Single-Node Optimization

Larry Carter
UCSD & SDSC

August 1998
Contents

Tools of performance programming
- Timing and profiling
- Visualizing computation with the ISG
- Reading (not writing) assembly code
- Tiling for locality and parallelism

Real-life examples
- Seismic migration
- UWTJ (from CFD code)
- Matrix-vector product
- FFT

Messages
- Measure cycles/iteration
- Get the inner loop right
- Improve locality of references
- Avoid thrashing
- 2x speed improvement is sometimes easy
Performance Measures

CPU time

User time: instructions, cache & TLB misses
System time: initiating I/O & paging, exceptions
Low resolution (typically 1/100 second)
   For higher resolution, time N iterations, divide by N

Good for whole programs, shared system

Wallclock time (real time, elapsed time)

Above + waiting for I/O, communication, other jobs
High resolution (e.g. 1 usec), but variable
To get reliable wallclock times ...
... run on dedicated machine
... take 5 timings, use minimum (not average)
   First run may be slow due to acquiring page frames
   Random runs will be affected by operating system

Good for inner loop and I/O timing
Fortran 90

integer starttime, stoptime, tick
call system_clock(count_rate = tick)
    ... 
call system_clock(count = starttime)
    ...timed section...
call system_clock(count = stoptime)
time = real(stoptime-starttime) / real(tick)

MPI

Fortran: DOUBLE PRECISION MPI_WTIME()
C: double MPI_Wtime()

MPI_WTIME_IS_GLOBAL
    if 1, then time before send ≤ time after receive
    if 0, then no guaranteed clock synchronization

MPI_Wtick() returns clock resolution

Other wallclock timers:

gettimeofday, ftime, rtc, get timer, ...
CPU Time

Timing entire execution

time or timex or ...

depending on shell and Unix flavor
Gives user, system & wallclock time, maybe more

Timing segments of code

ANSI C standard library

#include <times.h>
clock_t is type of CPU times
clock()/CLOCKS_PER_SEC

Other CPU timers

getrusage (also gives paging, I/O ...)
etime, etime_, dtime, mclock, ...
More About Timing

To time one loop iteration:

Time program with $2K$ iterations of loop spend at least 1 second in inner loop.

Time program with $K$ iterations

Subtract CPU times, divide by $K$

Code initialization time will cancel out

Alternate method:

Get wallclock time before and after statements

Time short segments to reduce chance of interrupt
But time at least 20x timer’s resolution
Same trick can cancel timer’s overhead
Don’t average; discard outlandishly large times

Highly Recommended:

Compute loop times in cycles/iteration

Compare to plausible estimate
Identifies unexpected problems
Unexpected Performance Problem
(Example from high-performance computing text)

\[
\begin{align*}
  &\text{do 100 } \text{K = 1, 64} \\
  &\quad \text{do 100 } \text{J = 1, 64} \\
  &\qquad \text{do 100 } \text{I = 1, 64} \\
  &\quad 100 \quad A(I,J,K) = B(I,J,K) + C(I,J,K)
\end{align*}
\]

Required .68 seconds on IBM RS6000/320H

25 MHz processor, 262K iterations → 65 cycles/it

This is terrible performance!

The book’s authors never noticed!!

Probable explanation:

A, B, and C are adjacent, 64^3-element arrays
All three use aligned TLB associativity class
TLB is 2-way set associative
Every iteration involves TLB misses

It’s easy to fix (when you know what’s happening)

Displace or enlarge power-of-2 matrices
Even A = B then A = A+C would be much faster

Morals:

Compute cycles/iteration
Beware of large powers of two
Profiling

To get time spent in each procedure:

Compile with `-pg` option (inserts mcount)
  Use `-G` option on HP machines

Run program (creates `gmon.out`)

Type `gprof foo >foo.profile` (creates profile)

To accumulate results from several runs:

Rename `gmon.out`'s as `gmon.1`, `gmon.2`, etc

Do `gprof foo gmon.1 gmon.2 >foo.profile`

T3E-specific tools:

`pat` - low overhead, no recompile
  Use `-lpat pat.cld` in load command
  Run program
  Invoke pat to query cycles, operations, cache misses,...

`apprentice` - full functionality
  Includes source-code browser

See www.npaci.edu/T3E/t3e.html for details.
Visualizing Computation

Iteration Space Graph

Nodes

For each input value

For each loop iteration (or value computed)

Directed edges (arrows) show dependences

Example

\[
\text{for } i = 0 \text{ to } T-1 \\
\quad \text{for } j = 0 \text{ to } S-1 \\
\quad A[j+1] = u \cdot A[j] + v \cdot A[j+1]
\]
**CFD example**

Total time: 152.99 seconds

<table>
<thead>
<tr>
<th>%time</th>
<th>seconds</th>
<th>calls</th>
<th>ms/call</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>33.4</td>
<td>51.08</td>
<td>1024</td>
<td>49.88</td>
<td>.uwtj</td>
</tr>
<tr>
<td>15.6</td>
<td>23.90</td>
<td>18084</td>
<td>1.32</td>
<td>.vpassm</td>
</tr>
<tr>
<td>6.1</td>
<td>9.37</td>
<td>192</td>
<td>48.80</td>
<td>.dipsxz</td>
</tr>
<tr>
<td>5.7</td>
<td>8.72</td>
<td>16</td>
<td>545.00</td>
<td>.h3dpsd</td>
</tr>
<tr>
<td>5.7</td>
<td>8.68</td>
<td>1024</td>
<td>8.48</td>
<td>.vtj</td>
</tr>
<tr>
<td>4.3</td>
<td>6.58</td>
<td></td>
<td></td>
<td>__mcount</td>
</tr>
<tr>
<td>3.4</td>
<td>5.15</td>
<td></td>
<td></td>
<td>.IOWrite</td>
</tr>
<tr>
<td>2.5</td>
<td>3.83</td>
<td></td>
<td></td>
<td>.WriteUnit</td>
</tr>
<tr>
<td>2.1</td>
<td>3.15</td>
<td></td>
<td></td>
<td>__sqrt</td>
</tr>
<tr>
<td>2.0</td>
<td>3.11</td>
<td>3212</td>
<td>0.97</td>
<td>.cfft99</td>
</tr>
<tr>
<td>1.8</td>
<td>2.72</td>
<td>1534940</td>
<td>0.00</td>
<td>__sin</td>
</tr>
<tr>
<td>1.7</td>
<td>2.55</td>
<td></td>
<td></td>
<td>__xlfWriteUfm</td>
</tr>
<tr>
<td>1.5</td>
<td>2.30</td>
<td>1326090</td>
<td>0.00</td>
<td>__cos</td>
</tr>
<tr>
<td>1.4</td>
<td>2.17</td>
<td>1024</td>
<td>2.12</td>
<td>.newj</td>
</tr>
<tr>
<td>1.0</td>
<td>1.49</td>
<td>1536</td>
<td>0.97</td>
<td>.fft99b</td>
</tr>
</tbody>
</table>

(88% so far)

**Note:** mcount is gprof overhead
INNER LOOPS FROM UWTJ

DO I = 2, IM+1
  DO K = 2, KM+1
    RUN = FUX(I+1, K) - FUX(I, K) + FVX(I, K) - FVXM(I, K) +
    FUZ(I, K) - FUZ(I, K-1) - Q4UGR*
    (W(I, K-1) + W(I+1, K-1) + W(I, K) + W(I+1, K))
    US(I, K) = U(I, K) + G1DT*RUN + G2DT*RU(I, K)
    RU(I, K) = RUN
    RWZ(K) = FUZ(I, K) - FUZ(I-1, K) + FVZ(I, K) - FVZM(I, K) +
    FWZ(I, K+1) - FWZ(I, K) + RIQ2*(T(I, K+1) + T(I, K))
    RTZ(K) = FTX(I, K) - FTX(I-1, K) + FTY(I, K) - FTYM(I, K) +
    FTZ(I, K) - FTZ(I, K-1) - (W(I, K) + W(I, K-1)) * TG2
  END
  DO K = 2, KM+1
    WS(I, K) = W(I, K) + G1DT*RWZ(K) + G2DT*RW(I, K)
    RW(I, K) = RWZ(K)
    TN(I, K) = T(I, K) + G1DT*RTZ(K) + G2DT*RT(I, K)
    RT(I, K) = RTZ(K)
  END
END
END
ISG for uwtj Loops

- First loop iteration
- Second loop iteration
- Input data

Original code executes row-wise
Against grain of matrices
Requires array for RWZ and RTZ
DO K = 2,KM+1 
  DO I = 2,IM+1 
    RUN=FUX(I+1,K)-FUX(I,K)+FVX(I,K)-FVXM(I,K)+ 
    . 
    . 
    FUZ(I,K)-FUZ(I,K-1) - Q4UGR* 
    . 
    (W(I,K-1)+W(I+1,K-1)+W(I,K)+W(I+1,K)) 
    US(I,K) =U(I,K) + G1DT*RUN + G2DT*RU(I,K) 
  RU(I,K)=RUN 
  RWZ=FUZ(I,K)-FUZ(I-1,K)+FVZ(I,K)-FVZM(I,K)+ 
    . 
    . 
    FWZ(I,K+1)-FWZ(I,K)+RIQ2*(T(I,K+1)+T(I,K)) 
  RTZ=FTX(I,K)-FTX(I-1,K)+FTY(I,K)-FTYM(I,K)+ 
    . 
    . 
    FTZ(I,K)-FTZ(I,K-1)-(W(I,K)+W(I,K-1))*TG2 
  WS(I,K) =W(I,K) + G1DT*RWZ + G2DT*RW(I,K) 
  RW(I,K)=RWZ 
  TN(I,K) =T(I,K) + G1DT*RTZ + G2DT*RT(I,K) 
  RT(I,K)=RTZ 
END 
END 

**Improvement from 659 to 112 cycles/iteration**

3-fold speedup of uwtj

28 load/store + 30 float instructions/iteration

About 3 cache misses per iteration

Pipeline delays remain

**Moral: Old vector code may have easy problems.**
Reading Assembly Code

Get listing via -S compiler option

Use -qsource -qlist on IBM’s

Learn conventions

Is first or last register the target?
Are there delayed branches?

How to use it:

Locate innermost loop

Check off statements that belong
  Arithmetic on program variables, necessary loads & stores

Be suspicious of residue. Typical problems:
  Extra store-load’s (compiler thinks arrays might be aliased)
  Spill code (loop too big for number of registers)
  Excessive address calculations
  Unnecessary type conversions

Do dependent ops have independent ones in between?
Example Assembly Code
Each example is for the code:

\[
\text{do } i = 3, n \\
\quad V(i) = M(i,j)*V(i) + V(i-2) \\
\quad \text{IF (} V(i).GT.\text{big) } V(i) = \text{big} \\
\text{enddo}
\]

Outputs are edited & annotated

Sun SPARC example:

- Compiled via: f77 -fast -S
- SUN's FORTRAN 3.0.1 compiler
- Execution about 20 cycles/iteration

IBM RS/6000 example:

- Compiled via: xlf -O2 -qsourse -qlist
- IBM's XL Fortran 3.02.0 compiler
- Execution about 14 cycles/iteration

DEC Alpha example:

- Compiled via f77 -O5 -unroll 1 -V -show code
- DEC Fortran V3.5-576 compiler
Sun SPARC Example

.L900120:

ldd [%12+%01],%f4   Load V(i) into %f4
fmuld %f2,%f4,%f2   Float multiply
ldd [%05+%01],%f6   Load V(i-2) into %f6
faddi %f2,%f6,%f2   Float multiply
std %f2,[%12+%01]   Store %f2 into V(i)
ldd [%12+%01],%f4   Reload it (!?!)  
fcmped %f4,%f8     Compare to "big"
nop

fbg,a .L77015       Float branch on greater

std %f8,[%12+%01]   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 [%11+%00],%f2  Load M(i,j) into %f2

"[x]" means memory location at address x.
"%o1" means contents of register o1.

Target register is given last.

The statement after a "delayed branch" is executed unconditionally along with the branch.
IBM RS/6000 Example

CL.136:
1 BF CL.137,cr1,0x20/flt  Branch if V(i-2).LE.big
3 STFL v(gr5,0)=fp8  Conditionally store big

CL.137:
  CFL cr1=fp8,fp1  Compare V(i-1) to big
2 STFDU gr5,v(gr5,8)=fp1  Store fp1 & update reg 5
1 LFL fp1=v(gr5,8)  Load V(i)
1 LFL fp6=v(gr5,-8)  Load V(i-2)
1 FMA fp1=fp6,fp7,fp1,fcr  Float multiply-add
0 LFDU fp7,gr7=m(gr7,8)  Load M(i,j), update reg 7
0 BCT ctr=CL.136  count--, branch on plus

"v(gr5,8)" means 8 + value in reg 5, and this happens to point to the user’s "v" array.

load and store instructions ending in "U" increment the index register as well as loading or storing.

Numbers in first column (4th in real listing) is guess of how many cycles the instruction will take.

Target register is first register mentioned.

Pipelining: first two statement complete the "IF" started by the CFL in previous iteration, which itself compares "big" to a value computed still earlier.
DEC Alpha Example

L$10:

ldt  f11, (r20)  Load M(i,j)
lda  r20, 8(r20)  Increment M pointer
ldt  f12, (r19)  Load V(i)
cmpule r20, r17, r23  Is this last address?
ldt  f13, -16(r19)  Load V(i-2)
da  r19, 8(r19)  Increment V pointer
ldah r18, 897(r31)  Junk? ld upper half r18
ldq  r27, 16(gp)  Junk? (gp = global ptr)
mult f11, f12, f11  Float multiply
lda  r18, -256(r18)  Junk?
addt f11, f13, f11  Add V(i-2) to f11
cmptle f11, BIG, f14  Is result <= BIG
fcmoveq f14, BIG, f11  If not, move BIG to f11
nop
stt  f11, -8(r19)  Store V(i)
bne  r23, L$10  Loop back

Target register is last (in 3-address code).
"8(r20)" means 8 + value in reg 20. r31 always 0.

r19 is pointer to V(i);  r20 = pointer to M(i,j)
r23 and f14 hold results of comparisons.
Conditional move avoids need for branch.
Loop ends when r20 reaches precomputed value.
Seismic Migration Inner Loops

Symmetric filter:

\[ p(i_{x-19}) \quad p(i_{x}) \quad p(i_{x+19}) \]

\[ e_{19} \quad e_{18} \quad e_{0} \quad e_{18} \quad e_{19} \]

Each point is weighted average of 39 points above it

In the same \[ xz \] plane

Inner loop:

\[
\text{for (ix=0; ix<nx; ix++)} \{ \\
\quad k = \ldots \quad \% \text{ filter depends on Velocity & w} \\
\quad q = \text{filter}[k][0] \time p[ix]; \\
\quad \text{for (j=1; j<20; j++)} \{ \\
\quad \quad s = p[ix-j] + p[ix+j]; \\
\quad \quad e = \text{filter}[k][j]; \\
\quad \quad q += e \times s; \\
\quad \} \\
\}\]
The Inner Loop

Complex arithmetic: \( q \ = \ e*s \) is
\[
qr \ = \ er*sr - ei*si; \ qi \ = \ er*si + ei*sr
\]

Inner Loop Operation Count:

<table>
<thead>
<tr>
<th>Statement</th>
<th>loads</th>
<th>float +</th>
<th>float x</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s=p[ix-j]+p[ix+j] )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( e=filter[k][j] )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( q+=e*s )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Features of the IBM RISC System 6000:
- **load-update** handles indexing (e.g. \( p[ix-j] \))
  - Branch unit handles the loop test
  - Floating-point multiply-add (fma) initiated each cycle
    - loads and fmas overlap
  - (Note: SP2 has 2 FXU’s and 2 FPU’s)

Prediction: ____ cycles per iteration

© Alpern & Carter 1998
RS/6000 Organization

MEMORY

40

TLB

8

CACHE

1

5

1

FX REG

FP REG

CR

3

7

BRANCH

FXU

addr comp

int arith

FPU

fp arith

© Alpern & Carter 1998
Closing The Gap

Predicted performance:  6 cycles per iteration
Actual performance:    13 cycles per iteration
What happened?
  Unnecessary CVLS instructions
  Converts to from double to single-precision
Why?
  RS/6000 processor internally double-precision
  s and q were declared float
  ANSI standard requires single-precision arithmetic
    User doesn’t care which precision is used
CVLS  easy to eliminate  (if you know they’re there)
  Declare temporaries (s, e, and q) double, or
  Use -qfloat=hsflt compiler option
Resulting performance:  9 cycles per iteration
 Inner Loop Assembly Code

CL.6: AFL fp1=fp12,fp5,fcr
LFSU fp12,gr9=(*)float(gr9,-4)
LFSU fp5,gr3=(*)float(gr3,4)
AFL fp0=fp4,fp9,fcr
LFSU fp4,gr11=(*)float(gr11,-4)
CVLS fp2=fp1,fcr
LFSU fp9,gr10=(*)float(gr10,4)
CVLS fp0=fp0,fcr
MFL fp1=fp11,fp2,fcr
MFL fp3=fp11,fp0,fcr
LFSU fp11,gr12=filterI[]0(gr12,4)
FMA fp0=fp1,fp13,fp0,fcr
FMS fp1=fp3,fp13,fp2,fcr
LFSU fp13,gr8=filterR[]0(gr8,4)
AFL fp0=fp10,fp0,fcr
AFL fp1=fp8,fp1,fcr
CVLS fp10=fp0,fcr
CVLS fp8=fp1,fcr
BCT ctr=CL.6

6 load-updates (LFSU), 1 branch, 4 CVLS’s, 8 float ops (4 add, 2 mult, 2 fused ops)
Improving Complex Arithmetic

Why 8 floating-point ops?

\[ qr += er*sr - ei*si \]

Compiles to

\[ qr = qr + (er*sr - ei*si) \]

Rather than

\[ qr = (qr + er*sr) - ei*si \]

Does it matter?

It does to ANSI C

\[ \text{tiny} + (\text{LARGE} - \text{LARGE}) = \text{tiny} \]
\[ (\text{tiny} + \text{LARGE}) - \text{LARGE} = 0 \]

Do you care?

Either answer probably acceptable

3 fma’s vs 2

4 registers vs 3 (assuming \( qr \) stays in a register)

Easy to fix (if you know what’s happening)

Put in explicit parentheses

Or, the –03 compiler option may do this for you

Result: 7 cycles per iteration

One unaccounted for cycle per iteration remains
Keeping the CPU Busy

CPU’s have multiple functional units

Floating-point: add, multiply, divide
Integer, Load/Store
Branch

And multistage pipelines

<table>
<thead>
<tr>
<th>Processor chip</th>
<th>Ins/cycle</th>
<th>Pipe int</th>
<th>len. flt</th>
<th>Data cache</th>
<th>Assoc tvty</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alpha 21164</td>
<td>2</td>
<td>4</td>
<td>7</td>
<td>8K¹</td>
<td>1</td>
</tr>
<tr>
<td>MIPS R8000</td>
<td>4?</td>
<td>2</td>
<td>?</td>
<td>16K</td>
<td>1</td>
</tr>
<tr>
<td>PA-risc 7300LC</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>64K</td>
<td>2</td>
</tr>
<tr>
<td>Pentium</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>8K</td>
<td>2</td>
</tr>
<tr>
<td>PowerPC 604</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>16K</td>
<td>4</td>
</tr>
<tr>
<td>Power2</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>off-chip</td>
<td>4</td>
</tr>
</tbody>
</table>

From [http://infopad.eecs.berkeley.edu/CIC/summary/](http://infopad.eecs.berkeley.edu/CIC/summary/)

Inner loop must have independent operations

1. Alpha 21164 also has on-chip 96K L2 cache
Matrix-Vector Product (DGEMV)

\[ y = Ax \]

Each node is \( y(i) = y(i) + A(i,j)x(j) \)

Note: \( x \) is horizontal; \( x(j) \) used by nodes in \( j \)-th column
DGEMV programs

Execute column-wise (DAXPY approach)

\[
\begin{align*}
&\text{do 10 } j = 0, N-1 \\
&\quad \text{do 10 } i = 0, N-1 \\
&\quad 10 \quad y(i) = y(i) + A(i,j) x(j)
\end{align*}
\]

2 loads, 1 store per multiply-add  
No innerloop dependences - great for Cray T90

Execute row-wise (DDOT approach)

\[
\begin{align*}
&\text{do 10 } i = 0, N-1 \\
&\quad \text{do 10 } j = 0, N-1 \\
&\quad 10 \quad y(i) = y(i) + A(i,j) x(j)
\end{align*}
\]

2 loads, 0 stores per multy-add. \((y(i) \text{ in register.})\)  
Sequentially dependent innerloop  
At best, one mult-add per pipeline-length cycles.  
Good for nothing.
DGEMV programs

Execute 4-high swath (hybrid approach)

\[
\begin{align*}
do\ 10\ &i = 0, N-4, 4 \\
y0 &= y(i+0);\ y1 = y(i+1) \\
y2 &= y(i+2);\ y3 = y(i+3) \\
do\ 20\ &j = 0, N-1 \\
y0 &= y0 + A(i+0,j)*x(j) \\
y1 &= y1 + A(i+1,j)*x(j) \\
y2 &= y2 + A(i+2,j)*x(j) \\
20\ &y3 = y3 + A(i+3,j)*x(j) \\
y(i+0) &= y0;\ y(i+1) = y1 \\
10\ &y(i+2) = y2;\ y(i+3) = y3 \\
\end{align*}
\]

C Don’t forget the last rows

\[
\begin{align*}
do\ 30\ &i = N-3, N-1 \\
do\ 30\ &j = 0, N-1 \\
30\ &y(i) = y(i) + A(i,j)*x(j) \\
\end{align*}
\]

5 loads, 0 stores per 4 mult-add’s. (x(j) in register.)
Four multi-add’s are independent.
Good habit! Temporaries y0, y1,... prevent re-loads.
Great for microprocessors!
FFT Inner Loop

Natural code for computing complex butterfly:

\[
\begin{align*}
  c[i] &+ w*c[j] \\
  c[i] &- w*c[j]
\end{align*}
\]

and storing results in-place

(assume complex \(c[i]\) is \(d[2*i+0], d[2*i+1]\)):

\[
\begin{array}{l}
  \text{for}(i = m; i \leq n; i += \text{istep}) \{ \\
  j = i + \text{spread}; \\
  \text{tempr} = \text{wr} \times d[2*j+0] - \text{wi} \times d[2*j+1]; \\
  \text{tempi} = \text{wr} \times d[2*j+1] + \text{wi} \times d[2*j+0]; \\
  d[2*j+0] = d[2*i+0] - \text{tempr}; \\
  d[2*i+0] = d[2*i+0] + \text{tempr}; \\
  d[2*i+1] = d[2*i+0] + \text{tempi};
  \}
\end{array}
\]

Prediction: **4** loads per iteration

Observation: **6** loads per iteration

What happened?

Compiler doesn’t know \(i, i+1, j, \text{and } j+1\) are all different.

Result: extra loads **and** bad instruction scheduling.

Solution: Copy \(d\)'s in temps during butterfly.
Two-Level Memory Model

Infinite slow memory

Small fast memory
  Holds \( n \) “blocks”
  with \( s \) items per block

Data is transferred in fixed size blocks
  \( l \) cycles to move block

Cost model: \( l \times \# \) of transfers
  Often more accurate than flops or ops or ...

Examples:
  Disk – main memory (block = page, often 4096Bytes)
  Remote memory – Local memory (block = message)
  Memory – TLB (block = page)
  TLB – Level 2 cache (block = cacheline)
  L2 cache – L1 cache
  Cache – Registers (block = data item)
### SP2 and T3E Caches

<table>
<thead>
<tr>
<th>System</th>
<th>Block size, bytes</th>
<th>Number of blocks</th>
<th>Associativity</th>
<th>Miss penalty, cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP2 P2SCthin cache</td>
<td>256</td>
<td>512</td>
<td>4</td>
<td>$18,38^1$</td>
</tr>
<tr>
<td>SP2 TLB</td>
<td>4096</td>
<td>256</td>
<td>2</td>
<td>15(?)</td>
</tr>
<tr>
<td>T3E L1 cache</td>
<td>32</td>
<td>256</td>
<td>1</td>
<td>8-10</td>
</tr>
<tr>
<td>T3E L2 cache</td>
<td>64</td>
<td>1536</td>
<td>3</td>
<td>50$^2$</td>
</tr>
<tr>
<td>T3E TLB (dtb)</td>
<td>64</td>
<td>8196</td>
<td>full</td>
<td>75(?)</td>
</tr>
</tbody>
</table>

1. Cycles for requested value, cycles for whole line
2. Approximate. Depends on page, bank conflicts, etc.

### Special T3E features:

Sequential access helped by 6 stream buffers
   Default is disabled at SDSC and by HPF

Random access helped by `!DIR$ CACHE_BYPASS` directive
   ```
   !DIR$ CACHE_BYPASS B
   DO 10 I = 1, N
   10 ... B(A(I))
   ```
Locality

**Great** – use data in block many times
“Temporal locality” (reusing data) necessary but insufficient

**Good** – use entire block once
“Spatial locality”
Example: sequential access
Perhaps factor of 2 slowdown for L2 cache, fine for TLB.

**Bad** – need new block for each reference
  - **thrashing** - very slow
Example: random or strided access to large array
Matrix Multiplication

Naive program (you should use vendor-tuned DGEMM):

\[
\begin{align*}
&\text{do 10 } i = 0, N-1 \\
&\quad \text{do 10 } j = 0, N-1 \\
&\quad \quad \text{do 10 } k = 0, N-1 \\
&10 \quad C(i,j) = C(i,j) + A(i,k) \times B(k,j)
\end{align*}
\]

Operation count: \( N^3 \) fma’s

Matrix multiplication iteration space graph
Is matrix multiplication $O(N^5)$?

log vs log graph (line is $10^{-4}N^5$)

actual and projected data points

$N = 2000$ took five days
$N = 12000$ would take 1095 years (and 3.5 GB)

Remember: $O(N^3)$ is attainable
Two Level Analysis

C matrix – Great

B matrix
    Great if entire matrix fits in small memory
    Good otherwise

A matrix
    Great if A and B fit in small memory
    Good if one row of A fits
        Beware of associativity problems!
    Bad otherwise
Mat Mult Performance Explained

Line 1: load on $A$ and $B$ per fma
Line 2: cache miss on $B$ every 16 fma’s
Line 3: TLB miss on $A$ every fma
Line 4: page miss on $B$ every 512 fma’s
also page miss on $C$ every $N$ fma’s
Line 5: page miss on $A$ every fma
Locality in Matrix Transpose

Blocking produces semilocal accesses on both matrices

inner loop

memory access pattern

= unused data moved into cache during non-local access

2 innermost loops
Tiling Matrix Transpose

Untitled code
   do 10 i = 0, n-1
       do 10 j = 0, n-1
       10       B(i,j) = A(j,i)

Stripmining
   do 10 ii = 0, n-1, stride
       ilimit = min(n,ii+stride)
       do 10 i = ii, ilimit-1
           do 10 jj = 0, n-1, stride
               jlimit = min(n,jj+stride)
               do 10 j = jj, jlimit-1
               10               B(i,j) = A(j,i)

Loop interchange
   do 10 ii = 0, n-1, stride
       ilimit = min(n,ii+stride)
       do 10 jj = 0, n-1, stride
           jlimit = min(n,jj+stride)
           do 10 i = ii, ilimit-1
               do 10 j = jj, jlimit-1
               10               B(i,j) = A(j,i)