# Parallel Memory hierarchies and address space organization

# Lecture 2

Technology

Motivating Applications

Memory Hierarchies

# Administrivia

- Have you received an email from SDSC about your Triton account? Have you logged in?
- I will set up accounts on Lincoln once you've let me know you are interested in using the GPU

- Office hours
  - Today (after class)
  - Remainder of the quarter: TBD
- Reschedule lecture on 10/1 → Weds10/7 @ 11AM

- Enrollment

# Text and readings

- Required text: Grama, Gupta, Karypis, and Kumar
  *Introduction to Parallel Computing,* 2nd Ed.
  Addison-Wesley, 2003, ISDN 0-201-64865-2
  **Be sure to get the 2nd edition.**
- Assigned class readings will include handouts and on-line material
- Be prepared to discuss the readings in class
- Lecture slides

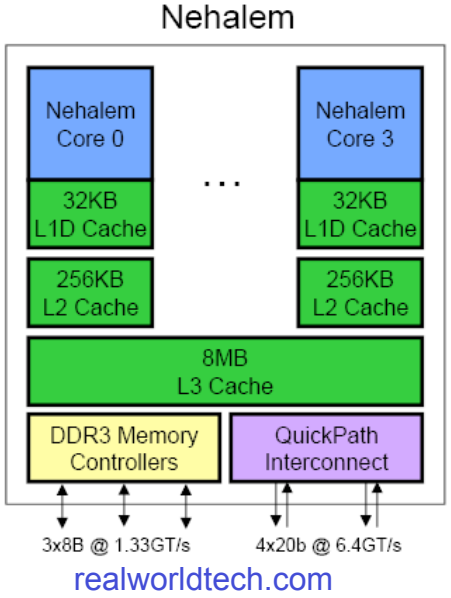
  http://www.cse.ucsd.edu/classes/fa09/cse260/Lectures

# Programming Assignments and projects

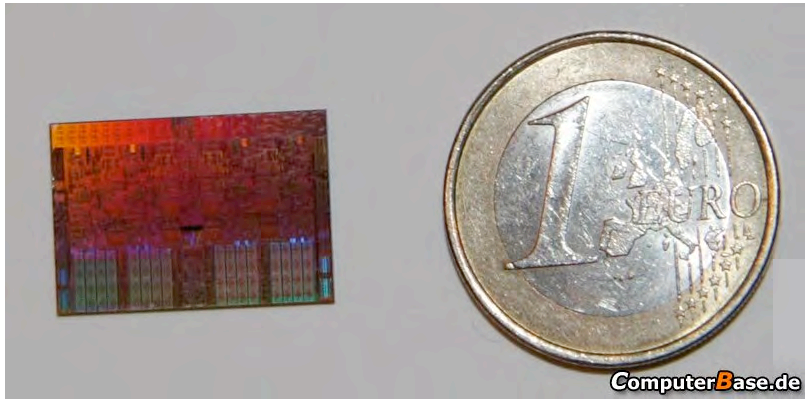
- Three tracks
  - Multicore (SDSC) - openMP, pthreads
  - Nvidia Tesla (NCSA) - CUDA
  - MPI on a cluster (SDSC/NCSA) - MPI
  - Hybrid: {GPU, multicore} + MPI
- Let me know which track you are interested in, see assignment #1
- Find a partner
- Propose a project on 10/15, teams of 3 permitted
  http://cseweb.ucsd.edu/classes/fa09/cse260/Projects/

# Recapping from last time…
# The age of the multi-core processor

- On chip parallel computer
- IBM Power4 (2001), many others follow (Intel, AMD)
- First dual core laptops (2005-6)
- GPUs (nVidia, ATI): supercomputer on a desktop



realworldtech.com




# The impact

- A renaissance in parallel computation
- Parallelism is no longer restricted to machine rooms, it is relevant to everyone
- In a few years, everyone will have an historically unprecedented amount of parallelism at their disposal
  - Don't need to know they are using one
  - Non HPC users

# Is it that simple?

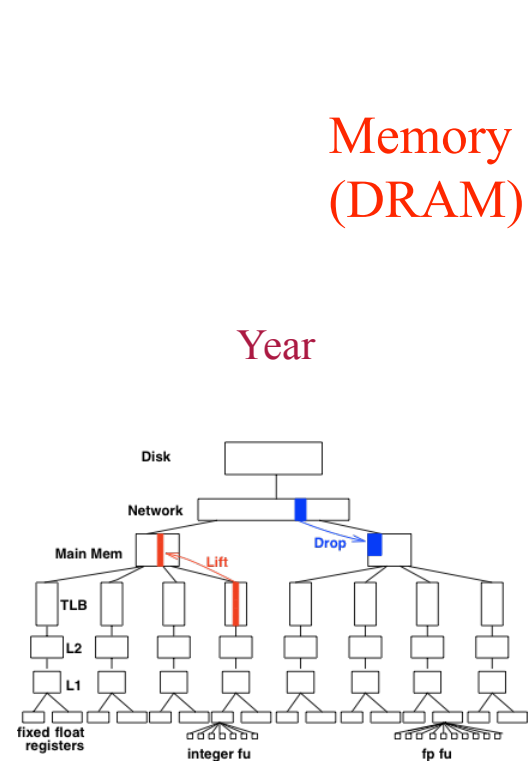
- Simplified processor design, but more user control over the hardware resources

- If we don't use the parallelism, we lose it
  - Amdahl's law - serial sections
  - Von Neumann bottleneck
  - Load imbalances

# Performance and Implementation Issues

- Big payoff for conserving locality
  - Less storage per core
  - Data motion is expensive
  - Computation is cheap
  - Parallel Memory Hierarchy
    Alpern et al. '93
- Lots of hand coding
  - No magical conversion of serial code
  - Migration costs, loss of portability
  - Parallelism is not a substitute for a "good" algorithm
- Little's law [1961]

$$T = p \times \lambda$$



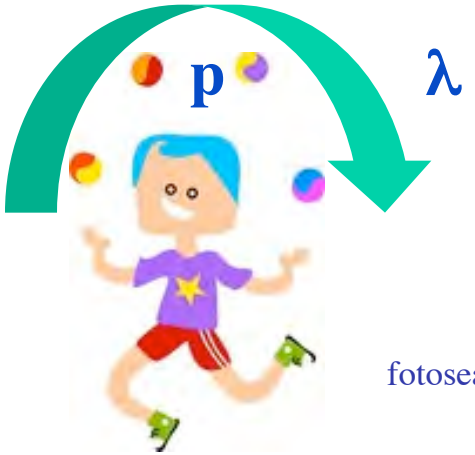
# Little's Law [1961]

- The number of threads = performance × latency

$$T = p \times \lambda$$

- p and λ increasing with time

p =1 - 8 flops/cycle
λ = 500 cycles/word




fotosearch.com

# What you'll take away from this course

- The principles and practice of solving problems on parallel computers
- A tool box of problem solving techniques
- Parallel computing generalizes problems we've encountered in single processor computers
  - Locality
  - Amortize fixed costs

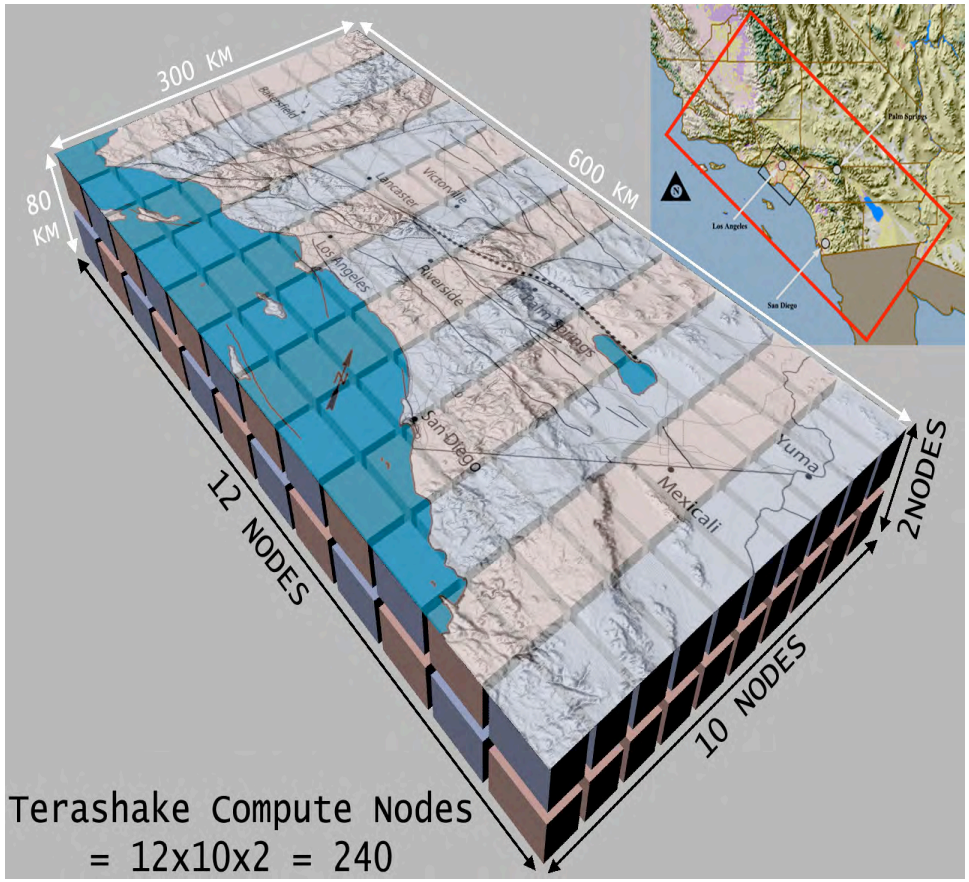
# Motivating Applications

# TeraShake

Simulates a 7.7 earthquake along the southern San Andreas fault near LA using seismic, geophysical, and other data from the Southern California Earthquake Center (240 IBM Power 4 processors for 5 days, 47TGB output)

How it works:

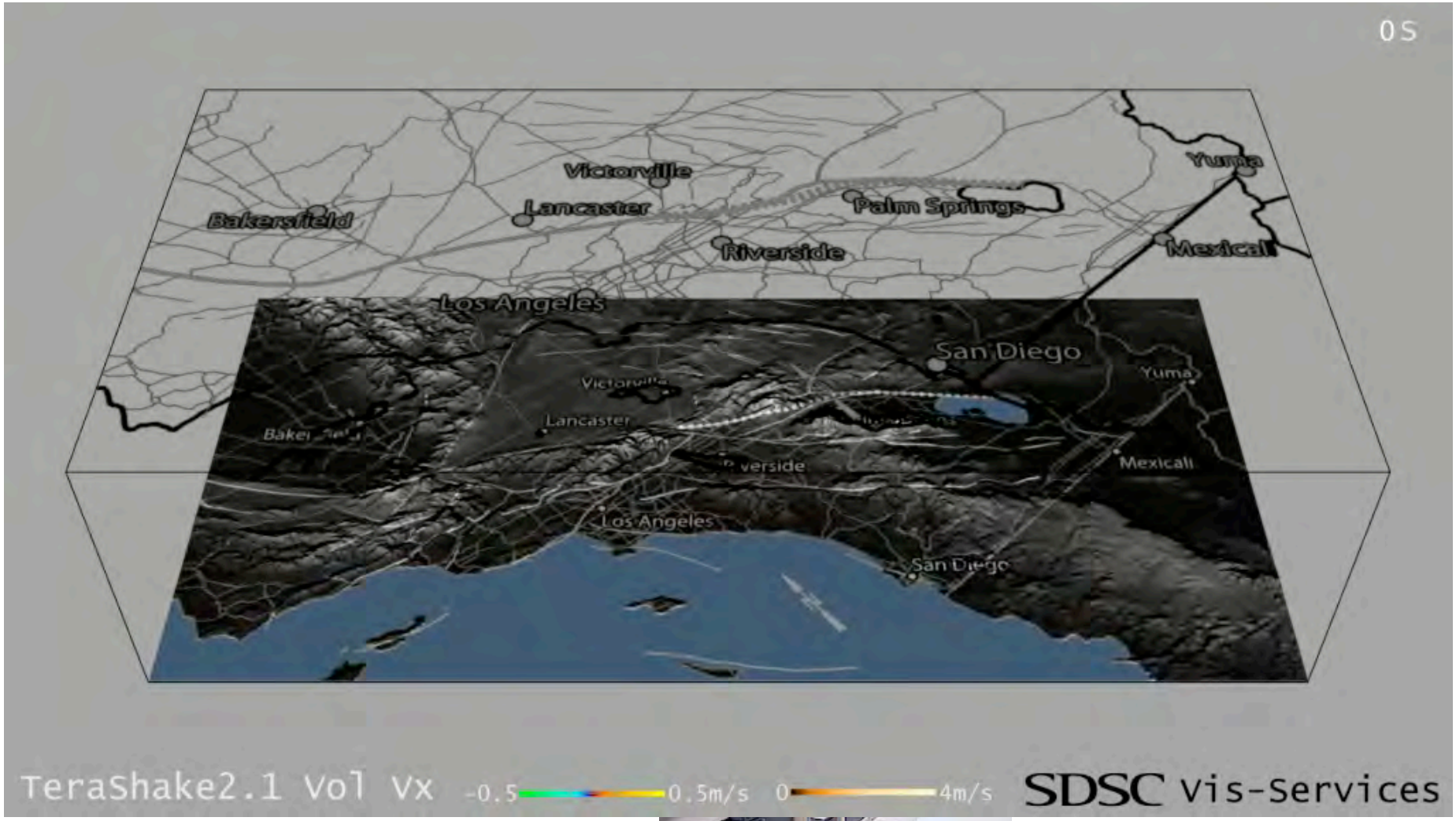
1. Divide up Southern California into "blocks"
2. For each block, get all the data on ground surface composition, geological structures, fault information, etc.






Slide Courtesy of DataCentral@SDSC

# How TeraShake Works



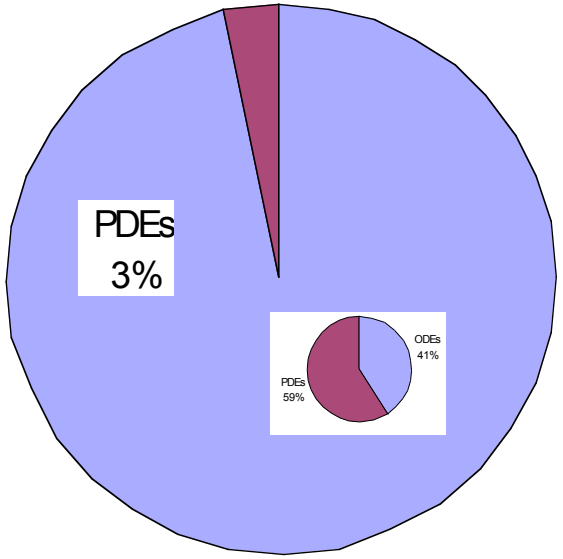

Slide Courtesy of DataCentral@SDSC

Room

SDSC's DataStar

# Application #2: modeling the heart on a GPU

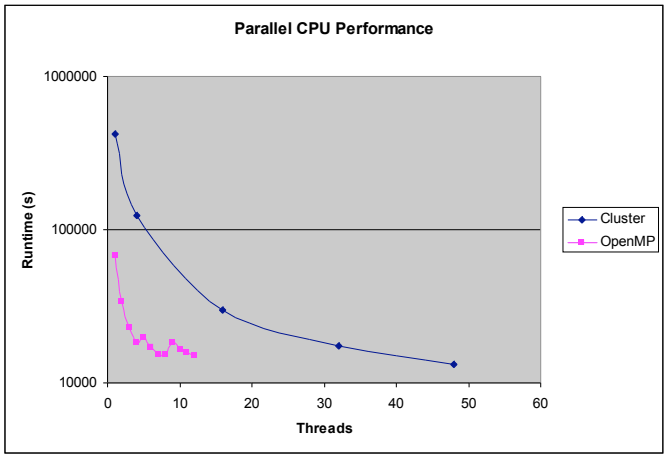
- Fred Lionetti and Andrew McCulloch
- nVidia GTX-295, single GPU: 240 single precision units @ 1.242 GHz
- ×55 speedup compared with openmp running on i7 (4-cores @ 2.93GHz with 12 GB RAM)

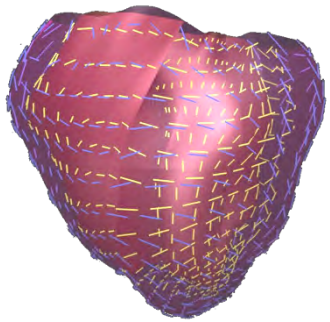



Cluster

Dual socket, AMD Opteron 2216 dual core processors @ 2.4 GHz, 4 GB of RAM

Programmed with MPI



# How do we know if we've succeeded?

- Capability
  - Solve a problem  under conditions that were not possible previously
- Performance
  - Solve the same problem in less time than before
  - This can provide a capability if we are solving many problem instances
- The result achieved must justify the effort
  - Enable new scientific discovery

# Available technologies and their impact on programming

# Parallel processing, concurrency, & distributed computing

- Parallel processing
  - Performance (and capacity) is the main goal
  - More tightly coupled than distributed computation, more frequent finer-grained interaction than distributed computation
- Concurrency
  - May or may not involve separate physical resources, e.g. multitasking "Virtual Parallelism"
  - Concurrency control: serialize certain computations to ensure correctness, e.g. database transactions
  - Performance need not be the main goal
- Distributed computation
  - Multiple unreliable resources communicating unreliably
  - Multiple launch sites, separate free standing programs
  - "Cloud" or "Grid" computing, large amounts of storage
  - Looser, coarser grained communication and synchronization

# Granularity

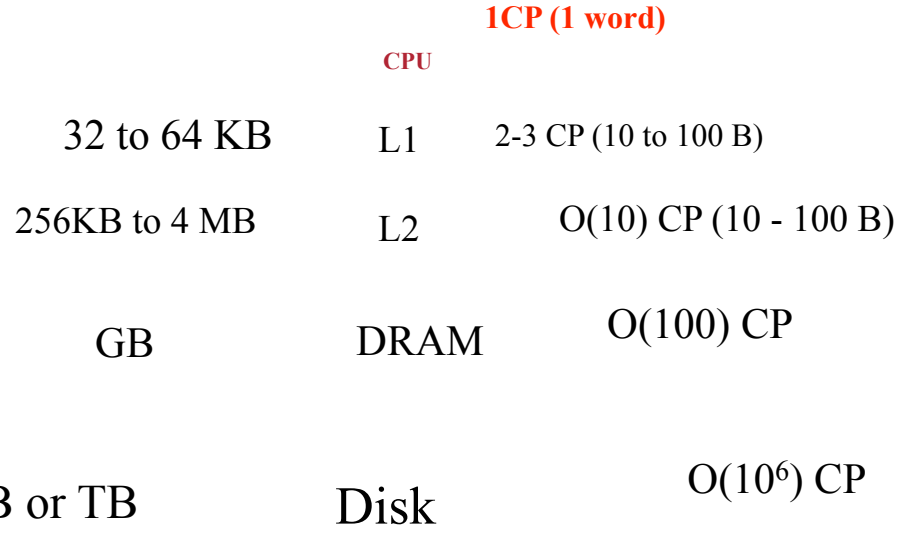
- A measure of how often a computation communicates, and what scale
  - Distributed computer: a whole program
  - Multicomputer: function, a loop nest
  - Multiprocessor: + memory reference
  - GPU: kernel thread
  - Instruction level parallelism: instruction, register

# Memory hierarchies

# An important universal: the locality principle

- Programs generally exhibit two forms of locality when accessing memory
  - Temporal locality (time)
  - Spatial locality (space)
- Often involves loops
- Opportunities for reuse

```
for t=0 to T-1
  for i = 1 to N-2
      u[i]= (u[i-1]  + u[i+1]) / 2
```

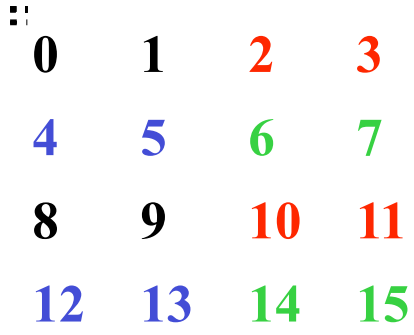


# Managing locality with loop interchange

- Data access order affects performance
- The success of caching depends on the ability to ***re-use*** previously cached data

```
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        a[i][j] += b[i][j];
```

```
for (j=0; j<N; j++)
    for (i=0; i<N; i++)
        a[i][j] += b[i][j];
```



# Testbed

- 2.7GHz Power PC G5 (970fx)
- http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/DC3D43B729FDAD2C00257419006FB955/$file/970FX_user_manual.v1.7.2008MAR14_pub.pdf
- Caches: 128 B line size
  - 512KB L2 (8-way, 12 CP hit time)
  - 32K L1 (2-way, 2 CP hit time)
- TLB: 1024 entries, 4-way
- gcc version 4.0.1 (Apple Computer, Inc. build 5370), -O2 optimization
- Single precision floating point

# Results

```
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        a[i][j] += b[i][j];
```

```
for (j=0; j<N; j++)
    for (i=0; i<N; i++)
        a[i][j] += b[i][j];
```

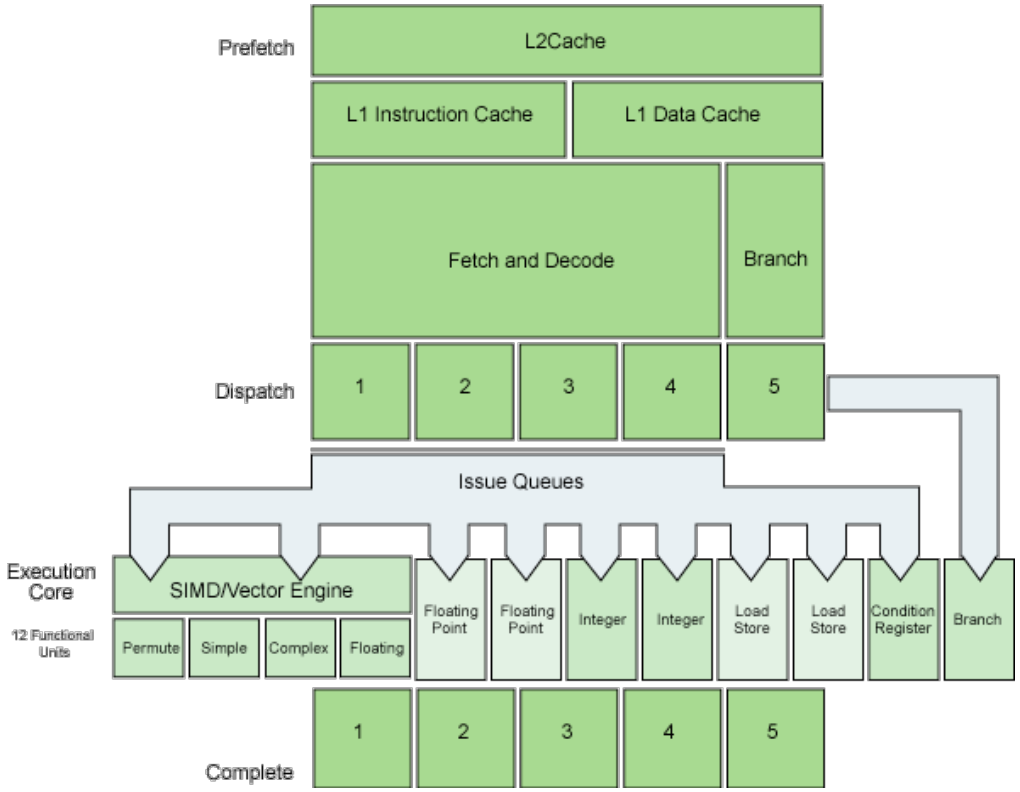
| N | IJ (ms) | JI (ms) | # Reps |
|---|---|---|---|
| 64 | 0.007 | 0.007 | $10^4$ |
| 128 | 0.027 | 0.083 | 10 |
| 512 | 1.1 | 37 | 10 |
| 1024 | 4.9 | 284 | 10 |
| 2048 | 18 | 2,090 | 10 |

| | | | |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 |
| 12 | 13 | 14 | 15 |

# Explaining the results

```
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
```

```
for (j=0; j<N; j++)
    for (i=0; i<N; i++)
```




| N | IJ (ms) | JI (ms) |
|---|---|---|
| 64 | 0.007 | 0.007 |
| 128 | 0.027 | 0.083 |
| 512 | 1.1 | 37 |
| 1024 | 4.9 | 284 |
| 2048 | 18 | 2,090 |

# Matrix multiplication

# Matrix Multiplication

- An important core operation in many numerical algorithms
- Given two *conforming* matrices A and B, form the matrix product A × B

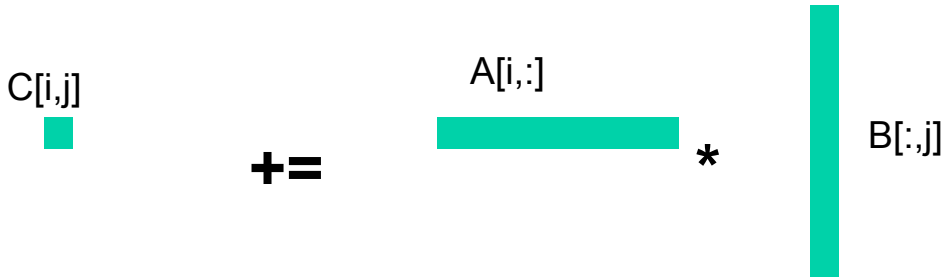
  A is m × n

  B is n × p
- Operation count: $O(n^3)$ multiply-adds for an n × n square matrix
- Discussion follows from Demmel

**www.cs.berkeley.edu/~demmel/cs267_Spr99/Lectures/Lect02.html**

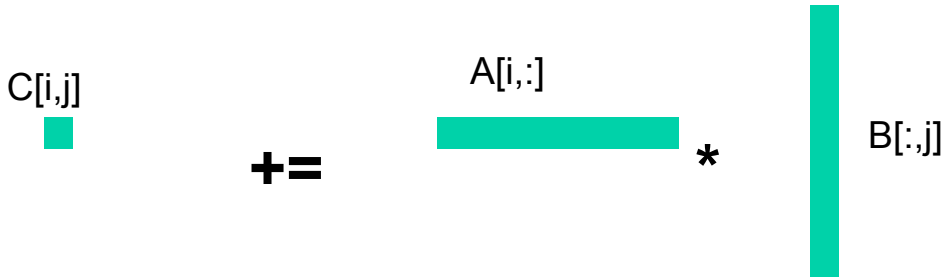
# Unblocked Matrix Multiplication

```
for i := 0 to n-1
    for j := 0 to n-1
      for k := 0 to n-1
            C[i,j] += A[i,k] * B[k,j]
```



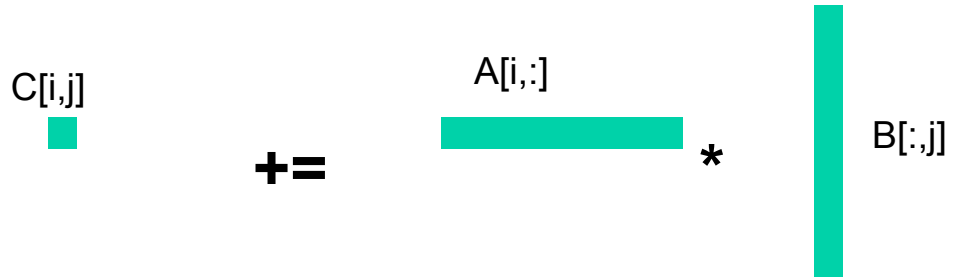
# Analysis of performance

```
for i = 0 to n-1
  // for each iteration i, load all of B into cache
   for j = 0 to n-1
      // for each iteration (i,j), load A[i,:] into cache
      // for each iteration (i,j), load and store C[i,j]
      for k = 0 to n-1
         C[i,j] += A[i,k] * B[k,j]
```



# Analysis of performance

```
for i = 0 to n-1
  // n × n^2 / b loads = n^3/b, b=cache line size   B[:,:]
  // n^2 / b loads = n^2/b                          A[i,:]
    for j = 0 to n-1
      //  n^2 / b loads + n^2 / b stores = 2n^2 / b   C[i,j]
      for k = 0 to n-1
          C[i,j] += A[i,k] * B[k,j]      Total:(n^3 + 3n^2) /b
```



# Flops to memory ratio

Let q = # flops / main memory reference

$$q = \frac{2n^3}{n^3 + 3n^2}$$

$\approx 2$ as $n \rightarrow \infty$

# Blocked Matrix Multiply

- Divide A, B, C into N × N sub blocks
- Each sub block is B × B
  - B=n/N is called the block size
  - how do we establish B?
  - assume we have a good quality library to perform matrix multiplication on subblocks

# Blocked Matrix Multiplication

```
for i = 0 to N-1
        for j = 0 to N-1
            // load each block C[i,j] into cache, once :                    n^2

            // B= n/N = cache line size
         for k = 0 to N-1
                // load each block A[i,k]  and B[k,j]  N^3 times
                //  = 2N^3 × (n/N)^2   =                                    2Nn^2
                  C[i,j] += A[i,k] * B[k,j]  // do the matrix multiply
             // write each block C[i,j] once :                               n^2
        Total:                                                             (2*N+2)*n^2
```

# Flops to memory ratio

Let q = # flops / main memory reference

$$q = \frac{2n^3}{(2N+2)n^2} = \frac{n}{N+1}$$

$$\approx n/N = \mathbf{b}$$

as $n \to \infty$

# The results

| N,B | Unblocked Time | Blocked Time |
|---|---|---|
| 256, 64 | 0.6 (0.035) | 0.002 |
| 512,128 | 15 (0.89) | 0.24 |

Amortize memory accesses by increasing memory reuse

# Programming Assignment #1

- Implement blocked matrix multiply
- Run on Triton
- Run on another machine, e.g. laptop
- Assignment will be posted this evening

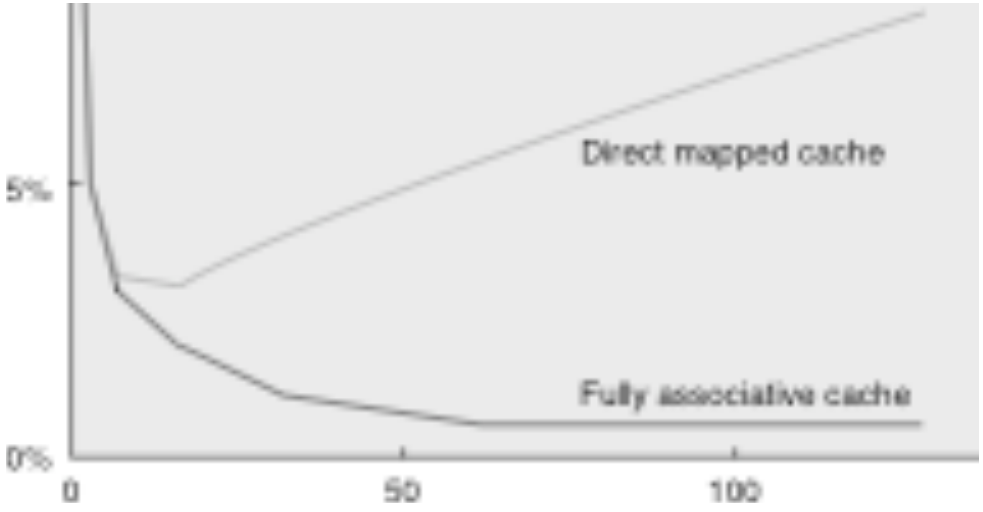
# More on blocked algorithms

- Data in the sub-blocks are contiguous within rows only
- We may incur  conflict cache misses
- Idea: since re-use is so high… let's copy the subblocks into contiguous memory before passing to our matrix multiply routine

"The Cache Performance and Optimizations of Blocked Algorithms," *ASPLOS IV*, 1991

http://www-suif.stanford.edu/papers/lam91.ps

**Miss rate**




**Blocking factor**