Single-node Optimization Techniques

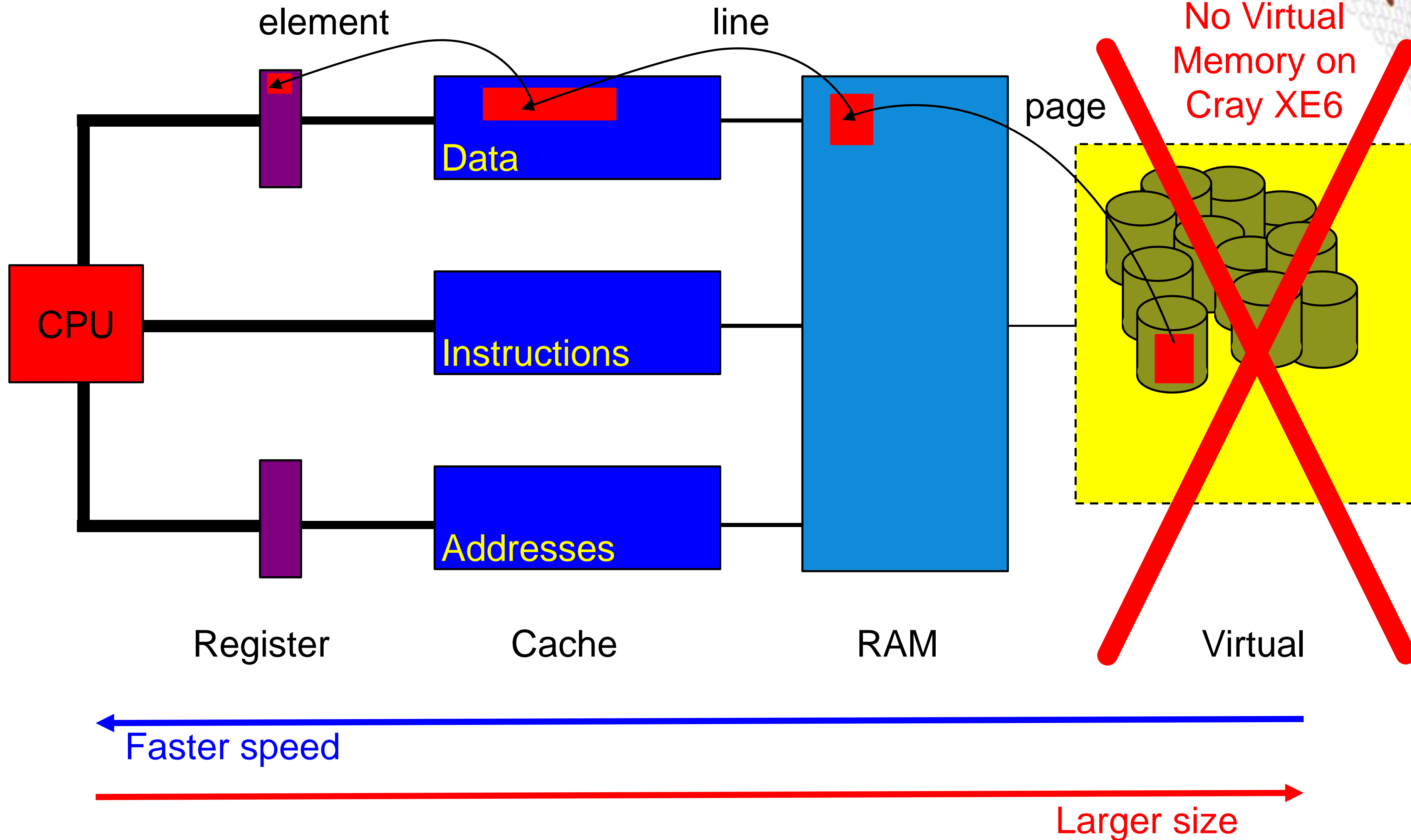
© Cray Inc 2013

Doesn't the compiler do all this for me?

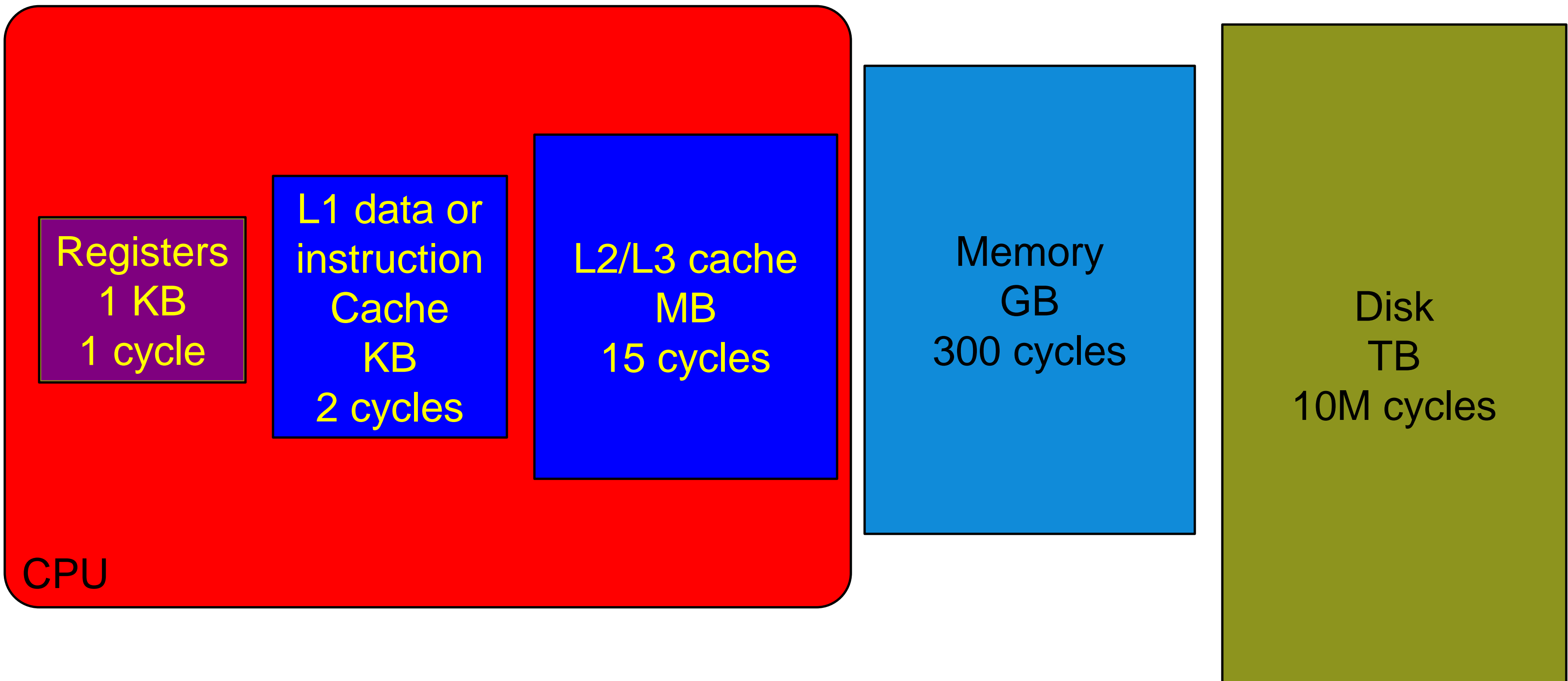
CRAY

- **Not yet...**
  - (standard answer, unchanged for last 50 or so years)
- **You can make a big difference to code performance**
  - Helping the compiler spot optimisation opportunities
  - Using specialist knowledge of your application
  - Removing obscure (and antique) "optimisations" in older code
    - simple code is always best, until proved otherwise
- **What we cover in this talk:**
  - Quick review of memory hierarchy
  - Cache blocking
  - TLB use optimisation
  - loop unrolling
  - vectorisation
- **No fixed rules: optimise on case-by-case basis**
  - But first, check what the compiler is already doing

Memory Hierarchy



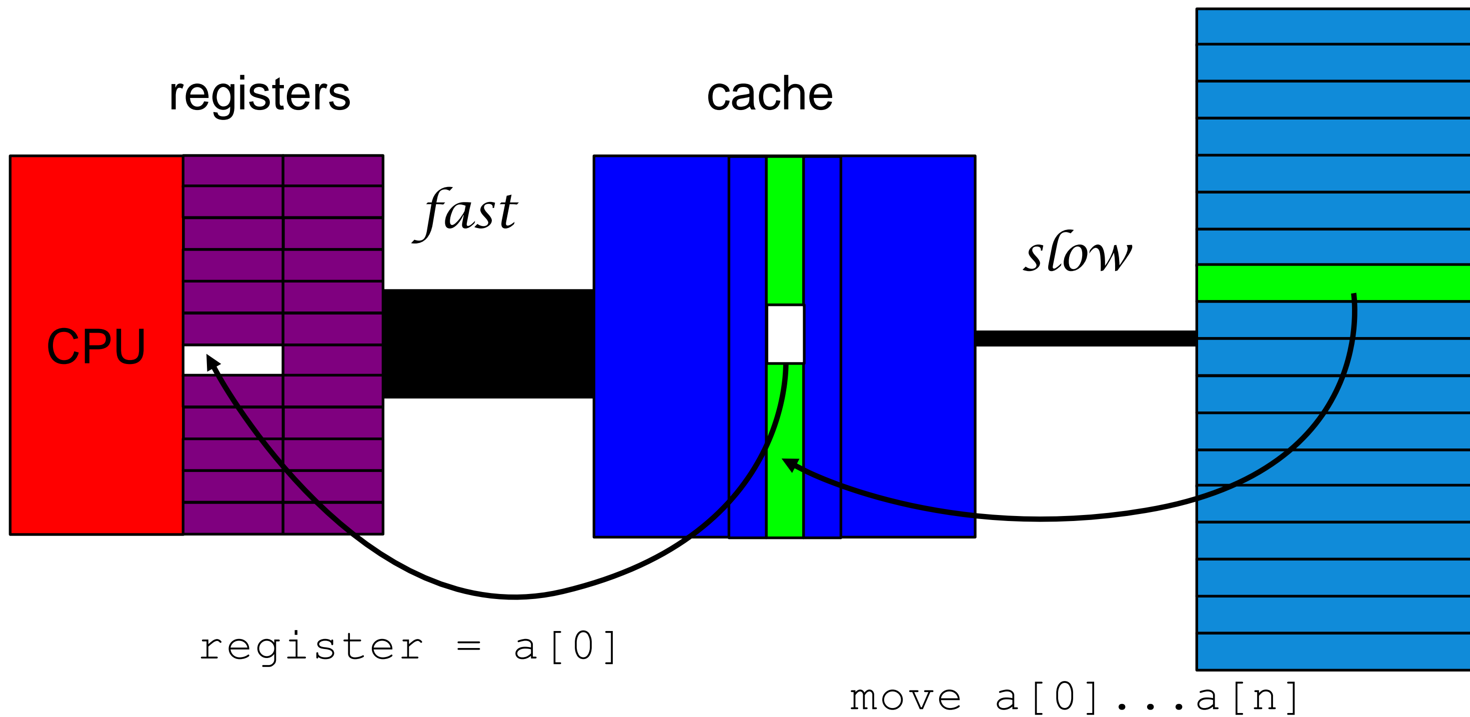
Memory Hierarchy



Cache Lines

Typically more than one element at once is transferred

`x = a[0]`



Bad Cache Alignment

CrayPAT profiling with `export PAT_RT_HWPC=2` (L1 and L2 metrics)

```
Time%                                                0.2%
Time                                             0.000003
Calls                                                   1
PAPI_L1_DCA                 455.433M/sec         1367 ops
DC_L2_REFILL_MOESI           49.641M/sec          149 ops
DC_SYS_REFILL_MOESI           0.666M/sec            2 ops
BU_L2_REQ_DC                 74.628M/sec          224 req
User time                     0.000 secs         7804 cycles
Utilization rate                                   97.9%
L1 Data cache misses         50.308M/sec          151 misses
LD & ST per D1 miss                                  9.05 ops/miss
D1 cache hit ratio                                 89.0%
LD & ST per D2 miss                                683.50 ops/miss
D2 cache hit ratio                                 99.1%
L2 cache hit ratio                                 98.7%
Memory to D1 refill           0.666M/sec            2 lines
Memory to D1 bandwidth       40.669MB/sec          128 bytes
L2 to Dcache bandwidth     3029.859MB/sec         9536 bytes
```

cf: 8

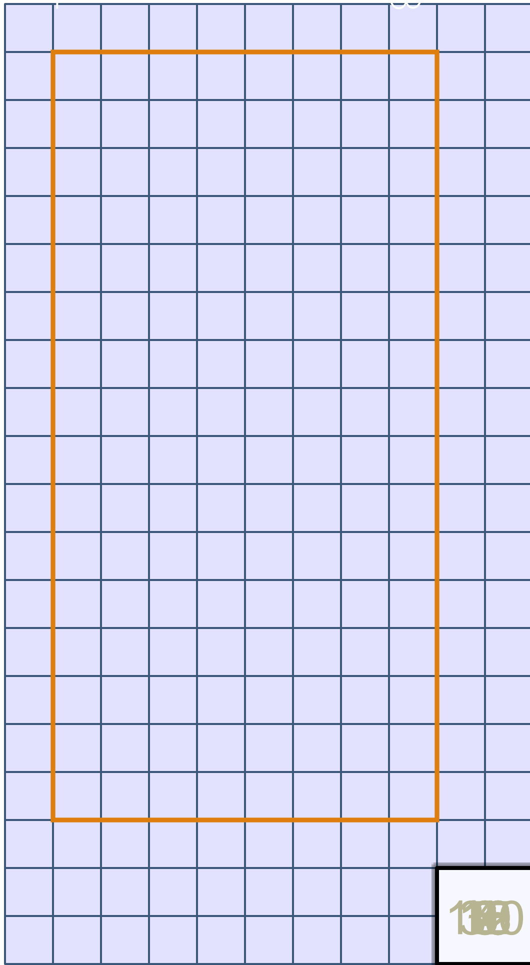
Good Cache Alignment

```
Time%                                              0.1%
Time                                           0.000002
Calls                                                 1
PAPI_L1_DCA              689.986M/sec              1333 ops
DC_L2_REFILL_MOESI        33.645M/sec                65 ops
DC_SYS_REFILL_MOESI                                   0 ops
BU_L2_REQ_DC              34.163M/sec                66 req
User time                  0.000 secs              5023 cycles
Utilization rate                                  95.1%
L1 Data cache misses      33.645M/sec                65 misses
LD & ST per D1 miss                               20.51 ops/miss
D1 cache hit ratio                                95.1%
LD & ST per D2 miss                             1333.00 ops/miss
D2 cache hit ratio                               100.0%
L2 cache hit ratio                               100.0%
Memory to D1 refill                                   0 lines
Memory to D1 bandwidth                                0 bytes
L2 to Dcache bandwidth  2053.542MB/sec             4160 bytes
```

Cache blocking

- **A combination of:**
  - strip mining (also called loop blocking, loop tiling...)
  - loop interchange
- **Designed to increase data reuse:**
  - temporal reuse: reuse array elements already referenced
  - spatial reuse: good use of cache lines
- **Many ways to block any given loop nest**
  - Which loops should be blocked?
  - What block size(s) will work best?
- **Analysis can reveal which ways are beneficial**
  - How big is your cache?
    - L1 is 16kB on Interlagos.
  - How many cache lines can it hold?
    - each line typically 64B, so
  - How many cache lines are needed per loop iteration?
  - ...
- **But trial-and-error is probably faster**
  - or autotuning of the code

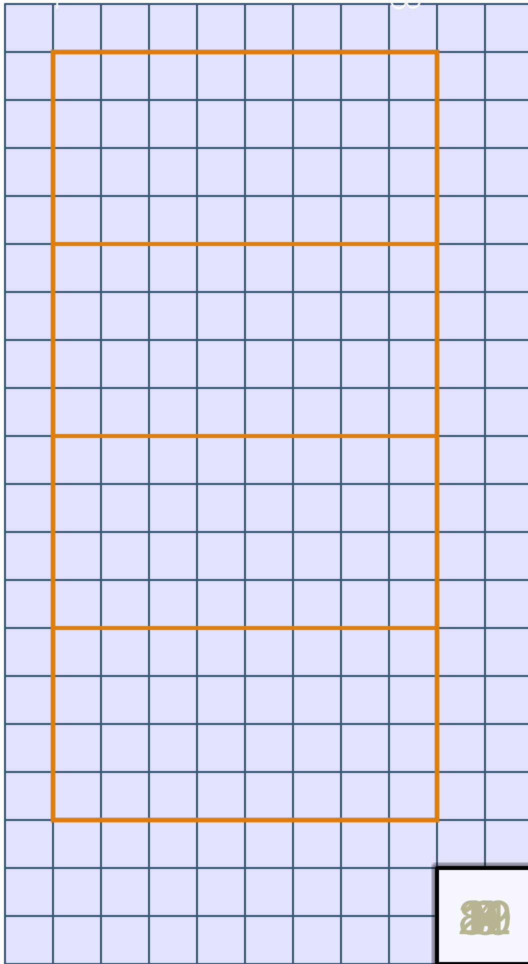
Cache Use in Stencil Computations



```
DO j = 1, 8
   DO i = 1, 16
      a = u(i-1,j) + u(i+1,j) &
        + u(i,j-1) + u(i,j+1) &
        - 4*u(i,j)
   ENDDO
ENDDO
```

- **Imagine a CPU architecture where:**
  - each cache line holds 4 array elements
  - cache can hold 12 lines of data
  - Each execution of i-loop needs:
    - 3*CEILING[(16+2)/4]=15 cache lines
- **No cache reuse b/w j-loop iterations**
  - Because 15 is greater than 12
    - iteration j loaded `u(i:i+3,j+1)` (4 elements)
    - iteration j+1 could reuse this (for central term)
    - but it's already been evicted from the cache
- **Cache misses per loopnest iteration**
  - 8*15 = 120

Blocking to Increase Reuse

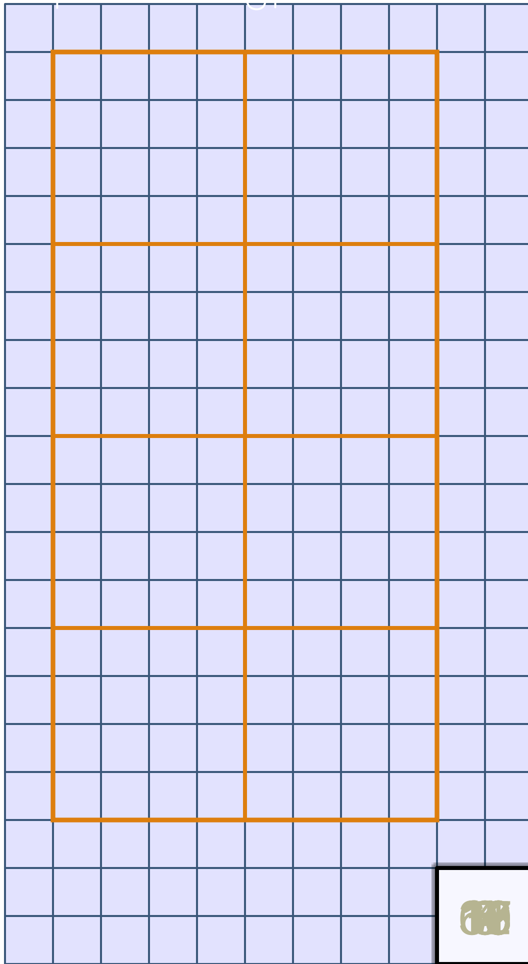


- **Block the inner loop**

```
DO ib = 1, 16, 4
   DO j = 1, 8
      DO i = ib, ib + 4-1
         a = u(i-1,j) + u(i+1,j) &
           + u(i,j-1) + u(i,j+1) &
           - 4*u(i,j)
      ENDDO
   ENDDO
ENDDO
```

- **Cache lines per j-loop iteration:**
  - 3*CEILING[(4+2)/4]=6 cache lines
  - So can hold data for 4 j-values in cache
    - because (4+2)*CEILING[(4+2)/4]=12
- **Cache liner per ib-loop iteration**
  - (8+2)*CEILING[(4+2)/4]=20
- **Cache misses per loopnest iteration**
  - (16/4)*20=80
  - reduced from 120
- **Better temporal locality**

Blocking to Increase Reuse



- Iterate over 4×4 blocks (or "tiles")

```
DO jb = 1, 8, 4
   DO ib = 1, 16, 4
      DO j = jb, jb + 4-1
         DO i = ib, ib + 4-1
            a = u(i-1,j) + u(i+1,j) &
              + u(i,j-1) + u(i,j+1) &
              - 4*u(i,j)
         ENDDO
      ENDDO
   ENDDO
ENDDO
```

- Cache lines per tile:
  - (4+2)*CEILING[(4+2)/4]=12
  - Can reuse for (some of) next tile
- Cache lines for each jb-iteration
  - (4+2)*CEILING[(16+2)/4]=30
- Cache misses per loopnest iteration
  - (8/4)*30=60
  - reduced from 80
    - which was reduced from 120
- Better spatial locality

Cache blocking with Cray Directives

CCE blocks well, but it sometimes blocks better with help

Original loopnest	Loopnest with help	Equivalent explicit code

<pre>do k = 6, nz-5
 do j = 6, ny-5
  do i = 6, nx-5
   ! stencil
  enddo
 enddo
enddo</pre>
	<pre>!dir$ blockable(j,k)
	!dir$ blockingsize(16)
	do k = 6, nz-5
	 do j = 6, ny-5
	  do i = 6, nx-5
	   ! stencil
	  enddo
	 enddo
	enddo</pre>
		<pre>do kb = 6,nz-5,16
		 do jb = 6,ny-5,16
		  do k = kb,MIN(kb+16-1,nz-5)
		   do j = jb,MIN(jb+16-1,ny-5)
		    do i = 6, nx-5
		     ! stencil
		    enddo
		   enddo
		  enddo
		 enddo
		enddo</pre>

Use the `-hlist=a` option to get a loopmark listing

- Identifies which loops were blocked
- Gives the block size the compiler chose
- See <source>.lst file

Further cache optimisations

- **If multiple loopnests process a large array**
  - First element of array will be out of cache when start second loopnest
- **Improving cache use**
  - Consider fusing the loopnests
    - Completely: just have one loopnest
    - Partial: have one outer loop, containing multiple inner loops

<table>
<tr><th>Original code</th><th>Complete fusion</th><th>Partial fusing</th></tr>
<tr><td><pre>do j = 1, Nj
 do i = 1, Ni
  a(i,j)=b(i,j)*2
 enddo
enddo

do j = 1, Nj
 do i = 1, Ni
  a(i,j)=a(i,j)+1
 enddo
enddo</pre></td><td><pre>do j = 1, Nj
 do i = 1, Ni
  a(i,j)=b(i,j)*2
  a(i,j)=a(i,j)+1
 enddo
enddo</pre></td><td><pre>do j = 1, Nj
 do i = 1, Ni
  a(i,j)=b(i,j)*2
 enddo
 do i = 1, Ni
  a(i,j)=a(i,j)+1
 enddo
enddo</pre></td></tr>
</table>

Further cache optimisations

- **Perhaps cache block before fusing**
  - Fuse one or more of the outer blocking loops
- **If multiple subprograms process the array**
  - Remove one or more outer loops (or all loops) from subprograms
  - Haul loop into parent routine, pass in index values instead
  - Might want to ensure that compiler is inlining this routine
  - This technique is very useful if you want to use OpenMP/OpenACC
- **Beware of Fortran**
  - array syntax often bad
    - `a(:,:)=b(:,:)*2`
    - `a(:,:)=a(:,:)+1`
  - compiler unlikely to fuse any loops

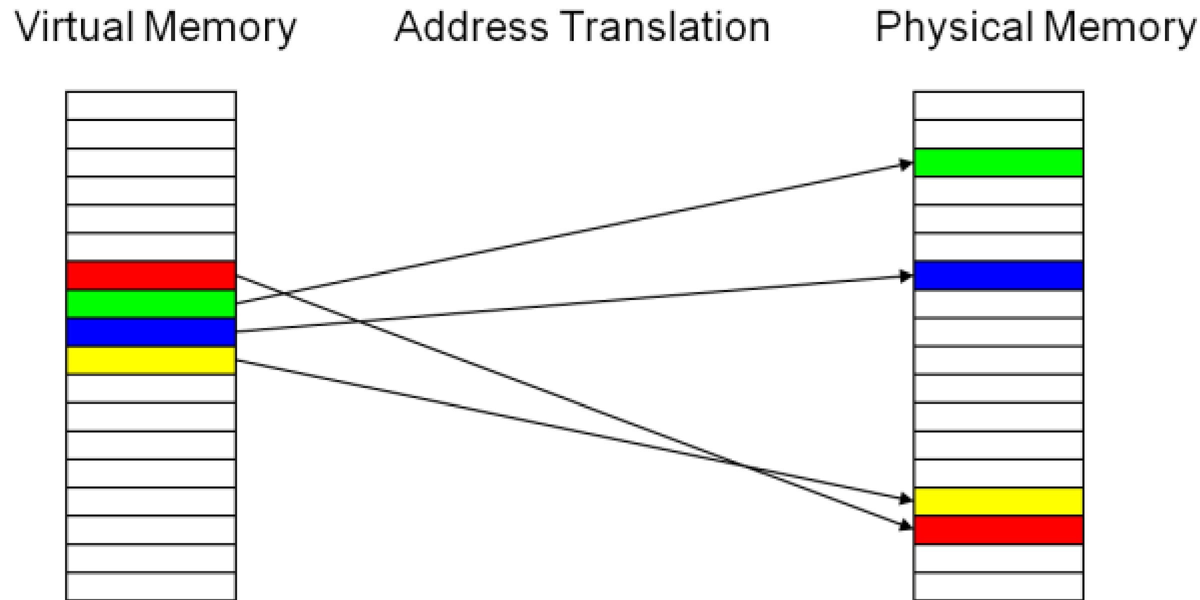
Original code

```
CALL sub1(a,b)
CALL sub2(a)

SUBROUTINE sub1(a)
 do j=1,Nj
  do i=1,Ni
   a(i,j)=b(i,j)*2
  enddo
 enddo
END SUBROUTINE sub1
```

After hauling

```
do j = 1, Nj
 CALL sub1(a,b,j)
 CALL sub2(a,j)
enddo

SUBROUTINE sub1(a,j)
 do i=1,Ni
  a(i,j)=b(i,j)*2
 enddo
END SUBROUTINE sub1
```

Virtual Memory vs Physical Memory



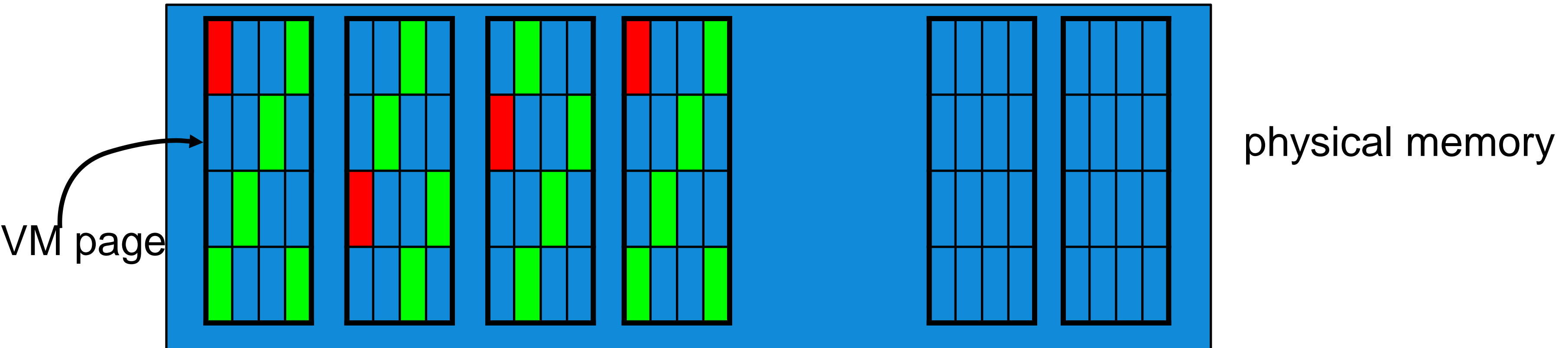
- **Translation page table is stored in main memory**
  - Each memory access logically takes twice as long – once to find the physical address, once to get the actual data
- **Use a hardware cache of least recently used addresses**
  - Called a Translation Lookaside Buffer or TLB
  - You should aim to reuse this cache wherever possible

The TLB cache

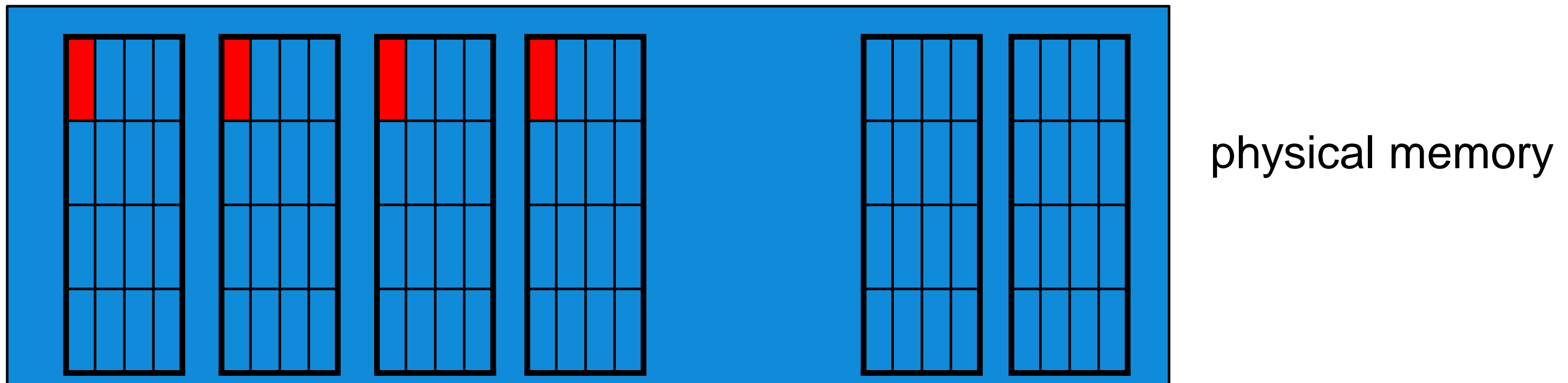
= new TLB entry created

= address already mapped

bad for the TLB
non unit stride through the data



VERY bad for the TLB
strides through the data which exceed the page size



Optimising for TLB

- **Aim to reuse data on a page**
  - i.e. treat similarly to a cache
- **Standard-sized pages are 4kB**
  - But you can use larger "huge" pages
    - 128kB, 512kB, 2MB,... 64MB
  - Almost always benefit HPC applications
    - regular data accesses)
    - huge pages give fewer TLB misses
  - Huge pages can also help communication performance
- **To use huge pages (see `man intro_hugepages`)**
  - Load chosen `craype-hugepages*` module
    - See `module avail craype-hugepages` for list of available options
    - 2M or 8M are usually most successful on Cray XE6
  - Compile as before
  - Make sure this module is also loaded in PBS jobscript
    - quick cheat: can load a different-sized hugepages module at runtime
      - compile-time module enables hugepages, runtime one determines actual size

Loop Unrolling (Theory)

- **Increases the work per loop iteration**
  - more computation per loop iteration
    - can pipeline better in CPU
    - more opportunities for vectorisation
  - higher computational intensity
    - more floating point operations per memory operation (load or store)
- **Combination of loop blocking and unwinding**
  - may completely unwind the loop
    - i.e. replace by complete set of scalar instructions

Original code	After partial unrolling
do i=1,N a(i)=a(i) + b(i) enddo	do i=1,N,4 a(i) =a(i) + b(i) a(i+1)=a(i+1) + b(i+1) a(i+2)=a(i+2) + b(i+2) a(i+3)=a(i+3) + b(i+3) enddo <cleanup if N%4!=0>

Loop Unrolling (Reality)

- **Most optimising compilers will unroll loops automatically**
  - But probably will concentrate on inner loops
- **When might we help?**
  - If the compiler didn't unroll (and should have done)
  - When the compiler doesn't know about tripcounts (this loop is small)
  - When we have a small outer loop
    - maybe move it to be the innermost loop and completely unroll
- **Avoid manually unrolling loops where possible**
  - reduces portability
  - optimal loop length for Interlagos may not suit Sandybridge
- **Instead ask compiler to do it using directives, e.g.:**
  - CCE: Force unrolling loop, optional `i` times.

```
!DIR$ unroll (i)
#pragma _CRI unroll i
```

  - PGI: Force unrolling loop, optional `i` times.

```
CPGI$  unroll n:i
#pragma loop unroll n:i
```

Vector Instructions (Vectorisation)

CRAY

- **Modern CPUs can perform multiple operations each cycle**
  - Use special SIMD (Single Instruction Multiple Data) instructions
    - e.g. SSE, AVX
  - Operate on a "vector" of data
    - typically 2 or 4 double precision floats (on Interlagos)
  - Potentially gives speedup in floating point operations
  - Usually only one loop is vectorisable in loopnest
    - And most compilers only consider inner loop

- **Optimising compilers will use vector instructions**
  - Relies on code being vectorisable
  - Or in a form that the compiler can convert to be vectorisable
    - Some compilers are better at this than others

Helping vectorisation

- **Is there a good reason for this?**
  - There is an overhead in setting up vectorisation; maybe it's not worth it
    - Could you unroll inner (or outer) loop to provide more work?

- **Does the loop have dependencies?**
  - information carried between iterations
    - e.g. counter: `total = total + a(i)`
  - No:
    - Tell the compiler that it is safe to vectorise
      - !dir$ IVDEP directive above loop (CCE, bur works with most compilers)
      - C99: restrict keyword (or compile with `-hrestrict=a` with CCE)
  - Yes:
    - Rewrite code to use algorithm without dependencies, e.g.
      - promote loop scalars to vectors (single dimension array)
      - use calculated values (based on loop index) rather than iterated counters, e.g.
        - Replace: `count = count + 2; a(count) = ...`
        - By: `a(2*i) = ...`
      - move `if` statements outside the inner loop
        - may need temporary vectors to do this
    - If you need to do too much extra work to vectorise, may not be worth it.

When does the Cray Compiler vectorise?

- **The Cray compiler will only vectorise loops**
  - Constant strides are best, indirect addressing is bad
  - Can vectorise across inlined functions
  - Needs to know loop tripcount (but only at runtime)
    - do/while loops should be avoided
  - No recursion allowed
    - if you have this, consider rewriting the loop
  - If you can't vectorise the entire loop, consider splitting it
    - so as much of the loop is vectorised as possible
- **Always check the compiler output to see what it did**
  - CCE: `-hlist=a`
  - GNU: `-ftree-vectorizer-verbose=1`
  - PGI: `-Minfo`
  - or (for the hard core) check the assembler generated
- **Clues from CrayPAT's HWPC measurements**
  - `export PAT_RT_HWPC=13` or `14` # Floating point operations SP,DP
  - Complicated, but look for ratio of operations/instructions > 1
    - expect 4 for pure AVX with double precision floats

Let's consider a non-vectorisable loop

CRAY

Look further down for associated messages

```
16.  + 1-------<    do j = 1,N
17.    1              x = xinit
18.  + 1 r4----<      do i = 1,N
19.    1 r4             x = x + vexpr(i,j)
20.    1 r4             y(i) = y(i) + x
21.    1 r4---->      end do
22.    1------->    end do
```

1.497ms

ftn-6254 ftn: VECTOR File = bufpack.F90, Line = 16

A loop starting at line 16 was **not vectorized** because a recurrence was found on "y" at line 20.

ftn-6005 ftn: SCALAR File = bufpack.F90, Line = 18

A loop starting at line 18 was **unrolled 4 times**.

ftn-6254 ftn: VECTOR File = bufpack.F90, Line = 18

A loop starting at line 18 was not vectorized because a recurrence was found on "x" at line 19.

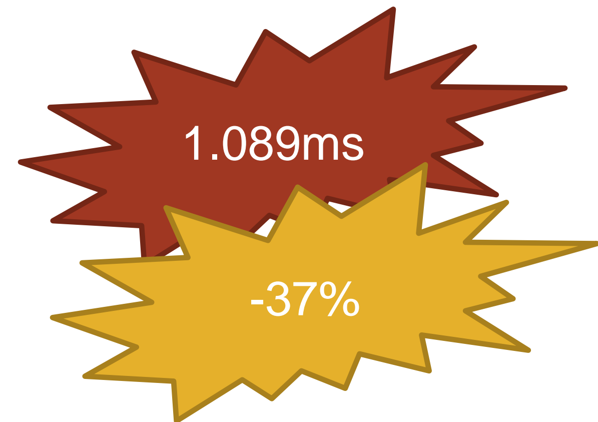
For more info, type:
`explain ftn-6254`

Now make a small modification

```
38.     Vf------<    do i = 1,N
39.     Vf             x(i) = xinit
40.     Vf------>    end do
41.
42.     ir4-----<    do j = 1,N
43.     ir4 if--<      do i = 1,N
44.     ir4 if           x(i) = x(i) + vexpr(i,j)
45.     ir4 if           y(i) = y(i) + x(i)
46.     ir4 if-->      end do
47.     ir4----->    end do
```

x promoted to vector:

- trade slightly more memory
- for better performance



ftn-6007 ftn: SCALAR File = bufpack.F90, Line = 42

A loop starting at line 42 was **interchanged** with the loop starting at line 43.

ftn-6004 ftn: SCALAR File = bufpack.F90, Line = 43

A loop starting at line 43 was **fused** with the loop starting at line 38.

ftn-6204 ftn: VECTOR File = bufpack.F90, Line = 38

A loop starting at line 38 was **vectorized**.

ftn-6208 ftn: VECTOR File = bufpack.F90, Line = 42

A loop starting at line 42 was **vectorized** as part of the loop starting at line 38.

ftn-6005 ftn: SCALAR File = bufpack.F90, Line = 42

A loop starting at line 42 was **unrolled 4 times**.

N.B. outer loop vectorisation here

Are there any questions?