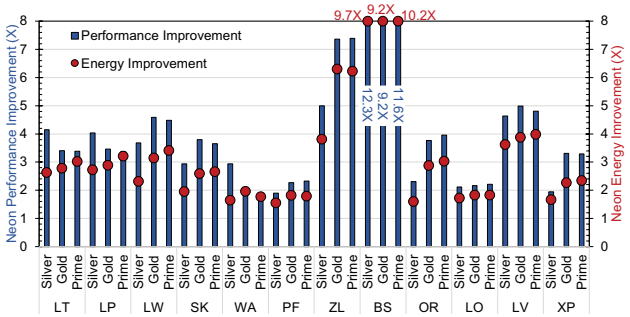

Figure 4: Performance analysis of core architectures.

Authorized licensed use limited to: University of Michigan Library. Downloaded on December 20,2024 at 14:35:50 UTC from IEEE Xplore. Restrictions apply.

6. Common Computation Patterns

In this section, we study common code patterns of the evaluated libraries and discuss their bottlenecks.

6.1. Reduction

Reduction is a computation pattern that reduces vector elements into a single result. Swan benchmark suite contains seven data-parallel kernels with reduction operations. These kernels exploit parallelism for vectorization in two ways:

(1) **Inter-Reduction Parallelism:** *Neon* exploits the parallelism *between* multiple reduction operations. Each vector instruction performs 1 step of reduction for *VRE* output results. For example, *Vertical/Horizontal Convolution* kernels of **SK** convolve multiple columns/rows in parallel. In this way, vectorization imposes no overhead on the kernel.

(2) **Intra-Reduction Parallelism:** Five kernels compute a single output in each invocation. *Neon* exploits the parallelism *within* a single reduction operation. *When reduction function is both Associative and Commutative*, vector implementation breaks the reduction to *VRE* partial reductions and processes *VRE* inputs in each iteration. The only overhead of this reduction pattern is reducing *VRE* partial results to one final result. An example is the *Audible* kernel of **WA**, which calculates the energy of an audio frame as $\sum_{i=0}^{N} s_i^2$.

Vector implementation parallelizes the sequential reduction and changes the order of the operations AND source operands. Consequently, *a reduction operation must be Associative AND Commutative to be vectorized.* Otherwise, the scalar reduction requires significant algorithm modifications to be vectorizable.

For example, *Adler-32* kernel of **ZL** calculates two checksum values for an array of N characters: $S1 = \sum_{i=0}^{N} b_i$ and $S2 = \sum_{i=0}^{N} (N-i) \times b_i$. Scalar implementation sequentially consumes characters and calculates $S1 += b_i$ and $S2 += S1$. While the first reduction operation ($S1$) is associative and commutative, the second ($S2$) is neither associative nor commutative. A naïve approach to vectorize these reduction patterns is loop distribution. For example, we can calculate $S1$ in a separate loop and store partial values ($PS1_i$) in the memory. In the second loop, we modify $S2$ reduction operation to $S2 = \sum_{i=0}^{N} PS1_i$. This way, both $S1$ and $S2$ reduction operations are associative, commutative, and, hence, parallelizable. Swan benchmark suite contains five data-parallel kernels with this computation pattern.

6.2. Random Memory Access

Gather and Scatter operations access arbitrary memory locations, enabling random access patterns in 7 data-parallel kernels of Swan benchmark suite. While RISC-V Vector Extension (RVV) [9] implements random memory accesses (a.k.a, *Indexed Vector Load/Store* intrinsics), Arm Neon lacks a general-purpose solution for these access patterns.

We observe that random access patterns in all seven data-parallel kernels are employed for gathering values from *Look-Up Tables* for two reasons: (a) transforming vector elements (keys) to another domain (values). or (b) converting multiple operations to single look-up table access. For example, in each round of *AES* cipher, **BS** substitutes AES states (keys) with other states (values). The following code listing shows the semantics of look-up table accesses in these kernels.

```
template <typename T1, typename T2>
void LU_TBL(T1*table, T2* keys, T1* vals, int len)
    for (int i = 0; i < len; i++)
        vals[i] = table[keys[i]];
```

If the table contains less than 64 8-bit values, one can load it in multiple vector registers and access the table registers using Arm Neon's `TBL` instructions. This pattern is not employed in these data-parallel kernels since the table contains more than 64 values. Arm Neon provides domain-specific acceleration for two kernels with Look-Up Table access, *i.e.*, *AES* and *CRC-32* Cryptography intrinsics. Developing Look-Up Table access for *Neon* requires exporting elements of `key` vector register to the Scalar registers, performing Look-Up Table access for each individual key using scalar instructions, and packing the results back to the `value` vector register. Due to the high overhead of this operation, four kernels give up the benefits of look-up tables, and one kernel (*DES*) of **BS** does not provide *Neon* implementation.

While we exclude *DES* kernel from this paper's evaluation, we developed a *Neon* implementation to study the overhead of Look-Up Table accesses in vectorization without Random Memory Access intrinsics. Our evaluation shows 11% slow-down compared to *DES* scalar implementation. Next, we deprecated Look-Up Table accesses from the *Scalar* and *Neon* baselines. In this case, *Neon* outperforms *Scalar* by 2.1×. Our evaluation shows Table Look-Up accesses take 73% of total instructions in *DES* Neon implementation.

Therefore, *supporting intrinsics for gathering values from Look-Up Tables benefits seven data-parallel workloads.*

6.3. Strided Memory Access

Arm Neon supports memory accesses with stride values up to 4. 4-stride memory accesses are frequently used in Image Processing and Graphics libraries to *load and de-interleave* or *interleave and store* 4-channel pixel values. Non-unit stride memory accesses are implemented using Arm's multi-register data types and instructions. A multi-register data type, `TWxExR_t` encompasses `R` vector registers, each of which contains `E` elements (*i.e.*, *VRE*) of `W`-bit `T`-type data. The following code listing shows a strided memory load that fetches $E(16) \times R(4)$ data values and interleaves them between $R(4)$ vector registers. Therefore, adjacent vector elements within a vector register are loaded from memory locations with stride $R = 4$. A stride memory store operates in the opposite direction, where E elements from R registers are stored in memory with the stride of R.

```
uint8x16x4_t vld4q_u8(uint8_t const *ptr) {
    uint8x16x4_t Result;
    for (int element = 0; element < 16; element++)
        for (int reg = 0; reg < 4; reg++)
            Results[reg][element] = *ptr++;
    return Result;
}
```


Authorized licensed use limited to: University of Michigan Library. Downloaded on December 20,2024 at 14:35:50 UTC from IEEE Xplore. Restrictions apply.

Arm Neon also supports UZP (*de-interleave*) and ZIP (*interleave*) instructions, which, instead of accessing memory, move elements between vector registers with the stride value of 2. Table 6 shows the number of kernels containing non-unit stride value instructions and the average portion of these instructions in those kernels.

TABLE 6: NUMBER OF KERNELS AND PORTION OF STRIDED MEMORY ACCESSES.

Stride	Instruction	#Kernels	Avg. Portion
2	LD	1	2.9%
	ST	4	2.3%
	ZIP	5	6.2%
	UZP	7	3.0%
4	LD	8	5.8%
	ST	8	4.7%

Arm Neon multi-register data types and instructions efficiently encode memory accesses with non-unit strides of up to 4 in Image Processing and Graphics libraries. However, when a higher stride value is required, one needs to use multiple instructions that hurt the performance of the kernel. *RVV* architecture implements *Strided Vector Load/Store*, which can efficiently encode arbitrary strides memory accesses.

6.4. Matrix Transposition

Matrix Transposition is a common computation pattern used in six data-parallel kernels. In addition, Matrix Transposition is one of the frequently-used primitives of **XP** to transpose the input of Neural Network layers for cache-friendly memory access. *FFT* kernels of **PF** use eight ZIP instructions to transpose a 4×4 matrix of 32-bit floating-point numbers. Matrix transposition is only used in pre- and post-processing steps of FFT using on average 3.3% of the total **PF** instructions. Arm Neon provides transpose intrinsics that use two instructions to transpose every two elements of two vector registers using the following code listing.

```
int16x8x2_t vtrnq_s16(int16x8_t a, int16x8_t b) {
    int16x8x2_t Result;
    for (int element = 0; element < 8; element+=2)
     {
        // {a[0],b[0],a[2],b[2],...}
        Results[0][element] = a[element];
        Results[0][element+1] = b[element];
        // {a[1],b[1],a[3],b[3],...}
        Results[1][element] = a[element+1];
        Results[1][element+1] = b[element+1];
    }
    return Result;
}
```

Forward and Inverse DCT kernels of **LV** use intrinsics similar to the code listing above to transpose 8×8 *DCT* blocks of 16-bit integer numbers in 32 instructions. While this is the most efficient way of implementing 8×8 matrix transposition using Arm Neon, it takes, on average, 24.1% of total **LV** instructions.

To transpose an arbitrary-sized $M \times N$ matrix, one only needs to use $VRE \times VRE$ matrix transposition primitives to transpose each square sub-block separately. Thus, we calculate the latency of matrix transposition (lat_{AT}) using the following equation:

$$lat_{AT}(M,N) = \lceil \frac{M}{VRE} \rceil \times \lceil \frac{N}{VRE} \rceil \times lat_{AT}(VRE,VRE) \qquad (2)$$

For example, **LV** library also requires matrix transposition of 16×16 DCT blocks of $16-bit$ values. **LV** breaks the block into four quarters and transposes them separately using the mentioned 8×8 matrix transposition primitive. Therefore, **LV** achieves 16×16 matrix transposition in $4 \times 32 = 128$ cycels.

Matrix Transposition is an intensive computation pattern of data-parallel kernels. Swan contains efficient matrix transposition primitives as a part of **PF** *and* **LV** *libraries.*

6.5. Portable Vector APIs

While we showed that explicit vectorization achieves a higher speedup compared to auto-vectorization, performance improvement comes with the cost of re-developing kernels for a new ISA or ISA extension. To avoid kernel re-development for a new ISA, applications add a new layer of abstraction using a library of *Portable Vector APIs* with common simple functions and use these APIs across the data-parallel kernels. Vector libraries provide a set of implementations for different ISAs, and select an implementation based on the capabilities of the target platform. Therefore, to support a new ISA, applications are only required to implement vector APIs for the target instruction set, while the algorithm of the data-parallel functions remains the same.

PF defines a set of macros for common vector intrinsics to operate on floating-point vector variables. Due to the different instruction sets of various vector architectures, **PF** only supports basic intrinsics and does not take advantage of sophisticated vector instructions. For example, **PF** uses a naïve vector *Complex Multiplication* implementation that takes six instructions and eight cycles of Cortex-A76 core [4]. Armv8.2-A architecture supports multiply-add and multiply-subtract operations, with which Complex Multiplication requires four instructions and five cycles. Armv8.3-A supports complex multiply-add intrinsics that only require one instruction and take two cycles of Arm Cortex-A710 core [3]. However, none of these sophisticated intrinsics are supported by Intel SSE; therefore, **PF** only uses basic vector APIs, dropping *Neon* performance improvement to only 2.3×.

Webaudio modules (**WA**) of *Chromium* and *WebRTC* projects use vector APIs with simple vector operations such as vector convolution, multiplication, clip, etc. These fine-grain vector APIs load input arrays in the vector variables, perform a simple operation, and store the results back to the output arrays. Therefore, **WA** requires a load and a store for every arithmetic operation. In fact, around 59% of **WA**'s vector instructions are loads and stores, dropping instruction reduction to 3.4× and *Neon* speedup to 1.8×.

While Vector APIs substantially reduce the cost of supporting different vector processing architectures, they increase the number of instructions and significantly limit the benefits of vector processing.

Authorized licensed use limited to: University of Michigan Library. Downloaded on December 20,2024 at 14:35:50 UTC from IEEE Xplore. Restrictions apply.

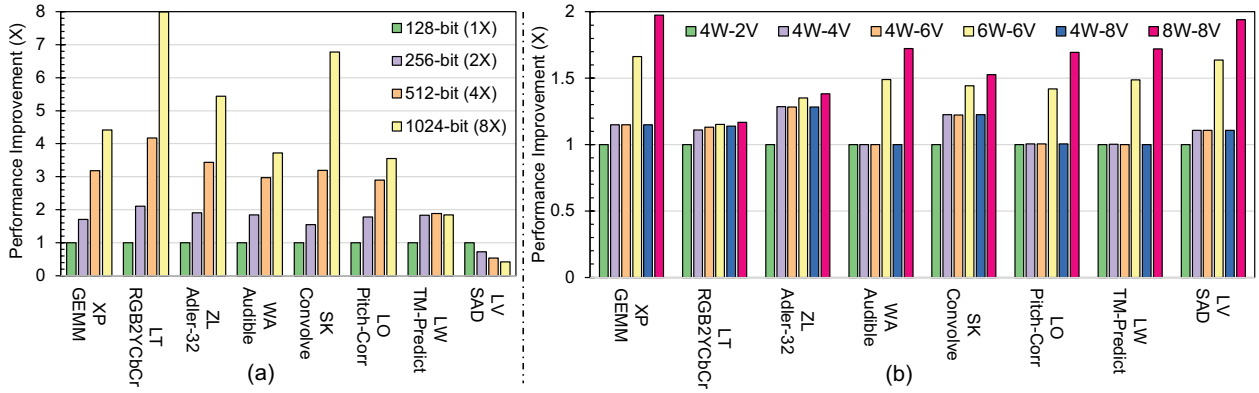

Figure 5: Neon performance scalability with: (a) wider vector registers and (b) more Vector execution units and O.o.O Ways.

7. Scalability Analysis

The evaluated Cortex-A76 Prime core baseline is equipped with two 128-bit ASIMD units. We study the performance scalability with wider vector registers and more vector execution units using eight kernels with different computation patterns that are representative of their libraries.

7.1. Wider Vector Registers

We redevelop kernels with wider registers using a fake Arm Neon library with 128, 256, 512, and 1024-bit registers and further optimize the algorithms based on the available set of vector instructions for each vector register width. Figure 5(a) shows the performance improvement of these implementations. Note that 2/4/8× wider registers exploit data-level parallelism to improve the performance at the cost of 2/4/8× larger vector register file and ASIMD units.

Wider registers are beneficial when a data-parallel kernel only requires streaming memory accesses. For example, **LT**'s *RGB-to-YCbCR* and **SK**'s Convolution enjoy high SIMD lane utilization of 99% and 98%. Therefore, 1024-bit implementations improve performance by 7.9× and 6.7× compared to the 128-bit implementations using 8× SIMD lanes.

GEMM-FP32 implementation exploits parallelism across the output matrix columns. When the number of output columns is not evenly divisible by *VRE*, *Neon* implementations use narrower registers that drop SIMD utilization from 98% of 128-bit to 89% of 1024-bit implementations.

WA's *Audible* kernel measures the energy of an audio channel using reduction. While Arm Neon efficiently reduces 128-bit vector registers to a scalar register with `U/SADDLV` instructions, we do not extend these instructions to wider implementations. This is because these instructions take 5/6 cycles for 128-bit vector registers [4], and reducing very wide registers results in longer latencies. Instead, we reduce wider registers to 128-bit in multiple iterations by breaking them into two halves. Hence, SIMD utilization and speedup of 1024-bit implementation drop to 74% and 3.7×.

When kernels process small multi-dimensional input data, wider SIMD lanes require numerous vector manipulation instructions. For example, **LV**'s *Sum of Absolute Differences (SAD)* and **LW**'s *TM-Prediction* kernels need to fetch data from 8×8 and 16×16 blocks of pixels. While 128-bit implementation encodes fetching a row of the input data efficiently with one vector load instruction, wider registers require loading each row and packing them before processing. These implementations are significantly bounded by the vector manipulation instructions due to the *Neon*'s inability to encode multi-dimensional strided accesses efficiently. Therefore, wider registers do not benefit these kernels.

7.2. Increasing the Number of ASIMD Units

Figure 5(b) shows the performance improvement of an Arm core with 2/4/6/8 128-bit ASIMD units (V). These configurations take advantage of vector Instruction-Level Parallelism (ILP) at the cost of more vector register file ports and ASIMD units. In addition, we employ 2/4/6/8 decode and commit ways (W) to issue enough instructions to the vector execution units. 4W-2V baseline is the evaluated Cortex-A76 core of Table 3.

We observe that increasing the number of vector execution pipelines to more than decode ways (*i.e.*, 4W-6V and 4W-8V implementations) provides limited performance improvement. This is because ASIMD units are under-utilized due to the lack of enough issued instructions. Hence, the core is unable to exploit the ILP of the workloads.

In configurations with enough decode ways, we observe that ASIMD unit utilization is limited by the inherent ILP of the workload. Note that we manually unroll loops to increase the vector ILP of the workloads. *GEMM* kernel of **XP** simultaneously computes 32 output columns using eight 128-bit registers. **LV**'s *SAD* kernel is unrolled to compute the Sum of Absolute Difference of a video frame in 32 accumulators of eight vector registers in parallel. *GEMM* and *SAD* provide the highest vector ILPs. Hence, 8W-8V configuration outperforms 4W-2V by 1.9× in both kernels using 4× ASIMD units.

RGB-to-YCbCr kernel of **LT** converts the color space of 16 pixels in four vector registers in parallel. However, due to the high vector register pressure of this kernel, LLVM's bottom-up list instruction scheduler [31] schedules vector instructions close to their use and reduces the ILP. Therefore, ILP is limited to computing three color spaces of four output

Authorized licensed use limited to: University of Michigan Library. Downloaded on December 20,2024 at 14:35:50 UTC from IEEE Xplore. Restrictions apply.

pixels, limiting the 8W-8V configuration performance to 1.2× of the 4W-2V configuration (4× ASIMD units).

8. Mobile Application Processors

Mobile SoCs are equipped with different domain-specific accelerators such as GPU and DSP. These can be alternatives to vector processing for data-parallel kernels. However, domain-specific accelerators suffer from data transfer and kernel launch overheads negating their acceleration benefits for fine-grain kernels. Vector processing, on the other hand, enjoys tight integration to the CPU and fine-grain instruction interleaving without data transfer or kernel launch overhead.

We observe that the first nine libraries of Table 2 are not accelerated on GPUs. Table 7 compares *only* kernel launch overhead with the average *Neon* execution time of these libraries. We measure the time of launching a dummy kernel on Adreno 640 GPU (Qualcomm OpenCL driver) and Hexagon 690 DSP (fastRPC). On average, GPU and DSP kernel launch overhead alone is 1.9× and 19% of the average execution time of the aforementioned nine libraries on Neon. These libraries provide fine-grained APIs that are necessary for workloads with parallel and serial code interleaving. Therefore, GPU and DSP are not employed in any of these libraries due to the data transfer cost and kernel launch overhead. In addition, GPU suffers from high power consumption, which reduces the battery life of mobile devices. Programming DSP is complex, and it only supports fixed-point operations.

TABLE 7: GPU AND DSP KERNEL LAUNCH OVERHEAD VS. NEON TOTAL EXECUTION TIME.

Kernel Launch Overhead		Neon Kernel Execution		
Adreno 640 GPU	Hexagon 690 DSP	Min.	Avg.	Max.
230μs	20μs	0.1μs	117μs	1209μs

Furthermore, we compare *Neon* and GPU performance of **XP**'s GEMM and SpMM (80% sparse) kernels for 156 different convolutional layers. We use 2 OpenCL matrix multiplication libraries [18, 35] for GPU and eliminate its memory copy overhead due to the unified memory of mobile SoCs [39]. Figure 6 shows that vector processing outperforms GPU in matrix multiplication with less than 4M FP32 operations despite 96× less throughput. The primary reason for better *Neon* performance is the lack of offload overhead, thanks to the tight integration with the code. This overhead is illustrated in the figure by horizontal dash lines.

9. Conclusion and Future Work

In this work, we presented Swan, the first mobile vector processing benchmark suite with 59 data-parallel kernels from four commonly-used mobile applications. Using our diverse set of workloads, we analyzed the performance, power, and energy consumption improvement of vectorized kernels, performance bottlenecks of vector processing, limitations of compiler vectorization, and common intensive computation patterns. In addition, we analyzed the performance scalability of vector processing with wider instructions and more vector execution pipelines. We discussed the inefficiency of domain-specific acceleration for the fine-grain data-parallel kernels

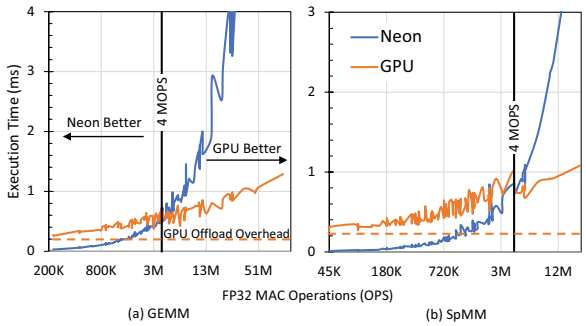

Figure 6: Neon and GPU performance comparison of XP's GEMM and SpMM kernels w.r.t. different operation counts.

and compared vector processing performance with GPU for different problem sizes.

Swan is maintained online on GitHub. We plan to extend Swan in these directions:

Other Vector ISA Extensions: While Arm Neon is the most widely-used vector architecture of Mobile devices, RISC-V Vector Extension (RVV) [9] and Arm Scalable Vector Extension (SVE) [43] provide wider registers and sophisticated operations such as Random Memory Accesses. This is appealing for workloads with irregular memory accesses. Prior work [14] provides an RVV benchmark suite for Desktop and Server applications. We aim to equip Swan with RVV and SVE implementations of Mobile applications.

Android Applications and Java Vector API: Java is the primary programming language for Android Application development, which supports vector operations using Java Vector API [36]. JVBench [12] provides a benchmark of Java applications and analyzes the performance limitations of Java Just-In-Time Compiler Auto-Vectorization. However, JVBench borrows data-parallel applications from prior CMP and GPU benchmark suites that are not suitable for vector processing. We plan to add vectorized Java Applications to Swan from various Mobile Application domains such as Social Media, Calendar, E-mail Client, and Navigation.

Vectorized Mobile Web Applications: While Swan contains 12 libraries of the Chromium Project, prior work [13, 37] show that V8 JavaScript and WebAssembly Engine take a significant portion of the browser's time and energy. Web Assembly [41] is a complementary language to JavaScript for high-performance Web Applications. V8 JavaScript and WebAssembly Engine supports vector operations through WebAssembly SIMD Proposal [26]. We plan to extend the Swan benchmark suite with WebAssembly SIMD applications.

Acknowledgments

We thank the anonymous reviewers for their suggestions which helped improve this paper. This work was supported in part by the NSF under the CAREER-1652294 and NSF-1908601 awards, JSPS KAKENHI Grant Number JP22K21284, and the Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA.

Authorized licensed use limited to: University of Michigan Library. Downloaded on December 20,2024 at 14:35:50 UTC from IEEE Xplore. Restrictions apply.

References

[1] Android - secure and reliable mobile operating system. [Online]. Available: https://www.android.com/

[2] Arm architecture reference manual for a-profile architecture. [Online]. Available: https://developer.arm.com/documentation/ddi0487/ja/

[3] Arm cortex-a710 core software optimization guide. [Online]. Available: https://developer.arm.com/documentation/PJDOC-466751330-14951/latest/

[4] Arm cortex-a76 software optimization guide. [Online]. Available: https://developer.arm.com/documentation/pjdoc466751330-7215/latest/

[5] Chromium. [Online]. Available: https://www.chromium.org/Home/

[6] "Multibench algorithms and workload datasheets," *The Embedded Microprocessor Benchmark Consortium*. [Online]. Available: https://www.eembc.org/multibench/docs/MultiBench_Algorithms_and_Workload_Datasheets.pdf

[7] Pdfium. [Online]. Available: https://pdfium.googlesource.com/pdfium/+/master/README.md

[8] Polybench. [Online]. Available: https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/

[9] riscv-v-spec. [Online]. Available: https://github.com/riscv/riscv-v-spec

[10] Webrtc. [Online]. Available: https://webrtc.org/

[11] Arm. Vfp11 vector floating-point coprocessor technical reference manual. [Online]. Available: https://developer.arm.com/documentation/ddi0274/latest/

[12] M. Basso, A. Rosà, L. Omini, and W. Binder, "Java vector api: Benchmarking and performance analysis," in *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction*, ser. CC 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 1–12. [Online]. Available: https://doi.org/10.1145/3578360.3580265

[13] D. H. Bui, Y. Liu, H. Kim, I. Shin, and F. Zhao, "Rethinking energy-performance trade-off in mobile web page loading," in *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, ser. MobiCom '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 14–26. [Online]. Available: https://doi.org/10.1145/2789168.2790103

[14] J. M. Cebrian, M. Jahre, and L. Natvig, "Parvec: Vectorizing the parsec benchmark suite," *Computing*, vol. 97, no. 11, p. 1077–1100, nov 2015. [Online]. Available: https://doi.org/10.1007/s00607-015-0444-y

[15] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, 2009, pp. 44–54.

[16] S. Chetoui, R. Shahi, S. Abdelaziz, A. Golas, F. Hijaz, and S. Reda, "Arbench: Augmented reality benchmark for mobile devices," in *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2022, pp. 242–244.

[17] J. Clemons, H. Zhu, S. Savarese, and T. Austin, "Mevbench: A mobile computer vision benchmarking suite," in *2011 IEEE International Symposium on Workload Characterization (IISWC)*, 2011, pp. 91–102.

[18] clMathLibraries. A software library containing sparse functions written in opencl. [Online]. Available: https://github.com/clMathLibraries/clSPARSE/

[19] T. N. Y. T. Company. (1994) Makers unveil pcs with intel's mmx chip. [Online]. Available: https://archive.nytimes.com/www.nytimes.com/library/cyber/week/010997intel.html

[20] A. Developers. Simpleperf. [Online]. Available: https://developer.android.com/ndk/guides/simpleperf

[21] R. Espasa, M. Valero, and J. E. Smith, "Vector architectures: past, present and future," in *Proceedings of the 12th international conference on Supercomputing*, 1998, pp. 425–432.

[22] S. Gal-On and M. Levy, "Exploring coremark a benchmark maximizing simplicity and efficacy," *The Embedded Microprocessor Benchmark Consortium*, 2012.

[23] D. Greenley, J. Bauman, D. Chang, D. Chen, R. Eltejaein, P. Ferolito, P. Fu, R. Garner, D. Greenhill, H. Grewal, K. Holdbrook, B. Kim, L. Kohn, H. Kwan, M. Levitt, G. Maturana, D. Mrazek, C. Narasimhaiah, K. Normoyle, N. Parveen, P. Patel, A. Prabhu, M. Tremblay, M. Wong, L. Yang, K. Yarlagadda, R. Yu, R. Yung, and G. Zyner, "Ultrasparc: the next generation superscalar 64-bit sparc," in *Digest of Papers. COMPCON'95. Technologies for the Information Superhighway*, 1995, pp. 442–451.

[24] B. Hawkins, B. Demsky, D. Bruening, and Q. Zhao, "Optimizing binary translation of dynamically generated code," in *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '15. USA: IEEE Computer Society, 2015, p. 68–78.

[25] HTMLSTRIP. Alexa top 1000 most visited websites. [Online]. Available: https://www.htmlstrip.com/alexa-top-1000-most-visited-websites

[26] ——. Simd proposal for webassembly. [Online]. Available: https://github.com/WebAssembly/simd

[27] V. Janapa Reddi, D. Kanter, P. Mattson, J. Duke, T. Nguyen, R. Chukka, K. Shiring, K.-S. Tan, M. Charlebois, W. Chou *et al.*, "Mlperf mobile inference benchmark: An industry-standard open-source machine learning benchmark for on-device ai," *Proceedings of Machine Learning and Systems*, vol. 4, pp. 352–369, 2022.

[28] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible dram simulator," *IEEE Computer Architecture Letters*, vol. 15, no. 1, pp. 45–49, 2016.

[29] R. Lee, "Realtime mpeg video via software decompression on a pa-risc processor," in *Digest of Papers. COMPCON'95. Technologies for the Information Superhighway*, 1995, pp. 186–192.

[30] W. Lee, J. Lee, B. K. Park, and R. Y. C. Kim, "Microarchitectural characterization on a mobile workload," *Applied Sciences*, vol. 11, no. 3, 2021. [Online]. Available: https://www.mdpi.com/2076-3417/11/3/1225

[31] B. C. Lopes and R. Auler, *Getting started with LLVM core libraries*. Packt Publishing Ltd, 2014.

[32] S. Maleki, Y. Gao, M. J. Garzar´n, T. Wong, and D. A. Padua, "An evaluation of vectorizing compilers," in *2011 International Conference on Parallel Architectures and Compilation Techniques*, 2011, pp. 372–382.

[33] N. Mammeri and B. Juurlink, "Vcomputebench: A vulkan benchmark suite for gpgpu on mobile and embedded gpus," in *2018 IEEE International Symposium on Workload Characterization (IISWC)*, 2018, pp. 25–35.

[34] C. Mendis, A. Jain, P. Jain, and S. Amarasinghe, "Revec: Program rejuvenation through revectorization," in *Proceedings of the 28th International Conference on Compiler Construction*, ser. CC 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 29–41. [Online]. Available: https://doi.org/10.1145/3302516.3307357

[35] C. Nugteren, "Clblast: A tuned opencl blas library," in *Proceedings of the International Workshop on OpenCL*, ser. IWOCL '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3204919.3204924

[36] OpenJDK. Jep 426: Vector api (fourth incubator). [Online]. Available: https://openjdk.org/jeps/426

[37] N. Peters, S. Park, S. Chakraborty, B. Meurer, H. Payer, and D. Clifford, "Web browser workload characterization for power management on hmp platforms," in *2016 International Conference on Hardware/-Software Codesign and System Synthesis (CODES+ISSS)*, 2016, pp. 1–10.

Authorized licensed use limited to: University of Michigan Library. Downloaded on December 20,2024 at 14:35:50 UTC from IEEE Xplore. Restrictions apply.

[38] C. J. Purcell, "The control data star-100: Performance measurements," in *Proceedings of the May 6-10, 1974, National Computer Conference and Exposition*, ser. AFIPS '74. New York, NY, USA: Association for Computing Machinery, 1974, p. 385–387. [Online]. Available: https://doi.org/10.1145/1500175.1500257

[39] Qualcomm. Heterogeneous memory management. [Online]. Available: https://developer.qualcomm.com/software/heterogeneous-compute-sdk/app-notes/buffers

[40] C. Ramírez, C. A. Hernández, O. Palomar, O. Unsal, M. A. Ramírez, and A. Cristal, "A risc-v simulator and benchmark suite for designing and evaluating vector architectures," *ACM Trans. Archit. Code Optim.*, vol. 17, no. 4, nov 2020. [Online]. Available: https://doi.org/10.1145/3422667

[41] A. Rossberg, "Webassembly core specification." [Online]. Available: https://www.w3.org/TR/wasm-core-1/

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

[43] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu, A. Reid, A. Rico, and P. Walker, "The arm scalable vector extension," *IEEE Micro*, vol. 37, no. 2, pp. 26–39, 2017.

[44] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *Center for Reliable and High-Performance Computing*, vol. 127, p. 27, 2012.

[45] W. Watson, "The ti asc: a highly modular and flexible super computer architecture," in *Proceedings of the December 5-7, 1972, fall joint computer conference, part I*, 1972, pp. 221–228.

[46] A. Yasin, "A top-down method for performance analysis and counters architecture," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014, pp. 35–44.

[47] Q. Zheng, Y. Chen, R. Dreslinski, C. Chakrabarti, A. Anastasopoulos, S. Mahlke, and T. Mudge, "Wibench: An open source kernel suite for benchmarking wireless systems," in *2013 IEEE International Symposium on Workload Characterization (IISWC)*, 2013, pp. 123–132.

Authorized licensed use limited to: University of Michigan Library. Downloaded on December 20,2024 at 14:35:50 UTC from IEEE Xplore. Restrictions apply.