Rethinking Irregular Algorithms for Massive Multithreading

Jacopo Pantaleoni

NVIDIA Research
Parallelizing a complex algorithm involves finding a balance between conflicting goals:

- Massive Parallelism
- Work Efficiency
- Preserving Locality
- Load Balancing
Many algorithms have an efficient serial solution, and parallelizing requires strictly more work.

Finding the best compromise between the amount of work and communication is the key.
e.g. reduction

optimal $O(N)$ **serial** solution:

```c
sum = 0;
for (i = 0; i < N; ++i)
    sum += a[i];
```
e.g. reduction

trivial $O(N \log(N))$ **fully parallel** version:

```c
for (stride = N/2; stride > 0; stride /= 2) {
    if (threadId < stride)
        shared[ threadId ] += shared[ threadId + stride ];
    __syncthreads();
}
```
e.g. reduction

trivial $O(N \log(N))$ **fully parallel** version:
Parallelism vs Efficiency

e.g. reduction

=> better to have each thread do more **serial** work:
Exposing parallelism may be at odds with preserving spatio-temporal locality.

**e.g. tree traversal**

**depth-first** has **good locality** but **poor parallelism**

**breadth-first** offers **good parallelism** but **poor locality**
Parallelism vs Locality

e.g. tree traversal

+ good locality
- poor parallelism
Parallelism vs Locality

e.g. tree traversal

- good parallelism
- poor locality
Parallelism vs Locality

e.g. tree traversal

+ decent parallelism
+ decent locality
Locality vs Load Balancing

e.g. ray tracing
e.g. ray tracing

By packing similar work together, one might destroy locality

Packing within a window might be more effective
Parallelizing existing serial algorithms may need a **fresh new look**.

e.g. k-d tree building:
Parallelizing existing serial algorithms may need a fresh new look.

e.g. k-d tree building:
void split(kd_node& node, float2* points)
{
    if (node.end - node.begin <= 1)
        return;

    bbox = node.bbox;
    while (1) // keep splitting bbox along longest axis until effective
    {
        int split_axis = max_comp(bbox);
        float split_plane = bbox.center[ split_axis ];

        int pivot = partition(
            points + node.begin,
            points + node.end, partitioner( split_axis, split_plane ) );

        if (pivot == node.begin) // left partition is empty
            bbox.min[ split_axis ] = split_dim;
        else if (pivot == node.end) // right partition is empty
            bbox.max[ split_axis ] = split_dim;
        else
        {
            // valid split found: recurse in both children
            node.left = kd_node( node.begin, pivot );
            node.right = kd_node( pivot, node.end );
            split( node.left ); // recurse left
            split( node.right ); // recurse right
            break;
        }
    }
}
We could try to parallelize every split step in a breadth-first manner

But it needs a lot of book-keeping!
New idea: we can first classify & sort the points by Morton code (i.e. Z curve)
New idea: we can first classify & sort the points by Morton code (i.e. Z curve)
The n-D points get mapped to a sorted sequence of bit vectors.

⇒ Emitting the hierarchy corresponds to finding the first set bit in each range
k-d tree building

The n-D points get mapped to a **sorted** sequence of bit vectors.

⇒ Emitting the hierarchy corresponds to finding the first set bit in each range

```c
input_queue.push( 0, n_points );
output_queue.clear();

while (input_queue.size())
{
    foreach (node in input_queue) // parallel loop
    {
        int split_index = find_split(node.begin, node.end, node.level);
        if (split_index != INVALID_SPLIT)
        {
            output_queue.push( node.begin, split_index, node.level+1 );
            output_queue.push( split_index, node.end, node.level+1 );
        }
    }
    swap( input_queue, output_queue );
}
```
k-d tree building

Summary:

• Primitive Sorting (parallel LSD radix sort)

• Hierarchy Emission (parallel searching + queueing)
Sorting points into a grid was easy: 1:1 mapping between points and cells.

What if we had to rasterize large objects?

1:N relationship
The problem becomes that of \textbf{sorting} a large list of elements, where the input data is \textbf{highly compressed}, and the decompression rate has \textbf{high variability}:

each prim $p$ expands to $N(p)$ cells, $\text{Var}(N) \gg 1$
The problem becomes that of sorting a large list of elements, where the input data is highly compressed, and the decompression rate has high variability:

each prim $p$ expands to $N(p)$ cells, $\text{Var}(N) >> 1$

Two stage pipeline:

1. decompression
2. sorting

high var $\Rightarrow$ need for load balancing
two-pass approach:

coarse & fine raster  per tile & per pixel queues
Queues are a great primitive for work distribution.

How do we implement them efficiently?
1:1 case is easiest.

Some pipeline stage processes a single element and outputs 0 or 1 elements into a single output queue.

=> Stream Compaction
1:1 case is easiest.

Some pipeline stage processes a single element and outputs 0 or 1 elements into a single output queue.

template<typename T>
void push(const T item, T* queue, uint* queue_size)
{
    uint active_mask = __ballot( true );
    uint pop_count   = __popc( active_mask );
    uint pop_scan    = __popc( active_mask >> (warp_tid+1) );

    // each warp does a single atomic to get an offset...
    __shared__ sm_broadcast[ NUM_WARPS ];
    if (pop_scan == 0)
        sm_broadcast[ warp_id ] = atomicAdd( queue_size, pop_count );

    // ...which is then broadcast to every thread
    uint queue_offset = sm_broadcast[ warp_id ] + pop_scan;

    queue[ queue_offset ] = item;
}
1:N requires distinction:

• \( N \leq 4 \) => pop counting is great. Each warp does a warp scan, where we can pack up to 4 8-bit pop.counters in each integer.

• \( N \leq 64 \) => local shared memory sorting is usually a win

• \( N \) large => shared / global memory atomics