Single-Node MultiThreaded Graph Computations with Kokkos

Jonathan W. Berry
Sandia National Laboratories

April 25, 2019
Outline/Intro

- MultiThreaded Graph Library (MTGL) history
- The MTGL vision
- The MTGL API
- Coding example: Two strategies for PageRank
- Status and conclusions
The MTGL’s History

- 2006-8: MultiThreaded Graph Library (Berry, Mackey)
- 2008-2014: MTGL stasis (no support)
- 2014: MTGL/Kokkos prototype
- 2019: MTGL/Kokkos maturing

Based on the Boost Graph Library, but multithreaded and without Boost
MTGL Contributors

- **Leadership**
  - J. Berry (PI)
  - Greg Mackey (main designer/developer)

- **Code Contributors**
  - W. McLendon (1400)
  - K. Devine (1400)
  - K. Landin (5800)
  - K. Madduri (Penn St.)

- **Software Engineering Help**
  - D. Dunlavy
  - N. Ellingwood
  - K. Landin
The MTGL Vision

- Application (e.g. KokkosKernels-based) init’s data structures

- “Dataless adapters” provide the MTGL (.hpp files) with access to the data

- MTGL algorithms run and generate artifacts (e.g. vertex, edge attributes)

- The dataless adapter goes out of scope and the KokkosKernels application continues
The MTGL API

- The vision is made possible by an API that hides data structure details

- Algorithms are written using
  - templated datatypes
  - Iterator objects
  - Generic functions

- Kokkos details like execution spaces, etc. are embedded in the templating and construction of these features

- The “same” code can run on adjacency lists, CSR, sponsor data structures, KokkosKernels, etc.
### The MTGL API: Whirdwind Tour

<table>
<thead>
<tr>
<th>Sizes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>size_type</td>
<td>The type that represents sizes used to index vertex and edge arrays</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Descriptors</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vertex_descriptor</td>
<td>The handle used to refer to a vertex</td>
</tr>
<tr>
<td>edge_descriptor</td>
<td>The handle used to refer to an edge</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Vector Iterators</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vertex_iterator</td>
<td>Random access iterator through all vertices</td>
</tr>
<tr>
<td>edge_iterator</td>
<td>Random access iterator through all edges</td>
</tr>
<tr>
<td>adjacency_iterator</td>
<td>Random access iterator through all adjs of one vertex</td>
</tr>
<tr>
<td>in_adjacency_iterator</td>
<td>Random access iterator through all in-adjs of one vertex (if avail.)</td>
</tr>
<tr>
<td>out_edge_iterator</td>
<td>Random access iterator through all incident edges leaving a vertex</td>
</tr>
<tr>
<td>in_edge_iterator</td>
<td>Random access iterator through all incident edges pointing to a vertex</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Thread Iterators</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>thread_vertex_iterator</td>
<td>Incremental access iterator through all vertices</td>
</tr>
<tr>
<td>thread_edge_iterator</td>
<td>Incremental access iterator through all edges</td>
</tr>
<tr>
<td>thread_adjacency_iterator</td>
<td>Incremental access iterator through all adjs of one vertex</td>
</tr>
<tr>
<td>thread_in_adjacency_iterator</td>
<td>Incremental access iterator through all in-adjs of one vertex (if avail.)</td>
</tr>
<tr>
<td>thread_out_edge_iterator</td>
<td>Incremental access iterator through all incident edges leaving a vertex</td>
</tr>
<tr>
<td>thread_in_edge_iterator</td>
<td>Incremental access iterator through all incident edges pointing to a vertex</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Categories</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>directed_category</td>
<td>{undirected, directed, or bidirectional}</td>
</tr>
<tr>
<td>iterator_category</td>
<td>the graph either supports vector iterators, thread iterators, or both</td>
</tr>
</tbody>
</table>
The MTGL API: Whirdwind Tour

<table>
<thead>
<tr>
<th>Handles for Graph Object</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Sizes</th>
<th>Descriptors</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>size_type</td>
<td>The type that represents sizes used to index vertex and edge arrays</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Vector Iterators</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>vertex_iterator</td>
<td>Random access iterator through all vertices</td>
</tr>
<tr>
<td>edge_iterator</td>
<td>Random access iterator through all edges</td>
</tr>
<tr>
<td>adjacency_iterator</td>
<td>Random access iterator through all adjs of one vertex</td>
</tr>
<tr>
<td>in_adjacency_iterator</td>
<td>Random access iterator through all in-adjs of one vertex (if avail.)</td>
</tr>
<tr>
<td>out_edge_iterator</td>
<td>Random access iterator through all incident edges leaving a vertex</td>
</tr>
<tr>
<td>in_edge_iterator</td>
<td>Random access iterator through all incident edges pointing to a vertex</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Thread Iterators</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>thread_vertex_iterator</td>
<td>Incremental access iterator through all vertices</td>
</tr>
<tr>
<td>thread_edge_iterator</td>
<td>Incremental access iterator through all edges</td>
</tr>
<tr>
<td>thread_adjacency_iterator</td>
<td>Incremental access iterator through all adjs of one vertex</td>
</tr>
<tr>
<td>thread_in_adjacency_iterator</td>
<td>Incremental access iterator through all in-adjs of one vertex (if avail.)</td>
</tr>
<tr>
<td>thread_out_edge_iterator</td>
<td>Incremental access iterator through all incident edges leaving a vertex</td>
</tr>
<tr>
<td>thread_in_edge_iterator</td>
<td>Incremental access iterator through all incident edges pointing to a vertex</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Categories</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>directed_category</td>
<td>{undirected, directed, or bidirectional}.</td>
</tr>
<tr>
<td>iterator_category</td>
<td>the graph either supports vector iterators, thread iterators, or both</td>
</tr>
</tbody>
</table>
The MTGL API: Whirdwind Tour

<table>
<thead>
<tr>
<th>Sizes</th>
<th>The type that represents sizes used to index vertex and edge arrays</th>
</tr>
</thead>
<tbody>
<tr>
<td>size_type</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Descriptors</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>vertex_descriptor</td>
<td>The handle used to refer to a vertex</td>
</tr>
<tr>
<td>edge_descriptor</td>
<td>The handle used to refer to an edge</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Vector Iterators</th>
<th>These provide random access to vertices and edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>vertex_iterator</td>
<td>Random access iterator through all vertices</td>
</tr>
<tr>
<td>edge_iterator</td>
<td>Random access iterator through all edges</td>
</tr>
<tr>
<td>adjacent_iterator</td>
<td>Random access iterator through all in-adj of one vertex (if avail.)</td>
</tr>
<tr>
<td>in_adjacency_iterator</td>
<td>Random access iterator through all incident edges leaving a vertex</td>
</tr>
<tr>
<td>out_edge_iterator</td>
<td>Random access iterator through all incident edges pointing to a vertex</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Thread Iterators</th>
<th>These provide incremental access to vertices and edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>thread_vertex_iterator</td>
<td>Incremental access iterator through all vertices</td>
</tr>
<tr>
<td>thread_edge_iterator</td>
<td>Incremental access iterator through all edges</td>
</tr>
<tr>
<td>thread_adjacency_iterator</td>
<td>Incremental access iterator through all adjs of one vertex</td>
</tr>
<tr>
<td>thread_in_adjacency_iterator</td>
<td>Incremental access iterator through all in-adj of one vertex (if avail.)</td>
</tr>
<tr>
<td>thread_out_edge_iterator</td>
<td>Incremental access iterator through all incident edges leaving a vertex</td>
</tr>
<tr>
<td>thread_in_edge_iterator</td>
<td>Incremental access iterator through all incident edges pointing to a vertex</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Categories</th>
<th>These describe global attributes of the graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>directed_category</td>
<td>{undirected, directed, or bidirectional}.</td>
</tr>
<tr>
<td>iterator_category</td>
<td>the graph either supports vector iterators, thread iterators, or both</td>
</tr>
</tbody>
</table>
The MTGL API: Whirdwind Tour

<table>
<thead>
<tr>
<th>Sizes</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>size_type</td>
<td>The type that represents sizes used to index vertex and edge arrays</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Descriptors</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>vertex_descriptor</td>
<td>The handle used to refer to a vertex</td>
</tr>
<tr>
<td>edge_descriptor</td>
<td>The handle used to refer to an edge</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Vector Iterators</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>vertex_iterator</td>
<td>Random access iterator through all vertices</td>
</tr>
<tr>
<td>edge_iterator</td>
<td>Random access iterator through all edges</td>
</tr>
<tr>
<td>adjacency_iterator</td>
<td>Random access iterator through all adjs of one vertex</td>
</tr>
<tr>
<td>in_adjacency_iterator</td>
<td>Random access iterator through all in-adjs of one vertex (if avail.)</td>
</tr>
<tr>
<td>out_edge_iterator</td>
<td>Random access iterator through all incident edges leaving a vertex</td>
</tr>
<tr>
<td>in_edge_iterator</td>
<td>Random access iterator through all incident edges pointing to a vertex</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Thread Iterators</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>thread_vertex_iterator</td>
<td>Incremental access iterator through all vertices</td>
</tr>
<tr>
<td>thread_out_edge_iterator</td>
<td>Incremental access iterator through all incident edges leaving a vertex</td>
</tr>
<tr>
<td>thread_in_edge_iterator</td>
<td>Incremental access iterator through all incident edges pointing to a vertex</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Categories</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>directed_category</td>
<td>{undirected, directed, or bidirectional}.</td>
</tr>
<tr>
<td>iterator_category</td>
<td>the graph either supports vector iterators, thread iterators, or both</td>
</tr>
</tbody>
</table>
Kokkos glue is embedded in Graph type

```
// find_triangles signature
template <typename Graph, typename Visitor>
void find_triangles(Graph& g, Visitor& tri_visitor);
```

The Visitor Pattern enables generic programming
In this case, “tri_visitor” tells us what to do with each triangle

```
// calling an MTGL algorithm (a Graph g exists already)
find_triangles(g, tri_visitor);
```

```
// calling an MTGL algorithm on a subgraph (identical function calling convention)
subgraph_adapter<Graph> subg(g);
init_vertices(bool_vertex_mask, subg);
find_triangles(subg, tri_visitor);
```

Call the same algorithm on any graph – even a subgraph of our graph
Key Point: Do things between calls

Ewvis is a visitor object encapsulating “result”

```cpp
find_triangles(g, ewvis);
std::cout << "found triangles" << std::endl;
forall_edges(g, KOKKOS_LAMBDA (const edge_descriptor& e)
{
    size_type num_tri = result[e];
    vertex_descriptor u = source(e, g);
    vertex_descriptor v = target(e, g);
    size_type udeg = out_degree(u, g);
    size_type vdeg = out_degree(v, g);
    if (udeg <= 1 && vdeg <= 1) {
        result[e] = 0.0;
    } else {
        result[e] = 2 * num_tri / (double)((out_degree(u, g)-1) +
                                            (out_degree(v, g)-1));
    }
});
```

- I’m done with alg call here
- The alg stored its result in an edge map “result”
- I loop through the edges using the algorithm result
- And obtain what I really need

Library Design Pain Point: what syntax for “forall_edges”? 
“Manhattan Loop Collapse”

- Work is grouped into blocks; within a block:
  - Each thread has an outer range (its vertices)
    - e.g. outer_range(green thread) = [1,3]
    - e.g. outer_range(blue thread) = [4,6]
  - Each thread has an inner range (its adjacencies)
    - e.g. inner_range(green thread)[1] = [0,1]
    - e.g. inner_range(blue thread)[4] = [4,5]
PageRank: Accumulate Ranks (S1)

- Strategy 1: one thread per block

```c++
parallel_for(0, num_blocks, KOKKOS_LAMBDA (const size_type block_id) {
    // assign outer, inner (not shown)
    // get an incremental iterator (allows us to run on non-CSC)
    thread_vertex_iterator tverts = thread_vertices(begin_outer, g);

    for (size_type i = begin_outer; i != end_outer; ++i, ++tverts)
    {
        size_type begin_inner =
            begin_manhattan_inner_range(accum_deg, begin_pos, begin_outer, i);
        size_type end_inner =
            end_manhattan_inner_range(accum_deg, end_pos, end_outer, i);

        vertex_descriptor u = *tverts;
        thread_adjacency_iterator tadj_verts =
            thread_adjacent_vertices(u, begin_inner, g);
        size_type deg = out_degree(u, g);

        for (size_type j = begin_inner; j != end_inner; ++j, ++tadj_verts)
        {
            vertex_descriptor v = *tadj_verts;
            Kokkos::atomic_fetch_add(&acc[v], rank[u] / deg);
        }
    }
});
```

acc[5] = r[4]/2 + r[8]/2
PageRank: Accumulate Ranks (S1)

- Strategy 1: one thread per block

```cpp
parallel_for(0, num_blocks, KOKKOS_LAMBDA (const size_type block_id) {
    // assign outer, inner (not shown)
    // get an incremental iterator (allows us to run on non-CSV)
    thread_vertex_iterator tverts = thread_vertices(begin_outer, g);
    for (size_type i = begin_outer; i != end_outer; ++i, ++tverts)
    {
        size_type begin_inner =
            begin_manhattan_inner_range(accum_deg, begin_pos, begin_outer, i);
        size_type end_inner =
            end_manhattan_inner_range(accum_deg, end_pos, end_outer, i);
        vertex_descriptor u = *tverts;
        thread_adjacency_iterator tadj_verts =
            thread_adjacent_vertices(u, begin_inner, g);
        size_type deg = out_degree(u, g);
        for (size_type j = begin_inner; j != end_inner; ++j, ++tadj_verts)
        {
            vertex_descriptor v = *tadj_verts;
            Kokkos::atomic_fetch_add(&acc[v], rank[u] / deg);
        }
    }
}}
```

Consider the green thread

```
"I AM RESPONSIBLE FOR GRAPH VERTICES 1,2,3!"
```
PageRank: Accumulate Ranks (S1)

- Strategy 1: one thread per block

```
parallel_for(0, num_blocks, KOKKOS_LAMBDA (const size_type block_id)
{
    // assign outer, inner (not shown)
    // get an incremental iterator (allows us to run on non-CSV)
    thread_vertex_iterator tverts = thread_vertices(begin_outer, g);

    for (size_type i = begin_outer; i != end_outer; ++i, ++tverts)
    {
        size_type begin_inner =
            begin_manhattan_inner_range(accum_deg, begin_pos, begin_outer, i);
        size_type end_inner =
            end_manhattan_inner_range(accum_deg, end_pos, end_outer, i);

        vertex_descriptor u = *tverts;
        thread_adjacency_iterator tadj_verts =
            thread_adjacent_vertices(u, begin_inner, g);
        size_type deg = out_degree(u, g);

        for (size_type j = begin_inner; j != end_inner; ++j, ++tadj_verts)
        {
            vertex_descriptor v = *tadj_verts;
            Kokkos::atomic_fetch_add(&acc[v], rank[u] / deg);
        }
    }
}
```

"I AM RESPONSIBLE FOR ADJACENCIES [0,1] of GRAPH VERTEX 1!"

Consider the green thread
PageRank: Accumulate Ranks (S1)

- Strategy 1: one thread per block

```
parallel_for(0, num_blocks, KOKKOS_LAMBDA (const size_type block_id)
{
    // assign outer, inner (not shown)
    // get an incremental iterator (allows us to run on non-CSQ)
    thread_vertex_iterator tverts = thread_vertices(begin_outer, g);
    for (size_type i = begin_outer; i != end_outer; ++i, ++tverts)
    {
        size_type begin_inner =
            begin_manhattan_inner_range(accum_deg, begin_pos, begin_outer, i);
        size_type end_inner =
            end_manhattan_inner_range(accum_deg, end_pos, end_outer, i);
        vertex_descriptor u = *tverts;
        thread_adjacency_iterator tadj_verts =
            thread_adjacent_vertices(u, begin_inner, g);
        size_type deg = out_degree(u, g);
        for (size_type j = begin_inner; j != end_inner; ++j, ++tadj_verts)
        {
            vertex_descriptor v = *tadj_verts;
            Kokkos::atomic_fetch_add(&acc[v], rank[u] / deg);
        }
    }
}
```

"I AM PROCESSING MY FIRST ADJECENCY NOW, AND INCREMENTING acc[4]"

Consider the green thread
PageRank: Accumulate Ranks (S1)

- Strategy 1: one thread per block

```cpp
parallel_for(0, num_blocks, KOKKOS_LAMBDA (const size_type block_id) {
    // assign outer, inner (not shown)
    // get an incremental iterator (allows us to run on non-CSV)
    thread_vertex_iterator tverts = thread_vertices(begin_outer, g);
    for (size_type i = begin_outer; i != end_outer; ++i, ++tverts) {
        size_type begin_inner =
            begin_manhattan_inner_range(accum_deg, begin_pos, begin_outer, i);
        size_type end_inner =
            end_manhattan_inner_range(accum_deg, end_pos, end_outer, i);
        vertex_descriptor u = *tverts;
        thread_adjacency_iterator tadj_verts =
            thread_adjacent_vertices(u, begin_inner, g);
        size_type deg = out_degree(u, g);
        for (size_type j = begin_inner; j != end_inner; ++j, ++tadj_verts) {
            vertex_descriptor v = *tadj_verts;
            Kokkos::atomic_fetch_add(&acc[v], rank[u] / deg);
        }
    }
});
```

"I HAVE MOVED ON, AND NOW I AM PROCESSING MY SECOND ADJACENCY, AND INCREMENTING acc[5]"

Consider the green thread
Strategy 2: explicitly use teams to stride more efficiently on GPU

Consider the second blue thread (T1)

Blue team (4 threads) handles this block
Strategy 2: explicitly use teams to stride more efficiently on GPU

```
strategy_policy work(num_blocks, Kokkos::AUTO());
Kokkos::parallel_for(work, KOKKOS_LAMBDA (const member_type& member) {
    // Each team searches 'block_size' indices. The threads of a team iterate together. Each iteration searches
    // [team_begin .. team_begin + member.team_size()].
    size_type team_begin = member.team_rank() * block_size;
    size_type team_end = team_begin + block_size;
    size_type member_index = team_begin + member.team_rank();
    size_type vert_index, adj_index, deg;
    if (member_index < total_work)
    {
        vert_index =
            begin_manhattan_outer_range(accu_deg, order, member_index);
        adj_index = member_index - accu_deg[vert_index];
        deg = accu_deg[vert_index + 1] - accu_deg[vert_index];
    } while (team_begin < team_end)
    {
        // If this thread's search index is in the range, do the work.
        if (member_index < total_work)
        {
            vertex_descriptor u = viter[vert_index];
            vertex_descriptor v = adjacent_vertices(u, g)[adj_index];
            Kokkos::atomic_fetch_add(acc[v], rank[u] / deg);
        }
    }
    // Move the team's search range forward by team size.
    team_begin = member.team_size();
    member_index = member.team_size();
    if (member_index < total_work)
    {
        // Move the vertex and adjacency forward to the next team range.
        size_type num_left = member.team_size();
        while ((deg - adj_index) <= num_left)
        {
            num_left = (deg - adj_index);
            ++vert_index;
            adj_index = 0;
            deg = accu_deg[vert_index + 1] - accu_deg[vert_index];
        }
    }
});
```

Consider the second blue thread (T2)

Blue team (4 threads) handles this block

"MY WORK INDEX IS 1!"
Strategy(2)

- Strategy 2: explicitly use teams to stride more efficiently on GPU

“MY GRAPH VERTEX IS 1!”

Blue team (4 threads) handles this block

Consider the second blue thread (T2)
Strategy(2)

- Strategy 2: explicitly use teams to stride more efficiently on GPU

```
// Each team searches 'block_size' indices. The threads of a team iterate together. Each iteration searches
// [team_begin .. team_begin + member.team_size()].
size_type team_begin = member.league_rank() * block_size;
size_type team_end = team_begin + block_size;
size_type member_index = team_begin + member.team_rank();
size_type vert_index, adj_index, deg;
if (member_index < total_work)
{
    vert_index =
        begin_manhattan_outer_range(accum_deg, order, member_index);
    adj_index = member_index - accum_deg[vert_index];
    deg = accum_deg[vert_index + 1] - accum_deg[vert_index];
}
```

```
vertex_descriptor u = viter[vert_index];
vertex_descriptor v = adjacent_vertices(u, g)[adj_index];
Kokkos::atomic_fetch_add(&acc[v], rank[u] / deg);
```

```
while (team_begin < team_end)
{
    // If this thread's search index is in the range, do the work.
    if (member_index < total_work)
    {
        vertex_descriptor u = viter[vert_index];
        vertex_descriptor v = adjacent_vertices(u, g)[adj_index];
        Kokkos::atomic_fetch_add(&acc[v], rank[u] / deg);
    }
    // Move the team's search range forward by team size.
    team_begin += member.team_size();
    member_index += member.team_size();
    if (member_index < total_work)
    {
        // Move the vertex and adjacency forward to the next team range.
        size_type num_left = member.team_size();
        while ((deg - adj_index) <= num_left)
        {
            num_left = (deg - adj_index);
            ++vert_index;
            adj_index = 0;
            deg = accum_deg[vert_index + 1] - accum_deg[vert_index];
        }
        adj_index += num_left;
    }
}
```

---

Blue team (4 threads) handles this block

Consider the second blue thread (T2)

"I HAVE THE SECOND ADJ OF V1!"
Strategy(2)

- Strategy 2: explicitly use teams to stride more efficiently on GPU

"I JUST DID MY WORK (incr acc[5])!"

Consider the second blue thread (T2)
Strategy(2)

- **Strategy 2**: explicitly use teams to stride more efficiently on GPU

```
    team_policy work(num_blocks, Kokkos::AUTO());
    Kokkos::parallel_for(work, KOKKOS_LAMBDA (const member_type& member) {
        // Each team searches 'block_size' indices. The threads of a team
        // iterate together. Each iteration searches
        // [team_begin .. team_begin + member.team.size()].
        size_type team_begin = member.league_rank() * block_size;
        size_type team_end = team_begin + block_size;
        size_type member_index = team_begin + member.team.rank();
        size_type vert_index, adj_index, deg;
        if (member_index < total_work) {
            vert_index =
                begin_manhattan_outer_range(accu_deg, order, member_index);
            adj_index = member_index - accu_deg[vert_index];
            deg = accu_deg[vert_index + 1] - accu_deg[vert_index];
        } while (team_begin < team_end) {
            // If this thread's search index is in the range, do the work.
            if (member_index < total_work) {
                vertex_descriptor u = viter[vert_index];
                vertex_descriptor v = adjacent_vertices(u, g)[adj_index];
                Kokkos::atomic_fetch_add(acc[v], rank[u] / deg);
            } // Move the team's search range forward by team size.
            team_begin += member.team.size();
            if (member_index < total_work) {
                // Move the vertex and adjacency forward to the next team range.
                size_type num_left = member.team.size();
                while ((deg - adj_index) <= num_left) {
                    num_left = (deg - adj_index);
                    ++vert_index;
                    adj_index = 0;
                    deg = accu_deg[vert_index + 1] - accu_deg[vert_index];
                }
                adj_index += num_left;
            }
    });
```

**“NOW I’VE MOVED ON!”**

Consider the second blue thread (T2)

Blue team (4 threads) handles this block

Block 0

```
1 2 3 4 5 6 7
```

Berry, – Kokkos User Group 2019
Strategy(2)

- Strategy 2: explicitly use teams to stride more efficiently on GPU

AND I’VE FIGURED OUT THAT MY NEXT JOB IS THE SECOND ADJACENCY OF V4!

Blue team (4 threads) handles this block

Consider the second blue thread (T2)
Comparison of Strategies

- Strategy 2 was more complicated to code
- Strategy 2 runs..... How much faster
- Guesses?
Comparison of Strategies

- Strategy 2 was more complicated to code
- Strategy 2 runs
  - About 40% faster on the “Brain Graph” (234M edges)
    - 0.77s for Strategy 2
    - 1.28s for Strategy 1

- Seems like a lot of work and complication for only 40% improvement

- I would welcome offline discussion of this
This talk is based on 2016 code

There is new 2019 code (SPP-sponsored)

We want to adapt this code to run over KokkosKernels::CSR

We want to release our code

- Either as the MTGL 2.0
- Or as a set of .hpp’s and API to enable graph computation as a part of KokkosKernels

Thank you!

2016 code: software.sandia.gov/trac/mtgl (“noxmt” branch)

New code: gitlab.sandia.gov:hpda-graphs

POC: jberry@sandia.gov