Memory Access Patterns: The Missing Piece of the Multi-GPU Puzzle

Tal Ben-Nun, Ely Levy, Amnon Barak and Eri Rubin
The Hebrew University of Jerusalem, Israel

Supercomputing ‘15, November 2015
Introduction

• Developing efficient parallel algorithms for GPUs is challenging
  • Memory I/O recurs as the bottleneck
  • Code clutter caused by device and memory management

• Current programming models for multi-GPU nodes are often:
  • Insufficient for specific programming needs
  • Overly complex
  • Hard to debug
Case Study – The Game of Life

• Famous cellular automaton
• Each cell requires its 3×3 neighborhood to compute next generation

• In GPUs, each thread computes one (or several) cells
• Similar to stencil operators, the Jacobi method and many more
Case Study – The Game of Life

- Famous cellular automaton
- Each cell requires its $3 \times 3$ neighborhood to compute next generation

- In GPUs, each thread computes one (or several) cells
- Similar to stencil operators, the Jacobi method and many more
Case Study – The Game of Life

- Famous cellular automaton
- Each cell requires its 3×3 neighborhood to compute next generation

- In GPUs, each thread computes one (or several) cells
- Similar to stencil operators, the Jacobi method and many more
Case Study – The Game of Life

• Famous cellular automaton
• Each cell requires its 3×3 neighborhood to compute next generation

• In GPUs, each thread computes one (or several) cells
• Similar to stencil operators, the Jacobi method and many more
Case Study – The Game of Life

• Famous cellular automaton
• Each cell requires its 3×3 neighborhood to compute next generation

In GPUs, each thread computes one (or several) cells
• Similar to stencil operators, the Jacobi method and many more
Case Study – The Game of Life

• Famous cellular automaton
• Each cell requires its $3 \times 3$ neighborhood to compute next generation

![Input](image1)  ![Output](image2)

• In GPUs, each thread computes one (or several) cells
• Similar to stencil operators, the Jacobi method and many more
Case Study – The Game of Life

• Famous cellular automaton
• Each cell requires its $3 \times 3$ neighborhood to compute next generation

![Input](image1) ![Output](image2)

• In GPUs, each thread computes one (or several) cells
• Similar to stencil operators, the Jacobi method and many more
Case Study – The Game of Life

• Famous cellular automaton
• Each cell requires its 3×3 neighborhood to compute next generation

![Input and Output Diagram]

• In GPUs, each thread computes one (or several) cells
• Similar to stencil operators, the Jacobi method and many more
Case Study – The Game of Life

• Famous cellular automaton
• Each cell requires its $3\times3$ neighborhood to compute next generation

• In GPUs, each thread computes one (or several) cells
• Similar to stencil operators, the Jacobi method and many more
Case Study – The Game of Life

Thread Pseudocode:

```c
// Load data to shared memory
smem[tidy * BW + tidx] = 
    M[(bidy * BH + tidy) * STRIDE + 
    (bidx * BW + tidx)]; // Wrap coords
// ...
__syncthreads();

neighbors = 0;
current_gen = smem[tidy * BW + tidx];
for (int ly = -1; ly <= 1; ++ly) {
    for (int lx = -1; lx <= 1; ++lx) {
        if (lx == 0 && ly == 0) continue;
        neighbors += smem[(ly + tidy) * BW + 
            (lx + tidx)];
    }
}

outM[(bidy * BH + tidy) * STRIDE + 
    (bidx * BW + tidx)] = ...;
```
Case Study – The Game of Life

Multi-GPU

Boundary Exchanges

GPU 1
GPU 2
GPU 3
GPU 4
Case Study – The Game of Life

M₁(0,0) = M[STRIDE]
M₁(x, -1) = M₄(x, \text{HEIGHT/4}) = M[(\text{HEIGHT-1}) \times \text{STRIDE} + x]
M₃(0,0) = M₂(0,\text{HEIGHT/4}+1) = M[((\text{HEIGHT/4}) \times 2 + 1) \times \text{STRIDE}]
Case Study – The Game of Life

- 3 indexing systems:
  - Node memory
  - Per-GPU memory
  - Shared memory/registers

- Error-prone
  - Index-wise
  - Synchronization-wise

- Difficult to debug/maintain

- Many lines-of-code

Multi-GPU
<table>
<thead>
<tr>
<th>Access Pattern</th>
<th>Description</th>
<th>Typical Examples</th>
<th>Illustrations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block (ND)</td>
<td>Each thread requires an entire dimension of a buffer</td>
<td>Matrix multiplication, Exact N-body simulation, Matrix transposition</td>
<td><img src="image1" alt="Diagram" /> <img src="image2" alt="Diagram" /> <img src="image3" alt="Diagram" /></td>
</tr>
<tr>
<td>Window (ND)</td>
<td>Each thread-block requires a spatially-local N-dimensional window</td>
<td>ND convolution, Jacobi method, Stencil operators</td>
<td><img src="image4" alt="Diagram" /> <img src="image5" alt="Diagram" /> <img src="image6" alt="Diagram" /></td>
</tr>
<tr>
<td>Adjacency</td>
<td>Sporadic access of a dense data structure with a fixed pattern</td>
<td>SpMV, Cloth simulation</td>
<td><img src="image7" alt="Diagram" /></td>
</tr>
<tr>
<td>Traversal (BFS, DFS)</td>
<td>Thread operates on neighbors of a vertex</td>
<td>Barnes-Hut N-body algorithm</td>
<td><img src="image8" alt="Diagram" /></td>
</tr>
<tr>
<td>Permutation</td>
<td>Thread-block operates on a permutation of the original data</td>
<td>Fast Fourier transform</td>
<td><img src="image9" alt="Diagram" /></td>
</tr>
<tr>
<td>Irregular</td>
<td>Patterns that cannot be determined in advance</td>
<td>Finite state machines</td>
<td><img src="image10" alt="Diagram" /></td>
</tr>
</tbody>
</table>
Output Memory Access Patterns

• Based on all possible mappings between number of threads and number of outputs per buffer:

- $n \rightarrow O(n)$
- $n \rightarrow m < n$
- $n \rightarrow$ Unpredictable

Structured Injective
Unstructured Injective
Reductive Static
Reductive Dynamic
Irregular
MAPS-Multi

• An automatic multi-GPU task partitioning framework:
  • By expressing the input/output access patterns of each task, automatically segments and copies memory
  
  • Based on concepts from the Partitioned Global Address Space (PGAS) model
  
  • No source-to-source compilation or other intrusive actions
  
  • Header only, standard C++11 (over CUDA) library
  
  • Can work in conjunction with other systems (e.g. MPI) and device-level libraries (e.g. CUBLAS)
Game of Life Code Sample

Host Code

```c
Scheduler sched;

typedef Window2D<T,1,WRAP,ILPX,ILPY> Win2D;
typedef StructuredInjective<T,2,ILPX,ILPY> SMat;

// Define data structures to be used
Matrix<T> A (width, height), B (width, height);

// Use existing host buffers as matrices
A.Bind(hbuffer_A);
B.Bind(hbuffer_B);

// Analyze memory access patterns for allocation
sched.AnalyzeCall(Win2D(A), SMat(B));
sched.AnalyzeCall(Win2D(B), SMat(A));

// Invoke the kernels
for (int i = 0; i < iterations; ++i)
    sched.Invoke(GameOfLifeTick,
                  Win2D((i % 2) ? B : A),
                  SMat((i % 2) ? A : B));

// Gather processed data back to host
if ((iterations % 2) == 0)
    sched.Gather(A);
else
    sched.Gather(B);
```

Device Code

```c
template <typename T, int ILPX, int ILPY>
__global__ void GameOfLifeTick MAPS_MULTIDEF(
    Window2D<T,1,WRAP,ILPX,ILPY> current_gen,
    StructuredInjective<T,2,ILPX,ILPY> next_gen) {

    MAPS_MULTI_INIT(current_gen, next_gen);

    #pragma unroll
    MAPS_FOREACH(nextgen_iter, next_gen) {
        int live_neighbors = 0, is_live = 0;
        #pragma unroll
        MAPS_FOREACH_ALIGNED(iter, current_gen, nextgen_iter) {
            // Set variables according to the rules
            if (iter.index() == 4)
                is_live = *iter;
            else
                live_neighbors += *iter;
        }
        int result = GameOfLifeConditions(...);
        *nextgen_iter = result;
    }
    next_gen.commit();
}
```
Code Sample – Host

`Scheduler sched;`

```c
typedef Window2D<T,1,WRAP,ILPX,ILPY> Win2D;
typedef StructuredInjective<T,2,ILPX,ILPY> SMat;
```

// Define data structures to be used
Matrix<T> A (width, height), B (width, height);

// Use existing host buffers as matrices
A.Bind(hbuffer_A);
B.Bind(hbuffer_B);

// Analyze memory access patterns for allocation
sched.AnalyzeCall(Win2D(A), SMat(B));
sched.AnalyzeCall(Win2D(B), SMat(A));

// Invoke the kernels
for (int i = 0; i < iterations; ++i)
    sched.Invoke(GameOfLifeTick,
        Win2D((i % 2) ? B : A),
        SMat((i % 2) ? A : B));

// Gather processed data back to host
if ((iterations % 2) == 0)
    sched.Gather(A);
else
    sched.Gather(B);```
Scheduler sched;

typedef Window2D<T,1,WRAP,ILPX,ILPY> Win2D;
typedef StructuredInjective<T,2,ILPX,ILPY> SMat;

// Define data structures to be used
Matrix<T> A (width, height), B (width, height);

// Use existing host buffers as matrices
A.Bind(hbuffer_A);
B.Bind(hbuffer_B);

// Analyze memory access patterns for allocation
sched.AnalyzeCall(Win2D(A), SMat(B));
sched.AnalyzeCall(Win2D(B), SMat(A));

// Invoke the kernels
for (int i = 0; i < iterations; ++i)
    sched.Invoke(GameOfLifeTick,
        Win2D((i % 2) ? B : A),
        SMat((i % 2) ? A : B));

// Gather processed data back to host
if ((iterations % 2) == 0)
    sched.Gather(A);
else
    sched.Gather(B);
```cpp
Scheduler sched;

typedef Window2D<T,1,WRAP,ILPX,ILPY> Win2D;
typedef StructuredInjective<T,2,ILPX,ILPY> SMat;

// Define data structures to be used
Matrix<T> A (width, height), B (width, height);

// Use existing host buffers as matrices
A.Bind(hbuffer_A);
B.Bind(hbuffer_B);

// Analyze memory access patterns for allocation
sched.AnalyzeCall(Win2D(A), SMat(B));
sched.AnalyzeCall(Win2D(B), SMat(A));

// Invoke the kernels
for (int i = 0; i < iterations; ++i)
    sched.Invoke(GameOfLifeTick,
                 Win2D((i % 2) ? B : A),
                 SMat((i % 2) ? A : B));

// Gather processed data back to host
if ((iterations % 2) == 0)
    sched.Gather(A);
else
    sched.Gather(B);
```

---

**Code Sample – Host**

- Scheduler `sched`;
- `typedef Window2D<T,1,WRAP,ILPX,ILPY> Win2D;`
- `typedef StructuredInjective<T,2,ILPX,ILPY> SMat;`

// Define data structures to be used
Matrix<T> A (width, height), B (width, height);

// Use existing host buffers as matrices
A.Bind(hbuffer_A);
B.Bind(hbuffer_B);

// Analyze memory access patterns for allocation
sched.AnalyzeCall(Win2D(A), SMat(B));
sched.AnalyzeCall(Win2D(B), SMat(A));

// Invoke the kernels
for (int i = 0; i < iterations; ++i)
    sched.Invoke(GameOfLifeTick,
                 Win2D((i % 2) ? B : A),
                 SMat((i % 2) ? A : B));

// Gather processed data back to host
if ((iterations % 2) == 0)
    sched.Gather(A);
else
    sched.Gather(B);
Scheduler sched;

typedef Window2D<T,1,WRAP,ILPX,ILPY> Win2D;
typedef StructuredInjective<T,2,ILPX,ILPY> SMat;

// Define data structures to be used
Matrix<T> A (width, height), B (width, height);

// Use existing host buffers as matrices
A.Bind(hbuffer_A);
B.Bind(hbuffer_B);

// Analyze memory access patterns for allocation
sched.AnalyzeCall(Win2D(A), SMat(B));
sched.AnalyzeCall(Win2D(B), SMat(A));

// Invoke the kernels
for (int i = 0; i < iterations; ++i)
    sched.Invoke(GameOfLifeTick,
        Win2D((i % 2) ? B : A),
        SMat((i % 2) ? A : B));

// Gather processed data back to host
if ((iterations % 2) == 0)
    sched.Gather(A);
else
    sched.Gather(B);
Scheduler sched;

typedef Window2D<T,1,WRAP,ILPX,ILPY> Win2D;
typedef StructuredInjective<T,2,ILPX,ILPY> SMat;

// Define data structures to be used
Matrix<T> A (width, height), B (width, height);

// Use existing host buffers as matrices
A.Bind(hbuffer_A);
B.Bind(hbuffer_B);

// Analyze memory access patterns for allocation
sched.AnalyzeCall(Win2D(A), SMat(B));
sched.AnalyzeCall(Win2D(B), SMat(A));

// Invoke the kernels
for (int i = 0; i < iterations; ++i)
  sched.Invoke(GameOfLifeTick,
               Win2D((i % 2) ? B : A),
               SMat((i % 2) ? A : B));

// Gather processed data back to host
if ((iterations % 2) == 0)
  sched.Gather(A);
else
  sched.Gather(B);
Scheduler sched;

typedef Window2D<T,1,WRAP,ILPX,ILPY> Win2D;
typedef StructuredInjective<T,2,ILPX,ILPY> SMat;

// Define data structures to be used
Matrix<T> A (width, height), B (width, height);

// Use existing host buffers as matrices
A.Bind(hbuffer_A);
B.Bind(hbuffer_B);

// Analyze memory access patterns for allocation
sched.AnalyzeCall(Win2D(A), SMat(B));
sched.AnalyzeCall(Win2D(B), SMat(A));

// Invoke the kernels
for (int i = 0; i < iterations; ++i)
    sched.Invoke(GameOfLifeTick,
                 Win2D((i % 2) ? B : A),
                 SMat((i % 2) ? A : B));

// Gather processed data back to host
if ((iterations % 2) == 0)
    sched.Gather(A);
else
    sched.Gather(B);
Scheduler sched;

typedef Window2D<T,1,WRAP,ILPX,ILPY> Win2D;
typedef StructuredInjective<T,2,ILPX,ILPY> SMat;

// Define data structures to be used
Matrix<T> A (width, height), B (width, height);

// Use existing host buffers as matrices
A.Bind(hbuffer_A);
B.Bind(hbuffer_B);

// Analyze memory access patterns for allocation
sched.AnalyzeCall(Win2D(A), SMat(B));
sched.AnalyzeCall(Win2D(B), SMat(A));

// Invoke the kernels
for (int i = 0; i < iterations; ++i)
    sched.Invoke(GameOfLifeTick,
                  Win2D((i % 2) ? B : A),
                  SMat((i % 2) ? A : B));

// Gather processed data back to host
if ((iterations % 2) == 0)
    sched.Gather(A);
else
    sched.Gather(B);
Code Sample – Host

```c
Scheduler sched;

typedef Window2D<T,1,WRAP,ILPX,ILPY> Win2D;
typedef StructuredInjective<T,2,ILPX,ILPY> SMat;

// Define data structures to be used
Matrix<T> A (width, height), B (width, height);

// Use existing host buffers as matrices
A.Bind(hbuffer_A);
B.Bind(hbuffer_B);

// Analyze memory access patterns for allocation
sched.AnalyzeCall(Win2D(A), SMat(B));
sched.AnalyzeCall(Win2D(B), SMat(A));

// Invoke the kernels
for (int i = 0; i < iterations; ++i)
    sched.Invoke(GameOfLifeTick,
                 Win2D((i % 2) ? B : A),
                SMat((i % 2) ? A : B));

// Gather processed data back to host
if ((iterations % 2) == 0)
    sched.Gather(A);
else
    sched.Gather(B);
```
Scheduler sched;

typedef Window2D<T,1,WRAP,ILPX,ILPY> Win2D;
typedef StructuredInjective<T,2,ILPX,ILPY> SMat;

// Define data structures to be used
Matrix<T> A (width, height), B (width, height);

// Use existing host buffers as matrices
A.Bind(hbuffer_A);
B.Bind(hbuffer_B);

// Analyze memory access patterns for allocation
sched.AnalyzeCall(Win2D(A), SMat(B));
sched.AnalyzeCall(Win2D(B), SMat(A));

// Invoke the kernels
for (int i = 0; i < iterations; ++i)
    sched.Invoke(GameOfLifeTick,
        Win2D((i % 2) ? B : A),
        SMat((i % 2) ? A : B));

// Gather processed data back to host
if ((iterations % 2) == 0)
    sched.Gather(A);
else
    sched.Gather(B);

Gather

GPU 1
A B
GPU 2
A B
GPU 3
A B
GPU 4
A B

hbuffer_A

hbuffer_B
template <typename T, int ILPX, int ILPY>
__global__ void GameOfLifeTick MAPS_MULTIDEF(
    Window2D<T,1,WRAP,ILPX,ILPY> current_gen,
    StructuredInjective<T,2,ILPX,ILPY> next_gen) {

    MAPS_MULTI_INIT(current_gen, next_gen);

    #pragma unroll
    MAPS_FOREACH(nextgen_iter, next_gen) {
        int live_neighbors = 0, is_live = 0;

        #pragma unroll
        MAPS_FOREACH_ALIGNED(iter, current_gen, nextgen_iter) {
            // Set variables according to the rules
            if (iter.index() == 4) is_live = *iter;
            else live_neighbors += *iter;
        }
        int result = GameOfLifeConditions(...);
        *nextgen_iter = result;
    }
    next_gen.commit();
}
template <typename T, int ILPX, int ILPY>
__global__ void GameOfLifeTick MAPS_MULTIDEF(
    Window2D<T,1,WRAP,ILPX,ILPY> current_gen,
    StructuredInjective<T,2,ILPX,ILPY> next_gen) {

    MAPS_MULTI_INIT(current_gen, next_gen);

    #pragma unroll
    MAPS_FOREACH(nextgen_iter, next_gen) {
        int live_neighbors = 0, is_live = 0;
        #pragma unroll
        MAPS_FOREACH_ALIGNED(iter, current_gen, nextgen_iter) {
            // Set variables according to the rules
            if (iter.index() == 4) is_live = *iter;
            else live_neighbors += *iter;
        }
        int result = GameOfLifeConditions(...);
        *nextgen_iter = result;
    }
    next_gen.commit();
}
template <typename T, int ILPX, int ILPY>
__global__ void GameOfLifeTick MAPS_MULTIDEF(
Window2D<T,1,WRAP,ILPX,ILPY> current_gen,
StructuredInjective<T,2,ILPX,ILPY> next_gen) {

MAPS_MULTI_INIT(current_gen, next_gen);

#pragma unroll
MAPS_FOREACH(nextgen_iter, next_gen) {
    int live_neighbors = 0, is_live = 0;

    #pragma unroll
    MAPS_FOREACH_ALIGNED(iter, current_gen, nextgen_iter) {
        // Set variables according to the rules
        if (iter.index() == 4) is_live = *iter;
        else live_neighbors += *iter;
    }
    int result = GameOfLifeConditions(...);
    *nextgen_iter = result;
}
next_gen.commit();
}
Code Sample – Device

```cpp
#include <device.h>

template<typename T, int ILPX, int ILPY>
__global__ void GameOfLifeTick MAPS_MULTIDEF(
    Window2D<T,1,WRAP,ILPX,ILPY> current_gen,
    StructuredInjective<T,2,ILPX,ILPY> next_gen) {
    MAPS_MULTI_INIT(current_gen, next_gen);

    #pragma unroll
    MAPS_FOREACH(nextgen_iter, next_gen) {
        int live_neighbors = 0, is_live = 0;
        #pragma unroll
        MAPS_FOREACH_ALIGNED(iter, current_gen, nextgen_iter) {
            // Set variables according to the rules
            if (iter.index() == 4) is_live = *iter;
            else live_neighbors += *iter;
        }
        int result = GameOfLifeConditions(...);
        *nextgen_iter = result;
    }
    next_gen.commit();
}
```

Global Memory

Thread (1,1)
Code Sample – Device

```
template <typename T, int ILPX, int ILPY>
__global__ void GameOfLifeTick MAPS_MULTIDEF(
    Window2D<T,1,WRAP,ILPX,ILPY> current_gen,
    StructuredInjective<T,2,ILPX,ILPY> next_gen) {

    MAPS_MULTI_INIT(current_gen, next_gen);

    #pragma unroll
    MAPS_FOREACH(nextgen_iter, next_gen) {
        int live_neighbors = 0, is_live = 0;

        #pragma unroll
        MAPS_FOREACH_ALIGNED(iter, current_gen, nextgen_iter) {
            // Set variables according to the rules
            if (iter.index() == 4) is_live = *iter;
            else live_neighbors += *iter;
        }
        int result = GameOfLifeConditions(...);
        *nextgen_iter = result;
    }
    next_gen.commit();
}
```
```cpp
template<typename T, int ILPX, int ILPY>
__global__ void GameOfLifeTick(MAPS_MULTIDEF<
    Window2D<T,1,WRAP,ILPX,ILPY> current_gen,
    StructuredInjective<T,2,ILPX,ILPY> next_gen>) {

    MAPS_MULTI_INIT(current_gen, next_gen);

    #pragma unroll
    MAPS_FOREACH(nextgen_iter, next_gen) {
        #pragma unroll
        MAPS_FOREACH_ALIGNED(iter, current_gen, nextgen_iter) {
            // Set variables according to the rules
            if (iter.index() == 4) is_live = *iter;
            else live_neighbors += *iter;
        }
        int result = GameOfLifeConditions(...);
        *nextgen_iter = result;
    }
    next_gen.commit();
}
```
Code Sample – Device

```cpp
template <typename T, int ILPX, int ILPY>
__global__ void GameOfLifeTick MAPS_MULTIDEF(
    Window2D<T,1,WRAP,ILPX,ILPY> current_gen,
    StructuredInjective<T,2,ILPX,ILPY> next_gen) {

    MAPS_MULTI_INIT(current_gen, next_gen);

    #pragma unroll
    MAPS_FOREACH(nextgen_iter, next_gen) {
        int live_neighbors = 0, is_live = 0;

        #pragma unroll
        MAPS_FOREACH_ALIGNED(iter, current_gen, nextgen_iter) {
            // Set variables according to the rules
            if (iter.index() == 4) is_live = *iter;
            else live_neighbors += *iter;
        }
        int result = GameOfLifeConditions(...);
        *nextgen_iter = result;
    }

    next_gen.commit();
}
```

Global Memory
Performance

- MAPS-Multi was tested on four different GPUs:
  - GTX 780 (Kepler), Titan Black (Kepler), Tesla K40m (Kepler), GTX 980 (Maxwell)
- Near linear scaling, up to 3.94× on 4 GPUs:
Performance vs. Optimized GPU Libraries

- Performance is on-par with production-level libraries
- Comparison with NVIDIA’s CUBLAS-XT (left) and CUB (right):

Matrix Multiplication

- Speedup
- Throughput [images/s]

(All multi-GPU results over MAPS-Multi)
Real-World Applications – Deep Learning

- Multi-GPU nodes are the platform of choice for deep neural network training

- Complex, multi-stage iterative process with many access patterns

- Two leading deep learning frameworks utilize GPUs by default: **Caffe** and **Torch**
  - Caffe has its own CPU/GPU memory manager (only for single GPU)
  - Torch uses a scripting engine, multi-GPU specifically partitioned by users
  - Both comprised of large codebases
Forward Propagation – Compute $f$

Backpropagation – Compute $\nabla f$
Deep Learning Performance

• Deep learning was implemented to train a digit classifier (LeNet)
• Implementation is compared with the Caffe and Torch frameworks:
Non-Negative Matrix Factorization

- Used for dimensionality reduction, finds matrices $W, H$ s.t. $V \approx WH$

- **NMF-mGPU** is a multi-GPU implementation of NMF
  - $\sim 15,000$ lines-of-code over multiple files
  - Specifically tailored for nodes with multiple Kepler GPUs
  - Uses MPI for inter-GPU communication

- The MAPS-Multi version consists of 870 lines-of-code

---

NMF Performance

GTX 780

Throughput [iterations / s] vs GPUs

Titan Black

Throughput [iterations / s] vs GPUs

GTX 980

Throughput [iterations / s] vs GPUs

- MAPS-Multi
- NMF-mGPU
Conclusions

Memory access pattern specification:

• Is a promising approach for ease-of-programming and improved performance
• Can dramatically reduce per-architecture GPU kernel tuning
• Alleviates manual device management
• Valid for many parallel algorithms
• Performance surpasses existing multi-GPU implementations
Future Research

• Explore additional memory access patterns

• Implications on power efficiency and resilience

• Other architectures (e.g., multi-core CPUs), clusters

• Compiler support for simplification, optimizations and automatic analysis

• Kernel workload estimation (for reordering and pipeline optimizations)
MAPS-Multi is open-source (New BSD license). The code is provided at:
http://www.cs.huji.ac.il/project/maps/

Thank You
Questions?

This research was supported by the Ministry of Science and Technology, Israel and by the German Research Foundation (DFG) Priority Program “Software for Exascale Computing” (SPP-EXA), research project FFKM.