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

Code Optimization Case Study and Demo

Klaus Mueller and Sungsoo Ha
Stony Brook University
Computer Science
Stony Brook, NY

Optimization Case Study

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

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 #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
- 256³, 512³, 1024³ volume reconstructions

Setup

Approach:
- each thread computes an array of voxels

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 * L^2 + row * L + col]

    // mapping voxel (x,y,z) to projection 1 and backprojection

    result += tex2D(tRef, (u + 0.5), (v + 0.5) / w^2)

//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 * L^2 + row * L + col] = result

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
  \[ w = a_2x + a_5y + a_8z + a_{11} \]
  \[ w' = rsqrt(w \cdot w) \]
  \[ u = (a_0x + a_3y + a_6z + a_9) \cdot w' \]
  \[ v = (a_1x + a_4y + a_7z + a_{10}) \cdot w' \]
  \[ result += tex2D(tRef, (u+0.5), (v+0.5) \cdot w' \cdot w') \]

- accumulation via atomic adds
- staged page-locks
- transpose volume at 45°
handing over to Sungsoo