Optimization Techniques I

Jan Thorbecke
jan@cray.com

CRAY THE SUPERCOMPUTER COMPANY

CSCS Swiss National Supercomputing Centre

HP2C

Methodology

1. **Optimize single core performance (this presentation)**
2. **Optimize MPI communication**
3. **Optimize IO**

For all these steps **CrayPat** is the best tool to analyze programs and find areas to tune for.

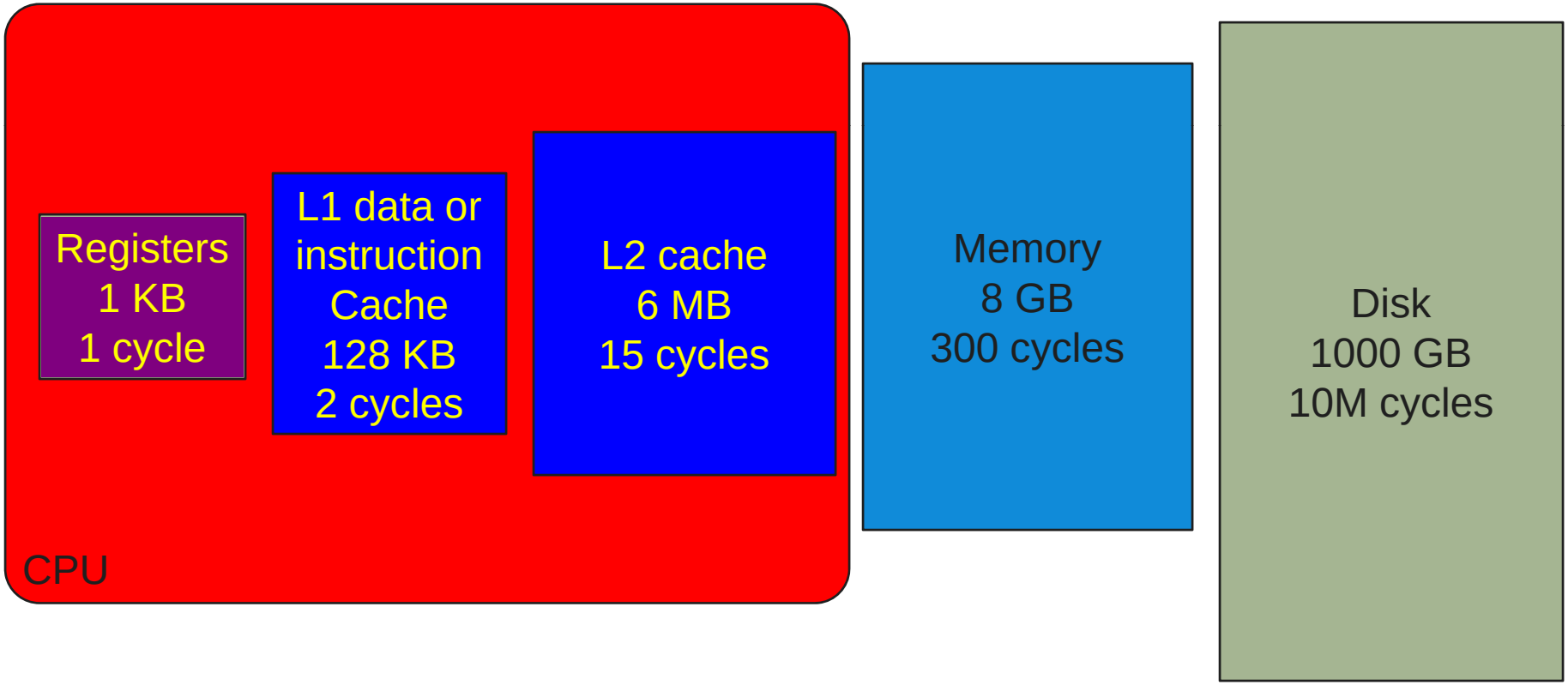
Basic CPU optimization

Outline

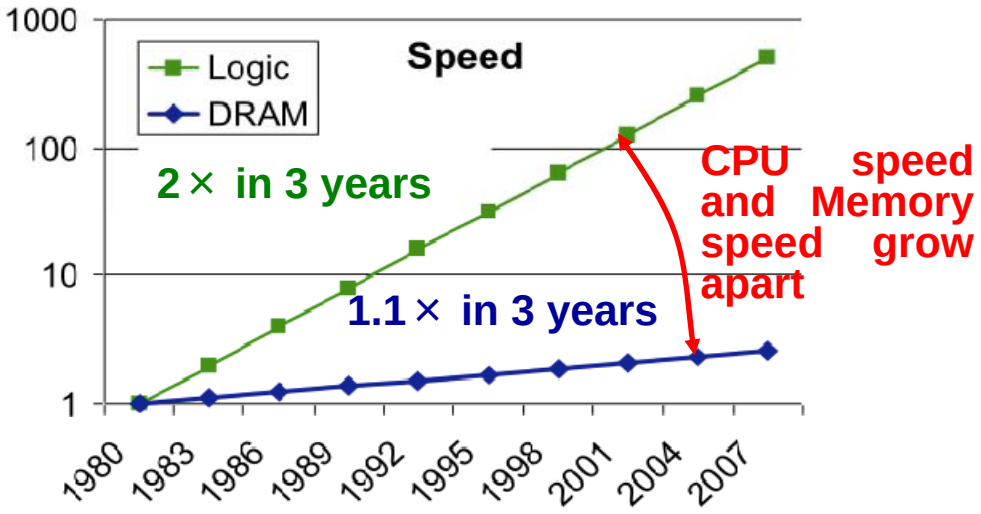
- Memory Hierarchy
- Loop Optimization
- Cache optimization
- Vectorization
- TLB for pages

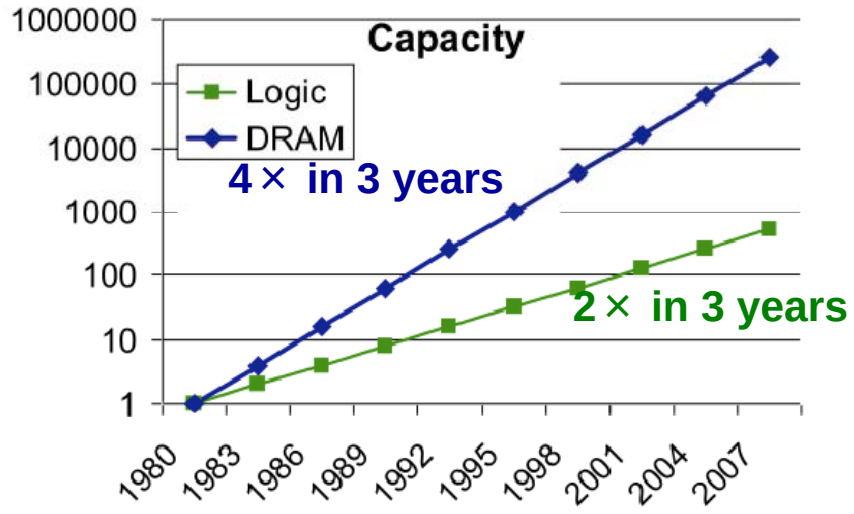
Memory Hierarchy

As you go further, capacity and latency increase



Technology Trends and Performance



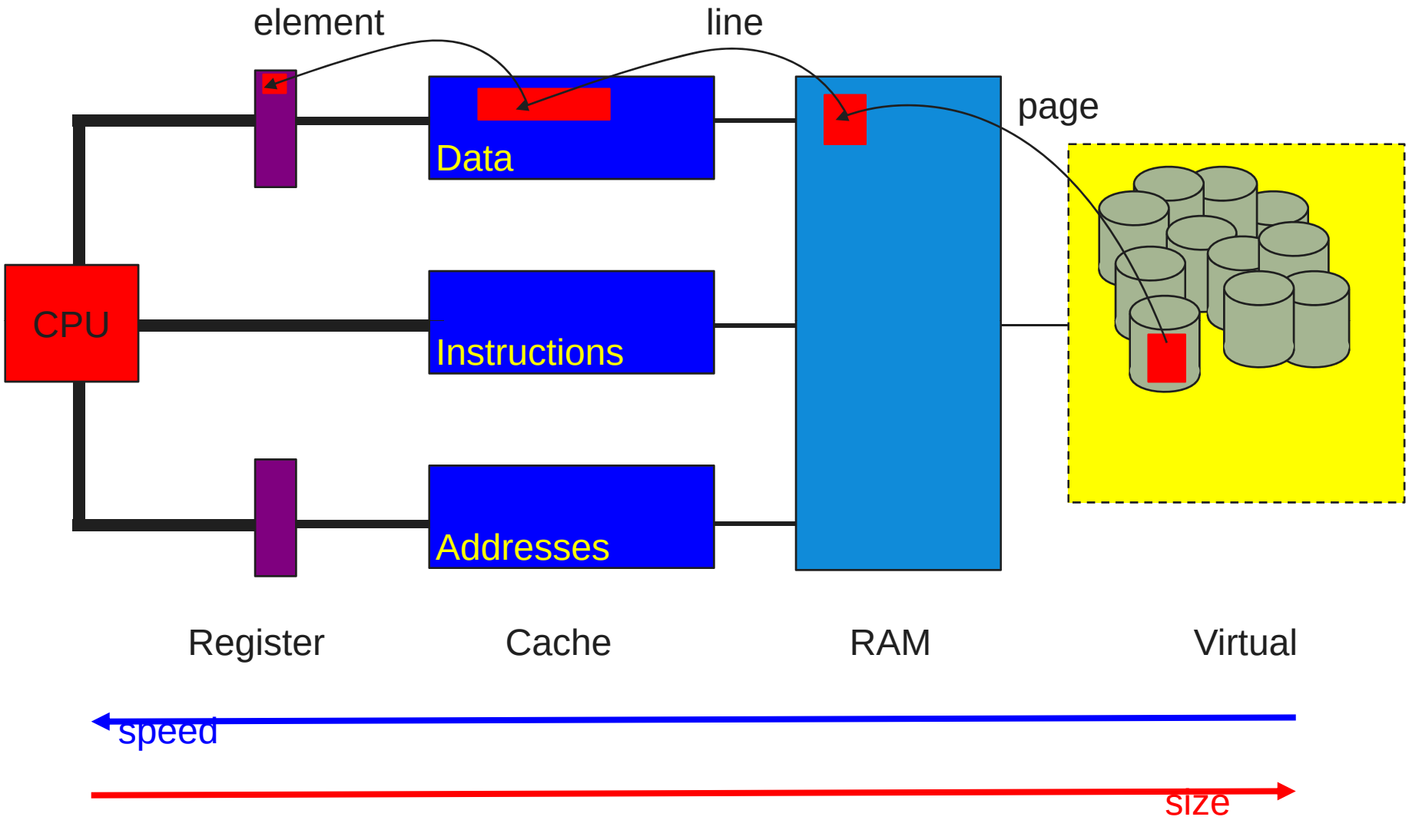


- Computing capacity: 4× per 3 years
  - If we could keep all the transistors busy all the time
  - Actual: 3.3× per 3 years
- Moore's Law: Performance is doubled every ~18 months

Summary Random Access Memory

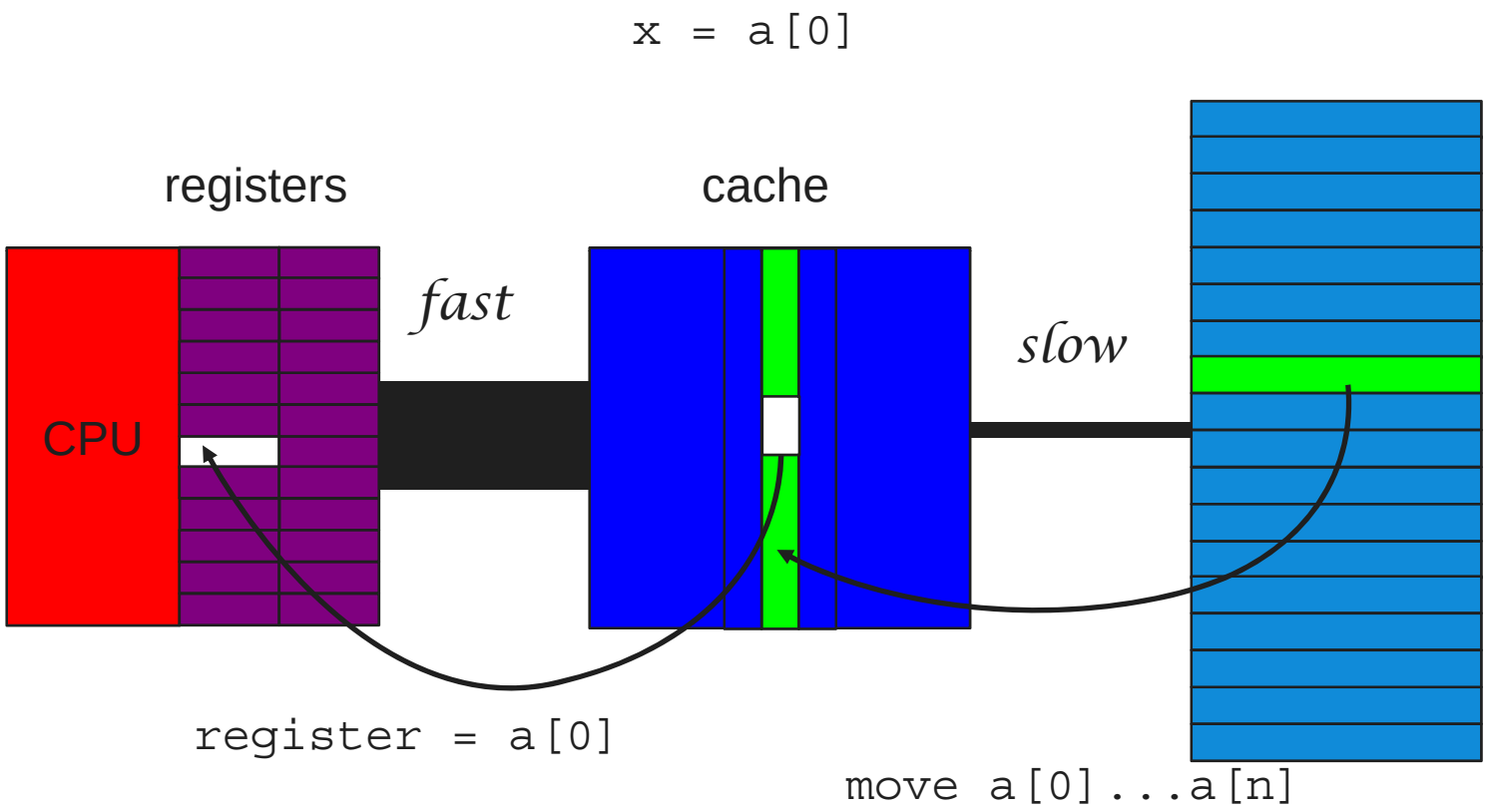
- Dynamic RAM (DRAM)
  - Each bit is stored in a capacitor
  - Uses one capacitor and one transistor per bit
  - Slower, but takes up less space in a chip
  - Must be refreshed periodically (milliseconds), since the capacitor leaks
- Static RAM (SRAM)
  - Each bit is stored in a type of flip-flop
  - Typically takes four or six transistors per bit
  - Faster, but takes up more space in a chip
  - Retains information as long as power is supplied

Memory Hierarchy

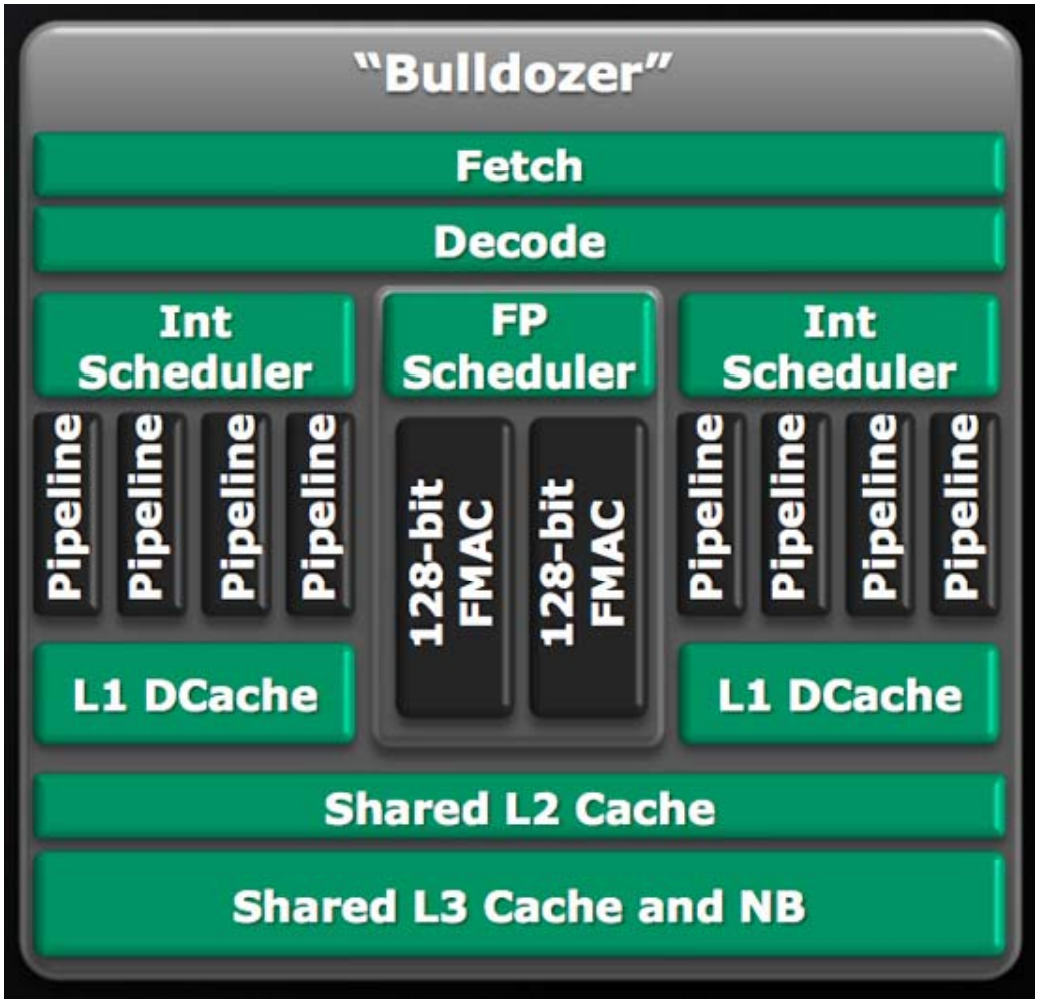


Cache Lines

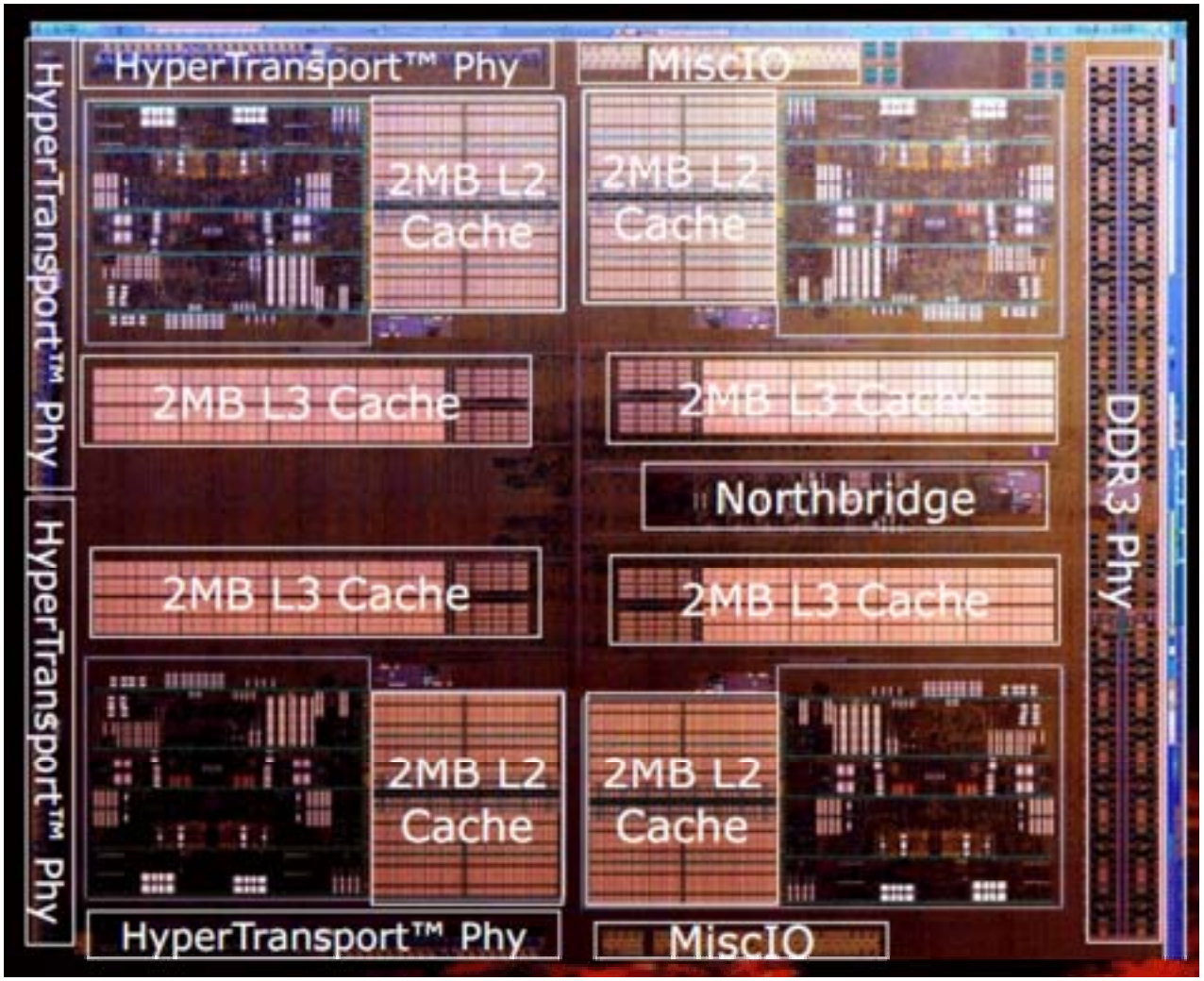
Typically more than one element at once is transferred



Interlagos Bulldozer module



Interlagos die



Basic loops optimizations techniques

CRAY THE SUPERCOMPUTER COMPANY

CSCS Swiss National Supercomputing Centre

HP2C

Loop Interchange

- **loop interchange** is the process of exchanging the order of two iteration variables.
- For example, in the code fragment:

```
for i from 0 to 10
        for j from 0 to 20
                a[i,j] = i + j;
```

loop interchange would result in:

```
for j from 0 to 20
        for i from 0 to 10
                a[i,j] = i + j
```

Loop unrolling

- Loop unrolling is the replication of loop body while at the same time incrementing the loop counter by the number of copies of that loop body

```
do i = 1,n
  a(i) = a(i) + b(i)
enddo
```

Unrolled loop:

```
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
```

Plus some clean up work

Why do loop unrolling?

- Enable register reuse
- Reduce the loop control overhead
- Improve scheduling
- Allow for more efficient software prefetching

- All excellent reasons to unroll a loop when optimizing for a microprocessor…

Circa 1990 - 2005

Loop unrolling today

- Compilers are very good at unrolling to
  - Enable register reuse
  - Improve cache reuse
  - Reduce loop control overhead
  - Improve scheduling
  - Allow for more efficient software prefetching
- Modern CPU has less need for unrolling, as they have
  - A very fast L1 cache
  - Extremely out-of-order
  - Very fast loop flow control
  - Good hardware prefetching and it is getting better all the time

Loop unrolling: when you should do it

First look to see if the compiler is already unrolling the loop for you. If it is not, consider the following cases

- Small loop body with indirect addressing or if tests
- An outer loop where unrolling would allow for a rapid reuse of a variable
- You are trying to get a very high percentage of peak and have already done everything else

- Unrolling by 'Hand' is getting less and less important

Strip mining

- Strip mining involves splitting a single loop into a nested loop. The resulting inner loop iterates over a section or strip of the original loop, and the new outer loop runs the inner loop enough times to cover all the strips, achieving the necessary total number of iterations. The number of iterations of the inner loop is known as the loop's strip length.
- Consider the Fortran code below:

```
DO I = 1, 10000
  A(I) = A(I) * B(I)
ENDDO
```

mining this loop using a strip length of 1000 yields the following loop nest

```
DO IOUTER = 1, 10000, 1000
  DO STRIP = IOUTER, IOUTER+999
        A(STRIP) = A(STRIP) * B(STRIP)
  ENDDO
ENDDO
```

Example: Matrix Multiply

- R = A B

Straight Forward Implementation (0)

```
for (i = 0 ; i < ROWS ; i++) {
    for (j = 0 ; j < COLUMNS ; j++) {
        sum = 0.0;
        for (k = 0 ; k < COLUMNS ; k++) {
            if (mask[i][k]==1) {
                sum += a[i][k]*b[k][j];
            }
        }
        r[i][j] = sum;
    }
}
```

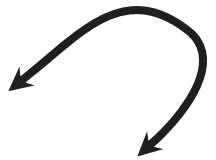
mask has 50% of its values randomly set to 1

Straight Forward Implementation (1)

```
for (i = 0 ; i < ROWS ; i++) {
    for (j = 0 ; j < COLUMNS ; j++) {
        sum = 0.0;
        for (k = 0 ; k < COLUMNS ; k++) {
            sum = sum + a[i][k]*b[k][j];
        }
        r[i][j] = sum;
    }
}
```

Another Implementation (2)

loop interchange



```
for (i = 0 ; i < ROWS ; i++) {
    for (k = 0 ; k < COLUMNS ; k++) {
        for (j = 0 ; j < COLUMNS ; j++) {
            r[i][j] = r[i][j] + a[i][k]*b[k][j];
        }
    }
}
```

Loop better for R and B

Masking to avoid zero work (2b)

```
for (i = 0 ; i < ROWS ; i++) {
    for (k = 0 ; k < COLUMNS ; k++) {
        for (j = 0 ; j < COLUMNS ; j++) {
            if (mask[i][k])
            r[i][j] = r[i][j] + a[i][k]*b[k][j];
        }
    }
}
```

Mask multiplication (2c)

```
for (i = 0 ; i < ROWS ; i++) {
    for (k = 0 ; k < COLUMNS ; k++) {
        for (j = 0 ; j < COLUMNS ; j++) {
            r[i][j] = mask[i][k]*
              (r[i][j] + a[i][k]*b[k][j]);
        }
    }
}
```

Using BLAS (3)

```
sgemm_(transa, transb, &row, &col, &col, &alpha,
       &matrix_a[0][0], &col,
       &matrix_b[0][0], &col, &beta,
       &matrix_r[0][0], &col);
```

Which is better? and why?

The Results:

straight_mask(0) = 5.402138
straightforward(1) = 5.647976
loop interchange(2a) = 0.194209
zero masks(2b) = 1.756004
mask mult(2c) = 0.312321
BLAS sgemm(3) = 0.079124

1020 vs 1024 sizes

```
-bash-3.00$ ./ClassicMatrixMultiply
Time multiply_matrices 0  = 36.274900
Time multiply matrices 1  = 36.075400
Time multiply_matrices 2  = 0.606903
Time multiply_matrices 2b = 2.313160
Time multiply_matrices 2c = 0.781150
Time multiply matrices 3  = 0.263671
```

```
-bash-3.00$ ./ClassicMatrixMultiply
Time multiply matrices 0  = 10.383177
Time multiply_matrices 1  = 10.957410
Time multiply_matrices 2  = 0.567045
Time multiply matrices 2b = 2.268844
Time multiply_matrices 2c = 0.734430
Time multiply_matrices 3  = 0.381610
```

Cache Optimization

Direct Mapped Caches

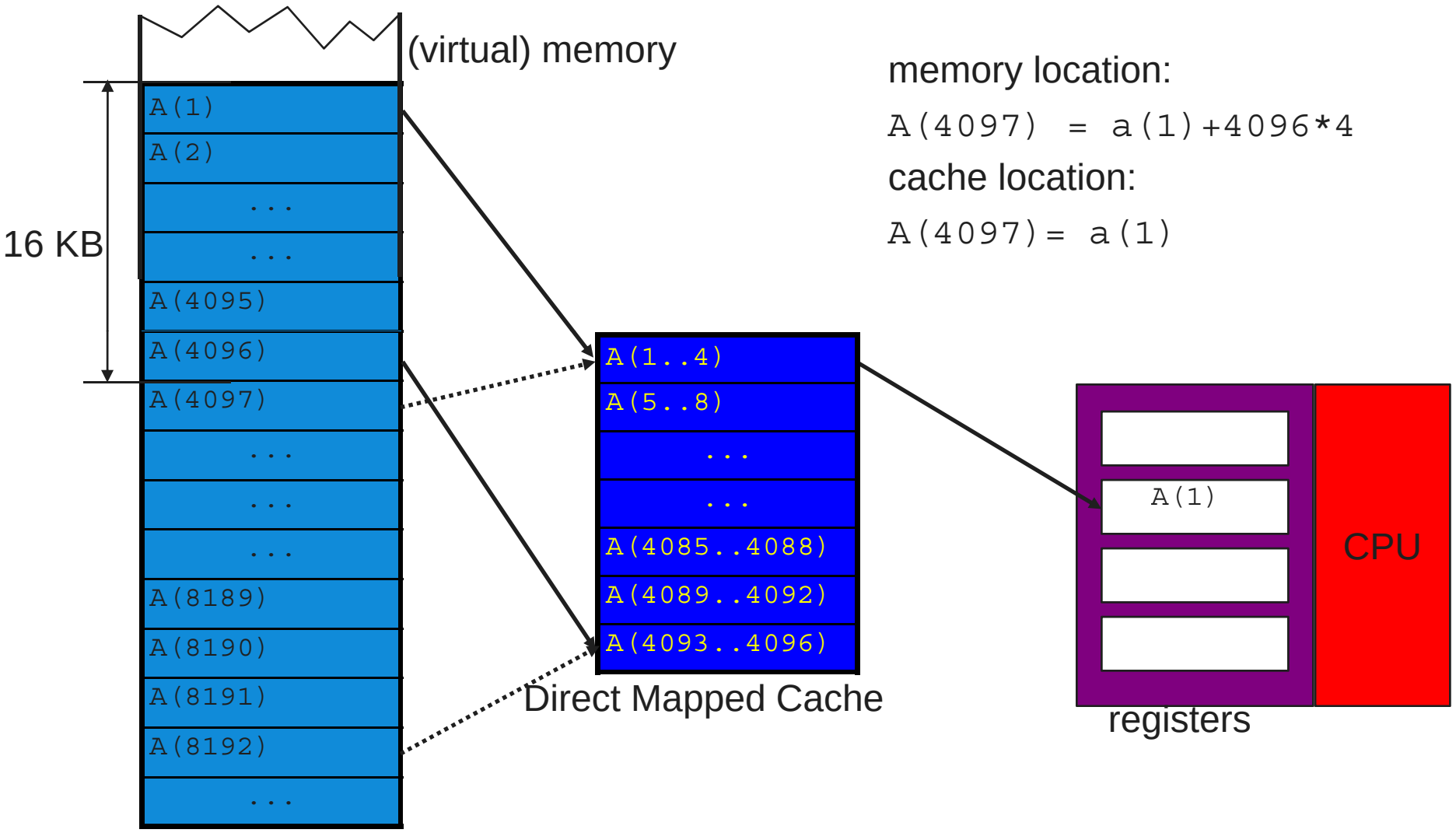
- Replacement Formula:

cache location = (memory address) modulo (cache size in words)

An example:

- Assume that a cache line is 4 words (=16 Bytes)
- Cache size = 16 KB = 16 (line size)*1024 (# of lines) Bytes
- This corresponds to 4*1024=4096 32-bit words
- Example: element 5000 goes to cache line (5000%4096=904)
- We have to load an array A with 8192 32-bit elements: i.e. twice the size of the cache

Direct Mapped Caches



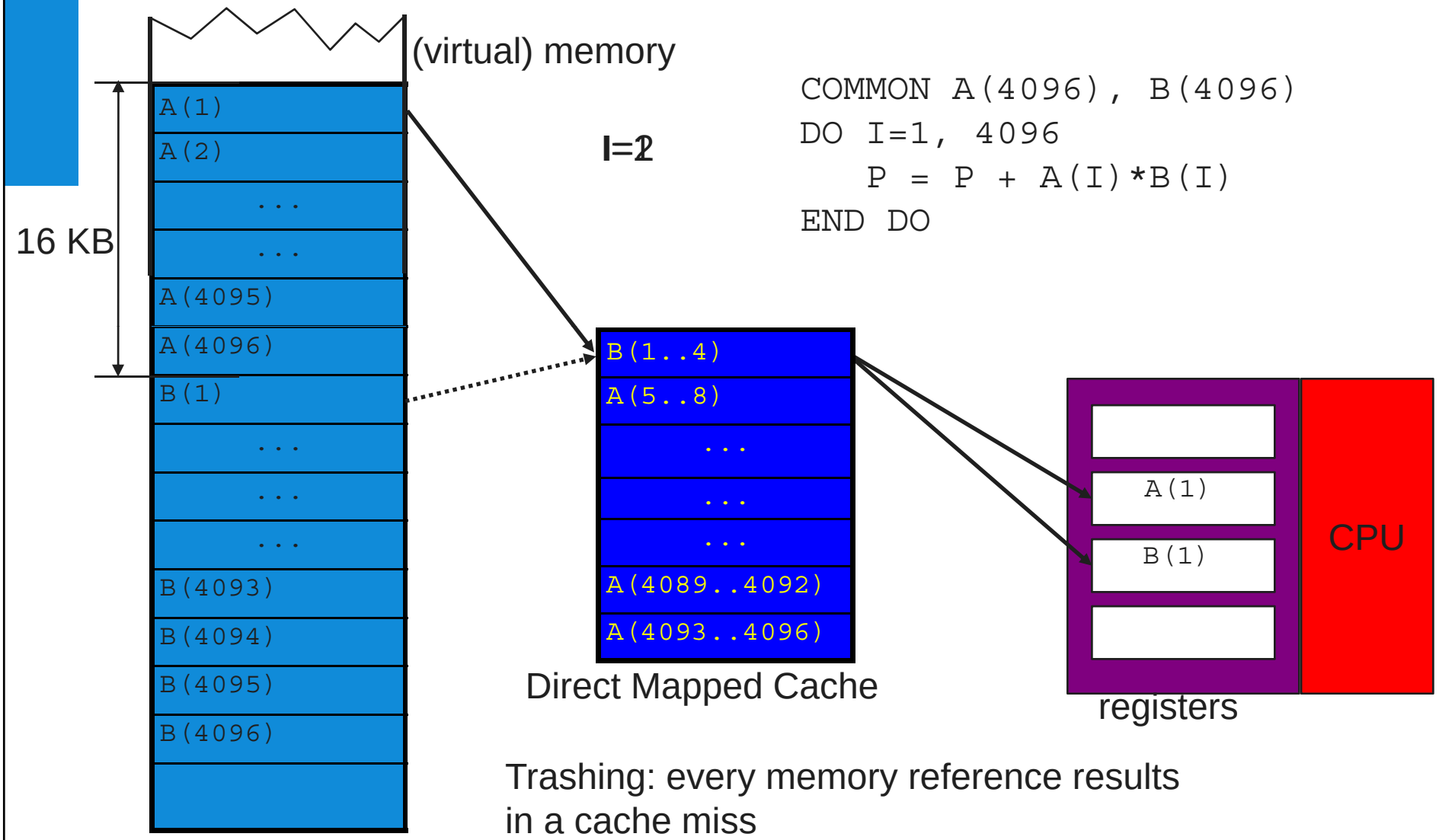
memory location:

`A(4097) = a(1)+4096*4`

cache location:

`A(4097)= a(1)`

Direct Mapped Caches - Trashing



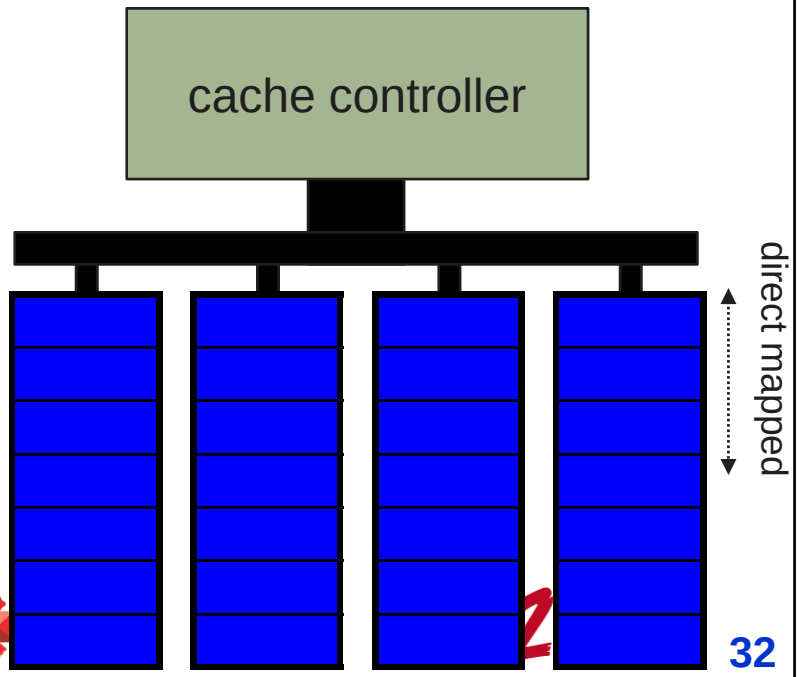
Trashing: every memory reference results in a cache miss

Set Associative Caches

Set Associative:

- The cache contains several direct mapped caches
- Data can go into one of these caches (called a ‘set’)
- The choice of a set is often (semi-) LRU

4-way set associative cache



Cache usage example

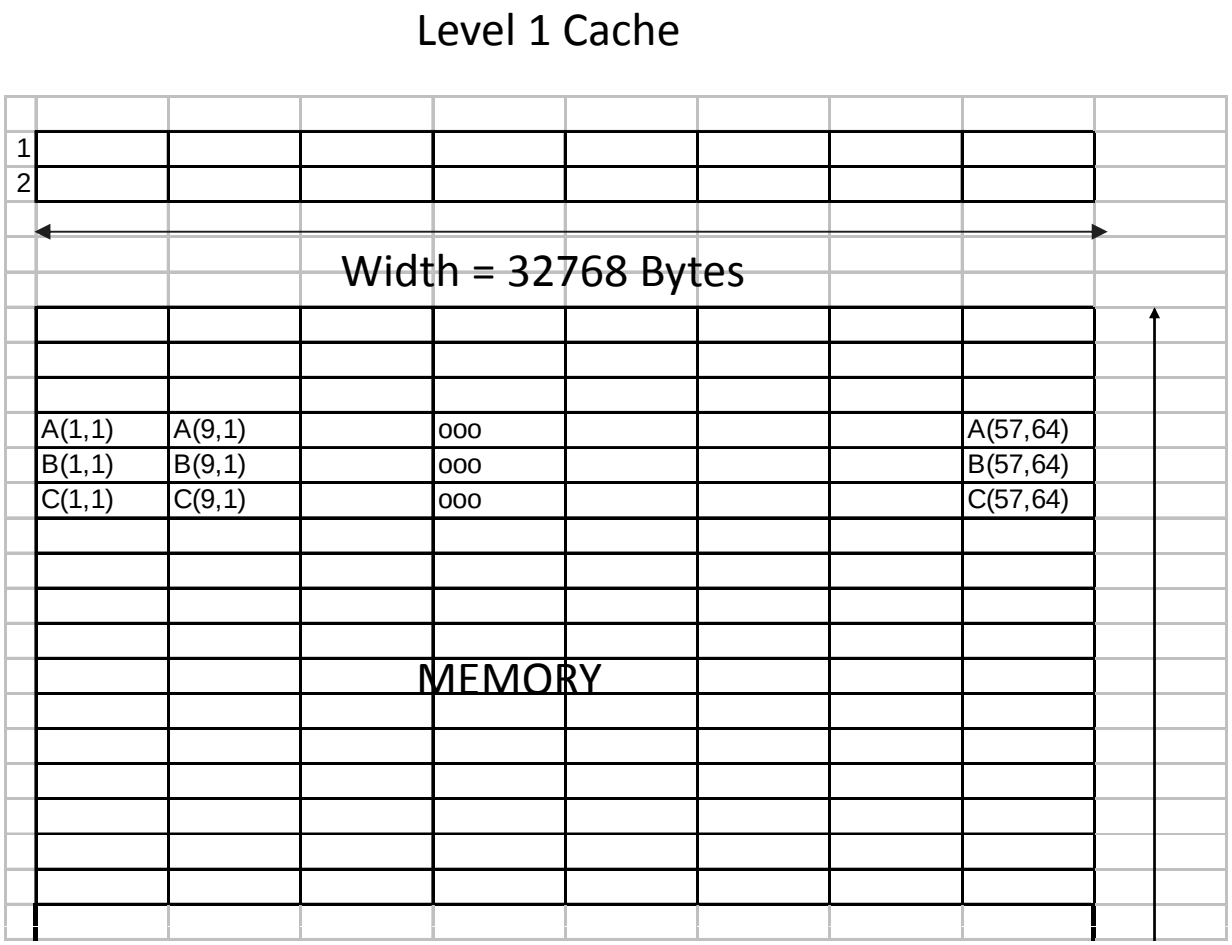
- Based on John Leveques presentation

Consider the following example

```
Real * 8        A(64,64),B(64,64),C(64,64)

DO I = 1,N
      C(I,1) = A(I,1) +B(I,1)
ENDDO
```

Memory and Cache Layout Visualization



Level 1 Cache
- 65536 B
- 1024 Lines
- 8192 8B Ws
- 16384 4B Ws
- 2 way Assoc

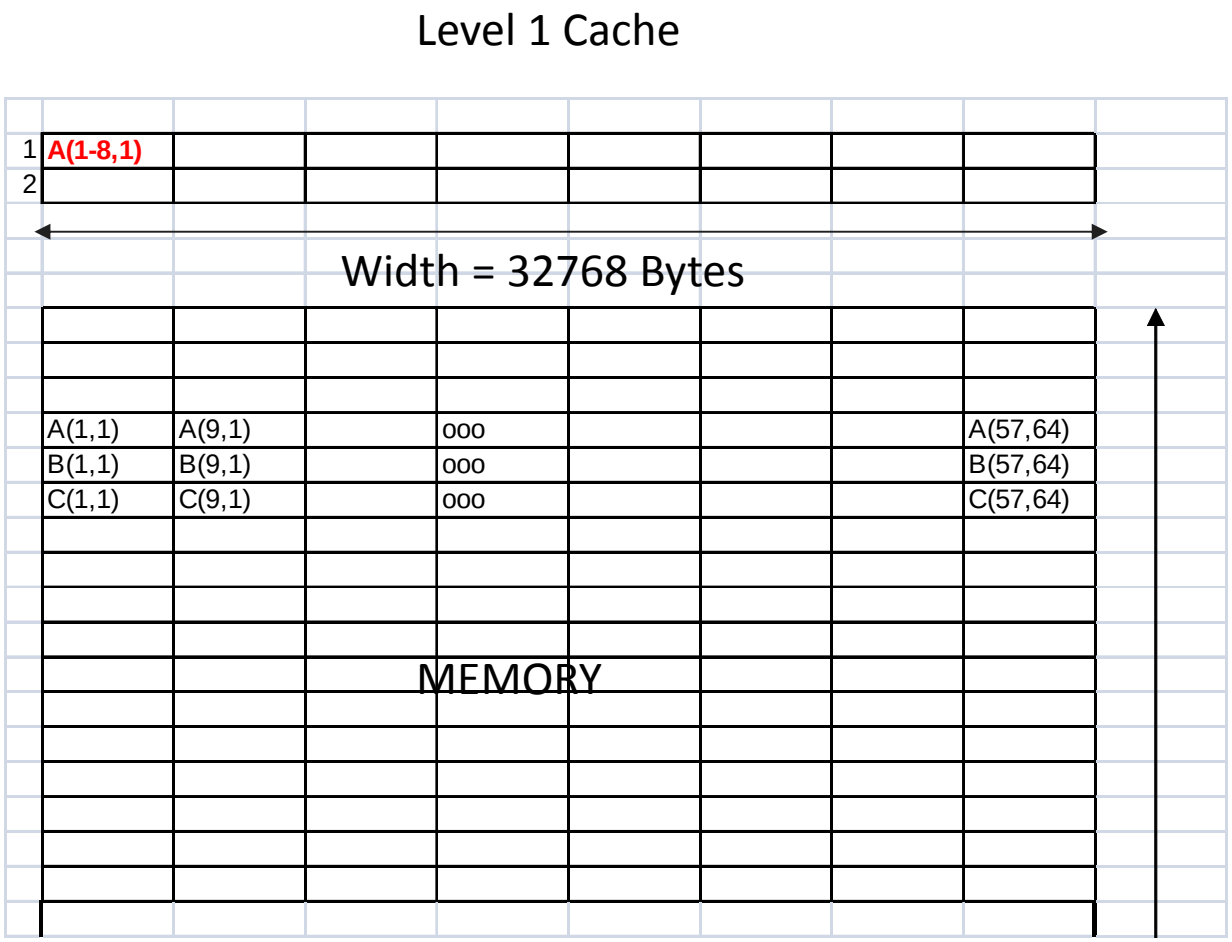
Associativity Class
- 32768 B
- 512 Lines
- 4096 8B Ws
- 8192 4B Ws

8 elements in one cache line
64*64*8 = 32768 B

Step 1 : Get the first element A(1,1)

Real * 8	A(64,64),B(64,64),C(64,64)				
DO I = 1,N					
C(I,1) = A(I,1) +B(I,1)					
ENDDO					
Fetch A(1,1)		Fetch from M	Uses 1 Associativity Class		

A(1-8,1) is loaded into the cache



Level 1 Cache

65536 B

1024 Lines

8192 8B Ws

16384 4B Ws

2 way Assoc

Associativity Class

32768 B

512 Lines

4096 8B Ws

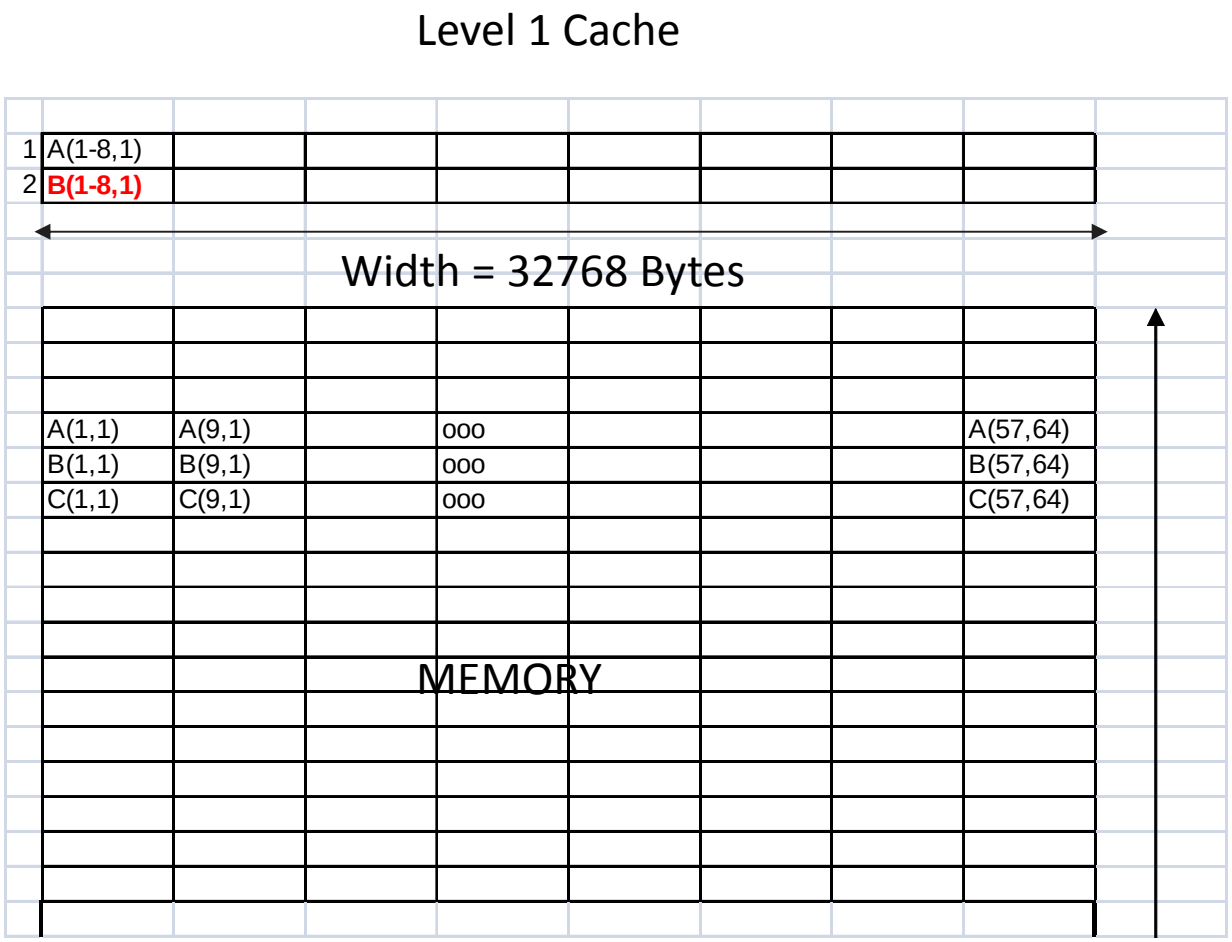
8192 4B Ws

64*64*8 = 32768 B

Step 2 : Load B(1,1)

Real * 8	A(64,64),B(64,64),C(64,64)				
DO I = 1,N					
C(I,1) = A(I,1) +B(I,1)					
ENDDO					
Fetch A(1,1)		Fetch from M	Uses 1 Associativity Class		
Fetch B(1,1)		Fetch from M	Uses 2 Associativity Class		

B(1-8,1) is loaded into the cache



Level 1 Cache

65536 B

1024 Lines

8192 8B Ws

16384 4B Ws

2 way Assoc

Associativity Class

32768 B

512 Lines

4096 8B Ws

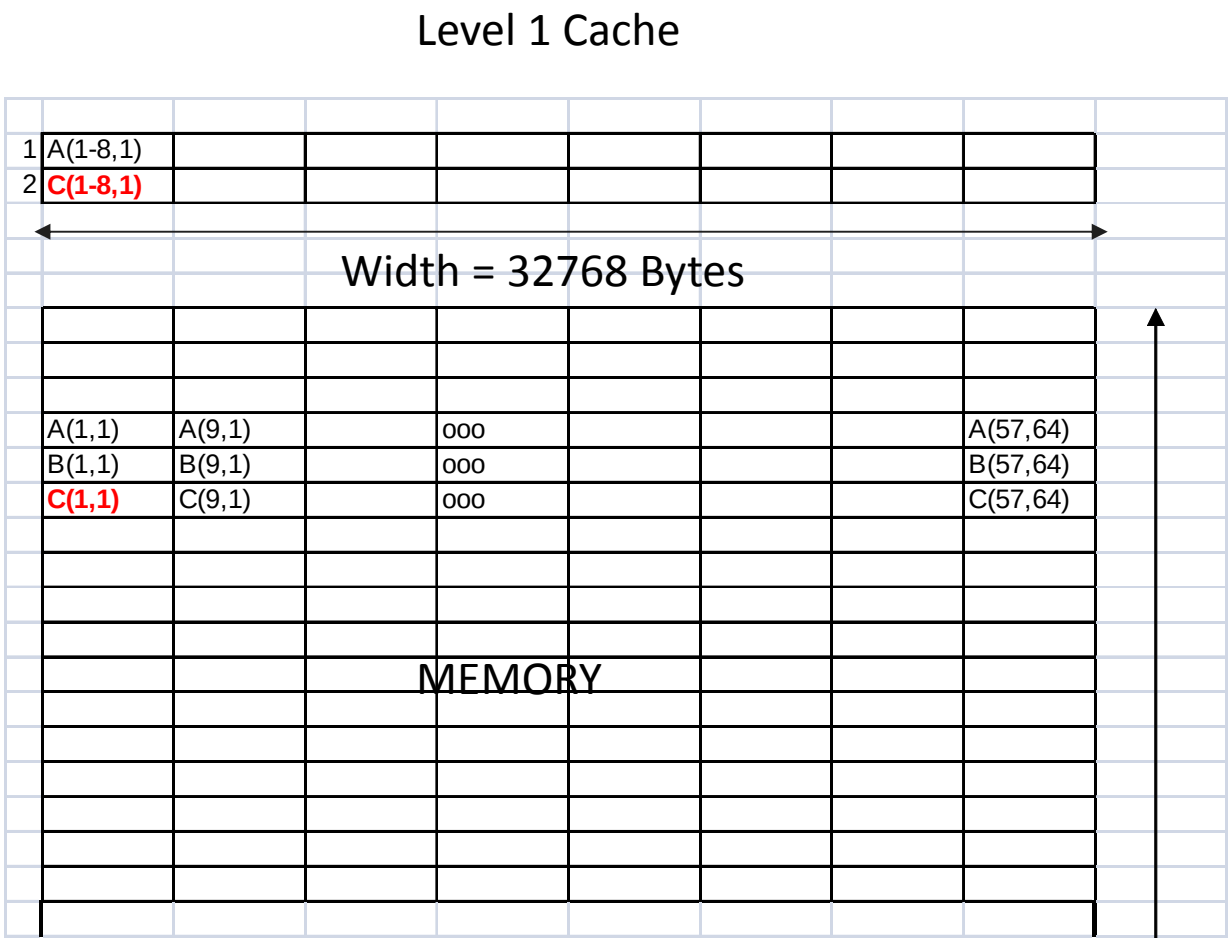
8192 4B Ws

64*64*8 = 32768 B

Step 3 : Load C(1,1), needed even if it‘s not read

Real * 8	A(64,64),B(64,64),C(64,64)				
DO I = 1,N					
C(I,1) = A(I,1) +B(I,1)					
ENDDO					
Fetch A(1,1)		Fetch from M	Uses 1 Associativity Class		
Fetch B(1,1)		Fetch from M	Uses 2 Associativity Class		
Add A(1,1) + B(1,1)					
Store C(1,1)		Fetch from M	Overwrites either 1 or 2 Associativity Class		

C(1-8,1) is loaded, B(1-8,1) is removed



Level 1 Cache

- 65536 B
- 1024 Lines
- 8192 8B Ws
- 16384 4B Ws
- 2 way Assoc

Associativity Class

- 32768 B
- 512 Lines
- 4096 8B Ws
- 8192 4B Ws

64*64*8 = 32768 B

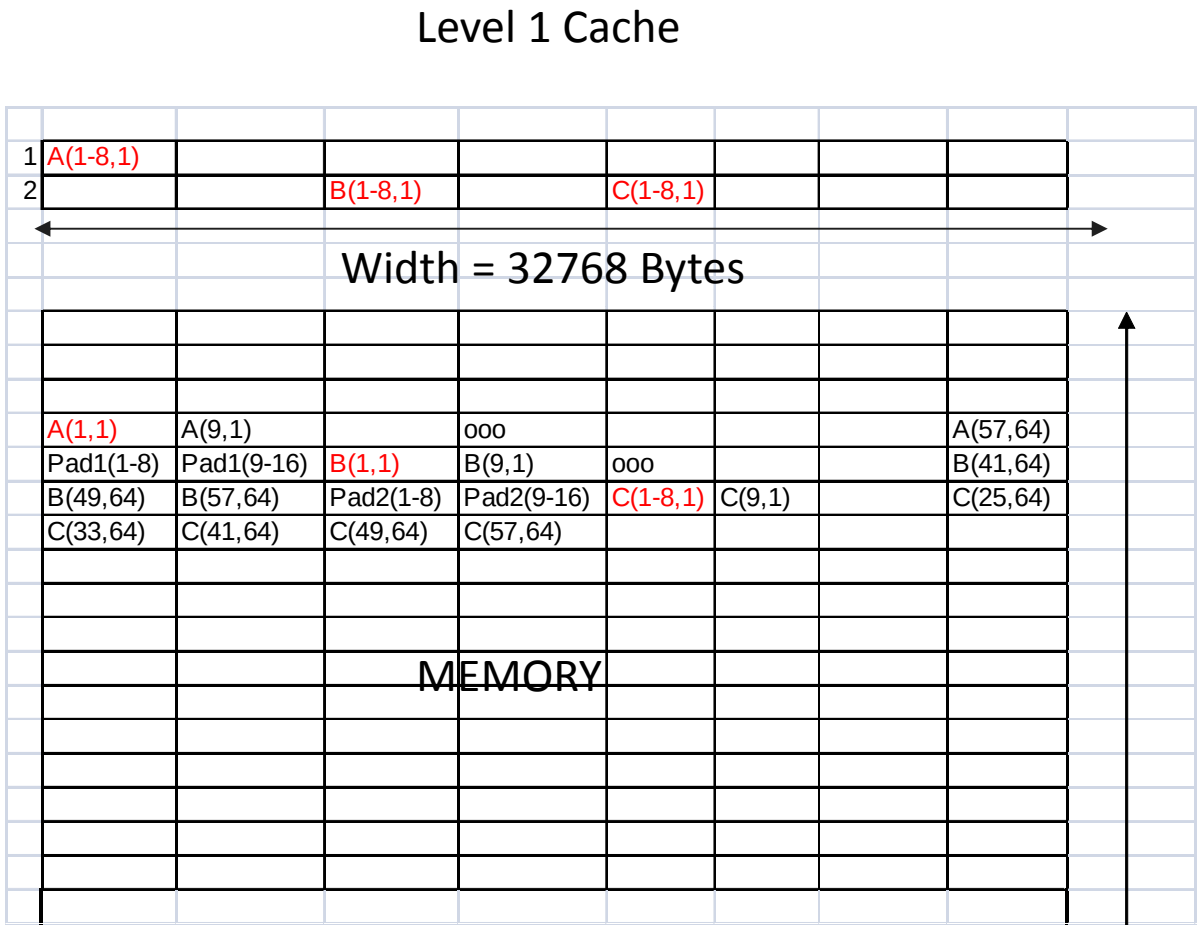
What happens

Real * 8	A(64,64),B(64,64),C(64,64)				
DO I = 1,N					
C(I,1) = A(I,1) +B(I,1)					
ENDDO					
Fetch A(1,1)		Fetch from M	Uses 1 Associativity Class		
Fetch B(1,1)		Fetch from M	Uses 2 Associativity Class		
Add A(1,1) + B(1,1)					
Store C(1,1)		Fetch from M	Overwrites either 1 or 2 Associativity Class		
Fetch A(2,1)		Fetch from L2	Overwrites either 1 or 2 Associativity Class		
Fetch B(2,1)		Fetch from L2	Overwrites either 1 or 2 Associativity Class		
Add A(2,1) + B(2,1)					
Store C(2,1)		Fetch from L2	Overwrites either 1 or 2 Associativity Class		

Must be a better Way :
Padding to change the memory layout

Real * 8	A(64,64),pad1(16),B(64,64),pad2(16),C(64,64)				
DO I = 1,N					
C(I,1) = A(I,1) +B(I,1)					
ENDDO					

Cache and memory layout with padding



Level 1 Cache

65536 B

1024 Lines

8192 8B Ws

16384 4B Ws

2 way Assoc

Associativity Class

32768 B

512 Lines

4096 8B Ws

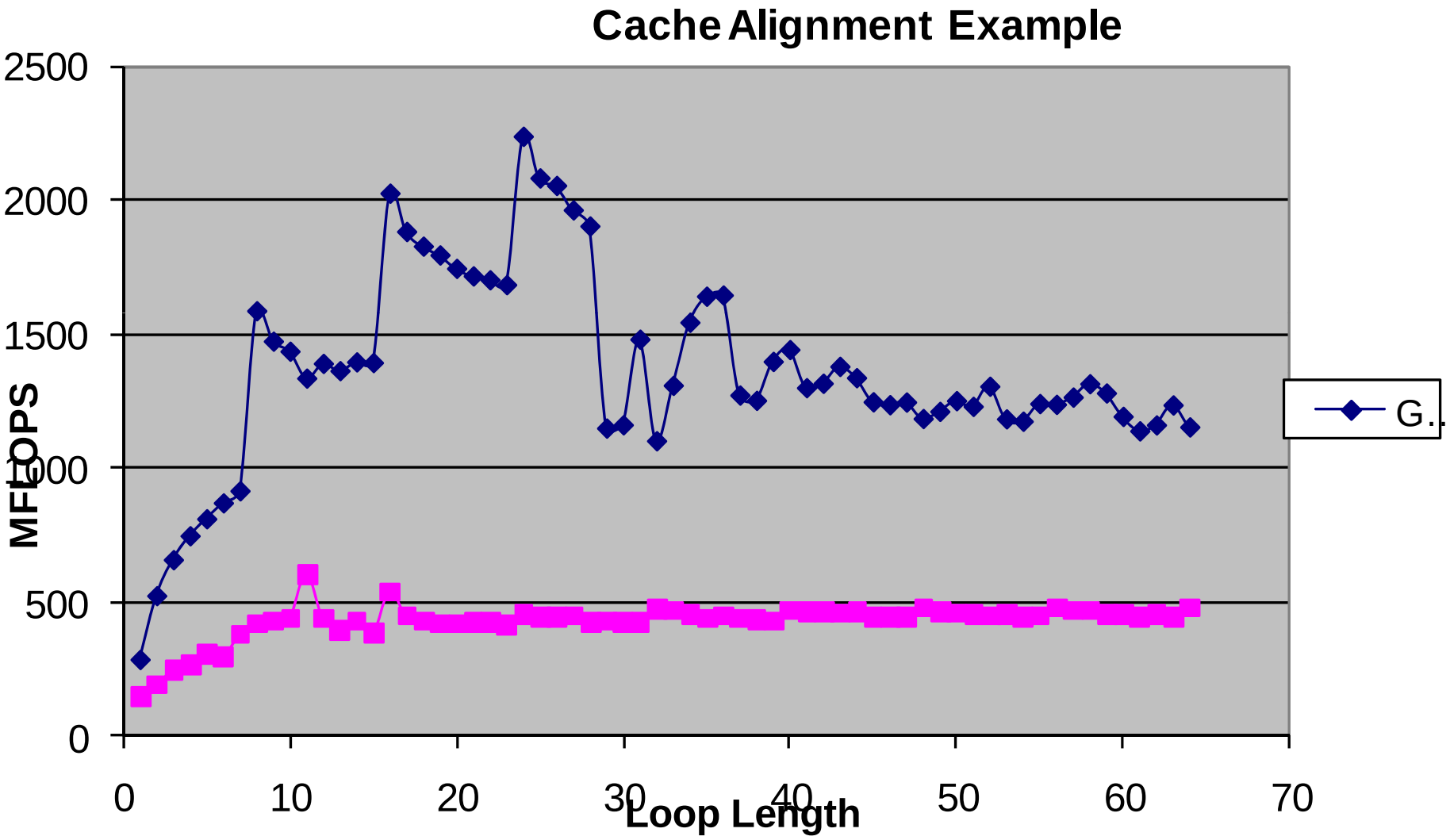
8192 4B Ws

64*64*8 = 32768 B

More reuse of cache

Real * 8	A(64,64),pad1(16),B(64,64),pad2(16),C(64,64)				
DO I = 1,N					
C(I,1) = A(I,1) +B(I,1)					
ENDDO					
Fetch A(1)		Uses 1 Associativity Class			
Fetch B(1)		Uses 2 Associativity Class			
Add A(1) + B(1)					
Store C(1)		Uses 1 Associativity Class			
Fetch A(2)		Gets from L1 Cache			
Fetch B(2)		Gets from L1 Cache			
Add A(2) + B(2)					
Store C(2)		Gets from L1 Cache			

Performance difference



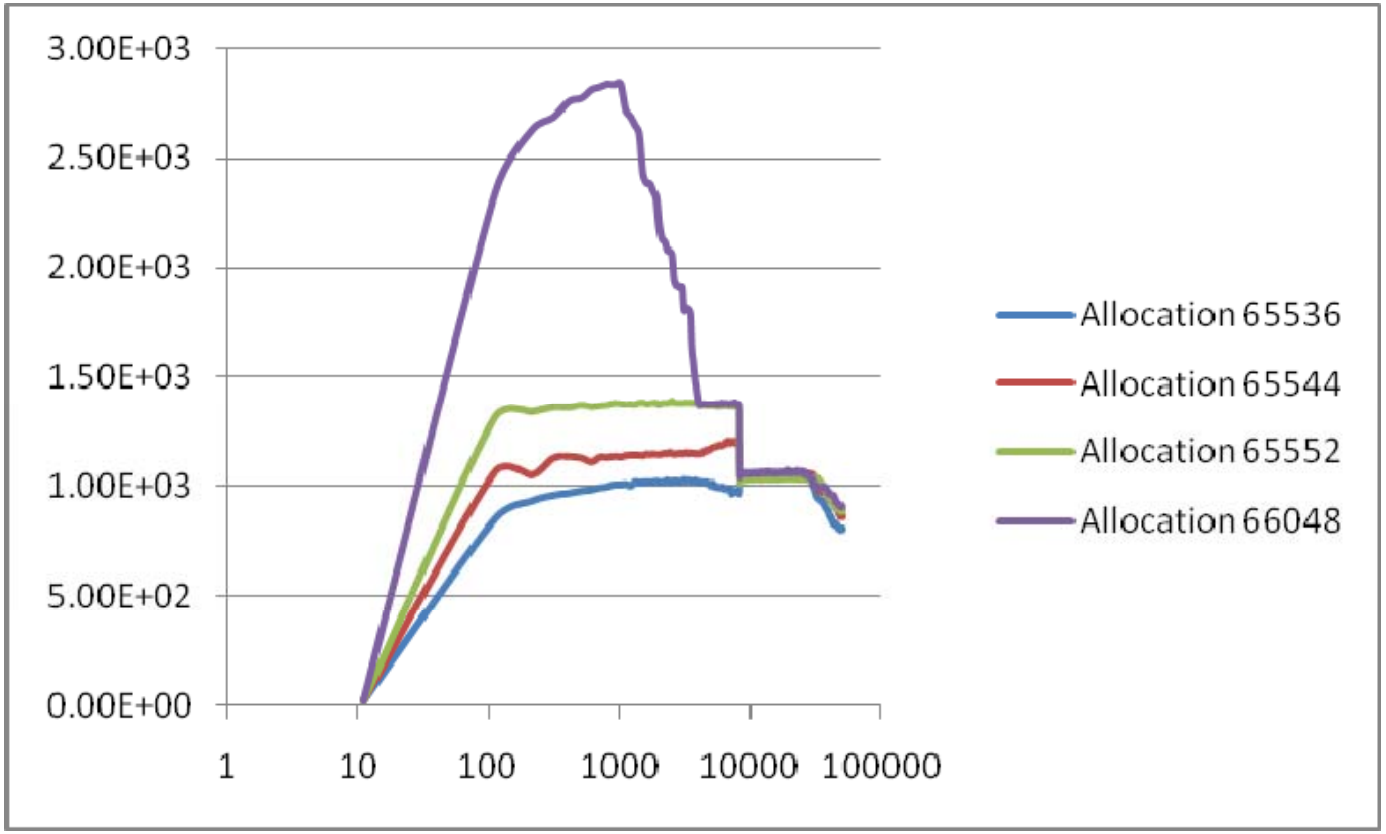
Bad Cache Alignment

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

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

Performance = F(Cache Utilization)



Cache Blocking from Start to Finish

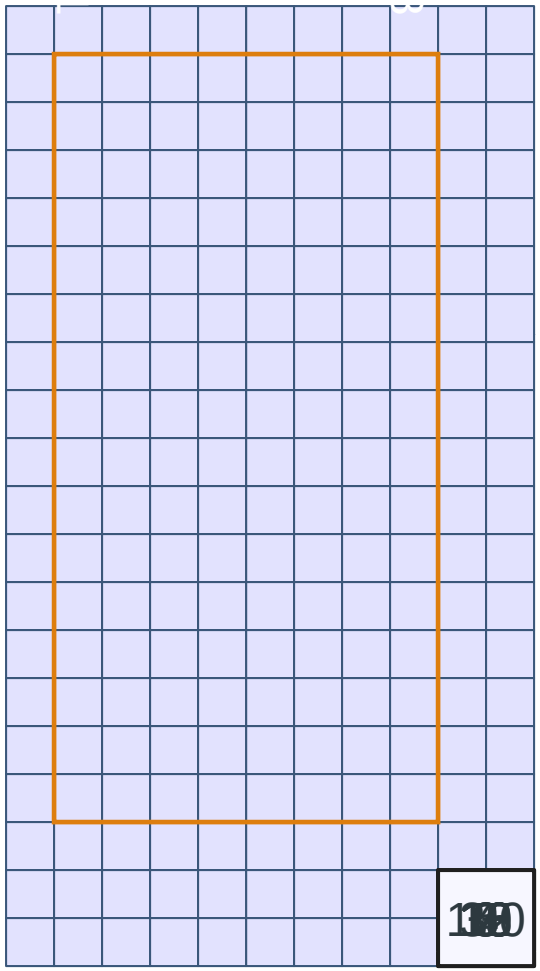
CSCS Workshop
November 2011

Based on Steve Whalen's work

Overview

- **Cache blocking** is a combination of strip mining and loop interchange, designed to increase data reuse.
  - Takes advantage of temporal reuse: re-reference array elements already referenced
  - Good blocking will take advantage of spatial reuse: work with the cache lines!
- Many ways to block any given loop nest
  - Which loops get blocked?
  - What block size(s) to use?
- Analysis can reveal which ways are beneficial
- But trial-and-error is probably faster

Cache Use in Stencil Computations

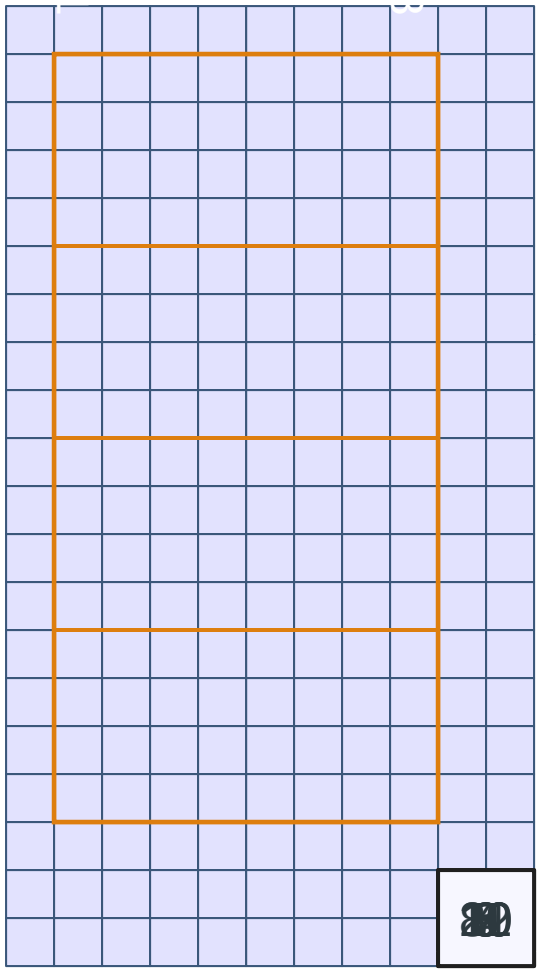


- 2D Laplacian

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

- Cache structure for this example:
  - Each line holds 4 array elements
  - Cache can hold 12 lines of **u** data
- No cache reuse between outer loop iterations

Blocking to Increase Reuse

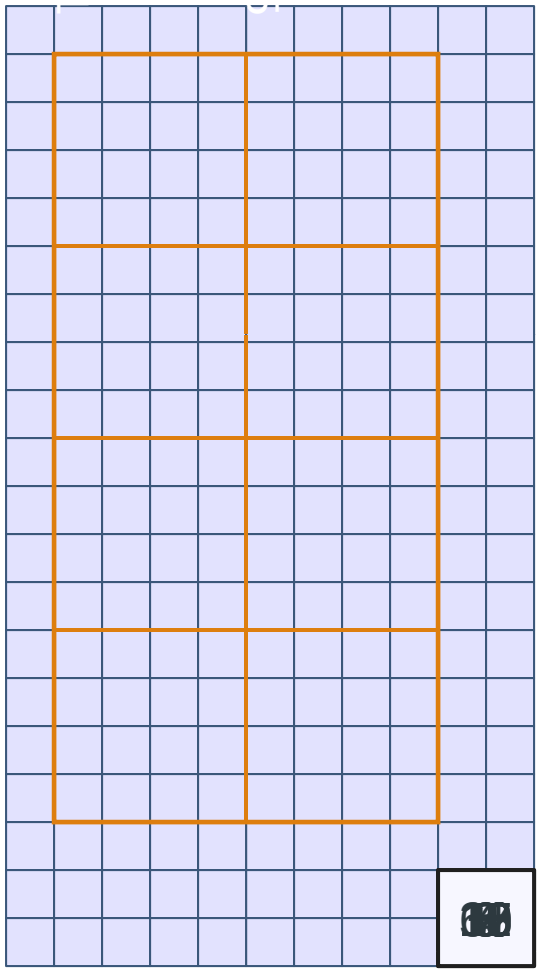


- Unblocked loop: 120 cache misses
- Block the inner loop

```
do IBLOCK = 1, 16, 4
   do j = 1, 8
      do i = IBLOCK, IBLOCK + 3
         a(i,j) = u(i-1,j) + u(i+1,j)  &
                  - 4*u(i,j)           &
                  + u(i,j-1) + u(i,j+1)
      end do
   end do
end do
```

- Now we have reuse of the "j+1" data

Blocking to Increase Reuse



- One-dimensional blocking reduced misses from 120 to 80
- Iterate over 4 × 4 blocks

```
do JBLOCK = 1, 8, 4
   do IBLOCK = 1, 16, 4
      do j = JBLOCK, JBLOCK + 3
         do i = IBLOCK, IBLOCK + 3
            a(i,j) = u(i-1,j) + u(i+1,j)  &
                     - 4*u(i,j)           &
                     + u(i,j-1) + u(i,j+1)
         end do
      end do
   end do
end do
```

- Better use of spatial locality (cache lines)

What Could Go Wrong?

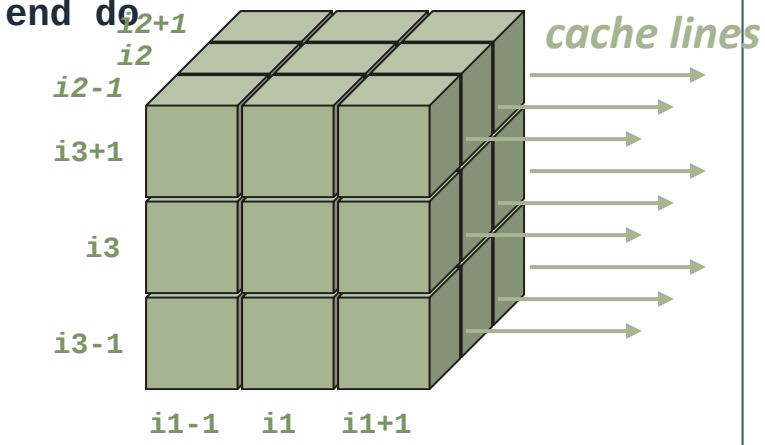
"I tried cache-blocking my code, but it didn't help"

- You're doing it wrong
  - Your block size is too small (too much loop overhead)
  - Your block size is too big (data is falling out of cache)
  - You're targeting the wrong cache level (?)
  - You haven't selected the correct subset of loops to block
- The compiler is already blocking that loop
- Prefetching is acting to minimize cache misses
- Computational intensity within the loop nest is very large, making blocking less important.

A Real-Life Example: NPB MG

- Multigrid PDE solver
- Class D, 64 MPI ranks
  - Global grid is 1024 × 1024 × 1024
  - Local grid is 258 × 258 × 258
- Two similar loop nests account for >50% of run time
- 27-point 3D stencil
  - There is good data reuse along leading dimension, even without blocking

```
do i3 = 2, 257
   do i2 = 2, 257
      do i1 = 2, 257
!        update u(i1,i2,i3)
!        using 27-point stencil
      end do
   end do
end do
```



I'm Doing It Wrong

- Block the inner two loops
- Creates blocks extending along **i3** direction

```
do I2BLOCK = 2, 257, BS2
   do I1BLOCK = 2, 257, BS1
      do i3 = 2, 257
         do i2 = I2BLOCK,                    &
                 min(I2BLOCK+BS2-1, 257)
            do i1 = I1BLOCK,                 &
                    min(I1BLOCK+BS1-1, 257)
!              update u(i1,i2,i3)
!              using 27-point stencil
            end do
         end do
      end do
   end do
end do
```

Block size	Mop/s/process
unblocked	*531.50*
16 × 16	279.89
22 × 22	321.26
28 × 28	358.96
34 × 34	385.33
40 × 40	408.53
46 × 46	443.94
52 × 52	468.58
58 × 58	470.32
64 × 64	512.03
70 × 70	506.92

That's Better

- Block the outer two loops
- Preserves spatial locality along **i1** direction

```
do I3BLOCK = 2, 257, BS3
   do I2BLOCK = 2, 257, BS2
      do i3 = I3BLOCK,                    &
              min(I3BLOCK+BS3-1, 257)
         do i2 = I2BLOCK,                 &
                 min(I2BLOCK+BS2-1, 257)
            do i1 = 2, 257
!              update u(i1,i2,i3)
!              using 27-point stencil
            end do
         end do
      end do
   end do
end do
```

Block size	Mop/s/process
unblocked	*531.50*
16 × 16	674.76
22 × 22	680.16
28 × 28	688.64
34 × 34	683.84
40 × 40	698.47
46 × 46	689.14
52 × 52	706.62
58 × 58	692.57
64 × 64	703.40
70 × 70	693.87

Example : Using Cray Directives

CCE blocks well, but it sometimes blocks better with help

Exercise 1 original loop	Exercise 1 loop with help

do k = 6, nz-5
  do j = 6, ny-5
  do i = 6, nx-5
  ! *stencil*
  end do
  end do
end do
	!dir$ blockable(j,k)
	!dir$ blockingsize(16)
	  do k = 6, nz-5
	  do j = 6, ny-5
	  do i = 6, nx-5
	  ! *stencil*
	  end do
	  end do
	  end do

- Use the **-r a** option to get a loopmark listing
  - Identifies which loops were blocked
  - Gives the block size the compiler used

Vectorization on AMD

And Intel as well

Vectorization through SSE : What is it ?

- **Streaming SIMD Extensions** (**SSE**) is a SIMD instruction set extension to the x86 architecture
- SSE originally added eight new 128-bit registers known as XMM0 through XMM7. The AMD64 extensions from AMD (originally called *x86-64* and later duplicated by Intel) add a further eight registers XMM8 through XMM15. There is also a new 32-bit control/status register, MXCSR. The registers XMM8 through XMM15 are accessible only in 64-bit operating mode.
- Each register packs together:
  - four 32-bit single-precision floating point numbers or
  - two 64-bit double-precision floating point numbers or
  - two 64-bit integers or
  - four 32-bit integers or
  - eight 16-bit short integers or
  - sixteen 8-bit bytes or characters

SSE : Example

- **Example :**
  The following simple example demonstrates the advantage of using SSE. Consider an operation like vector addition, which is used very often in computer graphics applications. To add two single precision, 4-component vectors together using x86 requires four floating point addition instructions
  vec_res.x = v1.x + v2.x;
  vec_res.y = v1.y + v2.y;
  vec_res.z = v1.z + v2.z;
  vec_res.w = v1.w + v2.w;
  This would correspond to four x86 FADD instructions in the object code. On the other hand, as the following pseudo-code shows, a single 128 bit '**packed-add**' instruction can replace the four scalar addition instructions.
  movaps xmm0,address-of-v1 ;xmm0=v1.w | v1.z | v1.y | v1.x
  addps xmm0,address-of-v2 ;xmm0=v1.w+v2.w | v1.z+v2.z | v1.y+v2.y | v1.x+v2.x movaps address-of-vec_res,xmm0

SSE

- The AMD Opteron is capable of generating 4 flops/clock in 64 bit mode and 8 flops/clock for 32 bit mode
  - Assembler must contain SSE instructions
  - Compilers only generate SSE instructions when it can vectorize the DO loops
  - Libraries must be Quad core (or higher) enabled
- Operands must be aligned on 128 bit boundaries
  - Operand alignment can be performed; however, it distracts from the performance.

When does the compiler vectorize

- What can be vectorized
  - Only loops
  - Stride 1 arrays, indirect addressing is bad
  - No recursion
- Check the compiler output listing and/or assembler listing
  - Look for packed SSE instructions

- **Note of caution** : Don‘t get to excited about vectorization
  The main limitation is often memory bandwidth

Interlagos: AVX (Advanced Vector Extensions)

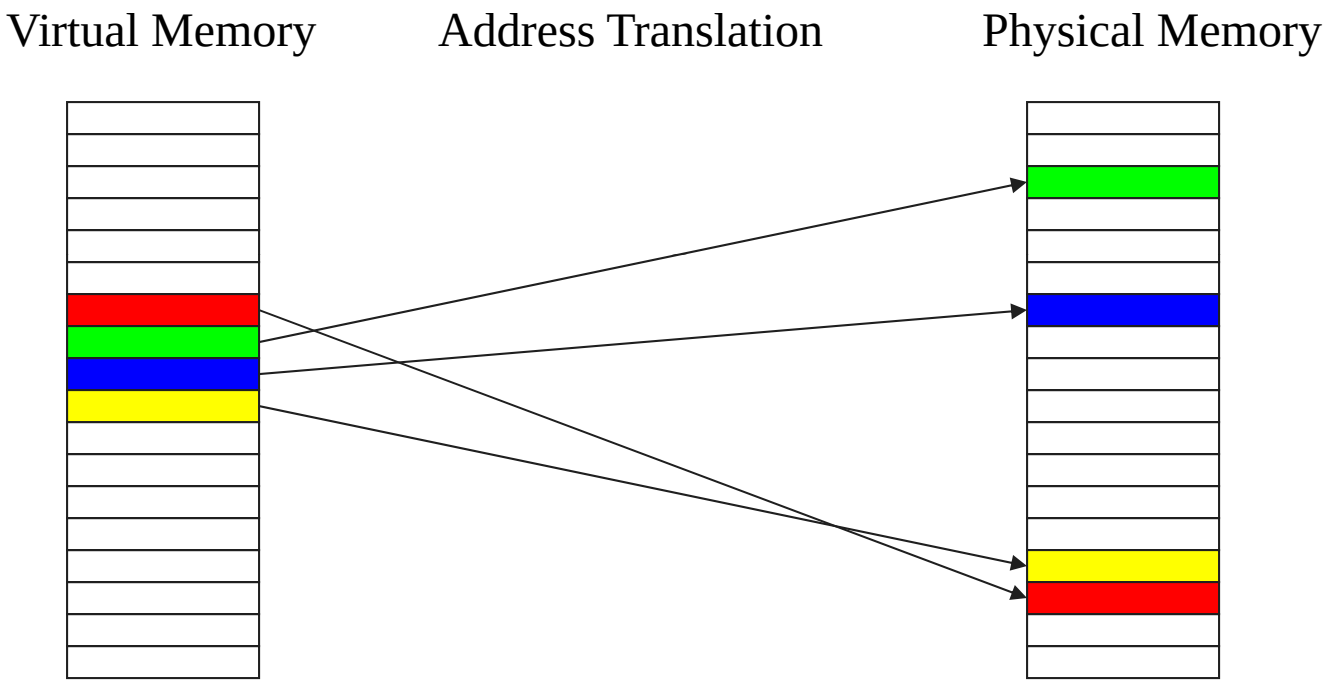
- Max Vector length doubled to 256 bit (Register)
- Much cleaner instruction set
  - Result register is unique from the source registers
  - Old SSE instruction set always destroyed a source register
- Floating point multiple-accumulate (FMA)
  - A(1:4) = B(1:4)*C(1:4) + D(1:4) ! Now one instruction
- Current processors of both AMD and Intel have AVX
- **Vectors are becoming more important, not less**

TLB Utilization

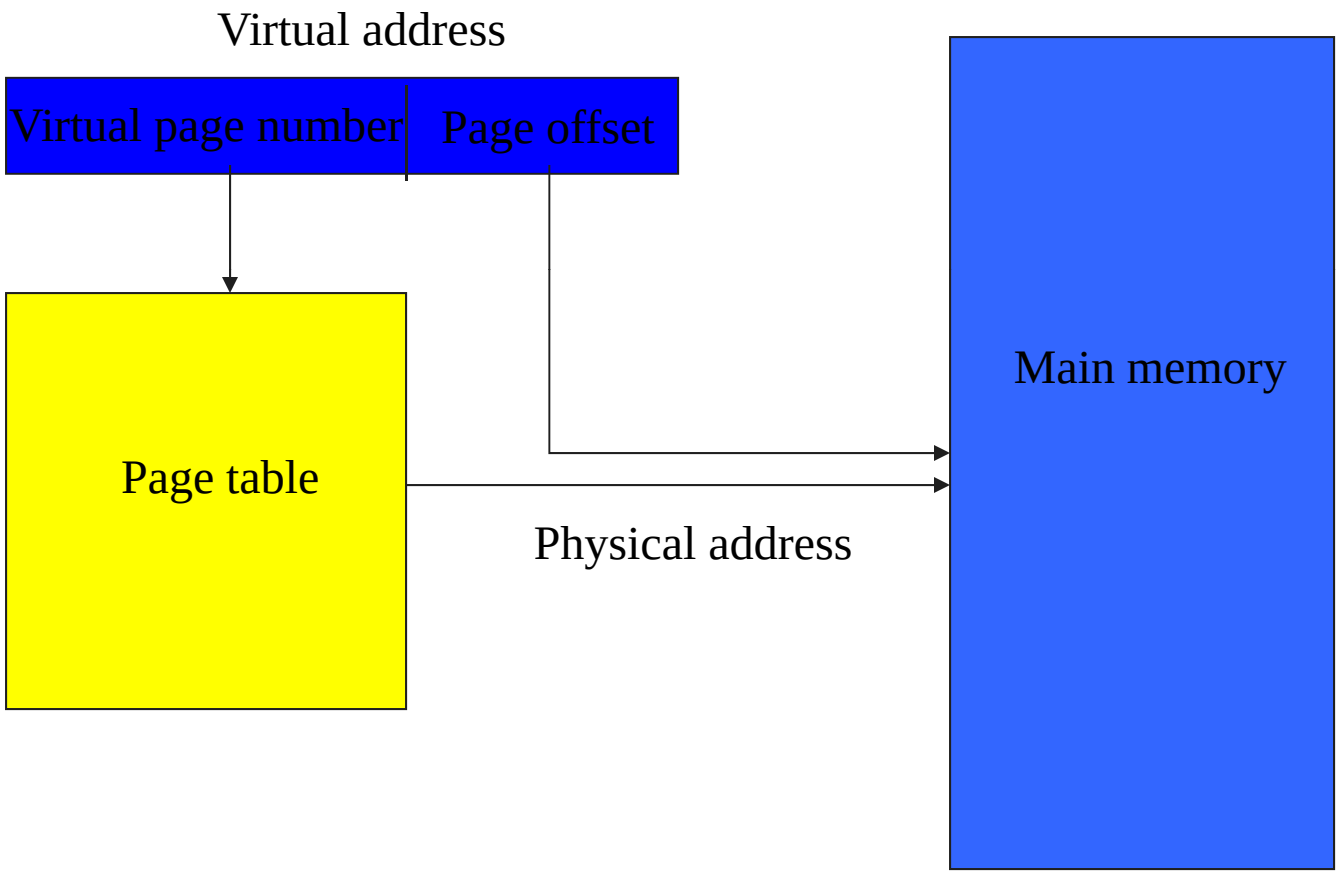
Background: Virtual Memory

- Modern programs operate in “virtual memory”
  - Each program thinks it has all of memory to itself
  - Fixed sized blocks (“pages”) vs variable sized blocks (“segments”)
- Virtual Memory benefits
  - Allow a program that is larger than physical memory to run
    - Programmer does not have to manually create overlays
  - Allow many programs to share limited physical memory
- Virtual Memory problems
  - Each virtual memory reference must be translated into a physical memory reference

Virtual Memory vs Physical Memory



Address Translation



Source: Computer Architecture A Quantitative Approach, by John L. Hennessy and David A. Patterson

Translation Speed

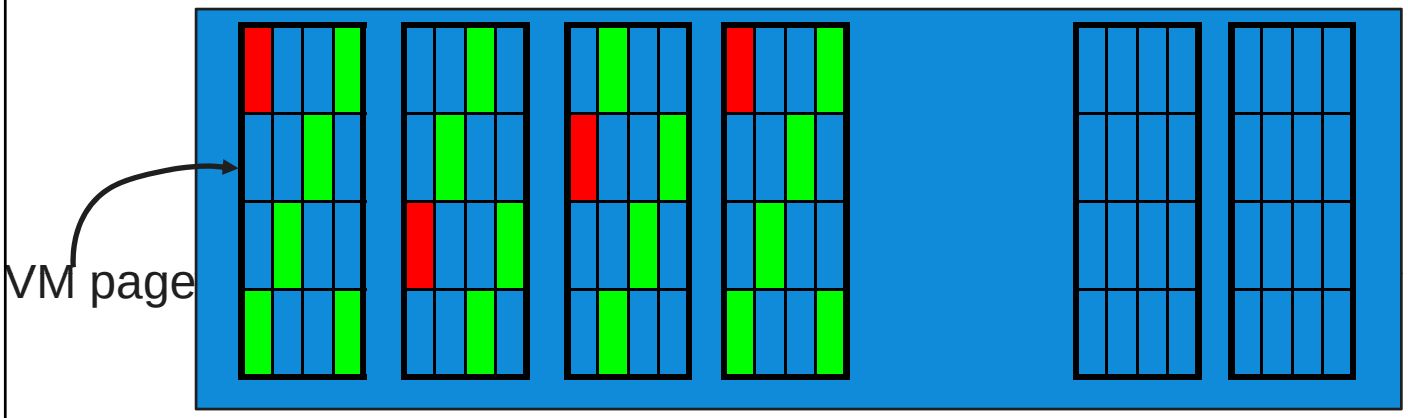
- Translation page table is stored in main memory
  - Each memory access logically takes twice as long – once to find the physical address, once to get the actual data
- Use a hardware cache of least recently used addresses
  - Called a Translation Lookaside Buffer or TLB

The TLB cache

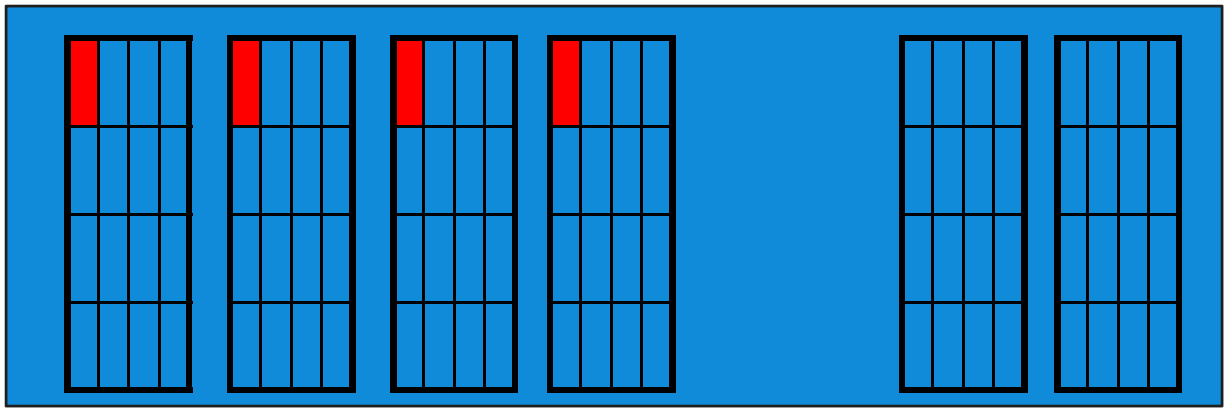
= new TLB entry created

= address already mapped

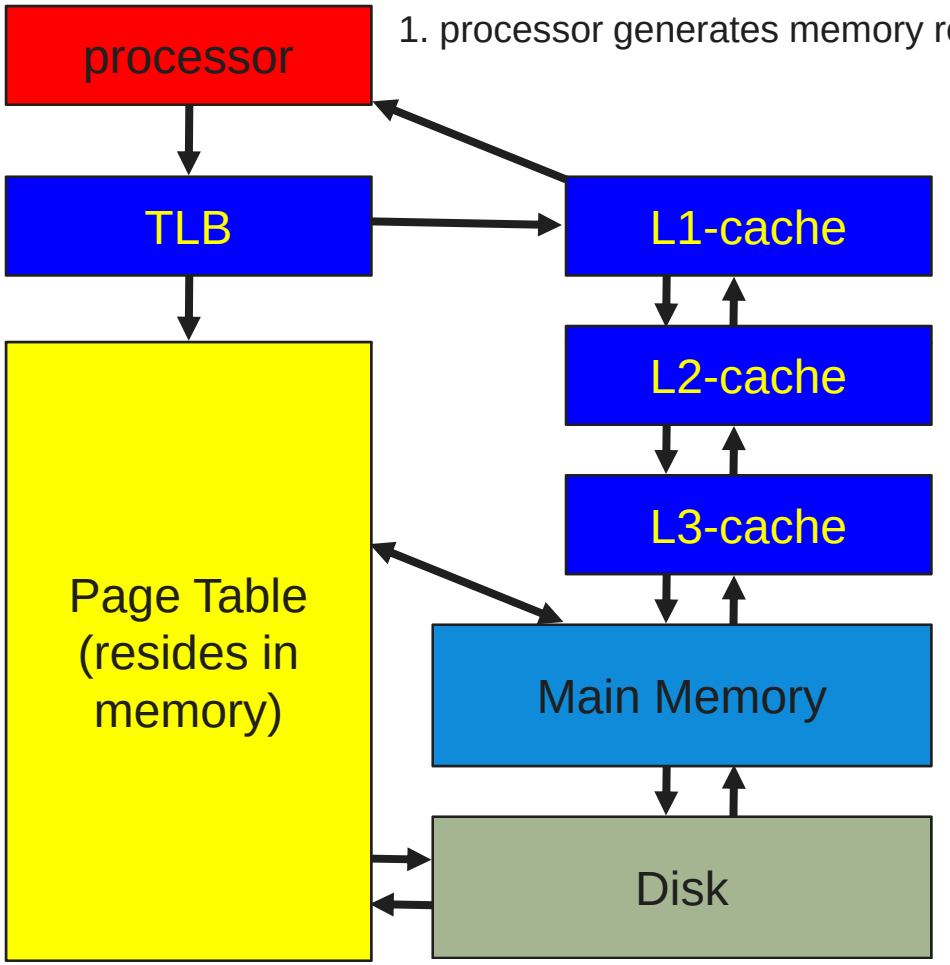
bad for the TLB
non unit stride through the data



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



Putting it all together



1. processor generates memory reference (ld or st)

2. check TLB if virtual mapping is present

3. If virtual mapping is not present, go to page table, retrieve mapping, and update TLB with the mapping

4. L1 cache miss

5. L2 cache miss

6. L3 cache miss

7. main memory hit

8. If there is a main memory miss, then a page fault would be generated.