Accelerated Gauss-Huard Algorithm on Hybrid GPU-CPU: Look-Ahead with the Delayed Algorithm Approach

Hisham G. Elzayyadi, Wafaa S. Sayed, Mona A. EL Naggar and Maha A. Hassanein
Engineering Mathematics and Physics Department, Faculty of Engineering, Cairo University
hisham.elzayadi@eng.cu.edu.eg wafaa.s.sayed@eng.cu.edu.eg
monaelnaggar@eng.cu.edu.eg hassanein.maha@eng.cu.edu.eg

***Abstract*—In this paper, we tackle a significant bottleneck - the panel factorization step - in the Gauss-Huard algorithm through a novel parallel computing approach. We address the open question in the literature regarding the feasibility of applying look-ahead in this context. Our strategy combines the use of the delayed Gauss-Huard algorithm with random butterfly transformations instead of the traditional partial pivoting. The proposed technique not only allows for the application of look-ahead but also enhances the overall efficiency of the Gauss-Huard algorithm in a parallel computing environment, presenting possibilities for further optimizations.**

I. Introduction

Dense linear systems of equations are integral to many computational applications, typically solved using direct methods like Gaussian-Elimination (GE), the Gauss-Jordan algorithm (GJ), and the Gauss-Huard algorithm (GH) [1], [2]. Despite both Gauss-Huard (GH) and Gauss-Jordan (GJ) reducing the matrix to a diagonal form, GH is as efficient as Gaussian elimination (GE), requiring $\frac{2}{3}n^3 + \mathcal{O}(n^2)$ floating point operations [3].

As the size of these systems increases, so does the computational complexity, making fine-grain parallelism essential. In this context, block algorithms have emerged as a pivotal breakthrough in linear algebra computations. By processing blocks of data instead of individual elements, these algorithms offer distinct advantages over elementwise counterparts. They excel in contemporary computer architectures that thrive on data locality and efficient exploitation of memory hierarchies. Moreover, block algorithms can leverage high-level Basic Linear Algebra subroutines (BLAS) operations, shifting the focus to dominant matrix multiplication operations. Thereby, the performance across modern systems like multi-core CPUs, GPUs, and other hardware accelerators is enhanced. Maintaining stability is another challenge for numerical algorithms. Pivoting techniques can improve stability, but introduce additional time complexity, which can be mitigated by parallel computing techniques.

While GH and GE have similar computational complexity, GE is often favored in high-performance computing (HPC) because of its potential for parallelism and its stability-enhancing pivoting techniques. Notably, GE's structure allows the application of the look-ahead technique, which optimizes overlapping operations [4], [5]. In contrast, GH's higher data dependency, particularly between panel factorization and other algorithmic steps, limits the applicability of the look-ahead technique [6]. GH initiates solutions before the full system is available, making it promising for large, dense linear systems [3]. Yet, the panel factorization represents a bottleneck that limits its use. Addressing the data dependency enhances the GH performance, making it more competitive in HPC. GH's high data dependency stems from its restriction to partial pivoting with column interchanges. If this is resolved, GH can incorporate look-ahead, improving its performance.

To maintain the stability when forgoing partial pivoting, we use random butterfly transformations (RBT), a preconditioning step that randomizes the matrix for parallel computation, making pivoting unnecessary [7], [8]. RBT stands out for its speed, minimal preprocessing, and postprocessing overhead [4].

This work addresses the research gap concerning the application of the look-ahead technique on the GH algorithm in hybrid CPU-GPU platform. Our research validates that GH, without pivoting, can apply look-ahead to resolve the panel factorization bottleneck. Also, the RBT enables the application of the look-ahead technique maintaining both the accuracy and the stability.

In implementations of this work, we use the GPU library cuBLAS, the CPU's BLAS library, and the MAGMA library, optimized for dense linear algebra on hybrid architectures.

II. Principles of GH Algorithm

GH algorithm is a method for transforming a square nonsingular matrix A into an identity matrix. It was first introduced by Pierre Huard in 1979 [2]. Huard, a French mathematician, developed this method to optimize the GJ process of solving systems of linear equations by applying elementary row operations to the matrix rows. To solve the linear system $Ax = b$, the GH algorithm is applied to the augmented matrix $(A|b)$.

The GH algorithm operates by iteratively constructing an identity matrix of order i in the upper-left corner of the matrix. This construction process involves three primary sequential steps: row elimination, scaling, and column elimination [3]:

1) **Row elimination:** The $(i-1)$ elements preceding the ith element in row i are annihilated by subtracting appropriate multiples of previous rows.

2769-5654/23/$31.00 ©2023 IEEE
DOI 10.1109/CSCI62032.2023.00180

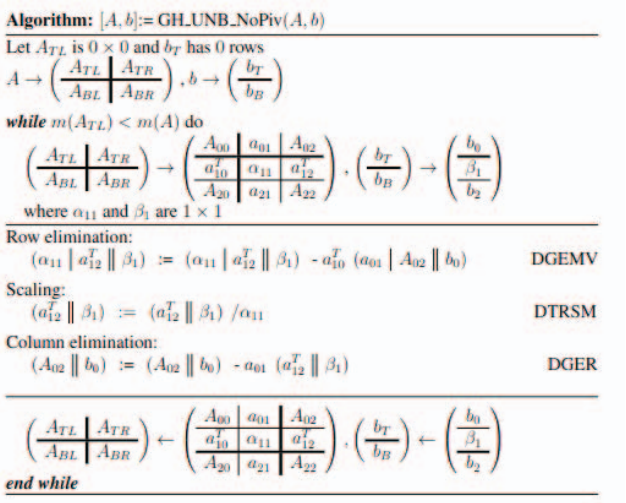

Fig. 1: Gauss-Huard Algorithm with no pivoting (GHNP), highlighting the use of cuBLAS Kernels at each step.

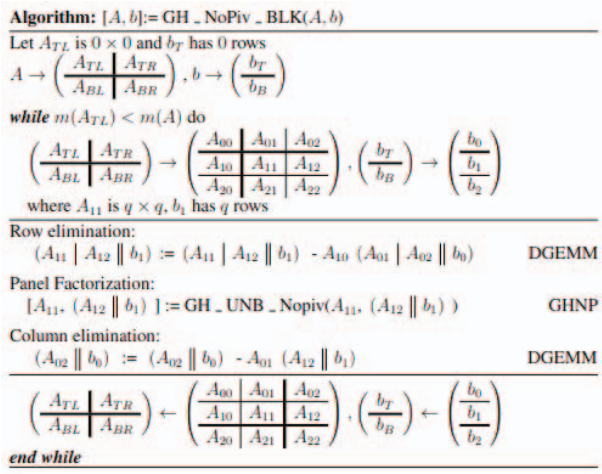

Fig. 2: Block Gauss-Huard Algorithm with no pivoting (BGHNP), highlighting the use of cuBLAS kernels and other functions at each step

2) **Scaling:** The ith row is adjusted by scaling it to ensure that the diagonal element becomes 1.
3) **Column elimination:** The initial $(i-1)$ elements of column i are eliminated by subtracting suitable multiples of row i

Figure 1 represents The operation of the GH algorithm with no pivoting (GHNP) algorithm in FLAME notation. At the end of the algorithm iterative process, the solution vector x replaces the right-hand side b.

A. Block GH Algorithm

Figure 2 represents the Block variant of the GH algorithm (BGHNP). This variant closely parallels the element-wise version, with the primary distinction observed in the scaling step.

In the Block GH approach, the scaling step modifies the sub-matrix A_{11} into an identity matrix. The ideal method to construct this identity matrix involves multiplying the augmented matrix $(A_{11}|A_{12}||b_1)$ by the inverse of A_{11}. However, it's unnecessary to directly compute the inverse of A_{11}. Instead, the update of the sub-matrix $(A_{12}||b_1)$ can be achieved by applying GHNP on the row panel.

This approach of avoiding the inverse computation of A_{11} then using GHNP allows the block GH algorithm to significantly improve its compute throughput compared to its unblocked counterpart.

B. GH Algorithm with Partial Pivoting

Partial pivoting is a pivoting technique adopted in numerical computations, due to its efficiency. In the context of the GH algorithm, the only viable pivoting technique is partial pivoting with column interchanges (GHPP) [3]. This technique, that is illustrated in Figure 3, involves swapping the matrix columns such that the pivot element (Diagonal element used to eliminate entries above or below it) has the largest absolute value in its row. This procedure is executed before the elimination phase, hence improving the numerical stability of the algorithm and enhancing its reliability in solving linear systems. The reason for this exclusive use of column interchanges is that the GH algorithm maintains a specific pattern of zeros in the upper triangular part of the matrix, a pattern that would be disrupted by row interchanges. To the best of our knowledge, partial pivoting with row interchanges is not typically utilized in GH due to these constraints. Consequently, other pivoting techniques like complete pivoting and rook pivoting aren't suitable.

C. Block GH Algorithm with Partial Pivoting

The block variant of the GH algorithm incorporating partial pivoting with column interchanges (BGHPP) is shown in Figure 4. During the panel factorization step, the GHPP algorithm is employed, and column interchanges are performed on the entire column, not just the panel's portion.

III. Application of Look-Ahead with GH

The application of the look-ahead technique in the GH algorithm is an open question in the literature [6]. The primary challenge lies in the inherent complexity and data dependencies associated with the Gauss-Huard algorithm implemented with partial pivoting.

As highlighted by Catalán *et al.* [6], the panel factorization presents a significant challenge in BGHPP, consuming up to half of the execution time on some machines. However, the same study also noted that as the machine's peak performance improves, this ratio decreases significantly, even though panel factorization remains the primary bottleneck of BGHPP.

The objective of the look-ahead technique is to overlap the time-consuming panel factorization step with some of the remaining operations. This approach aims to mitigate the impact of the bottleneck and enhance the overall execution

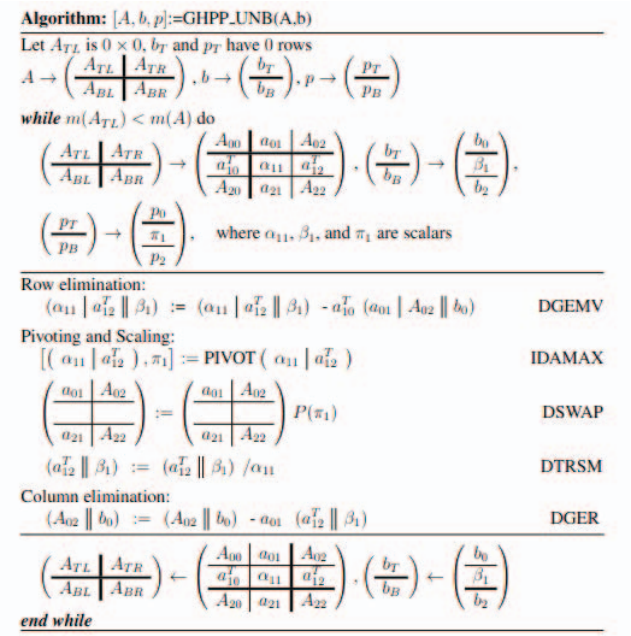

Fig. 3: Gauss-Huard Algorithm with partial pivoting (GHPP),highlighting the use of cuBLAS kernels and other functions at each step

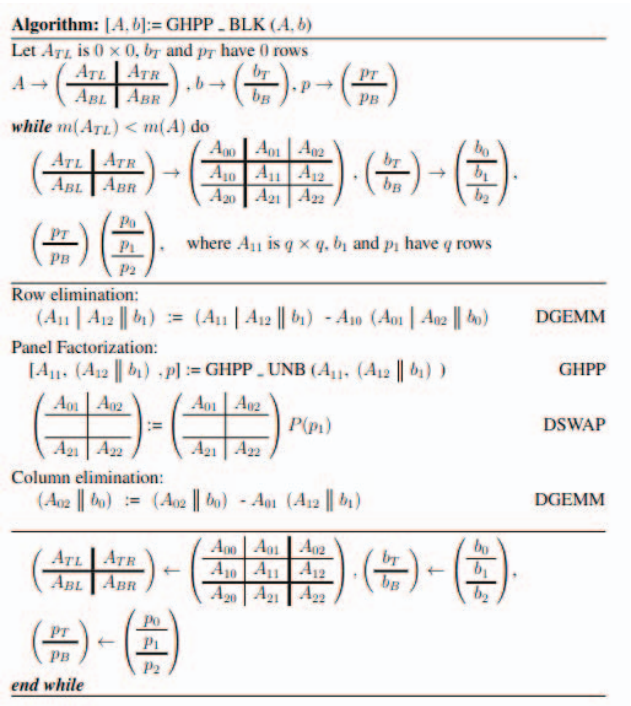

Fig. 4: Block Gauss-Huard algorithm with partial pivoting (BGHPP), highlighting the use of cuBLAS kernels and other functions at each step

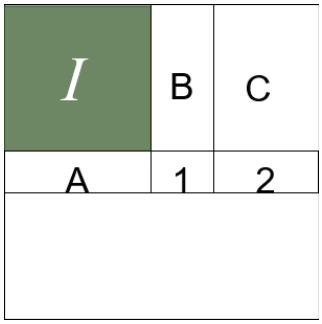

Fig. 5: Block diagram of BGHNP with Look-Ahead

efficiency of the BGHPP algorithm. The main challenge is that the row panel $(A_{11}|A_{12}||b_1)$ must be passed as a whole to GHPP to factorize A_{11} and to influence the subsequent updates to $(A_{12}||b_1)$ as demonstrated by Figure 4. A potential solution to this challenge involves initially factorizing A_{11} followed by updating $(A_{12}||b_1)$. Then we employ the look-ahead technique between factorizing A_{11} in the panel factorization step and updating $(A_{12}||b_1)$ in the row elimination step. However, this method necessitates abandoning partial pivoting, given that the pivoting step requires the entire panel.

The block diagram in Figure 5 illustrates the incorporation of the look-ahead technique into the GH algorithm.

The process proceeds as follows:

1) Tile (1) is updated through row elimination using the matrices A and B.
2) Updated tile (1) is factored without pivoting. Concurrently, the panel (2) is updated using matrices A and C, with the row elimination process utilizing the look-ahead technique.
3) Panel (2) is finally updated as part of the panel factorization step after factorizing tile (1) and row elimination process of panel (2) is finished.

The process of factorizing A_{11} and influencing the subsequent updates to $(A_{12}||b_1)$ can be effectively addressed by utilizing GHNP and the GH Delayed algorithm (GHDEL) [9], respectively.

IV. The GH Delayed Algorithm

The Gauss-Huard Delayed (GHDEL) algorithm was introduced to address the challenge of solving linear systems with multiple right-hand sides, particularly when these right-hand sides are delayed and only become available after solving for one right-hand side with GH [9]. Typically the GH algorithm transforms the coefficient matrix into an identity matrix. However, this process necessitates resolving the problem again after restoring the coefficient matrix with the new right-hand sides.

To overcome this complexity, the GH algorithm focuses solely on the coefficient matrix. It bypasses the computation of zeros and ones, and instead applies only the updates as illustrated in Figs 1,3,2, and 4. This process results in a

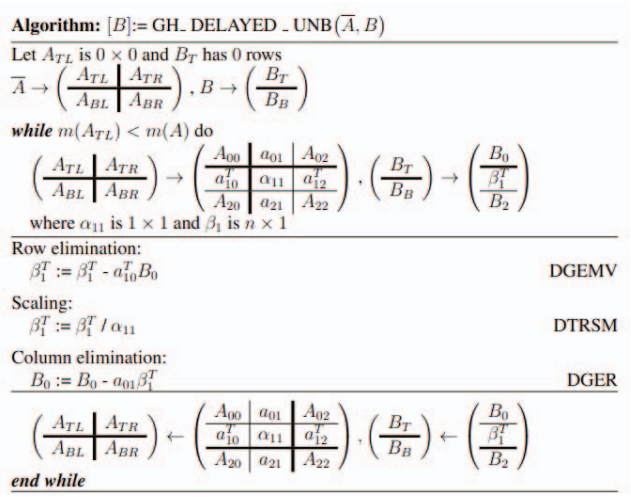

Fig. 6: Gauss-Huard delayed algorithm (GHDEL) given a factored coefficient matrix $\overline{A}$ by Gauss-Huard algorithm

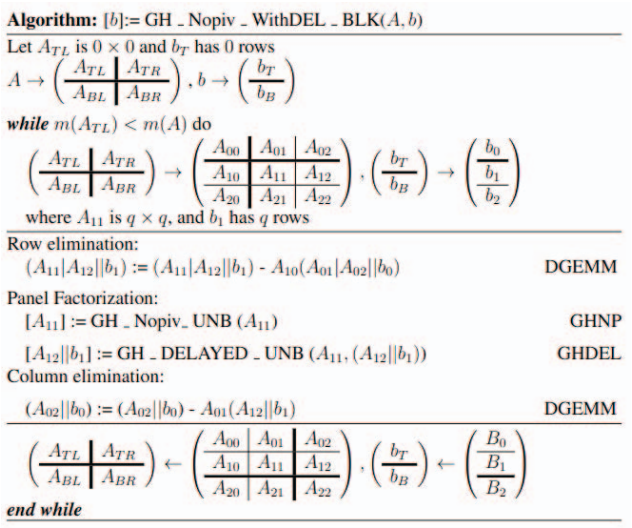

Fig. 7: Block Gauss-Huard algorithm without pivoting with the Delayed algorithm in the Panel Factorization step

factored coefficient matrix, denoted as $\overline{A}$, which can subsequently be solved with any right-hand side. Considering that the coefficient matrix A is transformed into $\overline{A}$ using the unblocked version of the GH algorithm, the solution matrix X is computed and takes the place of the right-hand side matrix B, as detailed in Figure 6.

The row operations of the GHDEL are confined to the right-hand sides B, with no involvement of the matrix $\overline{A}$. The algorithm carries out operations akin to triangular solvers in the LU-based method, resulting in a computational cost that is roughly $2n^2$ flops. This cost is matched to the cost of solving a system via LU factorization, provided the triangular factors L/U have been precomputed. A significant advantage of the GH method is its need for a single matrix sweep, compared to the double sweeps required by LU factorization [9].

Applying the delayed GH algorithm, the panel factorization step is implemented as follows: the tile A_{11} is factored using GHNP then the trailing row panel $(A_{12}||b)$ is updated using GHDEL as shown by Figure 7.

Although implementing look-ahead with BGHNP enhances the parallelism potential, it suffers from decreased accuracy and stability. we propose using Random Butterfly Transformations (RBT) to maintain accuracy and stability compared to those of BGHPP. BGHPP exhibits high stability [10] yet it isn't as suitable for parallel processing as BGHNP. Consequently, RBT can solve these problems by combining the stability of the butterfly scheme and the high parallelism potential in BGHNP.

V. Random Butterfly Transformations

A butterfly matrix is an $n \times n$ matrix defined as follows:

$$B = \frac{1}{\sqrt{2}} \begin{pmatrix} R_0 & R_1 \\ R_0 & -R_1 \end{pmatrix},$$

where R_0 and R_1 are two random non-singular $n/2 \times n/2$ diagonal matrices.

When examining a recursive butterfly matrix with a depth of d, it displays a recursive pattern as documented by Baboulin *et al.* [8]:

$$W^{(n,d)} = \begin{pmatrix} B_1^{(n/2^{d-1})} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & B_{2^{d-1}}^{(n/2^{d-1})} \end{pmatrix} \times .. \\ .. \times \begin{pmatrix} B_1^{(n/2)} & 0 \\ 0 & B_2^{(n/2)} \end{pmatrix} \times B^{(n)} \quad (1)$$

In this context, all the blocks denoted by $B_i^{(k)}$ are butterfly matrices of size k and $B^{(n)}$ is a butterfly matrix of size n. The diagonal values employed in the butterflies are determined randomly and are calculated as $exp(r/10)$, where the value of r is selected randomly from the range $[-0.5, 0.5]$.

The steps followed are taken to solve the general linear system $Ax = b$ using RBT:

1) Calculate the randomized matrix $A_r = U^T AV$, where both U and V are recursive butterfly matrices of depth d
2) Apply BGHNP on $A_r y = U^T b$ to get the solution y.
3) Compute the solution $x = Vy$.

When the depth d is less than $log_2 n$, the butterfly scheme is called partial random butterfly transformations (PRBT). It has been demonstrated in [8] that a depth of 1 or 2 is typically sufficient, although iterative refinement may be necessary in some cases.

VI. Results

A. Test Environment

The remote machine used for computations is A100-SXM4-40GB GPU having precision of $u = 2.22 \times 10^{-16}$ and is licensed by Google Colaboratory. This machine has a

theoretical peak performance at tensor cores double precision of 19.49 TFLOPS and 9.7 TFLOPS for CUDA cores double precision, L2 cache of 40 MB, L1 cache of 192 KB per streaming multiprocessor (SM), DRAM of 40 GB, and a memory bandwidth of 1555 GB/s. The CPU is an 8-core Intel Xeon @ 2.20 GHz with 53 GB of RAM. The MAGMA library version used is 2.7.2 and the linear system is initially read by the CPU and stored in the RAM, and then transferred to the global memory of the GPU for computation. Execution time is recorded from the start of the algorithm iterations until the production of the solution on the GPU. The initial transfer of the system from the CPU to the GPU and the final transfer of the solution back from the GPU to the CPU are not included in the recorded execution time.

The performance of our implementations is assessed by calculating the compute throughput because solving dense linear systems with the block versions of GE and GH attains relatively high arithmetic intensity. The compute throughput is defined as the ratio between the number of FLOPs and the execution time in seconds. Hence, it carries information about both the utilized resources and the execution time. In order to calculate the compute throughput for all implementations; the number of FLOPs for either GE or GH is calculated by the formula $2/3\ n^3 + 3/2\ n^2 - 7/6\ n$, where n is the matrix dimension. The execution time in seconds is calculated by taking the average of 100 runs and the block sizes are chosen so that the execution time is minimum.

In the scaling step with GHNP, GHPP, and GHDEL, we utilize the level 3 cuBLAS kernel DTRSM instead of the level 1 DSCAL. DSCAL in cuBLAS and its MAGMA wrapper only accept a "const double" pointer and pass-by-value scalars, respectively, necessitating additional operations to fetch and pass the scalar. However, by interpreting the scaling process as solving an upper triangular system with multiple right-hand sides (where the matrix is the scalar and the vectors are the right-hand sides), we bypass these complexities.

B. BGHPP Performance

The execution of BGHPP is exclusively GPU-based (GPUBGHPP) due to the absence of overlapping operations that can run concurrently on the CPU and GPU. The block sizes used are: 32 for ($n = 1k - 3k$), 64 for ($n = 4k - 7k$), and 128 ($n = 8k - 20k$). Figure 8 shows that our implementation exhibits significantly low compute-throughput, achieving less than 20% of the peak performance demonstrated by the tensor cores at matrix size 20k.

The Nvidia Nsight system profiler was utilized in our study to pinpoint performance bottlenecks. Figure 9 depicts a visual representation of its analysis. The level 1 cuBLAS kernel, IDAMAX, emerges as the kernel consuming the most time. This kernel is leveraged for partial pivoting to identify the index of the maximum absolute entry in the row.

Despite requiring no FLOPs, the IDAMAX kernel introduces a substantial overhead as it performs ($\frac{1}{2}n^2 - \frac{1}{2}n$) comparisons. To overcome this inefficiency, we substituted partial pivoting with PRBT (BGHPRBT), a method known

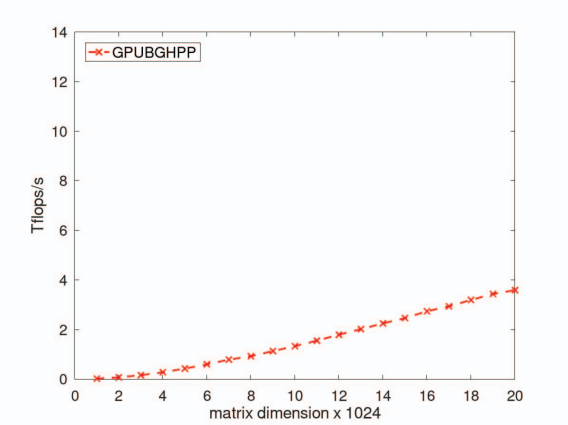

Fig. 8: Compute throughput of BGHPP in a GPU native environment

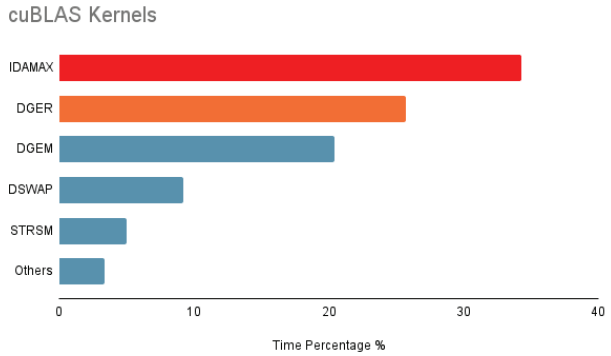

Fig. 9: The time percentage of each BGHPP cuBLAS kernel according to Nvidia Nsight systems

for its stability. Despite the overhead associated with the preprocessing and postprocessing steps of PRBT, it is significantly less than that of partial pivoting, especially for larger matrices. This leads to a significant enhancement in the overall performance of the block GH algorithm.

C. BGHPRBT with Delayed Algorithm-Based Panel Factorization

The BGHPRBT algorithm is implemented by firstly applying the randomization process, then employing BGHNP, as detailed in Figure 7. The panel factorization step is executed by factorizing the diagonal tile A_{11} using GHNP and updating the trailing panel $(A_{12}||b_1)$ using GHDEL as detailed in Figure 6.

The importance of the look-ahead technique becomes evident when comparing BGHPP and BGHPRBT with the delayed algorithm (BGHPRBT_DEL) without applying the look-ahead, as shown in Figure 10. The block sizes chosen for BGHPRBT_DEL are 64 for ($n = 1k - 2k$), and 128 for($n = 3k - 20k$). Both algorithms are implemented natively on the GPU, with BGHPRBT_DEL showing marginally higher

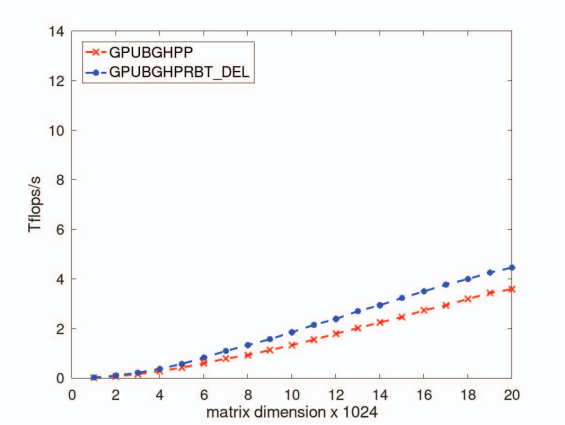

Fig. 10: Compute throughput comparison between BGHPP and BGHPRBT _ DEL for different matrix dimensions in a GPU native environment

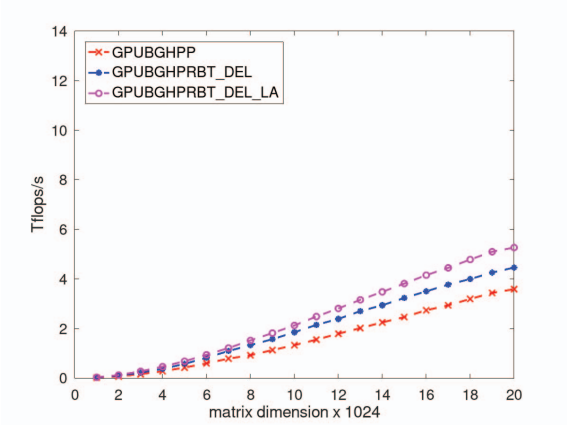

Fig. 11: Compute throughput comparison between BGHPP, BGHPRBT_DEL, and BGHPRBT_DEL_ LA for different matrix dimensions in GPU native environment

compute-throughput, primarily due to the absence of the level 1 cuBLAS kernel IDAMAX.

The use of look-ahead in algorithms allows greater overlap between computational steps, providing significant improvement in the compute-throughput, particularly within the algorithm's bottleneck panel factorization. In the GPU-native implementation of BGHPRBT_DEL with look-ahead (GPUBGHPRBT_DEL_LA), the block sizes used are the same as BGHPRBT_DEL. Figure 11 illustrates that the use of look-ahead technique improves the compute throughput, achieving a speedup of 1.18x with a matrix size of 20k.

Implementing the look-ahead with a hybrid approach allows concurrent usage of the CPU and GPU. Additionally, it transitions the computational architecture from Single Instructions Multiple Data (SIMD) to Multiple Instructions Multiple Data (MIMD) paradigm. Figure 12 illustrates that the hybrid approach yields a speedup of 1.03x at a matrix size of 20k compared to the GPU-native approach. However, it uses different block sizes: 64 for $n = 1k - 9k$, and 128 for $n = 10k - 20k$. This is due to the CPU's speed advantage when executing on small data sets.

The GPU experiences significant kernel latency, which becomes dominant when handling small data sets or when two kernels are executing concurrently, suggesting that the increased latency may result from resource conflicts. The CPU, with latency measured in *ns* rather than μs, outperforms the GPU for small data sets, contributing to the slight superiority of the hybrid implementation.

D. BGHPRBT with GHNP-Based Panel Factorization

In this work, we implemented BGHPRBT to perform the panel factorization step across the entire row panel using GHNP, as detailed in Fig 1 and Fig 2. In this implementation the look-ahead is not applied. Figure 13 shows a performance comparison between the GPU native BGHPRBT (GPUBGHPRBT) and both GPU native and hybrid variations of

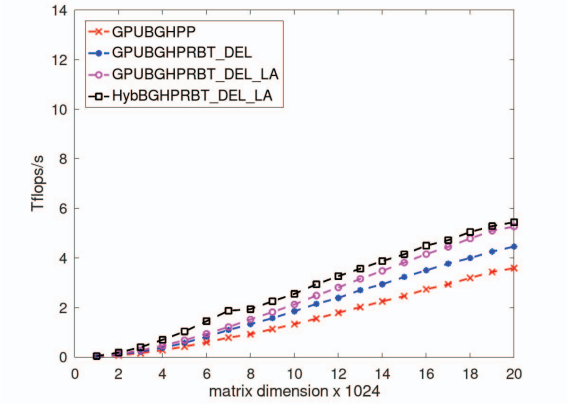

Fig. 12: Compute throughput comparison between BGHPP, BGHPRBT_DEL, and BGHPRBT_DEL_LA in a GPU native environment and BGHPRBT_DEL_LA in a hybrid environment for different matrix dimensions

BGHPRBT_DEL_LA. The block sizes for GPUBGHPRBT are: 32 for $n = 1k - 4k$, 64 for $n = 5k - 8k$, and 128 for $9k - 20k$. GPUBGHPRBT slightly outperforms HybBGHPRBT_DEL_LA, primarily due to the suboptimal performance of the delayed algorithm, as recorded by Benner *et al.* [9]. They demonstrated that the delayed algorithm underperforms the triangular solvers of LU factorization in the cuBLAS library or Intel MKL when dealing with more than 64 right-hand sides.

The better performance of GPUBGHPRBT over GPUBGHPRBT_DEL_LA is largely due to reduced kernel latency. Both GHNP and GHDEL utilize the same kernels and have identical inner loop iterations. The only difference lies in their memory access patterns. In BGHPRBT during the panel factorization, using GHNP for the entire row panel $(A_{11}|A_{12}||b_1)$ results in half the kernel calls compared to BGHPRBT_DEL_LA, which

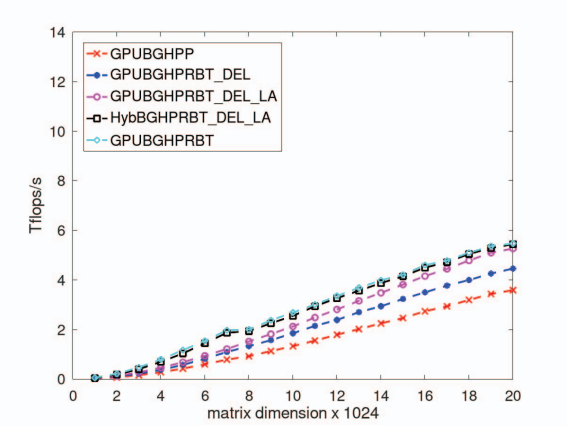

Fig. 13: Comparison of Compute Throughput between the GPU native BGHPRBT and the delayed algorithm BGHPRBT-DEL with look-ahead in a hybrid machine

uses GHNP for A_{11} factorization and GHDEL for updating $(A_{12}||b_1)$. Table I summarises the performance results for all variations of BGHPRBT, illustrating the relative speedup of each implementation in comparison to BGHPP.

TABLE I: Comparison of Speedups for Different Versions of Block GH with PRBT vs. BGHPP, with a Matrix Size of 20k and Block Size of 128.

Platform	Panel Factorization	Look-ahead	Speedup
GPU	GHNP-based	not applied	1.524
GPU	Delayed-based	not applied	1.243
GPU	Delayed-based	applied	1.468
CPU-GPU	Delayed-based	applied	1.516

VII. Conclusion

This work offers valuable insights into the Gauss-Huard algorithm, highlighting its potential particularly in heterogeneous environments. We propose a hybrid technique, termed Gauss-Huard with Partial Random Butterfly Transformations, which effectively addresses the bottleneck of panel factorization. This is accomplished by utilizing the delayed algorithm approach of GH in conjunction with the look-ahead technique for the first time and ensuring stability through the randomization scheme. Our results emphasize the potential for parallelism of the Gauss-Huard algorithm. The implementation of the look-ahead technique improves the performance on the hybrid CPU-GPU platform. However, it does not deliver the expected performance gains, primarily due to the high kernel latency of the delayed algorithm. Additionally, our proposed method demonstrates notable scalability characteristics. As the problem size increases, the compute throughput steadily approaches peak performance, indicating that our method can effectively handle larger problem sizes without significant performance degradation. This scalability, makes it a robust solution for large dense linear systems. Future research is expected to focus on optimizing the delayed algorithm or exploring alternative solutions to enable the look-ahead technique with Gauss-Huard to significantly enhance the performance.

Acknowledgment

The authors would like to thank the software engineer Hady Elzayyadi at Pixelogic media for his help in configuring the project dependencies.

References

[1] L. N. Trefethen and D. Bau III, "Numerical linear algebra, vol. 50," 1997.

[2] P. Huard, "La méthode simplex sans inverse explicite," *EDB Bull, Direction Etudes Rech. Sér. C Math. Inform*, vol. 2, pp. 79–98, 1979.

[3] W. Hoffmann, "The gauss-huard algorithm and lu factorization," *Linear algebra and its applications*, vol. 275, pp. 281–286, 1998.

[4] S. Donfack, J. Dongarra, M. Faverge, M. Gates, J. Kurzak, P. Luszczek, and I. Yamazaki, "A survey of recent developments in parallel implementations of gaussian elimination," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 5, pp. 1292–1309, 2015.

[5] M. Baboulin, S. Donfack, J. Dongarra, L. Grigori, A. Rémy, and S. Tomov, "A class of communication-avoiding algorithms for solving general dense linear systems on cpu/gpu parallel machines," *Procedia Computer Science*, vol. 9, pp. 17–26, 2012.

[6] S. Catalán, P. Ezzatti, E. S. Quintana-Ortí, and A. Remón, "The impact of panel factorization on the gauss-huard algorithm for the solution of linear systems on modern architectures," in *Algorithms and Architectures for Parallel Processing: 16th International Conference, ICA3PP 2016, Granada, Spain, December 14-16, 2016, Proceedings*. Springer, 2016, pp. 405–416.

[7] D. S. Parker, "A randomizing butterfly transformation useful in block matrix computations," 1995.

[8] M. Baboulin, J. Dongarra, J. Herrmann, and S. Tomov, "Accelerating linear system solutions using randomization techniques," *ACM Transactions on Mathematical Software (TOMS)*, vol. 39, no. 2, pp. 1–13, 2013.

[9] P. Benner, P. Ezzatti, E. S. Quintana-Ortí, and A. Remón, "Extending the gauss–huard method for the solution of lyapunov matrix equations and matrix inversion," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 9, p. e4076, 2017.

[10] T. Dekker, W. Hoffmann, and K. Potma, "Stability of the gauss-huard algorithm with partial pivoting," *Computing*, vol. 58, no. 3, pp. 225–244, 1997.