GPGPU Accelerated Reverse Time Migration

Nader Moussa, Stanford Exploration Project

13 May 2009

SEP-138, pg. 269-282

Future of High Performance Computing

- Scientific computing is changing rapidly
  - Serial programming may be insufficient for future problems
  - New architectures like FPGAs, GPGPUs, and massive distributed parallel systems provide significant performance improvement over conventional systems
- This new computational power will broaden the horizon of possible research
- Topics for investigation:
  - Evaluation of the cost versus performance for new computer architectures
  - Mapping the performance bottlenecks and limitations of each architecture
  - Bridging the gap between high-level geophysical abstraction and tunable implementation details

Talk Overview

GPGPU Overview

1. GPU Hardware Basics
2. CUDA Programming Methodology
3. Transitioning code to GPGPUs

RTM Implementation

1. RTM Algorithm parallelism
2. Optimizations for GPU
3. Taking advantage of the latest technology

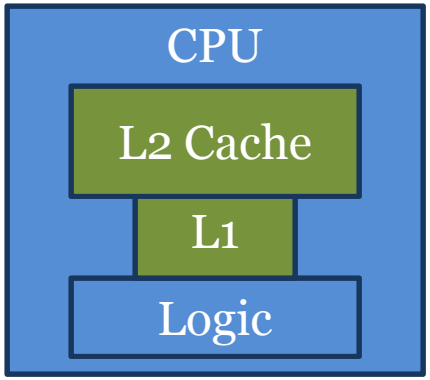
Introduction to scientific computing on GPU hardware

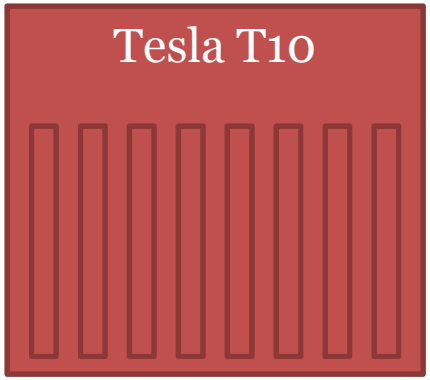
GPGPU OVERVIEW

CPU vs. GPU

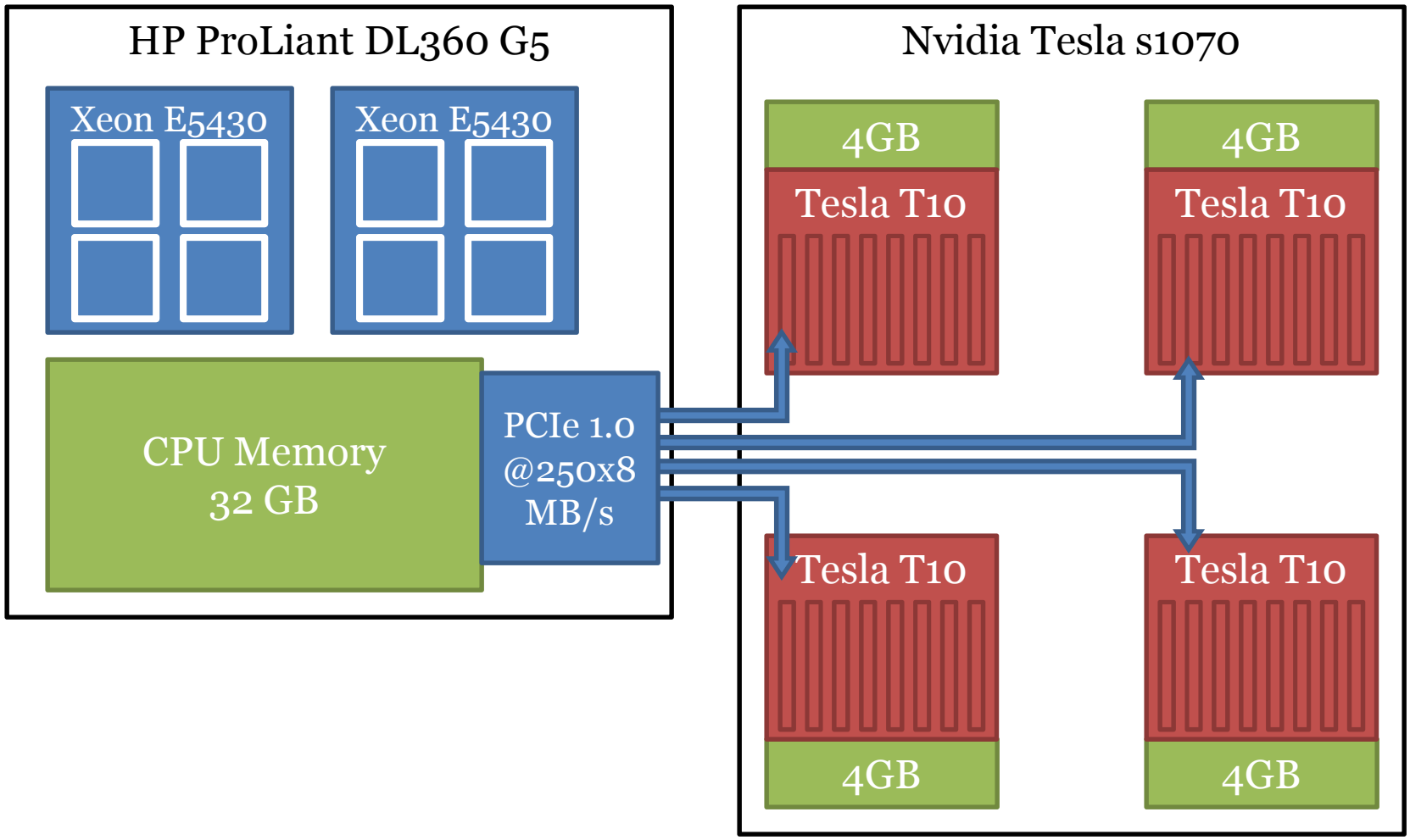
- Designed for general purpose serial operation
- Shared function units
- Hierarchical memory structure

- Highly parallelized
- Many Floating Point Units
- Parallel memory structure
- Specialized circuitry for certain mathematical logic (geometry, coordinates)





Tesla S1070 Architecture



T10 GPU Specifications

```
[nwmoussa@tesla0 ~]$ cuda_sdk/bin/linux/release/deviceQuery

There are 4 devices supporting CUDA

Device 0: "Tesla T10 Processor"
  Major revision number:                         1
  Minor revision number:                         3
  Total amount of global memory:                 4294705152 bytes
  Number of multiprocessors:                     30
  Number of cores:                               240
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       16384 byte
  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
  Maximum memory pitch:                          262144 bytes
  Texture alignment:                             256 bytes
  Clock rate:                                    1.30 GHz
  Concurrent copy and execution:                 Yes
```

GPU Hardware is great for imaging

- Massively parallel arithmetic hardware
  - 240 cores per GPU
  - Many Floating Point units
- Parallel memory management system
  - Traditional cache structure is replaced with block shared memories
  - Large number of very fast access registers
- Hierarchical parallelism
  - Coarse and fine-grained algorithm splitting

This fits well with our seismic processing algorithm structures

Introduction to scientific computing on GPU hardware

CUDA PROGRAMMING METHODOLOGY

General Purpose GPU

"Compute Unified Device Architecture" (CUDA)

- a software interface and compiler technology
- allows hardware access to the modern GPU without the abstraction of "graphical operations"

- Standard C programming with some extensions

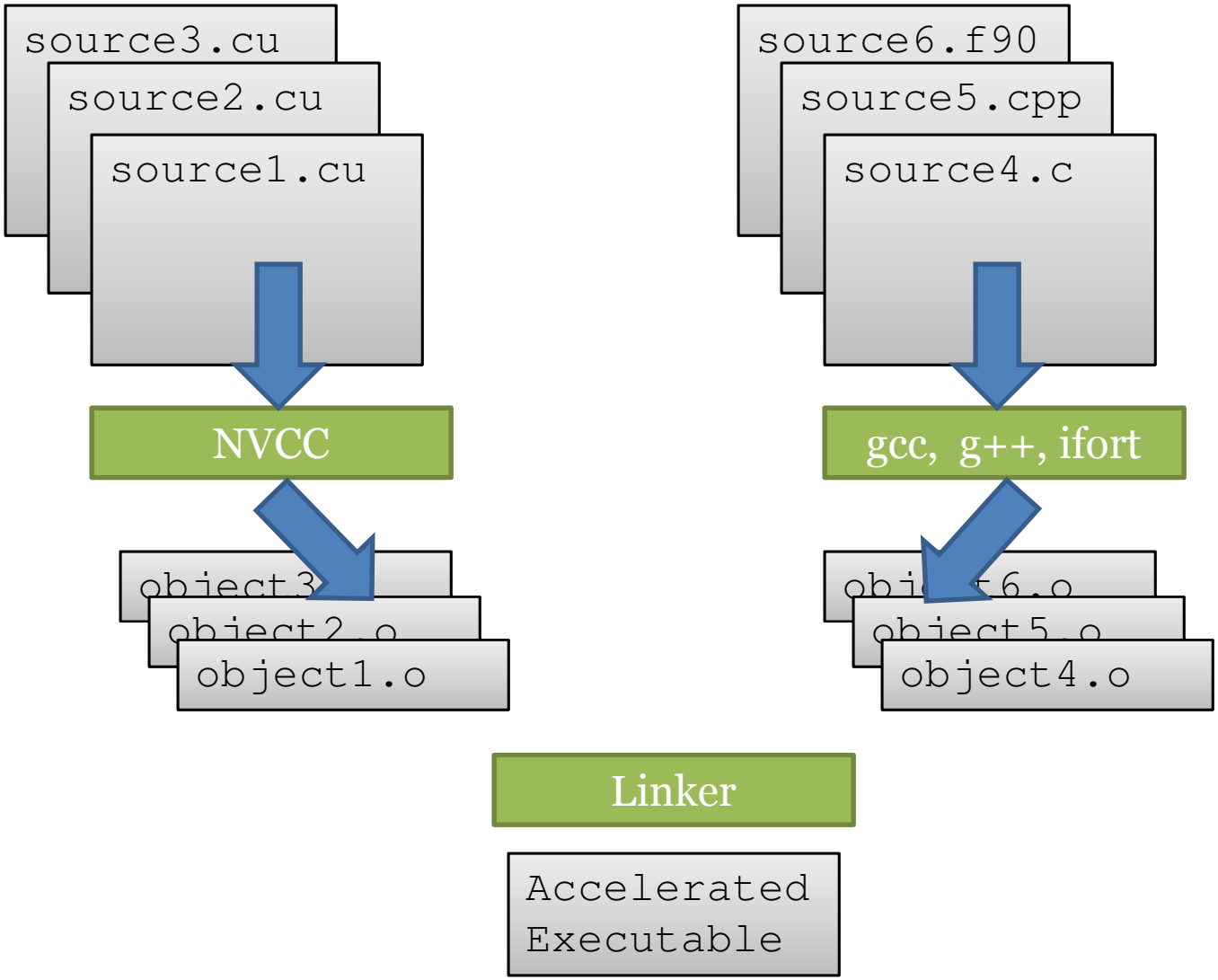
Programmers get easy access to the acceleration of massively parallel hardware, without relearning everything from scratch

CUDA Software Model

- Host code runs on CPU
- Device code (kernel) runs on GPU
- Software abstractions wrap the parallel hardware

Software Model		Hardware Model	
Element	Maximum	Physical Unit	#
Thread	512 threads per block Arranged in 3D block not exceeding $512 \times 512 \times 64$ in $< x, y, z >$ and 512 total	Scalar Processor (**SP**) or "Streaming **Core**" Each core executes one thread at a time	**8**
Warp	Each 32 threads are statically assigned to a warp	**SP Pipeline** A full warp (32 threads) executes in 4 clock cycles (pipelined 4-deep across 8 cores)	**16**
Block	Arranged in 2D grid not exceeding 65535×65535 in $< x, y >$	Streaming Multiprocessor (**SM**)	**30**
Kernel Grid	Problem or simulation representation	**GPU** Only one kernel is running on the GPU at a time (More are possible, but this is complicated).	**4**

CUDA Compiler



CUDA Threads

- Basic element of GPU parallelism
- GPU threads are very lightweight
- Single instruction can execute across multiple threads
- Ideal program will have tens of thousands of CUDA threads

Software Model		Hardware Model	
Element	Maximum	Physical Unit	#
Thread	512 threads per block Arranged in 3D block not exceeding $512 \times 512 \times 64$ in $< x, y, z >$ and 512 total	Scalar Processor (**SP**) or "Streaming **Core**" Each core executes one thread at a time	**8**
Warp	Each 32 threads are statically assigned to a warp	**SP Pipeline** A full warp (32 threads) executes in 4 clock cycles (pipelined 4-deep across 8 cores)	**16**
Block	Arranged in 2D grid not exceeding 65535×65535 in $< x, y >$	Streaming Multiprocessor (**SM**)	**30**
Kernel Grid	Problem or simulation representation	**GPU** Only one kernel is running on the GPU at a time (More are possible, but this is complicated).	**4**

CUDA Warps

- Up to 32 threads execute in a Warp
- Streaming Processor pipeline provides hardware acceleration
- Best performance by if all threads in a warp execute the same instruction

Software Model		Hardware Model	
Element	Maximum	Physical Unit	#
Thread	512 threads per block Arranged in 3D block not exceeding $512 \times 512 \times 64$ in $< x, y, z >$ and 512 total	Scalar Processor (**SP**) or "Streaming **Core**" Each core executes one thread at a time	**8**
Warp	Each 32 threads are statically assigned to a warp	**SP Pipeline** A full warp (32 threads) executes in 4 clock cycles (pipelined 4-deep across 8 cores)	**16**
Block	Arranged in 2D grid not exceeding 65535×65535 in $< x, y >$	Streaming Multiprocessor (**SM**)	**30**
Kernel Grid	Problem or simulation representation	**GPU** Only one kernel is running on the GPU at a time (More are possible, but this is complicated).	**4**

CUDA Blocks

- Grouping of threads into blocks
- Easy access to shared memory
- Thread control and synchronization occurs at this level

Software Model		Hardware Model	
Element	Maximum	Physical Unit	#
Thread	512 threads per block Arranged in 3D block not exceeding $512 \times 512 \times 64$ in $< x, y, z >$ and 512 total	Scalar Processor (**SP**) or "Streaming **Core**" Each core executes one thread at a time	**8**
Warp	Each 32 threads are statically assigned to a warp	**SP Pipeline** A full warp (32 threads) executes in 4 clock cycles (pipelined 4-deep across 8 cores)	**16**
Block	Arranged in 2D grid not exceeding 65535×65535 in $< x, y >$	Streaming Multiprocessor (**SM**)	**30**
Kernel Grid	Problem or simulation representation	**GPU** Only one kernel is running on the GPU at a time (More are possible, but this is complicated).	**4**

CUDA Kernels

- Kernels are the main structure of CUDA device code
- Execute on GPU
- Representations of the overall problem structure

Software Model		Hardware Model	
Element	Maximum	Physical Unit	#
Thread	512 threads per block Arranged in 3D block not exceeding $512 \times 512 \times 64$ in $< x, y, z >$ and 512 total	Scalar Processor (**SP**) or "Streaming **Core**" Each core executes one thread at a time	**8**
Warp	Each 32 threads are statically assigned to a warp	**SP Pipeline** A full warp (32 threads) executes in 4 clock cycles (pipelined 4-deep across 8 cores)	**16**
Block	Arranged in 2D grid not exceeding 65535×65535 in $< x, y >$	Streaming Multiprocessor (**SM**)	**30**
Kernel Grid	Problem or simulation representation	**GPU** Only one kernel is running on the GPU at a time (More are possible, but this is complicated).	**4**

Sample CUDA Code

- Standard C code with special syntax to call parallel kernels

```
// Device code to set up the kernel

/* Define the geometry of the parallel solver for the Kernels */
dim3  grid(    BLK_MAX_X,    BLK_MAX_Y, BLK_MAX_Z);
dim3  threads( COLS_PER_BLK, 1,         1        );

/* Allocate memory for the GPU */
cudaMalloc( (void**) &wavefield0, wavefield_size);
cudaMalloc( (void**) &wavefield1, wavefield_size);

/* Call the kernel in parallel with the desired geometry      */
waveprop_kernel<<<grid, threads, 0>>>(
                                    wavefield0,
                                    wavefield1,
                                    velocity_model
                                    );
```

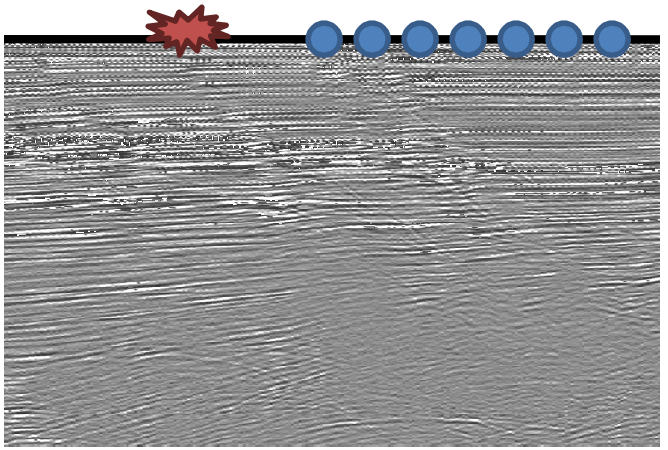
CUDA Summary

- Simple extensions to standard C
- Treat the GPU as a massively parallel coprocessor with its own memory

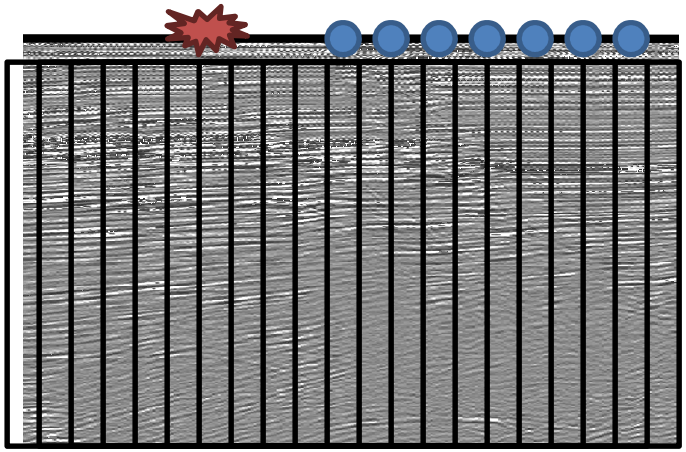
- Best performance:
  - Structure the problem for data-parallelism
  - Spread work simultaneously to thousands of threads

GPGPU Wave Propagation Parallelism

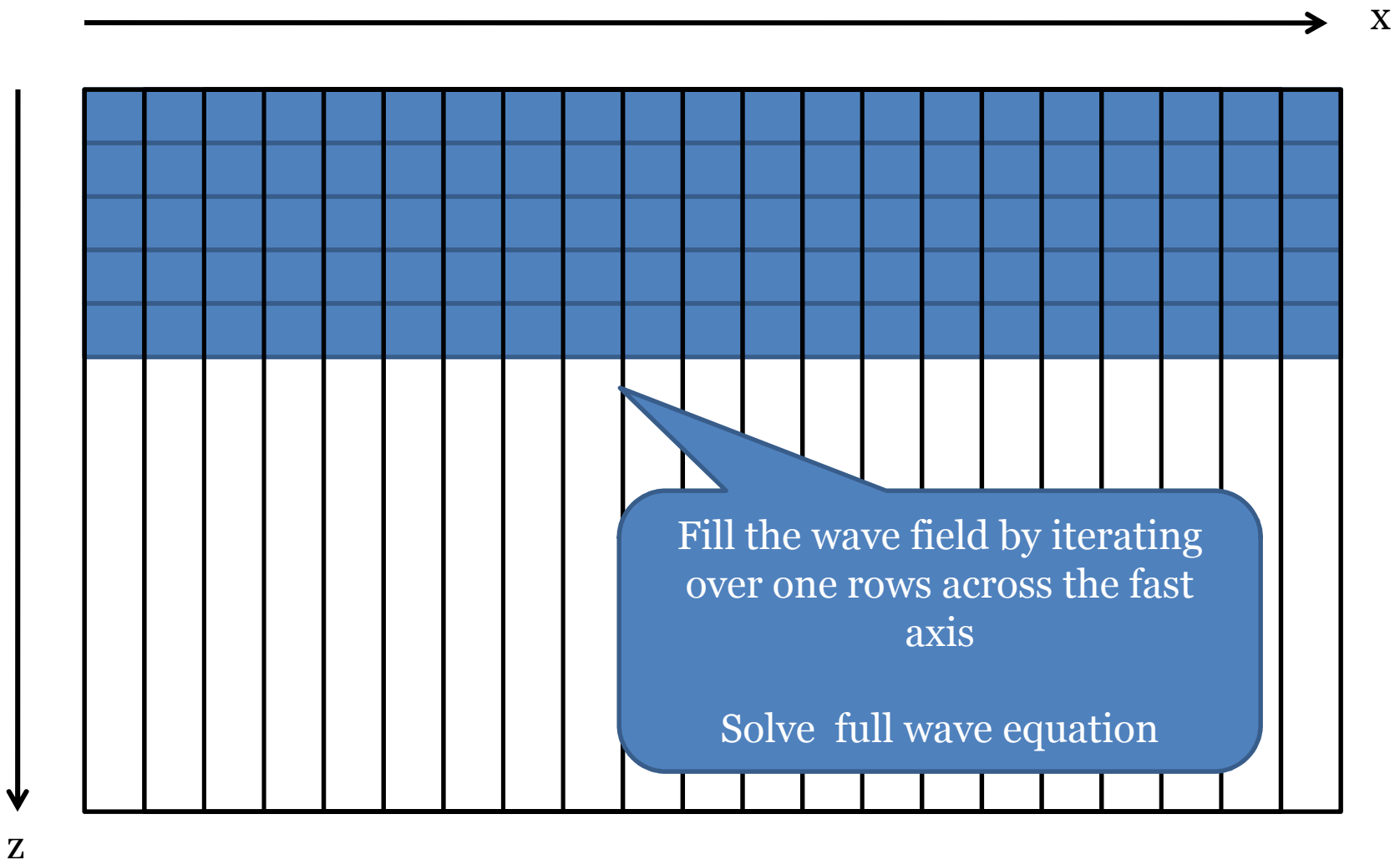
- Current approach:
- 1 thread for receiver or source



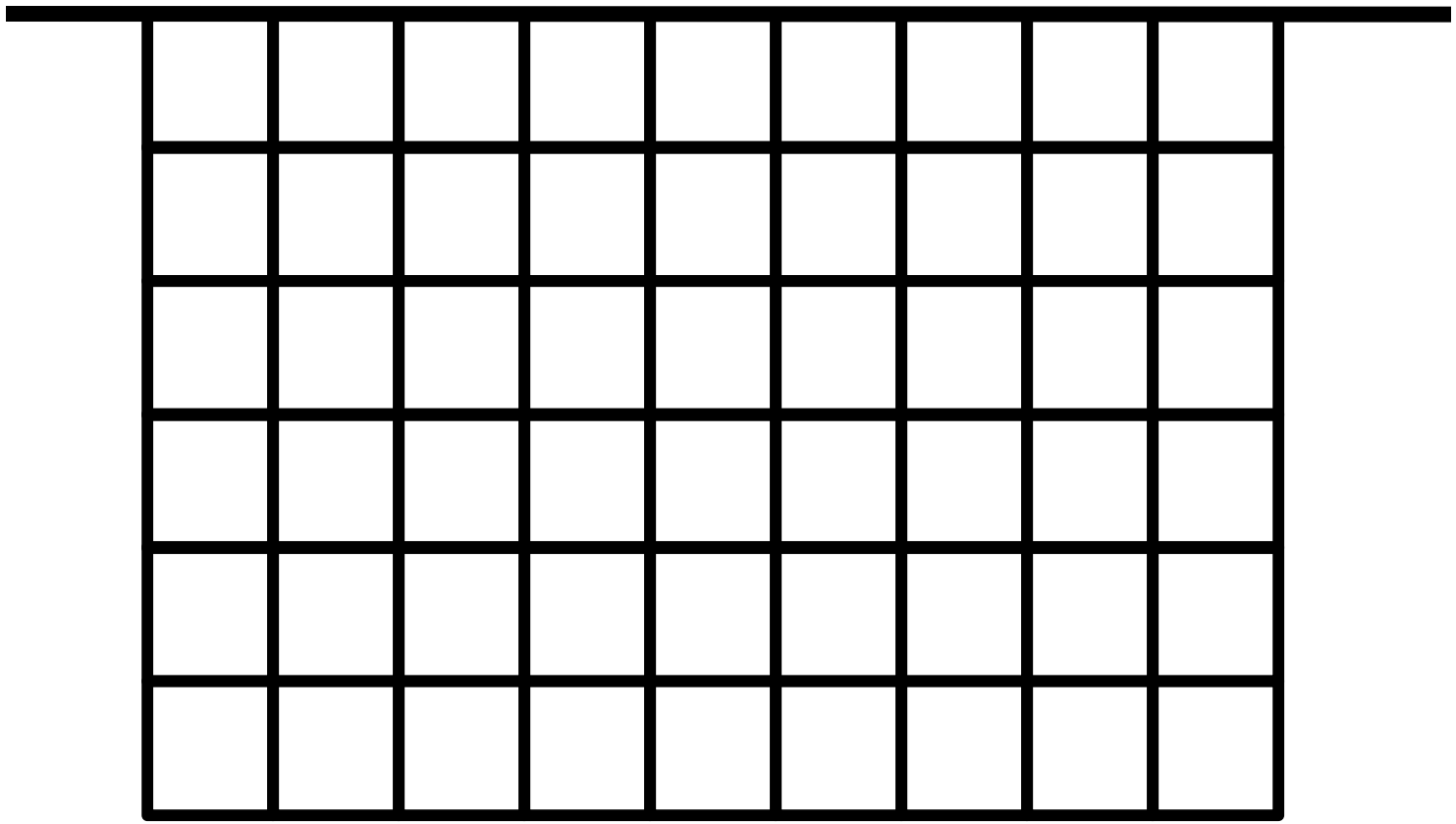
- New grid approach:
  - 1 thread for every grid point
  - 1 thread per column, in 2D wave equation



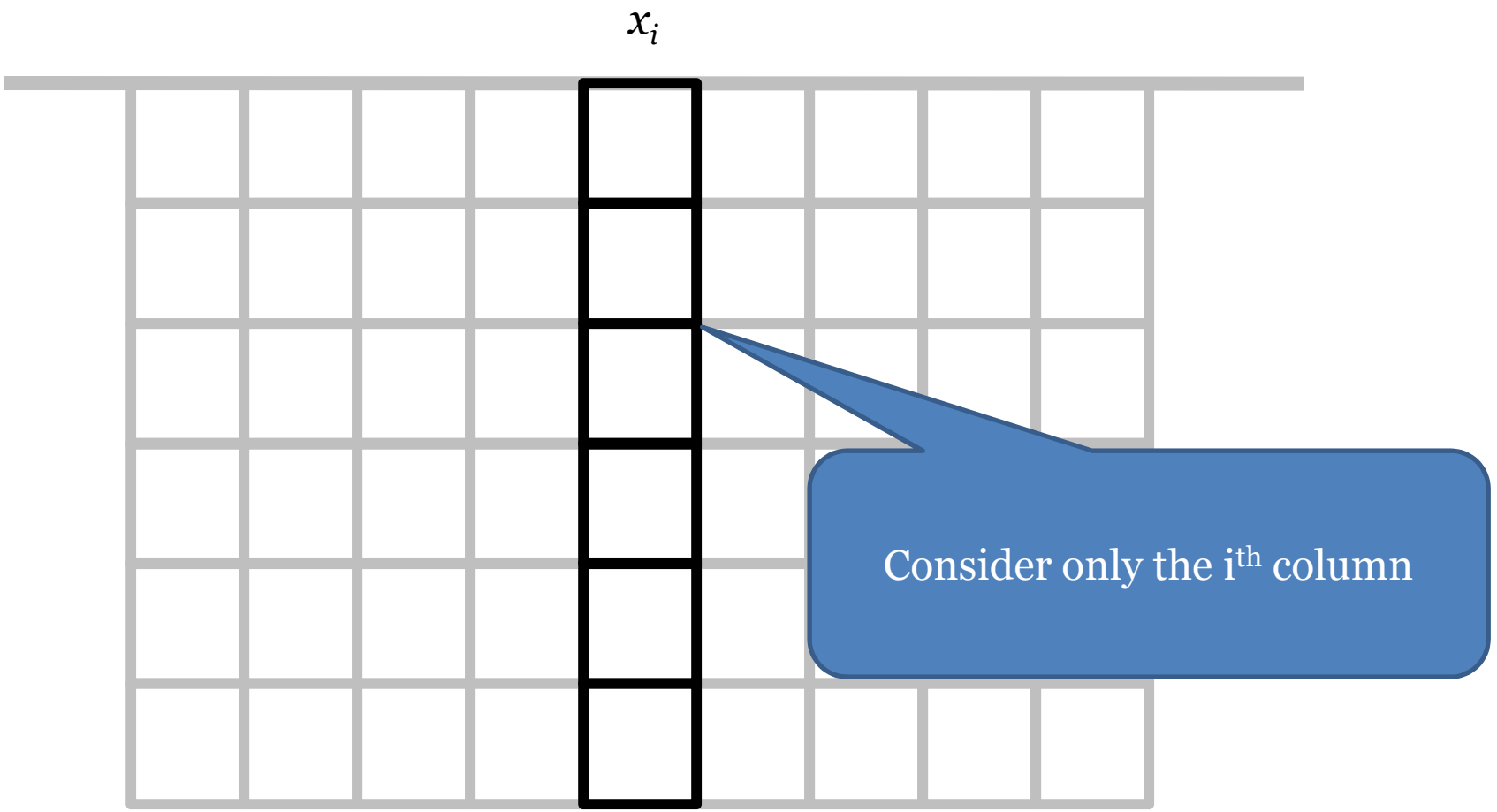
Grid Parallelism Macro-Algorithm



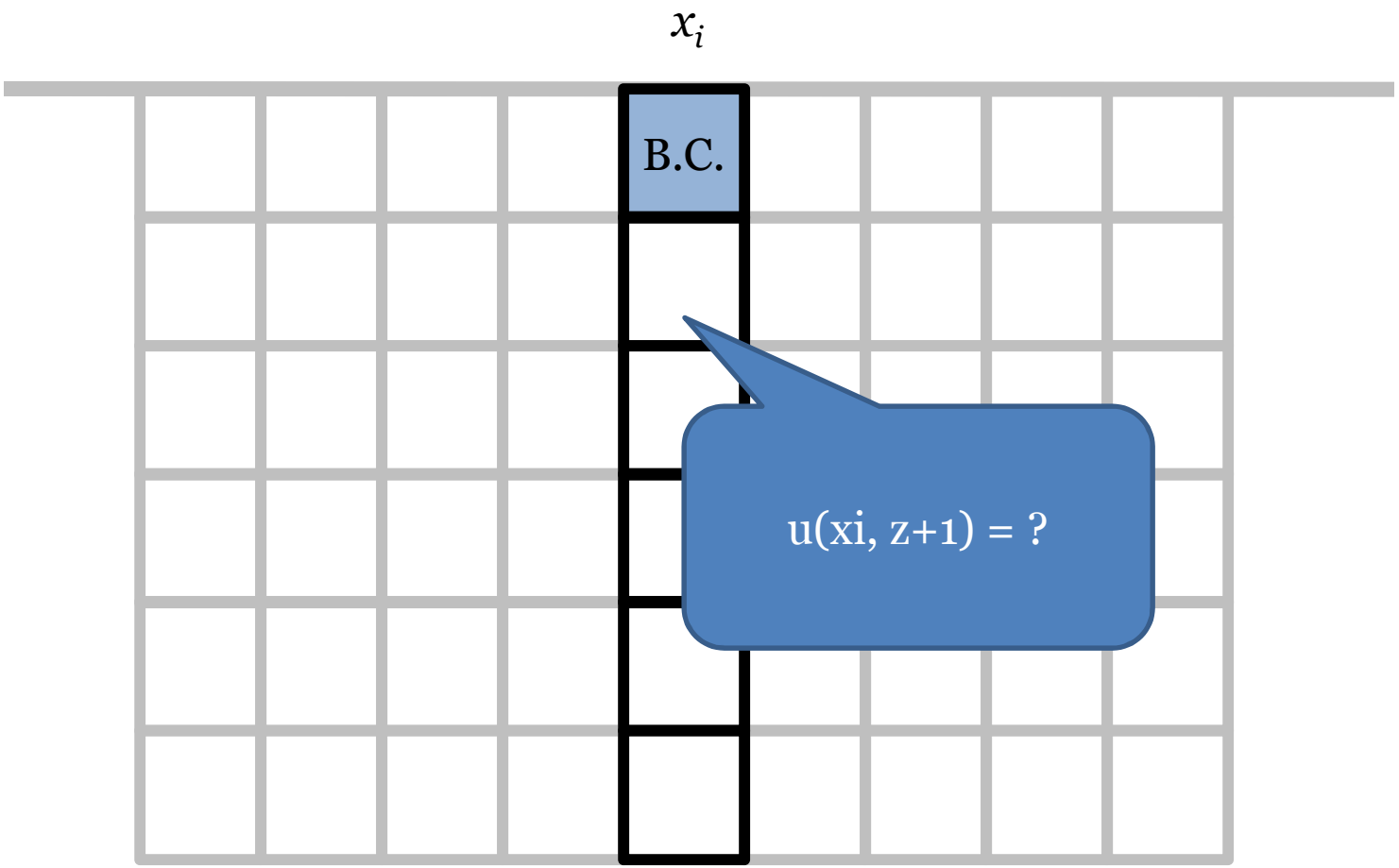
Grid Parallelism – Micro-Algorithm (Column kernel)



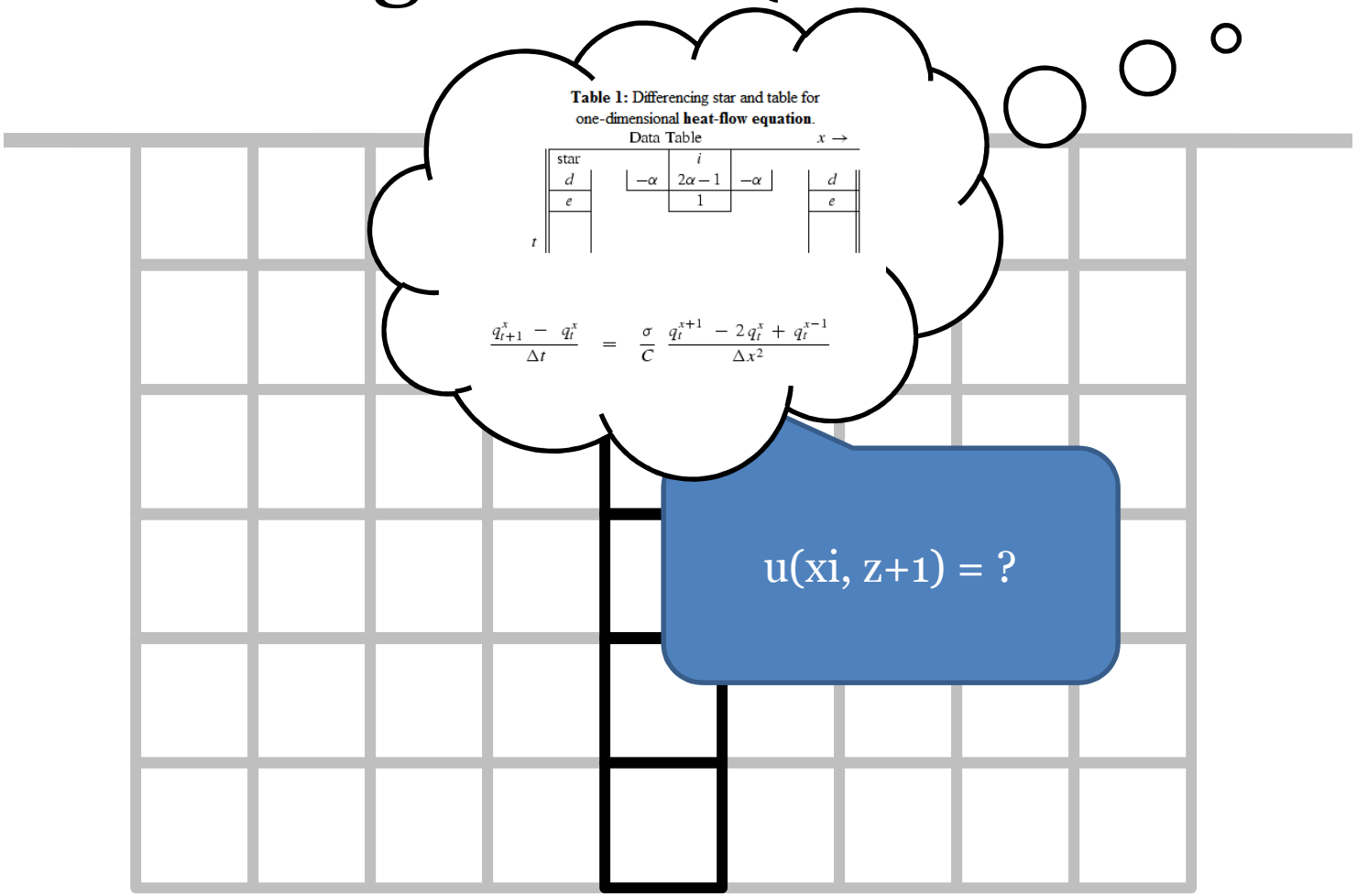
Grid Parallelism – Micro-Algorithm (Column kernel)



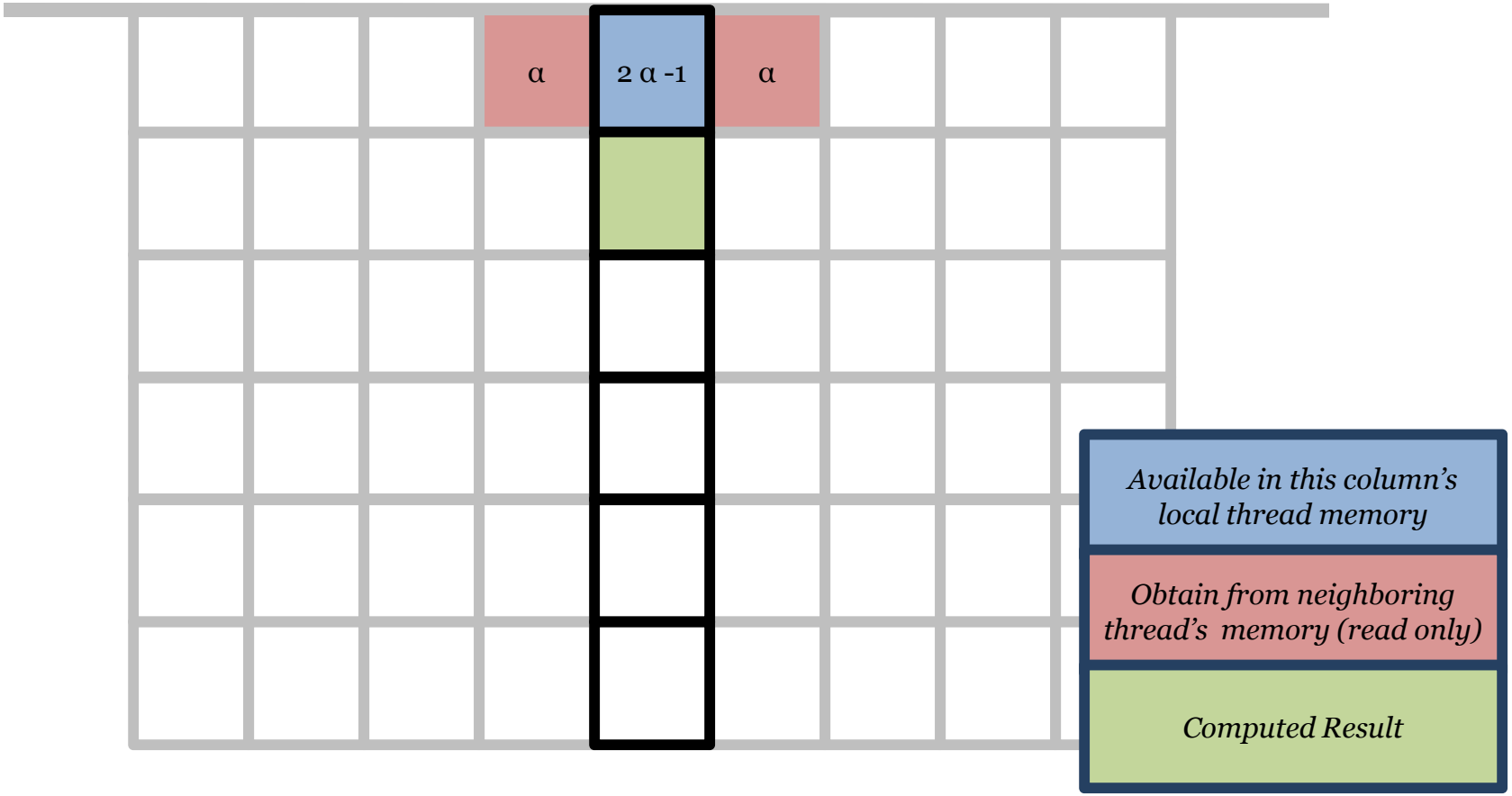
Grid Parallelism – Micro-Algorithm (Column kernel)



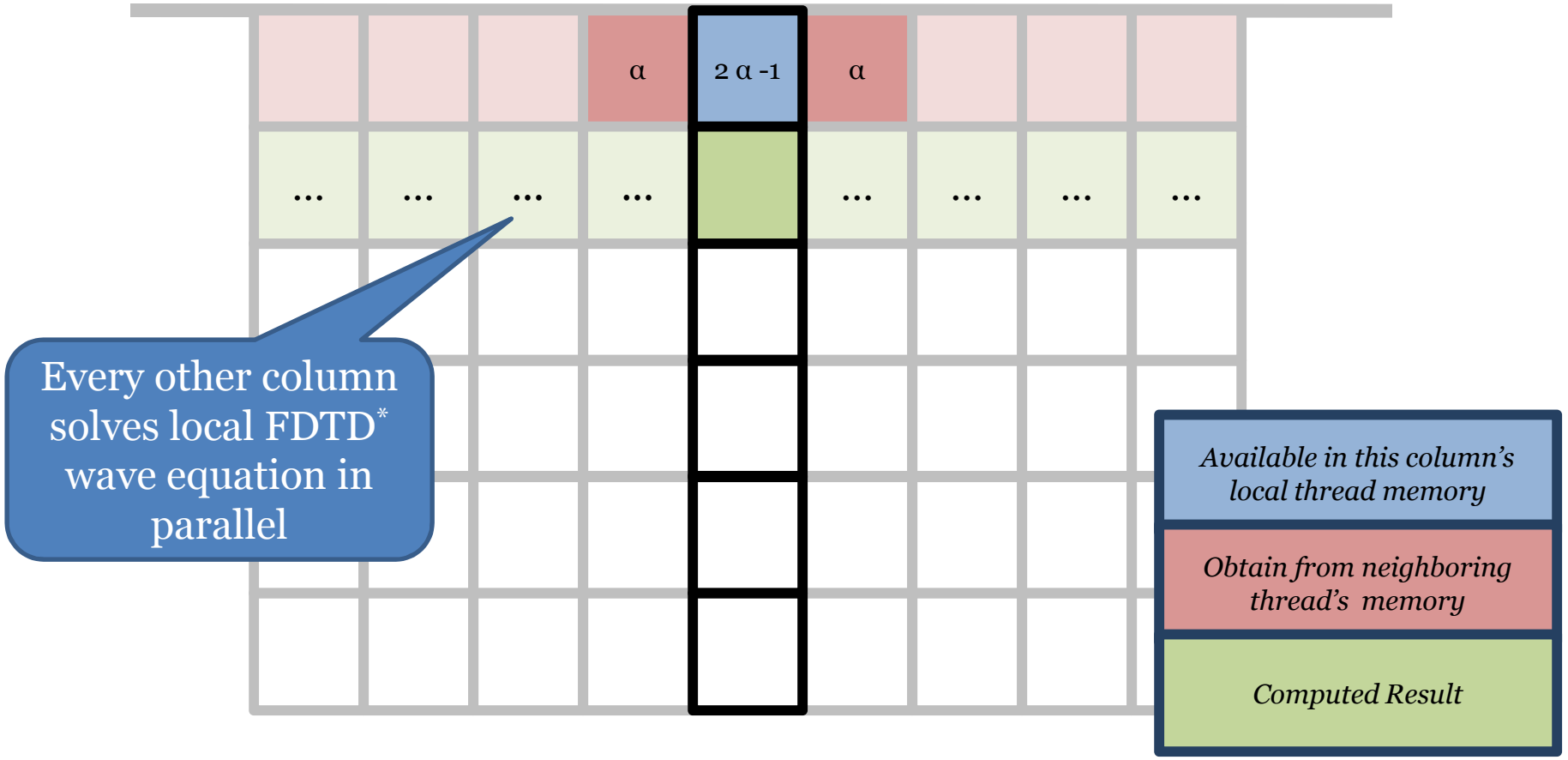
Grid Parallelism 3 – Micro-Algorithm (Column kernel)



Grid Parallelism 3 – Micro-Algorithm (Column kernel)



Grid Parallelism 3 – Micro-Algorithm (Column kernel)



Grid Parallelism 4 - Overhead

- Non-trivial communication stall to share data laterally between columns

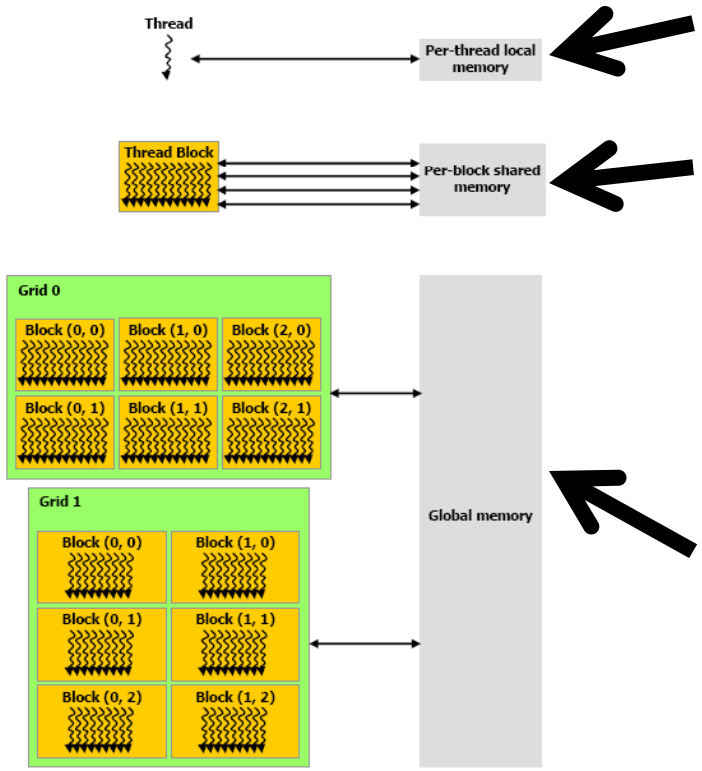
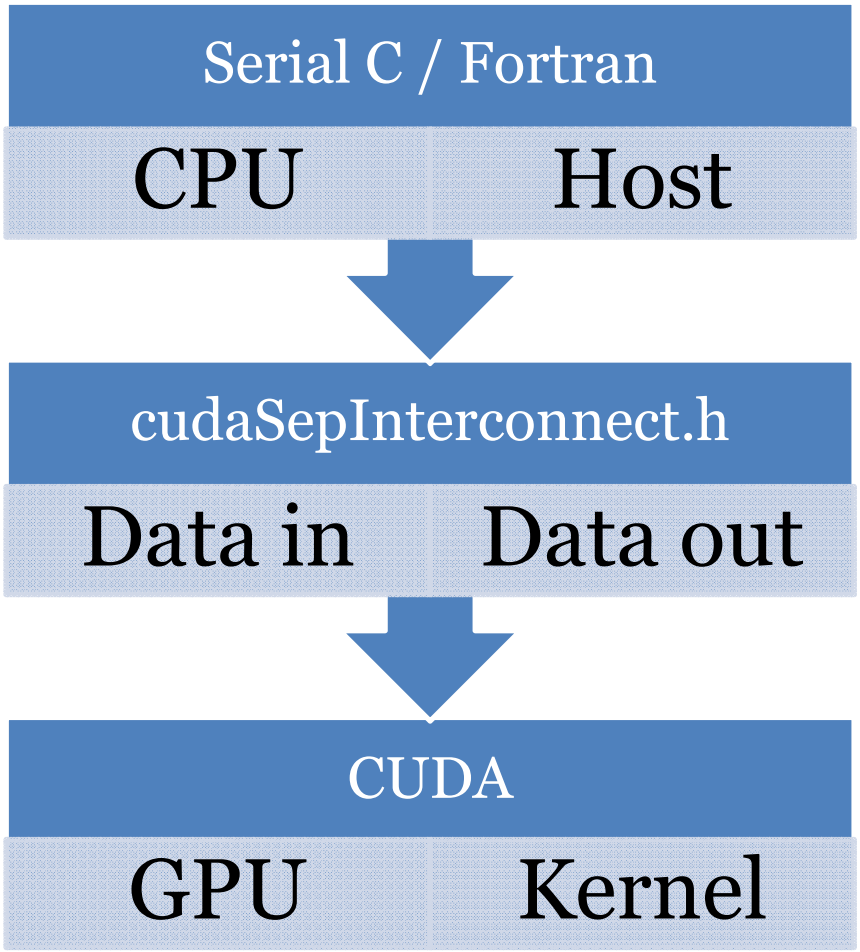


Figure 2-2. Memory Hierarchy

- Single column
- Neighboring columns sharing data in block shared memory
- Edge of thread-block – special boundary case
- Global data for longer-range lateral sharing

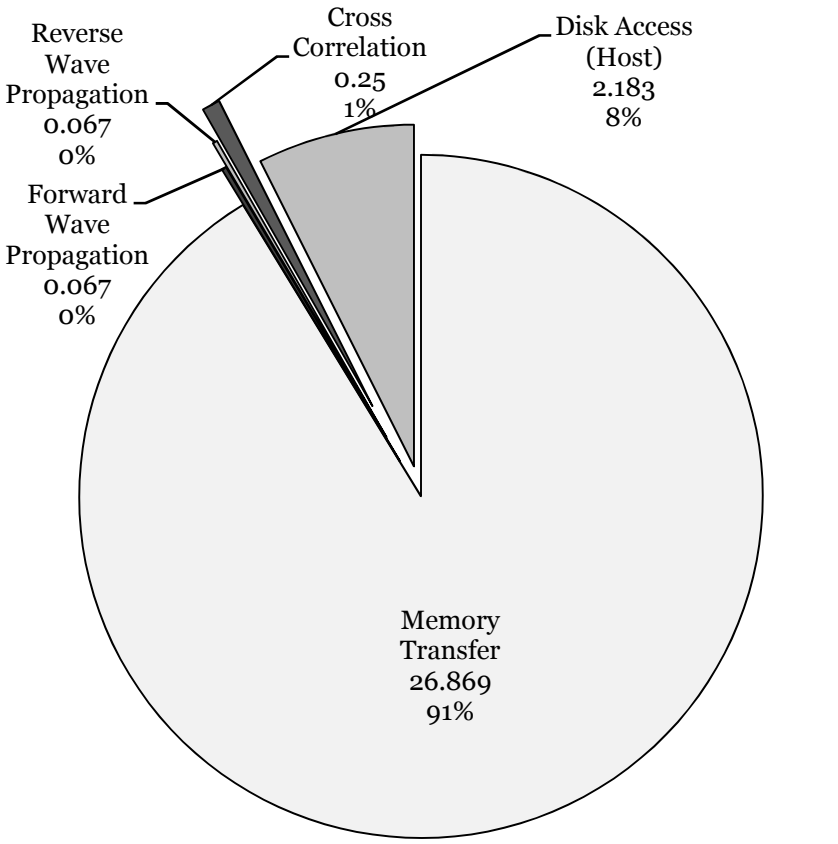
CUDA and SEPlib

- Loading and writing header (`.H`) files for data I/O
- Designing an API for wrapping CUDA accelerated kernels
- Standardized interchange format for arrays of velocity, wavefield, & image data



Benchmarking Progress

- 2D, 1000x1000 cell
- Forward wave
- Reverse wave
- Imaging (correlation)

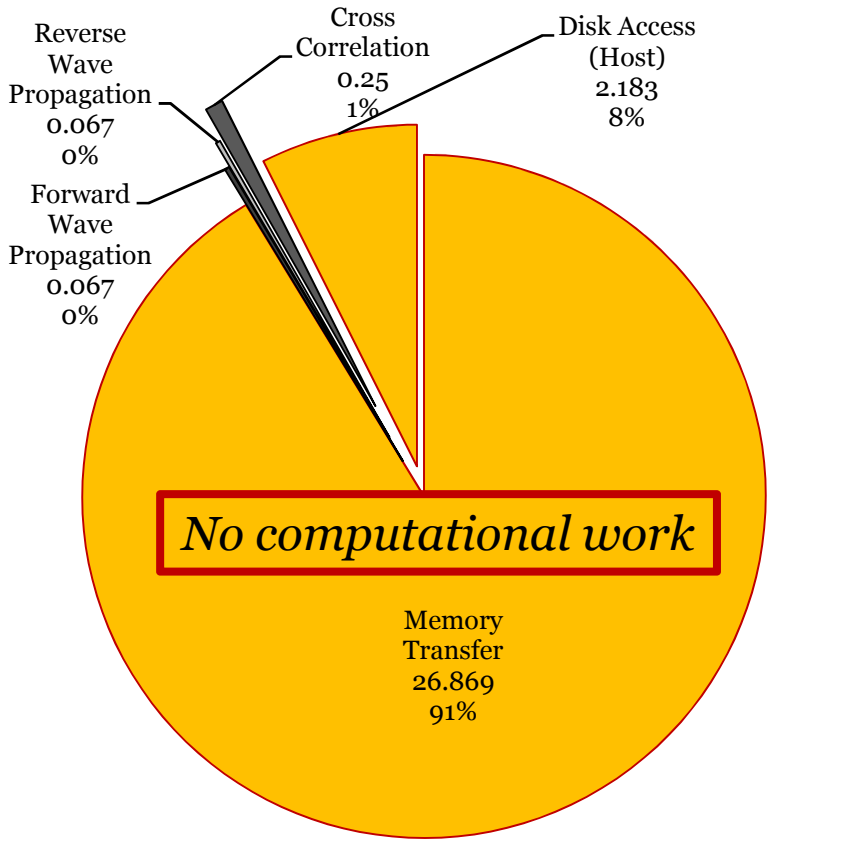


Execution Time (seconds)

Benchmarking Progress

- 2D, 1000x1000 cell
- Forward wave
- Reverse wave
- Imaging (correlation)

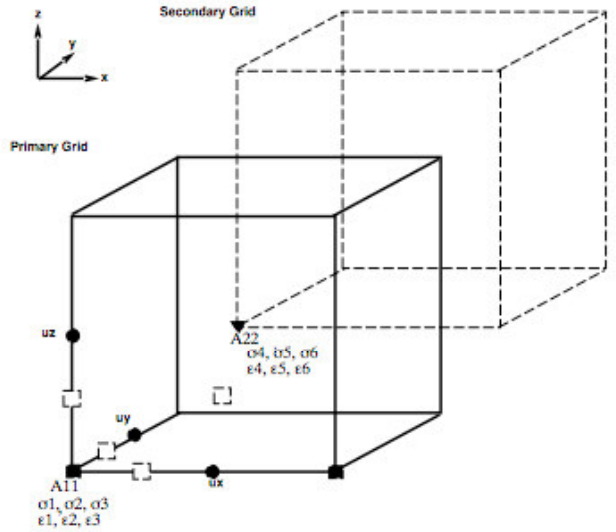
How do we fix this?



Execution Time (seconds)

Algorithm Ideas and GPGPUs

- Elastic Wave Equation
  - Necessary for complete inversion of multi-component data
- Staggered Mesh[1]
  - Eliminate interpolation with proper alignment
  - Meet stability criteria with coarser grid-spacing
- Yee Cell FDTD[2] :
  - staggered-spatial grid
  - staggered-time steps
  - reduce data dependency



1. Martin Karrenbach. *SEP-83: Elastic Tensor Wave Fields.* (§5.6.3 – Practical issues). 1995.
2. Inan, U.S. *Yee cell representation*. Numerical Electromagnetics. 2005.

Future developments

- Attacking the bottleneck
  - Data compression
  - Creative ways to keep data on the GPU
    - Diffuse-reflecting boundary conditions
    - Checkpointing
- Abstracting the acceleration hardware
  - Generalized stencil library
  - Comparison with alternative accelerators

Conclusion

- There is much power to be exploited using "exotic" computer architectures
- SEP (and seismic imaging community) must evaluate the return-on-investment, because the engineering and research overhead (NRE) may be high
- Preliminary GPGPU results are very promising, and there is still room for dramatic improvement