Implementing and Tuning Irregular Programs on GPUs

Martin Burtscher
Computer Science
Texas State University-San Marcos

Rupesh Nasre
Computational Engineering and Sciences
The University of Texas at Austin
GPU Advantages over CPUs

- Comparing GTX 680 to Xeon E5-2687W
  - Both released in March 2012
- Peak performance and main-memory bandwidth
  - 8x as many operations executed per second
  - 4x as many bytes transferred per second
- Cost-, energy-, and area-efficiency
  - 29x as much performance per dollar
  - 6x as much performance per watt
  - 11x as much performance per area
Codes Suitable for GPU Acceleration

- Clearly, we should be using GPUs all the time
  - So why aren’t we?

- GPUs can only accelerate some types of codes
  - Need lots of parallelism, data reuse, and regularity (in memory accesses and control flow)

- Mostly regular codes have been ported to GPUs
  - E.g., matrix codes executing many ops/word
    - Dense matrix operations (Level 2 and 3 BLAS)
    - Stencil codes (PDE solvers)
Codes Suitable for GPU Acceleration

- Clearly, we should be using GPUs all the time
  - So why aren’t we?
- GPUs can only accelerate some types of codes
  - Need lots of parallelism, data reuse, and regularity (in memory accesses and control flow)
- Mostly regular codes have been ported to GPUs
  - E.g., matrix codes executing many ops/word
    - Dense matrix operations (Level 2 and 3 BLAS)
    - Stencil codes (PDE solvers)

Our goal is to also handle irregular codes well
Regular Programs

- Typically operate on arrays and matrices
  - Data is processed in fixed-iteration FOR loops
- Have statically predictable behavior
  - Exhibit mostly strided memory access patterns
  - Control flow is mainly determined by input size
  - Data dependencies are static and not loop carried

Example

```c
for (i = 0; i < size; i++) {
    c[i] = a[i] + b[i];
}
```
Irregular Programs

- Are important and widely used
  - Social network analysis, data clustering/partitioning, discrete-event simulation, operations research, meshing, SAT solving, \( n \)-body simulation, etc.

- Typically operate on **dynamic** data structures
  - Graphs, trees, linked lists, priority queues, etc.
  - Data is processed in variable-iteration WHILE loops
Irregular Programs (cont.)

- Have statically **unpredictable** behavior
  - Exhibit pointer-chasing memory access patterns
  - Control flow depends on input *values* and may change
  - Data dependences have to be detected dynamically

- Example

```c
while (pos != end) {
    v = workset[pos++];
    for (i = 0; i < count[v]; i++){
        n = neighbor[index[v] + i];
        if (process(v, n)) workset[end++] = n;
    }
}
```
GPU Implementation Challenges

- Indirect and irregular memory accesses
  - Little or no coalescing [low bandwidth]
- Memory-bound pointer chasing
  - Little locality and computation [exposed latency]
- Dynamically changing irregular control flow
  - Thread divergence [loss of parallelism]
- Input dependent and changing data parallelism
  - Load imbalance [loss of parallelism]

Naïve implementation results in poor performance
Outline

- Introduction
- Irregular codes and basic optimizations
- Irregular optimizations
- Performance results
- General guidelines
- Summary
Our Irregular Programs

- Mostly taken from LonestarGPU benchmark suite
  - Set of currently seven irregular CUDA codes

http://iss.ices.utexas.edu/?p=projects/galois/lonestargpu

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
<th>Lines of Code</th>
<th>Number of kernels</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFS</td>
<td>Breadth-first search</td>
<td>275</td>
<td>2</td>
</tr>
<tr>
<td>BH</td>
<td>Barnes-Hut $n$-body simulation</td>
<td>1,069</td>
<td>9</td>
</tr>
<tr>
<td>DMR</td>
<td>Delaunay mesh refinement</td>
<td>958</td>
<td>4</td>
</tr>
<tr>
<td>MST</td>
<td>Minimum spanning tree</td>
<td>446</td>
<td>8</td>
</tr>
<tr>
<td>PTA</td>
<td>Points-to analysis</td>
<td>3,494</td>
<td>40</td>
</tr>
<tr>
<td>SP</td>
<td>Survey propagation</td>
<td>445</td>
<td>3</td>
</tr>
<tr>
<td>SSSP</td>
<td>Single-source shortest paths</td>
<td>614</td>
<td>2</td>
</tr>
</tbody>
</table>
Primary Data Structures

- Road networks
  - BFS, Max-flow, MST, and SSSP
- Triangulated meshes
  - DMR
- Control-flow graphs
  - PTA
- Unbalanced octrees
  - BH
- Bi-partite graphs
  - SP
Basic CUDA Code Optimizations

- Our codes include many conventional optimizations
  - Use structure of arrays, keep data on GPU, improve memory layout, cache data in registers, re-compute instead of re-load values, unroll loops, chunk work, employ persistent threads, and use compiler flags
- Conventional parallelization optimizations
  - Use light-weight locking, atomics, and lock-free code; minimize locking, memory fences, and volatile accesses
- Conventional architectural optimizations
  - Utilize shared memory, constant memory, streams, thread voting, and rsqrtf; detect compute capability and number of SMs; tune thread count, blocks per SM, launch bounds, and L1 cache/shared-memory configuration
Outline

- Introduction
- Irregular codes and basic optimizations
- Irregular optimizations
- Performance results
- General guidelines
- Summary
## Irregular CUDA Code Optimizations

<table>
<thead>
<tr>
<th>Category</th>
<th>Optimization</th>
<th>Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Synchronization</strong></td>
<td>Push versus pull</td>
<td>BFS, PTA, SSSP</td>
</tr>
<tr>
<td></td>
<td>Combined memory fences</td>
<td>BH-summ, BH-tree</td>
</tr>
<tr>
<td></td>
<td>Wait-free pre-pass</td>
<td>BH-summ</td>
</tr>
<tr>
<td><strong>Memory</strong></td>
<td>Kernel unrolling</td>
<td>BFS, SSSP</td>
</tr>
<tr>
<td></td>
<td>Warp-centric execution</td>
<td>BH-force, PTA</td>
</tr>
<tr>
<td></td>
<td>Computation onto traversal</td>
<td>BH-sort, BH-summ</td>
</tr>
<tr>
<td><strong>Algorithm</strong></td>
<td>Algorithm choice</td>
<td>Max-flow, SSSP</td>
</tr>
<tr>
<td></td>
<td>Implementation adaptation</td>
<td>MST, SSSP</td>
</tr>
<tr>
<td></td>
<td>Algebraic properties</td>
<td>BFS, SP, SSSP</td>
</tr>
<tr>
<td><strong>Scheduling</strong></td>
<td>Sorting work</td>
<td>BH-force, DMR, PTA</td>
</tr>
<tr>
<td></td>
<td>Dynamic configuration</td>
<td>DMR, PTA</td>
</tr>
<tr>
<td></td>
<td>Kernel fusion and fission</td>
<td>DMR, MST, SP</td>
</tr>
<tr>
<td></td>
<td>Communication onto computation</td>
<td>BH, PTA</td>
</tr>
</tbody>
</table>
## Push versus Pull

<table>
<thead>
<tr>
<th>Synchronization</th>
<th>Push versus pull</th>
<th>BFS, PTA, SSSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Combined memory fences</td>
<td></td>
<td>BH-summ, BH-tree</td>
</tr>
<tr>
<td>Wait-free pre-pass</td>
<td></td>
<td>BH-summ</td>
</tr>
</tbody>
</table>

- Information may flow in a push- or pull-based way
  - Push-based data-driven code is **work-efficient**
  - Pull-based code **minimizes synchronization**
- In BFS and SSSP, we use a push-based model
- In PTA, we use a pull-based model
  - Similar to scatter/gather

Implementing and Tuning Irregular Programs on GPUs
Combined Memory Fences

- Memory fence operations can be slow
  - Combining can help even with some load imbalance
- In BH-tree, the block-threads create subtrees, wait at a `syncthreads`, and install the subtrees
- In BH-summ, the block-threads summarize child info, run a `syncthreads`, then update the parents

<table>
<thead>
<tr>
<th>Synchronization</th>
<th>Push versus pull</th>
<th>BFS, PTA, SSSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Combined memory fences</td>
<td></td>
<td>BH-summ, BH-tree</td>
</tr>
<tr>
<td>Wait-free pre-pass</td>
<td></td>
<td>BH-summ</td>
</tr>
</tbody>
</table>
Wait-free Pre-pass

<table>
<thead>
<tr>
<th>Synchronization</th>
<th>Push versus pull</th>
<th>BFS, PTA, SSSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Combined memory fences</td>
<td></td>
<td>BHsumm, BH-tree</td>
</tr>
<tr>
<td>Wait-free pre-pass</td>
<td></td>
<td>BH-summ</td>
</tr>
</tbody>
</table>

- Use pre-pass to process ready consumers first
  - Instead of waiting, skip over unready consumers
  - Follow up with a ‘waitful’ pass in the end
- BH uses multiple pre-passes in summarization
  - Due to high fan-out of octree, most of work is done in pre-passes, thus minimizing the waiting in final pass
Kernel Unrolling

- **Kernel unrolling**
  - Instead of operating on a single graph element, a thread operates on a (connected) subgraph
  - Improves *locality* and *latency-hiding* ability
  - Helps *propagate* information *faster* through graph

- **Useful for memory-bound kernels like BFS & SSSP**

<table>
<thead>
<tr>
<th>Memory</th>
<th>Kernel unrolling</th>
<th>BFS, SSSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Warp-centric execution</td>
<td>BH-force, PTA</td>
<td></td>
</tr>
<tr>
<td>Computation onto traversal</td>
<td>BH-sort, BH-summ</td>
<td></td>
</tr>
</tbody>
</table>
Warp-centric Execution

- Forcing warps to stay synchronized in irreg code
  - Eliminates divergence and enables coalescing
- BH traverses entire warp’s union of tree prefixes
  - Divergence-free traversal & only need per-warp data
- PTA uses lists with 32 words per element
  - Enables fast set union & fully coalesced data accesses
Computation onto Traversal

- Combines multiple passes over data structure
- In BH-sort, top-down traversal sorts bodies and moves non-null children to front of child-array
- In BH-summ, bottom-up traversal computes center of gravity and counts number of bodies in subtrees
Algorithm Choice

- Best algorithm choice is architecture dependent
- For Max-flow, push-relabel algorithm is better for CPUs whereas MPM works better on GPUs
- For SSSP, Bellman-Ford algorithm is preferential on GPUs whereas Dijkstra’s algorithm yields better serial performance on CPUs
Implementation Adaptation

- Implementation may have to be adjusted for GPU
  - **Array-based** graph representations
  - **Topology**- instead of data-driven implementation
- For MST, do not explicitly contract edges
  - Keep graph static and record which nodes are merged
- For SSSP, use topology-driven implementation
  - Avoided synchronization outweighs work inefficiency

<table>
<thead>
<tr>
<th>Algorithmic</th>
<th>Algorithm choice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implementation adaptation</td>
<td>MST, SSSP</td>
</tr>
<tr>
<td>Algebraic properties</td>
<td>BFS, SP, SSSP</td>
</tr>
</tbody>
</table>
**Algebraic Properties**

- Exploiting monotonicity property
  - Enables **atomic-free** topology-driven implementation
- In BFS and SSSP, the node distances decrease monotonically; in SP, the product of probabilities decrease monotonically.
Sorting Work

- Each warp-thread should do similar work
  - Sorting minimizes divergence and load imbalance
- In BH, bodies are sorted by spatial location; in DMR, triangles are sorted by badness; in PTA, nodes are sorted by in-degrees

<table>
<thead>
<tr>
<th>Scheduling</th>
<th>Sorting work</th>
<th>BH-force, DMR, PTA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic configuration</td>
<td></td>
<td>DMR, PTA</td>
</tr>
<tr>
<td>Kernel fusion and fission</td>
<td></td>
<td>DMR, MST, SP</td>
</tr>
<tr>
<td>Communication onto computation</td>
<td></td>
<td>BH, PTA</td>
</tr>
</tbody>
</table>

Implementing and Tuning Irregular Programs on GPUs
Dynamic Configuration

- Computation should follow **parallelism profile**
  - Fewer threads = fewer conflicts & less sync overhead
- In DMR, the number of **aborts** due to conflicts reduces from 60% and 35% in the first two iterations to 1% and 4.4%
- **PTA** is similar

<table>
<thead>
<tr>
<th>Scheduling</th>
<th>Sorting work</th>
<th>BH-force, DMR, PTA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic configuration</td>
<td>DMR, PTA</td>
<td></td>
</tr>
<tr>
<td>Kernel fusion and fission</td>
<td>DMR, MST, SP</td>
<td></td>
</tr>
<tr>
<td>Communication onto</td>
<td>BH, PTA</td>
<td></td>
</tr>
<tr>
<td>computation</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Kernel Fusion and Fission

- Kernel fusion
  - Fast data-sharing and reduced invocation overhead
- Kernel fission
  - Enables individually tuned kernel configurations
- DMR uses fusion to transfer connectivity info
- MST uses fission for various computations

<table>
<thead>
<tr>
<th>Scheduling</th>
<th>Sorting work</th>
<th>BH-force, DMR, PTA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic configuration</td>
<td>DMR, PTA</td>
<td></td>
</tr>
<tr>
<td>Kernel fusion and fission</td>
<td>DMR, MST, SP</td>
<td></td>
</tr>
<tr>
<td>Communication onto computation</td>
<td>BH, PTA</td>
<td></td>
</tr>
</tbody>
</table>
## Communication onto Computation

<table>
<thead>
<tr>
<th>Scheduling</th>
<th>Sorting work</th>
<th>BH-force, DMR, PTA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Dynamic configuration</td>
<td>DMR, PTA</td>
</tr>
<tr>
<td></td>
<td>Kernel fusion and fission</td>
<td>DMR, MST, SP</td>
</tr>
<tr>
<td>Communication onto computation</td>
<td></td>
<td>BH, PTA</td>
</tr>
</tbody>
</table>

- **Overlap CPU/GPU transfer** with GPU computation
- In BH, the bodies’ positions can be sent to the CPU while the next time step starts running
- In PTA, points-to information is incrementally copied to the CPU while the GPU is computing more points-to information
Outline

- Introduction
- Irregular codes and basic optimizations
- Irregular optimizations
- Performance results
- General guidelines
- Summary
Experimental Methodology

- **Quadro 6000** GPU, 1.45 GHz, 6 GB, 448 PEs
- nvcc compiler v4.2 with -arch=sm_20 -O3 flags
- LonestarGPU program inputs
  - BFS, Max-flow, MST, SSSP: road networks, RMAT graphs, and random graphs
  - BH: random star clusters based on Plummer model
  - DMR: randomly generated 2D meshes
  - PTA: constraint graphs from open-source C/C++ progs
  - SP: randomly generated hard 3-SAT formulae
- Base case: ported optimized multi-core CPU code
- Best case: includes applicable irreg. optimizations
Irregular optimizations can be essential
- Performance improves by up to 2 orders of magnitude
Push versus Pull

1. **Synchronization**
   - Push versus pull
   - Combined memory fences
   - Wait-free pre-pass
   - BFS, PTA, SSSP
   - BHsumm, BH-tree
   - BH-summ

- **SSSP on various graphs**
  - Push (scatter) is better since it propagates info faster
  - Benefit is higher for denser RMAT graph
  - Both the push and pull codes are atomic-free
Combined Memory Fences

<table>
<thead>
<tr>
<th>Synchronization</th>
<th>Push versus pull</th>
<th>BFS, PTA, SSSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Combined memory fences</td>
<td></td>
<td>BH-summ, BH-tree</td>
</tr>
<tr>
<td>Wait-free pre-pass</td>
<td></td>
<td>BH-summ</td>
</tr>
</tbody>
</table>

- BH octree building on star clusters
  - Combining multiple memory fences into a single syncthreads
  - 37% to 44% faster despite barrier-induced waiting

Implementing and Tuning Irregular Programs on GPUs
Kernel Unrolling

- SSSP on various graphs
  - Increasing unroll factor improves computation-to-latency ratio
  - Almost halves running time for road networks

<table>
<thead>
<tr>
<th>Memory</th>
<th>Kernel unrolling</th>
<th>BFS, SSSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Warp-centric execution</td>
<td>BH-force, PTA</td>
<td></td>
</tr>
<tr>
<td>Computation onto traversal</td>
<td>BH-sort, BH-summ</td>
<td></td>
</tr>
</tbody>
</table>
Warp-centric Execution

<table>
<thead>
<tr>
<th>Memory</th>
<th>Kernel unrolling</th>
<th>BFS, SSSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Warp-centric execution</td>
<td>BH-force, PTA</td>
<td></td>
</tr>
<tr>
<td>Computation onto traversal</td>
<td>BH-sort, BH-summ</td>
<td></td>
</tr>
</tbody>
</table>

- BH force calculation on star clusters
  - Warp-based execution enables coalescing and iteration-stack sharing
  - About 8x speedup
Implementation Adaptation

<table>
<thead>
<tr>
<th>Algorithmic</th>
<th>Algorithm choice</th>
<th>Max-flow, SSSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implementation adaptation</td>
<td>MST, SSSP</td>
<td></td>
</tr>
<tr>
<td>Algebraic properties</td>
<td>BFS, SP, SSSP</td>
<td></td>
</tr>
</tbody>
</table>

- **SSSP on various graphs**
  - Work inefficient topology-driven implementation
  - Enables synchronization-free implementation
  - Outperforms data-driven
Algebraic Properties

3

<table>
<thead>
<tr>
<th>Algorithmic</th>
<th>Algorithm choice</th>
<th>Implementation adaptation</th>
<th>Algebraic properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max-flow, SSSP</td>
<td>MST, SSSP</td>
<td>BFS, SP, SSSP</td>
<td></td>
</tr>
</tbody>
</table>

- SP for hard 3-SAT inputs
  - Exploits monotonicity to avoid atomics
  - Performance benefit increases for larger problem sizes
Sorting Work

- BH force calculation and sorting on star clusters
  - Sorting reduces size of union of tree prefixes that warp threads traverse
  - 7.4x to 8.7x speedup

<table>
<thead>
<tr>
<th>Scheduling</th>
<th>Sorting work</th>
<th>BH-force, DMR, PTA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic configuration</td>
<td>DMR, PTA</td>
<td></td>
</tr>
<tr>
<td>Kernel fusion and fission</td>
<td>DMR, MST, SP</td>
<td></td>
</tr>
<tr>
<td>Communication onto computation</td>
<td>BH, PTA</td>
<td></td>
</tr>
</tbody>
</table>
Dynamic Configuration

- DMR on random mesh
  - Dynamic kernel configuration lowers abort ratio from 60% to 1% in first iteration
  - Overall performance improves by 14%

<table>
<thead>
<tr>
<th>Scheduling</th>
<th>Sorting work</th>
<th>BH-force, DMR, PTA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic configuration</td>
<td>DMR, PTA</td>
<td></td>
</tr>
<tr>
<td>Kernel fusion and fission</td>
<td>DMR, MST, SP</td>
<td></td>
</tr>
<tr>
<td>Communication onto computation</td>
<td>BH, PTA</td>
<td></td>
</tr>
</tbody>
</table>
GPU/CPU Performance Comparison

**Barnes Hut**

- Multi-core
- GPU

**Survey Propagation**

- Multi-core
- GPU

**Delaunay Mesh Refinement**

- Multi-core
- GPU

**Single Source Shortest Path**

- Multi-core
- GPU

**Inputs:**
- **BH**: 5M stars (Plummer model), 1 time step
- **DMR**: 10M triangles, 4.7M bad triangles
- **SP**: 1M literals, 4.2M clauses
- **SSSP**: 23M nodes, 57M edges
Outline

- Introduction
- Irregular codes and basic optimizations
- Irregular optimizations
- Performance results
- General guidelines
- Summary
CPU/GPU Implementation Differences

- **Irregular CPU code**
  - Dynamically (incrementally) allocated data structures
  - Structure-based data structures
  - Logical lock-based implementation
  - Global/local worklists
  - Recursive or iterative implementation

- **Irregular GPU code**
  - Statically (wholly) allocated data structures
  - Multiple-array-based data structures
  - Lock-free implementation
  - Local or no worklists
  - Iterative implementation
Exploit Irregularity-friendly Features

- Massive multithreading
  - Ideal for hiding latency of irregular memory accesses
- Wide parallelism
  - Great for exploiting large amounts of parallelism
- Shared memory
  - Useful for local worklists
  - Fast data sharing
- Fast thread startup
  - Essential when launching thousands of threads
- HW support for reduction and synchronization
  - Makes otherwise costly operations very fast
- Lockstep execution
  - Can share data without explicit synchronization
  - Allows to consolidate iteration stacks
- Coalesced accesses
  - Access combining is useful in irregular codes
Traits of Efficient Irregular GPU Codes

- **Mandatory**
  - Need vast amounts of data parallelism
  - Can do large chunks of computation on GPU

- **Very Important**
  - Cautious implementation
  - DS can be expressed through fixed arrays
  - Uses local (or implicit) worklists that can be statically populated

- **Important**
  - Scheduling is independent of previous activities
  - Easy to sort activities by similarity (if needed)

- **Beneficial**
  - Easy to express iteratively
  - Has statically known range of neighborhoods
  - DS size (or bound) can be determined based on input
Mandatory Traits

- Need vast amounts of data parallelism
  - Small inputs are not worth running on the GPU
  - Many ordered algorithms do not qualify

- Can do large chunks of computation on GPU
  - Time-intensive parts of algorithm must run on GPU
  - Avoids frequent CPU-GPU memory transfers
Very Important Traits

- **Cautious** implementation
  - No rollback mechanism is needed to handle conflicts
- Data struct can be expressed through **fixed arrays**
  - Avoids slow dynamic memory allocation
  - Guarantees contiguous memory
- Uses **local** (or implicit) **worklists** that can be statically populated
  - Avoids serialization point
  - Enables lock-free implementation
Important Traits

- **Scheduling is independent** of or can ignore previous activities
  - Unordered: all nodes active (once or always) or many nodes active (and easy to determine whether active)
  - Ordered: ordered by level, assigned to worklist in ‘correct’ order, poll until ready

- Easy to **approximately sort** activities by similarity
  - Reduces divergence and enables other optimizations
Beneficial Traits

- Easy to express *iteratively*
  - Recursion is not well supported

- Has statically known range of *neighborhoods*
  - Enables certain performance optimizations

- Data structure *size* (or upper bound) can be determined based on input
  - Allows one-time fixed allocation or over-allocation
Outline

- Introduction
- Irregular codes and basic optimizations
- Irregular optimizations
- Performance results
- General guidelines
- Summary
## Summary of Optimization Principles

<table>
<thead>
<tr>
<th>Category</th>
<th>Optimization</th>
<th>Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1 Synchronization</strong></td>
<td>Push versus pull</td>
<td>BFS, PTA, SSSP</td>
</tr>
<tr>
<td></td>
<td>Combined memory fences</td>
<td>BH-summ, BH-tree</td>
</tr>
<tr>
<td></td>
<td>Wait-free pre-pass</td>
<td>BH-summ</td>
</tr>
<tr>
<td><strong>2 Memory</strong></td>
<td>Kernel unrolling</td>
<td>BFS, SSSP</td>
</tr>
<tr>
<td></td>
<td>Warp-centric execution</td>
<td>BH-force, PTA</td>
</tr>
<tr>
<td></td>
<td>Computation onto traversal</td>
<td>BH-sort, BH-summ</td>
</tr>
<tr>
<td><strong>3 Algorithm</strong></td>
<td>Algorithm choice</td>
<td>Max-flow, SSSP</td>
</tr>
<tr>
<td></td>
<td>Implementation adaptation</td>
<td>MST, SSSP</td>
</tr>
<tr>
<td></td>
<td>Algebraic properties</td>
<td>BFS, SP, SSSP</td>
</tr>
<tr>
<td><strong>4 Scheduling</strong></td>
<td>Sorting work</td>
<td>BH-force, DMR, PTA</td>
</tr>
<tr>
<td></td>
<td>Dynamic configuration</td>
<td>DMR, PTA</td>
</tr>
<tr>
<td></td>
<td>Kernel fusion and fission</td>
<td>DMR, MST, SP</td>
</tr>
<tr>
<td></td>
<td>Communication onto computation</td>
<td>BH, PTA</td>
</tr>
</tbody>
</table>
Summary and Conclusion

- Presented 13 general optimization principles for efficiently implementing irregular GPU codes
  - Necessary because irregular algorithms can be very challenging to accelerate using GPUs
- Today’s GPUs can deliver high performance on many irregular codes and can accelerate them
- Code for GPU, do not merely adjust CPU code
  - Requires different data and code structures
  - Benefits from different optimizations
Further Reading on our Project

- GPU Optimization Principles for Irregular Programs, submitted
- Data-driven vs. Topology-driven Irregular Algorithms on GPUs, IPDPS 2013
- Atomic-free Irregular Computations on GPUs, GPGPU 2013
- Morph Algorithms on GPUs, PPoPP 2013
- A Quantitative Study of Irregular Programs on GPUs, IISWC 2012
- A GPU Implementation of Inclusion-based Points-to Analysis, PPoPP 2012
- An Efficient CUDA Implementation of the Tree-based Barnes-Hut n-Body Algorithm, GPU Computing Gems 2011
Acknowledgments

- Collaborators
  - Prof. Keshav Pingali – UT Austin
  - Molly O’Neil – Texas State University

- Funding and equipment
  - NSF grants 0833162, 1062335, 1111766, 1141022, 1216701, 1217231, and 1218568
  - NVIDIA Corporation
  - Qualcomm Corporation
  - Intel Corporation