Enhancing QR Decomposition: A GPU-based Approach to Parallelizing the Householder Algorithm with CUDA Streams

Uppu Eshwar, Soumyajit Chatterjee, and Sathya Peri

Indian Institute Of Technology Hyderabad, Telangana 502285, India
{ch21btech11034,ai22mtech02005}@iith.ac.in
sathya_p@cse.iith.ac.in

Abstract. Linear algebra algorithms, such as the Householder QR decomposition, are pivotal in various applications including signal processing, optimization, and numerical solutions to systems of linear equations. Traditional sequential implementations of the Householder algorithm face significant limitations in terms of performance and scalability when applied to large matrices. To overcome these constraints, this paper explores the parallelization of the Householder QR algorithm on Graphics Processing Units (GPUs) using CUDA, a parallel computing platform and programming model developed by NVIDIA. Our method ensures the availability of critical intermediate data, distinguishing it from standard libraries like cuSOLVER, which modify the processing order and often discard important intermediate computations. By leveraging CUDA streams, we achieve enhanced parallelism without compromising the integrity of the algorithm's sequence or the accessibility of intermediate data. Our performance analysis reveals that our implementation achieves efficiency comparable to cuSOLVER, making it a viable option. This study not only presents a novel implementation but also extends the potential for GPU-accelerated linear algebra procedures to benefit a wider range of scientific and engineering applications

Keywords: QR Decomposition · Parallel Computing · General Purpose GPU programming · CUDA streams.

1 Introduction

The goal of any constrained optimization method is to find the values of the decision variables that optimize the objective function while adhering to the given constraints[16][24]. The solution obtained must satisfy both the optimization objective and the constraints simultaneously.

Techniques to solve constrained optimization problems include mathematical programming, linear programming[17], quadratic programming[13], non-linear programming[22], and more specialized methods such as interior point methods[21] and evolutionary algorithms[23]. Numerical methods like Sequential Quadratic Programming (SQP)[2] solve non-linear programming problems with

both equality and inequality constraints and has been widely applied in various fields, including engineering design, optimal control, and operations research. It is effective for solving non-linear programming problems with moderate-sized dimensions and smooth objective functions and constraints.

However when solving optimization problems using Sequential Quadratic Programming (SQP) involving high dimensional data, the computational complexities can become significant. Some challenges and associated complexities include:

- High-dimensional decision variables: As the number of decision variables increases, the size of the quadratic programming sub-problem grows quadratically. The computational complexity of solving a quadratic programming problem is generally $O(n^3)$ where n represents the number of decision variables.
- Storage requirements: Large-scale optimization problems may require substantial memory for storing matrices and vectors used in the quadratic programming sub-problem. As the dimensionality increases, the memory requirements can become a limiting factor, especially if the problem involves dense matrices.
- Evaluation of gradients and Hessians: Computing gradients and Hessians of the objective function and constraint functions is essential for solving the quadratic programming subproblem. In large-scale problems, evaluating these derivatives can be time-consuming, particularly if the derivatives need to be computed numerically or if the functions are complex and computationally expensive.
- Numerical stability: The numerical stability of SQP can become an issue in large-scale problems due to the accumulation of rounding errors and ill-conditioning of the matrices involved. This can lead to numerical instabilities, convergence issues, and inaccurate results.
- Scalability: The scalability of SQP in terms of both computation time and memory usage can become problematic with large-scale dimensions. The time required for each iteration may be quite large, and memory limitations can prevent the use of direct methods for solving the quadratic programming sub-problem.

Prior works like [11] suggests mathematical variations on the standard SQP iteration with [15] proposing a further improvement to it which leads to a faster rate of convergence. While works like [19] suggest alternative methods for the approximation of Quasi-Newton matrices, works like [5] and [8] tackle the storage requirements for computing the Hessian in large scale problems. Also methods like [1] try to effectively utilize the sparsity of high dimensional matrices involved in the problem.

In our work we tackle the problem of scalability by optimizing the QR factorization subroutine which is used in iterations of the SQP algorithm where QR decomposition is utilized to transform the equality constraints into an orthogonal form, which simplifies the formulation and solution of the Karush-Kuhn-Tucker

(KKT)[4] system of equations. QR decomposition is also used to determine the active set of constraints at each iteration of SQP. By decomposing the constraint Jacobian matrix using QR factorization, the linear independence of the constraints can be assessed. The active set, which includes the constraints that are most likely to be active at the solution, can be identified based on the QR factorization. This information helps in constructing the working set of constraints and updating the constraint active set efficiently. QR decomposition can also be used in Hessian approximation and its inverse or in reduced Hessian approximation methods[3] to approximate only the portion of the Hessian matrix relevant to the current iteration of SQP.

Our contribution proposes improvements to the QR factorization kernel used in the Sequential Least Squares Quadratic Programming (SLSQP) [14] algorithm implemented in NLopt[12], a widely used open source software package, using parallel GPU programming methods.

2 Background

QR factorisation is an essential algorithm in linear algebra that decomposes a matrix of any size into the product of two matrices, Q and R, where Q is unitary[9], and R is upper triangular[9]. This technique is widely used in solving several problems such as systems of linear equations, least squares approximations, eigenvalue computation, matrix rank determination, orthogonalisation etc. Hence, developing an efficient solution for QR factorisation can benefit these applications.

The algorithm used in the NLopt [12] library to perform QR factorization is Householder Transformation as given in the `Algorithm 1` below.

Algorithm 1 HouseHolder Transform

```
1: for pivot = 0 to m do
2:   (x, y, z) = update_pivot(pivot row in mat)
3:   mat[pivot][pivot] = x
4:   for j = 1 to pivot-1 do
5:     sm = dot_product(jth row, pivot row)
6:     if sm ≠ 0 then
7:       for k = pivot+1 to n do
8:         mat[j][k] += sm * mat[pivot][k]
9:       end for
10:    end if
11:  end for
12: end for
```

The implementation of the Householder transformation within the library accepts a matrix of arbitrary size as input and transforms it into an upper triangular matrix in-place. The computational process within the for loop can be divided

into two primary subroutines: the update_pivot kernel and the dot_product kernel. The update_pivot kernel is responsible for processing the pivot row and updating the pivot element. The pivot row is the row utilized to update all other rows, excluding itself, during a single iteration of the for loop in Line 1 of `Algorithm 1`. The pivot element corresponds to the first element of the pivot row in the initial iteration, the second element in the subsequent iteration, and so forth. This is because, in each iteration of the for loop, not only is each row sequentially selected as the pivot in the matrix, beginning with the first row, but one column is also skipped after each iteration. The remaining submatrix is then used for further computations in the subsequent iterations.

After processing the pivot row and updating its pivot element, the algorithm proceeds to compute the dot product between the pivot row and each of the remaining rows in the matrix. The scalar value obtained from the dot product between the pivot row and any other row is then utilized to update the corresponding row in the matrix, as illustrated in Line 8 of `Algorithm 1`. The algorithm concludes once the last row of the matrix has been processed as the pivot row and its corresponding pivot element has been updated. Since no rows remain after the final row, no further computations occur following the processing of the last pivot row.

The algorithm exhibits a sequential nature due to the presence of two critical dependencies. The first dependency occurs within the same iteration: the algorithm cannot proceed to update the other rows until the pivot row has been fully processed. The second dependency spans across iterations: a row cannot be updated in the current iteration until its update from the previous iteration is complete, as each iteration's matrix update relies on the matrix produced in the prior iteration. These inter-dependencies present a significant challenge when attempting to parallelize the Householder Algorithm, as described in `Algorithm 1`

3 Methodology

3.1 Introduction to Parallel GPU Programming and CUDA

The advent of Graphics Processing Units (GPUs) has revolutionized computational methods in fields such as scientific computing and optimization. Unlike Central Processing Units (CPUs), which excel at sequential processing, GPUs are designed with numerous smaller cores capable of parallel thread execution. This architecture is ideal for data-parallel tasks, where the same operation is performed across multiple data elements simultaneously[18]. CUDA (Compute Unified Device Architecture) by NVIDIA extends C/C++ to harness GPU power, offering fine-grained control over thread hierarchy, memory access, and synchronization. In CUDA's model, threads are organized into blocks (up to 1024 threads each) and further into grids, enabling scalable parallel computations[25].
Linear algebra is fundamental to many optimization techniques, including Sequential Least Squares Quadratic Programming (SLSQP), which requires efficient matrix operations like QR decomposition and matrix multiplications.

GPUs, with their parallel architecture, are well-suited for these tasks [10]. Libraries like cuBLAS and cuSOLVER optimize these computations for GPUs, with cuBLAS providing accelerated BLAS routines [6] and cuSOLVER offering solvers for linear systems, eigenvalue problems, and matrix factorizations. As a GPU extension of LAPACK, cuSOLVER simplifies the integration of GPU-optimized solvers into existing codebases.[7].

3.2 Limitations of Traditional QR Factorization in cuSOLVER

cuSOLVER's QR factorization algorithm is highly optimized for parallel execution on GPUs, using batched operations to maximize memory efficiency and computational throughput. It designed as an extension of the LAPACK library for GPUs and typically employs a "left-looking" approach for general-purpose QR decomposition[20], which updates the matrix in-place, focusing on the left-most columns first. This method is efficient for batch processing but may not align well with algorithms requiring iterative updates with preserved intermediate states. This inherent limitation is why direct application of cuSOLVER's QR routines might not be suitable for the optimization context discussed in this research, where maintaining and utilizing intermediate values is critical.

To account for the limitations of cuSOLVER's QR decomposition routines, particularly its inability to preserve the iteration order and intermediate values required for iterative algorithms, we propose a novel approach that leverages the parallel processing capabilities of GPUs while maintaining intermediate states essential for iterative computations

3.3 Parallelization Strategy for QR Factorization

The sequential QR factorization algorithm, as previously described, involves two primary computations: (1) updating the pivot element of the pivot row and (2) computing the dot product of each row below the pivot row with the pivot row, followed by updating each row using the scalar value obtained from the dot product. Our approach to QR factorization centres on exploiting potential parallelism within the existing sequential code. Since each scalar product calculation of each row is independent of the others, these operations can be executed concurrently across multiple GPU processing elements. This parallel execution significantly reduces the overall computational time. Furthermore, the update for a next iteration's pivot element does not depend on the completion of all row updates in the current iteration. Instead, the update for the next pivot element can commence as soon as the scalar product involving the immediate next row is completed. This overlapping of computations enables a seamless transition between iterations, minimizing idle time on GPU processing units. To implement this, we utilized CUDA streams, which allow for asynchronous execution of these operations.

To efficiently parallelize these computations on a GPU, we divided the algorithm into two distinct CUDA kernels.

3.4 Kernel Design for Pivot Element Update

Algorithm 2 UpdatePivotElement Kernel

```
1: Kernel UpdatePivotElement(Matrix, PivotRow, NumCols)
2: Shared TempArray[512], SumOfSquares
3: Initialize TempArray[ThreadIndex] ← 0
4: Initialize SumOfSquares ← 0
5: Synchronize Threads
6: Compute Iterations ← (NumCols + BlockDim - 1) / BlockDim
7: for each iteration it from 0 to Iterations-1 do
8:    Calculate ColumnIndex ← it * BlockDim + ThreadIndex + PivotRow
9:    if ColumnIndex < NumCols then
10:      Load Matrix[PivotRow * NumCols + ColumnIndex] into TempValue
11:      TempArray[ThreadIndex] ← TempValue^2
12:   end if
13:   Synchronize Threads
      /*Before performing the reduction, ensure that the shared memory is fully
      loaded */
14:   Perform Reduction of TempArray to SumOfSquares
15: end for
16: Synchronize Threads
    //Only a single thread writes into the shared variable to avoid race conditions
17: if ThreadIndex == 0 then
18:   PivotValue ← sqrt(SumOfSquares)
19:   if Matrix[PivotRow * NumCols + PivotRow] > 0 then
20:      PivotValue ← -1 * PivotValue
21:   end if
22:   Update Matrix[PivotRow * NumCols + PivotRow] ← PivotValue
23:   Store Matrix[PivotRow * NumCols + PivotRow] - PivotValue in Updat-
      edPivot
24: end if
```

The first kernel is designed to update the pivot element by calculating the sum of squares of the matrix row elements. This computation is executed using a single block of threads, with the number of threads dynamically determined based on the matrix row size. The kernel loads the elements from the global memory and stores the squared values in shared memory, which is a faster on-chip memory space accessible by all threads within a block. Given that the block size may be smaller than the row size of the matrix, the kernel processes the row in chunks, iterating through each chunk, since a single block of threads is used to avoid atomics. After loading and squaring the elements, the kernel performs a

reduction sum on the shared memory array to compute the total sum of squares for the row. This value is then used to update the pivot element by taking its square root. The design of this kernel allows for efficient memory access and minimizes the number of global memory transactions, thereby enhancing performance.

3.5 Kernel Design for Row Update via Dot Product Computation

The second kernel focuses on the computation of the dot product between the pivot row and the remaining rows, followed by updating the rows using the dot product. This kernel is designed, keeping in mind the two-dimensional structure of thread blocks. Each block processes a fixed number of rows and columns, where the y-dimension handles rows and the x-dimension handles columns as shown in `figure 1` . By allocating multiple blocks along the y-dimension in the grid, the kernel exploits the independence of row updates among the rows which allows rows to be processed in chunks avoiding the need for explicit synchronization or atomic operations.

Algorithm 3 UpdateMatrix Kernel

```
1: Kernel UpdateMatrix(Matrix, PivotRow, NumRows, NumCols)
2: /* ThreadX - runs from 0 to ColsPerBlock
3: ThreadY - runs from 0 to RowsPerBlock*/
4:
5: Shared PivotValues[ColsPerBlock]
6: Shared TempResults[RowsPerBlock][ColsPerBlock]
7: Shared RowSums[RowsPerBlock]
8: Initialize TempResults[ThreadY][ThreadX] ← 0
9: Initialize RowSums[ThreadY] ← 0
10: Synchronize Threads
11:
12: RowIndex ← PivotRow + BlockIdxY * BlockDimY + ThreadY + 1
13: if RowIndex ≥ NumRows then
14:    Exit Kernel
15: end if
```

The algorithm begins by storing pivot row elements in shared memory for rapid access by all threads within a block. The kernel then calculates the dot product between the pivot row and each remaining row below the pivot row, by iterating column-wise with each thread computing the product of corresponding elements. These results are stored in shared memory, followed by a reduction sum to obtain the final dot product for each row, which is then used to update the rows. This efficient use of shared memory and thread/block organization minimizes latency and maximizes GPU throughput.

Algorithm 3 UpdateMatrix Kernel CONTD.

```
Iterations ← (NumCols - PivotRow + ColsPerBlock - 1) / ColsPerBlock
for it = 0 to Iterations - 1 do
    Compute ColStart ← PivotRow + it * BlockDimX
    if ThreadY == 0 and ColStart + ThreadX < NumCols then
        PivotValues[ThreadX] ← Matrix[(PivotRow ) * NumCols + ColStart + ThreadX]
    end if
    Synchronize Threads
    if ColStart + ThreadX < NumCols then
        TempResults[ThreadY][ThreadX] ← Matrix[RowIndex * NumCols + ColStart + ThreadX] * PivotValues[ThreadX]
    else
        TempResults[ThreadY][ThreadX] ← 0
    end if
    Synchronize Threads
    Reduce TempResults[ThreadY] to RowSums[ThreadY]
    Synchronize Threads
end for

if ThreadX == 0 then
    RowSums[ThreadY] ← RowSums[ThreadY] + Matrix[RowIndex * NumCols + PivotRow] * UpdatedPivot[PivotRow ]
    if RowSums[ThreadY] ≠ 0 then
        ScalingFactor ← 1 / RowSums[ThreadY]
        Matrix[RowIndex * NumCols + PivotRow - 1] ← Matrix[RowIndex * NumCols + PivotRow ] + ScalingFactor * UpdatedPivot[PivotRow ]
    end if
end if
Synchronize Threads

for it = 0 to Iterations - 1 do
    Compute ColIndex ← it * BlockDimX + ThreadX + PivotRow
    if ColIndex < NumCols then
        Matrix[RowIndex * NumCols + ColIndex] ← Matrix[RowIndex * NumCols + ColIndex] + RowSums[ThreadY] * Matrix[(PivotRow ) * NumCols + ColIndex]
    end if
end for
```

3.6 Host Function and Streams-Based Execution

In our parallelized QR factorization implementation, the host function is integral in managing the execution flow across multiple CUDA streams, ensuring

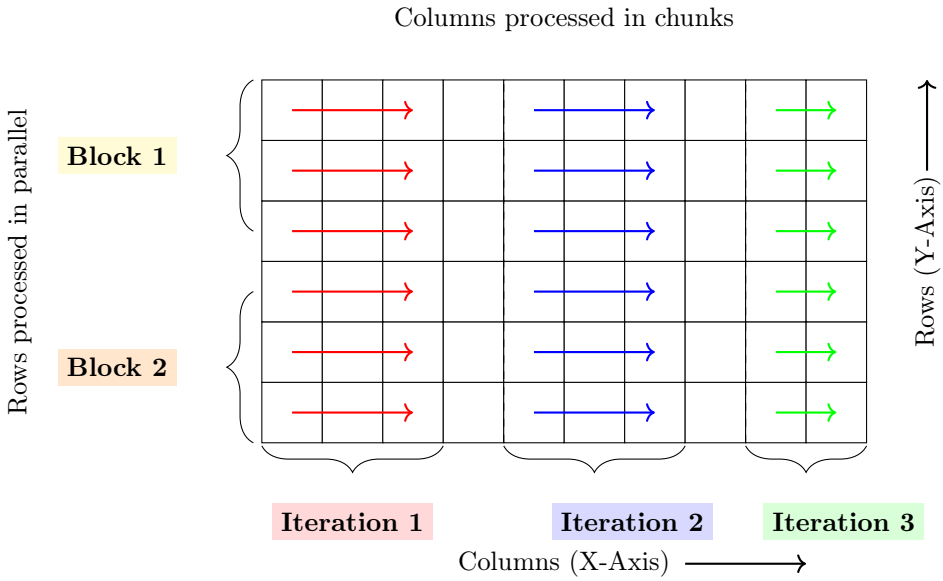

Fig. 1: Illustration of Algorithm 3: Parallel row processing with iterative column processing in chunks. Blocks process rows in parallel, with columns processed in chunks over multiple iterations.

efficient parallel processing. This function is responsible for memory allocation, data transfer between host and device, and synchronization of the various computational tasks executed by the GPU. By leveraging CUDA streams, we achieve concurrent kernel execution, significantly reducing idle time and enhancing overall computational efficiency.

Algorithm 4 Stream-based Execution of Kernels for Matrix Update

Input: Matrix A, Number of rows m, Number of columns n
Initialize: CUDA streams: `Stream 1`, `Stream 2`
Initialize: CUDA events: `Event 1`, `Event 2`
// Kernel 1 stands for update pivot element kernel and kernel 2 stands for update matrix kernel
// parameters for kernel 1 are Matrix, PivotRow, NumCols
// parameters for kernel 2 are Matrix, PivotRow, NumRows, NumCols
// Initial kernel launch to update the first pivot element
`kernel1`$<<<gridDim,BlockDim,stream>>>(A, 0, n)$ `on default stream`
`cudaDeviceSynchronize()`

The implementation strategically utilizes two CUDA streams—designated as Stream 1 and Stream 2—to orchestrate the execution of the two kernels. Stream 1 is tasked with executing the kernel that updates the pivot element, along with executing the kernel responsible for processing the row immediately following the pivot row. Concurrently, Stream 2 handles the updates for the remaining rows, specifically excluding the row next to pivot row which is getting updated concurrently in the Stream 1.

Algorithm 4 Stream-based Execution of Kernels for Matrix Update CONTD.

1: **for** $lpivot = 1$ to m **do**
2: **if** $lpivot == 1$ **then**
3: // Stream 1 processes the next row after the pivot row
4: `kernel2`<<<$gridDim$,$BlockDim$,$stream$>>>(A,$lpivot$,m,n) `on Stream 1`
5: // Stream 2 processes the remaining rows
6: /* compute grid computes the number of blocks required to process (m - lpivot - 1) rows in y-direction and n-lpivot coloumns in x-direction for a fixed number of threads per block in x and y directions*/
7: `gridDim` = computeGrid($m - lpivot - 1$)
8: `kernel2`<<<$gridDim$,$BlockDim$,$stream$>>>(A, $lpivot$, m, n) `on Stream 2`
9: `cudaEventRecord`(Event 2, Stream 2)
10: `kernel1`<<<$gridDim$,$BlockDim$,$stream$>>>(A, $lpivot + 1$, m, n) `on Stream 1`
11: `cudaEventRecord`(Event 1, Stream 1)
12: **else**
13: // wait for events to complete to ensure correct execution order
14: `cudaStreamWaitEvent`(Stream 1, Event 2, 0)
15: `kernel2`<<<$gridDim$,$BlockDim$,$stream$>>>(A, $lpivot$, m, n) `on Stream 1`
16: `cudaStreamWaitEvent`(Stream 2, Event 1, 0)
17: `kernel2`<<<$gridDim$,$BlockDim$,$stream$>>>(A, $lpivot$, m, n) `on Stream 2`
18: `cudaEventRecord`(Event 2, Stream 2)
19: `kernel1`<<<$gridDim$,$BlockDim$,$stream$>>>(A, $lpivot + 1$, m, n) `on Stream 1`
20: `cudaEventRecord`(Event 1, Stream 1)
21: **end if**
22: **end for**

During the initial iteration, Streams 1 and 2 function independently. However, as factorization advances, inter-stream dependencies are established to ensure data consistency through CUDA event synchronization mechanisms, such

as cudaEventRecord and cudaStreamWaitEvent. This synchronization prevents Stream 2 from initiating operations until the required computation in Stream 1 is complete, and vice versa as shown in `figure 2` . Given that the kernel in `Algorithm 3` is sufficiently large to achieve 100 % GPU occupancy, the decision to limit the implementation to two streams optimizes GPU utilization while mitigating resource contention and operational complexity. Although incorporating additional streams could theoretically enhance parallelism, it would also introduce greater complexity and diminish cache efficiency. Therefore, the use of two streams represents an optimal balance between performance and complexity.

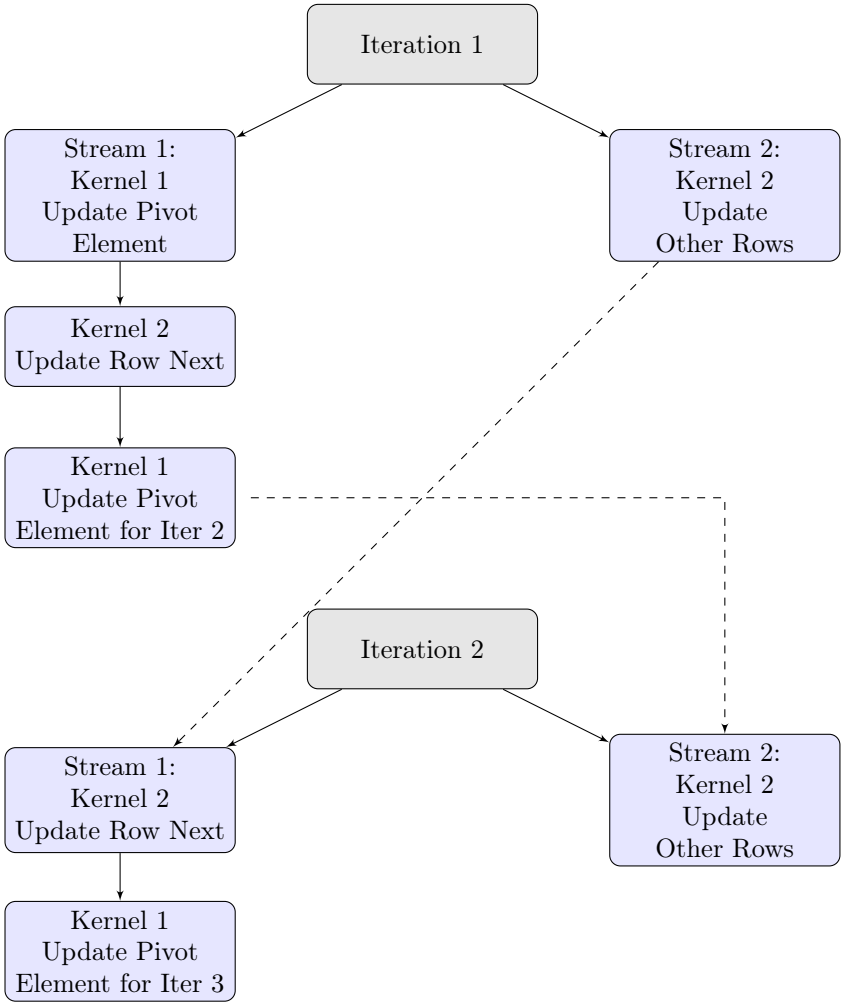

Fig. 2: Workflow of Host Function Using Two CUDA Streams: Dependencies Across Iterations. Stream 1 handles the pivot element and updating the next row, while Stream 2 concurrently processes the remaining rows. The streams synchronize using events recorded at the end of their execution.

4 Results

In this section, we present the results of our experiments, which were designed to optimize the kernel in `Algorithm 3`. We conducted four key experiments: first, we explored the optimal configuration of threads per block for `Algorithm 3`; second, we investigated the trade-offs between block dimensions in a 2D thread structure; third, we compared the performance of our approach with optimised configuration against the cuSOLVER library across varying matrix sizes; and finally we demonstrate the improvements in the original SLSQP algorithm[14] of the NLopt Library[12], using our proposed method.

4.1 Experimental Setup

The experiments were conducted on an Nvidia Quadro RTX 5000 GPU server, equipped with 48 Streaming Multiprocessors (SMs) supporting CUDA Compute Capability 7.5. Each multiprocessor can handle up to 1024 threads, distributed across multiple blocks, with a maximum of 64 active warps per SM. The server also includes 16 GB of GDDR6 memory, providing substantial bandwidth to support high-performance computation.

4.2 Experiment 1

In the first experiment, we aimed to identify the optimal number of threads per block for `Algorithm 3` by varying the thread count and observing its effect on GPU performance. For the experiment , we perform QR decomposition on a matrix of size 1024 x 1024 using our proposed approach. `Figure 3` presents the performance metrics for different thread configurations.

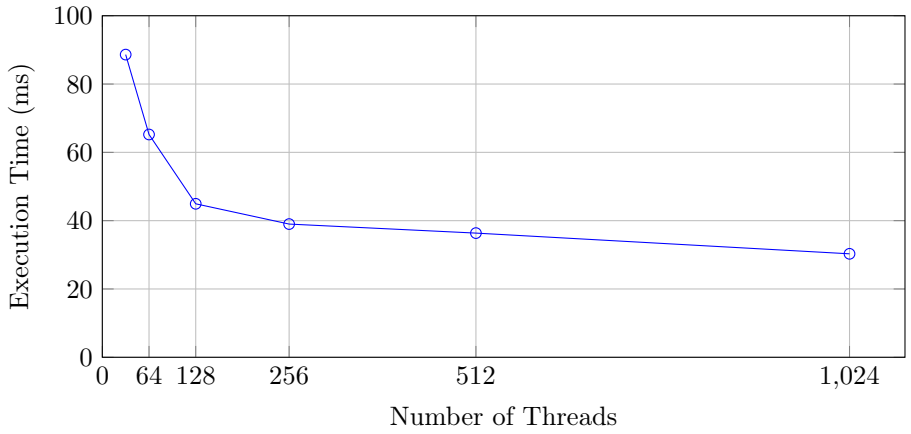

Fig. 3: Execution Time vs. Number of Threads

Our experiment demonstrated that utilizing the maximum number of threads per block i.e. 1024 was optimal, achieving 100% utilization of the GPU. It is important to note that while a block size of 512 threads allowed two active blocks per multiprocessor, the 1024-thread configuration resulted in only one active block per multiprocessor. However, the performance was not solely dictated by the number of active blocks. Despite the reduction in active blocks, the configuration with 1024 threads per block yielded better performance. This suggests that other factors, such as reduced scheduling overhead and improved memory bandwidth utilization, played a significant role in optimizing `Algorithm 3`'s execution. Therefore, we concluded that 1024 threads per block is the optimal configuration for `Algorithm 3` in our setup.

4.3 Experiment 2

In this experiment, we focused on further optimizing the kernel in `Algorithm 3`. Building on the results from the previous experiment, we kept the number of threads per block as 1024 and explored variations in the distribution of threads across the X and Y directions while performing the QR decomposition. `Figure 4` presents the performance metrics for different configurations.

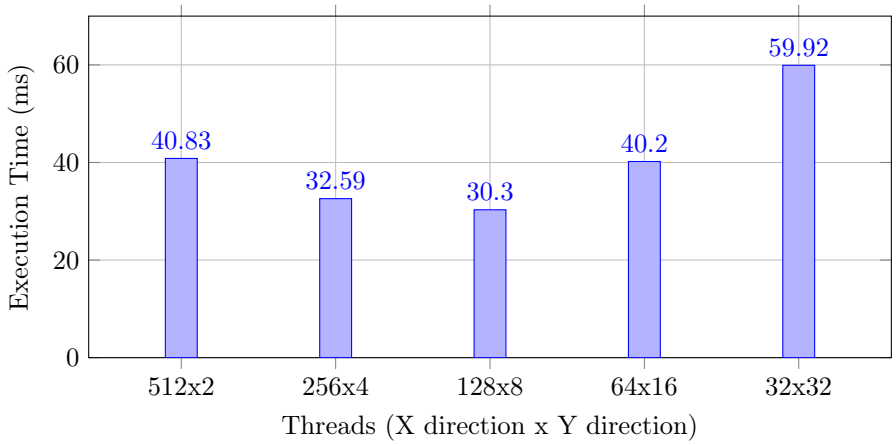

Fig. 4: Scaling with Threads in X and Y Direction for UpdateMatrix Kernel (1024x1024 Dataset)

Our primary objective was to identify an optimal balance between the number of blocks, influenced by the thread count in the Y direction, and the number of iterations each block needs to perform in the X direction to process all columns. The experimental findings suggest that the optimal thread configuration for the X and Y directions is (128 x 8).

4.4 Experiment 3

Based on the results obtained from the previous experiments, we conducted a comparative analysis of the QR decomposition algorithm from libraries like cuSolver(GPU), LAPACK(CPU) and PLASMA(CPU) against our proposed approach, across various square matrices of different sizes. The results are illustrated in `Figure 5`.

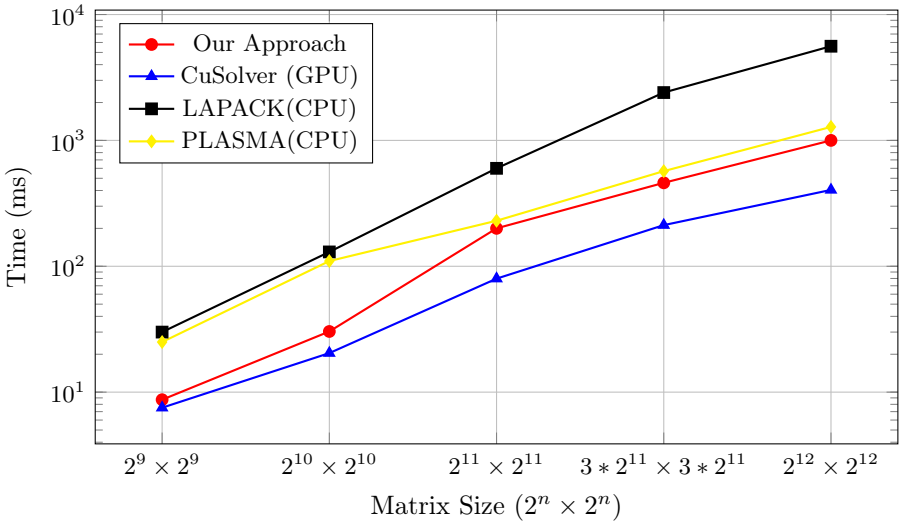

Fig. 5: Performance comparison for different matrix sizes

The graphical analysis of the results clearly demonstrates that our approach outperforms LAPACK (CPU) and PLASMA (CPU), while achieving performance comparable to cuSolver within a close margin. Additionally, as the matrix size increases, our method continues to maintain its efficiency, indicating that it scales effectively with larger datasets.

4.5 Experiment 4

In this study, we evaluated the performance of our optimized GPU-based method by integrating it into the SLSQP algorithm in the NLOPT library and comparing it with the original SLSQP algorithm. The goal was to assess the scalability and efficiency of our approach in managing large-scale matrices compared to the original sequential implementation. The experiment consists of measuring the execution time to complete the entire optimization process for a constrained non-linear problem. The dimensions of problem involves matrices with sizes: 640, 1250, 1728, and 2240. The results are illustrated in `Figure 6`

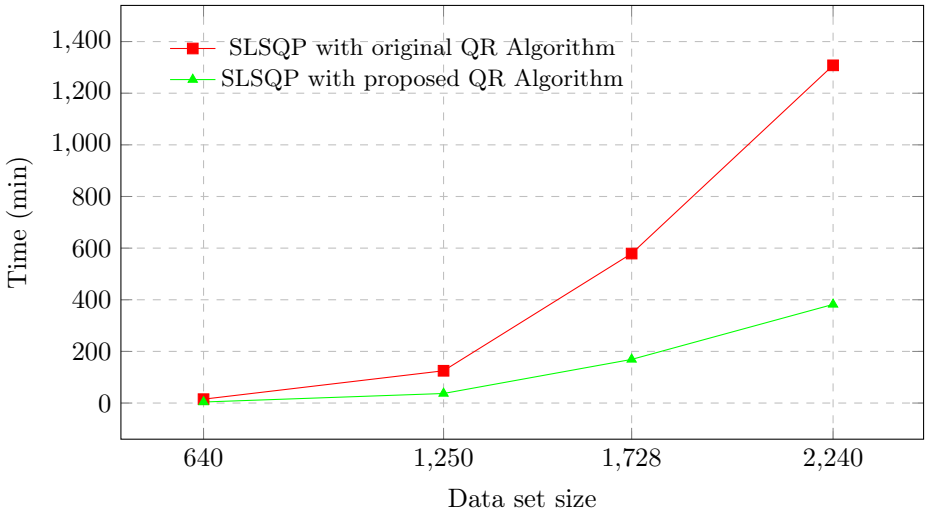

Fig. 6: Performance comparison Between the original SLSQP algorithm and Proposed QR decomposition algorithm on GPU for varying data set sizes

5 Conclusion

Our experimental results show that our approach outperforms LAPACK and PLASMA, and achieves performance comparable to cuSOLVER. Additionally, we demonstrate the effectiveness of our algorithm by integrating it into the SLSQP method in the NLOPT library, where QR factorization is frequently invoked. Replacing the existing QR factorization kernel with our proposed kernel results in significant performance improvements over the original algorithm.

References

1. John T Betts and Paul D Frank. A sparse nonlinear optimization algorithm. *Journal of Optimization Theory and Applications*, 82:519–541, 1994.
2. Paul T Boggs and Jon W Tolle. Sequential quadratic programming. *Acta numerica*, 4:1–51, 1995.
3. Paul T Boggs and Jon W Tolle. Sequential quadratic programming. *Acta numerica*, 4:24–25, 1995.
4. Stephen Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge University Press, Cambridge, 2004.
5. Richard H Byrd, Jorge Nocedal, and Robert B Schnabel. Representations of quasi-newton matrices and their use in limited memory methods. *Mathematical Programming*, 63(1-3):129–156, 1994.
6. NVIDIA Corporation. The api reference guide for cublas, the cuda basic linear algebra subroutine library. https://docs.nvidia.com/cuda/cublas/#introduction, 2024.

7. NVIDIA Corporation. The api reference guide for cusolver, a gpu accelerated library for decompositions and linear system solutions for both dense and sparse matrices. https://docs.nvidia.com/cuda/cusolver/index.html#:~:text=The%20intent%20of,shared%20sparsity%20pattern, 2024.
8. Philip E Gill, Walter Murray, and Michael A Saunders. Snopt: An sqp algorithm for large-scale constrained optimization. *SIAM review*, 47(1):99–131, 2005.
9. G.H. Golub and C.F. Van Loan. *Matrix Computations*. Johns Hopkins Studies in the Mathematical Sciences. Johns Hopkins University Press, 2013.
10. Ali Asghar Heidari, Seyedali Mirjalili, Hossam Faris, Ibrahim Aljarah, Majdi Mafarja, and Huiling Chen. Harris hawks optimization: Algorithm and applications. *Future Generation Computer Systems*, 97:849–872, 2019.
11. J. N. Herskovits and L. A. V. Carvalho. A successive quadratic programming based feasible directions algorithm. In A. Bensoussan and J. L. Lions, editors, *Analysis and Optimization of Systems*, pages 93–101, Berlin, Heidelberg, 1986. Springer Berlin Heidelberg.
12. Steven G. Johnson. The NLopt nonlinear-optimization package. https://github.com/stevengj/nlopt, 2007.
13. Stephen J. Wright Jorge Nocedal. *Numerical Optimization*. Springer New York, NY, 2006.
14. Dieter Kraft. Algorithm 733: TOMP–fortran modules for optimal control calculations. *ACM Transactions on Mathematical Software*, 20:262–281, 1994.
15. Craig T. Lawrence and André L. Tits. A Computationally Efficient Feasible Sequential Quadratic Programming Algorithm. *SIAM Journal on Optimization*, 11(4):1092–1118, 2001.
16. Joaquim R. R. A. Martins and Andrew Ning. *Engineering Design Optimization*. Cambridge University Press, 2021.
17. George Nemhauser and Laurence Wolsey. *Linear Programming*, chapter I.2, pages 27–49. John Wiley & Sons, Ltd, 1988.
18. John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. In *ACM SIGGRAPH 2008 Classes*, SIGGRAPH '08, New York, NY, USA, 2008. Association for Computing Machinery.
19. Jorge Nocedal. Updating quasi-newton matrices with limited storage. *Mathematics of computation*, 35(151):773–782, 1980.
20. Joe Eaton NVIDIA Corporation. Parallel direct solvers with cusolver: Batched qr. https://developer.nvidia.com/blog/parallel-direct-solvers-with-cusolver-batched-qr/#:~:text=cuSOLVER%20provides%20batch,deliver%20decent%20performance., 2015.
21. Florian A. Potra and Stephen J. Wright. Interior-point methods. *Journal of Computational and Applied Mathematics*, 124(1):281–302, 2000. Numerical Analysis 2000. Vol. IV: Optimization and Nonlinear Equations.
22. Andrzej Ruszczyński. *Nonlinear optimization*. Princeton University Press, Princeton, NJ, 2006.
23. Adam Slowik and Halina Kwasnicka. Evolutionary algorithms and their applications to engineering problems. *Neural Computing and Applications*, 32(16):12363–12379, Aug 2020.
24. Wenyu Sun and Ya-Xiang Yuan. *Optimization theory and methods: nonlinear programming*, volume 1. Springer Science & Business Media, 2006.
25. Wikipedia contributors. Thread block (cuda programming) — Wikipedia, the free encyclopedia, 2024. [Online; accessed 24-August-2024].