SGI® Altix™ Single CPU Optimizations

Reiner Vogelsang
SGI GmbH
reiner@sgi.com

January 19, 2005

Tuning Guide Outline

SGI® Altix™ 3000 architecture
Single-Processor Tuning:

Step 1. Port Code and Get the Right Answers

Step 2: Use Existing Tuned Code

Step 3: Let the Compiler Do the Work

Step 4: Find Out Where to Tune

Step 5: Tune Cache Performance

Libraries

- Using existing tuned code means exploit highly optimized library code as well

Scientific and Math Libraries

SCSL

- SGI® Scientific Computing Software Library

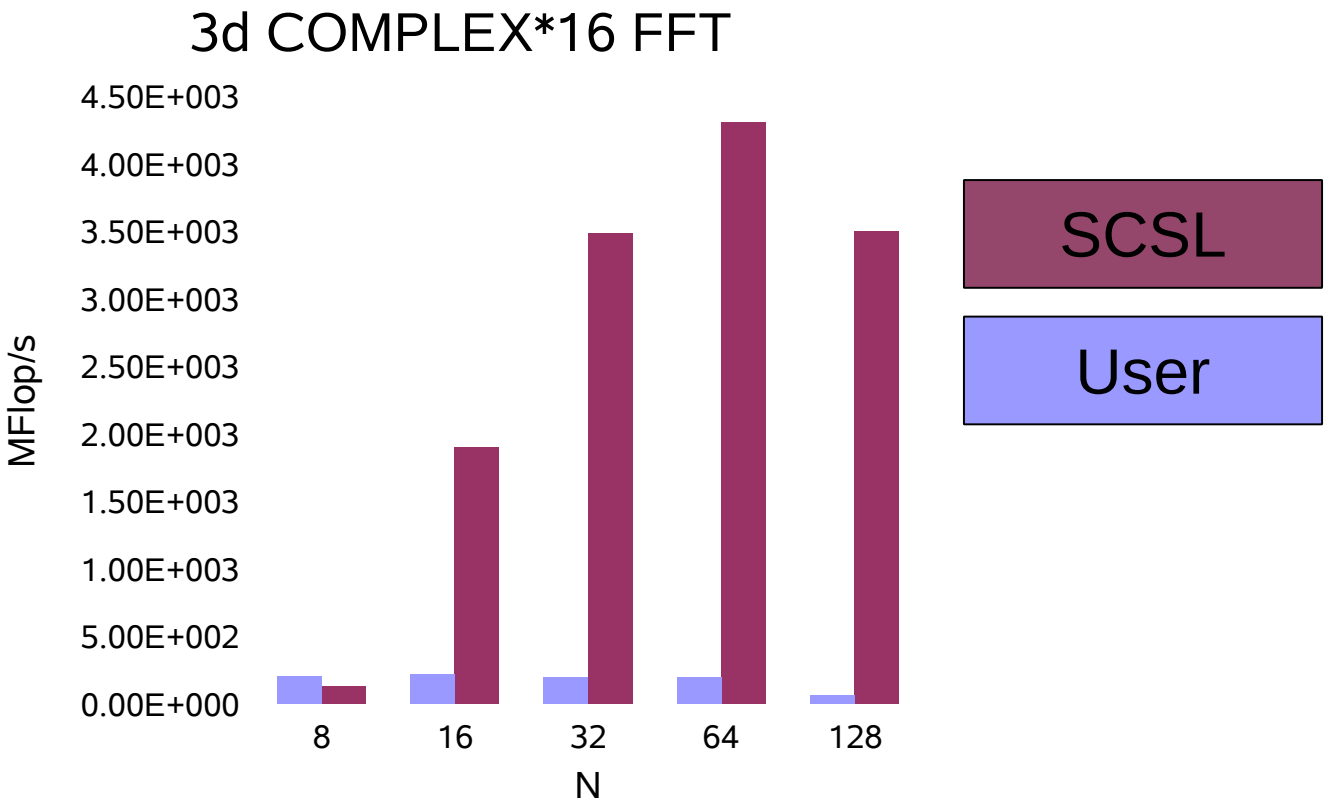
MKL

- Intel® Math Kernel Library

Standard math libraries

- `libimf` from Intel compiler suite instead of `libm` of glibc
  - Remove an explicit call of -lm from you link step.

SCSL – A Motivating Example



SCSL Contents

BLAS (Basic Linear Algebra Subprograms)

- Fundamental building blocks
  - Level 1: vector-vector operations
  - Level 2: matrix-vector operations
  - Level 3: matrix-matrix operations
- Most level-2 and level-3 BLAS optimized and parallelized
- Level-3 routines include “3M” complex matrix multiplication kernels
- Fortran90 interface to BLAS

SCSL Contents (continued)

- LAPACK (Linear Algebra Package)
  - Version 3.0
  - Linear systems of equations
  - Eigenvector/value calculations
  - Linear least squares
- Mostly optimized via BLAS
- LU, Cholesky, and QR factorizations
- Rewritten in house

SCSL Contents (continued)

- Signal processing library
  - Fast Fourier transforms (FFTs)
    - One-, two-, and three-dimensional mixed radix
    - Multiple one-dimensional mixed radix
    - Out-of-cache sizes run on multiple processors
  - Linear filtering operations
    - 1D and 2D convolution and correlation
    - Supports Finite Impulse Response (FIR) and Infinite Impulse Response (IIR) filters
    - 2D routines run on multiple processors

SCSL Contents (continued)

- Sparse solver library
  - Direct solver for sparse linear systems
    - Symmetric systems and non-symmetric systems with symmetric structure
    - Multiple ordering methods, including feedback
    - Supports both in-core and out-of-core
    - All parallelized
    - No pivoting

SCSL Contents (continued)

- Sparse solver library
  - Iterative solver for sparse linear systems
    - Methods:
      - Conjugate gradient
      - Conjugate residual
      - Conjugate gradient squared
      - BiCGSTAB
    - Preconditioners:
      - Jacobi
      - Symmetric successive over-relaxation
      - Incomplete LDLT by pattern
      - Incomplete LDLT by value

SCSL Contents (continued)

- Thread-safe, parallel random number generator
  - Based on linear congruent sequence
    $Y(n+1) = (a X(n) + c) \bmod 2^{64}$
  - a and c are chosen from Knuth, *The Art of Computer Programming*, Vol. 2, 3rd edition, pp. 106–108
  - Each thread generates an independent random sequence starting from predefined values
  - Single/double real precisions supported
    `man rand64` for details

SCSL Contents (continued)

- General features
  - Most routines come in multiple flavors, accepting single/double precision, real/complex data types

    Exceptions:
    - Direct sparse solver library supports only double-precision real and double-precision complex
    - Iterative sparse solver and DGEMMS (Strassen) routines support only double-precision real input matrices
    - Random number generator routines support only real single- and double-precision
  - A version of SCSL supporting 8-byte integer arguments is available

SCSL Contents (continued)

- Documentation: man pages
  - intro_scsl
  - intro_blas1, intro_blas2, intro_blas3 intro_cblas
  - intro_fft
  - intro_lapack
  - intro_solvers

  (These point to detailed man pages for individual functions and subroutines.)
- SCSL User Guide in the works

How to Use SCSL

- **Compiling and linking**
  - **Use appropriate header files—e.g.,**

```
#include <scsl_fft.h>
#include <scsl_cblas.h>
```

  - **Use the `-mp` compiler flag if using multiprocessing directives**
  - **Choose library at link time**
    - **Serial: `-lscs` or `-lscs_i8`**
    - **Parallel: `-lscs_mp` or `-lscs_i8_mp`**
    - **`-lthreads` is automatically linked in to support OpenMP**

SCSL Performance Tips

General parallel performance

- Make sure problem size is large enough to offset multiprocessing overhead
- Initialize data in parallel region
- Use round-robin placement (not available on Altix™ yet)
- Use dplace with OpenMP™ code (use –x[2|6] flag on Altix)
- If you are running on a quiescent system consider disabling dynamic threads
  - `setenv OMP_DYNAMIC FALSE`

SCSL Performance Tips (continued)

General parallel performance

- `KMP_STACKSIZE`, which is the private stack size for threads (default 4MB on Itanium® 2), set in init section for Altix™
- `OMP_NUMTHREADS` on Altix is set to 1 by default in init section unless set by user

–Performance-enhanced glibc memory management enabled in init section for Altix

- Turn off mmap
- Disable trimming

SCSL - Future Plans

- Performance enhancements, emphasis on sparse solvers and FFTs
- New LAPACK and ScaLAPACK
- Support for unique features of SGI architecture (UV vectors, FPGA?)
- Block tridiagonal solver
- Other features requested by you!

MKL 7.0

Intel Math Kernel Library Contents:

- BLAS (`-lmkl_itp`) – includes sparse BLAS
- LAPACK (`-lmkl_lapack32`, `-lmkl_lapack64`)
- FFT (`-lmkl_itp`)
- Vector math library (`-lmkl_vml_itp`)

MKL - List of Vector-Math Functions

Trigonometric
Sin
Cos
SinCos
Tan
Asin
Acos
Atan
Atan2

Error
Erf
Erfc

Hyperbolic
Sinh
Cosh
Tanh
Asinh
Acosh
Atanh

Exponential & Logarithmic
Exp
Ln
Log10

Power
Pow
Powx
Sqrt
Cbrt
InvSqrt
InvCbrt

Others
Inv
Div

MKL High Performance Functions

	Intel® Pentium® III processor				Intel® Pentium® 4 processor				Intel® Itanium® 2 processor			
	HA	LA	HA	LA	HA	LA	HA	LA	HA	LA	HA	LA
Inv	10.12	10.12	11.31	11.23	9.85	3.47	11.75	10.17	3.11	3.11	4.11	4.11
Div	11.07	10.96	16.43	16.33	9.83	3.74	11.68	11.68	4.11	4.11	5.18	5.18
Sqrt	15.73	15.73	33.14	33.05	9.41	4.94	27.08	25.71	5.12	5.12	7.15	7.15
InvSqrt	8.16	8.13	31.60	31.50	6.20	3.75	19.61	15.80	5.12	5.12	6.13	6.13
Cbrt	30.05	30.05	45.34	45.34	22.42	17.29	40.67	32.42	7.16	7.16	10.20	10.20
InvCbrt	31.30	31.25	42.31	42.03	23.43	15.83	40.72	26.87	7.17	7.17	9.18	9.18
Pow	106.83	66.51	145.14	95.92	38.82	38.82	105.93	79.24	11.26	11.26	21.36	21.36
Powx	105.98	105.98	143.88	143.88	39.22	39.22	81.36	80.48	9.44	9.44	21.44	21.43
Exp	19.93	19.93	37.10	37.07	10.76	9.22	26.14	16.74	4.16	4.15	6.17	6.17
Ln	20.83	20.83	37.72	37.60	15.78	12.58	25.18	23.91	7.17	7.17	11.19	11.19
Log10	20.97	20.86	37.65	37.58	19.07	12.82	25.68	24.57	7.17	7.17	11.20	11.20
Cos	44.33	30.48	52.28	52.28	22.36	12.07	35.89	35.87	7.17	7.17	9.19	9.19
Sin	42.12	26.70	48.90	48.36	19.51	11.14	35.94	35.82	6.15	6.15	8.17	8.17
SinCos	65.07	39.80	77.05	77.05	29.39	19.26	64.10	48.12	8.18	8.18	11.20	11.20
Tan	59.66	38.97	75.60	68.18	40.35	18.73	66.48	47.92	9.21	9.21	11.24	11.24
Acos	43.16	36.17	98.28	97.36	25.06	16.17	69.17	51.66	10.22	10.22	15.30	15.30
Asin	39.71	28.30	94.11	94.11	23.47	15.92	65.42	65.42	10.21	10.21	15.29	15.29
Atan	37.72	19.74	87.34	87.16	24.48	14.43	62.51	50.50	11.39	11.39	13.31	13.31
Atan2	78.98	32.94	139.96	139.96	56.10	28.60	113.27	67.22	12.32	12.32	19.45	19.45
Cosh	37.97	37.97	72.36	72.36	20.48	12.98	37.34	27.39	7.17	7.17	10.20	10.20

Math Libraries

- Standard math functions: `sin`, `exp`, etc.
  - Intel® compiler math library `libimf.a`
  - Automatically linked in by `efc` and `ecc`
  - Specifying `-lm` on link line results in symbol resolution from glibc's `libm`
- Whetstone benchmark, `libimf` vs. `libm` (higher is better):
  - Single precision: `814000` vs. `816500`
  - Double precision: `753000` vs. `744000`
- Consider using VML for operations involving large vectors

- **Tuning for cache and memory performance**

Cache terminology

- Cache line: minimum unit of transfer from next-higher cache into this one
  [64 bytes for L1, 128 bytes for L2 and L3]
- Cache hit: reference to a cache line which is present in the cache
- Cache miss: reference to a cache line which is not present in this cache level and must be retrieved from a higher cache (or memory or swap space)

Cache terminology

- Write-through cache: cache change immediately updates the next level in the hierarchy
- Write-back cache: cache change does not update next level in hierarchy until required
  - "clean" cache line is not modified
  - "dirty" line is modified and not yet written to next level of hierarchy

Cache terminology

- Prefetch: a load operation from memory to cache initiated prior to the time the CPU needs the result
- Set associativity: describes how memory is mapped into the cache
  - Direct-mapped: a cache line from memory has only one possible location in cache
  - N-way associative: a cache line from memory has N possible locations in cache

Memory hierarchy latencies

- Memory latency differs within the hierarchy: performance is affected by where the data resides
  - Registers: 0 cycles latency (cycle = 1/freq)
  - L1 cache: 1 cycle
  - L2 cache: 5-6 cycles
  - L3 cache: 12-17 cycles
  - Main memory: 130-1000+ cycles

Memory hierarchy latencies

- Virtual-to-physical memory map (cached in TLB) can also affect performance
  - DTLB (data) miss in first level: 4 cycle penalty
  - DTLB miss in second level: goes to Hardware Page Walker (HPW); 25 or 31 cycles latency if found in L2 or L3 cache
  - DTLB miss in second level, uncached:
    OS takes “minor” page fault: many cycles
  - Worse still if page has been swapped out

Waiting for memory...

- CPUs which are waiting for memory are not doing useful work
- Software should be “hierarchy-aware” to achieve best performance:
  - Perform as many operations as possible on data in registers
  - Perform as many operations as possible on data in the cache(s)
  - Keep data uses *spatially* and *temporally* local

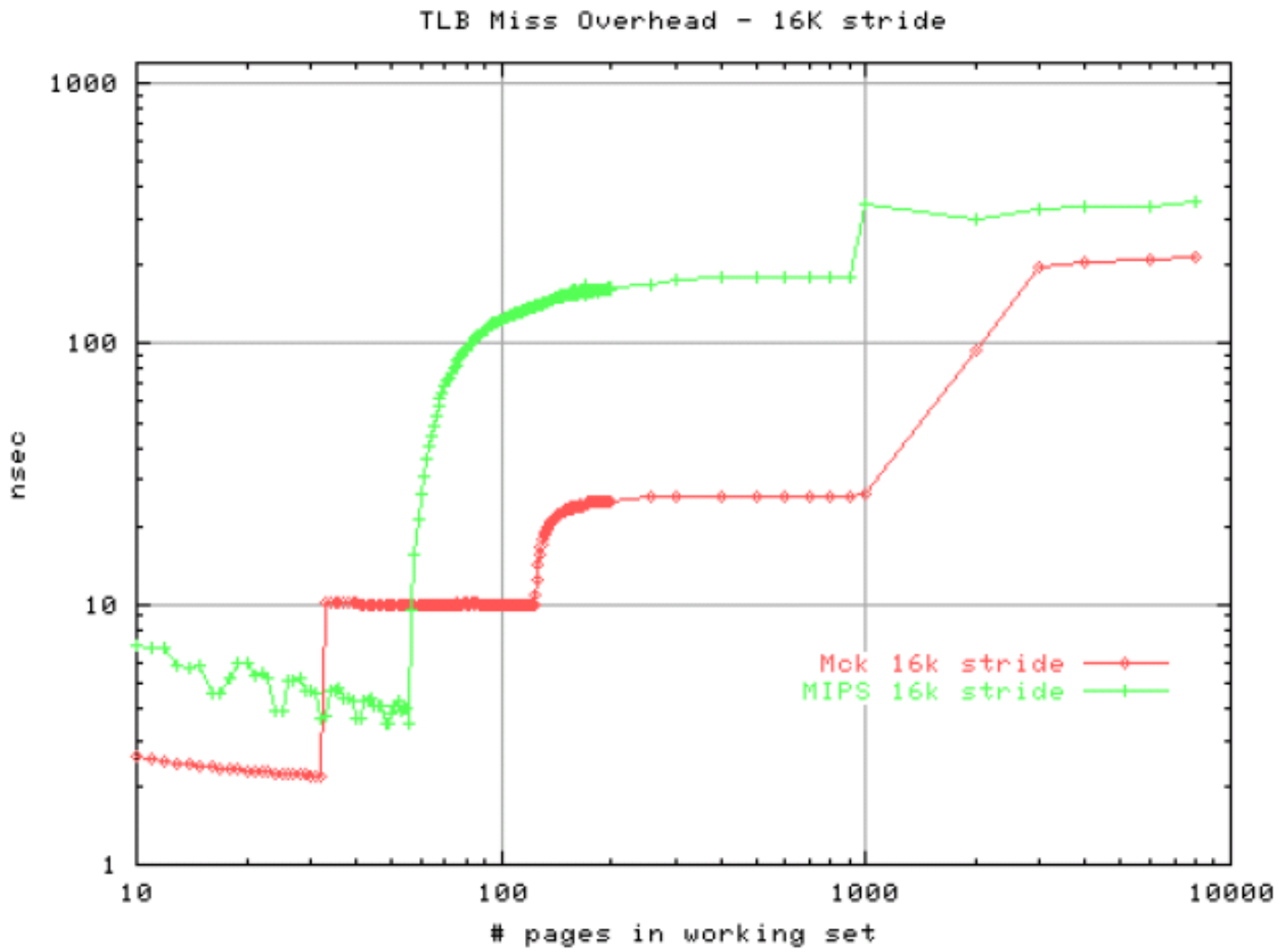
Cache basics: access patterns

- **Accesses to adjacent memory locations use multiple values from a single cache line: less time waiting for memory**

```
do i = 1, n                          do j = 1, m
   do j = 1, m                         do i = 1, n
      a(i,j) = b(i,j)        ⟶             a(i,j) = b(i,j)
   enddo                                 enddo
enddo                                enddo
```

```
for (j=0; j<m; j++)                  for (i=0; i<n; i++)
   for (i=0; i<n; i++)        ⟶          for (j=0; j<m; j++)
      a[i][j] = b[i][j];                   a[i][j] = b[i][j];
```

Impact of TLB Misses



Cache basics: access patterns

- **Grouping together data used at the same time allows fewer cache misses**

```
 d = 0.0
 do i = 1, n
   j=ind(i)
   d=d+sqrt(x(j)*x(j)+
&           y(j)*y(j)+
&           z(j)*z(j))
 enddo
```

→

```
 d = 0.0
 do i = 1, n
   j=ind(i)
   d=d+sqrt(r(1,j)*r(1,j)+
&           r(2,j)*r(2,j)+
&           r(3,j)*r(3,j))
 enddo
```

```
d=0.0
for (i=0; i<n; i++) {
   j=ind[i];
   d+= sqrt(x[j]*x[j]+
            y[j]*y[j]+
            z[j]*z[j]);
}
```

→

```
d=0.0
for (i=0; i<n; i++) {
   j=ind[i];
   d+= sqrt(r[j][1]*r[j][1]+
            r[j][2]*r[j][2]+
            r[j][3]*r[j][3]);
}
```

Cache basics: size/alignment

- Avoid array sizes that are multiples of the cache size
  - Use of such arrays tends to involve memory which maps to the same set of cache lines
  - Array sizes can be modified, or “padding” can be inserted into arrays
  - In C/C++, it is often better to allocate one large array and then use pointers to access sections of it, while avoiding cache line overwrite

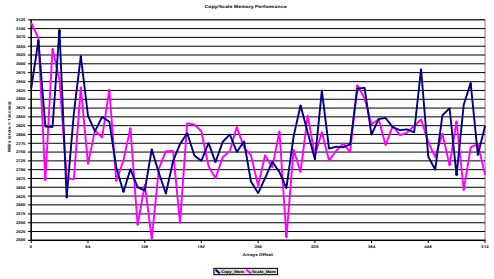
Cache basics: size/alignment

```
parameter (max = 12*1024*1024)
dimension a(max), b(max), c(max)
```

Better if becomes

```
Parameter (max = 12*1024*1024, pad=128)
common /arrays/ a(max), aa(pad), b(max),&
                bb(pad), c(max)
```

Alignment: Copy/Scale



Observation in local memory performance: Vector-Triad

```
REAL (SIZE): A,B,C,D
DO ITER=1,NITER
  DO i=1,N
     A(i) = B(i) + C(i) * D(i)
  ENDDO
   <OBSCURE>
ENDDO
```

Max. Performance - L2 cache: Half of Peak Performance

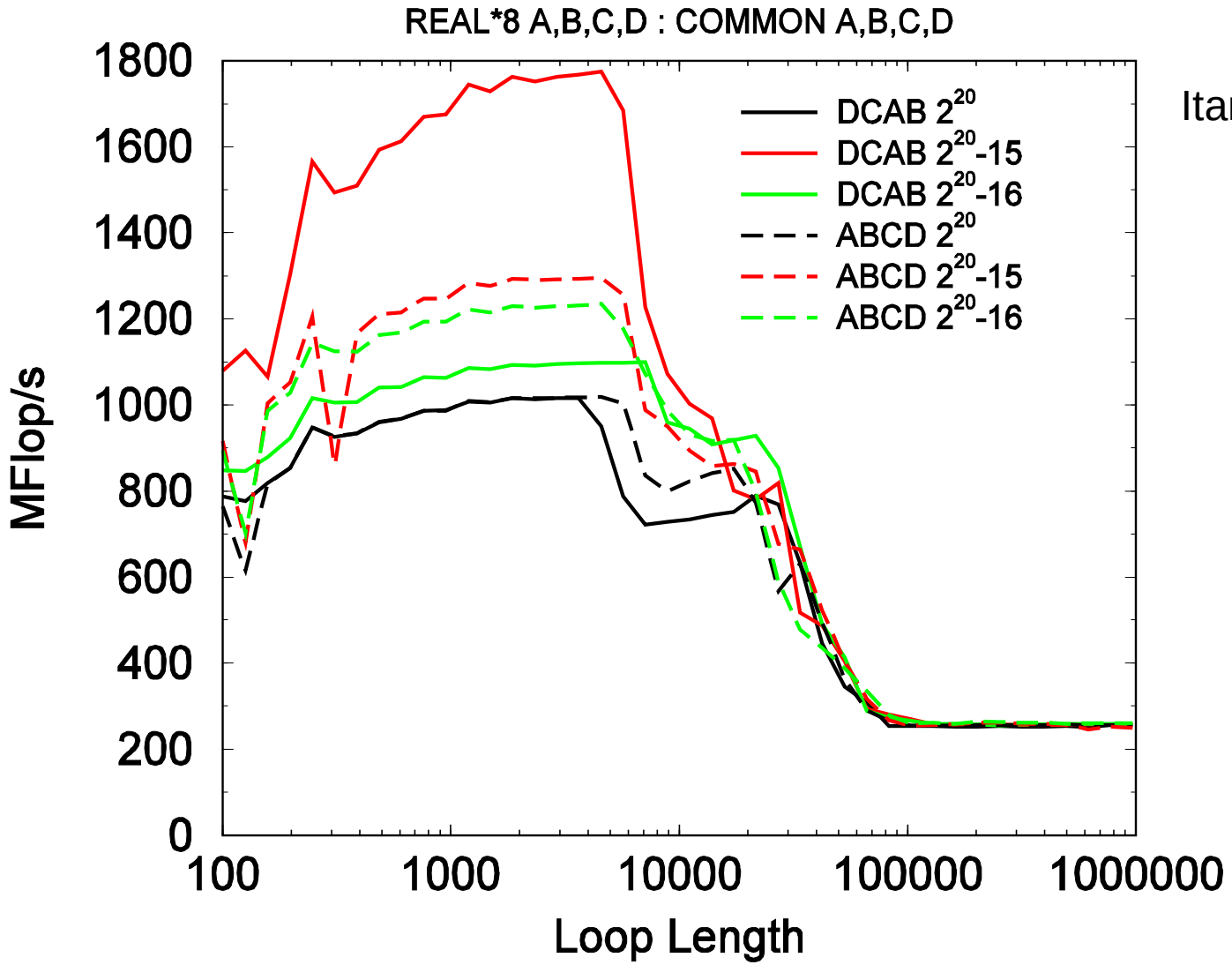
Available: 2 LOAD and (2 LOAD or 2 STORE) per cycle

2 Triads-> 4 Flop

2 Triads-> 6 LD& 2 ST ->1 cycle (2ST+2LD)+1 cycle (4 LD)-> 2 cycles

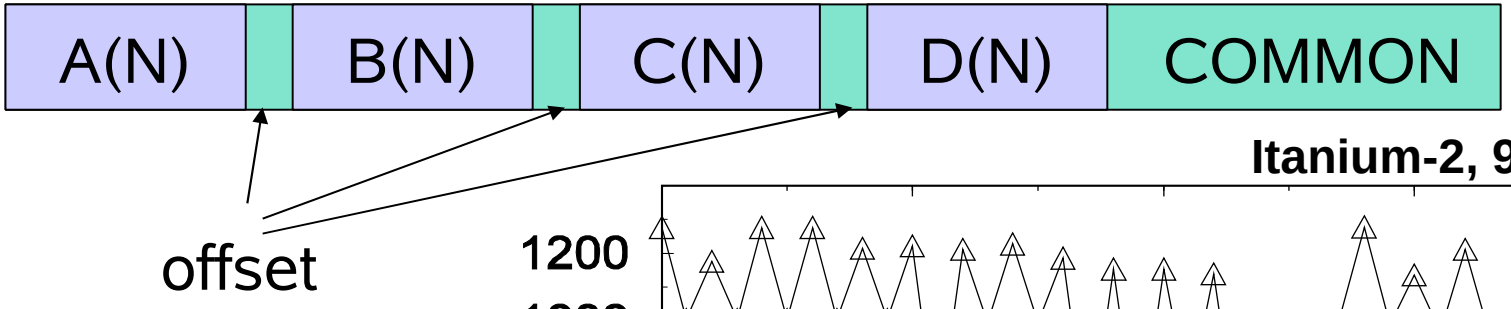
-> 4 Flop / 2 cycle = Half of Peak

Itanium2 – Impact of Memory Layout

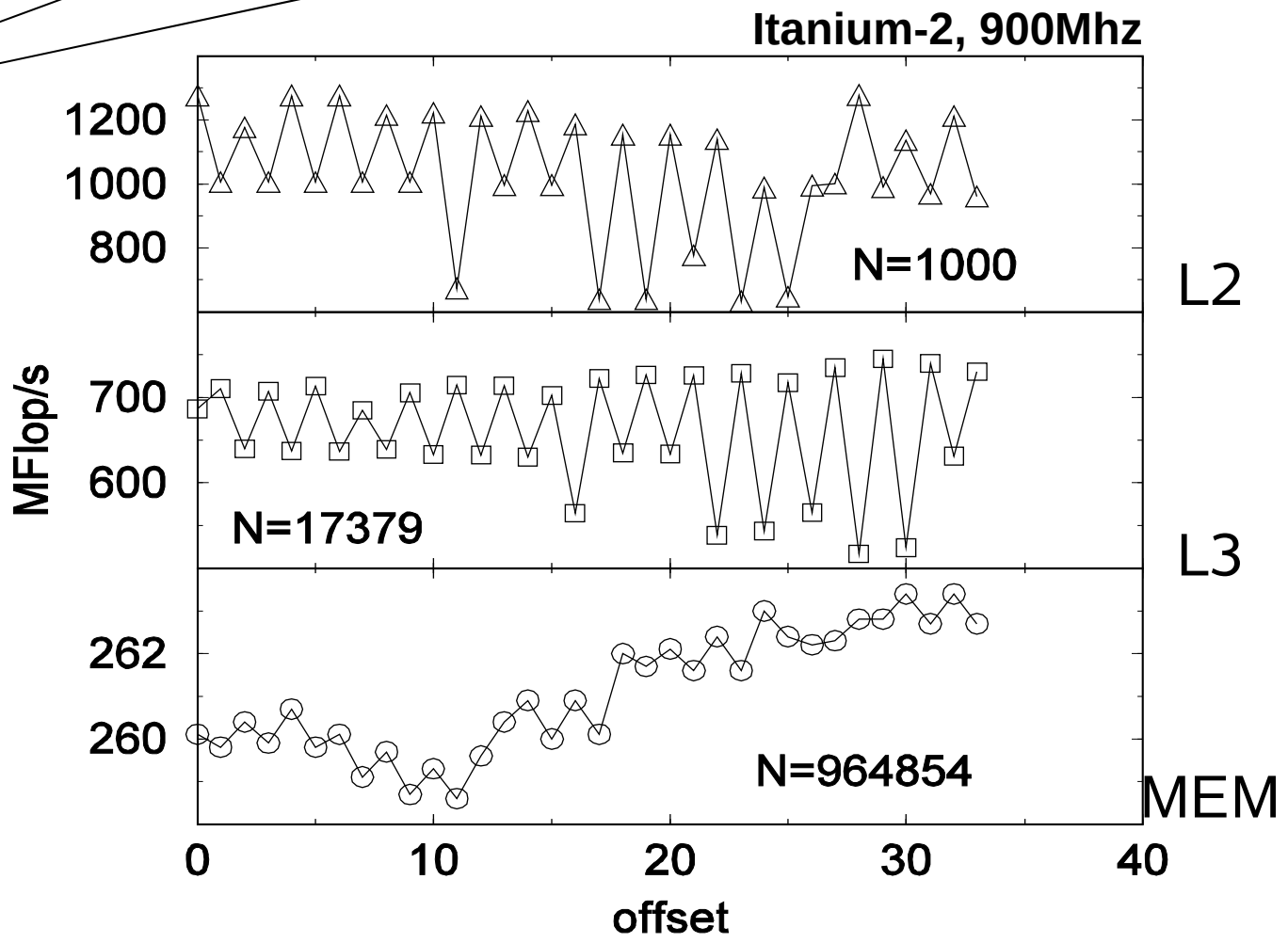


Itanium2-900MHz

Itanium2 – Impact of Memory Layout



offset



Caches are extremely sensitive to memory layout

81 % of peak mem-bw

Cache problem diagnosis

- Use performance monitoring tools to see where unexpected time is spent
- Example: 3-D ADI solver

```
double precision a(256,256,256)
do k = 1, nz
  do j = 1, ny
    call xsweep(a(1,j,k),1,nx)
  enddo
enddo
do k = 1, nz
  do i = 1, nx
    call ysweep(a(i,1,k),ldx,ny)
  enddo
enddo
do j = 1, ny
  do i = 1, nx
    call zsweep(a(i,j,1),ldx*ldy,nz)
  enddo
enddo
```

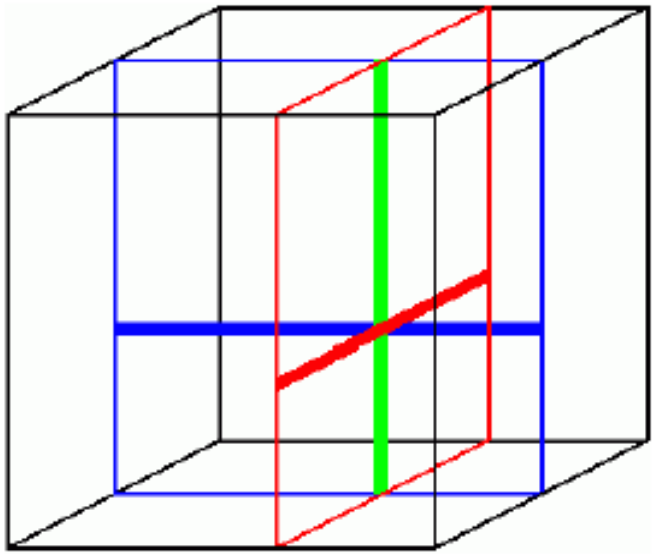
zsweep

```
subroutine zsweep(v,is,n)
      implicit none
      real*8 v(1+is*(n-1))
      integer is, n
      integer i
      real*8 half
      parameter (half = 0.5d0)
     do i = 2, n
         v(1+is*(i-1)) = v(1+is*(i-1)) + half*v(1+is*(i-2))
      enddo
      do i = n-1, 1, -1
         v(1+is*(i-1)) = v(1+is*(i-1)) - half*v(1+is*i)
      enddo
      return
      end
```

Poor cache reuse in zsweep

- **Same amount of work done in each sweep, but times are quite different:**

```
13998: *Total for thread 27974*
13292: a.out:*Total*
 9773: a.out:zsweep_
 1709: a.out:ysweep_
 1594: a.out:xsweep_
  706: libscs.so:*Total*
  706: libscs.so:drand64_
  164: a.out:main
   52: a.out:_init
```



Array padding required

- Array was dimensioned (256,256,256); this is bad for cache reuse
- Re-dimension as (257,257,256): performance better than doubles!

```
6506: *Total for thread 28114*
5816: a.out:*Total*
2388: a.out:zsweep_
1613: a.out:ysweep_
1592: a.out:xsweep_
 689: libscs.so:*Total*
 689: libscs.so:drand64_
 147: a.out:main
  76: a.out:_init
```

Avoid TLB Misses

```
parameter (nx=256, ny=256, nz=256)
parameter (ldx=257, ldy=257, ldz=256)
real*8 a(ldx,ldy,ldz), temp(ldx,ldz)
. . .
do j = 1, ny
  call copy(a(1,j,1),ldx*ldy,temp,ldx,nx,nz)
  do i = 1, nx
    call zsweep(temp(i,1),ldx,nz)
  enddo
  call copy(temp,ldx,a(1,j,1),ldx*ldy,nx,nz)
enddo
...
```

Avoid TLB Misses (continued)

```
subroutine copy(from,lf,to,lt,nr,nc
 )
real*8 from(lf,nc), to(lt,nc)
do j = 1, nc
  do i = 1, nr
    to(i,j) = from(i,j)
  enddo
enddo
end
```

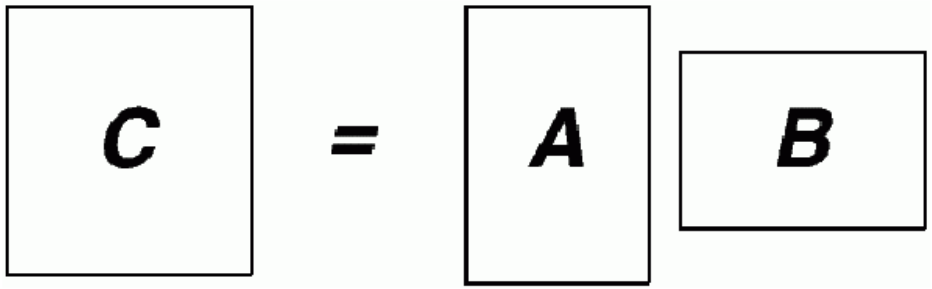
```
6235: *Total for thread 28
 372*
5540: a.out:*Total*
1613: a.out:ysweep_
1593: a.out:xsweep_
1387: a.out:zsweep_
 726: a.out:copy_
 695: libscs.so:*Total
 *
 694: libscs.so:drand6
 4_
 166: a.out:main
  55: a.out: init
```

Cache blocking

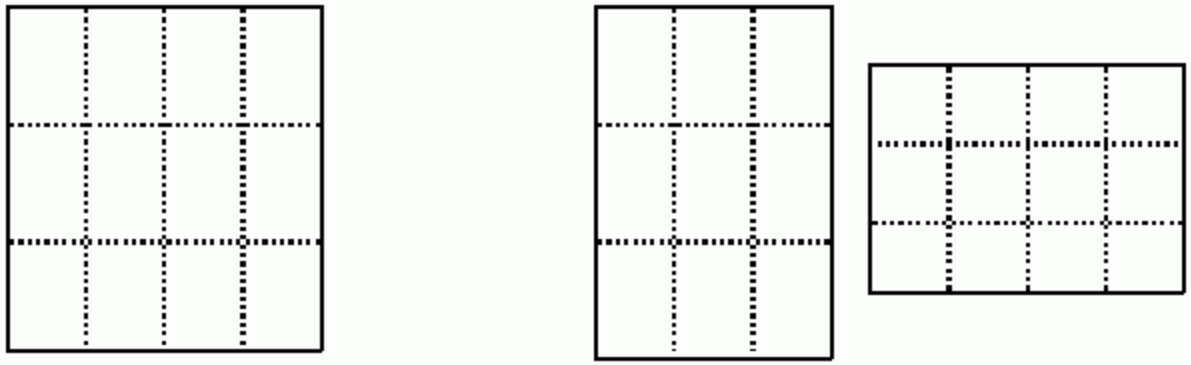
- Working with very large data sets can overwhelm the cache with streaming data
- To achieve the best performance, the data should be broken into “cache-sized” chunks
- This procedure is known as *cache blocking*

Cache blocking

- A very large matrix multiplication

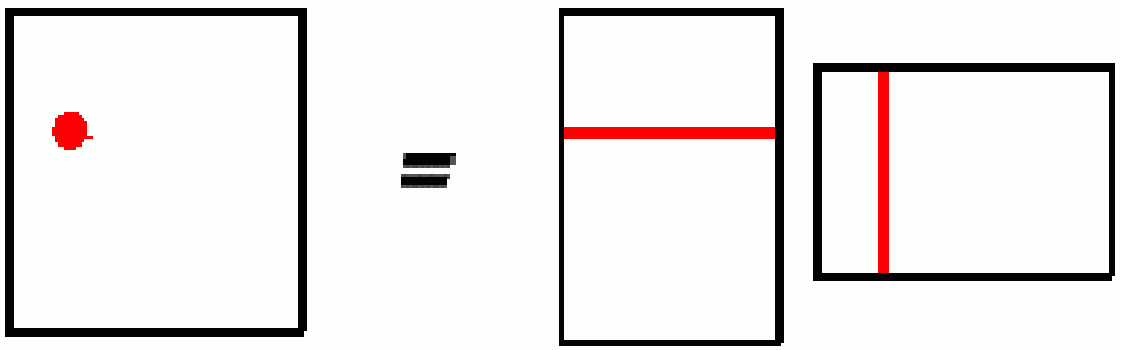


- Blocked into smaller sections



Why cache blocking works

- Computing an element of C requires an entire row of A and entire column of B
- Large A and B cause the beginnings of the row of A and column of B to be flushed from cache
- Blocks which fit in cache will not be flushed until their use is completed



Coding practice for performance

- Avoid aliasing of variables
  - Avoid EQUIVALENCE statements
  - Use `-fno-alias` flag where appropriate
- Write structured code
  - Loops are easy for the compiler to digest
  - Avoid GOTO
- Write standard code
  - Code which violates language standards is never a good idea

Other techniques: Compiler

- Intel's compilers provide the following optimizations at –O3:
  - Loop interchange
  - Loop fusion
  - Loop splitting or fission
  - Loop unrolling and jaming
  - Padding
  - Some cache blocking
  - Prefetching
  - Pipelining

Compiler Directives

Fortran:

`cdec$ ivdep`	no aliasing
`cdec$ swp`	try to software-pipeline
`cdec$ noswp`	disable software-pipelining
`cdec$ loop count (NN)`	hint for SWP
`cdec$ distribute point`	split this large loop
`cdec$ unroll (n)`	unroll *n* times
`cdec$ nounroll`	do not unroll
`cdec$ prefetch a`	prefetch array “a”
`cdec$ noprefetch c`	do not prefetch array “c”

Compiler Directives

C/C++:

`#pragma ivdep`	no aliasing
`#pragma swp`	try to software-pipeline
`#pragma noswp`	disable software-pipelining
`#pragma loop count (NN)`	hint for SWP
`#pragma distribute point`	split this large loop
`#pragma unroll (n)`	unroll *n* times
`#pragma nounroll`	do not unroll
`#pragma prefetch a`	prefetch array “a”
`#pragma noprefetch c`	do not prefetch array “c”

Indirect Addressing

`ivdep` directive is recognized by both `ecc` and `efc`

Particularly important for loops with indirect addressing, e.g.,

```
#pragma ivdep
for (i = 0; i < n; i++)
  a[b[i]] += c[i] * d[i];
```

- Without `ivdep`, loop will have poor SWP schedule or no SWP at all
- May also need to specify `-ivdep_parallel` to indicate no loop-carried dependencies

Loop Nest Optimization

Matrix multiplication with wrong index order:

```
line 3:  do j = 1, n
line 4:   do k = 1, n
line 5:    do i = 1, n
line 6:     c(i,j)=c(i,j)+a(k,i)*b(k,j)

% ifort -O3 -opt_report mtm.f

LOOP INTERCHANGE in mm_ at line 4
LOOP INTERCHANGE in mm_ at line 5

Block, Unroll, Jam Report:
Loop at line 3 unrolled and jammed by 4
Loop at line 4 unrolled and jammed by 4
```

Example: Matrix Multiplication

```
for (i = 0; i < M; i++)
 for (j = 0; j < N; j++)
  for (k = 0; k < K; k++)
   C[i*ldc+j] -= A[i*K+k] * B[j*ldb+k];
```

Both `A` and `B` arrays accessed with unit stride in innermost loop, should get good performance

Compile `-O3 -fno-alias`

Time for `M = 10000, N = 200, K = 200` case: 19.1 sec, 402 MFLOPS

Matrix Multiplication: Hand-Unrolled Loops

```
for (i = 0; i < M-3; i+=4) {
for (j = 0; j < N-3; j+=4) {
   t00 = 0; t10 = 0; t20 = 0; t30 = 0;
   t01 = 0; t11 = 0; t21 = 0; t31 = 0;
   t02 = 0; t12 = 0; t22 = 0; t32 = 0;
   t03 = 0; t13 = 0; t23 = 0; t33 = 0;
   for (k = 0; k < K; k++) {
      t00 += A[(i+0)*K+k] * B[(j+0)*ldb+k];
      t10 += A[(i+1)*K+k] * B[(j+0)*ldb+k];
      t20 += A[(i+2)*K+k] * B[(j+0)*ldb+k];
      t30 += A[(i+3)*K+k] * B[(j+0)*ldb+k];
      . . .
   }
   C[(i+0)*ldc+j+0] -= t00; C[(i+1)*ldc+j+0] -= t10;
   C[(i+2)*ldc+j+0] -= t20; C[(i+3)*ldc+j+0] -= t30;
   . . .
 }

}
```

Compile `-O3 -fno-alias`

Time = 4.4 sec, 1820 MFLOPS

Matrix Multiplication: Fortran Version

```
do j = 1, n-3, 4
do i = 1, m-3, 4
   t00 = 0.0
   t10 = 0.0
   t20 = 0.0
   t30 = 0.0

      . . .
   do k = 1, p
      t00 = t00 + A(k,j) * B(k,i)
      t10 = t10 + A(k,j+1) * B(k,i)
      t20 = t20 + A(k,j+2) * B(k,i)
      t30 = t30 + A(k,j+3) * B(k,i)
      . . .
   end do
   c(i,j) = c(i,j) - t00
   c(i,j+1) = c(i,j+1) - t10
   c(i,j+2) = c(i,j+2) - t20
   c(i,j+3) = c(i,j+3) - t30
end do

end do
```

Compile -o3

Time = 4.0 sec, 2000 MFLOPS

Stretching the Pipeline

Compiler assumes loads will be satisfied from lowest level cache with minimal latency (e.g., 6 cycles for fp)

- Achievable with good prefetching
- Good prefetching not always possible

Can use pfmon to examine stall and latency events, e.g., `BE_EXE_BUBBLE_FRALL`, `DATA_EAR_CACHE_LAT8`

Can try "stretching" the pipeline using undocumented, unsupported flag `-mP3OPT_ecg_mm_fp_ld_latency=##`

- Time for previous example with 20-cycle latency is 2.56 sec, 3123 MFLOPS

Or use previously tuned code (a general rule of thumb)

- Using SCSL DGEMM, example time is 2.2 sec, 3670 MFLOPS

Prefetching

- Loads which miss in L3 cache stall the Itanium2 until main memory responds with the data (hundreds of CPU cycles)
- Prefetches are “non-stalling” loads, in the sense that they request data from memory but do not require resolution before the processor can begin work on following instructions

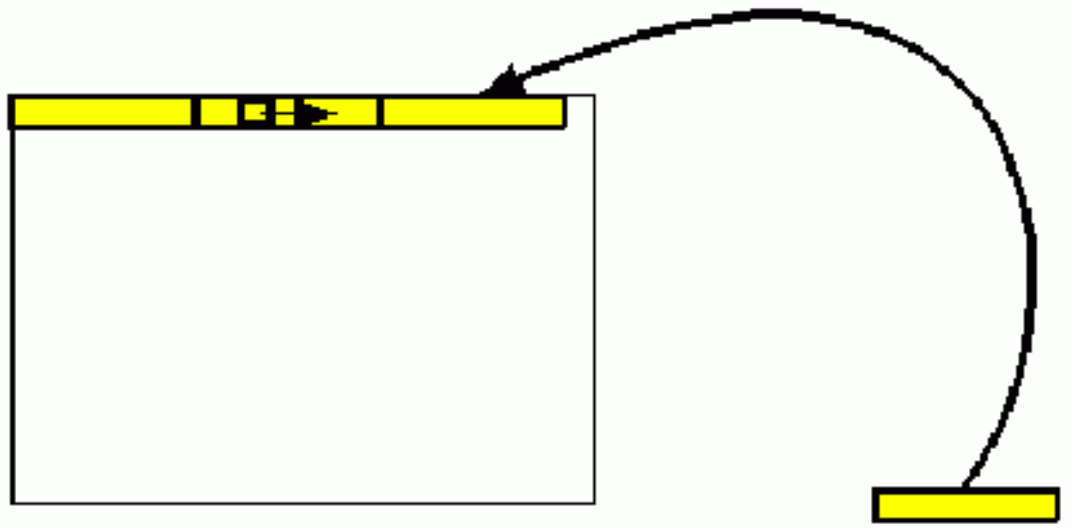
Prefetching

- Automatically enabled in code compiled –O3

```
double *b;
for (i=0;i<n;++i)
  a += *(b+i);
```

becomes

```
double *b;
for (i=0;i<n;++i)
  <prefetch *(b+i+
  a += *(b+i)
```



Manual prefetching

- Automatic prefetching is limited by the compiler's "understanding" of the code
- Since prefetches are a memory instruction, their presence can inhibit performance in tight loops
- Assistance can be provided to the compiler via directives

```
CDEC$ prefetch B
CDEC$ noprefetch A
```

Example: Radix-3 Fast Fourier Transform Kernel

```
      subroutine passf3 (ido,l1,cc,ch,wa1,wa2)

      if (ido .ne. 2) go to 102
      do 101 k=1,l1
      ...
101   continue
      return
102   do 104 k=1,l1
        do 103 i=2,ido,2
        ...
103     continue
104   continue
      return
      end
```

Radix-3 FFT Kernel

```
do 101 k = 1, l1
  tr2        = cc(1,2,k) + cc(1,3,k)
  cr2        = cc(1,1,k) + taur * tr2
  ch(1,k,1)  = cc(1,1,k) + tr2
  ti2        = cc(2,2,k) + cc(2,3,k)
  ci2        = cc(2,1,k) + taur * ti2
  ch(2,k,1)  = cc(2,1,k) + ti2
  cr3        = taui * (cc(1,2,k) - cc(1,3,k))
  ci3        = taui * (cc(2,2,k) - cc(2,3,k))
  ch(1,k,2)  = cr2 - ci3
  ch(1,k,3)  = cr2 + ci3
  ch(2,k,2)  = ci2 + cr3
  ch(2,k,3)  = ci2 - cr3
101 continue
```

6 loads
6 stores
12 fma

compile -O3 13 cycles/iteration

Radix-3 FFT Kernel

```
cdec$ noprefetch
do 101 k = 1, l1
      tr2 = cc(1,2,k) + cc(1,3,k)
      cr2 = cc(1,1,k) + taur * tr2
      ...
      ch(1,k,2) = cr2 - ci3
      ch(1,k,3) = cr2 + ci3
      ch(2,k,2) = ci2 + cr3
      ch(2,k,3) = ci2 - cr3
101 continue
```

compile `-O3 -mP3OPT_ecg_mm_fp_ld_latency=13`

8 cycles/iteration

Radix-3 FFT Kernel Summary

	-03	-O3 -mP3OPT_ecg_mm_fp_ld_latency=13
Original source	13 cycles	10 cycles
Original NO prefetch	10 cycles	8 cycles
Loop 101 only	13 cycles	10 cycles
Loop 101 NO Prefetch	10 cycles	7 cycles
ideal	6 cycles	

Example: CFD Solver

```
DO 33 IP=1,NCELL
  IE=LQ(6,IP); IW=LQ(5,IP); IN=LQ(4,IP)
  IS=LQ(3,IP); IT=LQ(2,IP); IB=LQ(1,IP)
  WSP=WS(IP)
  IF(IB.GT.0.AND.IB.LT.IP) WSP=WSP+DBLE(AC(1,IP))*WS(IB)
  IF(IT.GT.0.AND.IT.LT.IP) WSP=WSP+DBLE(AC(2,IP))*WS(IT)
  IF(IS.GT.0.AND.IS.LT.IP) WSP=WSP+DBLE(AC(3,IP))*WS(IS)
  IF(IN.GT.0.AND.IN.LT.IP) WSP=WSP+DBLE(AC(4,IP))*WS(IN)
  IF(IW.GT.0.AND.IW.LT.IP) WSP=WSP+DBLE(AC(5,IP))*WS(IW)
  IF(IE.GT.0.AND.IE.LT.IP) WSP=WSP+DBLE(AC(6,IP))*WS(IE)
  IF(LARBE) THEN                    ! LARBE = .FALSE.
        . . . more if-then for WSP computation
  ENDIF
  WS2(IP)=WSP*DBLE(D(IP))
33 CONTINUE
```

Compile -O3 **Time is 9.25 seconds**

CFD Solver (2 Loops)

```
IF(LARBE) THEN
 DO 33 IP=1,NCELL
  IE=LQ(6,IP); IW=LQ(5,IP); IN=LQ(4,IP)
  . . .
  IF(IE.GT.0.AND.IE.LT.IP) WSP=WSP+DBLE(AC(6,IP))*WS(IE)
  . . . more if-then for WSP computation
  WS2(IP)=WSP*DBLE(D(IP))
 33 CONTINUE
ELSE                              ! LARBE = .FALSE.
 DO 34 IP=1,NCELL
  IE=LQ(6,IP); IW=LQ(5,IP); IN=LQ(4,IP)
  . . .
  IF(IE.GT.0.AND.IE.LT.IP) WSP=WSP+DBLE(AC(6,IP))*WS(IE)
  WS2(IP)=WSP*DBLE(D(IP))
 34 CONTINUE
ENDIF
```

Compile -O3 **Time is 7.85 seconds**

CFD Solver (Directives)

```
DO 33 IP=1,NCELL                ! NCELL = 156000 -> 10.5 MB
cdec$ prefetch LQ, AC
  IE=LQ(6,IP); IW=LQ(5,IP); IN=LQ(4,IP)
  IS=LQ(3,IP); IT=LQ(2,IP); IB=LQ(1,IP)
  WSP=WS(IP)
  IF(IB.GT.0.AND.IB.LT.IP) WSP=WSP+DBLE(AC(1,IP))*WS(IB)
  IF(IT.GT.0.AND.IT.LT.IP) WSP=WSP+DBLE(AC(2,IP))*WS(IT)
  IF(IS.GT.0.AND.IS.LT.IP) WSP=WSP+DBLE(AC(3,IP))*WS(IS)
  IF(IN.GT.0.AND.IN.LT.IP) WSP=WSP+DBLE(AC(4,IP))*WS(IN)
  IF(IW.GT.0.AND.IW.LT.IP) WSP=WSP+DBLE(AC(5,IP))*WS(IW)
  IF(IE.GT.0.AND.IE.LT.IP) WSP=WSP+DBLE(AC(6,IP))*WS(IE)
  WS2(IP)=WSP*DBLE(D(IP))
33 CONTINUE
```

Compile -O3 **Time is still 7.85 seconds**

CFD Solver (Explicit Prefetch)

```
DO 33 IP=1,NCELL
  call lfetch_nta(LQ(1,IP+14))
  call lfetch_nta(AC(1,IP+14))
  IE=LQ(6,IP); IW=LQ(5,IP); IN=LQ(4,IP)
  IS=LQ(3,IP); IT=LQ(2,IP); IB=LQ(1,IP)
  wsp=ws(ip)
  IF(IB.GT.0.AND.IB.LT.IP) WSP=WSP+DBLE(AC(1,IP))*WS(IB)
  IF(IT.GT.0.AND.IT.LT.IP) WSP=WSP+DBLE(AC(2,IP))*WS(IT)
  IF(IS.GT.0.AND.IS.LT.IP) WSP=WSP+DBLE(AC(3,IP))*WS(IS)
  IF(IN.GT.0.AND.IN.LT.IP) WSP=WSP+DBLE(AC(4,IP))*WS(IN)
  IF(IW.GT.0.AND.IW.LT.IP) WSP=WSP+DBLE(AC(5,IP))*WS(IW)
  IF(IE.GT.0.AND.IE.LT.IP) WSP=WSP+DBLE(AC(6,IP))*WS(IE)
  WS2(IP)=WSP*DBLE(D(IP))
33 CONTINUE
```

Compile -O3 **Time is 4.55 seconds (9.25, 7.85)**

Math Libraries

Standard math functions: `sin`, `exp`, etc.

- Intel® compiler math library `libimf.a`
- Automatically linked in by `efc` and `ecc`
- Specifying `-lm` on link line results in symbol resolution from glibc's `libm`

Whetstone benchmark, `libimf` vs. `libm` (higher is better):

- Single precision: `814000` vs. `816500`
- Double precision: `753000` vs. `744000`

Consider using VML for operations involving large vectors

Glibc Issues

Glibc memory management routines emphasize system-wide efficiency over performance

- See *www.linuxshowcase.org/ezolt.html* for discussion

For code that uses glibc's `malloc()` / `free()`:

- `setenv MALLOC_TRIM_THRESHOLD_ -1`
- `setenv MALLOC_MMAP_MAX_ 0`
- Prevents glibc from using `mmap` (slow) and from returning memory to the system (which requires global TLB flush)
- Works with C/C++ codes and Intel® Fortran Cray™ pointers; does not work with Fortran `allocatable` arrays
- Automatically set when using MPT or SCSL

Fortran Memory Management

Intel® Fortran runtime libraries have their own memory management routines for handling automatic arrays, allocatable arrays, etc.

- `f90_allocate[1-16], f90_deallocate[1-13]`
  - **At least some of these rely on `mmap()`/`munmap()`.**
    - **This true for efc 7.1. Use -stack_temps**
  - **No knobs/environment variables to tweak**
  - **Users have observed poor scaling for stack-allocatable arrays**

Example Workaround for Stack-Allocatable Arrays

```
       SUBROUTINE fast_alloc (n, info)
      INTEGER :: n, info

      INTEGER  :: xxx(2*n)
#if defined (FAST_ALLOC)
      POINTER (xxx_p, xxx)
      xxx_p = malloc(8*n)
#endif

      xxx(1:2*n:1000) = 0
      info = xxx(2*n)

#if defined (FAST_ALLOC)
      call free (xxx_p)
#endif

      RETURN
      END
```

Compile with
`-safe_cray_ptr`

7.1 compiler supports Cray™ pointers in modu

Still need to set malloc environment variables

Beware Fortran 90 Array Sections

```
subroutine foo(n)

real, allocatable ::
a(:,:)

allocate(a(n,n))


call foobar(a)

call bar(a(:,:))


return

end
```

```
{ .mib
   mov      r41=r32
   nop.i    0
   br.call.sptk.many
b0=f90_allocate3# ;; }
{ .mib
   ld8      r40=[r36]
   mov      gp=r35
   br.call.sptk.many        b0=foobar_#
;; }
.....
{ .mib
   nop.m    0
   nop.i    0
   br.call.sptk.many
b0=f901_argcpy4# ;; }
{ .mib
   mov      gp=r35
   mov      r40=r8
   br.call.sptk.many        b0=bar_# ;; }
{ .mib
   ld8      r40=[r39]
   mov      gp=r35
   br.call.sptk.many
b0=f90_argclean# ;; }
```

Beware Fortran 90 Array Sections

- Latest Intel 8.1 compiler provide the option

  `-check arg_temp_created`

- With -traceback the application terminates at the offending subroutine call

Other Performance Caveats

Carefully examine loops that:

- **Are heavy in complex arithmetic; explicit manipulation of equivalenced real arrays might be faster**
- **Have complicated constant expressions; the compiler does not always pull the constants out of the loop**
- **Operate on F90 pointer arrays; use of Cray™ pointers may improve performance**

Note: All performance issues discussed here have been passed on to Intel® for resolution in future compiler updates.

Bcopy/Memcpy on Altix™

Standard glibc `bcopy`/`memcpy` routines are not highly optimized

MPT has optimized `bcopy` routine, `fastbcopy`

- Same calling sequence as `bcopy`
- `50%` faster than `bcopy` on large transfers
- For best performance, source and destination addresses should be word-aligned and transfer length should be a multiple of 8 bytes
- NOTE: `-lmpi` requires that executable be launched with `mpirun`

Lab

- Go to Altix/Single_CPU_lab/single-cpu.
- Study the lab sheet 'Single CPU Optimization Techniques'
- Proceed in accordance with the lab sheet and apply optimization techniques presented.

Alternative

- Go to Altix/Performance_analysis/labs/csrc
- Enhance adi2!
- Repeat measurement of performance metric.


sgi®