Midterm Review

Flynn's Taxonomy

Mike Flynn, "Very High-Speed Computing Systems," Proc. of IEEE, 1966

- **S**ingle **I**nstruction **S**ingle **D**ata
- **S**ingle **I**nstruction **M**ultiple **D**ata
  - Array Processor
  - Vector Processor
  - GPU
- **M**ultiple **I**nstruction **S**ingle **D**ata
  - Closest form: systolic array processor, streaming processor
- **M**ultiple **I**nstruction **M**ultiple **D**ata
  - Multiprocessor
  - Multithreaded processor

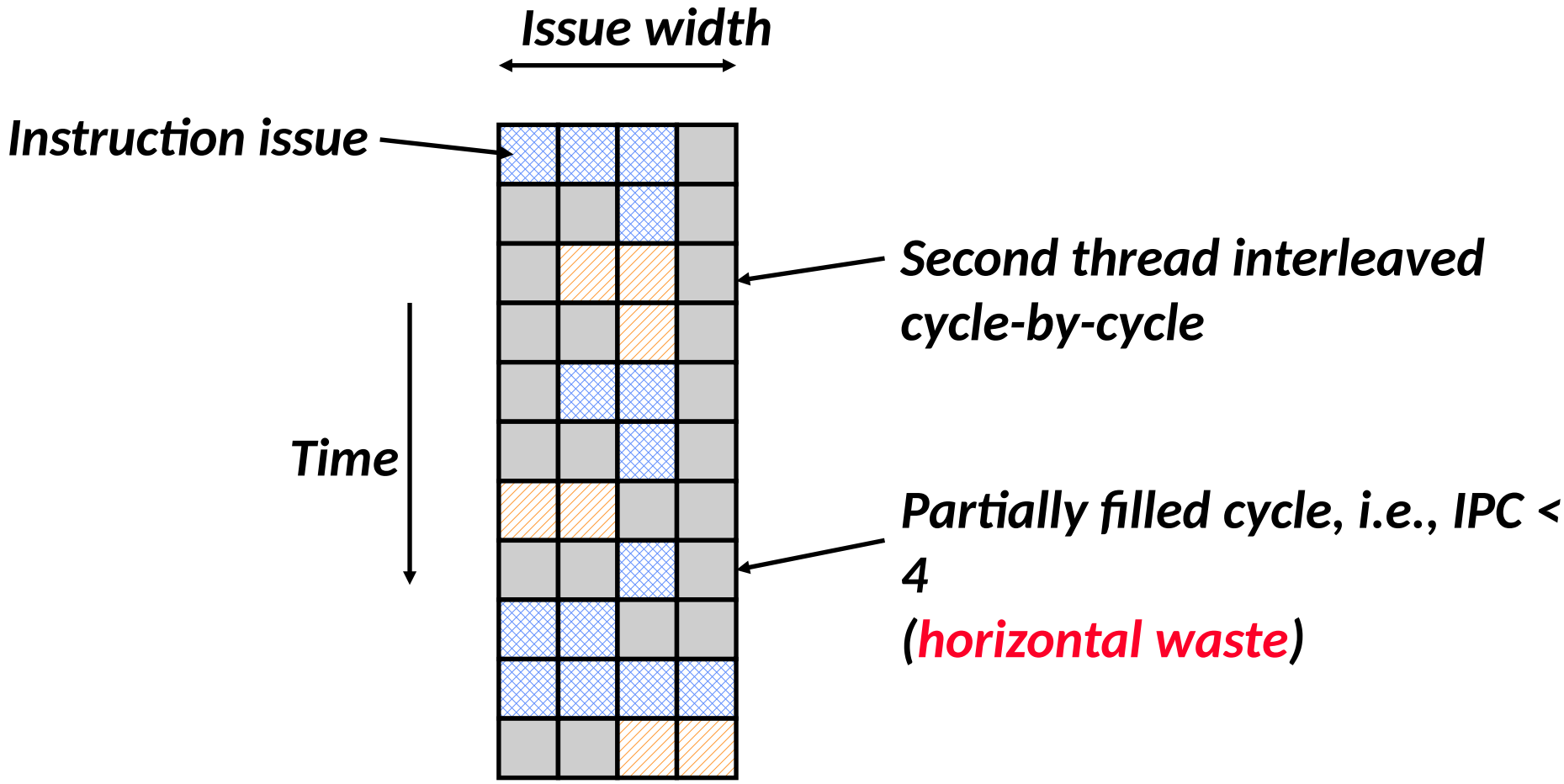
Instruction Level Parallelism (ILP)

- Improve processor performance by having multiple processor components or functional units simultaneously executing instructions.
- Pipelining - functional units are arranged in *stages*.
- Multiple issue - multiple instructions *might* be simultaneously initiated with resource duplication.
  - VLIW – Very long instruction word - functional units are scheduled at compile time (static scheduling).
  - Superscalar - functional units are scheduled at run-time (dynamic scheduling)

Multithreading

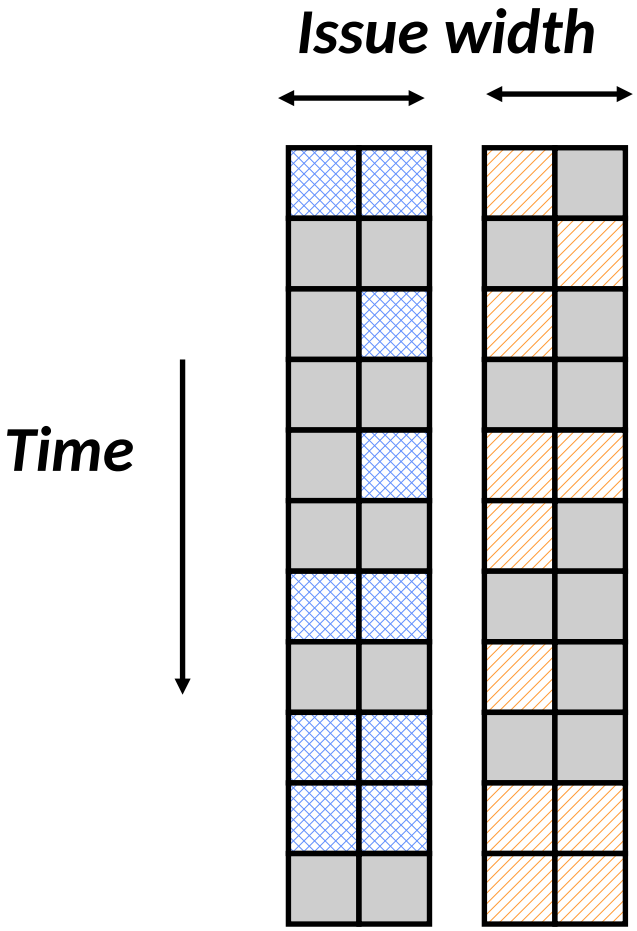
- Difficult to continue to extract instruction-level parallelism (ILP) from a single sequential thread of control
- Many workloads can make use of thread-level parallelism (TLP)
  - TLP from multiprogramming (run independent sequential jobs)
  - TLP from multithreaded applications (run one job faster using parallel threads)
- Multithreading uses TLP to improve utilization of a single processor

Vertical Multithreading



- Cycle-by-cycle interleaving removes vertical waste, but leaves some horizontal waste

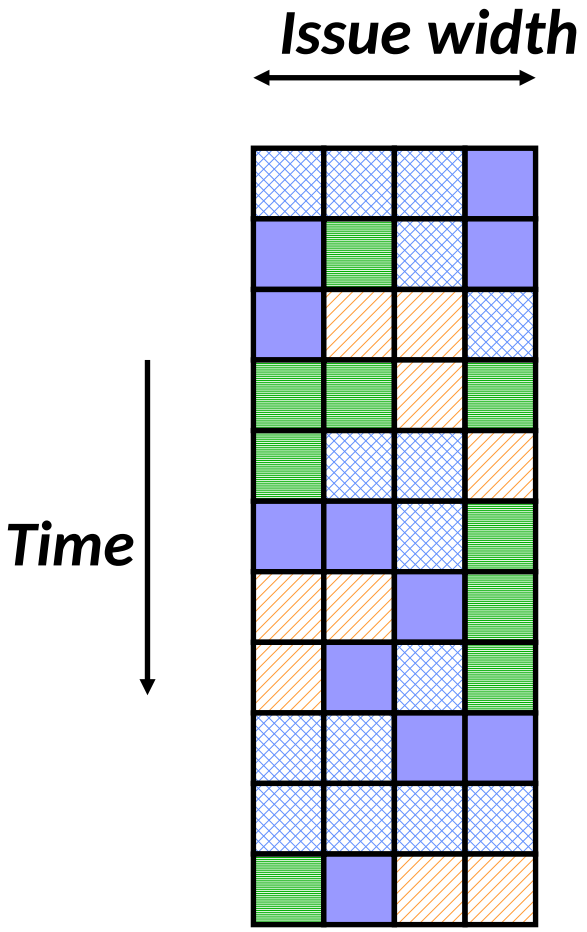
Chip Multiprocessing (CMP)



- What is the effect of splitting into multiple processors?
  - reduces horizontal waste,
  - leaves some vertical waste, and
  - puts upper limit on peak throughput of each thread.

Ideal Superscalar Multithreading

[Tullsen, Eggers, Levy, UW, 1995]



- Interleave multiple threads to multiple issue slots with no restrictions

Importance of Data Parallelism

- GPUs are designed for graphics
  - Highly parallel tasks
- Process *independent* vertices & fragments
  - No shared or static data
  - No read-modify-write buffers
- Data-parallel processing
  - GPU architecture is ALU-heavy
  - Performance depends on *arithmetic intensity*
    - Computation / Bandwidth ratio
  - Hide memory latency with more computation

Arithmetic Intensity

Lots of operations per word transferred

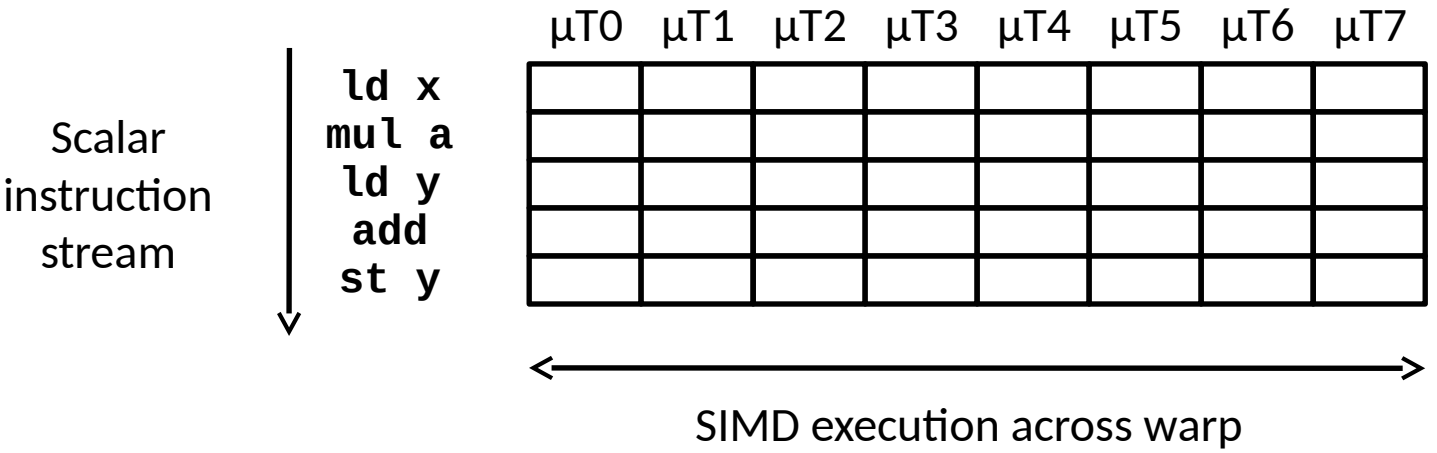
Graphics pipeline

- Vertex
  - Bandwidth: 1 triangle = 32 bytes;
  - Operations: 100-500 f32-ops / triangle
- Rasterization
  - Create 16-32 fragments per triangle
- Fragment
  - Bandwidth: 1 fragment = 10 bytes
  - Operations: 300-1000 i8-ops/fragment

Courtesy of Pat Hanrahan

“Single Instruction, Multiple Thread”

- GPUs use a **SIMT** model, where individual scalar instruction streams for each CUDA thread are grouped together for SIMD execution on hardware (Nvidia groups 32 CUDA threads into a *warp*)



Implications of SIMT model

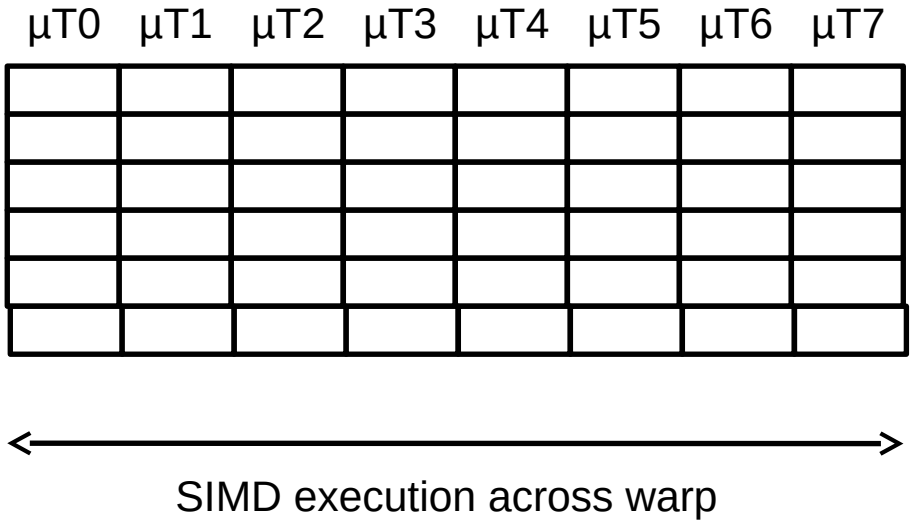
- All “vector” loads and stores are scatter-gather, as individual µthreads perform scalar loads and stores
  - GPU adds hardware to dynamically coalesce individual µthread loads and stores to mimic vector loads and stores
- Every µthread has to perform stripmining calculations redundantly (“am I active?”) as there is no scalar processor equivalent
- Illusion of many independent threads
- But for efficiency, programmer must try and keep µthreads aligned in a SIMD fashion
  - Try and do unit-stride loads and store so memory coalescing kicks in
  - Avoid branch divergence so most instruction slots execute useful work and are not masked off

Conditionals in SIMT model

- Simple if-then-else are compiled into *predicated execution*, equivalent to vector masking
- More complex control flow compiled into branches
- How to execute a vector of branches?

Scalar instruction stream

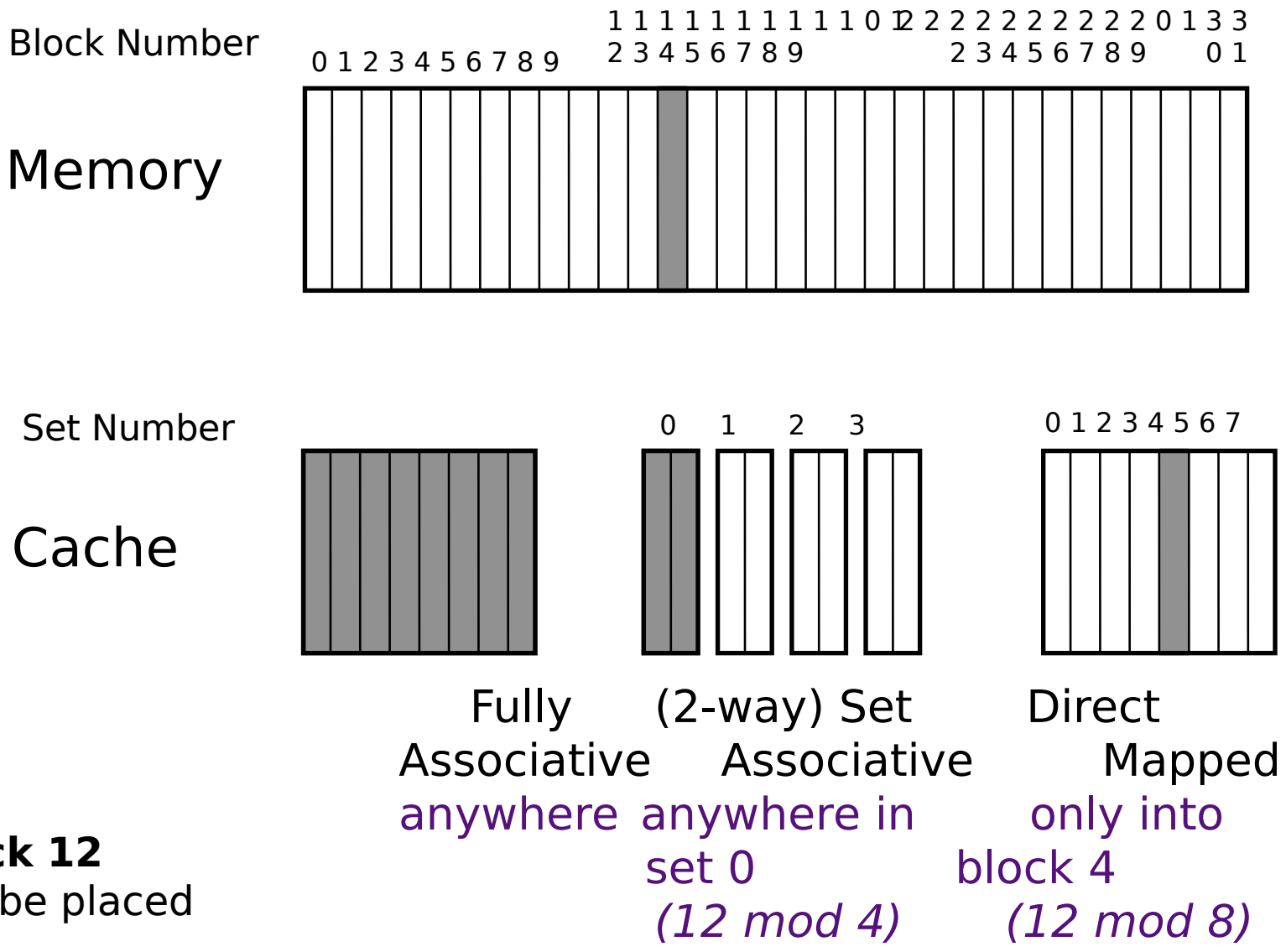
```
tid=threadid
If (tid >= n) skip
   Call func1
       add
       st y
skip:
```



Cache Organizations

- "Fully Associative": Block can go anywhere
  - Note: No Index field, but one comparator/block
- "Direct Mapped": Block goes one place
  - Note: Only 1 comparator
  - Number of sets = number blocks
- "N-way Set Associative": N places for a block
  - Number of sets = number of blocks / N
  - N comparators

Placement Policy



block 12
can be placed

Replacement Policy

In an associative cache, which block from a set should be evicted when the set becomes full?

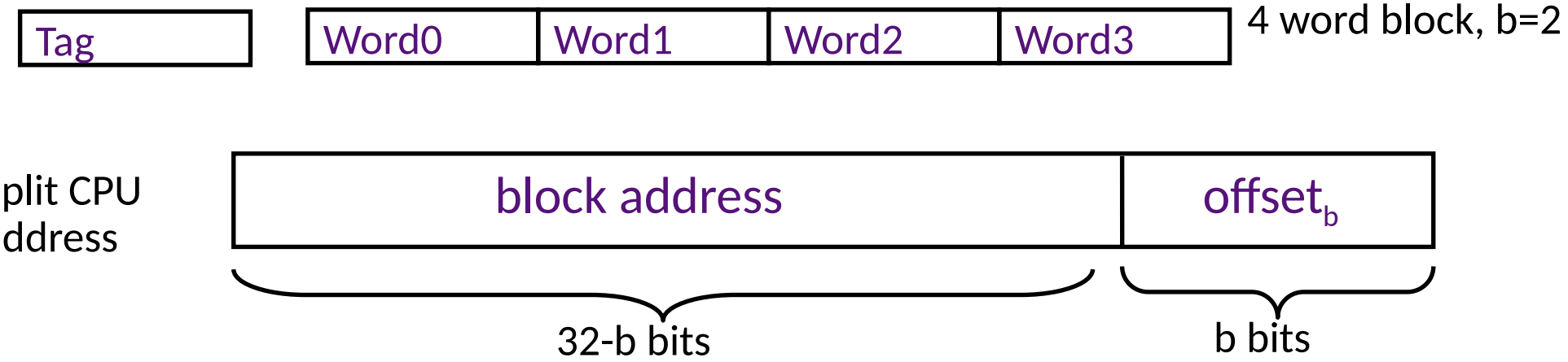
- Random
- Least-Recently Used (LRU)
  - LRU cache state must be updated on every access
  - true implementation only feasible for small sets (2-way)
  - pseudo-LRU binary tree often used for 4-8 way
- First-In, First-Out (FIFO) a.k.a. Round-Robin
  - used in highly associative caches
- Not-Most-Recently Used (NMRU)
  - FIFO with exception for most-recently used block or blocks

This is a second-order effect. Why?

Replacement only happens on misses

Block Size and Spatial Locality

Block is unit of transfer between the cache and memory



Larger block size has distinct hardware advantages

- less tag overhead
- exploit fast burst transfers from DRAM
- exploit fast burst transfers over wide busses

What are the disadvantages of increasing block size (assuming constant memory size)?

Fewer blocks => more conflicts. Can waste bandwidth.

False Sharing

A cache line contains more than one word

Cache-coherence is done at the line-level and *not* word-level

Suppose M_1 writes $word_i$ and M_2 writes $word_k$ and both words have the same line address.

What can happen?

Coherency misses

1) True sharing misses arise from the communication of data through the cache coherence mechanism
   - Invalidates due to 1st write to shared line
   - Reads by another CPU of modified line in different cache
   - Miss would still occur if line size were 1 word, i.e., truly shared
2) False sharing misses when a line is invalidated because a word in the line, other than the one being read, is written into
   - Invalidation does not cause a new value to be communicated, but only causes an extra cache miss
   - Line is shared, but no word in line is *actually shared*
     → miss would not occur if line size were 1 word

Blocked (Tiled) Matrix Multiply

```
// implements C = C + A*B
// A,B,C to be N-by-N matrices of b-by-b subblocks
for i = 1 to n
    for j = 1 to n
        for k = 1 to n
            // Matrix multiply over blocks of size b = n / N
            C(i,j) = C(i,j) + A(i,k) * B(k,j)
```

if n % b ~= 0 you need to have code for the “edges/fringes”

$2n^2$ to read and write each block of C once: $2N^2 * b^2 = 2n^2$

$N*n^2$ to read each block of A N^3 times: $N^3 * b^2 = N^3 * (n/N)^2$

$N*n^2$ to read each block of B N^3 times: $N^3 * b^2 = N^3 * (n/N)^2$

Computational Intensity, CI = f / m = $2n^3 / ((2N + 2) * n^2) \approx n / N = b$ for large n

Take-Aways

- Matrix vector and matrix matrix multiplication key (linear algebra)
- Matrix vector multiplication
  - Opportunities for better parallelism and use special instructions, Fused Multiply Add, etc.
  - Limited by bandwidth ($O(2n^2)$ flops on $O(n^2)$ data)
- Matrix matrix multiplication
  - Can improve computational intensity O(2n3) flops on O(3n2) data
  - Tiling (aka blocking)
- Optimizations in practice
  - Expose parallelism to compiler (SIMD, ILP, prefetching)
  - Help compiler with register use
  - Help hardware (data layout) with cache locality (temporal and spatial)
- Optimized libraries (BLAS) exist

Optimizing in Practice

- Tiling for registers
  - loop unrolling, use of named “register” variables
- Tiling for multiple levels of cache and TLB
- Exploiting fine-grained parallelism in processor
  - superscalar; pipelining
- Complicated compiler interactions (flags)
- Hard to do by hand

Note on Matrix Storage

- A matrix is a 2-D array of elements, but memory addresses are “1-D”
- Conventions for matrix layout
  - by column, or “column major” (Fortran default); A(i,j) at A+i+j*n
  - by row, or “row major” (C default) A(i,j) at A+i*n+j
  - recursive (later)
- Intense interaction with cache line size

Tips on Tuning

"We should forget bout small efficiences, say about 97% of the time: premature optimization is the root of all evil."– C.A.R. Hoare (quoted by Donald Knuth)

- Tradeoff: speed vs readability, debuggability, maintainability...
- Only optimize when needful
- Go for low-hanging fruit first: data layouts, libraries, compiler flags
- Concentrate on the bottleneck
- Concentrate on inner loops
- Get correctness (and a test framework) first

Loop Unrolling

- Expose instruction-level parallelism and reduce control overhead

```
float f0 = filter[0], f1 = filter[1], f2 = filter[2];
float s0 = signal[0], s1 = signal[1], s2 = signal[2];
*res++ = f0*s0 + f1*s1 + f2*s2;
do {
   signal += 3;
   s0 = signal[0];
   res[0] = f0*s1 + f1*s2 + f2*s0;
   s1 = signal[1];
   res[1] = f0*s2 + f1*s0 + f2*s1;
   s2 = signal[2];
   res[2] = f0*s0 + f1*s1 + f2*s2;
   res += 3;
} while( … );
```

Basic Types of Synchronization: Mutexes

Mutexes -- mutual exclusion aka locks

threads are working mostly independently

need to access common data structure

```
lock *l = alloc_and_init();    /* shared */
acquire(l);
 access data
release(l);
```

Locks only affect processors using them:

If a thread accesses the data without doing the acquire/release, locks by others will not help

Java, C++, and other languages have lexically scoped synchronization, i.e., synchronized methods/blocks (In Java: synchronized (l) {…})

Can't forgot to say "release"

Semaphores (a signaling mechanism) generalize locks to allow k threads simultaneous access; good for limited resources.

Unlike in a mutex, a semaphore can be decremented by another process (a mutex can only be unlocked by its owner)

Example Problem: Arithmetic Intensity

```
for (int i=1; i<n; i++)
    for (int j=1; j<n; j++) {
        u(i,j) = u(i,j) - 0.125*( u(i+1,j) + u(i-1,j) +
                 u(i+1,j-1) + u(i,j-1) + u(i-1,j-1) +
                 u(i+1,j+1) + u(i-1,j+1) + u(i,j+1));
    };
```

What is OpenMP?

- OpenMP = Open specification for Multi-Processing
  - openmp.org – Talks, examples, forums, etc.
  - Specification controlled by the Architecture Review Board (ARB)
- Motivation: capture common usage and simplify programming
- OpenMP Architecture Review Board (ARB)
  - A nonprofit organization that controls the OpenMP Spec
  - Latest spec: OpenMP 5.0 (Nov. 2018)
- High-level API for programming in C/C++ and Fortran
  - Preprocessor (compiler) directives (~ 80%)

    #pragma omp construct [clause [clause …]]
  - Library Calls (~ 19%)

    #include <omp.h>
  - Environment Variables (~ 1%)

    all caps

The OpenMP Common Core:
Most OpenMP programs use these 9+10 items

#pragma omp parallel	Parallel region, teams of threads, structured block, interleaved execution across threads
setenv OMP_NUM_THREADS N	Internal control variables. Setting the default number of threads with an environment variable
#pragma omp barrier #pragma omp critical	Synchronization and race conditions. Revisit interleaved execution.
#pragma omp for #pragma omp parallel for	Worksharing, parallel loops, loop carried dependencies
#pragma omp single	Workshare with a single thread
#pragma omp task #pragma omp taskwait	Tasks including the data environment for tasks.

The OpenMP Common Core:
Most OpenMP programs use these 9+10 items

int omp_get_thread_num() int omp_get_num_threads()	Create threads with a parallel region and split up the work using the number of threads and thread ID
double omp_get_wtime()	Speedup: False Sharing and other performance issues
reduction(op:list)	Reductions of values across a team of threads
schedule(dynamic [,chunk]) schedule (static [,chunk])	Loop schedules, loop overheads and load balance
private(list), firstprivate(list), shared(list)	Data environment
nowait	Disabling implied barriers on workshare constructs, the high cost of barriers, and the flush concept (but not the flush directive)

OpenMP basic syntax

- Most of the constructs in OpenMP are compiler directives.
- Most OpenMP* constructs apply to a “structured block”.
  - Structured block: a block of one or more statements with one point of entry at the top and one point of exit at the bottom.
  - It’s OK to have an exit() within the structured block.

	C and C++	Fortran
Compiler directive	#pragma omp construct [clause [clause] …]	!$OMP construct [clause [clause] …]

Example	#pragma omp parallel private(x)
	{
	  ...
	}
		!$OMP PARALLEL
		  ...
		!$OMP END PARALLEL

Function prototype	#include <omp.h>	use OMP_LIB

Thread creation: Parallel regions

- You create threads in OpenMP with the **parallel** construct.
- Example, To create a 4 thread Parallel region:

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int ID = omp_get_thread_num();
        pahrump(ID,A);
}
```

- Each thread calls pahrump(ID,A) for ID = 0 to 3
- OpenMP does not have to delete the threads here; it can create a thread pool and reuse it (implementation dependent)
- This is in contrast to pthreads, which is *imperative* (the library will kill all threads by definition during pthread_join)

Example problem

```
#define N 1000
Extern struct data member [ N ] ;
Extern int isgood (int i ) ;
int good_members [ N ] ;
int pos = 0 ;

void find_good_members ( )
{
#pragma omp parallel for
  For (i=0; i< N; i++) {
    If ( is_good ( i ) ) {
        good_members [ pos ] = i ;
#pragma omp atomic
        pos++
    }
  }
}
```

Why such poor scaling? False sharing

- If independent data elements happen to sit on the same cache line, each update will cause the cache lines to “slosh back and forth” between threads … This is called *“false sharing”.*
- If you promote scalars to an array to support creation of an SPMD program, the array elements are contiguous in memory and hence share cache lines … Results in poor scalability.
- Solution: Pad arrays so elements you use are on distinct cache lines.

Snoopy bus protocol (reprise)

- Basic idea:
  - Broadcast operations on memory bus
  - Cache controllers “snoop” on all bus transactions
  - Memory writes induce serial order
  - Act to enforce coherence (invalidate, update, etc)
- Problems:
  - Bus bandwidth limits scaling
  - Contending writes are slow
- There are other protocol options (e.g. directory-based).
  - But usually give up on *full* sequential consistency.

Synchronization: critical section

```
float  res;
#pragma omp parallel
{     float B;   int i, id, nthrds;
      id = omp_get_thread_num();
      nthrds = omp_get_num_threads();
      for(i=id;i<niters;i+=nthrds){
          B =  big_job(i);
#pragma omp critical
          res += consume (B); // Thread wait: One at a time...
      }
}
```

Synchronization: Barrier

- Barrier: a point in a program all threads much reach before any threads are allowed to proceed.
- It is a “stand alone” pragma meaning it is not associated with user code … it is an executable statement.

```
double Arr[8], Brr[8];
int numthrds;
omp_set_num_threads(8)
#pragma omp parallel
{    int id, nthrds;
     id = omp_get_thread_num();
     nthrds = omp_get_num_threads();
     if (id==0) numthrds = nthrds;
     Arr[id] = big_ugly_calc(id, nthrds);
#pragma omp barrier
     Brr[id] = really_big_and_ugly(id, nthrds, Arr);
}
```

Reduction

- OpenMP reduction clause: **reduction** (op : list)
  - Inside a parallel construct:
  - A local copy of each list variable is made and initialized depending on the “op” (e.g. 0 for “+”)
  - Updates occur on the *local copy*
  - Local copies are reduced into a single value and combined with the original global value.
- The variables in “list” must be shared in the enclosing parallel region.

```
double  ave=0.0, A[MAX];    int i;

#pragma omp parallel for reduction (+:ave)

  for (i=0;i< MAX; i++) {

        ave + = A[i];

  } // implicit barrier here

  ave = ave/MAX;
```

Takeaway

- Programming shared memory machines
  - May allocate data in large shared region without too many worries about where
  - Memory hierarchy is critical to performance
    - Even more so than on uniprocessors, due to coherence traffic
  - For performance tuning, watch sharing (both true and false)
- Semantics
  - Need to lock access to shared variable for read-modify-write
  - Sequential consistency is the natural semantics
    - Write race-free programs to get this
  - Architects worked hard to make this work
    - Caches are coherent with buses or directories
    - No caching of remote data on shared address space machines
  - But compiler and processor may still get in the way
    - Non-blocking writes, read prefetching, code motion…
    - Avoid races or use machine-specific fences carefully

Network Design

- Topology (how things are connected)
  - Crossbar; ring; 2-D, 3-D, higher-D mesh or torus; hypercube; tree; butterfly; perfect shuffle, dragon fly, …
- Routing algorithm:
  - Example in 2D torus: all east-west then all north-south (avoids deadlock).
- Switching strategy:
  - Circuit switching: full path reserved for entire message, like the telephone.
  - Packet switching: message broken into separately-routed packets, like the post office, or internet
- Flow control (what if there is congestion):
  - Stall, store data temporarily in buffers, re-route data to other nodes, tell source node to temporarily halt, discard, etc.

Performance Properties of a Network

- Diameter: the maximum (over all pairs of nodes) of the shortest path between a given pair of nodes.
- Latency: delay between send and receive times
  - Latency tends to vary widely across architectures
  - Vendors often report hardware latencies (wire time)
  - Application programmers care about software latencies (user program to user program)
- Observations:
  - Latencies differ by 1-2 orders across network designs
  - Software/hardware overhead at source/destination dominate cost (1s-10s usecs)
  - Hardware latency varies with distance (10s-100s nsec per hop) but is small compared to overheads
- Latency is key for programs with many small messages

Latency and Bandwidth Model

- Time to send message of length n is roughly

$$\text{Time} = \text{latency} + n*\text{cost\_per\_word}$$
$$= \text{latency} + n/\text{bandwidth}$$

- Topology is assumed irrelevant
- Often called "α-β model" and written

$$\text{Time} = \alpha + n*\beta$$

- Usually $\alpha >> \beta >>$ time per flop.
  - One long message is cheaper than many short ones.

$$\alpha + n*\beta \quad << \quad n*(\alpha + 1*\beta)$$

  - Can do hundreds or thousands of flops for cost of one message.
- Lesson: *Need large computation-to-communication ratio to be efficient*

Programming Distributed Memory Machines with Message Passing

- Overview of MPI
- Basic send/receive use
- Non-blocking communication
- Collectives

Message Passing Libraries

- All communication, synchronization require subroutine calls
  - *No shared variables*
  - Program run on a single processor just like any uniprocessor program, except for calls to message passing library
- Subroutines for
  - Communication
    - Pairwise or point-to-point: Send and Receive
    - Collectives all processor get together to
      - Move data: Broadcast, Scatter/gather
      - Compute and move: sum, product, max, prefix sum, … of data on many processors
  - Synchronization
    - Barrier
    - No locks because there are no shared variables to protect
  - Inquiries
    - How many processes? Which one am I? Any messages waiting?

MPI Basic Concepts

- Processes can be collected into groups
- Each message is sent in a context, and must be received in the same context
  - Provides necessary support for libraries
- A group and context together form a communicator
- A process is identified by its rank in the group associated with a communicator
- There is a default communicator whose group contains all initial processes, called **`MPI_COMM_WORLD`**

Blocking and Non-blocking Communication

- So far we have been using *blocking* communication:
  - `MPI_Recv` does not complete until the buffer is full (available for use).
  - `MPI_Send` does not complete until the buffer is empty (available for use).
- Completion depends on size of message and amount of system buffering.

What MPI Functions are in Use?

For simple applications, these are common:

- Point-to-point communication
  - MPI_Irecv, MPI_Isend, MPI_Wait, MPI_Send, MPI_Recv
- Startup, Shutdown
  - MPI_Init, MPI_Finalize
- Information on the processes
  - MPI_Comm_rank, MPI_Comm_size, MPI_Get_processor_name
- Collective communication
  - MPI_Allreduce, MPI_Bcast, MPI_Allgather

MPI Collective Routines

- Many Routines: **`Allgather, Allgatherv, Allreduce, Alltoall, Alltoallv, Bcast, Gather, Gatherv, Reduce, Reduce_scatter, Scan, Scatter, Scatterv`**
- **`All`** versions deliver results to all participating processes.
- V versions allow the hunks to have variable sizes.
- **`Allreduce`**, **`Reduce`**, **`Reduce_scatter`**, and **`Scan`** take both built-in and user-defined combiner functions.
- MPI-2 adds **`Alltoallw`**, **`Exscan`**, intercommunicator versions of most routines

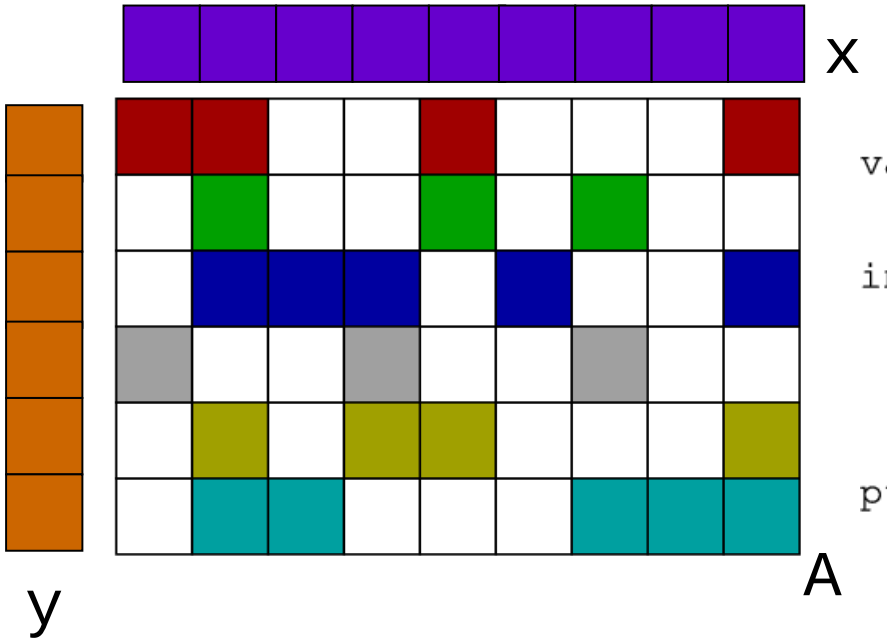
Summary of particle methods

- Model contains discrete entities, namely, particles
- Time is continuous – must be discretized to solve
- Simulation follows particles through timesteps
  - Force = external _force + nearby_force + far_field_force
  - All-pairs algorithm is simple, but inefficient, $O(n^2)$
  - *Particle-mesh* methods approximates by moving particles to a regular mesh, where it is easier to compute forces
  - *Tree-based* algorithms approximate by treating set of particles as a group, when far away

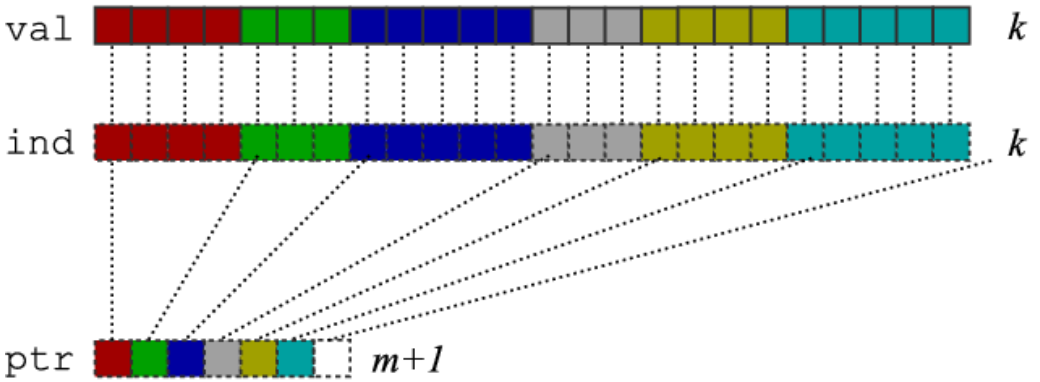
Compressed Sparse Row (CSR) Format

y = y + A*x, only store, do arithmetic, on nonzero entries

CSR format is simplest one of many possible data structures for A



Representation of A

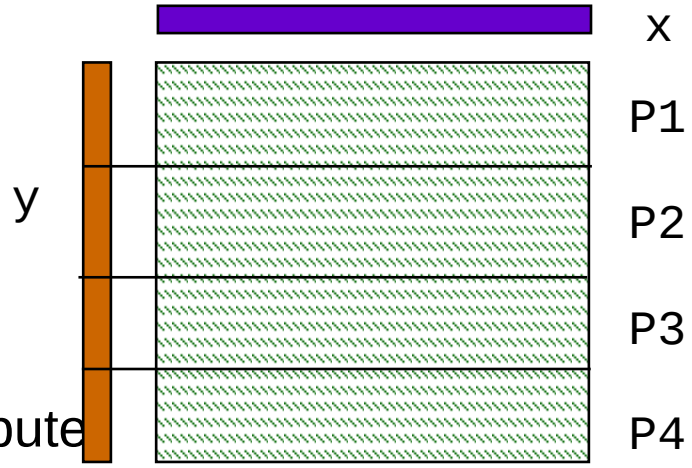


Matrix-vector multiply kernel: $y(i) \leftarrow y(i) + A(i,j) \cdot x(j)$

```
for each row i
  for k=ptr[i] to ptr[i+1]-1 do
        y[i] = y[i] + val[k]*x[ind[k]]
```

Parallel Sparse Matrix-vector multiplication

- y = A*x, where A is a sparse n x n matrix



- Questions
  - which processors store
    - y[i], x[i], and A[i,j]
  - which processors compute
    - y[i] = sum (from 1 to n) A[i,j] * x[j]
      = (row i of A) * x … a sparse dot product
- Partitioning
  - Partition index set {1,…,n} = N1 ∪ N2 ∪ … ∪ Np.
  - For all i in Nk, Processor k stores y[i], x[i], and row i of A
  - For all i in Nk, Processor k computes y[i] = (row i of A) * x
    - "owner computes" rule: Processor k compute the y[i]s it owns.

May require communication

Summary of common problems

- Load Balancing
  - Dynamically – if load changes significantly during job
  - Statically - Graph partitioning
    - Discrete systems
    - Sparse matrix vector multiplication
- Linear algebra
  - Solving linear systems (sparse and dense)
  - Eigenvalue problems will use similar techniques
- Fast Particle Methods
  - O(n log n) instead of $O(n^2)$

Summary of tree algorithms

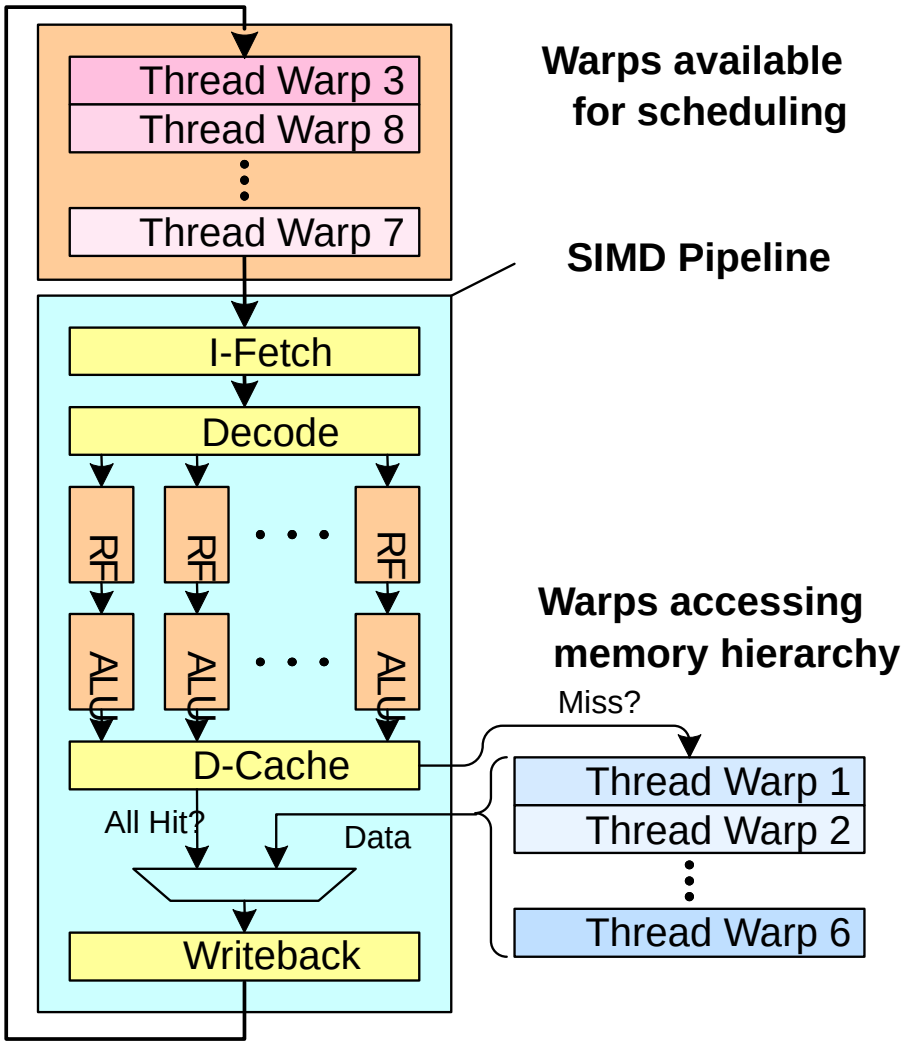
- Lots of problems can be done quickly - in theory - using trees
- Some algorithms are widely used
  - broadcasts, reductions, parallel prefix
  - carry look ahead addition
- Some are of theoretical interest only
  - Csanky's method for matrix inversion
  - Solving tridiagonal linear systems (without pivoting)
  - Both numerically unstable
- Embedded in various systems
  - MPI, NESL (CMU), other languages
  - CM-5 hardware control network

Latency Hiding via Warp-Level Fine Grain Multithreading

- Warp: A set of threads that execute the same instruction (on different data elements)

- Fine-grained multithreading
  - One instruction per thread in pipeline at a time (No interlocking)
  - Interleave warp execution to hide latencies

- Register values of all threads stay in register file
- Fine-Grain multithreading enables long latency tolerance
  - Millions of pixels



Threads

- Threads on desktop CPUs
  - Implemented via lightweight processes (for example)
  - General system scheduler
  - Thrashing when more active threads than processors
- An alternative approach
  - Hardware support for many threads / CPU
    - Modest example: hyperthreading
    - More extreme: Cray MTA-2 and XMT
  - Hide memory latency by thread switching
  - Want many more independent threads than cores
- GPU programming
  - Thread creation / context switching are basically free
  - Want lots of threads (thousands for efficiency?!)

CUDA

- CUDA is a programming model designed for:
  - Heterogeneous architectures
  - Wide SIMD parallelism
  - Scalability

- CUDA provides:
  - A thread abstraction to deal with SIMD
  - Synchronization & data sharing between small thread groups

- CUDA programs are written in C++ with minimal extensions

- OpenCL is inspired by CUDA, but HW & SW vendor neutral

CUDA Threads

- Independent thread of execution
  - has its own program counter, variables (registers),
    processor state, etc.
  - no implication about how threads are scheduled
- CUDA threads might be physical threads
  - as mapped onto GPUs
- CUDA threads might be virtual threads
  - might pick 1 block = 1 physical thread on multicore CPU

GPU Conclusions

- GPUs gain efficiency from simpler cores and more parallelism
  - Very wide SIMD (SIMT) for parallel arithmetic and latency-hiding
- Heterogeneous programming with manual offload
  - CPU to run OS, etc. GPU for compute
- Massive (mostly data) parallelism required
  - Memory coalescing helps
- Threads in block share faster memory and barriers
  - Blocks in kernel share slow device memory and atomics

Graph traversal: Depth-first search (DFS)

```
procedure DFS(vertex v)
    v.visited = true
    previsit (v)
    for all v s.t. (v, w) ∈ E
       if (!w.visited) DFS(w)
    postvisit (v)
```

Graph traversal: Serial Breadth-first search (BFS)

Push seed node onto queue and mark

While queue nonempty:

- Pop node from queue
- Visit node
- Push unmarked neighbors on queue
- Mark all neighbors

Load Balancing Approaches

- Static Load balancing: Divide into equal size parts
  - Straightforward if you know the problem size
  - Can be expensive to estimate / optimize (graph partitioning)
- Dynamic load balancing:
  - When work costs are unknown
  - Observation: Load balancing and locality often trade off

Self Scheduling

- “Self Scheduling” [Tang and Yew ‘86]
  - Shared queue of tasks
  - Processors (workers) take / add tasks to queue

Chunked Scheduling

- “Chunked” (Self) Scheduling [Kruskal and Weiss ‘85]
  - Shared queue of tasks, workers remove chunks of size K
  - Reduce queue contention (locking)
  - How to choose K?
    - Large chunks: lower contention / overhead
    - Small chunks: even finish time

Guided Self Scheduling

- “Guided” Self Scheduling [Kuck and Polychronopolous ‘87]
  - The chunk size K_i at the ith access to the task pool is given by K_i = ceiling(R_i/p) where R_i is the # of tasks remaining and p is the number of processors

Not covered

- FFT
- Cloud Computing
- Linear Algebra Applications