Reverse time migration and subsurface offset gathers computing on GPUs

Abdullah AlTheyab and Robert Clapp
SEP 138

GPU + CUDA = faster graduation, hopefully!

- Unlike many contemporary hardware accelerators, graphics processing unit (GPU) is easy to program.

- Compute unified device architecture (CUDA) is a C programming language extension for programming GPUs.

- Few weeks of learning is enough to master GPU programming.

Outline

- Governing equations
- Motivation for using GPUs
- GPU programming
- Wave propagation
- Reverse time migration (RTM)
- Offset domain common image gathers generation
- Synthetic and field data examples

Governing equations

- Wave propagation

$$\left(\nabla^2 - \frac{1}{v^2(\mathbf{x})}\frac{\partial^2}{\partial t^2}\right) P(\mathbf{x},t) \quad = \quad 0$$

- Upper boundary condition

$$\begin{aligned} P_f(\mathbf{x}_s,t) &= \int_0^t f_s(t')dt' \\ P_b(\mathbf{x}_{r_j},t) &= D_s(r_j,t) \end{aligned}$$

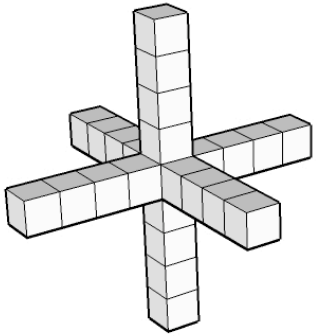
- Imaging condition

$$I(\mathbf{x},\mathbf{h}) = \sum_s \sum_t P_f(\mathbf{x}+\mathbf{h},t;s) P_b(\mathbf{x}-\mathbf{h},t;s)$$

s source	t time	P_f forward propagated wavefield
r receiver	v velocity	P_b backward propagated wavefield
$\mathbf{x}$ position	$\mathbf{h}$ subsurface offset	
D_s shot gather	f_s source signature	

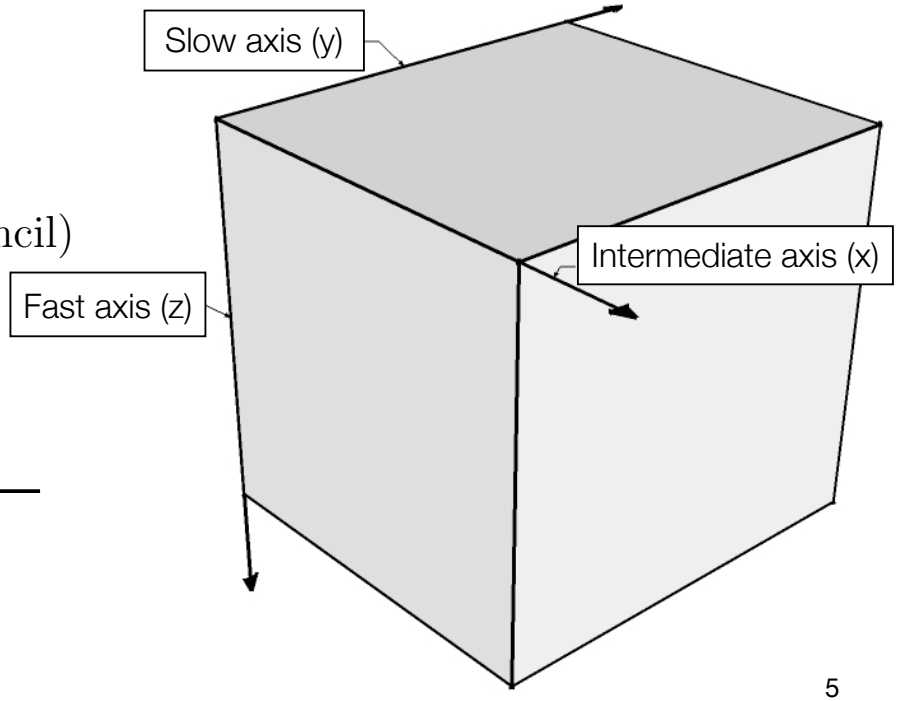
Wave propagation algorithm

- Time-domain finite-difference (FDTD)
  - 2nd order in time
  - 8th order in space
- FDTD Algorithm:



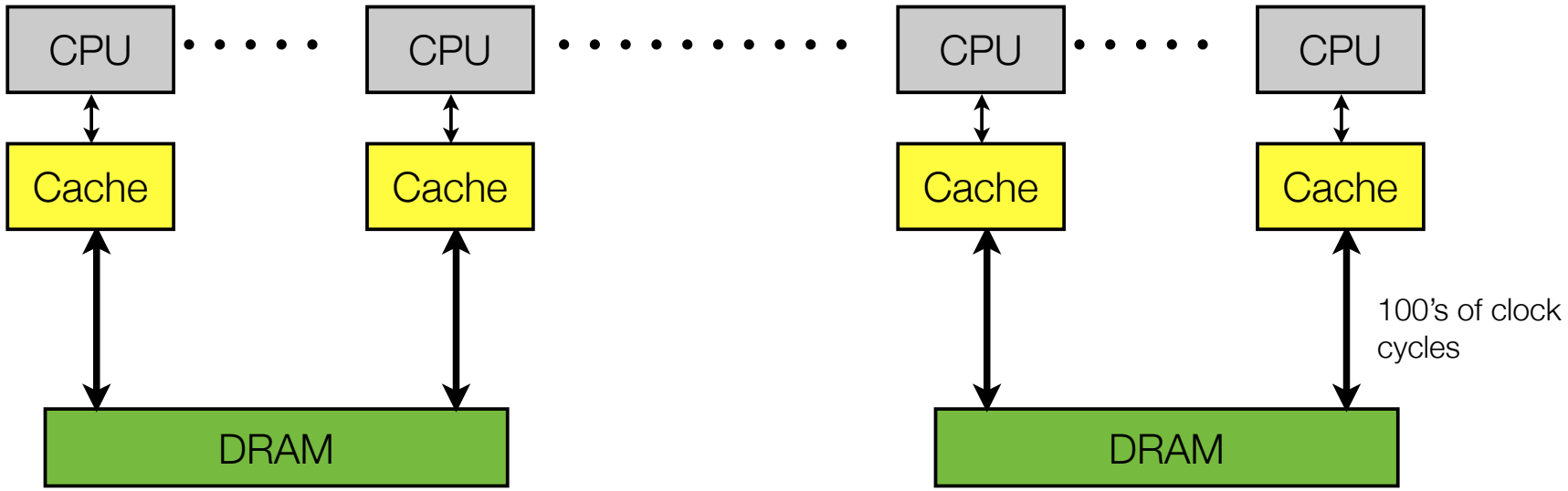
star-shaped spatial stencil

for $t = 1$ to $t_{\max}$ **do**
  set boundary condition
  for $y = 1$ to $y_{\max}$ **do**
    for $x = 1$ to $x_{\max}$ **do**
      for $z = 1$ to $z_{\max}$ **do**
        $P^{t+1}_{x,y,z} \leftarrow \text{extrapolate}(P^t, P^{t-1}, v, \text{stencil})$
      end for
    end for
  end for
end for



CPU processing and memory latency

- Clock frequency of CPUs is much higher than that of DRAM.
- Cache oblivious algorithms exploit spatial and temporal locality to minimize latency.
- The larger the number of processing units is the higher the cost of communication.

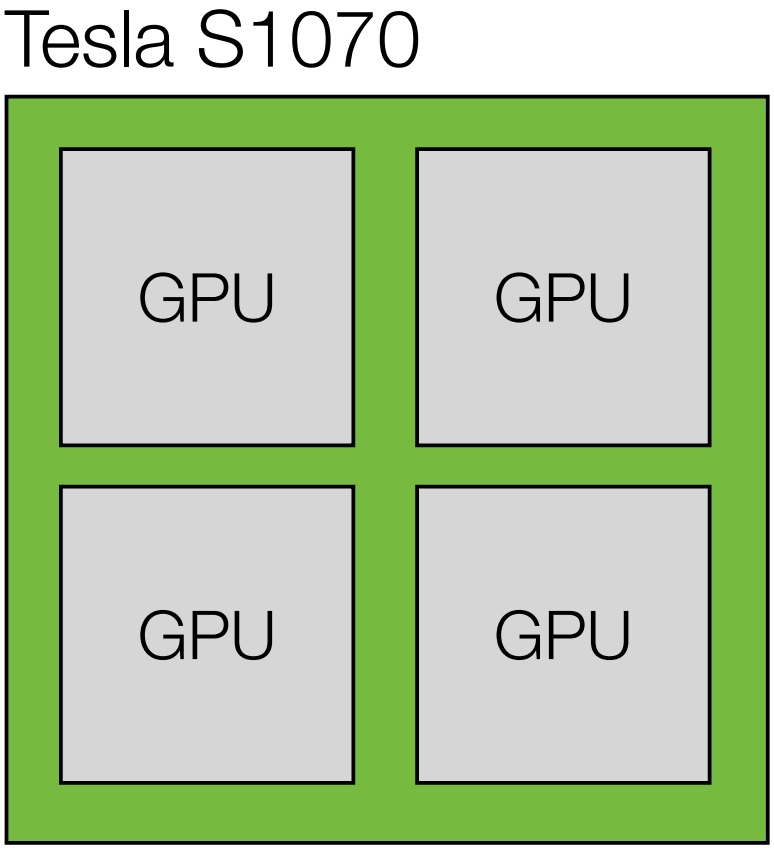


Streaming computing architectures

- Programming Field programable gate arrays (FPGAs) is not trivial.
- GPUs are much easier to program with CUDA.
- CUDA is one of many GPGPU programming tools among which is OpenCL and Brook.

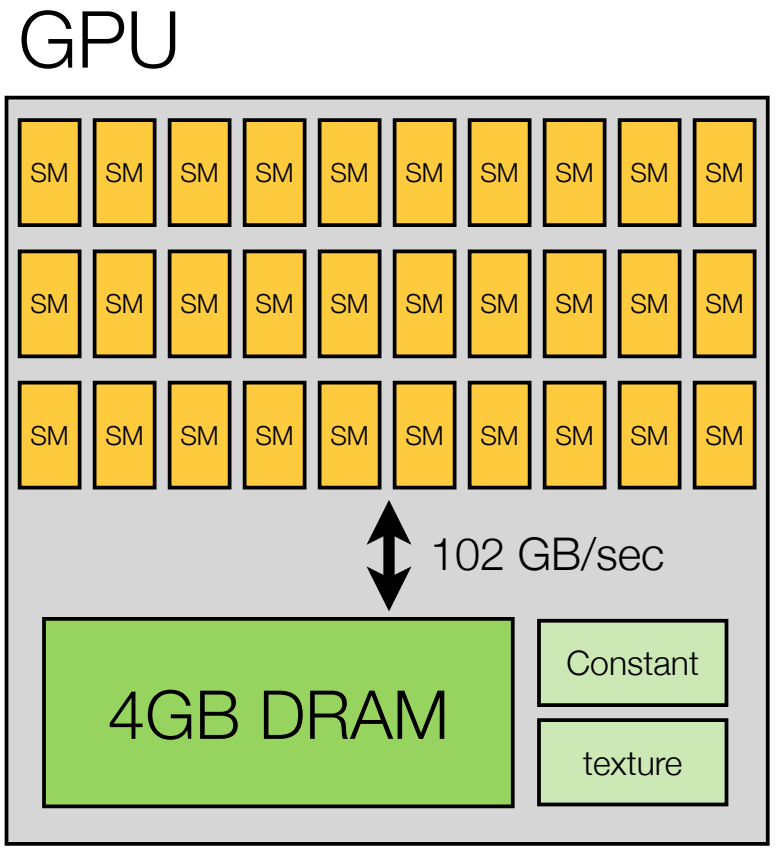
GPU programming

- 4 Tesla S1070 GPUs:



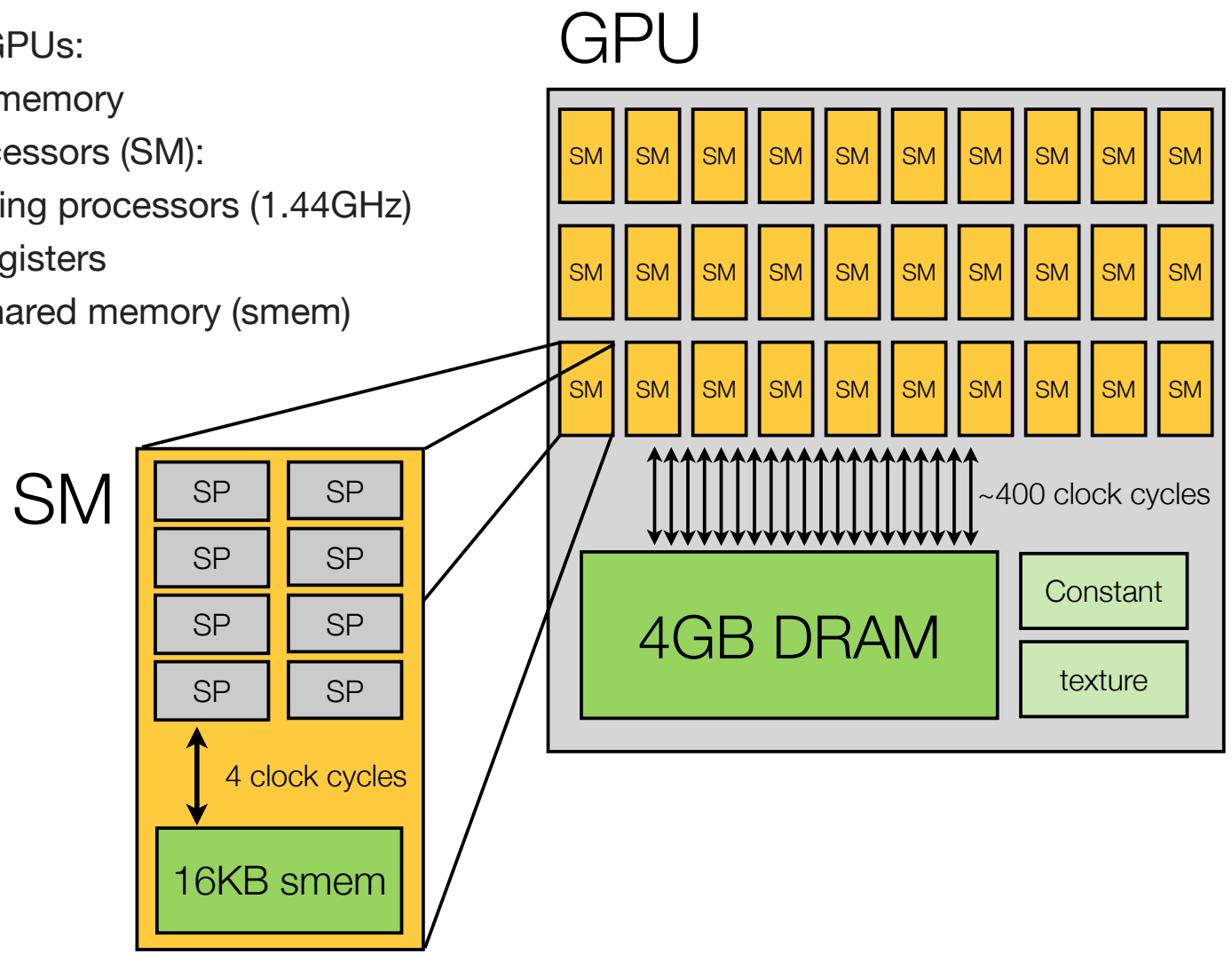
GPU programming

- 4 Tesla S1070 GPUs:
  - 4GB global memory
  - 30 Multiprocessors (SM):



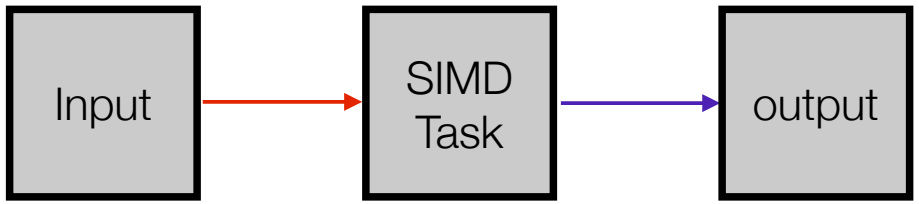
GPU programming

- 4 Tesla S1070 GPUs:
  - 4GB global memory
  - 30 Multiprocessors (SM):
    - 8 streaming processors (1.44GHz)
    - 16 KB registers
    - 16 KB shared memory (smem)

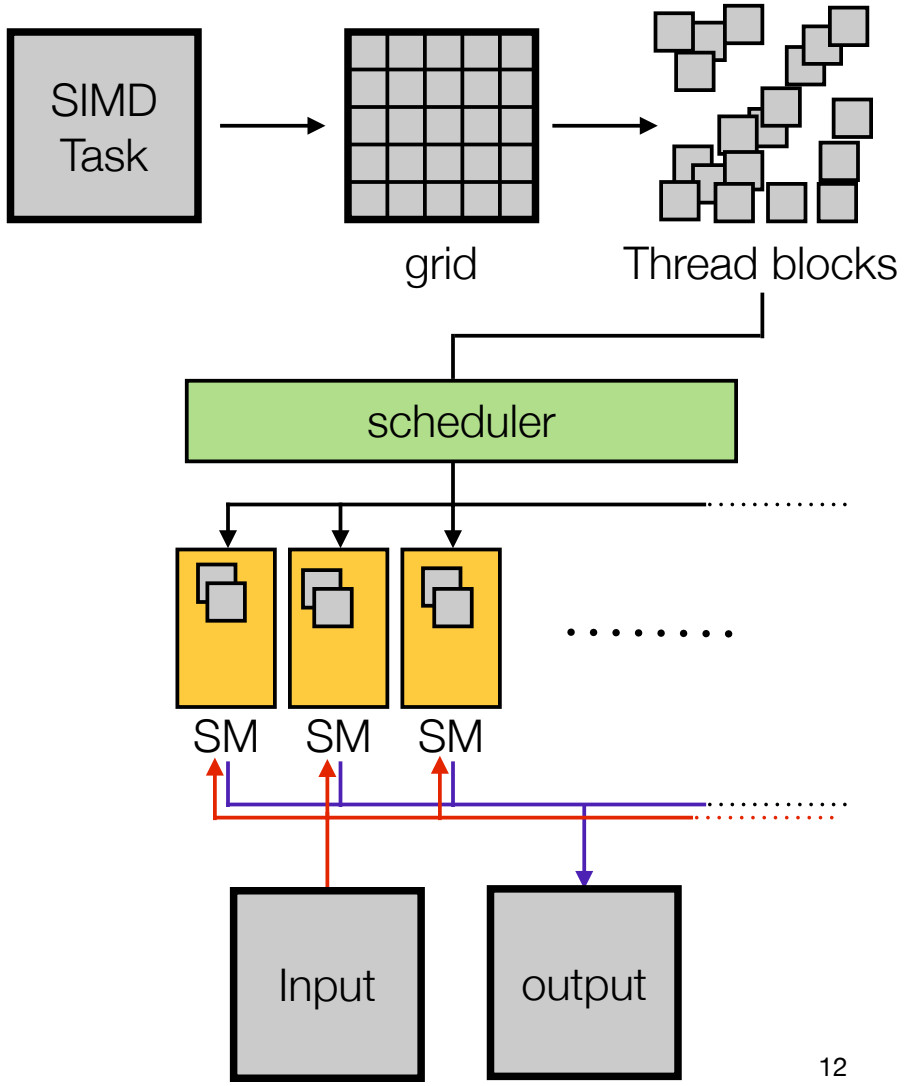


GPU programming

- GPUs are best suited for algorithms that have single instruction multiple data (SIMD) execution model.
- Memory latency is hidden by executing threads while other are waiting for memory access.

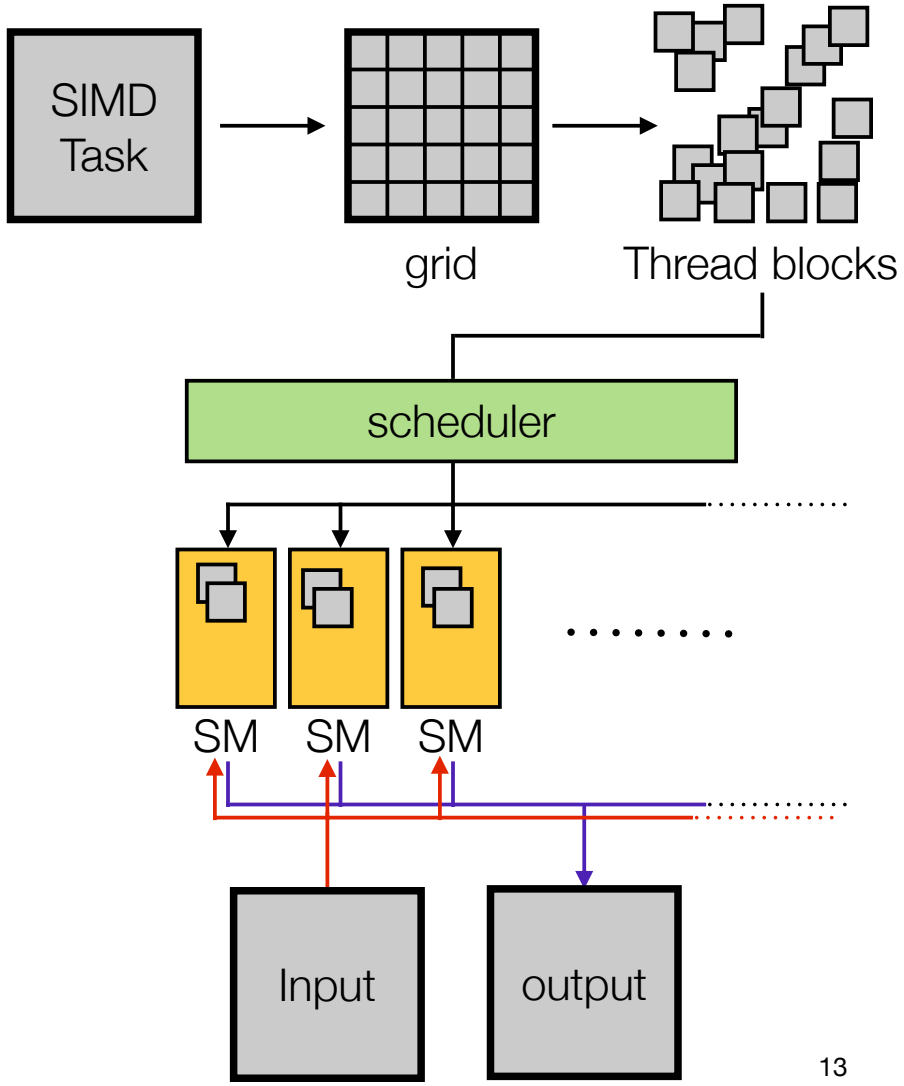


GPU programming



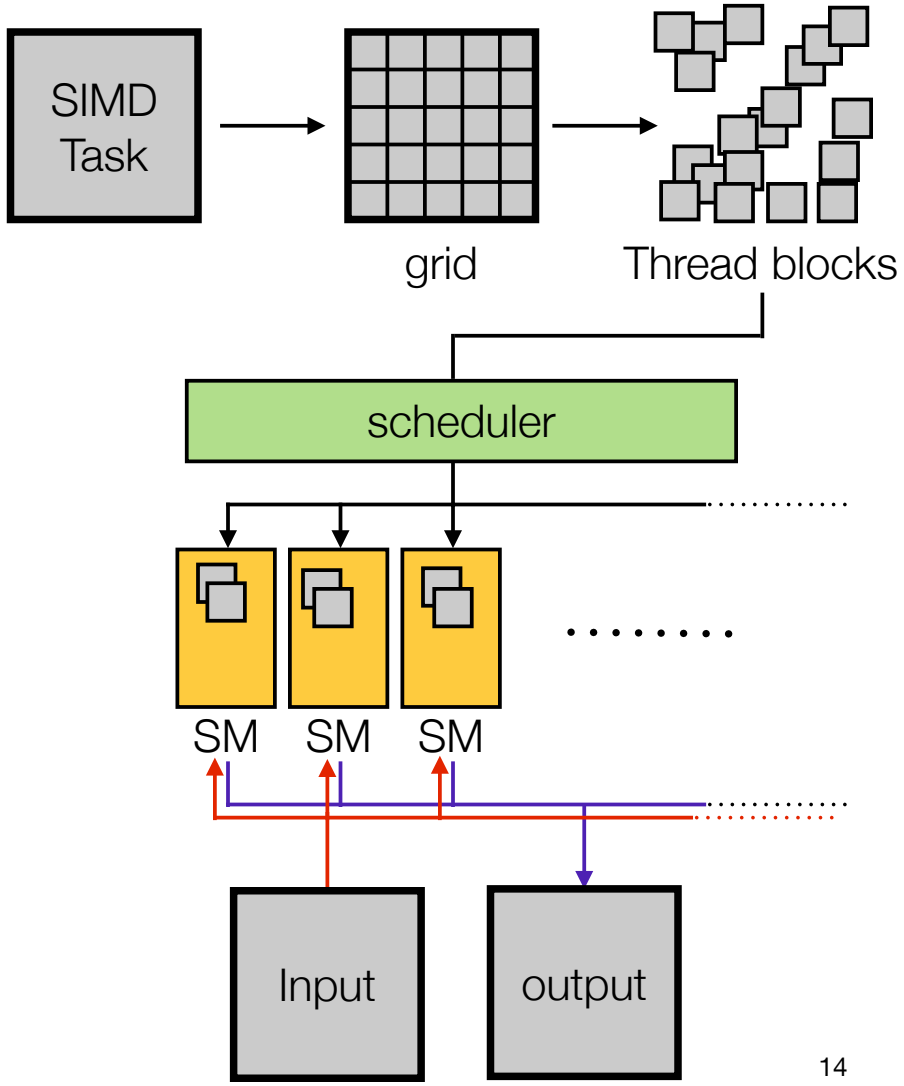
GPU programming

- CUDA constraints:
  - Maximum number of threads per block is 512.
  - The size of smem and registers limits SM occupancy.
  - Only 2D grid blocking is allowed.



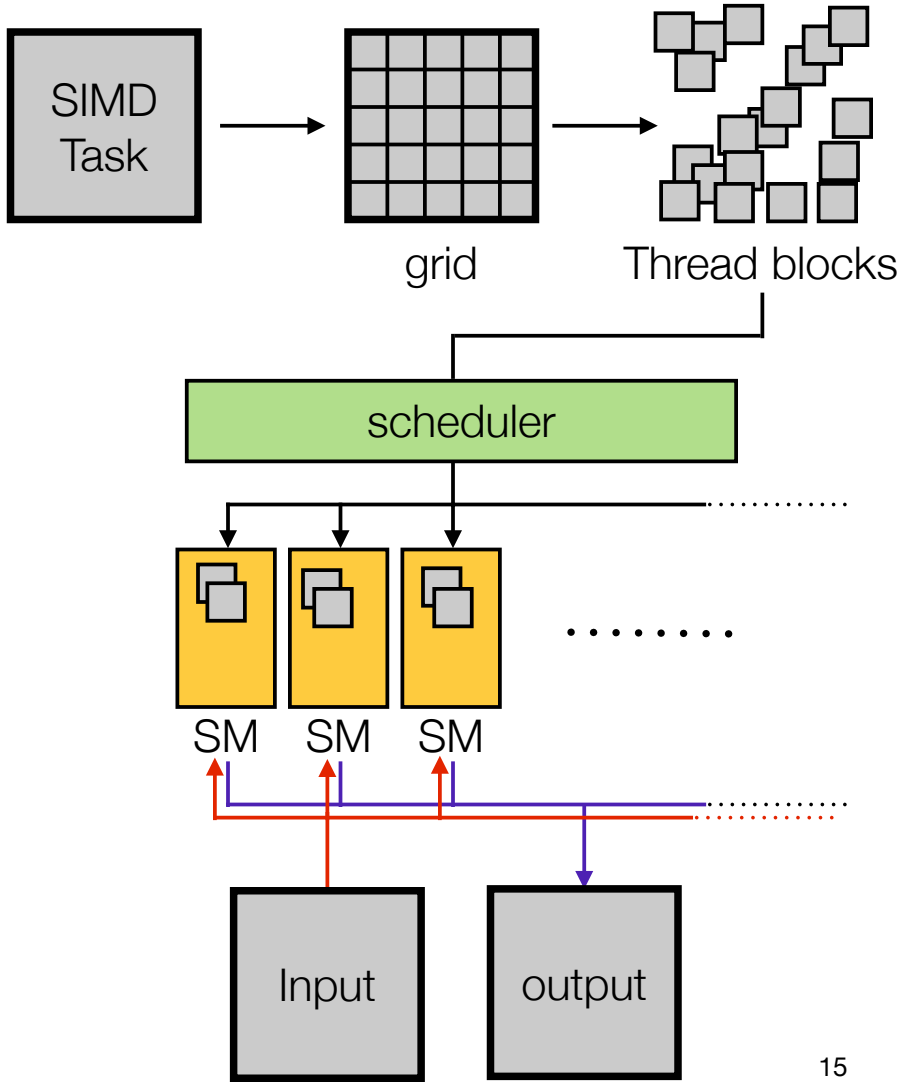
GPU programming

- CUDA constraints:
  - Maximum number of threads per block is 512.
  - The size of smem and registers limits SM occupancy.
  - Only 2D grid blocking is allowed.
- Optimization:
  - Exploit coalesced global memory access.
  - Use smem and avoid bank conflicts.
  - Maximize SM occupancy.
  - Avoid transfer from/to GPU.



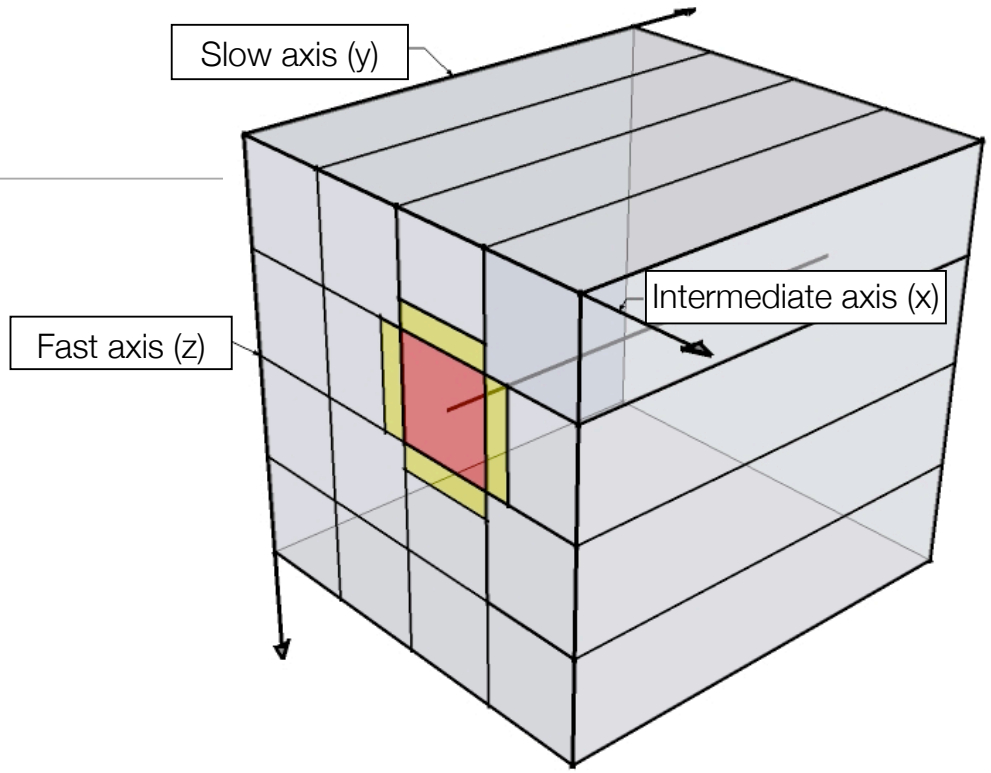
GPU programming

- CUDA constraints:
  - Maximum number of threads per block is 512.
  - The size of smem and registers limits SM occupancy.
  - Only 2D grid blocking is allowed.
- Optimization:
  - Exploit coalesced global memory access.
  - Use smem and avoid bank conflicts.
  - Maximize SM occupancy.
  - Avoid transfer from/to GPU.
- Performance evaluation:
  - A kernel's throughput is the number of output sample points per second.



Wave propagation

- Generalizing the FDTD implementation of Paulius Micikevicius increases the count of arithmetic operations and registers.
- Fast and intermediate axes are broken into 16x16 blocks.
- Each thread progresses to the end of the slow axis.
- Values needed to evaluate the laplacian term are copied to shared memory.



Host code:

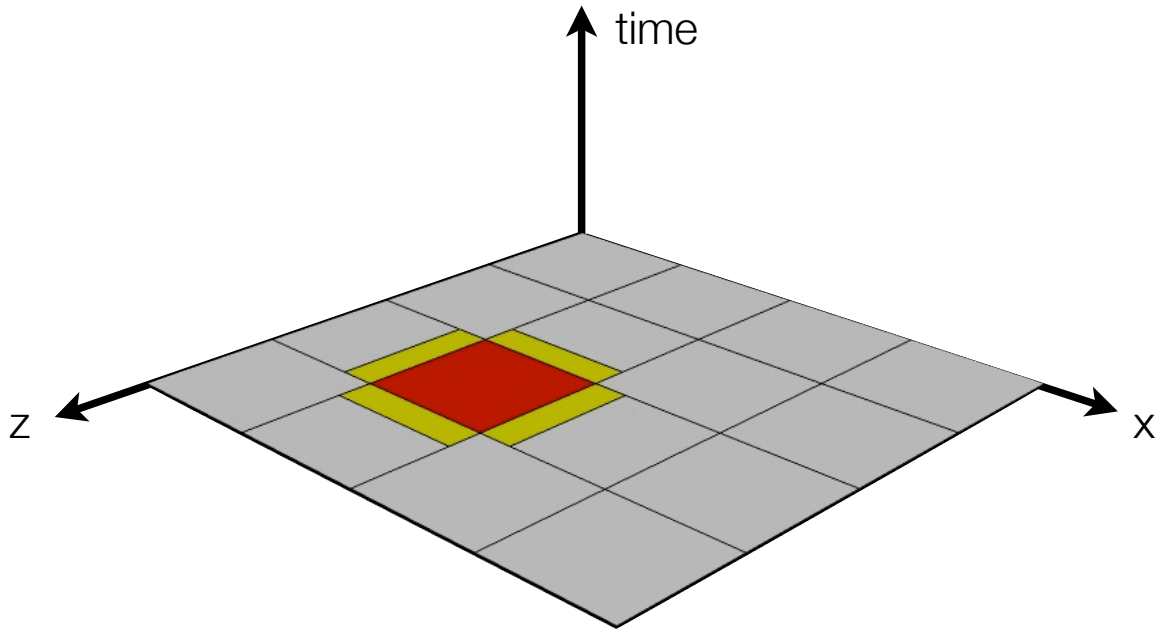
setup zx-grid
setup threadblocks
for $t = 1$ to $t_{\max}$ **do**
  call boundary condition kernel
  call FDTD kernel
end for

Device code:

FDTD kernel
compute x and z
for $y = 1$ to $y_{\max}$ **do**
  copy $P^t_{x,y,z}$ and halos to shared memory
  $P^{t+1}_{x,y,z} \leftarrow \text{extrapolate}(P^t, P^{t-1}, v, \text{stencil})$
end for

Multiple-time stepping

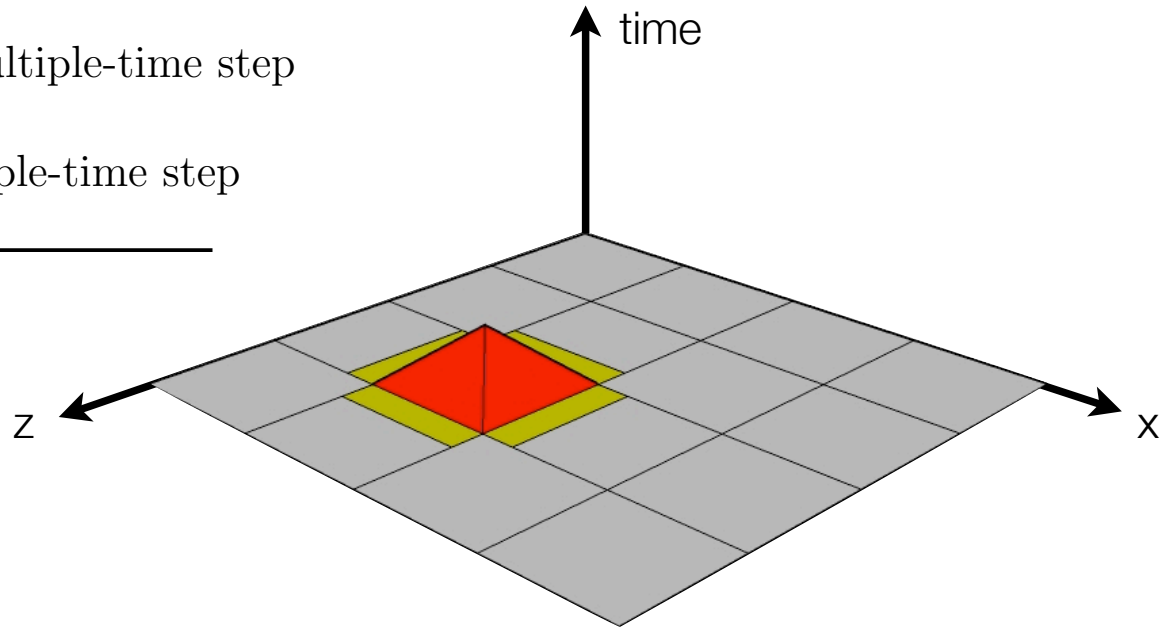
- For the original algorithm, each thread block accesses the global memory for corresponding values from four fields (P^{t+1}, P^{t}, P^{t-1}, v).
- Multiple-time stepping exploits temporal locality to reduce global memory accesses.



Multiple-time stepping

- For the original algorithm, each thread block accesses the global memory for corresponding values from four fields (P^{t+1}, P^t, P^{t-1}, v).
- Multiple-time stepping exploits temporal locality to reduce global memory accesses.
- Point-by-point time stepping:

for $t = 0$ to $t_{\max}$/step length **do**
  set boundaries for the next multiple-time step
  take a multiple-time step
  take the complementary multiple-time step
end for

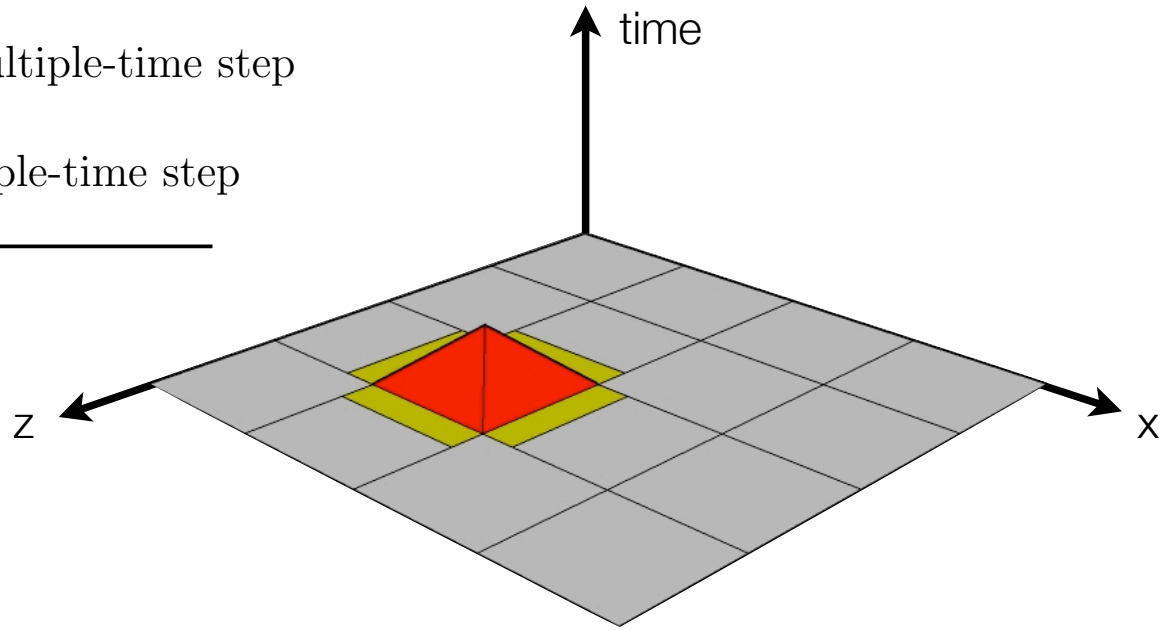


Multiple-time stepping

- For the original algorithm, each thread block accesses the global memory for corresponding values from four fields (P^{t+1}, P^t, P^{t-1}, v).
- Multiple-time stepping exploits temporal locality to reduce global memory accesses.
- Point-by-point time stepping:

for $t = 0$ to $t_{\max}$/step length **do**
  set boundaries for the next multiple-time step
  take a multiple-time step
  take the complementary multiple-time step
end for

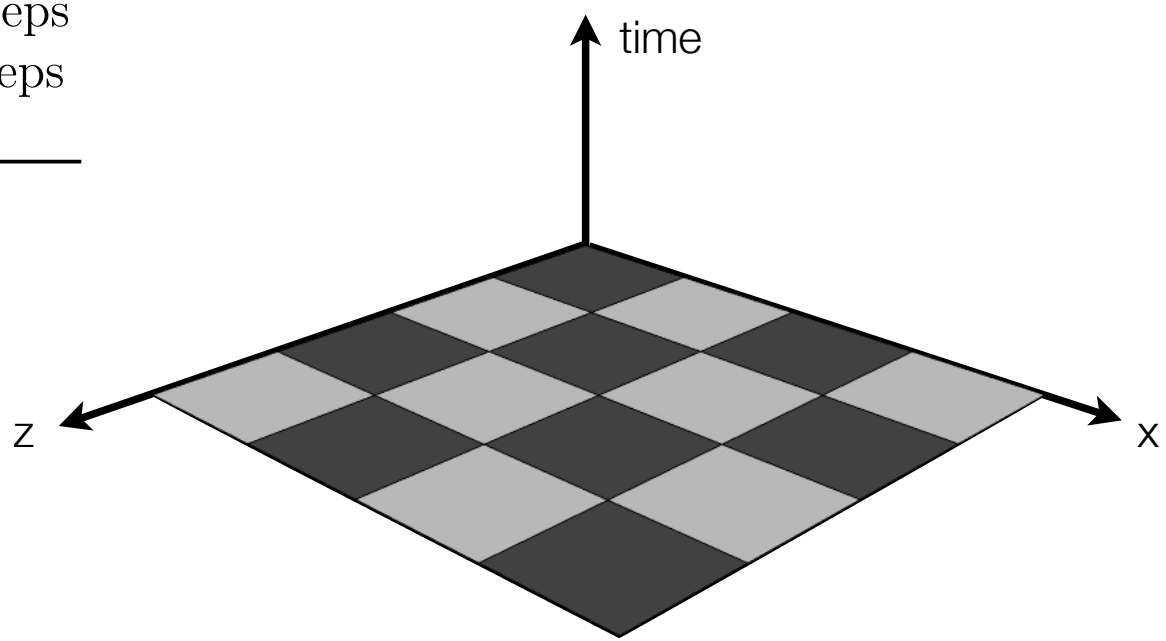
- Disadvantages:
  - divergent threads
  - not applicable for large stencils



Checkerboard multiple-time stepping

- The grid is divided into white and black thread blocks.
- checkerboard multiple-time stepping algorithm:

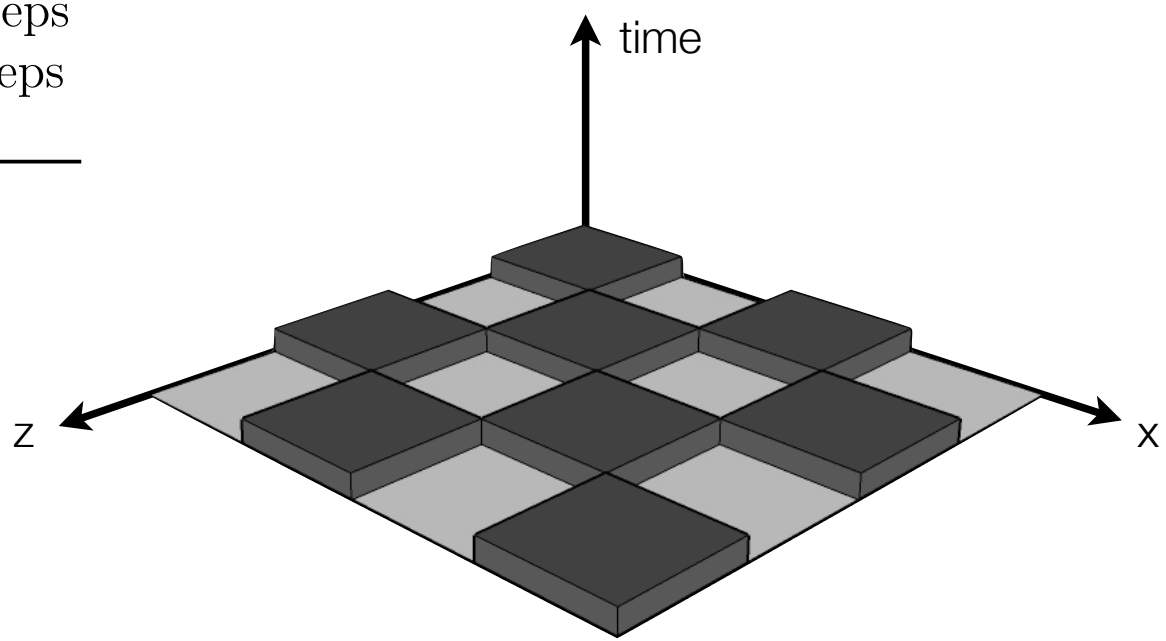
advance black blocks 1 step
for $t = 0$ to $t_{\max}/2$ **do**
  advance white blocks 2 steps
  advance black blocks 2 steps
end for



Checkerboard multiple-time stepping

- The grid is divided into white and black thread blocks.
- checkerboard multiple-time stepping algorithm:

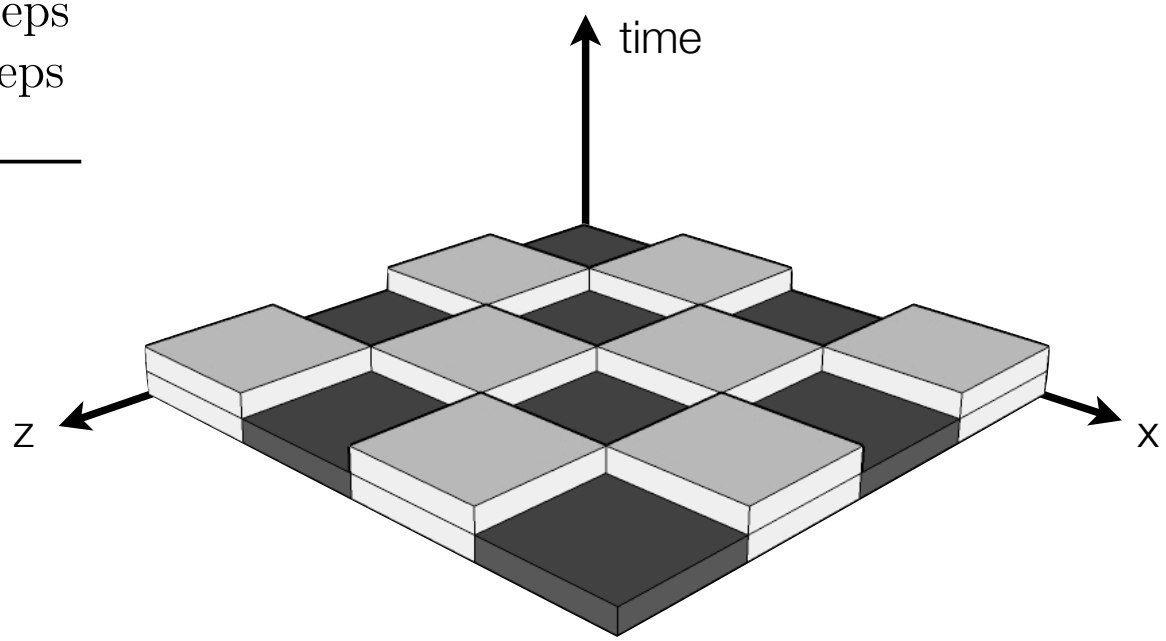
advance black blocks 1 step
for $t = 0$ to $t_{\max}/2$ **do**
  advance white blocks 2 steps
  advance black blocks 2 steps
end for



Checkerboard multiple-time stepping

- The grid is divided into white and black thread blocks.
- checkerboard multiple-time stepping algorithm:

advance black blocks 1 step
for $t = 0$ to $t_{\max}/2$ **do**
advance white blocks 2 steps
advance black blocks 2 steps
end for

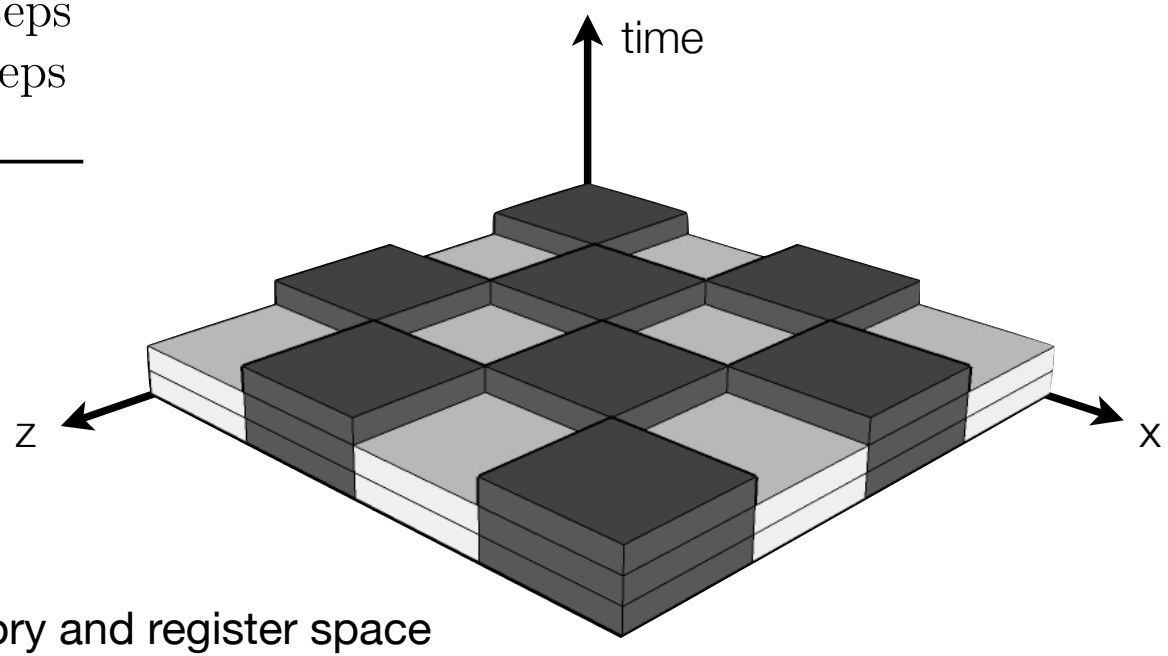


Checkerboard multiple-time stepping

- The grid is divided into white and black thread blocks.
- checkerboard multiple-time stepping algorithm:

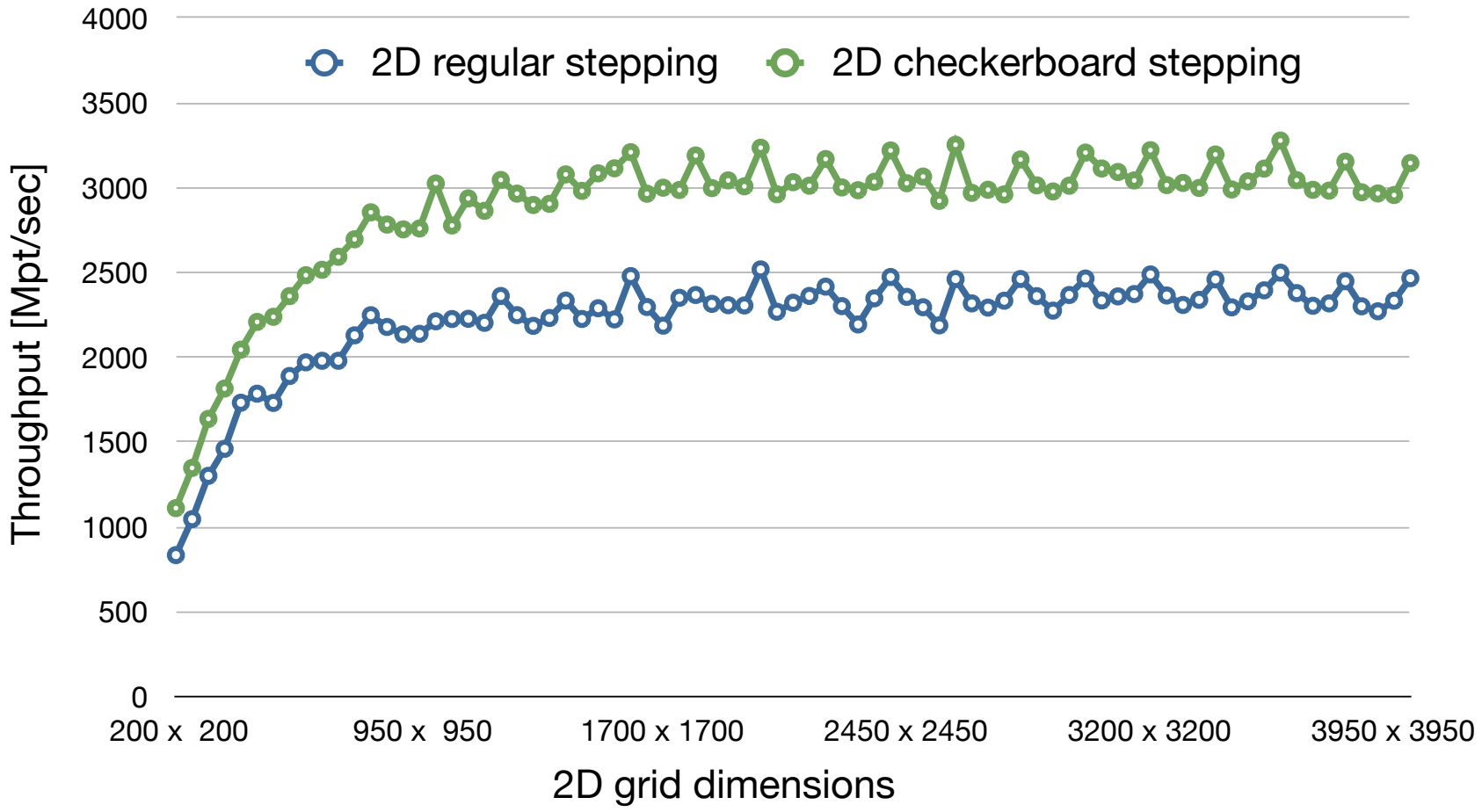
advance black blocks 1 step
for $t = 0$ to $t_{\max}/2$ **do**
  advance white blocks 2 steps
  advance black blocks 2 steps
end for

- Advantages:
  - saving of 3 global memory reads per thread
  - 35% reduction in global memory accesses
  - Applicable to large stencils
- Disadvantages:
  - Requires large shared memory and register space for 3D buffering.



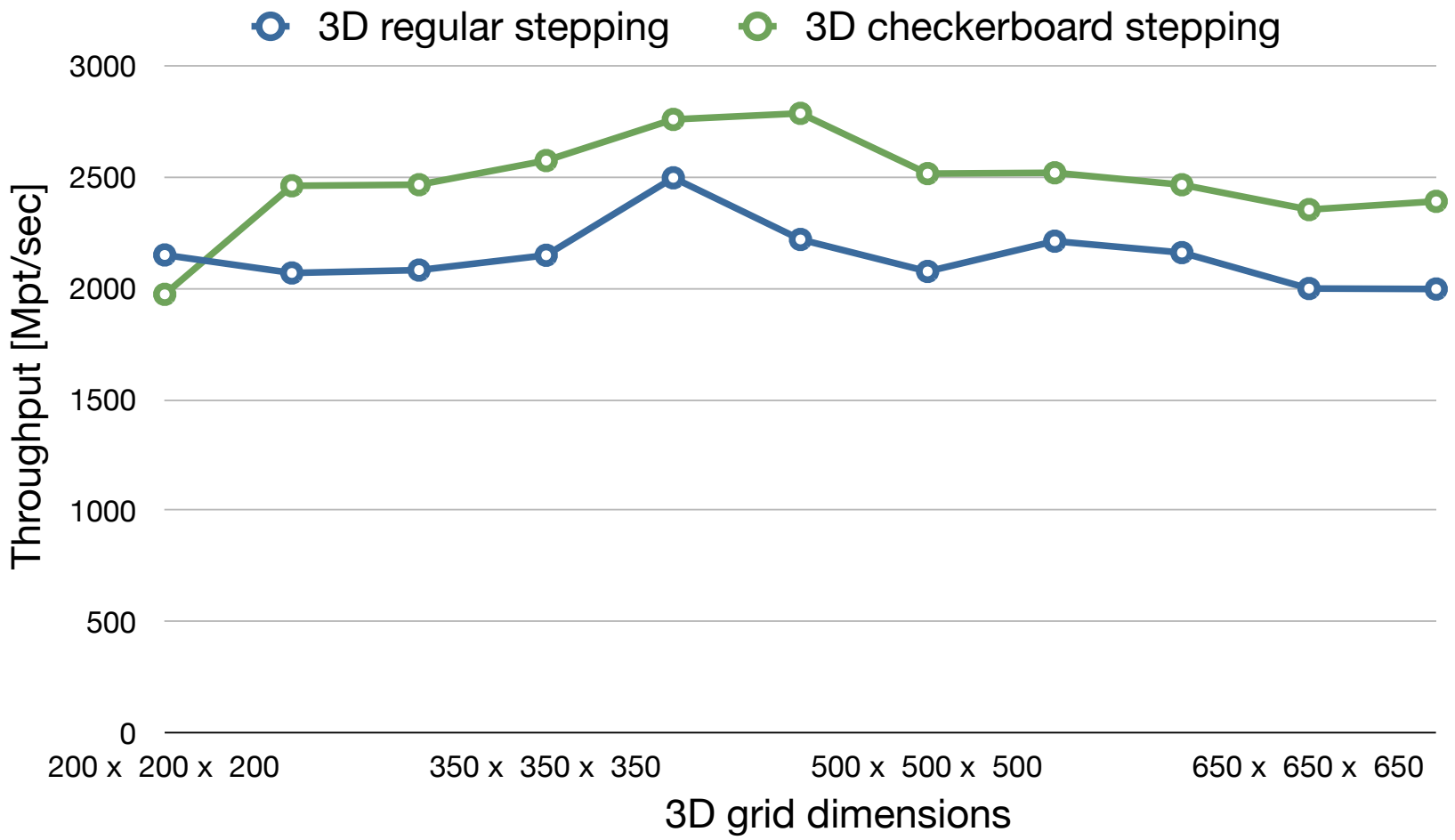
Checkerboard multiple-time steps (2D)

- 34% increase in throughput.



Checkerboard multiple-time steps (3D)

- 22% increase in throughput.



RTM with absorptive boundary condition

- A damping term is added to the wave equation.

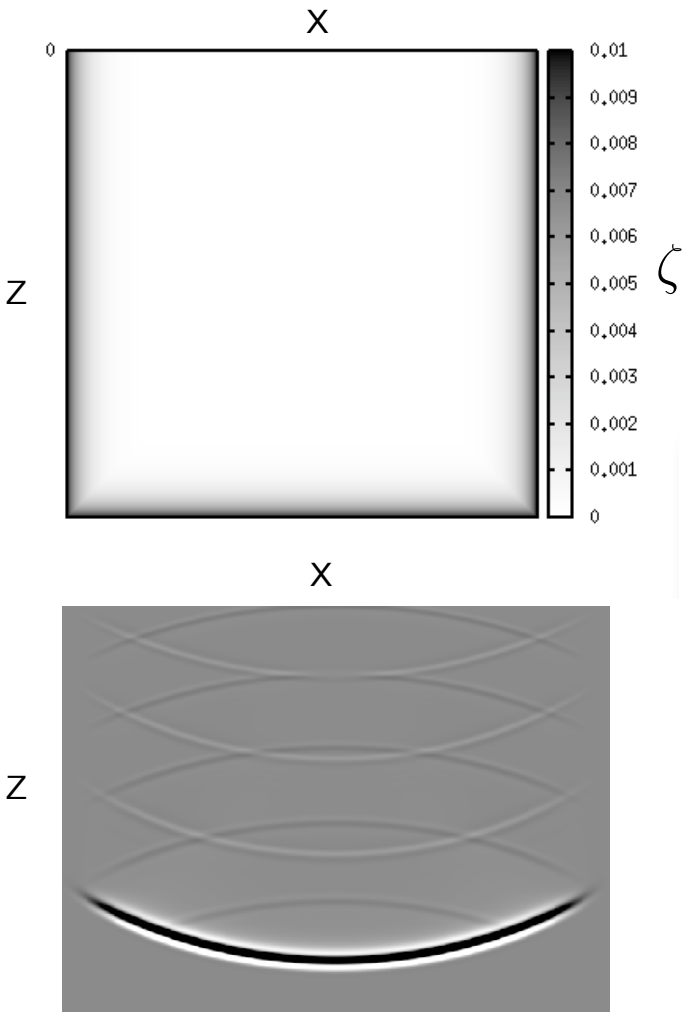
$$(\nabla^2 - \frac{1}{v^2}\frac{\partial^2}{\partial t^2} - \zeta\frac{\partial}{\partial t})P = f$$

$$\zeta = \alpha e^{-\min(|\Delta x_i|)/\beta}$$

α Scaling coefficient β Damping rate

- RTM algorithm with 2 passes of wave propagation:

for $t = 0$ to $t_{\max}$ **do**
  $P_f^{t+1} \leftarrow \text{extrapolate}(P_f^t, P_f^{t-1}, v, \text{stencil})$
  if t is imaging step **then**
➔ save P_f^t
  end if
end for
for $t = t_{\max}$ to 0 **do**
  $P_b^{t-1} \leftarrow \text{extrapolate}(P_b^t, P_b^{t+1}, v, \text{stencil})$
  if t is imaging step **then**
    call imaging(P_f^t, P_b^t)
  end if
end for



RTM with random boundary conditions

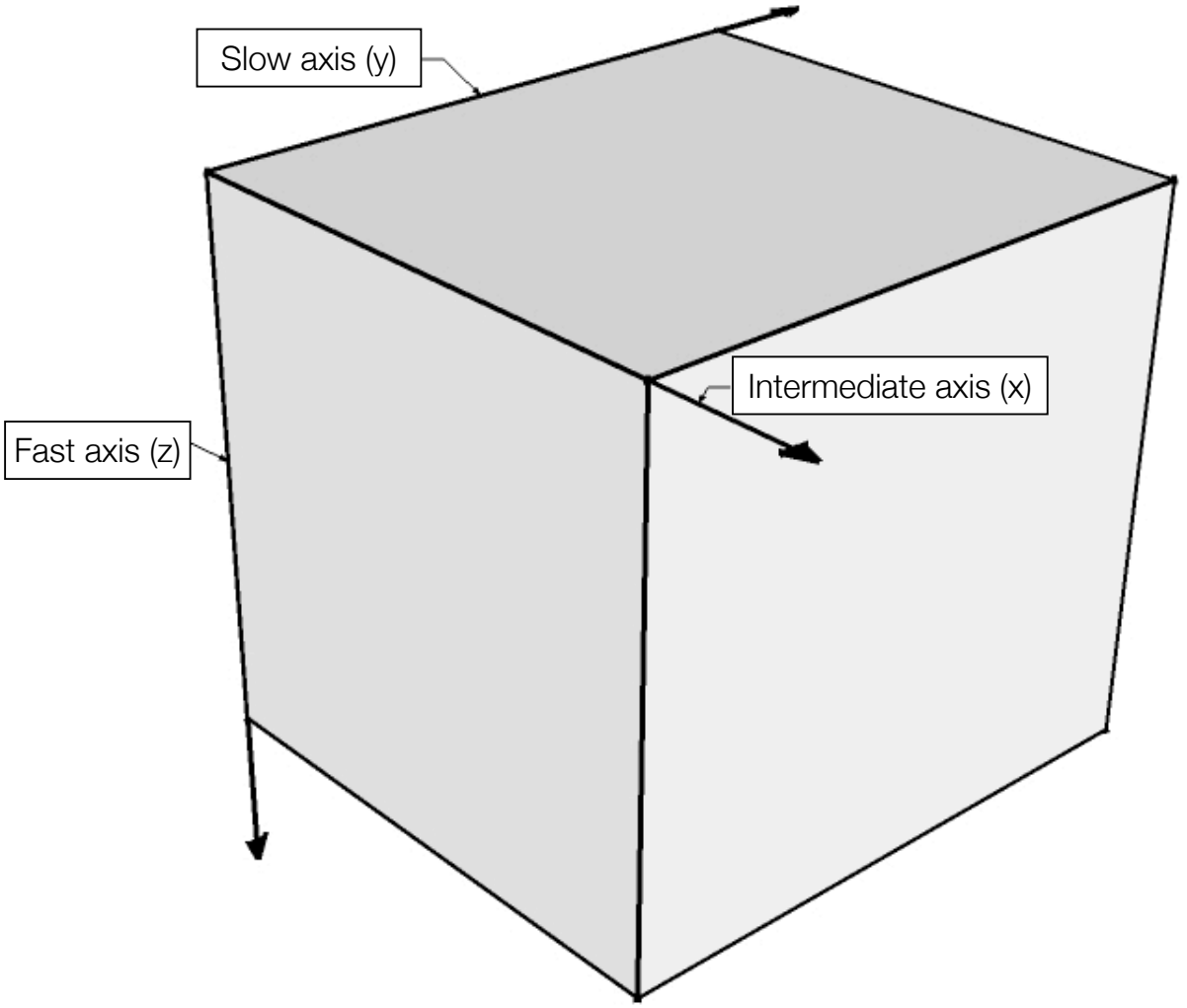
- 4GB of memory on the GPU is not sufficient to save wavefields needed for 3D imaging.
- Random boundary condition alleviates the need for saving wavefields at the cost of one additional wave propagation.
- RTM algorithm with 3 passes of wave propagation.

for $t = 0$ to $t_{\max}$ **do**
  $P_f^{t+1} \leftarrow \text{extrapolate}(P_f^t, P_f^{t-1}, v, \text{stencil})$
end for
for $t = t_{\max}$ to 0 **do**
  $P_b^{t-1} \leftarrow \text{extrapolate}(P_b^t, P_b^{t+1}, v, \text{stencil})$
  $P_f^{t-1} \leftarrow \text{extrapolate}(P_f^t, P_f^{t+1}, v, \text{stencil})$
  if t is imaging step **then**
    call imaging(P_f^t, P_b^t)
  end if
end for

GPU capacity for 3D RTM with random boundaries

- 1-Tesla S1070 GPU (4GB):
  - maximum cubic grid size:
    - 550 x 550 x 550*
- 4-Tesla S1070 GPUs (16GB):
  - maximum cubic grid size:
    - 900 x 900 x 900*

- * the remaining storage is for data, source, geometry, etc.



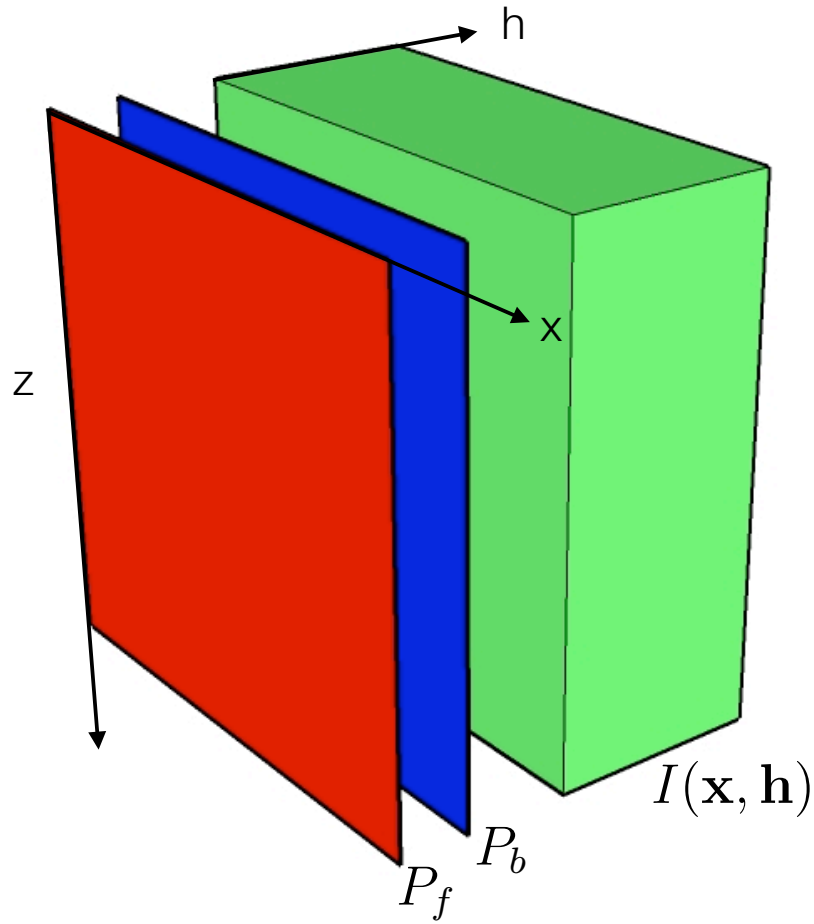
Offset domain common image gathers computing

- Imaging condition:

$$I(\mathbf{x},\mathbf{h}) = \sum_s \sum_t P_f(\mathbf{x}+\mathbf{h},t;s) P_b(\mathbf{x}-\mathbf{h},t;s)$$

- The output volume of ODCIGs is much larger (3D) than that of FDTD (2D).

- ODCIGs for 3D can not be hosted on GPU memory.



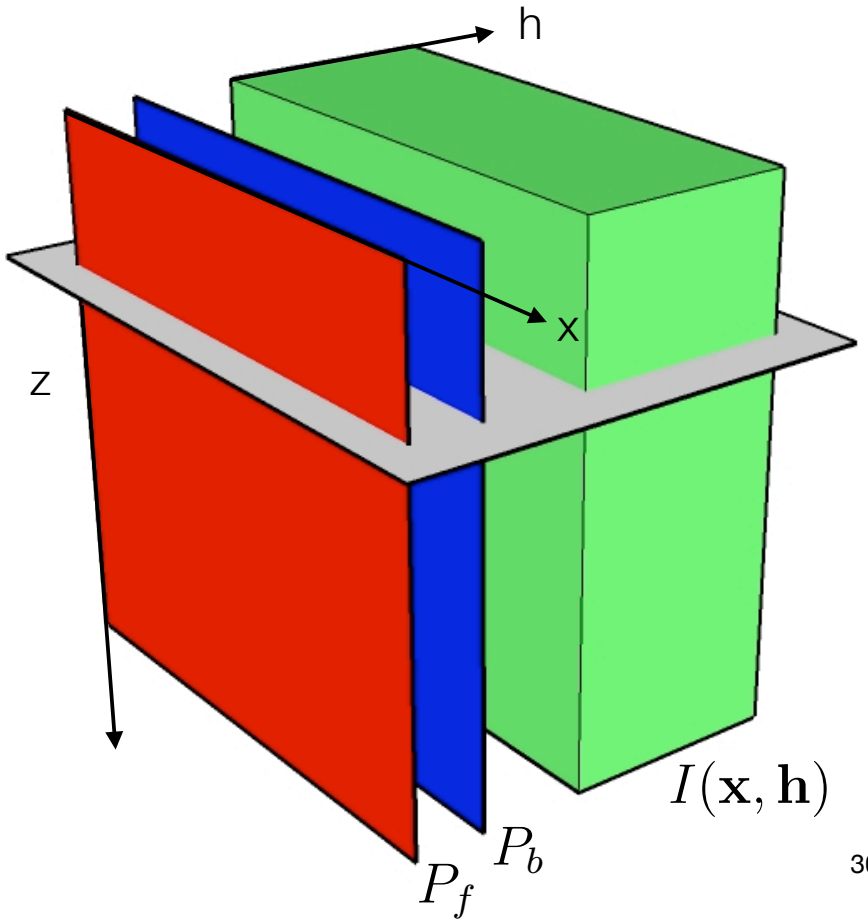
Offset domain common image gathers computing

- Imaging condition:

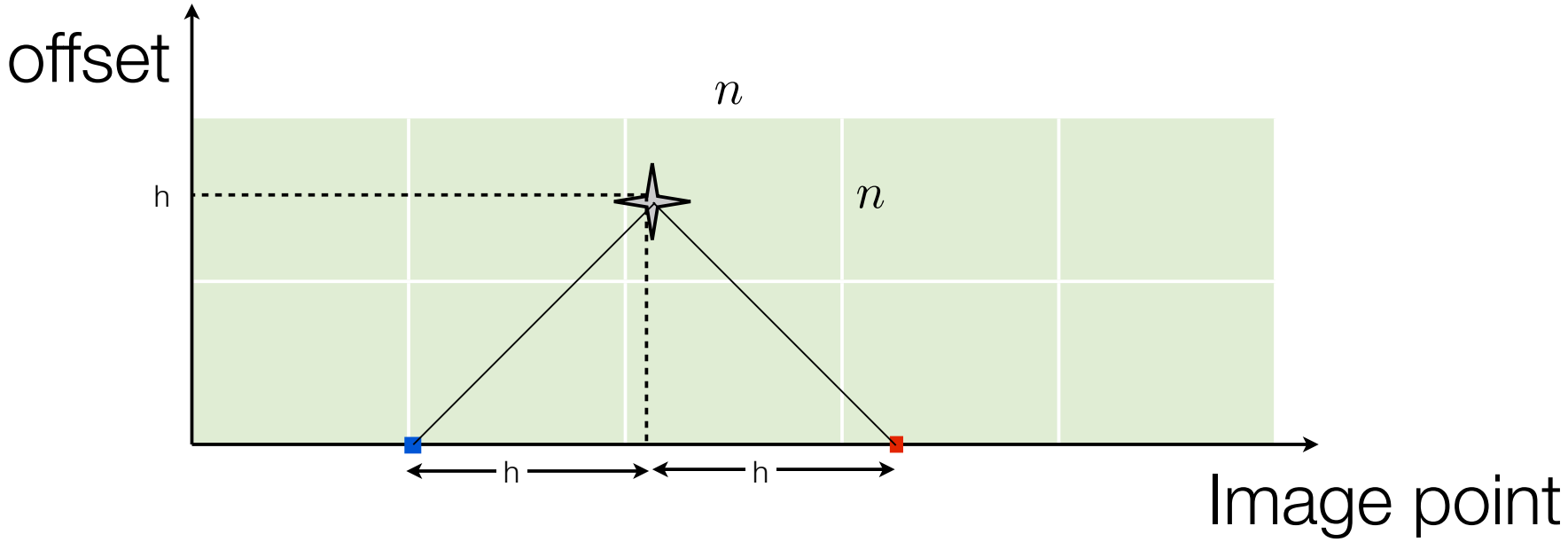
$$I(\mathbf{x},\mathbf{h}) = \sum_s \sum_t P_f(\mathbf{x}+\mathbf{h},t;s) P_b(\mathbf{x}-\mathbf{h},t;s)$$

- The output volume of ODCIGs is much larger (3D) than that of FDTD (2D).

- ODCIGs for 3D can not be hosted on GPU memory.

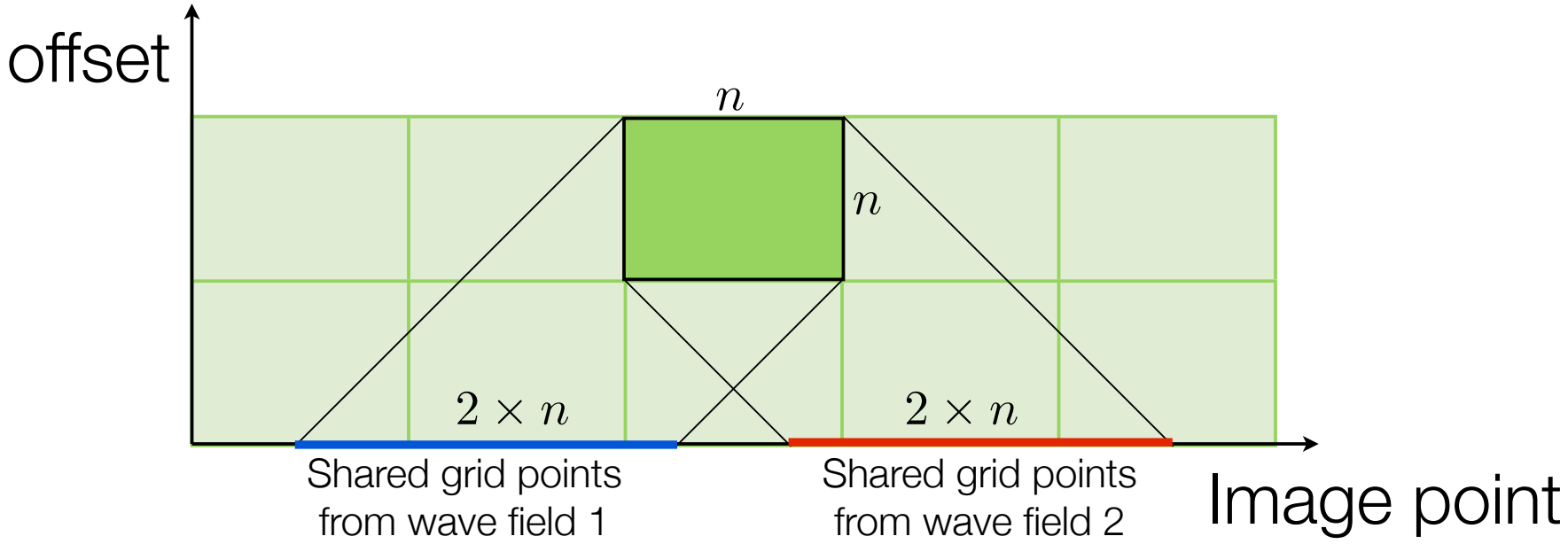


Computing ODCIGs (Naive)



global memory access count = $4 \times n^2$

Computing ODCIGs (Improved)

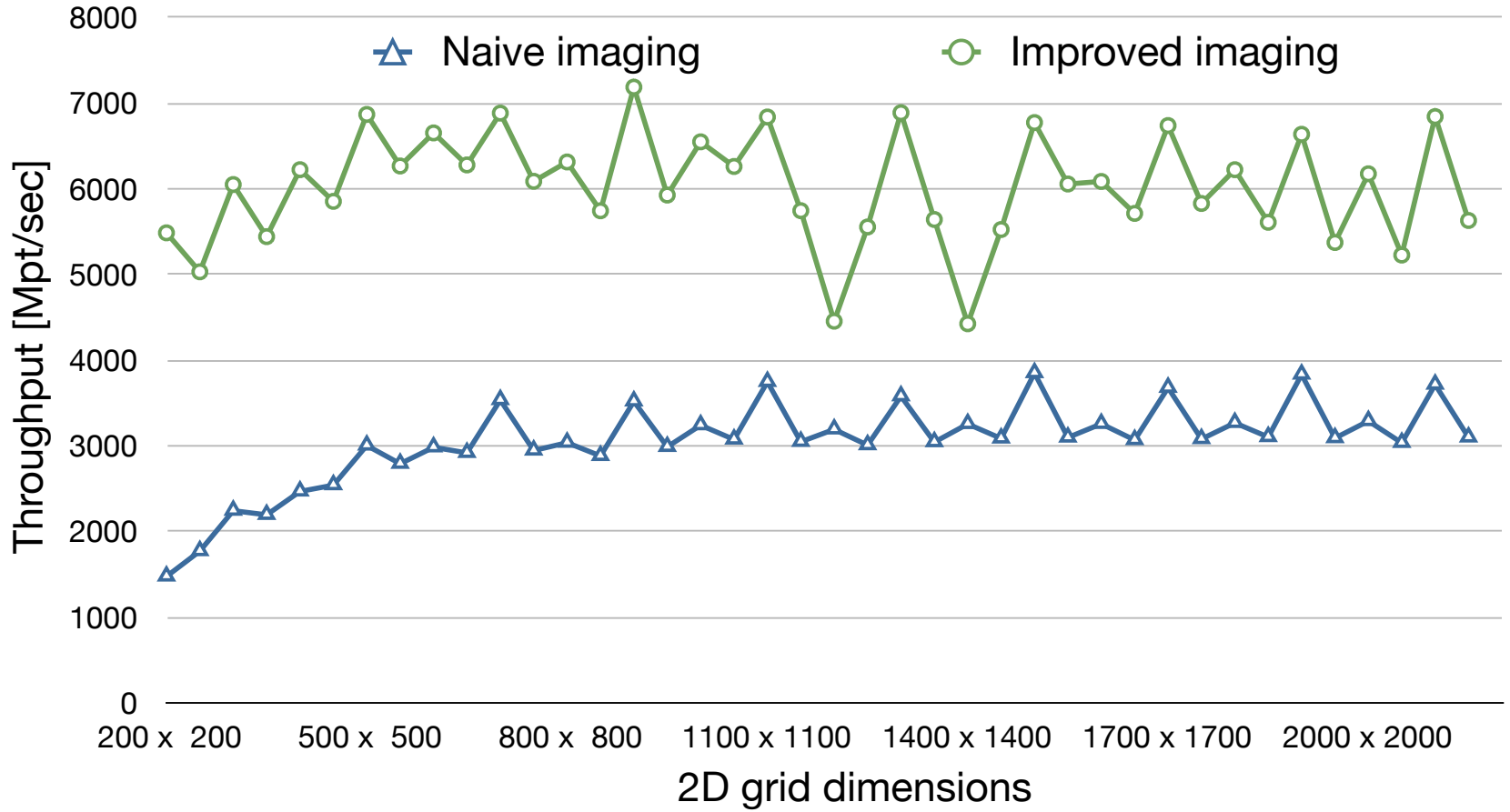


global memory access count = $2 \times n^2 + 4 \times n$

For 16x16 block size the number of global memory accesses is reduced from 1024 to 576.

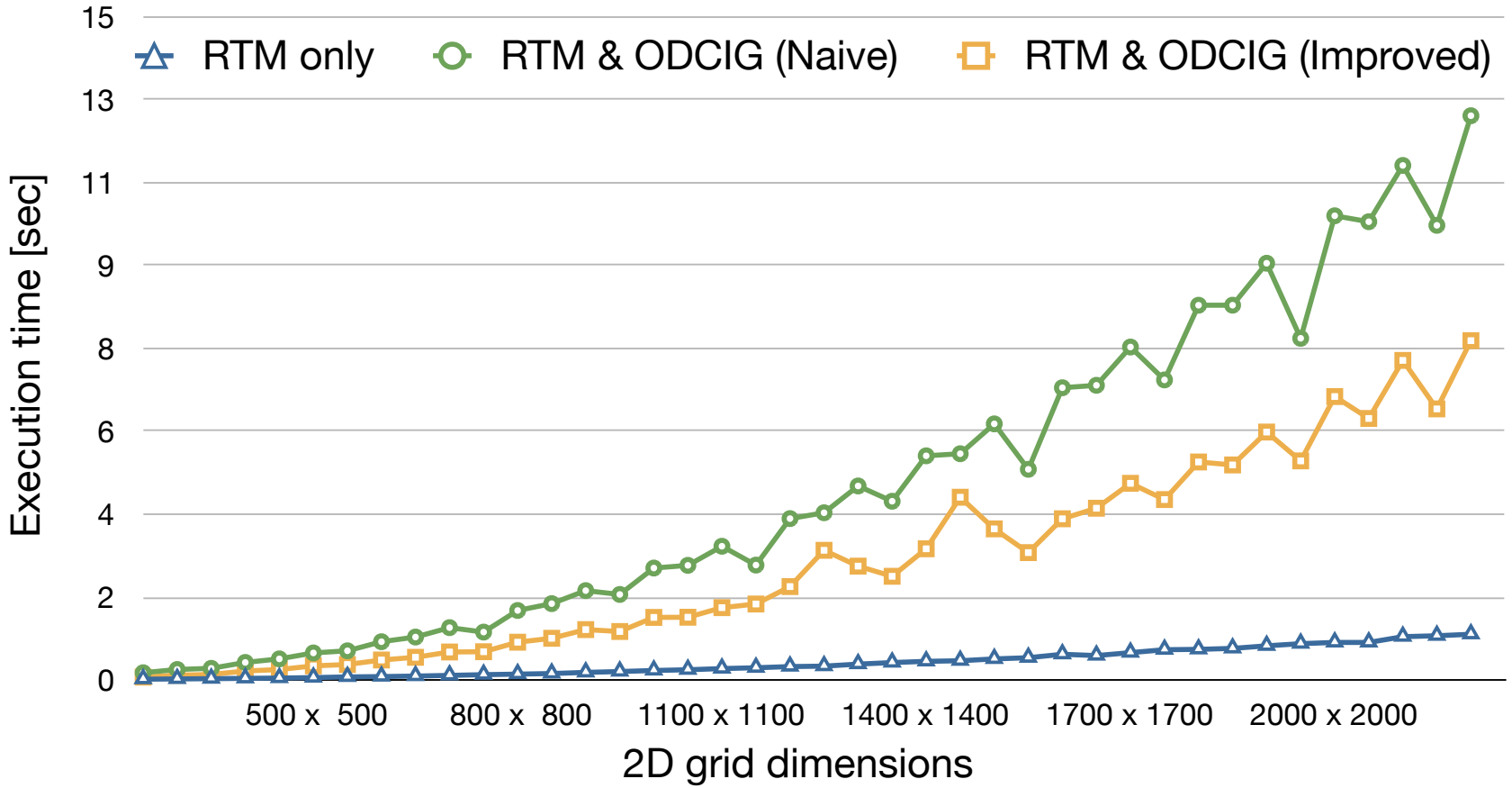
Kernel performance

- About 40% increase in throughput.



Algorithms' performance

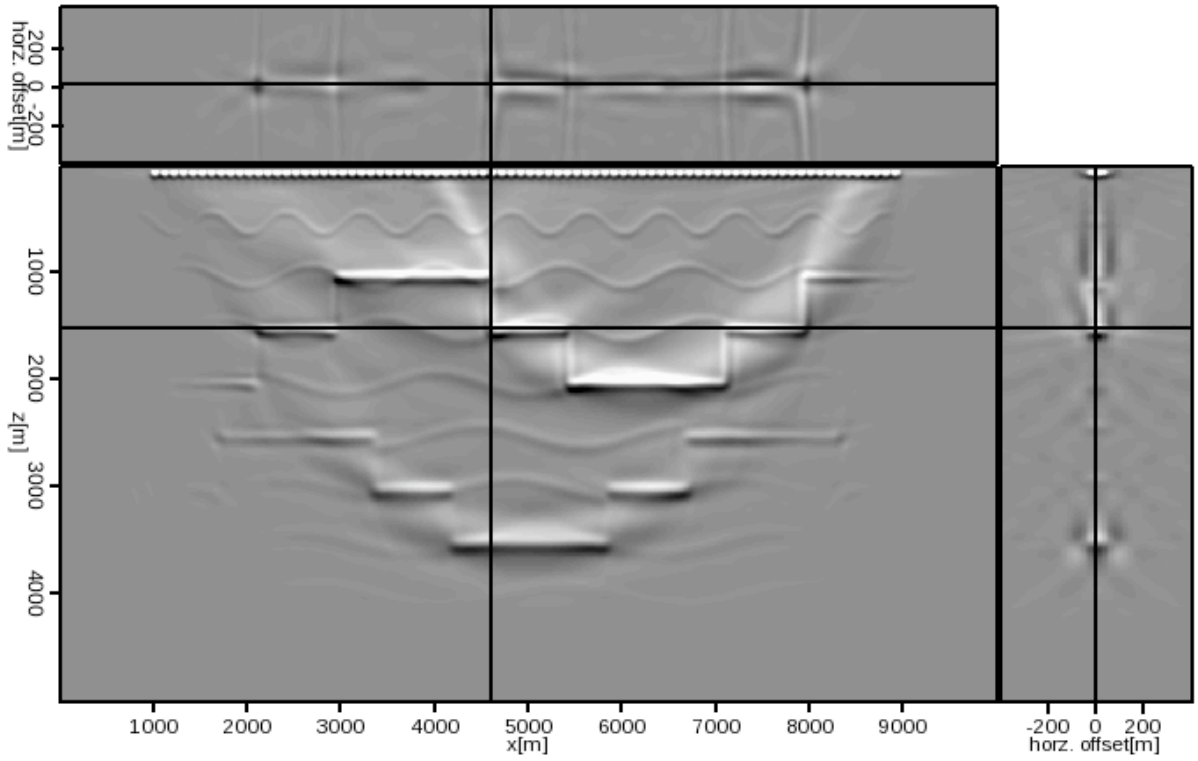
- 2D RTM Execution time for 1000 time steps and 100 imaging steps:



2D synthetic data example

- 2D RTM with ODCIGs generation
- Wavefield needed for imaging are saved on memory.

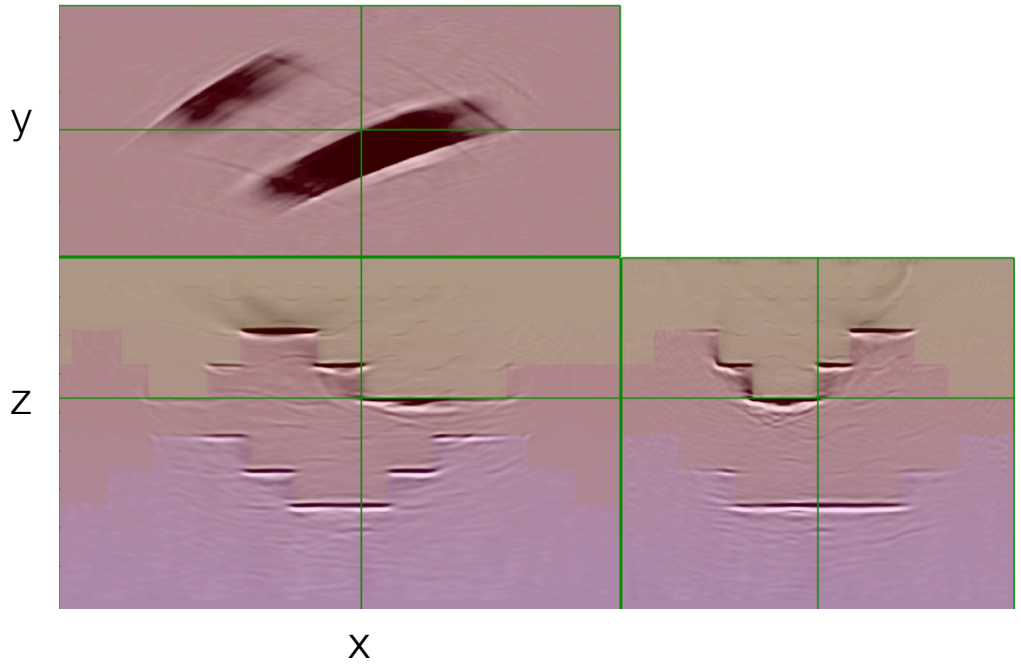
grid size	1000 x 1000
time steps	6000
#shots	90
#offsets	2x81
resources	1 GPU
modeling time	7 min
migration time	38 min



3D synthetic data example

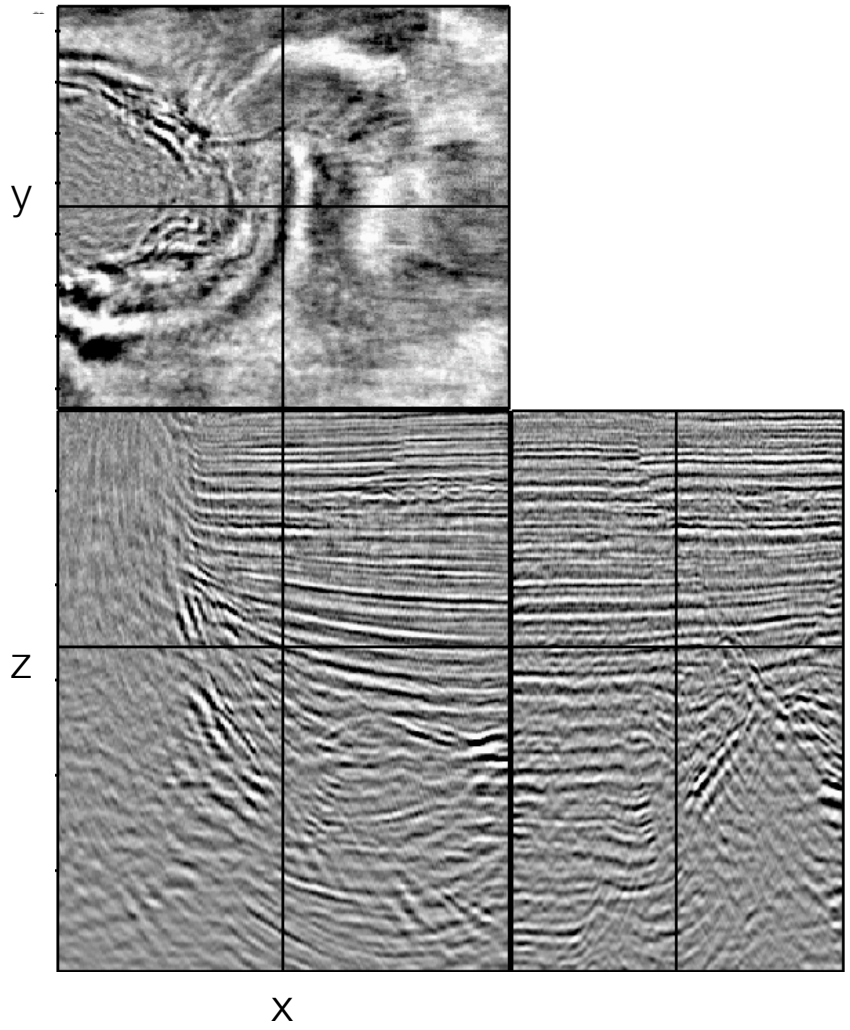
- pre-stack shot profile RTM
- without saving wavefields
- grid size: 450 x 450 x 450
- time steps: 4000
- imaging steps: 400

- 1 shot
  - modeling: 3 minutes
  - migration: 9 minutes
- 25 shots
  - modeling 1 hour & 16 minutes
  - migration 3 hours & 41 minutes

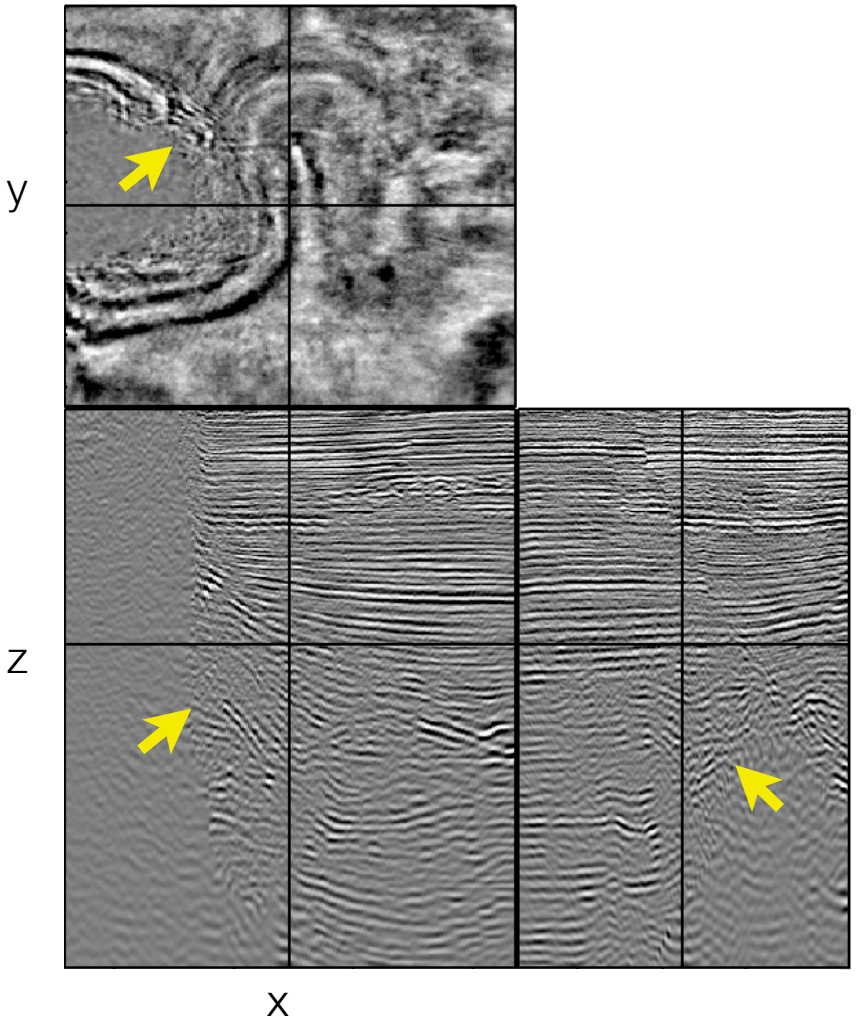


3D field data example

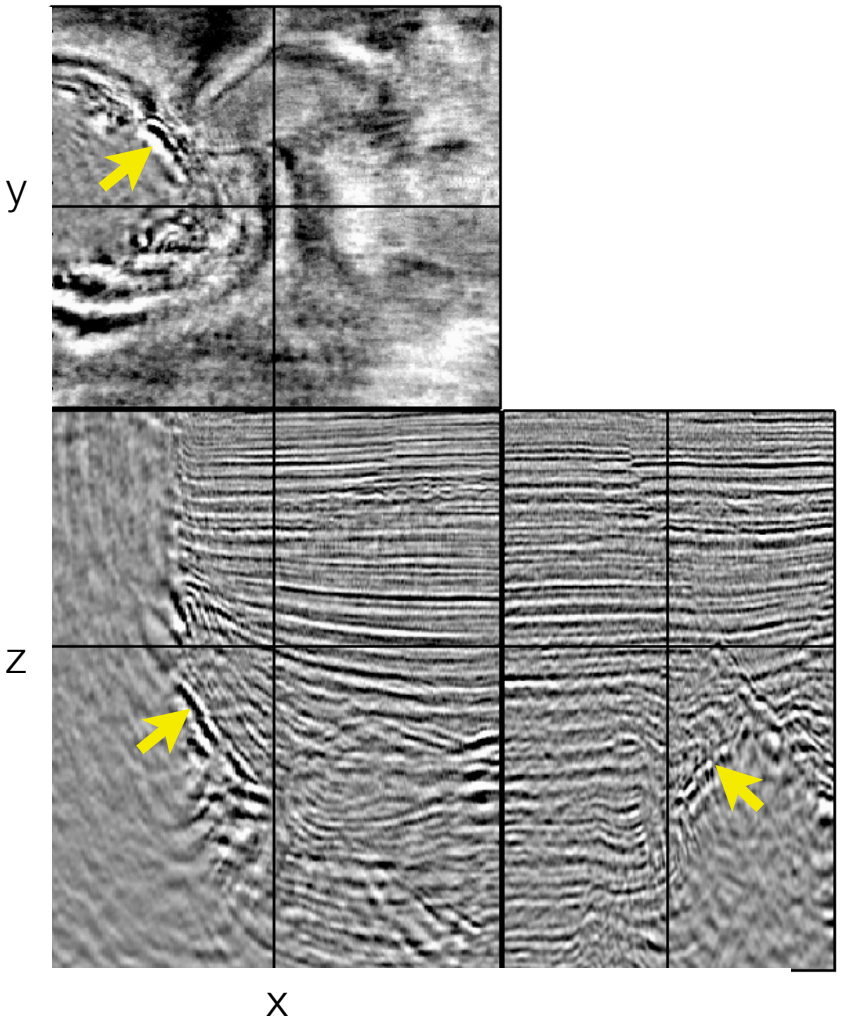
- zero-offset migration
- DMOed data
- grid size: 740x460x300
- time steps: 7000
- migration time: 6 minutes on 1 GPU



3D field data example



one-way



two-way

Summary

- GPUs and CUDA allow fast development of data parallel algorithms.

Summary

- GPUs and CUDA allow fast development of data parallel algorithms.
- Checkerboard multi-time stepping improves performance by 20%.

Summary

- GPUs and CUDA allow fast development of data parallel algorithms.
- Checkerboard multi-time stepping improves performance by 20%.
- Due to the memory access pattern of ODCIGs kernels, ODCIGs computing slow down RTM.

Summary

- GPUs and CUDA allow fast development of data parallel algorithms.
- Checkerboard multi-time stepping improves performance by 20%.
- Due to the memory access pattern of ODCIGs kernels, ODCIGs computing slow down RTM.
- Blocking and pre-loading to shared memory improves performance of ODCIGs generation by about 40%.

Summary

- GPUs and CUDA allow fast development of data parallel algorithms.
- Checkerboard multi-time stepping improves performance by 20%.
- Due to the memory access pattern of ODCIGs kernels, ODCIGs computing slow down RTM.
- Blocking and pre-loading to shared memory improves performance of ODCIGs generation by about 40%.
- Rapid execution of kernels can be advantageous for interactive processing.

Thank you

- Acknowledgments
  - Thanks to NVIDIA for donating Tesla S1070 GPUs to SEP.
  - Thanks to Paulius Micikevicius for sharing his FDTD kernel.
  - Unocal (Chevron) for providing the field data.