GPUs, Massive Parallelism, and Compute Abstractions

Andreas Kloeckner

University of Illinois

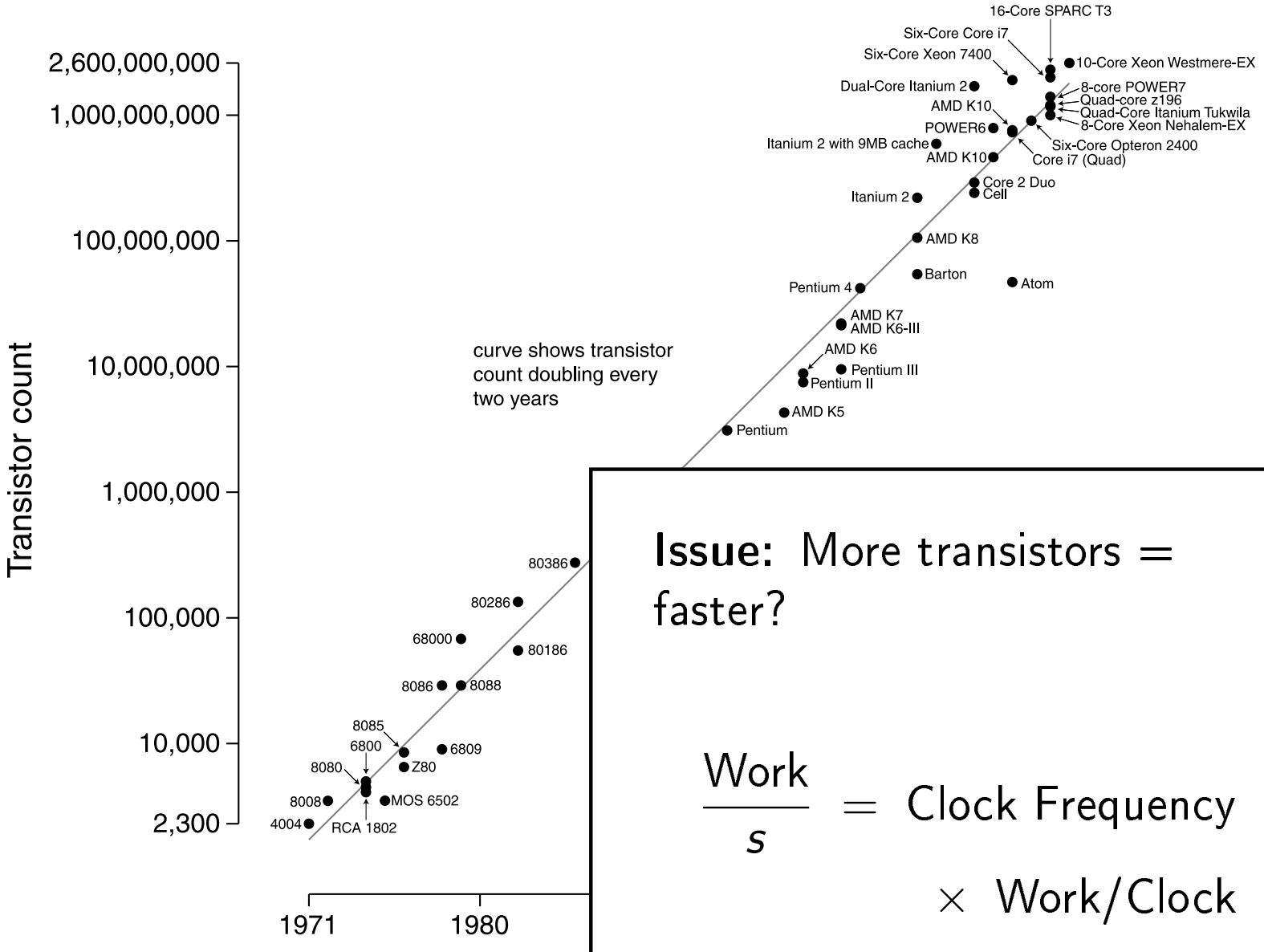
January 20–22, 2020

ILLINOIS

Outline

Outline

Moore's Law



Issue: More transistors = faster?

$$\frac{\text{Work}}{s} = \text{Clock Frequency} \times \text{Work/Clock}$$

Peak Architectural Instructions per Clock: Intel

CPU	IPC	Year
Pentium 1	1.1	1993
Pentium MMX	1.2	1996
Pentium 3	1.9	1999
Pentium 4 (Willamette)	1.5	2003
Pentium 4 (Northwood)	1.6	2003
Pentium 4 (Prescott)	1.8	2003
Pentium 4 (Gallatin)	1.9	20
Pentium D	2	2005
Pentium M	2.5	2003
Core 2	3	2006
Sandy Bridge...	3.5ish	2011

[Charlie Brej http://brej.org/blog/?p=15]

Discuss: How do we get out of this dilemma?

The Performance Dilemma

- IPC: Brick Wall
- Clock Frequency: Brick Wall

Ideas:

- Make one instruction do more copies of the same thing ("SIMD")
- Use copies of the same processor ("SPMD"/"MPMD")

Question: What is the *conceptual* difference between those ideas?

- SIMD executes multiple program instances in lockstep.
- SPMD has no synchronization assumptions.

The Performance Dilemma: Another Look

- Really: A crisis of the 'starts-at-the-top-ends-at-the-bottom' prorgramming model
- Tough luck: Most of our codes are written that way
- Even tougher luck: Everybody on the planet is *trained* to write codes this way

So:

- Need: Different tools/abstractions to write those codes

GPU Programmability

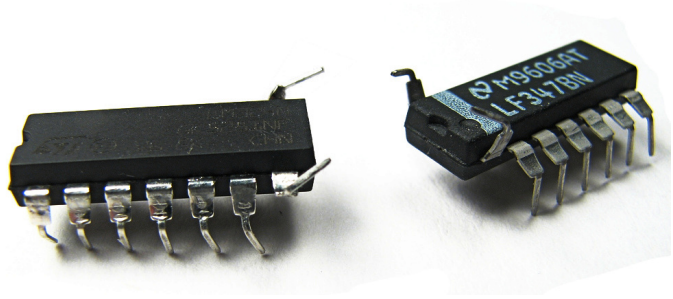
The 'nightmare limit':

- "Infinitely" many cores
- "Infinite" vector width
- "Infinite"memory/comm. latency

Further complications:

- Commodity chips
  - Compute only one design driver of many
- Bandwidth only achievable by *homogeneity*
- Compute bandwidth $\gg$ Memory bandwidth

$\rightarrow$ Programmability is key.



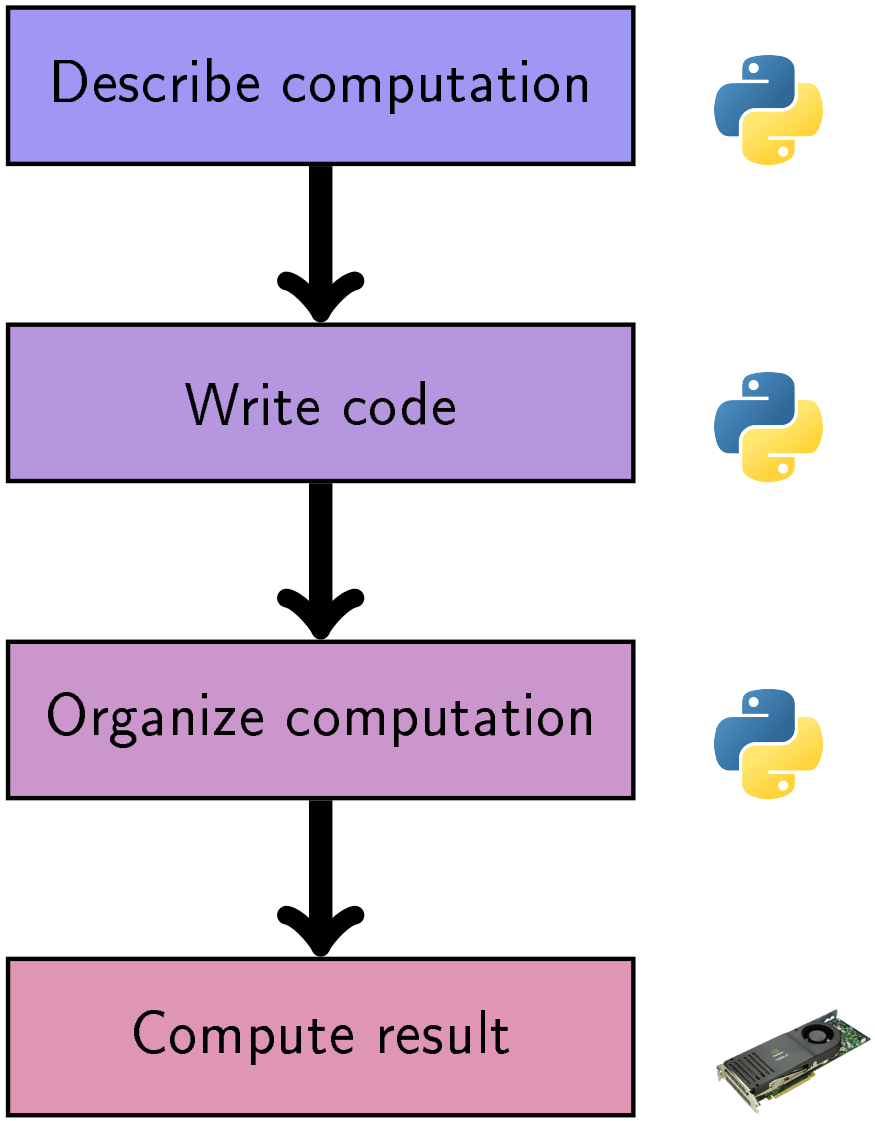
[flickr.com/oskay ©]

Why Python for HPC

- Mature, large and active community
- Emphasizes readability
- Written in widely-portable C
  - Easy coupling to C/C++ (pybind11) / Fortran (f2py)
- A 'multi-paradigm' language
- Rich ecosystem of sci-comp related software
- Great as a 'glue language'

[Python logo: python.org]

Addressing the Elephant in the Room: Slowness



Python + GPUs

- GPUs are everything that scripting languages are not.
  - Highly parallel
  - Very architecture-sensitive
  - Built for maximum FP/memory throughput

  $\rightarrow$ complement each other
- CPU: largely restricted to control tasks $\sim$1000/sec
  - Scripting fast enough
- Python + OpenCL = **PyOpenCL**
- Python + CUDA = **PyCUDA**

[GPU: Nvidia Corp.]

Outline

What is OpenCL?

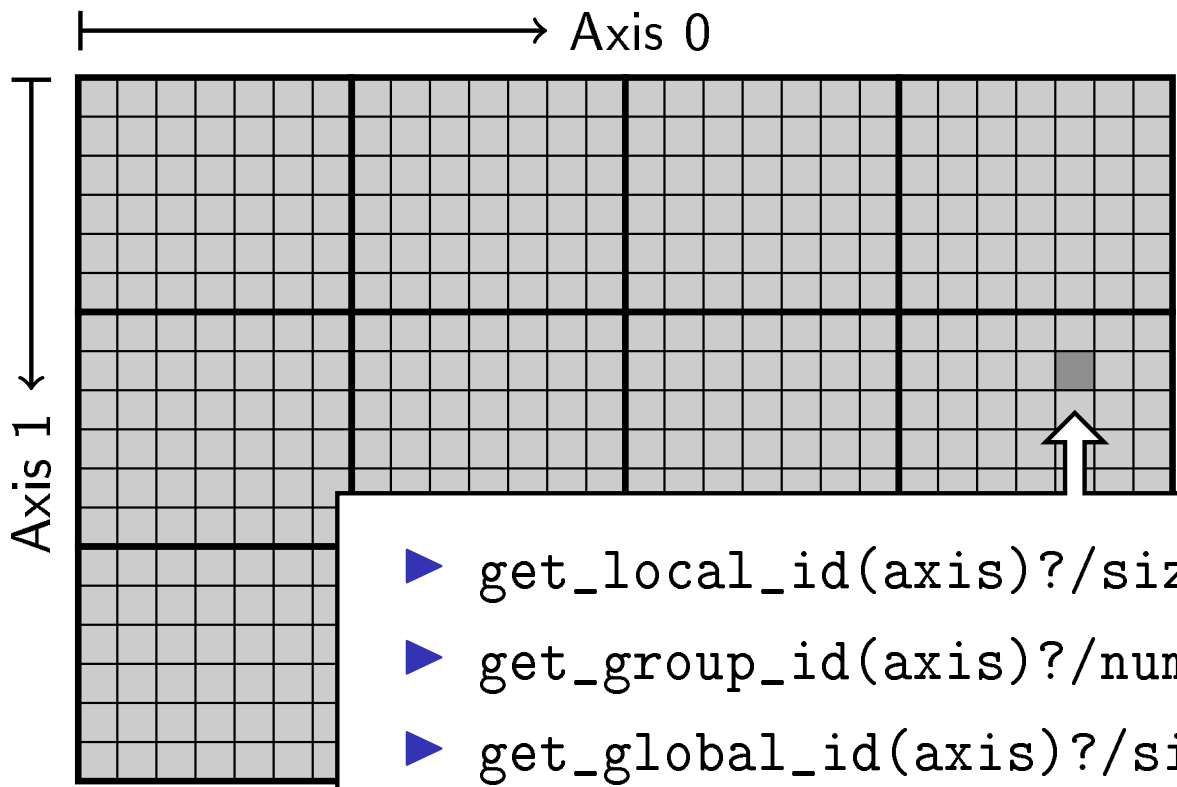
OpenCL (Open Computing Language) is an open, royalty-free standard for general purpose parallel programming across CPUs, GPUs and other processors. [OpenCL 1.1 spec]

- Device-neutral (Nv GPU, AMD GPU, Intel/AMD CPU)
- Vendor-neutral
- JIT built into the standard

Defines:

- Host-side programming interface (library)
- Device-side programming language (!)

Wrangling the Grid



- `get_local_id(axis)?/size(axis)?`
- `get_group_id(axis)?/num_groups(axis)?`
- `get_global_id(axis)?/size(axis)?`

`axis=0,1,2,...`

Machine Abstractions

Is OpenCL only for GPUs?

No. Implementations for CPUs exist.

How does OpenCL map onto CPUs?

- Two levels of concurrency, one for cores, one for vector lanes
- Use the same mapping idea for CPUs
- Realize that you're not programming the hardware: you're programming an abstract model of the hardware.

What is essential about programming in OpenCL, what is arbitrary?

- Essential: the semantics of the programming model (what does the program mean?)
- Arbitrary: the spelling of the programm

Demo

[DEMO: intro-01-hello-pyopencl]

To follow along: http://bit.ly/geilogpu20

Programming Approaches

Decisions that determine your approach to throughput computing:

- AOT vs JIT
- Meta vs not
- In-language vs Hybrid

Outline

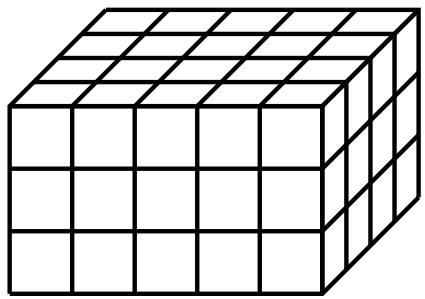
Why Arrays?

- Parallelism: best if applied in large quantities
- Arrays: The natural data structure to support large-scale concurrency
- Structured
  - Unstructured workloads: See Johannes's talks
- $O(1)$ element access
- Static (i.e. known-from-the-outset) control flow for traversal

Arrays in Numpy

Core attributes of an array:

- Shape
- `dtype` (data type)
- Strides
- Pointer
- (Lifetime relationship)



Demo: Host Arrays

[DEMO: arrays-01-numpy]

Device Arrays

Want: An array object that works just like `numpy` arrays, but on the GPU

Issues:

- Which command queue? (Which context?)
- Synchronization?
- When to generate code? For which data types?

Demo: Device Arrays

[DEMO: arrays-02-pyopencl]

Outline

Map

$$y_i = f_i(x_i)$$

where $i \in \{1, \ldots, N\}$.

Notation:

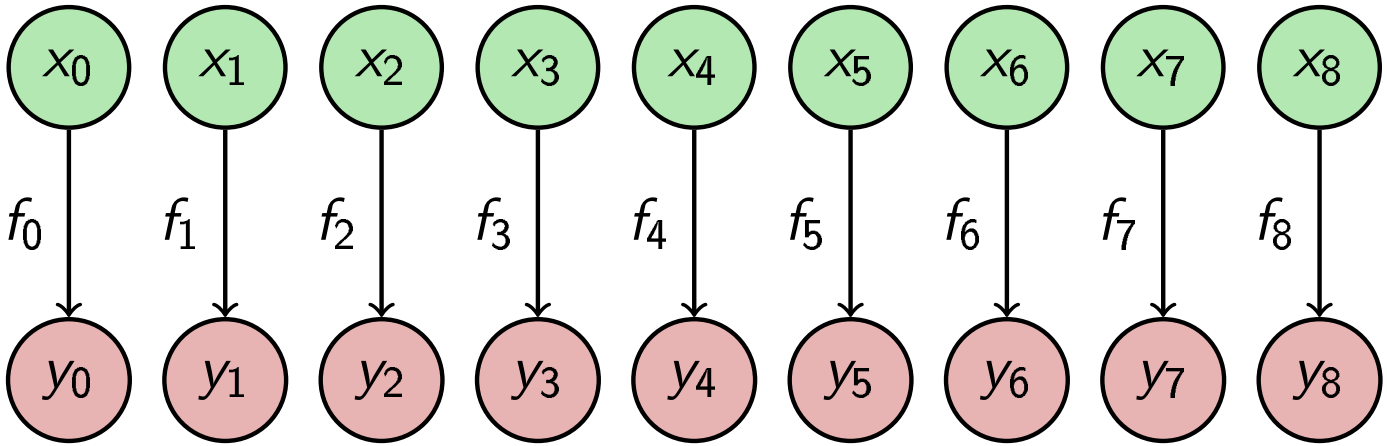
- x_i: inputs
- y_i: outputs
- f_i: (pure) functions (i.e. *no side effects*)

When does a function have a "side effect"?

In addition to producing a value, it

- modifies non-local state, or
- has an observable interaction with the outside world.

Map: Graph Representation



Embarrassingly Parallel: Examples

Surprisingly useful:

- Element-wise linear algebra:
  Addition, scalar multiplication (*not* inner product)
- Image Processing: Shift, rotate, clip, scale, ...
- Monte Carlo simulation
- (Brute-force) Optimization
- Random Number Generation
- Encryption, Compression
  (after blocking)

Demo

[DEMO: patterns-01-elementwise]

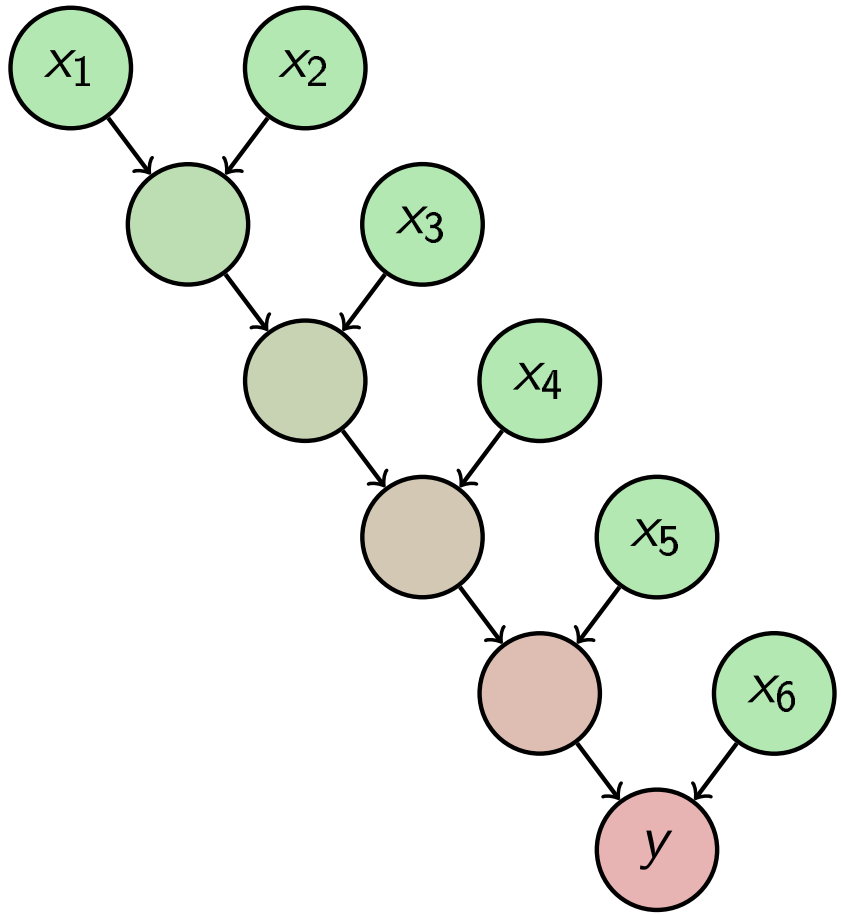
Reduction

$$y = f(\cdots f(f(x_1, x_2), x_3), \ldots, x_N)$$

where N is the input size.
Also known as

- Lisp/Python function `reduce` (Scheme: `fold`)
- C++ STL `std::accumulate`

Reduction: Graph



Approach to Reduction

Can we do better?

"Tree" very imbalanced. What property of f would allow 'rebalancing'?

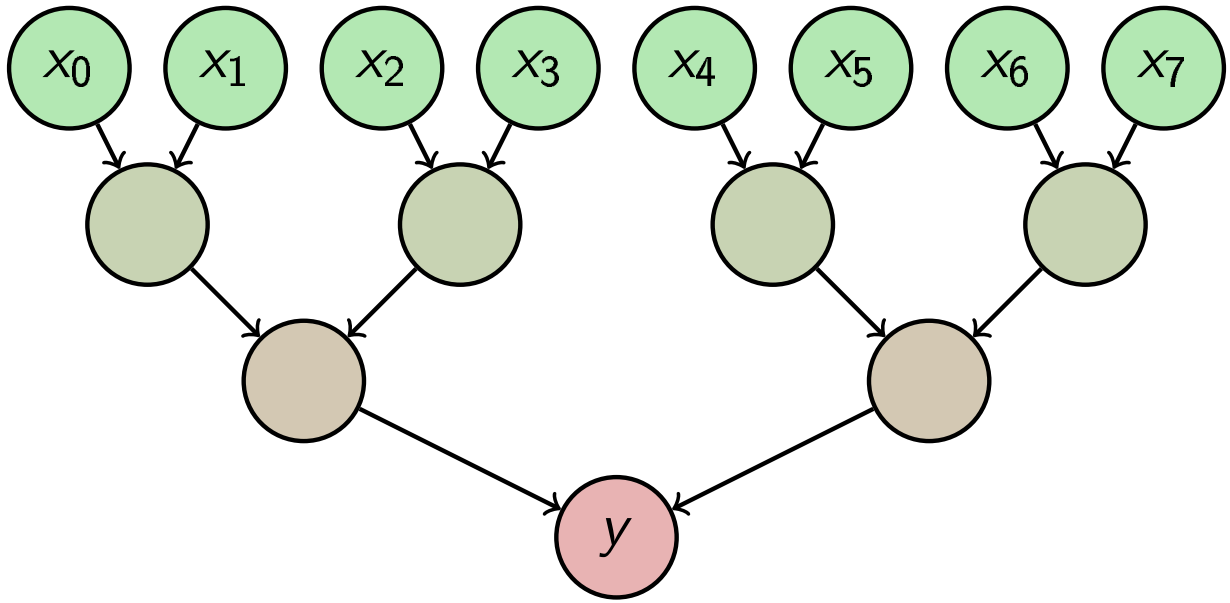
$$f(f(x,y),z) = f(x,f(y,z))$$

Looks less improbable if we let $x \circ y = f(x,y)$:

$$x \circ (y \circ z)) = (x \circ y) \circ z$$

Has a very familiar name: *Associativity*

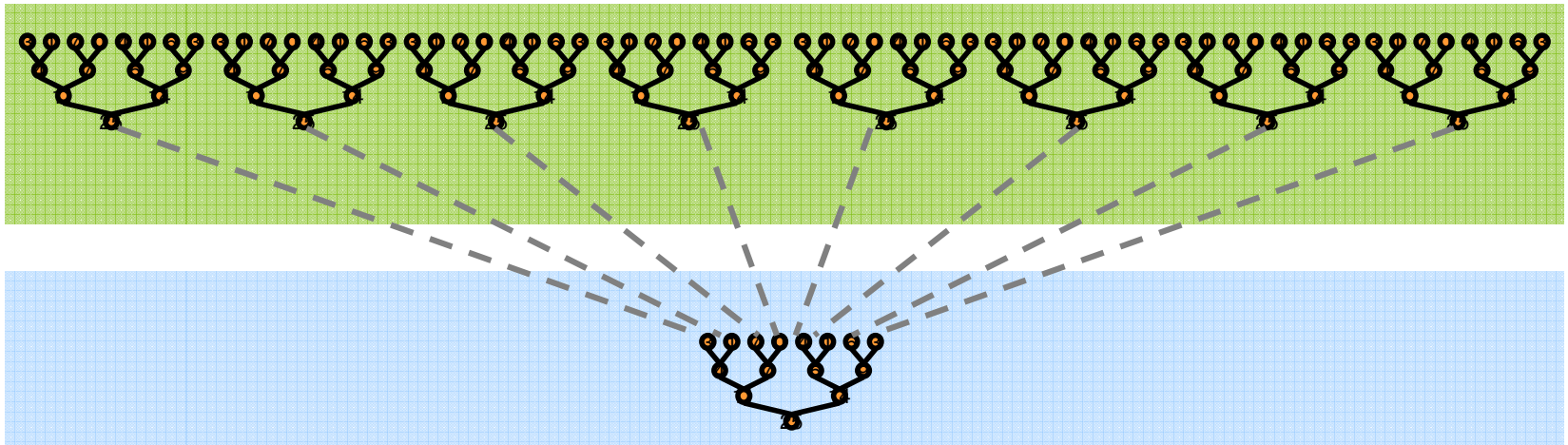
Reduction: A Better Graph



Processor allocation?

Mapping Reduction to SIMD/GPU

- Obvious: Want to use tree-based approach.
- Problem: Two scales, Work group and Grid
  - to occupy both to make good use of the machine.
- In particular, need synchronization after each tree stage.
- Solution: Use a two-scale algorithm.



In particular: Use multiple grid invocations to achieve inter-workgroup synchronization.

Demo

[DEMO: patterns-02-reduction]

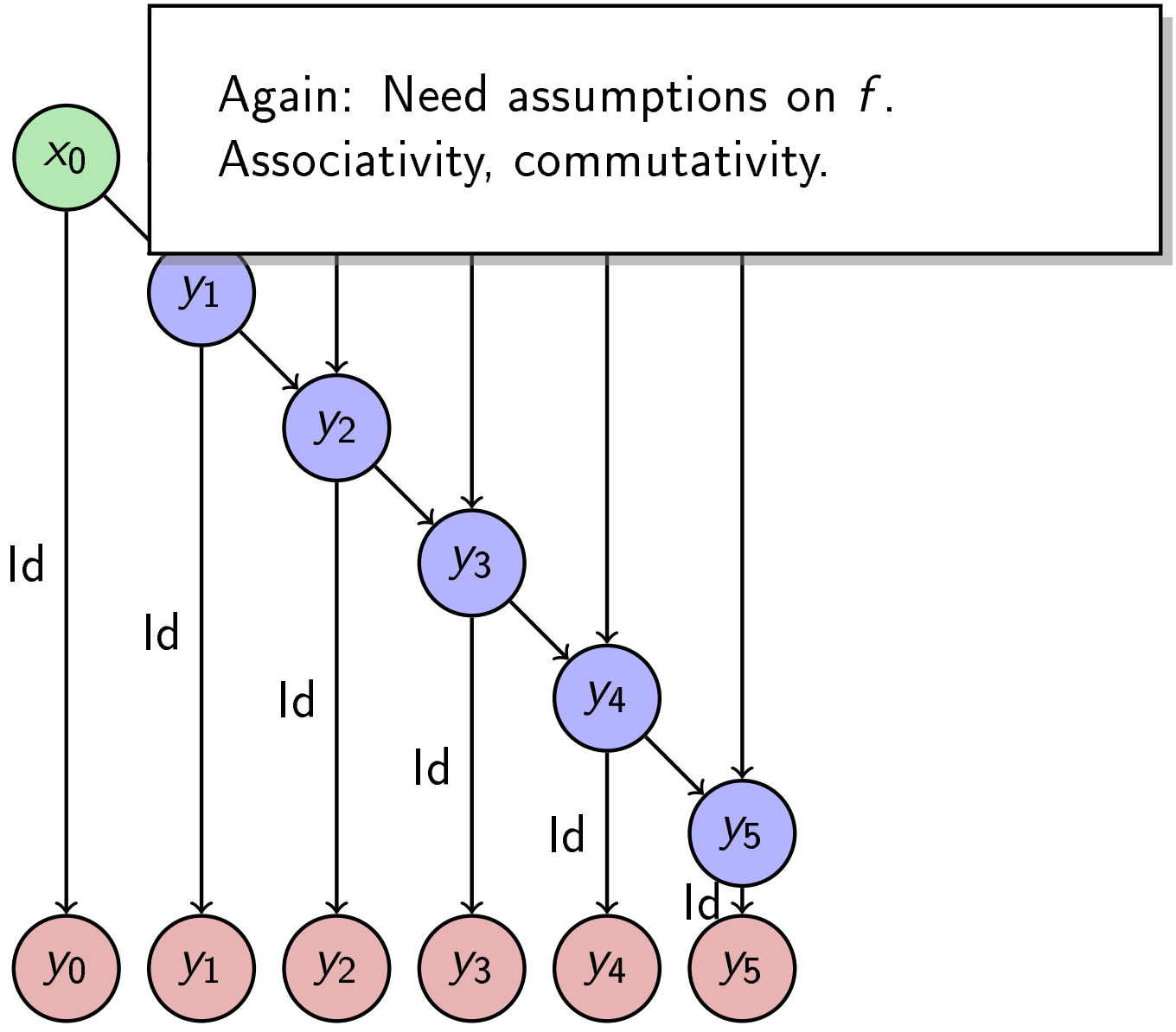
Scan

$$
\begin{aligned}
y_1 &= x_1 \\
y_2 &= f(y_1, x_2) \\
&\vdots = \vdots \\
y_N &= f(y_{N-1}, x_N)
\end{aligned}
$$

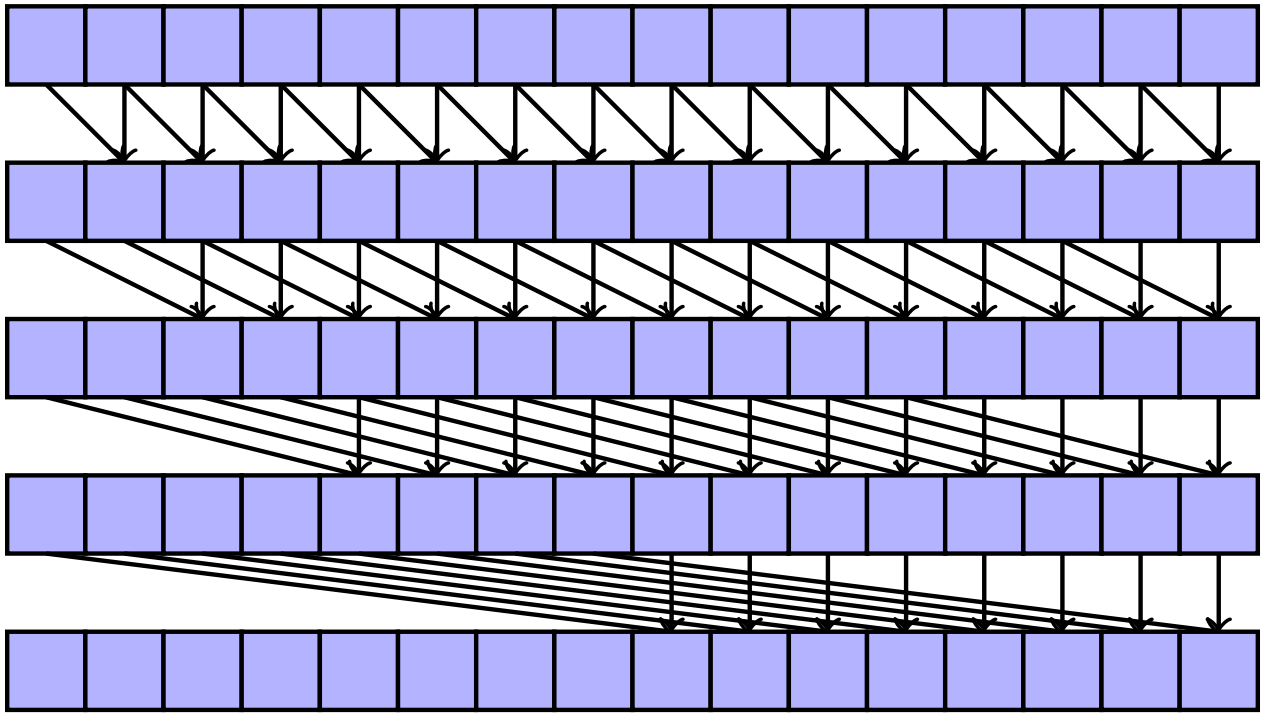
where N is the input size. (Think: N large, $f(x, y) = x + y$)

- Prefix Sum/Cumulative Sum
- Abstract view of: loop-carried dependence
- Also possible: Segmented Scan

Scan: Graph

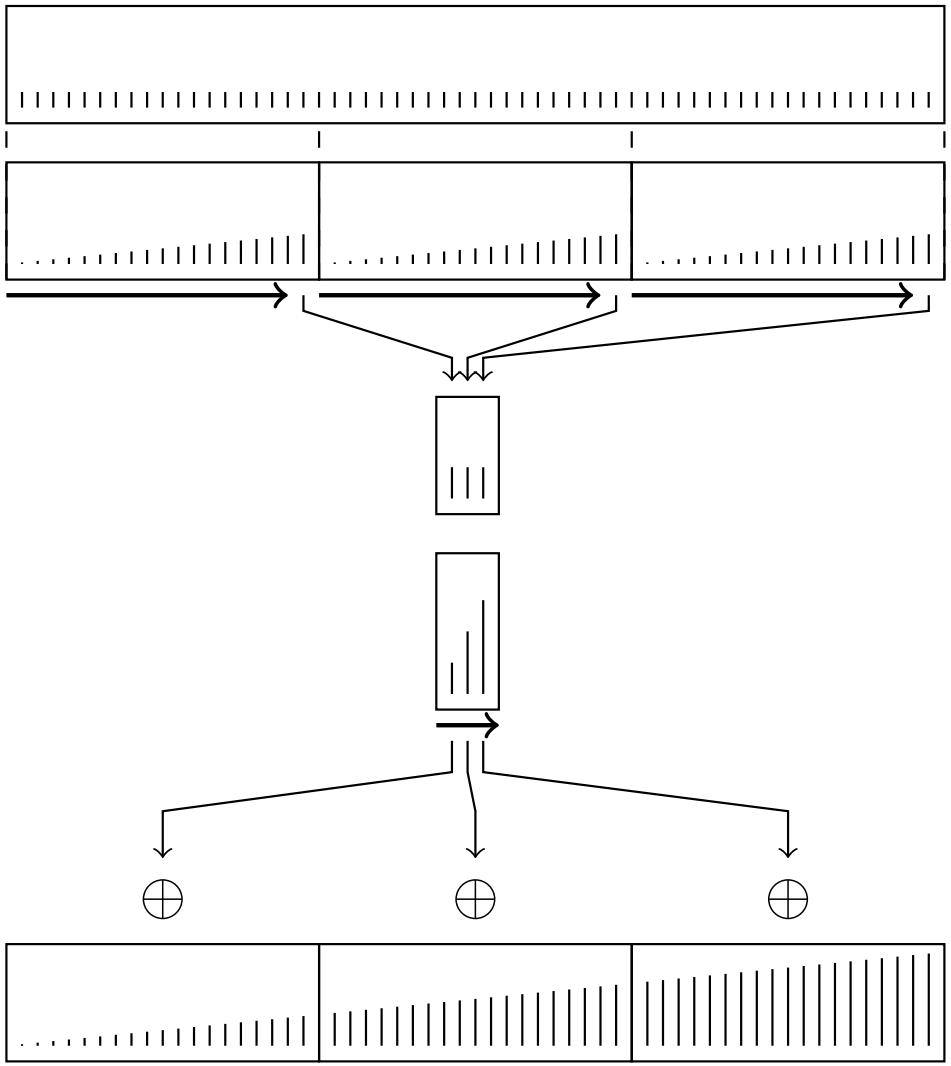

Again: Need assumptions on f.
Associativity, commutativity.
x_0
y_1
y_2
y_3
y_4
y_5
Id
Id
Id
Id
Id
Id
y_0
y_1
y_2
y_3
y_4
y_5

Scan: Implementation



Work-efficient?

Scan: Implementation II



Problem:

Trees alone often don't provide sufficient concurrency

Idea:

- Run multiple scans
- Combine results (also a scan)
- Run a final update

Scan: Examples

Name examples of Prefix Sums/Scans:

- Anything with a loop-carried dependence
- One row of Gauss-Seidel
- One row of triangular solve
- Segment numbering if boundaries are known
- Low-level building block for many higher-level algorithms algorithms, e.g. predicate filter, sort
- FIR/IIR Filtering
- Blelloch '93

Demo

[DEMO: patterns-03-scan]

Assignment

Use PyOpenCL scan to

- Generate 10,000,000 uniformly distributed single-precision random numbers in $[0, 1)$
- Make a new array that retains only the ones $\leq 1/2$

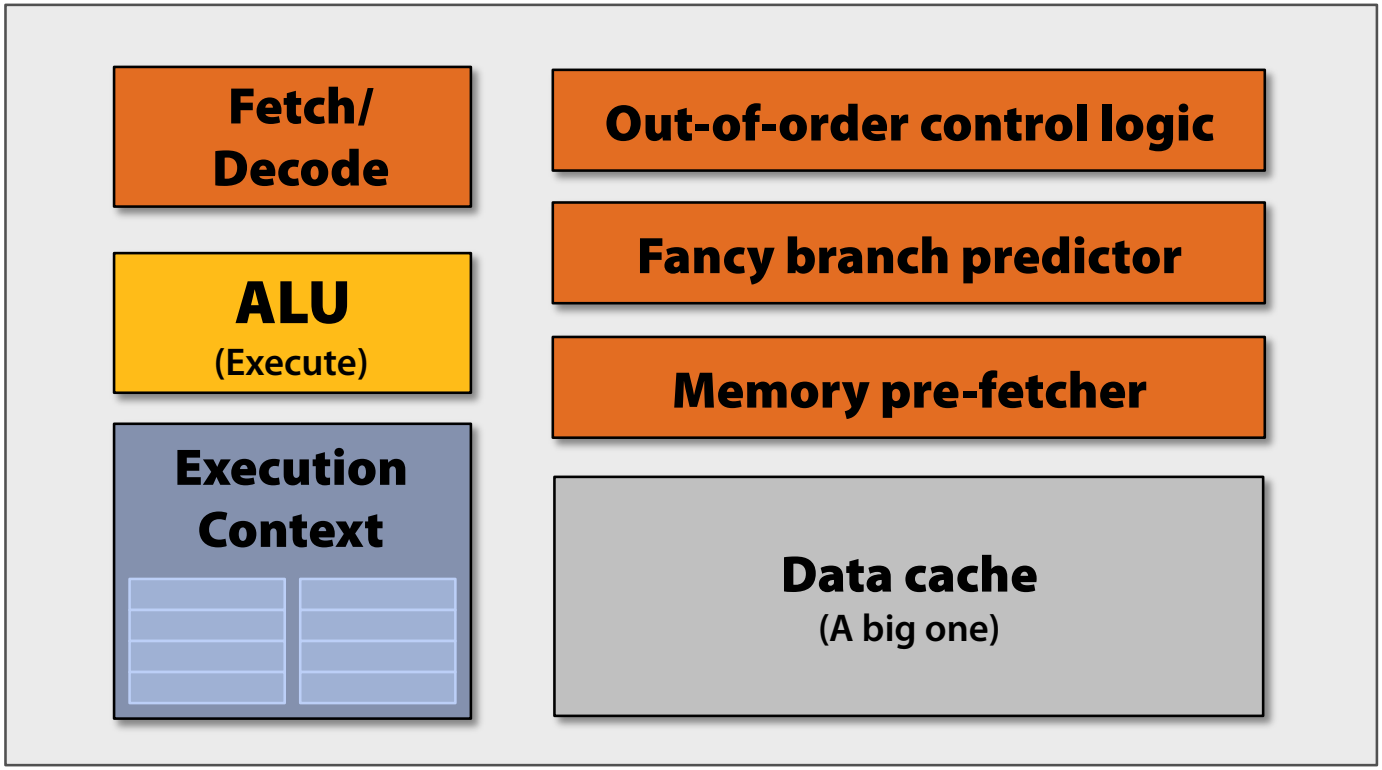
Practice

[DEMO: patterns-04-scan-practice]

Outline

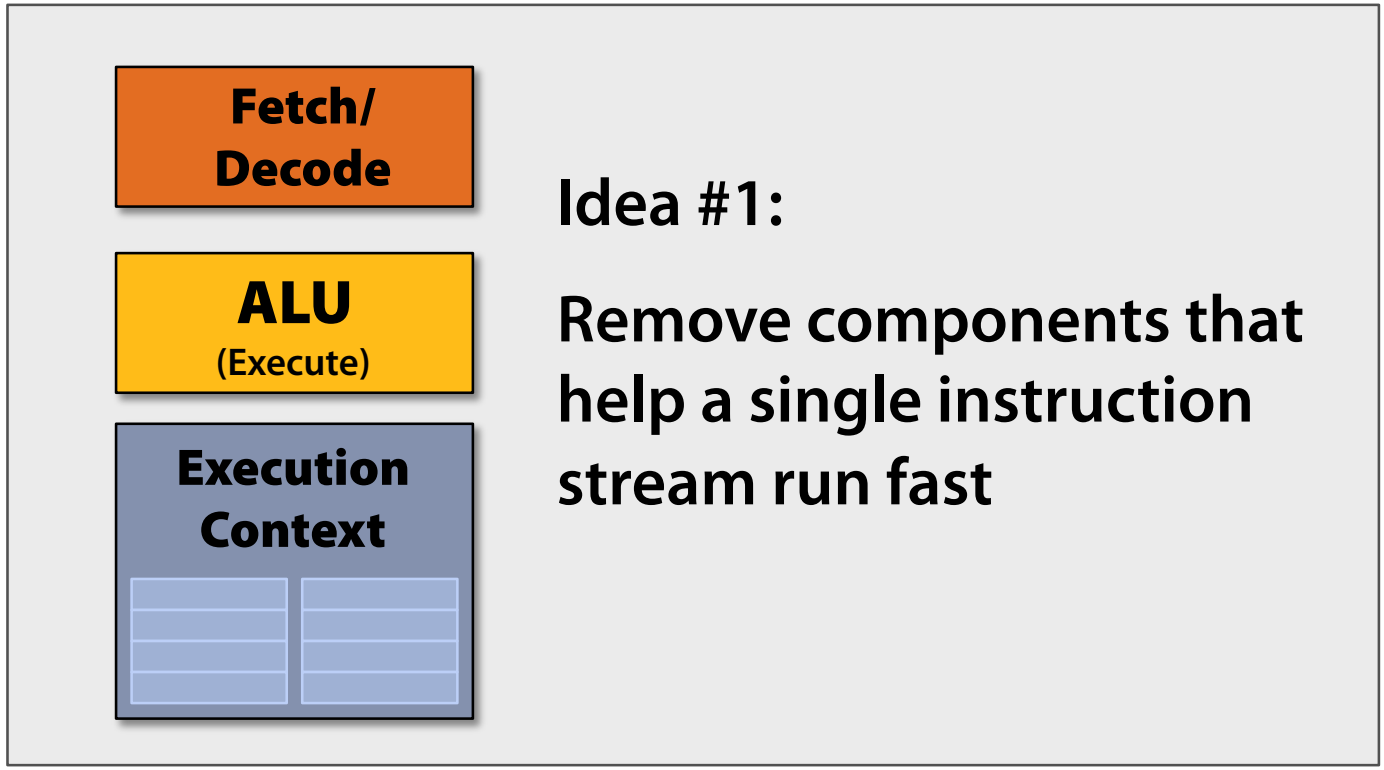
Outline

"CPU-style" Cores



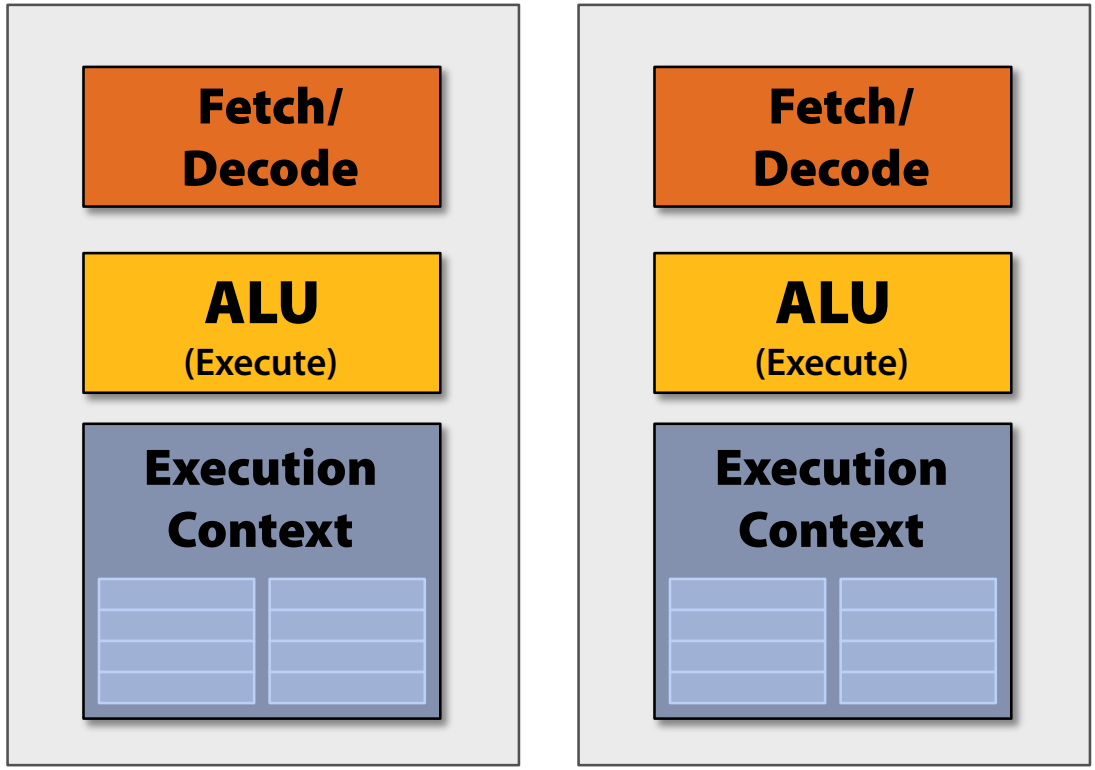
[Fatahalian '08]

Slimming down



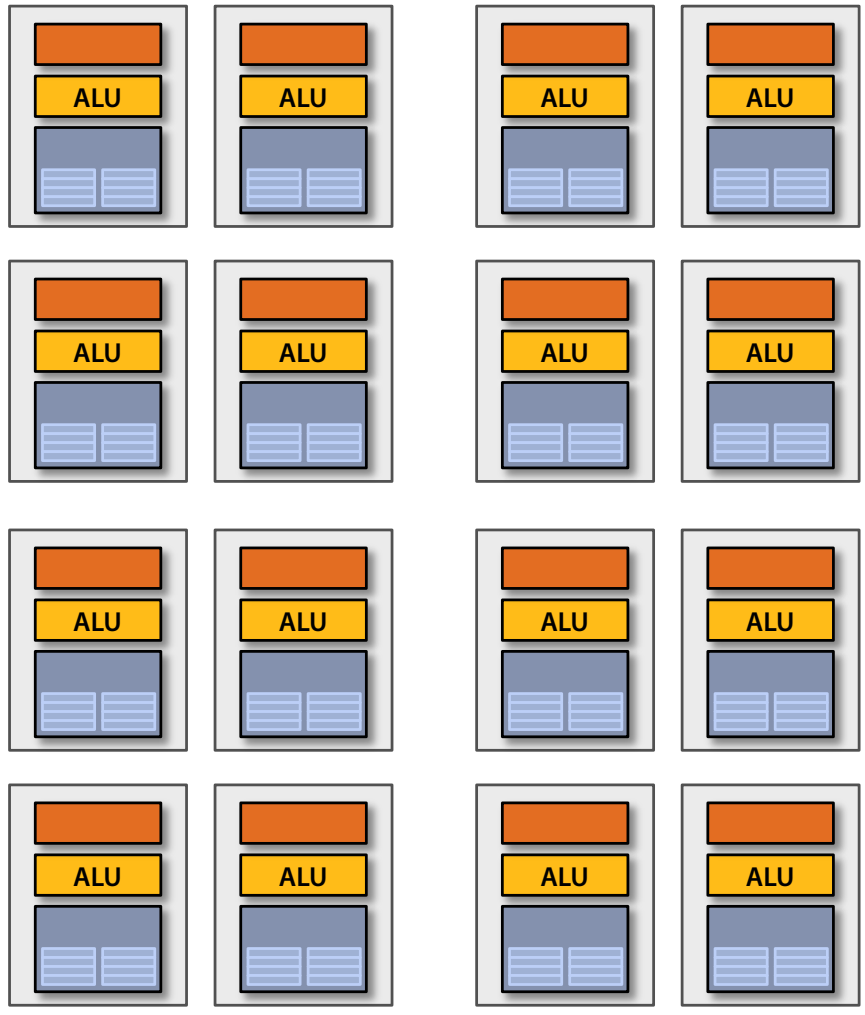
[Fatahalian '08]

More Space: Double the Number of Cores



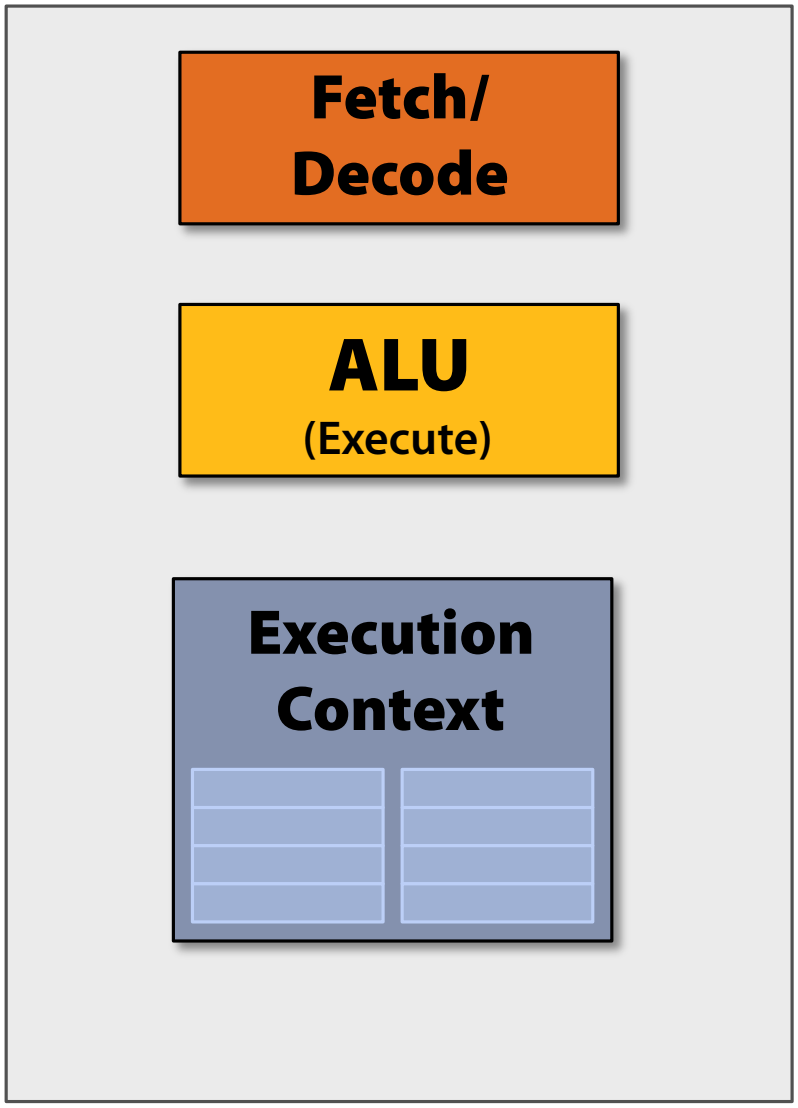
[Fatahalian '08]

Even more



[Fatahalian '08]

SIMD

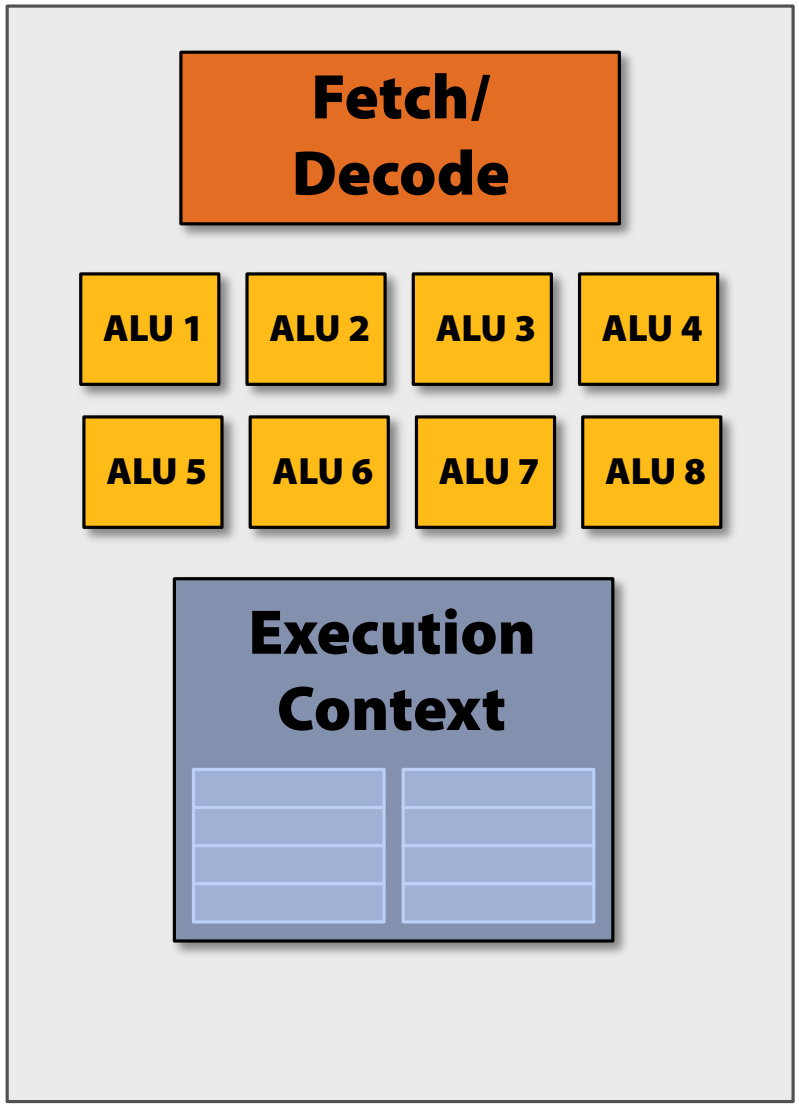


[Fatahalian '08]

Idea #2: SIMD

Amortize cost/complexity of managing an instruction stream across many ALUs

SIMD

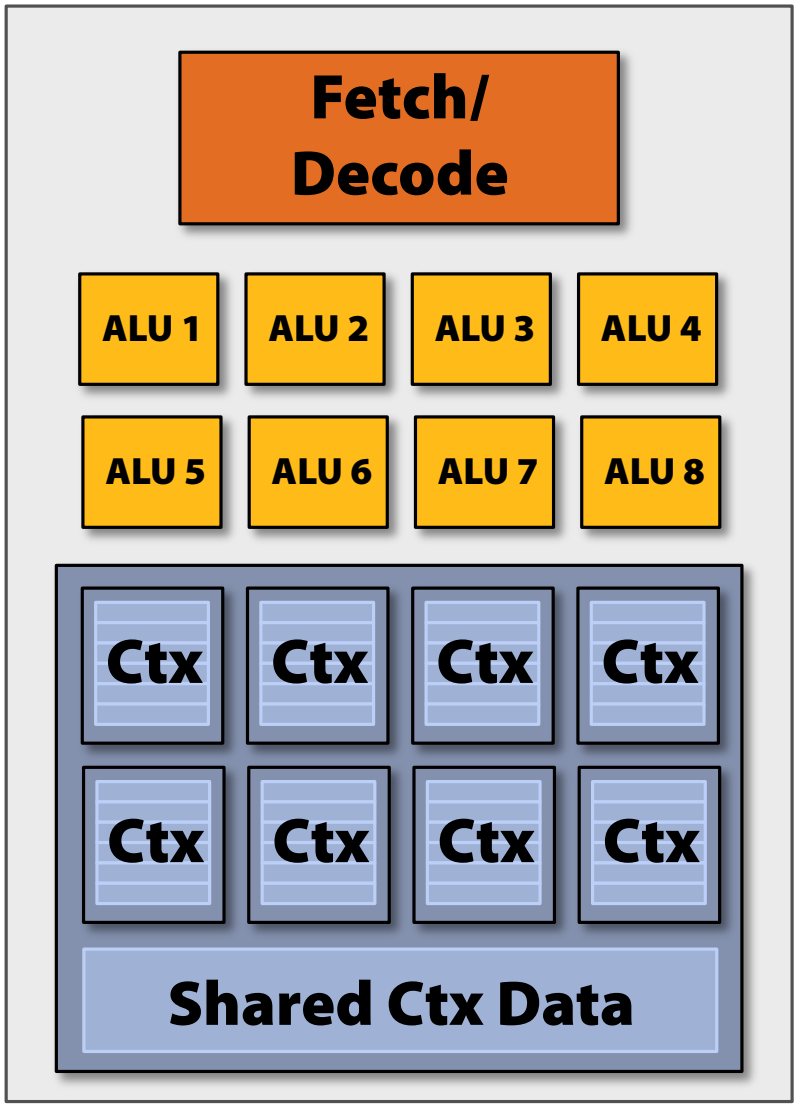


[Fatahalian '08]

Idea #2: SIMD

Amortize cost/complexity of managing an instruction stream across many ALUs

SIMD



[Fatahalian '08]

Idea #2: SIMD

Amortize cost/complexity of managing an instruction stream across many ALUs

Latency Hiding

- Latency (mem, pipe) hurts non-OOO cores
- Do *something* while waiting

What is the unit in which work gets scheduled on a GPU?

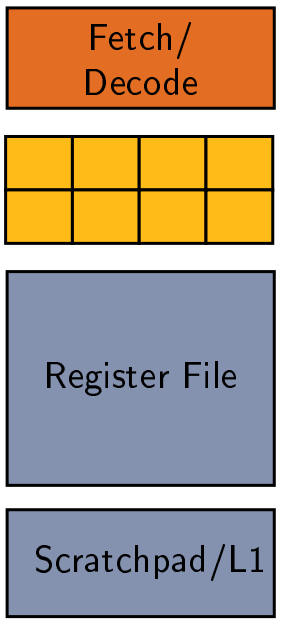
> A SIMD vector ('warp' (Nvidia), 'Wavefront' (AMD))

How can we keep busy?

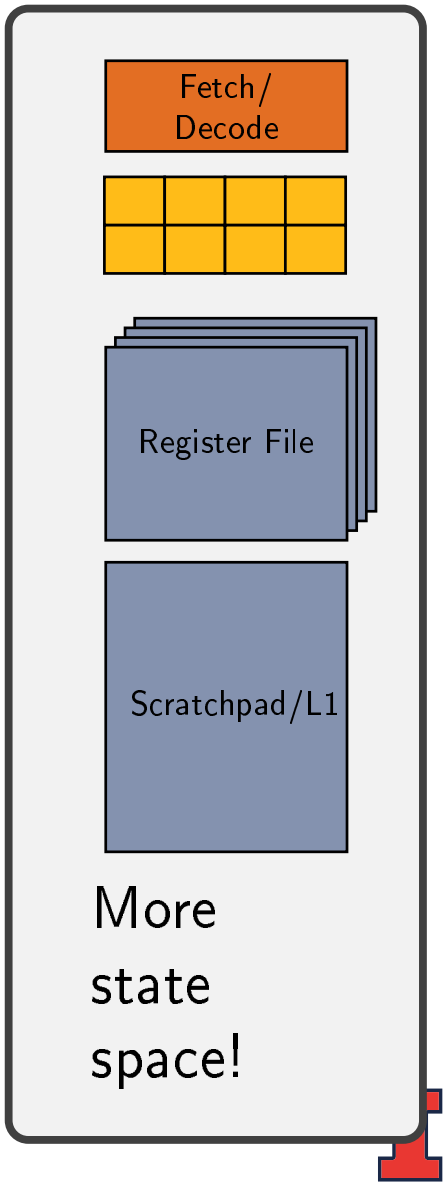
> - More vectors (bigger group)
> - ILP

Change in architectural picture?

Before:



After:



GPUs: Core Architecture Ideas

Three core ideas:

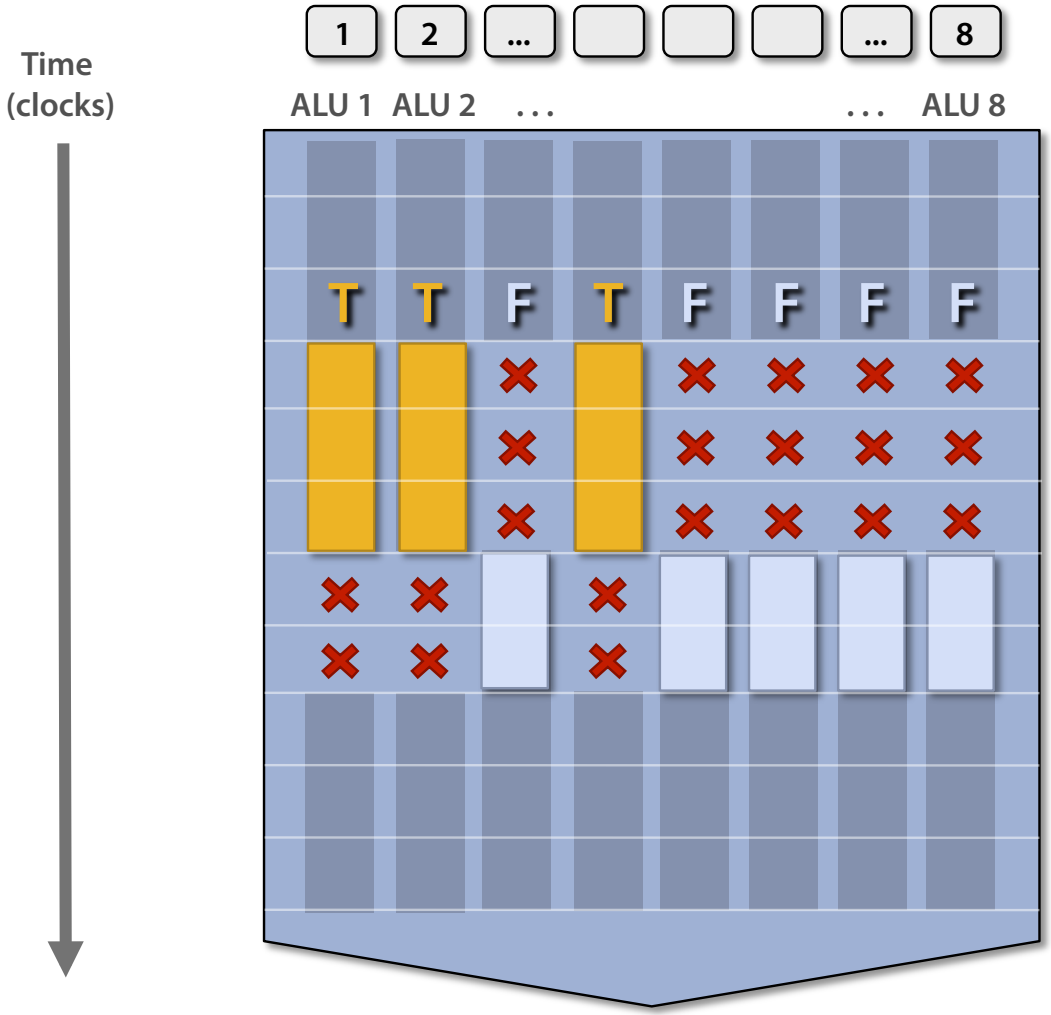
- Remove things that help with latency in single-thread
- Massive core and SIMD parallelism
- Cover latency with concurrency
  - SMT
  - ILP

GPU Abstraction: Core Model Ideas

How do these aspects show up in the model?

- View concrete counts as an implementation detail
  - SIMD lane
  - Core
  - Scheduling slot
- Program as if there are infinitely many of them
- Hardware division is expensive
  Make *n*D grids part of the model to avoid it
- Design the model to expose *extremely* fine-grain concurrency
  (e.g. between loop iterations!)
- Draw from the same pool of concurrency to hide latency

‘SIMT’ and Branches



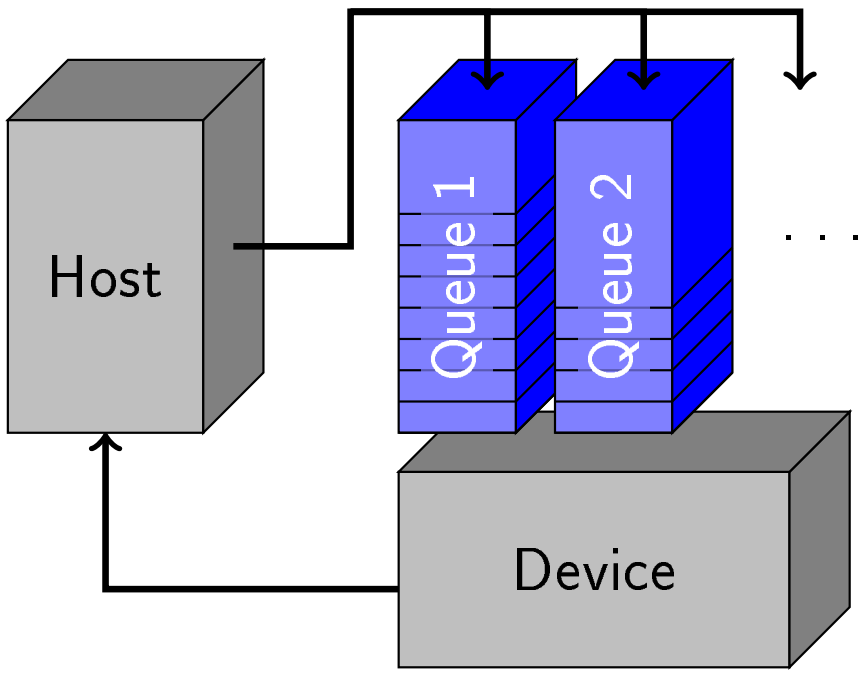

```
<unconditional
shader code>

if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}

<resume unconditional
shader code>
```

[Fatahalian ‘08]

Outline

Host-Device Concurrency

- Host and Device run asynchronously
- Host submits to queue:
  - Computations
  - Memory Transfers
  - Sync primitives
  - ...
- Host can wait for:
  - *drained* queue
  - Individual "events"
- Profiling



Demo: Timing GPU Work

[DEMO: gpu-01-timing-queues]

How do you find the execution time of a GPU kernel?

- Do a few 'warm-up' calls to the kernel
- Drain the queue
- Start the timer
- Run the kernel enough times to get to a few milliseconds run time
- Drain the queue
- Stop the timer, divide by the number of runs

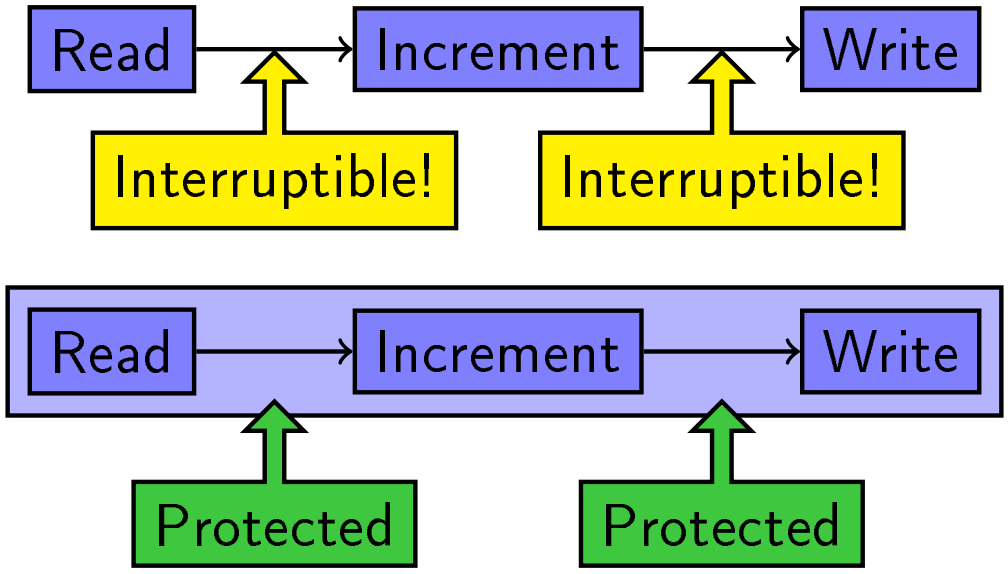
How do you do this asynchronously?

- Enqueue 'markers' instead of draining the queue.
- Find timing of 'markers' after work is complete

Demo: Intra-Group Synchronization

[DEMO: gpu-02-barrier-sync]

'Conventional' vs Atomic Memory Update



Atomic Operations: Compare-and-Swap

```
#include <stdatomic.h>
_Bool atomic_compare_exchange_strong(
        volatile A* obj,
        C* expected, C desired );
```

What does `volatile` mean?

> Memory may change at any time, do not keep in register.

What does this do?

> - Store `(*obj == *expected) ? desired : *obj` into `*obj`.
> - Return `true` iff memory contents was as `expected`.

How might you use this to implement atomic FP multiplication?

> Read previous, perform operation, try CAS, maybe retry

Memory Ordering

Why is Memory Ordering a Problem?

- Out-of-order CPUs reorder memory operations
- Compilers reorder memory operations

What are the different memory orders and what do they mean?

- Atomicity itself is unaffected
- Makes sure that 'and then' is meaningful

Types:

- Sequentially consistent: no reordering
- Acquire: later loads may not reorder across
- Release: earlier writes may not reorder across
- Relaxed: reordering OK

Implementing Locking?

Can we just do locking like we might do on a CPU?

- Indepdendent forward progress of all threads is not guaranteed.
  (true until recently)
- But: Device partitioning can help!

Discussion: Ways to Realize SpMV

What to parallelize over? Advantages/disadvantages?

- Rows of the matrix
  - Upside: no write races
  - Downsides: load balance? limited concurrency? data reuse?
- Matrix entries
  - Upside: load balance
  - Downside: write races?
- ...

GPU Communication 'Scopes'

Hardware	CL adjective	CL noun	CUDA
SIMD lane	private	Work Item	Thread
SIMD Vector	—	Subgroup	Warp
Core	local	Workgroup	Thread Block
Processor	global	`NDRange`	Grid
Machine	—	—	—

GPU: Communication

What forms of communication exist within each scope?

- Subgroup: Shuffles (!)
- Workgroup:
  - Scratchpad + barrier
  - local atomics + mem fence
- Grid: Global atomics
- Machine:
  - Global atomics (requires coherence)
  - Queues
  - Events

Host-Device Data Exchange

Sad fact: Must get data onto device to compute

- Transfers can be a bottleneck
- If possible, overlap with computation
- Pageable memory incurs difficulty in GPU-host transfers, often entails (another!) CPU side copy
- "Pinned memory": unpageable, avoids copy
  - Various *system-defined* ways of allocating pinned memory

"Unified memory" (CUDA)/"Shared Virtual Memory" (OpenCL):

- GPU directly accesses host memory
- Main distinction: Coherence
  - "Coarse grain": Per-buffer fences
  - "Fine grain buffer": Byte-for-byte coherent (device mem)
  - "Fine grain system": Byte-for-byte coherent (anywhere)

Outline

Outline

Performance: Ballpark Numbers?

Bandwidth host/device:

> PCIe v2: 8 GB/s — PCIe v3: 16 GB/s — NVLink: 200 GB/s

Bandwidth on device:

> Registers: ∼10 TB/s — Scratch: ∼10 TB/s — Global: 500 GB/s

Flop throughput?

> 10 TFLOPS single precision – 3 TFLOPS double precision

Kernel launch overhead?

> 10 microseconds

Good source of details: Wikipedia: List of Nidia GPUs

Qualifying Performance

- What is *good* performance?
- Is speed-up (e.g. GPU vs CPU? C vs Matlab?) a meaningful way to assess performance?
- How else could one *form an understanding* of performance?

> Modeling: how understanding works in science

Hager et al. '17
Hockney et al. '89

A Story of Bottlenecks

Imagine:

- A bank with a few service desks
- A revolving door at the entrance

What situations can arise at *steady-state*?

- Line inside the bank (good)
- Line at the door (bad)

What numbers do we need to characterize performance of this system?

- P_{peak}: [task/sec] Peak throughput of the service desks
- I: [tasks/customer] Intensity
- b: [customers/sec] Throughput of the revolving door

A Story of Bottlenecks (cont'd)

- P_{peak}: [task/sec] Peak throughput of the service desks
- I: [tasks/customer] Intensity
- b: [customers/sec] Throughput of the revolving door

What is the aggregate throughput?

Bottleneck is either

- the service desks (good) or
- the revolving door (bad).

$$P \leq \min(P_{\text{peak}}, I \cdot b)$$

Hager et al. '17

Application in Computation

Translate the bank analogy to computers:

- Revolving door: typicallly: Memory interface
- Revolving door throughput: Memory bandwidth [bytes/s]
- Service desks: Functional units (e.g. floating point)
- P_{peak}: Peak FU throughput (e.g.: [flops/s])
- Intensity: e.g. [flops/byte]

Which parts of this are task-dependent?

- All of them! This is not *a* model, it's a *guideline* for making models.
- Specifically P_{peak} varies substantially by task

$$P \leq \min(P_{\text{peak}}, I \cdot b)$$

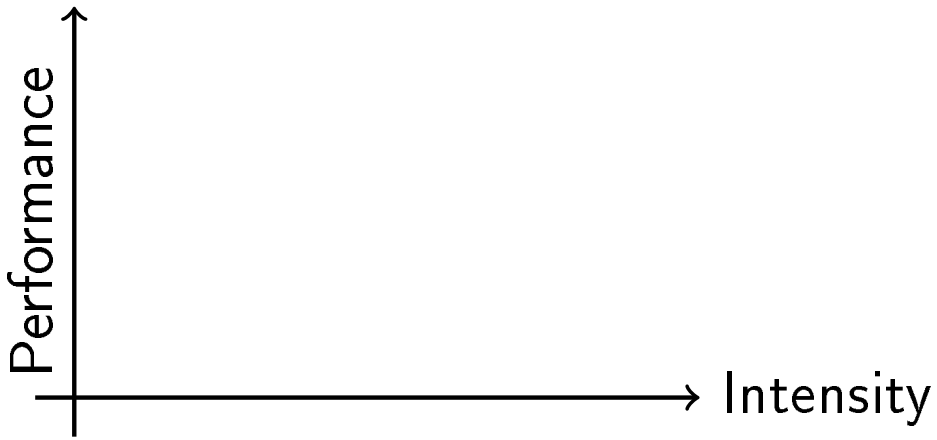
A Graphical Representation: 'Roofline'

Plot (often log-log, but not necessarily):

- X-Axis: Intensity
- Y-Axis: Performance

What does our inequality correspond to graphically?

$$P \leq \min(P_{\text{peak}}, I \cdot b)$$



What does the shaded area mean?

Achievable performance

Hager et al. '17

Example: Vector Addition

```
double r, s, a[N];
for (i=0; i<N; ++i)
  a[i] = r + s * a[i];}
```

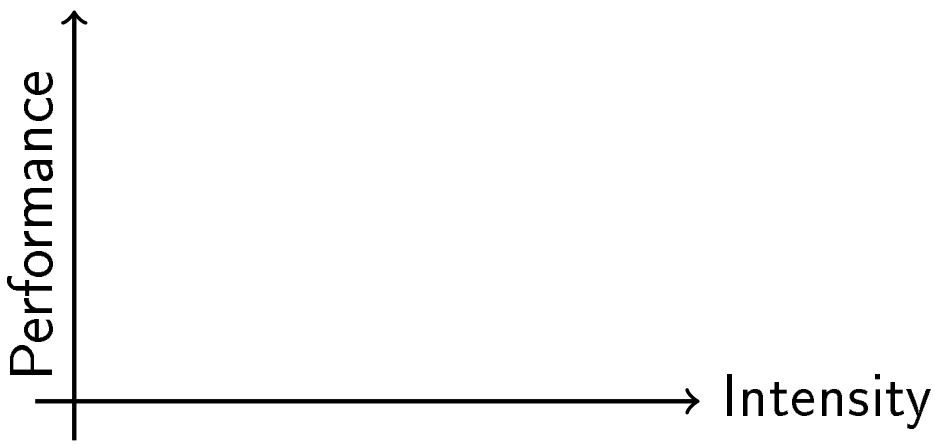
Find the parameters and make a prediction.

Machine model:

- Memory Bandwidth: e.g. $b = 10$ GB/s
- P_{peak}: e.g. 4 GF/s

Application model:

- $I = 2$ flops / 16 bytes $= 0.125$ flops/byte

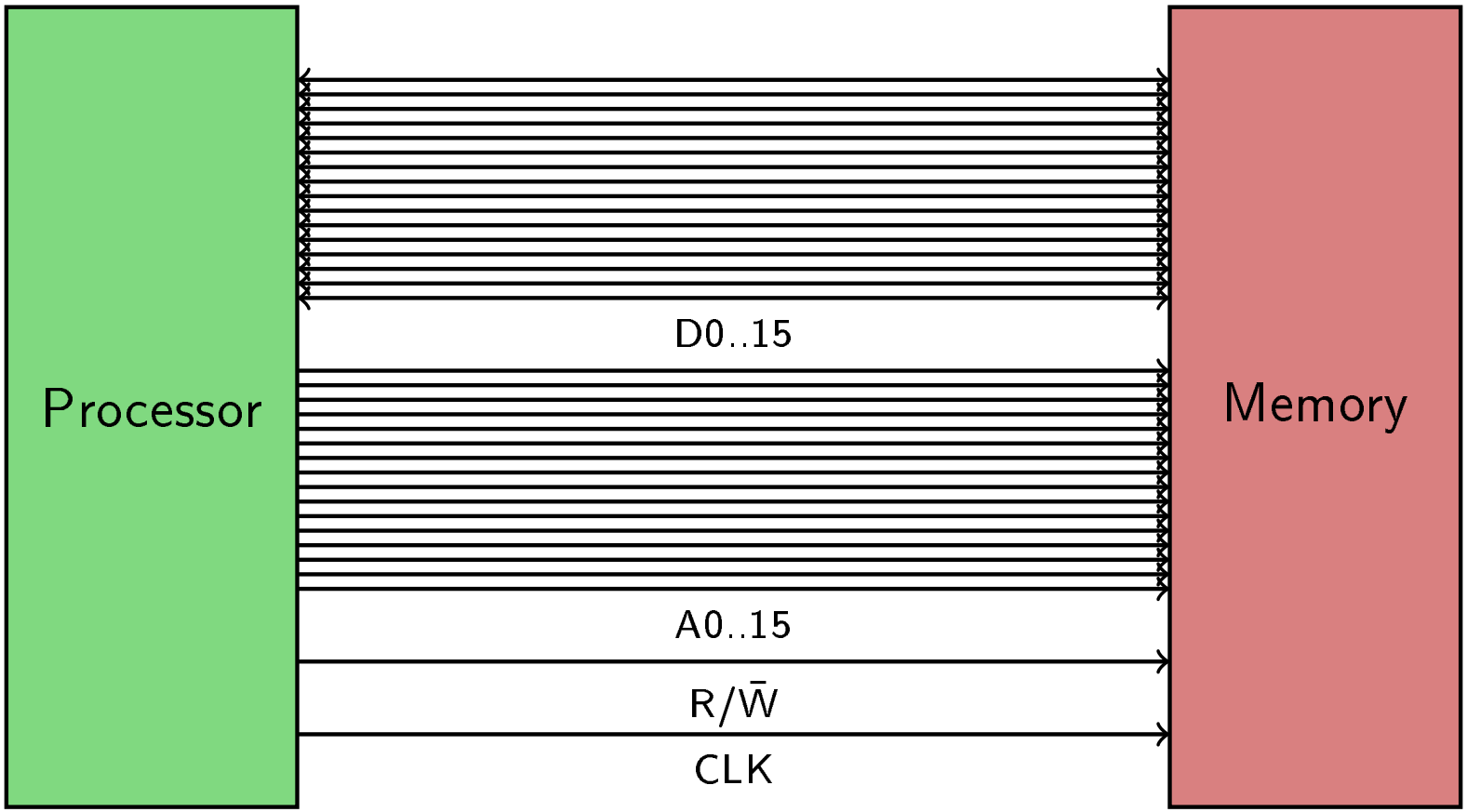


Demo: Performance Modeling

[DEMO: perf-01-modeling]

Outline

Memory Systems: Bird's Eye View



Somewhere Behind the Interconnect: Memory

Performance characteristics of memory:

- Bandwidth
- Latency

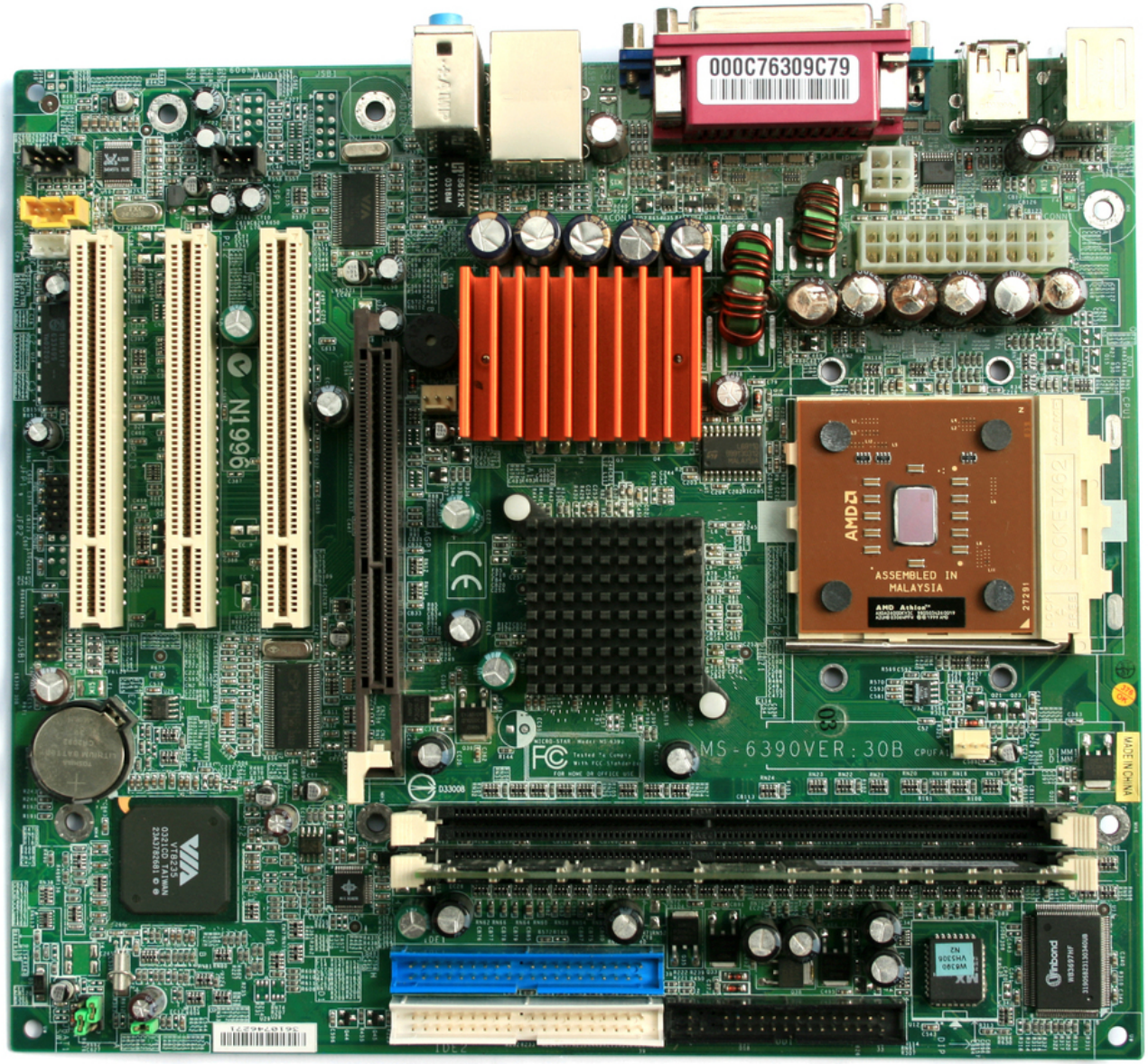
  Flops are cheap
  Bandwidth is money
  Latency is physics

  - *M. Hoemmen*

Minor addition (but important for us)?

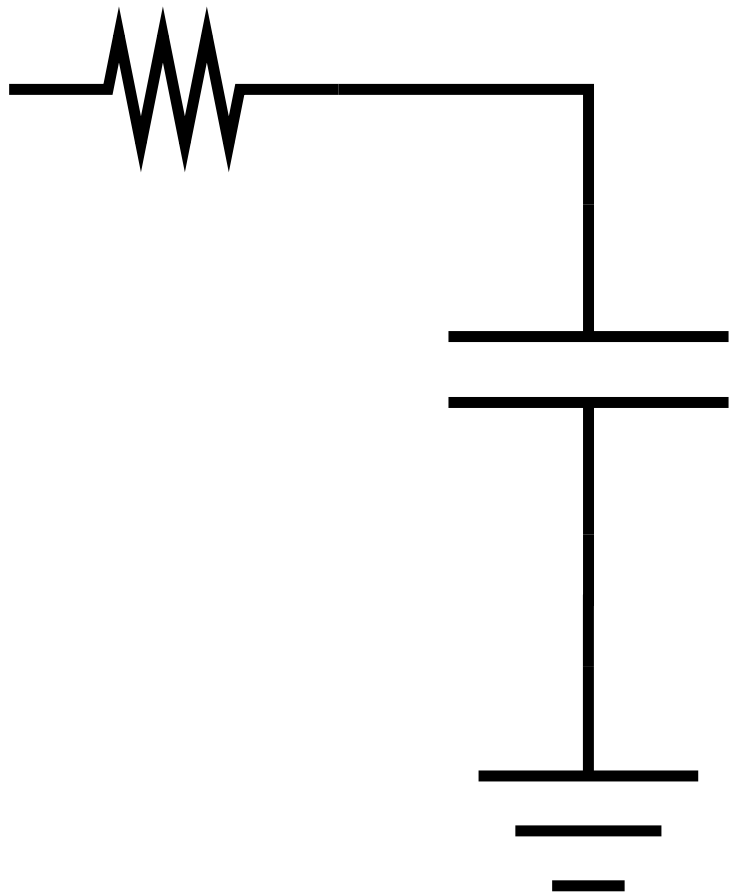
- Bandwidth is money and code structure

Latency is Physics: Distance

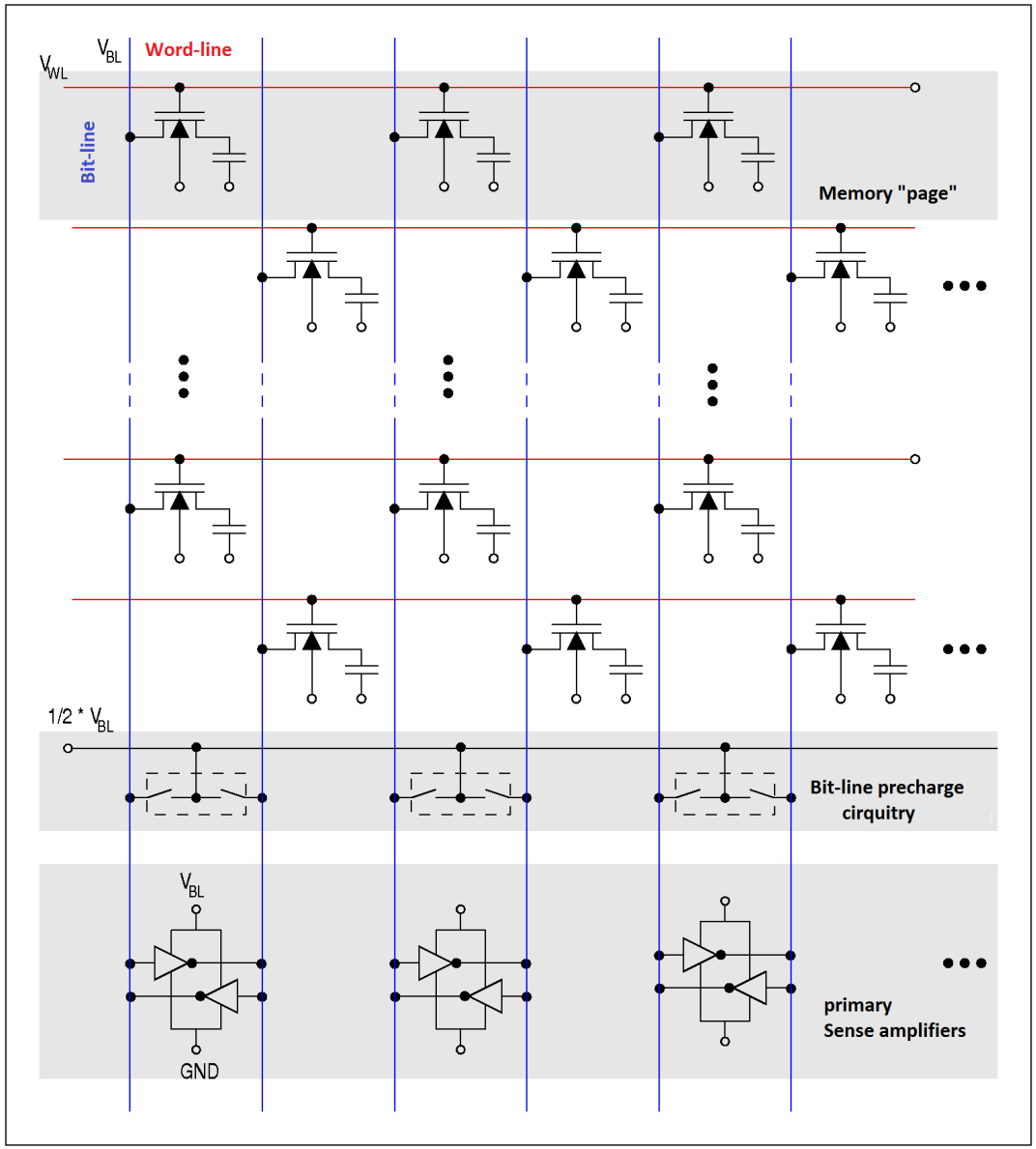


[Wikipedia ©]

Latency is Physics: Electrical Model



Latency is Physics: DRAM



[Wikipedia ©]

Alignment

Alignment describes the process of matching the base address of:

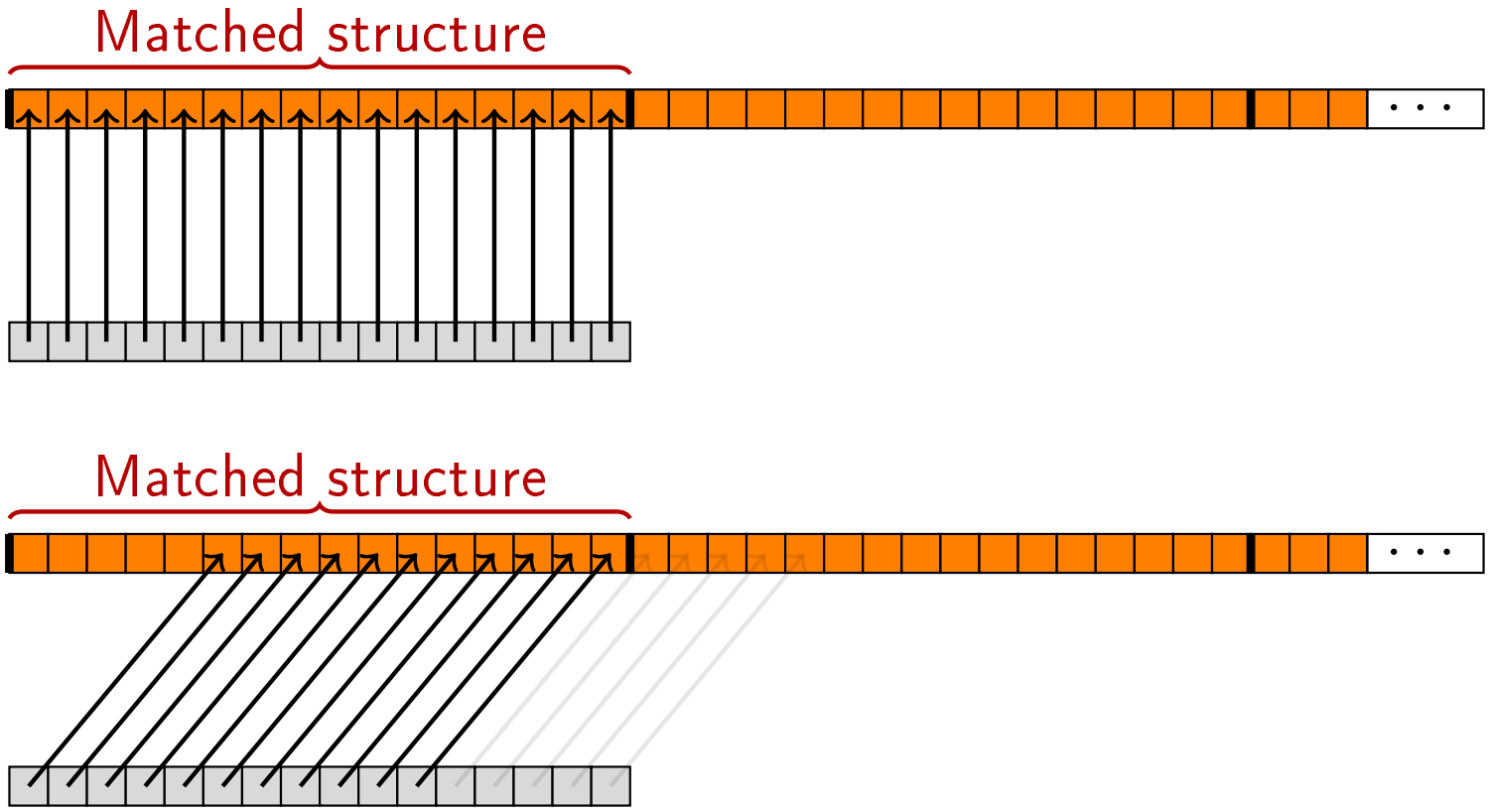
- Single word: `double`, `float`
- SIMD vector
- Larger structure

To machine granularities:

- Natural word size
- Vector size
- Cache line

Q: What is the performance impact of misalignment?

Performance Impact of Misalignment



Outline

Parallel Memories

Problem: Memory chips have only one data bus.
So how can multiple threads read multiple data items from memory simultaneously?

Broadly:

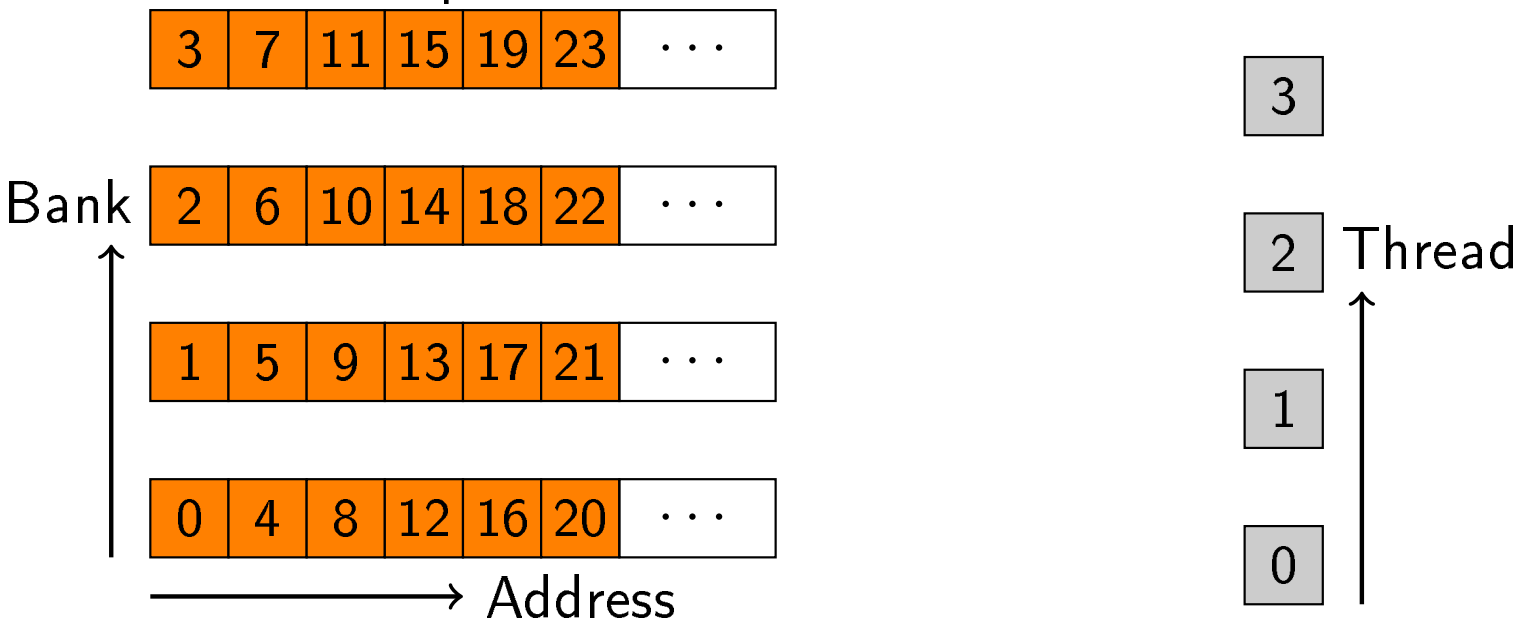
- Split a really wide data bus, but have only one address bus
- Have many 'small memories' ('*banks*') with separate data and address busses, select by address LSB.

Where does banking show up?

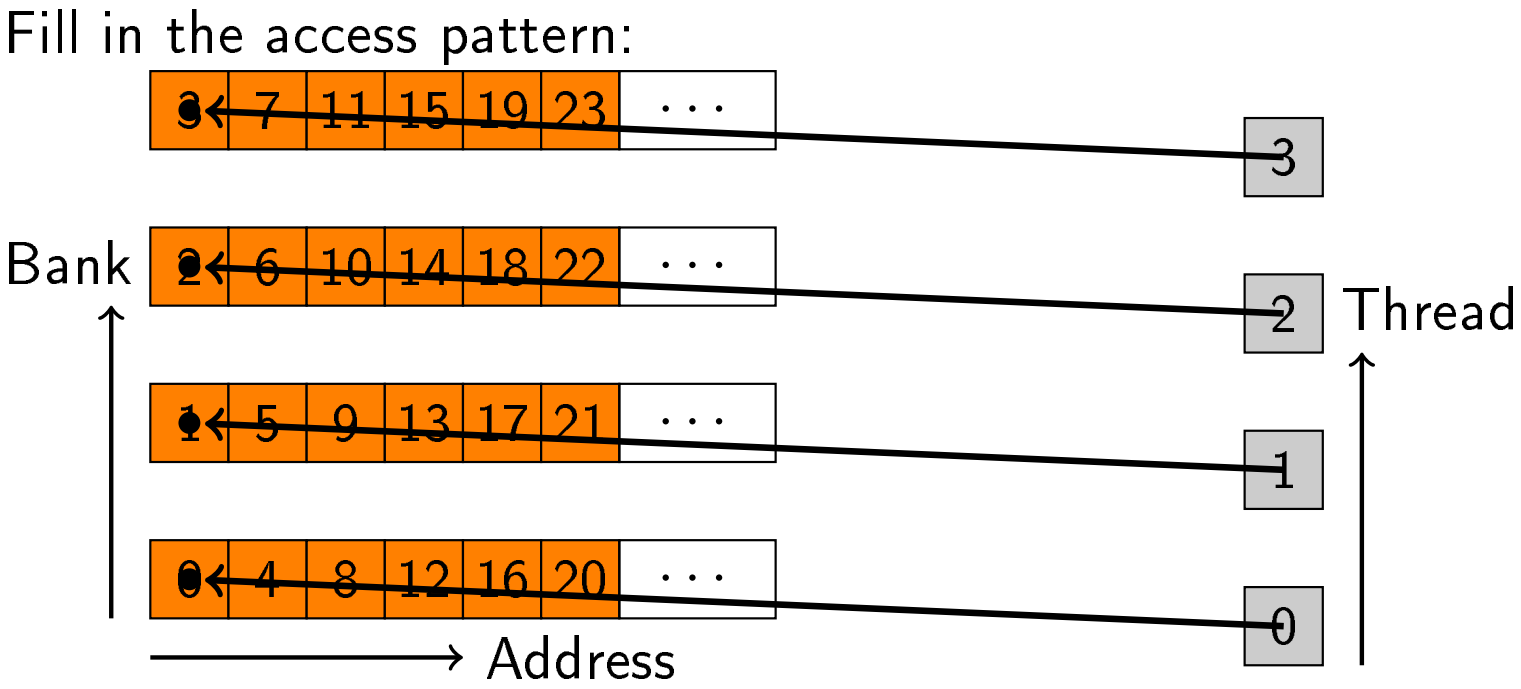
- Scratchpad
- GPU register file
- Global memory

Memory Banking

Fill in the access pattern:

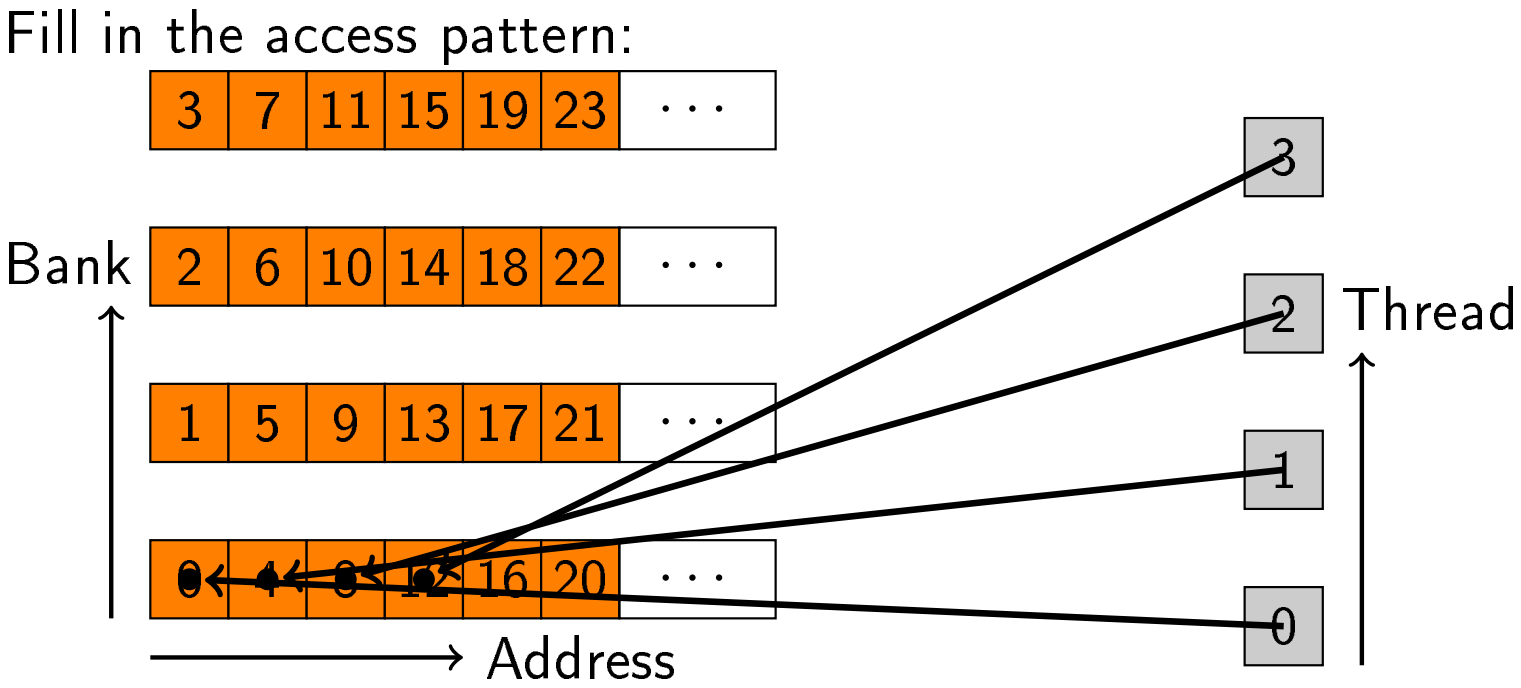


Memory Banking



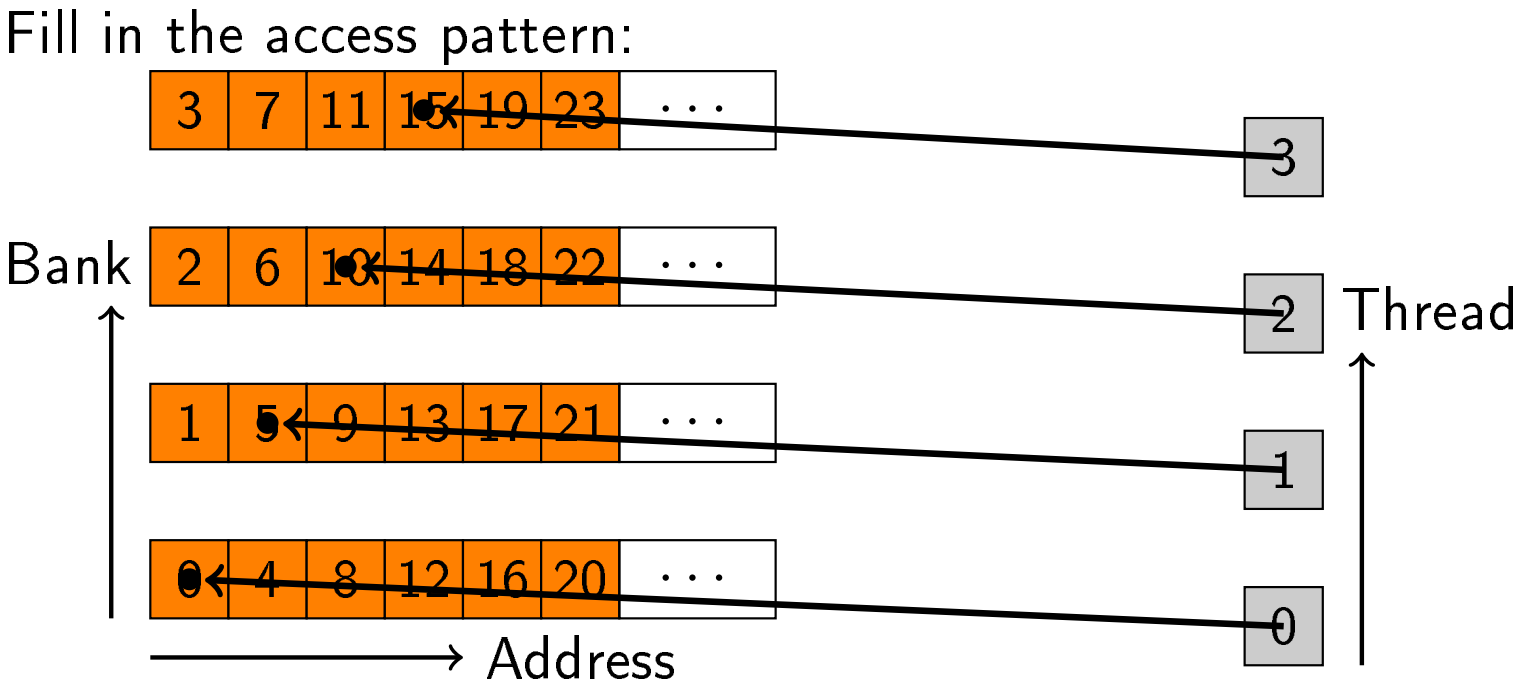

```
local_variable[lid(0)]
```

Memory Banking



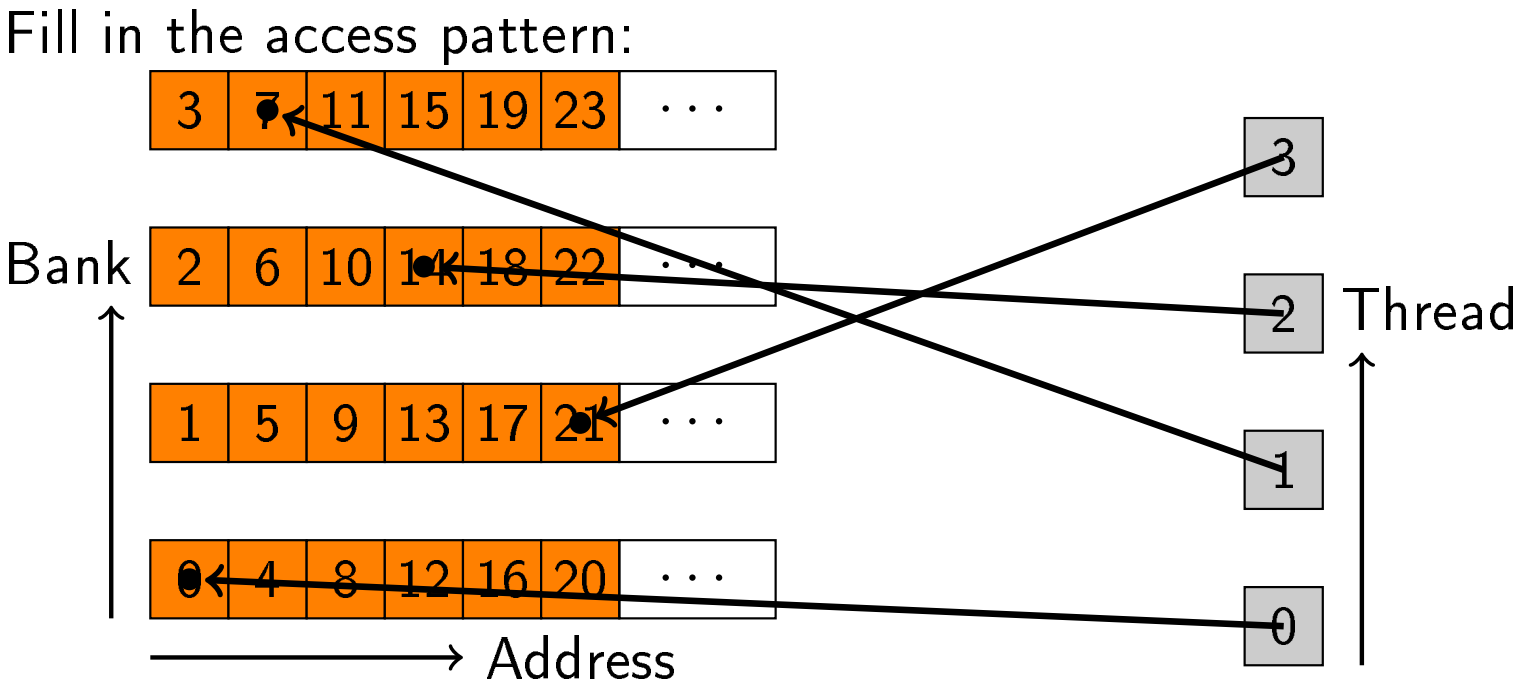

```
local_variable[BANK_COUNT*lid(0)]
```

Memory Banking




```
local_variable[(BANK_COUNT+1)*lid(0)]
```

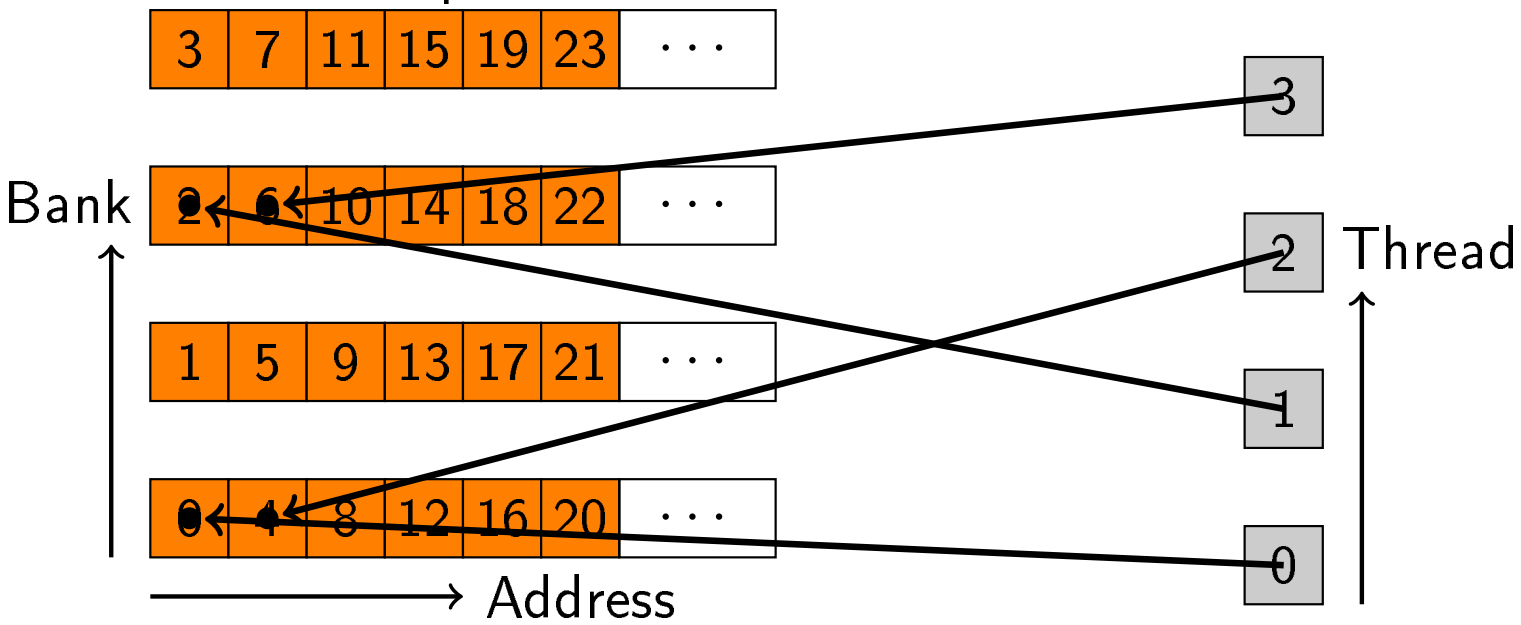
Memory Banking




```
local_variable[ODD_NUMBER*lid(0)]
```

Memory Banking

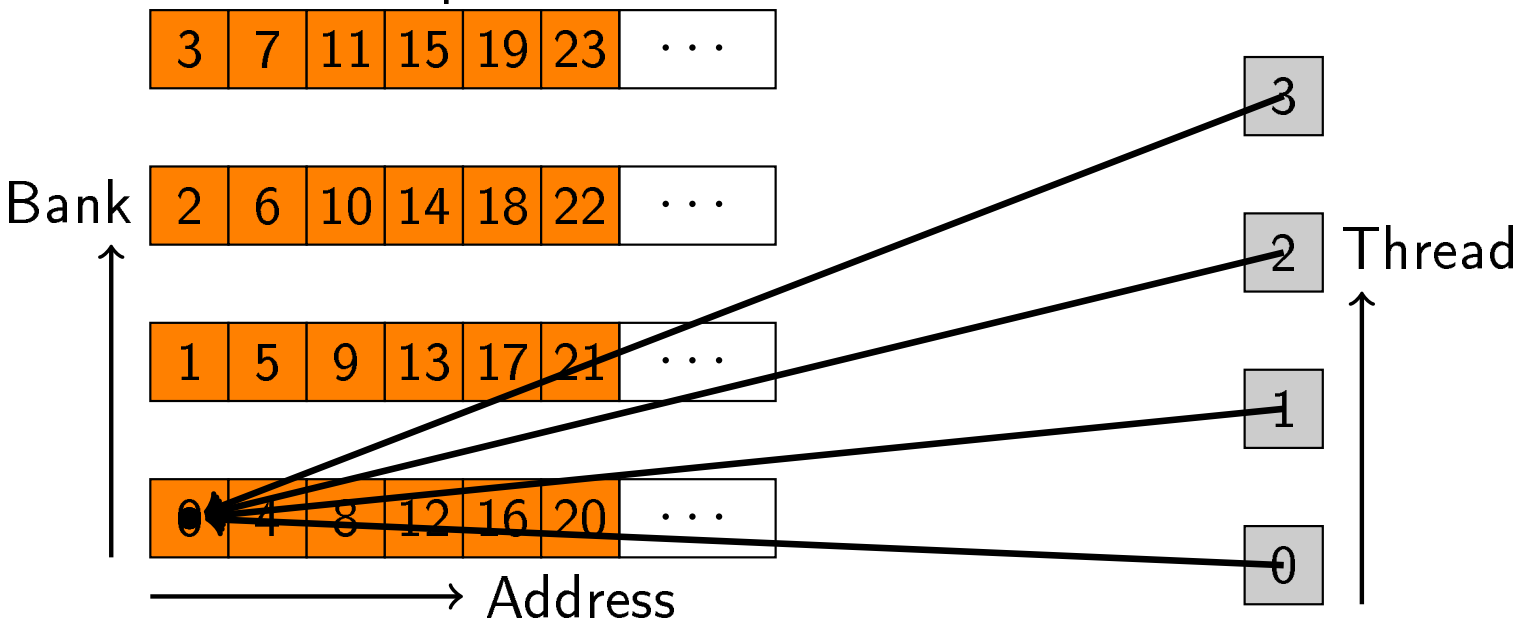
Fill in the access pattern:




```
local_variable[2*lid(0)]
```

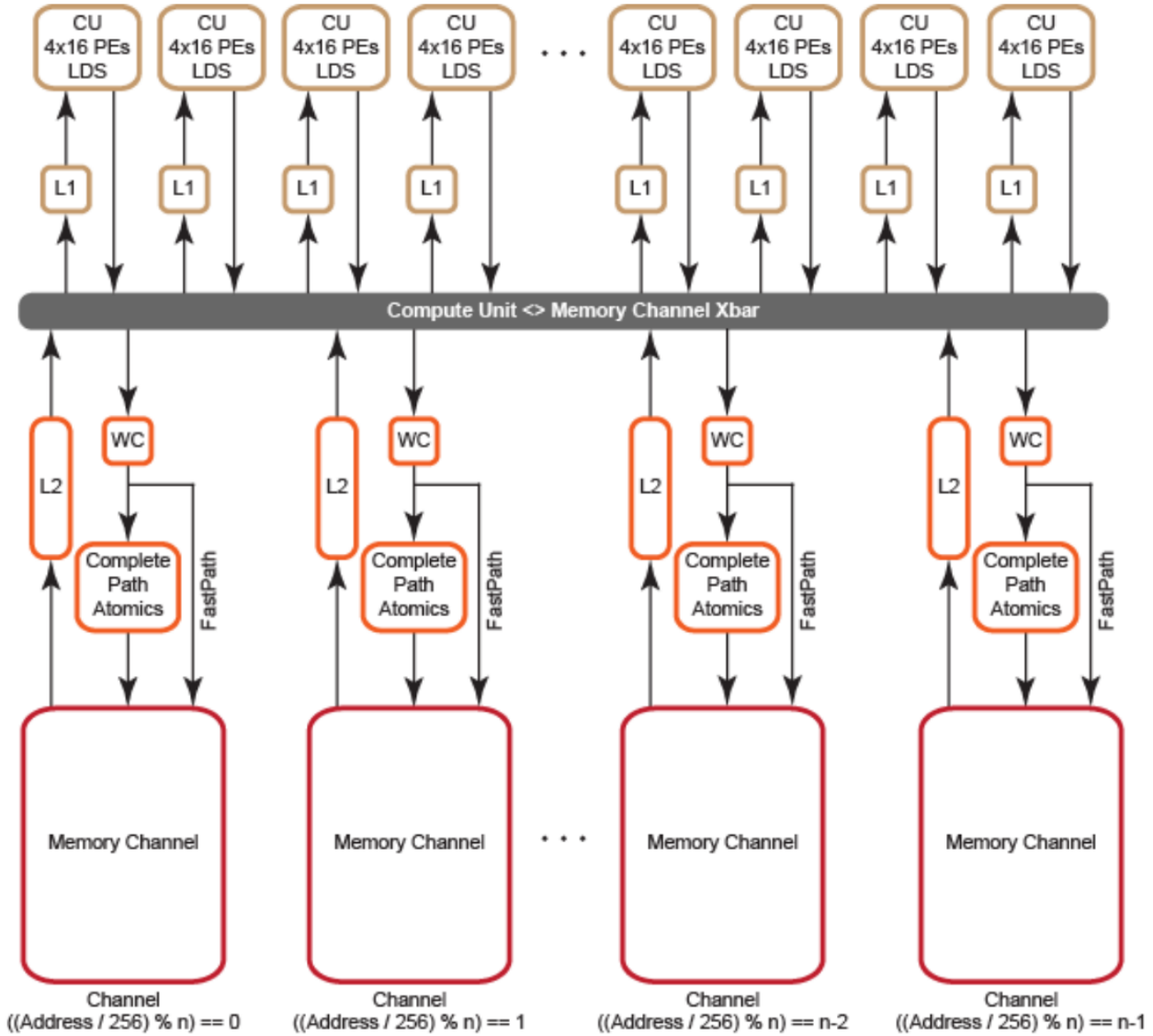
Memory Banking

Fill in the access pattern:




```
local_variable[f(gid(0))]
```

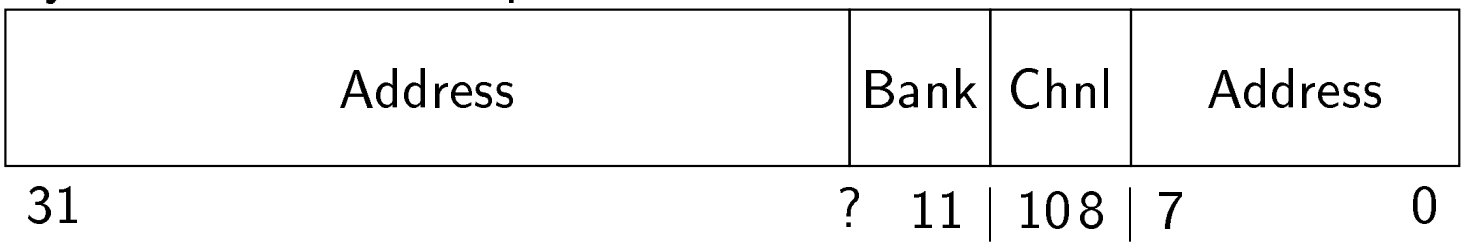
GPU Global Memory System



GCN Optimization Manual, AMD

GPU Global Memory Channel Map: Example

Byte address decomposition:



Implications:

- Transfers between compute unit and channel have granularity
  - Reasonable guess: warp/wavefront size $\times$ 32bits
  - Should strive for good utilization ('*Coalescing*')
- Channel count often *not* a power of two -> complex mapping
  - *Channel conflicts* possible
- Also *banked*
  - *Bank conflicts* also possible

GPU Global Memory: Performance Observations

Key quantities to observe for GPU global memory access:

- Stride
- Utilization

Are there any guaranteed-good memory access patterns?

Unit stride, just like on the CPU

- Need to consider access pattern *across entire device*
- *GPU caches:* Use for *spatial*, not for temporal locality
- Switch available: L1/Scratchpad partitioning
  - Settable on a per-kernel basis
- Since GPUs have meaningful caches at this point:
  Be aware of cache annotations (see later)

Demo: Matrix Transpose

[DEMO: perf-04-transpose]

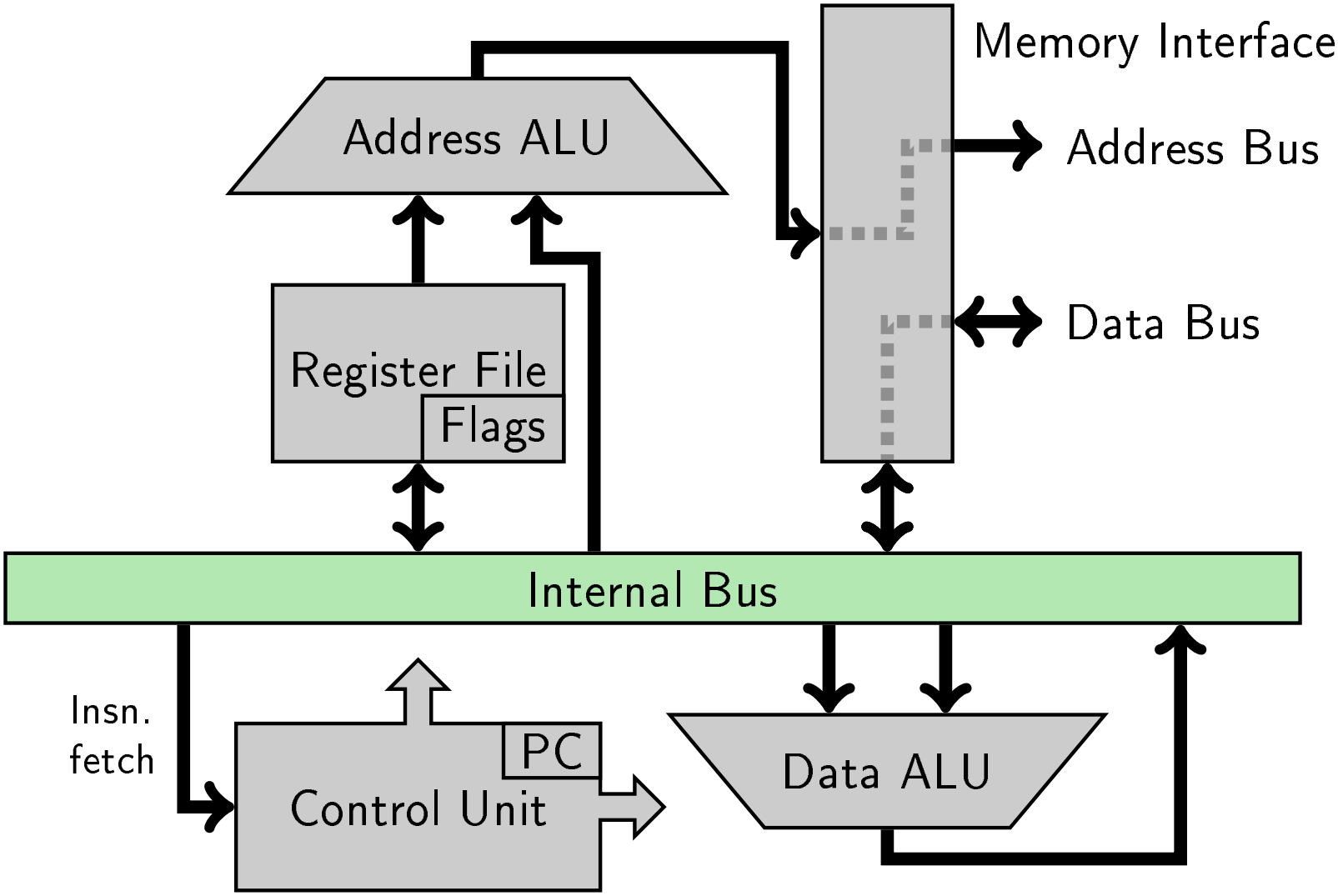
Performance: Limits to Concurrency

Concurrency is essential to good (memory) latency hiding.
What limits the amount of concurrency exposed to GPU hardware?

- Amount of register space
  Important: Size of (per-lane) register file is *variable*
- Amount of scratchpad space
  Size of (per-group) scratchpad space is *variable*
- Workgroup size
- Available ILP
- Number of scheduler (warp/group) slots (not really)
- Synchronization

Outline

A Basic Processor: Closer to the Truth



- loosely based on Intel 8086
- What's a bus?

A Very Simple Program

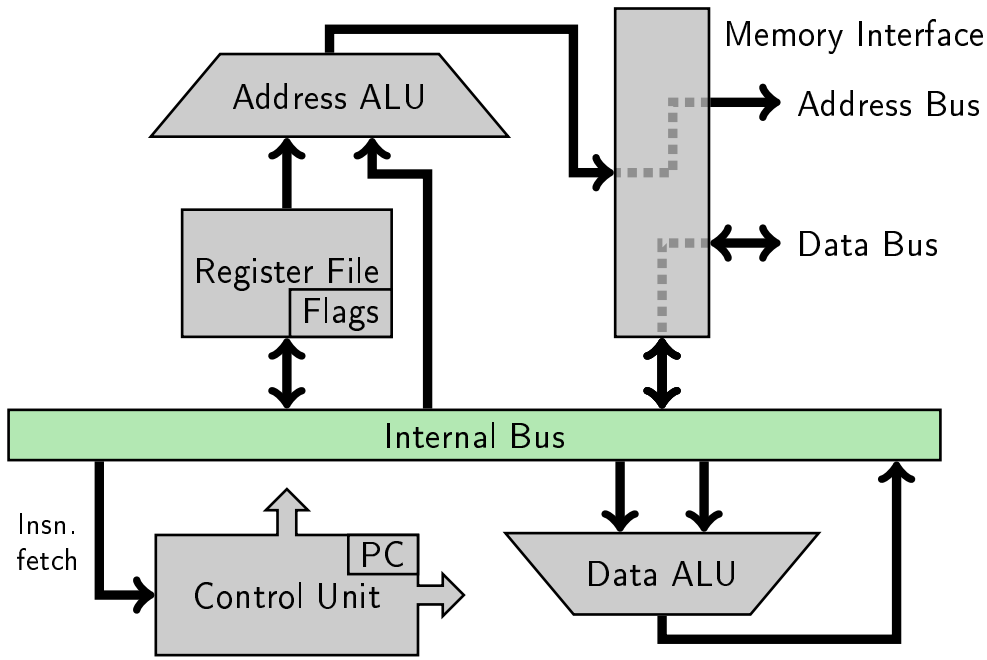
```
int a = 5;
int b = 17;
int z = a * b;
```

```
4:    c7 45 f4 05 00 00 00 movl   $0x5,-0xc(%rbp)
b:    c7 45 f8 11 00 00 00 movl   $0x11,-0x8(%rbp)
12:   8b 45 f4             mov    -0xc(%rbp),%eax
15:   0f af 45 f8          imul   -0x8(%rbp),%eax
19:   89 45 fc             mov    %eax,-0x4(%rbp)
1c:   8b 45 fc             mov    -0x4(%rbp),%eax
```

Things to know:

- Question: Which is it?
  - `<opcode> <src>, <dest>`
  - `<opcode> <dest>, <src>`
- Addressing modes (Immediate, Register, Base plus Offset)
- 0xHexadecimal

A Very Simple Program: Another Look




```
4:     c7 45 f4 05 00 00 00 movl   $0x5,-0xc(%rbp)
b:     c7 45 f8 11 00 00 00 movl   $0x11,-0x8(%rbp)
12:    8b 45 f4             mov    -0xc(%rbp),%eax
15:    0f af 45 f8          imul   -0x8(%rbp),%eax
19:    89 45 fc             mov    %eax,-0x4(%rbp)
1c:    8b 45 fc             mov    -0x4(%rbp),%eax
```

A Very Simple Program: Intel Form

```
4:    c7 45 f4 05 00 00 00    mov    DWORD PTR [rbp-0xc],0x5
b:    c7 45 f8 11 00 00 00    mov    DWORD PTR [rbp-0x8],0x1
12:   8b 45 f4                mov    eax,DWORD PTR [rbp-0xc]
15:   0f af 45 f8             imul   eax,DWORD PTR [rbp-0x8]
19:   89 45 fc                mov    DWORD PTR [rbp-0x4],eax
1c:   8b 45 fc                mov    eax,DWORD PTR [rbp-0x4]
```

- "Intel Form": (you might see this on the net)
  `<opcode> <sized dest>, <sized source>`
- Previous: "AT&T Form"
- Goal: Reading comprehension.
- Don't understand an opcode?
  https://en.wikipedia.org/wiki/X86_instruction_listings

Assembly Loops

```
int main()
{
  int y = 0, i;
  for (i = 0;
       y < 10; ++i)
    y += i;
  return y;
}
```

```
0:    55                          push   %rbp
1:    48 89 e5                    mov    %rsp,%rbp
4:    c7 45 f8 00 00 00 00        movl   $0x0,-0x8(%rbp)
b:    c7 45 fc 00 00 00 00        movl   $0x0,-0x4(%rbp)
12:   eb 0a                       jmp    1e <main+0x1e>
14:   8b 45 fc                    mov    -0x4(%rbp),%eax
17:   01 45 f8                    add    %eax,-0x8(%rbp)
1a:   83 45 fc 01                 addl   $0x1,-0x4(%rbp)
1e:   83 7d f8 09                 cmpl   $0x9,-0x8(%rbp)
22:   7e f0                       jle    14 <main+0x14>
24:   8b 45 f8                    mov    -0x8(%rbp),%eax
27:   c9                          leaveq
28:   c3                          retq
```

Things to know:

- Condition Codes (Flags): Zero, Sign, Carry, etc.
- Call Stack: Stack frame, stack pointer, base pointer
- ABI: Calling conventions

Demo Instructions: C → Assembly mapping from https://github.com/ynh/cpp-to-assembly

Demo: Assembly Reading Comprehension

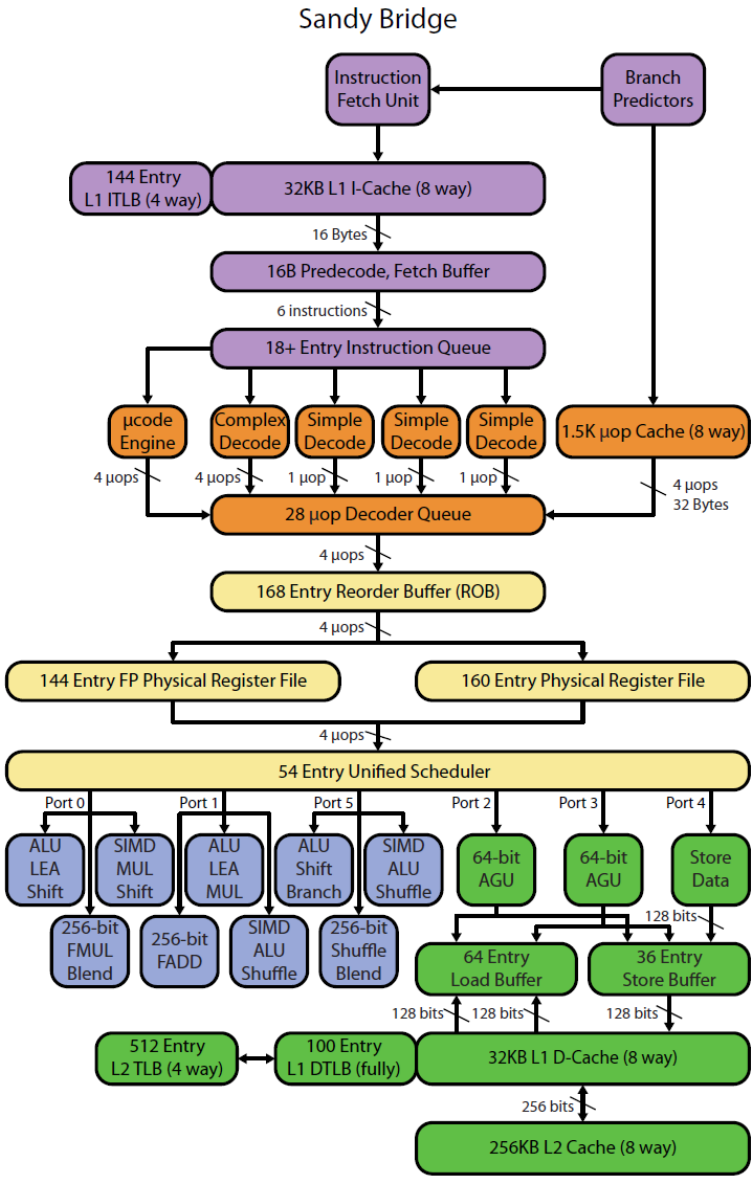
[DEMO: perf-02-assembly-reading]

Demo: Source-to-assembly mapping

Code to try:

```
int main()
{
  int y = 0, i;
  for (i = 0; y < 10; ++i)
    y += i;
  return y;
}
```

A Glimpse of a More Modern Processor



[David Kanter / Realworldtech.com]

PTX: Demo

[DEMO: perf-03-ptx-sass]
Nvidia PTX manual

SPIR-V

Currently: C (OpenCL C, GLSL, HLSL) used as intermediate representations to feed GPUs.

Downsides:

- Compiler heuristics may be focused on human-written code
- Parsing overhead (preprocessor!)
- C semantics may not match (too high-level)

SPIR-V:

- Goal: Common intermediate represenation ("IR") for all GPU-facing code (Vulkan, OpenCL)
- "Extended Instruction Sets":
  - General compute (OpenCL/CUDA) needs: pointers, special functions
- Different from "SPIR" (tweaked LLVM IR)

SPIR-V Example

```
          %2 = OpTypeVoid
          %3 = OpTypeFunction %2                              ; void ()
          %6 = OpTypeFloat 32                                 ; 32-bit float
          %7 = OpTypeVector %6 4                              ; vec4
          %8 = OpTypePointer Function %7                      ; function-local vec4*
         %10 = OpConstant %6 1
         %11 = OpConstant %6 2
         %12 = OpConstantComposite %7 %10 %10 %11 %10 ; vec4(1.0, 1.0, 2.0, 1.0
         %13 = OpTypeInt 32 0                                 ; 32-bit int, sign-less
         %14 = OpConstant %13 5
         %15 = OpTypeArray %7 %14
[...]
         %34 = OpLoad %7 %33
         %38 = OpAccessChain %37 %20 %35 %21 %36      ; s.v[2]
         %39 = OpLoad %7 %38
         %40 = OpFAdd %7 %34 %39
               OpStore %31 %40
               OpBranch %29
         %41 = OpLabel                                       ; else
         %43 = OpLoad %7 %42
         %44 = OpExtInst %7 %1 Sqrt %43                      ; extended instruction sq
         %45 = OpLoad %7 %9
         %46 = OpFMul %7 %44 %45
               OpStore %31 %46
```

Outline

Outline

The OpenCL model as a machine abstraction

Ideas:

- Abstract, n-dimensional index of cores
  - Limited communication/synchronization between cores
- Abstract, n-dimensional index of SIMD lanes with slightly more ability to communicate
  - Barriers and atomics
- Fairly implicit represenation of actual SIMD width

How would we achieve a more explicit representation of the hardware lane count?

> Use it as the length of the fastest-varying lane axis.

Intel SPMD Program Compiler (ISPC)

Goal: predictable vectorization of x86 code
Idea:

- Start from the CUDA/OpenCL model
- `taskIndex` for core index, `programIndex` for SIMD lane index
- `programIndex` is *precisely* the lane count (or 2x)
- Warn about code that gets scalarized
- `uniform` and `varying` types

[DEMO: lang-01-ispc]
https://ispc.github.io/

Outline

Loopy: a Code Generator for Computation with Arrays

Loopy is a code generator for computation with arrays.

Performance: human 'in the loop' for the foreseeable future.

- Capture math at a high level; target **number crunching**
- Progressively 'lower' through manual transformations
- Observe and control optimization steps
- 'Help me write the CUDA C/ISPC/... I would write'

Loopy: Program Representation

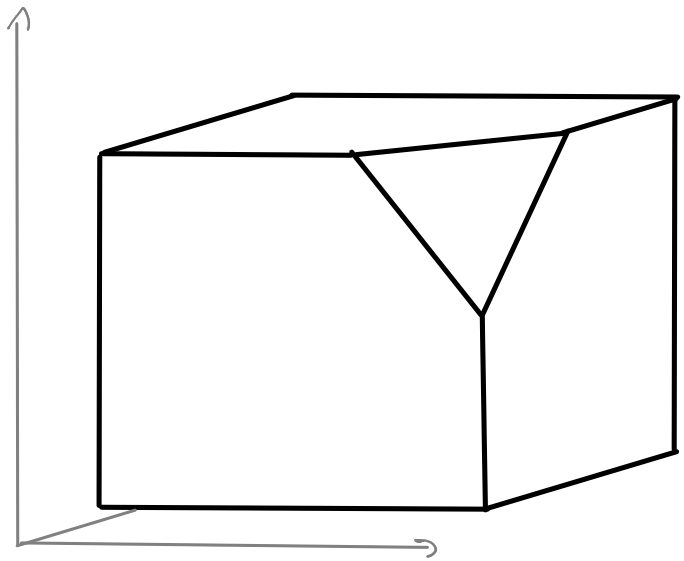
Primary design constraint:
Single Program Representation from UI to code gen

Must work for:

- Humans and Machines
- High Level and Fully Specified Hardware Mapping
- Static and (moderately) Data-Dependent Control Flow

Loopy: Program Representation

Polyhedron



```
{[i,j]:0 <= i,j < n and... }
```

```
b[i] = sum(j, A[i,j] * x[j])
```

Tree of Polyhedra

- (DAG of) Statements
- Per-loop 'mode' (seq/par)

= Semantics

Loopy: Execution and Transformation

Granularity: 'Kernel'

- May lower to multiple GPU 'kernels'
- One 'coherent computational step'

Transformations

```
knl = lp.split_iname(knl, "i", 16)
```

Loopy Demo

[DEMO: loopy-01-rank-one]

- *Seen:* Just-in-time mode in Python
- *Also possible:* Ahead-of-time mode from command line or Makefile

Kernel IR: Design Aspects

Criteria:

- Single shared medium across tools
- Shared medium between human and machine
- Ease of transformation
- Specified hardware mapping (no heuristics!)

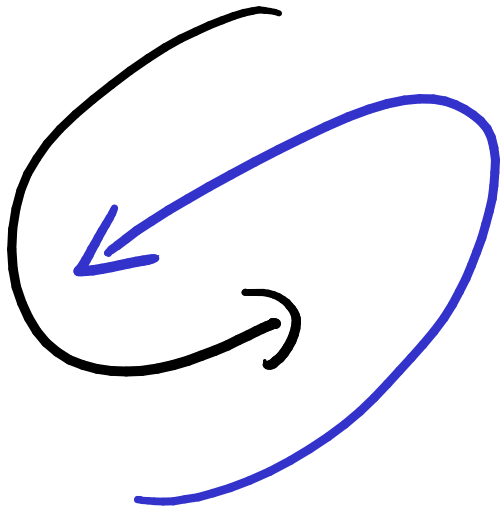
Other very recent IRs:

- C. Lattner, J. Pienaar "MLIR Primer: A Compiler Infrastructure for the End of Moore's Law." (2019).
- R. Baghdadi et al. "Tiramisu: A polyhedral compiler for expressing fast and portable code." Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization. IEEE Press, 2019.
- T. Ben-Nun et al. "Stateful Dataflow Multigraphs: A Data-Centric Model for High-Performance Parallel Programs.", SC '19.

More demos

- [DEMO: loopy-02-a-more-complex-code]
- [DEMO: loopy-03-fortran]
- [DEMO: loopy-04-data-layout]
- [DEMO: loopy-05-reduction]
- [DEMO: loopy-06-pde-to-code]

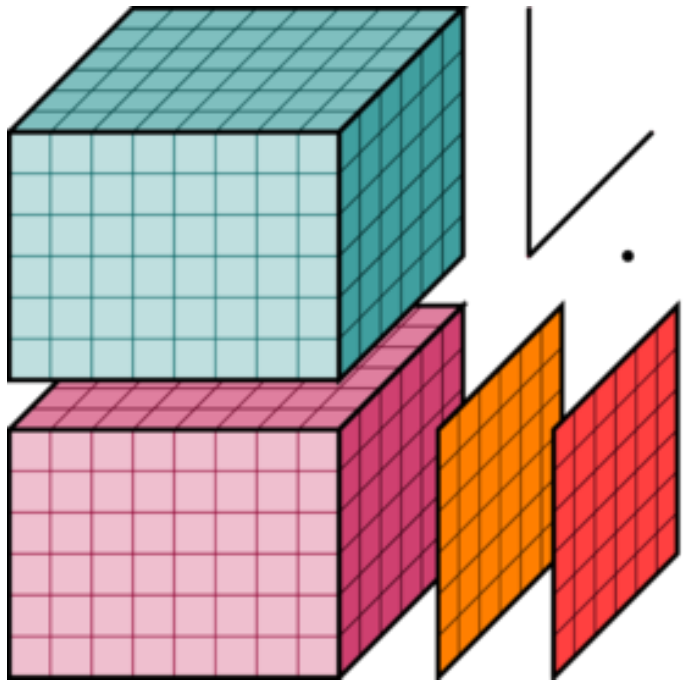
Basic Code Transforms



- Unroll
- Stride changes (Row/column/something major)
- Prefetch
- Precompute
- Tile
- Reorder loops
- Fix constants
- Parallelize (Thread/Workgroup)
- Affine map loop domains
- Texture-based data access
- Naming of array axes
- Loop collapse

Less Basic Code Transforms

- Kernel Fusion
- Splitting of Scans and Reductions
- Global Barrier by Kernel Fission
- Explicit-SIMD Vectorization
- Reuse of Temporary Storage
- SoA $\leftrightarrow$ AoS
- Buffering / Storage substitution
- Save flops using Distributive Law
- Arbitrary nesting of Data Layouts
- Realization of ILP



Further Features

- A-priori bounds checking
- Automatic Testing (against unopt. version)
- Symbolic operation counts
  - Flops
  - Memory access / Footprint size
  - Synchronization
- One Transformation Chain per Target Arch
- Script-Driven Transformation:
  - Share Transform Code
  - Build Transformation Abstractions
  - Build Simple Autotuners

Loopy: Example Users

- Firedrake finite element framework:
  https://arxiv.org/abs/1903.08243
- Dune PDElab finite element framework:
  http://arxiv.org/abs/1812.08075
- Pystella stencil-based cosmology solver:
  https://arxiv.org/abs/1909.12843,
  https://arxiv.org/abs/1909.12842
- Computational neuroscience:
  https://doi.org/10.3389/fninf.2018.00068
- SIMD/SIMT for chemical kinetics:
  https://doi.org/10.1016/j.combustflame.2018.09.008
- (My own numerics codes: Pytential, Grudge, Meshmode)

Conclusions: Loopy

- Goal: Allow near-peak performance
  (with some effort)
- 'Performance transparence'
  - Best if no further loop transforms are carried out
  - What is a good abstraction for the 'next layer down' from a tool like loopy?
- Common theme:
  Separation of concerns vs. Performance
- Human-in-the-loop seems unavoidable
  - Research question: What should the user interface to a compiler look like?

https://documen.tician.de/loopy

Play With Loopy Yourself

[DEMO: loopy-07-practice]

Outline

Defining an Expression DSL: Demo

- [DEMO: dsl-01-expression-trees]
- [DEMO: dsl-01-traversing-trees]
- [DEMO: dsl-03-defining-node-types]

https://documen.tician.de/pymbolic/