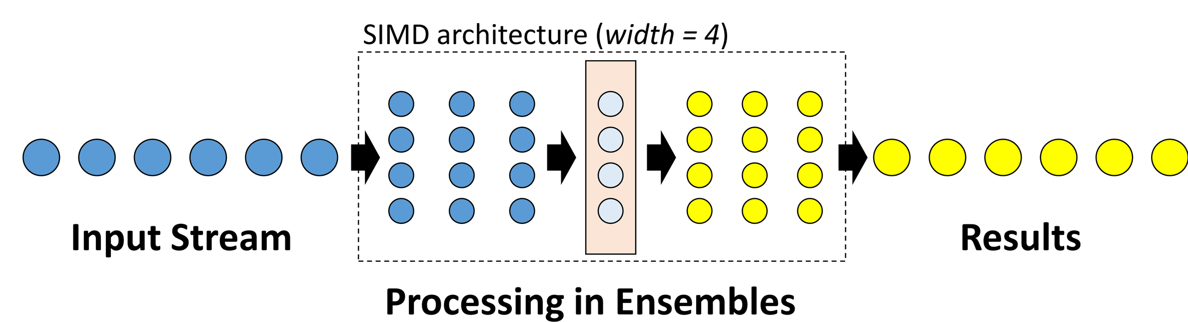
MERCATOR: Supporting Irregular Behaviors in Streaming SIMD Computations

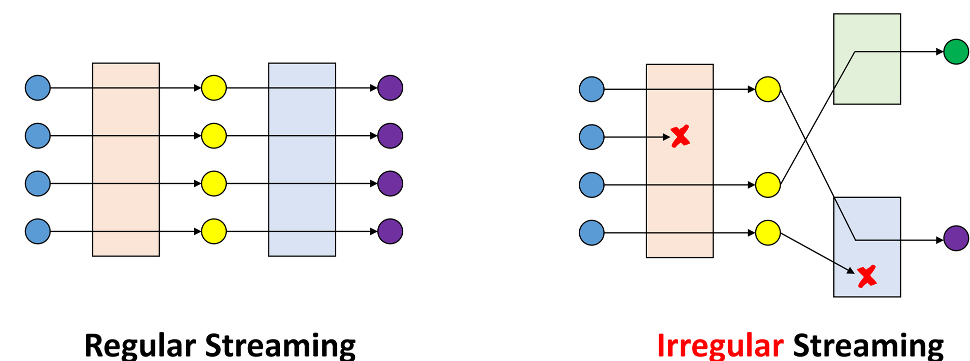
Steve Cole, Jeremy Buhler, and Roger Chamberlain

Dept. of Computer Science and Engineering, WUSTL

SIMD (Single-Instruction, Multiple-Data) architectures such as GPUs naturally support **streaming computations** that process a long stream of independent data items. Items may be grouped into **ensembles** of size equal to the SIMD width (e.g. GPU block size), with each element processed in parallel by a single **SIMD lane** (= one GPU thread).



Most streaming computing frameworks, such as StreamIt[2], primarily support **regular** streaming computations in which each item in an ensemble undergoes identical processing – hence every SIMD lane does the same work. In contrast, **irregular** computations may perform different, data-dependent work for each item. Irregularity leads to idled SIMD lanes and divergent computation, which reduce application throughput.



The MERCATOR framework[1] efficiently expresses irregular streaming computations for SIMD platforms.

- ❑ Abstracts irregular behaviors using dataflow representation of application
- ❑ Supports irregular behaviors (filtering, divergence) efficiently, transparently using SIMD-parallel algorithms
- ❑ Exposes opportunities for optimization

MERCATOR targets NVIDIA GPUs using the CUDA language.

Irregular streaming computations matter!

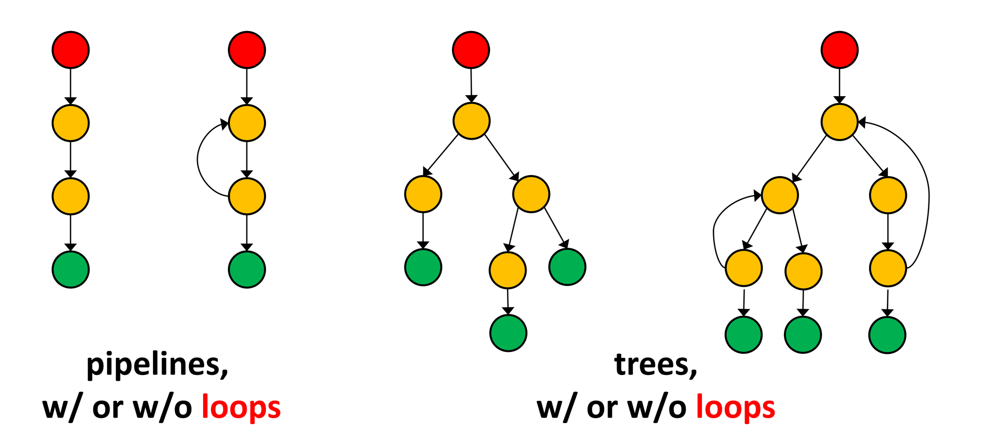
Decision cascades (image recognition, biosequence analysis, sensor data processing)

Pattern matching (e.g. network traffic)

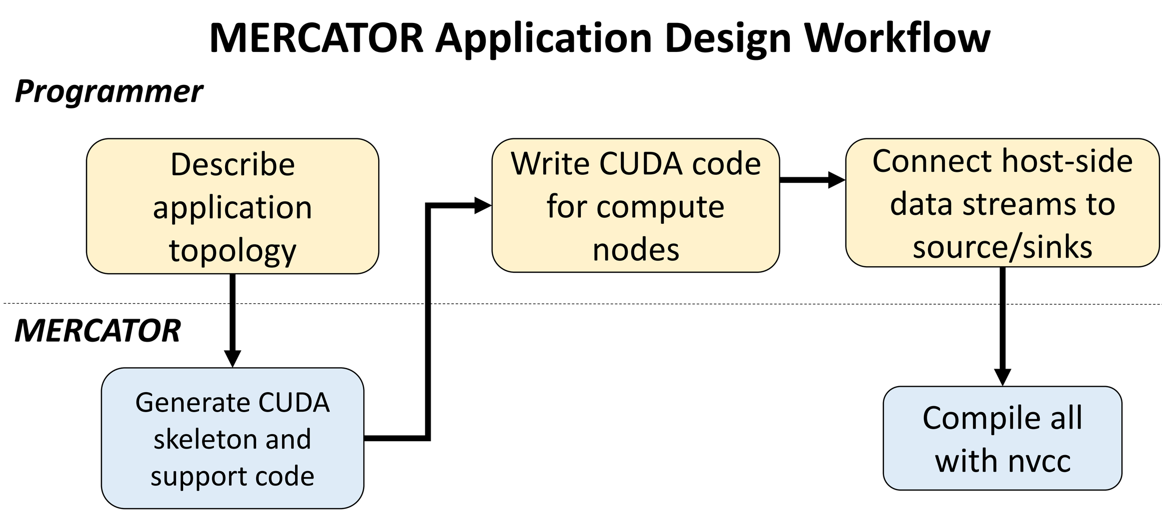
Tree traversal (decision tree evaluation, suffix tree string matching)

Big graph algorithms (belief propagation, page rank, shortest paths, clustering)

The Programmer's View



Applications are graphs of *compute nodes* connected by dataflow *edges*. Data flows in at the (unique) *source* and out at one or more *sinks*. Application graphs are **trees** but may be augmented with limited **back edges** to permit cycles.



Design principle: programmer need not explicitly code interaction among SIMD lanes!

Sample MERCATOR Application Topology Spec

```
// A module type describes one or more
// nodes implementing the same code
#pragma mtr module Filter (int[128]->acc<int>:?2)

// Node declarations (w/module types)
#pragma mtr node sourceNode : SOURCE
#pragma mtr node filterNode1: Filter
#pragma mtr node filterNode2: Filter
#pragma mtr node sinkNode : SINK<int>

// Edge declarations
#pragma mtr edge sourceNode::out->filterNode1
#pragma mtr edge filterNode1::acc->filterNode2
#pragma mtr edge filterNode2::acc->sinkNode
```

max inputs processed in one call to module; name of output port; these nodes produce 0-2 outputs per input

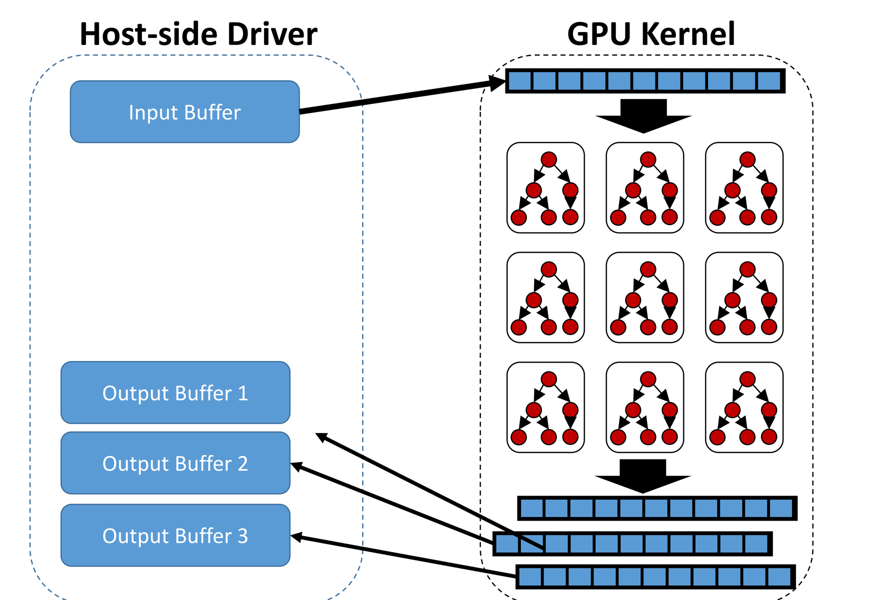
sourceNode, filterNode1, filterNode2, sinkNode

Nodes are specified with input and output data types, plus a *maximum* number of outputs produced per input on each outgoing edge. Nodes may filter or amplify their inputs data-dependently, so unlike in traditional SDF[3], output rates are dynamic.

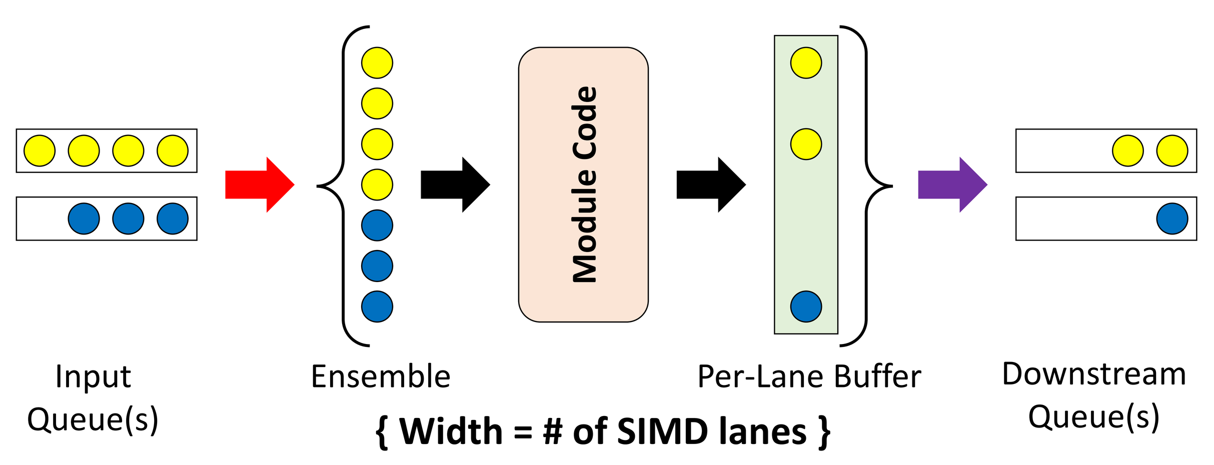
```
void Filter::run(int value, unsigned char nodeTag)
{
   int x = f(value);
   push(x, nodeTag, Filter::acc);

   int y = g(x);
   if (cond) push(y, nodeTag, Filter::acc);
}
```

The programmer implements each module type in a MERCATOR-generated function skeleton. ***Each SIMD lane*** receives an input value and a *node tag* indicating which node of this type received the value. Outputs are emitted ("*pushed*") to a specified port (connection to an edge) along with their tags, which preserve their node associations.

Under the Hood



A MERCATOR app is realized as an *uberkernel*[4], with all nodes and inter-node data management running on the GPU. The entire app runs independently on each GPU processor. Replicates on all processors share common source and sink data buffers.



Infrastructure Implementation Challenges

- ❑ Avoid deadlock in app graphs with cycles
- ❑ Limit allocation of scarce GPU per-processor memory
- ❑ Limit and coalesce GPU global memory traffic
- ❑ Reduce overhead using SIMD-friendly algorithms for
  - scheduling for module firings
  - managing queues between nodes
  - scatter/gather across queues for a module

MERCATOR inserts a **queue** in front of each node. When a module **fires**, enough queued items are *gathered* to occupy all SIMD lanes. If multiple nodes share the same module type, MERCATOR can combine their queues' contents in a single SIMD ensemble. Outputs are buffered in each SIMD lane, then *scattered* to the appropriate downstream queues based on the tags passed to `push()`.

Performance

Experiments[1] conducted on an NVIDIA GTX 980Ti GPU w/CUDA 8.0

Synthetic Benchmark

- Pipeline of 5 nodes + source, sink
- Each node passes on a fraction of its inputs between 0 and 1, drops rest
- Each node does compute-bound work (Black-Scholes iteration)
- Measured speedup of app with queues between nodes vs. baseline w/static work-to-SIMD-lane mapping

(Chart: Speedup vs. Filtering rate and Workload (ms per node))

Occupancy boost outweighed MERCATOR overhead for most filtering rates when node execution took 100+ ms.

(Chart: Execution Time (ms) vs. Query Size (DNA bases) 2000–10000; With queues / No queues)

DNA Sequence Comparison Kernel

- Seed matching, ungapped alignment pipeline from NCBI BLASTN[5]
- Compared human chromosome 1 (250 Mbases) to chicken seqs of 2-10 Kbases
- Again, compared MERCATOR app with queues between stages to baseline w/static mapping

Queueing boosted performance of BLASTN pipeline, especially for short queries.

MERCATOR efficiently abstracts irregular streaming GPU computations. Its runtime infrastructure helps performance in the presence of irregular behavior.

Future work

- ❑ Expand set of apps expressible in MERCATOR
  - *Nested loops*, with guaranteed deadlock freedom
  - *Tagged reduction* (e.g. edges of each vertex in a graph)
- ❑ Exploit optimization opportunities
  - *Trade off* occupancy vs. overhead to maximize throughput of nodes with small execution times
  - Allocate *multiple inputs* to each SIMD lane to hide latency
  - *Partition app execution* across GPU processors to load-balance nodes w/different service rates

References

1. SV Cole and J Buhler. "MERCATOR: a GPGPU framework for irregular streaming applications." *Proc. 15th Int'l Conf. High Performance Computing and Simulation*, to appear 2017.
2. W Thies, M Karczmarek, and S Amarasinghe. "StreamIt: a language for streaming applications." *Proc. Int'l Conf. Compiler Construction*, 179-96, 2002.
3. EA Lee and DG Messerschmitt. "Synchronous data flow." *Proc. IEEE* 75(9):1235-45, 1987.
4. S Tzeng, A Patney, and JD Owens. "Task management for irregular parallel workloads on the GPU." *Proc. Conf. High Performance Graphics*, 29-37, 2010.
5. SF Altschul, W Gish, W Miller, EW Myers, and DJ Lipman. "Basic local alignment search tool." *J. Molecular Biology* 215(3):403-10, 1990.

Research and development on MERCATOR is supported by NSF CISE award 1500173.