PROGRAMMABLE ACCELERATORS: HISTORICAL BLIP OR FUNDAMENTAL TENTPOLE?

NVIDIA

Michael Pellauer, PhD
Senior Research Scientist
Architecture Research Group
mpellauer@nvidia.com

COMPUTATIONAL EFFICIENCY IMPROVEMENTS

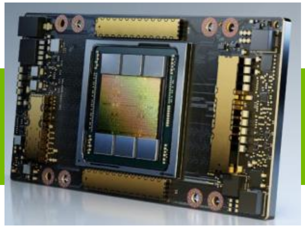
Modern world has come to rely on regular cadence

GK210, 2014

28 nm process, 7.1 billion transistors

GV100, 2017

12 nm process, 21.1 billion transistors



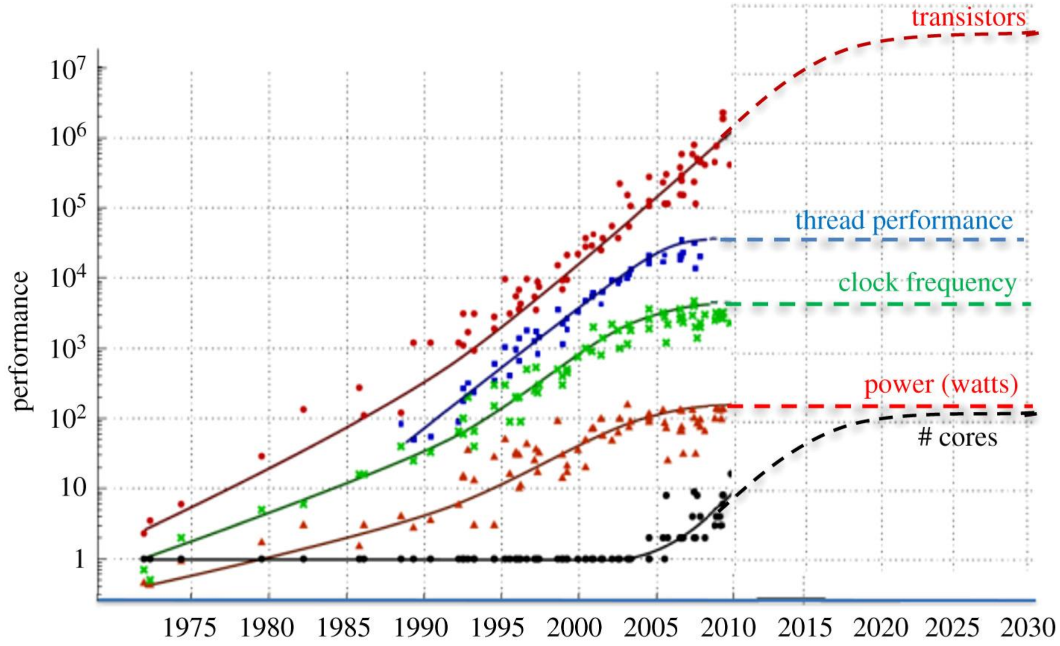
GA100, 2020

7 nm process, 54.2 billion transistors

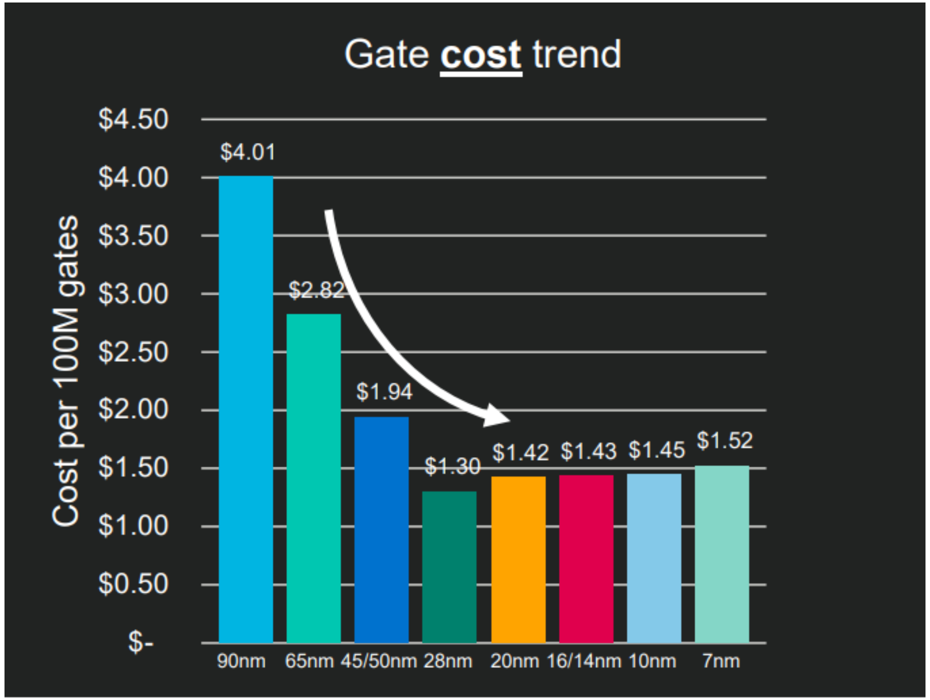
Enabling:

SLOWING TRANSISTOR SCALING

"Oil well" has had easy stuff extracted already



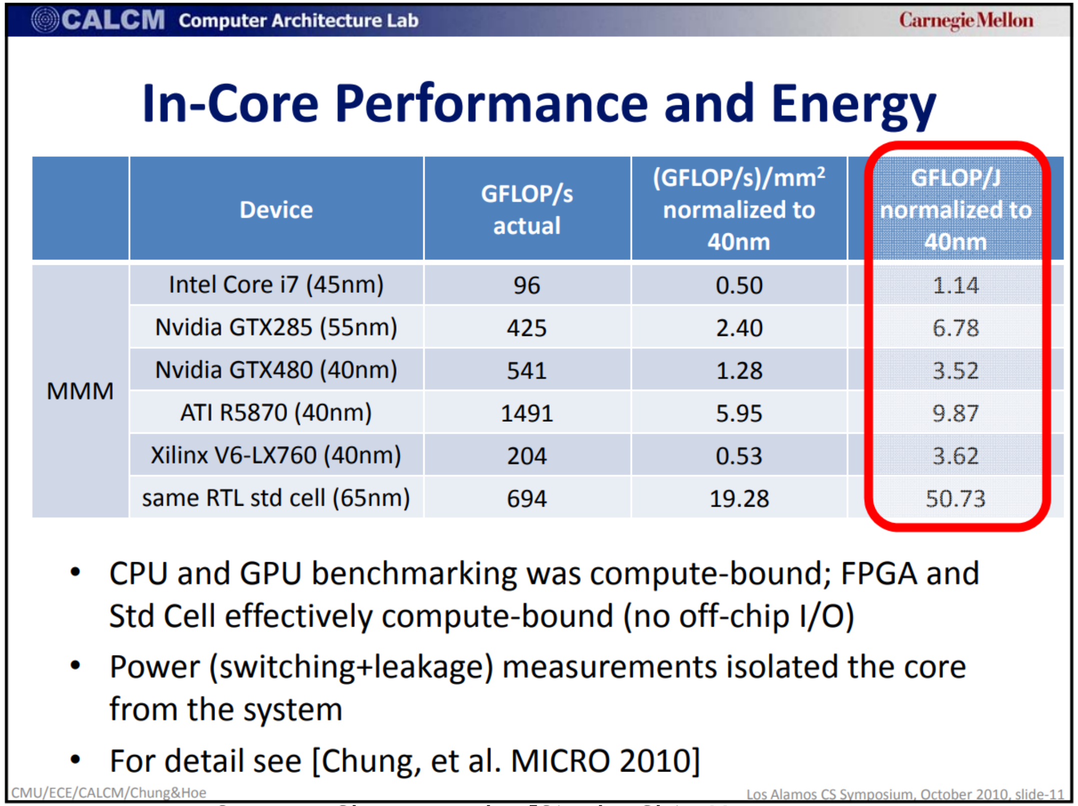
Source: Shalf, [The Future of Computing Beyond Moore's Law], projecting from data points by Olukotun, Hammond, Sutter, and Horowitz



Source: O'Laughlin, [The Rising Tide of Semiconductor Cost], data from International Business Strategies, Inc. and Marvell

HARDWARE SPECIALIZATION

Alternative “oil well” to increase efficiency



CALCM Computer Architecture Lab — Carnegie Mellon

In-Core Performance and Energy

	Device	GFLOP/s actual	(GFLOP/s)/mm^2 normalized to 40nm	GFLOP/J normalized to 40nm
MMM	Intel Core i7 (45nm)	96	0.50	1.14
	Nvidia GTX285 (55nm)	425	2.40	6.78
	Nvidia GTX480 (40nm)	541	1.28	3.52
	ATI R5870 (40nm)	1491	5.95	9.87
	Xilinx V6-LX760 (40nm)	204	0.53	3.62
	same RTL std cell (65nm)	694	19.28	50.73

- CPU and GPU benchmarking was compute-bound; FPGA and Std Cell effectively compute-bound (no off-chip I/O)
- Power (switching+leakage) measurements isolated the core from the system
- For detail see [Chung, et al. MICRO 2010]

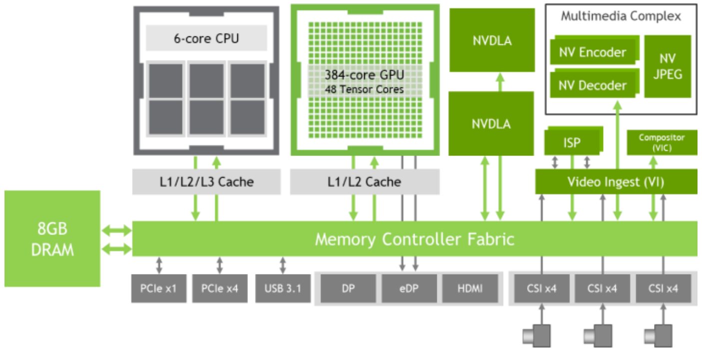
CMU/ECE/CALCM/Chung&Hoe — Los Alamos CS Symposium, October 2010, slide-11

Source: Chung et al., [Single-Chip Heterogeneous Computing: Does the Future Include Custom Logic, FPGAs, and GPGPUs?] 2010

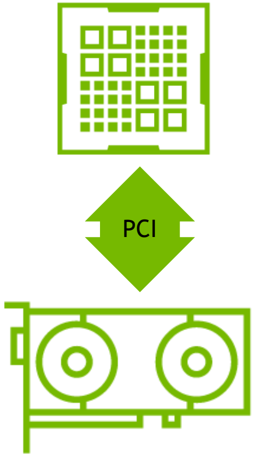
THE AGE OF ACCELERATORS

Offload work from Turing-Complete CPU “Jack-of-all-Trades”

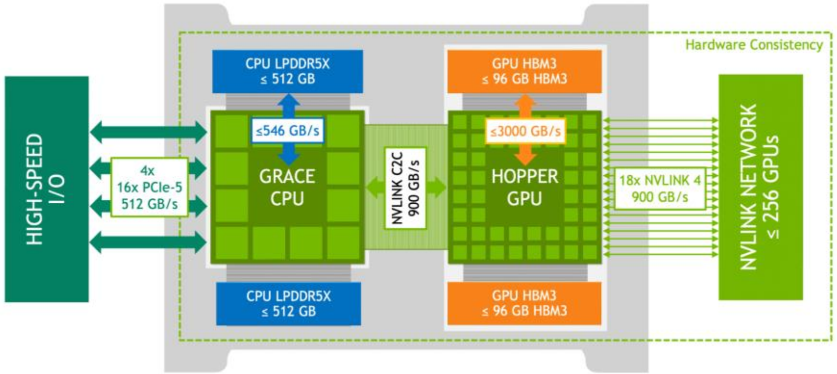
NVIDIA Jetson Xavier NX SoC



Desktop CPU + GeForce GPU



NVIDIA Grace Hopper Superchip



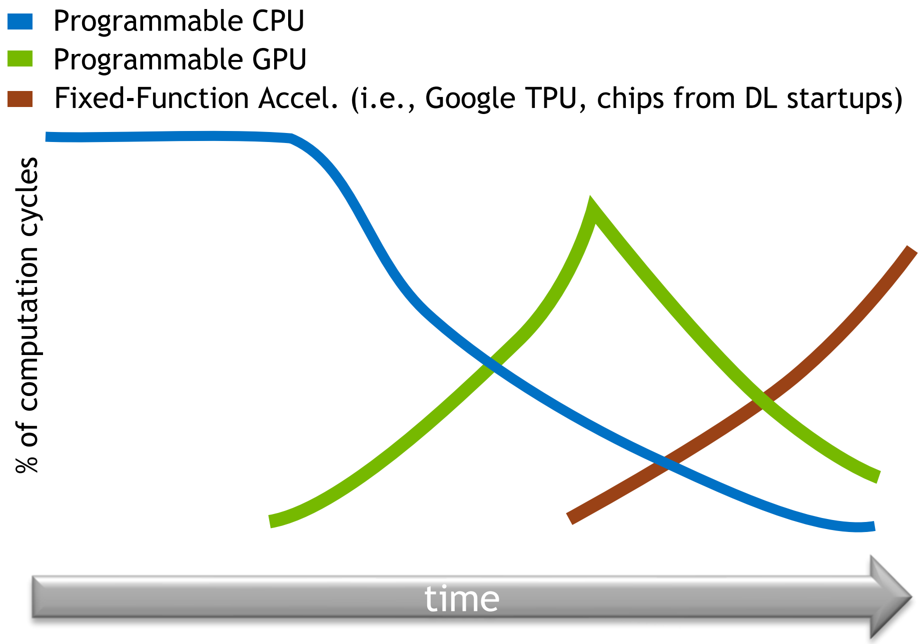
Heterogeneous System-On-Chip
(mobile, automotive)

Discrete Cards
(desktop graphics)

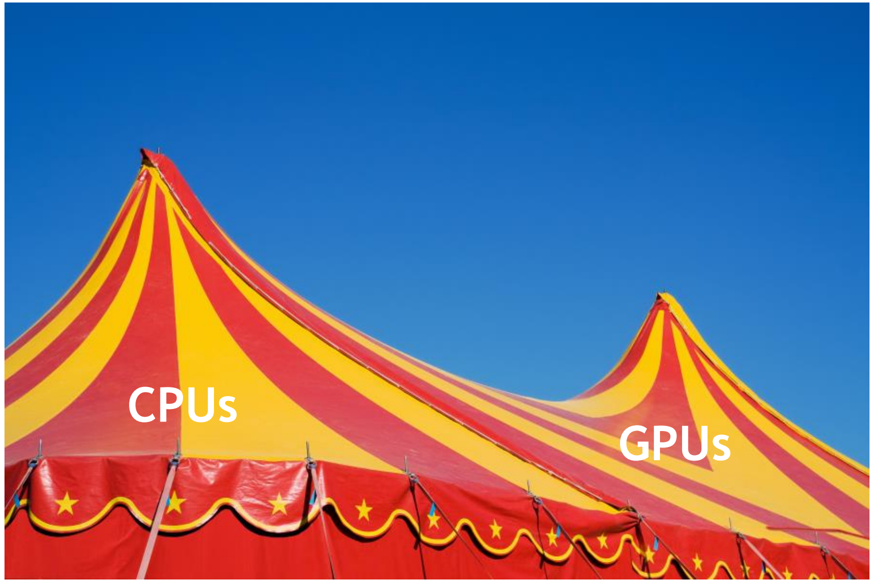
Multi-Chip Package
(datacenter AI)

TODAY'S QUESTION

If specialization increases efficiency, why are GPUs programmable?



Or



Programmable accelerators: historical blip or fundamental tentpole?

APPROACH: "A-HISTORICAL"

Put aside graphics/gaming for now

Sometimes it can be freeing to discard what evolved through history

- Re-examine using a lens of fundamental principles
- Avoid marketing or historical terminology to use unified vocabulary across parts

Goal: be "a-historical" but not "un-historical" or apocryphal

- Zoom in on the truly fundamental forces driving modern computer architecture
- Add history back at the end

Feel free to ask questions throughout (even those involving history!)

THOUGHT EXPERIMENT

MS Office Stock CEO Photo

Good news! All computer graphics and Deep Learning can now be done just using a single workload:

Vector-Vector Multiplication!

Forget all that fancy linear algebra stuff!

Make me a product that just does this one workload with highest performance possible!

```
for n in range(0, N):
  Z[n] = A[n] * B[n]
```

2	1	2
4	3	12
6	5	30
8	7	56
10	9	90
…	…	…
100	99	990

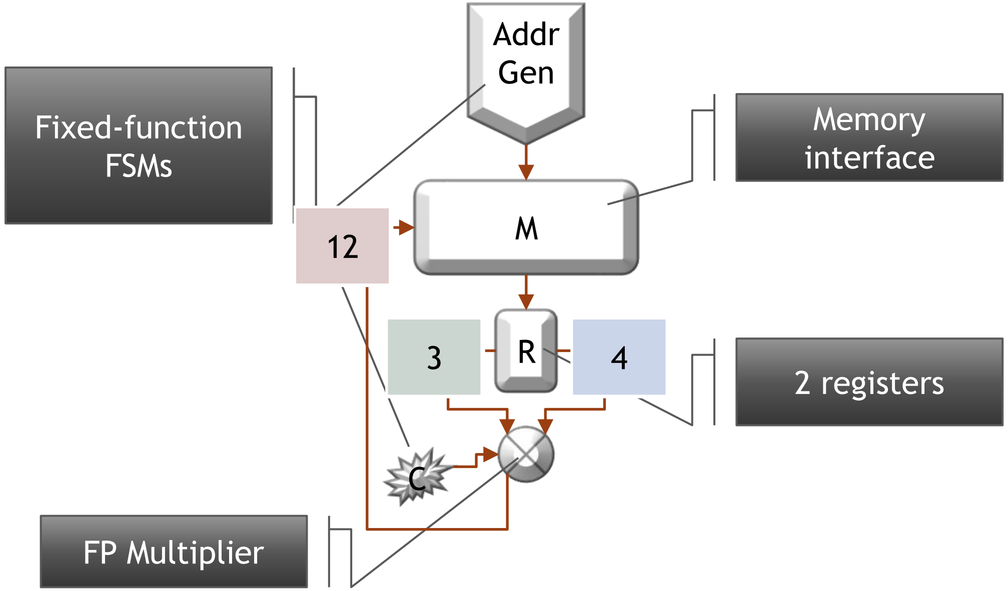
Let’s call this theoretical accelerator “VVMul”

SIMPLE VVMUL

Off-chip memory provides 1 Word in OR out per cycle

```
for n in range(0, N):
  Z[n] = A[n] * B[n]
```

2	1	2
4	3	12
6	5	30
8	7	56
10	9	90
…	…	…
100	99	990



Throughput: 1 FPMul every 3 cycles

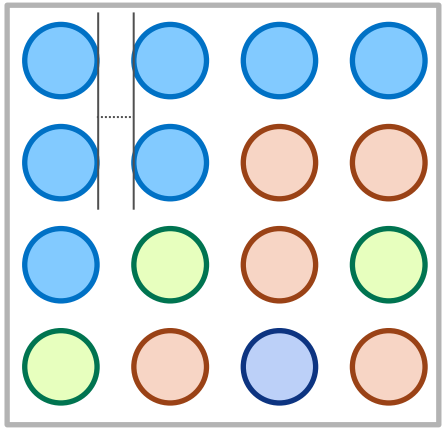
Die size: tiny

Area dominated by M Unit and its pins, rest is trivial

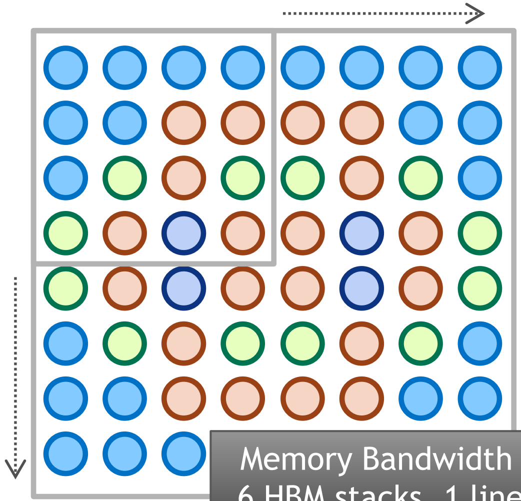
SCALING MEMORY BANDWIDTH

Simplified – you can spend your career studying just this problem

Step 1: Maximize Bandwidth density in given area

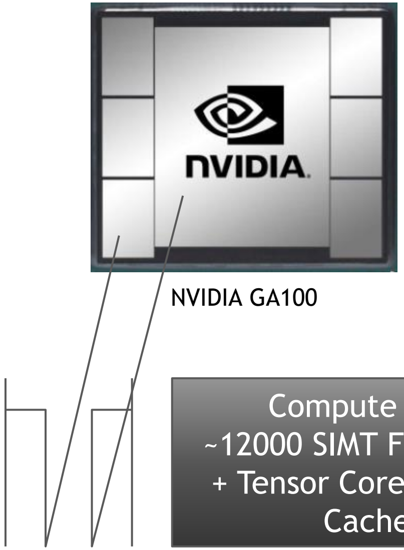


Step 2: Scale up die* area



Memory Bandwidth (ballpark)
6 HBM stacks, 1 line per cycle
64 words (32-bit) per line
= 384 words per cycle

Step 3: Etch compute and SRAM transistors into resulting area



NVIDIA GA100

Compute Bandwidth
~12000 SIMT FPMul datapaths
+ Tensor Cores, Ray Tracing,
Caches, etc.

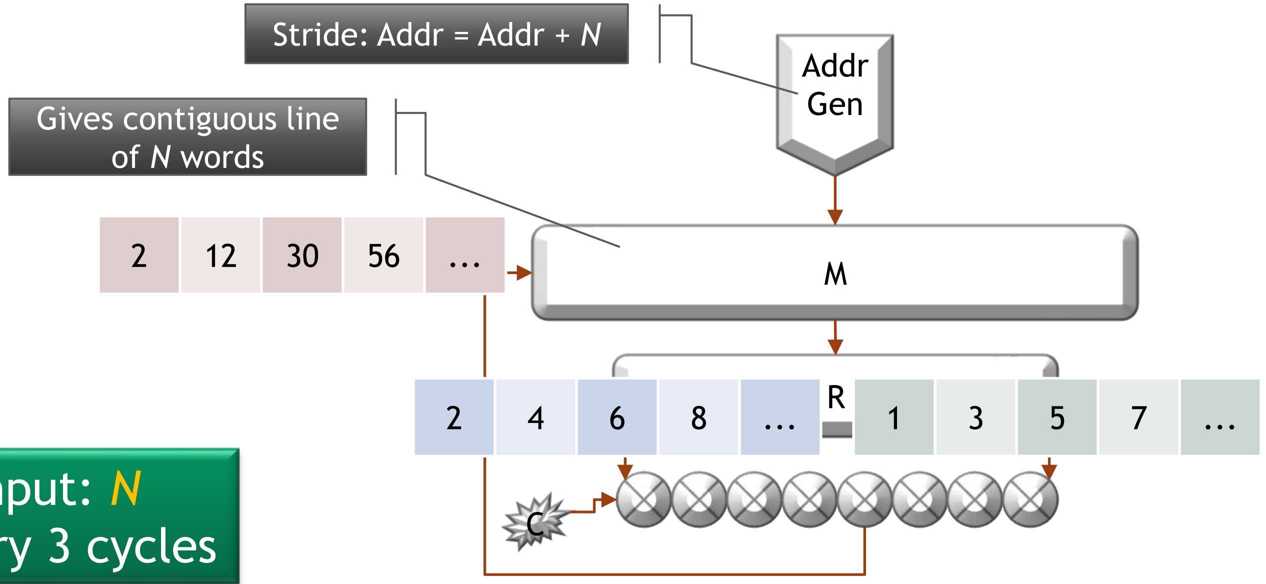
* In practice, there are times where “large package with multiple chips” is equivalent to “large die” (out of scope for today)

VVMUL + A100 MEMORY BANDWIDTH

Off-chip memory provides *N*=384 Words in OR out per cycle

```
for n in range(0, N):
  Z[n] = A[n] * B[n]
```

2	1	2
4	3	12
6	5	30
8	7	56
10	9	90
...	...	...
100	99	990



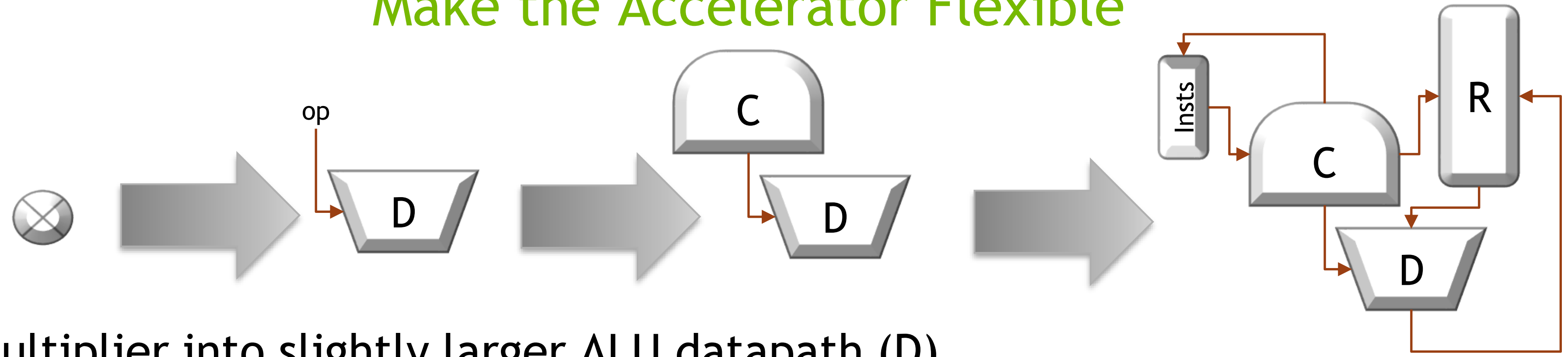
Throughput: *N* FPMuls every 3 cycles

Die size: large but empty

Only need 384 FPMuls, compared to ~12000 for A100!
Note: Benefit for unused die area is very minor (i.e., yield)
What to do with all that extra area?

PROPOSAL: MORE OPERATORS

Make the Accelerator Flexible

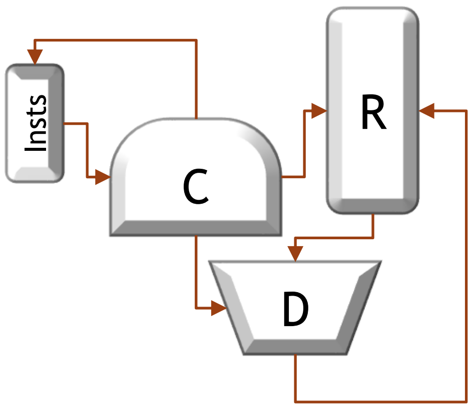


Turn multiplier into slightly larger ALU datapath (D)

- Add Control (C) block to indicate which operator to do currently
- Time-mux instructions from instruction memory (I) through ALU
- Add Register File (R) to move data from one operation to the next

Let's call this abstract arrangement of units a CIDR (pronounced "cider")

- To avoid confusing and ambiguous historical names (Core, Processor, SM, PE, etc.)



CIDR PROS AND CONS

Pros:

- Run arbitrary programs (in case people do still care about more than VVMul)
- Enables changes in the field after chip design time
- Enables “black swans” to emerge from users instead of company that designed chip
  - See [Joel Emer’s keynote to Young Architect’s Workshop (ASPLOS 2021)]

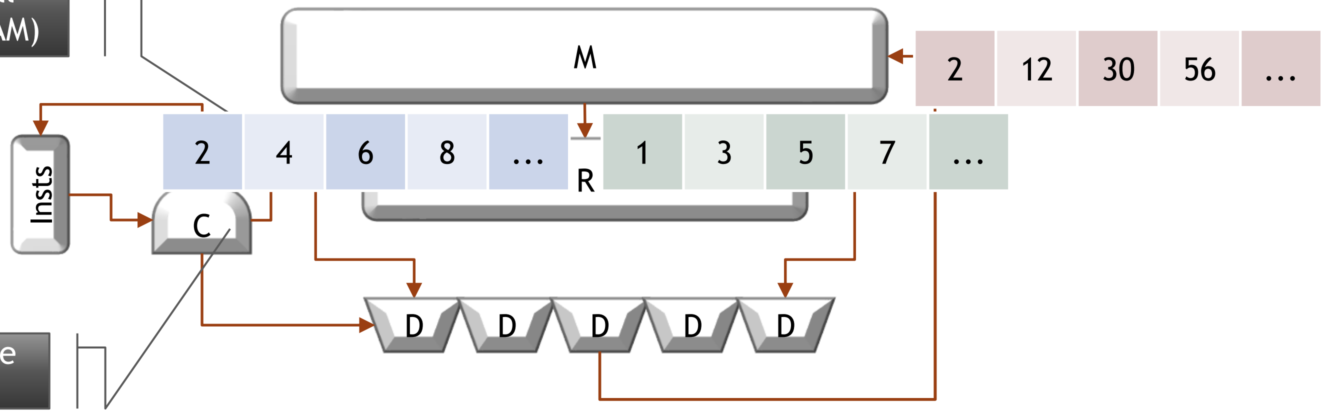
Cons:

- If D unit has latency >1, C must wait to issue dependent instructions
  - See: operand bypassing [and a million other techniques for optimizing processors]
- Latency of round-trip memory Load instructions (ignore for now, will discuss later)

FIRST-CUT PROGRAMMABLE VVMUL

Wide, contiguous line = Single Instruction Multiple Data (SIMD*)

Drive same index to all registers (i.e., wide SRAM)



Drive same opcode to all datapaths

Die size: large but mostly empty

Overhead of C and I Units amortized by a factor of *N*
N-wide M and R Units are also cheaper than *N* individual copies
But what performance did we achieve?

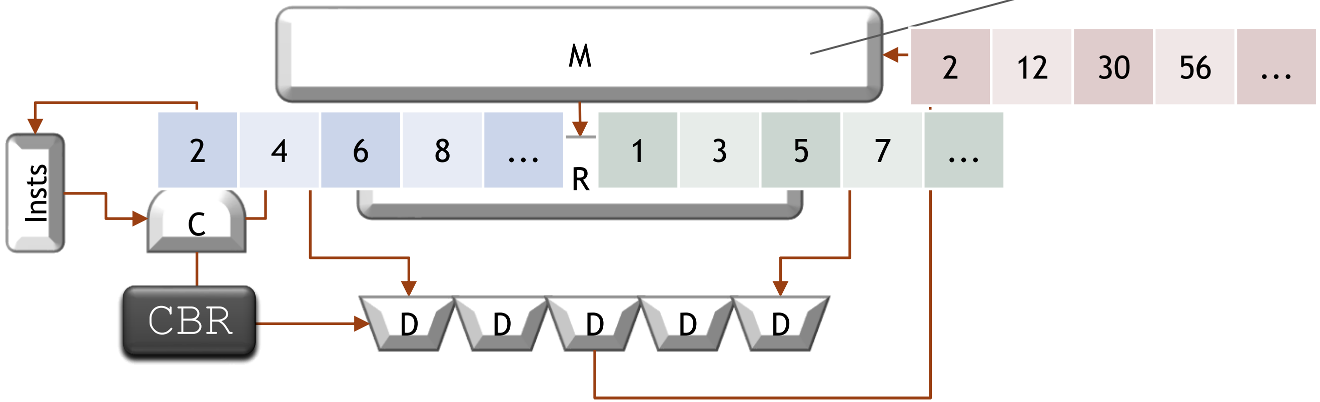
* Difference between SIMD and SIMT not relevant for today, see extra slides

VVMUL IN PROGRAMMABLE MODE

Demonstrates overheads of general-purpose programmability

```
loop:
  SIMD_LD r3 := @r1 + offset_A
  SIMD_LD r4 := @r1 + offset_B
  SIMD_MUL r5 := r3, r4
  SIMD_ST @r1 + offset_Z := r5
  INCR r1
  DECR r2
  CBRANCH loop if r2 > 0
```

Ignore memory latency for the moment to focus on throughput (discuss on future slide)



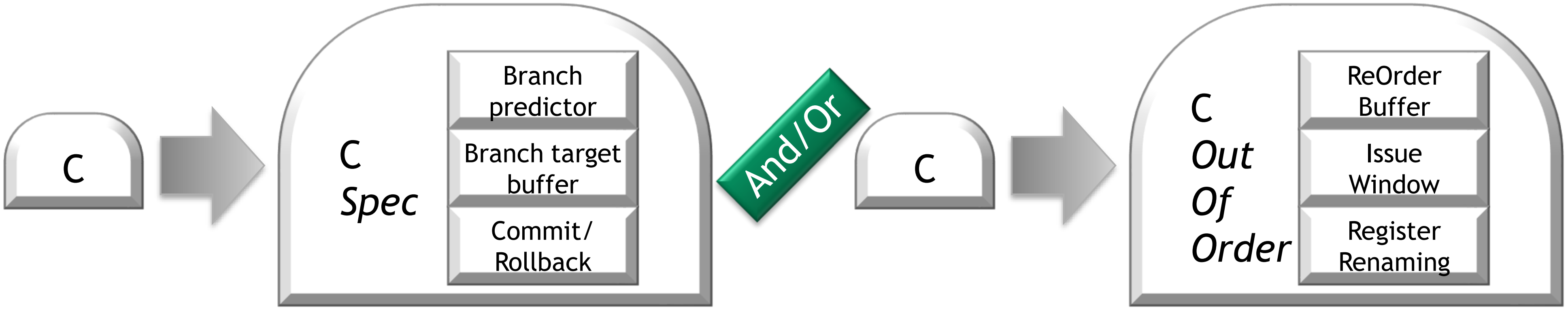
Throughput: *N* FPMuls every 7 cycles

Die size: large but empty

Note: minor energy tax also (dominated by DRAM in practice)
How to achieve performance parity?

APPROACH 1: INCREASE INSTS PER CYCLE

Out-of-Order, Superscalar, Speculation (i.e., modern CPU)



Two complementary techniques, often employed together

- Control (and value) speculation: bet that your instruction stream is predictable
- Out-of-order execution: Dynamically de-serialize PC into true dependency graph

Throughput: *N* FPMuls every 3 cycles

Die size: large but mostly empty

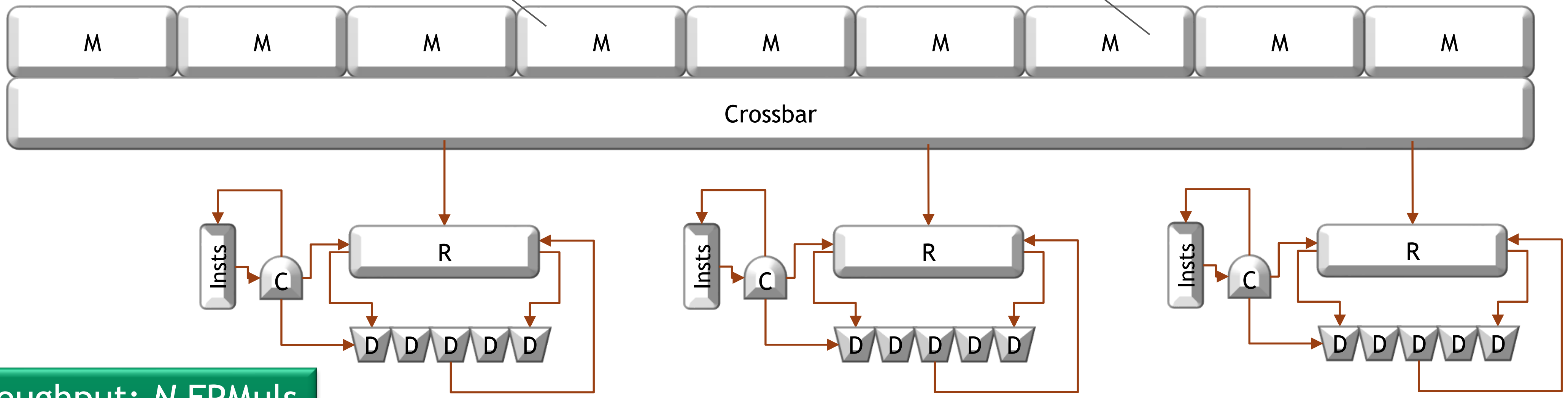
Why not faster? We call VVMul "memory bound"
Generally, means adding M Units makes it go faster
(since commensurate D units should fit in resulting area increase)

APPROACH 2: DUPLICATE, PARTITION WORK

I.e., Multiple Instruction Multiple Data (MIMD)

Split wide M into independent banks transferring smaller lines

Not shown: logic for mapping address to memory controller



Throughput: *N* FPMuls every 3 cycles

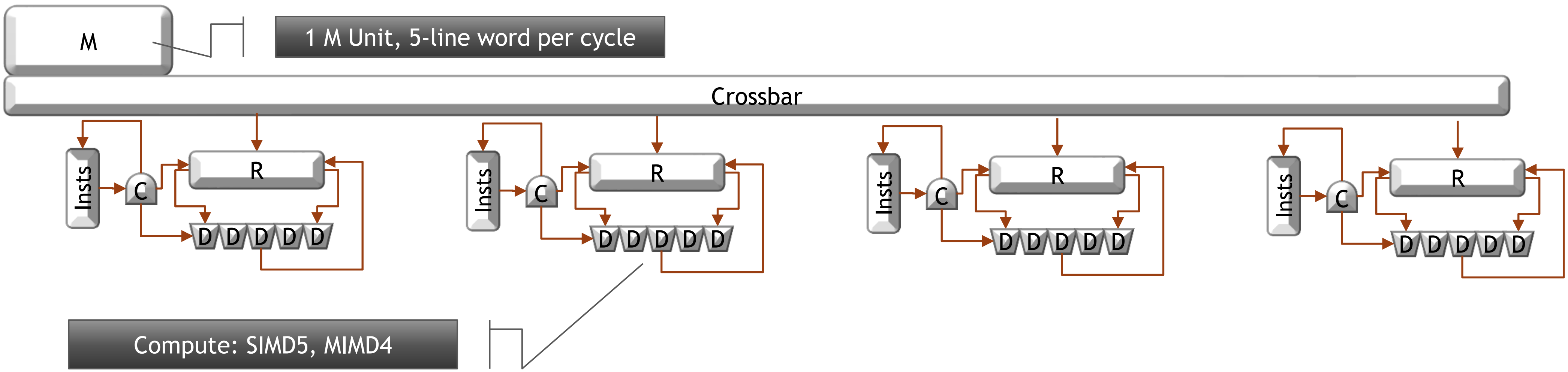
Die size: large, full

Still memory-bound (more precisely, memory bandwidth-bound)
But what if we do a problem that has actual data reuse?

Copy-and-paste until satisfied

COMPUTATION:MEMORY BANDWIDTH RATIO

Peak utilization is a matter of relative bandwidths, not absolutes

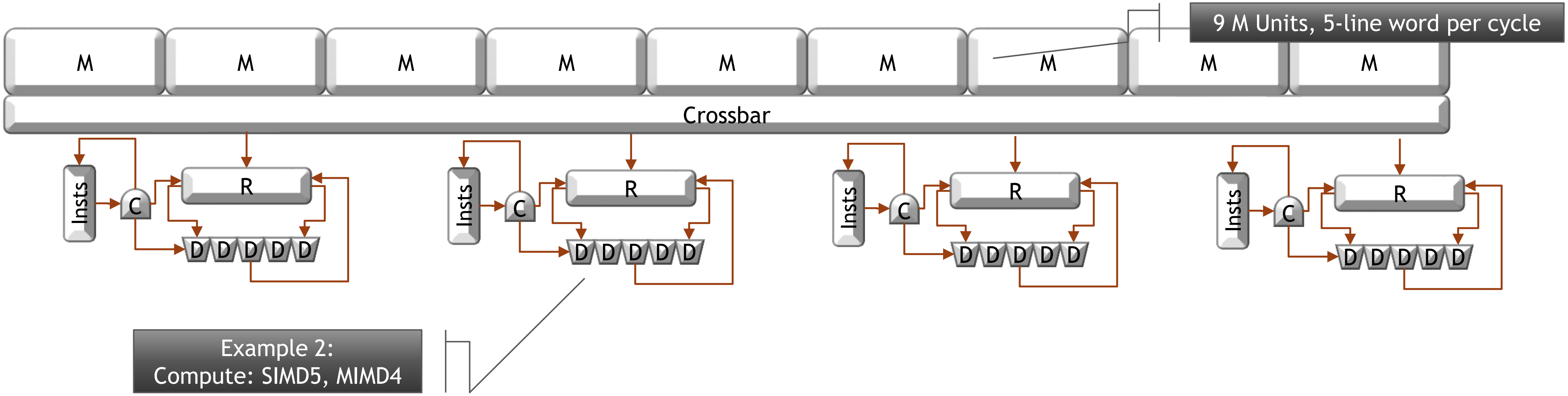


Saturate memory bandwidth:
At *least* 1 LD/ST every 4 cycles
from each CIDR

Saturate FPMul compute bandwidth:
At *most* 1 LD/ST every 4 cycles

COMPUTATIONAL THROUGHPUT

An unbalanced scenario

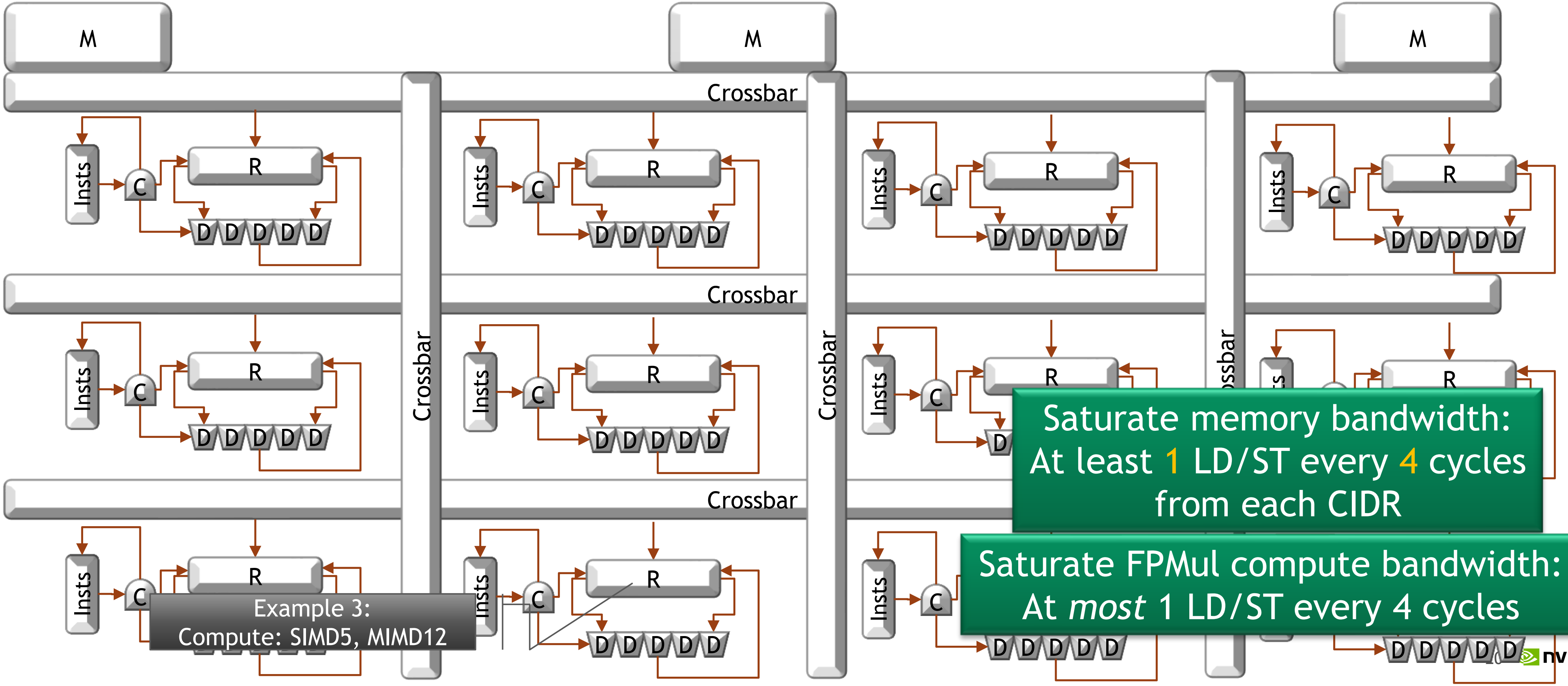


Saturate memory bandwidth:
Impossible, even if each CIDR generates 1 LD / cycle

Saturate FPMul compute bandwidth:
At *most* 1 LD/ST every 4 cycles

COMPUTATIONAL THROUGHPUT

Needed: better rules of thumb for how these ratios scale



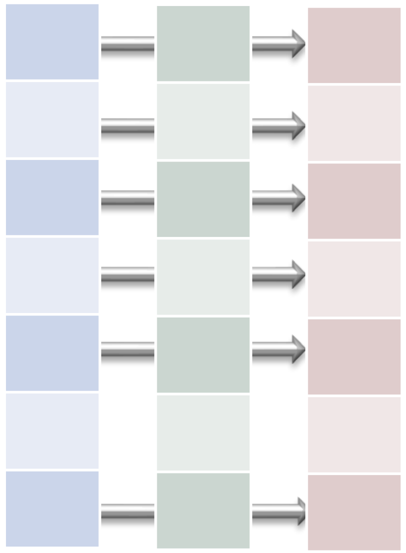
Saturate memory bandwidth:
At least 1 LD/ST every 4 cycles
from each CIDR

Saturate FPMul compute bandwidth:
At *most* 1 LD/ST every 4 cycles

INCREASING WORKLOAD DATA REUSE

Not as easy as one might think

```
for n in range(0, N):
  Z[n] = A[n] * B[n]
```



A: none, B: none, Z: none


```
for n in range(0, N):
  for k in range(0, K):
    Z[n] = A[n,k] * B[k]
```


A: none, B: O(N) XOR Z: O(K)
see also: [tiling techniques]

ARITHMETIC INTENSITY

Abstraction of potential data reuse (distinct from *achieved*)

Rule of thumb: there are known techniques to amortize, hide, or bury "C", both in area (i.e., SIMD) and perf (i.e., loop unrolling) whereas "D" is the real on-chip limiter

- Hence "arithmetic intensity" (also called "computational intensity")

$$Intensity = \frac{Total\ Ops}{Total\ Bytes\ from\ Offchip} = \frac{Ops}{Byte}$$

Nota Bene: Some authors count Multiply+Add (i.e., MACC) as 2 ops, some as 1

- Best practice for Deep Learning: just count FPMultiplications [personal opinion]

ARITHMETIC INTENSITY IN THE LIMIT

Two classes of workload: fixed or scaling data reuse

Elementwise:

$$\lim_{M\to\infty} \frac{M}{3M} = \frac{1}{3 * word\_size}$$

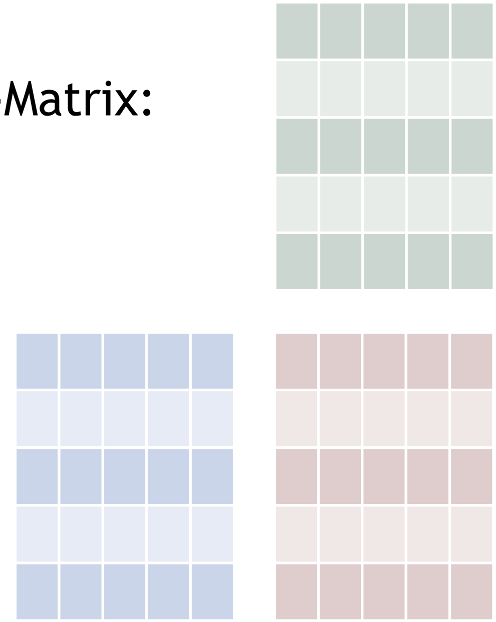
fixed

Matrix-Vector:

$$\lim_{K,N\to\infty} \frac{KN}{KN + K + N} = \frac{1}{word\_size}$$

fixed

Matrix-Matrix:



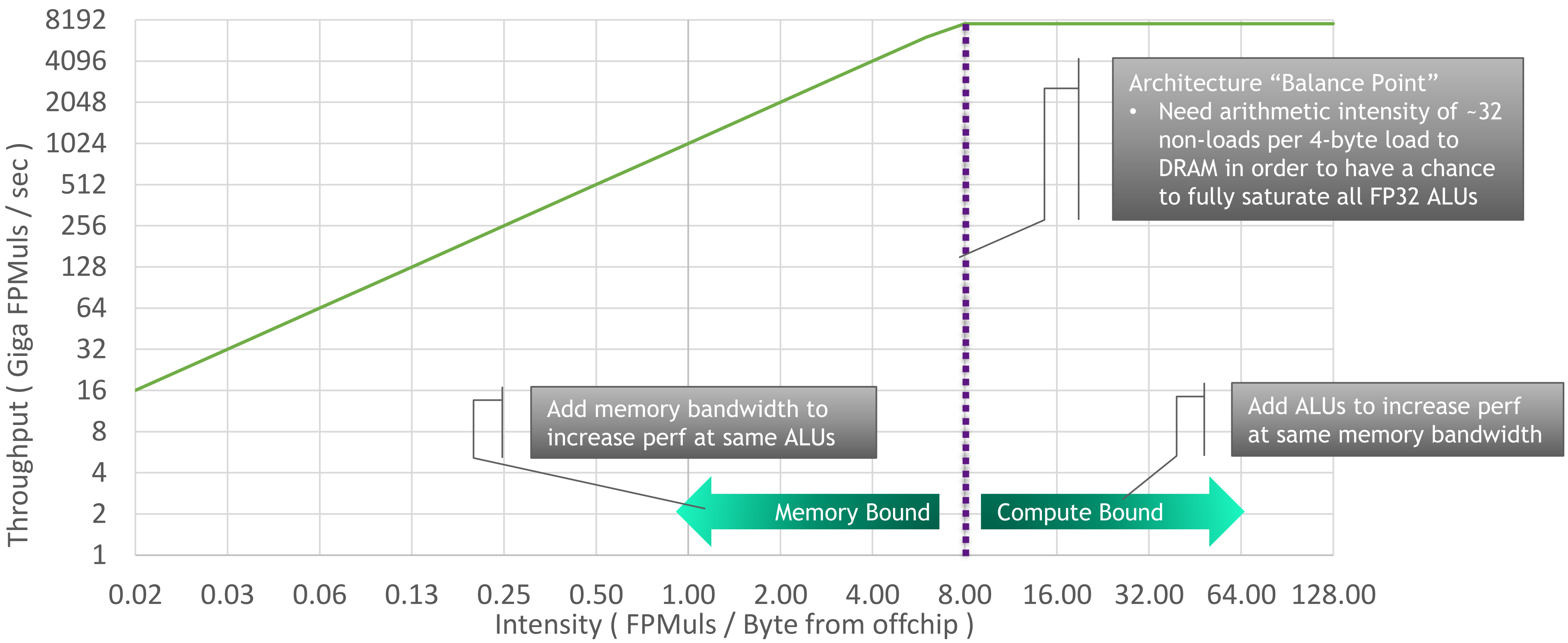
$$\lim_{K,M,N\to\infty} \frac{KMN}{KM + KN + MN} = \infty$$

scaling

Williams et al., [Roofline: An Insightful Visual Performance Model for Floating-Point Programs and Multicore Architectures]

ROOFLINE ANALYSIS

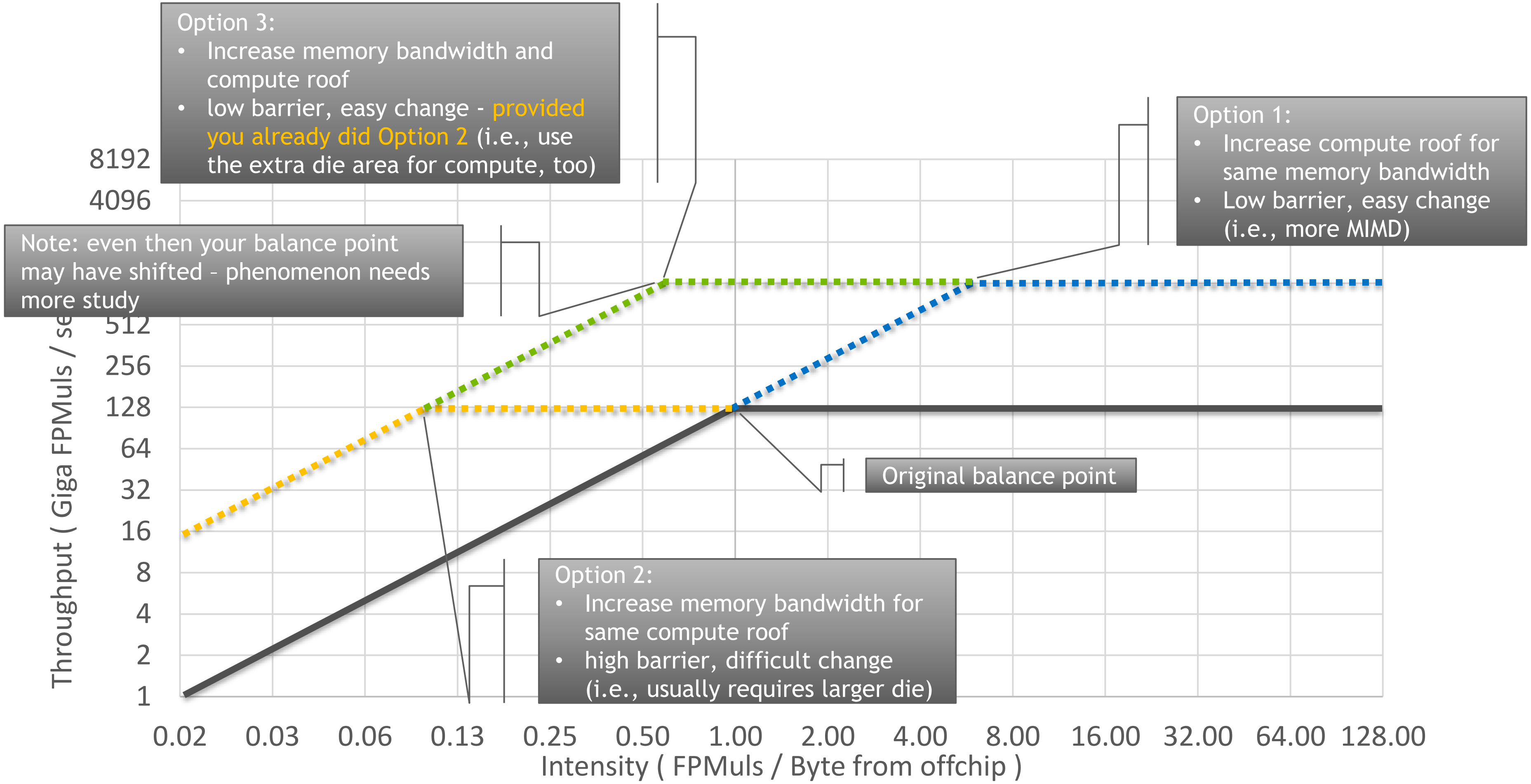
Plots the ratio between compute bandwidth and memory bandwidth



Job of the computer architect: balance hardware provisioning to achieve best result for all anticipated workloads

BALANCING AN ACCELERATOR

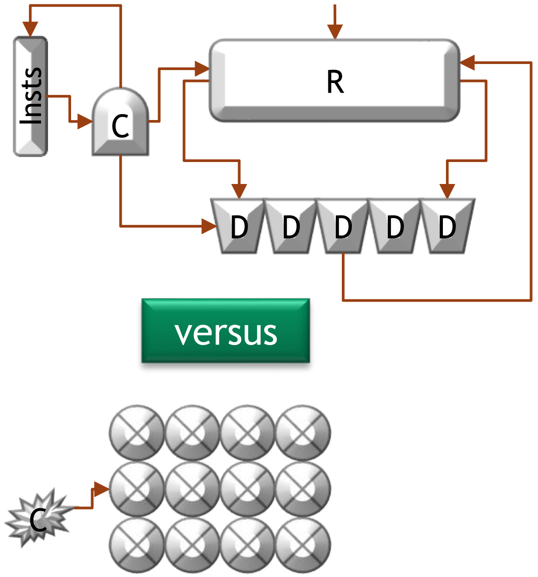
Insight: not all axes require equal investment



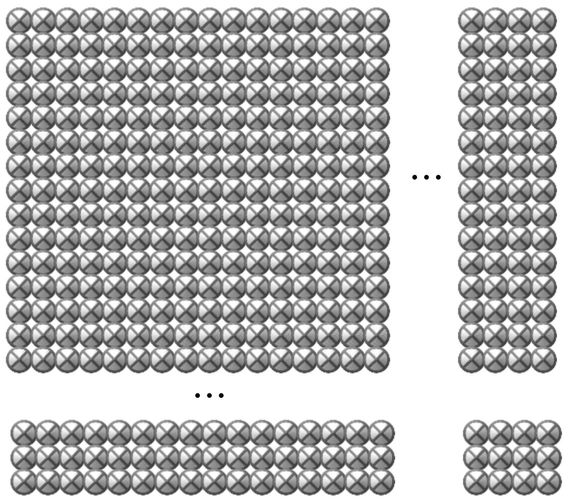
THE DATAPATH TRAP

It is possible (easy?) on modern dies to over-provision compute roof

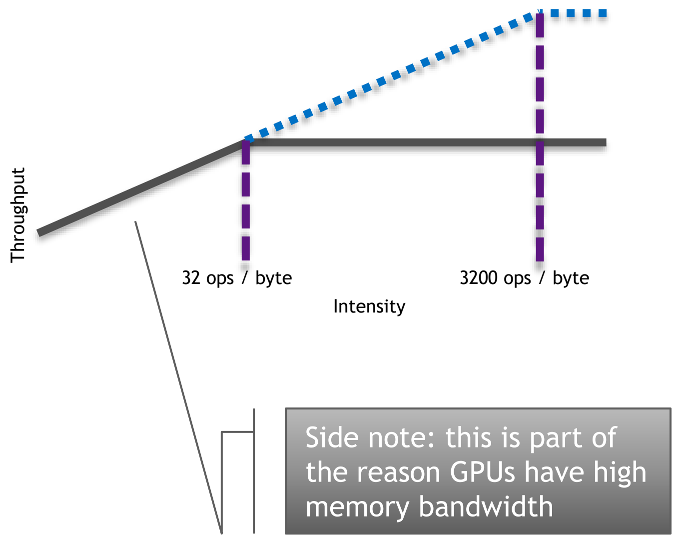
1. Specialized Datapaths take less area



2. Copy-and-paste D Unit can go farther before filling die



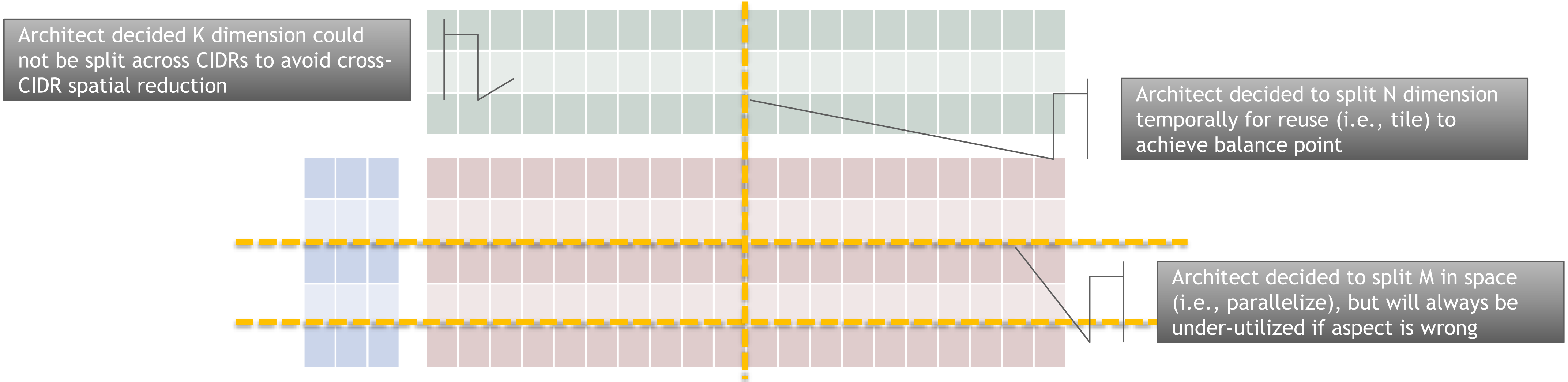
3. Compute roof could end up "un-saturatable" given memory bandwidth



Side note: this is part of the reason GPUs have high memory bandwidth

SCALING REUSE AND ASPECT RATIOS

Beware static decisions to parallelize/tile across certain dimensions



Useful to use *flexibility* to refer to set of ways a platform can parallelize/schedule a given workload

- In contrast to *programmability*, which is the total number of workloads it can run
- See Kao et al. [A Formalism for DNN Accelerator Flexibility], SIGMETRICS
- Note: in general, programmable architectures are also flexible

COROLLARY: AVAILABLE DIE AREA

If you shouldn’t over-invest in compute roof, then what?

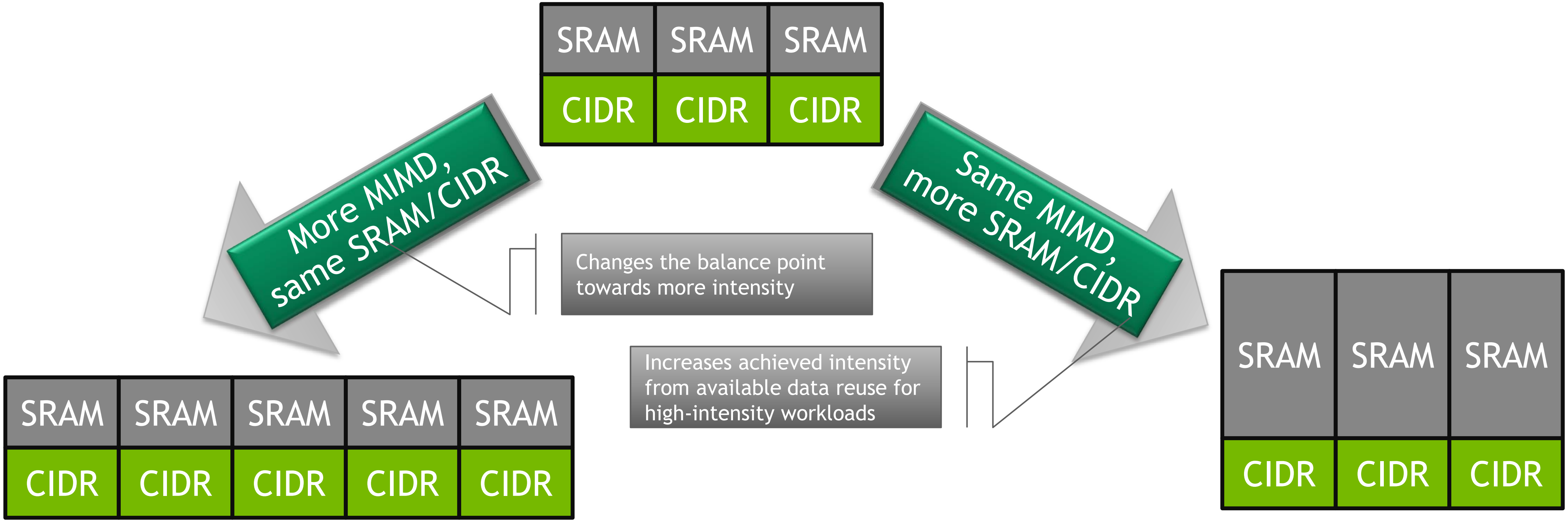
Key Research Hypothesis:

- For the highest-value workloads known today (i.e., ML, real-time graphics, HPC, genomics), reaching the balanced compute:memory bandwidth spot leaves plenty of die area free after provisioning sufficient datapaths

Question: What about SRAM buffers?

ONCHIP SRAM BUFFERS: NOT A PANACEA

Caches exploit the same phenomenon as MIMD (i.e, reuse)



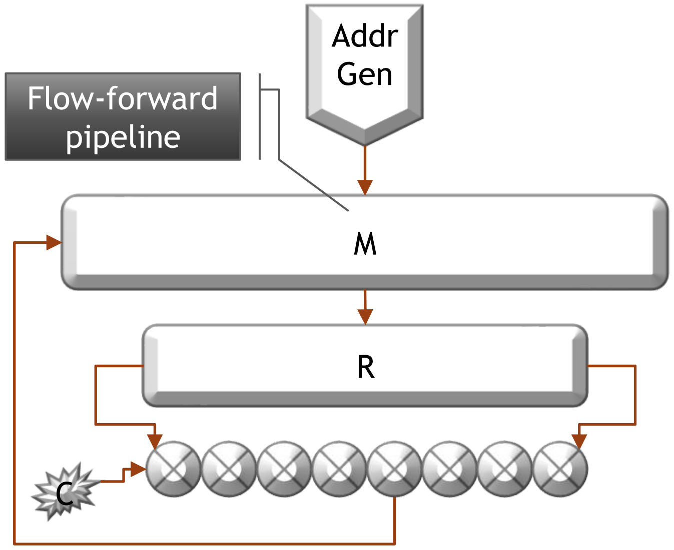
Once temporal reuse buffers reach sufficient capacity to allow saturation of compute roof, returns for more entries diminish significantly

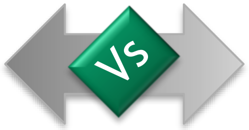
CIDR AND MEMORY LATENCY

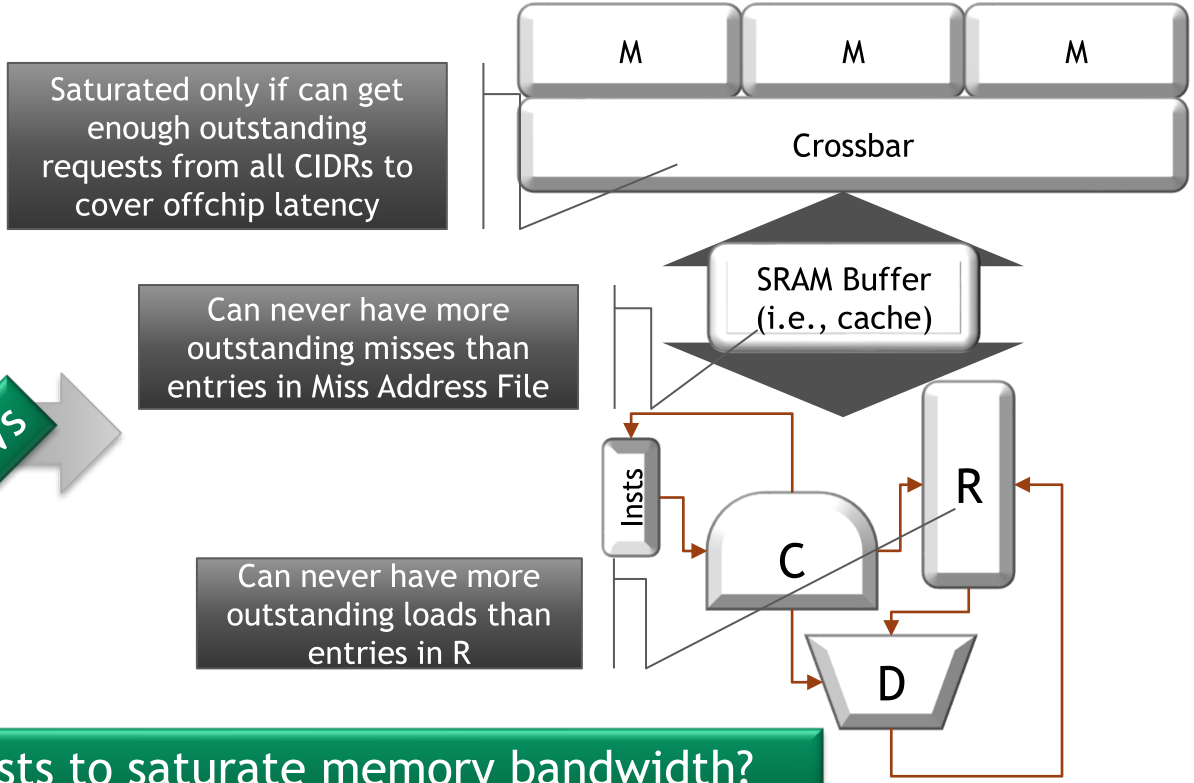
Loads are round-trip "pulls" instead of "pushes"

Decoupled Data Orchestration:

See: Pellauer et al., [Buffets] 2017

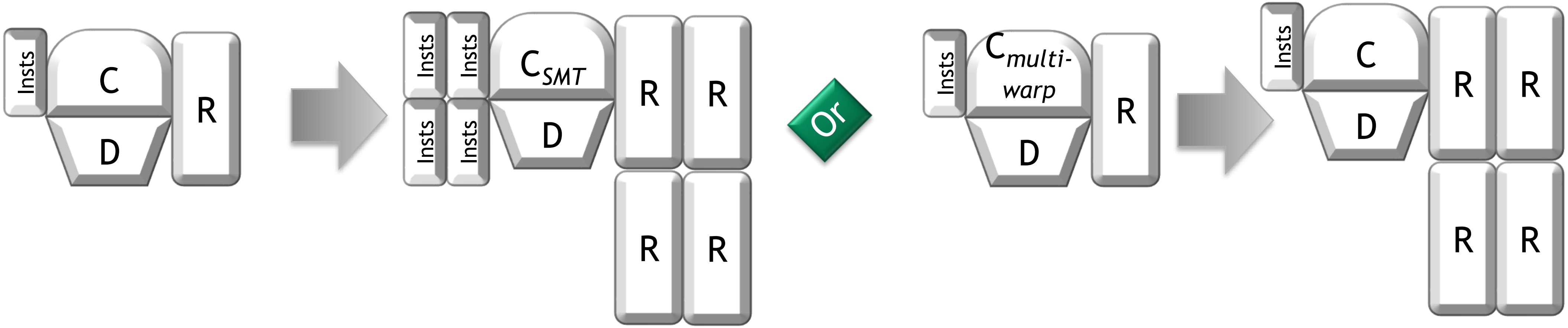






How do you see enough requests to saturate memory bandwidth?

APPROACH: DIFFERENT INST. STREAMS



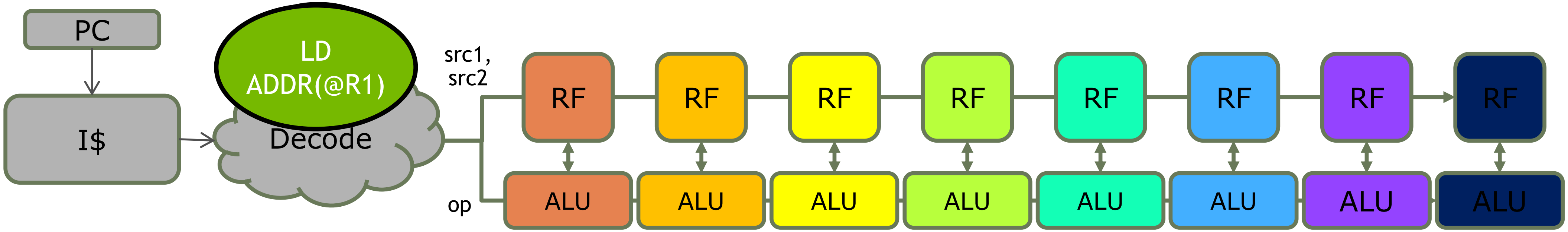
Simultaneous multi-threading (SMT): increase RAMs by N contexts (not logic)

Multiple “warps” : each context is running a different copy of same program

No branch predictor needed, also actually hides latency of D, can still be super-scalar if you want

SIMT-STYLE PROGRAMMING MODEL

In general, SIMT "feels" like individual threads on separate cores



- SIMD Load
  - `LD R0 := ADDR + THR.ID * SZ`
- SIMD Conditional
  - `MASK[THR.ID] = TEST R0`
  - `ADD R0 R1 R2 if MASK[THR.ID]`

- SIMT Load
  - `LD R0 := ADDR(@R1)`
- SIMT Conditional
  - `R1 := Test R0`
  - `CBranch R1 @After`
  - `ADD R0 R1 2`
  - `@After...`

HANDLING CONTROL DIVERGENCE

Add per-warp stack to store PCs and masks of non-taken paths

On a conditional branch:

- Push the current mask onto the stack
- Push the mask and PC for the non-taken path
- Set the mask for the taken path

At the end of the taken path:

- Pop mask and PC for the non-taken path and execute

At the end of the non-taken path:

- Pop the original mask before the branch instruction

If a mask is all zeros or all ones, skip the block

- No performance penalty if all threads make the same decision!

EXAMPLE: CONTROL DIVERGENCE

Assume 4 threads/warp,
initial mask 1111

```
if (m[i] != 0) {            1
  if (a[i] > b[i]) {        2
    y[i] = a[i] - b[i];
  } else {                  3
    y[i] = b[i] - a[i];
  }                         4
} else {
  y[i] = 0;                 5
}
```

1 Push mask 1111
Push mask 0011
Set mask 1100

2 Push mask 1100
Push mask 0100
Set mask 1000

3 Pop mask 0100

4 Pop mask 0011

5 Pop mask 1111

Also: Explicit sync instructions to force convergence when it matters

HANDLING MEMORY ACCESS DIVERGENCE

Each thread in a warp may load or store a completely different memory address (gather/scatter)

Address coalescing unit detects sequential and strided patterns, coalesces memory requests, but complex patterns can result in multiple lower bandwidth requests (memory divergence)

Writing efficient GPU code requires most accesses to not conflict, even though programming model allows arbitrary patterns!

PROS AND CONS OF SIMT

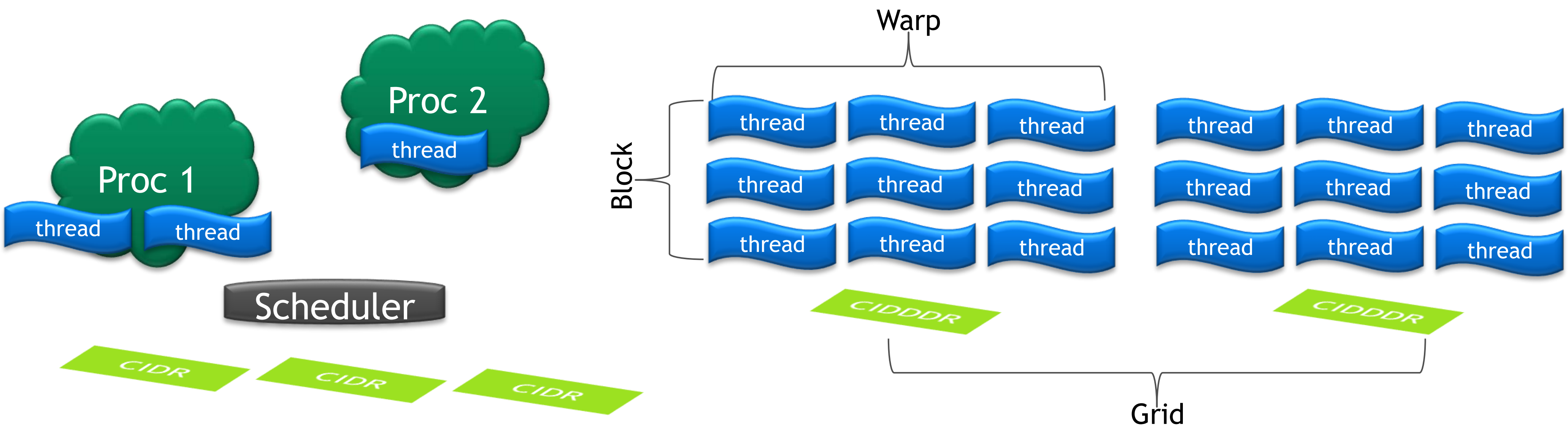
Pros:

- Allows programmers to view architecture as *N* x CIDR instead of CIDDDDD…DR
- While getting amortization benefits

Cons:

- Marginal area and engineering cost to support
  - C unit is more complex, but still less so than out-of-order/speculative
- Non-experts can be surprised by performance drop-offs from divergence

SCALING PROGRAMS ACROSS CIDR UNITS



- Flat Pool Approach:
  - Bunch of processes, each process many have N threads
  - No notion of which CIDR runs what [until you need it]
- Explicit Hierarchy Approach:
  - Warp -> Block -> Grid -> Stream
  - Can use special features of that level
  - E.g., inter-warp needs less sync., can use "Shared Memory" scratchpad within Block, etc.

PUTTING IT ALL TOGETHER

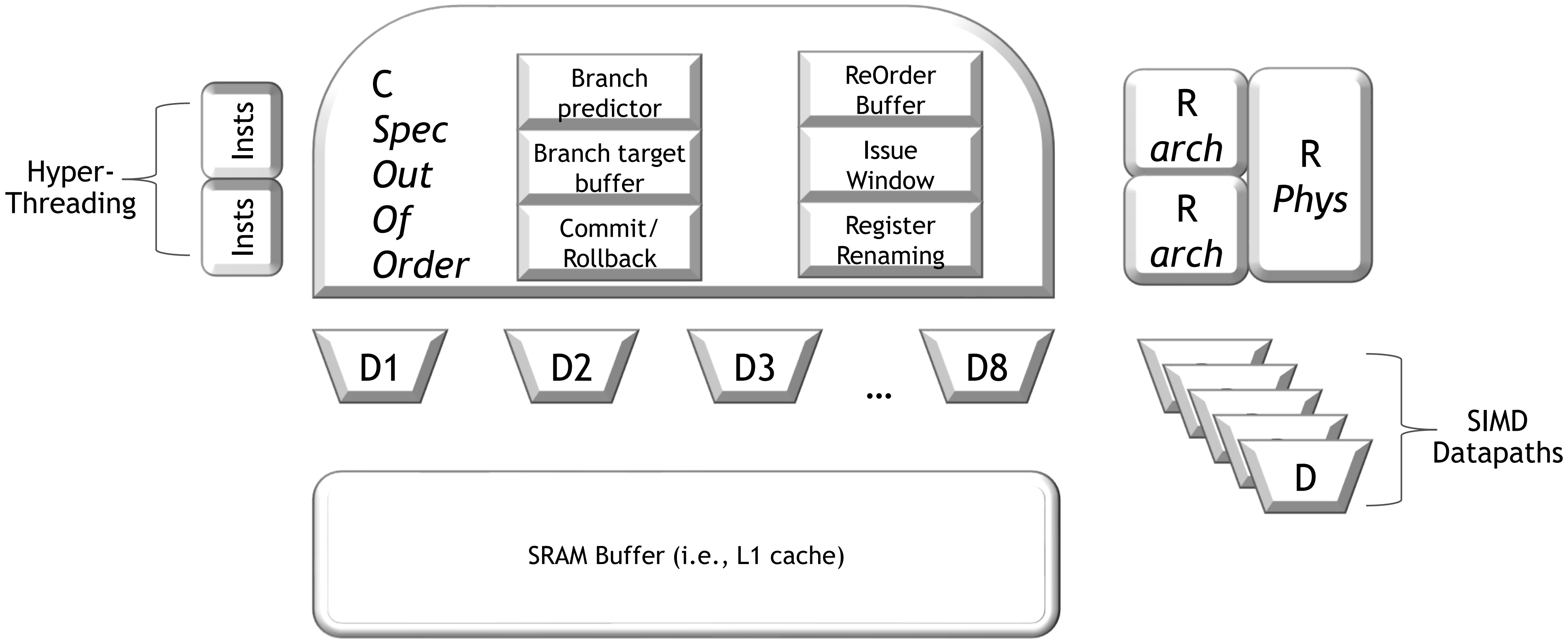
To provision a complete chip you need (at least):

- Physical constraints: package size/die(s) size/memory bandwidth/TDP
- Datapath strategy: Fixed-function or programmable
- Control + Instruction Mem + Datapath + RegFile (CIDR) organization (superscalar, SIMD, etc)
- Strategy for exposing enough memory requests to saturate bandwidth
- Time-multiplexing strategy for sharing CIDR units
- MIMD organization across CIDR units, and strategy for filling them with work
- Peak possible performance: total FP Datapath resources across all of chip
- Target balance point between bandwidth and reuse that allows practical saturation

CPU PROVISIONING

Control Unit	Speculative + Out-of-Order, complex branch predictors
Datapath Unit	O(8) Super-scalar w/ supplemental SIMD
Register File	Small architectural, larger renamed Phys Reg File
CIDR Time-multiplexing	O(2-4) simultaneous multi-threading, called “hyper-threading”
CIDR:Cache ratio	More area provisioned for cache
MIMD per die	O(32), process/thread pool
Peak D perf per die	O(32 * 8) = O(256) multiplies per cycle [NOTE: this excludes SIMD, which can be hard to power simultaneously across multiple cores in practice]
Real-world CIDR name	“Core”

CPU CORE VISUALIZED USING CIDR



CPU-STYLE PROS AND CONS

Pros:

- Programs with low parallelism can achieve full utilization
- Doesn't need much memory bandwidth
- Can focus instead on minimizing latency (e.g., low clock cycle times, fast $ hit times)
- Backwards compatible with historical programs, and can even sometimes provide speedups without rewriting/recompiling

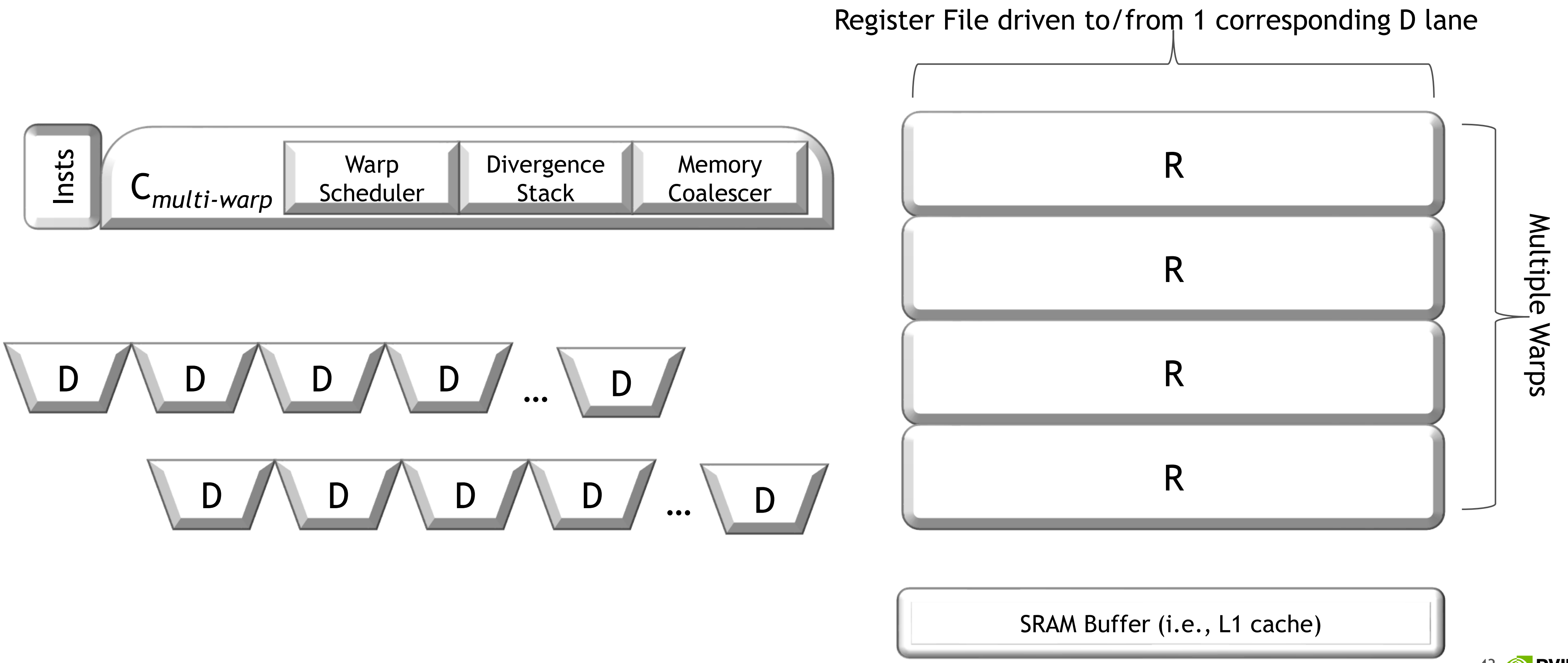
Cons:

- Peak performance is low for high-intensity, high-parallel workloads
- Insight: this is an extremely valuable market (Graphics, Deep Learning, HPC, etc.)

GPU PROVISIONING

Control Unit	In-Order, no branch predictors or speculation
Datapath Unit	O(2) Super-scalar w/ SIMT 32
Register File	Large
CIDR Time-multiplexing	O(32-64) multiple warps (depending on how many registers per warp)
CIDR:Cache ratio	More area provisioned for CIDR, small caches
MIMD per die	O(100), warp/block*/grid/stream** organization * Block is more for latency hiding than peak datapaths ** Stream is for dependencies between kernels
Peak D perf per die	O(100 x (32x2)) = O(6400) multiplies per cycle [SIMT only: Excludes Tensor Cores]
Real-world CIDR name	“SM” or “Streaming Multi-processor”

GPU SM VISUALIZED AS CIDR



GPU-STYLE PROS AND CONS

Pros:

- High peak performance since so many D units
- Doesn't need to minimize latencies, can focus on maximizing throughput (esp. memory bandwidth)

Cons:

- Doesn't work with historical programs
- Low-intensity, low-parallel workloads will result in under-utilization
  - Not just because of warp size or number of SMs, but also because minimum # of warps per SM needed to hide memory and datapath latency

BRINGING HISTORY BACK IN

Sadly, “CIDR” is not a historical term

Notion of a Turing-Complete C unit was fundamental for CPUs from the beginning

- GPUs started more fixed function but moved programmable because of strong desire for field programmability (graphics is a fast-moving field)

Costs of C unit were amortized down successfully in GPUs

- Value of programmability outweighed area/engineering overheads in the market (real cost was giant investment in compilation and debugging infrastructure, called “CUDA everywhere”)

Unlike Vector-Vector multiplication, real Graphics and Deep Learning workloads have high intensity and parallelism

- Large MIMD x SIMT provisioning of datapaths optimized for peak performance, balanced against given memory bandwidth
- Note: ray-tracing is a bit different, in an interesting way…

BRINGING HISTORY BACK IN (PART 2)

See also: Myer and Sutherland's "Wheel of Reincarnation" [On the Design of Display Processors, 1968]

Programmers will rewrite workloads that are valuable enough if benefit is large enough

- Fixed function -> Programmable: Rare event or fundamental force?

Pivotal moment: Emergence of Deep Learning black swan enabled by GPUs

- Another high-parallel + high-intensity valuable workload
- Economics played a role: widely available substrate with appropriate provisioning

[Personal opinion] Much of the time, you will find that running high-intensity, high-parallel workloads that have "wasted" operations will be faster in practice than focusing on optimizing instruction stream latency, as your optimizations often lower arithmetic intensity

REVISITING AVAILABLE DIE AREA

What is the realistic compute:memory bandwidth ratio in practice?

Key Research Hypothesis:

- For the highest-value workloads known today (i.e., ML, real-time graphics, HPC, genomics), reaching the balanced compute:memory bandwidth spot leaves plenty of die area free after provisioning sufficient datapaths and properly amortized control and sufficient onchip SRAM and mechanisms for exposing enough memory requests

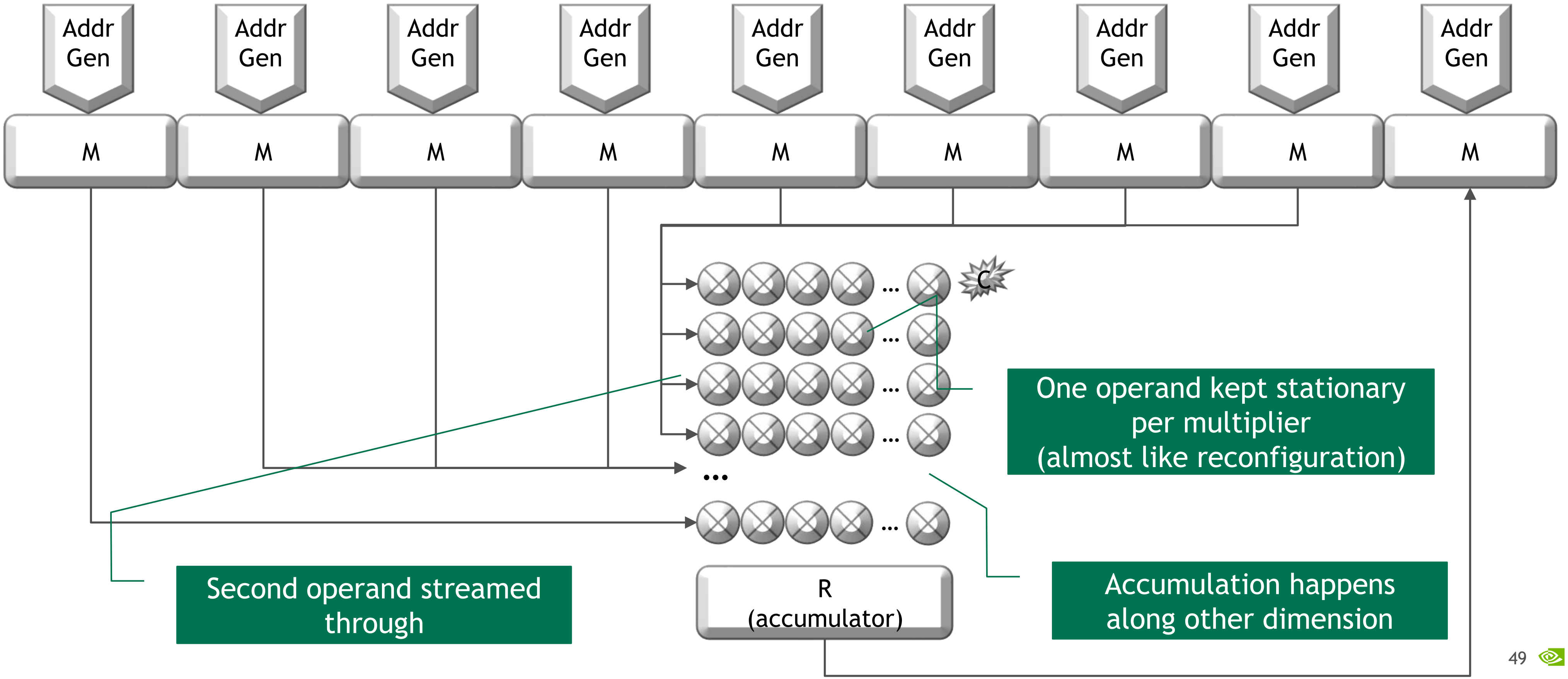
Question: What to fill remaining die area with?

- A) Programmability features
- B) Stay fixed function, but get more datapaths

TPU V3 MATRIX UNIT PROVISIONING

Control Unit	? [Not many public details on this]
Datapath Unit	128*128=16K Fixed multipliers per cluster
Register File	1 register per multiplier for weight, 1 for passing sum
CIDR Time-multiplexing	None (weight-stationary dataflow inside each GEMM)
CIDR:Cache ratio	Extremely high [Public materials focus on on-package memory instead of on-chip]
MIMD per die	O(2) clusters
Peak D perf per die	O(256K) [Note: High AI needed relative to memory bandwidth to maintain peak?]
Real-world CIDR name	Matrix Unit (MXU)

TPU MATRIX UNIT VISUALIZED AS CIDR



TENSOR PROCESSOR PROS AND CONS

Pros:

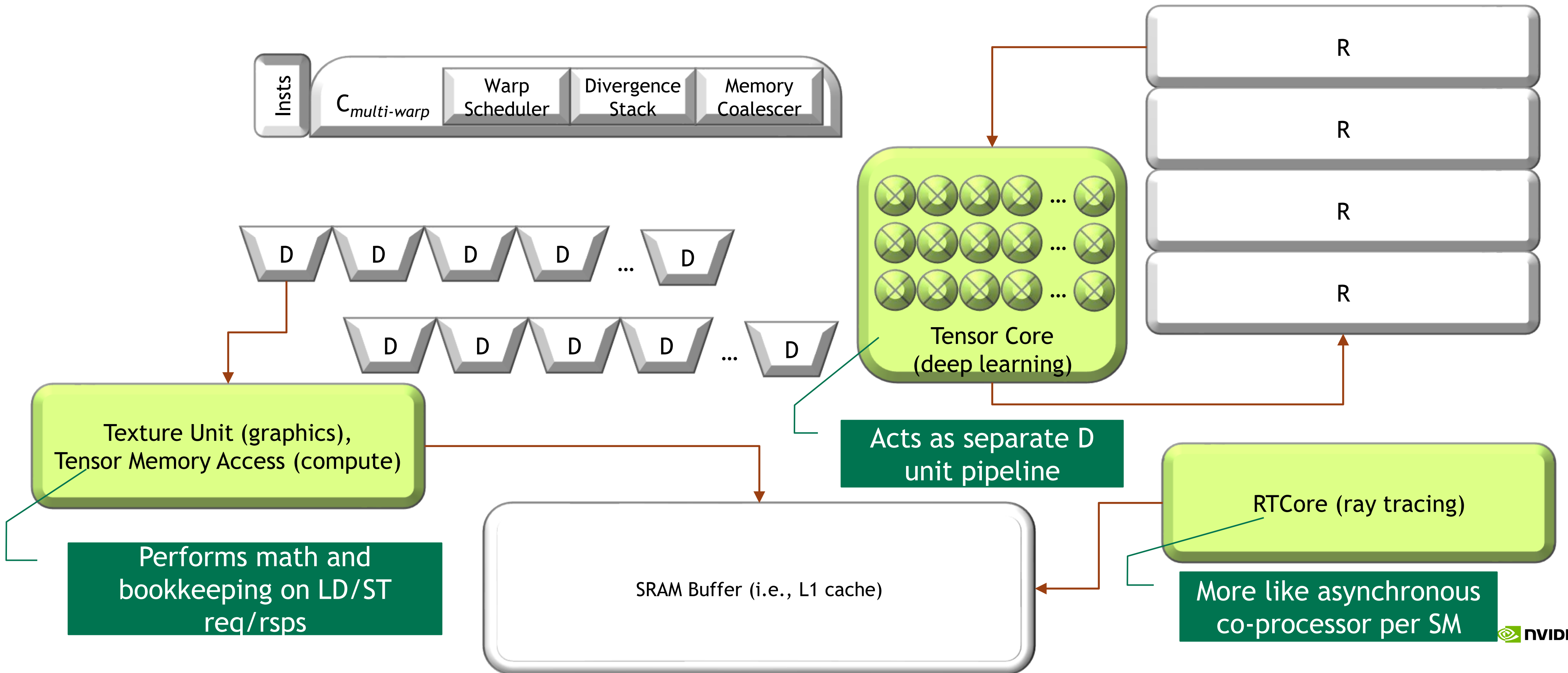
- Theoretical peak datapaths per chip is very high [intensity balance point is too!]

Cons:

- Removes programmability/black swans in extremely fast-moving field
- Trying to further amortize areas GPU has already amortized (i.e., C, I, R)
  - Nota Bene: Difference between 1/32 and 1/16000 is smaller in absolute terms than the difference between 1/1 and 1/32
- Doesn't do much to help with offchip DRAM
  - One change in the field using programmability that reduces DRAM traffic could immediately outweigh all benefits achieved through design-time specialization

CONTRAST: SPECIALIZATION IN GPUS

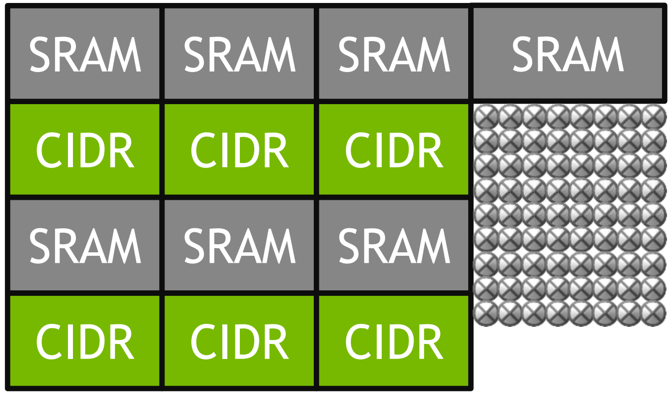
Supplement programmable CIDR for key workloads

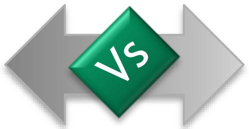


EMERGING SPECIALIZATION ORGANIZATION

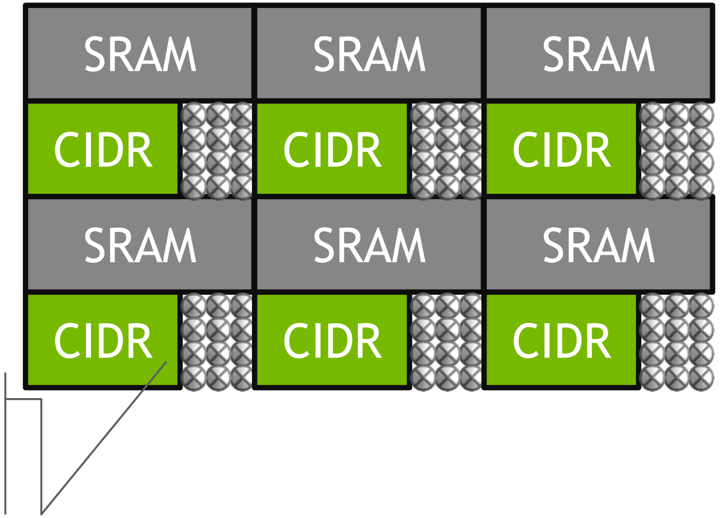
General principle: Disperse specialized blocks in programmable host

Disjoint:





Dispersed:



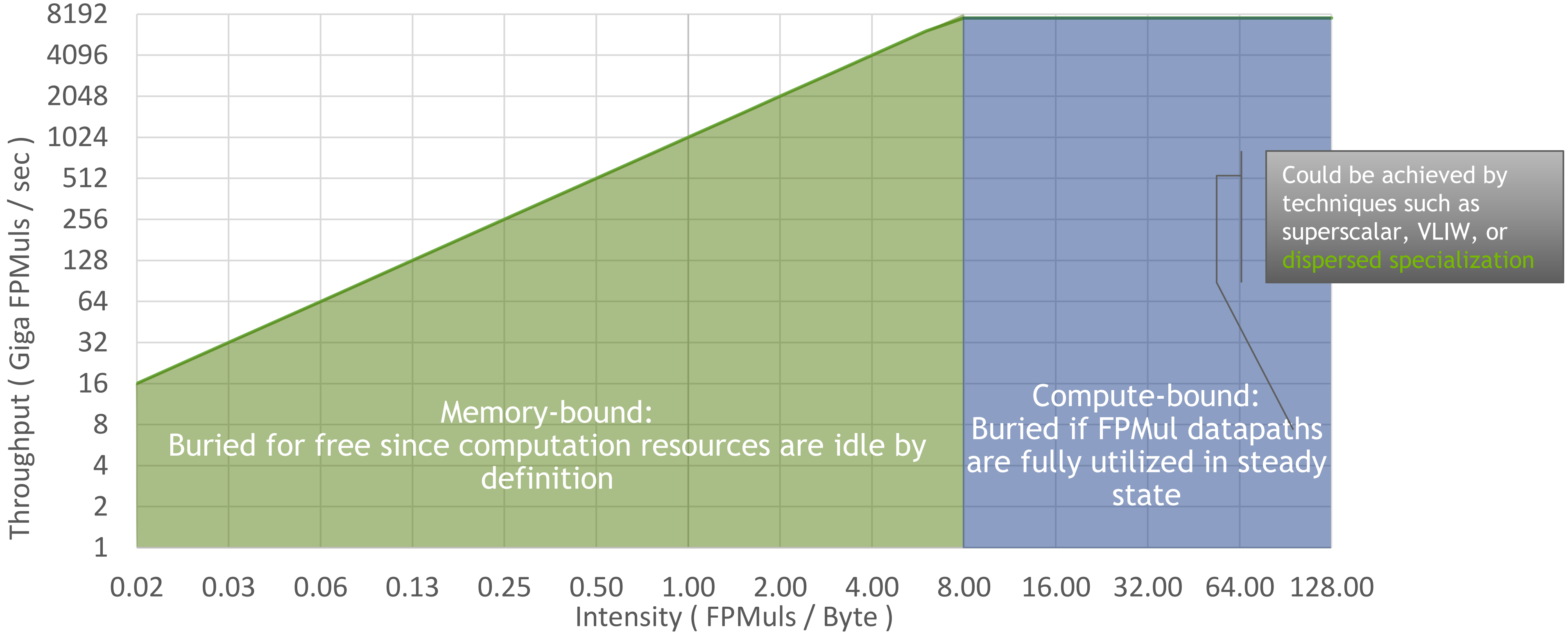
Benefits:

- Lower offload latency
- Share memory hierarchy
- Programming model?

Insight: dispersion amortizes cost while avoiding diminishing returns from over-amortization

"BURYING" PROGRAMMABILITY OVERHEAD

What do we need to achieve performance parity in the expected case?



PROGRAMMABILITY VALUE PROPOSITION

Potential Outcomes:
- Post field-deployment adaptability if workload changes
- Black swans from users
- More available optimizations (may even offset energy on-chip energy penalty if reduce off-chip traffic)

$$Value_{\{programmability\}} = \frac{Benefit_{\{programmability\}}}{Cost_{\{programmability\}}}$$

Memory-Bound Case:
- No area cost
- Minor on-chip energy penalty (DRAM dominated)
- No performance cost (free D unit cycles)

Compute-Bound Case:
- Minor area cost (available because of M units)
- On-chip energy penalty (quantification needed)
- No performance penalty (overheads buried by well-known techniques)

My take: From a pure computer architecture point-of-view there is no compelling hardware overhead to programmable accelerators (just design effort)

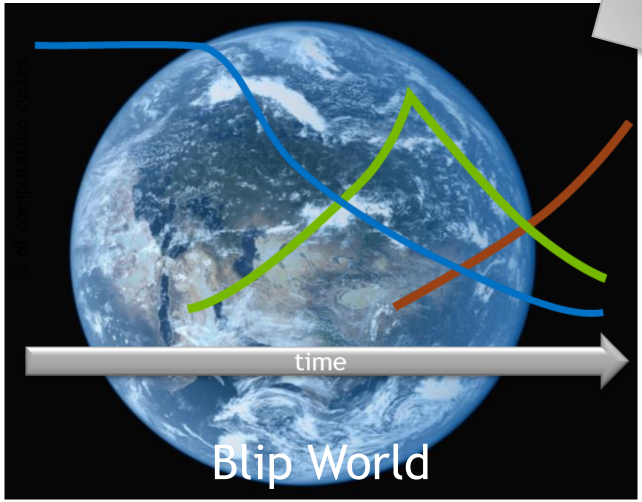
TENTPOLE QUESTION REVISITED

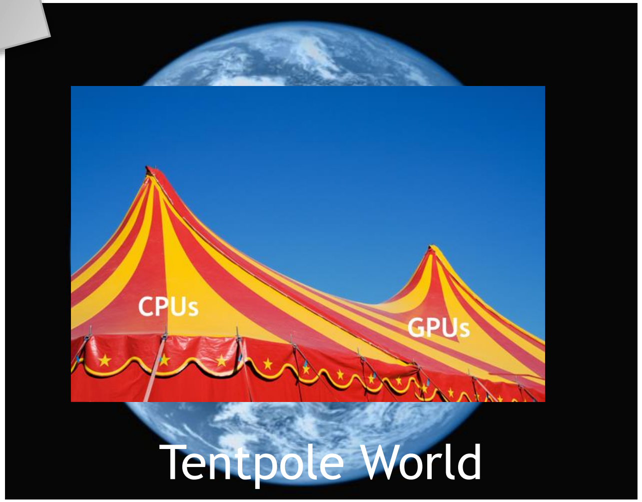
Real cost: software toolchain design, verification, and maintenance needed to effectively program the hardware

Market finds benefits of programmability worth overhead cost software toolchain?

No

Yes





KEY TAKEAWAYS

Benefits of programmability are massive, but difficult to quantify

Because of the need to choose a balanced memory:computation bandwidth, we often end up with more available die area than is intuitive

- Filling up with SRAM actually exploits same reuse principle as more datapaths
- There is no notable area or energy barrier to programmability features
- Real costs of programmability are non-architectural (i.e., programming toolchain)

Specialization is not dead, but needs to evolve to be dispersed throughout programmable CIDRs, instead of disjoint heterogeneous blocks

- Dispersion helps bury programmability performance overheads in compute-bound case, just like classical Instruction-Per-Cycle techniques (i.e., superscalar, VLIW)

mpellauer@nvidia.com

NVIDIA