EE5900 Spring 2012

Parallel VLSI CAD Algorithms

Lecture 8
Parallel simulation of large-scale on-chip interconnect on GPUs
Zhuo Feng
What is Graphics Processing Unit (GPU)

- Dedicated for graphics rendering
  - Graphics manipulation & display

GPU is connected to MCH via 16x PCI-Express

CPU (host)
General Purpose Computing on GPU (GPGPU)

- GPU has evolved into a very flexible and powerful processor
  - Calculation capability: 367 GFLOPS vs. 32 GFLOPS
  - Memory Bandwidth: 86 GB/s vs. 8 GB/s
  - Recent API allows lower learning curve than previous graphics API
    - NVIDIA’s CUDA language is an extension of C

History of GPU & CPU performance
Courtesy of Nvidia Corporation.
GPU outperforms CPU?

- GPU is designed for highly data parallel computation
  - All processors do the same job on different data sets (SIMD)
  - More transistors are devoted to data processing
  - Less transistors are used for data caching and control flow

- The fast-growing gaming market forces constant innovation
  - Quad-core CPU vs. 400-core GPU
  - # of transistors in GPU is doubling each year: exceeds Moore’s law
Geforce 8800 GPU

- 16 streaming multiprocessors (128 streaming processors)
  - Low-latency, fast on-chip shared memory (L2 cache in CPU)
  - High-volume, high-bandwidth global memory (DRAM in desktop)

On-Board Memory Bandwidth: >100 Gb/second

Host → Input Assembler → Thread Execution Manager

Parallel Data Cache
Texture
Read/write

Global Memory

Courtesy of Nvidia Corporation.
Streaming Multiprocessor (SM)

- Streaming Multiprocessor (SM)
  - 8 streaming processors (SP)
  - 2 super function units (SFU)
  - Multithreaded instruction fetch/dispatch unit

- Multi-threaded instruction dispatch
  - 1 to 512 active threads
  - 32 threads (a warp) share one instruction fetch
  - Cover memory load latency

- Some facts about an SM
  - 16 KB shared memory
  - 8,196 registers
  - >30 Gflops peak performance
Thread Grid, Block and Warps in CUDA

- **Grid** is a kernel function to be executed by many threads
  - Launched on SPA
  - Consists of many thread blocks
  - Up to 768 threads for as SM

- **Blocks** in a grid are serially distributed to the SMs
  - >1 thread block for each SM
  - Up to 512 threads in each Block

- **Warps** of threads are executed by SMs
  - A 32-thread pack is a warp
  - Each warp share the same instruction fetch
GPU Memory Space (CUDA Memory Model)

- Each thread:
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid texture/constant memory

<table>
<thead>
<tr>
<th></th>
<th>Local</th>
<th>Shared</th>
<th>Global</th>
<th>Texture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Write</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Size</td>
<td>Large</td>
<td>Small</td>
<td>Large</td>
<td>Large</td>
</tr>
<tr>
<td>BW</td>
<td>High</td>
<td>High</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td>Cached?</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Latency</td>
<td>500 cyc.</td>
<td>20 cyc.</td>
<td>500 cyc.</td>
<td>300 cyc.</td>
</tr>
</tbody>
</table>

Device Memory Comparison
Warp Scheduling of Streaming Multiprocessor (SM)

- Threads are grouped into blocks
  - Up to 8 Blocks go to one SM
  - Each SM occupies up to 768 threads
  - Threads are running concurrently

- Each Block is executed as 32-thread Warps
  - Warps are scheduling units in SM

- Warp scheduling of an SM
  - Warps are executed one by one
  - 32-thread warp executes the same instruction
  - 4 clock cycles for each instruction dispatch
Data Decomposition & Sharing

- Data decomposition is critical for GPU computing
  - Perform similar operations on different parts of the data
    - Single instruction multiple threads (SIMT)
    - Single program multiple data (SPMD)
  - Reduce dependencies among the tasks (on different partitions)
    - Avoid data dependencies between different thread blocks
  - Adjust partition sizes based on hardware characteristics
    - Memory sizes, bandwidth, …

- Data sharing is not always beneficial
  - Excessive data sharing should be avoided
  - Localized data sharing should be maximized
    - Efficiently utilize on-chip memory resources
  - Read-only sharing is more efficient than read-write sharing
    - Minimize synchronization times
Shared Memory Utilization

- **Fast access w/o bank conflicts**
  - All threads of a half-warp (16 threads) access different banks,
  - All threads of a half-warp access the identical address (broadcast)

- **Slow access due to bank conflicts**
  - Some (not all) threads in the same half-warp access the same bank
    - Will serialize the operations
Control Flow of GPU Computing

- Avoid branching (divergence)
  - Threads of a single warp take should take the same path
  - Different execution paths will be serialized by SMs

- Parallel reduction helps
  - “Reduce” an array of values to a single value in parallel
  - Examples: get sum/maximum of all values in an array

- Parallel implementation:
  - Formulate the original problem into a multi-level problem
  - Split the whole procedure into log(n) steps for n elements
Opportunities and Challenges For VLSI Design

- Accelerating some VLSI CAD applications on GPU is promising
  - Graphics-like problems may be more efficiently solved
    - E.g. on-chip interconnect analysis, clock mesh simulation, N-body problems, etc.

- Algorithm development on GPU can be challenging
  - Problem mapping is non-trivial
    - Solve a graphics-like problem that is close to the original one
  - Utilizing the hardware resources is critical
    - On-chip memory (registers, shared memory etc.)
    - Host-to-device data communication
    - Control flow and data access pattern
  - Software support (numerical packages for GPU) is limited
    - No sparse matrix solvers available

- Our first success: Fast on-chip power grid analysis on GPU
Motivation

- **On-chip power distribution network verification challenge**
  - Tens of millions of grid nodes (recent IBM design reaches ~400 M)
  - Prohibitively long simulation time for power grid verifications

- **Massively parallel circuit simulation algorithms on GPUs**
  - Pros: very cost efficient: 240-core GPU costs less than $400
  - Cons 1: Hardware resource usage limitations
    - Shared memory size: 16KB/SM, number of registers: 16K/SM
  - Cons 2: Algorithm and data structure design complexities
    - Multilevel iterative algorithms: Multigrid, Fast Multipoles, etc
    - Coalesced device memory access patterns and simple control flows

- **This work: a robust electrical/thermal simulation engine on GPU**
  - Multigrid preconditioned solver assures good memory/runtime scalability
  - GPU-friendly data structures guarantee coalesced memory accesses
IR Drop in IC Power Distribution Network

- **IR drop**: voltage drop due to non-ideal resistive wires
Simple 1D Circuit Analysis Problem

- Stamp circuit elements (1 Ohm resistors) into matrix $G$
  - Kirchhoff's Current Law (KCL) lead to linear system $G*V=b$

\[
V_1: (v_s - v_1) + (v_2 - v_1) = 0 \\
\Rightarrow 2v_1 - v_2 = v_s \\

V_2: (v_1 - v_2) + (v_3 - v_2) = 0 \\
\Rightarrow 2v_2 - v_1 - v_3 = 0 \\

V_3: (v_2 - v_3) + (v_4 - v_3) = I_3 \\
\Rightarrow 2v_3 - v_2 - v_4 = -I_3 \\
\vdots
\]

\[
\begin{bmatrix}
2 & -1 & & & & \\
-1 & 2 & -1 & & & \\
-1 & 2 & -1 & & & \\
-1 & 2 & -1 & & & \\
-1 & 2 & -1 & & & \\
-1 & 2 & -1 & & & \\
\end{bmatrix}
\begin{bmatrix}
v_1 \\
v_2 \\
v_3 \\
v_4 \\
v_5 \\
v_6 \\
v_7 \\
v_8 \\
\end{bmatrix}
= 
\begin{bmatrix}
v_s \\
0 \\
-I_3 \\
0 \\
-I_5 \\
0 \\
0 \\
0 \\
\end{bmatrix}
\]
More Realistic Power Grid Analysis Problems

- Multi-layer interconnects are modeled as 3D RC network
  - Switching gate effects are modeled by time-varying current loadings

- DC analysis solves linear system

\[ G \cdot \vec{v} = \vec{b} \]

Tens of millions of unknowns!

- Transient analysis solves

\[ G \cdot \vec{v}(t) + C \cdot \frac{d\vec{v}(t)}{dt} = \vec{b}(t) \]

\[
\begin{align*}
G & \in \mathbb{R}^{n \times n} : \text{Conductance Matrix} \\
C & \in \mathbb{R}^{n \times n} : \text{Capacitance Matrix} \\
\vec{v} & \in \mathbb{R}^{n \times 1} : \text{Node Voltage Vector} \\
\vec{b} & \in \mathbb{R}^{n \times 1} : \text{Current Loading Vector}
\end{align*}
\]
Prior Works

- Prior power grid analysis methods
  - Direct methods (LU factorization, Cholesky decomposition)
    - Adopted in commercial tools: HSPICE (Synopsys) Eldo (Mentor Graphcis)
    - State-of-the-art solver uses 7GB memory and >1,000 s for a 9-million grid
  - Iterative methods are memory efficient
    - Preconditioned conjugate gradient (T. Chen et al, DAC’01)
    - Multigrid methods (S. Nassif et al, DAC’00)
  - Stochastic method
    - Random walk (H. Qian et al, DAC’05)
GPU-Based Algorithms

- Recent GPU based power grid analysis methods
  - Hybrid multigrid method on GPU (Z. Feng et al, ICCAD’08)
    - Pros: very fast (solves four million unknowns in one second)
    - Cons: convergence rate depends on 2D grid approximation
  - Poisson Solver (J. Shi et al, DAC’09)
    - Pros: public domain CUFFT library -> easier implementation
    - Cons: only solves 2D regular grids

- GPU-based preconditioned algorithms
  - Existing preconditioners use incomplete Cholesky matrix factors
    - Cons: Matrix factors are difficult to store and process on GPU
  - Our hybrid multigrid preconditioned CG algorithm (Z. Feng et al, DAC’10)
    - SIMD geometric multigrid solver + sparse matrix-vector operations
Point relaxations (Jacobi iteration) update solution by its neighbors

- **DC analysis** for node $x$ (KCL Law):

\[
V_x = \sum g \frac{V_1}{g_1} + \sum g \frac{V_2}{g_2} + \cdots + g \frac{V_{\text{degree}(x)}}{g_{\text{degree}(x)}} - \frac{I_x}{\sum g}
\]

- **Transient analysis** for node $x$:  

Slow convergence 😞

\[
V_x(t) = \sum_{i=1}^{\text{degree}(x)} \frac{g_i}{g_i + \frac{C_x}{h}} V_i(t) + \frac{C_x}{h} \sum_{i=1}^{\text{degree}(x)} \frac{V_x(t-h)}{g_i + \frac{C_x}{h}} - \frac{I_x(t)}{\sum g_i + \frac{C_x}{h}}
\]

- $g_i$: Conductance Value
- $I_x$: Current Source Value
- $C_x$: Capacitance Value
- $h$: Time Step Size
Multigrid Methods

- Among fastest numerical algorithms for PDE-like problems
  - Linear complexity in the number of unknowns if well designed
- A hierarchy of exact to coarse replicas of the problem
  - High (low) frequency errors damped on fine (coarse) grids by relaxations
  - Direct/iterative solvers for coarsest grid

Error Distribution: \( e(x) = V(x) - V^*(x) \)

Slowly varying

Fast varying
Multigrid Methods (Cont.)

- Multigrid operations
  1. Smoothing: Gauss Jacobi, Gauss Seidel …
  2. Restriction: fine grid residue $\rightarrow$ coarse grid RHS
  3. Prolongation: coarse grid solution $\rightarrow$ fine grid error
  4. Correction: fine grid solution + fine grid error

- Algebraic MG (AMG) and Geometric MG (GMD)
Power Grid Physics

- Real-life power grid designs are not regular
  - High degree of global uniformity & local irregularity
- Store and process a regularized grid as a preconditioner on GPU
  - Topology regulation for the original 3D grid
  - Regular grid solver using geometric multigrid method (GMD)
  - Need further correction scheme to guarantee the solution quality (accuracy)

Top view of an industrial power grid design (partial)
Power Grid Topology Regularization (2D to 1D)

Original Grid

Regular Grid

\[ G_1 = 2g_{31} + g_{21}, \quad G_2 = 2g_{31} + g_{22}, \quad G_3 = 2g_{32}, \]
\[ G_4 = 2g_{32} + g_{23}, \quad G_5 = g_{33} + g_{24}. \]
Power Grid Topology Regularization (3D to 2D)

Location-based mapping (Z. Feng et al, ICCAD’08)

2D Regular Grid

Metal 1~2

Metal 3~4

Metal 5~6
Multigrid Hierarchy: Wires

- Vertical and horizontal conductance values are averaged

\[ G_H = \frac{1}{4} \left( \sum_{i=1}^{4} G_{H_i} \right) \]

\[ G_V = \frac{1}{4} \left( \sum_{i=1}^{4} G_{V_i} \right) \]

Inter-grid horizontal conductance computation
Multigrid Hierarchy: VDD/GND Pads

- Total Vdd / Gnd pad conductance is kept throughout all levels
  - Similar to the inter-grid restriction operator
  - Maintain the same pull-up & pull-down strengths in the coarser grids

Vdd/Gnd Pad Conductance

\[
G_z = \sum_{i=1}^{4} G_{z_i}
\]
Multigrid Restriction Operation

- **Simple residual collection scheme**
  - Minimize data communications between nodes
  - Simplify GPU implementation

**Restriction Operation: Fine Grid Residual to Coarse Grid RHS**

\[
R H S_z = \sum_{i=1}^{4} R_{z_i}
\]

Fine Grid

Coarse Grid
Multigrid Prolongation Operation

- **Simple interpolation scheme**
  - Minimize data communications between nodes
  - Simplify GPU implementation

**Prolongation Operation: Coarse Grid Solution to fine grid error**

![Prolongation Operation Diagram](image)

$V_{H_1} = V_H$  $V_{H_2} = V_H$

$V_{H_3} = V_H$  $V_{H_4} = V_H$

Fine Grid

Coarse Grid
Parallel Hybrid Multigrid Solver

- **3D (irregular) grid smoother + 2D (regular) grid multigrid solver**
  - 3D finest grid is stored using ELL sparse matrix format
  - 2D coarser to coarsest grids are processed geometrically as pixels
- Coalesced memory accesses are achieved on GPU
GMD Relaxation (Smoothing) On GPU

- **Mixed block-wise relaxation scheme** *(Z. Feng et al ICCAD’08)*
  - Global block-wise Gauss-Seidel relaxation *(GBG)*
    - Block solution is updated by neighboring node blocks
    - More than two blocks should be run concurrently on each SM
      - Reduce exposed memory latency
  
  - Local block-wise weighted Jacobi relaxation *(LBJ)*
    - Local nodes within a block are updated by themselves
    - GPU’s thread array (block) works fabulously fast
    - Require many times local iterations for each data access
      - Increase the arithmetic intensity

- **GPU Performance modeling and optimization**
  - Exploits warp level parallelism and work flow graphs *(Sara et al PPoPP’10)*
  - Parameterized runtime models help efficient kernel optimizations *(Z. Feng et al TCAD’11)*
GMD Smoother

- Mixed block-wise relaxation on GPU

- Weighted Jacobi iterations within each block
- Streaming processors
- Multiprocessors
- Gauss-Seidel iterations among blocks

Execution Time

Z. Feng  MTU EE5900
Shared Memory Access Pattern of Local Relaxation

\[
X_{\text{temp}} + = V(t_x, t_y + 1) G_h(t_x, t_y + 1)
\]

\[
X_{\text{temp}} + = V(t_x + 2, t_y + 1) G_h(t_x + 1, t_y + 1)
\]

\[
X_{\text{temp}} + = V(t_x + 1, t_y) G_v(t_x + 1, t_y)
\]

\[
X_{\text{temp}} + = V(t_x + 1, t_y + 2) G_v(t_x + 1, t_y + 1)
\]
Local Relaxation On GPU: Pseudo Code

// declare texture memory
texture<float, 1, cudaReadModeElementType> texGh, texGv, texGz;

__global__ void SmoothGPU(float* d_sol, float* d_rhs, int K)
{
    //Load grid data (RHS) from texture (global) memory to registers;
    float Gupper, Glower, Gright, Gleft, Rhs;
    ....

    //Load solution from global memory to shared memory (including boundary data)
    __shared__ float GpuSol[BLOCK_SIZE+2][BLOCK_SIZE+2];
    ....

    //Do K times block based relaxations using 2D thread array
    for(i=0; i<K; i++){
        ....
    }

    //synchronize threads
    __syncthreads();

    //Write the final solution back to the global memory
    d_sol[IdxStart2D] = GpuSol[ty+1][tx+1];
}

2D Grid data
Solution, RHS
GMD Dummy Grid Padding

- Dummy grids are padded to the original 2D regular grids
  - Adjust grid dimensions to be multiples of 16
  - Dummy grids should be well isolated from the original grids
    - Otherwise, may face slow convergence

Original Grid + Dummy Grid = Final Grid
Hybrid Multigrid Preconditioned Solver

- Multigrid preconditioned Krylov subspace iterative solver on GPU

**Host (CPU) Memory**

3D Multi-layer Irregular Power Grid

**GPU Global Memory**

Jacobi Smoother using Sparse Matrix

```
\begin{bmatrix}
  a_{11} & a_{12} & a_{13} & a_{14} \\
  a_{21} & a_{22} & a_{23} & a_{24} \\
  a_{31} & a_{32} & a_{33} & a_{34} \\
  a_{41} & a_{42} & a_{43} & a_{44}
\end{bmatrix}
```

Geometrica Multigrid Solver (GMD)

**MGPCG Algorithm on GPU**

- Set Initial Solution
- Get Initial Residual and Search Direction
- Update Solution and Residual
- Check Convergence
- Multigrid Preconditioning
- Update Search Direction
- Return Final Solution
- Converged
- Not Converged

**Original Grid**

\[ Gx = b \]

**Matrix + Geometrical Representations**

\[ Gx(t) + C \frac{dx(t)}{dt} = b(t) \]

**GPU-friendly Multi-level Iterative Algorithm**
MGPCG Memory Layout on GPU

- Mixed data structures

Original grid (finest grid level)

ELL-like Sparse Matrix

Level 0

Restriction

Prolongation

Level 1

Regularized coarse to coarsest grids

Level 2

Level 3
(coarsest grid)

Graphics Pixels on GPU

8.37

Z. Feng  MTU EE5900
Nodal Analysis Matrix for 3D Grid

- ELL-like sparse matrix storage

\[
\begin{bmatrix}
    a_{1,1} & a_{1,4} & a_{1,5} \\
    a_{2,2} & a_{2,3} & a_{2,6} \\
    a_{3,2} & a_{3,3} & a_{3,8} \\
    a_{4,1} & a_{4,4} \\
    a_{5,1} & a_{5,5} & a_{5,7} \\
    a_{6,2} & a_{6,6} & a_{6,8} \\
    a_{7,5} & a_{7,7} \\
    a_{8,3} & a_{8,6} & a_{8,8}
\end{bmatrix}
\]

\[A = D + M\]

- \(D\): Diagonal Elements of A
- \(M\): Off-Diagonal Elements of A

\[
M = \begin{bmatrix}
    a_{1,4} & a_{1,5} \\
    a_{2,3} & a_{2,6} \\
    \vdots & \vdots \\
    a_{7,5} & 0 \\
    a_{8,3} & a_{8,6}
\end{bmatrix}
\]

\[P = \begin{bmatrix}
    \frac{1}{a_{1,1}} \\
    \frac{1}{a_{2,2}} \\
    \vdots \\
    \frac{1}{a_{7,7}} \\
    \frac{1}{a_{8,8}}
\end{bmatrix}
\]

\[D^{-1} = \begin{bmatrix}
    \frac{1}{a_{1,1}} & \frac{1}{a_{2,2}} & \vdots & \frac{1}{a_{7,7}} & \frac{1}{a_{8,8}}
\end{bmatrix}\]
GPU Device Memory Access Pattern

- GPU-based Jacobi Iteration (smoother): \( x^{(k+1)} = D^{-1} \left[ b - Mx^{(k)} \right] \)

\[
\begin{align*}
t_1 \rightarrow a_{1,4} & \quad t_1 \rightarrow 4 & \quad t_1 \rightarrow a_{1,5} & \quad t_1 \rightarrow 5 & \quad t_1 \rightarrow \frac{1}{a_{1,1}} \\
t_2 \rightarrow a_{2,3} & \quad t_2 \rightarrow 3 & \quad t_2 \rightarrow a_{2,6} & \quad t_2 \rightarrow 6 & \quad t_2 \rightarrow \frac{1}{a_{2,2}} \\
\vdots & \quad \vdots & \quad \vdots & \quad \vdots & \quad \vdots \\
t_7 \rightarrow a_{7,5} & \quad t_7 \rightarrow 5 & \quad t_7 \rightarrow 0 & \quad t_7 \rightarrow 0 & \quad t_7 \rightarrow \frac{1}{a_{7,7}} \\
t_8 \rightarrow a_{8,3} & \quad t_8 \rightarrow 3 & \quad t_8 \rightarrow a_{8,6} & \quad t_8 \rightarrow 6 & \quad t_8 \rightarrow \frac{1}{a_{8,8}} \\
\vdots & \quad \vdots & \quad \vdots & \quad \vdots & \quad \vdots \\
\text{Execution Time} & \quad T_1 & \quad T_2 & \quad T_3 & \quad T_4 \\
S = b - Mx^{(k)} & \quad x^{(k+1)} = D^{-1}S
\end{align*}
\]
GpuGMD (128-Core GPU) vs. Cholmod (8-Core CPU)

- **Synthetic 2D regular grids**
  - 20~50X faster than the 8-threading CHOLMOD solver
  - 40~180X faster than the 1-threading CHOLMOD solver
  - ~20X memory saving than CHOLMOD
GpuGMD Scalability (240-Core GPU)

- Synthetic 2D regular grids (with up to 30 million nodes)
  - Linear simulation time (4 million nodes per second)
  - Linear memory consumption (50Mb per million nodes)
Transient Analysis: CKT5

CKT5 with 1.7M nodes
500 time steps
693 MGPCG iters.

Cholmod: 2,700s
GPU: 128s
22X Speedups

Z. Feng  MTU EE5900
3D-IC Microchannel Design Methodology

- **Full-chip thermal simulators for 3D-IC designs**
  - Develop novel preconditioners considering dominant heat dissipation paths
  - Parallelize computing tasks on latest heterogeneous computing platforms

![Diagram of 3D-IC Microchannel Design Methodology](image-url)
Two-Step Block Relaxation Scheme

- **Line smoother in the vertical direction Z**
  - Solution update for vertical heat dissipations
  - Block solution update from neighbors

- **Line smoother along the microchannel routing direction Y**
  - Solution update for lateral heat dissipation
  - Approximate heat flux along the channels

<table>
<thead>
<tr>
<th>Fine-Grained Microchannel Modeling</th>
<th>“Physically” Motivated Simulation Techniques</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Diagram of Microchannel Fluidics Cooling" /></td>
<td><img src="image2" alt="Diagram of Vertical and Y-Direction Relaxation" /></td>
</tr>
</tbody>
</table>

Equivalent circuit model  
Micro-channel fluidics cooling  
Substrate Layer 1  
Substrate Layer 2  
Block Vertical Relaxation  
Block Y-Direction Relaxation  
Block 1  
Block 2  
Block 3
Block Tri-Diagonal Preconditioner on GPU

Block Tri-Diagonal Matrix

\[
B_{k,k} = \begin{bmatrix}
b_{1,1} & c_{1,2} \\
c_{1,2} & b_{2,2} & c_{2,3} \\
c_{2,3} & b_{3,3} & \ddots \\
\vdots & \ddots & \ddots \\
c_{N-1,N} & b_{N,1} & c_{N,N}
\end{bmatrix}
\]

Original Matrix

\[
G = \begin{bmatrix}
\end{bmatrix}
\]

3-D FD Thermal Model

1. Data Loaded to 2-D shared memory

1-D Threads

2. SIMD Thomas Algorithm Executions

1-D Threads

Data Loaded to 2-D shared memory

2. SIMD Thomas Algorithm Executions
Thermal Simulation Runtime Results

- Preconditioned Conjugate Gradient Solvers on CPU and GPU
  - Multigrid preconditioned CG achieves up to 54X speedups

<table>
<thead>
<tr>
<th>Grid Size</th>
<th>CG (CPU)</th>
<th>CG (GPU)</th>
<th>DPCG (CPU)</th>
<th>DPCG (GPU)</th>
<th>MGPCG (GPU)</th>
<th>MGPCG* (GPU)</th>
<th>Cholmod (CPU)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.50M</td>
<td>85</td>
<td>3.4</td>
<td>32.6</td>
<td>1.4</td>
<td>0.8</td>
<td>1.9</td>
<td>294</td>
<td>40X</td>
</tr>
<tr>
<td>0.85M</td>
<td>141</td>
<td>5.5</td>
<td>43.2</td>
<td>1.8</td>
<td>1.2</td>
<td>2.3</td>
<td>N/A</td>
<td>36X</td>
</tr>
<tr>
<td>1.20M</td>
<td>198</td>
<td>7.5</td>
<td>82.3</td>
<td>3.2</td>
<td>1.8</td>
<td>3.4</td>
<td>N/A</td>
<td>51X</td>
</tr>
<tr>
<td>2.10M</td>
<td>384</td>
<td>15.2</td>
<td>131.5</td>
<td>5.4</td>
<td>3.15</td>
<td>6.1</td>
<td>N/A</td>
<td>42X</td>
</tr>
<tr>
<td>3.06M</td>
<td>774</td>
<td>25.5</td>
<td>230</td>
<td>8.8</td>
<td>5.10</td>
<td>8.5</td>
<td>N/A</td>
<td>35X</td>
</tr>
<tr>
<td>4.25M</td>
<td>862</td>
<td>32.4</td>
<td>300.3</td>
<td>11.7</td>
<td>8.5</td>
<td>14.6</td>
<td>N/A</td>
<td>35X</td>
</tr>
<tr>
<td>6.0M</td>
<td>1,210</td>
<td>54.5</td>
<td>553.2</td>
<td>21.6</td>
<td>10.3</td>
<td>22.5</td>
<td>N/A</td>
<td>54X</td>
</tr>
</tbody>
</table>
Thermal Simulation Runtime Results (Cont.)

- **Block Tri-Diagonal Preconditioned CG Solver**
  - Can more effectively handle thermal grids with strongly non-uniform thermal conductivities than multigrid preconditioned algorithms

<table>
<thead>
<tr>
<th></th>
<th>Thomas Algorithm Results (1,000 Solves)</th>
<th>Block Preconditioned CG Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5M</td>
<td></td>
<td>13.8s</td>
</tr>
<tr>
<td>2.1M</td>
<td></td>
<td>43.4s</td>
</tr>
<tr>
<td>6.0M</td>
<td></td>
<td>132s</td>
</tr>
</tbody>
</table>

Simulation of Large Scale On-Chip Interconnect on GPUs