StreamIt: A Language and Compiler for the Streaming Domain

Saman Amarasinghe

Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Outline

- Historical timeline of Stream Programming
- The StreamIt Language
- DSP Domain Specific Optimizations
- Architecture Specific Optimizations
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 determinacy, termination, queuing properties
Timeline: 1970’s

- 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

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

Languages / Compilers
- Lucid
- Id
- Sisal
- C
- lazy
- VAL
- Occam

Modeling Environments
- Gabriel

- **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**

<table>
<thead>
<tr>
<th>Year</th>
<th>Models of Computation</th>
<th>Languages / Compilers</th>
<th>Modeling Environments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1960</td>
<td>Petri Nets</td>
<td>Lucid, Id, Sisal, C</td>
<td>Ptolemy, Matlab/Simulink, Gabriel, Grape-II, etc.</td>
</tr>
<tr>
<td>1970</td>
<td>Comp Graphs</td>
<td>lazy VAL, Occam</td>
<td></td>
</tr>
<tr>
<td>1980</td>
<td>PN, CSP</td>
<td>Signal, LUSTRE</td>
<td></td>
</tr>
<tr>
<td>1990</td>
<td>SDF, FSM</td>
<td>Esterel</td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Synchronous Languages**: Signal, LUSTRE, etc.
  - Designed for expressiveness, verification more so 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
Outline

- Historical timeline of Stream Programming
- The StreamIt Language
- DSP Domain Specific Optimizations
- Architecture Specific Optimizations
Von Neumann Languages

- Why C (FORTRAN, C++ etc.) became very successful?
  - Abstracted out the differences of von Neumann machines
    - Register set structure
    - Functional units and capabilities
    - Pipeline depth/width
    - Memory/cache organization
  - Directly expose the common properties
    - Single memory image
    - Single control-flow
    - A clear notion of time
  - Can have a very efficient mapping to a von Neumann machine
  - “C is the portable machine language for von Neumann machines”
- Today von Neumann languages are a curse
  - We have squeezed out all the performance out of C
  - We can build more powerful machines
  - But, cannot map C into next generation machines
  - Switching to a data-flow programming paradigm will help
The StreamIt Project

- Goals:
  - Breaks the Von Neumann language barrier
  - Provide a High-Level Programming Paradigm
  - Improve Programmer Productivity
  - Be Compiler Friendly

- What’s new?
  - Practical language features for large application development
  - Stream-specific compiler technology
Language Features

- Circular Buffer Management
  - Reuse the data on the tapes using ‘peek’
- Structured Streams
  - Hierarchical
  - Parameterized
- Precise Event Handling
  - A formal notion of time
Example: Freq band detection

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

Source:
Application Report SPRA414
Texas Instruments, 1999
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 Detector(i/4);
            add LEDOutput(i);
        }
        join roundrobin(0);
    }
}
Freq band detection in StreamIt

```c
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);
    } // Band Pass Filter

    add splitjoin {
        split duplicate;
        for (int i=0; i<4; i++) {
            add Detector(i/4);
            add LEDOutput(i);
        }
        join roundrobin(0);
    }
}
```
Freq band detection in StreamIt

```c
void -> void pipeline FrequencyBand()
{
    float sFreq = 4000;
    float cFreq = 500/(sFreq*2*pi);
    float wFreq = 100/(sFreq*2*pi);
    add D2A Source(sFreq);
    add BandPassFilter(1, cFreq-wFreq, cFreq+wFreq, 100);
}
```

```c
float -> float filter LowPassFilter(float gain, float wS, float wp, int n)
{
    float[N] h;

    init{
        int OFF = N/2;
        for (int i=0; i<N; i++) {
            h[i] = g*sin(...);
        }
    }

    work peek N pop 1 push 1 {
        float sum = 0;
        for (int i=0; i<N; i++) {
            sum += h[i]*peek(i);
        }
        push(sum);
        pop();
    }
}
```

The New Laboratory for Computer Science and Artificial Intelligence
Freq band detection in StreamIt

```c
float->float filter LowPassFilter(float g, float cFreq, int N)
{
    float[N] h;

    init {
        int OFF = N/2;
        for (int i=0; i<N; i++) {
            h[i] = g*sin(...);
        }
    }

    work peek N pop 1 push 1 {
        float sum = 0;
        for (int i=0; i<N; i++) {
            sum += h[i]*peek(i);
        }
        push(sum);
        pop();
    }
}
```

```c
add LowPassFilter(1, wp, num);
```
Outline

- Historical timeline of Stream Programming
- The StreamIt Language
- DSP Domain Specific Optimizations
- Architecture Specific Optimizations
Conventional DSP Design Flow

Spec. (data-flow diagram) → Design the Datapaths (no control flow) → DSP Optimizations → Coefficient Tables → Rewrite the program → Architecture-specific Optimizations (performance, power, code size) → C/Assembly Code → Signal Processing Expert in Matlab → Software Engineer in C and Assembly
Any Design Modifications?

- 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 assembly
  - If using StreamIt
    - Change one constant
    - Recompile
Design Flow with StreamIt

Application-Level Design

StreamIt Program (dataflow + control)

DSP Optimizations

Architecture-Specific Optimizations

C/Assembly Code

Application Programmer

StreamIt compiler
Design Flow with StreamIt

- Benefits of programming in a single, high-level abstraction
  - Modular
  - Composable
  - Portable
  - Malleable

- The Challenge: Maintaining Performance
Our Focus: Linear Filters

- Most common target of DSP optimizations
  - FIR filters
  - Compressors
  - Expanders
  - DFT/DCT
  \[\text{Output is weighted sum of inputs}\]

- Example optimizations:
  - Combining Adjacent Nodes
  - Translating to Frequency Domain
Representing Linear Filters

- A linear filter is a tuple $\langle \mathbf{A}, \mathbf{b}, o \rangle$
  - $\mathbf{A}$: matrix of coefficients
  - $\mathbf{b}$: vector of constants
  - $o$: number of items popped

- Example

\[
x \xrightarrow{\langle \mathbf{A}, \mathbf{b}, o \rangle} y = x \mathbf{A} + \mathbf{b}
\]
Representing Linear Filters

- A linear filter is a tuple \( \langle \mathbf{A}, \mathbf{b}, \mathbf{o} \rangle \)
  - \( \mathbf{A} \): matrix of coefficients
  - \( \mathbf{b} \): vector of constants
  - \( \mathbf{o} \): number of items popped

- Example

\[
\mathbf{A} = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \quad \mathbf{o} = 1
\]
Extracting Linear Representation

- Resembles constant propagation
- Maintains linear form $\langle \hat{v}, \hat{b} \rangle$ for each variable
  - Peek expression: generate fresh $\hat{v}$
  - Push expression: copy $\hat{v}$ into $A$
  - Pop expression: increment $o$

```
work peek N pop 1 push 1 {
    float sum = 0;
    for (int i=0; i<N; i++) {
        sum += h[i]*peek(i);
    }
    push(sum);
    pop();
}
```
Optimizations using Linear Analysis

- Combining adjacent linear structures
- Shifting from time to the frequency domain
- Selection of ‘optimal’ set of transformations
1) Combining Linear Filters

- Pipelines and splitjoins can be collapsed
- Example: pipeline

\[ y = xA \]
\[ z = yB \]
\[ z = x \begin{array}{c} \text{Combined Filter} \\ A \end{array} \begin{array}{c} \text{Combined Filter} \\ B \end{array} \begin{array}{c} \text{Combined Filter} \\ C \end{array} \]

The New Laboratory for Computer Science and Artificial Intelligence

Massachusetts Institute of Technology
Combination Example

\[
A = \begin{bmatrix} 4 & 5 & 6 \end{bmatrix}
\]

\[
B = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}
\]

\[
C = [32]
\]
AB for any A and B??

Need to “expand” matrices to a steady-state cycle:

\[ A \quad pop = 1 \quad B \quad pop = 2 \]

\[ A' \quad B' \]
AB for any A and B??

Need to “expand” matrices to a steady-state cycle:

A

pop = 1

B

pop = 2

A’

B’
AB for any A and B??

Need to “expand” matrices to a steady-state cycle:

\( A \)

\( pop = 1 \)

\( B \)

\( pop = 2 \)

\( A' \)

\( B' \)
AB for any A and B??

Need to “expand” matrices to a steady-state cycle:
AB for any A and B??

Need to “expand” matrices to a steady-state cycle:
AB for any A and B??

Need to “expand” matrices to a steady-state cycle:

\[ \begin{align*}
\text{A} & : \quad \text{pop} = 1 \\
\text{B} & : \quad \text{pop} = 2
\end{align*} \]

\[ \begin{align*}
\text{A'} & : \\
\text{B'} & 
\end{align*} \]
AB for any A and B??

Need to “expand” matrices to a steady-state cycle:

A

\[ \text{pop} = 1 \]

A'

B

\[ \text{pop} = 2 \]

B'
AB for any A and B??

Need to “expand” matrices to a steady-state cycle:

\[ \begin{align*}
A & \quad pop = 1 \\
B & \quad pop = 2 \\
A' & \quad \text{\smaller \text{\texttimes\texttimes}} \\
B' & \quad \text{\smaller \text{\texttimes\texttimes}}
\end{align*} \]
AB for any A and B??

Need to “expand” matrices to a steady-state cycle:

A

\[ \text{pop} = 1 \]

B

\[ \text{pop} = 2 \]
AB for any A and B??

Need to "expand" matrices to a steady-state cycle:

A

\[
\text{pop} = 1
\]

B

\[
\text{pop} = 2
\]
AB for any A and B??

Need to “expand” matrices to a steady-state cycle:
AB for any A and B??

Need to “expand” matrices to a steady-state cycle:

\[ A \]
\[ \text{pop = 1} \]

\[ B \]
\[ \text{pop = 2} \]

\[ A' \]
\[ \text{pop = 2} \]

\[ B' \]
\[ \text{pop = 6} \]
AB for any A and B??

Expanded dimensions match a steady-state cycle:

- **A**: pop = 1
- **B**: pop = 2
- **A'**: pop = 2
- **B'**: pop = 6
AB for any A and B??

- Linear Expansion

Original

\[
[A] \quad \text{pop} = \sigma
\]

Expanded

\[
\begin{bmatrix}
\end{bmatrix}
\]
Floating-Point Operations Reduction

![Bar chart showing the percentage of floating-point operations removed for various benchmarks. The benchmarks include FIR, RateConvert, TargetDetect, FMRadio, Radar, FilterBank, Vocoder, Oversample, and DTOA. The percentage of flops removed ranges from 0% to 100%. The chart indicates that TargetDetect has the highest percentage of flops removed at 0.3%.](chart.png)
2) From Time to Frequency Domain

- Convolutions can be done cheaply in the Frequency Domain
  \[ \sum X_i \ast W_{n-i} \]

- Painful to do by hand
  - Blocking
  - Coefficient calculations
  - Startup etc.
Floating-Point Operations Reduction

![Bar chart showing flops removed (%)](chart.png)

- **FIR**
- **RateConvert**
- **TargetDetect**
- **FMRadio**
- **Radar**
- **FilterBank**
- **Vocoder**
- **Oversample**
- **DTDIA**

**Flops Removed (%)**

- **linear**
- **freq**

**Benchmarks**

- FIR
- RateConvert
- TargetDetect
- FMRadio
- Radar
- FilterBank
- Vocoder
- Oversample
- DToA

- 140%
- 0.3%

The New Laboratory for Computer Science and Artificial Intelligence

Massachusetts Institute of Technology
3) Transformation Selection

- When to apply what transformations?
  - Linear filter combination can increase the computation cost
  - Shifting to the Frequency domain is expensive for filters with pop > 1
    - Compute all outputs, then decimate by pop rate
  - Some expensive transformations may later enable other transformations, reducing the overall cost
Radar (Linear Analysis)
Radar (Maximal Linear Combination)
Radar (Maximal Linear Combination)
Radar (Maximal Frequency)

65% slower!

2.4 times as many FLOPS
Radar
Radar (Linear Analysis)
Radar (Maximal Linear Combination)
Radar (Maximal Linear Combination)
Radar (Maximal Linear Combination)
Radar (Maximal Frequency)

65% slower!

2.4 times as many FLOPS
Selection Algorithm

- Estimate minimal cost for each structure:
  - Linear combination
  - Frequency translation
  - No transformation
    - If hierarchical, consider all possible groupings of children

- Overlapping sub-problems allows efficient dynamic programming search
Radar (Transformation Selection)
Radar (Transformation Selection)

First compute cost of individual filters:
Radar (Transformation Selection)

First compute cost of individual filters:

- Linear Combination
- Frequency
- No Transform
Radar (Transformation Selection)

First compute cost of individual filters:

Linear Combination

Frequency

No Transform

1x1
Then, compute cost of 1x2 nodes:
Radar (Transformation Selection)

Then, compute cost of 1x2 nodes:

- Linear Combination
- Frequency
- No Transform

1x1

The New Laboratory for Computer Science and Artificial Intelligence
Massachusetts Institute of Technology
Radar (Transformation Selection)

Then, compute cost of 1x2 nodes:

1x2

1x1
Radar (Transformation Selection)

Continue with 1x3 2x1 3x1 4x1

1x4 2x2 3x2 4x2

2x3 3x3 4x3

2x4 3x4 4x4

Overall solution
Radar (Transformation Selection)
Radar (Transformation Selection)
Radar

Maximal Combination and Shifting to Frequency Domain

Using Transformation Selection

2.4 times as many FLOPS

half as many FLOPS
Floating-Point Operations Reduction

Flops Removed (%)

Benchmark

- 140%

- 40%
- 20%
- 0%
0%
20%
40%
60%
80%
100%

FIR
RateConvert
TargetDetect
FMRadio
Radar
FilterBank
Vocoder
Oversample
DTOA

linear
freq
autosel

0.3%
Outline

- Historical timeline of Stream Programming
- The StreamIt Language
- DSP Domain Specific Optimizations
- Architecture Specific Optimizations
RAW: A Wire Exposed Architecture

- A wire can cross a tile in a single clock cycle
  - Wire delay is not a issue in the processor design
- Ultra fast interconnect network
  - Exposes the wires to the compiler
  - Compiler orchestrate the communication → hide wire delay
Raw Chip

IBM SA-27E .15u 6L Cu
18.2 mm x 18.2 mm
16 Flops/ops per cycle
208 Operand Routes / cycle
2048 KB L1 SRAM
1657 Pin CCGA Package
1080 HSTL core-speed signal I/O

Core Tested @ 420 MHz!!

3.6 Peak GFLOPS (without FMAC!)
225 Gb/s on-chip bisection bandwidth
201 Gb/s off-chip I/O bandwidth
How to execute a Stream Graph?
Method 1: Time Multiplexing

- Run one filter at a time

- Pros:
  - Scheduling is easy
  - 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
  - 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 ≤ \# processing elements
Example: Radar Array Front End
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 dynamic programming
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:
- Splitter
- Filter0 → ⋯ → FilterN → Joiner
- Filter
- Filter0 → Filter1 → ⋯ → FilterN → 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
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 after executing the schedule
  - There exist a unique minimum steady state schedule

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

- 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 the after executing the schedule
  - There exist a unique minimum steady state schedule

- 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

- 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

- 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

- 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

- 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
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
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

- Blocked on Static Network
- Executing Instructions
- Pipeline Stall
Performance of Radar

- **C program**
  - 1 GHz Pentium III: 240 MFLOPS
  - 420 MHz single tile Raw: 19 MFLOPS

- **Unoptimized StreamIt**
  - 420 MHz 64 tile Raw: 969 MFLOPS

- **Optimized StreamIt**
  - 420 MHz 16 tile Raw: 2,066 MFLOPS

The New Laboratory for Computer Science and Artificial Intelligence

Massachusetts Institute of Technology
Application Performance

Throughput of StreamIt normalized to single tile C

- FIR
- FFT
- Radio
- 3GPP
- Sort
- Radar
- Filterbank
Scalability of StreamIt

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

Tile Utilization

Bitonic Sort

The New Laboratory for Computer Science and Artificial Intelligence

Massachusetts Institute of Technology
Conclusions

- Streaming Programming Model
  - Can break the von Neumann bottleneck
  - A natural fit for a large class of applications
  - An ideal model for communication-exposed architectures.

- Can we replace the DSP engineer from the design flow?
  - On the average 90% of the FLOPs eliminated
  - Average performance speedup of 450%

- Increased abstraction does not have to sacrifice performance
- The hardware is ready for the stream programming model

http://cag.lcs.mit.edu/commit/