UC Davis
UC Davis Electronic Theses and Dissertations

Title

Fast Sparse Matrix Reordering on GPU for Cholesky Based Solvers

Permalink
https://escholarship.org/uc/item/3s28q2h2

Author

Yuan, Changcheng

Publication Date

2024

Peer reviewed|Thesis/dissertation

eScholarship.org
Powered by the California Digital Library
University of California

Fast Sparse Matrix Reordering on GPU for Cholesky Based Solvers

By

CHANGCHENG YUAN

THESIS

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

John Owens, Chair

Jason Lowe-Power

Julian Panetta

Committee in Charge

2024

Copyright © 2024 by

Changcheng Yuan

All rights reserved.

To my parents and my grandparents.

CONTENTS

List of Figures v
List of Tables vi
Abstract vii
Acknowledgments viii

1 Introduction **1**
1.1 Overview 1
1.2 Our Contributions 3
1.3 Thesis Organization 4

2 Related Work **5**
2.1 GPU Geometry Processing Data Structures 5
2.2 Matrix Reordering Algorithms 6
2.2.1 Cuthill-McKee and Reverse Cuthill-McKee 6
2.2.2 Nested Dissection 7
2.2.3 Minimum Degree and Approximate Minimum Degree 7
2.2.4 Hybrid and Multilevel Approaches 8
2.3 GPU Graph Partitioning Implementations 9

3 Problem **10**
3.1 Reordering for Cholesky-based Solvers 11
3.2 Mesh Partitioning for Reordering 12
3.3 Our Focus 13

4 GPU-Accelerated Nested Dissection **14**
4.1 Algorithm Design 14
4.1.1 Initialization 15
4.1.2 Stage 1: Reordering for the Coarse Graph of Patches 16
4.1.3 Stage 2: Reordering for Each Patch 18

4.1.4 Stage 3: Combining Reordering Results 19
4.2 Implementation 19
4.2.1 Sparse Matrix and Dense Matrix Compatibility 19
4.2.2 Matlab Implementation 20
4.2.3 CUDA Implementation 22
4.2.4 Summary of Implementation 28

5 Evaluation **29**
5.1 Experimental Setup 30
5.2 Evaluation Results 30

6 Conclusion **40**
6.1 Future Work 41

List of Figures

1.1 Runtime distribution of the cuSolver linear solving process on GPU 2

4.1 A partition tree T for G_c using recursive bisecting k-means clustering. 16

4.2 A partition tree T for G_c with the Recursive Max-Matching approach. 17

4.3 (*nnz*) comparison of Matlab implementations. 21

5.1 Speed up of our method compare to METIS. 31

5.2 (*nnz*) increase of our method and no reorder compare to METIS. 33

5.3 Speedup of our method compared to METIS for the factorization time. 34

5.4 Speedup of our method compared to METIS for the total runtime. 35

5.5 Comparison of memory usage for Cholesky factorization. 36

5.6 Tradeoff between reordering time and the memory sonsumption. 37

5.7 Memory usage for Cholesky factorization with different stage 1 partition level. 38

List of Tables

5.1 Comparison of runtime for different stages with our reorder and with METIS . 39

ABSTRACT

Fast Sparse Matrix Reordering on GPU for Cholesky Based Solvers

Cholesky-based linear solvers are widely employed to solve large sparse positive semi-definite linear systems. Within the reordering, analyzing, factorizing, and solving pipeline, the reordering of sparse matrices is a critical step in reducing non-zero fill-ins, thereby decreasing runtime and memory consumption. However, this stage remains the most time-consuming component of the linear solving process. There is currently a lack of GPU implementations specifically designed for matrix reordering in linear solving. This thesis proposes, to our knowledge, the first GPU-based nested dissection reordering algorithm. Our approach aims to achieve significantly faster reordering times compared to traditional CPU-based methods while maintaining comparable quality in terms of non-zero fill-ins. We have implemented the proposed algorithm and conducted comprehensive performance comparisons with established CPU-based Nested Dissection implementations on various triangle mesh inputs. Our results demonstrate that the GPU-based reordering algorithm can achieve more than a 5 times speedup on average when applied to triangle mesh inputs. However, we produce an average of 6 times more non-zero elements after Cholesky factorization compared to METIS, a widely-used graph partitioning software, based on our tests. Future work focuses on refining our partitioning strategy to achieve better fill-in reduction without sacrificing the significant speed advantages. Finally, we discuss the insights gained from our current implementation and outline future directions for further optimization and analysis.

Acknowledgments

First and foremost, I would like to thank my advisor, Professor John Owens, for his support and guidance throughout my research. His mentorship has been instrumental in shaping my academic path, and I am particularly grateful for his support in attending SIGGRAPH 2023 and GTC 2024.

I extend my sincere thanks to Ahmed Mahmoud, with whom I worked closely over the past two years. His patience, technical expertise, and consistent support were essential in helping me navigate challenges and complete this thesis.

I am also grateful to Serban Porumbescu and Matthew Drescher for their advice on navigating the research process and for the thoughtful discussions that helped shape my research ideas.

Additionally, I would like to thank my committee members, Professor Jason Lowe-Power and Professor Julian Panetta, for their time and feedback. A special acknowledgment to Professor Panetta for sharing his research on linear solvers, which provided valuable insights that inspired this work.

I would also like to thank the faculty at UC Davis for the excellent courses and learning opportunities, as well as my labmates at the Owens lab for their support along the way.

Lastly, I am thankful to my friends, colleagues, and family for their continued support and encouragement throughout this journey.

Chapter 1

Introduction

1.1 Overview

Cholesky-based solvers for solving symmetric positive semi-definite (SPD) systems across scientific computing deciplines like physical simulation, finite element modeling and geometry processing. These solvers are typically the computation bottleneck of the application, especially when running on the CPU. For example, for many geometry processing applications, processing high-resolution triangle meshes with millions of vertices poses significant challenges for CPU-based solvers, while for such applications, GPU-based solvers have shown promising results in terms of running speed. As the complexity and scale of geometric data continue to grow, traditional CPU-based linear solvers face limitations in terms of performance and memory efficiency [8, 17, 21]. We first looked at the state of the art GPU linear solving pipeline with CUDA cuSolver [18] to identify the bottleneck for GPU solvers. The Cholesky linear solving process on GPU, as implemented in cuSolver using CUDA, consists of five main stages: matrix reordering, matrix analysis, memory allocation, Cholesky factorization, and solving. The matrix reordering stage is for reducing the number of non-zero entries, also known as fill-in, increased by the later Cholesky factorization stage. The matrix analysis stage is for analyzing the matrix structure and computing the Video Random Access Memory (VRAM) required for the Cholesky factorization stage and the solving stage. The memory allocation stage is for allocating the required memory for the Cholesky factorization stage and the solving stage. The Cholesky factorization stage is for computing the Cholesky factorization of the input matrix,

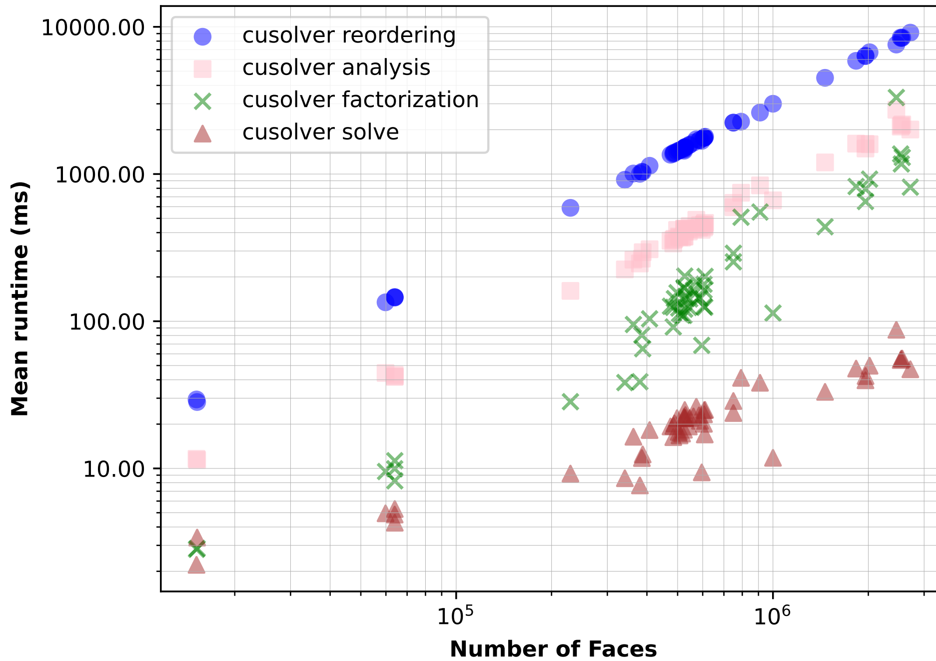

Figure 1.1: Runtime distribution of the four main stages (matrix reordering, matrix analysis, Cholesky factorization, and solving) of the cuSolver linear solving process on GPU with various input triangle mesh sizes. The running time is plotted in log scale.

generating the lower triangular matrix. The solving stage is for solving the linear system with a given right hand side using the lower triangular factor matrix generated by the Cholesky factorization stage. The cuSolver reordering stage is implemented by wrapping the METIS reordering method [11, 18] on the CPU. The sparse matrix is permuted on the CPU side and then copied to the GPU for the rest of the linear solving process. Through our analysis, we found that the matrix reordering stage used the most time for GPU solvers with various input triangle mesh sizes. We use vertex adjacency matrices from triangle meshes as input matrices for our analysis and get the result in Figure 1.1. Two primary factors contribute to the matrix reordering stage being the bottleneck:

- The reordering algorithm used in the matrix reordering stage is slow for using a CPU-based implementation while the rest of the solving pipeline is implemented on GPU.
- There is currently no efficient GPU-based matrix reordering algorithm for Cholesky-based solvers.

The matrix reordering stage is crucial for two reasons. First, it mitigates memory constraints by minimizing fill-in during the Cholesky factorization stage, which is critical given the limited VRAM on GPUs. Second, it substantially reduces factorization time, which constitutes the most computationally intensive phase of the linear solving process. Current state-of-the-art reordering algorithms, such as METIS [11] and SCOTCH [19], are predominantly CPU-based and exhibit suboptimal performance relative to other components of the GPU linear solver.

In addressing this challenge, we reviewed the current matrix reordering algorithms [22] and identified Nested Dissection Reordering as a promising candidate for Cholesky-based solvers and GPU acceleration. Both Nested Dissection and Approximate Minimum Degree [1] are widely used for matrix reordering. However, Nested Dissection is more suitable for our implementation with GPU mesh data structures and yields better results in terms of fill-in reduction and factorization time. Nested Dissection is a divide-and-conquer algorithm that recursively partitions a graph into two subgraphs by removing a separator set of vertices and putting them at the end of the two subgraphs in the reordering sequence. The algorithm continues to partition the two subgraphs until the subgraphs are small enough. Although Nested Dissection and Approximate Minimum Degree use different strategies to generate reordering, the underlying idea is the same: to eliminate the vertices with lowest degree first to reduce fill-in increase during the Cholesky factorization stage.

1.2 Our Contributions

In this paper, we present a novel GPU-based Nested Dissection algorithm and its implementation that leverages the pre-partition strategy employed in RXMesh, a GPU-based geometry processing library to utilize GPU parallelism. Our primary goals include:

1. The first GPU-based Nested Dissection algorithm for matrix reordering that reduces the number of non-zero elements after Cholesky factorization up to 10 times compared with not reordering the input matrix.

2. A fast, GPU-based matrix reordering implementation that is 10 times faster than METIS while maintaining 2 to 12-times increase in the number of non-zero elements after Cholesky factorization compared to METIS.

3. Insights on the trade-off between fill-in reduction and runtime performance, highlighting opportunity for future optimization improvements.

Our GPU-based Nested Dissection algorithm is implemented in CUDA and is designed to be compatible with existing GPU-based linear solvers. We have evaluated our implementation on a variety of input matrices generated from triangle meshes and compared it with METIS. Our results show that our implementation outperforms METIS in terms of speed while maintaining good fill-in reduction after Cholesky factorization, though currently our fill-in reduction performance is still worse than METIS. Through tests and analysis of our implementation, we have gained insights into potentional optimizations to further reduce the fill-in ratio and improve the end-to-end application performance, which we would leave for future work. By addressing the critical bottleneck of GPU Cholesky factorization, our work establishes a foundation for more efficient and scalable GPU-based linear solver implementations. This advancement enables researchers and practitioners to tackle increasingly complex problems with enhanced speed and efficiency using GPU-based solvers.

1.3 Thesis Organization

The remainder of this thesis is organized as follows. In Chapter 2, we review the related work on matrix reordering algorithms and GPU-based linear solvers. In Chapter 3, we present the problem statement and the details of our GPU-based Nested Dissection algorithm. In Chapter 4, we evaluate our implementation and compare it with METIS. In Chapter 5, we discuss the future work and conclude the thesis.

Chapter 2

Related Work

Our work builds upon and extends several areas of research in GPU-based geometry processing, matrix reordering algorithms, and GPU graph partitioning. This section provides an overview of the key developments in these fields that inform our approach. We start with the background on GPU-based geometry processing data structures, as we leverage these tools to implement our GPU-based matrix reordering algorithm. We then discuss the common matrix reordering approaches, algorithms and their implementations, and highlight the close relationship between the matrix reordering and graph partitioning algorithms. Finally, we review the existing GPU graph partitioning implementations that have inspired our work.

2.1 GPU Geometry Processing Data Structures

Recent advancements in GPU-based geometry processing have led to the development of efficient data structures that enable fast parallel computations on triangle meshes for geometry processing tasks. These structures are relevant to our work as they provide efficient ways to represent and manipulate mesh data on GPUs, which is crucial for our matrix reordering algorithm. RXMesh [16] introduced a novel GPU-based mesh data structure that allows for efficient parallel mesh traversal and computation. By employing a pre-computed partition strategy, RXMesh enables fast parallel processing of mesh elements on GPU shared memory. With a novel thread scheduling mechanism, RXMesh achieves high performance for a variety of geometry processing tasks such as mesh smoothing and remeshing. MeshTaichi [23] presented a domain-specific language and compiler for geometry processing on GPUs. Similar to RXMesh,

MeshTaichi first generates the mesh partitioning and generates efficient GPU code for mesh operations. The MeshTaichi compiler automatically optimizes the generated GPU code for parallel execution, enabling researchers to implement complex geometry processing algorithms with high performance.

2.2 Matrix Reordering Algorithms

Matrix reordering algorithms for solving large sparse linear systems with Cholesky based solvers have been extensively studied in numerical linear algebra [10]. These algorithms aim to reduce fill-in and improve the sparsity structure of the matrix, leading to faster and more memory-efficient factorization. Fill-in reduction is crucial as it directly impacts the computational complexity and memory requirements of the factorization process. We review several well-known matrix reordering algorithms and their implementations.

2.2.1 Cuthill-McKee and Reverse Cuthill-McKee

The Cuthill-McKee (CM) algorithm [3] is a refinement of the simple Breadth-First Search (BFS) ordering. Researchers have explored BFS based orderings as simple yet effective reordering strategies, particularly for matrices arising from finite element problems. The basic idea is to traverse the graph representation of the matrix using BFS and number the vertices in the order they are visited. This approach tends to produce banded matrices, which can be beneficial for certain sparse matrix operations. Specifically, the approach starts with a peripheral vertex (a vertex of minimum degree) and uses BFS to number the vertices. At each step, the neighbors of the current vertex are numbered in order of increasing degree. This strategy aims to reduce the bandwidth of the reordered matrix. The Reverse Cuthill-McKee (RCM) algorithm [4] is a variant of CM that simply reverses the ordering produced by CM. Interestingly, RCM often produces better orderings than CM in terms of fill-in reduction during factorization. Liu and Sherman [15] provided theoretical justification for this phenomenon, showing that RCM tends to push vertices with a higher degree to the end of the ordering, which can reduce fill-in. Both CM and RCM have been widely used due to their simplicity and effectiveness, especially for matrices arising from finite element and finite difference discretizations. However, they may not perform as well on more general sparse matrices.

2.2.2 Nested Dissection

Nested Dissection (ND) [11] is a recursive bi-partitioning algorithm that has been widely used for sparse matrix reordering. The basic idea is to divide the graph into two subgraphs by removing a separator set of vertices. ND is then run recursively on each subgraph, and the vertices in the separator are ordered last, which helps to reduce fill-in during factorization. The effectiveness of ND depends critically on finding good separators. For planar graphs and graphs with good separators, ND can provide asymptotically optimal orderings. However, finding optimal separators for general graphs is NP-hard, meaning there is no known algorithm that can find the optimal separator in polynomial time for all possible input graphs. This is due to the combinatorial nature of the problem, where the number of possible separators grows exponentially with the size of the graph. As a result, heuristic methods are often used in practice. Several improvements to the basic ND algorithm have been proposed. For example, multiway ND extends the idea to partitioning the graph into more than two subgraphs at each recursive step. This can lead to better load balancing in parallel implementations. Currently, there is no GPU-based implementation of ND for matrix reordering. Our work aims to fill this gap by developing a GPU-based ND algorithm that leverages the parallel processing capabilities of GPUs.

2.2.3 Minimum Degree and Approximate Minimum Degree

The Minimum Degree (MD) algorithm [6] is a local heuristic method that selects vertices with the lowest degree in the graph for elimination. The rationale is that eliminating low-degree vertices tends to create less fill-in. The algorithm updates the degrees of the remaining vertices after each elimination, which can be computationally expensive for large matrices. The Approximate Minimum Degree (AMD) [1] algorithm improves upon MD by using approximate degrees instead of exact degrees. This significantly reduces the computational complexity while maintaining good quality orderings. AMD has become one of the most widely used reordering algorithms due to its good balance between ordering quality and computational efficiency. Several variants of MD and AMD have been proposed [14], including the Multiple Minimum Degree (MMD) algorithm, which eliminates multiple vertices at each step, and the Constrained AMD algorithm, which can incorporate additional constraints on the ordering.

2.2.4 Hybrid and Multilevel Approaches

More recent work has explored hybrid approaches that combine different reordering strategies, with a focus on multilevel methods. These approaches have led to the development of powerful libraries such as METIS and SCOTCH, which are widely used in scientific computing. METIS [11] is a set of serial programs for partitioning graphs, partitioning finite element meshes, and producing fill-reducing orderings for sparse matrices. It uses a multilevel approach that consists of three phases:

- Coarsening: The graph is progressively coarsened by collapsing vertices and edges.
- Partitioning: A variant of Nested Dissection (ND) is applied to the coarsest graph.
- Uncoarsening: The partitioning is projected back to the original graph, with refinement at each level.

This approach has been shown to produce high-quality orderings for a wide range of matrices, often outperforming traditional single-level algorithms in both quality and speed. SCOTCH [19] is another widely used library for graph and mesh partitioning, static mapping, and sparse matrix ordering. Like METIS, SCOTCH uses a multilevel approach, but it incorporates a wider range of algorithms and heuristics. For matrix reordering, SCOTCH offers several strategies:

- A multilevel implementation of Nested Dissection
- A hereditary multi-section ordering algorithm for irregular matrices
- A multilevel implementation of the Gibbs-Poole-Stockmeyer algorithm for reducing the bandwidth of the sparse matrix.

SCOTCH also provides the ability to combine these strategies, allowing users to tailor the reordering process to their specific needs. Both METIS and SCOTCH have had a significant impact on the field of sparse matrix computations, providing robust, high-quality implementations of advanced reordering algorithms. They are widely used in many scientific computing applications and serve as benchmarks for new reordering algorithms. Another hybrid approach uses ND to obtain a global ordering and then applies Approximate Minimum Degree (AMD)

locally within the subgraphs [20]. This strategy combines the global perspective of ND with the local optimization of AMD, often resulting in improved orderings. Multilevel methods have also been applied more generally to matrix reordering. These methods coarsen the graph to reduce its size, apply a reordering algorithm on the coarsened graph, and then refine the ordering back to the original graph. This approach can be particularly effective for very large matrices, as it allows the application of expensive algorithms on a much smaller problem.

2.3 GPU Graph Partitioning Implementations

While matrix reordering algorithms have traditionally been implemented on CPUs, recent efforts have focused on leveraging GPU capabilities for graph partitioning, which is closely related to matrix reordering. Graph partitioning and matrix reordering are intrinsically linked because the sparsity pattern of a matrix can be interpreted as the adjacent matrix of a graph, where non-zero elements correspond to edges. Partitioning this graph effectively can lead to better matrix orderings, reducing fill-in during factorization. We review several GPU-based graph partitioning implementations that have inspired our work. Jet [7] is a GPU-based graph clustering framework that implements several partitioning algorithms, including a Label Propagation and Louvain method. While not specifically designed for matrix reordering, its techniques for efficient GPU graph processing are relevant to our work. G-kway [13] is a GPU-based graph partitioning library that implements the multilevel k-way partitioning algorithm. It uses a combination of CPU and GPU processing to achieve high-quality partitioning results. The library is designed for large-scale graphs and can handle millions of vertices and edges efficiently. Our work builds upon these foundations, specifically adapting the ND algorithm for efficient execution on GPUs and integrating it with modern GPU-based geometry processing frameworks.

In conclusion, this review of related work highlights the importance of efficient matrix reordering algorithms and the potential for GPU acceleration in this domain. Our research aims to bridge the gap between these established CPU-based methods and the emerging field of GPU-based graph algorithms, with a specific focus on improving the performance of Cholesky-based linear solvers for geometry processing applications.

Chapter 3

Problem

In this chapter, we define the problem of matrix reordering for Cholesky-based linear solvers, with a focus on mesh partitioning as a means to achieve reordering. We begin by explaining key mathematical terms and terminologies used throughout this work. Then we describe the process of Cholesky-based linear solving and the importance of reordering in this process. We then introduce the reordering process and the mesh partitioning problem for reordering. We also discuss the challenges and goals of this work.

Terminology:

- **Vertex-adjacency matrix**: A sparse symmetric matrix $A \in \mathbb{R}^{n \times n}$, where n is the number of vertices in the mesh. $a_{ij} \neq 0$ if vertices i and j are connected by an edge in the mesh, or $i = j$. A is symmetric positive semi-definite for our case.

- **Number of non-zero elements (*nnz*)**: The count of non-zero entries in a sparse matrix. For a sparse matrix A, $\mathit{nnz}(A) = \sum_{i,j} \mathbf{1}(a_{ij} \neq 0)$, where $\mathbf{1}(\cdot)$ is the indicator function.

- **Cholesky factorization**: Decomposition of a symmetric positive-definite matrix A into the product of LL^T, where L is a lower triangular matrix. The factorization process often introduces new non-zero elements, increasing the *nnz* of L compared to the lower triangular part of A.

- **Permutation array**: An array $P = (p_1, \ldots, p_n)$ where each $p_i \in \{1, \ldots, n\}$ and P is a permutation of $(1, \ldots, n)$, which can be transformed into a permutation matrix P_m by

permuting the rows of the Identity matrix aaccordint to P; applying it to a matrix A results in $P_{\mathrm{m}}AP_{\mathrm{m}}^T$, reordering both rows and columns of A.

3.1 Reordering for Cholesky-based Solvers

The problem of reordering for Cholesky-based solvers can be stated as follows. Given a triangle mesh, we want to generate a sparse vertex-adjacent matrix A and with a given right-hand side vector b, we want to solve the linear system $Ax = LL^Tx = b$ using Cholesky factorization. The primary challenge studied in this thesis is to find a permutation array P that minimizes the number of non-zero elements in the Cholesky factor L while maintaining the original structure of the mesh. The quality of the reordering is reflected in the number of non-zero elements in L, as fewer non-zero elements lead to reduced memory requirements and computational complexity of the subsequent factorization and forward, backward substitution operation.

Input:

- A triangle mesh $M = (V, E, F)$, where V is the set of vertices, E is the set of edges, and F is the set of faces.
- A right-hand side vector $b \in \mathbb{R}^n$.

Output: A solution vector $x \in \mathbb{R}^n$ satisfying $Ax = b$, where A is the vertex-adjacent matrix derived from M.

The process of linear solving using reordering involves the following steps:

1. Generate the vertex-adjacent matrix $A \in \mathbb{R}^{n \times n}$ from the input triangle mesh M:

$$a_{ij} = \begin{cases} 1 & \text{if } (v_i, v_j) \in E \text{ or } i = j \\ 0 & \text{otherwise} \end{cases}$$

2. Compute a permutation array $P = (p_1, \ldots, p_n)$, where each $p_i \in \{1, \ldots, n\}$ and P is a permutation of $(1, \ldots, n)$.

3. Apply the permutation P to A and b to obtain the reordered matrix A_P and vector b_P:

$$A_{Pij} = a_{p_i,p_j}, \quad b_{Pi} = b_{p_i}$$

4. Perform Cholesky factorization on A_P to obtain L such that $A_P = LL^T$.

5. Solve the linear system $A_P x_P = b_P$. This involves solving two triangular systems:

$$Ly = b_P$$
$$L^T x_P = y$$

6. Reorder the solution x_P back to the original ordering to obtain x:

$$x_{p_i} = x_{Pi} \quad \text{for } i = 1, \ldots, n$$

The resulting x satisfies the original system $Ax = b$.

3.2 Mesh Partitioning for Reordering

Mesh partitioning is an effective approach to generate reordering for sparse matrices. Both METIS [11] and SCOTCH [19] took advantage of mesh partitioning to generate reordering for sparse matrices. The quality of the partitioning is often measured by:

- The balance of the partitions: $\max_i |P_i| \approx |V|/k$, where k is the number of partitions, P_i represents the i-th partition, and $|V|$ is the total number of vertices.

- The size of the separator set: $|S| \ll |V|$, where S is the set of vertices whose removal will divide the graph into two subgraphs.

- The number of edges cut by the partitioning: $|\{(u, v) \in E : u \in P_i, v \in P_j, i \neq j\}|$, where E is the set of edges, u, v are the two end vertices of an edge.

We want to generate a partitioned mesh M_P that is well-balanced, with a small separator set and few edges cut. Previous work has shown partition with such properties can lead to high-quality reorderings with reduced fill-in during the Cholesky factorization stage.

The problem can be stated as:

Input: A triangle mesh $M = (V, E, F)$.

Output: A permutation array P generated from the partitioned mesh $M_P = (V, E, F, \mathcal{P})$, where $\mathcal{P} = \{P_1, \ldots, P_k\}$ is a partition of V.

The process of mesh partitioning for reordering involves:

1. Generate a partitioned mesh M_P from the input triangle mesh M:

   - Partition M_P into subgraphs, identifying vertex separator set S that divides the graph:
   $$V = V_1 \cup V_2 \cup S, \quad V_1 \cap V_2 = V_1 \cap S = V_2 \cap S = \emptyset$$
   where V_1 and V_2 are the two partitions and S is the separator set.
   - Recursively partition V_1 and V_2 until the subgraphs are small enough.

2. Generate a permutation array P from the partitioned mesh M_P:

   - For each recursion level, assign consecutive indices to vertices in each partition P_i: $p_v = \sum_{j<i} |P_j| + \text{local_index}(v)$ for $v \in P_i$. Assign higher indices to vertices in the separator set: $p_v = |V \setminus S| + \text{separator_index}(v)$ for $v \in S$.
   - Combine the permutation arrays from each recursion level to obtain the final permutation array P.

3.3 Our Focus

In the following chapters, we present our GPU-based approach to partition triangle meshes and generate high-quality reordering for Cholesky-based solvers. We focus on generating the mesh partitioning efficiently with good quality and get the corresponding reordering array leading to reduced fill-in during the Cholesky factorization stage. We aim to achieve a high-quality reordering that reduces the number of non-zero elements in the Cholesky factor while leveraging the parallelism of GPUs to improve performance by using a novel mesh partitioning algorithm that is suitable for GPU acceleration. For our current implementation, we primarily focus on triangle meshes as input. We choose to focus on triangle meshes because they are widely used in geometry processing applications and have a well-defined structure for easier processing. We use the vertex adjacency matrix derived from triangle meshes as input for our reordering algorithm, as it is a common representation for mesh data and can be easily converted to a positive semi-definite sparse matrix for Cholesky factorization. This representation allows us to take advantage of the mesh structure while still working within the framework of sparse matrix operations.

Chapter 4

GPU-Accelerated Nested Dissection

In this chapter, we present the algorithm design and implementation details of our GPU-based approach to matrix reordering for Cholesky-based solvers. We first introduce the algorithm design, general workflow, data structures, and our two-stage reordering approach. We then discuss the implementation details, including modifications to the RXMesh data structure, the Matlab implementation for preliminary results, and the CUDA implementation.

4.1 Algorithm Design

The goal of our algorithm is to generate a nested dissection reordering quickly on the GPU. Due to the recursive nature of the algorithm, it is challenging to simply parallelize the algorithm on the GPU. Our solution uses a divide-and-conquer approach to compute the reordering in two stages. The two-stage approach also allows us to leverage the RXMesh pre-partitioning and the shared memory of the GPU for faster queries.

To utilize the GPU architecture effectively, meshes are usually pre-partitioned into patches to fit into the shared memory of the GPU for faster queries. Our algorithm leverages the pre-partitioned mesh provided by RXMesh to achieve efficient parallel processing on GPUs. The pre-partition is calculated using the GPU based K-means algorithm clustering in RXMesh. Our algorithm could work with any initial k-way partitioning of the mesh, but we use the RXMesh implementation for our case.

We first take the input mesh and let RXMesh partition the mesh into patches using K-means. Then we generate the reordering for the coarse graph of patches and the reordering for each

patch. Finally, we combine the reordering results to get the final reordering array. It is worth noting that the reordering for the coarse graph of patches is a global reordering, and the reordering for each patch is a local reordering. The global reordering is for the entire mesh, and the local reordering is for each patch.

Before delving into the details, let's define some additional notation and terminology:

- $G = (V, E)$: The graph representation of the mesh, where V is the set of vertices and E is the set of edges.
- $\mathcal{P} = P_1, \ldots, P_k$: The set of patches from k-means partition result, where each $P_i \cap P_j = \emptyset, \forall i \neq j, \bigcup_{P_i} P_i \neq G$.
- $G_c = (V_c, E_c)$: The coarse graph where each vertex represents a patch in G, and edges indicated the patches are adjacent in G. The edge weight is the number of edges connecting the two patches.
- T: The partition tree with nodes P_i resulting from recursive bisection.
- $S \subset V$: A vertex separator of G is a subset of vertices whose removal disconnects G into two or more components. Formally, for a partition of $V \setminus S$ into A and B, there are no edges in E between A and B, i.e., $(A \times B) \cap E = \emptyset$.
- π: The final permutation array for reordering that combines the results from the two stages, π_1 and $\pi_{2,i}$. While π_1 is the permutation array for the coarse graph G_c and $\pi_{2,i}$ is the permutation array for each patch P_i.

4.1.1 Initialization

We use the k-means partition to generate the patches for the input mesh. The pre-partitioning process generates a set of patches $\mathcal{P}$ using a parallel K-means algorithm. Each patch P_i is a subgraph including vertices edges and faces from the input mesh. Any two patches P_i and P_j are disjoint, i.e., no vertices, edges, or faces are shared between them (RXMesh assign each vertex, edge, and face to only one patch, but the edges and faces may be adjacent to vertices or edges in other patches). The patches are then used to construct the coarse graph G_c where each

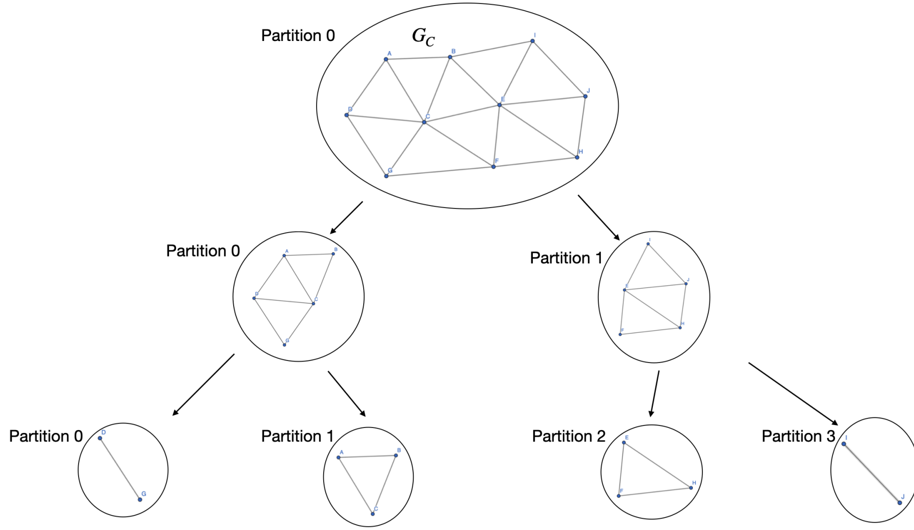

Figure 4.1: A partition tree T for G_c using recursive bisecting k-means clustering.

node in G_c represents a patch and edges are added between nodes if there exists an edge in G connecting vertices from these patches. The edge weight is the number of edges connecting the two patches. The initialization stage is not counted as part of our algorithm, where the patches are generated by the GPU mesh processing libraries like RXMesh for faster mesh processing. We take advantage of this pre-partitioning result to generate the reordering for the entire mesh. This process is not counted as part of our algorithm, hence the timing for the pre-partition is not counted.

4.1.2 Stage 1: Reordering for the Coarse Graph of Patches

In this stage, we work with the coarse graph G_c, where each nodes represents a patch from the k-means partition. We want to generate a multi-level bi-partitioning of G_c to obtain a global reordering for the entire mesh similar to the original Nested Dissection algorithm [11]. For a better explanation, we use partition tree T like Figure 4.1 to represent the multi-level bi-partitioning of G_c. The union of nodes at the same level in the partition tree T forms V_c. The root of the partition tree T represents the un-partitioned graph G_c, and the leaves represent the partitions at the end of the multi-level bi-partitioning.

We explore two methods to achieve this:

1. Recursive Bisecting k-means: We build the partition tree T from root to leaves using

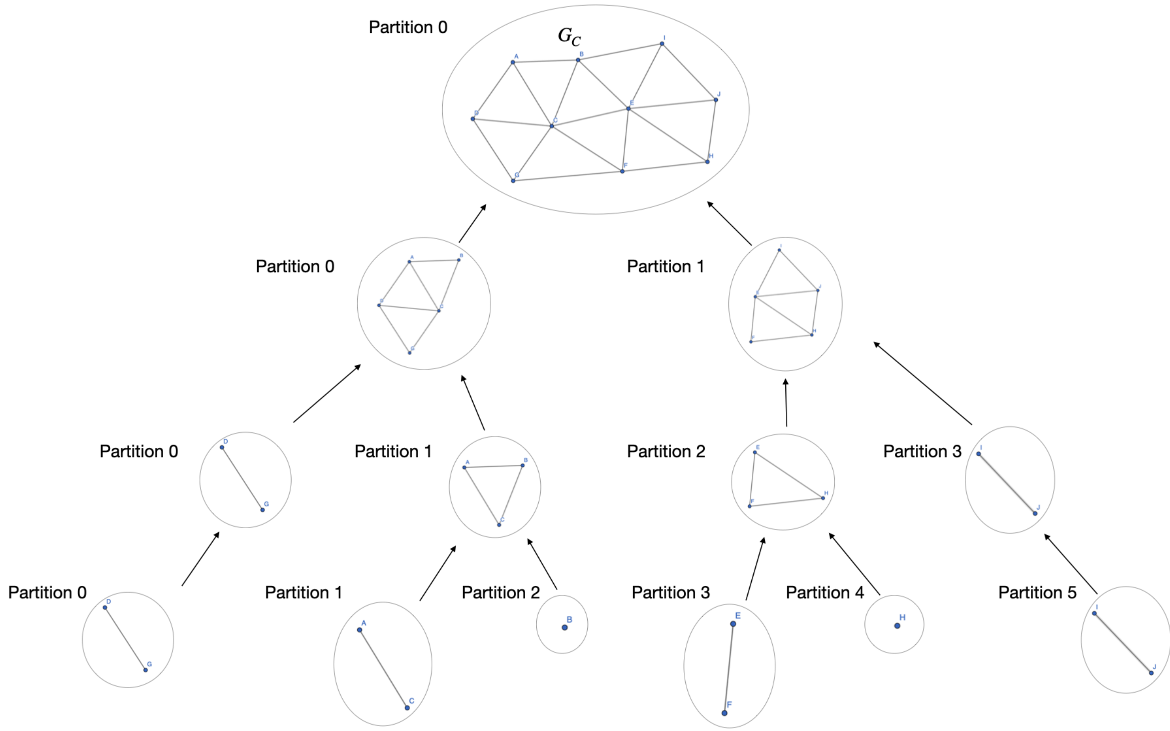

Figure 4.2: A partition tree T for G_c with the Recursive Max-Matching approach.

bisecting k-means clustering, to form the partition tree like Figure 4.1. This method recursively divides the graph into two clusters until we reach the desired number of partitions. Also for each partition in G_c, we find the corresponding vertices in G that separate this partition; and we add these vertices to the vertex separator; then we generate a reordering sequence for the paired partitions in G_c.

2. Recursive Max-Matching approach: We build the partition tree T from leaves to root using maximum matching. In the beginning, we treat every node as a partition. We try to find adjacent pairs of partitions, which we call matched partitions, that could cover as many partitions as possible in the graph. Or, in other words, we try to find the maximum independent set of edges in the graph. When two partitions are matched, we merge them into one partition in the higher level of the partition tree (the reverse of a partitioning process). For every pair of matched partitions in G_c, we find the vertices in G that separate the matched partition in G_c, and we add them to the vertex separator; then we generate a reordering sequence for the paired partitions in G_c. We repeat this Max-Matching process until all the partitions are merged into

one. Therefore, we could get a partition tree T like Figure 4.2. For each partition, we could get a vertex separator S, and two partitions A and B, where we put the vertices in the separator at the end of the two partitions in the reordering sequence.

The resulting partition tree T is then used to generate a permutation array π_1 for the coarse graph. For each bi-partition in the tree, we assign consecutive indices to vertices in each partition and higher indices to vertices in the separator set. We then combine the permutation arrays from each recursion level to obtain the final permutation array π_1.

We decide to use the recursive bisecting k-means approach to generate the reordering for our CUDA implementation. The sequential max matching algorithm's greedy nature makes it ensure the maximal matching ratio, but it is hard to parallelize, the parallel max matching algorithm is hard to achieve maximal matching on the GPU, requiring more iteration hence more running time to build the partition tree. The recursive bisecting k-means approach is more suitable for GPU parallel processing, is easy to implement and takes fewer iterations to generate the partition tree. We discuss the implementation details in the further implementation Section 4.2.

4.1.3 Stage 2: Reordering for Each Patch

In this stage, we focus on reordering the matrix within each patch. We explore two methods:

1. Nested Dissection (ND): We apply the original ND algorithm [11] to each patch independently. The key advantage here is that each patch can fit into shared memory, allowing for parallel processing of patches on the GPU. The algorithm is implemented on the shared memory of the GPU. We allocate the shared memory for each patch, where its size is calculated based on the maximum number of vertices per patch among all patches. We then generate the reordering for each patch independently in parallel.

2. Approximate Minimum Degree (AMD): As an alternative to ND, we could use AMD algorithm [1] for within-patch reordering. AMD is a widely used algorithm for matrix reordering that aims to reduce fill-in during the Cholesky factorization stage. We use a similar approach as ND to generate the reordering for each patch independently in parallel on the shared memory of the GPU.

This stage produces a set of local permutations $\{\pi_{2,i}\}$ for each patch P_i.

We used the Nested Dissection Algorithm for our single patch reordering for now since it's more suitable for sparse matrices arising from geometric processing and finite element problems, and it's easier to implement since we have already have a partition based reordering working on coarse graph G'. We leave the Approximate Minimum Degree approach for future optimization.

4.1.4 Stage 3: Combining Reordering Results

In the final stage, we combine the reordering results from stages 1 and 2 to obtain the final reordering. This involves merging the coarse permutation π_1 with the local permutations $\{\pi_{2,i}\}$ to produce the final permutation array π. In Stage 1, π_1 is used to reorder the patches in G, it assigns all the vertices in a patch the same index in the reordering sequence excluding the separator vertices. The separator vertices are assigned a special index based on the partition tree T. Then in Stage 2, $\pi_{2,i}$ is used to reorder the vertices within each patch, it assigns the vertices in a patch different indices in the reordering sequence $\pi_{2,i}$ for each patch P_i. For each vertex, we simply add the index assigned to it from π_1 and $\pi_{2,i}$ to get the final index in the reordering sequence π.

The resulting π is used to reorder the entire original matrix, completing our two-stage reordering process.

4.2 Implementation

Our implementation starts with modifying the RXMesh data structure to accommodate the linear solving process with cuSolver. Then, we prototype the algorithm in Matlab to verify its correctness and quality. After getting a good fill-in reduction from our Matlab-implemented algorithm compared with METIS, we then implemented the algorithm on the GPU using CUDA and the RXMesh data structure.

4.2.1 Sparse Matrix and Dense Matrix Compatibility

We first modified the RXMesh data structure to generate the sparse matrix A from the prepartitioned mesh. We calculated the prefix sum of the patch sizes sorted by the patch ID to get the unique index for each vertex in the sparse matrix A, then we filled the sparse matrix A in the Compressed Sparse Row (CSR) format. We are able to construct the sparse matrix A as

the adjacency matrix of the input triangle mesh easily because RXMesh provides the adjacency information of the mesh, which is the sparsity pattern of the matrix. We also added storage for the dense matrix b and x for the linear solving process which are stored in the column-major format. After implementing the sparse matrix construction, we verified the bottleneck of the linear solving process with cuSolver and found the matrix reordering stage used the most time for GPU solvers across various input triangle mesh sizes. Then we implemented the reordering algorithm on the CPU using Matlab to verify the correctness and obtain initial results for a number of non-zero elements (*nnz*) after Cholesky factorization. The initialization stage from the algorithm is provided by RXMesh and we use the RXMesh data structure to implement the following stages of the algorithm. RXMesh provides the pre-partitioned mesh with the K-means algorithm. The mesh is partitioned into patches P_i. We take the patches as the input for the reordering algorithm. We use the same notation as the algorithm design to describe the data structure in the following sections.

4.2.2 Matlab Implementation

To make sure our algorithm is correct and the quality of the reordering is good, we implemented our nested dissection approach with RXMesh pre-partitioning result on CPU using Matlab. We began by implementing the Modified Generalized Nested Dissection Algorithm (MGND) [12], which offers a straightforward implementation and provides valuable insights into the quality of our reordering algorithm.

The core principle of MGND is to first obtain a k-way partitioning and then extract the vertex separator. A vertex separator is a set of vertices whose removal disconnects the graph into two or more components. These separator vertices are then placed at the end of the remaining vertices, which are ordered by patches. As illustrated in Figure 4.3, our MGND implementation demonstrates great fill-in reduction compared to the unordered case, by reducing the *nnz* by 2 to 5 times. The y-axis of the figure is plotted in log scale and the naive MGND reordering algorithm label in the figure represents our naive implementation on Matlab. This is the foundation of our GPU-based Nested Dissection algorithm because essentially, we are adding Stage 1 and Stage 2 to the MGND algorithm to construct the final reordering.

Encouraged by these results, we proceeded to implement our full nested dissection algo-

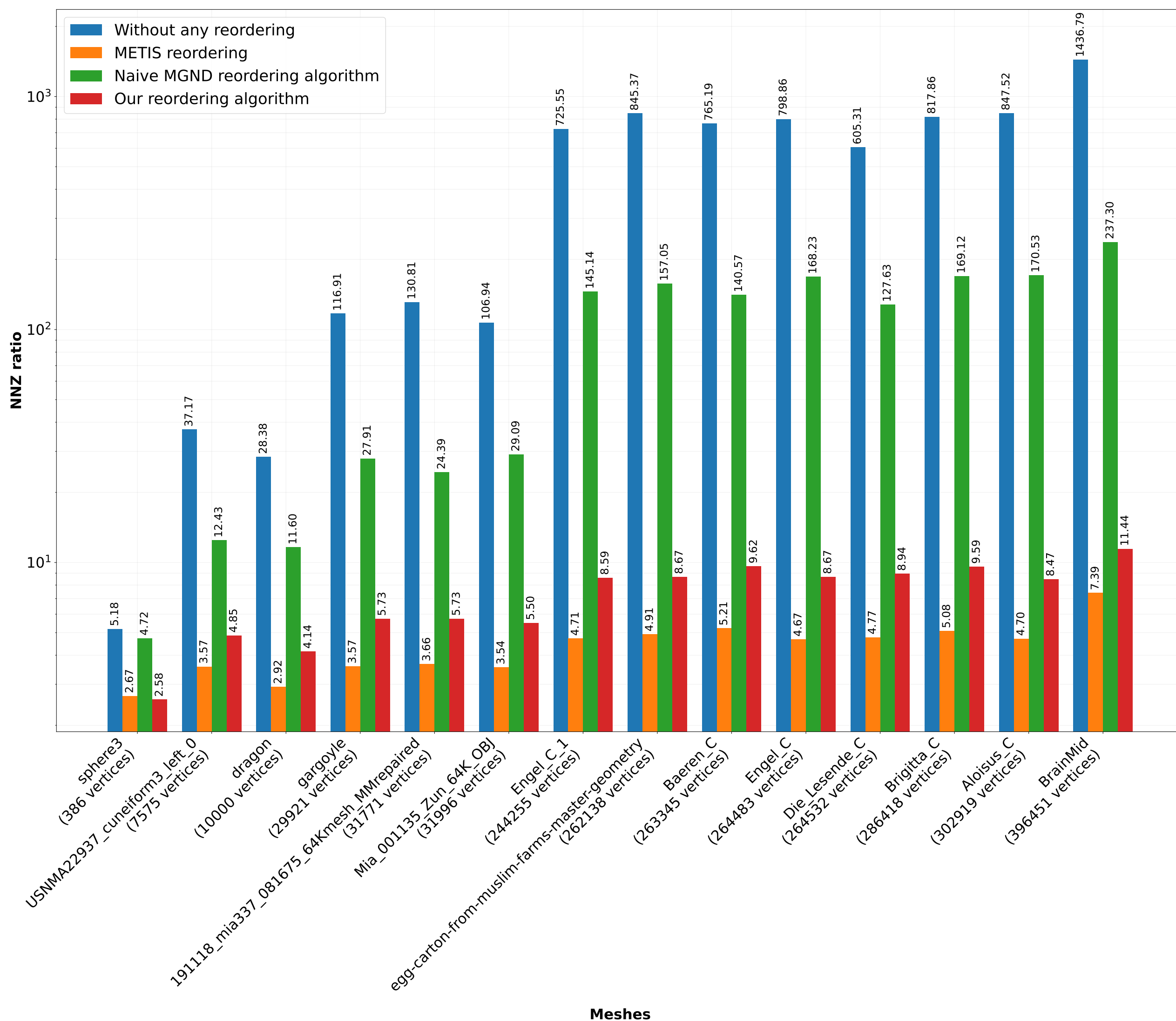

Figure 4.3: The comparison of the number of non-zero elements (*nnz*) after Cholesky factorization between the unordered case, MGND implementation, METIS, and our method in Matlab. The fill-in ratio is calculated as the number of non-zero elements after Cholesky factorization divided by the number of non-zero elements in the original matrix.

rithm in Matlab. We utilized the same RXMesh pre-partitioning result and employed a Recursive Max-Matching approach for Stage 1, followed by Matlab's built-in Nested Dissection implementation (which uses METIS underneath) for Stage 2. We use the Recursive Max-Matching approach because the greedy max-matching algorithm is easy to implement sequentially on Matlab. And we use the Nested Dissection implementation because it's one of Matlab's built-in methods.

For the Recursive Max-Matching in Stage 1, we adopted a greedy approach. Initially, we sorted the edges in G_c based on the number of edges connecting two patches. We then iteratively selected the edge with the highest number of connections between unselected patches, collapsing the selected edges into a single vertex. This process was repeated until all vertices were merged into one. We then used the resulting partition tree to generate the permutation array π_1. For Stage 2, we directly get the permutation array $\pi_{2,i}$ for each patch P_i from METIS. We then combine the results from both stages with prefix sum to assign the unique index for each vertex to get the final permutation array π.

We implemented the reordering algorithm sequentially on the CPU to verify its correctness and quality. As shown in Figure 4.3, our Matlab implementation achieves fill-in reduction comparable to METIS. Our Matlab implementation produces 1.5 to 2 times more *nnz* compare to METIS. This result validates the effectiveness of our approach.

4.2.3 CUDA Implementation

For the CUDA implementation of our reordering algorithm, we first directly work with the meshes to generate the nested dissection reordering on the GPU and also get the reordering array. Then we use the reordering array to reorder the sparse matrix A to make the following linear solving process faster and more memory efficient. We use the RXMesh data structure to implement the following stages of the algorithm.

4.2.3.1 Initialization

The initialization stage is provided by RXMesh, where the input mesh is pre-partitioned into patches. The coarse graph G_c is constructed from the patches and stored in adjacency list format on GPU global memory (VRAM). The sparse matrix A is generated from the mesh and stored in CSR format. The dense matrices b and x are initialized for the linear solving process.

4.2.3.2 Stage 1: Reordering for the Coarse Graph of Patches

We followed the same two-stage approach as in the Matlab implementation. We first implemented the Recursive Max-Matching approach for Stage 1 on the GPU. To achieve this, we first implemented the parallel algorithm to find the maximum independent set of edges to implement the Max-Matching approach. We choose to concatenate the edge ID before the edge weight using the bit shift operation to generate the hashing for the edges during the max-matching algorithm. The details of parallel max matching are shown in Algorithm 1.

Algorithm 1 Parallel Max-Matching Algorithm

Input: $G_c = (V_c, E_c)$: The coarse graph

Local: $active_e$: The active edges array where $active_e[i] = 1$ if the edge e_i is active, and 0 otherwise

Output: $matched_e$: The matching array of edges where $matched_e[i] = 1$ if the edge e_i is matched, and 0 otherwise

1: Initialize$\forall e_i \in E_c, active_e[i] = 1$
2: Initialize $\forall e_i \in E_c, matched_e[i] = 0$
3: **while** $\exists e_i \in E_c, active_e[i] = 1$ **do**
4:     **for do**$e_i \in E_c$, $active_e[i] = 1$ **in parallel**
5:         For edge e_i and all the edges e_j adjacent to the two end vertices of e_i, if $hash(e_i)$ is greater than all the $hash(e_j)$, then $matched_e[i] = 1$, $active_e[i] = 0$
6:         For all the e_j, $active_e[j] = 0$
7:     **end for**
8: **end while**

After getting the matched edges, we generate the vertex separator S from the original graph vertices in G that separates the matched partitions in G_c. We then generate the reordering sequence for the paired partitions in G_c. Then we collapse the matched edges, which merge the two partitions into one partition in the higher level of the partition tree. The parallel edge collapsing process is shown as below in Algorithm 2. We repeat the matching and collapsing process until all the partitions are merged into one. Therefore, we get a partition tree T like Figure 4.2.

Algorithm 2 Parallel Edge Collapsing Algorithm

Input: $G_c = (V_c, E_c)$: The coarse graph
Input: $matched_e$: The matching array of edges
Output: G'_c: The updated coarse graph after edge collapsing

1: Initialize $G'_c = \emptyset$
2: **for** $e_i \in E_c$ in parallel **do**
3: **if** $matched_e[i] = 1$ **then**
4: Mark the partition with lower partition index of the two end partitions of e_i as the remaining partition. Mark the partition with higher partition index to be deleted.
5: **end if**
6: **end for**
7: — Global Synchronization —
8: **for** $e_i \in E_c$ in parallel **do**
9: **if** $matched_e[i] = 1$ **then**
10: Mark edge e_i to be deleted.
11: **else**
12: **if** the two end partitions of e_i are to be deleted **then**
13: Mark edge e_i to be deleted.
14: **else if** the two end partitions of e_i are marked as the remaining partition **then**
15: Add edge e_i to G'_c with atomic increment edge ID in G'_c.
16: **else if** One end partition of e_i is marked as the remaining partition and the other end partition is marked as to be deleted **then**
17: Mark edge e_i to be deleted.
18: **else if** One end partition of e_i is not marked and the other end partition is marked as the remaining partition or not marked **then**
19: Add edge e_i to G'_c with atomic increment edge ID in G'_c.
20: **else if** One end partition of e_i is not marked and the other end partition is marked as to be deleted **then**
21: Create a new edge in G'_c with the partition ID of the not-marked partition and the partition ID of the remaining partition that the to-be deleted partition is going to be merged into.
22: **end if**
23: **end if**
24: **end for**

As we tested the algorithm on the GPU, we found that the algorithm was not as efficient as we expected. On the CPU running sequentially, the greedy algorithm guarantees the maximal matching in one iteration, the parallel matching algorithm on the GPU does not guarantee maximal matching on one iteration. Therefore, we need to run high iteration in the matching algorithm to get near-maximal matching. If we choose to proceed with sub-maximal matching,

the partition tree would be deeper, causing more time consumption due to increased iteration or increased recursion depth to build the partition tree. Other works suggest that a sophisticated algorithm to select the edges to match is needed to get a maximal matching [5]. Since it takes too many iterations to get a maximal matching, and too much effort to implement the edge selection algorithm, we decided to use the Recursive Bisecting k-means approach for Stage 1.

For bisecting k-means, we recursively divide the graph into two clusters until we reach the desired number of partitions or recursion depth. The partition tree T is constructed from root to leaves like Figure 4.1. We then generate the permutation array π_1 for the coarse graph G_c using the partition tree T. For each partition step, identify the vertices from G that separate the partitions in G_c and generate the reordering sequence. Each partition process generates a vertex separator S from G and two partitions A and B from the coarse graph G_c. We put the vertices in the separator at the end of the two partitions in the reordering sequence. Combining the results from each recursion level, we obtain the final permutation array π_1.

The detail of the bisecting k-means algorithm is shown in Algorithm 3. Although we run the k-means algorithm with k seeds, the coundary of the existing partition is not changed. For each existing partition, we only keep the two seeds for vertex assignment within the same partition. In this way, we could run the bisecting k-means algorithm in parallel with only launching one kernel on GPU.

For each bisecting k-means process, we generate the reordering sequence. Similar to the max-matching approach, we generate the vertex separator S from G that separates the two partitions in G_c. We then generate the reordering sequence which puts the vertices in the separator S at the end of the vertices in the two partitions in the reordering sequence. We repeat the bisecting k-means process until we reach the desired number of partitions or recursion depth. We then combine the results from each recursion level to obtain the final permutation array π_1. We took this approach for the evaluation of the algorithm on the GPU in the end because it is more efficient in constructing a partition tree T and generating the reordering sequence for the coarse graph G_c.

Algorithm 3 Bisecting k-means Algorithm

Input: $G_c = (V_c, E_c)$: The coarse graph

Input: $num_partitions$: The number of partitions

Input: $patch_partition$: The partition array for the coarse graph G_c. $patch_partition[i]$ indicates the partition index for vertex v_i in G_c.

Input: $num_partitions'$: The number of partitions after bisecting k-means

Output: $patch_partition'$: The partition array for the coarse graph G_c after bisecting k-means. Initialized as $INVALID$ for all vertices.

1: Initialize $num_seeds = 2 \times num_partitions$
2: Initialize $seeds = \emptyset$
3: Randomly select 2 seeds from each partition in G_c and add them to $seeds$.
4: Set seed partition index in $patch_partition'$ for each seed in $seeds$.
5: Initialize $kmeans_iter = 10$
6: **for do** $i = 1$ to $kmeans_iter$
7:     **while** $\exists v_i, patch_partition'[i] = INVALID$ **do**
8:         **for do**$v_i \in V_c$ **in parallel**
9:             Assign v_i to the nearest seed in $seeds$. The distance is measured by the shortest number of hops between v_i and the seed.
10:             $patch_partition'[i] = patch_partition[seed_index]$
11:         **end for**
12:     **end while**
13:     Set all vertices as active.
14:     **while do** there are still more than num_seeds vertices that is active
15:         **for do**$v_i \in V_c$ **in parallel**
16:             Deactivate the vertex v_i if it is adjacent to the vertices that are assigned to different seeds within the same partition or is adjacent to deactivated vertices within the same partition.
17:             The last two active vertices that are assigned to different seeds and in the same partition are selected as new seeds.
18:         **end for**
19:     **end while**
20: **end for**
21: **for** $v_i \in V_c$ **in parallel do**
22:     Assign v_i to the nearest seed in $seeds$. The distance is measured by the shortest number of hops between v_i and the seed.
23:     $patch_partition'[i] = patch_partition[seed_index]$
24: **end for**
25: $num_partitions' = num_seeds$.

4.2.3.3 Stage 2: Reordering for Each Patch

For Stage 2, we implemented a simple Nested Dissection algorithm on the GPU as the original Nested Dissection algorithm [11]. We implemented the algorithm for reordering each patch. Because the patches are stored in the shared memory of the GPU, we also implemented the nested dissection algorithm with cooperative groups working with the shared memory. The vertices in the patch that are marked as the vertex separator S in Stage 1 are excluded from the patch at this stage. The algorithm details are shown in Algorithm 4. The patch size of RXMesh is set to less than or equal to 512 vertices so that the patch can fit into the shared memory of the GPU. We generate the reordering sequence for each patch in parallel independently. The nested dissection algorithm first generates a coarse graph with recursive coarsening for the patch, then generates a multi-level bi-partitioning of the coarse graph, and finally projects the multi-level bi-partitioning to G. At last, we generate the reordering sequence for the patch. Since a single patch is already small, we recursively coarsen the graph 3 times until it is small enough for the multi-level bi-partitioning. We set the multi-level bi-partitioning depth to 1, generating two partitions for the patch. We will experiment more on the coarsening recursion depth and the multi-level bi-partitioning depth in the future as we are working on some programming bugs with the current implementation.

Algorithm 4 Shared Memory Nested Dissection Algorithm

Input: P_i: The patch $P_i = (V_{P_i}, E_{P_i})$ that already exists on the shared memory of the GPU
Output: $\pi_{2,i}$: The permutation array for the patch P_i

1: Initialize $coarse_level = 3$
2: Initialize $partition_level = 1$
3: Initialize $P_i^0 = P_i$
4: **for do** $i = 1$ to $coarse_level$
5: Coarsen the graph P_i^{n-1} with max matching algorithm to get the coarsen graph P_i^{n+1}
6: **end for**
7: Partition the coarsened graph $P_i^{coarse_level}$ with bisecting k-means algorithm for 1 level to get two partitions. (Call algorithm 3 once)
8: **for do** $i = coarse_level$ to 0
9: Project the partition result of the coarsen graph P_i^n to P_i^{n-1}.
10: **end for**
11: Generate the reordering sequence $\pi_{2,i}$ for the patch P_i with the partition result on P_i^0.

We have not implemented the Approximate Minimum Degree (AMD) algorithm on the GPU

yet. We will implement the AMD algorithm on the GPU in the future to compare the reordering quality and execution time with the Nested Dissection algorithm.

4.2.3.4 Stage 3: Combine reordering results

In the final stage, we combine the reordering results from stages 1 and 2 to obtain the final reordering. We first get the reordering array for the coarse graph G_c from Stage 1, which is the patch reordering array. Based on the Stage 1 result, we get a reordering π_1 to reorder patches and vertex separators that separate the patches. The vertices that are not vertex separators are the vertices in the patches that are not reordered yet, and they are assigned random consecutive reordering indices within the patch at this stage. Then we get the reordering array $\pi_{2,i}$ for each patch from Stage 2. This reordering array reorders the vertices in the patch that are not vertex separators from Stage 1. We then combine the reordering array π_1 and $\pi_{2,i}$ to get the final reordering array π.

4.2.4 Summary of Implementation

We evaluate our implementation with the RXMesh data structure and the CUDA implementation in the stages listed above. The initialization stage is provided by RXMesh, so the timing and quality of the pre-partitioning result are not counted in the evaluation. We use the recursive bisecting k-means approach for Stage 1 and the shared memory nested dissection algorithm for Stage 2 for our final set up for tests. We evaluate the quality of the reordering and the execution time of the algorithm on the GPU for our implementation.

Chapter 5

Evaluation

We evaluate our GPU-based Nested Dissection algorithm on a variety of input meshes and compare it with METIS. METIS is a CPU-based widely used graph partitioning library that provides reordering functionalities for linear solving tasks. We use RXMesh for the initialization of the input triangle mesh including the pre-partitioning of the mesh. The time for initialization of the input triangle mesh is not included in the evaluation. The CUDA cuSolver library is used for the Cholesky factorization and solving tasks. We evaluate the quality of the reordering by measuring the number of fill-in ratio and the time/memory usage of the Cholesky factorization from cuSolver. We also evaluate the runtime performance of our algorithm compared to METIS. We first present the experimental setup, followed by the evaluation results.

Before diving into the results, it is essential to discuss the fundamental tradeoff that our GPU-based Nested Dissection algorithm introduces, as this is a key outcome of the work. Specifically, the tradeoff lies between reordering time and the memory consumption required for Cholesky factorization. Our algorithm is designed to reduce reordering time, offering a notable speedup over the CPU-based METIS implementation. However, this comes at the cost of increasing the number of non-zero elements (*nnz*) during the Cholesky factorization, which in turn increases memory usage and slows down the factorization process. This tradeoff is particularly important because while the faster reordering is beneficial, the increased memory consumption and longer factorization time limit the overall end-to-end performance gains. Understanding this balance is critical before examining the detailed performance metrics, as it frames the limitations and potential optimizations for future work.

5.1 Experimental Setup

We tested our GPU-based Nested Dissection algorithm on a variety of input meshes from Common 3D Test Models [9] and Thingi10K [24]. We compared our results with METIS [11] to evaluate the quality of the reordering and the running time of the algorithm. It worth noting that the METIS implementation that we are testing against is from the cuSolver wrapper of METIS [18]. It uses 64-bit metis-5.1.0 and called the METIS_NodeND function, which is implemented on CPU side.

We run all experiments on a machine with an AMD EPYC 7402 24–Core Processor and an NVIDIA Tesla V100–SXM2–32GB GPU. The machine runs Ubuntu 22.04.2 LTS, Slurm 22.05.8 and CUDA Version: 12.4.

We measure the runtime of METIS using the `std::chrono` library in C++. The timer is wrapped around the cuSolver function `cusolverSpXcsrmetisndHost` calling `METIS_NodeND`. We measure the runtime of our GPU-based Nested Dissection algorithm using `cudaEvent` from CUDA. The timer is wrapped around the main body of the algorithm excluding the initial memory allocation and the final memory deallocation.

The sparse matrix is stored in the Compressed Sparse Row (CSR) format, and its sparsity pattern is generated by adding the identity matrix to the adjacency matrix of the input triangle mesh. The numerical values in the sparse matrix and the right-hand side vector are generated with the Mean Curvature Flow (MCF) calculation [2]. The matrix analysis, memory allocation, Cholesky factorization, and solving are done with the cuSolver library on GPU. The matrix reordering is done with METIS on CPU and our GPU-based Nested Dissection algorithm on GPU. With our GPU-based Nested Dissection, the whole linear solving process could be done on GPU.

5.2 Evaluation Results

We first evaluate the runtime of our GPU-based Nested Dissection algorithm compared to CPU-based METIS. Note that we are using a GPU based linear solving pipeline, hence the prepartition time is not included in the evaluation and both our method and METIS are tested with the prepartitioned mesh. From Figure 5.1, we could see that our reorder method is faster than

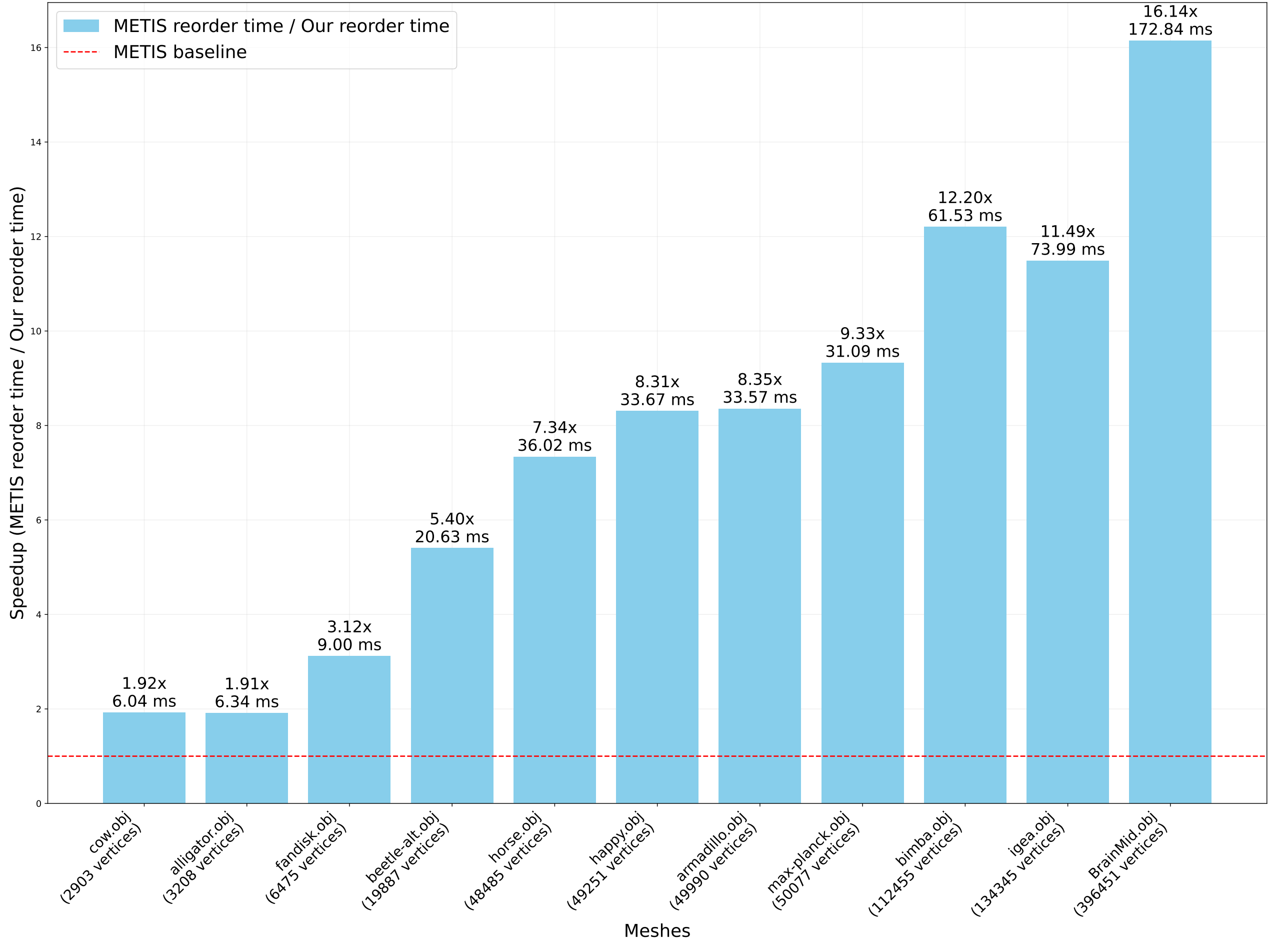

Figure 5.1: The Speedup of our method compare to METIS for reordering stage. The original running time of our method in milliseconds is also shown on the top of the bars. The speedup indicates the ratio of the METIS running time to our method running time. The speedup is greater than 1 if our method is faster than METIS. Our method is overall faster than METIS with a geometric mean of 6.34x

METIS for all tested inputs by an average of 6.23 times. The largest speedup is 16.1 times for the BrainMid.obj input. We expect the speedup to be larger for larger graphs as the reordering time is more significant for larger graphs.

The we look at the quality of the reordering by measuring the fill-in ratio after Cholesky factorization. The fill-in ratio is calculated as the number of non-zero elements in the lower triangular matrix after Cholesky factorization divided by the number of non-zero elements in the original matrix. Because the CUDA cuSolver library's wrapper does not provide a direct way to get the number of non-zero elements in the original matrix, we use the METIS_NodeND

function directly from including the METIS library to get the number of non-zero elements in the original matrix and the matrix after factorization. Note that for consistency, we only use METIS_NodeND from METIS library for the fill-in ratio calculation. All other experiments are done with the METIS wrapper from cuSolver.

The result is shown in Figure 5.2. Our algorithm generates more *nnz* than METIS by an average of 5.14 times. The largest result is 7.96 times more *nnz* for the BrainMid.obj input. The result indicates that our algorithm generates more fill-in ratio than METIS, which is expected as our focus is to reduce the reordering time, while getting an increase in *nnz*. This also indicates that the reordering quality is the current bottleneck of our algorithm. The main goal for our future work is to reduce the fill-in ratio with a better partitioning strategy.

Since our algorithm generates more *nnz* than METIS, the factorization time and memory usage are expected to be longer and larger for the Cholesky based linear solving process, respectively. We evaluate the runtime performance of the sum of the matrix analysis, memory allocation, Cholesky factorization, and solving stages after reordering and the total runtime performance of our GPU-based Nested Dissection algorithm compared to METIS.

Figure 5.3, the run time of the matrix analysis, memory allocation, Cholesky factorization, and solving stages with our reorder method is slower than METIS for all tested inputs by an average of 0.22 times. The largest slowdown is 0.03 times for the BrainMid.obj input. The result shows that our GPU-based Nested Dissection algorithm is slower than METIS in terms of Cholesky factorization time. The gap is larger for larger graphs as the factorization time is greatly influenced by the fill-in ratio. Our experiments have shown that we are trying to trade off the reordering time with the fill-in ratio, which leads to a longer factorization time.

Then we look at the total runtime performance of our GPU-based Nested Dissection algorithm compared to METIS to figure out whether the trade-off is worth it. The result is shown in Figure 5.4. The total runtime is the sum of reordering time and Cholesky factorization time. Our algorithm is slower than METIS for all tested inputs by an average of 0.61 times. The largest slowdown is 0.10 times for the BrainMid.obj input. Although our method is faster in terms of reordering time, it does not make up for the longer factorization time. The larger fill-in ratio leads to longer factorization time and larger memory usage, which results in a longer total

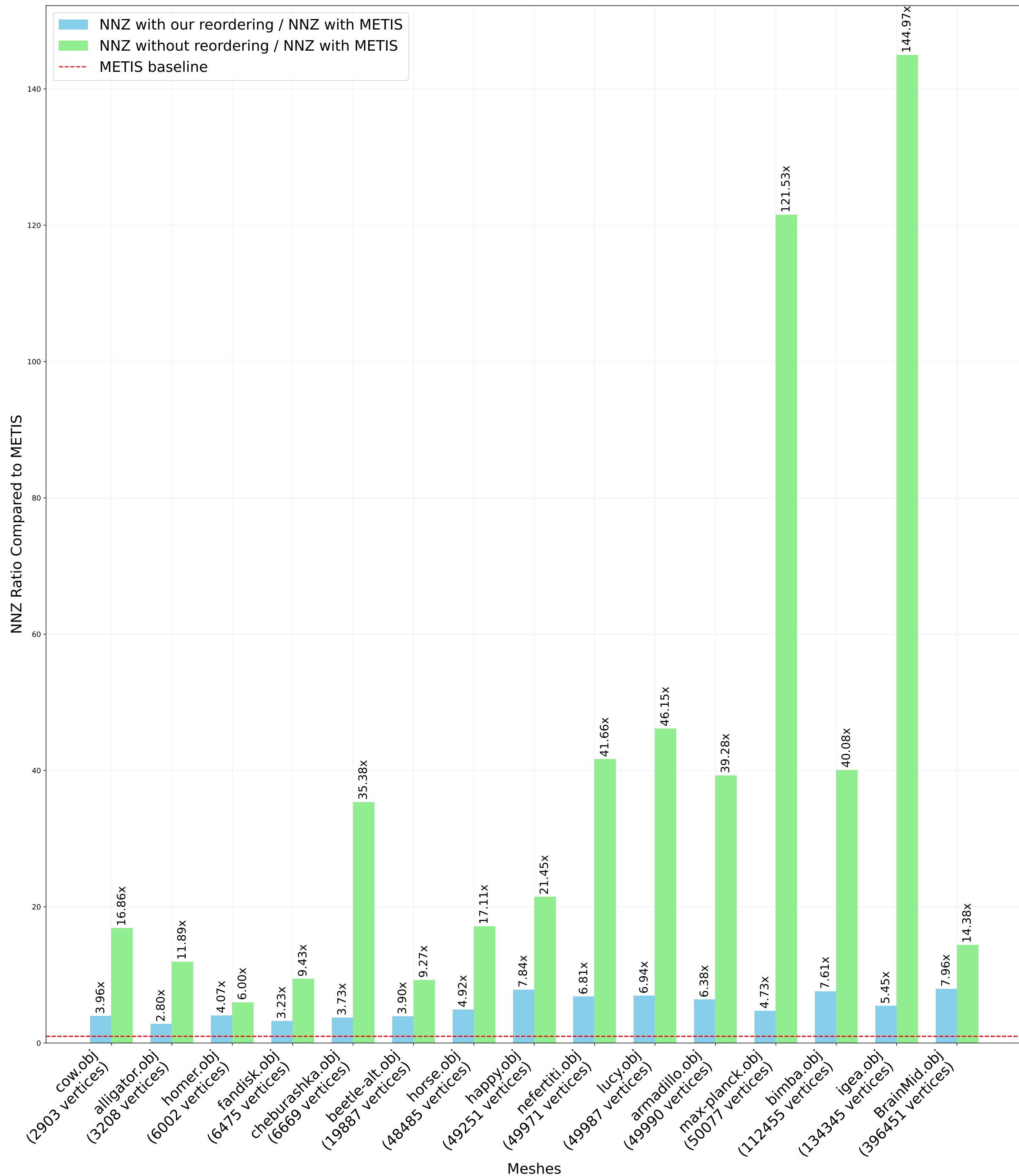

Figure 5.2: Illusterating the *nnz* increase of our method and no reorder used compared with using METIS reorder. The METIS reorder result is directly from METIS_NodeND function from METIS library, while all other experiments are done with the METIS wrapper from cuSolver. Our method generates 5.08x more *nnz* than METIS on average.

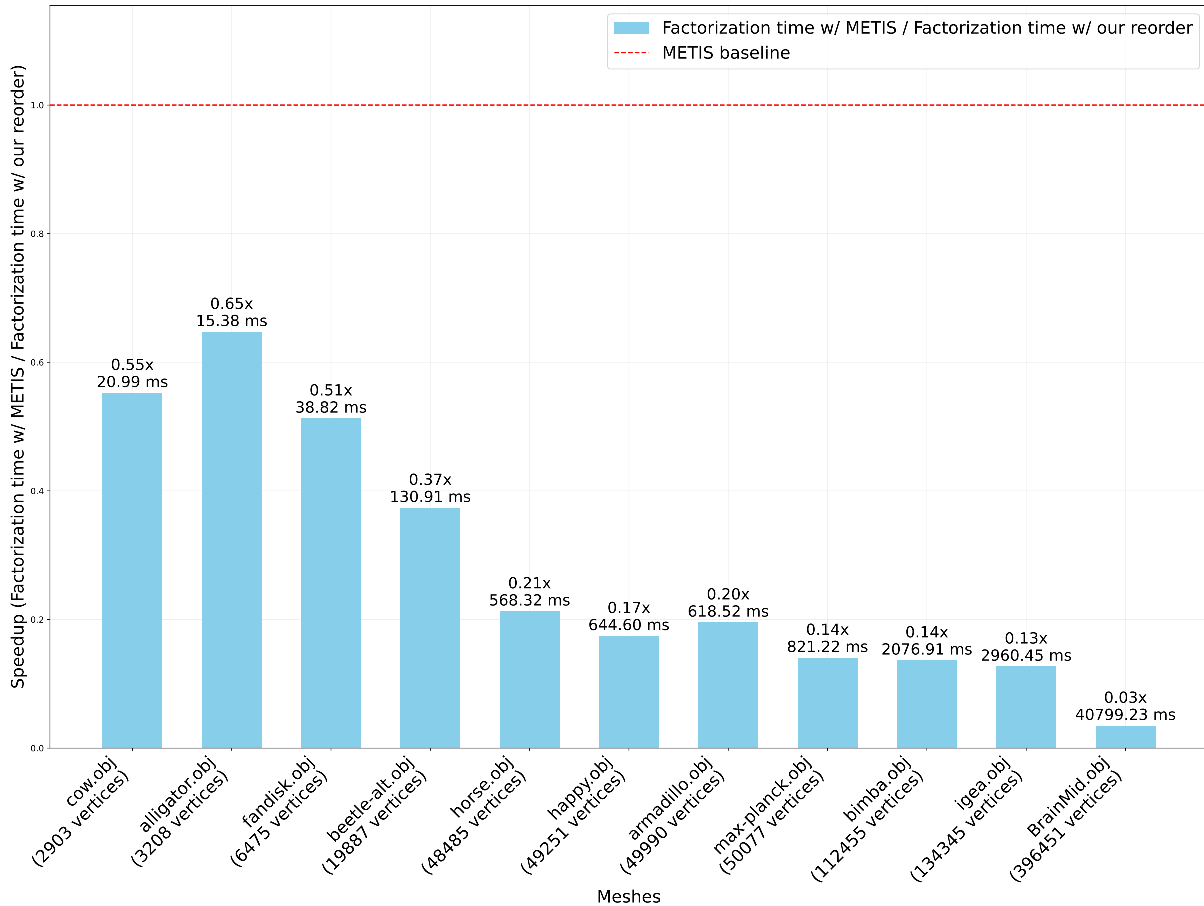

Figure 5.3: Speedup of our method compared to METIS for the factorization time. The factorization time represents the sum of runtime from the matrix analysis, memory allocation, Cholesky factorization, and solving stages. The time for reorder is not counted in this figure. The original runtime in milliseconds is also shown on the top of the bars. The speedup is less than 1 if our method is slower than METIS. Our method is slower than METIS for all tested inputs by an average of 0.21 times.

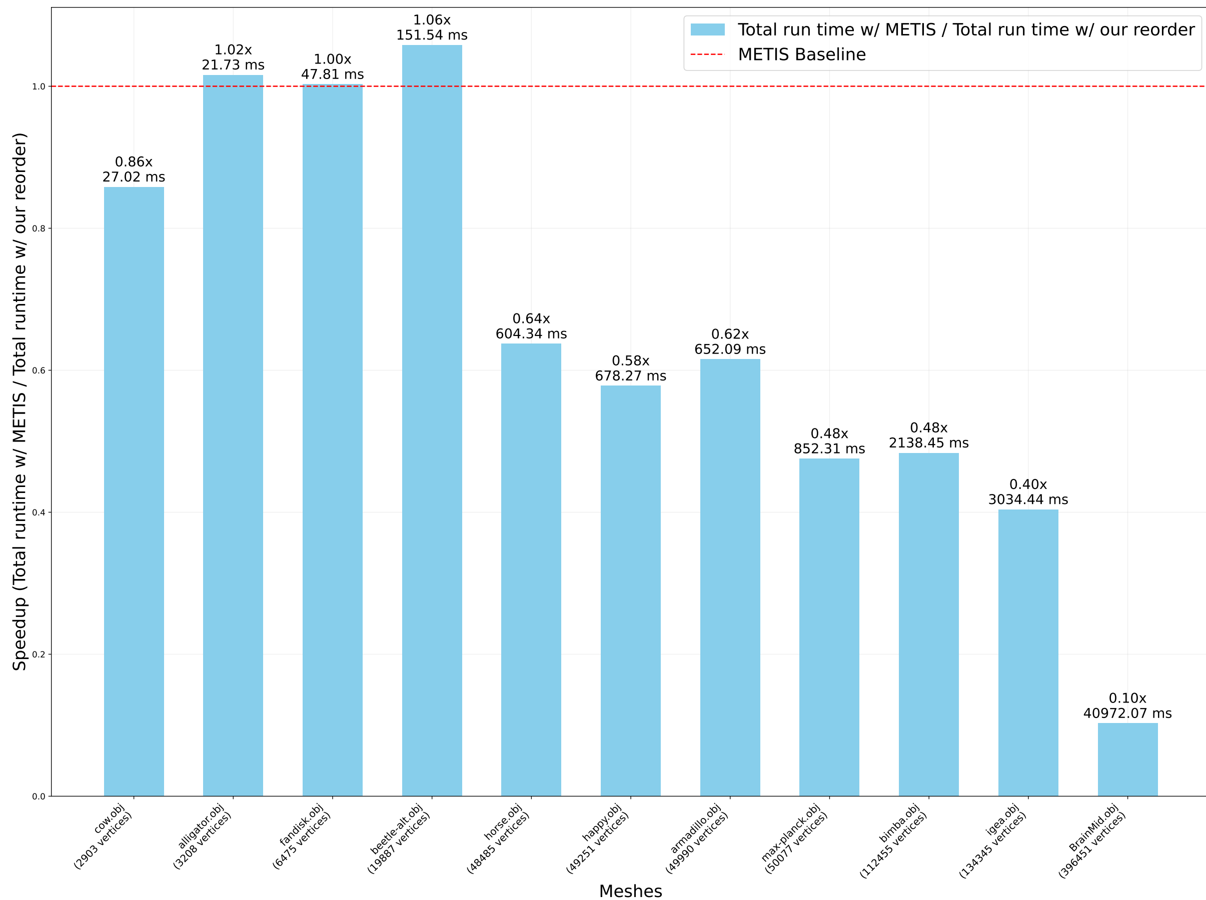

Figure 5.4: Speedup of our method compared to METIS for the total runtime including reorder time. The total runtime represents the sum of the reorder time the matrix analysis, memory allocation, Cholesky factorization, and solving stages. The speedup is calculated as the total runtime of METIS divided by the total runtime of our method. The speedup is less than 1 if our method is slower than METIS. Our method is slower than METIS for all tested inputs by an average of 0.51 times

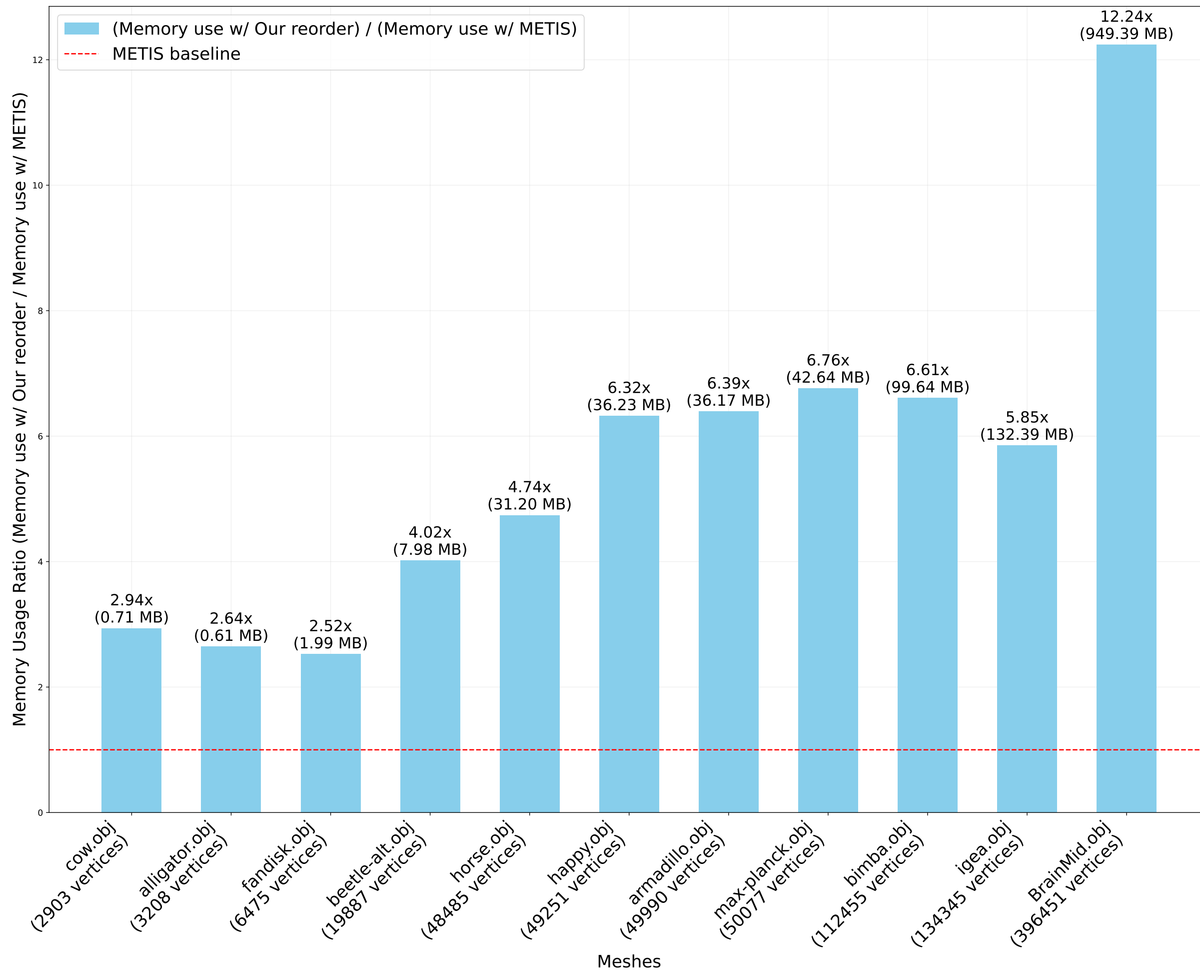

Figure 5.5: Showing the memory usage for Cholesky factorization after using our method and METIS. The memory usage is directly from cuSolver API and is measured during the memory allocation stage. The memory usage is normalized by the memory usage of METIS. The actual memory usage in megabytes with our method is shown on the top of the bars. The memory usage of our method is larger than METIS for all tested inputs by an average of 4.99 times.

runtime and the trade-off is not worth it based on the current implementation.

We extract the memory usage of the Cholesky factorization during the memory allocation stage to evaluate the memory consumption of our GPU-based Nested Dissection algorithm compared to METIS. The result aligns with the fill-in ratio comparison. Our algorithm consumes more memory than METIS for all tested inputs by an average of 5.17 times. The largest result is 12.24 times more memory usage for the BrainMid.obj input. The result is shown in Figure 5.5. The larger fill-in ratio leads to larger memory usage, which results in a longer total runtime.

The trade-off between reordering time and memory consumption for our current implemen-

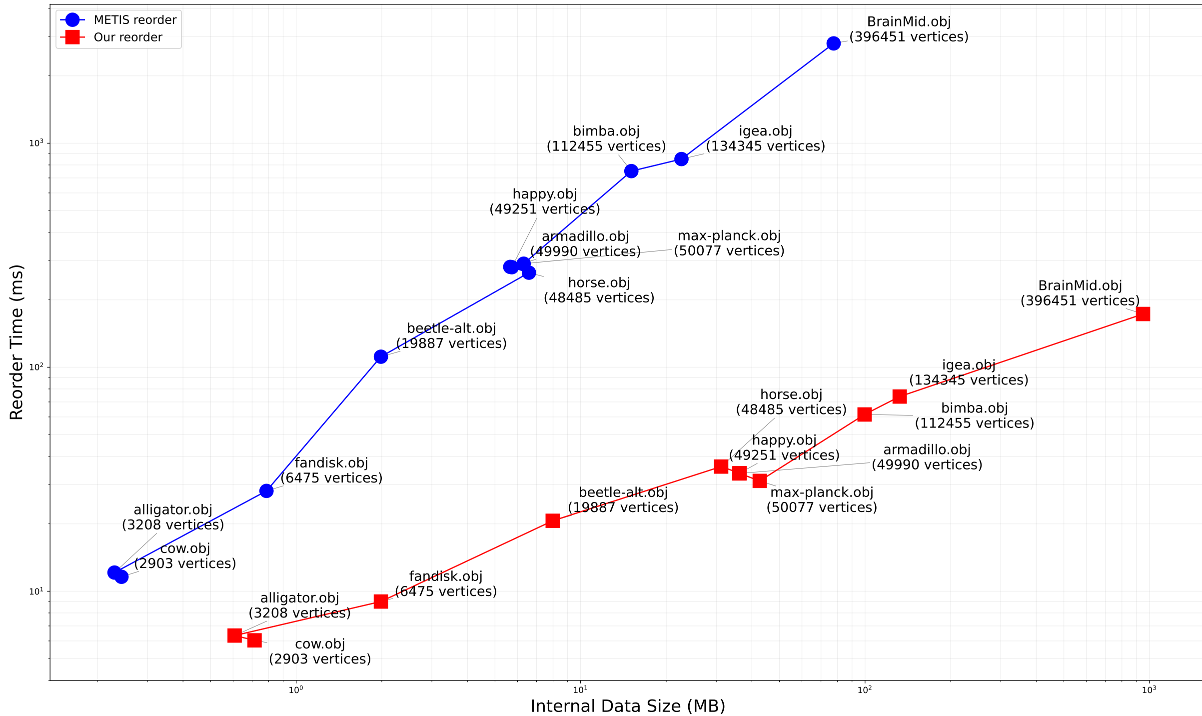

Figure 5.6: Show the tradeoff between reordering time and the memory consumption for our method and METIS for various input meshes. The memory consumption reflects the fill-in ratio. Essentially this figure is showing the trade-off between reordering time and the fill-in ratio.

tation is shown in Figure 5.6. Our result generally sits on the lower right side of the METIS result, which indicates that our algorithm is faster in terms of reordering time but consumes more memory. The result shows that we are trying to trade off the reordering time with the fill-in ratio, which leads to a longer factorization time and larger memory usage. The main goal for our future work is to reduce the fill-in ratio with a better partitioning strategy.

To better evaluate our current implementation, we test the memory usage of the Cholesky factorization with different Stage 1 partition levels for the BrainMid.obj input. The result is shown in Figure 5.7. We see that as the partition level increases, the memory usage decreases, while the reordering time increases. This happens because the as the partition level increases, we have a finer partitioning, hence finer reordering, which leads to a smaller fill-in ratio. However, as the partition level increases, the reordering time increases due to more partitioning. From the Figure 5.7, the decrease in memory usage slows down after the partition level reaches 4, while the increase in running time is consistent. This is because the partition from the current implementation is not balanced, leading to more edge cuts for compare to a balanced partition

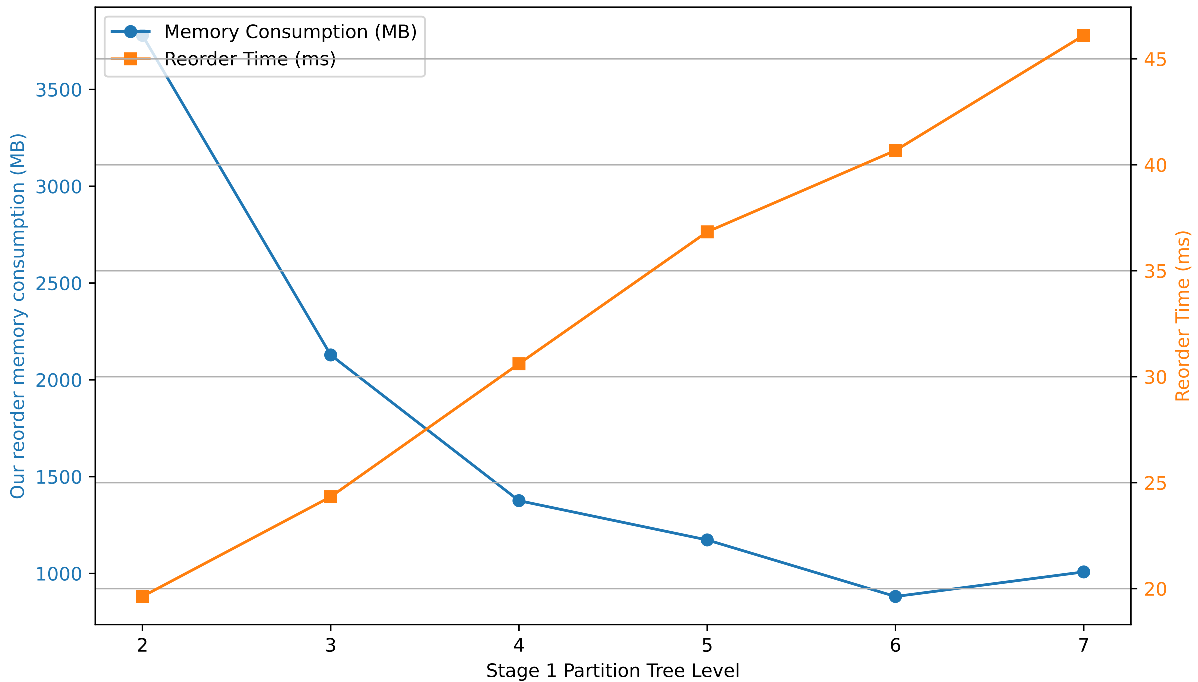

Figure 5.7: Show the memory usage of the Cholesky factorization and reorder time with different Stage 1 partition level for the BrainMid.obj input. The memory usage is directly from cuSolver API and is measured during the memory allocation stage.

for the same partition level in Stage 1, hence more *nnz*. This indicates that the main goal for our future work is to develop a more balanced partitioning strategy.

Table 5.1 shows the runtime for different stages of the linear solving process with our reordering and with METIS for various input meshes. The reorder time is either our GPU-based Nested Dissection runtime (Our Reorder) or CPU METIS runtime (METIS). The memory copy time (MC) is the time for copying the reordered matrix to the GPU. The analyze pattern time (AP) is the time for analyzing the matrix pattern. The post analyze alloc time (PAA) is the time for allocating memory for the Cholesky factorization on GPU. The factorize time (Factorization) is the time for the Cholesky factorization on GPU. The total time (Total) is the sum of all stages. The result is the raw data of the figures from Figure 5.1, Figure 5.3, and Figure 5.4.

Table 5.1: Comparison of runtime for different stages of the linear solving process with our reorder and with METIS for various input meshes. The runtime is in milliseconds.

Input	#V	Our Reorder	METIS	MC	AP	PAA	Factorization	Total
alligator	3208	4.70	/	0.12	10.16	0.74	4.74	20.46
		/	12.07	0.10	5.99	0.88	1.40	20.44
fandisk	6475	5.39	/	0.17	31.92	0.62	18.69	56.79
		/	30.14	0.16	15.13	0.71	3.93	50.07
dragon	10000	7.74	/	0.23	39.48	0.79	20.85	69.09
		/	50.24	0.22	14.81	0.74	3.57	69.58
beetle/alt	19887	9.18	/	0.41	74.31	0.93	43.11	127.94
		/	108.08	0.44	36.39	0.65	7.99	153.55
horse	48485	14.38	/	0.87	464.61	1.41	397.64	878.91
		/	268.50	0.92	96.98	2.16	21.26	389.82
happy	49251	15.31	/	0.90	361.21	1.19	352.37	730.98
		/	277.81	0.85	83.47	1.12	19.52	382.77
armadillo	49990	15.24	/	0.80	355.04	0.70	291.94	663.72
		/	308.38	0.94	88.71	0.98	15.79	414.80
max-planck	50077	14.90	/	0.89	417.07	0.90	463.68	897.44
		/	315.51	0.89	90.06	0.80	20.71	427.97
bimba	112455	20.40	/	1.82	992.43	1.48	1403.78	2419.91
		/	705.01	2.02	216.29	0.92	47.27	971.51
igea	134345	24.87	/	2.30	1525.19	1.38	2602.80	4156.54
		/	847.49	2.07	288.94	1.16	91.43	1231.09
BrainMid	396451	48.43	/	6.75	5773.32	1.50	25773.36	31603.36
		/	2794.41	6.77	955.83	1.81	411.78	4170.60

Chapter 6

Conclusion

This thesis presented the first GPU-based Nested Dissection algorithm for matrix reordering in Cholesky-based linear solvers, specifically for triangle mesh inputs. Our work addresses a critical bottleneck in GPU-accelerated linear solving processes, namely the matrix reordering stage, which has traditionally relied on CPU-based implementations. The key contributions of our work include:

1. The first GPU-accelerated Nested Dissection algorithm that leverages the pre-partitioning strategy of RXMesh to achieve high levels of parallelism.
2. A two-stage reordering approach that efficiently handles both coarse-grained (patch-level) and fine-grained (within-patch) reordering.
3. An average of 6.23 times, and up to 16.1 times faster reordering time compared to state-of-the-art CPU-based reordering methods like METIS.
4. Insights on the trade-off between reordering time and memory consumption, highlighting the challenges and opportunities for further optimization.

While our current implementation shows promising results in terms of reordering speed, there is significant room for improvement, particularly in reducing the fill-in ratio to more closely match that of METIS. Our results indicate that our method produces *nnz* ratios that are on average 5.14 times higher than METIS, leading to longer Cholesky factorization times

and larger memory footprints. This increase in fill-in ratio results in our method being slower than METIS by an average of 0.61 times in total runtime performance, despite the faster reordering time. Future work will focus on refining our partitioning strategy to achieve better fill-in reduction without sacrificing the significant speed advantages in reordering. This will involve exploring hybrid approaches or developing new heuristics specifically tailored for GPU execution.

6.1 Future Work

1. **Fill-in Reduction Optimization:** Given the current trade-off between reordering speed and fill-in reduction, future work will primarily focus on refining our algorithm to achieve better fill-in reduction without sacrificing the significant speed advantages in reordering. After investigating the detailed parameters of our GPU implementation, we have identified that the large fill-in ratio compared to METIS is primarily due to the imbalance of the recursive bisecting K-means partitioning strategy. We will explore the possibility of using a more balanced partitioning strategy to reduce the fill-in ratio without sacrificing the speed advantage. The imbalance of the bisection increases the number of edges between the two subgraphs, which leads to a larger fill-in ratio.

2. **Partition Level Optimization:** Our experiments with different partition levels for the BrainMid.obj input showed that increasing the partition level can decrease memory usage at the cost of increased reordering time. Future work will involve finding the optimal partition level that balances memory usage and reordering time across various input sizes.

3. **Efficiency Analysis:** We will perform a detailed analysis of our implementation's efficiency relative to the theoretical peak performance of GPU architectures. This will include roof-line analysis and performance profiling. By understanding where our implementation stands in relation to the GPU's theoretical capabilities, we can identify areas for further optimization and potentially improve overall runtime performance.

By addressing these areas, we aim to develop a GPU-based reordering algorithm that not only offers faster reordering times but also competes with METIS in terms of fill-in reduction

and overall linear solving performance.

REFERENCES

[1] Patrick R. Amestoy, Timothy A. Davis, and Iain S. Duff. An approximate minimum degree ordering algorithm. *SIAM Journal on Matrix Analysis and Applications*, 17(4):886–905, 1996. doi: 10.1137/S0895479894278952.

[2] Tobias Colding, William Minicozzi, and Erik Pedersen. Mean curvature flow. *Bulletin of the American Mathematical Society*, 52(2):297–333, January 2015. ISSN 1088-9485. doi: 10.1090/s0273-0979-2015-01468-0.

[3] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th National Conference*, pages 157–172. ACM Press, 1969. doi: 10.1145/800195.805928.

[4] Elizabeth Cuthill. Several strategies for reducing the bandwidth of matrices. In *Sparse Matrices and their Applications*, pages 157–166. Springer US, 1972. ISBN 9781461586753. doi: 10.1007/978-1-4615-8675-3_14.

[5] Bas O. Fagginger Auer and Rob H. Bisseling. *A GPU Algorithm for Greedy Graph Matching*, pages 108–119. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-30397-5. doi: 10.1007/978-3-642-30397-5_10.

[6] Alan George and Joseph W. H. Liu. The evolution of the minimum degree ordering algorithm. *SIAM Review*, 31(1):1–19, 1989. doi: 10.1137/1031001.

[7] Michael S. Gilbert, Kamesh Madduri, Erik G. Boman, and Sivasankaran Rajamanickam. Jet: Multilevel graph partitioning on graphics processing units. (arXiv:2304.13194), January 2024. URL `http://arxiv.org/abs/2304.13194`.

[8] Azzam Haidar, Ahmad Abdelfatah, Stanimire Tomov, and Jack Dongarra. High-performance Cholesky factorization for GPU-only execution. In *Proceedings of the Workshop on General Purpose Computing using GPUs*, GPGPU-10, pages 42–52, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349154. doi: 10.1145/3038228.3038237.

[9] Alec Jacobson. Common 3d test models. `https://github.com/alecjacobson/common-3d-test-models`, 2023.

[10] P. J. Jonge. A comparative study of algorithms for reducing the fill-in during Cholesky factorization. *Bulletin Géodésique*, 66(3):296–305, October 1992. ISSN 0007-4632, 1432-1394. doi: 10.1007/BF02033190.

[11] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, December 1998. ISSN 1064-8275. doi: 10.1137/S1064827595287997.

[12] Manpreet S Khaira, Gary L Miller, and Thomas J Sheffler. *Nested Dissection: A survey and comparison of various nested dissection algorithms*. Carnegie-Mellon University. Department of Computer Science, 1992.

[13] Wan Luan Lee, Dian-Lun Lin, Tsung-Wei Huang, Shui Jiang, Tsung-Yi Ho, and Yibo Lin. G-kway: Multilevel GPU-accelerated k–way graph partitioner. In *ACM/IEEE Design Automation Conference*, June 2024. doi: 10.1145/3649329.3656238.

[14] Joseph W. H. Liu. Modification of the minimum-degree algorithm by multiple elimination. *ACM Trans. Math. Softw.*, 11(2):141–153, June 1985. ISSN 0098-3500. doi: 10.1145/214392.214398.

[15] Wai-Hung Liu and Andrew H. Sherman. Comparative analysis of the Cuthill-McKee and the reverse Cuthill-McKee ordering algorithms for sparse matrices. *SIAM Journal on Numerical Analysis*, 13(2):198–213, 1976. doi: 10.1137/0713020.

[16] Ahmed H. Mahmoud, Serban D. Porumbescu, and John D. Owens. RXMesh: A GPU mesh data structure. *ACM Transactions on Graphics*, 40(4):104:1–104:16, August 2021. doi: 10.1145/3450626.3459748.

[17] NVIDIA. NVIDIA cuDSS (preview): A high-performance CUDA library for direct sparse solvers, . URL `https://developer.nvidia.com/cudss`. Accessed 08/02/2024.

[18] NVIDIA. cuSOLVER API reference, . URL `https://docs.nvidia.com/cuda/cusolver/index.html`. Accessed 08/02/2024.

[19] François Pellegrini. Scotch and PT-Scotch graph partitioning software: An overview. In Olaf Schenk Uwe Naumann, editor, *Combinatorial Scientific Computing*, pages 373–406. Chapman and Hall/CRC, 2012. doi: 10.1201/b11644-15. URL `https://inria.hal.science/hal-00770422`.

[20] Edward Rothberg and Stanley C. Eisenstat. Node selection strategies for bottom-up sparse matrix ordering. *SIAM Journal on Matrix Analysis and Applications*, 19(3):682–695, 1998. doi: 10.1137/S0895479896302692.

[21] Kasia Świrydowicz, Eric Darve, Wesley Jones, Jonathan Maack, Shaked Regev, Michael A. Saunders, Stephen J. Thomas, and Slaven Peleš. Linear solvers for power grid optimization problems: A review of GPU-accelerated linear solvers. *Parallel Computing*, 111:102870, July 2022. ISSN 0167-8191. doi: 10.1016/j.parco.2021.102870.

[22] James D. Trotter, Sinan Ekmekçibaşı, Johannes Langguth, Tugba Torun, Emre Düzakın, Aleksandar Ilic, and Didem Unat. Bringing order to sparsity: A sparse matrix re-ordering study on multicore CPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '23, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400701092. doi: 10.1145/3581784.3607046.

[23] Chang Yu, Yi Xu, Ye Kuang, Yuanming Hu, and Tiantian Liu. MeshTaichi: A compiler for efficient mesh-based operations. *ACM Transactions on Graphics*, 41(6):1–17, December 2022. ISSN 0730-0301, 1557-7368. doi: 10.1145/3550454.3555430.

[24] Qingnan Zhou and Alec Jacobson. Thingi10k: A dataset of 10,000 3d-printing models. *arXiv preprint arXiv:1605.04797*, 2016.