MHPC
Master in High Performance Computing

DAY 5: Memory hierarchy and associated optimization techniques

Stefano Cozzini

CNR/IOM and eXact-lab srl

SISSA
Scuola Internazionale Superiore di Studi Avanzati

ICTP The Abdus Salam International Centre for Theoretical Physics

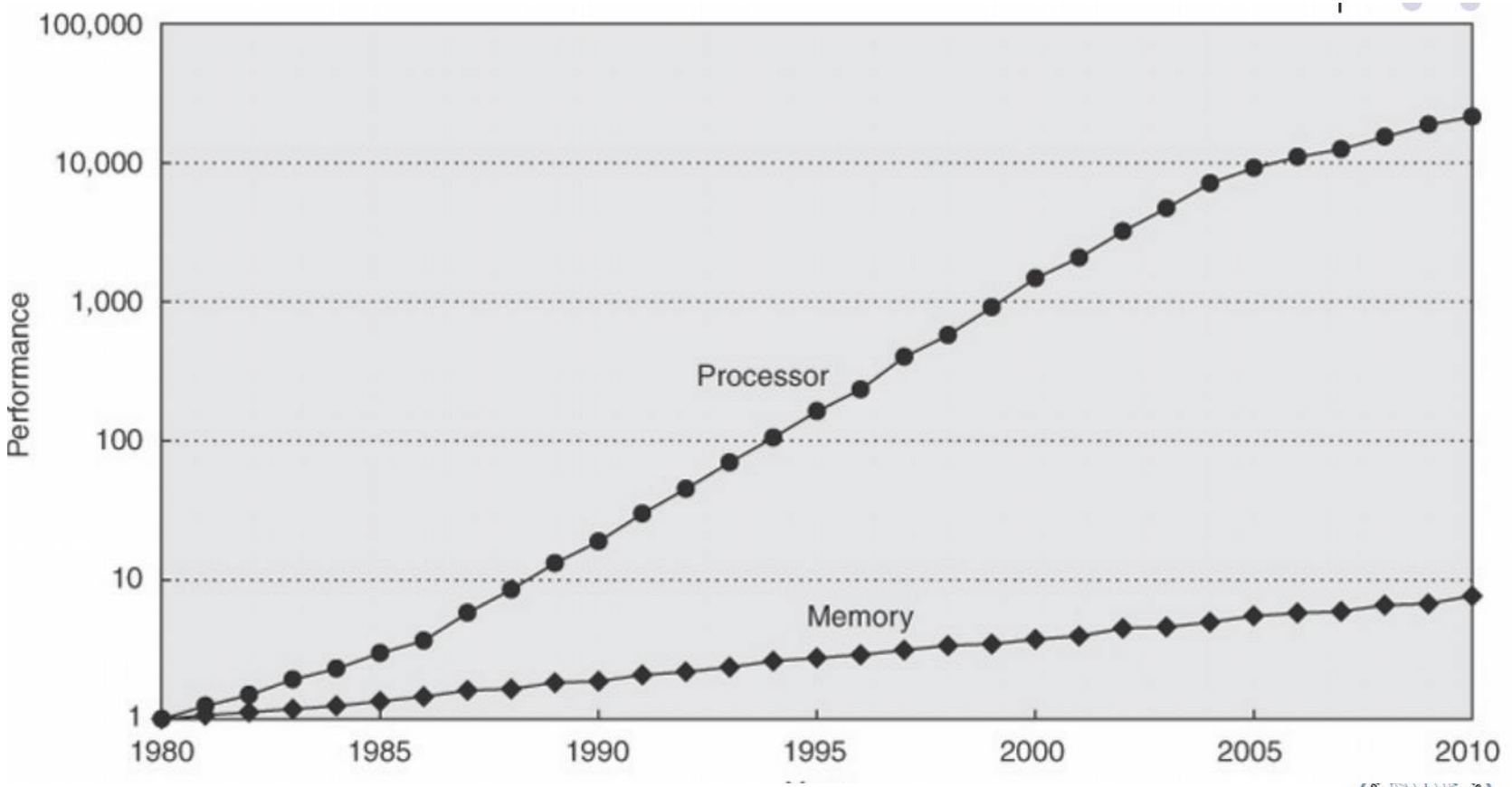
MHPC
Master in High Performance Computing

Outline

- Part 1: The memory hierarchy
  - –Concepts&ideas
  - –Caches structure
- Part 2: Optimization techniques for memory hierarchy

SISSA ICTP

The memory wall



MHPC
Master in High Performance Computing

The memory wall: some notes

- Memory latency is a barrier to performance improvements
- Current architectures have ever growing caches to improve the "average memory reference" time to fetch or write instructions or data
- Memory Wall: due to latency and limited communication bandwidth beyond chip boundaries.
- From 1986 to 2000, CPU speed improved at an annual rate of 55% while memory access speed only improved at 10%

SISSA ICTP

MHPC
Master in High Performance Computing

Memory Hierarchy

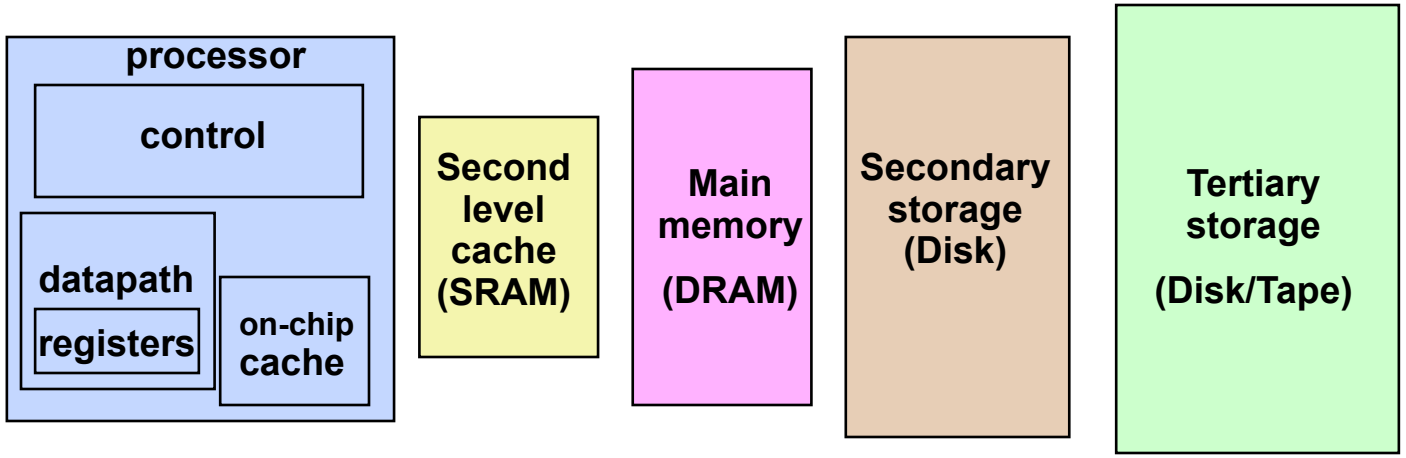
In modern computer system same data is stored in several storage devices during processing

The storage devices can be described & ranked by their speed and "distance" from the CPU

There is thus a *hierarchy of memory objects*

Programming a machine with memory hierarchy requires optimization for that memory structure.

Memory Hierarchy



Speed	1ns	10ns	100ns	10ms	10sec
Size	B	KB	MB	GB	TB

Memory Hierarchies

- Memory is divided into different levels:
  - Registers
  - Caches
  - Main Memory
- Memory is accessed through the hierarchy
  - registers where possible
  - ... then the caches
  - ... then main memory

Latency and Bandwidth

- The two most important terms related to performance for memory subsystems and for networks are:
  - Latency
    - How long does it take to retrieve a word of memory?
    - Units are generally nanoseconds (milliseconds for network latency) or clock periods (CP).
    - Sometimes addresses are predictable: compiler will schedule the fetch. Predictable code is good!
  - Bandwith
    - What data rate can be sustained once the message is started?
    - Units are B/sec (MB/sec, GB/sec, etc.)

MHPC
Master in High Performance Computing

Registers

- Highest bandwidth, lowest latency memory that a modern processor can access
  - –built into the CPU
  - –often a scarce resource
  - –not RAM
- Processors instructions operate on registers directly
  - have assembly language names names like:
    - eax, ebx, ecx, etc.
  - sample instruction:

```
addl  %eax, %edx
```

SISSA ICTP

Data Caches

- Between the CPU Registers and main memory
- L1 Cache: Data cache closest to registers
- L2 Cache: Secondary data cache, stores both data and instructions
  - Data from L2 has to go through L1 to registers
  - L2 is 10 to 100 times larger than L1
- L3 cache, ~10x larger than L2

MHPC
Master in High Performance Computing

Cache line

- The smallest unit of data transferred between main memory and the caches (or between levels of cache; every cache has its own line size)
- *N* sequentially-stored, multi-byte words (usually N=8 or 16).
- If you request one word on a cache line, you get the whole line
  - make sure to use the other items, you've paid for them in bandwidth
  - Sequential access good, "strided" access ok, random access bad

SISSA ICTP

Main Memory

- Cheapest form of RAM
- Also the slowest
  - lowest bandwidth
  - highest latency
- Unfortunately most of our data lives out here

Multi-core chips

- Cores may have separate L1/L2, shared L2/L3 cache
  - Depends on CPU model
  - Hybrid shared/distributed model
- Cache coherency problem: conflicting access to duplicated cache lines.

(more on this when we discuss multicore)

Cache and register access

- Access is transparent to the programmer
  - data is in a register or in cache or in memory
  - Loaded from the highest level where it's found
  - processor/cache controller/MMU hides cache access from the programmer
- … but you can influence it:
  - Access *x* (that puts it in L1), access 100k of data, access *x* again: it will probably be gone from cache
  - If you use an element twice, don't wait too long
  - If you loop over data, try to take chunks of less than cache size
  - C declare register variable, only suggestion

Register use

- y[i] can be kept in register
- Declaration is only suggestion to the compiler
- Compiler can usually figure this out itself

```
for (i=0; i<m; i++) {
  for (j=0; j<n; j++) {
    y[i] = y[i]+a[i][j]*x[j];
  }
}
```

```
register double s;
for (i=0; i<m; i++) {
  s = 0.;
  for (j=0; j<n; j++) {
    s = s+a[i][j]*x[j];
  }
  y[i] = s;
}
```

MHPC
Master in High Performance Computing

Hits, Misses, Thrashing

- Cache hit
  - location referenced is found in the cache
- Cache miss
  - location referenced is not found in cache
  - triggers access to the next higher cache or memory
- Cache thrashing
  - Two data elements can be mapped to the same cache line: loading the second "evicts" the first
  - Now what if this code is in a loop? "thrashing": really bad for performance

MHPC
Master in High Performance Computing

Cache Mapping

- Because each memory level is smaller than the next-closer level, data must be mapped

- Types of mapping
  - Direct
  - Set associative
  - Fully associative

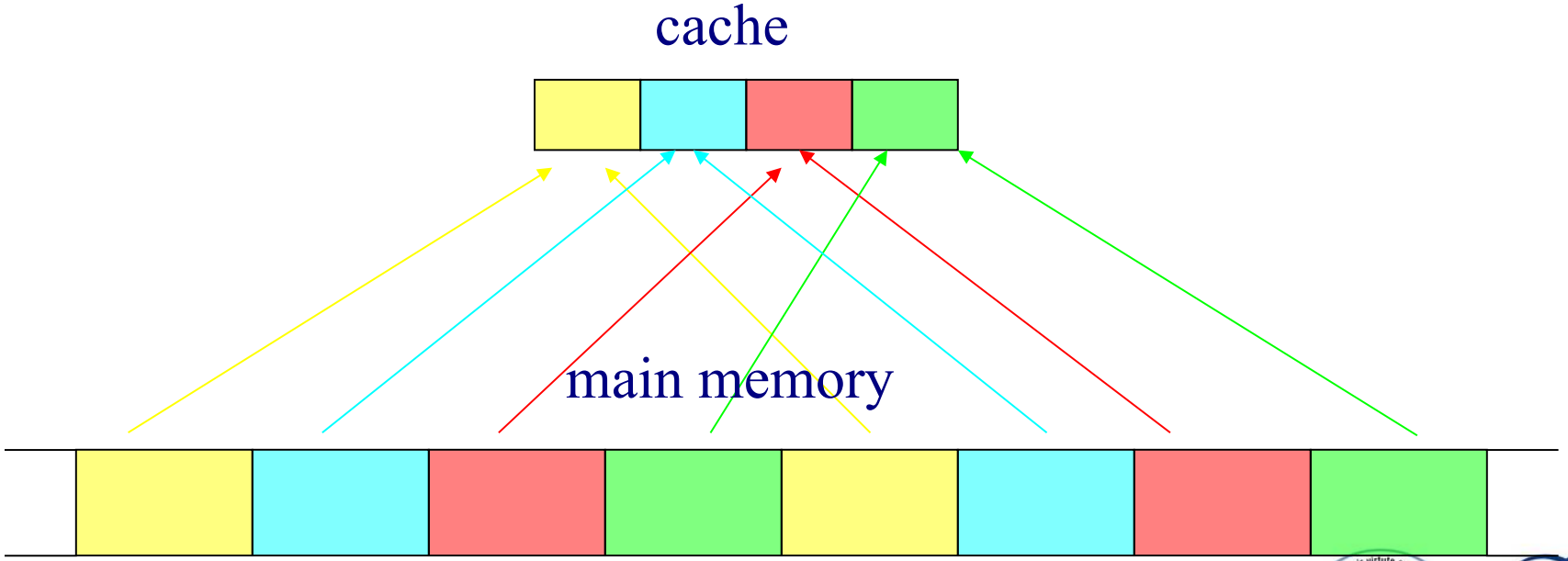
SISSA
ICTP

Direct Mapped Caches

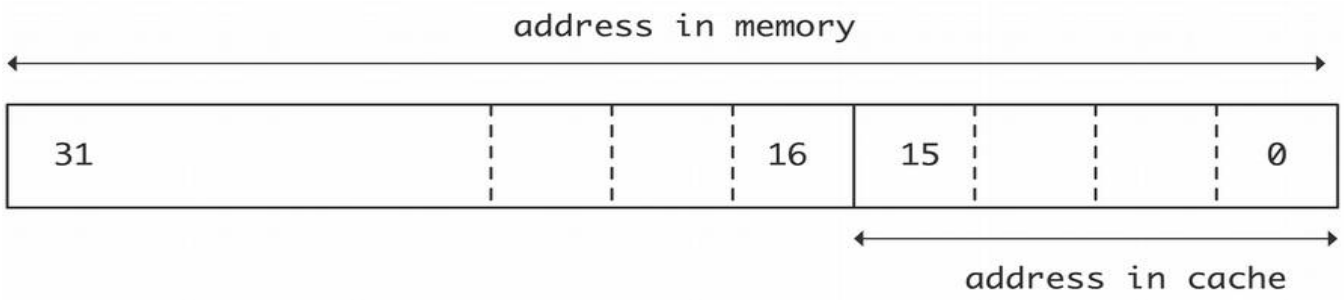
- If the cache size is N_c and it is divided into k lines, then each cache line Is N_c/k in size
- If the main memory size is N_m, memory is then divided into $N_m/(N_c/k)$ blocks that are mapped into each of the k cache lines
- Means that each cache line is associated with particular regions of memory

Direct Mapped Caches

Direct mapped cache: A block from main memory can go in exactly one place in the cache. This is called direct mapped because there is direct mapping from any block address in memory to a single location in the cache. Typically modulo calculation



Direct mapping example



- Memory is 4G: 32 bits
- Cache is 64K (or 8K words): 16 bits
- Map by taking last 16 bits

The problem with Direct Mapping

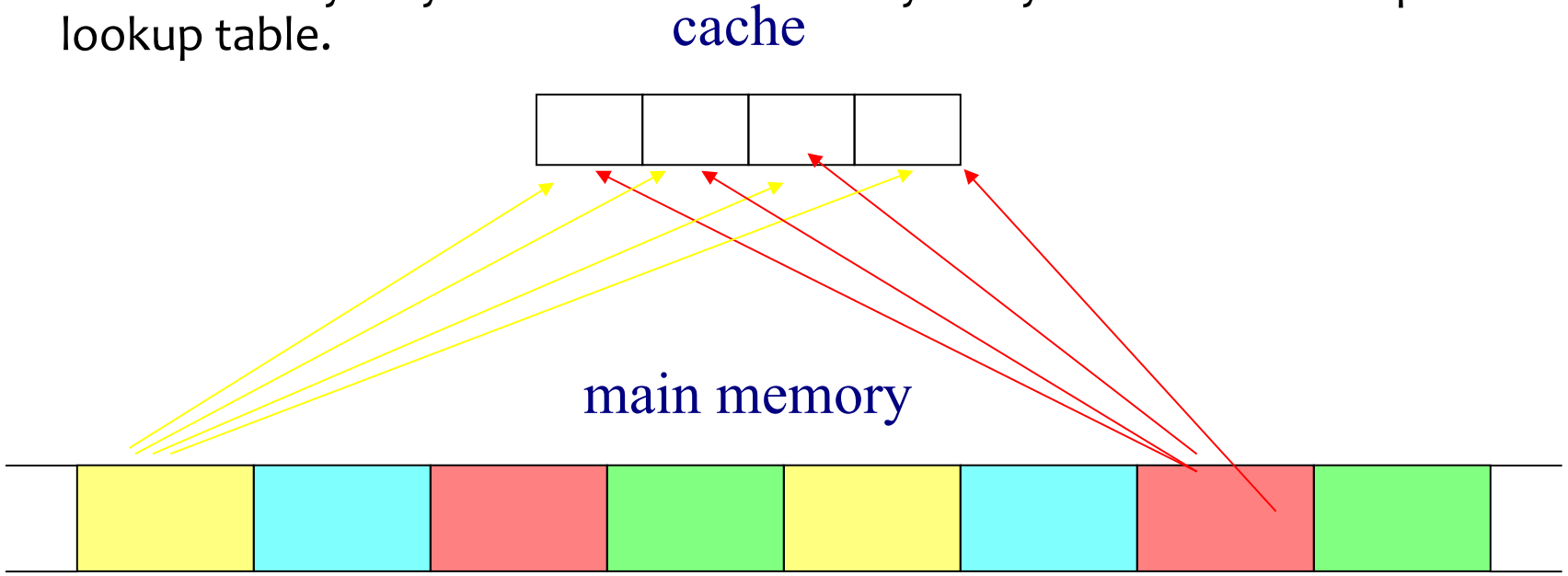
```
double a[8192],b[8192];
for (i=0; i<n; i++) {
  a[i] = b[i]
}
```

Example: cache size 64k=2^{16} byte = 8192 words

- a[0] and b[0] are mapped to the same cache location
- Cache line is 4 words
- Thrashing:
  - b[0]..b[3] loaded to cache, to register
  - a[0]..a[3] loaded, gets new value, *kicks b[0]..b[3] out of cache*
  - b[1] requested, so b[0]..b[3] loaded again
  - a[1] requested, loaded, *kicks b[0..3] out again*

Fully Associative Caches

Fully associative cache : A block from main memory can be placed in any location in the cache. This is called fully associative because a block in main memory may be associated with any entry in the cache. Requires lookup table.

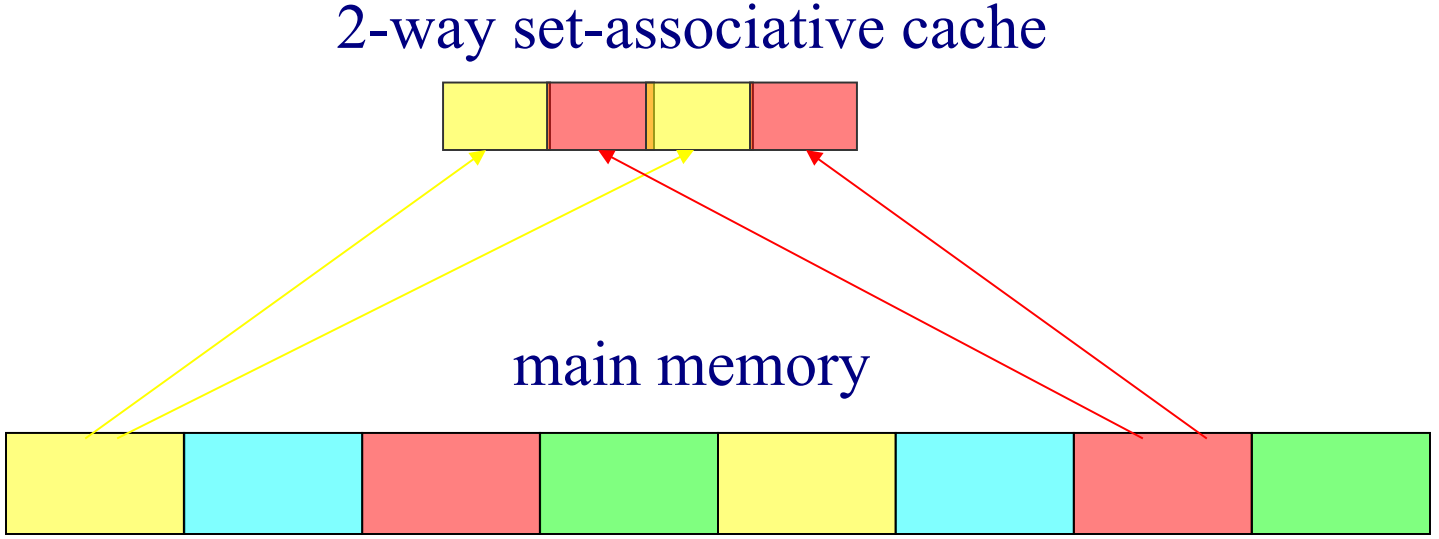


Fully Associative Caches

- Ideal situation
- Any memory location can be associated with any cache line
- Cost prohibitive

Set Associative Caches

Set associative cache : The middle range of designs between direct mapped cache and fully associative cache is called set-associative cache. In a n-way set-associative cache a block from main memory can go into n (n at least 2) locations in the cache.



Set Associative Caches

- Direct-mapped caches are 1-way set-associative caches
- For a k-way set-associative cache, each memory region can be associated with *k* cache lines
- Fully associative is k-way with k the number of cache lines

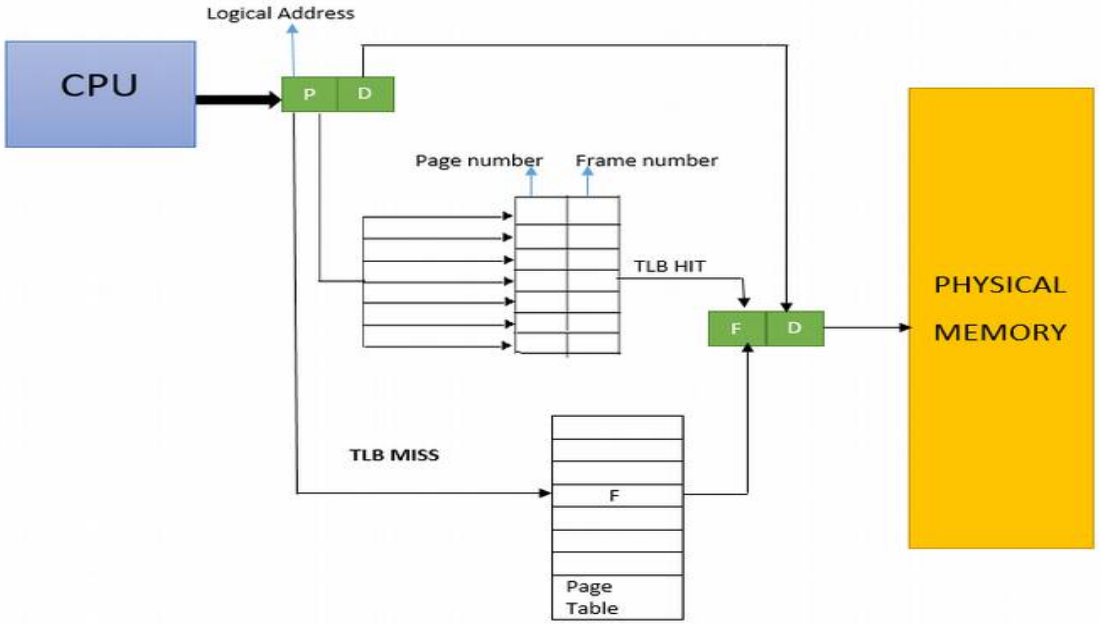
Last element: TLB :Translation Look-aside buffer

Translates between logical space that each program has and actual memory addresses

Memory organized in 'small pages', a few Kbyte in size

Memory requests go through the TLB, normally very fast

Pages that are not tracked through the TLB can be found through the 'page table': much slower



Jumping between more pages than the TLB can track has a performance penalty.

MHPC
Master in High Performance Computing

a top disaster: swapping..

virtual or swap memory:

This memory, is actually space on the hard drive. The operating system reserves a space on the hard drive for “swap space”.

time to access virtual memory VERY large:
this time is done by the system not by your program !
sometimes the system assumes a killer to kill your program..

(oom killer)

```
top - 08:57:07 up 6 days, 19:35,  7 users,  load average: 2.77, 0.73, 0.25
Tasks:  86 total,   2 running,  84 sleeping,   0 stopped,   0 zombie
Cpu(s):  0.3% us,  4.8% sy,  0.0% ni,  0.0% id, 94.2% wa,  0.6% hi,  0.0% si
Mem:    507492k total,   506572k used,      920k free,      196k buffers
Swap:  2048248k total,   941984k used,  1106264k free,     4740k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
11656 cozzini   18   0 2172m 408m  260 D  4.3 82.4   0:03.75 a.out
   33 root      15   0     0    0    0 D  0.7  0.0   0:00.54 kswapd0
 3195 root      15   0 20696 1432 1140 D  0.3  0.3   0:06.81 clock-applet
```

SISSA

top disaster example (1)

```
[cozzini@stroligo optimization]$ /usr/bin/time ./a.out
 provide an integer (suggested range 100-250)
 larger values can be very memory and time-consuming
300
 inizialisation time= 11.787208
10.86user 0.98system 0:14.22elapsed 83%CPU (0avgtext+0avgdata
0maxresident)k
0inputs+0outputs (5major+106090minor)pagefaults 0swaps

[cozzini@stroligo optimization]$ /usr/bin/time ./a.out
 provide an integer (suggested range 100-250)
 larger values can be very memory and time-consuming
320
Command terminated by signal 2
0.18user 1.81system 0:29.27elapsed 6%CPU (0avgtext+0avgdata
0maxresident)k
0inputs+0outputs (5846major+170788minor)pagefaults 0swaps
```

top disaster example (2)

```
[cozzini@stroligo optimization]$ /usr/bin/time ./a.out <300 &
[cozzini@stroligo optimization]$ free
             total       used       free     shared    buffers
cached
Mem:        507492     484916      22576          0       1156
10172
-/+ buffers/cache:     473588      33904
Swap:      2048248      78108    1970140


[cozzini@stroligo optimization]$ /usr/bin/time ./a.out <320 &
[cozzini@stroligo optimization]$ free
             total       used       free     shared    buffers
cached
Mem:        507492     506412       1080          0        252
3936
-/+ buffers/cache:     502224       5268
Swap:      2048248     546348    1501900
```

MHPC
Master in High Performance Computing

Intel Xeon E5 v2 2680 caches

- L1
  - 32 KB
  - 8-way set associative
  - 64 byte line size
- L2
  - 4 MB
  - 8-way set associative
  - 64 byte line size

TASK find out all details on your cache hierarchy

SISSA ICTP

MHPC
Master in High Performance Computing

PART 2: optimization technique for memory

SISSA
ICTP

MHPC
Master in High Performance Computing

Optimization Techniques

There are basically three different categories:

- Improve CPU performance
- Use already highly optimized libraries/subroutines
- Improve memory performance (The most important)
  - Better memory access pattern
  - Optimal usage of cache lines (improve spatial locality)
  - Re-usage of cached data (improve temporal locality)

SISSA ICTP

MHPC
Master in High Performance Computing

From Luca's slides

Some Exa-Scale facts

POWER WALL

- Cores need to be as simple as possible
- **Code optimization** becomes fundamental
- Concurrency programming → **software design**
- Cores' **specialization** must be exploited
- **Billion-way parallelism**

MEMORY WALL

- Memory power wall + spatial constraints + cost constraints → very small memory with high bw
- **Extreme multi-level NUMA hierarchy**: L1 -> L2 -> L3 -> local RAM (shared) -> non-local RAM -> distant RAM
- Possible PGAS paradigm
- **Data locality by design is mandatory**

SISSA ICTP

A few more remarks ..

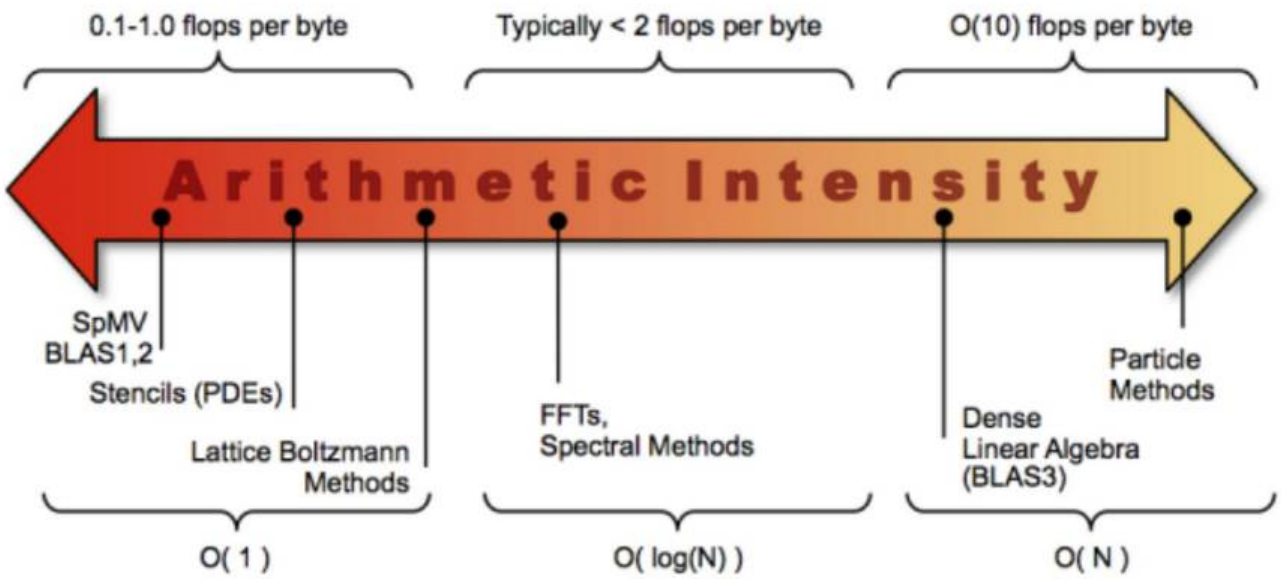
- Running time of an algorithm is the sum of 3 terms:
  - # flops * time_per_flop
  - # words moved / bandwidth
  - # messages * latency

} communication

- Time_per_flop << 1/ bandwidth << latency
  - Gaps growing exponentially with time

Annual improvements			
Time_per_flop		Bandwidth	Latency
59%	Network	26%	15%
	DRAM	23%	5%

Again on arithmetic intensity



MHPC
Master in High Performance Computing

Data reuse

- Performance is limited by data transfer rate
- High performance if data items are used multiple times
- Example: vector addition

  $x_i=x_i+y_i$: 1 op, 3 mem accesses

- Example: inner product
  - $s=s+x_i*y_i$:
  - 2 op, 2 mem access (s in register; also no writes)

SISSA
ICTP

Data reuse: matrix-matrix product

- Matrix-matrix product: $2n^3$ ops, $2n^2$ data

```
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
    s = 0;
    for (k=0; k<n; k++) {
      s = s+a[i][k]*b[k][j];
    }
    c[i][j] = s;
  }
}
```

Data reuse: matrix-matrix product

- Matrix-matrix product: $2n^3$ ops, $2n^2$ data
- If it can be programmed right, this can overcome the bandwidth/cpu speed gap
- Again only theoretically: naïve implementation inefficient
- *Do not code this yourself..*

MHPC
Master in High Performance Computing

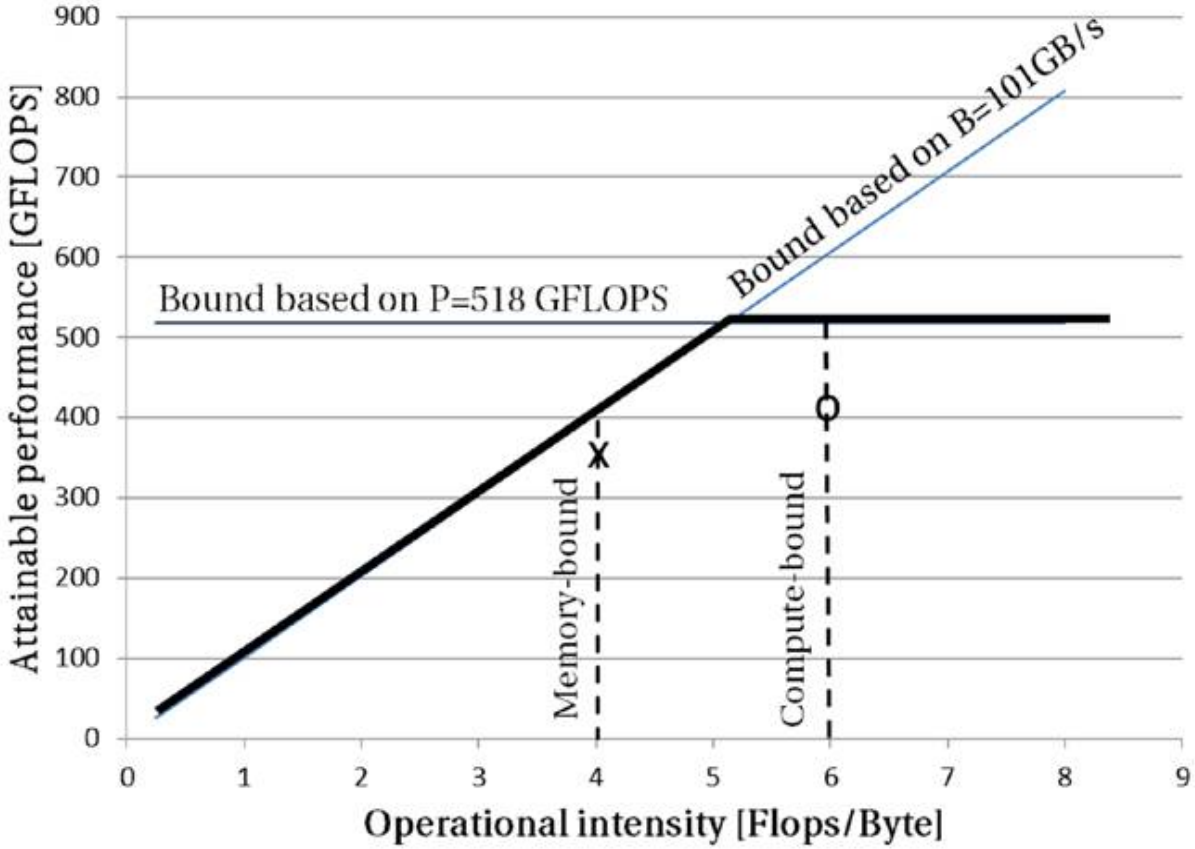
Arithmetic Intensity and BLAS..

Table 2: Basic Linear Algebra Subroutines (BLAS)

Operation	Definition	Floating point operations	Memory references	q
saxpy	$y_i = \alpha x_i + y_i,\ i = 1, \ldots, n$	$2n$	$3n+1$	$2/3$
Matrix–vector mult	$y_i = \sum_{j=1}^{n} A_{ij} x_j + y_i$	$2n^2$	$n^2 + 3n$	2
Matrix–matrix mult	$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj} + C_{ij}$	$2n^3$	$4n^2$	$n/2$

SISSA
ICTP

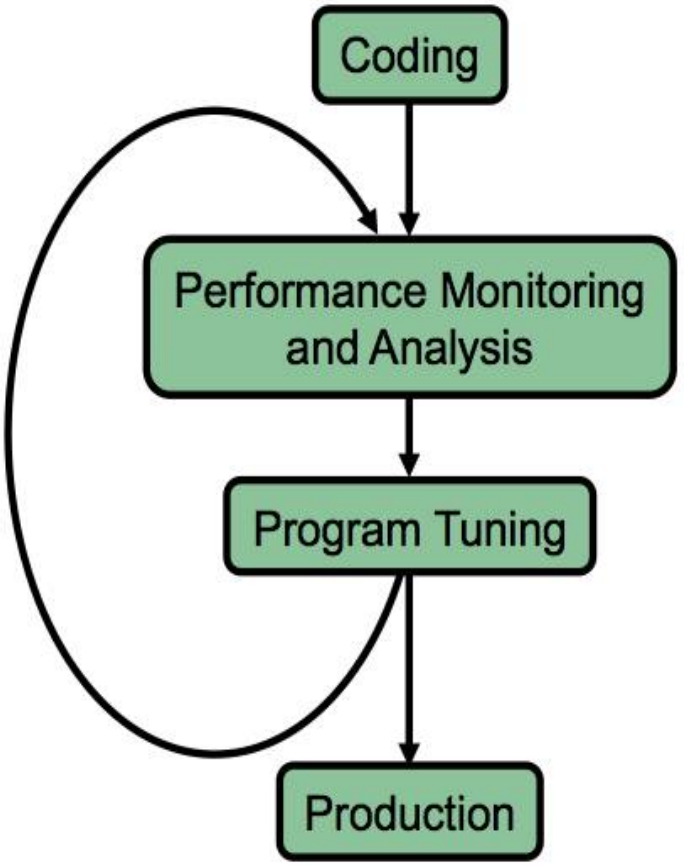
Roofline model



MHPC
Master in High Performance Computing

How to optimize...

- Iterative optimization
  - –1. Check for correct answers (program must be correct!)
  - –2. Profile to find the hotspots, e.g. most time-consuming routines
  - –3. Optimize these routines
  - –Repeat 1-3
- When satisifed go to production



SISSA ICTP

Locality of Reference

Temporal locality:

Recently referenced items (instr or data) are likely to be *referenced again* in the near future:

-iterative loops, subroutines, local variables

-*working set* concept

Spatial locality:

programs access data which is *near to each other*:

-operations on tables/arrays

-*cache line size* is determined by spatial locality

Sequential locality:

processor executes instructions in *program order*:

-branches/in-sequence ratio is typically 1 to 5

MHPC
Master in High Performance Computing

Optimization Techniques for memory

- Loop Interchanges
- Effective Reuse of Data Cache
- Loop Unrolling
- Loop Fusion/Fission
- Prefetching
- Floating Point Division

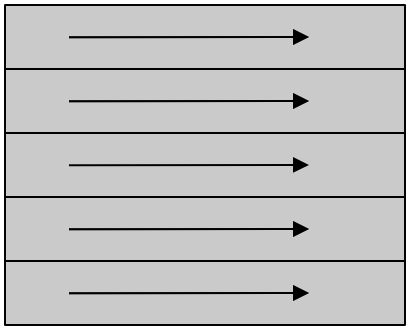
SISSA
ICTP

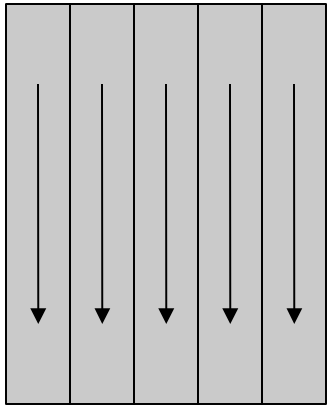
Storage in Memory

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(s)

(spatial locality: stride 1 access)

MHPC
Master in High Performance Computing

Array Indexing

There are several ways to index arrays:

```
Do j=1,M
    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

The addressing scheme can (and will) have an impact on the performance

SISSA ICTP

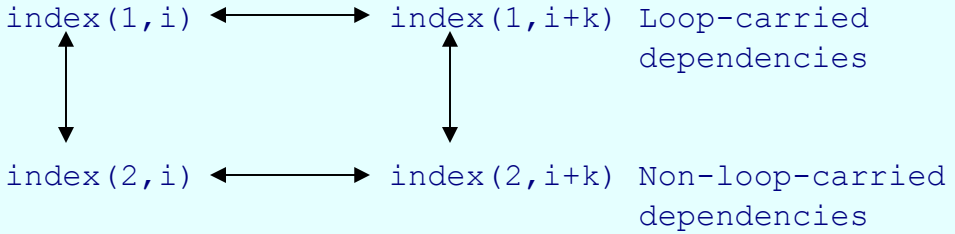
Data Dependencies

Independent instructions can be scheduled at the same time on the multiple execution units in superscalar CPU.

Independent operations can be (software) pipelined on the CPU

Loop-carried dependencies

```
do i=1,n
     a (index (1,i)) = b(i)
     a (index (2,i)) = c(i)
end do
```



Standard prog language (F77/F90/C/C_{++}) do not provide explicit information on data dependencies.

Compilers assume worse case for the data dependencies:

-problem for indirectly addressed arrays in Fortran

-problem for all pointers C

MHPC
Master in High Performance Computing

Loop Interchange

Basic idea: In a nested loop, examine and possibly change the order of the loop

Advantages:

Better memory access patterns (leading to improved cache and memory usage)

Elimination of data dependencies (to increase the opportunities for CPU optimization and parallelization)

Disadvantage:

May make a short loop innermost (which is not good for optimal performances)

Exercise: try this on your matrix-matrix multiplication program

SISSA ICTP

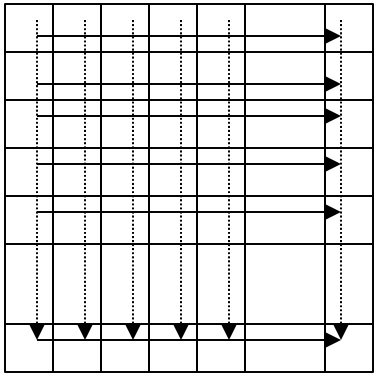
Loop Interchange - Example 1

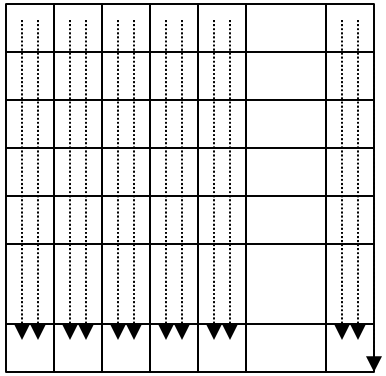
Original

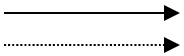
```
DO i=1,N
   DO j=1,M
     C(i,j)=A(i,j)+B(i,j)
   END DO
END O
```

Interchanged loops

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







Access order

Storage order

Loop Interchange in C

In C, the situation is exactly the opposite

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

interchange

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

index reversal

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

The performance benefit is the same in this case

In many practical situations, loop interchange is much easier to achieve than index reversal

Loop Interchange – Mnemonic rule

With row-major, the column or "rightmost" index varies most quickly (C/C+)

With column-major, the row of "leftmost" index varies most quickly.(Fortran/F90)

MHPC
Master in High Performance Computing

Loop Interchange - Example 2

```
DO i=1,300
   DO j=1,300
      DO k=1,300
         A (i,j,k) = A (i,j,k)+ B (i,j,k)* C (i,j,k)
      END DO
   END DO
END DO
```

orderLoop	2.4GHz(IBM)x335	1.4GHz(IBM)x330
kji	78	69
jki	67	26
kij	2	62
ikj	50	21
jik	90	51
ijk	40	21

Timings are in seconds

SISSA

ICTP

MHPC
Master in High Performance Computing

Loop Interchange Compiler Options

- Compilers accept option to automatically play this trick
- Gnu compiler

  –-floop-interchange

TEST IF WHAT THEY CLAIM TO DO IS WHAT THEY ACTUALLY DO

SISSA ICTP

MHPC
Master in High Performance Computing

Cache Thrashing

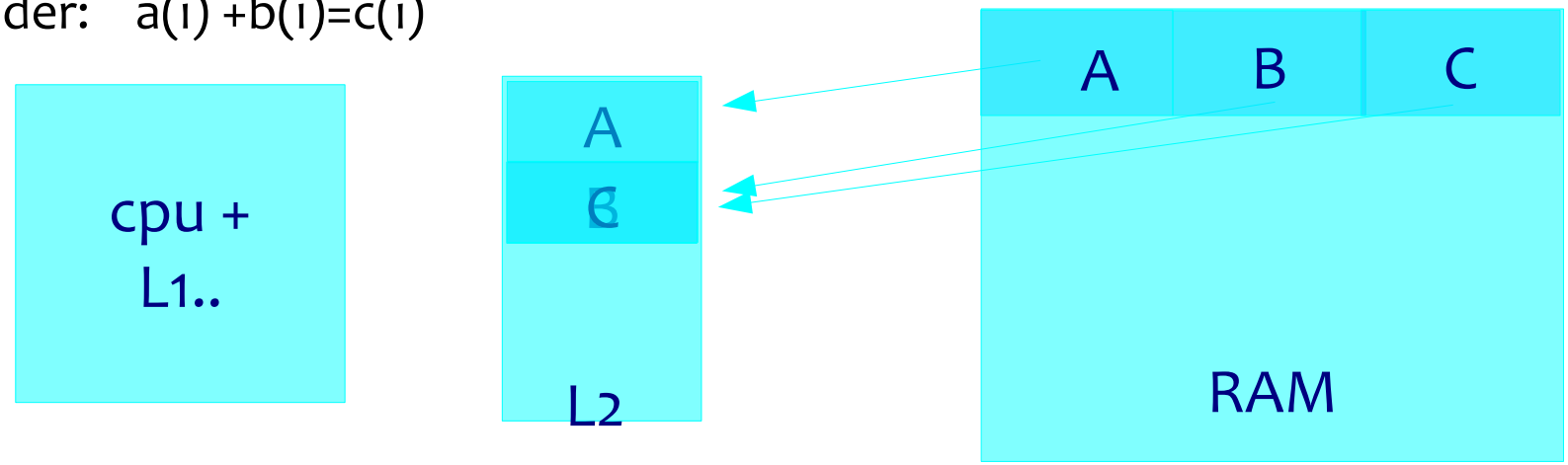
Typical problem in code performance is *cache thrashing*.

Cache trashing happens when data in cache are rewritten and fully reused.

In the previous case, the cache thrashing was minimized by loop interchange.

Another optimization technique aimed at minimizing cache thrashing is *COMMON block padding*.

consider: a(i) +b(i)=c(i)



MHPC
Master in High Performance Computing

Prefetching

Prefetching is the retrieval of data from memory to cache before it is needed in an upcoming calculation. This is an example of general optimization technique called latency hiding in which communications and calculations are overlapped and occur simultaneously.

The actual mechanism for prefetching varies from one machine to another.

When using GNU:

-fprefetch-loop-arrays

MHPC
Master in High Performance Computing

Loop Unrolling

Loop unrolling is an optimization technique which can be applied to loops which perform calculations on array elements.

Consists of replicating the body of the loop so that calculations are performed on several array elements during each iteration.

Reason for unrolling is to take advantage of pipelined functional units. Consecutive elements of the arrays can be in the functional unit simultaneously.

Programmer usually does not have to unroll loops "by hand" -- compiler options and directives are usually available. Performing unrolling via directives and/or options is preferable

Code is more portable to other systems

Code is self-documenting and easier to read

Loops with small trip counts or data dependencies should not be unrolled!

SISSA ICTP

Loop Unrolling Compiler Options

- Gnu
- -funroll-loops
  - Enable loop unrolling
- -funroll-all-loops
  - Unroll all loops; not recommended

Loop Unrolling Example

- Normal loop

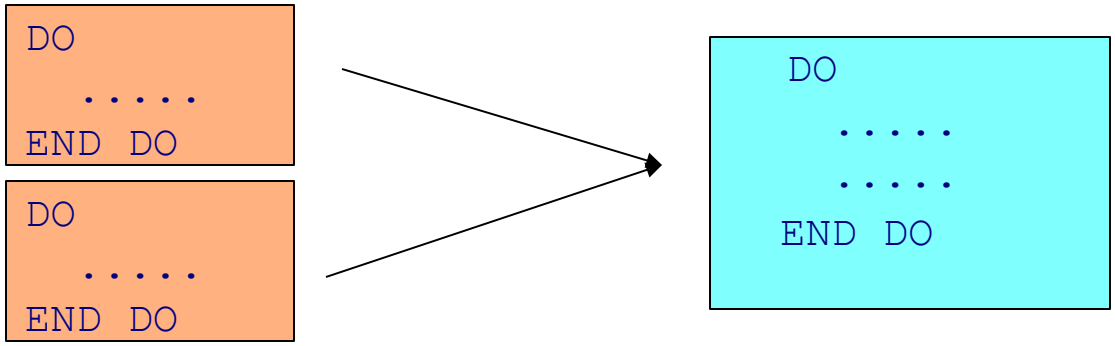
```
do i=1,N
   a(i)=b(i)+x*c(i)
enddo
```

- Manually unrolled loop

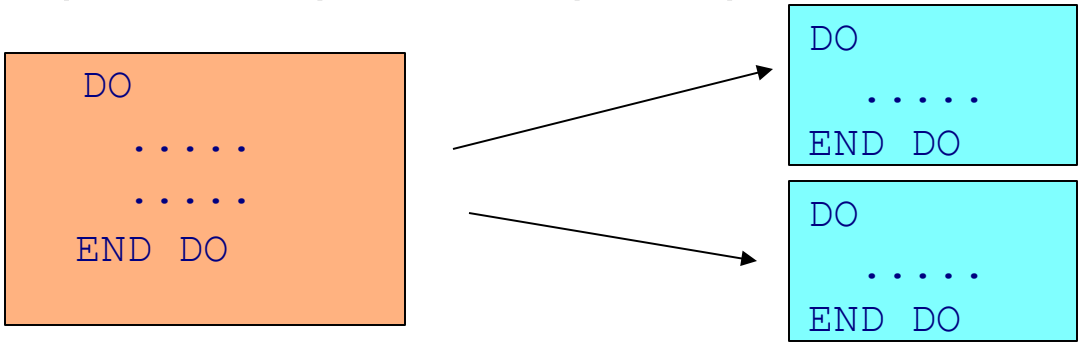
```
do i=1,N,4
   a(i)=b(i)+x*c(i)
   a(i+1)=b(i+1)+x*c(i+1)
   a(i+2)=b(i+2)+x*c(i+2)
   a(i+3)=b(i+3)+x*c(i+3)
enddo
```

Loop Fusion and Fission

Fusion: Merge multiple loops into one



Fission: Split one loop into multiple loops



MHPC
Master in High Performance Computing

Example of 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
```

Potential for Fusion: dependent operations in separate loops

Advantage:

Re-usage of array B()

Disadvantages:

In total 4 arrays now contend for cache space

More registers needed

Example of Loop Fission

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

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

```
DO ii=1,N
   D(ii)=D(ii-1)+C(ii)
END DO
```

Potential for Fission: independent operations in a single loop

Advantage:

First loop can be scheduled more efficiently and be parallelised as well

Disadvantages:

Less opportunity for out-of-order superscalar execution

Additional loop created (a minor disadvantage)

MHPC
Master in High Performance Computing

Blocking for cache (tiling)

Blocking for cache is:

-An optimization that applies for datasets that do not entirely fit in the (2nd level) data cache

-A way to increase spatial locality of reference i.e. exploit full cache lines

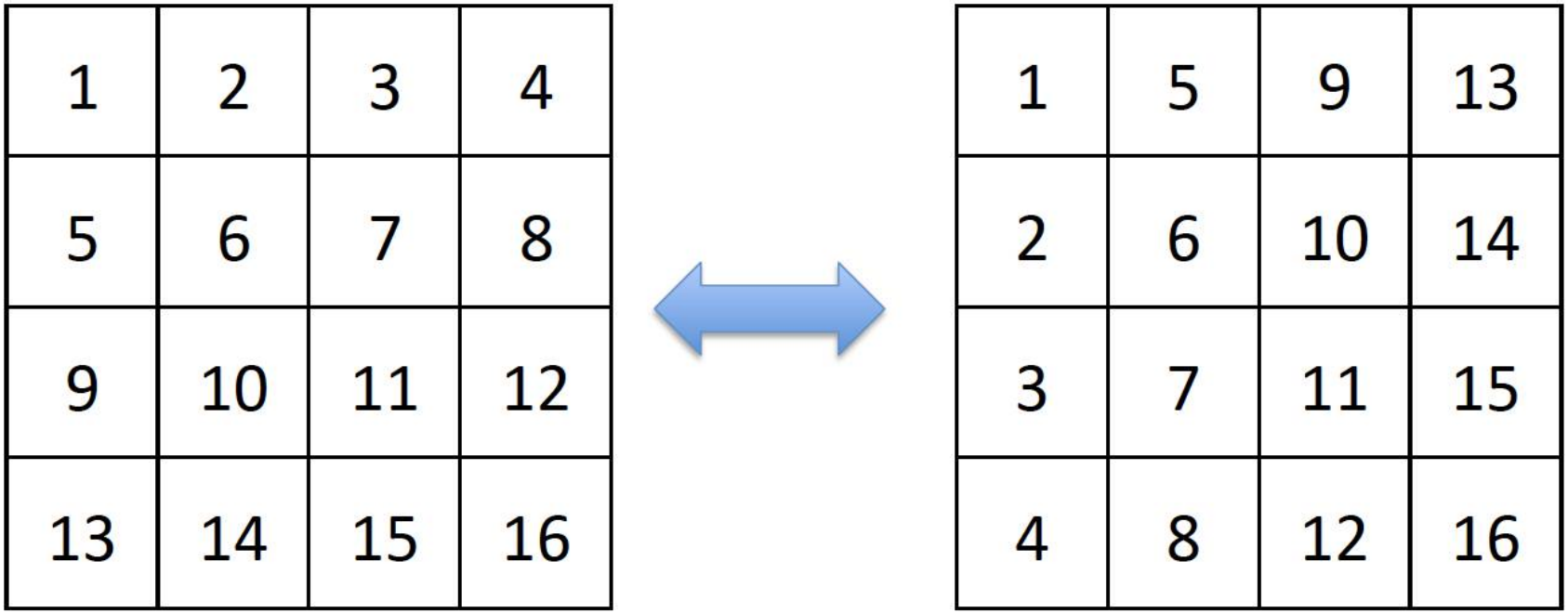
-A way to increase temporal locality of reference i.e. Improves data re-usage

By way of example, let discuss the transpose of a matrix...

```
do i=1,n
  do j=1,n
     a(i,j)=b(j,i)
  end do
end do
```


SISSA ICTP

Transpose



1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

1	2
5	6

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

Block algorithm for transposing a matrix:

block data size= bsize

mb=n/bsize

nb=n/bsize

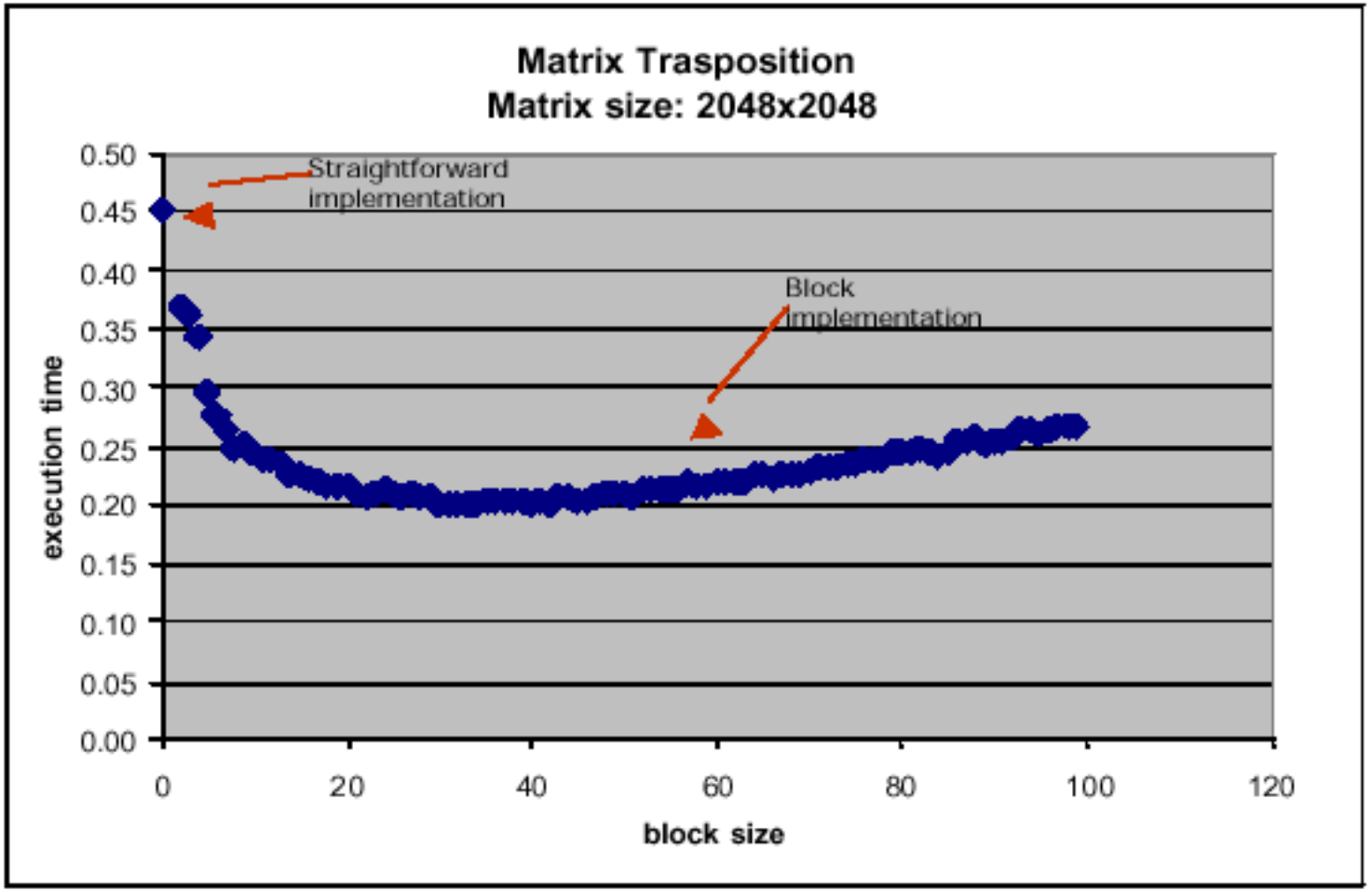
Code is a little bit more complicated if

MOD(n,bize) is not zero

MOD(m,bize) is not zero

```
do ib = 1, nb
 ioff = (ib-1) * bsiz
 do jb = 1, mb
   joff = (jb-1) * bsiz
    do j = 1, bsiz
      do i = 1, bsiz
        buf(i,j) = x(i+ioff, j+joff)
     enddo
    enddo
    do j = 1, bsiz
      do i = 1, j-1
       bswp = buf(i,j)
       buf(i,j) = buf(j,i)
       buf(j,i) = bswp
      enddo
     enddo
    do i=1,bsiz
      do j=1,bsiz
       y(j+joff, i+ioff) = buf(j,i)
      enddo
    enddo
  enddo
enddo
```

Results... (Carlo Cavazzoni data)



Optimizing Matrix Multiply for Caches

- Several techniques for making this faster on modern processors
  - heavily studied
- Some optimizations done automatically by compiler, but can do much better
- In general, you should use optimized libraries (often supplied by vendor) for this and other very common linear algebra operations
  - BLAS = Basic Linear Algebra Subroutines
- Other algorithms you may want are not going to be supplied by vendor, so need to know these techniques

Conclusions

- Wall in memory should be taken into account if you are looking for performance
  - –SMP is not always valid: NUMA
  - –not only RAM is shared but also L2/L3 Caches