Microarchitectural Mechanisms to Exploit Value Structure in SIMT Architectures

Ji Kim, Christopher Torng, Shreesha Srinath, Derek Lockhart, and Christopher Batten

Cornell University

Motivation

- SIMT architectures exploit:
  - Control Structure (i.e. common instruction fetch/decode/issue)
  - Memory-Access Structure (i.e. memory coalescing)

**Value Structure** occurs when the same operation uses values across threads which can be represented as a compact function.

- Primary research questions:
  - How does value structure impact control and memory-access structure?
  - How can we realistically implement hardware mechanisms to exploit value structure to improve performance and energy-efficiency?
Presentation Outline

- **General-Purpose vs. Fine-Grain SIMT**
- Characterizing Value Structure
- FG-SIMT Baseline Architecture
- Compact Affine Execution
- Evaluation
Why GP-SIMT and FG-SIMT?

- Holistic approach for evaluating on different SIMT architectures

- GP-SIMT as a model for traditional SIMT architecture
  - Focus on exploiting inter-warp parallelism

- FG-SIMT as our own alternative SIMT architecture that we are building from the ground up
  - Targeting flexible, compute-focused data-parallel accelerators
  - Focus on exploiting intra-warp parallelism, area-efficiency

- Build credibility with FG-SIMT with cycle time, area, and energy analysis
__global__ void vsadd( int y[], int a )
{
    int idx = // get thread index

    y[idx] = y[idx] + a;
    if ( y[idx] > THRESHOLD )
        y[idx] = Y_MAX_VALUE;
}

• Key difference is in how kernel is launched
  • GP-SIMT: HW-managed, coarse-grain kernel launch
  • FG-SIMT: HW/SW-managed, fine-grain kernel launch
GP-SIMT Microarchitecture

- Multi-warp execution
- Single-ported register file
- Wide, unbanked L1 cache
- Integrated fetch/decode/issue
- Distinct memory space

FG-SIMT Microarchitecture

- Single warp execution
- Multi-ported register file
- Shared, banked L1 cache
- SW-programmable control processor
- Unified memory space
Presentation Outline

- General-Purpose vs. Fine-Grain SIMT
- Characterizing Value Structure
- FG-SIMT Baseline Architecture
- Compact Affine Execution
- Evaluation
Identifying Value Structure

```c
__global__ void
vsadd( int y[], int a ) {
    int idx = // get thread index
    y[idx] = y[idx] + a;
    if ( y[idx] > THRESHOLD )
        y[idx] = Y_MAX_VALUE;
}
```

<table>
<thead>
<tr>
<th>...</th>
<th>...</th>
<th>...</th>
<th>...</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>R_a</strong></td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td><strong>R_ybase</strong></td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td><strong>R_max</strong></td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td><strong>IDX</strong></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td><strong>R_yptr</strong></td>
<td>32</td>
<td>36</td>
<td>40</td>
<td>44</td>
</tr>
<tr>
<td><strong>R_y</strong></td>
<td>19</td>
<td>89</td>
<td>8</td>
<td>127</td>
</tr>
</tbody>
</table>

T0 T1 T2 T3

Affine Value Structure: \[ V(i) = b + i \times s \]
Why does value structure occur?

```c
__global__ void vsadd( int y[], int a ) {
    int idx = // get thread index
    y[idx] = y[idx] + a;
    if ( y[idx] > THRESHOLD )
        y[idx] = Y_MAX_VALUE;
}
```

• Operating on or loading constants
• Common control flow (e.g., inner loops)
• Manipulating addresses for structured memory access
How often does value structure occur?

- GP-SIMT Hardware detection, Collange et al. HPPC-2009
  - On average, 34% of register reads and 22% of register writes are affine

- GP-SIMT Software detection, Lee et al. CGO-2013
  - On average, 31% of combined register reads/writes are affine

- Our own FG-SIMT functional simulation:
  - 30-80% of register reads and 20-70% of register writes are affine
Presentation Outline

• General-Purpose vs. Fine-Grain SIMT
• Characterizing Value Structure
• FG-SIMT Baseline Architecture
• Compact Affine Execution
• Evaluation
FG-SIMT Baseline Example Execution

vsadd:

\textbf{ld.sh} \ R_a, \ M[A] \\
\textbf{ld.sh} \ R_ybase, \ M[Y] \\
\textbf{add} \ R_{yptr}, \ R_ybase, \ IDX \\
\textbf{load} \ R_y, \ M[R_{yptr}] \\
\textbf{add} \ R_y, \ R_y, \ R_a \\
\textbf{store} \ R_y, \ M[R_{yptr}] \\
\textbf{branch} \ R_y, \ \text{THRESHOLD} \\
\textbf{imm} \ R_{\text{max}}, \ Y_{\text{MAX VALUE}} \\
\textbf{store} \ R_{\text{max}}, \ M[R_{yptr}] \\
\textbf{stop}
Presentation Outline

- General-Purpose vs. Fine-Grain SIMT
- Characterizing Value Structure
- FG-SIMT Baseline Architecture
- **Compact Affine Execution**
- Evaluation
Tracking Value Structure

- Store affine values in **Affine SIMT Register File (ASRF)**

- ASRF encodes affine values as base and stride pair with uniform/affine tags

- Registers are tagged as affine when:
  - Shared loads (e.g., `ld.param`, `ld.sh`)
  - Thread index (e.g., `tid.x`, `IDX`)
  - Result of affine arithmetic
Exploiting Value Structure

- Affine arithmetic
- Affine memory operations

\[ V_0(i) = b_0 + i \times s_0 \quad V_1(i) = b_1 + i \times s_1 \]
\[ V_0(i) + V_1(i) = (b_0 + b_1) + i \times (s_0 + s_1) \]

- addiu
- lui
- addu
- subu
- sll/sllv
- srl/srlv
- sra/srav
- mul

vsadd:
- \texttt{ld.sh} \quad R_a, M[A]
- \texttt{ld.sh} \quad R_ybase, M[Y]
- \texttt{add} \quad R_yptr, R_ybase, IDX
- \texttt{load} \quad R_y, M[R_yptr]
- \texttt{add} \quad R_y, R_y, R_a
- \texttt{store} \quad R_y, M[R_yptr]
- \texttt{branch} \quad R_y, \text{THRESHOLD}
- \texttt{imm} \quad R_{\text{max}}, \text{Y_MAX_VALUE}
- \texttt{store} \quad R_{\text{max}}, M[R_yptr]
- \texttt{stop}
Exploiting Value Structure

- Affine arithmetic
- Affine memory operations
- Affine branches

\[ V_0(i) = b_0 + i \times s_0 \quad V_1(i) = b_1 + i \times s_1 \]

\[ V_0(i) + V_1(i) = (b_0 + b_1) + i \times (s_0 + s_1) \]

- addiu
- lui
- addu
- subu
- sll/sllv
- srl/srlv
- sra/srav
- mul
- lw/lh/lb
- sw/sh/sb
- beq/bne
- blez/bgez
- bltz/bgtz
- vsadd:
  - ld.sh \( R_a, M[A] \)
  - ld.sh \( R_y\text{base}, M[Y] \)
  - add \( R_y\text{ptr}, R_y\text{base}, \text{IDX} \)
  - load \( R_y, M[R_y\text{ptr}] \)
  - add \( R_y, R_y, R_a \)
  - store \( R_y, M[R_y\text{ptr}] \)
  - branch \( R_a, \text{THRESHOLD} \)
  - imm \( R_{\text{max}}, \text{Y_MAX_VALUE} \)
  - store \( R_{\text{max}}, M[R_y\text{ptr}] \)
  - stop

Consider the common case of comparing uniform registers
Affine Arithmetic

Add parallel affine datapath for base/stride computation

vsadd:

- `ld.sh`: R_a, M[A]
- `ld.sh`: R_ybase, M[Y]
- `add`: R_yptr, R_ybase, IDX
- `load`: R_y, M[R_yptr]
- `add`: R_y, R_y, R_a
- `store`: R_y, M[R_yptr]
- `branch`: R_y, THRESHOLD
- `imm`: R_max, Y_MAX_VALUE
- `store`: R_max, M[R_yptr]
- `stop`
Affine Arithmetic

vsadd:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld.sh R_a, M[A]</td>
<td>Load immediate from memory</td>
</tr>
<tr>
<td>ld.sh R_ybase, M[Y]</td>
<td>Load base address from memory</td>
</tr>
<tr>
<td>add R_yptr, R_ybase, IDX</td>
<td>Add pointer and base address</td>
</tr>
<tr>
<td>load R_y, M[R_yptr]</td>
<td>Load value from memory</td>
</tr>
<tr>
<td>add R_y, R_y, R_a</td>
<td>Add values</td>
</tr>
<tr>
<td>store R_y, M[R_yptr]</td>
<td>Store value to memory</td>
</tr>
<tr>
<td>branch R_y, THRESHOLD</td>
<td>Branch on condition</td>
</tr>
<tr>
<td>imm R_max, Y_MAX_VALUE</td>
<td>Immediate value</td>
</tr>
<tr>
<td>store R_max, M[R_yptr]</td>
<td>Store immediate value to memory</td>
</tr>
<tr>
<td>stop</td>
<td>End program</td>
</tr>
</tbody>
</table>

The diagram illustrates the Affine Arithmetic execution flow, showing the interaction between the Control Processor (CP), SIMT Issue Unit (SIU), and SIMT Memory Unit, along with data memory and instruction memory interactions.
Affine Memory Operations

vsadd:
- \texttt{ld.sh} \hspace{1pt} R_a, M[A]
- \texttt{ld.sh} \hspace{1pt} R_ybase, M[Y]
- \texttt{add} \hspace{1pt} R_yptr, R_ybase, IDX
- \texttt{load} \hspace{1pt} R_y, M[R_yptr]
- \texttt{add} \hspace{1pt} R_y, R_y, R_a
- \texttt{store} \hspace{1pt} R_y, M[R_yptr]
- \texttt{branch} \hspace{1pt} R_y, \texttt{THRESHOLD}
- \texttt{imm} \hspace{1pt} R_{\text{max}}, Y_{\text{MAX\_VALUE}}
- \texttt{store} \hspace{1pt} R_{\text{max}}, M[R_yptr]
- \texttt{stop}
Affine Branches

vsadd:

\[
\text{ld.sh } R_a, M[A] \\
\text{ld.sh } R_ybase, M[Y] \\
\text{add } R_{y.ptr}, R_{y.base}, IDX \\
\text{load } R_y, M[R_{y.ptr}] \\
\text{add } R_y, R_y, R_a \\
\text{store } R_y, M[R_{y.ptr}] \\
\text{branch } R_a, \text{THRESHOLD} \\
\text{imm } R_{max}, Y_{MAX\_VALUE} \\
\text{store } R_{max}, M[R_{y.ptr}] \\
\text{stop}
\]
Motivation
GP-SIMT vs. FG-SIMT
Value Structure
FG-SIMT Baseline
Compact Affine Execution
Evaluation

CP
SIU
Lane0

CP
SIU
Lane0
Three Types of Affine Expansions

- **Affine Source Expansion**
  - When generic instructions read affine operands
  - Expand out source operands, then execute on SIMT lanes
  - No performance overhead

- **Affine Destination Expansion**
  - When affine instructions execute after divergence
  - Execute compactly on CP, then expand result on SIMT lanes
  - No performance overhead

- **Affine Pre-Destination Expansion**
  - When affine register is overwritten after divergence
  - Expand destination first, then execute on SIMT lanes
  - Adds performance overhead

See paper for more details
Compact Affine Execution on GP-SIMT

- Affine arithmetic avoids time spent in the operand collection, execution, and writeback stages.

- Affine memory operations and branches reduce the pressure on the operand collector.

- All mechanisms still improve energy-efficiency.
Presentation Outline

- General-Purpose vs. Fine-Grain SIMT
- Characterizing Value Structure
- FG-SIMT Baseline Architecture
- Compact Affine Execution
- Evaluation
Methodology

• GP-SIMT modeled in GPGPU-Sim 3.0 with PTX front-end

• FG-SIMT modeled in Verilog RTL
  • Area, cycle time, and energy results obtained using Synopsys DesignCompiler, IC Compiler, and PrimeTime PX
  • TSMC 40nm standard cell library
  • Cycle time is 3.1ns with critical path through memory system
  • 5% area overhead for adding compact affine execution

• Benchmarks from Parboil, Rodinia, and in-house applications
FG-SIMT Detailed Microarchitecture

- Eight SIMT lanes
- Dynamic reconvergence
- Five vector functional units with support for chaining
- Multi-ported banked regfile with support for executing 32 threads at a time
- Shared load cache for kernel input parameters
- Memory coalescing to dynamically create wide accesses
Motivation

GP-SIMT vs. FG-SIMT

Value Structure

FG-SIMT Baseline

Compact Affine Execution

Evaluation

FG-SIMT Performance Results

moticore

gsimt

+arithmetic

+arith/branch

+arith/branch/memop

Speedup

cmult

mfilt

bsearch

viterbi

rsort

dither

strsearch

rgb2cmyk

conv

kmeans

bfs

sgemm

bilat
FG-SIMT Energy vs. Performance Results

Motivation

GP-SIMT vs. FG-SIMT Value Structure FG-SIMT Baseline Compact Affine Execution Evaluation
FG-SIMT Energy vs. Performance Results

- **viterbi**
  - 54% saved within register file
  - 29% saved within functional units
  - 34% saved within memory system
FG-SIMT Energy vs. Performance Results

Motivation
GP-SIMT vs. FG-SIMT
Value Structure
FG-SIMT Baseline
Compact Affine Execution
Evaluation

- strsearch
  - 115 to 130 uJ per task
  - 89% of 15 uJ due to expansion units
Take-Away Points

• A significant amount of value structure exists in common SIMT workloads and is often overlooked

• Compact affine execution exploits value structure in arithmetic, branch, and memory instructions to improve performance and energy-efficiency

• FG-SIMT is a promising architectural paradigm for compute-focused, area-efficient data-parallel accelerators