IMAGE MOSAICKING IN GPU

Lu The Kiet

Mallipeddi Venkata Harish

Stephanus

Outline

- Objective
- Assumptions
- What we accomplished
- Why GPU
- Implementation + Results
- Conclusion
- Future Improvements

Objective

- To investigate how GPU aids in accelerating image registration
- 4 levels of challenge
  - **Entry**: Register two images with pan, tilt & zoom.
  - **Intermediate**: Register a sequence of images with pan & tilt.
  - **Advanced**: Same as intermediate but with zoom.
  - **Ultimate**: Register two videos real-time.

Assumptions

- Camera is moving slowly.
- Minimal change in light conditions.
- Camera is not too close to an object.

What we have accomplished

- Entry level
- Intermediate level
- Advanced level
- Ultimate level

Why GPU

- Current state of hardware

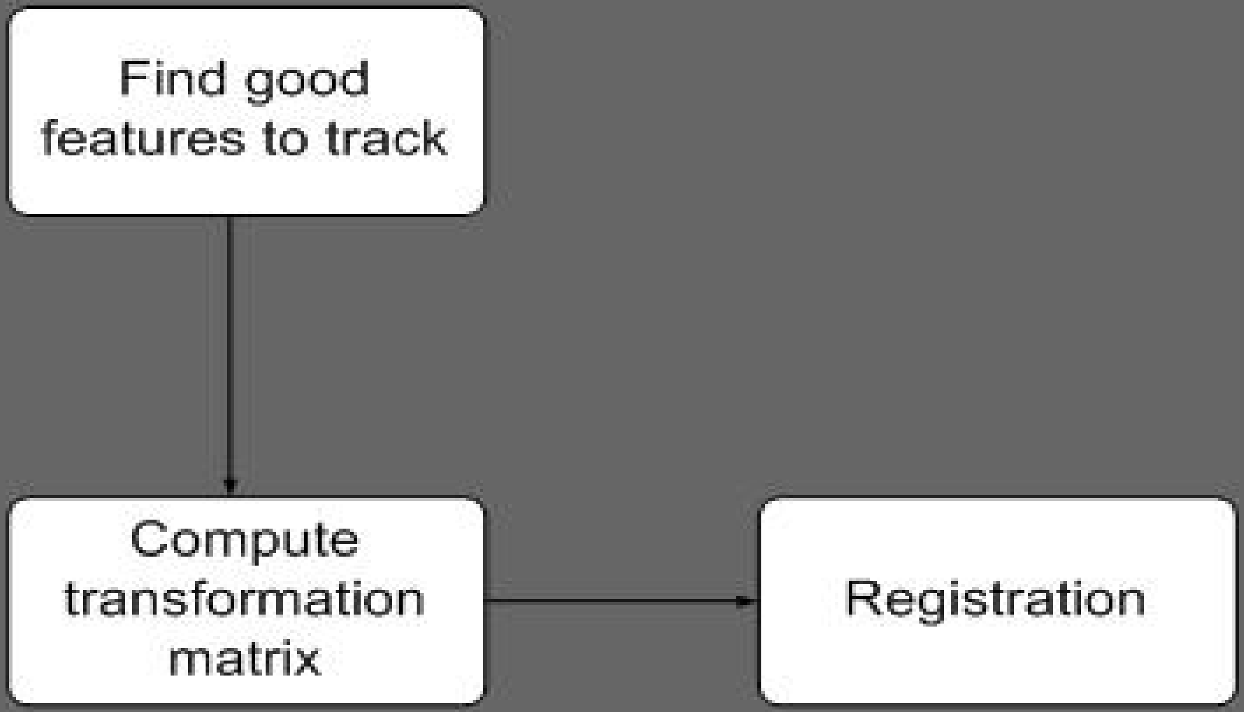
- ATI Radeon X1800XT
- 120 GFLOPs peak (fragment engine)
- 42 GB/s to video memory

- Intel 3.0 GHz Pentium 4
- 12 GFLOPs peak (MAD)
- 5.96 GB/s to main memory

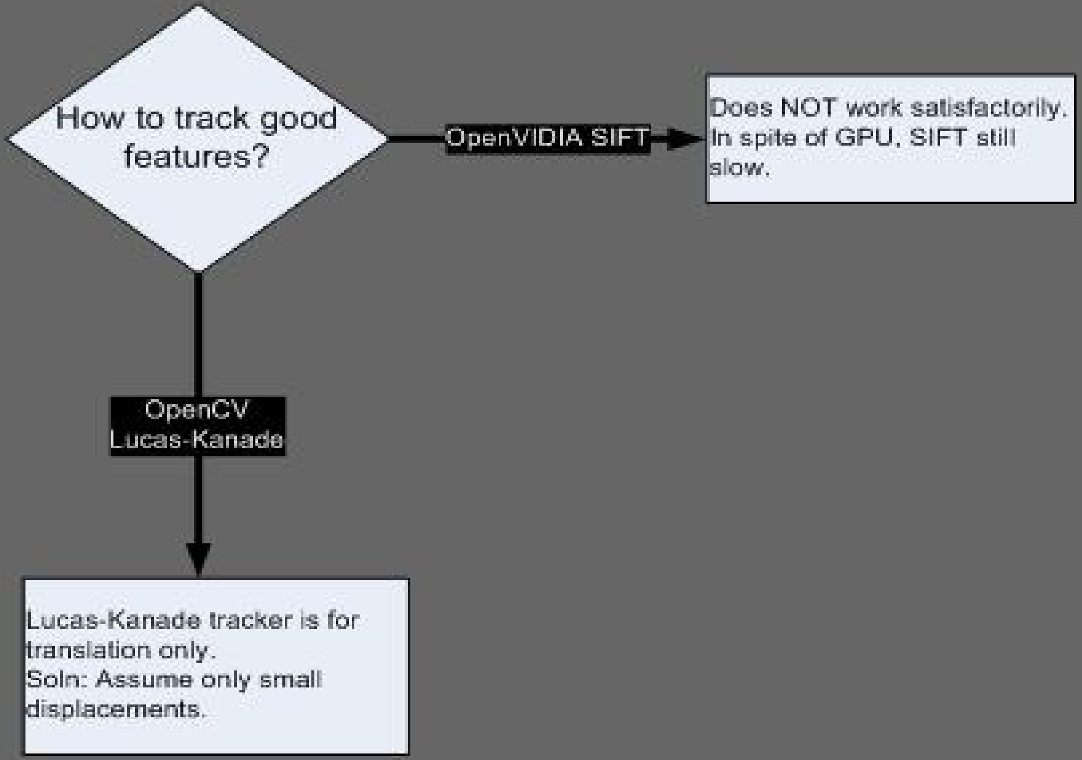
Why GPU

- Data parallelism:
  - Lots of data on which the same computation is being executed.
  - No dependencies between data elements in each step in the computation (kernel). But often requires redesign of traditional algorithms.

Implementation – 3 steps



Feature tracking



OpenVIDIA feature track

OpenCV's feature tracking

OpenVIDIA feature tracking

- Bad performance even on GPU.
- On previous images, OpenVIDIA Feature Tracking takes 1.21515 secs
  - Graphics card used: Nvidia Geforce 7600GT
- OpenCV's Lucas Kanade feature tracking takes 0.0617037 secs
  - CPU: AMD Athlon 2.01 GHz

Transformation matrix - affine vs. projective

- We used affine transformation initially.
- Affine does NOT give good results.
- In real life, we find transformations are NOT strictly affine.
- We switched to projective transformation model (homography).
- Projective gives much better results.

Affine Transformation

Projective Transformation

Projective transformation

- Extensively for 3-D affine modelling transformations and for perspective camera transformations

$$x = \frac{au + bv + c}{gu + hv + i}, \quad y = \frac{du + ev + f}{gu + hv + i}$$

- Manipulation is much easier in homogeneous matrix notation

Projective Transformation

$$\mathbf{p}_d = \mathbf{M}_{sd}\mathbf{p}_s$$

$$= \begin{pmatrix} x' \\ y' \\ w \end{pmatrix} = \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} \begin{pmatrix} u' \\ v' \\ q \end{pmatrix}$$

$$\begin{pmatrix} u_0 & v_0 & 1 & 0 & 0 & 0 & -u_0x_0 & -v_0x_0 \\ u_1 & v_1 & 1 & 0 & 0 & 0 & -u_1x_1 & -v_1x_1 \\ u_2 & v_2 & 1 & 0 & 0 & 0 & -u_2x_2 & -v_2x_2 \\ u_3 & v_3 & 1 & 0 & 0 & 0 & -u_3x_3 & -v_3x_3 \\ 0 & 0 & 0 & u_0 & v_0 & 1 & -u_0y_0 & -v_0y_0 \\ 0 & 0 & 0 & u_1 & v_1 & 1 & -u_1y_1 & -v_1y_1 \\ 0 & 0 & 0 & u_2 & v_2 & 1 & -u_2y_2 & -v_2y_2 \\ 0 & 0 & 0 & u_3 & v_3 & 1 & -u_3y_3 & -v_3y_3 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \\ d \\ e \\ f \\ g \\ h \end{pmatrix} = \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

Computing transformation matrix

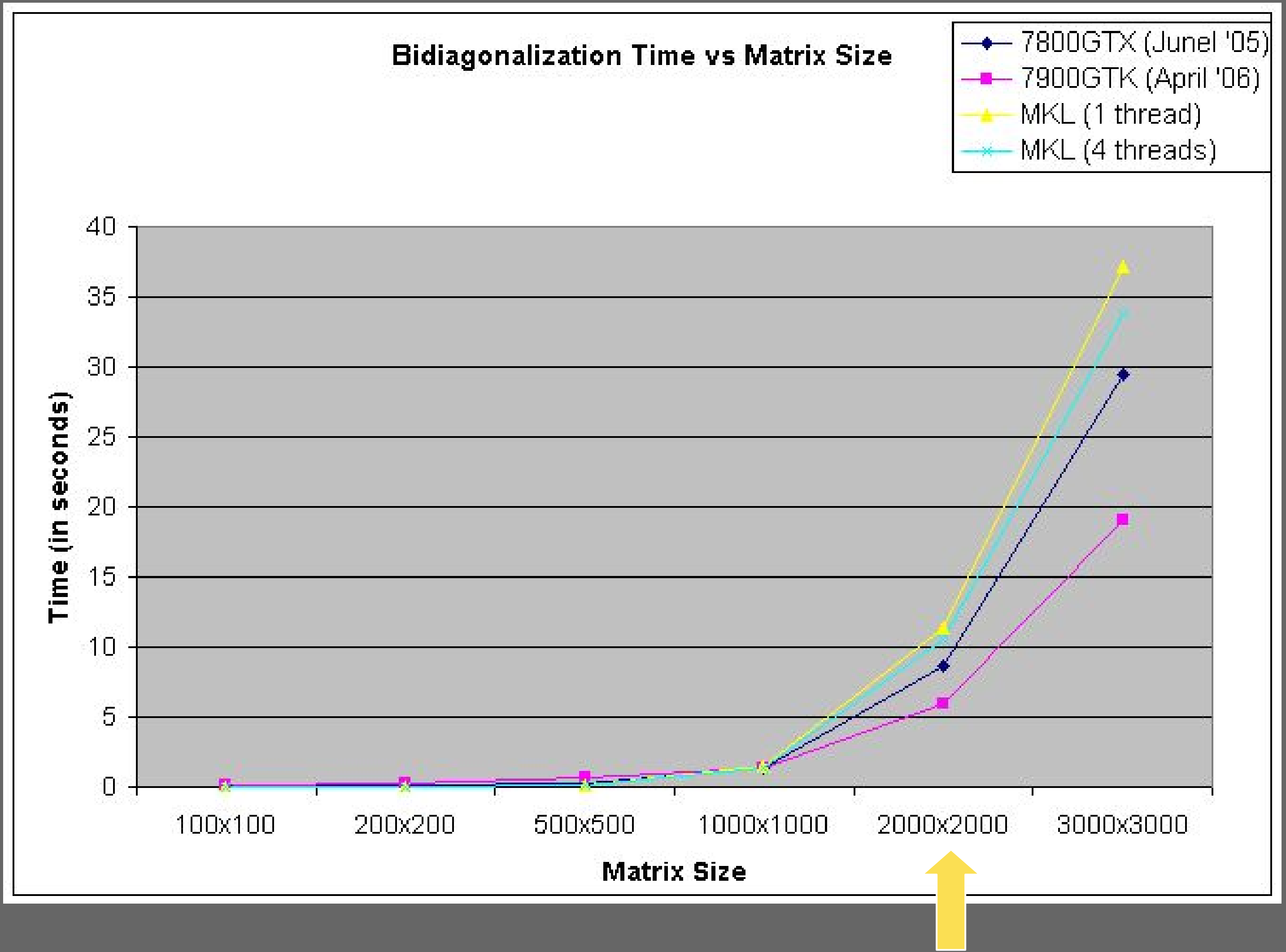
- We need to solve linear eqns.
- Considered 2 methods:
- Gaussian elimination vs. SVD
- Gaussian elimination is easy to implement in GPU.
- But:
  - Gaussian elimination requires the matrix to be a square matrix.
  - Gaussian elimination cannot give approximate solution unlike SVD which can give least square solution $x = A^{+} * b$

- Hence, we choose SVD.

Why we did SVD on CPU

- SVD algorithm is too complex.
- [Bondhugula et al] implemented SVD on GPU.
- Significant speedup only for large matrices (>= 2000x2000).
- We never have to compute SVD for such large matrices.

SVD on GPU


Bidiagonalization Time vs Matrix Size
7800GTX (Junel '05)
7900GTK (April '06)
MKL (1 thread)
MKL (4 threads)
Time (in seconds)
40
35
30
25
20
15
10
5
0
100x100
200x200
500x500
1000x1000
2000x2000
3000x3000
Matrix Size

GPU alpha blending

- Reason:
  - Data independency – best fit for parallel computation.
    - Each pixels is blended independently to each other.
      - Buffer(xi) = Alpha * Pi(xi) + (1 – Alpha) * Buffer(xi)
  - Fast data transfer rate in GPU : 42 GB/s to video memory of ATI X1800.

GPU alpha blending

However, instead of using linear blending, we employ Alpha Map Blending technique on GPU :

Without blending

With blending on GPU

GPU alpha blending

- Method : use alpha map as weight to "smoothly merge" the image into the buffer.
- Reason : to reduce sharp edges as well as less visual boundary artifact.

CPU – GPU conversion of alpha blending method

- Traditional algorithm:
  - For each pixel (pi) at Image i:
      - Get Alpha map weight → Alpha
    - IF position (xi) not in “blank” region:
      - Buffer(xi) = Alpha * Pi(xi) + (1 – Alpha) * Buffer(xi)
    - Else Buffer(xi) = Pi(xi) // copy image without blending

CPU – GPU conversion of alpha blending method

- Problem with GPU:
  - For each pixel (pi) at Image i:
    - Get Alpha map weight → Alpha
    - Float4 Blending = Alpha * Pi(xi) + (1 – Alpha) * Buffer(xi);

~~IF position (xi) not in "blank" region:~~

→ No "IF-Else" statement at GPU

- ~~Buffer(xi) = Alpha * Pi(xi) + (1 – Alpha) * Buffer(xi)~~

Else Buffer(xi) = Pi(xi)

CPU – GPU conversion of alpha blending method

1) GPU version for branching limitation:

- For each pixel (pi) at Image i:
  - Get Alpha map weight → Alpha
    - Float4 Blending = Alpha * Pi(xi) + (1 – Alpha) * Buffer(xi);

→GPU “IF - else” statement:

- Float t = step (0.0, Buffer) //return 1 if Buffer color NOT 0.0 and return 0 otherwise.
- OUTPUT = t * (Blending) + (1 - t) * Pi(xi)

CPU – GPU conversion of alpha blending method

2) Ping pong technique:

- "Buffer cannot be read and write and the same time."
  - Use 2 double buffers and pass data forward and backward between these two buffers and swap buffers after each loops.
  - (illustration here)

Our Results

- Mosaic of 9 image sequences (panning)

Our Results



Our Results

- Mosaic of 7 Images (Panning)

Our Results

- Tilting / Rotation of 4 Images

Our Results

- Vertical Translation of 5 Images

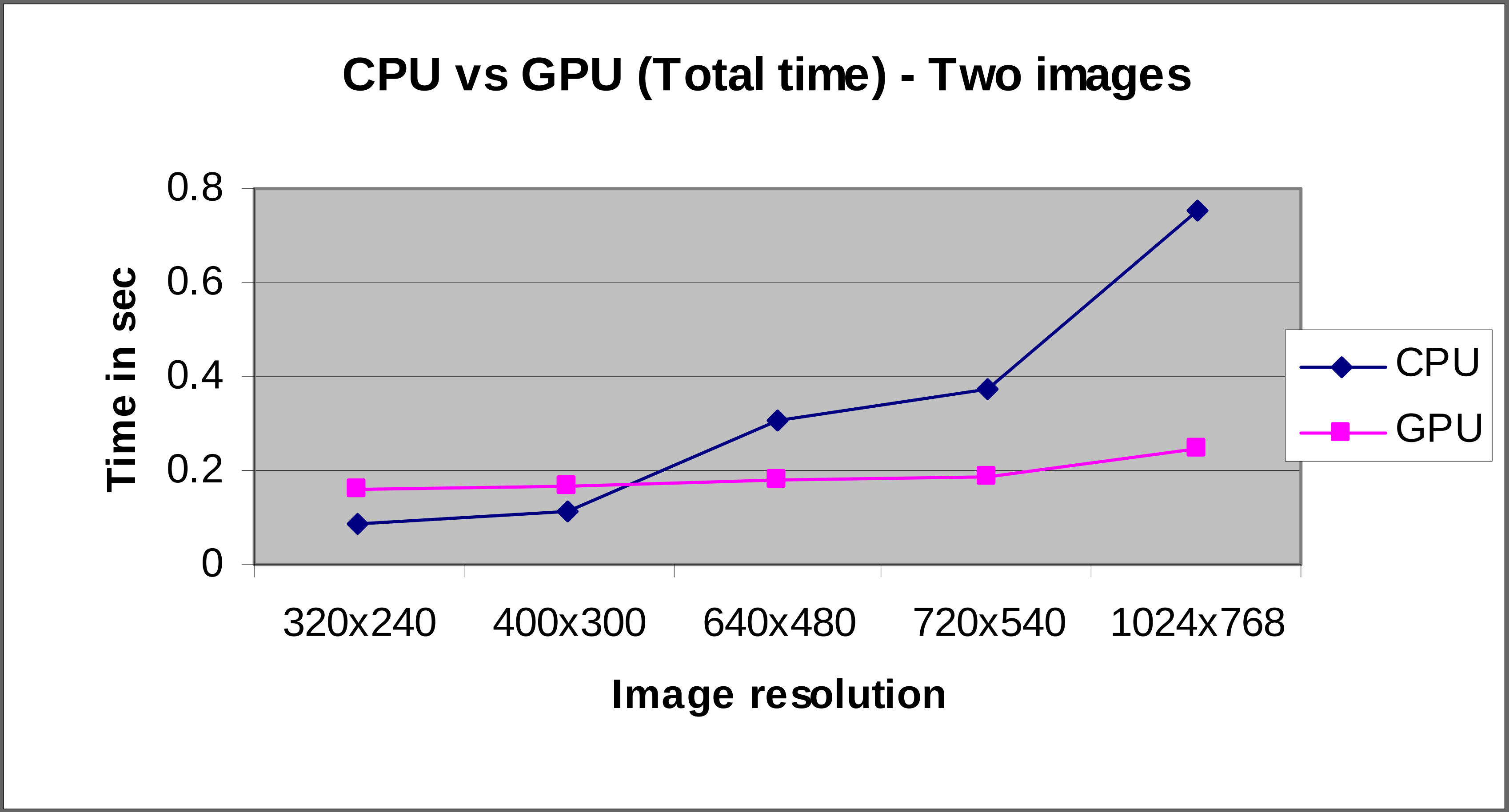
Our Results

- 5 Image Sequences of Zooming

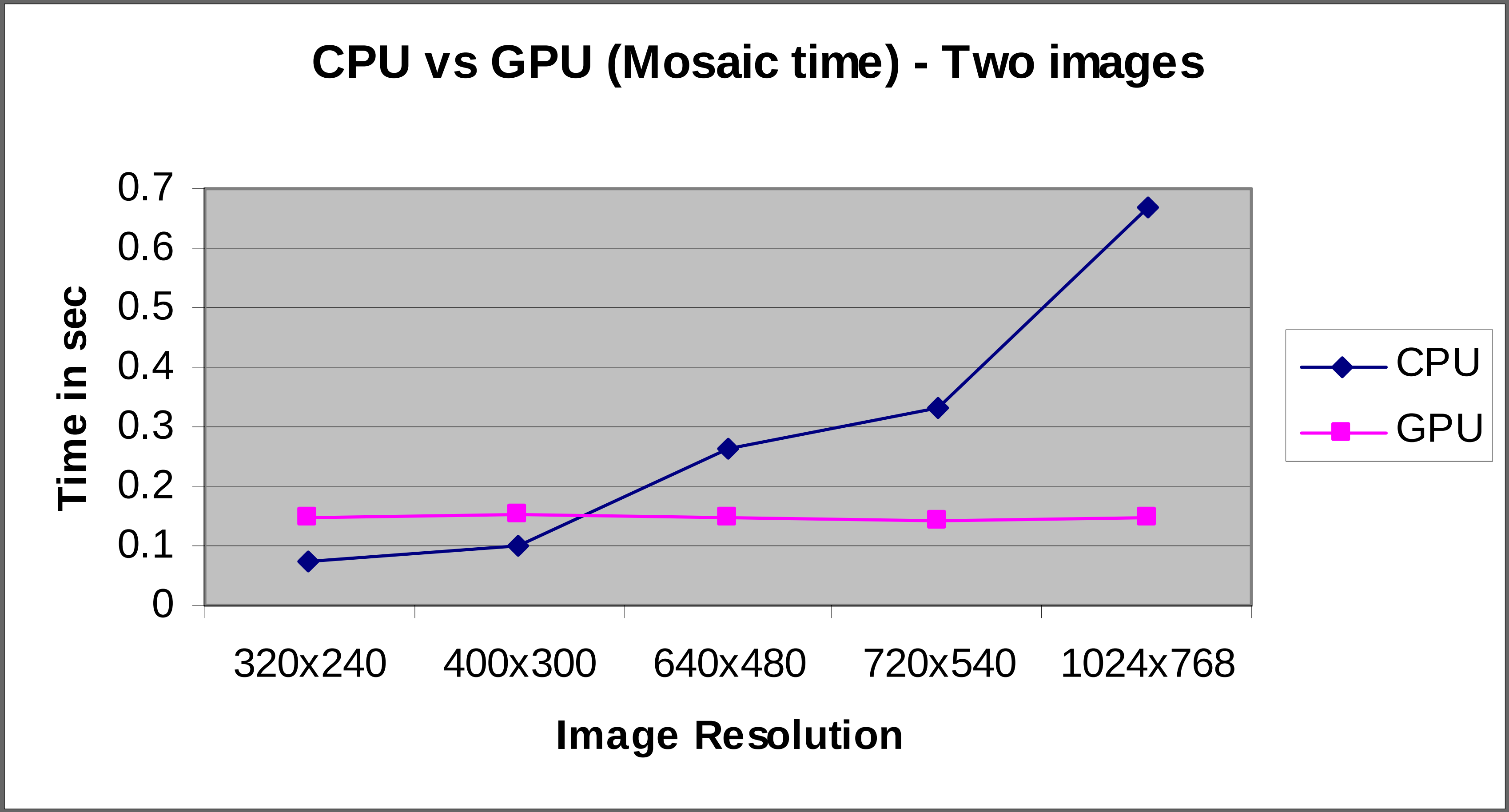
Our Results

- Translation, Rotation and Zooming of 24 Images

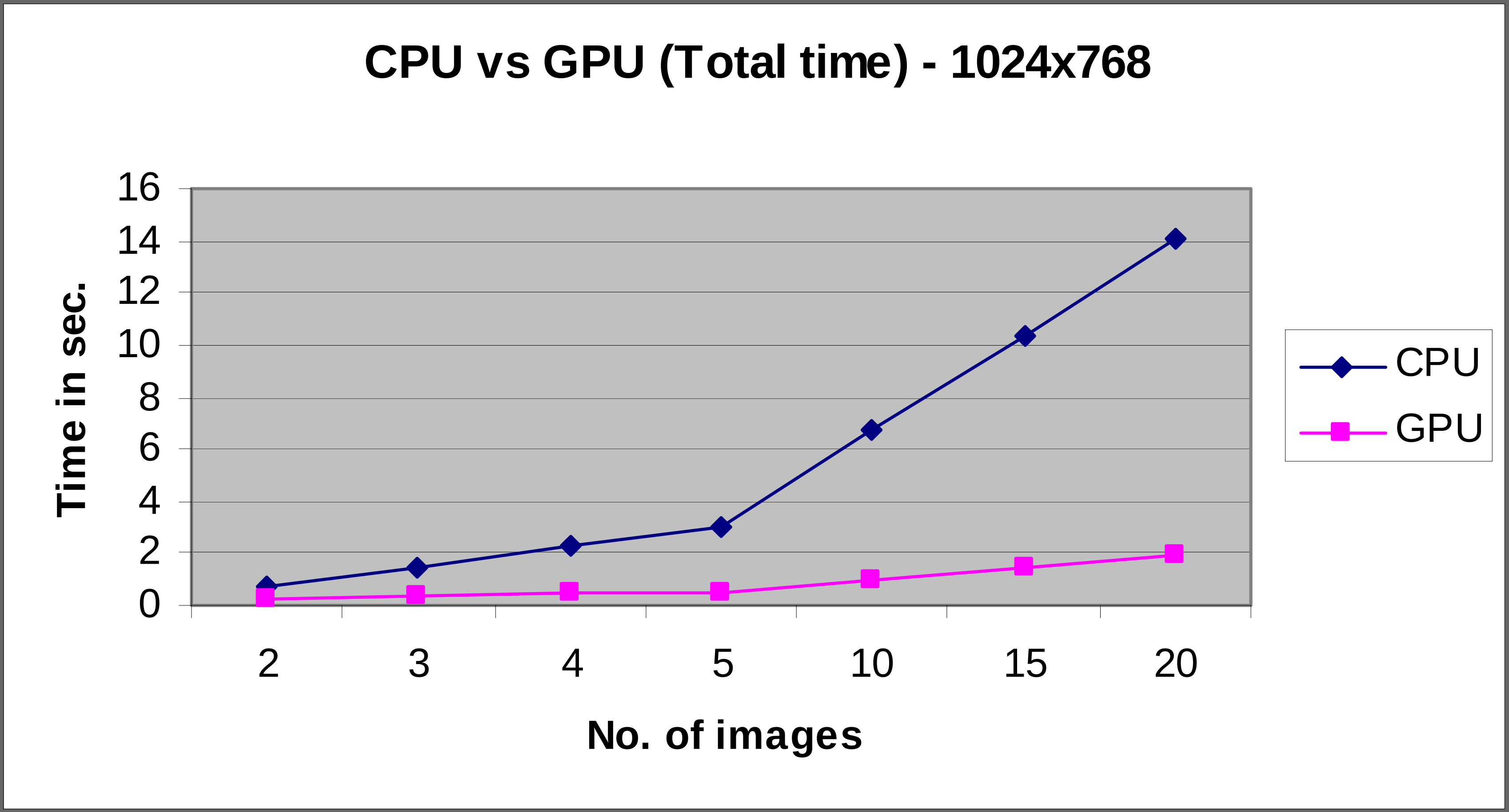
CPU vs. GPU


CPU vs GPU (Total time) - Two images
Time in sec
0.8
0.6
0.4
0.2
0
320x240
400x300
640x480
720x540
1024x768
Image resolution
CPU
GPU

CPU vs. GPU

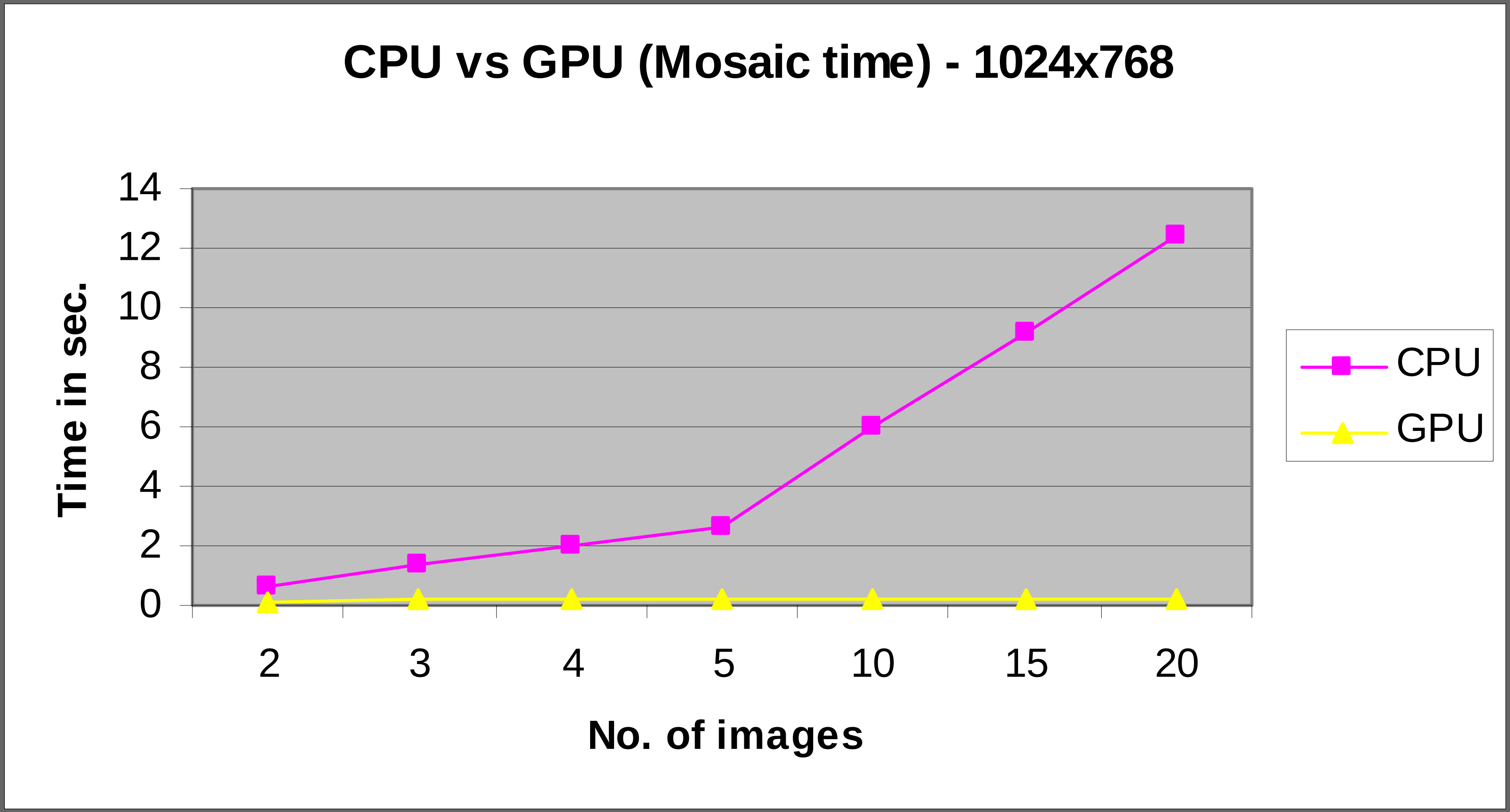

CPU vs GPU (Mosaic time) - Two images
Time in sec
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
320x240
400x300
640x480
720x540
1024x768
Image Resolution
CPU
GPU

CPU vs. GPU



CPU vs GPU (Total time) - 1024x768
Time in sec.
16
14
12
10
8
6
4
2
0
2
3
4
5
10
15
20
No. of images
CPU
GPU

CPU vs. GPU



CPU vs GPU (Mosaic time) - 1024x768
Time in sec.
14
12
10
8
6
4
2
0
2
3
4
5
10
15
20
No. of images
CPU
GPU

Comments on OpenVidia

- Feature Tracking does not work well.
- OpenVidia requires mid-to-high range NVIDIA graphics card.
- Hard to setup (dependencies: Glew, Glut, OpenGL, Cg).
- Extremely hard to compile from source (Pre-compiled binaries depend on MSVC 2005).
- Uses OpenGL stencil buffer.
- Size of input image has to be multiple of 4.
- Changes OpenGL internal state.
- Decided not to use OpenVidia.

Conclusion

- We have iteratively tried to solve Image Registration on GPU
- Using GPU for small problem is less efficient than using CPU
- We have studied the performance benefit of GPU on Image Registration
- Feature tracking – CPU
  Homography computation – CPU
- Registration, Alpha blending, bilinear interpolation - GPU

Real-time video demo

Thank you