Image Processing in Java
Running on GPU

Enabling Accelerators in Java
Automatic Kernel Generation
State-Of-The-Art Performances
Image processing use-case...

**OIL AND GAS: COHERENCE IMAGE PROCESSING ALGORITHM**
Coherence Image Processing Algorithm

- Coherence algorithm measures correlation between adjacent seismic traces. It is used to detect faults in the underground terrain.
- Coherence is one of the filters used by TOTAL S.A. for seismic image analysis.
- Another algorithm, Gradient Correlation Cube, is very similar computationally and developed by TOTAL S.A. (Patent: EP12306380.2)
- For each element in a neighborhood, the maximum correlation between shifted vertical slices is calculated.
- This algorithm is neither linear nor separable.
- The filter parameters are the size of the vertical slice, the size of the neighborhood and the maximum explored shift.
Coherence Image Processing Algorithm

Input is a cube of scalar floating-point values

\[ a[n] = d[i, j, k + n] \]

\[ b[n] = d[i + ii, j + jj, k + n + shift] \]

Correlation \[1\]

\[
\bigwedge_{i,j}[ii, jj, shift] = \sum_{n=-w}^{w} \frac{(a[n] - \langle a \rangle)(b[n] - \langle b \rangle)}{\sqrt{\langle a^2 \rangle} \sqrt{\langle b^2 \rangle}}
\]

Coherence Output \[1\]

\[
dout[i, j, k] = \sum_{ii=-ws}^{ws} \sum_{jj=-ws}^{ws} \max_{shift} \bigwedge_{i,j}[ii, jj, shift]
\]

Output is a cube of scalar floating-point values

Filter parameters: w, ws, shift

[1] - Seismic Attributes for Prospect Identification and Reservoir Characterization, Satinder Chopra et Kurt J. Marfurt
Issues and Pitfalls

• In a brute-force approach, data is explored an immense number of times (several hundreds):
  – Neighborhood search
  – Vertical slice size
  – Vertical slice shift

• The main difficulty is to express the algorithm to reduce this data read overflow.

• We present here different approaches to reduce the number of data reads and maximize data reuse.
Why Java

• Java has been around for very long
• Advanced graphical user interfaces can be built
• Rich runtime for complete application implementation
  – Network interaction
  – File management
  – Serialization
  – GUI
• Advanced tools in IDE
  – Unit testing
  – Refactoring
• Garbage collected
• Efficient JIT
• Multiplatform

Java is used in TOTAL S.A. image processing software that hosts filters such as Coherence and Gradient Correlation Cube

© Altimesh 2016 – all rights reserved
From Offline Processing To Interactive Feedback

OPTIMIZATION STEPS
The Journey From Source Code To Binary

• One may think that the language used for development will enforce performance level of the resulting binary. This is much less true than we may think.

• Java uses JIT which transforms java byte-code into binary, with advanced optimizations.

• This transformation can be taken further: re-expressing the byte-code in another language, and recompile.

Source
• Java
• DotNet

Byte-Code
• JVM
• MSIL

Other Source
• C/C++
• CUDA/C

Assembly
• X86
• PTXAS
Naive Approach

• Code written by image processing researcher developing the algorithm: no platform-dependent optimization
• Loop on Z,Y,X; then ii, jj, and n

<table>
<thead>
<tr>
<th></th>
<th>Java</th>
<th>Cuda (GTX Titan)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive</td>
<td>7500 s</td>
<td>85.0 s</td>
</tr>
</tbody>
</table>

• Performance expressed in seconds on a 512x512x512 grid of data, in single precision

• TOTAL S.A. is willing to have interactive feedback on parameter changes
Grouping Loops

• Some loops traverse same data and output some aggregate value
  – Merge loops on n to avoid multiple loading of same data
  – Calculate average, norm and correlation in single loop

• Extract loop invariants
  – Avoid recalculating costly operations
    • Inverse is less expensive than global load
    • Inverse is more expensive than register load

<table>
<thead>
<tr>
<th></th>
<th>Java</th>
<th>Cuda (GTX Titan)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive</td>
<td>7500 s</td>
<td>85.0 s</td>
</tr>
<tr>
<td>Grouping Loops</td>
<td>6750 s</td>
<td>47.7 s</td>
</tr>
</tbody>
</table>
Processing Several Items at a Time

- Processing several Z-planes in single pass
  - Further reduce data loading
  - At the expense of code *readability*

- This further improves benefits from Loop grouping approach

<table>
<thead>
<tr>
<th></th>
<th>Java</th>
<th>Cuda (GTX Titan)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive</td>
<td>7500 s</td>
<td>85.0 s</td>
</tr>
<tr>
<td>Grouping Loops</td>
<td>6750 s</td>
<td>47.7 s</td>
</tr>
<tr>
<td>MultiZ</td>
<td>2650 s</td>
<td>11.5 s</td>
</tr>
</tbody>
</table>
Mapping Arrays On Registers

- GPUs have a big flexible register file: up to 255 registers per context
- Changing loops
  - Does not reduce *lisibility* of code, yet much closer to original code than MultiZ
  - Could have been a first draft
- Using stackalloc for temporary buffers
  - Easily expressed in Java

- Using NVRTC to unroll loops completely: loop range parameter is a filter parameter (3 of them) => on the fly compilation yields register-only code !!
Java & NVRTC Build and Run Chain

**Compile time**

- **Java code**
  - Algorithm
  - Hybridizer annotations
  - Identify compile-time constants

- **JVM Bytecode**
  - Regular javac

- **Cuda code**
  - Hybridizer
  - Maven plugin

**Runtime**

- **Java code**
  - Client
  - Use nvrtc and driver API via runtime library
  - Call native code using JNA
  - Work distribution (grid and block sizes)

- **NVRTC**
  - Parameters passed as Cuda preprocessor macros
  - -D<parameter name>=<value>
  - Allows unrolling
  - Compilation time is 0.6 ms in this case (Core i7 Haswell)

- **No compromise on performance**
  - Keeping flexibility on parameters values
  - Compilation can be cached to reduce overhead
## Mapping Arrays On Registers

<table>
<thead>
<tr>
<th>Time in seconds</th>
<th>Java</th>
<th>Cuda (GTX Titan)</th>
<th>NVRTC (compile time)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive</td>
<td>7500 s</td>
<td>85.0 s</td>
<td></td>
</tr>
<tr>
<td>Grouping Loops</td>
<td>6750 s</td>
<td>47.7 s</td>
<td></td>
</tr>
<tr>
<td>MultiZ</td>
<td>2650 s</td>
<td>11.5 s</td>
<td></td>
</tr>
<tr>
<td>Registrified</td>
<td>N/A</td>
<td>2.84 s</td>
<td>(+) 0.6 s</td>
</tr>
</tbody>
</table>

Core i7 – 4771 @ 3.5 GHz  
Nvidia GTX Titan (Kepler)  
JDK 1.7.0_40  
Cuda 7.5

Memory transfer ~ 0.2 s
Combining MultiZ and Registrified

<table>
<thead>
<tr>
<th>Time in seconds</th>
<th>Java</th>
<th>Cuda (GTX Titan)</th>
<th>NVRTC (compile time)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive</td>
<td>7500 s</td>
<td>85.0 s</td>
<td></td>
</tr>
<tr>
<td>Grouping Loops</td>
<td>6750 s</td>
<td>47.7 s</td>
<td></td>
</tr>
<tr>
<td>MultiZ</td>
<td>2650 s</td>
<td>11.5 s</td>
<td></td>
</tr>
<tr>
<td>Registrified</td>
<td>N/A</td>
<td>2.84 s</td>
<td>(+) 0.6 s</td>
</tr>
<tr>
<td>MultiZ Registrified</td>
<td>N/A</td>
<td>1.07 s</td>
<td>(+) 0.7 s</td>
</tr>
</tbody>
</table>

Core i7 – 4771 @ 3.5 GHz
Nvidia GTX Titan (Kepler)
JDK 1.7.0_40
Cuda 7.5

Memory transfer ~ 0.2 s
On usage of intrinsic functions

- Using intrinsic might lead to a stalling pipeline
  - Using intrinsic for \texttt{frsqrt} does not adequately utilize the resources of the processor
  - Intuitive code and fastmath compilation performs better

\begin{itemize}
  \item Using intrinsic (\texttt{__frsqrt\_rn})
  \item No intrinsic (1 / \texttt{sqrtf})
\end{itemize}
How Close From Peak Usage?

- GTX 980 - CUDA 7.5
How Close From Peak Usage?

- GTX 980 - CUDA 7.5
- Pipe Utilization:
  - Arithmetic: 89.6%
How Close From Peak Usage?

- GTX 980 - CUDA 7.5
- Pipe Utilization:
  - Arithmetic: 89.6%
- Next in line:
  - Instruction fetch?
How Close Are We From Optimum?

DISCUSSION
Theoretical Peak?

<table>
<thead>
<tr>
<th>operation</th>
<th>Count / grid size</th>
<th>sample</th>
<th>OP</th>
<th>Per entry FLOP</th>
<th>GLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;&gt;</td>
<td>(2w+1)</td>
<td>11</td>
<td>ADD</td>
<td>1</td>
<td>1,44</td>
</tr>
<tr>
<td>&lt;&gt;²</td>
<td>(2w+1)</td>
<td>11</td>
<td>FMA</td>
<td>2</td>
<td>2,89</td>
</tr>
<tr>
<td>1/SQRT</td>
<td>(2w+1)</td>
<td>11</td>
<td>RSQRT</td>
<td>15</td>
<td>21,67</td>
</tr>
<tr>
<td>correl</td>
<td>(1/2 \times (1+2<em>ws)^2 \times (1+2</em>(shift)))</td>
<td>269,5</td>
<td>FMA</td>
<td>22</td>
<td>1557,60</td>
</tr>
<tr>
<td><strong>OPTIMAL FLOPS</strong></td>
<td></td>
<td></td>
<td></td>
<td>1583,60</td>
<td></td>
</tr>
</tbody>
</table>

In our sample, we need to perform 1.584 TFLOP at the minimum.
## Theoretical Peak?

<table>
<thead>
<tr>
<th>operation</th>
<th>Count / grid size</th>
<th>sample</th>
<th>OP</th>
<th>Per entry FLOP</th>
<th>GLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;&gt;</td>
<td>(2w+1)</td>
<td>11</td>
<td>ADD</td>
<td>1</td>
<td>1,44</td>
</tr>
<tr>
<td>&lt;&gt;²</td>
<td>(2w+1)</td>
<td>11</td>
<td>FMA</td>
<td>2</td>
<td>2,89</td>
</tr>
<tr>
<td>1/SQRT</td>
<td>(2w+1)</td>
<td>11</td>
<td>RSQRT</td>
<td>15</td>
<td>21,67</td>
</tr>
<tr>
<td>correl</td>
<td>1/2 * (1+2<em>ws)² * (1+2</em>(shift))</td>
<td>269,5</td>
<td>FMA</td>
<td>22</td>
<td>1557,60</td>
</tr>
</tbody>
</table>

**OPTIMAL FLOPS**

1583,60

**OPTIMAL NO SYMETRY**

3141,20

In our sample, we need to perform 1.584 TFLOP at the minimum. Moreover, if we don’t benefit from symmetry, we need 3.14 TFLOP.
Theoretical Peak ?

<table>
<thead>
<tr>
<th></th>
<th>GTX Titan</th>
<th>Jetson TX1</th>
<th>GTX 850m</th>
<th>GTX 980</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best time (s)</td>
<td>1.07</td>
<td>7.075</td>
<td>2.86</td>
<td>0.809</td>
</tr>
<tr>
<td>Compute Capability</td>
<td>3.5</td>
<td>5.3</td>
<td>5.0</td>
<td>5.2</td>
</tr>
<tr>
<td>Ratio peak (no sym)</td>
<td>45 %</td>
<td>87 %</td>
<td>95 %</td>
<td>84 %</td>
</tr>
</tbody>
</table>

- Hardly possible to use symmetry
  - *Average of max is not commutative*
- Naïve implementation would require very high bandwidth and low latency memory
Thank You

florent.duguet@altimesh.com
guillaume.de-roujoux@altimesh.com
http://www.altimesh.com