Floating Point Architecture Extensions for Optimized Matrix Factorization

Ardavan Pedram, Andreas Gerstlauer
Department of Electrical and Computer Engineering
The University of Texas at Austin
{ardavan,gerstl}@utexas.edu

Robert A. van de Geijn
Department of Computer Science
The University of Texas at Austin
rvdg@cs.utexas.edu

***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. 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 shows that our extensions to the MAC units can double the speed of required vector-norm operations while reducing energy by 60%. Similarly, up to 20% speedup with 15% savings in energy can be achieved for LU factorization. We show how such efficiency is maintained even in the complex inner kernels of these operations.

I. Introduction

Modern computers use floating-point representations for real numbers, which cause errors such as round-off, overflow, and underflow in computations. Quality implementations of numerical algorithms aim to ensure numerical stability and try to prevent spurious overflow and underflow errors. As a result, algorithms become more complex. The question is how to accommodate such changes when mapping these algorithms onto accelerators and/or into custom hardware.

A problem is that although the additional complexity does not constitute the bulk of the total operational cost, it often falls into the critical path of the algorithm. On general purpose processors, the performance overhead could include extra instructions, more memory traffic, and the cost of memory synchronization in the system. Overall, the complexities in the algorithm result in inherent overhead.

By contrast, accelerator designs achieve orders of magnitude improvement in power and area efficiency by specializing the compute data-path for only a specific application domain [1], [2]. They remove unnecessary overheads in general purpose architectures and use more fine-grain compute engines instead. The resulting architecture is less flexible but very efficient, both in terms of area and power consumption. For such architectures the mapping of algorithms with extra complexities can lead to significant implementation overhead due to their lack of flexibility. In the worst case, some of the most complex computations have to be offloaded to a (conventional) host processor.

Within the dense linear algebra domain, a typical computation can be blocked into sub-problems that expose highly parallelizable parts like GEneral Matrix-matrix Multiplication (GEMM). These can be mapped very efficiently to accelerators. However, many current solutions use heterogeneous computing for more complicated algorithms like Cholesky, QR, and LU factorization [3], [4]. Often, only the most parallelizable and simplest parts of these algorithms, which exhibit ample parallelism, are performed on the accelerator. Other more complex parts, which are added to the algorithm to overcome floating point limitations or which would require complex hardware to exploit fine grain parallelism, are off-loaded to a general purpose processor.

The problem with heterogeneous solutions is the overhead for communication back and forth with a general purpose processor. In the case of current GPUs, data has to be copied to the device memory and then back to the host memory through slow off-chip buses. Even when GPUs are integrated on the chip, data has to be moved all the way to off-chip memory in order to perform transfers between (typically) incoherent CPU and GPU 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 times the accelerator remains idle waiting for the data to be processed by the CPU, also wasting cycles. This is particularly noticeable for computation with small matrices.

In this paper, we propose a new solution that tries to avoid all inefficiencies caused by limitations in current architectures and thereby overcomes the complexities in matrix factorization algorithms. The problem is that architecture designers typically only have a high-level understanding of algorithms, while algorithm designers try to optimize for already existing architectures. Our solution is to revisit the whole system design by relaxing the architecture design space. By this we mean allowing architectural changes to the design in order to reduce complexity directly in the algorithm whenever possible. Thus, the solution is to exploit algorithm/architecture co-design.

We choose three complex linear algebra algorithms: the Cholesky, LU (with partial pivoting), and QR factorizations. In our methodology, we study these algorithms, their potential parallelism, and their mapping to current heterogeneous multi-core architectures. This exposes the limitations in current architectures that cause additional complexities. We start from a minimal-hardware Linear Algebra Processor (LAP) [5] that

was designed for GEMM and other similar matrix-matrix operations (level-3 BLAS [6]). Then, we add minimal, necessary but sufficient logic to avoid the need for running complex computations on a general purpose core.

The rest of the paper is organized as follows: in Section II, we present some of the related work on the implementations of the same algorithms on different platforms. In Section III, a brief description of the baseline accelerator is given. Section IV then describes the algorithms as well as the limitations and mapping challenges for the proposed accelerator. Section V proposes modifications to the conventional designs so 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 VI. Finally, we conclude with a summary of contributions and future work in Section VII.

II. 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 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 run on the GPU is presented in [7]. For LU factorization on GPUs, a technique to reduce matrix decomposition and row operations to a series of rasterization problems is used [8]. There, pointer swapping is used instead of data swapping for pivoting operations.

On FPGAs, [9] 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 wasted cycles. LAPACKrc [10] 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 [11]. Finally, a tiled matrix decomposition based on blocking principles is presented in [12].

III. 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. LAC achieves orders of magnitude better efficiency in power and area consumption compared to conventional general purpose architectures [5]. 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

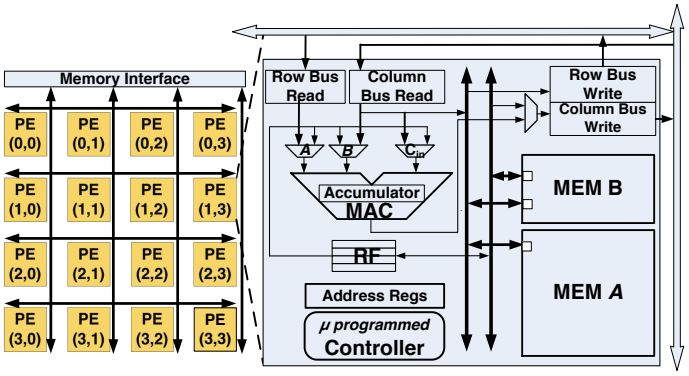

Fig. 1. LAC architecture is optimized for rank-1 updates to perform matrix multiplication [5].

of $n_r \times n_r$ Processing Elements (PEs), with $n_r = 4$ in the figure. 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.

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 [13]. 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 [14].

IV. Factorization Algorithms

In this section, we show the challenges and limitations in current architectures to perform efficient matrix factorizations. 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 focus on small problems that fit into the LAC memory. Bigger problem sizes can be blocked into smaller problems that are mainly composed of level-3 BLAS operations (discussed in [3]) and algorithms for smaller problems discussed here. 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.

A. 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 $A = LL^T$.

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

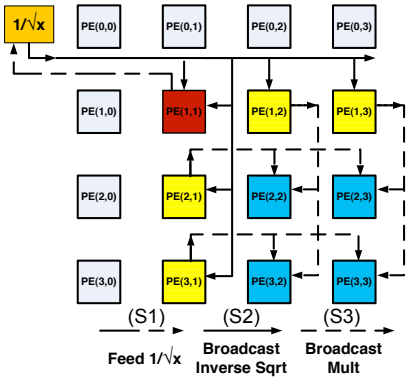

Fig. 2. 4×4 Cholesky decomposition mapping on the LAC, 2nd iteration.

The algorithm we will utilize can be motivated as follows: partition

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

where α_{11} and λ_{11} are scalars. Then $A = LL^T$ means that

$$\left(\begin{array}{c|c} \alpha_{11} & a_{12}^T \\ \hline a_{21} & A_{22} \end{array}\right) = \left(\begin{array}{c|c} \lambda_{11}^2 & * \\ \hline \lambda_{11} l_{21} & L_{22}L_{22}^T + l_{21}l_{21}^T \end{array}\right)$$

which in turn means that

$$\left(\begin{array}{c|c} \alpha_{11} = \lambda_{11}^2 & \star \\ \hline a_{21} = \lambda_{11} l_{21} & A_{22} - l_{21}l_{12}^T = L_{22}L_{22}^T \end{array}\right).$$

We can compute L from A via the operations

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

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, but we will not need these in our discussion. 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.

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.

Clearly, there are a lot of dependencies and there will be wasted cycles. However, what we will see is that this smaller subproblem is not where most computations happen when performing a larger Cholesky factorization. For this reason, we do not discuss details of how to fully optimize the LAC for this operation here.

The important idea is that by introducing an inverse square-root unit, that operation needs not to be performed on a host nor in software or emulation on the LAC, which yields a substantial savings in cycles.

B. 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: 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. Then $A = LU$ means that

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

so that

$$\left(\begin{array}{c|c} \alpha_{11} = \upsilon_{11} & a_{12}^T = u_{12}^T \\ \hline a_{21} = \upsilon_{11} l_{21} & A_{22} - l_{21}u_{12}^T = L_{22}U_{22} \end{array}\right).$$

We can thus compute L and U in place for matrix A. The diagonal elements of L are not stored (all of them are ones). 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:

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

Again, we do not need the blocked version of this algorithm for the discussion in this paper.

In practice, the use of finite precision arithmetic yields this naive algorithm for numerical accuracy reasons: 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

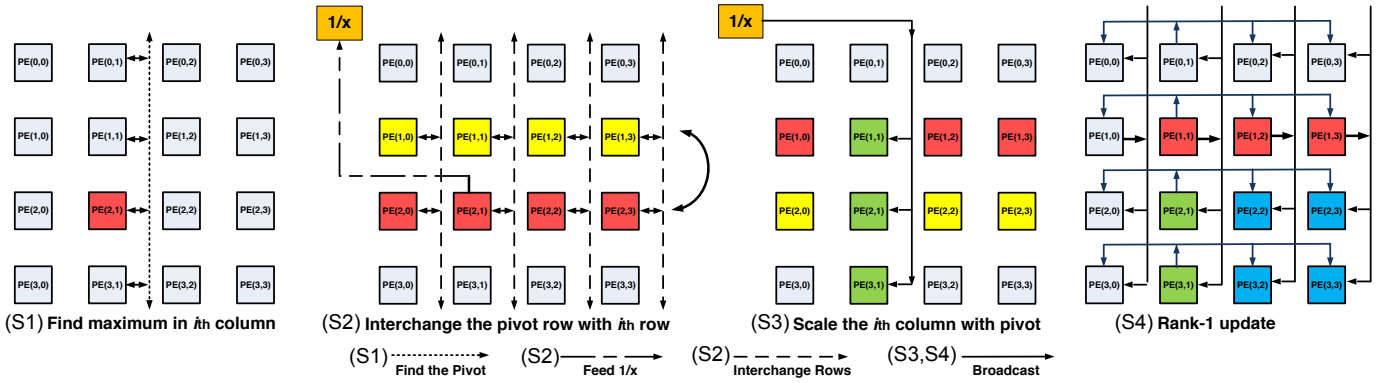

Fig. 3. Second iteration of a $K \times n_r$ LU factorization with partial pivoting on the LAC.

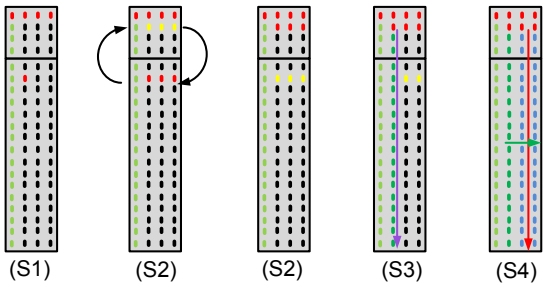

Fig. 4. Operations and data manipulation in the second iteration of a $k \times n_r$ LU factorization inner kernel.

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 such 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.

To study opportunities for corresponding architecture extensions, we focus on how to factor a $kn_r \times n_r$ submatrix (see Figure 4) 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 3, 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. Note that 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 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. Due to these extra complexities, most existing accelerators send the whole $kn_r \times n_r$ block to a host processor to avoid performing the factorization themselves [3], [4]. By contrast, we will discuss a small set of extensions that will allow us to efficiently perform all needed operations and hence the complete LU factorization within dedicated hardware.

C. QR Factorization and Vector Norm

Householder QR factorization is often used when solving a linear least-squares problem. The key to practical QR factorization algorithms is the Householder transformation. 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 $\begin{pmatrix} 1 \\ u_1 \end{pmatrix}$ 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:

$$\left(\begin{array}{c|c} \alpha_{11} & a_{12}^T \\ \hline a_{21} & A_{22} \end{array}\right) := \left(I - \begin{pmatrix} 1 \\ u_2 \end{pmatrix} \begin{pmatrix} 1 \\ u_2 \end{pmatrix}^T / \tau_1\right) \left(\begin{array}{c|c} \alpha_{11} & a_{12}^T \\ \hline a_{21} & A_{22} \end{array}\right)$$

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

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

Algorithm: $\begin{bmatrix}\begin{pmatrix}\rho_1\\ u_2\end{pmatrix}, \tau_1\end{bmatrix} = \text{HOUSEV}\begin{pmatrix}\alpha_1\\ a_{21}\end{pmatrix}$	
	$\chi_2 := \|a_{21}\|_2$
	$\alpha := \left\|\begin{pmatrix}\alpha_1\\ \chi_2\end{pmatrix}\right\|_2 \; (= \|x\|_2)$
$\rho_1 = -\text{sign}(\alpha_1)\|x\|_2$	$\rho_1 := -\text{sign}(\alpha_1)\alpha$
$\nu_1 = \alpha_1 + \text{sign}(\alpha_1)\|x\|_2$	$\nu_1 := \alpha_1 - \rho_1$
$u_2 = a_{21}/\nu_1$	$u_2 := a_{21}/\nu_1$
	$\chi_2 = \chi_2/\|\nu_1\| (= \|u_2\|_2)$
$\tau_1 = (1 + u_2^T u_2)/2$	$\tau_1 = (1 + \chi_2^2)/2$

Fig. 5. Computing the Householder transformation. Left: simple formulation. Right: efficient computation.

factorization of A will now proceed with submatrix A_{22}.

The new complexity introduced in this algorithm is in the computation of u_2, τ_1, and ρ_1 from α_{11} and a_{21}, captured in Figure 5, which require a vector-norm computation and scaling (division). This is referred to as the computation of the Householder vector. We first focus on the computation of the vector norm.

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

$$t = \max_{i=0}^{n-1} |x_i| \, ; \quad y = x/t; \quad \|x\|_2 := t \times \|y\|_2.$$

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, the maximum of all inputs must be found, the vector be normalized, and an extra multiplication is needed to scale the result back. As such, with the exception of the mentioned normalization and the introduction of a matrix-vector multiplication, the overall mapping of QR factorization to the LAC is similar to that of LU factorization. Due to space reasons, we focus our following discussions on the computation of this Householder vector and vector norm only.

To compute the vector norm, 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. On top of being slow, "this algorithm also involves more rounding errors than the unscaled evaluation, which could be obviated by scaling by a power of the machine base [15]". A one-pass algorithm has been presented in [16]. It uses three accumulators for different value sizes. This algorithm avoids overflow and underflow. However, it still needs to perform division. More details about how this is computed in software are discussed in [15], [17].

We now focus on how to perform a vector norm of a scaled $kn_r \times 1$ vector (see Figure 6) when stored in the local store and registers of the LAC (with $n_r \times n_r$ PEs). Recall that such a column is only stored in one column of the LAC. In Figure 6, we show the iterations for calculating a vector norm that is stored in the 3rd column of PEs. The algorithm performs three steps, S1 through S3.

In S1, the third column of PEs starts computing the inner product with half of the vector elements. Simultaneously,

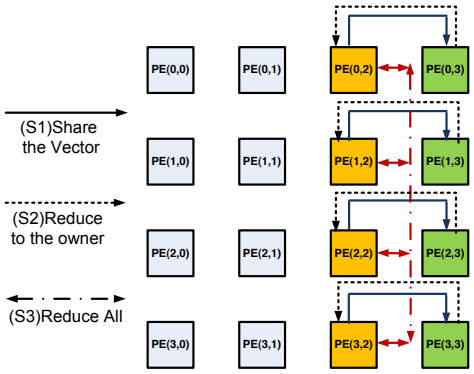

Fig. 6. Mapping of the Vector Norm operation of a single vector stored in the third column of the LAC.

the PEs in this row share the elements of the other half of the vector with the adjacent PEs in the next column (fourth column in Figure 6). PEs in the adjacent column also start performing inner products. After all the PEs in both columns have computed their parts, in S2 the partial inner products are reduced back into the original LAC column, leaving that column with n_r partial results. In S3, a reduce-all operation that requires n_r broadcast operations across the corresponding column bus produces the final vector norm result in all the PEs of the owner column. Thus, performing a vector norm in the LAC is straightforward. The real challenge is the extra complexity to find the maximum value and to scale the vector by it, which is introduced for avoiding overflow and underflow. This will be discussed in the next section.

V. Floating Point Extensions

In this section, we discuss how to overcome the challenges that were introduced in the previous section in relation to the mapping of factorization algorithms on an already very efficient accelerator. These extensions allow an architecture to perform more complex operations more efficiently. We will introduce architecture extensions that provide such improvements specifically for factorizations. However, such extensions also introduce a base overhead in all operations, since they add extra logic and cause more power and area consumption. Corresponding trade-offs will be analyzed in the results section.

A. Hardware Extensions

We start by analyzing opportunities for extensions targeting Cholesky and LU factorization, followed by solutions to complexities in vector norm operations.

1) Cholesky Factorization: In the previous section, we observed that the key complexity when performing Cholesky factorization is the inverse square-root operation. If we add this ability to the core's diagonal PEs, the LAC can perform the inner kernel of the Cholesky factorization natively. The last state of the $n_r \times n_r$ Cholesky factorization will save even more cycles if a square-root function is available. The $n_r \times n_r$ Cholesky factorization is purely sequential with minimal parallelism in rank-1 updates. However, it is a very small part of a bigger, blocked Cholesky factorization. Again, 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.

2) LU Factorization with Partial Pivoting: For LU factorization with partial pivoting, PEs in the LAC must be able to compare floating-point numbers to find the pivot (S1 in Section IV). In the blocked LU factorization, we have used the left-looking algorithm, which is the most efficient variant with regards to data locality [18]. 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 and its latency could be done implicitly without any extra latency and delay penalty.

The next operation that is needed for LU factorization is the reciprocal $(1/X)$. The reciprocal of the pivot needs to be computed for scaling the elements by the pivot (S2 in Section IV). This way, we avoid multiple division operations and simply multiply all the values by the reciprocal of the pivot and scale them.

3) QR Factorization and Vector Norm: In Section IV, we showed how the vector norm operation is performed in conventional computers to avoid overflow and underflow. 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 as in LU factorization, a reciprocal function to scale the vector by the maximum value, again just as in LU factorization, and a square-root unit to compute the length of the scaled vector just as what is needed to optimize the last iteration of a $n_r \times n_r$ Cholesky factorization. However, we can observe that all these extra operations are only necessary due to limitations in hardware representations of real numbers.

Consider a floating number f that, according to the IEEE floating-point standard, is represented as $1.m_1 \times 2^{e_1}$, where $1 \leq 1.m_1 < 2$. Lets 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.

B. Architecture

In this section, we describe the proposed architecture for our floating-point MAC unit and the extensions made to it for matrix factorization applications. We start from a single-cycle accumulating MAC unit and explain the modifications for LU and vector norm operations. Then, we describe the extensions for reciprocal, inverse square-root, and square-root operations.

1) Floating-Point MAC Unit: A floating-point MAC unit with single-cycle accumulation is presented in [20]. Using the same design principles, [13] presents a reconfigurable floating-point MAC that is also able to perform multiplication, addition and multiply-add operations. This design does not support operations on denormalized numbers [20]. We describe our modifications to the same design as shown in Figure 7(a).

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 [21]. It is also not a part of 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 shown how adding an extra exponent bit can overcome overflow/underflow problems in computing the vector norm without the need for performing extra operations to find the biggest value and scale the vector by it. In Figure 7(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 [20], 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.

2) Reciprocal and (Inverse) Square-root Units: In Cholesky factorization, we observed that the LAC needs a way to compute the inverse square-root of the diagonal elements and scale the corresponding column with the result. Adding a square-root unit can also save more cycles in the last iteration of a $n_r \times n_r$ Cholesky factorization. Furthermore, LU factorization needs a reciprocal operation to scale the elements by the pivot. As discussed in [6], a reciprocal unit is also mandatory for TRiangular Solve with Multiple right-hand side (TRSM) operations to support the complete Level-3 BLAS. In this section, we will give details and design options for such a unit.

Division, reciprocal, square-root, and inverse square-root functions are used in many applications in the domain of signal processing, computer graphics, and scientific computing [22], [23]. Several floating-point divide and square-root units have been introduced and studied in the literature [24], [25], [19]. There are mainly two categories of implementations in modern architectures: multiplicative (iterative) and subtractive methods. An extensive presentation of these methods and their

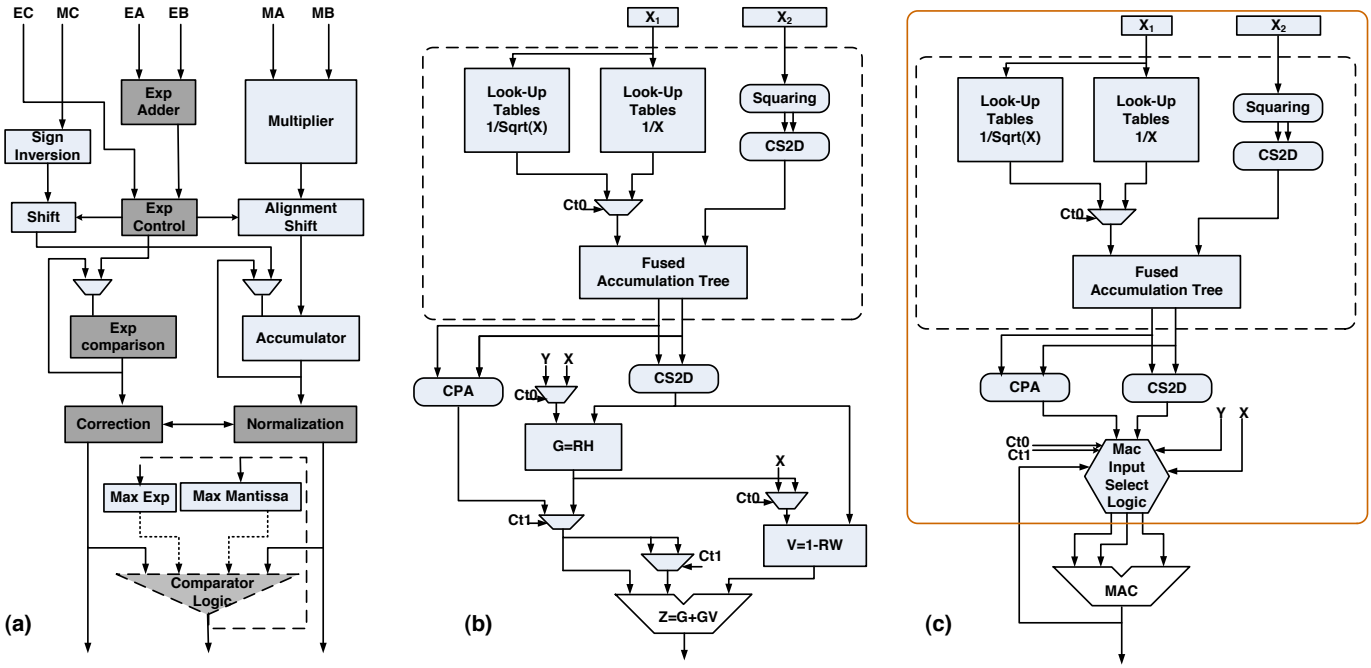

Fig. 7. Floating-point unit extensions: (a) extended reconfigurable single-cycle accumulation MAC unit [13] with addition of a comparator and extended exponent bit-width, where shaded blocks show which logic should change for exponent bit extension; (b) original divide, reciprocal, square-root and inverse square-root design with the Minimax logic [19] used for the isolate unit; (c) 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.

hardware implementations are presented in [23].

Two main multiplicative methods for calculating divide and square-root functions are Newton-Raphson and Goldschmidt's. These algorithms work iteratively to refine an initial approximation. They utilize a look-up table for initial approximation and the number of result digits doubles after each iteration (converging at a quadratic rate). In each iteration, a series of multiplication, subtraction, and shifts are performed, which means a multiply-add unit could be utilized for these operations. Hence, they can be implemented as an enhancement on top of such existing units. Goldschmidt's method, which is based on a Taylor series with two independent multiplication operations, is more suitable for pipelined floating-point units than the Newton-Raphson method.

Subtractive (digit recurrence), which are also known as SRT methods, directly calculate (multiple) digits of the desired result. They have high latency and generally are implemented as a dedicated, complex component. However, there are redundancies between the division and square-root units that allow a single unit to perform both operations. For the higher radix implementations with lower latencies, these designs become complex and area consuming.

In [23], [26], it is concluded that a separate SRT-based subtractive divide and square-root unit is more efficient for a Givens rotation application. This is because multiplicative methods occupy the Multiply-Add (MAD) unit and prevent it to do anything else, while subtractive methods work in parallel with an existing MAC unit, resulting into a faster design.

Given the nature of linear algebra operations and the mapping of algorithms on the LAC, a multiplicative method is chosen. The reason lies within the fact that there are many MAC units in the core, and exploiting one of them for divide or square-root will not harm performance. In our class of applications, a divide and square-root operation is often performed when other PEs are waiting in idle mode for the its result. As the iterations of Cholesky and LU 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.

The design we are considering for this work is the architecture presented in [19]. It uses a 29-30 bit approximation with a second-degree minimax polynomial approximation that is known as the optimal approximation of a function [27]. This approximation is performed by using table look-ups. Then, a single iteration of a modified Goldschmidt's method is applied. This architecture, which is shown in Figure 7(b), guarantees the computation of exactly rounded IEEE double-precision results [19]. It can perform all four operations: divide Y/X, reciprocal 1/X, square-root $\sqrt{X}$, and inverse square-root $1/\sqrt{X}$. While utilizing the same architecture for all operations, the divison/reciprocal operations take less time to be computed, since computing G and V can be done in parallel. In case of square-root/inverse square-root, all operations are sequential and, as a result, the latency is higher. Figure 8 shows the type of operations and control signals that are performed for all four functions.

The design in Figure 7(b) could be reduced to use a single reconfigurable MAC unit, which performs all the computations itself. This strategy reduces the design area and overhead. This reduction does not increase the 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. Therefore, the design with a single reconfigurable MAC unit as shown in Figure 7(c)

Operation	$G = RH$	$V = 1 - RW$	$Z = G + GV$	Ct0 Ct1
Division	$G = RY$	$V = 1 - RX$	$Z = G + GV$	00
Reciprocal	–	$V = 1 - RX$	$Z = R + RV$	01
Squar-root	$G = RX$	$V = 1 - GS$	$Z = G + GV/2$	10
Inv Sqrt	$G = RX$	$V = 1 - GS$	$Z = R + RV/2$	11

Fig. 8. Operations of the divide and square-root unit with control signals [19].

is preferred. 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, the changes we apply to the PEs in the LAC are as follows: all PEs in the LAC design will get the extra-exponent bit and the comparator logic for vector norm and LU with partial pivoting operations, respectively. There are three options for the divide and square-root unit implementation in the LAC: first, a separate unit can be used to be shared by all of PEs, or the top-left PE can be modified to hold the extra logic on top of its MAC unit. A third option is to add the divide and square-root logic to all diagonal PEs. We will evaluate these options and their trade-offs for our applications in the next section.

VI. Experimental Results and Implementations

In this section, we present area, power and performance estimates for the LAC with the modifications introduced in previous sections. We will compare the performance to a pure software-like (micro-coded) implementation of additional complex operations using existing components and micro-programmed state machines. We chose three different problem sizes and we perform an area, power, and efficiency study to evaluate the benefits of these architectural extensions.

A. Area and Power Estimation

Details of the basic PE and core-level implementation of a LAC in 45nm bulk CMOS technology are reported in [5]. For floating-point units, we use the power and area data from [28]. We combine it with complexity and delay reports from [19]. CACTI [29] is used to estimate the power and area consumption of memories, register files, look-up tables and buses.

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 (Figure 7(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 MAC unit in the PE; second, an isolated divide/square-root unit that performs the operation with the architecture in Figure 7(b); and third, an extension to the PEs that adds extra logic and uses the available MAC units in the diagonal PEs (Figure 7(c)).

The comparator is not on the critical path of the MAC pipeline and the extensions for the extra exponent bit are 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 [19]. For a software solution with multiple Godlschmidt iterations, we assume no extra power or area overhead for the micro-programmed state machine.

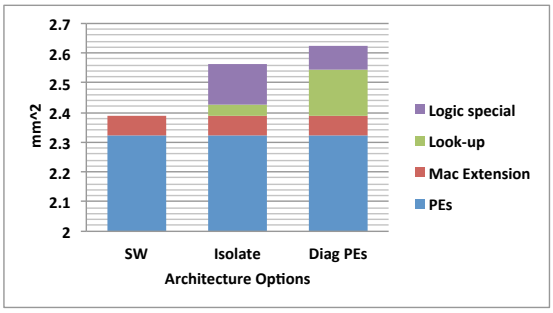

Fig. 9. LAC area break-down with different divide/square-root extensions.

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	524	340	340	62 nJ	60 nJ	60 nJ
128	700	644	644	121 nJ	119 nJ	119 nJ
256	1252	1252	1252	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
Vector norm						
64	282	158	150	32 nJ	29 nJ	29 nJ
128	338	214	206	59 nJ	56 nJ	56 nJ
256	418	294	286	114 nJ	111 nJ	111 nJ
Vector norm With Comparator						
64	276	152	144	23 nJ	20 nJ	20 nJ
128	308	184	176	41 nJ	38 nJ	38 nJ
256	372	248	240	78 nJ	75 nJ	75 nJ
Vector norm With Exponent bit extension						
64	154	80	76	12 nJ	10 nJ	10 nJ
128	170	96	92	21 nJ	19 nJ	19 nJ
256	202	128	124	39 nJ	37 nJ	38 nJ

Fig. 10. 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.

The area overhead on top of the LAC is shown in Figure 9. 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.

B. Performance and Efficiency Analysis

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. A very important point is that even larger problems sizes are usually blocked into smaller subproblems that cast most of the operations into a combination of highly efficient level-3 BLAS operations and the complex inner kernels that we discuss here. Many accelerators only support level-3 BLAS and perform more complex kernels on the host processor. The overhead of

sending the data associated with these computations back and forth is significant and affects the performance by wasting cycles. However, such issues are out of the scope of this paper. What we want to show here is how effective our proposed extensions are in achieving high performance for the inner kernels compared to the baseline architecture with a micro-coded software solution.

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 evaluate 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 inverse square-root operation. We observe (Figure 10) that the number of cycles drops by a third by switching from a software solution to hardware extensions on the LAC.

LU factorization with partial pivoting is not a 2D-scalable algorithm. The pivoting operation and scaling needs to be done for all rows of a given problem size. Hence, for a problem size of $k \times k$, the inner kernel that should be implemented on the LAC is a LU factorization of a $k \times n_r$ block of the original problem. For our studies, we use problems with different $k = 64, 128, 256$, 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 IV, 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 20% speed and 15% energy improvement with the comparator added to the MAC units.

Vector norm as part of a Householder transformation only utilizes a single column of PEs for the inner product and reduce. To measure the maximum achievable efficiency, we assume that there are four different vector norms completing concurrently one in each column. Note that the baseline is the original normalizing vector norm. We have three options for divide/square-root operations, and three options for MAC unit extensions. The first option is a micro-coded software solution, the second option is utilizing the comparator in the MAC unit without an exponent extension, and the last is a MAC unit with an extra exponent bit. The problem sizes are again $k = 64, 128, 256$ different vector lengths. As shown in Figure 10, we can observe that the exponent extension halves the total cycles, and the divide/square-root unit saves up to 30% cycles compared to the baseline. Energy savings reach up to 60% with the exponent bit extension. By contrast, different divide/square-root units do not differ in terms of dynamic energy consumption.

We assume a clock frequency of 1GHz for the LAC. 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. Efficiency in terms of power and area metrics are presented in Figure 11 and Figure 12, respectively. Another metric that we use is the inverse energy-delay. It shows how extensions reduce both latency and energy consumption. Note that for LU factorization, the pivoting operation is also taken into account. Therefore, we used GOPS instead of GFLOPS as performance metric. For LU factorization problems with $k = 64, 128, 256$, we estimated the corresponding total number of operations to be 1560, 3096 and 6168, respectively. For the vector norm, we use the original algorithm as the baseline, which requires 257, 769 or 1025 operations per corresponding vector norm of size $k = 64, 128, 256$. Since our implementation will result in an effective reduction in the number of actually required computations, the extensions have higher GOPS/W than what is reported as peak GFLOPS/W for the LAC in [5].

Results for LU factorization confirm that there is no improvement in efficiency with different reciprocal architectures when solving big problem sizes. Given this fact, isolated unit seems to be a better option for LU factorization. By contrast, vector norm benefits from all types of extension. However, the exponent bit is what brings significant improvements in efficiency.

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 efficiencies 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%.

VII. Conclusions and future work

In this paper, we presented the mapping of matrix factorizations on a highly efficient linear algebra accelerator. We propose two modifications to the MAC unit designs to decrease the complexity of the algorithm. We also show 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 extensions, we applied them to the mapping of Cholesky, LU and QR factorizations on such an improved architecture. Results show that our extensions significantly increase efficiency and performance. Future work includes comparison and mapping of big, tiled matrix factorization problems onto the LAC, including its integration 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] R. Hameed et al., "Understanding sources of inefficiency in general-purpose chips," *ISCA '10*, 2010.

[2] A. Pedram *et al.*, "Co-design tradeoffs for high-performance, low-power linear algebra architectures," *IEEE Trans. on Computers*, 2012.

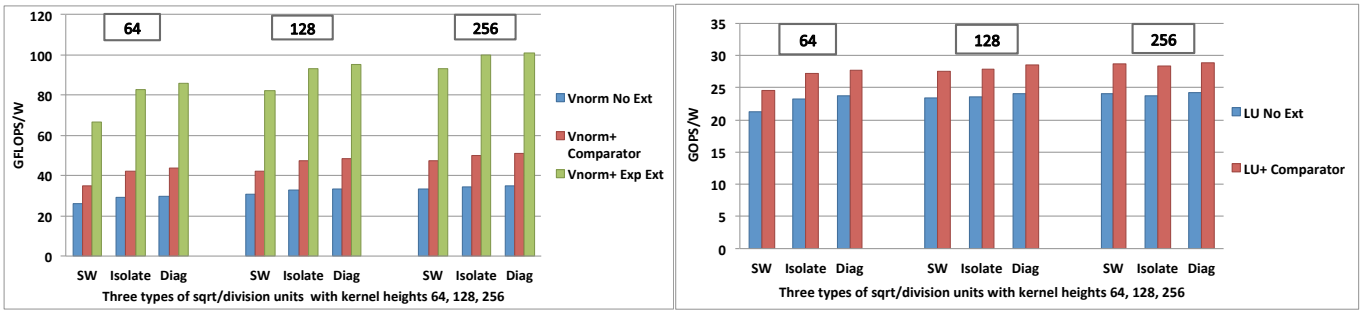

Fig. 11. The effect of hardware extensions and problem sizes on the power efficiency. Left: vector norm, Right: LU factorization

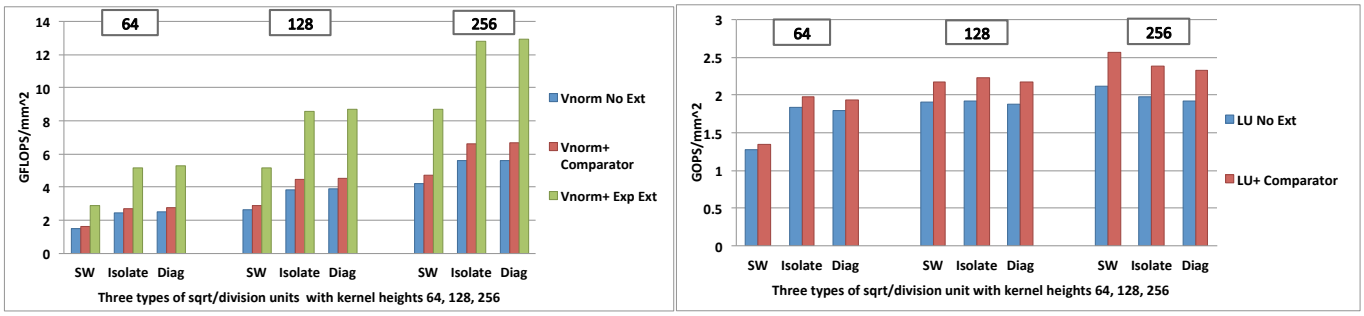

Fig. 12. The effect of hardware extensions and problem sizes on the area efficiency. Left: vector norm, Right: LU factorization

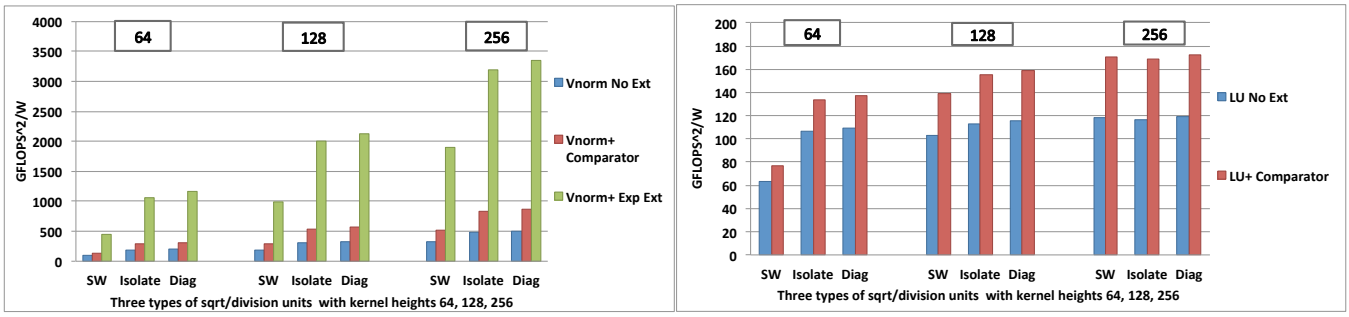

Fig. 13. The effect of hardware extensions and problem sizes on the inverse E-D metric. Left: vector norm, Right: LU factorization

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

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

[7] 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.

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

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

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

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

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

[13] 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.

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

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

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

[17] 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.

[18] 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.

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

[20] 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.

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

[22] S. F. Oberman *et al.*, "Design issues in division and other floating-point operations," *IEEE Trans. Comput.*, vol. 46, no. 2, February 1997.

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

[24] 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.

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

[26] P. Soderquist *et al.*, "Division and square root: choosing the right implementation," *IEEE Micro*, 1997.

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

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

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