Parallel data processing on GPU and CPU using OpenCL

Koen Michielsen
Dept. of Imaging and Pathology & Medical Imaging Research Center,
KU Leuven, Belgium
Digital Breast Tomosynthesis

- Limited angle tomography: (depending on the vendor)
  - 11 to 25 exposures
  - Angular range: 15 to 50 degrees
Digital Breast Tomosynthesis

• Data (sinogram) size:
  3584 x 2816 x 25 angles x 16 bit ≈ 500 MB

• Typical image (reconstruction) size:
  3584 x 2816 x 45 x 32 bit ≈ 1.7 GB

• Example study: 12 patients, ± lesion, ± scatter
  → 48 reconstruction (10 iterations of MLTR)

  → Originally: Intel Xeon E5440 @ 2.8 GHz, 1 thread:
  24h / reconstruction
Projection / Backprojection

• MLTR update step:

\[ \Delta \mu_j = \frac{\sum_i l_{ij} (\hat{y}_i - y_i)}{\sum_i l_{ij} (\sum_k l_{ik}) \hat{y}_i} \]

• Main computational bottleneck: \( l_{ij} \)

• Sinogram elements \( N = 2.5 \times 10^8 \)
• Reconstruction elements \( M = 4.5 \times 10^8 \)
• Projection matrix elements \( M^*N = 1.1 \times 10^{17} \sim 10^{11} \neq 0 \)
Projection / Backprojection

(a) Pixel driven
(b) Ray driven
(c) Distance driven

Projection / Backprojection

(a) Pixel driven
(b) Ray driven
(c) Distance driven

Embarrassingly parallel

GPU Structure

- CPU: few large cores optimized for serial processing
- GPU: many small cores optimized for parallel performance
GPU Programming

• GPU’s are programmable from 1994 using graphics languages (Cg / HLSL / GLSL) on graphics objects (vertices, textures)
  → Problems need to be translated

• General purpose computing
  • CUDA (C++) released 2007
  • OpenCL (C) released 2008

• Choosing OpenCL or CUDA?
GPU Programming

- GPU’s are programmable from 1994 using graphics languages (Cg / HLSL / GLSL) on graphics objects (vertices, textures)
  → Problems need to be translated

- General purpose computing
  - CUDA (C++) released 2007
  - OpenCL (C) released 2008

- Choosing OpenCL or CUDA?
  → Vendor agnostic: allows processing on GPU / CPU / any hardware with drivers
  → Similar performance if optimized
OpenCL basics

- Host code
  - Interface with main software
  - Device management
  - Memory management
  - Just in time compilation
  - …

- Compute device (GPU/CPU) code
  - Performs actual parallel workload
  - C99 subset, including:
    - Vector types
    - Image manipulations
    - Math functions
    - …
Projection / Backprojection

- Distance driven implementation
Projection / Backprojection

• Distance driven implementation

• Input / output data:
  - image
  - sinogram
  - detector elements (corner coordinates)
  - source coordinates
  - image size
  - sinogram size
  - image offset
  - voxel size
Projection / Backprojection

- Distance driven implementation:
  - Memory access: gather versus scatter coalesced access

Projection
→ parallel over sinogram

- Check ray direction
- For each plane:
  - Determine intersections
  - Calculate weights
  - Loop over elements (read values from volume)
- Write value to sinogram
Projection / Backprojection

• Distance driven implementation:
  • Memory access: gather versus scatter coalesced access

Backprojection
  → parallel over sinogram

  • Check ray direction
  • Read value from sinogram
  • For each plane:
    • Determine intersections
    • Calculate weights
    • Loop over elements (write value to volume)
### Projection / Backprojection

- **Distance driven implementation:**
  - Memory access: gather versus scatter coalesced access

### Backprojection

→ parallel over volume

- **For each angle:**
  - Check ray direction
  - Determine intersections
  - Calculate weights
  - Loop over elements (read value from volume)
  - Write value to volume
• Distance driven implementation
• Memory access: gather versus scatter
  coalesced access

Figure: ‘OpenCL Optimization Strategies’, www.cmsoft.com.br
Projection / Backprojection

• Distance driven implementation
  • Memory access: gather versus scatter coalesced access

• Inefficient (atomic) implementation for backwards compatibility

Projection of 45 planes to 25 angles:
  - Parallel over sinogram: 8.2s

Backprojection from 25 angles to 45 planes:
  - Parallel over sinogram: 38.4 s
  - Parallel over image: 18.3 s
Projection / Backprojection

• Distance driven implementation
  • Tomosynthesis versus general detector geometry:
    • flat detector, parallel to volume
    • one ray direction

```c
#ifdef TOMOSYNTHESIS
    temp_float4 = ((corner1 + corner4) * 0.5f) - source;
#else
    temp_float4 = ((corner1 + corner2 + corner3 + corner4) * 0.25f) - source;
#endif
... 
corner1 -= source;
#ifdef TOMOSYNTHESIS
    corner2 -= source;
    corner3 -= source;
#else
    corner4 -= source;
#endif
```
Projection / Backprojection

- Distance driven implementation
  - Tomosynthesis versus general detector geometry:
    - flat detector, parallel to volume
    - one ray direction

- Kernel timings (no overhead):
  Projection of 45 planes to 25 angles:
  - General detector geometry
    10 s
  - Flat detector geometry
    7.6 s
Projection / Backprojection

- Distance driven implementation
- Fast mathematical functions: fused, native, half-precision

```

Math Built-in Functions [6.12.2] [9.5.2]
T is type float, optionally double, or half if the half extension is enabled. Tn is the vector form of T, where n is 2, 3, 4, 8, or 16.
T is Tn and Tn, Q is qualifier __global__, __local__, or __private.
HN indicates that half and native variants are available using only the float or floatn types by prepending "half_" or "native_" to the function name. Prototypes shown in brown text are available in half_ and native_ forms only using the float or floatn types.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T acos (T)</td>
<td>Arc cosine</td>
</tr>
<tr>
<td>T acosh (T)</td>
<td>Inverse hyperbolic cosine</td>
</tr>
<tr>
<td>T acosp (T x)</td>
<td>acos (x) / π</td>
</tr>
<tr>
<td>T asin (T)</td>
<td>Arc sine</td>
</tr>
<tr>
<td>T asinh (T)</td>
<td>Inverse hyperbolic sines</td>
</tr>
<tr>
<td>T asinpi (T x)</td>
<td>asin (x) / π</td>
</tr>
<tr>
<td>T atan (T y_over_x)</td>
<td>Arc tangent</td>
</tr>
<tr>
<td>T atan2 (T y, T x)</td>
<td>Arc tangent of y/x</td>
</tr>
<tr>
<td>T tanh (T)</td>
<td>Hyperbolic tangent</td>
</tr>
<tr>
<td>T tanhp (T)</td>
<td>tanh (x) / π</td>
</tr>
<tr>
<td>T tanh2pi (T x, T y)</td>
<td>atan2 (x / π)</td>
</tr>
<tr>
<td>T cbrt (T)</td>
<td>Cube root</td>
</tr>
<tr>
<td>T ceil (T)</td>
<td>Round to integer toward + infinity</td>
</tr>
<tr>
<td>T copysign (T x, T y)</td>
<td>x with sign changed to sign of y</td>
</tr>
<tr>
<td>T cos (T)</td>
<td>Cosine</td>
</tr>
<tr>
<td>T coss (T x)</td>
<td>cos (π x)</td>
</tr>
<tr>
<td>T half_divide (T x, T y)</td>
<td>x / y</td>
</tr>
<tr>
<td>T native_divide (T x, T y)</td>
<td>(T may only be float or floatn)</td>
</tr>
<tr>
<td>T erf (T)</td>
<td>Calculates error function of T</td>
</tr>
<tr>
<td>T erf (T)</td>
<td>Exponential function of T</td>
</tr>
<tr>
<td>T exp (T)</td>
<td>Exponential function of e</td>
</tr>
<tr>
<td>T exp2 (T)</td>
<td>Exponential function of 2</td>
</tr>
<tr>
<td>T exp10 (T)</td>
<td>Exponential function of 10</td>
</tr>
<tr>
<td>T expm1 (T x)</td>
<td>Exponential function of e -1</td>
</tr>
<tr>
<td>T fabs (T)</td>
<td>Absolute value</td>
</tr>
<tr>
<td>T fdim (T x, T y)</td>
<td>Positive difference between x and y</td>
</tr>
<tr>
<td>T fma (T a, T b, T c)</td>
<td>Multiply and add, then round</td>
</tr>
<tr>
<td>T fmax (T x, T y)</td>
<td>Round to integer toward - infinity</td>
</tr>
<tr>
<td>T fmin (T x, T y)</td>
<td>Round to integer toward + infinity</td>
</tr>
<tr>
<td>T floor (T)</td>
<td>Round to integer toward - infinity</td>
</tr>
<tr>
<td>T fract (T x, Q T *ptr)</td>
<td>Fractional value in x</td>
</tr>
<tr>
<td>T frexp (T x, Q int *exp)</td>
<td>Extract mantissa and exponent</td>
</tr>
<tr>
<td>T hypot (T x, T y)</td>
<td>Square root of x^2 + y^2</td>
</tr>
<tr>
<td>T ln (T)</td>
<td>Natural logarithm</td>
</tr>
<tr>
<td>T log (T)</td>
<td>Base 2 logarithm</td>
</tr>
<tr>
<td>T log2 (T)</td>
<td>Base 2 logarithm</td>
</tr>
<tr>
<td>T log10 (T)</td>
<td>Base 10 logarithm</td>
</tr>
<tr>
<td>T log1p (T x)</td>
<td>In (1.0 + x)</td>
</tr>
<tr>
<td>T logb (T x)</td>
<td>Exponent of x</td>
</tr>
<tr>
<td>T mad (T a, T b, T c)</td>
<td>Approximates a * b + c</td>
</tr>
<tr>
<td>T maxmag (T x, T y)</td>
<td>Maximum magnitude of x and y</td>
</tr>
<tr>
<td>T minmag (T x, T y)</td>
<td>Minimum magnitude of x and y</td>
</tr>
<tr>
<td>T modf (T x, Q T *ptr)</td>
<td>Decompose floating-point number</td>
</tr>
<tr>
<td>T pow (T x, T y)</td>
<td>Compute x to the power of y</td>
</tr>
<tr>
<td>T pow2 (T x, T y)</td>
<td>Compute 2^x, where y is an integer</td>
</tr>
<tr>
<td>T pow2p (T x, T y)</td>
<td>Compute 2^x, where y is &gt;= 0</td>
</tr>
<tr>
<td>T pow2p (T x, T y)</td>
<td>Compute 2^x, where y is &gt;= 0</td>
</tr>
<tr>
<td>T round (T)</td>
<td>Integer value nearest to x</td>
</tr>
<tr>
<td>T rsqrt (T)</td>
<td>Inverse square root</td>
</tr>
<tr>
<td>T sin (T)</td>
<td>Sine</td>
</tr>
<tr>
<td>T sincos (T x, Q T *cosval)</td>
<td>Sine and cosine of x</td>
</tr>
<tr>
<td>T sinh (T)</td>
<td>Hyperbolic sine</td>
</tr>
<tr>
<td>T sinpl (T x)</td>
<td>sin (π x)</td>
</tr>
<tr>
<td>T sqrt (T)</td>
<td>Square root</td>
</tr>
<tr>
<td>T tan (T)</td>
<td>Tangent</td>
</tr>
<tr>
<td>T tanh (T)</td>
<td>Hyperbolic tangent</td>
</tr>
<tr>
<td>T tanpl (T x)</td>
<td>tan (π x)</td>
</tr>
<tr>
<td>T tgamma (T)</td>
<td>Gamma function</td>
</tr>
<tr>
<td>T trunc (T)</td>
<td>Round to integer toward zero</td>
</tr>
</tbody>
</table>
```

---
Projection / Backprojection

- Distance driven implementation
  - Fast mathematical functions: fused, native, half-precision
    - Fused operations:
      \[ \text{mad}(a, b, c) = a \times b + c \]
    - Native operations:
      uses hardware optimized instructions
    - Half-precision operations:
      uses 16 bit precision instead of 32 bit

→ Not necessarily IEEE 754 compliant!
Projection / Backprojection

- Distance driven implementation
  - Fast mathematical functions: fused, native, half-precision

```c
#ifdef FAST_MATH
  plane_p.z = mad(this_vox_z + 0.5f, vox_size.z, img_offset.z);
#else
  plane_p.z = (this_vox_z + 0.5f) * vox_size.z + img_offset.z;
#endif
...
```

```c
#ifdef FAST_MATH
  temp_float4 = native_recip(temp_float4);
#else
  temp_float4 = 1.0f / temp_float4;
#endif
```
Projection / Backprojection

- Distance driven implementation
  - Fast mathematical functions: fused, native, half-precision

- Kernel timings (no overhead):
  Projection of 45 planes to 25 angles:
    - Without ‘fast math’
      7.6 s
    - With ‘fast math’
      6.4 s
More Optimizations?

**Figure: XKCD 1205**

How long can you work on making a routine task more efficient before you're spending more time than you save? (Across five years)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Second</td>
<td>1 Day</td>
<td>2 Hours</td>
<td>30 Minutes</td>
<td>4 Minutes</td>
<td>1 Minute</td>
<td>5 Seconds</td>
</tr>
<tr>
<td>5 Seconds</td>
<td>5 Days</td>
<td>12 Hours</td>
<td>2 Hours</td>
<td>21 Minutes</td>
<td>5 Minutes</td>
<td>25 Seconds</td>
</tr>
<tr>
<td>30 Seconds</td>
<td>4 Weeks</td>
<td>3 Days</td>
<td>12 Hours</td>
<td>2 Hours</td>
<td>30 Minutes</td>
<td>2 Minutes</td>
</tr>
<tr>
<td>1 Minute</td>
<td>8 Weeks</td>
<td>6 Days</td>
<td>1 Day</td>
<td>4 Hours</td>
<td>1 Hour</td>
<td>5 Minutes</td>
</tr>
<tr>
<td>5 Minutes</td>
<td>9 Months</td>
<td>6 Months</td>
<td>5 Weeks</td>
<td>5 Days</td>
<td>1 Day</td>
<td>2 Hours</td>
</tr>
<tr>
<td>30 Minutes</td>
<td>10 Months</td>
<td>2 Months</td>
<td>10 Days</td>
<td>2 Days</td>
<td>5 Hours</td>
<td>2 Minutes</td>
</tr>
<tr>
<td>1 Hour</td>
<td>2 Months</td>
<td>2 Weeks</td>
<td>1 Day</td>
<td>2 Hours</td>
<td>5 Hours</td>
<td>2 Minutes</td>
</tr>
<tr>
<td>6 Hours</td>
<td>2 Months</td>
<td>2 Weeks</td>
<td>1 Day</td>
<td>2 Hours</td>
<td>5 Hours</td>
<td>2 Minutes</td>
</tr>
<tr>
<td>1 Day</td>
<td>8 Weeks</td>
<td>5 Days</td>
<td>5 Days</td>
<td>5 Days</td>
<td>5 Days</td>
<td>5 Days</td>
</tr>
</tbody>
</table>
More Optimizations?

• Using constant memory
  • pre-cached on GPU

• Using vector types
  • 1 float4 operation instead of 4 float operations (superfluous when using a smart compiler)

• Unrolling loops / fixed length loops
  • Branching is expensive on GPU

• Use #define macros to pass variables at compilation
  • Literal values are more efficient than variables

• Proper local memory use
  • Local memory is much faster than global memory

• …
More Optimizations?

- Using constant memory
- Using vector types
- Unrolling loops / fixed length loops
- Use `#define` macros to pass variables at compilation
- Proper local memory use
- ...

However: balance optimizations against flexibility

- Tomosynthesis only **OR** all projection geometries
- GPU and / or CPU optimization
Conclusions

• Example study: 12 patients, ± lesion, ± scatter
  → 48 reconstruction (10 iterations of MLTR)

→ Originally (Intel Xeon E5440 @ 2.8 GHz, 1 thread):
  24h / reconstruction

→ Currently (without ‘fast math’):
  - AMD Opteron 6166 HE @ 1.8 GHz, 32 threads:
    50m / reconstruction
  - nVidia Tesla C2075 @ 1.15 GHz, 448 threads
    8m / reconstruction
Thanks!
OpenCL Implementation

- Some benchmarks:

  ![Graph showing performance comparison between different devices and configurations.]

- Einstein
  - CPU: Intel Xeon E5440 @ 2.83 GHz

- Workstation
  - CPU: Intel Xeon E5606 @ 2.13 GHz
  - GPU: Nvidia Tesla C2075

- Avalok
  - CPU: AMD Opteron 6128HE @ 2.0 GHz