Optimizing Soft Vector Processing in FPGA-based Embedded Systems

NACHIKET KAPRE, Nanyang Technological University

Soft vector processors can augment and extend the capability of FPGA-based embedded SoCs such as the Xilinx Zynq. However, configuring and optimizing the soft processor for best performance is hard. We must consider architectural parameters such as precision, vector lane count, vector length, chunk size, DMA scheduling to ensure efficient execution of code on the soft vector processing platform. To simplify the design process, we develop a compiler framework and an auto-tuning runtime that splits the optimization into a combination of static and dynamic passes that map data-parallel computations to the soft processor. We compare and contrast implementations running on the scalar ARM processor, the embedded NEON hard vector engine, low-level streaming Verilog designs with the VectorBlox MXP soft vector processor. Across a range of data-parallel benchmarks, we show that the MXP soft vector processor can outperform other organizations by up to 4× while saving ≈10% dynamic power. 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. When constrained by IO bandwidth, soft vector processors are even competitive with spatial Verilog implementations of computation.

Categories and Subject Descriptors: C.1.2 [**Processor Architectures**]: Array and vector processors

General Terms: Vector Processors, Embedded Systems, Compilers

Additional Key Words and Phrases: vector processors, soft processors, streaming computations

ACM Reference Format:
Nachiket Kapre, 2014. Optimizing Soft Vector Processing in FPGA-Based Embedded Systems *ACM Trans. Reconfig. Technol. Syst.* 9, 4, Article 39 (March 2015), 17 pages.
DOI : http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Traditionally, embedded systems have been designed to be low-cost, low-power implementations with limited silicon resource capacities. They are usually organized around cheap central coordination processors based on ARM or MIPS-based ISAs. These embedded processors typically run a real-time host OS or firmware that must accommodate these scarce resource constraints. In contrast, modern embedded SoCs (systems-on-chip) with higher expectations of performance integrate a heterogeneous assembly of μarchitectures and co-processors within the same chip. Depending on system requirements, today we can choose amongst 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). Embedded software developers now have unparalleled compute capacity at their disposal in small embedded form factors (*e.g.* Raspberry Pi, Beaglebone Black, Jetson TK1) that are capable enough to run complete modern OSses. While this diversity brings greater freedom and versatility, developers that use these embedded platforms are expected to manually decide how to assign and optimize computational kernels for these heterogeneous SoC blocks.

Author's addresses: N. Kapre, School of Computer Engineering, Nanyang Technological University, 50 Nanyang Avenue, Singapore 639798. Email: nachiket@ieee.org

DOI : http://dx.doi.org/10.1145/0000000.0000000

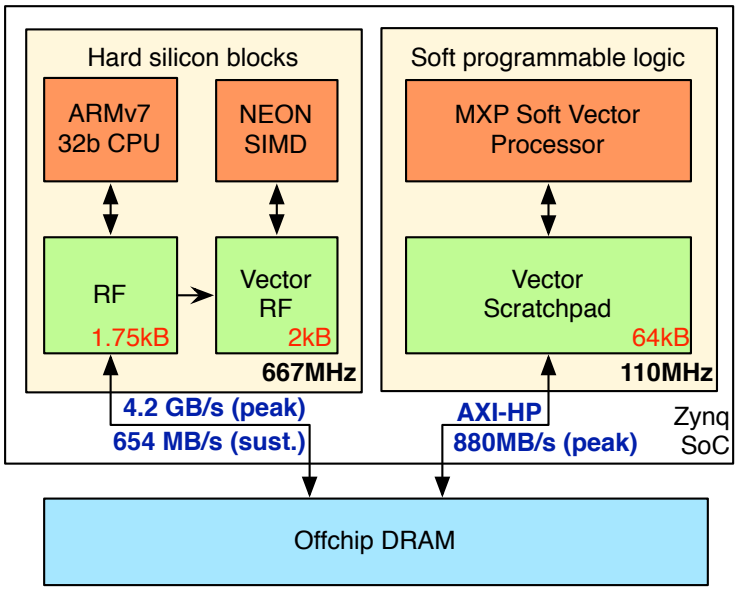

Fig. 1: Xilinx Zynq SoC Platform Compute Organization

Vector processors are capable of exploiting data-parallelism in the computation by organizing hardware into a tightly-coupled array of compute units that all execute the same instruction in the same cycle. They are popular in embedded application domains such as signal processing, multimedia, and control systems. On one hand, modern ARM processors already provide hard vector engines in the form on the NEON SIMD unit that executes the NEON instruction set. On the other hand, a soft vector processor can be programmed on top of reconfigurable FPGA logic to implement any user-supplied vector instruction set. In this paper, we show how to manage parallelism amenable to vectorization and quantify the performance and power improvements possible when using the MXP soft vector processor on the Xilinx Zynq ZC7020 SoC. The Xilinx ZedBoard platform, shown in Figure 1, is one such Zynq SoC-based system that includes the ARM Cortex A9-series CPU as the central co-ordination processor. This is augmented with the NEON SIMD hard vector engine under the same memory hierarchy as well as a tightly-coupled FPGA logic substrate connected over a low-latency, high-bandwidth AXI (ARM's Advanced Extension Interface) and ACP (Accelerator Coherency Port) 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 of many existing embedded applications through a recompile of the code. However, if low power utilization and absolute performance is desired, the FPGA logic may offer a superior alternative. To retain the fast development cycles and performance advantages of mapping to NEON, we consider the VectorBlox MXP [Severance and Lemieux 2013] soft vector processor as the alternative implementation. MXP is a fully-customizable 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, asynchronous 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 efficiently program the soft vector processor. Our framework also generates optimized NEON code using auto-vectorization and explicit generation of NEON intrinsics along with low-level Verilog datapaths of the data-parallel streaming

computation. Using our framework, we identify the conditions under which the MXP soft vector processor offers superior performance and power benefits compared to NEON hard vector engines or streaming FPGA logic. We also show how to automate the configuration and customization of the soft vector processor across a range of varying benchmarks with differing structural properties.

```
// simple for loop
for(int i=0;i<N;i++) {
        // computing ax^2 + bx + c
        y[i] = a*(x[i])*(x[i]) + b*x[i] + c;
}
```

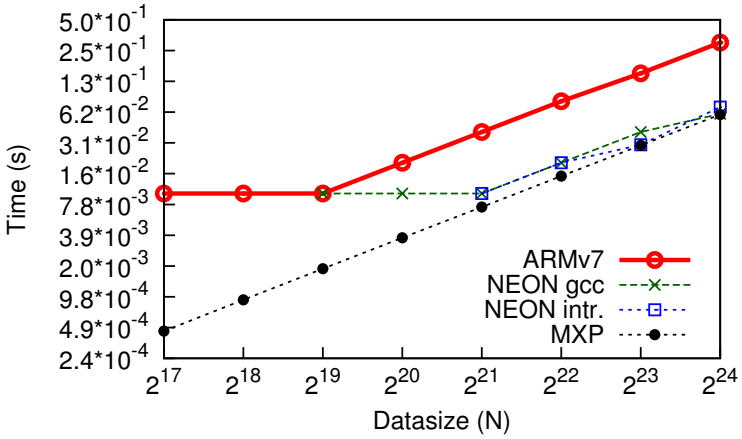

Fig. 2: Comparing runtime for $a \cdot x^2 + b \cdot x + c$

Consider the simple example $a \cdot x^2 + b \cdot x + c$ implemented inside a data-parallel for loop operating on 8-bit data as shown in Figure 2. When the loop trip count is too small, 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 ($>\approx$0.5M with uncached data), we can exploit automatic gcc auto-vectorization for NEON (or when using NEON intrinsics) for achieving a 3$\times$ speedup. With NEON intrinsics we can achieve slightly larger speedups in some instances compared to auto-vectorization due to unreliable vector code generation backend in `gcc`. Finally, the MXP soft vector processor delivers faster runtime at virtually all trip counts ($\approx$1.3$\times$ 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.

In Section 2, we describe the technical specifications of the Zynq platform and introduce the SCORE compiler framework. We then describe our new vector compiler backend and auto-tuning framework in Section 3. Our methodology is explained in Section 4 which is used to generate the results shown in Section 5. We discuss the implications of our experiments in Section 6 and the

Table I: Architecture Specifications

Metric	ARMv7 CPU	NEON	MXP	Ratio
μarch	Scalar	SIMD	Soft vector	
Clock Freq.	667 MHz	667 MHz	110 MHz	$1/6\times$
Lanes	1	2×32b	1–16×32b	8×
		4×16b	2–32×16b	8×
		8×8b	4–64×8b	8×
Throughput				
32b (Gops/s)	0.6	1.3	1.7	1.3×
16b (Gops/s)	0.6	2.6	3.5	1.3×
8b (Gops/s)	0.6	5.3	7	1.3×
Memory	1.75kB (Scalar RF)	2kB (Vector RF)	4–256kB (Scratchpad)	2–128×

academic context in Section 7. We outline ideas for future work in Section 8 followed by deriving overall conclusions in Section 9.

2. BACKGROUND

2.1. Zynq SoC Platform

In the past, FPGA developers could only rely on soft processors such as the Xilinx MicroBlaze or Altera Nios-II/f if they needed software-like co-ordination and control at their disposal. We now have SoC-based FPGA platforms such as the Xilinx Zynq that embed hard ARMv7 processors directly integrated with the FPGA fabric. We first study the datasheet specifications of the various computing blocks available on the Zynq SoC ZC7020 available on the Xilinx ZedBoard platform as shown in Table I. We list 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. When comparing the peak operating frequency, we observe that dedicated 28nm silicon (*i.e. hard*) implementations of ARMv7 CPU and the NEON SIMD engines runs at 667 MHz (speed grade -1). The MXP soft vector processor programmed on top of the FPGA logic can only achieve a peak frequency of 110 MHz, a gap of almost 6× when compared to the hard silicon fabric. While this seems disappointing, when we estimate the peak compute throughput on 8b data, we note that the 64-lane MXP processor (limited by FPGA device capacity) delivers a throughput of 7 Gops/s while the NEON units saturate at 5.3 Gops/s with the ARMv7 CPU lagging behind with a mere 0.667 Gops/s. These are peak throughput numbers that are rarely achieved in real-world designs. However, they do offer a perspective into the raw capabilities of the architectures.

When mapping data-parallel computation to the vector engines, performance will depend significantly on the capacity and bandwidth of the register files (or memories) associated with the compute units. For NEON, 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 Zynq chip 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. While the MXP can scale up to 128 lanes, the Zynq device on the ZedBoard limits us to 16 lanes at 32b per lane.

Alternatively, we can directly map streaming data-parallel code to the FPGA logic written in low-level Verilog. For this design option, the FPGA fabric offers a promising 560 kB of onchip RAM, 800 MB/s of AXI bandwidth to the DRAM and 44 Gops/s (16b DSP throughput at 200 MHz). As

Table II: Vector Architecture Parameter and Ranges

Parameter	NEON		MXP	
	Lo	Hi	Lo	Hi
Vector Lane (L)	2	8	1	16
Vector Width (W)	8b	32b	8b	32b
Memory Size (S)	-	-	4kB	256kB
Total Elements (N)	2^1	2^{14}	2^1	2^{14}

we observe later, even though the raw Verilog compute capacity looks superior, we rarely achieve it in practice when DRAM bandwidth limits performance.

2.2. Programming the Zynq Platform

For the heterogeneous Zynq SoC, programming the ARMv7 CPU and the NEON SIMD unit is 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.

Alternatively, we can program the FPGA portion of the SoC fabric using Verilog, VHDL or high-level synthesis flavored C/C++ code. This is supported through a more tedious compilation flow using the Xilinx Vivado CAD flow that typically takes 20 minutes to an hour to generate the executable FPGA bitstream.

The MXP soft vector processor is programmed with a somewhat simpler flow that requires generating (1) the soft vector engine based on static specifications such as vector lane counts, and scratchpad capacity, and (2) the assembly code that exercises the vector engines for the desired application.

In all these cases, generating optimized high-performance code requires a thorough understanding of the capabilities of the hardware and limitations of the existing software compilation tools. For example, the `gcc` 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. Verilog and VHDL-based flows must carefully schedule the computation into high-performance pipeline stages and perform a design space exploration to evaluate multiple implementation choices.

The programming challenge for an automated framework in this context is to optimize data-parallel code to tune architecture parameters directly without user intervention. To address these challenges, we select the open-source SCORE [Caspi et al. 2000] 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 variety of other backends such as sequential control [Kapre and DeHon 2011], and fixed-point circuit generation [Martorell and Kapre 2012; Ye and Kapre 2014]. In this paper, we adapt the compiler to support two extra backends for the NEON SIMD engine and the MXP soft processor.

3. VECTORIZING COMPILER AND AUTO-TUNING FRAMEWORK

When optimizing code for embedded platforms to achieve desired performance and power characteristics, system developers will usually rely on manual partitioning, mapping, and tuning heuristics. As discussed earlier in Section 1, the advent of heterogeneous compute elements further adds to developer burden. In Figure 3, we show a block diagram of our compiler and runtime system for Zynq platforms. The goal of our compiler framework is to generate optimized code for the multiple back-

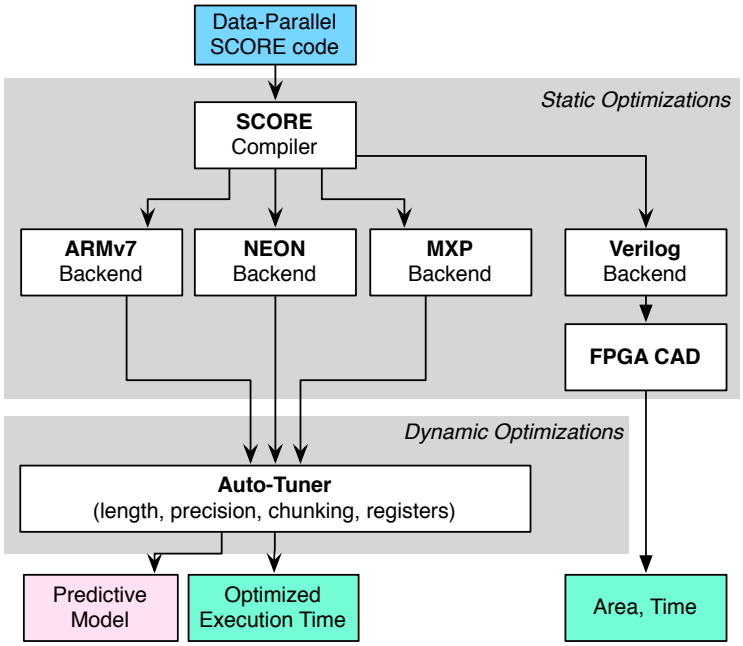

Fig. 3: Block Diagram of the Compiler and Runtime Framework

end targets on the Zynq platform such as scalar CPU, NEON SIMD unit, MXP soft vector engine as well as low-level Verilog. Our framework allows the developer to describe their data-parallel kernels at a high level of abstraction without requiring manual porting effort. Computation is described as data-parallel functions on stream inputs to generate stream outputs. Our compiler framework translates these streaming computations into vectorized implementations for mapping to NEON SIMD and MXP processors. To produce straight-line C code, we simply rewrite the dataflow expressions using a source-to-source translator. Verilog generation is more complex, we automatically generate pipelined datapaths as well as infer appropriate handshake controls for the stream inputs and outputs. The vector backend is the new addition to the SCORE compilation framework. The vector backend includes vector-friendly optimizations such as straightforward register reuse and vector strip mining. While the compiler performs static code optimizations, we rely on our auto-tuner to perform dynamic code optimization at runtime. This is necessary as certain application characteristics are closely linked to physical system execution conditions and data dependent. The goal of the auto-tuning step is to (1) explore and select 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. We tabulate the space of possible configurations in Table II. From these experimental results we identify the set of optimized parameters for our computation that is then used during the execution lifetime. Furthermore, we develop a predictive model that can guide the design space exploration process without invoking the full compilation and auto-tuning steps. The model acts as a recommendation engine for other data-parallel problems based on simple input characteristics (see Table V).

In Table III, we see the input SCORE program and the generated Verilog 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. The polynomial calculation is wrapped inside a `state` to guard the evaluation based on presence of tokens on the input x. In this model, SCORE operates in an actor-oriented manner where actions by the actor are triggered when input conditions are satisfied. As SCORE is

(a) SCORE input

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

(b) Verilog code

```
module poly(
input clk, rst,
input bool[7:0] x, a, b, c
output bool[7:0] y)
begin
  always@(posedge clk)
  if(rst)
    y <= 8'b0;
  else
    y <= a*x*x
         + b*x
         + c;
  end;
endmodule;
```

Table III: Code generation in SCORE for Verilog backend

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. This is possible by splitting the code generation process into two components (1) datapath (shown in the snippet), and (2) controller (not shown for simplicity). The controller manages the handshaking on the inputs and outputs to determine when to fire the operator. The datapath simply performs the arithmetic operations of the polynomial in a pipelined fashion. Since we support simple streaming interfaces, we can directly connect the IOs to the ACP interface (through Xillybus FIFOs). For most designs, as we will see later, performance is throttled by achievable ACP performance

(c) ARMv7 CPU

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

(d) NEON

```
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);
```

(e) MXP

```
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 IV: Scalar and Vector Code Generation

While SCORE is an FPGA-programming framework, ultimately it captures parallelism in the input problem. For vectorizable computations, we are interested in data-parallel forms of parallel expression. Hence, we consider a subset of SCORE that is described as straightforward, data-parallel computations operating on a single state with potentially multi-cycle operation and some internal registers updated in a reduction (accumulation) manner. This simplification enables vectorizable code generation for various backends as shown in Table IV. 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

Table V: 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

at specific bitwidth and lane-count 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 as shown in Figure IV. 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 for simplicity).

4. METHODOLOGY

We develop an experimental methodology to compare the different vector backends and architectures based on a range of benchmark problems. In Table V 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. Benchmarks with high IO requirements, we expect the performance to be dominated by memory bandwidth limits, while those with high use of arithmetic operations can exploit the ALU throughput of the vector engines (as we will see later in Figure 10 in particular).

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 Verilog compilation, we generate interface wrappers for streaming IO reusing code from [Vipin et al. 2013]. For the streaming Verilog operators we assume a buffer size of 32 elements per IO channel.

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. When measuring Verilog performance, we record area and delay of each implementation and consider

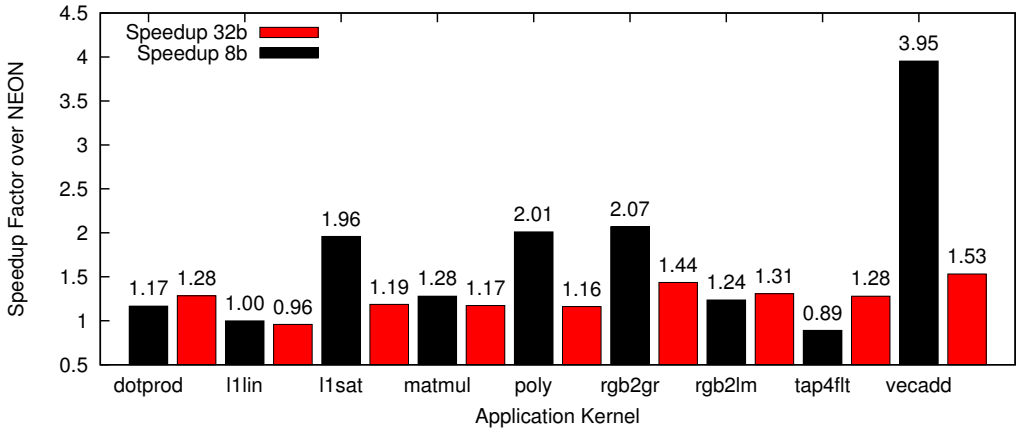

Fig. 4: Speedups of MXP over best NEON implementation

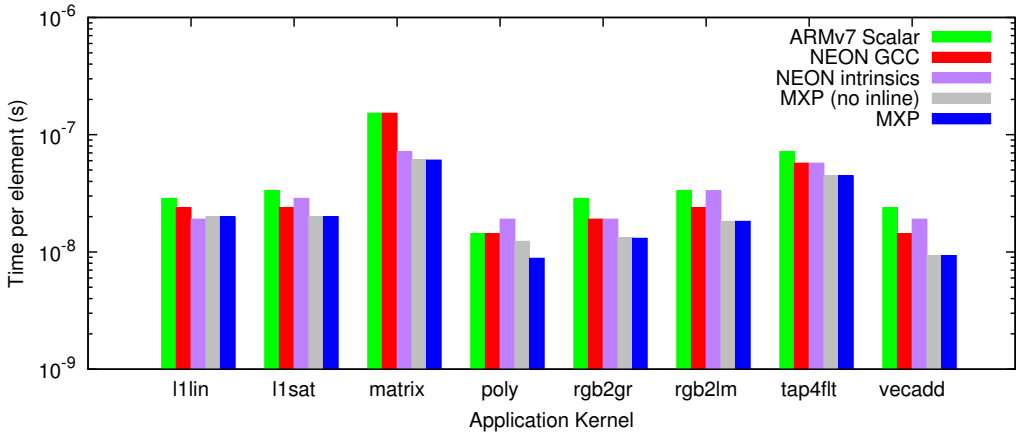

Fig. 5: Comparing runtime of the different application kernels

replicated datapaths that saturate the FPGA fabric capacity to bound the upper limit in achievable performance. In practice, the IO bandwidths limit achievable performance for the constrained embedded system.

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 Xillinux 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 Xillinux which rose to 5.5W with PROGUSB cable connected. We calculate runtime power utilization from these two baselines for the experiment as appropriate.

5. RESULTS

Speedups and Runtime Comparisons: We first compare speedups achieved by the 16-lane MXP soft processor when compared to optimized NEON vector runtimes in Figure 4. We observe that in most cases that MXP outperforms NEON implementations by as much as 3.95× at 8b computations

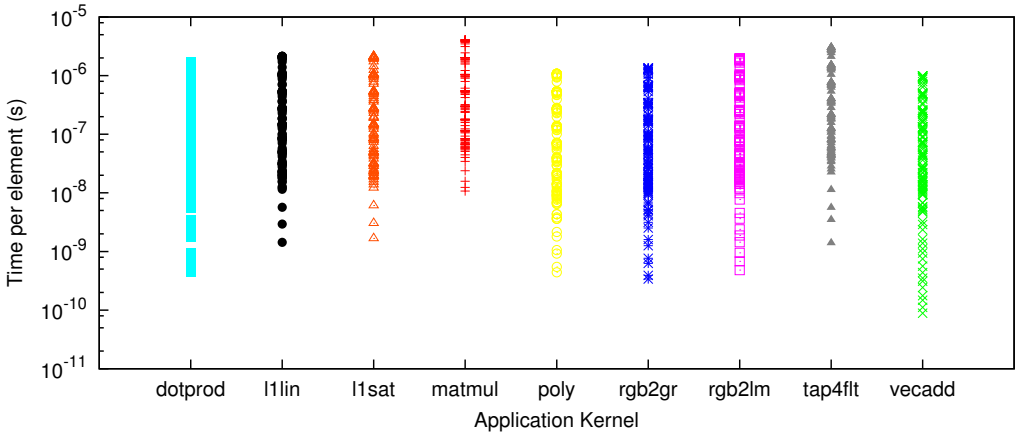

Fig. 6: MXP Auto-Tuning Dynamic Range

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 V).

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 5, 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 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 6. 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 in the form of more scratchpad memory and logic resource usage. However, for our experiments, we are already tightly constrained by the ZedBoard FPGA capacity limiting exploration beyond the 64-lane 64kB scratchpad design. Being a hard vector architecture with few programmable elements, the auto-tuning range is much lower for the NEON as shown in Figure 7. 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 evaluated preferred a lane count of 8 or 16 with sufficient parallel work. As shown

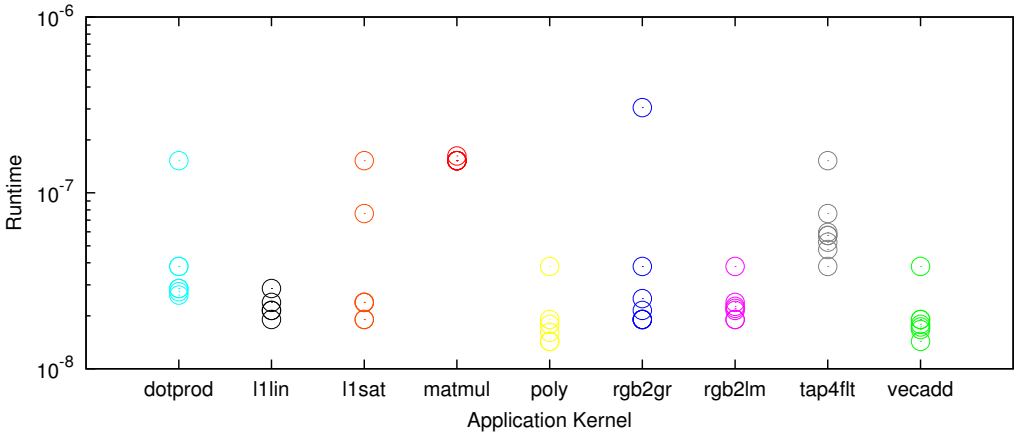

Fig. 7: NEON Auto-Tuning Dynamic Range

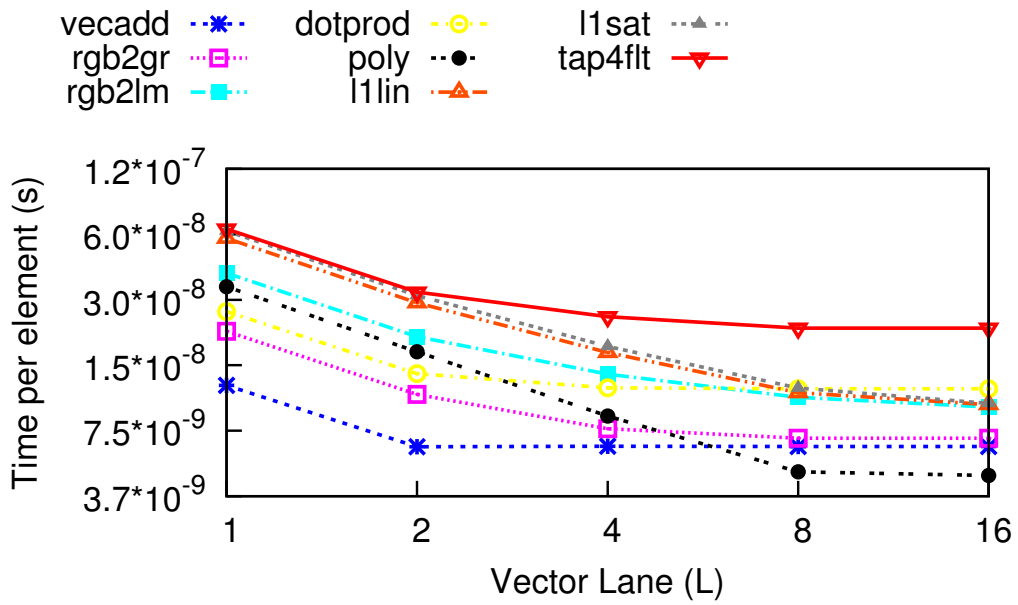

Fig. 8: Tuning Lane Width for the MXP soft-processor
(16b data and VL of 1024)

in Figure 8, our auto-tuner identifies `poly` benchmark as 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 9, we show the impact

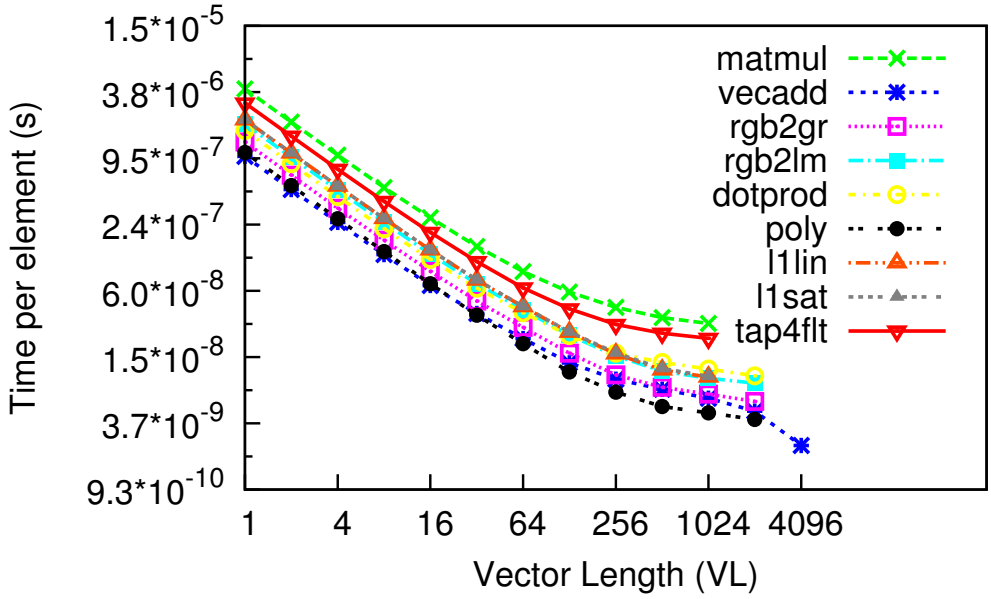

Fig. 9: Tuning Vector Length for MXP
(16b data and 16-lane design)

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 V). 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 10, we observe the impact of overall trip counts on performance. 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.

Streaming Verilog Implementation: In Figure 11, we compare the theoretical Verilog performance with ACP-limited throughputs. The FPGA utilization and performance characteristics of the Verilog implementations is shown in Table VI. These are based on the Verilog files generated by the SCORE Verilog backend with AXI-Stream interfaces. As we can see, while the raw Verilog datapaths can operate quite fast, this is rendered ineffective when limited by the IO bandwidths. In such circumstances, when performance is constrained by IO throughputs, a vector processor offers a more productive development flow with no compromise in overall performance. For larger FPGAs, with faster interfaces and larger internal buffering, we can accommodate bigger signal processing graphs directly within the FPGA fabric. When FPGA size and cost are not a constraint, a streaming Verilog approach is likely to deliver highest performance. However, development times for the streaming Verilog approach (FPGA place-and-route flow) will be much longer and more tedious than the programming approach based on soft vector processing.

Power Usage: Finally, in Table VII, 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 configuring the MXP vector processor over USB, the power consumption increases from an idle consumption of 5.5W (due to PROGUSB cable requirements) to 5.6W. This represents a modest

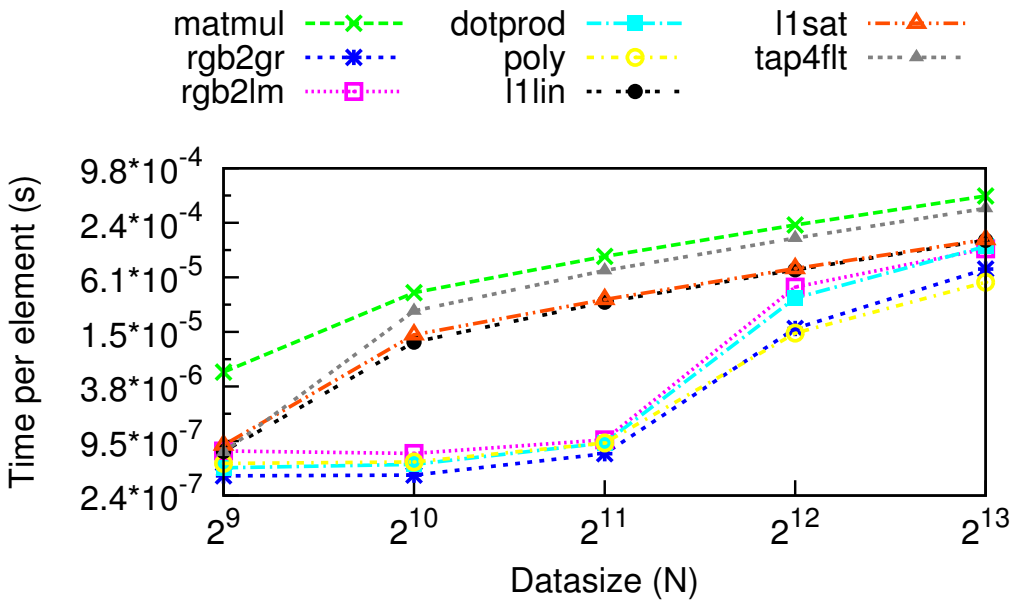

Fig. 10: Impact of Total Element Count
(16b data, 16-lane design)

Benchmark	SLICEs	DSPs	IOs	Clock (MHz)	Parallel Instances (limited by) Logic	DSP	IO
dotprod	111	6	5	81	479	36	3
l1lin	88	5	4	57	604	44	2
l1sat	115	5	4	52	462	44	2
matmul	256	24	12	84	207	9	1
poly	52	7	3	56	1023	31	3
rgb2gr	58	4	3	105	917	55	6
rgb2lm	88	6	4	79	604	36	3
tap4flt	199	12	9	68	267	18	1
vecadd	45	0	3	282	1182	Inf	18

Table VI: SCORE Verilog backend characteristics

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. In contrast, we also setup the MXP configuration flow to boot from an SDCARD thereby lowering idle power consumption but no effect was observed on MXP power draw. Due to the portable nature of the MXP soft vector processor, we also investigated the power utilization on the DE2-115 platform. In this case, we see slightly higher power numbers due the peripherals connected to the FPGA, but the active power when running the MXP is similar to the ZedBoard at 0.7W above Idle.

6. 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

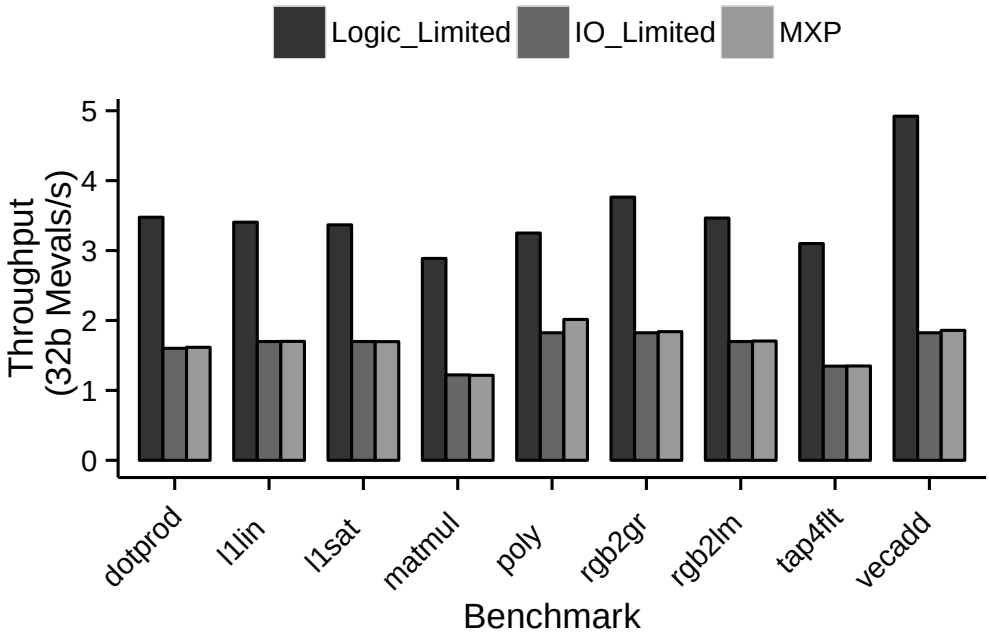

Fig. 11: Comparing Throughputs of Verilog-based Implementation with MXP

Table VII: 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
Idle+SDCARD	5.1W	0	0
MXP Vector	5.6W	0.5W	10
DE2-115	6W	0	0
DE2-115-NIOS2	6.2W	0.2W	3
DE2-115-MXP	6.9W	0.7W	11

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. Furthermore, for spatial Verilog implementations of the data-parallel computations, we are performance limited by IO bandwidth. In these conditions, the MXP implementation matches the achievable performance of the spatial designs.

7. CONTEXT

Many soft vector processors [Chou et al. 2011; Severance and Lemieux 2012; Yu et al. 2009; Yiannacouras et al. 2008; Kathiara and Leeser 2011] 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 [Yiannacouras et al. 2008] 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 [Yu et al. 2009] provided a Nios host to the soft vector engine and demonstrated the importance of memory scaling on performance. The improved VEGAS [Chou et al. 2011] soft vector processor added local scratchpads and demonstrated comparable performance against Intel SSE SIMD execution for the integer `matmul` benchmark ($\approx 2\times$ faster). VENICE [Severance and Lemieux 2012] is a resource-optimized VEGAS that shows 10–200$\times$ speedup against the Nios-II/f processor embedded in soft logic on Altera FPGA platforms such as the DE2-115 and DE4 boards. BlueVec [Naylor et al. 2013] 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 [Kathiara and Leeser 2011] works with floating-point operands. For our experiments, we use the MXP [Severance and Lemieux 2013] soft vector processor with support for 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 [Liu et al. 2012] 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 [Cong et al. 2012], 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.

8. 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 Z7020 part on the ZedBoard) to support larger MXP lane counts and consequently higher speedups for vectorizable computations with low IO requirements. The MXP processor is even compatible with the Altera

DE4-230 FPGA board and can deliver 3–4× better performance that the Zedboard MXP design due to a 2× increase in clock frequency, 2× more lanes, and 2× higher scratchpad capacity.

9. 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 autotuning framework, we demonstrate speedups as high as 4× 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 ≈10% dynamic 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. Under IO bandwidth constrained operation, the MXP implementation of computation can match the performance of spatial Verilog implementations. The SCORE compiler framework is available for open-source community use from github.com/nachiket/tdfc as discussed earlier version of this article [Jie and Kapre 2014].

REFERENCES

E Caspi, M Chu, R Huang, J Yeh, J Wawrzynek, and A DeHon. 2000. Stream computations organized for reconfigurable execution (SCORE). *Field-Programmable Logic and Applications: The Roadmap to Reconfigurable Computing* (2000), 605–614.

Christopher H Chou, Aaron Severance, Alex D Brant, Zhiduo Liu, Saurabh Sant, and Guy GF Lemieux. 2011. VEGAS: Soft vector processor with scratchpad memory. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*. ACM, 15–24.

Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Hui Huang, Bin Liu, Raghu Prabhakar, Glenn Reinman, and Marco Vitanza. 2012. Compilation and architecture support for customized vector instruction extension. In *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific*. IEEE, 652–657.

Soh Jun Jie and Nachiket Kapre. 2014. Comparing soft and hard vector processing in FPGA-based embedded systems. In *24th International Conference on Field Programmable Logic and Applications, FPL 2014, Munich, Germany, 2-4 September, 2014*. 1–7. DOI: http://dx.doi.org/10.1109/FPL.2014.6927467

Nachiket Kapre and Andre DeHon. 2011. VLIW-SCORE: Beyond C for Sequential Control of SPICE FPGA Acceleration. In *Field-Programmable Technology (FPT), 2011 International Conference on*. 1–9.

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

Zhiduo Liu, Aaron Severance, Satnam Singh, and Guy GF Lemieux. 2012. Accelerator compiler for the venice vector processor. In *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*. ACM, 229–232.

Helene Martorell and Nachiket Kapre. 2012. FX-SCORE: A Framework for Fixed-Point Compilation of SPICE Device Models using Gappa++. In *IEEE International Symposium on Field-Programmable Custom Computing Machines*. 77–84.

Matthew Naylor, Paul J Fox, A Theodore Markettos, and Simon W Moore. 2013. Managing the FPGA memory wall: Custom computing or vector processing?. In *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*. IEEE, 1–6.

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

Aaron Severance and Guy GF Lemieux. 2013. Embedded supercomputing in FPGAs with the VectorBlox MXP Matrix Processor. In *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2013 International Conference on*. IEEE, 1–10.

K Vipin, S Shreejith, D Gunasekera, S A Fahmy, and Nachiket Kapre. 2013. System-level FPGA device driver with high-level synthesis support. In *Field-Programmable Technology (FPT), 2013 International Conference on*. 128–135.

Deheng Ye and Nachiket Kapre. 2014. MixFX-SCORE: Heterogeneous Fixed-Point Compilation of Dataflow Computations. *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on* (2014), 206–209.

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

Jason Yu, Christopher Eagleston, Christopher Han-Yu Chou, Maxime Perreault, and Guy Lemieux. 2009. Vector processing as a soft processor accelerator. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 2, 2 (2009), 12.