Accelerating image registration on GPUs

Harald Köstler, Sunil Ramgopal Tatavarty

SIAM Conference on Imaging Science (IS10)
13.4.2010
Contents

- Motivation: Image registration with FAIR
- GPU Programming
- Combining Matlab and CUDA
- Future Work
Image registration in FAIR

MOTIVATION
Image Registration in FAIR

(a) $T(xc)$  
(b) $R(xc)$  
(c) $|T(xc) - R(xc)|$

(d) $T(xc)$ & grid $yc$  
(e) $T(yc)$  
(f) $|T(yc) - R(xc)|$
Variational model for image registration

Find mapping or deformation field

\[ u : \mathbb{R}^d \rightarrow \mathbb{R}^d, d \in \{2,3\} \]

such that the energy functional

\[ E(u) = D[J,u] + \alpha \cdot S[u] \]

with data term \( D \) and regularizer \( S \) is minimized.
FAIR: Flexible Algorithms for Image Registration

- Framework for image registration in Matlab
- Different transformations, data terms, regularizers
- Different (multi-level) numerical optimization techniques
- Constrained image registration
- FAIR, J. Modersitzki, SIAM, 2009
Image Registration cycle

Reference Image → Metric → Optimizer

Template Image → Interpolator → Transform

FAIR Implementation
Runtime distribution for rigid registration cycle

- Reference Image
- Template Image
- SSD
- Cubic Bspline
- Gauss Armijo
- Rigid

- FAIR Implementation

- Functional: 59%
- Optimization: 22%
- Interpolation: 15%
- Matlab plots: 4%
GPUs

COMPUTER ARCHITECTURE
Nvidia GeForce GTX 295

- **Costs**: 450 €
- **Interface**: PCI-E 2.0 x16
- **Shader Clock**: 1242 MHz
- **Memory Clock**: 999 MHz
- **Memory Bandwidth**: 2x112 GB/s
- **FLOPS**: 2x894 GFLOPS
- **Max Power Draw**: 289 W
- **Framebuffer**: 2x896 MB
- **Memory Bus**: 2x448 bit
- **Shader Processors**: 2x240
Why GPUs?

- Graphics Processing Units (GPUs) are a massively parallel computer architecture
- GPUs offer high computational performance at low costs
- **GPGPU**
  - General-Purpose computing on a GPU
  - Using graphic hardware for non-graphic computations
  - Programming Tools: CUDA or OpenCL
- Without explicitly parallel algorithms, the performance potential of current hardware architectures cannot be used any more
CPU vs. GPU

- CPUs are great for task parallelism
  - Fast caches
  - Branching adaptability

- GPUs are great for data parallelism
  - Multiple ALUs
  - Fast onboard memory
  - High throughput on parallel tasks
    - Executes program on each fragment

- Think of the GPU (device) as a massively-threaded co-processor
Memory Architecture

- Constant Memory
- Shared Memory
- Texture Memory
- Device Memory
What is CUDA?
- Compute Unified Device Architecture
- Software architecture for managing data-parallel programming

Write kernels (functions) that execute on the device and process multiple data elements in parallel

Massive threading

Local memory
Paradigm of the CUDA toolkit I

OpenMP
- Divide domain into huge chunks
- Equally distribute to threads
- Parallelize outer most loop to minimize overhead

Thread 0
Thread 1
Thread 2

0
1
2
3
4
5
CUDA

- Divide domain into small pieces

But!

- Data Mapping is different to en-block distribution of OpenMP
- Alignment constraints must be met
- No cache and cache lines need to be considered
Grid Size and Block Size

- Grid size: The size and shape of the data that the program will be working on
- Block size: The block size indicates the sub-area of the original grid that will be assigned to a multiprocessor
CUDA Example

GPU:

```c
__global__ void sum(float* a, float* b, float* c) {
    int idx = blockIdx.x*blockDim.x + threadIdx.x; // computes array index for current thread
    c[idx] = a[idx] + b[idx];                                         // adds array entry (global memory)
}
```

CPU:

```c
void vector_add(float *in_a, float *in_b, float *out, unsigned int length){
    float* a, *b, *c;

    cudaMalloc((void**)&a_GPU, sizeof(float)*length);
    cudaMalloc((void**)&b_GPU, sizeof(float)*length);
    cudaMalloc((void**)&out_GPU, sizeof(float)*length);

    cudaMemcpy ( a_GPU,in_a, sizeof ( float ) *length, cudaMemcpyHostToDevice );
    cudaMemcpy ( b_GPU,in_a, sizeof ( float ) *length, cudaMemcpyHostToDevice );
    cudaMemcpy ( out_GPU,out, sizeof ( float ) *length, cudaMemcpyDeviceToHost );

    dim3 dimblock(256,1);
    dim3 dimgrid(length/dimblock.x,1);

    sum<<<dimgrid,dimblock>>>(a,b,c);

    cudaMemcpy ( out_GPU,out, sizeof ( float ) *length, cudaMemcpyDeviceToHost );
}
```
OpenCL (Open Computing Language) is an open standard for heterogeneous parallel computing.

- Managed by non-profit technology consortium Khronos group
  - www.khronos.org/opencl

- Analogous to the open industry standards OpenGL for 3D graphics

- OpenCL exploits task-based and data-based parallelism

- Similar to CUDA, OpenCL includes a language (based on C99) for writing kernels executing on devices like GPU, cell, or multi-core processors

- Performance of high level heterogeneous tools?
COMBINING MATLAB AND CUDA
MATLAB MEX interface

MATLAB
A call to MEX-file func:

\[ \text{[C,D]} = \text{func(A,B)} \]
tells MATLAB to pass variables A and B to your MEX-file. C and D are left unassigned.

MATLAB
On return from MEX-file func:

\[ \text{[C,D]} = \text{func(A,B)} \]
plhs[0] is assigned to C and plhs[1] is assigned to D.

INPUTS

const mxArray *B
B = prhs[1]

const mxArray *A
A = prhs[0]

func.c

void mexFunction(
    int nlhs, mxArray *plhs[],
    int nrhs, const mxArray *prhs[])

In the gateway routine:

- Use the mxCreate functions to create the MATLAB arrays for your output arguments. Set plhs[0], [1], ... to the pointers to the newly created MATLAB arrays.
- Use the mxGet functions to extract your data from prhs[0], [1], ...
- Call your C subroutine passing the input and output data pointers as function parameters.

OUTPUTS

mxArray *D
D = plhs[1]

mxArray *C
C = plhs[0]
MATLAB script `nvmex.m` compiles CUDA code to create MATLAB function files

1. Allocate memory on the GPU
2. Transfer the data from the host to the GPU
3. Perform computation on GPU (library, custom code)
4. Transfer results from the GPU to the host
Basic interpolation schemes: Host side

texture<float, 2, cudaReadModeElementType> tex;

....

void mexFunction(int nlhs ...){
  ....
  // set texture parameters
  tex.addressMode[0] = cudaAddressModeClamp;
  tex.addressMode[1] = cudaAddressModeClamp;
  // access with normalized texture coordinates
  tex.normalized = false;
  ....
  // Bind the array to the texture
  cudaBindTextureToArray( tex, cu_array, channelDesc);
  ....
}
__global__ void Inter2DKernel()
{
    ....
    T = tex2D(tex, tx, ty);
    ....
}

**Nearest Neighbor**

\[ T_{nn}(x) = 0 \quad \text{for} \quad x \notin \Omega \]
\[ T_{nn}(x) := dataT(j) \]

```
tex.filterMode = cudaFilterModePoint;
```

**Low Precision Linear**

\[ T_{linear}(x) := dataT(p) \cdot (1 - \xi) + dataT(p + 1) \cdot \xi, \]

```
tex.filterMode = cudaFilterModeLinear;
```
B-Spline Interpolation [Sigg, C. and Hadwiger, M.]

Combine 2 linear interpolations to obtain B-spline interpolation

\[ T_{\text{spline}}(x) = c_{p-1} b(\xi + 1) + c_p b(\xi) + c_{p+1} b(\xi - 1) + c_{p+2} b(\xi - 2) \]

\[ T_{\text{linear}}(x) := \text{data}\, T(p) \cdot (1 - \xi) + \text{data}\, T(p + 1) \cdot \xi, \]

\[ (a + b) \cdot T_{\text{linear}}(x) := \text{data}\, T(p) \cdot a + \text{data}\, T(p + 1) \cdot b, \]

\[ T_{\text{spline}}(x) = g_0(\xi) \cdot c_{\text{linear}}^{p+h_0} + g_1(\xi) \cdot c_{\text{linear}}^{p+h_1} \]

where,

\[ g_0(\xi) = b(\xi + 1) + b(\xi) \]

\[ g_1(\xi) = b(\xi - 1) + b(\xi - 2) \]

\[ h_0 = \left( \frac{b(\xi)}{g_0(\xi)} \right) - 1 \]

\[ h_1 = \left( \frac{b(\xi - 2)}{g_1(\xi)} \right) + 1 \]
Runtime Results Interpolation I

Graph showing the relationship between Grid Size and Time (ms) for different interpolation methods:
- linearInter2D (FAIR)
- SplineInter2D (FAIR)
- SplineInter2D (NN texture)
- SplineInter2D (bilinear texture)
Runtime Results Interpolation II

Theoretical Best Case runtime

Theoretical worst Case runtime

SplineInter2D (bilinear texture)

Time (ms):
- 0.0001
- 0.001
- 0.01
- 0.1
- 1

Data points:
- 2048
- 8192
- 32768
- 131072
Discretization Error Accuracy

\[ |f+hf' - f(h)| \]

\[ |f - f(h)| \]

\[ h \]
Persistent Memory and Hybrid Memory

- Goal: avoid allocation and deallocation of memory for each call to a GPU kernel through Matlab
- Static variables
- Routine to clear CUDA MEX persistent variable
- Saves up to 17% runtime for bigger images (2048x1024)
CUDA MEX Registration cycle

CPU

- Reference Image
- Template Image
- FAIR/CUDA mem
- FAIR/CUDA mem
- Interpolator
- Metric
- Optimizer
- Transform

GPU

- Cuda Metric kernel functions
- Cuda Interpolator kernel functions
- Cuda Transform kernel functions

Existing FAIR Implementation
FAIR + CUDA Implementation
- Time includes data transfers and Matlab visualization (plots about 20%)
- GPU pays off for large images (speedup factor about 2x, pure GPU kernels about 20x)
- A lot more potential to move more parts to GPU

<table>
<thead>
<tr>
<th>xSize</th>
<th>ySize</th>
<th>Matlab</th>
<th>Matlab+GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>64</td>
<td>15 s</td>
<td>14 s</td>
</tr>
<tr>
<td>256</td>
<td>128</td>
<td>45 s</td>
<td>33 s</td>
</tr>
<tr>
<td>512</td>
<td>256</td>
<td>202 s</td>
<td>92 s</td>
</tr>
</tbody>
</table>
A Multigrid Solver for Diffusion-based PDEs

MULTIGRID ON GPU
What is Multigrid?

- May consume a lot of time during optimization step to find an updated transformation
- A very fast and efficient iterative solver for huge sparse, linear (and nonlinear) systems
  \[ Au = f \]
- Geometric multigrid on structured grids is well suited for GPU implementation because of regular data access patterns
- Multigrid exhibits a convergence rate that is independent of the number of unknowns \( N \), i.e. a multigrid solver has complexity \( O(N) \)
Multigrid Idea

Multigrid methods are based on two principles:

1. Smoothing property

Smooth error on fine grid
Multigrid Idea

Multigrid methods are based on two principles:

1. Smoothing property
2. Coarse grid principle

Approximate smooth error on coarser grids
Complex Diffusion on Nvidia GeForce GTX 295

Runtime on 2 x C2 Penryn (8 cores) @ 2.8 GHz, 8 x 22,4 GFLOPs is 1100 ms for 4096 x 4096 → Speedup factor 14
Summary and Future Work

- When to use GPGPU
  - Suitable algorithms?
  - Get the promised performance?
  - Get it at low effort and cost?

- Future Work
  - Multi-GPU
  - OpenCL
  - Adapt GPU multigrid to image registration
Questions?
Acknowledgements

- Nvidia Corporation
  - [developer.nvidia.com/CUDA](http://developer.nvidia.com/CUDA)
  - Technical Brief – Architecture Overview
  - CUDA Programming Guide

- Supercomputing 2008 Education Program: CUDA Introduction, Christian Trefftz / Greg Wolffe (Grand Valley State University)

- AMD/ATI
  - ATI Stream SDK User Guide
Possible Improvements
Runtime Results Interpolation II

The diagram shows a graph that plots the relationship between Grid Size and runtime for different methods.

- **Theoretical Best Case runtime**
- **Theoretical worst Case runtime**
- **SplineInter2D (bilinear texture)**

The y-axis represents the runtime in seconds, and the x-axis represents the Grid Size. The graph illustrates how runtime increases with grid size for different methods.
Full Multigrid Cycle

h

2h
Interpolation of solution \( u \)

4h
Exact solution

Smoothing

\[ \mu_{2h} \]

Restriction of residual \( r = f - Au \)

Interpolation of error and correction of solution \( u \)

V-cycle

\[ \mu_h \]
Multigrid on GTX 295 with red-black splitting

Image size

Runtime $V(2,2)$ in ms
Memory Bandwidth

In percent from maximum measured (rounded) streaming bandwidth (100 GB / s)

Percent of memory bandwidth

Image size

1024x1024 1024x2048 2048x2048 2048x4096 4096x4096
Frames per second for Image Stitching

- fps (stitching GTX 295)
- fps (solver GTX 295)
- fps (CPU)

CPU: Intel Core2 Quad Q9550@2.83GHz with OpenMP (4 cores)
Interpolation and Restriction