Practical Time Bundle Adjustment for 3D Reconstruction on the GPU

Siddharth Choudhary, Shubham Gupta, and P J Narayanan

Center for Visual Information Technology
International Institute of Information Technology
Hyderabad, India
{siddharth_ch@students.,shubham@students.,pjn@}iiit.ac.in

Abstract. Large-scale 3D reconstruction has received a lot of attention from the computer vision community recently. Bundle adjustment is a key component of the reconstruction pipeline. The bundle adjustment step requires a considerable amount of computational resources and is usually the slowest step in the pipeline. This step hasn't been parallelized effectively either. In this paper, we present a hybrid implementation of sparse bundle adjustment on the GPUs using the CUDA programming model, with the CPU working in parallel. The overall algorithm is decomposed into smaller steps. Each of which is scheduled on the GPU or the CPU. We develop efficient kernels most of the steps and exploit existing libraries for data parallel operations and matrix inverse. Our implementation outperforms the CPU implementation significantly, achieving a speedup of 8-10 times over the standard CPU implementation for datasets with upto 500 images on one quarter of an Nvidia Tesla S1070 GPU.

1 Introduction

The internet photo sharing sites have a large collection of unstructured photographs of many tourist sites today. Many new computer vision exploits them, especially for large scale sparse 3D reconstruction using structure from motion (SFM). The SFM pipeline has several steps, with a joint optimization of camera positions and point coordinates using Bundle Adjustment (BA) as the last step. Bundle adjustment is an iterative step, typically performed using the Levenberg-Marquardt (LM) non-linear optimization scheme. Large scale 3D reconstruction requires huge computations with bundle adjustment being the primary bottleneck, consuming half the total computation time. For example, reconstruction of a set of 715 images of Notre Dame data set took around two weeks of running time [1], dominated by iterative bundle adjustment. The current SFM pipeline uses a single core CPU for bundle adjustment, while the other steps of the reconstruction pipeline are performed on a cluster of processors [2]. Effective parallelization of bundle adjustment hasn't been reported as it is not easily parallelizable.

The rapid increase in the performance of graphics hardware have made the GPU a strong candidate for performing many compute intensive tasks. GPUs

are being used for many computer vision applications, such as Graph Cuts [3], tracking [4], and OpenVIDIA [5]. GPUs are increasingly being used as general purpose computing platforms due to their high compute power to cost ratio. No work has been done to implement bundle adjustment on the GPUs or other multicore or manycore architectures. In this paper, we present a hybrid implementation of sparse bundle adjustment with the GPU and the CPU working together. We achieve a speedup of 8-10 times on an Nvidia Tesla S1070 GPU on a dataset of about 500 images. The LM algorithm is decomposed into multiple small steps, each of which is performed using a kernel on the GPU or using a function on the CPU. The concerted work of the CPU and the GPU is critical to the overall performance gain.

Our goal is to develop a practical time implementation of Bundle Adjustment by exploiting all computing resources of the CPU and the GPU. Our implementation efficiently schedules the algorithm steps on CPU and GPU to minimize the computation time. The complexity of the problem grows rapidly with the number of images. However, the visibility aspects of points on cameras can place a natural limit on how many images need to be processed together. The current approach is to identify clusters of images and points that need to be processed together [6]. Large data sets are decomposed into mildly overlapping sets of manageable sizes. An ability to perform bundle adjustment on about 500 images quickly will suffice to process even data sets of arbitrarily large number of images as a result. We focus on exactly this problem in this paper.

The rest of the paper is organized as follows. Section 2 discusses the related work. In Section 3 we discuss the general Bundle Adjustment algorithm followed by our implementation of Sparse Bundle Adjustment on the GPU. In Section 4, we report the experimental results. In Section 5, the conclusion and future work are discussed.

2 Related Work

3D Reconstruction and Bundle Adjustment are active areas today. Brown and Lowe [7] presented 3D Reconstruction of unordered data sets using incremental SFM. Phototourism [1] is a nice application of 3D reconstruction for interactively browsing and exploring large collection of unstructured photographs. The problem of large scale 3D reconstruction has been addressed , which takes advantage of the redundancy available in the large collection of unordered dataset of images and maximizes the parallelization available in the SFM pipeline [2, 6]. Bundle Adjustment was originally conceived in photogrammetry [8], and has been adapted for large scale reconstructions. Ni et al. [9] solve the problem by dividing it into several submaps which can be optimized in parallel. In general, a sparse variant of Levenberg-Marquardt minimization algorithm [10] is the most widely used choice for Bundle Adjustment. A public implementation is available [8]. Byröd and Åström [11] attempt to solve the problem using preconditioned conjugate gradients, utilizing the underlying geometric layout of the problem. Cao et al. [12] try to parallelize the dense LM algorithm, but their method is

not well suited for sparse data. Agarwal et al. [2] design a system to maximize parallelization at each stage in the pipeline, which uses a cluster with 500 computer cores to perform the computations but bundle adjustment inspite of being the primary bottleneck uses a single core since it is not parallelized. There are many previous works available which solve the problem of bundle adjustment, there is very less progress to solve the problem using multicore architectures.

3 Sparse Bundle Adjustment on the GPU

Bundle adjustment is the optimal adjustment of bundles of rays that leave 3D feature points onto each camera centres with respect to both camera positions and point coordinates. It refines the reconstruction to produce jointly optimal 3D structure and viewing parameters by minimizing the cost function for a model fitting error [8, 13]. BA minimizes the re-projection error between the observed and the predicted image points, which is expressed for m images and n points as,

$$\min_{P,X} \sum_{i=1}^{n} \sum_{j=1}^{m} d(Q(P_j, X_i), x_{ij})^2 \tag{1}$$

where $Q(P_j, X_i)$ is the predicted projection of point i on image j and $d(x, y)$ the Euclidean distance between the inhomogeneous image points represented by x and y. Bundle Adjustment is carried out using the Levenberg-Marquardt algorithm [10, 14] because of its effective damping strategy to converge quickly from a wide range of initial guesses. Given the parameter vector $\mathbf{p}$, the functional relation f, and measured vector $\mathbf{x}$, it is required to find δ_p to minimize the quantity $\|x - f(\mathbf{p} + \delta_p)\|$. Assuming the function to be linear in the neighborhood of p, this leads to the equation

$$(\mathbf{J^T J} + \mu \mathbf{I})\delta_p = \mathbf{J^T}\epsilon \tag{2}$$

where J is the Jacobian matrix $J = \frac{\partial \mathbf{x}}{\partial \mathbf{p}}$. LM Algorithm performs iterative minimization by adjusting the damping term μ[15], which assure a reduction in the error ϵ.

BA can be cast as non-linear minimization problem as follows. A parameter vector $\mathbf{P} \in \mathbf{R}^M$ is defined by the m projection matrices and the n 3D points, as

$$\mathbf{P} = (\mathbf{a}_1^T, \ldots, \mathbf{a}_m^T, \mathbf{b}_1^T, \ldots, \mathbf{b}_n^T)^T, \tag{3}$$

where $\mathbf{a}_j$ is the j^{th} camera parameters and $\mathbf{b}_i$ is the i^{th} 3D point coordinates. A measurement vector $\mathbf{X} \in \mathbf{R}^n$ is the measured image coordinates across all cameras:

$$\mathbf{X} = (\mathbf{x}_{11}^T, \ldots, \mathbf{x}_{1m}^T, \mathbf{x}_{21}^T, \ldots, \mathbf{x}_{2m}^T, \ldots, \mathbf{x}_{n1}^T, \ldots, \mathbf{x}_{nm}^T)^T. \tag{4}$$

The estimated measurement vector $\hat{\mathbf{X}}$ using a functional relation $\hat{\mathbf{X}} = f(\mathbf{P})$ given by

$$\hat{\mathbf{X}} = (\hat{\mathbf{x}}_{11}^T, \ldots, \hat{\mathbf{x}}_{1m}^T, \hat{\mathbf{x}}_{21}^T, \ldots, \hat{\mathbf{x}}_{2m}^T, \ldots, \hat{\mathbf{x}}_{n1}^T, \ldots, \hat{\mathbf{x}}_{nm}^T)^T \tag{5}$$

with $\hat{\mathbf{x}}_{ij} = \mathbf{Q}(\mathbf{a_j}, \mathbf{b_i})$. BA minimizes the squared Mahalanobis distance $\epsilon^T \Sigma_x^{-1} \epsilon$, where $\epsilon = \mathbf{X} - \hat{\mathbf{X}}$, over $\mathbf{P}$. Using LM Algorithm, we get the normal equation as

$$(\mathbf{J^T \Sigma_X^{-1} J} + \mu \mathbf{I})\delta = \mathbf{J^T \Sigma_X^{-1}} \epsilon. \tag{6}$$

Apart from the notations above, *mnp* denotes the number of measurement parameters, *cnp* the number of camera parameters and *pnp* the number of point parameters. The total number of projections onto cameras is denoted by *nnz*, which is the length of vector $\mathbf{X}$.

The solution to Equation 6 has a cubic time complexity in the number of parameters and is not practical when the number of cameras and points are high. The Jacobian matrix for BA, however has a sparse block structure. Sparse BA uses a sparse variant of the LM Algorithm [8]. It takes as input the parameter vector $\mathbf{P}$, a function $\mathbf{Q}$ used to compute the predicted projections $\hat{\mathbf{x}}_{ij}$, the observed projections $\mathbf{x}_{ij}$ from i^{th} point on the j^{th} image and damping term μ for LM and returns as an output the solution δ to the normal equation as given in Equation 6. Algorithm 1 outlines the SBA and indicates the steps that are mapped onto the GPU. All the computations are performed using double precision arithmetic as it is a must for accuracy. However, double precision computations is very slow on the GPUs we use. We discuss the details of our BA implementation now.

3.1 Data Structure for the Sparse Bundle Adjustment

Since most of the 3D points are not visible in all cameras, we need a visibility mask to represent the visibility of points onto cameras. Visibility mask is a boolean mask built such that the $(i, j)^{th}$ location is true if i^{th} point is visible in the j^{th} image. We propose to divide the reconstruction consisting of cameras and 3D points into camera tiles or sets of 3D points visible in a camera. Since the number of cameras is less than number of 3D points and bundle of light rays projecting on a camera can be processed independent of other cameras, this division can be easily mapped into blocks and threads on fine grained parallel machines like GPU. The visibility mask is sparse in nature since 3D points are visible in nearby cameras only and not all. We compress the visibility mask using Compressed Column Storage (CCS) [16]. Figure 1 shows a visibility mask for 4 cameras and 4 points and its Compressed Column Storage. We do not store the *val* array as in standard CCS [16] as it is same as the array index in 3D point indices array. The space required to store this is $(nnz + m) \times 4$ bytes whereas to store the whole visibility matrix is $m \times n$ bytes. Since the projections $\hat{\mathbf{x}}_{ij}, \mathbf{x}_{ij}$ and the Jacobian $\mathbf{A}_{ij}, \mathbf{B}_{ij}$ is non zero only when the i^{th} 3D point is visible in the j^{th} camera, it is also sparse in nature and thereby stored in contiguous locations using CCS which is indexed through the visibility mask.

Algorithm 1 SBA $(\mathbf{P}, \mathbf{Q}, x, \mu)$

1: Compute the Predicted Projections $\hat{x}_{ij}$ using $\mathbf{P}$ and $\mathbf{Q}$. ▷ *Computed on GPU*

2: Compute the error vectors $\epsilon_{ij} \leftarrow x_{ij} - \hat{x}_{ij}$ ▷ *Computed on GPU*

3: Assign $\mathbf{J} \leftarrow \frac{\partial \mathbf{X}}{\partial \mathbf{P}}$ (Jacobian Matrix) where
$\mathbf{A}ij \leftarrow \frac{\partial \hat{x}_{ij}}{\partial a_j} = \frac{\partial \mathbf{Q}(a_j, b_i)}{\partial a_j}$ $(\frac{\partial \hat{x}_{ij}}{\partial a_k} = 0 \, \forall i \neq k)$ and
$\mathbf{B}ij \leftarrow \frac{\partial \hat{x}_{ij}}{\partial b_i} = \frac{\partial \mathbf{Q}(a_j, b_i)}{\partial b_i}$ $(\frac{\partial \hat{x}_{ij}}{\partial b_k} = 0 \, \forall j \neq k)$ ▷ *Computed on GPU*

4: Assign $\mathbf{J^T \Sigma_X^{-1} J} \leftarrow \begin{pmatrix} \mathbf{U} & \mathbf{W} \\ \mathbf{W^T} & \mathbf{V} \end{pmatrix}$ where U,V,W is given as
$\mathbf{U}_j \leftarrow \sum_i \mathbf{A}_{ij}^T \mathbf{\Sigma}_{x_{ij}}^{-1} \mathbf{A}_{ij}$, $\mathbf{V}_i \leftarrow \sum_j \mathbf{B}_{ij}^T \mathbf{\Sigma}_{x_{ij}}^{-1} \mathbf{B}_{ij}$ and
$\mathbf{W}_{ij} \leftarrow \mathbf{A}_{ij}^T \mathbf{\Sigma}_{x_{ij}}^{-1} \mathbf{B}_{ij}$ ▷ *Computed on GPU*

5: Compute $\mathbf{J^T \Sigma_X^{-1}} \epsilon$ as $\epsilon_{a_j} \leftarrow \sum_i \mathbf{A}_{ij}^T \mathbf{\Sigma}_{x_{ij}}^{-1} \epsilon_{ij}$,
$\epsilon_{b_i} \leftarrow \sum_j \mathbf{B}_{ij}^T \mathbf{\Sigma}_{x_{ij}}^{-1} \epsilon_{ij}$ ▷ *Computed on CPU*

6: Augment $\mathbf{U}_j$ and $\mathbf{V}_i$ by adding μ to diagonals to yield
$\mathbf{U}_j^*$ and $\mathbf{V}_i^*$ ▷ *Computed on GPU*

7: Normal Equation: $\begin{pmatrix} \mathbf{U}^* & \mathbf{W} \\ \mathbf{W^T} & \mathbf{V}^* \end{pmatrix} \begin{pmatrix} \delta_a \\ \delta_b \end{pmatrix} = \begin{pmatrix} \epsilon_a \\ \epsilon_b \end{pmatrix}$ ▷ Using Equation (6)

8: $\begin{pmatrix} \underbrace{\mathbf{U}^* - \mathbf{W}\mathbf{V}^{*-1}\mathbf{W}^T}_{\mathbf{S}} & 0 \\ \mathbf{W}^T & \mathbf{V}^* \end{pmatrix} \begin{pmatrix} \delta_a \\ \delta_b \end{pmatrix} = \begin{pmatrix} \underbrace{\epsilon_a - \mathbf{W}\mathbf{V}^{*-1}\epsilon_b}_{\mathbf{e}} \\ \epsilon_b \end{pmatrix}$ ▷ Using Schur Complement

9: Compute $\mathbf{Y}_{ij} \leftarrow \mathbf{W}_{ij} \mathbf{V}_i^{*-1}$ ▷ *Computed on GPU*

10: Compute $\mathbf{S}_{jk} \leftarrow \mathbf{U}_j^* - \sum_i \mathbf{Y}_{ij} \mathbf{W}_{ik}^T$ ▷ *Computed on GPU*

11: Compute $e_j \leftarrow \epsilon_{a_j} - \sum_i \mathbf{Y}_{ij} \epsilon_{b_i}$ ▷ *Computed on CPU*

12: Compute δ_a as $(\delta_{a1}^T, \ldots, \delta_{am}^T)^T = \mathbf{S}^{-1}(e_1^T, \ldots, e_m^T)^T$ ▷ *Computed on GPU*

13: Compute $\delta_{b_i} \leftarrow \mathbf{V}_i^{*-1}(\epsilon_{b_i} - \sum_j \mathbf{W}_{ij}^T \delta_{a_j})$ ▷ *Computed on GPU*

14: Form δ as $(\delta_a^T, \delta_b^T)^T$

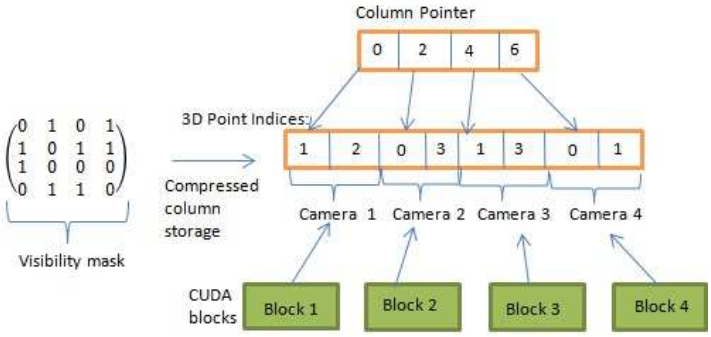


Fig. 1. An example of the compressed column storage of visibility mask having 4 cameras and 4 3D Points. Each CUDA Block processes one set of 3D points.

3.2 Computation of the Initial Projection and Error Vector

Given $\mathbf{P}$ and $\mathbf{Q}$ as input, the initial projection is calculated as $\hat{\mathbf{X}} = \mathbf{Q}(\mathbf{P})$ (Algorithm 1,line 1) where $\hat{\mathbf{X}}$ is the estimated measurement vector and $\hat{\mathbf{x}}_{ij} = \mathbf{Q}(\mathbf{a_j}, \mathbf{b_i})$ is the projection of point b_i on the camera a_j using the function $\mathbf{Q}$. The error vector is calculated as $\epsilon_{ij} = \mathbf{x}_{ij} - \hat{\mathbf{x}}_{ij}$ where $\mathbf{x}_{ij}$ and $\hat{\mathbf{x}}_{ij}$ are the measured and estimated projections. The estimated projections and error vectors consumes memory space of $nnz \times mnp$ each. Our implementation consists of m thread blocks running in parallel, with each thread of block j computing a projection to the camera j. The number of threads per block is limited by the total number of registers available per block and a maximum limit of number of threads per block. Since the typical number of points seen by a camera is of the order of thousands (more than the limit on threads) we loop over all the 3D points visible by a camera in order to compute projections. The GPU kernel to calculate the initial projection and error vector is shown in Algorithm 2.

Algorithm 2 CUDA_INITPROJ_KERNEL $(\mathbf{P}, \mathbf{Q}, \mathbf{X})$

1: CameraID ← BlockID
2: Load the camera parameters into shared memory
3: **repeat**
4: Load the 3D point parameters (given ThreadID and CameraID)
5: Calculate the Projection $\hat{x}_{ij}$ given 3D Point i and Camera j
6: Calculate the Error Vector using $\epsilon_{ij} = \mathbf{x}_{ij} - \hat{\mathbf{x}}_{ij}$
7: Store the Projections and Error Vector back into global memory
8: **until** all the projections are calculated

3.3 Computation of the Jacobian Matrix (J)

The Jacobian matrix is calculated as $\mathbf{J} = \frac{\partial \mathbf{X}}{\partial \mathbf{P}}$ (Algorithm 1,line 3). For $\hat{\mathbf{X}} = (\hat{\mathbf{x}}_{11}^T, \ldots, \hat{\mathbf{x}}_{n1}^T, \hat{\mathbf{x}}_{12}^T, \ldots, \hat{\mathbf{x}}_{n2}^T, \ldots, \hat{\mathbf{x}}_{1m}^T, \ldots, \hat{\mathbf{x}}_{nm}^T)^T$, the Jacobian would be $(\frac{\partial \hat{\mathbf{x}}_{11}}{\partial \mathbf{P}}^T, \ldots, \frac{\partial \hat{\mathbf{x}}_{n1}}{\partial \mathbf{P}}^T, \frac{\partial \hat{\mathbf{x}}_{12}}{\partial \mathbf{P}}^T, \ldots, \frac{\partial \hat{\mathbf{x}}_{n2}}{\partial \mathbf{P}}^T, \ldots, \frac{\partial \hat{\mathbf{x}}_{1m}}{\partial \mathbf{P}}^T, \ldots, \frac{\partial \hat{\mathbf{x}}_{nm}}{\partial \mathbf{P}}^T)$. Since $\frac{\partial \hat{x}_{ij}}{\partial a_k} = 0 \, \forall i \neq k$ and $\frac{\partial \hat{x}_{ij}}{\partial b_k} = 0 \, \forall j \neq k$, the matrix is sparse in nature.

For the example, shown in Figure 1, the Jacobian matrix would be

$$J = \begin{pmatrix} A_{10} & 0 & 0 & 0 & 0 & B_{10} & 0 & 0 \\ A_{20} & 0 & 0 & 0 & 0 & 0 & B_{20} & 0 \\ 0 & A_{01} & 0 & 0 & B_{01} & 0 & 0 & 0 \\ 0 & A_{31} & 0 & 0 & 0 & 0 & 0 & B_{31} \\ 0 & 0 & A_{12} & 0 & 0 & B_{12} & 0 & 0 \\ 0 & 0 & A_{32} & 0 & 0 & 0 & 0 & B_{32} \\ 0 & 0 & 0 & A_{03} & B_{03} & 0 & 0 & 0 \\ 0 & 0 & 0 & A_{13} & 0 & B_{13} & 0 & 0 \end{pmatrix} \tag{7}$$

where, $\mathbf{A}ij = \frac{\partial \hat{x}_{ij}}{\partial a_j} = \frac{\partial \mathbf{Q}(a_j, b_i)}{\partial a_j}$ and $\mathbf{B}ij = \frac{\partial \hat{x}_{ij}}{\partial b_i} = \frac{\partial \mathbf{Q}(a_j, b_i)}{\partial b_i}$. The matrix when stored in compressed format would be
$\mathbf{J} = (A_{10}, B_{10}, A_{20}, B_{20}, A_{01}, B_{01}, A_{31}, B_{31}, A_{12}, B_{12}, A_{32}, B_{32}, A_{03}, B_{03}, A_{13}, B_{13})$
The memory required is $(cnp + pnp) \times mnp \times nnz \times 4$ bytes. The CUDA grid structure used in Jacobian computation is similar to initial projection computation. Block j processes the A_{ij} and B_{ij}, corresponding to the j^{th} camera. The kernel to calculate the Jacobian Matrix is shown in Algorithm 3.

Algorithm 3 CUDA_JACOBIAN_KERNEL $(\mathbf{P}, \mathbf{Q})$

1: CameraID ← BlockID
2: **repeat**
3: Load the 3D point parameters and Camera parameters (given ThreadID and CameraID) into thread memory.
4: Calculate B_{ij} followed by A_{ij} using scalable finite differentiation
5: Store the A_{ij} and B_{ij} into global memory at contiguous locations.
6: **until** all the projections are calculated

3.4 Computation of $J^T \Sigma_X^{-1} J$

$\mathbf{J^T \Sigma_X^{-1} J}$ is given as $\begin{pmatrix} \mathbf{U} & \mathbf{W} \\ \mathbf{W^T} & \mathbf{V} \end{pmatrix}$ where $\mathbf{U}_j = \sum_i \mathbf{A}_{ij}^T \mathbf{\Sigma}_{x_{ij}}^{-1} \mathbf{A}_{ij}$, $\mathbf{V}_i = \sum_j \mathbf{B}_{ij}^T \mathbf{\Sigma}_{x_{ij}}^{-1} \mathbf{B}_{ij}$ and $\mathbf{W}_{ij} = \mathbf{A}_{ij}^T \mathbf{\Sigma}_{x_{ij}}^{-1} \mathbf{B}_{ij}$. For the example in Figure 1, $\mathbf{J^T \Sigma_X^{-1} J}$ is given as:

$$\mathbf{J^T \Sigma_X^{-1} J} = \begin{pmatrix} U_0 & 0 & 0 & 0 & 0 & W_{10} & W_{20} & 0 \\ 0 & U_1 & 0 & 0 & W_{01} & 0 & 0 & W_{31} \\ 0 & 0 & U_2 & 0 & 0 & W_{12} & 0 & W_{32} \\ 0 & 0 & 0 & U_3 & W_{03} & W_{13} & 0 & 0 \\ 0 & W_{01}^T & 0 & W_{03}^T & V_0 & 0 & 0 & 0 \\ W_{10}^T & 0 & W_{12}^T & W_{13}^T & 0 & V_1 & 0 & 0 \\ W_{20}^T & 0 & 0 & 0 & 0 & 0 & V_2 & 0 \\ 0 & W_{31}^T & W_{32}^T & 0 & 0 & 0 & 0 & V_3 \end{pmatrix} \tag{8}$$

Computation of U: The CUDA grid structure consists m blocks, such that each block processes U_j where j is the BlockID. Thread i in block j processes $\mathbf{A}_{ij}^T \mathbf{\Sigma}_{x_{ij}}^{-1} \mathbf{A}_{ij}$, which is stored in the appropriate segment. The summation is done using a segmented scan[17]. The memory space required to store $\mathbf{U}$ is $cnp \times cnp \times m \times 4$ bytes. The computation of $\mathbf{U}$ is done as described in Algorithm 4.

Computation of V: The CUDA grid structure and computation of V is similar to the computation of U. The basic difference between the two is that $\mathbf{B}_{ij}^T \mathbf{\Sigma}_{x_{ij}}^{-1} \mathbf{B}_{ij}$ is stored in the segment for point i for reduction using segmented scan. The memory space required to store $\mathbf{V}$ is $pnp \times pnp \times n \times 4$ bytes.

Algorithm 4 CUDA_U_KERNEL ($\mathbf{A}$)

1: CameraID ← BlockID
2: **repeat**
3: Load A_{ij} where j = CameraID (for a given thread)
4: Calculate $A_{ij} \times A_{ij}^T$ and store into appropriate global memory segment
5: **until** all the A_{ij} are calculated for the j_{th} camera
6: Run Segmented Scan to get the final sum.

Computation of W: The computation of each W_{ij} is independent of all other W_{ij} as there is no summation involved as in **U** and **V**. Therefore the computation load is equally divided among all blocks in GPU. $\lceil \frac{nnz}{10} \rceil$ thread blocks are launched with each block processing 10 W matrices. This block configuration gave us the maximum CUDA occupancy. The memory space required to store **W** is $pnp \times cnp \times nnz \times 4$ bytes. The computation of **W** is done as described in Algorithm 5.

Algorithm 5 CUDA_W_KERNEL ($\mathbf{A}, \mathbf{B}$)

1: Load A_{ij} and B_{ij} for each warp of threads.
2: Calculate $A_{ij} \times B_{ij}^T$
3: Store W_{ij} back into global memory at appropriate location.

3.5 Computation of $\mathbf{S = U^* - WV^{*-1}W^T}$

The computation of S is the most demanding step of all the modules (Algorithm 1, line 10). Figure 3 shows the split up of computation time among all components. After calculating U,V and W, augmentation of U,V is done by calling a simple kernel, with m, n blocks with each block adding μ to the respective diagonal elements. Since V^* is a block diagonal matrix, it's inverse can be easily calculated through a kernel with n blocks, with each block calculating the inverse of V^* submatrix (of size $pnp \times pnp$).

Computation of $\mathbf{Y = WV^{*-1}}$: Computation of **Y** is similar to the computation of **W**. $\lceil \frac{nnz}{10} \rceil$ thread blocks are launched with each block processing 10 Y matrices and each warp of thread computing $W_{ij} \times V_i^{*-1}$.

Computation of $\mathbf{U^* - YW^T}$: **S** is a symmetric matrix, so we calculate only the upper diagonal. The memory space required to store **S** is $m \times m \times 81 \times 4$ bytes. The CUDA grid structure consists of $m \times m$ blocks. Each block is assigned to a 9×9 submatrix in the upper diagonal, where each block calculates one $S_{ij} = U_{ij} - \sum_k Y_{ki} W_{kj}^T$. Limited by the amount of shared memory available and number of registers available per block, only 320 threads are launched. The algorithm used for computation is given in Algorithm 6.

Algorithm 6 CUDA_S_KERNEL ($\mathbf{U}^*, \mathbf{Y}, \mathbf{W^T}$)

1: **repeat** (for S_{ij})
2: Load 320 3D Point indices (given camera set i) into shared memory
3: Search for loaded indices in camera set j and load them into shared memory.
4: **for all** 320 points loaded in shared memory **do**
5: Load 10 indices of the camera set i and j from the shared memory.
6: For each warp, compute $Y_{ki}W_{kj}^T$ and store them in shared memory
7: SYNCTHREADS
8: Sum the shared memory and add to the final sum in a shared memory.
9: **end for**
10: **until** all the common 3D points are loaded.
11: **if** $i == j$ **then**
12: Compute $Y_{ii}W_{ii}^T \leftarrow U_{ii}^* - Y_{ii}W_{ii}^T$
13: **end if**
14: Store $Y_{ij}W_{ij}^T$ into global memory.

3.6 Computation of the Inverse of S

As the S Matrix is symmetric and positive definite, Cholesky decomposition is used to perform the inverse operation (Algorithm 1, line 12). Cholesky decomposition is done using the MAGMA library [18], which is highly optimized using the fine and coarse grained parallelism on GPUs as well benefits from hybrids computations by using both CPUs and GPUs. It achieves a peak performance of 282 GFlops for double precision. Since GPU's single precision performance is much higher than it's double precision performance, it used the mixed precision iterative refinement technique, in order to find inverse, which results in a speedup of more than 10 times than CPU.

3.7 Scheduling of Blocks on CPU and GPU

Figure 2 shows the way CPU and GPU work together, in order to maximize the overall throughput. While the computationally intense left hand side of the equations are calculated on GPU, the relatively lighter right hand side are computed on CPU. The blocks connected by the same vertical line are calculated in parallel on CPU and GPU. The computations on the CPU and the GPU overlap. The communications are also performed asynchronously, to ensure that the GPU doesn't lie idle from the start to the finish of an iteration.

4 Experimental Results

In this section we analyze the performance of our results with the CPU implementation of Bundle Adjustment [8]. We tested our algorithm on an Intel Core i7 CPU @ 2.66 GHz and a quarter of an NVIDIA Tesla S1070 [19] Graphics Processor with CUDA 2.2. All computations were performed in double precision,

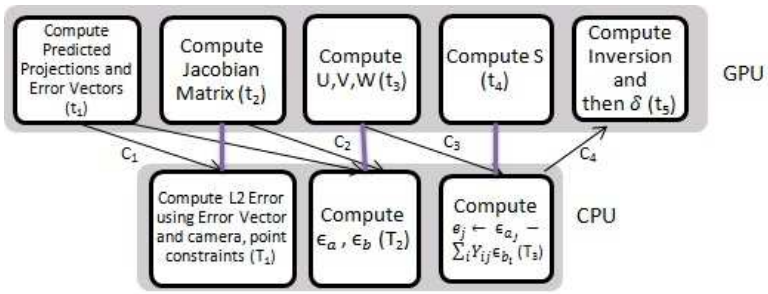


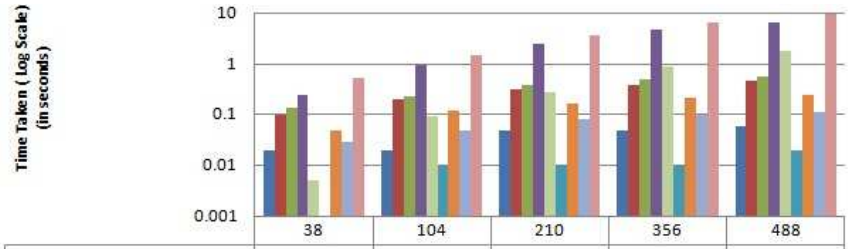
Fig. 2. Scheduling of Blocks on CPU and GPU. Arrows show the data dependency between various modules on CPU and GPU. Modules connected through a vertical line are computed in parallel on CPU and GPU.

though it is slow on this GPU. Single precision computations had correctness issues for this problem.

We used the Notre Dame 715 dataset [20] for our experiments. We ran the 3D reconstruction process on the data set and the input and output parameters $(\mathbf{P}, \mathbf{Q}, x, \mu, \delta)$ were extracted and stored for bundle adjustment. We focussed on getting good performance for a dataset of around 500 images as explained before. The redundancy is being exploited for larger data sets using a minimal skeletal subset of similar size by other researchers [2, 6]. We used a 488 image subset to analyze the performance and to compare it with the popular implementation of bundle adjustment [8].

Figure 3 shows the time taken for a single iteration by the major components. The **S** Computation takes most of the time, followed by the **S** inverse computation. The Schur complement takes about 70% of the computation time for **S**, as it involves $\mathcal{O}(m^2 \times mnp \times pnp \times cnp \times mnvis)$ operations, where $mnvis$ is the maximum number of 3D points visible by a single camera. On the GPU, each of the m^2 blocks performs $\mathcal{O}(mnp \times pnp \times cnp \times mnvis)$ computations. Part of **S** computation are raw computation(60% of the time), reducing the sum (30%) and search operation (10%). It is also limited by the amount of shared memory. The Jacobian computation is highly data parallel and maps nicely to GPU architecture which results in high speedup. Rest of the kernels (U, V, W and initial projection) are relatively light.

As shown in Figure 4, the total running time on GPU is $t = t_1 + t_2 + t_3 + t_4 + C_4 + t_5$ and on CPU is $T = T_1 + C_1 + T_2 + C_2 + T_3 + C_3$ where t_i is the time taken by GPU modules, T_i time taken by CPU modules and C_i communication time. The total time taken is $max(t, T)$. CPU-GPU parallelization takes place only when $max(t, T) < (t + T)$. For the case of 488 cameras, the time taken by GPU completely overlaps the CPU and communication time, so that there is no idle time for GPU. Figure 5 shows the comparison of time taken for each iteration of Bundle Adjustment between the CPU and GPU implementations for Notre Dame dataset. We get a speedup of 8-10 times over the CPU implementation.



	38	104	210	356	488
GPU - Init Proj Computation	0.02	0.02	0.05	0.05	0.06
GPU - Jacobian Computation	0.1	0.2	0.32	0.39	0.45
GPU - U,V,W Computation	0.14	0.23	0.39	0.5	0.56
GPU - S Computation	0.25	0.97	2.5	4.63	6.55
GPU - Inverse Calculation	0.005	0.09	0.282	0.8675	1.743
CPU - L2 Error Computation	0	0.01	0.01	0.01	0.02
CPU - εa, εb Computation	0.05	0.12	0.17	0.21	0.24
CPU - e Computation	0.03	0.05	0.08	0.1	0.11
Total Time	0.515	1.51	3.542	6.4375	9.363

Fig. 3. Time(sec) taken for each step in one iteration of Bundle Adjustment on GPU and CPU for various number of cameras of Notre Dame data set. Total time is the time taken by hybrid implementation of BA using CPU and GPU in parallel.

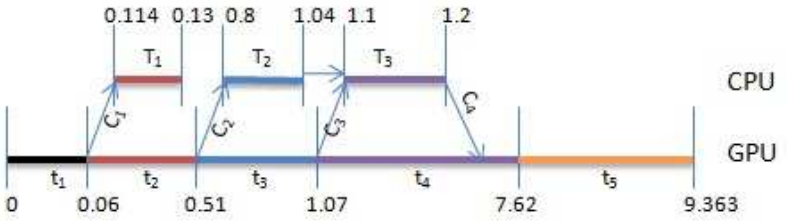


Fig. 4. Timings shown are the time taken by each component either on CPU or GPU including the memory transfer time in one iteration for 488 cameras

4.1 Memory Requirements

The total memory used is a major limiting factor in the scalability of bundle adjustment for large scale 3D reconstruction. As we can see in Figure 6, the total memory requirement is high due to temporary requirements in the segmented scan [17] operation. The extra memory required is of the size $3 \times nnz \times 81 \times 4$ bytes which is used to store the data, flag and the final output arrays for the segmented scan operation. The permanent memory used to store the permanent arrays such as **J, U, V, W,** and **S** is only a moderate fraction of the total memory required.

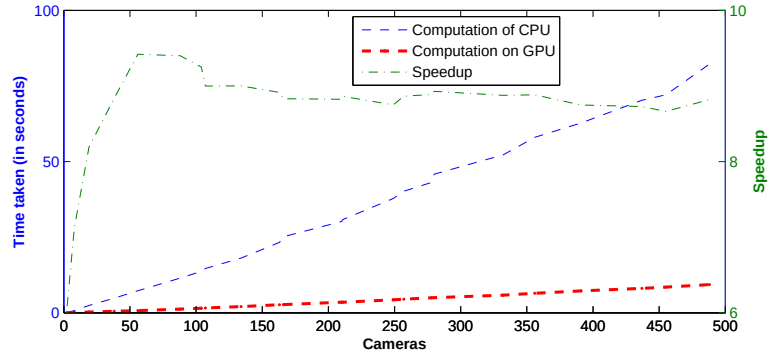


Fig. 5. Time Comparison (one iteration) of Bundle Adjustment Computation on CPU and GPU

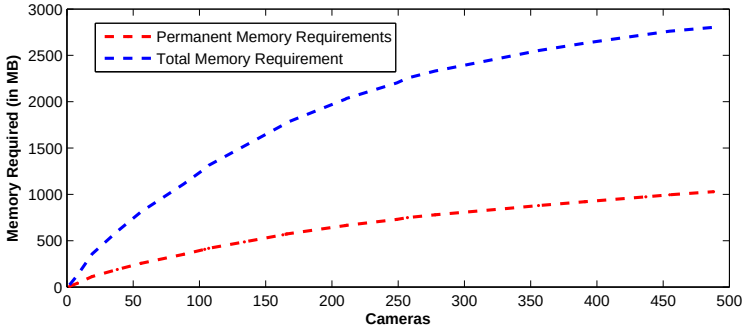


Fig. 6. Memory required (in MB) for bundle adjustment for various number of cameras.

5 Conclusions and Future Work

In this paper, we introduced a hybrid algorithm using the GPU and the CPU to perform practical time bundle adjustment. The time taken for each iteration for 488 cameras on using our approach is around 9 seconds, compared to 81 seconds on the CPU. This can reduce the computation time of a week on CPU to less than a day. This can make processing larger datasets practical. Most of the computations in our case is limited by the amount of available shared memory, registers and the limit on number of threads. The slow double precision computations also contribute to the computation time on today's GPUs. The newer GPU of the Fermi line has much better double precision performance and better memory architecture. We are in the process of adapting our approach to the Fermi and expect significant speedups on it. A multi GPU implementation of the same algorithm is also being explored for faster overall processing.

References

1. Snavely, N., Seitz, S.M., Szeliski, R.: Photo tourism: exploring photo collections in 3d. In: SIGGRAPH '06: ACM SIGGRAPH 2006 Papers, New York, NY, USA, ACM (2006) 835–846
2. Agarwal, S., Snavely, N., Simon, I., Seitz, S.M., Szeliski, R.: Building rome in a day. In: Twelfth IEEE International Conference on Computer Vision (ICCV 2009), Kyoto, Japan, IEEE (2009)
3. Vineet, V., Narayanan, P.J.: Cuda cuts: Fast graph cuts on the gpu. Computer Vision and Pattern Recognition Workshop **0** (2008) 1–8
4. Sinha, S.N., michael Frahm, J., Pollefeys, M., Genc, Y.: Gpu-based video feature tracking and matching. Technical report, In Workshop on Edge Computing Using New Commodity Architectures (2006)
5. Fung, J., Mann, S.: Openvidia: parallel gpu computer vision. In: MULTIMEDIA '05: Proceedings of the 13th annual ACM international conference on Multimedia, New York, NY, USA, ACM (2005) 849–852
6. Snavely, N., Seitz, S.M., Szeliski, R.: Skeletal graphs for efficient structure from motion. In: CVPR. (2008)
7. Brown, M., Lowe, D.G.: Unsupervised 3d object recognition and reconstruction in unordered datasets. In: 3DIM '05: Proceedings of the Fifth International Conference on 3-D Digital Imaging and Modeling, Washington, DC, USA, IEEE Computer Society (2005) 56–63
8. Lourakis, M.A., Argyros, A.: SBA: A Software Package for Generic Sparse Bundle Adjustment. ACM Trans. Math. Software **36** (2009) 1–30
9. Ni, K., Steedly, D., Dellaert, F.: Out-of-core bundle adjustment for large-scale 3d reconstruction. In: ICCV. (2007) 1–8
10. Lourakis, M.: levmar: Levenberg-marquardt nonlinear least squares algorithms in C/C++. [web page] `http://www.ics.forth.gr/~lourakis/levmar/` (Jul. 2004)
11. Byröd, M., Äström, K.: Bundle adjustment using conjugate gradients with multiscale preconditioning. In: BMVC. (2009)
12. Cao, J., Novstrup, K.A., Goyal, A., Midkiff, S.P., Caruthers, J.M.: A parallel levenberg-marquardt algorithm. In: ICS '09: Proceedings of the 23rd international conference on Supercomputing, New York, NY, USA, ACM (2009) 450–459
13. Triggs, B., Mclauchlan, P., Hartley, R., Fitzgibbon, A.: Bundle adjustment a modern synthesis. In: Vision Algorithms: Theory and Practice, LNCS, Springer Verlag (2000) 298–375
14. Ranganathan, A.: The levenberg-marquardt algorithm. Technical Report http://www.ananth.in/docs/lmtut.pdf, Honda Research Institute (2004)
15. Nielsen, H.: Damping parameter in marquardt's method. Technical Report http://www.imm.dtu.dk/ hbn, Technical University of Denmark (1999)
16. Dongarra, J.: Compressed column storage. [web page] `http://netlib2.cs.utk.edu/linalg/html_templates/node92.html` (1995)
17. Sengupta, S., Harris, M., Garland, M.: Efficient parallel scan algorithms for gpus. Technical report, NVIDIA Technical Report (2008)
18. Ltaief, H., Tomov, S., Nath, R., Dongarra, J.: Hybrid multicore cholesky factorization with multiple gpu accelerators. (Transaction on Parallel and Distributed Systems)
19. Lindholm, E., Nickolls, J., Oberman, S., Montrym, J.: Nvidia tesla: A unified graphics and computing architecture. IEEE Micro **28** (2008) 39–55
20. Snavely, N.: Notre dame dataset. [web page] `http://phototour.cs.washington.edu/datasets/` (2009)