IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received March 23, 2020, accepted April 10, 2020, date of publication April 17, 2020, date of current version May 5, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2988527

BAX: A Bundle Adjustment Accelerator With Decoupled Access/Execute Architecture for Visual Odometry

RONGDI SUN, PEILIN LIU, (Senior Member, IEEE), JIANWEI XUE, SHIYU YANG, JIUCHAO QIAN, AND RENDONG YING, (Member, IEEE)
School of Electronic and Electrical Engineering, Shanghai Jiao Tong University, Shanghai 200240, China

Corresponding author: Peilin Liu (liupeilin@sjtu.edu.cn)

This work was supported by the National Natural Science Foundation of China under Grant 61903246.

ABSTRACT As the demand for embedded-vision grows, solving large optimization problems in real-time with energy and cost budget is a challenge. We present BAX, a hardware accelerator of bundle adjustment (BA), which solves the least-squares problem of state estimation in visual odometry (VO). BAX consists of a frontend and a backend for control and computation, respectively. The frontend generates instructions on-the-fly executed at the backend to perform the BA algorithm. The backend adopts decoupled access/execute (DAE) architecture, which separates the memory access unit (MAU) from the pipeline. The MAU can prefetch vectors and matrices ahead of computations. To further reduce the latency of data reorganization, three transpose-free dataflows are proposed for matrix multiplication operations on the vector processing unit (VPU). Besides, a unified architecture for both forward and backward substitution is designed for matrix decomposition in the linear solver. All the data are stored in 442kB on-chip memory, and the local map is maintained efficiently by the hierarchical graph memory. Compared with the baseline architecture, the processing time is reduced by 53.9% through the above techniques. BAX is implemented in 32-bit floating-point precision with data normalization on FPGA. It completes a full BA in about 63.44ms at 200MHz, consuming 1.12W power. BAX is 1.73× and 22.38× faster than the desktop and embedded CPUs, respectively, and achieves 90% performance of the GPU at much less power consumption.

INDEX TERMS Hardware accelerator, decoupled architecture, FPGA, embedded system, bundle adjustment, visual odometry.

I. INTRODUCTION

Bundle adjustment (BA) is the problem of refining a visual reconstruction or navigation to produce jointly optimal 3D structure and viewing parameter (camera pose and calibration) estimates [1]. It refers to bundling up the light from each 3D point to the optical center and adjusting to minimize the projection errors of all points concerning both structure and viewing parameters. Compared to the filter-based optimization approach [2], BA trades more computational cost for higher accuracy [3]. It has become the last step of almost every feature-based visual tracking system and 3D reconstruction algorithm. Image features are the primitives such as points, lines, and regions with distinctive geometric and structural information. A general framework of visual navigation is visual odometry (VO) [4] or visual simultaneous localization and mapping (V-SLAM). VO is the pose estimation process of an agent by matching the features extracted at a stream of images. The poses over multiple frames describe the agent's long-term trajectory, and the points denoted by the features constructs a local map of the surrounding. The trajectory and local map can be further refined by BA [5]. V-SLAM usually takes VO as the frontend and performs a large global BA for loop-closure at the backend. This framework is widely used in applications such as unmanned aerial vehicles (UAVs), automatic driving, and argument reality (AR). A typical method for 3D reconstruction is Structure-from-Motion (SfM) [6]. SfM has a similar pipeline to V-SLAM, but the purpose is to restore the 3D structure of observations rather than estimate the agent's poses. Compared to SfM and V-SLAM, VO performs local BA with fewer frames and a smaller map but has the requirement of real-time processing.

The associate editor coordinating the review of this manuscript and approving it for publication was Yasar Amin.

TABLE 1. Processing time breakdown of VO.

Task	Time(ms)	Percentage
Feature extraction	237 / frame×15 frames	67%
Feature matching	11 / frame×15 frames	3.11%
Pose estimation	15 / frame×15 frames	4.24%
Local BA	1361 / keyframe	25.65%
Total	5306	
Avg. FPS	2.83	

TABLE 2. Processing time breakdown of BA.

Task	Time	Operation
Jacobian	4.16%	scalar arithmetic
Hessian	36.45%	matrix mul/add
Schur	30.93%	matrix inv/mul/add
Cholesky	8.76%	matrix decomposition
Update	3.19%	scalar arithmetic, matrix add
Total	83.49%	-

Large scale BA in offline visual reconstruction like SfM usually requires high-performance CPUs, GPUs, and even distributed computing [7]–[11]. These solutions are inappropriate for embedded systems because of the high power and hardware cost. For online visual navigation in embedded systems, BA is still a critical issue due to its high computational intensity and numerical precision. We profile a VO program with ORB feature [12] and local BA on an ARM CPU. As shown in Table 1, it achieves only less than three fps, and BA consumes over 25% of the total time. Many researchers have studied hardware acceleration of image feature extraction, which is beyond the scope of this work. This work focuses on the performance bottleneck of BA. Unlike many image processing algorithms, VO is not a streaming processing application. For a full-pipelined VO, the additional data memory is required as ping-pong buffers between pipeline stages. It will cost much on-chip memory and off-chip bandwidth, which should be avoided for embedded applications. Therefore, in a non-pipelined VO system, the latency of BA still drags the processing speed so that the optimized parameters may not be updated in real-time.

The profiling result of a BA program is also given in Table 2. It indicates that matrix operations, including multiplication, addition, inverse and decomposition, occupy near 80% of the total run time. Previous matrix processors can be classified into two categories. One uses 2D register array to access matrix by row and by column [13]. This architecture indexes each entry and each element in an entry. It brings additional hardware and power cost due to the complex addressing logics. Another one operates like SIMD or vector machines [14]. When data is loaded to the register file or feed to the processing unit, it has an initial latency that rearranging the data to fit the SIMD lanes. BA has many operations that a matrix is transposed and then multiplied to another. Thus, previous solutions are less optimal in hardware cost or performance for BA due to its particular computing pattern.

Besides, previous hardware acceleration of BA [15]–[19] still have shortcomings for feature-based embedded VO. In [15], a part of BA is accelerated by FPGA, and the intermediate data are stored in off-chip memory. The data movement results in more latency and power cost. The pose estimation engines in [16], [17] are lack of optimization process. The works in [18], [19] optimize only camera poses, and the errors caused by points are not reduced. For all these motivations, we propose BAX, a hardware accelerator running local BA in real-time for embedded VO. It addresses the issue of matrix operations, works without external memory access, and refines both camera poses and map points. The high performance of BAX is gained as a result of fourfold contribution:

- **Decouple access/execute architecture.** Matrix operations, which dominate the BA algorithm, can be performed by a vector processing unit (VPU). We adopt a decouple access/execute architecture [20] for the VPU pipeline. The initial latency of traditional single instruction multiple data (SIMD) execution is hidden by accessing memory and executing computations asynchronously.
- **Transpose-free matrix multiplication.** In the BA algorithm, a matrix is often transposed before multiplied with another one. We regulate three dataflows to perform transpose-free matrix multiplication on the VPU. Thus, the intial latency of matrix transpose is avoided. This computing paradigm can be applied to other matrix processors without much effort.
- **Unified linear system solver.** The BA algorithm constructs a linear system with a positive definite matrix. Exploiting the parallelism in the variant Cholesky matrix decomposition, we design a unified architecture for both the forward and backward substitutions. Compared with instruction-based processing, the dedicated datapath removes time-consuming data reorganization.
- **Hierarchical graph memory.** The BA algorithm uses a graph to represent the local map, including the camera poses, the 3D map points, and the 2D projections at different poses. We propose a hierarchical graph memory to maintain the local map. The memory access can be pipelined to save the timing cost when computing the projections and updating the parameters.

BAX is implemented on FPGA and operates at 200MHz. The evaluation results show that BAX improves processing speed by 1.73× and 22.38× compared with that of desktop and embedded CPUs.

The rest of this article is organized as follows. Section II reviews the related works of BA and pose estimation acceleration. Section III briefly introduces the matrix manipulations in BA. In Section IV, the architecture of BAX and its novelty is described in detail. Section V shows the experiment result about implementation, performance and accuracy. Finally, the conclusion is drawn in Section VI.

II. RELATED WORKS

A. BA ON GENERAL PURPOSE PROCESSORS

To improve computational efficiency, several researchers have comprehensively studied the optimization and implementation of BA on general-purpose processors (GPPs). Due to the lack of interaction among some subgroups of parameters, the sparsity in corresponding Jacobian matrices can be used to achieve considerable computation and memory savings. Based on this observation, Lourakis and Argyros *et al.* [7] implemented a software package to realize LM-based sparse BA with high efficiency and flexibility regarding parameterization on CPUs. To solve large-scale BA problems, Agarwal *et al.* [8] designed a truncated Newton style LM algorithm coupled with a simple preconditioner using the conjugate gradients (PCG). This method delivers high performance on advanced multicore CPUs at a fraction of the time and memory cost of methods based on factoring the Schur complement.

Given the constraints of the GPU programming model, it is not trivial to get bundle adjustment algorithms to run on the GPUs. Choudhary *et al.* [9] took a hybrid approach to run overlapping computations on GPU and CPU, where the Hessian matrices and Schur complements are constructed on GPU. Wu *et al.* [10] improved the BA performance on both CPU and GPU by inexact LM without storing block matrices in memory. Besides, single-precision arithmetic with suitable data normalization further saves memory access and timing cost yet still maintains high accuracy. Zheng *et al.* [11] also combined the PCG algorithm, the Schur complement, and GPU parallel computing technology to develop a fast and effective BA method for large-scale datasets.

B. BA HARDWARE ACCELERATORS

Although GPU solutions achieve significant performance improvement when executing the BA algorithm, they are not suitable for embedded platforms like micro UAVs with limited power budgets. This motivates researchers to design customized architecture for BA. Qin *et al.* [15] presented a hardware-software co-designed BA engine on an embedded FPGA-SoC with a novel co-observation optimization technique. The engine offloads Schur elimination to the FPGA side while runs the other steps on the CPU.

Besides, there have been several works on building hardware accelerators with the function of BA or pose estimation for complete visual tracking applications. Yoon *et al.* [16] designed a graphics and vision unified processor for AR with a pose estimation engine (PEE). The PEE calculates the device pose using an orthogonal iteration algorithm, which can be regarded as the simplicity of BA. Instead of the previous marker-based approach, Hong *et al.* [17] proposed a marker-less PEE for AR. The techniques of speculative execution and reconfigurable data-arrangement layer are used to reduce computing time and logarithmic computing to save power consumption. Li *et al.* [18] applied a convolution neural network (CNN) to extract features in SLAM and designed a CNN-SLAM processor, which executes pose-only BA in ASIC. Similarly, Suleiman *et al.* [19] presented a visual-inertial odometry (VIO) accelerator, which performs a factor graph optimization instead of BA. To the best of our knowledge, BAX is the first full hardware acceleration of BA for embedded VO that refines both camera poses and 3D map points simultaneously.

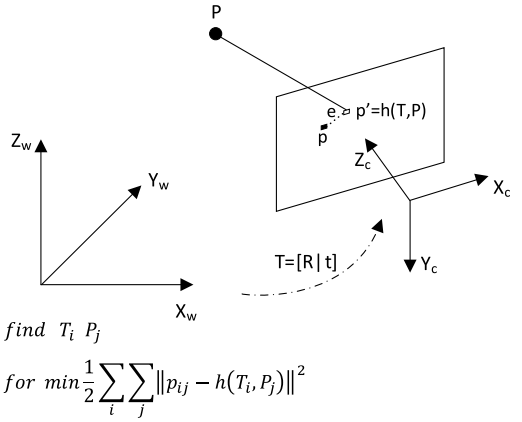


FIGURE 1. Problem formuation of BA.

III. MATRIX OPERATIONS IN BA

The basic idea of BA is shown by Fig.1. Assume the camera moves relative to the world reference by a rotation R and a translation t (i.e., camera extrinsic T) estimated using EPnP [21], then the 3D coordinates of a point P in the world reference can be transformed to the 2D coordinates of a point p' on the image plane. In addition, the same 3D point is directly captured and printed as a pixel p of the image. Due to the inaccurate measurement of T and P, p' and p have some error e in their 2D coordinates, which should have been identical in theory. Essentially, BA is a non-linear least squares problem to minimize the sum of the reprojection errors concerning all of the 3D points P_i and camera extrinsic T_j. The error is defined as the squared L_2 norm of the difference between the observed feature location and the projection of the corresponding 3D map points. The most popular method to solve BA is the Levenberg-Marquart (LM) algorithm. It expresses the non-linear optimization problem as an approximate linear system of equations and finds the optimal solution as a combination of steepest descent and the Gauss-Newton method. Using the Schur complement trick, the scale of the linear system can be reduced greatly. Since the reduced coefficient matrix is positive-define, Cholesky decomposition is suitable for solving the equations. The more details of BA and related numerical optimization methods can be found in [1]. The computing patterns and matrix operations dominated in BA is introduced as follows.

A. NORMAL EQUATION

Approximated by the LM method, the BA problem is actually to solve the normalization equations as follow:

$$\mathbf{H}\Delta\mathbf{x} = (\mathbf{J}^T\mathbf{J} + \lambda\mathbf{D}^T\mathbf{D})\Delta\mathbf{x} = -\mathbf{J}^T\mathbf{e} = \mathbf{g} \tag{1}$$

where $\mathbf{J}$ is the Jacobian, $\mathbf{D}$ is a diagonal matrix extracted from $\mathbf{J}^T\mathbf{J}$, λ is a non-negtavie coefficient that controls the damping strength, vector $\Delta\mathbf{x}$ is the corrections of optimized variables, and vector $\mathbf{e}$ is a collection of reprojection errors. Assume that the number of optimized camera poses and 3D points is a and b, respectively. Then the size of Hessian matrix $\mathbf{H} = \mathbf{J}^T\mathbf{J} + \lambda\mathbf{D}^T\mathbf{D}$ is $(6a + 3b) \times (6a + 3b)$, and the vectors $\Delta\mathbf{x}$, $\mathbf{e}$ and $\mathbf{g}$ have the same length of $(6a + 3b)$. It is also observed that all the components in Eq.1 can be partitioned into two parts: Jacobian matrix $\mathbf{J} = [\mathbf{J}_c\ \mathbf{J}_p]$, dampling matrix $\mathbf{D} = [\mathbf{D}_c\ \mathbf{D}_p]$, and corretion vector $\Delta\mathbf{x} = [\Delta\mathbf{x}_c\ \Delta\mathbf{x}_p]^T$, where the index c and p denote the camera pose part and the point part respectively. Thus, Eq.1 can be expressed as follow:

$$\begin{bmatrix} \mathbf{B} & \mathbf{E} \\ \mathbf{E}^T & \mathbf{C} \end{bmatrix} \begin{bmatrix} \Delta\mathbf{x}_c \\ \Delta\mathbf{x}_p \end{bmatrix} = \begin{bmatrix} \mathbf{v} \\ \mathbf{w} \end{bmatrix} \tag{2}$$

where $\mathbf{B} = \mathbf{J}_c^T\mathbf{J}_c + \lambda\mathbf{D}_c^T\mathbf{D}_c$ and $\mathbf{C} = \mathbf{J}_p^T\mathbf{J}_p + \lambda\mathbf{D}_p^T\mathbf{D}_p$ are a $6a \times 6a$ and a $3b \times 3b$ diagonal block matrix, $\mathbf{E} = \mathbf{J}_c^T\mathbf{J}_p$ is a $6a \times 3b$ sparse block matrix, and vector $\mathbf{v} = -\mathbf{J}_c^T\mathbf{e}$ and $\mathbf{w} = -\mathbf{J}_p^T\mathbf{e}$ has a length of $6a$ and $3b$ respectively.

B. SCHUR COMPLEMENT

Directly solving Eq.1 has a complexity of $O((6a + 3b)^3)$, which is computational intensive for large-scale BA problem. With block-wise Gaussian elimination, Eq.2 can be reduced as follow:

$$\begin{bmatrix} \mathbf{B} - \mathbf{E}\mathbf{C}^{-1}\mathbf{E}^T & \mathbf{0} \\ \mathbf{E}^T & \mathbf{C} \end{bmatrix} \begin{bmatrix} \Delta\mathbf{x}_c \\ \Delta\mathbf{x}_p \end{bmatrix} = \begin{bmatrix} \mathbf{v} - \mathbf{E}\mathbf{C}^{-1}\mathbf{w} \\ \mathbf{w} \end{bmatrix} \tag{3}$$

where $\mathbf{B} - \mathbf{E}\mathbf{C}^{-1}\mathbf{E}^T$ is a positive definite matrix, called the Schur complement. Therefore, solving Eq.1 is converted to solve the correction of camera pose part $\Delta\mathbf{x}_c$ first:

$$\left[\mathbf{B} - \mathbf{E}\mathbf{C}^{-1}\mathbf{E}^T\right]\Delta\mathbf{x}_c = \mathbf{v} - \mathbf{E}\mathbf{C}^{-1}\mathbf{w} \tag{4}$$

and then substitute $\Delta\mathbf{x}_c$ to figure out the point part $\Delta\mathbf{x}_p$:

$$\Delta\mathbf{x}_p = \mathbf{C}^{-1}\left[\mathbf{w} - \mathbf{E}^T\Delta\mathbf{x}_c\right] \tag{5}$$

Since the number of camera poses is much less than that of points, it is efficient to solve Eq.4 using Cholesky decomposition instead of Eq.1. Besides, it is simple to calculate $\mathbf{C}^{-1}$ because the diagonal of $\mathbf{C}$ is composed of 3×3 matrices.

C. CHOLESKY DECOMPOSITION

A variant Cholesky decomposition, called LDL decomposition, is given by $\mathbf{A} = \mathbf{L}\mathbf{D}\mathbf{L}^T$, where $\mathbf{A}$ is a $n \times n$ positive-definite matrix, $\mathbf{L}$ is a unit lower triangular matrix with unit elements on the diagonal, and $\mathbf{D}$ is a diagonal matrix. Thus, solving $\mathbf{A}\mathbf{x} = \mathbf{L}\mathbf{D}\mathbf{L}^T\mathbf{x} = \mathbf{b}$ is equivalent to solving three equations $\mathbf{L}\mathbf{z} = \mathbf{b}$,$\mathbf{D}\mathbf{r} = \mathbf{z}$ and $\mathbf{L}^T\mathbf{x} = \mathbf{r}$ in sequence:

$$\begin{cases} z_1 = b_1 \\ \vdots \\ z_j = b_j - \sum_{i=1}^{j-1} L_{j,i}z_i \quad j = 2 \sim n \end{cases} \tag{6a}$$

$$r_j = z_j/d_j \quad j = 1 \sim n \tag{6b}$$

$$\begin{cases} x_n = r_n \\ \vdots \\ x_j = r_j - \sum_{i=2}^{j+1} L_{i,j}x_i \quad j = n \sim 1 \end{cases} \tag{6c}$$

where the first two steps are forward substitution, and the third is backward substitution. Different from the standard Cholesky decomposition, the above computation avoids square root operation and division dependency at the cost of more iterations, which can be compensated by parallel processing.

IV. HARDWARE ARCHITECTURE OF BAX

Application-specific hardware accelerators usually adopt parallel processing, stream processing, and pipeline technique to boost the run-time performance. From a macroscopic view, the tasks of BA (see Table 2) have to be executed in serial due to the significant data dependency. Besides, the complex dataflow of conditional execution prevents stream processing. Although pipeline can improve throughput, it brings additional data memory between stages for non-stream processing, which is less appropriate for BA, since it has massive intermediate data. Therefore, we consider executing BA by the algorithm flow under the control of finite state machines (FSM).

The overall architecture of BAX is partitioned into the frontend and the backend (see Fig.2). At the frontend, a set of FSMs are switched by the central controller following the BA algorithm. The FSMs include 1) Constructor that builds the normalization equation by computing the Jacobian and Hessian matrices; 2) Marginalizer that reduces the scale of normalization equation through the Schur elimination method; 3) Solver that figures out the increment of camera states and 3D points coordinates by variant Cholesky decomposition and 4) Updater that calculates the value of optimized camera states and 3D points coordinates. The above steps are repeated until the termination conditions are satisfied. Due to the serial nature, all the FSMs at the frontend can share the computational resources of the backend without conflict.

Unlike the work in [19], which calls the shared computational units by the FSMs directly, BAX bridges the frontend and backend through an instruction queue (IQ). Specifically, the FSMs at the frontend generate 32-bit RISC instructions and write them into the IQ. Then the backend act as a GPP to execute these instructions on a scalar processing unit (SPU) and a vector processing unit (VPU), with no idea of the current step at the frontend. This loosely coupled architecture has two advantages. First, more advanced operations can be integrated to BAX conveniently by the simple modification of the frontend if necessary. Second, the instruction-based calling of the backend avoids the duplicated control logics in each FSM at the frontend. The SPU and VPU can access the data memory (DM) through separate register files. The DM includes a graph memory for the local map, an equation memory for the Jacobian and Hessian matrices, and a shared memory for

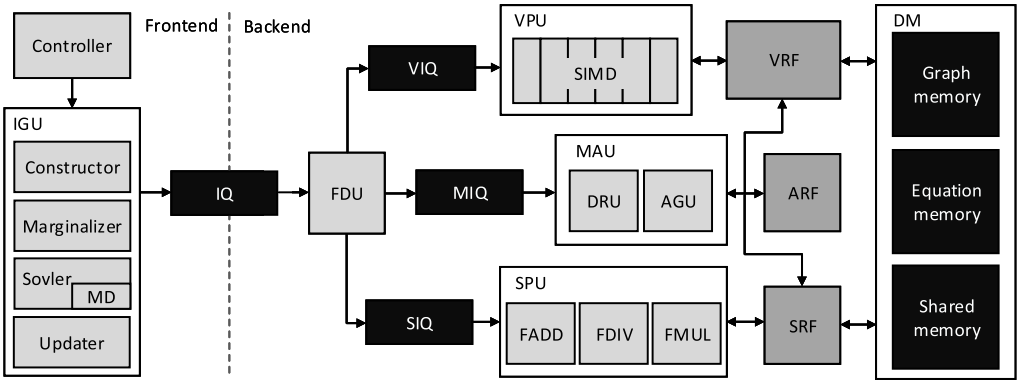


FIGURE 2. Overall architecture of BAX. (IGU: instruction generation unit; MD: matrix decomposition; IQ: instruction queue; FDU: fetch and decode unit; VIQ/MIQ/SIQ: vector/memory/scalar instruction queue; SPU: scalar processing unit; MAU: memory access unit; VPU: vector processing unit; DRU: data reorganization unit; AGU: address generation unit; VRF/ARF/SRF: vector/address/scalar register file; DM: data memory).

the other data. The DM capacity is compressed by making use of the sparsity and symmetry of the Hessian matrix. Besides, BAX implements a single-precision float-point data path with the data normalization method in [10], which saves much hardware and power cost at minimum accuracy loss compared to the double-precision.

A. DECOUPLED ACCESS/EXECUTE ARCHITECTURE

High-performance GPPs are usually enhanced by the SIMD technique to exploit the data-level parallelism (DLP). However, the SIMD execution units still suffer from the latency of data preparation (cache missing, permutation, packed and unpacked, etc.). To address this issue, modern GPUs further extend SIMD to single instruction multiple threads (SIMT). The SIMT technique can feed the pipeline with the other threads when one thread is data-hungry. Instead of the complex SIMT implementation, we expect a simple and efficient method to reduce the data preparation latency in SIMD. Decoupled access/execute (DAE) architecture was proposed in [20]. A typical decoupled processor has two independent processors [22]. They can execute memory reference and computation instructions asynchronously. Therefore, memory latency can be hidden by fetching data ahead of demand. The purpose of DAE architecture was to address the issue of memory access latency [23] at first. The DAE architecture fails in the market of GPP because it is inefficient for irregular data structures and control flows. However, computations on structured data like the massive matrix operations in BA is conformable to the concept of DAE. Although BAX has no external memory access, the data preparation in SIMD execution can be considered as internal memory latency. Therefore, BAX can benefit from in-order execution DAE architecture by prefetching vectors or matrices.

The instruction set architecture (ISA) of BAX derives from the RISC-V ISA [24], where only the load/store and arithmetic instructions (float-point scalar and vector) are retained for simplicity. Besides, a few customed instructions for matrix manipulations and data reorganizations are defined based on the features of BA (see Section III). As shown in Fig.2, the backend of BAX is decoupled into four components: the fetch and decode unit (FDU), the memory access unit (MAU), the scalar processing unit (SPU), and the vector processing unit (VPU). The FDU fetches and decodes instructions from the IQ. Then it issues them to the memory instruction queue (MIQ), the scalar instruction queue (SIQ), and the vector instruction queue (VIQ) according to the type of instructions. The instructions, classified into memory access (load, store, and permute), scalar arithmetics, and vector arithmetics, are handled by the MAU, SPU, and VPU, respectively. The MAU calculates the reference address through the address generation unit (AGU) with the address register file (ARF) and moves data between the data memory (DM), the scalar register file (SRF), and the vector register file (VRF). It can also execute data permutation instructions by the data reorganization unit (DRU). Both VPU and SPU have a 4-stage pipeline (no memory access) with various latencies in the execution stage for different float-point operations. Inside VRF and SRF, a state table for each entry is used to resolve data hazards between the MAU, SPU and VPU.

The pipeline timing of a code snippet executed on the DAE architecture is compared with traditional in-order RISC architecture in Fig.3. Assume that loading (storing) a vector from (to) memory costs three cycles, multiplying two vectors by elements requires four cycles, and both architecture support data forwarding before the write-back stage. In the beginning, the DAE architecture has more latency to complete the store instruction due to the additional FDU pipeline. However, the DAE architecture makes an effect as the loops proceed. The SPU can run the address increment instructions ahead of the completion of the previous store instruction conducted by the MAU since they are decoupled and have no data dependency. Therefore, the DAE architecture will improve the performance significantly, especially for programs with large loop counts and high latency of both memory access and computation. Different from out-of-order superscalar processors, the proposed DAE architecture exploits prefetching rather than dynamic scheduling.

FIGURE 3. Pipeline timing of traditional RISC architecture and DAE architecture. (pipeline stages of FDU are in lower-case).

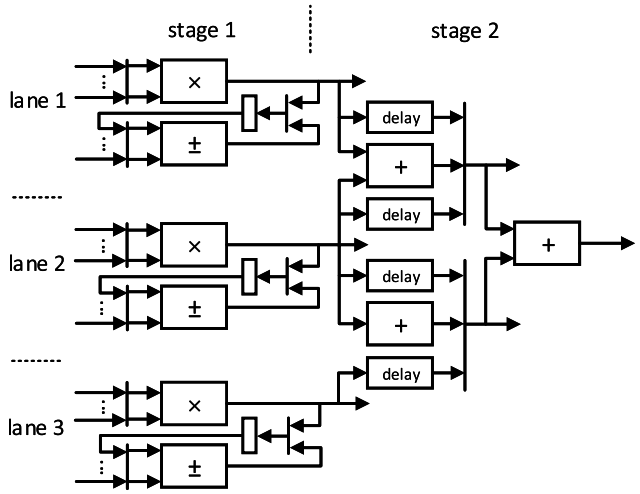


FIGURE 4. Architecture of two-stage VPU.

Thus, it still maintains in-order execution and a simple assembly programming model at a little hardware cost.

B. TRANSPOSE-FREE MATRIX MULTIPLICATION

To save the execution time of completing a matrix operation, we define a set of matrix arithmetic instructions and execute them on a canonical vector processing unit (VPU) with an adder tree. As shown in Fig.4, the VPU architecture is similar to previous matrix processors [18], [19]. The first stage can multiply or add the elements from two vectors as a naive vector processor. In each lane, the multiplier is chained to the adder to perform multiplication and accumulation. Together with the adder tree at the second stage, the VPU can also dot product two vectors and multiply two matrices. As described in Section III, when solving the linear system, the coefficients matrix (Hessian matrix) of the equation is manipulated by a series of block matrices. Since the largest size of a block is 6×6, the VPU of BAX is implemented with six lanes as a trade-off between performance and utilization. With the data normalization method in [10], a 32-bit single-precision float-point datapath guarantees the BA accuracy.

It frequently occurs that a matrix is transposed and then multiplied to another when constructing the Hessian matrix in BA. Although previous works like [13], [14] also perform matrix multiplication using multipliers and adder trees, they are less optimal for matrix transpose in hardware cost and performance. Except for fetching data, the memory access unit (MAU) at the backend of BAX also runs the data permutation instructions using the data reorganization unit (DRU). Therefore, the load and transpose operations can be packed into a single instruction. Using the DAE architecture, the latency of matrix transpose is likely to be hidden.

Another case is that the matrix to be transposed is already in the register file. To address this issue, we regulate three computing patterns for transpose-free matrix multiplication on the VPU, as shown in Fig.5. Assume two 3×3 matrices $\mathbf{A}$ and $\mathbf{B}$ are stored in vector registers with two read ports in row-wise. Therefore, a row vector or a single element from two matrices can be accessed at one cycle. To calculate $\mathbf{C} = \mathbf{AB}$, $\mathbf{A}_{1,x}$ $(x = 1, 2, 3)$ is multiplied with three rows of $\mathbf{B}$ respectively to produce the partial sum of the first row of $\mathbf{C}$. The partial sums are stored in the temp registers and accumulated to the first row of $\mathbf{C}$ (Fig.5a). Repeat this step for all the rows of $\mathbf{A}$ until the calculation of $\mathbf{C}$ completes. Similarly, $\mathbf{C} = \mathbf{A}^T\mathbf{B}$ can be calculated by accessing the element of $\mathbf{A}$ in column-wise (Fig.5b). Since a row of $\mathbf{A}$ and a column of $\mathbf{B}^T$ (i.e., a row of $\mathbf{B}$) can be accessed at the same cycle, each element of $\mathbf{C}$ can be calculated directly by the VPU (Fig.5c). Swapping the order to access $\mathbf{A}$ and $\mathbf{B}$ in Fig.5b, $\mathbf{C} = \mathbf{A}^T\mathbf{B}^T$ can be calculated by column. The result of $\mathbf{C}$ must be transposed before being written to the memory to unify the storage format. The above computing patterns have a time complexity of $O(n^2)$ with n being the matrix size. If the matrix size is larger than the number of VPU lanes, the VPU can compute the partial sums of several partitioned blocks, and then accumulate them to the final result.

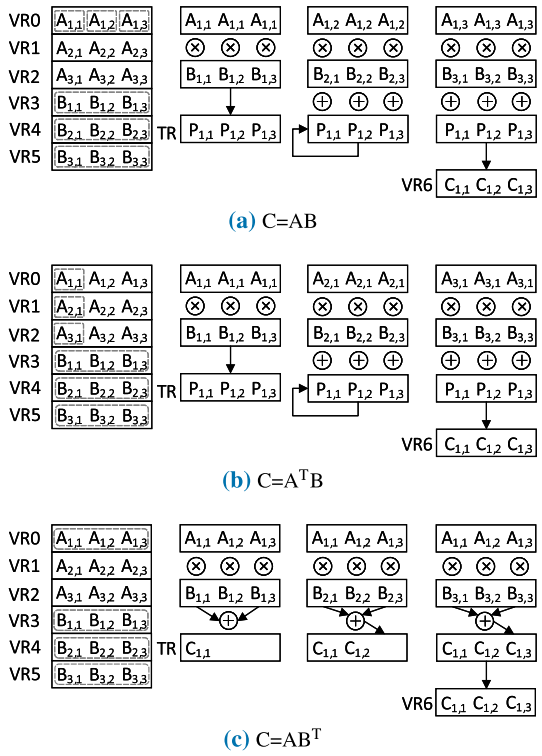


FIGURE 5. Transpose-free matrix multiplication. (VR: vector register; TR: temp register).

C. UNIFIED ARCHITECTURE OF LINEAR SOLVER

Although there is a VPU in BAX, instruction-based vector (matrix) operations are less efficient for the above calculations because the size of the coefficient matrix in Eq.4 is far beyond the number of the SIMD lanes in the VPU and the VRF capacity. If the matrix decomposition is conducted by the 6-width VPU, massive partial sums have to be moved in and out of the registers repeatedly. Despite the DAE architecture, a large amount of pipeline stall is still inevitable. Therefore, the Solver FSM at the frontend is allowed to use the functional units of the VPU and SPU, and access the data memory directly without instructions.

To solve the reduced normalization equation (Eq.4), we learn from [25] and apply the LDL decomposition on the coefficient matrix iteratively. The LDL decomposition of a symmetric matrix $\mathbf{A} = \mathbf{L}\mathbf{D}\mathbf{L}^T$ can be divided into four blocks as follow:

$$\left[\begin{array}{c|c} \mathbf{A}_{n-1} & \mathbf{t} \\ \hline \mathbf{t}^T & g \end{array}\right] = \left[\begin{array}{c|c} \mathbf{L}_{n-1} & \mathbf{0} \\ \hline \mathbf{w}^T & 1 \end{array}\right] \left[\begin{array}{c|c} \mathbf{D}_{n-1} & \mathbf{0} \\ \hline \mathbf{0} & d_n \end{array}\right] \left[\begin{array}{c|c} \mathbf{L}_{n-1}^T & \mathbf{w} \\ \hline \mathbf{0} & 1 \end{array}\right] \tag{7}$$

where $g = a_{nn}$ and d_n are scalars, $\mathbf{t}$ and $\mathbf{w}$ are vectors. By matching the block matrices of the two sides in Eq.7, it is easy to find:

$$\mathbf{A}_{n-1} = \mathbf{L}_{n-1}\mathbf{D}_{n-1}\mathbf{L}_{n-1}^T \tag{8a}$$

$$\mathbf{t} = \mathbf{L}_{n-1}\mathbf{D}_{n-1}\mathbf{w} \tag{8b}$$

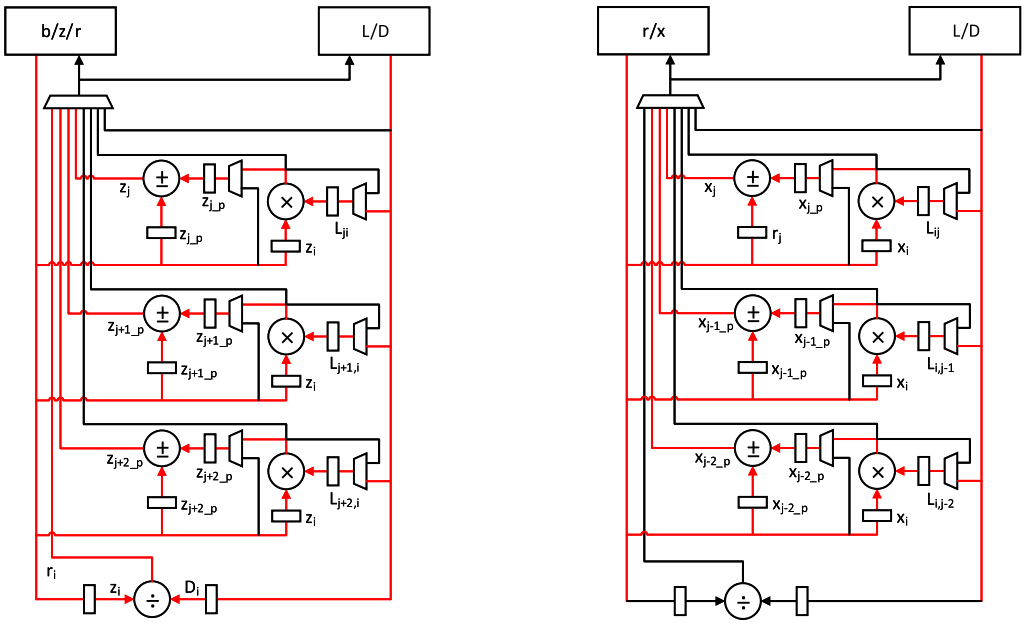
$$g = \mathbf{w}^T\mathbf{D}_{n-1}\mathbf{w} + d_n \tag{8c}$$

which indicates that given the LDL decomposition of its top-left $n-1$ order cofactor $\mathbf{A}_{n-1}$ as Eq.8a, $\mathbf{A}$ can be decomposed by solving $\mathbf{w}$ and d_n according to Eq.8b and Eq.8c. Therefore, the full decomposition can be iterated from $\mathbf{A}_1 = \mathbf{D}_1 = a_{11}$ and $\mathbf{L}_1 = 1$.

Fortunately, solving Eq.8b exactly follows the forward substitution of LDL composition as Eq.6a and Eq.6b. It indicates that the iterative LDL decomposition can be performed on a common datapath using similar dataflow, saving the hardware overhead. Therefore, we design a unified architecture for both LDL decomposition (forward substitution) and equation solving (backward substitution). Fig.6a and Fig.6b shows the dataflow of the forward and backward substitution to calculate Eq.6 on the unified linear solver, respectively. In the forward substitution, b_j is first assigned to z_j one-to-one ($j = 1 \sim n$). Then the multiple product terms $z_{j\_p} = L_{j,i}z_i$ (partial sums) associated with the same z_i are calculated in parallel and negatively accumulated to z_j until z_j is figured out ($j = 2 \sim n$, $i = 2 \sim n-1$). Once z_j is ready, $r_j = z_j/d_j$ is calculated immediately on the divider. After that, the backward substitution starts in the reverse order to calculate the final results x_j. Besides, the dataflow to solve Eq.8c is similar to the matrix multiplication on the VPU. All the intermediate data are stored in the on-chip data memory. One triangular iteration above has a timing complexity of about $O(n^2/(2p))$, where p is the parallelism degree. Thus, the total time cost for a complete decomposition after n iterations is about $O((n^3)/(6p))$. The linear solver borrows six adders and one divider from the VPU and SPU, respectively. With the elaborate pipeline timing, it can solve a 96×96 linear system of equation in about 80 thousand cycles. This unified architecture is flexible to support large scale matrix decomposition by more iterations or adding more computing resources.

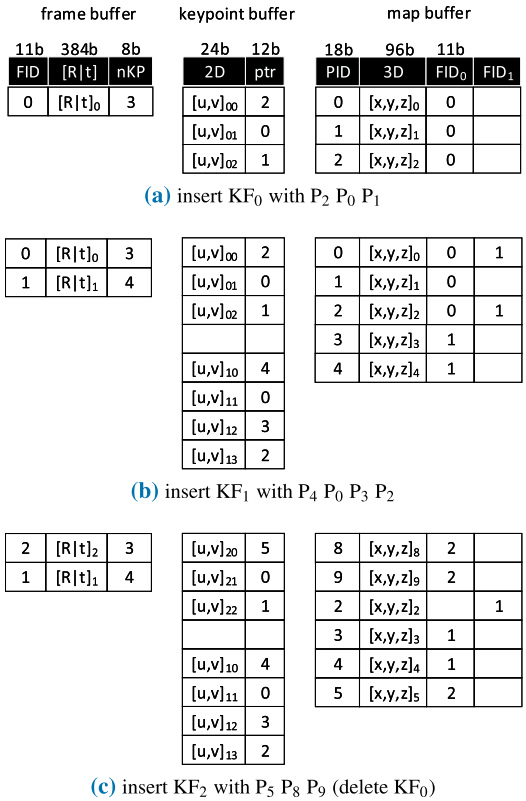
D. LOCAL MAP AND GRAPH MEMORY HIERARCHY

The camera pose of each frame, the map points observed per frame, and the keypoints (features) extracted in each frame are associated with each other to form a local map as a graph. When a new frame (pose) with map points and keypoints is inserted to the map, the earliest frame has to be deleted if beyond the memory capacity. In [19], a two-stage graph memory is proposed without the storage of 2D keypoints coordinates, which is not suitable for BAX. In [18], a hierarchical graph memory with a free-list FIFO is used to store a similar map, but the method to update the map is not explained in detail. In this work, we implement a graph memory including three buffers for frames, keypoints and map points, respectively. Each entry of the three buffers is partitioned into multiple fields with different attributes. The frame buffer has three fields for the frame index (FID), the camera pose ([R|t]), and the number of keypoints in this frame (nKP). The LSB of FID indicates whether the entry is used or not. The keypoint buffer is divided into multiple groups, and a group stores the 2D coordinates (2D) of keypoints in a frame. Each keypoint has a pointer (PTR) to



(a) forward: solve z/r **(b)** backward: solve x

FIGURE 6. Architecture and dataflow of linear solver. (red arrows are active datapath).



(a) insert KF_0 with P_2 P_0 P_1

(b) insert KF_1 with P_4 P_0 P_3 P_2

(c) insert KF_2 with P_5 P_8 P_9 (delete KF_0)

FIGURE 7. Map update in graph memory.

its corresponding 3D map point. The map buffer stores the indexes of map points (PID), their 3D coordinates (3D), and the index of frames (FID_i) that observe the map point.

We will explain how the graph memory handles the local map with an example in Fig.7. Assume that the frame buffer has only two entries, the map buffer has six entries, and the keypoint buffer has eight entries partitioned into two groups. KF_i denotes the i-th keyframe, and its camera pose is expressed by $[R|t]_i$. The 2D coordinates of the j-th keypoint in KF_i are denoted by $[u,v]_{ij}$. The m-th landmark (3D point) is denoted by P_m, where the index is numbered independently from any frame, and its 3D coordinates are given by $[x,y,z]_m$. In the beginning, KF_0 with three keypoints arrives. The first entry of the frame buffer is filled, and the three keypoints are stored in the first group of the keypoint buffer in order. These keypoints correspond to three landmarks P_2, P_0, and P_1, respectively. The landmarks are stored in the map buffer in sequence. The write address is given by m mod d, where $d = 8$ is the map buffer depth, and this address is also the pointer PTR of the associated keypoint. Then KF_1 is inserted after KF_0, and the 2D keypoints are stored in the second group of the keypoint buffer. Since P_0 and P_2 are co-observed in both KF_0 and KF_1, they share the same entries in the map buffer so long as the related FID fields are set. The next frame KF_2 will replace KF_0, but P_2 has to be retained because it is still observed in KF_1.

When running BA, the accelerator first reads a camera pose and its nKP from the frame buffer. Then the 2D coordinates of all the keypoints in this frame are fetched from the keypoint buffer according to nKP. Meanwhile, the corresponding 3D coordinates and the observed FIDs are indexed by PTR. After that, the normalization equation to solve BA can be constructed with the above information. In our implementation, the 128KB graph memory can maintain a local map with 16 frames, 256 keypoints per frame, and 4096 landmarks with eight co-observations at most. Any fine-coarse

TABLE 3. FPGA resource utilization of BAX.

Module	LUT	REG	DSP	BRAM[a]
Controller	1690	433	0	0
IGU	2458	503	0	0
IQ&V/M/SIQ	16	12	0	2
DM	48	27	0	90
V/A/SRF	4783	1394	0	0
FDU	573	392	0	0
VPU	4819	2623	37	0
MAU	847	1722	1	0
SPU	2015	1687	6	0
Total	17249	8793	44	92
(%)	(6.3)	(1.6)	(1.7)	(10.0)

[a]number of 36kb BRAM

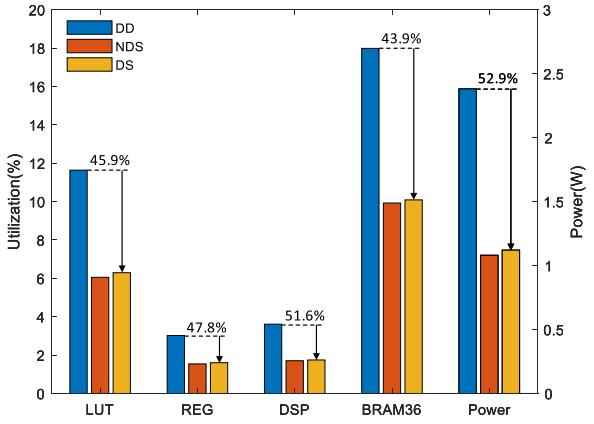


FIGURE 8. Resource and power comparison of different implementations.

operation to an entry of the buffers is completed in one or two cycles.

V. EXPERIMENT RESULTS

A. IMPLEMENTATION

The proposed accelerator is implemented on the FPGA side (XCZU9EG) of a Xilinx Zynq Ultrascale+ MPSoC (ZCU102) at 200MHz. Except for a 32-bit integer adder in the address generation unit (AGU) of the MAU, all the computational units in the SPU and VPU are 32-bit float-point. These float-point units work in non-block mode with the pipeline depth from four to twelve cycles. Table 3 reports the resource utilization of LUTs, REGs, DSPs, and BRAMs by different modules. Since the divider is generated without DSP by Vivado, it contributes to the most LUTs and REGs in the SPU. In addition, a BRAM in the FPGA is at least 18Kb, which exceeds the actual requirement of the small instruction queues. Nevertheless, the accelerator still consumes very few FPGA resources. To evaluate the effect of DAE architecture with single-precision datapath (DS) on hardware cost, we also implement two other versions of BAX, including non-decouple single-precision (NDS) and decouple double-precision (DD). As shown in Fig.8, the single-precision datapath saves about half of hardware resources and power. Compared with the non-decouple architecture, the proposed DAE architecture has only a few more hardware cost. It is mainly used for the FDU and MAU, and the multi-ported VRF and SRF.

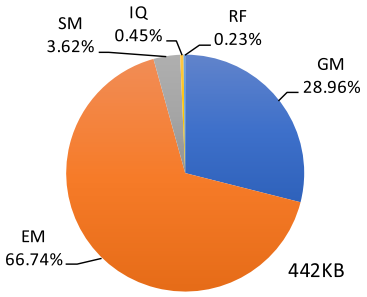


FIGURE 9. On-chip memory utilization breakdown. (GM: graph memory; EM: equation memory; SM: shared memory; IQ: instruction queues; RF: register files).

We also analyze the size of necessary on-chip memory by a quantitative approach. Different from the work in [10], BAX constructs the Hessian matrix using the 2×9 Jacobian associated with each error term without the storage of the global Jacobian in Eq.1. In our implementation, BAX supports a local map with at most 16 frames (camera poses) and 4096 landmarks (3D points) in total. According to Eq.2, we can infer that $\mathbf{B}$ and $\mathbf{C}$ consists of 16 6×6 and 4096 3×3 symmetric block matrices on the diagonal with the rest being all zeros, and $\mathbf{g}$ has a length of 12384. Since a landmark has max eight co-observations at different poses and a frame has most 256 keypoints, the sub-matrix $\mathbf{E}$ contains 2048 6×3 non-zero blocks, which can be encoded using the compressed sparse row (CSR) format by block. By exploiting the sparsity and symmetry above, the Hessian matrix $\mathbf{H}$ with the vector $\mathbf{g}$ can be stored in the 295KB equation memory of the data memory (DM), including a 4.75KB space to index the CSR formant. Besides, the DM also contains 128KB graph memory (see Section IV.D) for the local map and 16KB shared memory for intermediate data. In addition, BAX has four 128×32-bit instruction queues (IQ/VIQ/MIQ/SIQ), 16×384-bit VRF, 16×64-bit SRF and 8×32-bit ARF. As a result, the total on-chip memory required is about 442KB, and the breakdown of on-chip memory usage is shown in Fig.9.

B. PERFORMANCE

To evaluate the performance gain from the four techniques described in Section IV, we run RTL simulation of BA on different architectures. The baseline architecture is a canonical 5-stage RISC pipeline. It performs all the computations on an SPU and VPU with the same specs as BAX, and the local map is stored in a single-bank memory. Then one or more techniques are added to the baseline architecture, and BAX is the one with all techniques. Before the simulation, the test datasets from the BAL project [8] are pre-processed. One group from each of the five datasets is selected and trimmed to 16 frames, less than 256 keypoints per frame, and less than eight co-observations per landmark. All the data are normalized to single-precision, and the termination conditions are set. The time breakdown of different architecture

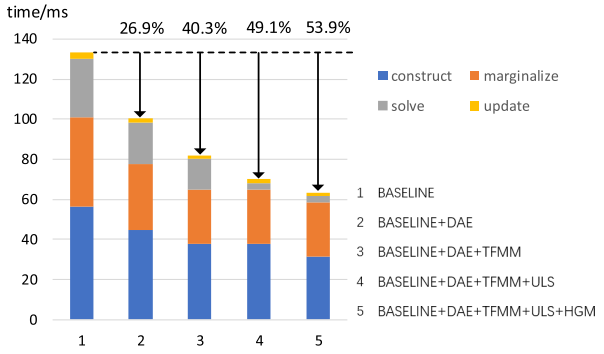
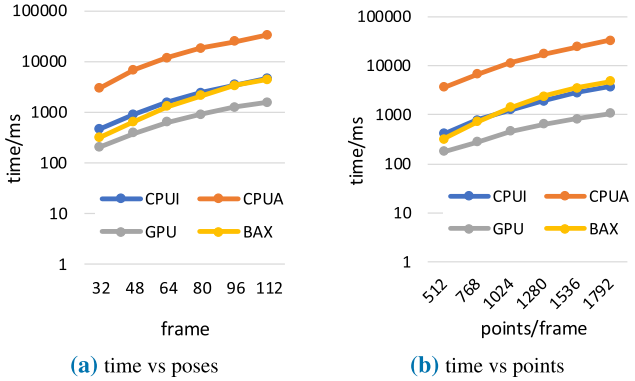


FIGURE 10. Processing time breakdown of different architectures. (DAE: decoupled access/execute; TFMM: transpose-free matrix multiplication; ULS: unified linear solver; HGM: hierarchical graph memory).



(a) time vs poses (b) time vs points

FIGURE 11. Processing time comparison of BAX and GPPs with DRAM traffic.

TABLE 4. Performance comparison of BAX and GPPs.

Dataset	CPU_I(ms)	CPU_A(ms)	GPU(ms)	BAX(ms)
Ladybug	106.26	1356	57.14	62.47
Trafalgr	100.05	1259	54.59	60.23
Dubrovnic	96.08	1226	54.93	58.16
Venice	126.72	1672	60.22	69.14
Final	123.31	1591	58.78	67.21
Average	110.48	1421	57.13	63.44
Speedup	1.73	22.38	0.90	-
Power(W)	65	1.67	130	1.12

is shown in Fig.10. It is observed that: 1) The performance of all steps in BA is improved by the DAE architecture, which hides the memory access latency; 2) The transpose-free matrix multiplication method further saves the processing time of matrix multiplication in all steps; 3) The unified linear solver accelerates the matrix decomposition when solving the equation; 4) The hierarchical graph memory reduces the processing time slightly due to the overlapped accessing of multiple parameters. With the combination of above features, the performance of BAX is improved by 53.9% than that of the baseline architecture.

The performance of BAX is also compared with that of different GPPs. The BA algorithm is programmed using the g2o [26] and OpenOF [27] library for CPU and GPU, respectively, with the same specifications as BAX. The BA program runs on an Intel i7-8700 CPU at 3.2GHz with 16GB DRAM, an ARM A53 CPU at 1.2GHz with 4GB DRAM, and an NVIDIA GTX960 GPU at 1.1GHz with 3GB graph DRAM. The experiment results are listed in Table 4. The execution time of the above platforms is 110.48ms, 1.42s and 57.13ms, respectively, and the power consumption is 65W, 1.67W and 130W. Although the GPU implementation achieves the best performance, it is not suitable for embedded VO due to its high power. On the contrary, BAX can complete a full BA in 63.44ms on average while consumes only 1.12W power, which is much less than that of the GPU and the Intel CPU. It achieves a 1.73× and 22.38× speedup compared with the Intel and ARM CPU, respectively. Besides, BAX has no external memory access during the entire processing. To evaluate the effect of DRAM traffic on performance, we add a DDR controller and a DMA (IP cores in Vivado) to BAX. The DRAM interface is set to 16bit and 1200MHz by default. Then we run simulations on BAX and the GPPs with larger datasets, respectively. As shown in Fig.11, the processing time of BAX and the GPPs grows nonlinearly as the number of poses or points per frame increases (Y-axis is in log). The curve slope of BAX is larger than that of GPPs for the limited on-chip memory and lower DDR bandwidth. Besides, the timing cost of GPU grows relatively slowly because of the massive streaming cores and high-parallel computing patterns. Therefore, one advantage of BAX is to solve a small BA problem for embedded VO efficiently.

Previous BA-related hardware accelerators [15], [18], [19] target to different applications and have various design spaces. The specifications and features of these works and BAX are listed in Table 5. The works [18] and [19] are visual (inertial) odometry accelerators implemented in ASIC while [15] and BAX are FPGA-based BA accelerators. The work [19] completes one iteration in 30.8ms using the factor graph method. It adopts double-precision float-point data to maintain high accuracy and achieves extremely low power through the adaptation technique. The work [18] adopts 32-bit fixed-point numerical precision with data normalization. It solves a pose-only BA with 20 frames and requires much fewer operations compared with the others. According to the architecture and the breakdown of chip die, we estimate that it competes one iteration in less than 17us, and the BA part costs less than 30mW. However, both works in [18], [19] optimize only the camera poses, and the errors caused by the map points are not reduced. In [15], only the Schur elimination step is implemented on FPGA, while the others still run on CPU. The data movement between the off-chip and on-chip memory results in additional latency and power cost. It completes one iteration on larger datasets in 110ms at the expense of more hardware resources. To the best of our knowledge, BAX is the first full-BA accelerator for embedded VO applications. It completes one iteration in about

TABLE 5. Specification of BAX and related works.

Work	Algorithm	Implementation	Configuration	Performance
[19][a]	factor graph	ASIC, 65nm, 410kB SRAM	20 frames, 4000 keypoints, 10 co-observations, 64b float-point	30.8ms/iteration, 9.6mW, @83.3MHz, 1V
[18][b]	pose-only BA	ASIC, 28nm, 336kB SRAM	20 frames, 4096 keypoints 8 co-observations, 32b fixed-point	<17us/iteration, <30mW, @240MHz, 0.9V
[15][c]	full BA	FPGA XC7Z030, ARM A9, 24534 LUTs, 229 BRAM36s, 164 DSPs, 28877 REGs, DRAM	50 frames, over 80000 keypoints 50 co-observations, 32b float-point	110ms/iteration, 2.8W, FPGA@180MHz, ARM@667MHz
BAX	full BA pose-only BA	FPGA XCZU9EG, 17249 LUTs, 8793 REGs, 92 BRAM36s, 44 DSPs	16 frames, 4096 keypoints, 8 co-observations, 32b float-point	10.57ms/iteration@1.12W@full BA 13us/iteration@479mW@pose-only BA FPGA@200MHz

[a]Only the BA-related part is counted; [b]Only the BA-related part is estimated; [c]Schur elimination is performed on FPGA, the other steps are on ARM.

TABLE 6. MSE of poses/points optimized by BAX.

Dataset	Double	Single w DN(full)	Sincgle w DN(pose-only)	Single w/o DN	w/o BA
Ladybug	2.11e-27/3.95e-7	7.25e-25/9.4e-6	4.16e-16/1.04	1.94e-16/3.29e-1	2.97e-13/1.02
Trafalgr	3.18e-27/9.41e-8	3.85e-25/7.88e-5	6.63e-16/1.07	2.24e-17/8.33e-1	3.04e-13/1.10
Dubrovnic	9.8e-28/6.54e-7	9.6e-26/8.16e-5	8.86e-17/1.12	6.34e-17/3.19e-1	3.11e-13/1.07
Venice	4.85e-27/7.72e-7	3.31e-25/2.68e-5	5.84e-16/1.09	3.32e-17/9.4e-2	3.01e-13/1.11
Final	1.63e-26/5.59e-7	2.04e-24/5.41e-5	2.32e-16/1.05	8.57e-17/4.15e-1	2.98e-13/1.09
Average	5.49e-27/4.95e-7	7.11e-25/5.01e-5	3.97e-16/1.07	7.97e-17/3.98e-1	3.02e-13/1.08
Accuracy gain[a]	5.55e13/2.18e6	4.25e11/2.16e4	7.61e2/1.01	3.79e3/2.71	

[a] MSE without BA to that of the other configurations.

10.57ms at 200MHz and overcomes the shortcoming, i.e., the lack of refinement on points in [18], [19] to achieve high accuracy. The support for full-BA and the FPGA-based implementation brings BAX more power cost than that of [18], [19]. However, at the configuration of pose-only mode, the power cost and processing speed of BAX can also be reduced. Compared to [15], BAX solves small BA problems more efficiently with full hardware implementation. As a result, it avoids the latency of external memory access and costs less hardware and power than that of [15].

C. ACCURACY

In this work, a 32-bit floating-point datapath with data normalization is used to save hardware resources. To evaluate its effect on the accuracy, we use the same datasets above to simulate on three configurations of BAX, including double-precision, single-precision with data normalization (DN), and single-precision without DN. The BA results calculated by the double-precision datapath are taken as the ground truth. Then BAX with different numerical precision performs BA using the same datasets applied by Gaussian noise. The results include camera poses and coordinates of 3D map points, among which the camera poses are converted to 6-element vectors using the Rodriguez formula [28]. After that, the accuracy is measured by the mean square errors (MSE) of all the three sets of results relative to the ground truth, as shown in Table 6. For single-precision without DN, the MSE of camera poses and map points are $10^{10}\times$ and $10^{6}\times$ higher than that of double-precision, respectively. Although it seems that the MSE is quite small for one BA, this error is still unacceptable because it will accumulate as the movement proceeds. However, the MSE of single-precision with DN is only increased by $100\times$ for both poses and points compared to the double-precision. Besides, the result of pose-only BA is a little worse than that of single-precision without DN, indicating the effectiveness of full BA. From the accuracy gain, it is observed that the DN method improves the accuracy of poses and points in single-precision BA by $10^{7}\times$ and $10^{4}\times$.

VI. CONCLUSION

We propose BAX, an FPGA accelerator for feature-based BA that refines both camera poses and map points. It consists of a frontend and a backend, which are loosely coupled by an instruction queue. Instructions are generated by the FSMs at the frontend. Then BA is performed at the backend under the control of the instructions. The backend exploits a DAE architecture to reduce the latency of data preparation in continuous matrix operations. A 6-width VPU with transpose-free matrix multiplication is used to reduce this latency further. In the linear solver, a unified architecture for both forward and backward substitution is proposed. BAX maintains a local map, including 16 camera poses, 256 keypoints per frame, and eight co-observations per landmark, in a hierarchical graph memory.

The accelerator completes a full BA in about 63.44ms at 200MHz while consuming 1.12W power without external memory access. It achieves a $1.73\times$ and $22.38\times$ speedup

compared with the desktop CPU and the embedded CPU, respectively, and 90% performance of the GPU. Since the backend works as a standalone processing unit, the function of BAX can be easily modified by configuring the FSMs at the frontend. Although this work targets embedded VO, the architecture of BAX also supports large scale BA or global BA if more computational resources and external memory are used. The concepts, such as the DAE architecture and transpose-free matrix multiplication, are suitable for other applications that contain massive vector and matrix operations.

REFERENCES

[1] B. Triggs, P. F. Mclauchlan, R. I. Hartley, and A. W. Fitzgibbon, "Bundle adjustment—A modern synthesis," in *Vision Algorithms: Theory and Practice* (Lecture Notes in Computer Science). Berlin, Germany: Springer, 2000, pp. 298–372.

[2] H. Strasdat, J. M. M. Montiel, and A. J. Davison, "Real-time monocular SLAM: Why filter?" in *Proc. IEEE Int. Conf. Robot. Automat.*, May 2010, pp. 2657–2664.

[3] G. Huang, "Visual-inertial navigation: A concise review," in *Proc. Int. Conf. Robot. Automat. (ICRA)*, May 2019, pp. 9572–9582.

[4] D. Scaramuzza and F. Fraundorfer, "Visual odometry [tutorial]," *IEEE Robot. Autom. Mag.*, vol. 18, no. 4, pp. 80–92, Dec. 2011.

[5] F. Fraundorfer and D. Scaramuzza, "Visual odometry: Part II: Matching, robustness, optimization, and applications," *IEEE Robot. Autom. Mag.*, vol. 19, no. 2, pp. 78–90, Jun. 2012.

[6] J. L. Schonberger and J.-M. Frahm, "Structure-from-motion revisited," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 4104–4113.

[7] M. I. A. Lourakis and A. A. Argyros, "SBA: A software package for generic sparse bundle adjustment," *ACM Trans. Math. Softw.*, vol. 36, no. 1, pp. 1–30, Mar. 2009.

[8] S. Agarwal, N. Snavely, S. M. Seitz, and R. Szeliski, "Bundle adjustment in the large," in *Proc. ECCV*, 2010, pp. 29–42.

[9] S. Choudhary, S. Gupta, and P. J. Narayanan, "Practical time bundle adjustment for 3D reconstruction on the GPU," in *Proc. ECCV*, 2010, pp. 423–435.

[10] C. Wu, S. Agarwal, B. Curless, and S. M. Seitz, "Multicore bundle adjustment," in *Proc. CVPR*, Jun. 2011, pp. 3057–3064.

[11] M. Zheng, S. Zhou, X. Xiong, and J. Zhu, "A new GPU bundle adjustment method for large-scale data," *Photogramm. Eng. Remote Sens.*, vol. 83, no. 9, pp. 633–641, Sep. 2017.

[12] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, "ORB: An efficient alternative to SIFT or SURF," in *Proc. Int. Conf. Comput. Vis.*, Nov. 2011, pp. 2564–2571.

[13] S. Chen, "FT-matrix: A coordination-aware architecture for signal processing," *IEEE Micro*, vol. 34, no. 6, pp. 64–73, Nov./Dec. 2014.

[14] M. Woh, S. Seo, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner, "AnySP: Anytime anywhere anyway signal processing," *IEEE Micro*, vol. 30, no. 1, pp. 81–91, Jan. 2010.

[15] S. Qin, Q. Liu, B. Yu, and S. Liu, "π-BA: Bundle adjustment acceleration on embedded FPGAs with co-observation optimization," in *Proc. IEEE 27th Annu. Int. Symp. Field-Program. Custom Comput. Mach.*, Apr./May 2019, pp. 100–108.

[16] J.-S. Yoon, J.-H. Kim, H.-E. Kim, W.-Y. Lee, S.-H. Kim, K. Chung, J.-S. Park, and L.-S. Kim, "A graphics and vision unified processor with 0.89 μW/fps pose estimation engine for augmented reality," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 2010, pp. 336–337.

[17] I. Hong, G. Kim, Y. Kim, D. Kim, B.-G. Nam, and H.-J. Yoo, "A 27 mW reconfigurable marker-less logarithmic camera pose estimation engine for mobile augmented reality processor," *IEEE J. Solid-State Circuits*, vol. 50, no. 11, pp. 2513–2523, Nov. 2015.

[18] Z. Li, Y. Chen, L. Gong, L. Liu, D. Sylvester, D. Blaauw, and H.-S. Kim, "An 879GOPS 243 mW 80 fps VGA fully visual CNN-SLAM processor for wide-range autonomous exploration," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 2019, pp. 134–136.

[19] A. Suleiman, Z. Zhang, L. Carlone, S. Karaman, and V. Sze, "Navion: A 2-mW fully integrated real-time visual-inertial odometry accelerator for autonomous navigation of nano drones," *IEEE J. Solid-State Circuits*, vol. 54, no. 4, pp. 1106–1119, Apr. 2019.

[20] J. E. Smith, "Decoupled access/execute computer architectures," SIGARCH *Comput. Archit. News*, vol. 10, no. 3, pp. 112–119, Apr. 1982.

[21] V. Lepetit, F. Moreno-Noguer, and P. Fua, "EPnP: An accurate O(n) solution to the PnP problem," *Int. J. Comput. Vis.*, vol. 81, no. 2, pp. 155–166, Feb. 2009.

[22] J. R. Goodman, "PIPE: A VLSI decoupled architecture," *SIGARCH Comput. Archit. News*, vol. 13, no. 3, pp. 20–27, Jun. 1985.

[23] L. Kurian, P. T. Hulina, and L. D. Coraor, "Memory latency effects in decoupled architectures," *IEEE Trans. Comput.*, vol. 43, no. 10, pp. 1129–1139, 1994.

[24] *RISC-V Instruction Set Manual*. Accessed: Jun. 2019. [Online]. Available: https://github.com/riscv/riscv-isa-manual

[25] D. Yang, G. D. Peterson, and H. Li, "Compressed sensing and Cholesky decomposition on FPGAs and GPUs," *Parallel Comput.*, vol. 38, no. 8, pp. 421–437, Aug. 2012.

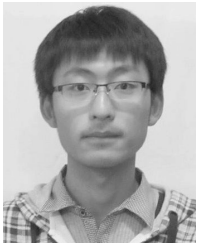
[26] G. Grisetti, R. Kümmerle, H. Strasdat, and K. Konolige, "g^{2}o: A general framework for (hyper) graph optimization," in *Proc. IEEE Int. Conf. Robot. Automat.*, May 2011, pp. 9–13.

[27] C. Wefelscheid and O. Hellwich, "OpenOF: Framework for sparse non-linear least squares optimization on a GPU," in *Proc. Int. Conf. Comput. Vis. Theory Appl.*, 2013, pp. 260–267.

[28] J. E. Mebius, "Derivation of the Euler-Rodrigues formula for three-dimensional rotations from the general formula for four-dimensional rotations," 2017, *arXiv:math/0701759*. [Online]. Available: https://arxiv.org/abs/math/0701759

RONGDI SUN received the B.S. degree in electronic engineering from Xi'an Jiao Tong University, China, in 2013. He is currently pursuing the Ph.D. degree in electronic engineering with Shanghai Jiao Tong University, China. He has rich experience in image processing based on FPGA. His research interests include signal processing, domain-specific architecture, and VLSI design.

PEILIN LIU (Senior Member,IEEE) received the Ph.D. degree in electronic engineering from the University of Tokyo, in 1998. She worked as a Research Fellow with the University of Tokyo, in 1999. From 1999 to 2003, she worked as a Senior Researcher with the Central Research Institute of Fujitsu, Tokyo. Her researches mainly focus on signal processing, low power computing architecture, and application-oriented SoC design and verification. She is currently a Professor with the Department of Electronic Engineering, Shanghai Jiao Tong University, the Director of the Brain-inspired Application Technology Center, and responsible for a series of important projects, such as BD-SoC platform development, low power and high-performance communication DSP. She is also the Chair of Shanghai chapter of IEEE Circuit and System.



JIANWEI XUE received the B.S. degree in radio and television engineering from the University of Nanjing University of Posts and Telecommunications, China, in 2012, and the M.S. degree in telecommunication and information system from the University of Chinese Academy of Sciences, Beijing, China, in 2016. He is currently pursuing the Ph.D. degree in electronic engineering with Shanghai Jiao Tong University, Shanghai, China. His research interests include neuromorphic computing, network on chips, and ultra-low-power computing.

SHIYU YANG received the B.S. degree from the University of Electronic Science and Technology of China, in 2018. He is currently pursuing the M.S. degree with the School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University, China. He is currently working on SoC architecture with the Brain-inspired Application Technology Center. His research interests include spiking neural networks, asynchronous network on chip, and FPGA-based deep neural network accelerator.

JIUCHAO QIAN received the B.S. degree from Shan Dong University, in 2004, the M.S. degree from Shanghai Maritime University, in 2008, and the Ph.D. degree from Shanghai Jiao Tong University, in 2015. He has worked on GNSS receiver technologies with the Shanghai Key Laboratory of Navigation and Location Based Service, Shanghai Jiao Tong University. His current researches focus on signal processing, indoor positioning, computer vision, and affective computing with the Brain-inspired Application Technology Center.

RENDONG YING (Member, IEEE) received the Ph.D. degree in circuits and systems from Shanghai Jiao Tong University, China, in 2007. He is currently an Associate Professor with the School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University. He has an in-depth study of digital signal processing and its hardware implementation techniques. His research interests include navigation signal processing, SoC architecture for high-performance digital signal processing systems, 3D visual signal processing, and machine thinking. He has long-term close cooperation with domestic research institutions and well-known companies. Participating projects include: ''Modeling and Analysis of High-Reliability Software Systems'', ''Application-Oriented Instruction Configurable Processors'', ''Distributed 3D Video Acceleration Engine Technology'', and ''Compressed Sensing-Based Spatial Audio Object Coding''.