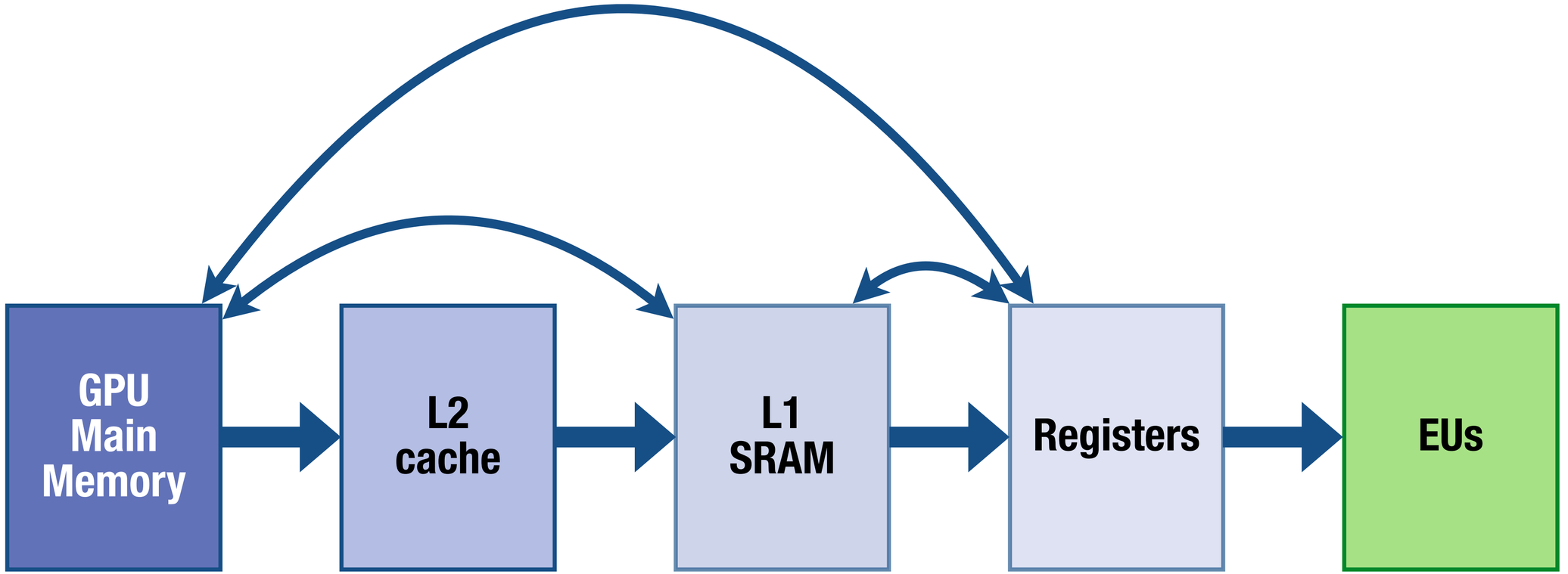
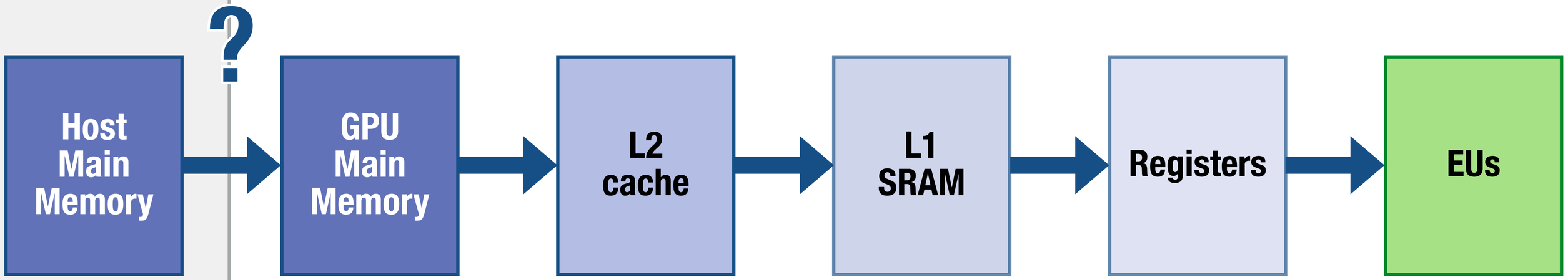
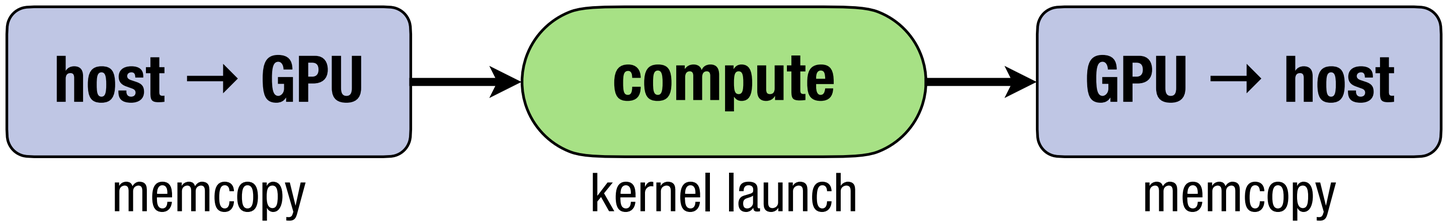
Overlapping compute & I/O


GPU
Main
Memory
L2
cache
L1
SRAM
Registers
EUs


?
Host Main Memory
GPU Main Memory
L2 cache
L1 SRAM
Registers
EUs

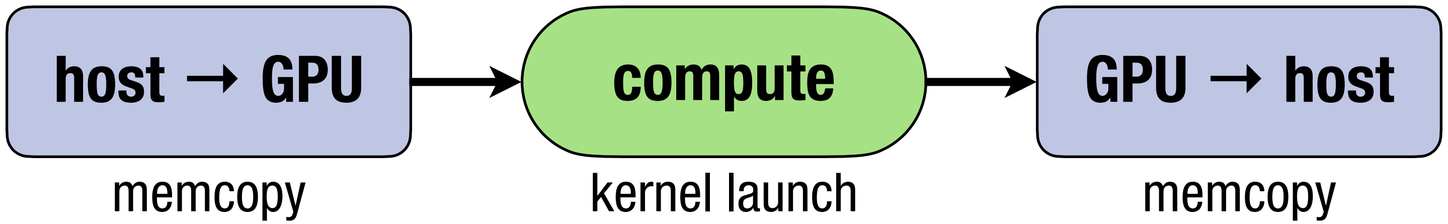
How do **GPU ↔ Host copies** happen? Does it use the processor cores (SMs)?



CPU code:

```
gpu_memcpy_host_device(cpu_buf, gpu_buf);
my_kernel<<<blocks, threads>>>(args…););
gpu_memcpy_device_host(gpu_buf, cpu_buf);
```

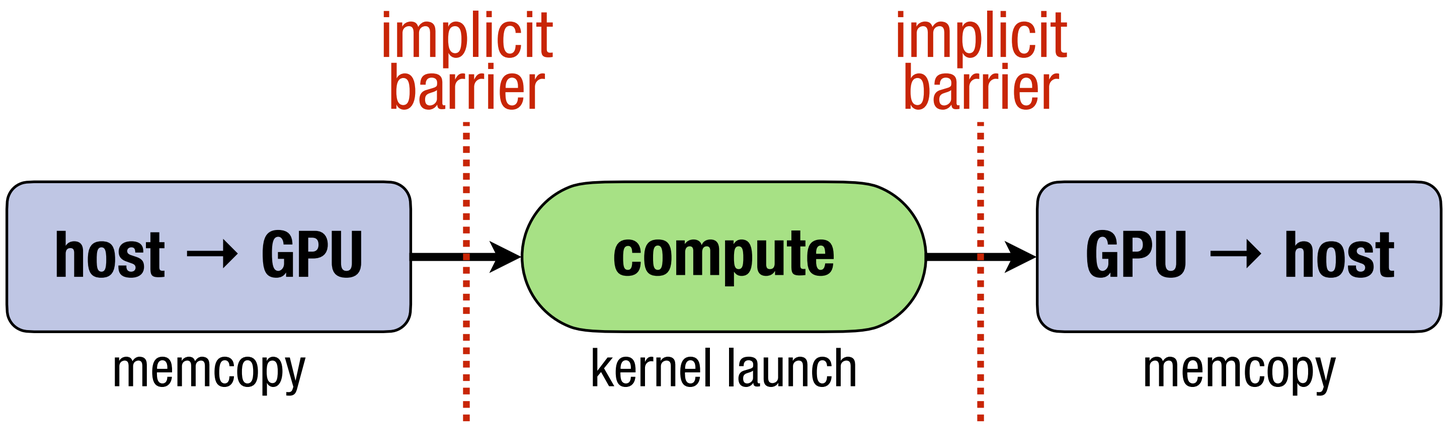
How do **GPU ↔ Host copies** happen? Does it use the processor cores (SMs)?



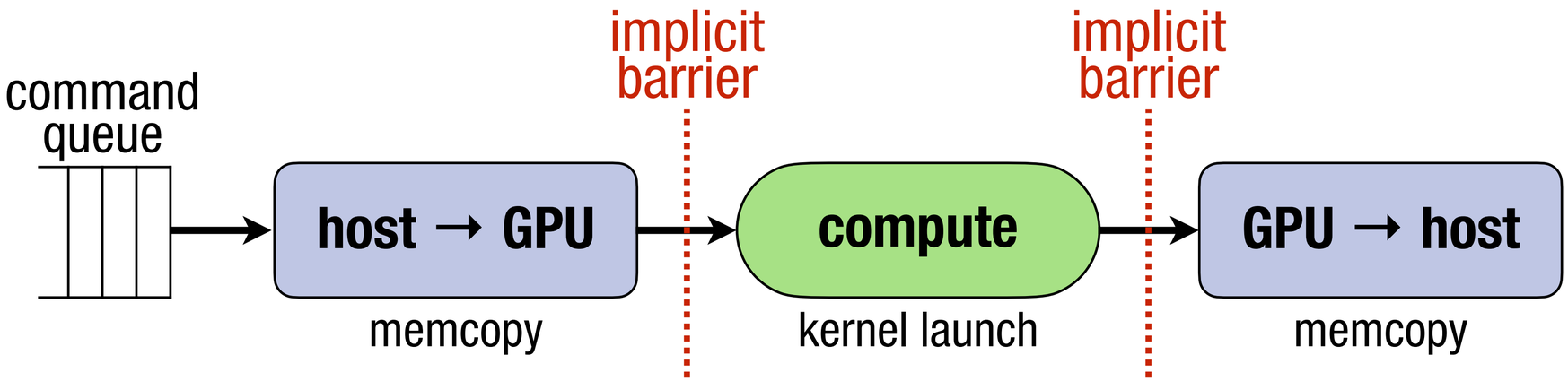
No!

Dedicated DMA
can run **fully in parallel**
with kernel computation

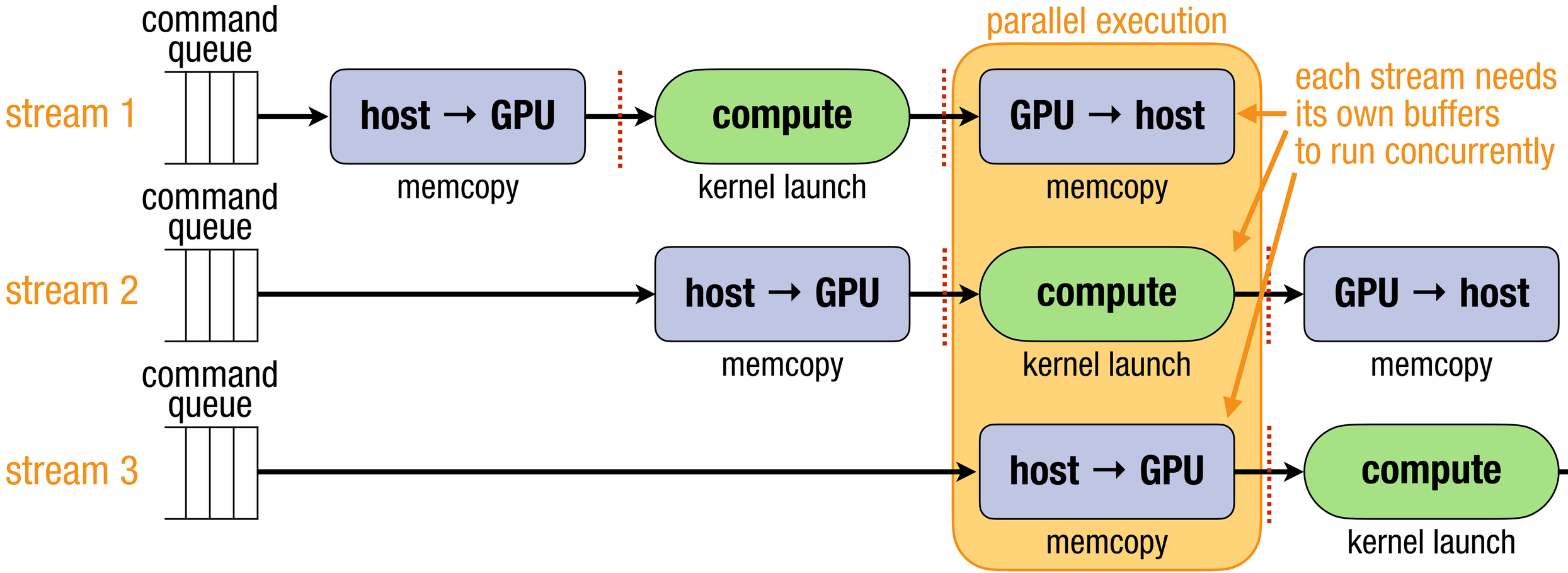
How does it **synchronize** these operations?



How can we **overlap** implicitly-ordered operations?



Multiple command queues (“streams”) allow **parallelism & overlapping**



Overlapping compute & **data movement** is essential for performance

Recurs at all **levels** of memory hierarchy and in many **different accelerators**

Common **design patterns** (double buffering, async DMA…)

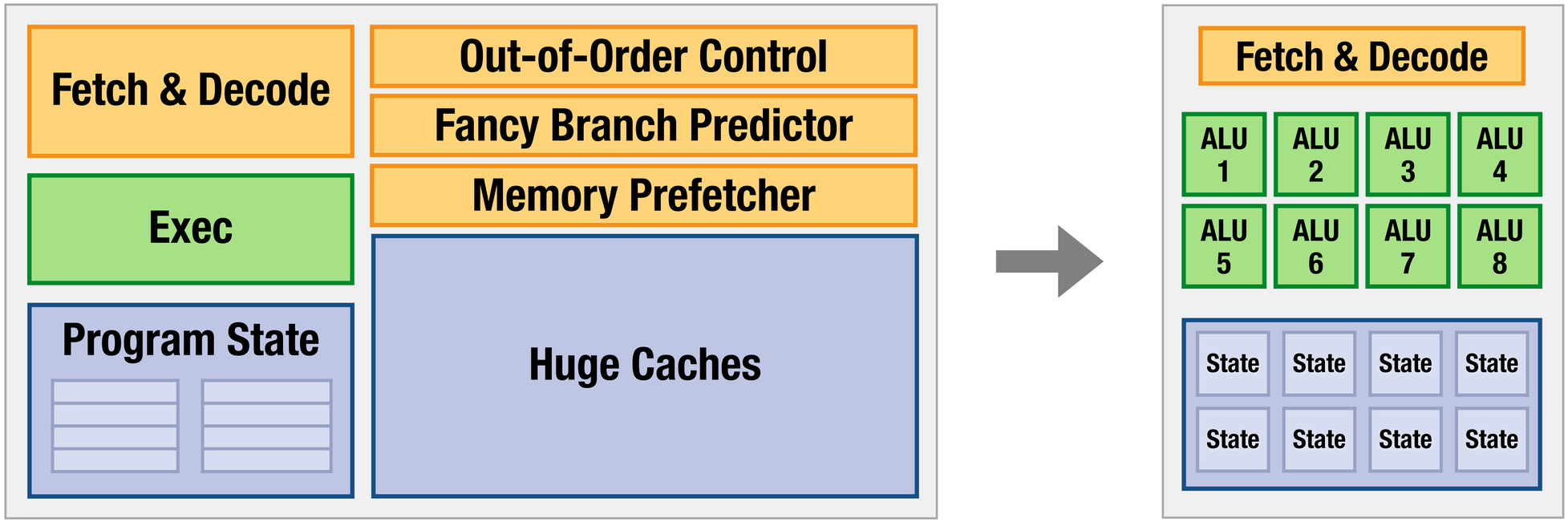
October 9, 2025

6.S894 Accelerated Computing

Lecture 6: Energy & Specialization

Jonathan Ragan-Kelley

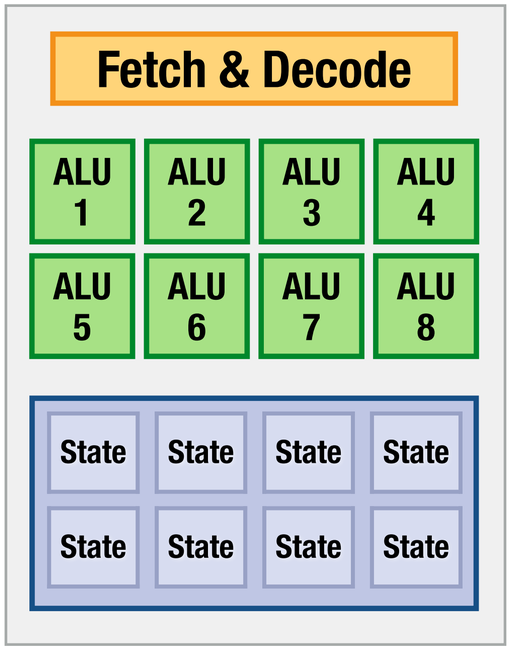
Throughput processor eliminates most **control overhead**

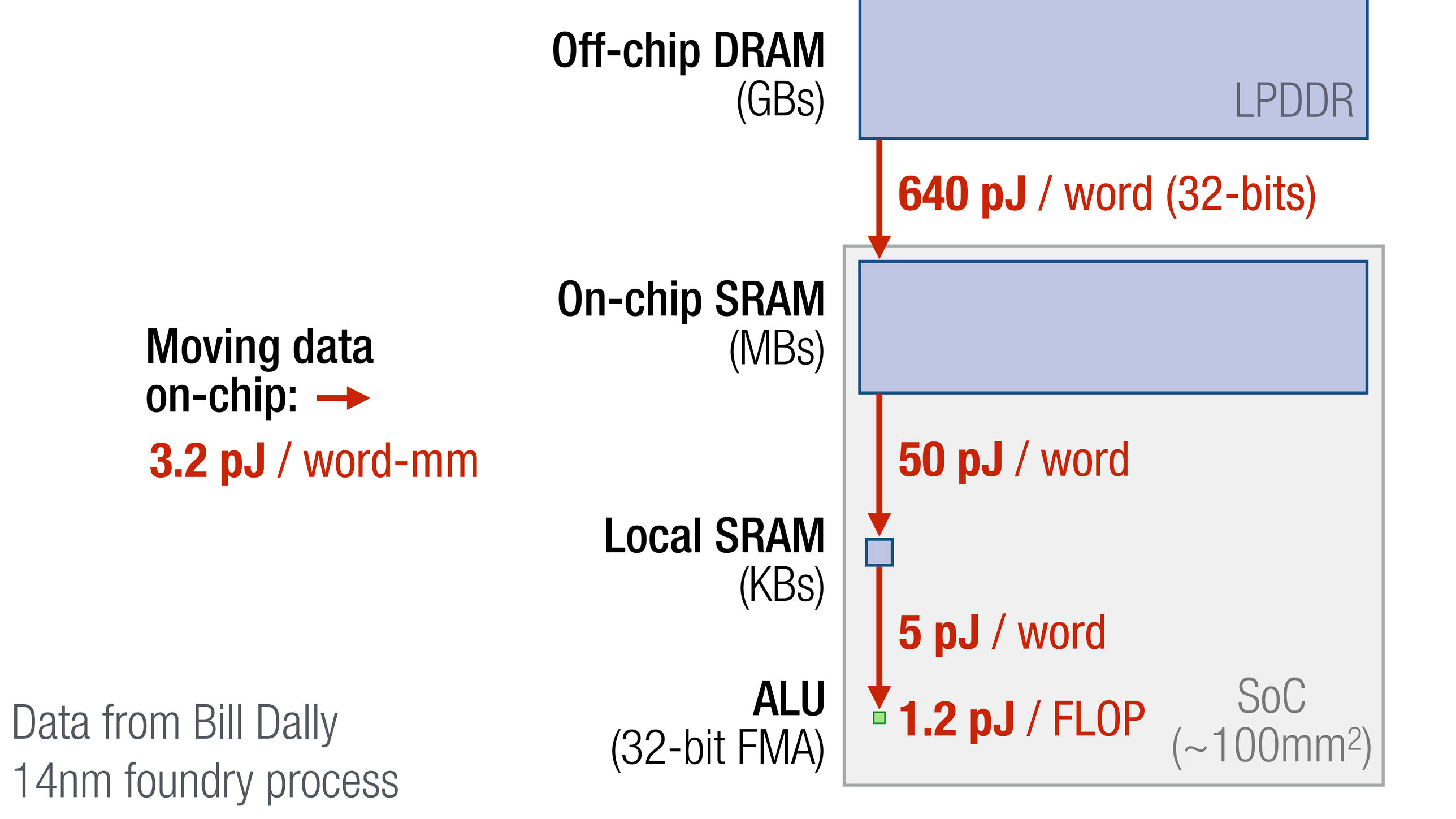


Throughput processor eliminates most **control overhead**

↳ most silicon goes to **useful work**

What about **energy?**



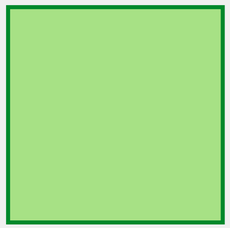

Off-chip DRAM
(GBs)
LPDDR
640 pJ / word (32-bits)
On-chip SRAM
(MBs)
Moving data
on-chip: →
3.2 pJ / word-mm
50 pJ / word
Local SRAM
(KBs)
5 pJ / word
ALU
(32-bit FMA)
1.2 pJ / FLOP
SoC
(~100mm²)
Data from Bill Dally
14nm foundry process

Data from Bill Dally
14nm foundry process

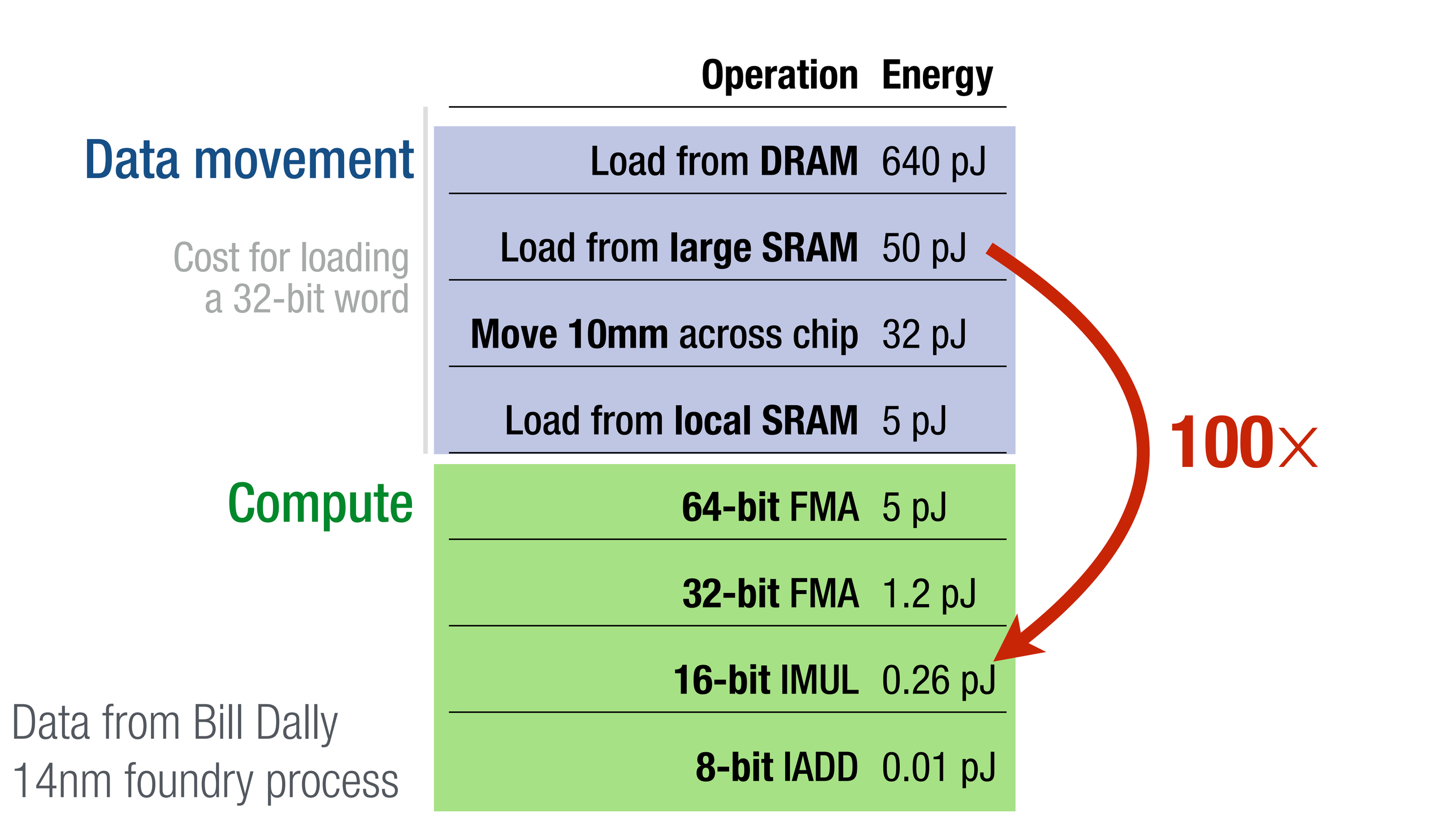
8-bit IADD **0.01 pJ** / OP

16-bit IMUL  **0.26 pJ** / OP

32-bit FMA  **1.2 pJ** / FLOP

64-bit FMA  **5 pJ** / FLOP

Data from Bill Dally
14nm foundry process


Operation Energy
Data movement
Cost for loading a 32-bit word
Load from DRAM 640 pJ
Load from large SRAM 50 pJ
Move 10mm across chip 32 pJ
Load from local SRAM 5 pJ
Compute
64-bit FMA 5 pJ
32-bit FMA 1.2 pJ
16-bit IMUL 0.26 pJ
8-bit IADD 0.01 pJ
100×
Data from Bill Dally
14nm foundry process

Data supply dwarfs computation
for primitive ALU operations

Operation	Data Supply **Overhead**
64-bit FMA	72%
32-bit FMA	84%
16-bit FMA	91%
8-bit IMAC	96%
32-bit IADD	99%

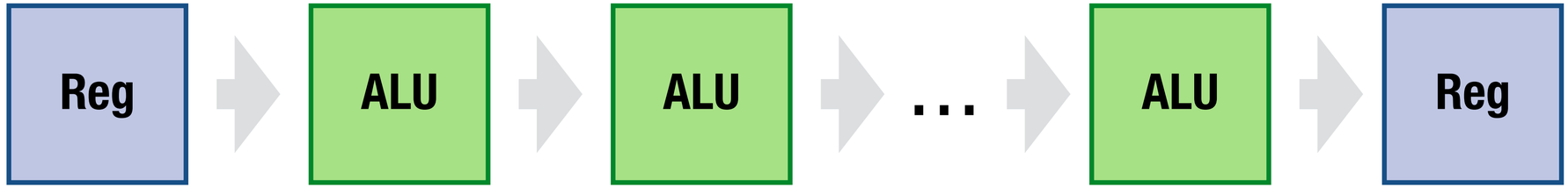
To be efficient, we need to **reduce data supply cost**

Assuming register file access costs 1.2 pJ / 32-bit word

Amortize data-supply with more **complex instructions**

Primitive op:
("RISC")

Complex op:
("CISC++")



e.g., AES, video encode/decode, DSP, texture filtering, …

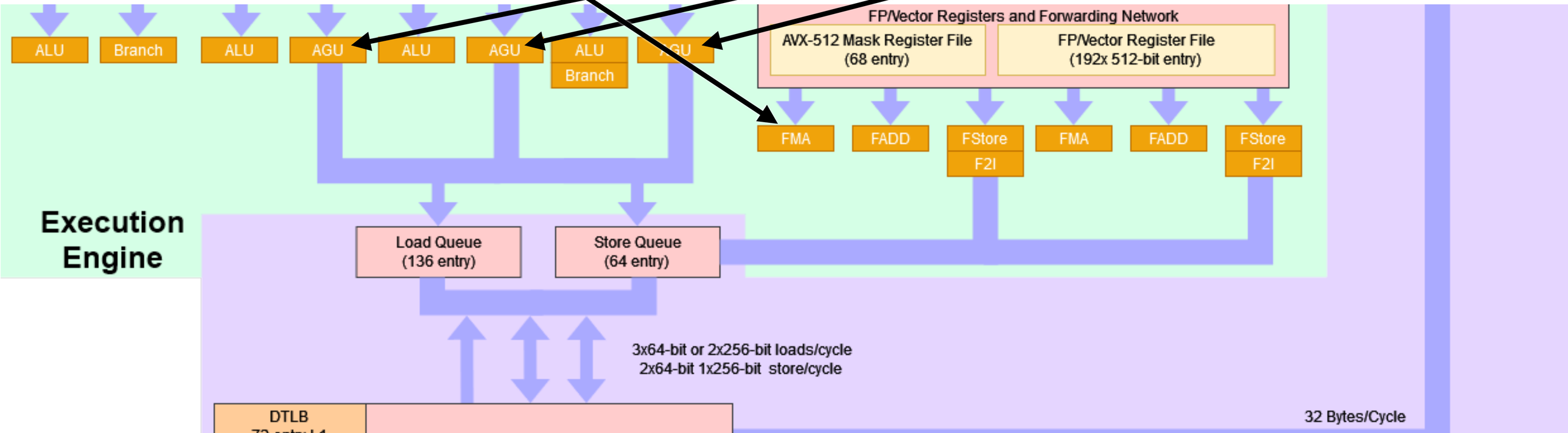
"ASIC-in-an-Instruction"

CISC: x86 memory operands

Perform FMA operation: `zmm2 += zmm0 * [memory]`

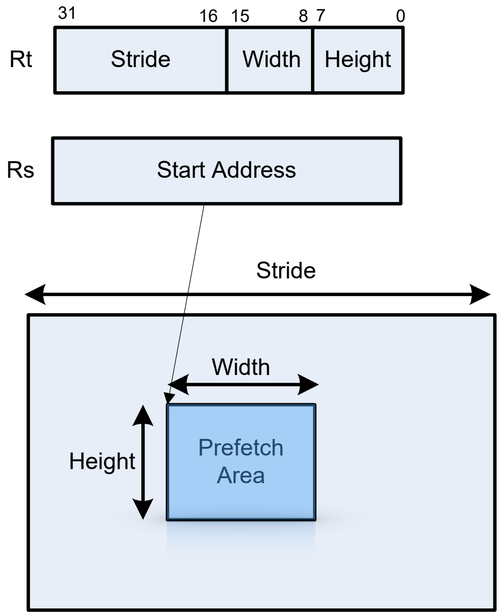
```
vfmadd231ps zmm2, zmm0, [rax + rcx*4 + 0x1234]
```

base + index × scale + displacement



Complex memory operations

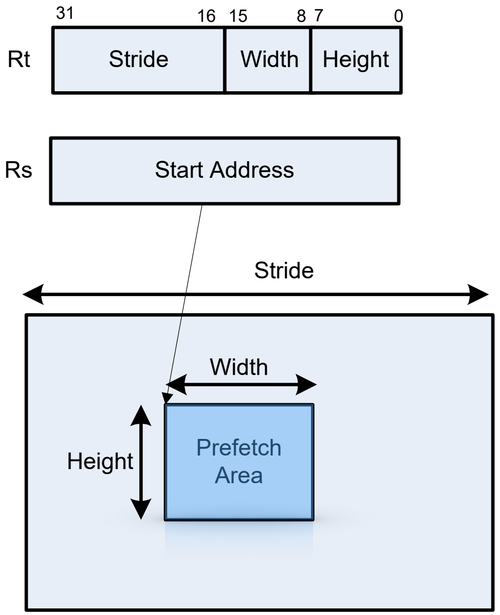
```
l2fetch *base, width,
        height, stride
```



Qualcomm Hexagon
DSP / NPU

Complex memory operations

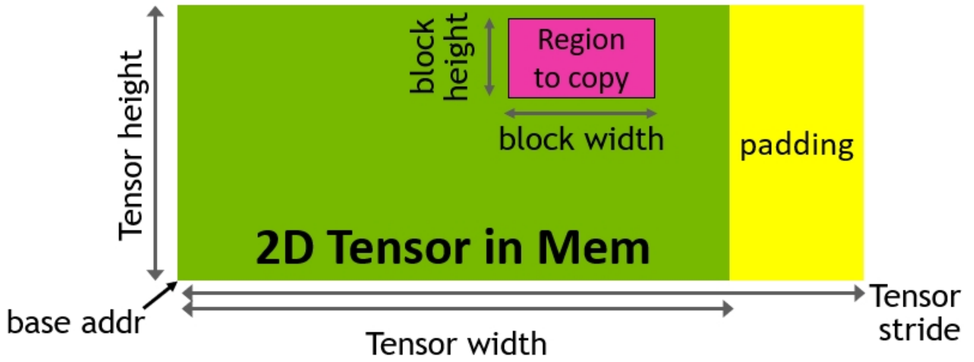
```
l2fetch *base, width,
        height, stride
```

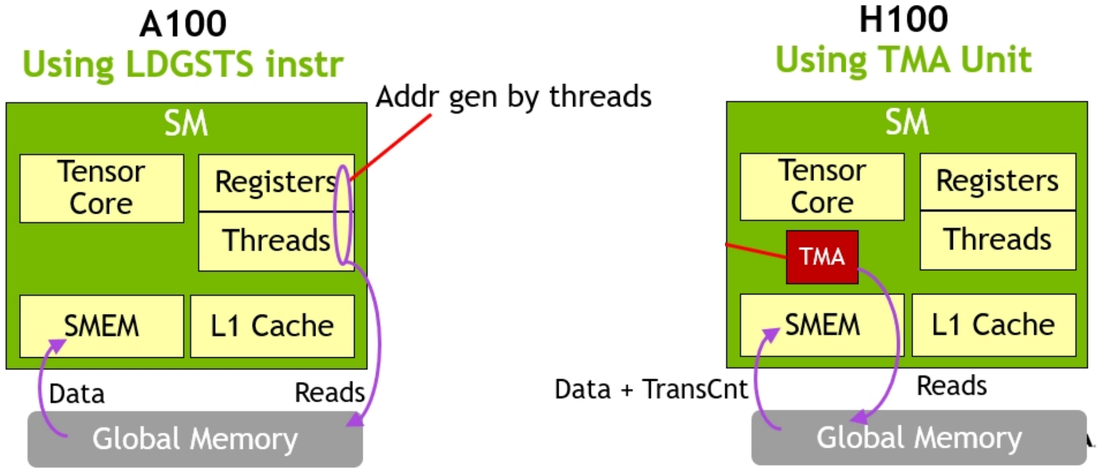


Qualcomm Hexagon
DSP / NPU

NVIDIA H100
Tensor Memory Accelerator

```
cp.async
→ cp.async.bulk.tensor
```

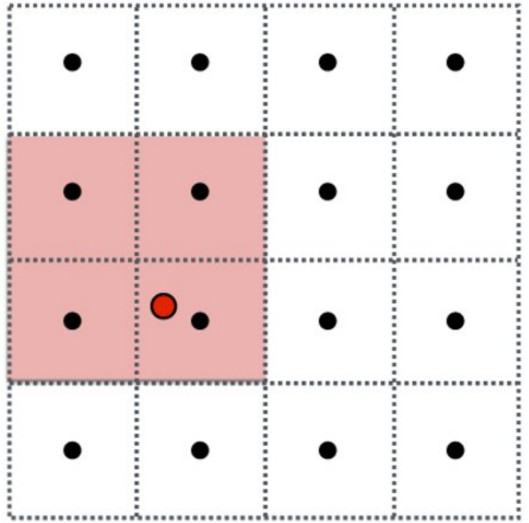




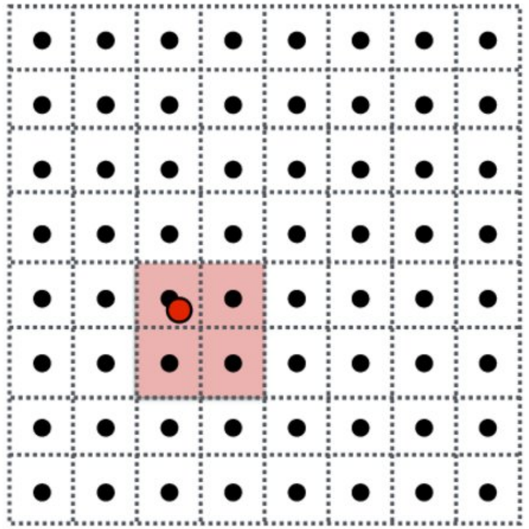
Texture mapping

One instruction:

- computes many addresses
- loads
- blends results

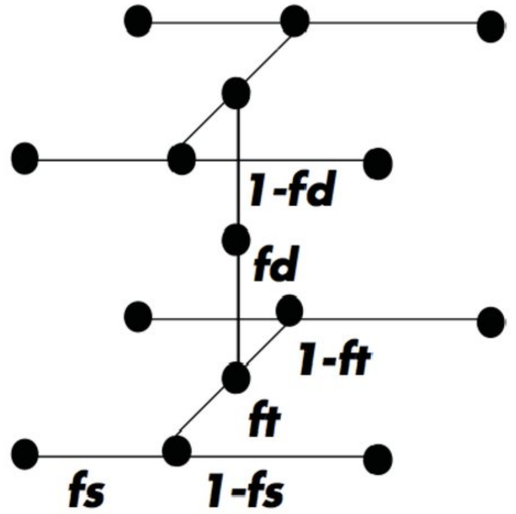


mip-map texels: level d+1



mip-map texels: level d

"Tri-linear" filtering



$$lerp(t, v_1, v_2) = v_1 + t(v_2 - v_1)$$

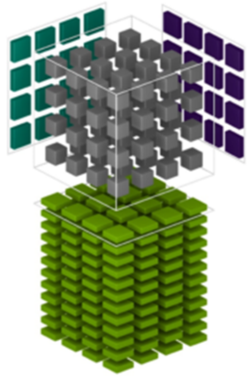
Bilinear resampling:
four texel reads
3 lerps (3 mul + 6 add)

Trilinear resampling:
eight texel reads
7 lerps (7 mul + 14 add)

Figs by Akeley, Hanrahan, Fatahalian

The quintessential complex instruction: **Matrix block multiply accumulate**

$$D = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}$$



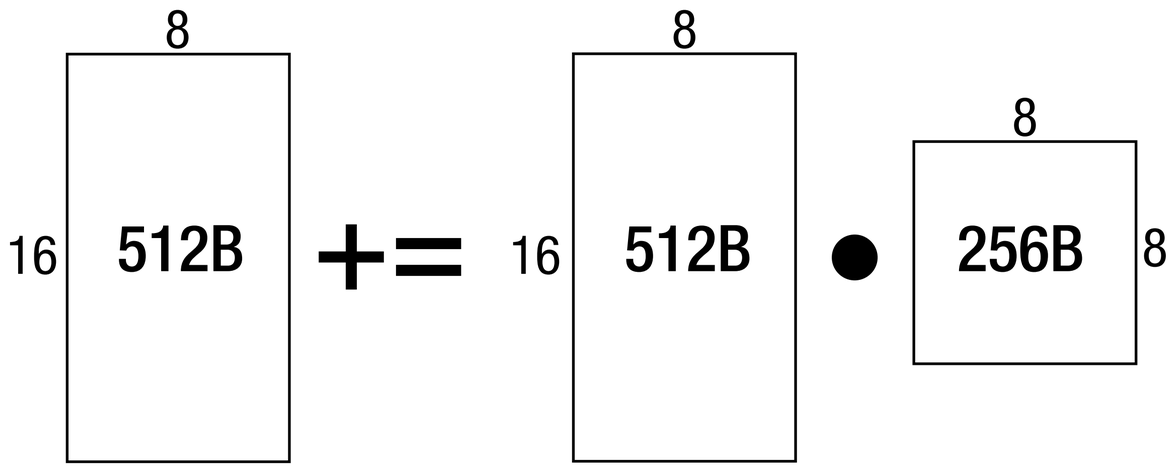
One instruction!

Examples:
ARM SME
AVX VNNI
Intel AMX
Google TPU
Neural Engine
Hexagon NPU
…

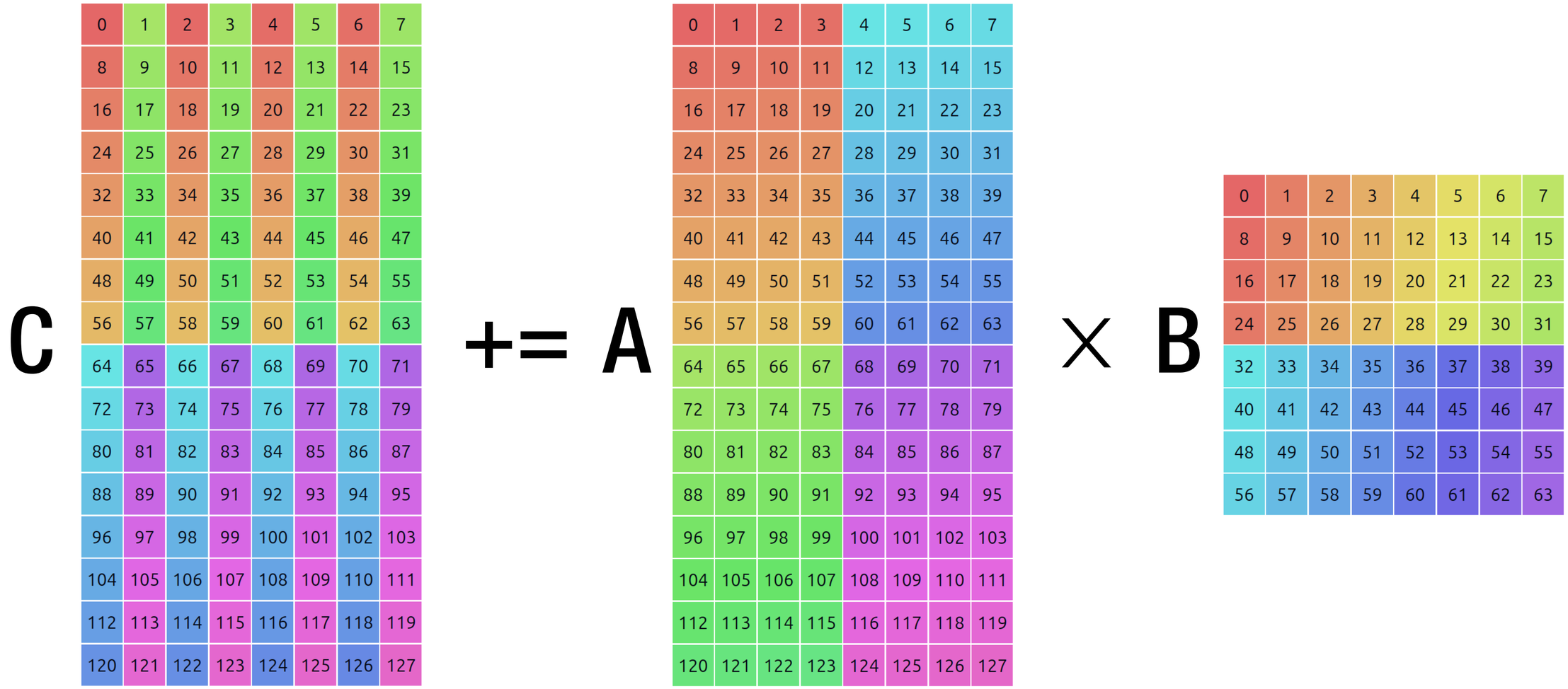
Tensor Cores
MMA on NVIDIA GPUs

PTX:
`mma.sync.aligned.m16n8k8`

Hardware (SASS):
`HMMA.1688.F32.TF32`



MMA operands packed into registers



Ampere TF32 16x8x8 Tensor Core 'A' Matrix Layout

Logical Layout

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63
64	65	66	67	68	69	70	71
72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87
88	89	90	91	92	93	94	95
96	97	98	99	100	101	102	103
104	105	106	107	108	109	110	111
112	113	114	115	116	117	118	119
120	121	122	123	124	125	126	127

Register Layout

Lanes

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Reg 0	0	1	2	3	8	9	10	11	16	17	18	19	24	25	26	27	32	33	34	35	40	41	42	43	48	49	50	51	56	57	58	59
Reg 1	64	65	66	67	72	73	74	75	80	81	82	83	88	89	90	91	96	97	98	99	104	105	106	107	112	113	114	115	120	121	122	123
Reg 2	4	5	6	7	12	13	14	15	20	21	22	23	28	29	30	31	36	37	38	39	44	45	46	47	52	53	54	55	60	61	62	63
Reg 3	68	69	70	71	76	77	78	79	84	85	86	87	92	93	94	95	100	101	102	103	108	109	110	111	116	117	118	119	124	125	126	127

Ampere TF32 16x8x8 Tensor Core 'B' Matrix Layout

Logical Layout

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

Register Layout

	Lanes																													
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Reg 0	0	8	16	24	1	9	17	25	2	10	18	26	3	11	19	27	4	12	20	28	5	13	21	29	6	14	22	30	7	15	23	31
Reg 1	32	40	48	56	33	41	49	57	34	42	50	58	35	43	51	59	36	44	52	60	37	45	53	61	38	46	54	62	39	47	55	63

Ampere TF32 16x8x8 Tensor Core 'C' Matrix Layout

Logical Layout

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63
64	65	66	67	68	69	70	71
72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87
88	89	90	91	92	93	94	95
96	97	98	99	100	101	102	103
104	105	106	107	108	109	110	111
112	113	114	115	116	117	118	119
120	121	122	123	124	125	126	127

Register Layout

Lanes

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Reg 0	0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38	40	42	44	46	48	50	52	54	56	58	60	62
Reg 1	1	3	5	7	9	11	13	15	17	19	21	23	25	27	29	31	33	35	37	39	41	43	45	47	49	51	53	55	57	59	61	63
Reg 2	64	66	68	70	72	74	76	78	80	82	84	86	88	90	92	94	96	98	100	102	104	106	108	110	112	114	116	118	120	122	124	126
Reg 3	65	67	69	71	73	75	77	79	81	83	85	87	89	91	93	95	97	99	101	103	105	107	109	111	113	115	117	119	121	123	125	127

Tensor Cores
MMA on NVIDIA GPUs

1 instruction

10 register operands

2048 FLOPs / 8 cycles

MMA instructions
amortize energy overhead

Instruction	Ops	Control & data **overhead**
HMMA (fp16)	128	19%
IMMA (int 8)	1024	12%

NVIDIA Tensor Cores

data from Bill Dally

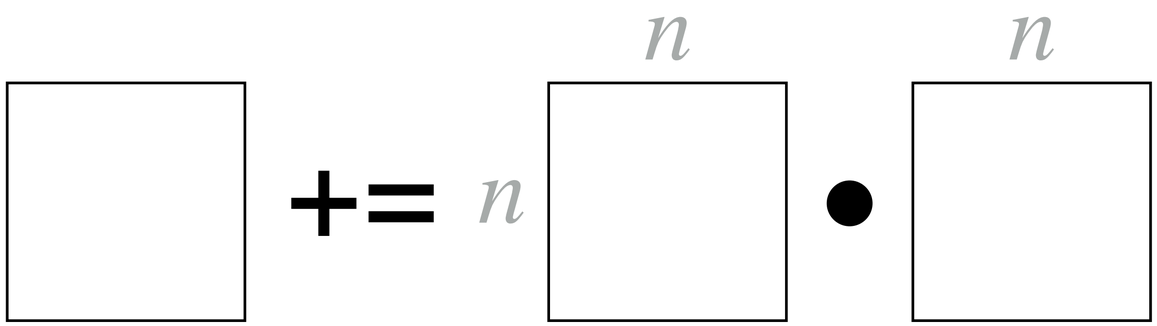
H100

Vector: 67 TF

MMA: 990 TF

Vector <7% peak

Matrix multiply has **high <u>arithmetic intensity</u>**



FLOPS: $2n^3$

Entries: $3n^2$

↳ **ratio grows with** n

$$\textbf{arithmetic intensity} = \frac{\text{work (ops)}}{\text{data (bytes)}}$$

Arithmetic intensity is what
makes matrix multiply special

All “acceleratable computations”
have high arithmetic intensity

Questions?