2023 IEEE International Conference on Robotics and Automation (ICRA 2023)
May 29 - June 2, 2023. London, UK

Improving the Performance of Local Bundle Adjustment for Visual-Inertial SLAM with Efficient Use of GPU Resources

Shishir Gopinath[1], Karthik Dantu[2], Steven Y. Ko[1]

***Abstract*— In this paper, we present our approach to efficiently leveraging GPU resources to improve the performance of *local bundle adjustment* for visual-inertial SLAM. We observe that for local bundle adjustment (i) the *Schur complement method*, a technique often used to speed up bundle adjustment, has the largest overhead when solving for the parameter update, and (ii) the workload consists of operations on small- to medium-sized matrices. Based on these observations, we develop and combine several techniques that efficiently handle small- to medium-sized matrices. We then implement these techniques as a drop-in replacement block solver for g2o, a library frequently used for bundle adjustment, and integrate it with ORB-SLAM3, a well-known open-source visual-inertial SLAM system. Our evaluation done with two popular datasets, EuRoC and TUM-VI, shows that we can reduce the time taken by local bundle adjustment by 13.81%-33.79% with our techniques across an embedded device and a desktop machine.**

I. 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 color 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/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 a technique called *bundle adjustment* (BA). Bundle adjustment resolves these inconsistencies by minimizing a cost function [2] over error constraints. 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].

[1]Simon Fraser University, Canada
[2]University at Buffalo, USA

Modern visual-inertial SLAM systems [1] perform local 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. Therefore, we consider it desireable to optimize the performance of local bundle adjustment as 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 paper, 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 of local visual-inertial bundle adjustment. First, when solving for the parameter update in local visual-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 [8] and TUM-

979-8-3503-2365-8/23/$31.00 ©2023 IEEE

VI [9], for our evaluation. Our GPU implementation uses the Vulkan API, via Kompute [10], due to its popularity and cross-platform support (including support for desktop GPUs, mobile GPUs, and all major OSes) [11].

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

II. Related Work

In this section, we discuss the existing literature related to bundle adjustment using GPU and FPGA hardware.

Bundle Adjustment Hardware Acceleration: Hänsch et al. [12] implement second-order Levenberg-Marquardt, as well as first-order non-linear conjugate gradient descent and alternating resection-intersection methods for GPU bundle adjustment using CUDA. Their experiments on large-scale datasets show that the first-order methods converge faster but do not reduce the mean squared reprojection error as well as second-order methods. However, the applicability of these methods for smaller online BA problems may vary, and furthermore, desktop GPUs are not subject to the constraints of embedded devices. Embedded GPUs have comparatively fewer cores and lack the dedicated memory found on desktop GPUs [13]. Therefore, it is unknown if these algorithms are suitable for smaller devices in real-time scenarios.

An alternative to GPU BA acceleration is demonstrated by Liu et al. [14]. They propose π-BA, an FPGA implementation of visual BA, and target the Schur complement as part of their optimizations [14]. Their double-precision implementation is 7.3 times faster than a single-threaded Ceres Solver-based implementation on ARM, and is energy-efficient, although they find that limited on-chip memory is a major constraint [14]. Still, this method assumes a fixed BA problem structure with only visual constraints. A more general solution is needed for SLAM systems which rely on non-visual information, e.g., inertial input.

Embedded GPU Acceleration: Ma et al. [15] investigate the applicability of embedded GPUs for visual SLAM. They implement CUDA programs to parallelize ORB feature extraction and feature matching. They integrate their improvements into ORB-SLAM2, and evaluate the performance on a Jetson TX2 using EuRoC Machine Hall datasets. They show that their implementation decreases the computation time by approximately $1/3$, allowing for a larger number of frames to be processed per second [15]. This comes at the cost of producing larger root mean square trajectory errors, although incorporating inertial information may alleviate this [15]. The results of their experiments demonstrate how parallel GPU algorithms can significantly improve the performance of visual SLAM.

III. Background

Since our main contribution is in developing techniques that efficiently perform local visual-inertial bundle adjustment, we provide a brief overview. We discuss three important aspects of local visual-inertial bundle adjustment directly relevant to our approach—local 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].

Local 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) [16].

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 large 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 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]:

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

$$b_{Schur} = -b_p + H_{pl}H_{ll}^{-1}b_l \tag{2}$$

$$\Delta x_l = H_{ll}^{-1}(-b_l - H_{pl}^T\Delta x_p) \tag{3}$$

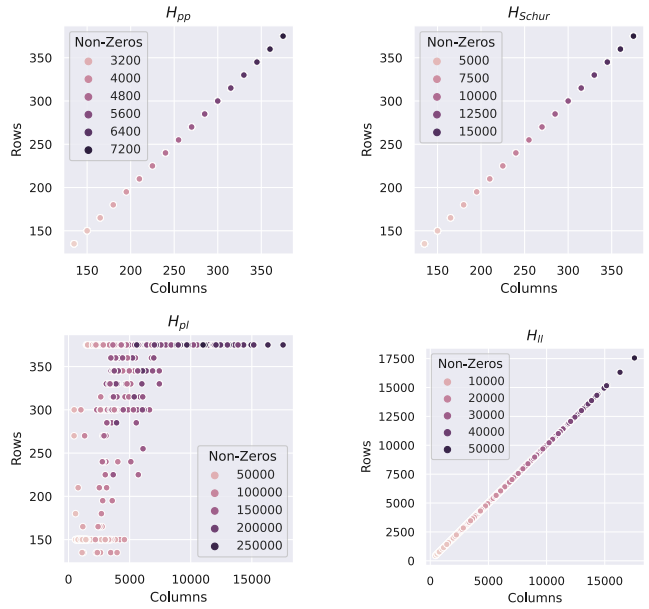


Fig. 1: The dimensions and number of non-zeros of sparse matrices for local-inertial bundle adjustment across indoor and outdoor EuRoC and TUM-VI sequences. Only the upper triangular blocks are needed for H_{pp} and H_{Schur}. Larger-scale problems, such as the 52 image, 64053 landmark Venice dataset from BAL [3], generate as many as 9373671 non-zeros in the H_{pl} matrix.

IV. Our Approach

Motivation: As mentioned in Section I, 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 V (Figure 6 and Figure 7). 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 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 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 III. 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*.

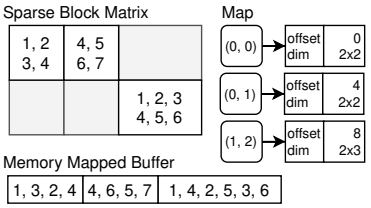


Fig. 2: A sparse block matrix example with three filled-in blocks (top left). A map that represents the blocks (top right). How the matrix is stored in GPU memory (bottom).

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 [17]. 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 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 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-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

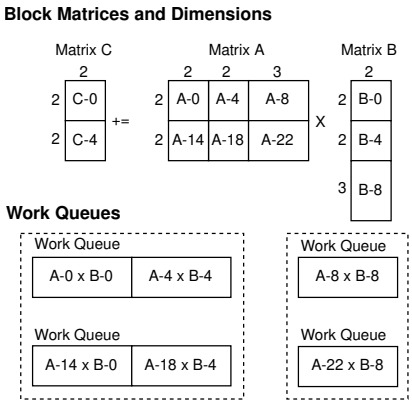


Fig. 3: Multiplication between sparse block matrices generates a set of work queues. Work queues which multiply blocks of the same dimensions and do not write to the same block in matrix C are grouped together for processing by a single shader dispatch.

for cache efficiency [5], [18]. A key change in visual-inertial bundle adjustment is that these blocks are not uniform in size. The sizes are determined by the input graph's vertices (as in g2o). An example is shown in Figure 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 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 GPU computation program (called 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 handle blocks with different

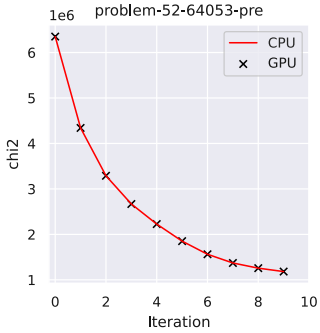


Fig. 4: χ^2 error when we run the original CPU version and our GPU version of block solver on the Venice BAL dataset for ten iterations. The behaviors match each other's.

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 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. As mentioned, a barrier is used to serialize the execution of the first and second queues.

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 [19] as provided by VulkanMemoryAllocator (VMA) [20], 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 V.

V. Evaluation

A. Experimental setup

We implement our techniques as a drop-in replacement block solver for g2o and integrate our modifications into

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 I: Average local BA run times (in ms) for ORB-SLAM3 on the desktop machine.

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 II: Average local BA run times (in ms) for ORB-SLAM3 on the Jetson Xavier NX.

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 8GB 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 and observe matching behaviour.

B. 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 $5.08\times$ for the Schur complement step before the linear solver step on V201, and a $2.87\times$ speedup on the Jetson (Figure 6). 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.

C. Overall Performance of Local Bundle Adjustment

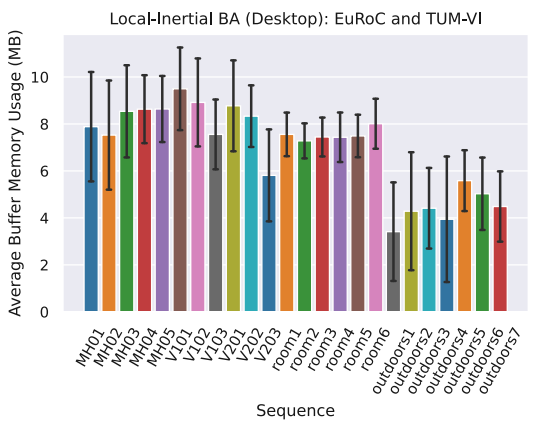
Table I and Table II show the running times of the local-inertial bundle adjustment averaged over five trials. Results show that our GPU solver *consistently reduces* the average local BA execution time for both platforms. As the workload for local BA is partly determined by visible landmarks and co-visible 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 7 shows an averaged breakdown of local-inertial bundle adjustment for EuRoC V201. The block solver performance improvement is reflected in the segment for optimization.

D. GPU Memory Usage

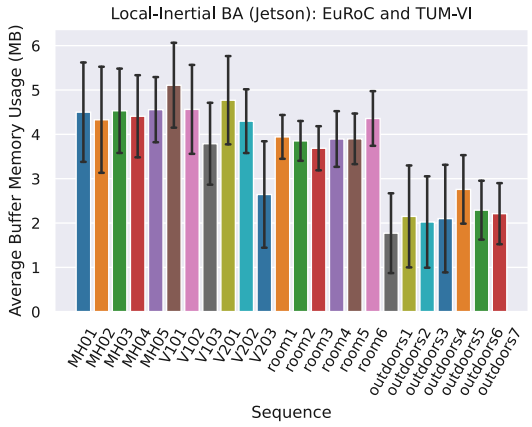
Figure 5 shows the average amount of memory occupied by GPU buffer allocations for local-inertial BA, totalled over all memory heaps. Memory usage is larger on the desktop machine due to the use of device-local memory. On both machines, TUM-VI outdoors sequences use the least amount of memory, while 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.

E. Effect of Memory Allocation Strategies

As mentioned in Section IV, we have discovered that the GPU memory allocation method influences the performance of bundle adjustment, especially for longer SLAM sequences. Figure 8 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.



(a) Desktop



(b) Jetson

Fig. 5: 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.

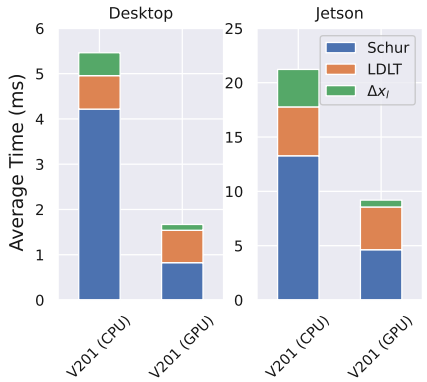


Fig. 6: Average execution time of the main solving steps for an iteration of the block solver for the EuRoC V201 sequence.

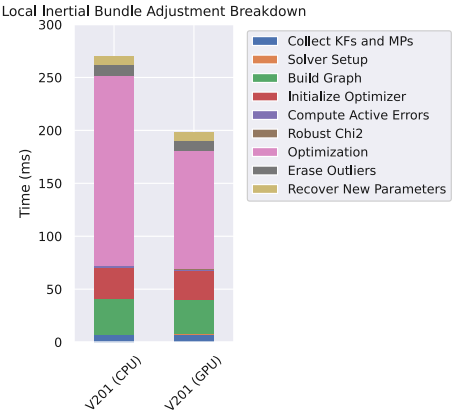


Fig. 7: A breakdown of the local-inertial bundle adjustment call for a run of EuRoC V201 using each solver on the Xavier NX.

F. 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 IV), 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

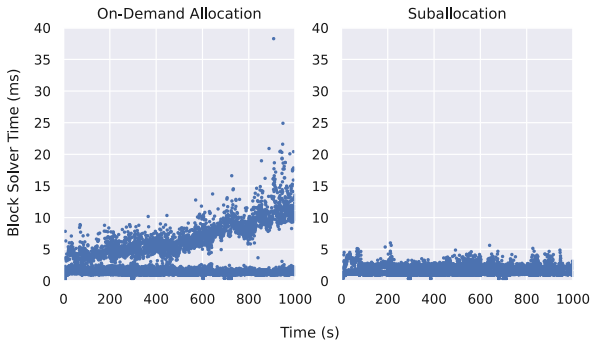


Fig. 8: The impact of the GPU memory allocation method on the block solver performance (desktop), for local-inertial BA on the TUM-VI outdoors6 sequence (first 1000 seconds). Creating on-demand allocations for each buffer as needed results in performance degradation over longer sequences. Using VMA to suballocate memory from larger blocks avoids this overhead.

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 [21], [12] have been shown to be more suitable for GPU execution.

VI. Conclusion

In this paper, we have shown how we use GPU resources to improve the performance of local bundle adjustment for visual-inertial SLAM. Our techniques specifically target small- to medium-sized matrices and perform GPU operations efficiently, mainly for the Schur complement. We implement our techniques as a drop-in replacement block solver for g2o, and integrate it with ORB-SLAM3. Our evaluation with two datasets, EuRoC and TUM-VI, shows that we can reduce the amount of time local bundle adjustment takes by up to 33.79% across indoor and outdoor SLAM sequences on different platforms (a desktop and an embedded board). Our results show that customizing solutions for different matrix sizes helps improve the performance of bundle adjustment, and point to a need for techniques that can gracefully handle small-scale matrices to large-scale matrices together.

REFERENCES

[1] C. Campos, R. Elvira, J. J. G. Rodríguez, J. M. M. Montiel, and J. D. Tardós, "Orb-slam3: An accurate open-source library for visual, visual–inertial, and multimap slam," *IEEE Transactions on Robotics*, vol. 37, no. 6, pp. 1874–1890, 2021.

[2] B. Triggs, P. F. McLauchlan, R. I. Hartley, and A. W. Fitzgibbon, "Bundle adjustment—a modern synthesis," in *International workshop on vision algorithms*. Springer, 1999, pp. 298–372.

[3] S. Agarwal, N. Snavely, S. M. Seitz, and R. Szeliski, "Bundle adjustment in the large," in *European conference on computer vision*. Springer, 2010, pp. 29–42.

[4] K. Konolige and M. Agrawal, "Frameslam: From bundle adjustment to real-time visual mapping," *IEEE Transactions on Robotics*, vol. 24, no. 5, pp. 1066–1077, 2008.

[5] R. Kümmerle, G. Grisetti, H. Strasdat, K. Konolige, and W. Burgard, "g2o: A general framework for graph optimization," in *2011 IEEE International Conference on Robotics and Automation*, 2011, pp. 3607–3613.

[6] F. Dellaert, "Factor graphs and gtsam: A hands-on introduction," Georgia Institute of Technology, Tech. Rep., 2012.

[7] S. Agarwal and K. Mierle, "Ceres solver: Tutorial & reference," *Google Inc*, vol. 2, no. 72, p. 8, 2012.

[8] M. Burri, J. Nikolic, P. Gohl, T. Schneider, J. Rehder, S. Omari, M. W. Achtelik, and R. Siegwart, "The euroc micro aerial vehicle datasets," *The International Journal of Robotics Research*, 2016. [Online]. Available: http://ijr.sagepub.com/content/early/2016/01/21/0278364915620033.abstract

[9] 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)*, October 2018.

[10] The Institute for Ethical AI and Machine Learning, "Kompute framework," 2022. [Online]. Available: https://github.com/KomputeProject/kompute

[11] The Khronos® Group Inc., "Vulkan," 2022. [Online]. Available: https://www.vulkan.org

[12] R. Hänsch, I. Drude, and O. Hellwich, "Modern methods of bundle adjustment on the gpu," *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, vol. III-3, pp. 43–50, 6 2016. [Online]. Available: https://www.isprs-ann-photogramm-remote-sens-spatial-inf-sci.net/III-3/43/2016/

[13] J. Jeon, S. Jung, E. Lee, D. Choi, and H. Myung, "Run your visual-inertial odometry on nvidia jetson: Benchmark tests on a micro aerial vehicle," *IEEE robotics and automation letters*, vol. 6, pp. 5332–5339, 2021.

[14] Q. Liu, S. Qin, B. Yu, J. Tang, and S. Liu, "π-ba: Bundle adjustment hardware accelerator based on distribution of 3d-point observations," *IEEE transactions on computers*, pp. 1–1, 2020.

[15] T. Ma, N. Bai, W. Shi, X. Wu, L. Wang, T. Wu, and C. Zhao, "Research on the application of visual slam in embedded gpu," *Wireless communications and mobile computing*, vol. 2021, pp. 1–17, 2021.

[16] M. I. Lourakis and A. A. Argyros, "Sba: A software package for generic sparse bundle adjustment," *ACM Transactions on Mathematical Software*, vol. 36, 3 2009.

[17] G. Guennebaud, B. Jacob, *et al.*, "Eigen v3," http://eigen.tuxfamily.org, 2010.

[18] K. Konolige, "Sparse sparse bundle adjustment," in *Proceedings of the British Machine Vision Conference (BMVC)*, 2010.

[19] 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.

[20] AMD, "Vulkan memory allocator," 2022. [Online]. Available: https://github.com/GPUOpen-LibrariesAndSDKs/VulkanMemoryAllocator

[21] C. Wu, S. Agarwal, B. Curless, and S. 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.