Acceleration of a Finite-Difference Time-Domain Method with General Purpose GPUs (GPGPUs)


Institute for Parallel and Distributed Systems, University of Stuttgart*
BIMAQ, University of Bremen&
Outline

- Introduction
- Finite Difference Time-Domain Method (FDTD)
- GPGPU Architecture and Programming Model
- Relevant Aspects for Achieving High Performance
  - Effects of Memory Access Patterns
  - Multiprocessor Utilization
  - I/O Bandwidth
- FDTD Computational Performance on GPGPU
- Conclusion
Introduction - GPGPUs

Recent developments (last 3 years):

- Massive parallelism
- Higher level of abstraction of the programming model
- C + parallel concepts
- Single-precision floating-point: >100 GFLOPs
- Double (since Jun 2008): 1/10 single-prec. performance

A = GeForce FX 5800
B = GeForce FX 5950 Ultra
C = GeForce 6800 Ultra
D = GeForce 7800 GTX
E = Quadro FX 4500
F = GeForce 7900 GTX
G = GeForce 8800 GTX

Source: nextbigfuture.com
Finite-Difference Time-Domain Method

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

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

**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

\(nx, nz, nt\) ~ space and time coordinates
\(\Delta t\) ~ constant time step
\(E_y\) ~ electric field
\(J_{ey}\) ~ excitation

Suitable for parallel processing across spatial domain!
GPGPU Architecture

Issues: Neighborhood Operations and Thread Synchronization

- 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: must exchange values of boundary block cells w. neighboring thread blocks between time iterations
- GPGPU Architecture limitation:
  - No message passing
  - Shared Memory – Yes, but exclusive partition for each thread block:
    - 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

IPVS 6
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 \( k \)th thread in the half-warp must access the \( k \)th word.

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

Conclusion of experiments also that branches have less impact than uncoalesced memory accesses on the kernel performance.
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
I/O Bandwidth

- Max 4GB/s (peak) bandwidth in each direction on PCI Express x16 v1.1
- In practice, sustained bandwidth depends on many factors

Data transfer rates as the function of the grid size

Test configuration:
AMD Athlon 4000+ CPU on 2.4GHz, 2GB RAM, nVidia GeForce 8800 GTX GPGPU.

CUDA FDTD Implementation max transfer rate of **3.4GB/s**
(burst mode for large data sets – more than 16 Million Grid Cells)
FDTD Computation on GPGPU: Performance Results

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

50x

Number of Blocks/Grid
Conclusions

• Massive parallelism based on SIMD

• Computational performance sensitive to the global memory access patterns

• Computational speedup of 50x on large data sets

• PCIExpress bandwidth is a limiting factor, if frequent data transfers between GPU and PC are required e.g. FDTD: 50x -> 6x (if transfer in each time step)

• GPGPUs well suited for the acceleration of data-parallel computations e.g. finite-difference methods