332
Advanced Computer Architecture

Matrix multiply exercise

October 2022
Paul H J Kelly

These lecture notes are partly based on the course text, Hennessy and Patterson's *Computer Architecture, a quantitative approach (4-6th ed), and on the lecture slides of David Patterson's Berkeley course (CS252)*

Matrix-matrix multiply

```
phjk@ray03:MM$ ./MM1.x86
mm1: 39.692964 s, 519.150335 MFLOPS

phjk@ray03:MM$ ./MM2.x86
mm2: 3.557066 s, 5793.149622 MFLOPS

phjk@ray03:MM$ ./MM3.x86
mm3: 2.599583 s, 7926.892718 MFLOPS

phjk@ray03:MM$ ./MM4.x86
mm4: 4.000465 s, 5151.055078 MFLOPS
```

What CPU do we have?

```
ray03(phjk) 155 % head /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 158
model name      : Intel(R) Core(TM) i7-7700K CPU @ 4.20GHz
stepping        : 9
microcode       : 0xb4
cpu MHz         : 987.006
cache size      : 8192 KB
```

>4400
when busy

Intel® Core™ i7-7700K Processor

8M Cache, up to 4.50 GHz

Add to Compare

Essentials

Export specifications

Product Collection	7th Generation Intel® Core™ i7 Processors
Code Name	Products formerly Kaby Lake
Vertical Segment	Desktop
Processor Number	i7-7700K
Status	Launched
Launch Date	Q1'17
Lithography	14 nm
Included Items	Please note: The boxed product does not include a fan or heat sink
Recommended Customer Price	$339.00 - $350.00

Performance

# of Cores	4
# of Threads	8
Processor Base Frequency	4.20 GHz
Max Turbo Frequency	4.50 GHz
Cache	8 MB SmartCache
Bus Speed	8 GT/s DMI3
# of QPI Links	0
TDP	91 W

https://ark.intel.com/content/www/us/en/ark/products/97129/intel-core-i7-7700k-processor-8m-cache-up-to-4-50-ghz.html

```c
/*
 * mm
 *
 * Multiply A by B leaving the result in C.
 * A is assumed to be an l x m matrix, B an m x n matrix.
 * The result matrix C is of course l x n.
 * The result matrix is assumed to be initialised to zero.
 *
 * Dumbest possible
 */
void mm1(A,B,C)
     FLOATTYPE A[SZ][SZ], B[SZ][SZ], C[SZ][SZ];
{
  int i, j, k;

  for (i = 0; i < SZ; i++){
    for (j = 0; j < SZ; j++){
      for (k = 0; k < SZ; k++){
        C[i][j] += A[i][k] * B[k][j];
      }
    }
  }
}
```

-UU-:----F1 MM1.c 43% L60 (C/*l Abbrev) ------

```c
/*
 * mm
 *
 * Multiply A by B leaving the result in C.
 * A is assumed to be an l x m matrix, B an m x n matrix.
 * The result matrix C is of course l x n.
 * The result matrix is assumed to be initialised to zero.
 *
 * Less dumb.
 */
void mm2(A,B,C)
     FLOATTYPE A[SZ][SZ], B[SZ][SZ], C[SZ][SZ];
{
  int i, j, k;

  for (i = 0; i < SZ; i++){
    for (k = 0; k < SZ; k++){
      for (j = 0; j < SZ; j++){
        C[i][j] += A[i][k] * B[k][j];
      }
    }
  }
}
```

-UU-:----F1 MM2.c 43% L71 (C/*l Abbrev) -------

```
ray03(phjk) 159 % diff MM1.c MM2.c
…
75,76c75,76
<       for (j = 0; j < SZ; j++){
<         for (k = 0; k < SZ; k++){
---
>       for (k = 0; k < SZ; k++){
>         for (j = 0; j < SZ; j++){
```

```
ray03(phjk) 159 % diff MM1.c MM2.c
…
75,76c75,76
<     for (j = 0; j < SZ; j++){
<       for (k = 0; k < SZ; k++){
---
>     for (k = 0; k < SZ; k++){
>       for (j = 0; j < SZ; j++){
```

```c
/*
 * mm
 *
 * Multiply A by B leaving the result in C.
 * A is assumed to be an l x m matrix, B an m x n matrix.
 * The result matrix C is of course l x n.
 * The result matrix is assumed to be initialised to zero.
 *
 * Less dumb.
 */
void mm2(A,B,C)
     FLOATTYPE A[SZ][SZ], B[SZ][SZ], C[SZ][SZ];
{
  int i, j, k;

  for (i = 0; i < SZ; i++){
    for (k = 0; k < SZ; k++){
      for (j = 0; j < SZ; j++){
        C[i][j] += A[i][k] * B[k][j];
      }
    }
  }
}
```

-UU-:----F1 MM2.c 43% L71 (C/*l Abbrev) ------

```c
/* The blocked version from Lam, Rothberg and Wolf
 */
void mm3(A,B,C,blocksize)
     FLOATTYPE A[SZ][SZ], B[SZ][SZ], C[SZ][SZ];
     int blocksize;
{
  int i, j, k, kk, jj;
  FLOATTYPE r;

  for (kk = 0; kk < SZ; kk += blocksize){
    for (jj = 0; jj < SZ; jj += blocksize){
      for (i = 0; i < SZ; i++){
        for (k = kk; k < min(kk+blocksize,SZ); k++){
          r = A[i][k];
          for (j = jj; j < min(jj+blocksize, SZ); j++){
            C[i][j] += r * B[k][j];
          }
        }
      }
    }
  }
}
```

-UU-:----F1 MM3.c 47% L79 (C/*l Abbrev) --

```
ray03(phjk) 159 % diff MM1.c MM3.c
…
<   for (i = 0; i < SZ; i++){
<     for (j = 0; j < SZ; j++){
<       for (k = 0; k < SZ; k++){
<       C[i][j] += A[i][k] * B[k][j];
---
>   for (kk = 0; kk < SZ; kk += blocksize){
>     for (jj = 0; jj < SZ; jj += blocksize){
>       for (i = 0; i < SZ; i++){
>       for (k = kk; k < min(kk+blocksize,SZ); k++){
>         r = A[i][k];
>         for (j = jj; j < min(jj+blocksize, SZ); j++){
>           C[i][j] += r * B[k][j];
>         }
>       }
```

```
ray03(phjk) 159 % diff MM1.c MM3.c
…
<   for (i = 0; i < SZ; i++){
<     for (j = 0; j < SZ; j++){
<       for (k = 0; k < SZ; k++){
<       C[i][j] += A[i][k] * B[k][j];
---
>   for (kk = 0; kk < SZ; kk += blocksize){
>     for (jj = 0; jj < SZ; jj += blocksize){
>       for (i = 0; i < SZ; i++){
>       for (k = kk; k < min(kk+blocksize,SZ); k++){
>         r = A[i][k];
>         for (j = jj; j < min(jj+blocksize, SZ); j++){
>           C[i][j] += r * B[k][j];
>         }
>       }
```

Consider matrix-matrix multiply (tutorial ex)

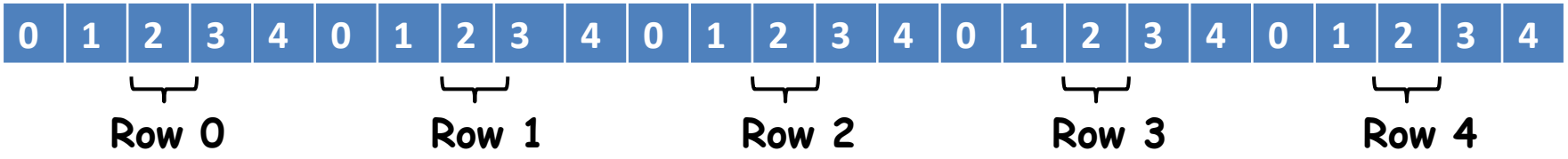
- MM1:

```
for (i=0;i<N;i++)
 for (j=0;j<N;j++)
   for (k=0;k<N;k++)
     C[i][j] += A[i][k] * B[k][j];
```

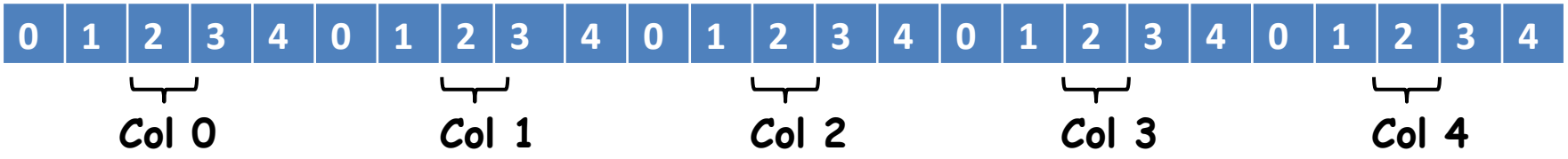
- **MM2:**

```
for (i=0;i<N;i++)
 for (k=0;k<N;k++)
   for (j=0;j<N;j++)
     C[i][j] += A[i][k] * B[k][j];
```

- **Row-major storage layout (default for C):**



- **Column-major storage layout (default for Fortran):**



Consider matrix-matrix multiply (tutorial ex)

- MM1:

```
for (i=0;i<N;i++)
 for (j=0;j<N;j++)
   for (k=0;k<N;k++)
     C[i][j] += A[i][k] * B[k][j];
```

- **MM2:**

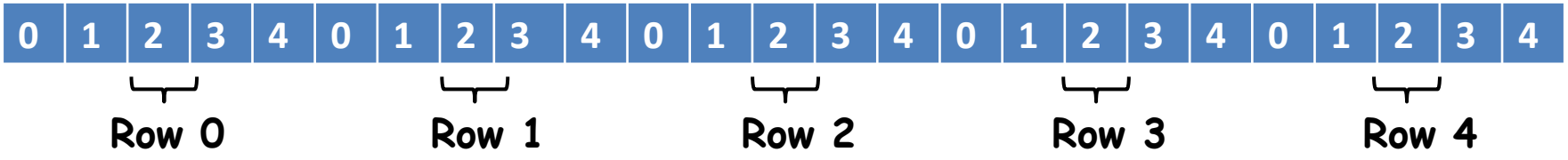
```
for (i=0;i<N;i++)
 for (k=0;k<N;k++)
   for (j=0;j<N;j++)
     C[i][j] += A[i][k] * B[k][j];
```

loop invariant

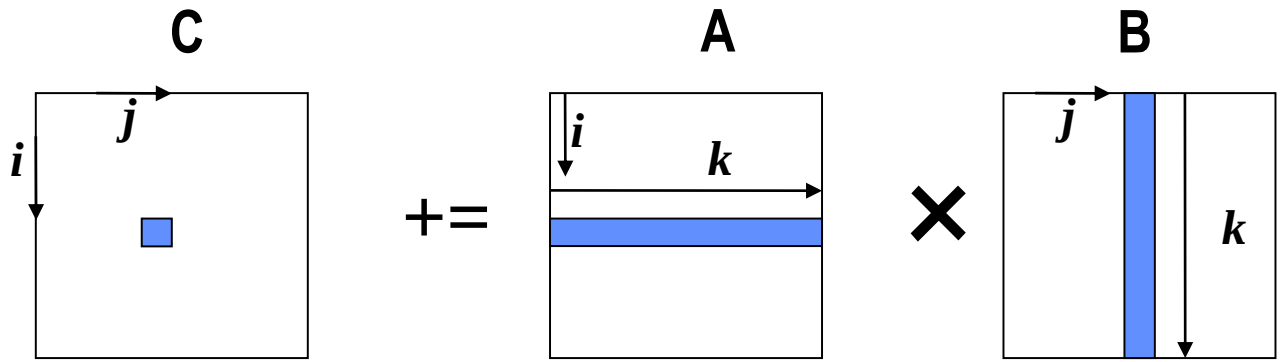
innermost loop

rightmost subscript

- **Row-major storage layout (default for C):**

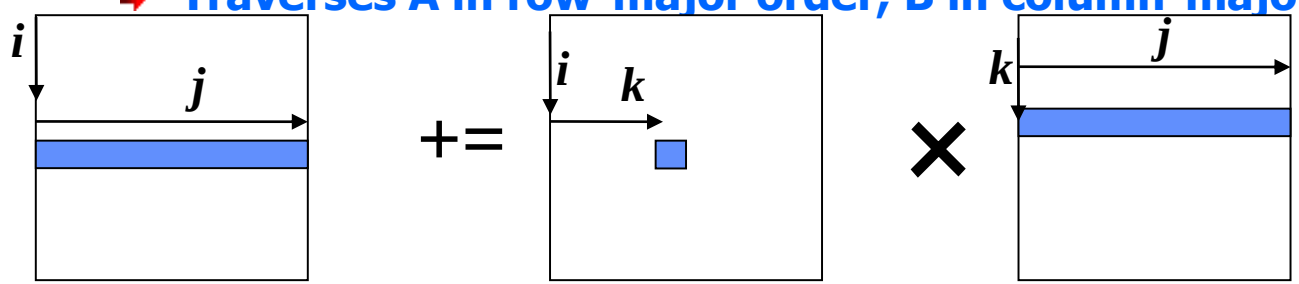


What was going on?



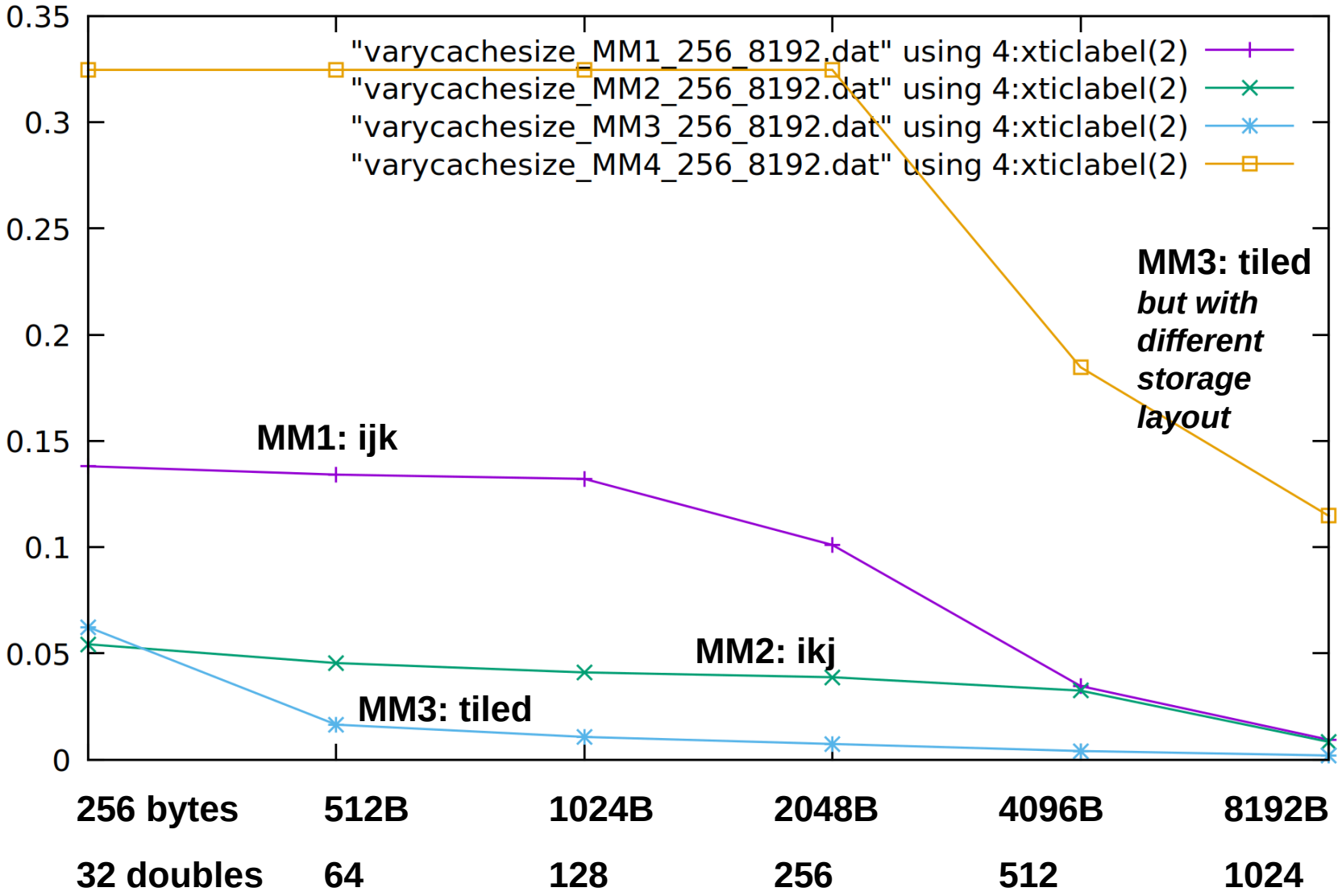

```
for (i=0; i<N; i++)
 for (j=0; j<N; j++)
  for (k=0; k<N; k++)
   C[i][j]+=A[i][k]*B[k][j]
```

- **IJK variant computes each element of result matrix C one at a time, as inner product of row of A and column of B**
  - **Traverses A in row-major order, B in column-major**

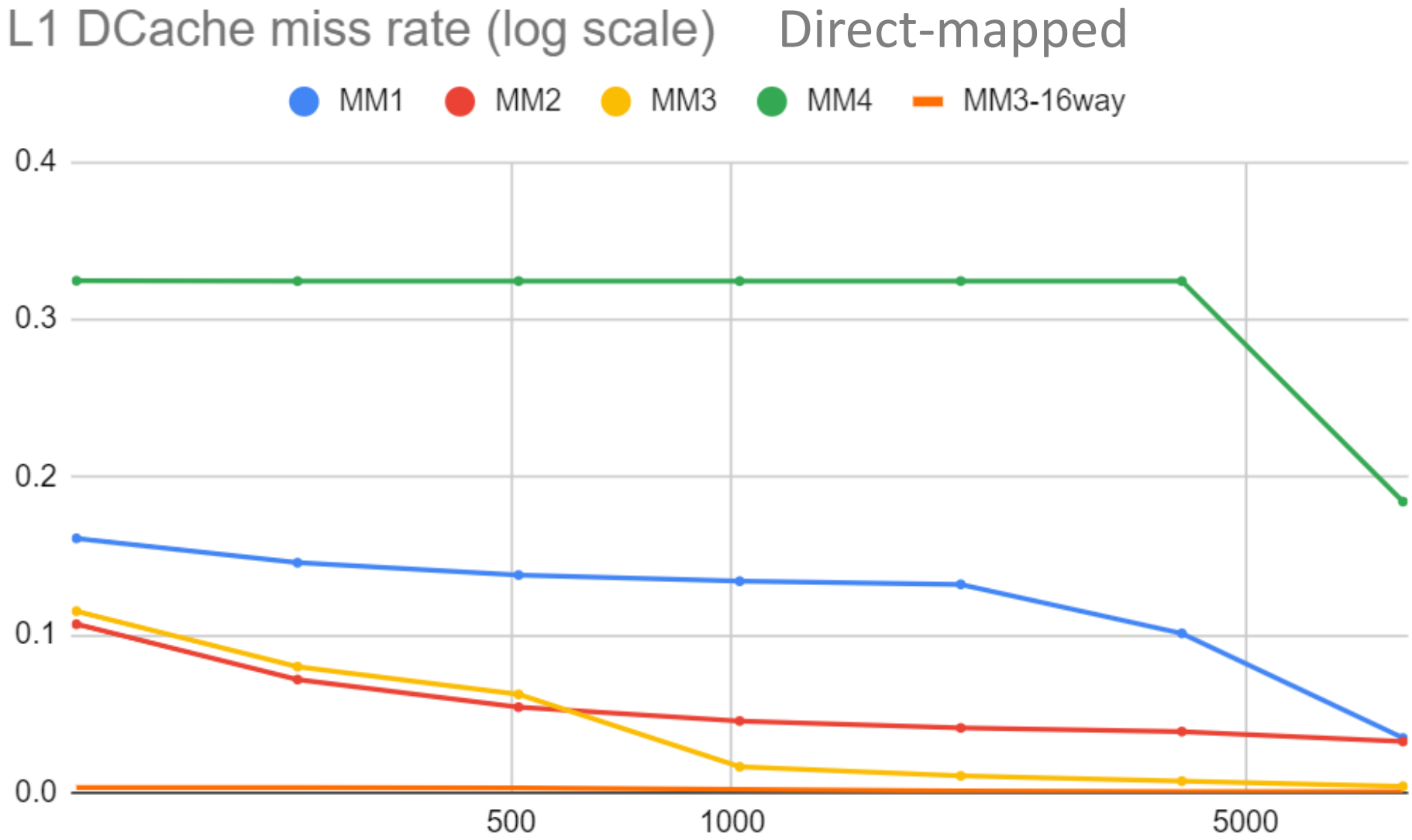



```
for (i=0; i<N; i++)
 for (k=0; k<N; k++)
  for (j=0; j<N; j++)
   C[i][j]+=A[i][k]*B[k][j]
```

- **IKJ variant accumulates partial inner product into a row of result matrix C, using element of A and row of B**
  - **Traverses C and B in row-major order**



Problem size: 192 doubles, 1536 bytes per row



Problem size: 192 doubles, 1536 bytes per row

MM3 : Blocking (a.k.a. “tiling")

- **Idea: reorder execution of loop nest so data isn't evicted from cache before it's needed again.**
- **Blocking is a combination of two transformations: “strip mining", followed by interchange; we start with**

```
for (i = 0; i < N; i++)
  for (k = 0; k < N; k++){
    r = A[i][k];
    for (j = 0; j < N; j++)
      C[i][j] += r * B[k][j]; }
```

- **Strip mine the k and j loops:**

```
for (i = 0; i < N; i++)
  for (kk = 0; kk < N; kk += S)
    for (k = kk; k < min(kk+S,N); k++){
      r = A[i][k];
      for (jj = 0; jj < N; jj += S)
        for (j = jj; j < min(jj+S, N); j++)
          C[i][j] += r * B[k][j];
}
```

Blocking/tiling – stripmine then interchange

- **Now interchange so blocked loops are outermost:**

```
for (kk = 0; kk < N; kk += S)
  for (jj = 0; jj < N; jj += S)
    for (i = 0; i < N; i++)
      for (k = kk; k < min(kk+S,N); k++){
        r = A[i][k];
        for (j = jj; j < min(jj+S, N); j++)
         C[i][j] += r * B[k][j];
}
```

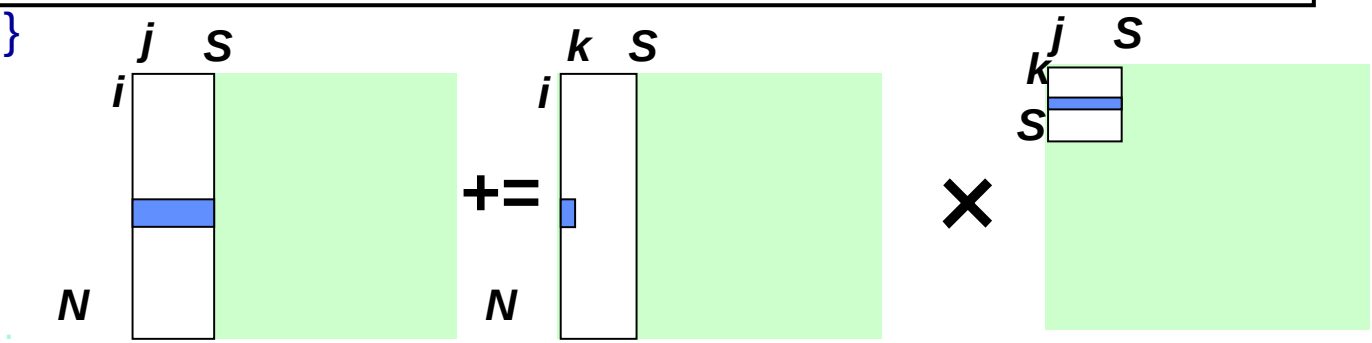
- **The inner i,k,j loops perform a multiplication of a pair of partial matrices.**
- **S is chosen so that a S x S submatrix of B and a row of length S of C can fit in the cache.**
- **What is the right value for S?**

Blocking/tiling – stripmine then interchange

- **Now interchange so blocked loops are outermost:**

```
for (kk = 0; kk < N; kk += S)
  for (jj = 0; jj < N; jj += S)
    for (i = 0; i < N; i++)
      for (k = kk; k < min(kk+S,N); k++)
        r = A[i][k];
        for (j = jj; j < min(jj+S, N); j++)
         C[i][j] += r * B[k][j];
}
```

```
for (kk = 0; kk < N; kk += S)
  for (jj = 0; jj < N; jj += S)
    for (i = 0; i < N; i++)
      // do a block of inner products
      C[i][kk:kk+S] +=
         A[i][kk:kk+S]*B[kk:kk+S][jj:jj+S]
```



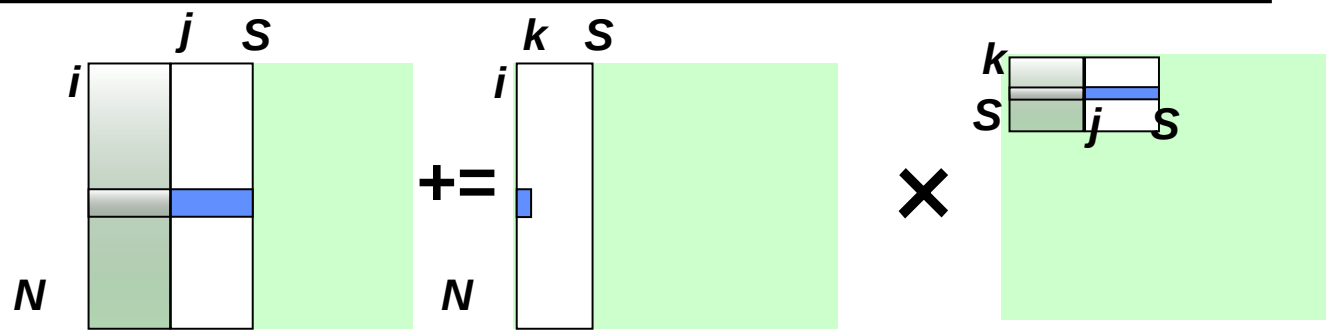
- **Load a chunk of B into cache; use it to compute the partial inner-products for a column of row-segments of C**
- **Each sweep over row-segment of C multiplies element of A by row segment of submatrix of B. Sweep again for each A[i][k]**

Blocking/tiling – stripmine then interchange

- **Now interchange so blocked loops are outermost:**

```
for (kk = 0; kk < N; kk += S)
  for (jj = 0; jj < N; jj += S)
    for (i = 0; i < N; i++)
      for (k = kk; k < min(kk+S,N); k++)
        r = A[i][k];
        for (j = jj; j < min(jj+S, N); j++)
         C[i][j] += r * B[k][j];
}
```

```
for (kk = 0; kk < N; kk += S)
  for (jj = 0; jj < N; jj += S)
    for (i = 0; i < N; i++)
      // do a block of inner products
      C[i][kk:kk+S] +=
        A[i][kk:kk+S]*B[kk:kk+S][jj:jj+S]
```



- **We compute a block of row segments of C at a time**
- **Repeatedly re-using a SxS sub-block of B**
- **Then we move on to the next block of row segments of C**
- No need to revisit the first submatrix of B

What goes wrong with MM4?

```
ray03(phjk) 159 % diff MM3.c MM4.c
…
40c47
< FLOATTYPE A[SZ][SZ];
---
> FLOATTYPE A[ARRAYSZ][ARRAYSZ];
42c49
< FLOATTYPE B[SZ][SZ];
---
> FLOATTYPE B[ARRAYSZ][ARRAYSZ];
44c51
< FLOATTYPE C[SZ][SZ];
---
> FLOATTYPE C[ARRAYSZ][ARRAYSZ];
…
```

```c
/* The blocked version from Lam, Rothberg and Wolf
 */
void mm3(A,B,C,blocksize)
     FLOATTYPE A[SZ][SZ],
               B[SZ][SZ],
               C[SZ][SZ];
     int blocksize;
{
  int i, j, k, kk, jj;
  FLOATTYPE r;

  for (kk = 0; kk < SZ; kk += blocksize){
    for (jj = 0; jj < SZ; jj += blocksize){
      for (i = 0; i < SZ; i++){
        for (k = kk; k < min(kk+blocksize,SZ); k++){
          r = A[i][k];
          for (j = jj; j < min(jj+blocksize, SZ); j++){
            C[i][j] += r * B[k][j];
          }
        }
      }
    }
  }
```

-UU-:----F1 MM3.c 47% L73 (C/*l Abbrev) -----

```c
/* The blocked version from Lam, Rothberg and Wolf
 */
void mm3(A,B,C,blocksize)
     FLOATTYPE A[ARRAYSZ][ARRAYSZ],
               B[ARRAYSZ][ARRAYSZ],
               C[ARRAYSZ][ARRAYSZ];
     int blocksize;
{
  int i, j, k, kk, jj;
  FLOATTYPE r;

  for (kk = 0; kk < SZ; kk += blocksize){
    for (jj = 0; jj < SZ; jj += blocksize){
      for (i = 0; i < SZ; i++){
        for (k = kk; k < min(kk+blocksize,SZ); k++){
          r = A[i][k];
          for (j = jj; j < min(jj+blocksize, SZ); j++){
            C[i][j] += r * B[k][j];
          }
        }
      }
    }
  }
```

-UU-:----F1 MM4.c 49% L78 (C/*l Abbrev) ---

```c
FLOATTYPE A[SZ][SZ];
FLOATTYPE B[SZ][SZ];
FLOATTYPE C[SZ][SZ];

/* The blocked version from Lam, Rothberg and Wolf
 */
void mm3(A,B,C,blocksize)
     FLOATTYPE A[SZ][SZ],
               B[SZ][SZ],
               C[SZ][SZ];
     int blocksize;
{
  int i, j, k, kk, jj;
  FLOATTYPE r;

  for (kk = 0; kk < SZ; kk += blocksize){
    for (jj = 0; jj < SZ; jj += blocksize){
      for (i = 0; i < SZ; i++){
        for (k = kk; k < min(kk+blocksize,SZ); k++){
          r = A[i][k];
          for (j = jj; j < min(jj+blocksize, SZ); j++){
            C[i][j] += r * B[k][j];
          }
        }
      }
    }
  }
}
```

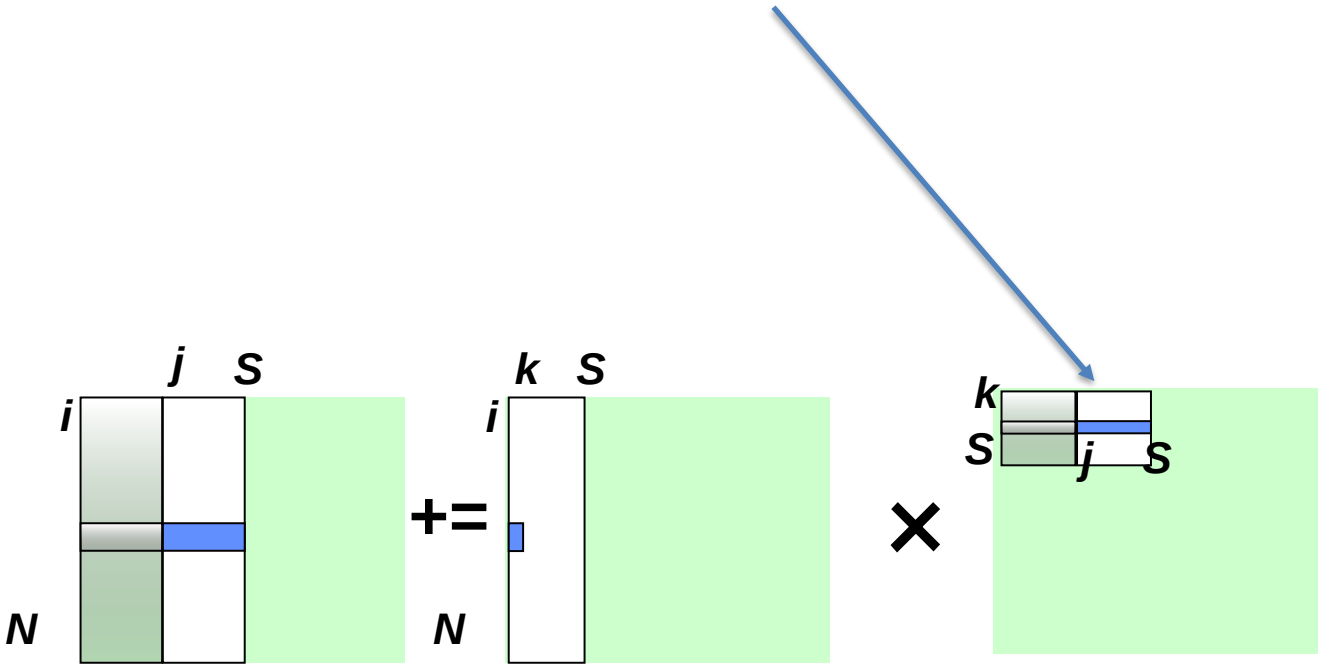
-UU-:----F1 MM3-cleanedup.c 29% L56 (C/*l Abbrev) --

```c
#define ARRAYSZ 8192

FLOATTYPE A[ARRAYSZ][ARRAYSZ];
FLOATTYPE B[ARRAYSZ][ARRAYSZ];
FLOATTYPE C[ARRAYSZ][ARRAYSZ];

/* The blocked version from Lam, Rothberg and Wolf
 */
void mm3(A,B,C,blocksize)
     FLOATTYPE A[ARRAYSZ][ARRAYSZ],
               B[ARRAYSZ][ARRAYSZ],
               C[ARRAYSZ][ARRAYSZ];
     int blocksize;
{
  int i, j, k, kk, jj;
  FLOATTYPE r;

  for (kk = 0; kk < SZ; kk += blocksize){
    for (jj = 0; jj < SZ; jj += blocksize){
      for (i = 0; i < SZ; i++){
        for (k = kk; k < min(kk+blocksize,SZ); k++){
          r = A[i][k];
          for (j = jj; j < min(jj+blocksize, SZ); j++){
            C[i][j] += r * B[k][j];
          }
        }
      }
    }
  }
}

void fillmatrix(A)
```

-UU-:----F1 MM4-cleanedup.c 25% L34 (C/*l Abbrev)

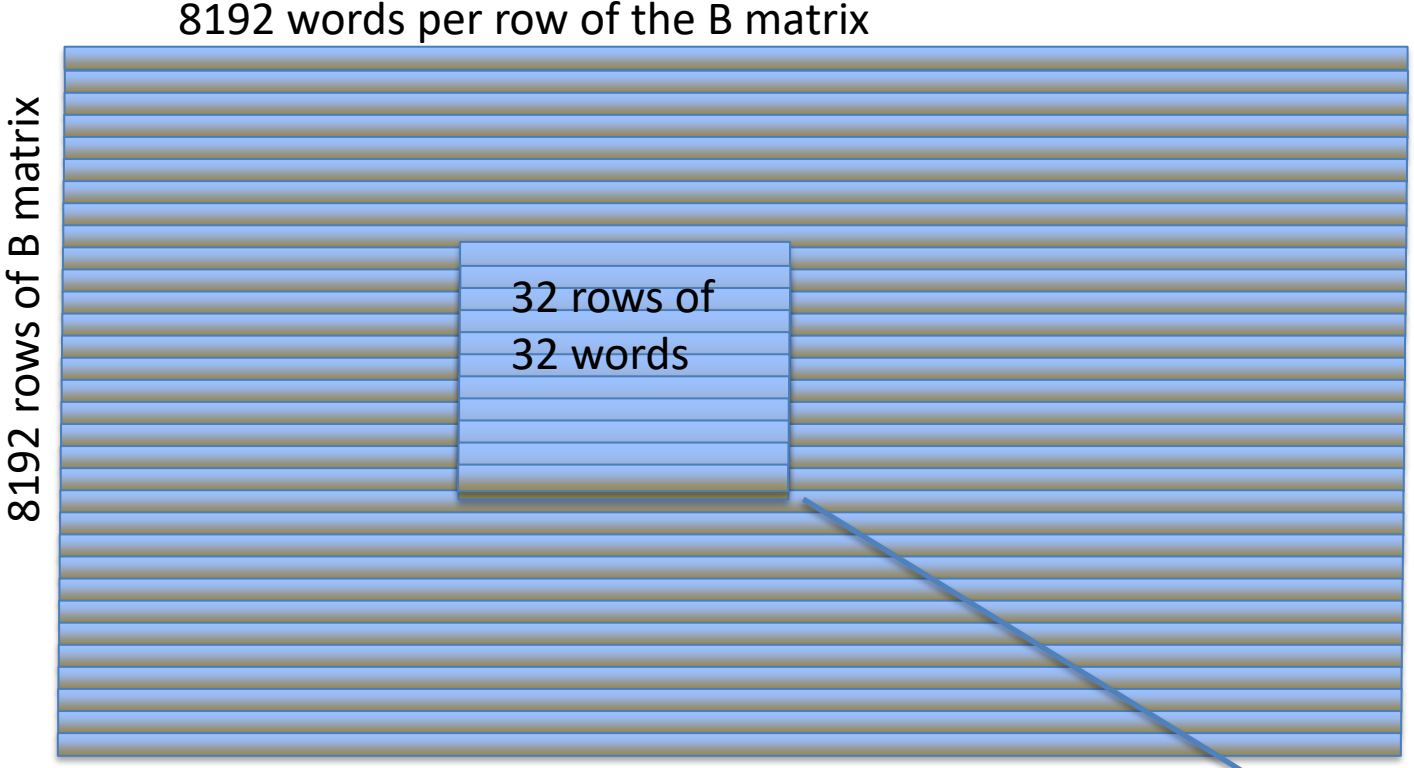
MM4

The optimisation relies on each sub-block fitting into the cache

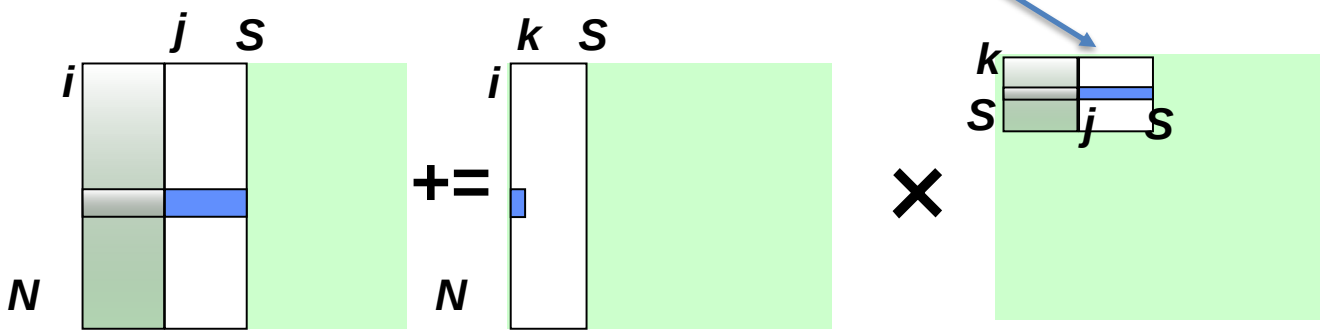


- **We compute a block of row segments of C at a time**
- **Repeatedly re-using a SxS sub-block of B**
- **Then we move on to the next block of row segments of C**
- No need to revisit the first submatrix of B

MM4

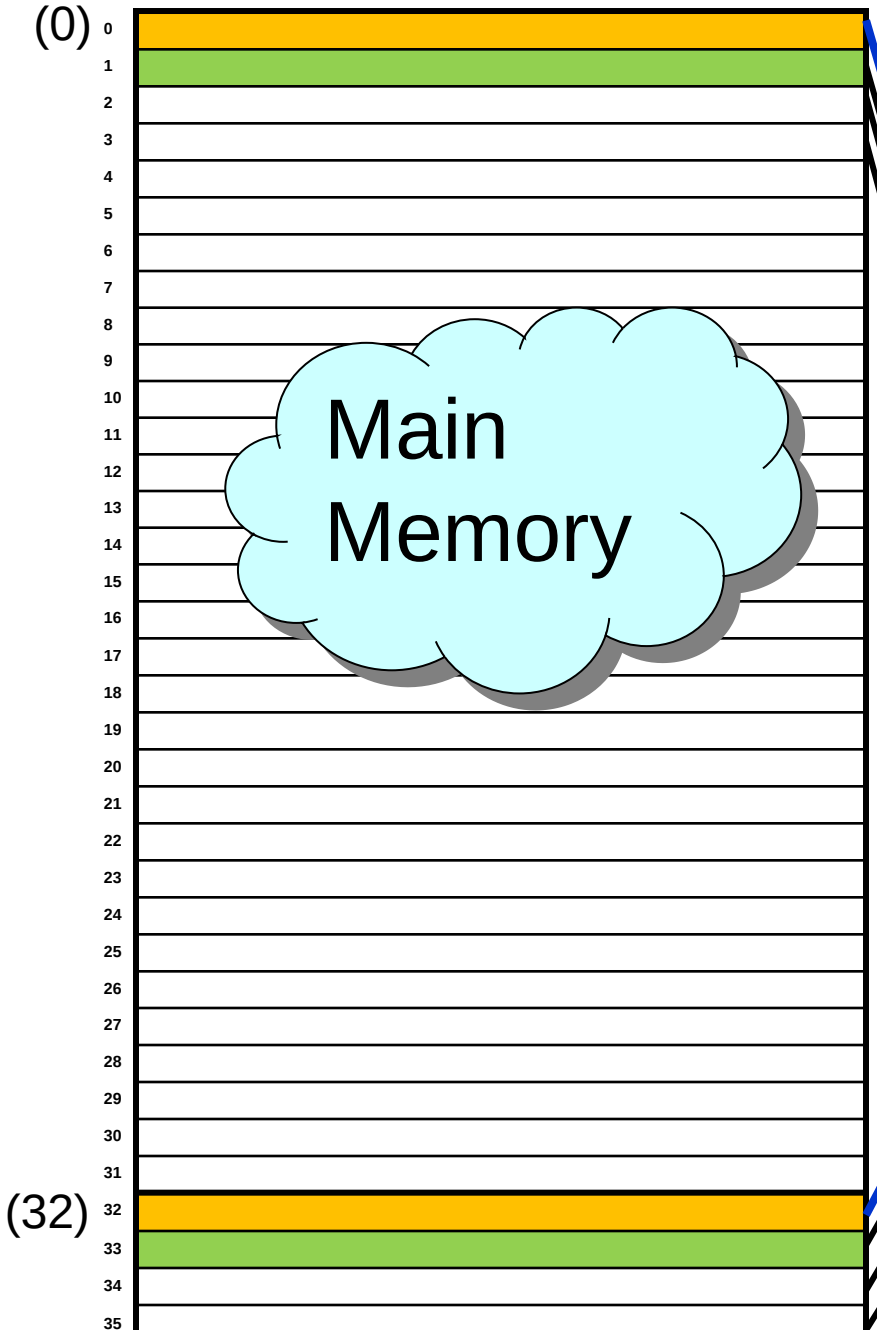


- Each 32x32 block of B needs to fit in the cache in order to be reused
- Each 32-word row of the block is aligned to a multiple of 8192

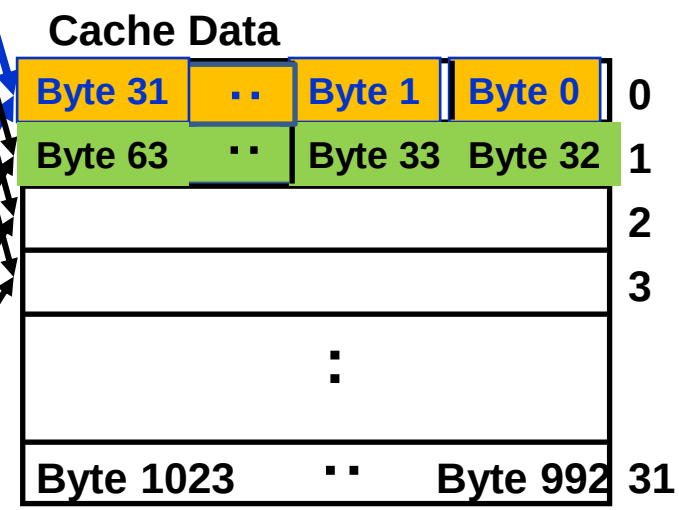


- **We compute a block of row segments of C at a time**
- **Repeatedly re-using a SxS sub-block of B**
- **Then we move on to the next block of row segments of C**
- No need to revisit the first submatrix of B

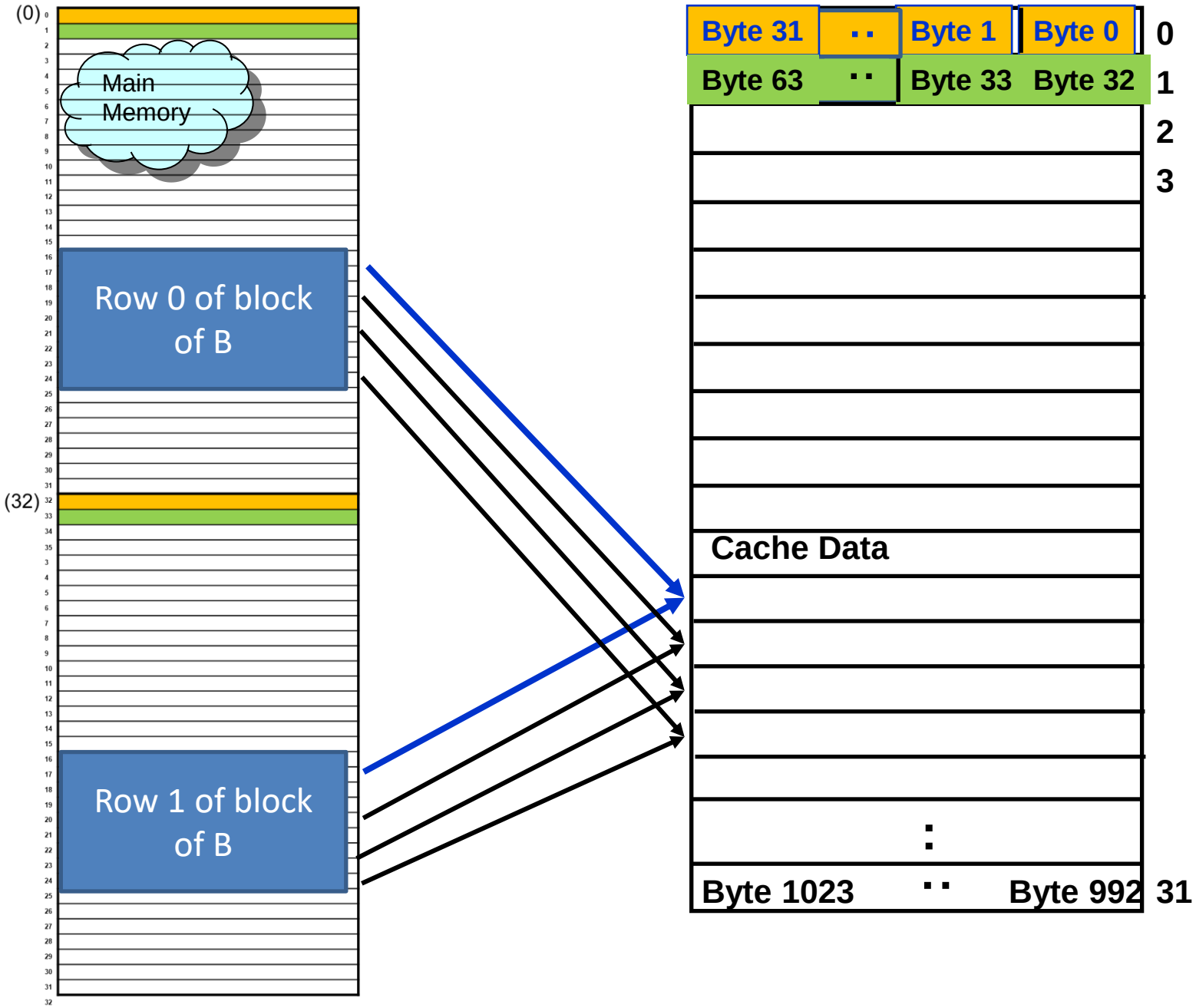
Recall: 1 KB Direct Mapped Cache, 32B blocks



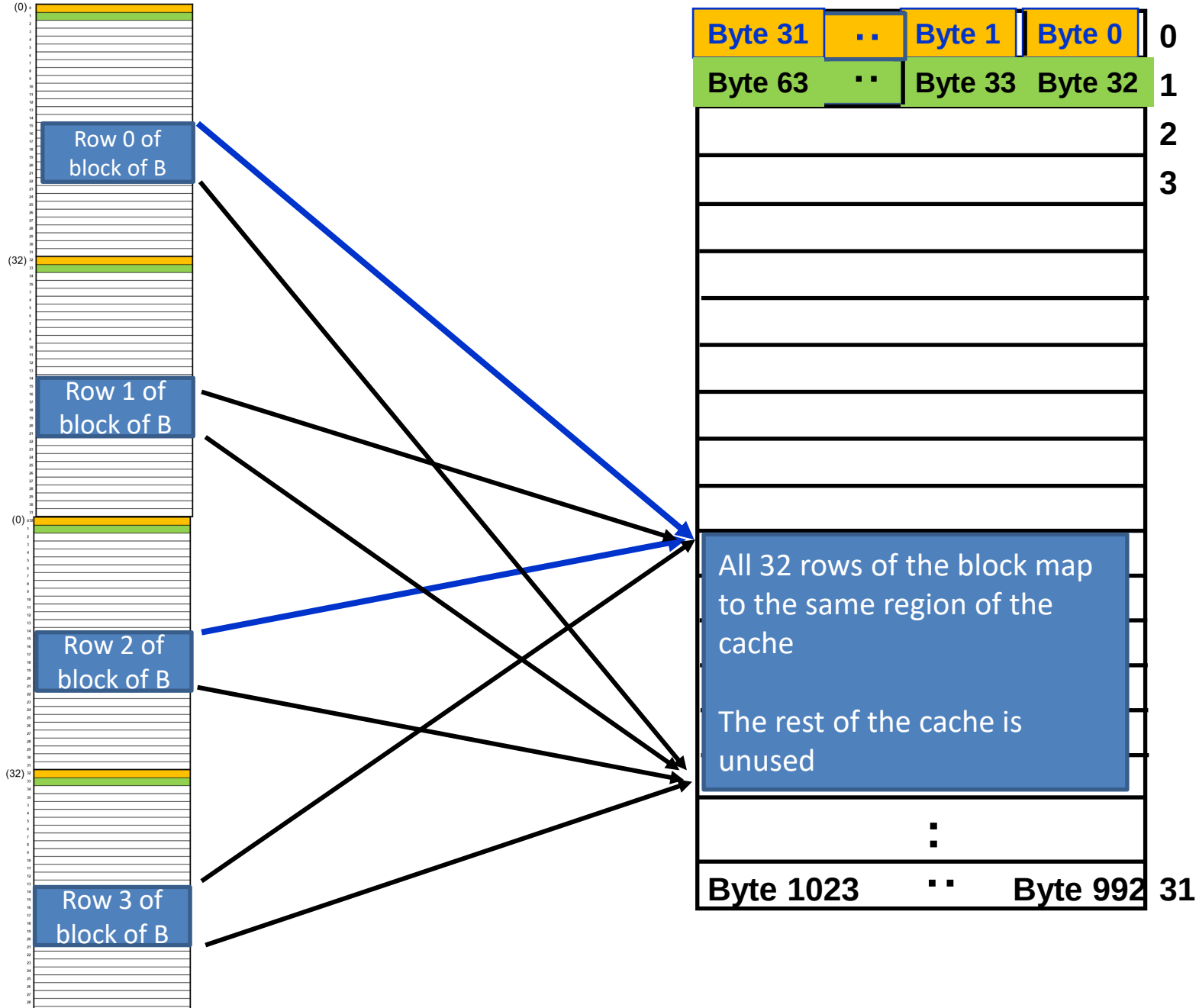
- **Cache location 0 can be occupied by data from main memory location 0, 32, 64, … etc.**
- **Cache location 1 can be occupied by data from main memory location 1, 33, 65, … etc.**
  - **In general, all locations with same Address<9:4> bits map to the same location in the cache Which one should we place in the cache?**
- **How can we tell which one is in the cache?**



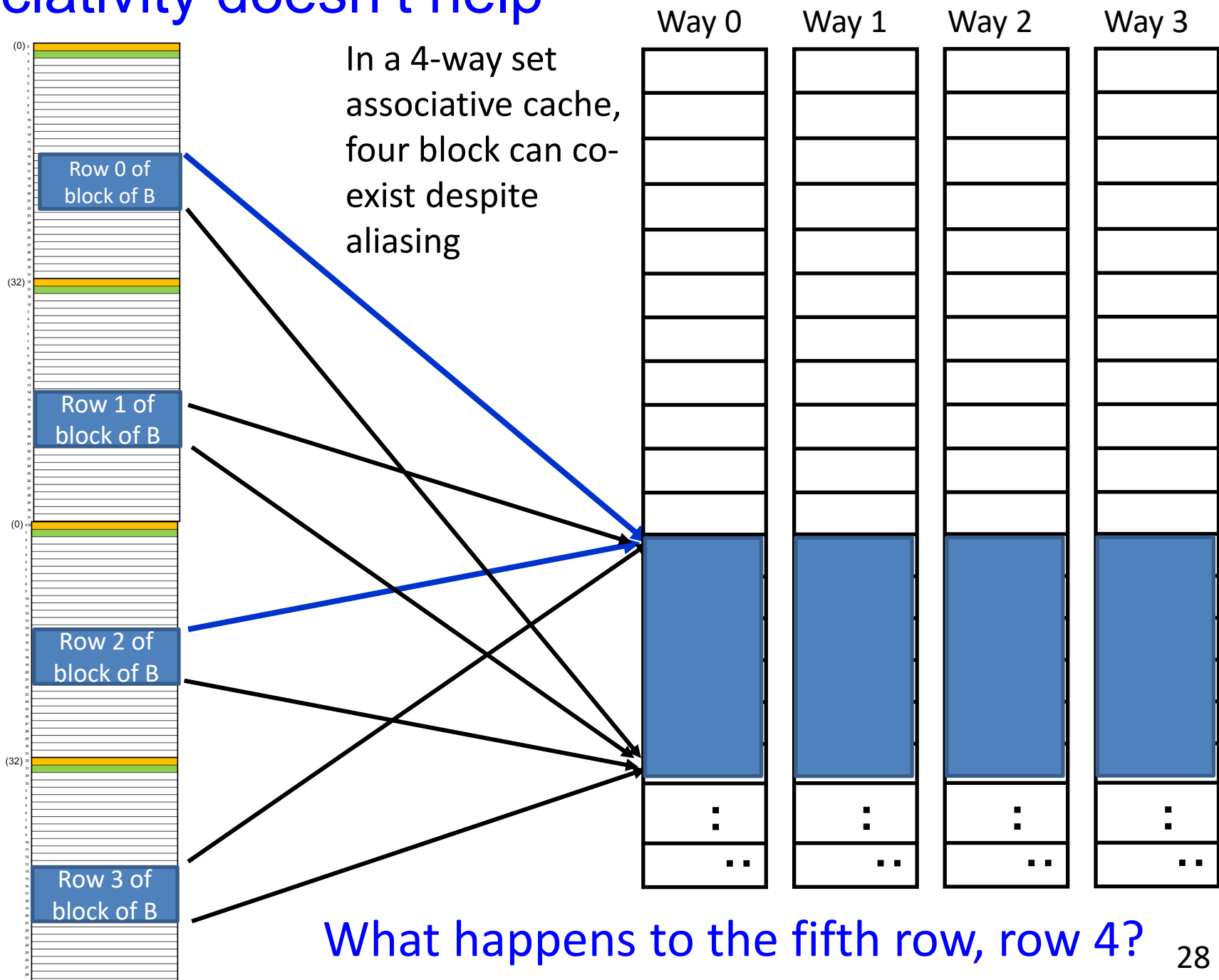
Different rows of the submatrix map to the same sets



Different rows of the submatrix map to the same sets



Associativity doesn't help



In a 4-way set associative cache, four block can co-exist despite aliasing

What happens to the fifth row, row 4?

What is possible?

```
ray03(phjk) 150 % ./scripts/make-big (SZ = 10112)

ray03(phjk) 151 % ./MM3.x86
mm3: 332.160784 s, 6225.766404 MFLOPS

ray03(phjk) 152 % setenv OPENBLAS_NUM_THREADS 1
ray03(phjk) 153 % ./MM5-blas.x86
mm5: 31.954071 s, 64716.494179 MFLOPS (per core)
mm5: 32.000000 s, 64623.607808 MFLOPS (all cores)

ray03(phjk) 154 % setenv OPENBLAS_NUM_THREADS 4
ray03(phjk) 155 % ./MM5-blas.x86
mm5: 35.034614 s, 59026.066331 MFLOPS (per core)
mm5: 9.000000 s, 229772.827762 MFLOPS (all cores)
```

Moral: *never* write your own matrix multiply!

(ray03: Intel(R) Core(TM) i7-7700K CPU @ 4.20GHz “Kaby Lake”)

What is possible?

```
ray03(phjk) 150 % ./scripts/make-big (SZ = 10112)

ray03(phjk) 151 % ./MM3.x86
mm3: 332.160784 s, 6225.766404 MFLOPS

ray03(phjk) 152 % setenv OPENBLAS_NUM_THREADS 1
ray03(phjk) 153 % ./MM5-blas.x86
mm5: 31.954071 s, 64716.494179 MFLOPS (per core)
mm5: 32.000000 s, 64623.607808 MFLOPS (all cores)

ray03(phjk) 154 % setenv OPENBLAS_NUM_THREADS 4
ray03(phjk) 155 % ./MM5-blas.x86
mm5: 35.034614 s, 59026.066331 MFLOPS (per core)
mm5: 9.000000 s, 229772.827762 MFLOPS (all cores)
```

(10x MM1)

wallclock time

total CPU time (4 CPU cores busy)

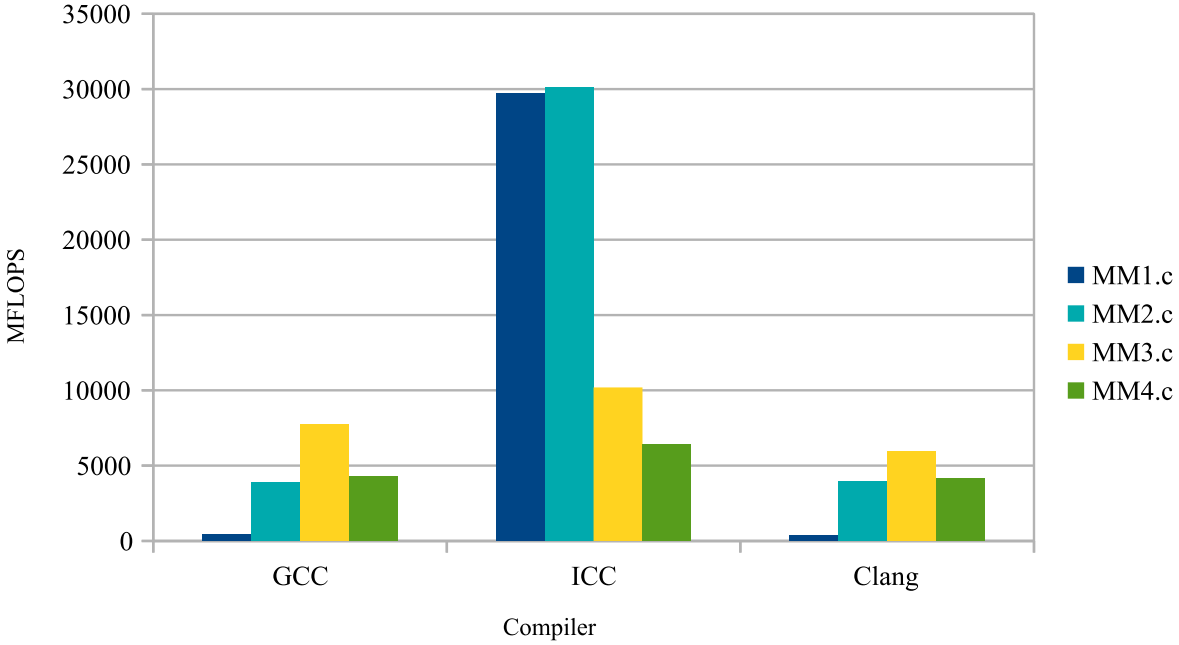
Moral: *never* write your own matrix multiply!

(ray03: Intel(R) Core(TM) i7-7700K CPU @ 4.20GHz "Kaby Lake")

What is possible?

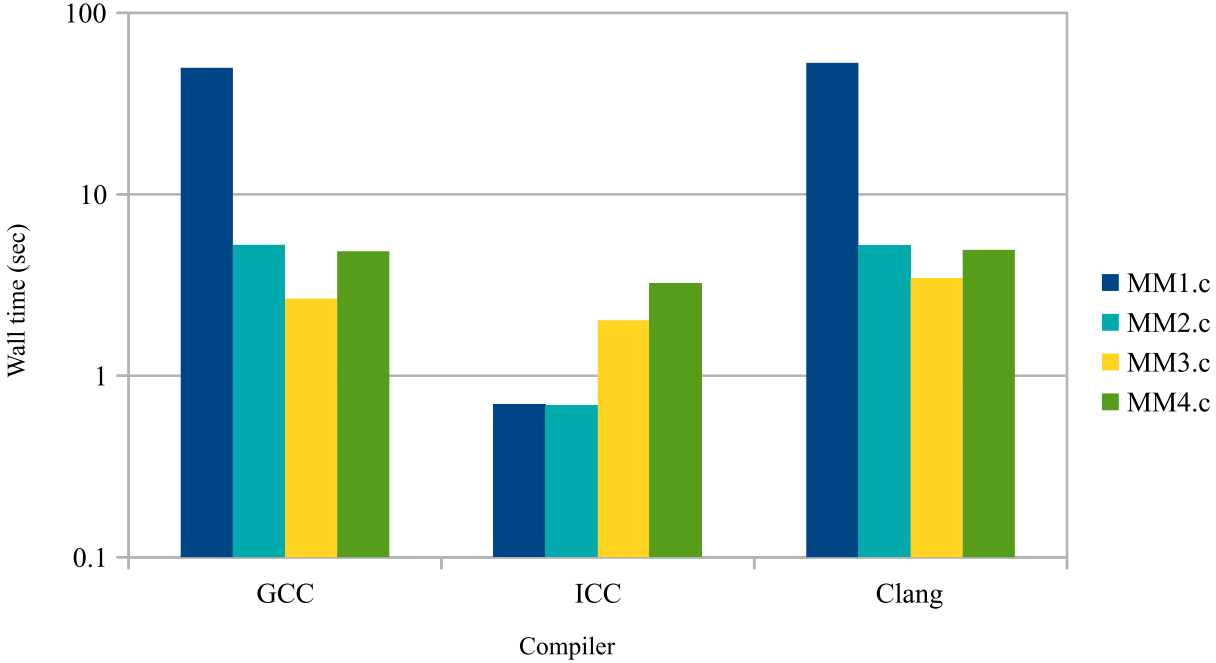
```
ray03(phjk) 150 % ./scripts/make-big (SZ = 10112)

ray03(phjk) 151 % ./MM3.x86
mm3: 332.160784 s, 6225.766404 MFLOPS

ray03(phjk) 152 % setenv OPENBLAS_NUM_THREADS 1
ray03(phjk) 153 % ./MM5-blas.x86
mm5: 31.954071 s, 64716.494179 MFLOPS (per core)
mm5: 32.000000 s, 64623.607808 MFLOPS (all cores)

ray03(phjk) 154 % setenv OPENBLAS_NUM_THREADS 4
ray03(phjk) 155 % ./MM5-blas.x86
mm5: 35.034614 s, 59026.066331 MFLOPS (per core)
mm5: 9.000000 s, 229772.827762 MFLOPS (all cores)
```

See: https://www.openblas.net/
And the code at https://github.com/xianyi/OpenBLAS
For example:
https://github.com/xianyi/OpenBLAS/blob/develop/kernel/x86_64/cgemm_kernel_4x8_sandy.S



Performance varies with different compilers

What happens with the Intel compiler?



```c
/*
 * mm
 *
 * Multiply A by B leaving the result in C.
 * A is assumed to be an l x m matrix, B an m x n matrix.
 * The result matrix C is of course l x n.
 * The result matrix is assumed to be initialised to zero.
 *
 * Less dumb.
 */
void mm2(A,B,C)
    FLOATTYPE A[SZ][SZ], B[SZ][SZ], C[SZ][SZ];
{
  int i, j, k;

  for (i = 0; i < SZ; i++){
    for (k = 0; k < SZ; k++){
      for (j = 0; j < SZ; j++){
        C[i][j] += A[i][k] * B[k][j];
      }
    }
  }
}

void fillmatrix(A)
    FLOATTYPE A[SZ][SZ];
{
  int i, j;
  FLOATTYPE drand48();

  for (i = 0; i < SZ; i++){
    for (j = 0; j < SZ; j++){
      A[i][j] = drand48();
```

```asm
        orb     %al, %r14b
        je      .L21
        cmpq    %r8, %rdx
        setnb   %al
        orb     %r15b, %al
        je      .L21
        vbroadcastsd    (%rdi), %ymm2
        xorl    %eax, %eax
.L15:
        vmovupd (%rcx,%rax), %xmm3
        vinsertf128     $0x1, 16(%rcx,%rax), %ymm3, %ymm0
        vmovupd (%rdx,%rax), %xmm4
        vinsertf128     $0x1, 16(%rdx,%rax), %ymm4, %ymm1
        vmulpd  %ymm2, %ymm0, %ymm0
        vaddpd  %ymm1, %ymm0, %ymm0
        vmovups %xmm0, (%rdx,%rax)
        vextractf128    $0x1, %ymm0, 16(%rdx,%rax)
        addq    $32, %rax
        cmpq    $17408, %rax
        jne     .L15
        addq    $17408, %rcx
        addq    $8, %rdi
        cmpq    %r11, %rcx
        jne     .L20
.L16:
        movq    %rbx, %rdx
        addq    $17408, %rsi
        addq    $17408, %r10
        addq    $17408, %r9
        cmpq    %r13, %rbx
        je      .L27
        addq    $17408, %rbx
```

Try this yourself using the Compiler Explorer, https://godbolt.org/

```c
/*
 * mm
 *
 * Multiply A by B leaving the result in C.
 * A is assumed to be an l x m matrix, B an m x n matrix.
 * The result matrix C is of course l x n.
 * The result matrix is assumed to be initialised to zero.
 *
 * Less dumb.
 */
void mm2(A,B,C)
    FLOATTYPE A[SZ][SZ], B[SZ][SZ], C[SZ][SZ];
{
  int i, j, k;

  for (i = 0; i < SZ; i++){
    for (k = 0; k < SZ; k++){
      for (j = 0; j < SZ; j++){
        C[i][j] += A[i][k] * B[k][j];
      }
    }
  }
}

void fillmatrix(A)
    FLOATTYPE A[SZ][SZ];
{
  int i, j;
  FLOATTYPE drand48();

  for (i = 0; i < SZ; i++){
    for (j = 0; j < SZ; j++){
      A[i][j] = drand48();
```

```asm
        jb          ..B1.36       # Prob 10%                          #76.7
                                  # LOE rax rdx rbx rdi r8 r9 r10 r11 r12
                                  # Execution count [4.73e+06]
        vbroadcastsd %xmm1, %ymm0                                     #77.20
        movl        %esi, %r15d                                       #76.7
        xorl        %ecx, %ecx                                        #76.7
                                  # LOE rax rdx rcx rbx rdi r8 r9 r10 r11
..B1.22:                          # Preds ..B1.22 ..B1.21
                                  # Execution count [1.03e+10]
                # optimization report
                # LOOP WAS BLOCKED BY 128
                # LOOP WAS VECTORIZED
                # VECTORIZATION SPEEDUP COEFFECIENT 4.554688
                # VECTOR TRIP COUNT IS ESTIMATED CONSTANT
                # VECTOR LENGTH 4
                # NORMALIZED VECTORIZATION OVERHEAD 0.140625
                # MAIN VECTOR TYPE: 64-bits floating point
        vmulpd      B(%r13,%rcx,8), %ymm0, %ymm2                      #77.30
        vaddpd      C(%rax,%rcx,8), %ymm2, %ymm3                      #77.9
        vmovupd     %ymm3, C(%rax,%rcx,8)                             #77.9
        addq        $4, %rcx                                          #76.7
        cmpq        %r10, %rcx                                        #76.7
        jb          ..B1.22       # Prob 99%                          #76.7
                                  # LOE rax rdx rcx rbx rdi r8 r9 r10 r11
..B1.24:                          # Preds ..B1.22 ..B1.36
                                  # Execution count [4.73e+06]
        movslq      %r15d, %r15                                       #76.7
        cmpq        %r10, %r15                                        #76.7
        jae         ..B1.28       # Prob 0%                           #76.7
                                  # LOE rax rdx rbx rdi r8 r9 r10 r11 r12
..B1.26:                          # Preds ..B1.24 ..B1.26
                                  # Execution count [1.03e+10]
```