Multicore Bundle Adjustment

Changchang Wu[1], Sameer Agarwal[2], Brian Curless[1], and Steven M. Seitz[1,2]

[1]University of Washington [2]Google Inc.

Abstract

We present the design and implementation of new inexact Newton type Bundle Adjustment algorithms that exploit hardware parallelism for efficiently solving large scale 3D scene reconstruction problems. We explore the use of multicore CPU as well as multicore GPUs for this purpose. We show that overcoming the severe memory and bandwidth limitations of current generation GPUs not only leads to more space efficient algorithms, but also to surprising savings in runtime. Our CPU based system is up to ten times and our GPU based system is up to thirty times faster than the current state of the art methods [1], while maintaining comparable convergence behavior. The code and additional results are available at `http://grail.cs.washington.edu/projects/mcba`.

1. Introduction

The emergence of multi-core computers represents a fundamental shift, with major implications for the design of computer vision algorithms. Most computers sold today have a multi-core CPU with 2-16 cores and a GPU with anywhere from 4 to 128 cores. Exploiting this hardware parallelism will be key to the success and scalability of computer vision algorithms in the future. One way to exploit parallelism is to build our systems on low level libraries like BLAS and LAPACK which have already been optimized to use hardware parallelism; indeed, for certain tasks this is good enough. But to get the best performance one must build systems that exploit as much of the structure of the problem at hand as possible.

Recently there has been renewed interest in large scale Structure from Motion (SfM) systems, especially those aimed at community photo collections on the internet [6, 2]. A key component of these systems is bundle adjustment, the joint non-linear refinement of camera and point parameters, and one which can consume a significant amount of time for large problems.

In this paper we explore the use of CPU and GPU parallelism to achieve an order of magnitude or higher speedups over previously published systems. Our CPU based system is up to 10x and our GPU system is up to 30x faster than the current state of the art [1]. Ours is also the first GPU based system that can scale to the largest published bundle adjustment problems and opens the door to solving even larger problems. We do this by observing that inexact step Levenberg Mardquardt can be implemented without storing any (Hessian, Schur complement or Jacobian) matrices in memory. For single core systems this would translate into trading memory for time, but on GPUs, this leads to a surprising win in both space and time. Another surprise is that single precision arithmetic, when combined with suitable normalization techniques, gives results comparable to the ones obtained by a solver using double precision arithmetic. This results in further space and time savings.

The rest of the paper is organized as follow. We begin in Section 2 by discussing some of the theoretical background for bundle adjustment and the mathematical structure of our inexact step algorithms. In Section 3 we describe the guiding principles and techniques for implementing these algorithms on multicore CPU and GPUs. We deal with the issue of single precision in Section 4. In Section 5 we report the performance of our system, and we conclude with a discussion and directions for future work in Section 6.

2. Theoretical Background

Given a set of measured image feature locations and correspondences, the goal of bundle adjustment is to find 3D point positions and camera parameters that minimize the reprojection error [13]. This optimization problem is usually formulated as a non-linear least squares problem, where the error is the squared L_2 norm of the difference between the observed feature location and the projection of the corresponding 3D point on the image plane of the camera.

Let x be a vector of parameters and $f(x) = [f_1(x), \ldots, f_k(x)]$ be the vector of residuals/reprojection errors for a 3D reconstruction. Then the optimization problem we wish to solve is the non-linear least squares problem:

$$x^* = \arg\min_x \sum_{i=1}^{k} \|f_i(x)\|^2. \quad (1)$$

The Levenberg-Marquardt (LM) algorithm [11] is the most popular algorithm for solving non-linear least squares problems, and is the algorithm of choice for bundle adjustment. LM operates by solving a series of regularized linear approximations to the original nonlinear problem. Let $J(x)$ be the Jacobian of $f(x)$, then in each iteration LM solves a linear least squares problem of the form

$$\delta^* = \arg\min_{\delta} \|J(x)\delta + f(x)\|^2 + \lambda\|D(x)\delta\|^2 \,, \qquad (2)$$

and updates $x \leftarrow x + \delta^*$ if $\|f(x+\delta^*)\| < \|f(x)\|$. Here, $D(x)$ is a non-negative diagonal matrix, typically the square root of the diagonal of the matrix $J(x)^\top J(x)$ and λ is a non-negative parameter that controls the strength of regularization. The regularization is needed to ensure a convergent algorithm. LM updates the value of λ at each step based on how well the Jacobian $J(x)$ approximates $f(x)$ [11].

Solving (2) is equivalent to solving the *normal equations*

$$(J^T J + \lambda D^T D)\delta = -J^T f. \qquad (3)$$

where we have dropped the dependence on x for notational convenience. The matrix $H_\lambda = J^T J + \lambda\, D^T D$ is known as the augmented Hessian matrix.

In bundle adjustment, the parameter vector is typically organized as $x = [x_c; x_p]$, where x_c is the camera parameter vector and x_p the point parameter vector. Similarly for D, δ, and J, we use subscripts c and p to denote the camera part and the point part respectively . Let $U = J_c^T J_c$, $V = J_p^T J_p$, $U_\lambda = U + \lambda {D_c}^T D_c$, $V_\lambda = V + \lambda {D_p}^T D_p$, and $W = J_c^T J_p$, then (3) can be re-written as the block structured linear system

$$\begin{bmatrix} U_\lambda & W \\ W^T & V_\lambda \end{bmatrix} \begin{bmatrix} \delta_c \\ \delta_p \end{bmatrix} = - \begin{bmatrix} J_c^T f \\ J_p^T f \end{bmatrix}. \qquad (4)$$

It is worth noting that for most bundle adjustment problems, U_λ and V_λ are block diagonal matrices. This observation lies at the heart of the *Schur complement trick* used to solve this linear system efficiently, where, by applying Gaussian elimination to the point parameters, we obtain a linear system consisting of just the camera parameters:

$$(U_\lambda - W V_\lambda^{-1} W^T)\delta_c = -J_c^T f + W V_\lambda^{-1} J_p^T f. \qquad (5)$$

The matrix $S = U_\lambda - W V_\lambda^{-1} W^T$ is the Schur complement or the reduced camera matrix. Given the solution to (5), δ_p, the point parameters vector can be obtained by back-substitution:

$$\delta_p = -V_\lambda^{-1}(J_p^T f + W^T \delta_c). \qquad (6)$$

Since S is symmetric positive-definite, Cholesky factorization is the method of choice for solving (5). Factorization methods, even ones like CHOLMOD [5] which exploit the sparsity structure of S are space and time intensive and can be prohibitively expensive for large problems.

Recently, the use of the Preconditioned Conjugate Gradients algorithm for solving these equations has drawn the attention of the computer vision community [1, 4, 9]. In particular [1] has shown that state of the art performance can be achieved by combining an Inexact Step LM algorithm and Preconditioned Conjugate Gradients with some simple and computationally cheap preconditioners. The algorithms presented in this paper further extend this line of work to the case of multicore processors.

2.1. Preconditioned Conjugate Gradients

Conjugate Gradients (CG) is an iterative approach for solving symmetric positive-definite linear systems [12]. One of the key advantages of CG when solving a linear system $Ax = b$ is that the only way it accesses the matrix A is through the matrix-vector product Ap for some vector p. Thus it is possible to solve $Ax = b$ without ever explicitly forming A in memory.

We will consider two approaches for solving (3). The first is the so called Conjugate Gradients for Least Squares (CGLS) algorithm applied to the augmented hessian matrix. Assuming that the Jacobian matrix J is available, the product $H_\lambda p$ is easily implemented in terms of the Jacobian as

$$H_\lambda q = J^T(Jq) + \lambda(D^T D)q. \qquad (7)$$

Note that the brackets are used to break up the product $H_\lambda p$ into a series of simpler matrix-vector products involving the Jacobian J and diagonal matrix $D^T D$.

To improve the rate of convergence of CG, we precondition using the block Jacobi preconditioner

$$M_\lambda = \begin{bmatrix} U_\lambda & 0 \\ 0 & V_\lambda \end{bmatrix}. \qquad (8)$$

As we noted earlier, U_λ and V_λ are block diagonal. They are easily computed from J and inverted in linear time and space. We will refer to this preconditioned CG approach using simple matrix-vector products as the *implicit-hessian* algorithm.

The second approach we will consider is the *implicit-schur* algorithm where we run CG on the Schur complement S [1, 9]. It can be shown that S is better conditioned than H_λ and therefore we expect CG to converge more quickly too. In [1], the authors show how to evaluate Sp_c without forming S explicitly. This is done by exploiting the structure of the Schur complement as

$$Sp_c = U_\lambda p_c - W(V_\lambda^{-1}(W^T p_c)). \qquad (9)$$

The authors recommend the use of $M_\lambda = U_\lambda$ as a preconditioner for these iterations. But we can go one step further; in the same way that running CG on the augmented hessian

does not require forming the matrix H_λ, we can run CG on the Schur complement S without forming the expensive sub-matrices W and W^T of H_λ explicitly:

$$Sq_c = J_c^T(J_c q_c - J_p\left(V_\lambda^{-1}(J_p^T\left(J_c q_c\right)\right))) + \lambda D_c^T D_c q_c. \tag{10}$$

The *implicit-hessian* and *implicit-schur* algorithms require essentially the same amount of memory, consists of the same set of Jacobian-matrix-vector multiplications, and have similar computation cost per iteration. Equations (7) and (10) not only allow CG to run without explicitly forming the augmented Hessian and the Schur complement, they have the additional benefit of breaking down the task of multiplying a vector with a complex block sparse matrix into a series of simpler, more easily parallelizable matrix-vector products. Further, they open up the possibility of a completely matrix free algorithm where we don't even store the matrix J in memory and instead compute its entries on the fly as needed. We will explore this direction in the following sections.

3. Parallel Implementation

Before we discuss the design and implementation of the two iterative solvers, let us consider the major sources of computational expense in an LM algorithm that uses an iterative linear solver.

1. Reprojection error computation: f;
2. Jacobian matrix computation: $J = [J_c,\ J_p]$;
3. Construction of the preconditioner: M_λ^{-1};
4. Matrix-vector Multiplication $Jx = J_c x_c + J_p x_p$;
5. Matrix-vector Multiplication $J^T y\ =\ [J_c^T y,\ J_p^T y]$;
6. Matrix-vector Multiplication $M_\lambda^{-1} v$.

Note that $\mathrm{diag}(J^T J)$ for damping is a byproduct of M_λ^{-1}.

To guide the development of our algorithms, we evaluated the amount of time spent in each of the major functions for the *implicit-hessian* based LM. (We will refer to procedures such as computing f or Jx as "functions.") Table 1 shows the results. The computation of $J^T x$ and Jy consume the most amount of time as they are called in every iteration of the CG algorithm. The computation of the Jacobian J and the preconditioner M_λ^{-1} though expensive is not a significant expense as they are done only once per LM iteration. Thus, most of our effort is focused on optimizing the two sparse matrix-vector multiplication: Jx and $J^T y$.

Observe that the aforementioned functions can be parallelized by dividing the computation task into camera-wise, point-wise, and measurement-wise threads. Specifically, computing f, J and Jx consists of performing per-measurement tasks, and $J^T y$, M_λ^{-1} and $M_\lambda^{-1} v$ consist of per-camera and per-point tasks. Compared to forming H_λ or S, these functions are much easier to parallelize. We additionally parallelized all the pure vector functions, which is easily done and will not be discussed.

	f	J	M_λ^{-1}	Jx	$J^T y$	$M_\lambda^{-1} v$
Dubrovnik	1.3%	6.1%	3.1%	**28%**	**54%**	2.8%
Venice	0.7%	2.8%	1.6%	**30%**	**57%**	2.7%

Table 1. Percentage of time spent on the major functions in a single-threaded CPU implementation during 50 LM iterations (single-precision, SSE-enabled, implicit-Hessian). We use the largest two final models from [1]. The total time for the two models are 778 and 4711 seconds respectively. The fraction of time not accounted for by the functions listed above was spent in pure vector operations.

3.1. Related Work

In recent years, GPU-based parallelization has drawn a lot research attention. This includes general purpose libraries like nVidia's Sparse Matrix-Vector Multiplication [3] and Li and Saad's work on various GPU-based PCG algorithms and preconditioning techniques [10]. While it is possible to build a bundle adjustment system using these general purpose libraries, better performance is obtained by building a system that exploits the structure of the problem as much as possible.

In the field of computer vision, many algorithms have been ported to the GPU with significant gains in speed. This includes feature detection, feature matching, and stereo reconstruction, etc. An impressive demonstration is the multi-GPU-based fast SfM system described in [6]. It is worth noting though that the bundle adjustment step in this system was still performed using single-threaded CPUs.

Given the constraints of the GPU programming model, it is not trivial to get bundle adjustment algorithms to run on the GPUs. Recently, Gupta et al [8] took a hybrid approach to run overlaping computations on GPU and CPU, where the Hessian matrices and Schur complements are constructed on GPU. This is not practical for large problems (thousands of images or more), especially for community photo collections that have dense Schur complements. Even state of the art workstation GPUs have a small amount of RAM, so storing the Schur complements (even the Hessian matrices or the Jacobians), may not be feasible. Even with enough memory, the construction of Schur complements would still be too expensive for large problems.

3.2. Parallelization Principles

Even though the underlying hardware architecture of multicore CPUs and GPUs are quite different, there are some common themes for developing high performance systems on multicore systems. Primary amongst them is the mismatch between processor speed and rate at which

data can be fetched from memory. This is already a consideration for single core systems; in a multicore system the problem gets even worse. A further complication is that in a multithreaded system multiple threads may want to write to the same memory location. Thus optimal performance requires that we

1. Maximize the processor occupancy,
2. Optimize memory access to reduce contention.

There are a number of ways in which we can solve these problems, and at this point there are no clear theoretical guidelines for choosing one over the other. Part of the reason is that even though different processors expose similar programming interfaces, the underlying hardware implementation can be quite different with widely varying performance for the same piece of code. Therefore, we implemented a number of different variations of each technique and profiled each one of them. The methods reported in this paper are the combinations that performed best in our experiments. As hardware evolves, the specific recommendations made in this paper may not give the best performance, but we hope that the reader will benefit from the general principles for organizing their computations described here.

3.2.1 Matrix Storage

A key design decision affecting performance is how various matrices are stored in RAM. For CPU and GPU, we use the Block Compressed Sparse Row (BCSR) format to store J_c, J_p and J_c^T when needed. We sort the observations by their 3D point id, which means that J_p is a block diagonal matrix, and J_p and J_p^T have the same BCSR representation and thus there is no need to store J_p^T separately.

3.3. CPU Parallelization

Multi-core CPUs typically can run 10s of threads simultaneously, with multi-level cached access to a large amount of RAM.

3.3.1 Maximizing Processor Occupancy

The SSE functionality on modern CPUs increases processor throughput by operating on 4 floats or 2 doubles with a single instruction, which makes it easy to accelerate simple vector operations, like addition, multiplication, norm, and dot product. To apply SSE to our relatively complicated functions, we align the number of camera parameters by 4, such that 8-vec storage is used even when the camera model has only 7 parameters. Consequently, the multiplication and addition of camera derivatives can be carried out by SSE instructions. In addition, we allocate aligned data to employ the fast aligned memory load and store. Table 2 demonstrates the significant speedup we achieve with SSE.

Operation	float	float SSE	double	double SSE
f	0.64	N/A	0.75	N/A
J	4.00	3.46	5.44	5.10
M_λ^{-1}	4.85	1.51	5.20	2.46
Jx	0.82	0.67	1.07	0.88
$J^T y$	3.02	1.32	3.62	2.11
$M_\lambda^{-1} v$	0.06	0.06	0.07	0.07

Table 2. Time consumed (in seconds) for the Venice final model as floating point precision and SSE usage are varied. SSE instructions lead to significant speedups for both single and double precision, particularly the M_λ^{-1} and $J^T y$ operations. As expected single precision operations with their lighter memory requirements are faster than double precision counterparts.

We divide large computation tasks into a number of threads that are suitable for the number of CPU cores, so that all the computation power is utilized. For instance, with 16-core processors, we would normally run at least 16 or 32 threads for the intensive tasks. Table 3 demonstrates the speedup we achieve by multi-threading on a computer that has dual quad-core Xenon E5520 CPUs.

3.3.2 Optimizing Memory Access Patterns

To reduce the contention between multiple threads needing access to the same indexing information, we store the necessary indexing structures to allow each thread to run independently. For example, $J^T y$ runs camera-wise threads to access all the projections of each camera, in contrast to running measurement-wise threads where different measurements for the same camera need exclusive access to its camera block in the output vector.

As needed, we also store shuffled copies of data (if possible) to make frequently-called functions have continuous memory access patterns. For example, $J^T y$ needs to access the Jacobian block of all the measurements seen by a camera. It is better to store the extra copy J_c^T in its row block order if possible rather than the indexed access of blocks in J_c. Table 5 shows that $J^T y$ using J_c^T is 40% faster than using indexed access of J_c.

3.4. GPU Parallelization

In contrast with CPUs, current GPUs can have *hundreds* of processing cores, but they have much less onboard RAM, and we must be more careful about contention between different cores for memory bandwidth.

3.4.1 Maximizing Processor Occupancy

A GPU is by definition SIMD. nVidia's CUDA organizes threads into thread blocks, which is further divided into 32-thread units called *warps*, which are essentially SIMD.

		Single-thread	Multi-thread	GPU
Dubrovnik Final	f	0.18	0.03	0.012
	J	1.22	0.14	0.038
	M_λ^{-1}	0.53	0.08	0.037
	Jx	0.19	0.04	0.016
	$J^T y$	0.38	0.08	0.035
	$M_\lambda^{-1} v$	0.02	0.01	0.002
	CG-*hessian*	5.90	1.51	0.57
	LM-*hessian*	7.92	1.85 **(4.3x)**	0.70 **(11x)**
	CG-*schur*	7.59	1.74	0.68
	LM-*schur*	7.77	2.06 **(3.8x)**	0.81 **(9.6x)**
Venice Final	f	0.64	0.09	0.045
	J	3.46	0.47	0.048
	M_λ^{-1}	1.51	0.24	0.190
	Jx	0.67	0.14	0.056
	$J^T y$	1.32	0.24	0.180
	$M_\lambda^{-1} v$	0.06	0.016	0.007
	CG-*hessian*	21.9	5.29	2.07
	LM-*hessian*	29.8	6.31 **(4.7x)**	2.36 **(13x)**
	CG-*schur*	22.9	5.86	2.69
	LM-*schur*	35.6	5.85 **(6.1x)**	2.98 **(12x)**

Table 3. Time(in seconds) comparison of CPU single-thread, CPU multi-thread and GPU. Both CPU implementations are SSE-optimized. CG-timing is obtained with 10 CG iterations. The LM-timing gives the time for a full LM iteration including linear system solver, parameter update and step validation. For Venice Final, J_c, J_c^T are not stored on GPU due to insufficient memory and were computed on the fly. The **x** next the LM timing gives the speed up compared with single-threaded CPU implementation regardless of the precision difference.

As is the case with SSE, achieving peak memory bandwidth on the GPU requires coalesced memory access. Thus, like our CPU code, we align the camera parameters and point parameters by 4, so that the processing of camera parameters and point parameters are aligned with the warp size. In cases where we are unable to have coalesced memory fetching, we apply the common technique of using the GPU texture memory as a cache.

Another organizing principle is that simple programs lead to higher occupancy on GPUs. Instead of having one thread per camera or per point, we try to map the problem such that there is one thread per parameter or one half-warp (16 threads) per camera, etc. The mapping additionally enables different threads of the same camera/point to share data on the fast shared memory. For example, we invert the 8×8 camera diagonal blocks matrices using 8 threads.

Speed	CSR	COO	HYB	Our GPU	Our CPU
Jx	0.130	0.060	0.019	0.016	0.044
$J^T y$	0.111	0.096	0.092	0.035	0.076

Table 4. GPU based Sparse Matrix Vector Multiply(SpMV). We compare our custom SpMV operations with the ones developed by Bell & Garland [3] on the Dubrovnik Final model. Time is reported in seconds. Please refer to their paper for the details of CSR, COO and HYB. Our methods give significant speedup on $J^T y$, which has varying number of non-zero elements per row. Our Jx is only slightly faster because we store J_c and J_p separately. Note that their method cannot handle the Venice Final problem due to insufficient memory.

3.4.2 Optimizing Memory Access Pattern

Similar to the CPU case, we store all of the matrices J_c^T, J_c and J_p in BCSR format when needed, allowing functions Jx, $J^T y$, and M_λ^{-1} to fetch Jacobian blocks continuously. By exploiting the block structures with texture fetching and shared memory, our Sparse Matrix-Vector multiplication (SpMV) outperforms the work of nVidia's Bell and Garland [3] particularly for $J^T y$ (Table 4), thus re-enforcing the point that for maximum performance we must exploit problem structure as much as possible. Also, like the CPU we avoid the dependency between different thread blocks by computing and explicitly storing the indices required for gathering operations across various threads.

As we mentioned earlier, GPUs typically have access to a much smaller amount of RAM as compared to CPUs. Thus it is important to minimize the memory usage of GPU algorithms. While the use of the *implicit-hessian* and the *implicit-schur* gets rid of the storage for the augmented Hessian and Schur complement matrices, the storage of the Jacobian J is still a substantial expense. To get around this we experimented with matrix-free versions of Jx, $J^T y$ and M_λ^{-1}, where the required entries of J and M_λ were computed on the fly as needed. This leads to a substantial reduction in memory usage. The Dubrovnik Final and Venice Final models, which are the largest publicly available models, consume only 543MB and 1797MB respectively. A GPU like the nVidia Tesla C1060 comes with 4096MB of on-board RAM, and thus can easily handle problems twice the size of the Venice Final model.

Surprisingly, even though the original decision to implement the matrix-free versions of Jx, $J^T y$ and M_λ^{-1} was to save space, it turns out that they can outperform the matrix based version in terms of time also. Table 5 compares the various versions of these functions as the underlying storage configuration is varied.

The reason for this unexpected behavior is that the enormous amounts of thread parallelism on the GPU leads to contention, and many of the threads idle while waiting for

Function	Store & Read	Compute On-the-Fly	GPU	CPU
Jx	$J_c,\ J_p$		0.023	0.044
		$J_c,\ J_p$	0.016	0.078
$J^T y$	$J_c^T,\ J_p$		0.035	0.076
	$J_c,\ J_p$		0.055	0.11
	J_p	J_c	0.032	0.16
		$J_c,\ J_p$	0.047	0.52 (*)
M_λ^{-1}	$J_c^T,\ J_p$		0.037	0.077
	$J_c,\ J_p$		0.052	NA
	J_p	J_c	0.025	0.15
		$J_c,\ J_p$	0.041	1.34 (*)

Table 5. Time consumed (in seconds) for the Dubrovnik final model as a function of different storage configurations. The CPU functions are multi-threaded except for the two marked with (*). Note that it is possible to achieve higher speed by computing some derivatives on the fly, such as the GPU matrix-free version of Jx.

Systems	Trafalgar		Dubrovnik	
	Schur	Hessian	Schur	Hessian
Tesla C1060, Linux	0.165	0.146	0.81	0.70
8800 Ultra, Linux	0.188	0.168	1.51	1.06
8800 Ultra, Win7	0.204	0.180	1.53	1.07
8800 GTX, Winxp	0.220	0.194	1.81	1.28
GTX 295, Winxp	0.167	0.150	1.08	0.86

Table 6. Speed (seconds) of one full LM iteration on a few systems for the Trafalgar Final and the Dubrovnik Final problem (without tuning of GPU settings). The LM iteration is evaluated with a 10-iteration CG solver. The Dubrovnik problem runs without storing any Jacobians on the 4 non-Tesla systems.

data from the memory. Therefore, it is better to read a small mount of memory that can be well cached, and recompute the results. After many experiments, we settled on the following priority: $J_p > J_c^T > J_c$, to guide the memory allocation. In the CPU world, matrix-free algorithms translate into trading time for memory savings, but in the GPU world, the trade-off vanishes in some cases, and both time and memory can be saved.

Another nice property of our implementation is that once the calculation of the LM step has been offloaded to the GPU, the memory transfers between GPU and CPU are reduced to a few vectors and scalars, essentially eliminating the CPU to GPU memory transfer bottleneck.

To get the best performance out of the GPU code, we empirically determined the best parallelization strategies like the thread block size of each function. Given different problem sizes or different GPUs, the storage configuration that give the highest speed can vary. For example, unlike the order in Table 5, we observe that for small and medium sized problems, memory contention is not an issue, and it is actually faster to compute $J^T y$ by precomputing and storing J_p and J_c in GPU memory.

Table 6 demonstrates the performance for the two algorithms on several high-end commodity GPUs .

4. Precision

Most bundle adjustment algorithms used double precision arithmetic due to its large numerical range and high accuracy. However we have found that with some care we can use single precision arithmetic without a significant loss in numerical performance and a substantial gain in runtime performance. The use of single precision not only reduces the memory usage of our system by half, but a can be seen in Table 2, it also increases the speed of memory transfers between the memory and the floating point units leading to higher throughput for both CPUs and GPUs. To deal with the reduced precision and numerical range of single precision floats we do two things.

First, we apply a pre-processing data normalization step to improve the distribution of the Jacobian values. It is easy to see that the Jacobian of a reprojection error term scales as F/z^2, where F is the focal length of the camera and z is the depth of the 3D point in that camera. To better utilize the range of single-precision, we scale the system to bring the distribution of F and $1/z$ to the middle of the valid value range, so as to better condition the Jacobian. Given a normalization parameter C_n (we use 0.5), the data normalization steps are

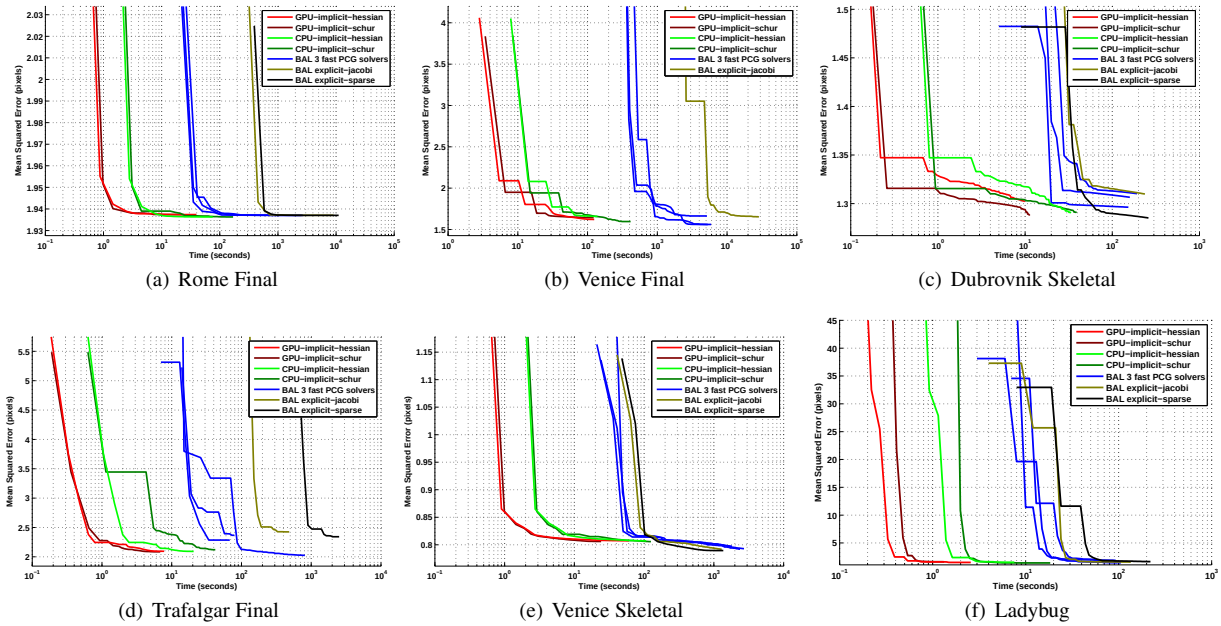
1. Find $F_m = \text{median}\{F\}$ and $z_m = \text{median}\{z\}$,
2. Scale all focal lengths and measurements by $\frac{C_n}{F_m}$,
3. Scale all camera translations and 3D points by $\frac{1}{z_m C_n}$.

Second, we follow [1] and scale the columns of the Jacobians by the square root the diagonal of the initial augmented Hessian H_0.

5. Experiments

In this section, we evaluate the optimization performance of the *implicit-hessian* and the *implicit-schur* algorithms. For both algorithms, we evaluate the performance of GPU parallelization and CPU multi-threading. Only single-precision floating point are included in the paper. As expected, the double precision implementation is slightly slower than the single-precision implementation. Results using the double precision implementation can be found on the project website.

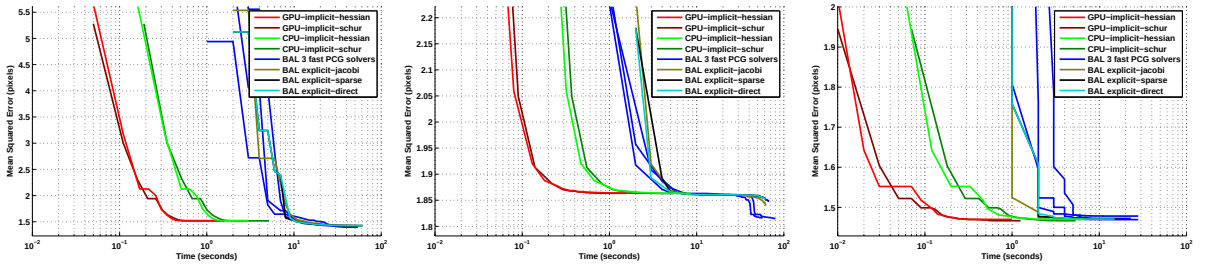
We compare the speed and convergence of our system with the state of the art system of Agarwal et al. [1]. Agarwal et al. describe six different algorithms, we compare our system to all of them except for the dense-factorization



(a) Rome Final (b) Venice Final (c) Dubrovnik Skeletal

(d) Trafalgar Final (e) Venice Skeletal (f) Ladybug

Figure 1. Runtimes for large scale problems. Our GPU solvers exhibit competitive convergence and a speedup of about 10x-30x against all the BAL solvers. Our method perform equally well for sparse reconstruction problems (e.g. Ladybug is a video sequence). Our multi-threaded CPU solvers achieves about $\frac{1}{3}$ the speed of our GPU implementation.



(a) $m = 52, n = 64063, k = 347143$ (b) $m = 88, n = 64298, k = 383937$ (c) $m = 138, n = 19878, k = 85217$

Figure 2. Runtime for small problems where m, n and k the number of cameras, points and measurements. Our bundle adjustments achieve consistent speedup except but sometimes fail to reach the same solution as the BAL's double precision solvers, e.g. Figure (b).

algorithm that does not fit large-scale problem. The remaining five algorithms are *explicit-sparse* which computes a sparse factorization of the Schur complement *S*, *explicit-jacobi* which explicitly computes *S* and runs PCG on it using a block-Jaocbi precondoitioner, *normal-jacobi* which runs PCG on the H_λ with M_λ as the preconditioner, *implicit-jacobi* and *implicit-ssor*, which compute the product Sp_c by using (9). We refer to these algorithm as BAL (Bundle Adjustment in the Large). BAL algorithms use double precision arithmetic everywhere and do not offer a choice to use single precision arithmetic.

For each BAL algorithm, we run for a maximum 50 LM iterations. We run our own system for a 100 iterations in order to compensate for its lower precision. As we will see, even with this extra computational burden, our system significantly outperforms the BAL algorithms. The rest of the parameters of the LM and CG algorithms were identical to the ones used by BAL.

We use the datasets provided by [1] for our evaluation. The datasets include the skeletal and final models reconstructed from community photo collections on the Internet, which typically have relatively dense blocks in Schur complement. 3D models reconstructed from Ladybug street-side image are also presented for evaluating the performance on the sparser problems.

All experiments were conducted on a workstation that has: dual Quad-core 2.27Ghz CPUS with 2x hyper-threading; Tesla C1060 Graphic card with 4GB graphic memory; 64-bit Linux OS. Our CPU multi-threading is done through low-level threading functions, and our GPU code is implemented with CUDA. The thread settings were profiled for large problems and then kept constant.

5.1. Results

Overall, we observe (Figures 1 and 2), a speedup between 10x-30x with the GPU implementation of our algorithms, and a speedup of about 5X-10X with the multi-threaded CPU implementation. The speedup is also consistent across various sizes of problems, and varying sparsity of problems.

Despite the fact that we focused our system design and optimization efforts on large scale problems, we found that our system performed equally well on small and medium sized problems. Figure 2 shows the comparison with all the BAL solvers on three small problems. Note that the speed ratios between the GPU version and the CPU version are highest for medium size problems. It appears that as size of the problem grows larger, the cost of accessing GPU memory increases. This is due to the limited size of the global texture cache on the GPUs.

As shown in Table 3, the cost for an LM step is more expensive for *implicit-schur* than *implicit-hessian*, but *implicit-schur* makes up for this with better convergence behavior per LM iteration because of its better preconditioning behavior. Finally, even though we report results on a high end Tesla GPU, as can be seen from Table 6, similar performance can be expected on consumer grade GPUs.

6. Conclusions

In this paper we presented multicore solutions to the problem of bundle adjustment that run on currently available CPUs and GPUs. These systems deliver a 10x to 30x boost in speed over existing systems while reducing the amount of memory used. This is done by carefully restructuring the matrix vector product used in the PCG iterations into easily parallelizable operations. This restructuring also opens the door to a matrix free implementation which leads to substantial reductions in the memory consumption as well as execution time. We also showed that single precision arithmetic when combined with appropriate normalization gives numerical performance comparable to double precision based solvers while further reducing the memory and time cost. The resulting system enabled running the largest bundle adjustment problems to date (from the Rome-in-a-day effort [2]) on a single GPU.

While the problem addressed in this paper is bundle adjustment, we believe that the strategies presented here can be applied to other large scale optimization problems in computer vision and elsewhere.

In the future, we would like to further improve the numeric stability of our single-precision solvers and experiment with double-precision arithmetic on GPU. We would also like to port this work to non nVidia based platforms.

Finally, we put a large amount of work into the extensive benchmarking needed to tune the various GPU compute kernels. The lack of documentation about the internal memory access hardware of GPUs makes this kind of empirical approach necessary. An interesting direction for future work is a framework along the lines of FFTW [7] or ATLAS [14] for automatically tuning the parameters of the system either at compile time or at runtime.

Acknowledgements We thank Noah Snavely for providing the datasets. The authors would also like to acknowledge the discussions with Christopher Zach. This work was supported in part by National Science Foundation grants IIS-0811878 and IIS-0963657, SPAWAR, the University of Washington Animation Research Labs, Intel, Microsoft, and Google.

References

[1] S. Agarwal, N. Snavely, S. Seitz, and R. Szeliski. Bundle adjustment in the large. In *ECCV10*, pages II: 29–42, 2010. 3057, 3058, 3059, 3062, 3063

[2] S. Agarwal, N. Snavely, I. Simon, S. M. Seitz, and R. Szeliski. Building Rome in a day. In *ICCV*, 2009. 3057, 3064

[3] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC '09*, pages 1–11, 2009. 3059, 3061

[4] M. Byrod and K. Astrom. Conjugate gradient bundle adjustment. In *ECCV10*, pages II: 114–127, 2010. 3058

[5] Y. Chen, T. Davis, W. Hager, and S. Rajamanickam. Algorithm 887: CHOLMOD, Supernodal Sparse Cholesky Factorization and Update/Downdate. *TOMS*, 35(3), 2008. 3058

[6] J. Frahm, P. Fite Georgel, D. Gallup, T. Johnson, R. Raguram, C. Wu, Y. Jen, E. Dunn, B. Clipp, S. Lazebnik, and M. Pollefeys. Building rome on a cloudless day. In *ECCV10*, pages IV: 368–381, 2010. 3057, 3059

[7] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proc. of the IEEE*, 93(2):216–231, 2005. 3064

[8] S. Gupta, S. Choudhary, and P.J.Narayanan. Practical time bundle adjustment for 3d reconstruction on gpu. In *ECCV Workshop on Computer Vision on GPUs*, 2010. 3059

[9] Y. Jeong, D. Nister, D. Steedly, R. Szeliski, and I. Kweon. Pushing the envelope of modern methods for bundle adjustment. In *CVPR10*, pages 1474–1481, 2010. 3058

[10] R. Li and Y. Saad. GPU-accelerated preconditioned iterative linear solvers,. Technical Report UMSI-2010-xx3, Minnesota Supercomputer Institute, University of Minnesota, 2010. 3059

[11] J. Nocedal and S. Wright. *Numerical optimization*. Springer, 2000. 3058

[12] L. Trefethen and D. Bau. *Numerical linear algebra*. SIAM, 1997. 3058

[13] B. Triggs, P. McLauchlan, H. R.I, and A. Fitzgibbon. Bundle Adjustment - A modern synthesis. In *Vision Algorithms'99*, pages 298–372, 1999. 3057

[14] R. C. Whaley and A. Petitet. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience*, 35(2):101–121, February 2005. 3064