ACCELERATING LINEAR ALGEBRA KERNELS ON A MASSIVELY PARALLEL RECONFIGURABLE ARCHITECTURE

A. Soorishetty⋆*, J. Zhou*⋆*, S. Pal* †*, D. Blaauw* †*, H. Kim* †*,T. Mudge* †*, R. Dreslinski* †*, C. Chakrabarti*⋆

⋆Arizona State University, Tempe, AZ 85287
†University of Michigan, Ann Arbor, MI 48109

ABSTRACT

Much of the recent work on domain-specific architectures has focused on bridging the gap between performance/efficiency and programmability. We consider one such example architecture, Transformer, consisting of light-weight cores interconnected by caches and crossbars that supports run-time reconfiguration between shared and private cache mode operations. We present customized implementation of a select set of linear algebra kernels, namely, triangular matrix solver, LU decomposition, QR decomposition and matrix inversion, on Transformer. The performance of the kernel algorithms is evaluated with respect to execution time and energy efficiency. Our study shows that each kernel achieves high performance for a certain cache mode and that this cache mode can change when the matrix size changes, making a case for run-time reconfiguration.

1. INTRODUCTION

Linear algebra algorithms are important data processing kernels in scientific computing, statistics, and machine learning. Due to their inherent high complexity and different data access patterns, these kernels form the bottlenecks in real-time applications.

Linear algebra kernels have been accelerated by Application-Specific Integrated Circuits (ASIC) [1, 2], general-purpose graphic processing units (GPGPU) [3, 4, 5, 6] and Field Programmable Gate Arrays (FPGA) [7, 8]. A popular alternative to FPGAs is Coarse-Grained Reconfigurable Architectures (CGRA) which have the advantages of shorter reconfiguration time and lower power consumption [9]. Some examples include REDEFINE [10], ADRES [11], DySER [12], and LAC [13, 14, 15], where the data-path functional units can be reconfigured based on the computation requirements. These architectures achieve ASIC-level efficiencies while supporting multiple memory access patterns. For instance, LAC shows that orders of magnitude improvements in efficiency is possible with relatively simple customizations and fine-tuning of memory hierarchy configurations. PLASTICINE is a recently proposed architecture which achieves high performance per watt compared to FPGAs [16]. Here both the data-path computing units and the on-chip scratchpad access patterns are configurable.

In this work, we demonstrate the superior performance of linear algebra kernel algorithms mapped onto a reconfigurable architecture, Transformer. Transformer consists of multiple in-order cores connected to a memory interface through a two-level on-chip memory hierarchy consisting of reconfigurable caches and crossbars. It is programmable using standard high-level languages such as C/C++. We focus on the implementation of certain key linear algebra kernels, namely, Triangular Matrix Solver (TRSM), LU Decomposition (LUD), QR Decomposition (QRD) and Matrix Inversion on Transformer. Other linear algebra kernels such as Matrix-Vector and Matrix-Matrix multiplications have been well studied and are not included here. For each of the linear algebra kernels, we focus on different scheduling schemes and different Transformer L1 and L2 cache configurations. We evaluate the performance of the kernel algorithms with respect to execution times, giga-operations per second per Watt (GOPS/W) and giga-floating-point-operations per second per Watt (GFLOPS/W) for 14nm technology node. This paper makes the following contributions:

- Describes efficient implementations of linear algebra kernels such as TRSM, LUD, QRD and Matrix Inversion on a multicore reconfigurable architecture.
- Provides performance analysis when operating on different cache modes for a wide range of input data sizes.
- Demonstrates peak performance of 97.5, 59.4, 133.0 and 82.5 GFLOPS/W for TRSM, LUD, QRD and Matrix Inverse, respectively, on a 16x4 multicore architecture.

The rest of the paper is divided into the following sections. Section 2 gives a brief introduction to the Transformer architecture. Section 3 discusses the algorithm mappings, followed by results and evaluation in Section 4, and conclusion in Section 5.

2. TRANSFORMER ARCHITECTURE

Transformer is a scalable, energy-efficient, reconfigurable design similar to a recent sparse matrix multiplication accelerator [17, 18, 19]. It consists of many in-order General-purpose Processing Elements (GPEs), distributed on-chip cache memories, crossbars and a high-bandwidth DDR interface. A block diagram with 4 tiles and 16 GPEs per tile is shown in Fig. 1.

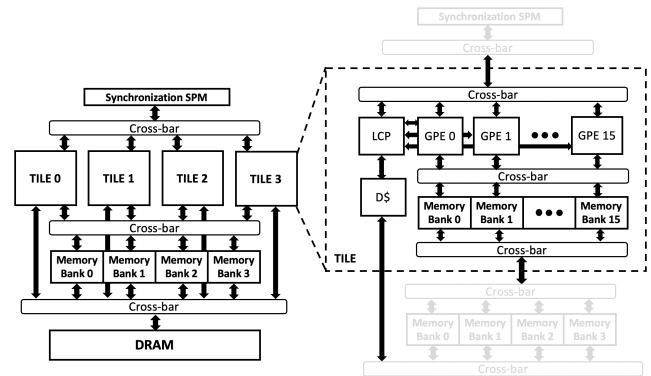


Fig. 1. Transformer architectural overview.

The GPE is a simple, in-order core that has a small silicon footprint. Multiple GPEs are coordinated and synchronized by a Local

Authorized licensed use limited to: University of Michigan Library. Downloaded on June 02,2020 at 13:48:31 UTC from IEEE Xplore. Restrictions apply.

Control Processor (LCP). Both types of cores have hardware support for single-precision floating point operations. Transformer has two layers of cache-crossbar hierarchy, namely the L1 and L2. The cache layers are write-back and multi-banked for high throughput and scalability. The number of L1 banks is equal to the number of GPEs in a tile, and the number of L2 banks is the same as the number of tiles. This allows for the cache hierarchy to be easily reconfigured at run-time between private mode and shared mode. The crossbars are based on swizzle-switch networks that have been shown to be more scalable and energy-efficient compared to other on-chip networks [20]. The LCP has a separate datapath to main memory. Lastly, Transformer has a global synchronization SPM that can be accessed by all GPEs and LCP. This on-chip memory can be used for implementation of software coherence and standard primitives such as locks, condition variables, barriers, and semaphores.

3. MAPPING ON TRANSFORMER

In this section, we give a brief introduction of the four linear algebra kernels along with their implementation on Transformer.

3.1. Triangular Matrix Solver (TRSM)

This Level-3 BLAS (Basic Linear Algebra Subprograms) routine solves a system of linear equations of the form $AX = B$, where A is an upper or lower triangular matrix, and X & B are dense matrices. Depending on whether A is an upper or a lower triangular matrix, this algorithm employs backward or forward substitution. In both cases, a column of X can be computed using matrix A and a column of B.

The A and B matrices are initially stored in the DRAM and copied into L1 banks through L2. As the columns of X can be computed in parallel and since there is a serial dependency of computations within a column, each GPE is assigned the task of computing one or more columns of X. Columns of B and matrix A are copied into each L1 bank if a private cache is used or a single copy is shared among all the L1 banks if a shared cache is used. Each GPE computes the X values in a column one-by-one and stores them in the L1 cache. After an entire column of X is solved, the updated values are flushed to the DRAM through L2.

When the size of the matrix is very large, the computations within a column can be partitioned into n blocks. In such a case, the L1 cache holds a part of A, B and X. The X values from block (i) are used to compute the values of block $(i + 1)$, and so on. This method also requires flushing data into DRAM / L2 after computation of every block.

3.2. LU Decomposition (LUD)

LUD involves factorizing a square matrix A into a product of a lower triangular matrix, L and an upper triangular matrix, U, given by $A = LU$. Here, LUD is computed by Gaussian elimination where U overwrites A; L is stored separately.

Unlike TRSM, where the computations on columns are done in parallel, in LUD there is a computational dependency across columns. Assume that each row is assigned to a GPE, so row k is assigned to GPE k. The first approach (v1) is to parallelize each column-update, that involves computing one column of L and one row of U. For $N = 64$, in the first column-update, the first row is used as a pivot to compute the multiplication coefficients for all the rows, as shown by Loop 1 in Fig.2. These coefficients are then used to update the matrix A shown in Loop 2 by GPEs 2 through 64. The updated pivot row is the first row of U, and the updated A values along the first column form the first column of L. Now, for the second column-update, the second row is used as pivot and the same procedure is followed to get the second row of U and second column of L. This process repeats until we reach $A_{64,64}$. The updated values of each column-update stay in L1 and are flushed to DRAM by performing a cache flush after every column-update.

```
        for k = 1:N
Loop 1 [  for j = k:N
            L_jk = A_jk / A_kk
Loop 2 [  for j = k+1:N
            for i = k+1:N
              A_ij = A_ij – L_ik * A_kj
```

Fig. 2. Pseudo code for LUD when A is of size $N \times N$

The main drawback in this mapping is that the rows above the pivot row are not used in further computations and so the GPEs assigned to those rows become idle. Thus, the number of idle GPEs increase with every column-update. In a 4-tile Transformer design, if each tile is assigned $N/4$ consecutive rows, then GPEs in tile 0 become inactive after $N/4$ rows have been processed. Even inside a tile, the workload distribution is uneven. Assigning non-consecutive rows to tiles may help all the tiles remain active until the end, but the overall GPE utilization still stays low. In Section 4.4, we demonstrate how uneven workload distribution across tiles can be utilized to power down idle tiles, thereby saving power.

For higher utilization, an alternative method (v2) [21] is to divide the matrix into blocks and solve using a combination of LUD, GEMM (General Matrix - Matrix multiplication) and TRSM calls. Dividing the matrix into four blocks makes use of the following set of computations:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{bmatrix}$$

LUD: $A_{11} = L_{11}U_{11}$;
TRSM: $A_{21} = L_{21}U_{11}$;
TRSM: $A_{12} = L_{11}U_{12}$;
GEMM followed by LUD: $A_{22} - L_{21}U_{12} = L_{22}U_{22}$

The LUD step ($A_{11} = L_{11}U_{11}$), is implemented using the above method. Since the LUD works on a smaller matrix, the imbalance in the workload distribution is not as much. This four block method can be extended to support dividing the matrix into many more blocks.

3.3. QR Decomposition (QRD)

QRD involves factorizing a square / non-square matrix A into a product of an orthogonal matrix Q and an upper triangular matrix R, given by $A = QR$. Here, R overwrites A. Each element below the diagonal is annihilated from the last row using Given's rotation [22].

QRD has computational dependency along rows as well as along columns. Since the computations across the columns are independent of each other, each column is assigned to a GPE. The computations start by first annihilating the last value of the first column and proceeding to annihilate elements above (along column) and to the right (along row). For the case when $N = 64$, every GPE is assigned one column. All GPEs in tile 1 have to wait till tile 0 has annihilated the first 16 entries of a row. In the same way tile 2 has to wait till tile 1 has annihilated the next 16 entries. As each GPE is responsible for annihilating one column of A, all the rows are updated multiple times across all the tiles. To ensure that tiles 1 and 2 work on the updated rows, a L1 cache flush is required after every row update. So

Authorized licensed use limited to: University of Michigan Library. Downloaded on June 02,2020 at 13:48:31 UTC from IEEE Xplore. Restrictions apply.

in this mapping a total of $48 + 32 + 16$ cache flushes are required. Each cache flush costs 0.2 - 0.4 μs and so in order to avoid overhead of cache flushes, we chose to implement QRD using only one tile.

In the single tile implementation, after the computation of the first 16 columns is done, computations on the next set of 16 columns start. We ensure that the serial dependency across columns is respected by storing the status of the columns (annihilated or not) using the synchronization scratchpad. For all computations, data is read from the L1 banks and the updated row values are stored back into the L1 banks. The final Q and R values reside in the L1 banks.

3.4. Matrix Inversion

The inverse of a matrix is one which when multiplied by the original matrix results in an identity matrix, $AA^{-1} = I$ (or) $A^{-1}A = I$, where A, A^{-1} and I are square matrices.There are many methods to invert a matrix. Here, we use a combination of LUD and TRSM to compute A^{-1}. The steps are (i) LUD: $A = LU$, (ii) TRSM: $LY = I$ and (iii) TRSM: $UX = Y$. The TRSM and LUD kernels are mapped as described in Sections 3.1 and 3.2, respectively.

4. RESULTS

Transformer is modeled using the gem5 [23] architectural simulator. The architectural configuration used here has 4 tiles with 16 GPEs per tile and two levels of cache hierarchy. The L1 cache bank size is fixed at 4kB and the L2 cache bank size is fixed at 64kB. The simulation statistics such as execution time, power, GOPS and GFLOPS are obtained from gem5 statistics. The cache configurations used here are:

1. L1 shared cache, L2 shared cache (L1S, L2S)
2. L1 shared cache, L2 private cache (L1S, L2P)
3. L1 private cache, L2 shared cache (L1P, L2S)
4. L1 private cache, L2 private cache (L1P, L2P)

The power calculation is based on the raw power parameters, i.e. static power and dynamic power (or transaction energy per access) for the 14 nm node and the gem5 statistics that include active cycles and access numbers. The static power and transaction energy per access of the reconfigurable cache banks are obtained from CACTI 7.0 cache models [24] and those for the crossbars are obtained from Sewell et al. [20] (scaled from 32 nm to 14 nm). The power data has been validated against a prototype chip in 40 nm [19, 18] of the sparse matrix multiplication accelerator [17] and scaled down to 14 nm technology node.

4.1. Triangular Matrix Solver

Table 1 shows the execution times (in ms) for different cache configurations when operating on matrix sizes of 128×128 to 1024×1024. We see that for small matrix sizes, L1P, L2P has the best performance while for larger matrix sizes L1S, L2P does better. L1P, L2P mode does not perform as well for larger sizes since the memory requirement is larger than what the L1 cache bank can support. The maximum requirement for the computations is 2*N*4 which translates to 4kB (the size of L1 cache) when N=512.

N	L1S, L2S	L1S, L2P	L1P, L2S	L1P, L2P
128	0.15	0.15	0.137	**0.132**
256	1.28	1.33	0.989	**0.981**
512	10.33	10.12	9.32	**7.22**
1024	104.55	**85.4**	178.61	143.7

Table 1. TRSM: Execution times (in ms) for different matrix sizes

Fig.3 shows the GFLOPS/W for different matrix sizes for all the cache modes. Here too we see that L1P, L2P does better until $N = 512$; when $N > 512$, L1S, L2P has significantly better performance. Our GFLOPS/W numbers are fairly high with 84-96 GFLOPS/W for $N < 512$. The GOPS/W values for $N = 128$, 256, 512 and 1024 are also high at 240, 250, 261 and 205, respectively.

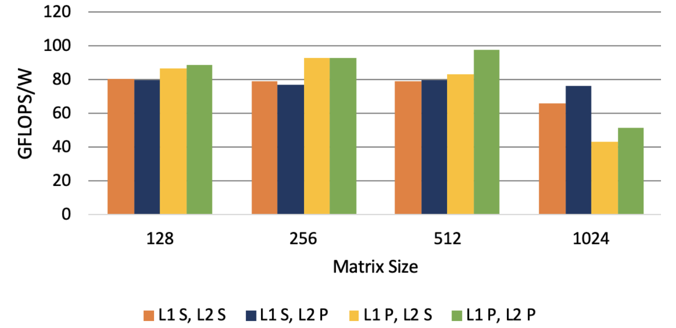
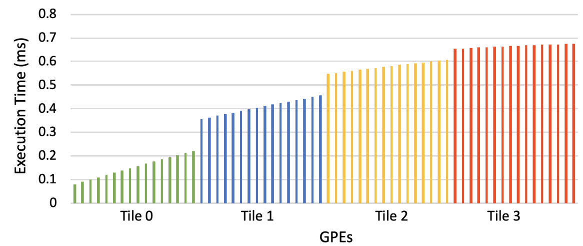


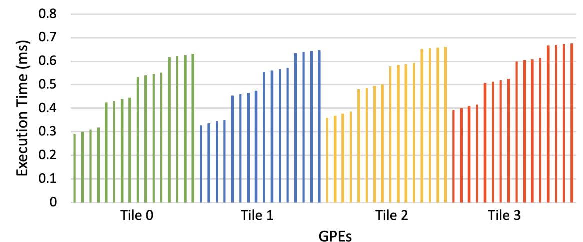
Fig. 3. TRSM: GFLOPS/W for different matrix sizes.

4.2. LU Decomposition

As mentioned in Section 3.2, LUD v1 can be scheduled by assigning consecutive or non-consecutive rows to each GPE. Fig. 4 shows the time a GPE is active when N=128. In the first approach (Fig. 4(a)), 32 consecutive rows are assigned to each tile, and so GPE0 in tile 0 is assigned 1st and 17th rows. In the second approach (Fig. 4(b)), consecutive rows are assigned to consecutive GPEs across tiles and so GPE0 is assigned 1st and 65th rows. In both approaches, each GPE computes on two rows. The main advantage of approach 1 is that we can shut down the tiles after they are done computing their rows, thus saving power. This technique, on average, saves power up to 27%.



(a) Consecutive rows assigned to GPEs



(b) Non-consecutive rows assigned to each GPE

Fig. 4. LUD: The time for which each GPE is active in the two scheduling schemes for v1.

Table 2 shows the execution times (in ms) of both v1 and v2 for different cache configurations when operating on matrix sizes of

Authorized licensed use limited to: University of Michigan Library. Downloaded on June 02,2020 at 13:48:31 UTC from IEEE Xplore. Restrictions apply.

128×128 to 1024×1024. We can clearly see that v2 outperforms v1 for all data sizes and for all cache modes except for L1P, L2S. Recall that v2 uses v1 mapping for computing LUD at the block level. We also see that until $N = 256$, L1S, L2S performs well

N	L1S, L2S		L1S, L2P		L1P, L2S		L1P, L2P	
	v1	v2	v1	v2	v1	v2	v1	v2
128	0.67	**0.46**	0.69	0.5	0.69	0.57	0.67	0.55
256	3.58	**2.06**	3.44	2.1	3.52	3.78	3.64	2.9
512	25.61	12.96	25.52	**12.62**	27.63	40.05	24.5	19.52
1024	168.66	99.6	169.62	**97.25**	371.25	382.6	157.17	143.77

Table 2. LUD: Execution times (in ms) for different matrix sizes

and then for $N > 256$, L1S, L2P has slightly better performance. Here, L1 private cache does not have good performance because the scheduling pattern in v1 divides each column-update across the tiles. So, a shared configuration for L1 does better.

Fig. 5 shows the GFLOPS/W for v2. There is a linear increase in GFLOPS/W as the matrix size increases until N=512. The numbers for $N = 256, 512$ are close to 59 GFLOPS/W. The GOPS/W for the best cache configuration for N=128, 256, 512 and 1024 are 99, 131, 155.83 and 154.04, respectively.

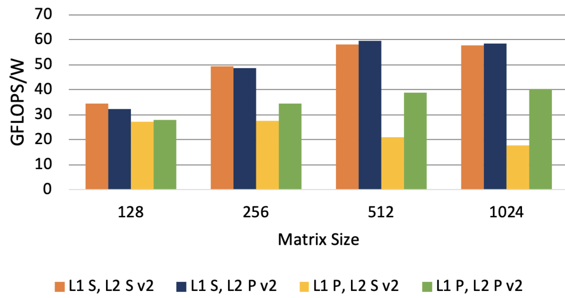


Fig. 5. LUD: GFLOPS/W for different matrix sizes

4.3. QR Decomposition

We evaluate the performance of QRD on matrix sizes of 64×64, 128×128, and 256×256. As we use only a single tile, there is no difference in using L2 in shared or private mode. The execution times for L1P, L2S are 0.35 ms, 1.54 ms and 13.3 ms for $N = 64$, 128 and 256, respectively. L1P, L2S works better for all the matrix sizes as each GPE works independently on a column.

Fig.6 shows the GFLOPS/W for different matrix sizes. We see that the GFLOPS/W is as high as 130 for N=256. The GOPS/W for N=64, 128 and 256 are 124.93, 183.7 and 182.75, respectively.

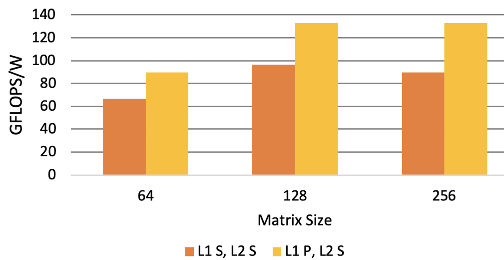


Fig. 6. QRD: GFLOPS/W for different matrix sizes

4.4. Matrix Inversion

The Matrix Inversion kernel is composed of three different kernels: LUD, forward substitution and backward substitution. An analysis of execution time shows that LUD is dominant with 42.79%, followed by forward substitution at 29.96% and backward substitution at 27.75%. Table 3 shows the execution times (in ms) for different cache configurations when operating on matrix sizes of 128×128 to 1024×1024. The 'reconfiguration' scheme, which uses the best cache mode for each kernel, has the best performance.

N	L1S, L2S	L1S, L2P	L1P, L2S	L1P, L2P	Reconfig.
128	0.78	0.82	0.84	0.86	**0.72**
256	4.83	4.95	5.76	5.14	**3.9**
512	33.71	32.93	58.7	33.96	**27.32**
1024	279.66	**268.06**	739.82	385.22	**268.06**

Table 3. Matrix Inverse: Execution times (in ms) for different matrix sizes

Next, we present GFLOPS/W results for the different configurations in Fig.7. We see that reconfiguration helps increase GFLOPS/W. For instance, for $N = 512$ use of reconfiguration helps increase GFLOPS/W from 72.67 (L1S, L2P) to 83.05. Since reconfiguring cache mode only costs 1 cycle, support for reconfiguration is a winning proposition.

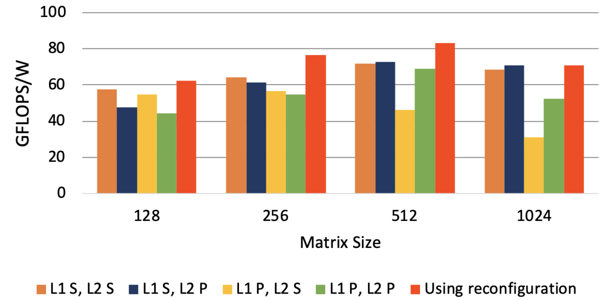


Fig. 7. Matrix Inverse: GFLOPS/W for different matrix sizes

5. CONCLUSION

In this paper, we presented the implementations of four linear algebra kernels onto a massively parallel reconfigurable architecture, Transformer. We investigated the performance of these kernels when the L1 and L2 caches operated in shared and private modes. We found that the optimal cache mode was different for each kernel. For instance, for smaller matrix sizes, L1P, L2P was best for TRSM while L1S, L2S was best for LUD and L1P, L2S was best for QRD. For each kernel, the optimal cache mode changed when the matrix size increased. For instance, for LUD, the optimal cache mode changed from L1S, L2S to L1S, L2P for $N > 512$. The reconfigurable cache mode features are utilized in the implementation of matrix inverse. For $N = 512$, implementing the LUD step using L1S, L2P and the TRSM step using L1P, L2P resulted in an improvement of 10.38 GFLOPS/W compared to operating in the L1S, L2P mode.

Acknowledgment: The material is based on research sponsored by Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under agreement number FA8650-18-2-7864. The views and conclusions contained herein are those of the authors and do not represent the official policies or endorsements, either expressed or implied, of ARFL and DARPA or the U.S. Government.

Authorized licensed use limited to: University of Michigan Library. Downloaded on June 02,2020 at 13:48:31 UTC from IEEE Xplore. Restrictions apply.

6. REFERENCES

[1] S. Steffl and S. Reda, "Lacore: A supercomputing-like linear algebra accelerator for soc-based designs," in *2017 IEEE International Conference on Computer Design (ICCD)*, Nov 2017, pp. 137–144.

[2] D. Patel, M. Shabany, and P. G. Gulak, "A low-complexity high-speed qr decomposition implementation for mimo receivers," in *2009 IEEE International Symposium on Circuits and Systems*, May 2009, pp. 33–36.

[3] J. Humphrey, D. Price, K. Spagnoli, A. Paolini, and E. Kelmelis, "CULA: hybrid GPU accelerated linear algebra routines," in *Modeling and Simulation for Defense Systems and Applications V*. International Society for Optics and Photonics, 2010, vol. 7705, pp. 9 – 15, SPIE.

[4] F. Magoules and A. Ahamed, "Alinea: An advanced linear algebra library for massively parallel computations on graphics processing units," *The International Journal of High Performance Computing Applications*, vol. 29, no. 3, pp. 284–310, 2015.

[5] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra, "Dense linear algebra solvers for multicore with gpu accelerators," in *2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*, April 2010, pp. 1–8.

[6] R. Nath, S. Tomov, and J. Dongarra, "Accelerating gpu kernels for dense linear algebra," in *High Performance Computing for Computational Science – VECPAR 2010*, Berlin, Heidelberg, 2011, pp. 83–92, Springer Berlin Heidelberg.

[7] F. A. Khan, R. A. Ashraf, Q. H. Abbasi, and A. A. Nasir, "Resource efficient parallel architectures for linear matrix algebra in real time adaptive control algorithms on reconfigurable logic," in *2008 Second International Conference on Electrical Engineering*, March 2008, pp. 1–9.

[8] W. José, A. R. Silva, H. Neto, and M. Véstias, "Analysis of matrix multiplication on high density virtex-7 fpga," in *2013 23rd International Conference on Field programmable Logic and Applications*, Sep. 2013, pp. 1–4.

[9] M. Wijtvliet, L. Waeijen, and H. Corporaal, "Coarse grained reconfigurable architectures in the past 25 years: overview and classification," in *Proceedings - 16th International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation, SAMOS 2016*, 1 2017, pp. 235–244.

[10] M. Alle, K. Varadarajan, A. Fell, N. Joseph, S. Das, P. Biswas, J. Chetia, A. Rao, SK. Nandy, R. Narayan, et al., "Redefine: Runtime reconfigurable polymorphic asic," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 9, no. 2, pp. 11, 2009.

[11] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "Adres: An architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix," in *International Conference on Field Programmable Logic and Applications*. Springer, 2003, pp. 61–70.

[12] V. Govindaraju, C. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim, "Dyser: Unifying functionality and parallelism specialization for energy-efficient computing," *IEEE Micro*, vol. 32, no. 5, pp. 38–51, 2012.

[13] A. Pedram, A. Gerstlauer, and R. A. Van De Geijn, "Floating point architecture extensions for optimized matrix factorization," in *2013 IEEE 21st Symposium on Computer Arithmetic*, 2013, pp. 49–58.

[14] A. Pedram, A. Gerstlauer, and R. A. Van De Geijn, "Algorithm, architecture, and floating-point unit codesign of a matrix factorization accelerator," *IEEE Transactions on Computers*, vol. 63, no. 8, pp. 1854–1867, 2014.

[15] A. Pedram, S. Z. Gilani, N. S. Kim, R. v. d. Geijn, M. Schulte, and A. Gerstlauer, "A linear algebra core design for efficient level-3 blas," in *2012 IEEE 23rd International Conference on Application-Specific Systems, Architectures and Processors*, July 2012, pp. 149–152.

[16] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun, "Plasticine: A reconfigurable architecture for parallel patterns," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017, pp. 389–402.

[17] S. Pal, J. Beaumont, D. Park, A. Amarnath, S. Feng, C. Chakrabarti, H. Kim, D. Blaauw, T. Mudge, and R. Dreslinski, "Outerspace: An outer product based sparse matrix multiplication accelerator," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2018, pp. 724–736.

[18] Dong-Hyeon Park, Subhankar Pal, Siying Feng, Paul Gao, Jielun Tan, Austin Rovinski, Shaolin Xie, Chun Zhao, Aporva Amarnath, Timothy Wesley, et al., "A 7.3 m output non-zeros/j, 11.7 m output non-zeros/gb reconfigurable sparse matrix-matrix multiplication accelerator," *IEEE Journal of Solid-State Circuits*, 2020.

[19] S. Pal, D. Park, S. Feng, P. Gao, J. Tan, A. Rovinski, S. Xie, C. Zhao, A. Amarnath, T. Wesley, et al., "A 7.3 m output non-zeros/j sparse matrix-matrix multiplication accelerator using memory reconfiguration in 40 nm," in *2019 Symposium on VLSI Technology*. IEEE, 2019, pp. C150–C151.

[20] K. Sewell, R. Dreslinski, T. Manville, S. Satpathy, N. Pinckney, G. Blake, M. Cieslak, R. Das, T. Wenisch, D. Sylvester, et al., "Swizzle-switch networks for many-core systems," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 2, no. 2, pp. 278–294, 2012.

[21] K. A. Gallivan, R. J. Plemmons, and A. H. Sameh, "Parallel algorithms for dense linear algebra computations," *SIAM Rev.*, vol. 32, no. 1, pp. 54–135, Mar. 1990.

[22] W. Givens, "Computation of plane unitary rotations transforming a general matrix to triangular form," *Journal of the Society for Industrial and Applied Mathematics*, vol. 6, no. 1, pp. 26–50, 1958.

[23] N. Binkert, B. Beckmann, G. Black, S. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. Hower, T. Krishna, S. Sardashti, et al., "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[24] R. Balasubramonian, A. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "Cacti 7: New tools for interconnect exploration in innovative off-chip memories," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 2, pp. 14:1–14:25, June 2017.

Authorized licensed use limited to: University of Michigan Library. Downloaded on June 02,2020 at 13:48:31 UTC from IEEE Xplore. Restrictions apply.