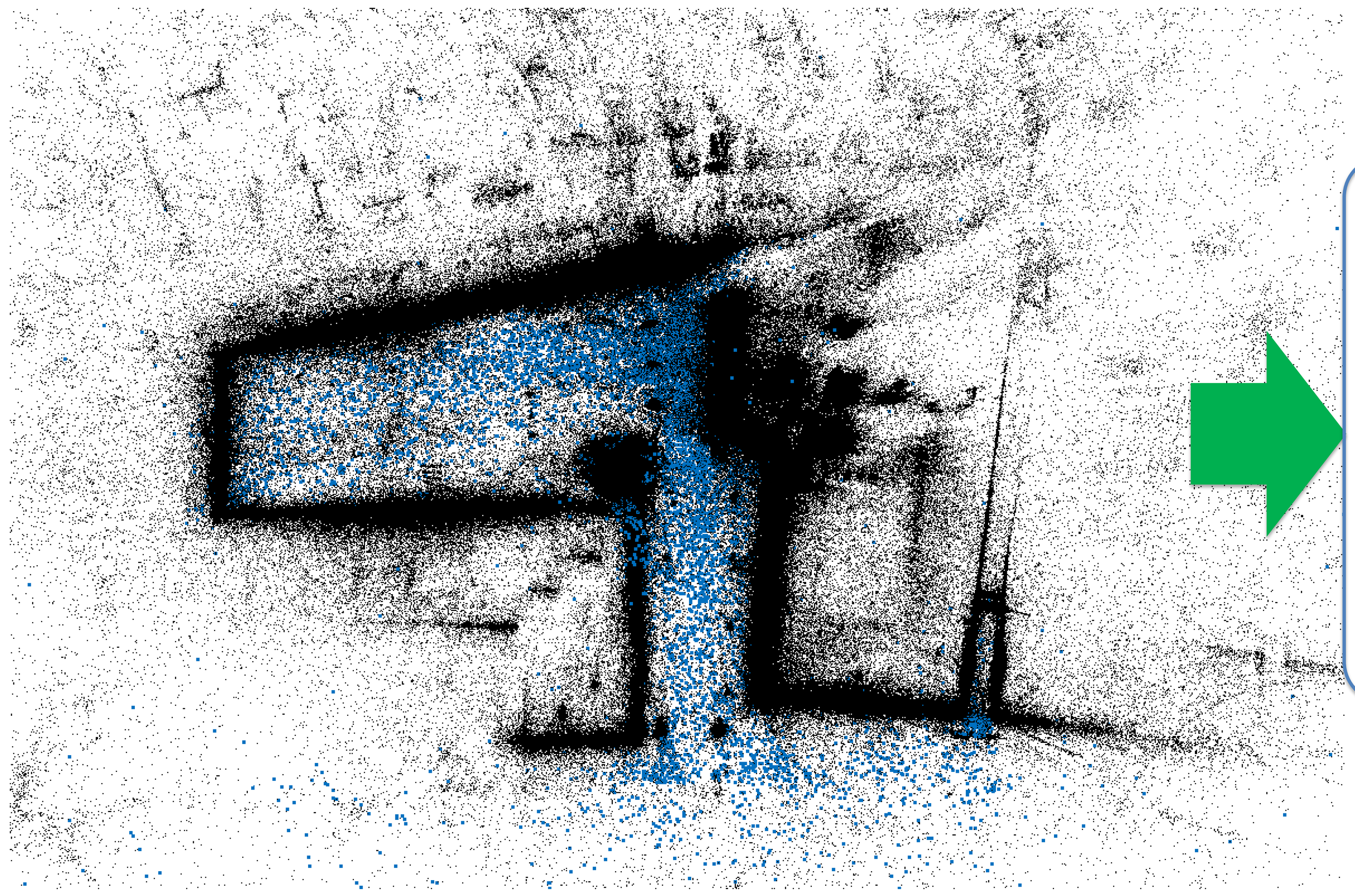
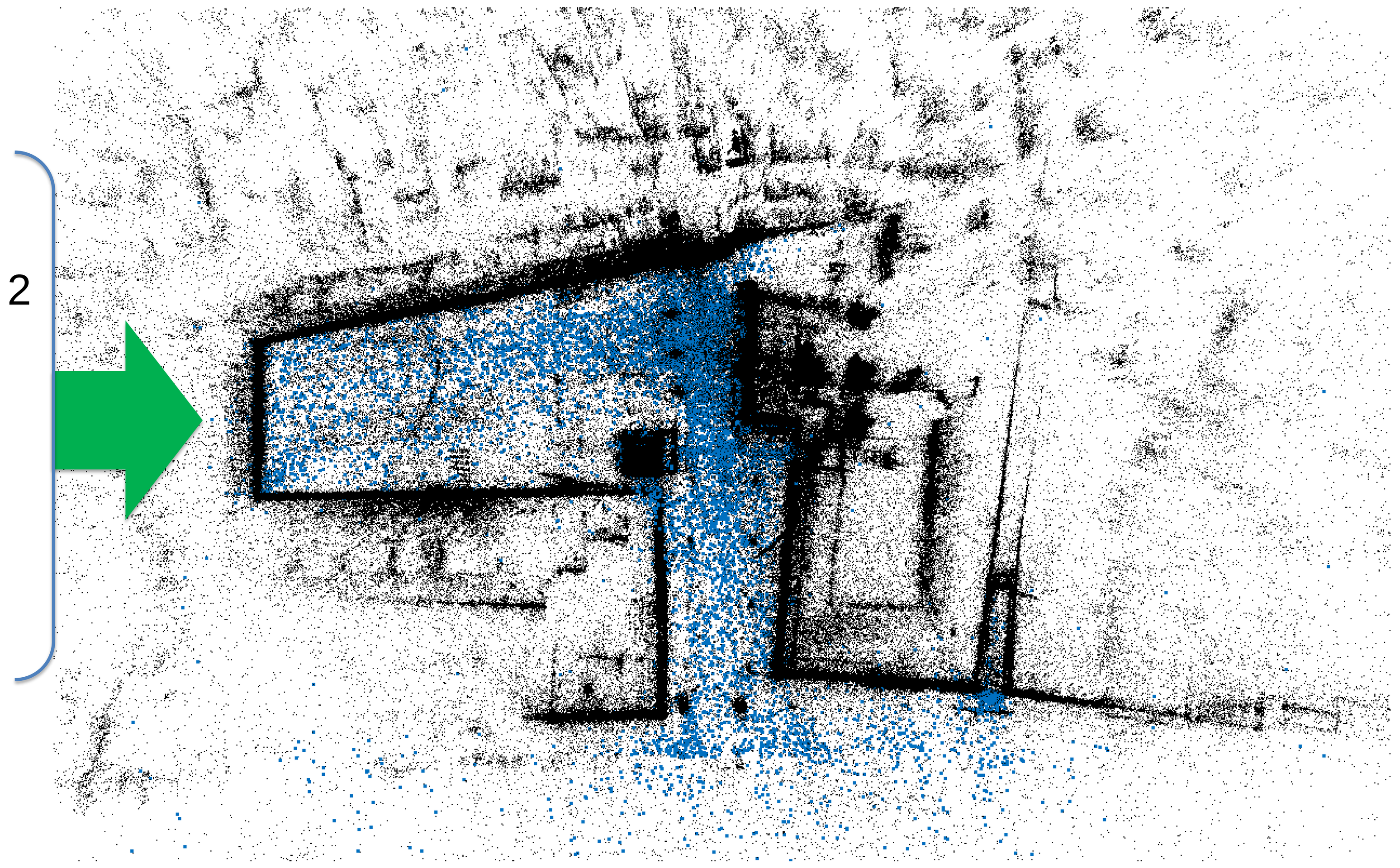


Multicore Bundle Adjustment

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

[1] University of Washington at Seattle, [2] Google Inc.



14K cameras, 4.5M points and 30M measurements in 2 minutes!

Code available at http://grail.cs.washington.edu/projects/mcba/

Our Multicore Solution

- Problem restructuring to make bundle adjustment easily parallelizable.
- **10x-30x** Speedup on nVidia Tesla C1060 GPU.
- **5x-10x** Speedup on Dual Intel Xenon E5520 (16 cores).
- Up to **80 %** reduction in memory usage.

Bundle Adjustment

Bundle adjustment is the joint non-linear refinement of camera and point parameters. Levenberg-Marquardt (LM) is the most popular method for solving bundle adjustment. Let J be the Jacobian, each step of LM solves a regularized linear least squares problem:

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

which is equivalent to solving the normal equations:

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

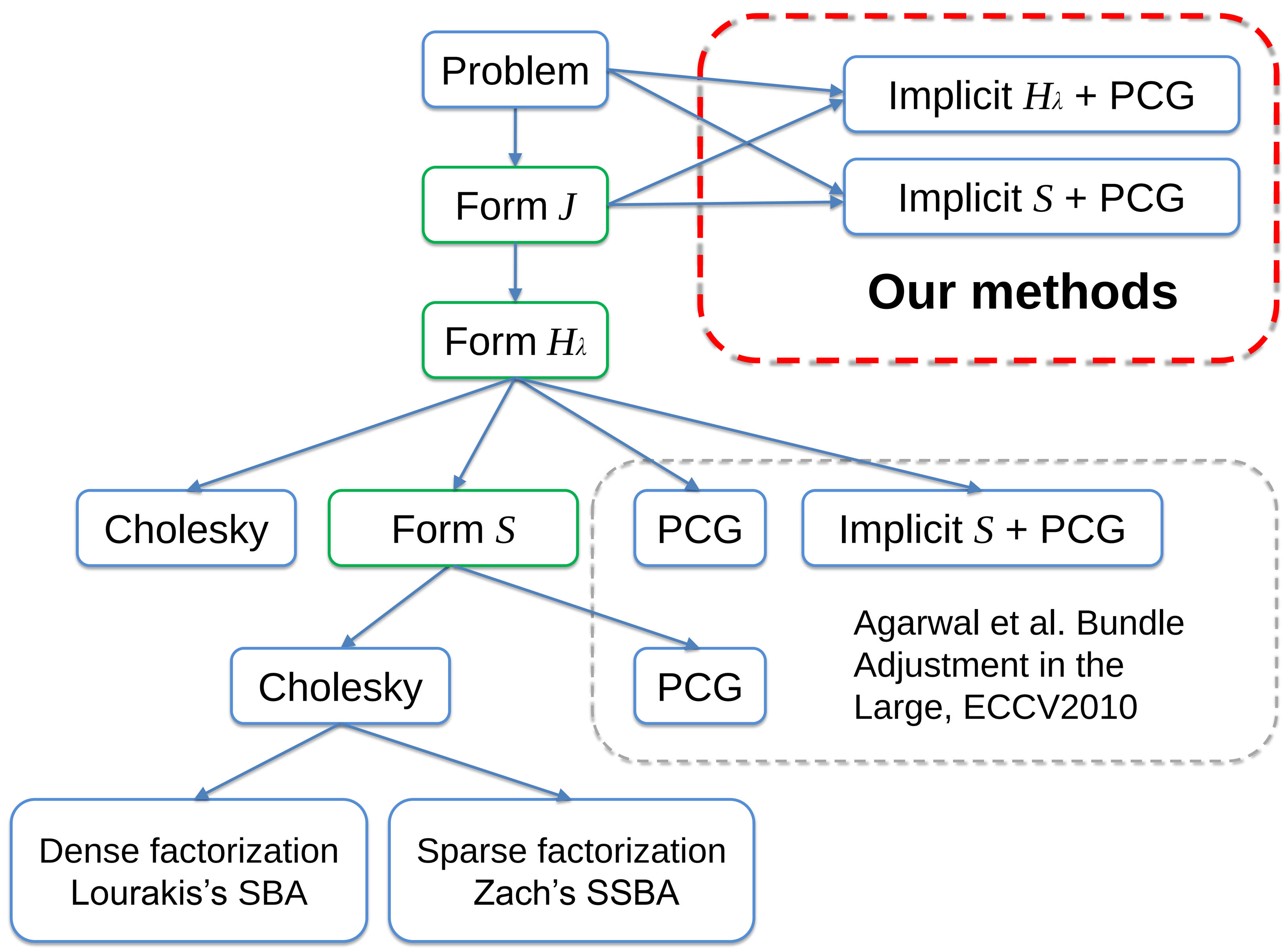
where $H_\lambda = J^T J + \lambda D^T D$ is called the augmented Hessian Matrix.

The parameters consist of the camera part and the point part ($\delta = [\delta_c; \delta_p]$, $J = [J_c, \ J_p]$, *etc.*) and most methods first solve the reduced camera system

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

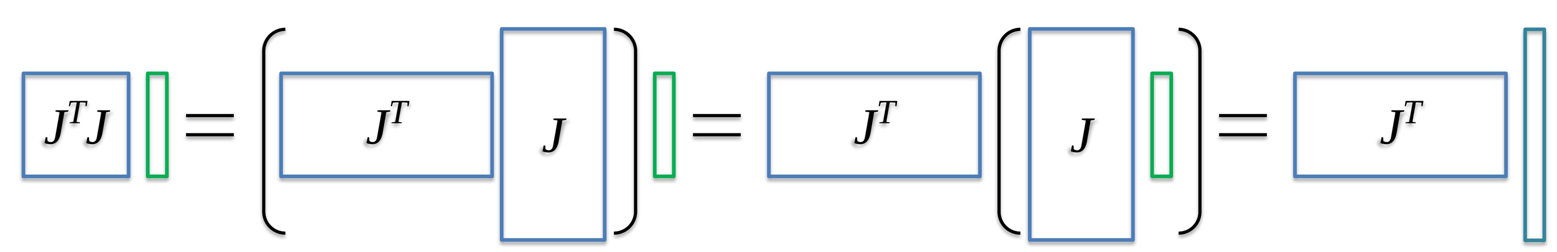
where $S = U_\lambda - W V_\lambda^{-1} W^T$ is called the Schur complement,

$U_\lambda = J_c^T J_c + \lambda D_c^T D_c, V_\lambda = J_p^T J_p + \lambda D_p^T D_p$ and $W = J_c^T J_p$.



Problem Restructuring → Fine-grained Parallelization; On-the-fly Jacobian

‣ Exploit associativity of multiplication to eliminate matrix products



Using the augmented Hessian matrix without forming it

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

Using the Schur complement without forming it or forming the Hessian

$$S q_c = J_c^T(J_c q_c - J_p(V_\lambda^{-1}(J_p^T(J_c q_c)))) + \lambda D_c^T D_c q_c$$

‣ Map problem structure to use both multi-threading and SIMD

- Map computation loops to threads on compute cores
  - *A few threads on CPU; many threads on GPU*
- Align parameter size to 4 and employ SIMD arithmetic
  - *CPU SSE operates on 4 floats; CUDA Warp operates on 32 floats*

Naïve CPU
Non-SSE, Single-threaded

CPU SSE Single-threaded

Jx	1.2X
J^Ty	2.3X
LM	2.0X

CPU SSE Multi-threaded

Jx	5.9X
J^Ty	12X
LM	**10X**

GPU (Not storing Jc)

Jx	15X
J^Ty	23X
LM	**25X**

Venice Final : 14K Cameras, 4.5M points, and 30M Measurements.
(LM is profiled with a fixed number of 10 CG iterations).

‣ Use single-precision arithmetic with proper normalization

- Normalize parameters to precondition the distribution of Jacobians.
- Maintain accuracy while achieving higher throughput.

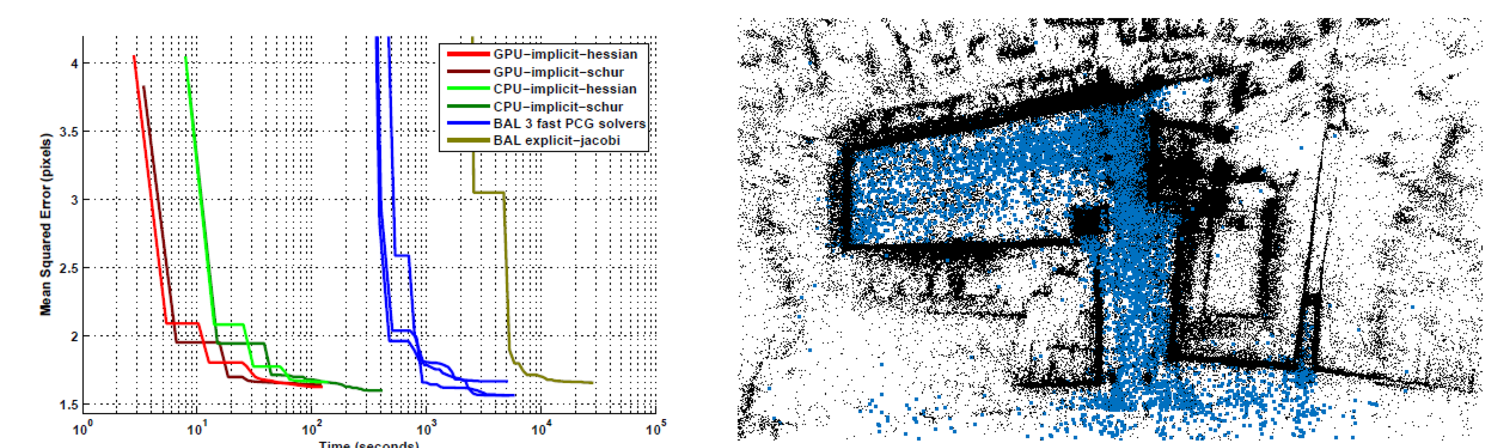
‣ Replace large matrices with on-the-fly computation

- Substantial memory savings.
- Increased GPU throughput due to reduced memory contention.

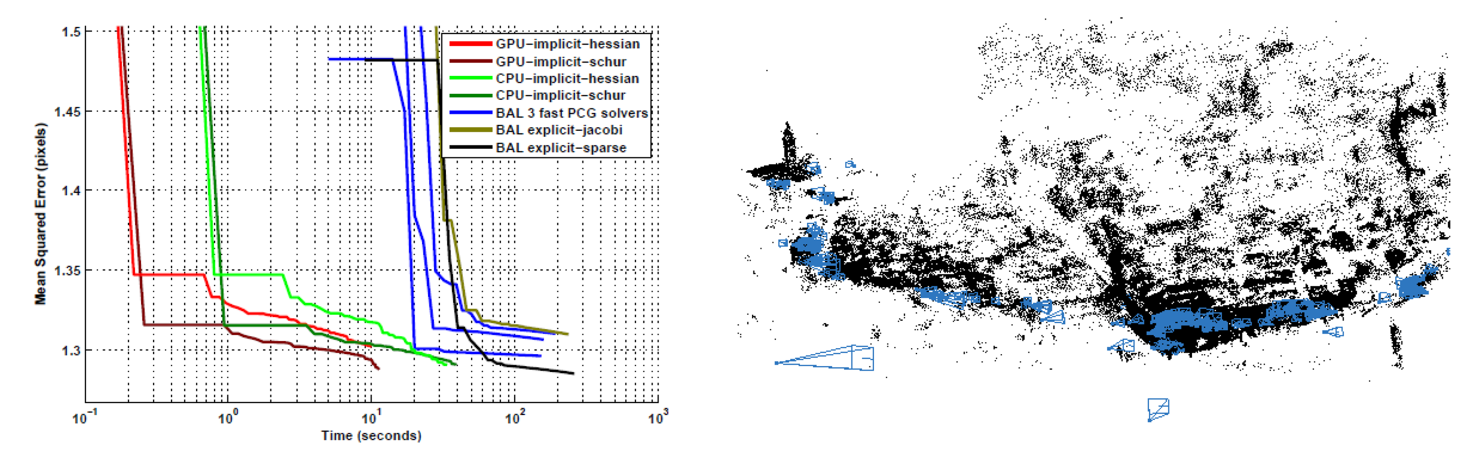
	CPU	GPU
Jx	0.56X	1.44X
J^Ty	0.48X	1.09X
LM	0.46X	**1.27X**

Dubrovnik Final: 4.6K cameras, 1.3M points, and 8M measurements
Memory usage can be reduced from 1.9G to 0.55G

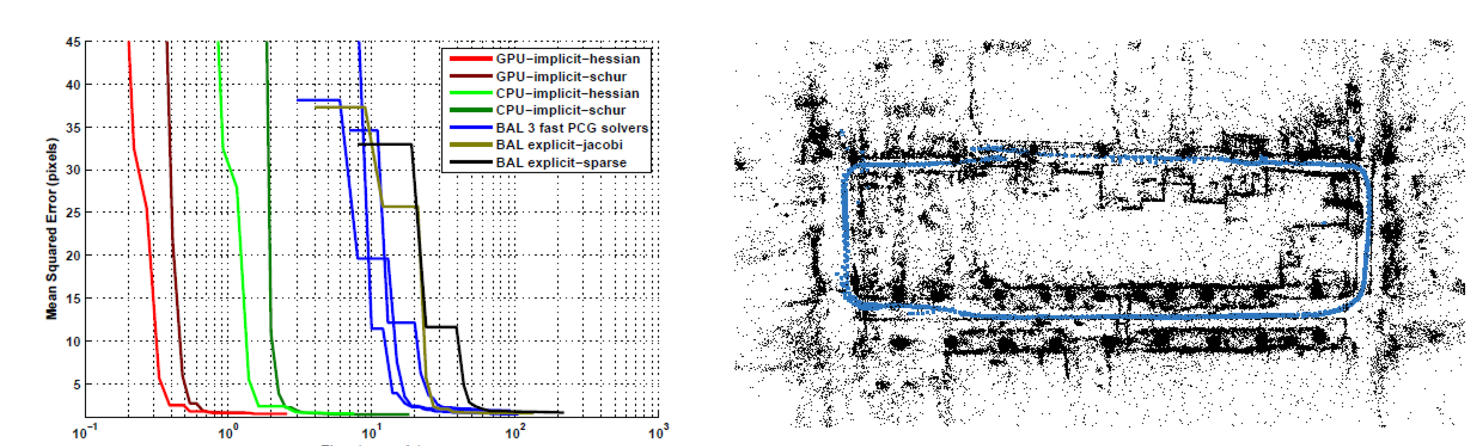
Experiments (comparing with *Agarwal et al. Bundle Adjustment in the Large,ECCV2010*)



Venice Final (13775 cameras, 4.5M points, 50 LM steps in 2 minutes)



Dubrovnik Skeletal (356 cameras, 226730pts, 50 LM steps in 5 seconds)



Ladybug (1723 cameras, 156502pts, 50 LM steps in 2 seconds)

- Comparable convergence behaviors.