Exploiting Coarse-Grained Task, Data, and Pipeline Parallelism in Stream Programs

Michael Gordon, William Thies, and Saman Amarasinghe
Massachusetts Institute of Technology

ASPLOS
October 2006
San Jose, CA

http://cag.csail.mit.edu/streamit
Multicores Are Here!
Multicores Are Here!

For uniprocessors, C was:

• Portable
• High Performance
• Composable
• Malleable
• Maintainable

Uniprocessors: C is the common machine language

# of cores


1 2 4 8 16 32 64 128 256 512
Multicores Are Here!

What is the common machine language for multicores?

# of cores

Common Machine Languages

<table>
<thead>
<tr>
<th>Uniprocessors:</th>
<th>Multicores:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Common Properties</strong></td>
<td><strong>Common Properties</strong></td>
</tr>
<tr>
<td>Single flow of control</td>
<td>Multiple flows of control</td>
</tr>
<tr>
<td>Single memory image</td>
<td>Multiple local memories</td>
</tr>
<tr>
<td><strong>Differences:</strong></td>
<td><strong>Differences:</strong></td>
</tr>
<tr>
<td>Register File</td>
<td>Number and capabilities of cores</td>
</tr>
<tr>
<td>ISA</td>
<td>Communication Model</td>
</tr>
<tr>
<td>Functional Units</td>
<td>Synchronization Model</td>
</tr>
</tbody>
</table>

von-Neumann languages represent the common properties and abstract away the differences

Need common machine language(s) for multicores
Streaming as a Common Machine Language

- Regular and repeating computation
- Independent filters with explicit communication
  - Segregated address spaces and multiple program counters
- Natural expression of Parallelism:
  - Producer / Consumer dependencies
  - Enables powerful, whole-program transformations
Types of Parallelism

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

Scatter

Gather
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
Outline

• **StreamIt language overview**

• **Mapping to multicores**
  – Baseline techniques
  – Our 3-phase solution
The StreamIt Project

• **Applications**
  – DES and Serpent [PLDI 05]
  – MPEG-2 [IPDPS 06]
  – SAR, DSP benchmarks, JPEG, …

• **Programmability**
  – StreamIt Language (CC 02)
  – Teleport Messaging (PPOPP 05)
  – Programming Environment in Eclipse (P-PHEC 05)

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

• **Architecture Specific Optimizations**
  – Compiling for Communication-Exposed Architectures (ASPLOS 02)
  – Phased Scheduling (LCTES 03)
  – Cache Aware Optimization (LCTES 05)
  – Load-Balanced Rendering (Graphics Hardware 05)
Model of Computation

- Synchronous Dataflow [Lee ‘92]
  - Graph of autonomous filters
  - Communicate via FIFO channels

- Static I/O rates
  - Compiler decides on an order of execution (schedule)
  - Static estimation of computation
float→float filter FIR (int N, float[N] weights) {

    work push 1 pop 1 peek N {
        float result = 0;

        for (int i = 0; i < N; i++) {
            result += weights[i] * peek(i);
        }

    }

    pop();
    push(result);

}
Example StreamIt Filter

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

Stateful
StreamIt Language Overview

- StreamIt is a novel language for streaming
  - Exposes parallelism and communication
  - Architecture independent
  - Modular and composable
    - Simple structures composed to create complex graphs
  - Malleable
    - Change program behavior with small modifications

![Diagram of StreamIt language constructs: filter, pipeline, splitjoin, feedback loop]
Outline

• StreamIt language overview
• Mapping to multicores
  – Baseline techniques
  – Our 3-phase solution
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 speedup over single core

• What about 4, 16, 1024, ... 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

![Good Parallelism! Too Much Synchronization!]

The chart shows the throughput normalized to single core stream. The tasks and fine-grained data are compared. Some tasks have good parallelism, while others show too much synchronization.
Outline

• StreamIt language overview

• Mapping to multicores
  – Baseline techniques
  – Our 3-phase solution
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

- Good Parallelism!
- Low Synchronization!
Simplified Vocoder

Data Parallel

Data Parallel, but too little work!

Data Parallel

Target a 4 core machine
Data Parallelize

Target a 4 core machine
Data + Task Parallel Execution

Target 4 core machine

Time

Cores

32
We Can Do Better!

Cores

Target 4 core machine

Time

Splitter

Joiner

Splitter

Splitter

Joiner

Splitter

Splitter

Joiner
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

Cores

16
Evaluation: Coarse-Grained Task + Data + Software Pipelining

<table>
<thead>
<tr>
<th>Task</th>
<th>Fine-Grained Data</th>
<th>Coarse-Grained Task + Data</th>
<th>Coarse-Grained Task + Data + Software Pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td>Throughput Normalized to Single Core Stream</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Best Parallelism! Lower Synchronization!
Generalizing to Other Multicores

• Architectural requirements:
  – Compiler controlled local memories with DMA
  – Efficient implementation of scatter/gather

• To port to other architectures, consider:
  – Local memory capacities
  – Communication to computation tradeoff

• Did not use processor-to-processor communication on Raw
Related Work

- Streaming languages:
  - Brook [Buck et al. ’04]
  - StreamC/KernelC [Kapasi ’03, Das et al. ’06]
  - Cg [Mark et al. ’03]
  - SPUR [Zhang et al. ’05]

- Streaming for Multicores:
  - Brook [Liao et al., ’06]

- Ptolemy [Lee ’95]

- Explicit parallelism:
  - OpenMP, MPI, & HPF
Conclusions

- Streaming model naturally exposes task, data, and pipeline parallelism
- This parallelism must be exploited at the correct granularity and combined correctly

<table>
<thead>
<tr>
<th>Parallelism</th>
<th>Fine-Grained Data</th>
<th>Coarse-Grained Task + Data</th>
<th>Coarse-Grained Task + Data + Software Pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Not matched</td>
<td>Good</td>
<td>Good</td>
<td>Best</td>
</tr>
<tr>
<td>Synchronization</td>
<td></td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>Not matched</td>
<td></td>
<td></td>
<td>Lowest</td>
</tr>
</tbody>
</table>

- Good speedups across varied benchmark suite
- Algorithms should be applicable across multicores