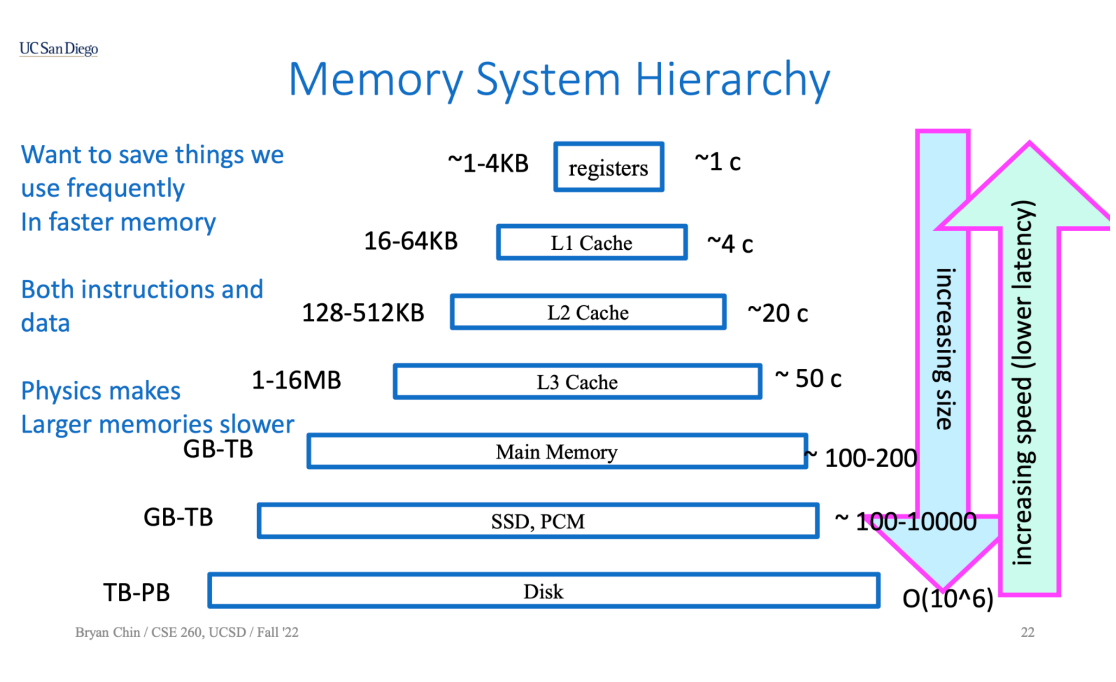
Amdahl's Law – strong scaling - sees the percentage of non-parallelizable code as fixed limit for speedup

$$S_{sys} = \frac{1}{(1-f)+\frac{f}{p}}$$ ***f = parallel fraction, p = n_procs***

Gustafson's Law – weak scaling - assumes that parallel part of the program increases with the problem size and the sequential part stays fixed

$$S_{sys} = p - (p-1)\cdot(1-f)$$

- Little's Law $T = p \times \lambda$
  parallelism = performance (arrival rate) x latency (both on RHS increase with time)
- Speedup (Sp) and Efficiency (Ep) - Ep = Sp / P



Moore's Law - number of transistors in a dense integrated circuit (IC) doubles about every two years - still increases exponentially but every > 2 years

Dennard Scaling - $P \propto CfV^2$

- voltage and current should be proportional to the linear dimensions of a transistor
- as the size of the transistors shrunk, and the voltage was reduced, circuits could operate at higher frequencies at the same power
- **End of DS**: leakage current and threshold voltage not considered by Dennard
- As transistors get smaller, power density increases as these don't scale with size
- These created a "Power Wall" limiting practical processor frequency to around 4 GHz

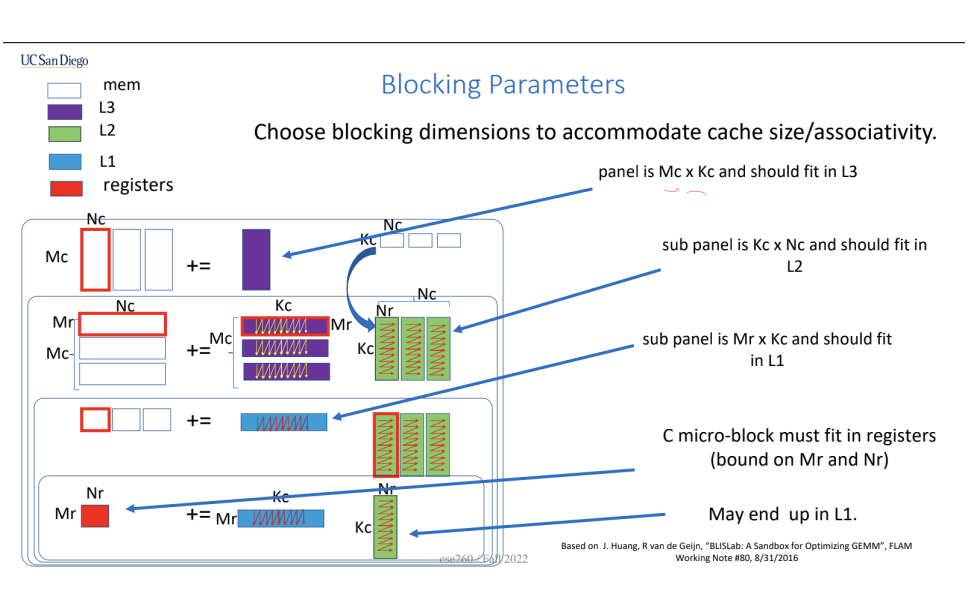
---

Cache

- Locality
  - Temporal - reuse something recently used
  - Spatial - something we used will be near something else we will use
- Misses - cold, capacity, conflict
- Types
  - Fully-associative - any memory block can be in any cache line
  - Direct-mapped - a memory block can go into exactly 1 cache line
  - Set-associative - N lines where a memory block can go
- Levels
  - Inclusive cache: things in L1 must be in L2
  - Exclusive cache: things in L1 must not be in L2
  - Non-inclusive cache: things in L1 might be in L2

Matrix Multiply

- q (computational intensity) = computation to communication ratio
  - Time = $ft_f + mt_m$ (Best case = only ft_f); second term = 1 to reach ½ peak perf.
    m = #words transferred, t_m = slow mem latency, f = #fops, t_f = time per fop
  - ***q = f / m*** (often substituted in time formula)
  - Naive $q = \frac{2n^3}{n^3+3n^2}$ Blocked Inner $q = \frac{2n^3}{(2N+2)n^2/L}$
  - Blocked Outer $q = \frac{2n^3}{(3N+1)n^2/L} \sim \frac{Ln}{N}$ = Lb **(N = n_blocks = n / b)**
    b = blockdim
  - Small N => max q. Fit one block of A,B,C in Mfast => $3b^2 \leq M_{fast}$
    For **GotoBLAS**, only A and B in Mfast => factor of **2; not 3** in equations
  - => $N \geq n \times (3/M_{fast})^{1/2}$ => $q \sim L \times (M_{fast}/3)^{1/2}$
  - GPU Naive $q = \frac{2n^3}{2n^3+3n^2}$
- Tiling
  - Fitting Mr x Nr microblock into register
  - GPU: reduces memory accesses, more ops per thread, increase ILP
- Blocking - divide matrix into submatrices to fit in cache
- Packing - making access pattern in memory contiguous
  - actual multiplication is done on packed A and B




```
/*
 * pack one subpanel of A
 *
 * pack like this
 * if A is row major order
 *
 *    a c e g
 *    b d f h
 *    i k m o
 *    j l n p
 *    q r s t
 *
 * then pack into a sub panel
 * each letter represents sequantial
 * addresses in the packed result
 * (e.g. a, b, c, d are sequential
 * addresses).
 * - down each column
 * - then next column in sub panel
 * - then next sub panel down (on subsegent call)

 *    a c e g
 *    b d f h
 *    -------
 *    i k m o
 *    j l n p
 *    -------
 *    q r s t
 *    0 0 0 0
 */
```

```
/*
 * pack one subpanel of B
 *
 * pack like this
 * if B is
 *
 * row major order matrix
 *    a b c j k l s t
 *    d e f m n o u v
 *    g h i p q r w x
 *
 * each letter represents sequantial
 * addresses in the packed result
 * (e.g. a, b, c, d are sequential
 * addresses).
 *
 * Then pack
 *   - across each row in the subpanel
 *   - then next row in each subpanel
 *   - then next subpanel (on subsequent call)
 *
 *    a b c |  j k l |  s t 0
 *    d e f |  m n o |  u v 0
 *    g h i |  p q r |  w x 0
 *
 *    ^^^^^
 *    each call packs one subpanel
 */
```

Roofline model

- Interpretation of roofline model
  - sets an upper bound on kernel performance depending on its q
- Relationship to arithmetic capability and memory bandwidth
  - Slope = bandwidth; more BW => steeper slope (corner moves left, peak perf at lower q)
  - HW has multiplier and adder but algo uses only adders => perf <= ½ peak
- Compute AI: < ridge point => memory bound; else compute bound

SIMD

- Vectorization - use registers
- Stripmining - technique to tile the code
  - SIMD: divide into VLEN
  - GPU:
    - break loop into block length chunks
    - Break inner loop into chunks of VL (warp size)
- Control divergence
  - Not all loop iterations same - branch paths, loop_cnt % VLEN != 0
  - SIMD Solution: scalar fix up code, blending + masking
  - GPU Solution: predication
- Loop carried dependencies - data dependency
  - A[i + 1] <- A[i] so A[i:i+4] can't be computed in parallel

Multithreading

- OpenMP – compiler hints (pragmas) to parallelize
- Parallelization of loops – types of dependencies (true, output, anti)
  - True (RAW)
    - X <- Y
    - Z <- X
    - Also, loop #2 uses a[ ] produced by loop #1
  - Output (WAW)
    - X <- Y
    - X <- Z
  - Anti (WAR)
    - X <- Y
    - Y <- Z
    - Also, loop #2 overwrites b[ ] before loop #1 uses b[ ]
- OpenMP and dataraces
  - Conflict updating a shared quantity i.e. >= 1 writer on shared data
  - Atomicity errors most difficult to detect
  - Fix: atomic vars, critical sections (mutex), barriers, determinism
  - Mutex - #pragma omp critical

  - Atomic - #pragma omp atomic
  - Implicit barrier reqd even when update atomic to ensure all threads done
- OpenMP reductions - reduce list to single value
  #pragma omp parallel reduction(operation: variable)
- Fork-join parallelism: larger task divided into smaller independent tasks whose solutions can then be combined.

Principles of GPU architecture

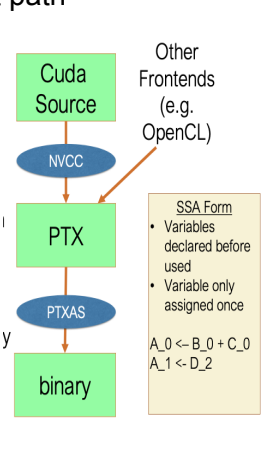
- GPU Memory Hierarchy
  - DRAM takes lots of cycles to access
  - Partition on-chip shared mem, L1 cache
  - RDONLY data cache (Kepler: 48KB, Turing: 2KB)
  - L2 cache
- GPU Thread Hierarchy
  - **Thread** = smallest work unit executed in SM (SIMD lane)
  - Grouped into **thread blocks** (min. dispatch & retirement unit allocated to SM)
  - **Grid** contains multiple blocks; kernel executed on grid
  - Each thread - executed by core; each block - **exec** by **1** SM but > 1 can reside
  - SIMT: HW dispatches thread blocks to SMs which schedule threads from a block to run on cores in **warps** (group of 32 threads)
- GPU warp scheduling
  - 0 overhead switching between warps
  - HW schedules between warps to hide latency
- GPU thread mapping to warps
  - Thread block = 1/2/3D; linearized to 1D, assigned to warps
  - Warps assigned in groups of 32 threads until < 32 threads remain from arrangement
- Estimated speedup from tiling - What is reduction in number of global memory accesses
- Parallel Speedup computation
  - How much did GPU implementation improve over traditional proc
  - $S = \frac{\text{Running time of the fastest program on conventional processors}}{\text{Running time of the accelerated program}}$ $S_p = \frac{T_s}{T_p}$
- Occupancy and interpretation of occupancy
  - Number of active warps / max warps supported by SM
  - Active warp = available for issue within SM
  - Max # limited by shared memory, registers, block size, how much each thread / block consumes
  - Interpretation: higher occupancy leads to better HW utilization (and hiding of latency)
  - Don't maximize occupancy if BW is limited. If more on-chip BW can be used, increase occupancy to reduce off-chip BW
  - Increase threads/block => increase warps/block
  - In case warps/block > 64 (m/c dependent), lesser number of warps used
- GPU Memory Interleaving

  - Use multiple memory controllers
  - Only 1 of 8 => using 1/8th possible BW
- GPU Memory Coalescing
  - Combine requests into a single burst to reduce number of bank accesses
  - Reduce memory transactions per thread
  - Global memory fastest when successive threads read or write locations with least stride
  - G[5 * tid.x] - non contiguous coalescing
- GPU Bank access and bank conflict avoidance
  - Conflict: >= 2 threads access same bank for different 32 words
  - Avoidance: each thread should access different bank
- Thread Level and Instruction Level Parallelism
  - Ways to deal w latency
  - Goal: improve throughput, hide latency
  - TLP - instructions can come from other warps (threads)
  - ILP - instructions can come from same warp (thread)
- Little's Law and GPUs
  - Goal: hide latency by doing other things while waiting
  - Arithmetic instructions = arithmetic throughput x arithmetic latency
  - Memory instructions = memory BW x memory latency
- Cusp behavior – why
  - For small arithmetic intensity, code memory bound
  - Occ = Latency * $min(mem_{thru}, \frac{alu_{thru}}{\alpha}, \frac{issue_{thru}}{\alpha+1})$
  - When arithmetic intensity starts increasing, mem_thru is not minimum; more computations => compute bound
  - At highest point $mem_{thru} == \frac{alu_{thru}}{\alpha}$ equally bound by mem, compute
  - Graph: How much occupancy to reach max GPU utilization
  - High arithmetic intensity doesn't always need highest occupancy

Performance Prediction $\frac{W(n)}{D(n)} = O(\frac{(n^3)}{\log(n)})$

- Larger W(n)/D(n) => more parallelizable - number of "nodes" vs depth of longest path
- W(n) = total work nodes; work nodes on critical path (serial) = D(n)
- n parallel multiplies for n^2 parallel vector inner products, n^2logn adds
- Tcomp = D(n) + [W(n) – D(n)]/p

More GPU



- CUDA compilation flow to achieve source code portability between generations
- PTX portable
- Static Single Assignment
- (nvcc->ptx->ptxas->binary)

Predication vs. branching - Predication reduces thread divergence by converting control dependency to data dependency

GPU architecture

- Pipeline hazards, dependencies
  - Data
    - RAW (true dependency) - all machines
    - WAW (output), WAR (anti) out-of-order machines
  - Functional unit - physical resources unavailable
  - Control - branch outcome not determined, next instruction unknown
- CPU pipelines vs GPU pipelines
  - CPU
    - 2 or 3 cycle cache access, 1 cycle ALU (super-pipelined)
    - Can issue back to back independent instructions
    - Cannot issue back to back dependent instructions
  - GPU
    - Superscalar (2 ALU)
    - No worry about in order completion
- Scoreboarding in a GPU – detect when dependencies are resolved
  - Keep track of warps ready to issue (ready operands)
- Control divergence in a GPU - causes reissue of warp to follow different paths
- Costs – why does it happen - arriving at next path and popping to other previous
- How hardware deals with divergence and associated penalties
  - Goes to original path, arrives at next, pops, goes to new path, rejoins at next
  - Can be nested
- Reduction operation in a GPU and how to lessen control divergence
  - All do same operation, use larger strides, entire warp, all banks of shared mem

Message Passing

- Bulk Synchronous Parallelism model - 3 phases (supersteps)
  - Compute, communicate, synchronize
- What kinds of problems are amenable to stencil methods
  - Physical problems simulated on 1D/2D/3D uniform mesh
  - Mapping from ordered pairs ro physical observables
- What are ghost cells/halo cells
  - Boundaries of tile from other thread blocks
  - Mimic values of adjacent tiles
- Basic MPI concepts : Rank, ordering guarantees/causality, synchronous and asynchronous models. Mpi communicators, mpi tags.
  - Rank - processor number of current processor number in execution
  - Guarantees / Causality
    - multiple msgs sent will be delivered in same order as sent
    - No ordering implied between independent senders to same receiver

---

    - ^ Closer node's data might be received before further node
  - Synchronous model - waits for completion of first to send next (high latency)
  - Asynchronous model - non-blocking, immediate return, optimize msg flow
    - Phase 1 - initiate communication
    - Phase 2 - synchronize
  - Communicators - create subdomains for processes (namespaces), msgs stay within the communicator
  - Tags - allow processes to organize/screen msgs; receiver selects types of msgs to receive - <u>don't affect order</u>
- MPI_Bcast, Scatter, Gather Reduce, etc. and why you might use them
  - Bcast - distribute data from root to all
  - Scatter - evenly distribute amongst communicator procs
  - Scatterv - unevenly distribute
  - Gather - collect data from all communicator procs
  - Reduce - combine data from all processes in a communicator applying function
- Difference in user semantics for blocking and non-blocking send/recv the buffers, what you have to do to synchronize non-blocking comms)
  - Buffer cannot be accessed between IRecv(), ISend() and accompanying Wait()
  - Can compute other stuff between above 3 fn calls
  - Send( ) <u>may or may not</u> complete before a Recv ( ) has been posted
  - Ssend() always blocks
  - Isend() doesn't block
- One-sided communications
  - move data without requiring that the remote process synchronize
  - Each process exposes a part of its memory to other processes
  - Other processes can directly read from or write to this memory

Modeling Performance of MPI (γ, α, β model)

- TotalTIme = γ x #ops + #msgs x (α + β_∞^{-1}n) where n = msg size, β_∞ = peak BW
- α = fixed msg overhead (**latency**), β = network bandwidth, β^{-1} = time to send a word
- Short msg => α dominates; Long msg => BW dominates
- α probably increases as load increases
- Communication costs -
- 1D decomposition $2(\alpha + 8\beta^{-1}N)$ $E_p = \frac{1}{1+\frac{\alpha P + 8NP\beta^{-1}}{8N^2\beta^{-1}}}$
  - Assumptions
    - $N\%P == 0$
    - N / P > 2
    - 1 word = double precision FP = 8 bytes
- 2D decomposition $4(\alpha + 8\beta^{-1}N/\sqrt{P})$ $E_p = \frac{1}{1+\frac{\alpha P + 8N\sqrt{P}\beta^{-1}}{4N^2\beta^{-1}}}$
  - Assumptions
    - Ignore cost of packing msg buffers
    - 1 word = double precision FP = 8 bytes
    - $N\%\sqrt{P} == 0$
    - $N/\sqrt{P}$ > 2

- LogP model (latency, overhead, gap b/w msgs, Processors); α β model

Super linear speedup (Sp > P => Ep > 1)

- Fake vs Real
  - Real - More procs => more mem/cache, mem/cache BW, NW BW
  - Fake - Inaccurate measurement, cache cold/warm, noise in measurement,
    - Ts > PTp => serial implementation not fastest; run Tp twice
- Amdahl's Law – strong scaling
- Gustafson – weak scaling
- What do we mean by isoefficiency function N = f(P)
  - Increase n_ops (n) with P to get weak scaling

Grind time $T_\gamma(P,(m,n)) = \frac{T(P,(m,n))}{m \cdot n \cdot Niter}$ T(P,(m,n)) = running time on P processors

- Why would one use a strip instead of a rectangular submesh $N < \frac{\alpha}{8\beta^{-1}}\frac{\sqrt{P}}{(\sqrt{P}-2)}$
  - Strip decomposition comm cost < box decomposition comm cost
  - Given P, solve for N
- Communication parameters
- Surface vs volume effect
  - Communication proportional to surface area
  - Computation proportional to volume
  - As dimension increases, surface/vol increases => comm/comp increases
  - Volume increases faster than perimeter
- As N increases, grind time can increase because working set may not fit in cache to compute a point - TLB misses, not in memory, larger stride
- Effect of higher dimensions on cache locality. - same point as above
  - Memory strides apart (1, dimX, dimX * dimY)

Parallel Matrix multiplication

- Cannon's algorithm (Tp and Ep) $T_p = 2(n^3/p)\gamma + 2(1+\sqrt{p})(\alpha + \beta n^2/p)$ $E_p = \frac{1}{1+O(\frac{\sqrt{p}}{n\gamma})}$
  - Pre-skew row i of A by i to the left
  - Pre-skew col j of B by j up
  - After each partial product, A circ shift left, B circ shift up
  - += next partial product
  - Ep approaches serial algo as n >> p
  - ***Gamma = compute speed***
- Communication avoidance
  - Each row has own communicator (key = myrank / sqrt(P))
  - Use redundant copies (like shared mem on PA2, some blocks loaded into multiple shared mems)
- Theoretical lower bound on communication (# of words moved and # of messages).

  - #messages = $\Omega(\sqrt{p})$
  - #words = $\Omega(\frac{n^2}{\sqrt{p}})$
- Johnson's 3D algorithm
  - Each processor computes $O(\frac{n^2}{p^{\frac{2}{3}}})$ words of C
  - log(p^1/3) messages do reduction; > 3 not optimal from msg pov
- 2.5D algorithm
  - Each layer does 1/c of Cannon problem
  - $p^{1/3} > c > 1$
  - Then sum them all up along c
  - => c copies of A, B
  - => 1/c * $\sqrt{p/c}$ Cannon steps on each copy of A and B
  - BW: $\Omega(n^2/\sqrt{cp})$
  - Msgs: $\Omega(\sqrt{p}/c^{3/2})$ = cannon steps + reduction (logc)

Special Purpose Accelerators

- How systolic array does matrix multiplication
  - SA: data moves like a wave - process bit by bit, combine, modify
  - W coefficients stored in array
  - FMAs send to all neighbors
- Difference between Cannon's algorithm and TPU's systolic array
  - In Cannon, C computed at each node - As and Bs come
  - In SA, all outputs propagate, results generated at bottom edge
  - Results generated as data moves through array
- Why CISC instructions in the TPU
  - Doesn't have to do general purpose, need small set of instructions
  - Say MM long vector, instruction could stay active for many cycles
  - RISC has instruction overhead

Higher Level abstractions

- Major differences between pthreads, openMP, Cuda, MPI
- **OpenMP** - program on shared memory devices => parallelism occurs where every parallel thread has access to all of your data
  - manages thread creation and synchronization implicitly (unlike pthreads)
- **MPI** - program on distributed memory devices => parallelism occurs where every parallel process works in its own memory space in isolation from the others.
  - higher level, more portable, no language restriction like pthreads
- **CUDA** = shared memory paradigm for GPUs => limited in parallelism and memory it can handle
  - Does selected tasks in parallel, unlike MPI (does everything in parallel)
- **pthreads** - fine-grained control over thread management, mutexes, etc. (low level)