Implementing and Tuning Irregular Programs on GPUs

Martin Burtscher

Computer Science

Texas State University-San Marcos

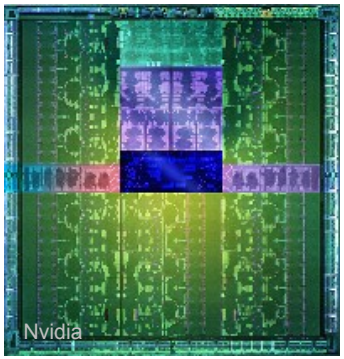
Rupesh Nasre

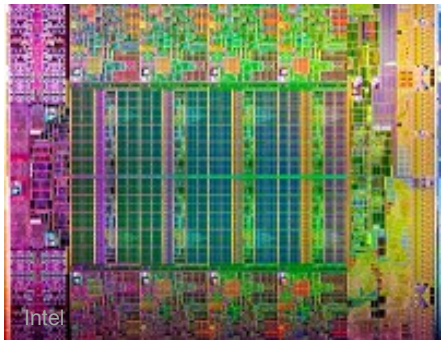
Computational Engineering and Sciences

The University of Texas at Austin

GPU Advantages over CPUs

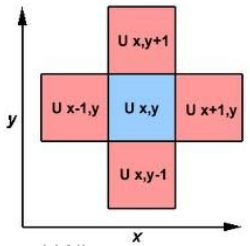
- Comparing GTX 680 to Xeon E5-2687W
  - Both released in March 2012
- Peak performance and main-memory bandwidth
  - 8x as many operations executed per second
  - 4x as many bytes transferred per second
- Cost-, energy-, and area-efficiency
  - 29x as much performance per dollar
  - 6x as much performance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

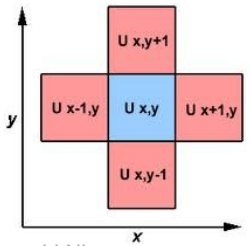


LLNL

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

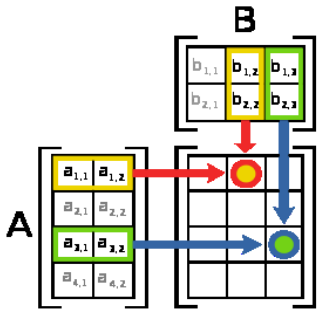


LLNL

Regular Programs

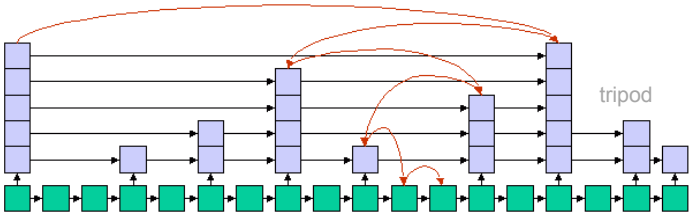
- Typically operate on arrays and matrices
  - Data is processed in fixed-iteration FOR loops
- Have statically predictable behavior
  - Exhibit mostly strided memory access patterns
  - Control flow is mainly determined by input *size*
  - Data dependencies are static and not loop carried
- Example

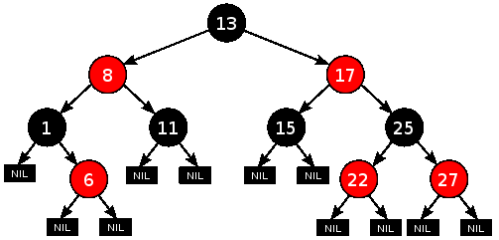
```
for (i = 0; i < size; i++) {
  c[i] = a[i] + b[i];
}
```



Irregular Programs

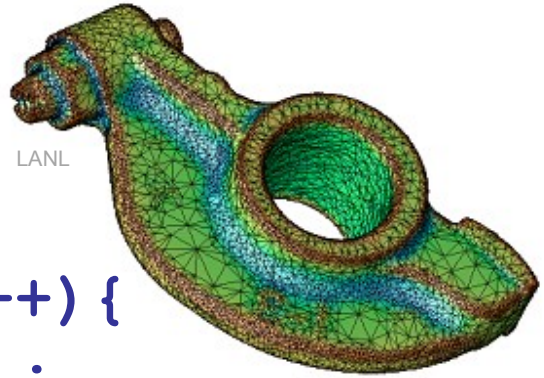
- Are important and widely used
  - Social network analysis, data clustering/partitioning, discrete-event simulation, operations research, meshing, SAT solving, *n*-body simulation, *etc*.
- Typically operate on dynamic data structures
  - Graphs, trees, linked lists, priority queues, *etc*.
  - Data is processed in variable-iteration WHILE loops





Irregular Programs (cont.)

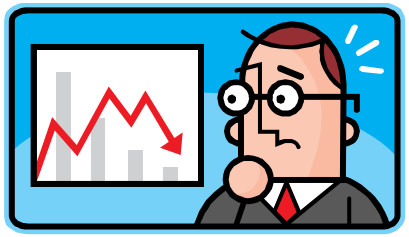
- Have statically unpredictable behavior
  - Exhibit pointer-chasing memory access patterns
  - Control flow depends on input *values* and may change
  - Data dependences have to be detected dynamically
- Example




```
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

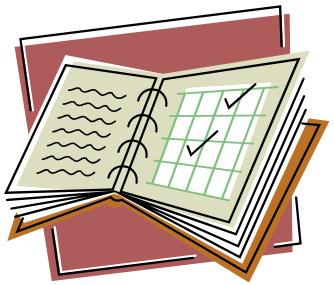
- Indirect and irregular memory accesses
  - Little or no coalescing [low bandwidth]
- Memory-bound pointer chasing
  - Little locality and computation [exposed latency]
- Dynamically changing irregular control flow
  - Thread divergence [loss of parallelism]
- Input dependent and changing data parallelism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

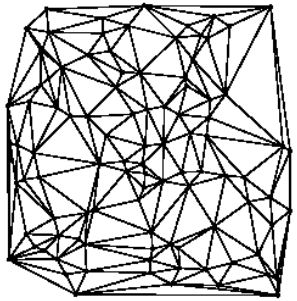
- Mostly taken from LonestarGPU benchmark suite
  - Set of currently seven irregular CUDA codes

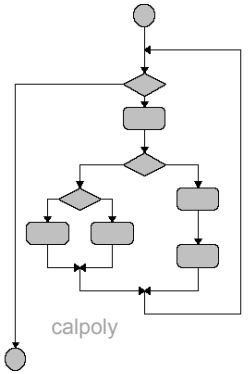
http://iss.ices.utexas.edu/?p=projects/galois/lonestargpu

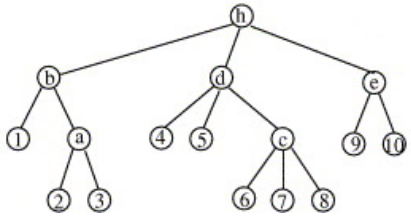
Benchmark	Description	Lines of Code	Number of kernels
BFS	Breadth-first search	275	2
BH	Barnes-Hut *n*-body simulation	1,069	9
DMR	Delaunay mesh refinement	958	4
MST	Minimum spanning tree	446	8
PTA	Points-to analysis	3,494	40
SP	Survey propagation	445	3
SSSP	Single-source shortest paths	614	2

Primary Data Structures

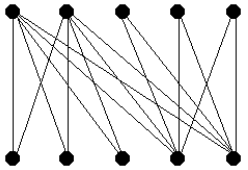
- Road networks
  - BFS, Max-flow, MST, and SSSP
- Triangulated meshes
  - DMR
- Control-flow graphs
  - PTA
- Unbalanced octrees
  - BH
- Bi-partite graphs
  - SP


wikipedia





sciencedirect

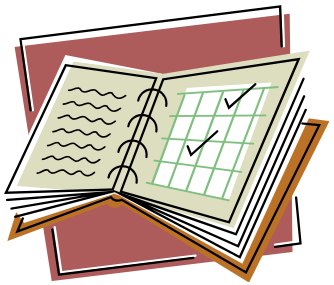

ucdenver

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

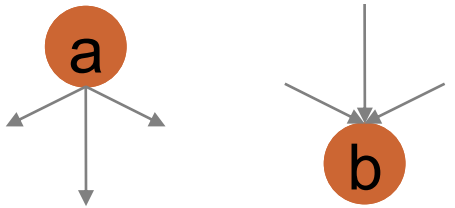
	Category	Optimization	Applications
1	**Synchronization**	Push versus pull	BFS, PTA, SSSP
		Combined memory fences	BH-summ, BH-tree
		Wait-free pre-pass	BH-summ
2	**Memory**	Kernel unrolling	BFS, SSSP
		Warp-centric execution	BH-force, PTA
		Computation onto traversal	BH-sort, BH-summ
3	**Algorithm**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

Push versus Pull

1

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

- 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



Combined Memory Fences

1

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

- 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

Wait-free Pre-pass

1

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

- 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

2

Memory	Kernel unrolling	BFS, SSSP
	Warp-centric execution	BH-force, PTA
	Computation onto traversal	BH-sort, BH-summ

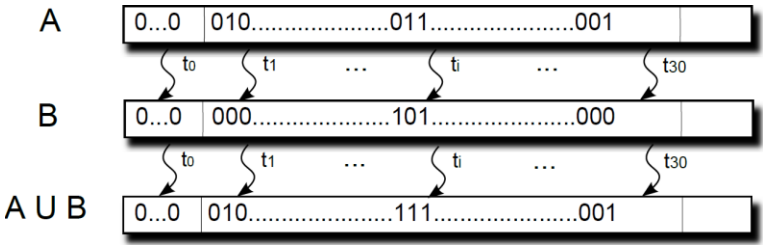
- Kernel unrolling
  - Instead of operating on a single graph element, a thread operates on a (connected) subgraph
  - Improves locality and latency-hiding ability
  - Helps propagate information faster through graph
- Useful for memory-bound kernels like BFS & SSSP

Warp-centric Execution

2

Memory	Kernel unrolling	BFS, SSSP
	Warp-centric execution	BH-force, PTA
	Computation onto traversal	BH-sort, BH-summ

- 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

2

Memory	Kernel unrolling	BFS, SSSP
	Warp-centric execution	BH-force, PTA
	Computation onto traversal	BH-sort, BH-summ

- 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

3

Algorithmic	Algorithm choice	Max-flow, SSSP
	Implementation adaptation	MST, SSSP
	Algebraic properties	BFS, SP, SSSP

- 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

3

Algorithmic	Algorithm choice	Max-flow, SSSP
	Implementation adaptation	MST, SSSP
	Algebraic properties	BFS, SP, SSSP

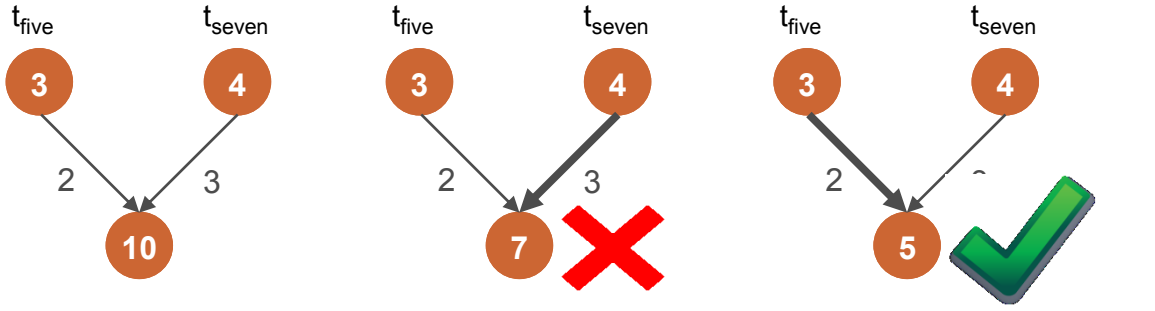
- Implementation may have to be adjusted for GPU
  - Array-based graph representations
  - Topology- instead of data-driven implementation
- For MST, do not explicitly contract edges
  - Keep graph static and record which nodes are merged
- For SSSP, use topology-driven implementation
  - Avoided synchronization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

Algebraic Properties

3

Algorithmic	Algorithm choice	Max-flow, SSSP
	Implementation adaptation	MST, SSSP
	Algebraic properties	BFS, SP, SSSP

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



Atomic-free update

Lost-update problem

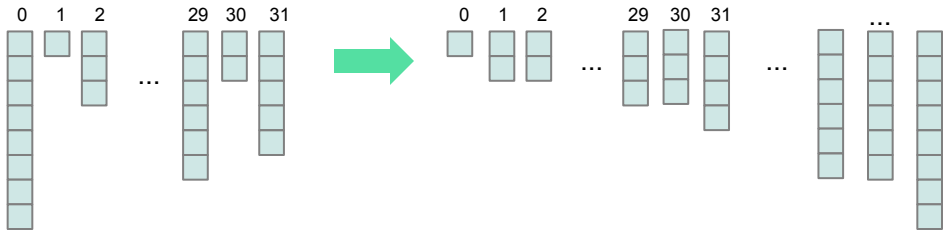
Correction by topology-driven processing, exploiting monotonicity

Sorting Work

4

Scheduling	Sorting work	BH-force, DMR, PTA
	Dynamic configuration	DMR, PTA
	Kernel fusion and fission	DMR, MST, SP
	Communication onto computation	BH, PTA

- 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

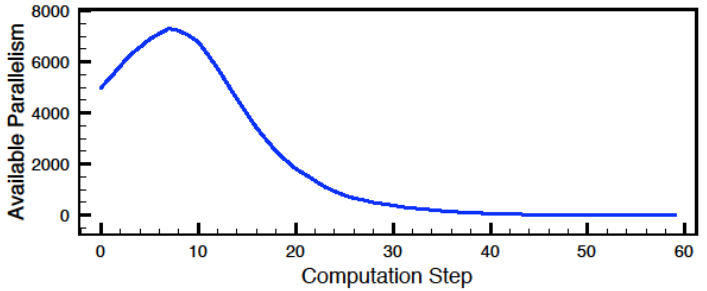


Dynamic Configuration

4

Scheduling	Sorting work	BH-force, DMR, PTA
	Dynamic configuration	DMR, PTA
	Kernel fusion and fission	DMR, MST, SP
	Communication onto computation	BH, PTA

- 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



Kernel Fusion and Fission

4

Scheduling	Sorting work	BH-force, DMR, PTA
	Dynamic configuration	DMR, PTA
	Kernel fusion and fission	DMR, MST, SP
	Communication onto computation	BH, PTA

- 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

Communication onto Computation

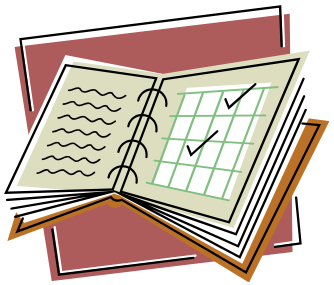
4

Scheduling	Sorting work	BH-force, DMR, PTA
	Dynamic configuration	DMR, PTA
	Kernel fusion and fission	DMR, MST, SP
	Communication onto computation	BH, PTA

- 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

Overall Speedup over Naïve Porting

Benchmark	Input	Speedup
BFS	RMAT20	**75.3**
	USA	**72.8**
BH	1M	**16.0**
	5M	**17.9**
DMR	1M	**55.1**
	10M	**74.4**
MST	RMAT20	**1.0**
	USA	**14.4**
PTA	vim	**1.1**
	tshark	**1.5**
SP	1M	**5.5**
	4M	**5.5**
SSSP	RMAT20	**297.4**
	USA	**196.8**

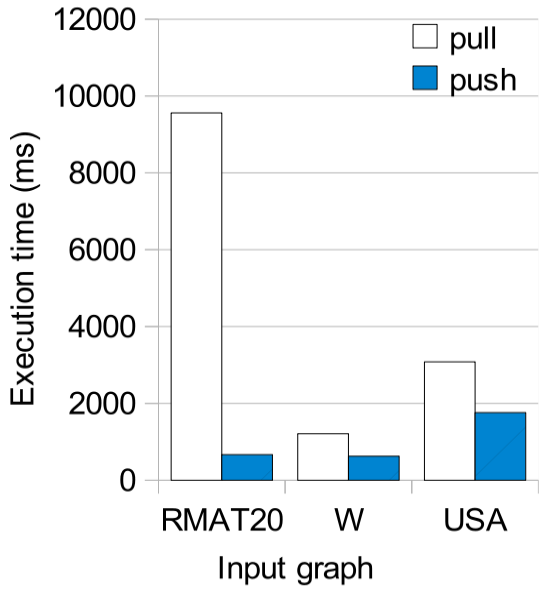
- Irregular optimizations can be essential
  - Performance improves by up to 2 orders of magnitude

Push versus Pull

1

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

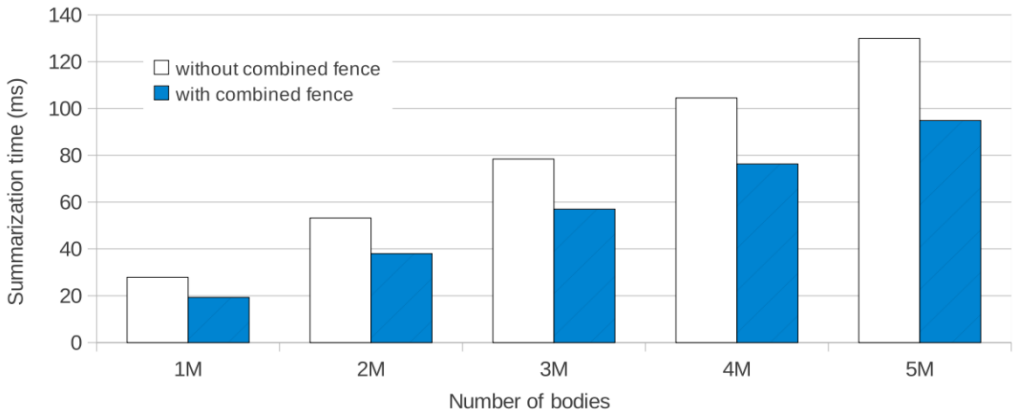


Combined Memory Fences

1

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

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

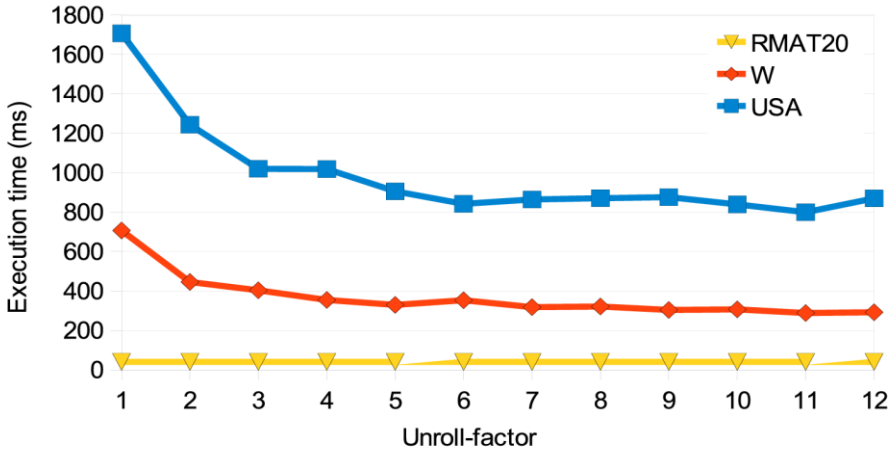


Kernel Unrolling

2

Memory	Kernel unrolling	BFS, SSSP
	Warp-centric execution	BH-force, PTA
	Computation onto traversal	BH-sort, BH-summ

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

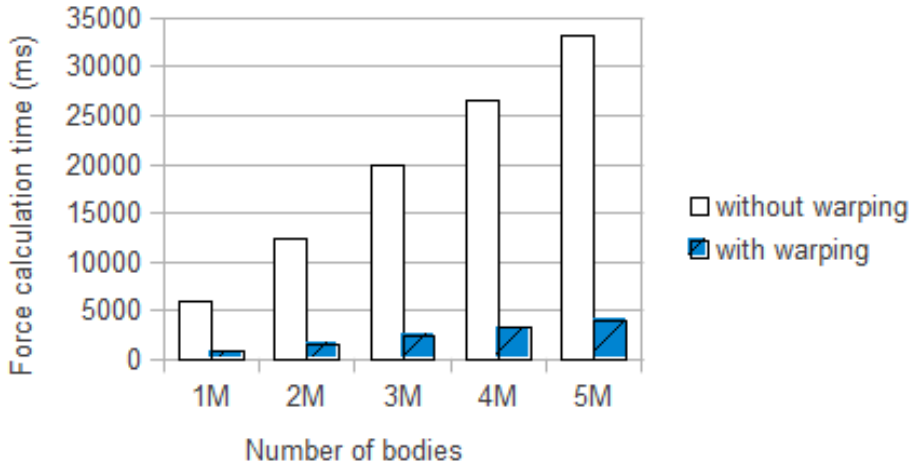


Warp-centric Execution

2

Memory	Kernel unrolling	BFS, SSSP
	Warp-centric execution	BH-force, PTA
	Computation onto traversal	BH-sort, BH-summ

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

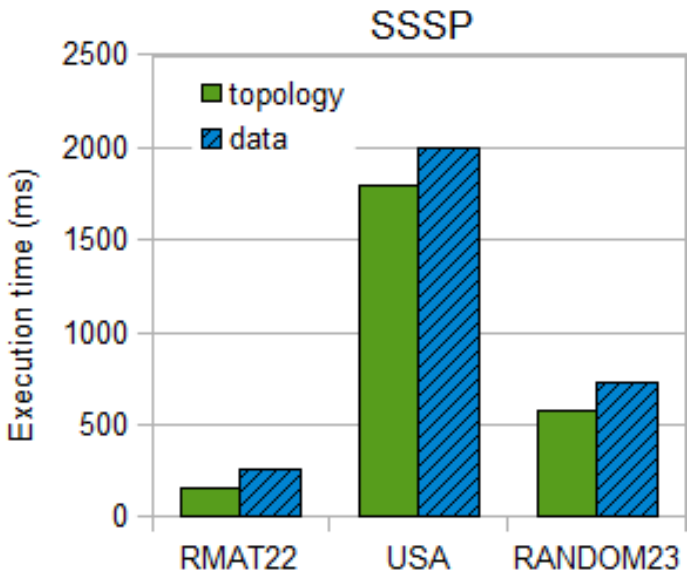


Implementation Adaptation

3

Algorithmic	Algorithm choice	Max-flow, SSSP
	Implementation adaptation	MST, SSSP
	Algebraic properties	BFS, SP, SSSP

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

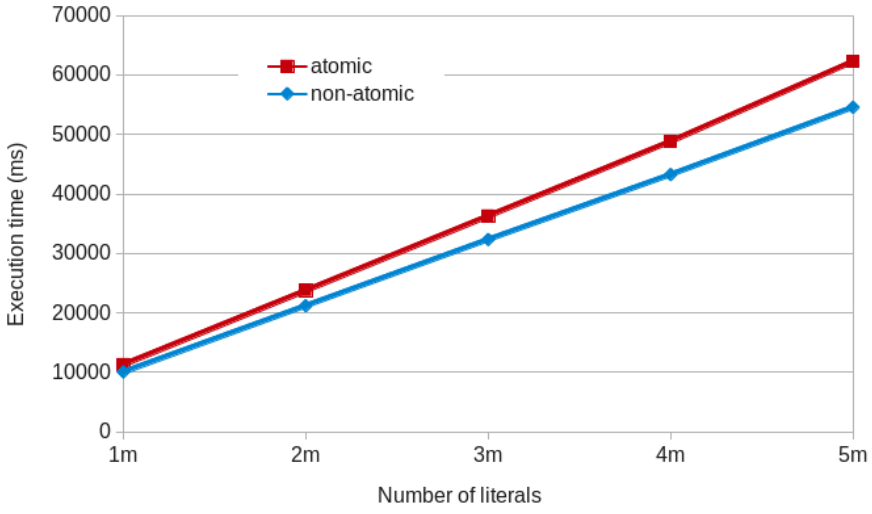


Algebraic Properties

3

Algorithmic	Algorithm choice	Max-flow, SSSP
	Implementation adaptation	MST, SSSP
	Algebraic properties	BFS, SP, SSSP

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

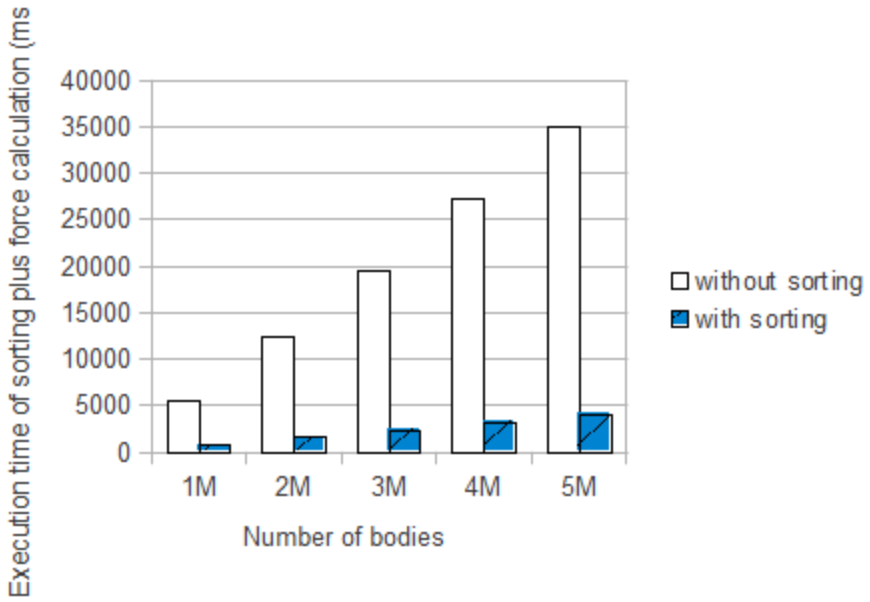


Sorting Work

4

Scheduling	Sorting work	BH-force, DMR, PTA
	Dynamic configuration	DMR, PTA
	Kernel fusion and fission	DMR, MST, SP
	Communication onto computation	BH, PTA

- 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

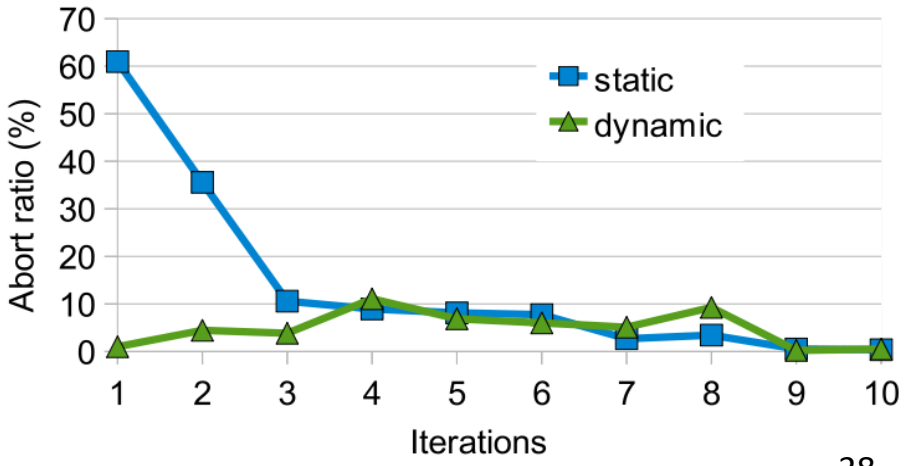


Dynamic Configuration

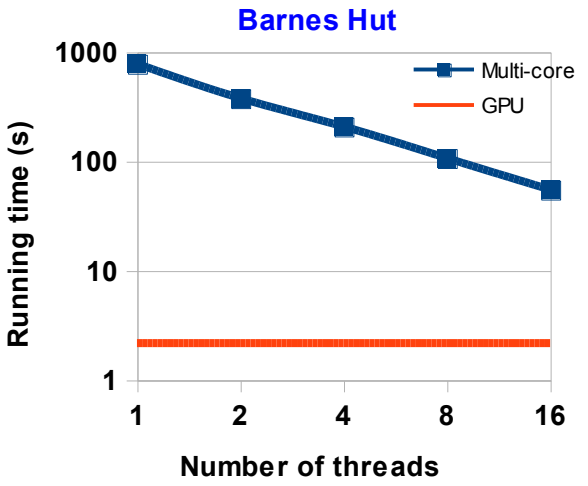
4

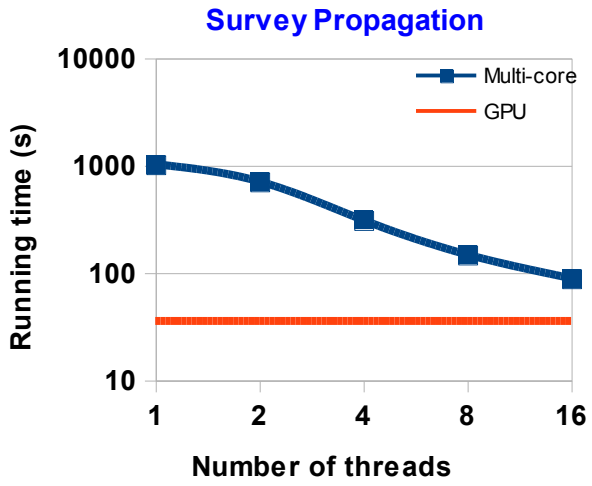
Scheduling	Sorting work	BH-force, DMR, PTA
	Dynamic configuration	DMR, PTA
	Kernel fusion and fission	DMR, MST, SP
	Communication onto computation	BH, PTA

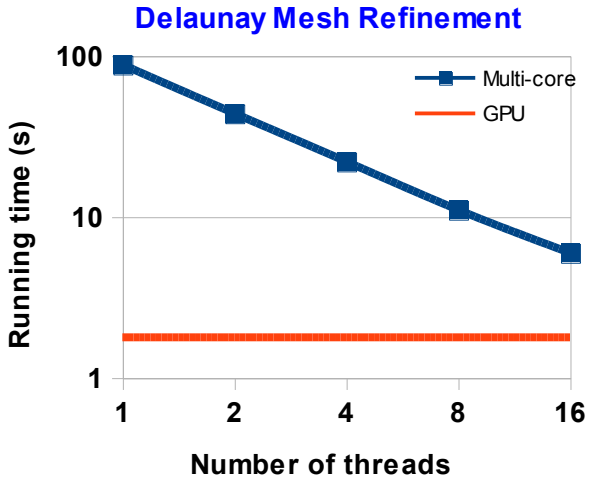
- DMR on random mesh
  - Dynamic kernel configuration lowers abort ratio from 60% to 1% in first iteration
  - Overall performance improves by 14%

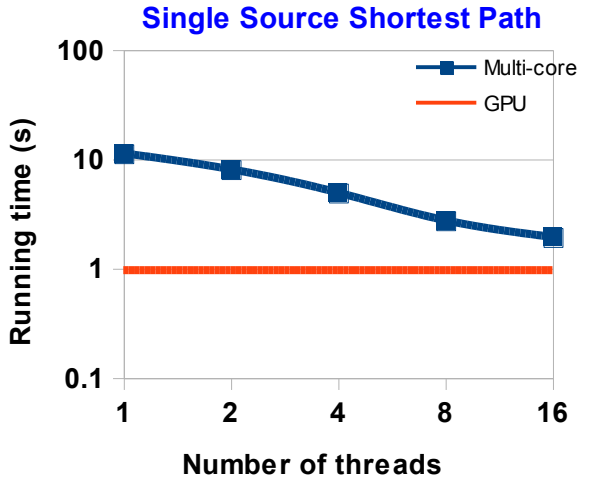


GPU/CPU Performance Comparison





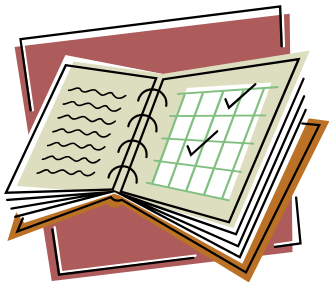




Inputs: BH : 5M stars (Plummer model), 1 time step
DMR : 10M triangles, 4.7M bad triangles
SP : 1M literals, 4.2M clauses
SSSP : 23M nodes, 57M edges

Outline

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



CPU/GPU Implementation Differences

- Irregular CPU code
  - Dynamically (increm.) allocated data struct
  - Structure-based data structures
  - Logical lock-based implementation
  - Global/local worklists
  - Recursive or iterative implementation
- Irregular GPU code
  - Statically (wholly) allocated data struct
  - 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 mem. 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

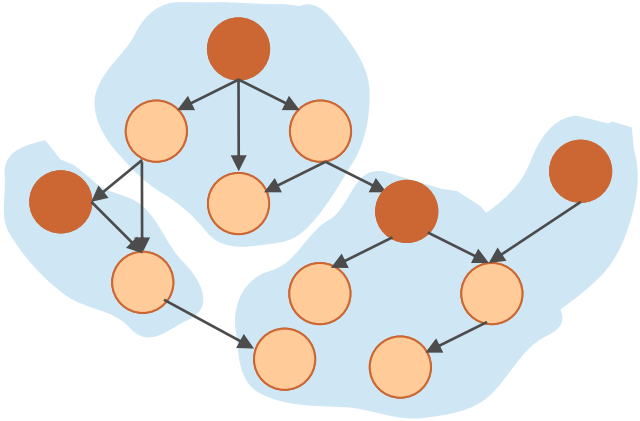
- Cautious implementation
  - No rollback mechanism is needed to handle conflicts
- Data struct can be expressed through fixed arrays
  - Avoids slow dynamic memory allocation
  - Guarantees contiguous memory
- Uses local (or implicit) worklists that can be statically populated
  - Avoids serialization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

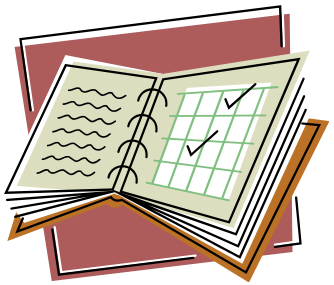
Beneficial Traits



- 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

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

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