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.choudhary@research.,shubham@students.,pjn@}iiit.ac.in

Abstract. Large-scale 3D reconstruction has received a lot of attention recently. Bundle adjustment is a key component of the reconstruction pipeline and often its slowest and most computational resource intensive. It hasn't been parallelized effectively so far. In this paper, we present a hybrid implementation of sparse bundle adjustment on the GPU using CUDA, with the CPU working in parallel. The algorithm is decomposed into smaller steps, each of which is scheduled on the GPU or the CPU. We develop efficient kernels for the steps and make use of existing libraries for several steps. Our implementation outperforms the CPU implementation significantly, achieving a speedup of 30-40 times over the standard CPU implementation for datasets with upto 500 images on an Nvidia Tesla C2050 GPU.

1 Introduction

Large scale sparse 3D reconstruction from community photo collections using the structure from motion (SfM) pipeline is an active research area today. The SfM pipeline has several steps. The joint optimization of camera positions and point coordinates using Bundle Adjustment (BA) is the last step. Bundle adjustment is an iterative step, typically performed using the Levenberg-Marquardt (LM) non-linear optimization scheme. Bundle adjustment is the primary bottleneck of the SfM, consuming about 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 BA step is still performed on a single core, though most other steps are performed on a cluster of processors [2]. Speeding up of BA by parallelizing it can have a significant impact on large scale SfM efforts.

The rapid increase in the performance has made the graphics processor unig (GPU) a viable candidate for many compute intensive tasks. GPUs are being used for many computer vision applications [3], such as Graph Cuts [4], tracking [5] and large scale 3D reconstruction [6]. 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. The computation requirements

K.N. Kutulakos (Ed.): ECCV 2010 Workshops, Part II, LNCS 6554, pp. 423–435, 2012.

of BA grows rapidly with the number of images. However, the visibility aspects of points on cameras places a natural limit on how many images need to be processed together. The current approach is to identify clusters of images and points to be processed together [7]. 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.

Our goal is to develop a practical time implementation by exploiting the computing resources of the CPU and the GPU. We decompose the LM algorithm into multiple steps, each of which is performed using a kernel on the GPU or a function on the CPU. Our implementation efficiently schedules the steps on CPU and GPU to minimize the overall computation time. The concerted work of the CPU and the GPU is critical to the overall performance gain. The executions of the CPU and GPU are fully overlapped in our implementation, with no idle time on the GPU. We achieve a speedup of 30-40 times on an Nvidia Tesla C2050 GPU on a dataset of about 500 images.

2 Related Work

Brown and Lowe presented the SfM pipeline for unordered data sets [8]. Phototourism is an application of 3D reconstruction for interactively browsing and exploring large collection of unstructured photographs [1]. The problem of large scale 3D reconstruction takes advantage of the redundancy available in the large collection of unordered dataset of images and maximizes the parallelization available in the SFM pipeline [2,7]. Bundle Adjustment was originally conceived in photogrammetry [9], and has been adapted for large scale reconstructions. Ni et al. solve the problem by dividing it into several submaps which can be optimized in parallel [10]. In general, a sparse variant of Levenberg-Marquardt minimization algorithm [11] is the most widely used choice for BA. A public implementation is available [9]. Byröd and Åström solve the problem using preconditioned conjugate gradients, utilizing the underlying geometric layout [12]. Cao et al. parallelize the dense LM algorithm, but their method is not suited for sparse data [13]. Agarwal et al. design a system to maximize parallelization at each stage in the pipeline, using a cluster of 500 cores for rest of the computations but a single core for bundle adjustment [2]. Frahm et al. uses GPUs to reconstruct 3 million images of Rome in less than 24 hours [6]. They don't use the GPUs for the BA step. No prior work has been reported that parallelizes BA or the LM algorithm.

3 Sparse Bundle Adjustment on the GPU

Bundle adjustment refers to 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 produces jointly optimal 3D structure and

viewing parameters by minimizing the cost function for a model fitting error [9,14]. 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 [11,15] 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[16], 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 $\mathbf{X}$ is the measured image coordinates in 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 $\hat{\mathbf{X}} = f(\mathbf{P})$ is 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 [9]. 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 to gain accuracy.

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 $(\delta_{a_1}^T, \ldots, \delta_{a_m}^T)^T = \mathbf{S}^{-1}(e_1^T, \ldots, e_m^T)^T$ ▷ *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 $(\delta_a^T, \delta_b^T)^T$

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

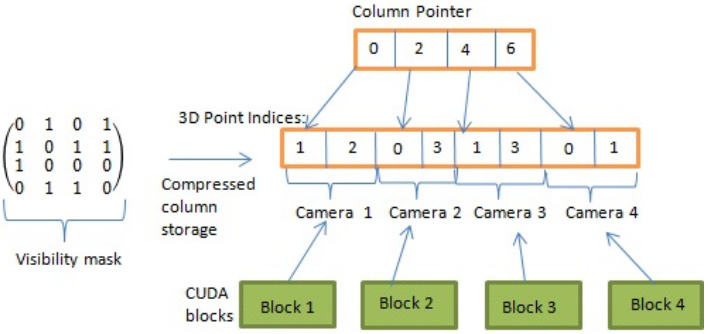


Fig. 1. An example of the compressed column storage of visibility mask having 4 cameras and 4 3D Points. Each CUDA Block processes one set of 3D points.

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) [17]. 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 [17] 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.

3.2 Computation of the Initial Projection and Error Vector

Given $\mathbf{P}$ and $\mathbf{Q}$ as input, the initial projection is calculated as $\hat{\mathbf{X}} = \mathbf{Q(P)}$ (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 $\leftarrow$ 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 $\leftarrow$ 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 faster when using a segmented scan[18] on Tesla S1070 whereas a shared memory reduction is faster on the Fermi GPU. 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.

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

1: CameraID $\leftarrow$ 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: Perform a shared memory reduction to get final sum on Fermi. Write to global memory and perform a segmented scan on Tesla S1070.

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 on Tesla S1070 where as a shared memory reduction is done on Fermi. The memory space required to store $\mathbf{V}$ is $pnp \times pnp \times n \times 4$ bytes.

Computation of W: The computation of each W_{ij} is independent of all other W_{ij} as there is no summation involved as in $\mathbf{U}$ and $\mathbf{V}$. 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 $\mathbf{W}$ is $pnp \times cnp \times nnz \times 4$ bytes. The computation of $\mathbf{W}$ 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). Table 1 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 $\mathbf{Y}$ is similar to the computation of $\mathbf{W}$. $\lceil \frac{nnz}{10} \rceil$ 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 $\mathbf{U^* - YW^T}$: $\mathbf{S}$ is a symmetric matrix, so we calculate only the upper diagonal. The memory space required to store $\mathbf{S}$ 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 add to the partial sum for each warp in shared memory
7: **end for**
8: Synchronize Threads
9: **until** all the common 3D points are loaded.
10: Sum up the partial summations in the shared memory to get the final sum.
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 [19], 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 over the CPU.

3.7 Scheduling of Steps 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.

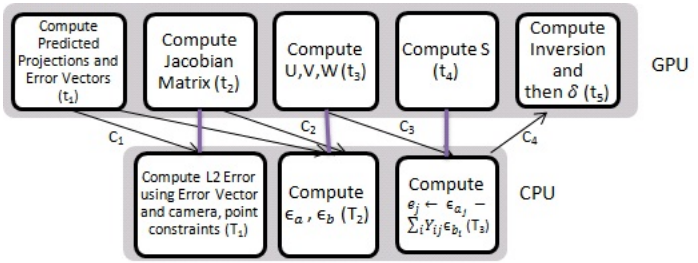


Fig. 2. Scheduling of steps on CPU and GPU. Arrows indicate data dependency between modules. Modules connected through a vertical line are computed in parallel on CPU and GPU.

4 Experimental Results

In this section, we analyze the performance of our approach and compare with the CPU implementation of Bundle Adjustment [9]. We use an Intel Core i7, 2.66GHz CPU. For the GPU, we use a quarter of an Nvidia Tesla S1070 [20] with CUDA 2.2 and an Nvidia Tesla C2050 (Fermi) with CUDA 3.2. All computations were performed in double precision, as single precision computations had correctness issues for this problem.

We used the Notre Dame 715 dataset [21] 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 [2,7]. We used a 488 image subset to analyze the performance and to compare it with the popular implementation of bundle adjustment [9].

Table 1 shows the time taken for a single iteration for each major step. 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. 60% of **S** computation is to find the partial sums, 30% for the reduction, and 10% for the search operation. It is also limited by the amount of shared memory. The Jacobian computation is highly data parallel and maps nicely to the GPU architecture. Rest of the kernels (U, V, W and initial projection) are light.

As shown in Figure 3, the total running time on the 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 parallel operations take place only when $max(t, T) < (t + T)$. For the case of 488 cameras, the time taken by GPU completely overlaps the CPU computations and communication, so that there is no idle time for the GPU. Figure 4 compares the time taken by our hybrid algorithm for each iteration of Bundle Adjustment with the CPU

Table 1. Time in seconds for each step in one iteration of Bundle Adjustment for different number of cameras on the Notre Dame data set. Total time is the time taken by hybrid implementation of BA using CPU and GPU in parallel. GPU1 is a quarter of Tesla S1070 and GPU2 is Tesla C2050.

Computation Step	Time Taken (in seconds)									
	GPU1	GPU2	GPU1	GPU2	GPU1	GPU2	GPU1	GPU2	GPU1	GPU2
	38 Cameras		104 Cameras		210 Cameras		356 Cameras		488 Cameras	
Initial Proj	0.02	0.01	0.02	0.03	0.05	0.04	0.06	0.04	0.06	0.05
Jacobian	0.1	0.04	0.2	0.07	0.32	0.12	0.39	0.16	0.45	0.17
U, V, W Mats	0.14	0.04	0.23	0.09	0.39	0.15	0.5	0.18	0.56	0.2
S Matrix	0.25	0.09	0.97	0.27	2.5	0.56	4.63	1.01	6.55	1.3
S Inverse	0.01	0	0.09	0.02	0.28	0.08	0.87	0.19	1.74	0.39
L2 Err (CPU)	0		0.01		0.01		0.01		0.02	
ϵa, ϵb (CPU)	0.05		0.12		0.17		0.21		0.24	
e (CPU)	0.03		0.05		0.08		0.1		0.11	
Total Time	**0.52**	**0.19**	**1.51**	**0.51**	**3.54**	**0.97**	**6.44**	**1.61**	**9.36**	**2.15**

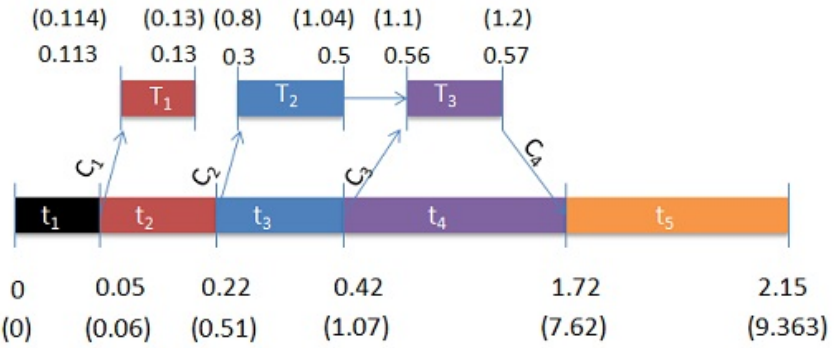
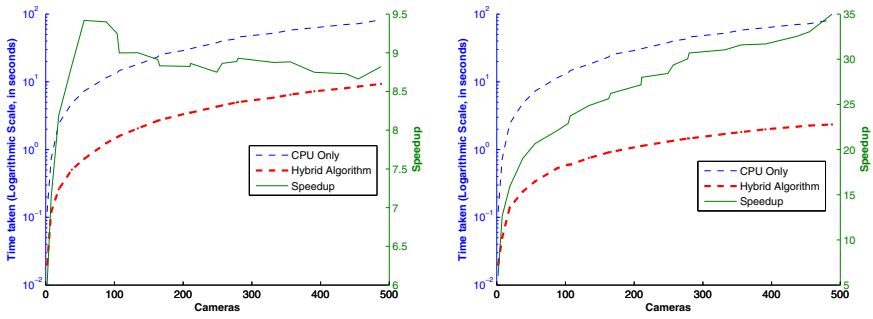


Fig. 3. Starting and ending times for each step including memory transfer for one iteration using 488 cameras. Times in paranthesis are for the use of the S1070 and others for the C2050.

only implementation on the Notre Dame dataset. The hybrid version with Tesla C2050 gets a speedup of 30-40 times over the CPU implementation.

Memory Requirements: The total memory used can be a limiting factor in the scalability of bundle adjustment for large scale 3D reconstruction. As we can see in Figure 5, the total memory requirement is high due to temporary requirements in the segmented scan [18] operation on the earlier GPU. 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. The Fermi has



(a) Using Tesla S1070 (b) Using Tesla C2050

Fig. 4. Time and speedup for one iteration of Bundle Adjustment on the CPU using Tesla S1070 and Tesla S2050.

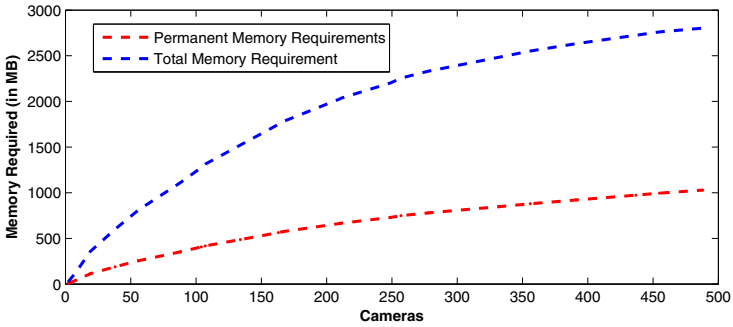


Fig. 5. Memory required (in MB) on the GPU for different number of cameras.

a larger shared memory and the reduction is performed in the shared memory itself. Thus, the total memory requirement is the same as the permanent memory requirement when using Tesla C2050.

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 2 seconds on Tesla C2050 and 9 seconds on Tesla S1070, compared to 81 seconds on the CPU. This can reduce the computation time of a week on CPU to less than 10 hours. 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 double precision performance is critical to the GPU computation; the better performance using Fermi GPUs may also be due to this.

Faster bundle adjustment will enable processing of much larger data sets in the future. One option is to explore better utilization of the CPU. Even the single-core CPU is not used fully in our implementation currently. The 4-core and 8-core CPUs that are freely available can do more work, and will need a relook at the distribution of the tasks between the CPU and the GPU. The use of multiple GPUs to increase the available parallelism is another option. Expensive steps like the computation of **S** matrix can be split among multiple GPUs without adding enormous communication overheas. This will further change the balance between what can be done on the CPU and on the GPU.

References

1. Snavely, N., Seitz, S.M., Szeliski, R.: Photo tourism: exploring photo collections in 3d. ACM Trans. Graph 25, 835–846 (2006)
2. Agarwal, S., Snavely, N., Simon, I., Seitz, S.M., Szeliski, R.: Building rome in a day. In: International Conference on Computer Vision, ICCV (2009)

3. Fung, J., Mann, S.: Openvidia: parallel gpu computer vision. In: MULTIMEDIA 2005: Proceedings of the 13th Annual ACM International Conference on Multimedia, pp. 849–852 (2005)
4. Vineet, V., Narayanan, P.J.: Cuda cuts: Fast graph cuts on the gpu. In: Computer Vision and Pattern Recognition Workshop (2008)
5. 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)
6. Frahm, J.-M., Fite-Georgel, P., Gallup, D., Johnson, T., Raguram, R., Wu, C., Jen, Y.-H., Dunn, E., Clipp, B., Lazebnik, S., Pollefeys, M.: Building Rome on a Cloudless Day. In: Daniilidis, K., Maragos, P., Paragios, N. (eds.) ECCV 2010, Part IV. LNCS, vol. 6314, pp. 368–381. Springer, Heidelberg (2010)
7. Snavely, N., Seitz, S.M., Szeliski, R.: Skeletal graphs for efficient structure from motion. In: CVPR (2008)
8. Brown, M., Lowe, D.G.: Unsupervised 3d object recognition and reconstruction in unordered datasets. In: 3DIM 2005: Proceedings of the Fifth International Conference on 3-D Digital Imaging and Modeling, pp. 56–63 (2005)
9. Lourakis, M.A., Argyros, A.: SBA: A Software Package for Generic Sparse Bundle Adjustment. ACM Trans. Math. Software 36, 1–30 (2009)
10. Ni, K., Steedly, D., Dellaert, F.: Out-of-core bundle adjustment for large-scale 3d reconstruction. In: International Conference on Computer Vision, ICCV (2007)
11. Lourakis, M.: levmar: Levenberg-marquardt nonlinear least squares algorithms in C/C++ (July 2004), http://www.ics.forth.gr/~lourakis/levmar/
12. Byröd, M., Åström, K.: Bundle adjustment using conjugate gradients with multiscale preconditioning. In: BMVC (2009)
13. Cao, J., Novstrup, K.A., Goyal, A., Midkiff, S.P., Caruthers, J.M.: A parallel levenberg-marquardt algorithm. In: ICS 2009: Proceedings of the 23rd International Conference on Supercomputing, pp. 450–459 (2009)
14. Triggs, B., McLauchlan, P.F., Hartley, R.I., Fitzgibbon, A.W.: Bundle adjustment - a modern synthesis. In: Proceedings of the International Workshop on Vision Algorithms: Theory and Practice, ICCV 1999 (2000)
15. Ranganathan, A.: The levenberg-marquardt algorithm. Technical Report Honda Research Institute (2004), http://www.ananth.in/docs/lmtut.pdf
16. Nielsen, H.: Damping parameter in marquardt's method. Technical Report hbn, Technical University of Denmark (1999), http://www.imm.dtu.dk/~hbn
17. Dongarra, J.: Compressed column storage (1995), http://netlib2.cs.utk.edu/linalg/html_templates/node92.html
18. Sengupta, S., Harris, M., Garland, M.: Efficient parallel scan algorithms for gpus. Technical report, NVIDIA Technical Report (2008)
19. Ltaief, H., Tomov, S., Nath, R., Dongarra, J.: Hybrid multicore cholesky factorization with multiple gpu accelerators. Technical report, University of Tennessee (2010)
20. Lindholm, E., Nickolls, J., Oberman, S., Montrym, J.: Nvidia tesla: A unified graphics and computing architecture. IEEE Micro 28, 39–55 (2008)
21. Snavely, N.: Notre dame dataset (2009), http://phototour.cs.washington.edu/datasets/