MIC-GPU: High-Performance Computing for Medical Imaging on Programmable Graphics Hardware (GPUs)

SPIE Medical Imaging

Code Optimization Case Study and Demo

Klaus Mueller and Sungsoo Ha

Stony Brook University

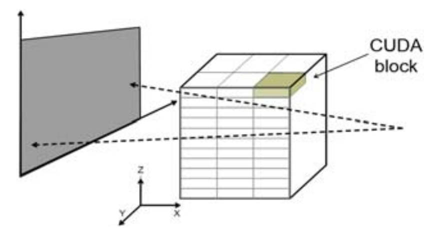
Computer Science

Stony Brook, NY

Optimization Case Study

Goal:
- test various optimization strategies and tweak to maximize impact
- using Filtered Backprojection for this case study

Optimization #1: minimize shared memory usage
- update a block of voxels per thread (optimum was 16 × 16 × 4)
- orientation-neutral block minimizes “shadow” on projections



Optimization Case Study

Optimization #2: exploit special GPU (ASIC) hardware
- we store projection data in texture memory
- allows fast bilinear interpolation
- frees up registers without penalty since texture is cached

Optimization #3: exploit constant memory
- we store projection (system) matrix in constant memory
- frees up shared memory and reduces global memory accesses

Optimization #4: increase thread granularity
- backproject multiple projections in one thread (optimum was 4)
- reduces global memory accesses and number of kernel invocations

Optimization Case Study

Optimization #5: Pre-fetching
- pre-fetch data while computing on previous data
- incurs some shared memory overhead but worked out OK

Optimization #6: Page-locked memory
- page-lock the result array
- forces OS to store this data on one contiguous page of memory
- eliminates the need for page swaps

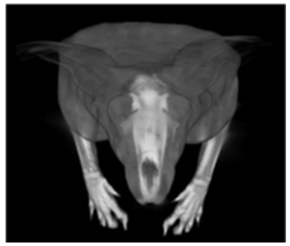
Other optimization strategies: loop unrolling, fast math
- we tried these but they did not yield much benefits in this specific case

Case Study – Rabbit CT

Benchmarking framework:
- developed by Rohkohl et al.
- FDK backprojection algorithm
- 496 projections of a rabbit
- 1,248 X 960 pixels each

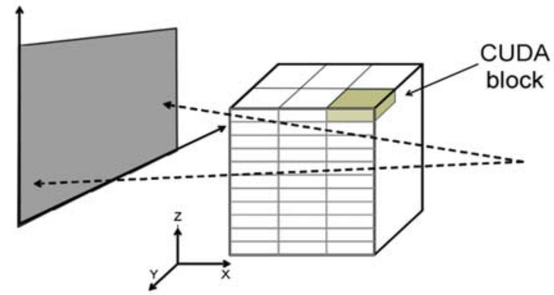
Advantages:
- enables true comparisons
- embeds the system matrix already
- 'just' accelerate the backprojection
- measures timings
- measures reconstruction errors

Leaderboard
- benchmark new code
- 256^3, 512^3, 1024^3 volume reconstructions



Setup

Approach:
- each thread computes an array of voxels



Thread Block Dimension: 16 x 16 x 4

Winning Implementation #1

```
texture<float, 2> tRef, tRef2, tRef3, tRef4
__constant__ float A[48]

row = blockIdx.y * blockDim.y + threadIdx.y
col = blockIdx.x * blockDim.x + threadIdx.x
FOR k = 0 to L
  result = f_L[k * L² + row * L + col]
          . . .
  // mapping voxel (x,y,z) to projection 1 and backproject
  w = A[2] * x + A[5] * y + A[8] * z + A[11]
  u = (A[0] * x + A[3] * y + A[6] * z +A[9]) / w
  v = (A[1] * x + A[4] * y + A[7] * z + A[10]) / w
  result += tex2D ( tRef, (u + 0.5), (v + 0.5)) / w²

  //repeat for projection 2 with A[12-23] and tRef2
  //repeat for projection 3 with A[24-35] and tRef3
  // repeat for projection 4 with A[36-47] and tRef4

  f_L[k * L² + row * L + col] = result
END
```

+ page-lock memory
+ pre-fetch data

E. Papenhausen, Z. Zheng, K. Mueller, "GPU-Accelerated Back-Projection Revisited: Squeezing Performance by Careful Tuning,"

Winning Implementation #2

RapidRabbit strikes back
- re-order projection and voxel loop (improves locality)
- faster perspective divide and depth weighting using fast inverse square root

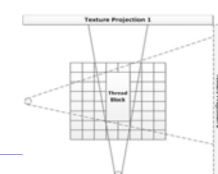
  $w = a_2x + a_5y + a_8z + a_{11}$

  $w' = rsqrt(w * w)$

  $u = (a_0x + a_3y + a_6z + a_9) * w'$

  $v = (a_1x + a_4y + a_7z + a_{10}) * w'$

  $result \mathrel{+}= tex2D(tRef, (u+0.5), (v+0.5)) * w' * w'$
- accumulation via atomic adds
- staged page-locks
- transpose volume at 45°



E. Papenhausen, Z. Zheng, K. Mueller, "Rapid Rabbit: Highly Optimized GPU Accelerated Cone-Beam CT Reconstruction," IEEE Medical Imaging Conference,

SPIE Medical Imaging 2016

Demo

handing over to Sungsoo