Improving the Performance of Bundle Adjustment for On-Device SLAM using GPU Resources

by

Shishir Gopinath

B.Sc., Simon Fraser University, 2021
B.A.Sc., University of British Columbia, 2017

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

in the
School of Computing Science
Faculty of Applied Sciences

SIMON FRASER UNIVERSITY
Spring 2023

Declaration of Committee

Name:	**Shishir Gopinath**
Degree:	**Master of Science**
Thesis title:	**Improving the Performance of Bundle Adjustment for On-Device SLAM using GPU Resources**
Committee:	**Chair:** Alaa Alameldeen Associate Professor, Computing Science
	Steven Ko Supervisor Associate Professor, Computing Science
	Jiangchuan Liu Committee Member Professor, Computing Science
	Keval Vora Examiner Assistant Professor, Computing Science

Abstract

Visual-inertial SLAM systems estimate the state and trajectory of a moving device while building a map of the environment using sensors such as cameras and inertial measurement units. These systems apply a general form of bundle adjustment to reduce error accumulated in the relationships between keyframes, map points, and inertial states. We present techniques to accelerate bundle adjustment for on-device SLAM using GPU resources. First, we develop Vulkan compute shaders for calculating the Schur complement of a sparse matrix to accelerate local visual-inertial bundle adjustment. Next, we extend this work for larger-scale global bundle adjustment problems by developing an iterative linear solver for explicit and implicit approaches. To evaluate the performance, we integrate our methods into a graph optimization library, g2o, and visual-inertial SLAM system, ORB-SLAM3, and process a mix of indoor and outdoor datasets on desktop and embedded devices. We also test our methods on large-scale bundle adjustment datasets.

Keywords: Bundle Adjustment; Visual-Inertial SLAM; GPU Acceleration; Compute Shaders; Non-linear Optimization; Embedded Devices

Acknowledgements

Firstly, many thanks to my committee for their support and feedback, especially to my supervisor Dr. Steven Ko of the Reliable Systems Lab at Simon Fraser University for his guidance, and knowledge of SLAM, mobile devices, software systems, and programming languages. I would also like to give a special thanks to Dr. Karthik Dantu from the University at Buffalo for providing his time and expertise on SLAM and robotics.

To everyone from RSL, thank you. You have made my time here an enriching experience, and I wish you all continued success in your efforts.

I would also like to acknowledge the efforts of the instructors and staff at SFU, with whom I had the pleasure of interacting with. I had a great time learning about different topics in computer science.

Of course, to my family, thank you for your support.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Simultaneous localization and mapping (SLAM) systems estimate the state and trajectory of a mobile robot while also building a map of the environment using onboard sensors such as cameras and LiDARs. Visual-inertial SLAM systems such as ORB-SLAM3 [1] achieve this by identifying and tracking incoming colour or depth images with significant changes (called *keyframes*), extracting 3D map points (landmarks) from these images, and estimating the camera pose for each keyframe using the map points as well as inertial input. Such SLAM systems are common building blocks in robot perception for navigation and manipulation, as well as in augmented and virtual reality systems to determine the position and orientation of a user's device relative to the environment.

A common challenge in visual-inertial SLAM systems is drift or the accumulation of inconsistencies in the relationships between the keyframes, map points, and relative odometry. A popular technique to enforce consistency, both *locally* between nearby keyframes and related map points, as well as *globally* when a place is revisited (i.e. for loop closure), is *bundle adjustment* (BA). Bundle adjustment resolves these inconsistencies by minimizing a cost function, such as the weighted sum of squares [2], over error constraints. It co-optimizes the growing set of pose and landmark parameters from the map in order to obtain more refined estimates. Measured, or virtual observations, serve as the reference for computing the error for each constraint. In visual bundle adjustment, reprojection constraints model the difference between the measured image location of an observed map point and its projection computed from estimated keyframe pose and map point parameters [3]. The parameters are refined numerically using iterative non-linear least squares optimization methods [2, 3, 4].

Modern visual-inertial SLAM systems [1] perform bundle adjustment using an optimization library such as g2o [5], GTSAM [6], or Ceres Solver [7]. In ORB-SLAM3, local bundle adjustment is performed by the local mapping thread upon processing one or more incoming keyframes [1]. Although local bundle adjustment operates on a subset of the map, it is still a computationally expensive process, and this can cause delays in the local mapping pipeline. Global bundle adjustment, which operates on the entire map, is even more time-consuming [1]. Even if executed in a dedicated thread, large global bundle adjustment

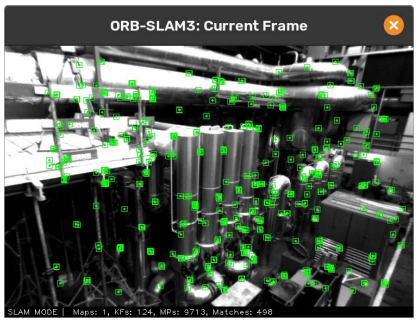


Figure 1.1: Left camera view of ORB-SLAM3 processing the EuRoC Machine Hall 01 visual-inertial dataset.

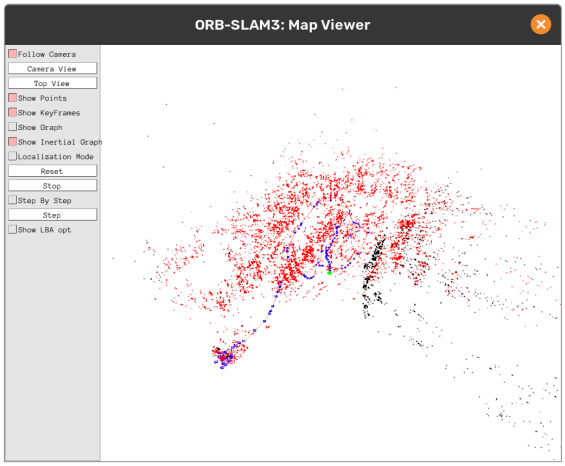


Figure 1.2: The map generated by ORB-SLAM3 when processing the EuRoC Machine Hall 01 visual-inertial dataset.

latencies may result in early termination of the optimization, or delay related processes such as monocular map initialization [8]. Decreasing these latencies would enable a SLAM system to enforce consistency more frequently, and perform additional BA iterations in the same amount of time. Therefore, we consider it desirable to optimize the performance of both local and global bundle adjustment, since larger computation times may prevent a SLAM system from accurately performing its mapping and localization tasks. To exacerbate the problem, SLAM systems run on a diverse set of platforms, including resource constrained micro-air vehicles. Thus, it is critical to optimize the compute-heavy parts of the SLAM pipeline, such as bundle adjustment.

In this thesis, we investigate how we can efficiently use GPU resources to accelerate both local and global bundle adjustment for on-device visual-inertial SLAM. In particular, we look at general numerical methods for small and large sparse block matrices, and how we can execute them quickly on a GPU with low latency. We first focus on improving the performance of local visual-inertial bundle adjustment in Chapter 4 and implement

our techniques for g2o and ORB-SLAM3. In Chapter 5, we expand this work to accelerate global bundle adjustment. We evaluate our methods using well-known visual-inertial datasets, EuRoC [9] and TUM-VI [10], as well as the popular Bundle Adjustment in the Large (BAL) datasets [3]. Our experiments show that we can obtain a reasonable performance improvement in a memory-efficient manner over the existing g2o implementation with our techniques, even when using an embedded GPU. We have made our implementation available online. [1]

Material Reuse

The work presented in this thesis is primarily based on our conference paper, "Improving the Performance of Local Bundle Adjustment for Visual-Inertial SLAM with Efficient Use of GPU Resources" by Shishir Gopinath, Karthik Dantu, and Steven Y. Ko, accepted for presentation at the 2023 IEEE International Conference on Robotics and Automation (ICRA) [11] © 2023 IEEE.

Text, as well as figures, and tables are reused, primarily in Chapter 1, Section 2.1, Chapter 4 except for Section 4.3, and Chapter 6 before Section 6.1. Other chapters and sections are mostly new material, except for where indicated otherwise. If the reused material is relevant to your work, please refer to the final paper [11].

IEEE Copyright Notice

[1] Our code is available at `https://github.com/sfu-rsl/gpu-block-solver`.

Chapter 2

Background

2.1 Bundle Adjustment

Since our main contribution is in developing techniques that efficiently perform visual-inertial bundle adjustment, we provide a brief overview. We discuss three important aspects of visual-inertial bundle adjustment directly relevant to our approach—bundle adjustment as a graph optimization problem, non-linear least squares optimization, and the Schur complement method. This discussion is mainly based on g2o—for details, please refer to its description [5].

Bundle Adjustment as a Graph Optimization Problem: Modern SLAM systems describe visual bundle adjustment problems as graphs, representing keyframe camera poses and map points as vertices, connected by edges modelling reprojection errors. Visual-inertial bundle adjustment introduces additional vertices and constraints for relative motion and IMU bias residuals between consecutive keyframes [1]. The optimization of these constraints is formulated as a non-linear least squares problem (i.e. minimization of the sum of squared errors) and solved using an iterative method such as the Levenberg-Marquardt algorithm (LMA) [12].

Non-Linear Least Squares Optimization: The problem of finding a parameter update to minimize non-linear constraints in local bundle adjustment (i.e., reprojection errors) can be represented using a linearized sparse system of equations. For LMA, this is expressed as $(H + \lambda I)\Delta x = -b$, where H is a symmetric positive semi-definite sparse block matrix approximating the Hessian, λ is a damping factor added to the diagonal, and Δx and $-b$ are vectors corresponding to the parameter update and gradient respectively [5, 2]. Each iteration of LMA attempts to solve for a new Δx to reduce the error. However, for bundle adjustment, it is common to exploit the problem structure to solve the system more efficiently, by partitioning it in terms of pose (p) and landmark (l) variables. The vector Δx is partitioned into pose update Δx_p and the landmark update Δx_l, while $(H + \lambda I)$ is partitioned into submatrices H_{pp}, H_{pl}, H_{ll}, and H_{pl}^T [5]. Similarly, b is split into b_p and b_l. For visual-inertial bundle adjustment, variables added by inertial constraints are also treated

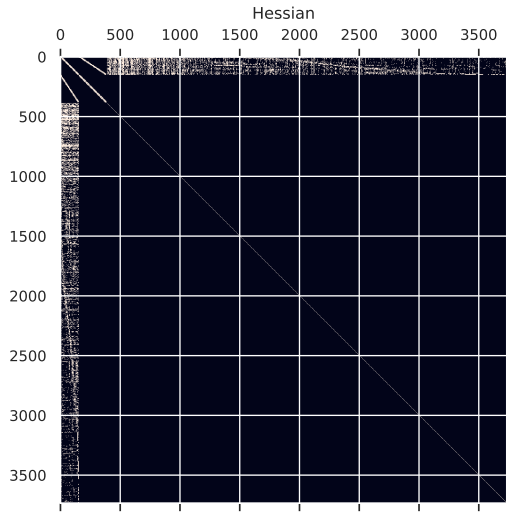


Figure 2.1: The structure of the Hessian H generated by local-inertial bundle adjustment.

as pose variables, hence H_{pp} contains off-diagonal blocks. Submatrix H_{ll} is block diagonal when constraints between landmark parameters are absent, making it trivially invertible for 3D map points. This is ideal for the purposes of the Schur complement method, described next.

Schur Complement Method: Using forward substitution, the Schur complement of $(H + \lambda I)$ is used to find a reduced system $H_{Schur}\Delta x_p = b_{Schur}$ [2, 5]. Solving the reduced system is relatively efficient, as landmark parameters generally outnumber pose parameters [3]. After solving for Δx_p, the landmark update can be computed via back substitution. These relations are summarized by the following equations [5]:

$$H_{Schur} = H_{pp} - H_{pl}H_{ll}^{-1}H_{pl}^{T} \tag{2.1}$$

$$b_{Schur} = -b_p + H_{pl}H_{ll}^{-1}b_l \tag{2.2}$$

$$\Delta x_l = H_{ll}^{-1}(-b_l - H_{pl}^{T}\Delta x_p) \tag{2.3}$$

The Schur complement method may also be referred to as *Schur elimination* in bundle adjustment literature [13, 14]. Some bundle adjustment implementations use explicit evaluation, where H_{Schur} is computed and stored [5], while others evaluate matrix-vector products with H_{Schur} implicitly [3, 15, 14]. We explore both options in this thesis.

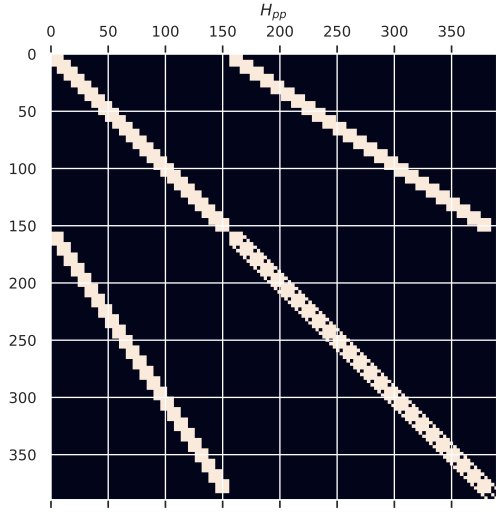


Figure 2.2: The structure of H_{pp} generated by local-inertial bundle adjustment. Only the upper triangular blocks are generated at runtime.

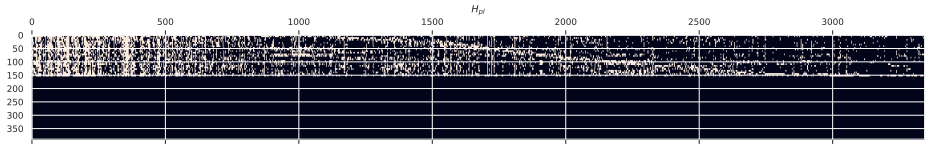


Figure 2.3: The structure of H_{pl} generated by local-inertial bundle adjustment. Note the absence of non-zeros for inertial pose variables after the 156th row, which corresponds to 26 keyframes with 6 parameters each. These variables do not form constraints with landmark variables.

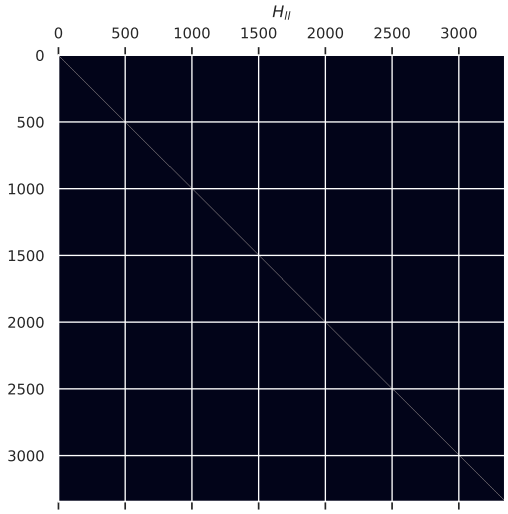


Figure 2.4: The structure of H_{ll} generated by local-inertial bundle adjustment. It consists of 3×3 blocks along the diagonal.

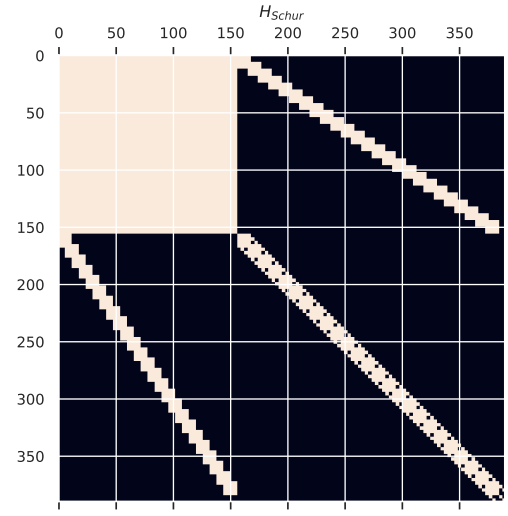


Figure 2.5: The structure of H_{Schur} generated by local-inertial bundle adjustment, which inherits the structure from H_{pp}. The lower triangular blocks are shown for convenience, but are not generated at runtime.

2.2 Vulkan Compute Shader Programming Model

In this section, we provide a brief overview of compute shaders in the context of the Vulkan API, on aspects which are relevant to our implementation. We provide a description of the execution model, memory model, synchronization primitives, constants, and subgroup operations. For further details, please refer to the Vulkan and OpenGL Shading Language (GLSL) specifications [16, 17]. We additionally refer to the CUDA C++ Best Practices Guide [18], since we evaluate our methods using NVIDIA GPUs, and also due to the similarities between the CUDA and compute shader programming models.

2.2.1 Execution Model

A compute shader describes a computation across one or more workgroups, which consist of work items [16]. Each workgroup contains one or more shader invocations, which execute the `main` function of a shader program [16]. The shader program itself may be written in GLSL or another language which can be compiled into SPIR-V bytecode. For our methods, we generate this bytecode during library compile-time, but it can be prepared at run-time as well. This bytecode cannot be used directly. At application run-time, pipeline objects are constructed from the bytecode, which can then be dispatched for execution on a GPU. The Vulkan implementation may also perform additional processing on the bytecode, including optimizations such as loop unrolling and dead-code elimination [19].

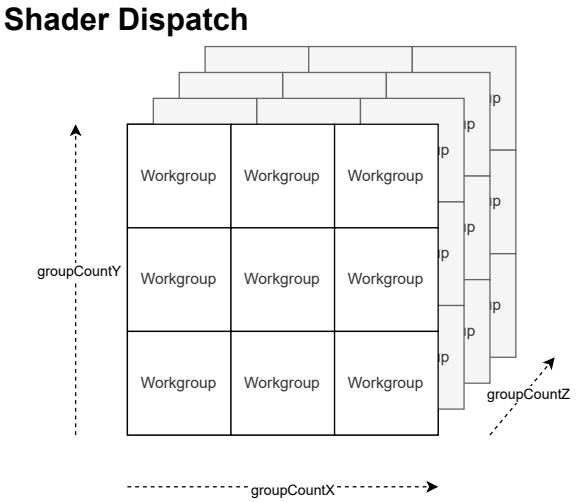


Figure 2.6: A compute shader dispatch describes the execution of a shader across one or more workgroups.

A compute shader dispatch can be organized into three levels. First, the number of workgroups executed by a single compute shader dispatch is represented by a rectangular volume computed from the product of x, y, and z dimensions [16], as shown in Figure 2.6. Within each workgroup, the workgroup size, which also consists of three dimensions, determines the

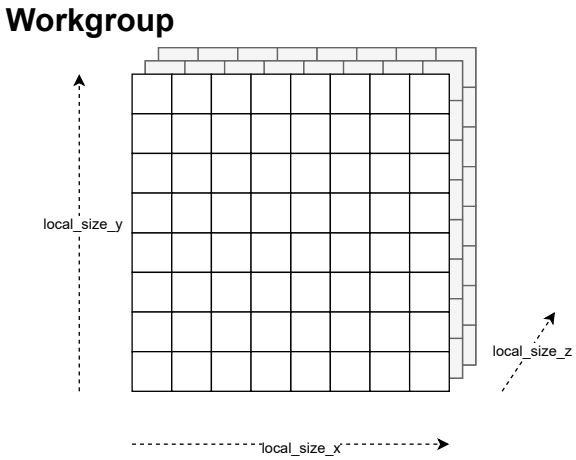


Figure 2.7: A workgroup consists of one or more shader invocations. Each invocation is represented as a cell in the diagram. Invocations in the same workgroup can synchronize execution and memory accesses using barriers, and access the same shared variables. Invocations are also implicitly organized into subgroups. Within the same subgroup, invocations can interact with each other using subgroup functions.

number of invocations within a single workgroup, depicted in Figure 2.7. Workgroups in the same dispatch do not have any execution dependencies between them, and thus are allowed to execute in parallel. Invocations may also execute in parallel if they are active [20]. Each is assigned to a subgroup, defined as "a set of invocations that can synchronize and share data with each other efficiently" [16]. Subgroup operations allow invocations to communicate and perform computations, such as reductions, without the use of shared variables.

A feature of the compute shader execution model is that each invocation can read information about itself from built-in variables, and use that to determine its task. This information includes the current workgroup ID, local invocation ID (among invocations in a workgroup), or global invocation ID (among invocations across all workgroups) [17]. These are three-dimensional values, due to the organization described previously. It is also possible to retrieve one-dimensional IDs with respect to the current subgroup [20].

2.2.2 Memory Model

Compute shaders may access several types of memory, and we describe a simplified model here. Global memory is used to back shader storage buffer objects (SSBOs). SSBOs are used for storing a computation's input and output data, and persist beyond the scope of a shader dispatch. Shaders can also access shared variables, which exist in the scope of a single workgroup's execution and may be restricted to a few kilobytes per workgroup. They are faster to access than values in global memory [18], making them well-suited for storing intermediate computations. Invocations in the same workgroup can read and write to the same shared variables, though accesses must be synchronized with the use of barriers. SSBOs

and shared variables may be cached in L1 and L2 memory caches [18]. Register memory is used to store sufficiently small variables for fast access by a single invocation [18].

2.2.3 Memory Properties

In Vulkan, memory must be separately allocated prior to creating an SSBO. Allocations must specify a memory type. A memory heap may support one or more memory types, with each type having a set of properties. These properties provide guarantees about the availability and visibility of values written to the memory, which in turn also affect the performance of memory accesses. For our purposes, we are interested in memory types supporting the properties described below [16]. *Host* refers to the processor and environment of the application, on which the Vulkan API is used to invoke tasks, while *device* refers to the processor and environment on which these tasks are executed [16].

- **Device Local:** The memory is the most efficient for devices accesses.
- **Host Visible:** The memory can be mapped for host access.
- **Host Cached:** The memory is cached by the host, allowing for faster access on the host application side. However, it lacks the visibility guarantees provided by host coherent memory. Memory ranges must be flushed for the device to see host writes, and invalidated for the host to see device writes.
- **Host Coherent:** Flushing and invalidating memory ranges is not necessary, but the memory may be slower to use.

2.2.4 Command Buffers and Queues

Vulkan commands are recorded into command buffers, which can be submitted to a queue for execution [16]. Command buffers may be submitted to a queue more than once, allowing for a prepare-once, execute-many-times approach. They can also be submitted to different queues, allowing for asynchronous execution.

2.2.5 Synchronization Primitives

Vulkan provides a set of synchronization primitives for various situations. Shader invocation control functions provide functionality to control the relative execution of invocations in a workgroup [17]. Invocations within the same workgroup must synchronize access to shared variables or buffers using memory and execution barriers in order for each other's writes to become visible. Likewise, pipeline barriers, which are inserted by the host, specify execution and memory dependencies between shader dispatches and other commands. Other primitives include fences, which can be used to signal the host when a command buffer has finished execution in a queue [16].

2.2.6 Push Constants and Specialization Constants

GLSL provides two methods to specify constants in a compute shader at run-time. Push constants are a small bank of writable values representing a high speed path to update constant data [16]. Push constant updates are recorded into a command buffer prior to recording a compute shader dispatch command. Alternatively, specialization constants can be specified at pipeline creation time. Specialization constants allow for control flow optimizations and unrolling of loops [19]. However, the exact behaviour is implementation-dependent.

2.2.7 Subgroup Operations

Subgroup operations allow data to be shared between invocations in a subgroup without using shared variables. An invocation may be inactive or active, depending on factors such as the workgroup size and dynamic branching [20]. Whether an invocation participates in a subgroup operation depends on if it is active, which is useful for conditional data-sharing [20].

We briefly outline a subset of the subgroup-aware functionality offered by GLSL extensions [20, 17], which we use in our implementation. Arithmetic extensions provide functionality for reductions, including the sum, product, maximum, or minimum of a variable across active subgroup invocations [20]. This also includes variants for inclusive and exclusive scan operations. Ballot functionality includes support for broadcasting one invocation's value to all other invocations in a subgroup [20]. Basic subgroup functionality includes support for reading information about the current subgroup, the number of subgroups, and also support for subgroup-level barriers [20]. A very useful function for selecting a single active invocation from a subgroup is `subgroupElect()`.

Chapter 3

Related Work

We now provide a brief overview of existing work on SLAM, non-linear optimization libraries, bundle adjustment acceleration, GPU acceleration for SLAM on embedded devices, and related HPC work.

3.1 Visual-Inertial SLAM

Visual-inertial odometry and SLAM systems use inertial information to estimate changes in motion between visual frames, improving localization accuracy and robustness when visual tracking loss occurs [21, 22, 1, 23]. VINS-Mono fuses preintegrated IMU measurements [21] with monocular observations of visual features to improve state estimation [22]. Similarly, ORB-SLAM3 builds upon previous ORB-SLAM works [8, 24], incorporating inertial information into monocular, stereo, and RGB-D SLAM, while supporting both pinhole and fisheye camera types, as well as multiple maps in the same session [1]. Kimera takes a modular approach, where visual-inertial odometry is decoupled from mapping for improved localization [23]. It supports both fast mesh-based reconstruction for obstacle avoidance, and slower volumetric reconstruction for semantic annotation [23].

3.2 Bundle Adjustment and Non-linear Optimization

Many specialized optimization libraries have been developed for bundle adjustment and other non-linear least squares SLAM problems. Lourakis and Argyros [12] implemented Levenberg-Marquardt-based generic sparse bundle adjustment for large BA problems. Later, Agarwal et al. [3] developed a large-scale solver based on direct and iterative methods for non-linear least squares optimization, and published the Bundle Adjustment in the Large (BAL) datasets, which are now commonly used to evaluate BA performance. This work was further developed as Ceres Solver [7], which is used in systems such as VINS-Mono [22], and supports exact automatic differentiation via template and operator overloading [25]. A follow-up work by Kushal and Agarwal [26] explores the impact of visibility-based preconditioners for bundle adjustment using preconditioned conjugate gradients. GTSAM [6],

used in Kimera, is another popular library for estimation problems in robotics. It describes problems as bipartite factor graphs, in which unknown variables are connected via edges to factors representing functions [6]. The g2o library, which is used for non-linear least squares optimization in ORB-SLAM [8] and derivatives [24, 1], implements a similar model, in which hyper-edges directly represent ordered constraints between one or more vertices [5]. Recently, Demmel et al. [27] proposed RootBA, which implements a parallel algorithm based on QR decomposition and outperforms Ceres Solver on medium to large BAL problems.

3.3 Bundle Adjustment Acceleration

3.3.1 GPU Acceleration

A popular strategy to improve the performance of large-scale bundle adjustment is to execute massively parallel calculations on a GPU [28, 15, 29, 30]. This is often achieved with the use of a GPU compute API such as CUDA, in which programs called kernels (similar to compute shaders) carry out various calculations. GPU methods may use direct or iterative linear solving techniques, and there are tradeoffs for each.

Methods which use direct approaches usually compute and store the reduced pose system, and use it to calculate an exact parameter update. By using CUDA, Choudhary, Gupta, and Narayanan [28] compute the Schur complement on the GPU and the column vector on the CPU. They use MAGMA Cholesky decomposition [31] to solve for the pose update and achieve a $30-40\times$ speedup for datasets with up to 500 images [28]. The CUDA Bundle Adjustment library [32] takes a similar approach and targets the subset of the g2o functionality used in ORB-SLAM2 [24], achieving a speedup of roughly $8-10\times$ for maps generated by processing KITTI datasets [33]. Cao et al. [30] use tile-based block matrix multiplication for the Schur complement. They use LDLT decomposition to solve the pose update and find that their method outperforms other multicore methods in both speed and error reduction for BAL datasets.

On the other hand, iterative methods, which compute inexact parameter updates, have been shown to scale well on GPUs for very large bundle adjustment problems without compromising on the solution quality [15, 29]. Wu et al. [15] develop multicore methods for large-scale bundle adjustment. They implement methods based on preconditioned conjugate gradients (PCG) using implicit-Hessian and implicit-Schur matrix-vector evaluation. Their multicore CPU implementation provides a $10\times$ speedup over the previous methods proposed by Agarwal et al. [3] while their CUDA-based GPU implementation achieves a $30\times$ speedup [15]. Hänsch et al. [29] also investigate iterative methods on GPUs such as PCG for LMA, non-linear conjugate gradient descent, and alternating resection-intersection. They show that their methods converge faster on some problems while using up to half as much memory compared to the implementation by Wu et al. [15]. They also find that LMA methods converge slower but achieve better error reduction [29]. Another PCG-based

method is DeepLM [34], which implements visual bundle adjustment on top of the PyTorch deep-learning framework. More recently, MegBA implements distributed bundle adjustment using implicit-Schur PCG [14]. It uses edge-based partitioning to distribute blocks of the Hessian across multiple machines and GPUs.

3.3.2 FPGA Acceleration

Liu et al. [13] propose π-BA, which implements energy-efficient FPGA acceleration for expensive visual BA calculations, including the Schur complement. Their method outperforms Ceres-Solver running on a single ARM core by 7.3 times [13] for double-precision experiments on small BAL datasets. Sun et al. [35] developed another FPGA implementation, which avoids external memory accesses, called BAX. It also targets small BA problems using single-precision floats. By evaluating their methods on cropped BAL datasets, they find that BAX outperforms g2o on Intel and ARM processors, and is nearly as fast as a GPU implementation, while requiring the least amount of power [35].

3.3.3 Comparison to Existing Solutions

As the main goal of our work is to improve the performance of visual-inertial bundle adjustment, most existing GPU and FPGA solutions are not suitable due to lacking support for inertial constraints. We propose a more general solution using a hybrid approach, in which constraint specific calculations remain on the CPU, but operations involving sparse block matrices and dense vectors are executed on a GPU. We develop GPU-friendly techniques for handling sparse block matrices with variable-sized blocks, discussed in Chapter 4. This allows our solution to handle not only visual-inertial bundle adjustment, but also other similarly structured problems compatible with the reduced system partitioning scheme described in Chapter 2. Below, we outline several other key differences between our work and existing methods:

- We target both desktop machines and resource-constrained embedded devices [36] by using different allocation strategies to balance memory usage and performance.
- Since our methods do not require support for floating point atomic operations, we are able to target a wider range of GPUs using the Vulkan API.
- We demonstrate techniques to reduce setup latencies in order to keep up with frequent local bundle adjustment workloads on a GPU.
- We evaluate the performance of our methods on real visual-inertial bundle adjustment workloads generated on-the-fly.

3.4 Embedded GPU Acceleration for Visual SLAM

There are other parts of the visual SLAM pipeline which can benefit from GPU acceleration as well, even on low-power embedded devices. Ma et al. [37] parallelize ORB feature extraction and matching for ORB-SLAM2 using CUDA. On a Jetson TX2, their methods increased trajectory errors, but cut down the processing time for EuRoC sequences by a third, allowing for higher frame processing rates [37]. Lu et al. [38] parallelize optical flow tracking, non-linear least squares optimization, and marginalization for VINS-Mono using CUDA. Also using a Jetson TX2, they demonstrate speedups of 1.5-1.7× for marginalization, while optical flow tracking speeds improved by 1.9× [38]. Their methods nearly double the optical flow tracking framerate, but do not improve the performance of non-linear optimization [38].

3.5 High Performance Computing

In this section, we discuss how aspects of our work are related to research in high performance computing (HPC). In particular, we discuss sparse matrix formats and basic linear algebra subprograms (BLAS).

3.5.1 Sparse Matrix Formats

Sparse matrix formats are used to efficiently store matrices with few non-zero entries. These compressed formats typically require less memory than dense representations, while also reducing the amount of computations necessary for operations [39]. Conventional sparse matrix representations include the coordinate (COO), compressed sparse row (CSR), and compressed sparse column (CSC) formats [39]. Blocked variants of these representations, such as block compressed sparse row (BCSR), store indices for fixed-size sub-matrices (blocks) rather than scalar elements [40, 39]. An extension to BCSR is variable block compressed sparse row (VBR), for representing sparse matrices which contain blocks of different dimensions [40]. Block representations are cache-efficient [5, 41] since values of the same block are stored together in memory. Generally, all values of filled-in blocks are treated non-zeros. While CSR and CSC formats can achieve better compression, storing indices for non-zero scalars may significantly increase the memory usage.

Specialized sparse block matrix representations are often used for SLAM optimization problems [12, 25, 6, 42], due to the block-parameterization (the grouping of scalar components) of multidimensional state variables and observations. This block-parameterization is the underlying reason for the block structure of the large Hessian matrix described in Section 2.1. A key feature of block representations is that they allow for efficient updates when the linear system is rebuilt.

The reason why custom formats are used rather than formats such as VBR, is that often there is a need for blocks to be randomly accessed in constant time, and this can be enabled by storing additional information in the matrix data structure. Furthermore, implementations may take different approaches to reduce the overhead of allocation and modifications to the block structure [5, 41]. Another major limitation to existing block formats such VBR is that they are not directly supported by standard GPU libraries such as cuSPARSE [43], which is used to carry out operations involving sparse matrices efficiently on CUDA-enabled hardware.

To support visual-inertial bundle adjustment, our library also implements a custom sparse block matrix representation for variable block sizes. This representation allows for fast access of values both on the host (by g2o) and on a GPU (by compute shaders). Our representation differs from that of Polok et al. [42] in how operations are carried out on a GPU. We store additional information to quickly generate data structures and pre-record GPU commands for block-wise multiplications, as described in Section 4.2.

3.5.2 BLAS

Basic linear algebra subprograms (BLAS) perform low-level operations involving matrices and vectors efficiently [44]. There are multiple specifications and implementations of BLAS, and in general, they are organized into multiple levels [44]. Level 1 BLAS focuses on scalar and vector operations, Level 2 BLAS focuses on matrix-vector operations, and Level 3 BLAS is concerned with matrix-matrix operations [44]. BLAS operations may be carried out on a GPU, using an implementation such as cuBLAS [45] or MAGMA [31]. For our methods, since we use a non-standard matrix representation to support multiple block sizes, we implement a library of compute shaders to carry out low-level operations on sparse matrices, rather than use an existing BLAS implementation. This gives us additional control over exactly how operations such as multiplications are carried out on a GPU and allows us to implement strategies for different types of workloads (discussed in Section 4.3 and Section 5.4). In general, the techniques presented in this thesis may be useful for applications where BLAS is used for batched dense matrix multiplication.

Chapter 4

Improving The Performance of Local Bundle Adjustment

4.1 Introduction

In this chapter, we investigate how we can use GPUs to improve the performance of local bundle adjustment for ORB-SLAM3, on both a desktop machine and an NVIDIA Jetson Xavier NX embedded board. As discussed in Chapter 3, previous works have successfully applied GPU acceleration methods to large-scale bundle adjustment problems to obtain large speedups in offline scenarios. However, local bundle adjustment problems not only involve fewer keyframes and map points, but also introduce inertial constraints in the case of visual-inertial SLAM. These constraints change the structure of the matrices involved in bundle adjustment, as described in Chapter 2. Therefore, while investigating GPU acceleration for local bundle adjustment, we consider the following questions:

- Can computations across smaller bundle adjustment problems still benefit from GPU acceleration?
- How can we multiply sparse block matrices, with varying block matrix sizes, efficiently on a GPU?
- How can we mitigate the setup cost associated with moving calculations to the GPU?
- How well do our methods perform, in terms of execution time, when operating in the context of a real-time SLAM system?

To address these questions, we develop a solution based on Vulkan compute shaders. We show that by efficiently utilizing GPU resources, we can significantly improve the performance of local bundle adjustment for visual-inertial SLAM. This performance improvement comes from two observations we have made regarding the overhead and the workload. First, when solving for the parameter update in local-inertial bundle adjustment, computing the

Schur complement has the largest overhead, hence it is the most promising as an optimization candidate. Second, the workload consists of operations on *small- to medium-sized sparse matrices*, and we can tailor the use of GPU resources specifically for this workload to achieve better performance.

Based on these observations, we develop several techniques that efficiently handle small- to medium-sized sparse matrices, mainly for the Schur complement. First, we use a compact sparse block matrix representation that manages memory with low overhead for matrices of these sizes. Second, we design specialized work queues that enable us to parallelize sparse block matrix multiplication operations on a GPU. Third, we hide the latency of setting up GPU computation programs (called *compute shaders*) by pipelining their setup operations while their input data is being produced. Lastly, we use a particular GPU memory allocation strategy that boosts performance significantly, especially for longer SLAM sequences.

To concretely evaluate our approach, we develop a library for sparse matrix operations on GPUs based on our techniques and use it to implement a new block solver for g2o, which computes the Schur complement. We then replace the block solver in g2o with our own for ORB-SLAM3 and execute well-known SLAM datasets, EuRoC [9] and TUM-VI [10], for our evaluation. Our GPU implementation uses the Vulkan API, via Kompute [46], due to its popularity and cross-platform support (including support for desktop GPUs, mobile GPUs, and all major OSes) [47].

Our evaluation shows that our method achieves up to a 33.79% reduction in execution time for local visual-inertial bundle adjustment on a desktop machine with a dedicated GPU, and up to a 26.68% reduction on the Jetson Xavier NX embedded board using an integrated Volta GPU.

4.2 Our Approach

Motivation: As mentioned in Section 4.1, our performance improvement starts with two observations we have made from our analysis of the workload for local BA. Our first observation is that Schur complement has the largest overhead when solving for the parameter update in local visual-inertial bundle adjustment, as we show later in Section 4.4 (Figure 4.9 and Figure 4.10). This occurs especially with local BA with fewer keyframes. Our second observation is that the workload for local visual-inertial bundle adjustment consists of small- to medium-sized matrix operations. This observation can be seen in Figure 4.1, which shows the dimensions and the number of non-zero elements of the matrices used to compute the Schur complement in local visual-inertial bundle adjustment. We have generated the data by executing ORB-SLAM3 with EuRoC and TUM-VI indoor and outdoor sequences.

As we can see from Figure 4.1, the sizes of these matrices range from small (hundreds of rows and columns) to medium (tens of thousands of rows and columns). In comparison, a popular dataset for bundle adjustment, BAL [3] (Bundle Adjustment in the Large),

generates matrices with millions of rows and columns. The underlying reason why we see small- to medium-sized matrices for local visual-inertial bundle adjustment is that robot motion at typical speeds, outdoors as well as indoors, limits the number of map points from prior images that can be reprojected into the current image. Furthermore, the number of keyframes is limited in visual-inertial bundle adjustment, as the inertial constraints increase the number of associated pose variables [1].

Overview: Algorithm 1 describes the main solving step performed in each iteration of the non-linear least squares optimization described in Section 2.1. The goal is to solve for parameter update Δx, and the Schur complement is used to perform this process efficiently. In g2o, this is done by a component called a *block solver*.

In our approach, we allocate sparse block matrices in GPU-visible memory, perform computation-heavy parts of the Schur complement step in parallel using the GPU, and read back the computed H_{Schur} and b_{Schur} to solve for Δx_p on the CPU using sparse LDLT factorization from Eigen [48]. Lastly, we use this result to compute Δx_l also using the GPU. To perform these tasks efficiently, we develop and combine several techniques, which we describe next.

Algorithm 1 GPU Block Solver: Solving Step

Input: H_{pp}, H_{pl}, H_{ll}, M_1, M_2, H_{Schur}, b_{Schur}, v_1, b_p, b_l
Output: Δx_p, Δx_l

1: $H_{Schur} \leftarrow H_{pp}$ ▷ cpu
2: $b_{Schur} \leftarrow -b_p$ ▷ cpu
3: $v_1 \leftarrow -b_l$ ▷ cpu
4: $M_1 \leftarrow H_{ll}^{-1}$ ▷ gpu
5: $M_2 \leftarrow H_{pl} M_1$ ▷ gpu
6: $H_{Schur} \leftarrow H_{Schur} - M_2 H_{pl}^T$ ▷ async-gpu
7: $b_{Schur} \leftarrow b_{Schur} - M_2 v_1$ ▷ async-gpu
8: $\Delta x_p \leftarrow \text{LDLT}(H_{Schur}, b_{Schur})$ ▷ cpu
9: $v_1 \leftarrow v_1 - (\Delta x_p^T H_{pl})^T$ ▷ gpu
10: $\Delta x_l \leftarrow M_1 v_1$ ▷ gpu

Sparse Matrix Representation: One type of overhead that comes with the use of a GPU is the cost of data transfer between CPU-accessible memory and GPU-accessible memory, especially for systems with dedicated GPU memory. We manage this cost by developing a compact sparse block matrix representation. Figure 4.2 shows an example. In our representation, values for block matrices are stored in column-major order. A map is used to store the starting offset for each block matrix in the memory mapped buffer, along with the block dimensions. We also manage extra block indices and track dimensions for each block-row and block-column, which allow us to quickly check for filled-in blocks of a matrix. These are not shown in Figure 4.2. Our representation only stores non-zero values of the matrix in the buffer, which reduces the overall GPU memory requirement.

Work Queues: In order to perform matrix multiplications efficiently on a GPU, we design specialized work queues that manage computational tasks. Our unit of task is a block-

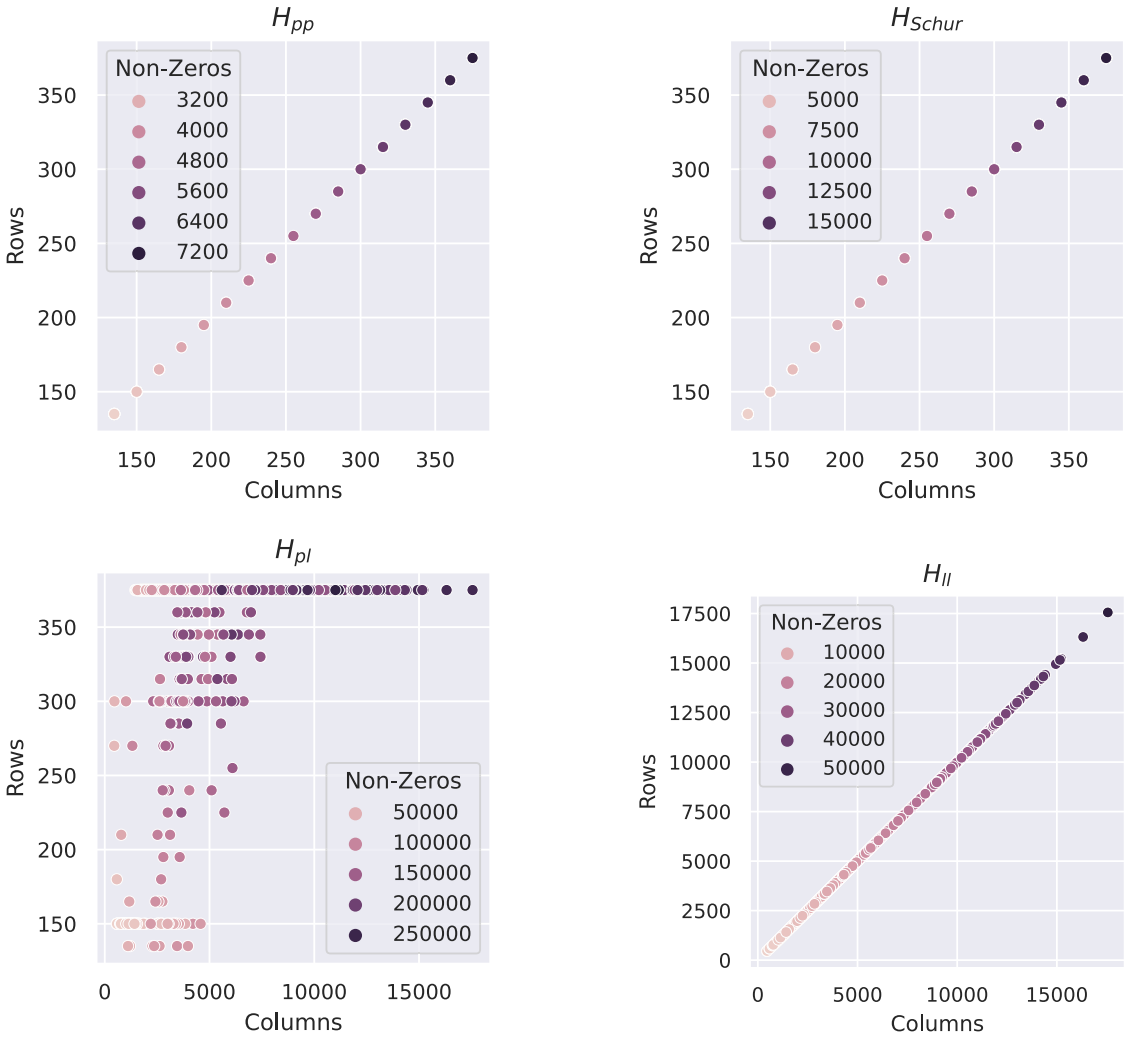


Figure 4.1: The dimensions and number of non-zeros of sparse matrices for local-inertial bundle adjustment across indoor and outdoor EuRoC and TUM-VI sequences. Only the upper triangular blocks are needed for H_{pp} and H_{Schur}. Larger-scale problems, such as the 52 image, 64053 landmark Venice dataset from BAL [3], generate as many as 9373671 non-zeros in the H_{pl} matrix. Figure from Gopinath, Dantu, and Ko [11] © 2023 IEEE.

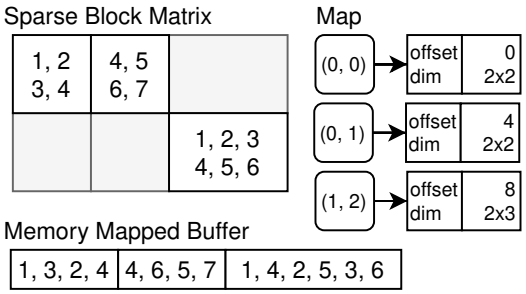


Figure 4.2: A sparse block matrix example with three filled-in blocks (top left). A map that represents the blocks (top right). How the matrix is stored in GPU memory (bottom). Figure from Gopinath, Dantu, and Ko [11] © 2023 IEEE.

by-block multiplication, where a block is a submatrix. This follows the design of g2o that divides a matrix into multiple blocks and performs block-by-block multiplications for cache efficiency [5, 49]. A key change in visual-inertial bundle adjustment is that these blocks are not uniformly-sized. The sizes are determined by the input graph's vertices (as in g2o). An example is shown in Figure 4.3. In the example, matrix A is divided into six blocks, where four blocks (A-0, A-4, A-14, and A-18) are 2×2 in size, and the two remaining blocks (A-8 and A-22) are 2×3 in size. Matrix B is divided into three blocks, where the first two blocks (B-0 and B-4) are 2×2 in size, and the last block (B-8) is 3×2 in size. These two matrices are multiplied and the output is written to matrix C with two blocks of the same size (2×2).

These block multiplications provide a natural opportunity for parallelization using a GPU, where block-by-block multiplications execute in parallel. However, there are two challenges with GPU parallelization, which our work queue design addresses. First, there are block multiplications that write to the same destination block, which means that we need to handle data races. For example, in Figure 4.3, tasks A-0 $\times$ B-0, A-4 $\times$ B-4, and A-8 $\times$ B-8 all write to block C-0. Second, each block-by-block multiplication needs to be carried out by a compute shader, and there is a cost associated with setting up a shader and dispatching it for GPU execution. Thus, it is important to manage this cost carefully.

We address these challenges by creating multiple work queues where block multiplication tasks that satisfy the following criteria altogether get inserted into the same work queue—(i) tasks that write to the same destination block, and (ii) tasks that handle the same left and right block dimensions. Work queues with the same block dimensions share one shader and each can execute in parallel. Within the same work queue, each task is executed in a serialized fashion, one by one. Since we insert tasks that write to the same destination block in a single work queue (hence serialized), we avoid data race problems. Batching work queues with the same left and right block dimensions also reduces the overhead from setting up and dispatching new shaders, which are specialized with constant values for loop bounds. However, there are cases where tasks that write to the same destination block must

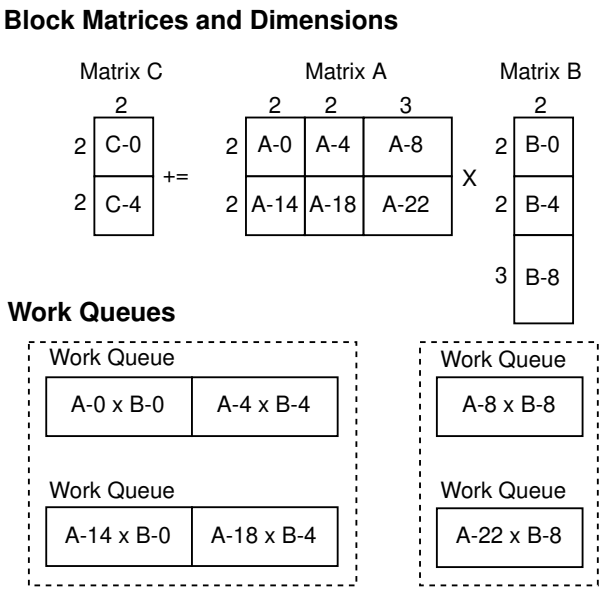


Figure 4.3: Multiplication between sparse block matrices generates a set of work queues. Work queues which multiply blocks of the same dimensions and do not write to the same block in matrix C are grouped together for processing by a single shader dispatch. Figure from Gopinath, Dantu, and Ko [11] © 2023 IEEE.

handle blocks with different dimensions, and thus get inserted into different work queues. Since these cases still have data race problems, we use pipeline barriers to manually enforce serialization between shader dispatches.

Following these rules, for the example in Figure 4.3, we create four work queues. The first work queue has two tasks, A-0 $\times$ B-0 and A-4 $\times$ B-4, which write to the same destination block, C-0, and have the same dimensions. The second work queue has one task, A-8 $\times$ B-8. Though it writes to the same C-0 block as the tasks in the first queue, the dimensions are different, which is why we use a different work queue. The work queue generation process is repeated for the second block, C-4, producing two more work queues. As mentioned, a pipeline barrier is inserted to introduce memory and execution dependencies between each batch of work queues, preventing data races when accessing values in each destination block. The reason we use pipeline barriers rather than finer-grained floating point atomic operations is that these operations require an extension [16], which to the best of our knowledge, is only available for devices with CUDA support. In order to support other GPUs, these extensions are not used. An additional advantage of using work queues is that they reduce the amount of reads and writes to the destination matrix.

Shader Setup Pipelining: To further mitigate the cost of setting up shaders for GPU execution, we pipeline this process along with the operations that produce the input for the Schur complement step. For Algorithm 1, the inputs H_{pp}, H_{pl}, H_{ll}, b_p, and b_l are computed on the CPU. Next, steps 1-3 also execute on the CPU and initialize memory used for

computing and storing the reduced system. Thus, for the first block solver iteration, we hide the latency of shader set-up, work queue generation, and recording of GPU commands by executing these in parallel, mainly while building the linear system and with steps 1-3. This allows us to execute steps 4-7 on a GPU as soon as the input is ready.

Memory Allocation: We have discovered that GPU memory allocation strategies play a significant role in the overall performance of our implementation for local visual-inertial bundle adjustment. When allocating GPU memory on-demand as needed, we have observed performance degradation over time, more noticeably across longer SLAM sequences. We have then switched our GPU memory allocation strategy to the TLSF allocation algorithm [50] as provided by VulkanMemoryAllocator (VMA) [51], which allows for suballocation of existing GPU memory allocations. This allocation algorithm has shown to be robust to long-term uses and does not show any performance degradation. The effect on the block solver execution time is shown in Section 4.4.

4.3 Implementation Details

This section provides additional details regarding the pipelining of shader setup, how work queues are generated and stored in GPU buffers, and how matrix operations are carried out using shaders. Specifics regarding matrix construction and GPU buffer memory allocation are also outlined.

4.3.1 Pipelining

Figure 4.4 shows how tasks that are responsible for initializing the block solver are pipelined. In the diagram, each task is represented by a node. Tasks may also be further decomposed into subtasks, which may be executed in parallel, but these are not shown. A description of each task is given below:

- **Resize and Reserve:** The sizes of the block-rows and block-columns for each sparse block matrix are determined by scanning the vertices of the graph, as in the original g2o implementation. Next, the sparse block matrix data structures are allocated and initialized with these sizes. GPU buffers for vectors involved in the block solver computations are allocated during this task. Lastly, blocks are reserved for H_{pp}, H_{pl}, and H_{ll} according to the structure of the BA graph. Reserving a block designates that part of the matrix as filled-in, creating appropriate block indices, and ensuring that memory for its values will be allocated.

- **Allocate 1:** This task is responsible for creating buffers and allocating memory of the specified type, which is dependent on the platform. It also generates the appropriate commands for synchronizing host-visible memory with device-local memory and zeros the host-visible part if needed. Later, this task is extended to set up shaders and

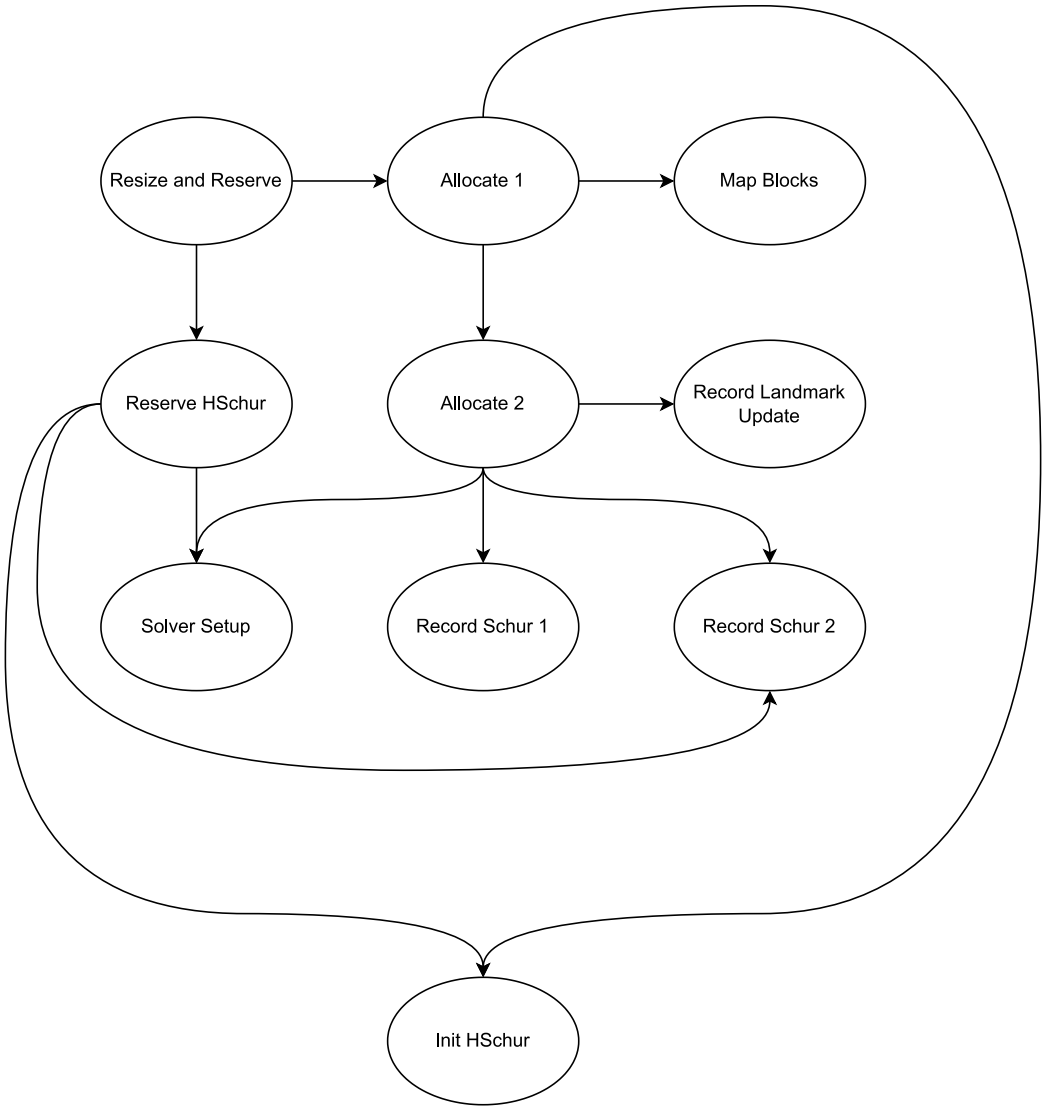


Figure 4.4: Visualization of task dependencies when pipelining the block solver setup.

commands for applying the damping factor to the diagonals of H_{pp} and H_{ll} (discussed in Chapter 5).

- **Allocate 2:** GPU memory for H_{ll}^{-1} and $H_{pl}H_{ll}^{-1}$ is allocated in this task.
- **Record Schur 1:** The shaders and commands for the inversion of the diagonal blocks in H_{ll} are prepared in this task. For explicit Schur elimination, two multiplication operations, one for computing $H_{pl}H_{ll}^{-1}$ and another for computing b_{Schur} are also recorded.
- **Record Schur 2:** For explicit Schur elimination, we generate work queues and set up appropriate shaders for the multiplication between $H_{pl}H_{ll}^{-1}$ and H_{pl}^{T}. This is an expensive process, especially for larger BA problems, and we will explore further

optimizations in Chapter 5, along with less costly implicit methods. For implicit Schur elimination, we record matrix-vector multiplications for computing b_{Schur}.

- **Record Landmark Update:** This task is responsible for setting up the multiplication operations necessary for computing the landmark update Δx_l, corresponding to steps 9-10 in Algorithm 1.
- **Solver Setup:** Any steps necessary for setting up the linear solver are performed by this task. While we do not require any special set up for direct solving methods (i.e. LDLT, LLT), we record matrix vector operations for solvers based on the preconditioned conjugate gradients method (discussed in Chapter 5).
- **Reserve HSchur:** This task generates the structure for H_{Schur} based on the BA graph. It also simultaneously sorts the row and column indices while allocating GPU memory in parallel. The sorted indices are used for outputting the upper triangular of H_{Schur} in compressed sparse column (CSC) format when solving for Δx_p using direct methods.
- **Init HSchur:** For explicit Schur elimination, this task is responsible for performing $H_{Schur} = H_{pp}$ and records synchronization operations for making values of H_{Schur} visible to the host for the linear solver step.
- **Map Blocks:** Active vertices and edges in the graph are mapped to host-visible memory for H_{pp}, H_{pl}, and H_{ll}. This allows g2o to compute Hessian blocks directly in memory that is either visible to the GPU or memory that supports transfers into device-local memory, depending on the memory allocation scheme which is used.

4.3.2 Work Queue Generation

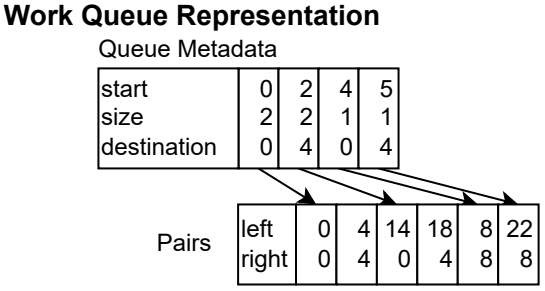


Figure 4.5: How work queues are stored in GPU-accessible arrays.

Following the work queue example in Figure 4.3, we show how work queues are concretely represented in Figure 4.5. Each pair in the *Pairs* array represents a task, consisting of the locations of the left and right block matrices which are to be multiplied together.

These locations are offsets into their corresponding value arrays. The *Queue Metadata* array contains information needed to process all the tasks in a work queue. The *start* field corresponds to the location of the first pair in the pairs array, while *size* is the number of pairs in the queue. The *destination* field is the offset of the target block matrix in the array of values for the destination matrix.

The work queue generation algorithm varies for each of the operations discussed in Section 4.3.3. A general description is given in Algorithm 2. In this description, we denote the left matrix as matrix A, the right matrix as matrix B, and the destination matrix as matrix C. We only generate work queues for blocks in matrix C that have been reserved. This is an important optimization inherited from g2o since it only reserves upper triangular blocks for H_{Schur} due to its symmetry. For Algorithm 2, we also assume that A and B have compatible dimensions, but this must be checked at runtime. The algorithm iterates through each of the reserved blocks in C and generates the list of block pairs for each unique combination of block dimensions. This is represented with the $blockPairs$ map, where each key is four integers representing the left and right block dimensions. The pairs are written into a list of pairs for the entire operation (line 9 of Algorithm 2) and the corresponding metadata is written into a map of lists, where each list is for a different combination of block dimensions (line 8). For matrix sizes observed in local bundle adjustment (Figure 4.1), the overhead of work queue construction is low.

The process for generating and binning work queue pairs is described in Algorithm 3. As before, each list of pairs in the $blockPairs$ map corresponds to a single work queue. The algorithm traverses a sparse block-row of matrix A, where the block-row matches the block-row of block c. Each of these blocks is designated as a. It then checks for the existence of a block b in matrix B, where the block-row matches the block-column of a, and the block-column matches that of c. If a match is found, the offsets of a and b are appended to the list of pairs for the corresponding block dimensions.

Next, Algorithm 4 shows how we generate GPU commands to execute work queues. We use an object m to represent the sequence of operations that will be recorded. This serves as an abstraction for a Vulkan command buffer and handles compute shader creation as well. For each pair of block dimensions, the algorithm copies the queue metadata into the $gmetadata$ buffer and records a single shader dispatch along with a barrier operation. The pairs for all block dimensions are already stored contiguously in correct order due to Algorithm 2, and thus can be copied into $gpairs$ with a single call to std::memcpy.

4.3.3 Linear Algebra Operations

In order to implement *Algorithm* 1, we develop techniques for performing operations involving matrices and vectors using compute shaders. These operations include matrix-matrix multiplications, matrix-vector and vector-matrix multiplications, and block diagonal matrix inversion. These are described in further detail next.

Algorithm 2 Work Queue Generation

Input: A, B, C
Output: Map of lists *metadata* and *pairs* list

for all block c in matrix C **do**
  $blockPairs \leftarrow$ getBlockPairs(c)
  for all $(dims, dimPairs) \in blockPairs$ **do** ▷ Iterate over keys and values
    $start \leftarrow pairs$.size() ▷ Set offset of first pair in work queue
    $size \leftarrow dimPairs$.size() ▷ Set number of tasks in work queue
    $destination \leftarrow c$.offset ▷ Set write location
    $md \leftarrow Metadata(start, size, destination)$
    $metadata[dims]$.append(md)
    $pairs$.concatenate($dimPairs$)
  end for
end for
return *metadata*, *pairs*

Algorithm 3 Pair Binning

Input: Sparse block matrices A, B, and block c
Output: Map of lists of pairs $blockPairs$

for each filled-in block $a \in A$.blocksAtRow(c.blockRow) **do**
  $b \leftarrow B$.blockAt(a.blockColumn, c.blockColumn)
  if $b \neq nil$ **then**
    $p \leftarrow$ Pair(a.offset, b.offset)
    $blockPairs[(a.\text{dim}, b.\text{dim})]$.append($p$)
  end if
end for
return $blockPairs$

Matrix-Matrix Multiplication: The block solver primarily relies on two types of matrix multiplication. The first is multiplication, where the matrix on the right side is transposed. For step 6 of Algorithm 1, rather than compute and store H_{pl}^T, the implementation treats the block-rows of the right matrix as block-columns when constructing work queues. In order for this to work, the behaviour of the multiplication shader program is altered using specialization constants. First, the right-hand side block matrix dimensions (the number of rows and columns) are swapped. Next, we set a boolean constant which indicates that the right blocks are transposed. This boolean parameter is *transposeRight* in Algorithm 4. It modifies the behaviour of the shader to read the right-hand side block matrices as if they were stored in a row-major order, instead of the default column-major order used by Eigen. The difference between each memory layout is shown in Figure 4.6. The second type of matrix-matrix multiplication we implement is for the case where one matrix is block diagonal. We use this for computing $H_{pl}H_{ll}^{-1}$. Although storing another H_{pl}-sized matrix is expensive, we do this to avoid recomputing the same blocks in $H_{pl}H_{ll}^{-1}$ when computing $H_{pl}H_{ll}^{-1}H_{pl}^T$. Work queue generation is simplified for block diagonal matrices. Due to the matrix structure, we can omit queue metadata (Figure 4.5) entirely as an optimization, and only generate left and right pairs, since there is only one block per block-row or block-column in a block diagonal matrix.

Algorithm 4 Recording Work Queue Dispatch

Input: A, B, C, $metadata$, $pairs$, add, $transposeRight$
Output: Multiplication object m

Create GPU buffers $gmetadata$, $gpairs$
Copy $pairs$ into $gpairs$
Initialize m with $gmetadata$, $gpairs$, A.buffer, B.buffer, and C.buffer
$writeLoc = 0$
for all ($dims$, $metaList$) $\in$ $metadata$ **do**
    $firstLoc = writeLoc$
    Copy $metaList$ into $gmetadata$ at $writeLoc$
    $n \leftarrow metaList$.size()
    $writeLoc \leftarrow writeLoc + n$
    m.recordShader($dims$, $firstLoc$, n, add, $transposeRight$)
    m.insertBarrier()
end for
return m

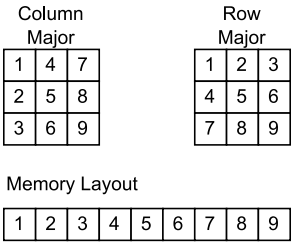


Figure 4.6: How column major and row major blocks are laid out in memory.

Matrix-Vector and Vector-Matrix Multiplication: As per Algorithm 1, the block solver must compute products between matrices and vectors. Matrix-vector multiplication uses the same shaders as matrix-matrix multiplication. The primary change in adapting these shaders for vectors is in how work queues are generated. Since the vectors are dense, it is necessary to manually calculate the read and write offsets of the source and destination vectors respectively. Another variant used by the block solver is vector-matrix multiplication, where the vector is on the left-side of the multiplication. This is implemented for calculating $H_{pl}^T \Delta x_p$, which we compute as $(\Delta x_p^T H_{pl})^T$, following the design of g2o. Transposing the input and output vectors is free, since the memory layouts do not change.

Block Diagonal Matrix Inversion: We invert sparse block diagonal matrices, such as H_{ll} (step 4 of Algorithm 1) by inverting each block in parallel. We do this for block matrix sizes up to 4×4 using built-in GLSL functions. This is sufficient, since visual bundle adjustment uses map points with three parameters for landmark variables, resulting in 3×3 Hessian blocks in H_{ll}. The block inversion shader uses specialization constants to specify the block size, which can be used to optimize away unused functions for other block sizes at the time of pipeline creation. The inversion shader requires offsets to access each block. Unlike the multiplication shaders, which compute each element of a destination block independently, each shader invocation in the inversion shader is responsible for loading and inverting an entire block.

4.3.4 Matrix Multiplication Shader

As previously mentioned, specialized compute shaders are dispatched to carry out block-by-block matrix multiplications on a GPU. Each work queue in a batch can be processed independently by a different *workgroup*. Although tasks in a work queue are processed one by one, the shader implementation further divides the workload of each task over the invocations in a workgroup. Therefore, each element of the destination matrix can be calculated in parallel. A visual description of this is provided in Figure 4.7. For more details on the compute shader execution model, please refer to Section 2.2.

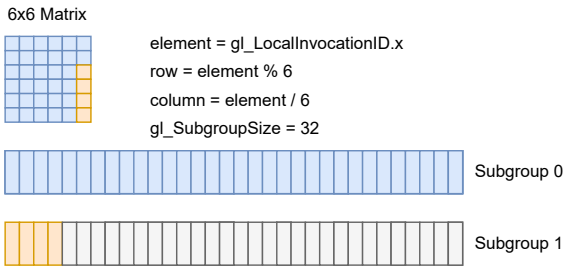


Figure 4.7: How work is distributed in the matrix multiplication shader for a single workgroup. The 6×6 matrix shown corresponds to the destination matrix. In this example, each subgroup consists of 32 invocations. The unshaded invocations are inactive.

The process of initializing a shader can be relatively expensive, since the pre-processed bytecode (discussed in Section 2.2) undergoes additional processing by the Vulkan implementation at runtime. This cost can increase due to the use of additional specialization constants. As mentioned previously, the multiplication shader, as well as other shaders, use specialization constants to encode the dimensions of block matrices into the program itself and modify behaviour through static conditionals. To speed up this process, we modify Kompute to use a single pipeline cache, so that the results of pipeline creation can be reused [16].

Since the values of each block matrix are stored compactly together for cache-efficiency, blocks are not loaded into shared memory. In our testing, we observed that storing these blocks in shared memory, and synchronizing access with a barrier across the workgroup, slowed down multiplications for small-sized local-inertial BA matrices. For these reasons, our design does not use shared memory.

There are inefficiencies with this design. The first is that tasks in a work queue are processed in serial by a workgroup. However, it is only when we read from or write to an element in the destination block that we must worry about data races. The block-by-block multiplications themselves can be executed in parallel. Another inefficiency is that many invocations are left inactive in the second subgroup, due to the dimensions of the destination matrix. An earlier design used subgroup operations to distribute different pairs of a work

queue across invocations in a subgroup, but was also abandoned due to poor performance. We later revise how blocks are multiplied in Chapter 5.

4.3.5 Matrix Construction

Sparse block matrices must undergo multiple initialization steps before use. We optimize for bundle adjustment workloads under the assumption that the structure of the graph is static, meaning that filled-in blocks of H are not added or removed after initial setup for a bundle adjustment function call. This is tied to the allocation model we use, since creating buffers and allocating backing memory is expensive even when using the suballocation strategy described in Section 4.2. As a consequence, the BA graph must be processed twice. The first pass is for reserving blocks, while the second pass is for mapping blocks to the graph. These two passes are also separated by an allocation stage, in which storage is acquired for the non-zero values in each matrix. As in the g2o implementation, the dimensional offsets of the block-rows and block-columns must be initialized using the sizes of the sorted pose and landmark variables.

Reserving block matrices in the first pass involves updating the tracking information stored in the sparse block matrix. Reserving a block creates a record in the matrix of the block's dimensions and offset (shown in Figure 4.2). It is also possible to determine the dimensions of a block without this information, by taking the difference between adjacent block-columns and block-row offsets, but we cache these for convenience. After updating the map, we store two integer indices, one for the corresponding block-row and corresponding block-column. Lastly, we update the total number of values in the overall matrix.

Buffer allocation is performed for all values in a sparse block matrix at once. The size of the buffer is determined using the total number of non-zeros computed when reserving blocks. Depending on the platform, we allocate memory differently. One option is a single-buffer scheme, intended for platforms with integrated GPUs. For platforms with discrete GPUs, we use a two-buffer scheme. These are described in further detail in Section 4.3.6.

The second pass through the BA graph is responsible for mapping memory for Hessian blocks back to their corresponding vertices and edges. In the original g2o block solver, this is not a separate pass, since memory can be allocated for each block in the matrix on-demand. To map a block, we look up the block offset and return a pointer to the memory-mapped buffer at that offset. This pointer is stored and used by g2o to construct an `Eigen::Map` which can be accessed to update values in the matrix directly.

The underlying reason why this second pass is necessary is that all non-generic computations, such as the application of the parameter update, as well as the error and Jacobian matrix calculations, are still performed by the host. This design is what allows the revised block solver to function as a drop-in replacement for different types of bundle adjustment problems. Still, these computations can benefit from g2o's existing parallelization scheme, which uses OpenMP.

4.3.6 Memory Allocation

For systems such as the Jetson Xavier NX, on which the GPU shares memory with the host, the block solver is configured to exclusively use host-visible, host-coherent memory. Therefore, on these systems, it is not necessary for the block solver to flush or invalidate mapped memory, in order to maintain up-to-date visibility of values. Moreover, since all buffers are backed by host-visible memory, explicit data transfers for device access are eliminated completely.

For systems with discrete GPUs, on which dedicated memory is available in large amounts, we use a two-buffer scheme for shader storage buffer objects. We extend the functionality provided by Kompute [46] to introduce a new tensor type, `DeviceCached`, which is based on the existing `Device` tensor type. Here, tensor refers to a Kompute data structure which serves as a wrapper around a Vulkan buffer and its allocated memory. The first buffer is backed by host-visible, host-cached memory and is used for calculations by the host. While this type of memory needs special handling to ensure the visibility of written values, it is much faster to use for host operations, such as for building the linear system (computing H and b), than host-coherent memory. The second buffer is allocated with device-local memory for efficient device access [16]. This allocation strategy assumes that this memory is not host-visible, although in practice this may not always be the case. Using this two-buffer strategy adds additional overhead, since it becomes necessary to transfer values between the host-visible buffer and the device-local buffer for each LMA iteration. Even with this overhead, our evaluation on the desktop machine in Section 4.4 shows that this strategy performs well.

Matrices such as H_{ll}^{-1} store intermediate values that do not need to be accessed by the host. For such values, we use the `Storage` tensor type, which creates a single buffer using one of the GPU-accessible memory types.

4.3.7 Linear Solver

As mentioned in Section 4.2, we use Eigen sparse LDLT decomposition to solve for Δx_p on the CPU, as in the original implementation. This introduces additional overhead, since H_{Schur} must first be copied back to the CPU, and then converted into an upper triangular CSC matrix. There are a few reasons as to why this step is still executed on a CPU. Firstly, the dimensions of H_{Schur} are constrained, either explicitly by the keyframe limit for local-inertial bundle adjustment, or implicitly due to the number of co-visible keyframes available. This results in relatively little time spent on the linear solver step, as shown in Section 4.4, thus implementing GPU acceleration may yield little benefit. For larger BA problems, we implement preconditioned conjugate gradients instead, in Chapter 5, which allows for memory-efficient implicit evaluation [3, 15, 29]. The development of a custom LDLT solver is left as an area for future work, as discussed in Chapter 6.

4.4 Evaluation

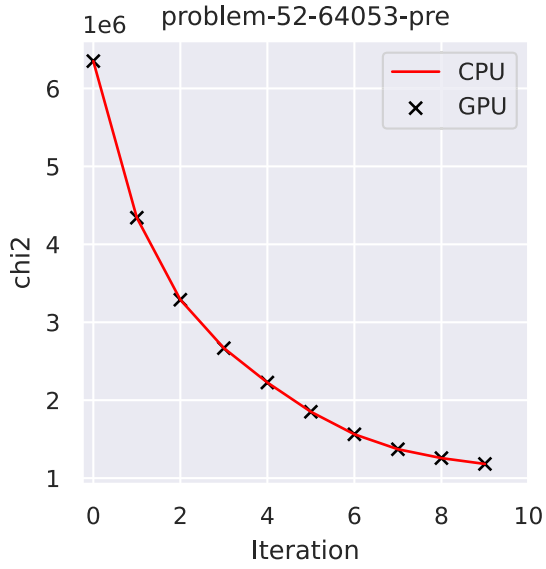


Figure 4.8: χ^2 error when we run the original CPU version and our GPU version of block solver on the Venice BAL dataset for ten iterations. The behaviours match each other's. Figure from Gopinath, Dantu, and Ko [11] © 2023 IEEE.

4.4.1 Experimental setup

We implement our techniques as a drop-in replacement block solver for g2o and integrate our modifications into ORB-SLAM3. We evaluate the runtime performance of the original CPU block solver (from g2o) and the GPU-enabled block solver (our own) and measure GPU memory usage. Vectorization is enabled and OpenMP is disabled for both solvers, which use double-precision. We run experiments on a desktop machine with an 8-core AMD Ryzen 5800X processor operating at 3.8 GHz and 32 GB of RAM, using an NVIDIA RTX 3080. We also use an NVIDIA Jetson Xavier NX board with a 6-core ARM Carmel CPU running in the 15W, 6-core power mode at 1.4 GHz, with 8 GB of RAM, and a Volta GPU.

We process a mix of indoor and outdoor sequences from EuRoC and TUM-VI datasets using ORB-SLAM3 in the stereo-inertial configuration. To demonstrate that our implementation performs correctly, we compare the reduction in error for a BAL dataset in Figure 4.8 and observe matching behaviour.

4.4.2 Block Solver Performance

There is consistent performance improvement across all the sequences on both machines with our GPU solver. On the desktop, our method achieves a speedup of 5.08× for the Schur complement step before the linear solver step on V201, and a 2.87× speedup on the

Sequence	CPU Time (ms)	GPU Time (ms)	Avg Diff. (%)
MH01	52.66 ± 21.52	35.97 ± 15.92	-31.69
MH02	45.77 ± 18.69	32.56 ± 12.90	-28.86
MH03	55.69 ± 19.51	40.53 ± 14.10	-27.22
MH04	50.99 ± 13.92	37.40 ± 9.47	-26.66
MH05	52.03 ± 13.12	37.17 ± 9.31	-28.56
V101	61.49 ± 13.77	43.22 ± 9.40	-29.72
V102	55.74 ± 15.92	41.03 ± 12.70	-26.39
V103	49.75 ± 14.07	34.18 ± 9.27	-31.30
V201	53.85 ± 13.32	35.65 ± 8.30	-33.79
V202	54.59 ± 12.84	38.20 ± 9.84	-30.03
V203	35.07 ± 11.79	25.97 ± 8.51	-25.96
outdoors1	35.08 ± 19.05	27.55 ± 12.82	-21.47
outdoors2	41.74 ± 22.07	32.03 ± 14.78	-23.27
outdoors3	46.37 ± 16.56	37.20 ± 11.98	-19.78
outdoors4	38.93 ± 23.07	29.08 ± 15.08	-25.30
outdoors5	58.15 ± 17.99	45.39 ± 14.34	-21.95
outdoors6	51.69 ± 16.70	42.55 ± 14.75	-17.67
outdoors7	43.07 ± 14.28	34.17 ± 10.73	-20.66
room1	75.85 ± 18.29	60.47 ± 17.04	-20.28
room2	70.25 ± 19.70	56.97 ± 17.49	-18.90
room3	75.91 ± 18.00	63.56 ± 18.36	-16.27
room4	68.40 ± 18.35	52.25 ± 15.30	-23.61
room5	75.65 ± 15.87	61.08 ± 15.51	-19.25
room6	73.85 ± 17.20	54.79 ± 15.04	-25.81

Table 4.1: Average local BA run times (in ms) for ORB-SLAM3 on the desktop machine. From Gopinath, Dantu, and Ko [11] © 2023 IEEE.

Jetson (Figure 4.9). Likewise, there is a significant improvement for the landmark update calculation, while the linear solver step performs similarly despite the readback overhead for GPU memory.

4.4.3 Overall Performance of Local Bundle Adjustment

Table 4.1 and Table 4.2 show the running times of the local-inertial bundle adjustment averaged over five trials. Results show that our GPU solver *consistently reduces* the average local BA execution time for both platforms. As the workload for local BA is partly determined by visible landmarks and co-visible keyframes, there are large variations in each trial for both solvers. The average performance improvement in local BA (decrease in time) using our GPU solver ranges from 16.27% to 33.79% on the desktop, and 13.81% to 26.68% on the Jetson Xavier NX. Figure 4.10 shows an averaged breakdown of local-inertial bundle adjustment for EuRoC V201. The block solver performance improvement is reflected in the segment for optimization.

Sequence	CPU Time (ms)	GPU Time (ms)	Avg Diff. (%)
MH01	290.11 ± 90.14	218.14 ± 71.84	-24.81
MH02	266.54 ± 87.47	198.80 ± 64.99	-25.41
MH03	293.20 ± 82.64	229.28 ± 62.79	-21.80
MH04	259.37 ± 63.69	201.51 ± 46.47	-22.31
MH05	265.93 ± 58.42	204.59 ± 44.35	-23.07
V101	319.78 ± 67.03	242.61 ± 50.34	-24.13
V102	279.96 ± 76.11	223.42 ± 60.53	-20.20
V103	232.96 ± 60.27	184.85 ± 42.39	-20.65
V201	270.85 ± 54.41	198.60 ± 36.17	-26.68
V202	263.80 ± 57.52	209.39 ± 46.17	-20.63
V203	156.71 ± 67.21	132.58 ± 49.78	-15.40
outdoors1	174.06 ± 79.79	139.35 ± 53.19	-19.94
outdoors2	194.50 ± 96.58	156.31 ± 66.33	-19.63
outdoors3	194.88 ± 87.45	159.08 ± 59.54	-18.37
outdoors4	189.94 ± 95.33	150.47 ± 66.36	-20.78
outdoors5	268.61 ± 93.56	214.33 ± 67.61	-20.21
outdoors6	227.14 ± 66.35	188.45 ± 50.36	-17.03
outdoors7	210.96 ± 65.75	169.21 ± 45.45	-19.79
room1	349.89 ± 83.33	288.99 ± 70.97	-17.40
room2	351.81 ± 98.73	284.01 ± 84.96	-19.27
room3	321.51 ± 73.31	277.10 ± 69.85	-13.81
room4	314.02 ± 77.29	254.69 ± 67.64	-18.89
room5	342.29 ± 76.55	284.34 ± 64.29	-16.93
room6	355.83 ± 85.74	284.19 ± 68.85	-20.13

Table 4.2: Average local BA run times (in ms) for ORB-SLAM3 on the Jetson Xavier NX. From Gopinath, Dantu, and Ko [11] © 2023 IEEE.

4.4.4 GPU Memory Usage

Figure 4.11 shows the average amount of memory occupied by GPU buffer allocations for local-inertial BA, totalled over all memory heaps. Memory usage is larger on the desktop machine due to the use of device-local memory. On both machines, TUM-VI outdoors sequences use the least amount of memory, while EuRoC sequences use the most, possibly due to differences in co-visible keyframes and the sparsity of features. We observe that the largest memory usage is for the V101 sequence and the smallest for outdoors1.

4.4.5 Effect of Memory Allocation Strategies

As mentioned in Section 4.2, we have discovered that the GPU memory allocation method influences the performance of bundle adjustment, especially for longer SLAM sequences. Figure 4.12 shows the difference between on-demand allocation (allocating memory for a buffer as needed) and TLSF allocation using VMA (suballocating memory from an existing block). As shown, on-demand allocation exhibits performance degradation over longer sequences. The reason is that, in the first iteration of each optimization, the block solver must wait increasingly longer for allocations to finish. Suballocating memory avoids this problem, which leads to better performance.

Figure 4.9: Average execution time of the main solving steps for an iteration of the block solver for the EuRoC V201 sequence. Figure from Gopinath, Dantu, and Ko [11] © 2023 IEEE.

4.4.6 Threats to Validity

The findings of our research are largely based on our analysis of two datasets, EuRoC and TUM-VI, as well as our implementation based on g2o and ORB-SLAM3. Though we have reasons to believe that our results are generally applicable to other platforms and scenarios (see our motivation in Section 4.2), they need to be interpreted in the context laid out in this paper.

For local visual-only BA problems in which the sizes of the pose variables are uniform, fixed-size matrices are used, which may allow the CPU block solver to benefit more from vectorization. Bottlenecks which must be resolved for global BA include work queue generation, building the linear system, and the linear solver step. For large problems, implicit iterative methods [15, 29] have been shown to be more suitable for GPU execution.

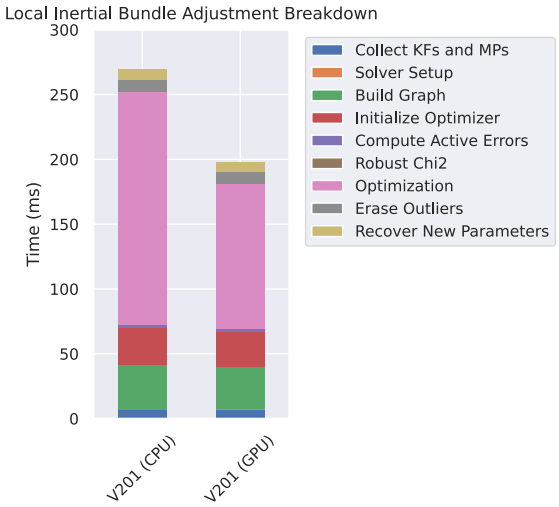
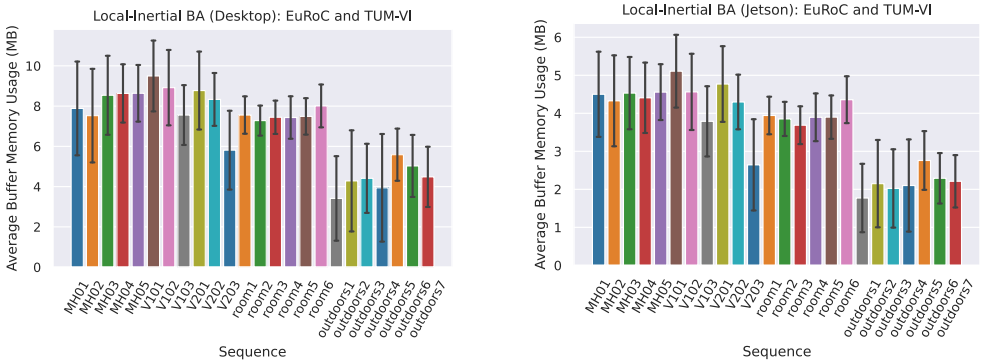


Figure 4.10: A breakdown of the local-inertial bundle adjustment call for a run of EuRoC V201 using each solver on the Xavier NX. Figure from Gopinath, Dantu, and Ko [11] © 2023 IEEE.



(a) Desktop (b) Jetson

Figure 4.11: Average memory usage of buffer allocations (total across all heaps) as reported by VulkanMemoryAllocator for various sequences over five runs. The memory is suballocated from larger blocks which are reserved during initialization. The allocator reserves approximately 100.66 MB on the desktop system and 33.55 MB on the Jetson. Figure from Gopinath, Dantu, and Ko [11] © 2023 IEEE.

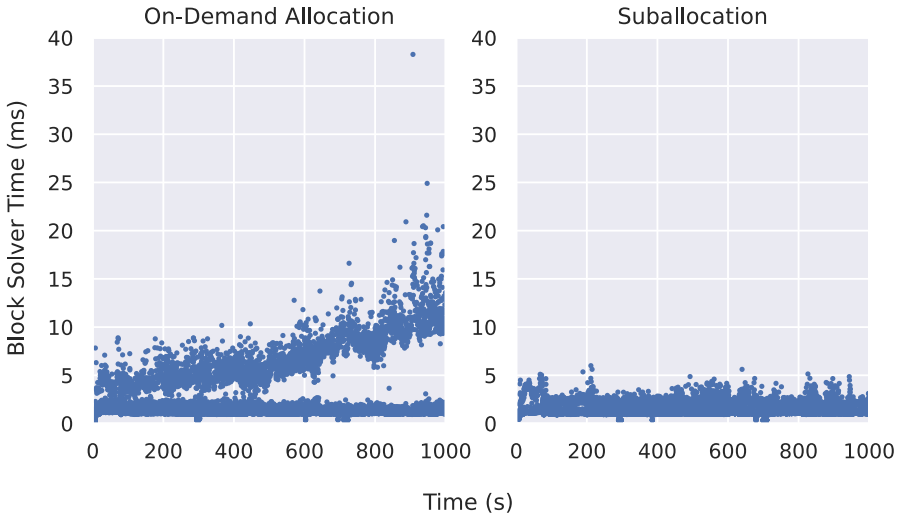


Figure 4.12: The impact of the GPU memory allocation method on the block solver performance (desktop), for local-inertial BA on the TUM-VI outdoors6 sequence (first 1000 seconds). Creating on-demand allocations for each buffer as needed results in performance degradation over longer sequences. Using VMA to suballocate memory from larger blocks avoids this overhead. Figure from Gopinath, Dantu, and Ko [11] © 2023 IEEE.

Chapter 5

GPU Acceleration for Global Bundle Adjustment

5.1 Introduction

We now turn our attention to how we can extend our techniques to accelerate global bundle adjustment for visual-inertial SLAM. Specifically, we focus on improving the performance of our implementation for visual BAL problems [3] and also for full-inertial BA workloads generated by ORB-SLAM3 [1]. As we saw in Chapter 4, it is possible to execute some block solver calculations on a GPU and obtain a moderate speedup for local-inertial bundle adjustment problems, which involve operations on small to medium-sized sparse matrices. In this chapter, we modify our approach to improve the scalability of the block solver and handle larger matrices generated by full-map optimization. We show the sizes of such matrices, generated by executing indoor and outdoor EuRoC and TUM-VI sequences with ORB-SLAM3 in Figure 5.1. These larger matrices pose a different set of challenges which impact performance, as we describe next.

Work Queue Generation Time: We have seen for small workloads that the amount of time needed to generate work queues, especially for the multiplication between $H_{pl}H_{ll}^{-1}$ and H_{pl}^{T}, is not a major bottleneck due to the pipelining of the setup as described in Chapter 4. This does not hold for large bundle adjustment problems, since the upper bound for the number of work queues which must be generated for matrix-matrix multiplication grows non-linearly when the dimensions of the destination matrix increase. Therefore, we consider three options. First, we can reduce the amount of time spent on work queue generation, mainly for computing $H_{pl}H_{ll}^{-1}H_{pl}^{T}$, by avoiding this exact operation entirely. We can achieve this by switching to implicit methods instead, when using an iterative linear solver. Alternatively, we can try to avoid work queue generation for some operations and look for blocks to multiply in the multiplication shader itself. Lastly, we consider improving how we generate work queues by rewriting the generation algorithm to take advantage of GPU hardware. We implement and evaluate all three of these methods, each of which has different tradeoffs.

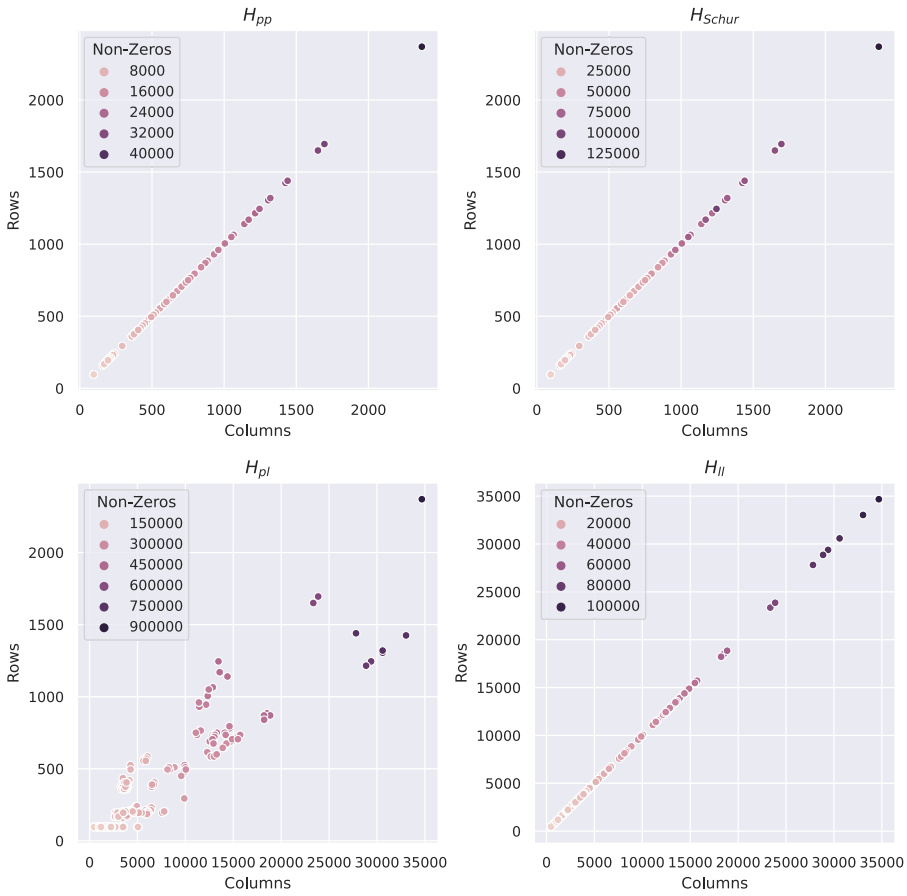


Figure 5.1: The sizes and number of non-zeros of the matrices generated during full-inertial bundle adjustment in ORB-SLAM3. Unlike local-inertial bundle adjustment, the number of keyframes used to construct the BA graph is not capped.

Linear Solver: As we will see, the linear solver step, which uses LDLT factorization in ORB-SLAM3, is the most computationally expensive step for large bundle adjustment problems due to the increased dimensions of H_{Schur}, which grow beyond 2000 rows and columns as shown in Figure 5.1. This is a departure from local visual-inertial bundle adjustment, for which computing the reduced system was the most expensive part of the main block solver steps. Therefore, we examine iterative methods based on conjugate gradients, which have been successfully applied to large-scale bundle adjustment problems using GPUs in the past [15, 29], and investigate their applicability to global visual-inertial bundle adjustment for on-device SLAM.

Memory Usage: The methods we developed for local bundle adjustment problems were designed for workloads consisting of small to medium-sized matrices. It becomes apparent, however, that these methods can become costly when the sizes of matrices such as H_{pl} increase, since we make extra allocations to store the results of intermediate computations including $H_{pl}H_{ll}^{-1}$. Iterative methods allow us to forgo computing and storing the Schur

complement, lowering memory usage. This can affect which preconditioners are readily available, and therefore have consequences for the convergence. Thus, we develop methods to work around these limitations efficiently using shaders that take advantage of shared memory.

We will now take a look at how we can address these challenges in more detail. To evaluate our methods, we use BAL datasets and maps generated by processing EuRoC and TUM-VI sequences with ORB-SLAM3.

5.2 Work Queue Generation

As mentioned, the work queue generation method developed and described in Chapter 4 scales poorly for multiplications between large matrices. While it is possible to avoid work queue generation for computing H_{Schur} by using implicit methods (Section 5.4), we present two alternative methods to address this problem, for when matrix construction cannot be avoided.

5.2.1 Dynamic Matrix Multiplication Based on the Block Compressed Sparse Row Representation

Our first approach uses a new multiplication shader which dynamically determines which blocks should be multiplied together, each time that it is executed. This involves making the indices of filled-in blocks in each matrix available to the shader. We use a representation similar to block compressed sparse row (BCSR) [52]. An example of BCSR is shown in Figure 5.2. In the figure, the difference between consecutive values in the row pointers array corresponds to the number of filled-in blocks in a block-row. For example, the first block-row has $2 - 0 = 2$ blocks, while the second has $3 - 2 = 1$ block. The column indices array stores the block-column of each block, sorted in order for each block-row. A limitation is that the block sizes are uniform across the entire matrix. For this reason, it is not suitable for storing matrices such as H_{pp}, since pose variables may have a varying number of parameters for visual-inertial bundle adjustment.

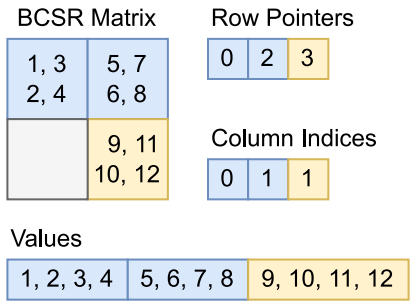


Figure 5.2: An example of a sparse block matrix stored in BCSR format, where values for each block are stored in column-major order. The block sizes are uniform.

We now describe the key details of our dynamic matrix multiplication algorithm, which reuse the existing row and column block indices stored by each sparse block matrix. These indices are populated at the time that each block in the matrix is reserved. They are not compressed. Separate vectors are stored for the indices of the filled-in blocks, for each block-row and block-column.

First, we prepare the indices in a format suitable for in-shader processing. For execution on the GPU, we require these indices to be (i) sorted and (ii) grouped together by the block dimensions. Therefore, we first copy the existing indices, sort them, and then simultaneously convert these indices into a compressed format (either sparse-row or sparse-column depending on the operation) while also storing them separately for each block size. In this scheme, the values array from BCSR is replaced with an array containing the offsets of each same-sized block, into the non-zeros buffer for the entire matrix. Then, these compressed indices are uploaded into GPU buffers. This process is shown in Figure 5.3, where matrix A corresponds to the left operand, and matrix B corresponds to the right operand.

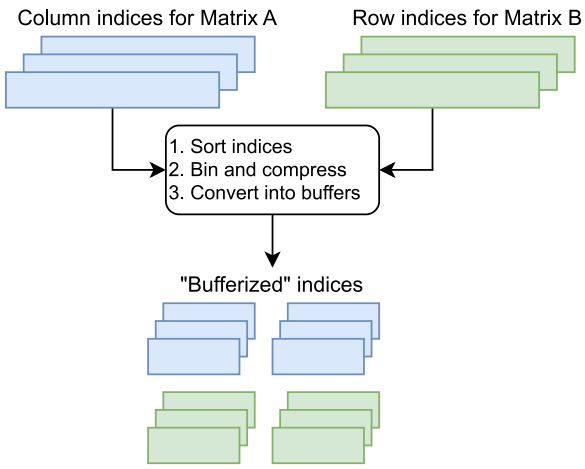


Figure 5.3: How the indices for each matrix are processed. When the right matrix is transposed, column-indices may be used instead.

Using these transformed indices, we record dispatch commands for the dynamic multiplication shader, for valid combinations of left and right block matrix dimensions. This shader is similar to the work queue-based multiplication shader in Chapter 4 except that it simply traverses the indices of both matrices and multiplies blocks when there is a match. Since the indices are separated for different block sizes, the dimensions do not need to be checked in the shader, and are encoded as specialization constants as before. An additional array is used to store the block-row, block-column, and offset of each block in the destination matrix. This information is used to determine which destination block is being computed by the current workgroup, and which indices should be accessed.

5.2.2 Parallel Work Queue Generation

We now describe our alternative GPU work queue generation method. First, we observe that work queues can be generated independently for each destination block and combination of block dimensions. We could rewrite the existing algorithms from Chapter 4 to take advantage of multiple CPU threads, but a better option is to offload this work onto the numerous cores found on modern GPUs. An advantage of this approach is that work queues computed in GPU memory do not need to be transferred from the host. Therefore, we look at how we can achieve work queue generation using compute shaders.

Work queues can be generated in-shader by repurposing the bufferized indices which were generated for dynamic matrix multiplication. Since these indices are already grouped together by block dimensions, grouping the work queues by pairs of block dimensions is simplified. This method uses two new shaders. The first shader is used for *estimation* and the second shader is used for *construction*. Each is described next.

The estimation shader traverses the bufferized indices for each destination block, and determines the exact sizes of the corresponding work queues. The dispatch is organized so that each shader invocation handles one work queue. An invocation increments the number of items for a single work queue when it finds a pair of blocks with matching indices. The number of work queues, and the total number of all items across all work queues, for one pair of block dimensions, are recorded in a buffer representing an allocator. To do this, we use subgroup reductions to sum up the total number of non-empty work queues and items. Then for each subgroup, an elected invocation adds the summed values to the buffer atomically. These sizes are read back on the host and used to allocate buffers for the work queues which are to be written.

The construction shader writes out the work queue metadata and pairs into the newly allocated buffers. The sizes of each work queue are not stored by the previous step and must be determined again by traversing the indices. Then space is suballocated from the buffers at two levels. As before, subgroup-level operations are used to determine the total number of work queues and pairs handled by a subgroup, and the allocator buffer, which has been reset, is updated atomically. The values returned by these atomic operations serve as the starting offsets into the metadata and pairs buffers and are broadcasted to all invocations in the current subgroup. Next, each active invocation in the subgroup (which handles a non-empty work queue) determines its own write offsets using an exclusive scan (via the `subgroupExclusiveAdd` function). This exclusive scan returns the sum of the number of items for active invocations with a lower subgroup invocation ID than the caller's [20]. Lastly, the work queue data is written out to both buffers at these offsets, requiring another traversal of the indices.

After constructing these work queues, we can use our existing work queue multiplication shader to carry out the operation. The main change is that when recording shader

dispatches, the starting offset into the queue metadata buffer is always zero, since we no longer pack work queues for different block dimensions into the same buffer.

	Original	In-shader Lookup	GPU Work Queue Generation
Setup	28.834	**0.823**	2.017
First Multiplication	0.229	1.704	**0.185**

Table 5.1: The timings (in seconds) for setting up the large multiplication operation for computing H_{Schur} and carrying it out for the Final-4585 dataset from BAL [3], averaged over ten runs.

As shown in Table 5.1, although GPU work queue generation takes longer to set up than dynamic matrix multiplication, the multiplication operation becomes much faster for large bundle adjustment problems, since it avoids the overhead of looking up indices and CPU-GPU work queue transfers. Therefore, it is the final method used for preparing the multiplication operation between $H_{pl}H_{ll}^{-1}$ and H_{pl}^T when computing H_{Schur}.

5.3 Linear Solver

With GPU work queue generation, our block solver can now efficiently compute the Schur complement in order to obtain a reduced system, but as discussed in Section 5.7, solving this reduced system can become the most expensive step for global bundle adjustment problems. To restate the problem, we require an optimized linear solver to find the solution of the reduced linear system

$$H_{Schur}\Delta x_p = b_{Schur}$$

where H_{Schur} is a symmetric semi-positive definite matrix. That is, given a vector $v \in \mathbb{R}^n$, $v^T H_{Schur} v \geq 0$. We have seen that for local bundle adjustment, this matrix can be relatively small, in comparison to the entire matrix H. However, for global bundle adjustment and similar problems, H_{Schur} may be much larger, as shown in Figure 5.1, thereby increasing the computation time.

Bundle adjustment implementations often employ direct methods for solving symmetric systems [2]. The main advantage of direct methods is that they allow for the solution to be computed with low error. Direct methods require the matrix of the linear system to be computed and stored in memory. Common choices are Cholesky (LL^T) decomposition and square-root free Cholesky (LDL^T) decomposition.

There are several challenges to consider when using methods based on Cholesky decomposition. To begin with, H_{Schur} must be converted into a compatible format for the solver. In the case of the existing implementation, it must be converted into the compressed sparse column (CSC) format. We must also consider methods to minimize fill-in, which is the increase in the number of non-zeros when decomposing the matrix. This is commonly achieved with a variable reordering method, such as approximate minimum degree [2, 12].

Another challenge is that sparse Cholesky methods are difficult to parallelize, especially for GPUs, due to the dependencies between sub-problems [53].

Iterative methods present an alternative to direct approaches. They can be used to compute inexact solutions very quickly [3]. A well-known, memory efficient choice is preconditioned conjugate gradients (PCG) [3, 5]. A simplified description of PCG is given in Algorithm 5, primarily based on the implementation in g2o [5, 54]. In the algorithm, t is the tolerance and $maxIter$ refers to the maximum number of iterations before the algorithm is terminated.

Agarwal et al. [3] previously demonstrated that PCG can be applied to large visual bundle adjustment problems, without explicitly constructing H_{Schur}. This is achieved by evaluating matrix-vector products implicitly. In fact, it can be observed in Algorithm 5 that no matrix-matrix products are computed. Thus, given a vector $\vec{v}$, we can implicitly evaluate $H_{Schur}\vec{v}$ as

$$H_{Schur}\vec{v} = H_{pp}\vec{v} - H_{pl}(H_{ll}^{-1}(H_{pl}^{T}\vec{v})) \tag{5.1}$$

In practice, implicit evaluation may not yield the same result due to floating point round-off, but it is sufficient for our purposes.

These properties, along with the simplicity of the algorithm, make PCG relatively straightforward to parallelize and execute on GPUs [15, 29]. Another advantage of PCG is that it can directly operate on our existing block representation, improving cache efficiency. Therefore, we choose to implement PCG for our linear solver, described in the next section.

5.4 PCG Implementation

We now describe key parts of our PCG implementation, as well as additional relevant changes. We implement Algorithm 5 using compute shaders. Our solver supports both explicit and implicit methods for Schur elimination, as given by Equation 2.1 and Equation 5.1 respectively. Operations inside the main loop, excluding those for initialization, are recorded into a single command buffer, in order to minimize the cost from queue submission.

5.4.1 Preconditioner

PCG uses a preconditioner to improve the condition number of a linear system [26]. Using a preconditioner is necessary to improve the convergence across PCG iterations, and therefore improve the quality of the computed solution. We mainly consider block-Jacobi preconditioners [3], since they are easy to generate and invert. When PCG is invoked on an explicit matrix representation, we use the block diagonal of that matrix. For the implicit method, we consider both the block diagonal of the H_{pp} matrix which is already available to us, and also the block diagonal of H_{Schur} [3], which can be calculated relatively cheaply

Algorithm 5 Preconditioned Conjugate Gradients

Input: A, $\vec{b}$, $maxIter$, t
Output: $\vec{x}$

$\vec{x}, \vec{q}, \vec{s} \leftarrow \vec{0}$
$\vec{r} \leftarrow \vec{b}$
$M \leftarrow$blockDiagonal(A) ▷ Block-Jacobi preconditioner matrix
$J \leftarrow M^{-1}$
$\vec{d} \leftarrow J\vec{r}$
$d_n \leftarrow \vec{r} \cdot \vec{d}$
$d_0 \leftarrow td_n$
if $absTol$ **then** ▷ Initially $absTol \leftarrow True$
    if $res > 0$ and $res > d_0$ **then** ▷ Initially $res \leftarrow -1$
        $d_0 \leftarrow res$
    end if
end if
for all $i \leftarrow 1$ to $maxIter$ **do**
    if $d_n \leq d_0$ **then**
        return $\vec{x}$
    end if
    $\vec{q} \leftarrow A\vec{d}$
    $\alpha = d_n/(\vec{d} \cdot \vec{q})$
    $\vec{x} \leftarrow \vec{x} + \alpha\vec{d}$
    $\vec{r} \leftarrow \vec{r} - \alpha\vec{q}$
    $\vec{s} \leftarrow J\vec{r}$
    $d_{old} = d_n$
    $d_n = \vec{r} \cdot \vec{s}$
    $\beta = d_n/d_{old}$
    $\vec{d} \leftarrow \vec{s} + \beta\vec{d}$
end for
$res \leftarrow 0.5d_n$ ▷ Preserved for next call
return $\vec{x}$

without separately storing $H_{pl}H_{ll}^{-1}$. The matrices of the block diagonal are inverted block-by-block on the host, since GLSL only supports built-in matrix types up to 4×4, whereas for visual-inertial BA, the matrix sizes can go up to 6×6.

5.4.2 Reduction for Vector Dot Products

We implement parallel reduction for inner products by sharing data at both the subgroup and workgroup level. After reading values from the input buffer, invocations within the same subgroup perform a summation reduction using `subgroupAdd`. Following this, an elected invocation from each subgroup stores the result of the reduction in shared memory for the entire workgroup. Afterwards, invocations of the first subgroup load these values from shared memory, and perform a second subgroup reduction. The resulting value is written out to an intermediate buffer. This method allows a single dispatch of the shader to reduce the number of values down to the number of workgroups used. For increased efficiency, each invocation in a workgroup reads multiple values from the input array. If the number of values to be reduced exceeds the capacity for a single workgroup, then the reduction shader is dispatched multiple times until there are no more values to reduce.

5.4.3 Memory-Efficient Preconditioner Computation for Implicit Schur Elimination

We have developed a method to compute only the block diagonal of H_{Schur} for the implicit mode of the PCG solver, without storing intermediate products such as $H_{pl}H_{ll}^{-1}$, by taking advantage of the following observations. For the structure of these matrices, please refer to Chapter 2. Recalling Equation 2.1, when computing diagonal blocks of H_{Schur}, it is only necessary to multiply the block-row i of $H_{pl}H_{ll}^{-1}$ with block-column i of H_{pl}^{T} to obtain the term which is subtracted from $\text{block}_{i,i}$ of H_{pp}. Next, due to the transposition of H_{pl}, the block-row and block-column have equivalent structure, meaning that it is sufficient to store offset pairs for H_{pl} and H_{ll}^{-1} using regular work queues, rather than triplets for three matrices. Lastly, since H_{ll}^{-1} is block diagonal, for a work queue with n pairs, only $2n$ matrix multiplications are required.

From these observations, we develop a memory-efficient solution. A single shader is used to multiply all three matrices, H_{pl}, H_{ll}^{-1}, and H_{pl}^{T} together. When recording the dispatch operation, the size of the workgroup is increased, so that it is large enough to process each element of a final or intermediate block using a dedicated shader invocation. For example, for a 6×3 H_{pl} block multiplying a 3×3 block from H_{ll}^{-1}, a workgroup size of at least 18 invocations is required to compute this intermediate result. However, we must also consider that the final block subtracted from H_{Schur}, which is initialized with H_{pp}, has a size of $6 \times 6 = 36$ elements. Therefore, we use the maximum of these two quantities to determine the workgroup size.

For processing each task in a work queue, we first compute the block in $H_{pl}H_{ll}^{-1}$ and then use this result to multiply the transposed block from H_{pl}. Since we do not allocate buffer memory for the intermediate product, we temporarily store it in shared memory. This requires the use of a GLSL barrier to make values written to the shared memory become visible to the rest of the workgroup. Thus, we avoid the cost of recomputing elements of $H_{pl}H_{ll}^{-1}$ multiple times.

5.4.4 Workload Distribution for Matrix Multiplication

To process multiplications more efficiently, we revise how tasks are distributed across a GPU according to the type of multiplication. For matrix-matrix multiplications, we utilize larger workgroup sizes to process parts of the same work queue in parallel, shown in Figure 5.4. For matrix-vector multiplications, which involve fewer elements per work queue, a single workgroup now processes multiple work queues (Figure 5.5). This strategy not only impacts small-matrix operations, such as those seen in local bundle adjustment, but also operations for large matrices and vectors generated during global bundle adjustment. This is especially important for PCG, as the impact of the workload distribution for operations involving large matrices and vectors is magnified over several iterations.

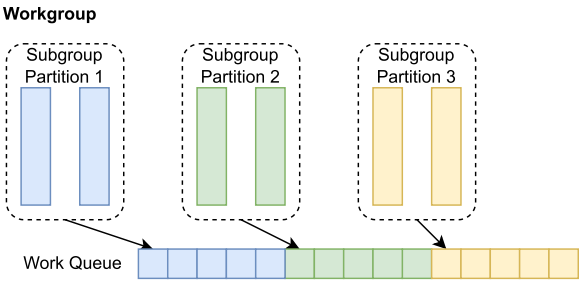


Figure 5.4: How chunks of a work queue are distributed to different subgroups within a workgroup when performing matrix-matrix multiplications. In this example, each partition consists of two subgroups. The results of each partition are stored in shared memory and reduced before being written to the destination matrix.

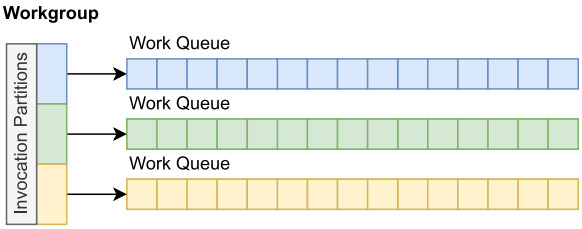


Figure 5.5: How invocations processing different work queues are packed together for performing matrix-vector multiplications.

5.5 Block Solver Improvements

The block solver was modified to better handle larger BA problems. First, we revised our method for initializing H_{Schur} with the blocks from H_{pp}. Previously, these blocks were copied to the mapped buffer by the host. In the revised implementation, we use specialized shaders to copy blocks, thus parallelizing this process. Another change is that the LMA damping factor λ is now added to the diagonals of H_{pp} and H_{ll} in-shader, rather than on the host. Likewise, the backup and restoration of the diagonals are also performed by shaders. These operations are not expensive for local BA workloads, but may become more costly as the number of pose and landmark parameters grow, so the decision was made to parallelize them. Lastly, changes were made to the setup pipelining. Primarily, the implementation defers waiting for the landmark update calculation setup until after the linear solver step.

5.6 Evaluation on BAL Datasets with OpenMP

We first evaluate our methods using Bundle Adjustment in the Large (BAL) datasets [3]. As before, we perform our experiments on two machines. We use a desktop machine with an AMD Ryzen 7 5800X CPU running at 3.8 GHz, with 32 GB of RAM, and an NVIDIA RTX 3080 GPU. For experiments on an embedded board, we use a Jetson Xavier NX, which has a 6-core ARM Carmel CPU operating at 1.4 GHz and 8 GB of RAM. It uses an integrated 384-Core Volta GPU [36] and is configured for the 6-core 15W power mode. From each BAL dataset, we pick the largest problem with less than 1000 poses to avoid thrashing on the Jetson, due to limited available memory. The properties of each dataset are summarized in Table 5.2.

Dataset	Images	Points	Observations	Initial MSE
Trafalgar-257	257	65 132	225 911	217
Dubrovnik-356	356	226 730	1 255 268	168
Ladybug-969	969	105 826	474 627	72
Venice-951	951	708 276	3 748 892	101
Final-961	961	187 103	1 692 975	51

Table 5.2: The number of images, points, observations, and initial mean squared error (MSE) for the BAL problems [3] used for evaluation.

We test each configuration with support for OpenMP and Eigen vectorization enabled, and record the final mean squared error (MSE) and the total BA time. To run our experiments, we modify the BAL example application from g2o [5] to process each dataset problem in different modes. The experiments use the default optimization parameters and compute the initial λ automatically. The PCG solvers use a tolerance of 1×10^{-6} and are limited to 50 iterations. Block ordering is used for the direct solvers, which performs block-wise permutation to reduce fill-in. The maximum number of BA iterations is set to 20 and the timeout to 3000 seconds. On the Jetson, it was necessary to increase the GPU watchdog timeout from five seconds to ten seconds.

Configuration	Block Solver	Linear Solver	H_{Schur}	Preconditioner
LLT / LDLT (CPU)	g2o	LLT / LDLT (Eigen)	Explicit	—
LLT / LDLT (GPU)	ours	LLT / LDLT (Eigen)	Explicit	—
PCG (CPU)	g2o	Block PCG (g2o)	Explicit	Block-Jacobi (H_{Schur})
PCG (GPU)	ours	Block PCG (ours)	Explicit	Block-Jacobi (H_{Schur})
PCG Implicit (GPU)	ours	Block PCG (ours)	Implicit	Block-Jacobi (H_{Schur})
PCG Implicit-H_{pp} (GPU)	ours	Block PCG (ours)	Implicit	Block-Jacobi (H_{pp})

Table 5.3: The different solver configurations used for experiments.

We summarize the tested configurations in Table 5.3. All solvers use double-precision. The Eigen sparse linear solver provided by g2o uses LLT decomposition internally. Older versions of g2o, such as the version included with ORB-SLAM3, use LDLT decomposition.

For consistency with the latest release, we also use LLT decomposition for our GPU block solver when performing experiments on BAL datasets.

The purpose of these experiments is to compare how the overall performance of visual bundle adjustment changes when we replace parts of the existing OpenMP implementation with GPU parallelization. It is possible to achieve lower MSE in less time by adjusting the initial λ, the maximum number of iterations, and by making changes to LMA itself. For example, setting $\lambda = 1$ initially can improve the error reduction, but for the largest Ladybug and Final problems, we observed that this results in Schur matrices that are no longer positive definite, causing Cholesky decomposition to fail.

Dataset	**Metric**	**LLT (CPU)**	**LLT (GPU)**	**PCG (CPU)**	**PCG (GPU)**	**PCG Implicit (GPU)**	**PCG Implicit-H_{pp} (GPU)**
Trafalgar-257	MSE	0.92	0.92	0.92	0.92	0.92	1.05
	Time	4.39	3.57	2.38	1.17	1.40	1.41
Dubrovnik-356	MSE	1.02	1.02	1.02	1.02	1.02	1.02
	Time	52.91	45.75	10.44	4.84	5.27	5.22
Ladybug-969	MSE	4.75	4.75	4.74	4.74	4.74	27.71
	Time	119.86	119.08	4.91	2.06	2.00	2.36
Venice-951	MSE	2.32	2.32	2.32	2.32	2.32	2.32
	Time	735.89	715.99	31.75	14.80	14.37	13.72
Final-961	MSE	1.88	1.91	1.88	1.88	1.88	1.88
	Time	1866.28	1717.02	134.36	11.79	8.85	7.47

Table 5.4: The performance on BAL datasets (desktop) with OpenMP enabled. Time is given in seconds.

Dataset	**Metric**	**LLT (CPU)**	**LLT (GPU)**	**PCG (CPU)**	**PCG (GPU)**	**PCG Implicit (GPU)**	**PCG Implicit-H_{pp} (GPU)**
Trafalgar-257	MSE	0.92	0.92	0.92	0.91	0.92	1.05
	Time	23.59	18.93	12.94	5.62	8.36	8.22
Dubrovnik-356	MSE	1.02	1.02	1.02	1.02	1.02	1.02
	Time	248.71	225.99	60.44	34.15	33.32	33.79
Ladybug-969	MSE	4.75	4.75	4.74	4.74	4.74	28.46
	Time	540.67	464.26	25.89	12.14	11.52	13.63
Venice-951	MSE	2.93[1]	3.83[2]	2.32	2.32	2.32	2.32
	Time	2902.24	2920.51	185.71	156.17	81.13	76.58
Final-961	MSE	2.07[3]	2.07[4]	1.88	1.88	1.88	1.88
	Time	2957.61	2947.97	443.90	282.47	80.29	72.76

1 LLT (CPU) times out after 18 iterations.
2 LLT (GPU) times out after 17 iterations.
3 LLT (CPU) times out after 9 iterations.
4 LLT (GPU) times out after 9 iterations.

Table 5.5: The performance on BAL datasets (Jetson) with OpenMP enabled. Time is given in seconds.

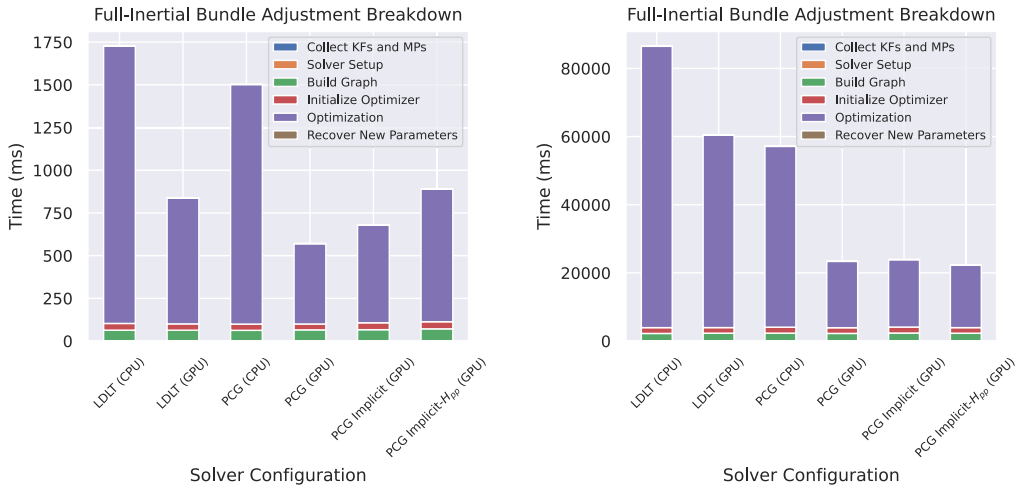
Table 5.4 and Table 5.5 show the results of our experiments. Since the linear solver step is the main bottleneck for these larger problems, computing the reduced system and the landmark update on the GPU yields only minor performance improvement. Due to this, on the Jetson, LLT configurations reach the timeout for the Venice-951 and Final-961 problems. On the other hand, GPU PCG configurations significantly improve performance compared to CPU PCG and direct solver configurations, even when OpenMP is enabled. In general, all configurations perform similarly in terms of reducing the MSE when they are not terminated early, with a few exceptions. On Trafalgar-257 and Ladybug-969, using the block diagonal of the H_{pp} matrix as the preconditioner results in higher MSE at the end of the optimization. The GPU LLT configuration also stops at a higher MSE compared to others for Final-961 on the desktop. It is also interesting to observe that for smaller problems such as Trafalgar-257 and Dubrovnik-356, implicit PCG configurations are not always the fastest.

During development, we have determined that our solution is able to handle problems up to Final-4585 on the desktop machine, which is the second largest in the BAL dataset. The largest problem, Final-13682, cannot be processed by GPU solver configurations, since it requests allocations larger than the 4 GB limit imposed by the Vulkan implementation.

5.7 Evaluation on SLAM Datasets

As our primary goal is to improve the performance of global bundle adjustment for visual-inertial SLAM, we evaluate our methods using maps generated by ORB-SLAM3 running in stereo-inertial mode. To generate these maps, we again process EuRoC [9] and TUM-VI [10] visual-inertial datasets. Unlike local-inertial bundle adjustment, global bundle adjustment is only applied as part of processes such as monocular map initialization and loop closure, and thus occurs infrequently. This is further complicated by the fact that loop detection may fail to trigger for a given run of a sequence. For these reasons, the method of evaluation used in Chapter 4, in which the average local BA execution times were computed over multiple runs, is not suitable for evaluating the performance of global bundle adjustment.

To address this problem, we modify ORB-SLAM3 to perform full- (global-) inertial bundle adjustment (FIBA) on the entire map produced after processing each dataset sequence. The main benefit of this approach is that by disabling the parameter recovery step, in which the optimized parameters stored in the BA graph are retrieved, we can run multiple experiments on the same map. As shown for a large and small map in Figure 5.6, parameter recovery amounts to less of 1% of the total FIBA time on average. Thus, at the end of each sequence, we test each solver configuration and perform ten trials, using ten BA iterations each. For these experiments, OpenMP support is left disabled and Eigen vectorization is enabled, as in the original build configuration.



(a) room4 (b) outdoors6

Figure 5.6: FIBA breakdown for the Jetson.

5.7.1 Block Solver Performance

Table 5.6 and Table 5.7 summarize the average amount of time spent on different steps in the block solver for a single LMA iteration and the average BA time, for end-of-sequence

full-inertial bundle adjustment, for each dataset category. For large maps generated by processing longer outdoors sequences, the linear solver step is the most expensive, and switching to the GPU PCG solver in any configuration greatly reduces the execution time. Yet for some datasets, the linear solver is not always the bottleneck. Such behaviour is observed for desktop CPU solver configurations on TUM-VI room maps, where the average amount of time spent on the Schur complement step exceeds the amount of time spent on the linear solver, due to the smaller map sizes.

Dataset	**LDLT (CPU)**	**LDLT (GPU)**	**PCG (CPU)**	**PCG (GPU)**	**PCG Implicit (GPU)**	**PCG Implicit-H_{pp} (GPU)**
EuRoC						
Schur	16.36	0.56	16.56	0.55	0.17	0.18
Linear Solver	31.20	31.26	13.63	3.40	5.81	5.20
Δx_l	2.23	0.25	2.16	0.14	0.15	0.15
Total BA	799.88	634.77	491.67	213.31	230.14	224.75
TUM-VI room						
Schur	10.34	0.42	10.31	0.41	0.11	0.12
Linear Solver	9.67	9.90	7.65	3.12	4.69	4.16
Δx_l	0.81	0.19	0.80	0.09	0.09	0.09
Total BA	343.59	227.81	302.19	134.44	150.22	145.05
TUM-VI outdoors						
Schur	247.26	7.67	247.26	7.70	0.94	0.94
Linear Solver	3284.47	3302.91	955.91	28.41	31.32	30.95
Δx_l	25.71	0.75	25.87	0.56	0.57	0.57
Total BA	77 102.95	73 394.76	24 866.70	4170.95	4075.39	4077.66

Table 5.6: The average block solver and BA times (ms) on the desktop for full-map FIBA.

Dataset	LDLT (CPU)	LDLT (GPU)	PCG (CPU)	PCG (GPU)	PCG Implicit (GPU)	PCG Implicit-H_{pp} (GPU)
EuRoC						
Schur	52.53	4.20	54.05	4.35	0.74	0.74
Linear Solver	111.72	113.38	66.55	17.20	33.55	31.68
Δx_l	9.96	0.81	10.22	0.40	0.41	0.41
Total BA	3580.35	2227.73	1982.79	938.04	1045.04	1025.42
TUM-VI room						
Schur	34.97	3.53	35.07	3.56	0.49	0.48
Linear Solver	42.85	42.78	38.32	12.44	21.40	20.72
Δx_l	5.03	0.66	5.04	0.28	0.28	0.28
Total BA	1752.04	1138.24	1485.72	685.53	764.56	757.36
TUM-VI outdoors						
Schur	455.89	45.23	455.61	45.19	4.34	4.33
Linear Solver	3882.10	3896.52	1952.26	256.91	301.37	317.35
Δx_l	75.65	2.56	74.67	2.15	2.13	2.13
Total BA	96 265.39	89 251.17	63 510.52	16 331.78	16 329.50	16 166.17

Table 5.7: The average block solver and BA times (ms) on the Jetson for full-map FIBA.

5.7.2 Overall Performance of Full-Inertial Bundle Adjustment

We evaluate the performance of FIBA using the strategy described above. The solver configurations from Table 5.3 are used, except that LDLT replaces LLT, as in the original implementation. Table 5.8 summarizes the statistics of the maps generated by each sequence for both platforms. The Xavier NX generates smaller maps due to different performance characteristics. The average FIBA times are given in Table 5.9 and Table 5.10. This time excludes the time spent on recovering and applying optimized parameters to the map, due to the approach used.

Sequence	Keyframes (Desktop)	Map Points (Desktop)	Observ. (Desktop)	Keyframes (Jetson)	Map Points (Jetson)	Observ. (Jetson)
MH01	126	9403	42 327	126	9139	41 544
MH02	114	8450	38 227	114	8443	37 836
MH03	148	11 516	56 160	135	10 261	50 165
MH04	169	14 119	65 201	157	12 548	58 945
MH05	161	13 830	65 977	154	12 381	60 653
V101	105	8547	37 524	110	9038	40 793
V102	115	10 872	44 054	109	9541	38 578
V103	130	9717	37 782	119	8523	34 355
V201	92	7780	31 948	89	7582	30 186
V202	127	10 333	40 774	115	8463	33 998
V203	284	15 571	65 778	225	10 665	45 247
room1	78	2622	21 926	78	2499	21 423
room2	82	2925	23 009	80	2897	22 378
room3	81	3353	22 574	85	3483	23 669
room4	83	3301	20 995	77	2919	19 669
room5	81	2651	23 787	75	2414	21 067
room6	72	2320	21 486	73	2367	22 125
outdoors1	3738	88 768	431 608	3342	71 179	333 573
outdoors2	2510	72 079	362 589	2251	58 061	292 052
outdoors3	2943	72 929	431 305	2125	45 115	246 289
outdoors4	1770	43 247	227 468	1599	35 935	181 294
outdoors5	2365	71 880	447 025	1849	55 099	312 450
outdoors6	5183	187 253	907 125	4091	122 048	564 374
outdoors7	3262	99 731	504 594	2847	78 247	387 359
outdoors8	2075	62 077	338 256	1820	49 589	269 489

Table 5.8: The number of keyframes, map points, and observations used for end-of-sequence FIBA experiments. This does not include variables and observations for inertial constraints introduced between consecutive keyframes.

In order to observe the behaviour of each configuration, we recorded the robustified χ^2 metric over time. We use the χ^2 error metric, based on the weighted sum of squared errors, as provided by g2o rather than MSE, since FIBA uses a robust Huber kernel to reduce the influence of feature mismatches, whereas no kernel is used for inertial residuals [1]. Figure 5.7 and Figure 5.8 show the convergence behaviour for EuRoC, while Figure 5.9 and Figure 5.10 show it for TUM-VI room sequences. Lastly, the behaviour for larger maps generated by TUM-VI outdoors sequences is shown in Figure 5.11 and Figure 5.12.

Sequence	LDLT (CPU)	LDLT (GPU)	PCG (CPU)	PCG (GPU)	PCG Implicit (GPU)	PCG Implicit-H_{pp} (GPU)
MH01	0.68	0.54	0.50	0.22	0.24	0.23
MH02	0.59	0.41	0.44	0.18	0.20	0.19
MH03	0.78	0.54	0.61	0.26	0.28	0.28
MH04	1.03	0.93	0.63	0.28	0.30	0.30
MH05	0.74	0.48	0.64	0.29	0.30	0.31
V101	0.65	0.40	0.31	0.13	0.15	0.14
V102	0.61	0.42	0.41	0.19	0.21	0.20
V103	0.62	0.28	0.30	0.14	0.15	0.15
V201	0.40	0.27	0.30	0.14	0.16	0.15
V202	0.73	0.37	0.35	0.15	0.17	0.16
V203	1.97	2.34	0.92	0.36	0.38	0.36
room1	0.33	0.23	0.24	0.11	0.12	0.12
room2	0.37	0.33	0.24	0.12	0.13	0.12
room3	0.33	0.21	0.33	0.15	0.17	0.16
room4	0.29	0.15	0.24	0.11	0.12	0.11
room5	0.47	0.28	0.47	0.19	0.22	0.21
room6	0.28	0.16	0.30	0.13	0.14	0.14
outdoors1	51.47	44.82	26.35	4.30	4.24	4.26
outdoors2	30.44	28.16	19.91	3.39	3.39	3.61
outdoors3	296.18	286.92	52.06	4.69	4.07	4.07
outdoors4	17.72	15.34	13.33	2.39	2.42	2.24
outdoors5	81.64	95.37	16.81	3.45	3.35	3.28
outdoors6	60.15	49.93	32.93	7.70	7.68	7.68
outdoors7	50.08	43.54	18.88	4.20	4.20	4.22
outdoors8	29.16	23.08	18.66	3.25	3.26	3.25

Table 5.9: The average execution times (seconds) across the ten trials on the desktop for full-map FIBA.

In terms of overall speedup, our methods on the desktop machine achieve up to about a 72.9× speedup over the base CPU LDLT configuration, which corresponds to approximately a 12.8× speedup over the CPU PCG configuration, when using implicit PCG methods. On the Jetson, there is a smaller speedup of up to 19.1× over CPU LDLT and 8.24× over CPU PCG for implicit PCG methods. To reiterate, the reason implicit configurations are faster is that the computation of the Schur complement is skipped. Both implicit configurations largely perform similarly, although the H_{pp} variant has a slight advantage on the Jetson.

It should also be noted that although PCG configurations are shown to be much faster than LDLT configurations in our experiments, LDLT configurations tend to achieve better error reduction for larger maps. This is especially noticeable for maps generated by outdoors sequences on the desktop machine, which are larger than their counterparts on the Jetson. The reason for this is that PCG configurations compute inexact LMA steps [3] since they solve for Δx_p iteratively. This can be compensated for by adjusting the PCG stopping criteria, but finding a good balance between the error reduction quality and the speedup

Sequence	LDLT (CPU)	LDLT (GPU)	PCG (CPU)	PCG (GPU)	PCG Implicit (GPU)	PCG Implicit-H_{pp} (GPU)
MH01	3.56	2.57	2.19	0.99	1.08	1.06
MH02	3.43	1.54	1.83	0.85	0.93	0.96
MH03	3.77	2.48	2.15	1.02	1.15	1.15
MH04	3.90	1.79	2.50	1.20	1.32	1.20
MH05	2.28	1.76	3.10	1.42	1.59	1.57
V101	3.42	2.33	1.54	0.75	0.84	0.82
V102	2.57	1.71	1.40	0.73	0.82	0.81
V103	2.30	1.84	1.43	0.75	0.84	0.83
V201	2.47	0.90	1.27	0.68	0.77	0.75
V202	3.53	2.49	1.59	0.76	0.85	0.84
V203	8.17	5.09	2.81	1.18	1.30	1.28
room1	1.52	1.05	1.60	0.69	0.77	0.76
room2	1.44	0.76	0.76	0.44	0.45	0.45
room3	2.63	1.80	1.33	0.72	0.77	0.76
room4	1.44	1.00	1.56	0.70	0.79	0.78
room5	1.99	1.30	2.12	0.90	1.04	1.03
room6	1.50	0.93	1.54	0.67	0.76	0.76
outdoors1	80.57	82.75	72.89	18.44	18.32	19.08
outdoors2	70.66	55.19	55.15	14.65	15.15	15.08
outdoors3	263.16	263.78	113.83	19.28	15.87	13.81
outdoors4	38.19	37.13	30.19	8.92	9.14	9.46
outdoors5	70.13	58.33	53.61	14.59	15.08	14.75
outdoors6	106.44	93.76	77.62	24.67	25.72	26.59
outdoors7	77.39	73.75	57.08	17.34	18.24	17.43
outdoors8	63.59	49.31	47.71	12.76	13.11	13.14

Table 5.10: The average execution times (seconds) across the ten trials on the Jetson for full-map FIBA.

for both large and small bundle adjustment problems may be non-trivial. Possible ways to address this challenge are discussed in Chapter 6.

Jetson Performance

The maximum speedup observed on the Jetson is smaller, primarily due to the differences in hardware. The Volta GPU on the Jetson Xavier NX has 384 CUDA cores [36], whereas the RTX 3080, as reported by the nvidia-smi tool, has 8704 CUDA cores and also operates at faster clock speeds (1965 MHz vs 1100 MHz). The larger number of CUDA cores allows the desktop machine to process more work queues in parallel, giving it a major performance advantage when performing bundle adjustment on large outdoors maps, which have thousands of keyframes. This is reflected in Table 5.9 and Table 5.10. The maximum speedups over LDLT (CPU) and PCG (CPU), excluding outdoors sequences, are 5.52× and 2.57× respectively on the desktop. On the Jetson, the corresponding speedups are 6.94×

and 2.38×. It may be possible to improve the efficiency further by adjusting the number of work queues processed by each workgroup on the Jetson, as well as the workgroup sizes.

LDLT Slowdown

Occasionally, the LDLT (GPU) configuration takes longer than LDLT (CPU). The most severe cases occur for outdoors5 on the desktop and outdoors1 on the Jetson. On the desktop, as shown in Figure 5.11, LDLT (GPU) not only takes longer but also achieves worse error reduction. The reason for the increased execution time is not transfer overhead, but that the average number of inner LMA iterations (for adjusting the damping factor) per outer optimization iteration is higher (1.80 compared to 1.44). The average amount of time spent in the block solver for an LMA iteration is still lower when using the GPU configuration. LMA iterations may be repeated in situations where decomposition fails, or when the computed Δx increases the error instead. Numerical differences in the reduced system calculation for each configuration may also result in different behaviour for the linear solver step. On the Jetson, LDLT (GPU) achieves better error reduction for outdoors1 (Figure 5.12), but again performs more LMA iterations on average (1.90 vs 1.60), resulting in larger BA times. Overall, the performance degradation observed can likely be avoided by switching to a GPU-accelerated LDLT solver.

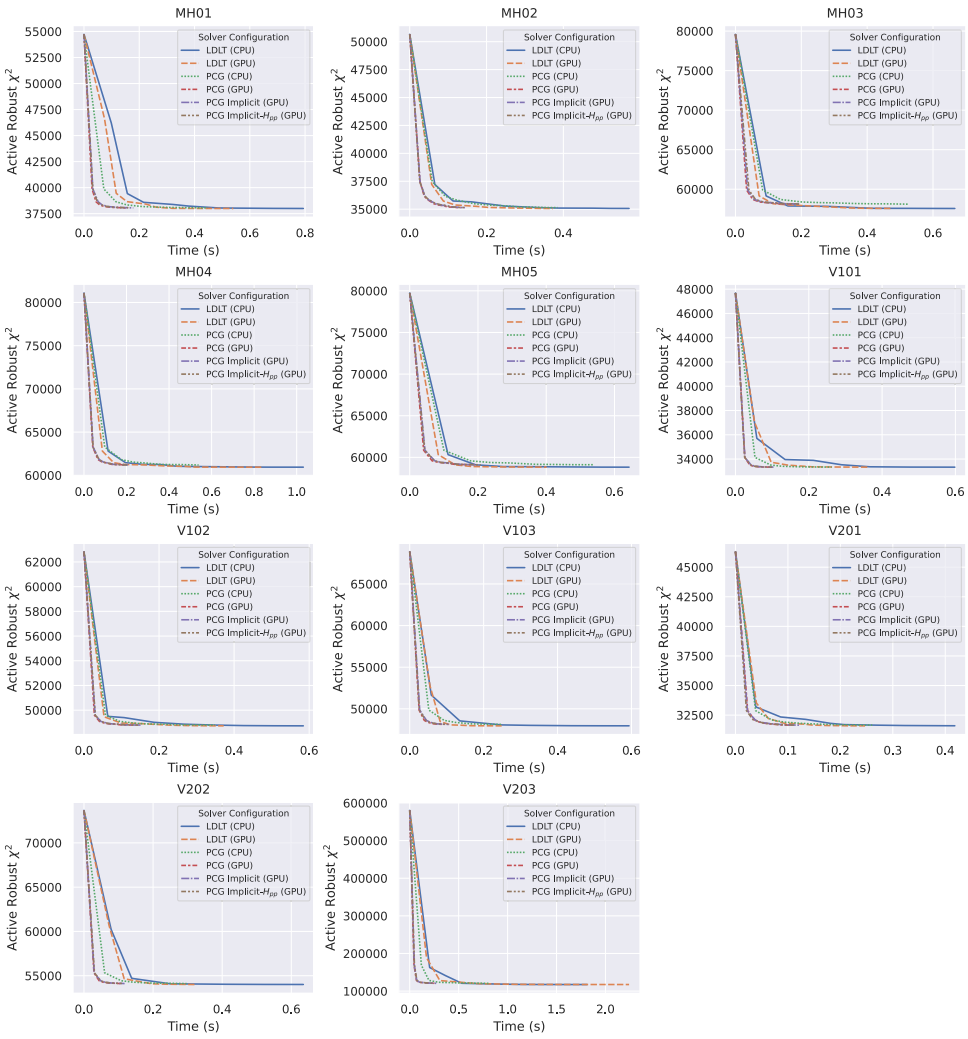


Figure 5.7: Convergence over time for the desktop on EuRoC sequences.

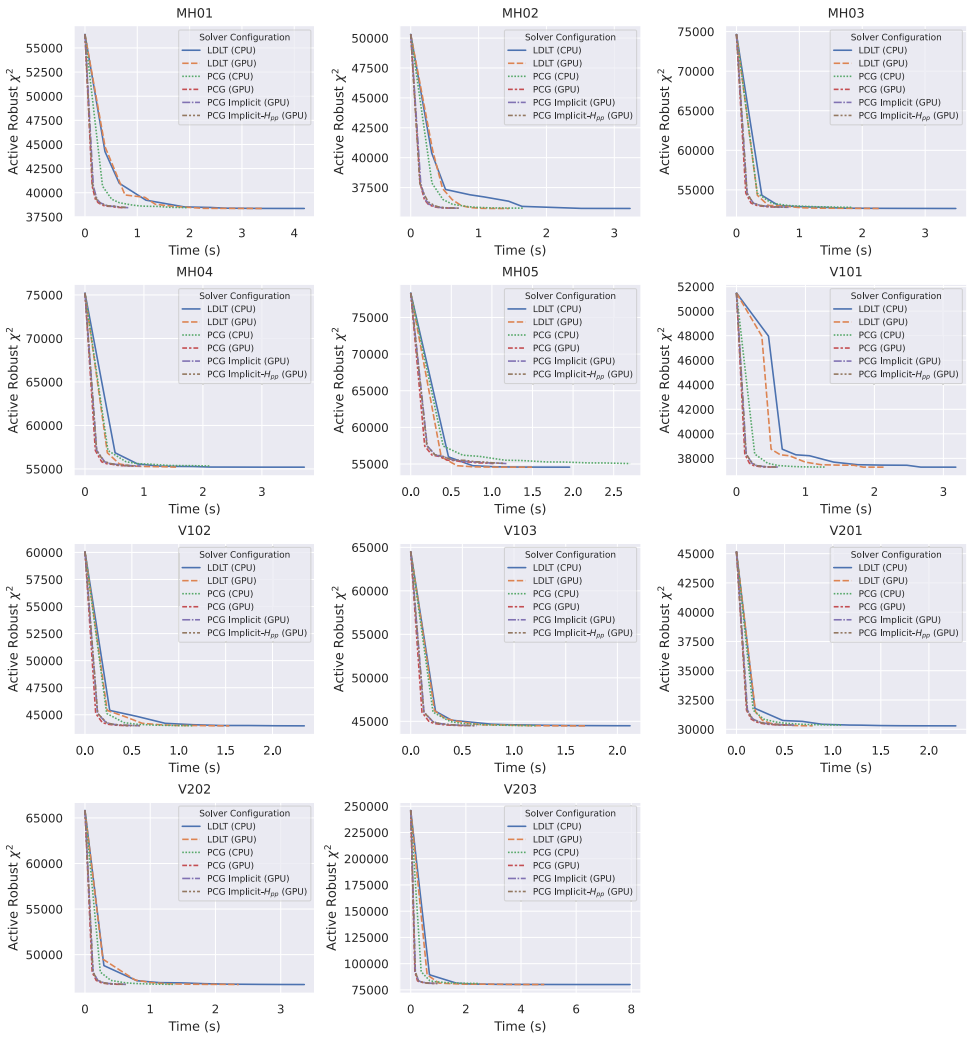


Figure 5.8: Convergence over time for the Jetson on EuRoC sequences.

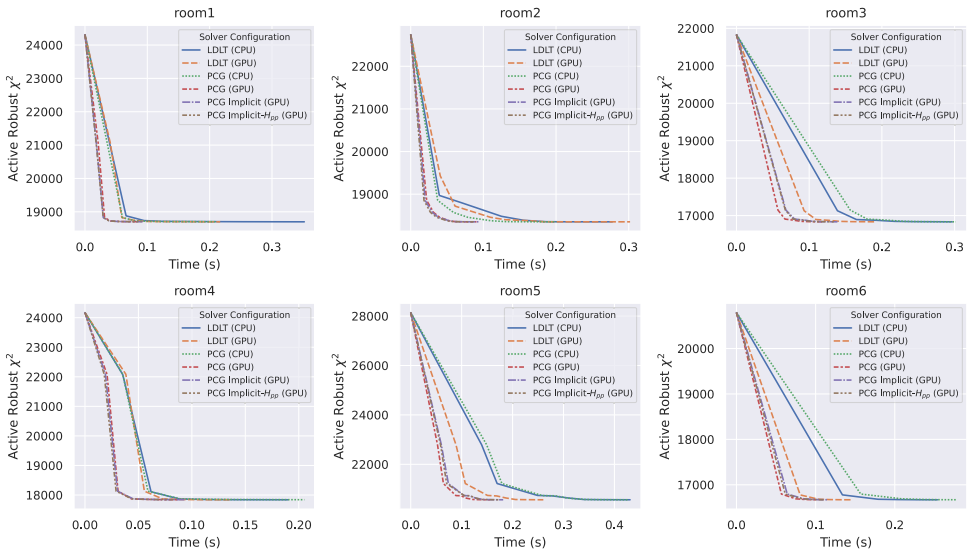


Figure 5.9: Convergence over time for the desktop on TUM-VI room sequences.

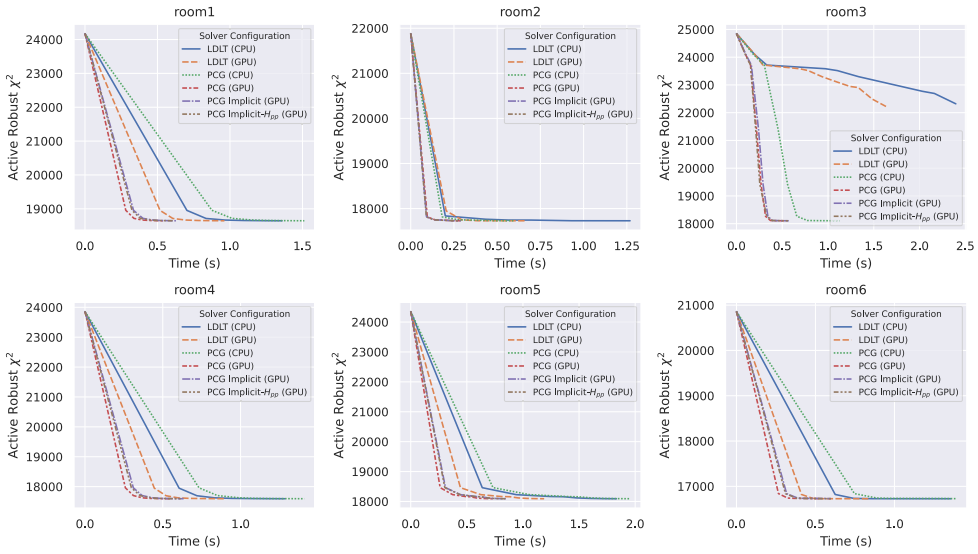


Figure 5.10: Convergence over time for the Jetson on TUM-VI room sequences.

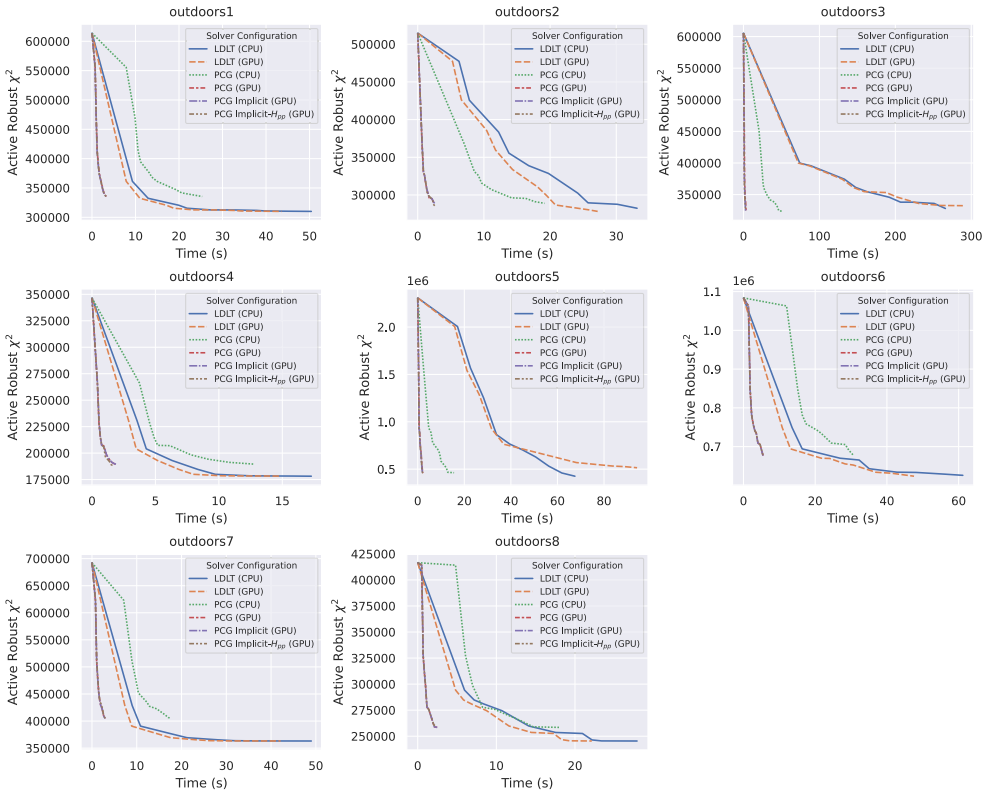


Figure 5.11: Convergence over time for the desktop on TUM-VI outdoors sequences.

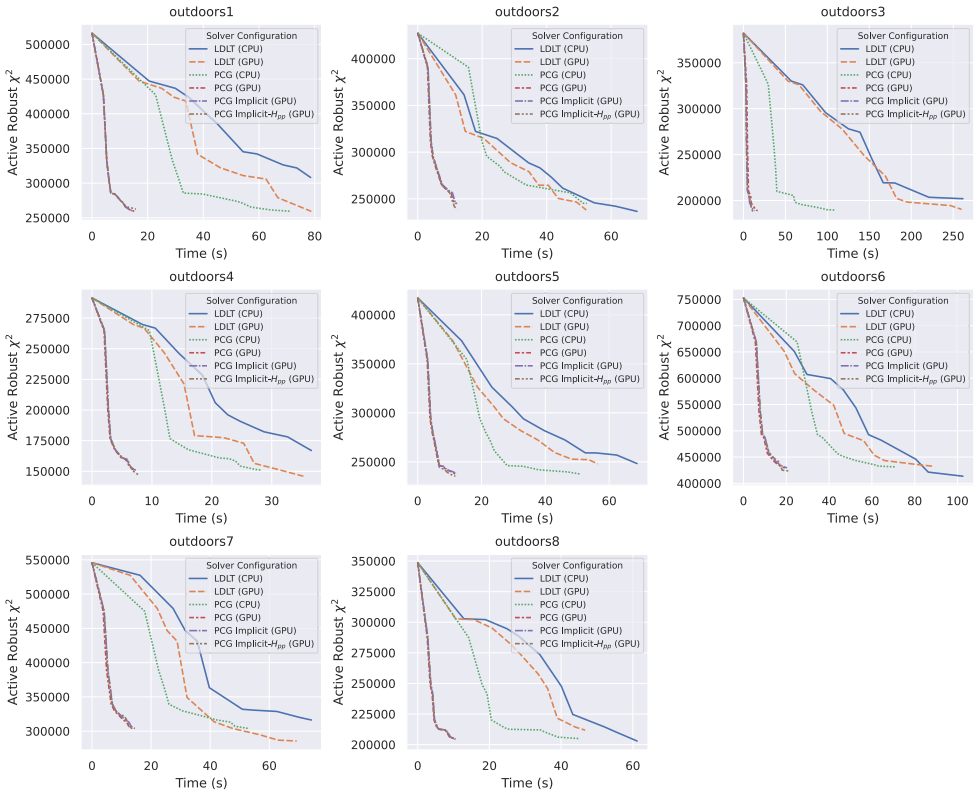


Figure 5.12: Convergence over time for the Jetson on TUM-VI outdoors sequences.

5.7.3 Trajectory Error

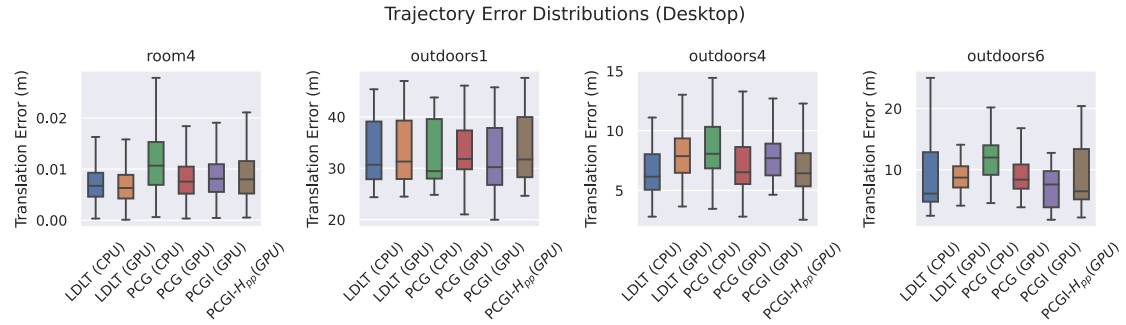
We perform an additional set of experiments in which each solver configuration is used for all full-inertial bundle adjustment calls throughout each sequence, rather than at the end. For selected TUM-VI sequences, chosen by final map sizes (Table 5.8), we aggregate the trajectory error across five runs. The trajectory error for each run is computed using evo [55], which first aligns the estimated trajectory with a ground truth trajectory before computing the error. The root-mean-square absolute trajectory error (RMS ATE) in meters is reported for each sequence and configuration in Table 5.11 and Table 5.12. Additionally, the distributions of the trajectory errors, across all three dimensions, are shown in Figure 5.13a and Figure 5.13b. There does not appear to be any consistent change in error due to the configuration. It should be noted that trajectory alignment is less stable for outdoors sequences, since ground truth data is only available at the start and the end of the sequence. The differences in the error on each platform are due to the different performance characteristics. As summarized in Table 5.8, fewer keyframes, map points, and observations are collected for longer sequences on the Xavier NX.

Sequence	**LDLT (CPU)**	**LDLT (GPU)**	**PCG (CPU)**	**PCG (GPU)**	**PCG-Implicit (GPU)**	**PCG-Implicit-H_{pp} (GPU)**
room4	0.01	0.01	0.01	0.01	0.01	0.01
outdoors1	33.95	34.15	33.60	34.12	32.28	34.51
outdoors4	7.47	8.33	8.82	7.54	8.01	7.15
outdoors6	11.53	9.10	12.42	10.36	7.57	10.06

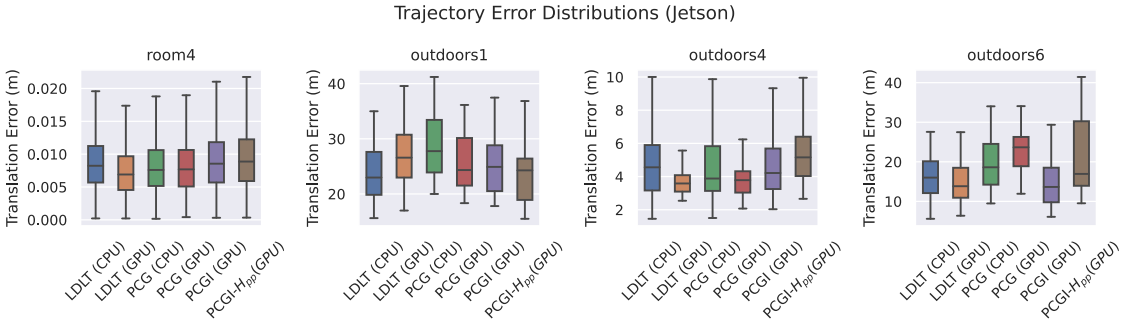
Table 5.11: The RMS ATE (m) computed from five runs for each sequence (desktop).

Sequence	**LDLT (CPU)**	**LDLT (GPU)**	**PCG (CPU)**	**PCG (GPU)**	**PCG-Implicit (GPU)**	**PCG-Implicit-H_{pp} (GPU)**
room4	0.01	0.01	0.01	0.01	0.01	0.01
outdoors1	24.42	28.03	29.64	26.46	26.15	24.90
outdoors4	5.17	4.07	5.09	4.11	5.05	5.71
outdoors6	16.97	15.56	20.64	23.58	15.95	23.39

Table 5.12: The RMS ATE (m) computed from five runs for each sequence (Jetson).



(a) Desktop



(b) Jetson

Figure 5.13: The trajectory translation error distributions, across all dimensions.

5.7.4 GPU Memory Usage

The total buffer memory usage of each GPU method, as reported by VulkanMemoryAllocator, for end-of-sequence FIBA is shown in Figure 5.14, Figure 5.15, and Figure 5.16. As expected, implicit PCG methods require the least amount of memory, since at most, only the block diagonal of H_{Schur} must be constructed for the preconditioner. Thus, for implicit methods, we entirely avoid allocating memory for intermediate result $H_{pl}H_{ll}^{-1}$. Explicit PCG uses the largest amount of memory, since we allocate additional memory for vectors and work queues in addition to the memory for computing the Schur complement. Lower buffer memory usage is observed on the Jetson. This is because only host-coherent memory is used, while the desktop machine creates additional buffers with device-local memory to speed up global memory access in-shader. Furthermore, smaller maps are produced on the Jetson. Implicit PCG using the block diagonal of H_{pp} as the preconditioner matrix uses the least amount of memory for buffers, although this amount is only slightly less than using the block diagonal of the Schur matrix instead.

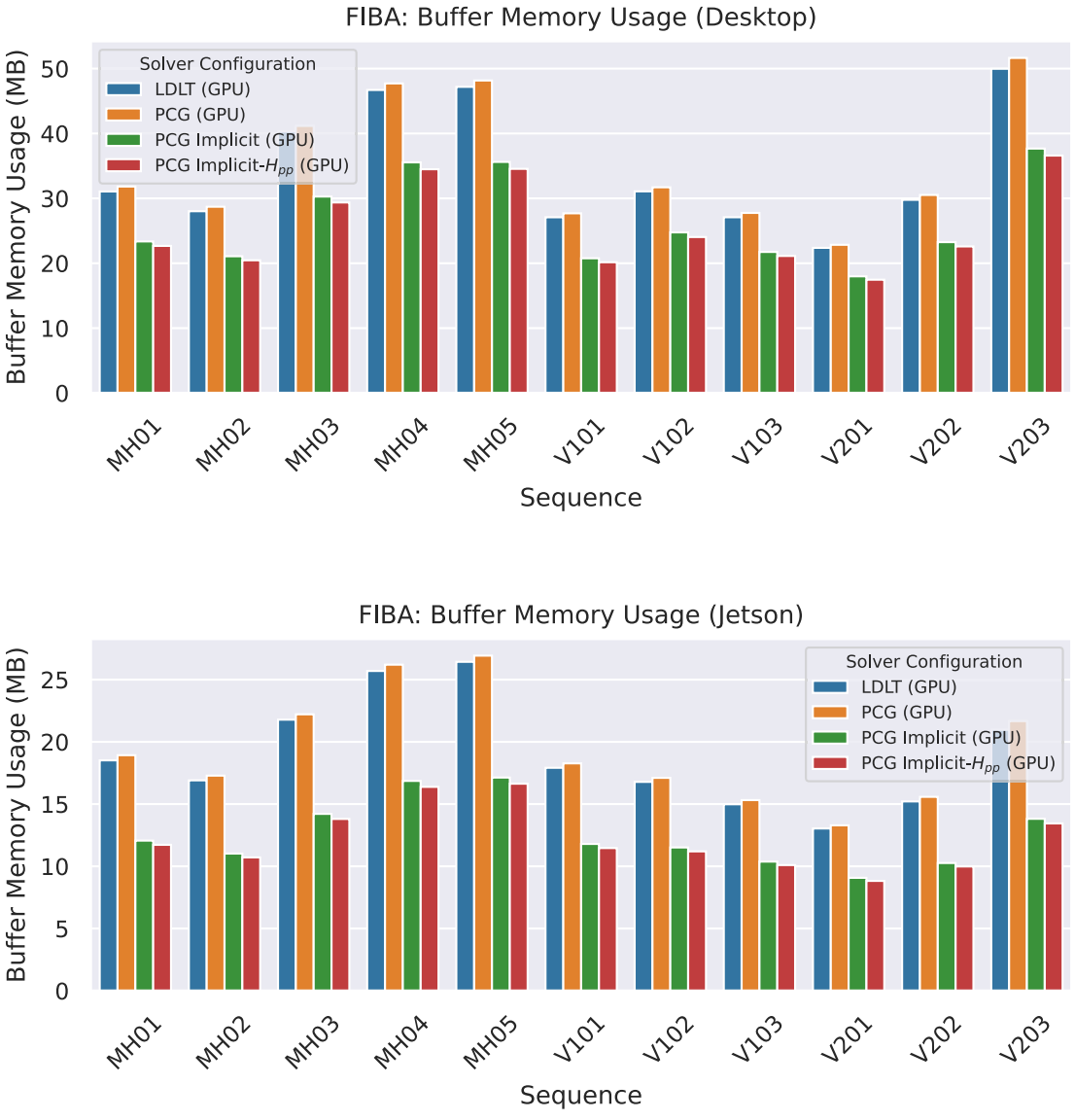


Figure 5.14: GPU buffer memory usage for EuRoC sequences.

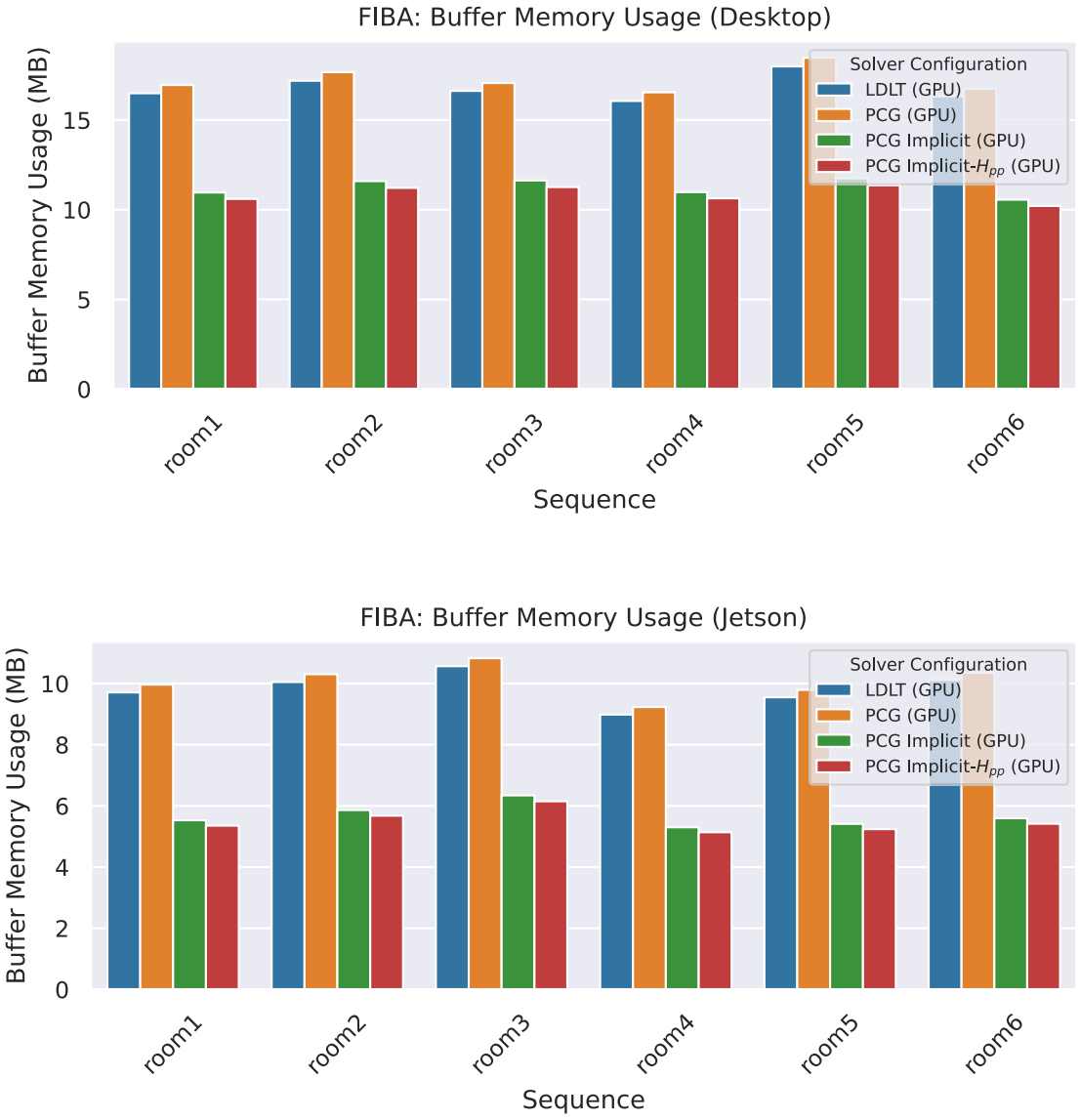


Figure 5.15: GPU buffer memory usage for TUM-VI room sequences.

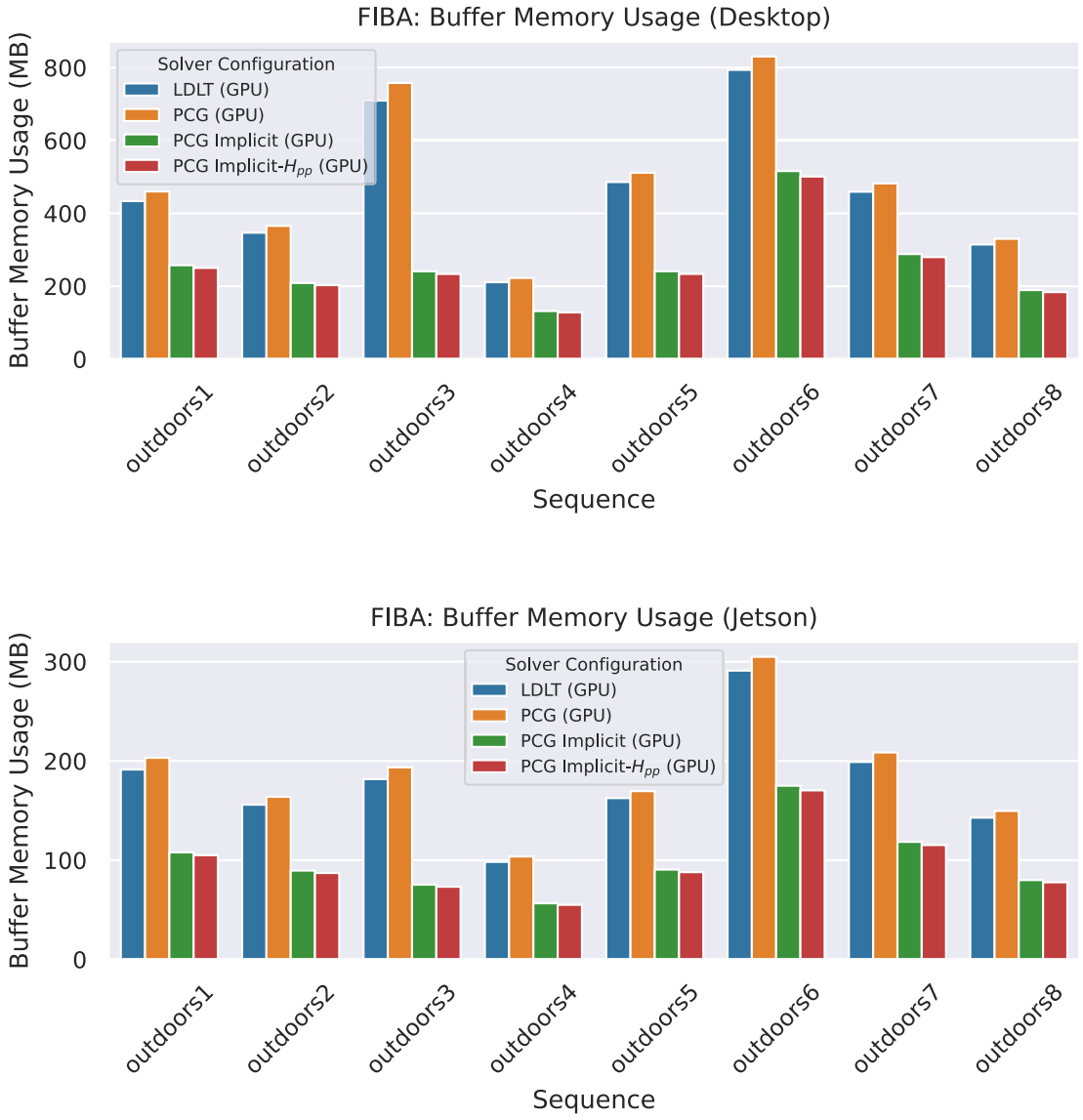


Figure 5.16: GPU buffer memory usage for TUM-VI outdoors sequences.

5.7.5 Revised Performance of Local-Inertial Bundle Adjustment

We rerun the experiments for local-inertial bundle adjustment from Chapter 4, mainly to determine how the changes to the block solver and work queue generation have affected the performance. We do not evaluate the performance of the PCG solver, since too many iterations are needed to converge to a solution with low residual error, and each PCG iteration costs at least one queue submission, so LDLT for small matrices yields better performance. Table 5.13 and Table 5.14 compare the local-inertial BA execution times for the original CPU block solver and the revised GPU block solver. While there is a small decrease in performance on outdoors sequences for both platforms, there is also a larger performance improvement for the Jetson on EuRoC sequences. The decrease in execution time of the revised solver ranges from 17.97% to 37.72% ($1.22 - 1.61\times$ speedup) on the desktop, and 15.87% to 32.55% ($1.19-1.48\times$ speedup) on the Jetson. For the V201 sequence, the total fraction of time spent in the local mapping thread on local bundle adjustment decreased from 17.41% to 11.31% on the desktop and from 48.39% to 36.93% on the Jetson.

Sequence	**CPU Time (ms)**	**GPU Time (ms)**	**Avg Diff. (%)**
MH01	52.66 ± 21.52	34.30 ± 12.85	-34.86
MH02	45.77 ± 18.69	31.55 ± 11.82	-31.05
MH03	55.69 ± 19.51	38.32 ± 13.74	-31.19
MH04	50.99 ± 13.92	35.62 ± 8.89	-30.14
MH05	52.03 ± 13.12	35.56 ± 8.76	-31.66
V101	61.49 ± 13.77	41.51 ± 8.34	-32.49
V102	55.74 ± 15.92	39.19 ± 11.68	-29.68
V103	49.75 ± 14.07	33.06 ± 8.59	-33.55
V201	53.85 ± 13.32	33.53 ± 7.23	-37.72
V202	54.59 ± 12.84	36.39 ± 9.03	-33.34
V203	35.07 ± 11.79	25.29 ± 7.78	-27.89
outdoors1	35.08 ± 19.05	28.67 ± 12.91	-18.28
outdoors2	41.74 ± 22.07	33.14 ± 15.14	-20.61
outdoors3	46.37 ± 16.56	38.04 ± 12.35	-17.97
outdoors4	38.93 ± 23.07	29.58 ± 14.69	-24.00
outdoors5	58.15 ± 17.99	44.67 ± 13.11	-23.18
outdoors6	51.69 ± 16.70	40.83 ± 11.48	-21.01
outdoors7	43.07 ± 14.28	34.98 ± 10.99	-18.79
room1	75.85 ± 18.29	57.01 ± 17.04	-24.85
room2	70.25 ± 19.70	56.22 ± 17.49	-19.97
room3	75.91 ± 18.00	59.10 ± 18.36	-22.14
room4	68.40 ± 18.35	49.07 ± 15.30	-28.26
room5	75.65 ± 15.87	58.86 ± 15.51	-22.20
room6	73.85 ± 17.20	54.29 ± 15.04	-26.49

Table 5.13: Revised average local-inertial BA run times for ORB-SLAM3 on the desktop machine. CPU timings from Gopinath, Dantu, and Ko [11] © 2023 IEEE.

Sequence	CPU Time (ms)	GPU Time (ms)	Avg Diff. (%)
MH01	290.11 ± 90.14	196.20 ± 59.85	-32.37
MH02	266.54 ± 87.47	180.75 ± 57.81	-32.19
MH03	293.20 ± 82.64	209.73 ± 59.86	-28.47
MH04	259.37 ± 63.69	185.31 ± 43.11	-28.55
MH05	265.93 ± 58.42	187.86 ± 39.44	-29.36
V101	319.78 ± 67.03	221.36 ± 46.16	-30.78
V102	279.96 ± 76.11	201.45 ± 54.99	-28.04
V103	232.96 ± 60.27	168.80 ± 39.94	-27.54
V201	270.85 ± 54.41	182.68 ± 31.99	-32.55
V202	263.80 ± 57.52	189.26 ± 42.56	-28.26
V203	156.71 ± 67.21	119.02 ± 42.38	-24.06
outdoors1	174.06 ± 79.79	144.56 ± 56.67	-16.95
outdoors2	194.50 ± 96.58	162.60 ± 68.52	-16.40
outdoors3	194.88 ± 87.45	163.79 ± 61.70	-15.96
outdoors4	189.94 ± 95.33	156.30 ± 66.92	-17.71
outdoors5	268.61 ± 93.56	218.82 ± 71.87	-18.54
outdoors6	227.14 ± 66.35	191.09 ± 51.08	-15.87
outdoors7	210.96 ± 65.75	172.21 ± 48.37	-18.36
room1	349.89 ± 83.33	274.82 ± 71.53	-21.46
room2	351.81 ± 98.73	274.71 ± 81.28	-21.92
room3	321.51 ± 73.31	270.27 ± 69.44	-15.94
room4	314.02 ± 77.29	242.78 ± 69.64	-22.69
room5	342.29 ± 76.55	273.27 ± 65.79	-20.17
room6	355.83 ± 85.74	273.96 ± 72.95	-23.01

Table 5.14: Revised average local-inertial BA run times for ORB-SLAM3 on the Jetson Xavier NX. CPU timings from Gopinath, Dantu, and Ko [11] © 2023 IEEE.

Chapter 6

Conclusion

In this thesis, we have demonstrated how we can improve the performance of visual-inertial bundle adjustment for on-device SLAM using GPU resources. In Chapter 4, we developed techniques that specifically target operations on small- to medium-sized matrices, mainly for the Schur complement. We implemented our techniques as a drop-in replacement block solver for g2o, and integrated it with ORB-SLAM3. Our evaluation with EuRoC and TUM-VI datasets showed that we can speed up local-inertial bundle adjustment by up to 1.51× across indoor and outdoor SLAM sequences on a desktop machine, and by up to 1.36× on a Jetson Xavier NX embedded board.

Later, in Chapter 5, we extended our methods to accelerate global bundle adjustment problems as well. Primarily, we developed a new work queue generation method and an iterative linear solver based on the method of preconditioned conjugate gradients. Our experiments on BAL datasets showed that our methods provide an additional speedup over the existing OpenMP parallelization for g2o. For full-inertial bundle adjustment on maps produced by ORB-SLAM3, GPU implicit PCG methods obtained a speedup of up to 12.8× on the desktop, and 8.24× on the Jetson Xavier NX board, over the existing explicit CPU PCG implementation. This corresponds to a 72.9× speedup on the desktop and a 19.1× speedup on the Jetson over the base CPU LDLT configuration, although in some cases LDLT configurations achieved better error reduction for larger maps. Lastly, reevaluating our methods for local-inertial bundle adjustment showed an improved speedup of up to 1.61× and 1.48× across indoor and outdoor SLAM sequences on the desktop and the Jetson Xavier NX, respectively.

Our results show that accelerating generic operations for matrices and vectors can greatly improve the performance of both local-inertial and full-inertial bundle adjustment, and point to a need for memory-efficient techniques to gracefully handle constraint-specific calculations.

6.1 Limitations and Future Work

6.1.1 Evaluation and Datasets

From our experiments in Chapter 5, we have found that evaluating the performance of full-inertial bundle adjustment is particularly challenging due to factors such as nondeterministic behaviour resulting in generated workload variability. Furthermore, the map generation process can be time-consuming, depending on the sequence length and hardware characteristics of the target device. There is a clear need for datasets that represent small to large visual-inertial BA problems, using compact parameterization for pose, inertial, and landmark variables that would typically be found in a real-time SLAM system. Another limitation of the evaluation is that it does not assess the effect on mapping quality due to the lack of absolute ground truth data. This may be addressed through the use of simulated sequences, where the geometry of the virtual environment is readily available.

6.1.2 Parallelizing Constraint-Specific Calculations

Our solution does not yet address all bottlenecks for bundle adjustment. Although we have implemented GPU acceleration for the main solving steps (Schur complement, Δx_p, Δx_l), our techniques remain general and do not provide any speedup for calculations which vary between constraints, as shown in Figure 6.1. Meanwhile, existing solutions for visual bundle adjustment handle Hessian and error computations as well [15, 13]. Improving the performance of these calculations for visual-inertial SLAM requires a more specialized solution, and achieving this in a user-friendly manner is non-trivial. Extending our existing shader-based solution to support the expression of non-linear constraints would expose API-specific details to the user, complicating development. Moreover, vertex parameterization may vary across SLAM systems due to differences in sensor configurations or system design. For this reason, distributing shader-based vertices and constraints as part of a library may be an inadequate solution. Using a cross-platform abstraction layer [56] rather than Vulkan directly may help address some of these problems. Alternatively, techniques may be adapted from domain-specific languages, which have demonstrated promising results for solving non-linear least squares problems on GPUs [57, 58].

6.1.3 GPU Direct Methods

Direct solving methods allow for a more exact computation of the parameter update in each optimization iteration, and thus it may be beneficial to investigate the performance of GPU-accelerated LLT/LDLT solvers for both local and global bundle adjustment problems. Still, off-the-shelf solvers require converting H_{Schur} to an intermediate sparse matrix format, which adds overhead and increases memory usage. Therefore, it would also be advantageous to investigate the viability of GPU-acceleration for direct methods on sparse block matrices with variable-sized blocks, as we have done with our PCG solver.

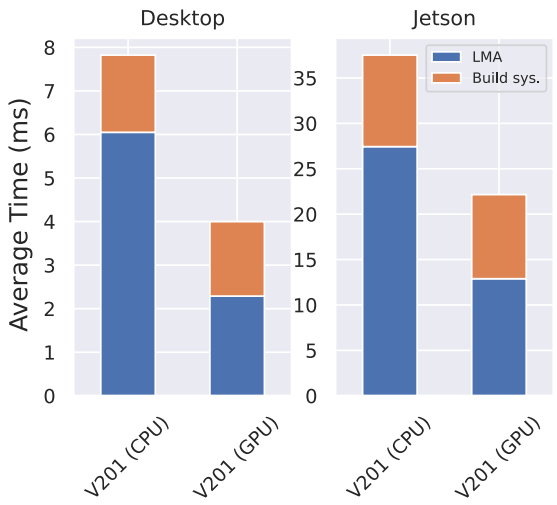


Figure 6.1: The overhead of computing the linear system (orange) for each optimization iteration. Accelerating constraint-specific calculations would reduce this overhead.

6.1.4 Reusing Partial Computations

There are also additional opportunities to reuse intermediate results, which can be exploited to improve the overall performance of bundle adjustment. Given that the structure of the Schur matrix does not change between successive iterations in the same bundle adjustment call, incremental updates to the factorization may provide further performance improvement when using direct methods such as Cholesky decomposition [41]. Furthermore, across multiple LMA iterations in the same optimization iteration, the main change to the linear system is the damping factor. Therefore, in future work, it may be beneficial to investigate alternative methods to speed up this backtracking process, as demonstrated by RootBA which uses QR decomposition [27].

6.1.5 Choosing Optimal Parameters

As seen in Chapter 5, using an iterative solving method such as PCG can provide a large speedup for visual-inertial bundle adjustment problems, but may result in worse error reduction since the computed pose parameter update is inexact. To compensate for this, we may consider more effective preconditioners or adjusting LMA and PCG parameters, including the initial damping factor, number of iterations, and stopping criteria. However, choosing these parameters to improve the error reduction for large problems may adversely impact the speedup for small problems due to unnecessary iterations. Therefore, we consider two types of approaches in order to balance the error reduction and execution performance. The first is a learning-based approach, in which an online model is used to determine optimal parameters based on the problem size and structure. The second is to focus on directly improving the PCG strategy used, by developing alternative preconditioners [26] and modifying the

algorithm to terminate early when stagnation is detected. It may also be advantageous to further investigate the relationship between the inexact step computed by the block solver and the overall error reduction quality achieved by the outer algorithm.

6.1.6 Beyond Bundle Adjustment

The methods we have developed can be applied to other types of non-linear optimization problems, as well as problems involving sparse matrix operations. SLAM systems such as ORB-SLAM3 perform other types of g2o-based optimizations, including inertial-only optimization, pose-graph optimization, and essential graph optimization [1]. Beyond SLAM, examples of interesting non-linear least squares optimization problems in vision and graphics include human pose estimation [59], mesh and volumetric deformation, optical flow, recovery of shape from image shading [57], and blendshape fitting [58]. Determining how well our methods perform for these requires additional experiments, which would involve porting the necessary constraints to a test application.

Bibliography

[1] Carlos Campos, Richard Elvira, Juan J. Gómez Rodríguez, José M. M. Montiel, and Juan D. Tardós. "ORB-SLAM3: An Accurate Open-Source Library for Visual, Visual–Inertial, and Multimap SLAM". In: *IEEE Transactions on Robotics* 37.6 (2021), pp. 1874–1890. DOI: 10.1109/TRO.2021.3075644.

[2] Bill Triggs, Philip F McLauchlan, Richard I Hartley, and Andrew W Fitzgibbon. "Bundle adjustment—a modern synthesis". In: *International workshop on vision algorithms*. Springer. 1999, pp. 298–372.

[3] Sameer Agarwal, Noah Snavely, Steven M Seitz, and Richard Szeliski. "Bundle adjustment in the large". In: *European conference on computer vision*. Springer. 2010, pp. 29–42.

[4] Kurt Konolige and Motilal Agrawal. "FrameSLAM: From bundle adjustment to real-time visual mapping". In: *IEEE Transactions on Robotics* 24.5 (2008), pp. 1066–1077.

[5] Rainer Kümmerle, Giorgio Grisetti, Hauke Strasdat, Kurt Konolige, and Wolfram Burgard. "g2o: A general framework for graph optimization". In: *2011 IEEE International Conference on Robotics and Automation*. 2011, pp. 3607–3613. DOI: 10.1109/ICRA.2011.5979949.

[6] Frank Dellaert. *Factor graphs and GTSAM: A hands-on introduction*. Tech. rep. Georgia Institute of Technology, 2012.

[7] Sameer Agarwal and Keir Mierle. "Ceres solver: Tutorial & reference". In: *Google Inc* 2.72 (2012), p. 8.

[8] Raúl Mur-Artal, J. M. M. Montiel, and Juan D. Tardós. "ORB-SLAM: a Versatile and Accurate Monocular SLAM System". In: *IEEE Transactions on Robotics* 31.5 (2015), pp. 1147–1163. DOI: 10.1109/TRO.2015.2463671.

[9] Michael Burri et al. "The EuRoC micro aerial vehicle datasets". In: *The International Journal of Robotics Research* (2016). DOI: 10.1177/0278364915620033. eprint: http://ijr.sagepub.com/content/early/2016/01/21/0278364915620033.full.pdf+html. URL: http://ijr.sagepub.com/content/early/2016/01/21/0278364915620033.abstract.

[10] D. Schubert, T. Goll, N. Demmel, V. Usenko, J. Stueckler, and D. Cremers. "The TUM VI Benchmark for Evaluating Visual-Inertial Odometry". In: *International Conference on Intelligent Robots and Systems (IROS)*. Oct. 2018.

[11] Shishir Gopinath, Karthik Dantu, and Steven Y. Ko. "Improving the Performance of Local Bundle Adjustment for Visual-Inertial SLAM with Efficient Use of GPU Resources". In: *2023 IEEE International Conference on Robotics and Automation (ICRA)*. 2023, to be published.

[12] Manolis I.A. Lourakis and Antonis A. Argyros. "SBA: A software package for generic sparse bundle adjustment". In: *ACM Transactions on Mathematical Software* 36 (1 Mar. 2009). ISSN: 00983500. DOI: 10.1145/1486525.1486527.

[13] Qiang Liu, Shuzhen Qin, Bo Yu, Jie Tang, and Shaoshan Liu. "π-BA: Bundle Adjustment Hardware Accelerator based on Distribution of 3D-Point Observations". In: *IEEE transactions on computers* (2020), pp. 1–1. ISSN: 0018-9340. DOI: 10.1109/TC.2020.2984611.

[14] Jie Ren, Wenteng Liang, Ran Yan, Luo Mai, Shiwen Liu, and Xiao Liu. "MegBA: A GPU-Based Distributed Library for Large-Scale Bundle Adjustment". In: *European Conference on Computer Vision*. 2022.

[15] Changchang Wu, Sameer Agarwal, Brian Curless, and Steven M. Seitz. "Multicore bundle adjustment". In: *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. IEEE Computer Society, 2011, pp. 3057–3064. ISBN: 9781457703942. DOI: 10.1109/CVPR.2011.5995552.

[16] The Khronos® Group Inc. *Vulkan 1.3 Specification*. 2023. URL: https://registry.khronos.org/vulkan/specs/1.3-extensions/html/vkspec.html (visited on 03/29/2023).

[17] John Kessenich, Dave Baldwin, and Randi Rost. *The OpenGL ® Shading Language, Version 4.60.7*. 2019. URL: https://registry.khronos.org/OpenGL/specs/gl/GLSLangSpec.4.60.pdf.

[18] NVIDIA. *CUDA C++ Best Practices Guide*. 2023. URL: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/ (visited on 03/29/2023).

[19] ARM. *Arm® GPU Best Practices Developer Guide Version 3.1*. 2023. URL: https://developer.arm.com/documentation/101897/latest/.

[20] Neil Henning. *Vulkan Subgroup Tutorial*. 2018. URL: https://www.khronos.org/blog/vulkan-subgroup-tutorial.

[21] Christian Forster, Luca Carlone, Frank Dellaert, and Davide Scaramuzza. "On-Manifold Preintegration for Real-Time Visual-Inertial Odometry". In: *IEEE Transactions on Robotics* 33 (1 Feb. 2017), pp. 1–21. ISSN: 15523098. DOI: 10.1109/TRO.2016.2597321.

[22] Tong Qin, Peiliang Li, and Shaojie Shen. "VINS-Mono: A Robust and Versatile Monocular Visual-Inertial State Estimator". In: *IEEE Transactions on Robotics* 34 (4 Aug. 2018), pp. 1004–1020. ISSN: 15523098. DOI: 10.1109/TRO.2018.2853729.

[23] Antoni Rosinol, Marcus Abate, Yun Chang, and Luca Carlone. "Kimera: an Open-Source Library for Real-Time Metric-Semantic Localization and Mapping". In: IEEE, May 2020, pp. 1689–1696. ISBN: 978-1-7281-7395-5. DOI: 10.1109/ICRA40945.2020.9196885. URL: https://ieeexplore.ieee.org/document/9196885/.

[24] Raúl Mur-Artal and Juan D. Tardós. "ORB-SLAM2: an Open-Source SLAM System for Monocular, Stereo and RGB-D Cameras". In: *IEEE Transactions on Robotics* 33.5 (2017), pp. 1255–1262. DOI: 10.1109/TRO.2017.2705103.

[25] Sameer Agarwal, Keir Mierle, and The Ceres Solver Team. *Ceres Solver*. Version 2.1. Mar. 2022. URL: https://github.com/ceres-solver/ceres-solver.

[26] Avanish Kushal and Sameer Agarwal. "Visibility Based Preconditioning for bundle adjustment". In: *2012 IEEE Conference on Computer Vision and Pattern Recognition*. 2012, pp. 1442–1449. DOI: 10.1109/CVPR.2012.6247832.

[27] Nikolaus Demmel, Christiane Sommer, Daniel Cremers, and Vladyslav Usenko. "Square Root Bundle Adjustment for Large-Scale Reconstruction". In: (Mar. 2021). DOI: 10.1109/CVPR46437.2021.01155. URL: http://arxiv.org/abs/2103.01843%20http://dx.doi.org/10.1109/CVPR46437.2021.01155.

[28] Siddharth Choudhary, Shubham Gupta, and P. J. Narayanan. *Practical Time Bundle Adjustment for 3D Reconstruction on the GPU*. 2012. DOI: 10.1007/978-3-642-35740-4_33. URL: http://link.springer.com/10.1007/978-3-642-35740-4_33.

[29] Ronny Hänsch, Igor Drude, and Olaf Hellwich. "MODERN METHODS OF BUNDLE ADJUSTMENT ON THE GPU". In: *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences* III-3 (June 2016), pp. 43–50. ISSN: 2194-9050. DOI: 10.5194/isprs-annals-III-3-43-2016. URL: https://www.isprs-ann-photogramm-remote-sens-spatial-inf-sci.net/III-3/43/2016/.

[30] Mingwei Cao, Liping Zheng, Wei Jia, and Xiaoping Liu. "Fast incremental structure from motion based on parallel bundle adjustment". In: vol. 18. Springer Science and Business Media Deutschland GmbH, Apr. 2021, pp. 379–392. DOI: 10.1007/s11554-020-00970-3.

[31] Stanimire Tomov, Jack Dongarra, and Marc Baboulin. "Towards dense linear algebra for hybrid GPU accelerated manycore systems". In: *Parallel Computing* 36.5-6 (June 2010), pp. 232–240. ISSN: 0167-8191. DOI: 10.1016/j.parco.2009.12.005.

[32] Fixstars Corporation. *Cuda Bundle Adjustment*. 2022. URL: https://github.com/fixstars/cuda-bundle-adjustment.

[33] Andreas Geiger, Philip Lenz, and Raquel Urtasun. "Are we ready for Autonomous Driving? The KITTI Vision Benchmark Suite". In: *Conference on Computer Vision and Pattern Recognition (CVPR)*. 2012.

[34] Jingwei Huang, Shan Huang, and Mingwei Sun. "DeepLM: Large-scale Nonlinear Least Squares on Deep Learning Frameworks using Stochastic Domain Decomposition". In: IEEE Computer Society, 2021, pp. 10303–10312. ISBN: 9781665445092. DOI: 10.1109/CVPR46437.2021.01017.

[35] Rongdi Sun, Peilin Liu, Jianwei Xue, Shiyu Yang, Jiuchao Qian, and Rendong Ying. "BAX: A Bundle Adjustment Accelerator with Decoupled Access/Execute Architecture for Visual Odometry". In: *IEEE Access* 8 (2020), pp. 75530–75542. ISSN: 21693536. DOI: 10.1109/ACCESS.2020.2988527.

[36] Jinwoo Jeon, Sungwook Jung, Eungchang Lee, Duckyu Choi, and Hyun Myung. "Run Your Visual-Inertial Odometry on NVIDIA Jetson: Benchmark Tests on a Micro Aerial Vehicle". In: *IEEE robotics and automation letters* 6 (3 2021), pp. 5332–5339. ISSN: 2377-3766. DOI: 10.1109/LRA.2021.3075141.

[37] Tianji Ma et al. "Research on the Application of Visual SLAM in Embedded GPU". In: *Wireless communications and mobile computing* 2021 (2021), pp. 1–17. ISSN: 1530-8669. DOI: 10.1155/2021/6691262.

[38] Quan Lu, Jianli Xu, Likun Hu, and Minghui Shi. "Parallel VINS-Mono algorithm based on GPUs in embedded devices". In: *International Journal of Advanced Robotic Systems* 19 (1 Jan. 2022). ISSN: 17298814. DOI: 10.1177/17298814221074534.

[39] Jianhua Gao et al. "A Systematic Survey of General Sparse Matrix-matrix Multiplication". In: *ACM Computing Surveys* 55 (12 Dec. 2023), pp. 1–36. ISSN: 0360-0300. DOI: 10.1145/3571157.

[40] Sandra Carney, Michael A. Heroux, Guangye Li, Roldan Pozo, Karin A. Remington, and Kesheng Wu. *A Revised Proposal for a Sparse BLAS Toolkit*. 1996, pp. 1–34. URL: https://sdm.lbl.gov/~kewu/ps/SparseBLAS96.pdf.

[41] Lukas Polok, Marek Solony, Pavel Smrz, Viorela Ila, and Pavel Zemcik. "Incremental cholesky factorization for least squares problems in robotics?" In: vol. 8. IFAC Secretariat, 2013, pp. 172–178. ISBN: 9783902823366. DOI: 10.3182/20130626-3-AU-2035.00027.

[42] Lukas Polok, Marek Solony, Viorela Ila, Pavel Smrz, and Pavel Zemcik. "Efficient implementation for block matrix operations for nonlinear least squares problems in robotic applications". In: 2013, pp. 2263–2269. ISBN: 9781467356411. DOI: 10.1109/ICRA.2013.6630883.

[43] NVIDIA. *cuSPARSE Documentation*. URL: https://docs.nvidia.com/cuda/cusparse/ (visited on 04/20/2023).

[44] L Susan Blackford et al. "An updated set of basic linear algebra subprograms (BLAS)". In: *ACM Transactions on Mathematical Software* 28 (2 June 2002), pp. 135–151. ISSN: 0098-3500. DOI: 10.1145/567806.567807. URL: https://dl.acm.org/doi/10.1145/567806.567807.

[45] NVIDIA. *cuBLAS Documentation*. URL: https://docs.nvidia.com/cuda/cublas/ (visited on 04/20/2023).

[46] The Institute for Ethical AI and Machine Learning. *Kompute Framework*. 2022. URL: https://github.com/KomputeProject/kompute.

[47] The Khronos® Group Inc. *Vulkan*. 2022. URL: https://www.vulkan.org (visited on 03/29/2023).

[48] Gaël Guennebaud, Benoît Jacob, et al. *Eigen v3*. http://eigen.tuxfamily.org. 2010.

[49] Kurt Konolige. "Sparse Sparse Bundle Adjustment". In: *Proceedings of the British Machine Vision Conference (BMVC)*. 2010.

[50] M. Masmano, I. Ripoll, A. Crespo, and J. Real. "TLSF: a new dynamic memory allocator for real-time systems". In: *Proceedings. 16th Euromicro Conference on Real-Time Systems, 2004. ECRTS 2004.* 2004, pp. 79–88. DOI: 10.1109/EMRTS.2004.1311009.

[51] AMD. *Vulkan Memory Allocator*. 2022. URL: https://github.com/GPUOpen-LibrariesAndSDKs/VulkanMemoryAllocator.

[52] Ryan Eberhardt and Mark Hoemmen. "Optimization of block sparse matrix-vector multiplication on shared-memory parallel architectures". In: Institute of Electrical and Electronics Engineers Inc., July 2016, pp. 663–672. ISBN: 9781509021406. DOI: 10.1109/IPDPSW.2016.42.

[53] Steven C. Rennich, Darko Stosic, and Timothy A. Davis. "Accelerating sparse Cholesky factorization on GPUs". In: *Parallel Computing* 59 (Nov. 2016), pp. 140–150. ISSN: 01678191. DOI: 10.1016/j.parco.2016.06.004.

[54] R. Barrett et al. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. Philadelphia, PA: SIAM, 1994. URL: https://netlib.org/linalg/html_templates/Templates.html.

[55] Michael Grupp. *evo: Python package for the evaluation of odometry and SLAM.* https://github.com/MichaelGrupp/evo. 2017.

[56] The Khronos® Group Inc. *SYCL*. 2023. URL: https://www.khronos.org/sycl/ (visited on 03/29/2023).

[57] Zachary Devito et al. "Opt: A domain specific language for non-linear least squares optimization in graphics and imaging". In: *ACM Transactions on Graphics* 36 (5 Oct. 2017). ISSN: 15577368. DOI: 10.1145/3132188.

[58] Michael Mara, Felix Heide, Michael Zollhöfer, Matthias Nießner, and Pat Hanrahan. "Thallo – Scheduling for High-Performance Large-Scale Non-Linear Least-Squares Solvers". In: *ACM Transactions on Graphics* 40 (5 Oct. 2021), pp. 1–14. ISSN: 0730-0301. DOI: 10.1145/3453986. URL: https://dl.acm.org/doi/10.1145/3453986.

[59] Haixun Sun, Yanyan Zhang, Yijie Zheng, Jianxin Luo, and Zhisong Pan. "G2O-Pose: Real-Time Monocular 3D Human Pose Estimation Based on General Graph Optimization". In: *Sensors* 22 (21 Nov. 2022). ISSN: 14248220. DOI: 10.3390/s22218335.