GPGPU LAB

Case study: Finite-Difference Time-Domain Method on CUDA

Ana Balevic

Finite-Difference Time-Domain Method

- Numerical computation of solutions to partial differential equations
- Explicit E-Field update (wave) equation:

$$E_y(nx,nz,nt+1) = 2\left[1-2(\Delta t)^2\right]E_y(nx,nz,nt) - E_y(nx,nz,nt-1)$$
$$+(\Delta t)^2\left[E_y(nx+1,nz,nt)+E_y(nx,nz+1,nt)+E_y(nx,nz-1,nt)+E_y(nx-1,nz,nt)\right]$$
$$+\Delta t\left[J_{ey}(nx,nz,nt) - J_{ey}(nx,nz,nt-1)\right].$$

nx,nz,nt ~ space and time coordinates
Δt ~ constant time step
E_y ~ electric field
J_{ey} ~ excitation

Pseudocode:

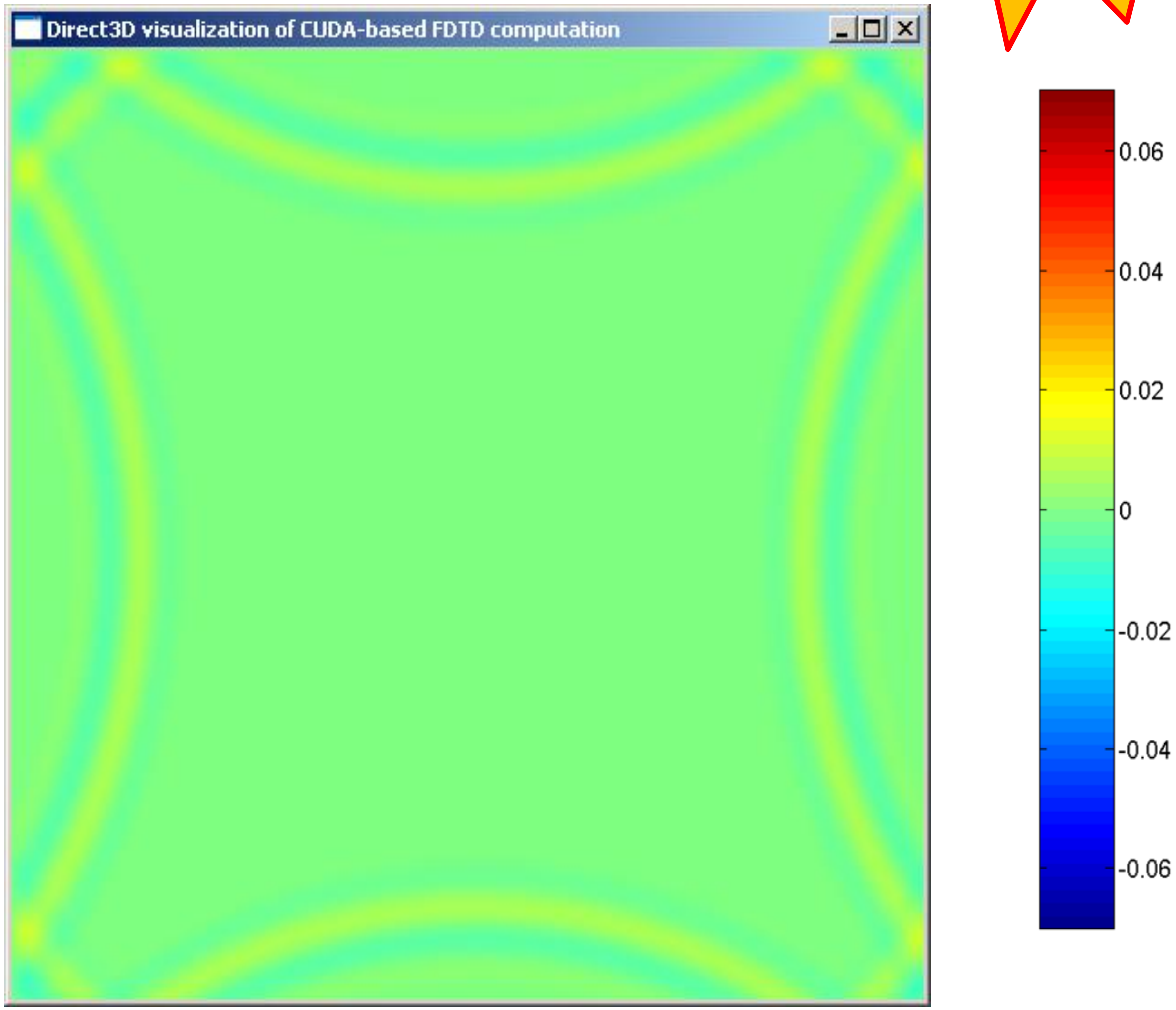
```
for nt=1 to NT do
    for nx = 1 to NX do
        for nz = 1 to NZ do
            Wave Equation
            Apply Excitation
            Apply Boundary Condition
        end
    end
end
```

Suitable for parallel processing across spatial domain!

Visual Demonstration

Visualization programmed by Tjark!

- Grid size: 256 x 256
- Time steps: 260

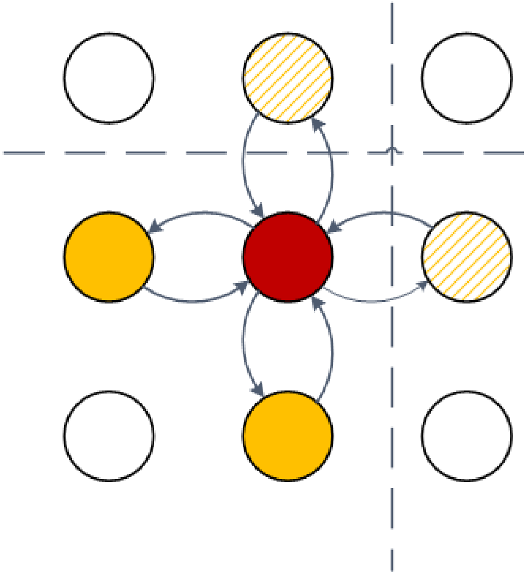


Issues: Neighborhood Operations

- Mapping: data element – processing thread

- Data partitioning causing dependencies of data blocks:
    - Cells on the boundary of each data block are used for the computation by the neighboring thread block

    - Avoid RAW data hazard (design to avoid race conditions!):
        - must exchange values of boundary block cells w. neighboring thread blocks between time iterations

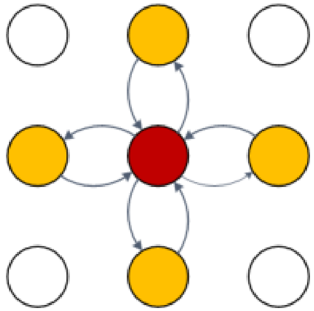


Issues: Neighborhood Operations

- GPGPU Architecture limitation:
  - No message passing
  - Shared Memory – Yes, but exclusive partition for each thread block
  - Synchronization:
    - Barrier synchronization on the thread block level
    - No synchronization mechanism on the grid level
    - Requires synchronization between time steps by terminating and again launching kernel on device, and overlapping loads of block boundary cells

Finite-Difference – Mapping to GPGPU cont'd

- FDTD: Inherent data dependencies



1. Flow control instructions (if, switch, do, for , while) impact the effective instruction throughput by causing threads of the same warp* to diverge => serialized execution.
2. Avoidable by different memory access patterns => inefficient?

⇨ Explore design space and compare tradeoffs:
branching vs. memory access patterns

Note: warp = set of threads = scheduling unit on GPU

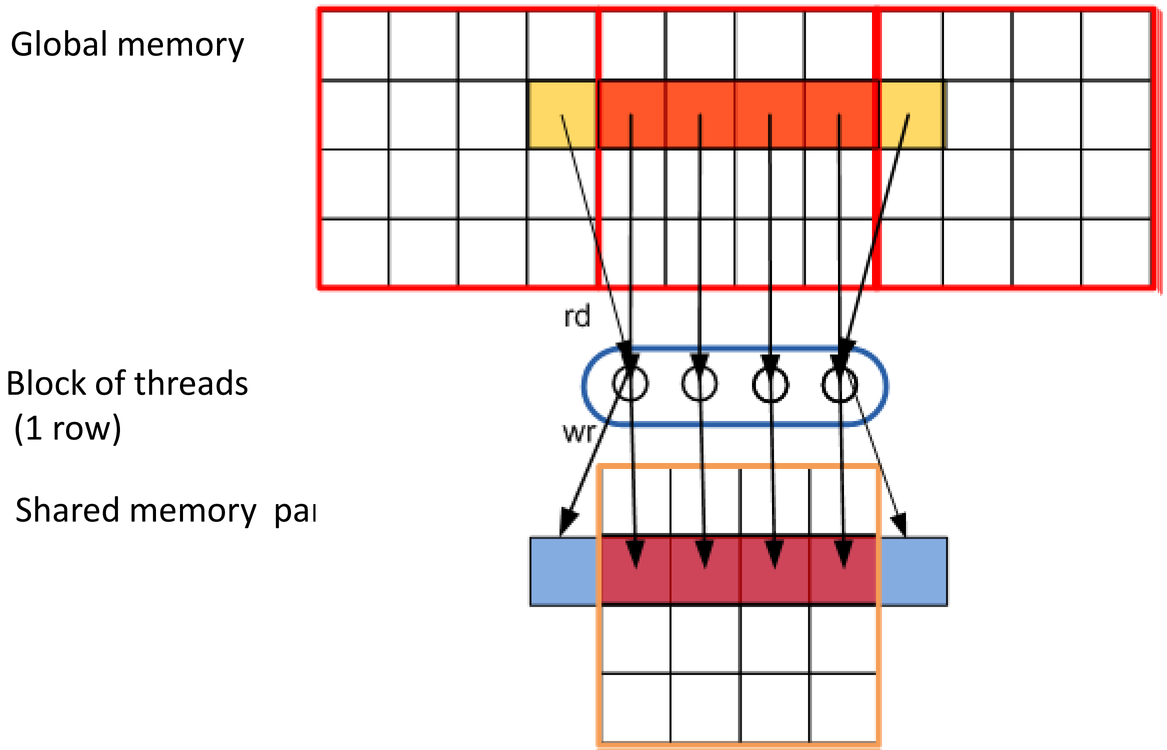
Effects of Memory Access Patterns

- Mode 1:
  - Additional loads per thread for fetching elements from the boundary of neighboring blocks
  - e.g. 16x16 Data Block => 16x16 Thread Block
  - Requires branching logic

- Mode 2:
  - Additional threads for fetching elements from the boundary of neighboring blocks
  - e.g. 16x16 Data Block => 18x18 Thread Block
  - No branching logic, but unaligned memory access

FDTD Computation: Mode 1

- Simple example: 1 row of the surface containing 4x12 cells:



1. Partitioning of simulation data into data blocks (in the GPU global memory)

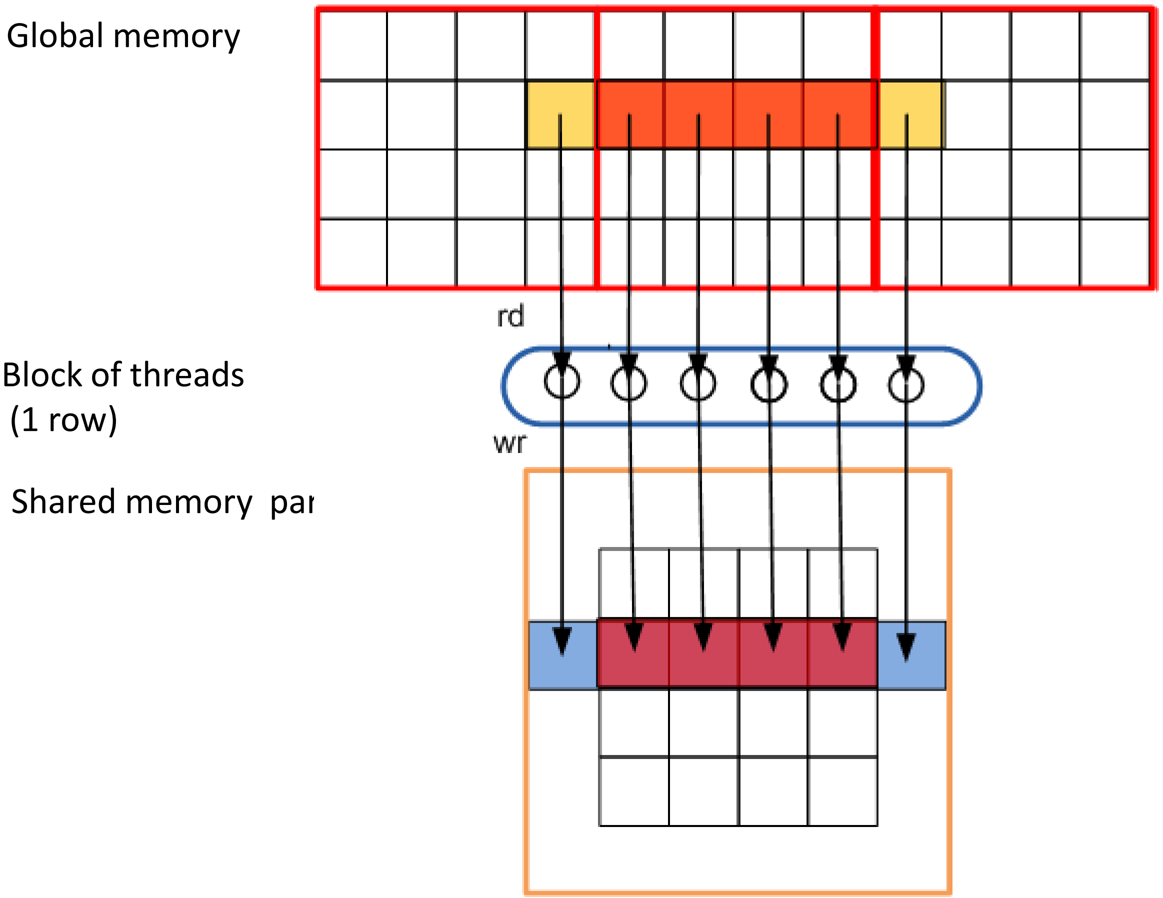
2. Parallel load of data blocks into the shared memory (1 or more loads/thread)

3. Boundary threads perform additional loads from the global memory: Missing neighboring data into registers.

4. Computation (and storage) of new values by block threads in parallel (all working).

FDTD Computation: Mode 2

- Example: 1 row of the surface containing 4x12 cells:



1. Partitioning of simulation data into data blocks (in the GPU global memory)

2. Parallel load of data blocks into the shared memory (only 1 load/thr)

3.Additional threads load neighboring data into shared memory. (more threads, larger SM partition required)

4. Parallel computation of new values by threads (only inner threads working).

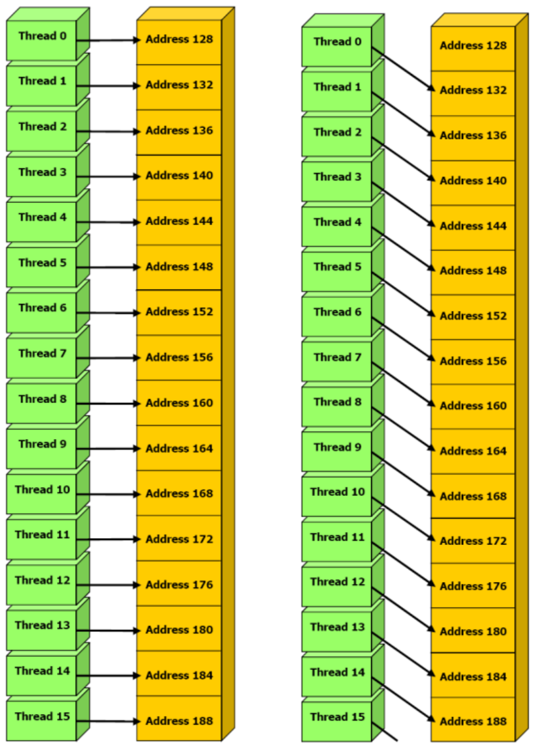
Analysis: Coalesced Memory Accesses

- The global memory space is not cached and memory latency high => important to follow the right access pattern (coalesced access) to get maximum memory bandwidth

- ## The coalesced global memory access conditions:

1. Threads must access 32-bit words, resulting in one 64-byte memory transaction
2. All 16 words must lie in the same segment of size equal to the memory transaction size
3. Threads must access the words in sequence: The *kth thread in the half-warp must* access the *kth word.*

- Otherwise, a separate memory transaction is issued for each thread. Order of magnitude lower bandwidth for uncolaesced access on single-precision floats!



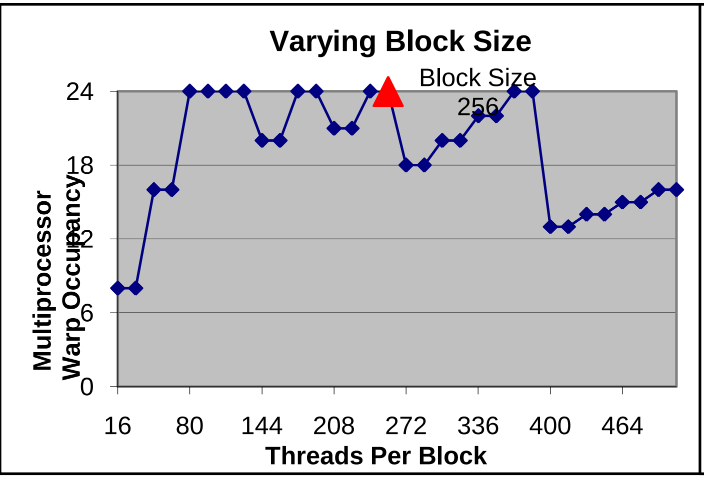
coalesced uncoalesced

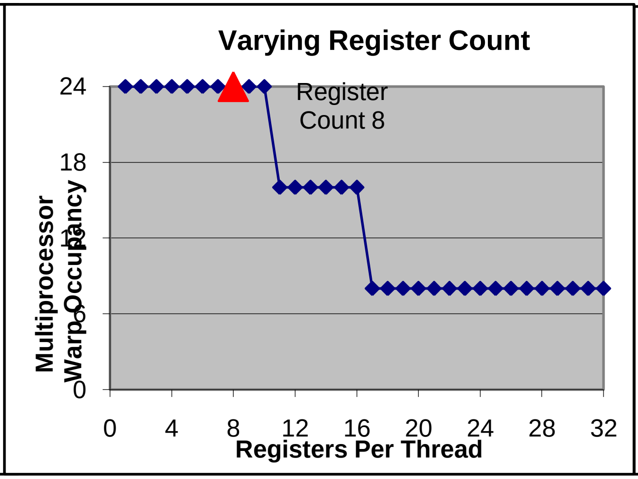
Conclusion of experiments: branches have less impact on the kernel performance than uncoalesced memory accesses! **Optimize memory accesses first!**

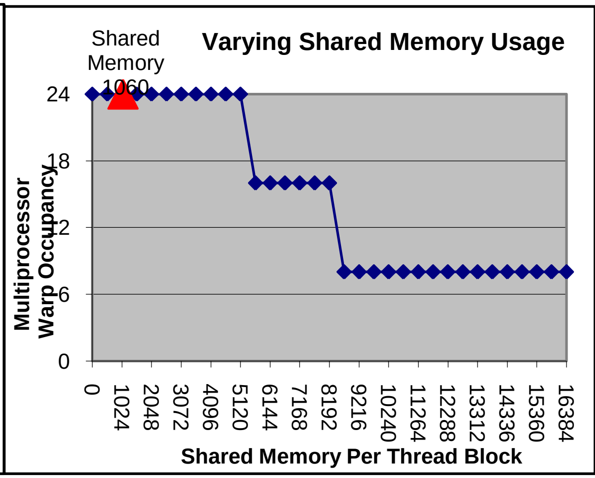
Multiprocessor Utilization

Goal: maximize utilization of the GPGPU multiprocessors

Design space: underlying hardware architecture, kernel configuration parameters, memory footprint of the kernel



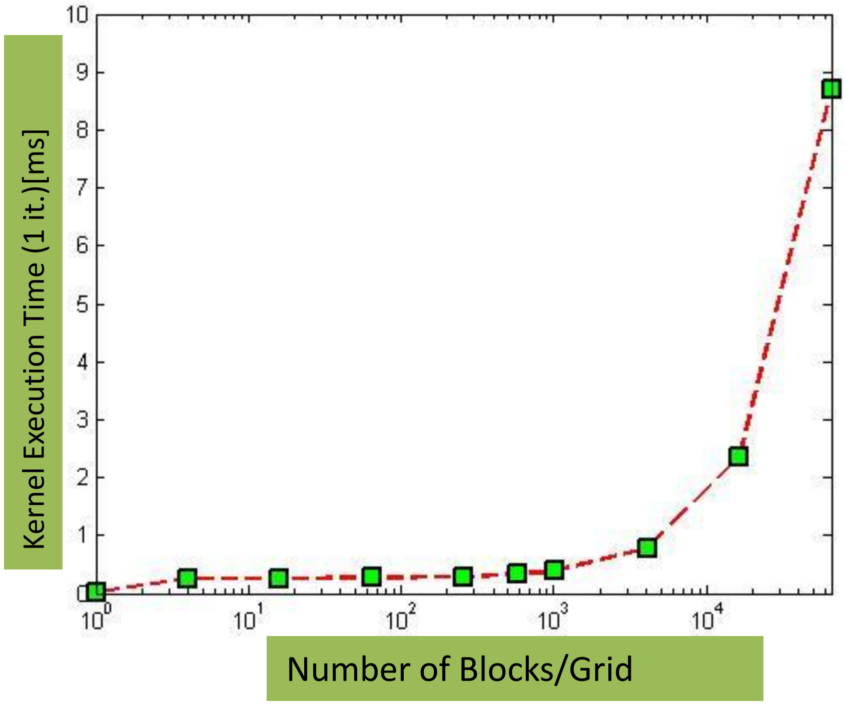


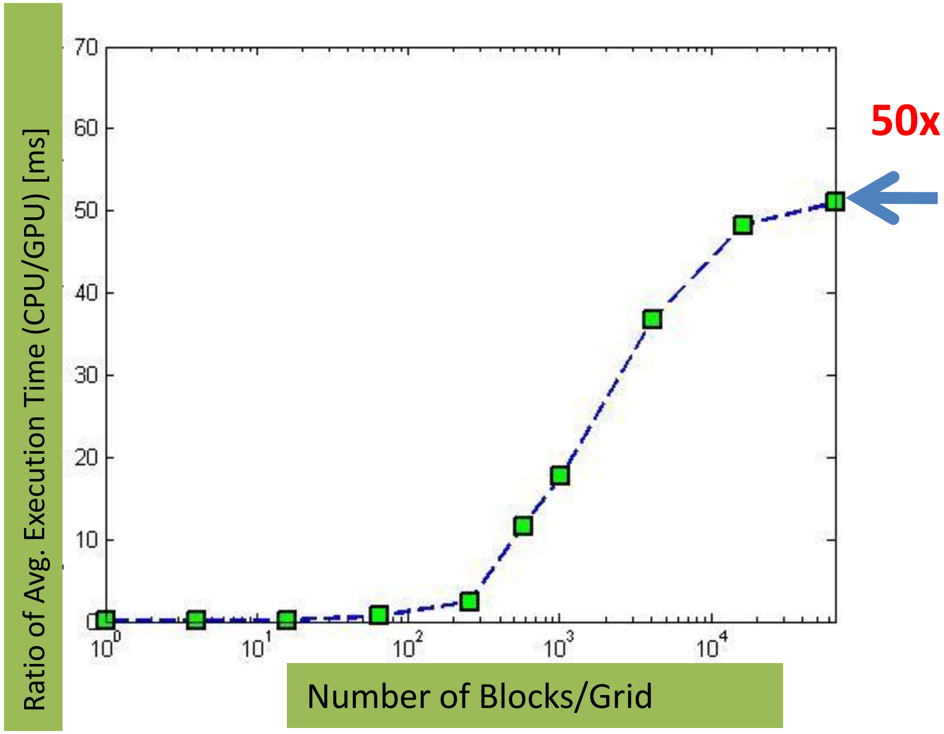


Resource Utilization:	
Threads Per Block	256
Registers Per Thread	8
Shared Memory Per Block [B]	1060

GPU Occupancy Data	
Active Threads per Multiprocessor	**768**
Active Warps per Multiprocessor	**24**
Active Thread Blocks per Multiprocessor	**3**
Occupancy of each Multiprocessor	**100%**
Maximum Simultaneous Blocks per GPU	**48**

FDTD Computation on GPGPU: Performance Results





Surface Size (Cells)	Grid Size (Blocks)	GPU (ms)	Data Transf. (ms)	CPU (ms)	Ratio CPU /GPU
1048576	64x64	0.78	5.71	28.61	36.68
4194304	128x128	2.36	19.44	113.89	48.26
16777216	256x256	8.69	68.95	443.65	51.05

Direct3D visualization of CUDA-assisted scientific calculations

Introduction

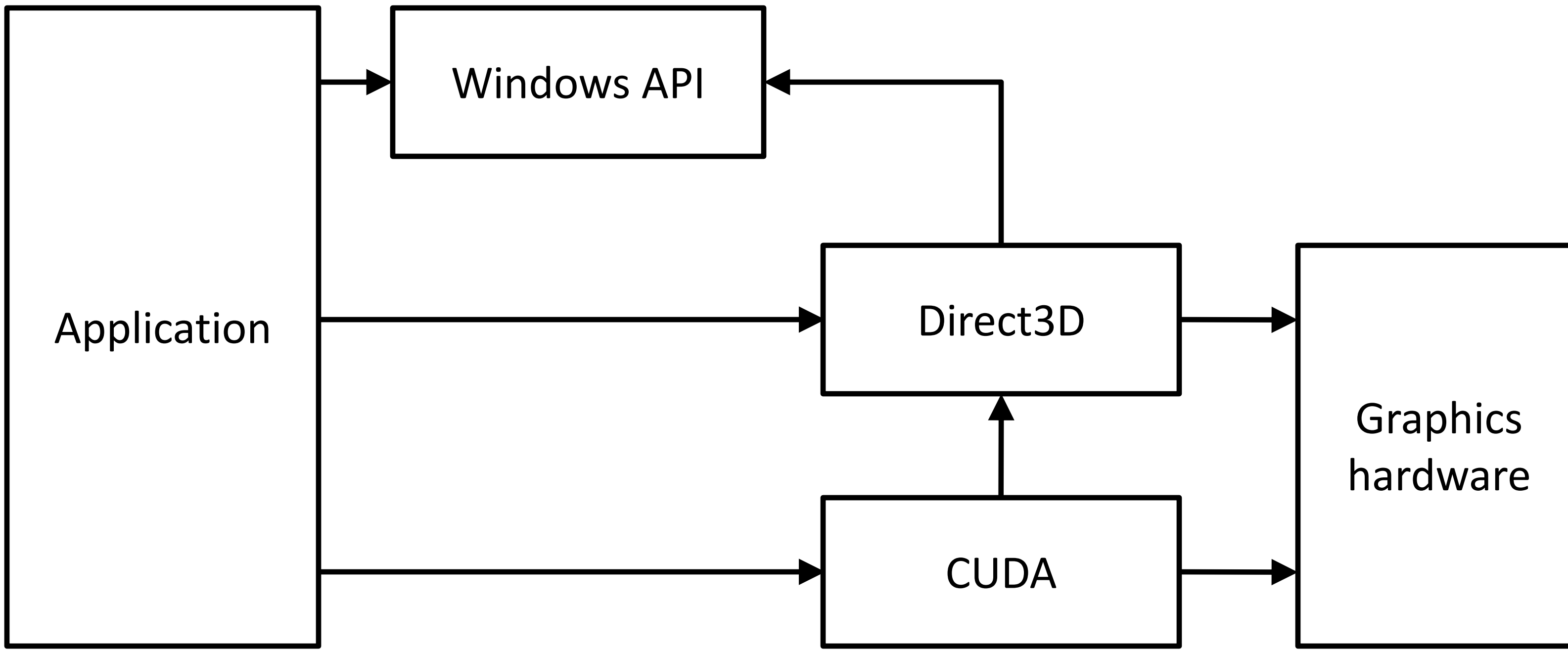
- CUDA – allows compute-intensive tasks to be off-loaded onto the GPU

- Goal: since all data are already in video memory, display them on the fly by making use of the GPU's traditional rendering capabilities

➢ Requires CUDA to interact with a graphics API (Direct3D, OpenGL)

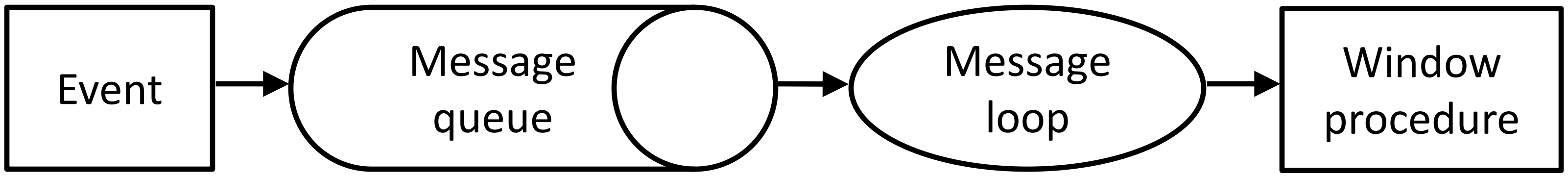
CUDA-Direct3D Interaction

- Framework interconnection:



Windows API: Event Model

- Workflow:



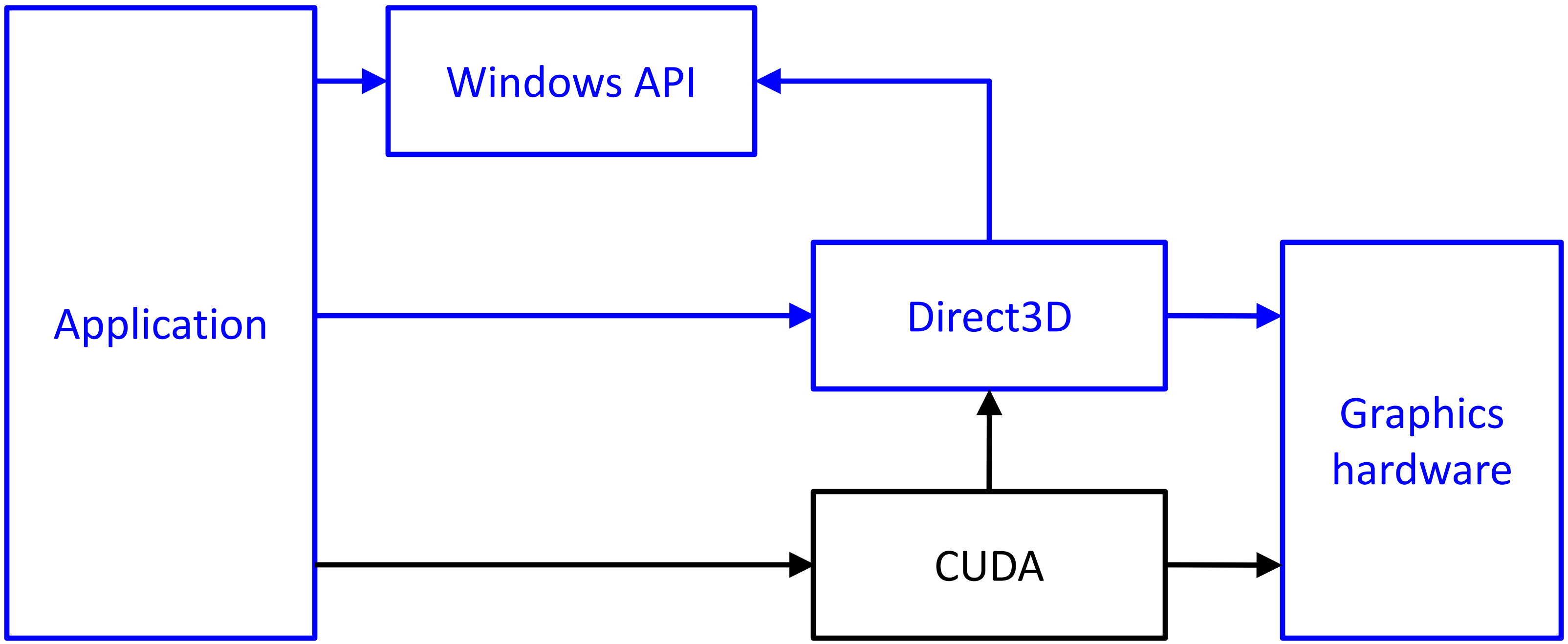
- Event occurs
- Windows sends a message to the application the event occurred for
- Message is added to the application's message queue
- Application constantly checks its message queue in a message loop
- If it receives a message, it dispatches it to the window procedure of the particular window the message is for

Windows API: A Minimal Application

- Required components and steps:
  - WinMain(args) function
    - Register a window class
    - Create a window based on the newly registered class
    - Show the window
    - Enter the message loop
  - Window procedure
    - Handle selected events
    - Pass unhandled events to a default window procedure

CUDA-Direct3D Interaction

- Framework interconnection:



Direct3D: Prerequisites

- Download and install the DirectX SDK

  http://msdn.microsoft.com/en-us/directx/default.aspx

- Add d3d9.lib and d3dx9.lib to the linker input files

  Project > Properties > Linker > Input > Additional Dependencies when working with Visual Studio

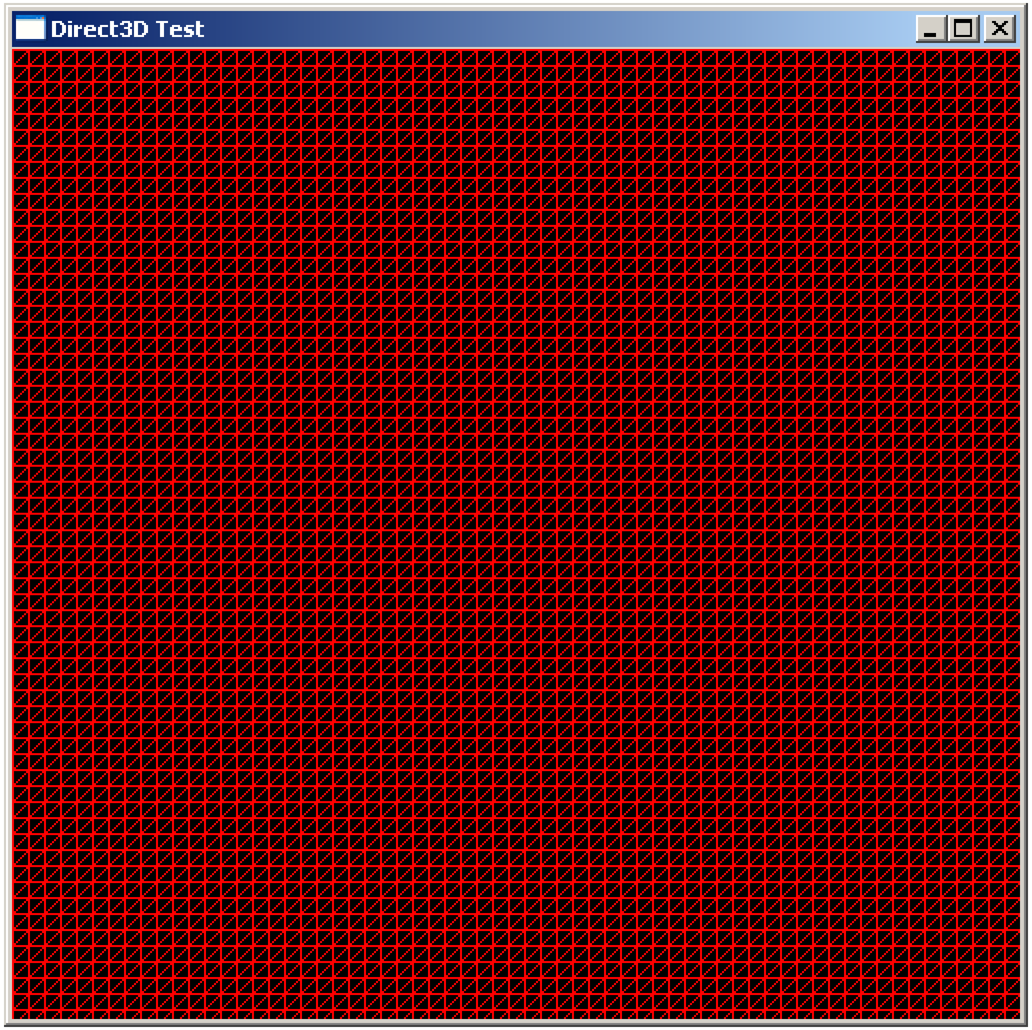
- Update include and library search paths

  should not be necessary when working with Visual Studio

Direct3D: A Minimal Application

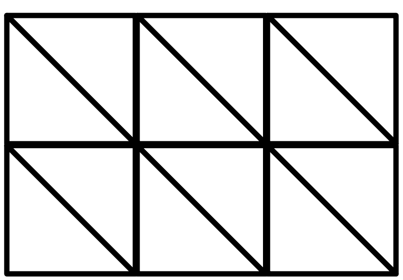
- WinMain(args) function
  - Register a window class
  - Create a window based on the newly registered class
  - Call InitD3D(args)
  - Call InitGeometry()
  - Show the window
  - Enter the message loop calling Render() as idle function
- InitD3D(args) function
  - Create a Direct3D context and associate it with the newly created window
- InitGeometry() function
  - Create the geometry that we want to display
- Render() function
  - Render the afore-created geometry

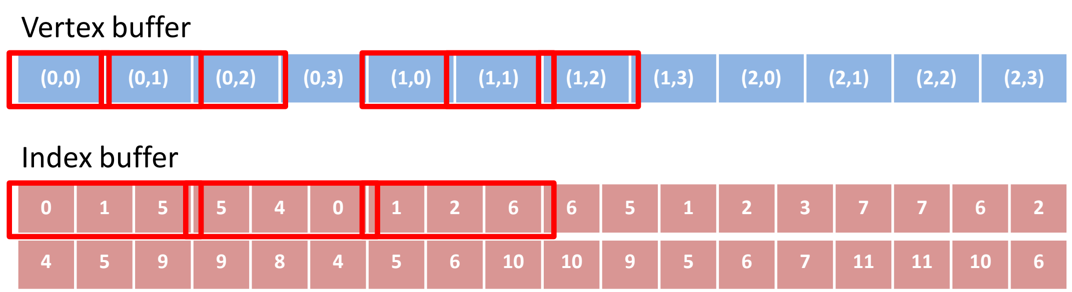
Direct3D: A Minimal Application (2)



Direct3D: Vertex and Index Buffers

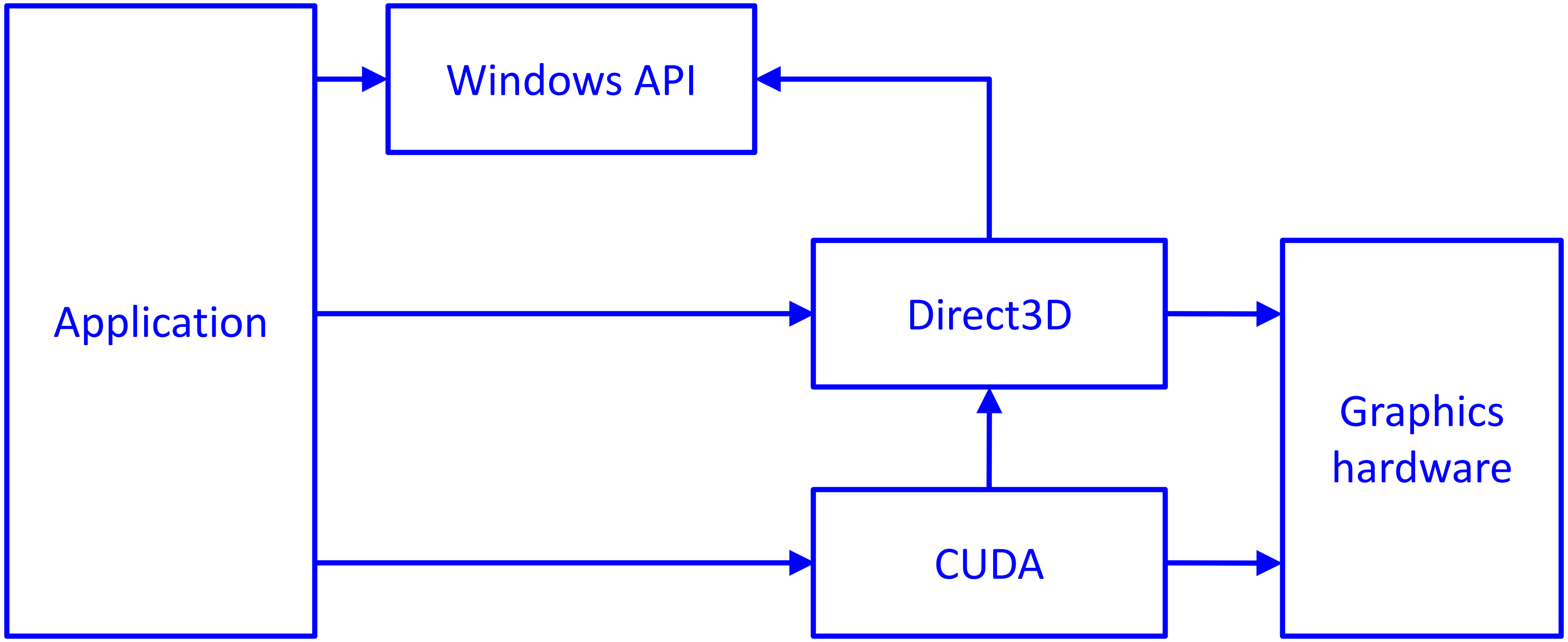
- Geometry is stored on the graphics hardware in the form of vertex buffers and index buffers
- Vertex buffer: array of vertices (unstructured geometry)
- Index buffer: array of indices into the vertex buffer (topology)
- Vertex: a point in 3D space that may have additional properties (e.g. color)
- Index: an integer identifying a certain element in a vertex buffer





CUDA-Direct3D Interaction

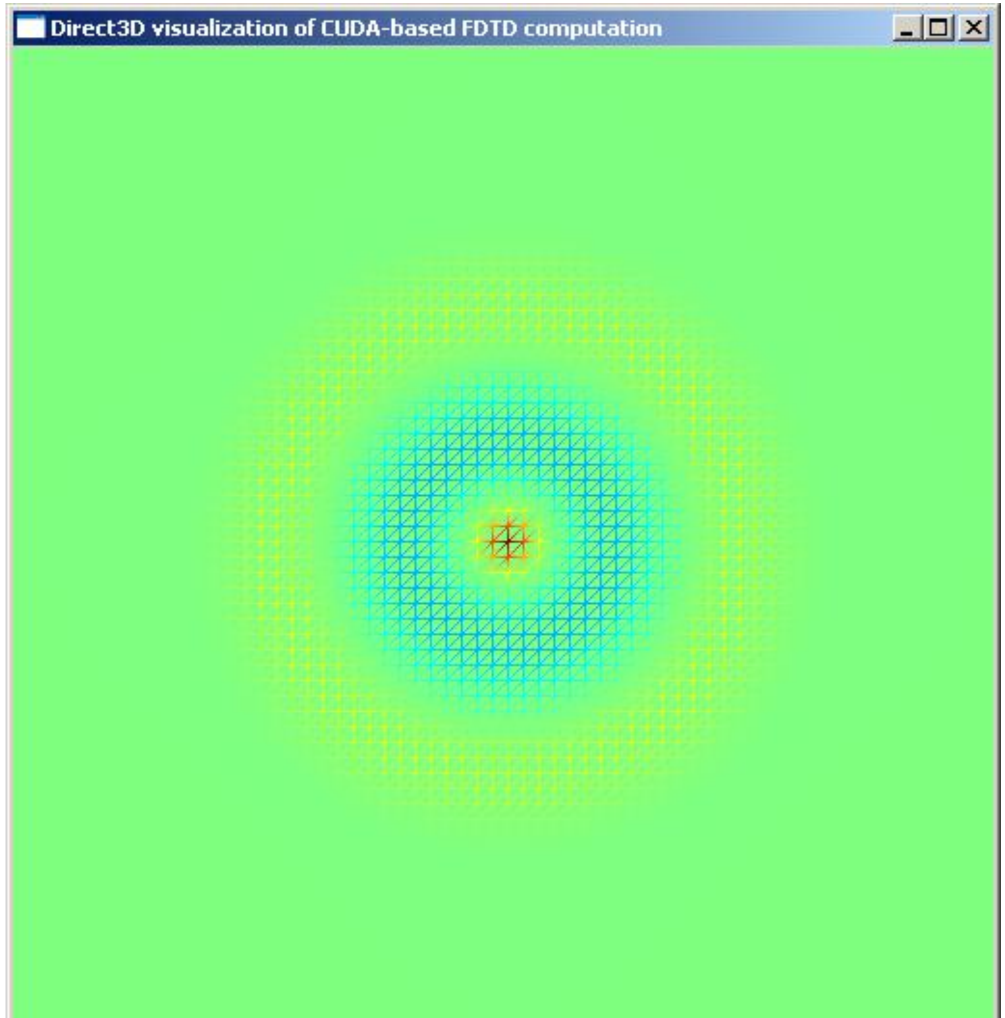
- Framework interconnection:



FDTD Visualization

- Idea: represent the simulation grid by a flat triangle mesh
  - Each vertex corresponds to the respective grid cell
  - Vertex color represents the data value
  - Update vertex colors after each simulation step by mapping the vertex buffer into the CUDA address space

FDTD Visualization (2)



CUDA & Direct3D: Final Application

- WinMain(args) function
  - Register a window class
  - Create a window based on the newly registered class
  - Call InitD3D(args)
  - Call InitGeometry()
  - Start a CUDA-Direct3D interoperability session
  - Register the vertex buffer to CUDA
  - Show the window
  - Enter the message loop calling Render() as idle function
- InitD3D(args) function
  - Set CUDA and Direct3D to operate on the same device
  - Create a Direct3D context and associate it with the newly created window
- Render() function
  - Run the CUDA computation and update the vertex buffer accordingly
  - Render the triangle mesh

FDTD Visualization: Final Output