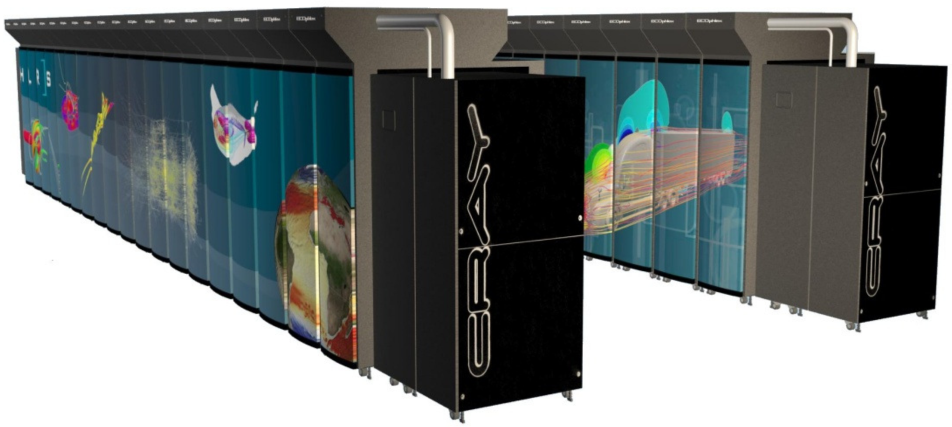
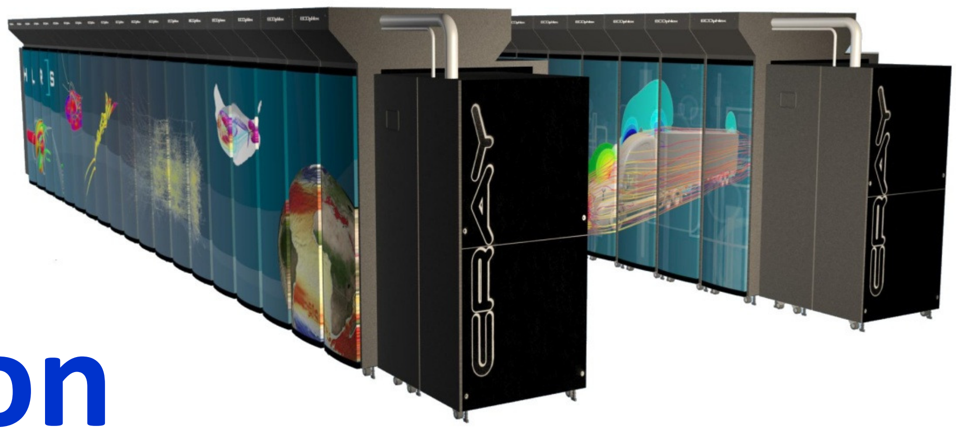


Basic CPU optimization overview

Stefan Andersson
stefan@cray.com

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.

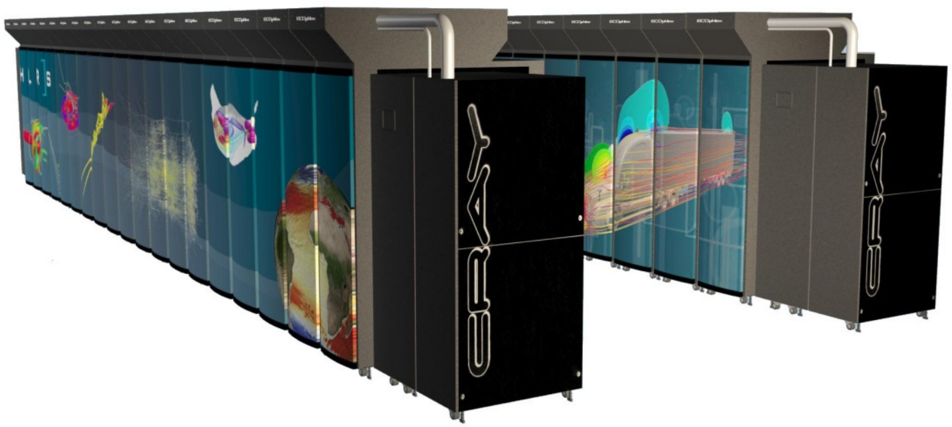
When does the compiler vectorize

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

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

Next Generation : 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
- Next processor generation of both AMD and Intel will have AVX
- **Vectors are becoming more important, not less**



Basic loops optimizations techniques

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

- This optimization is being used less and less frequently

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

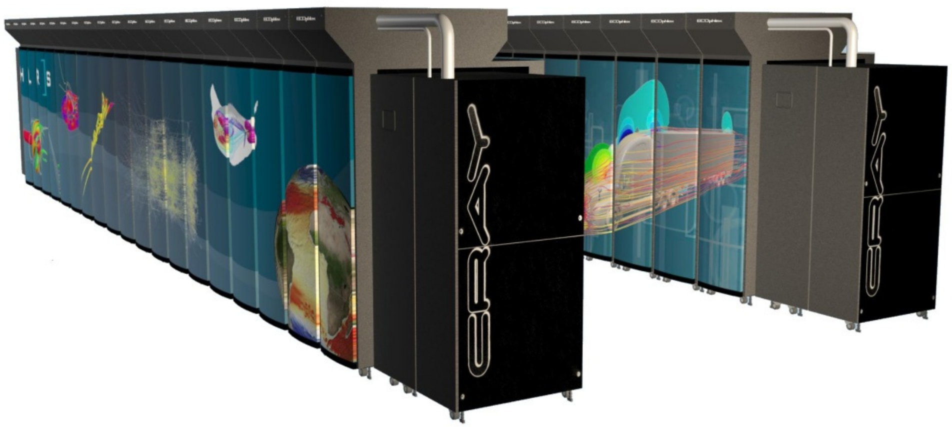
```
DO IOUTER = 1, 10000, 1000
  DO STRIP = IOUTER, IOUTER+999
        A(STRIP) = A(STRIP) * B(STRIP)
  ENDDO
ENDDO
```

CRAY THE SUPERCOMPUTER COMPANY

H L R S

When should I optimize matmul?

- Never
  - Use a vendor provided DGEMM library
  - Write simple triple nested loop or array syntax and let the compiler pattern match it
- Other BLAS level 1 and 2 libraries are also provided and should be considered for use unless those array operations are part of a larger loop nest and using BLAS routine will inhibit compiler analysis and optimization



Cache Optimization

Based on John Leveques presentation

THE SUPERCOMPUTER COMPANY

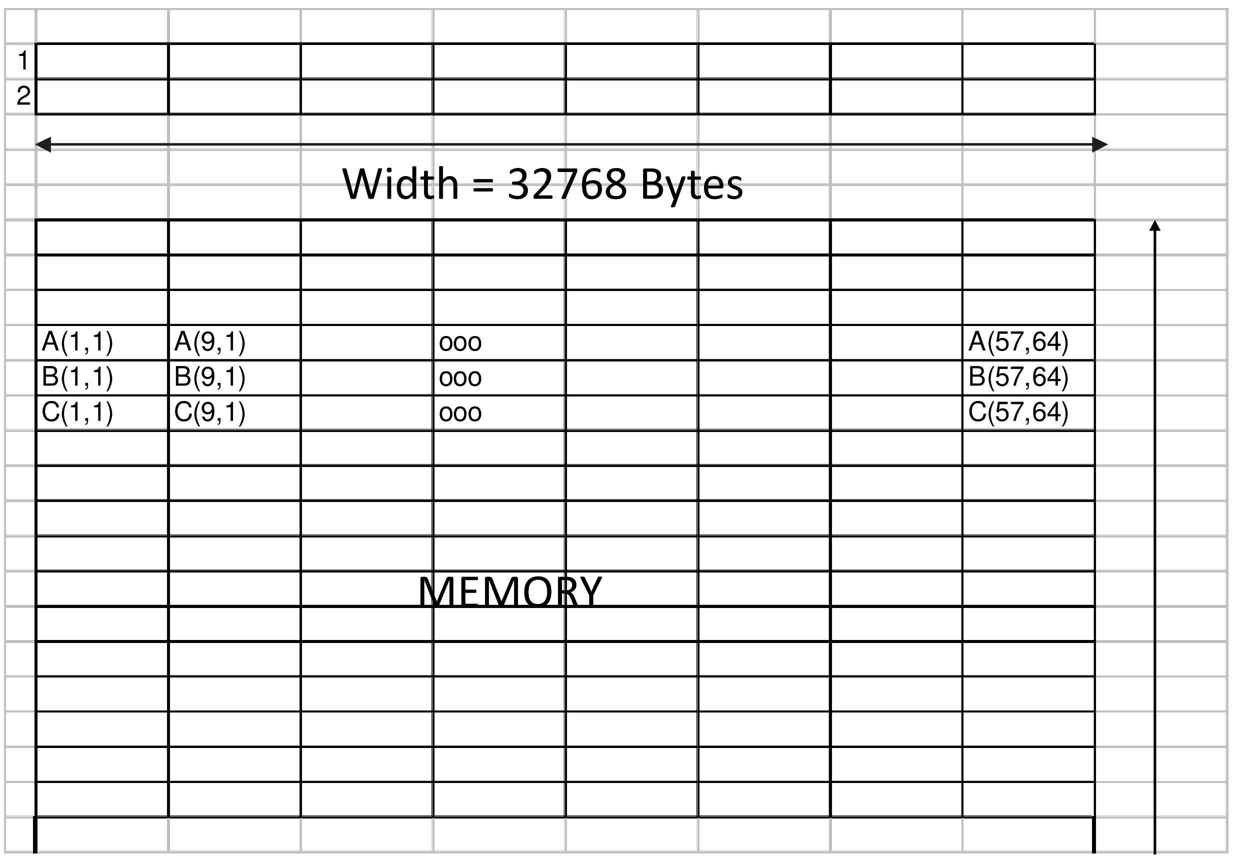
Consider the following example

```
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



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

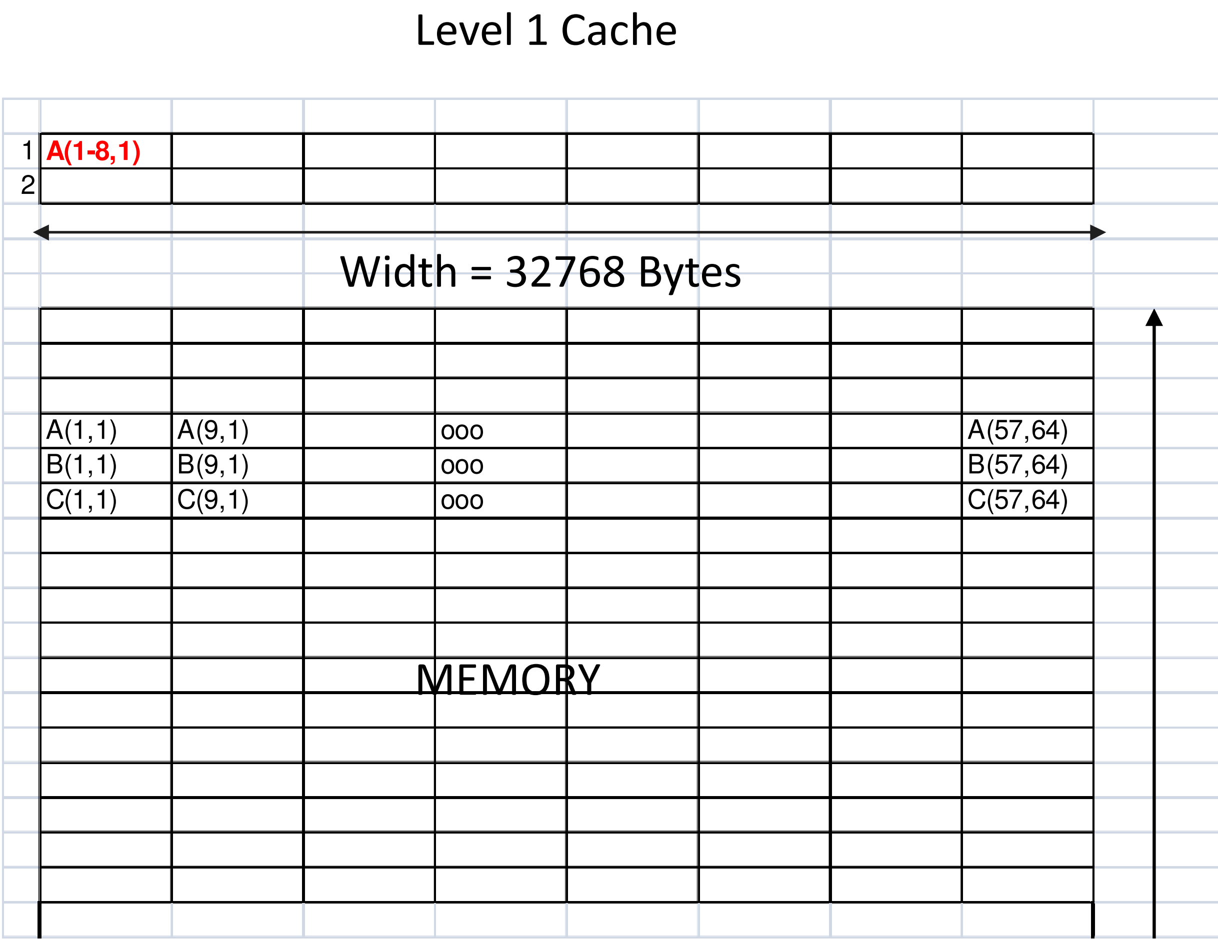
Associativity Class
- 32768 B
- 512 Lines
- 4096 8B Ws
- 8192 4B Ws

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

CRAY THE SUPERCOMPUTER COMPANY

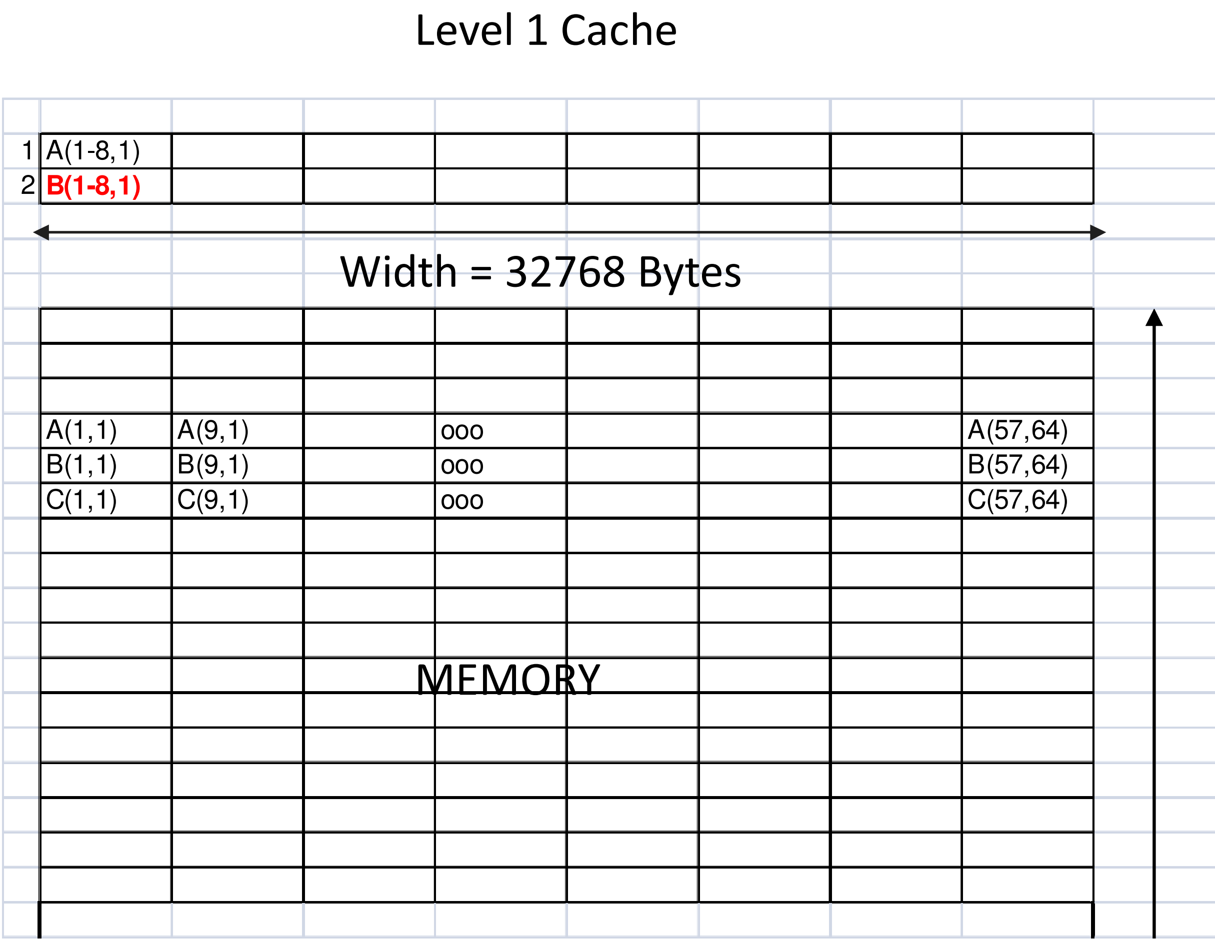
HLRS

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		

THE SUPERCOMPUTER COMPANY

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

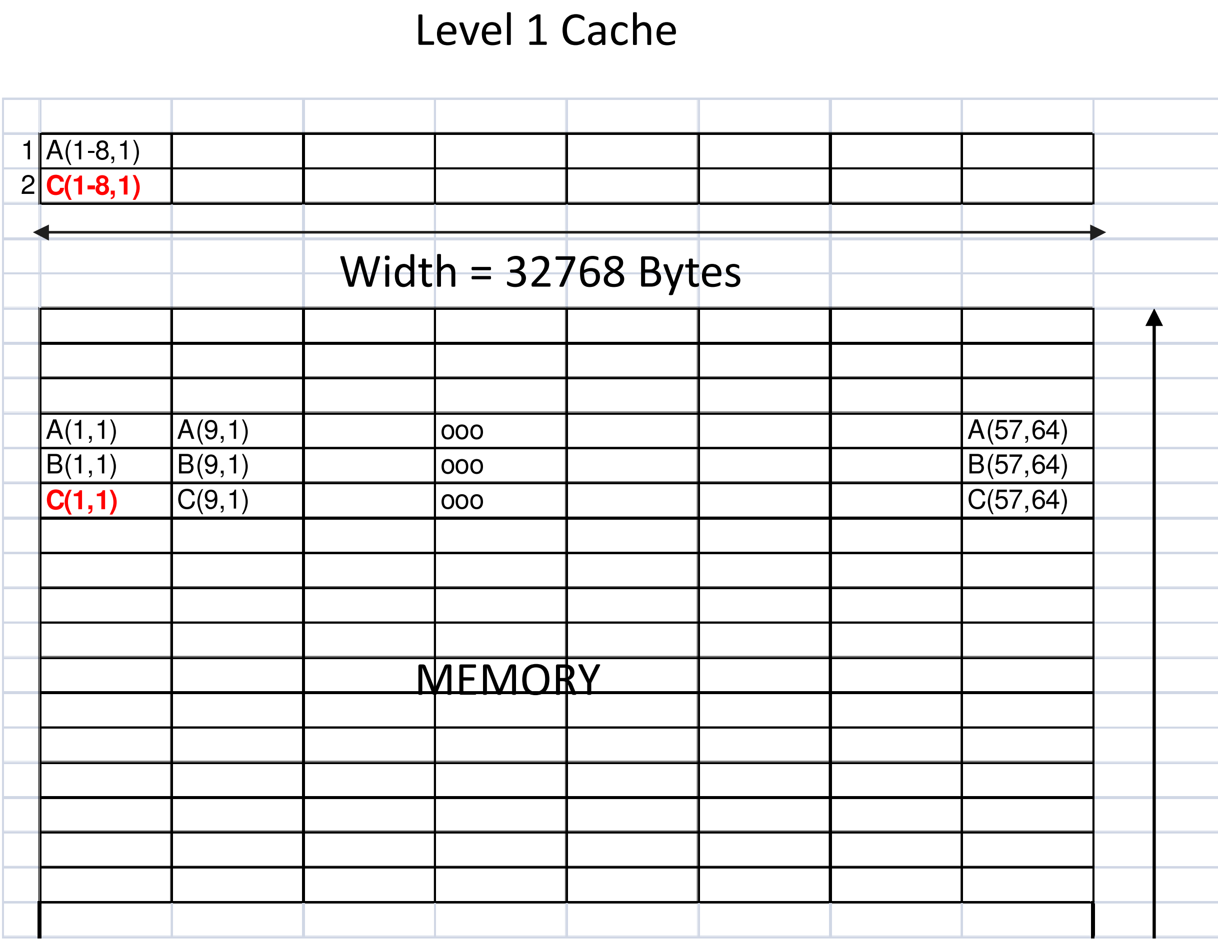
Step 3 : Load C(1,1), needed even if it's not read

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		

CRAY THE SUPERCOMPUTER COMPANY

HLRS

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

CRAY THE SUPERCOMPUTER COMPANY HLRS

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

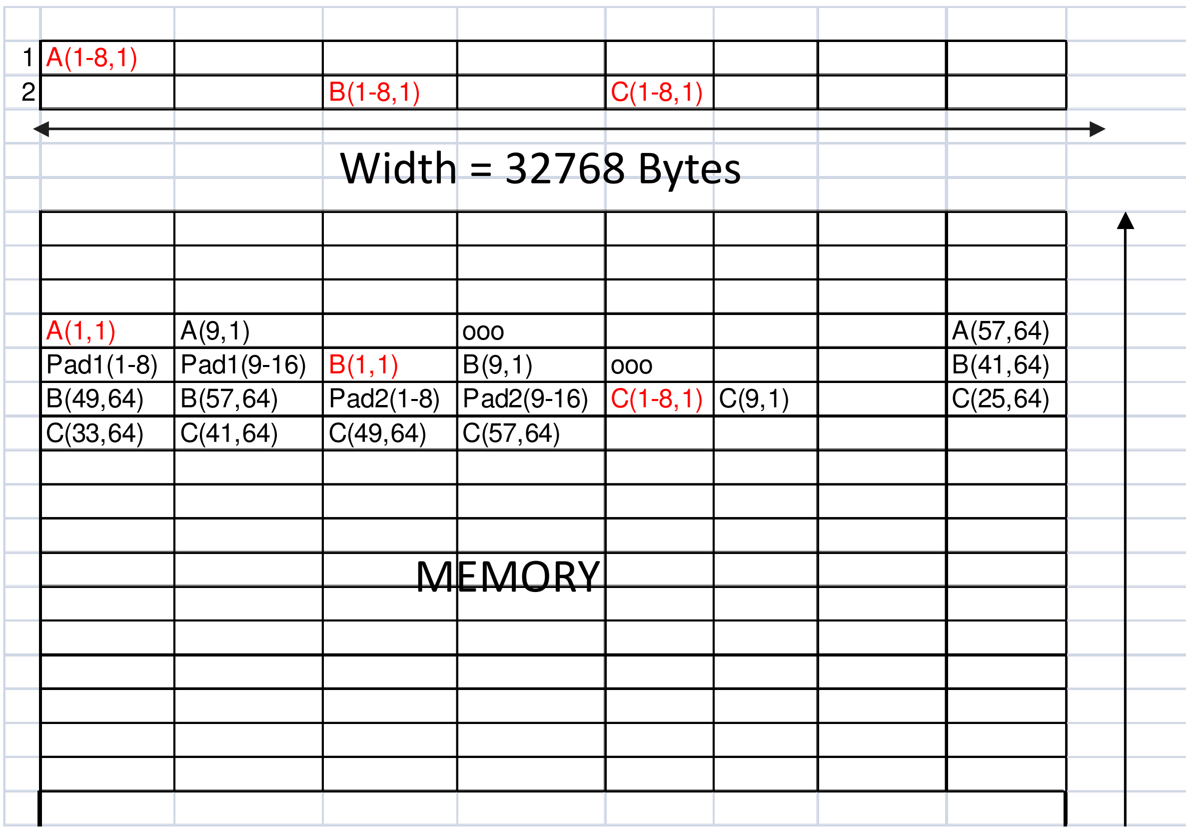
```
Real * 8    A(64,64),pad1(16),B(64,64),pad2(16),C(64,64)


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

THE SUPERCOMPUTER COMPANY

Cache and memory layout with padding

Level 1 Cache



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

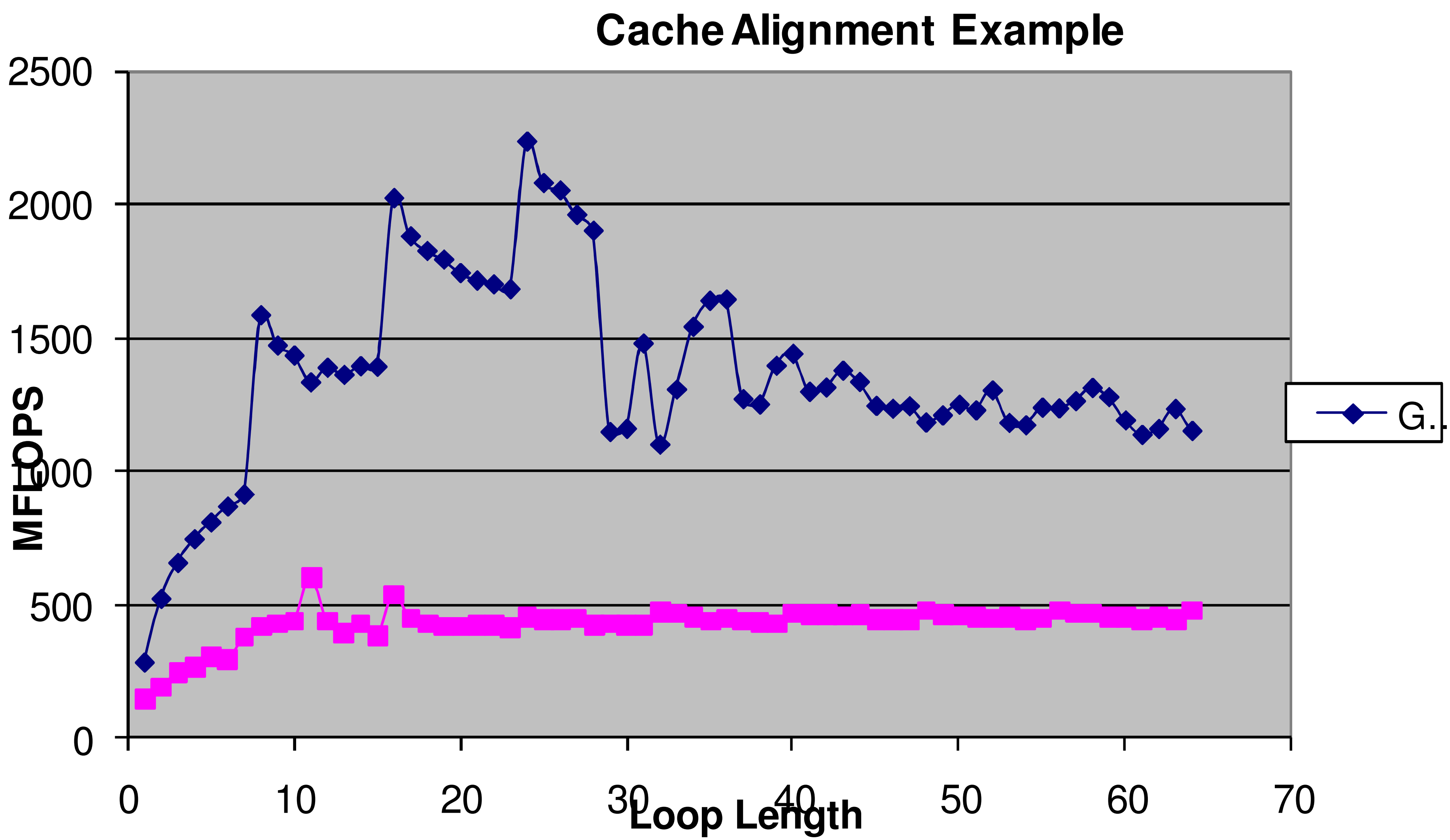
Associativity Class
- 32768 B
- 512 Lines
- 4096 8B Ws
- 8192 4B Ws

64*64*8 = 32768 B

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

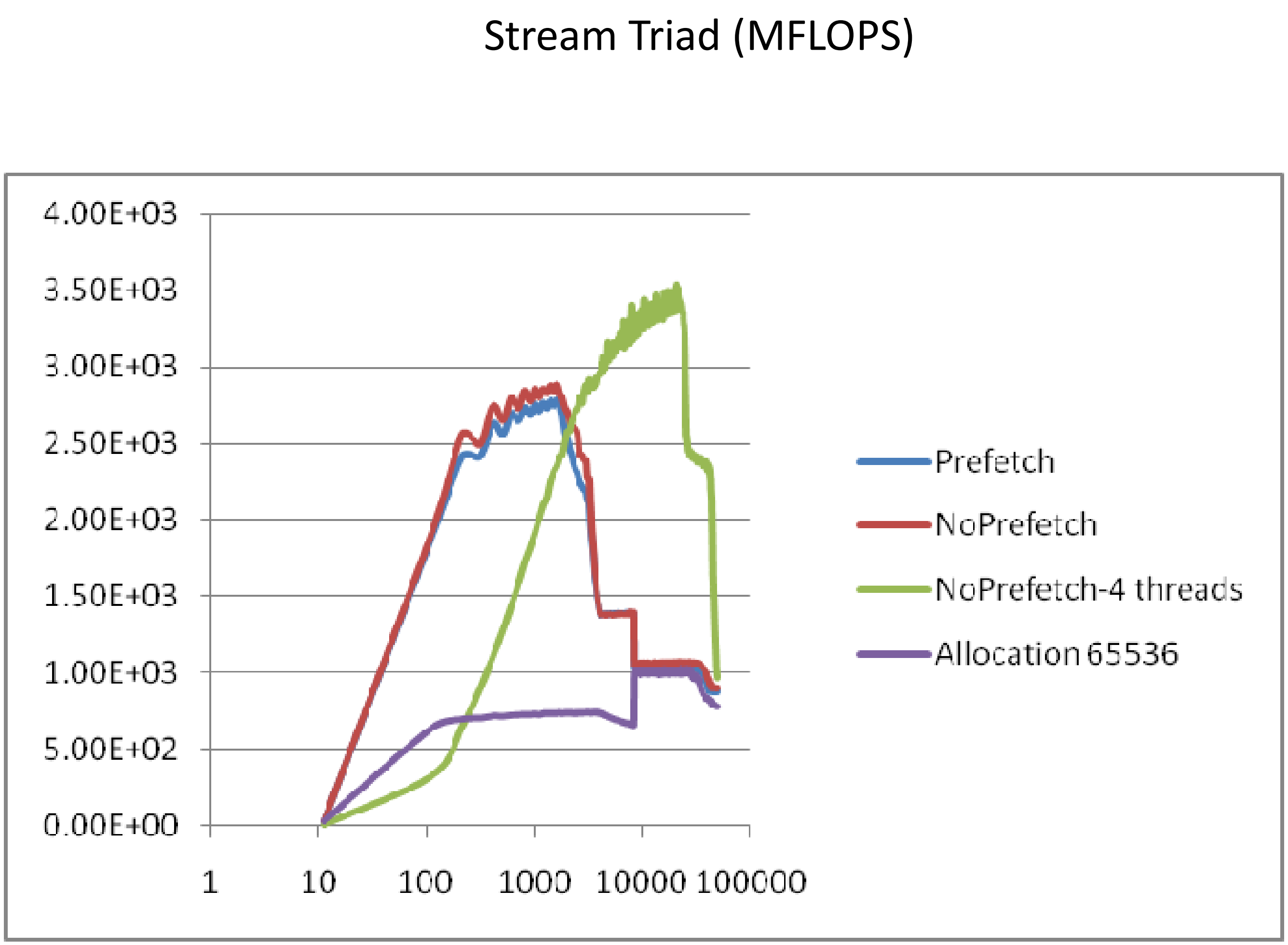
```
Time%                                              0.2%
Time                                           0.000003
Calls                                                 1
PAPI_L1_DCA              455.433M/sec          1367 ops
DC_L2_REFILL_MOESI        49.641M/sec           149 ops
DC_SYS_REFILL_MOESI        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

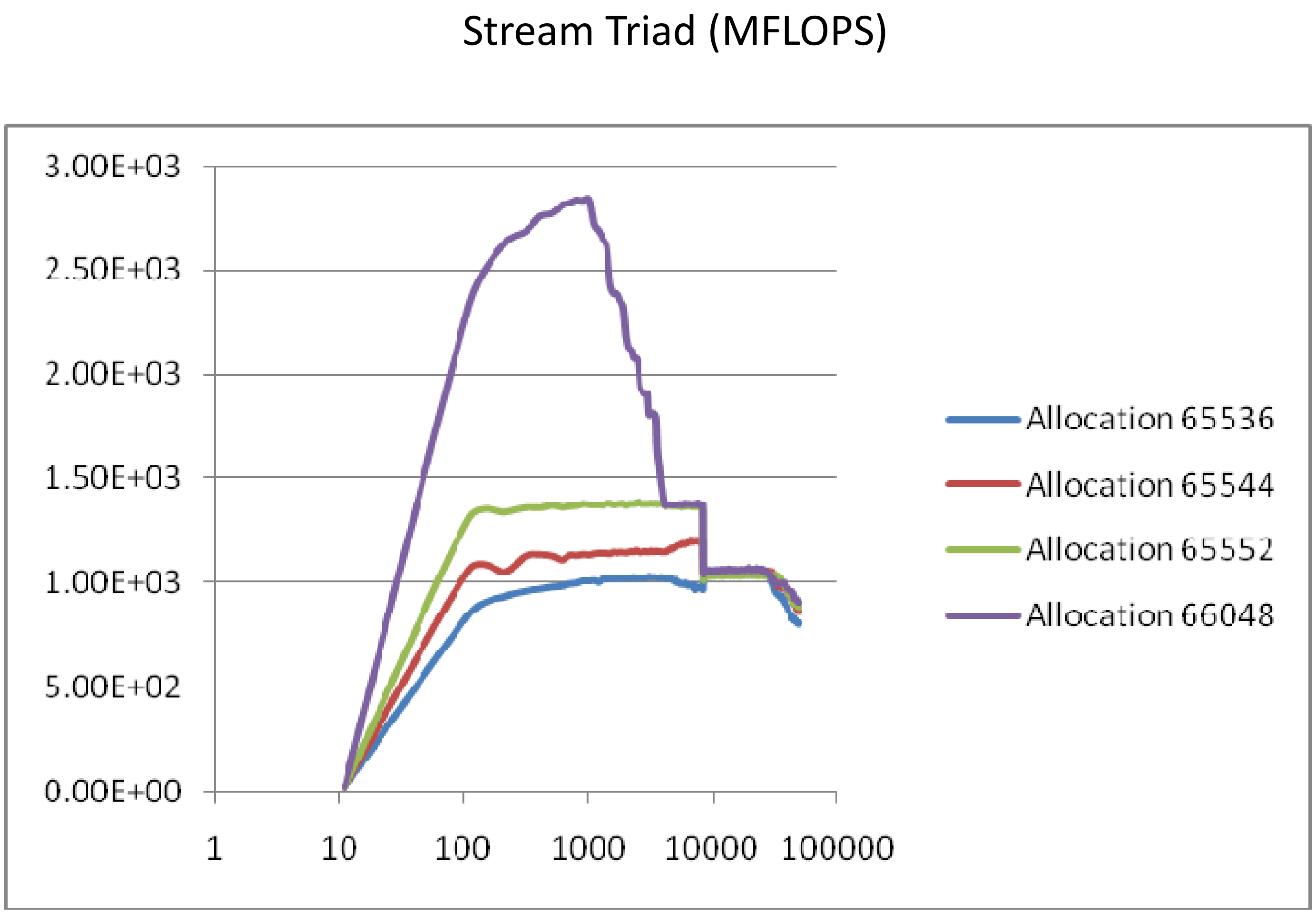
```
Time%                                             0.1%
Time                                          0.000002
Calls                                                1
PAPI_L1_DCA                 689.986M/sec          1333 ops
DC_L2_REFILL_MOESI           33.645M/sec            65 ops
DC_SYS_REFILL_MOESI                                  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
```

THE SUPERCOMPUTER COMPANY

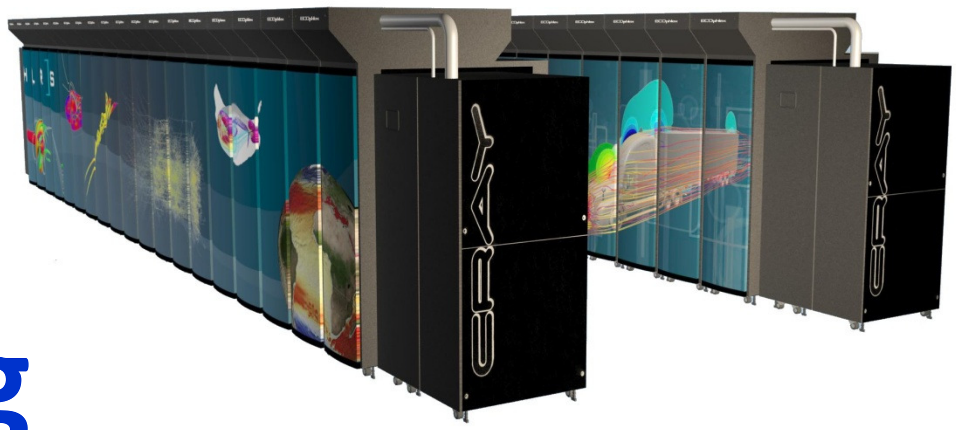
Performance = F(Cache Utilization)



Performance = F(Cache Utilization)



THE SUPERCOMPUTER COMPANY



Cache Blocking from Start to Finish

HLRS Workshop

3. Feb. 2011

Stefan Andersson (Based on Steve Whalen's work)
stefan@cray.com

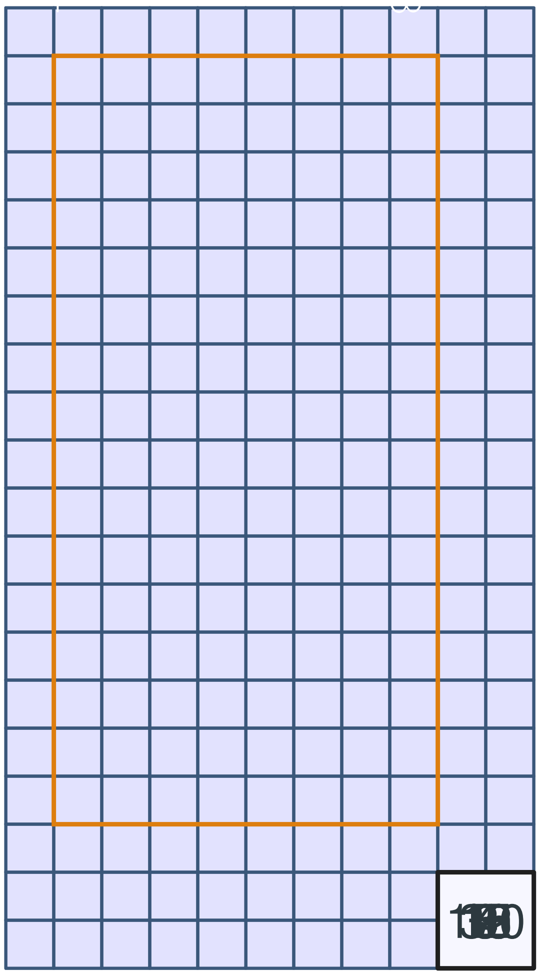
CRAY THE SUPERCOMPUTER COMPANY

H L R I S

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

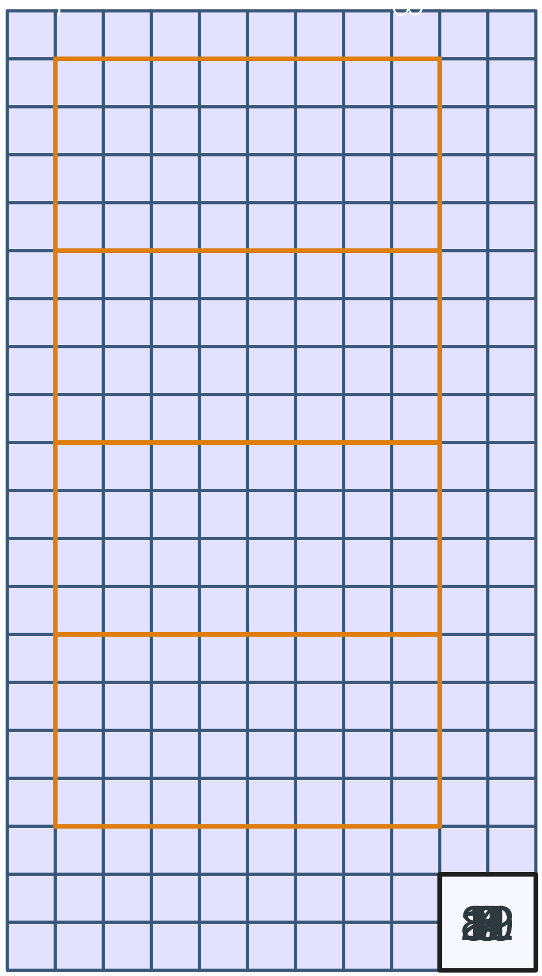


- 2D Laplacian

```
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

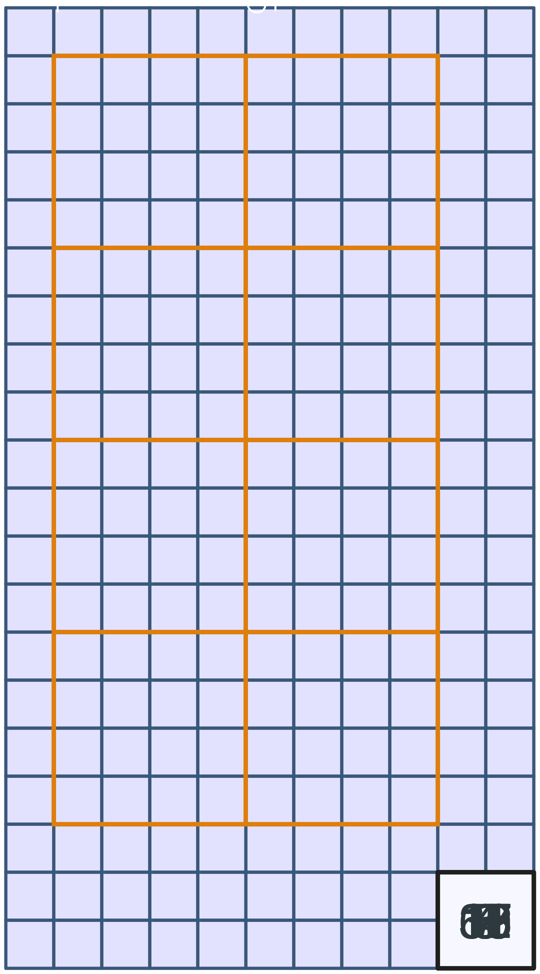


- Unblocked loop: 120 cache misses
- Block the inner loop

```
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?

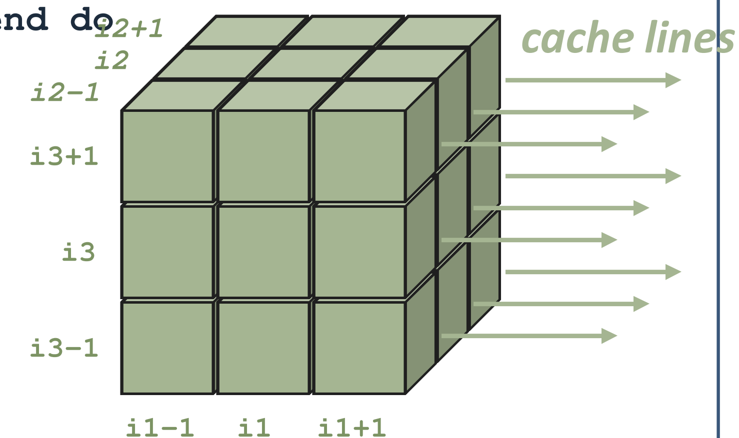
“I tried cache-blocking my code, but it didn’t help”

- 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

Thank You

T h a n k Y o u

Questions ?

CRAY
THE SUPERCOMPUTER COMPANY

HLRS