A Compiler Infrastructure for Stream Programs

William Thies

Joint work with Michael Gordon, Michal Karczmarek, Jasper Lin, Andrew Lamb, David Maze, Rodric Rabbah and Saman Amarasinghe

Massachusetts Institute of Technology

IBM PL Day
May 21, 2004
Streaming Application Domain

• Based on audio, video, or data stream
• Increasingly prevalent and important
  - Embedded systems
    • Cell phones, handheld computers
  - Desktop applications
    • Streaming media  ▪ Real-time encryption
    • Software radio ▪ Graphics packages
  - High-performance servers
    • Software routers (Example: Click)
    • Cell phone base stations
    • HDTV editing consoles
Properties of Stream Programs

• A large (possibly infinite) amount of data
  - Limited lifetime of each data item
  - Little processing of each data item

• Computation: apply multiple filters to data
  - Each filter takes an input stream, does some processing, and produces an output stream
  - Filters are independent and self-contained

• A regular, static communication pattern
  - Filter graph is relatively constant
  - A lot of opportunities for compiler optimizations
The StreamIt Project

• Goals:
  - Provide a high-level stream programming model
  - Invent new compiler technology for streams

• Contributions:
  - Language Design, Structured Streams, Buffer Management \textit{(CC 2002)}
  - Exploiting Wire-Exposed Architectures \textit{(ASPLOS 2002, ISCA 2004)}
  - Scheduling of Static Dataflow Graphs \textit{(LCTES 2003)}
  - Domain Specific Optimizations \textit{(PLDI 2003)}
  - Public release: Fall 2003
Outline

• Introduction
• StreamIt Language
• Domain-specific Optimizations
• Targeting Parallel Architectures
• Public Release
• Conclusions
Outline

• Introduction
• **StreamIt Language**
• Domain-specific Optimizations
• Targeting Parallel Architectures
• Public Release
• Conclusions
Model of Computation

- Synchronous Dataflow [Lee 1992]
  - Graph of independent filters
  - Communicate via channels
  - Static I/O rates

Freq band detector
Filter Example: LowPassFilter

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

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

    work peek N pop 1 push 1 {
        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 peek N pop 1 push 1 {
        float result = 0;
        for (int i=0; i<weights.length; i++) {
            result += weights[i] * peek(i);
        }
        push(result);
        pop();
    }
}
```
float->float filter LowPassFilter (int N, float freq) {
    float[N] weights;

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

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

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

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

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

N

filter
Composing Filters: 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(100, cFreq-wFreq, cFreq+wFreq);

    add splitjoin {
        split duplicate;
        for (int i=0; i<4; i++) {
            add pipeline {
                add Detect (i/4);
            }
        }
        join roundrobin(0);
    }
}
Radar-Array Front End
Filterbank
FM Radio with Equalizer
Bitonic Sort
FFT
Block Matrix Multiply
MP3 Decoder
Outline

• Introduction
• StreamIt Language
• Domain-specific Optimizations
• Targeting Parallel Architectures
• Public Release
• Conclusions
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

Source: Application Report SPRA414 Texas Instruments, 1999
Ideal DSP Design Flow

Application-Level Design

High-Level Program (dataflow + control)

DSP Optimizations

Architecture-Specific Optimizations

C/Assembly Code

Application Programmer

Compiler

Challenge: maintaining performance
Our Focus: Linear Filters

• Most common target of DSP optimizations
  • FIR filters
  • Compressors
  • Expanders
  • DFT/DCT

Output is weighted sum of inputs
Extracting Linear Representation

```
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();
}
```

Linear Dataflow Analysis

\[ \langle A, \vec{b} \rangle \]

\[ y = x A + \vec{b} \]
1) Combining Linear Filters

- Pipelines and splitjoins can be collapsed
- Example: pipeline

\[
\begin{align*}
  x &\quad \rightarrow \\
  \text{Filter 1} &\quad \rightarrow \quad y = x A \\
  y &\quad \rightarrow \\
  \text{Filter 2} &\quad \rightarrow \quad z = y B \\
  z &\quad \rightarrow \\
  \text{Combined Filter} &\quad \rightarrow \quad z = x C
\end{align*}
\]
Combination Example

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

Filter 1

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

Filter 2

\[ C = [32] \]

Combined Filter

6 mults output

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

1 mults output
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

- Multiple outputs
  - Interfacing with FFT library
  - Verification
3) When to Apply Transformations?

- Estimate minimal cost for each structure:
  - Linear combination
  - Frequency translation
  - No transformation
    - If hierarchical, consider all rectangular groupings of children
- Overlapping sub-problems allows efficient dynamic programming search
Radar (Transformation Selection)

Diagram showing the flow of data through various transformations such as decimation (Dec), CFilt, CFilt2, filter (Filter), magnetic field (Mag), and detection (Detect) stages.
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
Outline

• Introduction
• StreamIt Language
• Domain-specific Optimizations
• Targeting Parallel Architectures
• Public Release
• Conclusions
Compiling to the Raw Architecture
1. Partitioning: adjust granularity of graph
Compiling to the Raw Architecture

1. Partitioning: adjust granularity of graph
2. Layout: assign filters to tiles
Compiling to the Raw Architecture

1. Partitioning: adjust granularity of graph
2. Layout: assign filters to tiles
3. Scheduling: route items across network
Scalability Results
Raw vs. Pentium III

16-Tile Raw vs. Pentium III

- FIR: Speedup (cycles) = 12
- Filterbank: Speedup (cycles) = 16
- Radar: Speedup (cycles) = 6
- FMRadio: Speedup (cycles) = 8
- Bitonic Sort: Speedup (cycles) = 4
- FFT: Speedup (cycles) = 6
Outline

• Introduction
• StreamIt Language
• Domain-specific Optimizations
• Targeting Parallel Architectures
• Public Release
• Conclusions
StreamIt Compiler Infrastructure

- Built on Kopi Java compiler (GNU license)
  - StreamIt frontend is on MIT license
- High-level hierarchical IR for streams
  - Host of graph transformations
    - Filter fusion, filter fission
    - Synchronization removal
    - Splitjoin refactoring
    - Graph canonicalization
- Low-level “flat graph” for backends
  - Eliminates structure; point-to-point connections
- Streaming benchmark suite
Compiler Flow

StreamIt Front-End

Kopi Front-End

SIR Conversion

Graph Expansion

Raw Backend

Uniprocessor Backend

Linear Optimizations

Scheduler

Any Java Compiler

Class file

Legal Java file

Parse Tree

SIR (unexpanded)

SIR (expanded)

C code for tiles
Assembly code for switch

ANSI C code
Building on StreamIt

- StreamIt to VIRAM [Yelick et al.]
  - Automatically generate permutation instructions
- StreamBit: bit-level optimization [Bodik et al.]
- Integration with IBM Eclipse Platform
StreamIt Graphical Editor

StreamIt Component-Shortcuts
- Create Filters, Pipelines, SplitJoins, Feedback Loops, FIFOs

Juan C. Reyes
M.Eng. Thesis
StreamIt Debugging Environment

Compiler and Output Consoles

StreamIt Text Editor

General Debugging Information

StreamIt Graph Components

not shown: the StreamIt On-Line Help Manual

expanded and collapsed views of basic programmable unit

communication buffer with live data

StreamIt Graph Zoom Panel

Kimberly Kuo
M.Eng. Thesis
Outline

• Introduction
• StreamIt Language
• Domain-specific Optimizations
• Targeting Parallel Architectures
• Public Release
• Conclusions
Related Work

• **Stream languages**
  - KernelC/StreamC, Brook: augment C with data-parallel kernels
  - Cg: allow low-level programming of graphics processors
  - SISAL, functional languages: expose temporal parallelism
  - StreamIt exposes more task parallelism, easier to analyze

• **Control languages for embedded systems**
  - LUSTRE, Esterel, etc.: can verify of safety properties
  - Do not expose high-bandwidth data flow for optimization

• **Prototyping environments**
  - Ptolemy, Simulink, etc.: provide graphical abstractions
  - StreamIt has more of a compiler focus
Future Work

• Backend optimizations for linear filters
  - Template assembly code: asymptotically optimal

• Fault-tolerance on a cluster of workstations
  - Automatically recover if machine fail

• Supporting dynamic events
  - Point-to-point control messages
  - Re-initialization for parts of the stream
Conclusions

• **StreamIt**: compiler infrastructure for streams
  - Raising the level of abstraction in stream programming
  - Language design for both programmer and compiler

<table>
<thead>
<tr>
<th>Compiler Analysis</th>
<th>Language features exploited</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear Optimizations</td>
<td>- peek primitive (data reuse)</td>
</tr>
<tr>
<td></td>
<td>- atomic work function</td>
</tr>
<tr>
<td></td>
<td>- structured streams</td>
</tr>
<tr>
<td>Backend for Raw Architecture</td>
<td>- exposed communication patterns</td>
</tr>
<tr>
<td></td>
<td>- exposed parallelism</td>
</tr>
<tr>
<td></td>
<td>- static work estimate</td>
</tr>
</tbody>
</table>

• Public release: many opportunities for collaboration
  
Extra Slides
# StreamIt Language Summary

<table>
<thead>
<tr>
<th>Feature</th>
<th>Programmer Benefit</th>
<th>Compiler Benefit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Autonomous Filters</td>
<td>• Encapsulation • Automatic scheduling</td>
<td>• Local memories • Exposes parallelism • Exposes communication</td>
</tr>
<tr>
<td>Peek primitive</td>
<td>• Automatic buffer management</td>
<td>• Exposes data reuse</td>
</tr>
<tr>
<td>Structured Streams</td>
<td>• Natural syntax</td>
<td>• Exposes symmetry • Eliminate corner cases</td>
</tr>
<tr>
<td>Scripts for Graph Construction</td>
<td>• Reusable components • Easy to maintain</td>
<td>• Two-stage compilation</td>
</tr>
</tbody>
</table>
AB for any A and B?

- Linear Expansion

Original

Expanded

\[
[A] \quad \Rightarrow \quad \begin{bmatrix}
[A] \\
[A] \\
[A]
\end{bmatrix}
\]

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

[Image of a diagram showing the linear expansion process]
Backend Support for Linear Filters

• Can generate custom code for linear filters
  - Many architectures have special support for matrix mult.
  - On Raw: assembly code templates for tiles and switch
    • Substitute coefficients, peek/pop/push rates

• Preliminary result: FIR on Raw
  - StreamIt code: 15 lines
  - Manually-tuned C code: 352 lines
  - Both achieve 99% utilization of Raw FPU’s
    • Asymptotically optimal

• Current focus: integrating with general backend