Performance and Optimisation

Jan Thorbecke

Contents

- Performance
- Benchmarks
- Optimization techniques
- Examples
  - matrix multiplication
  - correlation
  - 2D convolution

Summary from hardware part

- Machine details are important for performance:
  - processor and memory system
  - before going to parallelisation first get serial performance right
- There is parallelism hidden within the processor
  - superscalar, SIMD(vector), pipelining
- Locality is at least as important as computation
  - Temporal: re-use of recently used data
  - Spatial: using data nearby recently used
- Machines have memory hierarchy
  - reading from DRAM takes ~100 cycles
  - cache is fast memory to bridge the gap with DRAM

Performance limitations

- Clock bound
  - all data in cache
  - faster clock improves performance
- Memory Bound
  - most data has to come from main-memory
  - faster memory improves performance
- IO Bound
  - wait for IO to be completed
  - faster IO hardware, or hide IO processes
- Communication bound
  - parallel wait for data from other processing elements

Amdahl's scaling (law)

$$t_{improved} = \frac{t_{affected}}{r_{speedup}} + t_{unaffected}$$

Example:
"Suppose a program runs in 100 seconds on a machine, where multiplies are executed 80% of the time. How much do we need to improve the speed of multiplication if we want the program to run 4 times faster?"

25 = 80/r + 20 r = 16x

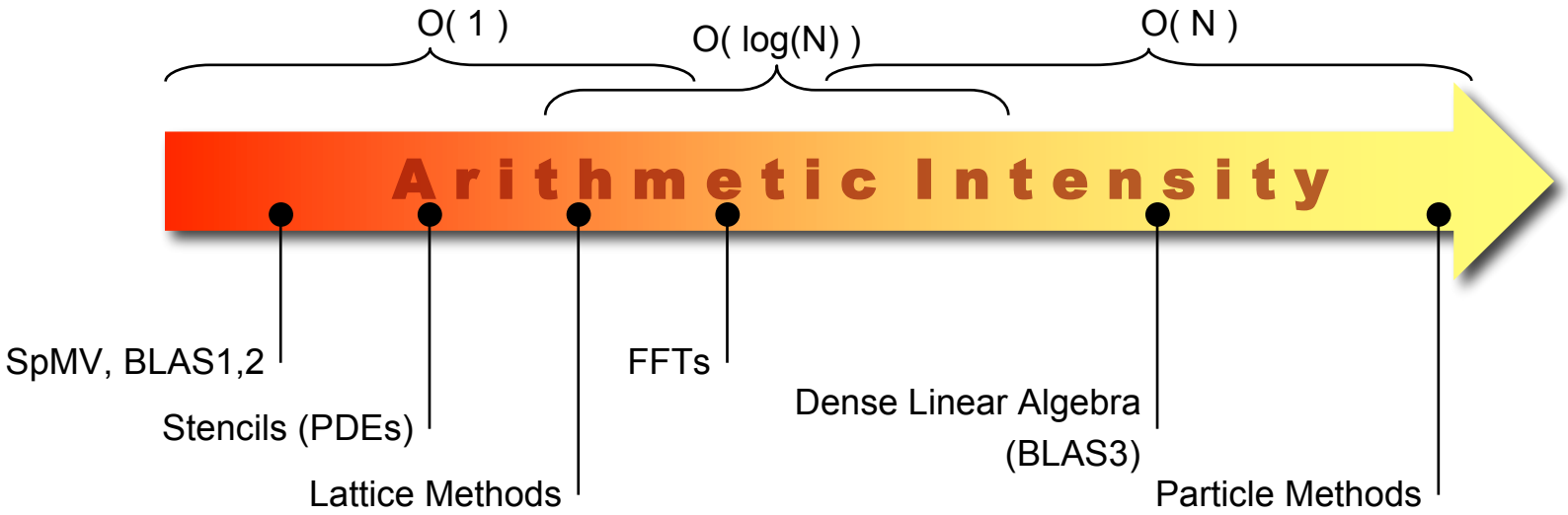
How about making it 5 times faster?

20 = 80/r + 20 r = ?

Principle: Make the common case fast

Performance limitations model

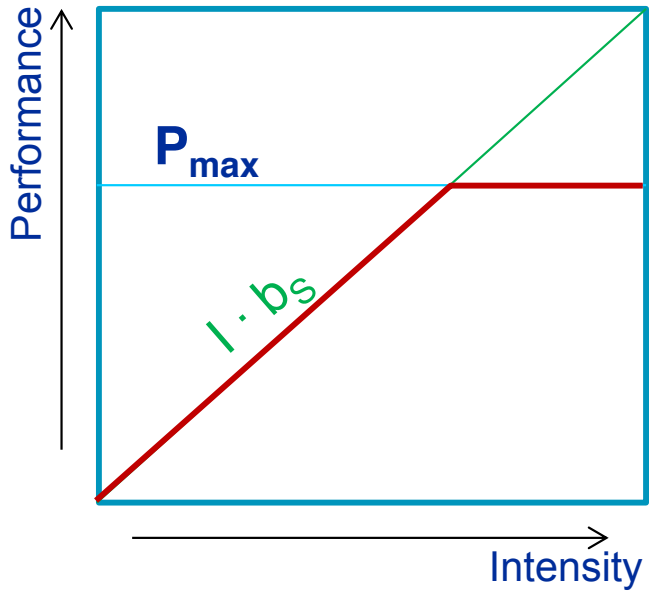
- Roofline model
  - Arithmetic Intensity is the ratio of total floating-point operations to total data movement (bytes).



- http://crd.lbl.gov/departments/computer-science/PAR/research/roofline/
- http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-134.pdf

Roofline Model[1,2]

- P = number of tasks processed / second
- Bottleneck is either
  - max instructions / sec : P_{max}
  - max data needed by instrcuctions / sec : $I * b_s$
- P = min (P_{max}, $I * b_s$)



[1] W. Schönauer: Scientific Supercomputing: Architecture and Use of Shared and Distributed Memory Parallel Computers. (2000)

[2] S. Williams: Auto-tuning Performance on Multicore Computers. UCB Technical Report No. UCB/EECS-2008-164. PhD thesis (2008)

Roofline Model

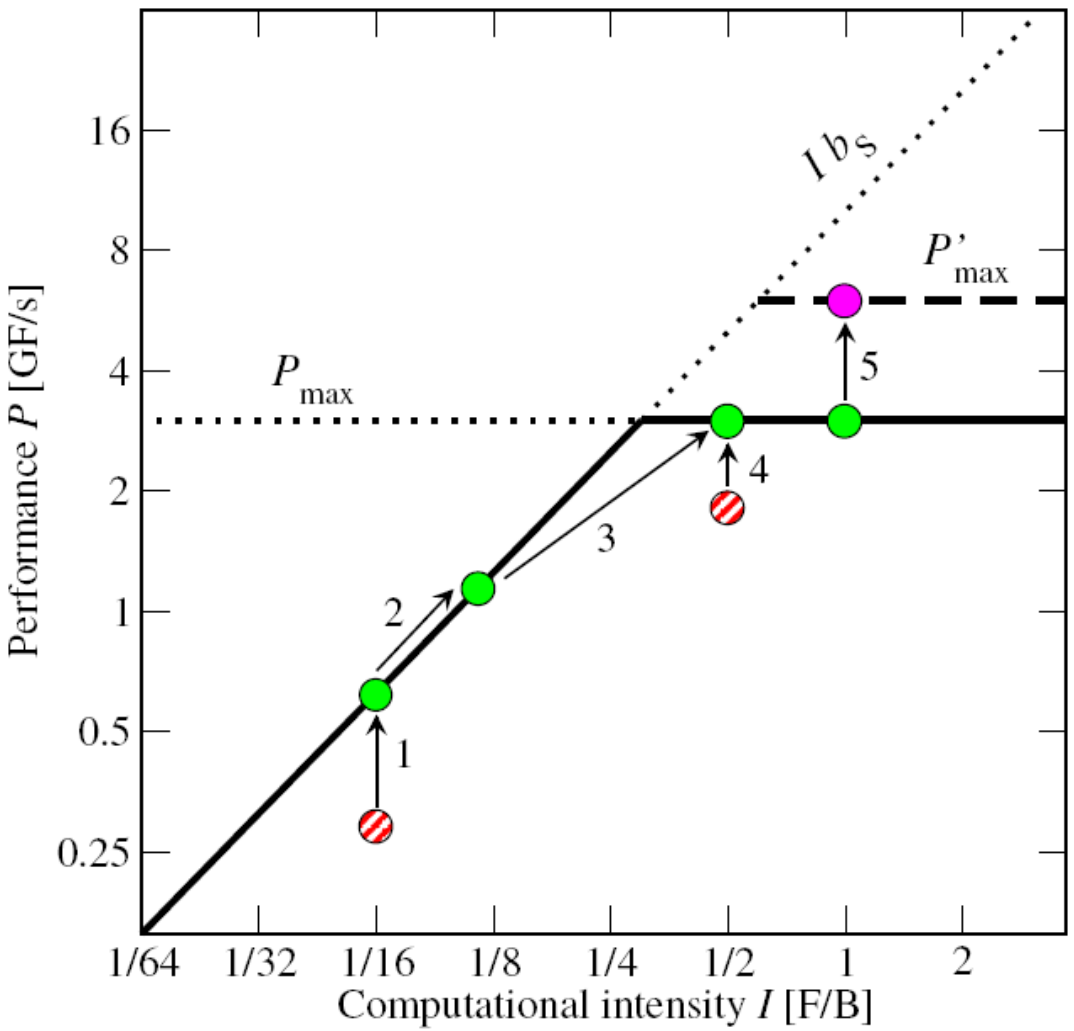
- P_{max} = Applicable peak performance of a loop, assuming that data comes from L1 cache (this is not necessarily P_{peak})
- *I* = Computational intensity ("work" per byte transferred) over the slowest data path utilized ("the bottleneck")
- b_S = Applicable peak bandwidth of the slowest data path utilized

Flop/Byte Byte/s

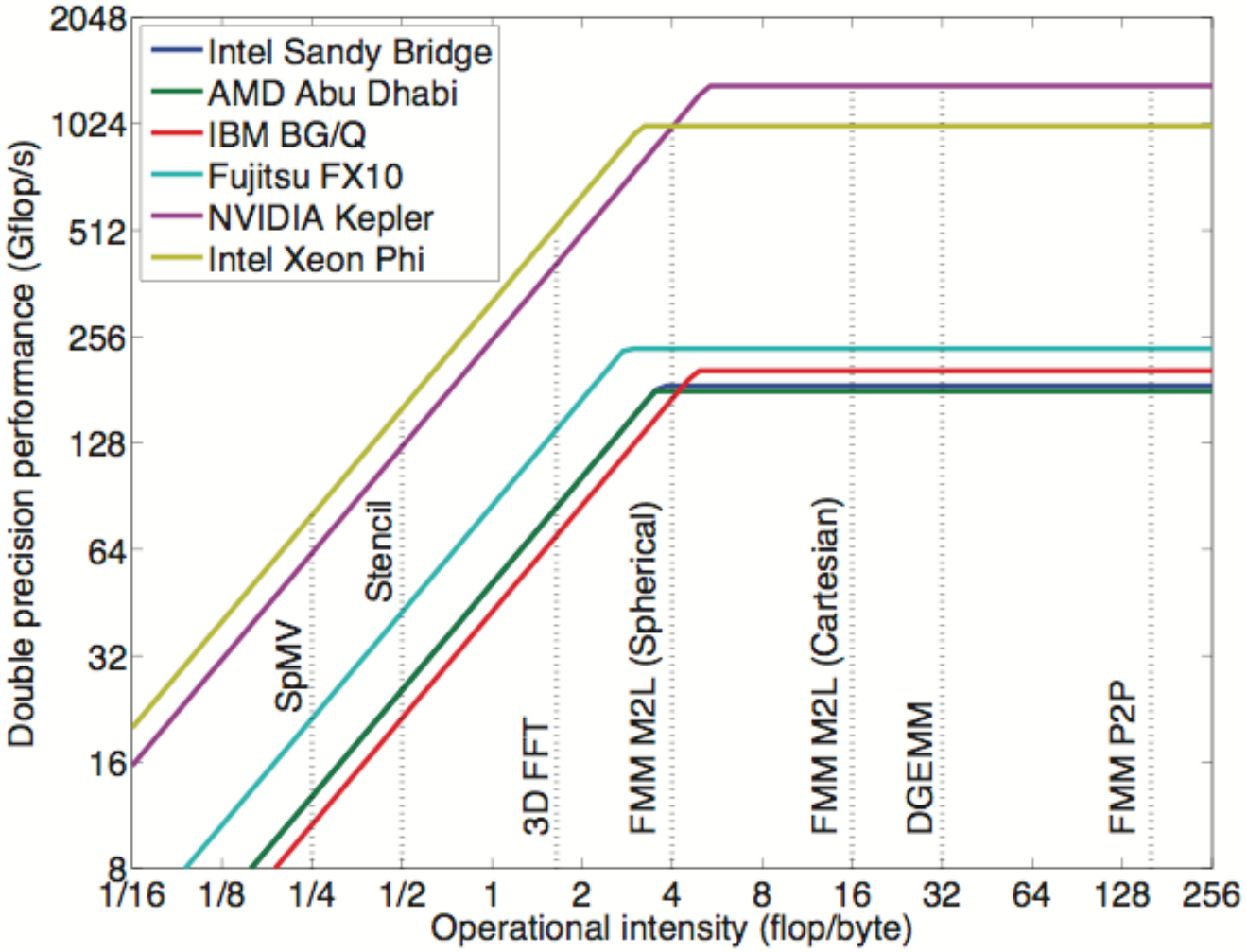
$$P = \min (P_{max}, I * b_s)$$

Typical code optimizations in the Roofline Model

1. Hit the BW bottleneck by good serial code
2. Increase intensity to make better use of BW bottleneck
3. Increase intensity and go from memory-bound to core-bound
4. Hit the core bottleneck by good serial code
5. Shift P_{max} by accessing additional hardware features or using a different algorithm/implementation



Roofline model



Optimization

- Finally the real work.

Understand the target system Hardware and Software

- Hardware
  - Node Architecture
  - Interconnect
  - Input-Output
- Software
  - Operating System
  - Parallel I/O software
  - Programming Environment
    - Compilers
- Programming Considerations
  - Cache Optimization
  - Vectorization
  - Efficient MPI
  - OpenMP

Optimization Categories

- Improve algorithm
  - new algorithm
  - re-use of already computed data
- Improve CPU performance
  - vectorization (high level)
  - create more opportunities to go superscalar (high level)
  - better instruction scheduling (low level)
  - Strength reduction: replace expensive operations with less expensive
- Improve memory performance (often most important!)
  - better memory access pattern
  - Optimal use of cache lines (spatial locality)
  - re-usage of cached data (temporal locality)

Optimization Techniques

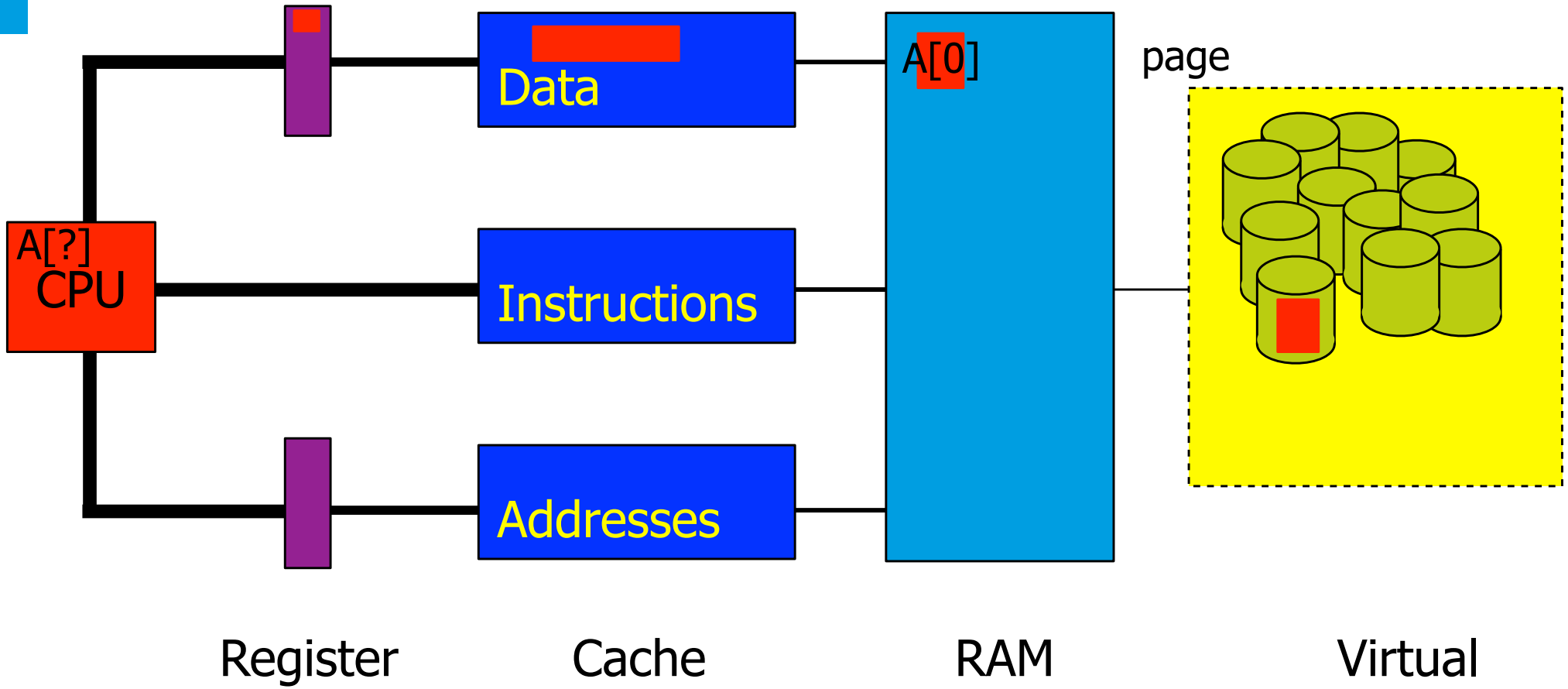
- Loop based
  - Interchange
  - Unrolling
  - Fusion and Fission
  - Blocking
- Other
  - procedure/function inlining
  - array padding
  - prefetching
- Some of these techniques are built into the compiler.

Warning before you start

- During optimization work always keep a copy of the original code
  - or use a software version control system (like git).
- Compare the results of your optimized code with a reference result.
  - expect small changes (difference ~1e-6) in answers during optimization
- After every step always verify with your reference answer.

Memory

waiting for memory



Memory Reference

- TEMPORAL LOCALITY: if a location is referenced it is likely to be referenced again in the near future

- SPATIAL LOCALITY: if a location is referenced it is likely that locations nearby will be referenced in the near future.

Near Future

- For a loop, compute the number of iterations separating two consecutive use of the same data: reuse distance

```
DO I= 1, 10000
      = A(I)
      = A(I+K)   REUSE DISTANCE K
ENDDO
```

The basic idea will be to try to keep A(I+K) (referenced at iteration I) in a fast memory level until its next reuse by A(I) at iteration I+K.

Nearby Locations

- Assuming FORTRAN storage rules: column oriented for 2D arrays: A(1000,1000)

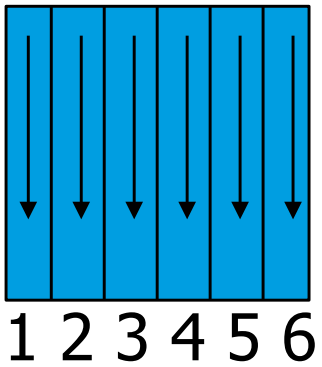
```
DO I= 1, 1000
   DO J = 1, 1000
      = A(J,I)   Perfect Spatial locality : consecutive references are accessing
                 consecutive memory locations on A
      = B(I,J)   Far away: two consecutive references are 1000 x8 Bytes away !
```

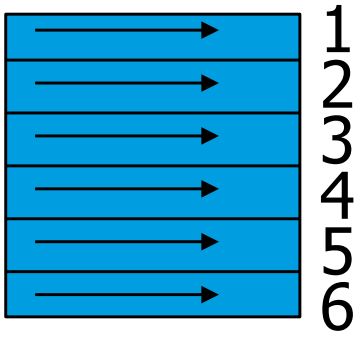
- Usually, exploitable spatial locality (for caches) implies distance less than 64-128 Bytes.

- You have to understand how your data structure are mapped into the address space.

Storage in memory

- The storage order is language dependent
  - Fortran stores "column-wise"
  - C-stores "row-wise"





- Accessing elements in storage order greatly enhances the performance for problem sizes that do not fit in the cache.

Array indexing

```
DO j=1, M                    o
   DO i=1, N
      A(i,j)..
   END DO
END DO
```
Direct

```
DO j=1, M                    +
   DO i=1, N
      A(i+(j-1)*N)..
   END DO
END DO
```
Explicit

```
DO j=1, M                    +
   DO i=1, N
      k = k + 1
      A(k)..
   END DO
END DO
```
Loop carried

```
DO j=1, M                    -
   DO i=1, N
      A(index(i,j))..
   END DO
END DO
```
Indirect

Array Indexing C-example

AMD (pgi):

```
Direct index time         = 2.173686e-02   3
Explicit index time       = 2.063994e-02   2
Loop carried index time = 2.012025e-02   1
Indirect time             = 2.593643e-02   4
```

Intel (icc):

```
Direct index time         = 2.629449e-02   3
Explicit index time       = 2.561279e-02   2
Loop carried index time = 2.391711e-02   1
Indirect time             = 2.712800e-02   4
```

Differences are also compiler dependent !
see exercise ArrayIndex

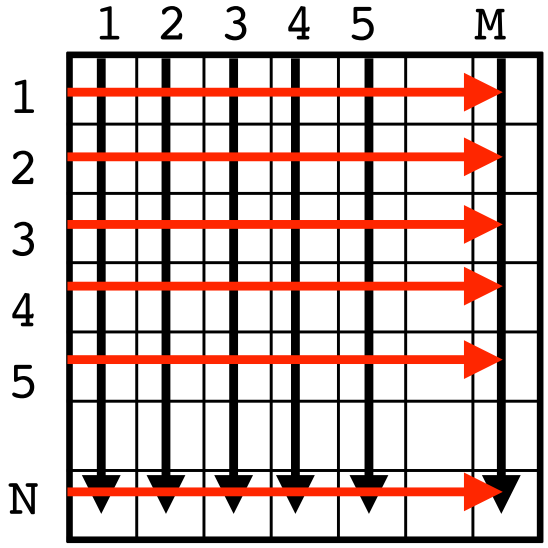
Loop interchange

- Basic idea: in a nested loop, examine and possibly change the order of the loops
- Advantages:
  - Better memory access pattern (*improved cache and memory usage*)
    - improves spatial locality (*re-use of data already in cache*)
  - Elimination of data dependencies (*to increase CPU optimisation*)
- Drawbacks:
  - May make a short loop most innermost (*which is not good for optimal software pipelining (SWP)*)

Loop Interchange (Fortran)

Original (slow)

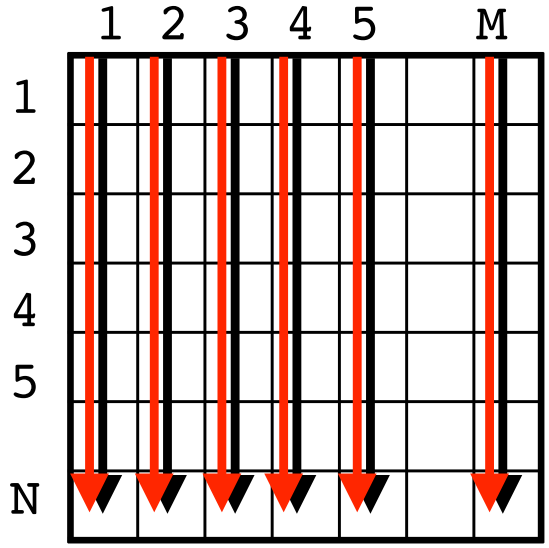
```
DO i=1, N
   DO j=1, M
      C(i,j)=A(i,j)+B(i,j)
   END DO
END DO
```



Access order

Interchanged (fast)

```
DO j=1, M
   DO i=1, N
      C(i,j)=A(i,j)+B(i,j)
   END DO
END DO
```



Storage order

Loop Interchange (C)

In C, the situation is exactly the opposite

Original (slow)

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

Interchanged (fast)

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

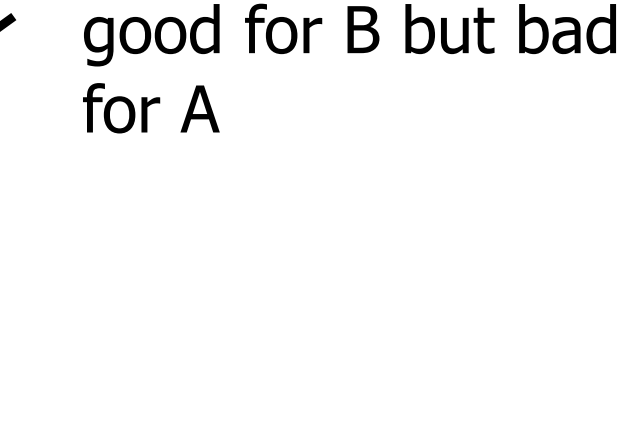
Loop Interchange Example

```
DO i=1, 400
   DO j=1, 400
      DO k=1, 400
         A(i,j,k) = A(i,j,k)+B(i,j,k)*C(i,j,k)
      END DO
   END DO
END DO
```

```
time i j k =    5.3078700
time i k j =    4.3588700
time j i k =    2.1240350
time j k i =    0.2539180
time k i j =    0.7396700
time k j i =    0.2315980
```

Loop Interchange

```
COMMON /SHARE/B(M,N)
......
SUBROUTINE WORK(A,N,M)

REAL A(N,M)
DO i=1, M
   DO j=1, N
      A(j,i) = A(j,i)+B(i,j)
   END DO
END DO
```



Index reversal on B (i.e. allocate B as B(N,M)) will do the trick, but work must de done everywhere B is used. Sometimes a tmp variable which contains the transpose of B can also help.

This also holds for Matlab
Matlab uses Fortran array order

```
parfor n=1:nt%nt/2+1
  P_w(n,:,:)=squeeze(W_w(n,:,:))*
             squeeze(Q_w(n,:,:))*squeeze(W_w(n,:,:));
end
```

=> 77 seconds

```
parfor n=1:nt%nt/2+1
  P_w(:,:,n)=squeeze(W_w(:,:,n))*
             squeeze(Q_w(:,:,n))*squeeze(W_w(:,:,n));
end
```

=> 57 seconds

Python has C ordering

```
import time
import numpy as np

start_time = time.time()

data = np.ones(shape=(1000, 1000), dtype=np.float)

for i in range(1000):
    for j in range(1000):
        data[i][j] *= 1.0000001
        data[i][j] *= 1.0000001
        data[i][j] *= 1.0000001
        data[i][j] *= 1.0000001
        data[i][j] *= 1.0000001

end_time = time.time()

print("Run time = {}".format(end_time - start_time))
```

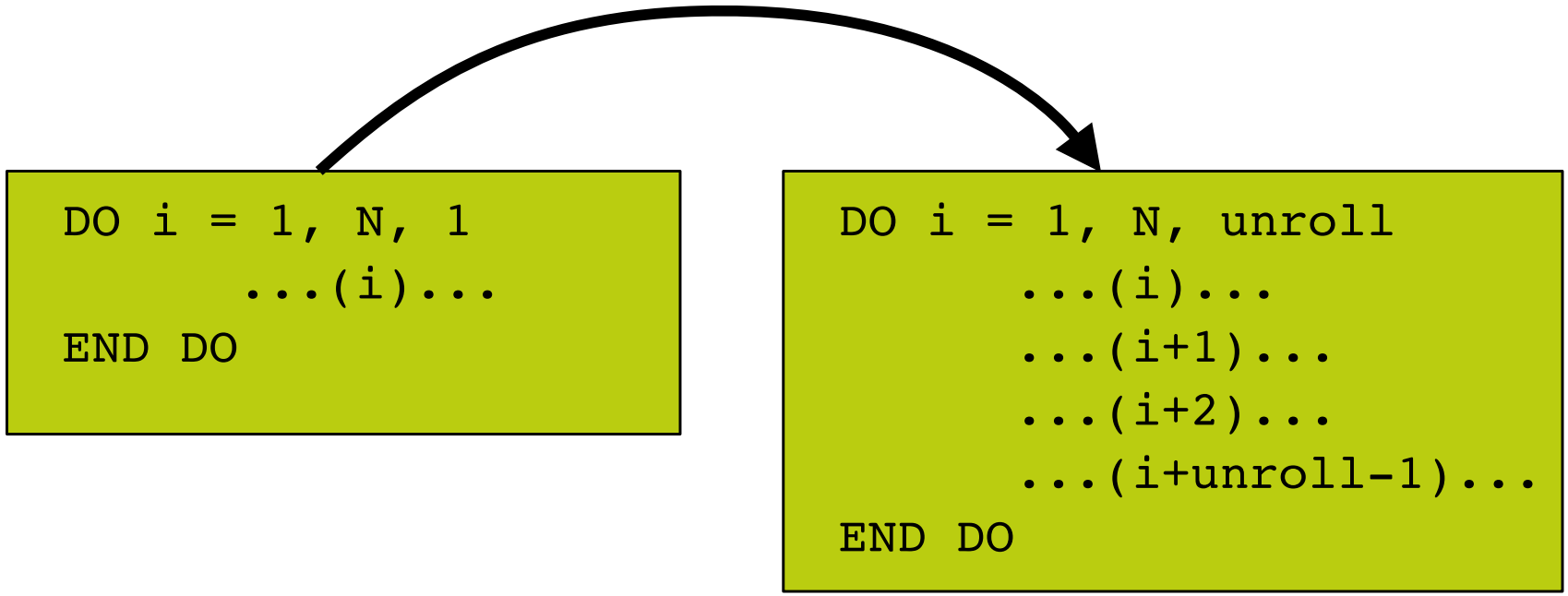
Run time = 3.079

Python try to use numpy arrays

```
import time
import numpy as np

start_time = time.time()

data = np.ones(shape=(1000, 1000), dtype=np.float)


for i in range(5):
    data *= 1.0000001


end_time = time.time()

print("Run time = {}".format(end_time - start_time))
```

Run time = 0.003

Loop Unrolling



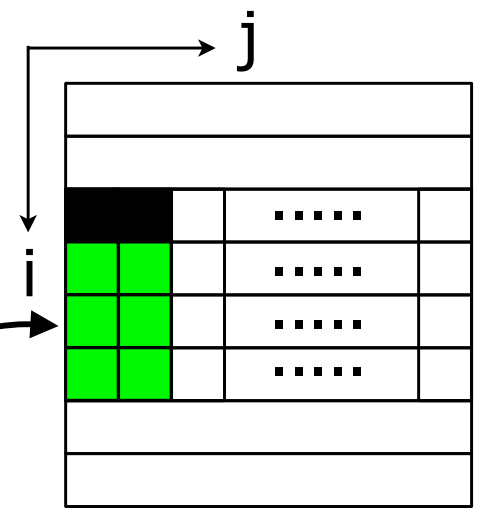
Loop Unrolling

- Advantages:
  - increase basic block sizes and always applicable.
  - more opportunities for superscalar code
  - more data re-usage, exploit cache lines
  - reordering the statements with the increased loop body.
  - reduction in loop overhead (minor advantage)

- Drawbacks:
  - Choose the right degree of unrolling not always obvious
  - Increases register usage and code size
  - Does not always improve performance in particular on modern Out of Order machines. Hardware is performing dynamically loop unrolling.
  - Don’t forget tail code when iteration number is not a multiple of unrolling degree. The tail code is in charge of dealing with the “remainder”

Loop unrolling - example

```
DO i=1, N
   DO j=1, N
      A(i)=A(i)+B(i,j)*C(j)
   END DO
END DO
```



Unrolled by 4

```
DO i=1, N, 4
   DO j=1, N
      A(i)  =A(i)  +B(i  ,j)*C(j)
      A(i+1)=A(i+1)+B(i+1,j)*C(j)
      A(i+2)=A(i+2)+B(i+2,j)*C(j)
      A(i+3)=A(i+3)+B(i+3,j)*C(j)
   END DO
END DO
```

unroll factor should match cache line size

Loop Unrolling (rule of thumb)

- The code generation / optimization strategy is completely dependent upon the number of iterations: if the number of iterations is systematically equal to 5, choose systematically full unrolling. If the number of iterations is large, then don't use full unrolling

- When deciding how many times to unroll a loop, a key consideration is how many instructions need to be in-flight to keep all the pipelines and execution units busy in every iteration. For example if there are two identical FP pipelines, each having a 5 cycle latency for the execution phase. This means that a total of ten floating point instructions need to be in flight in every loop iteration to keep both FP pipelines and the 5-stage FP adder busy on every cycle.

Loop unrolling (unwinding) example

- Original code 335 seconds runtime

```
SUBROUTINE MATMUL(C, leadc, A, B, leadb, n, o)

CT = 0.
BT = TRANSPOSE(B(1:n,:))
DO iNonZero = 1, A%nNonZero(n)
   i = A%NonZeroIndex1(iNonZero)
   j = A%NonZeroIndex2(iNonZero)
   CT(:,i) = CT(:,i) + BT(:,j)*A%NonZero(iNonZero)
ENDDO


C(1:n,:) = C(1:n,:) + TRANSPOSE(CT)
```

(:) implicit F90 loop over array dimensions, this case 9 iterations.

Loop unrolling (unwinding) example

- Unwinding code 165 seconds runtime

```
SUBROUTINE MATMUL(C, leadc, A, B, leadb, n, o)

CT = 0.
BT = TRANSPOSE(B(1:n,:))
DO iNonZero = 1, A%nNonZero(n)
   i = A%NonZeroIndex1(iNonZero)
   j = A%NonZeroIndex2(iNonZero)
   CT(1,i) = CT(1,i) + BT(1,j)*A%NonZero(iNonZero)
   .....
   CT(9,i) = CT(9,i) + BT(9,j)*A%NonZero(iNonZero)
ENDDO
C(1:n,:) = C(1:n,:) + TRANSPOSE(CT)
```

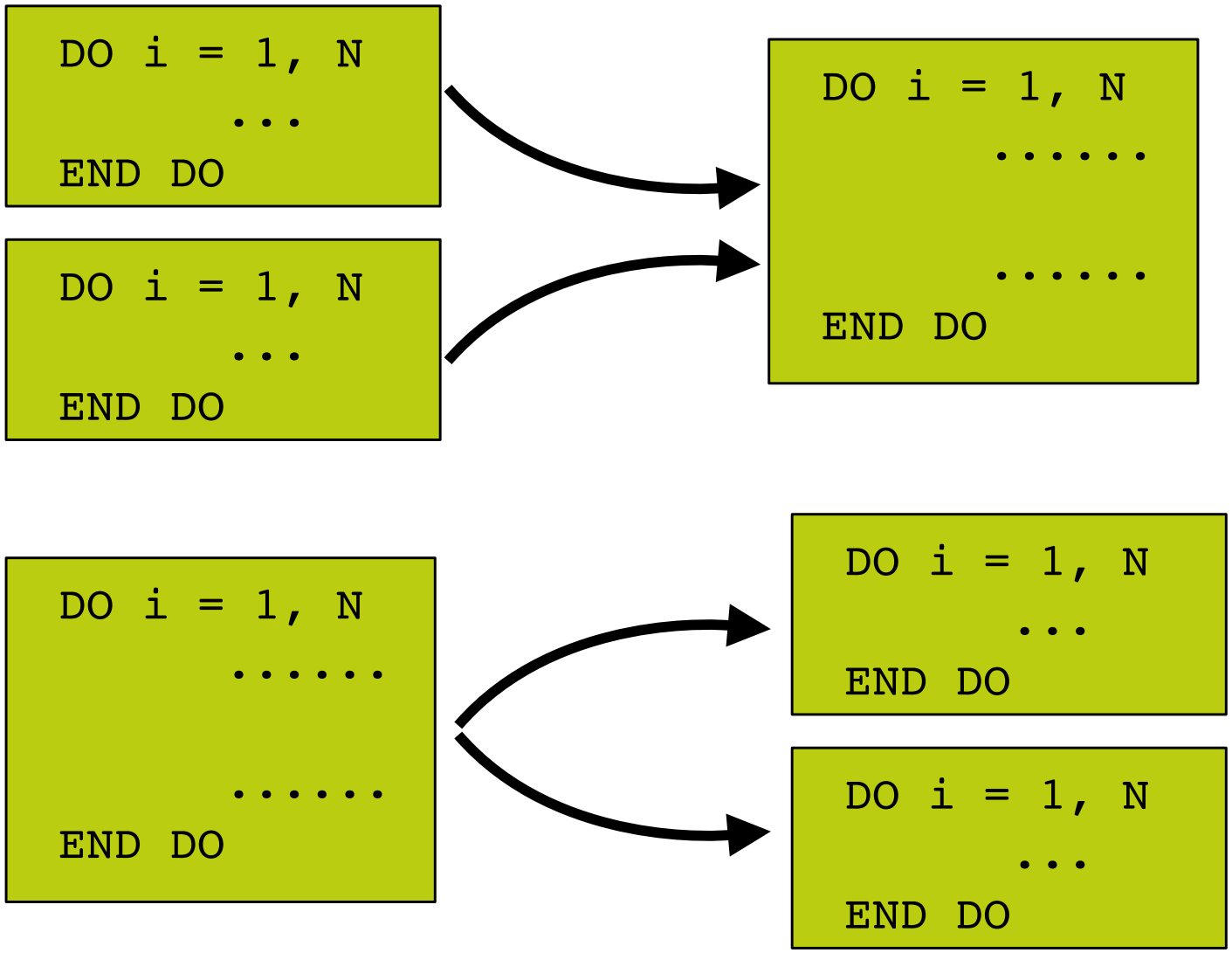
Loop unwinding and reuse example

```
grads(1:2,i) =&
& wchi0*( ws0*c_equ(30:31,is,  ichi) +
ws1*c_equ(30:31,is+1,  ichi) ) +&
& wchi1*( ws0*c_equ(30:31,is,ichi+1) +
ws1*c_equ(30:31,is+1,ichi+1) )

gradchi(1:2,i) =&
& wchi0*( ws0*c_equ(27:28,is,  ichi) +
ws1*c_equ(27:28,is+1,  ichi) ) +&
& wchi1*( ws0*c_equ(27:28,is,ichi+1) +
ws1*c_equ(27:28,is+1,ichi+1) )
```

Loop unwinding and reuse

```
wsc00 = wchi0*ws0
wsc01 = wchi0*ws1
wsc10 = wchi1*ws0
wsc11 = wchi1*ws1

grads(1,i) =&
& wsc00*c_equ(30,is,  ichi) + wsc01*c_equ(30,is+1,  ichi) +&
& wsc10*c_equ(30,is,ichi+1) + wsc11*c_equ(30,is+1,ichi+1)
grads(2,i) =&
& wsc00*c_equ(31,is,  ichi) + wsc01*c_equ(31,is+1,  ichi) +&
& wsc10*c_equ(31,is,ichi+1) + wsc11*c_equ(31,is+1,ichi+1)
```

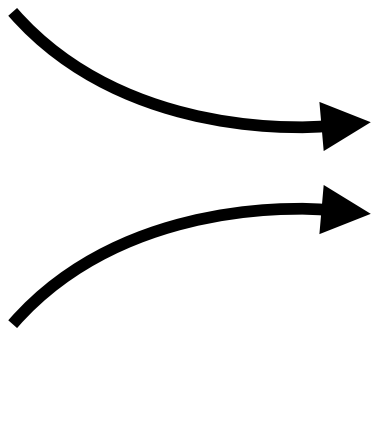
Loop Fusion and Fission



Loop Fusion

```
DO i = 1, N
    B(i)=2*A(i)
END DO
```

```
DO k = 1, N
    C(k)=B(k)+D(k)
END DO
```



```
DO ii = 1, N
    B(ii)=2*A(ii)
    C(ii)=B(ii)+D(ii)
END DO
```

```
DO ii = 1, N
    b=2*A(ii)
    C(ii)=b+D(ii)
END DO
```

- Advantage:
  - Re-usage of array B()
- Disadvantages:
  - in total now 4 arrays compete for cache size
  - more registers are needed

Optimisations: Loop Fusion

- Assume each element in `a` is 4 bytes, 32KB cache, 32 B / line

```
for (i = 0; i < 10000; i++)
  a[i] = 1 / a[i];
for (i = 0; i < 10000; i++)
  sum = sum + a[i];
```

  - First loop: hit 7/8 of iterations
  - Second loop: array > cache ⇒ same hit rate as in 1st loop

- Fuse the loops

```
for (i = 0; i < 10000; i++) {
  a[i] = 1 / a[i];
  sum = sum + a[i];
}
```

  - First line: hit 7/8 of iterations
  - Second line: hit all

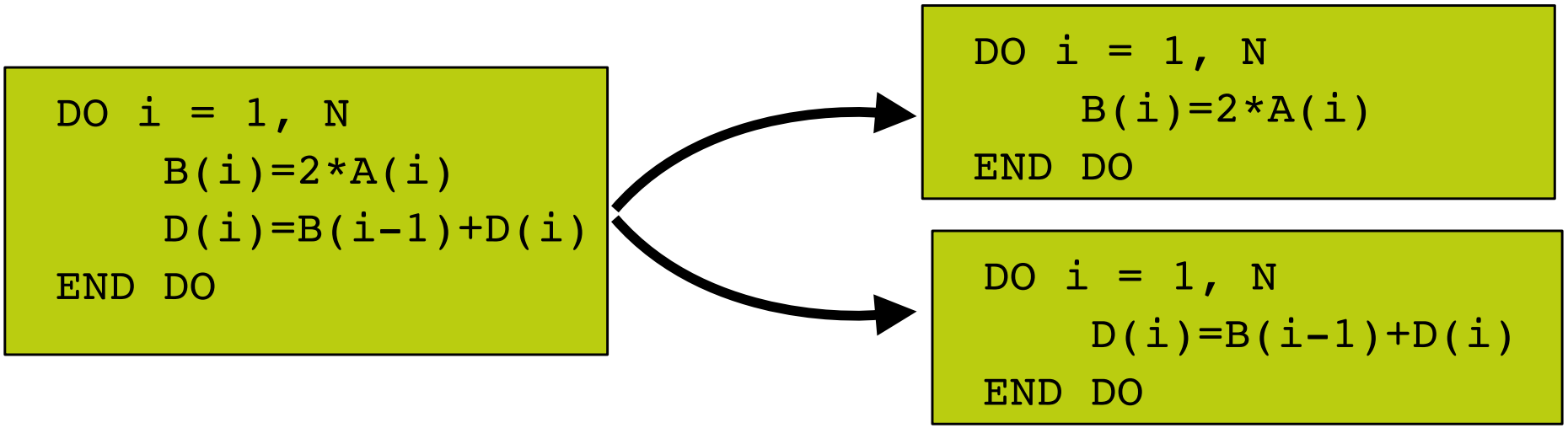
Optimisations: Merging Arrays

- Merge 2 arrays into a single array of compound elements

```
/* Before: two sequential arrays */
int val[SIZE];
int key[SIZE];

/* After: One array of structures */
struct merge {
   int val;
   int key;
} merged_array[SIZE];
```

- Reduce conflicts between **`val`** and **`key`**
- Improves spatial locality

Loop Fission



- Advantage:
  - First loop can be scheduled more efficiently and vectorized (SSE) and/or parallelized
  - More efficient use of cache
- Disadvantages:
  - additional loop created (minor disadvantage)

Loop peeling

- Peeling extracts one iteration of a loop

```
DO i=1, N
   A(i)=A(i)+B(i)
   B(i+1)=C(i)+D(i)
END DO
```

```
A(1) = A(1) + B(1)
DO i=2, N-1
   B(i+1)=C(i)+D(i)
   A(i+1)=A(i+1)+B(i+1)
END DO
B(N+1) = C(N) + D(N)
```

iterations can be computed in parallel

Procedure Inlining

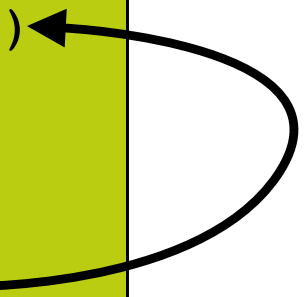
- Inlining: replace a function/subroutine call by the source code
- Advantages:
  - increases the opportunities for CPU optimisations
  - more opportunities for loop interchange
  - in crease parallelisation opportunities
  - reduces calling overhead (usually a minor advantage)

Procedure Inlining

- Candidates for inlining are modules that:
  - are 'small' and
  - are called very often and
  - do not take much time per call

- A typical example:

```
DO i = 1, N
    A(i)=2*B(i)
    call dowork(A(i), C(i))
END DO

SUBROUTINE dowork (x,y)
y = 1 + x*(1.0+x*0.5)
```



Direct Mapped Caches

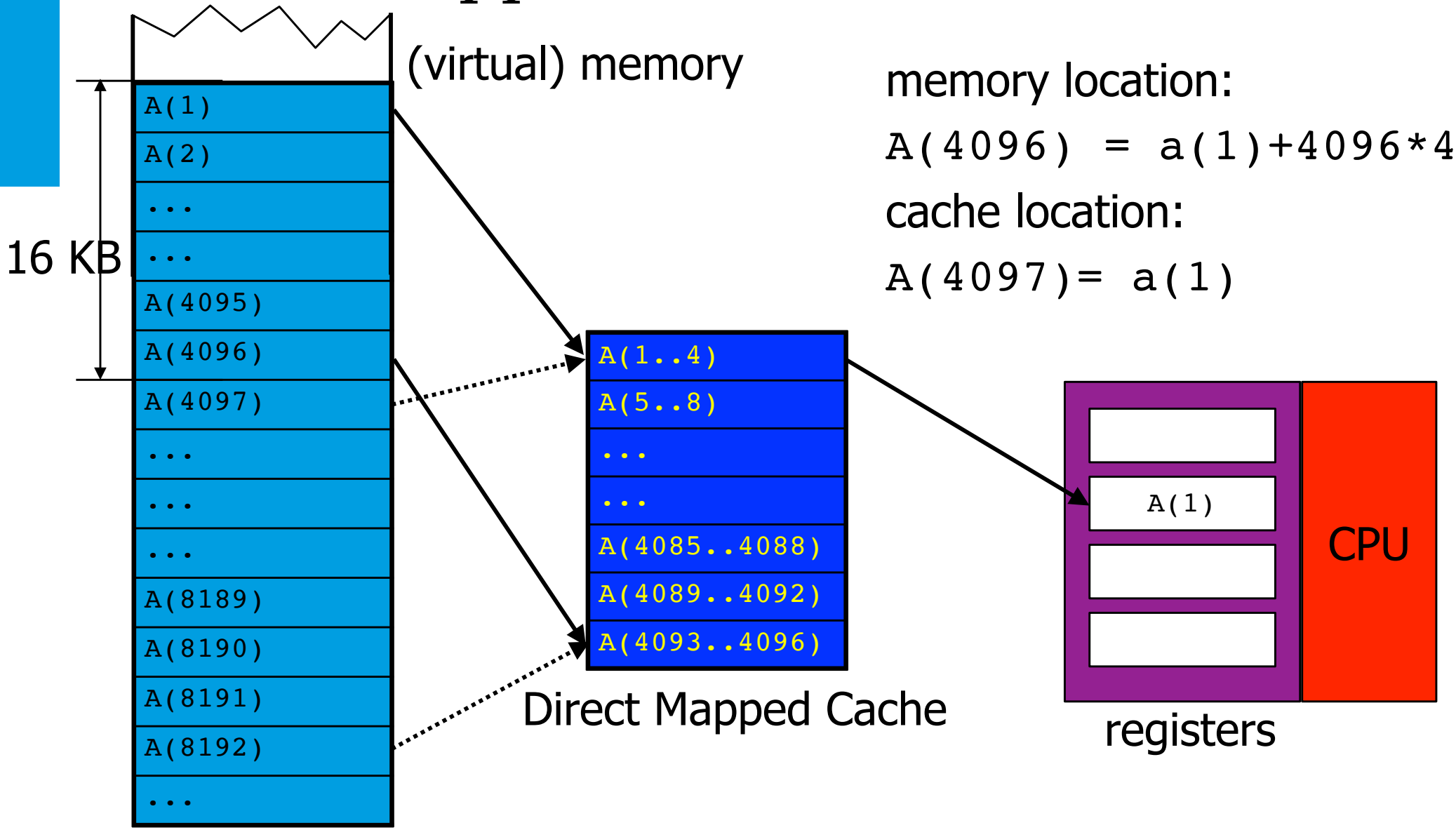
- Replacement Formula:

  cache location = memory address modulo size

An example:

- Assume that a cache line is 4 words (=16 Bytes)
- Cache size = 16 KB = 16*1024 Bytes
- This corresponds to 4*1024=4096 32-bit words
- 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(4096) = a(1)+4096*4`

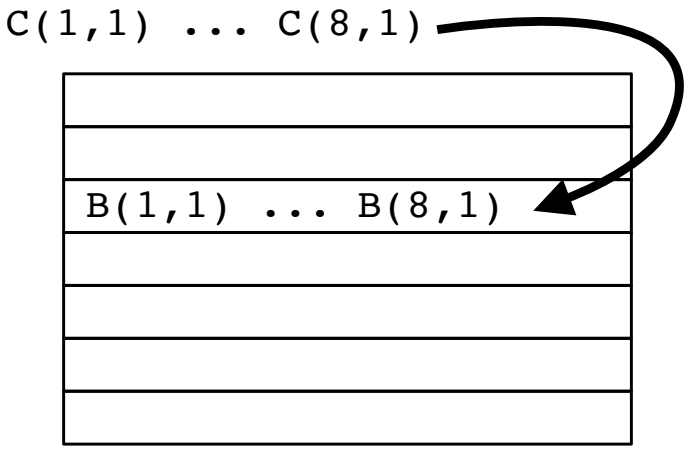
cache location:

`A(4097)= a(1)`

Array padding

- Out of the 3 references A(i,j), B(i,j) and C(i,j), two will map on the same cache line. This will cause cache-trashing:

```
 REAL A(1024,1024),
+     B(1024,1024),
+     C(1024,1024)

DO j=1,1024
   DO i=1,1024
      A(i,j)=A(i,j)+B(i,j)C(i,j)
   END DO
END DO
```

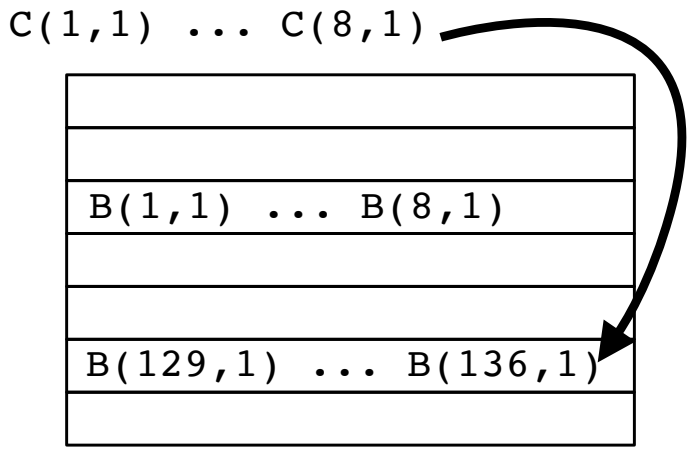


L1 cache (32 KB)
2-way set associative

C(1,1) = B(1,1) + 1024*1024*4
= B(1,1) + mod(32KB)

Array padding

```
 REAL A(1024,1024),PAD1(129),
+     B(1024,1024),PAD2(129).
+     C(1024,1024)

DO j=1,1024
   DO i=1,1024
      A(i,j)=A(i,j)+B(i,j)C(i,j)
   END DO
END DO
```



L1 cache (32 KB)
2-way set associative

padding will cause cache lines to be placed onto different cache locations

C(1,1) = B(1,1) + 1024*1024*4 +129*4
= B(129,1) + mod(32KB)

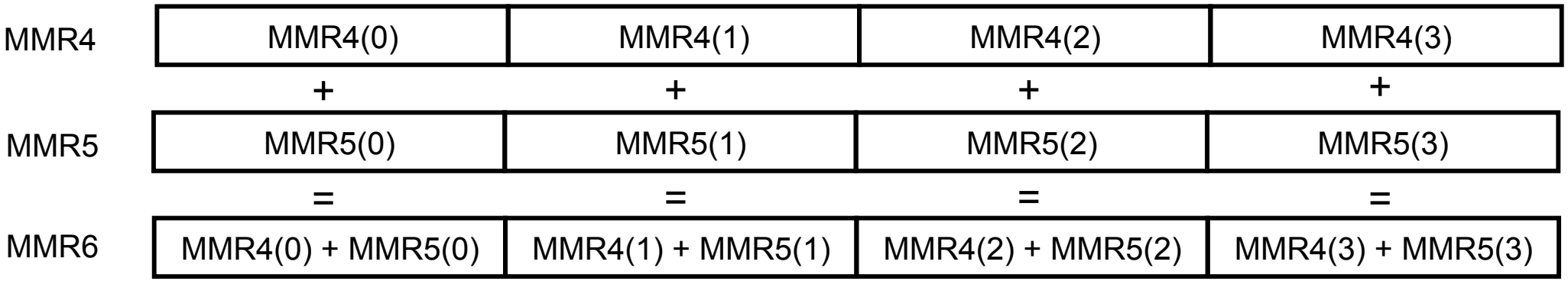
Vectorization: multimedia instructions

- Idea: provide special instructions to speedup multimedia code (don't forget multimedia = mass market)
- Very often multimedia code contains loops operating on arrays (each element of the array being accessed in turn) and perform the same operation.

Vectorization principle

Instead of operating on a single operand at a time, operate on a group of operands and for each element of the group perform the same (or similar) operation: SIMD Single Instruction Multiple Data

addps MMR6, MMR4, MMR5



Vectorization

- One instruction can handle multiple data objects.
- Loop example

```
for (j=0; j<nt; j++) {
    C[l] += A[j]*B[j+l];
}
```

- Intel compiler will vectorise this loop (-O3):

```
corrOpt.c(121): (col. 5) remark: LOOP WAS VECTORIZED.
```

- Remark: no array dependencies, only stride 1, and 'simple' loops,

Vectorization in i86_64

- Stride one memory accesses
- No IF tests branches: compilers are getting much better
- No subroutine calls
  - Inline
- What is size of loop
- Loop nest
  - Stride one on inside
  - Longest on the inside
- Unroll small loops
- Increase computational intensity
  - CU = (vector flops/number of memory accesses)

Vectorization example

```
s = 0.0;
for (k=0; k<Loop; k++){
    #pragma vector
    for (j=0; j<N; j++) {
        s += a[j]*b[j];
    }
}
```

```
icc -O3 -vec-report -c vect.c
vect.c(47): (col. 3) remark: LOOP WAS VECTORIZED.
```

Results

```
non-vector time=9.635107 gives 2125.560204 Mflop/s
loop used 1 cycles per instruction 1.176161
```

```
vector time=2.744634 gives 7461.832798 Mflop/s
loop used 0 cycles per instruction 0.335038
```

Example on git HPCourse/Vect

vector masking

```
DO i=1, N
   IF (A(i) > 0.0) THEN
      C(i)=A(i)+..
   ENDIF
ENDDO
```

```
DO i=1, N
   IF (A(i) > 0.0) THEN
      M(j)=i
      j = j +1
   ENDIF
ENDDO
```

```
DO i=1, N
   IF (A(i) > 0.0) M(i)=1
  ELSE M(i)=0
ENDDO
```

```
DO i=1, j
     C(M(i))=A(M(i))+..
ENDDO
```

```
DO i=1, N
     C(i)=M(i)*(A(i)+..)
ENDDO
```

vectorized intrinsics

- There are vectorized version of sin, cos, ... in Intel's MKL and AMD ACML libraries
  - for example: sinv, cosv,

Eliminate constant expressions from loops

```
pi = 3.14159265358979d0
a = 2.0 * pi + 3.0
do k = 1, 1000000
   x(k) = (2.0 * pi + 3.0) * y(k)
end do

b = exp(2)
do k = 1, 1000000
   x(k) = exp(2) * y(k)
end do

do k = 1, 1000000
   x(k) = my_func(2.0)* y(k)
end do
```

Prefetching

- Normal mode: fetching data is done when it is asked for.
- Prefetching is key to hide latency
  - Prefetch data that will be used later on
  - Prefetch sufficiently ahead of time: if not, latency will be only partially hidden
  - Avoid that data prefetched data displace useful data : cache pollution
- WARNING: prefetching is not free of charge, it consumes memory bandwidth which is a critical resource.
- There is a limit in how many outstanding memory references a CPU can handle (32-48 range)

Remember Exercise 1….

- Some instructions take much longer than others


avoid using instructions which take many cycles.

Exercise: 1 Cycles

- Counting cycles of basic operations; addition, multiplication, division, ...
- On your git clone: cd HPCourse/Cycles
- Check the README for instructions.
- Links:
  http://en.wikipedia.org/wiki/Streaming_SIMD_Extensions

Cycles results

CPU	mul	addmul	unrol	div	pow	mul	sin
core 2 Duo 2.5 GHz	2	1.5	1.5	12	93	2	45
Pentium 4 2.8 GHz	3	2	1	45	3	3	184
core 2 Duo 2.3 GHz	3	1	1	37	3	3	90
Opteron 2.2 GHz	3	2	1	22	154	3	137
Magny-Cours 2.0 GHz	3	1	1	12	5	2	44
Interlagos 2.3 GHz	3	2	1	9	5	3	93
Core i7 2.9 GHz	3	1	1	11	3	2	28
Haswell 2.0 GHz 4c	1	1	1	4	47	1	15
apple M1	1	0.5	0.5	1	25	1	7

blue: cygwin result

Programming Tricks for lower cycles

- division is an expensive (takes a lot of cycles) operation: try to replace it with a multiplication with its reciprocal

  `(a[i+1]-a[i-1])/(2*h) => (a[i+1]-a[i-1])*hi (with hi=1.0/(2*h) defined outside the loop)`

- integer powers can be better replaced with multiplication's

  `pow(a,2) => a*a`

  `pow(a,1.5) => a*sqrt(a)`

- CPU can do a mult and add at the same cycle

  `a[i] = b[i]*c[i]` costs the same time as

  `a[i] = c[i] + b[i]*c[i]`

Programming Tricks for lower cycles

- sin and similar functions take a lot of time. Store pre-computed sin function in a table and read from that table instead of using the sin function:

```
static float sintab[500]=
    {
                0.000000,
                0.012566,
                0.025130,
                0.037690,
                0.050244,
                0.062791,
                0.075327,
                ......
```

```
k=2*PI*arg/500 (or better k=arg*PI*0.004)
```

Instruction Cycles

Table 1: AMD Athlon™ Processor Floating Point Engine Features/Latency/Throughput Comparison

	AMD Seventh Generation	Intel Previous Generation
Feature	**AMD Athlon™ Processor**	**Intel Pentium® III and Pentium III Xeon**
No. of FP execution pipelines	3	1
Latency / Throughput Rates for Classes of FP Instructions*		
❑ FADD reg	4/1	3/1
❑ FMUL reg	4/1	5/2
❑ FDIV reg (single)	16/13	17/17
❑ FDIV reg (double)	20/17	32/32
❑ FDIV reg (extended)	24/21	37/37
❑ FSQRT (single)	19/16	28/28
❑ FSQRT (double)	27/24	57/57
❑ FSQRT (extended)	35/31	68/68
❑ FCOM	2/1	1/1

* Latency/throughput rates are measured in processor clocks; lower numbers mean higher throughput.

Intel Instruction set

Table C-15. x87 Floating-point Instructions

Instruction	Latency[1]		Throughput		Execution Unit [2]
CPUID	0F_3H	0F_2H	0F_3H	0F_2H	0F_2H
FABS	3	2	1	1	FP_MISC
FADD	6	5	1	1	FP_ADD
FSUB	6	5	1	1	FP_ADD
FMUL	8	7	2	2	FP_MUL
FCOM	3	2	1	1	FP_MISC
FCHS	3	2	1	1	FP_MISC
FDIV Single Precision	30	23	30	23	FP_DIV
FDIV Double Precision	40	38	40	38	FP_DIV
FDIV Extended Precision	44	43	44	43	FP_DIV
FSQRT SP	30	23	30	23	FP_DIV
FSQRT DP	40	38	40	38	FP_DIV
FSQRT EP	44	43	44	43	FP_DIV
F2XM1[4]	100-200	90-150		60	
FCOS[4]	180-280	190-240		130	
FPATAN[4]	220-300	150-300		140	
FPTAN[4]	240-300	225-250		170	
FSIN[4]	160-200	160-180		130	
FSINCOS[4]	170-250	160-220		140	

Things not to do: Generality

- Go against generality (against sound principles of software engineering)

```
C STANDARD DAXPY CODE
DO I = 1, N
 Y(INCY*I) = Y(INCY*I) + a * X(INCX*I)
ENDDO
```

Notice that very often, `INCX = INCY = 1` generate a specific version for this very common case.

Advantages:

- almost always usable

- Drawbacks:
  - increase code size, too many specialized versions.

Be careful with F90/C++ matrix ops

```
! write out matrix loops
      tmpT=0.0
     auxMatrix=0.0
     DO i= 1, n1
      DO j = 1, n1
       DO k = 1, n1
          tmpT(j,1) = tmpT(j,1)+(locA(j,k)+locabsA(j,k))*iT(k,i)
       ENDDO
      ENDDO
      DO j = 1, n1
       DO k = 1, n1
        auxMatrix(j,i) = auxMatrix(j,i)+ T(j,k)*tmpT(k,1)
       ENDDO
      ENDDO
      DO j = 1, n1
       auxMatrix(j,i) = auxMatrix(j,i)*geoSurfaces(iSide,iElem)
      ENDDO
     ENDDO

!Original code    auxMatrix =
MATMUL( T,MATMUL(locA+locabsA,iT) )*geoSurfaces(iSide,iElem)
```

Help the Compiler

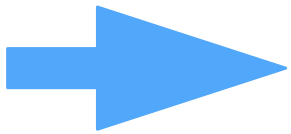
- Write simple code
- Use compiler flags to specify some optimisations to be performed: force vectorization, unrolling, etc…
- Use pragmas inserted in the code: can specify precisely optimisation scope (a given loop level)
- Unroll and Unroll and Jam: avoid complex/long code generated by hand unrolling
- No alias: informs the compiler that the memory regions are disjoints: allows load and store reordering
- Use libraries but check performance first and don’t use a matrix multiply (N1 x N2) (N2 x N3) to perform a dot product (1 x N) (N x 1)
- Use state of the art compiler: use Intel, PGI, gcc

Blocking for Cache

- An optimization that applies for datasets that do not entirely fit in the data cache
- A way to increase spatial locality or reference: exploit full cache lines
- A way to increase temporal locality or reference: improves full data re-usage

1D blocking: strip-mining

```
do i = 1, n
  do j = 1, m
    c = c + a(i) * b(j)
  enddo
enddo
```



```
do jout = 1, m, block
  do i = 1, n
    do j = jout, jout+block
      c = c + a(i) * b(j)
    enddo
  enddo
enddo
```

Design subset b(jout:jout+block) such that it fits into the largest level of cache on the processor

The number of values from main memory is:

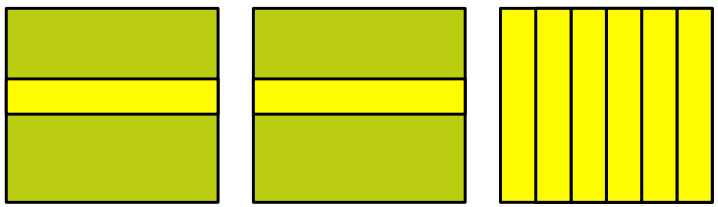
(m/block) * (n+block) ~ m*n/block.

So, the total amount of data from main memory to the processor has been reduced by the size of block.

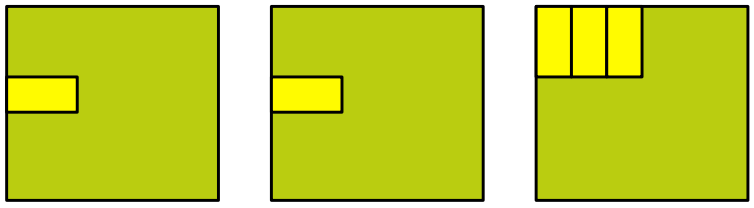
Example: cache blocking

- Re-organise data accesses so that a piece of data is used a number of times before moving on… in other words, artificially create temporal locality

```
for (i=0;i<N;i++)
   for (j=0;j<N;j++) {
      r=0;
      for (k=0;k<N;k++)
         r = r + y[i][k] * z[k][j];
      x[i][j] = r;
   }
```

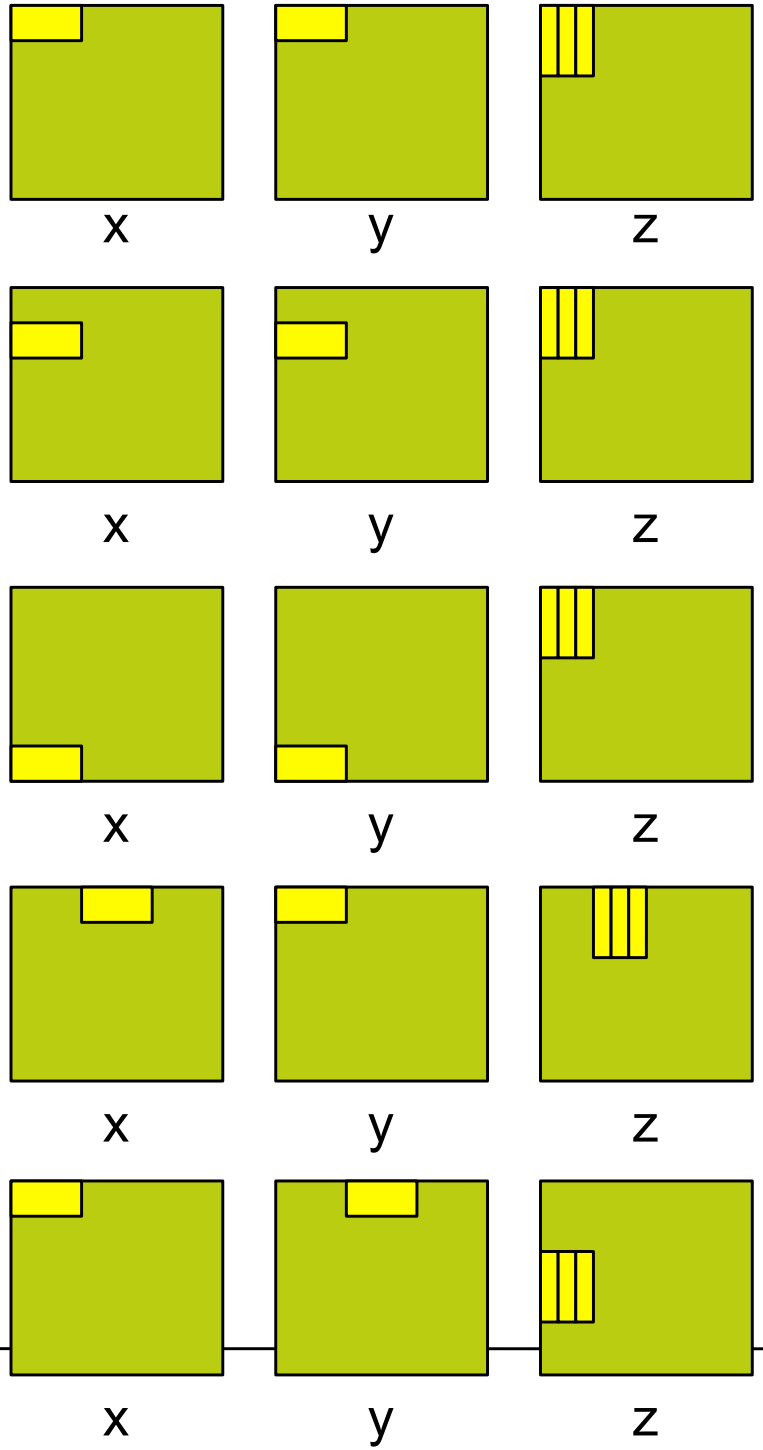



```
for (jj=0; jj<N; jj+= B)
for (kk=0; kk<N; kk+= B)
for (i=0;i<N;i++)
   for (j=jj; j< min(jj+B,N); j++) {
      r=0;
      for (k=kk; k< min(kk+B,N); k++)
         r = r + y[i][k] * z[k][j];
      x[i][j] = x[i][j] + r;
   }
```



Example

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

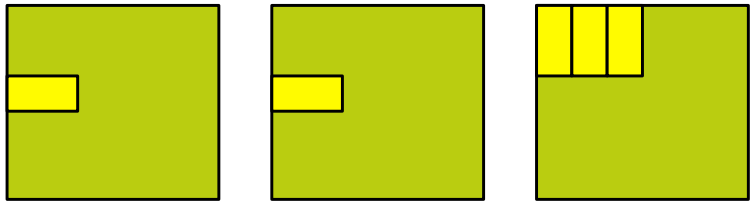


Example

- Original code could have $2N^3 + N^2$ memory accesses, while the new version has $2N^3/B + N^2$

```
for (i=0;i<N;i++)
   for (j=0;j<N;j++) {
      r=0;
      for (k=0;k<N;k++)
         r = r + y[i][k] * z[k][j];
      x[i][j] = r;
   }
```

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



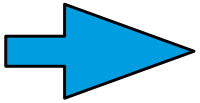
Blocking for cache

- Blocking is very powerful and simple to do (local impact only)

  BUT

- The resulting code does not get any prettier ....

```
DO i = 1, N
      ...
END DO
```



```
DO i1 = 1, N, NBlocking
   DO i2 = 0, MIN(N-i1, NBlocking)-1
      ...i = i1 + i2 ....
   END DO
END DO
```

Blocking for Cache

- Many ways to block any given loop nest
  - Which loops get blocked, inner, outer, all, ...?
  - What block size(s) to use?

  Analysis can reveal which ways are beneficial

  But trial-and-error is probably faster

Goto PPT presentation of Steve

Optimization Benefits

Technique	Instruction	Memory
Loop Interchange	+	++
Loop Unrolling	++	+
Loop Blocking	-	++
Loop Fusion	+	++
Loop Fission	+	++
Inlining	++	++
Array padding	0	++

Branches

Conditional branches

```
DO I = 1, 100
 IF (A(I) > 0) THEN B(I) = C(I) (statement S1(I))
         ELSE F(I) = G(I) (statement S2(I))
 ENDIF
ENDDO
```

Basically, you need to know the outcome of the test before proceeding with execution of S1(I) or S2(I)

- Solution: the hardware will bet on the result on the test and decide to execute ahead one of the two outcomes of the branch (branch prediction). The hardware will use the past to predict the future.
- Drawback: if the hardware has miss-predicted, recovery might be costly
- Branch predictors can reach up to 97% correct predictions (and even more) success because …. There are many loops
  'Real' Solution: avoid branches in your loops

The use of combined conditions

- In the most active areas of a program, order the expressions in compound branch conditions to take advantage of short circuiting of compound conditional expressions. In this way subsequent operands are not evaluated at all.

- Example: in a bunch of AND (&&) conditions place the one which will likely be **false** as the first (left to right) operand. This will then terminate the evaluation.

- Example in a bunch of OR (||) conditions place the one which will likely be **true** as the first (left to right) operand. This will then terminate the rest of the evaluation.

- C and C++ compilers guarantee short-circuit evaluation of the boolean operators && and ||.

Branches and loops

Avoid

```
for (i...) {
   if (CONSTANT0) {
      DoWork0(i); // Does not affect CONSTANT0.
   }
   else {
      DoWork1(i); // Does not affect CONSTANT0.
    }
}
```

Preferred

```
if (CONSTANT0) {
   for (i...) DoWork0(i);
else {
    for (i...) DoWork1(i);
}
```

GNU's gcov tool

- Counts the number of times a code line is executed
- Very useful to find out how much a certain branch is taken

Steps:

- compile the source files you want to investigate with -fprofile-arcs -ftest-coverage and link with -fprofile-arcs
- run the executable. After the run two additional files, for every profiled source code, will be found in the object directory:
  - imd_forces_nbl.gcno
  - imd_forces_nbl.gcda

  The first file is produced during compilation and the second file after running the code.

  The files are analyzed with gcov
  gcov -b gives branch-probabilities
  gcov -c gives the number of times a branch in taken

Example gcov usage

```
923658657792:  330:              if (r2 <= pair_pot.end[col]) {
     branch  0 taken 91% (fallthrough)
     branch  1 taken 9%
        -:  331:
        -:  332:
839358310912:  333:                  r2a = r2 - pair_pot.begin[col];
839358310912:  334:                  istep = pair_pot.invstep[col];
        -:  335:
839358310912:  336:                  if (r2a < 0) {
     branch  0 taken 0% (fallthrough)
     branch  1 taken 100%
    #####:  337:                      p0 = pair_pot.table[col];
    #####:  338:                      p1 = pair_pot.table[col+4];
    #####:  339:                      p2 = pair_pot.table[col+8];
        -:  340:
    #####:  341:                      dv = p1 - p0;
    #####:  342:                      d2v = p2 - 2.0 * p1 + p0;
        -:  343:                      pot = p0;
    #####:  344:                      grad = 2.0 * istep * (dv - 0.5 * d2v);
        -:  345:                  }
        -:  346:                  else {
839358310912:  347:                      r2a = r2a * istep;
839358310912:  348:                      l = r2a;
839358310912:  349:                      chi = r2a - l;
839358310912:  350:                      chi2 = 0.5 * chi * (chi - 1);
        -:  351:
839358310912:  352:                      p0 = pair_pot.table[col+l*4];
839358310912:  353:                      p1 = pair_pot.table[col+l*4+4];
839358310912:  354:                      p2 = pair_pot.table[col+l*4+8];
        -:  355:
839358310912:  356:                      dv = p1 - p0;
839358310912:  357:                      d2v = p2 - 2 * p1 + p0;
839358310912:  358:                      pot = p0 + chi * dv +  chi2 * d2v;
839358310912:  359:                      grad = 2 * istep * (dv + (chi - 0.5) *
d2v);
        -:  360:                  }
```

Inlining

subroutine/function calls are also branches

- Solution = inlining: at the call site, insert the subroutine
  - Advantages of inlining: increase basic block size, allows specialization
  - Drawbacks of inlining: increase of code size

Examples

- Test machine (this laptop):

- ARM M1 600-3.220 GHz
- L1-cache 128 KB, x-way, 64 byte line size, write-back
- L2-cache 24 MB z-way, 64 byte line
- no L3-cache?

Example: Matrix Multiply

- R = A B

```
#define ROWS      1020 // Number of rows in matrix
#define COLUMNS   1020 // Number of columns in matrix
```

- On git clone: cd HPCourse/MatMul

(a)

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

$$b_0 = \overset{0}{\boxed{a_{00}x_0}} + \overset{4}{\boxed{a_{01}x_1}} + \overset{6}{\boxed{a_{02}x_2}} + \overset{5}{\boxed{a_{03}x_3}}$$

$$b_1 = \overset{7}{\boxed{a_{10}x_0}} + \overset{11}{\boxed{a_{11}x_1}} + \overset{13}{\boxed{a_{12}x_2}} + \overset{12}{\boxed{a_{13}x_3}}$$

(b)

$$b_2 = \overset{14}{\boxed{a_{20}x_0}} + \overset{18}{\boxed{a_{21}x_1}} + \overset{20}{\boxed{a_{22}x_2}} + \overset{19}{\boxed{a_{23}x_3}}$$

$$b_3 = \overset{21}{\boxed{a_{30}x_0}} + \overset{25}{\boxed{a_{31}x_1}} + \overset{27}{\boxed{a_{32}x_2}} + \overset{26}{\boxed{a_{33}x_3}}$$

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

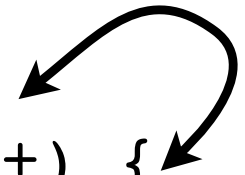
Straight Forward Implementation (1)

```
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 ;
    }
}
```

50% more computational work than on previous slide

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?

Results Intel i7:

straight_mask(0) = 1.160705
straightforward(1) = 1.410938
loop interchange(2a) = 0.105770
zero masks(2b) = 0.659546
mask mult(2c) = 0.179615
BLAS sgemm(3) = 0.048067

Exercise 6 - Matrix multiplication

ROWS 1020;COLUMNS 1020

Time multiply_matrices 0 =0.910852

Time multiply_matrices 1 =1.695003

Time multiply_matrices 2 =0.224845

Time multiply_matrices 2b =0.113121

Time multiply_matrices 2c =0.210068

Time multiply_matrices 3 =0.031279

ROWS 1024;COLUMNS 1024

Time multiply_matrices 0 =1.263933

Time multiply_matrices 1 =6.973396

Time multiply_matrices 2 =0.249361

Time multiply_matrices 2b =0.120076

Time multiply_matrices 2c =0.214714

Time multiply_matrices 3 =0.014480

size 1020 en -O3:

Time multiply_matrices 0 =1.916440
Time multiply_matrices 1 =1.402665
Time multiply_matrices 2 =0.290312
Time multiply_matrices 2b =0.143238
Time multiply_matrices 2c =0.311288
Time multiply_matrices 3 =0.021026

size 1024 en -O3:

Time multiply_matrices 0 =2.121419
Time multiply_matrices 1 =1.427654
Time multiply_matrices 2 =0.288243
Time multiply_matrices 2b =0.151350
Time multiply_matrices 2c =0.313108
Time multiply_matrices 3 =0.018369

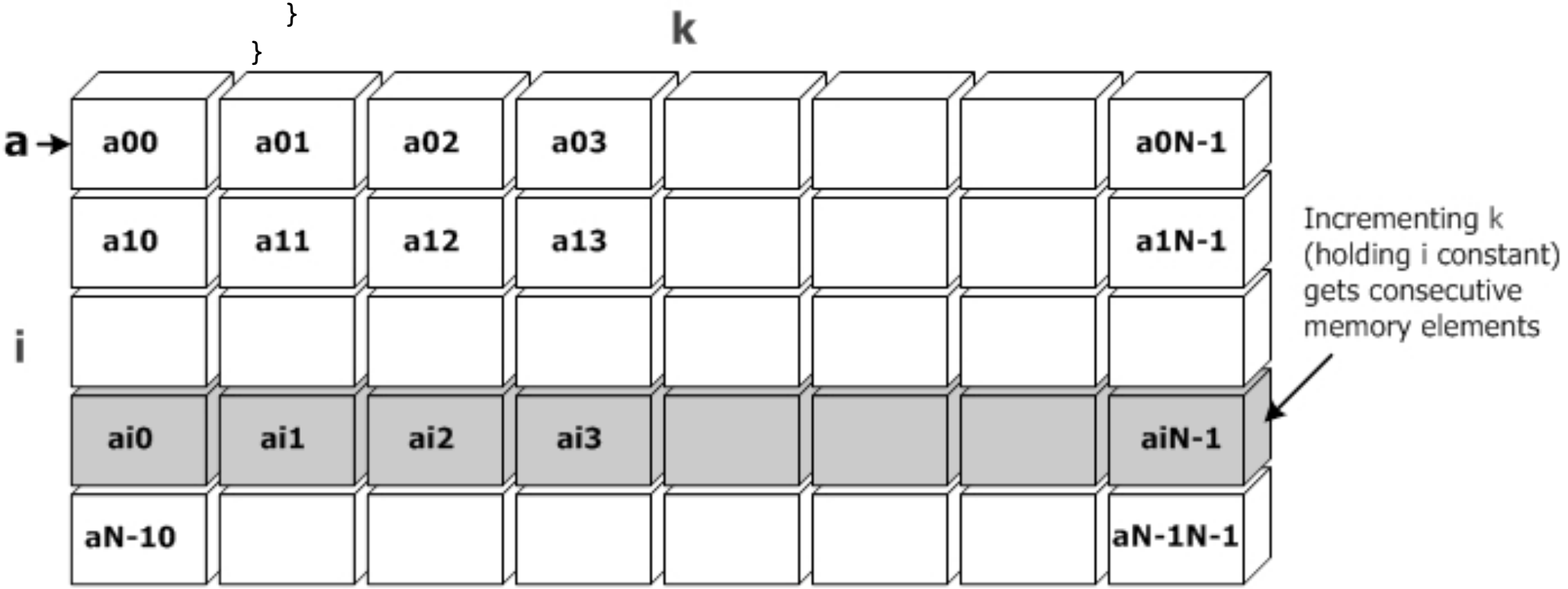
1020 vs 1024 sizes

```
-bash-3.00$ ./ClassicMatrixMultiply
Time multiply_matrices 0 =5.075668
Time multiply_matrices 1 =5.002930
Time multiply_matrices 2 =0.120282
Time multiply_matrices 2b =0.745493
Time multiply_matrices 2c =0.183339
Time multiply_matrices 3 =0.049466
```

```
-bash-3.00$ ./ClassicMatrixMultiply
Time multiply_matrices 0 =1.160705
Time multiply_matrices 1 =1.410938
Time multiply_matrices 2 =0.105770
Time multiply_matrices 2b =0.659546
Time multiply_matrices 2c =0.179615
Time multiply_matrices 3 =0.048067
```

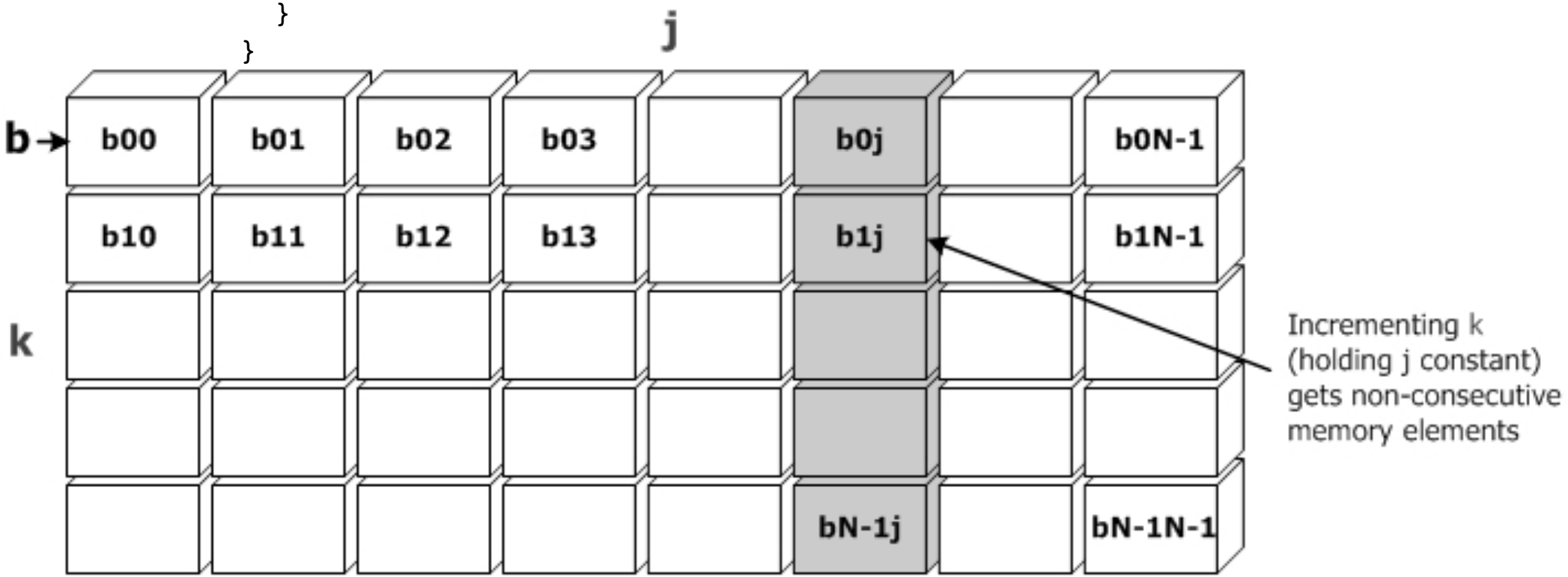
cache-access (1)

```
// Non-Unit Stride access problem with b[k][j]
     for(i=0;i<NUM;i++) {
       for(j=0;j<NUM;j++) {
         for(k=0;k<NUM;k++) {
           r[i][j] =r[i][j] + a[i][k] * b[k][j];
         }
       }
     }
```



cache-access (2)

```
// Non-Unit Stride access problem with b[k][j]
     for(i=0;i<NUM;i++) {
       for(j=0;j<NUM;j++) {
         for(k=0;k<NUM;k++) {
           r[i][j] =r[i][j] + a[i][k] * b[k][j];
         }
       }
     }
```

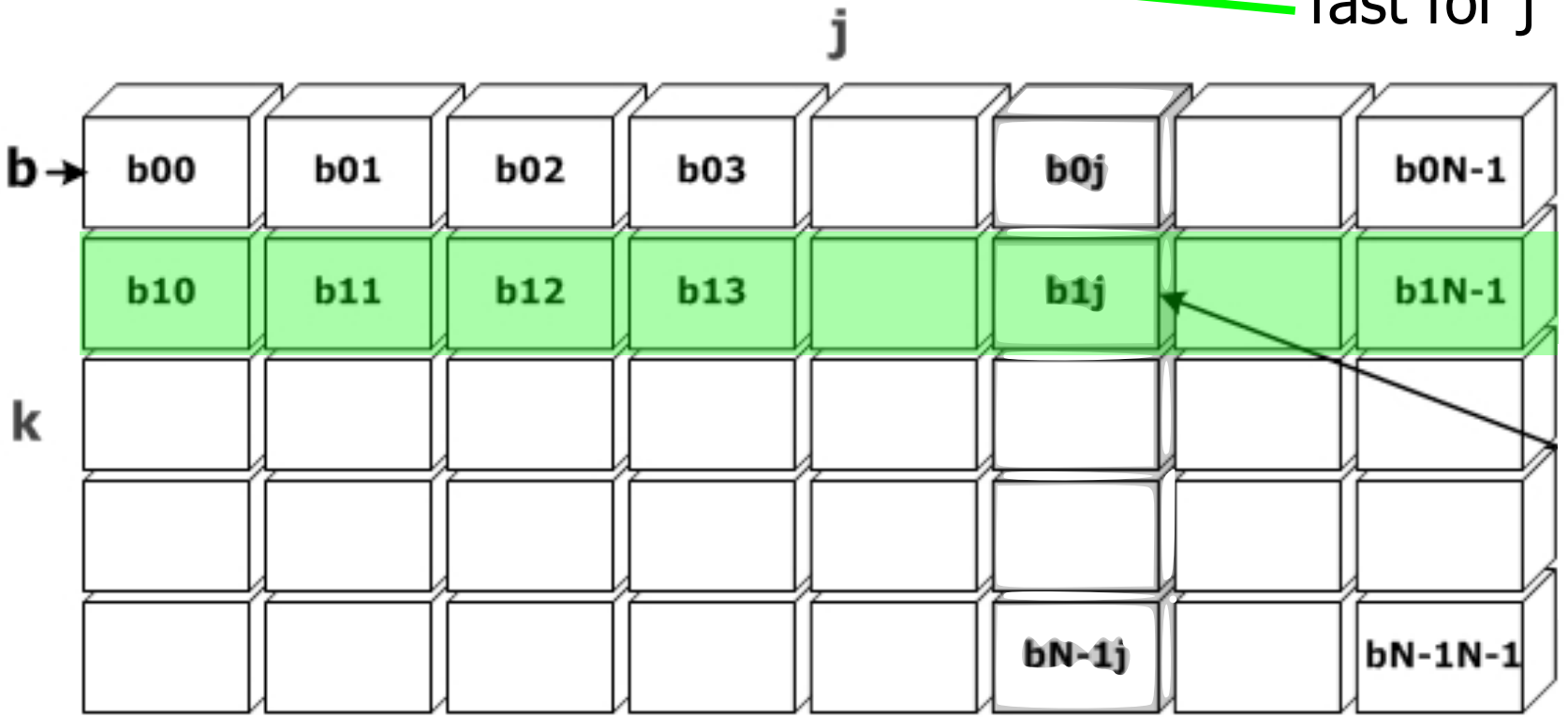


cache-access (2)

```
// After loop interchange of k and j loops
     for(i=0;i<NUM;i++) {
       for(k=0;k<NUM;k++) {
         for(j=0;j<NUM;j++) {
           r[i][j] =r[i][j] + a[i][k] * b[k][j];
         }
```

constant for j

fast for j



incrementing j (holding k constant) get consecutive memory elements

Matrix Multiply observations

- Consider changing loops for better cache and memory access
- If possible use basis BLAS and LAPACK functions for matrix-vector operations
- Using Mask array with an if statement should be avoided
- Multiplication with a mask array performs much better

Matrix multiply & unrolling inner

```
for (i = 0; i < ROWS; i++) {
    for (k = 0; k < COLUMNS; k++) {
        tmp = a[i][k];
        for (j = 0; j < COLUMNS; j+=4) {
            r[i][j+0] = r[i][j+0] + tmp*b[k][j+0];
            r[i][j+1] = r[i][j+1] + tmp*b[k][j+1];
            r[i][j+2] = r[i][j+2] + tmp*b[k][j+2];
            r[i][j+3] = r[i][j+3] + tmp*b[k][j+3];
        }
    }
}
```

reuse of tmp

Matrix multiply & unrolling middle

```
for (i = 0; i < ROWS; i++) {
    for (k = 0; k < COLUMNS; k+=4) {
        tmp0 = a[i][k+0]; tmp1 = a[i][k+1];
        tmp2 = a[i][k+2]; tmp3 = a[i][k+3];
        for (j = 0; j < COLUMNS; j++) {
            r[i][j] = r[i][j] + tmp0*b[k+0][j]
                              + tmp1*b[k+1][j]
                              + tmp2*b[k+2][j]
                              + tmp3*b[k+3][j];
        }
    }
}
```

less load-stores on r[i][j]

Matrix multiply & unrolling outer

```
for (i = 0; i < ROWS; i+=4) {
    for (k = 0; k < COLUMNS; k++) {
        tmp0 = a[i+0][k]; tmp1 = a[i+1][k];
        tmp2 = a[i+2][k]; tmp3 = a[i+3][k];
        for (j = 0; j < COLUMNS; j++) {
            r[i+0][j] = r[i+0][j] + tmp0*b[k][j];
            r[i+1][j] = r[i+1][j] + tmp1*b[k][j];
            r[i+2][j] = r[i+2][j] + tmp2*b[k][j];
            r[i+3][j] = r[i+3][j] + tmp3*b[k][j];
        }
    }
}
```

reuse of b[k][j] and tmp0-3

Matrix multiply & unrolling results

GNU compiler:

Time multiply_matrices 2 =0.907248
Time multiply_matrices Unroll 1 =1.147101
Time multiply_matrices Unroll 2 =0.997442
Time multiply_matrices Unroll 3 =0.548240
Time multiply_matrices 3 =0.075967

Intel compiler

Time multiply_matrices 2 =0.182839
Time multiply_matrices Unroll 1 =1.214950
Time multiply_matrices Unroll 2 =0.260684
Time multiply_matrices Unroll 3 =0.371170
Time multiply_matrices 3 =0.081323

Matrix Blocking

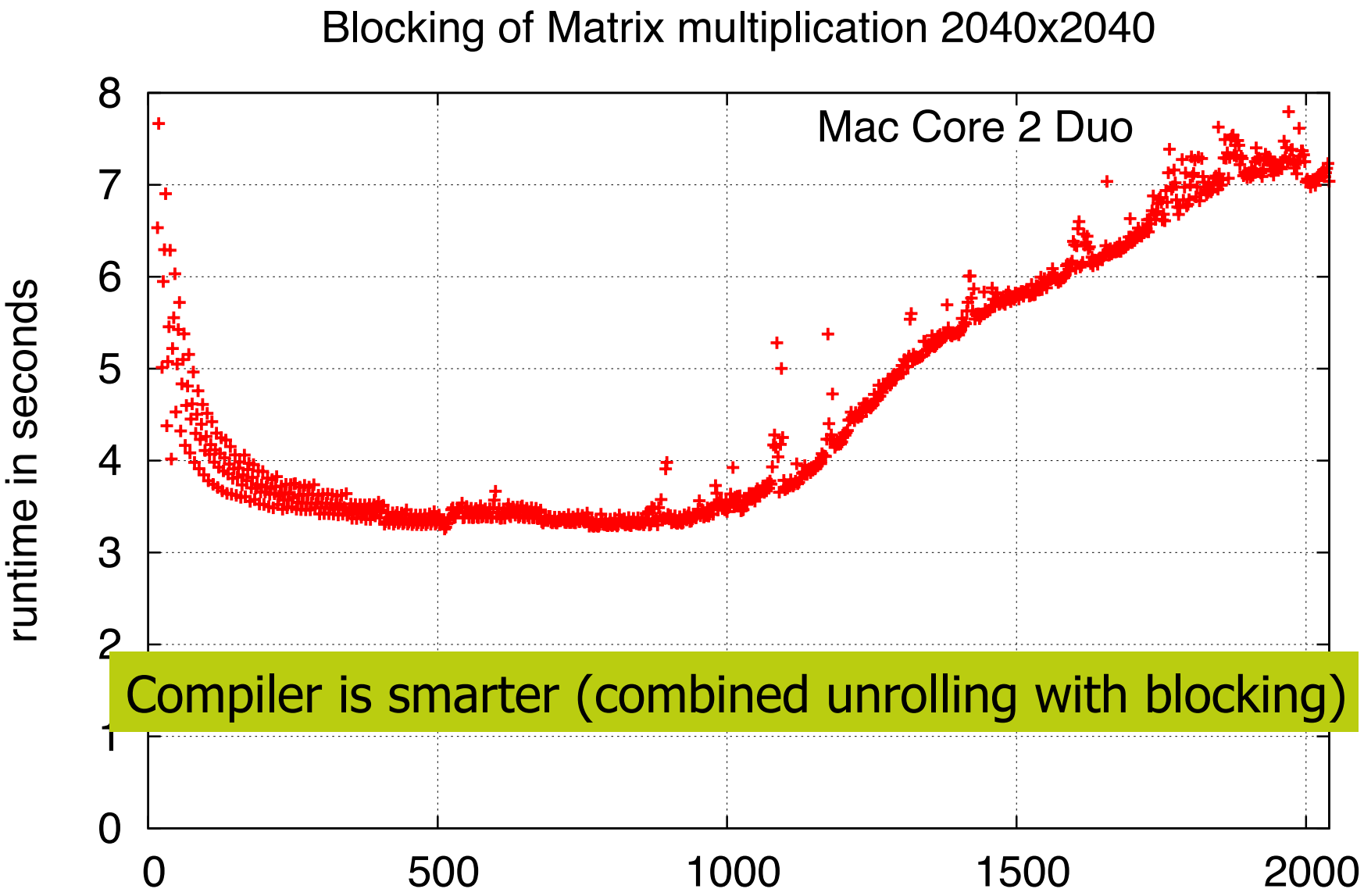
```
for (jb = 0; jb < COLUMNS; jb+=bin) {
for (kb = 0; kb < COLUMNS; kb+=bmid) {
for (ib = 0; ib < ROWS; ib+=bout) {
 for (i = ib; i < MIN(ROWS,ib+bout); i++) {
  for (k = kb; k < MIN(COLUMNS,kb+bmid); k++) {
   for (j = jb; j < MIN(COLUMNS,jb+bin); j++) {
     r[i][j] = r[i][j] + a[i][k] * b[k][j];
   }
  }
 }
}
}
}
```

Matrix Blocking 2040x2040 (50 MB)

- loop interchange time =1.379454
- loop unroll time = 2.775398
- sgemm time =0.471136

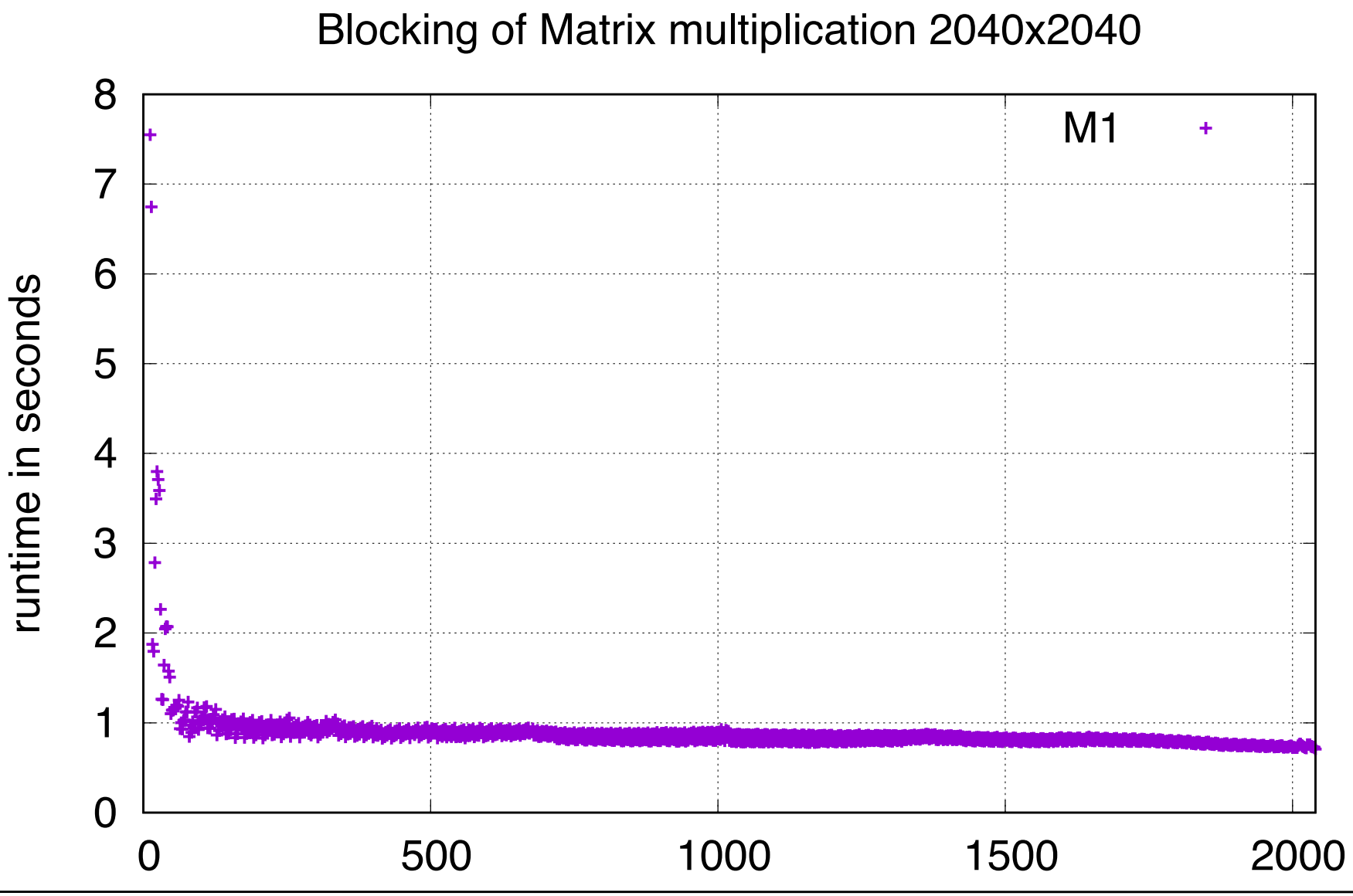
- L2 cache = 6 MB
- blocksize = sqrt(L2/(3*4)) = 724 optimal blocking size.

Results



Compiler is smarter (combined unrolling with blocking)

Results M1


Blocking of Matrix multiplication 2040x2040
M1
runtime in seconds
block size
0
1
2
3
4
5
6
7
8
0
500
1000
1500
2000

Example: Correlation

$$c_{ab}[i\Delta t] = \sum_{k=0}^{N_w-1} \sum_{j=kN_k}^{(k+1)N_k-1} a[j\Delta t]b[(j+i)\Delta t]$$

$$i \in [0, N_c - 1]$$

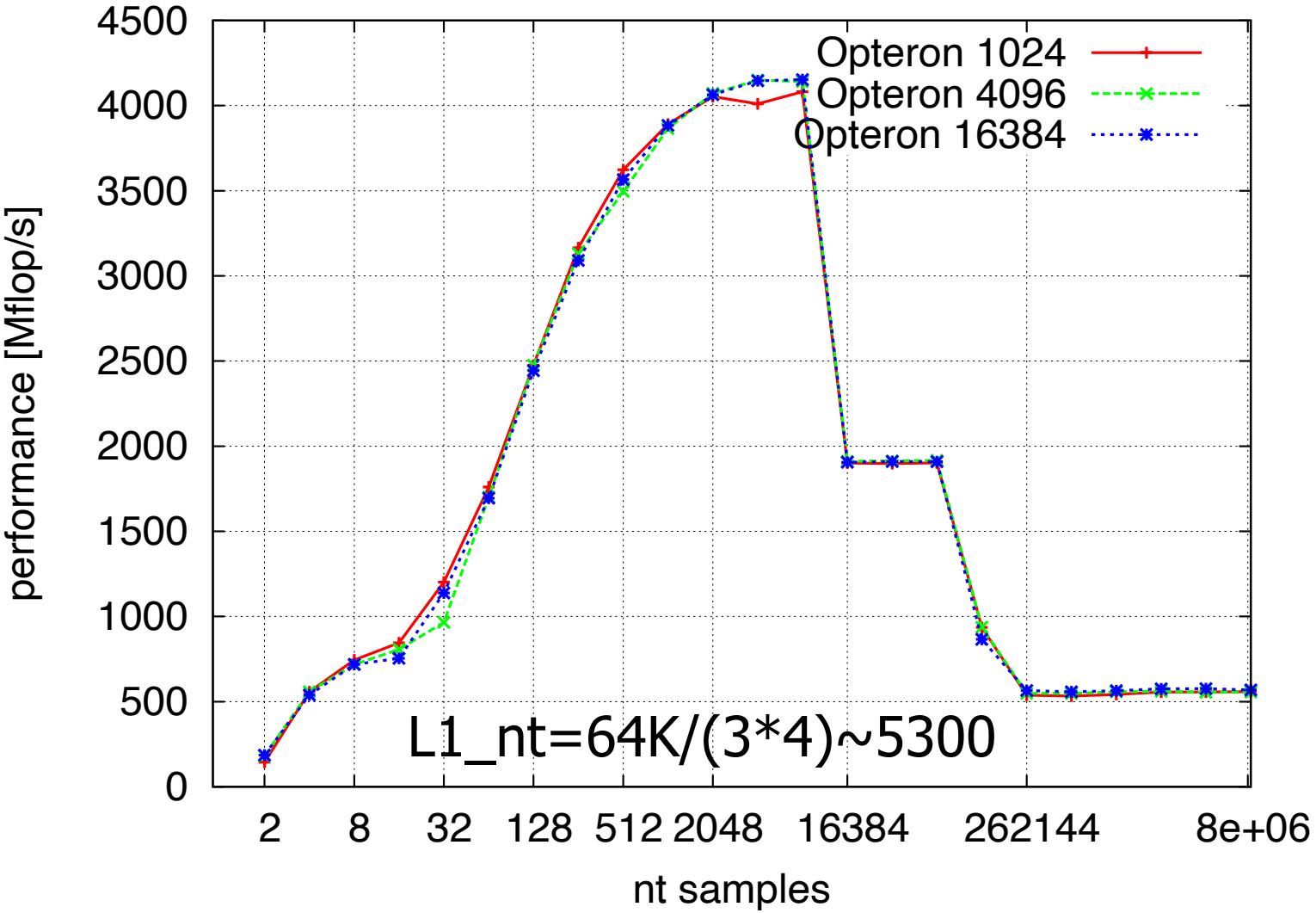
on git HPCourse/Corr

Example: Correlation

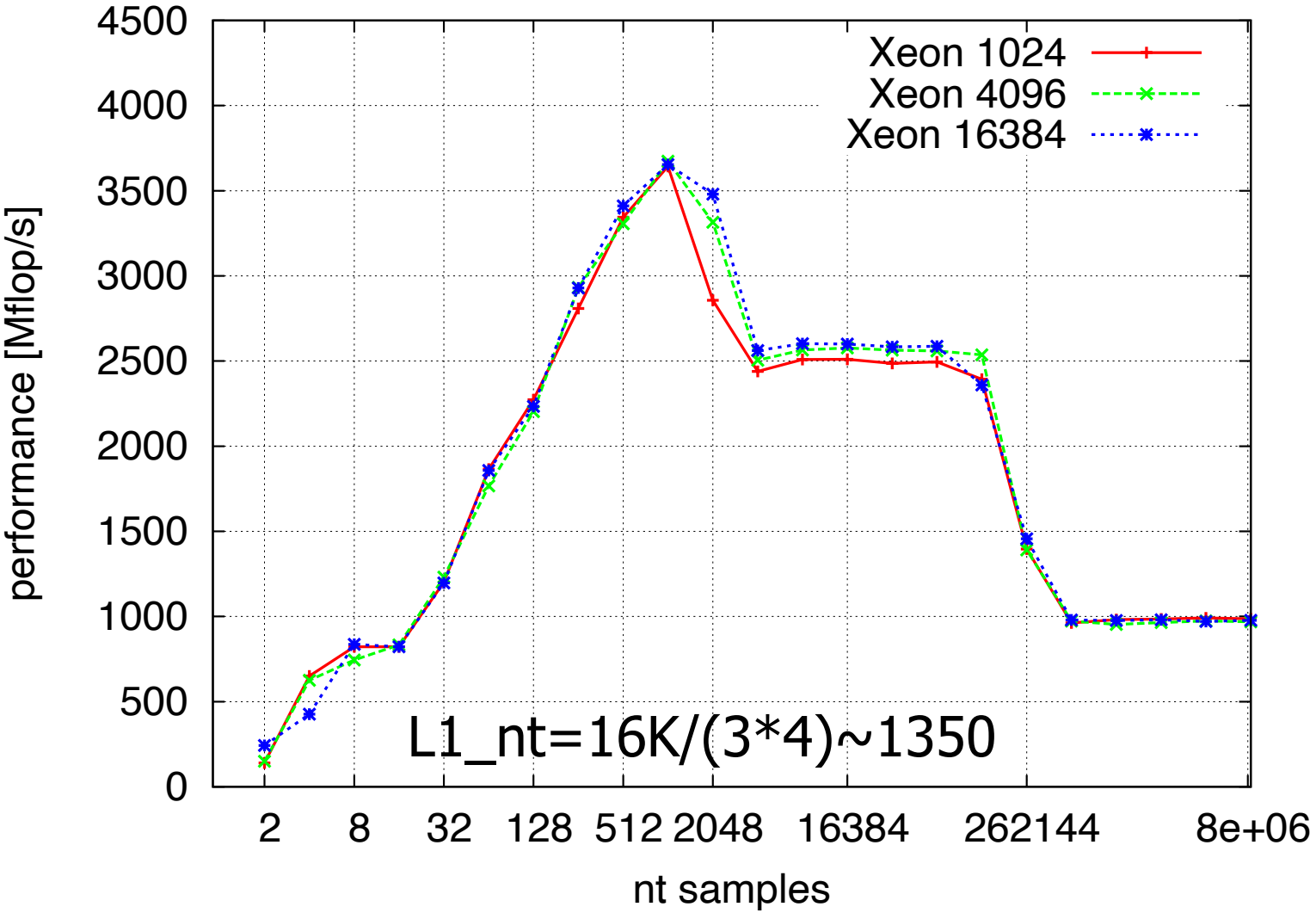
```
for (l=0; l<ntout; l++) {
        for (j=0; j<nt; j++) {
            C[l] += A[j]*B[j+l];
         }
    }
```

examined as function of nt (block size).

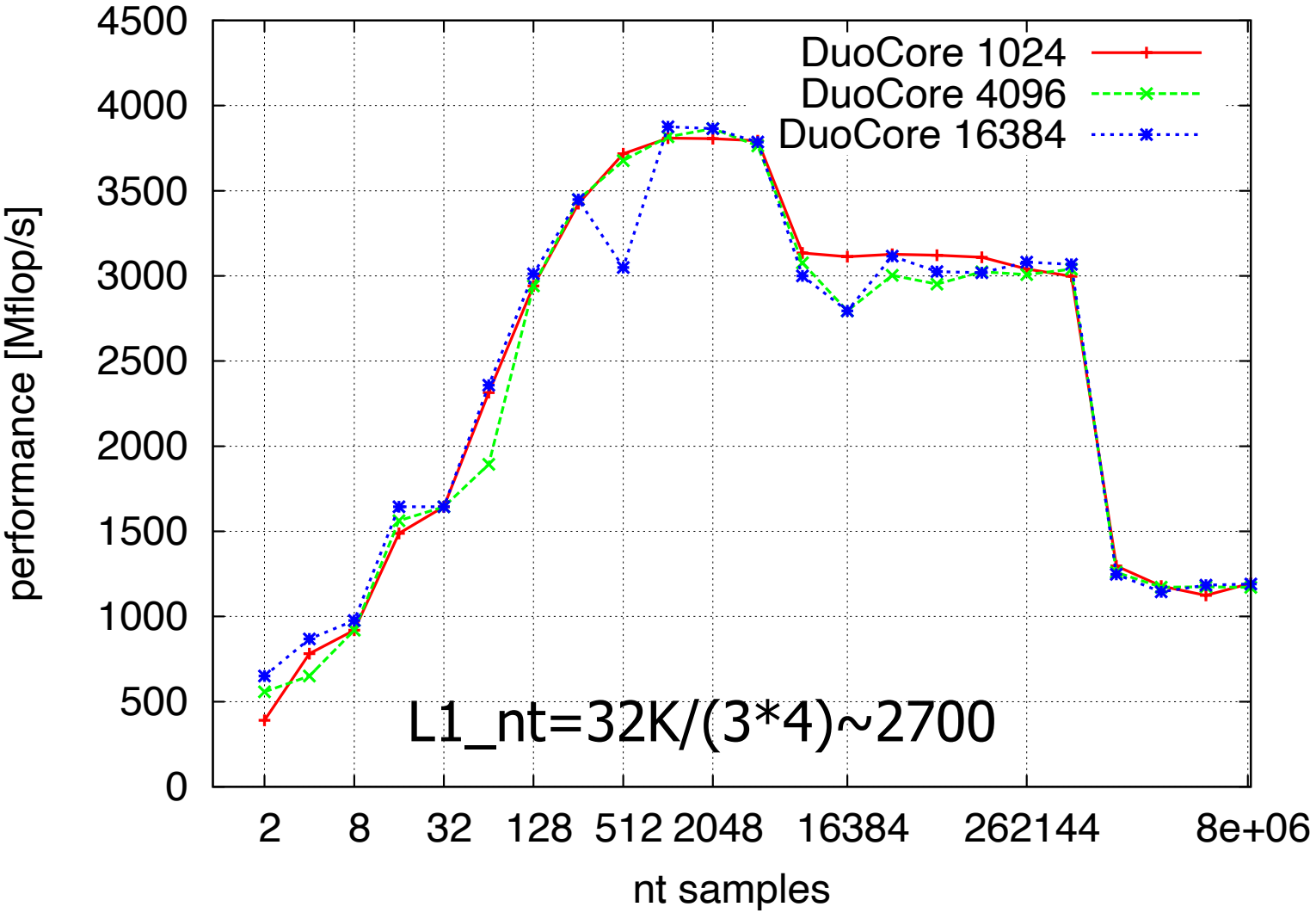
Correlation: Opteron 2.2 GHz, L1 64KB, L2 1MB



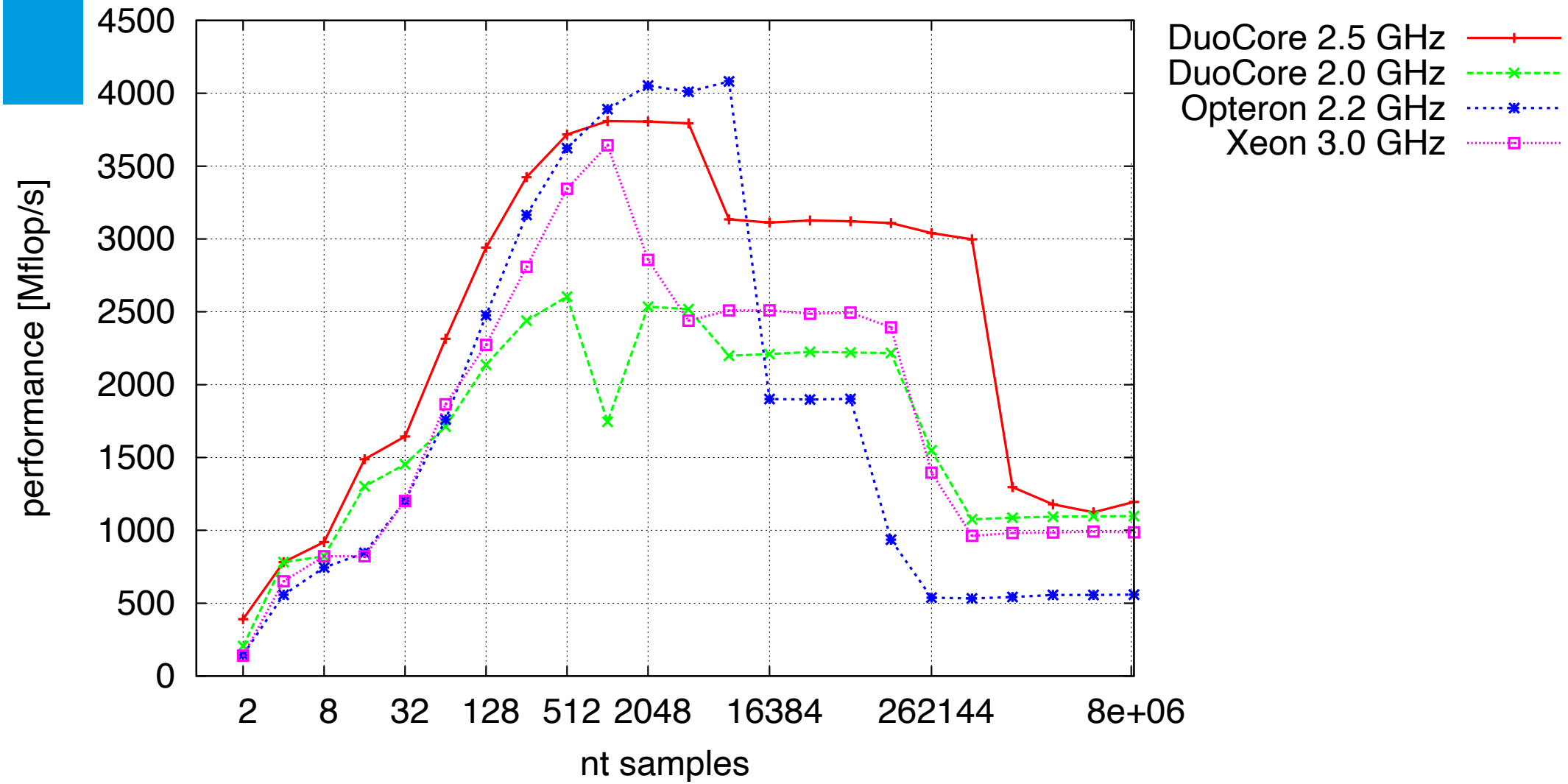
Correlation: Xeon 3.0 GHz, L1 16KB, L2 2MB



Correlation: DuoCore 2.5 GHz, L1 32KB, L2 6MB



Correlation results

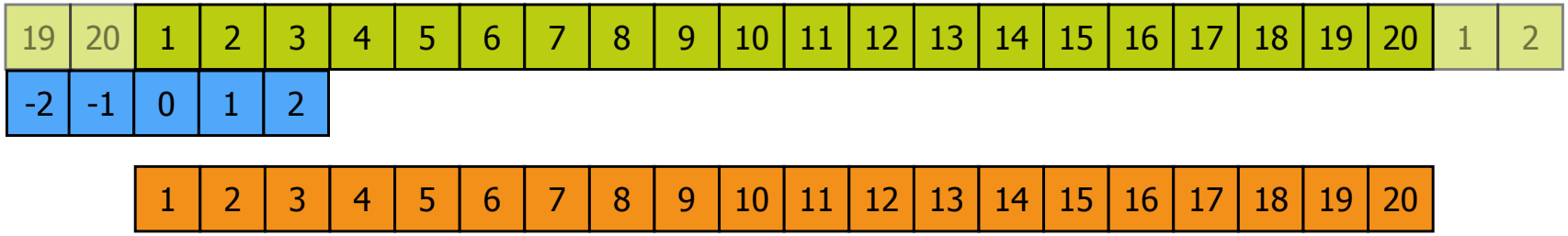


Correlation

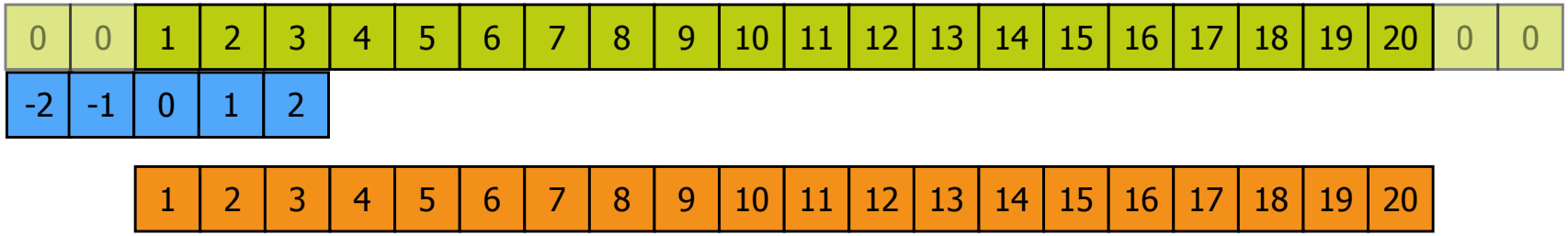
- cache-blocking helps to improve the code a lot !
- memory interface is the bottleneck in current AMD/Intel hardware.

1D convolution

- Convolution with
  - periodic boundaries



  - non-periodic boundaries



periodic boundaries

```
do i2=1,ndat
   do i1=0,n1
      tt = 0.0
      do l=lowfil,lupfil
         k=modulo(i1+l,n1+1)
         tt=tt+x(k,i2)*fil(l)
      enddo
      y(i2,i1)=tt
   enddo
enddo
```

non-periodic boundaries

```
do i2=1,ndat
   do i1=0,n1
      tt = 0.0
      do l=max(lowfil,-i1),min(lupfil,n1-i1)
         k=i1+l
         tt=tt+x(k,i2)*fil(l)
      enddo
      y(i2,i1)=tt
   enddo
enddo
```

Exercise: 1D-convolution

- Is this the best and fastest implementation?
- How many cycles takes the modulo operator?
- Inner loop dependent on outer iterator, does this vectorise?

- On git HPCource/Conv1D

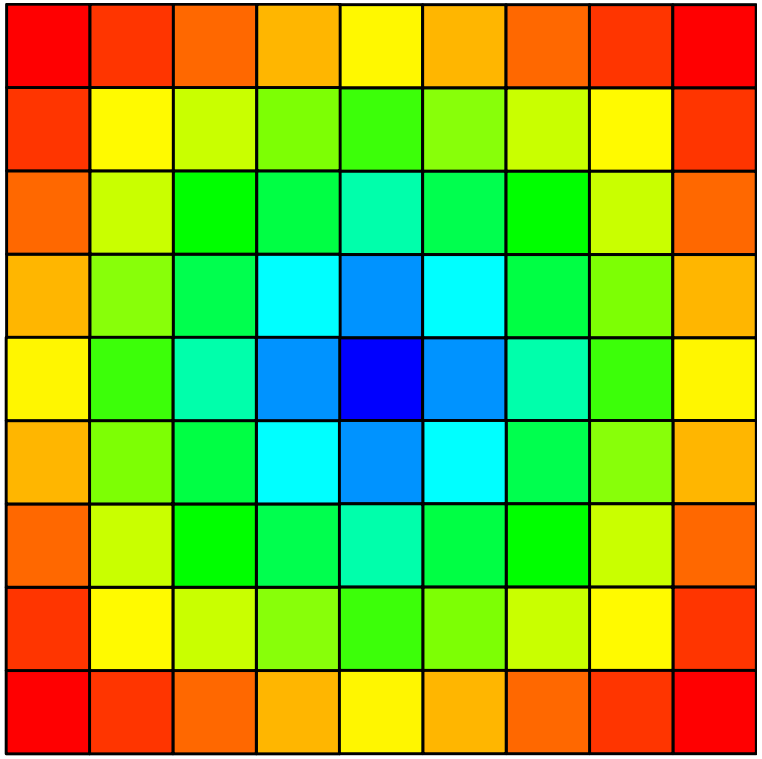
- Can you change the code and make it faster?

Example 2D convolution

$$P(x, y, z_{m+1}) = \int_{\partial D} W(x', y') P(x - x', y - y', z_m) \mathrm{x}' \mathrm{y}'$$

$$P(x, y) = \sum_{i=-hoply}^{hoply} \sum_{j=-hoplx}^{hoplx} W(x_j, y_i) P(x - x_j, y - y_i)$$

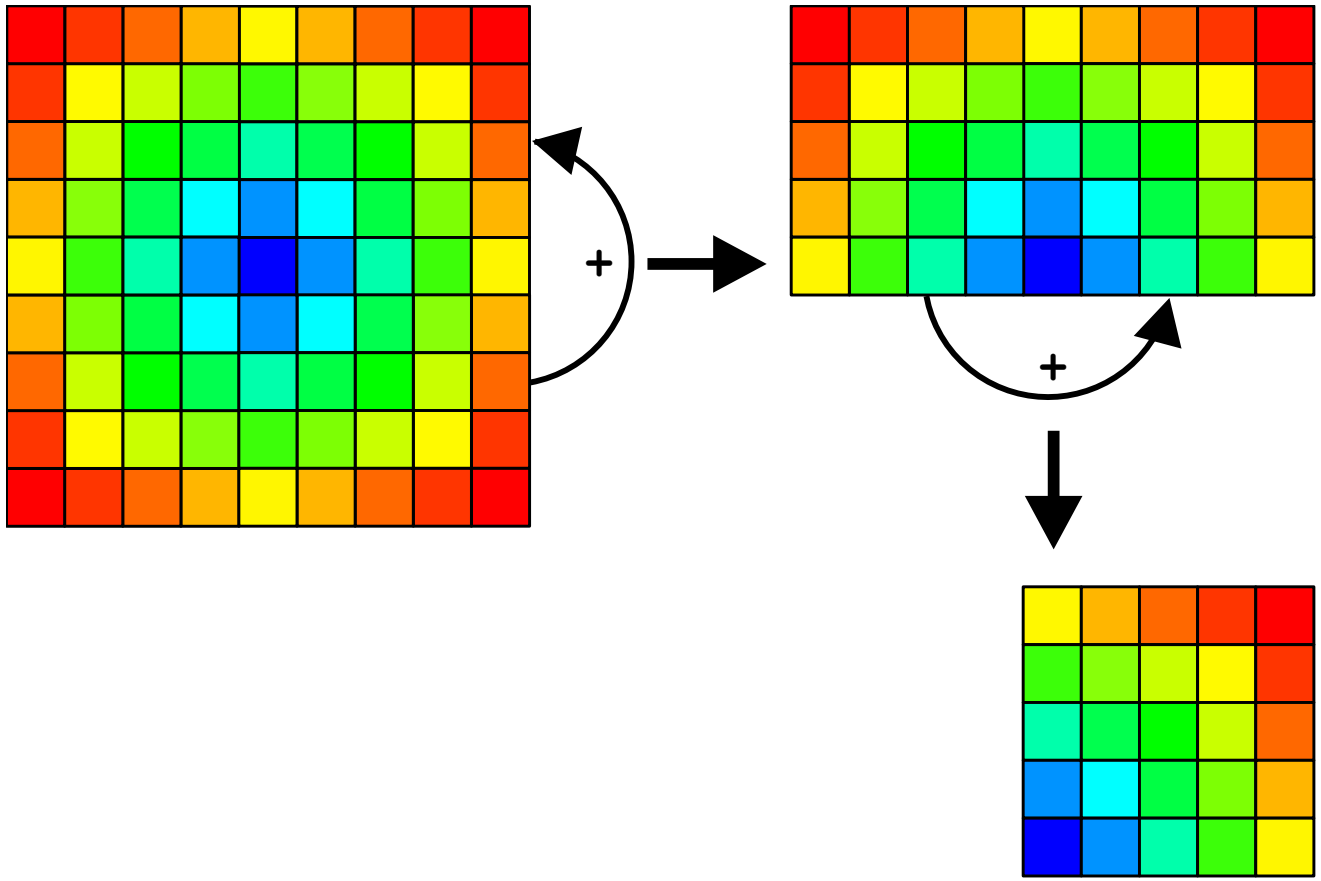
2D convolution



Implementation

```
for (iy = 0; iy < ny; iy++) {
 starty = MAX(iy-hoply+1, 0);
 endy   = MIN(iy+hoply, ny);
 for (ix = 0; ix < nx; ix++) {
     startx = MAX(ix-hoplx+1, 0);
     endx   = MIN(ix+hoplx, nx);
     dum.r = dum.i = 0.0;
     k = MAX(hoply-1-iy, 0);
     for (i = starty; i < endy; i++) {
         l = MAX(hoplx-1-ix, 0);
         for (j = startx; j < endx; j++) {
             dum.r += data[i*nx+j].r*opx[k*oplx+l].r;
             dum.r -= data[i*nx+j].i*opx[k*oplx+l].i;
             dum.i += data[i*nx+j].r*opx[k*oplx+l].i;
             dum.i += data[i*nx+j].i*opx[k*oplx+l].r;
             l++;
         }
         k++;
     }
     convr[iy*nx+ix] = dum;
 }
}
```

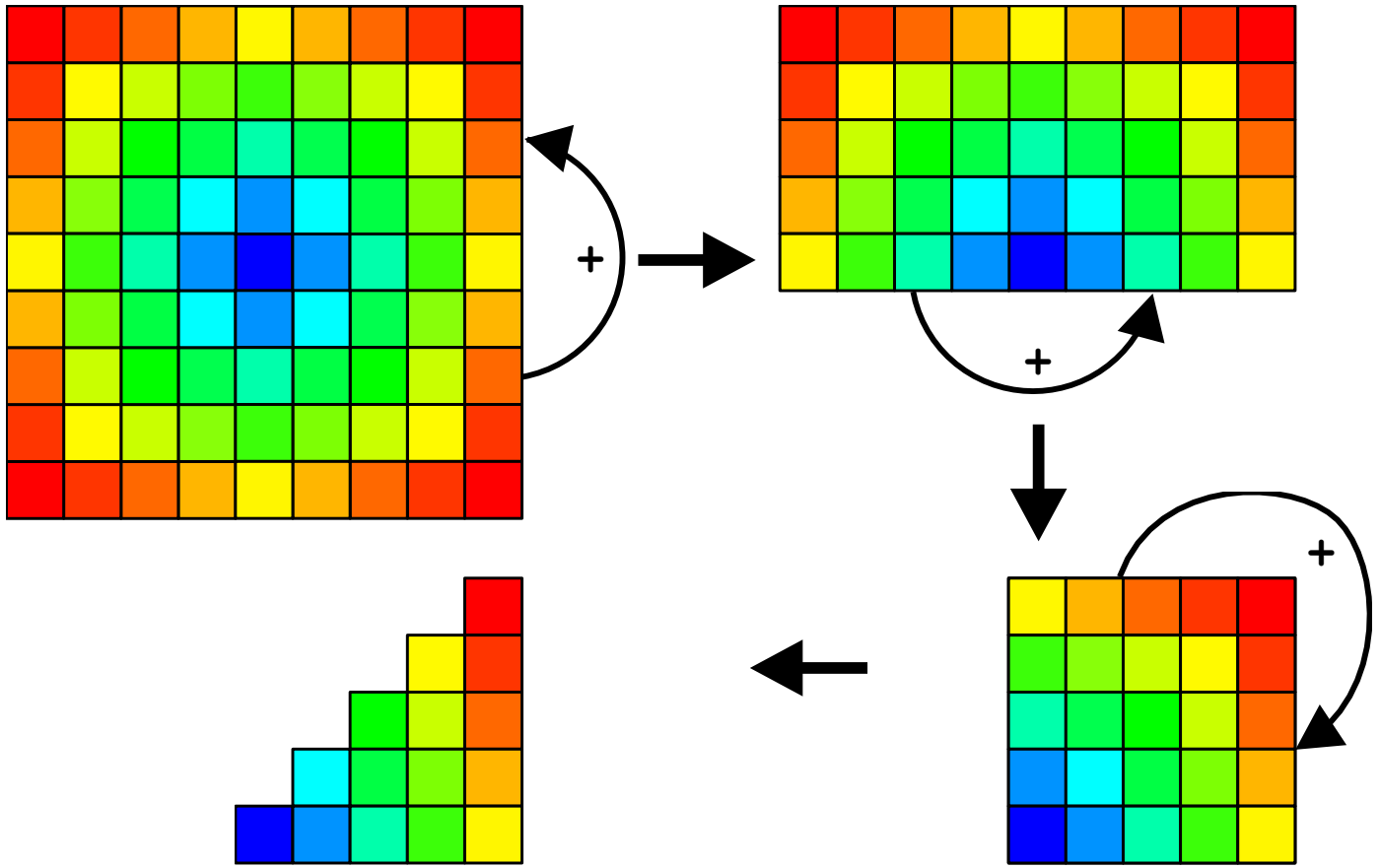
2D convolution: symmetry (1)



2D convolution: symmetry (1)

```
for (iy = hy2; iy < nyo-hy2; iy++) {
 for (ix = hx2; ix < nxo-hx2; ix++) {
     dum.r = 0.0;
     dum.i = 0.0;
     for (i = 0; i < hoply; i++) {
         for (j = 0; j < hoplx; j++) {
             dum.r += quad[i*hoplx+j].r*hopx[i*hoplx+j].r;
             dum.r += quad[i*hoplx+j].i*hopx[i*hoplx+j].i;
             dum.i += quad[i*hoplx+j].i*hopx[i*hoplx+j].r;
             dum.i -= quad[i*hoplx+j].r*hopx[i*hoplx+j].i;
         }
     }
     data[(iy-hy2)*nx+ix-hx2] = dum;
 }
}
```

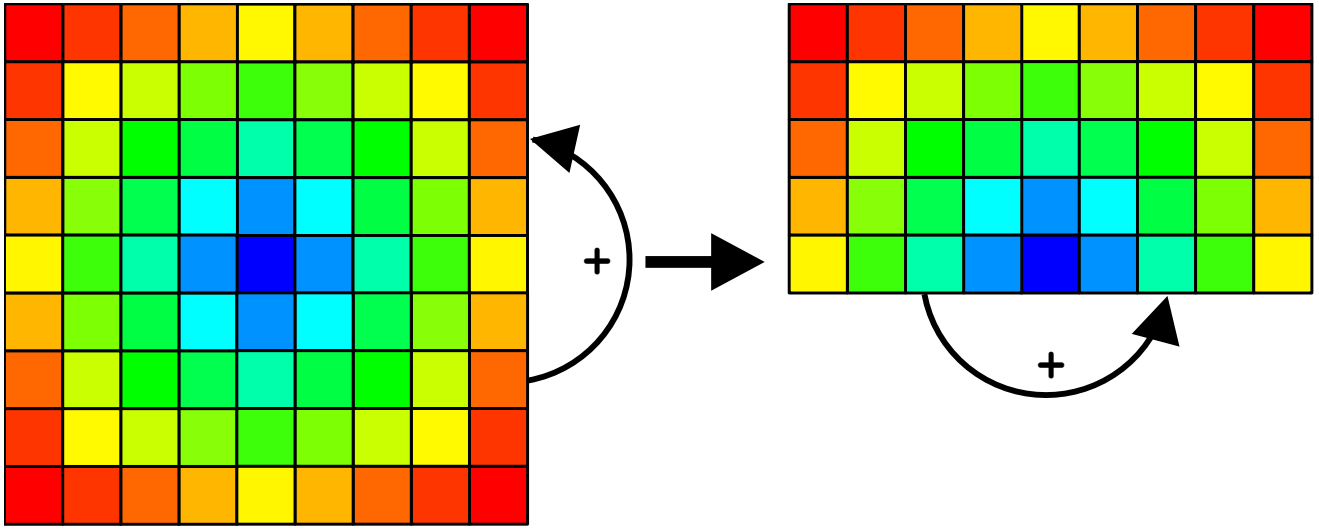
2D convolution: symmetry (2)



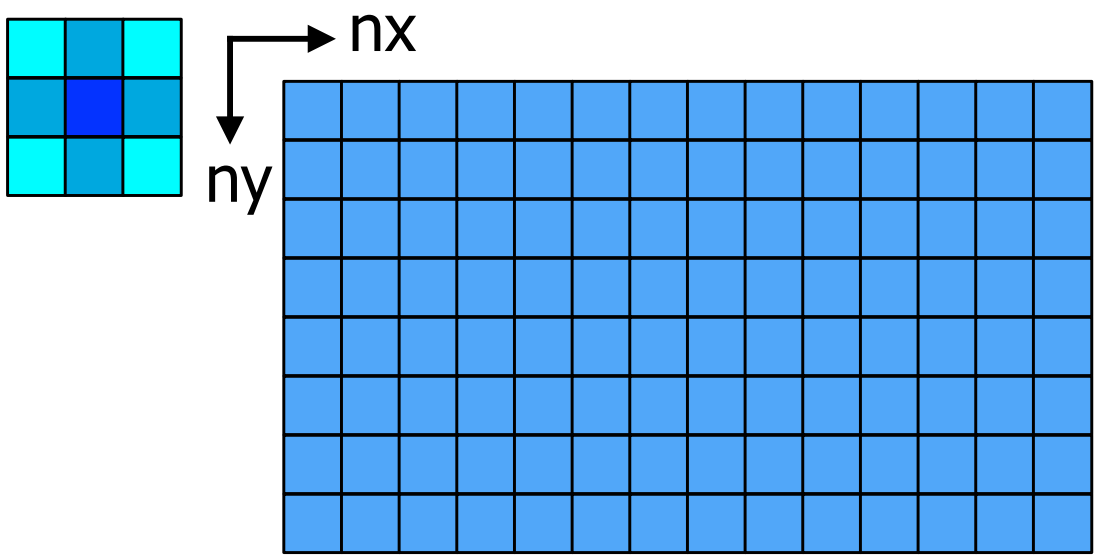
2D convolution: symmetry (2)

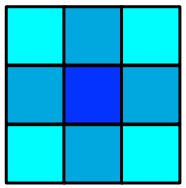
```
    for (iy = hy2; iy < nyo-hy2; iy++) {
        for (ix = hx2; ix < nxo-hx2; ix++) {
            dum.r = 0.0;
            dum.i = 0.0;
            for (i = 0; i < hoply; i++) {
                for (j = 0; j <= i; j++) {
                    dum.r +=
oct[i*hoplx+j].r*hopx[i*hoplx+j].r;
                    dum.r +=
oct[i*hoplx+j].i*hopx[i*hoplx+j].i;
                    dum.i +=
oct[i*hoplx+j].i*hopx[i*hoplx+j].r;
                    dum.i -=
oct[i*hoplx+j].r*hopx[i*hoplx+j].i;
                }
            }
```

Operator: cache



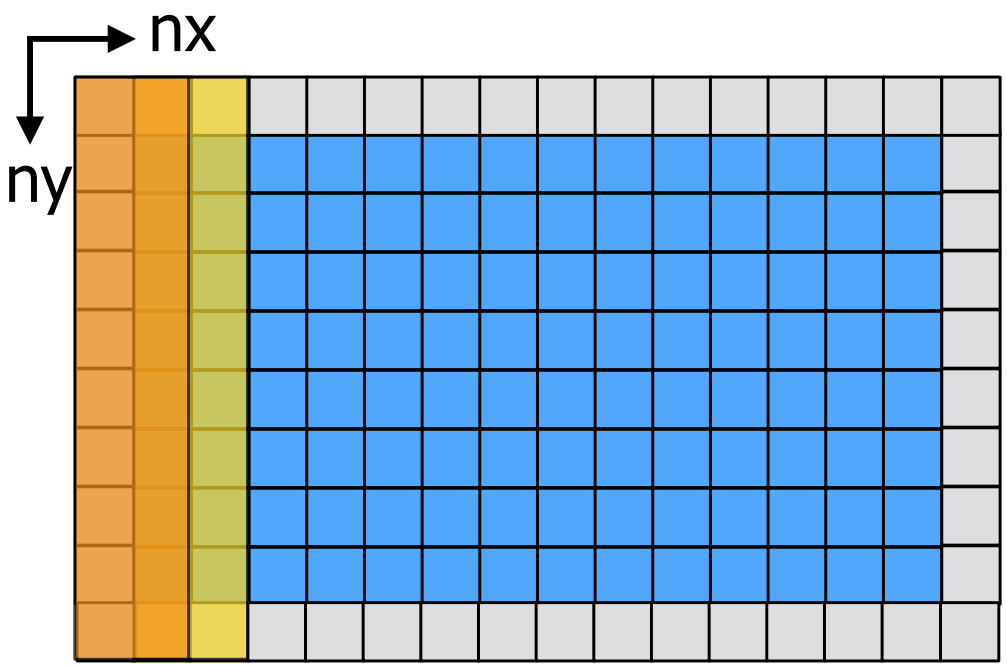
2D convolution cache-optimised



assume 3x3 symmetric operator 

2D convolution cache-optimised

adding zero's to edges : half of operator (lenght-1)/2



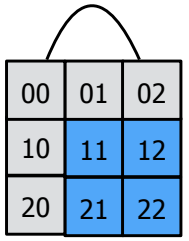
ny*((opl+1)/2) +nx

tmp1= 01 02 11 12 21 22 31 … 91 92

tmp2= 01 00 11 10 21 20 31 … 80 91 90

tmp3=tmp1[ix]+tmp2[nx+ix]

2D convolution cache-optimised



ny*((opl+1)/2) +nx

tmp1= 01 02 11 12 21 22 31 … 91 92

tmp2= 01 00 11 10 21 20 31 … 91 90

tmp3=tmp1[ix]+tmp2[nx+ix]

01 02 11 12 21 22 31 … 91 92

\+

01 00 11 10 21 20 31 … 91 90

=

0x 0y 1x 1y 2x 2y 3x … 9x 9y

0x=01+01 0y=02+00

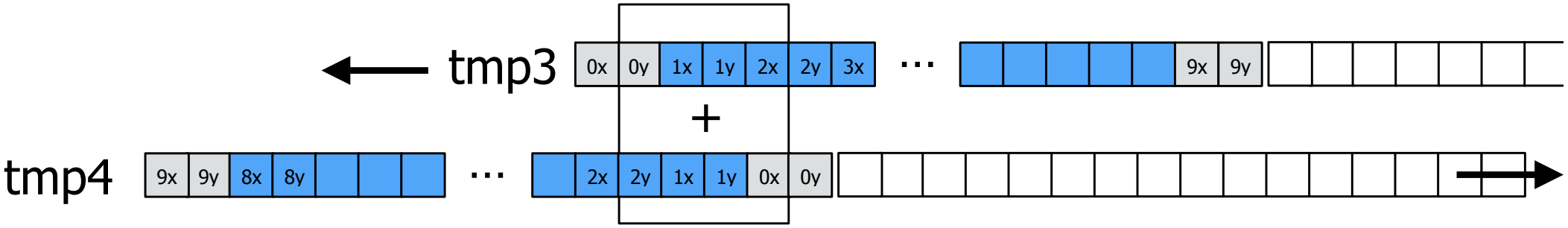
1x=11+11 1y=12+10

tmp4=tmp3[ny-1-iy]…

9x 9y 8x 8y … 2x 2y 1x 1y 0x 0y

2D convolution cache-optimised

00	01	02
10	11	12
20	21	22



0x=01+01
1x=11+11
2x=21+21

0y=02+00
1y=12+10
2y=22+20

0y+2y=00+02+20+22
0x+2x=01+21
1y=10+12
1x=11

Operator: cache

```
 for (ix = 0; ix < nx; ix++) {
       for (iy = 0; iy < ny; iy++) {
           index3 = (hoply2+iy)*hoplx;
           index4 = (leny-hoply2-iy-1)*hoplx;
           for (j = 0; j < opersize; j++) {
               dumr = tmp3[index3+j].r +
tmp4[index4+j].r;
               dumi = tmp3[index3+j].i +
tmp4[index4+j].i;
               data[iy*nx+ix].r += dumr*hopx[j].r;
               data[iy*nx+ix].r += dumi*hopx[j].i;
               data[iy*nx+ix].i += dumi*hopx[j].r;
               data[iy*nx+ix].i -= dumr*hopx[j].i;
           }
       }
```

Convolution Results

- Show timings array sizes 19x19 operator in domain 111x111 DuoCore, Opteron, Barcelona, Xeon
- Mflop/s (simple most Mflop/s (great !) but... also slowest)

Method	DuoCore 2.5 GHz icc	DuoCore 2.5 GHz gcc
conv2D simple	31	28.8
conv2D Q4	21.9	24.8
conv2D Q8	30.2	35.9
conv2D cache	19	14.6

Single Core Tuning

- Get the right answers
- Use existing tuned code (BLAS, LAPACK, FFT)
- Find out where to tune
- Let the compiler do the work:
  - compilers are good at instruction level optimizations
  - memory optimisations are performed, but often benefit from high level insight
- The cache hierarchy and loop nest optimizations

Optimal Cache Performance

Re-use data in the cache
Temporal locality

Use all the data in one cache line
Spatial locality

General guidelines

- Use the compiler options -O3 -fast -unroll -fastmath, ...
- Loop fusion or fission
- use of tmp arrays or tmp scalars
- re-use of already computed data
- complexity within a loop for vectorisation
- hiding expensive operations with less expensive one's
- If-statements in a loop may stall the pipeline.

References

- Good tutorial to performance optimization
  http://icl.cs.utk.edu/~mucci/latest/pubs/Notur2009-new.pdf
- What every programmer should know about memory
  http://www.akkadia.org/drepper/cpumemory.pdf
- More software optimization web resources
  http://www.agner.org/optimize/
- How To Write Fast Numerical Code: A Small Introduction
  http://www.ece.cmu.edu/~franzf/papers/gttse07.pdf

Exercise: Correlation

- The code implements time correlation
- Download code from:
  http://www.xs4all.nl/~janth/HPCourse/code/Correlation.tar
- Unpack tar file and check the README for instructions.

Exercise: 2D Convolution

- The code implements 2D convolution in different ways
- On your git clone: cd HPCourse/Conv2D
- Unpack tar file and see if you can this code to compile and link correctly by making changes in the Makefile
- The bash script puls.scr runs the different implementations.

Exercise: Matrix Multiply + Block

- To be discussed in the optimization lecture next week.
- On the git clone: cd HPCourse/MatMulBlock
- Check the README for instructions.
- In the Makefile change LIBS= to a path where the BLAS library is. if you can not find BLAS just set LIBS= (empty)
-

Exercise: Shared Cache Trashing

- First experiment in parallel programming with OpenMP
- On your git clone: cd HPCourse/CacheTrash
- Unpack tar file and check the README for instructions.
- You might have to make the stacksize of your system larger.
- Run the program shared_cache
  are the runtimes expected?
- Will be discussed in the next part of the course

Exercise: Advanced Cache Blocking

- Three loop nests to block
  - 19-point sixth-order stencil
  - Another 19-point stencil loop
  - Tensor reduction
- On your git clone: cd HPCourse/CacheBlock
- Try to get the loop faster using any technique you have learned at the course. Note, this is not easy.
- Modifying data layout is not allowed (but if you do it, show me anyway!)

Exercise: OpenMP scheduling

- How OpenMP can help parallelize a loop, but as a user you can help in making it better ;-)
- On the git clone: cd HPCourse/OMP_schedule
- The README has instructions:
  This works best with an Intel compiler that uses an auto-parallelizer to generate threaded code
- This exercise requires already some knowledge about OpenMP.
  The OpenMP website at
  https://www.openmp.org/
  can be helpful to get started.