Seismic Imaging on NVIDIA GPUs

Scott Morton
Hess Corporation

Seismic Imaging
Outline

- Seismic data & imaging
- NVIDIA GPUs + CUDA
  - Why?
  - How?
- Three imaging methods
  - Algorithm
  - Challenges
  - Performance

Seismic Imaging

Data

gas

oil

H_2O

Seismic Imaging
Data

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

Seismic Imaging
An iterative process

Computation scales as size of data and image/model

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

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
  - Several years ago there were indicators we can get 10X or more on GPUs
    - Peak performance
    - Benchmarks
    - Simple prototype kernels

Seismic Imaging

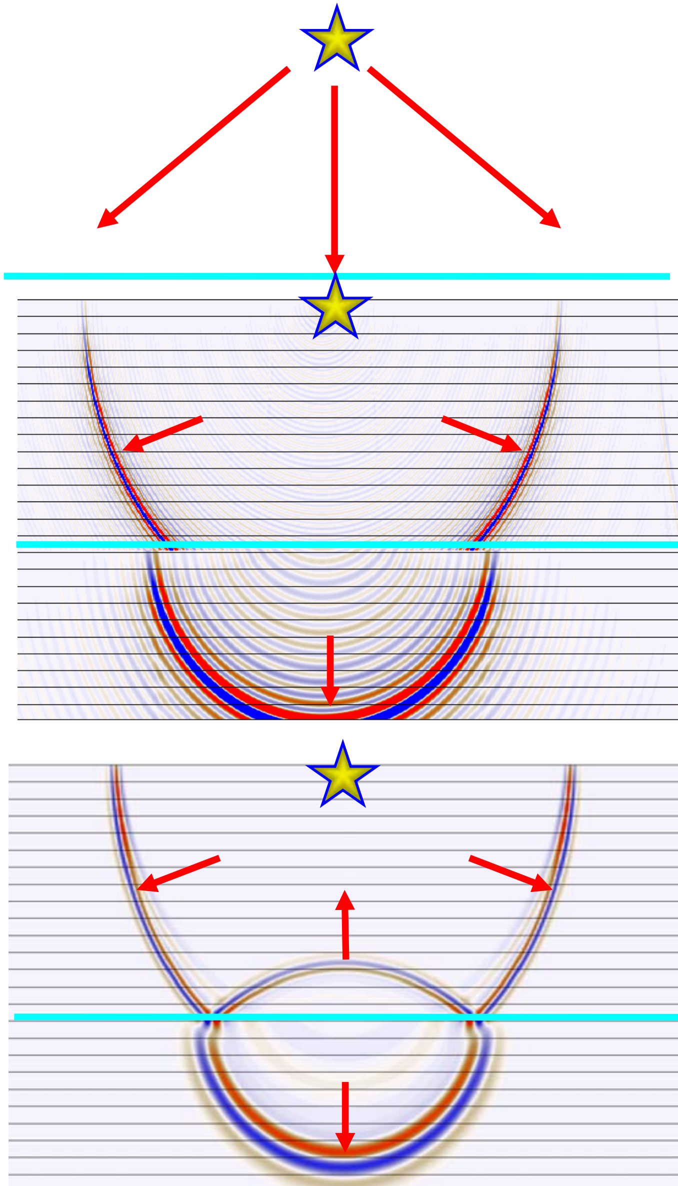
Why CUDA & NVIDIA GPUs?

- Ease of programming
  - Must be able to port, maintain & modify production codes (relatively) easily
    - These costs must be included
  - 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 the biggest challenge

Seismic Imaging
How to port a code?

- **Design GPU algorithm**
  - Optimize for memory hierarchy
  - 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:
    - $N_I \sim 10^9$ is the number of output image points
    - $N_D \sim 10^8$ is the number of input data traces
    - $f \sim 10$ is the number of cycles/point/trace
    - $f\, N_I\, N_D \sim 10^{18}$ cycles ~ 10 CPU-years

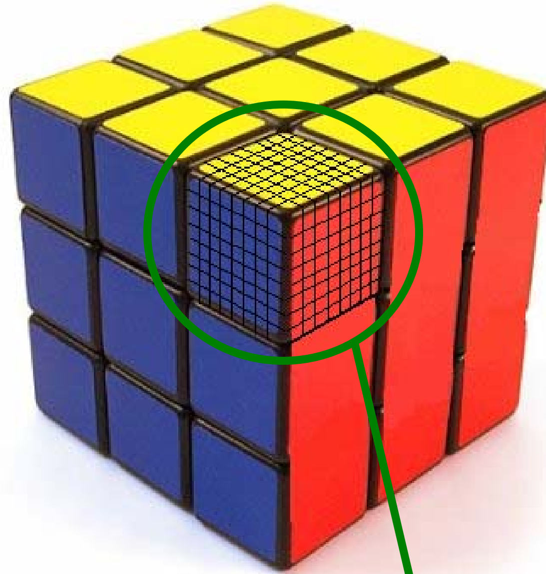
Kirchhoff Imaging

Computational kernel

1 migration contribution

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

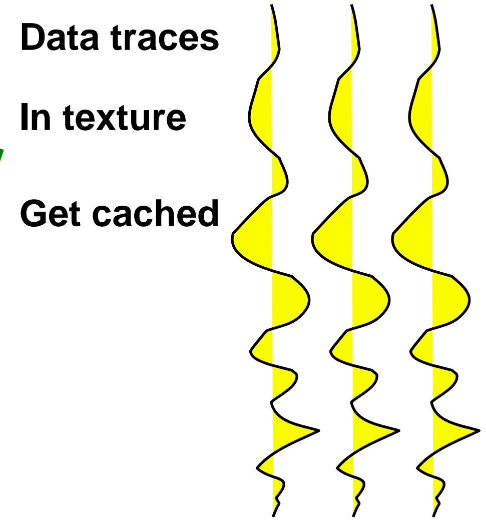
Kirchhoff Imaging
CUDA kernel

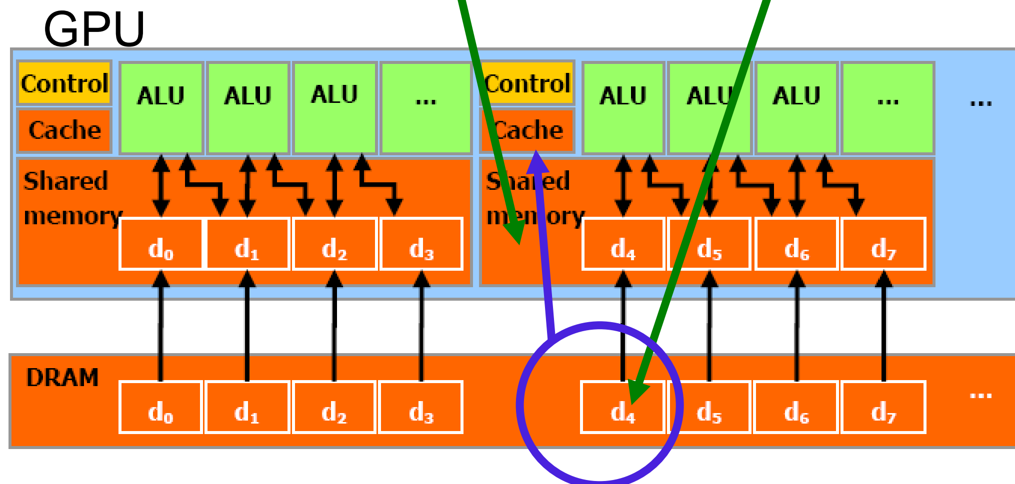


Data traces

In texture

Get cached





Kirchhoff Imaging
Kernel optimization

Performance

Billions of Migration Contributions per Second

10.00
9.00
8.00
7.00
6.00
5.00
4.00
3.00
2.00
1.00
0.00

0 1 2 3 4 5 6 7 8 9

Code Version

GPU
CPU

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

GT200

Typical parameter range

G80

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

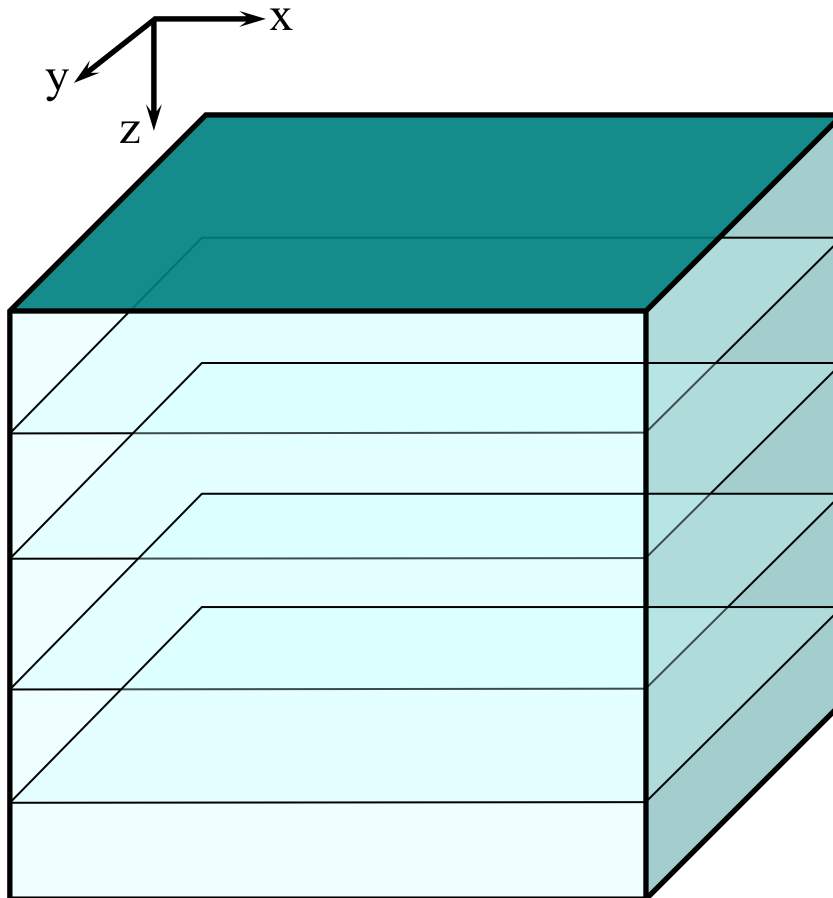
Kirchhoff Imaging
Production status

- GPU kernel incorporated into production code
  - Large kernel speed-ups results in “CPU overhead” for task setup dominating GPU production runs
- Further optimizations
  - create GPU kernels for most “overhead” components
  - optimized left-over CPU code (which helps CPU version also)

Time (hr)	Set-up	Kernel	Total	Speed-up
Original CPU code	5	20	25	
Main GPU kernel	5	0.5	5.5	5
Further optimizations	0.5	0.5	1	25

“Wave-equation” Imaging

One-way propagation



- Based on scalar wave equation

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

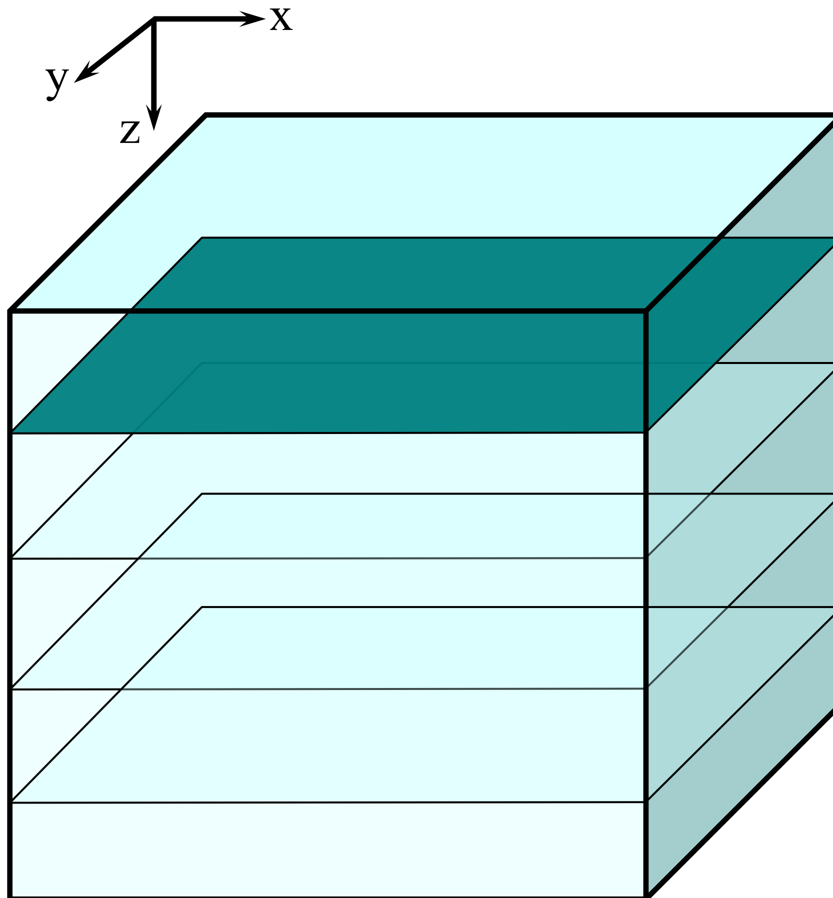
- Frequency-domain
- Preferred direction of propagation: z ~ t

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

  - Evolution in depth

"Wave-equation" Imaging

One-way propagation



- Based on scalar wave equation

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

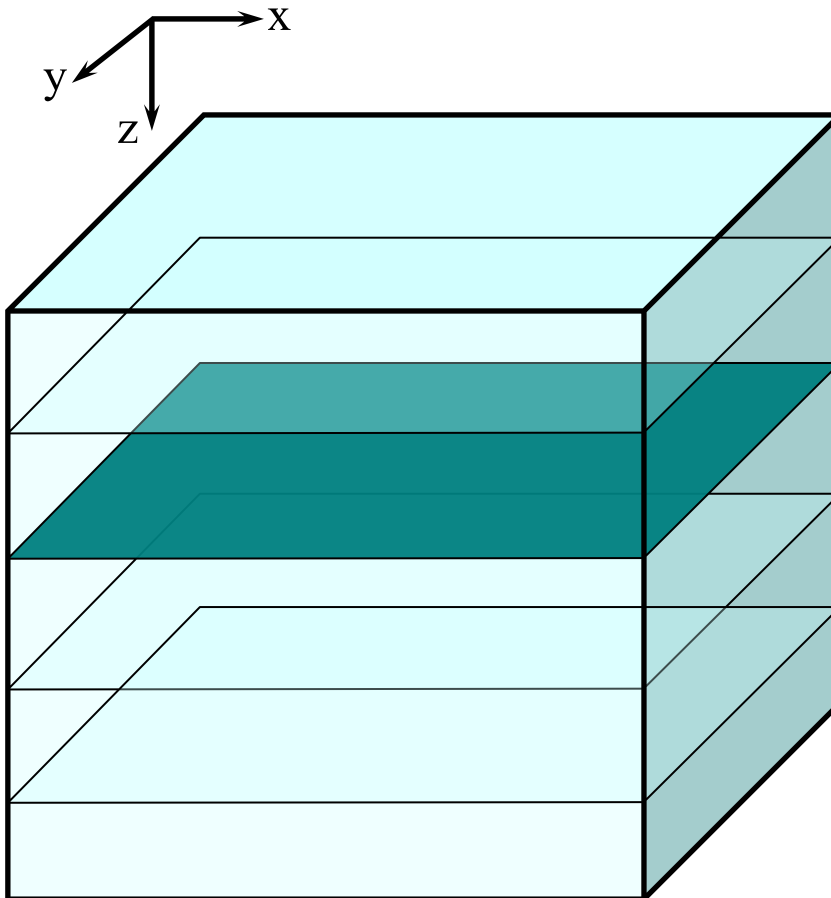
- Frequency-domain
- Preferred direction of propagation: z ~ t

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

  - Evolution in depth

"Wave-equation" Imaging

One-way propagation



- Based on scalar wave equation

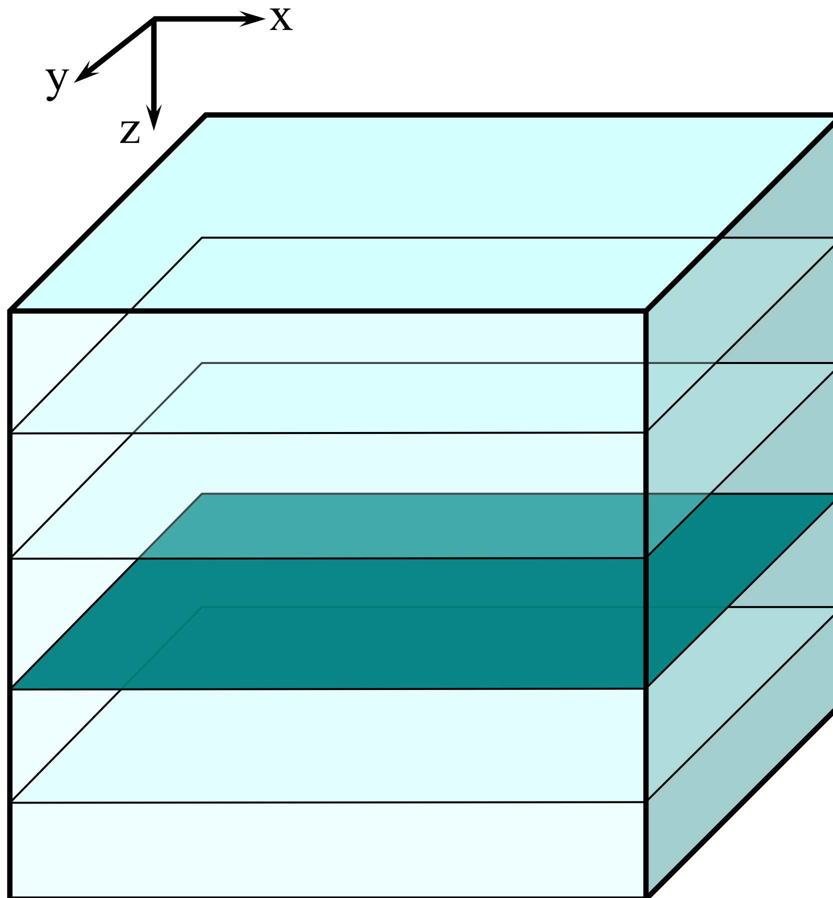
$$\frac{1}{V(\vec{\mathbf{x}})^2}\frac{\partial^2 P}{\partial t^2} = \nabla^2 P$$

- 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

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

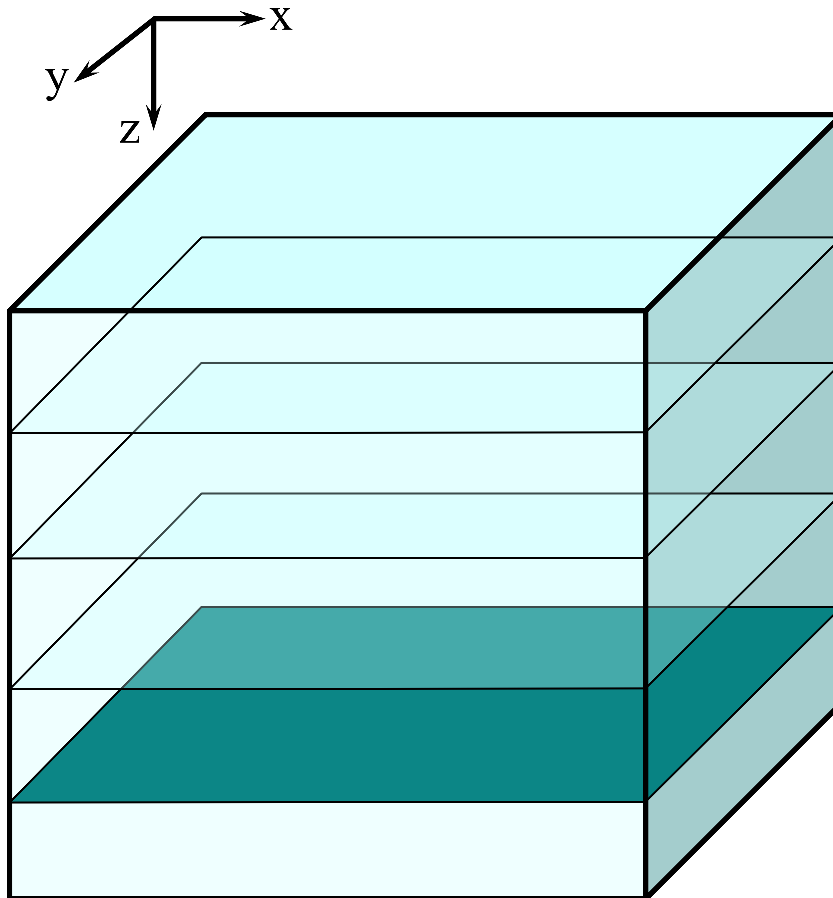
- Frequency-domain
- Preferred direction of propagation: z ~ t

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

- Evolution in depth

“Wave-equation” Imaging

One-way propagation



- Based on scalar wave equation

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

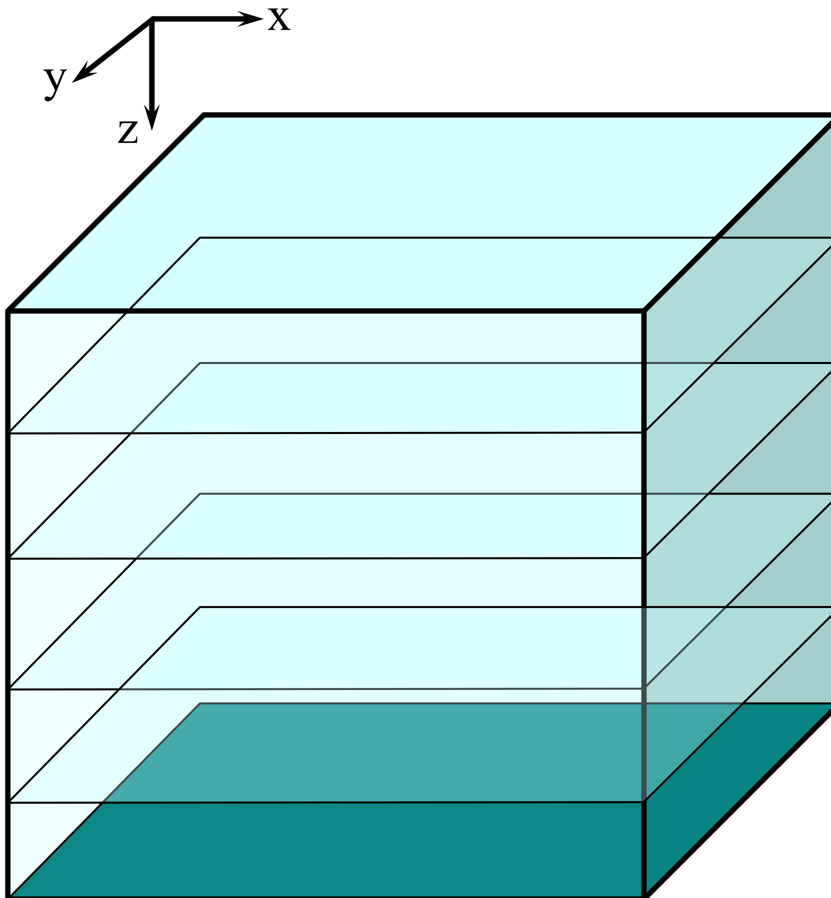
- Frequency-domain
- Preferred direction of propagation: z ~ t

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

- Evolution in depth

"Wave-equation" Imaging

One-way propagation



- Based on scalar wave equation

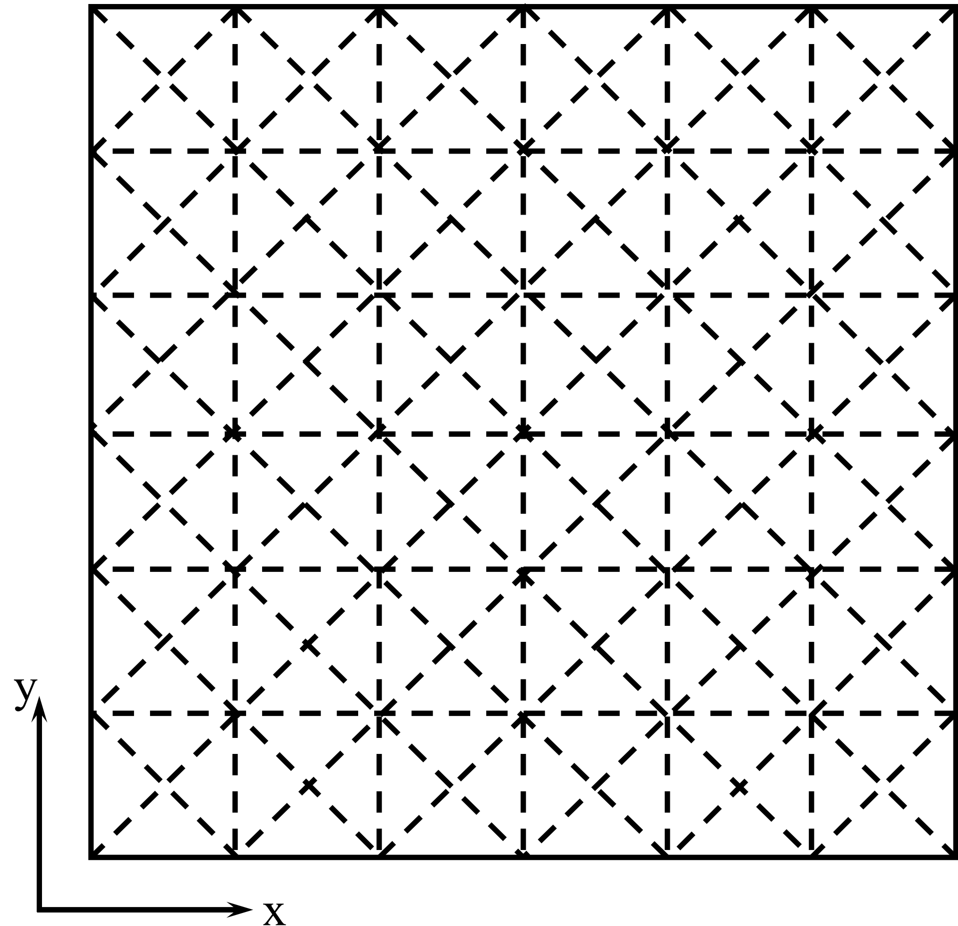
$$\frac{1}{V(\vec{\mathbf{x}})^2}\frac{\partial^2 P}{\partial t^2} = \nabla^2 P$$

- 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

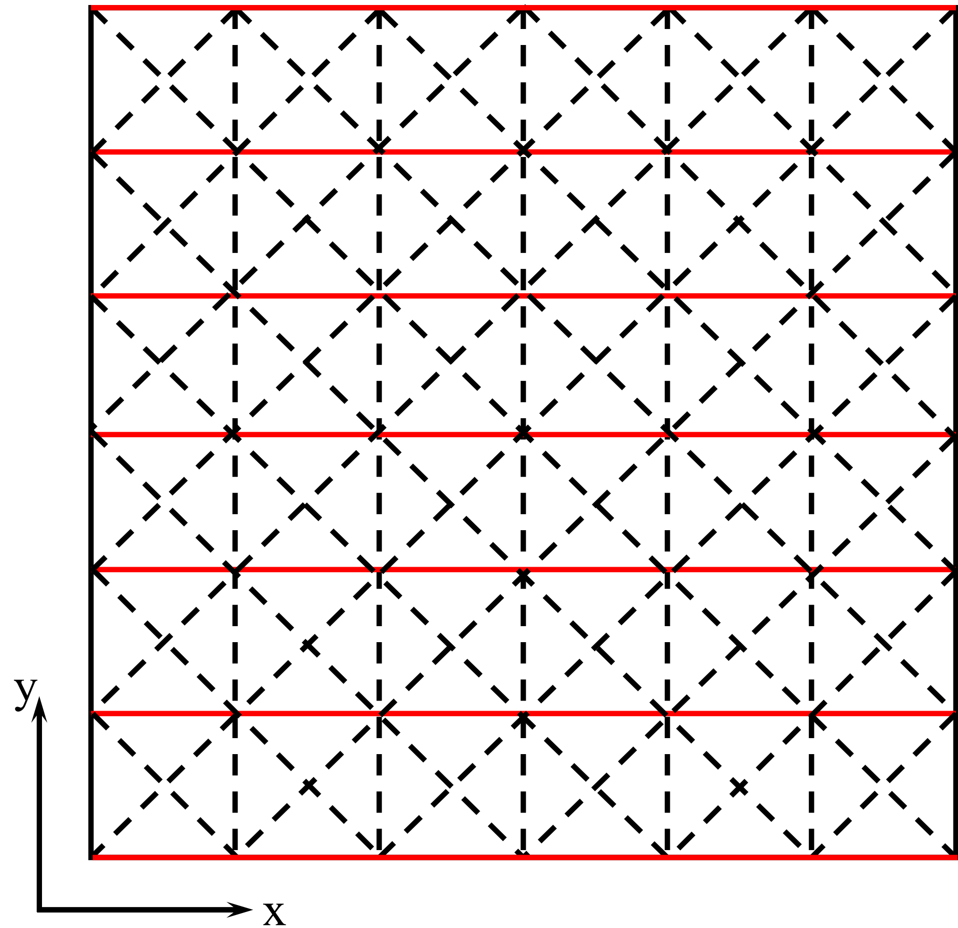
"Wave-equation" Imaging
One-way propagation



- 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

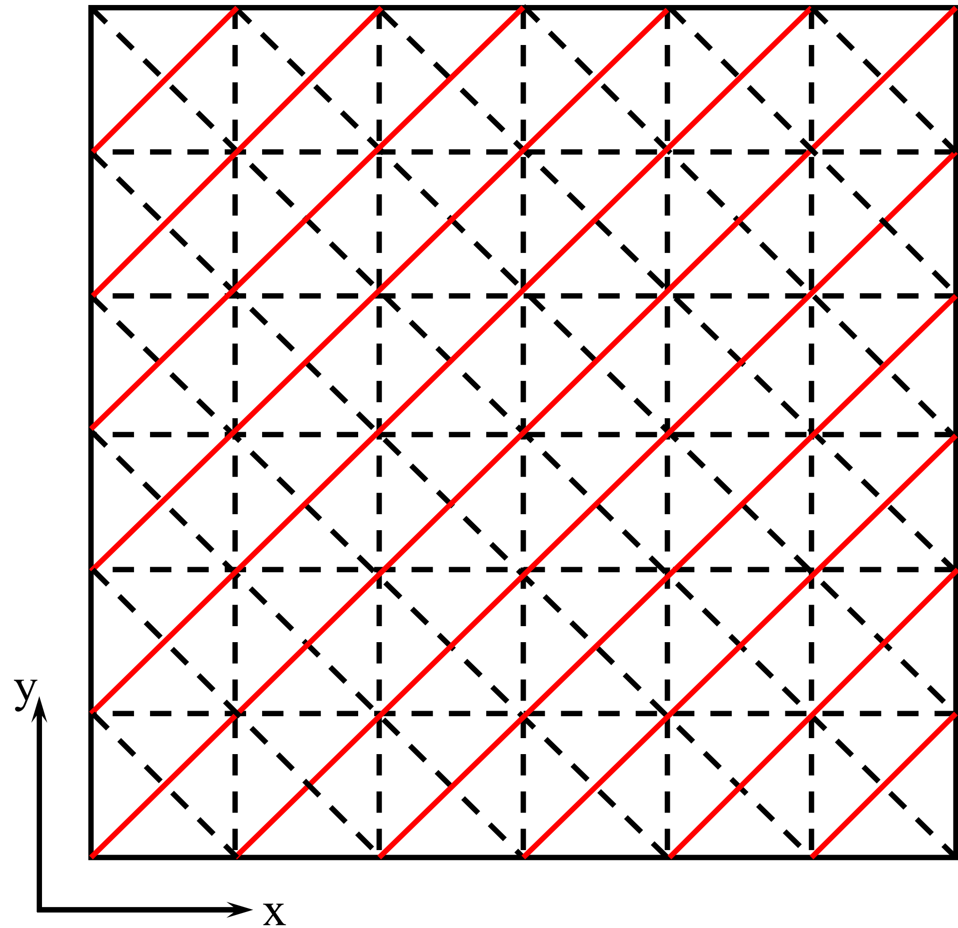
"Wave-equation" Imaging

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

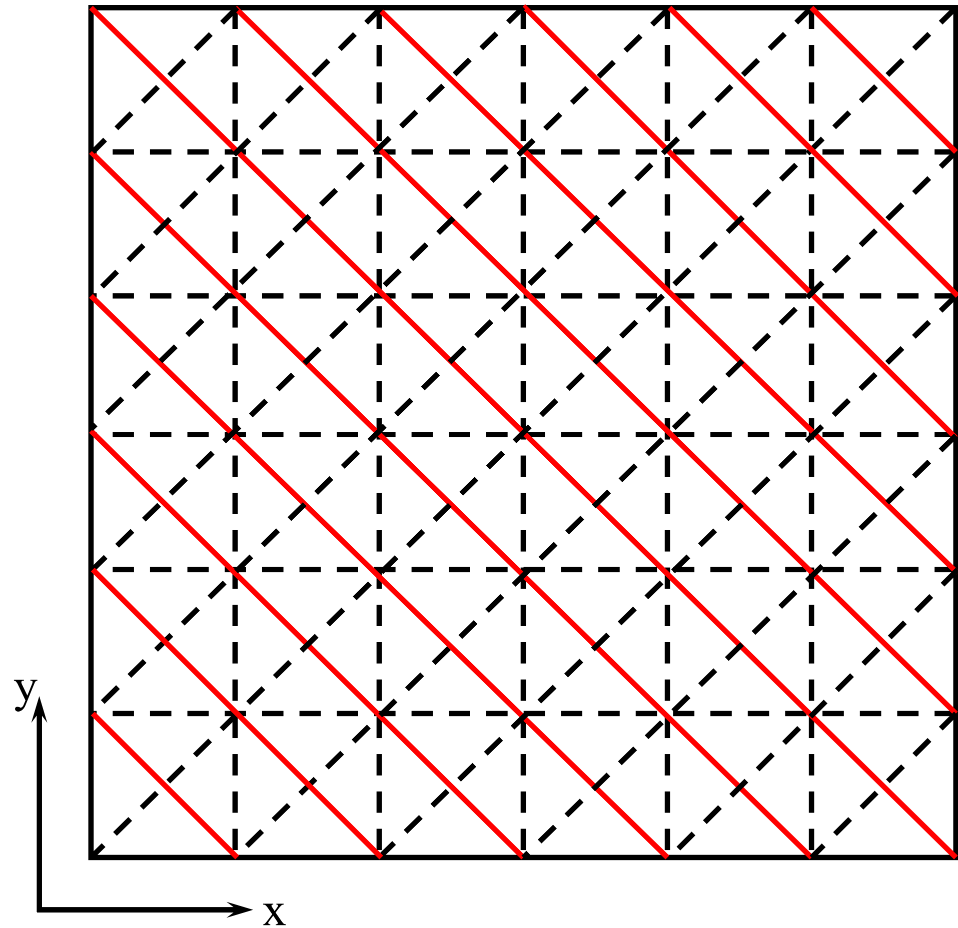
One-way propagation



- 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

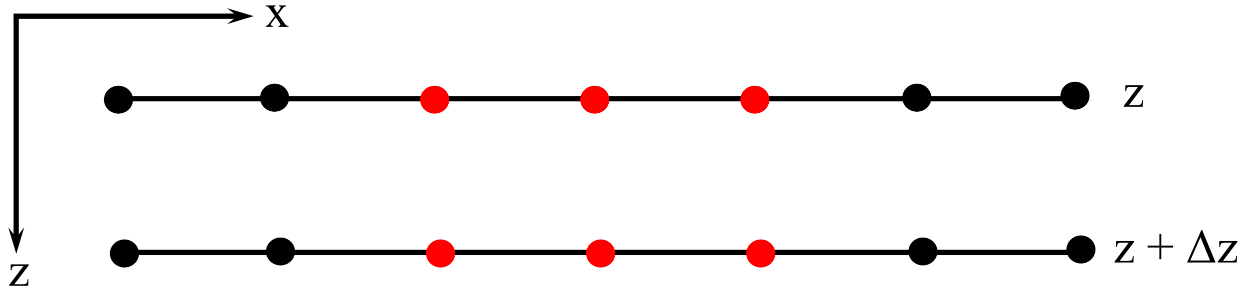
One-way propagation



- 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

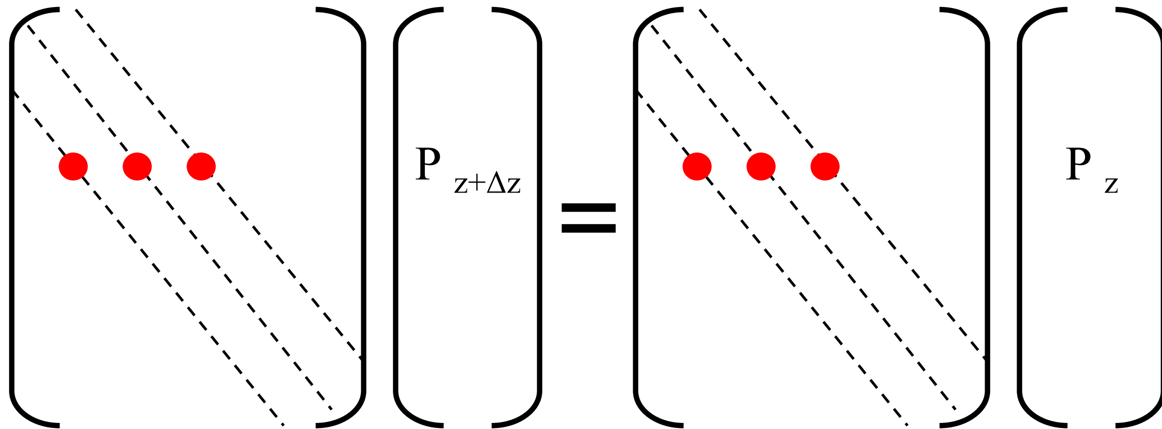
"Wave-equation" Imaging

Implicit complex tri-diagonal linear systems



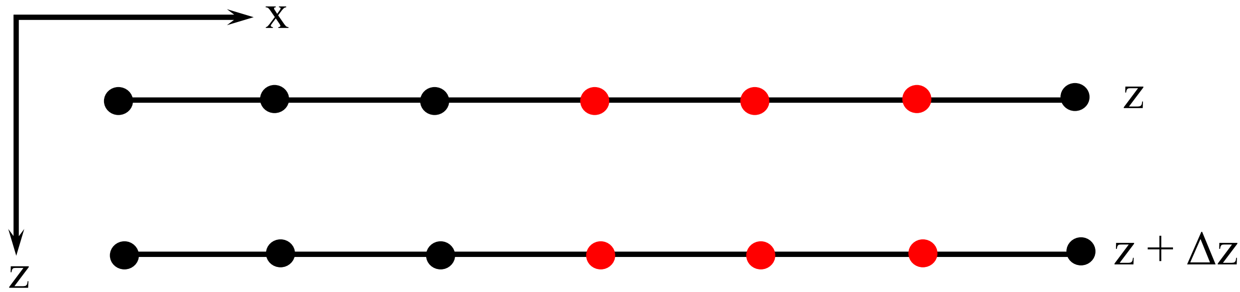
$$A P_{x-\Delta x,y}^{z+\Delta z} + (1-2A) P_{x,y}^{z+\Delta z} + A P_{x+\Delta x,y}^{z+\Delta z} =$$

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



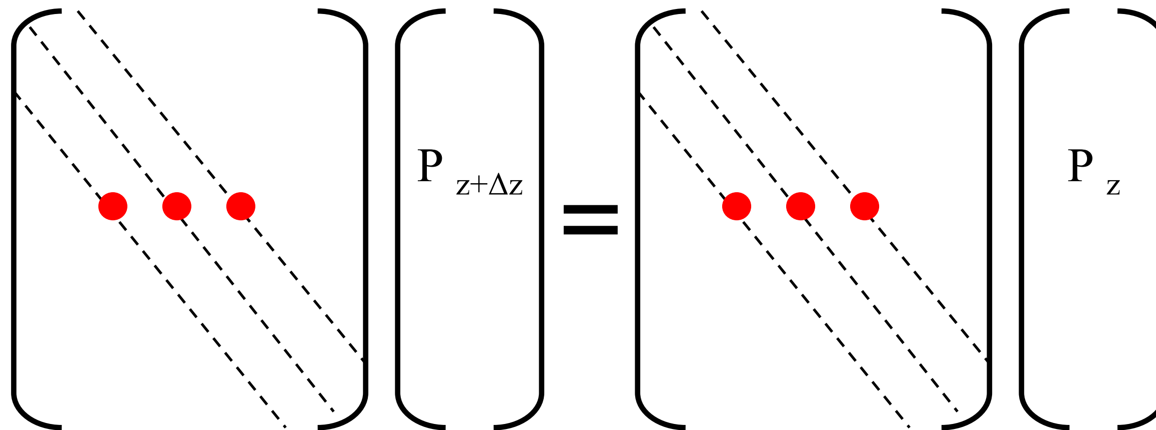
"Wave-equation" Imaging

Implicit complex tri-diagonal linear systems



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

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

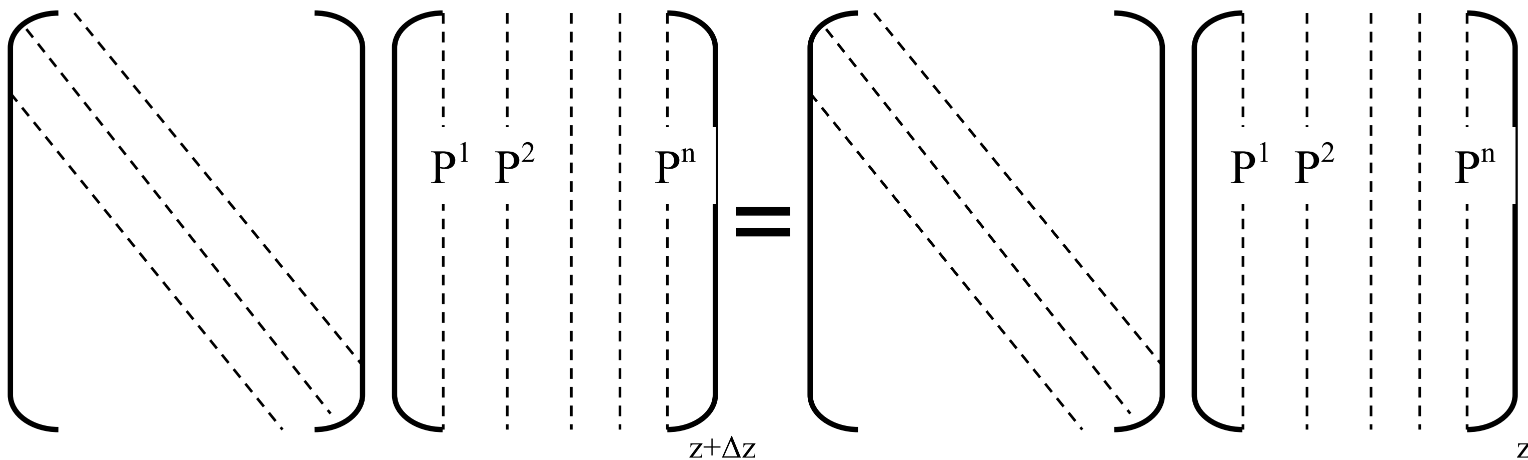


The evaluation and solution of these complex tri-diagonal systems dominates the computational cost of our wave-equation imaging code.

"Wave-equation" Imaging

Low level parallelism

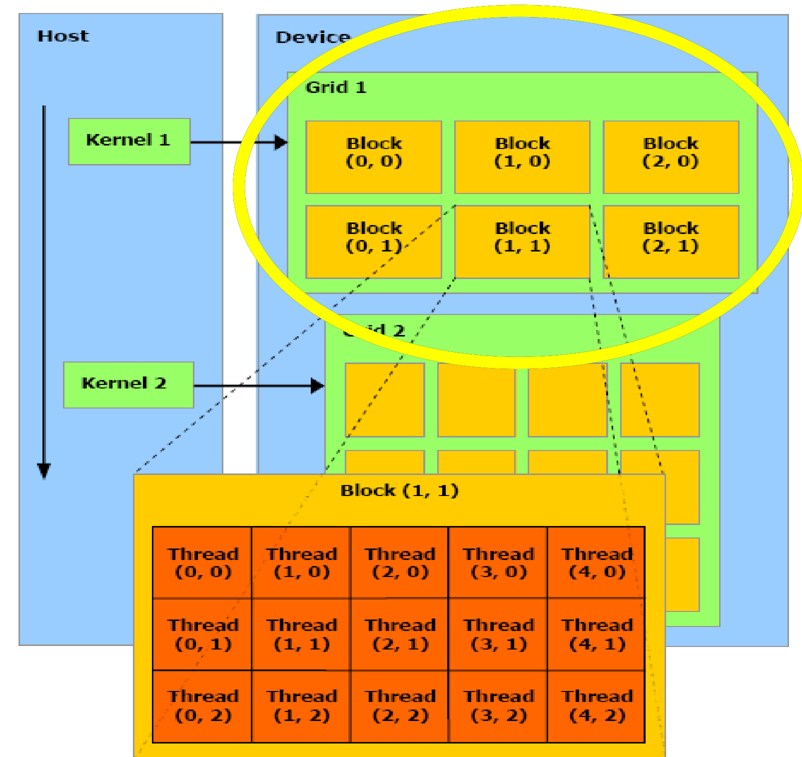
- **Common work between shot-records**
  - Calculating the coefficients of the matrices
    - Dependent on frequency & local velocity
  - Part of the solving of the tri-diagonal system
- **Parallelize over shot-records in the kernel**



"Wave-equation" Imaging

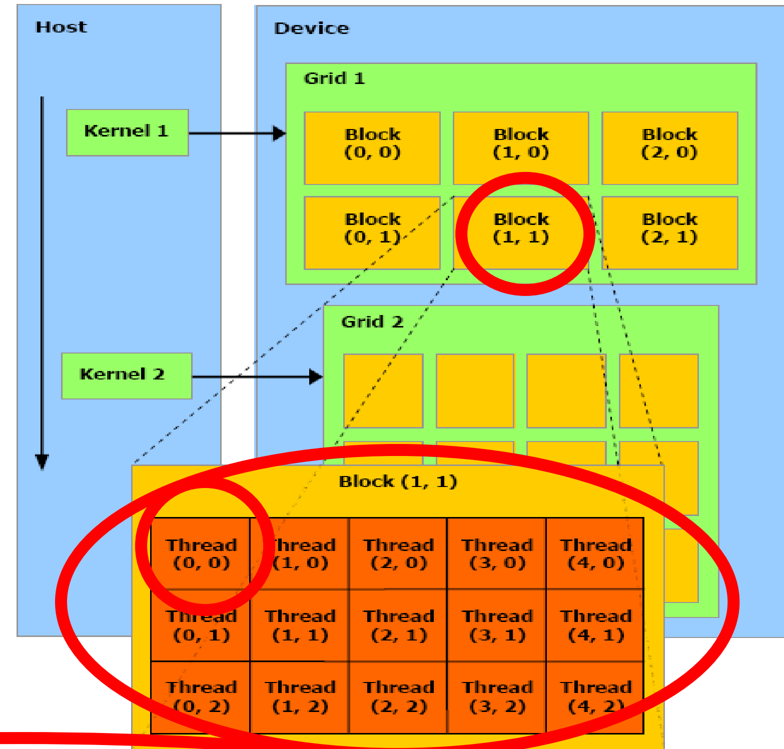
CUDA kernels

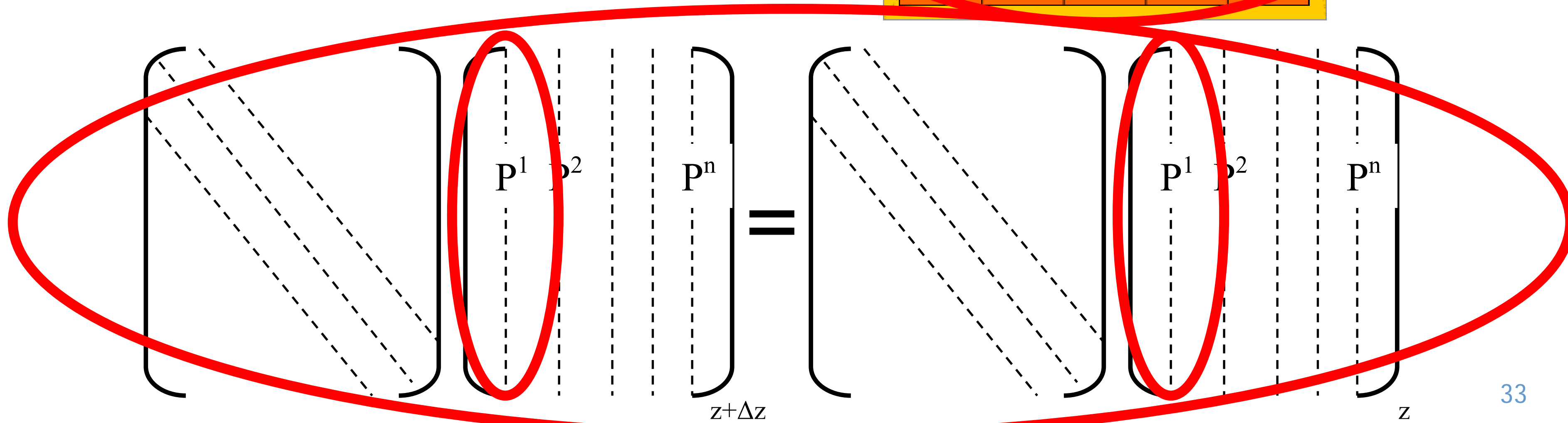
- Separate kernels for each operator
  - x, y, x+y and x-y



"Wave-equation" Imaging
CUDA kernels

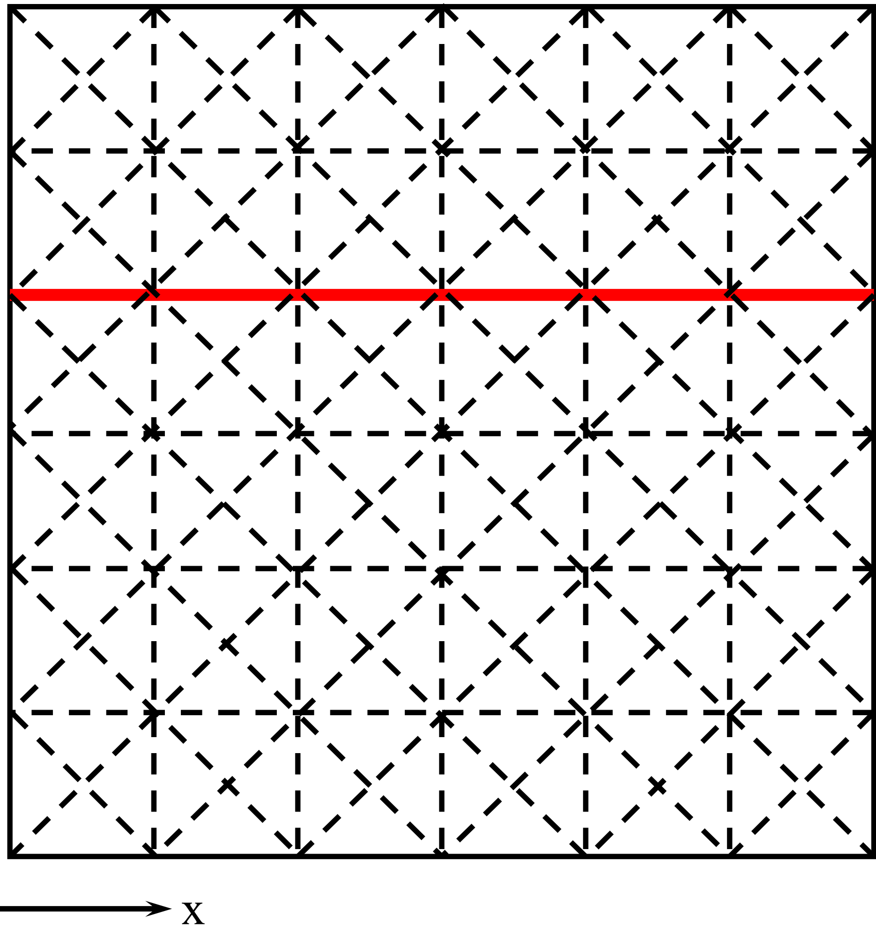
- Separate kernels for each operator
  - x, y, x+y and x-y

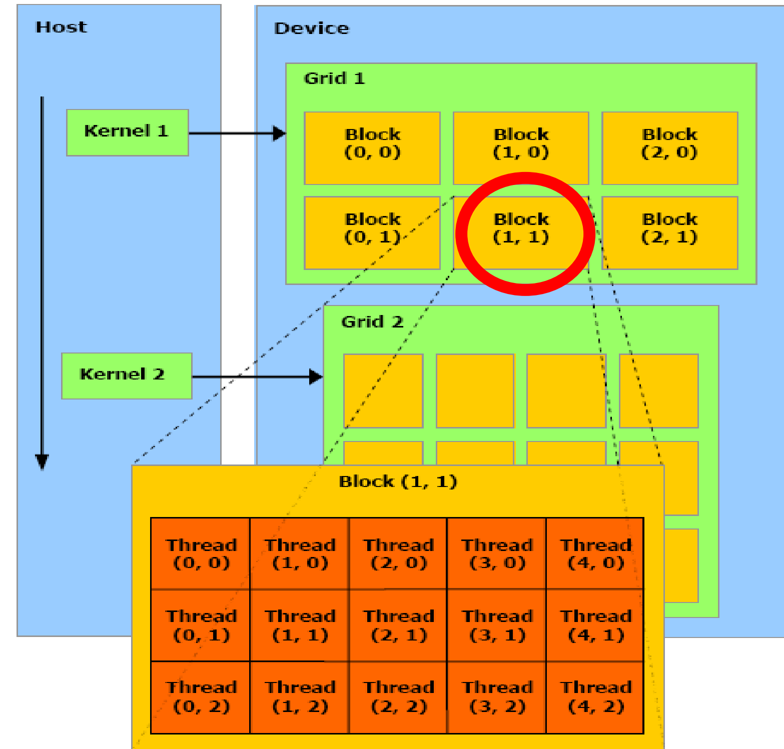




"Wave-equation" Imaging

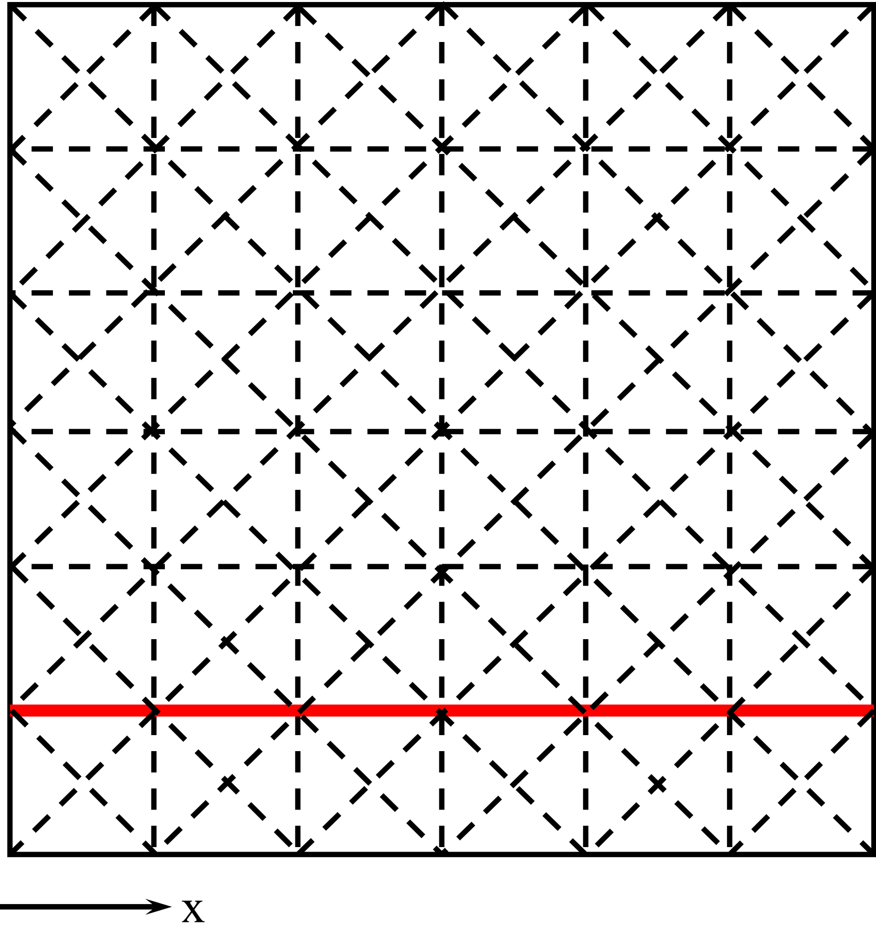
CUDA kernels

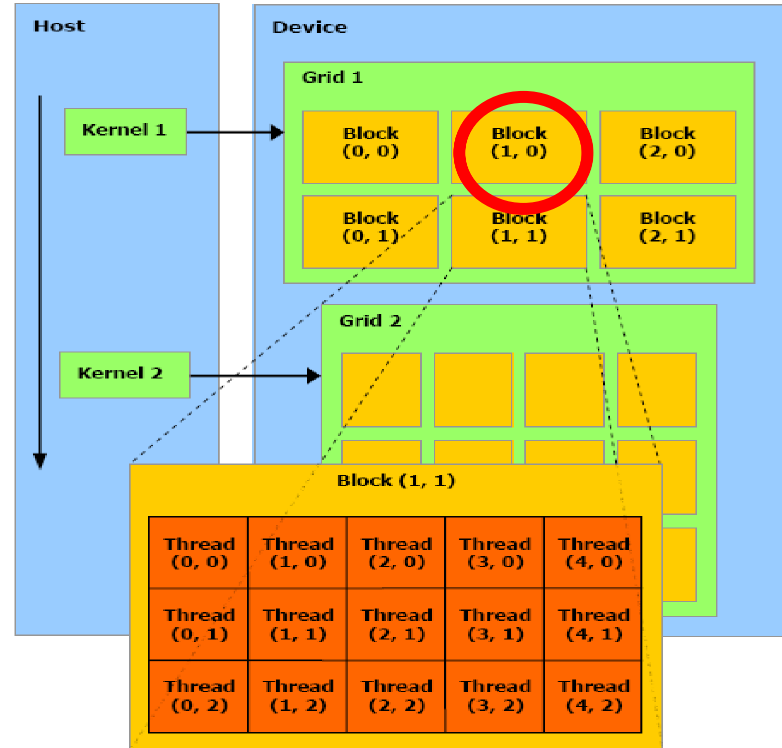




"Wave-equation" Imaging

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
- Production CUDA kernels
  - Single kernel handles x, y, x+y & x-y operators
  - Several kernels calculate coefficients
  - Performance: 300-500 Mpoints/sec

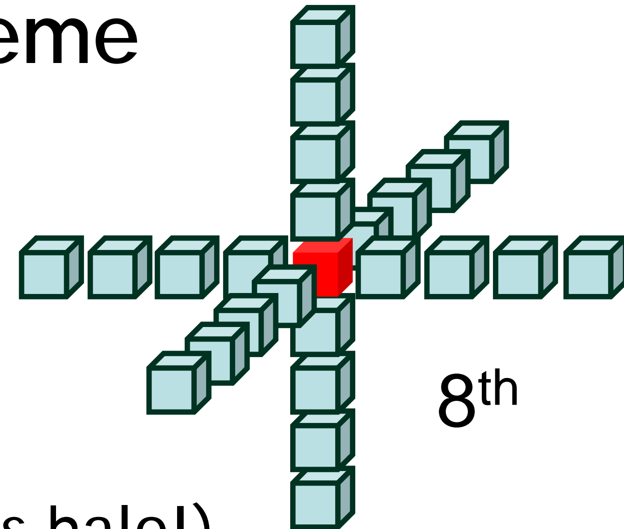
"Reverse-time" Imaging

Two-way propagation

- Based on the scalar wave equation

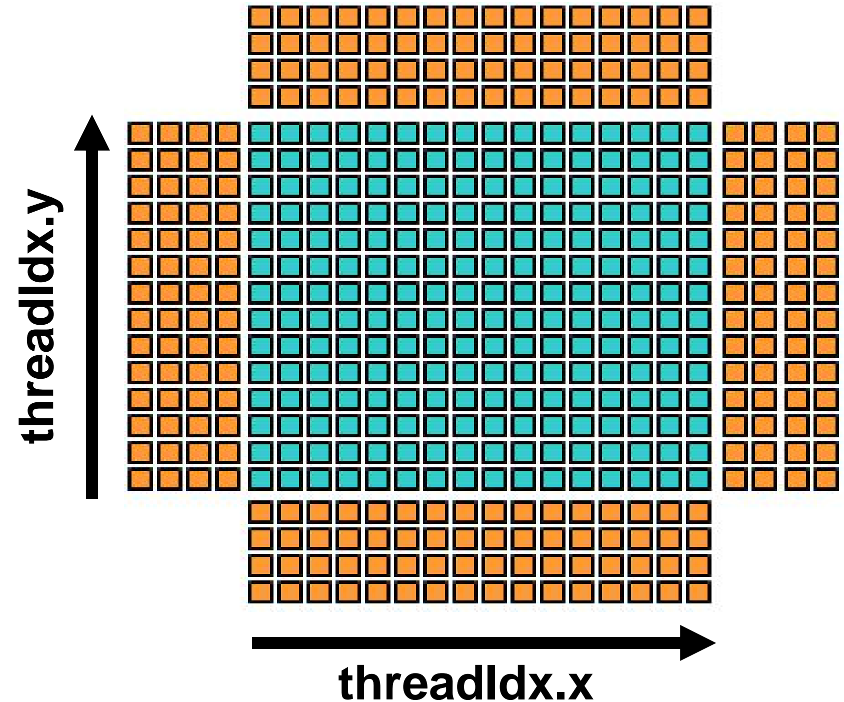
$$\frac{1}{V(\mathbf{x})^2}\frac{\partial^2 P}{\partial t^2} = \nabla^2 P$$

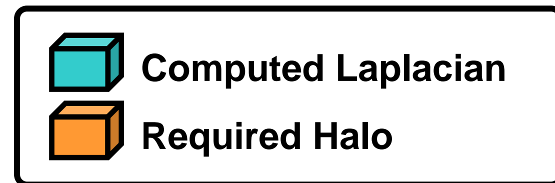
- Explicit finite-difference scheme
  - 2nd order in time
  - Variable order in space: 6th - 16th
    - Most of the computation
  - Bandwidth
    - Read *P(**x**,t)*, *P(**x**,t-dt)* & *V(**x**)* (plus halo!)
    - Write *P(**x**,t+dt)*
    - Max performance is 4+ Gpt/s



"Reverse-time" Imaging
CUDA algorithm

- Paulius's algorithm
  - Each thread specifies an (x,y) point, marching in z.
  - Each thread block handles a 2-D rectangle.
  - Each 2-D slice + halo is read into shared memory.
  - Threads in a block re-use these values.

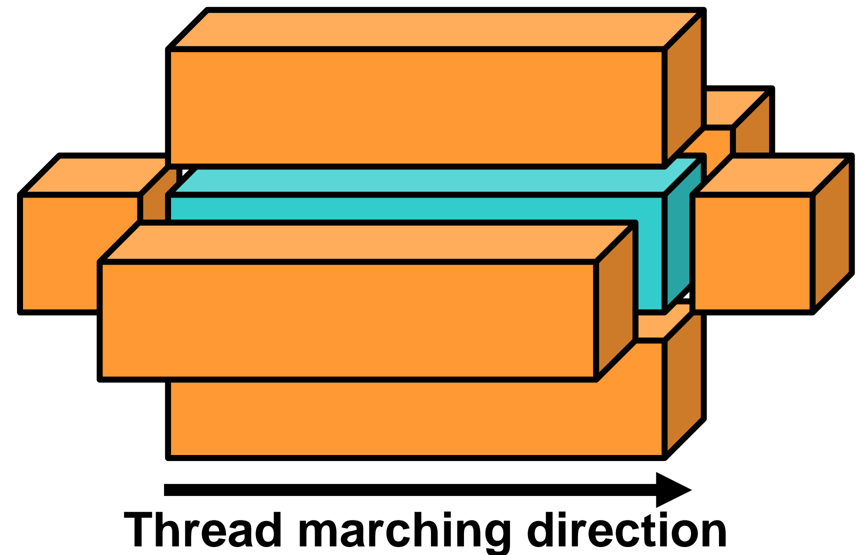




"Reverse-time" Imaging

CUDA algorithm

- Paulius's algorithm
  - Threads in a block march in z, storing values "in-front" & "behind" in registers.
  - Number of registers limits the block size and the core-to-halo ratio.
  - GPU performance is predictable: 2.5 – 3 Gpt/s.



"Reverse-time" Imaging

Kernel performance

8th order, RTM

Speedup

30
25
20
15
10
5
0

128x128x200
256x96x200
96x256x200
256x256x200
480x480x200
512x512x512
640x640x200
640x640x8
960x704x200
704x960x200

Grid size

const, 16x16
shared, 16x16
const, 16x32
shared, 16x32

"Reverse-time" Imaging
Inter-GPU communication

- High frequency requires
  - Dense sampling
  - Large memory
  - Multiple GPUs
  - Halo exchange
  - Inter-GPU communication
- Device ⇔Host
  - Use pinned memory
  - PCIe bus predictably yields ~ 5 GB/s
  - ~ 10 % of kernel time
  - Easily hidden

"Reverse-time" Imaging

Inter-GPU communication

- CPU process ⇔process
  - Currently using MPI
    - From legacy code
  - Performance variable
  - Comparable to kernel time
  - Solutions
    - OpenMP?
    - single controlling process?
- Node ⇔node
  - Currently Gigabit Ethernet
  - Solution? Infiniband? 10-GigE?

Seismic Imaging
Summary

- Seismic imaging CUDA codes
  - All 3 main codes written & verified
    - Two in production
    - One in production testing/optimization
  - All done with about two-man years of effort
  - Kernel speed-ups vary from 10 - 80 X on GT200
  - 456-GPU cluster out-performs 3000-CPU cluster
- GPU cluster
  - Jan 2008: bought 32-nodes (128 G80 GPUs)
  - Dec 2008: upgraded & expanded to 456 GT200s
  - Nov 2009: expanding to 1200 GT200s

Seismic Imaging
Summary

GPU adoption

Wave-eqn migration

Purchase

Kirchhoff migration

Reverse-time migration

time

Seismic Imaging
Acknowledgements

- **Code co-authors/collaborators**
  - Thomas Cullison (Hess & Colorado School of Mines)
  - Paulius Micikevicius (NVIDIA)
  - Igor Terentyev (Hess & Rice University)
- **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