Duality Cache

for Data Parallel Acceleration

Daichi Fujiki

Scott Malke

Reetuparna Das

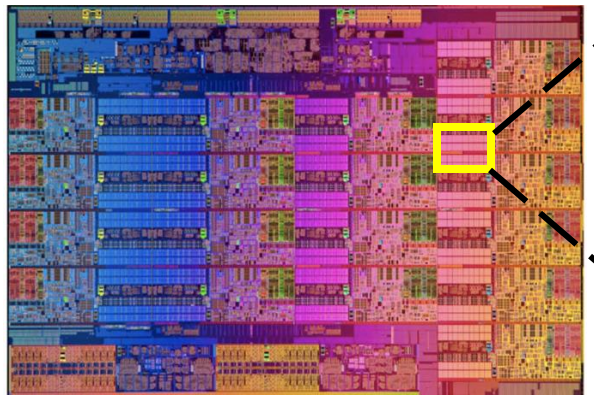
Mbits Research Group
University of Michigan

Duality = Storage + Compute

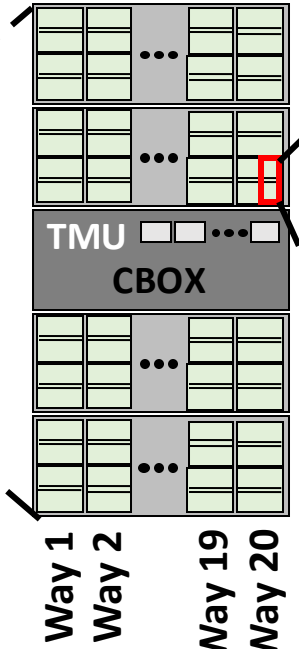
Why compute in-cache ?

Transforming caches into massively parallel vector ALUs

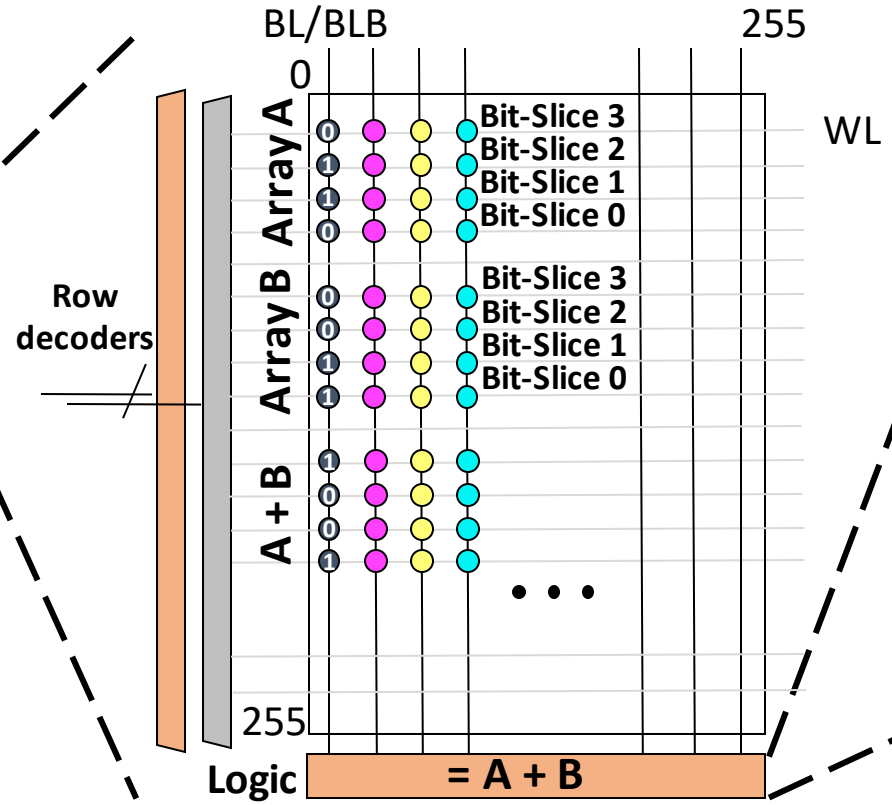
18-core Xeon processor
45 MB LLC



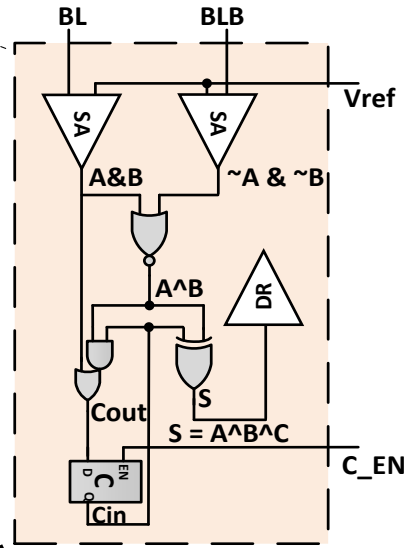
2.5MB LLC slice

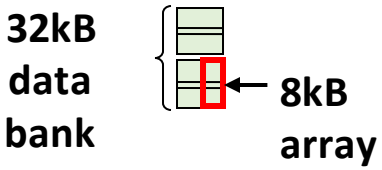


8kB SRAM array



Bitline ALU





18 LLC slices

360 ways

5760 arrays

1,474,560 ALUs

Transforming caches into massively parallel vector ALUs

Passive Last Level Cache transformed into ~ 1 million bit-serial active ALUs

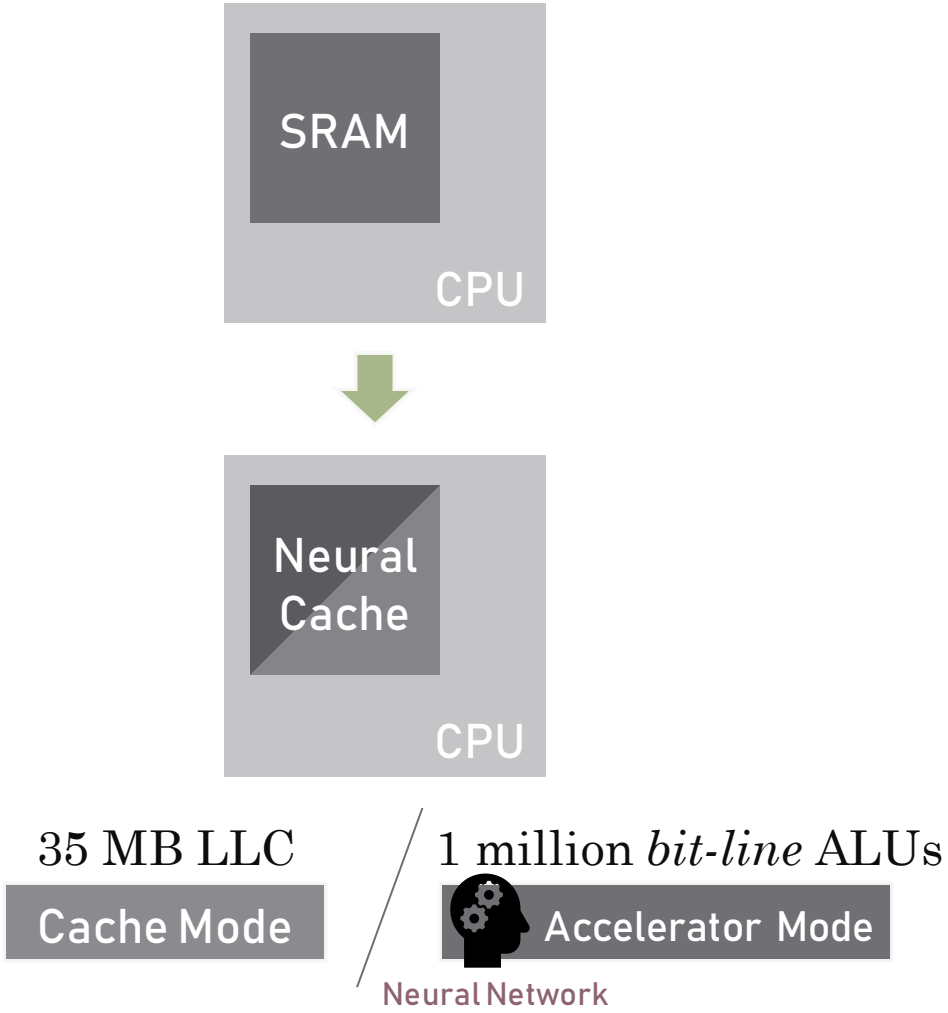
✓ **Multiply** ✓ **Divide** ✓ **Add**

Fixed Point (Configurable Precision)

Bit-serial operation @2.5 GHz

18 LLC slices	**360 ways**	**5760 arrays**	**45 MB LLC → 1,474,560 ALUs**

Neural Cache [ISCA '18]



Digital Bit-Serial Operations @ 2.5GHz

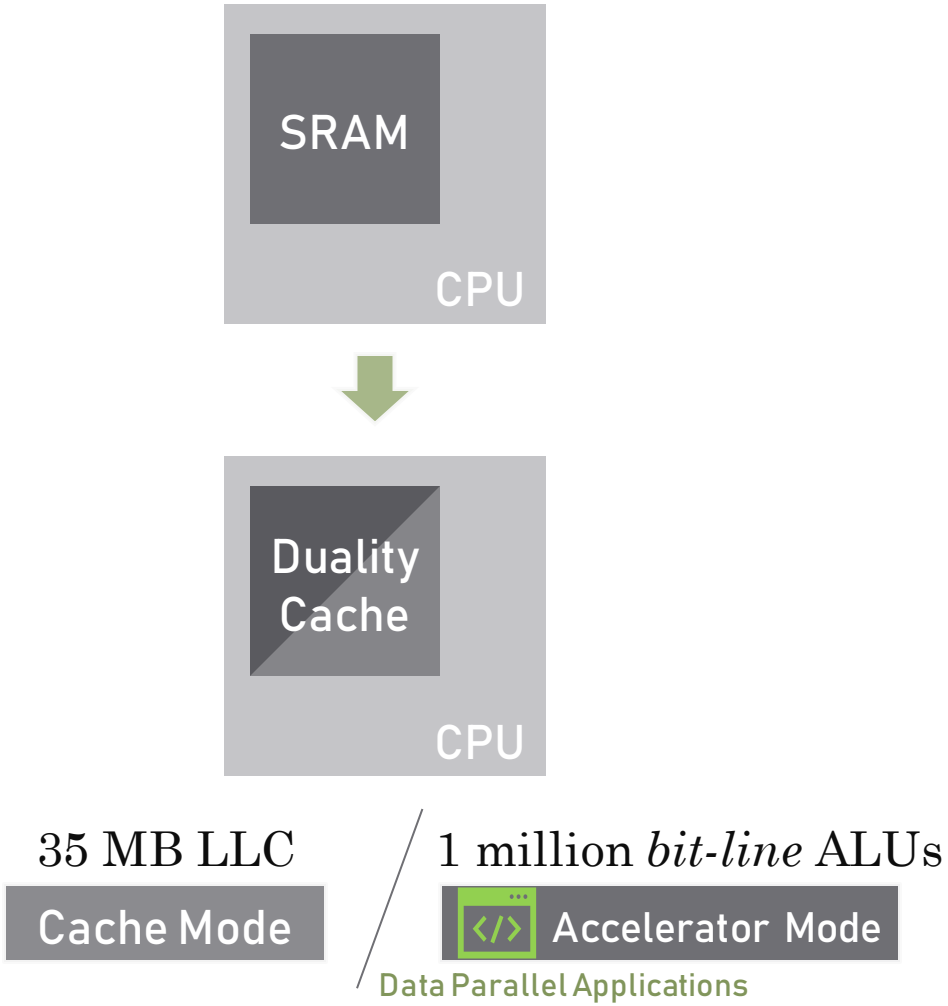
Supported Operations

- ✓ Integer Operations

Programming Model

- ✓ Manually Mapped CNN Kernels

Duality Cache



Digital Bit-Serial Operations @ 2.5GHz

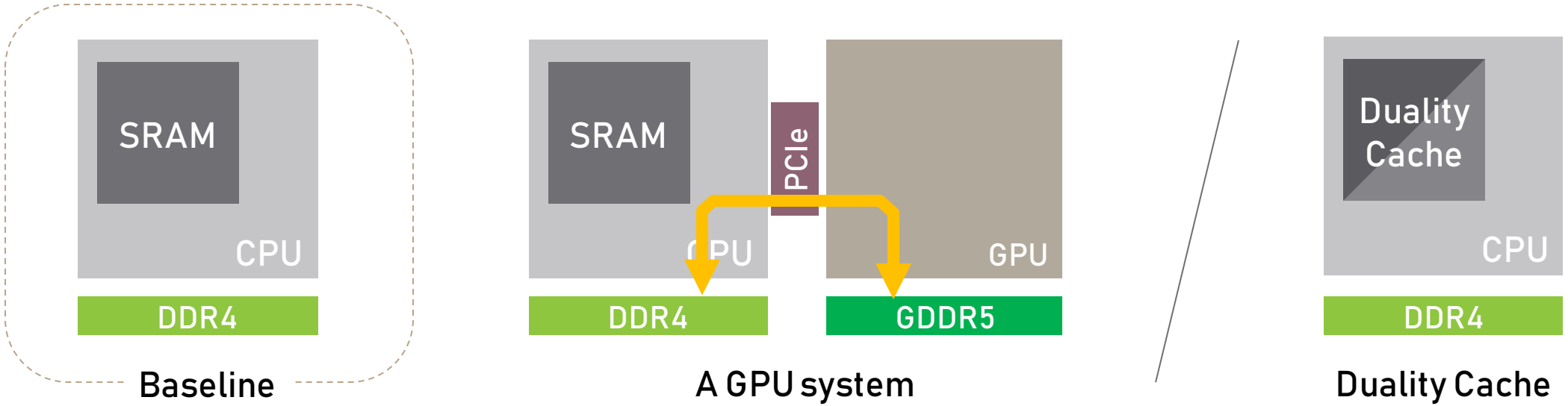
Supported Operations

- ✓ Integer Operations
- ✓ Floating Point Operations
- ✓ Transcendental Functions (sin, cos, log etc.)

Programming Model

- ✓ SIMT & VLIW execution
- ✓ CUDA / OpenACC programs

Duality Cache Benefits



1. Reduced data movement ✓ No memcpy. Efficient serial/parallel interleaving.
2. Cost + 471 mm^2 +250W (GPU) vs. ✓ +30 mm^2 +6W = ~3.5% (DC)
3. On chip memory capacity ✓ ~10x thread capacity. Flexible cache allocation.

Outline

Background

- Integer / Logical Operations

Floating Point Operations

Transcendental Functions

Execution Model

Programming Model

Compiler

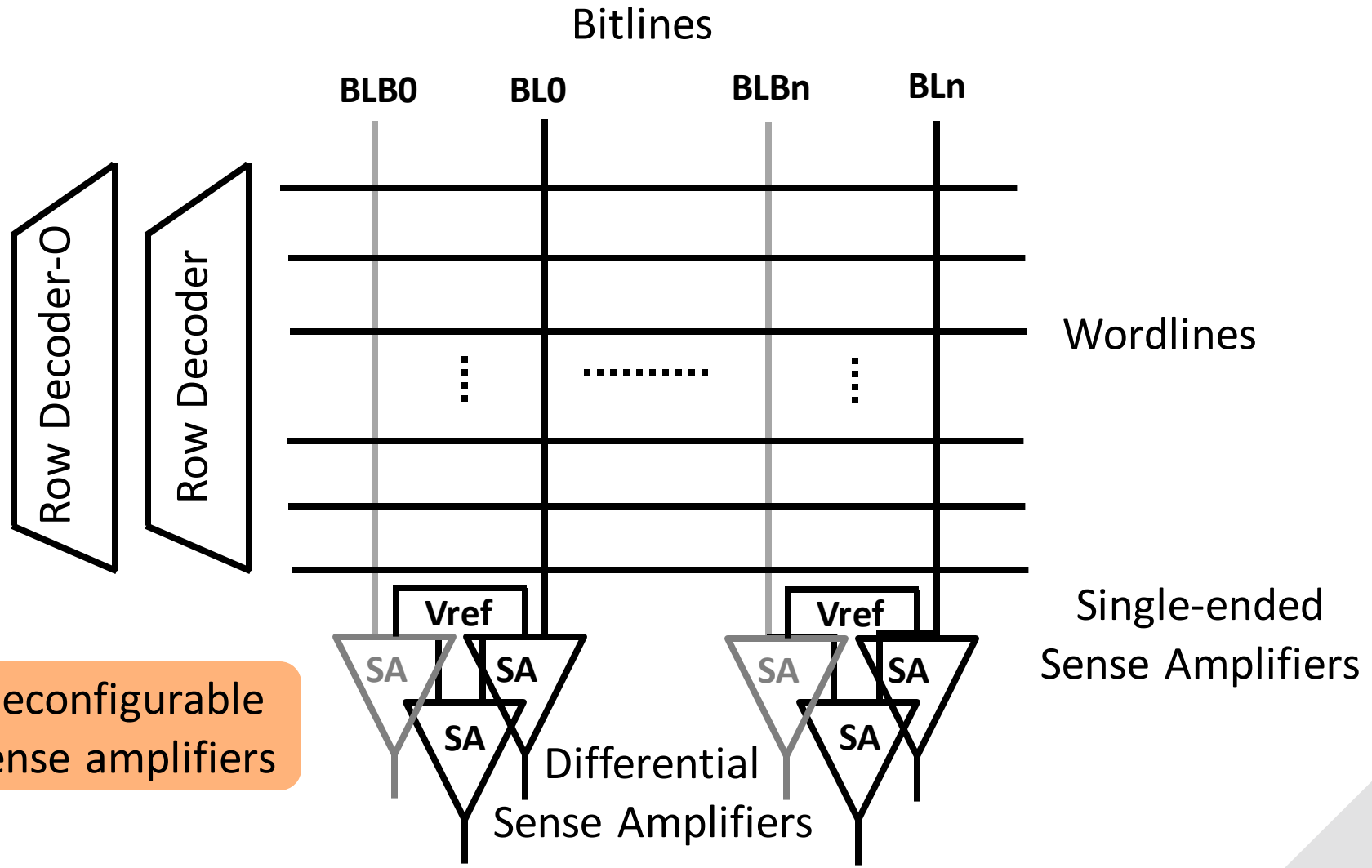
Methodology / Results

Logical Operations In-SRAM

Changes

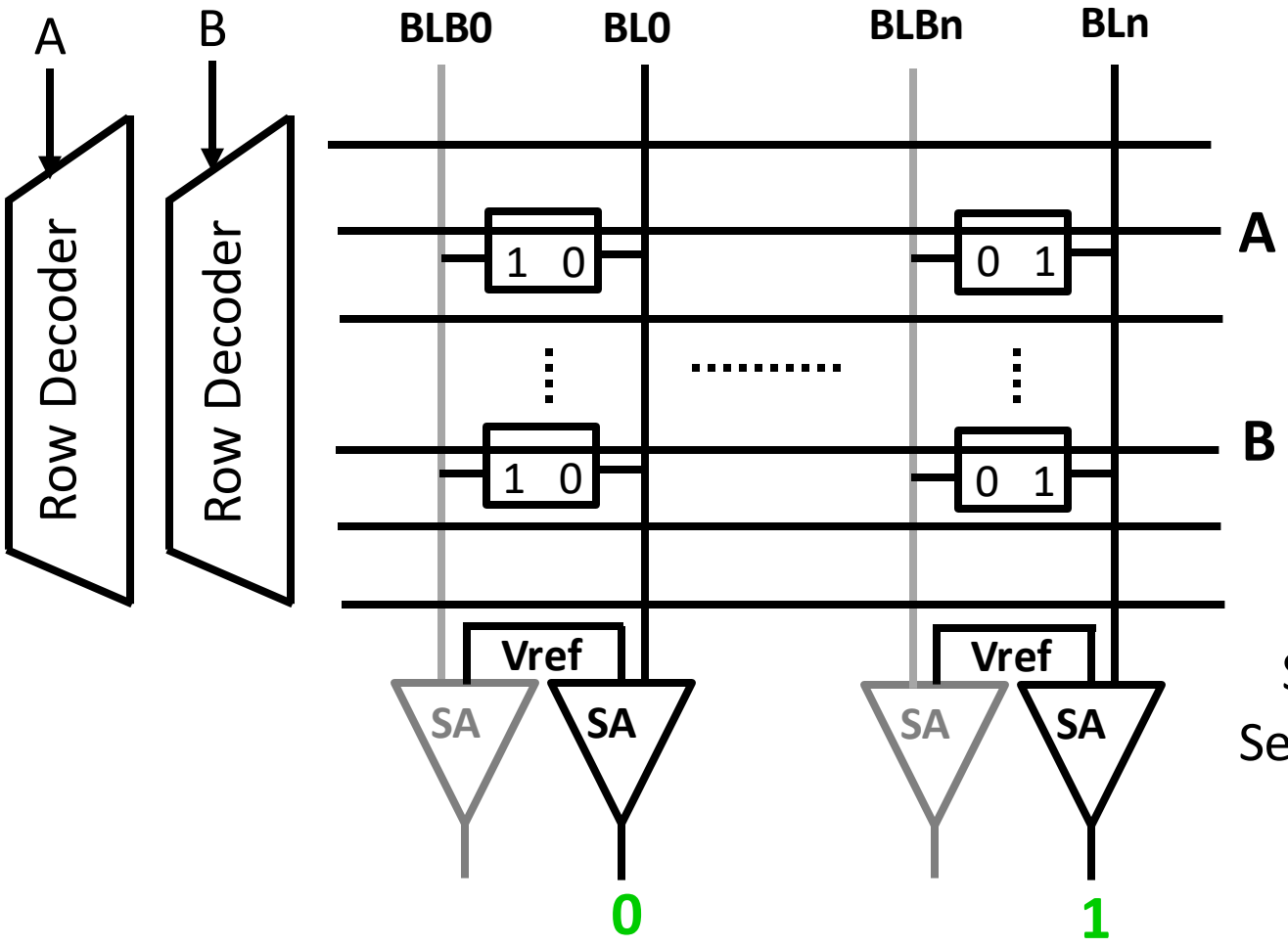
Additional row decoder

Reconfigurable sense amplifiers



Logical Operations In-SRAM

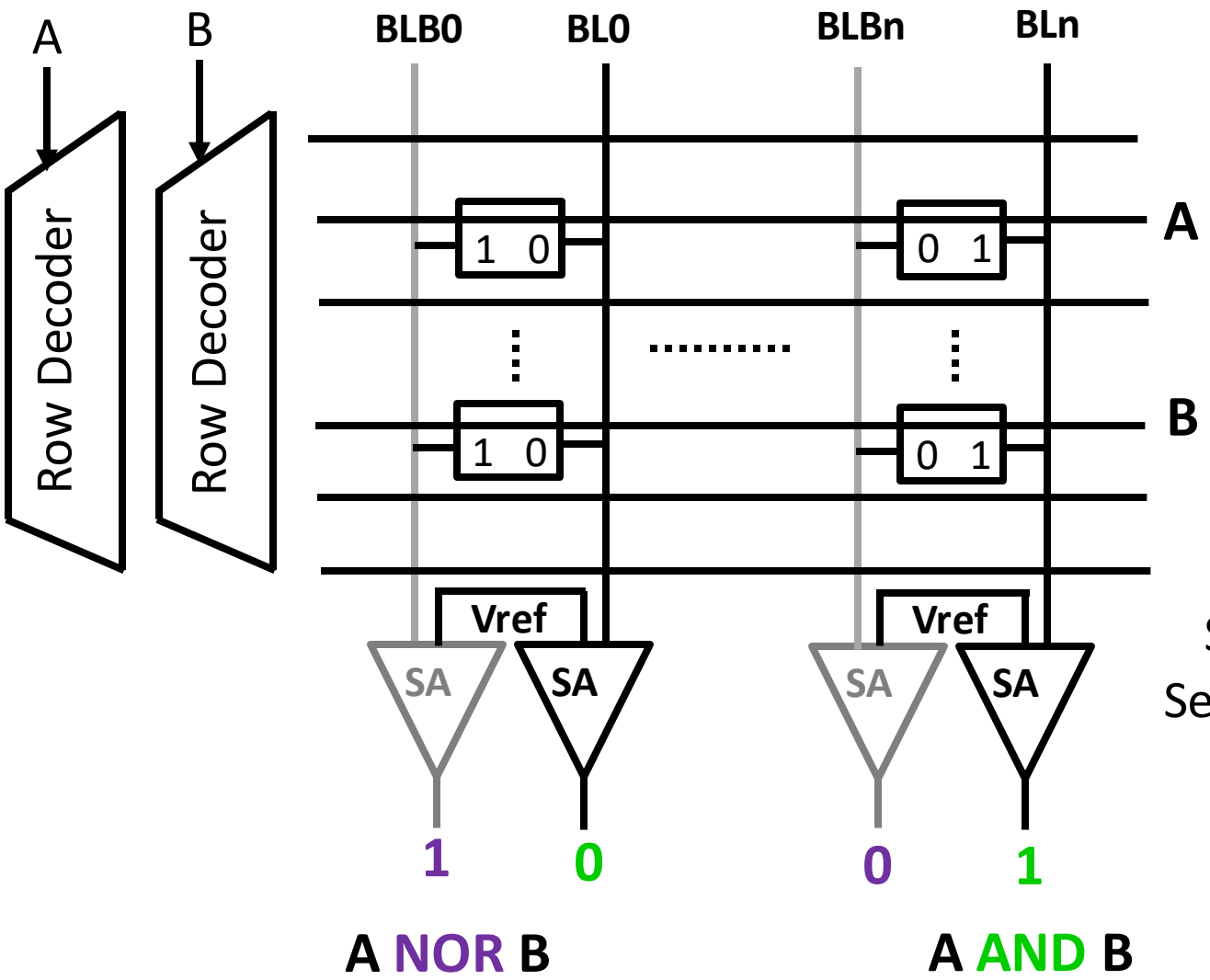
A **AND** B



Single-ended Sense Amplifiers

A AND B

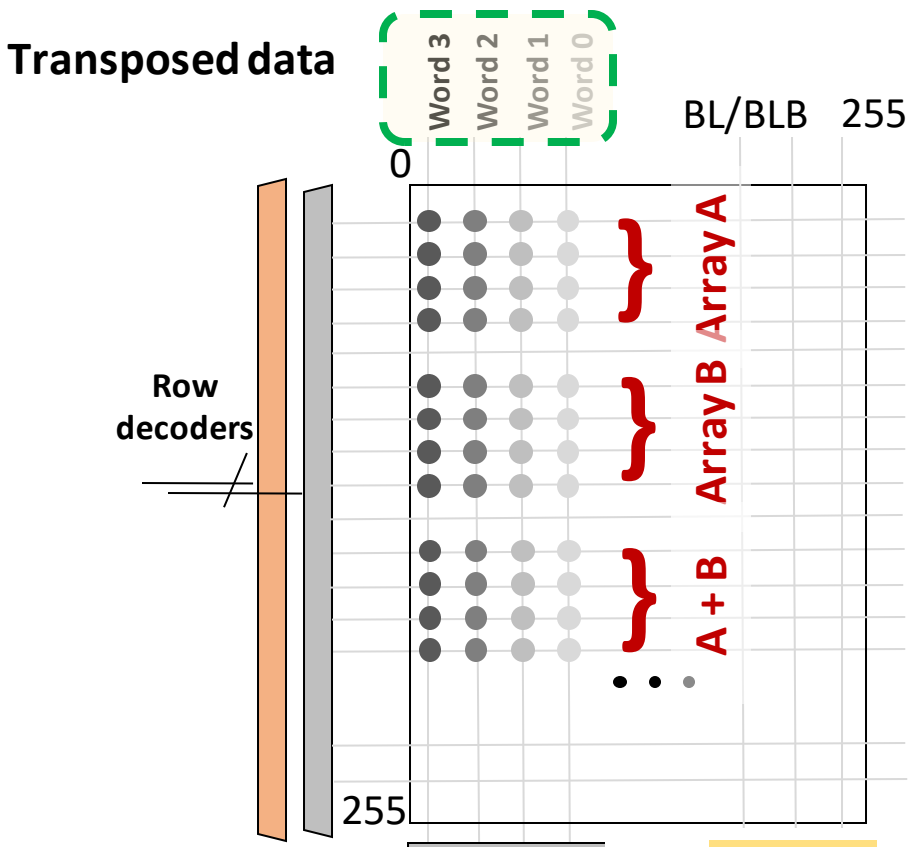
Logical Operations In-SRAM



Single-ended Sense Amplifiers

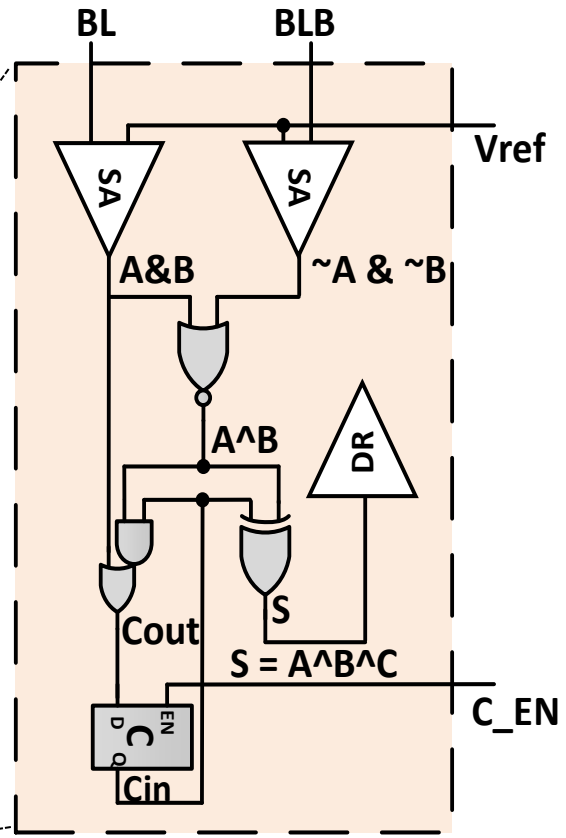
Bit-Serial Integer Operations

A + B



$$s_i = a_i \oplus b_i \oplus c_{i-1}$$

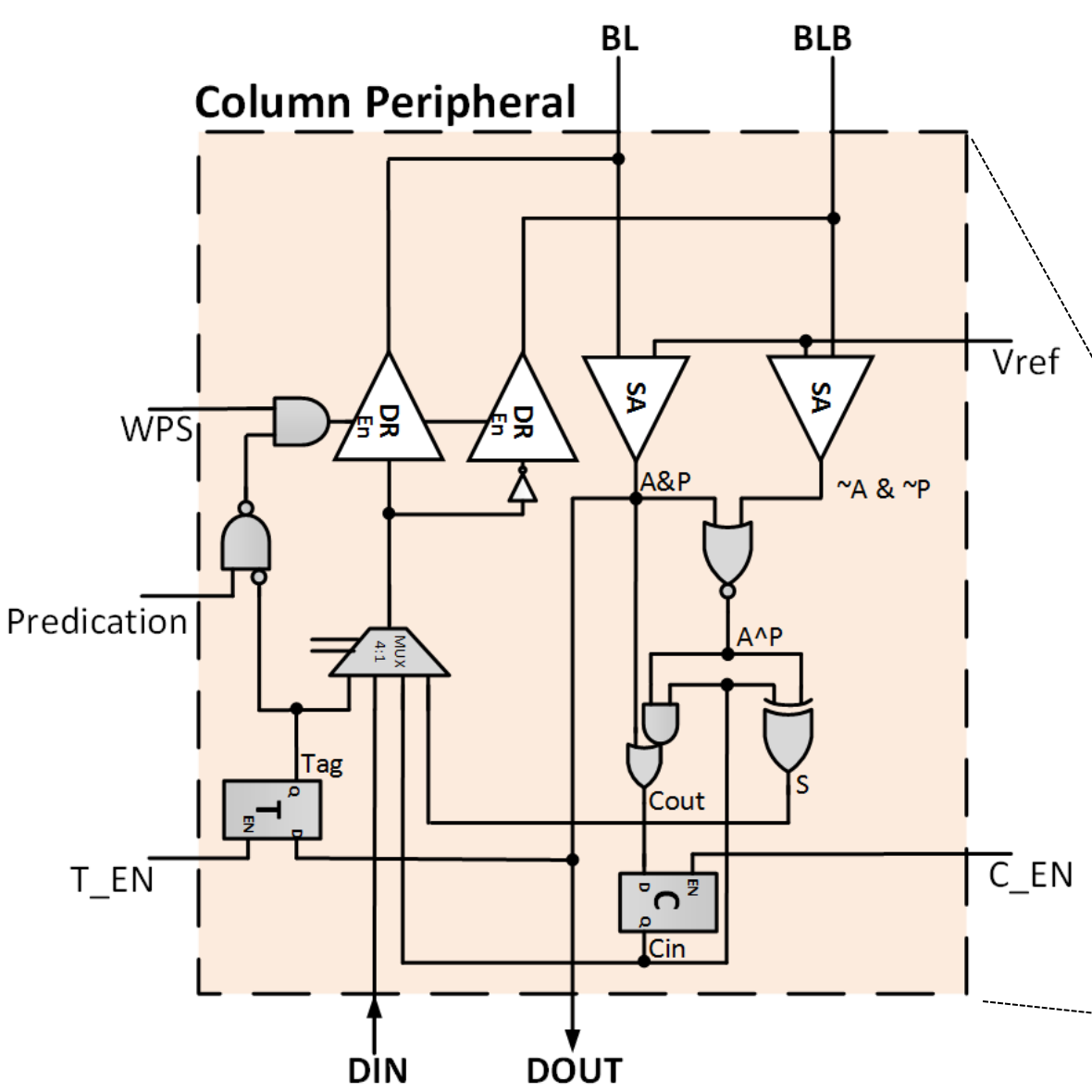
$$c_i = a_i \times b_i + (a_i \oplus b_i) \times c_{i-1}$$

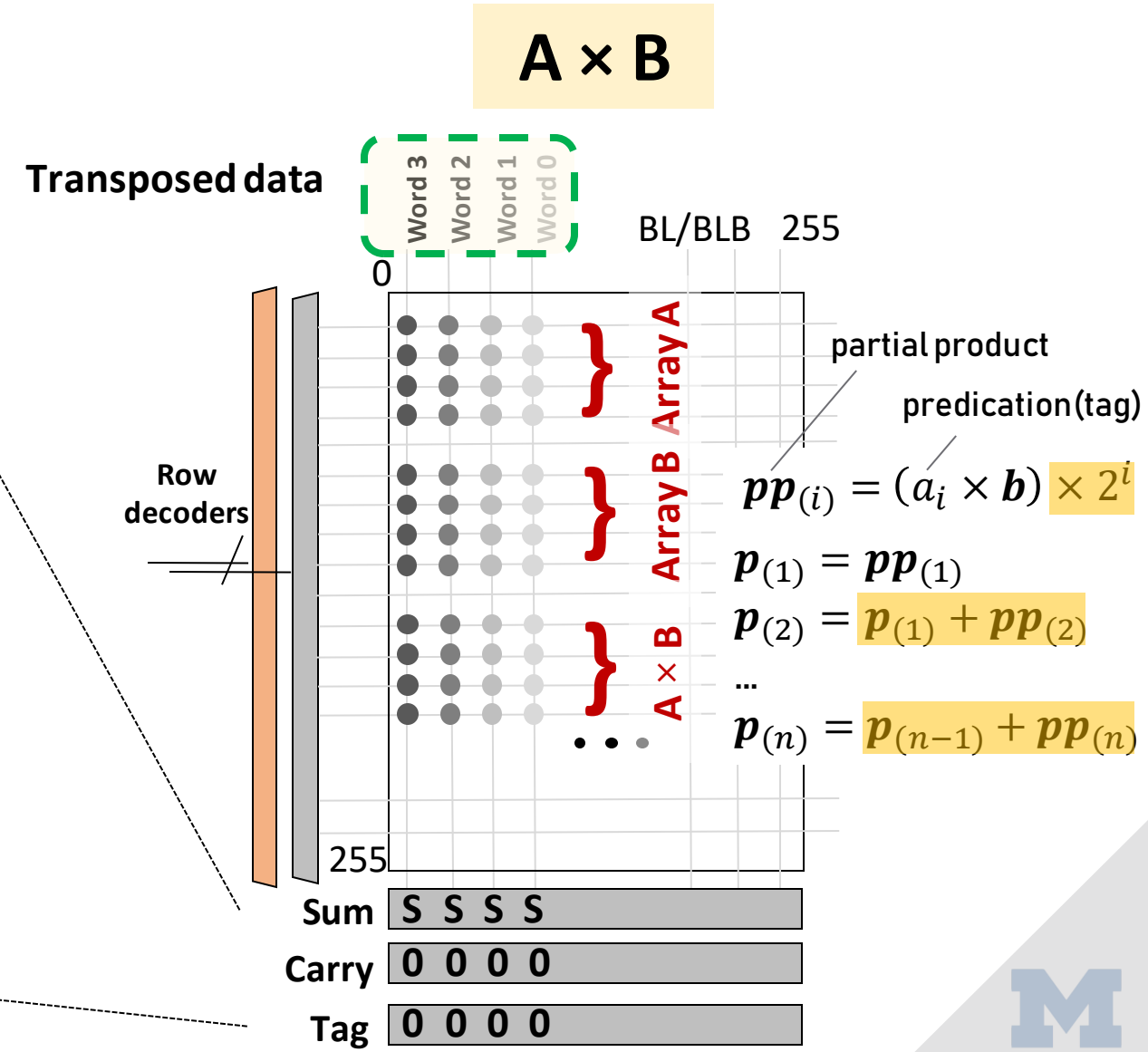


In-memory operations

Bit-Serial Integer Operations

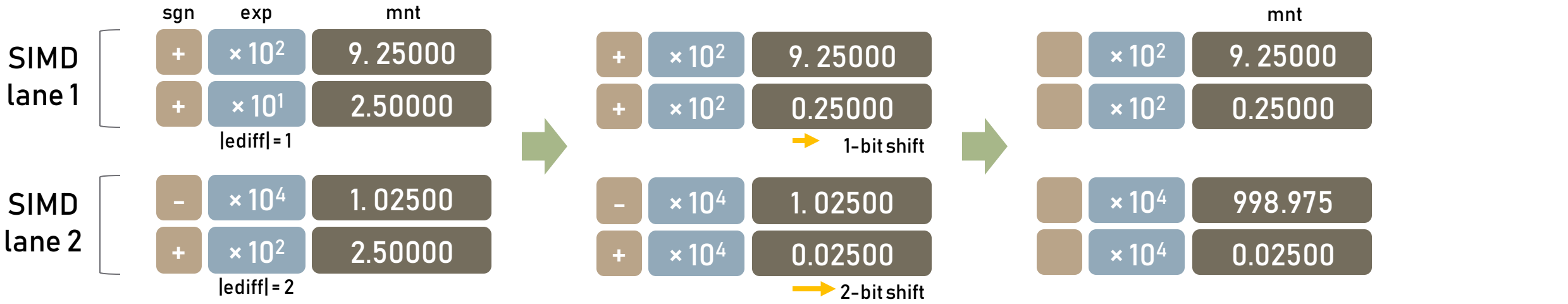
a_i := i^{th} bit of bit vector $\boldsymbol{a}$
$\boldsymbol{a}_{(i)}$:= bit vector $\boldsymbol{a}$ after i^{th} iteration





Bit-Serial Floating Point Operations

A + B



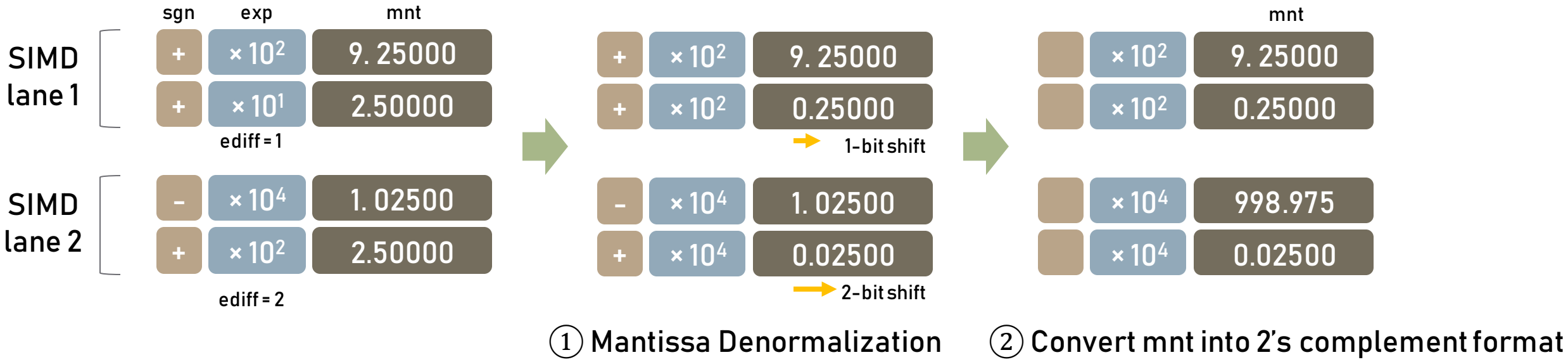
① Mantissa Denormalization ② Convert mnt into 2's complement format

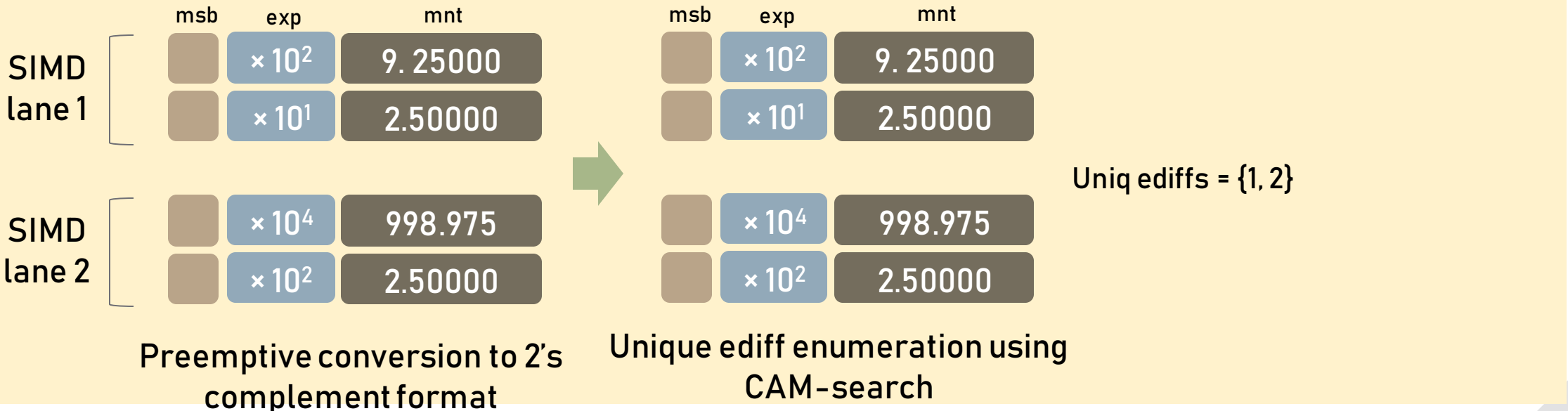
! **Up to mnt_{bit} * #lanes cycles for shift ops**

! **Conversion cost (signed <--> 2's complement)**

Bit-Serial Floating Point Operations

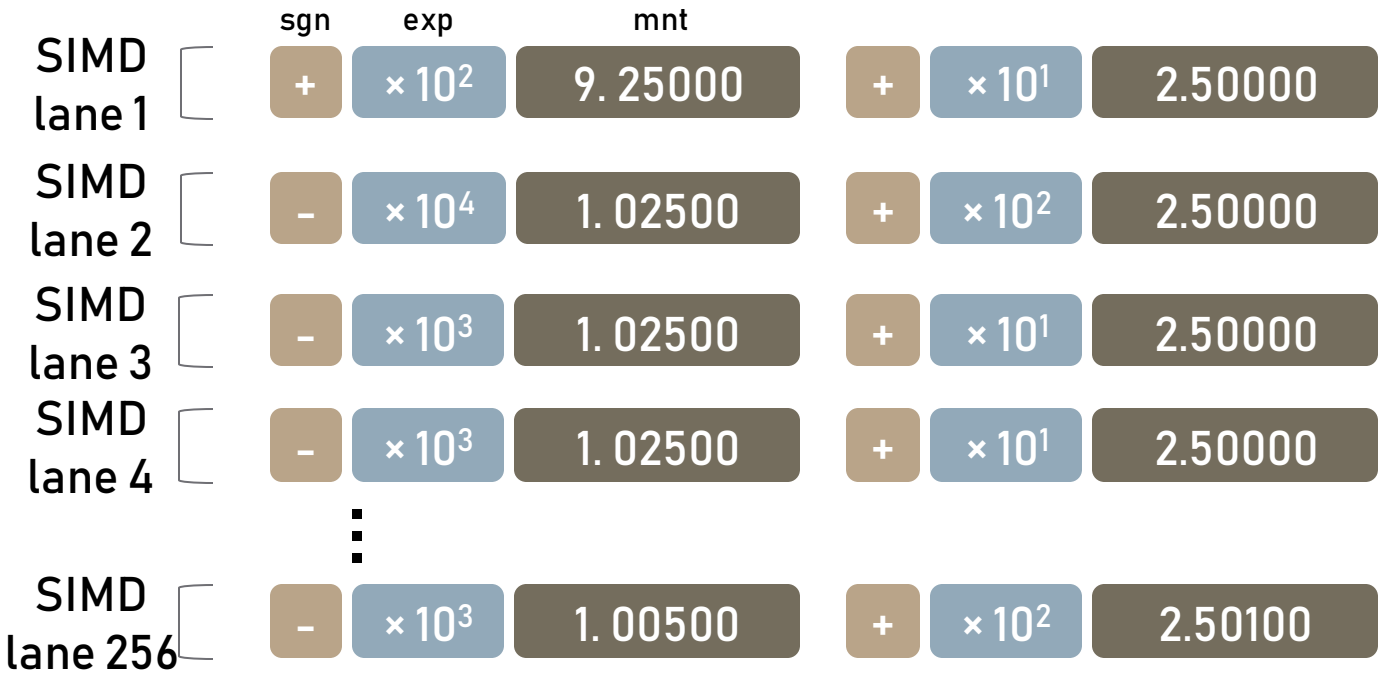
A + B





Bit-Serial Floating Point Operations

A + B



|ediff| = 1 — 8 cycle ediff read + 23 cycle mnt shift

|ediff| = 2 — 8 cycle ediff read + 23 cycle mnt shift

|ediff| = 2 — 8 cycle ediff read + 23 cycle mnt shift

|ediff| = 2 — 8 cycle ediff read + 23 cycle mnt shift

⋮

|ediff| = 1 — 8 cycle ediff read + 23 cycle mnt shift

7,936 cycles (total)

! **Up to mnt_{bit} * #lanes cycles for shift ops**

✓ **Unique ediff enumeration using CAM-search**

Uniq ediffs = {1, 2}

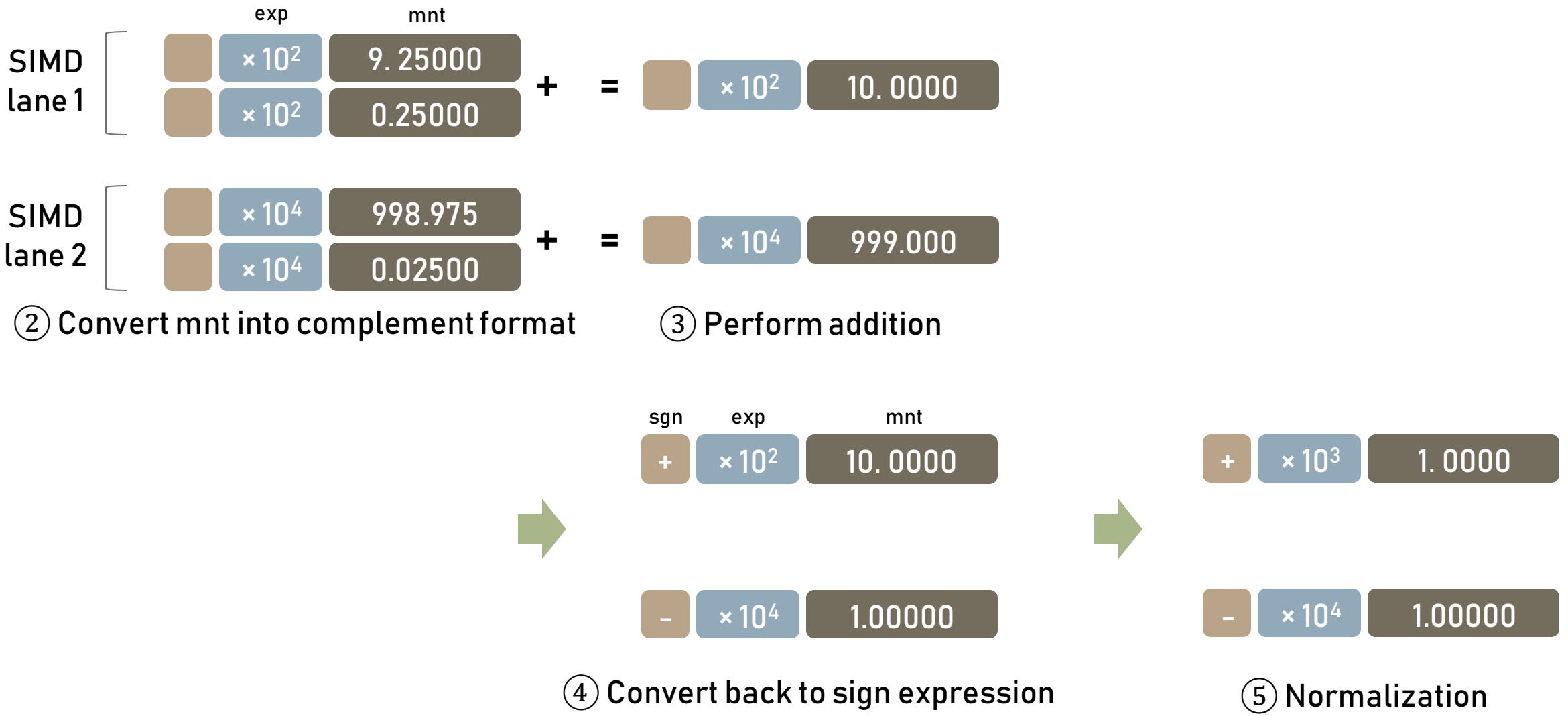
|ediff| = 1 — 23 cycle mnt shift

|ediff| = 2 — 23 cycle mnt shift

+ CAM-search cycle

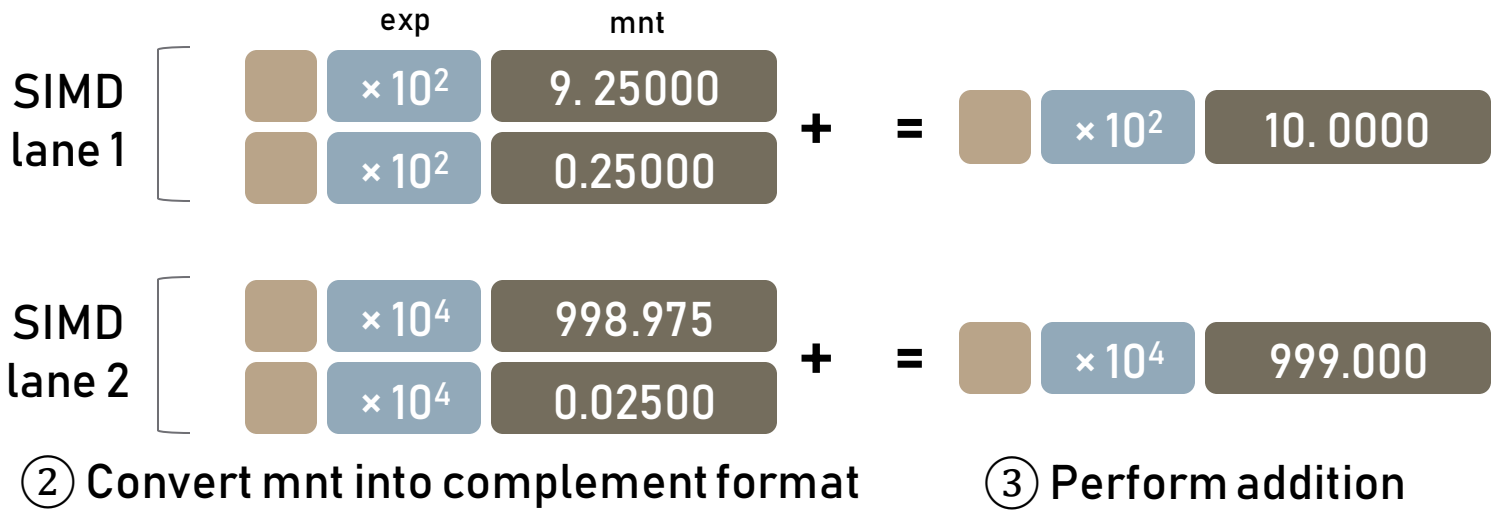
Bit-Serial Floating Point Operations

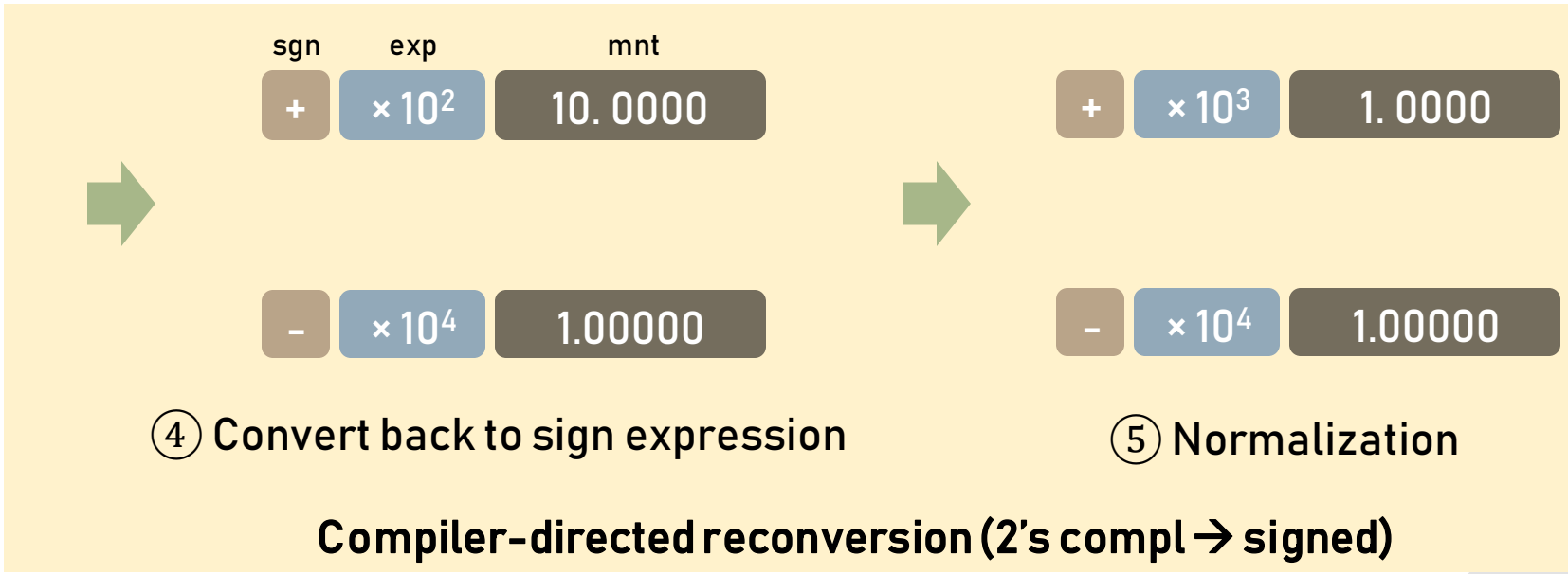
A + B



Bit-Serial Floating Point Operations

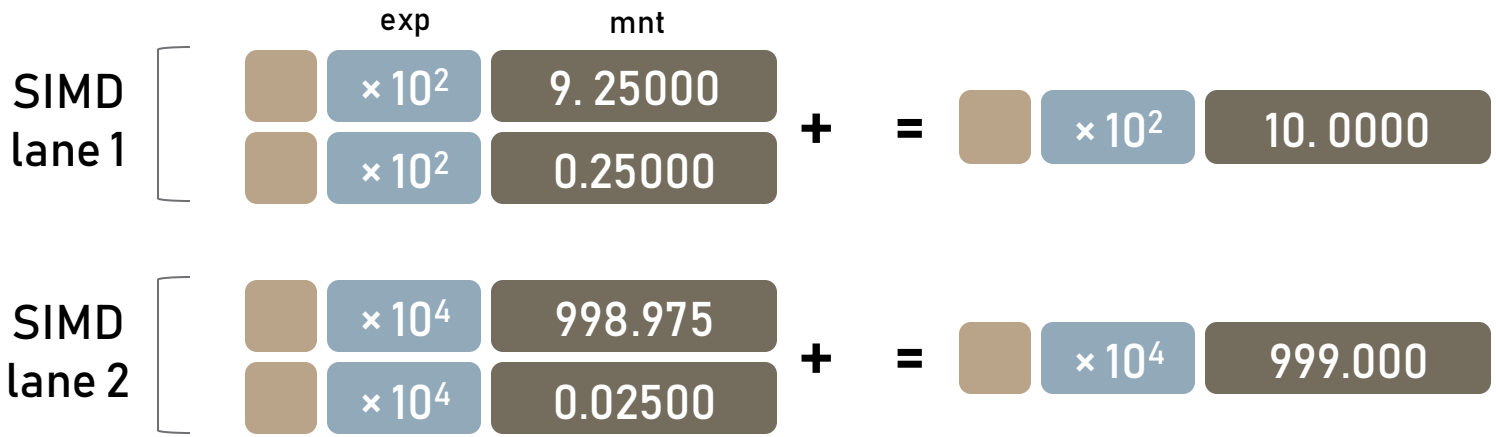
A + B





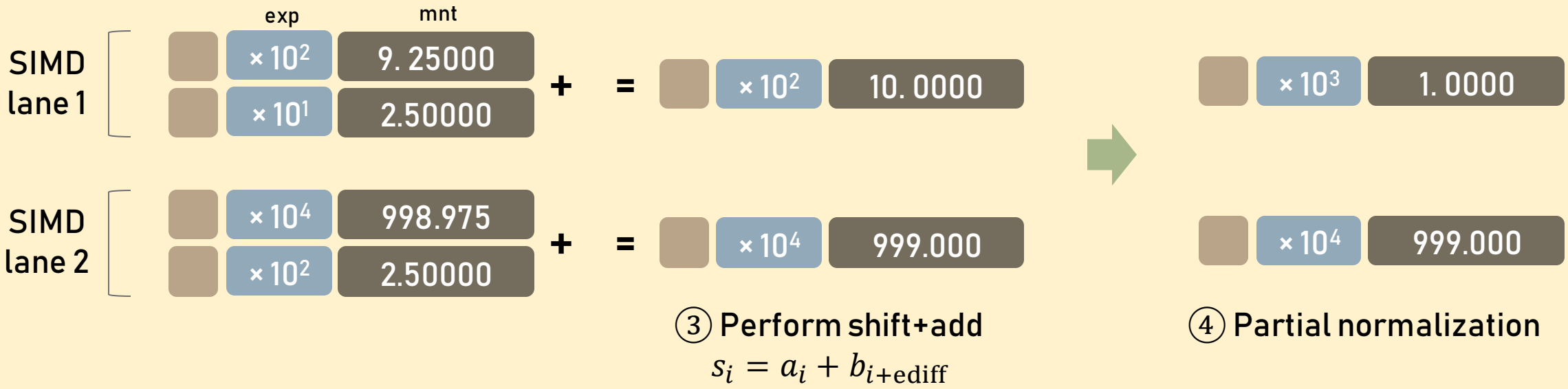
Bit-Serial Floating Point Operations

A + B



② Convert mnt into complement format

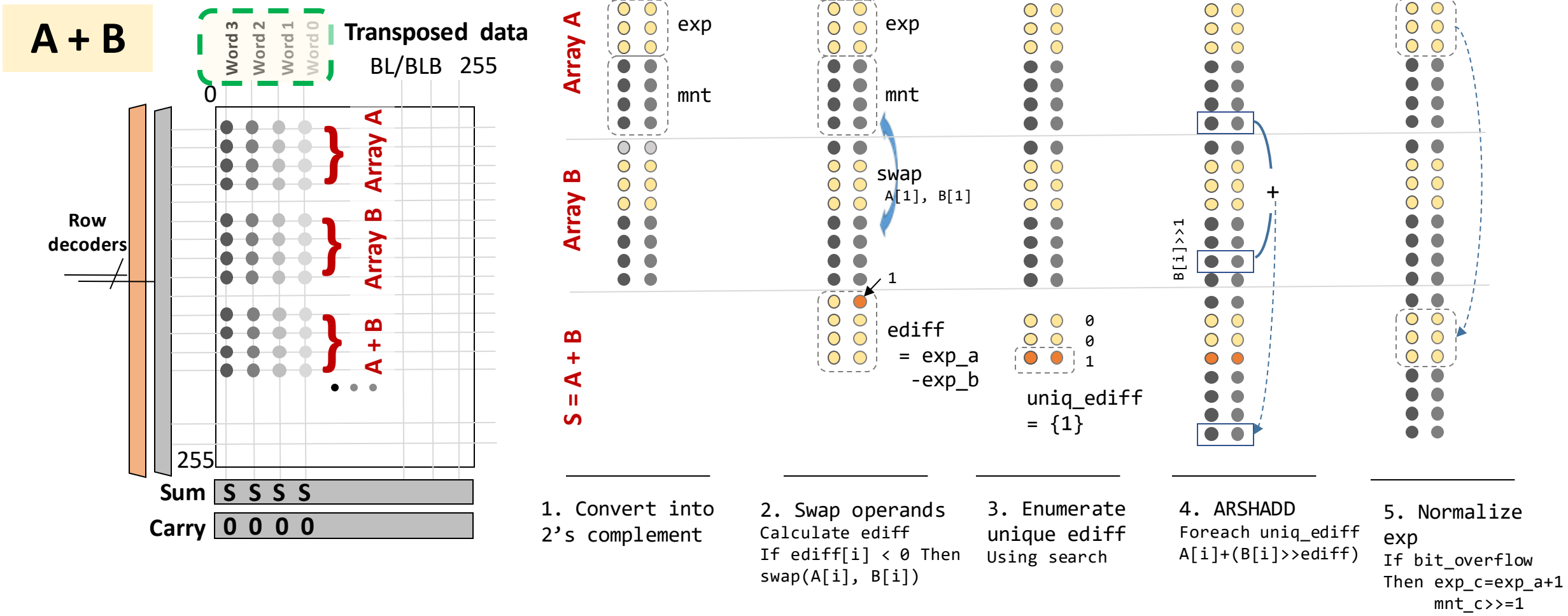
③ Perform addition



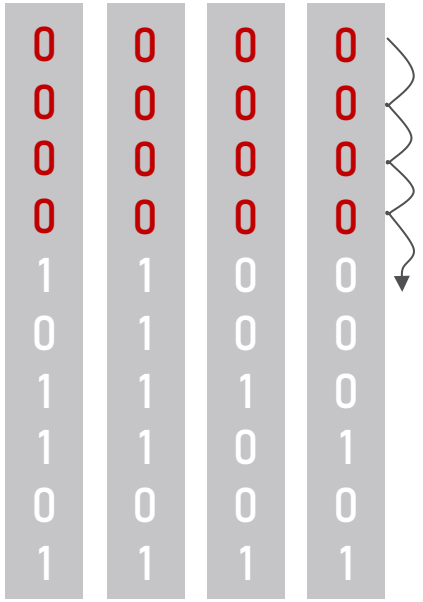
③ Perform shift+add

$$s_i = a_i + b_{i+\text{ediff}}$$

④ Partial normalization


A + B
Word 3
Word 2
Word 1
Word 0
Transposed data
BL/BLB 255
0
Array A
Array B
A + B
Row decoders
255
Sum S S S S
Carry 0 0 0 0
Array A
Array B
S = A + B
sgn
exp
mnt
msb
exp
mnt
swap
A[1], B[1]
1
ediff
= exp_a
-exp_b
0
0
1
uniq_ediff
= {1}
B[i]>>1
+
1. Convert into 2's complement
2. Swap operands
Calculate ediff
If ediff[i] < 0 Then swap(A[i], B[i])
3. Enumerate unique ediff
Using search
4. ARSHADD
Foreach uniq_ediff
A[i]+(B[i]>>ediff)
5. Normalize exp
If bit_overflow
Then exp_c=exp_a+1
mnt_c>>=1

Latency Optimizations (Integer, FP)



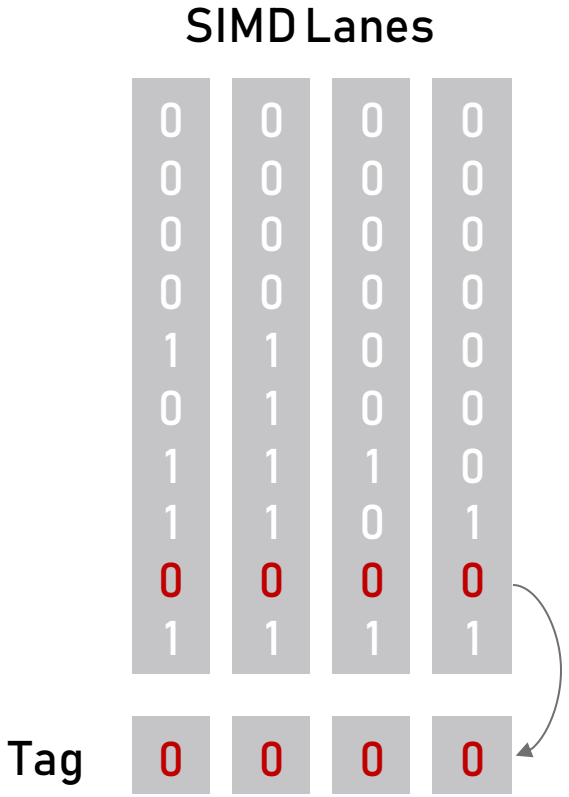
Search leading zeros.

Skip inner loop iterations & reduce inner loop cycles.

$001010_{n_a} \times 000101_{n_b}$

e.g. Multiplication $O(n^2) \rightarrow O(n_a \times n_b)$

Leading Zero Search



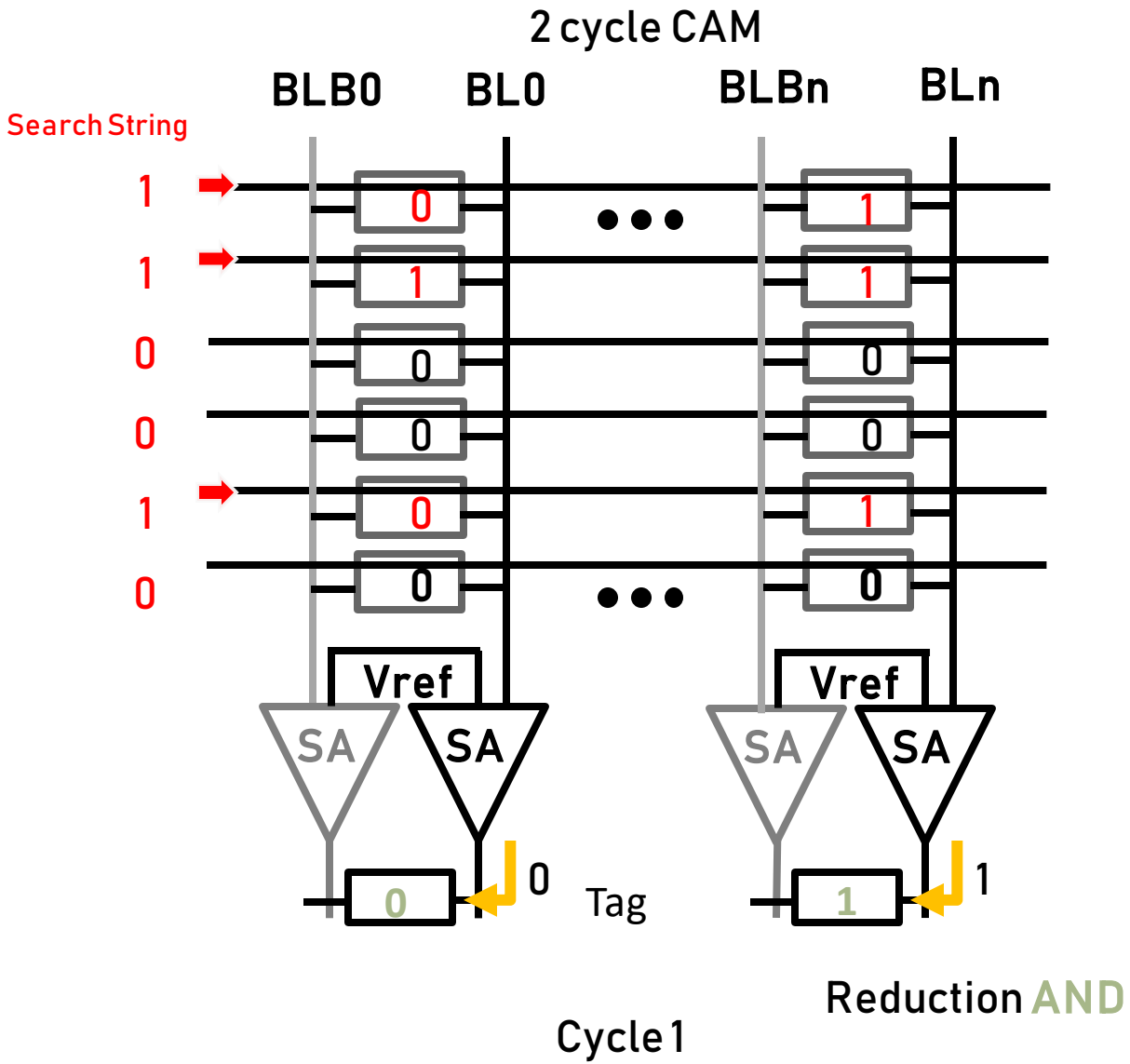
Check tag bits are all zero.

Skip the inner loop iteration.

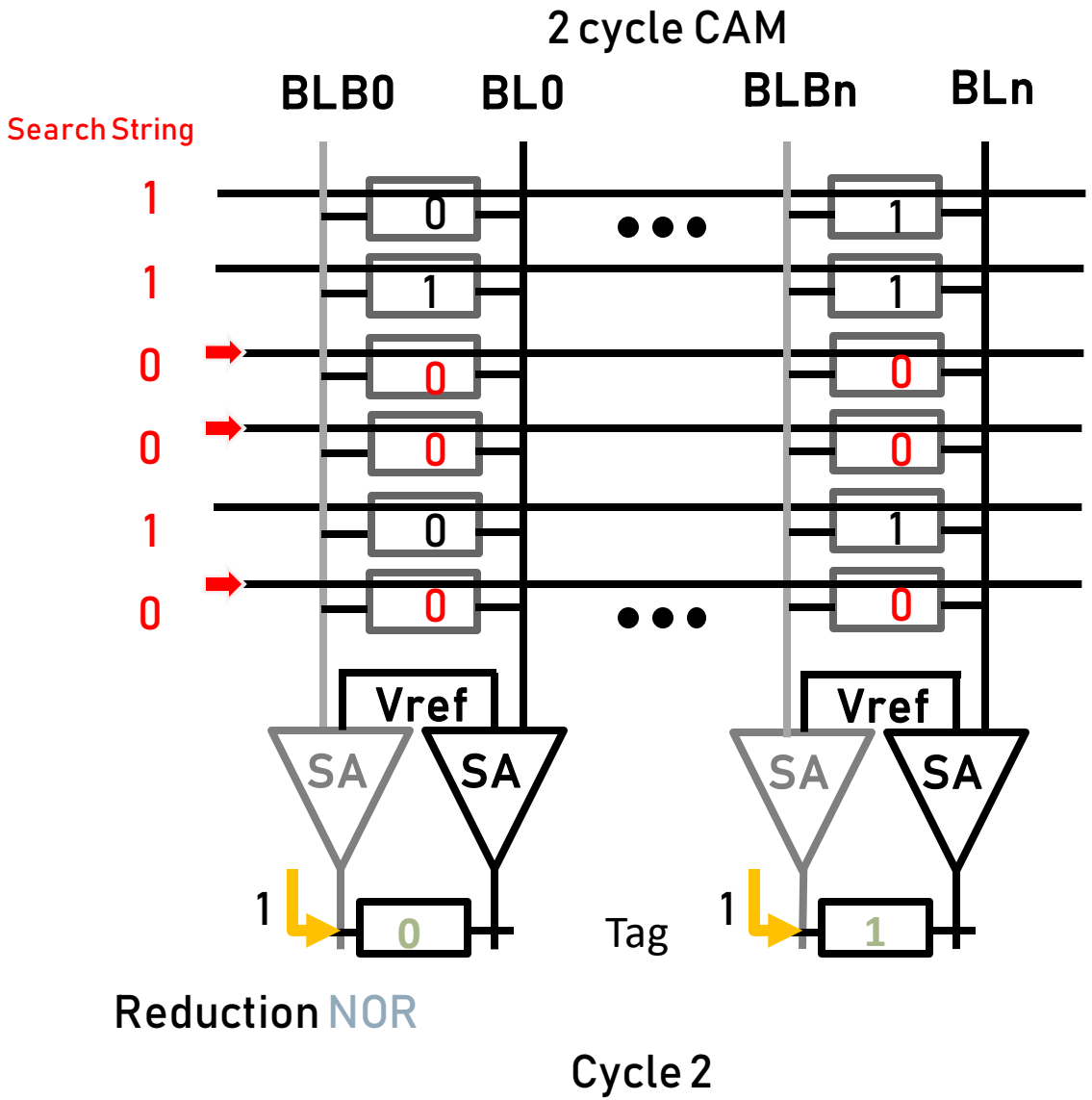
Useful for (const) integer multiplications.

Zero Tag Search

Optimizations with CAM Search



Optimizations with CAM Search



Ediff enumeration

1. Perform leading zero search to limit the search space for CAM search.
   (e.g. $\forall \text{ediff} < 8 \Rightarrow$ 4-bit CAM search)

2. Perform CAM search over ediff vector for values $0 \leq i \leq 2^n$.

3. If any hit (wired OR of tag), perform ARSHADD

Supported In-Cache Operations

Operation	Type	Algorithm	Latency	Optimizations
add, sub	uint, int	[2], Bit-serial	O(n)	-
mul	uint, int	[2], Bit-serial	$O(n^2)$	LZS (multiplicand), ZTS
div, rem				visor), ZTS,
and, or, xor				
shl, shr				arch
add, sub				earch
mul				
div				S
sin, cos				RDIC
exp				RDIC
log	fixed point	CORDIC	O(nk)	ORDIC
sqrt	fixed point	CORDIC	O(nk)	Parallel CORDIC
rsqrt	float	Fast Inv Sqrt	$O(n^2)$	

Algorithm
[1] Compute Caches (HPCA'17)
[2] Neural Cache (ISCA'18)

Latency
n: Data Bit Length (bit)
k: CORDIC iteration count

Optimizations
LZS: Leading Zero Search
ZTS: Zero Tag Search
ZRS: Zero Residue Search

Transcendental functions using CORDIC

Express an angle θ using add/sub of a series. $(0 \le \theta < \pi/2)$

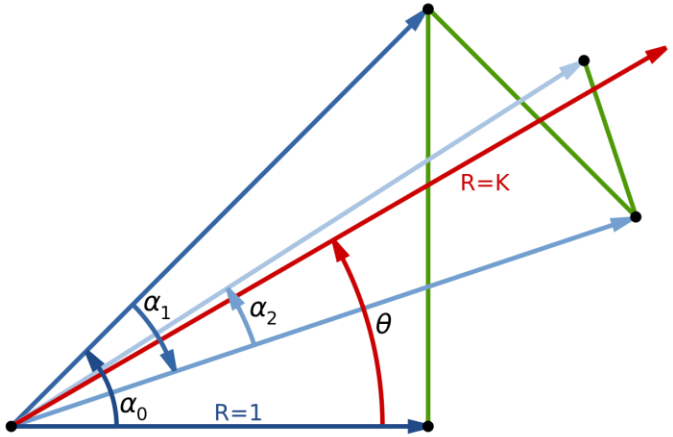
$$\theta = \sum \pm\alpha_i$$

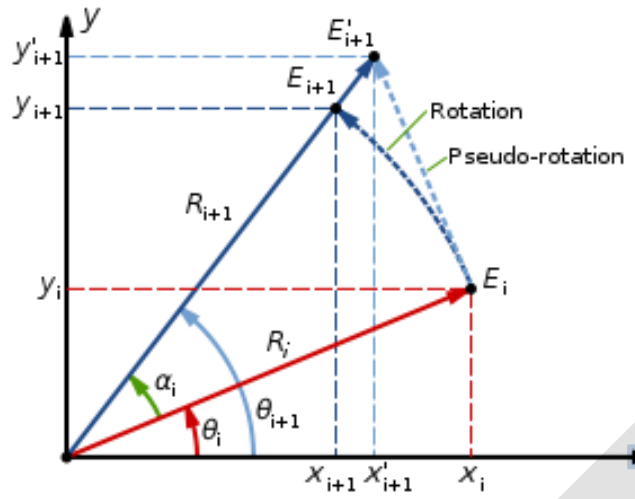
Vector rotation requires multiplications with $\tan(\alpha_i)$.

$$\begin{pmatrix} x_i \\ y_i \end{pmatrix} = \begin{pmatrix} \cos(\alpha_i) & -\sin(\alpha_i) \\ \sin(\alpha_i) & \cos(\alpha_i) \end{pmatrix} \begin{pmatrix} x_{i-1} \\ y_{i-1} \end{pmatrix} = \cos(\alpha_i) \begin{pmatrix} 1 & -\tan(\alpha_i) \\ \tan(\alpha_i) & 1 \end{pmatrix} \begin{pmatrix} x_{i-1} \\ y_{i-1} \end{pmatrix}$$

CORDIC performs pseudo rotation so that $\tan(\alpha_i) = \pm 2^{-i}$

$$\begin{pmatrix} x_i \\ y_i \end{pmatrix} = K \begin{pmatrix} x_{i-1} \mp y_{i-1} \times 2^{-i} \\ y_{i-1} \pm x_{i-1} \times 2^{-1} \end{pmatrix}, \qquad \theta = \sum \pm \arctan 2^{-i}$$





Benefits

- No multiplications.
- Can be combined with Bit-serial algorithms.
- Constant static α_i series for all possible inputs. No LUT required.

  Highly parallelizable for large SIMD processor.
- Operation level parallelism.

Outline

Background

Integer / Logical Operations

Floating Point Operations

Transcendental Functions

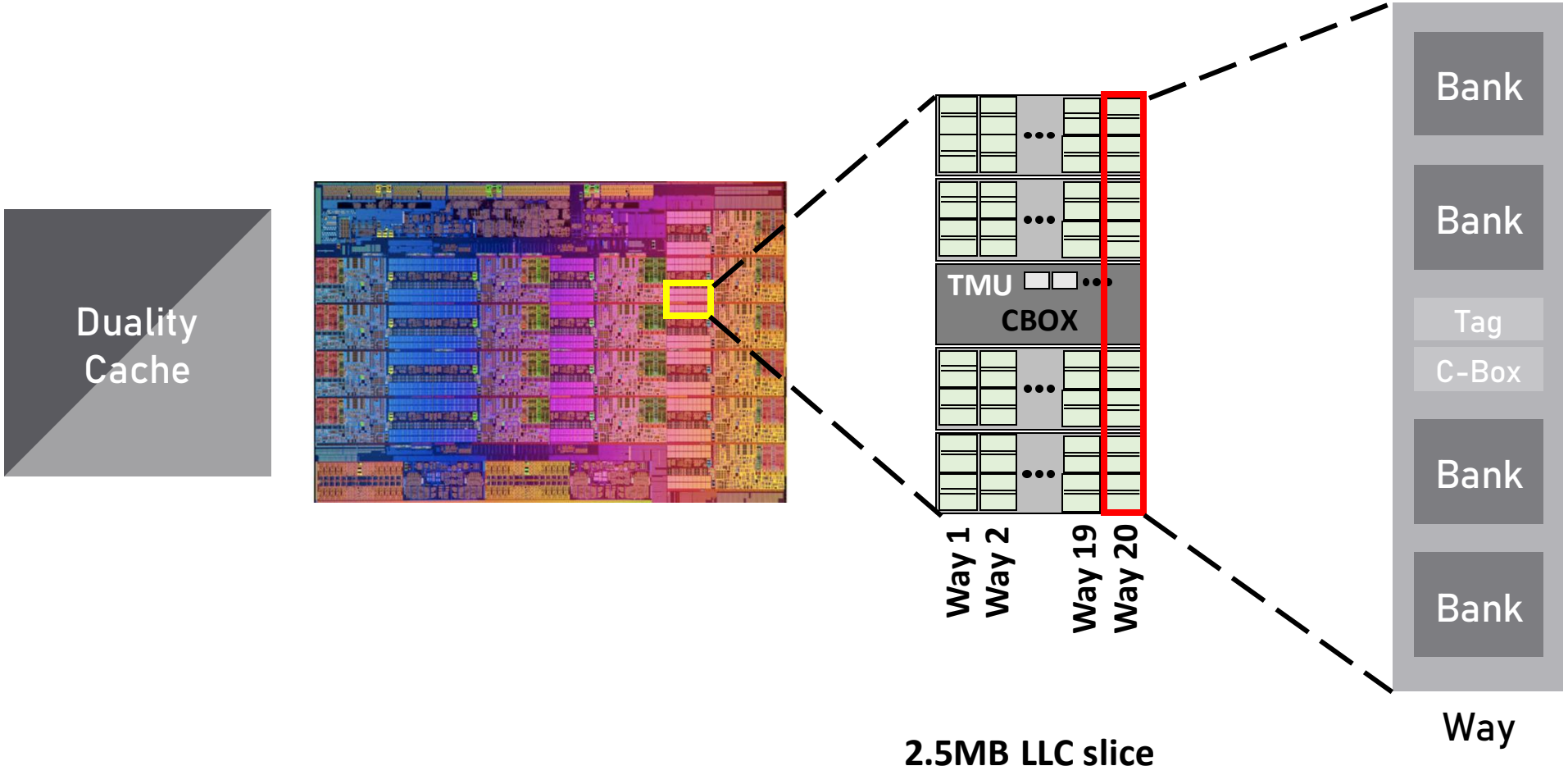
Execution Model

Programming Model

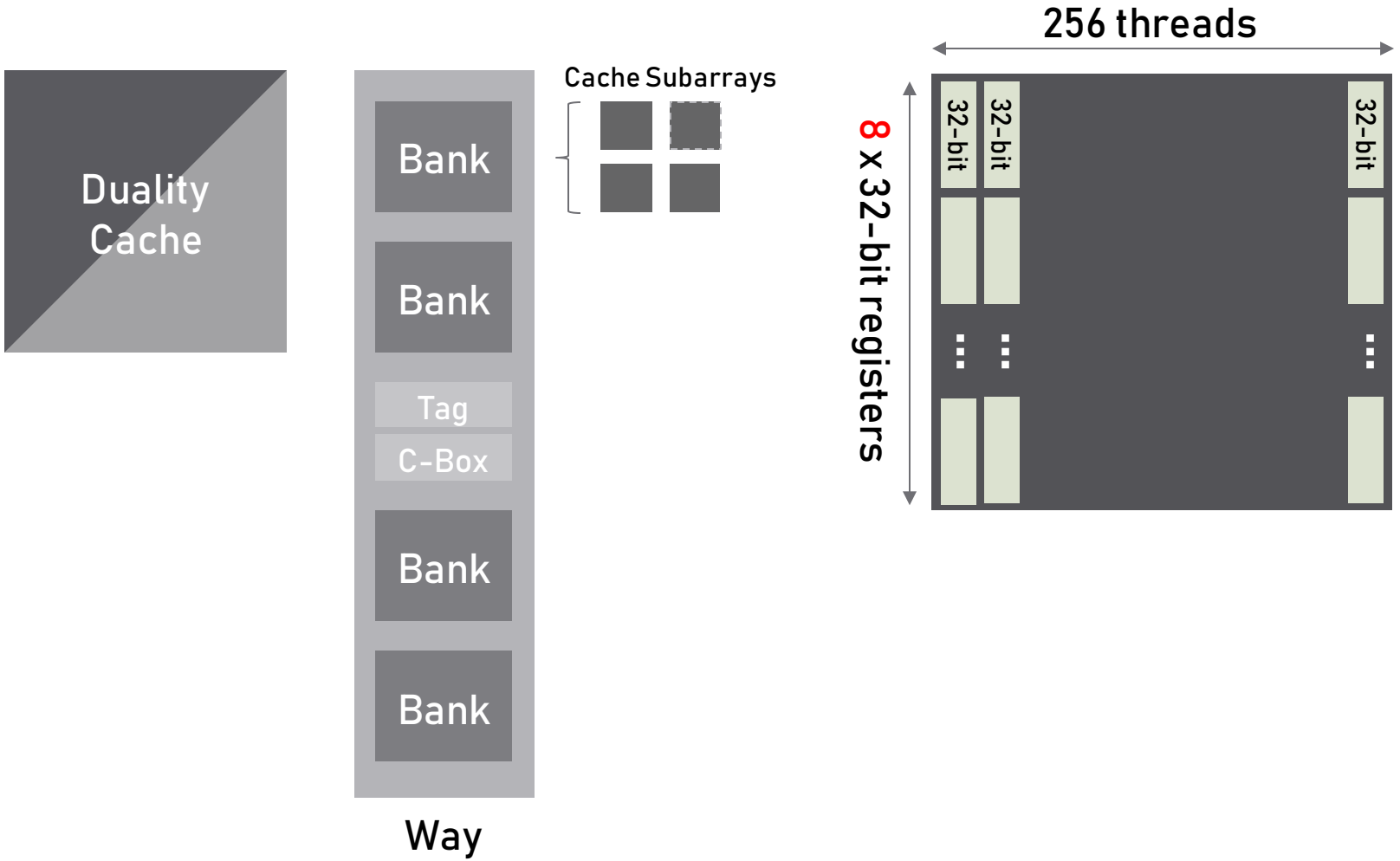
Compiler

Methodology / Results

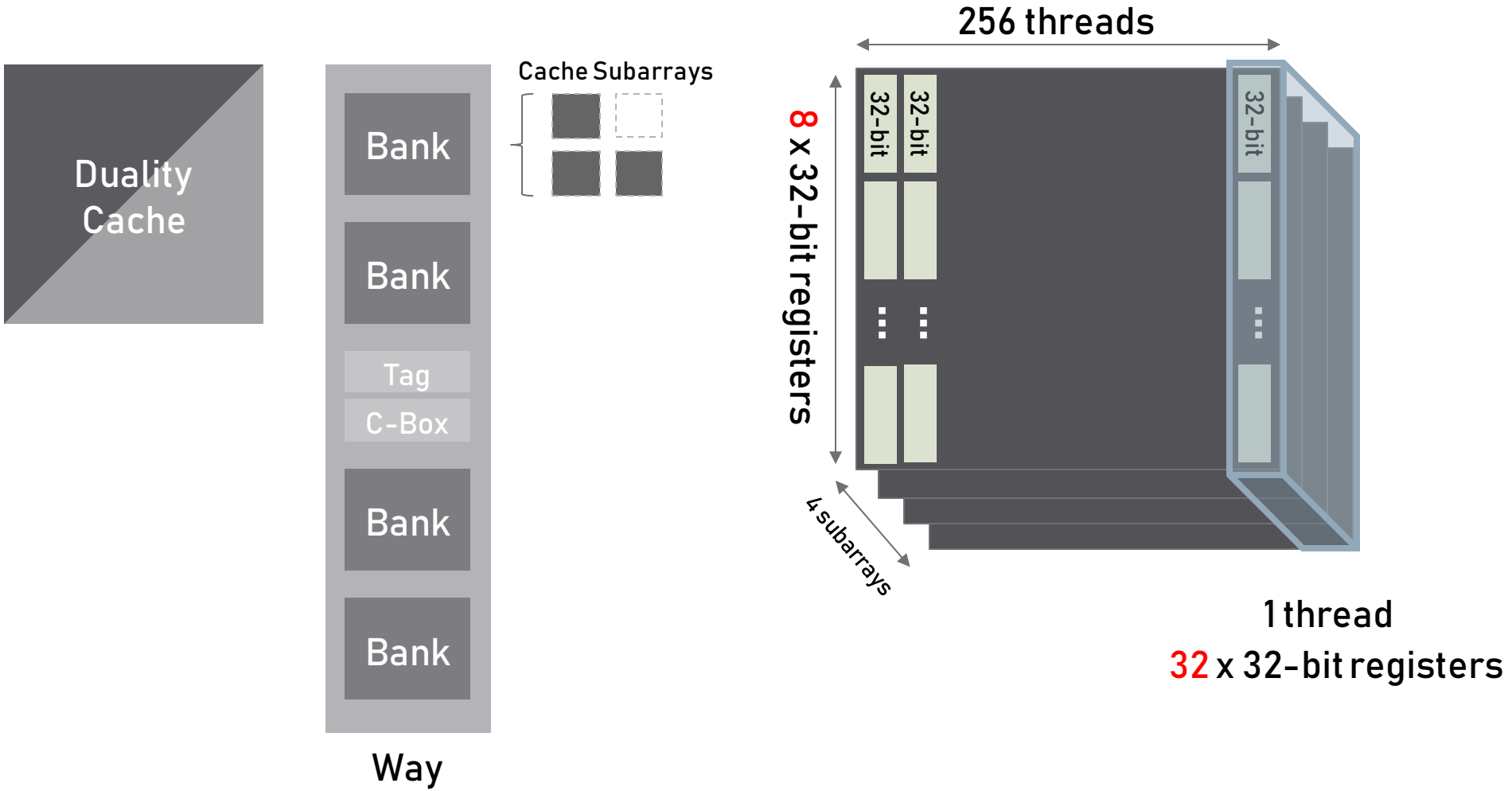
Execution Model



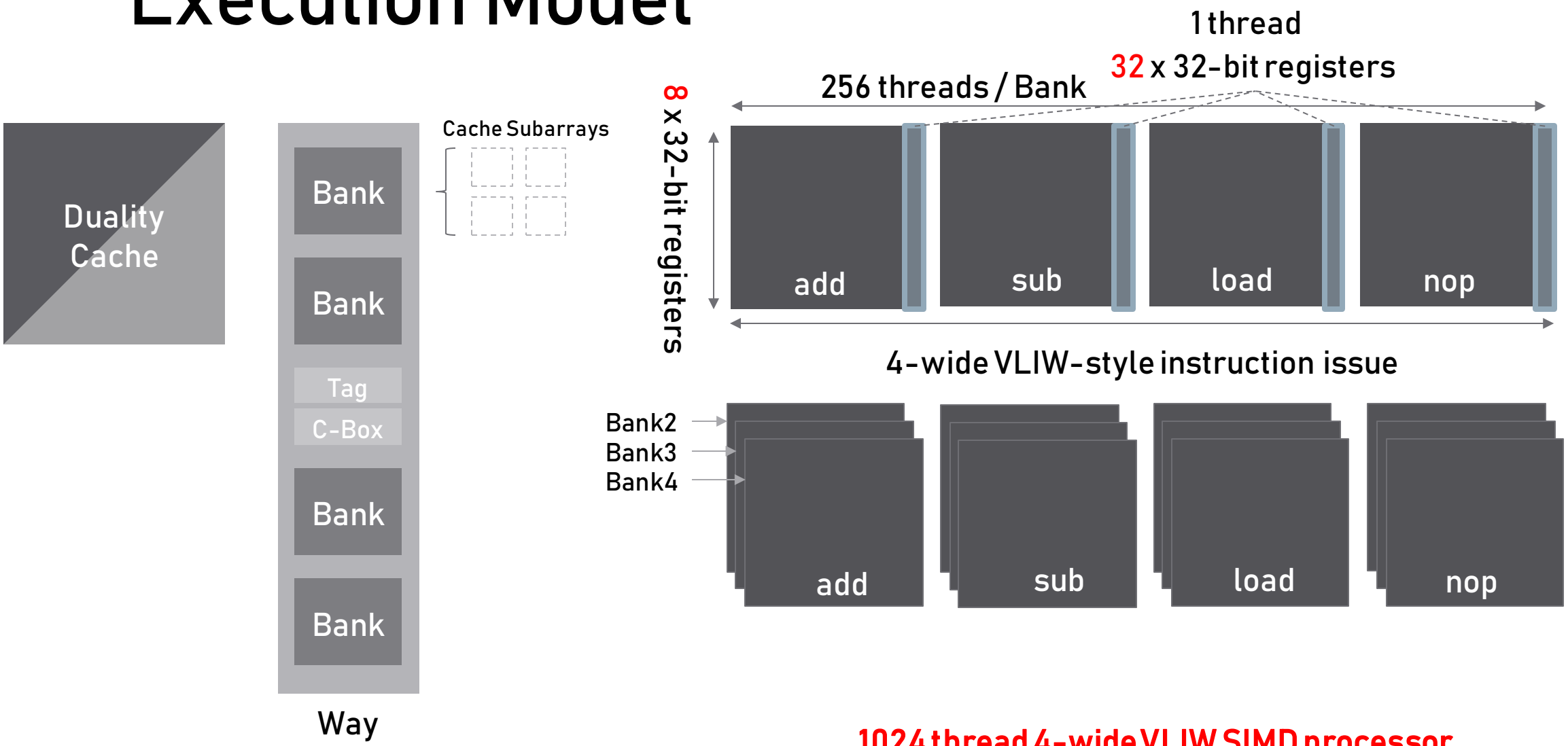
Execution Model



Execution Model

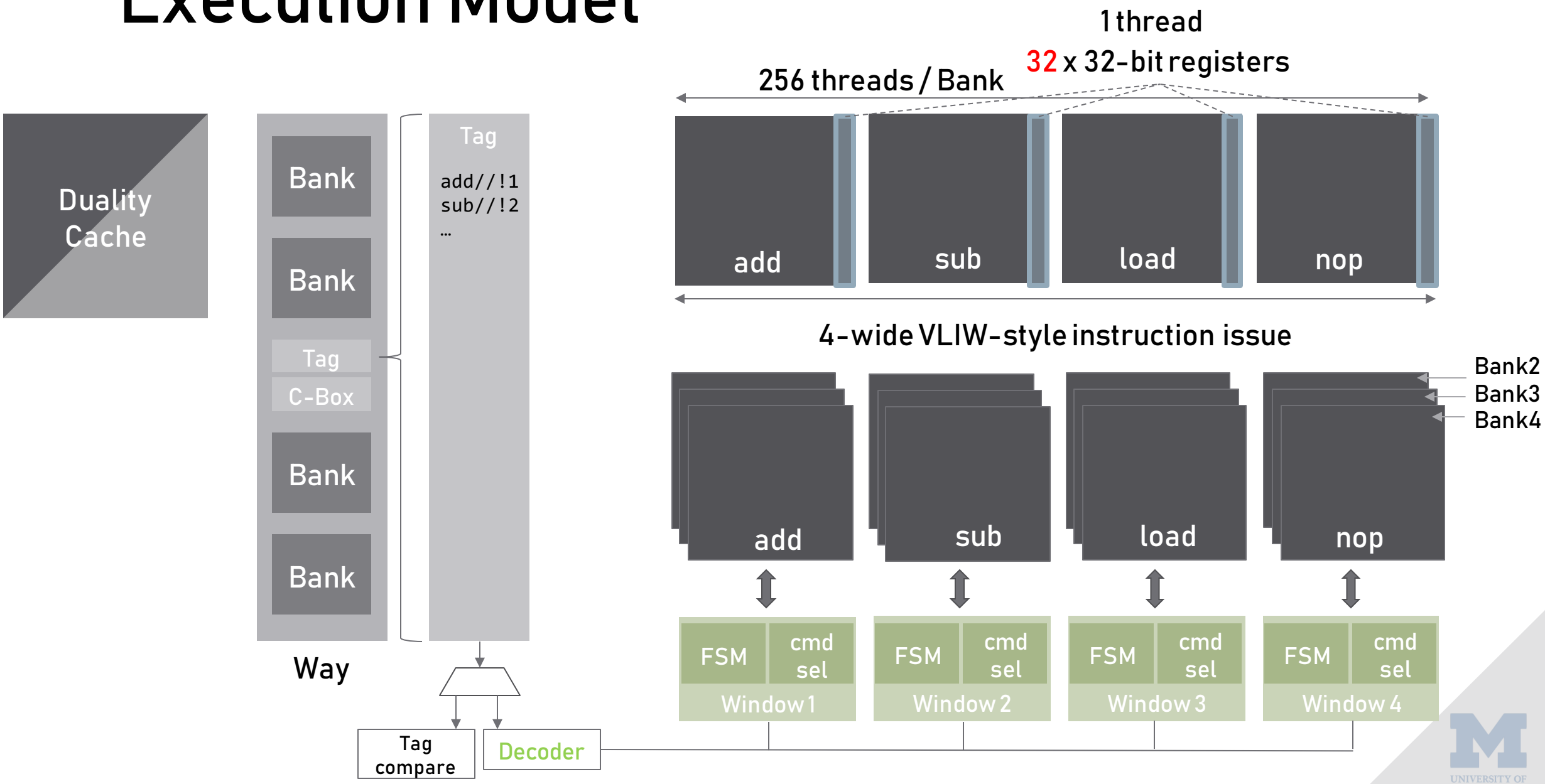


Execution Model

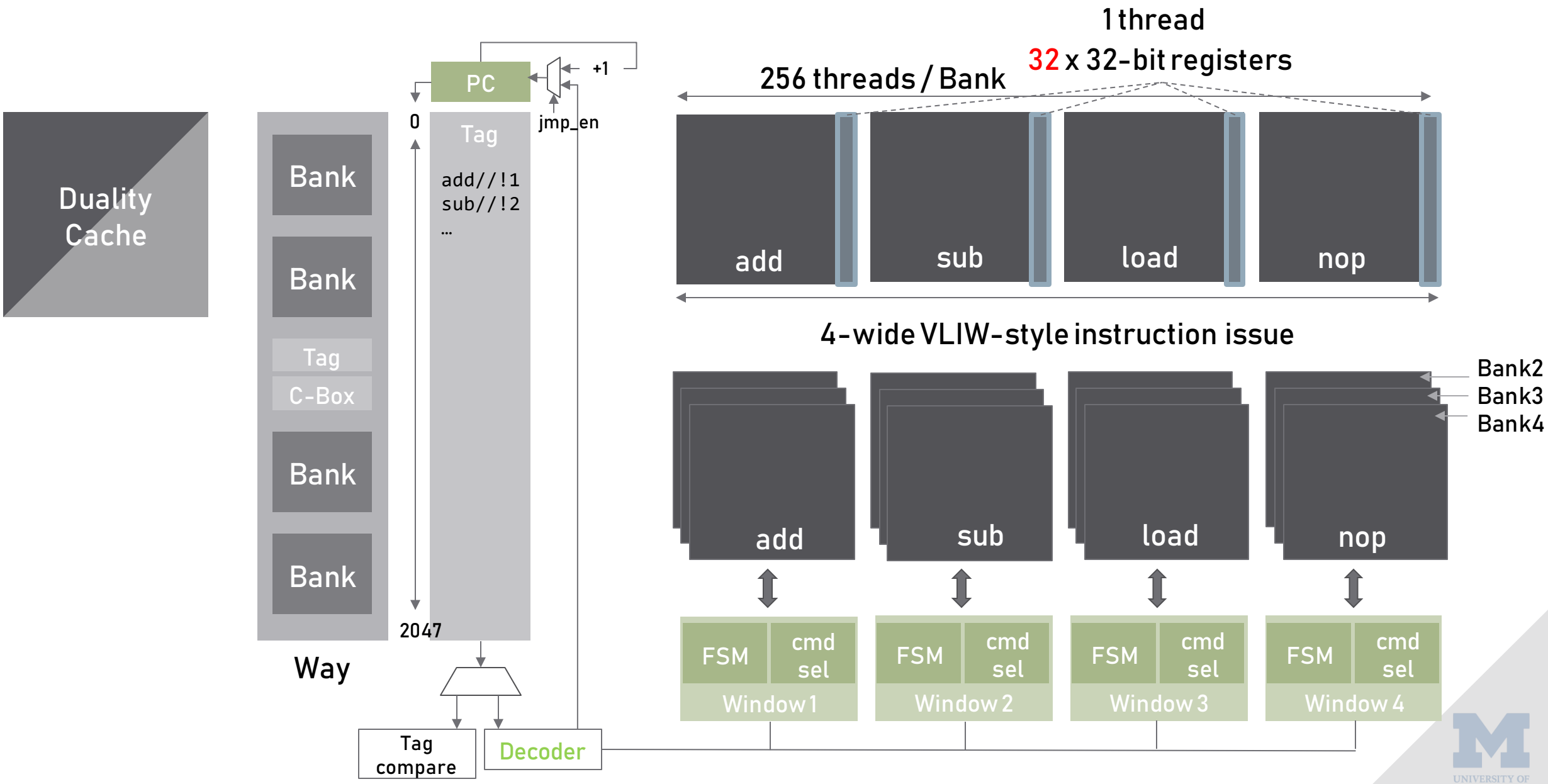


1024 thread 4-wide VLIW SIMD processor

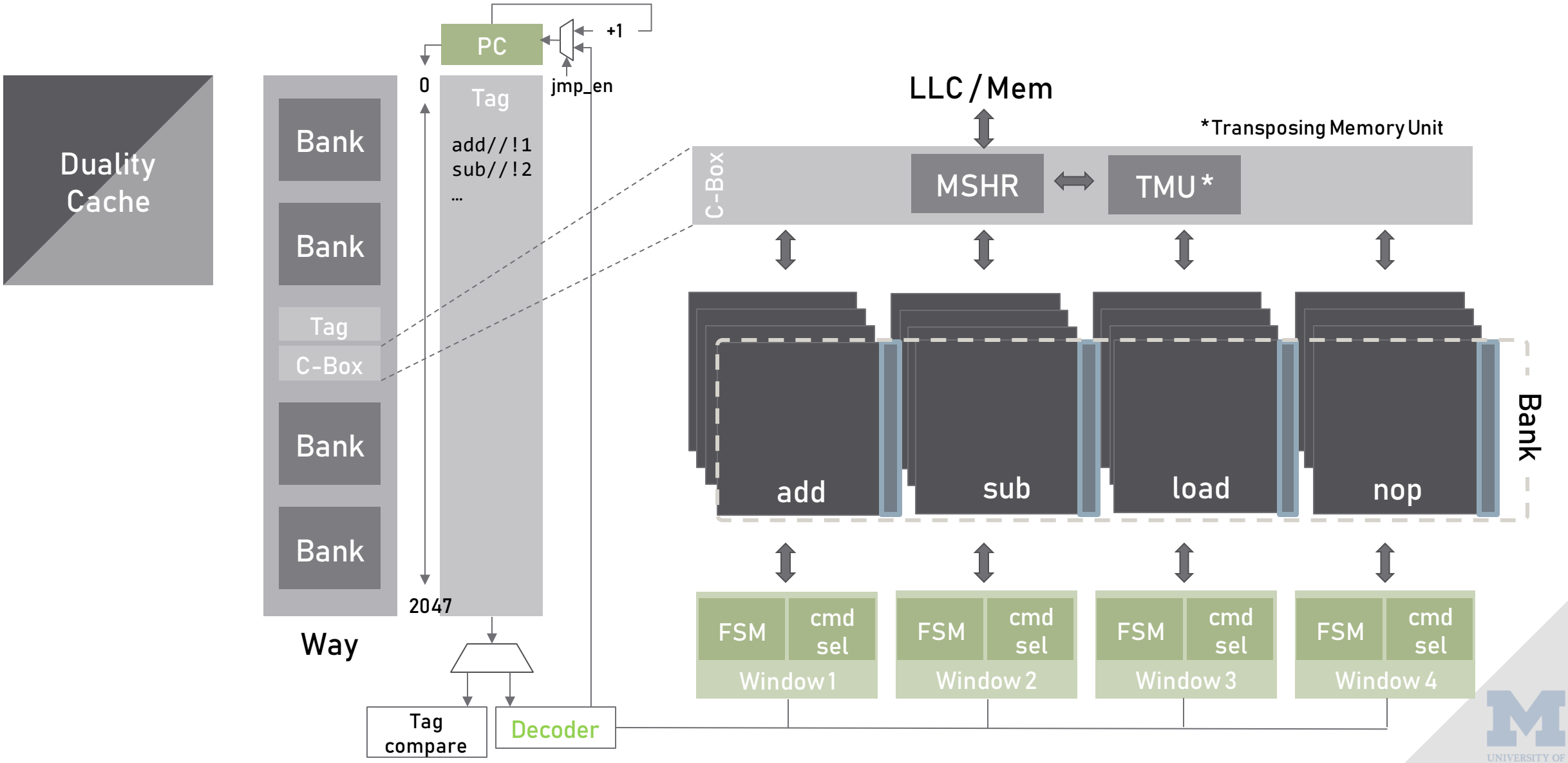
Execution Model



Execution Model



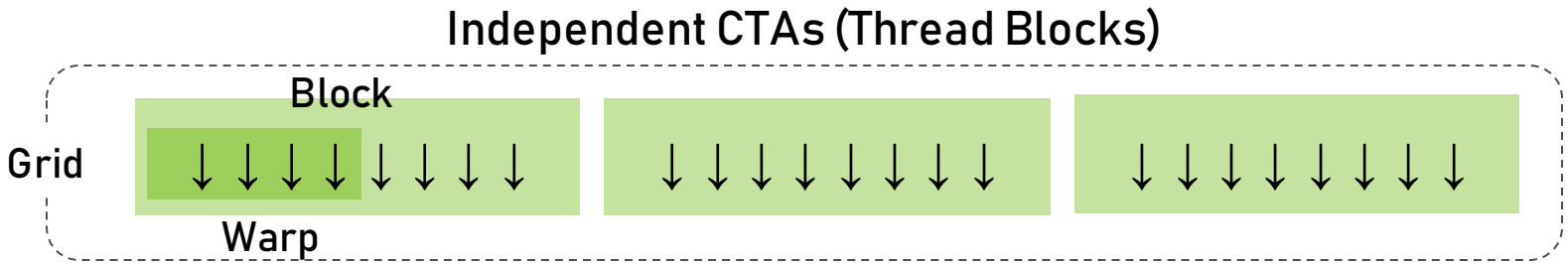
Execution Model



Programming Model

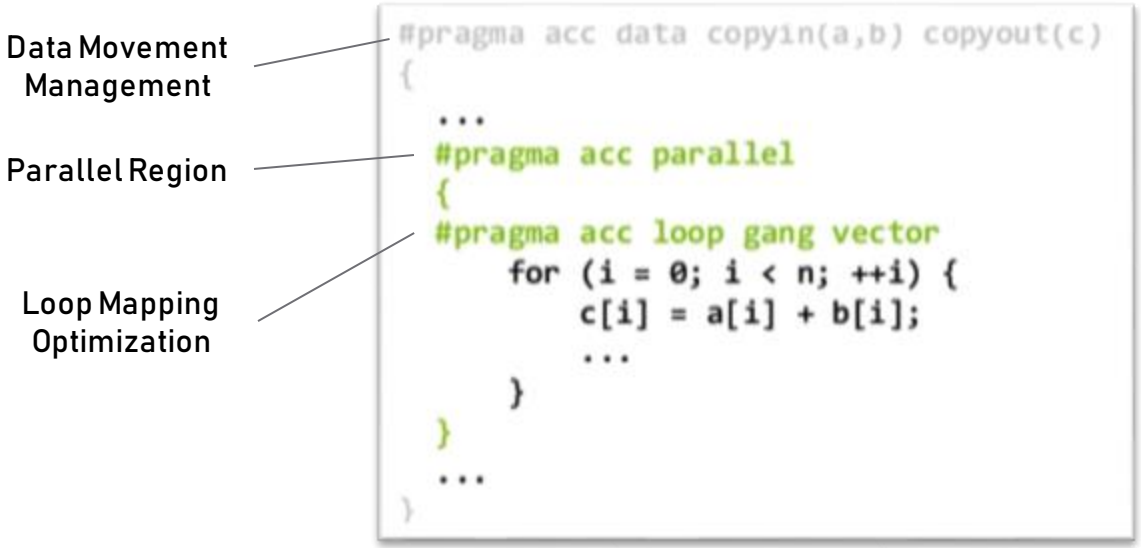

```
// kernel
__global__ void vecadd (const float* A, const float* B, float* C, int n_el) {
        int tid = blockDim.x * blockIdx.x + threadIdx.x;
        if (tid < n_el)
                C[tid] = A[tid] + B[tid];
}

// kernel invocation
vecadd <<<blocksPerGrid, threadsPerBlock>>> (A, B, C, n_el);
```



Programming Model

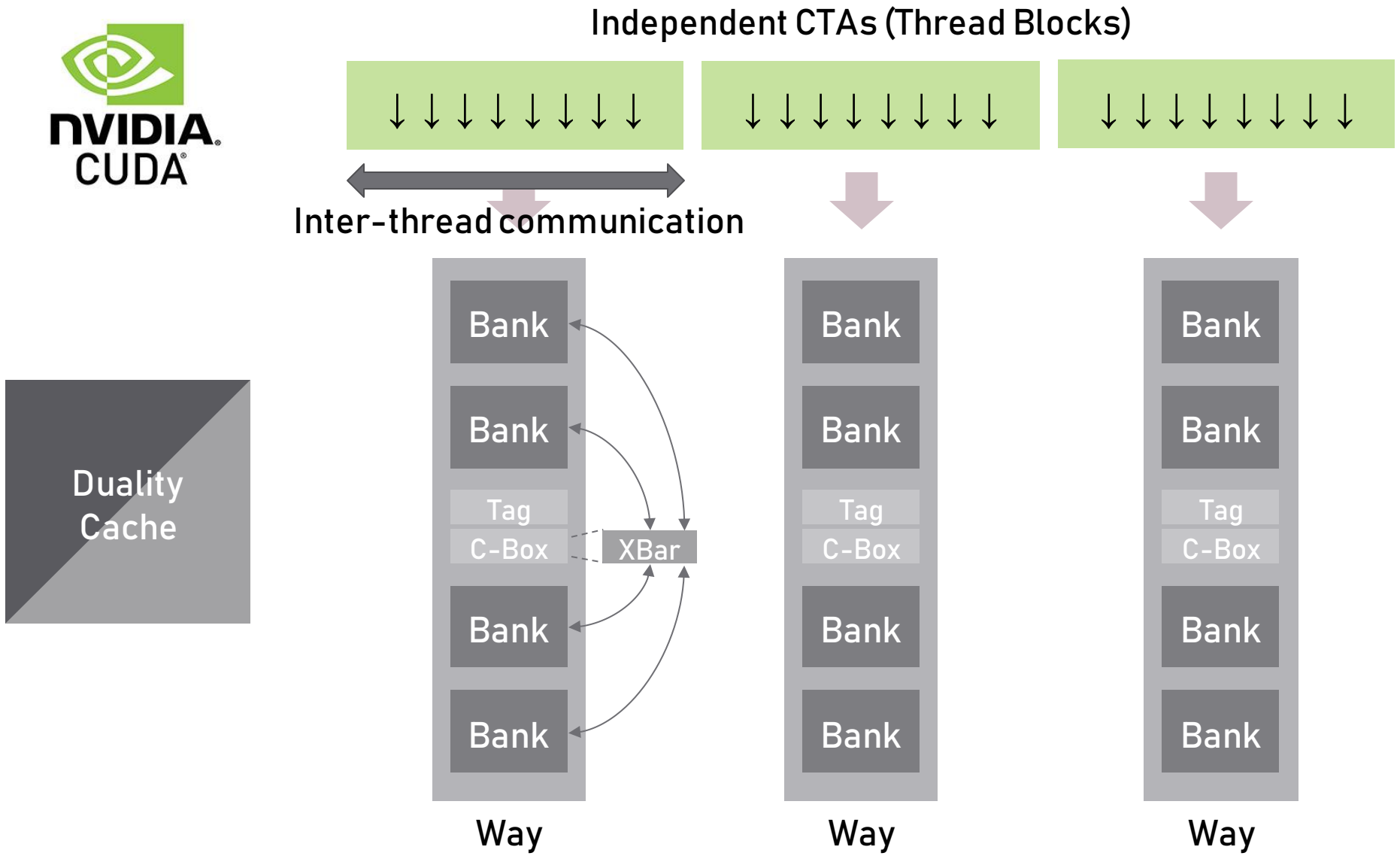
OpenACC
More Science, Less Programming



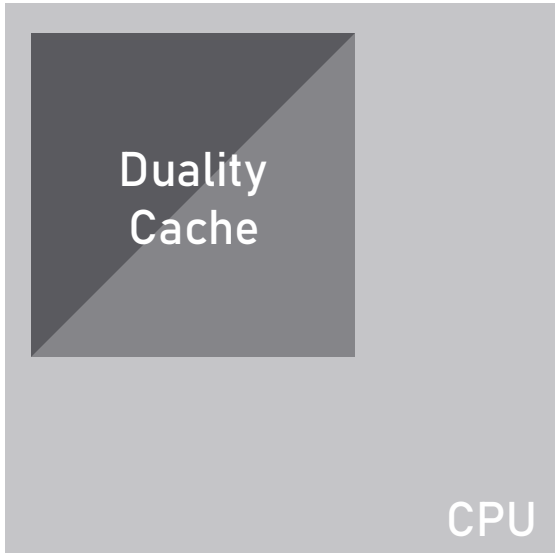
Simple and seamless directive-based parallel programming paradigm for heterogeneous architectures.

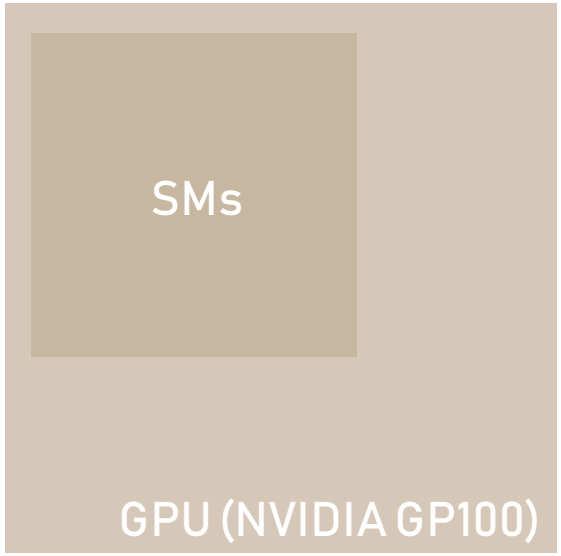
Target Architectures: Native x86, NVIDIA GPU, OpenCL
Compiler Support: **PGI**, **gcc9.1**, Riken Omni, OpenUH

Programming Model



Resource Comparison





Resource Comparison

Xeon E5 35MB LLC

Way

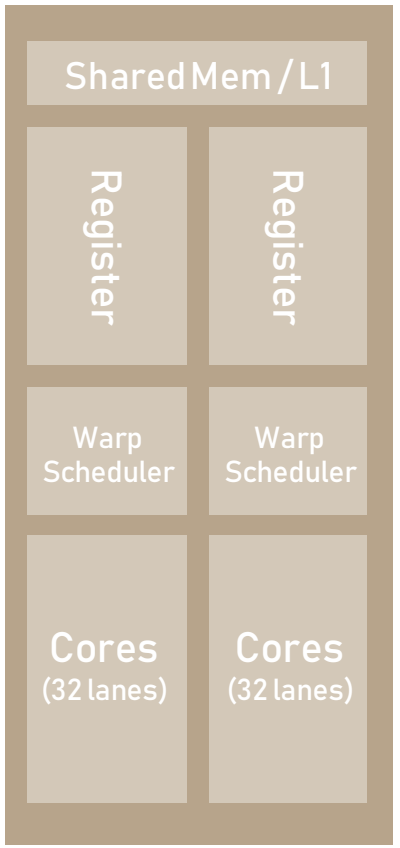
Capacity

Bank × 4 = Register

PU Resources

256 PUs × 4 ≥ 1,024 threads / 32 PUs

32 x 32-bit reg. / thread



SM

NVIDIA GP100 SMs

Resource Comparison

Xeon E5 35MB LLC

Way

x 560

Capacity

Bank × 4 = Register

PU Resources

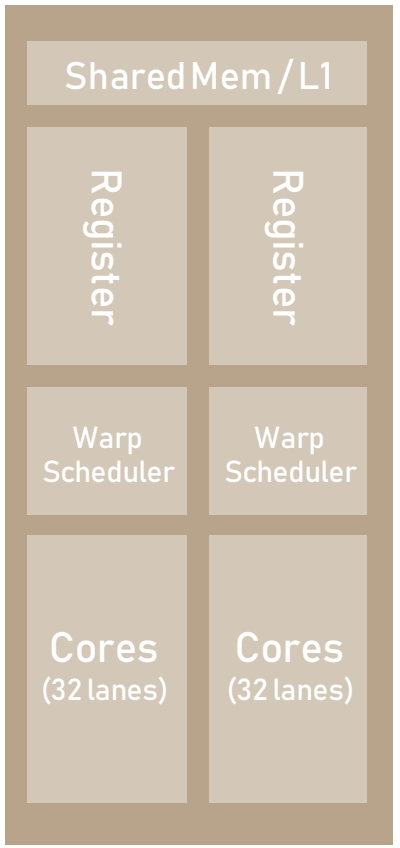
256 PUs × 4 ≥ 1,024 threads / 32 PUs

32 x 32-bit reg. / thread

System Resources
(2 socket CPU + GPU)

150x more PUs

10x more Threads

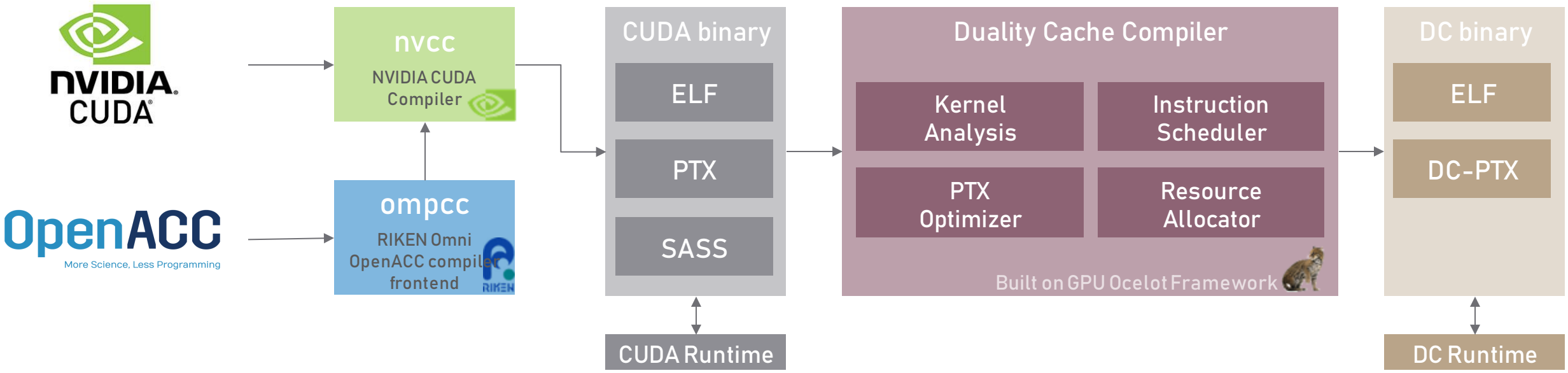


SM

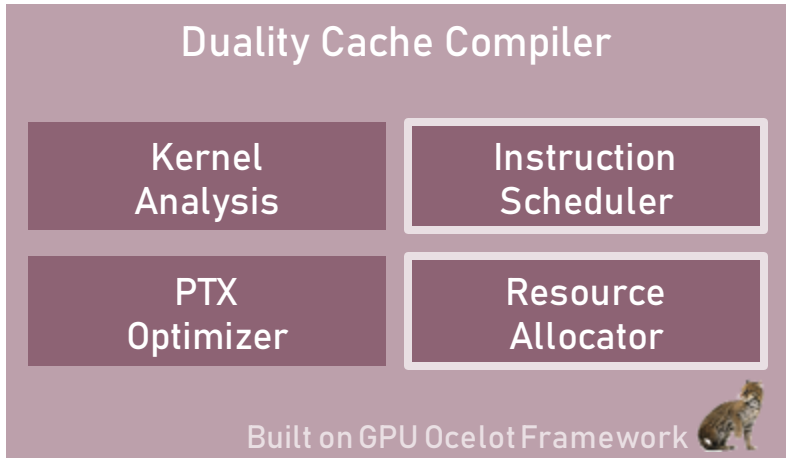
x 28

NVIDIA GP100 SMs

Compiler


NVIDIA CUDA
OpenACC
More Science, Less Programming
nvcc
NVIDIA CUDA Compiler
ompcc
RIKEN Omni OpenACC compiler frontend
RIKEN
CUDA binary
ELF
PTX
SASS
CUDA Runtime
Duality Cache Compiler
Kernel Analysis
Instruction Scheduler
PTX Optimizer
Resource Allocator
Built on GPU Ocelot Framework
DC binary
ELF
DC-PTX
DC Runtime

Compiler Optimization
(1) Register Pressure Aware Instruction Scheduling



Instruction Scheduling First

✓ Maximize parallelism utilizing abundant shared register resources in VLIW architecture

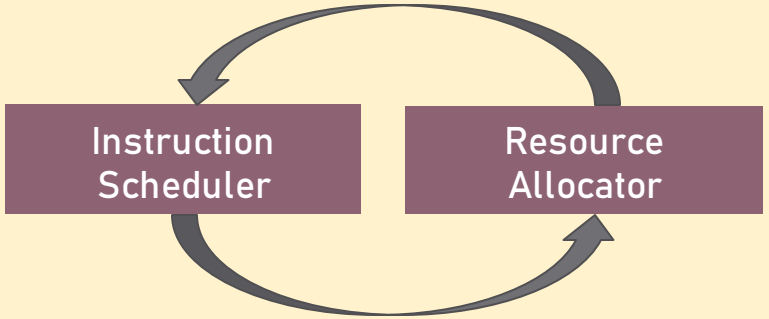
! Result in frequent register spill

Resource Allocation First

✓ Optimized (small) resource usage and reuse distance

! Introduces many false dependencies (less parallelism)

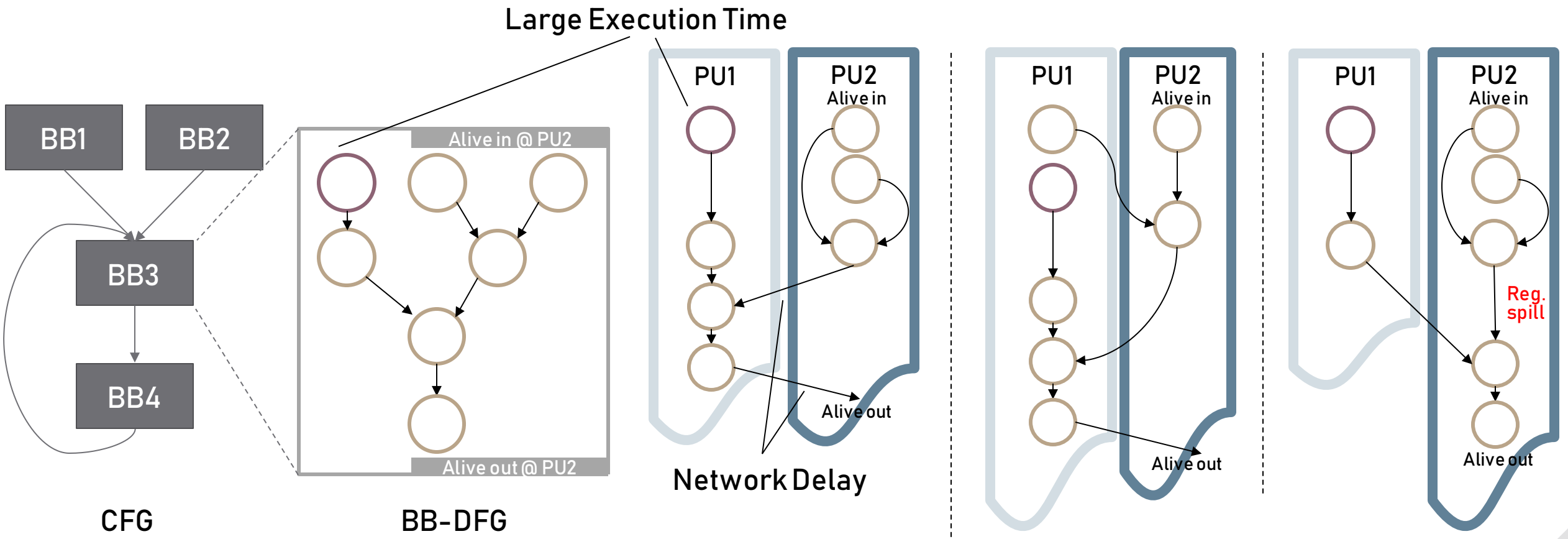
Bottom-Up Greedy (BUG) based



Linear Scan Register Allocation based

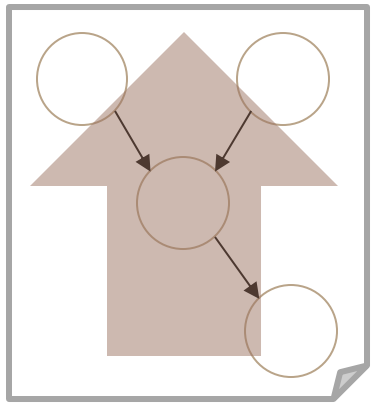
Interactively perform resource allocation and instruction scheduling.

Compiler Optimization
(1) Register Pressure Aware Instruction Scheduling

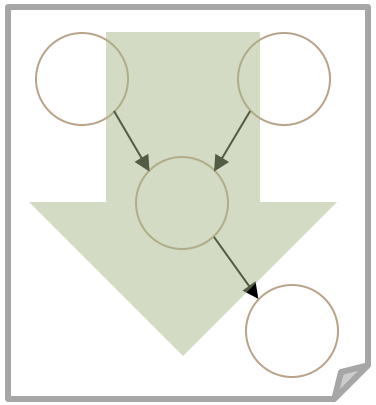


Compiler Optimization
(1) Register Pressure Aware Instruction Scheduling

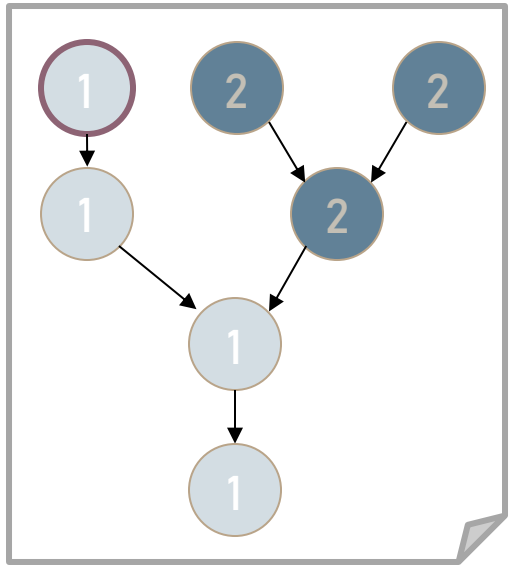
Bottom-Up Greedy [Ellis 1986]

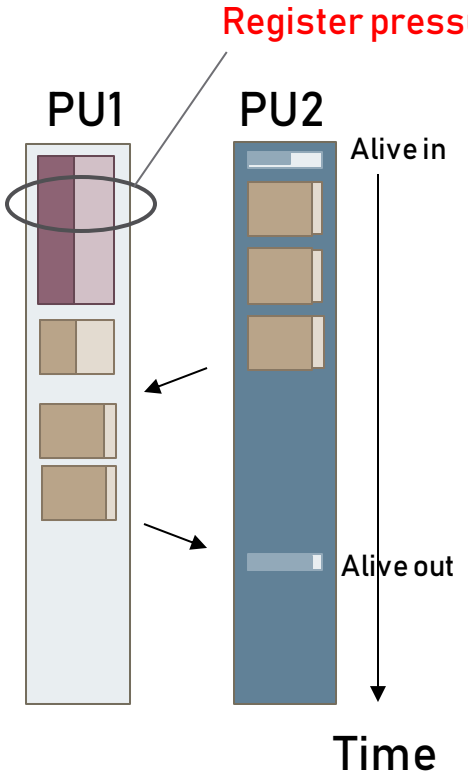


Collect candidate assignments

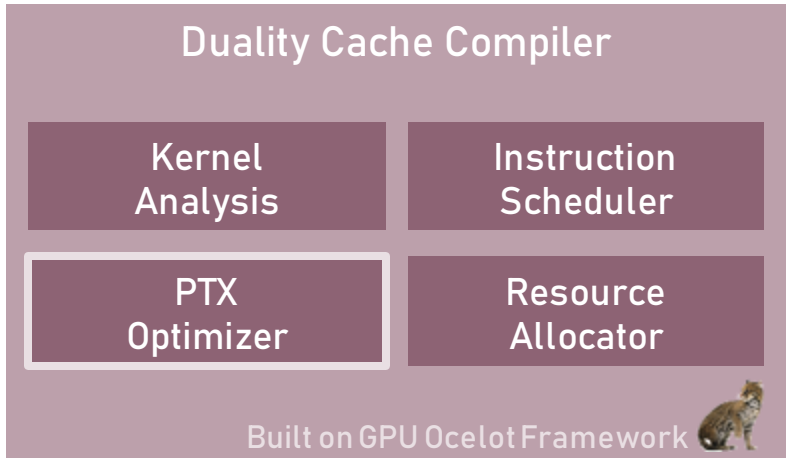


Make final assignments
Minimize data transfer latency by taking both operand & successor location into consideration

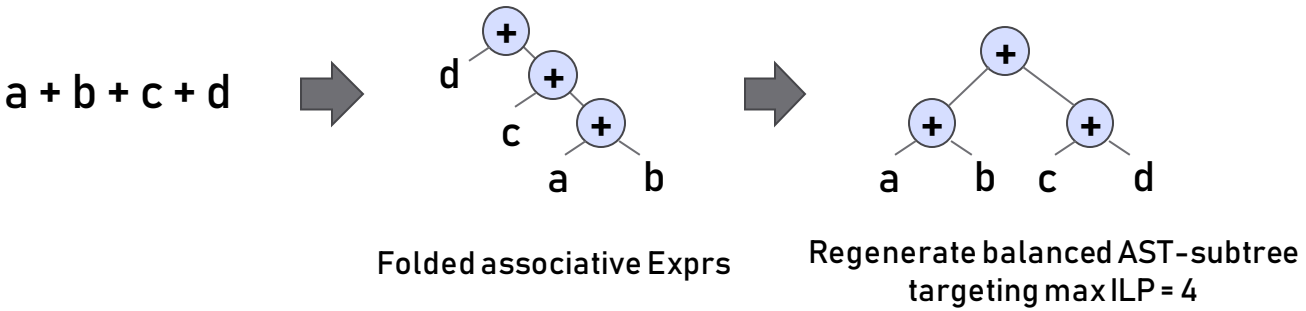




Compiler Optimization
(2) PTX Optimizations



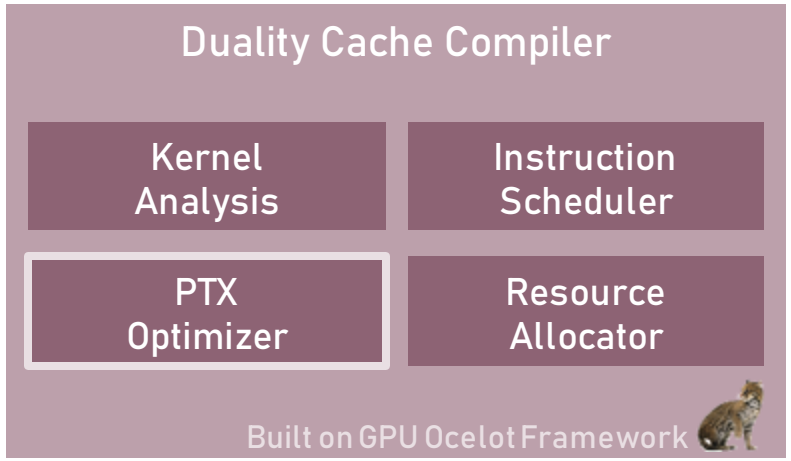
I. AST Balancing



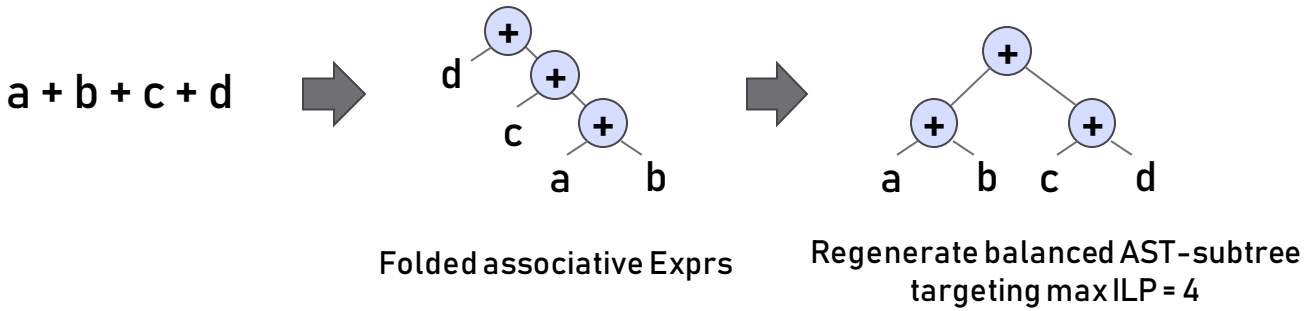
Folded associative Exprs

Regenerate balanced AST-subtree targeting max ILP = 4

Compiler Optimization
(2) PTX Optimizations



I. AST Balancing



Folded associative Exprs

Regenerate balanced AST-subtree targeting max ILP = 4

II. Thread Independent Variable Isolation

```
__device__ kernel
{
        int bid = blockIdx.x;
        for (int i = 0; i < ITER_CNT; ++i)
        {
                // Do something
        }
}
```

bid and i are thread independent variable.

⇓

Loop unroll / const fold.

Do not store them in thread private registers to reduce reg pressure.

Outline

Background

Integer / Logical Operations

Floating Point Operations

Transcendental Functions

Execution Model

Programming Model

Compiler

- Methodology / Results

Evaluation Methodology

- Benchmarks
  - Rodinia
    - backprop
    - bfs
    - b+tree
    - dwt2d
    - hotspot
    - hotspot3D
    - hybridsort
    - nw
    - streamcluster
    - gaussian
    - heartwall
    - leukocyte
    - lud
    - nn
  - PathScale OpenACC benchmark
    - divergence
    - gradient
    - lapsgrb
    - laplacian
    - tricubic
    - tricubic2
    - uxx1
    - vecadd
    - wave13pt
    - gameoflife
    - gaussblur
    - matvec
    - whispering

	CPU 2-sockets	GPU 1-card, PCIe v3	Duality Cache
Processor	Intel Xeon E5-2597 v3, 2.6GHz, 28 cores, 56 threads	NVIDIA GP100, 1.6GHz, 3840 cuda cores	35MB Duality Cache, 2.6 GHz
On-chip Memory	78.96 MB	9.14 MB	78.96 MB
Off-chip Memory	64 GB DDR4	12GB GDDR5 + 64GB DDR4 (host)	64 GB DDR4
Total System Area	912 mm^2	1383 mm^2	942 mm^2
TDP	290 W	640 W	296 W
Profiler / Simulator (Performance)	perf	NVPROF	GPU Ocelot + Ramulator
Profiler / Simulator (Energy)	Intel RAPL Interface	NVIDIA System Management Interface	Trace based simulation

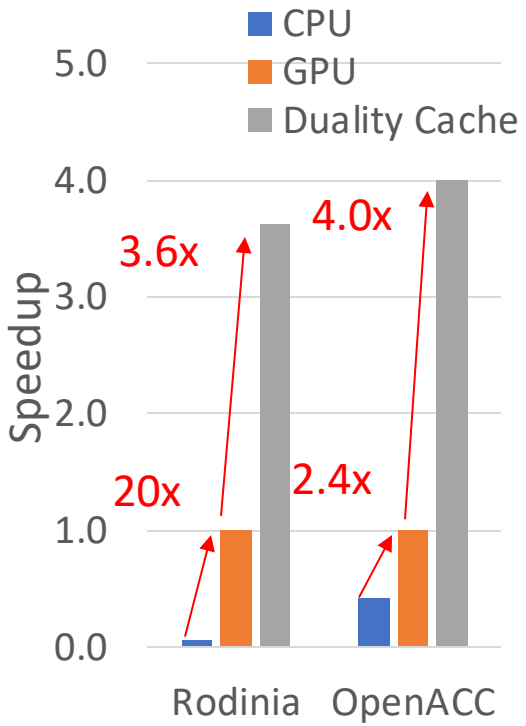
Area / Power Overhead

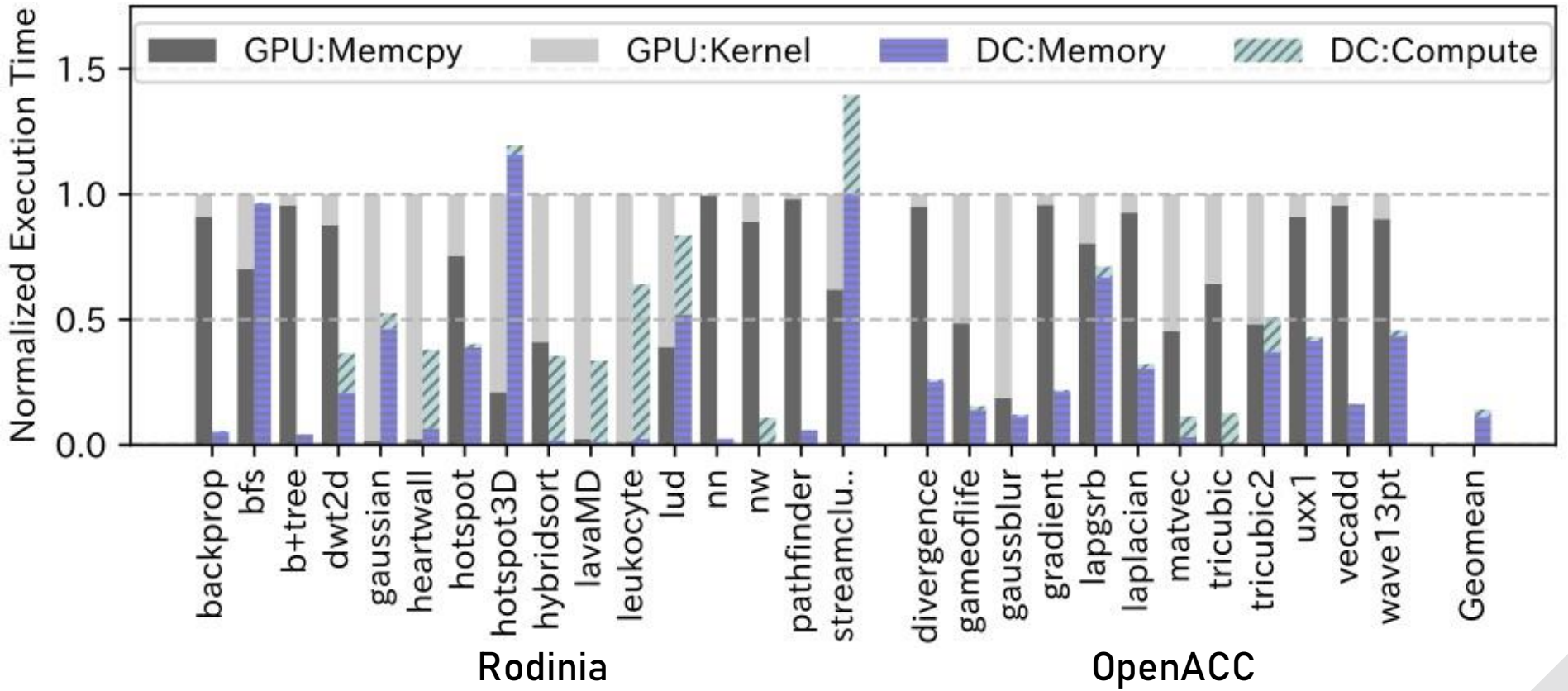
	Area (mm2)	Power (W)	Area Overhead
CPU	456	145	-
Compute Cache Peripheral	3.15	2.96	0.69%
TMU	5.32	0.06	1.17%
Controller / FSM	6.16	0.33	1.35%
MSHR	0.86	0.05	0.19%
Local Crossbar	0.28	0.01	0.06%
Total	471.77	148.4	3.50%

Duality Cache has only 3.5% area overhead

System Speedup

CPU, DC: DDR4, GPU: GDDR5 + memcpy (DDR4 ↔ PCIe v3 ↔ GDDR5)





Key performance factors

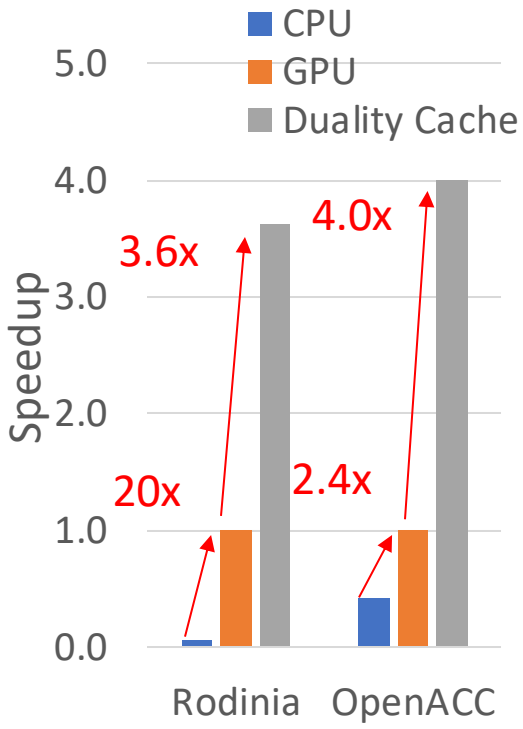
- Reduced Memcpy time
- Massively parallel execution
- Compute / memory access overlap
- Flexible cache allocation

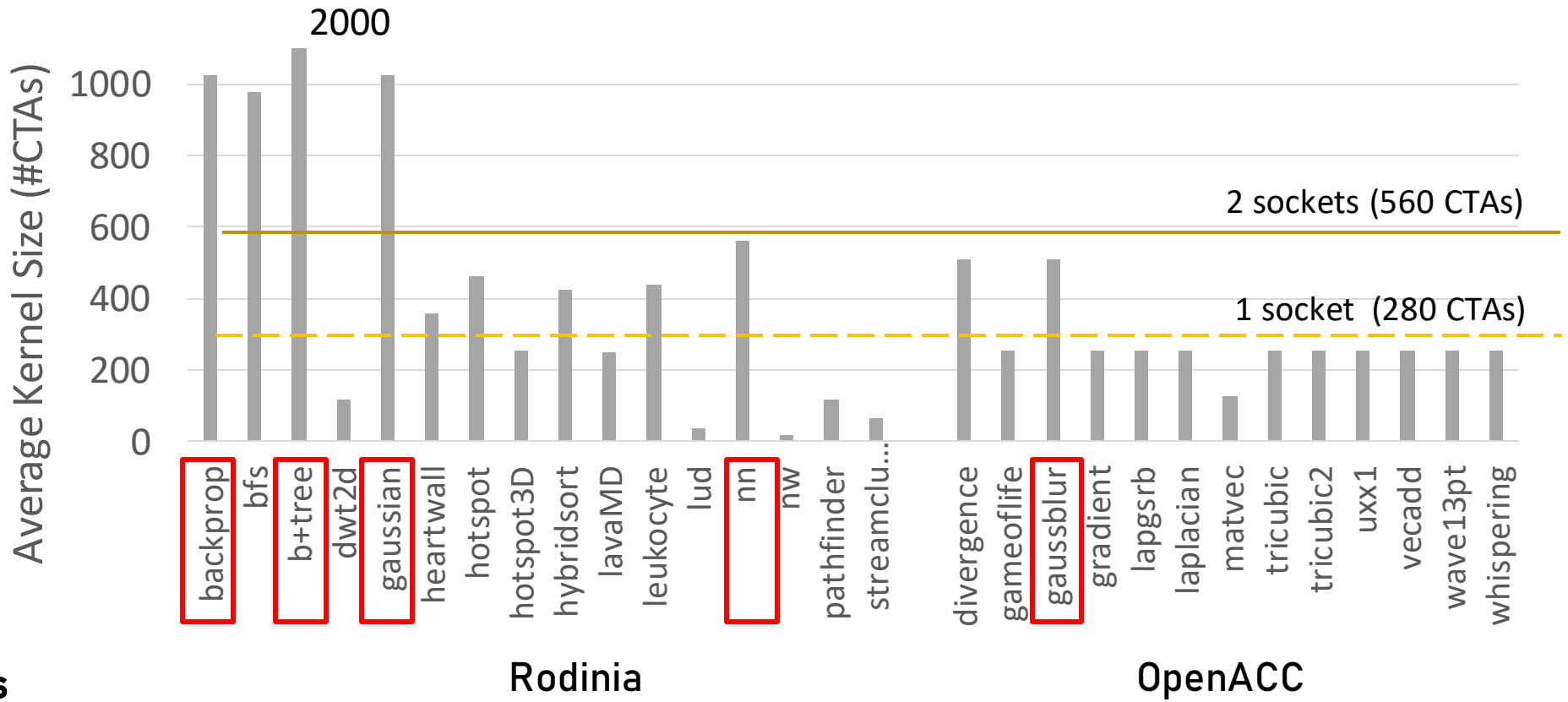
Reduced data transfer (memcpy) cost via external bus (PCIe).

Small reuse factor / large working set make memcpy costly for GPU.

Massively Parallel Execution

PU Utilization





Key performance factors

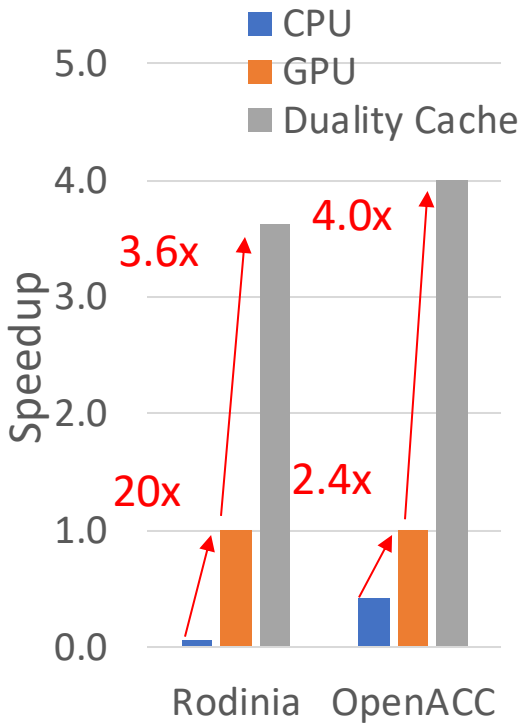
- Reduced Memcpy time
- Massively parallel execution
- Compute / memory access overlap
- Flexible cache allocation

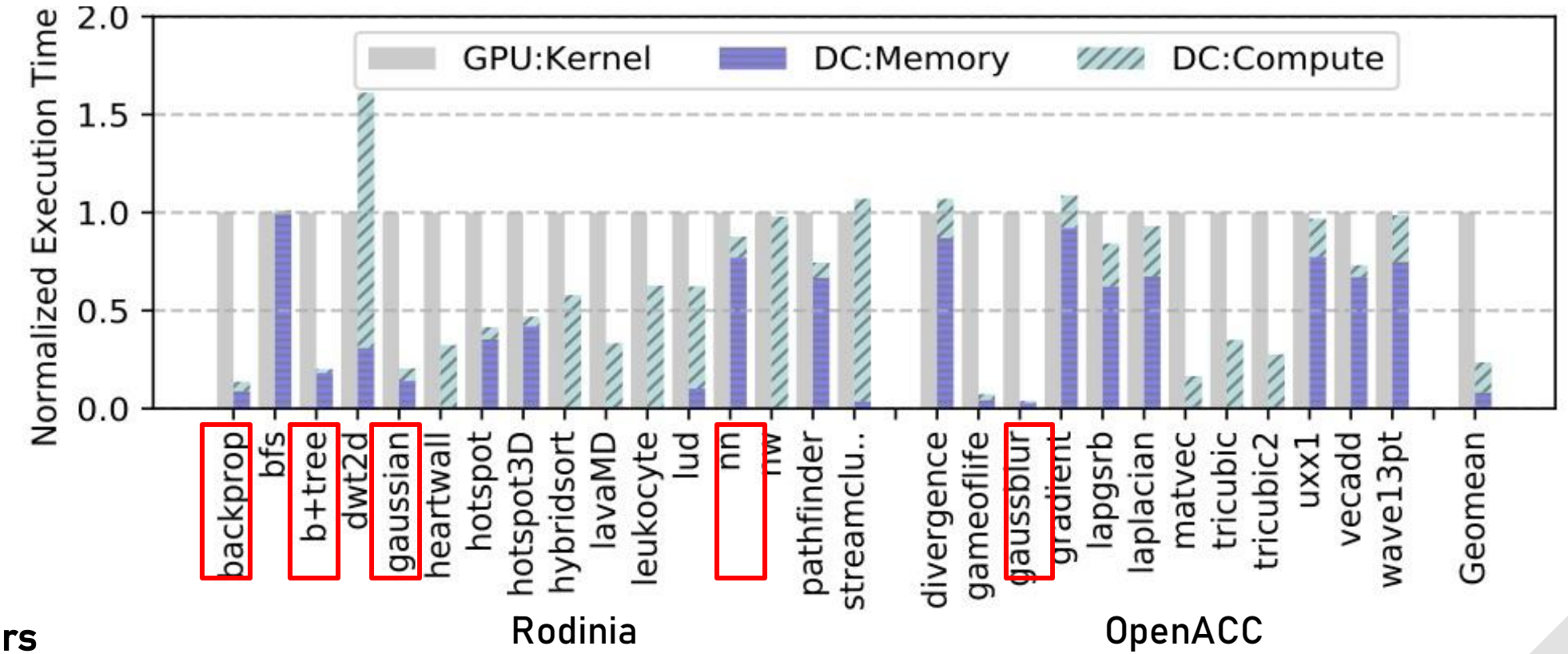
Some kernels have high level of parallelism (e.g. backprop, b+tree, nn, gaussian, gausblur, etc.)

280 CTA/ socket (DC) vs. 60 CTA / card (GPU) (reg. capacity based)

Kernel Speedup

DC: GDDR5, GPU: GDDR5 + No memcpy





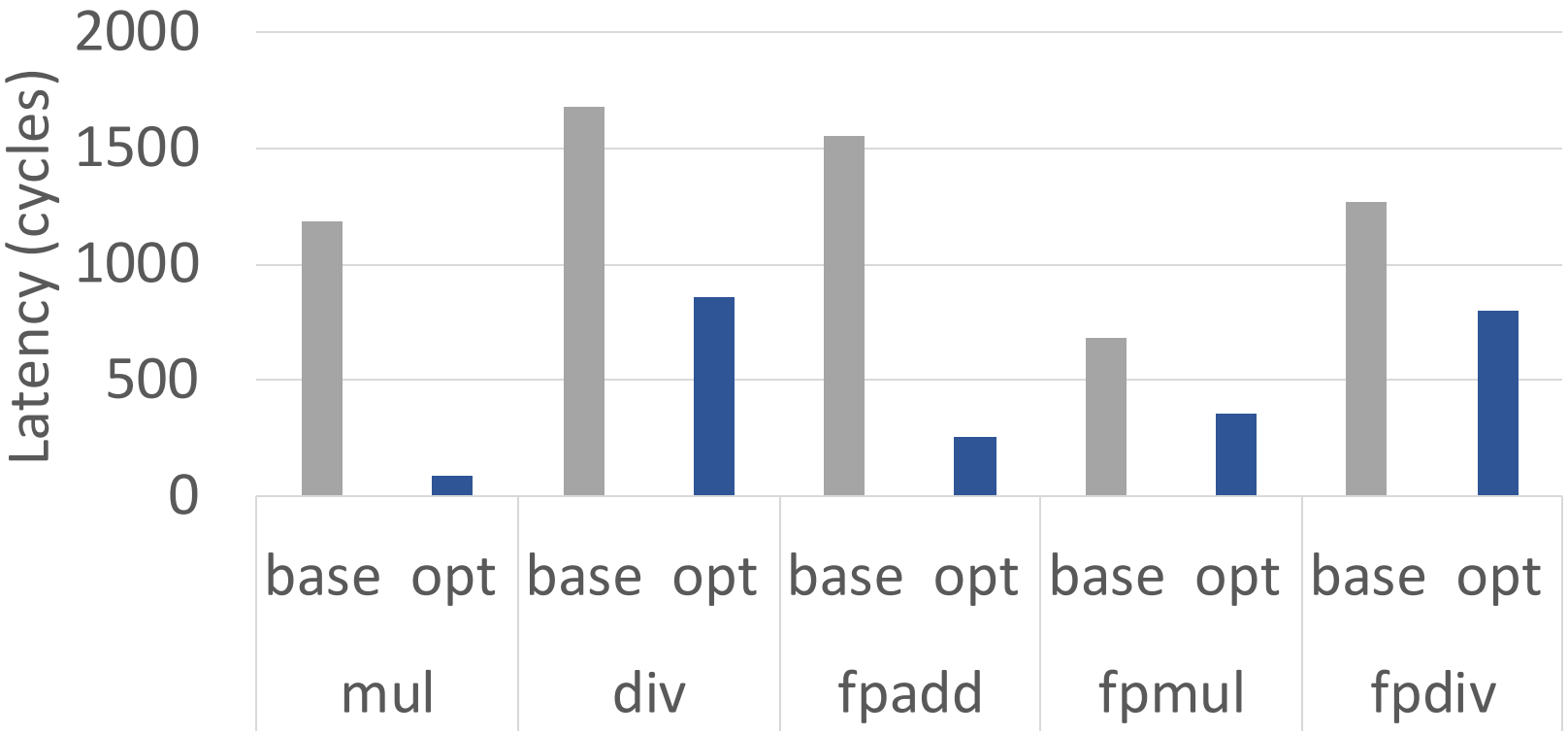
Key performance factors

- Reduced Memcpy time
- Massively parallel execution
- Compute / memory access overlap
- Flexible cache allocation

Some kernels with high level of parallelism (e.g. backprop, b+tree, nn, gaussian, gaussblur, etc.) significantly reduce execution time.

Memory bounded applications (hotspot3d, streamcluster) will benefit from GDDR5

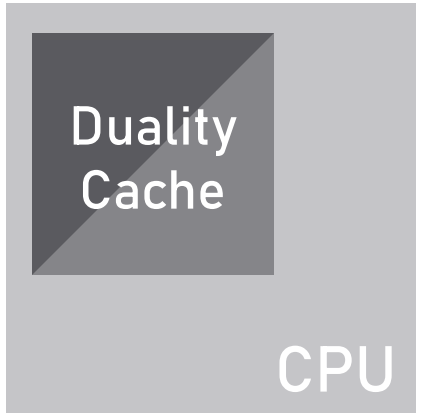
Operation Latency



Integer multiplication is often used to calculate address or some variables based on induction variables as an operand
→ They contain many leading zeros which we can skip by our optimization. (13x faster)

Floating point addition has small dynamic range.
→ The number of unique ediff found usually has its peak at 1 in the distribution. (6.1x faster)

Conclusion



Contributions

In-cache computing framework for general purpose programming

- Used SIMT programming model for the programming frontend
- Developed a compiler for in-cache computing
- Enhanced computation primitives (all digital)

Results

Overall Efficiency

1450x over CPU
52x over GPU

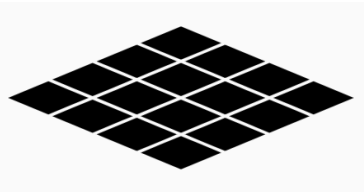
Performance

72x over CPU
4x over GPU

Energy

20x over CPU
5.8x over GPU

Area



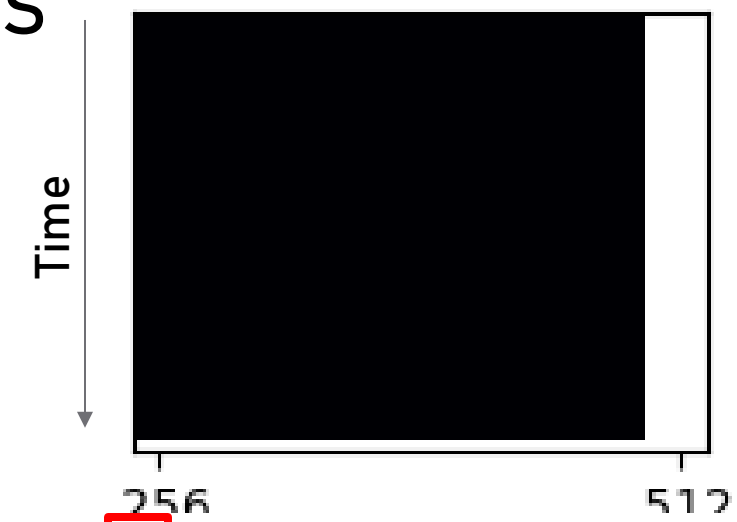
Duality Cache needs 15.7 mm^2
== 3.5% over CPU
GPU = 471 mm^2

Backup Slides

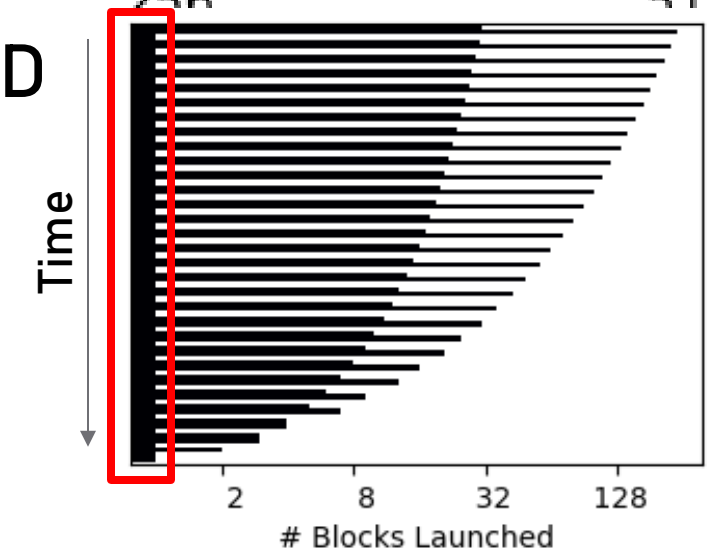
DC + CPU Hybrid Execution

Kernel Launch Pattern
(time vs. #CTAs launched)

BFS

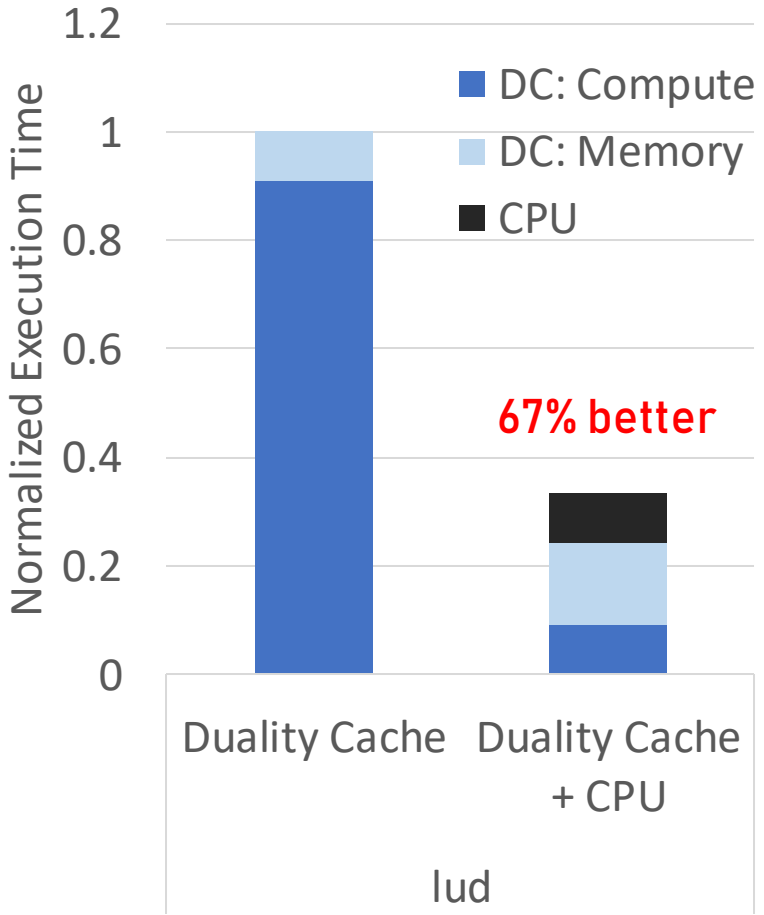


LUD

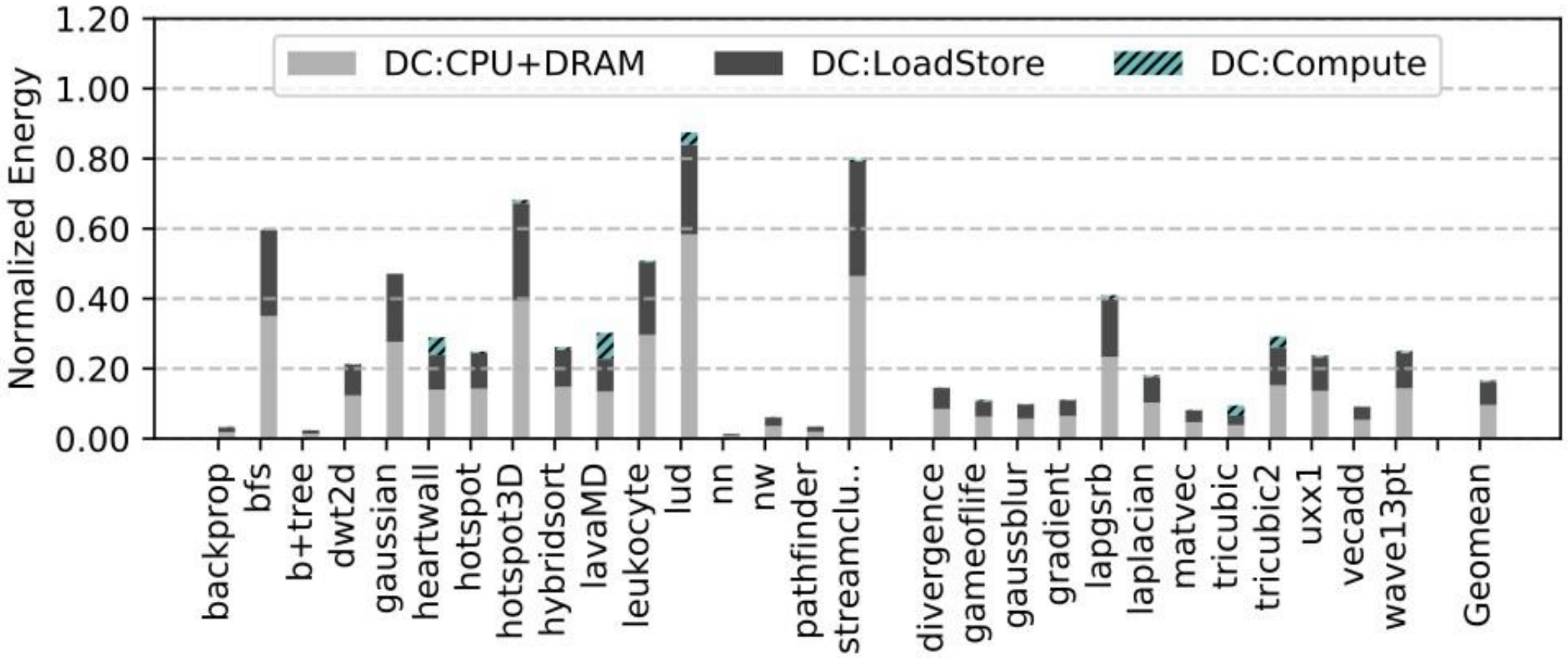


Enough parallelism to keep all PUs busy

Execute small kernels on CPU threads

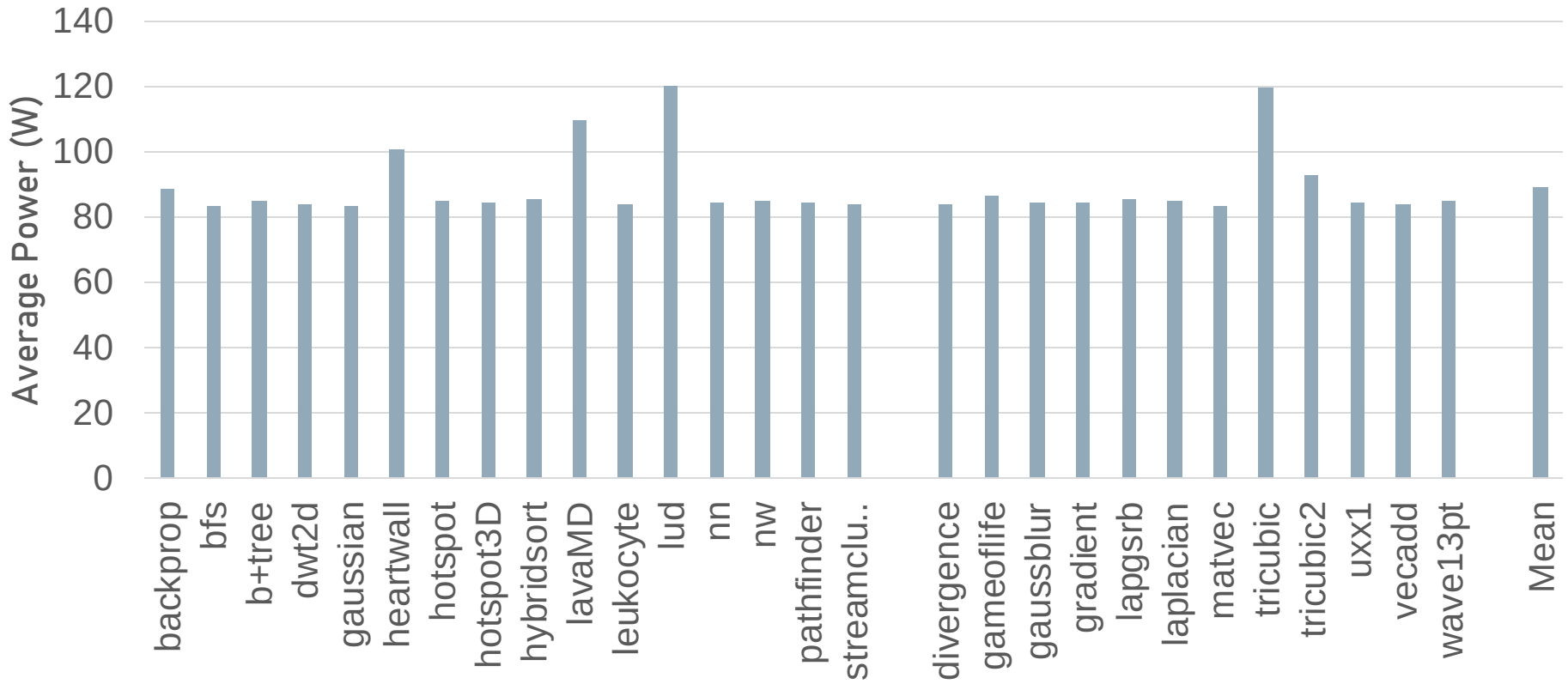


System Energy



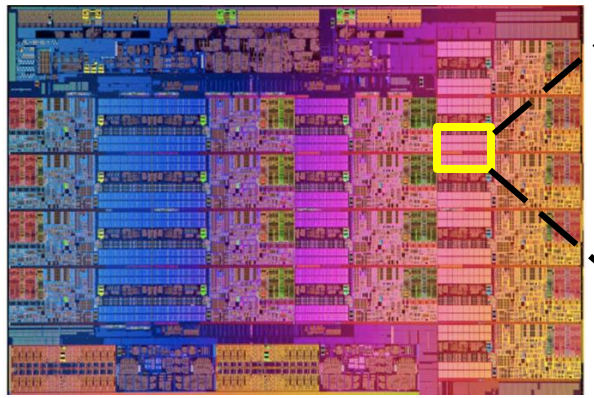
Because of the reduced execution time, we achieve 5.34× energy efficiency compared to GPU. One core is active during execution to serve instructions. This makes CPU and DRAM access dominant in energy consumption.

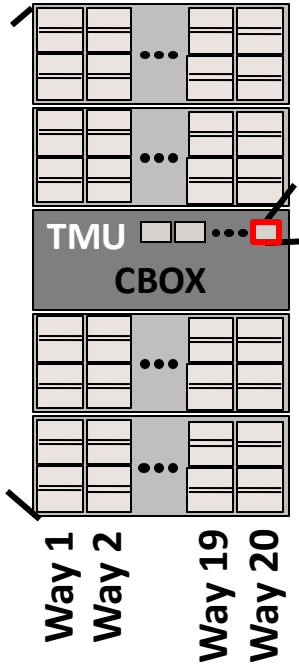
Average Power

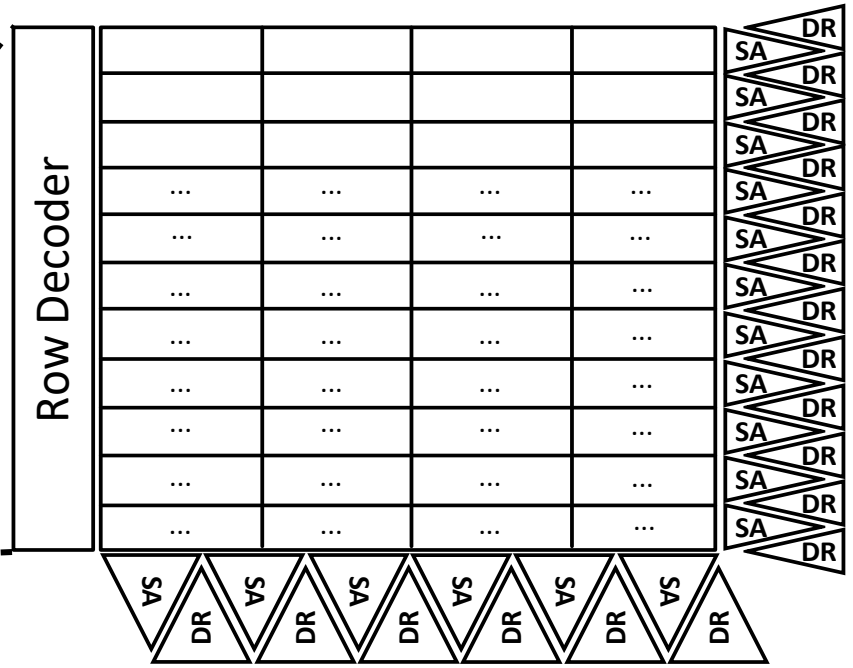


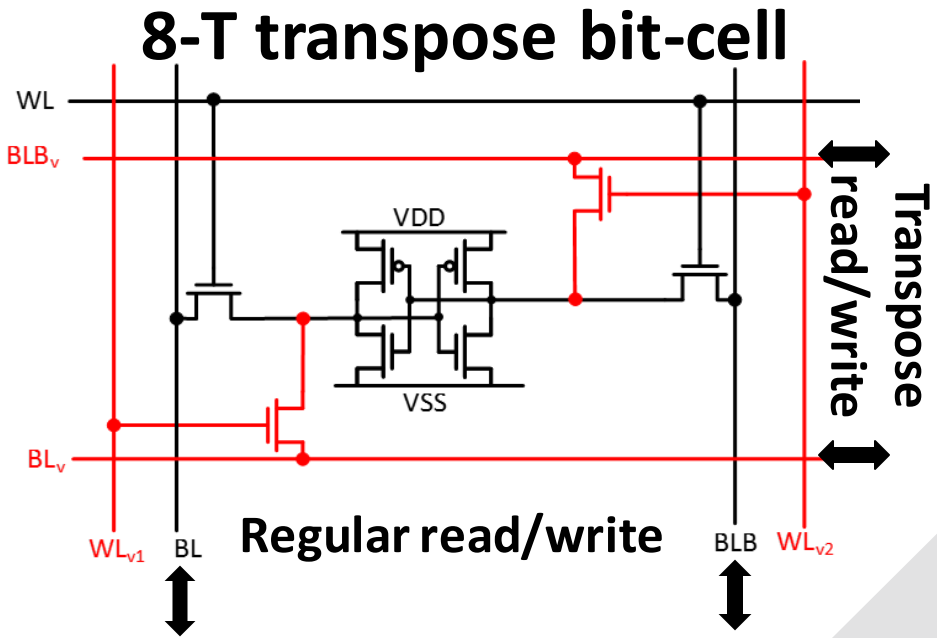
Mean: 88.8 W
Max: 120.1 W

Transpose

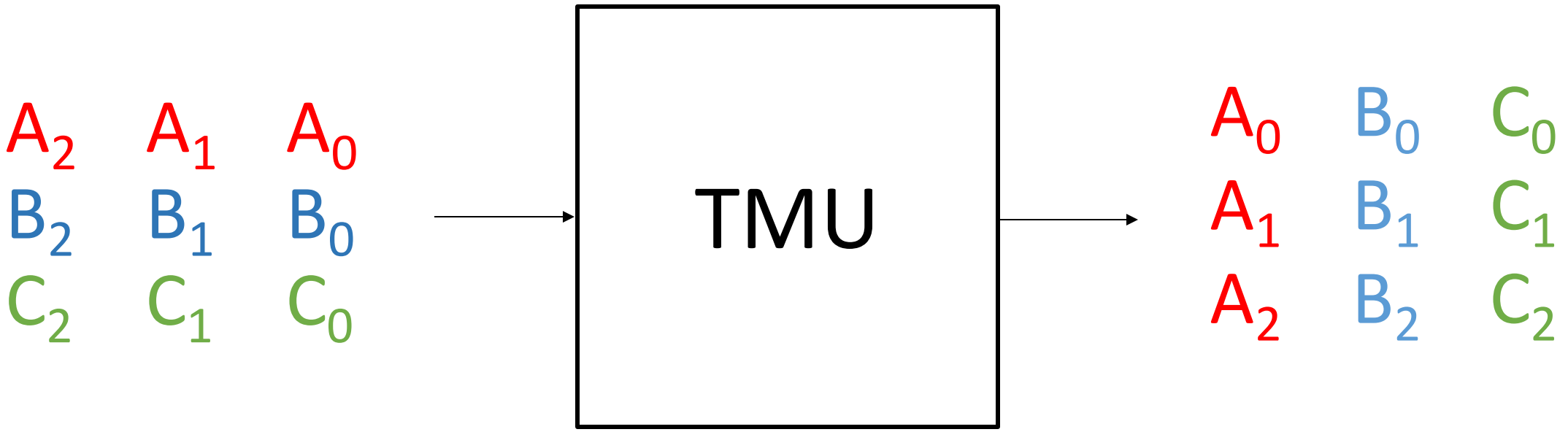



TMU
CBOX
Way 1
Way 2
Way 19
Way 20

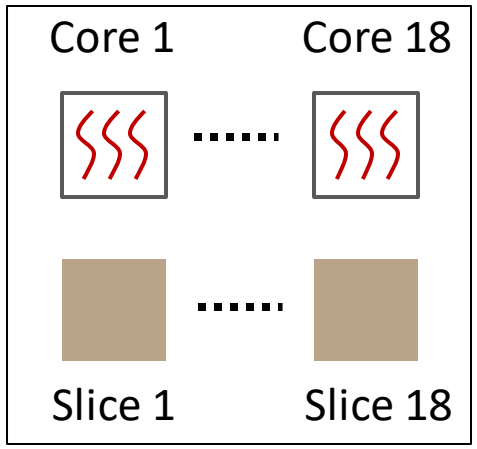

Row Decoder
SA
DR


8-T transpose bit-cell
WL
BLB$_v$
VDD
VSS
BL$_v$
WL$_{v1}$
BL
Regular read/write
BLB
WL$_{v2}$
Transpose read/write

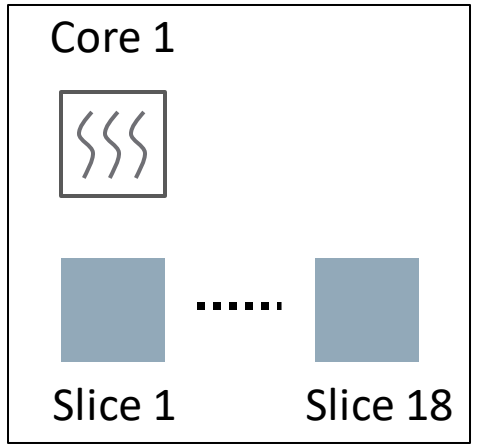
Transpose



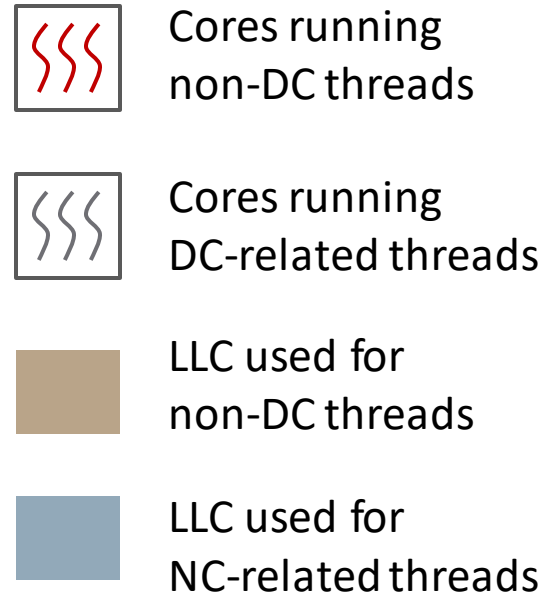
Duality Cache Operation Modes

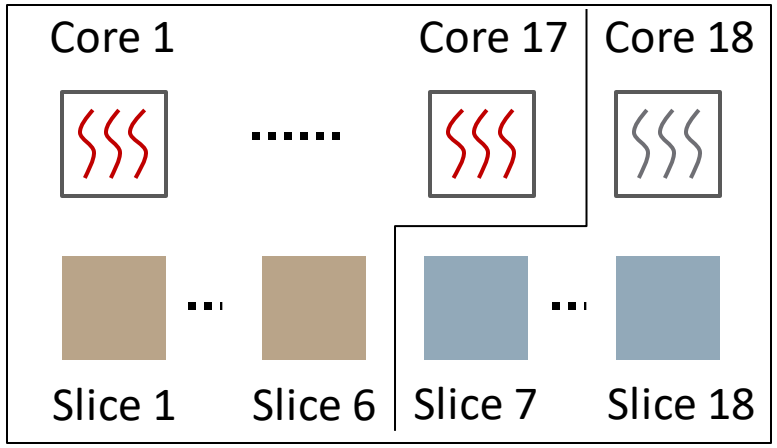


CPU only mode

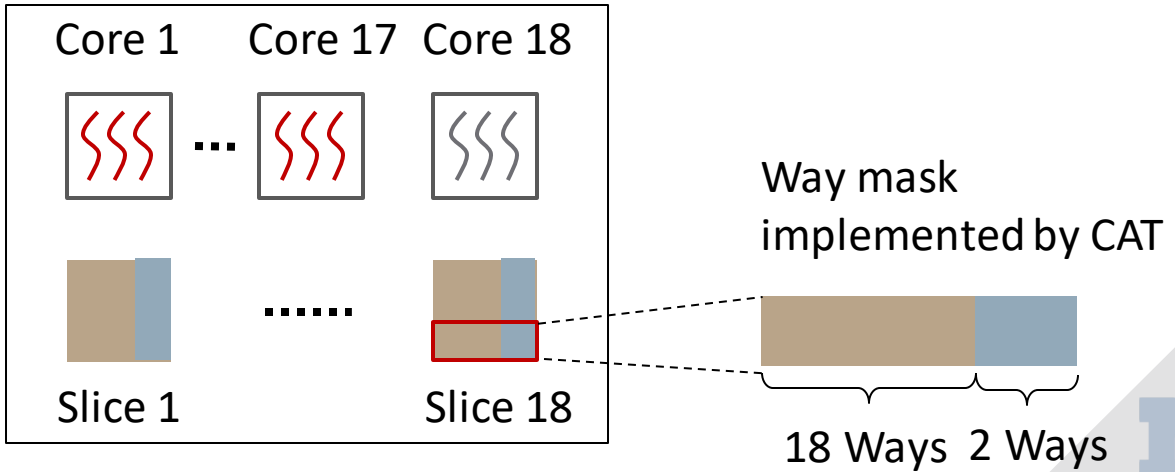


Duality Cache only mode





Hybrid mode - Slice partitioned



Hybrid mode - Way partitioned