This document is downloaded from DR-NTU (https://dr.ntu.edu.sg)
Nanyang Technological University, Singapore.

Comparing soft and hard vector processing in FPGA-based embedded systems

Soh, Jun Jie; Kapre, Nachiket

2014

Soh, J. J., & Kapre, N. (2014). Comparing soft and hard vector processing in FPGA-based embedded systems. 2014 24th International Conference on Field Programmable Logic and Applications (FPL).

https://hdl.handle.net/10356/81218

https://doi.org/10.1109/FPL.2014.6927467

© 2014 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works. The published version is available at: [http://dx.doi.org/10.1109/FPL.2014.6927467].

Downloaded on 26 Jul 2024 03:55:14 SGT

Comparing Soft and Hard Vector Processing in FPGA-based Embedded Systems

Soh Jun Jie
School of Computer Engineering
Nanyang Technological University
50 Nanyang Avenue, S639798
Email: jjsoh001@e.ntu.edu.sg

Nachiket Kapre
School of Computer Engineering
Nanyang Technological University
50 Nanyang Avenue, S639798
Email: nachiket@ieee.org

***Abstract*—**
Soft vector processors can augment and extend the capability of embedded hard vector processors in FPGA-based SoCs such as the Xilinx Zynq. We develop a compiler framework and an auto-tuning runtime that optimizes and executes data-parallel computation either on the scalar ARM processor, the embedded NEON engine or the Vectorblox MXP soft vector processor as appropriate. We consider computational conditions such as precision, vector length, chunk size, IO requirements under which soft vector processing can outperform scalar cores and hard vector blocks. Across a range of data-parallel benchmarks, we show that the MXP soft vector processor can outperform the NEON engine by up to 3.95$\times$ while saving 9% dynamic power (0.1W absolute). Our compilation and runtime framework is also able to outperform the gcc NEON vectorizer under certain conditions by explicit generation of NEON intrinsics and performance tuning of the auto-generated data-parallel code.

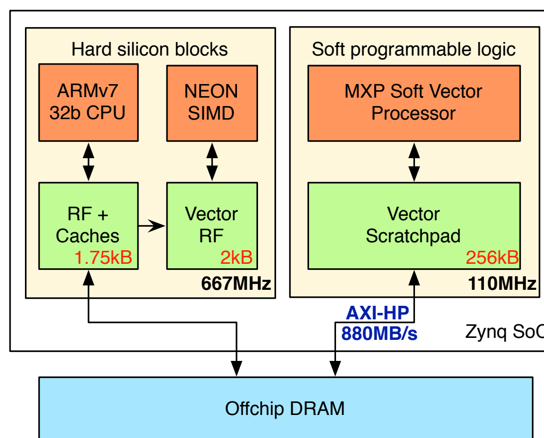

Figure 1: Xilinx Zynq SoC Platform Compute Organization

I. Introduction

Embedded computing SoCs (systems-on-chip) increasingly support a heterogeneous assembly of μarchitectures and co-processors organized around a central co-ordination processor. This co-ordination processor is typically a low-power, low-cost ARM or MIPS implementation that runs the host OS or firmware that drives the rest of the system. Depending on system requirements, we may pick SoCs that include vector processors (*e.g.* ARM NEON), graphics engines (*e.g.* NVidia Tegra, ARM Mali), custom analog blocks (*e.g.* Cypress SoC), or FPGA logic (*e.g.* Xilinx Zynq and Altera SOC solutions). While this diversity brings freedom and versatility, designers of these embedded systems are expected to manually decide how to assign and optimize computational kernels for these heterogeneous SoC blocks.

The Xilinx ZedBoard platform, shown in Figure 1, has a Zynq SoC which includes the ARM Cortex A9-series CPU as the central co-ordination processor augmented with the NEON SIMD hard vector engines under the same memory hierarchy as well as a tightly-coupled FPGA logic substrate connected over a low-latency, high-bandwidth AXI and ACP interconnect. In this case, the designer has to decide whether to map data-parallel code to (1) simply run as scalar code on the ARMv7 CPU, (2) run as data-parallel code on the NEON hard vector engine, or (3) use FPGA logic. The proximity and tight integration of the NEON engine offers a tempting option for effortless acceleration for many existing embedded applications through a recompile of the code. However, if power utilization and absolute performance is desired, the FPGA logic may offer a superior alternative. To retain the fast development times and performance of mapping to NEON, we consider the Vectorblox MXP [12] soft vector processor as the alternative implementation. MXP is an FPGA-based soft vector processor designed to perform data-parallel tasks with high performance. It is able to deliver high performance due to parallel scratchpad accesses and support for fast DMA transfers. When using the MXP soft vector engine, the developers can choose the number of vector lanes, scratchpad capacity, precision and other parameters that match application requirements. We show specifications of the NEON and MXP block in Table I. In this paper, we develop an automated framework to select between soft vector processors (programmed on FPGA logic) and embedded hard vector blocks for acceleration of data-parallel code in embedded SoC platforms. Our framework also generates optimized NEON code using auto-vectorization and explicit generation of NEON intrinsics if desired.

With this framework, we address the following questions: *Under what conditions does the MXP soft vector processor*

offer superior performance and power benefits compared to NEON hard vector engines? How do we automate this decision process while generating optimized code for these systems? How do we optimize the configuration and operation of the soft vector processor without relying on manual tuning?

Consider the simple example $a \cdot x^2 + b \cdot x + c$ implemented inside a data-parallel for loop operating on 8-bit data. When the loop trip count is low (<128 with warmed-up caches), we might as well avoid vectorization (hard or soft) and simply run code as scalar instructions on the ARM CPU. For a larger trip count (>≈0.5M with uncached data), we can exploit automatic gcc vectorization for NEON (or when using NEON intrinsics) for achieving a 3.7× speedup. With NEON intrinsics we can achieve somewhat larger speedups in some instances compared to auto-vectorization. Finally, the MXP soft vector processor delivers faster runtime at virtually all trip counts (≈1.3× faster than NEON). The result of these decisions change with the benchmark, precision and internal state requirement of the computation. Our compiler and runtime system help the developer navigate this space of choices.

The key contributions of this paper include:

- Development of a compiler backend that targets ARM NEON intrinsics, ARM NEON `gcc` and the MXP soft vector processor.
- Design of an auto-tuning framework that helps select between ARM scalar, NEON hard vector and MXP soft vector engine.
- Performance and Power characterization of the different hardware configurations using the ZedBoard across a range of data-parallel kernels.

II. Background

A. Zynq SoC Platform

In Table I, we show the key architecture specifications of the scalar ARM CPU, the NEON SIMD engine as well as the fully parallel, best-case configuration of the MXP soft processor on the Xilinx ZedBoard platform. As we can see, the 28nm hard silicon ARM Cortex A9 CPU and NEON SIMD engines are configured to run at 667MHz (-1 speed grade part) yielding an 8b peak throughput of 0.667 Gop/s and 3.9 Gops/s respectively. The 16-lane MXP processor can only reach 110 MHz but is still capable of 8b peak throughput of 7.04 Gop/s. While the MXP can scale up to 128 lanes, the Zynq device on the ZedBoard limits us to 16 lanes. When mapping data-parallel computation to the NEON engine, we must load/store the vector operands into a constrained vector register file (32×64b) whereas the MXP processor permits loads/stores from a software-managed scratchpad up to 256kB (again limited by the ZedBoard capacity). This disparity in register storage state allows the MXP to support multiple intermediate live variables without incurring memory transfer penalties. Despite being a hard vector core, the NEON SIMD engine does permit a degree of programmability; we are allowed to switch between 8 lanes at 8b, 4 lanes as 16b and 2 lanes at 32b when using doubleword vectors. The programming challenge is to optimize code under these user-selectable metrics and architecture parameters.

Table I: Architecture Specifications

Metric	ARMv7 CPU	NEON [1]	MXP [12]
μarch	Scalar	SIMD	Soft vector
Clock Freq.	667 MHz	667 MHz	110 MHz
Throughput			
32b (Gops/s)	0.6	1.3	1.7
16b (Gops/s)	0.6	2.6	3.5
8b (Gops/s)	0.6	3.9	7
Memory	1.75kB	2kB	4–256kB
	(Scalar RF)	(Vector RF)	(Scratchpad)
Lanes	1	2×32b	1–16×32b
		4×16b	2–32×16b
		8×8b	4–64×8b

B. Programming Vector Architectures

Programming the NEON SIMD engine can be as trivial as invoking `gcc` with the right compiler options. The `gcc` compiler has an auto-vectorizing backend that detects suitable for loops and converts them into NEON-compatible code. However, the compiler might miss several opportunities for parallelization including register reuse in the small vector register file and potential memory transfer optimizations. Furthermore, when programming the MXP soft processor, the programmer may be unable to easily pick the optimal set of resource parameters (*e.g.* vector lanes, scratchpad size) that best match the design requirements. This may result in overprovisioned resources and sub-optimal performance.

To address these challenges, we select the open-source SCORE [2] compiler framework as the development environment for constructing our vectorizing backends. SCORE is a stream-oriented framework for reconfigurable execution that supports automated compilation of high level parallel dataflow operators to low level hardware. Computations described in SCORE obey the dataflow compute model. Originally developed for streaming circuit design, it has been extended to support a varity of other backends such as sequential control [6], and fixed-point ciruit generation [9], [5]. In this paper, we adapt the compiler to support two extra backends for the NEON SIMD engine and the MXP soft processor.

(a) SCORE

```
poly(
input int x,
output int y)
{
  state only(x):
    y = a*x*x
        + b*x
        + c;
}
```

(b) ARMv7 C

```
for(i=0;i<N;i++)
{
  y[i] = a*x[i]*x[i]
         + b*x[i]
         + c;
}
```

(c) NEON intrinsics

```
int8x8_t t1, t2, t3;
t1 = vmul_s8(a, x);
t2 = vmul_s8(t1, x);
t3 = vmul_s8(b, x);
t1 = vadd_s8(t2, t3);
y = vadd_s8(t1, c);
```

(d) MXP code

```
vbx_dma_to_vector(x,x_host);
vbx_byte_t *t1, *t2, *t3;
vbx(SVB, VMUL, t1, a, x);
vbx(VVB, VMUL, t2, t1, x);
vbx(SVB, VMUL, t3, b, x);
vbx(VVB, VADD, t1, t2, t3);
vbx(SVB, VADD, y, c, t1);
vbx_dma_from_vector(y,y_host);
```

Table II: Comparing code generation for the different backends

III. Vectorizing Compiler and Auto-Tuning Framework

To achieve best performance and power characteristics from the embedded platform, system developers will usually rely on manual mapping, tuning and optimization heuristics. As discussed earlier in Section I, the availability of heterogeneous compute elements further adds to developer burden. In Figure 2, we show a block diagram of our compiler and runtime system to help alleviate developer costs.

The goal of our compiler framework is to generate optimized code for the three targets on the Zynq platform. This allows the developers to describe their data-parallel kernels once without manual porting effort. Furthermore, our compiler framework also supports optimizations such as straightforward register reuse and vector strip mining.

The goal of the auto-tuning step is to (1) pick the best hardware target for a given problem configuration, and (2) perform runtime optimizations on data transfer mechanics. Our auto-tuner runs optimized code under various configurations of bitwidth, lane count, vector length and datasize to identify the best backend target for our code. From our experimental results we also identify a simplistic and preliminary predictive model that can guide the mapping process.

In Table II, we see the input SCORE program and the generated backend code for the simple $a \cdot x^2 + b \cdot x + c$ example. The SCORE program is a stateless, data-parallel operation that consumes inputs x to generate y. As SCORE is originally designed to generate FPGA circuits, this code is converted to streaming, pipelined Verilog where a new set of values is injected into the circuit in each cycle. For the scalar C backend, we perform a simple source-source translation and wrap the code around a for-loop of user-specified length. For the NEON backend, we first attempt `gcc` auto-vectorization of the scalar code by simply using an appropriate build setup. When we generate NEON intrinsics directly, we use a dataflow expression rewriting engine that parallelizes the dataflow expressions and runs a register reuse optimization pass to conserve the vector registers utilized by our code. Here, an intrinsic is an inline-assembly function used in the generated C code that directly calls a documented ARM NEON instruction at specific bitwidth and lane-count

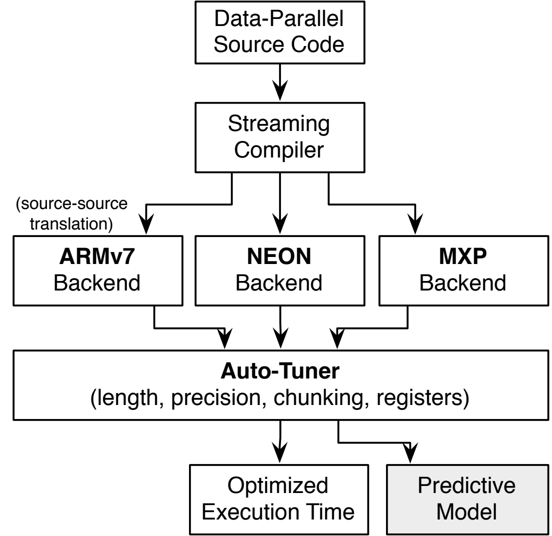

Figure 2: Block Diagram of the Compiler and Runtime Framework

combination thereby overriding `gcc` compiler. This could be useful in cases where the compiler may not correctly detect NEON optimization opportunities. This may be seen in the way temporary variable $t1$ gets used twice in the code block. We use a minor modification of the same backend to also generate optimized MXP code. In addition to these simple dataflow code-generation passes, we wrap the code blocks with double-buffered loads and stores to allow overlapped evaluation of computation and communication phases (not shown in Figure II for simplicity).

IV. Methodology

We develop an experimental methodology to compare the different vector backends and architectures based on a range of benchmark problems. In Table III we show our set of benchmark problems that were derived from kernels from EEMBC, SCORE and certain OpenCV functions. These benchmarks are characterized by variations in the compute (arithmetic), intermediate storage requirements, IO requirements, and precision. We tested the data-parallel problems on a range of trip counts up to 2^{14} elements.

Table III: Benchmark Properties

Name	Description	Ops.	Reg.	IOs
EEMBC-derived				
`rgb2gr`	rgb filter	4	3	3
`rgb2lm`	rgb filter	7	6	4
OpenCV-derived				
`tap4flt`	fir filter	8	6	9
FX-SCORE-derived				
`vecadd`	vector addition	1	1	3
`poly`	degree-2 polynomial	3	2	3
`matmul`	2x2 matrix multiply	12	8	12
`dotprod`	4-elem dot product	3	2	5
`l1lin`	transistor linear	7	18	4
`l1sat`	transistor saturation	8	19	4

We use the Xilinx ZedBoard for our experiments. We use the Xillinux OS v1.1 (Ubuntu) when executing scalar ARMv7 and NEON code. We use `gcc` v4.4 for generating ARM binaries and exploit auto-vectorization for a portion of our experimental flow. To compile NEON code using the automatic vectorization pass we use the `-ftree-vectorize` switch in conjunction with the `-mfpu=neon` option. We use `<arm_neon.h>` v4.4 when directly generating NEON intrinsics from the SCORE compiler. In this case, we simply supply the `-mfpu=neon` option. For generating bitstreams for the ZedBoard for various MXP experiments, we use Xilinx ISE 14.7.

For timing measurements, we use ARM hardware timers for extracting runtime information of the ARM, NEON and MXP binaries. For the MXP soft processor, we use the MXP timing API. For ensuring statistical significance of measured data, we consider averaged values measured across several timing runs to minimize the effect of measurement noise. This also considers the impact of caching to ensure we perform fair comparisons across the different hardware backends.

We measure power usage of the ZedBoard using Energenie Power Meter (2% accuracy). We record power measurements after reaching steady state on the vectorized code execution. We measure steady state power utilization for the scalar and NEON implementation when running Xillyinux from the SDCARD. The MXP implementations are programmed and executed over the PROGUSB cable connected to a host PC which adds a non-trivial power overhead. We recorded a stable 5.1W steady-state power utilization when running Xillyinux which rose to 5.5W with PROGUSB cable connected. We calculate runtime power utilization from these two baselines for the experiment as appropriate.

V. Results

Speedups and Runtime Comparisons: We first compare speedups achieved by the 16-lane MXP soft processor when

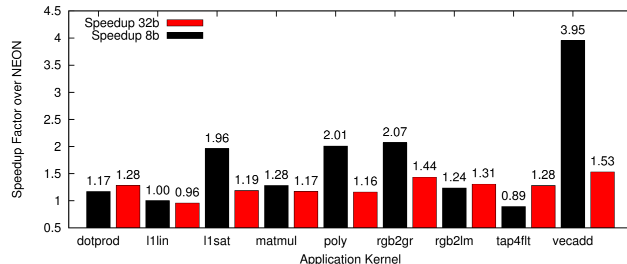

Figure 3: Speedups of MXP over best NEON implementation

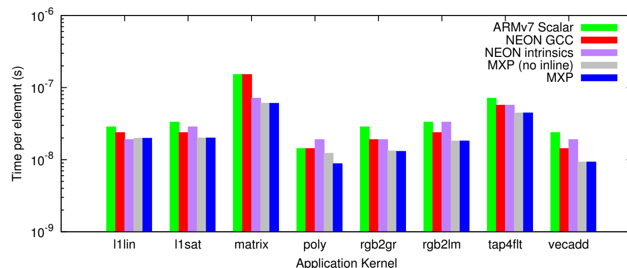

Figure 4: Comparing runtime of the different application kernels

compared to optimized NEON vector runtimes in Figure 3. We observe that in most cases that MXP outperforms NEON implementations by as much as 3.95× at 8b computations which drops to 2.07× at 32b computations. We would expect this drop as NEON has 8 8b lanes and only 2 32b lanes (See Table I). These speedups for the MXP are primarily due the superior double-buffered memory transfer optimizations made possible by the fast AXI-FPGA interconnect. We note that certain application kernels like `vecadd` show substantial speedups due to simple data-parallelism with few intermediate states. In contrast, kernels such as `l1lin` and `tap4flt` are unable to beat NEON speeds due to low IO to compute ratios (See Table III).

In addition to MXP and NEON performance, we are also interested in understanding the quality of code generated by our compiler framework. The `gcc` compiler is already capable of generating vectorized code for NEON backend. However, in conjunction with explicit intrinsic generation, dynamic auto-tuning and register allocation passes, our framework offers the ability to improve NEON performance. For the MXP soft processor we use the inlining compiler optimization to lower overheads and tune lane count and vector strip length as appropriate. To illustrate this, in Figure 4, we show the runtime comparison between the ARMv7 scalar CPU, NEON SIMD evaluation and MXP soft processor mappings for 32b computations. We observe that the inlining allows MXP code to run marginally faster due to lower function call overheads. When using explicitly generated intrinsics for the NEON, we see a mix of

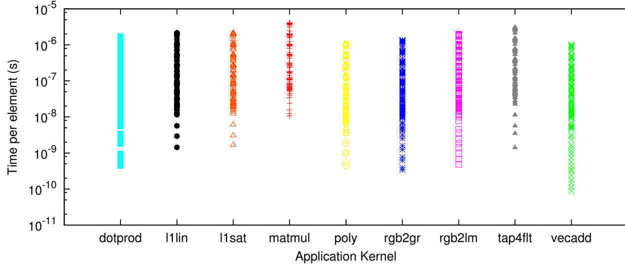

Figure 5: MXP Auto-Tuning Dynamic Range

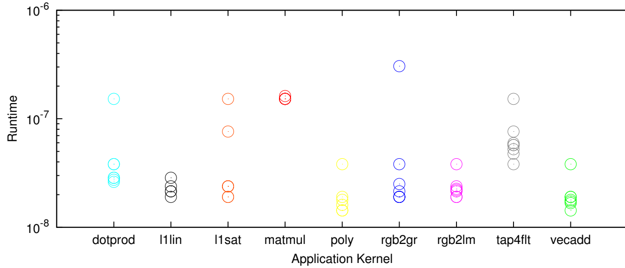

Figure 6: NEON Auto-Tuning Dynamic Range

improvements and slowdowns across the benchmark set suggesting additional optimizations may be necessary to comprehensively beat the `gcc` auto-vectorizer. Additionally, for certain cases like `matrix` and `poly`, we do not observe benefits for NEON SIMD execution over scalar CPU due to simplistic calculations and memory-bottlenecks.

Need for Auto-Tuning: To understand the importance of auto-tuning on user code, we represent the space of measured runtimes on the MXP soft vector processor in Figure 5. As you can see, the dynamic range of possible combinations can be as high as four orders of magnitude. This large range suggests a critical need to pick the best parameters. The choice of parameters also affects resource costs (not shown) but for our experiments, we are already tightly constrained by the 16-lane limit imposed by the ZedBoard FPGA capacity. Being a hard vector architecture with few programmable elements, the auto-tuning range is much lower for the NEON as shown in Figure 6. However, the order of magnitude range is still large enough to merit a tuner-assisted optimization.

Tuning MXP Performance: When designing soft vector processors, we have the unique capability of being able to select the best lane configuration desired for optimum performance while keeping resource utilization low. For small datasets, we would expect fewer parallel lanes to offer better value for resource consumption as we will be unable to fully saturate all lanes. Under bandwidth-limited circumstances, excessively large lane counts would be equally wasteful of resources. Most applications we characterized preferred a lane count of 8 or 16 with sufficient parallel work. As shown in Figure 7, our auto-tuner identifies `poly` benchmark as

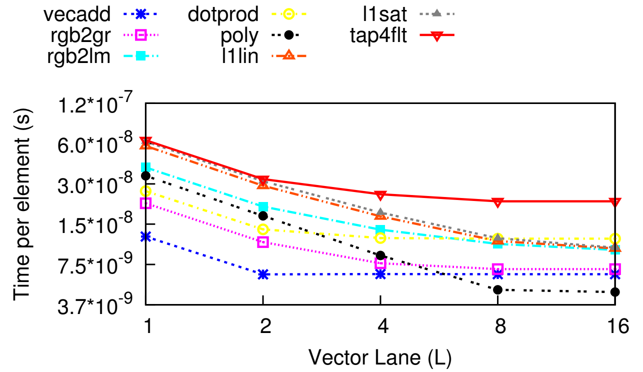

Figure 7: Tuning Lane Width for the MXP soft-processor (16b data and VL of 1024)

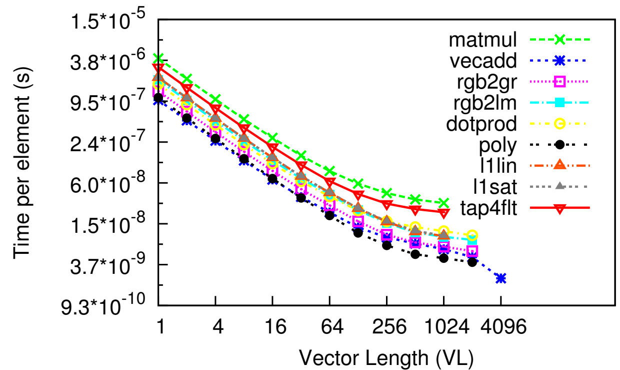

Figure 8: Tuning Vector Length for MXP (16b data and 16-lane design)

the most scalable design with continuing improvements at 16 lanes, while the `vecadd` benchmark saturates quickly at 2 lanes due to the accumulation loopback limit. Rest of the benchmarks show saturated speedups above 8 lanes.

An additional tunable feature of the MXP soft processor, is our ability to choose the vector strip length. The MXP eschews the vector register file in favor of a user-managed scratchpad. In certain cases, we can achieve performance parity with a larger lane count design simply by choosing an appropriate vector length for best mapping to a given scratchpad size. As onchip memory resources can be substantially denser than logic, this optimization can offer a cheaper alternative to improving vector code performance than adding expensive additional lanes. In Figure 8, we show the impact of varying the vector length with performance. For our benchmarks, the largest achievable vector length can vary between 1024–4096 due to variations in IO capacity and internal state requirements (see Table III). However, performance of most benchmarks saturates above a vector length of 1024 in any case.

Different data-parallel workloads require the vectorizable loop body to run for varying trip counts. In Figure 9, we observe the impact of overall trip counts on performance.

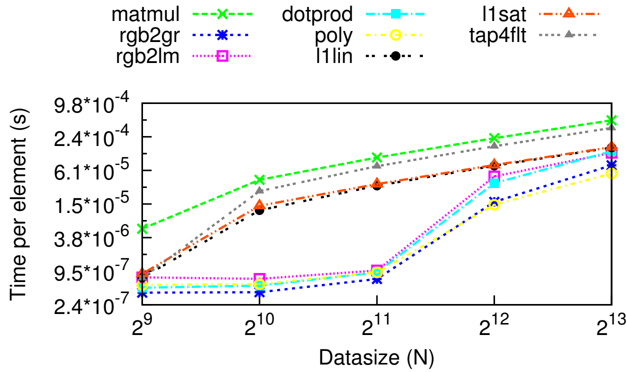

Figure 9: Impact of Total Element Count (16b data, 16-lane design)

Table IV: Power Consumption Measurements

Configuration	Power	Δ	(% over Idle)
Idle	5.1W	0	0
ARMv7 Scalar	5.9W	0.8W	15
NEON Vector	5.7W	0.6W	11
Idle+PROGUSB	5.5W	0	0
MXP Vector	5.6W	0.1W	2

We note peculiar scaling behavior for certain workloads where performance stays stable even at larger datasizes. These workloads are those that have high arithmetic intensity *i.e.* fewer IO load/stores and register requirements per vector computation. For workloads with larger IO and intermediate register needs, runtime is dominated by memory bottlenecks.

Power Usage: Finally, in Table IV, we show the overall power consumption of the ZedBoard used in our experiments. We note that the use of ARMv7 CPU or NEON for running our benchmarks increases power consumption from an idle consumption of 5.1W to 5.9W and 5.7W respectively (mean). This indicates an improvement of 0.2W when accelerating code on the NEON engine. When using MXP, the power consumption increases from an idle consumption of 5.5W (due to PROGUSB cable requirements) to 5.6W. This represents a modest increase of 0.1W in power usage when running code on the FPGA-based soft vector processor. Even in absolute terms, this is still a 0.1W improvement over NEON execution.

VI. Discussion

From our experiments we can draw the higher-level conclusion that soft vector processors such as the MXP architecture are capable of matching and exceeding the performance of hard vector processors such as NEON in FPGA-based heterogeneous embedded systems. For simple, data-parallel computations with low compute requirements (`poly`), vector performance scales with increasing lane counts. For computations with accumulation (`matmul`, `tap4flt`), we observe limited scalability due to the feedback loop. An optimized reduction primitive would be a useful addition to the vector architecture for such computations. As expected, 8b vectors permit larger lane counts and consequently higher speedups.

VII. Context

Many soft vector processors [3], [11], [14], [13], [7] and their building blocks have been reported in literature. Each of these studies represent increasingly refined designs of FPGA-based vector units for improved performance and reduced resource utilization. VESPA [13] introduced the idea of soft-processors that use VIRAM-like vector processing engines on FPGA fabrics while showing speedups over ordinary scalar soft processors. VIPERS [14] provided a NIOS host to the soft vector engine and demonstrated the importance of memory scaling on performance. The improved VEGAS [3] soft vector processor added local scratchpads and demonstrated comparable performance against Intel SSE SIMD execution for the integer `matmul` benchmark (≈2× faster). VENICE [11] is a resource-optimized VEGAS that shows 10–200× speedup against the NIOS-II/f processor embedded in soft logic on Altera FPGA platforms such as the DE2-115 and DE4 boards. BlueVec [10] is a Bluespec-based vector processor developed for applications that are constrained by the FPGA memory wall. While VIPERS, VESPA, VEGAS, VENICE and BlueVec operate on integer data, the vector architecture presented in [7] works with floating-point operands. For our experiments, we use the MXP [12] soft vector processor with supports integer and fixed-point calculations only. While existing vector architectures offer superior performance when using FPGA logic, the performance comparisons have always been against other soft processors with poor performance *e.g.* MicroBlaze, NIOS, other lightweight soft processors or 1-lane self-comparisons. In this study, we compare the MXP soft vector engines against the 667MHz hard silicon processor ARM Cortex A9 along with NEON SIMD vector engine that is of direct relevance in embedded system environments. Furthermore, the ARMv7 CPU and NEON engines offer substantially higher throughput over the NIOS, MicroBlaze and other soft processor used in earlier studies.

For a long time, soft vector architectures lacked a high-level programming model that can make it easy to target these platforms. While vectorizing compilers are not new, even `gcc` supports auto-vectorization, it is not trivial to adapt these existing backends to support newer, custom vector architectures easily. Hence, earlier versions of soft vector processors were programmed directly in low-level API calls to the instruction handling logic. The Accelerator [8] compiler showed how to auto-generate Venice-compatible code from high-level descriptions of parallel programs. The SCORE framework is a similar high-level programming environment for data-parallel stream computations. Unlike Accelerator, we develop an auto-tuning pass

that helps choose the best soft vector configuration (number of lanes, chunk size, computation-communication overlap) without programmer involvement. The custom vector compiler [4], extracts custom vector instructions to generate CVUs (custom vector units) based on frequently occurring patterns in data-parallel programs. Our emphasis, in this paper, is on a simpler RISC-like vector architecture observed in NEON and the MXP processor and an associated compiler framework to target these architectures.

VIII. Future Work

At present, we rely on `gcc` auto-vectorization but intend to investigate the potential of using `armcc` for generating better vector code that may close the gap with SCORE-generated code. As part of future work, we will also consider partitioning computation across all three backends (scalar, NEON SIMD and MXP soft vector) together. Furthermore, larger Zynq SoCs will permit MXP configurations larger than 16 lanes and a larger scratchpad. While MXP currently uses a single 64b AXI-HP lane, the switch to ACP and use of multiple AXI-HP ports will offer further improvements in DMA throughput. Furthermore, we can consider larger Zynq devices (than the Z7010 part on the ZedBoard) to support larger MXP lane counts and consequently higher speedups for vectorizable computations with low IO requirements.

IX. Conclusions

Heterogeneous FPGA-based SoCs like Xilinx Zynq offer a novel computing platform for embedded processing that allows us to unify a scalar ARMv7 core, hard vector NEON SIMD engines as well as the FPGA-based MXP soft vector processor in the same chip. Using our compiler and auto-tuning framework, we demonstrate speedups as high as 3.95$\times$ for offloaded data-parallel computation when comparing an MXP soft vector processor to optimized NEON mappings across a range of embedded data-parallel kernels. MXP is able to outperform NEON due to better customizability of vector lane counts, scratchpad size and memory transfer optimizations. In addition to performance improvements, we also measure $\approx$9% power savings when choosing FPGA-based vector acceleration over NEON. On larger FPGA-based SoCs with the already-available faster interconnect options, we expect MXP code to provide scalable performance when compared to the hard silicon NEON engines. We will be making the benchmark codes and the compiler frameworks open-source for community use.

References

[1] ARM Ltd. Cortex-A9 NEON Media Processing Engine. pages 1–49, July 2011.

[2] E. Caspi, M. Chu, R. Huang, J. Yeh, J. Wawrzynek, and A. DeHon. Stream computations organized for reconfigurable execution (SCORE). *Field-Programmable Logic and Applications: The Roadmap to Reconfigurable Computing*, 2000.

[3] C. H. Chou, A. Severance, A. D. Brant, Z. Liu, S. Sant, and G. G. Lemieux. VEGAS: Soft vector processor with scratchpad memory. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pages 15–24. ACM, 2011.

[4] J. Cong, M. A. Ghodrat, M. Gill, H. Huang, B. Liu, R. Prabhakar, G. Reinman, and M. Vitanza. Compilation and architecture support for customized vector instruction extension. In *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific*, pages 652–657. IEEE, 2012.

[5] Y. Deheng and N. Kapre. MixFX-SCORE: Heterogeneous Fixed-Point Compilation of Dataflow Computations. pages 1–4, Mar. 2014.

[6] N. Kapre and A. DeHon. VLIW-SCORE: Beyond C for Sequential Control of SPICE FPGA Acceleration. In *Field-Programmable Technology (FPT), 2011 International Conference on*, pages 1–9, Dec. 2011.

[7] J. Kathiara and M. Leeser. An Autonomous Vector/Scalar Floating Point Coprocessor for FPGAs. In *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, pages 33–36. IEEE, 2011.

[8] Z. Liu, A. Severance, S. Singh, and G. G. Lemieux. Accelerator compiler for the venice vector processor. In *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, pages 229–232. ACM, 2012.

[9] H. Martorell and N. Kapre. FX-SCORE: A Framework for Fixed-Point Compilation of SPICE Device Models using Gappa++. In *IEEE International Symposium on Field-Programmable Custom Computing Machines*, pages 77–84, Mar. 2012.

[10] M. Naylor, P. J. Fox, A. T. Markettos, and S. W. Moore. Managing the FPGA memory wall: Custom computing or vector processing? In *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, pages 1–6. IEEE, 2013.

[11] A. Severance and G. Lemieux. VENICE: A compact vector processor for FPGA applications. In *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, pages 245–245. IEEE, 2012.

[12] A. Severance and G. G. Lemieux. Embedded supercomputing in FPGAs with the VectorBlox MXP matrix processor. In *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2013 International Conference on*, pages 1–10. IEEE, 2013.

[13] P. Yiannacouras, J. G. Steffan, and J. Rose. VESPA: portable, scalable, and flexible FPGA-based vector processors. In *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, pages 61–70. ACM, 2008.

[14] J. Yu, C. Eagleston, C. H.-Y. Chou, M. Perreault, and G. Lemieux. Vector processing as a soft processor accelerator. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 2(2):12, 2009.