

# CSE 260 – Introduction to Parallel Computation

Topic 7: A few words about  
performance programming

October 23, 2001

# Announcements

- Office hours tomorrow: 1:30 – 2:50
- Highly recommended talk tomorrow (Weds) at 3:00 in SDSC's auditorium:  
    “Benchmarking and Performance Characterization  
    in High Performance Computing”

John McCalpin

IBM

For more info, see [www.sdsc.edu/CSSS/](http://www.sdsc.edu/CSSS/)

Disclaimer: I've only read the abstract, but it sounds relevant to this class and interesting.

# Approach to Tuning Code

- Engineer's method:

- ~~- DO UNTIL (exhausted)~~
- ~~- tweak something~~
- ~~- IF (better) THEN accept\_change~~

- Scientific method:

- DO UNTIL (enlightened)
- make hypothesis
- experiment
- revise hypothesis

# IBM Power3's power ... and limits

↖ Processor in Blue Horizon

- Eight pipelined functional units
  - 2 floating point
  - 2 load/store
  - 2 single-cycle integer
  - 1 multi-cycle integer
  - 1 branch
- Powerful operations
  - Fused multiply-add (FMA)
  - Load (or Store) update
  - Branch on count

Launch  $\leq 4$  ops per cycle

Can't launch 2 stores/cyc

FMA pipe 3-4 cycles long

Memory hierarchy speed

# Can its power be harnessed?

```
for (j=0; j<n; j+=4){  
    p00 += a[j+0]*a[j+2];  
    m00 -= a[j+0]*a[j+2];  
    p01 += a[j+1]*a[j+3];  
    m01 -= a[j+1]*a[j+3];  
    p10 += a[j+0]*a[j+3];  
    m10 -= a[j+0]*a[j+3];  
    p11 += a[j+1]*a[j+2];  
    m11 -= a[j+1]*a[j+2];  
}
```

8 FMA's

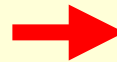
4 Loads

Runs at 4.6 cycles/iteration

( 1544 MFLOP/S on 444 MHz processor)

CL.6:

```
FMA  fp31=fp31,fp2,fp0,fc  
LFL  fp1=(*)double(gr3,16)  
FNMS fp30=fp30,fp2,fp0,fc  
LFDU fp3,gr3=(*)double(gr3,32)  
FMA  fp24=fp24,fp0,fp1,fc  
FNMS fp25=fp25,fp0,fp1,fc  
LFL  fp0=(*)double(gr3,24)  
FMA  fp27=fp27,fp2,fp3,fc  
FNMS fp26=fp26,fp2,fp3,fc  
LFL  fp2=(*)double(gr3,8)  
FMA  fp29=fp29,fp1,fp3,fc  
FNMS fp28=fp28,fp1,fp3,fc  
BCT          ctr=CL.6,
```



# Can its power be harnessed (part II)

- 8 FMA, 4 Loads: 1544 MFLOP/sec (1.15 cycle/load)

(previous slide)

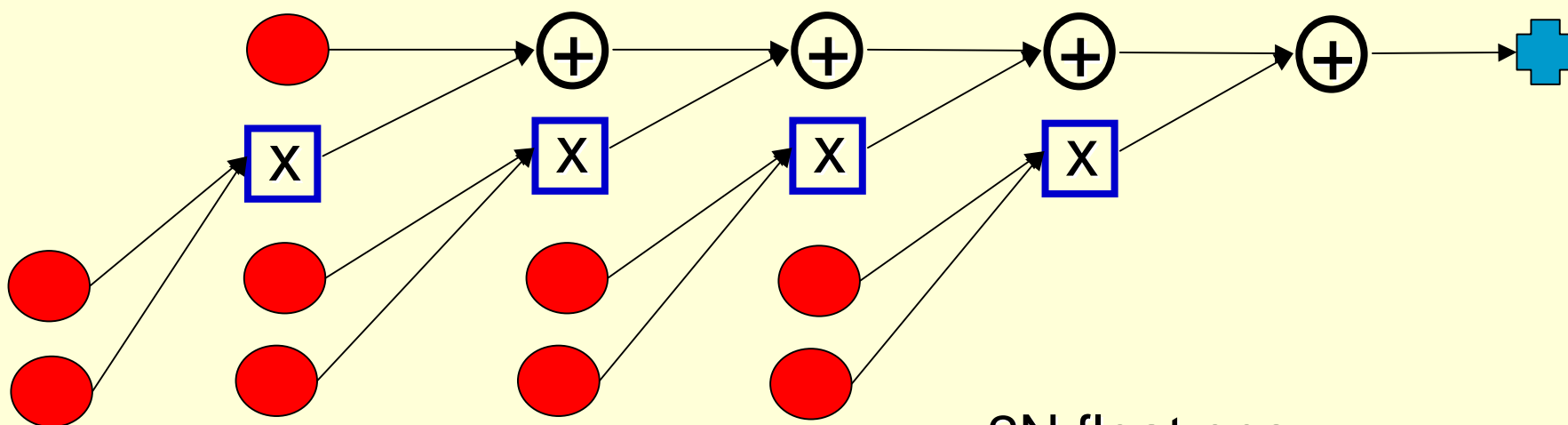
- 8 FMA, 8 Loads

```
for (j=0; j<n; j+=8){  
    p00 += a[j+0]*a[j+2];  
    m00 -= a[j+0]*a[j+2];  
    p01 += a[j+1]*a[j+3];  
    m01 -= a[j+1]*a[j+3];  
    p10 += a[j+4]*a[j+6];  
    m10 -= a[j+4]*a[j+6];  
    p11 += a[j+5]*a[j+7];  
    m11 -= a[j+5]*a[j+7]; }
```

- 1480 MFLOP/sec (0.6 cycle/load)
- "Interactive" node: 740 MFLOP/sec (1.2 cycle/load)
  - Interactive nodes have 1 cycle/MemOp barrier!
  - the *AGEN* unit is disabled @!#%\$

## A more realistic computations

Dot Product (DDOT):  $z += \sum x[i]*y[i]$



2N float ops

2N+2 load/stores

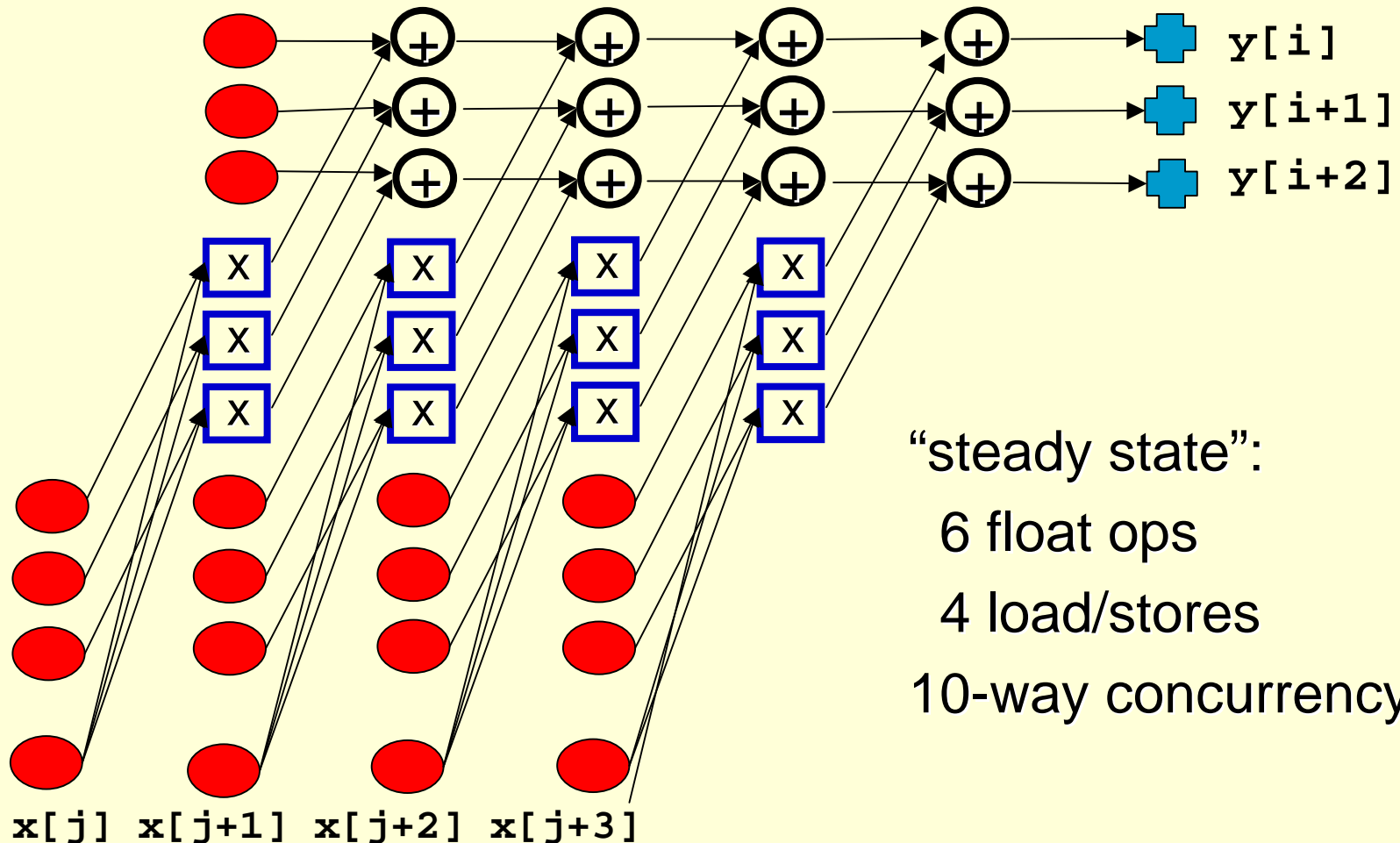
4-way concurrency

● = load

■ = store

# Optimized matrix x vector product:

$$\mathbf{y} = \mathbf{A} \mathbf{x}$$



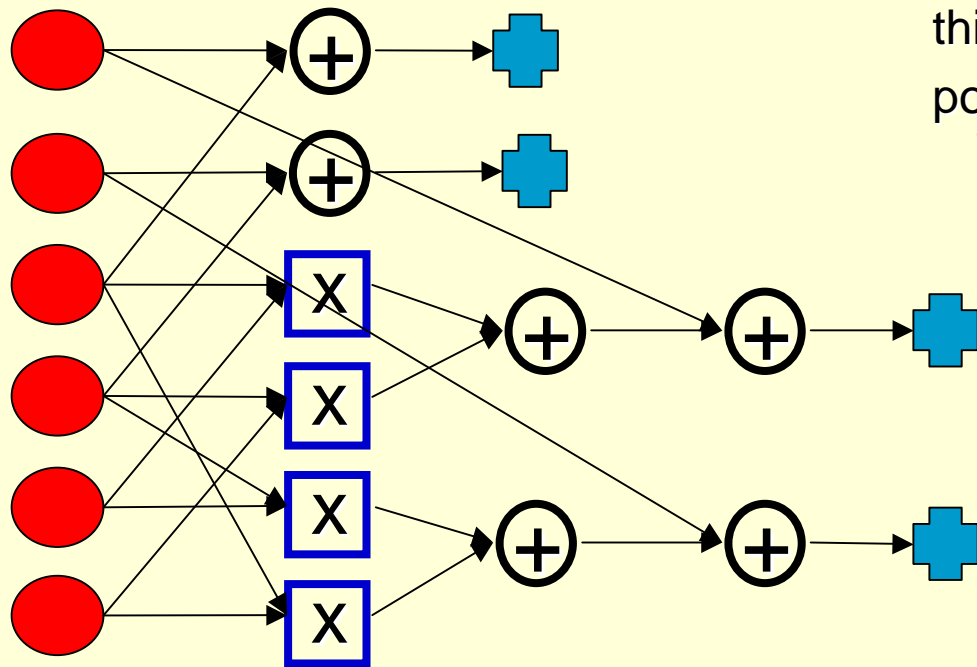
“steady state”:

6 float ops

4 load/stores

10-way concurrency

# FFT "butterfly"



Note: on typical processor,  
this leaves half the ALU  
power unused.

10 float ops

10 load/stores

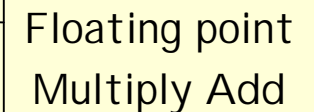
4-way concurrency

● = load      ⊕ = store

# FLOP to MemOp ratio

- Most programs have at most one FMA per MemOp
  - DAXPY ( $Z[i] += A X[i] + Y[i]$ ): 2 Loads, 1 Store, 1 FMA
  - DDOT ( $Z += \sum X[i] Y[i]$ ): 2 Loads, 1 FMA
  - Matrix-vector product:  $(k+1)$  loads,  $k$  FMA's
  - FFT butterfly: 8 MemOps, 10 floats (5 or 6 FMA)
- A few have more (but they are in libraries)
  - Matrix multiply (well-tuned): 2 FMA's per load
  - Radix-8 FFT
- Your program is probably limited by loads and stores!

Floating point  
Multiply Add



# Need independent instructions

- Remember Little's Law!

4 ins/cyc x 3 cycle pipe → need 12-way independence

Many recent and future processors need even more.

- Out-of-order execution helps.

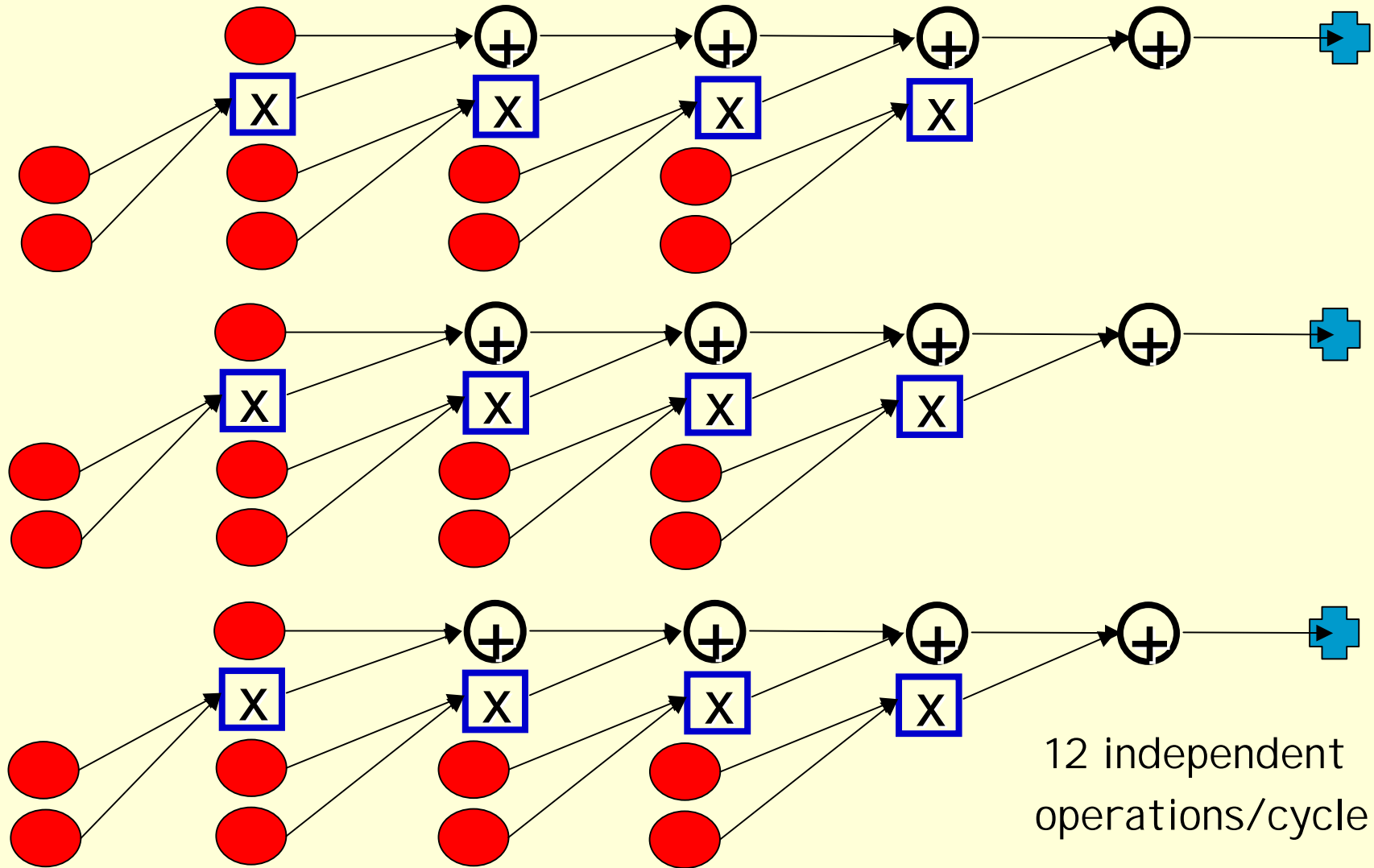
But limited by instruction window size & branch prediction.

- Compiler unrolling of inner loop also helps.

Compiler has inner loop execute, say, 4 points, then interleaves the operations.

Requires lots of registers.

# How unrolling gets more concurrency



# Improving the MemOp to FLOP Ratio

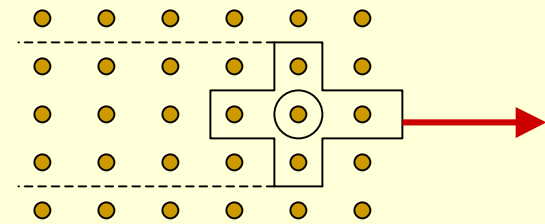
```

for (i=1; i<N; i++)
  for (j=1; j<N; j++)
    b[i,j] = 0.25 *
      (a[i-1][j] + a[i+1][j]
       + a[i,j-1] + a[i][j-1]);
  
```

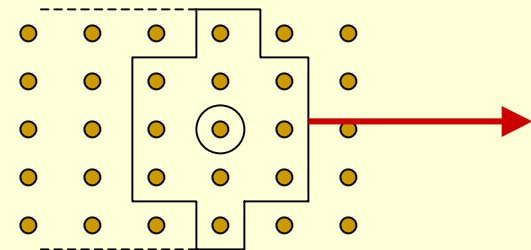


```

for (i=1; i<N-2; i+=3) {
  for(j=1; j<N; j++) {
    b[i+0][j] = ... ;
    b[i+1][j] = ... ;
    b[i+2][j] = ... ;
  }
}
for (i = i; i < N; i++) {
  ... ; /* Do last rows */
}
  
```



3 loads / 4 floats  
1 store



5 loads / 12 floats  
3 store

# Overcoming pipeline latency

```
for (i=0; i<size; i++) {  
    sum = a[i] + sum;  
}
```

→ 3.86 cycles/addition

Next add can't start until previous is finished (3 to 4 cycles later)

---

```
for (i=0; i<size; i+=8) {  
    s0 += a[i+0]; s4 += a[i+4];  
    s1 += a[i+1]; s5 += a[i+5];  
    s2 += a[i+2]; s6 += a[i+6];  
    s3 += a[i+3]; s7 += a[i+7];  
}  
sum = s0+s1+s2+s3+s4+s5+s6+s7;
```

→ 0.5 cycle/addition

May change answer due to different rounding.

# The SP Memory Hierarchy

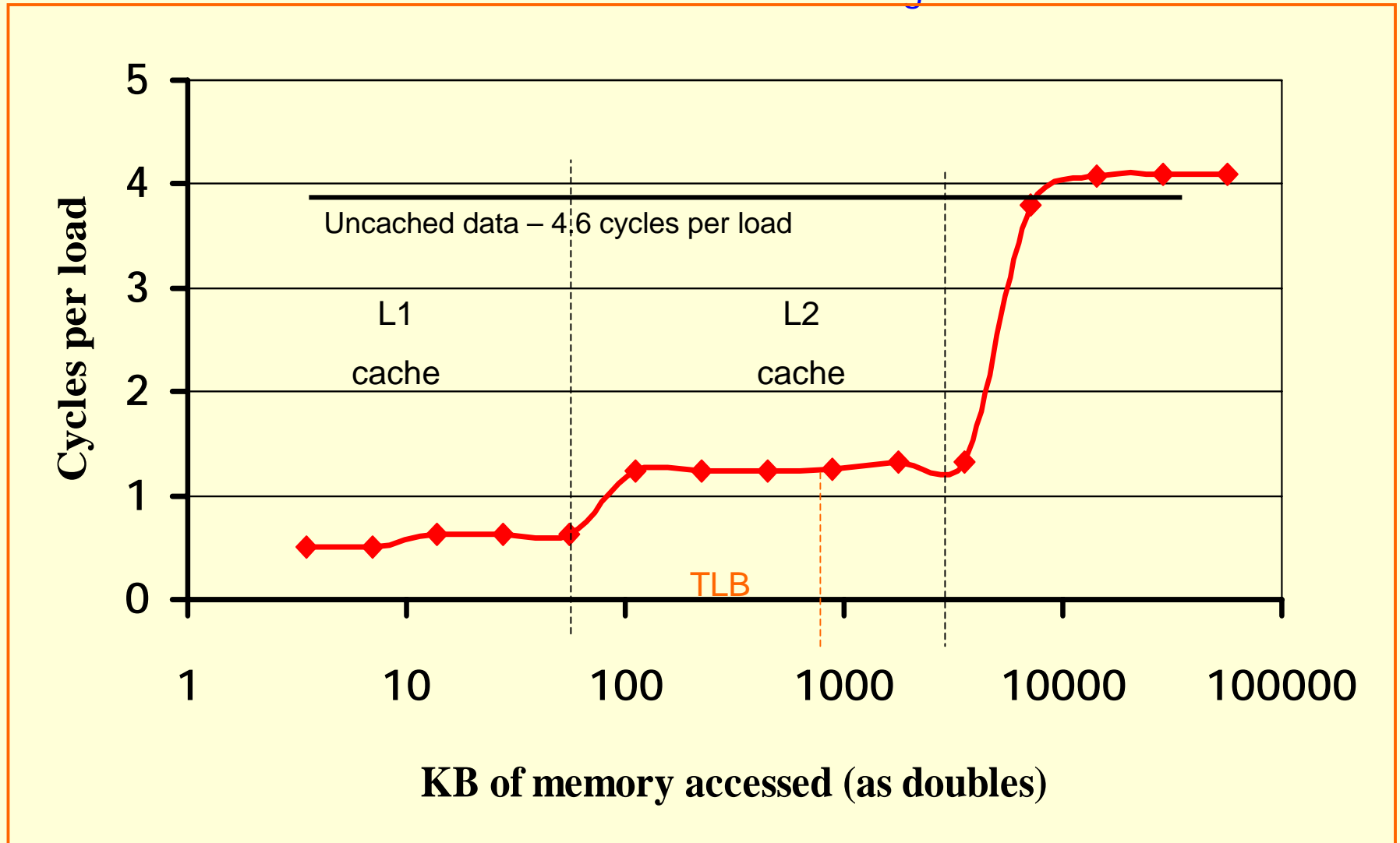
- L1 data cache
  - 64 KBytes = 512 lines, each 128 Bytes long
  - 128-way set associative (!!)
  - Prefetching for up to 4 streams
  - 6 or 7 cycle penalty for L1 cache miss
- Data TLB
  - 1 MB = 256 pages, each 4KBytes
  - 2-way set associative
  - 25 to 100's cycle penalty for TLB miss
- L2 cache
  - 4 MByte = 32,768 lines, each 128 Bytes long
  - 1-way (4-way on Nighthawk 2) associative, randomized (!!)
  - Only can hold data from 1024 different pages
  - 35 cycle penalty for L2 miss

## So what??

- Sequential access (or small stride) are good
- Random access within a limited range is OK
  - Within 64 KBytes – in L1; 1 cycle per MemOp
  - Within 1 MByte – up to 7-cycle L1 penalty per 16 words (prefetching hides some cache miss penalty)
  - Within 4 MByte – May get 25 cycle TLB penalty
  - Larger range – huge (80 – 200 cycle) penalties

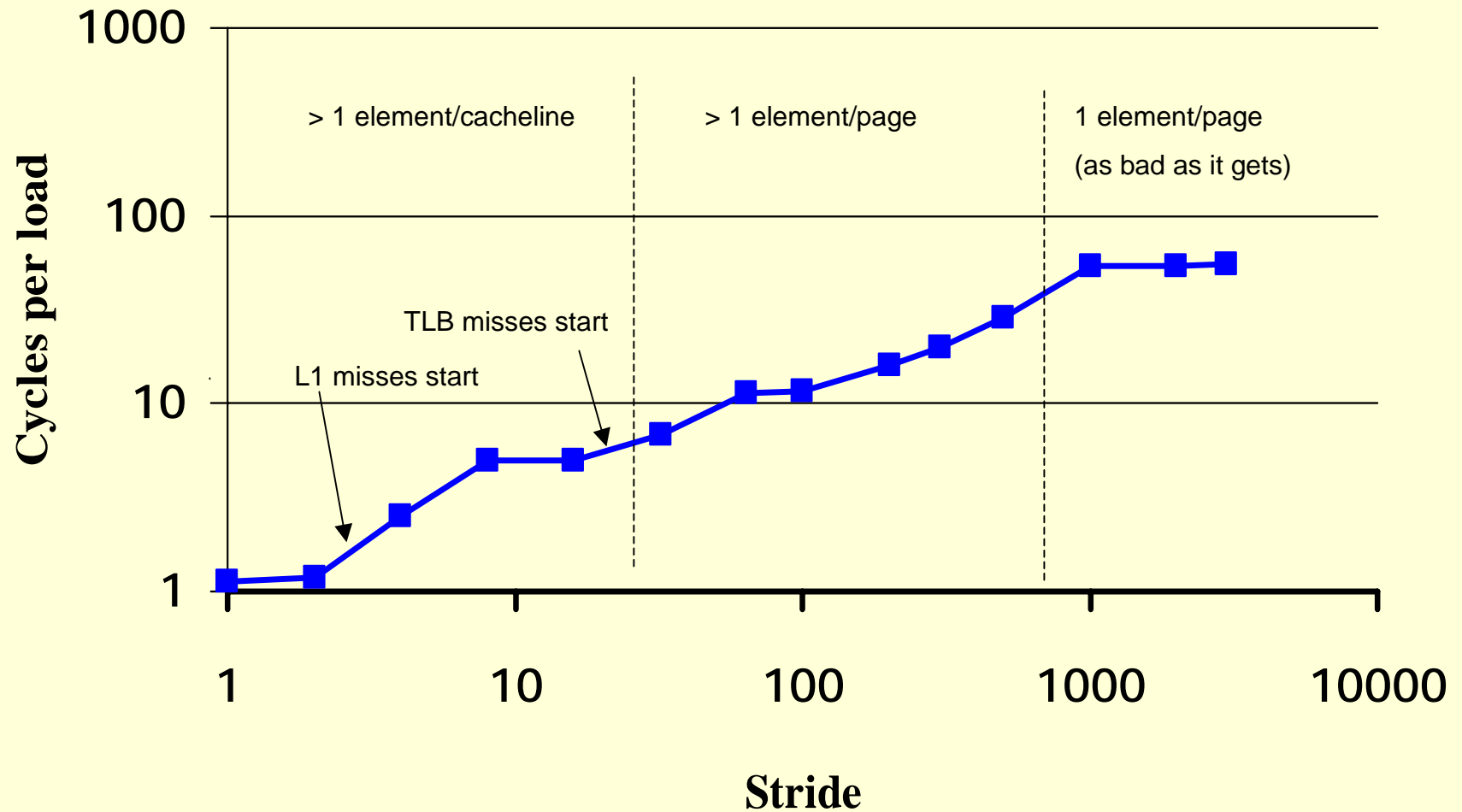
# Stride-one memory access

sum list of floats  
times for second time through data



# Strided Memory Access

Program adds 4440 4-Byte integers located at given stride  
(performed on “interactive node”)



# Sun E10000's Sparc II processor

See [www.sun.com/microelectronics/manuals/ultrasparc/802-7220-02.pdf](http://www.sun.com/microelectronics/manuals/ultrasparc/802-7220-02.pdf)

400 MHz on ultra, 336 MHz on gaos

9 stage instruction pipe (3 stage delay due to forwarding)

Max of 4 instructions initiated per cycle

At most two floats/cycle (no FMA instruction)

Max of one load or store per cycle

16 KB Data cache (+ 16 KB I cache), 4 MB L2 cache

64 Byte line size in L2.

Max bandwidth to L2 cache: 16 Bytes/cycle (4 cycles/line)

Max bandwidth to memory: 4 Bytes/cycle (16 cycles/line)

Shared among multiple processors

Can do prefetching

# Reading Assembly Code

- Example code:

```
do i = 3, n
    v(i) = M(i,j)*v(i) + v(i-2)
    IF (v(i).GT.big) v(i) = big
enddo
```

- Compiled via "f77 -fast -S"
  - On Sun's Fortran 3.0.1 - this is 7 years old
- Sun idiosyncrasies:
  - Target register is given last in 3-address code
  - Delayed branch jumps after *following* statement
  - Fixed point registers have names like %o3, %l2, or %g1.

# SPARC Assembly Code

```
.L900120:
ldd      [%l2+%o1],%f4      Load V(i) into register %f4
fmuld    %f2,%f4,%f2      Float multiply double
ldd      [%o5+%o1],%f6      Load V(i-2) into %f6
fadd     %f2,%f6,%f2
std      %f2,[%l2+%o1]     Store %f2 into V(i)
ldd      [%l2+%o1],%f4     Reload it (!?!)
fcmped   %f4,%f8          Compare to "big"
nop
fbg,a    .L77015          Float branch on greater
std      %f8,[%l2+%o1]     Conditionally store "big"
.L77015:
add      %g1,1,%g1        Increment index variable
cmp      %g1,%o7          Are we done?
add      %o0,8,%o0        Increment offset into M
add      %o1,8,%o1        Increment offset into V
ble,a    .L900120        "Delayed" branch
ldd      [%l1+%o0],%f2     Load M(i,j) into %f2
```