On the Efficiency of Register File versus Broadcast Interconnect for Collective Communications in Data-Parallel Hardware Accelerators

***Abstract*—Reducing power consumption and increasing efficiency is a key concern for many applications. How to design highly efficient computing elements while maintaining enough flexibility within a domain of applications is a fundamental question. In this paper, we present how broadcast buses can eliminate the use of power hungry multi-ported register files in the context of data-parallel hardware accelerators for linear algebra operations. We demonstrate an algorithm/architecture co-design for the mapping of different collective communication operations, which are crucial for achieving performance and efficiency in most linear algebra routines, such as GEMM, SYRK and matrix transposition.**

We compare a broadcast bus based architecture with conventional SIMD and flat register files for these operations in terms of area and energy efficiency. Results show that a prototypical linear algebra core with very fast data movement abilities can achieve performance around 32 DP-GFLOPS for a typical linear algebra operation, while consuming around 0.7 Watts in standard 45nm CMOS technology. This is on par with a full-custom design and up to $50\times$ and $10\times$ better than CPUs and GPUs, respectively, in terms of power efficiency.

I. Introduction

Application-specific design of hardware accelerators can provide orders of magnitude improvements in power and area efficiency [1]. However, full-custom design is costly in many aspects. As we are entering the era of heterogeneous computing, a key question therefore becomes how to design specialized cores that maintain the efficiency of full custom hardware while providing enough flexibility to execute whole classes of coarse-grain operations.

Data-parallel and streaming processors, such as GPUs, have received widespread attention as an integral component for such heterogeneous architectures. Within this context, matrix operations, which are at the core of many high-performance computing problems, are often a prime target for acceleration. Matrix operations exhibit ample computational parallelism that can be relatively easily exploited. However, a crucial concern and often limiting factor is the efficient realization of data movements on a communication architecture that is able to optimally exploit locality, minimize overhead, effectively overlap computation with communication and hide communication latencies.

Linear algebra computations can be efficiently reduced down to a canonical set of Basic Linear Algebra Subroutines (BLAS), such as matrix-matrix and matrix-vector operations [2]. In previous work [3], [4], we examined the design of a proposed Linear Algebra Core (LAC). The LAC is based on broadcast communication among a 2D array of PEs. In this paper, we focus on the LAC's data-parallel broadcast interconnect and on showing how representative collective communication operations can be efficiently mapped onto this architecture. Such collective communications are a core component of many matrix or other data-intensive operations that often demand matrix manipulations.

We compare our design with typical SIMD cores with equivalent data parallelism and with L1 and L2 caches that amount to an equivalent aggregate storage space. To do so, we examine efficiency and performance of the cores for data movement and data manipulation in both GEneral matrix-matrix multiplication (GEMM) and SYmmetric Rank-K update (SYRK) operations. SYmmetric Rank-K and Rank-2K update (SYRK and SYR2K) are level-3 BLAS operations that we demonstrate because they need matrix transposition. The SYRK operation computes $C := C \pm AA^T$ with a rectangular matrix $A \in \mathbb{R}^{n\times m}$, updating only the lower triangular part of the symmetric matrix $C \in \mathbb{R}^{n\times n}$. To implement SYRK efficiently an architecture must be able to transpose matrices efficiently.

Our results show that the 2D architecture with broadcast interconnect is able to significantly reduce overhead for data movement. This leads to increased utilization, and hence performance and efficiency. We estimate the power consumption of key components of the LAC micro-architecture, and it should be possible to achieve a performance of 47 DP-GFLOPS/W in 15 GFLOPS/mm^2 for 45nm technology. This represents two orders of magnitude improvement over current CPUs and an order of magnitude improvement over GPUs.

A. 1D vs. 2D Architectures

Existing solutions for multimedia and matrix operations mostly focus on 1D [5] and 2D [6] arrangements of processing elements [7]. In this section, we use matrix multiplication to give a perspective of differences between 1D and 2D arrangement of PE for matrix computations.

In early FPGA designs with limited logic blocks on the chip, most of the approaches targeted an array arrangement of PEs that pipelines the data in and out of the PEs [8], [9]. Nowadays, with sufficient area on the chip, the design choice between a 1D or 2D arrangement of PEs becomes again valid. There are three major benefits of a 2D versus a 1D solution: scalability, addressing, and data movement. The 2D arrangement is proven to be scalable with regard to the ratio of problem size to local memory size for BLAS level operations [10]. Furthermore, address computations and data accesses in local stores of PEs become simpler with fewer calculations as compared to a 1D arrangement. This is especially true for more complicated algorithms. Finally, with 2D arrangements, different types of interconnects can be explored, yielding various types of algorithms for BLAS operations. A 2D arrangement of PEs facilitates operations like vector and matrix transpose that are used in matrix-vector multiplication and many other routines like SYRK, LU, and Cholesky factorization.

A taxonomy of matrix multiplication algorithms on 2D grids of PEs and their interconnect requirements is presented in [11]. The algorithms for matrix multiplication are based on three basic classes: Cannon's algorithms (roll-roll-multiply) [12], [13], Fox's algorithm (broadcast-roll-multiply) [14], [15], [11], and SUMMA (broadcast-broadcast-multiply) [16], [17]. Cannon's algorithm shifts the data in two of the three matrices circularly and keeps the third one stationary. Required initial and final alignment of the input matrices needs extra cycles and adds control complexity. In addition, a Torus interconnect is needed to avoid data contention. Fox's algorithms and its improvements broadcast one of the matrices to overcome alignment requirements. However, a shift operation is still required and such algorithms may show poor symmetry and sub-optimal performance. Finally, the SUMMA algorithm does not need any initial or post-computation alignment. The broadcast is a simple and uniform, single communication primitive. It does not have any bandwidth contention as in circular shifting. In addition, SUMMA is much easier to generalize to non-square meshes of processing units.

The flexibility of the SUMMA algorithm has made it the most practical solution for distributed memory systems [17] and FPGAs [18]. The SUMMA class of algorithms builds the basis for our design. A broadcast operation is an efficient way of data movement to achieve high performance in other BLAS and LAPACK operations. Furthermore, our cores are designed such that the cost and latency of broadcast operation does not add extra overhead.

B. Related Work

Matrix computations on general-purpose machines [19] and in recent years on GP-GPUs [20] have been studied extensively. Modern general-purpose GPUs (GP-GPUs) can be effectively used for matrix computations [21], [20] with throughputs of more than 360 double-precision GFLOPS when running many level-3 BLAS for large matrices, utilizing around 30-70% of the theoretical peak performance. However, in all cases, instruction handling, in core data movement and register file overheads limit efficiency.

Adding vector units to conventional processors has been a solution to increase efficiency of CPUs [22], [23]. Energy-efficiency potentials of vector accelerators for high performance computing systems are discussed in [24]. Three main limitations of conventional 1D vector architectures are known to be complexity of the central register file, implementation difficulties of precise exception handling, and expensive on-chip memory [25]. A detailed review of SIMD multimedia extensions and their bottlenecks are presented in [5], [26]. Associated costs are amplified by the fact that in each step a complete vector has to be transferred through multiple ports of a register file, wide wires, and complex point-to-point interconnects such as crossbars. The details of scalar operand network and bypass paths for ILP workloads are discussed in [27], [28].

Over the years, many other parallel architectures for high-performance computing have been proposed and in most cases benchmarked using GEMM as a prototypical application. Systolic arrays were popularized in the 80s [29], [30]. Different optimizations and algorithms for matrix multiplication and more complicated matrix computations are compared and implemented on both 1D [31], [32] and 2D systolic arrays [31], [7], [33]. In [34], the concept of a general systolic array and a taxonomy of systolic array designs is discussed. Systolic arrays pipeline data and have one-sided interfaces. As discussed earlier, the broadcast concept performs better for GEMM types of algorithms.

With increasing memory walls, recent approaches have brought the computation units closer to memory, including hierarchical clustering of such combined tiles [35], [36]. Despite such optimization, utilizations for GEMM range from 60% down to less than 40% with increasing numbers of tiles. Instead of a shared-memory hierarchy, the approach in [37] utilizes a dedicated network-on-chip interconnect with associated routing flexibility and overhead. It only achieves around 40% utilization for matrix multiplication. Finally, ClearSpeed CSX700 is an accelerator that specifically targets scientific computing with BLAS and LAPACK library facilities. It delivers up to 75 DGEMM GFLOPS at 78% of its theoretical peak [38].

As utilization numbers indicate, inherent characteristics of data paths and interconnects coupled with associated instruction inefficiencies in general-purpose architectures make it difficult to exploit fully all available parallelism and locality. By contrast, while we will build on the SIMD and GPU concept of massive parallelism, we aim to provide a natural extension that leverages the specifics of matrix operations.

Finally, specialized realizations of matrix computation routines on FPGAs have been explored, either standalone [39] or in combination with a flexible host architecture [40]. Such approaches show promising results [41], [42], but are limited by inefficiencies in area, performance, and power due to programmable routing and logic overheads.

II. Architectures

In the following, we describe our proposed architecture and briefly review three other models that we will explore in this paper: a flat register file with full access flexibility, a 1D wide SIMD, and a 2D array of SIMD units.

A. LAC Architecture

The LAC proposed in [3] and illustrated in Fig. 1 consists of a 2D array of $n_r \times n_r$ processing elements (PEs), each of

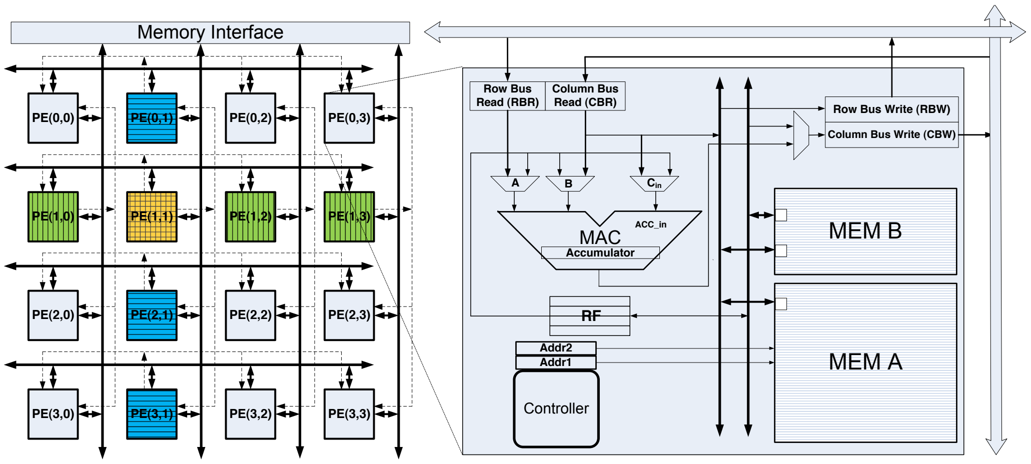

Fig. 1. Core architecture. The highlighted PEs on the left illustrate the second iteration (rank-1 update $p = 1$) of a GEMM operation, where roots owning the current colum and row (the PEs in the second column and row) broadcast elements of A and B over the busses for other PEs to process.

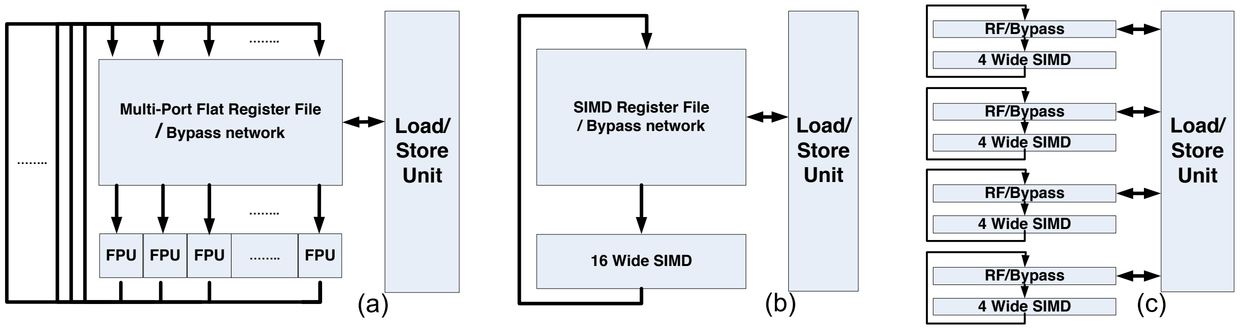

Fig. 2. Flat and SIMD register file organizations.(a) Flat flexible multi-ported register file with 16 FPUs, (b) 1D 16-wide SIMD register file, (c) Four 1D 4-wide SIMD register files

which has a MAC unit with a local accumulator, local storage, simple distributed control, and bus interfaces to communicate data within rows and columns.

Details of the PE-internal architecture are shown in Fig. 1-(left). At the core of each PE is a MAC unit to perform the inner dot-product computations central to almost all level-3 BLAS operations. Apart from preloading accumulators with initial values, all accesses to elements of a $n_r \times n_r$ matrix being updated are performed directly inside the MAC units, avoiding the need for any register file or memory accesses.

We utilize pipelined MAC units that can achieve a throughput of one MAC operation per cycle [43]. Local storage in each PE consists of a bigger single-ported and a smaller dual-ported memory. Typically in dense linear algebra problems, access patterns are predictable and in most cases sequential, and there is no need for complex caching schemes. LAC control is distributed and each PE has a basic state machine that drives a predetermined, hardcoded sequence of communication, storage and computation steps for each supported BLAS operation. The basic state machine in each PE requires two address registers, one loop counter and less than 10 states per BLAS operation.

In our design, we connect PEs by horizontal and vertical broadcast buses. The interconnect is realized as simple, data-only buses that do not require overhead for address decoding or complex control. This interconnect is specifically designed to efficiently realize all collective communications, such as broadcast or transposition, necessary to support the execution of level-3 BLAS operations. While other architectures waste cycles and instruction to move data to their desired destination, the LAC architecture can inherently and transparently overlap computation, communication, and transposition.

B. Register File Architectures

In the following, we discuss details of flat, 1D SIMD, and 2D SIMD register file organizations with L1 and L2 data caches that we will use to compare with the LAC in terms of efficiency. In all cases, we only consider data movement between the memory hierarchy on the core.

a) Flat register file: A flat register file architecture (Fig. 2(a)) uses a typical multi-ported register file. All ALUs are connected via dedicated ports and can access any of the registers. At the expense of power consumption and area, this allows for complex data movement between different ALUs through the common register file. However, this flexibility is not necessary for a typical matrix multiplication. For this operation, either 1D or 2D SIMD solutions can be applied directly. The benefit of a flat register file is in its flexibility for data handling of more complex operations like SYRK, matrix transpose, and matrix-vector multiplication.

b) 1D SIMD: A 1D-SIMD organization (Fig. 2(b)) performs the same operation for all ALUs with wide vector inputs and outputs. The number of ports is independent of the SIMD width or the number of ALUs. As such, data movement for

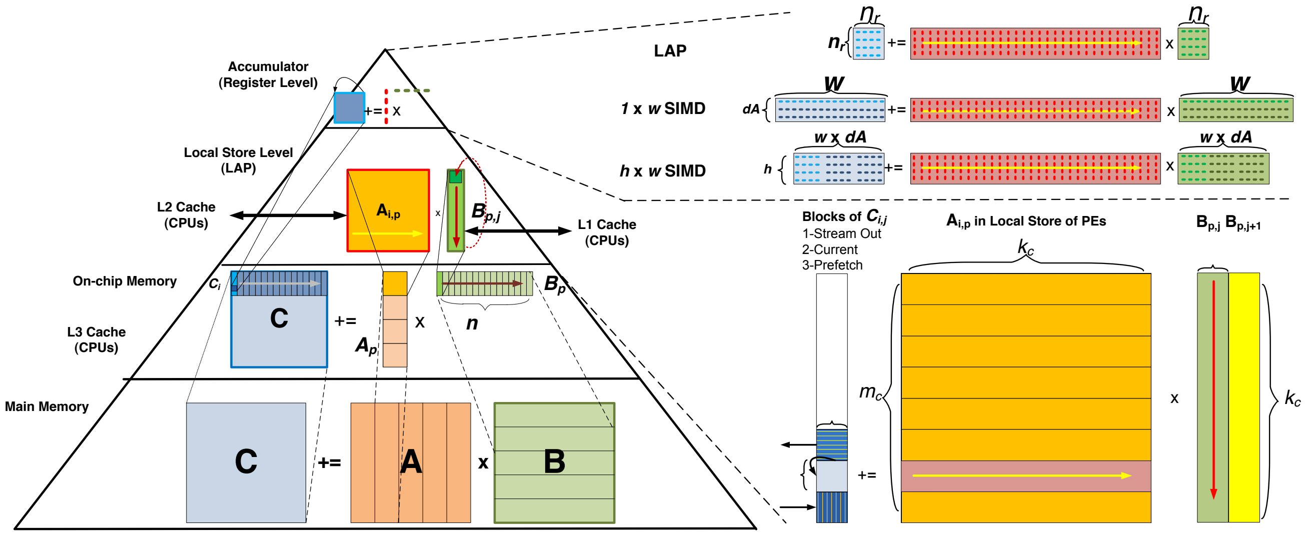

Fig. 3. Matrix multiplication mapping across layers of memory hierarchy on all four architectures. The difference is in low level inner kernels.

complex matrix manipulations is not simple. Typical SIMD architectures provide special shuffling instructions that make the transposing of matrices possible [44]. But this incurs penalties of several instructions and CPU clock cycles.

c) 2D array of SIMD units: A 2D SIMD array (Fig. 2(c)) contains a few shorter 1D-SIMD units, each having its own register file connected to a central load/store unit. Having a larger number of shorter SIMD units provides more flexibility for performing different operations in the different SIMD units. The data movement flexibility depends on the architecture of the bypass network. There might be a global interconnect and bypass network between the register files of different SIMD units. This facilitates data movement and is ideal for blocked algorithms.

III. Algorithm Mapping

In the following, we will describe the mapping of GEMM, SYRK and matrix transposition operations onto the four considered architectures. These algorithms are representative of the communication complexity of level-3 BLAS operations. It is important to observe that other level-3 BLAS operations require similar computations and data movements [19].

A. GEMM

a) LAC: A matrix multiplication can be performed iteratively as a sequence of smaller steps that can be efficiently mapped onto our hardware core. Let C, A, and B be $n_r \times n_r$, $n_r \times k_c$, and $k_c \times n_r$ matrices, respectively, then $C \mathrel{+}= AB$ can be computed as a loop of so-called rank-1 updates $C \mathrel{+}= \sum_{p=0}^{k_c} A_p B_p$, where C is repetitively updated with the product of a column of A, A_p, and a row of B, B_p, respectively. Let us assume that the $n_r \times k_c$ matrix A and $k_c \times n_r$ matrix B are distributed to the local PE memories in a 2D cyclic round-robin fashion, much like distributing matrices on distributed memory architectures. In other words, elements $\alpha_{i,j}$ and $\beta_{i,j}$ of matrices A and B are assigned to PE $(i \mod n_r, j \mod n_r)$. Also, element $\gamma_{i,j}$ of matrix C is assumed to reside in an accumulator of PE (i, j). Then, a simple and efficient algorithm for performing this operation among the PEs is: For $p = 0, \ldots, k_c - 1$, broadcast the pth column of A within PE rows and the pth row of B within PE columns, after which a local MAC operation on each PE updates the local element of C. The operation of the LAC for one iteration of the GEMM algorithm is highlighted on the left in Figure 1. Overall, this setup allows the LAC to keep elements of C at all times only resident in the accumulators, amortizing movement of C in and out of the core over a large number of updates. By contrast, matrices A and B are only read, and their distributed storage enables efficient broadcasts in each iteration.

In accordance with the blocking at the upper memory levels (Fig. 3), we assume that each core locally stores a larger $m_c \times k_c$ block of A, a $n_r \times n_r$ sub-block of C and a $k_c \times n_r$ panel of B (replicated across PEs). GEMM computations for larger matrices can be mapped onto our LAC by multiplying successive $n_r \times k_c$ panels of A ($m_c \times k_c$ block of A) by the same $k_c \times n_r$ panel of B to update successive $n_r \times n_r$ blocks (for a complete $k_c \times n_r$ panel) of C. A complete $m_c \times k_c$ block of A is stored distributed among the PEs. At the next level, with n being the dimension of original matrix C, n/n_r successive panels of B are multiplied with the same, resident $m_c \times k_c$ block of A to update successive $k_c \times n_r$ panels of a larger $m_c \times n$ panel of C. While individual updates are running, we overlap computation with communication to stream the next panels of B and C into the LAC. Higher levels of blocking for memory hierarchy and details about the mapping of GEMM onto the LAC can be found in [3].

b) Register file architectures: Here we use the same notation as was used for describing the mapping of GEMM on the LAC. Matching the blocking at upper memory levels in the LAC, the larger $m_c \times k_c$ block of A is stored in the L2 cache. Except for block sizes as indicated in Fig. 3, the GEMM algorithms are almost the same as the one described for the LAC. Inner kernels perform successive rank-1 updates on a block of C stored in the register file. However, since the FPUs incur latency in performing accumulations/additions (Fig. 4),

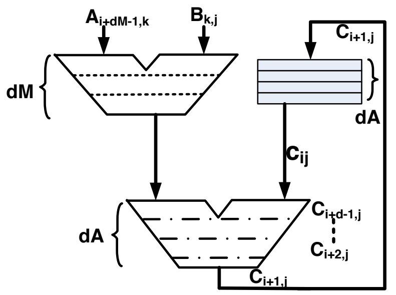

Fig. 4. Multiply-accumulate in register file architectures with pipelined FPUs.

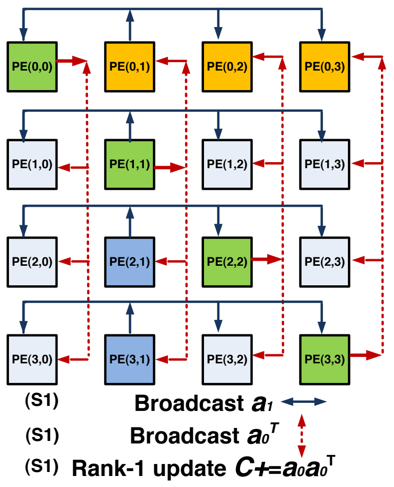

Fig. 5. Second iteration of a 4×4 SYRK on the LAC.

block sizes are increased. This allows FPUs to hide addition latencies and alternate between performing accumulations on different elements of C.

Three major factors determine the dimensions of a $g \times f$ block of C held in the register file and, accordingly, the width f of the panel of B stored in the L1 cache. Depending on the architecture, as shown in Fig. 3, these factors are the width w and height h of the SIMD array and the number of pipeline stages dA in the floating-point adder (see Fig. 4). While the LAC has a broadcast mechanism built-in, SIMD architectures need to perform explicit move instructions. This is done by replicating each element of A into a wide SIMD register to multiply it by different elements of B using a single SIMD instruction.

Following the same algorithm as in the LAC, GEMM computations for larger matrices can be performed by multiplying successive $g \times k_c$ panels of A ($m_c \times k_c$ block of A) by the same $k_c \times f$ panel of B to update successive $g \times f$ blocks (for a complete $k_c \times f$ panel) of C. A complete $m_c \times k_c$ block of A is stored in the L2 cache. At the next level, n/f successive panels of B are multiplied with the same, resident $m_c \times k_c$ block of A to update successive $k_c \times f$ panels of a larger $m_c \times n$ panel of C.

B. SYRK

a) LAC: The SYRK operation computes $C := C + AA^T$ with a rectangular matrix $A \in \mathbb{R}^{n \times m}$, updating only the lower triangular part of the symmetric matrix $C \in \mathbb{R}^{n \times n}$.

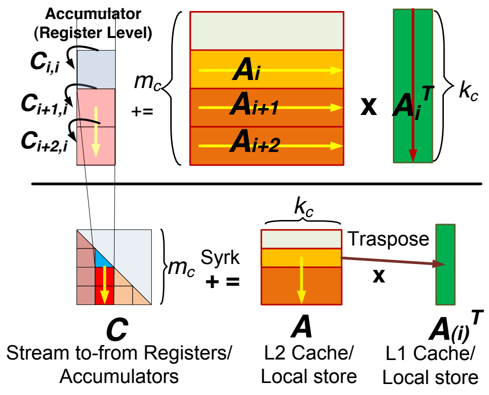

Fig. 6. SYRK hierarchy on all architectures.

To compute the SYRK of a $n_r \times n_r$ submatrix of C stored in the accumulators of the LAC from a $n_r \times k_c$ sub-matrix of A, three different operations take place in the same cycle in each iteration. Fig. 5 illustrates the second ($i = 1$) iteration of a SYRK operation.

The ith column of PEs broadcasts the values of the ith column of A, a_i, across the row busses, where the PEs in each row keep a copy of these values in their register file for use in the next iteration. At the same time, the values a_{i-1} from the previous iteration are transposed along the diagonal PEs by broadcasting them over the column busses. Hence, all PEs now have copies of elements of a_{i-1} and a_{i-1}^T, and a rank-1 update is performed to compute $C := C + a_{i-1} \times a_{i-1}^T$. The a_{i-1}^T is also kept in $(i-1)$th row of PEs to store A^T. This is repeated for $i = 0, \ldots, k_c$ cycles.

A bigger SYRK problem for C of size $m_c \times m_c$ and A of size $m_c \times k_c$ can be blocked into smaller subproblems using a lower order SYRK to update the diagonal $n_r \times n_r$ lower triangular blocks of C and produce transpose of the corresponding $n_r \times k_c$ panels of A in a single iteration. As before, the complete $m_c \times k_c$ block of A is stored distributed among the PEs. As shown in Fig. 6, after computing the diagonal block $C_{ii} = A_i A_i^T$, the produced panel A_i^T and the remaining row panels $A_{k>i}$ of A are used to update other blocks $C_{ki} = A_k A_i^T$ of C with $k > i$. Most of the computations are thereby cast into GEMM operations using the produced panel of A^T and the remaining panels of A.

b) Register file architectures: In the same way as for the LAC, SYRK problems can be blocked using a lower order SYRK to update the diagonal lower triangular blocks of C and produce a transpose of the corresponding panels of A (see Fig. 6). Here, a row panel of matrix A_i is brought into the L1 cache and being replaced by its transpose while also computing $C_{ii} = A_i A_i^T$ and subsequently updating all $C_{ki} = A_k A_i^T, k > i$. As before, each step in a SYRK computation maps down to a matrix transposition and a series of matrix multiplications. The transpose operation is straightforward in a flat, multi-ported register file due to full flexibility of data exchange and register access. It can provide transposed data for a certain small block size that fits into the register file without wasting FPU cycles, where the transpose is done by directly routing data from different places to the right FPU.

TABLE I
GEMM MEMORY HIERARCHY REQUIREMENTS.

	Flat Register	1D SIMD	Array of SIMD
Regs for A	$2(dA)$	$2wdA$	$(h+1)wdA$
Regs for B	$2(w)$	$2w$	$2w(dA)$
Regs for C	$2(wdA)$	$2wdA$	$2w(hdA)$
Regs Total	$2(wdA+w+dA)$	$4wdA+2w$	$3wdA(h+1)$
RF access A	2	$2w$	$hw(1+1/dA)$
RF access B	$w+w/dA$	$w(1+1/dA)$	$2hw$
RF access C	$2w(1+1/k_c)$	$2w(1+1/k_c)$	$2hw(1+dA/k_c)$
Ports for A	2 R-W	$2w$ R-W	$2hw$ R-W
Ports for B	$2w$ R-W	$2w$ R-W	$2hw$ R-W
Ports for C	$3w$ R-W-D	$3w$ R-W-D	$3hw$ R-W-D
Ports Total	5w+2	7w	7hw
L1 access A	*	1+1	$h/da(1+1)$
L1 access B	*	$w+w/k_c$	$hw(1+1/k_c)$
L1 access C	*	$2w/k_c$	$2hw/k_c$
L2 access A	*	$1+w/n$	w/k_c
L2 access B	*	w/k_c	hw/k_c
L2 access C	*	$2w/k_c$	$2hw/k_c$

TABLE II
SYRK REQUIREMENTS.

	Flat Register	1D SIMD	Array of SIMD
Regs for A	$l^2(1+dA)$	$2w^2$	$2hw^2$
Regs for A^T	0	w^2+w	hw^2+hw
Regs for C	$l^2(1+dA)$	w^2	hw^2
Regs Total	$2l^2(1+dA)$	$4w^2+w$	$4hw^2+4hw$
RF access A	$2l$	$3w+w$	$3hw+hw$
RF access A^T	$2l$	$2w$	$2hw$
RF access C	$2w$	$2w+1$	$2hw+1$
RF access Total	$4l+2w$	$8w+1$	$8hw+1$
Ports for A	$2l$:R-D	$4w$:2R-2W	$4hw$:2R-2W
Ports for A^T	$2l$:W-D	$2w$:R-D	$2hw$:R-D
Ports for C	$2w$:R-W	$2w$:R-W	$2hw$:R-W
Ports Total	$4l+2w$	$8w$	$8hw$
L1 access A	$2l$	$w^2/(l(w \log w))$	$hw^2/(l(w \log w))$
L1 access A^T	$2l$	$w^2/(l(w \log w))$	$hw^2/(l(w \log w))$
L1 access C	$2w/k_c$	$2w/k_c \log w$	$2hw/k_c \log w$
L2 access A	$2l$	$1/l(w \log w)$	$h/l(w \log w)$
L2 access A^T	$2l$	$1/l(w \log w)$	$h/l(w \log w)$
L2 access C	$2w/k_c$	$2w/k_c \log w$	$2hw/k_c \log w$

TABLE III
MAXIMUM (MINIMUM) CYCLE COUNTS OF COMMUNICATIONS ON LAC.

Configuration	Point to Point $2n$	Broadcast $2n$	Transpose $n \times n$	Rank-1 Update
Broadcast	$2n(2n/n_r)$	$2n(2n/n_r+n)$	n^2/n_r	1
Mesh	$2n(n/n_r)$	$2n(2n/n_r+n)$	$n^2/2n_r$	1

Other SIMD organizations have to waste core cycles to arrange the data in the desired order without using the FPUs by using special shuffling and move instructions [44].

IV. EXPERIMENTS AND RESULTS

In previous work, we have developed both cycle-accurate simulation and analytical power and performance models of the LAC. In this work, we develop similar analytical models for selected register file organizations of SIMD units [23].

To provide a fair comparison, we chose similar system configurations for LAC and register file architectures. In all cases, the number of FPUs is 16 and the aggregate storage size is over 256 KBytes of space. We chose the size of L1 caches to be 32 Kbytes, which is a typical L1 size for state of the art cores. This size is also sufficient for the blocking of matrix operation algorithms that we target. The L2 cache is chosen to be 256 Kbytes. This is equal to the aggregate on-chip memory size of PEs in the LAC and matches L2 sizes in recent architectures.

The studied architecture configuration are: a 1×16 wide SIMD, a 16-FPU flat register file, a group of 4×4-wide SIMD units, and a 4×4 LAC. We assume that all architectures use the same type of FPUs and we can hence ignore the power and area consumption of FPUs. We compare the storage, area, and power requirements of these architectures for GEMM and SYRK operations. We chose GEMM to profile the coarse-grain data movement capabilities of an architecture, while SYRK indicates their fine-grain data manipulation abilities.

A. Design Space Exploration

We performed an analytical study to derive the required memory and register file storage sizes, number of register file ports, accesses and bandwidth requirements for best possible performance of a given architecture. To do so, we used the proposed algorithms in Section III and calculated the register file, L1 and L2 configurations. We assume that there are no restrictions in the core architecture or the load/store unit that limit the amount of data that can be transferred between L1 cache and the core. We also assume that the inner kernel hides all cache access latencies, and that the problem fits into L1 and L2 caches. With these assumptions, we aim for peak possible performance in GEMM and SYRK.

The system parameters include the SIMD width 'w', accumulator delay 'dA', SIMD array height 'h' and the L1 cache line size 'l'. Analytical results include the number of floating-point registers required to store the necessary elements of each input matrix. The number of accesses for each of the input matrices shows the average number of floating-point words read or written from/to the register file, L1 or L2 cache per clock cycle. The port types 'R', 'W', and 'D' stand for "Read", "Write" and "Dual R/W", correspondingly. For SIMD architectures, each entry in the table shows the number of single floating-point word registers, accesses or ports. However, all accesses happen over wide ports for transfers of complete SIMD vectors with w words each. To determine the number of wide vector registers, accesses or ports, the entries in the table have to be divided by 'w'.

Results summarized in Tables I and II indicate what an ideal, minimum configuration of each architecture should be to achieve peak performance with the least overhead, e.g. for shuffling instructions. For example, if one designs an architecture with the goal of achieving high utilization in linear algebra applications like GEMM, the parameters of all memory layers and requirements for utilizing the FPUs efficiently are shown in Table I.

a) Matrix Multiplication: We used the algorithm described in Section III-A for mapping of GEMM on all four different architectures. The details of the memory subsystem are presented in Table I. The flat register file can be configured to implement either the 1D-SIMD or 2D-SIMD solution. We used the wide 1D-SIMD solution.

b) SYRK: The SYRK operation can be viewed as combination of the GEMM and transpose operations. Most of the operations are cast in terms of GEMM kernels, but the SYRK

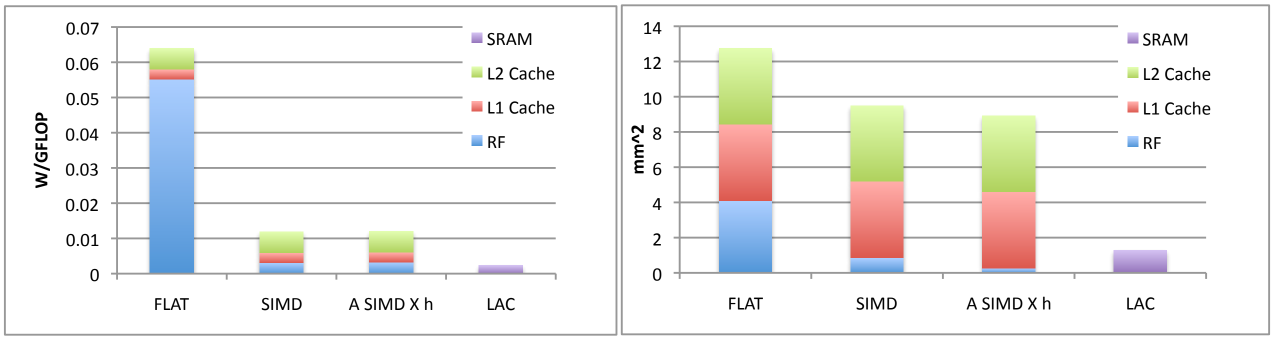

Fig. 7. Different platforms performing GEMM with 16 FPUs: Register file based platforms with 256KB L2 and 32KB L1 caches, LAC with 288KB SRAM. Left: Inverse power efficiency, Right: Total area.

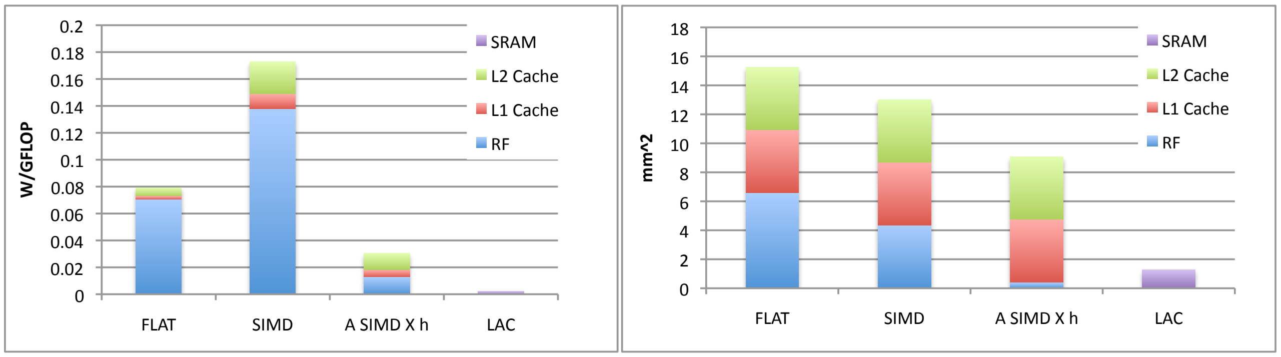

Fig. 8. Different platforms performing SYRK with 16 FPUs: Register file based platforms with 256KB L2 and 32KB L1 caches, LAC with 288KB SRAM. Left: Inverse power efficiency, Right: Total area.

part needs fine-grain data manipulation for transposing the input matrix in its critical path. We assume that the behavior of the GEMM part of SYRK can be estimated from the GEMM studies above. In SIMD solutions, the problem is compute bounded. Since the cores do not transpose matrices efficiently, the overall SYRK performance is limited by the matrix transposition behavior. With SIMD units, it takes "$w \log w$" cycles to transpose a $w \times w$ block of data with a bypass network as in current SSE architectures. By contrast, the LAC's broadcast buses facilitate very efficient transposition with n_r cycles for a $n_r \times n_r$ input matrix.

c) Other collective communication operations: Table III shows the cycle count for representative collective communications on the LAC. Based on the location of source and destination PEs, we derived minimum and maximum cycle counts for these operations. We can improve the performance of these communication operations by modifying the interconnect and adding mesh-like peer-to-peer connections. As shown in Table III, this modification could double the performance of matrix transpose and point-to-point communications. More details about trade-offs of various interconnect options for coarse-grained architectures based on a 2D arrangement of ALUs can be found in [45].

B. Area and Power Estimation

To estimate power and area of different parts of the memory subsystem, we applied the analytical configurations derived in the previous section to CACTI-6.5 [46]. Cache line sizes are chosen to be 64 bytes, which is equal to eight double-precision floating-point numbers. We set the associativity to four, which is less power hungry than current 8-way state of the art caches. We use an inverse power efficiency metric of (W/GFLOP) to show a power breakdown of different parts of the memory subsystem. The operation frequency is assumed to be 1 GHz, and for simplicity, we assumed that adder latency dA is 1 cycle. We also neglected the power consumption of the bypass network for register file based architectures.

We assumed a fixed maximum power consumption for L1 and L2 caches. The register file organization is designed to sustain the core data transfer requirements, but there are no extra ports beyond that. For this study, we examined multiple possible configurations for caches and register files and chose the most efficient one. According to CACTI, the optimum configuration turned out to be a single-banked cache.

Fig. 7 demonstrates that SIMD architectures can very effectively reduce the power consumption compared to flat register files. The 1D SIMD consumes the least power among the other two configurations, but the $4{\times}4$-wide array of SIMD requires less area. The flexibility of flat register files is of no use for GEMM.

Fig. 8 presents the SYRK case at the other end of the spectrum, where flexibility becomes an important factor. Even when considering a bypass network that is able to shuffle words, a wide SIMD architecture requires a large number of cycles and instructions to complete the transpositions. The $4{\times}4$-wide array of SIMD performs much better than both 1D SIMD and flat register file. This is because transposing a smaller array turns to be simpler and faster than a wide SIMD. Also, the array of SIMD consumes much less power compared to a flat register file.

In all cases, the LAC removes the overhead of L1 and L2 caches and has less power and area consumption while it is more effective to perform broadcast and transpose operations necessary for GEMM and SYRK.

V. Conclusion

This paper presents the algorithm/architecture co-design of a linear algebra core, with an emphasis on data movement and memory subsystem. We presented detailed analysis of four different architectures. We presented power estimations and compared our core with the other register file organizations for selected matrix operations. Power estimation results show that our core can provide orders of magnitude improved efficiencies compared to other architectures. Our analysis clearly shows the fundamental architectural tradeoffs for efficient execution of linear algebra computations.

References

[1] R. Hameed et al., "Understanding sources of inefficiency in general-purpose chips," *ISCA '10*, 2010.

[2] J. Dongarra et al., "A set of level 3 basic linear algebra subprograms," *ACM Trans. Math. Soft.*, vol. 16, no. 1, 1990.

[3] "Ommited for blind review."

[4] "Ommited for blind review."

[5] M. Hassaballah *et al.*, "A review of simd multimedia extensions and their usage in scientific and engineering applications," *The Computer Journal*, vol. 51, 2008.

[6] F. Bouwens *et al.*, "Architectural exploration of the adres coarse-grained reconfigurable array," in *ARC 2007. LNCS*. Springer.

[7] H. Jagadish *et al.*, "A family of new efficient arrays for matrix multiplication," *Computers, IEEE Transactions on*, vol. 38, no. 1, pp. 149 – 155, 1989.

[8] V. Kumar and Y. Tsai, "On synthesizing optimal family of linear systolic arrays for matrix multiplication," *Computers, IEEE Transactions on*, vol. 40, no. 6, pp. 770 – 774, 1991.

[9] L. Zhuo et al, "Scalable and modular algorithms for floating-point matrix multiplication on reconfigurable computing systems," *IEEE Trans. on Parallel and Distributed Systems*, vol. 18, no. 4, 2007.

[10] V. Eijkhout, *Introduction to High Performance Scientific Computing*. www.lulu.com, 2011.

[11] J. Li, "A poly-algorithm for parallel dense matrix multiplication on two-dimensional process grid topologies," *Citeseer*, Jan 1996.

[12] L. E. Cannon, "A cellular computer to implement the kalman filter algorithm," Ph.D. dissertation, Bozeman, MT, USA, 1969.

[13] K. K. Mathur *et al.*, "Multiplication of matrices of arbitrary shape on a data parallel computer," *Parallel Computing*, vol. 20, no. 7, pp. 919 – 951, 1994.

[14] G. C. Fox *et al.*, *Solving problems on concurrent processors. Vol. 1: General techniques and regular problems*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1988.

[15] J. Choi *et al.*, "Pumma: Parallel universal matrix multiplication algorithms on distributed memory concurrent computers," *Concurrency: Practice and Experience*, vol. 6, no. 7, pp. 543–570, 1994.

[16] R. C. Agarwal *et al.*, "A high-performance matrix-multiplication algorithm on a distributed-memory parallel computer, using overlapped communication," *IBM J. Res. Dev.*, vol. 38, pp. 673–681, November 1994.

[17] R. van de Geijn *et al.*, "SUMMA: Scalable universal matrix multiplication algorithm," *Concurrency: Practice and Experience*, vol. 9, 1997.

[18] Y. Dou *et al.*, "64-bit floating-point FPGA matrix multiplication," ser. FPGA '05.

[19] K. Goto et al, "High-performance implementation of the level-3 BLAS," *ACM Trans. Math. Softw.*, vol. 35, no. 1, pp. 1–14, 2008.

[20] V. Volkov and J. Demmel, "Benchmarking GPUs to tune dense linear algebra," *SC 2008*, 2008.

[21] G. Tan *et al.*, "Fast implementation of DGEMM on fermi GPU," *SC '11*.

[22] R. Espasa *et al.*, "Vector architectures: past, present and future," ser. ICS '98.

[23] S. Rixner *et al.*, "Register organization for media processing," *HPCA-6*, pp. 375 – 386, 2000.

[24] C. Lemuet *et al.*, "The potential energy efficiency of vector acceleration," ser. SC '06, 2006.

[25] C. Kozyrakis *et al.*, "Overcoming the limitations of conventional vector processors," *ISCA'03*, pp. 399 – 409, 2003.

[26] D. Talla *et al.*, "Bottlenecks in multimedia processing with simd style extensions and architectural enhancements," *Computers, IEEE Transactions on*, 2003.

[27] M. Taylor *et al.*, "Scalar operand networks: On-chip interconnect for ilp in partitioned architectures," in *ISCA'02*.

[28] P. Gratz *et al.*, "Implementation and evaluation of a dynamically routed processor operand network," in *NOCS 2007*.

[29] H. Kung, "Why systolic architectures?" *Computer*, vol. 15, no. 1, pp. 37 – 46, 1982.

[30] S. Kung, "Vlsi array processors," *ASSP Magazine, IEEE*, vol. 2, no. 3, pp. 4 – 22, jul 1985.

[31] R. Urquhart *et al.*, "Systolic matrix and vector multiplication methods for signal processing," *IEEE Communications, Radar and Signal Processing,*, vol. 131, no. 6, pp. 623 – 631, 1984.

[32] V. Kumar *et al.*, "Synthesizing optimal family of linear systolic arrays for matrix computations," *ICSA '88*, pp. 51 – 60, 1988.

[33] T. Lippert et al., "Hyper-systolic matrix multiplication," *Parallel Computing*, 2001.

[34] K. Johnson *et al.*, "General-purpose systolic arrays," *Computer*, vol. 26, no. 11, pp. 20 – 31, 1993.

[35] C. Takahashi et al., "Design and power performance evaluation of on-chip memory processor with arithmetic accelerators," *IWIA2008*, 2008.

[36] J. Kelm et al., "Rigel: an architecture and scalable programming interface for a 1000-core accelerator," *ISCA '09*, 2009.

[37] S. Vangal et al., "An 80-tile sub-100-w teraflops processor in 65-nm cmos," *IEEE J. of Solid-State Circuits*, vol. 43, no. 1, 2008.

[38] "CSX700 Floating Point Processor," ClearSpeed Technology Ltd, Datasheet 06-PD-1425 Rev 1, 2011.

[39] L. Zhuo and V. Prasanna, "High-performance designs for linear algebra operations on reconfigurable hardware," *IEEE Trans. on Computers*, vol. 57, no. 8, 2008.

[40] G. Kuzmanov et al, "Floating-point matrix multiplication in a polymorphic processor," *ICFPT 2007*, pp. 249 – 252, 2007.

[41] M. Parker, "High-performance floating-point implementation using FPGAs," in *MILCOM*, 2009.

[42] J.-W. Jang *et al.*, "Energy- and time-efficient matrix multiplication on FPGAs," *IEEE Trans on VLSI Systems,*, vol. 13, no. 11, 2005.

[43] S. Vangal et al., "A 6.2-GFlops floating-point multiply-accumulator with conditional normalization," *IEEE J. of Solid-State Circuits*, vol. 41, no. 10, 2006.

[44] J. Leiterman, *32/64-bit 80x86 Assembly Language Architecture*. Wordware Publishing Inc., 2005.

[45] A. Lambrechts *et al.*, "Energy-aware interconnect optimization for a coarse grained reconfigurable processor," in *VLSID 2008*.

[46] N. Muralimanohar *et al.*, "Architecting efficient interconnects for large caches with cacti 6.0," *IEEE Micro*, vol. 28, pp. 69–79, January 2008.