Objective

- to learn the role of privatization in algorithm scalability
- to learn how to implement queue structures that support efficient dynamic data extraction
- to learn efficient kernel structures to accommodate dynamic variation of working set size

Started with Data-Independent Problems

We started with applications
  - with data-independent access patterns.
  - In DCS and GEMM,
    - all accesses are known
    - once you know the problem parameters.

So we focused on optimizing the access patterns.

Scatter to Gather Transformation

In many applications, inputs not easily identified.
Then Examined Data Reorganization

Using a cutoff algorithm was more challenging.

But binning sufficed to reorganize the data
– so that accesses are again defined
– before we start computing grid potentials.

There’s still some variation with variable-sized bins,
but the access patterns are mostly well-defined.

Today, Look at Highly Data-Dependent Accesses

Now we look at
• highly data-dependent access patterns
• that cannot be known in advance.
In other words,
• the only way to know the access pattern
• is to perform the computation.
• No competitive algorithm is known to find the pattern in a different way (to enable, for example, data reorganization).

Example: Finding a Shortest Path in a Graph

Consider finding a shortest path in a graph.

• Finding a path depends on the graph itself.

• Which nodes to examine depends on
  – the edges from the starting point, and
  – the edges from the neighbors, and
  – so on.

Alternatives are Not Competitive and Not Scalable

Can we precompute?

Yes! Can precompute shortest paths for all pairs, then optimize accesses for a given graph.
• That’s not a competitive algorithm if all we want is one shortest path.
• And it’s not scalable regardless.
  – Google maps does not store shortest paths for all pairs of points in the world,
  – Although they probably do maintain derived hierarchical representations of the world map.
Dynamic Extraction: Step Identifies Accesses for Next Step

- We need to solve
  - a problem of **dynamic extraction**:
  - **at every step**, the algorithm **identifies which part** of the data must be **considered in the next** step.
- This kind of problem
  - is **extremely difficult for practical architectures**, as it
  - **stresses the memory system and**
  - **offers widely varying** levels of **parallelism** in each step.
  (DARPA still has big graph challenges to address this mismatch between algorithms and architectures.)

Start with Breadth-First Search

- We **start simple**, with **BFS**.
  - BFS is the **entry-level computation** for graphs:
    - one of the **simplest** problems,
    - **but gives good insights**
    - on how to **address irregular, dynamic behavior**.

Queues and Privatization will Improve Performance

- As part of our discussion,
  - we **will introduce queues**,
  - not so different than queues you’ve seen,
  - but **with extremely high throughput**.
- The only way to achieve the necessary throughput
  - is **through privatization**:
    - replicating data to expose parallelism.

Privatization is Not a Panacea for Performance

- **Privatization** is probably not new to you:
  - it’s **used with histograms** (in 408)
  - **to transform global atomic** operations
  - **into atomic operations on shared memory**.
- In histogram,
  - privatization is relatively easy
    - as the operation is associative and commutative.
- **Privatization doesn’t solve all such problems**.
Dynamic Data Extraction Necessary

- For good GPU performance, we need massive parallelism.
- With multidimensional data, array indices suffice to enable each thread to find values in parallel.
- With binning, we reorganized data to enable parallel access.
- Now that's not possible: data must be extracted dynamically.

Graph Algorithms Often Require Dynamic Extraction

- Data for each phase are dynamically determined and extracted from a bulk data structure.
- May also need to reorganize the data for access!
- Graph algorithms are popular examples, and are widely used in EDA (electronic design automation / computer-aided design) and large-scale optimization.
- We use Breadth-First Search (BFS) as an example.

Graphs Represented as Adjacency Matrices

- easier to modify, but slower to access
- faster to access, but harder to modify

Main Challenges of Dynamic Data

1. organize data for locality, coalescing, and avoiding contention during execution

2. available parallelism may grow and shrink dynamically during execution
   - different kernel strategies fit different data sizes
   - difficult to encode into a single CUDA kernel launch, in which neither code nor parallelism can change*

*Unless using CUDA’s dynamic parallelism, which we discuss later.
Graphs Represented as Adjacency Matrices

Graphs typically sparse.
Adjacency matrix can be compacted into CSR format—we know how to use that!

CSR is the Basic Form for Graphs

- CSR format (as shown) is convenient if you only need to follow the edge directions.
- Can just as easily build CSR for reversed edges (sometimes useful, as we’ll see).
- Not trivial to transpose / use in reverse.
- More complex representations are possible, and trade additional data for access efficiency.

Sparse Matrix Format and Use is Important

- That’s why we teach sparse matrix format and use in 408: they are important for all types of graph algorithms (and many other applications).
- Techniques usually transfer between sparse matrix uses for algebra and for graphs.
- We’ll discuss some matrix-based approaches to BFS later.

Definition of Breadth-First Search

BFS:
- Given a source node,
- calculate the minimum number of edges that must be traversed to reach a destination node (or to any node).
BFS Used Heavily in VLSI Routing

(Maze Routing example)

How does one find the path?
Backtrack through distance, or use a heuristic to pick.
Example: Prefer straight lines.

More BFS Applications in VLSI CAD

- reachability analysis
- finding connected components
- logic simulation/timing analysis
- logic synthesis
- and so forth

Sequential BFS is Efficient!

ECE students see this algorithm in their first class!

- **Use a queue** to explore the graph
  (building a logical tree).
  1. Add source node to queue.
  2. Explore queue nodes in order.
  3. For each queue node, add all unvisited neighbors
e     to queue (recording distance and/or predecessor).

Initialize BFS with the Source

distance 0

queue 0
Exploring the Node 0 to Find 1 and 2

Explore Node 2 to Find 5, 6, and 7

Explore Node 5 to Find 8

Exploring the Node 1 to Find 3 and 4
BFS Outcome for Source Node 0

Complexity $O(V+E)$—linear in size of graph!
We also need per-node distance labels (initially -1).

And Now for a Volunteer from Our Audience…
So …

Where is the parallelism?

Parallelism Varies Over Algorithm Execution
Even in a tiny graph, parallelism
• starts out small,
• grows, then
• shrinks again
• (highly variable).
One Kernel per Frontier

What happens between frontiers?

Simplest answer:
• Wait until frontier complete (synchronize),
• but >1 thread block, so
• launch one kernel per frontier.

Avoid Backtracking with Distance Values

Is there a queue?
• No, need to look at edge destination distance.
• Examples:
  – 7 reaches 0, but node 0 has distance below 3
  – 3, 4, and 6 reach 8 … better add to next frontier atomically!

Example: Source Node 2, 0-Hop Frontier

Example: Source Node 2, 1-Hop Frontier

Frontier processed in parallel!
Hard to Map BFS to Our Usual Strategies

The implications of dynamic extraction and variable parallelism?

- Must **find enough parallelism** to keep GPU busy.
- **Not easy to coalesce** accesses to graph data.
- Threads in charge of different nodes from frontier to frontier.
- **Not easy to amortize access cost** through reuse.

Potential Pitfall of Parallel Algorithms

Remember: best sequential code is the baseline!

Massively parallel $O(N \log N)$ algorithm still **slower than $O(N)$ sequential approach on large data sets.**

Narrow Parallelism Fits within a Thread Block

Do we really need one kernel per frontier?  
**Not quite.**

When little parallelism available, **can work within one thread block** and use barriers.

Use CPU to Expand First Few Levels

One thread block makes sense at the start, 
- but the **CPU is probably faster, and** 
- is typically **used in practice** to start.

The **end is trickier:**  
- narrow parallelism doesn’t imply completion, so 
- may **not be easy to shift models.**
Correctness Does Not Require Synchronization

But … do we really even need to synchronize?
Not exactly.

Updating a node with an overly long distance creates extra work, but we will eventually fix it.

Many Challenges if Synchronization Dropped

The challenges are…
• to avoid creating too much useless work,
• to prioritize important work (even if we can prioritize, how can we express priority in CUDA?),
• to squash useless work as quickly as possible, and
• to obey CUDA’s memory consistency rules.

We won’t try it in lecture.

Always Consider the Big Picture

Always important to see the big picture.

• Need to solve many graphs?
  Let parallelism interleave naturally to occupy GPU.
• Many sources on the same graph?
  Faster to solve all pairs shortest-paths?
• One source on one graph ASAP?
  Isn’t possibly useless work better than idling?

Early Attempts to Parallelize BFS

• Node-Oriented Parallelization
  threads assigned statically to nodes

• Matrix approaches
  – Edge-Oriented, Adjacency-Matrix-Based Parallelization
  – matrix multiplication for frontier propagation
One Thread per Node on 8800 GTX

Early (2007) work: BFS with parallelization over nodes.*
*• Arrays track whether nodes are in frontier and visited.
*• In an iteration, each thread tests whether node is in frontier.
*• If so, thread
  – Marks node visited and removes node from frontier.
  – Goes over all unvisited neighbors
  – Adding to frontier and updating distance.
*• Claims to predate atomics, but … algorithm requires atomics for correctness (and doesn’t use them).


Complexity Too High for Many Graphs

• L iterations required,
  – where L is the maximum shortest path length,
  – O(diameter of graph), 1000+ for large graphs.
• complexity O(VL + E)
  – too much extra work!
  – especially for sparse / high diameter graphs
  – usually slower than CPU for large graphs
• And, if you add atomics…

BFS Using Sparse Matrix-Vector Multiply

CAD community, 2009*
• BFS via sparse matrix-vector multiply (SpMV):
• matrix is “transposed” adjacency in CSR: directed edges from columns to rows, and
• vector is reachable set (in some # of hops)
  with path count per element.
• SpMV gives next reachable set with path count per element.
• Path counts also have value in EDA.
Slower than CPU for Large Graphs

Complexity $O((V+E)L)$ (L executions of SpMV)
- Slower than CPU for large graphs.
- Can precompute matrix powers to speed up,
  - but matrix-matrix is slow, and
  - matrix powers are larger.
$A^kv$ is a superset of nodes at distance $k$.
- If desired, remove nodes at distance $< k$ between multiplications.
- Added algorithm-specific work makes algebraic solutions less attractive.

Simplification Produces a Gather Pattern

Can also reduce computation (losing path counts):
1. Start with vector of “infinity” (except for source).
2. Check vector value for each row and skip computation of non-infinite row values.
3. Check that edge destinations have value “infinity”; stop if non-infinite value found, producing (dest value + 1).
With these changes,
- computation is a Gather pattern (no atomics!):
  - in each iteration, elements “gather” from neighbors.
Complexity still $O(V+EL)$, but constants smaller.

Need More Flexibility and Information

To address the problem more generally, need to be more flexible.

- Keep the big picture in mind:
  - What needs to get done?
  - What are the characteristics of the data?
  - Where are the data?
  - What are the capabilities of the hardware?

Make Timely Choices from Multiple Algorithms

In practice, often need several algorithms.

Sometimes, we can choose the right one when writing the application.
Sometimes, we need to determine the right one dynamically.

Fortunately, this approach is well understood!
Example ca. 1990: CMSSL (CM-2)

Lennart Johnson put together the Connection Machine Scientific Software Library with these goals in mind.
- **Track** dynamic information about **distribution of data** (in a distributed memory machine).
- **Given the next** desired **operation**,
  - **compute cost models** for several algorithms based on data distribution, then
  - **choose the fastest**.
- Execute and **update info on** dynamic data **distribution**.
- Could optimize over several operations, too.

Examples Include FFTW and NVIDIA Libraries

- **FFTW** is a more **recent example** focused on FFT (DFT).
  - Use C++ templates to reduce overhead.
  - Hard to read the code until you’ve seen some of the techniques explained.

Another Example: Matrix Multiplication

Algorithm variations come up in 408 projects, too.

Consider **dense matrix multiplication**:
- **approaches** we’ve **discussed**
- **work** well for **square matrices**…

Certain Shapes Require More Parallelism

**What about this shape?**

Not many outputs to parallelize!

Instead, parallelize over inner-product terms and use reductions per output.
Be Sure to Examine Known Techniques

What about the architecture?
Important to realize that
• distributed resources in modern architectures are an outcome of the laws of physics.
• They haven’t changed recently.
• Techniques from distributed memory machines can also be applied in GPUs, as
  “shared” memory is actually private scratchpads distributed amongst SMs, as you know.

Alternate Between Local and Global

Distributed memory demands hierarchy.
Fast data are implicitly private, so privatization is key.
• Need to execute fast locally.
• Need to share information globally.
• Most effective algorithms go back and forth (local/global) as few times as possible.

Example: Sorting on a Cluster
A fun variant of these ideas
• won the world record for sorting back in 1997.*
• The context—a cluster of workstations—forces privatization and hierarchy.
• Sample globally for splitters, split locally, exchange data globally, sort locally.

Need a Queue to Adapt Sequential BFS

Back to the problem at hand: BFS.
We have…
• irregular data (somewhat tamed with CSR),
• irregular access patterns (data-dependent), and
• varying levels of parallelism.

Sequential solution relies on queue.

Review Execution on a Simple Graph

Frontier processed in parallel!

Need a More General Technique

- Why “queue” instead of queue?
  - Serializable parallel queue
  - Parallel operations “appear” to have executed in some serial order on a sequential queue.
  - Same order observed by all users.

- BFS has little computation—requires efficiency:
  - hierarchical scalable queue implementation,
  - and hierarchical kernels.

Global Queue is Too Slow

First attempt: serializable queue in global memory.

- Parallelize over nodes in frontier.
- Dequeue in parallel.
- Enqueue using global atomic operations:
  - poor coalescing, and
  - low global atomic throughput.
- Complexity $O(V+E)$.
- No speedup.

Can Reduce Pressure by Compacting Later

GPU community did try alternatives.*

- Separate generation and compaction, using a fixed-size output array per thread, thus no atomics and no contention.
- Overhead of compaction may outweigh gains in some cases (with little work per node, as in BFS).

Leverage Throughput of Shared Memory

**Maxwell generation** of GPUs
- substantially increased throughput
- for shared memory atomics,
- enabling GPUs to leverage older techniques.

**Graph algorithms were a big reason**, along with image processing, vision, and feature extraction.

Privatization Reduces Contention

Reduce contention by
- replicating data,
- accessing with fewer (or 1) threads,
- and aggregating more efficiently later.
This technique is called **privatization**.

Privatization Enables Use of Private Memories

Privatization requires
- extra storage, but
- enables copies to reside in private memories, such as
  - shared memory in GPUs.

Privatization Applies to BFS Queues

- For BFS, each **thread processes one** (or more) node.
- When node found for next frontier,
  - insert to local queue in shared memory
  - (shared among threads in block).
- When all threads in block done,
  - allocate space (one atomic) in global queue, then
  - copy from local to global.
Two-level Hierarchy

Result is two-level hierarchy:
- block queue (b-queue)
  - in shared memory
  - used by threads in a block,
- global queue (g-queue) inserted only when block completes.

B-Queue Must Fit in Shared Memory

Shared memory is limited.
One option: overflow into global queue.
Another option: flush and continue. May be tricky to synchronize—want all threads to cooperate in flush.

Benefits of Two-Level Hierarchy

privatization benefits:
- contention on global queue reduced to one operation per block
- contention on block queues still a problem (but less so)

shared memory benefits:
- higher throughput atomic queue insertions
- cooperative, coalesced writes to global queue.

Threads in Warp ALWAYS Collide!

b-queue contention is a problem.
- The worst part? SIMD execution to one resource!
- Threads executed together in a cycle are guaranteed to collide.
All threads inserting to b-queue must do so in separate cycles.

Collisions within a Warp are Slow

What happens?

Threads normally executed in a cycle are serialized across many cycles / issue slots.

Solution: Add a Warp-level Queue

- Solution? Add a level of hierarchy!
- Provide queues for SIMD lanes:
  - shared across warps in a block;
  - reduces conflicts in a cycle;
  - more queues, fewer conflicts (up to warp size).

Warp Queue Reduces SIMD Contention

Warp queues (w-queues)
  - placed in shared memory and
  - split across SIMD lanes in warp (executed in same cycle), but
  - shared across warps in thread block (less likely to conflict).
  - Example: use wQueue[threadIdx.x % 8].
  - Must still use atomic operations (different warps can execute simultaneously).

BFS Extended to Use Warp Queues

- For BFS, each thread processes one (or more) node.
- When node found for next frontier,
  - insert to warp queue in shared memory
  - (shared among threads in block).
- When all threads in block done,
  - consolidate warp queues into block queue,
  - allocate space (one atomic) in global queue, then
  - copy from local to global.
Need to Think About Bank Conflicts

**Problem: bank conflicts!**

Multi-ported memory is expensive.
- No one builds memory that way.
- Instead, SM’s **shared memory has many banks.**

**Adjacent memory locations**
- placed in different banks, and
- serviced in parallel.

**Strided memory locations**
- may hit the same bank(s) and
- create bank conflicts.

---

Need to Think About Bank Conflicts

**Why didn’t we mention earlier (in 408)?**
- Dropped after first few years, as
- usually **less important for performance.**

**What does this have to do with w-queues?**
- Likely to use $2^k$ w-queues of length $2^m$, but
- **power-of-2 strides of lead to bank conflicts.**

**Solution:** interleave (swap dimensions)!

---

Hierarchical Queue Memory Management

**Shared Memory**
- divided amongst w-queues and b-queues.
- interleaved w-queue layout
  reduces bank conflicts
  (example with $N = 8$)

Global Memory holds g-queue;
- writes from b-queue to g-queue coalesced
Texture memory stores graph structure
- (random access, no coalescing)

---

Three-Level Hierarchy Illustration
Notice the sparse filling due to imbalance between w-queues.

Interleaved Warp-Level Queue Layout

How many w-queues?

More w-queues
• requires more space (or more overflows)
• leads to more load imbalance, and
• causes more bank conflicts (serialization).

Fewer w-queues causes
more SIMD atomic conflicts (serialization).

Be sure to allow >1 thread block per SM!

Benefits of Three-Level Hierarchy

privatization benefits:
• contention on g-queue and b-queues reduced to one operation per block
• less contention on w-queues

shared memory benefits:
• higher throughput atomic queue insertions
• cooperative, coalesced writes to global queue.

Queue Strategy Reuse and Limitations

Techniques
• applicable to any sequential data structure
• provided that operations can be reordered
• (harder for a heap, for example, hence many more BFS papers than shortest path papers!).

Local queues
• limited by capacity of shared memory.
• If node degree is bounded, can estimate well.
• Later, we’ll use dynamic parallelism to bound.
Thoughts on the Details

**Why keep the b-queue?**
- w-queues overflow to b-queue, which
- **Smooths load** before overflow to g-queue.

**Why not copy w-queue directly to g-queue?**
- That is, scan across w-queues and b-queue, allocate space for all once, then copy all using scan results.
- **Not clear.** Possibly control complexity. **Try it?**

More Thoughts on the Details

**Why not use Cub’s warp scans to avoid conflicts instead of adding w-queues?**
- Fairly new … try it?

**Why not use register tiling to accumulate 4-8 nodes first?**
- **Try it? (Hint: loop must be unrolled…)**

Using CPU and Kernels to Deal with Variable Parallelism

- **How can we reduce the cost of kernel launches?**
  - As mentioned before, **use CPU to explore first few levels.**
  - Then **use multiple kernels** (and dynamic parallelism) to adjust to available parallelism.

Hierarchical Kernel Arrangement

- **Customize kernels based on the size of frontiers.**
- Use barrier synchronization (__syncthreads()) when frontier is small.

![0-hop kernel](image)

![1-hop kernel](image)

![2-hop kernel](image)

![3-hop kernel](image)

![Kernel 1: Intra-block Sync.](image)

![Kernel 2: Kernel re-launch](image)
First Kernel: So Long as Frontier Fits a Block

Kernel 1: small-sized frontiers
- one thread block
- \_syncthreads() between frontiers
- uses only w-queues and b-queue

Switch kernels when b-queue overflows to g-queue.

Second Kernel Handles Large Frontiers

Kernel 2: large frontiers
- multiple thread blocks
- kernel re-launch to synchronize (overhead acceptable compared to large frontier processing time)
- uses full hierarchy of queues

Or use dynamic parallelism (Kepler or later GPUs)
- to launch Kernel 2 from Kernel 1
- when frontier size threshold exceeded.

Still Need Global Atomics … ?

Actually, we didn’t eliminate global atomics.

- Using hierarchical queues
  - reduces contention and
  - improves throughput.
  - However, it also
decouples the threads’ actions.

Competing for Insertion Rights is Atomic

What if two nodes in a frontier share an unvisited neighbor?

We want only one copy in the next frontier!
Must coordinate insertion.

Here we expect little contention,
- so continue to use global atomics
- to guarantee desired behavior.
(Use atomic exchange on visited array in Lab 5.)
Performance Data: Not as Easy as It Sounds…

Lumetta’s implementations
We’ll do this as times to beat…

<table>
<thead>
<tr>
<th>Nodes of Degree</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0.1029 msecs</td>
</tr>
<tr>
<td>4</td>
<td>17.2 msecs</td>
</tr>
<tr>
<td>8</td>
<td>30.8 msecs</td>
</tr>
</tbody>
</table>

Some hints: (use exclusive queue for final timing!)
- lots of threads
- plenty of queue
- don’t think you’re going to fit everything at once
- just use bqueues
- try inventing some tricks…

ANY QUESTIONS?