Seismic Imaging on NVIDIA GPUs

Algorithms and Porting & Production Experiences

Scott Morton

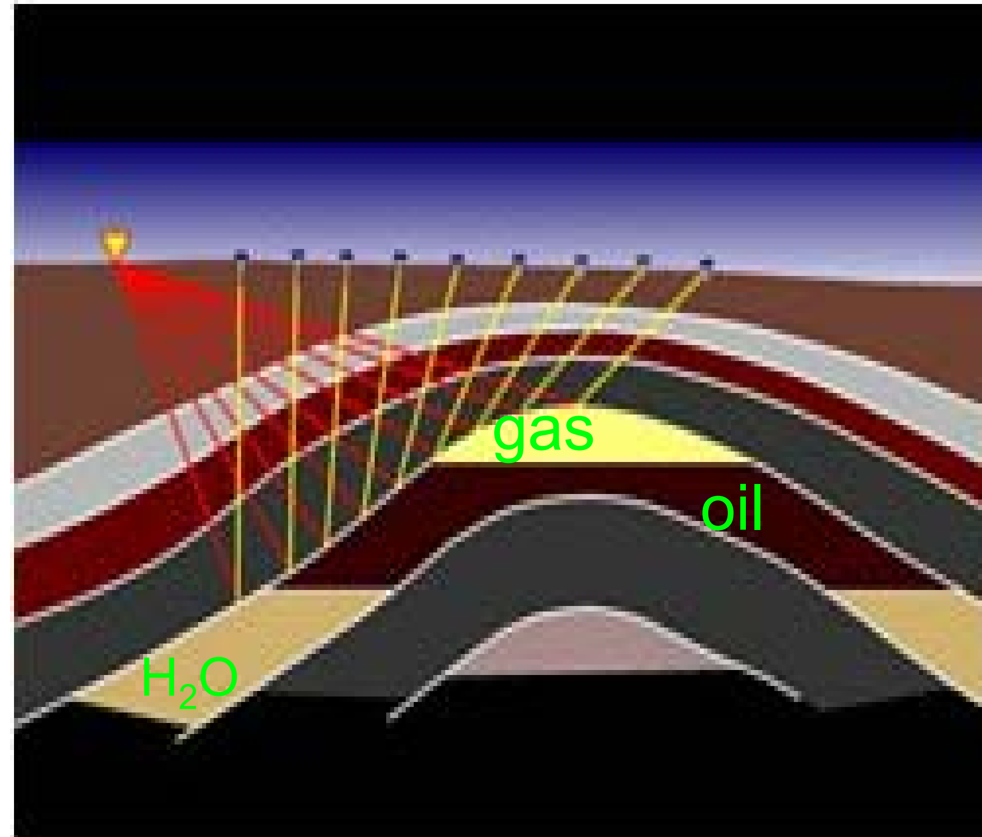
Hess Corporation

Seismic Imaging
Outline

- Introduction
  - Seismic data
  - Seismic imaging
- NVIDIA GPUs + CUDA
  - Why?
  - How?
- Porting imaging algorithms
  - High-frequency propagation
  - One-way propagation
  - Two-way propagation
- Production experiences

Seismic Imaging
Data collection

- Each experiment
  - impulsive source: “shot”
  - record echoes
    - hundreds of “receivers”
    - record for ~ 10 sec
    - at few ms sampling
    - recording called a “trace”
  - “shot record”



Seismic Imaging

"shot record"

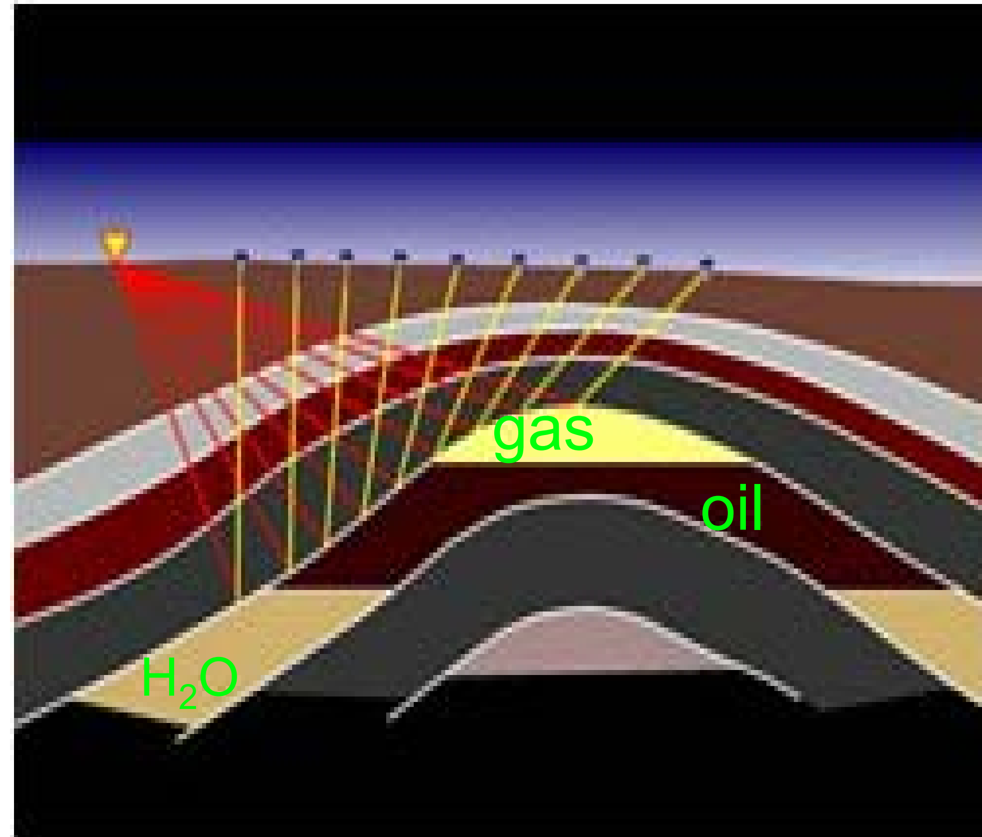
Receiver

Time (ms)

0
500
1000
1500
2000
2500
3000
3500
4000
4500
5000
5500
6000
6500
7000
7500
8000
8500
9000
9500
10000
10500
11000
11500
12000
12500
13000
13500
14000

Seismic Imaging
Data collection

- Information content
  - travel time
    - reflector location
    - geologic structure
    - *oil "floats" on water*
    - *gas "floats" on oil*
  - reflection strength
    - rock properties



Seismic Imaging
Data collection

- Seismic survey
  - area ~ 10^3 km^2
  - shots ~ 10^5
  - traces ~ 10^8
  - sizeof(trace) ~ 10 kB
  - sizeof(survey) ~ 1 TB

Seismic Imaging
Physical process

- Use an approximate model of the earth
  - "smooth" large-scale model
- Propagate all the data to all possible reflection locations in the earth model
  - This all-to-all process sets the scale of computation
    - Computational cost ~ sizeof(data) x sizeof(image)
    - Terabytes of data to billions of image locations
    - Can be parallelized in many ways
- These reflection signals will
  - Constructively interfere at the correct location
  - Destructively interfere at the incorrect locations

Seismic Imaging
Iterative inversion

Construct initial earth model

Perform imaging

Image Self-consistent ?

No → Update earth model → Perform imaging

Yes → Done

Seismic Imaging

Why GPUs?

HESS

- Price-to-performance ratio improvement
  - Want 10X to change platforms
    - Payback must more than cover effort & risk
    - Got 10X ten years ago in switching from supercomputers to PC clusters

Seismic Imaging
Why GPUs?

- Price-to-performance ratio improvement
  - Want 10X to change platforms
    - Payback must more than cover effort & risk
    - Got 10X ten years ago in switching from supercomputers to PC clusters
  - Raw performance, benchmarks and simple prototypes on GPUs were of order 10X or more

Seismic Imaging
Why CUDA & NVIDIA GPUs?

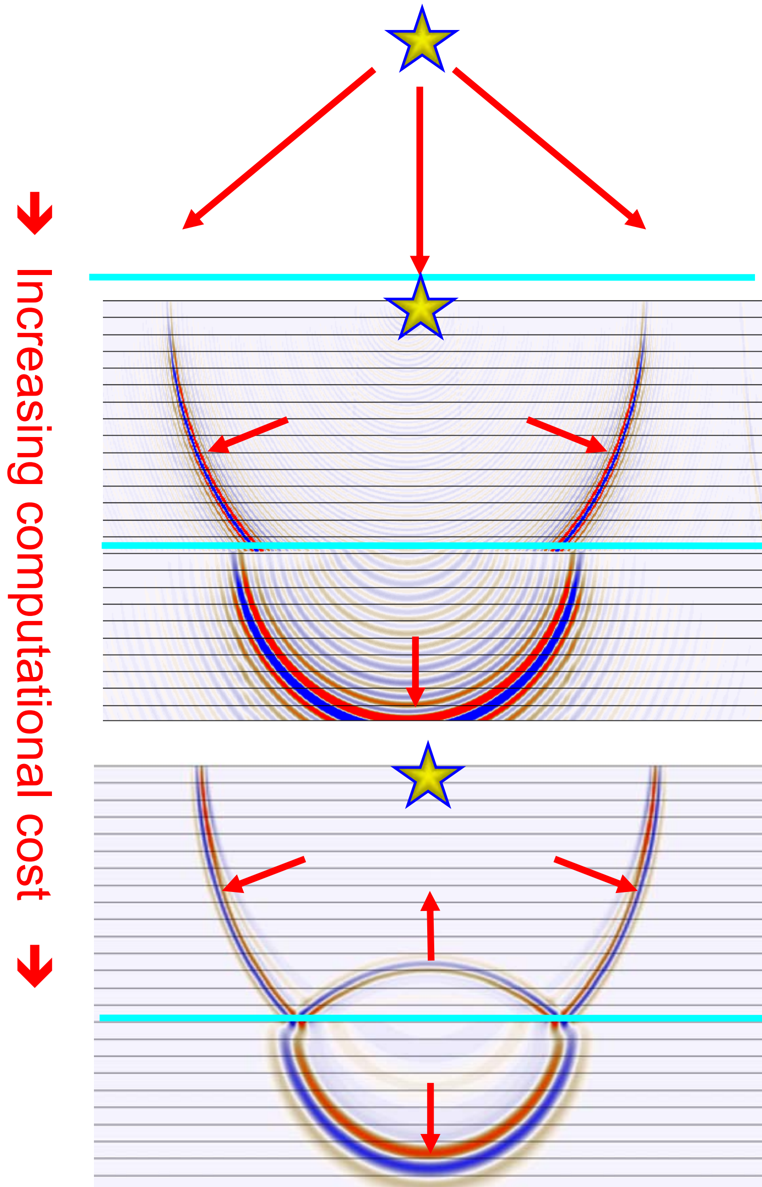
- Ease of programming
  - Must be able to port, maintain & modify production codes (relatively) easily
  - Have tried Cg, Brook and Peakstream
    - All lacking in some aspect
  - CUDA programming model straightforward
    - SIMD-like thread-based parallelism
    - In 1.5 days
      - Took "intro to CUDA" class
      - Wrote a working 2-D seismic modeling code
    - Programming memory hierarchy for optimization is biggest challenge

Seismic Imaging
How?

- **Steps to performance testing & porting**
  - Design GPU algorithm
    - Should keep main data structures in GPU memory
  - Create prototype GPU kernel
    - Include main computational characteristics
    - Test performance against CPU kernel
    - Iteratively refine prototype
  - Port full kernel & compare with CPU kernel
    - Verify numerical results
    - Compare performance results
  - Incorporate into production code & system

Seismic Imaging

Imaging methods



- **Kirchhoff imaging**
  - High-frequency propagation
  - Ray or eikonal travel-times
- **"Wave-equation" imaging**
  - One-way propagation: z ~ t
  - Frequency-domain method
  - ADI (alternating direction implicit) finite difference
- **"Reverse-time" imaging**
  - Two-way propagation
  - Time-domain
  - Explicit finite-difference

Kirchhoff Imaging
Physical algorithm

- Based on the Kirchhoff integral
  - Pre-compute coarse travel-times for propagation from surface locations to image points: $\mathrm{T}(\vec{\mathbf{s}}, \vec{\mathbf{x}})$
  - 4-D surface integral through a 5-D data set

$$\mathrm{I}(\vec{\mathbf{x}}) = \iint d^2\mathbf{s}\; d^2\mathbf{r}\; \mathrm{D}\big(\vec{\mathbf{s}}, \vec{\mathbf{r}}, t = \mathrm{T}(\vec{\mathbf{s}}, \vec{\mathbf{x}}) + \mathrm{T}(\vec{\mathbf{x}}, \vec{\mathbf{r}})\big)$$

  - Computational complexity:
    - $N_\mathrm{I} \sim 10^9$ is the number of output image points
    - $N_\mathrm{D} \sim 10^8$ is the number of input data traces
    - $f \sim 10$ is the number of cycles/point/trace
    - $f\, N_\mathrm{I}\, N_\mathrm{D} \sim 10^{18}$ cycles ~ 10 CPU-years

Kirchhoff Imaging

Computational kernel

Source s

Receiver r

y

x

z

T_S

T_R

$t = T_S + T_R$

Image point x

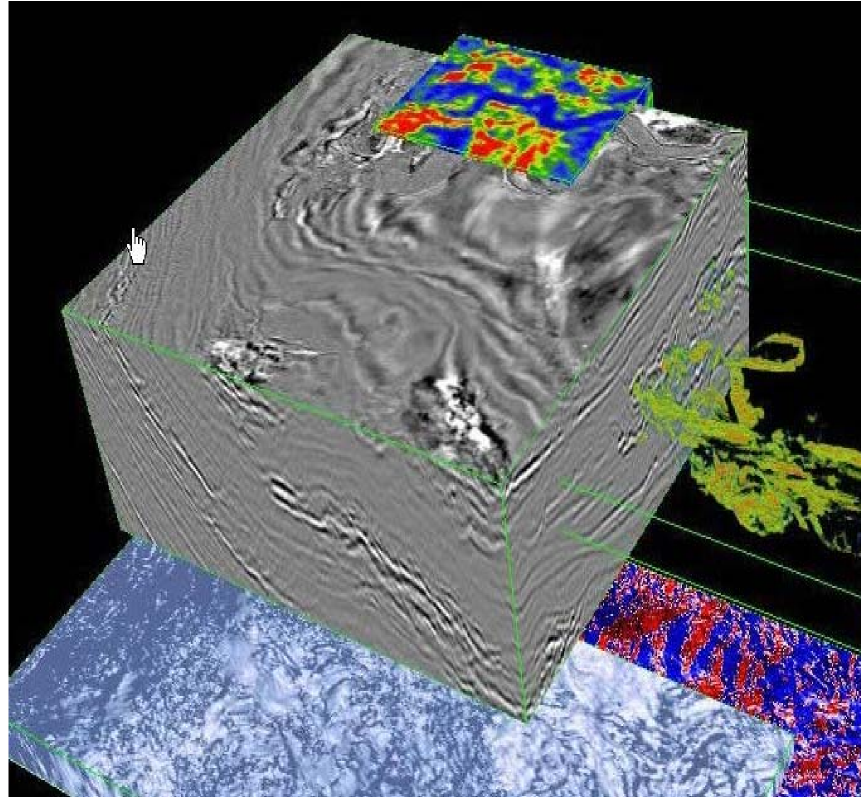
Add to Image

Image trace

Data trace

$$I(\vec{\mathbf{x}}) = \sum_{\vec{\mathbf{s}},\vec{\mathbf{r}}} D\left(\vec{\mathbf{s}}, \vec{\mathbf{r}}, t = T(\vec{\mathbf{s}}, \vec{\mathbf{x}}) + T(\vec{\mathbf{x}}, \vec{\mathbf{r}})\right)$$

Kirchhoff Imaging
Computational task



The problem is broken up into tasks each with a subset of the image and of the data.

$$I(\vec{\mathbf{x}}) = \sum_{\vec{\mathbf{s}},\vec{\mathbf{r}}} D\left(\vec{\mathbf{s}}, \vec{\mathbf{r}}, t = T(\vec{\mathbf{s}}, \vec{\mathbf{x}}) + T(\vec{\mathbf{x}}, \vec{\mathbf{r}})\right)$$

Kirchhoff Imaging
CUDA kernel

y

x

Travel-Time Cell

Column of
Travel-Time Cells

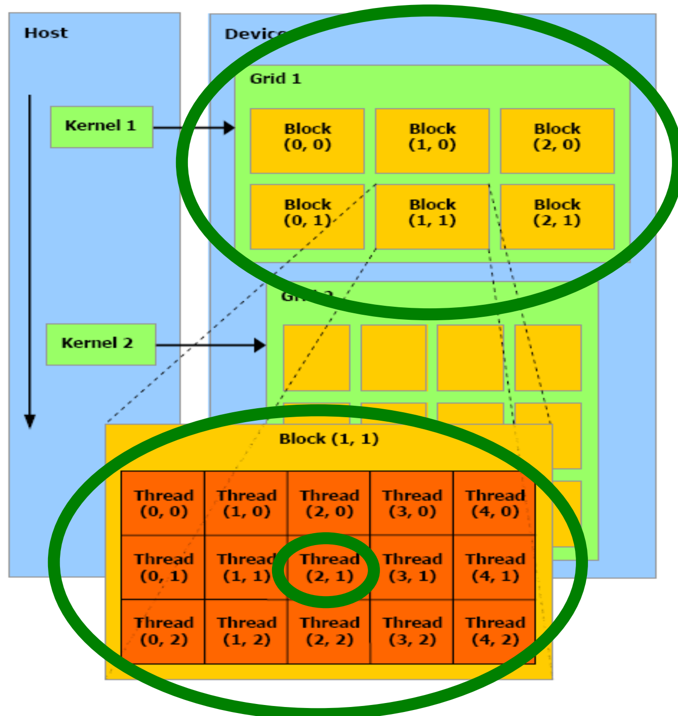
Image Point

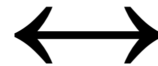
Travel-time Point
$\mathrm{T}(\vec{\mathbf{s}}, \vec{\mathbf{x}}, \vec{\mathbf{r}})$

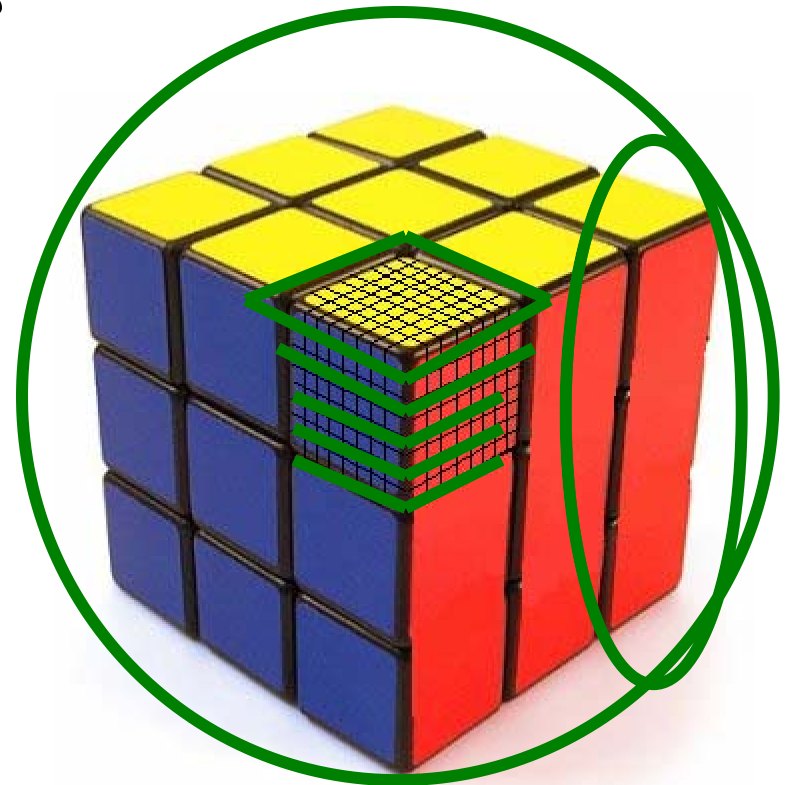
z

Kirchhoff Imaging
CUDA kernel

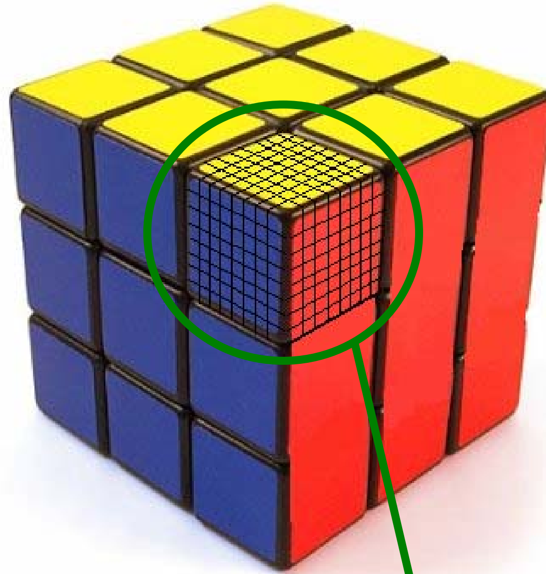
- Grid block: the full image for the task
- Thread block: one travel-time column
- Thread: one image trace (fixed x & y), looping over z & data traces







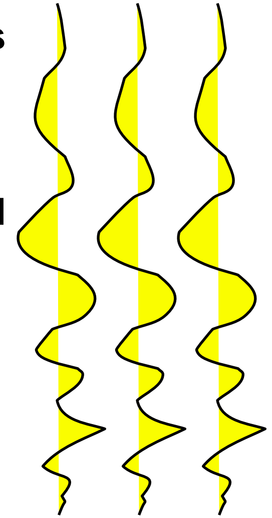
Kirchhoff Imaging
CUDA kernel

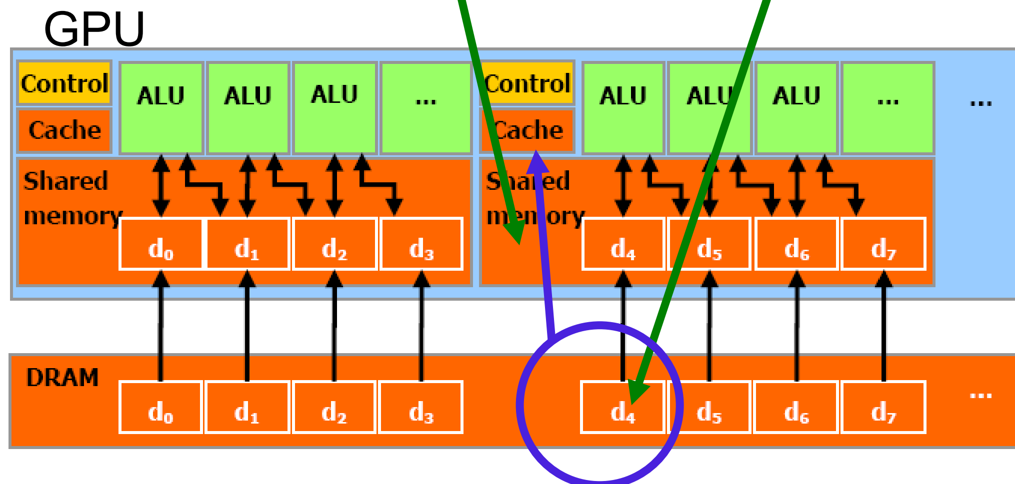


Data traces

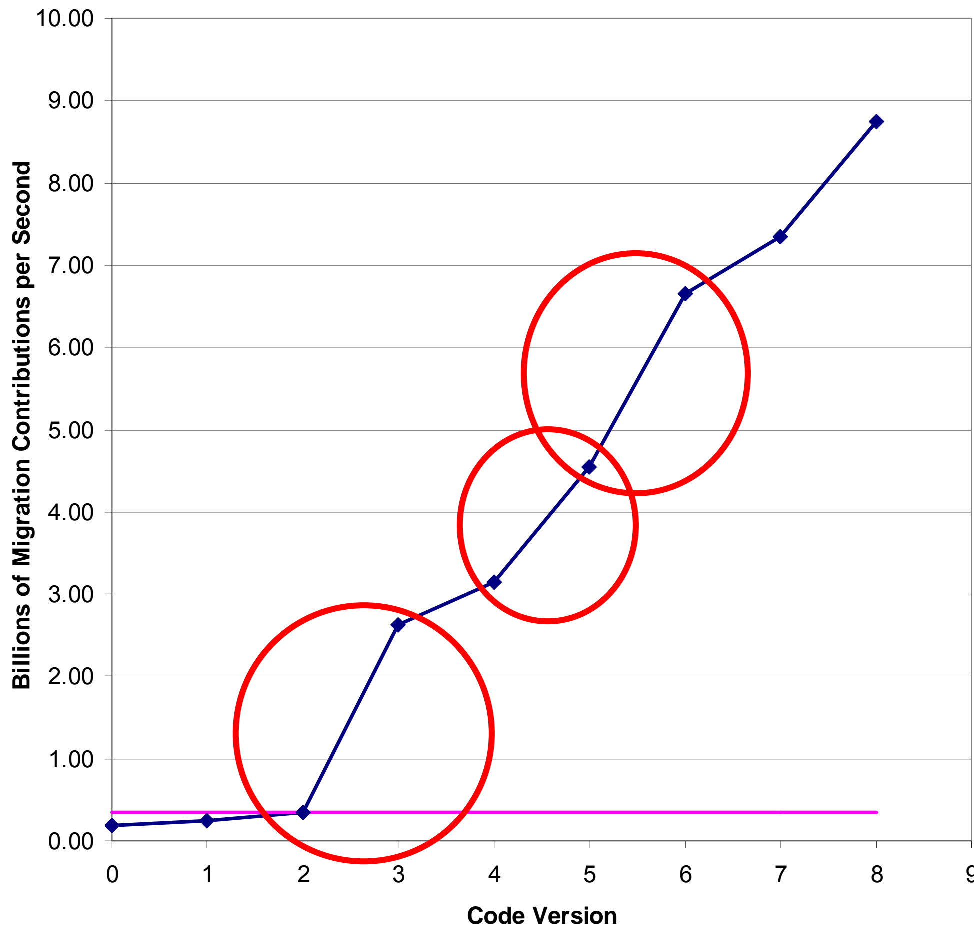
In texture

Get cached





Kirchhoff Imaging
Kernel optimization



0 – Initial Kernel

1 – Used Texture Memory

2 – Used Shared Memory

3 – Global Memory Coalescing

4 – Decreased Data Trace Shared Memory Use

5 – Optimized Use of Shared Memory

6 – Consolidated "if" Statements, Eliminated or Substituted Some Math Operations

7 – Removed an "if" and "for"

8 – Used Texture Memory for Data-Trace Fetch

Kirchhoff Imaging
Kernel performance

GPU-to-CPU Performance Ratio

GPU Speed-up

100
80
60
40
20
0

2 4 6 8 10 12 14 16 18

image points per travel-time cell in x or y

- CUDA 2D Tex (nix=4)
- CUDA Linear Tex (nix=4)
- CUDA 2D Tex (niy=4)
- CUDA Linear Tex (niy=4)
- CUDA 2D Tex (nix=4) G2
- CUDA 2D Tex (nix=4) G2 PIN
- CUDA Linear Tex (nix=4) G2
- CUDA Linear Tex (nix=4) G2 PIN
- CUDA 2D Tex (niy=4) G2
- CUDA 2D Tex (niy=4) G2 PIN
- CUDA Linear Tex (niy=4) G2
- CUDA Linear Tex (niy=4) G2 PIN

Kirchhoff Imaging

Kernel performance

GPU-to-CPU Performance Ratio

GPU Speed-up

0
20
40
60
80
100

2
4
6
8
10
12
14
16
18

image points per travel-time cell in x or y

CUDA 2D Tex (nix=4)
CUDA Linear Tex (nix=4)
CUDA 2D Tex (niy=4)
CUDA Linear Tex (niy=4)
CUDA 2D Tex (nix=4) G2
CUDA 2D Tex (nix=4) G2 PIN
CUDA Linear Tex (nix=4) G2
CUDA Linear Tex (nix=4) G2 PIN
CUDA 2D Tex (niy=4) G2
CUDA 2D Tex (niy=4) G2 PIN
CUDA Linear Tex (niy=4) G2
CUDA Linear Tex (niy=4) G2 PIN

Tesla
C870

Kirchhoff Imaging
Kernel performance

GPU-to-CPU Performance Ratio

Tesla C1060

GPU Speed-up

100
80
60
40
20
0

2 4 6 8 10 12 14 16 18

image points per travel-time cell in x or y

CUDA 2D Tex (nix=4)
CUDA Linear Tex (nix=4)
CUDA 2D Tex (niy=4)
CUDA Linear Tex (niy=4)
CUDA 2D Tex (nix=4) G2
CUDA 2D Tex (nix=4) G2 PIN
CUDA Linear Tex (nix=4) G2
CUDA Linear Tex (nix=4) G2 PIN
CUDA 2D Tex (niy=4) G2
CUDA 2D Tex (niy=4) G2 PIN
CUDA Linear Tex (niy=4) G2
CUDA Linear Tex (niy=4) G2 PIN

Kirchhoff Imaging

Kernel performance

GPU-to-CPU Performance Ratio

Typical parameter range

GPU Speed-up: 0, 20, 40, 60, 80, 100

image points per travel-time cell in x or y: 2, 4, 6, 8, 10, 12, 14, 16, 18

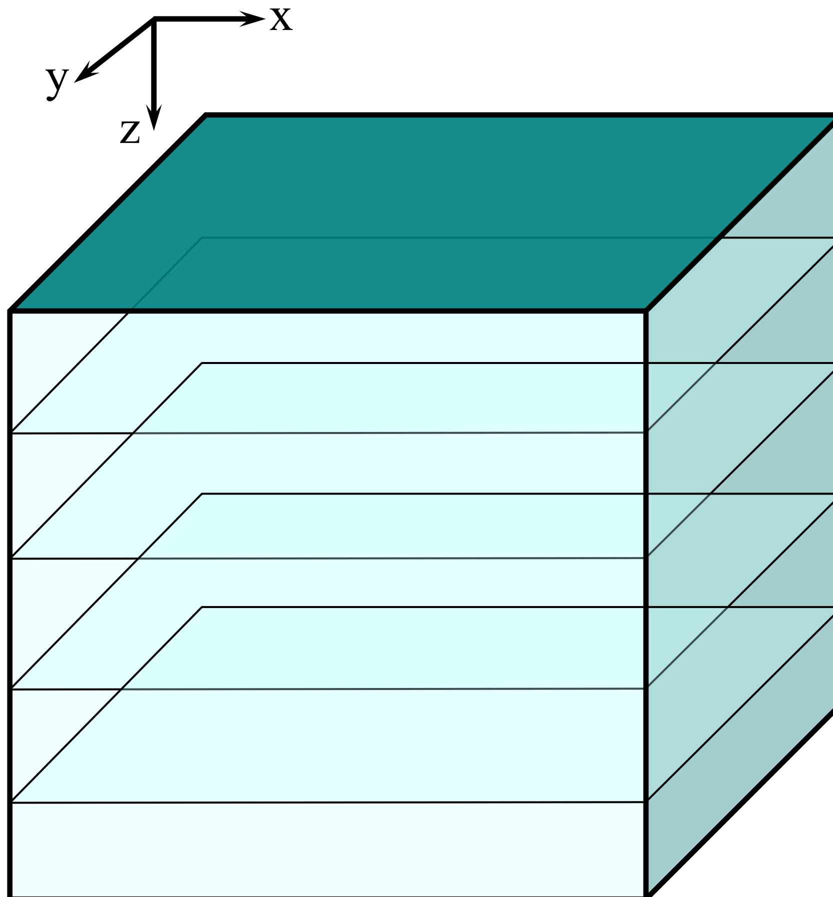
- CUDA 2D Tex (nix=4)
- CUDA Linear Tex (nix=4)
- CUDA 2D Tex (niy=4)
- CUDA Linear Tex (niy=4)
- CUDA 2D Tex (nix=4) G2
- CUDA 2D Tex (nix=4) G2 PIN
- CUDA Linear Tex (nix=4) G2
- CUDA Linear Tex (nix=4) G2 PIN
- CUDA 2D Tex (niy=4) G2
- CUDA 2D Tex (niy=4) G2 PIN
- CUDA Linear Tex (niy=4) G2
- CUDA Linear Tex (niy=4) G2 PIN

Kirchhoff Imaging
Production status

- Kernel incorporated into production code
- Starting production testing
  - Large kernel speed-ups results in “overhead” dominating GPU production runs
  - To do: create kernels for “overhead” components

"Wave-equation" Imaging

One-way propagation



- Based on scalar wave equation

$$\frac{1}{V(\vec{\mathbf{x}})^2}\frac{\partial^2 P}{\partial t^2} = \nabla^2 P$$

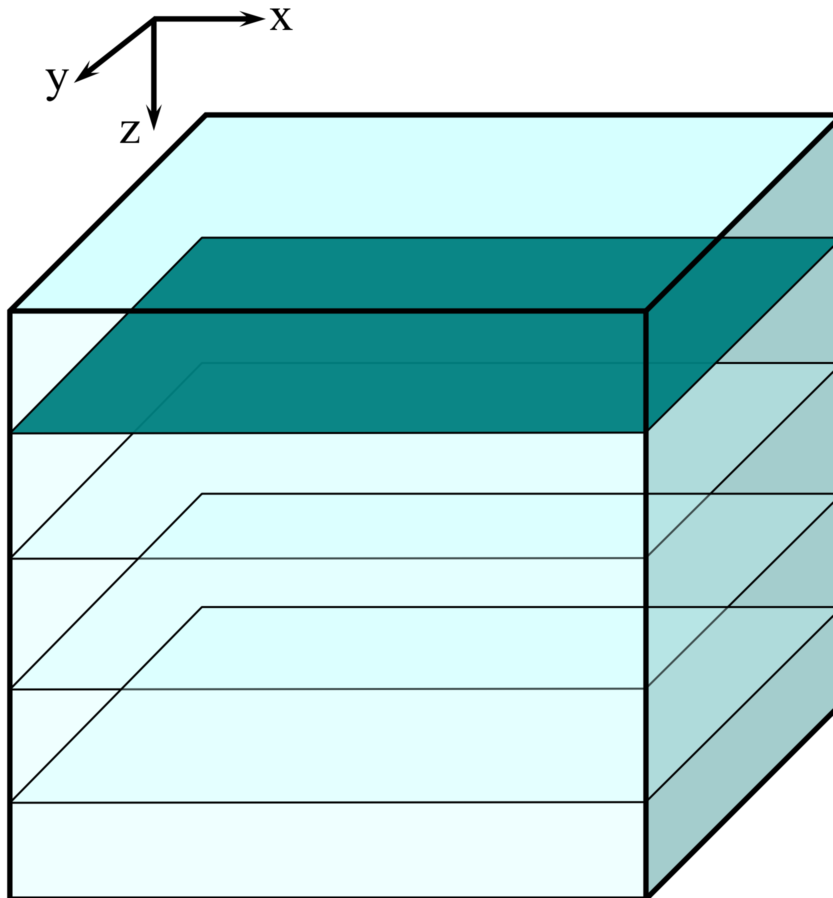
- Frequency-domain
- Preferred direction of propagation: z ~ t

$$\frac{\partial P}{\partial z} = \frac{\pm i\omega}{V(\vec{\mathbf{x}})}\sqrt{1 + \frac{V^2(\vec{\mathbf{x}})}{\omega^2}\left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}\right)}\; P$$

- Evolution in depth

"Wave-equation" Imaging

One-way propagation



- Based on scalar wave equation

$$\frac{1}{V(\vec{\mathbf{x}})^2}\frac{\partial^2 P}{\partial t^2} = \nabla^2 P$$

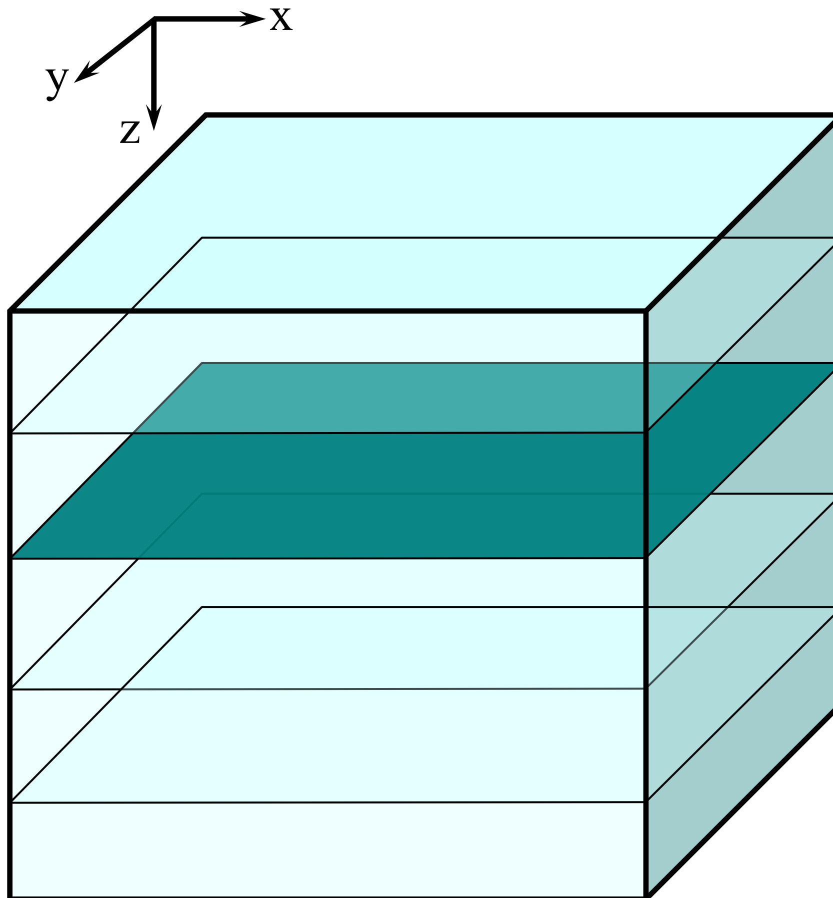
- Frequency-domain
- Preferred direction of propagation: z ~ t

$$\frac{\partial P}{\partial z} = \frac{\pm i\omega}{V(\vec{\mathbf{x}})}\sqrt{1 + \frac{V^2(\vec{\mathbf{x}})}{\omega^2}\left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}\right)}\; P$$

- Evolution in depth

"Wave-equation" Imaging

One-way propagation



- Based on scalar wave equation

$$\frac{1}{V(\vec{\mathbf{x}})^2}\frac{\partial^2 P}{\partial t^2} = \nabla^2 P$$

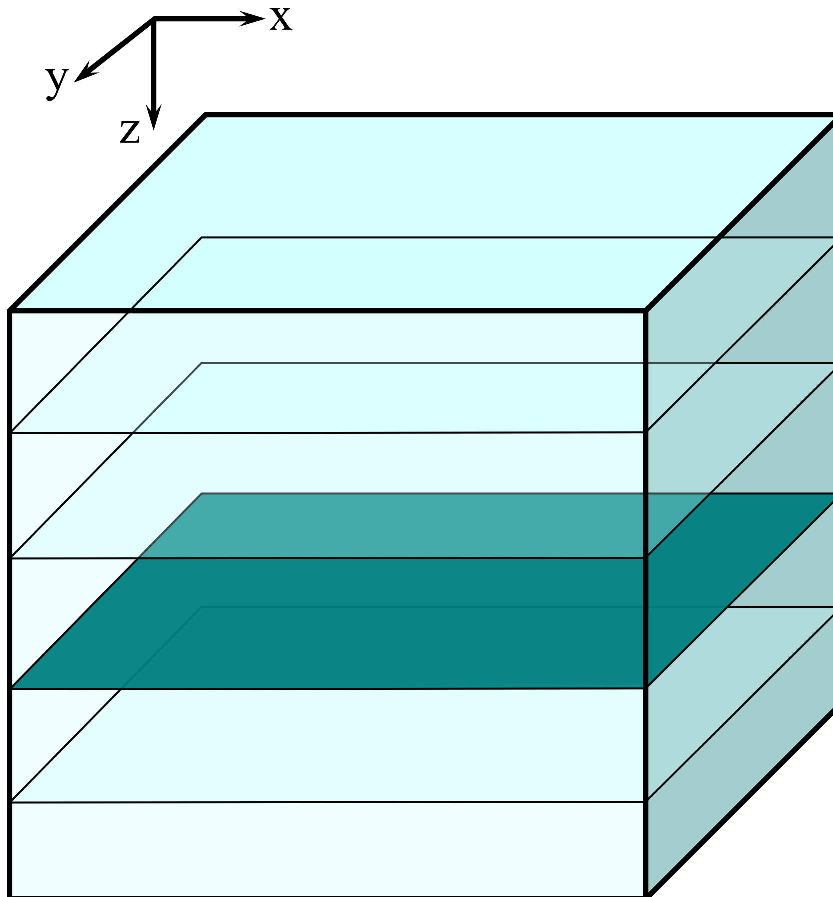
- Frequency-domain
- Preferred direction of propagation: z ~ t

$$\frac{\partial P}{\partial z} = \frac{\pm i\omega}{V(\vec{\mathbf{x}})}\sqrt{1+\frac{V^2(\vec{\mathbf{x}})}{\omega^2}\left(\frac{\partial^2}{\partial x^2}+\frac{\partial^2}{\partial y^2}\right)}\; P$$

- Evolution in depth

"Wave-equation" Imaging

One-way propagation



- Based on scalar wave equation

$$\frac{1}{V(\vec{\mathbf{x}})^2}\frac{\partial^2 P}{\partial t^2} = \nabla^2 P$$

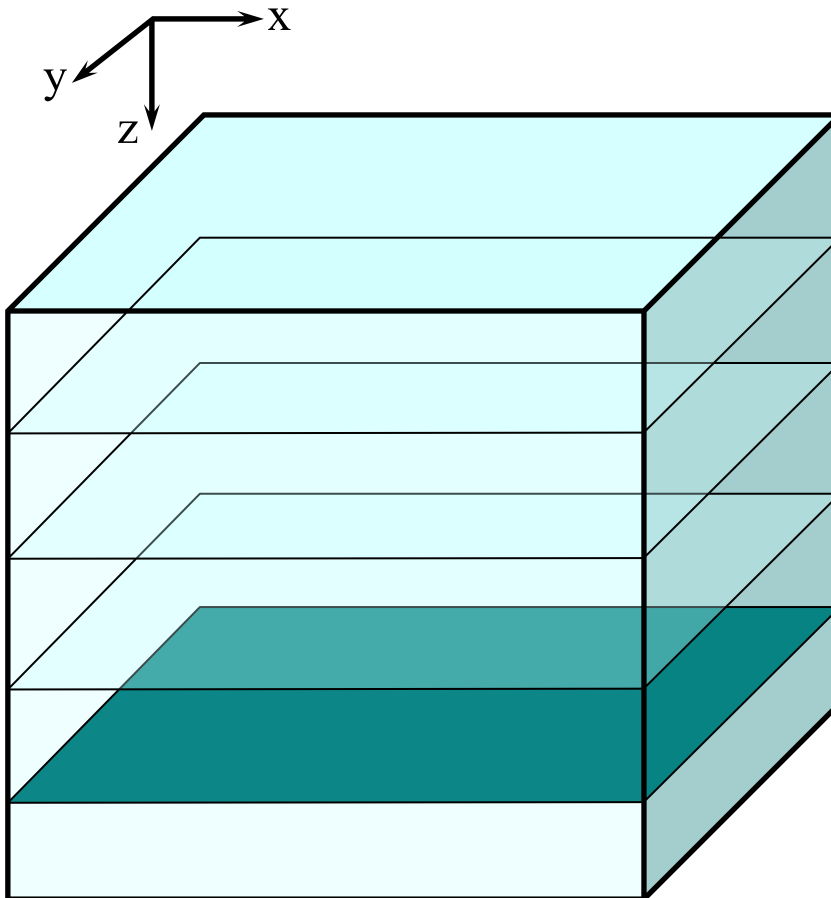
- Frequency-domain
- Preferred direction of propagation: z ~ t

$$\frac{\partial P}{\partial z} = \frac{\pm i\omega}{V(\vec{\mathbf{x}})}\sqrt{1+\frac{V^2(\vec{\mathbf{x}})}{\omega^2}\left(\frac{\partial^2}{\partial x^2}+\frac{\partial^2}{\partial y^2}\right)}\; P$$

- Evolution in depth

"Wave-equation" Imaging

One-way propagation



- Based on scalar wave equation

$$\frac{1}{V(\vec{\mathbf{x}})^2}\frac{\partial^2 P}{\partial t^2} = \nabla^2 P$$

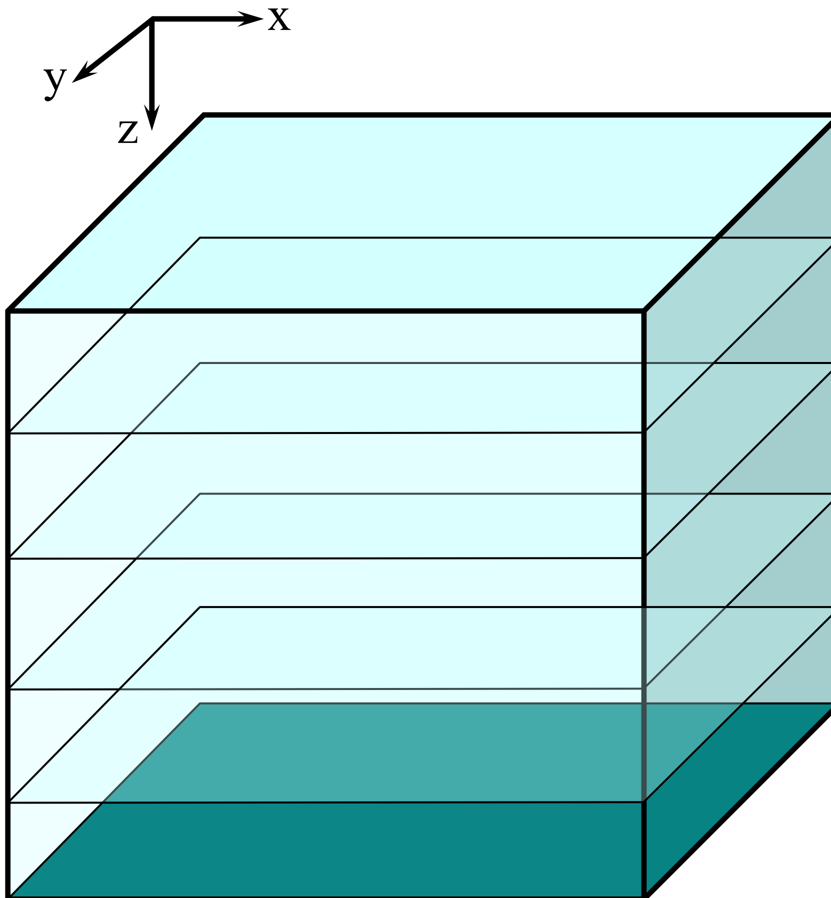
- Frequency-domain
- Preferred direction of propagation: z ~ t

$$\frac{\partial P}{\partial z} = \frac{\pm i\omega}{V(\vec{\mathbf{x}})}\sqrt{1+\frac{V^2(\vec{\mathbf{x}})}{\omega^2}\left(\frac{\partial^2}{\partial x^2}+\frac{\partial^2}{\partial y^2}\right)}\; P$$

- Evolution in depth

“Wave-equation” Imaging

One-way propagation



- Based on scalar wave equation

$$\frac{1}{V(\vec{\mathbf{x}})^2}\frac{\partial^2 P}{\partial t^2} = \nabla^2 P$$

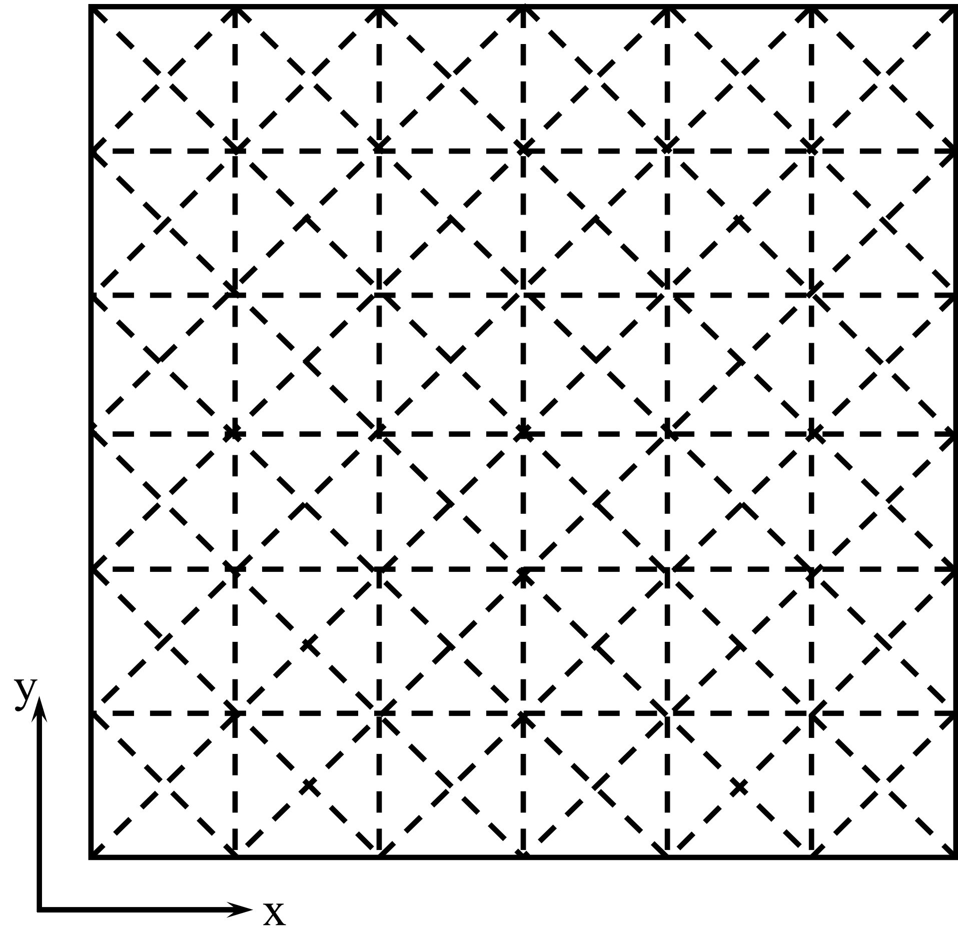
- Frequency-domain
- Preferred direction of propagation: z ~ t

$$\frac{\partial P}{\partial z} = \frac{\pm i\omega}{V(\vec{\mathbf{x}})}\sqrt{1 + \frac{V^2(\vec{\mathbf{x}})}{\omega^2}\left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}\right)}\; P$$

  - Evolution in depth

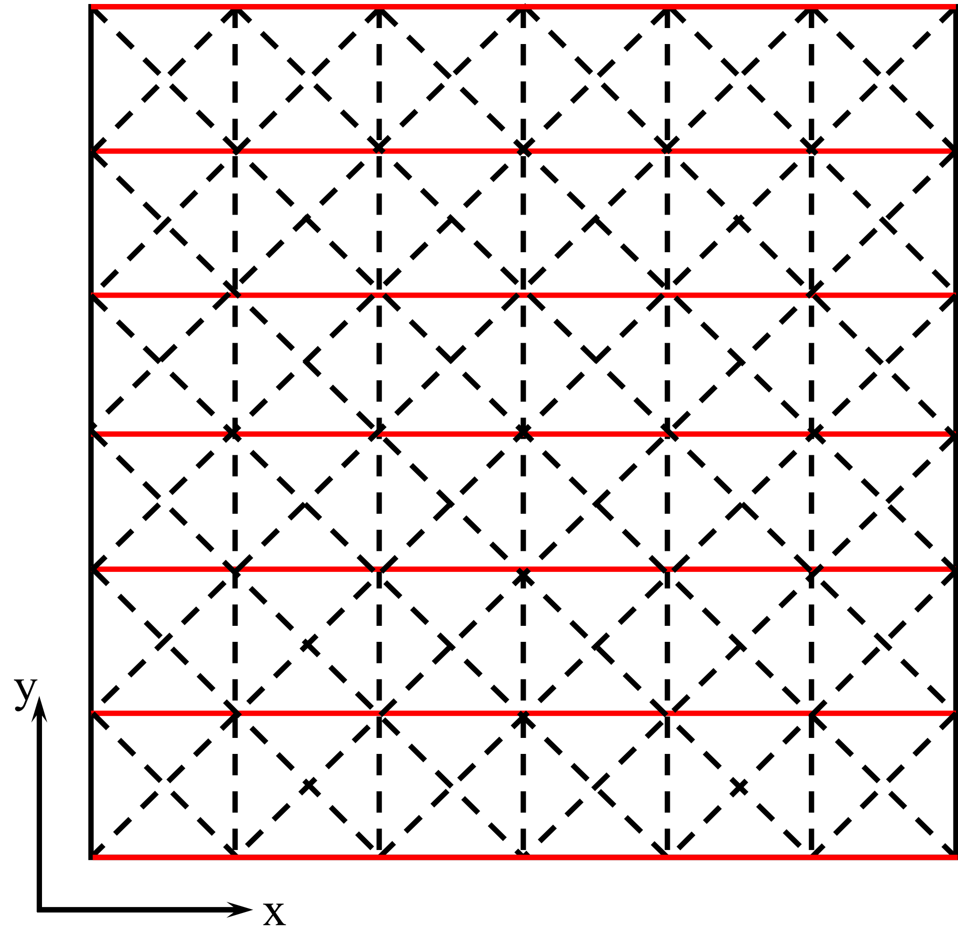
"Wave-equation" Imaging

One-way propagation



- Evolution eqn uses
  - Continued fractions
  - Operator splitting
  - ADI finite difference
- Each depth step requires applying four operators
  - Along x
  - Along y
  - Along x + y
  - Along x – y

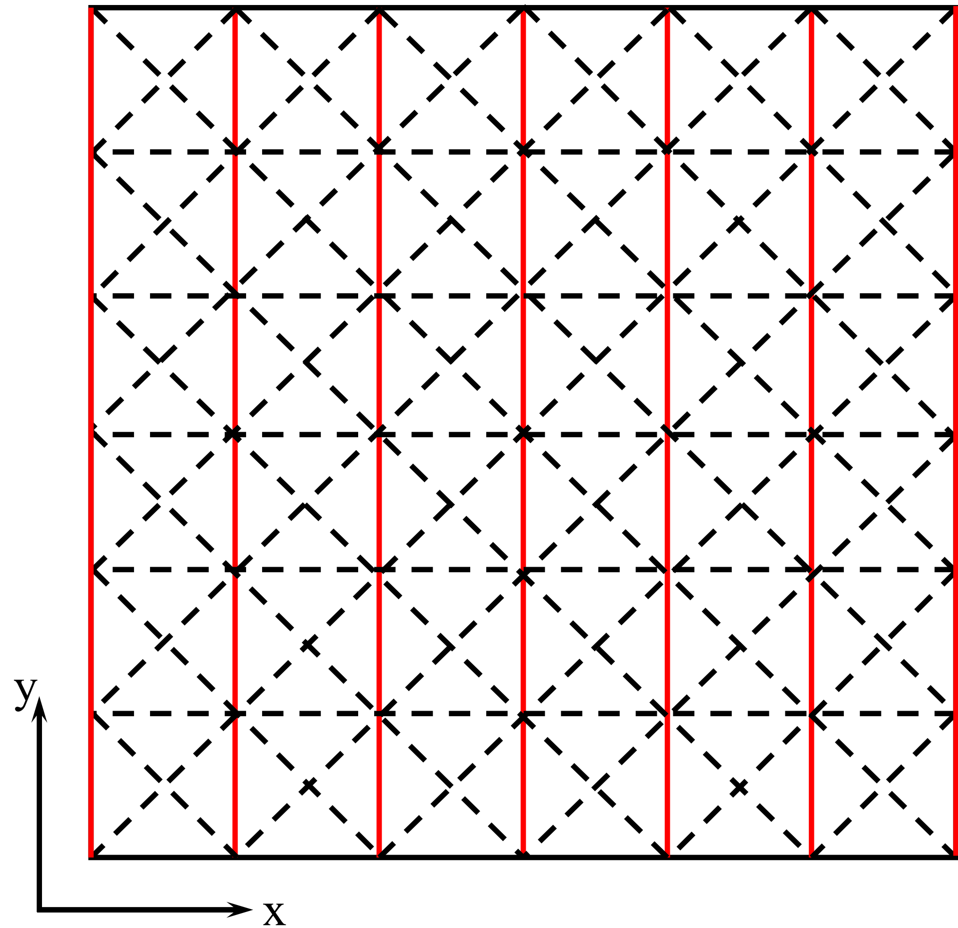
“Wave-equation” Imaging
One-way propagation



- Evolution eqn uses
  - Continued fractions
  - Operator splitting
  - ADI finite difference
- Each depth step requires applying four operators
  - Along x
  - Along y
  - Along x + y
  - Along x – y

“Wave-equation” Imaging

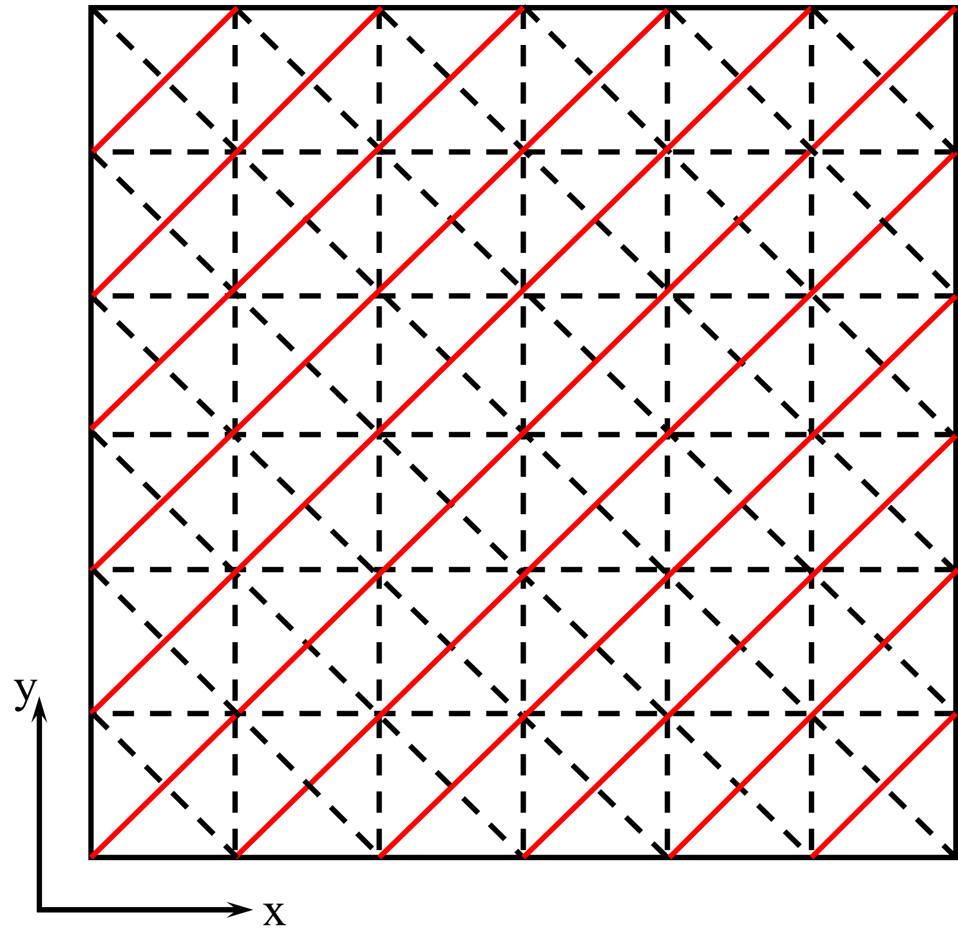
One-way propagation



- Evolution eqn uses
  - Continued fractions
  - Operator splitting
  - ADI finite difference
- Each depth step requires applying four operators
  - Along x
  - Along y
  - Along x + y
  - Along x – y

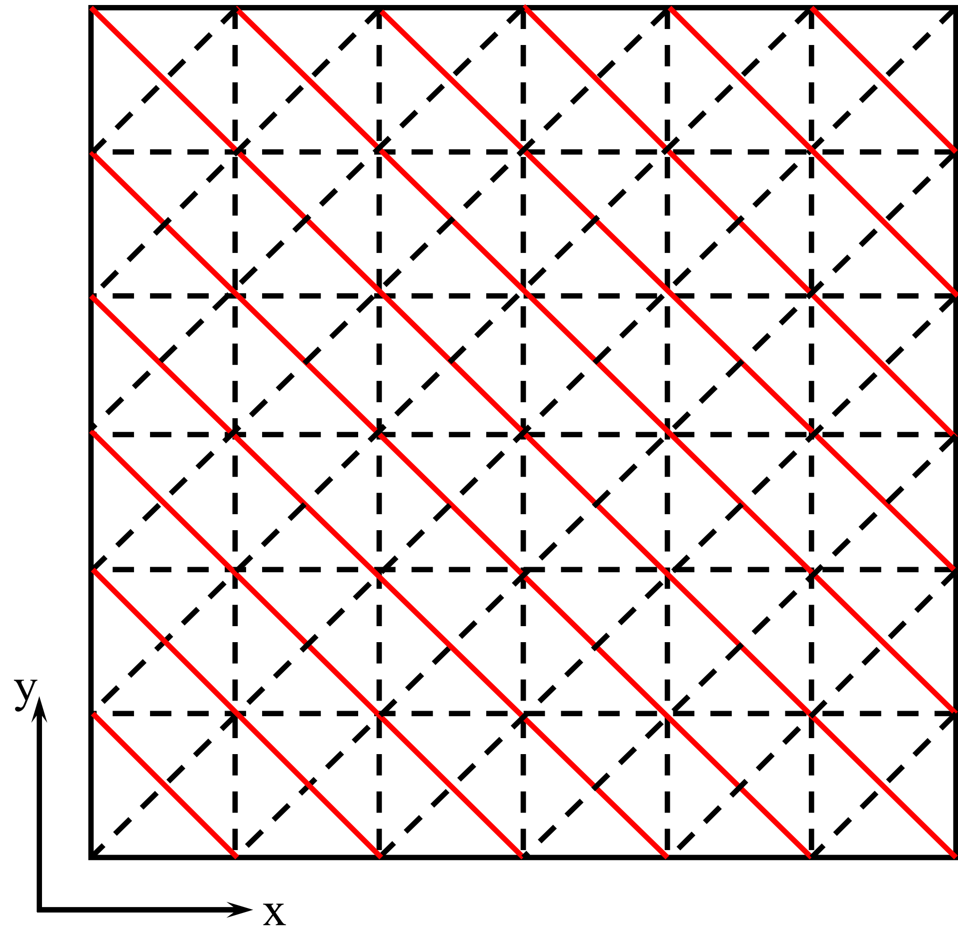
"Wave-equation" Imaging

One-way propagation



- Evolution eqn uses
  - Continued fractions
  - Operator splitting
  - ADI finite difference
- Each depth step requires applying four operators
  - Along x
  - Along y
  - Along x + y
  - Along x – y

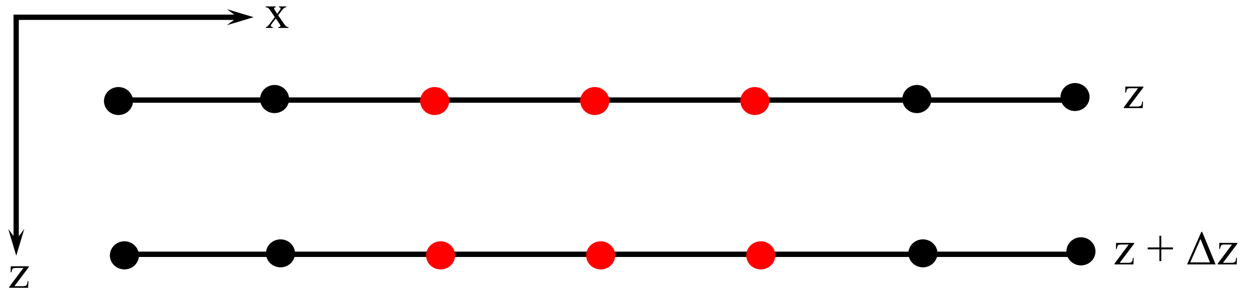
"Wave-equation" Imaging
One-way propagation



- Evolution eqn uses
  - Continued fractions
  - Operator splitting
  - ADI finite difference
- Each depth step requires applying four operators
  - Along x
  - Along y
  - Along x + y
  - Along x – y

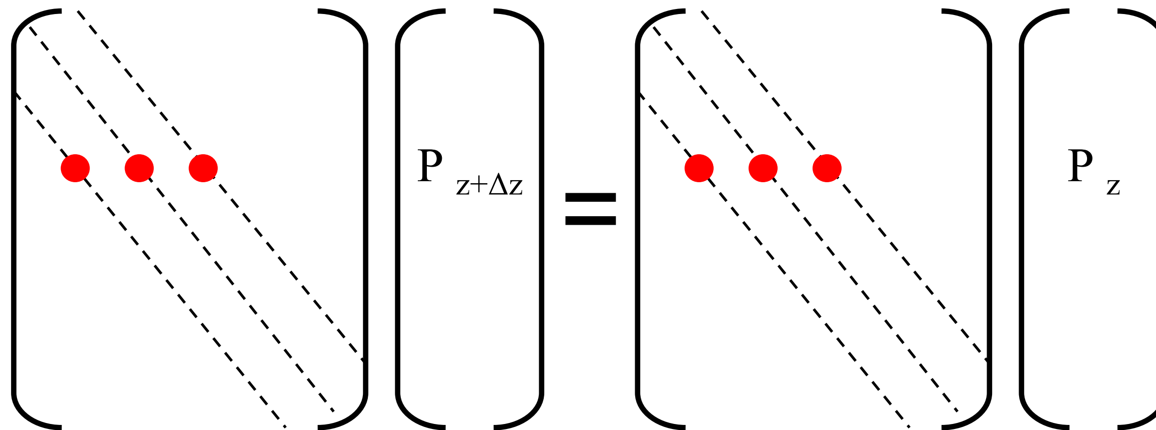
"Wave-equation" Imaging

Implicit complex tri-diagonal linear systems



$$A P^{z+\Delta z}_{x-\Delta x,y} + (1-2A) P^{z+\Delta z}_{x,y} + A P^{z+\Delta z}_{x+\Delta x,y} =$$

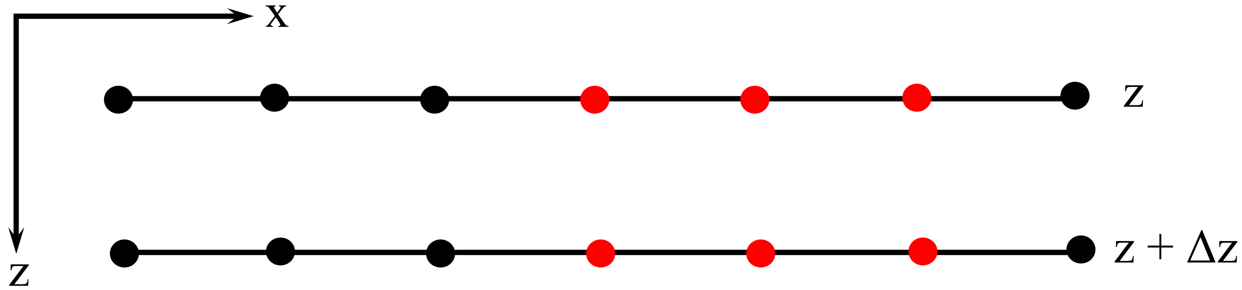
$$A^{*} P^{z}_{x-\Delta x,y} + (1-2A^{*}) P^{z}_{x,y} + A^{*} P^{z}_{x+\Delta x,y}$$



The evaluation and solution of these complex tri-diagonal systems dominates the computational cost of our wave-equation imaging code.

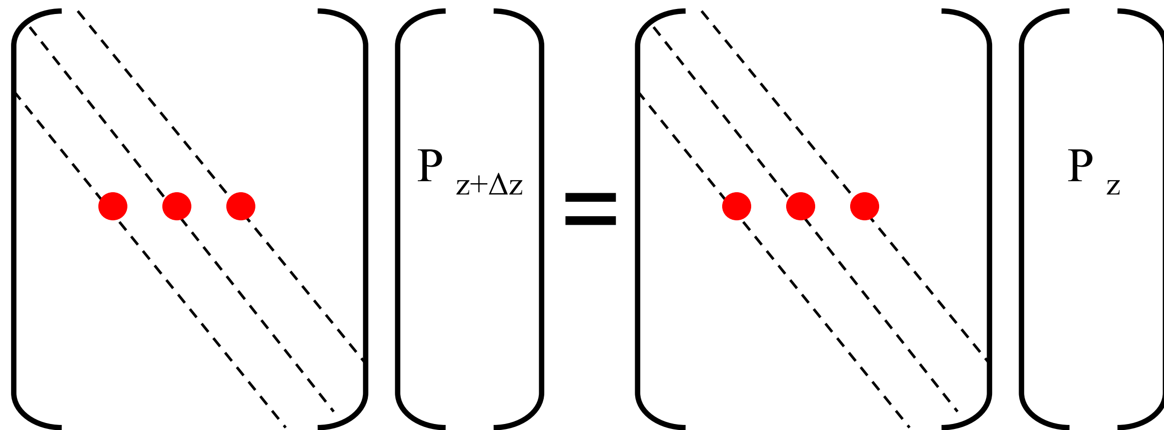
"Wave-equation" Imaging

Implicit complex tri-diagonal linear systems



$$AP^{z+\Delta z}_{x-\Delta x,y} + (1-2A)P^{z+\Delta z}_{x,y} + AP^{z+\Delta z}_{x+\Delta x,y} =$$

$$A^{*}P^{z}_{x-\Delta x,y} + (1-2A^{*})P^{z}_{x,y} + A^{*}P^{z}_{x+\Delta x,y}$$

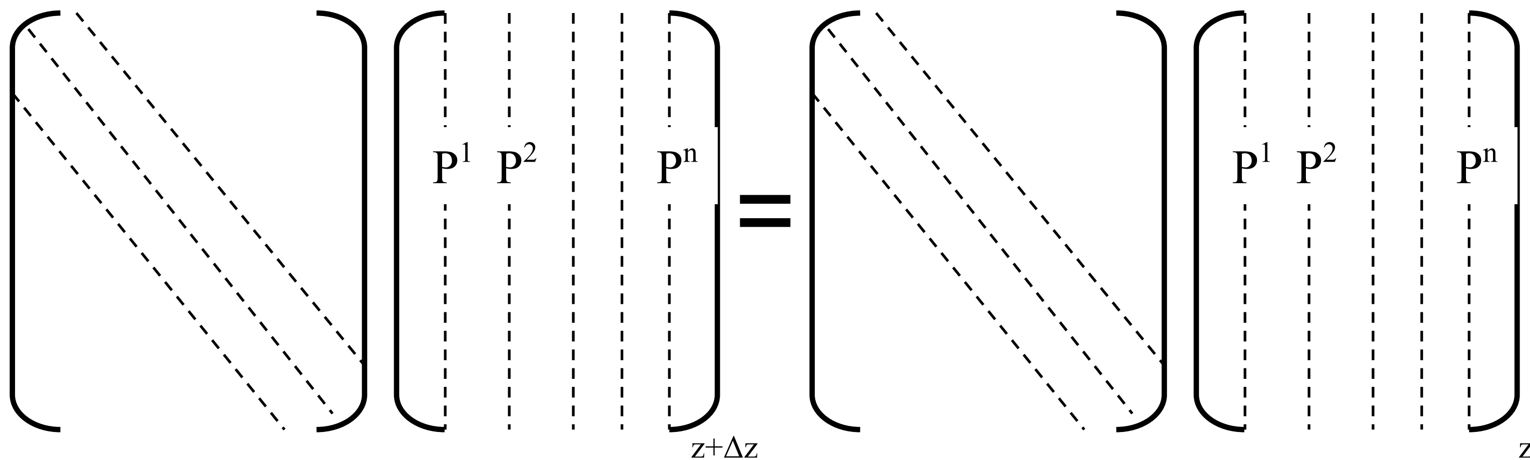


The evaluation and solution of these complex tri-diagonal systems dominates the computational cost of our wave-equation imaging code.

“Wave-equation” Imaging

Low level parallelism

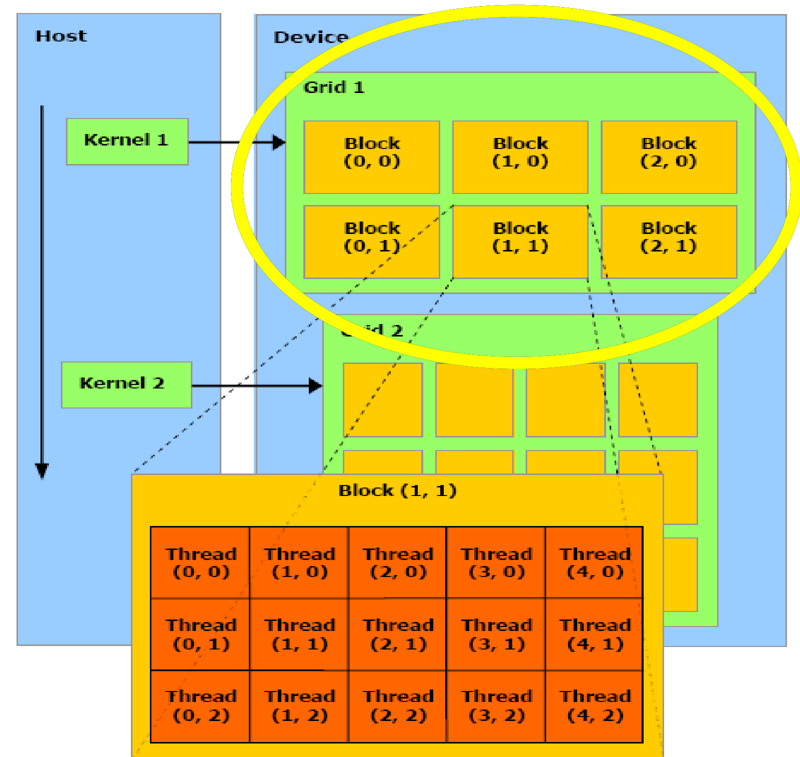
- Common work between shot-records
  - Calculating the coefficients of the matrices
    - Dependent on frequency & local velocity
  - Part of the solving of the tri-diagonal system
- Parallelize over shot-records in the kernel



"Wave-equation" Imaging

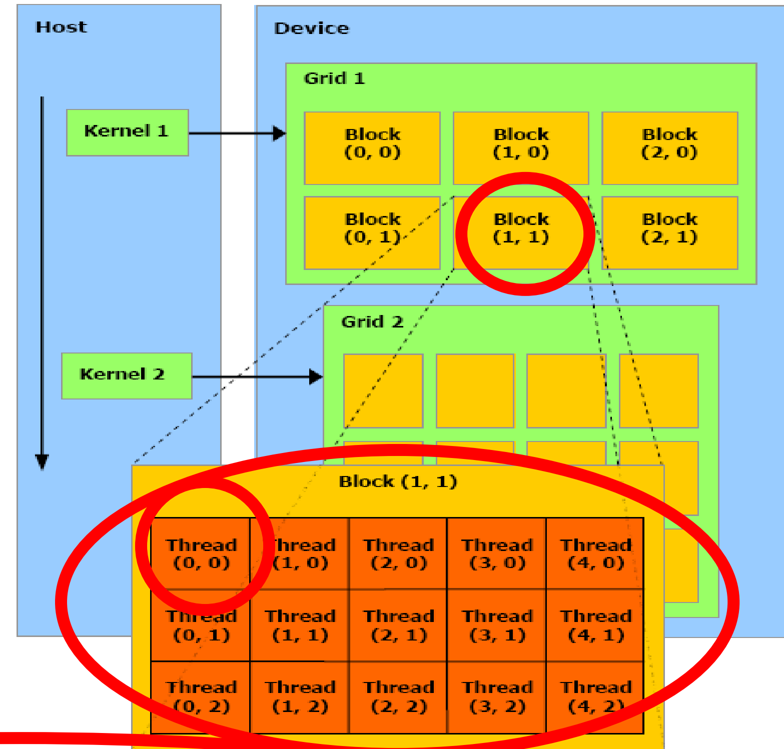
CUDA kernels

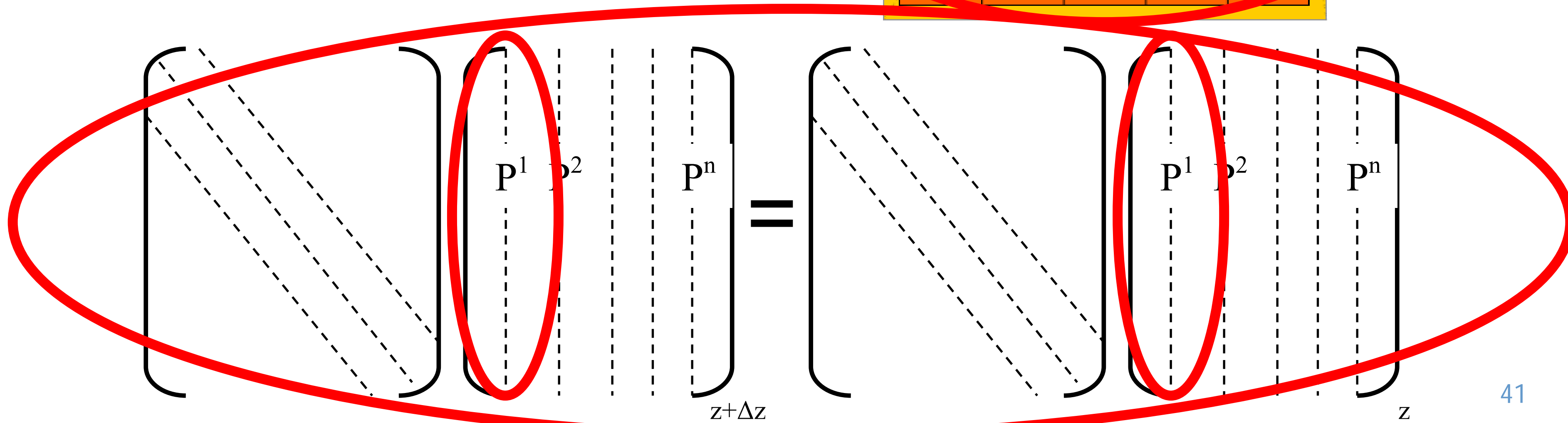
- Separate kernels for each operator
  - x, y, x+y and x-y



"Wave-equation" Imaging
CUDA kernels

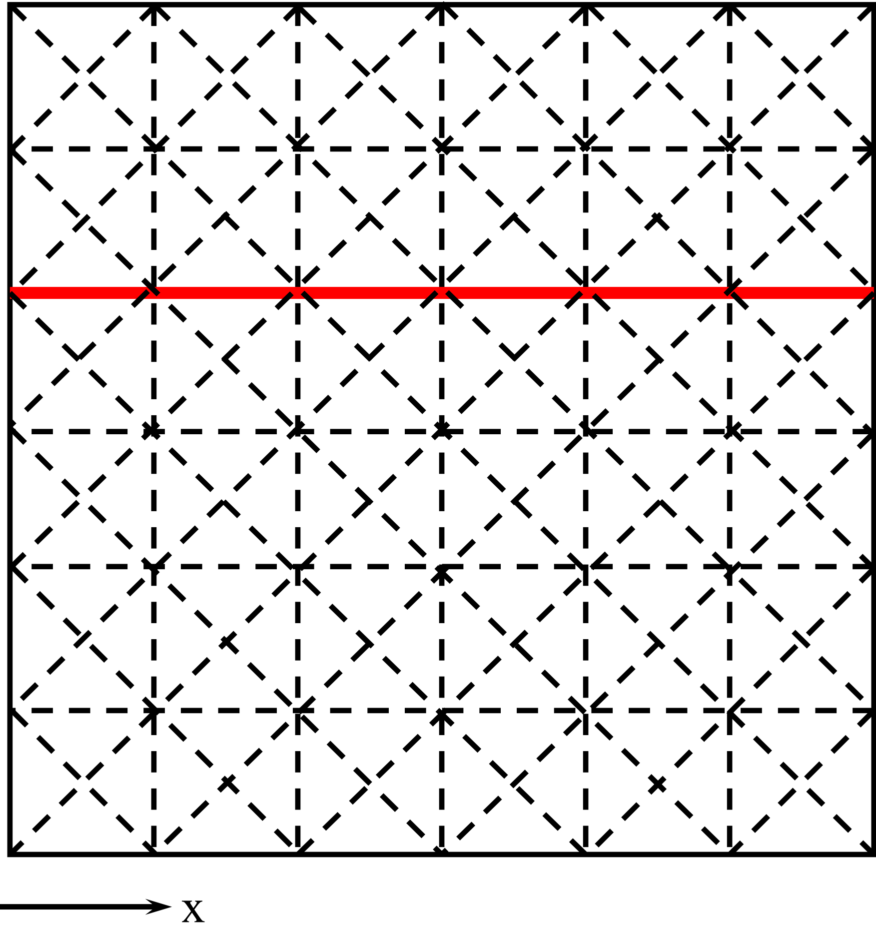
- Separate kernels for each operator
  - x, y, x+y and x-y

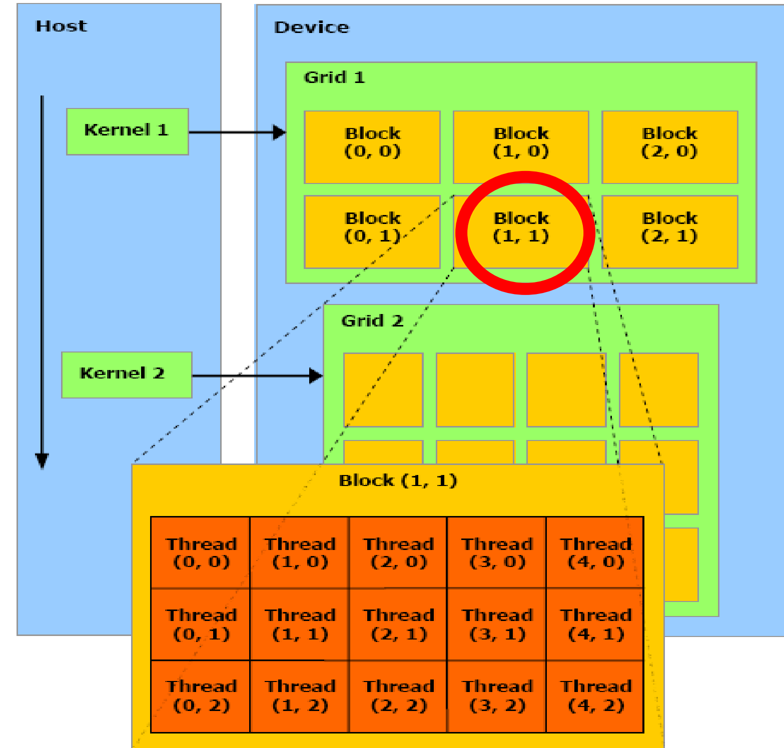




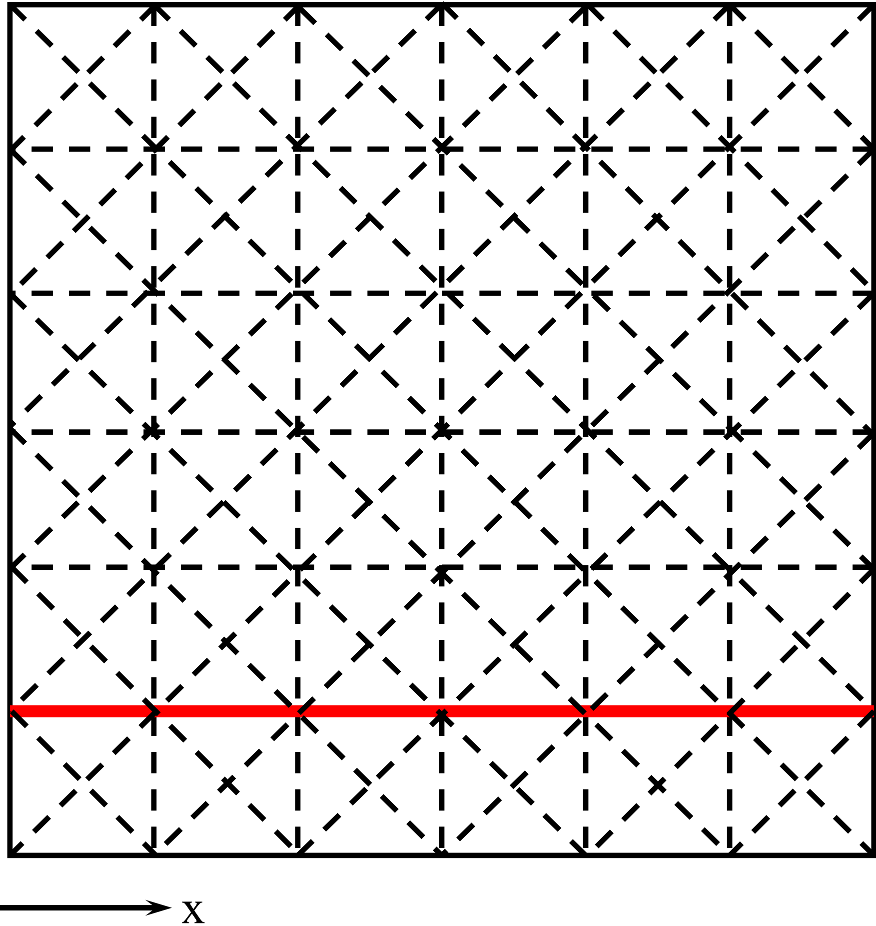
"Wave-equation" Imaging

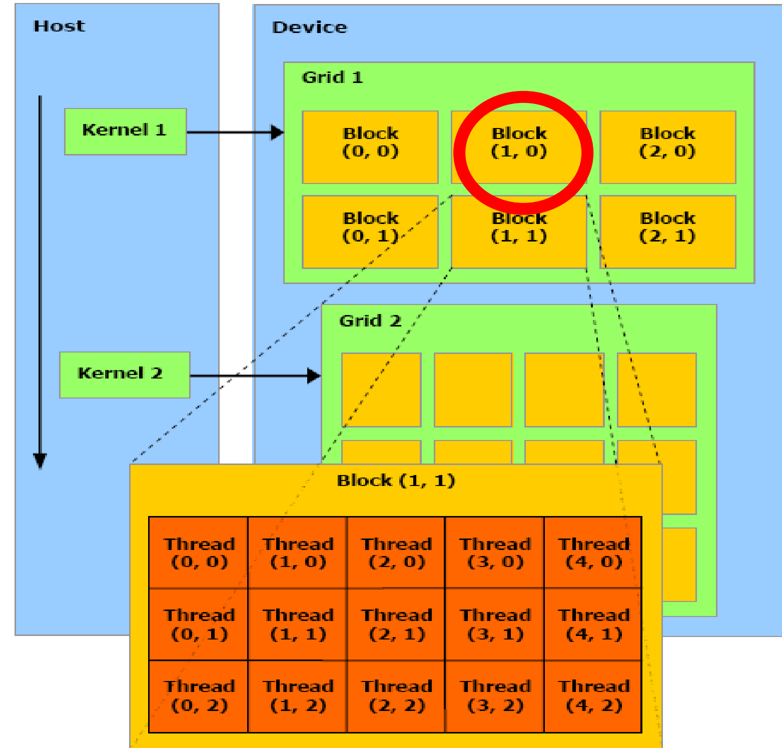
CUDA kernels





"Wave-equation" Imaging
CUDA kernels





"Wave-equation" Imaging
Performance

- Production CPU kernel
  - Performance: 15 – 50 Mpoints/sec
- Prototype CUDA kernel
  - Single tri-diagonal system
  - Constant coefficients
  - Performance: 700 Mpoints/sec

"Wave-equation" Imaging

Performance

Prototype Wave-equation Kernel

Normalized Execution Time

50
40
30
20
10
0

1 C870 GPU
1 Xeon (3.6 GHz)
2 Xeon (3.6 GHz)
1 CPU (dual quad-core)
2 CPUs (dual quad-core)
4 CPUs (dual quad-core)
6 CPUs (dual quad-core)

processor(s)

“Wave-equation” Imaging

Performance

- Production CPU kernel
  - Performance: 15 – 50 Mpoints/sec
- Prototype CUDA kernel
  - Single tri-diagonal system
  - Constant coefficients
  - Performance: 700 Mpoints/sec
- Production CUDA kernels
  - Single kernel handles x, y, x+y & x-y operators
  - Several kernels calculate coefficients
  - Performance: 280 Mpoints/sec

"Wave-equation" Imaging
Performance

Production Wave-equation Kernel

“Wave-equation” Imaging
Details swept under the rug

- Each shot-record requires the propagation of two wave-fields
  - one for the source $S_{shot}(\mathbf{x}, \omega)$
  - one for the receiver $R_{shot}(\mathbf{x}, \omega)$
- There is an additional kernel which calculates the image from these wave-fields

$$I(\mathbf{x}) = \sum_{\omega, shot} S^*_{shot}(\mathbf{x}, \omega) R_{shot}(\mathbf{x}, \omega)$$

  - Consumes 5 -10 % of run time

Production Imaging
In-house systems

- Prototype GPU cluster
  - 32 nodes, each containing
    - Dual quad-core system, 8 GB memory & 700 GB disk
    - Tesla C870: 4 G80 GPUs each with 1.5 GB memory
- Production CPU cluster
  - 1500 nodes, each containing
    - Dual Xeon (3.6 GHz)
    - 8 GB memory
    - ¼ or ½ TB disk

Production Imaging
Integration issues

- Do CPU & GPU kernels use the same data structures?
  - Yes: may not be conducive to best performance
    - Example: Kirchhoff "overhead" dominates
  - No: CPU & GPU codes diverge
    - Increased code maintenance effort/cost
    - Possible solution: common language?

Production Imaging
Integration issues

- **How to assign batch tasks for multi-GPU systems?**
  - C assign N tasks to each N-CPU or N-GPU system
  - For shared-memory multi-CPU systems, the OS manages the task processes & CPU resources
  - For distributed-memory multi-GPU (Tesla) systems, the task management must be done by “hand”
    - cudaSetDevice() enables the specification of GPU
    - How does the application know which GPUs are in use?
    - Currently must keep track of which GPUs are in use external to application

Production Imaging
Experiences

HESS

- Problem attribution
  - Few CUDA bugs
  - Infrequently due to hardware
    - None attributable to non-ECC memory
  - Some problems at “driver” level
    - Some “solved” by rebooting
    - Some require driver upgrade
  - Some due to other parts of the system
    - nfs, network, disks, etc
  - Some unexplained
    - Perhaps due to improper task-GPU allocation

Increasing problem frequency ↓

"Reverse-time" Imaging
Two-way propagation

- Based on the scalar wave-equation

$$\frac{1}{V(\vec{\mathbf{x}})^2}\frac{\partial^2 P}{\partial t^2} = \nabla^2 P$$

- Explicit finite difference scheme
  - 2nd order in time
  - 8th order in space
  - For time-step performance, see Paulius' talk
    - Using 16x16 tile algorithm (to fit 2 copies)
    - 5 reads-writes/point
    - 2500 - 3000 Mpoints/sec on C1060

"Reverse-time" Imaging

Two-way propagation

- Each shot-record requires the propagation of two wave-fields
  - one for the source $S_{shot}(\mathbf{x}, t)$
  - one for the receiver $R_{shot}(\mathbf{x}, t)$
- Calculate the image from these wave-fields

$$I(\mathbf{x}) = \sum_{t,shot} S_{shot}(\mathbf{x}, t) R_{shot}(\mathbf{x}, t)$$

- Since several 3-D volumes must be keep in GPU memory, each shot is imaged separately

"Reverse-time" Imaging
Two-way propagation

- Estimated GPU performance
  - Measured per wave-field point
  - Image condition adds ½ (1 read & 1 write) = 1
  - Re-use of velocity subtracts ½
  - 5.5 reads-writes/point
  - 2200 - 2700 Mpoints/sec on C1060
- Measured CPU performance
  - ~ 100 Mpoints/sec
- Next step full reverse-time CUDA kernel

Seismic Imaging
Conclusions

- Seismic imaging CUDA codes
  - Two written, verified & in production testing
  - One prototyped
  - Done with about one-man year of effort
  - Kernel speed-ups vary from 5 – 50 X on C1060
  - 128-GPU cluster competes with 4000-CPU cluster

Seismic Imaging
Acknowledgements

- Code authors/collaborators
  - Thomas Cullison (Hess & Colorado School of Mines)
  - Paulius Micikevicius (NVIDIA)
- Hess GPU systems
  - Jeff Davis, Mac McCalla
- NVIDIA support & management
  - Ty Mckercher, Paul Holzhauer, Jeff Saunders, Philip Nenon
- Hess management
  - Jacques Leveille, Vic Forsyth, Jim Sherman