StreamIt – A Programming Language for the Era of Multicores

Saman Amarasinghe

http://cag.csail.mit.edu/streamit
Moore’s Law


From David Patterson
Uniprocessor Performance (SPECint)

**Uniprocessor Performance (SPECint)**

- **General-purpose unicore unicoses have stopped historic performance scaling**
  - Power consumption
  - Wire delays
  - DRAM access latency
  - Diminishing returns of more instruction-level parallelism


*From David Patterson*
Two choices:

- Bend over backwards to support old languages like C/C++
- Develop high-performance architectures that are hard to program
Parallel Programmer’s Dilemma

\[ F(u, v) = \frac{2}{N} C(u) C(v) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} f(x, y) \cos\left(\frac{(2x + 1)\pi}{2N}\right) \cos\left(\frac{(2y + 1)\pi}{2N}\right) \]

Rapid prototyping
- MATLAB
- Ptolemy

Automatic parallelization
- FORTRAN compilers
- C/C++ compilers

Manual parallelization
- C/C++ with MPI

Optimal parallelization
- assembly code

Malleability
- high
- low

Portability
- high
- low

Productivity
- high
- low

Natural parallelization
- StreamIt

Parallel Performance
- high
- low

MIT CSAIL
Stream Application Domain

- Graphics
- Cryptography
- Databases
- Object recognition
- Network processing and security
- Scientific codes
- …
StreamIt Project

- **Language Semantics / Programmability**
  - StreamIt Language (CC 02)
  - Programming Environment in Eclipse (P-PHEC 05)

- **Optimizations / Code Generation**
  - Phased Scheduling (LCTES 03)
  - Cache Aware Optimization (LCTES 05)

- **Domain Specific Optimizations**
  - Linear Analysis and Optimization (PLDI 03)
  - Optimizations for bit streaming (PLDI 05)
  - Linear State Space Analysis (CASES 05)

- **Parallelism**
  - Teleport Messaging (PPOPP 05)
  - Compiling for Communication-Exposed Architectures (ASPLOS 02)
  - Load-Balanced Rendering (Graphics Hardware 05)

- **Applications**
  - SAR, DSP benchmarks, JPEG,
  - MPEG [IPDPS 06], DES and Serpent [PLDI 05], …
Compiler-Aware Language Design

boost productivity, enable faster development and rapid prototyping

programmability

domain specific optimizations

simple and effective optimizations for domain specific abstractions

enable parallel execution

target multicores, clusters, tiled architectures, DSPs, graphics processors, …
Streaming Application Design

- Structured block level diagram describes computation and flow of data
- Conceptually easy to understand
  - Clean abstraction of functionality
StreamIt Philosophy

- Preserve program structure
  - Natural for application developers to express

- Leverage program structure to discover parallelism and deliver high performance

- Programs remain clean
  - Portable and malleable
StreamIt Philosophy

MPEG bit stream

- VLD
- macroblocks, motion vectors
- splitter
- frequency encoded macroblocks
- differentially coded motion vectors
- ZigZag
- IQuantization
- Motion Vector Decode
- IDCT
- Saturation
- Repeat
- motion vectors
- joiner
- spatially encoded macroblocks
- add VLD(QC, PT1, PT2);
- add splitjoin {
  split roundrobin(N*B, V);
}
- add pipeline {
  add ZigZag(B);
  add IQuantization(B) to QC;
  add IDCT(B);
  add Saturation(B);
}
- add pipeline {
  add MotionVectorDecode();
  add Repeat(V, N);
}
- join roundrobin(B, V);
- add splitjoin {
  split roundrobin(4*(B+V), B+V, B+V);
}
- add MotionCompensation(4*(B+V)) to PT1;
- for (int i = 0; i < 2; i++) {
  add pipeline {
    add MotionCompensation(B+V) to PT1;
    add ChannelUpsample(B);
  }
}
- join roundrobin(1, 1, 1);
- add PictureReorder(3*W*H) to PT2;
- add ColorSpaceConversion(3*W*H);
Stream Abstractions in StreamIt

MPEG bit stream

- **filters**
  - **VLD**
  - **splitter**
    - **ZigZag**
      - **IQuantization**
        - **IDCT**
          - **Saturation**
  - **joiner**
    - **Motion Vector Decode**
      - **Repeat**
    - **splitjoin**
      - **pipeline**
        - **add pipeline**
          - **add MotionVectorDecode();**
          - **add Repeat(V, N);**
        - **join roundrobin(B, V);**

- **splitjoins**
  - **split join**
    - **split roundrobin(N*B, V);**

- **pipelines**
  - **add VLD(QC, PT1, PT2);**
  - **add pipeline**
    - **add ZigZag(B);**
    - **add IQuantization(B) to QC;**
    - **add IDCT(B);**
    - **add Saturation(B);**
  - **add pipeline**
    - **add MotionCompensation(4*(B+V)) to PT1;**
    - **for (int i = 0; i < 2; i++) {**
      - **add pipeline**
        - **add MotionCompensation(B+V) to PT1;**
        - **add ChannelUpsample(B);**
      - **}**
    - **join roundrobin(1, 1, 1);**
  - **add PictureReorder(3*W*H) to PT2;**
  - **add ColorSpaceConversion(3*W*H);**
StreamIt Language Highlights

• Filters

• Pipelines

• Splitjoins

• Teleport messaging
Example StreamIt Filter

```
float → float filter FIR (int N) {
  work push 1 pop 1 peek N {
    float result = 0;
    for (int i = 0; i < N; i++) {
      result += weights[i] * peek(i);
    }
    push(result);
    pop();
  }
}
```
FIR Filter in C

void FIR(
    int* src,
    int* dest,
    int* srcIndex,
    int* destIndex,
    int srcBufferSize,
    int destBufferSize,
    int N) {

    float result = 0.0;
    for (int i = 0; i < N; i++) {
        result += weights[i] * src[(*srcIndex + i) % srcBufferSize];
    }
    dest[*destIndex] = result;
    *srcIndex = (*srcIndex + 1) % srcBufferSize;
    *destIndex = (*destIndex + 1) % destBufferSize;
}

• FIR functionality obscured by buffer management details

• Programmer must commit to a particular buffer implementation strategy
StreamIt Language Highlights

• Filters

• Pipelines

• Splitjoins

• Teleport messaging
Example StreamIt Pipeline

• Pipeline
  – Connect components in sequence
  – Expose pipeline parallelism

```c
float→float pipeline 2D_iDCT (int N)
{
  add Column_iDCTs(N);
  add Row_iDCTs(N);
}
```
Preserving Program Structure

From Figures 7-1 and 7-4 of the MPEG-2 Specification (ISO 13818-2, P. 61, 66)
In Contrast: C Code Excerpt

- Explicit for-loops iterate through picture frames
- Frames passed through global arrays, handled with pointers
- Mixing of parser, motion compensation, and spatial decoding

```c
EXTERN unsigned char *backward_reference_frame[3];
EXTERN unsigned char *forward_reference_frame[3];
EXTERN unsigned char *current_frame[3];
...etc...
```

```c
Decode_Picture {
    for (;;) {
        parser();
        for (;;) {
            decode_macroblock();
            motion_vectors();
            for (comp=0; comp<block_count; comp++) {
                parser();
                Decode_MPEG2_Block();
            }
        }
    }
}
```

```c
motion_vectors() {
    parser();
    decode_motion_vector
    parser();
}
```

```c
Decode_MPEG2_Block() {
    for (int i = 0; i++ ) {
        parsing();
        ZigZagUnordering();
        inverseQuantization();
        if (condition) then break;
    }
}
```

```c
motion_compensation() {
    decode_macroblock();
    motion_compensation();
    if (condition) then break;
}
```

```c
frame_reorder();
```
StreamIt Language Highlights

- Filters
- Pipelines
- Splitjoins
- Teleport messaging
Example StreamIt Splitjoin

- **Splitjoin**
  - Connect components in parallel
  - Expose task parallelism and data distribution

```c
float→float splitjoin Row_iDCT (int N) {
    split roundrobin(N);
    for (int i = 0; i < N; i++) {
        add 1D_iDCT(N);
    }
    join roundrobin(N);
}
```
Example StreamIt Splitjoin

```c
float→float pipeline 2D_iDCT (int N)
{
    add Column_iDCTs(N);
    add Row_iDCTs(N);
}

float→float splitjoin Column_iDCT (int N)
{
    split roundrobin(1);
    for (int i = 0; i < N; i++) {
        add 1D_iDCT(N);
    }
    join roundrobin(1);
}

float→float splitjoin Row_iDCT (int N)
{
    split roundrobin(N);
    for (int i = 0; i < N; i++) {
        add 1D_iDCT(N);
    }
    join roundrobin(N);
}
```
Naturally Expose Data Distribution

scatter macroblocks according to chroma format

```cpp
add splitjoin {
    split roundrobin(4*(B+V), B+V, B+V);
    add MotionCompensation();
    for (int i = 0; i < 2; i++) {
        add pipeline {
            add MotionCompensation();
            add ChannelUpsample(B);
        }
    }
    join roundrobin(1, 1, 1);
}
```

gather one pixel at a time
Stream Graph Malleability

4:2:0 chroma format

4:2:2 chroma format
StreamIt Code Sample

red  = code added or modified to support 4:2:2 format

// C = blocks per chroma channel per macroblock
// C = 1 for 4:2:0, C = 2 for 4:2:2
add splitjoin {
    split roundrobin(4*(B+V), 2*C*(B+V));

    add MotionCompensation();
    add splitjoin {
        split roundrobin(B+V, B+V);

        for (int i = 0; i < 2; i++) {
            add pipeline {
                add MotionCompensation()
                add ChannelUpsample(C,B);
            }
        }

        join roundrobin(1, 1);
    }

    join roundrobin(1, 1, 1);
}

join roundrobin(1, 1, 1);
In Contrast: C Code Excerpt

red = pointers used for address calculations

/* Y */
form_component_prediction(src[0]+(sfield?lx2>>1:0),dst[0]+(dfield?lx2>>1:0),
     lx, lx2, w, h, x, y, dx, dy, average_flag);

if (chroma_format!=CHROMA444) {
    lx1>>=1;  lx2>>=1;  w1>>=1;  x1>>=1;  dx1/=2;
} if (chroma_format==CHROMA420) {
    h1>>=1;  y1>>=1;  dy1/=2;
}

/* Cb */
form_component_prediction(src[1]+(sfield?lx2>>1:0),dst[1]+(dfield?lx2>>1:0),
     lx, lx2, w, h, x, y, dx, dy, average_flag);

/* Cr */
     lx, lx2, w, h, x, y, dx, dy, average_flag);

Adjust values used for address calculations depending on the chroma format used.
StreamIt Language Highlights

• Filters

• Pipelines

• Splitjoins

• Teleport messaging
Teleport Messaging

- Avoids muddling data streams with control relevant information
- Localized interactions in large applications
  - A scalable alternative to global variables or excessive parameter passing
Motion Prediction and Messaging

portal\langle Motion\text{Compensation} \rangle \ PT;

add splitjoin {
    split roundrobin(4*(B+V), B+V, B+V);
    add Motion\text{Compensation}() to PT;
    for (int i = 0; i < 2; i++) {
        add pipeline {
            add Motion\text{Compensation}() to PT;
            add Channel\text{Upsample}(B);
        }
    }
    join roundrobin(1, 1, 1);
}

Teleport Messaging Overview

- Looks like method call, but timed relative to data in the stream

```java
TargetFilter x;
if newPictureType(p) {
    x.setPictureType(p) @ 0;
}

void setPicturetype(int p) {
    reconfigure(p);
}
```

- Simple and precise for user
  - Exposes dependences to compiler
  - Adjustable latency
  - Can send upstream or downstream
Messaging Equivalent in C

The MPEG Bitstream

Decode Picture

Decode Macroblock

Decode Block

ZigZagUnordering

Inverse Quantization

Global Variable Space

Frame Reordering

Output Video

File Parsing

Decode Motion Vectors

Motion Compensation

Saturate

IDCT

Motion Compensation For Single Channel
Compiler-Aware Language Design

boost productivity, enable faster development and rapid prototyping

programmability

domain specific optimizations

simple and effective optimizations for domain specific abstractions

enable parallel execution

target multicores, clusters, tiled architectures, DSPs, graphics processors, …
Multicores Are Here!
Von Neumann Languages

• Why C (FORTRAN, C++ etc.) became very successful?
  – Abstracted out the differences of von Neumann machines
  – Directly expose the common properties
  – Can have a very efficient mapping to a von Neumann machine
  – “C is the portable machine language for von Neumann machines”

• von Neumann languages are a curse for Multicores
  – We have squeezed out all the performance out of C
  – But, cannot easily map C into multicores
Common Machine Languages

### Unicores:

<table>
<thead>
<tr>
<th>Common Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single flow of control</td>
</tr>
<tr>
<td>Single memory image</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Differences:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register File</td>
</tr>
<tr>
<td>ISA</td>
</tr>
<tr>
<td>Functional Units</td>
</tr>
</tbody>
</table>

### Multicores:

<table>
<thead>
<tr>
<th>Common Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiple flows of control</td>
</tr>
<tr>
<td>Multiple local memories</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Differences:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number and capabilities of cores</td>
</tr>
<tr>
<td>Communication Model</td>
</tr>
<tr>
<td>Synchronization Model</td>
</tr>
</tbody>
</table>

von-Neumann languages represent the common properties and abstract away the differences.
Bridging the Abstraction layers

- **StreamIt** exposes the data movement
  - Graph structure is architecture independent
- **StreamIt** exposes the parallelism
  - Explicit task parallelism
  - Implicit but inherent data and pipeline parallelism
- Each multicore 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 cores, memory and the communication substrate
Types of Parallelism

Task Parallelism
- Parallelism explicit in algorithm
- Between filters *without* producer/consumer relationship

Scatter

Gather

Task
Types of Parallelism

- **Task Parallelism**
  - Parallelism explicit in algorithm
  - Between filters *without* producer/consumer relationship

- **Data Parallelism**
  - Between iterations of a *stateless* filter
  - Place within scatter/gather pair (*fission*)
  - Can’t parallelize filters with state

- **Pipeline Parallelism**
  - Between producers and consumers
  - *Stateful* filters can be parallelized
Types of Parallelism

Traditionally:

Task Parallelism
- Thread (fork/join) parallelism

Data Parallelism
- Data parallel loop (forall)

Pipeline Parallelism
- Usually exploited in hardware
Problem Statement

Given:
– Stream graph with compute and communication estimate for each filter
– Computation and communication resources of the target machine

Find:
– Schedule of execution for the filters that best utilizes the available parallelism to fit the machine resources
Our 3-Phase Solution

1. **Coarsen**: Fuse stateless sections of the graph
2. **Data Parallelize**: parallelize stateless filters
3. **Software Pipeline**: parallelize stateful filters

Compile to a 16 core architecture
- 11.2x mean throughput speedup over single core
Baseline 1: Task Parallelism

- Inherent task parallelism between two processing pipelines

- Task Parallel Model:
  - Only parallelize explicit task parallelism
  - Fork/join parallelism

- Execute this on a 2 core machine
  \(~2x \text{ speedup over single core}\)

- What about 4, 16, 1024, \ldots \text{ cores?}
Evaluation: Task Parallelism

Parallelism: Not matched to target!
Synchronization: Not matched to target!
Cycle accurate simulator
Baseline 2: Fine-Grained Data Parallelism

- Each of the filters in the example are stateless
- Fine-grained Data Parallel Model:
  - *Fiss* each stateless filter $N$ ways ($N$ is number of cores)
  - Remove scatter/gather if possible
- We can introduce data parallelism
  - Example: 4 cores
- Each fission group occupies entire machine
Evaluation:
Fine-Grained Data Parallelism

Throughput Normalized to Single Core StreamIt

- Task
- Fine-Grained Data

BitonicSort
ChannelVocoder
DCT
DES
FFT
Filterbank
FMRadio
Serpent
TDE
MPEG2Decoder
Vocoder
Radar
Geometric Mean

Good Parallelism!
Too Much Synchronization!
Phase 1: Coarsen the Stream Graph

- Before data-parallelism is exploited
- *Fuse* stateless pipelines as much as possible without introducing state
  - Don’t fuse stateless with stateful
  - Don’t fuse a peeking filter with anything upstream
Phase 1: Coarsen the Stream Graph

- Before data-parallelism is exploited
- *Fuse* stateless pipelines as much as possible without introducing state
  - Don’t fuse stateless with stateful
  - Don’t fuse a peeking filter with anything upstream

- Benefits:
  - Reduces global communication and synchronization
  - Exposes inter-node optimization opportunities
Phase 2: Data Parallelize

Data Parallelize for 4 cores

Fiss 4 ways, to occupy entire chip
Phase 2: Data Parallelize

Data Parallelize for 4 cores

Task parallelism!
Each fused filter does equal work
Fiss each filter 2 times to occupy entire chip
Phase 2: Data Parallelize

Data Parallelize for 4 cores

- Task-conscious data parallelization
  - Preserve task parallelism
- Benefits:
  - Reduces global communication and synchronization

Task parallelism, each filter does equal work
Fiss each filter 2 times to occupy entire chip
Evaluation:
Coarse-Grained Data Parallelism

Throughput Normalized to Single Core StreamIt

Task
Fine-Grained Data
Coarse-Grained Task + Data

Good Parallelism!
Low Synchronization!
Simplified Vocoder

Data Parallel

Data Parallel, but too little work!

Target a 4 core machine
Data Parallelize

Target a 4 core machine
Data + Task Parallel Execution

Target 4 core machine
We Can Do Better!

Target 4 core machine
Phase 3: Coarse-Grained Software Pipelining

- New steady-state is free of dependencies
- Schedule new steady-state using a greedy partitioning
Greedy Partitioning

To Schedule:

Cores

Time

Target 4 core machine
Evaluation: Coarse-Grained Task + Data + Software Pipelining

Best Parallelism!
Lowest Synchronization!
Compiler-Aware Language Design

boost productivity, enable faster development and rapid prototyping

programmability

domain specific optimizations

simple and effective optimizations for domain specific abstractions

enable parallel execution

target multicores, clusters, tiled architectures, DSPs, graphics processors, …
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
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
Focus: Linear State Space Filters

• Properties:
  1. Outputs are linear function of inputs and states
  2. New states are linear function of inputs and states

• Most common target of DSP optimizations
  – FIR / IIR filters
  – Linear difference equations
  – Upsamplers / downsamplers
  – DCTs
Representing State Space Filters

- A state space filter is a tuple \( \langle A, B, C, D \rangle \)

\[
x' = Ax + Bu
\]

\[
y = Cx + Du
\]
Representing State Space Filters

- A state space filter is a tuple \( \langle A, B, C, D \rangle \)

\[
x' = Ax + Bu
\]

\[
y = Cx + Du
\]

float->float filter IIR {
  float x1, x2;
  work push 1 pop 1 {
    float u = pop();
    push(2*(x1+x2+u));
    x1 = 0.9*x1 + 0.3*u;
    x2 = 0.9*x2 + 0.2*u;
  }
}
Representing State Space Filters

- A state space filter is a tuple $\langle A, B, C, D \rangle$

```c
float->float filter IIR {
    float x1, x2;
    work push 1 pop 1 {
        float u = pop();
        push(2*(x1+x2+u));
        x1 = 0.9*x1 + 0.3*u;
        x2 = 0.9*x2 + 0.2*u;
    }
}
```

\[ x' = Ax + Bu \]
\[ y = Cx + Du \]

Linear dataflow analysis
Linear Optimizations

1. Combining adjacent filters
2. Transformation to frequency domain
3. Change of basis transformations
4. Transformation Selection
1) Combining Adjacent Filters

\[ y = D_1 u \]
\[ z = D_2 y \]
\[ z = D_2 D_1 u \]
\[ z = Eu \]
Combination Example

IIR Filter
\[ x' = 0.9x + u \]
\[ y = x + 2u \]

Decimator
\[ y = [1 \ 0] \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \]

IIR / Decimator
\[ x' = 0.81x + [0.9 \ 1] \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \]
\[ y = x + [2 \ 0] \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \]

8 FLOPs
---
output

6 FLOPs
---
output
Combination Example

IIR Filter
\[ x' = 0.9x + u \]
\[ y = x + 2u \]

Decimator
\[ y = [1 \ 0] \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \]

IIR / Decimator
\[ x' = 0.81x + [0.9 \ 1] \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \]
\[ y = x + [2 \ 0] \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \]

As decimation factor goes to $\infty$, eliminate up to 75% of FLOPs.
Floating-Point Operations Reduction

- FIR Rat
- eCo
- TargetDetect
- FM Radio
- Radar
- FilterBank
- Vocoder
- Oversample
- DTOA

Flops Removed (%)

Benchmark

linear

0.3%
2) From Time to Frequency Domain

- Convolutions can be done cheaply in the Frequency Domain

\[ \sum X_i * W_{n-i} \]

- Painful to do by hand
  - Blocking
  - Coefficient calculations
  - Startup etc.
FLOPs Reduction
3) Change-of-Basis Transformation

\[ x' = Ax + Bu \]
\[ y = Cx + Du \]

\[ T = \text{invertible matrix}, \ z = Tx \]

\[ z' = A'z + B'u \]
\[ y = C'z + D'u \]

\[ A' = TAT^{-1} \quad B' = TB \]
\[ C' = CT^{-1} \quad D' = D \]

Can map original states \( x \) to transformed states \( z = Tx \) without changing I/O behavior.
Change-of-Basis Optimizations

1. State removal
   - Minimize number of states in system

2. Parameter reduction
   - Increase number of 0’s and 1’s in multiplication

→ Formulated as general matrix operations
4) 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
FLOPs Reduction with Optimization Selection

Benchmark

FIR
RateConvert
TargetDetect
FMRadio
Radar
FilterBank
Vocoder
Oversample
DTOA

Flops Removed (%)

linear
freq
autosel

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

0.3%
-140%
Execution Speedup

On a Pentium IV
Conclusion

• Streaming programming model
  – Can break the von Neumann bottleneck
  – A natural fit for a large class of applications
  – An ideal machine language for multicores.

• Natural programming language for many streaming applications
  – Better modularity, composability, malleability and portability than C

• Compiler can easily extract explicit and inherent parallelism
  – Parallelism is abstracted away from architectural details of multicores
  – Sustainable Speedups (5x to 19x on the 16 core Raw)

• Can we replace the DSP engineer from the design flow?
  – On the average 90% of the FLOPs eliminated, average speedup of 450% attained

• Increased abstraction does not have to sacrifice performance

• The compiler and benchmarks are available on the web
  http://cag.csail.mit.edu/commit/
Thanks for Listening!

Any questions?

http://cag.csail.mit.edu/streamit