



Automatic Compiler-Based Optimization of Graph Analytics for the GPU

Sreepathi Pai

The University of Texas at Austin

May 8, 2017

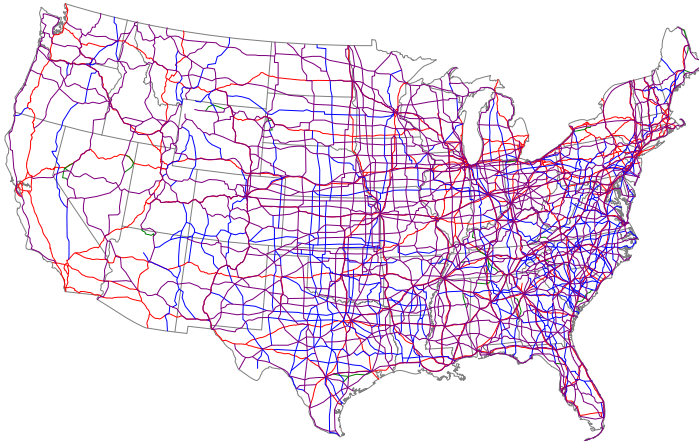
NVIDIA GTC

Parallel Graph Processing is not easy

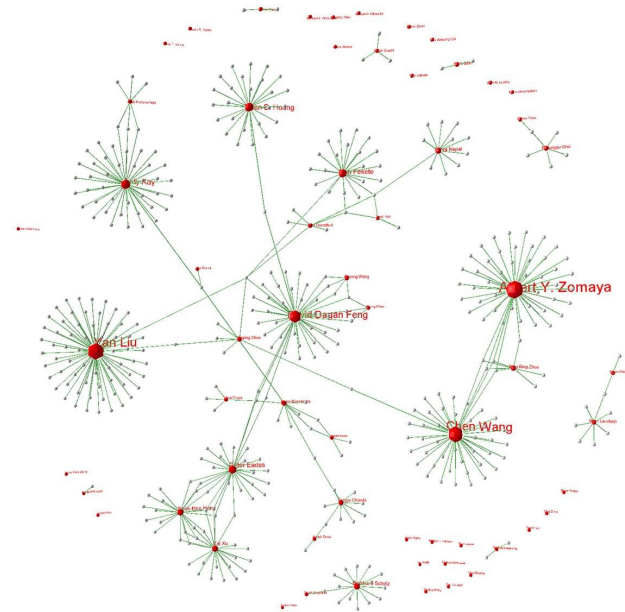
299ms

HD-BFS

84ms



USA Road Network
24M nodes, 58M edges



LiveJournal Social Network
5M nodes, 69M edges

692ms

LB-BFS

41ms



Observations from the “field”

- Different algorithms require different optimizations
 - BFS vs SSSP vs Triangle Counting
- Different inputs require different optimizations
 - Road vs Social Networks
- *Hypothesis:* High-performance graph analytics code must be customized for inputs and algorithms
 - No “one-size fits all” implementation
 - If true, we'll need a lot of code



How IrGL fits in

- IrGL is a language for graph algorithm kernels
 - *Slightly* higher-level than CUDA
- IrGL kernels are compiled to CUDA code
 - Incorporated into larger applications
- IrGL compiler applies 3 *throughput* optimizations
 - User can select exact combination
 - Yields multiple implementations of algorithm
- Let the compiler generate all the interesting variants!



Outline

- **IrGL Language**
- IrGL Optimizations
- Results



IrGL Constructs

- Representation for irregular data-parallel algorithms
- Parallelism
 - ForAll
- Synchronization
 - Atomic
 - Exclusive
- Bulk Synchronous Execution
 - Iterate
 - Pipe

IrGL Synchronization Constructs

- Atomic: Blocking atomic section

```
Atomic (lock) {  
    critical section  
}
```

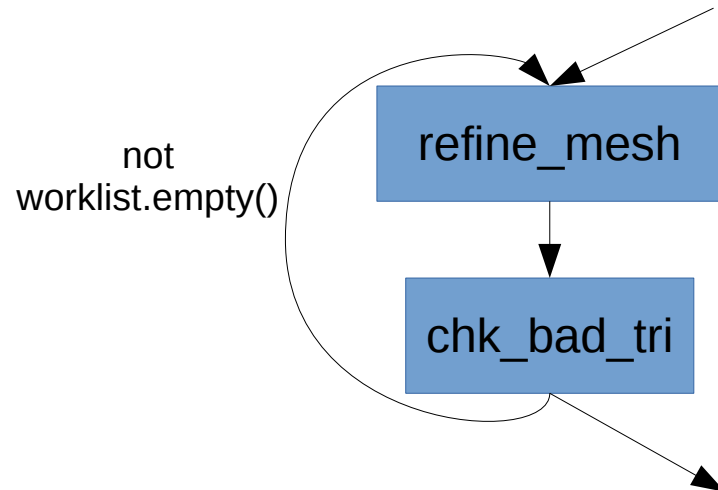
- Exclusive: Non-blocking, atomic section to obtain multiple locks with priority for resolving conflicts

```
Exclusive (locks) {  
    critical section  
}
```

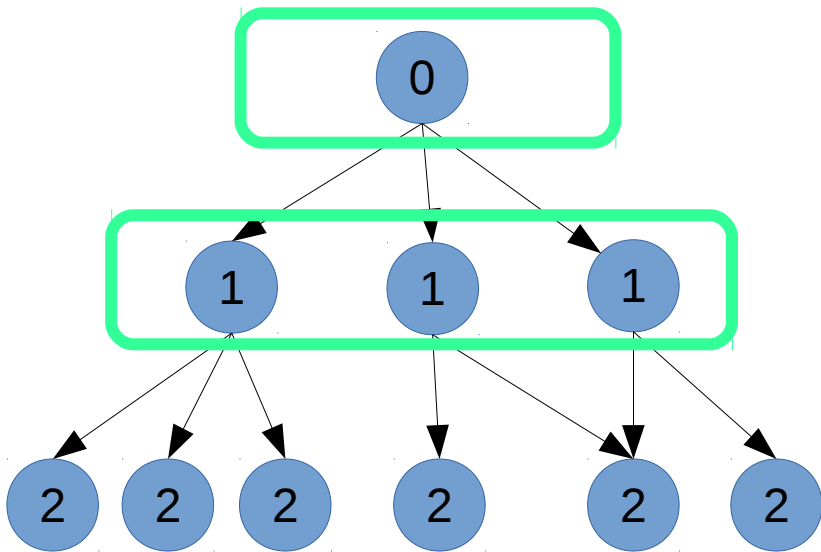
IrGL Pipe Construct

- IrGL kernels can use worklists to track work
- Pipe allows multiple kernels to communicate worklists
- All items put on a worklist by a kernel are forwarded to the next (dynamic) kernel

```
Pipe {  
  // input: bad triangles  
  // output: new triangles  
  Invoke refine_mesh(...)  
  
  // check for new bad tri.  
  Invoke chk_bad_tri(...)  
}
```



Example: Level-by-Level BFS



Kernel bfs(graph, LEVEL)

```
ForAll(node in Worklist)  
  ForAll(edge in graph.edges(node))  
    if(edge.dst.level == INF)  
      edge.dst.level = LEVEL  
      Worklist.push(edge.dst)
```

src.level = 0

```
Iterate bfs(graph, LEVEL) [src] {  
  LEVEL++  
}
```

Three Optimizations for Bottlenecks

1. Iteration Outlining

- Improve GPU utilization for short kernels

2. Nested Parallelism

- Improve load balance

3. Cooperative Conversion

- Reduce atomics

• Unoptimized BFS

- ~15 lines of CUDA
- 505ms on USA road network

• Optimized BFS

- ~200 lines of CUDA
- 120ms on the same graph

4.2x Performance Difference!



Outline

- IrGL Language
- **IrGL Optimizations**
- Results



Optimization #1: Iteration Outlining

Bottleneck #1: Launching Short Kernels

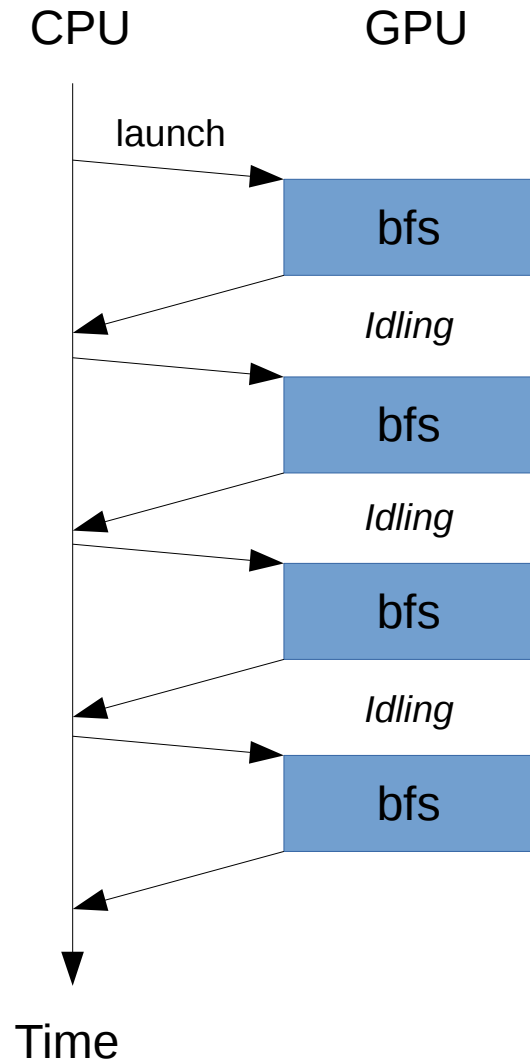
```
Kernel bfs(graph, LEVEL)
  ForAll(node in Worklist)
    ForAll(edge in graph.edges(node))
      if(edge.dst.level == INF)
        edge.dst.level = LEVEL
        Worklist.push(edge.dst)
```

```
src.level = 0
```

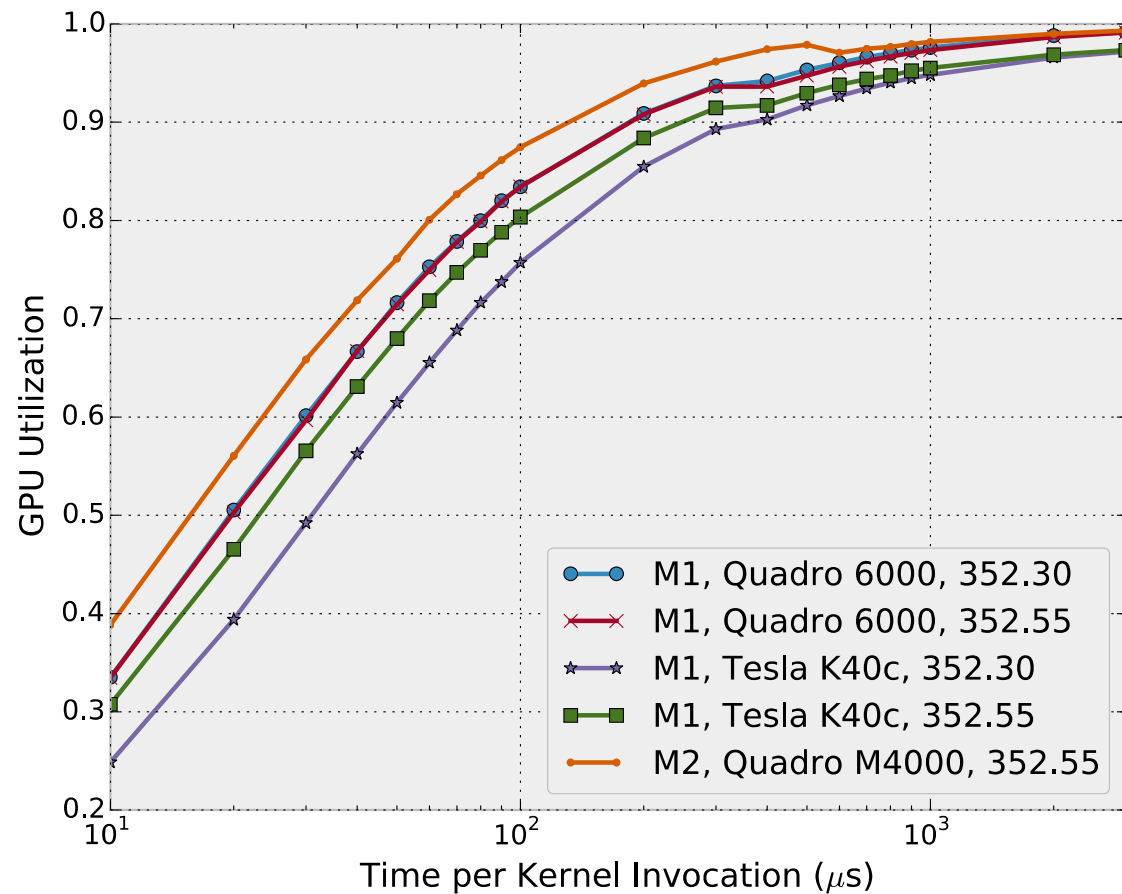
```
Iterate bfs(graph, LEVEL) [src] {
  LEVEL++
}
```

- USA road network: 6261 bfs calls
- Average bfs call duration: 16 μ s
- Total time should be $16 * 6261 = 100$ ms
- Actual time is 320 ms: 3.2x slower!

Iterative Algorithm Timeline

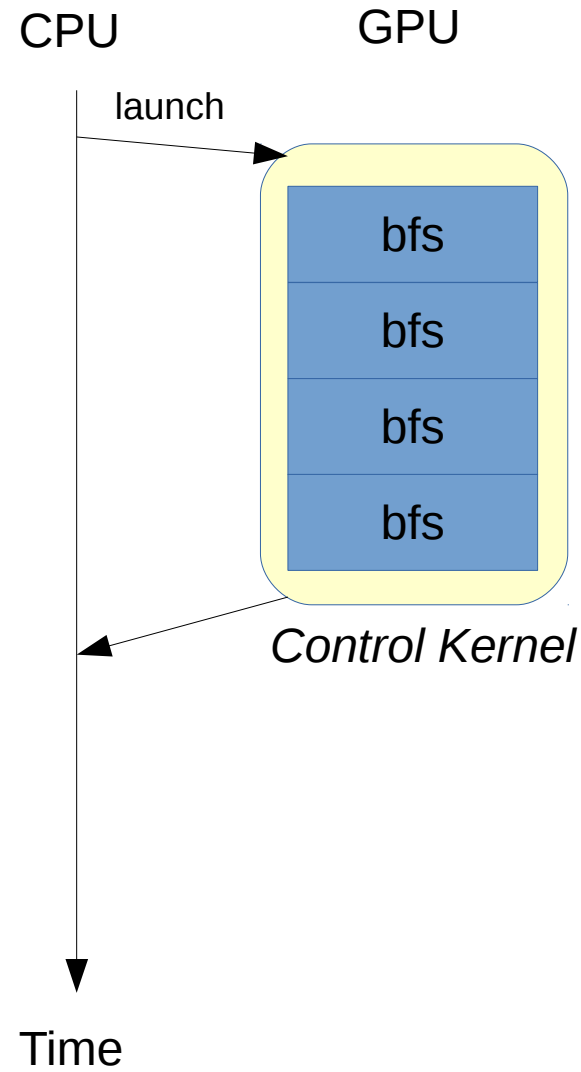


GPU Utilization for Short Kernels



Improving Utilization

- Generate Control Kernel to execute on GPU
- Control kernel uses function calls on GPU for each iteration
- Separates iterations with device-wide barriers
 - Tricky to get right!





Benefits of Iteration Outlining

- Iteration Outlining can deliver up to 4x performance improvements
- Short kernels occur primarily in high-diameter, low-degree graphs
 - e.g. road networks

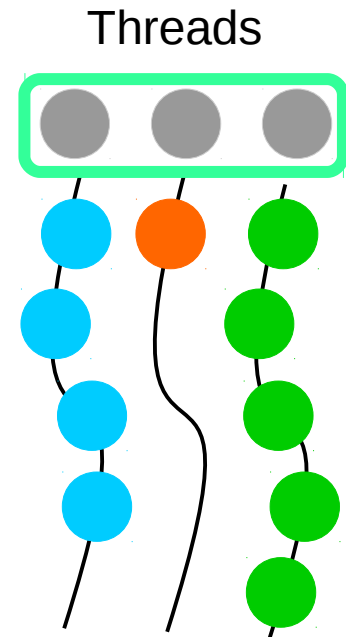
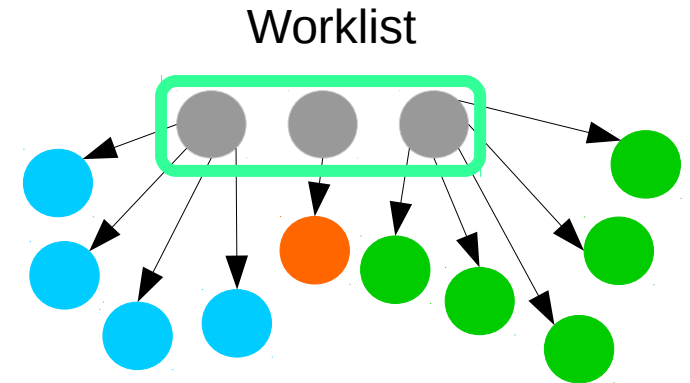


Optimization #2: Nested Parallelism

Bottleneck #2: Load Imbalance from Inner-loop Serialization

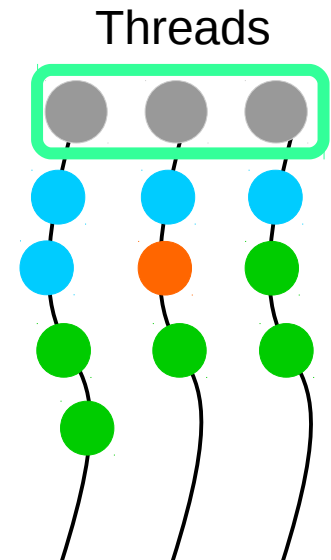
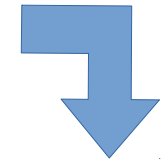
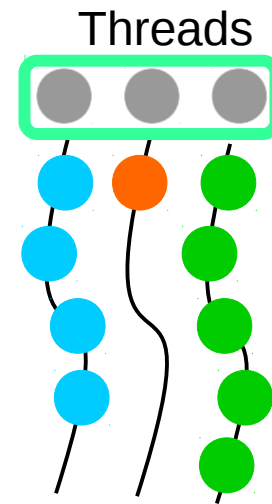
```
Kernel bfs(graph, LEVEL)
  ForAll(node in Worklist)
    ForAll(edge in graph.edges(node))
      if(edge.dst.level == INF)
        edge.dst.level = LEVEL
        Worklist.push(edge.dst)
```

```
src.level = 0
Iterate bfs(graph, LEVEL) [src] {
  LEVEL++
}
```

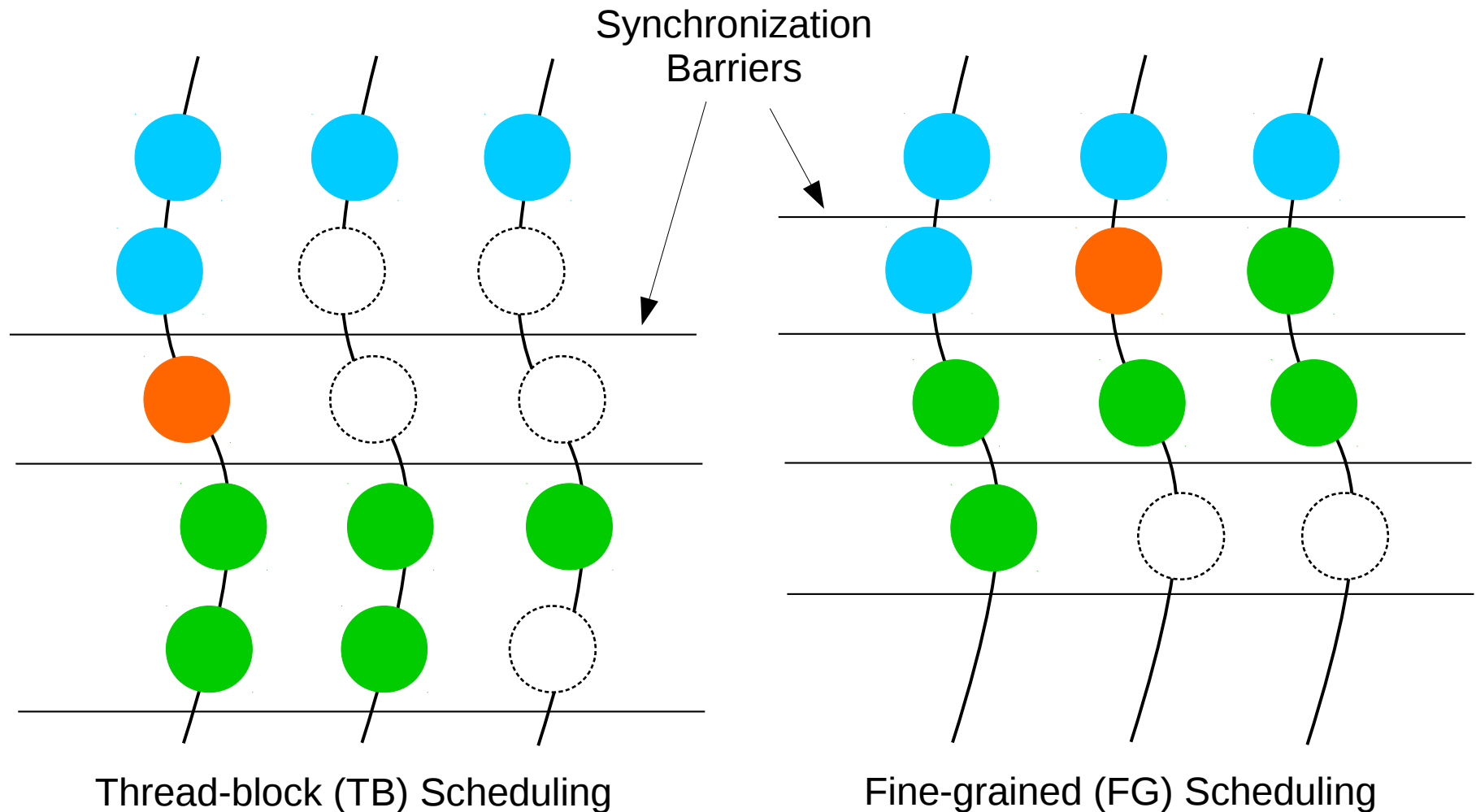


Exploiting Nested Parallelism

- Generate code to execute inner loop in parallel
 - Inner loop trip counts not known until runtime
- Use Inspector/Executor approach at runtime
- Primary challenges:
 - Minimize Executor overhead
 - Best-performing Executor varies by algorithm and input



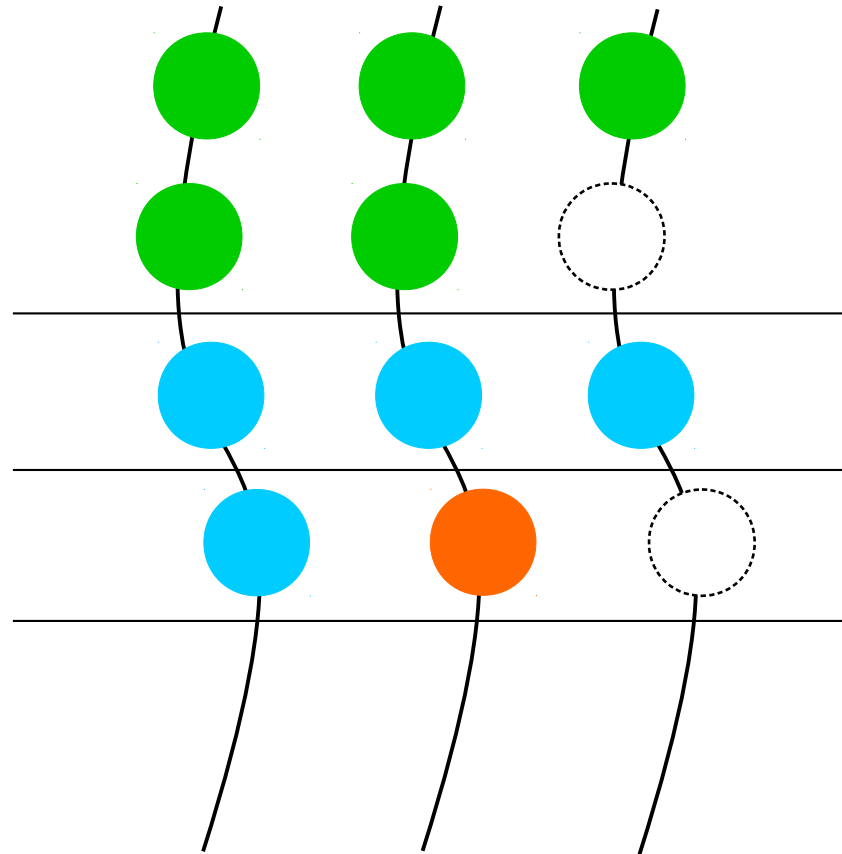
Scheduling Inner Loop Iterations



Multi-Scheduler Execution

Use thread-block (TB)
for high-degree nodes

Use fine-grained (FG)
for low-degree nodes



Thread-block (TB) + Finegrained (FG) Scheduling

Which Schedulers?

Policy	BFS	SSSP-NF	Triangle
Serial Inner Loop	1.00	1.00	1.00
TB	0.25	0.33	0.46
Warp	0.86	1.42	1.52
Finegrained (FG)	0.64	0.72	0.87
TB+Warp	1.05	1.40	1.51
TB+FG	1.10	1.46	1.55
Warp+FG	1.14	1.56	1.23
TB+Warp+FG	1.15	1.60	1.24

Speedup relative to Serial execution of inner-loop iterations on a synthetic scale-free RMAT22 graph. Higher is faster. Legend: SSSP NF -- SSSP NearFar



Benefits of Nested Parallelization

- Speedups depend on graph, but seen up to 1.9x
- Benefits graphs containing nodes with high degree
 - e.g. social networks
- *Negatively* affects graphs with low, uniform degrees
 - e.g. road networks
 - Future work: low-overhead schedulers



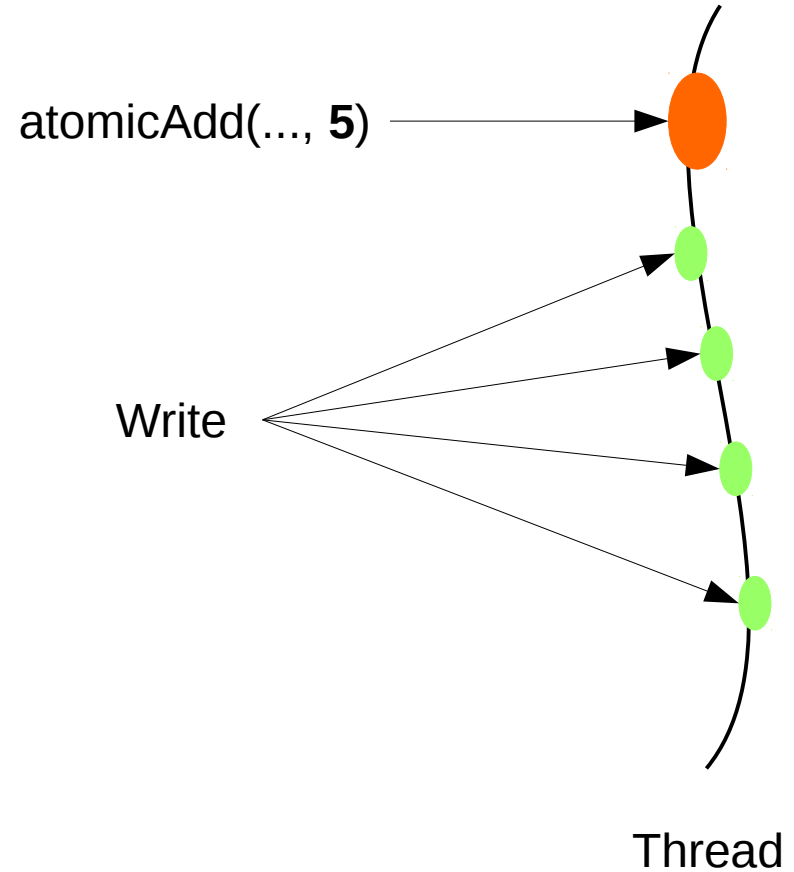
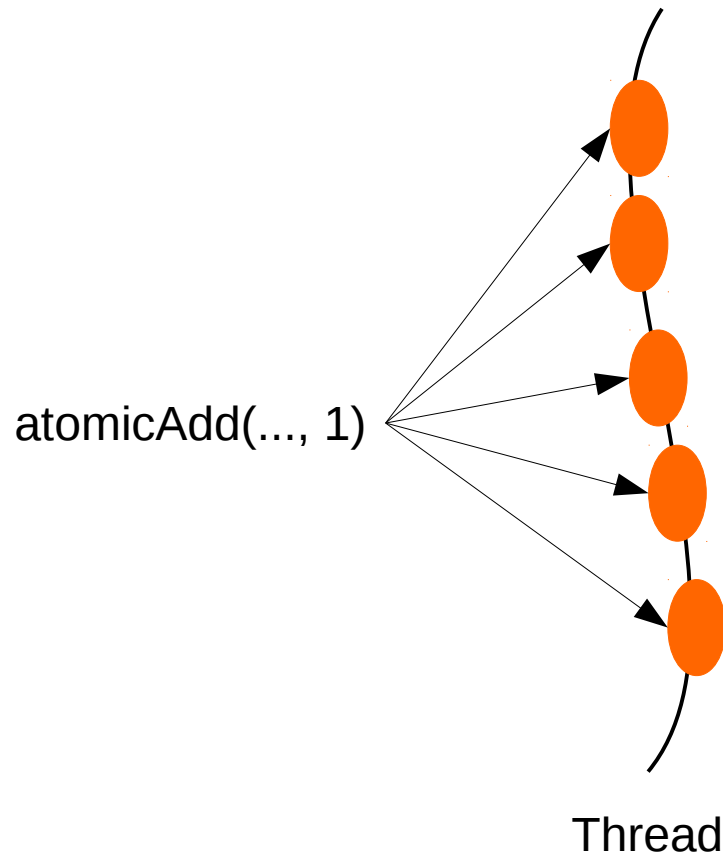
Optimization #3: Cooperative Conversion

Bottleneck #3: Atomics

```
Kernel bfs(graph, LEVEL)
  ForAll(node in Worklist)
    ForAll(edge in graph.edges(node))
      if(edge.dst.level == INF)
        edge.dst.level = LEVEL
        Worklist.push(edge.dst)
src.level
It
  pos = atomicAdd(Worklist.length, 1)
  Worklist.items[pos] = edge.dst
}
```

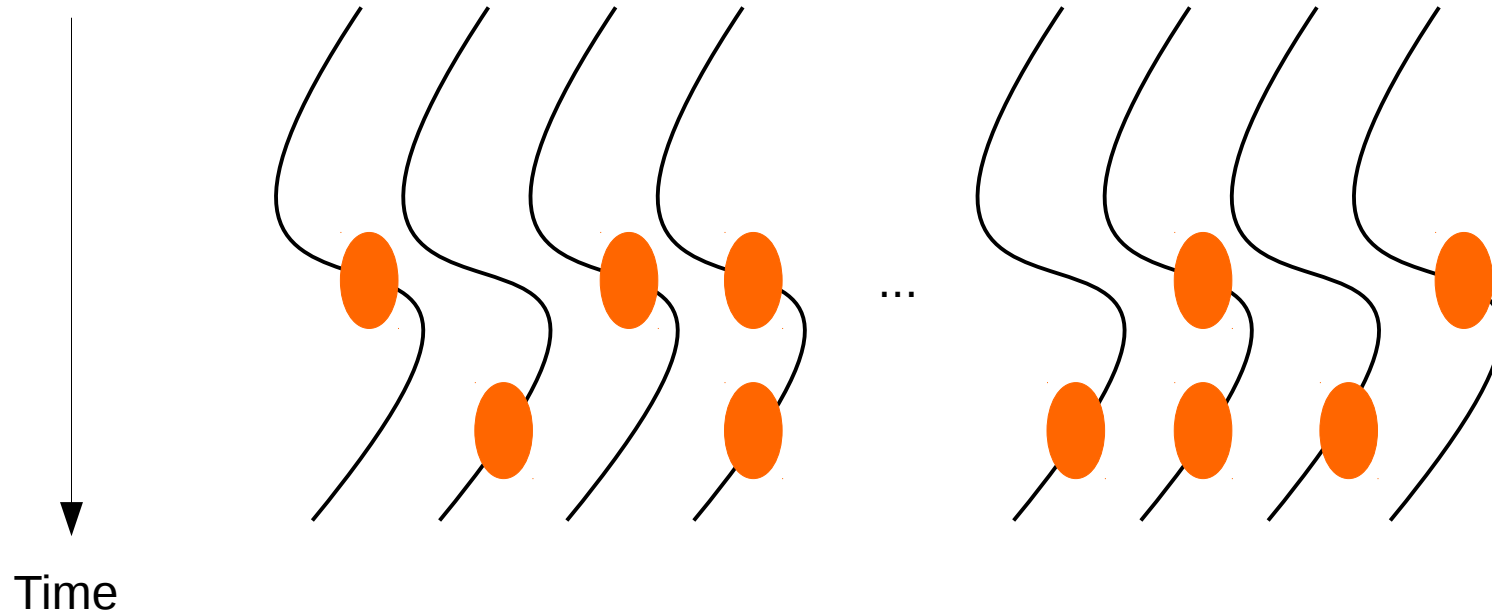
- Atomic Throughput on GPU: 1 per clock cycle
 - Roughly translated: 2.4 GB/s
 - Memory bandwidth: 288GB/s

Aggregating Atomics: Basic Idea



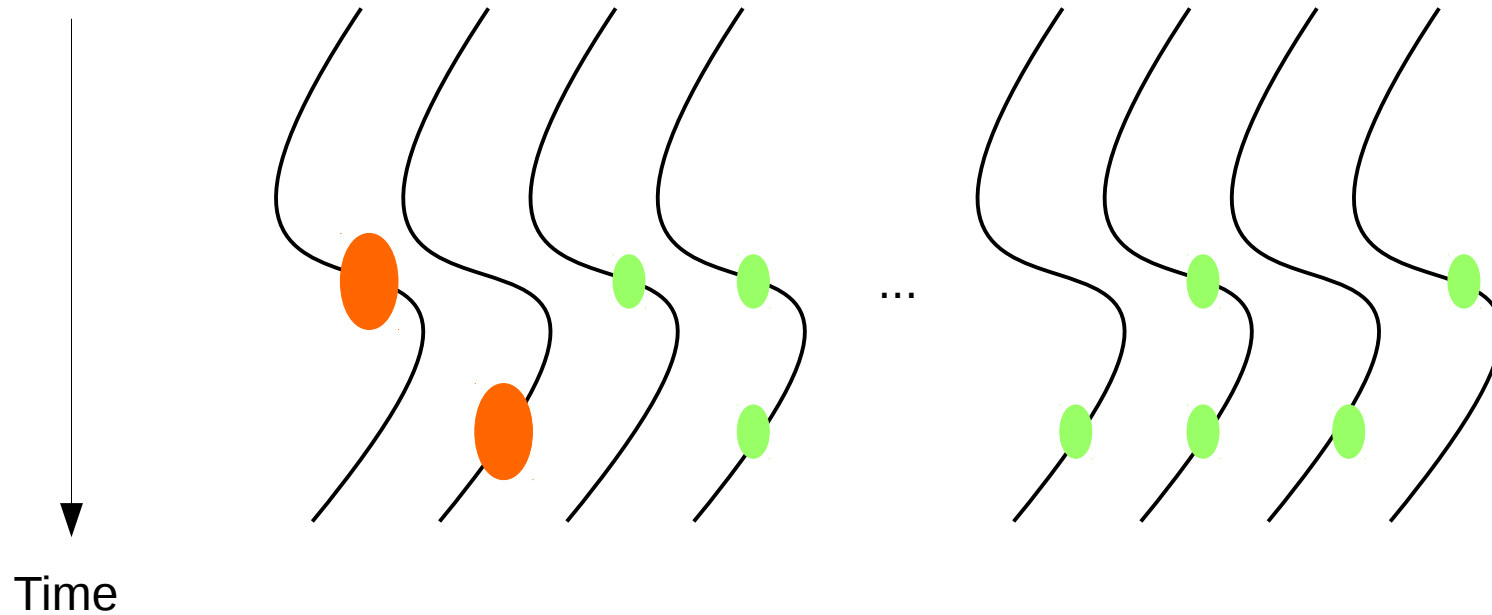
Challenge: Conditional Pushes

```
if(edge.dst.level == INF)  
  Worklist.push(edge.dst)
```



Challenge: Conditional Pushes

```
if(edge.dst.level == INF)
  Worklist.push(edge.dst)
```



Must aggregate atomics *across* threads



Cooperative Conversion

- Optimization to reduce atomics by cooperating across threads
- IrGL compiler supports all 3 possible GPU levels:
 - Thread
 - Warp (32 contiguous threads)
 - Thread Block (up to 32 warps)
- Primary challenge:
 - Safe placement of barriers for synchronization
 - Solved through novel Focal Point Analysis

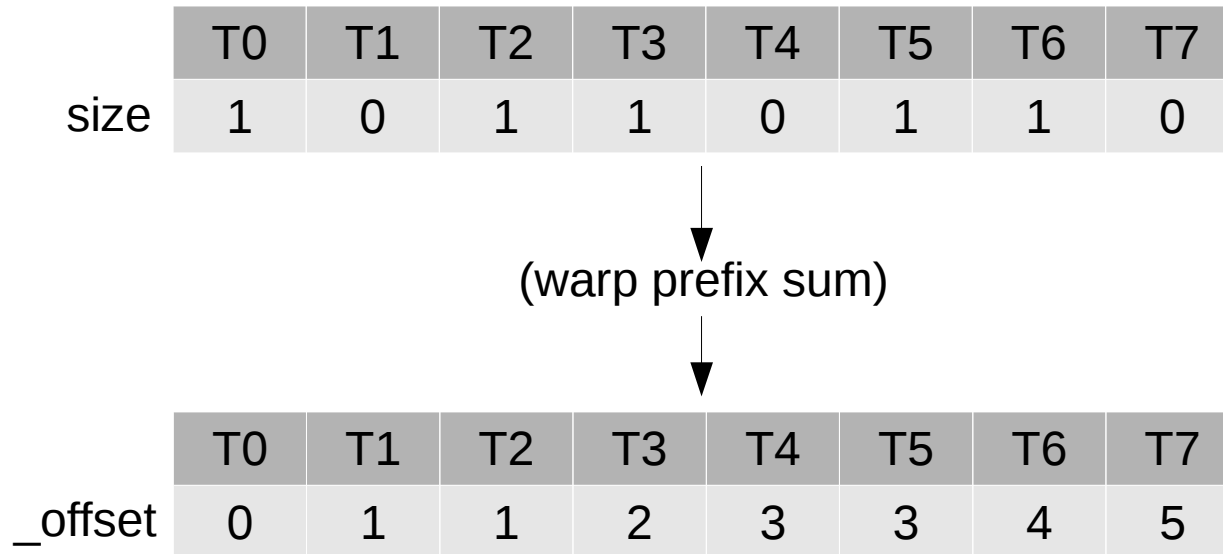
Warp-level Aggregation

```
Kernel bfs_kernel(graph, ...)
  ForAll(node in Worklist)
    ForAll(edge in graph.edges(node))
      if(edge.dst.level == INF)
        ...
        start = Worklist.reserve_warp(1)
        Worklist.write(start, edge.dst)
```

Inside reserve_warp

reserve_warp

(assume a warp has 8 threads)



```
T0: pos = atomicAdd(Worklist.length, 5)  
    broadcast pos to other threads in warp
```

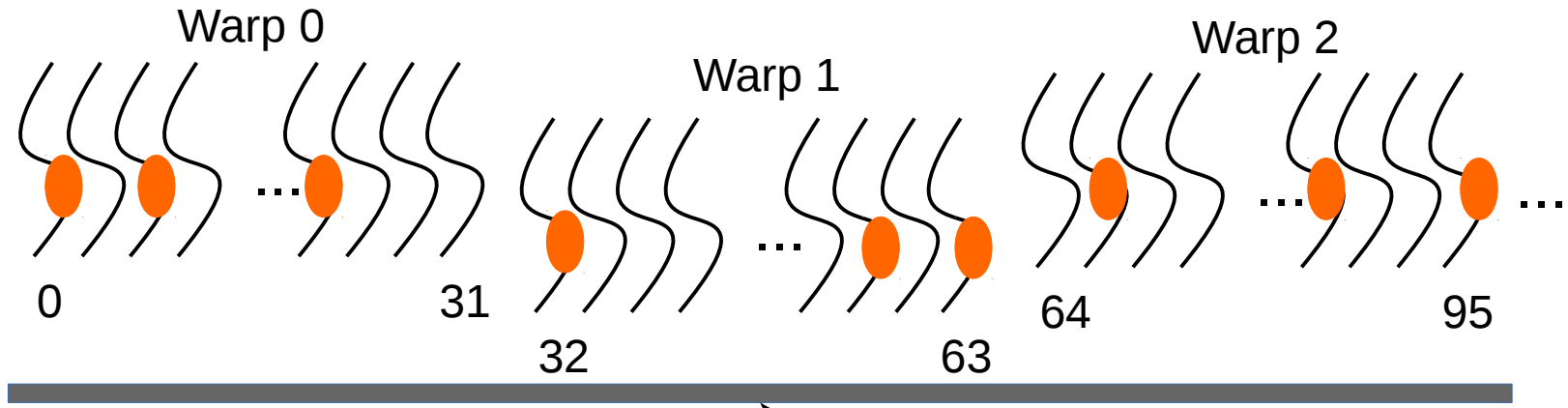
```
return pos + _offset
```


Thread-block aggregation?

```
Kernel bfs(graph, ...)
  ForAll(node in Worklist)
    ForAll(edge in graph.edges(node))
      if(edge.dst.level == INF)
        start = Worklist.reserve_tb(1)
        Worklist.write(start, edge.dst)
```

Inside reserve_tb

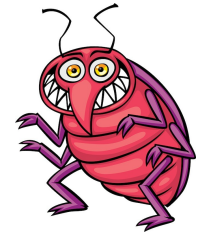
`reserve_tb`



Barrier required to synchronize warps, so can't be placed in conditionals

reserve_tb is incorrectly placed!

```
Kernel bfs(graph, ...)
  ForAll(node in Worklist)
    ForAll(edge in graph.edges(node))
      if(edge.dst.level == INF)
        start = Worklist.reserve_tb(1)
        Worklist.write(start, edge.dst)
```



Solution: Place `reserve_tb` at a Focal Point

- Focal Points [Pai and Pingali, OOPSLA 2016]
 - All threads pass through a focal point all the time
 - Can be computed from control dependences
 - Informally, if the execution of some code depends only on uniform branches, it is a focal point
- Uniform Branches
 - branch decided the same way by all threads [in scope of a barrier]
 - Extends to loops: *Uniform loops*

reserve_tb placed

Made uniform
by nested parallelism

```
Kernel bfs(graph, ...)  
  ForAll(node in Worklist)  
    UniformForAll(edge in graph.edges(node))  
      will_push = 0  
      if(edge.dst.level == INF)  
        will_push = 1  
        to_push = edge  
  
      start = Worklist.reserve_tb(will_push)  
      Worklist.write_cond(willpush, start, to_push)
```



Benefits of Cooperative Conversion

- Decreases number of worklist atomics by 2x to 25x
 - Varies by application
 - Varies by graph
- Benefits all graphs and all applications that use a worklist
 - Makes concurrent worklist viable
 - Leads to work-efficient implementations



Summary

- IrGL compiler performs 3 key optimizations
- Iteration Outlining
 - eliminates kernel launch bottlenecks
- Nested Data Parallelism
 - reduces inner-loop serialization
- Cooperative Conversion
 - reduces atomics in lock-free data-structures
- Allows auto-tuning for optimizations



Outline

- IrGL Language
- IrGL Optimizations
- **Results**

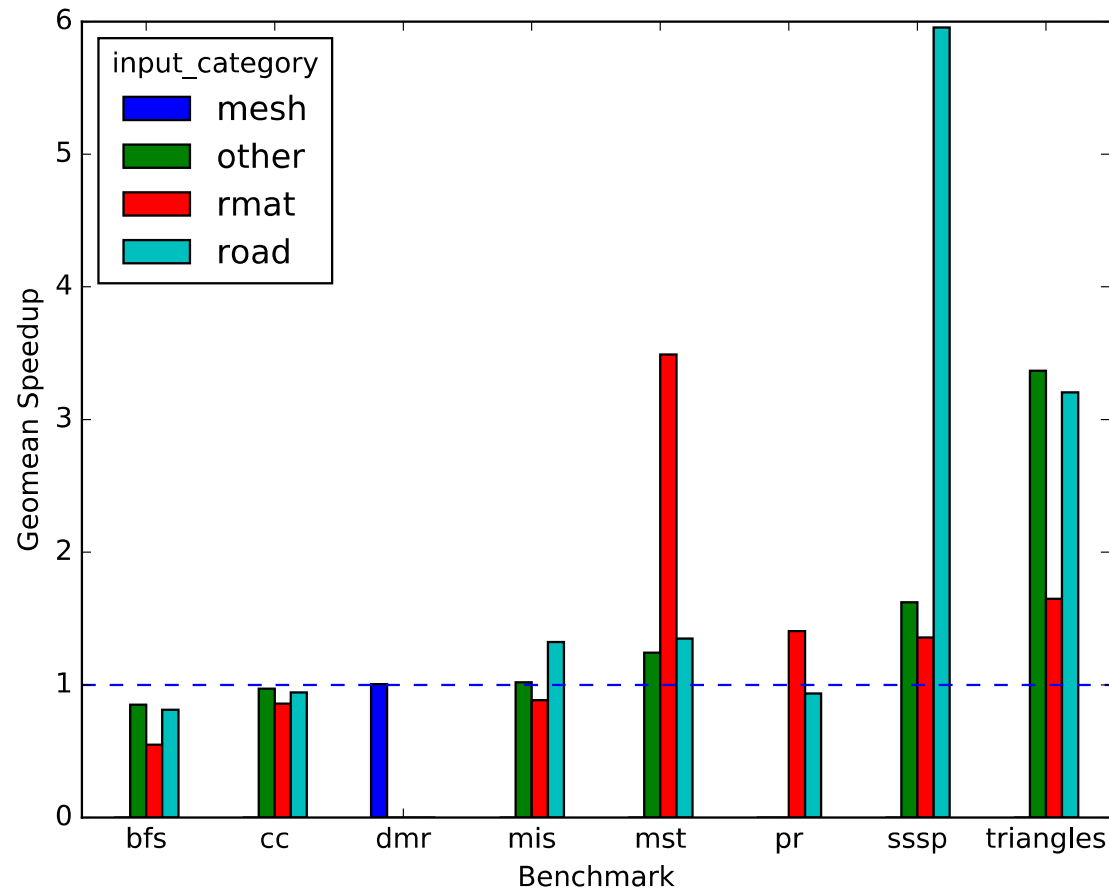
Evaluation

- Eight irregular algorithms
 - Breadth-First Search (BFS) [Merrill et al., 2012]
 - Connected Components (CC) [Soman et al., 2010]
 - Maximal Independent Set (MIS) [Che et al., 2013]
 - Minimum Spanning Tree (MST) [da Silva Sousa et al. 2015]
 - PageRank (PR) [Elsen and Vaidyanathan, 2014]
 - Single-Source Shortest Path (SSSP) [Davidson et al. 2014]
 - Triangle Counting (TRI) [Polak et al. 2015]
 - Delaunay Mesh Refinement (DMR) [Nasre et al., 2013]

System and Inputs

- Tesla K40 GPU
- Graphs
 - Road Networks
 - USA: 24M vertices, 58M edges
 - CAL: 1.9M vertices, 4.7M edge
 - NY: 262K vertices, 600K edges
 - RMAT (synthetic scale-free)
 - RMAT22: 4M vertices, 16M edges
 - RMAT20: 1M vertices, 4M edges
 - RMAT16: 65K vertices, 256K edges
 - Grid (1024x1024)
 - DMR Meshes: 10M points, 5M points, 1M points

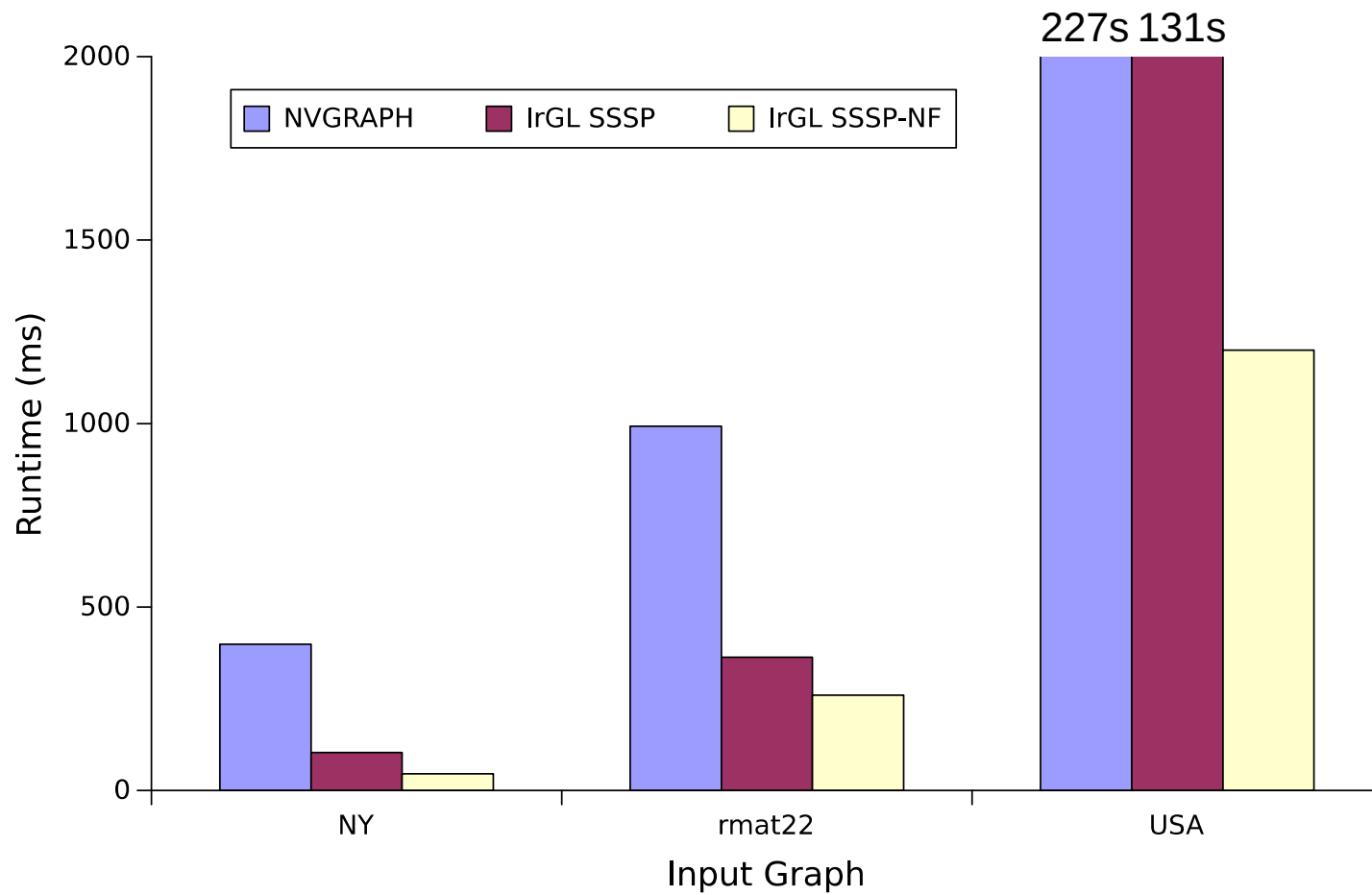
Overall Performance



Best
Handwritten
Code

Note: Each benchmark had a single set of optimizations applied to it

Comparison to NVIDIA nvgraph SSSP





Irregular Data-Parallel Algorithms

- Graph Algorithms
- Sparse Linear Algebra
- Discrete-event Simulation
- Adaptive Simulations
- Brute-force Searches
 - Constraint solvers
- Graph databases
- ...

Conclusion

- Graph analytics on GPUs requires 3 key *throughput* optimizations to obtain good performance
 - Iteration Outlining
 - Nested Parallelism
 - Cooperative Conversion
- The IrGL compiler automates these optimizations
 - Faster by up to 6x, median 1.4x
 - Faster than nvgraph



Thank you!
Questions?

sreepai@ices.utexas.edu