Stream Compilers

Saman Amarasinghe and William Thies
Massachusetts Institute of Technology

PACT 2003
September 27, 2003
## Schedule

<table>
<thead>
<tr>
<th>Time</th>
<th>Session</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:30-1:40</td>
<td>Overview (Saman)</td>
</tr>
<tr>
<td>1:40-2:20</td>
<td>Stream Architectures (Saman)</td>
</tr>
<tr>
<td>2:20-3:00</td>
<td>Stream Languages (Bill)</td>
</tr>
<tr>
<td>3:00-3:30</td>
<td>Break</td>
</tr>
<tr>
<td>3:30-3:55</td>
<td>Stream Compilers (Saman)</td>
</tr>
<tr>
<td>3:55-4:35</td>
<td>Scheduling Algorithms (Bill)</td>
</tr>
<tr>
<td>4:35-5:00</td>
<td>Domain-specific Optimizations (Saman)</td>
</tr>
</tbody>
</table>
How to execute a Stream Graph?
Method 1: Time Multiplexing

- Run one filter at a time

- Pros:
  - Perfectly load balanced
  - Allows SIMD control
  - Synchronization from Memory

- Cons:
  - If a filter run is too short
    - Filter load overhead is high
  - If a filter run is too long
    - Data spills down the cache hierarchy
    - Long latency
  - Lots of memory traffic
    - Bad cache effects
    - Could require storage to offset (SRF)
  - Does not scale with spatially-aware architectures
How to execute a Stream Graph?  
Method 2: Space Multiplexing

- Map filter per tile and run forever

- Pros:
  - No filter swapping overhead
  - Exploits spatially-aware architectures
    - Scales well
  - Reduced memory traffic
  - Localized communication
  - Tighter latencies
  - Smaller live data set

- Cons:
  - Load balancing is critical
  - Not good for dynamic behavior
  - Requires \# filters \leq \# processing elements
Example: Radar Array Front End

```c
void pipeline BeamFormer(int numChannels, int numBeams) {
    add splitjoin {
        split duplicate;
        for (int i=0; i<numChannels; i++) {
            add pipeline {
                add FIR1(N1);
                add FIR2(N2);
            };
        }
    };
    join roundrobin;
}

add splitjoin {
    split duplicate;
    for (int i=0; i<numBeams; i++) {
        add pipeline {
            add VectorMult();
            add FIR3(N3);
            add Magnitude();
            add Detect();
        };
    }
    join roundrobin(0);
}
```
Radar Array Front End on Raw

- Blocked on Static Network
- Executing Instructions
- Pipeline Stall
Bridging the Abstraction layers

- StreamIt language exposes the data movement
  - Graph structure is architecture independent
- Each architecture is different in granularity and topology
  - Communication is exposed to the compiler
- The compiler needs to efficiently bridge the abstraction
  - Map the computation and communication pattern of the program to the PE’s, memory and the communication substrate
Bridging the Abstraction layers

- StreamIt language exposes the data movement
  - Graph structure is architecture independent
- Each architecture is different in granularity and topology
  - Communication is exposed to the compiler
- The compiler needs to efficiently bridge the abstraction
  - Map the computation and communication pattern of the program to the PE’s, memory and the communication substrate
- The StreamIt Compiler
  - Partitioning
  - Placement
  - Scheduling
  - Code generation
Partitioning: Choosing the Granularity

- Mapping filters to tiles
  - # filters should equal (or a few less than) # of tiles
  - Each filter should have similar amount of work
    - Throughput determined by the filter with most work

- Compiler Algorithm
  - Two primary transformations
    - Filter fission
    - Filter fusion
  - Uses a greedy heuristic
Partitioning - Fission

- Fission - splitting streams
  - Duplicate a filter, placing the duplicates in a SplitJoin to expose parallelism.

-Split a filter into a pipeline for load balancing
Partitioning - Fusion

- Fusion - merging streams
  - Merge filters into one filter for load balancing and synchronization removal

![Diagram showing the process of partitioning and fusion with nodes labeled as Splitter, Joiner, Filter0, Filter1, ..., FilterN, and Filter.]
Example: Radar Array Front End (Original)
Example: Radar Array Front End
Example: Radar Array Front End
Example: Radar Array Front End
Example: Radar Array Front End
Example: Radar Array Front End
Example: Radar Array Front End

Diagram showing the flow of processes from a splitter to a joiner, with intermediate nodes for filtering and detection.
Example: Radar Array Front End
Example: Radar Array Front End
Example: Radar Array Front End (Balanced)
Placement: Minimizing Communication

- Assign filters to tiles
  - Communicating filters $\rightarrow$ try to make them adjacent
  - Reduce overlapping communication paths
  - Reduce/eliminate cyclic communication if possible

- Compiler algorithm
  - Uses Simulated Annealing
Placement for Partitioned Radar Array Front End
Scheduling: Communication Orchestration

- Create a communication schedule

- Compiler Algorithm
  - Calculate an initialization and steady-state schedule
  - Simulate the execution of an entire cyclic schedule
  - Place static route instructions at the appropriate time
Steady-State Schedule

- All data pop/push rates are constant
- Can find a Steady-State Schedule
  - # of items in the buffers are the same before and the after executing the schedule
  - There exist a unique minimum steady state schedule
  - *More details later, in section on scheduling*

- Schedule = {}
Steady-State Schedule

- All data pop/push rates are constant
- Can find a Steady-State Schedule
  - # of items in the buffers are the same before and the after executing the schedule
  - There exist a unique minimum steady state schedule
  - More details later, in section on scheduling

- Schedule = \{ A \}
Steady-State Schedule

- All data pop/push rates are constant
- Can find a Steady-State Schedule
  - # of items in the buffers are the same before and after executing the schedule
  - There exist a unique minimum steady state schedule
  - More details later, in section on scheduling

- Schedule = \{ A, A \}
Steady-State Schedule

- All data pop/push rates are constant
- Can find a Steady-State Schedule
  - # of items in the buffers are the same before and the after executing the schedule
  - There exist a unique minimum steady state schedule
  - *More details later, in section on scheduling*

- Schedule = \{ A, A, B \}
Steady-State Schedule

- All data pop/push rates are constant
- Can find a Steady-State Schedule
  - # of items in the buffers are the same before and the after executing the schedule
  - There exist a unique minimum steady state schedule
  - *More details later, in section on scheduling*

- Schedule = \{ A, A, B, A \}
Steady-State Schedule

- All data pop/push rates are constant
- Can find a Steady-State Schedule
  - # of items in the buffers are the same before and the after executing the schedule
  - There exist a unique minimum steady state schedule
  - *More details later, in section on scheduling*

- Schedule = \{ A, A, B, A, B \}
Steady-State Schedule

- All data pop/push rates are constant
- Can find a Steady-State Schedule
  - # of items in the buffers are the same before and the after executing the schedule
  - There exist a unique minimum steady state schedule
  - More details later, in section on scheduling

- Schedule = \{ A, A, B, A, B, C \}
Initialization Schedule

- When peek > pop, buffer cannot be empty after firing a filter
- Buffers are not empty at the beginning/end of the steady state schedule
- Need to fill the buffers before starting the steady state execution
- More details later, in section on scheduling
Initialization Schedule

- When peek > pop, buffer cannot be empty after firing a filter
- Buffers are not empty at the beginning/end of the steady state schedule
- Need to fill the buffers before starting the steady state execution
- More details later, in section on scheduling
Code Generation: Optimizing tile performance

- Creates code to run on each tile
  - Optimized by the existing node compiler
- Generates the switch code for the communication
Performance Results for Radar Array Front End

- Blocked on Static Network
- Executing Instructions
- Pipeline Stall
Performance of Radar Array Front End*

* As of Summer, 2002

- **C program**: 240 MFLOPS (1 GHz Pentium III)
- **C program**: 11 MFLOPS (250 MHz single tile Raw)
- **Unoptimized StreamIt**: 577 MFLOPS (250 MHz 64 tile Raw)
- **Optimized StreamIt**: 1,230 MFLOPS (250 MHz 16 tile Raw)
Utilization of Radar Array Front End*

* As of Summer, 2002

<table>
<thead>
<tr>
<th>MFLOPS per Tile</th>
<th>C program</th>
<th>Unoptimized StreamIt</th>
<th>Optimized StreamIt</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>250 MHz single tile Raw</td>
<td>250 MHz 64 tile Raw</td>
<td>250 MHz 16 tile Raw</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>10</td>
<td>99</td>
</tr>
</tbody>
</table>
Application Performance*

* As of Summer, 2002

Throughput of StreamIt normalized to single tile C

FIR  FFT  Radio  3GPP  Sort  Radar  Filterbank

Architectures, Languages, and Compilers for the Streaming Domain
PACT 2003 Tutorial - Saman Amarasinghe, William Thies - MIT CSAIL
Scalability of StreamIt*

* As of Summer, 2002

![Bar Chart]

- Bitonic Sort

- Normalized Throughput

- 1 x 1
- 2 x 2
- 3 x 3
- 4 x 4
- 5 x 5
- 6 x 6
Scalability of StreamIt*

* As of Summer, 2002

Bitonic Sort
Related Work

- Stream-C / Kernel-C (Dally et. al)
  - Compiled to Imagine with time multiplexing
  - Extensions to C to deal with finite streams
  - Programmer explicitly calls stream “kernels”
  - Need program analysis to overlap streams / vary target granularity

- Brook (Buck et. al)
  - Architecture-independent counterpart of Stream-C / Kernel-C
  - Designed to be more parallelizable

- Ptolemy (Lee et. al)
  - Heterogeneous modeling environment for DSP
  - Many scheduling results shared with StreamIt
  - Don’t focus on language development / optimized code generation

- Other languages
  - Occam, SISAL – not statically schedulable
  - LUSTRE, Lucid, Signal, Esterel – don’t focus on parallel performance
Conclusion

- Streaming Programming Model
  - An important class of applications
  - Can break the von Neumann bottleneck
  - A natural fit for a large class of applications
  - Straightforward mapping to the architectural model

- StreamIt: A Machine Language for Communication Exposed Architectures
  - Expose the common properties
    - Multiple instruction streams
    - Software exposed communication
    - Fast local memory co-located with execution units
  - Hide the differences
    - Granularity of execution units
    - Type and topology of the communication network
    - Memory hierarchy

- A good compiler can eliminate the overhead of abstraction