Maia: Matrix Inversion Acceleration Near Memory

Bahar Asgari*, Dheeraj Ramchandani+, Amaan Marfatia+, and Hyesoon Kim+

*University of Maryland, College park
+Georgia Institute of Technology

***Abstract*—Matrix inversion is an essential and challenging operation in several application domains, such as scientific computing, social networks, and recommendation systems. Since matrix inversion is a memory-bound task, it has the potential of being implemented near memory to efficiently use high memory bandwidth. However, data-dependency patterns in the common matrix-inversion algorithms limit memory bandwidth utilization. To minimize the negative impact of such dependencies on performance, we propose matrix inversion acceleration (Maia), a near-memory FPGA-based implementation of matrix inversion that converts the mathematical dependencies to gate-level dependencies thus reduces the critical-path latency. We implement and evaluate Maia on a high-end Xilinx Ultrascale+ xcu280 FPGA connected to a high-bandwidth memory (HBM2), targeting the data-center Alveo U280 boards. Maia performs matrix inversion $4\times$ faster than a baseline FPGA implementation without the proposed techniques for resolving dependencies.**

***Index Terms*—Near-Memory Processing, Matrix Inversion, Memory Bandwidth.**

I. Introduction

Inversion of matrices is an essential operation in several scientific computing applications [1]–[3], such as model reduction, polar decomposition, optimal control and prediction, statistics, dynamics analysis, and model reduction, as well many data scientific applications [4]–[6] such as signal processing, network analysis and collaborative recommendation. Various methods including Gauss-Jordan Elimination, Cholesky Decomposition and Monte-Carlo exist to calculate the matrix inversion. Unlike sparse matrix multiplication that can be accelerated using several recently proposed techniques [7]–[9], the inversion of *large* matrices is a key challenge as the time complexity of a numerically stable procedures for matrix inversion is $O(n^3)$. Therefore, prior studies have proposed optimized implementations for various platforms. For instance, a study [10] implemented the Gauss-Jordan algorithm for matrix inversion on GPUs, using the inherent parallelism in the algorithm on the CUDA cores by decomposing the matrix into columns, each of which processed by a group of threads.

For further acceleration, hardware and FPGA implementations of the matrix inversion have been suggested [11]–[17]. For instance, a hardware implementation [17] improved the pipelining in matrix inversion by implementing the lower-upper (LU) decomposition method on systolic arrays – decomposition techniques such as LU decomposition are used to simplify the time-consuming matrix inversion. However, such hardware and FPGA solutions are not scalable and are fast and efficient only in small scales. The systolic-based solution [17], for example, suffers from the scalability issues inherent with systolic arrays. The other hardware-based solution for matrix inversion is block-based approaches [1], [3], [18]–[20]. For instance, a blocked-LU on network on chip [21] divides the matrix into sub-matrices which are processed by processing nodes connected in a mesh. In this method, improvement in the execution time as a result of parallelism was offsetted by the communication overhead between processing nodes. Another work [3] uses a similar approach for parallelizing the matrix inversion using Gauss-Jordan method by dividing the matrix into several sub-matrices and then mapping each sub-matrix to a processing core where main core performs the operation, and the update process is parallelized.

The key operation behind matrix inversion is a reduction operation that can potentially be parallelized. However, because of data dependencies in the algorithm of matrix inversion, the aforementioned GPU optimizations and FPGA implementations are not able to fully parallelize the inversion. In addition, because of its low ratio of operations per byte, matrix inversion is a memory intensive kernel that can potentially take advantage of near-data processing if data dependencies did not exist. To reduce the negative impact of such dependencies, this paper proposes Maia[1], a **ma**trix **i**nversion **a**ccelerator near memory. Maia breaks down the mathematical dependencies into smaller ones and implements them as gate-level dependencies to reduce the critical-path latency. Maia represents the resulted dependencies as a dependency graph and implements it on an Ultrascale Xilinx FPGA connected to a high bandwidth memory (HBM2). Maia proposes an optimal scheme to partition the graph to reduce the total execution time of the inversion algorithm. Maia also investigates the trade-off between throughput and resource utilization of FPGA. Maia performs matrix inversion $4\times$ faster than an optimized baseline FPGA implementation without the proposed techniques for resolving dependencies.

II. Challenge

In the following, we explore calculating the invert of A using a decomposed matrix (i.e., the outcome of LU). LU decomposition factors a matrix as the product of lower and upper triangular matrices: $A = LU$. Therefore, in the following, whenever A is used, it indicates a matrix consisting of L and U. After decomposition, the inverse of A is calculated as

[1]Maia is a star in the constellation of Taurus.

```
for j = 0 to W
   for i = 0 to W                                   Phase 1
      for k = 0 to i
         IA[i][j] = IA[i][j] - A[i][k] * IA[k][j]

   for i = W-1 to 0                                 Phase 2
         for k = i+1 to k
            IA[i][j] = IA[i][j] - A[i][k] * IA[k][j]

         IA[i][j] = IA[i][j] / A[i][j]
```

Fig. 1. **LU algorithm:** calculating the inverse of A. In this paper, we call this nested loop (which is a matrix-vector operation) an *invert*: $IA = invert(A, IA)$, which reads matrix A and the vector IA and updates IA.

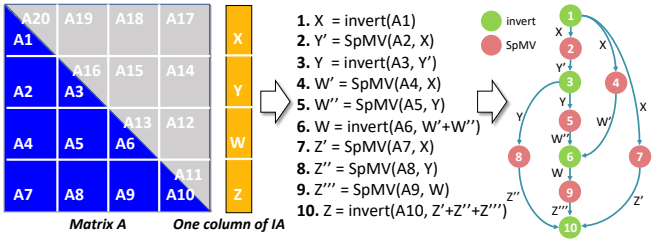

Fig. 3. **Key insight:** Breaking the main invert on matrix A into parallizable SpMV and small fast invert and representing their dependencies as a graph.

$A^{-1} = U^{-1}L^{-1}$. The decomposition itself is more straightforward and parallelizable compared to the invert algorithm. Therefore, this paper focuses on the invert algorithm to find the performance bottleneck. Figure 1 lists a simple pseudo-code for LU invert, which consists of an outer loop for traversing W columns of A. The outer loop comprises two inner loops for traversing A from top to bottom (i.e., phase 1) and bottom to top (i.e., phase 2), respectively. At each of these inner loops, an entire column of invert matrix, IA, and a triangular of A are read, and simultaneously the *same* column of IA is updated. As a results, the iterations of i are dependent. Therefore, we must process the rows of A, *sequentially*. In other words, *reading and updating the same column of IA potentially creates a data dependency that prevents further parallelism and limits fully utilization of memory bandwidth.*

In an optimized FPGA implementation, we can parallelize the iterations of k by a factor of n. To this end, not only do we unroll the iterations of k by a factor of n, but also we interleave the elements of A into n partitions of BRAM, so that we can access them in parallel. We pipeline these chunks of size n. If each multiplication takes 2 cycles and the subtraction and write operation take 1 cycle each, we can have a pipeline as shown in Figure 2a. Such an optimized implementation is our baseline implementation, in which the number of cycles to process a rectangle of A depends on width of matrix (W) and the parallelism factor (n). Since there is no pipeline or parallelism *across the rows*, the total time to process W rows is: $4{+}6{+}\ldots{+}2(W/n{+}1)$. Figure 2b shows the invert latency for various W and n. As illustrated, when W doubles, cycles increases approximately $4\times$.

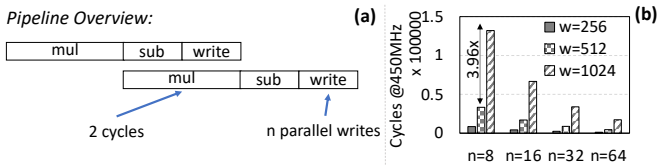

Fig. 2. **Baseline optimized implementation: (a)** pipelining within each row, *but not across the rows*; and **(b)** invert latency when W and n vary.

III. MAIA

Key Insight – To solve the key challenge and improve the level of parallelism in an FPGA, we have to enable parallelism across the rows or at least partial rows. To to do so, we can break down matrix A into blocks as shown in Figure 3 left. Once we are done with processing A1, all rows in A2 can be processed in parallel. However, after that, we still need to sequentially process all rows of A3. As a result, even if we increase parallelism in processing some blocks of A, the overall latency would be still limited by the same number of sequential operations. Only if could we accelerate the sequential blocks (e.g., A3) will this challenge be resolved. Thanks to blocking, the sequential operations became small, which allow us to implement them as a small optimized logic with a fixed small latency. In other word, our key insight is to *implement the mathematical dependencies as gate-level dependencies*.

Partitioning – To implement our approach, after dividing matrix A into blocks, we apply sparse matrix vector multiplication (SpMV) on the squares and invert on the triangles. In other words, the main large invert operation will be broken down into several SpMVs and very small invert operations as shown in Figure 3 middle. Although we increase the level of parallelism and also accelerate the sequential operations, there would be still dependencies in between the SpMV and invert operations on blocks of A as shown in Figure 3 right. As long as the dependencies are not violated, the small SpMVs can be combined together to create a larger SpMV (e.g., nodes 4, 5, 7, and 8 as shown in Figure 4). Similarly, recursively, two small invert nodes and one small SpMV can be combined

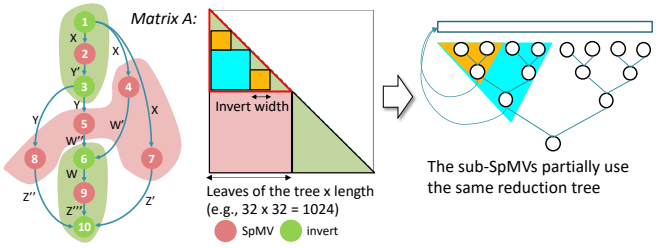

Fig. 4. **Mapping the graph to hardware:** partitioning the dependency graph and mapping it into a reduction tree.

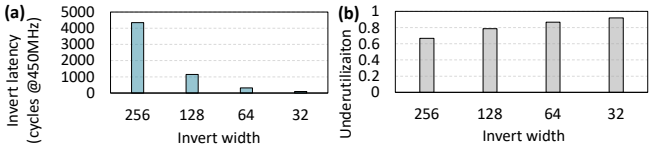

Fig. 5. **Granularity of Partitioning:** the impact on **(a)** latency and **(b)** underutilization of tree.

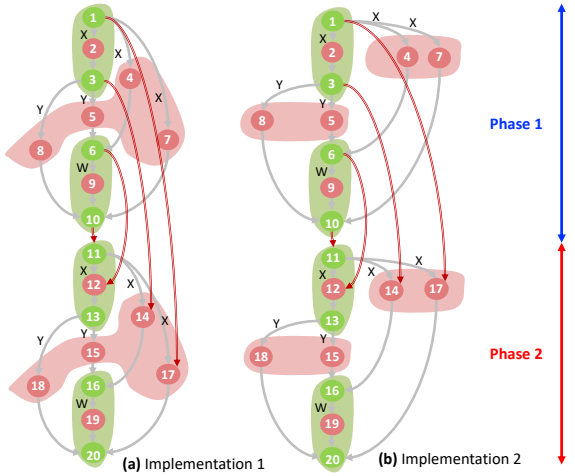

Fig. 6. **Partitioning schemes:** the operation unit at hardware are **(a)** separate SpMV and invert (Imp1), **(b)** invert + SpMV + invert (Imp2).

to create a larger invert (e.g., nodes 1, 2, and 3 as shown in Figure 4). We call the process of breaking down the graph into variable size kernel partitioning. Partitioning the tree into right size SpMV and invert kernels is done by the software of Maia. To run a partitioned graph, the hardware of Maia includes (i) a fixed-sized dot product followed by a balanced reduction tree and in which the largest SpMV fits; (ii) a fixed-sized small invert engine in which one green invert node of Figure 3 fits. The other sizes of SpMV would partially use the same tree, and the other sizes of invert recursively use the invert engine and the partial tree. An example of mapping the graph to the tree is shown in Figure 4. In the following we explain the granularity of partitioning (that defines the invert width) and the shape of partitions both of which impact performance.

The size of invert kernel creates a trade-off between latency and the underutilization of tree: while a smaller invert results in a lower latency, it also results in more underutilization of the tree. Figure 5a and Figure 5b respectively show the invert latency and the underutiliziation of the tree, based on which we

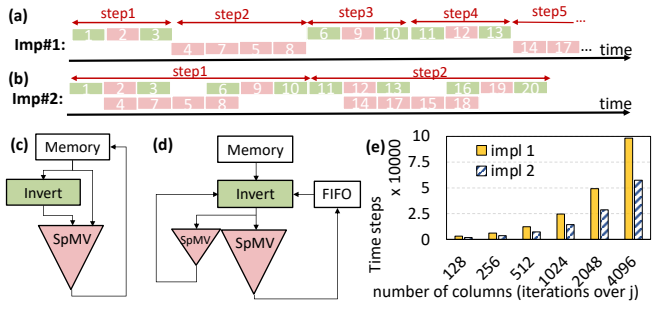

Fig. 7. **Comparing two partitioning schemes:** timeline of **(a)** implementation 1 and **(b)** implementation 2; block diagrams of **(c)** implementation 1 and **(d)** implementation 2; and **(e)** the overall impact on execution time.

choose the size of 128 as the invert width. The simple scheme of partitioning shown in Figure 4 is not necessarily the best scheme. The reason is that if we extend that partitioning to the phase 2 of the algorithm as shown in Figure 6a, one large chain of dependencies will occur (nodes 6 to 13). Therefore, we modify the partitioning into the implementation shown in Figure 6b in which a unit of operation at hardware is *invert + SpMV + invert* rather than separate SpMV and invert kernels. As a result, as the timelines shown in Figure 7a and Figure 7b clarify, one large SpMV can be overlapped with the execution of two inverts, which reduces the overall time to process phase 1 and 2. Since in implementation 2, we need to perform parallel SpMVs (e.g., blocks 2 and 4, or blocks 12 and 14), in hardware we include two SpMV engines, one small and one large as shown in Figure 7d. In both implementations (Figure 7c and Figure 7d), phase 1 writes into a first-in-last-out buffer (i.e., stack), from which phase 2 reads the intermediate results. Figure 7e illustrates the overall impact of partitioning scheme when the size of matrix (i.e., the number of columns) increases. Implementation 2 achieves $\sim 1.7\times$ for matrix size of 4096. with only 3.4% increase in resource utilization – which is small compared to the total 42.8% resource utilization of implementation 1).

SpMV – In addition to partitioning, the implementation of SpMV engine also plays a key role in the performance of matrix inversion. In a baseline simple implementation, to use a dot product and balanced adder tree for executing SpMV, we can stream the sparse matrix from the memory (where it is stored in a compressed format) through the tree. In such an implementation that we call SpMV1, the distribution of non-zero values in the original sparse matrix would determine the load of tree, which would not be necessarily a balanced load across the levels of the tree. In an alternative implementation that we call SpMV2, we deploy a small load-balance unit in the hardware. As Figure 8a shows, for a five-layer tree with 32 leaves, the load of the layers and in turn their processing time increase more dramatically in SpMV1 compared to SpMV2 as we move from leaves (i.e., L0) to the root (i.e., L4). The load imbalance of SpMV1 also causes non-deterministic total processing time. As Figure 8b illustrates, the maximum latency of SpMV1 can be $\sim 13\times$ the minimum latency for various distributions of the same degree of sparsity. Therefore, in the best case, SpMV1 is $3.3\times$ faster than SpMV2 and in the worst case, it is $4.1\times$ slower than SpMV2. The load distribution in a tree also impacts the achievable throughput at a given BRAM utilization since the required BRAM to buffer the inputs at each level of the tree varies. Based on Figure 8c, to achieve a close-to-peak throughput, SpMV1 requires buffering 128×128 rather than 64×64 for SpMV2. In other words, by utilizing the same BRAM for buffering 64×64 inputs, SpMV2 achieves $4.3\times$ higher throughput.

IV. Results

Experimental Setup – We describe Maia in C++ and use it as input to Vivado high-level synthesis (HLS) tool to generate Verilog. After simulations in Vivado_HLS, we

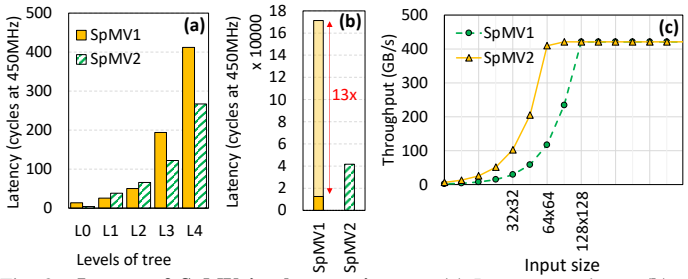

Fig. 8. **Impact of SpMV implementation on: (a)** Latency per layer, **(b)** Total latency, and **(c)** Throughput at a given input size to be bufferred. Max throughput is 421.67 GB/s (13.15 GB/s per pseudo channel of HMB2).

use Vivado to synthesise and implement our hardware on an UltraScale+ XCU280 Xilinx FPGA targeting the Alveo-U280 data-center boards with a High Bandwidth Memory (HBM2). The clock frequency is set to 450 MHz. We verify the functionality of Maia using synthetic data as a testbench for C/RTL co-simulation. The inputs to our hardware platform are sparse matrices from SuitSparse [22] listed in Table I with various densities ($Density = 1 - Sparsity$). In addition to the analysis in the previous sections, here we compare the execution time, resource utilization, and power consumption of two configurations: Imp1+SpMV1 and Imp2+SpMV2.

Speedup – First, in Figure 9, we compare the execution time of two optimized implementations of Maia (i.e., Imp1+SpMV1 and Imp2+SpMV2) normalized to the baseline optimized implementation of matrix inversion in which parallelism and pipelining is fully implemented withing the rows but not across them (see Figure 2). As Figure 9 shows, as expected, because of (i) more overlapped tasks (i.e., overlapping invert and SpMV kernels) at hardware and (ii) better load balanced among the level of reduction tree, Imp2+SpMV2 always work more quickly than Imp1+SpMV1 – on average, $1.5\times$ faster. Depending on the pattern of sparsity which in turn defines the opportunity for cross-row parallelism, pipelining, cuncurrency, and load-balancing, Imp1+SpMV1 performs matrix inversion $2.2\times$ and up to $2.9\times$ faster than baseline where Imp2+SpMV2 provides a speedup between $2.8\times$ to $4.8\times$ over the baseline.

Resource Utilization – The greater speedup of Imp2+SpMV2 comes at the cost of more recourse utilization. As Figure 10 illustrates, compared to Imp1+SpMV1, Imp2+SpMV2 utilizes more lookup tables (LUT), flip flops (FF), and BRAM blocks because of the following reasons: (i) although each compute unit is simpler in SpMV2, it uses some additional resources to arrange data before sending them to the compute units and balance the load; (ii) SpMV2 must also coordinate a synchronization across the compute units since load balance could cause some synchronization in

TABLE I
MATRICES FROM SUITESPARSE [22]

Sparse Matrix Name	Dimension (M)	NNZ (M)	Density (%)
2cubes_sphere	0.101	1.647	0.01615
Freescale2	2.9	14.3	0.00017
N_reasctome	0.016	0.043	0.01680
dwt_918	0.000918	0.0073	0.86624
hcircuit	0.1	0.51	0.00510
hugebubbles-00000	18.3	54.9	0.00002
rajat31	4.6	20.3	0.0001
thermomech_dK	0.2	2.8	0.007

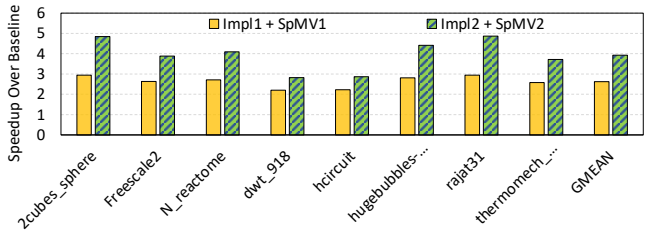

Fig. 9. **Speedup over the baseline:** comparing the normalized execution time of two configuration when $n = 16$, $w = 1024$, $width = 128$.

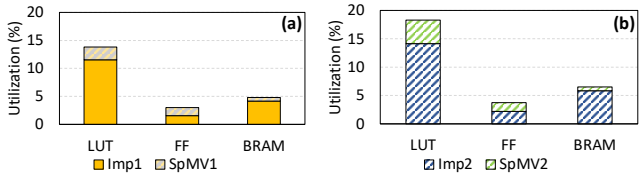

Fig. 10. **Resource Utilization:** comparing the FPGA resource utilization of **(a)** Imp1+SpMV1 and **(b)** Imp2+SpMV2.

computations; (iii) As the breakdown in Figure 10 shows, if we fix the size of input buffers for SpMV1 and SpMV2 (we set it to 64×64), the Imp2+SpMV2 would still utilize higher BRAM because of Imp2. As discussed in Figure 8, we can devote larger buffers to SpMV2 to increase its throughput.

Power Consumption – As Figure 11a illustrates, Imp2+SpMV2 consumes $1.8\times$ less dynamic power compared to Imp1+SpMV1, mainly because of signals and logic used for more complex compute units in SpMV1. S As Figure 11b shows, the static power consumption of both implementations are almost the same. Therefore, in total, Imp2+SpMV2 consumes only $1.08\times$ less power compared to Imp1+SpMV1.

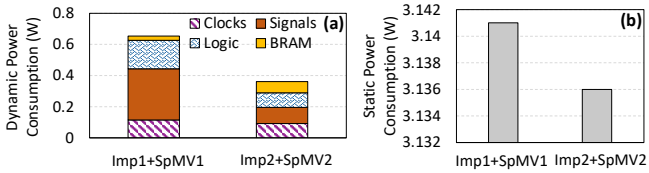

Fig. 11. **Power Consumption: (a)** the breakdown of dynamic and the **(b)** static power consumption of Imp1+SpMV1 and Imp2+SpMV2.

V. CONCLUSIONS

This paper proposed Maia, a fast and efficient FPGA implementation for matrix inversion, an important operation in widespread domains of scientific computing applications. Maia is the first solution that resolves data dependencies to enable near-memory processing for the memory-intensive matrix inversion on a high-end FPGA platform consisting of HBM2 memory to utilize its peak memory bandwidth.

ACKNOWLEDGEMENTS

We gratefully acknowledge the support of Booz Allen Hamilton Inc. and the Laboratory for Physical Sciences (LPS), and thank Dr. Eric Cheng from LPS who provided insight and expertise that greatly assisted this research.

REFERENCES

[1] P. Ezzatti, E. S. Quintana-Orti, and A. Remon, "High performance matrix inversion on a multi-core platform with several gpus," in *2011 19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing*. IEEE, 2011, pp. 87–93.

[2] G. Zhou, Y. Feng, R. Bo, and T. Zhang, "Gpu-accelerated sparse matrices parallel inversion algorithm for large-scale power systems," *International Journal of Electrical Power & Energy Systems*, vol. 111, pp. 34–43, 2019.

[3] K. Yang, Y. Li, and Y. Xia, "A parallel method for matrix inversion based on gauss-jordan algorithm," *Journal of Computational Information Systems*, vol. 9, no. 14, pp. 5561–5567, 2013.

[4] J. Liu, Y. Liang, and N. Ansari, "Spark-based large-scale matrix inversion for big data processing," *IEEE Access*, vol. 4, pp. 2166–2176, 2016.

[5] C. Misra, S. Haldar, S. Bhattacharya, and S. K. Ghosh, "Spin: A fast and scalable matrix inversion method in apache spark," in *Proceedings of the 19th International Conference on Distributed Computing and Networking*, 2018, pp. 1–10.

[6] R. Gower, F. Hanzely, P. Richtárik, and S. U. Stich, "Accelerated stochastic matrix inversion: general theory and speeding up bfgs rules for faster second-order optimization," in *Advances in Neural Information Processing Systems*, 2018, pp. 1619–1629.

[7] A. K. Jain, H. Omidian, H. Fraisse, M. Benipal, L. Liu, and D. Gaitonde, "A domain-specific architecture for accelerating sparse matrix vector multiplication on fpgas," in *2020 30th International conference on field-programmable logic and applications (FPL)*. IEEE, 2020, pp. 127–132.

[8] B. Asgari, R. Hadidi, and H. Kim, "Ascella: Accelerating sparse computation by enabling stream accesses to memory," 2020.

[9] B. Asgari, R. Hadidi, J. Dierberger, C. Steinichen, A. Marfatia, and H. Kim, "Copernicus: Characterizing the performance implications of compression formats used in sparse workloads," in *2021 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2021, pp. 1–12.

[10] B. B. Girish Sharma, Abhishek Agarwala, "A fast parallel gauss jordan algorithm for matrix inversion using cuda," *Elsevier Computers and Structures*, vol. 128, pp. 31–37, 2013.

[11] B. Holanda, R. Pimentel, J. Barbosa, R. Camarotti, A. Silva-Filho, L. Joao, V. Souza, J. Ferraz, and M. Lima, "An fpga-based accelerator to speed-up matrix multiplication of floating point operations," in *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. IEEE, 2011, pp. 306–309.

[12] M. Karkooti, J. R. Cavallaro, and C. Dick, "Fpga implementation of matrix inversion using qrd-rls algorithm," in *Asilomar Conference on Signals, Systems, and Computers*, 2005.

[13] J. Arias-García, R. P. Jacobi, C. H. Llanos, and M. Ayala-Rincón, "A suitable fpga implementation of floating-point matrix inversion based on gauss-jordan elimination," in *2011 vii southern conference on programmable logic (SPL)*. IEEE, 2011, pp. 263–268.

[14] A. Irturk, B. Benson, S. Mirzaei, and R. Kastner, "An fpga design space exploration tool for matrix inversion architectures," in *2008 Symposium on Application Specific Processors*. IEEE, 2008, pp. 42–47.

[15] Y. Xu, D. Li, Y. Xi, J. Lan, and T. Jiang, "An improved predictive controller on the fpga by hardware matrix inversion," *IEEE Transactions on Industrial Electronics*, vol. 65, no. 9, pp. 7395–7405, 2018.

[16] B. Asgari, R. Hadidi, N. S. Ghaleshahi, and H. Kim, "Pisces: Power-aware implementation of slam by customizing efficient sparse algebra," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.

[17] Q. Sun, M. Zhao, and X. Zhang, "Implementation matrix inversion of lu algorithm with systolic array," *Microelectronics and Computers*, vol. 24, pp. 138–141, 2007.

[18] N. Melab, E.-G. Talbi, and S. Petiton, "A parallel adaptive version of the block-based gauss-jordan algorithm," in *Proceedings 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing. IPPS/SPDP 1999*. IEEE, 1999, pp. 350–354.

[19] L. M. Aouad and S. G. Petiton, "Parallel basic matrix algebra on the grid'5000 large scale distributed platform," in *2006 IEEE International Conference on Cluster Computing*. IEEE, 2006, pp. 1–8.

[20] L. Shang, S. Petiton, and M. Hugues, "A new parallel paradigm for block-based gauss-jordan algorithm," in *2009 Eighth International Conference on Grid and Cooperative Computing*. IEEE, 2009, pp. 193–200.

[21] Y. He, Y. Song, G. Du, and D. Zhang, "Research of matrix inversion acceleration method," in *2009 International Conference on Computational Intelligence and Software Engineering*. IEEE, 2009, pp. 1–4.

[22] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, p. 1, 2011.