This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2014.2315627, IEEE Transactions on Computers

Algorithm, Architecture, and Floating-Point Unit Codesign of a Matrix Factorization Accelerator

Ardavan Pedram, Andreas Gerstlauer, and Robert A. van de Geijn

Abstract—This paper examines the mapping of algorithms encountered when solving dense linear systems and linear least-squares problems to a custom Linear Algebra Processor. Specifically, the focus is on Cholesky, LU (with partial pivoting), and QR factorizations and their blocked algorithms. As part of the study, we expose the benefits of redesigning floating point units and their surrounding data-paths to support these complicated operations. We show how adding moderate complexity to the architecture greatly alleviates complexities in the algorithm. We study design trade-offs and the effectiveness of architectural modifications to demonstrate that we can improve power and performance efficiency to a level that can otherwise only be expected of full-custom ASIC designs. A feasibility study of inner kernels is extended to blocked level and shows that, at block level, the Linear Algebra Core (LAC) can achieve high efficiencies with up to 45 GFLOPS/W for both Cholesky and LU factorization, and over 35 GFLOPS/W for QR factorization. While maintaining such efficiencies, our extensions to the MAC units can achieve up to 10%, 12%, and 20% speedup for the blocked algorithms of Cholesky, LU, and QR factorization, respectively.

Index Terms—Low-power design, Energy-aware systems, Performance analysis and design aids, Floating-point arithmetic, Matrix decomposition, Special-purpose hardware, LU factorization, Partial pivoting, QR factorization, Cholesky factorization

1 Introduction

Matrix factorizations are typically the first (and most compute intensive) step towards the solution of dense linear systems of equations or linear least-squares problems [1], which are fundamental to scientific computations. Within the dense linear algebra domain, a typical computation can be blocked into sub-computations like GEneral Matrix-matrix Multiplication (GEMM) that contain most of the computational load, are highly parallelizable, and can be mapped very efficiently to accelerators. Typical accelerator designs remove unnecessary overheads in general purpose architectures and use more fine-grain compute engines instead. As such, accelerator architectures are less flexible, but very efficient, both in terms of area and power consumption [2].

For more complicated algorithms, such as matrix factorizations, many current approaches use heterogeneous solutions [3], [4]. Often, in cases where the problem size is large enough, only the simplest sub-computations of these algorithms, e.g. GEMM and other matrix-matrix operations are performed on the accelerator. Other, more irregular computations, which are added to the algorithm to overcome floating point limitations or which would require complex hardware logic to exploit fine grain parallelism, are performed by a general purpose host processor.

- *Ardavan Pedram, and Andreas Gerstlauer are with the Department of Electrical and Computer Engineering, The University of Texas at Austin, Austin, Texas.*
  E-mail: ardavan@utexas.edu, gerstl@ece.utexas.edu
- *Robert van de Geijn is with the Department of Computer Science, The University of Texas at Austin, Austin, Texas*
  E-mail:rvdg@cs.utexas.edu

This research was partially sponsored by NSF grant CCF-1218483.

The problem with heterogeneous solutions is the overhead for communicating back and forth between the accelerator and the host processor. Even when integrated on the chip, data often has to be moved all the way to off-chip memory in order to perform transfers between (typically) incoherent address spaces. While the CPU could be used to perform other tasks efficiently, it is wasting cycles synchronizing with the accelerator and copying data. Often, the accelerator remains idle waiting for the data that is on the critical path to be processed by the CPU. This is particularly noticeable for computations with small pieces of data. Therefore, the compute kernel has to be large enough to amortize the cost of data movements. Otherwise, it is more efficient to perform it on the host processor alone.

We propose that, instead of only focusing on the most repeated kernels like GEMM, accelerators can be designed to natively and efficiently support the irregular parts of an application. We study such an approach for three matrix factorization algorithms: Cholesky, LU (with partial pivoting), and QR factorizations. Our solution is to allow architectural changes to the accelerator design in order to reduce complexity directly in the algorithm whenever possible. Thus, the solution is to exploit algorithm/architecture co-design.

In previous work [5], we started from a base design of a Linear Algebra Core (LAC) that is highly optimized for GEMM and level-3 BLAS [6], [7]. We showed that focusing on complex kernels and providing logic extensions for the Floating-Point Units (FPUs) of the LAC can significantly improve the performance and reduce the complexity and power consumption of the inner kernels of matrix factorizations. The resulting architecture avoids the need for running complex computations on a general-purpose host processor.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2014.2315627, IEEE Transactions on Computers

Our previous work focused on improvements for the inner kernels of Cholesky factorization, LU factorization (with partial pivoting), and the vector-norm in QR factorization. In this paper, we extend our studies to show how much improvements the algorithm/architecture codesign brings to the low-level inner kernel of QR factorization and whole, overall blocked algorithms of all three matrix factorizations. Results show that improvements of intricate low-level kernels have noticeable effects on the overall blocked solution. The main outcomes are as follows: first, smaller-sized complicated problems can be performed natively on the accelerator, with much higher efficiency, instead of being run completely on the host. Second, no data movement and corresponding overheads are incurred when offloading complicated kernels that are parts of a larger problem to the host processor. Instead, they can be completely performed by the accelerator.

The rest of the paper is organized as follows: in Section 2, we present related work on the implementations of the same algorithms on different platforms. In Section 3, a brief description of the baseline accelerator is given. Section 4 then describes the algorithms as well as the limitations and mapping challenges for the proposed accelerator. Section 5 details necessary modifications to the conventional designs such that all computation can be performed on the accelerator itself. The evaluation of the proposed architecture with sensitivity studies of energy efficiency, area overheads, and utilization gains are followed in Section 6. Finally, we conclude with a summary of contributions and future work in Section 7.

2 Related Work

Implementation of matrix factorizations on both conventional high performance platforms and accelerators has been widely studied. Many existing solutions perform more complex kernels on a more general purpose (host) processor while the high-performance engine only computes paralellizable blocks of the problem [3], [4]. The mapping of matrix factorizations have also been studied on accelerators like the Cell processor [8], [9], [10], and the Clearspeed CSX architecture [11]. Both of these architectures are heterogenous and the complex operations are run on the general purpose processor.

A typical solution for LU factorization on GPUs is presented in [4]. The details of multi-core, multi-GPU QR factorization scheduling are discussed in [3]. A solution for QR factorization that can be entirely executed on the GPU is presented in [12]. For LU factorization on GPUs, a technique to reduce matrix decomposition and row operations to a series of rasterization problems is used [13]. There, pointer swapping is used instead of data swapping for pivoting operations.

On FPGAs, [14] discusses LU factorization without pivoting. However, when pivoting is needed, the algorithm mapping becomes more challenging and less efficient due to complexities of the pivoting process and resulting wasted cycles. LAPACKrc [15] is a FPGA library with functionality that includes Cholesky, LU and QR factorizations. The architecture has similarities to the LAP. However, due to limitations of FPGAs, it does not have enough local memory. Similar concepts as in this paper for FPGA implementation and design of a unified, area-efficient unit that can perform the necessary computations (division, square root and inverse square root operations that will be discussed later) for calculating Householder QR factorization is presented in [16]. Finally, a tiled matrix decomposition based on blocking principles is presented in [17].

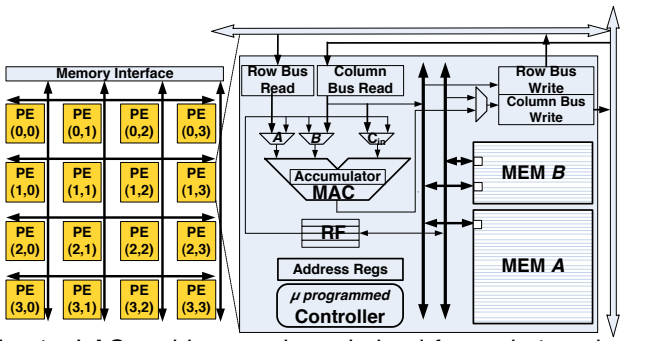

Fig. 1. LAC architecture is optimized for rank-1 updates to perform matrix multiplication [6].

3 Base Architecture

We study opportunities for floating-point extensions to an already highly-optimized accelerator. The microarchitecture of the Linear Algebra Core (LAC) is illustrated in Figure 1. A LAC achieves orders of magnitude better efficiency in power and area consumption compared to conventional general purpose architectures [6]. It is specifically optimized to perform rank-1 updates that form the inner kernels of parallel matrix multiplication. The LAC architecture consists of a 2D array of $n_r \times n_r$ Processing Elements (PEs), with $n_r = 4$ in Figure 1. Each PE has a Multiply-ACcumulate (MAC) unit with a local accumulator, and local Static Random-Access Memory (SRAM) storage divided into a bigger single-ported and a smaller dual-ported memory. PEs are connected by simple, low-overhead horizontal and vertical broadcast buses that each can move a double word per cycle. This is sufficient bandwidth to support all operations with GEMM and Symmetric Rank-k Update (SYRK) having the highest requirements [6], [7].

MAC units 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. To achieve high performance and register level locality, the LAC utilizes pipelined MAC units that can achieve a throughput of one MAC operation per cycle [18]. Note that this is not the case in current general-purpose architectures, which require extra data handling to alternate computation of multiple sub-blocks of the matrix being updated [19].

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2014.2315627, IEEE Transactions on Computers

4 Factorization Algorithms

In this section, we show the challenges and limitations in current architectures related to performing efficient matrix factorizations, and how to overcome these limitations through appropriate architecture extensions. We examine the three most commonly-used operations related to the solution of linear systems: Cholesky, LU (with partial pivoting), and QR factorization. Here, we first focus on small problems that fit into the LAC registers. Then we describe the blocked algorithm for problems that fit in the aggregate SRAM memories of the LAC. Bigger problem sizes can be blocked into smaller problems that are mainly composed of level-3 BLAS operations (discussed in [3]). We briefly review the relevant algorithms and their microarchitecture mappings. The purpose is to expose specialized operations, utilized by these algorithms, that can be supported in hardware.

4.1 Cholesky Factorization

Cholesky factorization is the most straightforward factorization operation. It is representative of a broad class of linear algebra operations and their complexities. Given a symmetric positive definite matrix[1], $A \in \mathbb{R}^{n \times n}$, the Cholesky factorization produces a lower triangular matrix, $L \in \mathbb{R}^{n \times n}$ such that $A = LL^T$. The basic algorithm [5] we will utilize can be motivated as follows: partition

$$A = \left(\begin{array}{c|c} \alpha_{11} & a_{12}^T \\ \hline a_{21} & A_{22} \end{array}\right) \text{ and } L = \left(\begin{array}{c|c} \lambda_{11} & 0 \\ \hline l_{21} & L_{22} \end{array}\right),$$

where α_{11} and λ_{11} are scalars. We can compute L from A via the operations

$$\begin{array}{c|c} \alpha_{11} := \lambda_{11} = \sqrt{\alpha_{11}} & \star \\ \hline a_{21} := l_{21} = (1/\lambda_{11}) a_{21} & A_{22} := L_{22} = \mathrm{Chol}(A_{22} - l_{21} l_{12}^T) \end{array},$$

overwriting A with L. For high performance, it is beneficial to also derive a blocked algorithm that casts most computations in terms of matrix-matrix operations, which we will describe later in this section. The observation is that the "square-root-and-reciprocal" operation $\alpha_{11} := \sqrt{\alpha_{11}}; t = 1/\alpha_{11}$ is important, and that it should therefore be beneficial to augment the microarchitecture with a unit that computes $f(x) = 1/\sqrt{x}$ when mapping the Cholesky factorization onto the LAC.

4.1.1 Basic Cholesky Factorization $n_r \times n_r$

We now focus on how to factor a $n_r \times n_r$ submatrix when stored in the registers of the LAC (with $n_r \times n_r$ PEs). In Figure 2, we show the second iteration of the algorithm. For this subproblem, the matrix has also been copied to the upper triangular part, which simplifies the design.

In each iteration $i = 0, \ldots, n_r - 1$, the algorithm performs three steps $S1$ through $S3$. In $S1$, the invert-square-root is computed. In $S2$, the element in PE(i,i) is updated with its inverse square root. The result is broadcast within the ith PE row and ith PE column. It is then multiplied into all elements of the column and row which are below and to the right of PE(i,i). In $S3$, the results of these computations are broadcast within the columns and rows to be multiplied by each other as part of a rank-1 update of the remaining part of matrix A. This completes one iteration, which is repeated for $i = 0, \ldots, n_r - 1$. Given a MAC unit with p pipeline stages and an inverse square root unit with q stages, this $n_r \times n_r$ Cholesky factorization takes $2p(n_r - 1) + q(n_r)$ cycles. Due to the data dependencies between different PEs within and between iterations, each element has to go through p stages of MAC units while other stages are idle. The last iteration only replaces the PE($n_r - 1$,$n_r - 1$) value by its square root, which only requires q additional cycles.

1. A condition required to ensure that the square root and inversion of a non-positive number is never encountered.

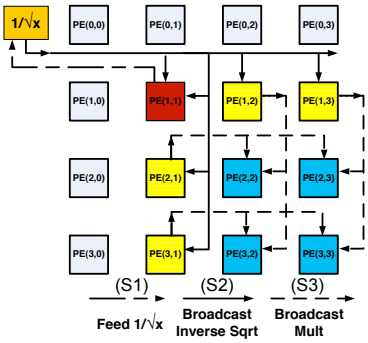

Fig. 2. 4×4 Cholesky decomposition mapping on the LAC, 2nd iteration.

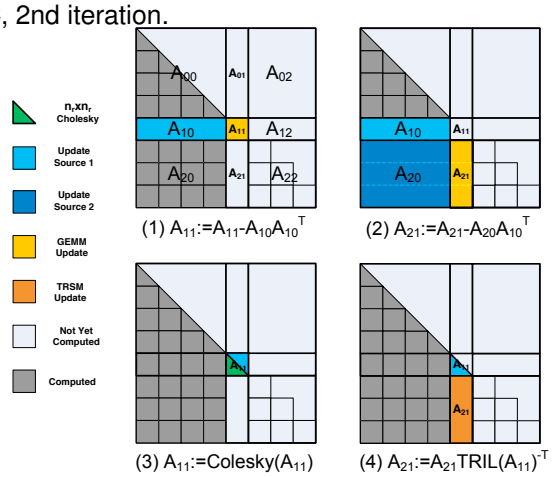

Fig. 3. Blocked Cholesky Factorization fifth Iteration.

4.1.2 Blocked Cholesky Factorization $kn_r \times kn_r$

Let us now assume that a larger matrix, $kn_r \times kn_r$ is distributed among the local stores of PEs in a 2D round-robin fashion. We will describe how a single iteration of the blocked left-looking algorithm is performed by the LAC. In Figure 3 we show the case where $k = 8$ and a block in that figure fits in the registers of the PEs. Highlighted is the data involved in the fifth iteration.

We describe the different operations to be performed:

(1) $A_{11} := A_{11} - A_{10}A_{10}^T$**:** Diagonal block A_{11} is moved to the accumulators of the PEs. $A_{11} - A_{10}A_{10}^T$ is a matrix multiplication, which is orchestrated as series of rank-1 updates that perform GEMM [6].

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2014.2315627, IEEE Transactions on Computers

(2) $A_{21} := A_{21} - A_{20}A_{10}^T$**:** Blocks of A_{21} are moved to the accumulators of the PEs. A block is updated much like $A_{11} - A_{10}A_{10}^T$ was, except that the first A_{10} is replaced by the row block of A_{21} that corresponds to the current block of A_{21} being updated.
(3) $A_{11} := \text{Chol}(A_{11})$**:** For this operation, the described $n_r \times n_r$ Cholesky factorization is employed.
(4) $A_{21} := A_{21}A_{11}^{-T}$**:** A_{11} is brought into the LAC registers, much like it was for the $n_r \times n_r$ Cholesky factorization. Blocks of A_{21} are streamed through. This operation is known as Triangular Solve with Multiple Right-hand Side (TRSM), details of which could be found in [7].

We chose a blocked left-looking algorithm primarily because a block is repeatedly updated by data that can be streamed, enhancing both temporal and spatial locality. The important insight is that by designing the algorithm and architecture hand-in-hand, the appropriate algorithm can guide the hardware design and vice versa.

The key complexity when performing Cholesky factorization is the inverse square-root and square-root operations. If we add this special function to the core, the LAC can perform the inner kernel of the Cholesky factorization natively. The $n_r \times n_r$ Cholesky factorization is purely sequential with minimal parallelism in rank-1 updates. However, as we observed, it is a small part of a bigger, blocked Cholesky factorization. The goal here is to avoid sending data back and forth to a general purpose processor or performing this operation in emulation on the existing MAC units, which would keep the rest of the core largely idle.

4.2 LU Factorization with Partial Pivoting

LU factorization with partial pivoting is a more general solution for decomposing matrices. The LU factorization of a square matrix A is the first and most computationally intensive step towards solving $Ax = b$. It decomposes a matrix A into a unit lower-triangular matrix L and an upper-triangular matrix U such that $A = LU$. We again briefly motivate the algorithm that we utilize [5]: partition

$$A = \left(\begin{array}{c|c} \alpha_{11} & a_{12}^T \\ \hline a_{21} & A_{22} \end{array}\right),\ L = \left(\begin{array}{c|c} 1 & 0 \\ \hline l_{21} & L_{22} \end{array}\right),\ U = \left(\begin{array}{c|c} \upsilon_{11} & u_{12}^T \\ \hline 0 & U_{22} \end{array}\right),$$

where α_{11}, and υ_{11} are scalars. We can compute L and U in place for matrix A. The diagonal elements of L are not stored (all of them are equal to one). The strictly lower triangular part of A is replaced by L. The upper triangular part of A, including its diagonal elements, is replaced by U as follows:

$$\begin{array}{c|c} \alpha_{11} := \upsilon_{11} \text{ (no-op)} & a_{12}^T := u_{12}^T \text{ (no-op)} \\ \hline a_{21} := l_{21} = a_{21}/\upsilon_{11} & A_{22} := \text{LU}(A_{22} - l_{21}u_{12}^T) \end{array}.$$

The LU factorization described so far is only computable if no zero-valued α_{11} is encountered. In practice, even if this condition is satisfied, the use of finite precision arithmetic can cause failure in the naive algorithm. The update to matrix A in the first iteration is given by

$$\left(\begin{array}{c|ccc} \alpha_{11} & \alpha_{12} & \cdots & \alpha_{1,n} \\ \hline 0 & \alpha_{22} - \lambda_{21}\alpha_{12} & \cdots & \alpha_{2,n} - \lambda_{21}\alpha_{1,n} \\ 0 & \alpha_{32} - \lambda_{31}\alpha_{12} & \cdots & \alpha_{3,n} - \lambda_{31}\alpha_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & \alpha_{n,2} - \lambda_{n,1}\alpha_{12} & \cdots & \alpha_{n,n} - \lambda_{n,1}\alpha_{1,n} \end{array}\right),$$

where $\lambda_{i,1} = \alpha_{i,1}/\alpha_{11}$, $2 \leq i \leq n$. The algorithm clearly fails if $\alpha_{11} = 0$. If $\alpha_{11} \neq 0$ and $|\alpha_{i,1}| \gg |\alpha_{11}|$, then $\lambda_{i,1}$ will be large in magnitude, and it can happen that for some i and j the value $|\alpha_{i,j} - \lambda_{i,1}\alpha_{i,j}| \gg |\alpha_{i,j}|$, $2 \leq j \leq n$; that is, the update greatly increases the magnitude of $\alpha_{i,j}$. This is a phenomenon known as large element growth and leads to numerical instability. The problem of element growth can be solved by rearranging (pivoting) the rows of the matrix (as the computation unfolds). Specifically, the first column of matrix A is searched for the largest element in magnitude. The row that contains that element, the *pivot row*, is swapped with the first row, after which the current step of the LU factorization proceeds. The net effect is that $|\lambda_{i,1}| \leq 1$ so that $|\alpha_{i,j} - \lambda_{i,1}\alpha_{1,j}|$ is of a magnitude comparable to the largest of $|\alpha_{i,j}|$ and $|\alpha_{1,j}|$, thus keeping element growth bounded. This is known as the *LU factorization with partial pivoting*. The observation is that finding the (index of the) largest value in magnitude in a vector is important for this operation.

4.2.1 Basic LU Factorization $kn_r \times n_r$

To study opportunities for corresponding architecture extensions, we focus on how to factor a $kn_r \times n_r$ submatrix (see Figures 4, 5) stored in a 2D round-robin fashion in the local store and registers of the LAC (with $n_r \times n_r$ PEs). In Figure 4 we show the second iteration of the right-looking unblocked algorithm ($i = 1$).

In each iteration $i = 0, \ldots, n_r - 1$, the algorithm performs four steps, $S1$ through $S4$. In $S1$, the elements in the ith column below the diagonal are searched for the maximum element in magnitude. This element can be in any of the ith column's PEs. Here, we just assume that it is in the row with $j = 2$. After the row with maximum value (the pivot row) is found, in $S2$, the pivot value is sent to the reciprocal (1/X) unit and the pivot row is swapped with the diagonal (ith) row concurrently. In $S3$, the reciprocal (1/X) is broadcast within the ith column and multiplied into the elements below PE(i,i). In $S4$, the results of the division (in the ith column) are broadcast within the rows. Simultaneously, the values in the ith (pivot) row to the right of the ith column are broadcast within the columns. These values are multiplied as part of a rank-1 update of the remaining part of matrix A. This completes the current iteration, which is repeated for $i = 0, \ldots, n_r - 1$.

According to the above mapping, most of the operations are cast as rank-1 updates and multiplications that are already provided in the existing LAC architecture. In addition to these operations, two other essential computations are required: first, a series of floating-point

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2014.2315627, IEEE Transactions on Computers

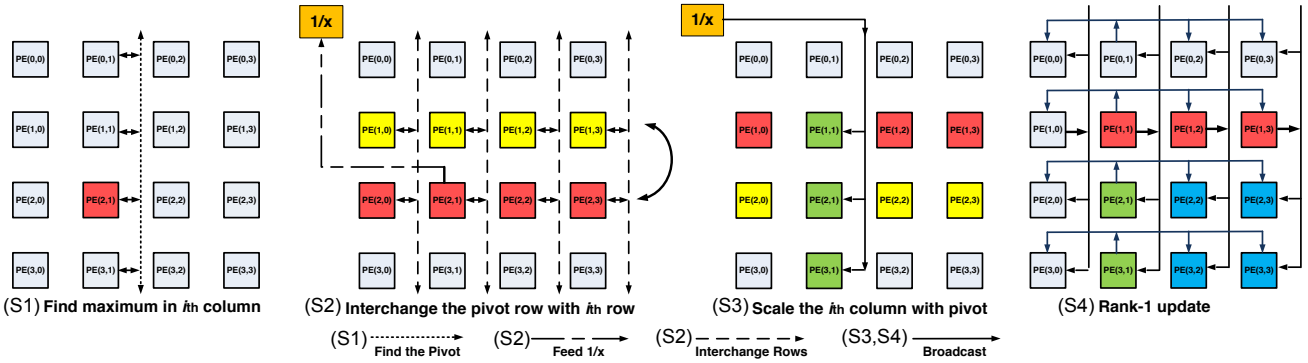

Fig. 4. Second iteration of a $kn_r \times n_r$ LU factorization with partial pivoting on the LAC.

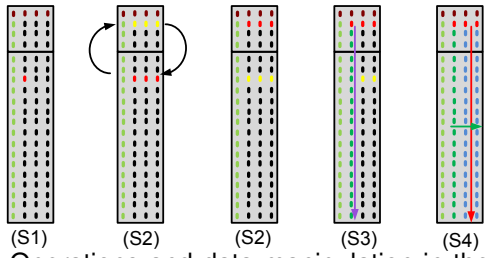

Fig. 5. Operations and data manipulation in the second iteration of a $kn_r \times n_r$ LU factorization inner kernel.

comparisons to find the maximal value in a vector (column); and second, a reciprocal (1/X) operation needed to scale the values in the ith column by the pivot (S2 in Section 4). Due to these extra complexities, solutions for most existing accelerators send the whole $kn_r \times n_r$ block to a host processor to avoid performing the factorization themselves [3], [4].

4.2.2 *Blocked LU Factorization* $kn_r \times kn_r$

Let us now assume that a larger matrix, $kn_r \times kn_r$ is distributed among the local stores much like the matrix was for Cholesky (wrapped). We will describe how a single iteration of the blocked left-looking algorithm is performed by the LAC.

To formally include row swapping in the blocked LU factorization, we introduce permutation matrices, which have the effect of rearranging the elements of vectors and entire rows or columns of matrices. A matrix $P \in \mathbb{R}^{n \times n}$ is said to be a *permutation* matrix (or permutation) if, when applied to the vector $x = (\chi_0, \chi_0, \ldots, \chi_{n-1})^T$, it merely rearranges the order of the elements in that vector. Such a permutation can be represented by the vector of integers $p = (\pi_0, \pi_0, \ldots, \pi_{n-1})^T$, where $\{\pi_0, \pi_1, \ldots, \pi_{n-1}\}$ is a permutation of $\{0, 1, \ldots, n-1\}$, and the scalars π_is indicate that the permuted vector is given by $Px = \left(\chi_{\pi_0}, \chi_{\pi_1}, \ldots, \chi_{\pi_{n-1}}\right)^T$.

The blocked LU factorization with partial pivoting can then be performed by computing L, U, and a permutation p of $\{0, 1, \ldots, n-1\}$ to satisfy $P(p)A = LU$. Note that in the blocked LU factorization with pivoting, the search space for the pivot is all the elements below the diagonal. Therefore, a whole column panel of A, which includes the diagonal block, must be processed and factorized.

In Figure 6, we show the case where $k = 8$ and therefore, A_{11} in that figure fits in the registers of the PEs.

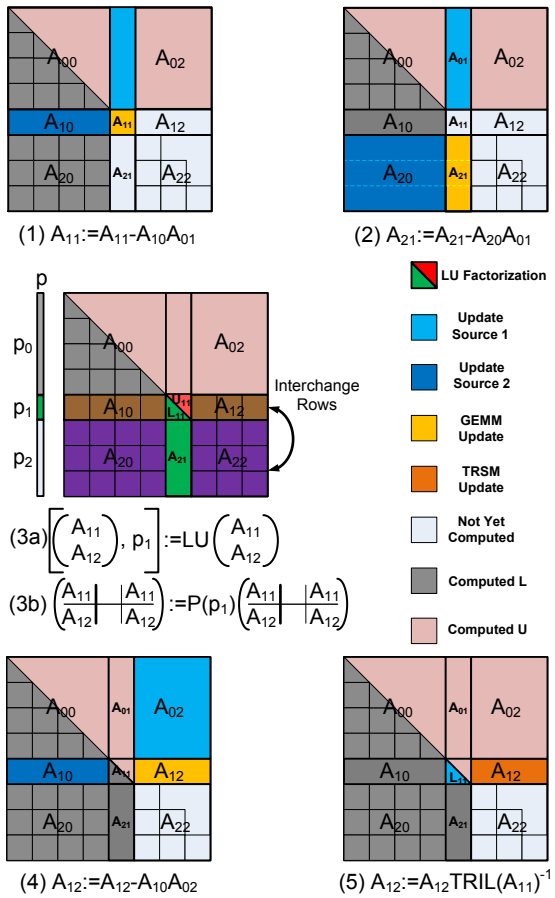

Fig. 6. Left-looking Blocked LU factorization, fifth iteration, for a matrix stored in the LAC local memory.

Highlighted is the data involved in the fifth iteration. We describe the different operations to be performed:

(1) $A_{11} := A_{11} - A_{10}A_{01}$**:** Diagonal block A_{11} is moved to the accumulators of the PEs. $A_{11} - A_{10}A_{01}$ is a matrix multiplication, which is orchestrated as series of rank-1 updates that perform GEMM [6].

(2) $A_{21} := A_{21} - A_{20}A_{01}$**:** Blocks of A_{21} are moved to the accumulators of the PEs. A block is updated much like $A_{11} - A_{10}A_{01}$ was, except that the first A_{10} is replaced by the row block of A_{21} that corresponds to the current block of A_{21} being updated.

(3a) $\left[\left(\frac{A_{11}}{A_{21}}\right), p_1\right] := \text{LU\_piv}\left(\frac{A_{11}}{A_{21}}\right)$: For this operation, the described inner kernel for $k_i n_r \times n_r$ LU factorization is employed.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2014.2315627, IEEE Transactions on Computers

(3b) $\left(\begin{array}{c|c} A_{10} & A_{12} \\ \hline A_{20} & A_{22} \end{array}\right) := P(p_1)\left(\begin{array}{c|c} A_{10} & A_{12} \\ \hline A_{20} & A_{22} \end{array}\right)$: This operation applies the pivoting on the other rows of the matrix and switches them. The PEs that own the elements of the pivot row switch them with the diagonal block rows. This operation can be merged with the inner kernel for LU factorization.

(4) $A_{12} := A_{12} - A_{10}A_{02}$**:** Blocks of A_{12} are moved to the accumulators of the PEs. A block is updated much like $A_{11} - A_{10}A_{01}$ was.

(5) $A_{12} := \text{TRILU}(A_{11})^{-1}A_{12}$: $L_{11} = \text{TRILU}(A_{11})$ (unit lower triangular part of A_{11}) is brought into the LAC registers, much like it was for the $n_r \times n_r$ Cholesky factorization. Blocks of A_{21} are streamed through and the TRSM operation is performed.

In LU factorization with partial pivoting, PEs in the LAC must be able to compare floating-point numbers to find the pivot (S1 in Section 4). In the blocked LU factorization, we have used the left-looking algorithm, which is the most efficient variant with regards to data locality [20]. In the left-looking LU factorization, the PEs themselves are computing the temporary values that they will compare in the next iteration of the algorithm. Knowing this fact, the compare operation could be done implicitly without any extra latency and delay penalty. The other operation that is needed for LU factorization is the reciprocal (1/X) to avoid multiple division operations and simply scale all the values by the reciprocal of the pivot.

4.3 QR Factorization

Householder QR factorization is often used when solving a linear least-squares problem. QR factorization decomposes a matrix $A \in \mathbb{R}^{m\times n}(m \geq n)$ into an orthonormal matrix matrix $Q \in \mathbb{R}^{m\times n}$ and an upper-triangular matrix $R \in \mathbb{R}^{n\times n}$ such that $A = QR$. The key to practical QR factorization algorithms is the Householder transformation [21]. Given $u \neq 0 \in \mathbb{R}^n$, the matrix $H = I - uu^T/\tau$ is a reflector or Householder transformation if $\tau = u^T u/2$. In practice, u is scaled so that its first element is "1". We will now show how to compute $A \rightarrow QR$, the QR factorization, of $m \times n$ matrix A as a sequence of Householder transformations applied to A.

In the first iteration, we partition $A \rightarrow \left(\begin{array}{c|c} \alpha_{11} & a_{12}^T \\ \hline a_{21} & A_{22} \end{array}\right)$. Let $\begin{pmatrix} 1 \\ u_{21} \end{pmatrix}$ and τ_1 define the Householder transformation that zeroes a_{21} when applied to the first column. Then, applying this Householder transform to A yields:

$$\left(\begin{array}{c|c} \alpha_{11} & a_{12}^T \\ \hline a_{21} & A_{22} \end{array}\right) := \left(I - \begin{pmatrix} 1 \\ u_{21} \end{pmatrix}\begin{pmatrix} 1 \\ u_{21} \end{pmatrix}^T/\tau_1\right)\left(\begin{array}{c|c} \alpha_{11} & a_{12}^T \\ \hline a_{21} & A_{22} \end{array}\right)$$

$$= \left(\begin{array}{c|c} \rho_{11} & a_{12}^T - w_{12}^T \\ \hline 0 & A_{22} - u_{21}w_{12}^T \end{array}\right),$$

where $w_{12}^T = (a_{12}^T + u_{21}^T A_{22})/\tau_1$. Computation of a full QR factorization of A will now proceed with A_{22}.

QR factorization with Householder reflectors is a more complicated operation compared to all of the previous cases we have presented so far. The new complexity introduced in this algorithm is in the computation of u_{21}, τ_1, and ρ_{11} from α_{11} and a_{21}, captured in Figure 7, which require a vector norm computation and scaling (division). This is referred to as the computation of the Householder vector.

The 2-norm of a vector x with elements $\chi_0, \cdots, \chi_{n-1}$ is given by $\|x\| := (\sum_{i=0}^{n-1}|\chi_i|^2)^{1/2} = \sqrt{\chi_0^2 + \chi_2^2 + \ldots + \chi_{n-1}^2}$. The problem is that intermediate values can overflow or underflow. This is avoided by

Algorithm: $\left(\begin{pmatrix} \rho_{11} \\ u_{21} \end{pmatrix}, \tau_1\right) = \text{HOUSEV}\begin{pmatrix} \alpha_{11} \\ a_{21} \end{pmatrix}$

	$\chi_2 := \|a_{21}\|_2$
	$\beta := \left\|\begin{pmatrix} \alpha_{11} \\ \chi_2 \end{pmatrix}\right\|_2 (= \|x\|_2)$
$\rho_{11} = -\text{sign}(\alpha_{11})\|x\|_2$	$\rho_{11} := -\text{sign}(\alpha_{11})\beta$
$\nu = \alpha_{11} + \text{sign}(\alpha_{11})\|x\|_2$	$\nu := \alpha_{11} - \rho_1$
$u_{21} = a_{21}/\nu$	$u_{21} := a_{21}/\nu$
	$\chi_2 = \chi_2/\|\nu\|(= \|u_{21}\|_2)$
$\tau_1 = (1 + u_{21}^T u_{21})/2$	$\tau_1 = (1 + \chi_2^2)/2$

Fig. 7. Computing the Householder transformation. Left: simple formulation. Right: efficient computation.

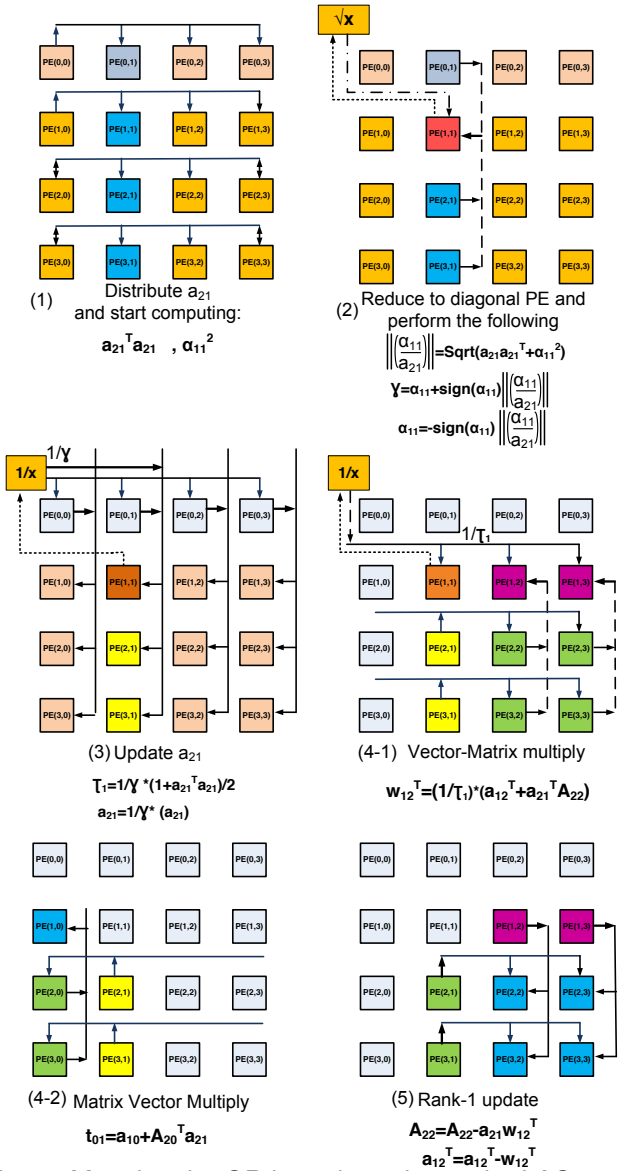

Fig. 8. Mapping the QR inner kernel onto the LAC.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2014.2315627, IEEE Transactions on Computers

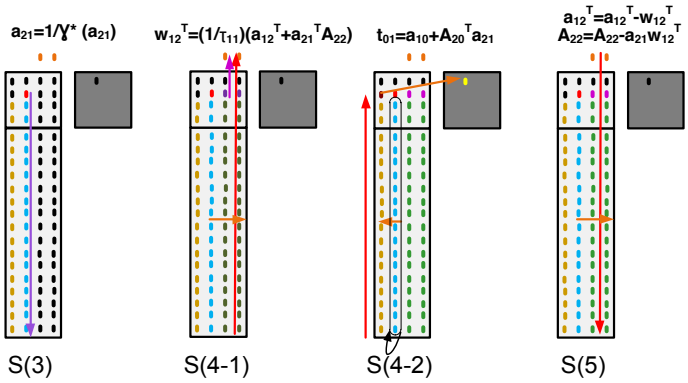

Fig. 9. Operations and data manipulation in the second iteration of a $kn_r \times n_r$ QR factorization S3 through S5.

normalizing x and performing the following operations instead: $t = \max_{i=0}^{n-1} |\chi_i|$; $y = x/t$; $\|x\|_2 := t \times \|y\|_2$. If not for overflow and underflow, the operation would be no more complex than an inner product followed by a square root. To avoid overflow and underflow, two passes over the data should be performed: a first pass to search and find the largest value in magnitude followed by a second pass to scale the vector elements and accumulate the inner-product. A one-pass algorithm has been presented in [22] that uses three accumulators for different value sizes. More details about how this is computed in software are discussed in [23], [24].

The extra operations that are needed to perform vector norm in a conventional fashion are the following: a floating-point comparator to find the maximum value in the vector just , a reciprocal function to scale the vector by the maximum value, and a square-root unit to compute the length of the scaled vector . However, we can observe that all these extra operations are only necessary due to limitations in hardware representations of real numbers [5].

Consider a floating number f that, according to the IEEE floating-point standard, is represented as $1.m_1 \times 2^{e_1}$, where $1 \leq 1.m_1 < 2$. Let us investigate the case of an overflow for $p = f^2$, and as a result $p = (1.m_2) \times 2^{e_2} = (1.m_1)^2 \times 2^{2e_1}$, where $1 \leq (1.m_1)^2 < 4$. If $(1.m_1)^2 \leq 2$, then $e_2 = 2e_1$. But, if $2 \leq (1.m_1)^2$, then $2 \leq (1.m_1)^2 = 2 \times 1.m_2 \leq 2$ and therefore $e_2 = 2e_1 + 1$. In both cases, a single extra exponent bit suffices for avoiding overflow and underflow in computations of the square of a floating-point number. Still, there might be the possibility of overflow/underflow due to accumulation of big/small numbers that could be avoided by adding a second exponent bit. However, the square-root of such inner product is still out of the bounds of a standard floating-point number. Therefore, only a single additional bit suffices. Hence, what is needed is a floating-point unit that has the ability to add one exponent bit for computing the vector norm to avoid overflows and corresponding algorithm complexities.

4.3.1 *Basic QR Factorization* $kn_r \times n_r$

We focus on how to factor a $kn_r \times n_r$ submatrix (see Figure 9) stored in a 2D round-robin fashion in the local store and registers of the LAC (with $n_r \times n_r$ PEs). The first step is to perform a vector norm of a scaled $kn_r \times 1$ vector (see Figure 9). Recall that such a column vector is only stored in one column of the PEs. In Figure 8, we show the second iteration of the right-looking unblocked algorithm ($i = 1$). In each iteration $i = 0, \ldots, n_r - 1$, the algorithm performs five steps $S1$ through $S5$.

In $S1$, the partial inner products of $a_{21}^T a_{21}$, and α_{11}^2 should be produced. Also, the elements of a_{21} are distributed to the other columns of PEs using the row broadcast buses. Here, a_{21} contains the elements below the diagonal of matrix that are located in the ith column of PEs. At the end of this stage, the ith column is left with n_r partial results.

In $S2$, a reduction across all the PEs of the ith column and a square-root operation is performed to compute $\alpha := \left\| \begin{pmatrix} \alpha_1 \\ \chi_2 \end{pmatrix} \right\|_2 (= \|x\|_2)$, which is the desired vector norm operation . The result is used to produce γ and update α_{11} in PE(i,i) with the help of a square-root unit. The LAC needs to save the result of $a_{21}^T a_{21}$ to avoid extra computations in $S3$. The reciprocal unit gets γ to produce $1/\tau_1$.

In $S3$, the result of the reciprocal operation $1/\gamma$ updates a_{21} and τ_1. All the PEs in the LAC each receive $1/\gamma$ via broadcast buses and update those parts of a_{21} that they received in the initial distribution of a_{21} in $S1$.

Note that elements of A_{22} are stored in columns to the right of column i, elements of A_{20} are also stored in columns to the right of column i, and a_{21} is distributed as mentioned earlier. In $S4$, two separate operations that utilize different PEs in the core are performed concurrently: In S4-1, the reciprocal unit receives τ_1 from PE(i,i) to produces $1/\tau_1$. The LAC also starts computing w_{12} by first performing a vector-matrix multiply $a_{21}^T A_{22} + a_{12}^T$, which uses row buses to distribute a_{21}^T and performs $a_{21}^T A_{22}$ in parallel. Then, a reduction across the column buses to the right of the ith column is performed to compute $a_{21}^T A_{22} + a_{12}^T$. The result of the vector-matrix multiplication is then scaled by broadcasting $1/\tau_1$ in the ith row and multiplying it into the PEs to the right of the diagonal PE in the that row. In $S4 - 2$, another matrix-vector multiplication is performed following the same principles, but this time to the left of the ith column in order to compute t_{01}.

In $S5$, row and column buses are used to broadcast elements of a_{21} and a_{12}^T to the PEs to the right of the ith row and below, including the ith column. A rank-1 update is performed to update $\begin{pmatrix} a_{12}^T \\ A_{22} \end{pmatrix}$. Here, we merged two operations into one to exploit the similar access pattern behavior and utilize the PEs. This completes the current iteration, which is repeated for $i = 0, \ldots, n_r - 1$.

4.3.2 *Blocked QR Factorization* $kn_r \times kn_r$

Let us now assume that a larger matrix, $kn_r \times kn_r$ is distributed among the local stores of the LAC. We will describe how a single iteration of the blocked QR algorithm is performed by the LAC. In Figure 10, we

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2014.2315627, IEEE Transactions on Computers

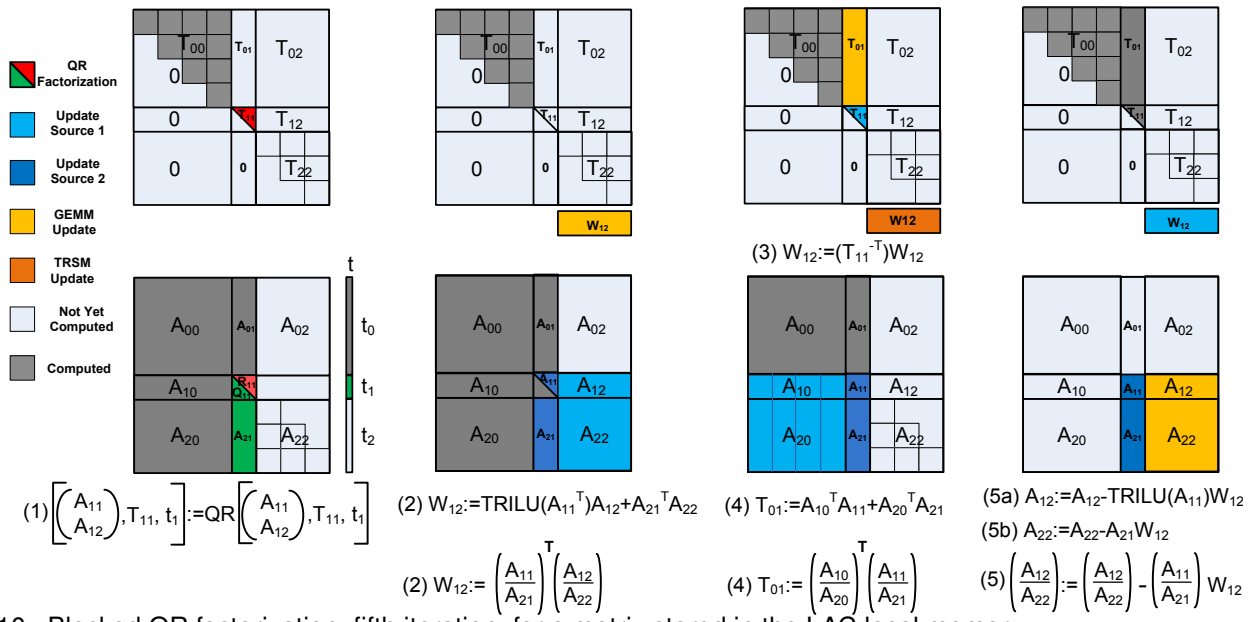

Fig. 10. Blocked QR factorization, fifth iteration, for a matrix stored in the LAC local memory.

show the case where $k = 8$ and therefore, A_{11} in that figure fits in the registers of the PEs. Highlighted is the data involved in the fifth iteration.

We describe the different operations to be performed:

(1) $\left[\begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix}, T_{11}, t_1\right] := \text{QR\_UNB}\left[\begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix}, T_{11}, t_1\right]$: For this operation, the described inner kernel of a $kn_r \times n_r$ QR factorization is employed.

(2) $W_{12} := \text{TRILU}(A_{11})^T A_{12} + A_{21}^T A_{22} = \begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix}^T \begin{pmatrix} A_{12} \\ A_{22} \end{pmatrix}$: Blocks of W_{12} are moved to the accumulators of the PEs. This update is a matrix multiplication.

(3) $W_{12} := T_{11}^{-T} W_{12}$: For this operation, an inner TRSM kernel is employed to update W_{12}.

(4) $T_{01} := U_{10}^T \text{ TRILU}(A_{11}) + A_{20}^T A_{21} = \begin{pmatrix} A_{10} \\ A_{20} \end{pmatrix}^T \begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix}$: Blocks of T_{01} are moved to the accumulators of the PEs. This update is also a matrix multiplication.

(5) $\begin{pmatrix} A_{12} \\ A_{22} \end{pmatrix} := \begin{pmatrix} A_{12} \\ A_{22} \end{pmatrix} - \begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix} W_{12}$:
This operation is a rank-n_r update of the $\begin{pmatrix} A_{12} \\ A_{22} \end{pmatrix}$ block of A with W_{12}. The LAC needs to bring each element of this block into the accumulators and update each in n_r cycles.

Most of the matrix multiplications that compute panels of W, and T (steps (2) and (3)) are in the form that performs optimal in the LAC. We chose the right looking variant because it exhibits a lower computation load. This variant avoids TRSM operations with big triangular blocks if not necessary. Other variants perform the TRSM operations with T_{00}, which is a big triangular block and yields into very low utilization on the LAC. Although this variant performs step (5) suboptimally, the LAC will not loose utilization but will perform normalization much more frequently, and the corresponding step will use around $10\% \sim 15\%$ more power compared to the optimal GEMM on the LAC [2]. In step (5), the blocks to the right (A_{22}) are updates, meaning that they need to be read and written.

The main requirement for the blocked QR factorization is the reciprocal unit to perform TRSM operations. Note that the blocked algorithm remains the same regardless of how the inner kernel of the QR is computed. The inner kernel that we described previously assumes that there is no normalization overhead for computing the vector norm. If there is no exponent bit added, the inner unblocked kernel will require additional steps and functions, such as the square-root.

5 Architecture

In this section, briefly review the extensions made to the LAC and our FPUs for matrix factorization applications. Figure 11 highlights our modifications to a baseline reconfigurable floating-point MAC unit with single-cycle accumulation taken from [25], [18]. This design does not support operations on denormalized numbers [25]. Details of our FPU extensions can be found in [5].

The first extension is for LU factorization with partial pivoting, where the LAC has to find the pivot by comparing all the elements in a single column. We noted that PEs in the same column have produced temporary results by performing rank-1 updates. To find the pivot, we add a comparator after the normalization stage in the floating-point unit of each PE. There is also a register that keeps the maximum value produced by the corresponding PE. If the new normalized result is greater than the maximum, it replaces the maximum and its index is saved by the external controller. An extra comparator is a simple logic in terms of area/power overhead [26]. It is also not on the critical path of the MAC unit and does not add any delay to the original design. With this extension, finding the pivot is simplified to a search among only n_r elements that are the maximum values produced by each PE in the same column.

The second extension is for vector norm operations in the Householder QR factorization. Previously, we have

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2014.2315627, IEEE Transactions on Computers

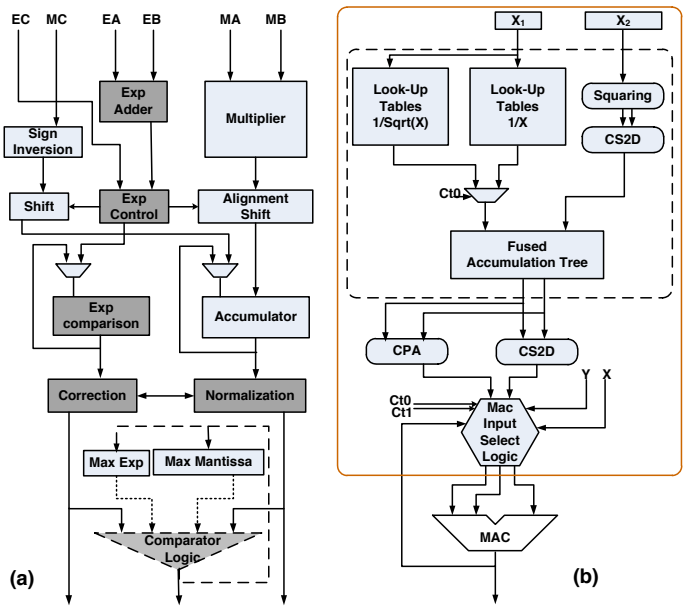

Fig. 11. Floating-point unit extensions: (a) extended reconfigurable single-cycle accumulation MAC unit [18] with addition of a comparator and extended exponent bit-width, where shaded blocks show which logic should change for exponent bit extension; (b) a single MAC unit design to support special functions. The overheads on top of an existing MAC unit are encapsulated in the big rounded rectangle. PEs in the LAC with that overhead can perform special functions.

shown how adding an extra exponent bit can overcome overflow/underflow problems in computing the vector norm. In Figure 11(a), the shaded blocks show where the architecture has to change. These changes are minimal and their cost is negligible. Specifically, with the architecture in [25], the same shifting logic for a base-32 shifter can be used. The only difference here is that the logic decides between four exponent input bits instead of three. Note that the result with extra exponent is directly fed to the square-root unit and is not visible to the user.

The third extension is the addition of division, reciprocal, square-root, and inverse square-root functions, which are required to perform Cholesky, LU and QR factorizations (as well as TRSM [7]). Several floating-point divide and square-root units have been introduced and studied in the literature [27], [28], [29]. There are mainly two categories of implementations [30]: multiplicative (iterative) and subtractive methods. Given the nature of linear algebra operations and the mapping of algorithms on the LAC, a multiplicative method is chosen. In our class of applications, a divide and square-root operation is often performed when other PEs are waiting in idle mode for its result and exploiting one of them for divide or square-root will not harm performance. As the iterations of Cholesky, LU, and QR factorization go forward, only a part of the LAC is utilized, and the top left parts are idle. Therefore, a diagonal PE is the best candidate for such extensions on top of its MAC unit.

We base our special function extensions on the architecture presented in [29]. It uses a 29-30 bit second-degree minimax polynomial approximation by using table look-ups [31]. Then, a single iteration of a modified Goldschmidt method is applied. This architecture guarantees the computation of exactly rounded IEEE double-precision results [29]. It can perform all four operations: divide Y/X, reciprocal 1/X, square-root $\sqrt{X}$, and inverse square-root $1/\sqrt{X}$. We can integrate such a design into single reconfigurable MAC unit, which performs all the computations itself (Figure 11(b)). This strategy reduces the design area and overhead, does not increase latency, but reduces the throughput. However, as indicated before, for our class of linear algebra operations, there is no need for a high-throughput division/square root unit. The extra overhead on top of an unmodified MAC unit includes the approximation logic and its look-up tables. A simple control logic performs the signal selection for the MAC inputs.

In summary, for our study, we assumed two types of extensions for the MAC units in the LAC, which include the maximum finder comparator and the extra exponent bit for LU with partial pivoting and QR factorization operations, respectively (Figure 11(a)). We also assumed three different LAC architectures with three options for divide/square-root extensions: first, a software-like implementation that uses a micro-programmed state machine to perform Goldschmidt's operation on the existing MAC units; second, an isolated divide/square-root unit that performs the operation with the architecture in [29]; and third, an extension to the PEs that adds extra logic and uses the available MAC units in the diagonal PEs (Figure 11(b)).

6 Experimental Results

In this section, we present area, power and performance estimates for the LAC with the modifications introduced in previous sections. We pursue two goals. First: to show how effective our proposed extensions are in achieving high performance for the inner kernels of matrix factorizations compared to the baseline architecture with a micro-coded software solution. Second: to study the impact of our extensions on problem sizes that fit in the aggregate LAC memory and are blocked. Such problem sizes are not large enough to amortize the cost of sending data for partial subproblems back and forth to an accelerator and, as such, are typically performed only by a host processor in a heterogenous system. We examine whether the augmented LAC with the new extensions maintains its high efficiency and high performance when performing complete matrix factorizations as well as level-3 BLAS. We perform an area, power, bandwidth, efficiency, and energy delay study to evaluate the benefits of these architectural extensions. We see the overall performance improvements for the different options and compare the computation load on the LAC in each iteration of the blocked algorithms.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2014.2315627, IEEE Transactions on Computers

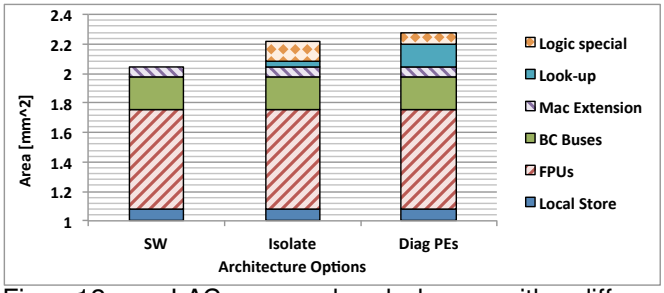

Fig. 12. LAC area break-down with different divide/square-root extensions.

6.1 Area and Power Estimation

Details of the basic PE and core-level implementation of a LAC in 45nm bulk CMOS technology are reported in [6]. For floating-point units, we use the power, area, and latency data from [32]. We combine it with complexity and delay reports from [29]. CACTI [33] is used to estimate the power and area consumption of memories, register files, look-up tables and buses.

The comparator is not on the critical path of the MAC pipeline and the overhead for the extra exponent bit is negligible. Therefore, we assume that there is no extra latency added to the existing MAC units with these extensions. The divide and square-root unit's timing, area, and power estimations are calculated using the results in [29]. For a software solution with multiple Godlschmidt iterations, we assume no extra power or area overhead for the micro-programmed state machine.

The area overhead on top of the LAC is shown in Figure 12. The area overhead for diagonal PEs includes the selection logic and the minimax function computation. In case of a 4×4 LAC, we observe that the overhead for these extensions is around 10% if an isolated unit is added to the LAC. If the extensions are added to all the diagonal PEs, more area is used. However, with an isolated unit more multipliers and multiply-add unit logic is required. The benefit of using the diagonal PEs is in avoiding the extra control logic and in less bus overhead for sending and receiving data.

6.2 Inner Kernel Performance and Efficiency

In this part, we analyze the unblocked inner kernels of the three factorization algorithms. We study the performance and efficiency behavior of our extensions for these algorithms and different inner kernel problem sizes as compared to the baseline architecture. We assume a LAC design in 45nm bulk CMOS technology running at 1GHz with 8 cycles pipeline latency for the MAC units.

As described in Section 4.1, Cholesky factorization can be blocked in a 2D fashion by breaking the problem down to a few level-3 BLAS operations and a Cholesky inner kernel. For our experiment, we evaluated a 4×4 unblocked Cholesky. We study the effects of different divide/square-root schemes on the performance of this inner kernel. The kernel performance and utilization is low because of the dependencies and the latency of the

Problem Size	Total Cycles			Dynamic Energy		
	SW	Isolated	Diagonal	SW	Isolated	Diagonal
Cholesky						
4	496	192	176	4 nJ	1 nJ	1 nJ
LU Factorization						
64	652	468	468	62 nJ	60 nJ	60 nJ
128	828	772	772	121 nJ	119 nJ	119 nJ
256	1380	1380	1380	239 nJ	236 nJ	236 nJ
LU Factorization with comparator						
64	500	316	316	53 nJ	51 nJ	51 nJ
128	612	556	556	103 nJ	101 nJ	101 nJ
256	1036	1036	1036	202 nJ	200 nJ	200 nJ
QR						
64	2256	1360	1296	94 nJ	89 nJ	89nJ
128	2336	1440	1376	178 nJ	173 nJ	173 nJ
256	2540	1644	1580	347 nJ	342 nJ	342 nJ
QR with comparator						
64	2040	1144	1080	84 nJ	79 nJ	79 nJ
128	2136	1240	1176	160 nJ	154 nJ	155 nJ
256	2328	1432	1368	310 nJ	305 nJ	305 nJ
QR with exponent bit extension						
64	1552	856	808	73 nJ	69 nJ	69 nJ
128	1584	888	840	139 nJ	135 nJ	135 nJ
256	1648	952	904	271 nJ	267 nJ	268 nJ

Fig. 13. Total cycle counts and dynamic energy consumption for different architecture options (columns for divide/square-root options, and row sets for MAC unit extension options), algorithms and problem sizes.

inverse square-root operation. We observe (Figure 13) that the number of cycles drops to a third by switching from a software solution to hardware extensions.

LU factorization with partial pivoting is not a 2D-scalable algorithm (see Section 4.2). The pivoting operation and scaling needs to be done for all rows of a given problem size. Hence, for a problem size of $kn_r \times kn_r$, the inner kernel that should be implemented on the LAC is a LU factorization of a $kn_r \times n_r$ block of the original problem. For our studies, we use problems with different $kn_r = 64, 128, 256$, which are typical problem sizes that fit on the LAC. We compare the performance of a LAC with different divide/square-root unit extensions in different columns and with/without the built-in comparator to find the pivot. As we have shown in Section 4, the reciprocal operation and pivoting (switching the rows) can be performed concurrently in the LAC owing to the column broadcast buses. The pivoting delay is the dominating term. Hence, bigger problem sizes are not sensitive to the latency of the reciprocal unit architecture. However, there is a 30% speed and 18% energy improvement with the comparator added to the MAC units (see Figure 13).

The QR factorization inner kernel algorithm differs depending on the exponent bit extension and whether there is a comparator or not. We compare the optimized algorithm for all three cases to have a fair comparison. In case of no comparator, a_{21} can get distributed concurrently while the comparison operation is being performed. In this way the inner product and scaling (for normalization) can be performed utilizing all of the PEs in the LAC. With only comparators included or when there is exponent bit extension, a_{21} does not initially get distributed. Therefore, the scaling (in case with only comparators) and inner-product operations only utilize

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2014.2315627, IEEE Transactions on Computers

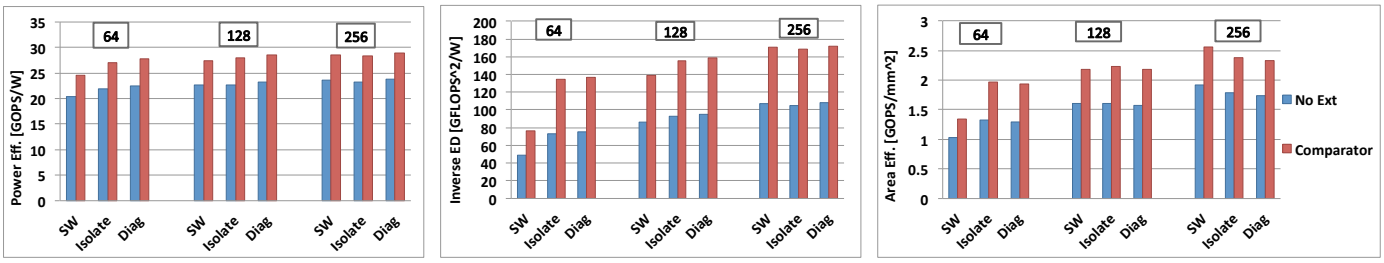

Fig. 14. Effect of hardware extensions on the inner kernels of LU factorization with three types of sqrt/division units and kernel heights of 64, 128, 256; power efficiency (left), inverse energy delay (middle) and area efficiency (right).

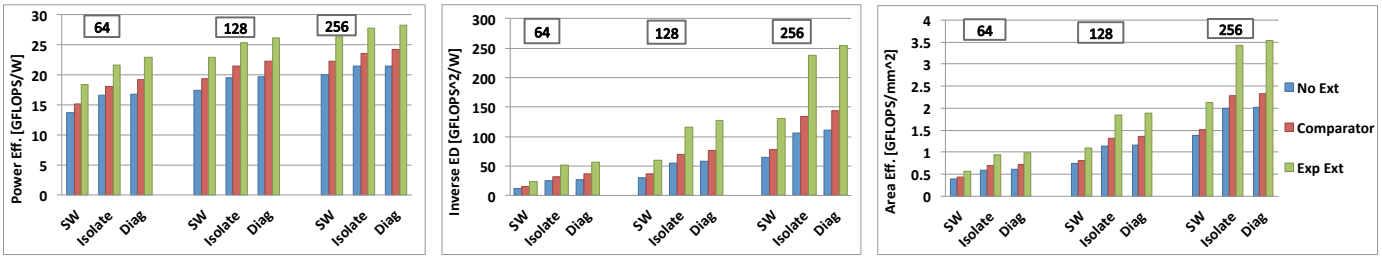

Fig. 15. Effect of hardware extensions on the inner kernels of QR factorization with three types of sqrt/division units and kernel heights of 64, 128, 256; power efficiency (left), inverse energy delay (middle) and area efficiency (right).

one column of the PEs. The rest of the operations are same for all of three cases, where there are again three options for divide/square-root operations. The problem sizes are $kn_r = 64, 128, 256$. Figure 13 demonstrates that exponent extensions save over 45% of the total cycles, and the divide/square-root unit saves up to 40% cycles compared to the baseline. When combined, these extensions save 65% of the cycles compared to the baseline. Energy savings reach up to 22% with the exponent bit extension. By contrast, different divide/square-root units do not differ in terms of dynamic energy consumption.

Utilization and efficiency can be calculated from the number of total cycles the hardware needs to perform an operation and the number of operations in each factorization. Another metric that we use is the inverse energy-delay. It shows how extensions reduce both latency and energy consumption.

The efficiency metrics and inverse energy-delay for LU factorization are presented in the Figure 14. Note that the pivoting operation is also taken into account. Therefore, we used GOPS instead of GFLOPS as performance metric. For LU factorization problems with $kn_r = 64, 128, 256$, we estimated the corresponding total number of operations to be 1560, 3096 and 6168, respectively. Results for LU factorization confirm that there is no improvement in efficiency with different reciprocal architectures when solving big problem sizes.

Figure 15 report the efficiency metrics and inverse energy-delay of the QR factorization inner kernels. For the QR factorization, we use problem sizes of $kn_r = 64, 128, 256$ as the baseline. Our implementation with the option of extended exponent bits results in an effective reduction in the number of actually required computations. QR benefits from all types of extensions, but the exponent bit is what brings significant improvements.

Since there are not many options for Cholesky, we only summarize the numbers here in the text. The number of operations in a 4×4 Cholesky kernel is 30. For different divide/square unit architectures (software, isolated, and on diagonal PEs), the achieved results are as follows: 1.95, 4.67 and 5.75 GFLOPS/W; 0.52, 4.95, and 5.15 GFLOPS^2/W; and 0.03, 0.06, 0.07 GFLOPS/mm^2. The reason for the very poor efficiency (less than 5 GFLOPS/W) is the small size of the kernel and limited available parallelism. Still, adding the special function unit improves efficiency around ten times, while reducing dynamic energy consumption by 75%.

6.3 Blocked Level Analyses

As discussed earlier, a typical linear algebra problem is blocked to get high performance. In Section 4, we saw how most of the computations in the matrix factorizations are cast in terms of GEMM and other level-3 BLAS operations. In the previous section, we showed the effect of extensions on the LAC performance and efficiency for the inner kernels of matrix factorizations. In this section, we take the most effective extensions and study their impact on the bigger blocked problems.

Figures 16, 17, and 18 show the computation cycles spent and the load on the broadcast buses in each of the 32 iterations of a typical 128×128 problem size for all three matrix factorizations on the LAC. Bus loads are shown for each operation, as averages per iteration and as total averages across the whole kernel for internal computations and external column bus transfers.

Figure 16 (left) plots the computation cycles for Cholesky factorization on the LAC. We observe that most of the cycles are used for GEMM operations. For the Cholesky inner kernel, we chose two options, one with the micro-coded software solution and one with the addition of the divide/square-root extensions on diagonal PEs. The extended architecture reduces the cycle counts by 10%, but does not save significant amounts of energy compared to the base design. Figure 16 (right) shows

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2014.2315627, IEEE Transactions on Computers

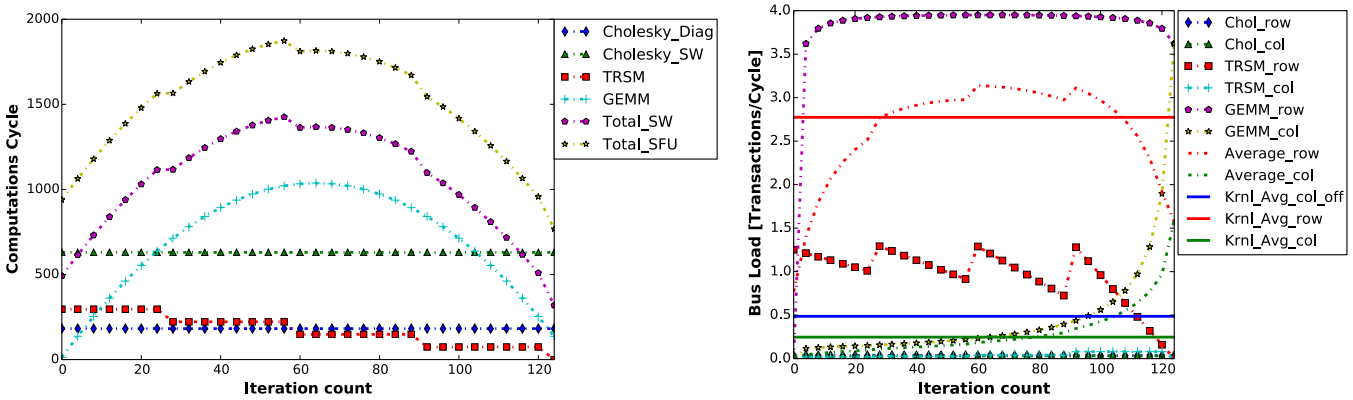

Fig. 16. 128×128 blocked Cholesky factorization with and without special function unit extensions on a 4×4 LAC: Cycles consumed (left) and bus load (right).

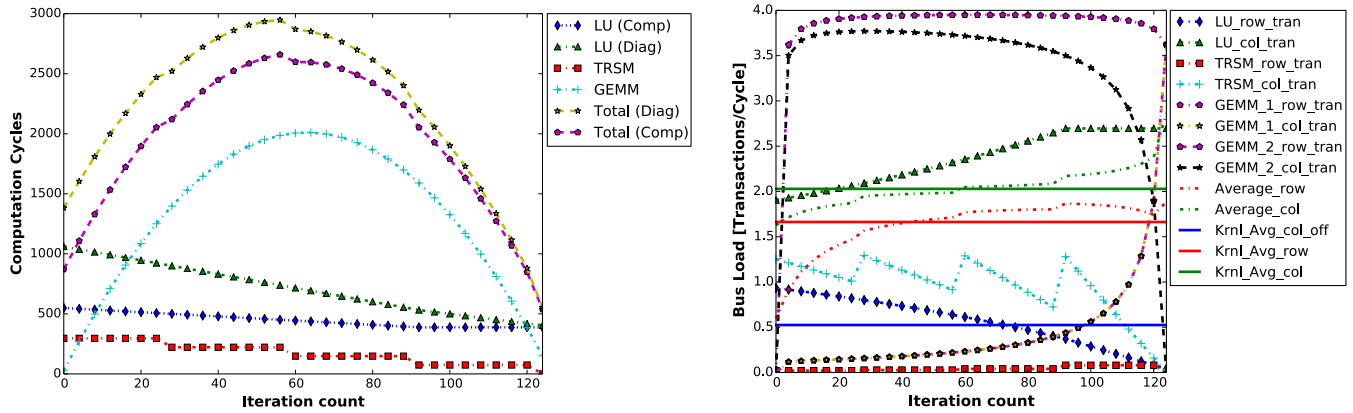

Fig. 17. 128×128 blocked LU factorization with partial pivoting with and without comparator extension on a 4×4 LAC: Cycles consumed (left) and bus load (right).

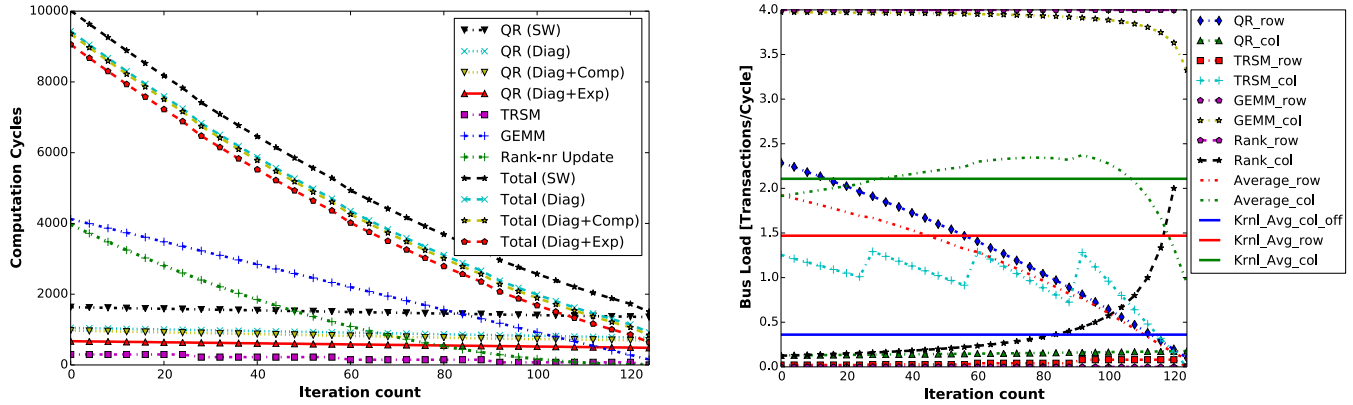

Fig. 18. 128×128 blocked Householder QR factorization with no extensions, only divide/square-root extension, comparator extension, and exponent bit extension on a 4×4 LAC: Cycles consumed (left) and bus load (right).

that the row broadcast buses are fully loaded when performing GEMM operation. The column buses are mostly idle and only loaded in the GEMM operations of the final iterations. The other operations do not occupy the broadcast buses as much. The row buses are occupied around 70% of the cycles. The column buses, counting off-core transfers, are only occupied 20% of the cycles.

As discussed previously for LU factorization with partial pivoting, the inner kernels do not save cycles when the divide/square-root extension is added. Therefore, we only discuss the effects of the comparator extension. Figure 17 (left) shows the cycles spent for LU factorization on the LAC. Adding a comparator is beneficial specifically in the initial iterations. The reason is that the search space for the pivot shrinks as the algorithm progresses. Adding a comparator saves up to half of the cycles for the LU inner kernel and 12% of the total cycles. Figure 17 (right) depicts the broadcast bus loads for each part of the blocked LU. *GEMM_1* refers to steps (1) and (2), and *GEMM_2* refers to step (4). While *GEMM_1* occupies the row buses and puts only a negligible load on column buses, *GEMM_2* behaves exact opposite way. On average, in each iteration, both row and column buses are 50% busy. The load for external transfer is around 25% of the bandwidth of the column buses.

QR factorization benefits from all divide/square-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2014.2315627, IEEE Transactions on Computers

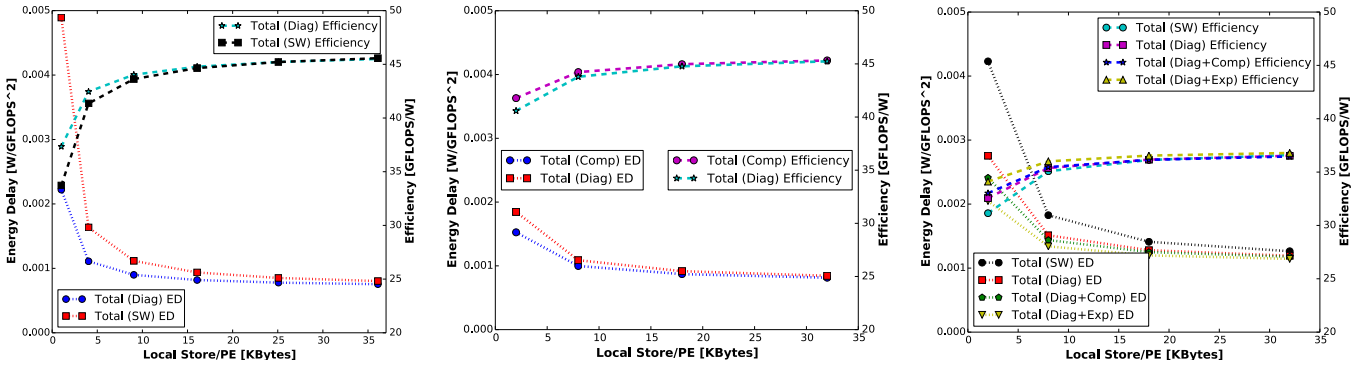

Fig. 19. The achievable energy delay and efficiency for different problem/PE local store sizes for Cholesky (left), LU (middle), and QR factorizations (right).

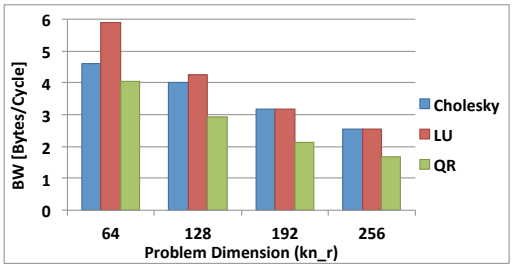

Fig. 20. Required off-core bandwidth for different problem sizes and different factorization algorithms.

root, comparator, and exponent bit extensions significantly. We study the effect of combinations of diagonal divide/square-root with either comparator or exponent bit extensions in the blocked algorithm. Figure 18 (left) shows cycles spent on the LAC. The extra exponent bit in combination with diagonal divide/square-root extensions is the most effective modification that can save up to 20% of the cycles. As in the cases of the Cholesky and LU factorizations, most of the computations are cast as matrix-matrix multiplications. Figure 18 (right) indicates the bus loads for the QR operation. *Rank* refers to rank-n_r updates (step (5)) and *GEMM* refers to all other steps that contain matrix multiplication. Rank-n_r updates fully occupy the column buses, and we can observe that the row buses are mostly busy in the initial iterations of the blocked QR. Here, the column buses are more busy than row buses. However, there are still many idle cycles left to perform off-core transfers.

Figure 20 summarizes the required off-core bandwidths to satisfy full overlap of communication with computation. Overall, the off-core bandwidth requirements are very low (around half a word per cycle). We observe that with smaller problem sizes, the required bandwidth is almost twice as much as for the larger problem sizes. QR factorization is the most complicated algorithm. It performs more computations per problem and therefore the required bandwidth is the least. LU and Cholesky behave similar for larger problem sizes.

Finally, Figure 19 shows the efficiency and energy delay of all three factorizations with respect to the required PE local store size for problem sizes of $64 \times 64, 128 \times 128, 192 \times 192$, and 256×256[2]. Figure 19 (left) shows how the energy efficiency is affected by the extensions for Cholesky factorization, especially in small problem sizes. As the problem size grows and the ratio of inner kernel computations drops, the two curves get closer. The energy delay is affected by both efficiency and cycle savings. Hence, we see a larger difference between the two options even for larger problem sizes. The LAC with 20 KBytes/PE can reach over 80% utilization for Cholesky factorization. We can observe in Figure 19 (middle) that LU factorization sees even less improvement in energy efficiency because most of the computation load and energy consumption is due to the level-3 BLAS operations. The energy delay savings are noticeable for smaller problem sizes. Although the search space for the pivot grows linearly with respect to the increase in problem size, the ratio of level-3 BLAS computations to the LU kernel grows cubically, and that overshadows the LU kernel savings. As a result, the extended LAC with 20 KBytes/PE can reach over 84% utilization for LU factorization with partial pivoting. Figure 19 (right) indicates that the QR factorization gains both efficiency and energy delay improvements especially in small problem sizes. The efficiency improvement is caused by leakage energy savings and reduction in extra computations due to avoided normalizations. The exponent bit extension affects the energy delay for all different problem sizes due to significant cycle and energy savings. The LAC with 20 KBytes/PE with both exponent and diagonal divide/square-root extensions can reach over 75% utilization for QR factorization.

7 Conclusions and Future Work

In this paper, we presented the mapping of both inner kernels and blocked matrix factorization problems onto a highly efficient linear algebra accelerator. We propose two modifications to the MAC unit designs to decrease the complexity of the algorithms. We also showed how existing processing elements can be enhanced to perform special functions such as divide and square-root operations. To demonstrate the effectiveness of our proposed

2. Due to less local store requirement of Cholesky, problem sizes up to 384×384 are presented.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2014.2315627, IEEE Transactions on Computers

extensions, we applied them to the mapping of Cholesky, LU and QR factorizations on such an improved architecture. We studied both inner kernels and blocked-level algorithms and presented the resulting performance and efficiency benefits. Results show that our extensions significantly increase efficiency and performance of inner kernels and are effective for bigger problem sizes that fit on the LAC. Future work includes the integration of the LAC into a heterogeneous system architecture next to general purpose CPUs and a heterogeneous shared memory systems, which will allow comparisons between the trade-offs of complexity and flexibility.

REFERENCES

[1] G. H. Golub *et al.*, "An analysis of the total least squares problem," Ithaca, NY, USA, Tech. Rep., 1980.

[2] A. Pedram *et al.*, "Codesign tradeoffs for high-performance, low-power linear algebra architectures," *IEEE Transactions on Computers, Special Issue on Power efficient computing*, vol. 61, no. 12, pp. 1724–1736, 2012.

[3] E. Agullo *et al.*, "QR factorization on a multicore node enhanced with multiple GPU accelerators," in *IPDPS2011*, 2011.

[4] V. Volkov *et al.*, "Benchmarking GPUs to tune dense linear algebra," *SC 2008*, 2008.

[5] A. Pedram *et al.*, "Floating point architecture extensions for optimized matrix factorization," in *ARITH*. IEEE, 2013.

[6] ——, "A high-performance, low-power linear algebra core," in *ASAP*. IEEE, 2011.

[7] ——, "A linear algebra core design for efficient Level-3 BLAS," in *ASAP*. IEEE, 2012.

[8] J. Kurzak *et al.*, "Solving systems of linear equations on the cell processor using cholesky factorization," *IEEE Trans. Parallel Distrib. Syst.*, vol. 19, pp. 1175–1186, September 2008.

[9] ——, "Qr factorization for the cell broadband engine," *Sci. Program.*, vol. 17, pp. 31–42, January 2009.

[10] T. Chen *et al.*, "Cell broadband engine architecture and its first implementation: a performance view," *IBM J. Res. Dev.*, vol. 51, pp. 559–572, September 2007.

[11] Y. Yamamoto *et al.*, "Accelerating the singular value decomposition of rectangular matrices with the CSX600 and the integrable SVD," in *PaCT*, ser. Lecture Notes in Computer Science, V. E. Malyshkin, Ed., vol. 4671. Springer, 2007, pp. 340–345.

[12] A. Kerr *et al.*, "QR decomposition on GPUs," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, ser. GPGPU-2, 2009.

[13] N. Galoppo *et al.*, "LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware," ser. SC '05, 2005.

[14] G. Wu *et al.*, "A high performance and memory efficient LU decomposer on FPGAs," *IEEE Trans on Computers*, 2012.

[15] J. Gonzalez *et al.*, "LAPACKrc: fast linear algebra kernels/solvers for FPGA accelerators," *SciDAC 2009*, no. 180, 2009.

[16] S. Aslan *et al.*, "Realization of area efficient QR factorization using unified division, square root, and inverse square root hardware," in *EIT '09*, 2009.

[17] Y.-G. Tai *et al.*, "Synthesizing tiled matrix decomposition on fpgas," in *FPL2011*, 2011.

[18] S. Jain et al., "A 90mW/GFlop 3.4GHz reconfigurable fused/continuous multiply-accumulator for floating-point and integer operands in 65nm," *VLSID '10.*, 2010.

[19] A. Pedram *et al.*, "On the efficiency of register file versus broadcast interconnect for collective communications in data-parallel hardware accelerators," *SBAC-PAD*, 2012.

[20] P. Bientinesi *et al.*, "Representing dense linear algebra algorithms: A farewell to indices," The University of Texas at Austin, Tech. Rep. TR-2006-10, 2006.

[21] T. Joffrain *et al.*, "Accumulating Householder transformations, revisited," *ACM Trans. Math. Softw.*, vol. 32, no. 2, pp. 169–179, June 2006.

[22] J. L. Blue, "A portable Fortran program to find the euclidean norm of a vector," *ACM Trans. Math. Softw.*, vol. 4, no. 1, 1978.

[23] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd ed. Philadelphia, PA, USA: SIAM, 2002.

[24] C. L. Lawson *et al.*, "Basic linear algebra subprograms for Fortran usage," *ACM Trans. Math. Soft.*, vol. 5, no. 3, pp. 308–323, Sept. 1979.

[25] 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.

[26] J. Stine *et al.*, "A combined two's complement and floating-point comparator," in *ISCAS 2005*, 2005.

[27] M. D. Ercegovac *et al.*, "Reciprocation, square root, inverse square root, and some elementary functions using small multipliers," *IEEE Trans. Comput.*, vol. 49, no. 7, July 2000.

[28] S. Oberman, "Floating point division and square root algorithms and implementation in the AMD-K7TM microprocessor," in *Arith14th*, 1999.

[29] J. A. Piñeiro *et al.*, "High-speed double-precision computation of reciprocal, division, square root and inverse square root," *IEEE Trans. Comput.*, 2002.

[30] P. Soderquist *et al.*, "Area and performance tradeoffs in floating-point divide and square-root implementations," *ACM Comput. Surv.*, vol. 28, no. 3, 1996.

[31] J. A. Piñeiro *et al.*, "High-speed function approximation using a minimax quadratic interpolator," *IEEE Trans. Comput.*, 2005.

[32] S. Galal et al, "Energy-efficient floating point unit design," *IEEE Trans. on Computers*, vol. PP, no. 99, 2010.

[33] N. Muralimanohar *et al.*, "Architecting efficient interconnects for large caches with CACTI 6.0," *IEEE Micro*, vol. 28, 2008.

Ardavan Pedram received the masters degree in computer engineering from the University of Tehran in 2006 and the PhD from the University of Texas at Austin in 2013. He currently is a Postdoctoral fellow in the University of Texas at Austin. His research interests include high performance computing and computer architecture. He specifically works on hardware-software co-design (algorithm for architecture) of special purposed accelerators for high-performance energy-efficient linear algebra and signal processing applications.

Andreas Gerstlauer received the Dipl-Ing degree in electrical engineering from the University of Stuttgart, Germany, in 1997, and the MS and PhD degrees in information and computer science from the University of California, Irvine (UCI), in 1998 and 2004, respectively. Since 2008, he has been with the University of Texas at Austin, where he is currently an assistant professor in electrical and computer engineering. Prior to joining UT Austin, he was an assistant researcher with the Center for Embedded Computer Systems, UCI. His research interests include system-level design automation, system modeling, design languages and methodologies, and embedded hardware and software synthesis. He is a senior member of the IEEE.

Robert A. van de Geijn is a Professor of Computer Science and member of the Institute for Computational Engineering and Sciences at the University of Texas at Austin. He received his PhD in Applied Mathematics from the University of Maryland College Park, in 1987. He heads the FLAME project, which pursues foundational research in the field of linear algebra libraries and has led to the development of the libflame library. One of the benefits of this library lies with its impact on the teaching of numerical linear algebra, for which Prof. van de Geijn received the UT Presidents Associates Teaching Excellence Award. He has published several books and more than a hundred refereed publications.