

# Fast Seismic Modeling and Reverse Time Migration on a GPU Cluster



AUBURN  
UNIVERSITY

R. Abdelkhalek <sup>1</sup>

H. Calandra <sup>1</sup>

O. Coulaud <sup>2</sup>

G. Latu <sup>3</sup>

J. Roman <sup>2</sup>

1. TOTAL

2. INRIA/Scalapplix Project

3. Strasbourg University, INRIA/Calvi Project

(this paper appears in HPCS '09)



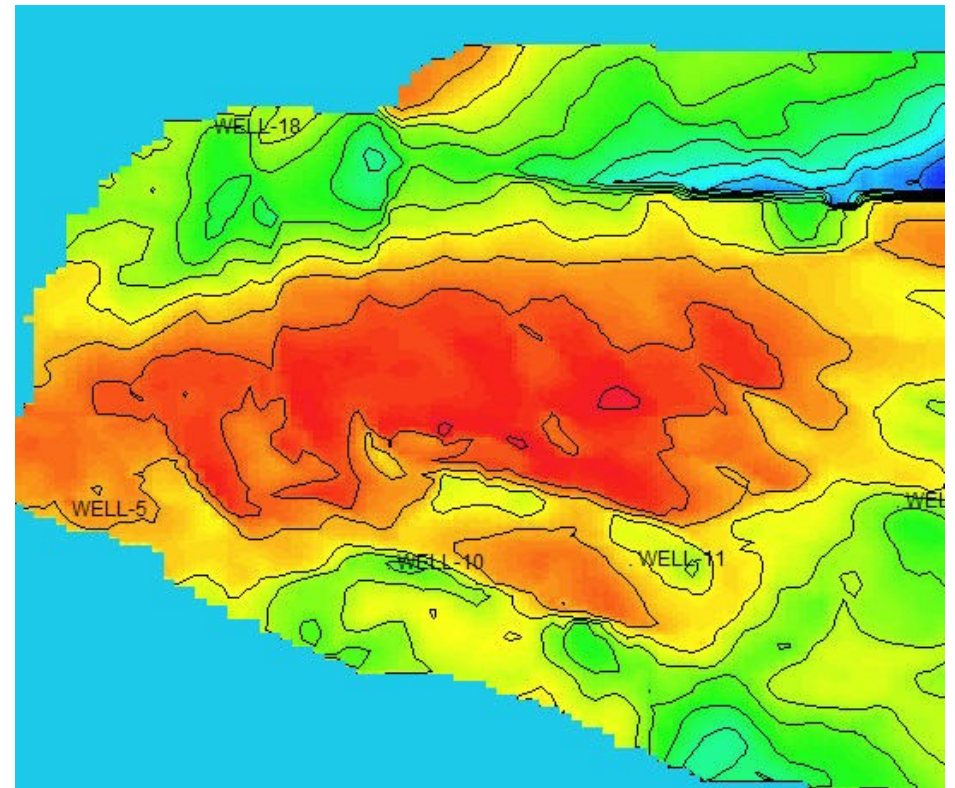
# Talk Outline (1 of 6)

- **Introduction & Background**
- Computational Challenges
- CPU Algorithms
- GPUs and CUDA
- GPU Implementations
- Results & Conclusions



# Introduction: The Need (1 of 3)

- Oil & Gas Industry needs accurate representation of subsurface
- Data generated by controlled explosions and measurement
- Data requires analysis to find geological layer interfaces



(Sub surface map at approx. 3000m below sea level, showing the outline of an oilfield in red, courtesy of Cambridge Petroleum Software Consultants [<http://www.velocitymanager.com>] {{Attribution}})



# Introduction: The Tools (2 of 3)

- Important computational tools highly compute-intensive:
  - Seismic Modeling (SM)
  - Reverse Time Migration (RTM)
- Based on full acoustic wave discretization (to be discussed later):

$$\frac{1}{c^2 \rho} \frac{\partial^2 u}{\partial t^2} = \nabla \cdot \left( \frac{1}{\rho} \nabla u \right)$$

$c(x, y, z)$ : velocity

$\rho(x, y, z)$ : density

$u(x, y, z, t)$ : pressure field

# Introduction: The Idea (3 of 3)

- GPGPU an attractive optimization technique
  - Massively thread- and data-parallel
  - (Relatively) cheap, commodity devices
  - Unified architecture
  - General-purpose language support
  - *Wave of the future?*



NVIDIA GeForce GTX 280  
Courtesy Hardware Canucks  
(<http://www.hardwarecanucks.com>)



# Talk Outline (2 of 6)

- Introduction & Background
- **Computational Challenges**
- CPU Algorithms
- GPUs and CUDA
- GPU Implementation
- Results & Conclusions



# Challenges: SM (1 of 3)

- Seismic Modeling consists of two stages:
  - Simulate seismic wave propagation in geological layers
  - Generate synthetic seismograms based on assumed structure
- Direct methods are very accurate
- Direct methods are incredibly computationally intensive

# Challenges: RTM (2 of 3)

- Reverse time migration is the other part of the picture
- Used to create seismic images in areas of complex wave propagation
- Introduced in the 1980s
- Not widely used since then due to its impractically large computational demands

# Challenges: Basis (3 of 3)

- Based on the full acoustic wave discretization:

$$\frac{1}{c^2 \rho} \frac{\partial^2 u}{\partial t^2} = \nabla \cdot \left( \frac{1}{\rho} \nabla u \right)$$

$c(x, y, z)$ : velocity  
 $\rho(x, y, z)$ : density  
 $u(x, y, z, t)$ : pressure field

- Simplifying case:  $\rho(x, y, z) = \rho$  (const.):

$$\frac{1}{c^2} \frac{\partial^2 u}{\partial t^2} = \Delta u$$

$c(x, y, z)$ : velocity  
 $u(x, y, z, t)$ : pressure field

- This is discretized using a 2<sup>nd</sup> order time leap-frog, 8<sup>th</sup> order space centered difference scheme... for details, consult paper.



# Talk Outline (3 of 6)

- Introduction & Background
- Computational Challenges
- **CPU Algorithms**
- GPUs and CUDA
- GPU Implementation
- Results & Conclusions



# CPU Algorithms: SM (1 of 3)

## Sequential Variable Density Seismic Modeling

1. **for** time  $\leftarrow$  t\_start **to** t\_end
2.   **do** add source term
3.   **for all** Domain grid points
4.     **do** compute forward first derivatives
5.   **for all** Domain grid points
6.     **do** compute backward first derivatives
7.     update wave field
8.   save seismogram



# CPU Algorithms: RTM (2 of 3)

## Reverse Time Migration Algorithm

1. **for** time  $\leftarrow$  t\_start **to** t\_end
2.   **do** add source term
3.   **for all** Domain grid points
4.     **do** take forward time step
5.   save boundaries
6. **for** time  $\leftarrow$  t\_end **to** t\_start
7.   **do** read saved boundary
8.   **for all** Domain grid points
9.     **do** backward time step source wave field
10. add receiver term
11. **for all** Domain grid points
12.   **do** backward time step receiver wave field
13. **for all** Domain grid points
14.   **do** apply imaging condition



# CPU Algorithms: Concurrent (3 of 3)

- In practice, these algorithms can be parallelized on CPUs
  - Employs subdomain decomposition
  - Ghost cells passed asynchronously via MPI
  - Improves access times along slow axes by cache blocking

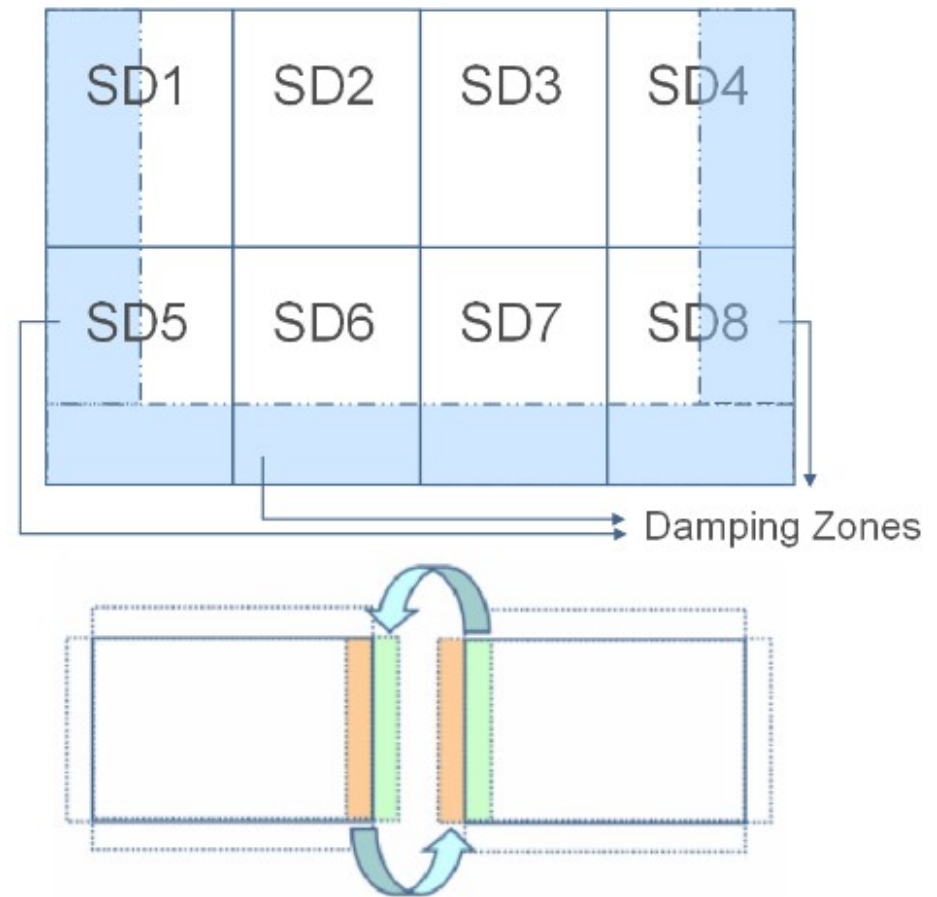


Figure 3

# Talk Outline (4 of 6)

- Introduction & Background
- Computational Challenges
- CPU Algorithms
- **GPUs and CUDA**
- GPU Implementation
- Results & Conclusions



# GPUs & CUDA: Basics (1 of 8)

- Some terminology...
  - *Host*: Sequential CPU
  - *Device*: Parallel GPU
- Basic GPGPU programming model
- *Host* performs I/O, offloads computations to *device*
  - *Device* executes many concurrent threads, asynchronously to *host*
  - *Host* retrieves/records results



# GPUs & CUDA: Basics (2 of 8)

- Overview of GPU Architecture
  - GPU consists of several *multiprocessors*
  - Multiprocessors have several *streaming processors*
  - *Streaming processors* run in SIMD-like way
- Overview of GPU Memory Layout
  - Each *multiprocessor* contains several registers and some fast *shared memory* accessible by it only.
  - All *multiprocessors* can access a larger amount of slower *global memory*.
  - Also small amount of faster *constant memory*.



# GPUs & CUDA: Basics (3 of 8)

- Introduction to CUDA
  - Stands for **C**ompute **U**nified **D**evice **A**rchitecture
  - Designed by NVIDIA for non-graphics programmers
  - Enables easy (well, easier) GPGPU
    - CUDA-C: slight extension of C for programming GPU
    - Code is portable between CUDA-enabled devices
- Basic CUDA programming model
  - *Host* thread copies input data to device
  - *Device* processes data with multi-threaded *kernel*
  - *Host* thread copies output data from device



# GPUs & CUDA: Basics (4 of 8)

- CUDA threading model
  - *Kernels* executed as concurrent *threads*
  - *Threads* grouped into 1D, 2D, or 3D *blocks*
  - *Blocks* grouped into 1D or 2D *grids*
  - Groups of concurrent *threads* scheduled as *warps*

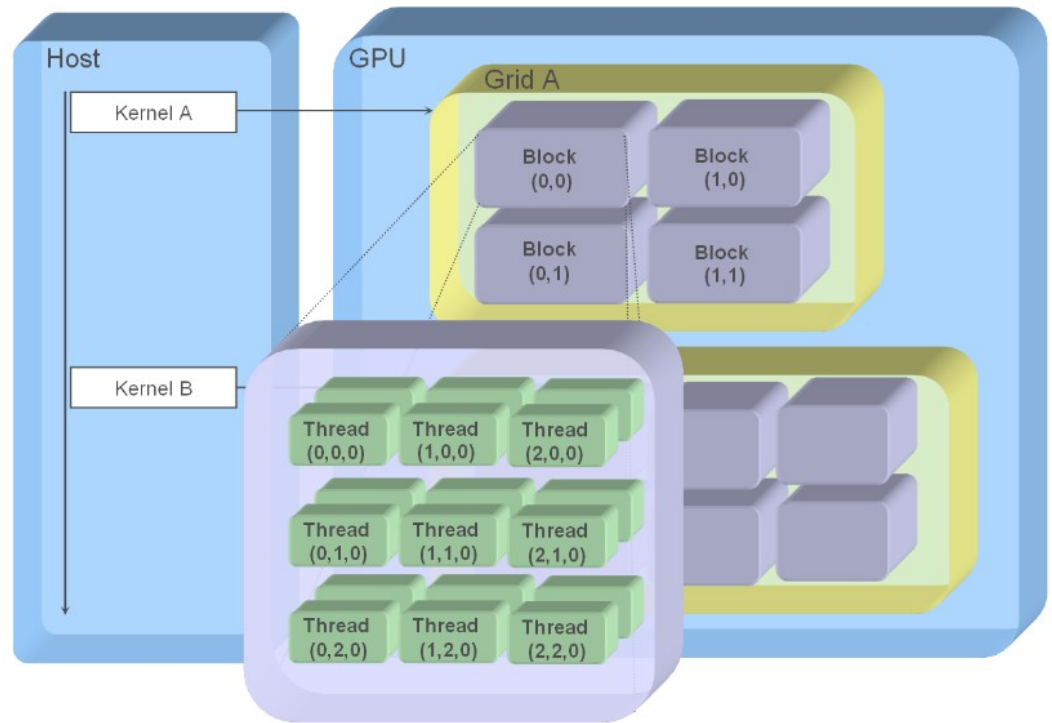


Figure 4

# GPUs & CUDA: Example (5 of 8)

## Device 0: "GeForce GTX 280"

CUDA Driver Version:	3.20
CUDA Runtime Version:	3.20
CUDA Capability Major/Minor version number:	1.3
Total amount of global memory:	1073020928 bytes
Multiprocessors x Cores/MP = Cores:	30 (MP) x 8 (Cores/MP) = 240
Total amount of constant memory:	65536 bytes
Total amount of shared memory per block:	16384 bytes
Total number of registers available per block:	16384
Warp size:	32
Maximum number of threads per block:	512
Maximum sizes of each dimension of a block:	512 x 512 x 64
Maximum sizes of each dimension of a grid:	65535 x 65535 x 1
Clock rate:	1.43 GHz

# GPUs & CUDA: Optimization (6 of 8)

- Some easy CUDA/GPU Optimizations
  - Avoid divergence of threads within warps (safer: within blocks or, better, within kernels)
  - Increase computational load of threads to hide memory latency
  - Tweak data structures to expose useful data parallelism
  - Not everything is faster on the GPU – exploit asynchronous CPU execution!

# GPUs & CUDA: N-Body (7 of 8)

Simple example: N-Body problem.

CPU and GPU acceleration computation pseudocode:

**cpu\_derivs**(t, x[1..n], dxdt[1..n])

1. dxdt[1..n]  $\leftarrow$  zeros[1..n]
2. **for** i  $\leftarrow$  1 **to** n
3.   **do** dxdt[i\*2+0]  $\leftarrow$  x[i\*2+1]
4.   **for** j  $\leftarrow$  i + 1 **to** n
5.     **do** update(dxdt[i\*2+1], i, j)
6.     update[dxdt[j\*2+1], j, i)

**gpu\_derivs**(t, x[1..n], dxdt[1..n], tid)

1. dxdt[2\*tid..2\*tid+1]  $\leftarrow$  zeros[1..2]
2. dxdt[tid\*2+0]  $\leftarrow$  x[i\*2+1]
3.   **for** j  $\leftarrow$  1 **to** tid - 1
4.     **do** update(dxdt[tid\*2+1], tid, j)
5.   **for** j  $\leftarrow$  tid + 1 **to** n
6.     **do** update(dxdt[tid\*2+1], tid, j)

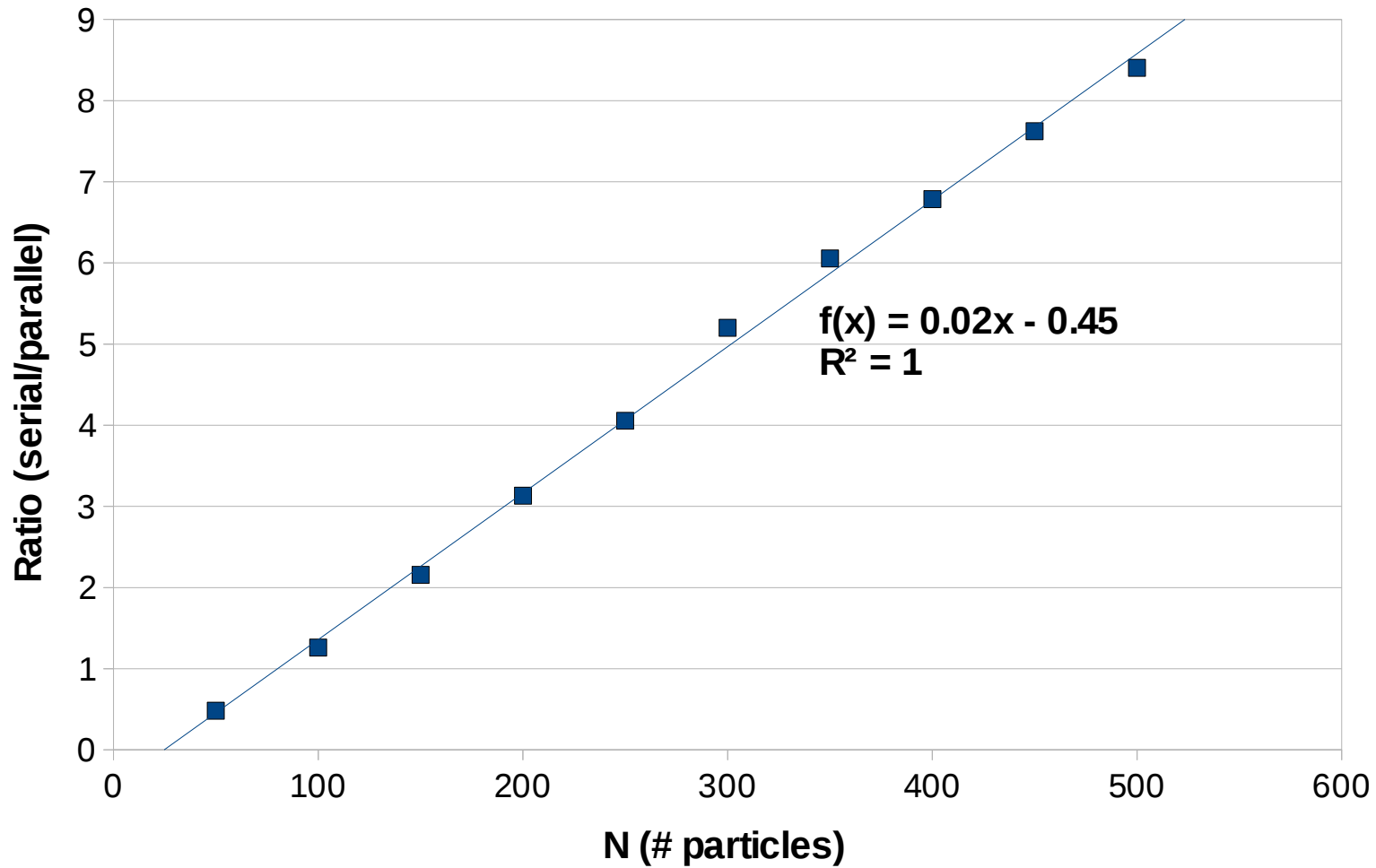
gpu\_derivs executed n times (one per particle)

cpu\_derivs is  $O(n^2)$ , gpu\_derivs is  $O(n)$  (*well, not really*)



# GPUs & CUDA: N-Body (8 of 8)

## Ratio of Serial and Parallel Total Times



# Talk Outline

- Introduction & Background
- Computational Challenges
- CPU Algorithms
- GPUs and CUDA
- **GPU Implementation**
- Results & Conclusions



# GPU Implementation (1 of 3)

- Occupancy: ratio of # of active warps per multiprocessor to max. # of warps possible
- First approach:
  - Keep domain in global GPU memory
  - Allocate one thread per grid point
- Problem: memory bounded
  - 25 global memory reads
  - 1 global memory write
- Achieved 66% occupancy, but memory slow



# GPU Implementation (2 of 3)

- Second approach:
  - Load global data into shared memory
- Problem: limited shared memory
  - 1 block/multiprocessor => max 25% occupancy!
- Final approach:
  - Use a sliding window approach to limit shared memory
  - Faster memory + higher occupancy => best performance

# GPU Implementation (3 of 3)

- What about RTM?
  - Device: forward propagation
  - Device & Host: receiver wave field propagation & source wave field retropropagation, imaging condition (asynchronous)
  - Overlapped communication & I/O reduce cost of collaboration

# Talk Outline

- Introduction & Background
- Computational Challenges
- CPU Algorithms
- GPUs and CUDA
- GPU Implementation
- **Results & Conclusions**



# Results & Conclusions (1 of 4)

Blade	CPU Node	NVIDIA S1070
Number	10	5
Processor	Xeon 5405	T10
Sockets x Cores	2 x 4	4 x 240
Clock Frequency (GHz)	2.00	1.44
Memory per Blade (Gbytes)	16	16 (4 x 4)
Cache/Shared Memory	4 x 6 MB	30 x 16 KB
Peak Performance (GFlops)	64	1036

# Results & Conclusions (2 of 4)

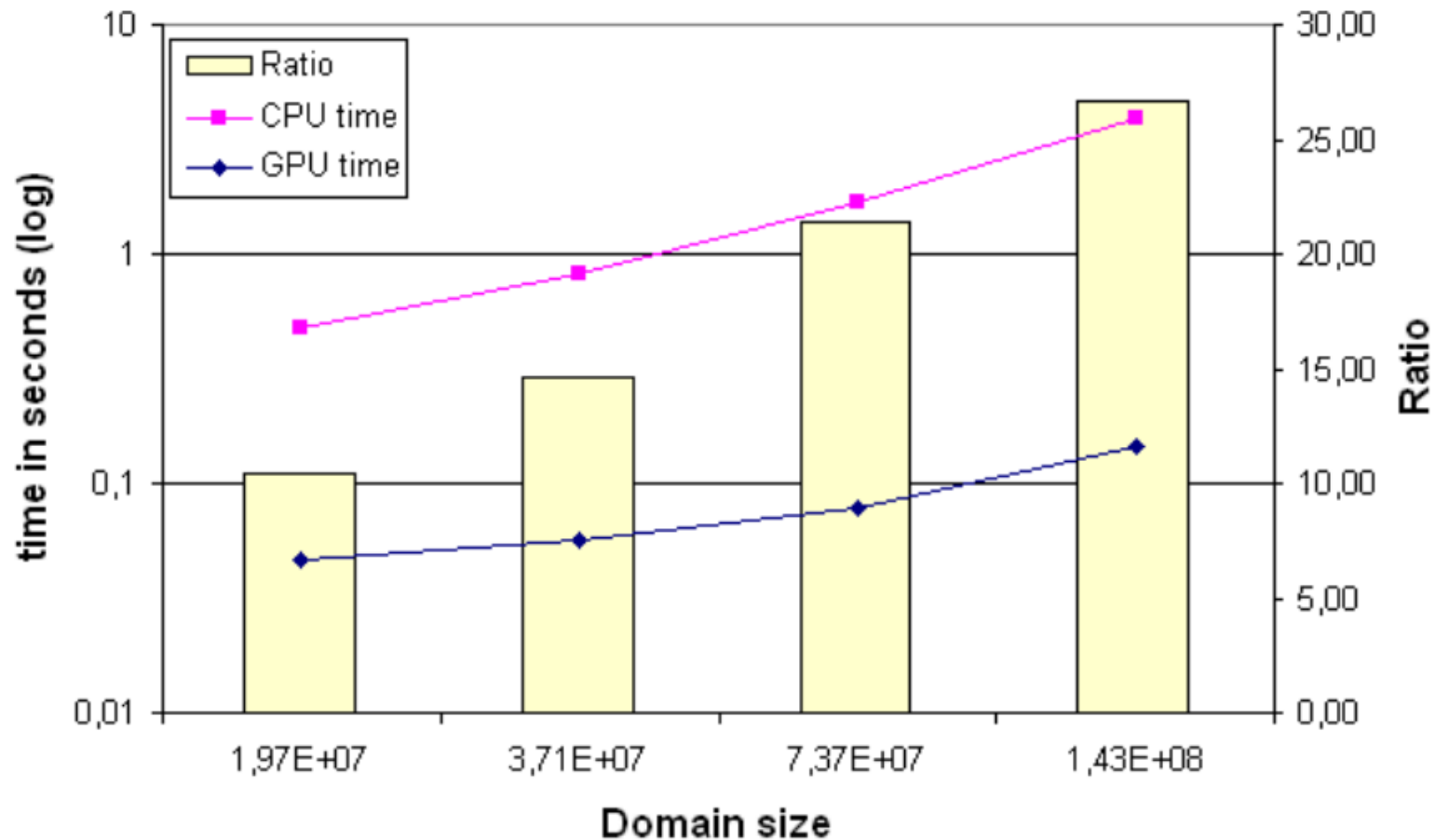


Figure 7: 3D Constant Density Modeling Times.

# Results & Conclusions (3 of 4)

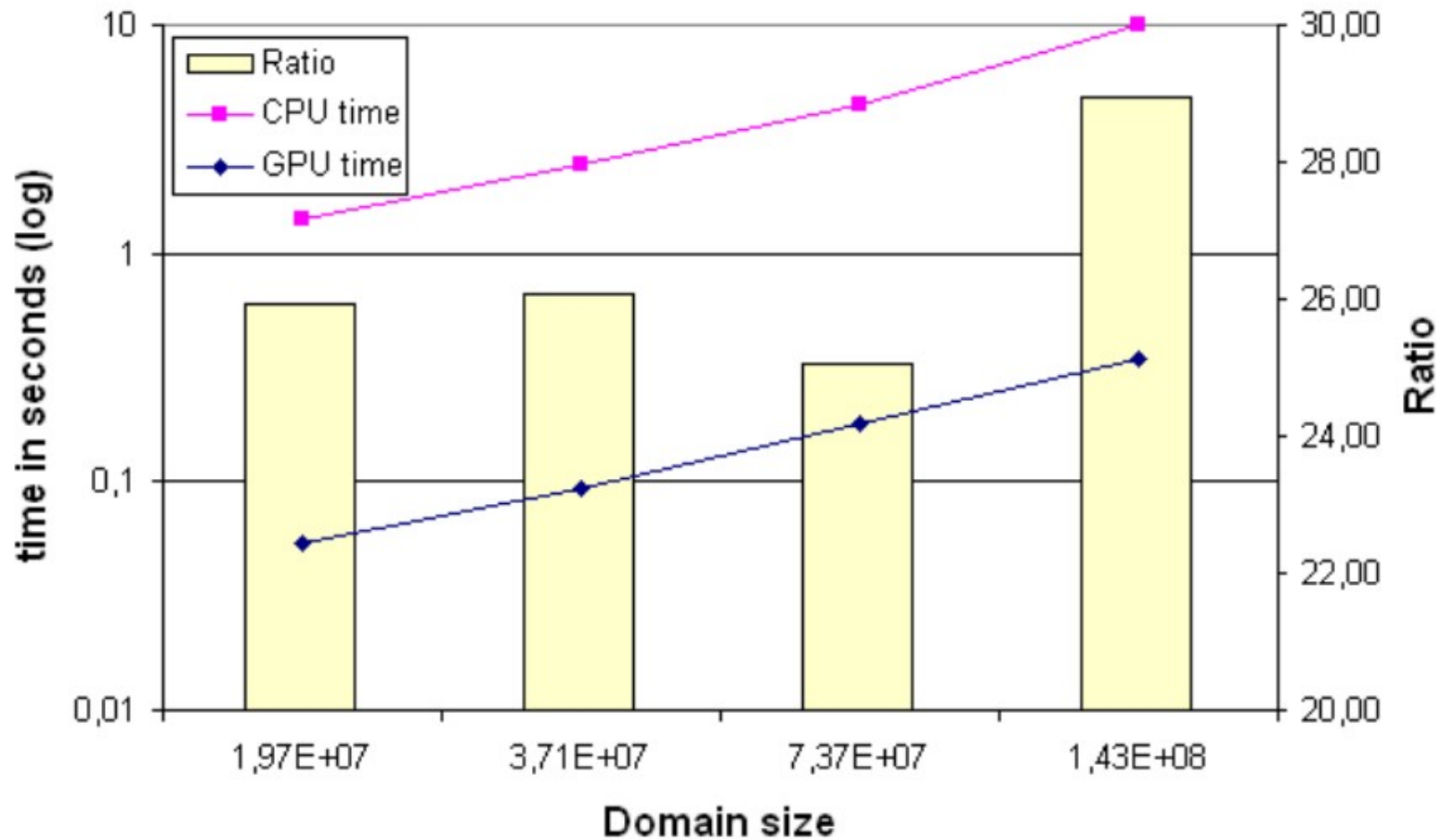


Figure 8: 3D Variable Density Modeling Times.

# Results & Conclusions (4 of 4)

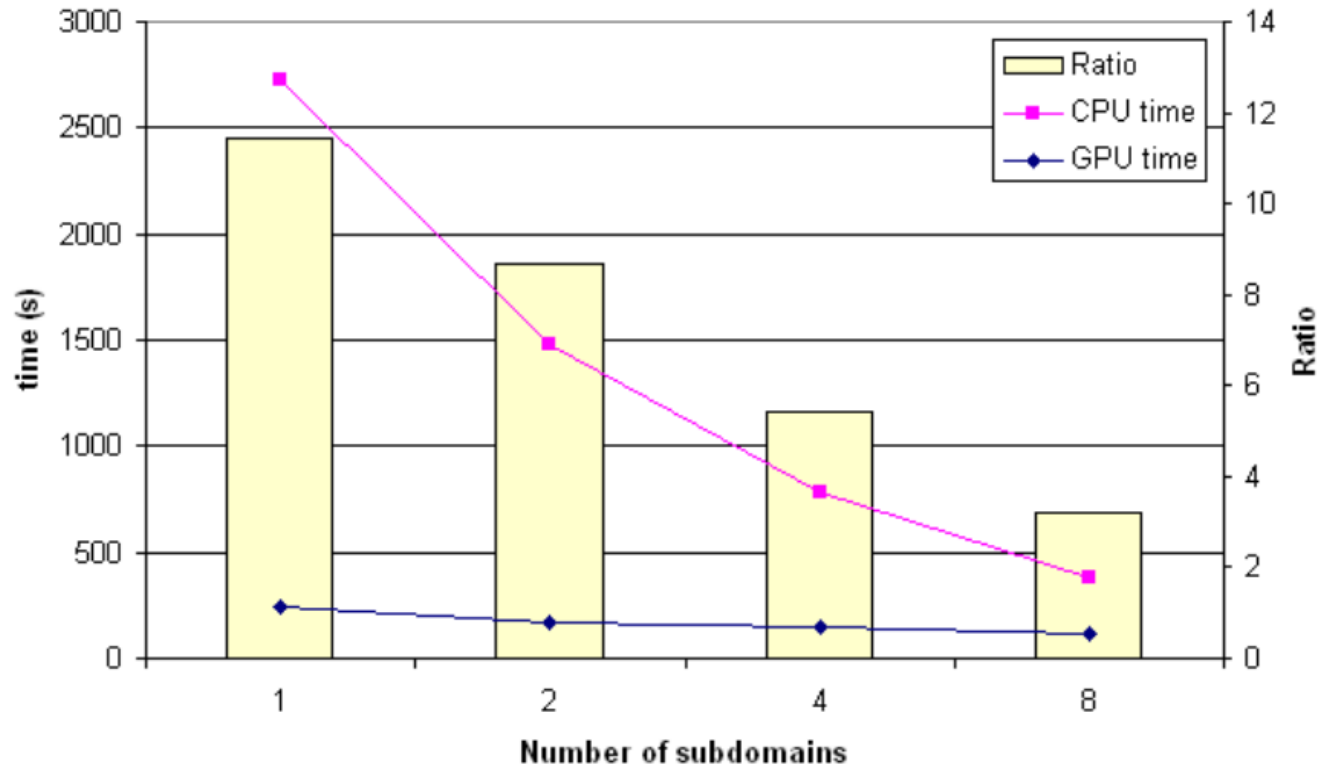


Figure 9: Reverse Time Migration Times in Seconds for one Iteration on a  $288 \times 118 \times 338$  Test Case.

The End



AUBURN

---

UNIVERSITY

