Speeding Up Transform Algorithms for Image Compression Using GPUs

Leif Christian Larsen, Anne Cathrine Elster (supervisor), Tore Fevang (supervisor)

{leifchl,elster}@idi.ntnu.no, tfevang@slb.com
Outline

- Motivation
- Transform Coding Algorithms: Discrete Cosine Transform and Lapped Orthogonal Transform
- GPU Implementations, Results
- Future Look
Motivation

- Seismological client-server visualization application, running on a cluster
- Large data sets (terabyte range)
- Minimize intra-node/client-server communication: Need to compress seismic data
Transform Coding Compression

Transform coding compression methods are the most popular methods for image/signal compression.

Transform computation is the most time-consuming part of compression. Desirable to implement on the GPU for both off-loading CPU and quicker computation.
GPU Architecture

Massively parallel, employing data and instruction-level parallelism [Fernando05]
Block Transforms on GPU

- Data-parallel approach; parallelize block transforms

Single-channel texture is of size $N \times N$ with one channel per texture element, RGBA texture is of size $N/4 \times N/2$ with four channels per element ($N \times N$ in total).

Input texture — One value per texture coordinate
8 values in each RGBA texture

RGBA texture 1 — Four values per texture coordinate

RGBA texture 2 — Four values per texture coordinate
2D Example
GPU Implementations

- Popular choice for transform: Discrete Cosine Transform (DCT), applied separably along each dimension, with block size 8

\[
Y_f = \sqrt{\frac{2}{n}} C_f \sum_{t=0}^{n-1} x_t \cos \left[ \frac{(2t + 1)f \pi}{2n} \right]
\]
DCT on GPU

- DCT can be computed from the FFT
- Most popular approach is using Arai, Agui and Nakajima’s algorithm [Arai88]
- More suitable for GPU: Feig and Linzer’s algorithm for multiply-add architectures [Feig91]
- Efficient GPU implementation requires using SIMD vectorization, and preferably multiply-add instructions
DCT on GPU — Feig-Linzer multiply-add algorithm

- Example: Upper half of the first stage below can be computed using a single 4-component vector instructions:
  
  \[
  \text{out1} = \text{in0\_3.\text{xyzw}} + \text{in4\_7.wzxy};
  \]

- Also use fused multiply-add instructions in the final stages
DCT Results (Walltime)

- Quad-Core Xeon 5160 3GHz vs NVIDIA Quadro 3500, 32 bits of precision
- $4096 \times 4096$: Speedup over single-core version 1.18
DCT Results (CPU time)

- Multi-core versions use more total CPU time; GPU version uses least CPU time of all with large data sets.

![Bar Chart]

- **Input set size**
  - 1024x1024
  - 2048x2048
  - 4096x4096

- **CPU time/seconds**
  - gpu-fimad-32
  - cpu-fftw-simple-32
  - cpu-fimad-32
  - cpu-fimad-2cpu-32
  - cpu-fimad-4cpu-32
Lapped Orthogonal Transform

- DCT basis function coefficients are considered for each block *independently*; large compression ratios gives artificial blocks in the image

- Most promising for seismic data seem to be transforms based on the Lapped Orthogonal Transform by Malvar [Malvar89].

- These algorithms compute DCT and proceed to “fix” the blocking effects.
Lapped Orthogonal Transform

- Fast type-I LOT by H.S. Malvar

![Diagram of Lapped Orthogonal Transform]

Stanford 50: State of the Art and Future Directions of Computational Mathematics and Numerical Computing
Lapped Orthogonal Transform

- Slight modification of Malvar’s original LOT computation scheme
Lapped Orthogonal Transform

- GPU implementation done in several passes. Serial dependency between certain stages.
- Again, heavy use of vectorized instructions

```plaintext
FLMADD DCT + 1st butterflies on rows → Row scatter → 2nd butterflies + Z-block on rows → Row scatter

FLMADD DCT + 1st butterflies on columns → Column scatter → 2nd butterflies + Z-block on columns → Column scatter
```
LOT Results (Walltime)

- Quad-Core Xeon 5160 3GHz vs NVIDIA Quadro 3500, 32 bits of precision
- Speedup: 2.77 over quad-core version. Bottleneck is upload/download time.
LOT Results (CPU time)

About 11 times speedup
Future Peek

- LOT only considers neighbouring blocks.
- Solution proposed is a generalized LOT (GenLOT) which extends into blocks beyond neighbouring blocks [deQuerioz96].
- GenLOT can be generalized to use basis functions of unequal length, named GULLOT [Nagai00].
- These transforms are better approximations to the optimal, signal-dependent Karhunen-Loeve transform.
Future Peek: GenLOT/GULLLOT On The GPU

- Can use precisely the same techniques as for the LOT; GenLOT/GULLLOT simply adds more stages. Will require more passes.
- Will likely gain even better speedups!
Future Peek: KLT On The GPU

- Can be implemented through matrix operations using the covariance method.
- Will be relatively easy with NVIDIA’s recent CUDA library for general GPU programming, which supports matrix/vector operations.
Questions?
Thank you for your attention!


Thank you for your attention!

