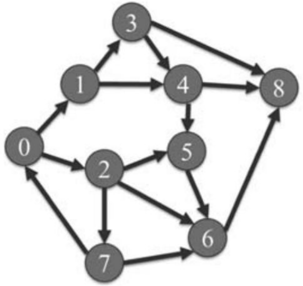
Graph Traversal

Graph Search

Graph and Matrix

- A graph is a data structure that represents the relationships between entities
  - A graph is a data structure that represents the relationships between entities
- Graphs are intrinsically related to sparse matrices
  - An intuitive representation of a graph is an adjacency matrix
  - Thus, graph computation can also be formulated in terms of sparse matrix operations
- Store them can be done using a different format
  - CSR/CSC
  - COO



destination vertex

source vertex

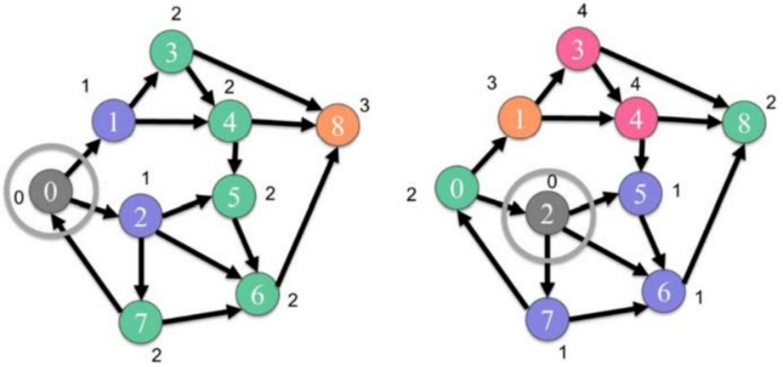
	0	1	2	3	4	5	6	7	8
0		1	1						
1				1	1				
2						1	1	1	
3					1				1
4						1			1
5							1		
6									1
7	1						1		
8									

Graph Usage Examples

- They are used to model different problems from different fields
  - Social media connection graphs
  - Driving directions
  - Telecommunication networks
  - Manufacturing process dependencies
  - Computation graph
  - 3D Meshes
  - Graphical models
- These graphs tend to be massive and sparse

Breadth-First Search (BFS)

- Given a source node S, find the number of steps required to reach each node N in the graph
  - BFS is often used to discover the shortest number of edges that one needs to traverse to go from one vertex to another vertex of the graph

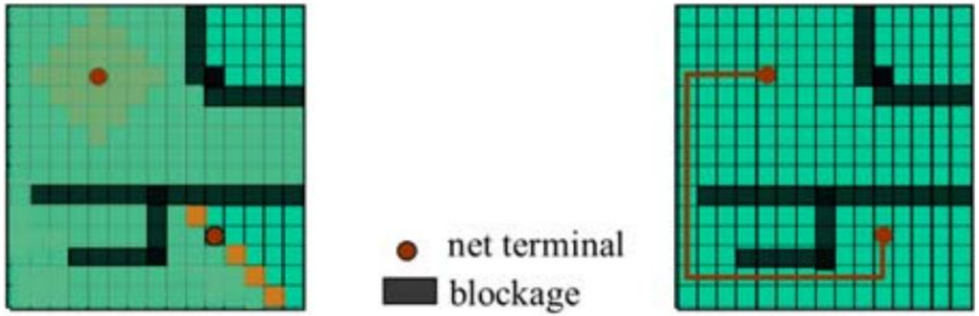


(A) Vertex 0 is root

(B) Vertex 2 is root

BSF Graph Computation Example

- While designing an integrated circuit chip, many electronic components need to be connected to complete the design
  - The routing software represents the chip as a grid of wiring blocks in which each block can potentially serve as a piece of a wire
- The maze routing application represents the chip as a graph
  - The routing blocks are vertices
  - An edge from vertex i to vertex j indicates that one can extend a wire from block i to block j



(A) Breadth-first search

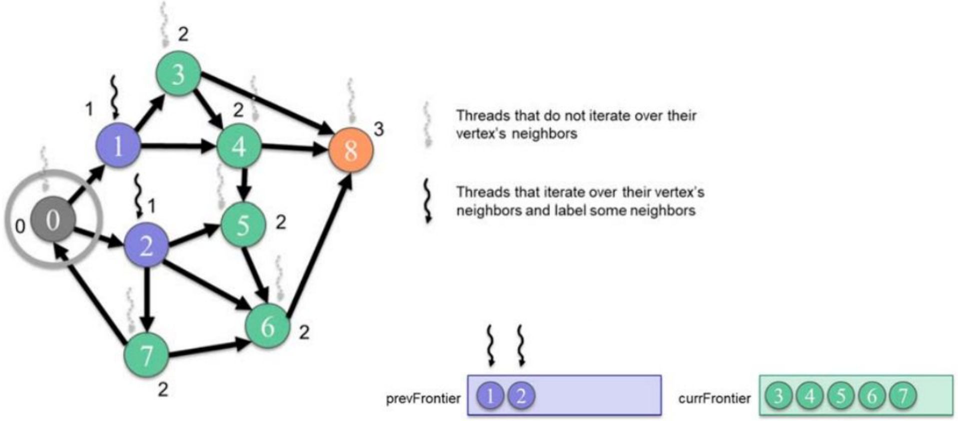
(B) Identifying a routing path

Vertix-Center BFS Computation

- Iterate or assign each thread to a vertex
  - For each iteration, check all incoming edges to see if the source vertex was just visited in the last iteration; if so, mark as visited in this iteration
  - Not very work efficient, complexity of $O(VL)$
    - V number of vertices
    - L length of the longest path
  - Difficult to detect stopping criterion

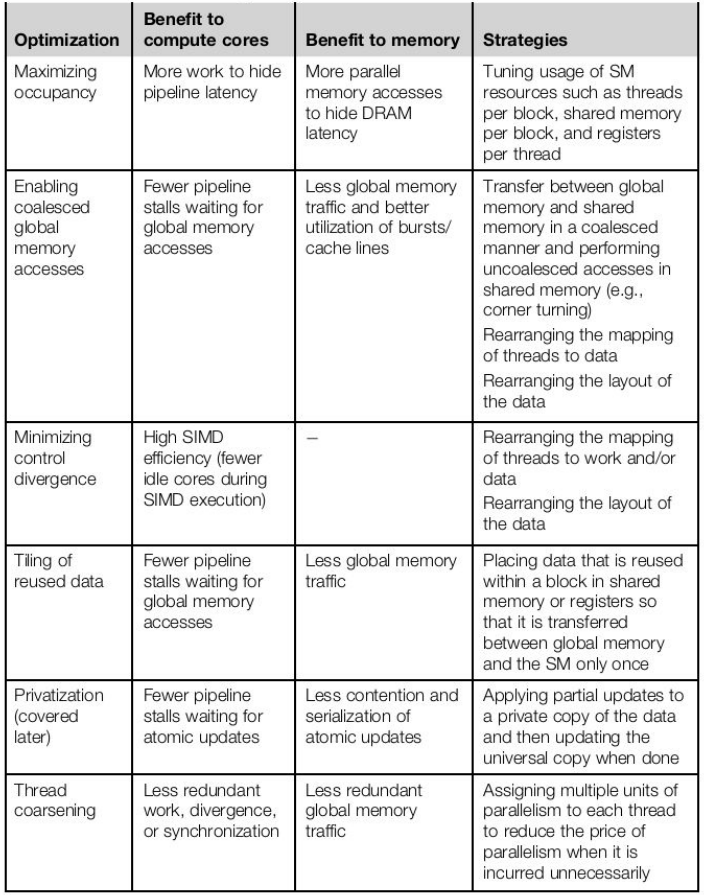
Frontier-Center Parallel BFS

- Assign threads to frontier vertices from the previous iteration
  - Add all the non-visited neighbors to the next frontier
  - The source will be the first element in the frontier
- Parallel execution between threads
  - A variable number of unnecessary threads are launched


0
0
1
1
2
1
3
2
4
2
5
2
6
2
7
2
8
3
Threads that do not iterate over their vertex's neighbors
Threads that iterate over their vertex's neighbors and label some neighbors
prevFrontier
1 2
currFrontier
3 4 5 6 7

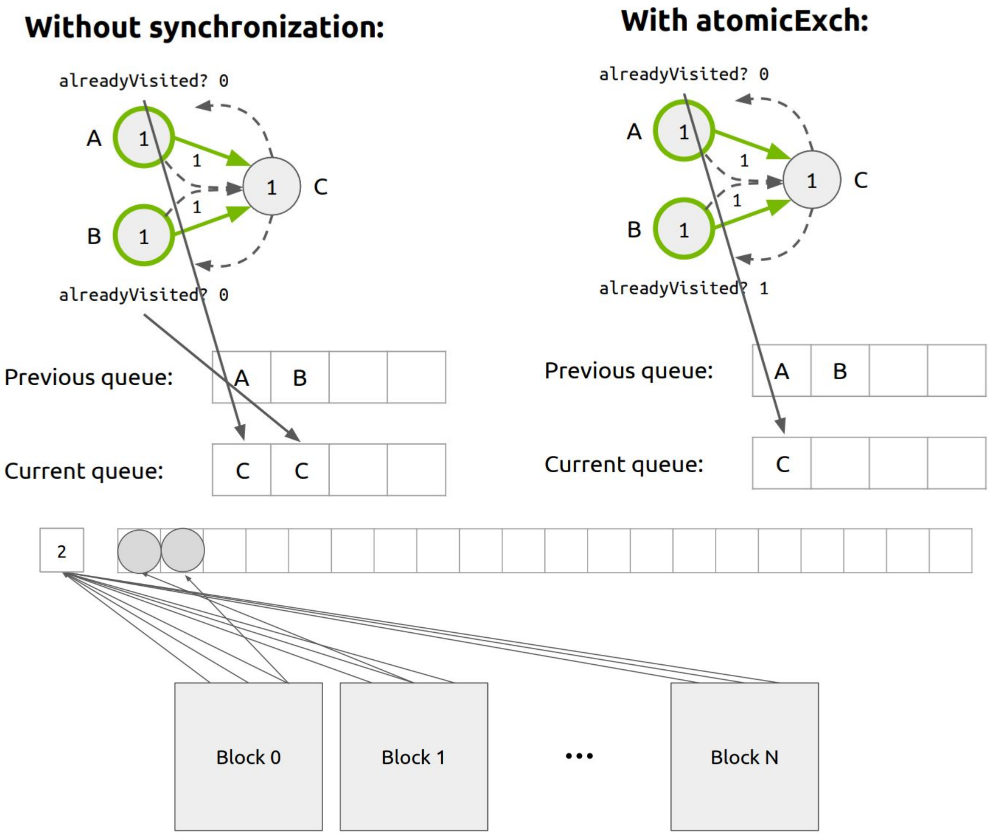
Problems and Optimizations

- Privatization
- Texture Memory
- Kernel launch overhead
- Sub Block-Level Queue
- Load Imbalance



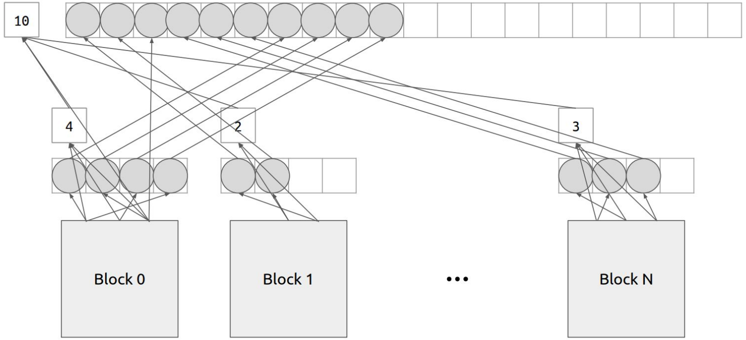
Optimization	Benefit to compute cores	Benefit to memory	Strategies
Maximizing occupancy	More work to hide pipeline latency	More parallel memory accesses to hide DRAM latency	Tuning usage of SM resources such as threads per block, shared memory per block, and registers per thread
Enabling coalesced global memory accesses	Fewer pipeline stalls waiting for global memory accesses	Less global memory traffic and better utilization of bursts/ cache lines	Transfer between global memory and shared memory in a coalesced manner and performing uncoalesced accesses in shared memory (e.g., corner turning) Rearranging the mapping of threads to data Rearranging the layout of the data
Minimizing control divergence	High SIMD efficiency (fewer idle cores during SIMD execution)	–	Rearranging the mapping of threads to work and/or data Rearranging the layout of the data
Tiling of reused data	Fewer pipeline stalls waiting for global memory accesses	Less global memory traffic	Placing data that is reused within a block in shared memory or registers so that it is transferred between global memory and the SM only once
Privatization (covered later)	Fewer pipeline stalls waiting for atomic updates	Less contention and serialization of atomic updates	Applying partial updates to a private copy of the data and then updating the universal copy when done
Thread coarsening	Less redundant work, divergence, or synchronization	Less redundant global memory traffic	Assigning multiple units of parallelism to each thread to reduce the price of parallelism when it is incurred unnecessarily

Frontier Output Interference



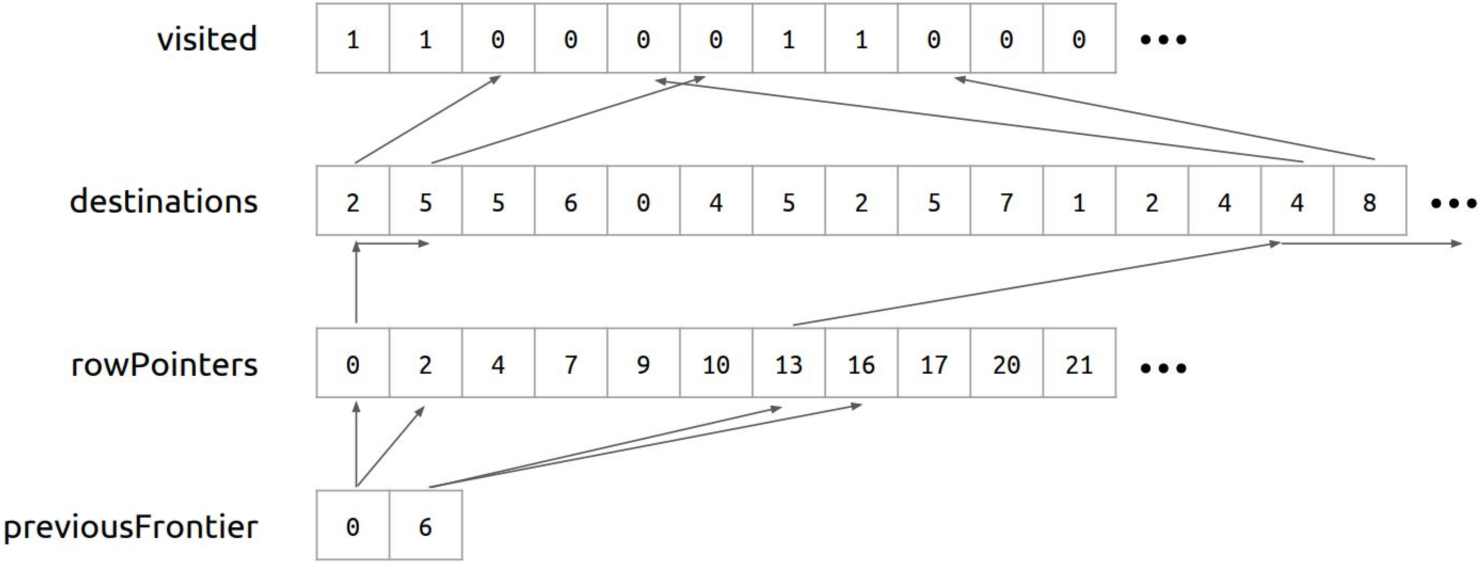
Privatization

- Privatization reduces contention of atomics by applying partial updates to a private copy of the data, then updating the public copy when done
  - Threads will contend on the same data only with other threads in the same block
  - The local frontier and its counter can be stored in shared memory
    - Which enables lower-latency atomic operations



Irregular global memory access

- Access patterns depend on graph structure and are unpredictable



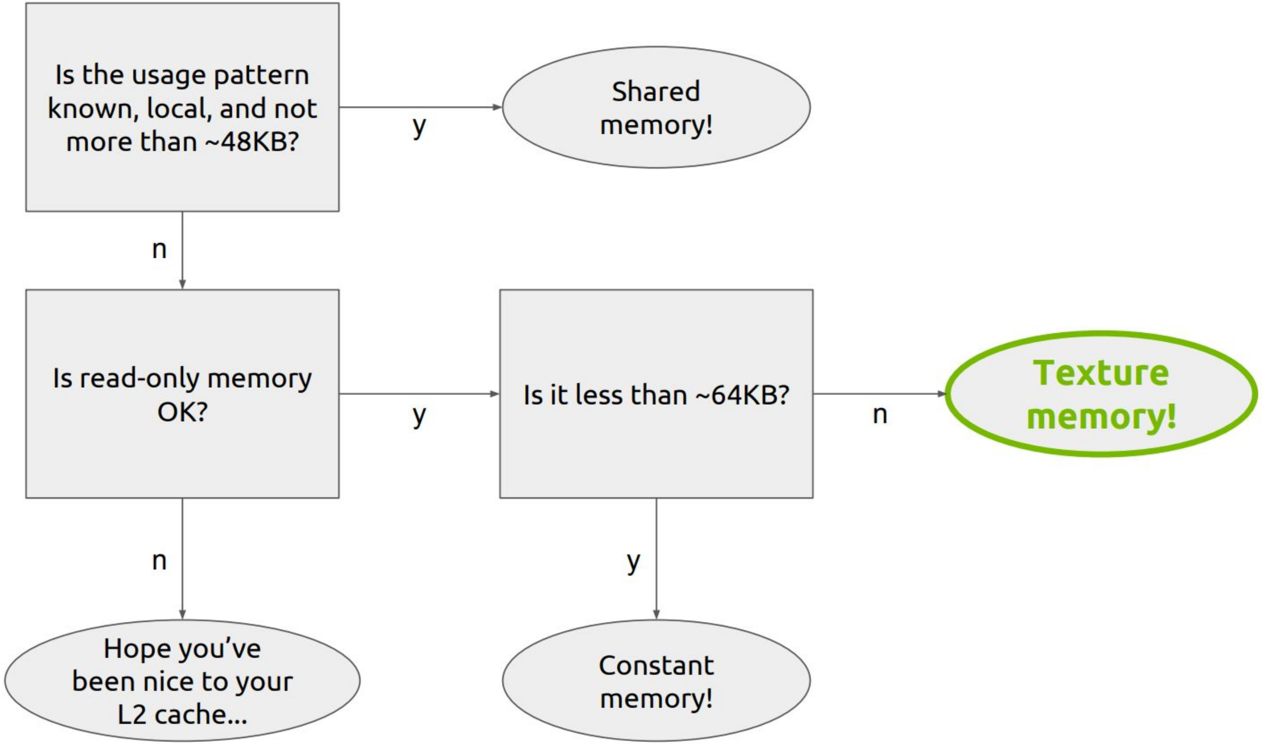
visited: 1 1 0 0 0 0 1 1 0 0 0 •••

destinations: 2 5 5 6 0 4 5 2 5 7 1 2 4 4 8 •••

rowPointers: 0 2 4 7 9 10 13 16 17 20 21 •••

previousFrontier: 0 6

Memory Optimization Flowchart


Is the usage pattern known, local, and not more than ~48KB?
y
Shared memory!
n
Is read-only memory OK?
y
Is it less than ~64KB?
n
Texture memory!
n
y
Hope you've been nice to your L2 cache...
Constant memory!

Texture Memory

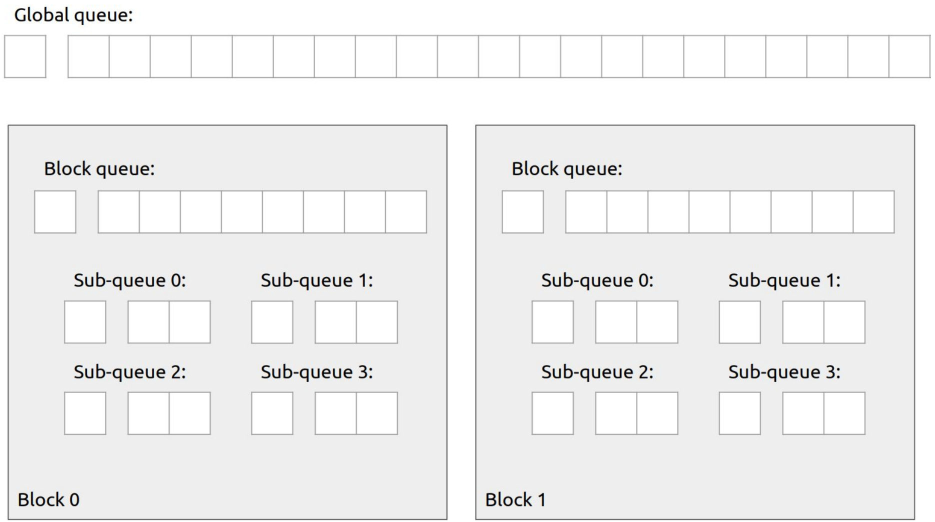
- Texture memory is another form of global memory
  - Like constant memory, it is aggressively cached for read-only access
- Originally developed and optimized for storing and reading textures for graphics applications
  - Has hardware-level support for 1-,2-, or3-D layouts and interpolated reads
  - The texture cache is also spatial layout-aware
- It can be useful for irregular access patterns with un-coalesced reads

Kernel Launch Overhead

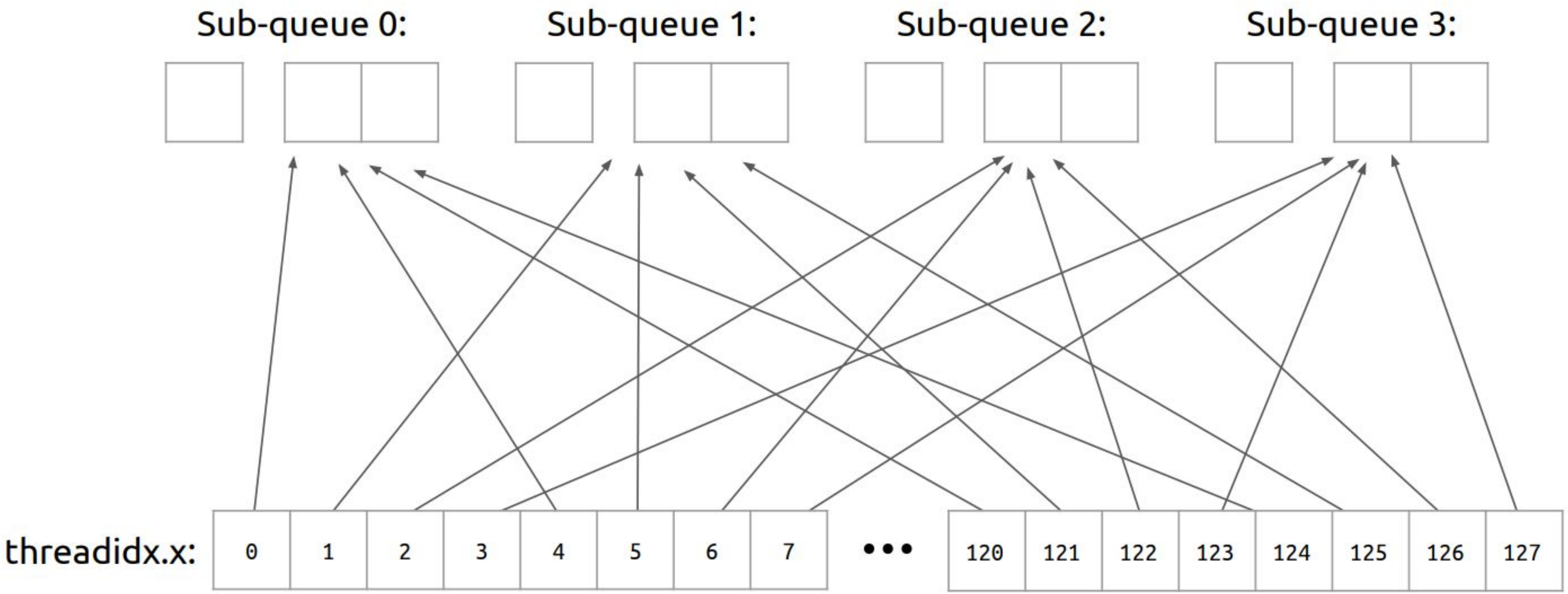
- For some iterations of BFS (especially near the beginning), the frontier can be quite small
  - The benefits of parallelism only outweigh the kernel launch overhead when the frontier becomes large enough
- Use the CPU if the frontier size dips below some threshold

Block-level queue length counter contention

- While the block-level queues reduced contention for global memory, the block-level counter is now the bottleneck
- We can extend the hierarchy by further splitting the block-level queue



Sub-Queue


Sub-queue 0:
Sub-queue 1:
Sub-queue 2:
Sub-queue 3:
threadidx.x:
0
1
2
3
4
5
6
7
•••
120
121
122
123
124
125
126
127

Credits

- CSE 599 I Accelerated Computing - Programming GPUS - Parallel Pattern: Graph Traversal, Author *Tanner Schmidt*

Thanks for your attention!

Gianmarco Accordi
gianmarco.accordi@polimi.it