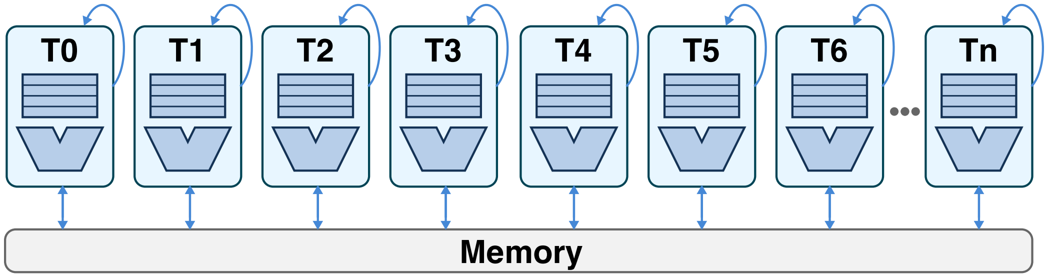
Microarchitectural Mechanisms to Exploit Value Structure in SIMT Architectures

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

Cornell University

IEEE/ACM International Symposium on Computer Architecture 2013 (ISCA-40)

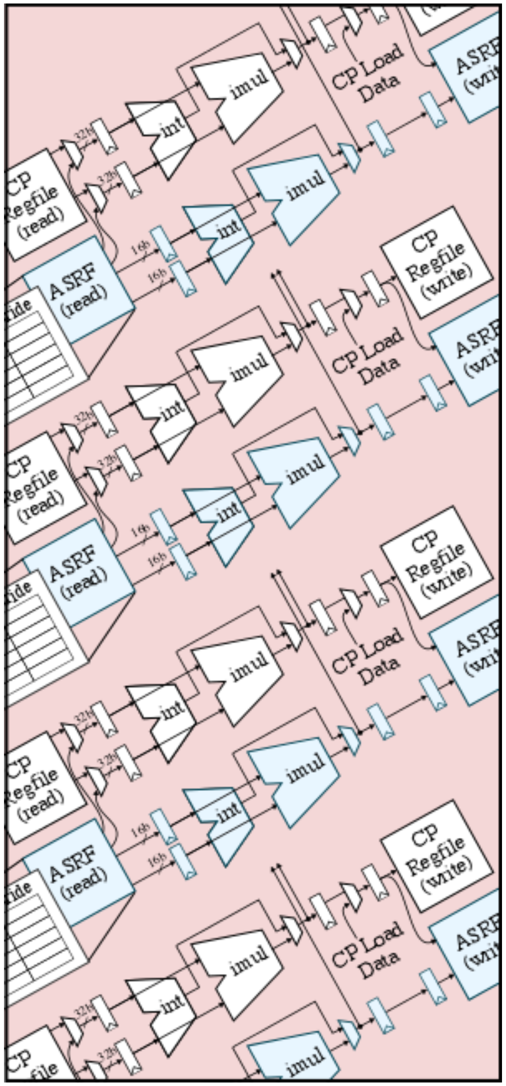
Motivation



- 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.

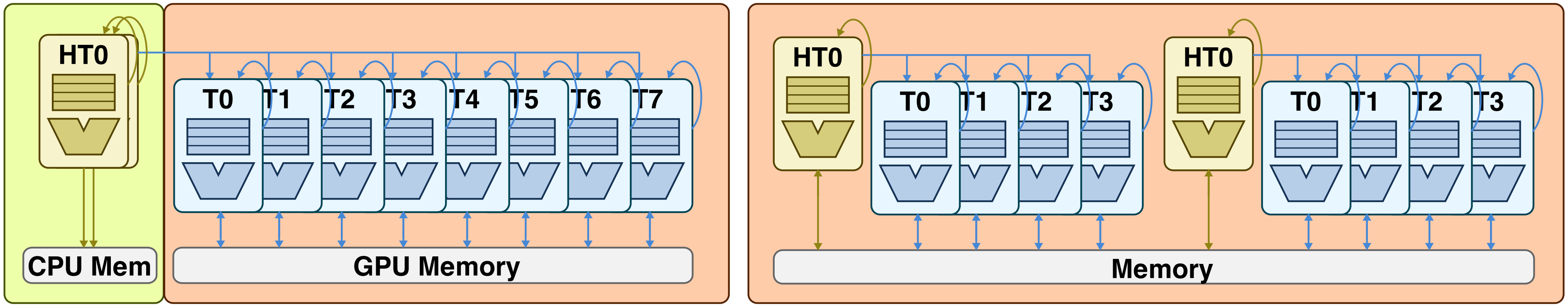
- Primary research questions:
  - How does value structure impact control and memory-access structure?
  - How can we realistically implement hardware mechanisms to exploit value structure to improve performance and energy-efficiency?



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?



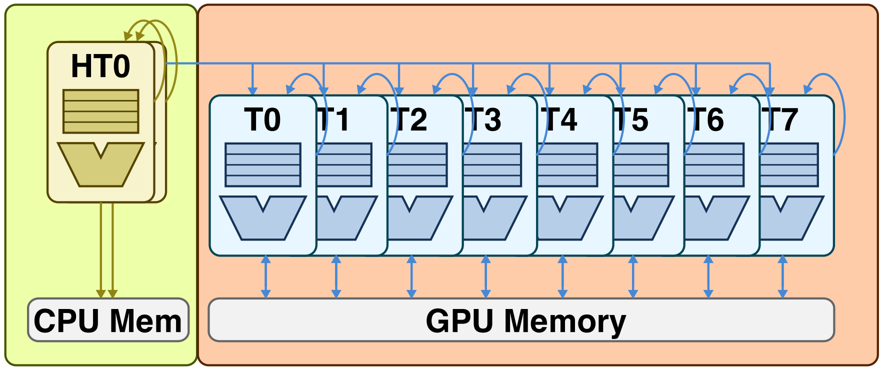
- Holistic approach for evaluating on different SIMT architectures

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

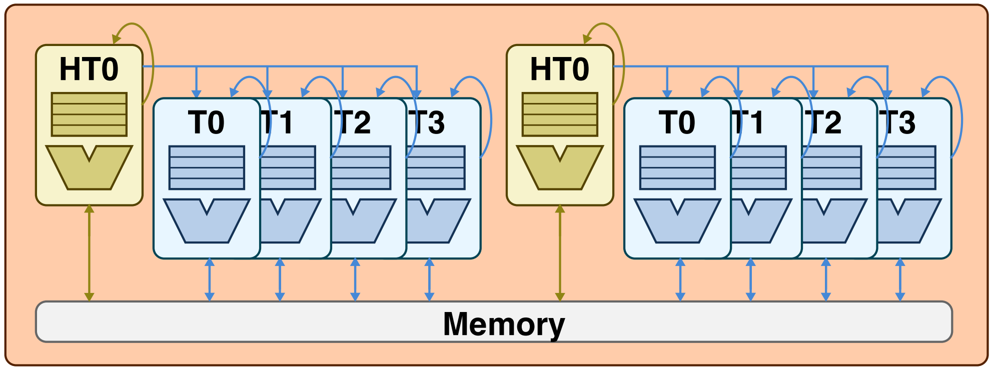
- FG-SIMT as our own alternative SIMT architecture that we are building from the ground up
    - Targeting flexible, compute-focused data-parallel accelerators
    - Focus on exploiting intra-warp parallelism, area-efficiency

- Build credibility with FG-SIMT with cycle time, area, and energy analysis

GP-SIMT Programming Model



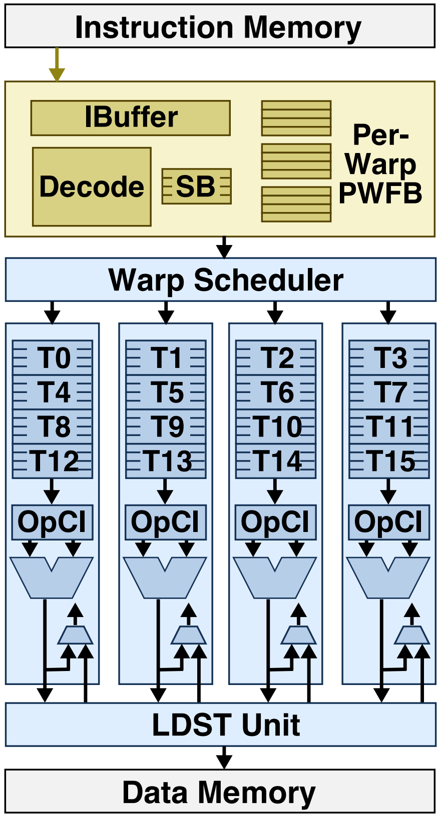
FG-SIMT Programming Model



```
__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;
}
```

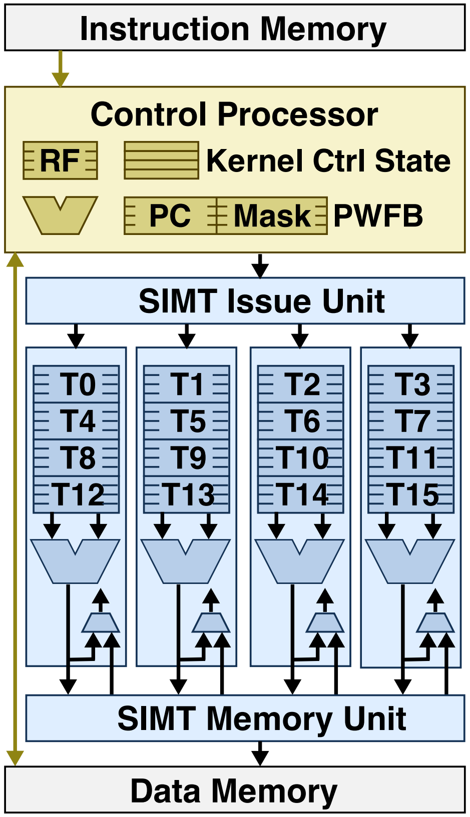
- Key difference is in how kernel is launched
  - GP-SIMT: HW-managed, coarse-grain kernel launch
  - FG-SIMT: HW/SW-managed, fine-grain kernel launch

GP-SIMT Microarchitecture

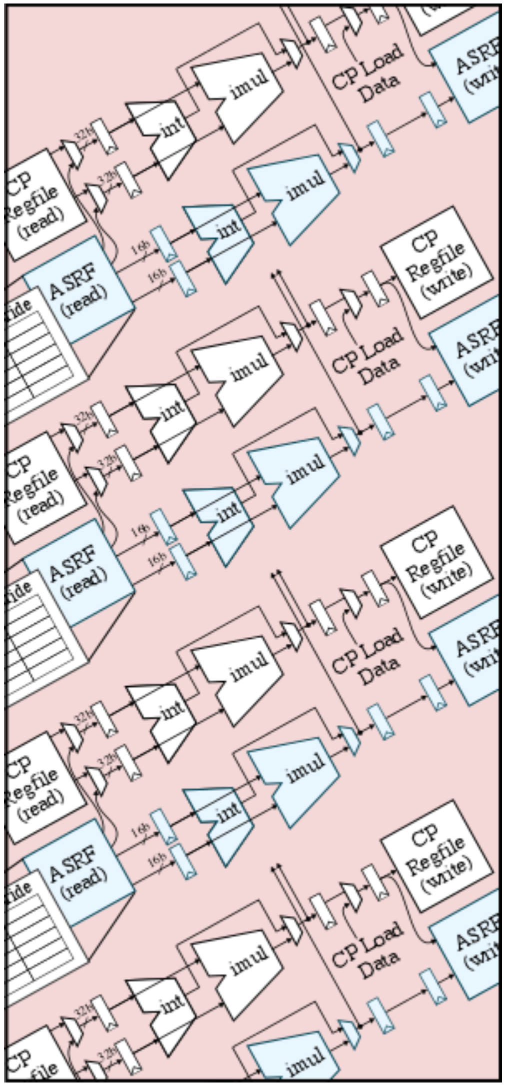


- Multi-warp execution
- Single-ported register file
- Wide, unbanked L1 cache
- Integrated fetch/decode/issue
- Distinct memory space

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

```
__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;
}
```

```
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
```

	T0	T1	T2	T3
R_a	2	2	2	2
R_ybase	32	32	32	32
R_max	40	40	40	40
IDX	0	1	2	3
R_yptr	32	36	40	44
R_y	19	89	8	127

Affine Value Structure: $V(i) = b + i \times s$

Why does value structure occur?

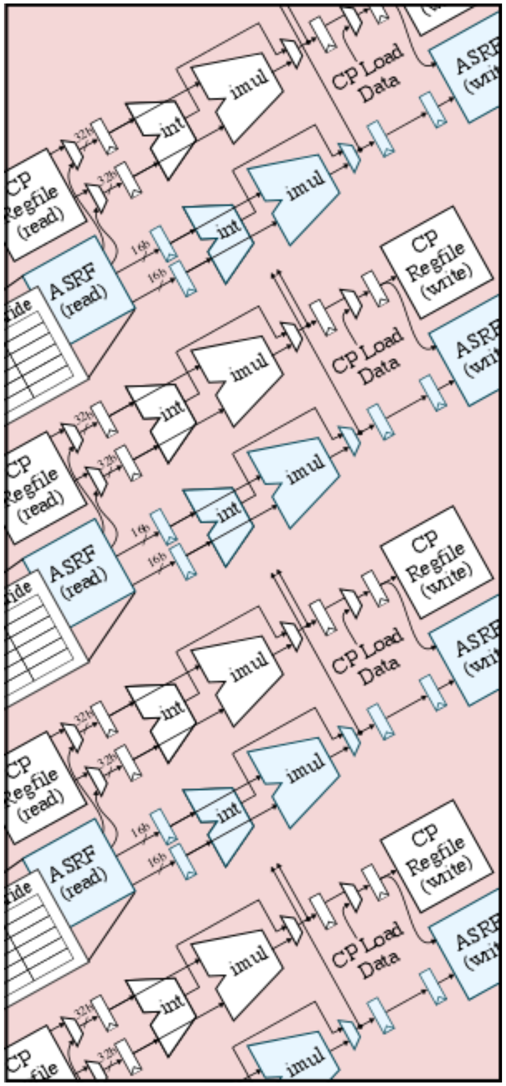
```
__global__ void
vsadd( int y[], int a ) {
  int idx = // get thread index

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

```
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
```

- 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

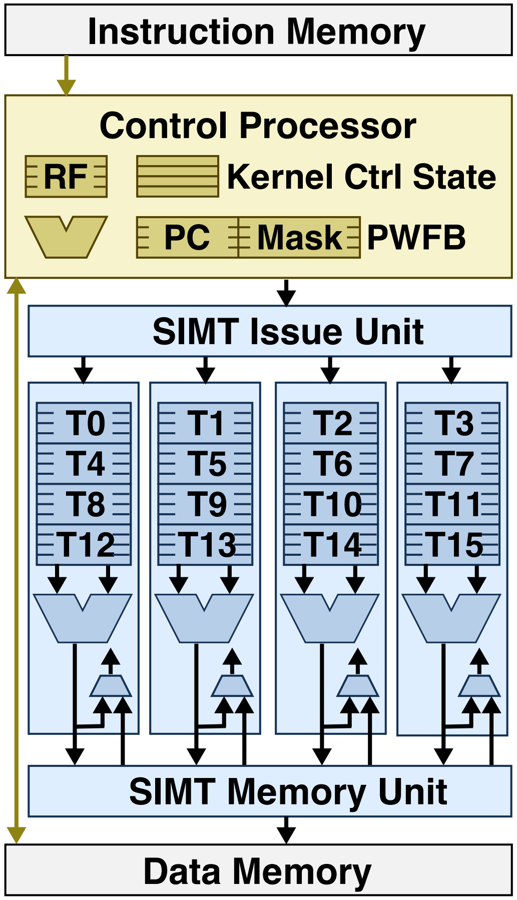


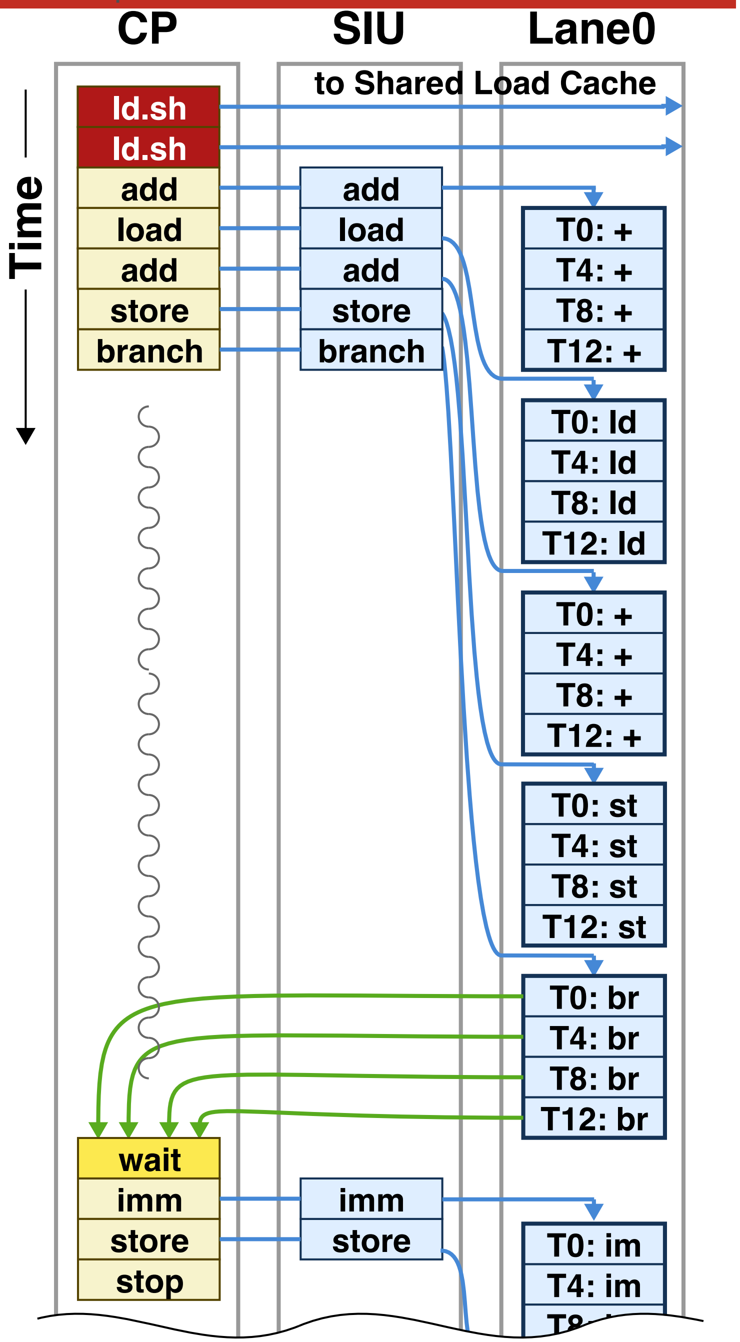
Presentation Outline

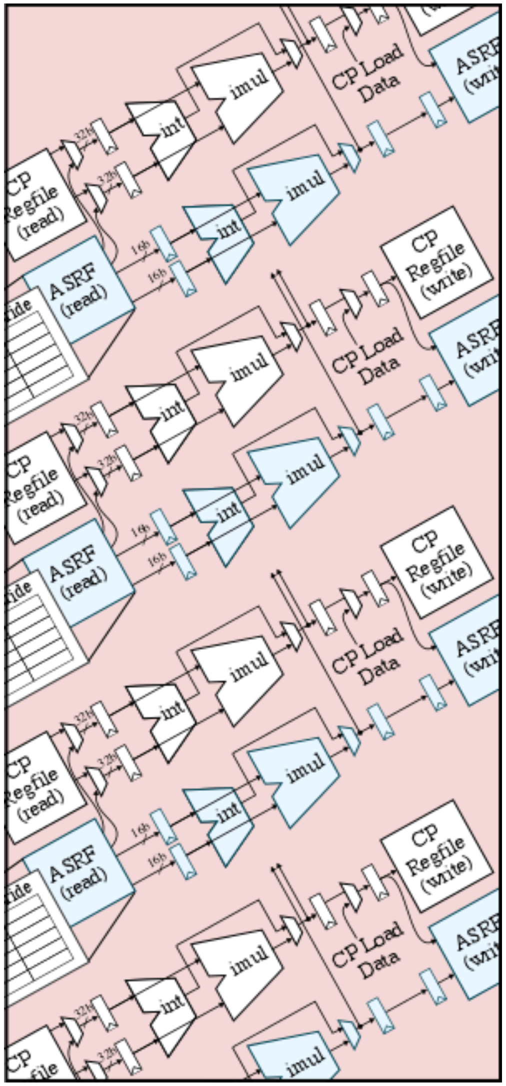
- General-Purpose vs. Fine-Grain SIMT
- Characterizing Value Structure
- **FG-SIMT Baseline Architecture**
- Compact Affine Execution
- Evaluation

FG-SIMT Baseline Example Execution

```
vsadd:
 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
```





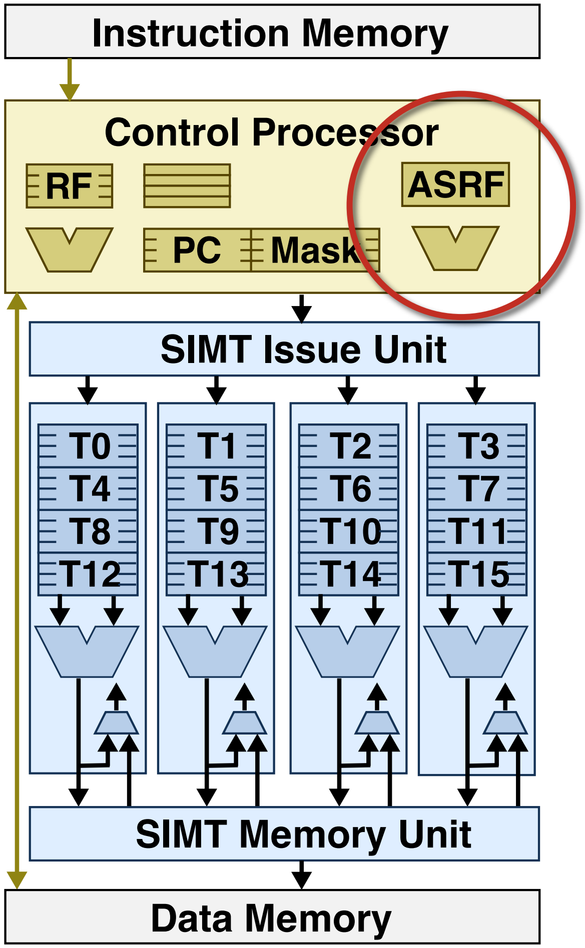


Presentation Outline

- 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`

- `lw/lh/lb`
- `sw/sh/sb`

```
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
```

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

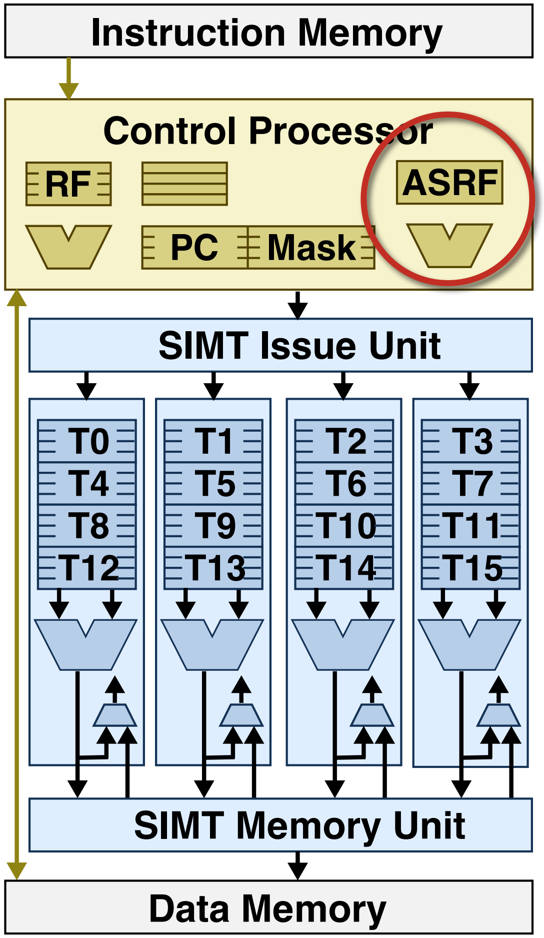
```
vsadd:
  ld.sh  R_a, M[A]
  ld.sh  R_ybase, M[Y]
  add    R_yptr, R_ybase, IDX
  load   R_y, M[R_yptr]
  add    R_y, R_y, R_a
  store  R_y, M[R_yptr]
  branch R_a, THRESHOLD
  imm    R_max, Y_MAX_VALUE
  store  R_max, M[R_yptr]
  stop
```

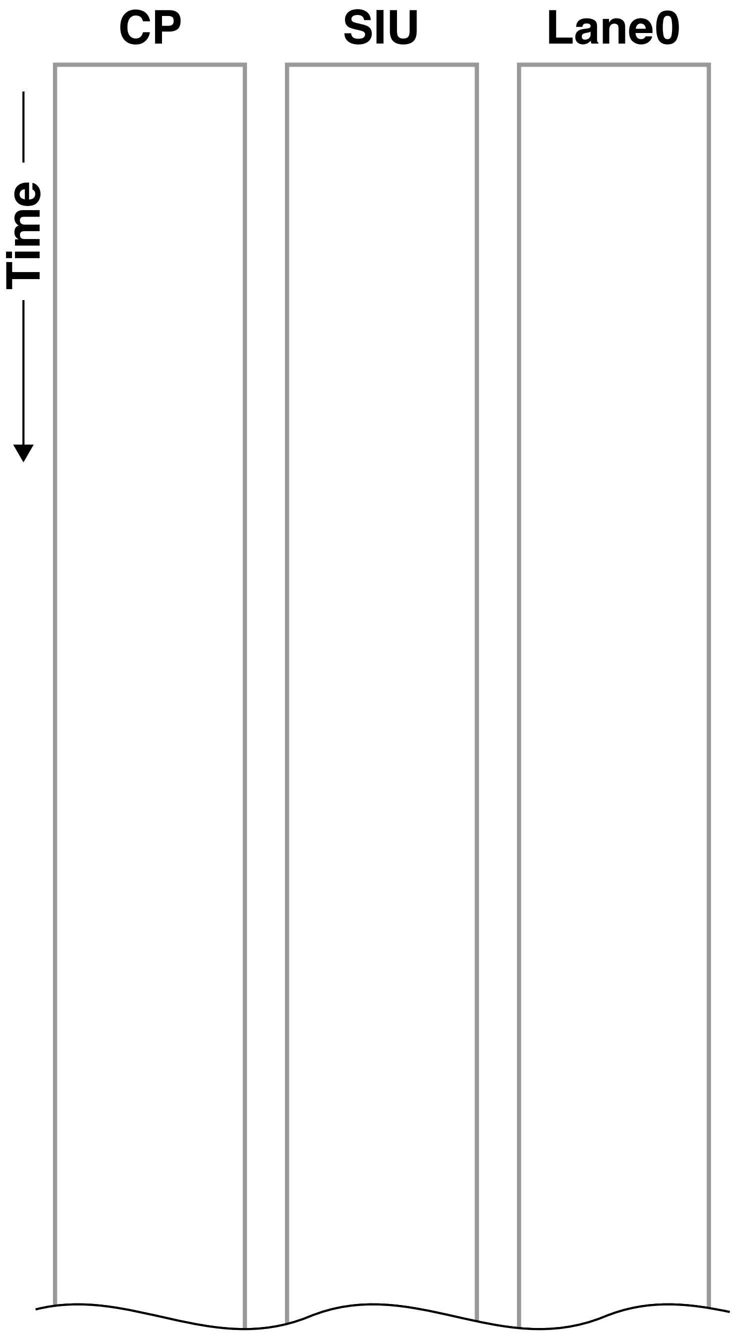
Consider the common case of comparing uniform registers

Affine Arithmetic

```
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
```

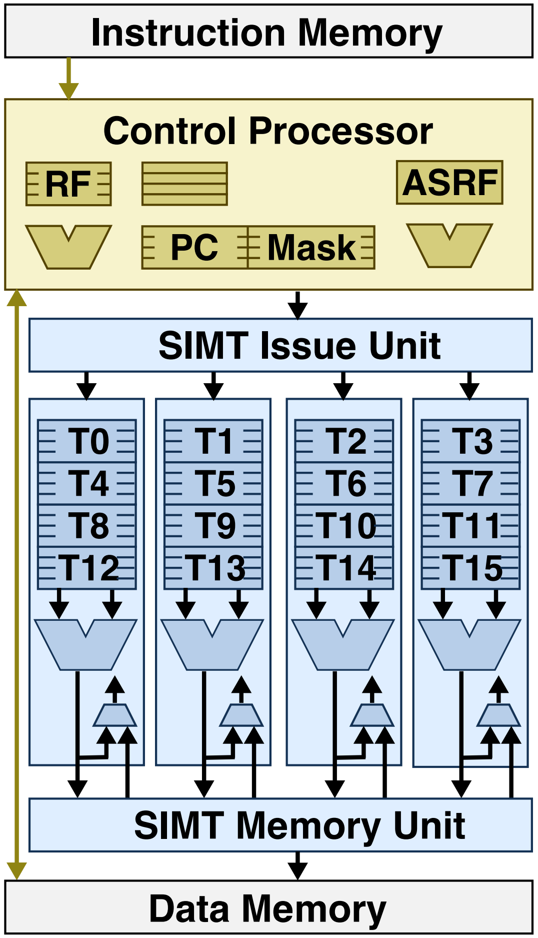
Add parallel affine datapath for base/stride computation

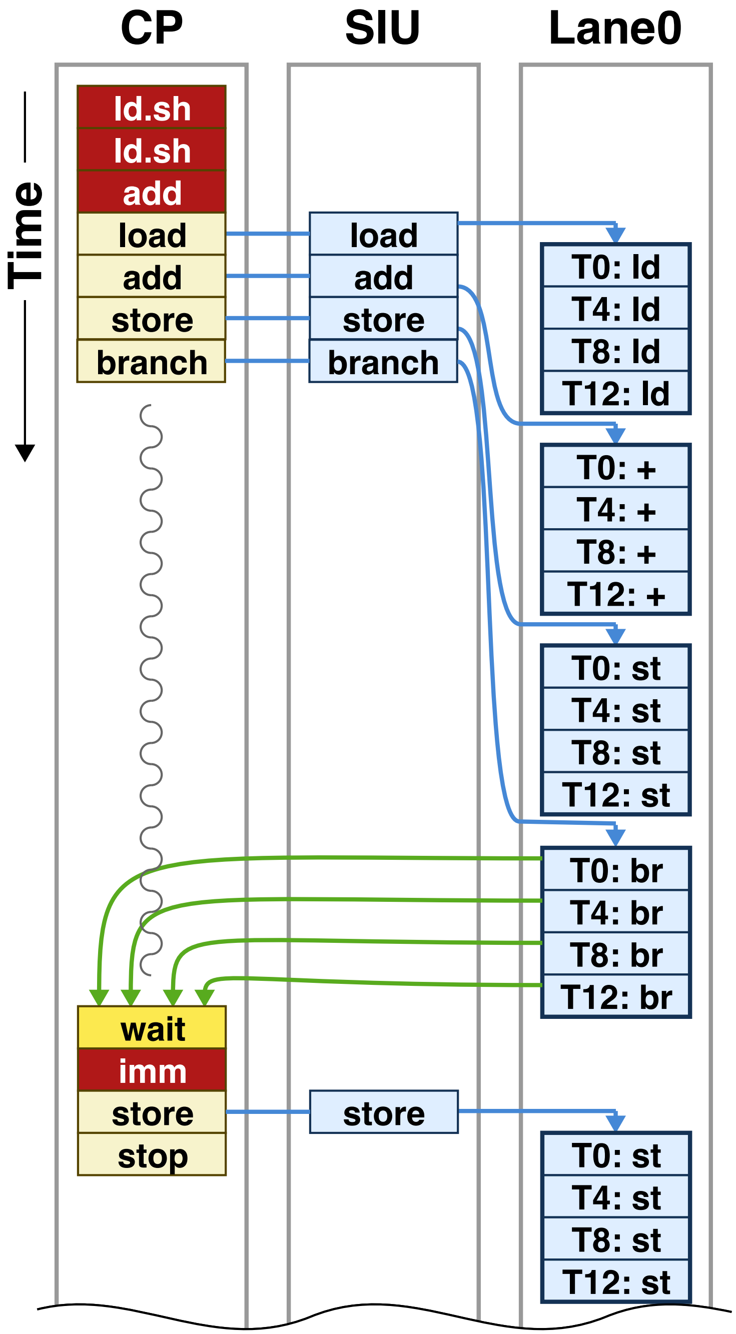




Affine Arithmetic

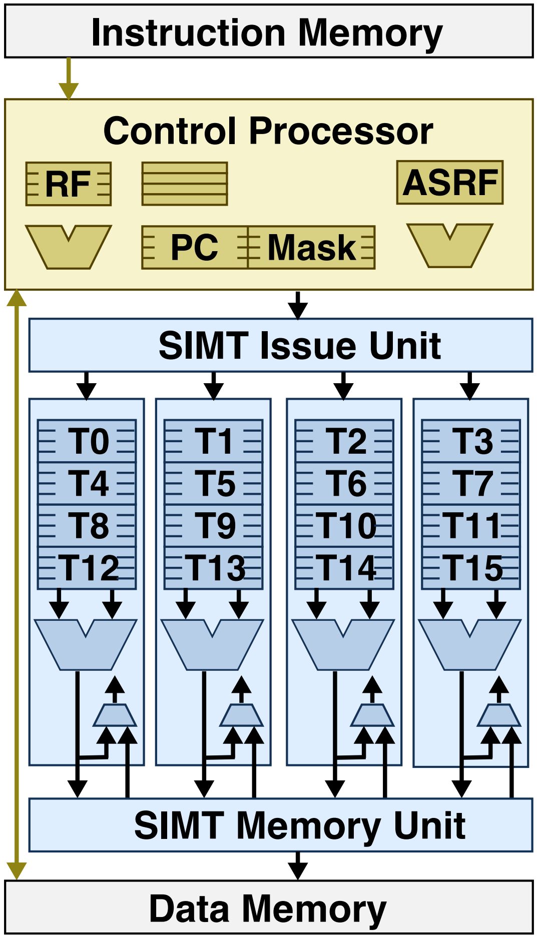
```
vsadd:
 ld.sh   R_a, M[A]
 ld.sh   R_ybase, M[Y]
 add     R_yptr, R_ybase, IDX
 load    R_y, M[R_yptr]
 add     R_y, R_y, R_a
 store   R_y, M[R_yptr]
 branch  R_y, THRESHOLD
 imm     R_max, Y_MAX_VALUE
 store   R_max, M[R_yptr]
 stop
```

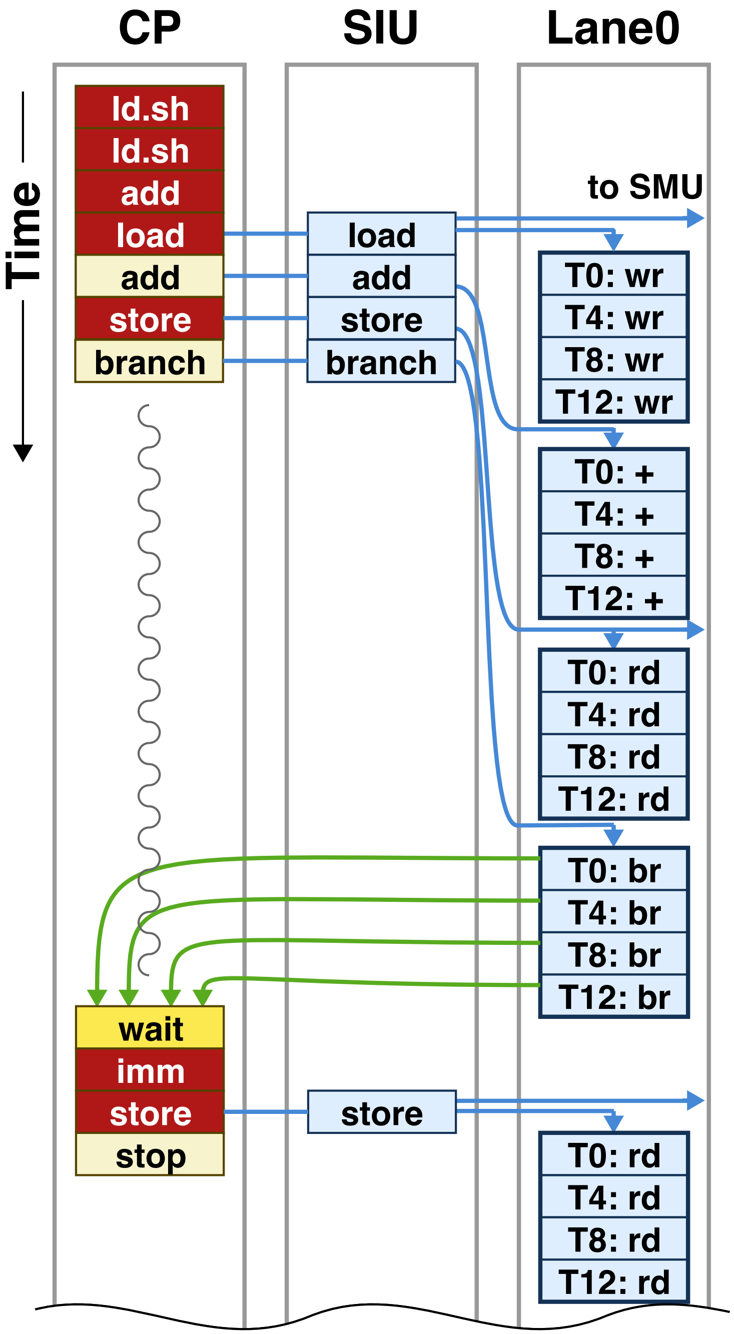




Affine Memory Operations

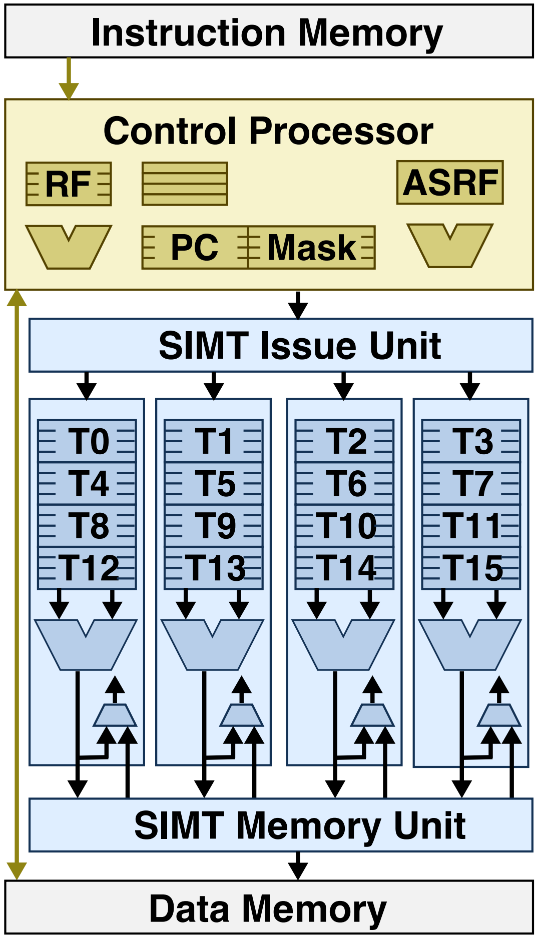
```
vsadd:
 ld.sh  R_a, M[A]
 ld.sh  R_ybase, M[Y]
 add    R_yptr, R_ybase, IDX
 load   R_y, M[R_yptr]
 add    R_y, R_y, R_a
 store  R_y, M[R_yptr]
 branch R_y, THRESHOLD
 imm    R_max, Y_MAX_VALUE
 store  R_max, M[R_yptr]
 stop
```

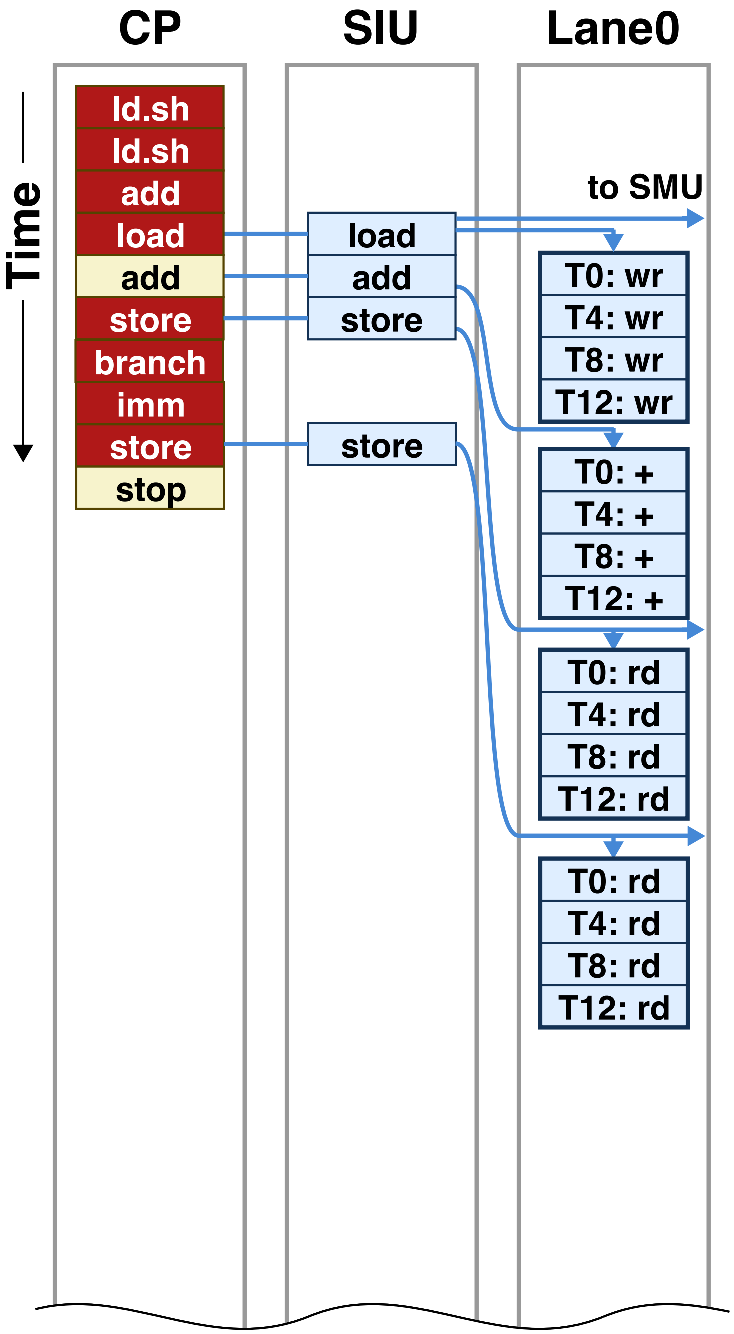


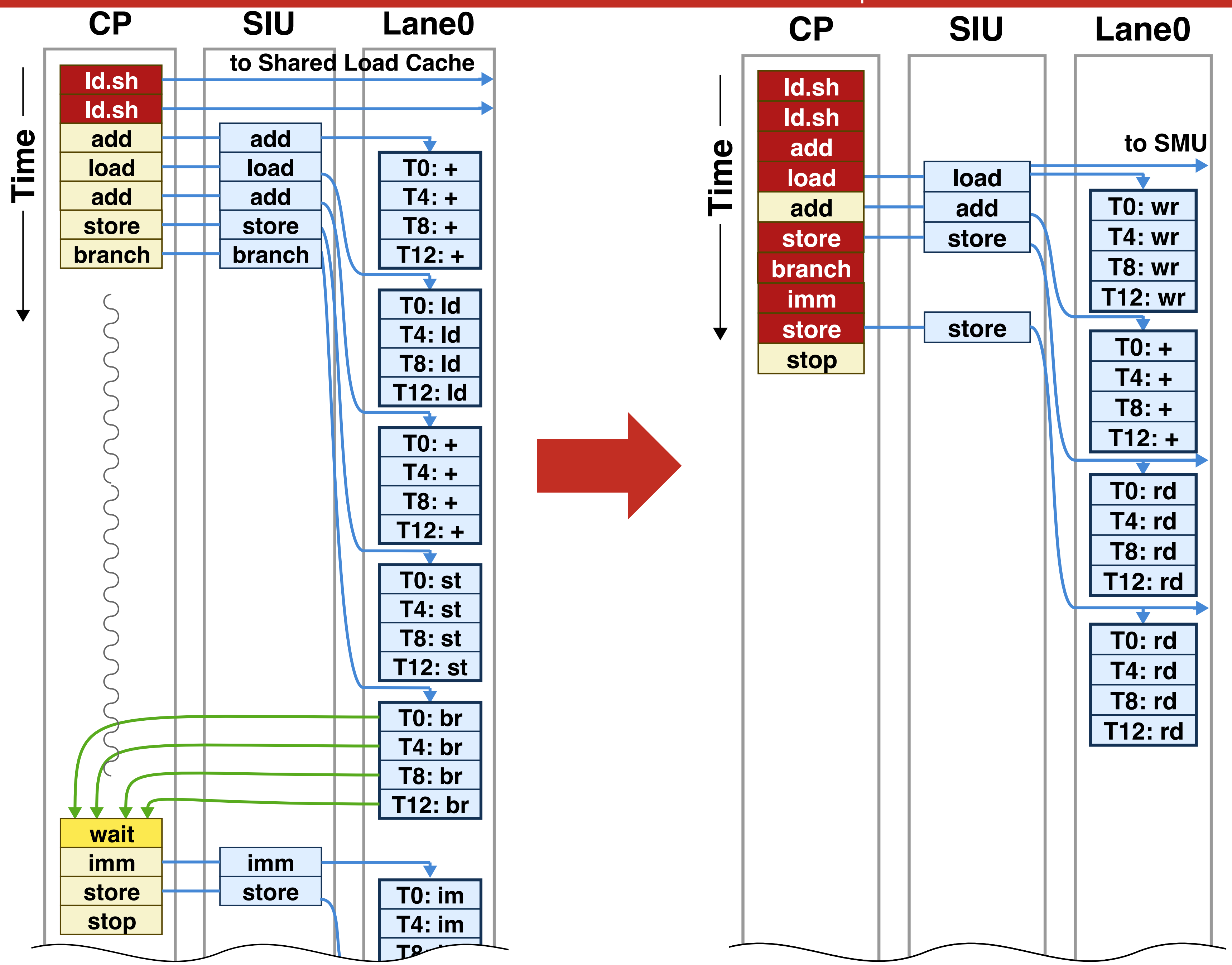


Affine Branches

```
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_a, THRESHOLD
 imm     R_max, Y_MAX_VALUE
 store   R_max, M[R_yptr]
 stop
```






CP
SIU
Lane0
to Shared Load Cache
Time
ld.sh
ld.sh
add
load
add
store
branch
add
load
add
store
branch
T0: +
T4: +
T8: +
T12: +
T0: ld
T4: ld
T8: ld
T12: ld
T0: +
T4: +
T8: +
T12: +
T0: st
T4: st
T8: st
T12: st
T0: br
T4: br
T8: br
T12: br
wait
imm
store
stop
imm
store
T0: im
T4: im
T8:
CP
SIU
Lane0
Time
ld.sh
ld.sh
add
load
add
store
branch
imm
store
stop
to SMU
load
add
store
store
T0: wr
T4: wr
T8: wr
T12: wr
T0: +
T4: +
T8: +
T12: +
T0: rd
T4: rd
T8: rd
T12: rd
T0: rd
T4: rd
T8: rd
T12: rd

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

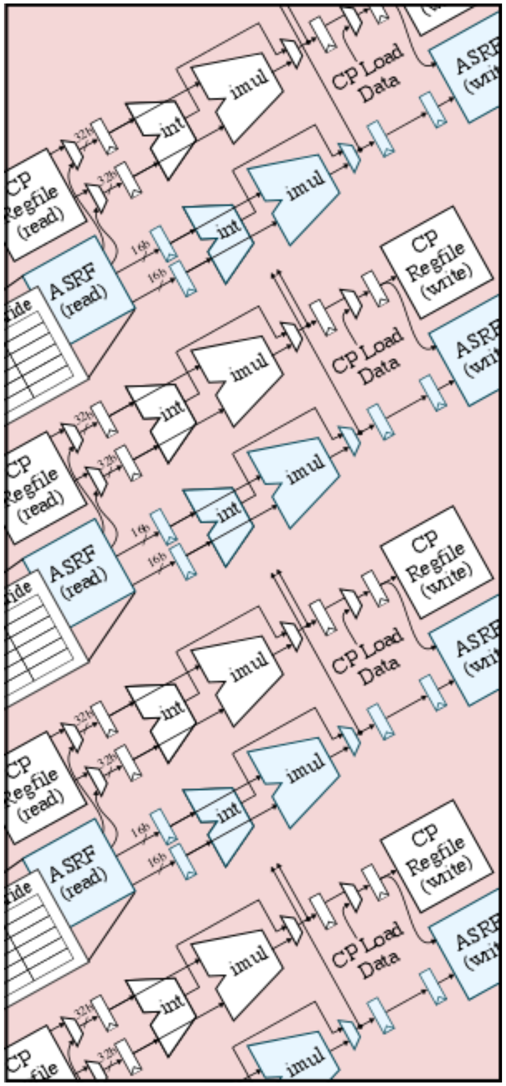
See paper for more details

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



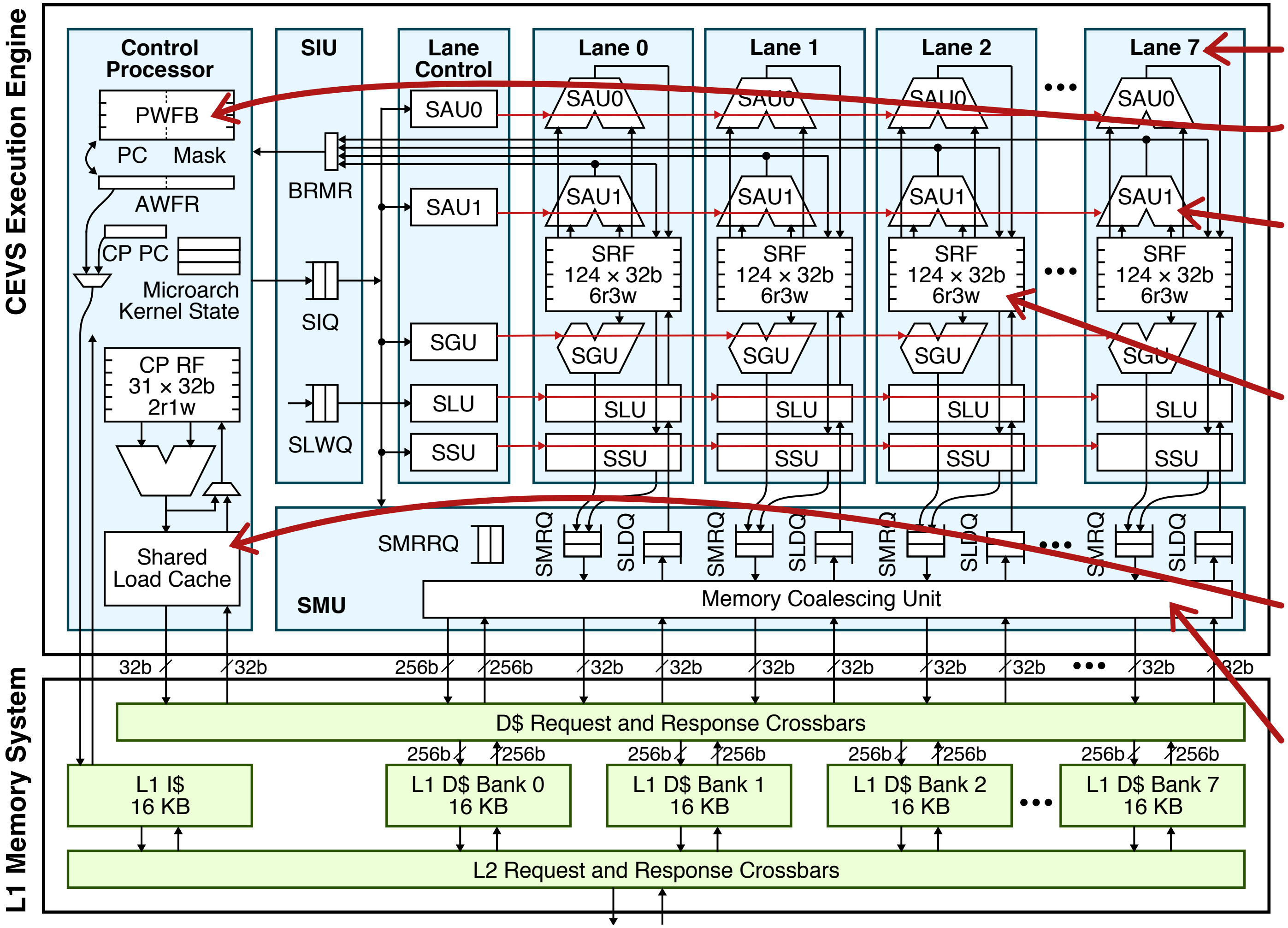
Presentation Outline

- 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

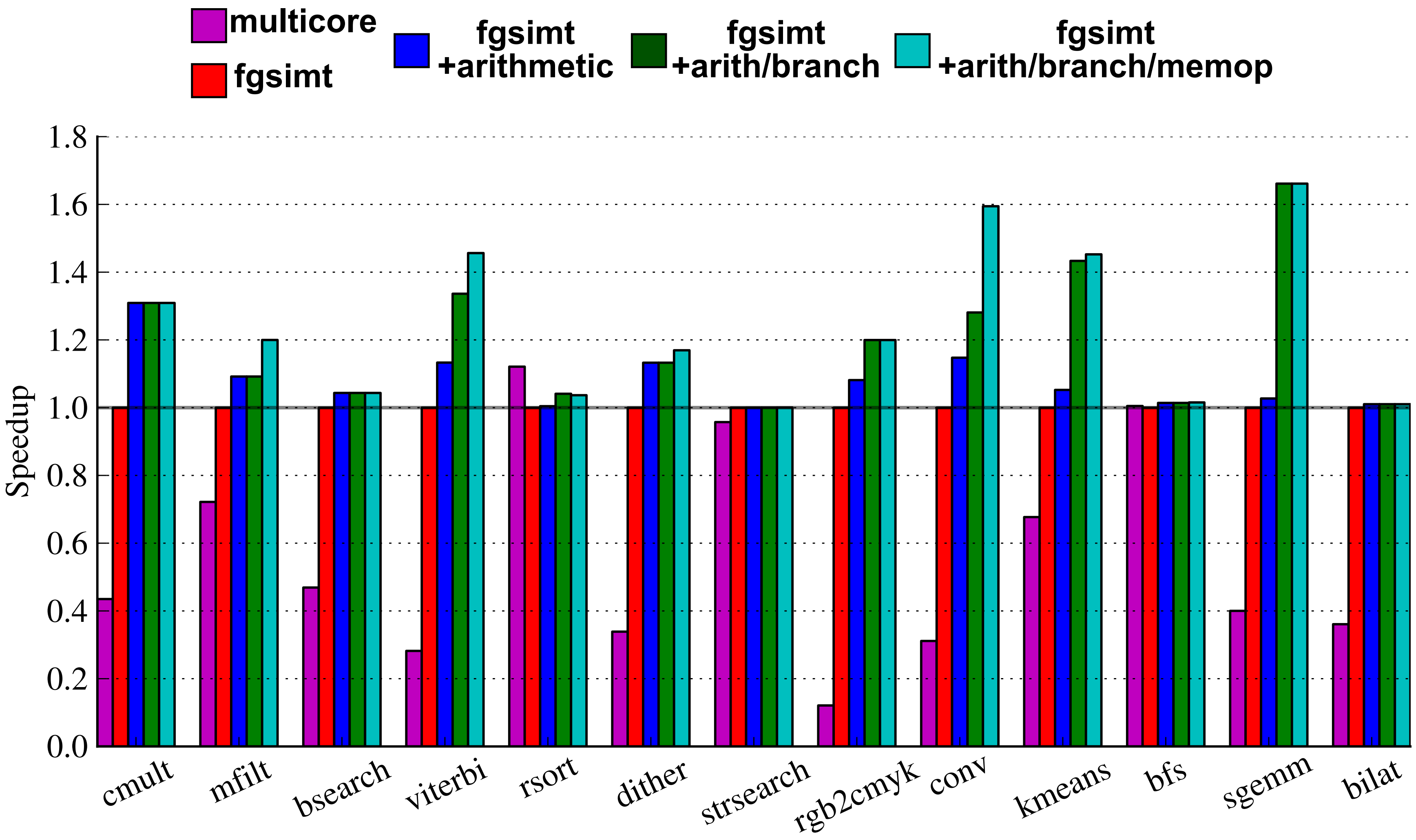
Five vector functional units with support for chaining

Multi-ported banked regfile with support for executing 32 threads at a time

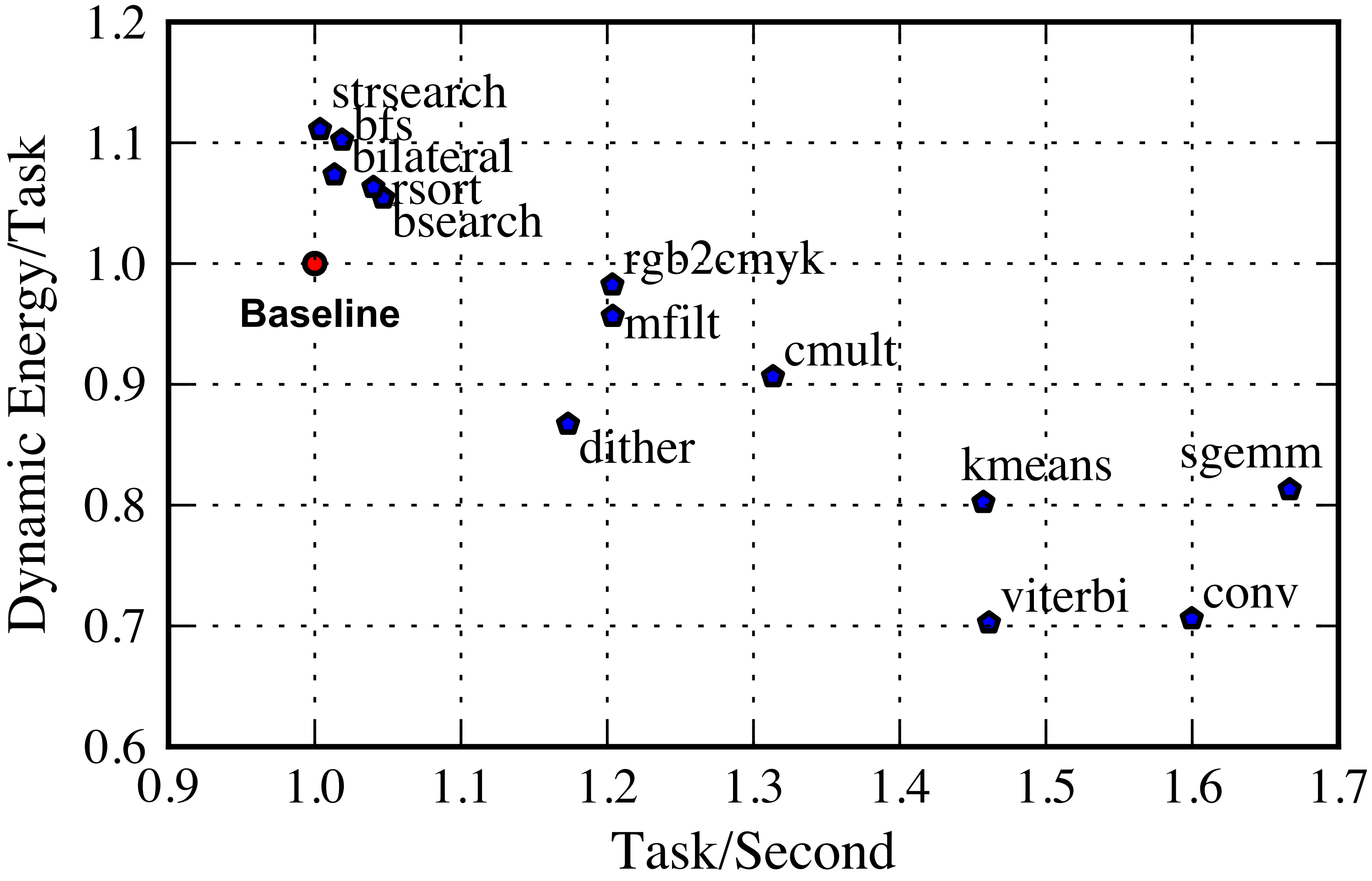
Shared load cache for kernel input parameters

Memory coalescing to dynamically create wide accesses

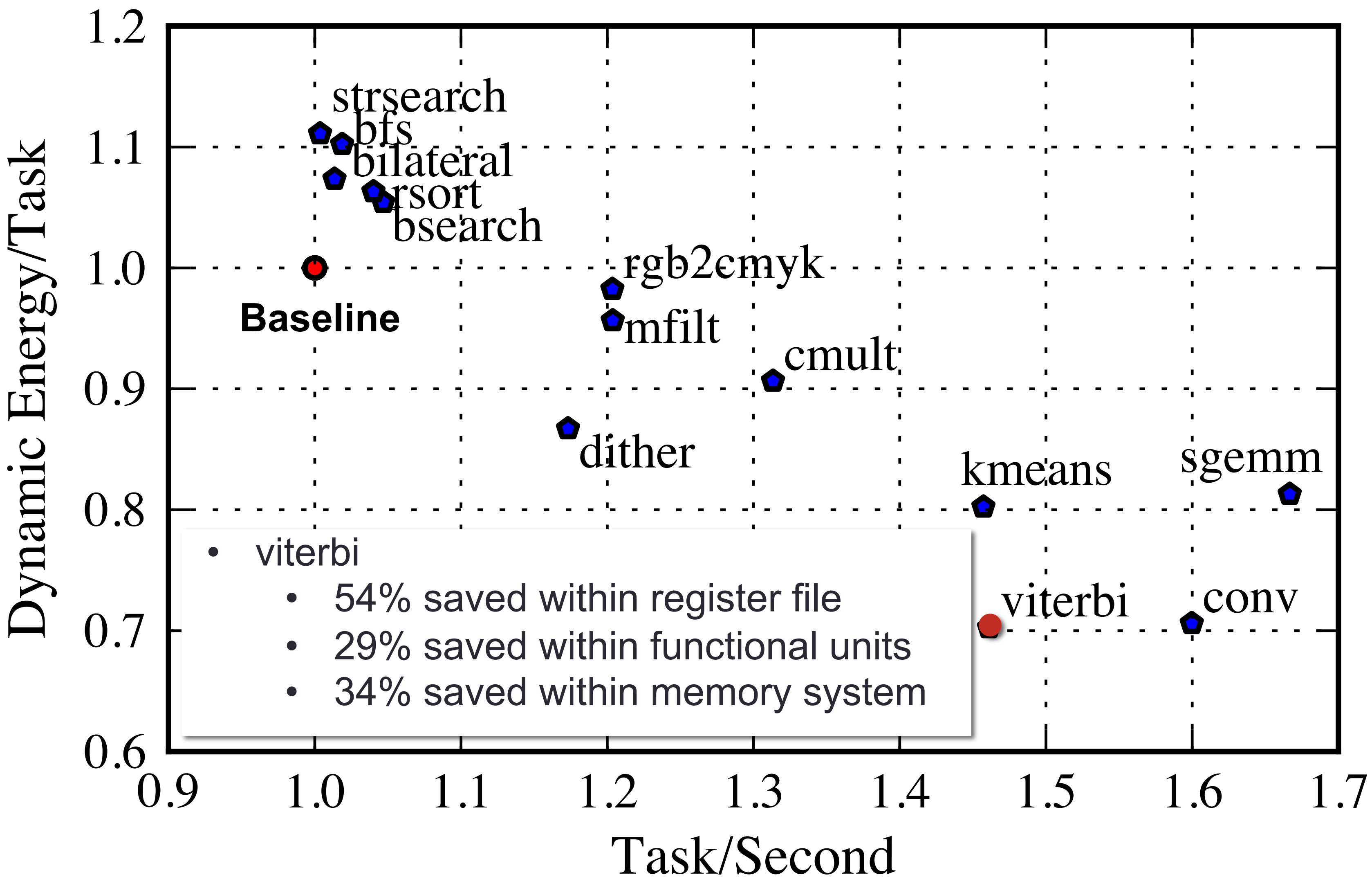
FG-SIMT Performance Results



FG-SIMT Energy vs. Performance Results

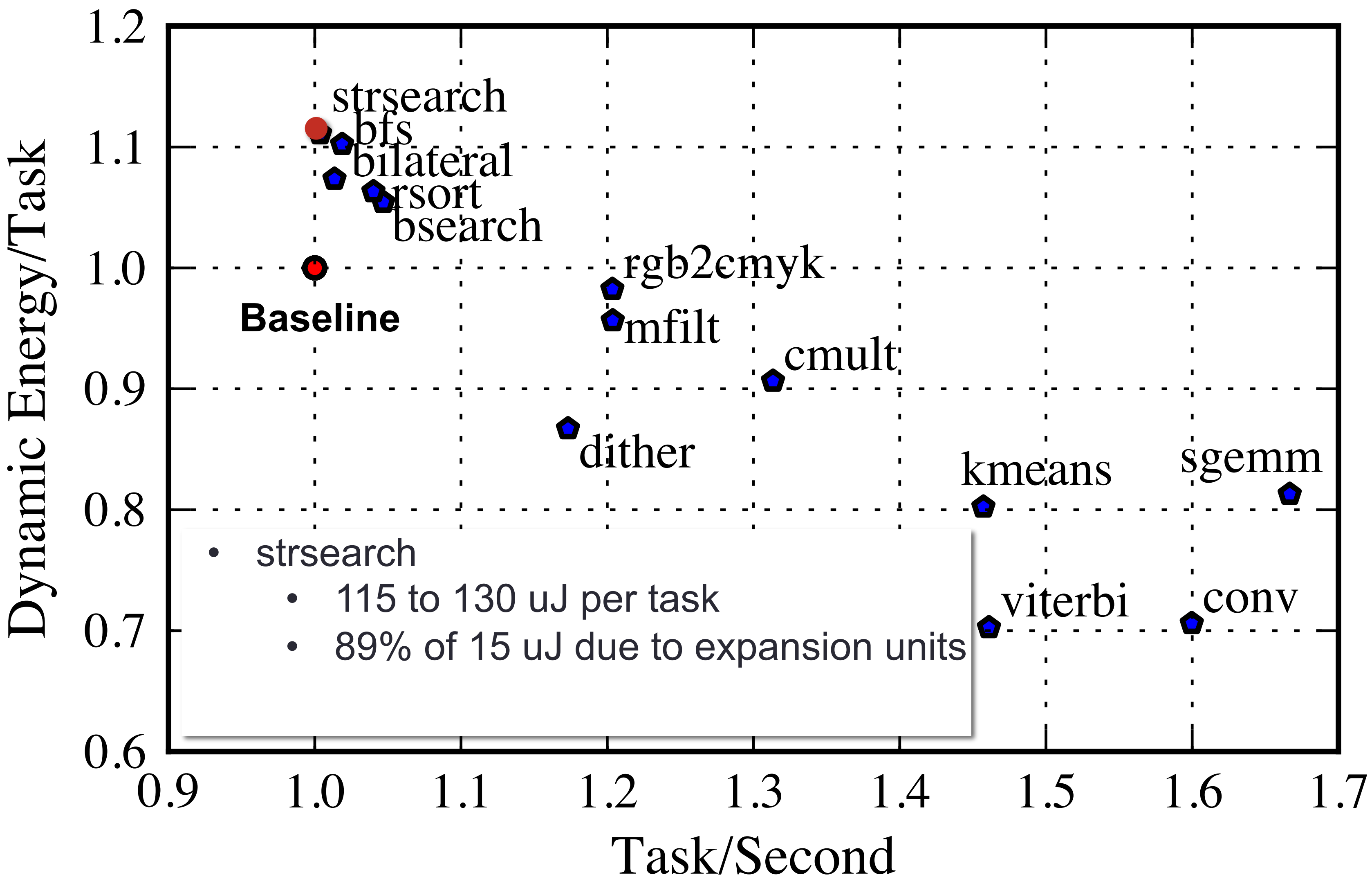


FG-SIMT Energy vs. Performance Results



- viterbi
  - 54% saved within register file
  - 29% saved within functional units
  - 34% saved within memory system

FG-SIMT Energy vs. Performance Results



- 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