JPEG 2000
Discrete Wavelet Transform
by Christof Kobylko

Seminar „Multi-Core Architectures and Programming“, 17.07.2013
Table of Contents

● JPEG-2000
  ● Overview of JPEG-2000
  ● Idea of Discrete Wavelet Transform (DWT)
  ● DWT-Example

● Optimizing DWT for CPU-based computation
  ● Definition of DWT
  ● Lifting Scheme
  ● Further optimizations
  ● Benchmarks

● Optimizing DWT for GPU-based computation
  ● GPU-specific adaption of CPU-implementation
  ● Benchmarks

● Summary

● Literature
Table of Contents

- JPEG-2000
  - Overview of JPEG-2000
  - Idea of Discrete Wavelet Transform (DWT)
  - DWT-Example
- Optimizing DWT for CPU-based computation
  - Definition of DWT
  - Lifting Scheme
  - Further optimizations
  - Benchmarks
- Optimizing DWT for GPU-based computation
  - GPU-specific adaption of CPU-implementation
  - Benchmarks
- Summary
- Literature
Overview of JPEG-2000

- Image compression standard, successor to the popular JPEG
- Codec processing chain:

  ![Codec Processing Chain Diagram]

- **Topic of this talk:** Forward Discrete Wavelet Transform
- The results of this project are compared to the JPEG-2000 reference implementation **OpenJPEG**
Idea of Discrete Wavelet Transform (DWT)

- Iterative procedure ⇒ at each step:
  - Filter the image by 2D-lowpass and highpass filter
  - Sub-sample the results by factor 2
  - Decompose the image into 4 sub-bands (LL, LH, HL, HH)

⇒ Huge decrease in entropy in high-frequency bands (LH, HL, HH)
DWT-Example

Original Image:
DWT-Example (2)

DWT with 2 resolution levels:
DWT-Example (3)

DWT with 3 resolution levels:
DWT-Example (4)

DWT with 4 resolution levels:
DWT-Example (5)

DWT with 5 resolution levels:

⇒ And everything is still lossless!
# Table of Contents

- JPEG-2000
  - Overview of JPEG-2000
  - Idea of Discrete Wavelet Transform (DWT)
  - DWT-Example
- Optimizing DWT for CPU-based computation
  - Definition of DWT
  - Lifting Scheme
  - Further optimizations
  - Benchmarks
- Optimizing DWT for GPU-based computation
  - GPU-specific adaption of CPU-implementation
  - Benchmarks
- Summary
- Literature
Definition of DWT

- 2D-convolution with a lowpass and a highpass filter
- Filters are separable
  - Decompose filtering into a vertical and a horizontal 1D-convolution
- Coefficients for the used Cohen-Daubechies-Feauveau 9/7 wavelets:

<table>
<thead>
<tr>
<th>n</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lowpass $h_L[n]$</td>
<td>0.602949</td>
<td>0.266864</td>
<td>-0.078223</td>
<td>-0.016864</td>
<td>0.026749</td>
</tr>
<tr>
<td>Highpass $h_H[n]$</td>
<td>0.557543</td>
<td>0.295636</td>
<td>-0.028772</td>
<td>-0.045636</td>
<td>0</td>
</tr>
</tbody>
</table>

- Filters are symmetric
  - Efficient implementation of the convolution possible:

$$o_L[n] = i[2n] \cdot h_L[0] + \sum_{k=1}^{4} (i[2n+k] + i[2n-k]) \cdot h_L[k]$$
$$o_H[n] = i[2n+1] \cdot h_H[0] + \sum_{k=1}^{3} (i[2n+1+k] + i[2n+1-k]) \cdot h_H[k]$$

⇒ 11 memory transactions, 14 additions and 9 multiplications per output pixel pair
Lifting Scheme

- Technique to reduce computational complexity of convolutions with symmetric filters (very simplified formulation)
- Approach: Rewrite convolution operation in series of “lifting steps”

\[
\begin{align*}
    d_1[n] &= d_0[n] - \alpha_0 \cdot (s_0[n+1] + s_0[n]) \\
    s_1[n] &= s_0[n] - \alpha_1 \cdot (d_1[n] + d_1[n-1]) \\
    d_2[n] &= d_1[n] + \alpha_2 \cdot (s_1[n+1] + s_1[n]) \\
    s_2[n] &= s_1[n] + \alpha_3 \cdot (d_2[n] + d_2[n-1]) \\
    o_L[n] &:= s[n] = \beta_0 \cdot s_2[n] \\
    o_H[n] &:= d[n] = \beta_1 \cdot d_2[n]
\end{align*}
\]

\[
\begin{array}{|c|c|c|c|c|}
\hline
n & 0 & 1 & 2 & 3 \\
\hline
\alpha_n & 1.586134 & 0.052980 & 0.882911 & 0.443506 \\
\hline
\beta_n & 0.812893 & 0.615087 & / & / \\
\hline
\end{array}
\]

- Actual performance of this scheme depends heavily on the realization

Christof Kobylko - JPEG 2000 / Discrete Wavelet Transform
Lifting Scheme (2)

Naïve implementation: Re-compute all intermediate values every time

- 11 memory transactions, 20 additions and 12 multiplications per output pixel pair ⇒ Very inefficient
Lifting Scheme (3)

OpenJPEG approach: Decompose computation graph into its layers

- 20 memory transactions, 8 additions and 6 multiplications per output pixel pair
  ⇒ Computation effort traded against memory load
Lifting Scheme (4)

**Loop folding:** Compute only right-most branch in each iteration

- 4 memory transactions, 8 additions and 6 multiplications per output pixel pair ⇒ **Minimum possible effort**
Further optimizations

- OpenJPEG is storing the sub-bands pixel-wise interleaved
  - Another 4 memory transactions per output pixel pair for de-interleaving
- Optimized version instantaneously de-interleaves sub-bands and transposes the images
  - No overhead for de-interleaving
  - Only the vertical filter must be implemented (inherent transposing)
    - Easy column-wise parallelization (especially for GPUs!)
    - Column-wise vectorization possible (4 columns at once with SSE)

Transposed, de-interleaved writing scheme
Benchmarks

- 5 Different test scenarios
  - OpenJPEG reference implementation [OPJ]
  - Optimized implementation (with loop folding) [OPT]
  - Optimized implementation with column-wise parallelization [OPT_MT]
  - Vectorized optimized implementation [SSE]
  - Vectorized and parallelized optimized implementation [SSE_MT]

- 3 Different test environments
  - Intel Core i7-2630QM @ 2.00 GHz (4 CPUs), NVidia Quadro 1000M
  - Intel Xeon E5640 @ 2.67 GHz (4 CPUs), NVidia Tesla C2050
  - Intel Core i7-3930K @ 3.20 GHz (6 CPUs), NVidia Quadro 2000

- All tests are performed with 6 DWT resolution-layers
- Evaluation of raw computation time and memory throughput
Benchmarks (2)

- Test data size = 1920 * 1080 * 3 pixels

**Computation Time [ms]**

**Memory load [GByte/s]**

- i7-2630QM
- E5640
- i7-3930K
Table of Contents

● JPEG-2000
  ● Overview of JPEG-2000
  ● Idea of Discrete Wavelet Transform (DWT)
  ● DWT-Example

● Optimizing DWT for CPU-based computation
  ● Definition of DWT
  ● Lifting Scheme
  ● Further optimizations
  ● Benchmarks

● Optimizing DWT for GPU-based computation
  ● GPU-specific adaption of CPU-implementation
  ● Benchmarks

● Summary

● Literature
GPU-specific adaption of CPU-implementation

- GPU is interpreted as
  - massively parallel compute architecture with
  - 32 element-wide vector units (Warp-Size)

- Plain porting of the vectorized code to CUDA results in a relatively huge computation time!
  - Transposed scatter-write slows down GPU memory controllers extremely

- Optimization step 1:
  - Write the output values of the DWT kernel aligned to the input values
  - Transpose image with separate kernel via shared memory

  - Great performance improvement, but transpose kernel still needs up to 80 % of the total computation time!
GPU-specific adaption of CPU-implementation (2)

- Optimization step 2:
  - Create separate kernel for horizontal DWT transform
    - Compute 32 output pixel pairs in parallel via shared memory
    - Write output values aligned back to global memory
      ⇒ Transpose of images becomes unnecessary
  - Example with 4 output pixel pairs per iteration:
Benchmarks

• 5 Different test scenarios
  • Reference CPU-implementation:
    Vectorized and parallelized optimized implementation [SSE_MT]
  • Naïve CUDA-implementation of vectorized code [C_NAIVE]
  • CUDA-implementation with shared memory transpose [C_TRANS]
  • Optimized CUDA-implementation (separate kernels) [C_OPT]
  • Kernel execution time for opt. CUDA-implementation [C_KERNEL]

• 3 Different test environments
  • Intel Core i7-2630QM @ 2.00 GHz (4 CPUs), NVidia Quadro 1000M
  • Intel Xeon E5640 @ 2.67 GHz (4 CPUs), NVidia Tesla C2050
  • Intel Core i7-3930K @ 3.20 GHz (6 CPUs), NVidia Quadro 2000

• All tests are performed with 6 DWT resolution-layers
• Evaluation of raw computation time and effective PCIe-throughput
Benchmarks (2)

- Test data size = 1920 * 1080 * 3 pixels

---

**Computation Time [ms]**

<table>
<thead>
<tr>
<th>Method</th>
<th>SSE_MT</th>
<th>C NAIVE</th>
<th>C TRANS</th>
<th>C OPT</th>
<th>C KERNEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>i7-2630QM</td>
<td>40</td>
<td>70</td>
<td>50</td>
<td>20</td>
<td>10</td>
</tr>
<tr>
<td>E5640 / Tesla C2050</td>
<td>30</td>
<td>60</td>
<td>40</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>i7-3930K</td>
<td>50</td>
<td>80</td>
<td>60</td>
<td>40</td>
<td>30</td>
</tr>
</tbody>
</table>

**Eff. PCIe throughput [GByte/s]**

- **i7-2630QM / Quadro 1000M**
- **E5640 / Tesla C2050**
- **i7-3930K / Quadro 2000**
Summary

● Efficiency of OpenJPEG DWT has been improved drastically through
  ● smarter implementation of the lifting scheme
  ● elimination of unnecessary memory transactions
  ● multi-threaded processing of the columns of the image components
  ● simultaneous multi-column processing via SSE

● Memory access patterns must be changed for an efficient CUDA-implementation

● Usage of GPUs must be thought through thoroughly
  ● Even low-end GPUs outperform the computation power of high-end CPUs, but
  ● PCIe-transfer times eat up the performance gain

⇒ Porting the whole encoder chain to CUDA would be heavily beneficial, porting only the DWT offers almost no gain with respect to the multi-threaded SSE implementation
Literature


Questions ?