R-Stream: Enabling efficient development of portable, high-performance, parallel applications

Peter Mattson, Allen Leung, Ken Mackenzie, Eric Schweitz, Peter Szilagi, Richard Lethin
{mattson, leunga, kenmac, schweitz, szilagyi, lethin @ reservoir.com}
Goals and Benefits

Portable, consistently high-performance compiler for streaming applications on morphable architectures

Relative to fixed-function hardware, R-Stream and a morphable architecture yields:

- Cost savings
- Shorter time to deployment
- Multipurpose hardware
- More advanced algorithms
- Dynamic mission changes
- In-mission adaptation
- Multi-sensor
- Bug fixes and upgrades less costly

Streaming Compilation

- Streaming is a pattern in efficient implementations of computation- and data-intensive applications
  - Three key characteristics:
    - Processing is largely parallel
    - Data access patterns are apparent
    - Control is high-level, steady, simple
  - Caused by intersection of application domain and architecture limits:

<table>
<thead>
<tr>
<th>Applications process, simulate, or render physical systems</th>
<th>Architectural limits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processing is largely parallel</td>
<td>VLIW, SIMD, multiprocessor demand for parallelism</td>
</tr>
<tr>
<td>Data access patterns are apparent</td>
<td>Small local memories</td>
</tr>
<tr>
<td>Control is high-level, steady, simple</td>
<td>Handling unpredictability in hardware is expensive</td>
</tr>
</tbody>
</table>

- Streaming languages and architectures designed for this pattern are emerging:
  - Streaming languages enforce the pattern, making parallelism and data-flow apparent
  - Streaming architectures provide maximum parallel resources and minimize control overhead by exposing resources to the compiler
- R-Stream uses these to expand the scope of optimization and resource choreography
  - Streaming languages avoid limits on compiler analysis (e.g. aliasing), enabling top-down optimizations
  - Streaming architectures allow the compiler to lay out all computation, data, and communication

Implementation Plan

- Spiral development
  - Helps to ensure eventual results
  - Enables early usage by partners
- Three major releases:
  - Release 1.0: Functionality, but no performance optimization
  - Release 2.0: Common-case, phase-ordered performance optimization, static morphing
  - Release 3.0: General, unified performance optimization, dynamic morphing

<table>
<thead>
<tr>
<th>Year</th>
<th>Q1</th>
<th>Q2</th>
<th>Q3</th>
<th>Q4</th>
<th>Q5</th>
<th>Q6</th>
<th>Q7</th>
<th>Q8</th>
<th>Q9</th>
<th>Q10</th>
<th>Q11</th>
<th>Q12</th>
<th>Q13</th>
<th>Draft</th>
<th>Final</th>
</tr>
</thead>
<tbody>
<tr>
<td>2022</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2023</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2024</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2025</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Kernels perform computations on streams. This kernel computes pair-wise sum.

Use of streamAdd kernel to double stream.

Use of stream operator to read from array into stream.

Stream is 1D, but elements can be arrays. This is a stream of 3x3 arrays.

Represents $s3.push(s1.pop() + s2.pop())$.

Compiler Flow

Front end

Mapper

1. Unfold graph
2. Cluster based on edge/node weights
3. Convert communication
4. Schedule and assign storage

Doors Open

PCB Machine Model for Target Architecture

(simple example used in algorithm illustration)

Kernels mapped to processors

Streaming Virtual Machine Code

Streams mapped to memory

Whole application size-limited

Kernels mapped to processors

Low-level compiler for target architecture

Binary Executable
Streaming Intermediate Representation (IR)

- Represents an application within the compiler as a series of kernels or loop nests connected by streams or arrays.
- Enables the compiler to directly analyze and optimize streaming applications.
- Makes task, pipeline, and data parallelism readily apparent.

Top-down, unified, streaming IR mapping

- Mapping will include top-down optimizations, such as software pipelining an entire application to cover communication latency.
- Mapping of computation, data, and communication layout will be performed by a single, unified pass for greater efficiency.
- Mapping will exploit all degrees of parallelism exposed by the streaming IR.

Compiler driven architecture morphing

- Morphable architectures reconfigure high-level resources such as processors, memories, and networks to meet application demands.
- Compiler must choose a configuration and map to that configuration – this expands the compiler mapping space.
- Compiler can employ one or multiple configurations:
  1. Static Morphing pick one configuration and mapping for whole application
  2. Dynamic Morphing pick multiple configurations and mappings for different parts of the application; orchestrate morphing between them.