BLAS Comparison on FPGA, CPU and GPU

Srinidhi Kestur† John D. Davis‡ Oliver Williams‡

† Dept. of Computer Science and Engineering
The Pennsylvania State University
University Park, PA 16802
kesturvy@cse.psu.edu

‡ Microsoft Research
Mountain View, CA 94043
{john.d, olliew}@microsoft.com

***Abstract*—High Performance Computing (HPC) or scientific codes are being executed across a wide variety of computing platforms from embedded processors to massively parallel GPUs. We present a comparison of the Basic Linear Algebra Subroutines (BLAS) using double-precision floating point on an FPGA, CPU and GPU. On the CPU and GPU, we utilize standard libraries on state-of-the-art devices. On the FPGA, we have developed parameterized modular implementations for the dot-product and Gaxpy or matrix-vector multiplication. In order to obtain optimal performance for any aspect ratio of the matrices, we have designed a high-throughput accumulator to perform an efficient reduction of floating point values. To support scalability to large data-sets, we target the BEE3 FPGA platform. We use performance and energy efficiency as metrics to compare the different platforms. Results show that FPGAs offer comparable performance as well as 2.7 to 293 times better energy efficiency for the test cases that we implemented on all three platforms.**

I. Introduction

Recently, scientific computing or high performance computing (HPC) on non-traditional computing platforms, including embedded processors and GPUs, has gained significant traction [1], [2]. One of the most essential libraries for HPC is the Basic Linear Algebra Subroutines(BLAS) [3]. BLAS is classified into levels based on the degree of complexity. BLAS level 1 includes the Dot-product, which has $O(N)$ complexity. BLAS level 2 includes *Gaxpy* or matrix-vector multiplication, which has $O(N^2)$ complexity. BLAS level 3 includes matrix-matrix multiplication which has $O(N^3)$ complexity [4]. These common matrix operations are fundamental to most computations in scientific applications and the BLAS libraries were created to speed the implementation of these HPC codes. Thus, the BLAS libraries serve as the basic building blocks for many numerical linear algebra applications, including the solution of linear systems of equations, linear least square problems, eigenvalue problems and singular value problems [4].

Unlike CPUs, FPGAs are able to continue to ride the Moore's Law curve, providing more logic and memory resources with each new generation of FPGAs. This increased capacity has recently enabled mapping HPC applications [5]–[7] to FPGAs. Likewise, GPUs, which are tuned for the graphics pipeline, have grown considerably more programmable and the combination of high floating point performance and a C-like programming model has garnered wider acceptance by the scientific computing community for HPC [2]. While modern GPUs offer very high peak floating point performance, the true strength of FPGAs lies in the fraction of the peak performance that can be extracted for a particular application [8].

In this paper, we describe FPGA optimizations for BLAS level 1. Using this as a basis for BLAS level 2, we compare the performance and energy efficiency of the CPU, GPU, and FPGA platforms. We use state-of-the-art off-the-shelf devices and standard high performance libraries for the CPU and GPU and all implementations use double-precision floating point numbers. While there are many published works on BLAS architectures for FPGAs and reported numbers for GPUs, our implementation use double-precision floating, measures full system power, and provides a systematic comparison of performance and energy efficiency across these three platforms, which does not exist.

We have developed a custom implementation for the dot-product and Gaxpy for FPGAs. As both these operations require a reduction of floating point values, we have designed a novel reduction circuit using IEEE compliant double-precision floating point units. Further, we introduce an efficient *vector memory* and a novel *matrix memory* which support burst-read and write and enable extraction of a high degree of parallelism for matrix computations on FPGAs. We use the BEE3 [9] as our FPGA platform though our design is portable to any FPGA platform. We perform the comparison on small data-sets initially to only consider computation time without memory bottlenecks. This is done by varying the aspect ratio of the matrix while maintaining the same overall size and complexity. We use execution time, average power and energy efficiency (iterations/joule) to compare the BLAS performance of the FPGA, CPU, and GPU implementations.

The rest of the paper is organized as follows - Section II discusses related work, Section III discusses the BLAS level 1 FPGA implementation and optimizations, Section IV describes the BLAS level 2 implementation using novel banked memory systems, Section V presents the experimental setup, Section VI provides the evaluation of BLAS level 2 implementations on the FPGA, CPU and GPU platforms. Section VII concludes the paper.

II. Related Work

BLAS has been a topic of considerable interest since it was first introduced in Fortran [3]. As a result, BLAS has been used as a benchmark by the industry to compare performance of new devices [5], [6].

There are several works which focus solely on the floating point summation/reduction on Reconfigurable platforms (RC), which is a critical component in BLAS. Zhuo et al. [10], [11] propose techniques such as Dual-Strided Adder and Single-Strided Adder using IEEE compliant pipelined floating point units. Some other works suggest modifying traditional floating point arithmetic to achieve low-latency addition, simplifying the reduction circuit. Examples are delayed addition [12], group-alignment based summation [13] and parameterized accumulation [14]. In this work, we design a custom reduction circuit that uses IEEE compliant double-precision floating point arithmetic to cater to the entire range of numerical values, and a variety of vector lengths and matrix aspect ratios that might appear in scientific applications.

There has been substantial interest in implementing BLAS levels 2 and 3 on RC platforms. Zhuo et al. have proposed a parallel implementation for matrix multiplication [15] and have mapped it to the SRC MAP platform [16], which has 2 FPGAS with SRAM and DRAM. They also suggest hardware/software co-design frameworks for RC platforms such as SRC MAP in [17], [18]. Smith et al. proposed a simple partitioning strategy for multi-FPGA implementation [7] on the SRC Map platform. Kumar et al. have implemented matrix multiplication using rank-1 update algorithm [19]. In this work, while we concentrate on the BLAS Level 2 or Gaxpy, each module is designed to be used for matrix computations in general. The most relevant work for Gaxpy is [16] which describes an interleaved reduction circuit to provide high throughput matrix-vector multiplications. This design uses custom floating-point cores and thrives on the high bandwidth of the SRAM external memory on the SRC Map platform. However, the interleaving is not practical, since floating-point adders are deeply pipelined and memory bandwidth would seriously limit the ability to sustain the interleaved computation. This would result in stalling when the input buffers are not filled.

In our FPGA implementation, we load a portion of data into the local memory on the FPGA, which is used as a scratch-pad/cache memory and complete the computation on the cached data, similar to [19]. However, our design exploits the data locality better than [19] by using novel local memory design and offers similar throughput for the entire range of matrix dimensions owing to the novel accumulator design. In the future, we are going to overlap the computation with memory access and maximize data reuse. While data reuse is minimal for Gaxpy, matrix-multiplication has significant reuse. Our approach is to build a design whose parallelism can scale with resource availability.

III. BLAS Level 1 on FPGA

The dot-product forms the basic operation for many matrix computations. The dot-product, $a = X \cdot Y$, of 2 vectors of length N is a sum-of-products operation with N multiplications and $N-1$ additions. While the multiplications can be done in parallel, their products must be accumulated to provide a scalar result. The accumulation is achieved by a reduction circuit/accumulator. The accumulator has the challenge of adding the sequentially delivered N floating-point values to provide a scalar result. Since floating-point additions are deeply pipelined, the result of an addition is available at the output, a few cycles after the operands were input. We first develop a naive implementation for the accumulator and then perform architectural optimizations to maximize throughput and minimize stalling and buffer requirement as described in the following sub-sections.

A. Single-Accumulator

The intuitive implementation of the accumulator consists of an adder whose output is fed-back to the input and an input FIFO buffer as shown in Figure 1(a). The accumulator state machine is shown in Figure 1(b). In the *Fill* state, the adder input pin A is tied to zero and the input pin B is fed with the first p values to be accumulated, where p is the pipeline depth of the adder. When the result of the first addition is ready (after p cycles), the accumulator moves to the *Steady* state, in which the sum of the adder appears at A and input element $p+i$ is added with input i. This continues until all the n values in the input set are fed into the accumulator, after which the accumulator moves into *Coalesce* state.

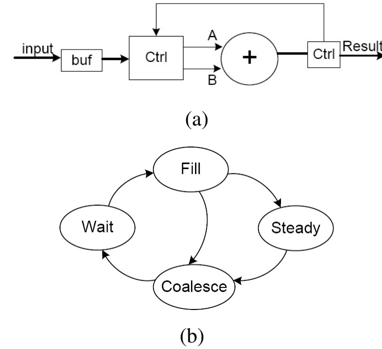

Fig. 1. Single Accumulator (a) circuit and (b) state machine

In the *Coalesce* state, the p partial sums in the pipeline are reduced to a single value by a staggered addition. Pipeline registers are enabled at one of the input pins of the adder. Each cycle, a partial sum is flushed out of the adder pipeline and is fed into the adder which is buffered in pipeline registers until the next partial sum arrives. When both input pins of the adder have partial sums, they are fed into the adder. The reduction of the partial sums to a single value requires flushing the adder pipeline log_2p times.

The circuit then moves into the *Wait* state where it waits for a new input set to arrive. Hence, the accumulator has to be stalled for $p \times log_2p$ cycles which results in a large input buffer requirement when there are multiple sets to be accumulated in sequence. The overall time taken by the *Single-Accumulator* to reduce m sets of length n is $L = m[n + (plog_2p)]$. In our implementation, the adder is generated using Xilinx coregen [20] with $p = 8$.

B. Double-Accumulator

A simple optimization to minimize the effective latency of the accumulator is to‘ have two identical accumulators A and B with a common input buffer, similar to the Dual-strided adder (DSA) in [10]. At the end of an input set, A enters the *Coalesce* state and B can begin accepting a new input set. When B enters the *Coalesce* state, if A is in *Wait* it can accept a new input set. If neither accumulator is ready, the input set is buffered until one of them enters the *Wait* state. Hence, A and B alternate to reduce successive sets of inputs, hiding the latency of the accumulator whenever the length of the input set $n \geq plog_2p$. However, if successive input sets of shorter length are to be reduced, stalling is still required for each set, which increases the input buffer requirement. The overall time taken by the *Double-Accumulator* to reduce m sets of length n is given by $L = (n \geq plog_2p)?[mn+plog_2p] : [m(plog_2p)];$

C. Dual-stage Accumulator

The accumulator coalesce latency is the performance bottleneck. The ideal optimization is to separate the coalesce operation from the accumulator, having an additional adder stage to perform the coalescing. The adder stage has to be a feed-forward circuit, eliminating pipeline stalls completely. The *dual-stage accumulator* circuit which implements the above modifications is shown in Figure 2. It consists of a *Feedback adder* stage which is the original accumulator implementing the *Fill* and *Steady* states and the *Feed-forward*

adder stage, which implements the staggered addition for coalescing and removes the need for a coalesce state in the state machine. At the end of an input set, the Feedback adder stage goes into the *Wait* state and Feed-forward adder stage takes over the coalesce operation.

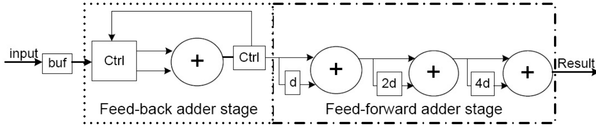

Fig. 2. Dual-stage accumulator circuit

The *Feed-forward adder* stage consists of $log_2 p$ adders in sequence, each having pipeline registers at their input pins. This stage effectively implements a binary adder tree to reduce the p partial sums. Each cycle, a partial sum is fed into the first adder which is buffered in pipeline registers until the next partial sum arrives. When both input pins of the adder have partial sums, they are fed into the adder. Successive adders also implement a similar buffer-and-add strategy where the buffer depth for the ith adder is 2^{i-1} cycles.

The accumulator can accept a new input set right on the next cycle, and eliminates the stalling and the input buffer requirement, for any length of the input set. The overall time taken by the Dual-stage Accumulator to reduce m sets of length n is given by $L = [mn + plog_2 p]$. Hence, successive sets of vectors, irrespective of their lengths can be reduced with no latency and memory bottlenecks.

IV. BLAS Level 2 on FPGA

Gaxpy or matrix-vector multiplication for a matrix A and vectors X and Y is given as, $\widehat{Y} = A \cdot X + Y$. It can be computed efficiently by performing a dot-product on a row of the matrix A and the vector X and repeating over all rows of A. This is a row-major Gaxpy using dot-product as the primitive. We exploit two levels of parallelism in this Gaxpy core. *Inter-row parallelism* in which multiple pipelines can perform the dot-product on different rows of the matrix concurrently because each dot-product in the Gaxpy is an independent operation; and *intra-row parallelism*: within each dot-product the multiplications are independent of one another, several multiplications can be performed concurrently and their results can be accumulated.

A. Vector Memory

FPGAs have a small amount of local memory called Block RAMs, which are used as scratchpad memories (RAMs) or as FIFOs. The typical dual-port RAM available on platform FPGAs has two ports and each port can be configured as a read or write port. However, for parallel vector computations, multiple vector elements need to be accessed concurrently, requiring multiple read and write ports. To support this, we have implemented a *vector memory* by utilizing bank interleaving.

The contents of the local memory are interleaved into B banks as shown in Figure 3(a), where each bank is a standard dual-port RAM. This enables the memory to support B reads and B writes per cycle as long as the read and write addresses access different banks each cycle. If care is taken to make sure no two reads and no two writes per cycle access the same bank, then the memory read/write bandwidth is B. The width and depth of each bank is a HDL parameter and B must be a power of two.

Each port has an address and data bus. A $B \times B$ switch serves as a bank assignment module and uses the least-significant $log_2(B)$ bits of the read/write address to assign a bank-ID for port p. If the operation was a read, the output data from the corresponding bank is assigned to output port p by an output-port assignment module. The additional control logic results in higher latency ($L > 1$ cycles) for the *vector memory*. However, the control logic is completely pipelined so that the vector memory can sustain a throughput of B reads and writes every cycle.

This vector memory is used as local memory for the vectors X and Y in our implementation and the number of banks, B, is set to 4. This enables 4 concurrent reads and writes for vector computations.

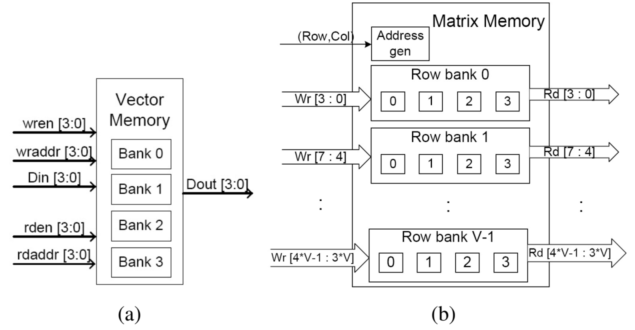

Fig. 3. (a) Vector memory (b) Matrix memory

B. Matrix Memory

For matrix computations, we use a two-dimensional memory to simultaneously access the matrix elements using the row and column indices. Further, to enable parallel matrix computations, it is ideal if a two-dimensional sub-block of the matrix can be accessed concurrently. In order to support the above, we extend the concept of bank interleaving to 2 dimensions by introducing a novel *matrix memory*.

The matrix memory can be visualized as an array of V vector memory elements called *row banks*. Each row-bank is a vector memory itself having B banks. As shown in Figure 3(b), a matrix is organized in the matrix memory such that rows of the matrix are interleaved into the V row banks, where row i of the matrix is mapped to row bank r where $r = mod(r, V)$. Within each row, the columns are interleaved within the banks of that particular row bank. For simplicity, we choose V to be a power of two.

The access to the matrix memory is a two-dimensional burst access. The external interface to the matrix memory has read and write data bus of width $V \times B$, a read and a write enable signal and a row and column index. Each cycle the matrix memory accepts a row index and column index and internally generates the $V \times B$ addresses which correspond to a 2D sub-block of the matrix. The read and write datapaths are independent pipelines and hence the matrix memory can sustain a throughput of $V \times B$ reads and writes per cycle.

This matrix memory has the advantage that it provides a 2D memory and allows easy access to matrix elements. While the vector memory supports access to any B (non-contiguous) elements of the vector stored, the matrix memory operates on the burst-access feature, which limits memory accesses to a

contiguous sub-block of the matrix. This is because the vector memory implements a bank assignment switch to perform dynamic bank assignment, whereas the matrix memory has no such control overhead. Moreover, matrix accesses in BLAS are usually in bursts, whereas vector accesses can be isolated. Hence, this is a very efficient technique to improve local memory bandwidth for matrix computations with minimal control overhead.

The matrix memory is used to store the matrix A and we have chosen the number of row banks, $V = 4$, with each row bank having $B = 4$. This allows a 4×4 sub-matrix to be read and written each cycle.

C. Gaxpy Architecture

A Gaxpy pipeline, *PIPE*, as shown in Figure 4, is the implementation of the dot-product of a row of the matrix A with the vector X. As mentioned before, the multiplications within the dot-product are independent and can be done in parallel. The PIPE can perform P multiplications in parallel, provided the inputs to P multipliers can be generated per cycle. Since P represents the maximum number of values that can be read per cycle from the vector memory X and a row bank of the matrix memory, $P = B$. Each cycle, the PIPE accepts P values of vector X and P values from a row of matrix A. The PIPE performs P multiplications in parallel and accumulates the results through an adder tree followed by the accumulator. Once the accumulator result is ready, the corresponding value in vector memory Y is read and added to the accumulated result. The final result is written back into vector memory Y at the same address.

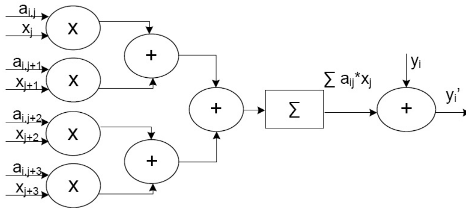

Fig. 4. Gaxpy Pipeline, *PIPE*

In our implementation, $P = B = 4$. The multipliers and adders are double-precision floating point Xilinx IP cores [20], which are generated using the Xilinx core generator.

The overall architecture for Gaxpy is shown in Figure 5. It includes multiple PIPEs in parallel, each operating on a different row of the matrix A. The vector X output data is broadcast to all the PIPEs. The number of PIPEs, Q, is determined by the number of rows that can be accessed in parallel from the matrix memory and is limited by the availability of FPGA resources. Since the maximum number of rows that can be accessed at a time is equal to the number of row banks, $Q \leq V$. In our implementation $V = 4$ and hence $Q \leq 4$.

Each cycle, the control logic generates a pair of (row, col) indices for the matrix memory and addresses corresponding to $(col : col + B - 1)$ of the vector memory X. The matrix memory returns a read burst of $V \times B$ sub-matrix elements corresponding to indices $(row : row + V - 1, col : col + B - 1)$. The read values from each row bank of the matrix memory are mapped to one of the Q PIPEs. The vector memory X returns B values which are broadcast to all the PIPEs. Each PIPE computes the dot-product and updates an element of the vector

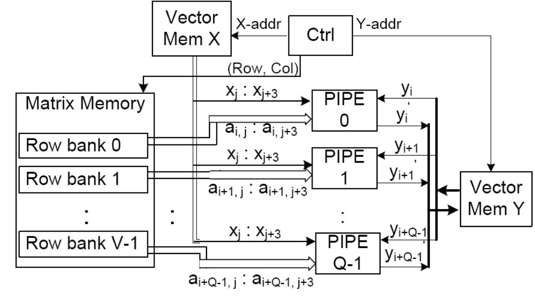

Fig. 5. Overall Gaxpy Implementation

$\widehat{Y}$. The control logic generates Q addresses corresponding to $(row : row + Q - 1)$ into the vector memory Y which are used to read from, and then write into the vector memory Y. If the number of PIPEs, $Q = V$, then the design is optimal with Degree of parallelism, $DoP = Q \times P = V \times B$. In our implementation, the $DoP = 4 \times 4 = 16$.

V. Experimental Setup

The BLAS level 2 kernels were evaluated on the FPGA, CPU, and GPU platforms for performance and energy efficiency. We developed and validated the kernels, verifying that all platforms generate the same result for the BLAS computations. The BLAS kernels are run at least three times consecutively and each run has between 100,000 to 1 million iterations of the kernel. This reduces the initialization overhead and provides a sustained kernel run that is long enough to measure the full system AC power using the *WattsUp? Pro* digital power meter.

A. CPU, GPU and FPGA platforms

For evaluating the naive C, Intel Math Kernel library (MKL) [21] and CUDA BLAS kernels [22] we use an HP xw4600 workstation with a 3.16 GHz Intel Core 2 Duo E8500 processor with 4 MB of L2 cache and 4 GB of DDR2-800 RAM, running a 64-bit version Windows Server 2008. We installed an Nvidia 9500 GT as the primary graphics card when running the C and MKL kernels. The idle system power for this configuration is about 70 Watts. We provide results for both a single threaded MKL implementation and a parallel MKL implementation for comparison. For the GPU evaluation an Nvidia Tesla C1060 [23] was added. This increased the idle power of the system to just under 128 Watts. The CUDA BLAS library CUBLAS 2.2 was used to implement the BLAS kernel [22] on the GPU. The Nvidia Tesla C1060 has 240 streaming processor cores running at 1.3 GHz. The Tesla C1060 has 4 GB of GDDR3 operating at 800 MHz.

We compare these systems to the BEE3 FPGA platform [9] running a Verilog BLAS kernel. This FPGA platform has four Virtex5 LX155T FPGAs [24] and each FPGA has 24320 logic slices, 212 Block RAMs and 128 DSP48E units. Each FPGA has two DRAM channels with two DIMM slots per channel and a maximum capacity of 16 GB of DDR2-400 DRAM (64 GB per BEE3 platform). The matrix dimensions are provided as parameters to the Verilog design and Xilinx ISE 11.1 [25] was used to generate the bitstream. The bitstreams are downloaded onto all four FPGAs on the BEE3 board and executed. Once the FPGAs are programmed, we use the global reset push button to synchronously start the BLAS kernel on all four FPGAs. The BLAS kernel has an operating frequency

of 100 MHz. A operating frequency is possible [19], but requires longer bit file generation times.These kernels are run for 1 million iterations to provide sufficient time to measure power consumption. We instrumented the BLAS kernel with performance counters to measure the execution time and used Chipscope [25] to read the counters at run-time. The FPGA resource utilization for the BLAS kernel is shown in Table I.

TABLE I
RESOURCE UTILIZATION FOR GAXPY ON VIRTEX5 LX155T

Num PEs	Slice Reg	Slice LUT	BRAM/Fifo	DSP48E
8	29600 (30%)	24900 (25%)	137 (64%)	72 (56%)
16	61977 (63%)	52900 (54%)	138 (65%)	108 (84%)

B. Power Measurement

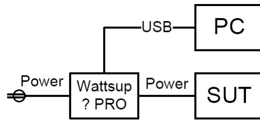

Fig. 6. Power measurement setup

The *full system power* is measured using a *Wattsup? Pro* [26] power meter. Our measurement infrastructure has three main components: the system under test (SUT), the digital power meter used to collect the full AC power consumption, and the computer that collects the AC power data from the meter (PC). Figure 6 illustrates the connections between these components. We interpose the WattsUp? Pro digital power meter between the SUT and the wall power to capture the AC power and power factor once per second. We connect this meter to a separate PC over USB to capture and log the power measurements once every second. The SUT is either the BEE3 or the PC in two different configurations, one with the Nvidia Tesla card running the CUDA BLAS code or without the Tesla card running the naive C or MKL implementations.

VI. RESULTS

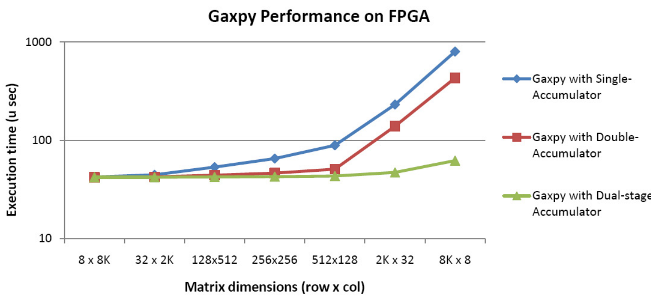

Fig. 7. Performance impact of accumulator circuit on Gaxpy implementation

We verified the results of all the implementations to ensure a fair comparison between the naive C, single thread Intel Math Kernel Library (MKL-ST) and parallel Intel Math Kernel Library (MKL-PAR), CUDA, and FPGA Gaxpy implementations. Due to the on-chip memory limitations of the FPGA, the Gaxpy implementations are evaluated using a set of relatively small matrix sizes to measure on-chip computation time on the 3 platforms without considering the I/O impact of main memory access, which is the focus of our future work. We use linear feedback shift registers to generate a known and repeatable sequence of values for the input matrix and vector for all the kernel implementations. The total on-chip memory requirement for Gaxpy is $M \times N + N + M$ and this is a constant for any aspect ratio of the matrices used in our study. On the Virtex5 FPGA, the largest square matrix whose size is a power of 2 that can be stored in local memory is 256×256. Hence, we choose various aspect ratios for a total matrix size of 2^{16} elements as our test cases. Different matrix aspect ratios are used as inputs to test robustness in handling a range of short to long vectors for the dot-product calculation. On the CPU platform, the on-chip L2 cache is big enough to hold the matrix and vectors and the same is applicable on the GPU platform.

We implemented multiple FPGA solutions for the BLAS level 2 kernel using the dot product. Section III described the optimization of the FPGA dot-product pipeline, which reduced pipeline latency and enabled near-perfect pipelining of the computation. Figure 7 illustrates the performance impact of these changes on the execution time of this Gaxpy kernel. The single-accumulator implementation suffers due to the latency of the accumulator coalesce operation for each set. For short and fat matrices($N \gg M$), this impact is nominal but for long and thin matrices ($M \gg N$) the performance degradation is high. This is synonymous with short vector operations on vector machines because we are performing row-major operations. The double-accumulator implementation performance curve shows similar trends: short vector lengths of less than $p \log_2 p$ stall the pipeline.

Our goal was to build a flexible FPGA platform for Gaxpy and this is demonstrated by the relatively flat curve of the *dual-stage accumulator*. There is some performance degradation due to the 1 cycle state machine refresh required for correct operation as can be seen in the figure for short vectors. As Figure 7 illustrates, the dual-stage accumulator is the the best design for the FPGA Gaxpy implementation. Further optimizations to this design would improve timing and not change the Gaxpy architecture.

Figure 8(a) reports the execution times of the kernels for a variety of matrix dimension combinations running on the three platforms. We provide both the single thread (MKL-ST) and parallel implementation (MKL-PAR) results for MKL, which provides the best performance results. MKL-PAR is 67%-88% faster than the MKL-ST implementation. It is 165% - 265% better than the dual-stage accumulator FPGA implementation and 3.3 to 4.3 times faster than the naive C implementation. It should be noted that the FPGA is operated at a nominal frequency of 100 MHz; several optimizations are possible to allow higher operating frequencies with marginal increase in power consumption. Finally, MKL-PAR is 1.8 to 88.4 times faster than the CUDA implementation. However, it should be noted that the CUDA BLAS implementation is dominated by the system and driver overhead for short and fat matrix dimensions and short vector effects for long and thin matrices.

Our FPGA implementation for Gaxpy offers a peak of 3.1 GFLOPS which is superior to the closest FPGA implementation [16] by a factor of 2.3. The peak GFLOPS and the variance over the matrix aspect ratios is given in Table II.

TABLE II
FLOPS VARIATION FOR GAXPY

	MKL-ST	MKL-PAR	CUDA	FPGA
Peak GFLOPS	5.812	9.818	3.023	3.113
Max Variance	29.35%	21.1%	97.1%	32.28%

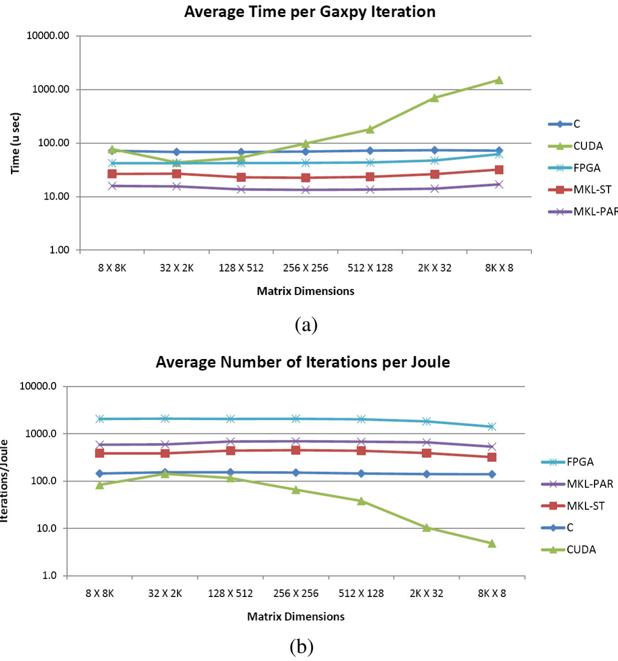

Fig. 8. Comparison of Gaxpy kernel on naive C, MKL-single thread, MKL-parallel, CUDA BLAS and our FPGA implementation in terms of (a) Execution time and (b) Average number of iterations per Joule

Power measurements are omitted for brevity. However, it should be noted that we run the same Gaxpy kernel on all four of the BEE3 FPGAs. We take the measured power and divide it by four to get the average per FPGA power. Approximate power values can be derived from Figure 8.

Given the average power and the runtime of the iteration, we can calculate energy efficiency which we report as the average number of iterations of Gaxpy that can be completed per Joule. Figure 8(b) reports the results. Even though the FPGA implementation is not the fastest, it uses an order of magnitude lower average power. Even if we used a computer with a quad core processor and the same power envelope as the dual core system, the BEE3 platform would still be more energy efficient than the general-purpose CPU. Future work will investigate larger problem sizes across all platforms.

VII. Conclusion

We present a comparison of BLAS Level 2 on CPU, FPGA and GPU platforms in terms of performance and energy efficiency. On the FPGA, we developed a custom implementation for *Gaxpy* (matrix-vector multiplication), which can be easily extended to matrix-matrix multiplication. We have introduced a bank-interleaved vector memory and a novel matrix memory which can support 2 dimensional burst access. The Gaxpy implementation consists of a parallel pipeline to compute the dot-product and support for several parallel pipelines to obtain both coarse-grain and fine-grain parallelism. Performance results for small matrix sizes show that FPGA has similar performance at higher energy efficiency when compared to the CPU and GPU platforms. Furthermore, the FPGA architecture was able to provide a flexible platform that could handle a variety of matrix aspect ratios without performance degradation. Finally, the BEE3 platform provides the opportunity to expand the data foot print up to 16 GB for one FPGA and partition the computation across four FPGAs and up to 64 GB, increasing the computational capacity, making an even more compelling customizable platform with a large memory footprint.

References

[1] L. Oliker, "Green flash: Designing an energy efficient climate supercomputer," in *IPDPS '09: IEEE International Symposium on Parallel Distributed Processing*, may 2009.

[2] "Supercomputing at 1/10th the cost." [Online]. Available: http://www.nvidia.com/object/tesla_computing_solutions.html

[3] "BLAS (basic linear algebra subprograms)." [Online]. Available: http://www.netlib.org/blas/

[4] G. H. Golub and C. F. Van Loan, *Matrix Computations*, 3rd ed. Baltimore, USA: Johns Hopkins University Press, 1996.

[5] K. Underwood and K. Hemmert, "Closing the gap: CPU and FPGA trends in sustainable floating-point blas performance," in *Field-Programmable Custom Computing Machines, 2004. IEEE Symposium on*, 2004, pp. 219–228.

[6] "FPGA coprocessing evolution: Sustained performance approaches peak performance," *Altera White Paper*, 2009.

[7] M. Smith, J. Vetter, and S. Alam, "Scientific computing beyond CPUs:FPGA implementations of common scientific kernels," in *MAPLD*, 2005.

[8] B. Sukhwani, M. Chiu, M. A. Khan, and M. C. Herbordt, "Effective floating point applications on FPGAs: Examples from molecular modeling," in *HPEC '09: Proc. of the Workshop on High Performance Embedded Computing*, 2009.

[9] J. Davis, C. Thacker, and C. Chang, "Bee3: Revitalizing computer architecture research," Microsoft Research, Tech. Rep. MSR-TR-2009-45, 2009.

[10] L. Zhuo, G. R. Morris, and V. K. Prasanna, "High-performance reduction circuits using deeply pipelined operators on FPGAs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 18, pp. 1377–1392, 2007.

[11] L. Zhuo, G. R. Morris, and V. K. Prasanna, "Designing scalable FPGA-based reduction circuits using pipelined floating-point cores," in *IPDPS '05: Proc. of the 19th IEEE International Parallel and Distributed Processing Symposium*. IEEE Computer Society, 2005, p. 147.1.

[12] Z. Luo and M. Martonosi, "Accelerating pipelined integer and floating-point accumulations in configurable hardware with delayed addition techniques," *IEEE Trans. Comput.*, vol. 49, no. 3, pp. 208–218, 2000.

[13] C. He, G. Qin, M. Lu, and W. Zhao, "Group-alignment based accurate floating-point summation on FPGAs," in *ERSA '06: Proc. of the 6th International Conference on Engineering of Reconfigurable Systems and Algorithms*, 2006, pp. 136–142.

[14] F. de Dinechin, B. Pasca, O. Cret, and R. Tudoran, "An FPGA-specific approach to floating-point accumulation and sum-of-products," in *FPT '08: International Conference on Field Programmable Technology*, December 2008, pp. 33–40.

[15] L. Zhuo and V. K. Prasanna, "Scalable and modular algorithms for floating-point matrix multiplication on FPGAs," in *IPDPS '04: Proc. of the 18th IEEE International Parallel and Distributed Processing Symposium (IPDPS'04)*. IEEE Computer Society, 2004.

[16] L. Zhuo and V. Prasanna, "High performance linear algebra operations on reconfigurable systems," in *SC '05: Proc. of the 2005 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, 2005, p. 2.

[17] L. Zhuo and V. Prasanna, "Scalable hybrid designs for linear algebra on reconfigurable computing systems," *IEEE Transactions on Computers*, vol. 57, no. 12, pp. 1661–1675, 2008.

[18] L. Zhuo and V. Prasanna, "Hardware/software co-design for matrix computations on reconfigurable computing systems," in *IPDPS '07: Proc. of IEEE International Parallel and Distributed Processing Symposium*, 2007, pp. 1–10.

[19] V. B. Y. Kumar, S. Joshi, S. B. Patkar, and H. Narayanan, "FPGA based high performance double-precision matrix multiplication," in *VLSID '09: Proc. of the International Conference on VLSI Design*. IEEE Computer Society, 2009, pp. 341–346.

[20] "Xilinx floating point operator v5.0." [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/floating_point_ds335.pdf

[21] "Intel math kernel library." [Online]. Available: http://software.intel.com/en-us/intel-mkl/

[22] "Nvidia CUBLAS." [Online]. Available: http://developer.download.nvidia.com/compute/cuda/sdk/website/samples.html#simpleCUBLAS

[23] "Nvidia tesla C1060 computing processor." [Online]. Available: http://www.nvidia.com/object/product_tesla_c1060_us.html

[24] "Virtex-5 FPGA user gude," 2009. [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug190.pdf

[25] [Online]. Available: http://www.xilinx.com/

[26] "Electronic educational devices, operators manual: Wattsup? and wattsup? pro." [Online]. Available: https://www.wattsupmeters.com/secure/downloads/manual_rev_9_corded0812.pdf