BRNO UNIVERSITY OF TECHNOLOGY

FIT

FACULTY OF INFORMATION TECHNOLOGY

double negative visual effects

IMP·ART

Jeff Clifford (Double Negative VFX)
Lukáš Polok (Brno University of Technology)
Simon Pabst (Double Negative VFX)

Talk Overview

1. The need in production (Jeff)
2. The algorithm on the GPU (Lukáš)
3. Integration into DNeg’s pipeline (Simon)

About DNeg

double negative visual effects

- Started in 1998 with a team of 30 people. Now 1250 people approx.
- Latest film work was *Interstellar*
- Offices in London, Singapore & Vancouver
- R&D challenges have changed
- Unique challenges for handling of on-set data appropriate for GPU

IMPART

- Intelligent Management Platform for Advanced Real-time Media Processes
- EU Research Project
- Two Industrial Partners
- Four Universities

On-set Data Capture

- Data captured on-set vital for digital feature film post production
- Reference Photos, HDRIs, Panoramas, LIDAR, GPS, witness cameras, …
- One use-case: Photogrammetry
- FF6 required 8 hours to process on CPU
- IMPART provided opportunity to accelerate that as a POC initially in OpenCL
- Latest CUDA prototype means we can process same data in 1h on a laptop
- Allows for processing of material on-set!

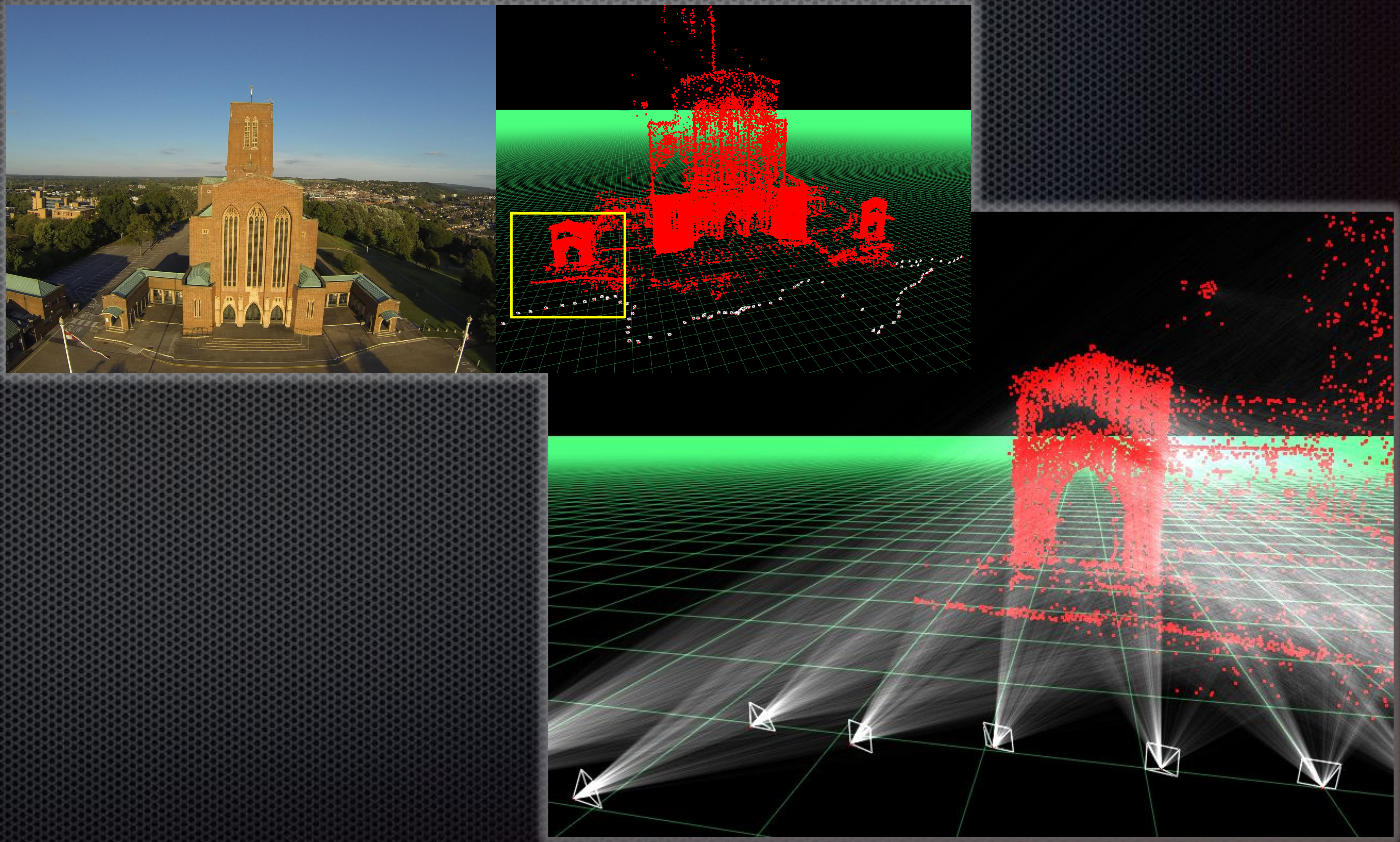
Bundle Adjustment (BA)

- 3D reconstruction from stills (N cameras)
- Optimization problem, solvable using MLE
- Strives to reduce *reprojection errors* (in 2D)
- Related problems in computer vision
- Subtly different from SfM (one camera)
  - Different from SLAM (reduces errors in 3D)

Bundle Adjustment as a Graph

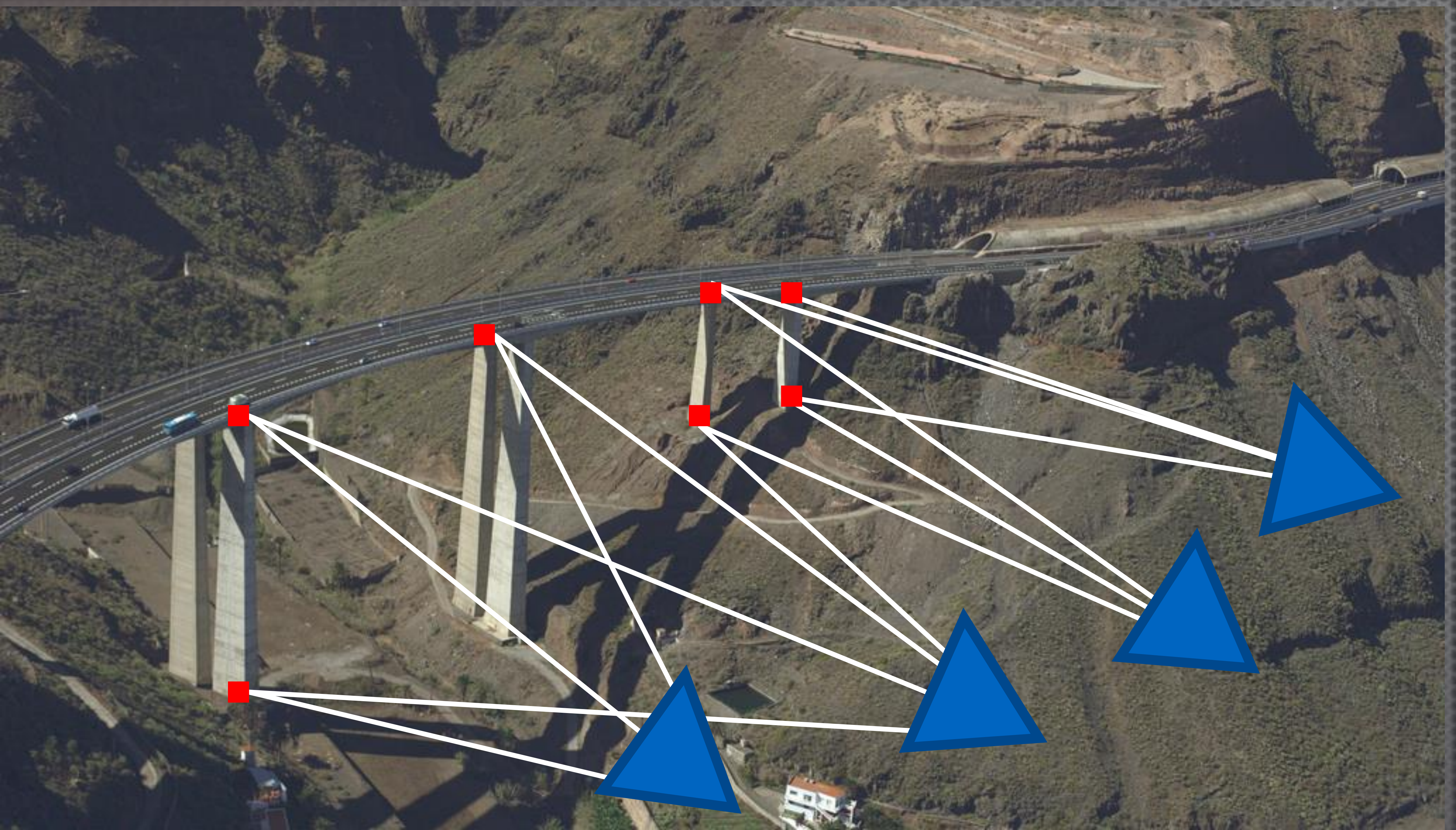
- Vertices:
- 3D point positions
- Camera poses
- Camera parameters

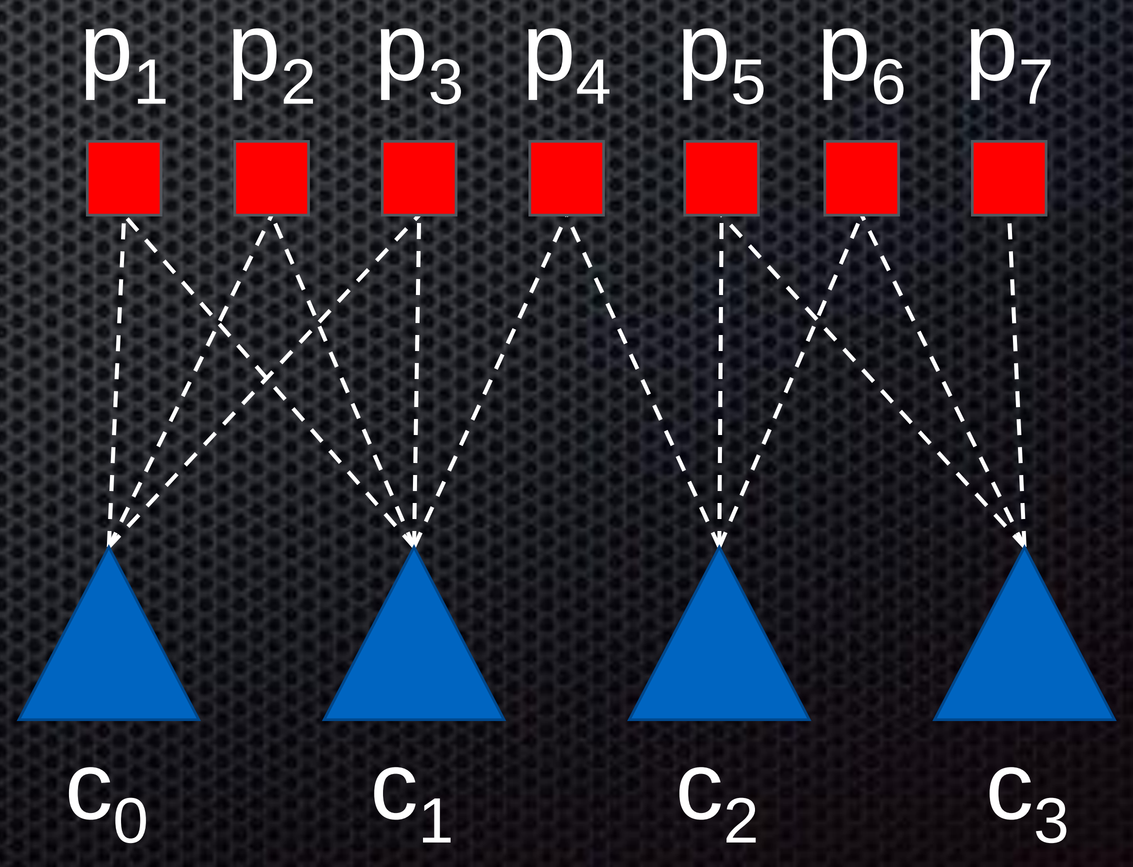
- Edges:
- 3D point observations
- Any other constraints

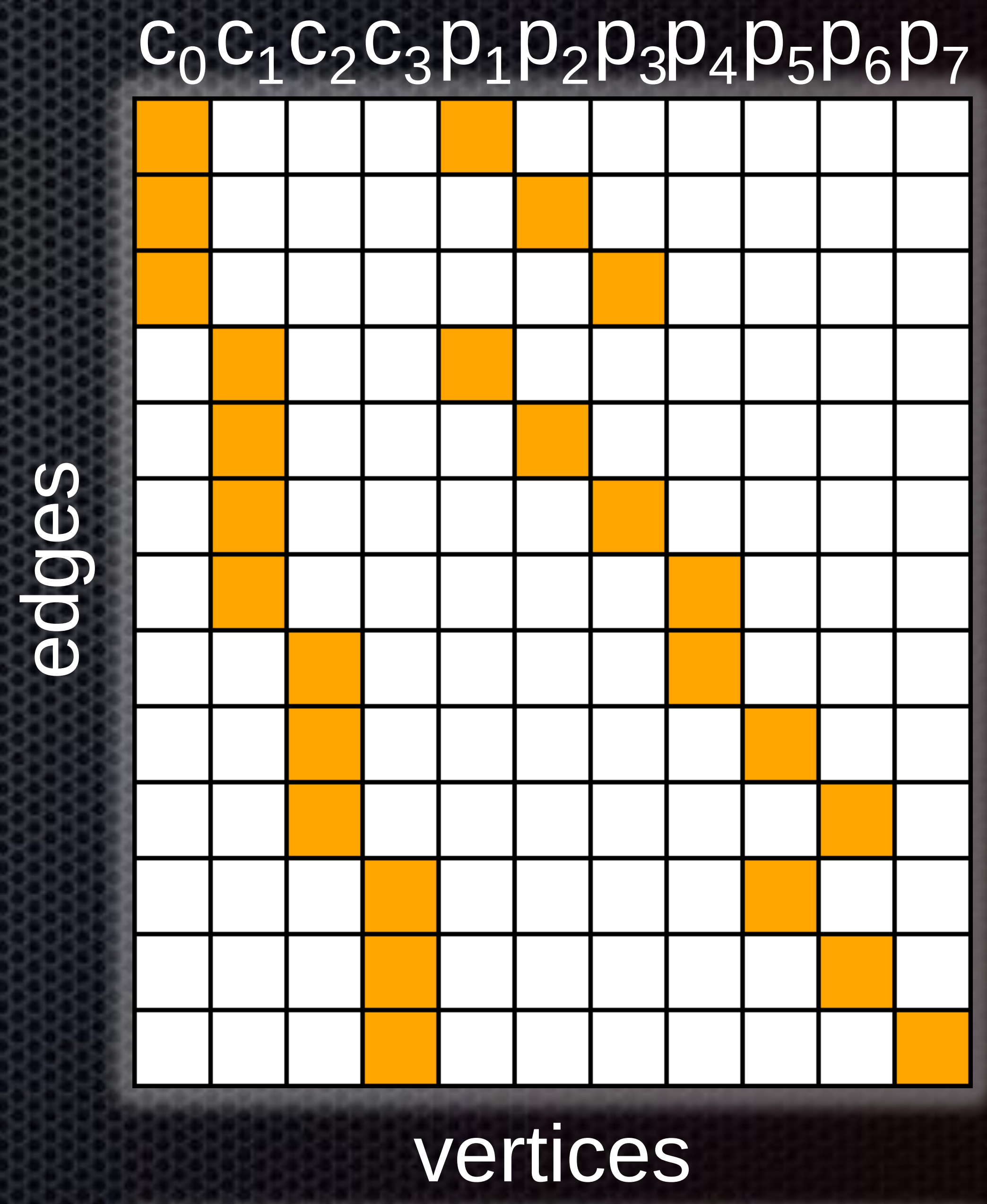


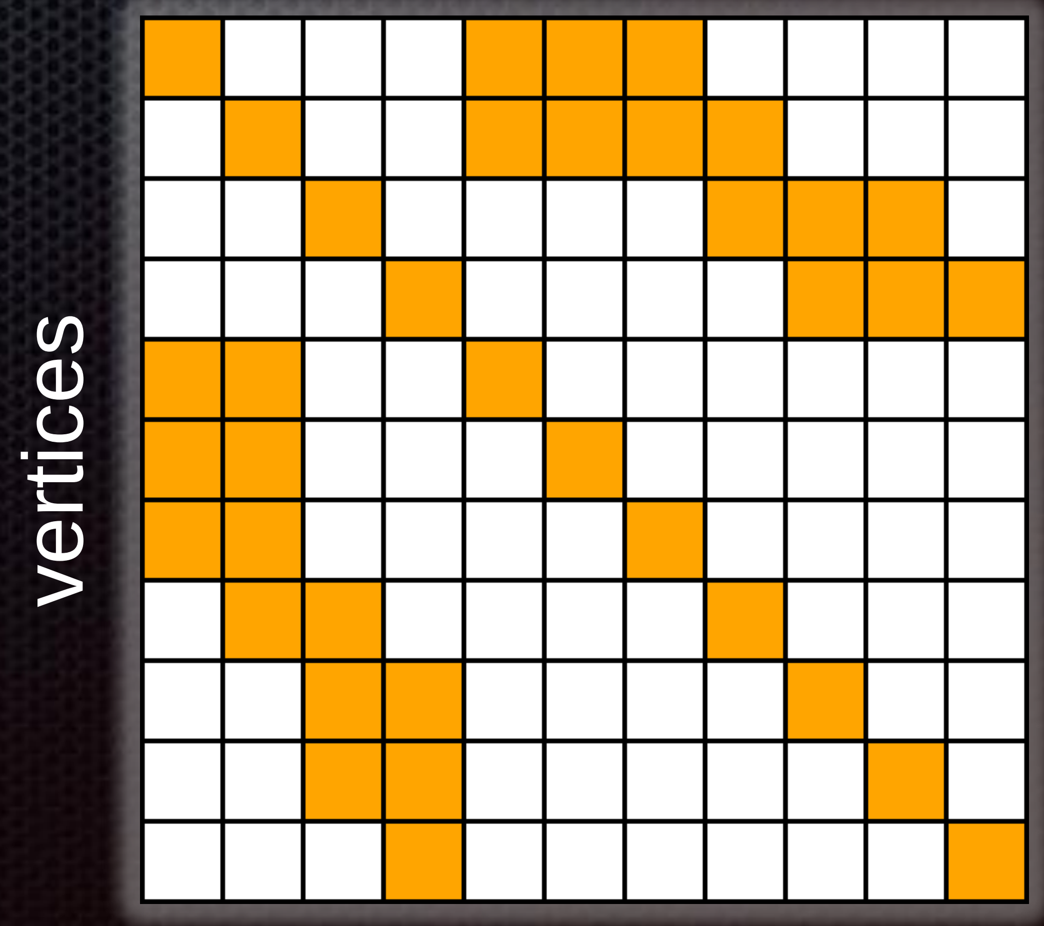
Graph Representation

- Represented by a *sparse* matrix
- Incidence (Jacobian) matrix A
- Adjacency (*Hessian*) matrix Λ
- Has a *block structure*



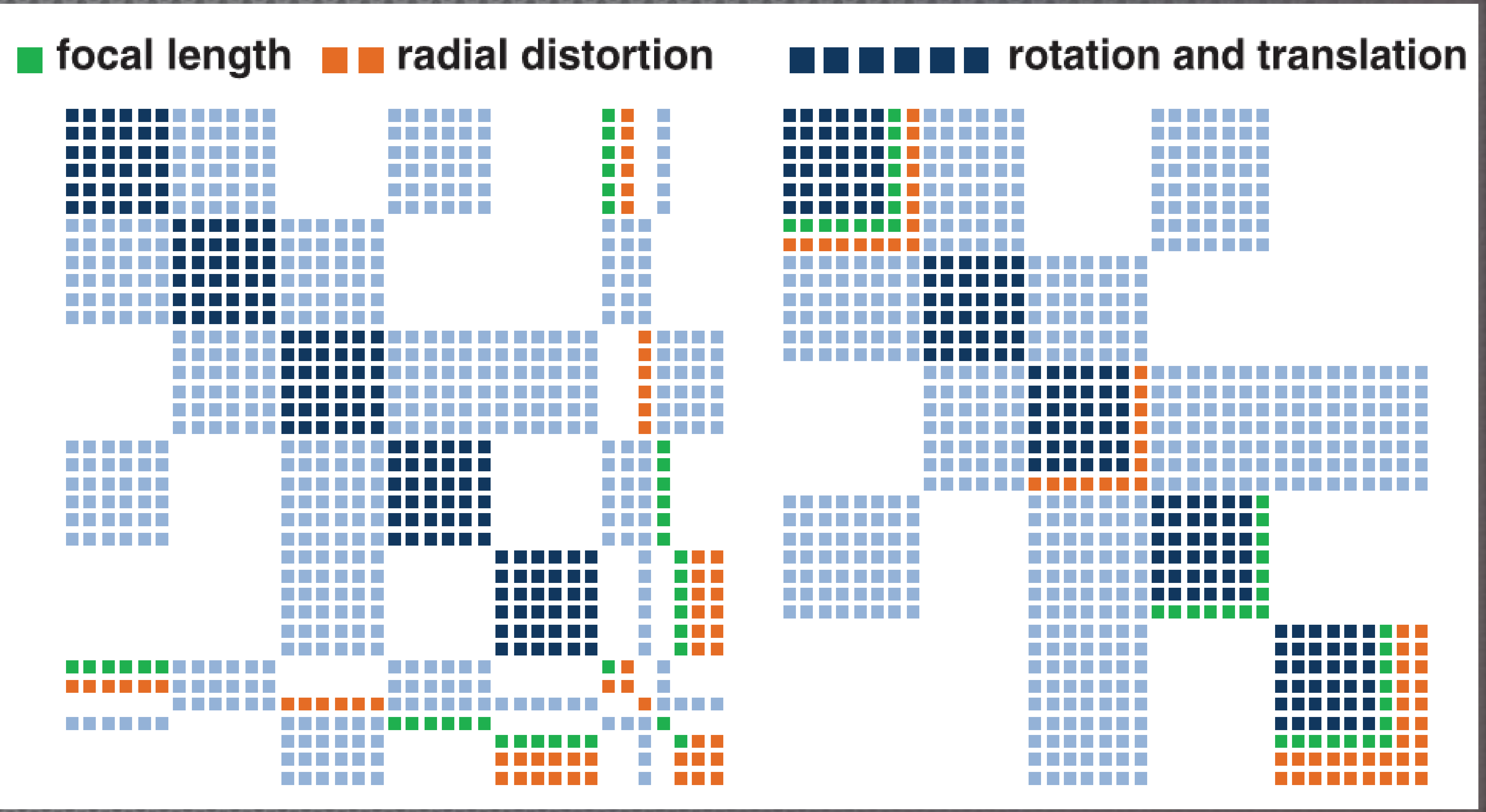






Variable Block Structure

- Size of blocks in a single matrix
- Decompose camera blocks [Jeong12]
- Solved on a GPU [Rennich12, Tawara12]
- Variable block size schemes
- Known at compile-time [Polok13]
- Applies to GPUs as well



Yekeun Jeong et. al., „Pushing the Envelope of Modern Methods for Bundle Adjustment,“ PAMI, 2012
Steve Rennich, „Leveraging Matrix Block Structure In Sparse Matrix-Vector Multiplication,“ talk on GTC 2012
Tetsuo Tawara, „Levenberg-Marquardt Using Block Sparse Matrices on CUDA,“ talk on GTC 2012
Lukas Polok et. al., "Cache efficient implementation for block matrix operations," HPC, 2013

Solving Bundle Adjustment

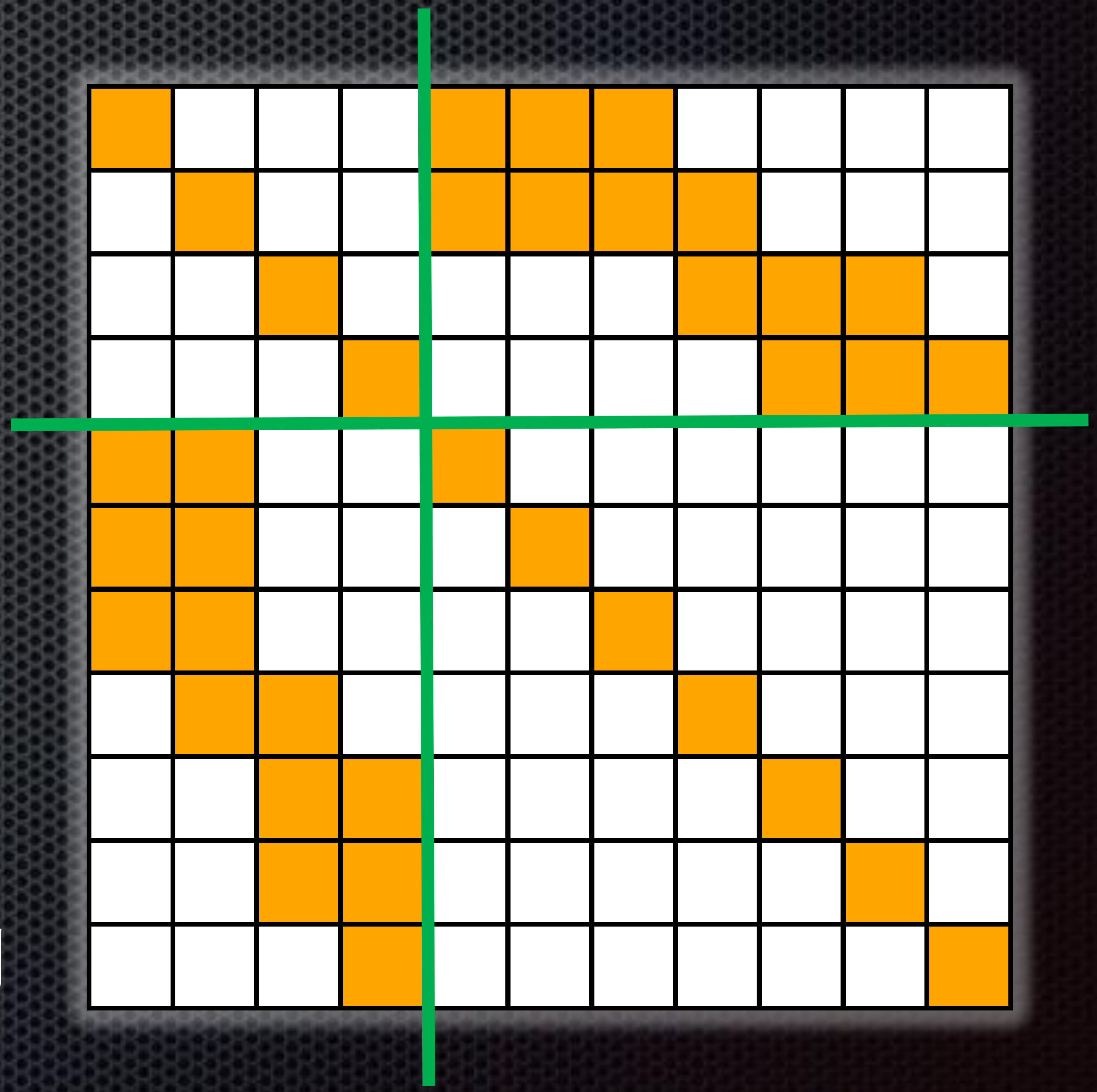
- (Damped) Gauss-Newton methods
- Repeatedly solve for $\Lambda u = r$
- Serial direct methods [Kummerle11, Kaess11]
- Serial sparse factorization, backsubstitution
- Or parallel gradient descent [Wu2013]
- Easy to implement, less numerically robust
- Implemented a *parallel direct* solver

```
while 1
  build linearized system (Λ, r)
  solve u = Λ / r
  if norm(u) < thresh
    done
  update x = x ⊕ u
```

Kummerle, Rainer, et al., „g2o: A general framework for graph optimization," *ICRA, 2011*
Kaess, Michael, et al. „iSAM2: Incremental smoothing and mapping using the Bayes tree," *IJRR,* 2011
Wu, Changchang. „Towards linear-time incremental structure from motion," *3DV, 2013*

Solving Bundle Adjustment Quickly

- A bipartite graph: 3D points not interrelated
- Can use Schur complement
- Maps well to GPU
- Parallel matrix multiplication [Polok15]
- Parallel factorization of reduced camera system
- Can be nested
- Can use maximum independent set for explicit ordering



Lukas Polok et. al., „Fast Sparse Matrix Multiplication on GPU," to appear at HPC, 2015

Solving Time Breakdown

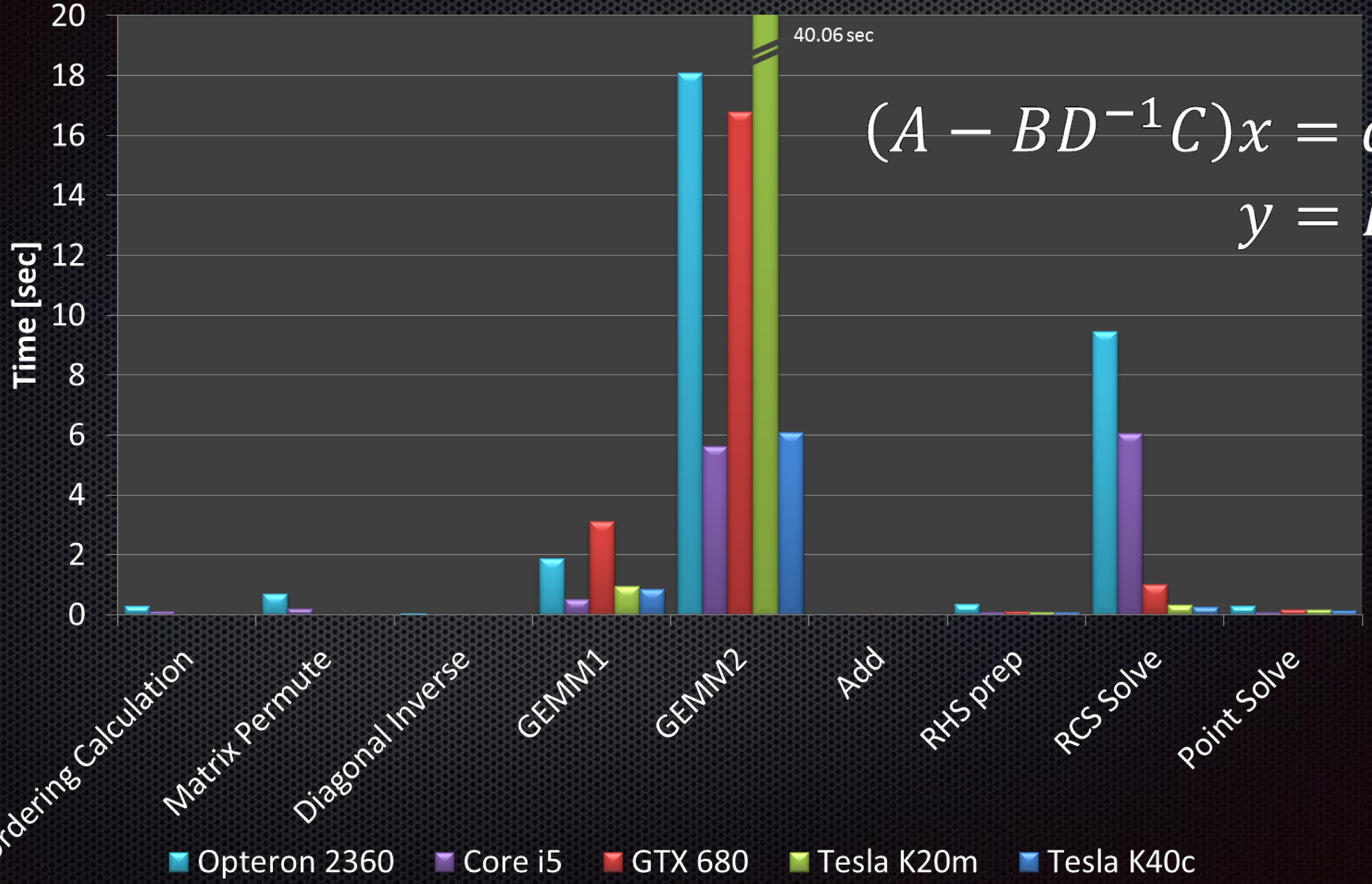
$$\Lambda u = r$$

$$\Lambda = \begin{bmatrix} A & B \\ C & D \end{bmatrix}$$

$$u = \begin{bmatrix} x \\ y \end{bmatrix}$$

$$r = \begin{bmatrix} a \\ b \end{bmatrix}$$

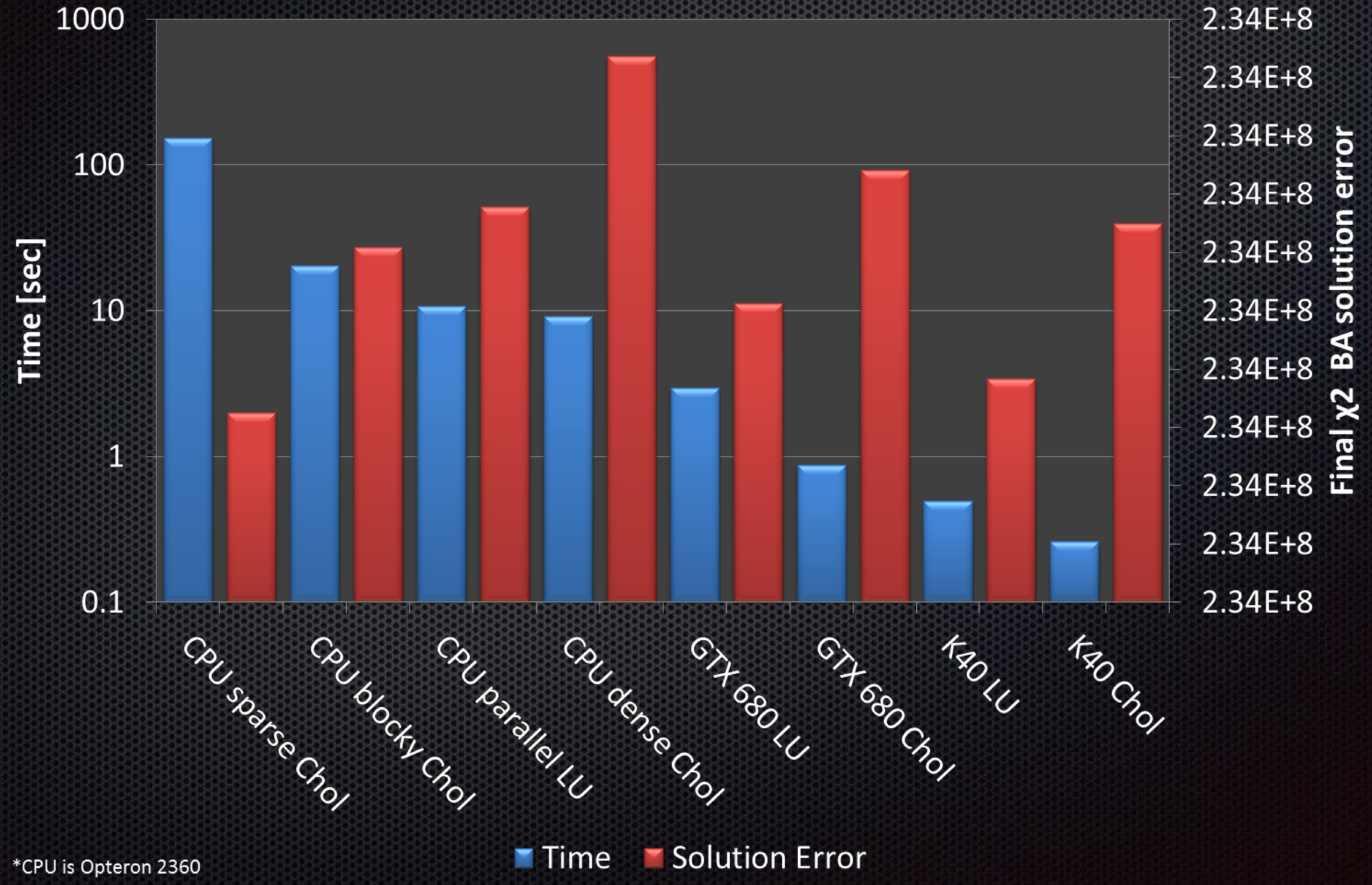
$$Ax + By = a$$
$$Cx + Dy = b$$



$$(A - BD^{-1}C)x = a - BD^{-1}b$$
$$y = D^{-1}(b - Cx)$$

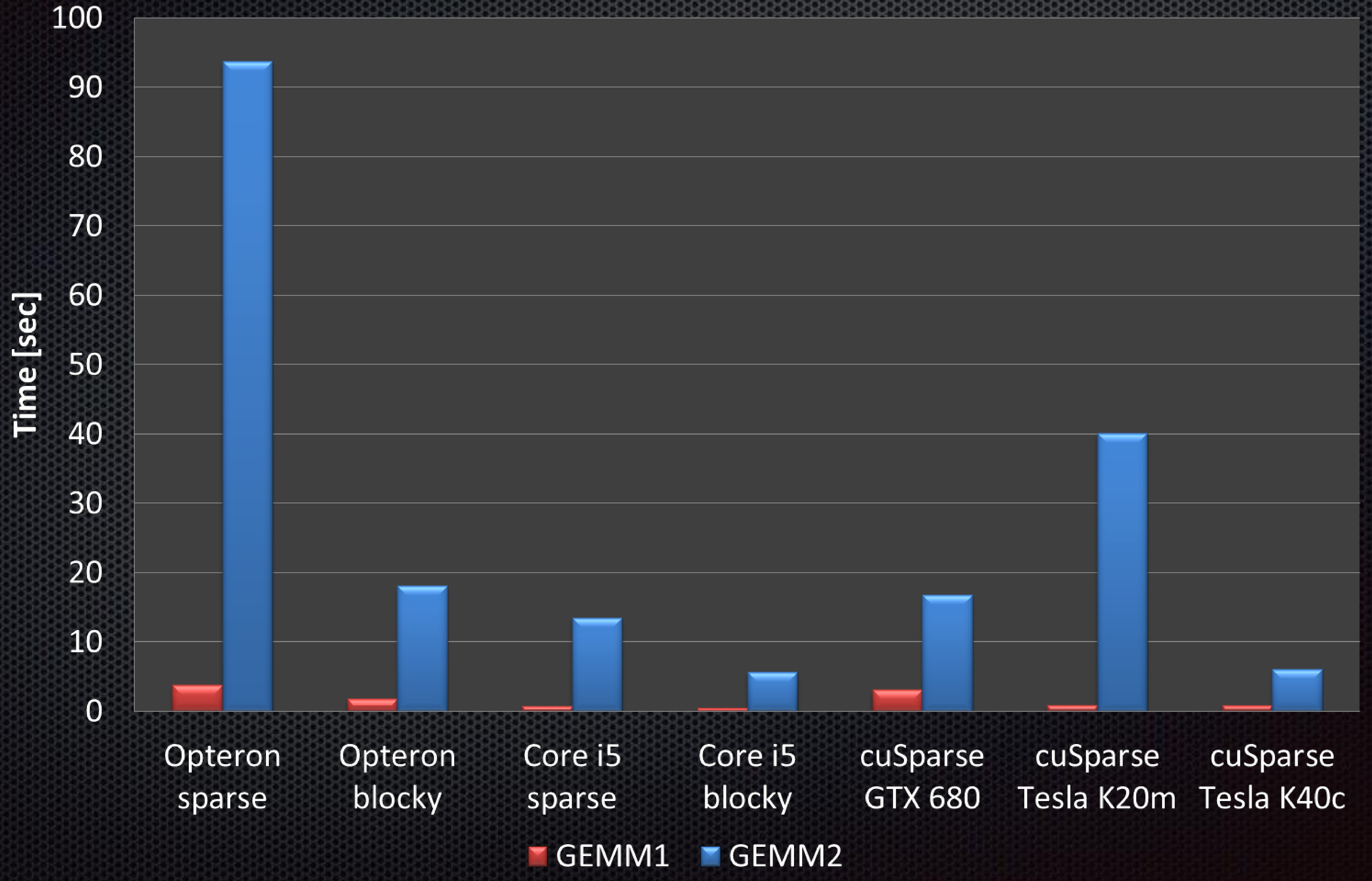
all in double precision

Matrix Factorization Time Comparison



5226 x 5226, 40.06% dense

Matrix Multiplication Time Comparison



Fast Matrix Multiplication in SW

```
BlockMatrix A, B, C, D;
// lambda sections

typedef TypeList(Size<6, 3>, Size<5, 3>) BS;
typedef TransposeSizes<BS>::Result BS_T;
typedef TypeList(Size<3, 3>) D_invS;
// block sizes specifications

BlockMatrix BD_inv, SC; // the results
BD_inv = SpDGEMM<BS, D_invS>(B, D_invS); // calculate BD^-1
SC = SpDGEMM<BS, BS_T>(BD_inv, C); // calculate BD^-1C
```

Lukas Polok et. al., "Cache efficient implementation for block matrix operations," HPC, 2013

Fast Matrix Multiplication in HW

- ESC algorithm [Dalton13, Polok15]
- Expansion
- Sorting
- Compression

Algorithm 1 Setup stage of PSpGEMM.

```
function GEMM(A, B)
    b_cols = AllocInt(nnz(B))
    b_prods = AllocInt(nnz(B) + 1)
    kernel (i = 0 ... nnz(B))
        b_cols[i] = 0
        row = B.i[i]
        b_prods[i] = A.p[row + 1] − A.p[row]
    end kernel ▷ the last element of b_prods not initalized
    kernel (i = 0 ... Cols(B))
        b_cols[B.p[i + 1] − 1] = 1
    end kernel
    b_cols = ExclusiveScan(b_cols)
    b_prods = ExclusiveScan(b_prods)
    exp_size = b_prods[nnz(B)]                ▷ expansion size
```

Algorithm 2 Expansion and sorting stages.

```
    ex_cols = AllocInt(exp_size)
    ex_rows = AllocInt(exp_size)
    ex_values = AllocFloat(exp_size)
    ex_hf = AllocBit(exp_size)   ▷ head flags bit array
    kernel (i = 0 ... (N = GPU_hardware threads))
        begin = ⌊exp_size · i/N⌋
        count = ⌊exp_size · (i + 1)/N⌋ − begin
        elemB = upper_bound(b_prods, begin) − 1
        col_skip = begin − b_prods[elemB]
        for (prod = 0; prod < count; ++ elemB) do
            rowB = B.i[elemB]
            elemA = col_skip + A.p[rowB]
            endA = A.p[rowB + 1]
            while (elemA < endA and p < count) do
                dest = begin + p
                cur_col = ex_cols[dest] = b_cols[elemB]
                ex_rows[dest] = A.i[elemA]
                ex_values[dest] = A.x[elemA] · B.x[elemB]
                ex_hf[dest] = cur_col > b_cols[elemB − 1]
                ++ elemA, ++ prod
            end while
            col_skip = 0     ▷ skip in the first iteration only
        end for
    end kernel
    SegmentedSort(ex_hf, ex_rows, ex_values)
    tail_blocks = ⌈exp_size/block_size⌉
    tail_counts = AllocInt(tail_blocks + 1)
                    ▷ or reuse b_prods which is not needed below
    kernel (i = 0 ... exp_size − 1)
        local int flags[block_size]          ▷ in local memory
        flags[i] = ex_cols[i] < ex_cols[i + 1] or
                   ex_rows[i] < ex_rows[i + 1]
        g = ⌊i/block_size⌋      ▷ cooperating thread group
        tail_counts[g] = cooperative_reduce(flags)
    end kernel
    tail_counts = ExclusiveScan(tail_counts)
    product_NNZ = tail_counts[tail_blocks] + 1
```

Algorithm 3 Compression stage.

```
    C.p = AllocInt(cols(B) + 1)
    C.i = AllocInt(product_NNZ)
    C.x = AllocFloat(product_NNZ)
    kernel (i = 0 ... exp_size − 1)
        g = ⌊i/block_size⌋      ▷ cooperating thread group
        col_tail = ex_cols[i] < ex_cols[i + 1]
        elem_tail = ex_rows[i] < ex_rows[i + 1] or col_tail

        local int flags[block_size]          ▷ in local memory
        flags[i] = elem_tail
        flags = cooperative_scan(flags)
        compressed_index = tail_counts[g] + flags[i]
        if (elem_tail and i < exp_size) then
            C.i[compressed_index] = i   ▷ write indices of
        end if    ▷ reduced values of elements in expansion
        if (col_tail and i < exp_size − 1) then
            C.p[ex_cols[i] + 1] = compressed_index + 1
        end if  ▷ write positions of beginnings of columns
    end kernel
    C.p[0] = 0                     ▷ need to write this explicitly
    ex_values = SegmentedReduction(C.i, ex_values)

    kernel (i = 0 ... product_NNZ)
        expansion_index = C.i[i]
        C.i[i] = ex_rows[expansion_index]
        C.x[i] = ex_values[expansion_index]
    end kernel
    return C
end function
```


Steven Dalton et. al., "Optimizing sparse matrix-matrix multiplication for the GPU," 2013
Lukas Polok et. al., „Fast Sparse Matrix Multiplication on GPU," to appear at HPC, 2015

Fast Matrix Multiplication in HW

- ESC algorithm [Dalton13, Polok15]
- Expansion
- Sorting
- Compression

- 480 MFLOP/s (0.0336%)
- Blocks to the rescue!

Algorithm 1 Setup stage of PSpGEMM.

```
function GEMM(A, B)
    b_cols = AllocInt(NNZ(B))
    b_prods = AllocInt(NNZ(B) + 1)
    kernel (i = 0 ... NNZ(B))
        b_cols[i] = 0
        row = B.i[i]
        b_prods[i] = A.p[row + 1] − A.p[row]
    end kernel ▷ the last element of b_prods not initalized
    kernel (i = 0 ... Cols(B))
        b_cols[B.p[i + 1] − 1] = 1
    end kernel
    b_cols = ExclusiveScan(b_cols)
    b_prods = ExclusiveScan(b_prods)
    exp_size = b_prods[NNZ(B)]                    ▷ expansion size
```

Algorithm 2 Expansion and sorting stages.

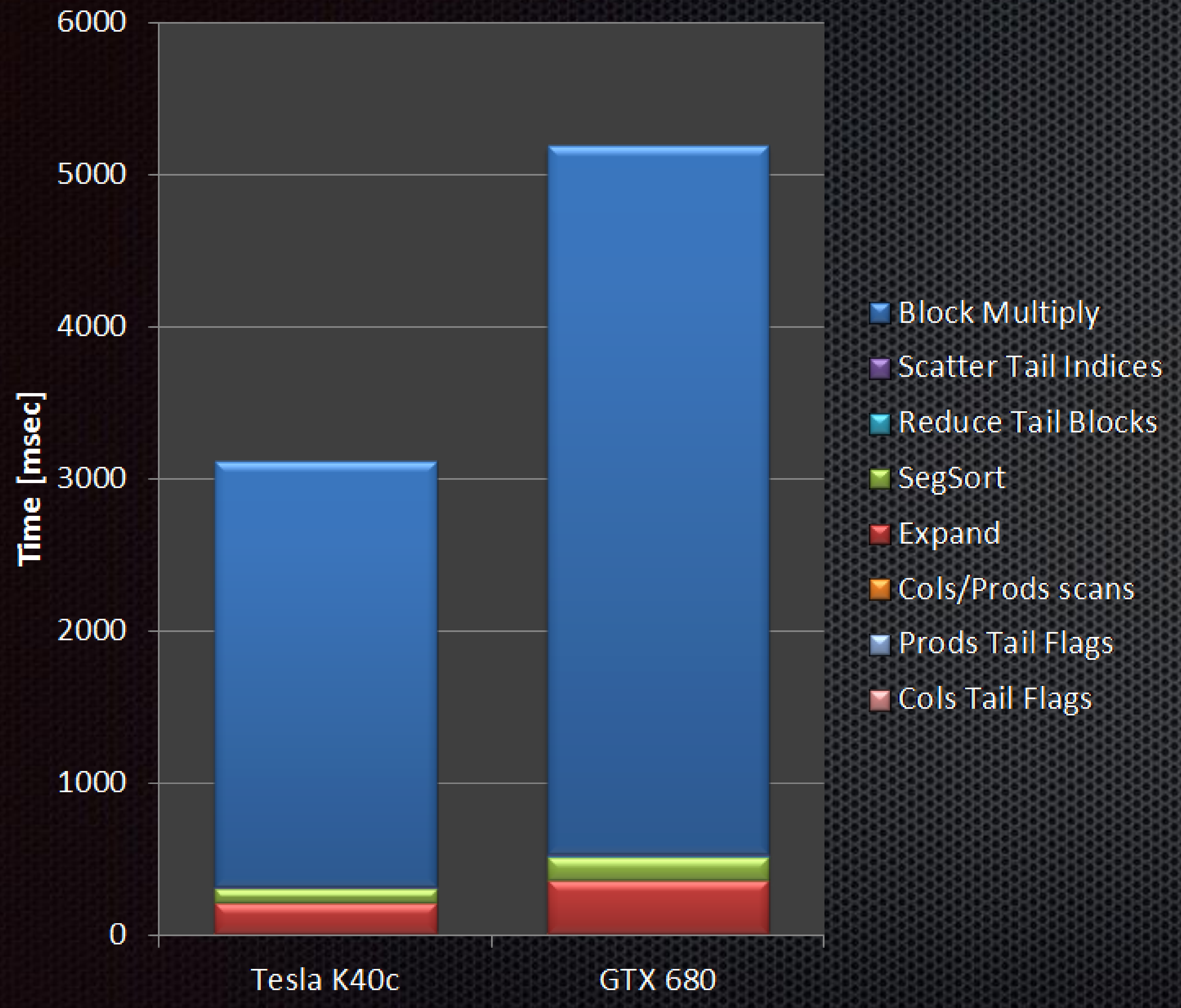
```
    ex_cols = AllocInt(exp_size)
    ex_rows = AllocInt(exp_size)
    ex_values = AllocFloat(exp_size)
    ex_hf = AllocBit(exp_size)   ▷ head flags bit array
    kernel (i = 0 ... (N = GPU_hardware threads))
        begin = ⌊exp_size · i/N⌋
        count = ⌊exp_size · (i + 1)/N⌋ − begin
        elemB = UPPER_BOUND(b_prods, begin) − 1
        col_skip = begin − b_prods[elemB]
        for (prod = 0; prod < count; ++ elemB) do
            rowB = B.i[elemB]
            elemA = col_skip + A.p[rowB]
            endA = A.p[rowB + 1]
            while (elemA < endA and p < count) do
                dest = begin + p
                cur_col = ex_cols[dest] = b_cols[elemB]
                ex_rows[dest] = A.i[elemA]
                ex_values[dest] = A.x[elemA] · B.x[elemB]
                ex_hf[dest] = cur_col > b_cols[elemB − 1]
                ++ elemA, ++ prod
            end while
            col_skip = 0       ▷ skip in the first iteration only
        end for
    end kernel
    SegmentedSort(ex_hf, ex_rows, ex_values)
    tail_blocks = ⌈exp_size/block_size⌉
    tail_counts = AllocInt(tail_blocks + 1)
                        ▷ or reuse b_prods which is not needed below
    kernel (i = 0 ... exp_size − 1)
        local int flags[block_size]          ▷ in local memory
        flags[i] = ex_cols[i] < ex_cols[i + 1] or
                       ex_rows[i] < ex_rows[i + 1]
        g = ⌊i/block_size⌋       ▷ cooperating thread group
        tail_counts[g] = COOPERATIVE_REDUCE(flags)
    end kernel
    tail_counts = ExclusiveScan(tail_counts)
    product_NNZ = tail_counts[tail_blocks] + 1
```

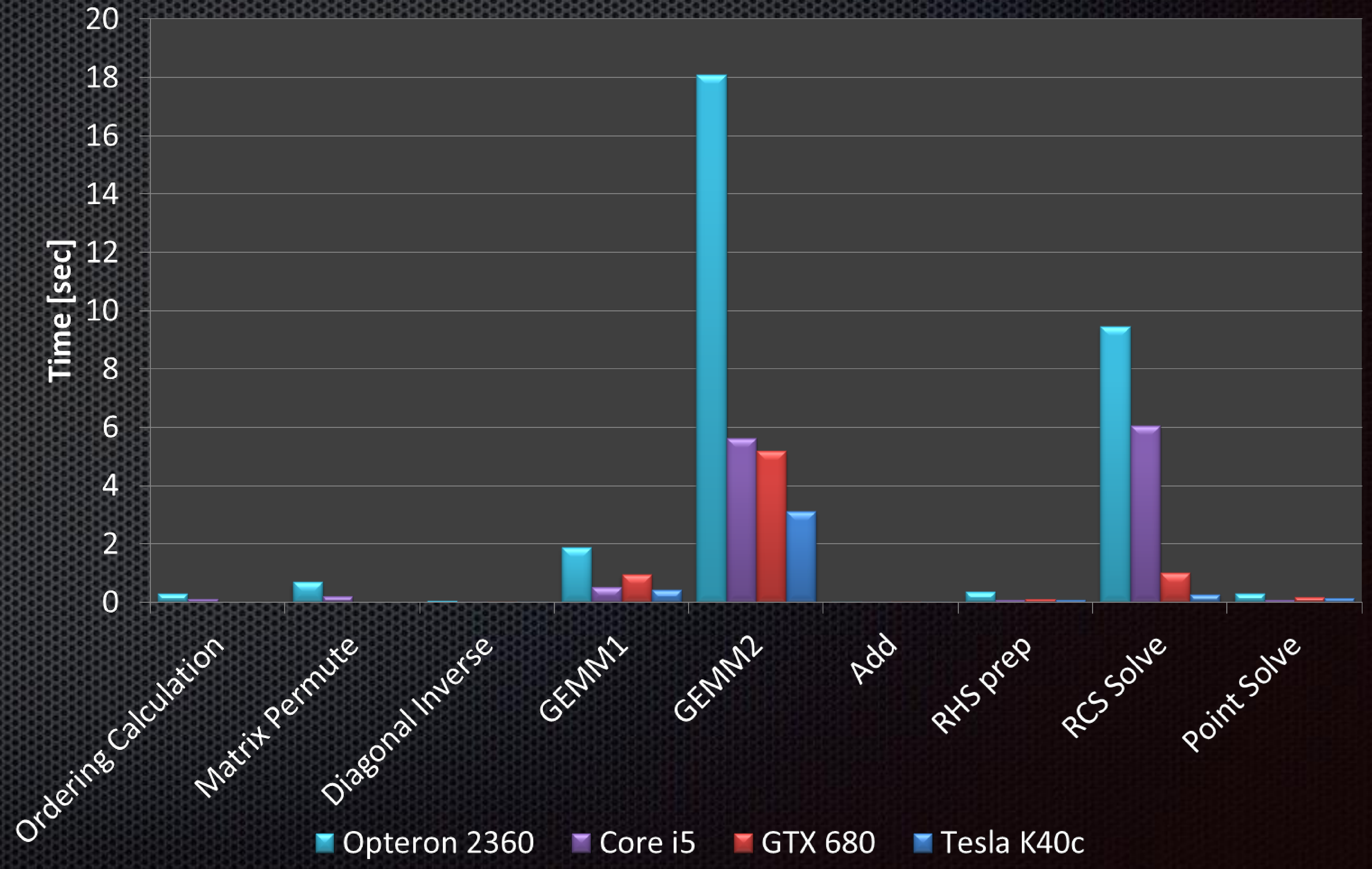
Algorithm 3 Compression stage.

```
    C.p = AllocInt(cols(B) + 1)
    C.i = AllocInt(product_NNZ)
    C.x = AllocFloat(product_NNZ)
    kernel (i = 0 ... exp_size − 1)
        g = ⌊i/block_size⌋       ▷ cooperating thread group
        col_tail = ex_cols[i] < ex_cols[i + 1]
        elem_tail = ex_rows[i] < ex_rows[i+1] or col_tail

        local int flags[block_size]          ▷ in local memory
        flags[i] = elem_tail
        flags = COOPERATIVE_SCAN(flags)
        compressed_index = tail_counts[g] + flags[i]
        if (elem_tail and i < exp_size) then
            C.i[compressed_index] = i  ▷ write indices of
        end if    ▷ reduced values of elements in expansion
        if (col_tail and i < exp_size − 1) then
            C.p[ex_cols[i] + 1] = compressed_index + 1
        end if   ▷ write positions of beginnings of columns
    end kernel
    C.p[0] = 0                        ▷ need to write this explicitly
    ex_values = SegmentedReduction(C.i, ex_values)

    kernel (i = 0 ... product_NNZ)
        expansion_index = C.i[i]
        C.i[i] = ex_rows[expansion_index]
        C.x[i] = ex_values[expansion_index]
    end kernel
    return C
end function
```


Steven Dalton et. al., "Optimizing sparse matrix-matrix multiplication for the GPU," 2013
Lukas Polok et. al., „Fast Sparse Matrix Multiplication on GPU," to appear at HPC, 2015

Block Matrix Multiplication Time





Estimating 3D reconstruction errors

- Important for practical use on-set
- Involves system matrix inverse (fully dense!)

Estimating 3D reconstruction errors

Can calculate parts of the inverse [Björck96]

$$\Sigma_{ii} = \frac{1}{R_{ii}} \left[\frac{1}{R_{ii}} - \sum_{k=i+1, R_{ik} \neq 0}^{n} R_{ik} \Sigma_{ki} \right]$$

$$\Sigma_{ij} = \frac{1}{R_{ii}} \left[- \sum_{k=i+1, R_{ik} \neq 0}^{j} R_{ik} \Sigma_{kj} - \sum_{k=j+1, R_{ik} \neq 0}^{n} R_{ik} \Sigma_{jk} \right]$$

Difficult to parallelize

A. Björck, „Numerical methods for least squares problems,“ SIAM, 1996

Estimating 3D reconstruction errors

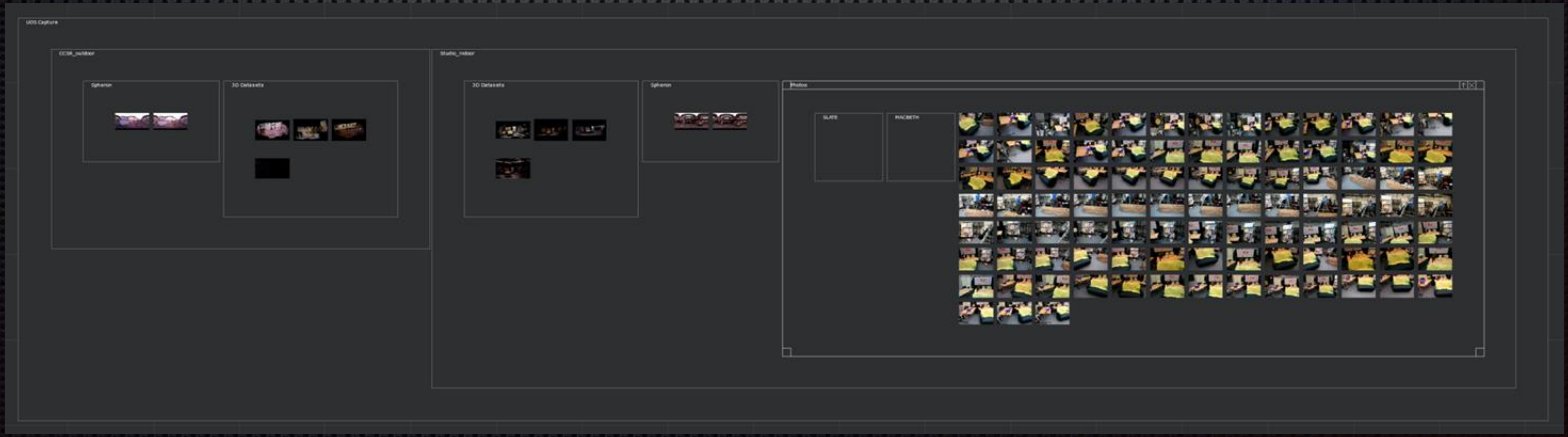
Can update it incrementally very fast! [Ila15]

$$\Delta\Sigma = \hat{\Sigma} A_u^\top (I - A_u \hat{\Sigma} A_u^\top)^{-1} A_u \hat{\Sigma}$$

Viorela Ila et. al, „Fast Covariance Recovery in Incremental Nonlinear Least Square Solvers“, to appear at ICRA, 2015

Jigsaw

- DNeg’s in-house tool to ingest and process data captured on-set
- Handles photos, LIDAR, witness cameras, HDRIs, …
- Can dispatch processing jobs to the farm or locally (on-set)
- Easy to extend



jigsaw : jigsaw

Jigsaw: ./FF6_REF_HELI_SHOOT_v002.jgs

File Edit View S

jigsaw : jigsaw

```
dng reader added
Version: 1.8.40-19-
Loading prefs from
Error loading stati

Executing startup s

connecting to shotg
done
sys.version_info(ma


The following pytho

./pythonScripts/TAU
./pythonScripts/arc
./pythonScripts/ass
./pythonScripts/ass
./pythonScripts/bat
./pythonScripts/exp
./pythonScripts/exp
./pythonScripts/foc
./pythonScripts/foc
./pythonScripts/gig
./pythonScripts/imp
./pythonScripts/ivy
./pythonScripts/ivy
./pythonScripts/ivy
./pythonScripts/ivy
./pythonScripts/lis
./pythonScripts/loa
./pythonScripts/loa
./pythonScripts/mea
./pythonScripts/mer
./pythonScripts/mov
./pythonScripts/mov
./pythonScripts/new
./pythonScripts/pub
./pythonScripts/pub
./pythonScripts/ren
./pythonScripts/rod
./pythonScripts/run
./pythonScripts/sav
./pythonScripts/sav
./pythonScripts/sca
./pythonScripts/sea
./pythonScripts/sea
./pythonScripts/sea
./pythonScripts/set
./pythonScripts/set
./pythonScripts/set
./pythonScripts/set
./pythonScripts/set
./pythonScripts/snap
./pythonScripts/sor
./pythonScripts/sor
./pythonScripts/tim
/tools/SITE/data/ji
/tools/SITE/data/ji
/tools/SITE/data/ji

End of imported scr
```

Questions ?

DNeg is hiring!!! Join our teams in London, Singapore and Vancouver (event next week!)

BRNO UNIVERSITY OF TECHNOLOGY

FIT

FACULTY OF INFORMATION TECHNOLOGY

double negative visual effects