Preprocessing Volumetric CT Enterography Data on Modern Hardware

Jan Horáček

Charles University in Prague
Faculty of Mathematics and Physics

June 3, 2010

Table of contents

CT Enterography

Figure 1: CT enterography example

Introduction to CT enterography

- Diagnosis of Crohn's disease and other problems with small intestine
- Noninvasive, relatively safe procedure
- Speed and resolution of multidetector CT
- Large volumes of ingested neutral enteric contrast material
- Good visualization of intestinal wall and lumen
- Clearly shows small intestine inflammation by displaying thickening of intestinal wall

CT enterography

- Oral contrast agent in several doses starting about 60 minutes prior to examination
- About 1 minute before examination is injected intravenous contrast agent
- CT scan with 1-3mm thick slices
- Radiologist inspection

Thick vs. thin slices

- Thick slices unsuitable for automatic/semiautomatic processing
  - Good for radiologist examination
  - Connectivity of thin details lost in thick slices
- Thin slices burdened with too much noise
  - Difference between lumen and wall mean value may even become smaller than standard deviation of noise inside homogeneous lumen

Denoising

Figure 2: Noise-burdened CT data

Denoising

- Even with enterography approach - strong noise covering important details
- Axial slices usually already lowpass-filtered from the machine
- Human eyes are able to see details, but only with correct WL settings and with the help of passing through slices
- (Semi)automatic segmentation very difficult

Denoising

- Large volume of data for each patient
  - 512x512 images, approx. 400-600 slices
- Must preserve small details
- Gaussian lowpass fitering - blurs high contrast areas
- Median filtering - good for edges, removes thin details

Nonlocal means algorithm

- Introduced by [Buades et al. 2005]
- The best current algorithm for denoising
- Removes noise, but preserves details
- Approaches for automatic parameter tunning
- Computationally very expensive
  - May run for hours on thin-slice abdominal dataset
- Not iterative, easy parallelization
- Successfuly used for denoising 2D images, 3D data, surface meshes, etc.

NL-means in 2D

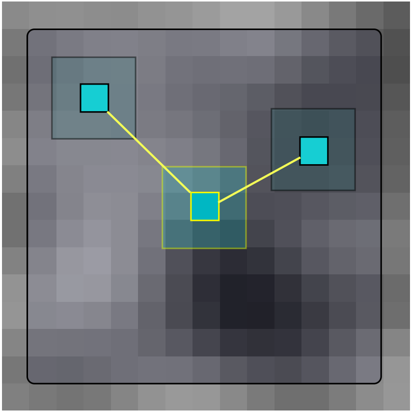


Figure 3: Nonlocal means algorithm schema

NL-means in 2D

$$NL(u)(x_i) = \sum_{x_j \in \Omega^3} w(x_i, x_j) u(x_j) \quad (1)$$

$$w(x_i, x_j) = \frac{1}{Z_i} e^{-\frac{||u(N_i) - u(N_j)||^2_{2,a}}{h^2}} \quad (2)$$

Accelerating NL-means on CPU

- Very effective optimization introduced by [Coupé et al. 2008]
- Select only relevant voxels
  - Compare local *mean* and *variance* values
- Automatic tuning of smoothing parameter h
- Blockwise approach

Accelerating NL-means on CPU 2

- Both voxel selection and blockwise approach improve time complexity by an *order*
- Coupé et al. claim improvement as much as 30x-66x
- Implementation details not given

Accelerating NL-means on GPU

- Many examples of NL-Means algorithm on GPU, but only for 2D images
- Modern GPU example (GT200):
  - Max. 512 threads in work group
  - 16kb local memory
  - Processing single instruction on 8 threads
  - More than 500MB memory
  - Hundreds of cores

Primitive algorithm

- Reimplement Optimized NL-Means algorithm (with mean & variance selection) in OpenCL

Primitive algorithm

- Reimplement Optimized NL-Means algorithm (with mean & variance selection) in OpenCL
- GeForce 275 GTX approx. the same speed as Optimized NLM on Core i7 running with 8 threads

Primitive algorithm

- Reimplement Optimized NL-Means algorithm (with mean & variance selection) in OpenCL
- GeForce 275 GTX approx. the same speed as Optimized NLM on Core i7 running with 8 threads
- 6-7x slower than Blockwise Optimized CPU version on Core i7
- Bottleneck:
  - Global memory reads
  - Breaking thread consitency with voxel selection

GPU optimized algorithm

- NOT optimization by design, but rather implementation optimizations on given architecture
- Base is *standard* NL-Means
  - Without selections and other optimizations
  - Only L2 norm not Gauss-filtered
- Massive usage of parallelism hides computational complexity
- About 2-4 times faster than fastest CPU implementation

GPU optimized algorithm 2

- Prevent global memory reads
- Fit as much data into local memory as possible
- Use barriers to make the code run efficiently
- Parameters:
  - Search radius = 4 voxels
  - Local neighbourhood size = 2 voxels

Algorithm - workgroups

- One workgroup for one column of voxels
- One thread for each voxel in the neighbourhood
- One workgroup processes a single Z-column of values

Algorithm - kernel

1. Each thread reads the first $2 \cdot (4 + 2) + 1$ voxels in the Z direction from global memory to local memory
2. For each relevant voxel, compute the weight function with central voxel
3. Compute sum of weigths
4. Compute sum of weight $\cdot$ value for each relevant voxel
5. Normalize with weight sum
6. Store result into global memory
7. In each thread move all voxels in local memory by 1 and read one new voxel
8. Continue with step 2

Voxels

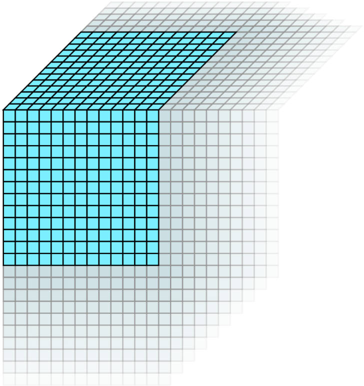


Figure 4: Volume needed for one voxel

Voxels 2

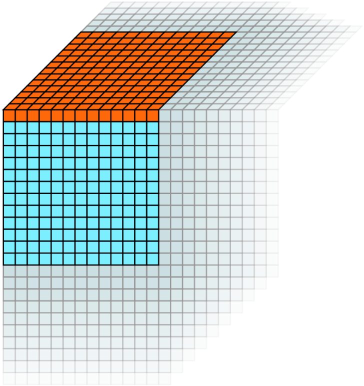


Figure 5: One thread for each voxel in X/Y loads data

Voxels 3

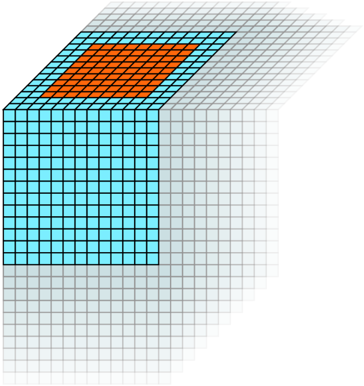


Figure 6: Threads actually computing weights

Voxels 4

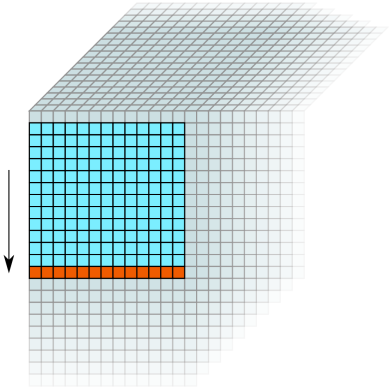


Figure 7: Loading next slice - one thread per voxel

Memory consumption

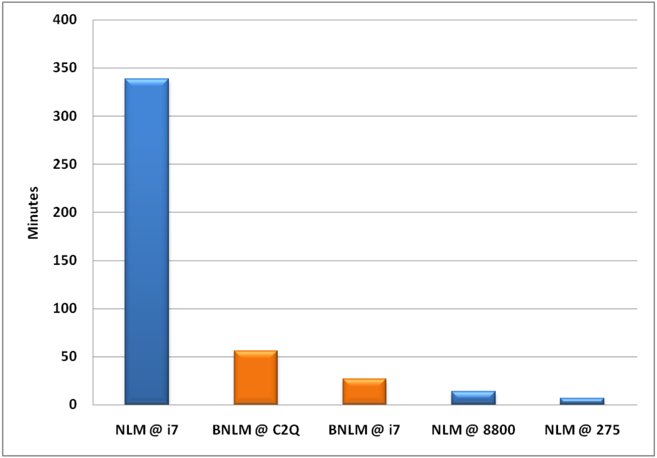
- Number of voxels $= (2 \cdot (4+2)+1)^3 = 2197$
- Source data size $= 2197 \cdot 4$ bytes per float $= 8788$ bytes
- Temporary memory for weights $= (2 \cdot 4+1)^3 = 729$ floats $= 2916$ bytes
- Additional memory for summing weights $= 360$ bytes
- Some local variables
- Fits into 16k OpenCL local memory

Results

Algorithm	Processor	Threads/WkGrp	Time
NL-Means	Core i7 3.07GHz	8	5:38:15
Blockwise NLM	Core i7 3.07GHz	8	0:26:35
NL-Means	C2Quad 2.4GHz	4	—
Blockwise NLM	C2Quad 2.4GHz	4	0:55:57
NL-Means	GeForce 8800GT	13^2	0:13:41
NL-Means	GeForce 275GTX	13^2	0:06:44

Figure 8: Measured on dataset of size 512x512x548

Results - graph



Example of denoised data

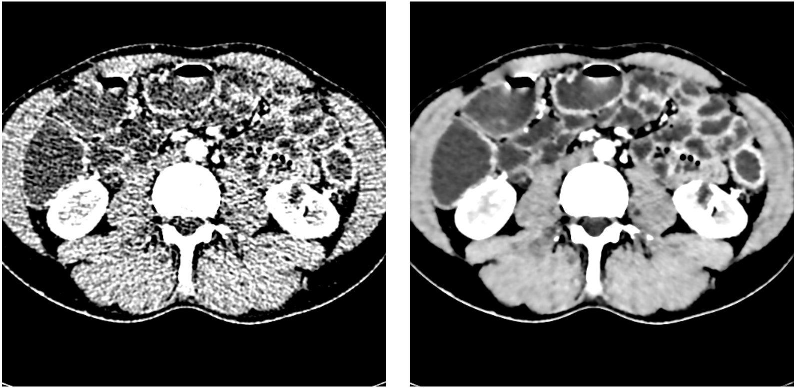


Figure 9: Axial slice

Example of denoised data 2

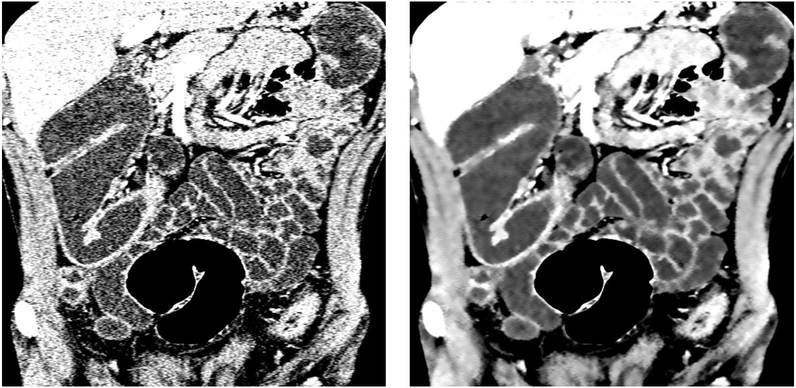


Figure 10: Frontal slice

Conclusion

- The slowest and most computationally expensive algorithm on CPU runs faster on GPU than the fastest solution on CPU
- Not computing something may mean slowing down on GPU
- Global memory on GPU is prohibitively expensive to access more than once and unaligned
- 7 minutes per patient is much better than 5.5 hours, but still not enough for practical use
- Optimizations that work very well on CPU are not easily applicable on GPU

Future work

- Blockwise approach
- Try other optimizations, to use all 512 threads with the given memory

Thank you for your attention

Questions?