A Hardware Accelerator for Computing an Exact Dot Product

Jack Koenig, David Biancolin, Jonathan Bachrach, and Krste Asanović
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
Berkeley, USA
{jack.koenig3, biancolin, jrb, krste}@eecs.berkeley.edu

***Abstract*—We study the implementation of a hardware accelerator that computes a dot product of IEEE-754 floating-point numbers exactly. The accelerator uses a wide (640 or 4288 bits for single or double-precision respectively) fixed-point representation into which intermediate floating-point products are accumulated. We designed the accelerator as a generator in Chisel, which can synthesize various configurations of the accelerator that make different area-performance trade-offs.**

We integrated eight different configurations into an SoC comprised of RISC-V in-order scalar core, split L1 instruction and data caches, and unified L2 cache. In a TSMC 45 nm technology, the accelerator area ranges from 0.05 mm^2 to 0.32 mm^2, and all configurations could be clocked at frequencies in excess of 900 MHz. The accelerator successfully saturates the SoC's memory system, achieving the same per-element efficiency (1 cycle-per-element) as Intel MKL running on an x86 machine with a similar cache configuration.

I. Introduction

Floating-point arithmetic is a mainstay of modern computing, yet lay programmers are often unaware of the numerical issues caused by the non-associativity of addition and multiplication. The trend towards multicore CPUs over the last decade has only exacerbated these concerns, as parallelization of a floating-point kernel over multiple threads of execution can easily introduce non-deterministic reorderings of intermediate operations. While sequential floating-point code can sometimes also produces unexpected results, these results can be easily reproduced to help debugging, whereas problematic execution interleavings in parallel codes can be difficult to capture, let alone reproduce.

Ideally, for a given sequence of floating-point operations, a computer would return the exact result, with no loss of information in the intermediate calculations. An exact result is reproducible by definition, as any valid interleaving of the operations would necessarily produce the same outcome. One could guarantee exactness by using an arbitrary-precision arithmetic library like GNU Multiple Precision Floating-Point Reliably (MPFR) [1]. Unfortunately, when configured to retain full precision, MPFR is two to three orders of magnitude slower than performing the same operations at native precision. For applications that are more compute-intensive, one could use Exact BLAS (ExBLAS) [2], a parallel and exact implementation of the BLAS algorithms [3]. However, here too there is an appreciable performance loss over computing a conventional dot product [4].

In many cases, the programmer is content with a result that is inexact but reproducible. Sequential code produces the same result on every invocation since operations are always performed in the same order, but to exploit the resources provided by modern multicore CPUs, multithreaded code is desired. Relaxing the exactness constraint enables parallel reproducible software libraries, like ReproBLAS [5], that are nearly as fast as their high-performance, irreproducible counterparts.

If, however, both exactness and high-performance are required, hardware acceleration becomes an attractive alternative. Moore's law has given computer architects a wealth of transistors with which to build increasingly diverse and powerful arithmetic units. Naïvely, one could perform floating-point operations in the equivalent fixed-point arithmetic. The fixed-point representation of an IEEE-754 double requires 2100 bits [6]. Performing *arbitrary* arithmetic on such large values is intractable. For example, successive multiplications result in exponential growth of the number of bits required to represent the product.

Nevertheless, use of fixed-point arithmetic as a stand-in for floating point may be feasible in hardware when: first, a bound can be placed on the widest necessary fixed-point representation; and second, operations on wide-fixed point representations can be restricted. This is the case for dot products and scalar sums. Given the ubiquity of the dot product across a wide range of applications, it may be expedient, especially in application-specific domains, to have hardware acceleration of an exact dot product. Exact dot products may be computed by performing unrounded floating-point multiplications and accumulating the result into a single fixed-point representation called a complete register [7].

In this paper, we conduct a design-space exploration of hardware accelerators for computing exact dot products of floating-point numbers using a wide fixed-point accumulator. The designs studied herein are based on those first proposed by Kulisch et al. [7] and implemented in the 1990s as off-chip coprocessors. Recent work explored FPGA implementations of the accelerator in isolation [8]. We build on the prior work by assessing the cost of similar accelerators when they have been integrated on die and are closely coupled to the processor core. We study single and double-precision implementations of two different accelerator microarchitectures, each of which can be attached to various levels of the cache hierarchy. To

measure area and delay, we implemented the design in a TSMC 45 nm technology. Finally, we contrast the performance of the complete SoCs with the performance of Intel MKL and ReproBLAS running on an Intel Xeon with a similar cache organization.

II. Background

The idea of an exact sum originates in mechanical calculators. In addition to the four elementary operations (addition, subtraction, multiplication, and division), calculators often had a fifth operation: "running total." The result register was much wider than the input registers, allowing for accumulation without loss of precision.

A. Motivating the Need for an Exact Dot Product

Dot product is a kernel of many applications that would benefit from greater precision. One could simply accumulate into a wider precision—using a quadruple precision accumulator for double precision dot product, for instance—there are still cases where this approach may not be sufficient. For example, computers solve large systems of linear equations by iterative refinement. In calculating the residual, cancellation in floating-point arithmetic can lead to difficult error analysis and slow down or even *prevent* convergence. An exact dot product precludes these issues, and can even speed up convergence compared to inexact arithmetic. For more detailed discussion of these problems, see [7], [9].

Another motivation for exact dot product is the general importance and cost of error analysis in numerical applications. Accumulating into higher precision (like quad) can simplify this analysis, but not remove it. An *exact* dot product provides a primitive that simplifies error analysis because it cannot introduce any error.

B. Previous Exact Dot Product Hardware

In the early 1990s, Kulisch et al. created a vector arithmetic coprocessor, called the XPA 3233, to compute exact dot products [7]. The XPA 3233 consisted of 207,000 transistors and was connected to a PC through a PCI bus. The long latency of the PCI bus greatly limited the usefulness of the coprocessor.

III. Exact Dot Product Accelerator Design

Although modern CPUs are comprised of billions of transistors, not all of them can be switching simultaneously at maximum frequency without exceeding practical power densities. Specialization, in the form of hardware accelerators, improves performance under these constraints by using less area and less energy to perform the same computation [10].

Our accelerator uses a modest amount of area to accelerate a common arithmetic task at a precision existing FPUs cannot provide natively. The accelerator avoids the instruction fetching and intermediate operations required by an exact or reproducible software library while fully utilizing available cache bandwidth.

A. Principle of Operation

The basis of our accelerator is a fixed-point representation of the entire space produced by the product of two floating-point numbers. The product of two floating-points numbers with the representation $(-1)^s \times m \times 2^e$ requires $1+2\times(2^{e_{bits}}+m_{bits})$ bits to be represented exactly, where e_{bits} and m_{bits} are the number of bits in the exponent and the mantissa respectively. To prevent overflow from a long accumulation, we add an additional k bits, where k is sufficiently large such that the machine would fail before overflow could occur from accumulation. We calculate the size of the complete register as follows:

- For IEEE Double Precision: $m_{bits} = 53$, $e_{bits} = 2047$, $k = 92$: $CR_{bits} = 4288$.
  (67×64-bit words)
- For IEEE Single Precision: $m_{bits} = 24$, $e_{bits} = 256$, $k = 86$: $CR_{bits} = 640$.
  (10×64-bit words)

We divide the complete register into 64-bit words because it is a standard width in SRAM macros.

Let a and b be two corresponding elements of the input vectors; computation proceeds as follows:

1) a and b are fetched from memory and fed into the datapath.
2) The product of the mantissas ($prod_{mant}$) and the sum of the exponents (sum_{exp}) of a and b are calculated in parallel.
3) The high-order bits of sum_{exp} index into the complete register and select the correct words for accumulation.
4) The low-order bits of sum_{exp} are used to align the $prod_{mant}$ with corresponding words of the complete register.
5) The aligned $prod_{mant}$ and the selected words from the complete register are summed and written back to the complete register.
6) If necessary, any carry or borrow produced by the initial sum is propagated.

B. SoC Architecture

To perform a design-space exploration, we implemented our accelerator in the parameterizable hardware construction language Chisel [11] to plug into the open-source Rocket Chip System-on-Chip (SoC) Generator [12], which is based on the free and open RISC-V ISA [13].

We integrated the accelerator into the Rocket Chip SoC generator in order to evalute it in the context of a realistic system. Rocket Chip has been used to fabricate over a dozen academic and industrial SoCs. We chose a Rocket Chip configuration consisting of an in-order scalar pipeline, L1 instruction and data caches, and a unified L2 cache.

Figure 1 shows the architecture of the full SoC. The accelerator connects to the CPU through the Rocket Custom Coprocessor (RoCC) interface [14]. RoCC is an extension to the RISC-V ISA that supports decoupled accelerators. We provided a small set of RoCC instructions to program the

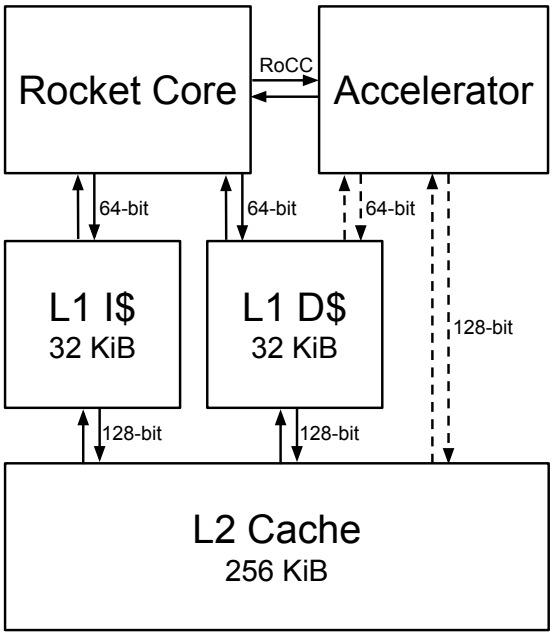

Fig. 1: High-Level System Architecture

accelerator as shown in Table I. These instructions include clearing the complete register (`CLR_CR`), loading the complete register from memory (`LD_CR`), storing the complete register to memory (`ST_CR`), summing the complete register with another stored in memory (`ADD_CR`), rounding the complete register to single or double precision (`RD_DBL`/`RD_FLT`), and running a dot product on two vectors stored in memory. Executing a dot product requires two instructions: `PRE_DP`, which provides the starting memory addresses for each input vector, and `RUN_DP`, which provides the length of the vectors and begins execution.

The accelerator can be configured to connect to memory through either the L1 data cache or the L2 cache. The L1 provides a 64-bit interface that is sufficient for performing one single-precision FMA per cycle or one double-precision FMA every other cycle. The L2 provides a 128-bit interface which allows for a double-precision FMA every cycle. While the L2 provides double the bandwidth of the L1, fully utilizing this bandwidth requires a non-trivial amount of logic and buffering to handle multiple outstanding requests that can be reordered. We provide both interfaces to evaluate the trade-off between memory bandwidth and performance vs. accelerator area.

C. Microarchitectural Implementation

The accelerator is organized in four major components.

1) Control unit: decodes RoCC instructions and drives the other units.
2) Memory unit: fetches operands from memory.
3) Front-end datapath: Computes products of incoming operands and shifts them into alignment with complete register.
4) Accumulator and complete register: Accumulates incoming products into the complete register.

D. Control Unit

The control unit decodes commands from the host processor and utilizes the rest of the accelerator to execute them. The control unit has two parts: the control state machines and rounding logic.

1) State Machines: For simplicity and verifiability, we implemented mutually exclusive state machines for each of the supported instructions. This decision led to a small amount of functional and state replication but simplified designing and verifying the accelerator.

2) Rounding Logic: The rounding logic takes the most significant words from the complete register and outputs an IEEE floating-point number. Using a priority encoder to find the most significant bit, the rounding logic selects the appropriate most significant 23 or 52 bits for the mantissa (ignoring the implicit most significant bit of 1). If the number is negative, the rounding logic negates the mantissa. This is due to the fact that the complete register is stored in a signed fixed-point format in contrast to the unsigned format of an IEEE floating-point mantissa. The exponent is calculated based on the position of the most significant bit of the mantissa relative to the bit representing the coefficient of 2^0 in the complete register.

Our accelerator only supports round towards zero, but it would be straightforward to implement other rounding modes. Round up, down, away from zero, and to nearest all require logic to determine when to increment as well as associated incrementers. Deciding to increment is a function of the rounding mode, the sign, and all bits in the complete representation less significant than the mantissa. While there are potentially thousands of bits to consider, microarchitectural techniques like the `all_ones` and `all_zeros` registers described in Section III-G2 can make this process both simpler and more efficient.

E. Memory Unit

The memory unit in Figure 2 shows the logic and state elements required to issue memory requests every cycle. It contains two *arrays* corresponding to the vector operands for the dot product operation. Each array keeps track of the address of the current element as well as the number of elements remaining in the vector.

An arbiter issues requests from each array in a round-robin manner. When the arbiter grants an array's request, the array pushes the tag of the corresponding memory request into a FIFO for memory reordering. The array also reserves a slot in the memory response reordering queue. When requests return from memory, they are stored into the reordering queue. When the tag of a request in the reordering queue matches a tag from the front of either array's tag queue, the floating-point data is issued to the datapath.

31 25	24 20	19 15	14	13	12	11 7	6 0
func7	rs2	rs1	xd	xs1	xs2	rd	opcode
6	5	5	1	1	1	5	7
CLR_CR	00000	00000	0	0	0	00000	custom-0
RD_DBL/RD_FLT	00000	00000	1	0	0	rd	custom-0
LD_CR	00000	rs1	0	1	0	00000	custom-0
ST_CR	00000	rs1	0	1	0	00000	custom-0
ADD_CR	00000	rs1	0	1	0	00000	custom-0
PRE_DP	rs2	rs1	0	1	1	00000	custom-0
RUN_DP	00000	rs1	0	1	0	00000	custom-0

TABLE I: Accelerator Instruction Encodings

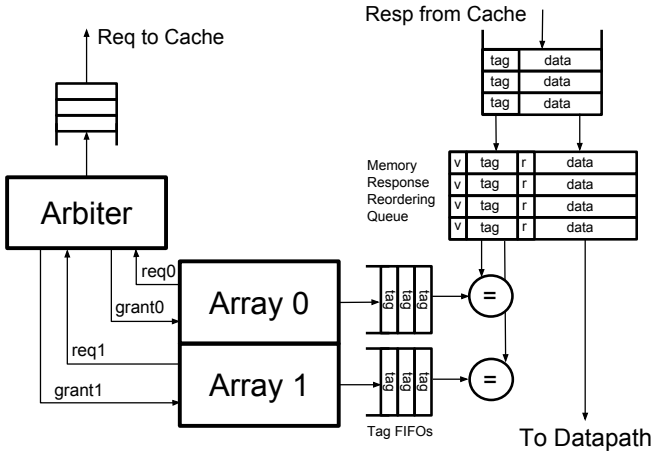

Fig. 2: Memory Unit

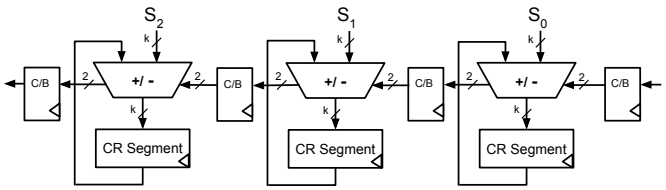

Fig. 3: Segmented Accumulator and Complete Register

F. Front-end Datapath

The front-end datapath consists of three modules: exponent adder, multiplier, and shifter. Aside from some initial floating-point decoding (eg. prepending one to the mantissa or zero to properly handle denormalized numbers), the functionality of these blocks are expressed with Chisel's addition, multiplication, and shift operators respectively. Chisel maps these operations to the equivalent Verilog HDL operator so that the VLSI toolchain can infer optimized implementations for these operators in logic synthesis. These modules are parameterized by pipeline latency, which inserts the requisite number of registers on the output of the shifter. These registers are retimed by the design tools across the three modules to pipeline the datapath as a whole.

G. Complete Register

We explore two complete register implementations which are described at length in [7]. This first is a *segmented* implementation, which has a separate adder for each word of the complete register. The second is a *centralized* implementation, which shares a single wide adder and stores the complete register in SRAM.

1) Segmented Accumulator and Complete Register: This implementation divides the complete register into segments of width k, each with its own k-bit adder. In a single cycle, a segment adds a portion of the $prod_{mant}$ and incorporates an incoming carry or borrow if present. If a given segment produces a carry or borrow, the carry or borrow is latched and propagated to the next segment in the next cycle (akin to the implementation of a carry-save multiplier). To ease carry/borrow propagation, and simplify most-significant word detection, each segment includes two flags to denote if the segment's bits are all ones or all zeros.

2) Centralized Accumulator and Complete Register: Given that the multiplicand for double-precision floating-point is 106 bits while the total fixed-point space is 4288 bits, it is much more efficient to implement a complete register where a single adder is shared for accumulation. The centralized accumulator microarchitecture is shown in Figure 4.

The following explanation assumes a double-precision implementation: a similar discussion applies for single precision. The accumulator receives the 104-bit summand—zero-padded to 192-bits and aligned to the 64-bit word boundaries of the complete register—from the shifter. It reads the appropriate four words (as indicated by the summed exponents), and adds the summand to the lower three words. The fourth word is incremented or decremented in the event of a carry or borrow respectively. The resulting sum is then written back to the complete register during the next cycle.

It is possible for a carry or borrow to propagate beyond the fourth word. A common example of this is when the sign of the entire complete register flips—requiring carry/borrow propagation all the way past the top word. We make this process more efficient by introducing two registers: `all_zeros` and `all_ones`. Both of these registers have one bit for each word in the complete register which indicates whether the word is composed of all zeros or all ones respectively.

When a carry propagates beyond the fourth word, the carry logic uses the `all_ones` register to find the next word in the complete register that is not all ones. If there is such a

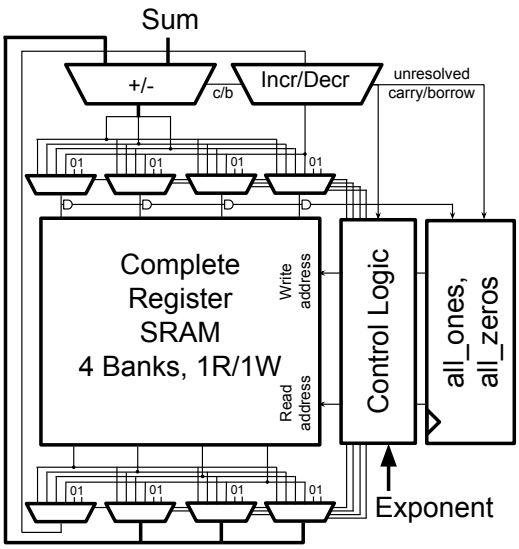

Fig. 4: Centralized Accumulator and Complete Register

word, the pipeline stalls while the word is read, incremented, and then written back. Every word that was skipped over on account for being all ones is switched to being indicated as all zeros without any reads or writes to the complete register. If the carry propagates past the highest word, then no pipeline stall is necessary as the carry is handled by modifying the all ones and all zeros registers.

Borrows work in the same way except that borrows propagate past words that are all zeros, turning those words to all ones. One subtle point is that any time a word is read from the complete register, the `all_zeros` and `all_ones` registers are used to select whether the output of the complete register or the appropriate all zero or all ones constant should be used.

We implemented the centralized accumulator as a 4-bank, dual-ported 8T SRAM that supports writing back the accumulated sum while reading words for the next accumulation. There is a forwarding path for each bank since back-to-back accumulates may touch the same words in the complete register.

IV. Methodology

To justify the addition of an accelerator for computing an exact dot product, we compare its performance against a range of software implementations and measure the cost of its hardware implementation. Software libraries are more flexible and have no additional hardware cost since they use structures already present in desktop and server class microprocessors. Specialized hardware should be more efficient at performing a fixed-function, but incurs additional hardware cost.

A. Benchmarking

We compare the performance of our accelerator against two important software solutions: Intel MKL, one of the fastest inexact BLAS libraries, and ReproBLAS, a slower but reproducible library. These libraries are highly tuned to run on commercial architectures (x86) using SIMD instructions which have no equivalent in the Rocket Chip generator. As such, it is difficult to fairly compare a port of these libraries to our microarchitecture. In lieu of this, we compare cycles-per-element (CPE). Since the product of every pair of elements is independent of one another, it is trivial to deeply pipeline our accelerator. As such it should be possible to achieve clock-periods comparable to those present on a commercial x86 machine.

Secondly, these libraries rely deeply on multi-threading to achieve their performance (and libraries like ReproBLAS only become necessary once they are run in parallel). Our accelerator has the capacity to add complete representations stored in memory, so it is possible to divide the arithmetic load across multiple cores each with their own accelerator before doing a final exact reduction into a single accelerator. In this paper, we measure only the single-threaded performance of these libraries, reducing our comparison to a challenge of how well each library can use the memory bandwidth provided to a single core.

We ran the library implementations on an Intel Xeon E5-2667 V2, running at 3.3 GHz, with 32 KiB L1 instruction and data caches and a shared 256 KiB L2 cache, supporting SSE4.1, 4.2 and AVX SIMD extensions. We compiled with ICC (version 14.0.1) with -O3 enabled. We used PAPI [15] to query the core's performance counters to measure cycle count and calculate CPE. As benchmarks, we calculated the dot product of vectors of randomly generated double and single-precision floating-point numbers with uniformly distributed mantissas and exponents (we excluded infinity and NaN). We warmed the caches to control for memory hierarchy effects. Each vector spanned a contiguous region of virtual memory. We did not measure sparse vector representations because they are not supported by our accelerator.

We ran benchmarks on Rocket Chip "bare metal", without an operating system and accessing physical memory directly, to simplify gate-level simulation to measure energy. We measured cycle counts using the RISC-V RDCYCLE instruction. All Rocket Chip benchmarks were simulated using Synopsys VCS [16]. The Rocket Chip configurations we used for benchmarking had a similar cache hierarchy (L1 - 32 KiB, L2 - 256 KiB), providing 64 and 128 bits/cycle of memory bandwidth respectively.

B. VLSI Flow

Since both the accelerator and Rocket Chip are implemented in Chisel as hardware generators, we had the ability to sweep a wide space of design points that varied both the host Rocket core and the accelerator itself. We present 8 representative design points that are the product of three binary parameters.

1) Complete register organization: how the complete register is implemented in silicon. Segmented (S) or centralized (C).
2) Cache interface: the cache layer from which operands are fetched: L1 or L2 cache interface.

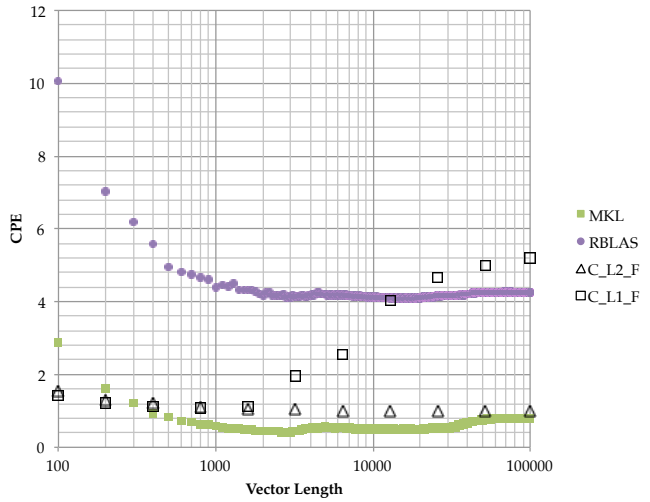

Fig. 5: Single-Precision CPE vs Vector Length

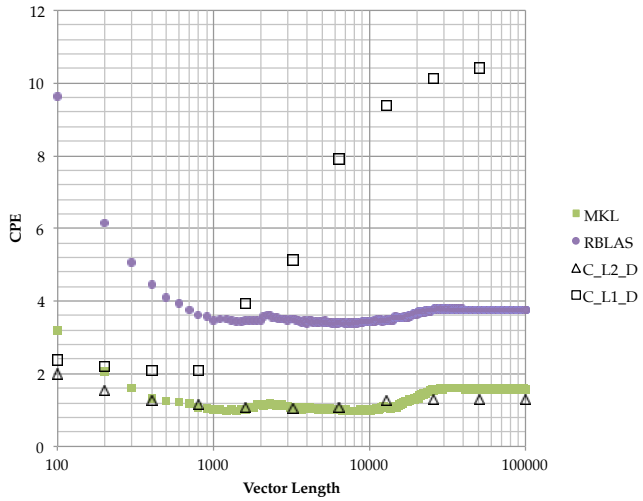

Fig. 6: Double-Precision CPE vs Vector Length

3) Operand precision: The largest floating-point operand type supported by the accelerator. Single/float (F), or double (D) precision.

We refer to each design point by concatenating parameters like so: {organization}_{cache interface}_{precision}. For example, the *C_L1_F* configuration refers to an accelerator with a centralized CR, L1 cache interface, and support for single-precision operands.

We used Synopsys tools for synthesis and place-and-route (Design Compiler and IC Compiler respectively), and targeted an industry standard 45 nm process. Since we did not have access to a memory compiler in this technology, we used an in house pseudo-compiler that used CACTI [17] to generate timing and area models that were black-boxed into cells.

V. Results

A. Benchmarking Results

Figure 5 and Figure 6 present the performance of our accelerator versus ReproBLAS (RBLAS) and Intel MKL. Across both precisions, MKL succeeds in fully utilizing the data cache bandwidth provided, giving 1 and 0.5 CPE on single and double-precision respectively.

Only the centralized CPE is shown as the segmented version has roughly the same performance. The datapath stalls in the centralized implementation only in the extremely rare case in which a carry or borrow creates a structural hazard on one of the banks being accessed by the following accumulation. The segmented implementation never stalls, however it may take several cycles for an outstanding carry or borrow to propagate the length of register upon completion.

The L1 variant performance of the accelerator is considerably worse than L2 variants as soon as the input vectors no longer fit in the L1 cache. Our L1 cache implementation accepts 4 outstanding 64-bit requests, a grossly insufficient number to hide the miss penalty. With an appropriate prefetcher, it should be possible to achieve 1 and 2 CPE for single and double-precision variants respectively.

Finally, since our accelerator can only issue a single product per cycle, single-precision L2 variants cannot achieve a CPE less than 1 whereas MKL can. This is the only case in which the accelerator is not memory bound. To rectify this, the datapath could be duplicated. Sharing the same accumulator across multiple datapaths is easy to implement with a segmented complete register. However, sharing a centralized accumulator is more difficult. It requires either additional read and write ports commensurate with the increase in the number of datapaths (which is very expensive) or stall logic to circumvent constant structural hazards.

B. VLSI Results

Figure 7 shows the SoC area (excluding L2) breakdown for each of the 8 representative design points mentioned in IV-B. Because the L2 occupies about 70% ($1.65mm^2, \sigma = 0.006mm^2$) of the total SoC area, we excluded it for clarity. We distinguished the Rocket Chip FPU from the Rocket Core to show the cost of a standard floating-point unit. As expected, segmented complete registers tend to occupy more area than centralized, double-precision support requires more area than single precision, and interfacing the L2 requires more area than interfacing with the L1. Figure 8 further illustrates these observations by breaking down the accelerator itself into its major components.

Given that the segmented complete register uses registers instead of SRAM cells to store the accumulator and includes an adder for each segment, it follows that the segmented complete register should be much larger than the centralized configuration. Notably, the penalty for using a segmented complete register is reduced for single-precision configurations. This is due to the fact that this configuration of the centralized complete register also uses registers instead of SRAMs since only 10×64-bit words are necessary for

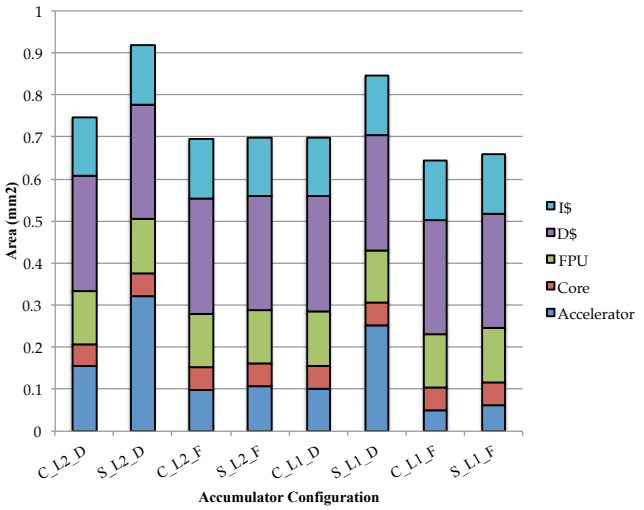

Fig. 7: SoC Area (excluding L2)

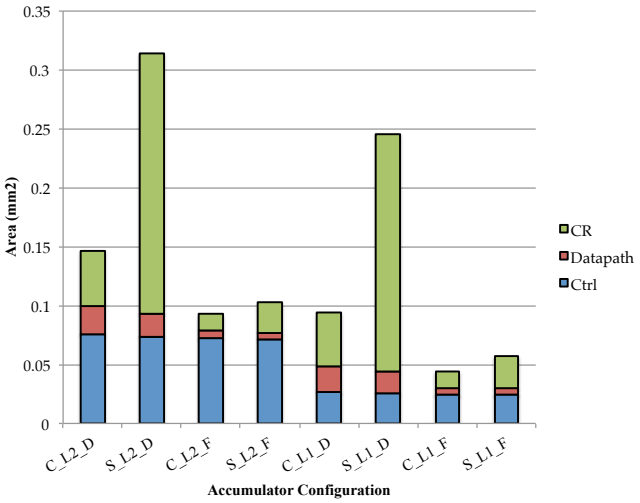

Fig. 8: Accelerator Area

single-precision. There are also thus only 10 64-bit adders in the segmented version compared to 1 128-bit adder in the centralized configuration.

The control module is responsible for buffering responses from the caches, it thus makes sense that buffering 64-bit L1 responses takes far less area than 512-bit (128-bit $\times$ 4 beats) L2 responses.

The accelerator ranges from 2.1% to 12.4% of the total SoC area. The most sensible configuration (centralized complete register, L2 cache interface, double precision operands) occupies 6.4% of the total area. Due to a critical path through the L2, ICC found each design point to have a clock period of 1.45 ns. Without the L2, ICC found the following critical paths:

- Centralized: 1.24 ns (from accumulator SRAM out to accumulator SRAM in)
- Segmented: 1.09 ns (through the Rocket Chip FPU)

VI. Conclusion

In this paper, we presented an accelerator for computing an exact dot product, the first of its kind to share the same die as the host core. Implemented in Chisel, the accelerator can be tailored at different area costs to support single or double precision operands, and interface with different levels of the cache hierarchy to exploit more memory bandwidth if it is available.

Our accelerator achieves peak memory throughput, providing equivalent performance to sequential Intel MKL dot product kernels leveraging AVX (1 CPE for double precision vectors). Versus a fast reproducible implementation, ReproBLAS, the accelerator is approximately 3$\times$ faster. Implemented in an industry-standard 45 nm process, the area cost of the accelerator is small relative to a modern SoC, at only 6.4% of the area of our test setup, which includes typical-size caches but a very small in-order scalar processor pipeline.

Whether an accelerator such as this one should be integrated into a general-purpose computer remains an outstanding question. One notable problem with this scheme is that it challenges cache-blocking approaches used in BLAS-2 and BLAS-3 kernels. For example, in GEMM, block dimensions should tend away from squares to help amortize the cost of flushing and fetching the complete register to memory. While this presents longer sub-vectors for the accelerator to process, it comes at the expense of temporal reuse of cache-resident data.

However, in an application-specific SoC that may require strong assurances of accuracy and whose workload is well matched to our accelerator, the area costs are modest. These costs can be further ameliorated if the accelerator can be more tightly coupled to the floating-point pipeline of the host core, allowing the reuse of already present arithmetic units. More aggressively, an existing SIMD or vector unit could be augmented to behave like our accelerator. It is not uncommon for a vector machine to have 64 element 64-bit vectors, only 192 bits short of the double-precision complete representation.

Other avenues for future work include: supporting the accelerator in a multi-programmed environment, parallelizing the datapath to achieve $<$ 1 CPE, and supporting additional arithmetic operations between multiple complete registers within an accelerator. Finally, a more thorough evaluation comparing the energy trade-offs of our accelerator versus modern software libraries that provide reproducibility and greater accuracy, like ReproBLAS and EXBLAS, would shed more light on the accelerator's utility.

Acknowledgment

This research was partially funded by DARPA Award Number HR0011-12-2-0016 and ASPIRE Lab industrial sponsors and affiliates Intel, Google, HPE, Huawei, LGE, Nokia, NVIDIA, Oracle, and Samsung. Any opinions, findings, conclusions, or recommendations in this paper are solely those of

the authors and does not necessarily reflect the position or the policy of the sponsors.

J.K. and D.B. would like to deeply thank Jim Demmel, Hong Diep Nguyen, and William Kahan for their insights into computer arithmetic. They'd also like to thank Colin Schmidt, who shepherded them through the early stages of project.

REFERENCES

[1] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, "MPFR: A Multiple-precision Binary Floating-point Library with Correct Rounding," *ACM Trans. Math. Softw.*, vol. 33, no. 2, Jun. 2007. [Online]. Available: http://doi.acm.org/10.1145/1236463.1236468

[2] R. Iakymchuk, S. Collange, D. Defour, and S. Graillat, "ExBLAS: Reproducible and Accurate BLAS Library," Jul. 2015. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01202396

[3] BLAS (Basic Linear Algebra Subprograms). [Online]. Available: http://www.netlib.org/blas/

[4] R. Iakymchuk, S. Collange, D. Defour, and S. Graillat, "ExBLAS: Reproducible and Accurate BLAS Library," in *NRE: Numerical Reproducibility at Exascale*, Austin, TX, United States, Nov. 2015, numerical Reproducibility at Exascale (NRE2015) workshop held as part of the Supercomputing Conference (SC15). [Online]. Available: https://hal.archives-ouvertes.fr/hal-01202396

[5] P. Ahrens, H. D. Nguyen, and J. Demmel, "Efficient reproducible floating point summation and blas," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-229, Dec 2015. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-229.html

[6] "IEEE Standard for Binary Floating-Point Arithmetic," *ANSI/IEEE Std 754-1985*, 1985.

[7] U. Kulisch, *Computer Arithmetic and Validity: Theory, Implementation, and Applications*, 2nd ed. de Gruyter, 2013.

[8] Y. Uguen and F. de Dinechin, "Design-space exploration for the Kulisch accumulator ," Mar. 2017, working paper or preprint. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01488916

[9] T. Ogita, S. M. Rump, and S. Oishi, "Accurate Sum and Dot Product," *SIAM J. Sci. Comput.*, vol. 26, no. 6, pp. 1955–1988, Jun. 2005. [Online]. Available: http://dx.doi.org/10.1137/030601818

[10] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark Silicon and the End of Multicore Scaling," *SIGARCH Comput. Archit. News*, vol. 39, no. 3, pp. 365–376, Jun. 2011. [Online]. Available: http://doi.acm.org/10.1145/2024723.2000108

[11] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing Hardware in a Scala Embedded Language," in *Proceedings of the 49th Annual Design Automation Conference*, ser. DAC '12. New York, NY, USA: ACM, 2012, pp. 1216–1225. [Online]. Available: http://doi.acm.org/10.1145/2228360.2228584

[12] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman, "The rocket chip generator," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17, Apr 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[13] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović, "The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-54, May 2014. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.html

[14] C. Schmidt, "Risc-v 'rocket chip' tutorial," Jan. 2015. [Online]. Available: http://riscv.org/workshop-jan2015/riscv-rocket-chip-tutorial-bootcamp-jan2015.pdf

[15] P. J. Mucci, S. Browne, C. Deane, and G. Ho, "PAPI: A Portable Interface to Hardware Performance Counters," in *In Proceedings of the Department of Defense HPCMP Users Group Conference*, 1999, pp. 7–10.

[16] Synopsys VCS. [Online]. Available: https://www.synopsys.com/verification/simulation/vcs.html

[17] S. J. E. Wilton and N. P. Jouppi, "CACTI: An Enhanced Cache Access and Cycle Time Model," *IEEE Journal of Solid-State Circuits*, vol. 31, pp. 677–688, 1996.