Optimized GPU-Based Matrix Inversion

Through The Use of Thread-Data Remapping

Samuel J. Monson

June 8, 2024

Abstract

This research paper focuses on the development and analysis of a matrix inversion program capable of efficiently utilizing GPU hardware though the use of Thread-Data Remapping. We showcase existing published inverse algorithms and implementations while highlighting the importance of considering matrix characteristics and computational optimizations. We introduce the idea of Thread-Data Remapping into the fields of matrix inversion as a method of optimizing for sparse matrix inversion while retaining the ability to invert all kinds of matrices. Our final results demonstrate an implementation that is able to outperform traditional implementations when applied to sparse matrices.

1 Introduction

Matrix inversion is a fundamental component of Linear Algebra that has wide practical application in nearly all math-adjacent fields. Most notably, it has proven to be the best algebraic method for solving linear equations on a computer [1]. This has made matrix inversion a critical component of everything from Computer Graphics to Machine Learning [2].

However, a significant drawback of matrix inversion lies in its computationally intensive nature. Traditional methods such as Gauss-Jordan [3] and LU-Decomposition [1] possess a multiplicative asymptotic time complexity of $O(n^3)$ for any given $n \times n$ matrix. In 1969 Strassen [4] introduced a new method that brought the complexity down to $O(n^{2.808})$. Following this discovery, various groups competed to improve on Strassen's results culminating in the Coppersmith and Winograd method [5] which achieved a time complexity of $O(n^{2.376})$. Theoretically, the best sequential cost achievable for matrix inversion is $O(n^2)$ [6]; however, no general algorithm has been found that achieves this cost.

While no general $O(n^2)$ inversion method has been found, there exist various methods that optimize for specific matrix types. Triangular matrices can be inverted using Gaussian Elimination in $O(n^2)$ time [1]. The inverse of an orthogonal matrix is its transpose [2], thus the time complexity to invert an orthogonal matrix corresponds to the number of swap operations required to flip rows and columns, specifically $\frac{n^2}{2} - n$. Alternatively, if the matrix is simply read from memory in the new order, the time complexity is O(1). Cholesky Decomposition [1] improves on LU-Decomposition for symmetric, positive-definite matrices and has a time complexity of $O\left(\frac{3}{4}n^3 + \frac{3}{2}n^2\right)$. QR-decomposition [1] is slower then LU in most cases excluding where the inverses of multiple similar matrices are needed. For example, when we have a series of matrices where $A_{i+1} = A_i + B$ intermediate steps of the QR process of A_i can be reused for A_{i+1}.

The Graphics Processing Unit (GPU) is a specialized co-processor originally created for the task of translating data representations into computer graphics. As modern display technology consists of a grid of independent points, GPUs have evolved to excel at embarrassingly parallel workloads. Due to this specialization, GPUs have found significant success outside of their originally designed function, particularly in scientific fields where extensive independent computation is crucial [7]. Since a grid of points is essentially a physical example of a matrix, GPUs are especially well-suited for operations involving matrices.

Due to the prolific use of matrix inversion in computing, an algorithm that improves on the time-complexity of existing inverse methods is of high demand. Thus, the goal of this paper is to outline a matrix inversion implementation that outperforms existing implementations in some or all scenarios. To achieve this goal, we built upon existing methods of paralleling matrix inversion and introduced a novel approach utilizing Thread-Data Remapping.

2 Related Work

2.1 Strassen Matrix Inversion

In 1969 **Strassen [4]** published a simple 3 page paper that, for the first time, showed algorithms for matrix multiplication and inversion that were sub-$O(n^3)$. Though not fully recognized until later, this was a huge milestone for computing. While the matrix multiplication algorithm is an achievement with arguably greater impact we will focus on inversion. Strassen's matrix inversion operates by splitting the matrix into quadrants and running recursively on two of them. The algorithm is a prime example of a divide and conquer approach and is thus well suited for a parallel implementation.

However, there are some problems that prevent Strassen matrix inversion from being the ideal algorithm. Due to the recursive inversion only taking place on two of the quadrants, certain matrices can end up with recursive quadrants that are very close to singular, leading to high numerical error. A solution is to pivot the matrix at each level to whichever of the top-most quadrants has a higher determinant [8], but this adds extra complexity to the algorithm. Another issue is that each level of recursion contains many intermittent matrix multiplications. All of these calculations must be stored, which more then doubles the space requirements of a typical inversion.

2.2 Parallel Gauss-Jordan

While Gauss-Jordan is no longer the most efficient matrix inversion method, **Sharma et al [9]** were able to redesign it into a highly parallel GPU based Gauss-Jordan algorithm. Their implementation showed impressive results; achieving $O(n)$ latency, though with some major caveats.

Sharma et al hits many of the limitations of GPU hardware. In order to avoid the extra I/O penalty associated with global GPU memory, the algorithm must be able to store one row and one column of the matrix in shared memory. Additionally, the GPU algorithm improves on the parallel $O(n^3)$ by doing n^2 of the work in parallel, thus if the GPU does not have at least n^2 parallel threads, than the latency will increase exponentially.

2.3 In-Place Approach

Typically, the Gauss-Jordan algorithm requires appending a $n \times n$ unit matrix to the original matrix. However, in 2013 **DasGupta [10]** introduced a modified Gauss-Jordan algorithm that handles the inversion in-place. While this algorithm improves the space efficiency of Gauss-Jordan, it retains the time complexity of $O(n^3)$. In 2023, **Xuebin et al [11]** created a parallel modification of DasGupta's algorithm that is optimized for inverting many small matrices at a time on GPU. This algorithm is bound by similar limitations to the one in Sharma et al [9] (see 2.2). Assuming $n \times \textit{number of matrices}$ is small enough to fit into the total number of parallel threads, then the algorithm runs in $O(n^2)$ time.

2.4 GPU Thread-Data Remapping

Due to the nature of GPU architecture, threads within the same warp are not able to execute different paths in parallel. This limits the performance of workloads that contain conditional branching or uneven allocation of work as branches are serialized. Largely divergent workloads can try to avoid this overhead by periodically reshuffling data to reduce the divergence inside warps; this technique is called Thread-Data Remapping (TDR). The most common form of TDR involves stopping all work at set intervals and performing synchronization. This approach is less than ideal since full workload synchronization requires the CPU to step in and handle workload discrepancies between runs. Communication between the CPU and GPU is expensive and should be avoided if possible.

A better approach, introduced by **Cuneo and Bailey [12]**, handles TDR entirely on-GPU by implementing a work scheduling mechanism that is reminiscent of the promise and future concurrency model. While not the first on-GPU TDR, Cuneo and Bailey's method is the first to allow remapping across blocks without synchronization. The results of their research are available for other to use in Harmonize

[13] a CUDA® C++ and Python™ library developed by Cuneo.

3 Background

3.1 Gauss-Jordan Elimination

In linear algebra we can utilize matrix multiplication to transform a matrix row-by-row. For instance the multiplication

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}\begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} c & d \\ a & b \end{bmatrix} \tag{3.1}$$

swaps the rows of the right hand side matrix. Utilizing this technique we can define similar transformation matrices for scaling rows and adding multiples of one row to another (hence shifting a row by a multiple of another).

1. Swap one row with another (See 3.1)
2. Scale a row

$$\begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} 2a & 2b \\ c & d \end{bmatrix} \tag{3.2}$$

3. Shift a row by a multiple of another

$$\begin{bmatrix} 1 & 0 \\ 3 & 1 \end{bmatrix}\begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} a & b \\ c+3a & d+3b \end{bmatrix} \tag{3.3}$$

The process of Gauss-Jordan elimination utilizes these transformations to convert matrices to a canonical form where leading entries are 1 and 0s are present both above and below each leading entry. This form is called the reduced row-echelon form. If the matrix is fully reducible, then for a matrix with n rows, the first n columns form an identity matrix of size n. Thus, for an $n \times n$ matrix M, the given result of applying transformations T_1 to T_i is the identity matrix $\mathbf{I}_n$,

$$T_n \cdots T_2 T_1 M = \mathbf{I}_n \tag{3.4}$$

Given that $M^{-1}M = \mathbf{I}_n = MM^{-1}$ we can show that,

$$\begin{aligned} T_n \cdots T_2 T_1 M &= M^{-1} M \\ T_n \cdots T_2 T_1 M M^{-1} &= M^{-1} M M^{-1} \\ T_n \cdots T_2 T_1 \mathbf{I}_n &= M^{-1} \end{aligned} \tag{3.5}$$

Therefore applying the same T_1 though T_n operations to the identity matrix will result in the inverse of our matrix M. Utilizing this relationship we can invert a square matrix by performing Gauss-Jordan on the matrix $M|\mathbf{I}$, M concatenated with an identity matrix. The resulting matrix after Gauss-Jordan will be $\mathbf{I}|M^{-1}$.

Rather than performing the full matrix multiplication for every Gauss-Jordan operation we can merely apply the arithmetic directly to the row, given that we represent the result of each transformation as an algebraic operation on a given row. For example, the transformation of doubling row 3 in a matrix can be written as $R_3 \leftarrow 2 \times R_3$ and thus it is sufficient to multiply each element of row 3 by 2.

While the combination matrix of all transformation $T_1 T_2 \cdots T_n$ is unique, the individual operations are not. For example

$$T_{R_1 \leftarrow 2R_1} T_{swap(R_1,R_2)} = T_{swap(R_1,R_2)} T_{R_2 \leftarrow 2R_2} \tag{3.6}$$

Thus, there are many methods of deriving a combination of operations. For this research we focus on the algorithm utilized by Sharma et al [9] given in (Algorithm 1).

3.1.1 Parallel Gauss-Jordan

The parallel Gauss-Jordan method, introduced by Sharma et al. [9], leverages the algorithm outlined in (Algorithm 1) by executing each set of associative row operations concurrently. To determine the sets of operations that exhibit associativity and can thus be executed in parallel, we will assume everything is associative and disprove individual cases. The first evident disproof arises with the swap operation, which, as demonstrated in (Eqn 3.6), lacks associativity with other operations on the affected rows. Thus in our algorithm the swap, denoted *step 1*, must be taken in serial with its surrounding operations. This further causes the enclosing loop at line 1 to be serial; since each iteration introduces a new swap operation.

For our remaining operations, *step 2* performs a scale on the ith row, while the loop, *step 3*, performs a shift on every other row. Scaling is algebraically equivalent to scalar multiplication on a vector and thus scaling operations are associative with each other following the associativity of scalar multiplication. Shifting operations consist of a multiplication followed by an addition; since multiplication and addition together are not strictly associative it seems that shift operations are non-associative.

However, it is also possible to perform operations in parallel if they are linearly independent. Each operation has a strict set of rows that it operates on so we can consider an operation linearly independent from operations on other rows. The shift operation technically operates on two rows but only transforms one of those rows. Thus shift operations can generally be considered linearly independent with other shift operations that utilize the same multiple row, but not with operations that modify the row. Therefore, *step 2* and *step 3* necessitate serial execution due to their shared operation on the row, whereas all *step 3.1* actions can be carried out in parallel, given their linear independence.

So far we have been assuming row operations are atomic in that they perform operations on whole rows simultaneously. However, large enough matrices will necessitate that we split row operations into two or more step in order to process the entire row. This presents a problem in both *step 2* and *step 3.1* because we assume that the ith element of the targeted row has not been modified during the operation. Therefore we must either ensure that the ith element is modified last or that we store the ith element elsewhere before performing the operation.

3.1.2 In-Place Gauss-Jordan

In the Gauss-Jordan method introduced above we operate on the matrix $M|\mathbf{I}$ and utilize (Algorithm 1) to transform it to $\mathbf{I}|M^{-1}$. However, the only resulting component of the matrix we care about is M. Further, after each iteration of the outer loop in our algorithm, half of our augmented matrix will be columns of the identity matrix. For example, given the matrix

$$A = \left[\begin{array}{ccc|ccc} 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 2 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 \end{array}\right] \tag{3.7}$$

after 2 iterations of the outer loop we will end up with

$$A = \left[\begin{array}{ccc|ccc} 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1/2 & 0 & 1/2 & 0 \\ 0 & 0 & -1/2 & -1 & -1/2 & 1 \end{array}\right] \tag{3.8}$$

The first 2 columns have already been inverted by the algorithm and the last column has been untouched, thus those columns put together form the identity matrix. Therefore, at every step we are wasting computational time and space processing columns of the identity matrix.

The in-place method introduced by DasGupta [10] saves us this extra computation by storing the only the column of the inverse necessary to complete each iteration. To perform the in-place method we start with the matrix M and perform the updated algorithm shown in (Algorithm 2).

At the beginning of each iteration i, we add an additional *step 0* where we store the ith column of

Algorithm 1: Gauss-Jordan Elimination

```
Input: An augmented matrix $M$ that has $n$ rows
foreach row $R_i$ in $M$ do
    // Step 1: Swap our current row for one with a non-zero $i$th element.
    Find $R_k$ where $R_{ki} \neq 0$
    swap($R_i, R_k$)
    // Step 2: Divide our current row by its $i$th element.
    $R_i \leftarrow R_i / R_{ii}$
    // Step 3: From every other row
    foreach row $R_j$ in $M$ where $j \neq i$ do
        // Step 3.1: subtract the $R_{ji}$ multiple of the $i$th row.
        $R_j \leftarrow R_j - R_{ji} \times R_i$
    end
end
```

the matrix and replace it with the ith column of the identity. In *step 1* we additionally swap the stored i and k elements. Then, in *step 2* we divide by the stored ith element rather than its current value. And finally in *step 3.1* we multiply by the jth stored element rather than the ith element of the jth row.

4 Main Results

4.1 Deliverables

During the research, we developed three key implementations of matrix inversion. These implementations demonstrated our understanding of the problem and validated our research results. The first implementation, called `cpu-inverse`, runs Algorithm 2 in a single serial CPU thread. This program verified the validity of the inverse algorithm and helped in troubleshooting race conditions in the parallel implementations. The next program, `inverse`, applies Algorithm 2 on the GPU using standard CPU-based kernel synchronization. This implementation is comparable to the work of Sharma et al. [9], with the optimizations from DasGupta [10]. The final program, `tdr-inverse`, summarizes our research results and uses the Harmonize [13] library to optimize matrix inversion through Thread-Data Remapping.

4.2 Performance Analysis

To evaluate the performance of our matrix inversion, we conducted tests using the standard on-GPU `inverse` and TDR-based `tdr-inverse` programs on a suite of matrices, comparing the runtime of each method. The tests were carried out on a NVIDIA® A30 Tensor Core GPU (Driver 545.23.08) paired with an Intel® Xeon® Silver 4316 CPU. The timing for each matrix inversion was initiated after memory allocation and loading the matrix into the GPU. Matrices were stored and inverted as IEEE 754 64-bit floating-point numbers, making them susceptible to floating-point inaccuracy. To calculate the error of each inverse, we used the inverse method from the SciPy Python™ library [14] as a baseline and computed an element-wise mean absolute error.

Two sets of matrices were tested. Figures 1 and 2 correspond to a collection of randomly generated whole number matrices ranging in size from 3×3 to 4096×4096. Figures 3, 5, and 4 utilized sparse matrices from the University of Florida SuiteSparse Matrix Collection [15], ranging from 1083×1083 to 19366×19366 and containing a significant number of zero elements.

Figure 1 illustrates that in the general case represented by our random matrices, `tdr-inverse` performs consistently worse than the standard on-

Algorithm 2: In-Place Gauss-Jordan Elimination

```
Input: An augmented matrix $M$ that has $n$ rows
foreach row $R_i$ in $M$ do
    // Step 0: Store the $i$th column in $C$.
    foreach row $R_m$ in $M$ do
        $C_m \leftarrow R_{mi}$
        $R_{mi} \leftarrow \mathbf{I}_{mi}$
    end
    // Step 1: Swap our current row for one with a non-zero $i$th element.
    Find $R_k$ where $R_{ki} \neq 0$
    swap($R_i, R_k$)
    swap($C_i, C_k$)
    // Step 2: Divide our current row by its $i$th element.
    $R_i \leftarrow R_i / C_i$
    // Step 3: From every other row
    foreach row $R_j$ in $M$ where $j \neq i$ do
        // Step 3.1: subtract the $R_{ji}$ multiple of the $i$th row.
        $R_j \leftarrow R_j - C_j \times R_i$
    end
end
```

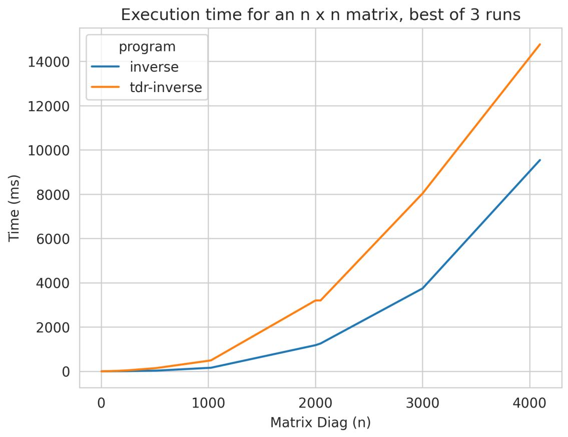

Figure 1: Random $n \times n$ matrices

GPU `inverse`; however, the trend of each line indicates they belong to the same polynomial family. This difference is likely due to the overhead incurred by Thread-Data Remapping on each thread call. We make exponentially more TDR thread calls in `tdr-inverse` than kernel launches in the standard `inverse`; the workload assigned to each thread is so minimal that we do not observe any benefit from the flexibility provided by TDR.

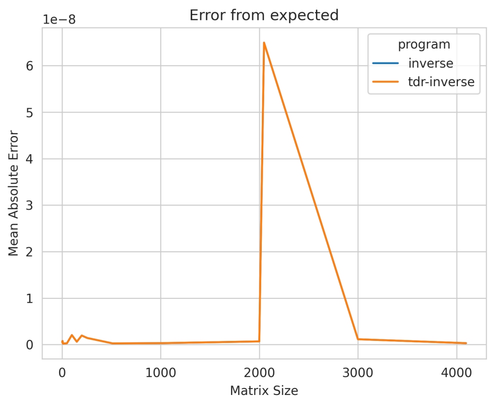

Figure 2: Random $n \times n$ matrices error

The error of our random matrix set (Figure 2) remains within an acceptable range below 10^{-7}, attributable to precision loss and inconsequential to the research objectives. Moreover, both `inverse` and `tdr-inverse` exhibit identical error rates, confirming the accuracy of `tdr-inverse`. Although there is a peculiar peak for the matrix $n = 2048$, it still falls within the low error range observed in other results.

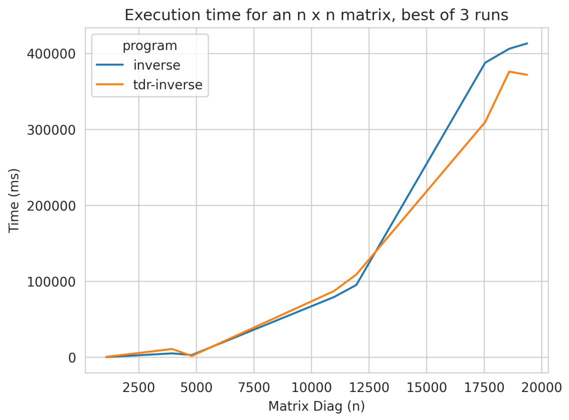

Figure 3: Sparse $n \times n$ matrices

Our sparse matrix results (Figure 3) present a more favorable outcome for `tdr-inverse`. As the matrix size increases, `tdr-inverse` matches and eventually surpasses the performance of the standard `inverse`. The initial point where `tdr-inverse` shows lower latency than `inverse` is at $n = 4800$; however, `tdr-inverse` loses this advantage at $n = 10974$. By examining the sparsity of our matrices (Figure 4), we observe that the overall sparsity of $n = 10974$ is lower than that of its neighboring matrices, explaining this discrepancy.

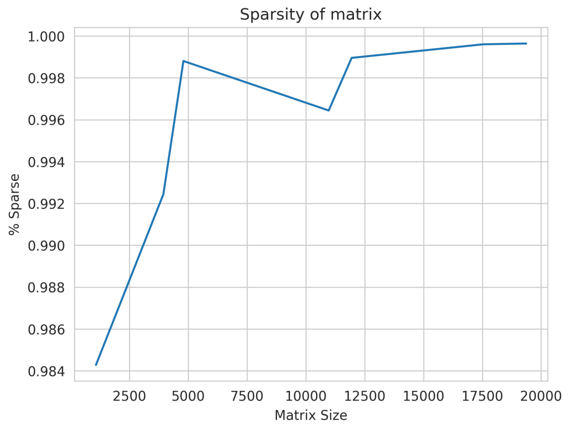

Figure 4: Sparisity of matrices

The error for our sparse matrix set (Figure 5) follows a similar pattern follows a similar pattern to the random matrix results and is low enough to not raise concerns.

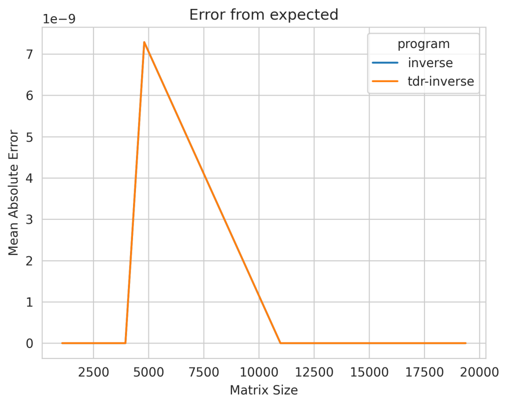

Figure 5: Sparse $n \times n$ matrices error

5 Conclusion

The results of our study revealed interesting performance trends across both randomly generated and sparse matrices. While `tdr-inverse` exhibited higher runtime compared to the standard on-GPU `inverse` for random matrices, it showcased promising performance improvements for larger sparse matrices. Both implementations maintained low error rates, validating the accuracy and reliability of the inversion results.

References

[1] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, “Solution of linear algebraic equations,” in *Numerical Recipes: The Art of Scientific Computing*, 3rd ed. 32 Avenue of the Americas, New York, NY 10013-2473, USA: Cambridge University Press, 2007, ch. 2, pp. 37–109, ISBN: 978-0-521-88068-8 (cit. on p. 1).

[2] H. Anton and C. Rosses, “Applications of linear algebra,” in *Elementary Linear Algebra*, 11th ed. 111 River Street, Hoboken, NJ 07030-5774, USA: John Wiley & Sons, Inc., 2014, ch. 10, pp. 527–713, ISBN: 978-1-118-43441-3 (cit. on p. 1).

[3] S. C. Althoen and R. McLaughlin, “Gauss-jordan reduction: A brief history,” *American Mathematical Monthly*, vol. 94, no. 2, pp. 130–142, Feb. 1987, ISSN: 0002-9890. DOI: 10.2307/2322413. [Online]. Available: https://www.jstor.org/stable/2322413 (cit. on p. 1).

[4] V. Strassen, “Gaussian elimination is not optimal,” *Numerische Mathematik*, vol. 13, pp. 354–356, Aug. 1969. DOI: 10.1007/BF02165411 (cit. on pp. 1, 2).

[5] D. Coppersmith and S. Winograd, “On the asymptotic complexity of matrix multiplication,” in *22nd Annual Symposium on Foundations of Computer Science (sfcs 1981)*, 1981, pp. 82–90. DOI: 10.1109/SFCS.1981.27 (cit. on p. 1).

[6] H. Cohn, R. Kleinberg, B. Szegedy, and C. Umans, “Group-theoretic algorithms for matrix multiplication,” in *46th Annual IEEE Symposium on Foundations of Computer Science (FOCS’05)*, 2005, pp. 379–388. DOI: 10.1109/SFCS.2005.39 (cit. on p. 1).

[7] H. Nguyen, *Gpu gems 3*, First. Addison-Wesley Professional, 2007, ISBN: 9780321545428 (cit. on p. 1).

[8] D. Bailey and H. Ferguson, “A strassen-newton algorithm for high-speed parallelizable matrix inversion,” in *Supercomputing ’88:Proceedings of the 1988 ACM/IEEE Conference on Supercomputing, Vol. I*, 1988, pp. 419–424. DOI: 10.1109/SUPERC.1988.44680 (cit. on p. 2).

[9] G. Sharma, A. Agarwala, and B. Bhattacharya, “A fast parallel gauss jordan algorithm for matrix inversion using cuda,” *Computers & Structures*, vol. 128, pp. 31–37, 2013, ISSN: 0045-7949. DOI: 10.1016/j.compstruc.2013.06.015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0045794913002095 (cit. on pp. 2, 3, 5).

[10] D. DasGupta, “In-place matrix inversion by modified gauss-jordan algorithm,” *Applied Mathematics*, vol. 4, no. 10, pp. 1392–1396, Sep. 2013. DOI: 10.4236/am.2013.410188 (cit. on pp. 2, 4, 5).

[11] J. Xuebin, C. Yewang, F. Wentao, Z. Yong, and D. Jixiang, “Fast algorithm for parallel solving inversion of large scale small matrices based on gpu,” *The Journal of Supercomputing*, vol. 79, no. 16, pp. 18 313–18 339, May 2023, ISSN: 1573-0484. DOI: 10.1007/s11227-023-05336-7. [Online]. Available: https://link.springer.com/article/10.1007/s11227-023-05336-7 (cit. on p. 2).

[12] B. Cuneo and M. Bailey, “Divergence reduction in monte carlo neutron transport with on-gpu asynchronous scheduling,” *ACM Trans. Model. Comput. Simul.*, vol. 34, no. 1, Jan. 2024, ISSN: 1049-3301. DOI: 10.1145/3626957. [Online]. Available: https://dl.acm.org/doi/10.1145/3626957 (cit. on p. 2).

[13] B. Cuneo, *Harmonize*, version d2eb926, May 7, 2024. [Online]. Available: https://github.com/CEMeNT-PSAAP/harmonize (cit. on pp. 3, 5).

[14] P. Virtanen, R. Gommers, T. E. Oliphant, *et al.*, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020. DOI: 10.1038/s41592-019-0686-2 (cit. on p. 5).

[15] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, Dec. 2011, ISSN: 0098-3500. DOI: 10.1145/2049662.2049663 (cit. on p. 5).

[16] J. R. Gardner, G. Pleiss, D. Bindel, K. Q. Weinberger, and A. G. Wilson, "Gpytorch: Blackbox matrix-matrix gaussian process inference with gpu acceleration," in *Advances in Neural Information Processing Systems*, 2018. DOI: 10.48550/arXiv.1809.11165. [Online]. Available: https://arxiv.org/abs/1809.11165.

[17] I. Dimov, T. Dimov, and T. Gurov, "A new iterative monte carlo approach for inverse matrix problem," *Journal of Computational and Applied Mathematics*, vol. 92, no. 1, pp. 15–35, 1998, ISSN: 0377-0427. DOI: 10.1016/S0377-0427(98)00043-0. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0377042798000430.

[18] 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*, ser. ICDCN '18, Varanasi, India: Association for Computing Machinery, 2018, ISBN: 9781450363723. DOI: 10.1145/3154273.3154300.

[19] E. Agullo, H. Bouwmeester, J. Dongarra, J. Kurzak, J. Langou, and L. Rosenberg, "Towards an efficient tile matrix inversion of symmetric positive definite matrices on multicore architectures," in *High Performance Computing for Computational Science – VECPAR 2010*, J. M. L. M. Palma, M. Daydé, O. Marques, and J. C. Lopes, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 129–138, ISBN: 978-3-642-19328-6. DOI: 10.1007/978-3-642-19328-6_14.