Languages and Abstractions for High-Performance Scientific Computing

CS598 APK

Andreas Kloeckner

Fall 2018

Outline

Outline

Outline

Why this class?

- Setting: Performance-Constrained Code
  When is a code performance-constrained?

  > A desirable quality (fidelity/capability) is limited by computational cost on a given computer.

- If your code is performance-constrained, what is the *best* approach?

  > Use a more efficient method/algorithm.

- If your code is performance-constrained, what is the *second-best* approach?

  > Ensure the current algorithm uses your computer efficiently. Observe that this is a *desperate measure*.

Examples of Performance-Constrained Codes

- Simulation codes
  - Weather/climate models
  - Oil/gas exploration
  - Electronic structure
  - Electromagnetic design
  - Aerodynamic design
  - Molecular dynamics / biological systems
  - Cryptanalysis
- Machine Learning
- Data Mining

Discussion:

- In what way are these codes constrained?
- How do these scale in terms of the problem size?

What Problem are we Trying To Solve?

$$(C_{ij})_{i,j=1}^{m,n} = \sum_{k=1}^{\ell} A_{ik} B_{kj}$$

Reference BLAS DGEMM code:
https://github.com/Reference-LAPACK/lapack/blob/master/BLAS/

OpenBLAS DGEMM code:
https://github.com/xianyi/OpenBLAS/blob/develop/kernel/x86_6

Demo: intro/DGEMM Performance

Demo Instructions: Compare OpenBLAS against Fortran BLAS on large square matrix

Goals: What are we Allowed to Ask For?

- Goal: "make efficient use of the machine"
- In general: not an easy question to answer
- In theory: limited by *some* peak machine throughput
  - Memory Access
  - Compute
- In practice: many other limits (Instruction cache, TLB, memory hierarchy, NUMA, registers)

Class web page

https://bit.ly/hpcabstr-f18

contains:

- Class outline
- Slides/demos/materials
- Assignments
- Virtual Machine Image
- Piazza
- Grading Policies
- Video
- HW1 (soon)

Welcome Survey

Please go to:

https://bit.ly/hpcabstr-f18

and click on 'Start Activity'.

If you are seeing this later, you can find the activity at Activity: welcome-survey.

Grading / Workload

Four components:

- Homework: 25%
- Paper Presentation: 25%
  - 30 minutes (two per class)
  - Presentation sessions scheduled throughout the semester
  - Paper list on web page
  - Sign-up survey: soon
- Paper Reactions: 10%
- Computational Project: 40%

Approaches to High Performance

- Libraries (seen)
- Black-box Optimizing Compilers
- Compilers with Directives
- Code Transform Systems
- "Active Libraries"

Q: Give examples of the latter two.

- Code Transform System: CHiLL
- Active Library: PyTorch

Libraries: A Case Study

$$(C_{ij})_{i,j=1}^{m,n} = \sum_{k=1}^{\ell} A_{ik} B_{kj}$$

Demo: intro/DGEMM Performance

Demo Instructions: Compare OpenBLAS on large square and small odd-shape matrices

Do Libraries Stand a Chance? (in general)

- Tremendously successful approach — Name some examples

  > (e.g.) LAPACK, Eigen, UMFPACK, FFTW, Numpy, Deal.ii

- Saw: Three simple integer parameters suffice to lose 'good' performance
  - Recent effort: "Batch BLAS" e.g. http://www.icl.utk.edu/files/publications/2017/icl-utk-1
- Separation of Concerns

  Example: Finite differences – e.g. implement ∂_x, ∂_y, ∂_z as separate (library) subroutines — What is the problem?

  > Data locality → data should be traversed once, ∂_x, ∂_y, ∂_z computed together
  > Separation of concerns → each operator traverses the data separately.

- Flexibility and composition

(Black-Box) Optimizing Compiler: Challenges

Why is black-box optimizing compilation so difficult?

- Application developer knowledge lost
  - Simple example: "Rough" matrix sizes
  - Data-dependent control flow
  - Data-dependent access patterns
  - Activities of other, possibly concurrent parts of the program
  - Profile-guided optimization can recover some knowledge
- Obtain proofs of required properties
- Size of the search space
  Consider
  http://polaris.cs.uiuc.edu/publications/padua.pdf

Directive-Based Compiler: Challenges

What is a directive-based compiler?

Demo Instructions: Show `l2dformta_qbx` from `pyfmmlib/vec_wrappers.f90`.

- Generally same as optimizing compiler
- Make use of extra promises made by the user
- What should the user promise?
- Ideally: feedback cycle between compiler and user
  - Often broken in both directions
  - User may not know what the compiler did
  - Compiler may not be able to express what it needs
- Directives: generally not mandatory

Lies, Lies Everywhere

- Semantics form a contract between programmer and language/environment
- Within those bounds, the implementation is free to do as it chooses
- True at every level:
  - Assembly
  - "High-level" language (C)

  Give examples of lies at these levels:

  - Assembly: Concurrent execution
  - "High-level" language (C): (e.g.) strength reduction, eliminated ops

One approach: *Lie to yourself*

- "Domain-specific languages" ← A fresh language, I can do what I want!
- Consistent semantics are notoriously hard to develop
  - Especially as soon as you start allowing subsets of even (e.g.) C's integers

Class Outline

High-level Sections:

- Intro, Armchair-level Computer Architecture
- Machine Abstractions
- Performance: Expectation, Experiment, Observation
- Programming Languages for Performance
- Program Representation and Optimization Strategies
- Code Generation/JIT

Survey: Class Makeup

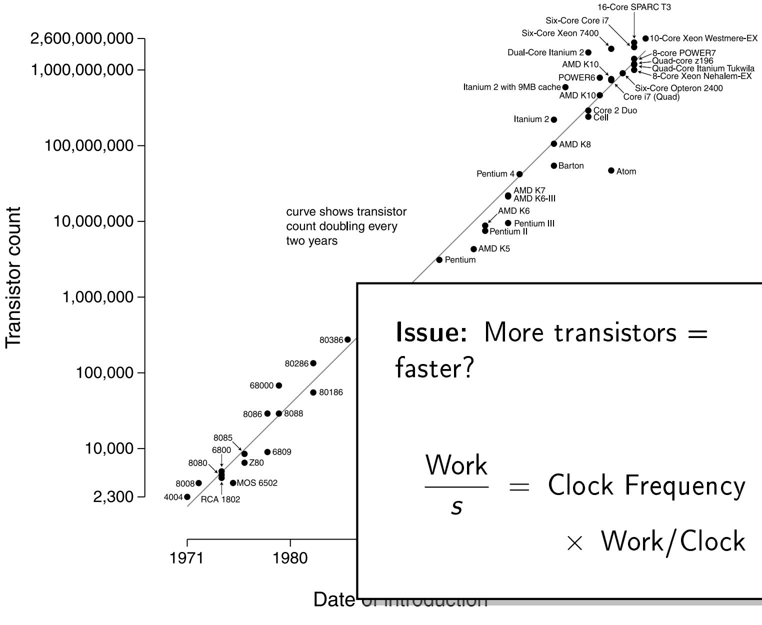
- Compiler class: 11 no, 3 yes
- HPC class: 10 yes, 4 no
- C: very proficient on average
- Python: proficient on average
- Assembly: some have experience
- GPU: Half the class has experience, some substantial
- CPU perf: Very proficient
- 10 PhD, 4 Masters, mostly CS (plus physics, CEE, MechSE)

Survey: Learning Goals

- How to use hardware efficiently to write fast code (1 response)
- I want to learn about commonly encountered problems in HPC and efficient ways to approach and solve them. (1 response)
- about writing high performance code for large scale problems. (1 response)
- more (and more) about high-performance computing beyond parallel programming. (1 response)
- This summer (while interning at Sandia national labs), I got familiar with GPU programming using Kokkos as the back end. I enjoyed this work immensely, and hope to continue learning about it, especially so that I can become better at writing GPU code myself. I am also interested in the relationship between a higher level abstraction (Kokkos), the compiler, and the actual compute device (GPU/CPU) relate together, and what tricks we have to help fix issues regarding this. For example, Kokkos uses a small amount of template metaprogramming to convert the source code into actual code. (1 response)
- Some GPU stuff, course description sounded interesting for my research in HPC/Parallel Computing. Would be interesting to look at different programming models or abstractions for HPC. (1 response)
- Getting better at doing high performance computing. (1 response)
- become more familiar with abstractions (1 response)
- I want to be able to auto generate performance portable C++ code, specifically for small batched tensor contractions. (1 response)
- Languages and abstractions for high-performance scientific computing (1 response)
- Investigating problems in high performance computing and looking for solutions, especially large-scale and using GPUs. (1 response)
- Better ways to efficiently (in terms of human time) write high-performance code that may be useful to/readable by others (1 response)
- about high-level languages and frameworks for high performance computing, the different interfaces they expose, compilation and runtime techniques they use, and the tradeoffs of these for an application developer. (1 response)

Outline

Moore's Law



Issue: More transistors = faster?

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

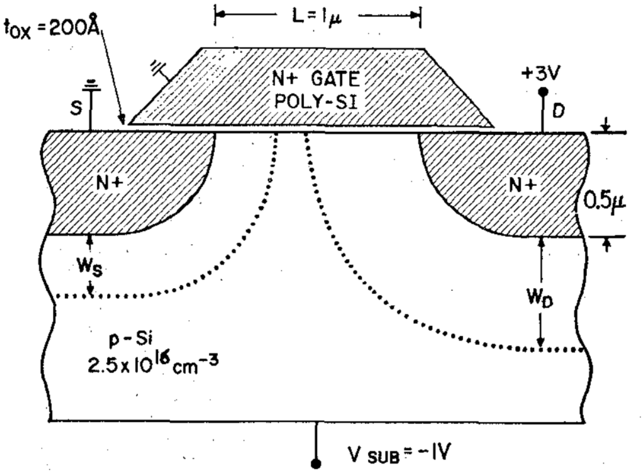
Dennard Scaling of MOSFETs

Parameter	Factor
Dimension	$1/\kappa$
Voltage	$1/\kappa$
Current	$1/\kappa$
Capacitance	$1/\kappa$
Delay Time	$1/\kappa$
Power dissipation/circuit	$1/\kappa^2$
Power density	1

[Dennard et al. '74, via Bohr '07]

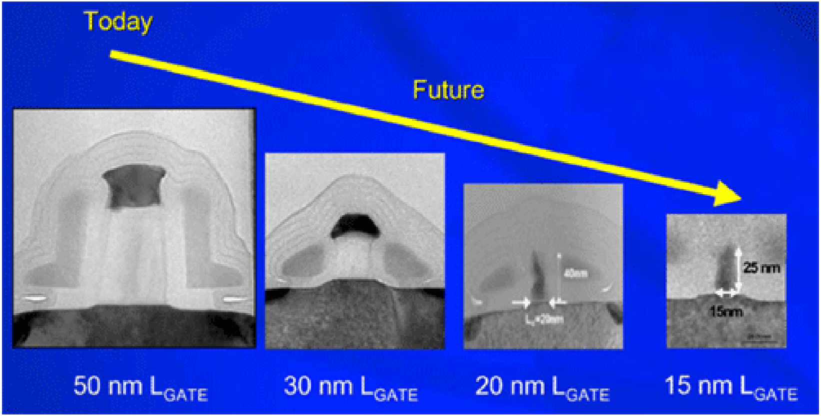
- Frequency = Delay time^{-1}

MOSFETs (“CMOS” – “complementary” MOS): Schematic



[Dennard et al. ‘74]

MOSFETs: Scaling



[Intel Corp.]

- 'New' problem at small scale:
  Sub-threshold leakage (due to low voltage, small structure)
  Dennard scaling is over – and has been for a while.

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...	4ish	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

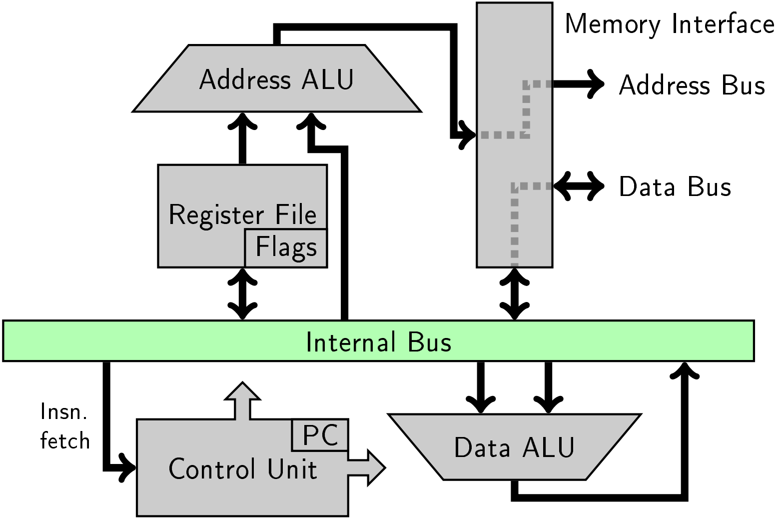
- Really: A crisis of the 'starts-at-the-top-ends-at-the-bottom' prorgramming model
- Tough luck: Most of our codes are written that way
- Even tougher luck: Everybody on the planet is *trained* to write codes this way

So:

- Need: Different tools/abstractions to write those codes

Outline

A Basic Processor: Closer to the Truth



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

A Very Simple Program

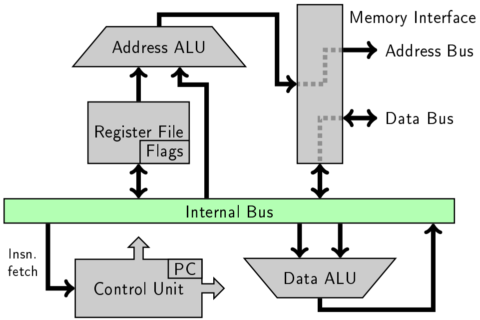
```
int a = 5;
int b = 17;
int z = a * b;
```

```
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": (we'll use this)
- Goal: Reading comprehension.
- Don't understand an opcode?
  https://en.wikipedia.org/wiki/X86_instruction_listings

Assembly Loops

```c
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

Demos

Demo: intro/Assembly Reading Comprehension

Demo: Source-to-assembly mapping
Code to try:

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

Outline

Modern Processors?

All of this can be built in about 4000 transistors.
(e.g. MOS 6502 in Apple II, Commodore 64, Atari 2600)

So what exactly are Intel/ARM/AMD/Nvidia doing with the other billions of transistors?

Execution in a Simple Processor

- [IF] Instruction fetch
- [ID] Instruction Decode
- [EX] Execution
- [MEM] Memory Read/Write
- [WB] Result Writeback

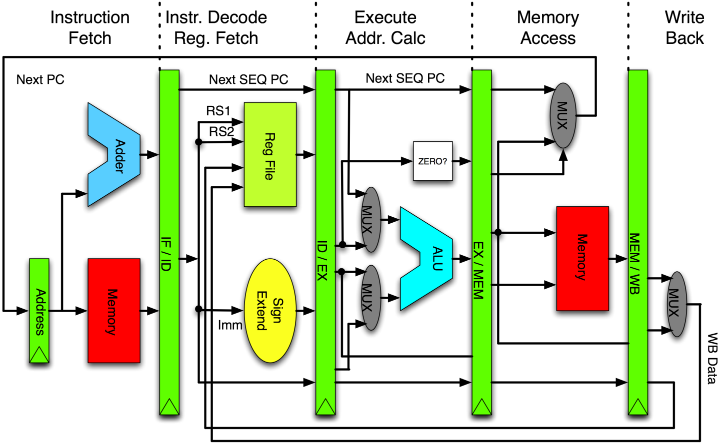
[Wikipedia ©]

Solution: Pipelining

	IF	ID	EX	MEM	WB				
$i \downarrow$		IF	ID	EX	MEM	WB			
$t \rightarrow$			IF	ID	EX	MEM	WB		
				IF	ID	EX	MEM	WB	
					IF	ID	EX	MEM	WB

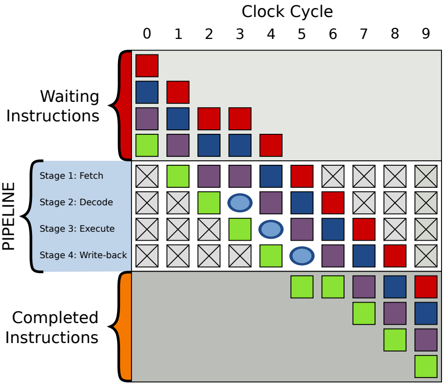
[Wikipedia ©]

MIPS Pipeline: 110,000 transistors



[Wikipedia ©]

Hazards and Bubbles



Q: Types of Pipeline Hazards? (aka: what can go wrong?)

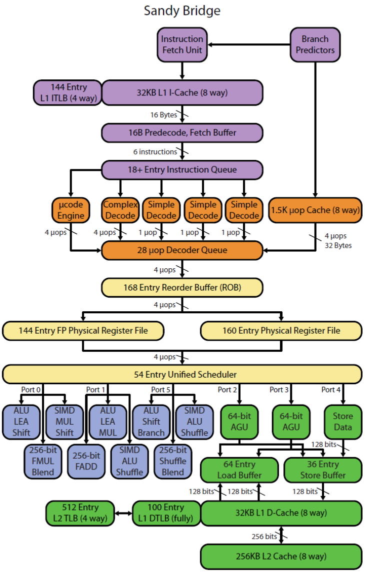
- Data
- Structural
- Control

Demo

Demo: intro/Pipeline Performance Mystery

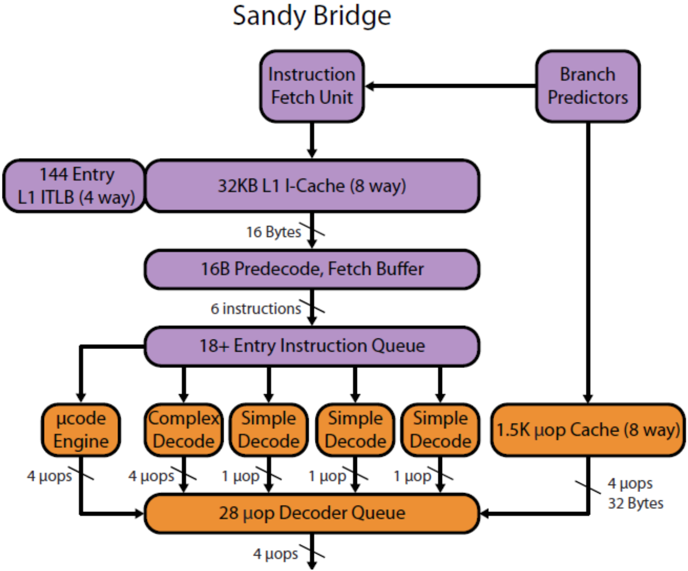
- a, a: elapsed time 3.83603 s
- a, b: elapsed time 2.58667 s
- a, a unrolled: elapsed time 3.83673 s
- aa, bb unrolled: elapsed time 1.92509 s
- a, b unrolled: elapsed time 1.92084 s

A Glimpse of a More Modern Processor



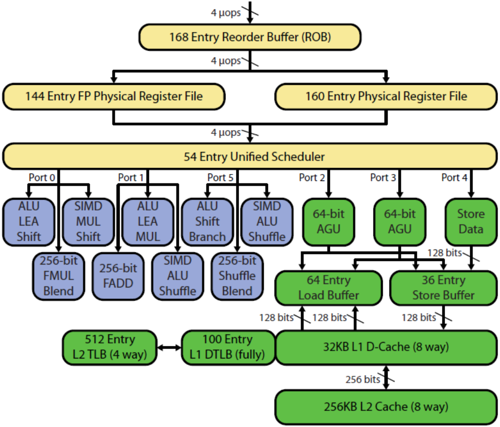
[David Kanter / Realworldtech.com]

A Glimpse of a More Modern Processor: Frontend



[David Kanter / Realworldtech.com]

A Glimpse of a More Modern Processor: Backend



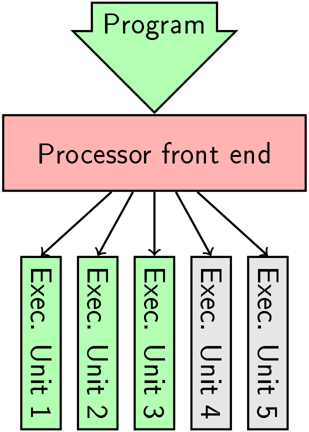
- New concept: Instruction-level parallelism ("ILP", "superscalar")
- Where does the IPC number from earlier come from?

[David Kanter / Realworldtech.com]

Demo

[Demo: intro/More Pipeline Mysteries]

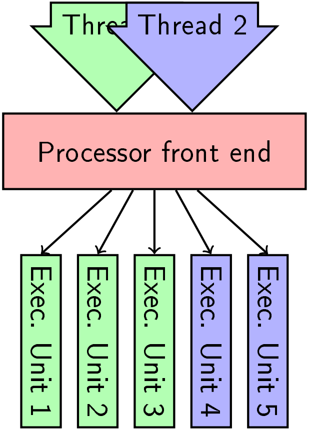
SMT/“Hyperthreading”



Q: Potential issues?

- $n\times$ the cache demand!
- Power?
- Some people just turn it off and manage their own ILP.

SMT/"Hyperthreading"



Q: Potential issues?

- $n\times$ the cache demand!
- Power?
- Some people just turn it off and manage their own ILP.

Outline

More Bad News from Dennard

Parameter	Factor
Dimension	$1/\kappa$
Line Resistance	κ
Voltage drop	κ
Response time	1
Current density	κ

[Dennard et al. '74, via Bohr '07]

- The above scaling law is for on-chip interconnects.
- Current $\sim$ Power *vs.* response time

Getting information from

- processor to memory
- one computer to the next

is

- slow (in *latency*)
- power-hungry

Somewhere Behind the Interconnect: Memory

Performance characteristics of memory:

- Bandwidth
- Latency

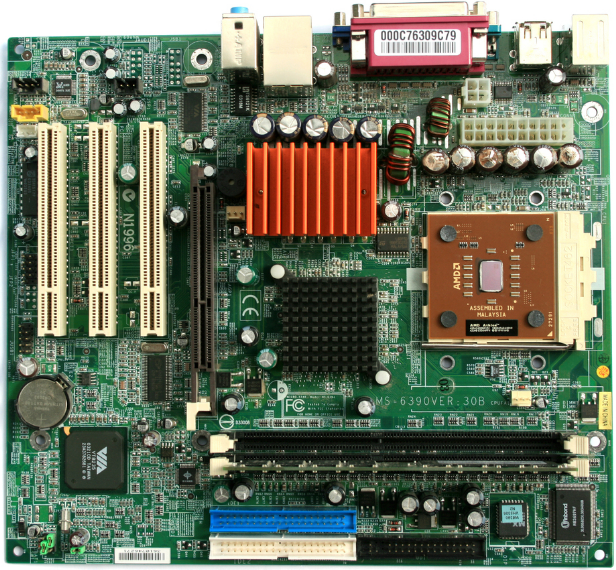
  Flops are cheap
  Bandwidth is money
  Latency is physics

  - *M. Hoemmen*

Minor addition (but important for us)?

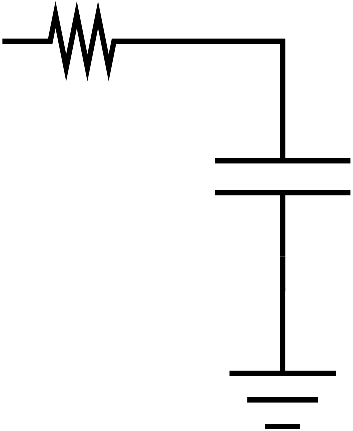
- Bandwidth is money and code structure

Latency is Physics: Distance

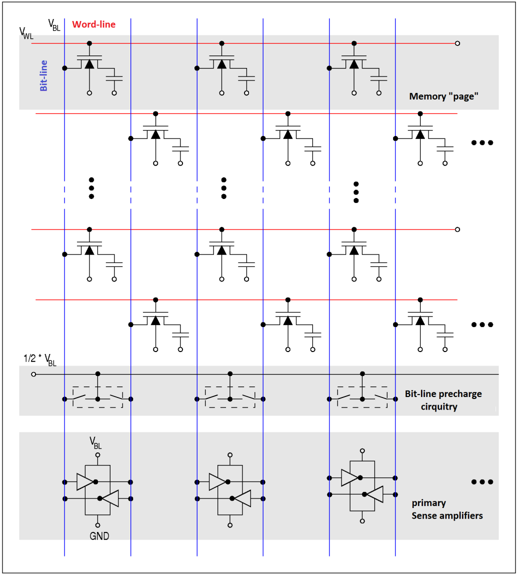


[Wikipedia (cc)]

Latency is Physics: Electrical Model



Latency is Physics: DRAM



[Wikipedia ©]

Latency is Physics: Performance Impact?

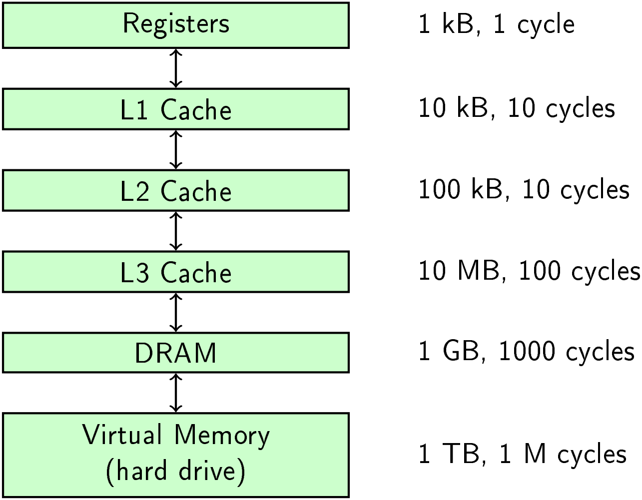
What is the performance impact of high memory latency?

> Processor stalled, waiting for data.

Idea:

- Put a look-up table of recently-used data onto the chip.
- Cache

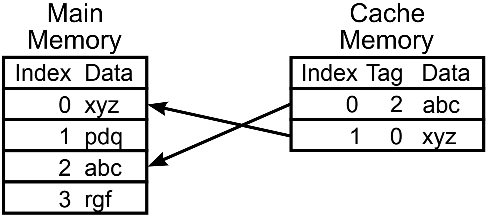
Memory Hierarchy



A Basic Cache

Demands on cache implementation:

- Fast, small, cheap, low power
- Fine-grained
- High "hit"-rate (few "misses")



Design Goals: at odds with each other. Why?

Address matching logic expensive

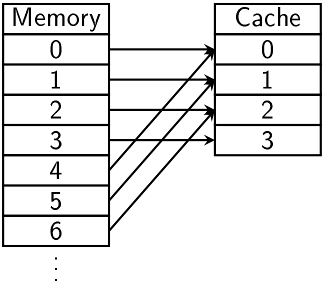
Caches: Engineering Trade-Offs

Engineering Decisions:

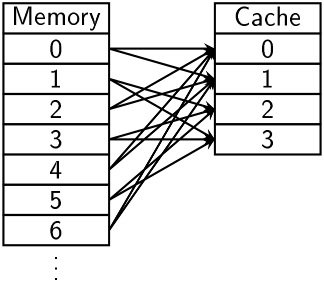
- More data per unit of access matching logic
  $\rightarrow$ Larger "Cache Lines"
- Simpler/less access matching logic
  $\rightarrow$ Less than full "Associativity"
- Eviction strategy
- Size

Associativity

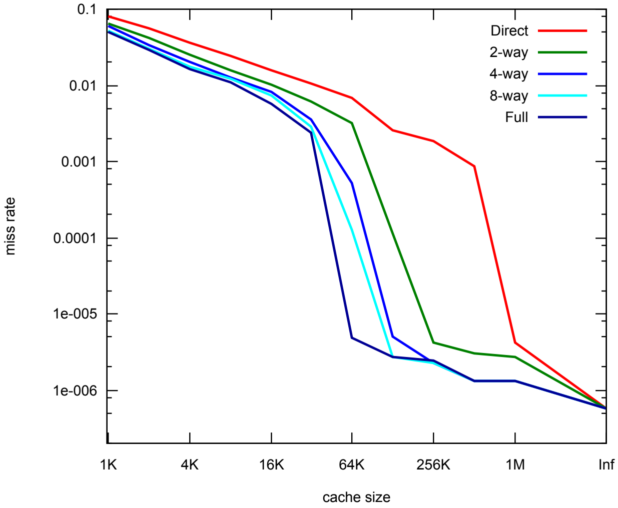
Direct Mapped:



2-way set associative:



Size/Associativity vs Hit Rate



Miss rate versus cache size on the Integer portion of SPEC CPU2000 [Cantin, Hill 2003]

Demo: Learning about Caches

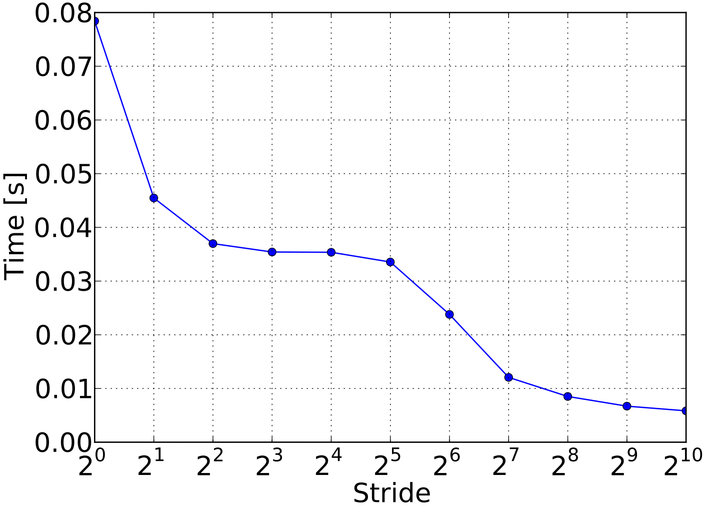
Demo: intro/Cache Organization on Your Machine

Experiments: 1. Strides: Setup

```
int go(unsigned count, unsigned stride)
{
  const unsigned array_size = 64 * 1024 * 1024;
  int *ary = (int *) malloc(sizeof(int) * array_size);

  for (unsigned it = 0; it < count; ++it)
  {
    for (unsigned i = 0; i < array_size; i += stride)
      ary[i] *= 17;
  }

  int result = 0;
  for (unsigned i = 0; i < array_size; ++i)
      result += ary[i];

  free(ary);
  return result;
}
```

What do you expect? [Ostrovsky '10]

Experiments: 1. Strides: Results

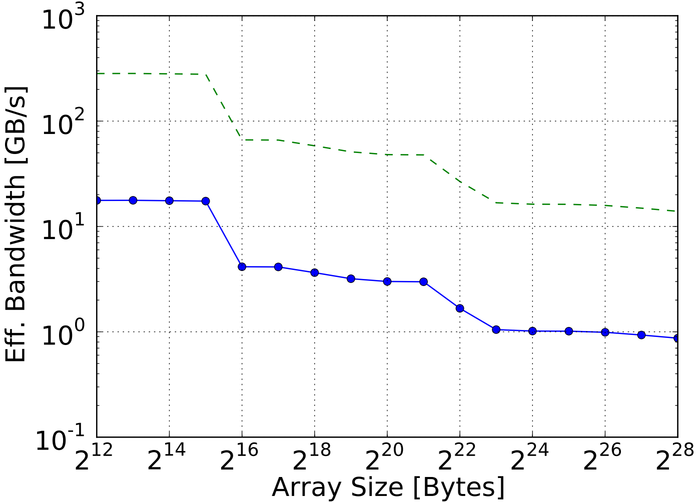


Experiments: 2. Bandwidth: Setup

```
int go(unsigned array_size, unsigned steps)
{
  int *ary = (int *) malloc(sizeof(int) * array_size);
  unsigned asm1 = array_size - 1;

  for (unsigned i = 0; i < 100*steps;)
  {
    #define ONE ary[(i++*16) & asm1] ++;
    #define FIVE ONE ONE ONE ONE ONE
    #define TEN FIVE FIVE
    #define FIFTY TEN TEN TEN TEN TEN
    #define HUNDRED FIFTY FIFTY
    HUNDRED
  }

  int result = 0;
  for (unsigned i = 0; i < array_size; ++i)
      result += ary[i];

  free(ary);
  return result;
}
```

What do you expect? [Ostrovsky '10]

Experiments: 2. Bandwidth: Results

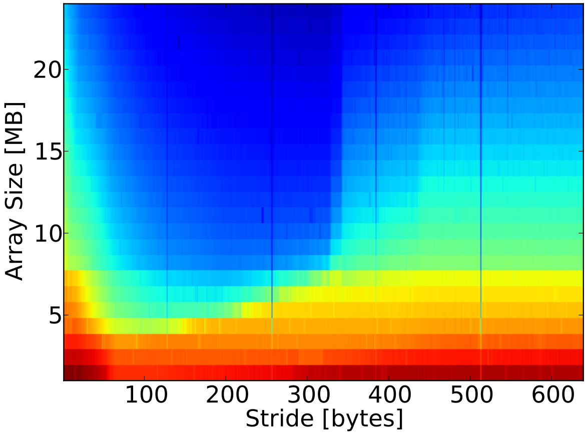


Experiments: 3. A Mystery: Setup

```c
int go(unsigned array_size, unsigned stride, unsigned steps)
{
  char *ary = (char *) malloc(sizeof(int) * array_size);

  unsigned p = 0;
  for (unsigned i = 0; i < steps; ++i)
  {
    ary[p] ++;
    p += stride;
    if (p >= array_size)
      p = 0;
  }

  int result = 0;
  for (unsigned i = 0; i < array_size; ++i)
      result += ary[i];

  free(ary);
  return result;
}
```

What do you expect? [Ostrovsky '10]

Experiments: 3. A Mystery: Results



Color represents achieved bandwith:

- Red: high
- Blue: low

Thinking about the Memory Hierarchy

- What is a working set?
- What is data locality of an algorithm?
- What does this have to with caches?

Case Study: Streaming Workloads

Q: Estimate epxected throughput for `saxpy` on an architecture with caches. What are the right units?

$$z_i = \alpha x_i + y_i \quad (i = 1, \ldots, n)$$

- Units: GBytes/s
- Net memory accessed: $n \times 4 \times 3$ bytes
- Actual memory accessed: $n \times 4 \times 4$ bytes
  (To read z read into the cache before modification)

Demo: https://github.com/lcw/stream_ispc

Special Store Instructions

At least two aspects to keep apart:

- Temporal Locality: Are we likely to refer to this data again soon? (*non-temporal* store)
- Spatial Locality: Will (e.g.) the entire cache line be overwritten? (*streaming* store)

What hardware behavior might result from these aspects?

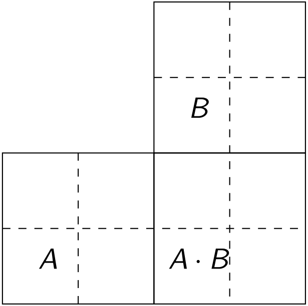
- Non-temporal: Write past cache entirely (/invalidate), or evict soon
- Spatial: Do not fetch cache line before overwriting

- Comment on what a compiler can promise on these aspects.
- Might these 'flags' apply to loads/prefetches?

(see also: [McCalpin '18])

Case study: Matrix-Matrix Mult. ('MMM'): Code Structure

- How would you structure a high-performance MMM?
- What are sources of concurrency?
- What should you consider your working set?



- Sources of concurrency: row, column loop, summation loop (?)
- Working set: artificially created blocks
- Provide enough concurrency: SIMD, ILP, Core

Case study: Matrix-Matrix Mult. ('MMM') via Latency

Come up with a simple cost model for MMM in a two-level hierarchy based on latency:

$$\begin{aligned}\text{Avg latency per access} =& \\ &(1 - \text{Miss ratio}) \cdot \text{Cache Latency} \\ &\quad + \text{Miss ratio} \cdot \text{Mem Latency}\end{aligned}$$

Assume: Working set fits in cache, No conflict misses

Calculation:

- Total accesses: $4N_B^3$ (N_B: block size)
- Misses: $3N_B^2$
- Miss rate:

$$\frac{3}{4N_B \cdot \text{cache line size}}$$

[Yotov et al. '07]

Case study: Matrix-Matrix Mult. ('MMM') via Bandwidth

Come up with a cost model for MMM in a two-level hierarchy based on bandwidth:

- FMA throughput: 16×2 SP FMAs per clock (e.g.)
- Cycle count: $2N^3/(2 \cdot 32) = N^3/32$
- Required cache bandwidth: (words accessed)/(cycles) $= 4N^3/(N^3/32) = 128$ floats/cycle (GB/s?)
- Total mem. data motion: $\#$ blocks $\cdot\, 4 \cdot$ (block size) $= (N/N_B)^3 \cdot 4N_B^2 = 4N^3/N_B$
- Required mem. bandwidth: (Mem.motion)/(cycles) $=$ $4N^3/N_B/(N^3/32) = 128/N_B$ floats/cycle (GB/s?)
- What size cache do we need to get to feasible memory bandwidth?

[Yotov et al. '07]

Case study: Matrix-Matrix Mult. ('MMM'): Discussion

Discussion: What are the main simplifications in each model?

Bandwidth:

- Miss assumptions
- Multiple cache levels
- Latency effects

Latency:

- Miss assumptions
- Concurrency/parallelism of memory accesses
- (HW) prefetching
- Machine Limits

[Yotov et al. '07]

General Q: How can we analyze cache cost of algorithms in general?

Hong/Kung: Red/Blue Pebble Game

Simple means of I/O cost analysis: “Red/blue pebble game”

- A way to quantify I/O cost on a DAG (why a DAG?)
- “Red Hot” pebbles: data that can be computed on
- “Blue Cool” pebbles: data that is stored, but not available for computation without I/O

Note: Can allow “Red/Purple/Blue/Black”: more levels

Q: What are the cost metrics in this model?

- I/O Cost: Turn a red into a blue pebble and vice versa
- Number of red pebbles (corresponding to size of 'near' storage)

[Hong/Kung '81]

Cache-Oblivious Algorithms

Annoying chore: Have to pick multiple machine-adapted block sizes in cache-adapted algorithms, one for each level in the memory hierarchy, starting with registers.

Idea:

- Step 1: Express algorithm recursively in divide & conquer manner
- Step 2: Pick a strategy to decrease block size
  Give examples of block size strategies, e.g. for MMM:

  - All dimensions
  - Largest dimension

Result:

- Asymptotically optimal on Hong/Kung metric

Cache-Oblivious Algorithms: Issues

What are potential issues on actual hardware?

- In pure form:
  - Function call overhead
  - Register allocation
- With good base case:
  - I-cache overflow
  - Instruction scheduling

[Yotov et al. '07]

Recall: Big-O Notation

Classical Analysis of Algorithms (e.g.):

$$\mathrm{Cost}(n) = O(n^3).$$

Precise meaning? Anything missing from that statement?

Missing: 'as $n \to \infty$'

There exists a C and an N_0 independent of n so that for all $n \geq N_0$,

$$\mathrm{Cost}(n) \leq C \cdot n^3.$$

Comment: "Asymptotically Optimal"

Comments on asymptotic statements about cost in relation to high performance?

- No statement about finite n
- No statement about the constant

Net effect: Having an understanding of asymptotic cost is *necessary*, but *not sufficient* for high performance.

HPC is in the business of minimizing C in:

$$\text{Cost}(n) \leq C \cdot n^3 \quad (\text{for all } n)$$

Alignment

Alignment describes the process of matching the base address of:

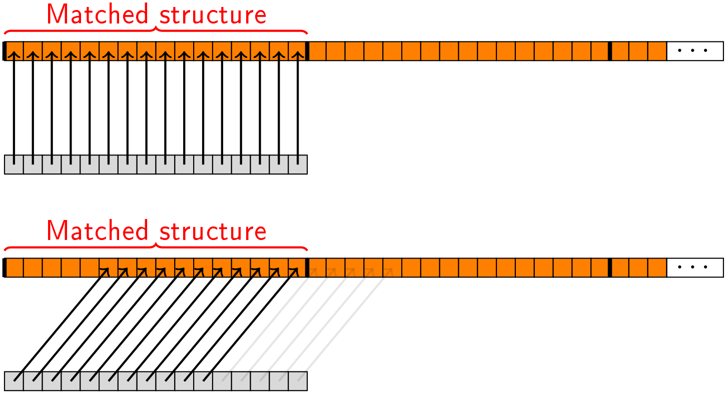
- Single word: `double`, `float`
- SIMD vector
- Larger structure

To machine granularities:

- Natural word size
- Vector size
- Cache line

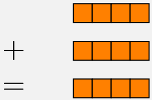
Q: What is the performance impact of misalignment?

Performance Impact of Misalignment



SIMD: Basic Idea

What's the basic idea behind SIMD?



What architectural need does it satisfy?

- Insufficient instruction decode/dispatch bandwidth
- Tack more operations onto one decoded instruction

Typically characterized by width of data path:

- SSE: 128 bit (4 floats, 2 doubles)
- AVX-2: 256 bit (8 floats, 4 doubles)
- AVX-512: 512 bit (16 floats, 8 doubles)

SIMD: Architectural Issues

Realization of inter-lane comm. in SIMD? Find instructions.

- Misaligned stores/loads? (no)
- Broadcast, Unpack+Interleave, Shuffle, Permute
- Reductions (“horizontal”)

Name tricky/slow aspects in terms of expressing SIMD:

- Divergent control flow
  - Masking
  - Reconvergence
- Indirect adressing: gather/scatter

x86 SIMD suffixes: What does the “`ps`” suffix mean? “`sd`”?

- `ps`: Packed single precision
- `sd`: Scalar double precision

SIMD: Transposes

Why are transposes important? Where do they occur?

- Whenever SIMD encounters a mismatched data layout
- For example: MMM of two row-major matrices

Example implementation aspects:

- HPTT: [Springer et al. '17]
- github: springer13/hptt 8x8 transpose microkernel
- Q: Why 8x8?

Outline

Multiple Cores vs Bandwidth

Assume (roughly right for Intel):

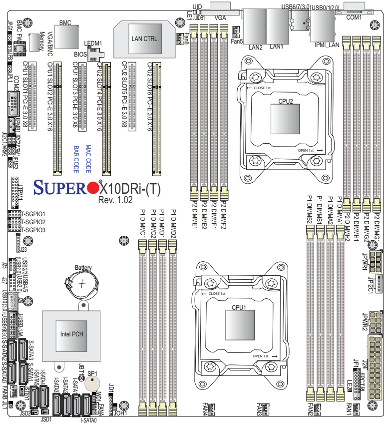
- memory latency of 100 ns
- peak DRAM bandwidth of 50 GB/s (per socket)

How many cache lines should be/are in flight at one time?

- $100\text{ns} \cdot 50\text{GB}/s = 5000\text{bytes}$
- About 80 cache lines
- Oops: Intel hardware can only handle about 10 pending requests per core at one time
- $10 \cdot 64/100\text{ns} \approx 6.4\text{GB}/s$

[McCalpin '18]

Topology and NUMA



[SuperMicro Inc. ‘15]

Demo: Show `lstopo` on porter, from hwloc.

Placement and Pinning

Who decides on what core my code runs? How?

- The OS scheduler: “Oh, hey, look! A free core!”
- You, explicitly, by pinning:
  - `OMP_PLACES=cores`
  - `pthread_setaffinity_np()`

Who decides on what NUMA node memory is allocated?

- `malloc` uses 'first touch'
- *You* can decide explicitly (through `libnuma`)

Demo: intro/NUMA and Bandwidths

What is the main expense in NUMA?

Latency (but it impacts bandwidth by way of queuing)

Cache Coherence

What is *cache coherence*?

- As soon as you make a copy of (cache) something, you risk inconsistency with the original
- A set of guarantees on how (and in what order) changes to memory become visible to other cores

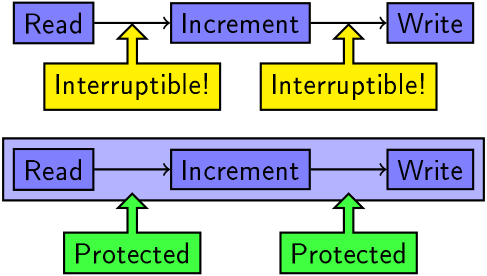
How is cache coherence implemented?

- Snooping
- Protocols, operating on cache line states (e.g. "MESI")

What are the performance impacts?

- Demo: intro/Threads vs Cache
- Demo: intro/Lock Contention

'Conventional' vs Atomic Memory Update



Outline

Outline

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

Example: A Semaphore With Atomics

```
#include <stdatomic.h> // mo_->memory_order, a_->atomic
typedef struct { atomic_int v;} naive_sem_t;
void sem_down(naive_sem_t *s)
{
  while (1) {
    while (a_load_explicit(&(s->v), mo_acquire) < 1)
        spinloop_body();
    int tmp=a_fetch_add_explicit(&(s->v), -1, mo_acq_rel
    if (tmp >= 1)
        break; // we got the lock
    else // undo our attempt
      a_fetch_add_explicit(&(s->v), 1, mo_relaxed);
  }
}
void sem_up(naive_s_t *s) {
  a_fetch_add_explicit(&(s->v), 1, mo_release);
}
```

[Cordes '16] — Hardware implementation: how?

C: What is 'order'?

C11 Committee Draft, December '10, Sec. 5.1.2.3, "Program execution":

- (3) *Sequenced before* is an asymmetric, transitive, pair-wise relation between evaluations executed by a single thread, which induces a partial order among those evaluations. Given any two evaluations A and B, if A is sequenced before B, then the execution of A shall precede the execution of B. (Conversely, if A is sequenced before B, then B is sequenced after A.) If A is not sequenced before or after B, then A and B are unsequenced. Evaluations A and B are *indeterminately sequenced* when A is sequenced either before or after B, but it is unspecified which. The presence of a sequence point between the evaluation of expressions A and B implies that every value computation and side effect associated with A is sequenced before every value computation and side effect associated with B. (A summary of the *sequence points* is given in annex C.)

Q: Where is this definition used (in the standard document)?

> In defining the semantics of atomic operations.

C: What is 'order'? (Encore)

C11 Draft, 5.1.2.4 "Multi-threaded executions and data races":

- All modifications to a particular atomic object M occur in some particular total order, called the *modification order* of M.
- An evaluation A *carries a dependency* to an evaluation B if ...
- An evaluation A is *dependency-ordered* before an evaluation B if...
- An evaluation A *inter-thread happens before* an evaluation B if...
- An evaluation A *happens before* an evaluation B if...

Why is this so subtle?

- Many common optimizations depend on the ability to reorder operations.
- Two options:
  1. Lose the ability to do those optimizations
  2. Specify precisely how much of the order should be externally observable

C: How Much Lying is OK?

C11 Committee Draft, December '10, Sec. 5.1.2.3, "Program execution":

- (1) The semantic descriptions in this International Standard describe the behavior of an abstract machine in which issues of optimization are irrelevant.
- (2) Accessing a volatile object, modifying an object, modifying a file, or calling a function that does any of those operations are all *side effects*, which are changes in the state of the execution environment. [. . .]

C: How Much Lying is OK?

- (4) In the abstract machine, all expressions are evaluated as specified by the semantics. An actual implementation need not evaluate part of an expression if it can deduce that its value is not used and that no needed side effects are produced (including any caused by calling a function or accessing a volatile object).
- (6) The least requirements on a conforming implementation are:
  - Accesses to volatile objects are evaluated strictly according to the rules of the abstract machine.
  - At program termination, all data written into files shall be identical to the result that execution of the program according to the abstract semantics would have produced.
  - The input and output dynamics of interactive devices shall take place as specified in 7.21.3. The intent of these requirements is that unbuffered or line-buffered output appear as soon as possible, to ensure that prompting messages actually appear prior to a program waiting for input.

  This is the observable behavior of the program.

Arrays

Why are arrays the dominant data structure in high-performance code?

- Performance is mostly achieved with *regular*, *structured* code (e.g. SIMD, rectangular loops)
- Arrays are a natural fit for that type of code
- Abstractions of linear algebra map directly onto arrays

Any comments on C's arrays?

- 1D arrays: fine, no surprises
- *n*D arrays: basically useless: sizes baked into types
  - Interestinglly: Fortran is (incrementally) smarter

Arrays vs Abstraction

Arrays-of-Structures or Structures-of-Arrays? What's the difference? Give an example.

- Example: Array of XYZ coordinates:
  - XYZXYZXYZ...
  - XXX....YYY...ZZZ...
- Which of these will be suitable for SIMD? (e.g. computing a norm?)
- Structures-of-Arrays if at all possible – to expose regularity

Language aspects of the distinction? Salient example?

- C `struct` forces you into arrays-of-structures
  - AoS: more "conceptually sound"
  - SoA: better for performance
- Complex numbers

C and Multi-Dimensional Arrays: A Saving Grace

```c
// YES:
void f(int m, int n, double (*)[m][n]);

// NO:
struct ary {
  int m;
  int n;
  double (*array)[m][n];
};

// YES:
struct ary {
  int m;
  int n;
  double a[];
};
```

SIMD

Name language mechanisms for SIMD:

- Inline Assembly
- Intrinsics
- Vector Types `typedef int v4si __attribute__ ((vector_size (16)));`
- `#pragma simd`
- Merging of scalar program instances (in hw/sw)

Demo: machabstr/Ways to SIMD

Outer-Loop/inner-Loop Vectorization

Contrast *outer-loop* vs *inner-loop vectorization*.

- Inner-loop: Inner-most loop vectorized
- Outer loop: Vectorize a whole kernel. Requires:
  - Changed memory layout
  - Must be able to express *all* control flow

Side q: Would you consider GPUs outer- or inner-loop-vectorizing?

Alignment: How?

The old way:

```
int __attribute__ ((aligned (8))) a_int;
```

Difference between these two?

```
int __attribute__ ((aligned (8))) * ptr_t_1;
int *__attribute__ ((aligned (8))) ptr_t_2;
```

The 'new' way (C/C++11):

```
struct alignas(64) somestruct_t { /* ... */ };
struct alignas(alignof(other_t))
  somestruct_t { /* ... */ };
struct
  alignas(
    std::hardware_destructive_interference_size)
    somestruct_t { /* ... */ };
```

What is *constructive interference*?

Alignment: Why?

What is the concrete impact of the constructs on the previous slide?

- Compiler needs to *know* whether data is aligned
  - Generate the correct instructions (which encode alignment promises)
  - Stack-allocate memory of the correct alignment
- Heap-allocated memory needs to actually satisfy the alignment promise!
  - `posix_memalign`
  - Hack it by overallocating
  - In `numpy`: overallocate in bytes, get base address, offset, obtain view

Pointers and Aliasing

Demo: machabstr/Pointer Aliasing

Register Pressure

What if the register working set gets larger than the registers can hold? What is the performance impact?

- "Register Spill": save/reload code being generated
- CPU: L1 is relatively fast
- Other architectures: can be quite dramatic

Demo: machabstr/Register Pressure

Object-Oriented Programming

Object-oriented programming: The weapon of choice for encapsulation and separation of concerns!
Performance perspective on OOP?

- Fine-grain OOP leads to an AoS disaster
- Long expressions create many temporaries
  - Memory traffic
- Return values
- Run-time polymorphism (`virtual` methods) lead to fine-grain flow control

Summary: No good, very bad. *Must* have sufficient granularity to offset cost.

Demo: machabstr/Object Orientation vs Performance

Being Nice to Your Compiler

Some rules of thumb:

- Use indices rather than pointers
- Extract common subexpressions
- Make functions static
- Use `const`
- Avoid store-to-load dependencies

What are the concrete impacts of doing these things?

Outline

Introduction

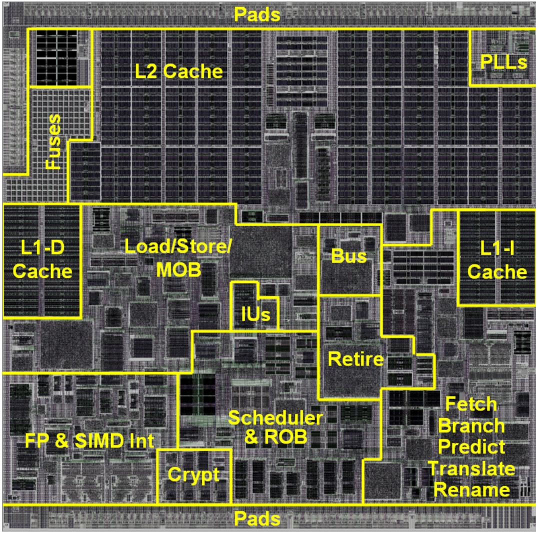
Machine Abstractions
- C
- OpenCL/CUDA
- Convergence, Differences in Machine Mapping
- Lower-Level Abstractions: SPIR-V, PTX

Performance: Expectation, Experiment, Observation

Performance-Oriented Languages and Abstractions

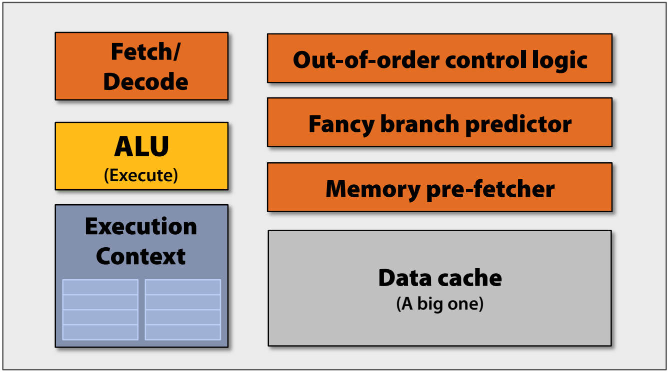
Polyhedral Representation and Transformation

Chip Real Estate



Die floorplan: VIA Isaiah (2008).
65 nm, 4 SP ops at a time, 1 MiB L2.

"CPU-style" Cores



[Fatahalian '08]

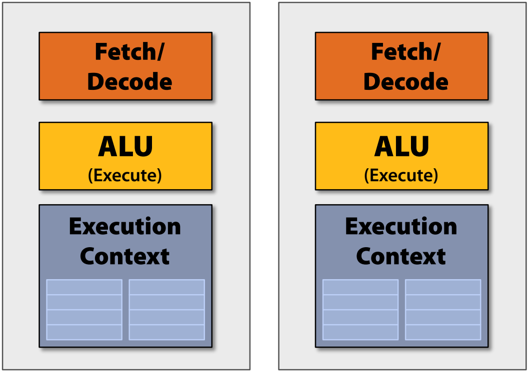
Slimming down

Idea #1:

Remove components that help a single instruction stream run fast

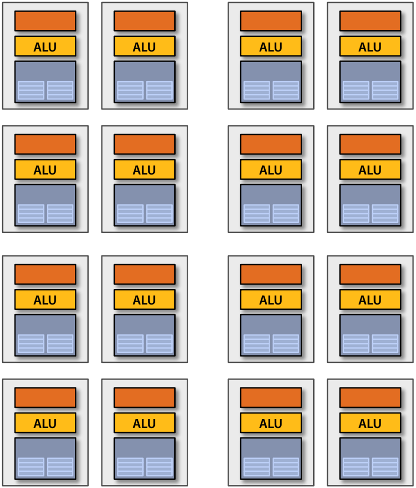
[Fatahalian '08]

More Space: Double the Number of Cores



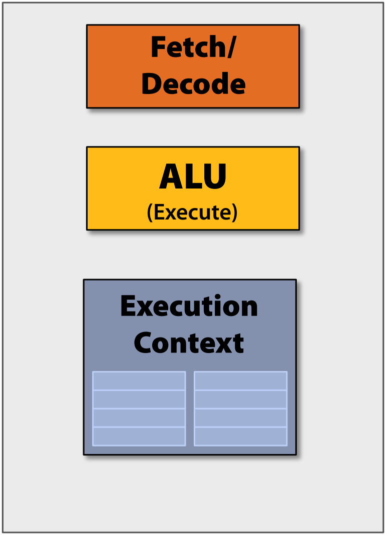
[Fatahalian '08]

Even more



[Fatahalian '08]

SIMD

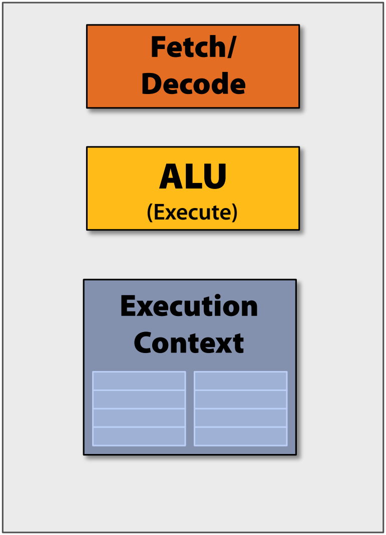


[Fatahalian '08]

Idea #2: SIMD

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

SIMD

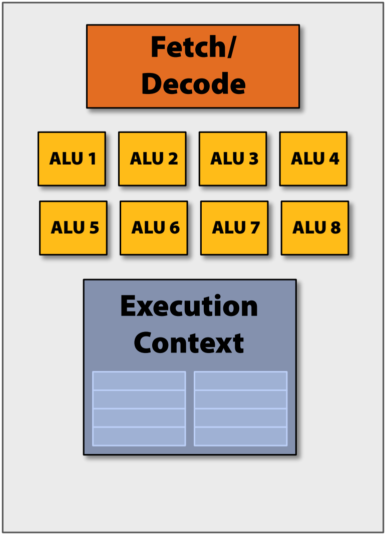


[Fatahalian '08]

Idea #2: SIMD

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

SIMD

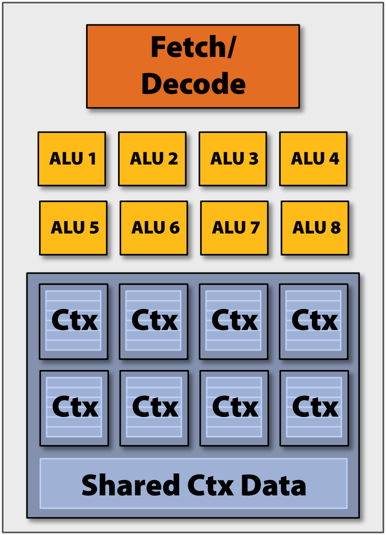


[Fatahalian '08]

Idea #2: SIMD

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

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?

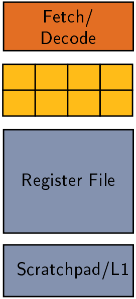
> A SIMD vector ('warp' (Nvidia), 'Wave-front' (AMD))

How can we keep busy?

> - More vectors (bigger group)
> - ILP

Change in architectural picture?

Before:



After:

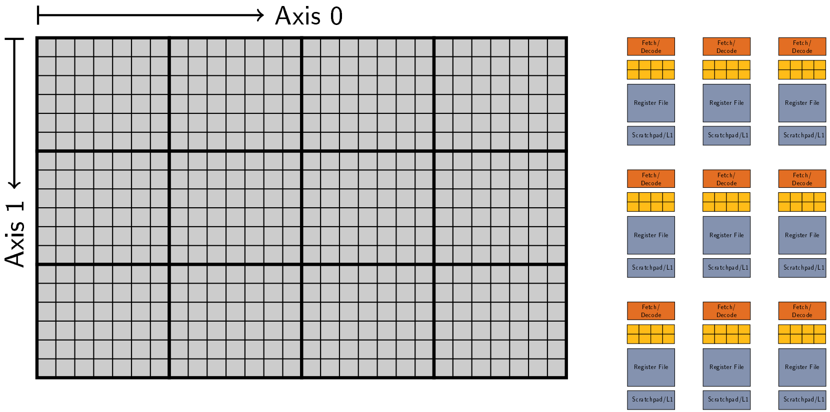
More state space!

GPUs: Core Architecture Ideas

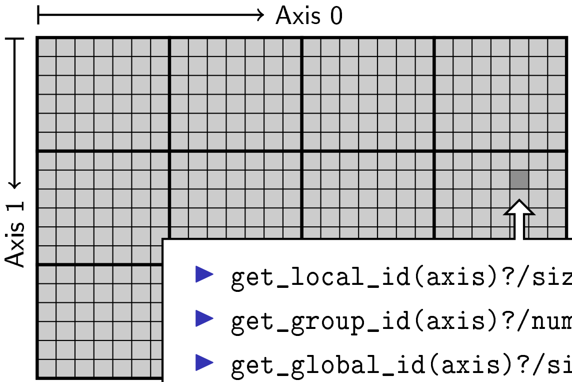
Three core ideas:

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

'SIMT'



Wrangling the Grid



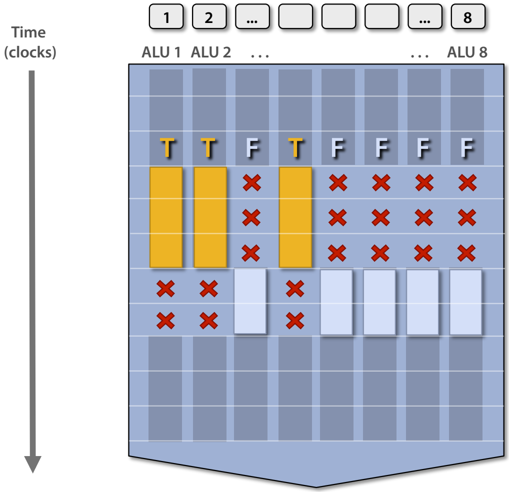
- get_local_id(axis)?/size(axis)?
- get_group_id(axis)?/num_groups(axis)?
- get_global_id(axis)?/size(axis)?

axis=0,1,2,...

Demo CL code

Demo: machabstr/Hello GPU

'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]

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

GPU Program 'Scopes'

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

GPU: Communication

What forms of communication exist at each scope?

- Subgroup: Shuffles (!)
- Workgroup: Scratchpad + barrier, local atomics + mem fence
- Grid: Global atomics

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

- Indepdendent forward progress of all threads is not guaranteed: no.
  (true until recently)
- But: Device partitioning can help!

GPU Programming Model: Commentary

Advantage:

- Clear path to scaling in tmers of core count
- Clear path to scaling in tmers of SIMD lane count

Disadvantages:

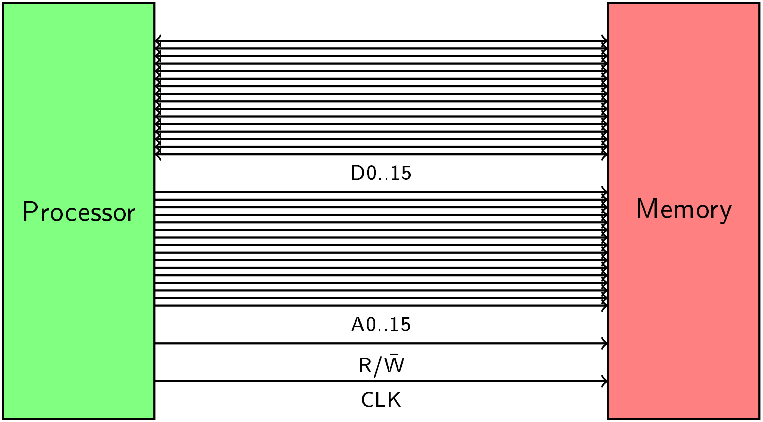
- “Vector” / “Warp” / “Wavefront”
  - Important hardware granularity
  - Poorly/very implicitly represented
- What is the impact of reconvergence?

Performance: Limits to Concurrency

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*
- Block size
- Available ILP
- Number of scheduler (warp/group) slots (not really)
- Synchronization

Memory Systems: Recap



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:

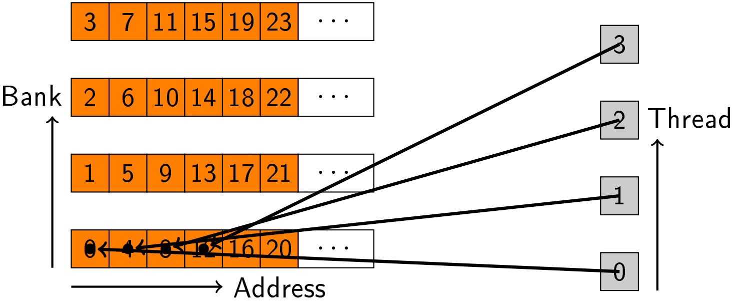
Memory Banking

Fill in the access pattern:


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

Memory Banking

Fill in the access pattern:



`local_variable[BANK_COUNT*lid(0)]`

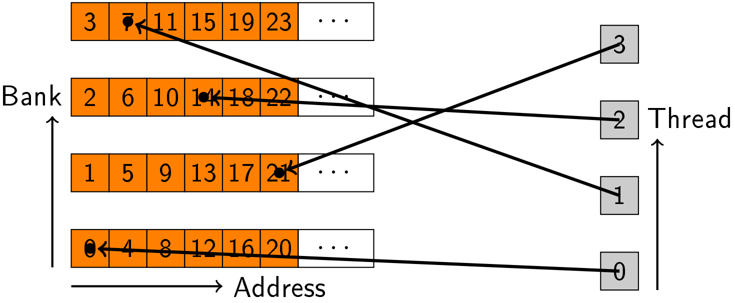
Memory Banking

Fill in the access pattern:

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

Memory Banking

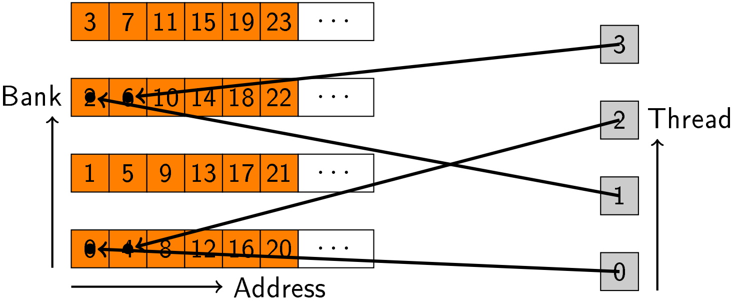
Fill in the access pattern:



`local_variable[ODD_NUMBER*lid(0)]`

Memory Banking

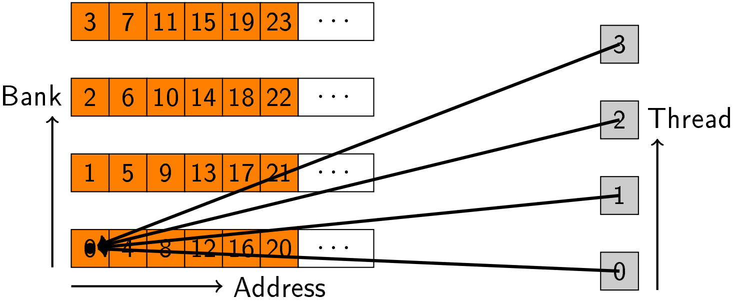
Fill in the access pattern:



`local_variable[2*lid(0)]`

Memory Banking

Fill in the access pattern:




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

Memory Banking: Observations

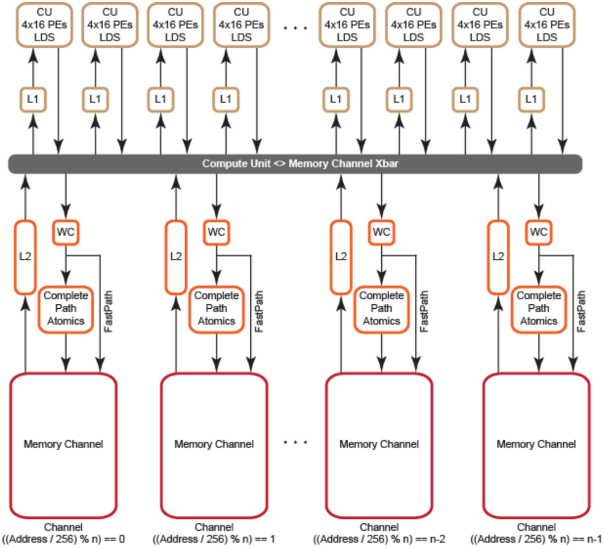
- Factors of two in the stride: generally bad
- In a conflict-heavy access pattern, padding can help
  - Usually not a problem since scratchpad is transient by definition
- Word size (bank offset) may be adjustable (Nvidia)

Given that unit strides are beneficial on global memory access, how do you realize a transpose?

Workgroup size (e.g.): 16x16

```
__local float tmp[16 * 17];
tmp[lid(0)*17 + lid(1)] = a[lid(1) * 16 +
lid(0)];
barrier(CLK_LOCAL_MEM_FENCE);
```

GPU Global Memory System



[GCN Optimization Manual, AMD]

GPU Global Memory Channel Map: Example

Byte address decomposition:

Address	Bank	Chnl	Address
31 ?	11	10 8	7 0

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

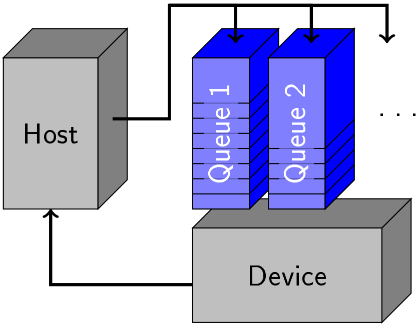
Are there any guaranteed-good memory access patterns?

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)

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



Timing GPU Work

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

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)

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

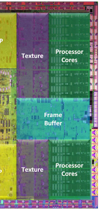
Kernel launch overhead?

10 microseconds

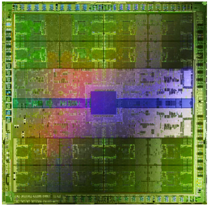
Good source of details: [Wikipedia: List of Nidia GPUs]

Outline

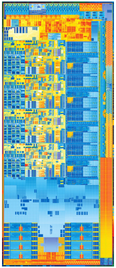
Die Shot Gallery



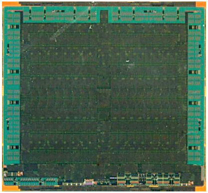
T200
08)



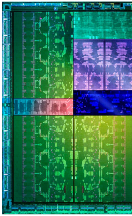
Nv Fermi
(2010)



Intel IVB
(2012)



AMD Tahiti
(2012)



Nv GK1
(2012

Trends in Processor Architecture

What can we expect from future processor architectures?

- Commodity chips
- "Infinitely" many cores
- "Infinite" vector width
- Must hide memory latency ($\rightarrow$ ILP, SMT)
- Compute bandwidth $\gg$ Memory bandwidth
- Bandwidth only achievable by *homogeneity*
- Can't keep the whole thing powered all the time anyway. Consequence?
  Lots of weird stoff springs up. Examples: "Raytracing Cores", "Tensor Cores"

Common Challenges

What are the common challenges encountered by both CPUs and GPUs?

- Dealing with Latency (ILP/SMT/Caches)
- Exposing concurrency
- Expose a coherent model for talking to SIMD
- Making memory system complexity manageable

Goal: Try to see CPUs and GPUs as points in a design space 'continuum' rather than entirely different things.

Outline

PTX: Demo

[Demo: machabstr/PTX and SASS]

[Nvidia PTX manual]

PTX: Cache Annotations

Loads:

`.ca`	Cache at all levels–likely to be accessed again
`.cg`	Cache at global level (cache in L2 and below and not L1)
`.cs`	Cache streaming–likely to be accessed once
`.lu`	Last use
`.cv`	Consider cached system memory lines stale–fetch again

Stores:

`.wb`	Cache write-back all coherent levels
`.cg`	Cache at global level (cache in L2 and below and not L1)
`.cs`	Cache streaming–likely to be accessed once
`.wt`	Cache write-through (to system memory)

Lost/hidden at the C level!

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

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

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

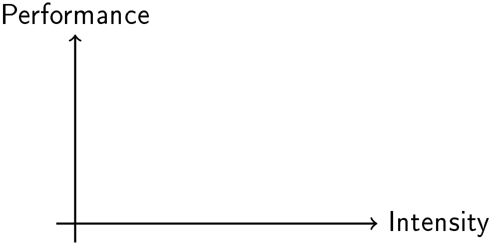
A Graphical Representation: 'Roofline'

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];}
```

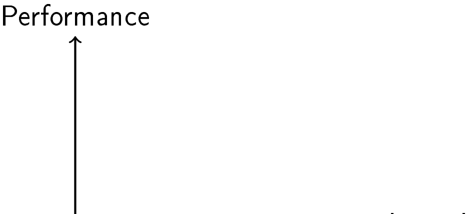
Find the parameters and make a prediction.

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



Refining the Model

- P_{max}: Applicable peak performance of a loop, assuming that data comes from the fastest data path (this is not necessarily P_{peak})
- Computational intensity ("work" per byte transferred) over the slowest data path utilized
- b: Applicable peak bandwidth of the slowest data path utilized

Hager et al. '17

Calibrating the Model: Bandwidth

Typically done with the STREAM benchmark.
Four parts: Copy, Scale, Add, Triad `a[i] = b[i] + s\cdot c[i]`
Do the four measurements matter?

- No–they're a crude attempt at characterizing intensity.
- On a modern machine, all four measurements should be identical.

Any pitfalls?

Streaming stores, remember?

McCalpin: STREAM

Calibrating the Model: Peak Throughput

Name aspects that should/could be factored in when determining peak performance:

- Types of operation (FMA? Or only adds?)
- SIMD
- Pipeline utilization / operation latency
- Throughput of faster datapaths

Practical Tool: IACA

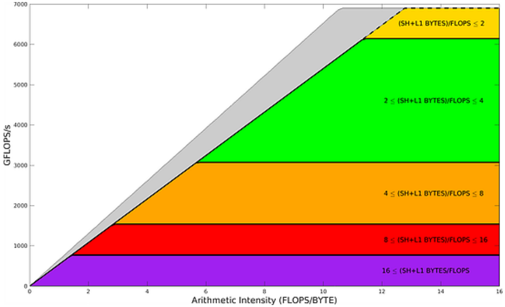
Question: Where to obtain an estimate of P_{max}?

Demo: perf/Forming Architectural Performance Expectations

Questions:

- What does IACA do about memory access? / the memory hierarchy?

An Example: Exploring Titan V Limits



- Memory bandwidth: 652 GB/s theoretical, 540 GB/s achievable
- Scratchpad / L1 throughput:
  80 (cores) x 32 (simd width) x 4 (word bytes) x 1.2 (base clock) ˜= 12.288 TB/s
- Theoretical peak flops of 6.9 TFLOPS/s [Wikipedia]

Warburton '18

Rooflines: Assumptions

What assumptions are built into the roofline model?

- Perfect overlap
  (What would imperfect overlap be in the bank analogy?)
- Only considers the dominant bottleneck
- Throughput-only (!)
  No latency effects, no start-up effects, only steady-state

Important to remember:

- It is what you make of it–the better your calibration, the more info you get
- But: Calibrating on experimental data loses predictive power (e.g. SPMV)

Modeling Parallel Speedup: A 'Universal' Scaling Law

Develop a model of *throughput* $X(N)$ for a given load N, assuming execution resources scale with N.

$$X(N) = \frac{\gamma N}{1 + \alpha \cdot (N-1) + \beta N \cdot (N-1)}$$

What do the individual terms model?

- γ: Throughput increase per load increase
- α: Contention due to waiting/queueing for shared resources/sequential sections ("Amdahl's law")
- β: Incoherence penalty due to waiting for data to become coherent through *point-to-point exchange*

[Gunther '93]

Outline

Combining Multiple Measurements

How can one combine multiple performance measurements? (e.g. “average speedup”?)

Example: Which computer should you buy?

Execution time [s]	Computer A	Computer B	Computer C
Program 1	1	10	20
Program 2	1000	100	20

	Computer A	Computer B	Computer C
Arithmetic mean	500.5	55	20
Geometric mean	31.622	31.622	20

Combining Multiple Measurements: Observations

	Computer A	Computer B	Computer C
Arithmetic mean	500.5	55	20
Geometric mean	31.622	31.622	20

- Depending on normalization, the arithmetic mean will produce an arbitrary ranking
- Geometric mean $\sqrt[n]{a_1 \cdots a_n}$: consistent ranking
- Is geomean good? (What is the meaning of multiplying times?)

Take-home message:

- Be mindful of units when combining measurements (e.g. sums of times make sense, but products of times may not)
- Avoid combined measurements if you can
- Ideally: purposefully choose a weighting

Wikipedia

Timing Experiments: Pitfalls

What are potential issues in timing experiments? (What can you do about them?)

- Warm-up effects (do a few runs before timing to only time steady state)
- Timing noise
  - Know your timer granularity
  - Know your clock kinds (wall, montone, process)
  - Know your clock sources (RTC, PIT, APIC, TSC (nominal), TSC (actual))
  - Know your overheads (function call/kernel launch)
  - Make sure your timing granularity is appropriate (On-node: one second is a reasonable number)

Timing Experiments: Pitfalls (part 2)

What are potential issues in timing experiments? (What can you do about them?)

- NUMA placement (use `numactl`, `libnuma`, or respect first-touch)
- Thread migration between cores (and resulting cache effects)
  - Pin your threads to cores
- Uninitialized pages are never fetched
  - Is `calloc` good enough?
- Frequency Scaling
  - Turn it off or run long enough for thermal steady-state
  - Understand how RAPL ("Running average power limit") and "power leases" work
  - Realize there's a dependency on what instructions you execute
- Noise from other users

Outline

Profiling: Basic Approaches

Measurement of "quantities" relating to performance

- Exact: Through binary instrumentation (valgrind/Intel Pin/...)
- Sampling: At *some* interval, examine the program state

We will focus on profiling by *sampling*.
Big questions:

- What to measure?
- At what intervals?

Defining Intervals: Performance Counters

A *performance counter* is a counter that increments every time a given event occurs.
What events?

- Demo: perf/Using Performance Counters
- see also Intel SDM, Volume 3

Interaction with performance counters:

- Read repeatedly from user code
- Interrupt program execution when a threshold is reached
- Limited resource!
  - Only a few available: 4-8 per core
  - Each can be configured to count one type of event
  - Idea: Alternate counter programming at some rate (requires steady-state execution!)

Profiling: What to Measure

- Raw counts are hard to interpret
- Often much more helpful to look at *ratios* of counts per core/subroutine/loop/...

What ratios should one look at?

Demo: perf/Using Performance Counters

Profiling: Useful Ratios

Basic examples:

- (Events in Routine 1)/(Events in Routine 2)
- (Events in Line 1)/(Events in Line 2)
- (Count of Event 1 in X)/(Count of Event 2 in X)

Architectural examples:

- instructions / cycles
- L1-dcache-load-misses / instructions
- LLC-load-misses / instructions
- stalled-cycles-frontend / cycles
- stalled-cycles-backend / cycles

Issue with 'instructions' as a metric?

May or may not correlate with 'amount of useful work'

“Top-Down” Performance Analysis

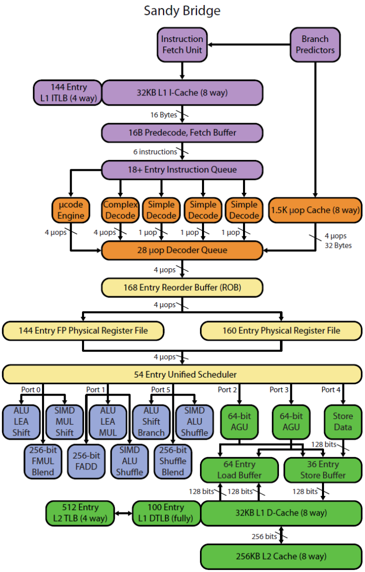
Idea: Account for useful work per available issue slot

What is an issue slot?

> A clock cycle that passed at an interface to an execution pipeline

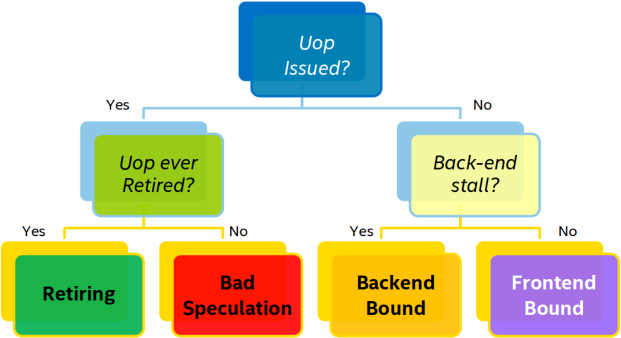
[Yasin ‘14]

Issue Slots: Recap



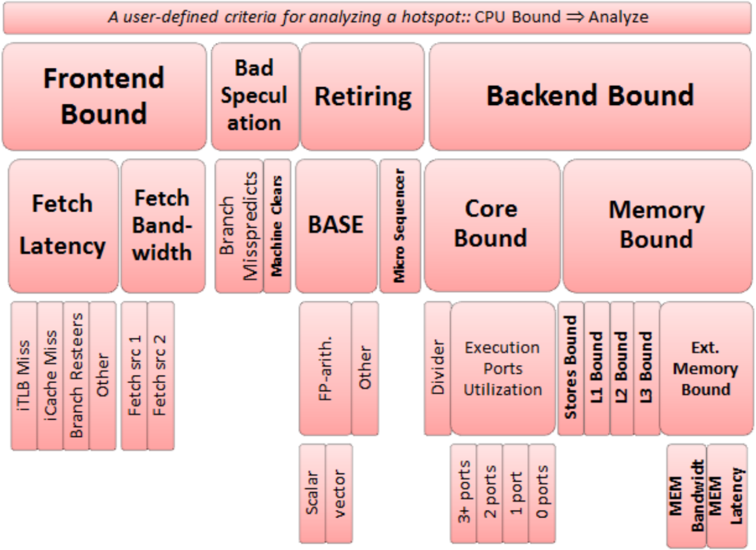
[David Kanter / Realworldtech.com]

What can happen to an issue slot: at a high level?



[Yasin '14]

What can happen to an issue slot: in detail?



[Yasin '14]

Outline

Demo: Performance Counters

Show the rest of:
Demo: perf/Using Performance Counters

Outline

Outline

Expression Trees and Term Rewriting

Demos:

- Demo: lang/01 Expression Trees
- Demo: lang/02 Traversing Trees
- Demo: lang/03 Defining Custom Node Types
- Demo: lang/04 Common Operations

How do expression trees come to be? (not our problem here)

- Partitioning, classification of input stream into tokens (lexing)
- Extraction of higher-level constructs from token stream (parsing)
  - Recursive descent
  - Table/automata-based (e.g. Yacc, ANTLR, PLY, boost::spirit)

Embedded languages

Main challenge: Obtaining a syntax tree. Approaches?

- Symbolic execution (seen above, runtime)
- Type system abuse
  - Demo: lang/Expression Templates
- boost::metaparse (string → tree at compile time)
- "Reflection"
  - Demo: lang/05 Reflection in Python

Macros: Goals and Approaches

What is a macro?

- In C: Simple textual replacement with parameters
- Generally: any type of compile-time computation (that operates on the code)
  - Question: How would you express loops in the C preprocessor?

What data do macro systems operate on?

- Character-streams
- Syntax/expression trees

Macros: Textual and Syntactic, Hygiene

Macros: What can go wrong if you're not careful?

```
#define INCI(i) do { int a=0; ++i; } while(0)
int main(void)
{
    int a = 4, b = 8;
    INCI(a);
    INCI(b);
    printf("a is now %d, b is now %d\n", a, b);
    return 0;
}
```

How can the problem above be avoided?

Ensure macro-internal identifiers (e.g. a above)

Towards Execution

Demo: lang/06 Towards Execution

Outline

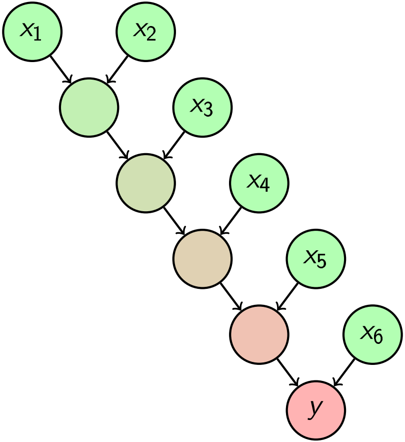
Reduction

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

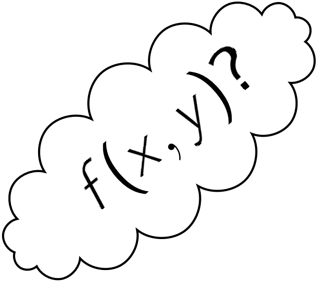
where N is the input size.
Also known as

- 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'?

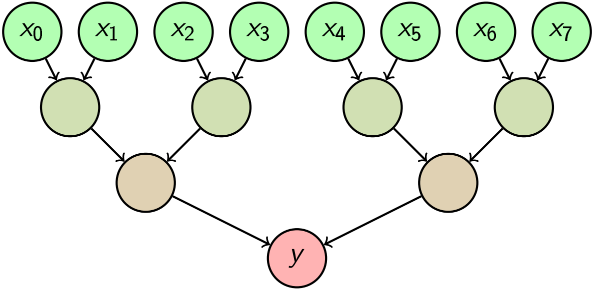
$$f(f(x,y),z) = f(x,f(y,z))$$

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*

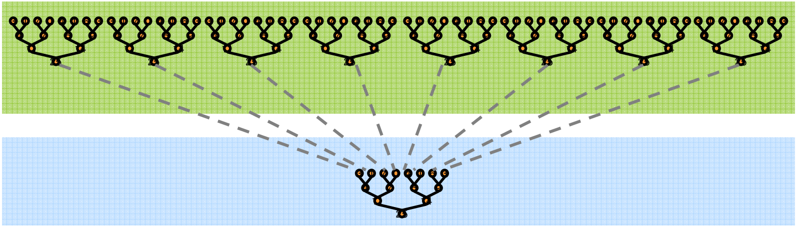
Reduction: A Better Graph



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.

Map-Reduce

But no. Not even close.

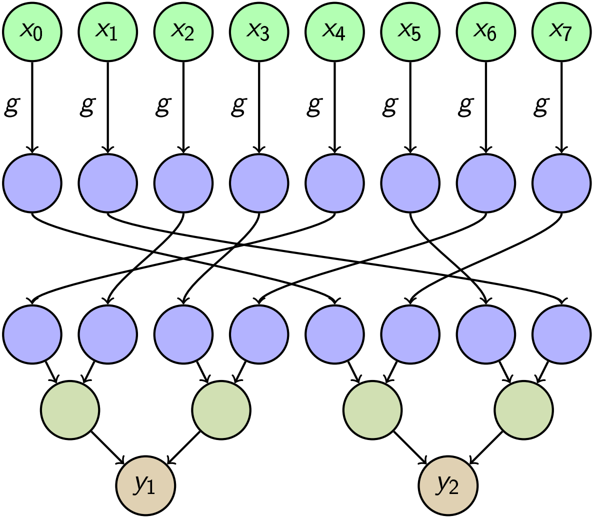
Sounds like this:

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

where N is the input size.

- Lisp naming, again
- Mild generalization of reduction

Map-Reduce: Graph



Scan

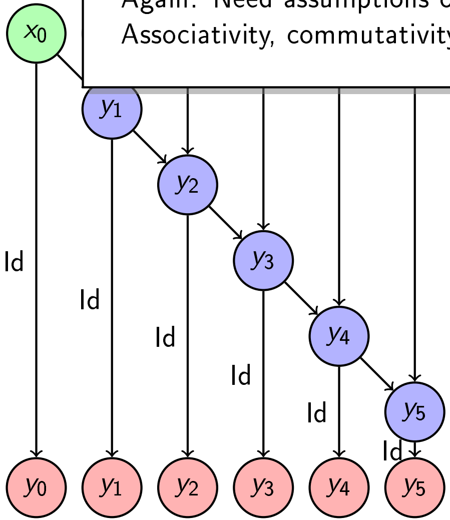
$$
\begin{aligned}
y_1 &= x_1 \\
y_2 &= f(y_1, x_2) \\
\vdots &= \vdots \\
y_N &= f(y_{N-1}, x_N)
\end{aligned}
$$

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

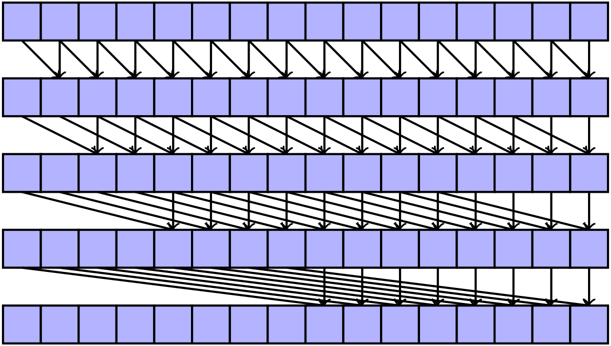
- Prefix Sum/Cumulative Sum
- Abstract view of: loop-carried dependence
- Also possible: Segmented Scan

Scan: Graph

Again: Need assumptions on f.
Associativity, commutativity.



Scan: Implementation



Work-efficient?

Scan: Implementation II

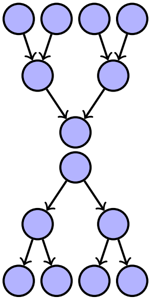
Two sweeps: Upward, downward, both tree-shape

On upward sweep:

- Get values `L` and `R` from left and right child
- Save L in local variable `Mine`
- Compute `Tmp = L + R` and pass to parent

On downward sweep:

- Get value `Tmp` from parent
- Send `Tmp` to left child
- Sent `Tmp+Mine` to right child



Scan: Examples

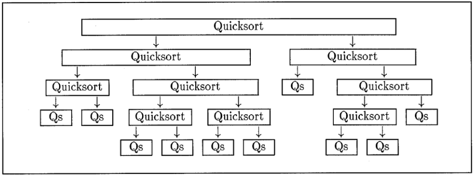
Name examples of Prefix Sums/Scans:

- 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

Data-parallel language: Goals

Goal: Design a full data-parallel programming language

Example: What should the (asymptotic) execution time for Quicksort be?



$O(log(N))$

Question: What parallel primitive could be used to realize this?

- Segmented Scan, i.e. a scan with data-dep. boundaries
- Any basic scan operation can be segmented while retaining

Blelloch '95

NESL Example: String Search

```
teststr = "string  strap asop string" : [char]
>>> candidates = [0:#teststr-5];
candidates = [0, 1, 2, 3, .... : [int]
>>> {a == `s: a in teststr -> candidates};
it = [T, F, F, F, F, F, F, T, F, F....] : [bool]
>>> candidates = {c in candidates;
...          a in teststr -> candidates | a == `s};
candidates = [0, 7, 13, 20, 24] : [int]
>>> candidates = {c in candidates;
...          a in teststr -> {candidates+1:candidates}
...          | a == `t};
```

- Work and depth of this example?
- NESL specifies work and depth for its constructs
- How can scans be used to realize this?

Blelloch '95

Array Languages

Idea:

- Operate on entire array at once
- Inherently data-parallel

Examples:

- APL, numpy
- Tensorflow (talk on Friday), Pytorch

Important axes of distinction:

- Lazy or eager
- Imperative (with in-place modification) or pure/functional

Outline

Outline

Basic Object: Presburger Set

Think of the problem statement here as representing an arbitrary-size (e.g.: dependency) graph.
Presburger sets correspond to a subset of predicate logic acting on tuples of integers.
Important: Think of this as a mathematical tool that can be used in many settings.

Basic Object: Presburger Set

Terms:

- Variables, Integer Constants
- $+, -$
- $\lfloor \cdot / d \rfloor$

Predicates:

- $(\text{Term}) \leq (\text{Term})$
- $(\text{Pred}) \wedge (\text{Pred})$, $(\text{Pred}) \vee (\text{Pred})$, $\neg(\text{Pred})$
- $\exists v : (\text{Pred})(v)$

Sets: integer tuples for which a predicate is true
Verdoolaege '13

Presburger Sets: Reasoning

What's "missing"? Why?

- Multiplication, Division
- Most questions become undecidable in its presence

Why is this called 'quasi-affine'?

- Affine: $\vec{a} \cdot \vec{x} + b$.
- Quasi: inclusion of modulo/existential quantifier

Presburger Sets: Reasoning

What do the resulting sets have to do with polyhedra? When are they convex?

- Each constraint specifies a half-space
- Intersection of half-spaces is a convex polyehdron
- Unions can be used to make non-convex polyhedra

Why polyhedra? Why not just rectangles?

- Rectangular domains are not closed under many transformations
- E.g. strip-mining

Demo: Constructing and Operating on Presburger Sets

Demo: lang/Operating on Presburger Sets

Making Use of Presburger Sets

- Loop Domains
- Array Access Relations (e.g. write, read: per statement)
- Schedules, with “lexicographic time”
- Dependency graphs
- (E.g. cache) interference graphs

Q: Specify domain and range for the relations above.

Example: Dependency Graph

Given:

- Write access relation W: Loop domain $\to$ array indices
- Read access relation R
- Schedule S for statement S_i: Loop domain $D \to$ lex. time of statement instance
- Relation $\prec$: Lexicographic 'before'

Find the dependency graph:

$$D = ((W^{-1} \circ R) \cup (R^{-1} \circ W) \cup (W^{-1} \circ W)) \cap (S \prec S)$$

Verdoolaege '13

Example: Last Instance

Given:

- Write access relation W: Loop domain $\to$ array indices
- Read access relation R
- Schedule S for statement S_i: Loop domain $D \to$ lex. time of statement instance
- Relation $\prec$: Lexicographic 'before'

Find the statement instances accessing array element:

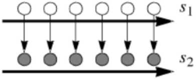
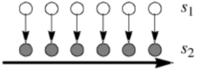
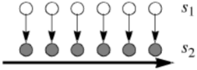
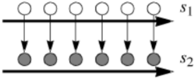
$$(R \cup W)^{-1}$$

Find the *last* statement instance accessing array element:

$$\text{lexmax}((R \cup W)^{-1})$$

Verdoolaege '13

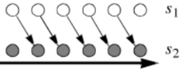
Primitive Transformations (I)

<table>
<tr><th>Source Code</th><th>Partition</th><th>Transformed Code</th></tr>
<tr>
<td><pre>for (i=1; i<=N; i++)
  Y[i] = Z[i]; /*s1*/
for (j=1; j<=N; j++)
  X[j] = Y[j]; /*s2*/</pre>

</td>
<td>Fusion
$s_1 : p = i$
$s_2 : p = j$</td>
<td><pre>for (p=1; p<=N; p++){
  Y[p] = Z[p];
  X[p] = Y[p];
}</pre>

</td>
</tr>
<tr>
<td><pre>for (p=1; p<=N; p++){
  Y[p] = Z[p];
  X[p] = Y[p];
}</pre>

</td>
<td>Fission
$s_1 : i = p$
$s_2 : j = p$</td>
<td><pre>for (i=1; i<=N; i++)
  Y[i] = Z[i]; /*s1*/
for (j=1; j<=N; j++)
  X[j] = Y[j]; /*s2*/</pre>

</td>
</tr>
</table>

[Aho/Ullman/Sethi '07]

Primitive Transformations (II)

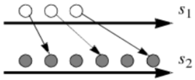
```
for (i=1; i<=N; i++) {
  Y[i] = Z[i];     /*s1*/
  X[i] = Y[i-1]; /*s2*/
}
```



Re-indexing
$s_1 : p = i$
$s_2 : p = i - 1$

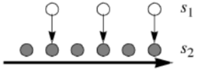
```
if (N>=1) X[1]=Y[0];
for (p=1; p<=N-1; p++){
  Y[p]=Z[p];
  X[p+1]=Y[p];
}
if (N>=1) Y[N]=Z[N];
```


```
for (i=1; i<=N; i++)
  Y[2*i] = Z[2*i]; /*s1*/
for (j=1; j<=2N; j++)
  X[j]=Y[j];         /*s2*/
```



Scaling
$s_1 : p = 2 * i$
$(s_2 : p = j)$

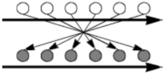
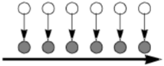
```
for (p=1; p<=2*N; p++){
  if (p mod 2 == 0)
    Y[p] = Z[p];
  X[p] = Y[p];
}
```



[Aho/Ullman/Sethi '07]

Primitive Transformations (III)

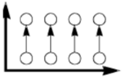
Source Code	Partition	Transformed Code

`for (i=0; i>=N; i++)`
`  Y[N-i] = Z[i];    /*s1*/`
`for (j=0; j<=N; j++)`
`  X[j] = Y[j];    /*s2*/`

 	
	Reversal
	$s_1 : p = N - i$
	$(s_2 : p = j)$
		`for (p=0; p<=N; p++){`
		`  Y[p] = Z[N-p];`
		`  X[p] = Y[p];`
		`}`
		

[Aho/Ullman/Sethi '07]

Primitive Transformations (IV)

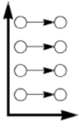
```
for (i=1; i<=N; i++)
  for (j=0; j<=M; j++)
    Z[i,j] =
        Z[i-1,j];
```



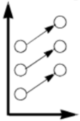
Permutation

$$\begin{bmatrix} p \\ q \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix}$$

```
for (p=0; p<=M; p++)
  for (q=1; q<=N; i++)
     Z[q,p] = Z[q-1,p]
```



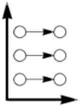
```
for (i=1; i<=N+M-1; i++)
  for (j=max(1,i+N);
       j<=min(i,M); j++)
    Z[i,j] =
      Z[i-1,j-1];
```



Skewing

$$\begin{bmatrix} p \\ q \end{bmatrix} = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

```
for (p=1; p<=N; p++)
  for (q=1; q<=M; q++)
     Z[p,q-p] =
        Z[p-1,q-p-1]
```



[Aho/Ullman/Sethi '07]

Example: Last Instance

Given:

- Dependency relation D

Check whether a transformed schedule S' is valid:

$$\vec{i} \to \vec{j} \in D \Rightarrow S'(\vec{i}) \prec S'(\vec{j})$$

A peek under the hood: Fourier-Motzkin Elimination

INPUT: A polyhedron S with variables $x1, \ldots, x_n$
OUTPUT: $x1, \ldots, x_{n-1}$

- Let C be all the constraints in S involving x_n.
- For every pair of a lower and an upper bound on x_m in C:

$$\begin{aligned} L &\leq c_1 x_n, \\ &\leq c_2 x_n \leq U, \end{aligned}$$

  create the new constraint $c_2 L \leq c_1 U$.
- Divide by the GCD of c_1, c_2 if applicable.
- If the new constraint is not satisfiable, S was empty.
- Let $S' = S \setminus C \cup \bigcup_{L,U} \{c_2 L \leq c_1 U\}$.

[Aho/Ullman/Sethi '07]
Q: How can this help implement an emptiness check?