Stream Languages and Programming Models

Saman Amarasinghe and William Thies
Massachusetts Institute of Technology

PACT 2003
September 27, 2003
<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>3:55-4:20</td>
<td>Domain-specific Optimizations (Saman)</td>
</tr>
<tr>
<td>4:20-5:00</td>
<td>Scheduling Algorithms (Bill)</td>
</tr>
</tbody>
</table>
Timeline: 1960’s

- **Models of Computation**
  - Petri Nets
  - Comp Graphs

- **Languages / Compilers**

- **Modeling Environments**

- “Stream” (P.J. Landin) – 1960
  - Linking Algol 60 and lambda calculus, used for loop histories

- Petri Nets (C.A. Petri) – 1966
  - Places, transitions, tokens

- Computation Graphs (Karp, Miller) – 1967
  - Graph with firing actors, minimal firing requirements
  - Formulate determinancy, termination, queuing properties
Timeline: 1970’s

Models of Computation
- Petri Nets
- Comp Graphs
- PN
- CSP

Languages / Compilers
- Lucid
- Id
- lazy VAL

Modeling Environments

- Process Networks (Kahn) – 1974
  - Sequential threads communicate with unbounded FIFO’s
  - Deterministic
- CSP: Communicating Sequential Processes (Hoare) – 1978
  - Sequential threads communicate with rendezvous message-passing
  - Non-deterministic due to guards
- Dataflow languages
  - First version dataflow procedure language (Dennis)
  - Lucid (Ashcroft, Wadge), Id (Arvind, Gostelow), VAL (Dennis)
- Functional languages with lazy evaluation for streams
  - lazy evaluator (Henderson, Morris); Sieve of Eratosthenes (Friedman, Wise)
Timeline: 1980’s

- SDF: Synchronous Dataflow (Lee, Messerschmitt) – 1987
  - Actors have static, non-uniform rates; firing is atomic and data-driven
  - Allows static scheduling
- Sisal: Streams and Iteration in a Single Assignment Language – 1983
  - Adds recursion, finite streams to VAL
  - Implementations on many parallel machines
  - IF1 intermediate format
- Occam – 1983
  - Strongly typed procedural language
  - Practical implementation of CSP
- More work on dataflow and functional languages (e.g., M. Broy)
Timeline: 1990’s

Models of Computation
- Petri Nets
- Comp Graphs
- PN
- CSP
- SDF
- FSM

Languages / Compilers
- Lucid
- Id
- Sisal
- Signal
- Esterel
- C
- lazy
- VAL
- Occam
- LUSTRE
- pH

Modeling Environments
- Ptolemy
- Matlab/Simulink
- Gabriel
- Grape-II
- etc.

- Synchronous Languages: Signal, LUSTRE, etc.
  - Designed for expressiveness, verification moreso than high performance
- Esterel
  - For reactive programming; event-driven and control-oriented
  - Often implemented in either hardware or software
- pH: Parallel Haskell (Nikhil, Arvind, et al.)
  - Combines lazy functional and dataflow philosophies for high performance
- Ptolemy: Heterogeneous Modeling Environment (Lee et al.)
  - Many contributions to formalisms, scheduling, graph-level optimization
- Commercial Environments (Matlab, SPW, COSSAP, ADS, etc.)
  - Becoming increasingly prevalent
Stream Programming Models

- Prototyping environments
- Conventional languages
  - Object Oriented
  - Procedural
  - Assembly
- Stream languages
  - StreamIt
  - Brook
  - Cg
Actor-Oriented Design in the Ptolemy Project (UC Berkeley)

Model of Computation:
- Messaging schema
- Flow of control
- Concurrency

Examples:
- Dataflow
- Process networks
- Synchronous
- Time triggered
- Discrete-event systems
- Publish & subscribe

Most Ptolemy II models of computation are “actor oriented.” But the precise semantics depends on the selected “director,” which implements a model of computation.

called a “kernel,” “step,” ...
Focus on Dataflow (a few variants)

- Computation graphs [Karp & Miller - 1966]
- Process networks [Kahn - 1974]
- Static dataflow [Dennis - 1974]
- Dynamic dataflow [Arvind, 1981]
- K-bounded loops [Culler, 1986]
- Synchronous dataflow [Lee & Messerschmitt, 1986]
- Structured dataflow [Kodosky, 1986]
- PGM: Processing Graph Method [Kaplan, 1987]
- Synchronous languages [Lustre, Signal, 1980’s]
- Well-behaved dataflow [Gao, 1992]
- Boolean dataflow [Buck and Lee, 1993]
- Multidimensional SDF [Lee, 1993]
- Cyclo-static dataflow [Lauwereins, 1994]
- Integer dataflow [Buck, 1994]
- Bounded dynamic dataflow [Lee and Parks, 1995]
- ...

Many tools, software frameworks, and hardware architectures have been built to support one or more of these.
Synchronous Dataflow (SDF)
Fixed Production/Consumption Rates

- Schedulable statically
- Decidable:
  - buffer memory requirements
  - deadlock

Will address in detail in section on scheduling
Selected Generalizations

- **static**
  - Multidimensional Synchronous Dataflow (1993)
    - Arcs carry multidimensional streams
    - One balance equation per dimension per arc
  - Cyclo-Static Dataflow (Lauwereins, et al., 1994)
    - Periodically varying production/consumption rates
  - Heterochronous Dataflow (1997)
    - Combines state machines with SDF graphs
    - Very expressive, yet decidable
  - Boolean & Integer Dataflow (1993/4)
    - Balance equations are solved symbolically
    - Permits data-dependent routing of tokens
    - Heuristic-based scheduling (undecidable)

- **dynamic**
  - Dynamic Dataflow (1981-)
    - Firings scheduled at run time
    - Challenge: maintain bounded memory, deadlock freedom, liveness
    - Demand driven, data driven, and fair policies all fail
  - Kahn Process Networks (1974-)
    - Replace discrete firings with process suspension
    - Challenge: maintain bounded memory, deadlock freedom, liveness
Other Stream-Like Models of Computation
(all implemented in Ptolemy II)

- Push/Pull
  - dataflow with disciplined nondeterminism
  - e.g. Click (Kohler, 2001)

- Discrete events
  - data tokens have time stamps
  - e.g. NS

- Continuous time
  - streams are a continuum of values
  - e.g. Simulink

- Synchronous languages
  - sequence of values, one per clock tick
  - fixed-point semantics
  - e.g. Esterel

- Time triggered
  - similar, but no fixed-point semantics
  - e.g. Giotto

- Modal models
  - state machines + stream-like MoCs, hierarchical
  - e.g. Hybrid systems

All of these include a logical notion of time.
Software Legacy of the Ptolemy Project

- Gabriel (1986-1991)
  - Written in Lisp
  - Aimed at signal processing
  - Synchronous dataflow (SDF) block diagrams
  - Parallel schedulers
  - Code generators for DSPs
  - Hardware/software co-simulators

- Ptolemy Classic (1990-1997)
  - Written in C++
  - Multiple models of computation
  - Hierarchical heterogeneity
  - Dataflow variants: BDF, DDF, PN
  - C/VHDL/DSP code generators
  - Optimizing SDF schedulers
  - Higher-order components

- Ptolemy II (1996-2022)
  - Written in Java
  - Domain polymorphism
  - Multithreaded
  - Network integrated and distributed
  - Modal models
  - Sophisticated type system
  - CT, HDF, CI, GR, etc.

Each of these served, first-and-foremost, as a laboratory for investigating design.

- PtPlot (1997-??)
  - Java plotting package

  - Itcl/Tk GUI framework

- Diva (1998-2000)
  - Java GUI framework

Focus has always been on embedded software.
Ptolemy II:

Our current framework for experimentation with actor-oriented design, concurrent semantics, visual syntaxes, and hierarchical, heterogeneous design.

http://ptolemy.eecs.berkeley.edu
Implementing High-Performance Streaming Applications

- Modeling environments are good for prototyping, algorithmic optimizations
- However, embedded systems have tight resource constraints:
  - Real-time requirements (throughput, latency)
  - Limited battery life (power)
  - Limited instruction and data memory
- Current practice: re-implement stream algorithm in high-performance language
  - C / assembly
  - C++ runtime system (e.g., Spectrumware)
- New class of “stream languages” aim to raise abstraction level, provide unified development environment
  - StreamIt
  - Brook
  - Cg
Stream Programming Models

- Prototyping environments
- Conventional languages
  - Object Oriented
  - Procedural
  - Assembly
- Stream languages
  - StreamIt
  - Brook
  - Cg
Streaming in Object Oriented Style

- Each actor is an object
- Scheduled by pull model

Control (Runtime System)
Streaming in Object Oriented Style

- Each actor is an object
- Scheduled by pull model

```java
class FIRFilter extends Stream {
    int N;
    float[] input;
    void getData(float[] output,
                 int offset, int length) {
        if (input==null) {
            input = new float[MAX_LENGTH];
            source.getData(input, 0, N+length);
        } else
            source.getData(input, N, length);
        for (int i=0; i<length; i++) {
            float sum = 0;
            for (int j=0; j<N; j++)
                sum = sum + data1[i+j]*h[j][N];
            output[i+offset] = sum;
        }
        input[i] = input[i+length];
    }
}
```
Streaming in Object Oriented Style

- Each actor is an object
- Scheduled by pull model

```java
void main() {
    DataSource datasource = new DataSource();
    FIRFilter filter = new FIRFilter(5);
    Display display = new Display();
    filter.source = datasource;
    display.source = filter;
    display.run();
}
```
Streaming in Object Oriented Style

Pro:
- Modular
- Shows structure of graph
- Automatic scheduling

Con:
- Overhead of objects
  - Communication is static; don’t need virtual dispatch
- Coarse-grained communication
  - Block size is architecture-dependent
  - Obscures fine-grained algorithm
- Overhead of run-time scheduler
  - Lots of method calls
  - Impossible to keep persistent data in registers
- Compiler can’t optimize across module boundaries
Stream Programming Models

- Prototyping environments
- Conventional languages
  - Object Oriented
  - Procedural
  - Assembly
- Stream languages
  - StreamIt
  - Brook
  - Cg
Streaming in Procedural Style

```c
int N = 5;
int BLOCK_SIZE = 100;

void main() {
    float input[] = new float[N];
    float output[] = new float[BLOCK_SIZE];
    int i, j;
    for (i=0; i<N; i++)
        input[i] = getData();
    while (true) {
        for (out=0; i<N; i++, j++)
            step(input, output, i, j);
        int wholeSteps = (BLOCK_SIZE-j)/N;
        for (int k=0; k<wholeSteps; k++)
            for (i=0; i<N; i++, j++)
                step(input, output, i, j);
        displayBlock(output);
    }
}
```

- Complicated loop nest
  - Statements in loops represent actors
  - Circular buffers for data items
  - Scheduling done by hand
  - Loop bounds adjusted for cache size

```c
void step(float[] input, float[] output, int i, int j)
{
    float sum = 0;
    for (int k=0; k<i; k++)
        sum = sum + input[k]*h[k+i][N];
    for (int k=i; k<N; k++)
        sum = sum + input[k]*h[k-i][N];
    output[j] = sum;
    input[i] = getData();
}
```
Streaming in Procedural Style

```c
int N = 5;
int BLOCK_SIZE = 100;

void main() {
    float input[] = new float[N];
    float output[] = new float[BLOCK_SIZE];
    int i, j;
    for (i=0; i<N; i++)
        input[i] = getData();
    while (true) {
        for (out=0; i<N; i++, j++)
            step(input, output, i, j);
        int wholeSteps = (BLOCK_SIZE-j)/N;
        for (int k=0; k<wholeSteps; k++)
            for (i=0; i<N; i++, j++)
                step(input, output, i, j);
        for (i=0; j<BLOCK_SIZE; i++, j++)
            step(input, output, i, j);
        displayBlock(output);
    }
}
```

- **Pro:**
  - Better performance than object-oriented style

- **Con:**
  - Obscures parallelism and communication patterns
  - Scheduling and buffer management done by hand
    - Difficult to get it right
    - Hard to maintain
    - Impossible for compiler to optimize for given resources
  - No modularity
    - Actors are mixed with global variables and control flow
  - Hard to visualize computation
Stream Programming Models

- Prototyping environments
- Conventional languages
  - Object Oriented
  - Procedural
  - Assembly
- Stream languages
  - StreamIt
  - Brook
  - Cg
Streaming in Assembly Code

- Example: Freq band detection

- Used in...
  - metal detector
  - garage door opener
  - spectrum analyzer

Source:
Application Report SPRA414
Texas Instruments, 1999
**Event Manager Module Reset**

```
EVENT MANAGER - starts here

- This section resets all the Event Manager Module registers.
- This is necessary for all silicon revision 1.1; however, for silicon revisions 2.0 and later, this is not necessary.

- Event Manager Module Reset
```

```
EVENT MANAGER - starts here

- This section resets all the Event Manager Module registers.
- This is necessary for all silicon revision 1.1; however, for silicon revisions 2.0 and later, this is not necessary.

- Event Manager Module Reset
```

```
EVENT MANAGER - starts here

- This section resets all the Event Manager Module registers.
- This is necessary for all silicon revision 1.1; however, for silicon revisions 2.0 and later, this is not necessary.

- Event Manager Module Reset
```

```
EVENT MANAGER - starts here

- This section resets all the Event Manager Module registers.
- This is necessary for all silicon revision 1.1; however, for silicon revisions 2.0 and later, this is not necessary.

- Event Manager Module Reset
```
The document is a programming guide or manual for a specific electronic device or system, likely related to a microcontroller or similar hardware. The text contains detailed descriptions of various registers and their functions, along with bit definitions and usage examples. The language used is technical and specific to the field of electronics and computer science, indicating that the document is intended for engineers or developers working with the hardware described. The content is dense and requires a good understanding of digital electronics and programming for microcontrollers.
Source: Application Report SPR414, Texas Instruments, 1999

Architectures, Languages, and Compilers for the Streaming Domain
PACT 2003 Tutorial - Saman Amarasinghe, William Thies - MIT CSAIL
; Output in below THRESHOLD1 & THRESHOLD2, & THRESHOLD3. Turn off LEDS
; BELOW SPK #0010h, LEADC
; OUTPUT
; Output in below THRESHOLD1, but above THRESHOLD2. Turn off LEDS
; BELOW SPK #0011h, LEADC
; OUTPUT
; Output in below THRESHOLD1, but above THRESHOLD2. Check if above
; THRESHOLD1.
; ABOVE 1 SPK
; TURN ON D2;
; OUTPUT
; Output in below THRESHOLD1, but above THRESHOLD2 & THRESHOLD3. Check if above
; THRESHOLD1.
; ABOVE 2 SPK
; TURN ON D3 - D5;
; OUTPUT
; Output in below THRESHOLD1, but above THRESHOLD2 & THRESHOLD3 & THRESHOLD4. Check if above
; THRESHOLD1.
; ABOVE 3 SPK
; TURN ON D6 - D8;
; OUTPUT
; Output in above THRESHOLD1 & THRESHOLD2, but below THRESHOLD3 &
; THRESHOLD4. Turn on LEDS
; ABOVE SPK #0002h, LEADC
; OUTPUT
; Output in above THRESHOLD1 & THRESHOLD2, but below
; THRESHOLD3. Turn on LEDS
; ABOVE SPK #0003h, LEADC
; OUTPUT
; Output in above THRESHOLD1 & THRESHOLD2, but below
; THRESHOLD3 & THRESHOLD4. Check if above
; THRESHOLD3.
; ABOVE 4 SPK
; TURN ON D1 - D3;
; OUTPUT
; Output in above THRESHOLD1 & THRESHOLD2, but below
; THRESHOLD3 & THRESHOLD4 & THRESHOLD5. Turn on LEDS
; ABOVE SPK #0007h, LEADC
; OUTPUT
; Output in above THRESHOLD1 & THRESHOLD2, but below
; THRESHOLD3 & THRESHOLD4 & THRESHOLD5 & THRESHOLD6. Check if above
; THRESHOLD5.
; ABOVE 5 SPK
; TURN ON D4 - D6;
; OUTPUT
; Output in above THRESHOLD1 & THRESHOLD2, but below
; THRESHOLD3 & THRESHOLD4 & THRESHOLD5 & THRESHOLD6 & THRESHOLD7. Turn on LEDS
; ABOVE SPK #000Fh, LEADC
; OUTPUT
; Output in above THRESHOLD1 & THRESHOLD2, but below
; THRESHOLD3 & THRESHOLD4 & THRESHOLD5 & THRESHOLD6 & THRESHOLD7 & THRESHOLD8. Check if above
; THRESHOLD7.
; ABOVE 6 SPK
; TURN ON D7 - D9;
; OUTPUT
; Output in above THRESHOLD1 & THRESHOLD2, but below
; THRESHOLD3 & THRESHOLD4 & THRESHOLD5 & THRESHOLD6 & THRESHOLD7 & THRESHOLD8 & THRESHOLD9. Turn on LEDS
; ABOVE SPK #001Fh, LEADC
; OUTPUT
; Output in above THRESHOLD1 & THRESHOLD2, but below
; THRESHOLD3 & THRESHOLD4 & THRESHOLD5 & THRESHOLD6 & THRESHOLD7 & THRESHOLD8 & THRESHOLD9 & THRESHOLD10. Check if above
; THRESHOLD9.
; ABOVE 7 SPK
; TURN ON D10 - D12;
; OUTPUT
; Output in above THRESHOLD1 & THRESHOLD2, but below
; THRESHOLD3 & THRESHOLD4 & THRESHOLD5 & THRESHOLD6 & THRESHOLD7 & THRESHOLD8 & THRESHOLD9 & THRESHOLD10 & THRESHOLD11. Turn on LEDS
; ABOVE SPK #003Fh, LEADC
; OUTPUT
; Output in above THRESHOLD1 & THRESHOLD2, but below
; THRESHOLD3 & THRESHOLD4 & THRESHOLD5 & THRESHOLD6 & THRESHOLD7 & THRESHOLD8 & THRESHOLD9 & THRESHOLD10 & THRESHOLD11 & THRESHOLD12. Check if above
; THRESHOLD11.
; ABOVE 8 SPK
; TURN ON D13 - D15;
; OUTPUT
; Output in above THRESHOLD1 & THRESHOLD2, but below
; THRESHOLD3 & THRESHOLD4 & THRESHOLD5 & THRESHOLD6 & THRESHOLD7 & THRESHOLD8 & THRESHOLD9 & THRESHOLD10 & THRESHOLD11 & THRESHOLD12 & THRESHOLD13. Turn on LEDS
; ABOVE SPK #00FFh, LEADC

Source: Application Report SPRA414, Texas Instruments, 1999

Architectures, Languages, and Compilers for the Streaming Domain
PACT 2003 Tutorial - Saman Amarasinghe, William Thies - MIT CSAIL
DSP Implementation (Excerpt)

;The following section determines if the value meets the threshold
;requirement
LDP #0 ;DP = 0; Addresses 0000h to 007fh
;All variables used are in B2
;Need to remove the DC offset because if the FIR result is 0 it will
;equal 7ffh which is already 50% of the maximum input value
LACC MAXIN ;ACC = MAXIN
SUB #7FFh ;Subtract the DC offset
SACL DIFFIN ;DIFFIN = MAXIN - 7ffh
LACC MAXOUT ;ACC = MACOUT
SUB #7FFh ;Subtract the DC offset
SACL DIFFOUT ;DIFFOUT = MAXOUT - 7ffh
;Check if the output exceeds the middle threshold value, THRESHOLD4
LT DIFFIN ;TREG = DIFFIN
TH4 MPY THRESHOLD4 ;PREG = DIFFIN * THRESHOLD4
PAC ;ACC = PREG
SACH TEMP,1 ;TEMP = ACC*2; Shift to remove
;extra sign bit
LACC TEMP ;ACC = TEMP
SUB DIFFOUT ;Subtract DIFFOUT
BCND ABOVE4,LT ;If DIFFOUT is greater than
;TEMP, then the FIR result is
;greater than VALUEIN * THRESHOLD4,
;else, it is below THRESHOLD4 value
;Output is below THRESHOLD4. Check if above THRESHOLD2
BELOW4 LT DIFFIN
TH2 MPY THRESHOLD2
PAC
...

Source: Application Report SPRA414, Texas Instruments, 1999
Streaming in Assembly Code

Pro: Fast!

Con:
- Extremely tedious, costly, and error-prone
- Not portable between architectures
- Very hard to maintain
  - Move center frequency from 500 Hz to 1200 Hz?
  - According to TI, in the conventional design flow:
    - Redesign filter in MATLAB
    - Cut-and-paste values to EXCEL
    - Recalculate the coefficients
    - Update the assembly code
- Will address this issue again later today, in section on Domain Specific Optimizations

Architectures, Languages, and Compilers for the Streaming Domain
PACT 2003 Tutorial - Saman Amarasinghe, William Thies - MIT CSAIL
Stream Languages to the Rescue

- Goals of a stream language:
  - Expose parallelism
  - Expose communication patterns
  - Encapsulate common idioms
    - Autonomous filters
    - Circular buffers

  Improve BOTH performance and programmer productivity

- Vision:
  A unified, high-level programming environment that achieves the performance of hand-coded assembly
Stream Programming Models

- Prototyping environments
- Conventional languages
  - Object Oriented
  - Procedural
  - Assembly
- Stream languages
  - StreamIt
  - Brook
  - Cg
The StreamIt Language

- A high-level, architecture-independent language for streaming applications
- Current focus domain: Synchronous Dataflow
- Contributions
  - Language Design, Structured Streams, Buffer Management (*CC 2002*)
  - Exploiting Wire-Exposed Architectures (*ASPLOS 2002*)
  - Scheduling of Static Dataflow Graphs (*LCTES 2003*)
  - Domain Specific Optimizations (*PLDI 2003*)
Representing Streams

- Conventional wisdom: streams are graphs
  - Graphs have no simple textual representation
  - Graphs are difficult to analyze and optimize
- Insight: stream programs have structure

unstructured

structured
Structured Streams

- Hierarchical structures:
  - Pipeline
  - SplitJoin
  - Feedback Loop

- Basic programmable unit: Filter
void->void pipeline FrequencyBand {
    float sFreq = 4000;
    float cFreq = 500/(sFreq*2*pi);
    float wFreq = 100/(sFreq*2*pi);

    add D2ASource(sFreq);

    add BandPassFilter(1, cFreq-wFreq,
                       cFreq+wFreq, 100);

    add splitjoin {
        split duplicate;
        for (int i=0; i<4; i++) {
            add pipeline {
                add Detector(i/4);
                add LEDOutput(i);
            }
        }
    }

    join roundrobin(0);
}
void->void pipeline FrequencyBand {
  float sFreq = 4000;
  float cFreq = 500/(sFreq*2*pi);
  float wFreq = 100/(sFreq*2*pi);

  add D2ASource(sFreq);

  float->float pipeline BandPassFilter(float gain, float ws, float wp, int num) {
    add LowPassFilter(1, wp, num);
    add HighPassFilter(gain, ws, num);
  }

  add splitjoin {
    split duplicate;
    for (int i=0; i<4; i++) {
      add pipeline {
        add Detector(i/4);
        add LEDOutput(i);
      }
    }
    join roundrobin(0);
  }
}

float -> float pipeline BandPassFilter(float gain, float ws, float wp, int num) {
  add LowPassFilter(1, wp, num);
  add HighPassFilter(gain, ws, num);
}
float->float **filter** LowPassFilter (int N, float freq) {
    float[N] weights;

    **init** {
        weights = calcWeights(N, freq);
    }

    **work** **push** 1 **pop** 1 **peek** N {
        float result = 0;
        for (int i=0; i<weights.length; i++) {
            result += weights[i] * **peek**(i);
        }
    }
    **push**(result);
    **pop**();
}

Filter Example: LowPassFilter

```c
float->float filter LowPassFilter (int N, float freq) {
    float[N] weights;

    init {
        weights = calcWeights(N, freq);
    }

    work push 1 pop 1 peek N {
        float result = 0;
        for (int i=0; i<weights.length; i++) {
            result += weights[i] * peek(i);
        }
        push(result);
        pop();
    }
}
```
Filter Example: LowPassFilter

```c
float->float filter LowPassFilter (int N, float freq) {
  float[N] weights;

  init {
    weights = calcWeights(N, freq);
  }

  work push 1 pop 1 peek N {
    float result = 0;
    for (int i=0; i<weights.length; i++) {
      result += weights[i] * peek(i);
    }
    push(result);
    pop();
  }
}
```
Filter Example: LowPassFilter

```c
float->float filter LowPassFilter (int N, float freq) {
    float[N] weights;

    init {
        weights = calcWeights(N, freq);
    }  

    work push 1 pop 1 peek N {
        float result = 0;
        for (int i=0; i<weights.length; i++) {
            result += weights[i] * peek(i);
        }

        push(result);
        pop();
    }
}
```
Filter Example:  LowPassFilter

```c
float->float filter LowPassFilter (int N, float freq) {
    float[N] weights;

    init {
        weights = calcWeights(N, freq);
    }

    work push 1 pop 1 peek N {
        float result = 0;
        for (int i=0; i<weights.length; i++) {
            result += weights[i] * peek(i);
        }
        push(result);
        pop();
    }
}
```
Why Structured Streams?

- Compare to structured control flow

GOTO statements

If / else / for statements

- Tradeoff:
  - **PRO:** - more robust - more analyzable
  - **CON:** - “restricted” style of programming
Structure Helps Programmers

- Modules are hierarchical and composable
  - Each structure is single-input, single-output

- Encapsulates common idioms
- Good textual representation
  - Enables parameterizable graphs
N-Element Merge Sort (3-level)
N-Element Merge Sort (K-level)

pipeline MergeSort (int N, int K) {
    if (K == 1) {
        add Sort(N);
    } else {
        add splitjoin {
            split roundrobin;
            add MergeSort(N/2, K-1);
            add MergeSort(N/2, K-1);
            joiner roundrobin;
        }
    }
    add Merge(N);
}

Architectures, Languages, and Compilers for the Streaming Domain
PACT 2003 Tutorial - Saman Amarasinghe, William Thies - MIT CSAIL
Structure Helps Compilers

- Enables local, hierarchical analyses
  - Scheduling
  - Optimization
  - Parallelization
  - Load balancing
Structure Helps Compilers

- Enables local, hierarchical analyses
  - Scheduling
  - Optimization
  - Parallelization
  - Load balancing

Examples:

- Pipeline Fusion
- Pipeline Fission
- SplitJoin Fusion
- SplitJoin Fission
Structure Helps Compilers

- Enables local, hierarchical analyses
  - Scheduling
  - Optimization
  - Parallelization
  - Load balancing

- Examples:

  ![Diagram of structure helps compilers]

  Filter Hoisting
Structure Helps Compilers

- Enables local, hierarchical analyses
  - Scheduling
  - Optimization
  - Parallelization
  - Load balancing

- Disallows non-sensical graphs
- Simplifies separate compilation
  - All blocks single-input, single-output
CON: Restricts Coding Style

- Some graphs need to be re-arranged
- Example: FFT

Bit-reverse order

Butterfly (2 way)

Butterfly (4 way)

Butterfly (8 way)
Example: FM Radio with Equalizer

Low Pass filter

FM Demodulator

Duplicate splitter

Low pass filter  Low pass filter  Low pass filter  Low pass filter  Low pass filter  Low pass filter  Low pass filter  Low pass filter  Low pass filter

Float Diff filter  Float Diff filter  Float Diff filter  Float Diff filter  Float Diff filter  Float Diff filter  Float Diff filter  Float Diff filter  Float Diff filter

Round robin joiner

Float Diff filter

Float Adder filter
Example: Vocoder
Example: GSM decoder
Example: 3GPP Physical Layer
Control Messages

- Structures for regular, high-bandwidth data
- But also need a control mechanism for irregular, low-bandwidth events

- Change volume on a cell phone
- Initiate handoff of stream
- Adjust network protocol
Supporting Control Messages

- Option 1: Embed message in stream
  PRO: - message arrives with data
  CON: - complicates filter code
         - complicates structure
         - runtime overhead

- Option 2: Synchronous method call
  PRO: - delivery transparent to user
  CON: - timing is unclear
         - limits parallelism
StreamIt Messaging System

- Looks like method call, but semantics differ

```java
void raiseVolume(int v)
  myVolume += v;
}
```

- No return value
- Asynchronous delivery
- Can broadcast to multiple targets
StreamIt Messaging System

- Looks like method call, but semantics differ

```java
TargetFilter x;
work {
    ...
    if (lowVolume())
        x.raiseVolume(10) at 100;
}
```

- No return value
- Asynchronous delivery
- Can broadcast to multiple targets

- Timed relative to data
  - User gains precision; compiler gains flexibility
Message Timing

- A sends message to B with zero latency
Message Timing

- A sends message to B with zero latency
Message Timing

- A sends message to B with zero latency
Message Timing

- A sends message to B with zero latency
Message Timing

- A sends message to B with zero latency
Message Timing

- A sends message to B with zero latency
Message Timing

- A sends message to B with zero latency
Message Timing

- A sends message to B with zero latency
Message Timing

- A sends message to B with zero latency
Message Timing

- A sends message to B with zero latency
Message Timing

- A sends message to B with zero latency
Message Timing

- A sends message to B with zero latency
Message Timing

- A sends message to B with zero latency

Distance between wavefronts might have changed
Message Timing

- A sends message to B with zero latency
General Message Timing

- Latency of N means:
  - Message attached to wavefront that \textit{sender} sees in N executions
General Message Timing

- Latency of N means:
  - Message attached to wavefront that sender sees in N executions

- Examples:
  - A → B, latency 1
General Message Timing

- Latency of N means:
  - Message attached to wavefront that sender sees in N executions

- Examples:
  - A → B, latency 1
General Message Timing

- Latency of N means:
  - Message attached to wavefront that *sender* sees in N executions

- Examples:
  - A → B, latency 1
  - B → A, latency 25
General Message Timing

- **Latency of N means:**
  - Message attached to wavefront that *sender* sees in N executions

- **Examples:**
  - A → B, latency 1
  - B → A, latency 25
Rationale

- Better for the programmer
  - Simplicity of method call
  - Precision of embedding in stream
- Better for the compiler
  - Program is easier to analyze
    - No code for timing / embedding
    - No control channels in stream graph
  - Can reorder filter firings, respecting constraints
  - Implement in most efficient way
StreamIt Language Summary

- High-level, machine-independent stream language
  - Structured streams for high-bandwidth dataflow
  - Messaging system for control
  - Working on new dynamic constructs
- Compiler-conscious language design can improve both programmability and performance
Stream Programming Models

- Prototyping environments
- Conventional languages
  - Object Oriented
  - Procedural
  - Assembly
- Stream languages
  - StreamIt
  - Brook
  - Cg
The Brook Language (Stanford)

- Also an architecture-independent stream language
  - Evolved out of StreamC / KernelC, which targets Imagine

<table>
<thead>
<tr>
<th><strong>StreamIt</strong></th>
<th><strong>Brook/StreamC</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Think structured synchronous data flow</td>
<td>Think pointer-less C, with embedded dataflow graphs instead of loop nests</td>
</tr>
<tr>
<td>Single stream graph</td>
<td>Multiple stream graphs, surrounded by C-subset</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 require sequential processing</td>
<td>“Kernels” must be state-less, allow parallel processing</td>
</tr>
<tr>
<td>Designed by compiler people, clean but more constrained</td>
<td>Designed by application and architecture people, rough but more expressive</td>
</tr>
</tbody>
</table>
Stream Programming Models

- Prototyping environments
- Conventional languages
  - Object Oriented
  - Procedural
  - Assembly
- Stream languages
  - StreamIt
  - Brook
  - Cg
The Cg language (NVIDIA)

- Cg is both a language and a system
  - Cg language is for writing stream kernels
  - Cg system targets graphics hardware
- Developed by NVIDIA
  - In collaboration with Microsoft
  - Runs on lots of hardware (not just NVIDIA’s)
- Widely deployed
  - Shipping for over a year
  - Anyone can download it
- Lots of information available:
  - Cg language specification – via download
  - Cg tutorial – buy on amazon.com
  - Paper in SIGGRAPH 2003
GPUs are now programmable

- Vertex Processor
- Triangle Assembly & Rasterization
- Fragment Processor
- Framebuffer Operations
- Texture Decompression & Filtering
- Textures

```
MOV R4.y, R2.y;
ADD R4.x, -R4.y, C[3].w;
MAD R3.xy, R2, R3.xyww, C[2].z;
...
```

```
ADD R3.xy, R3.xyww, C11.z;
TEX H5, R3, TEX2, 2D;
TEX H6, R3, TEX2, 2D;
...
```
Programmable units in GPUs are stream processors

- The programmable unit executes a computational kernel for each input element
- Streams consist of ordered elements
Design Decisions in Cg

- **Cg**: C for graphics
  - Like C, directly map to underlying hardware
  - General purpose (not just a shading language)

- **A program for each pipeline stage**
  - Alternative: write one program and have compiler do the partitioning
  - Chose to separate at programmer level to guarantee valid mapping; e.g., for outer-level control dependences

- **A language for expressing stream kernels**
  - Unlike StreamIt/Brook, does not express high-level connections in stream graph
  - Instead, write kernels for hardware resources and use connections of hardware
  - Use auxiliary namespace (bind-by-name) as dataflow interface between vertex and fragment processors
  - Use Cg runtime API to control kernel execution
void simpleTransform(float4 objectPosition : POSITION,  
                   float4 color : COLOR, 
                   float4 decalCoord : TEXCOORD0,  
                   out float4 clipPosition : POSITION,  
                   out float4 Color : COLOR,  
                   out float4 oDecalCoord : TEXCOORD0,  
                   uniform float brightness,  
                   uniform float4x4 modelViewProjection)
{
    clipPosition = mul(modelViewProjection, objectPosition);
    oColor = brightness * color;
    oDecalCoord = decalCoord;
}
How should system support different levels of HW?

- HW capabilities change each generation
  - Data types
  - Support for branch instructions, ...
- We expect this problem to persist
  - Future GPUs will have new features
- Mandate exactly one feature set?
  - Must strand older HW or limit newer HW
Two options for handling HW differences

- Emulate missing features?
  - Too slow on GPU
  - Too slow on CPU, especially for fragment HW

- Expose differences to programmer?
  - They chose this option
  - Differences exposed via subsetting
  - A profile is a named subset
  - Cg supports function overloading by profile
Cg is closely related to other recent languages

- Microsoft HLSL
  - Largely compatible with Cg
  - NVIDIA and Microsoft collaborated
- OpenGL ARB shading language
- All three languages are similar
  - Overlapping development
  - Extensive cross-pollination of ideas
  - Designers mostly agreed on right approach
- Systems are different
Summary

- There are many prototyping environments for streaming applications
- However, industry still relies on C, C++, and assembly code for implementations
  - Tedious, costly, error-prone
- Stream languages have potential to improve both performance and programmability
  - Expose communication patterns
  - Expose parallelism
  - Encapsulate common idioms
- Examples: StreamIt, Brook, Cg