“Streaming” as a pattern

Peter Mattson, Richard Lethin
Reservoir Labs
“Streaming” as a pattern

• Streaming is a pattern in efficient implementations of computation- and data-intensive applications

• Pattern has three key characteristics:
  – Processing is largely parallel
  – Data access patterns are apparent
  – Control is high-level, steady, simple
## Causes of the pattern

<table>
<thead>
<tr>
<th>Processing is largely parallel</th>
<th>Applications process, simulate, or render physical systems</th>
<th>Architectural limits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data access patterns are apparent</td>
<td>Independence of spatially and/or temporally distributed data</td>
<td>VLIW, SIMD, multiprocessor demand for parallelism</td>
</tr>
<tr>
<td>Control is high-level, steady, simple</td>
<td>Few tasks per application, few unpredictable events</td>
<td>Handling unpredictability in hardware is expensive</td>
</tr>
</tbody>
</table>

**Application qualities**
- Streaming
- Architecture limits
Streaming is not only 1D

- Multidimensional data, data rearrangement
- Multidimensional architecture topologies
- Time is favored dimension, yielding 1D tendency
  - Unmapped, coarse-grain streams of whole-array “elements” moving between “actors” with whole-loop-nest invocations
  - Serialization for transmission, especially address stream to memory to get data stream to processor
Using streaming

- Streaming applications expose maximum opportunities, information to compiler
- Streaming architectures expose maximum resources, control to the compiler

- Streaming languages should enforce the pattern
  - Force programmer thought to reach pattern
  - Guarantee pattern to compiler

- Streaming compilers should exploit the pattern
  - Expand scope of application optimization
  - Expand scope of resource choreography
Streaming languages

- Enforce similar, but not identical patterns

<table>
<thead>
<tr>
<th>StreamIt</th>
<th>Brook/StreamC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Think structured synchronous</td>
<td>Think pointer-less C, with embedded dataflow graphs instead of loop nests</td>
</tr>
<tr>
<td>data flow</td>
<td>Multiple stream graphs, surrounded by C-subset</td>
</tr>
<tr>
<td>Single stream graph</td>
<td></td>
</tr>
<tr>
<td>Streams are infinite length</td>
<td>Streams are finite length</td>
</tr>
<tr>
<td>Static rates</td>
<td>Dynamic rates</td>
</tr>
<tr>
<td>“Filters” can have state, may</td>
<td>“Kernels” must be state-less, allow parallel processing</td>
</tr>
<tr>
<td>require sequential processing</td>
<td></td>
</tr>
<tr>
<td>Designed by compiler people,</td>
<td>Designed by application and architecture people, rough but more expressive</td>
</tr>
<tr>
<td>clean but more constrained</td>
<td></td>
</tr>
</tbody>
</table>
Streaming compilers

- Expand scope of optimization
  - Application is transparent to compiler
    - Access patterns, aliasing, etc. completely known
    - Top-down, not just bottom-up optimization
    - Exploit task parallelism
- Expand scope of choreography
  - Compiler lays out all computation, data, and communication
    - Closely model architecture
    - VLIW scheduling writ large
    - Difficult! Many variables, phase ordering…

```
Whole program
Multiple loop nests
Single loop nest
CSE etc.
ALUs, registers
Cache
Local memory
Multiprocessor, routing
```
R-Stream

- Reservoir Labs is developing the R-Stream high-level compiler
  - Goal is *portable, consistently high-performance* compilation
R-Stream

- Reservoir Labs is developing the R-Stream high-level compiler
  - Goal is portable, consistently high-performance compilation

- Streaming IR
- Top-down optimization
- Unified mapping of computation and data
Streaming Pattern

- Characterized by parallelism, apparent data access, high-level etc. control
- Driven by application class, architectural limits
- Enforced by languages, exploited by compilers
- Used by R-Stream; goal of portable, consistently high-performance compilation