

CS Brown Bag Lunch

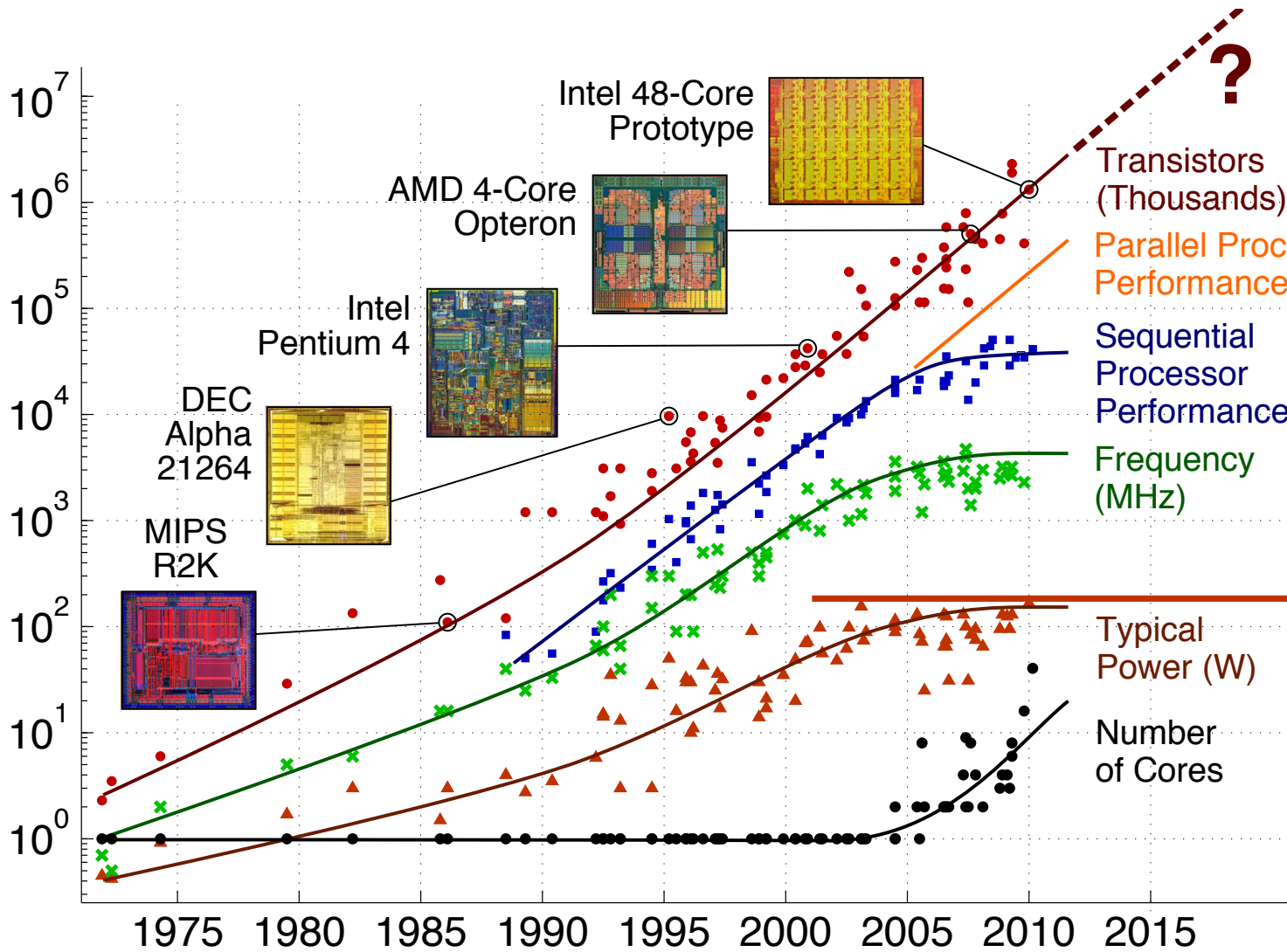
**Microarchitectural Mechanisms to  
Exploit Value Structure in  
SIMT Architectures**

Christopher Batten

Computer Systems Laboratory  
School of Electrical and Computer Engineering  
Cornell University

Spring 2013

# Motivating Trends in Computer Architecture



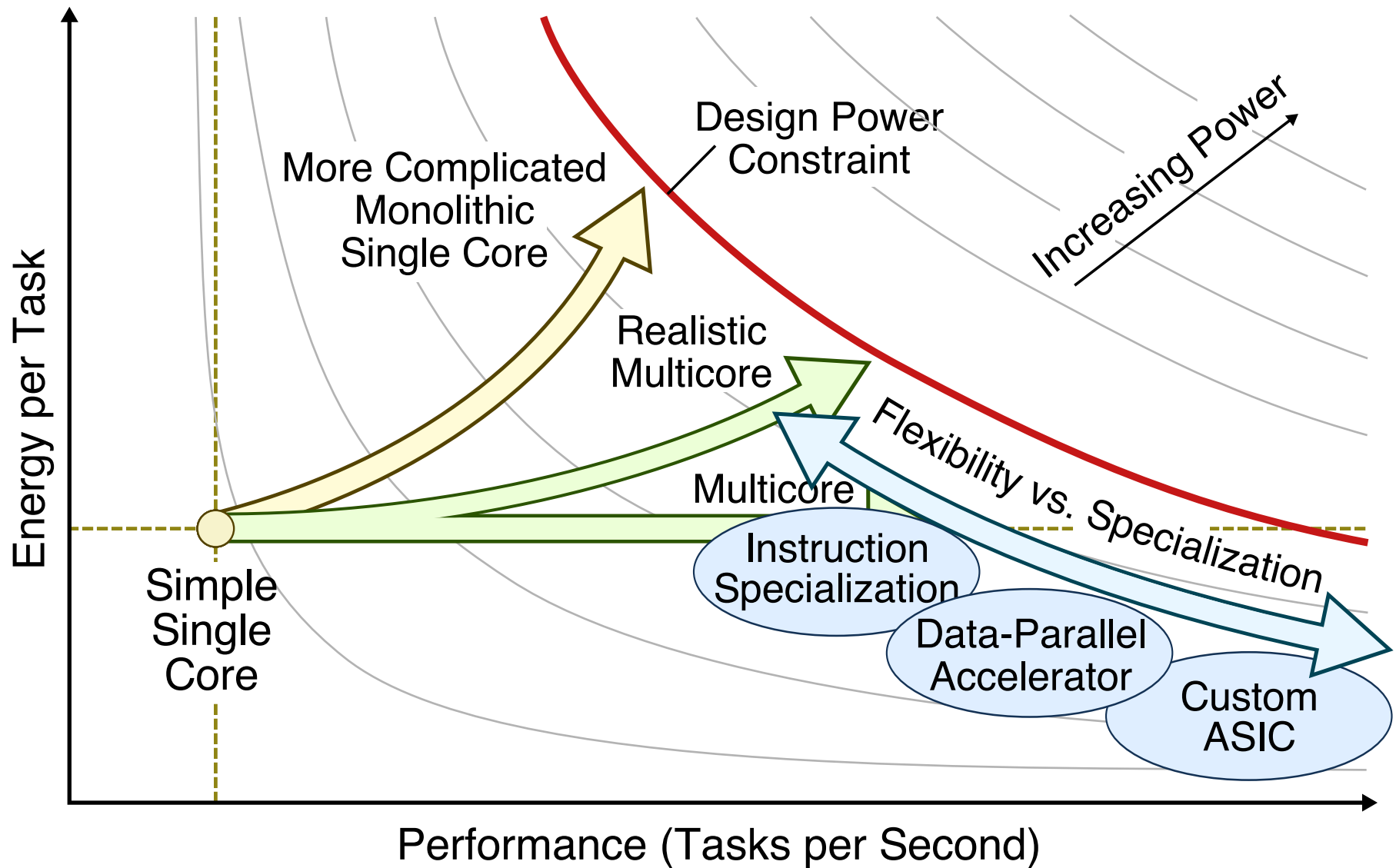
Data partially collected by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond

**Trend 1**  
Power & energy  
constrain all  
systems

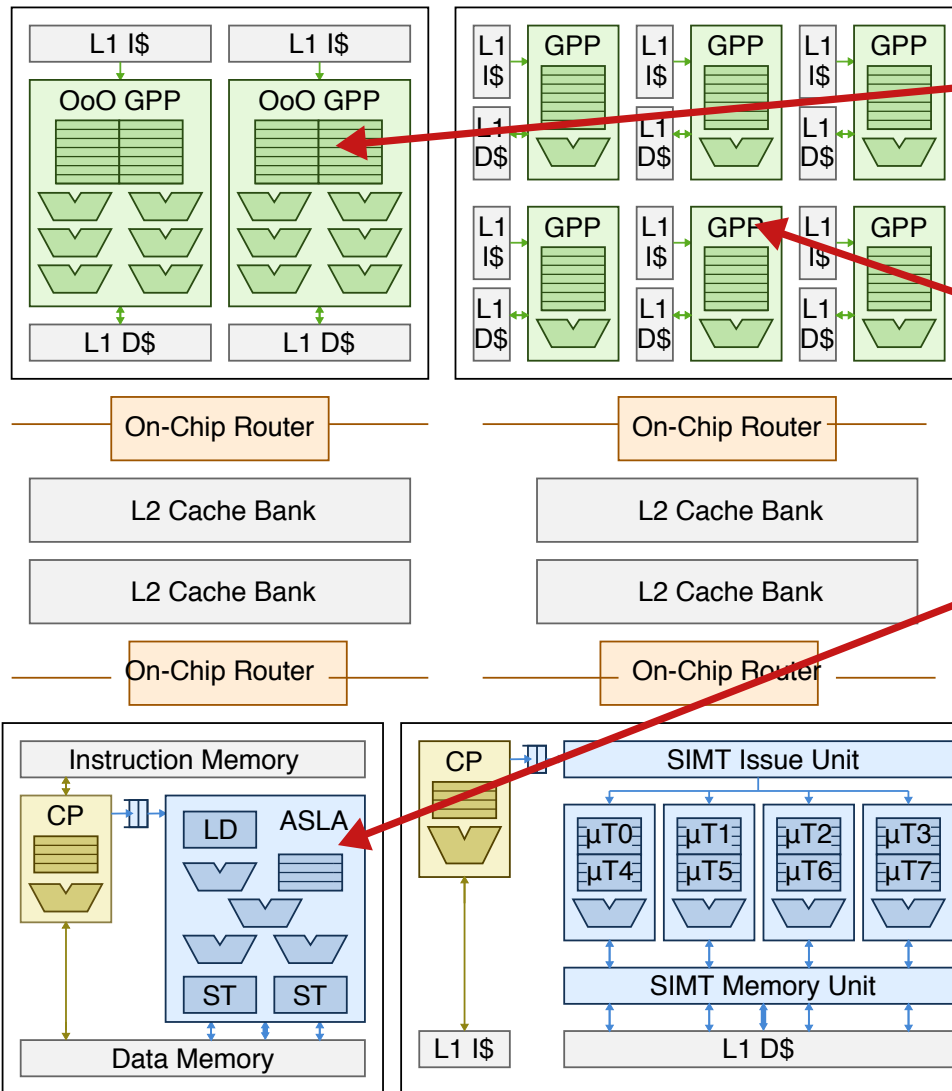
**Trend 2**  
Transition to  
multicore  
processors

**Trend 3**  
Inevitable end  
of Moore's law

# Flexibility versus Specialization



# Pervasive Heterogeneous Specialization



**Latency-Optimized Tile:**  
few large out-of-order cores

**Throughput-Optimized Tile:**  
many small in-order cores

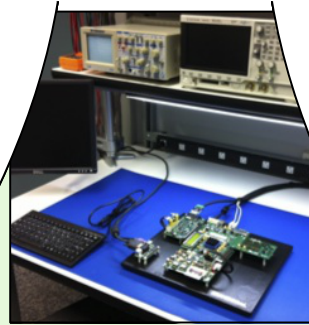
**ASLA Tile:** application  
specific loop accelerator

**Fine-Grain SIMT Tile:**  
flexible data-parallel  
accelerator

# Projects Within the Batten Research Group

## Data-Parallel Specialization

- Fine-Grain Single-Instruction Multiple-Thread Architectures
- XPC: Explicit-Parallel-Call Architectures

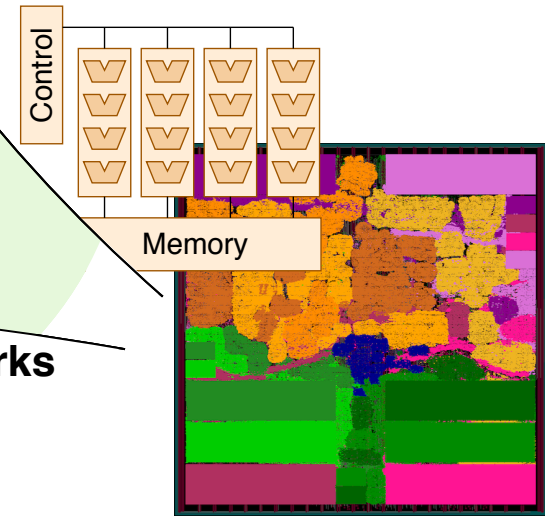


## Domain-Specific Specialization

- Polymorphic Algorithm and Data-Structure Specialization
- Coarse-Grain Reconfigurable Accelerators

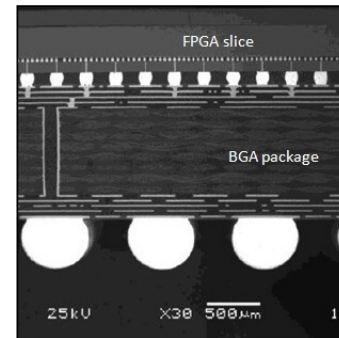
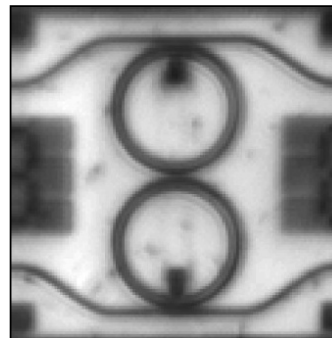
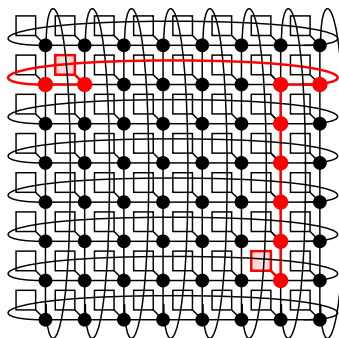
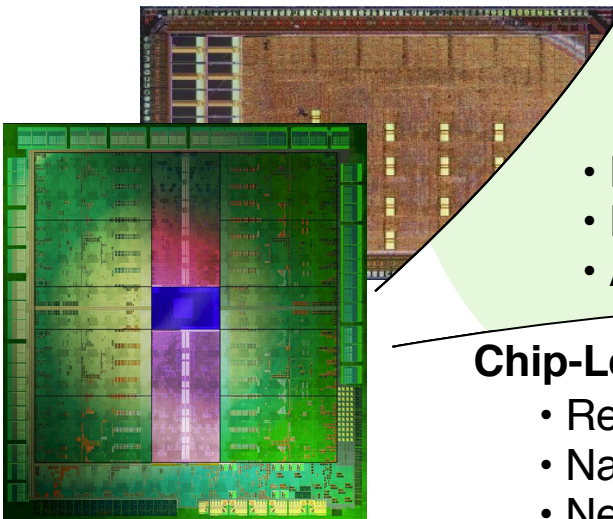
## Vertically Driven Research Approach

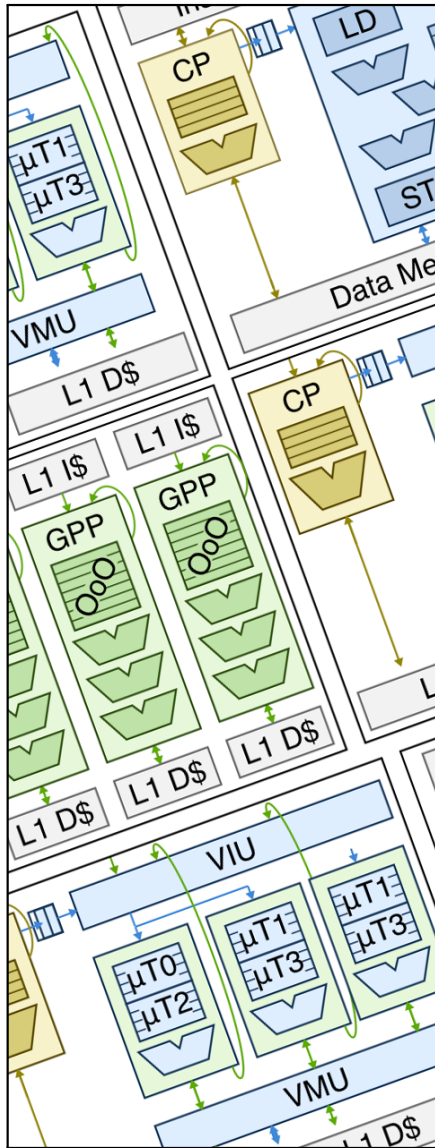
- Python modeling framework
- FPGA prototypes/emulation
- Architecture test chips



## Chip-Level Interconnection Networks

- Realistic On-Chip Networks
- Nanophotonic Networks
- Networks for Silicon Interposers





# Agenda

---

Research Overview

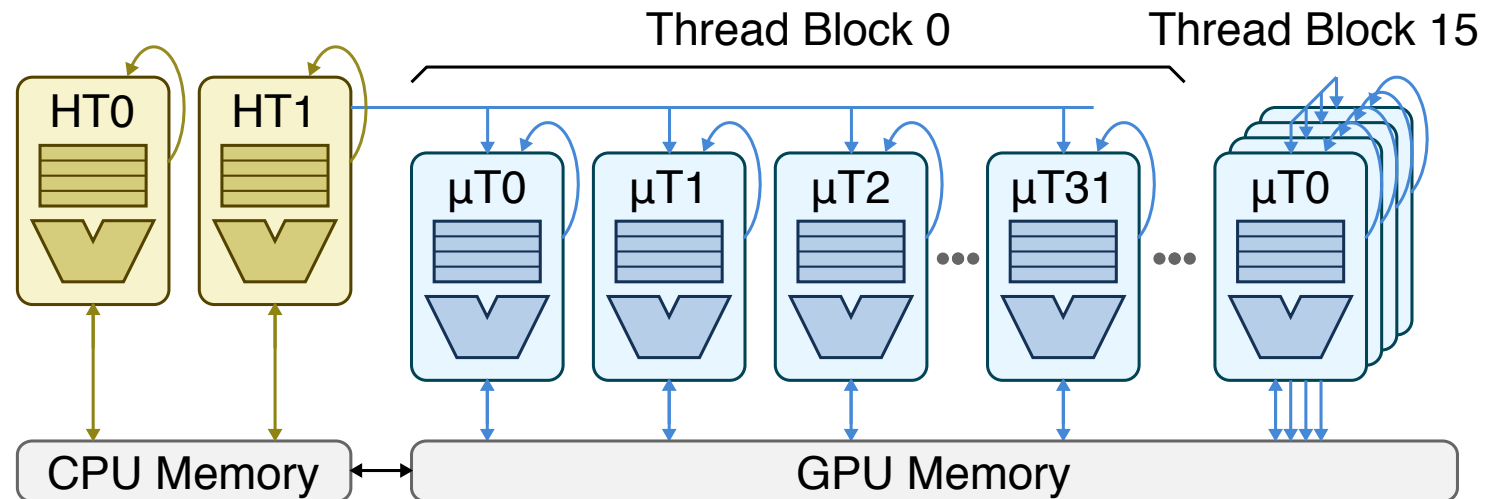
Value Structure in SIMT Kernels

FG-SIMT Baseline Microarchitecture

FG-SIMT Compact Affine Execution

Evaluation

# General-Purpose SIMT Programming Model

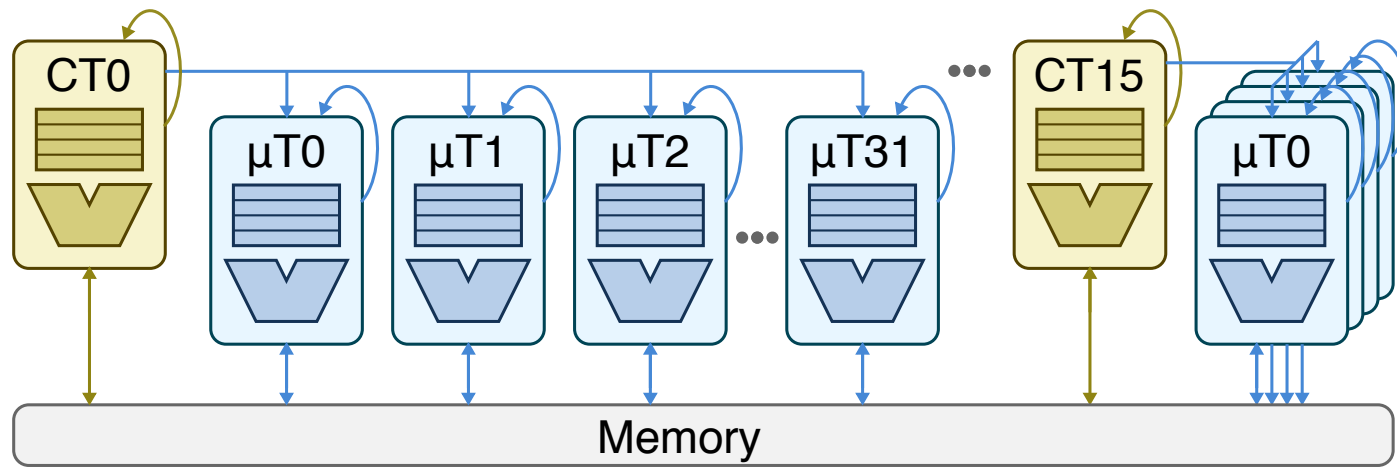


```

__global__ void vsadd_kernel( int y[], int a ) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    y[idx] = y[idx] + a;
}
...
void vsadd( int y[], int a, int n ) {
    // move data from CPU to GPU
    vsadd_kernel<<<32,n/32>>>( y, a );
    // move data from GPU to CPU
}

```

# Fine-Grain SIMT Programming Model



```

__kernel__ void vsadd_kernel( int y[], int a ) {
    int idx = fgsimt::init_kernel( y, a );
    y[idx] = y[idx] + a;
}
...
void vsadd( int y[], int a, int n ) {
    // distribute work among control threads
    // x = base pointer for this control thread
    fgsimt::launch_kernel( n/32, &vsadd_kernel, x, a );
}

```



# Fragment from Viterbi Application

```
__kernel__ void
calc_fwd_paths_kernel( ... ) {
    int idx = fgsimt::init_kernel( ... );
    ...

    // Inner loop
    for (int j = 0; j < vrate; j++)
        metric += bt_ptr[idx+j*STATES/2] ^ symbols[s*vrate+j];
    ...

    // More complicated array indexing
    m0 = old_error[idx] + metric;
    m1 = old_error[idx+STATES/2] + (max - metric);
    m2 = old_error[idx] + (max - metric);
    m3 = old_error[idx+STATES/2] + metric;
    ...

    // Data-dependent control flow
    new_error[2*idx] = decision0 ? m1 : m0;
    new_error[2*idx+1] = decision1 ? m3 : m2;
    ...
}
```

# Control and Memory Access Structure

## Regular Data Access Regular Control Flow

```
for ( i = 0; i < n; i++ )  
    C[i] = A[i] + B[i];  
  
for ( i = 0; i < n; i++ )  
    C[i] = x * A[i] + B[2*i];
```

## Irregular Data Access Regular Control Flow

```
for ( i = 0; i < n; i++ )  
    E[C[i]] = D[A[i]] + B[i];
```

## Regular Data Access Irregular Control Flow

```
for ( i = 0; i < n; i++ )  
    x = ( A[i] > 0 ) ? y : z;  
    C[i] = x * A[i] + B[i];
```

## Irregular Data Access Irregular Control Flow

```
for ( i = 0; i < n; i++ )  
    if ( A[i] > 0 )  
        C[i] = x * A[i] + B[i];  
  
for ( i = 0; i < n; i++ )  
    C[i] = false; j = 0;  
    while ( !C[i] & ( j < m ) )  
        if ( A[i] == B[j++] )  
            C[i] = true;
```


# FG-SIMT Pseudo-Assembly Example

```
__kernel__ void
ex_kernel( int y[], int a ) {
    int idx
    = fgsimt::init_kernel(y,a);

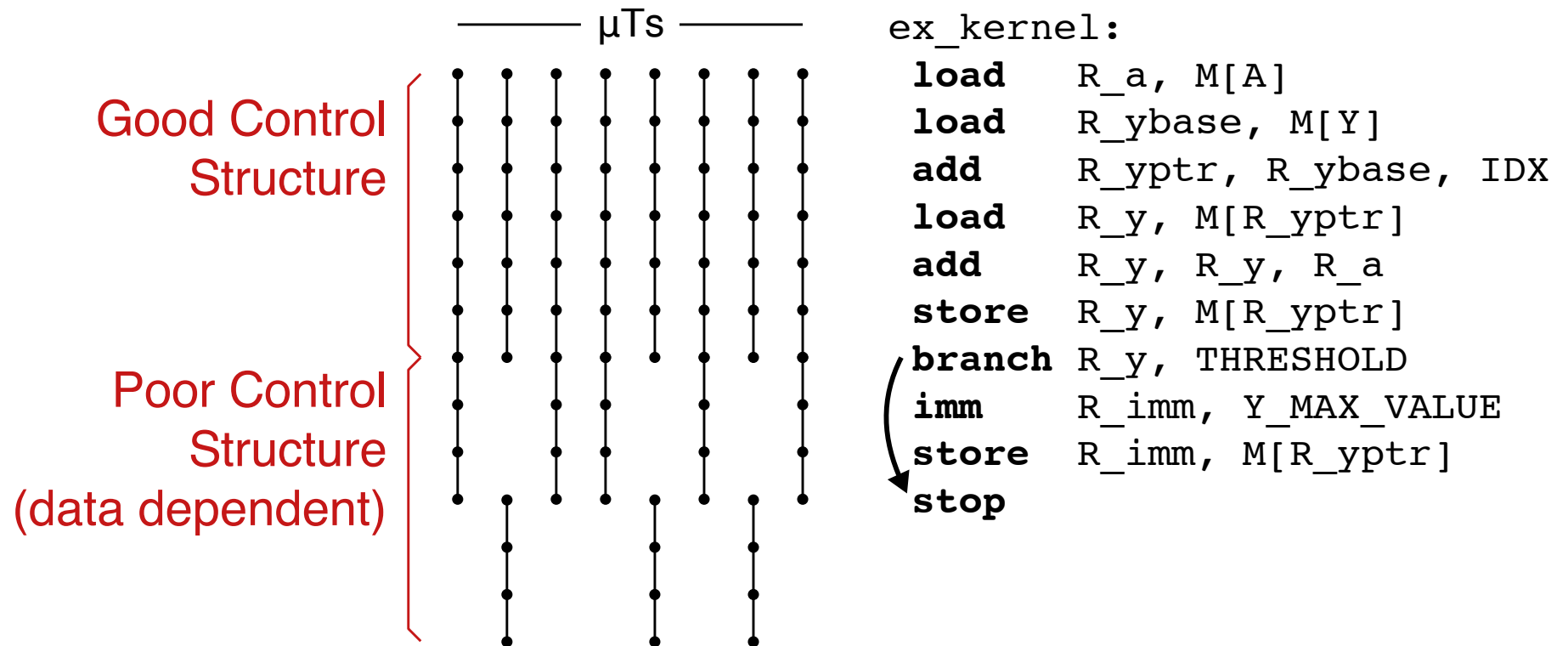
    y[idx] = y[idx] + a;

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

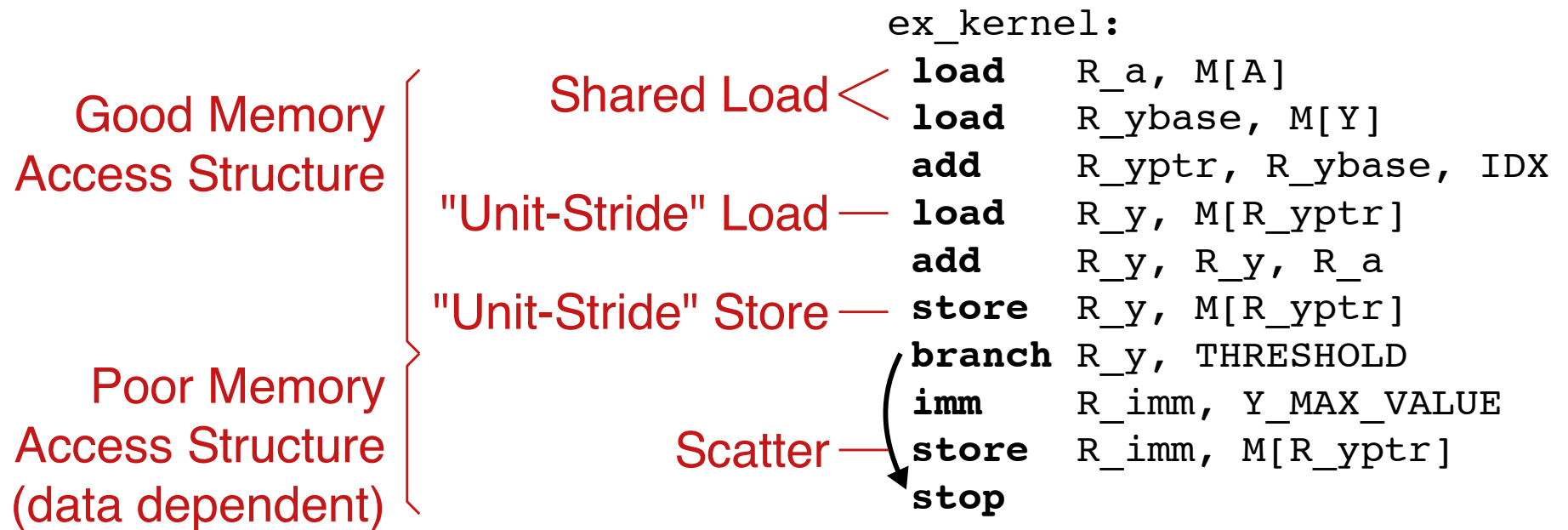
```
ex_kernel:
    load    R_a, M[A]
    load    R_ybase, M[Y]
    add     R_yptr, R_ybase, IDX
    load    R_y, M[R_yptr]
    add     R_y, R_y, R_a
    store   R_y, M[R_yptr]
    branch  R_y, THRESHOLD
    imm     R_imm, Y_MAX_VALUE
    store   R_imm, M[R_yptr]
    stop
```



# Control Structure in FG-SIMT Kernels



# Memory Access Structure in FG-SIMT Kernels



# Value Structure in FG-SIMT Kernels

## Affine Value Structure:

$$V(i) = b + i \times s$$

## Affine Arithmetic

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

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

Affine branches and affine memory operations are also possible

ex\_kernel:

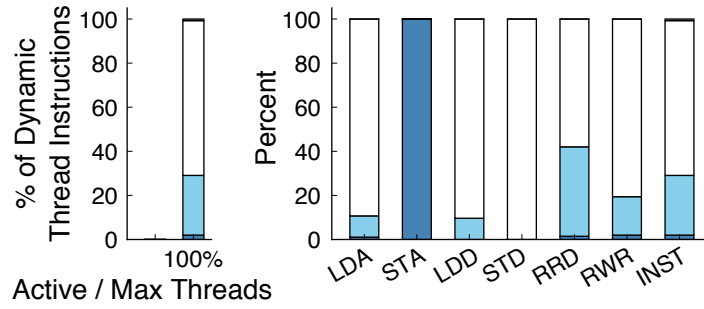
```

load  R_a, M[A]
load  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_imm, Y_MAX_VALUE
store R_imm, M[R_yptr]
stop

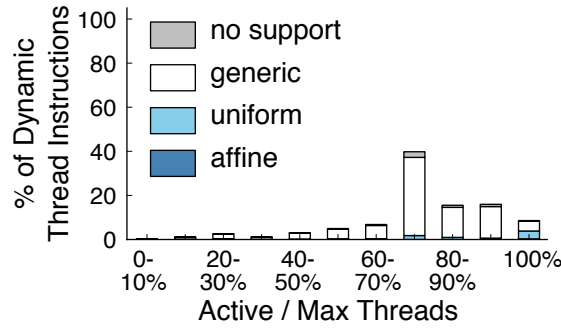
```

- Uniform values across threads; If both inputs are uniform, we can execute the instruction once on the control processor
- Affine values across threads; If inputs are affine/uniform, we can still potentially execute instruction once on the control processor
- Generic values across threads (i.e., no structure); we must ensure values are expanded and explicitly execute the instruction for each thread

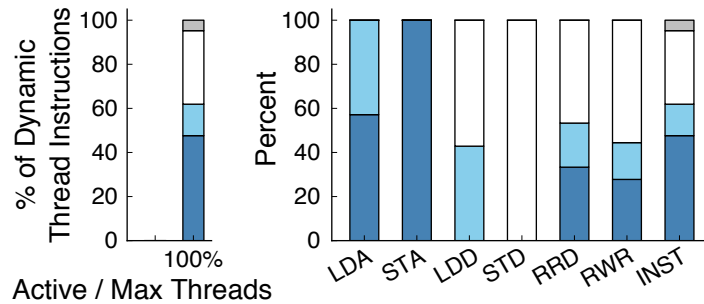
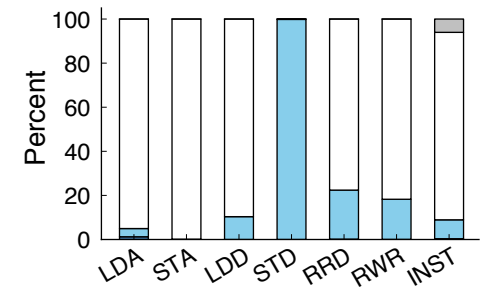
# Characterizing Structure in FG-SIMT Kernels



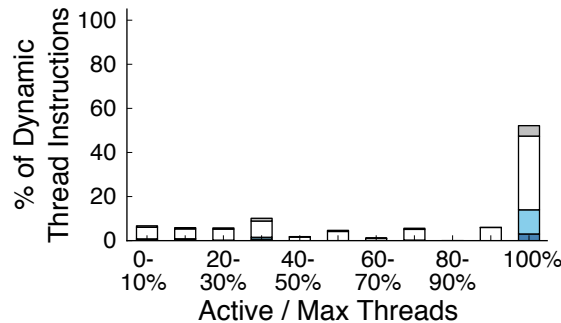
**Dense Matrix Multiplication**



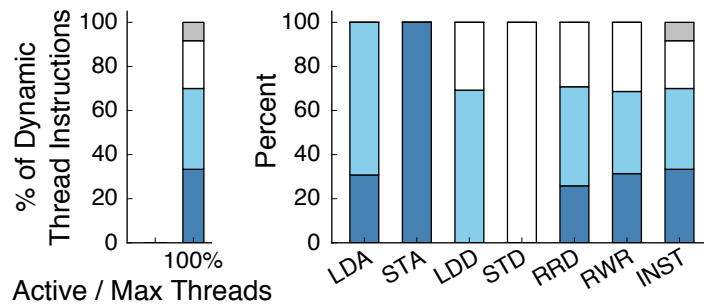
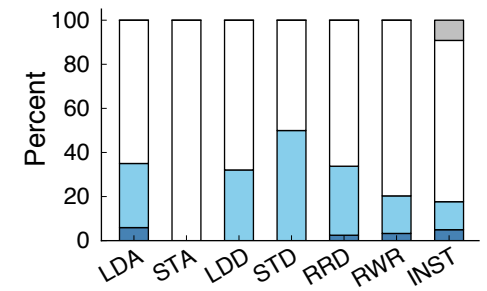
**Searching for Multiple Strings in Multiple Documents**



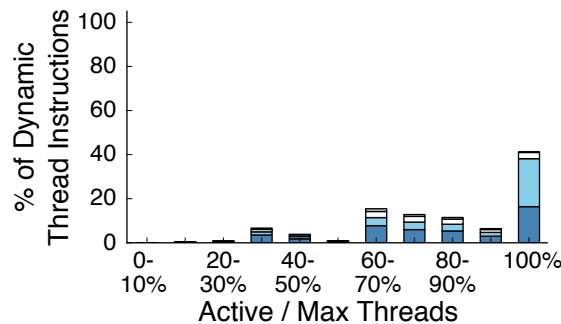
**Vector-Vector Complex Multiply**



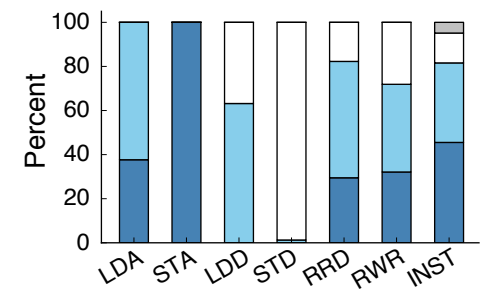
**Breadth-First Search from Source to All Other Nodes**

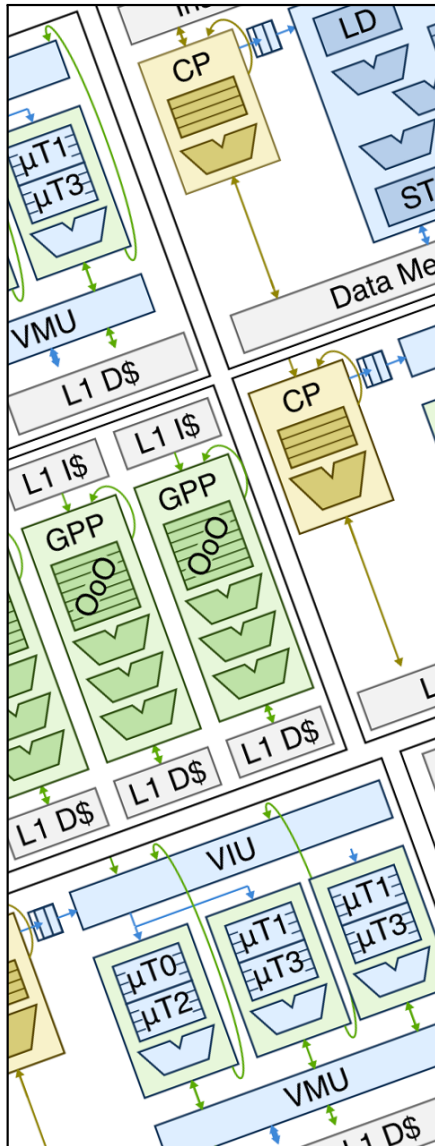


**Viterbi Decoder**



**Gaussian Image Blur Under Mask**





## Agenda

---

Research Overview

Value Structure in SIMT Kernels

**FG-SIMT Baseline Microarchitecture**

FG-SIMT Compact Affine Execution

Evaluation

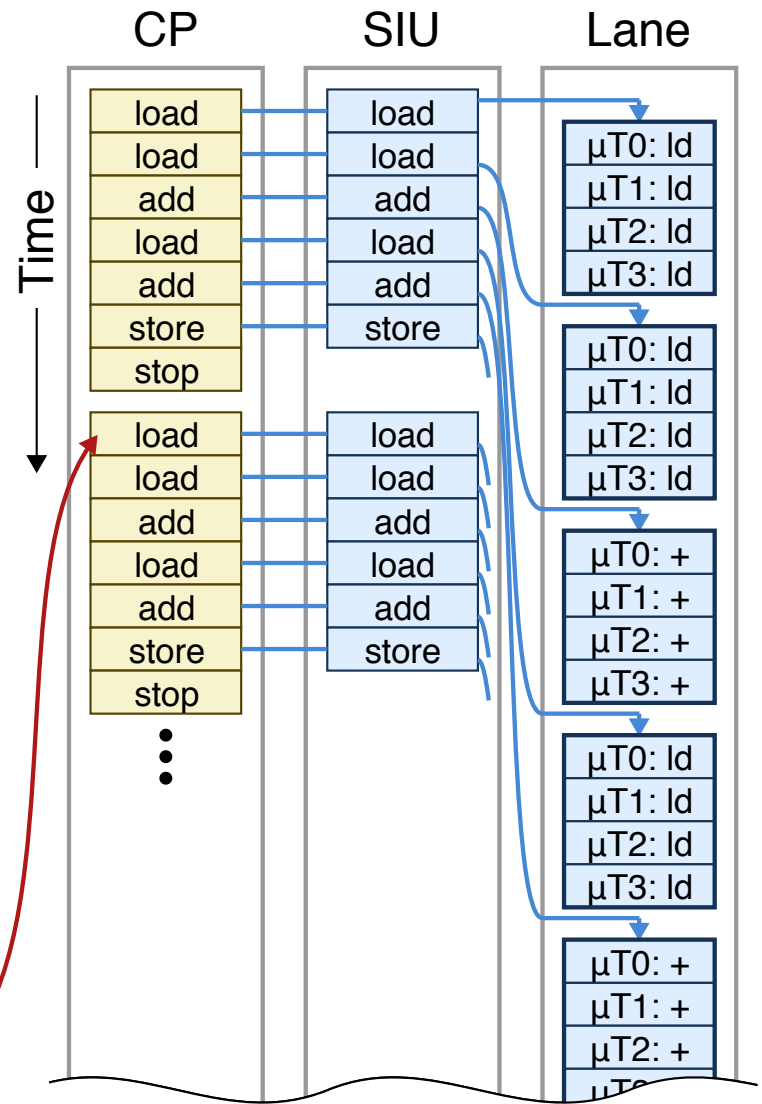
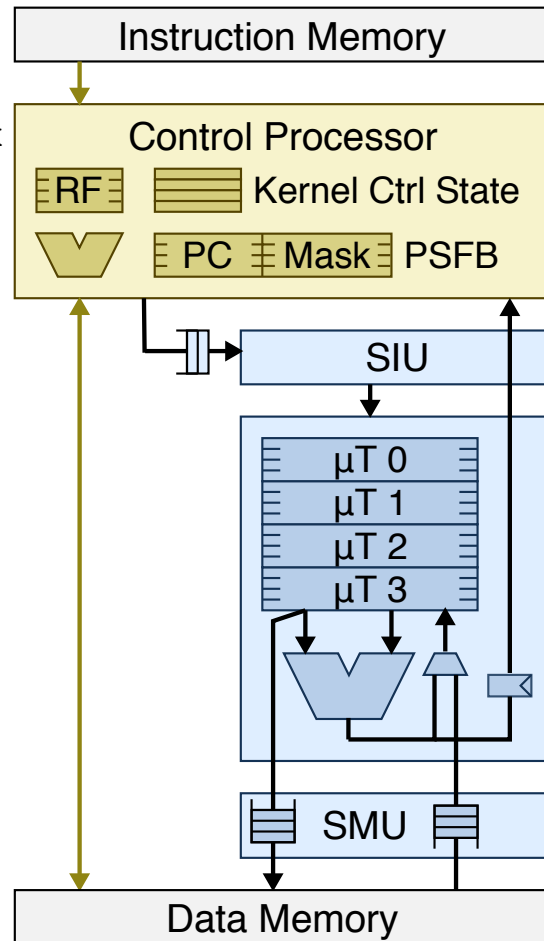


# FG-SIMT Microarchitecture: Regular Control Flow

ex\_kernel:

```

load R_a, M[A]
load 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]
stop
    
```



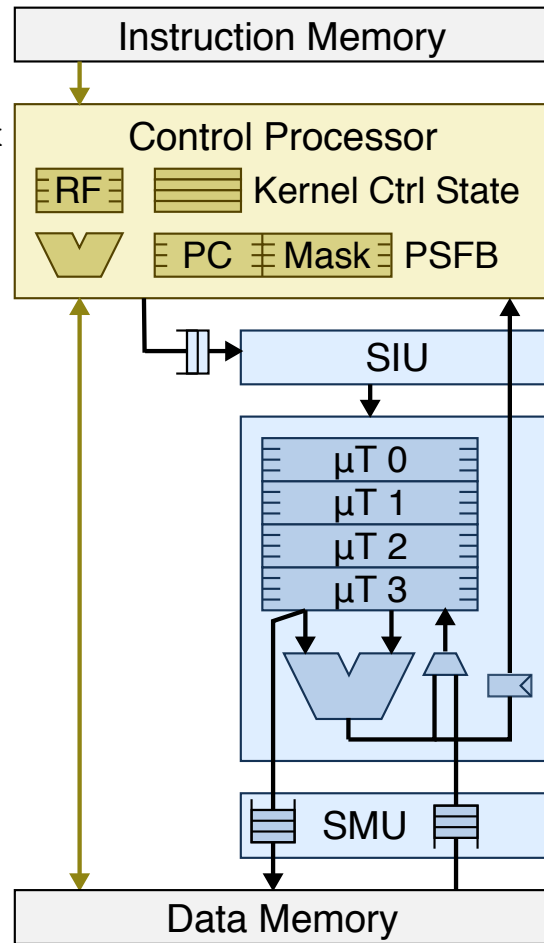
Next iteration of hardware stripmined loop

# FG-SIMT Microarchitecture: Irregular Control Flow

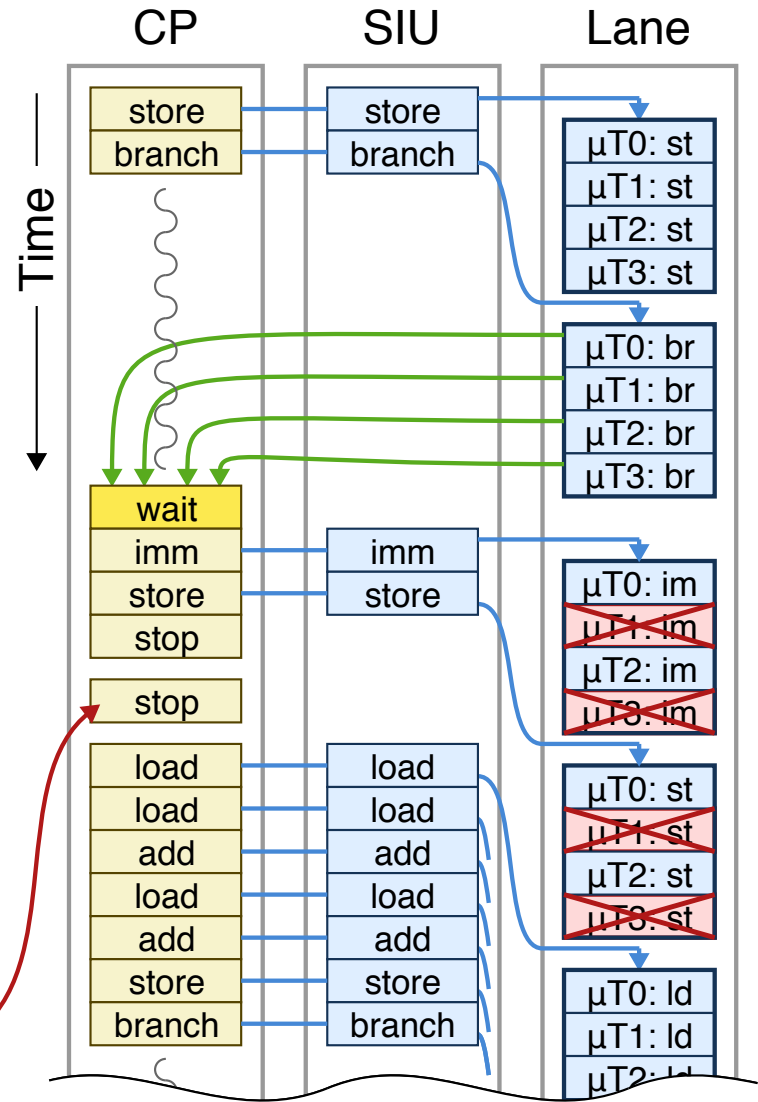
ex\_kernel:

```

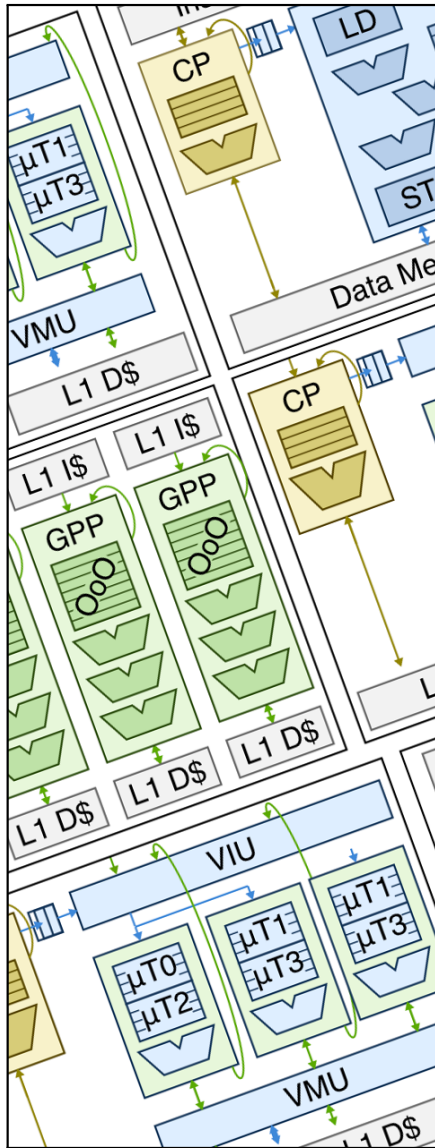
load R_a, M[A]
load R_ybase, M[Y]
add R_yptr, R_ybase, IDX
load R_y, M[R_yptr]
add R_y, R_y, R_a
store R_y, M[R_yptr]
branch R_y, THRESHOLD
imm R_imm, Y_MAX_VALUE
store R_imm, M[R_yptr]
stop
    
```



Finish pending SIMT fragment







## Agenda

---

Research Overview

Value Structure in SIMT Kernels

FG-SIMT Baseline Microarchitecture

**FG-SIMT Compact Affine Execution**

Evaluation



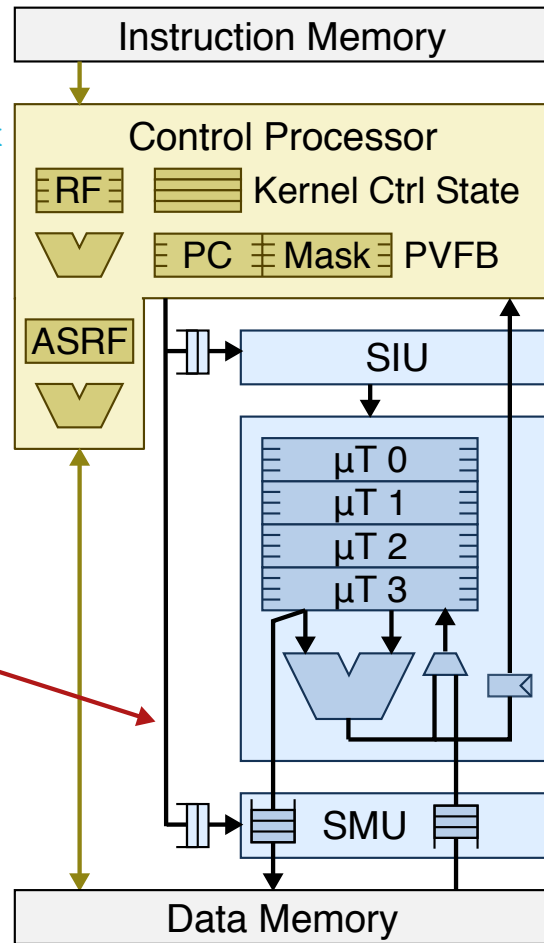
# Compact Affine Execution: Affine Memory Operations

ex\_kernel:

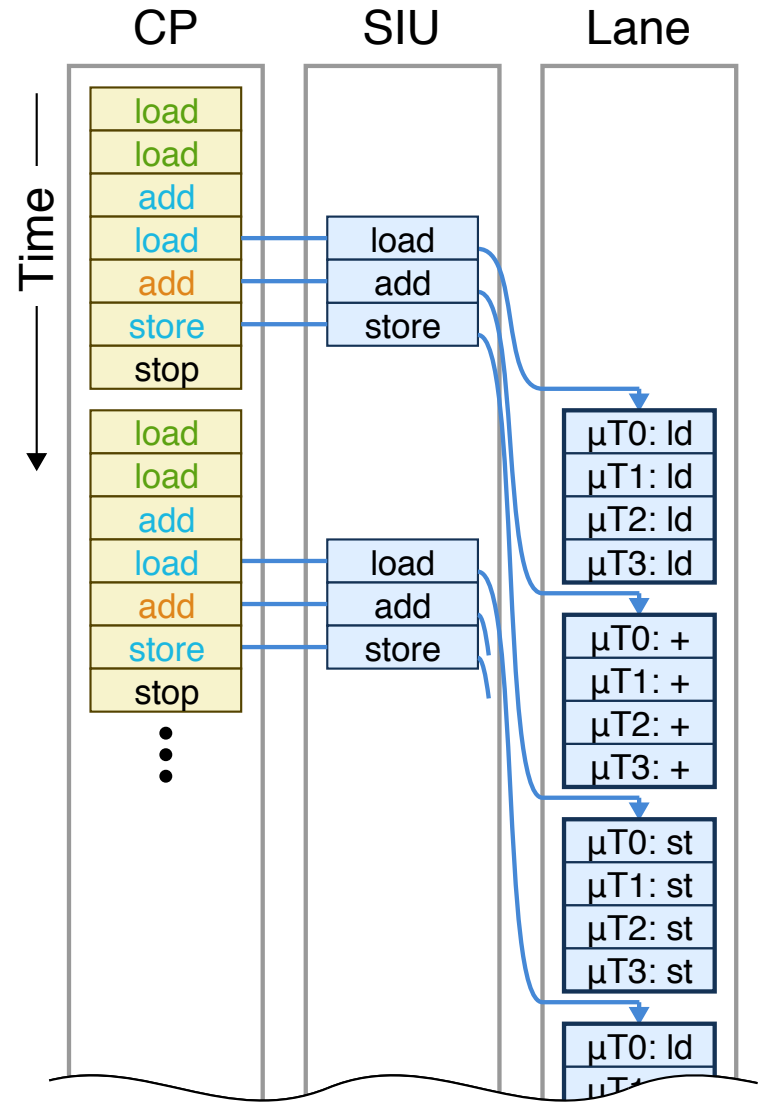
```

load R_a, M[A]
load 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]
stop
    
```

Add direct access to SMU for affine memory accesses



- Uniform
- Affine (not uniform)
- Generic

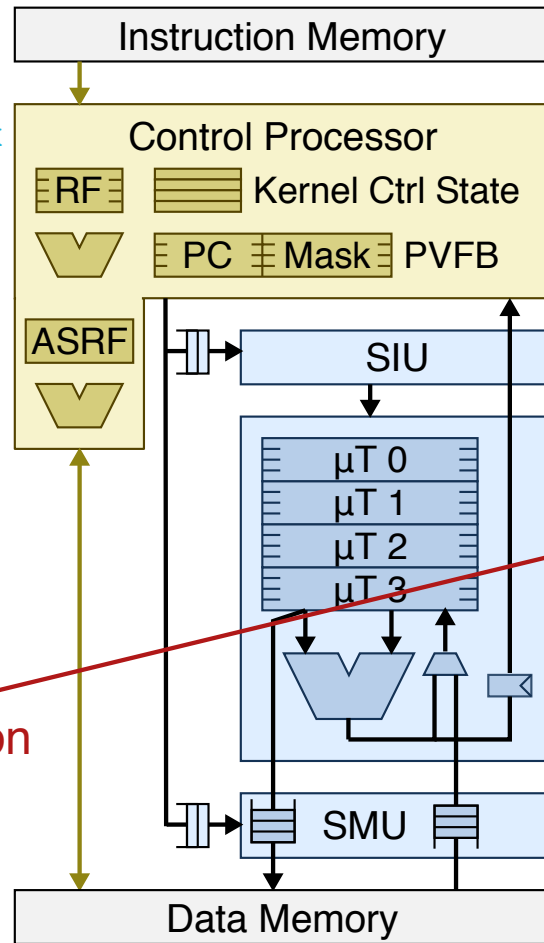


# Compact Affine Execution: Without Divergence

ex\_kernel:

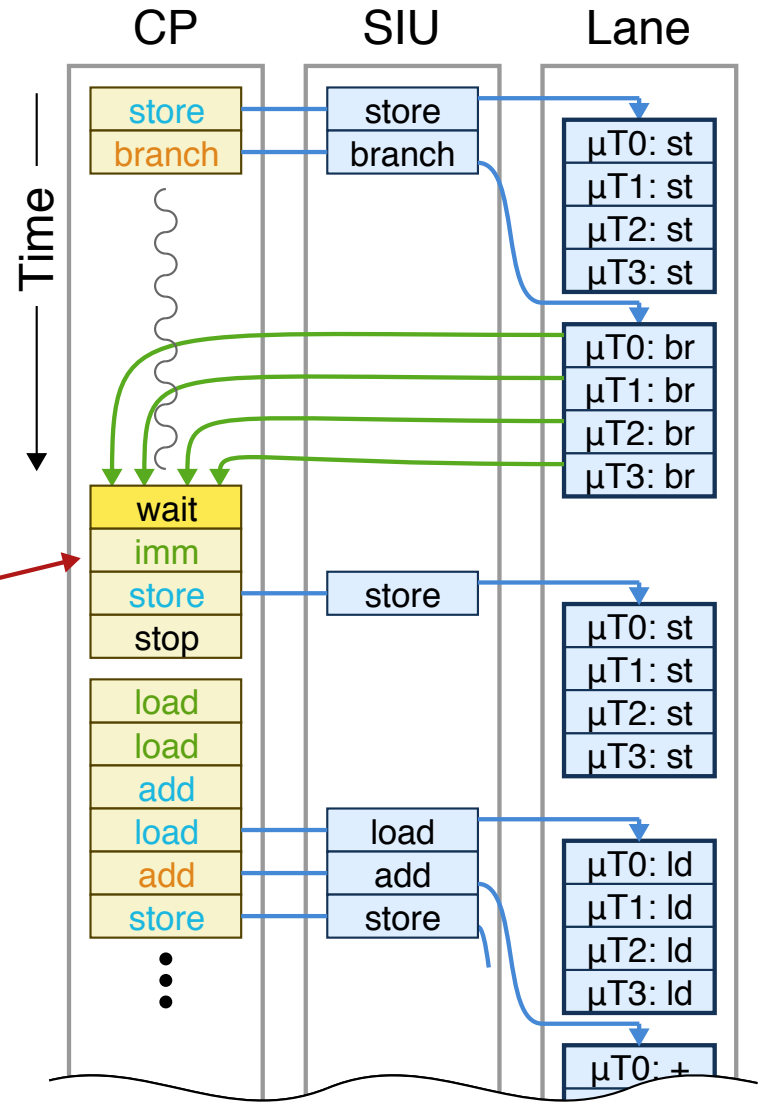
```

load R_a, M[A]
load 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_imm, Y_MAX_VALUE
store R_imm, M[R_yptr]
stop
    
```



Compact affine execution still possible after a branch if there is no divergence

- Uniform
- Affine (not uniform)
- Generic



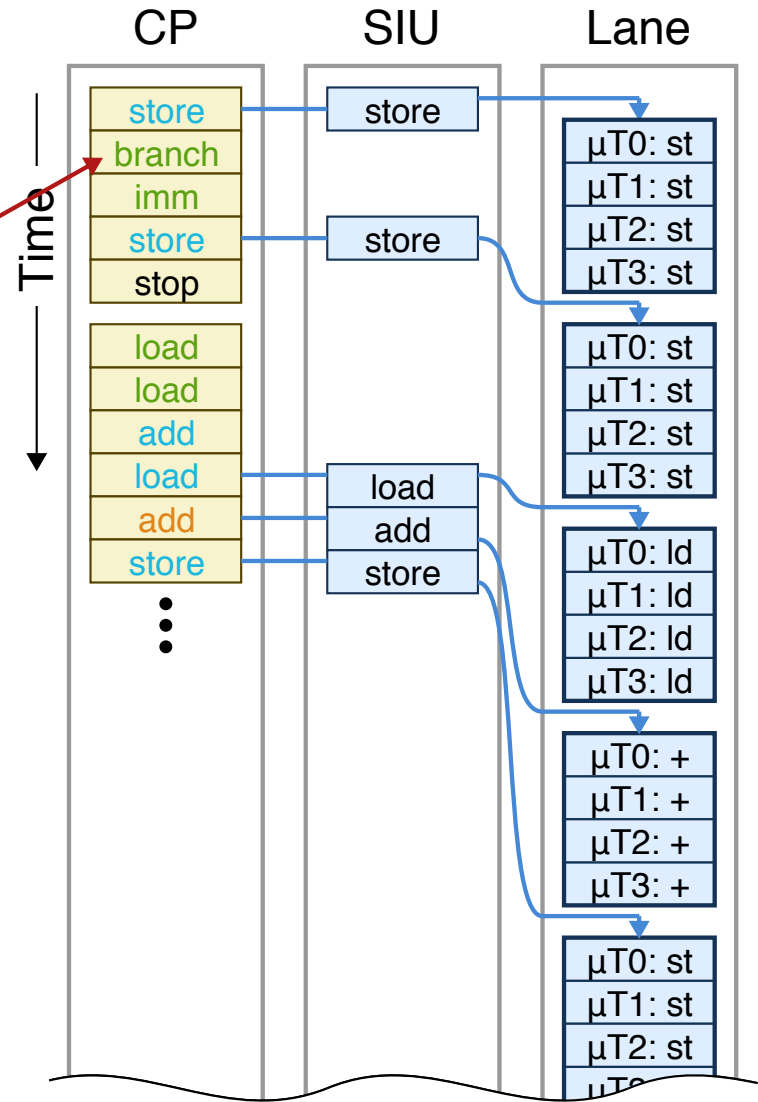
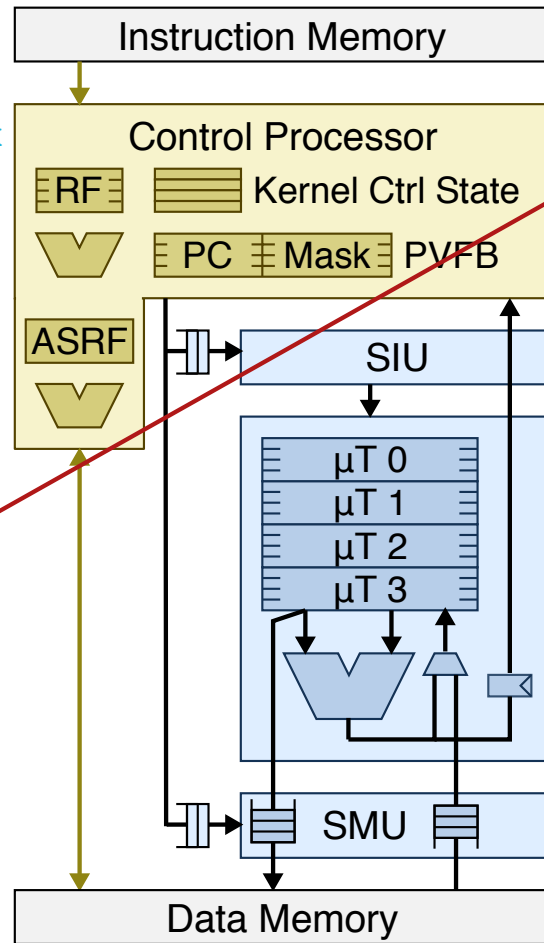
# Compact Affine Execution: Affine Branches

ex\_kernel:

```

load R_a, M[A]
load 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_imm, Y_MAX_VALUE
store R_imm, M[R_yptr]
stop
    
```

If branch operands are uniform, then branch can be completely resolved in CP



- Uniform
- Affine (not uniform)
- Generic

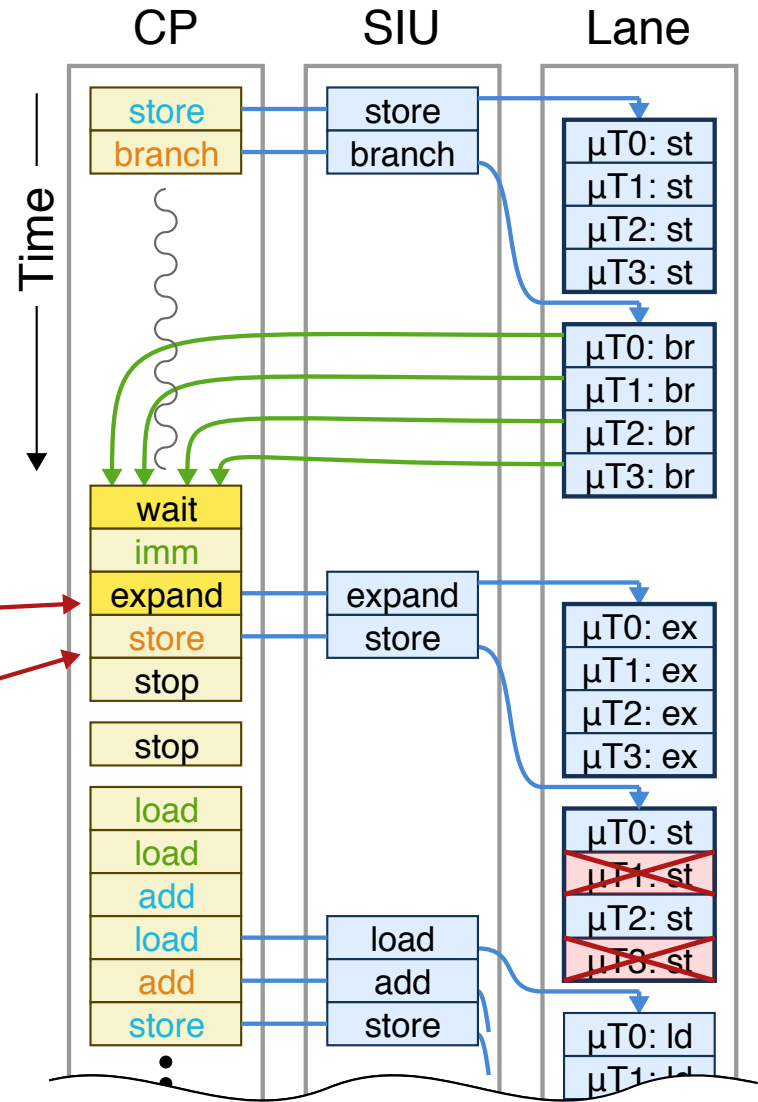
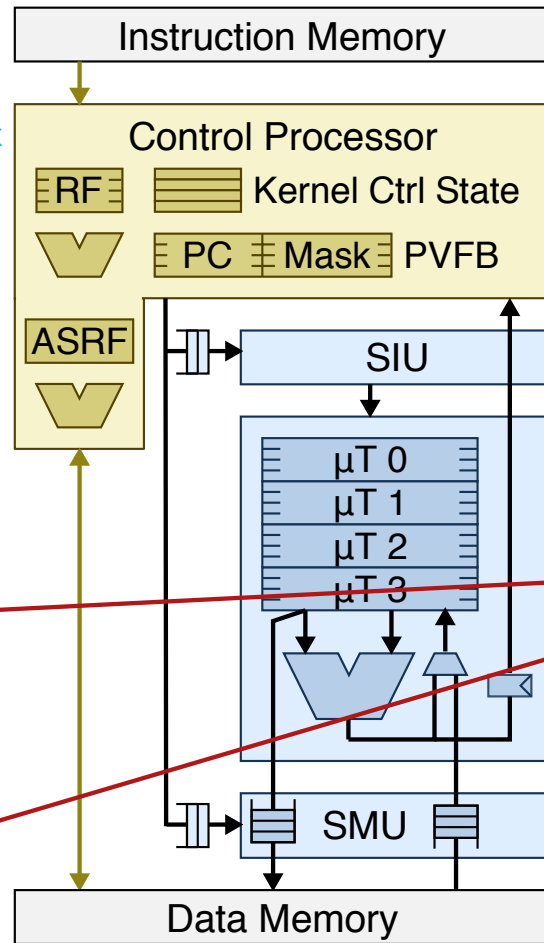


# Compact Affine Execution: With Divergence

ex\_kernel:

```

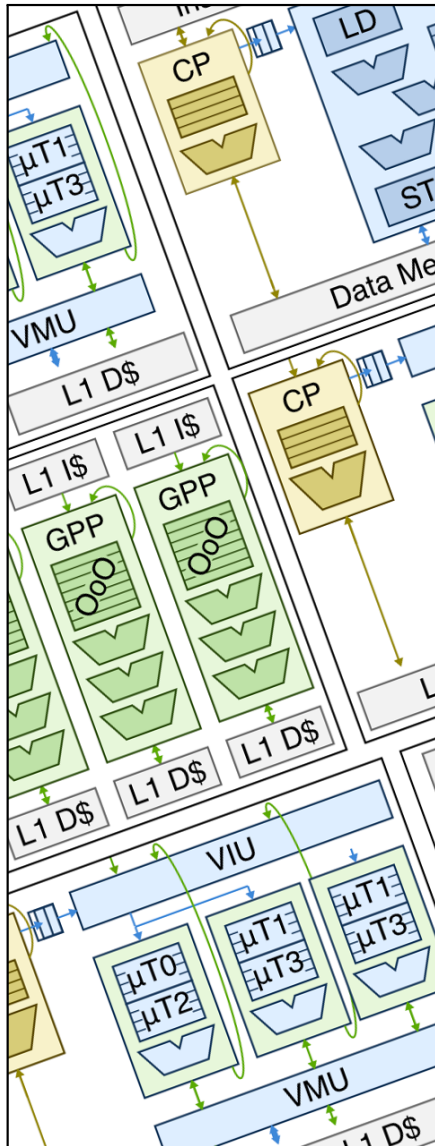
load  R_a, M[A]
load  R_ybase, M[Y]
add   R_yptr, R_ybase, IDX
load  R_y, M[R_yptr]
add   R_y, R_y, R_a
store R_y, M[R_yptr]
branch R_y, THRESHOLD
imm   R_imm, Y_MAX_VALUE
store R_imm, M[R_yptr]
stop
    
```



Compact affine execution still possible, but must expand out result

Store cannot now be executed compactly

- Uniform
- Affine (not uniform)
- Generic



# Agenda

---

Research Overview

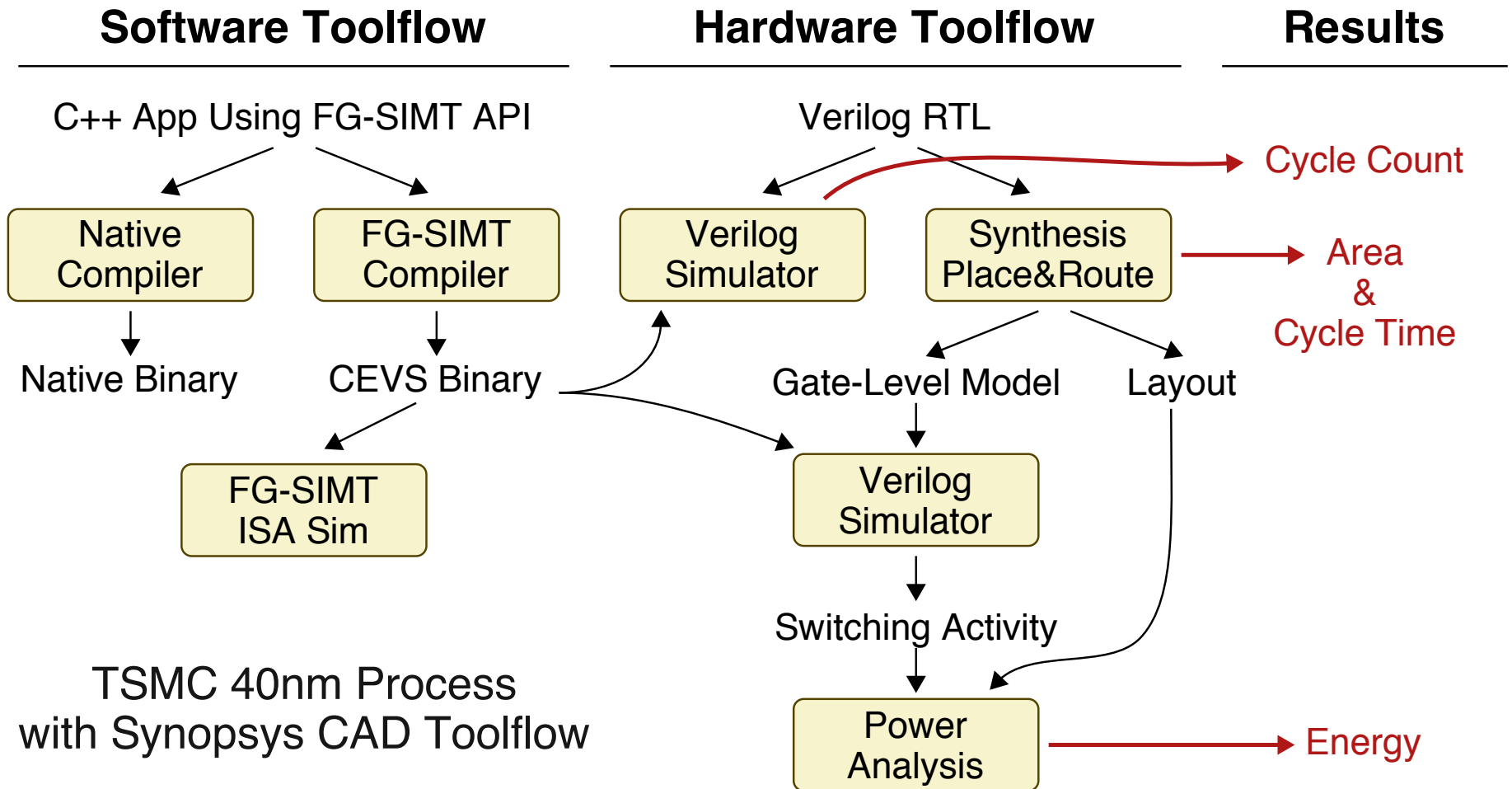
Value Structure in SIMT Kernels

FG-SIMT Baseline Microarchitecture

FG-SIMT Compact Affine Execution

Evaluation

# Software/Hardware Evaluation Methodology

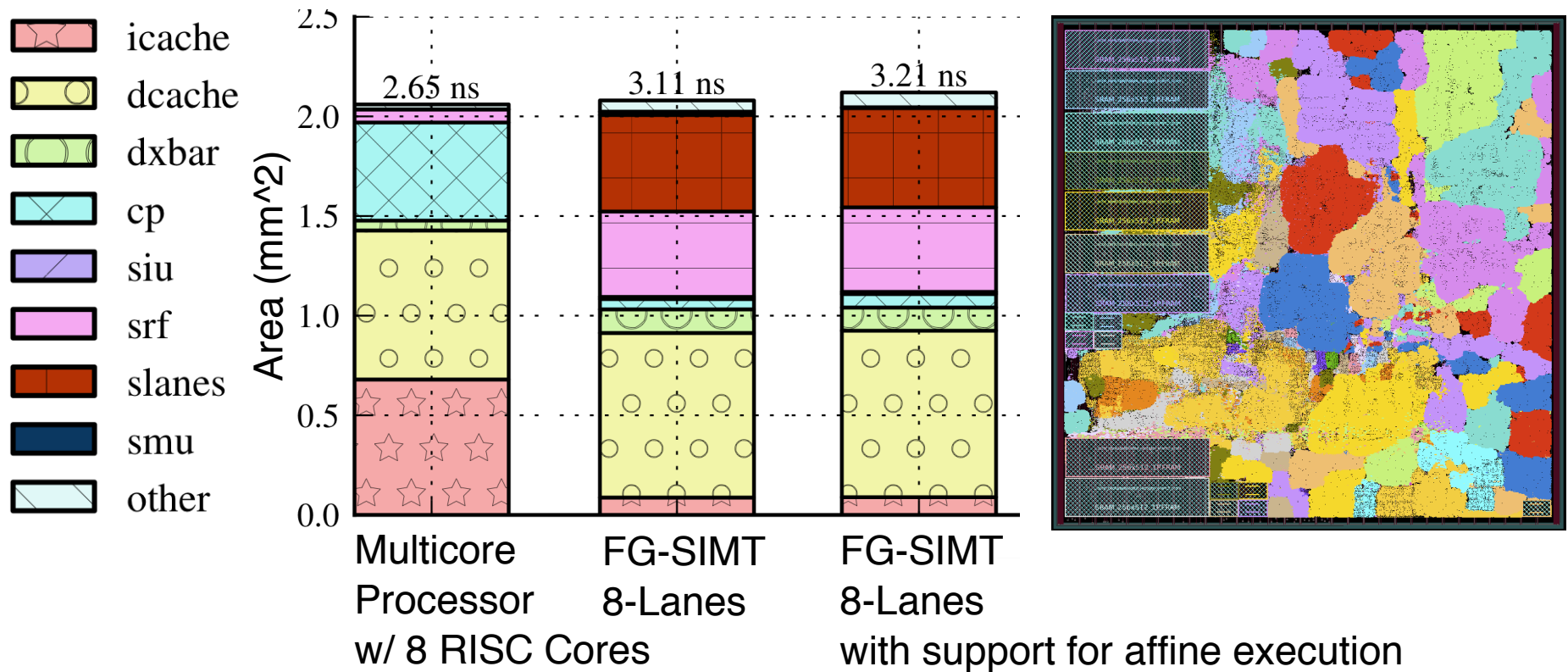


# Application Kernels

---

<b>bfs</b>	Breadth-first search from source to every other node
<b>bilat</b>	Bilateral image filtering, optimized Taylor series for intensity
<b>bsearch</b>	Parallel binary searches in sorted linear array of key/value pairs
<b>cmult</b>	Vector-vector complex single-precision multiplication
<b>conv</b>	1D spatial convolution with large 20-element kernel
<b>dither</b>	Floyd-Steinberg image dithering from gray-scale to black-and-white
<b>kmeans</b>	KMeans clustering
<b>mfilt</b>	Apply Gaussian blur filter to gray-scale image under mask
<b>rgb2cmyk</b>	RGB-to-CMYK color conversion
<b>rsort</b>	Incremental radix sort of integers
<b>sgemm</b>	Dense single-precision matrix-matrix multiply
<b>strsearch</b>	Knuth-Morris-Pratt search for multiple strings in multiple docs
<b>viterbi</b>	Decode frames of convolutionally encoded data using Viterbi algo

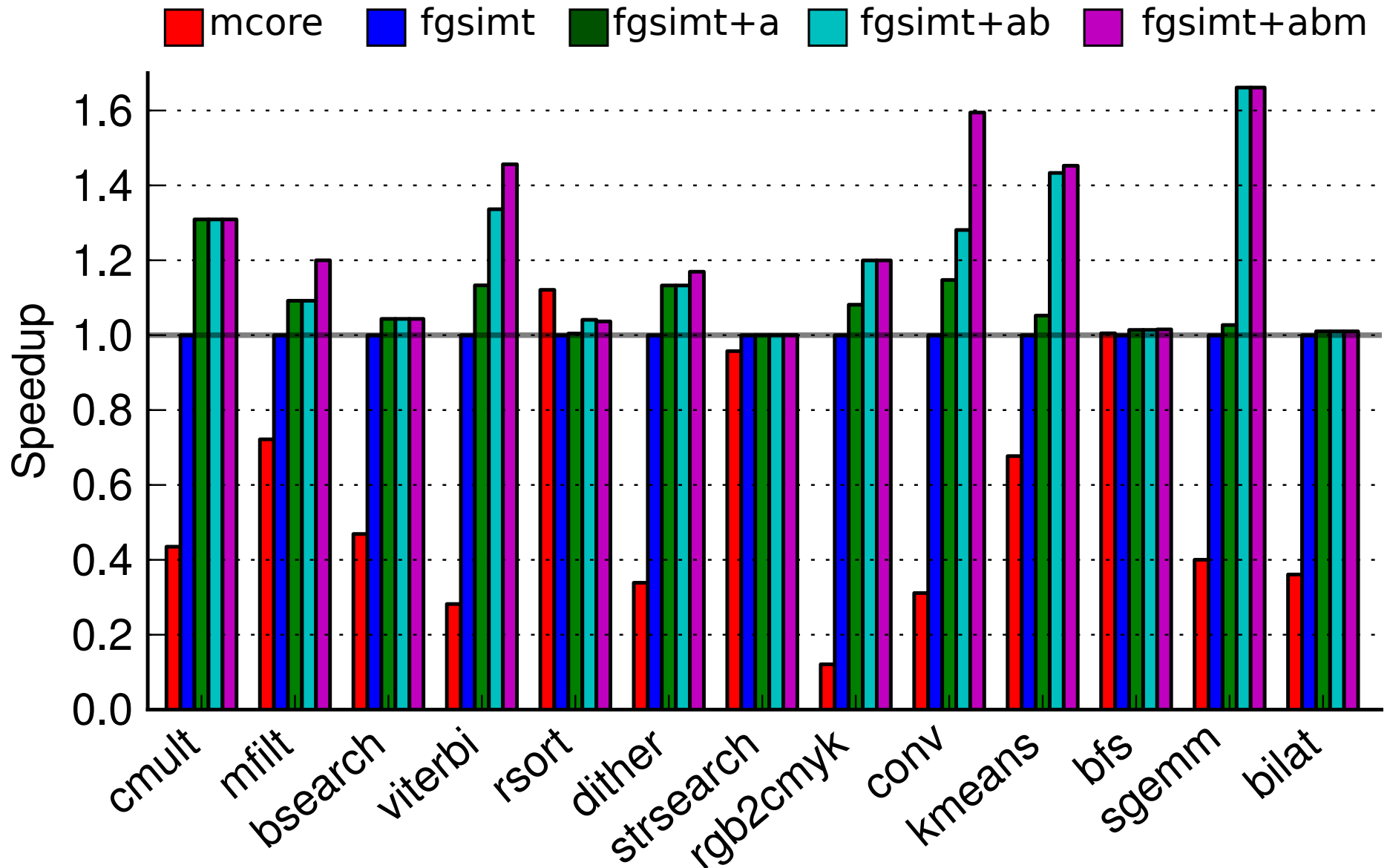
# Area and Cycle Time Results



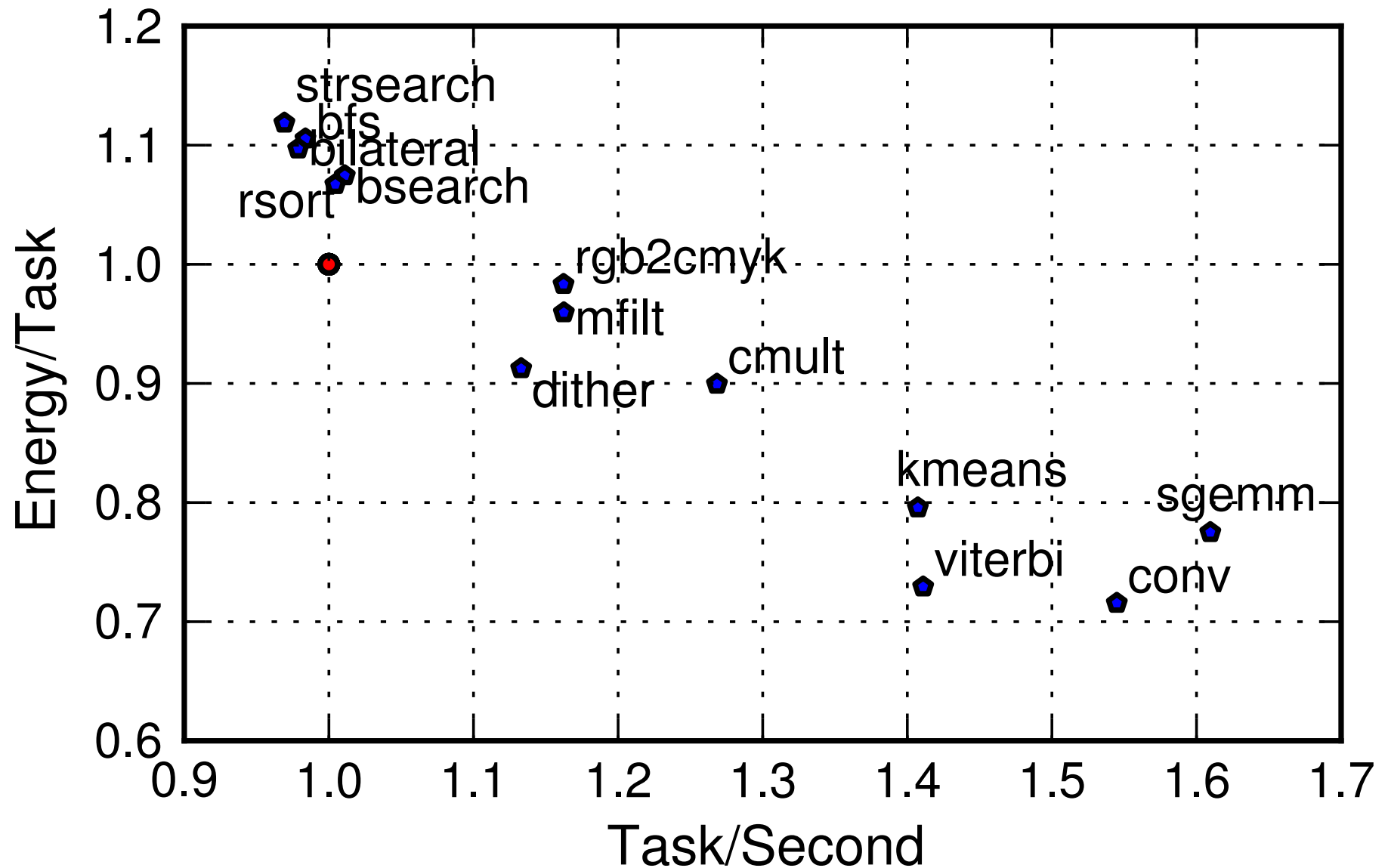
FG-SIMT has comparable area to area to a multicore processor with equivalent floating-point and memory bandwidth resources

Compact affine execution adds relatively little overhead

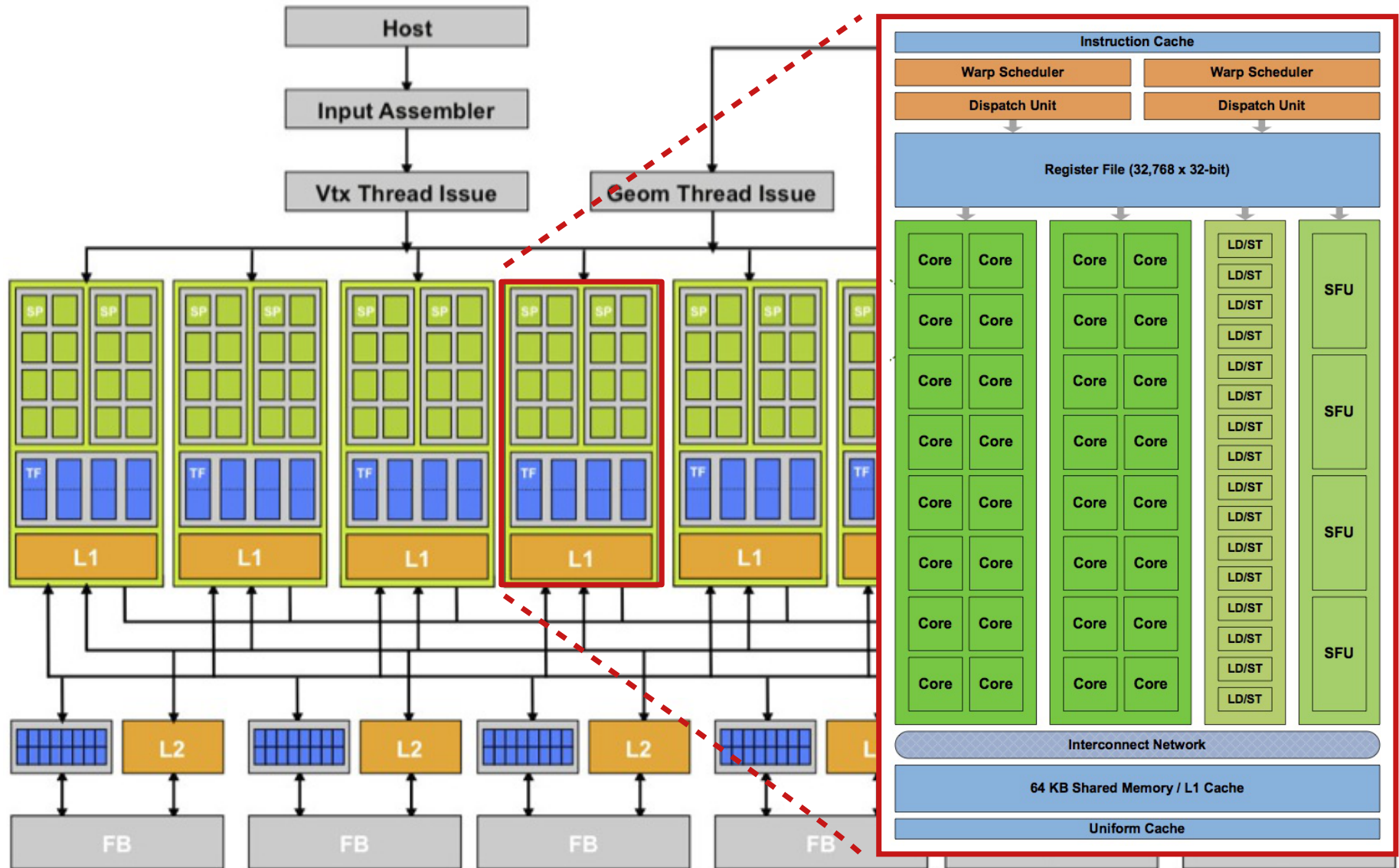
# FG-SIMT Cycle-Level Performance Results



# FG-SIMT Energy-Performance Results

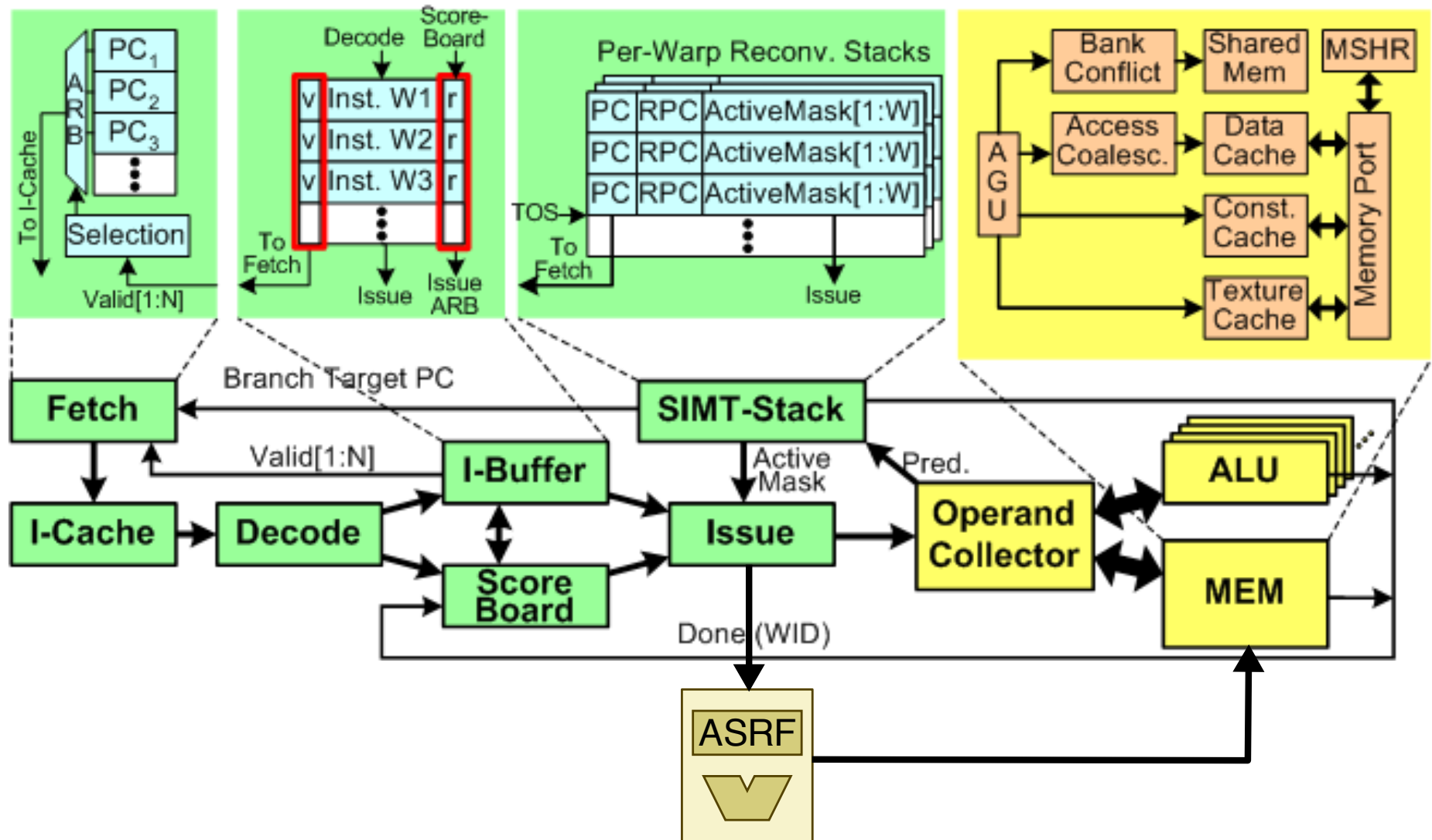


# General-Purpose SIMT Microarchitecture





# Exploiting Value Structure in General-Purpose SIMT



# Take-Away Points

SIMT kernels contain **ample value structure** that is not exploited by current SIMT microarchitectures

Compact affine execution of **affine arithmetic, branches, and memory operations** are a promising way to exploit value structure for improved performance and reduced energy

For more information see our ISCA'13 paper



Christopher  
Torng

Shreesha  
Srinath

Ji  
Kim

Christopher  
Batten

Derek  
Lockhart