The 17th Annual International Symposium on High Performance Computing Systems and Applications

17ième symposium international sur les applications et systèmes de calcul de haute performance

Code Optimization

Presented by

Carol Gauthier

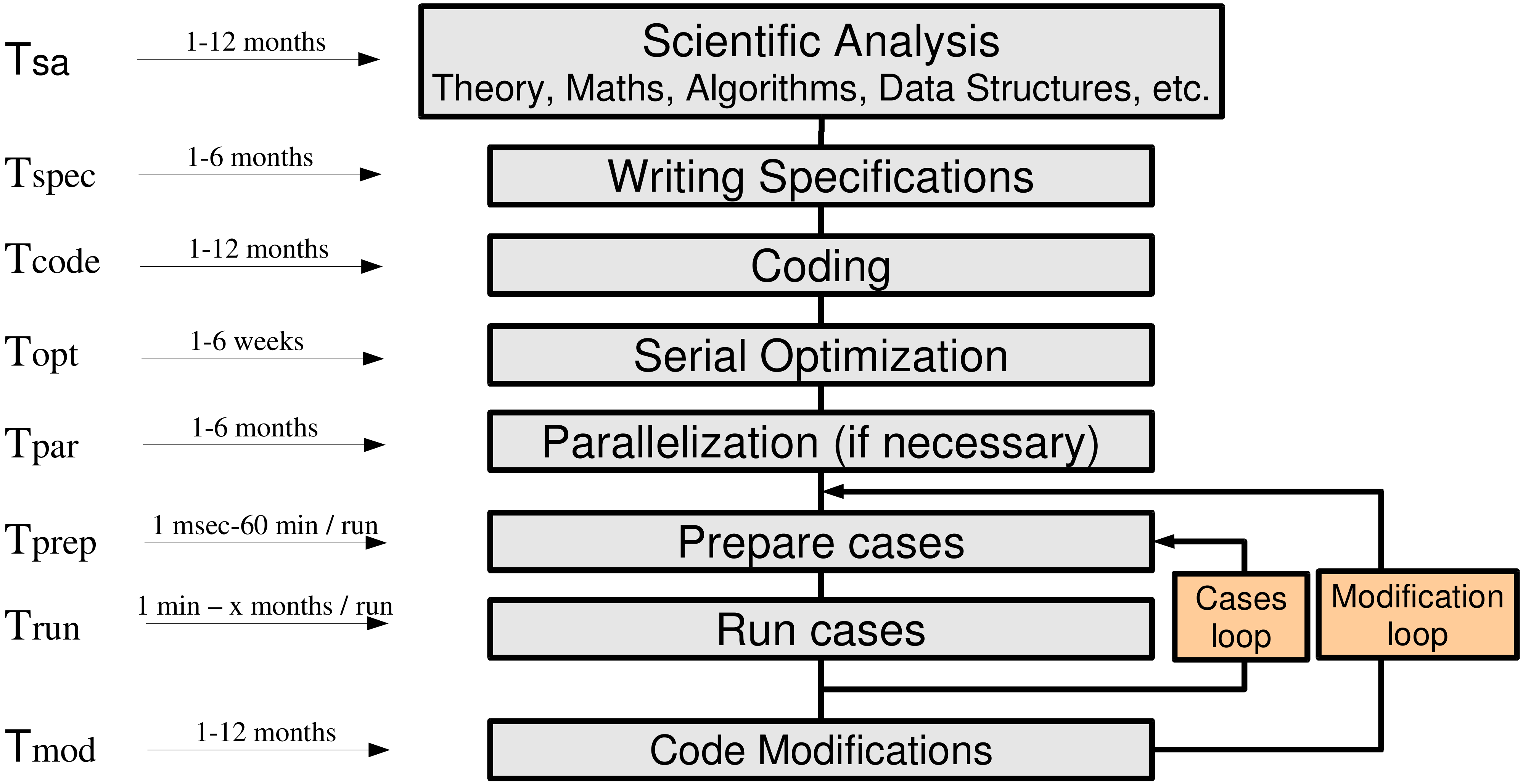
Centre de Calcul Scientifique (CCS)
Université de Sherbrooke

Réseau québécois de calcul de haute performance (RQCHP)

Summary of the presentation

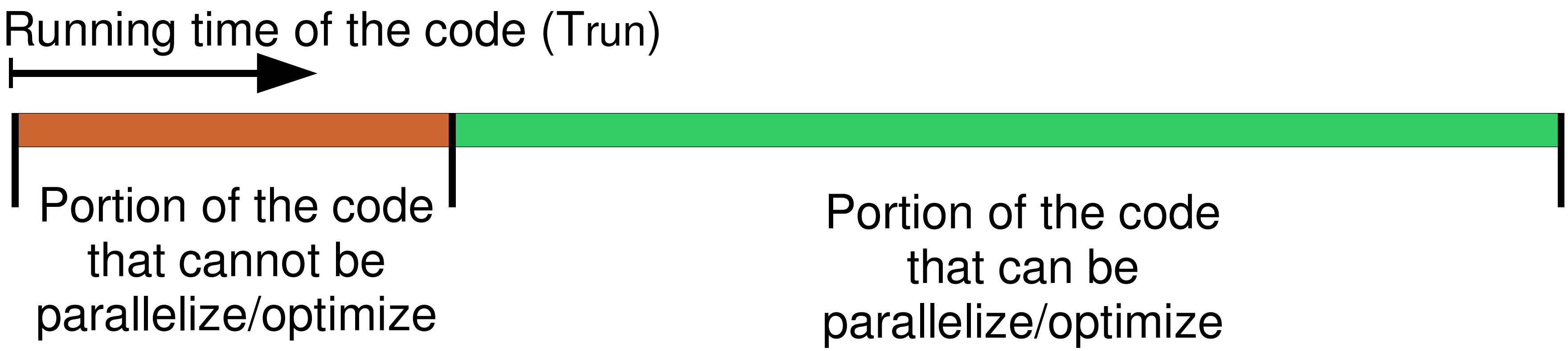
Overall Performance Strategy

Scientific Application Developpment Steps

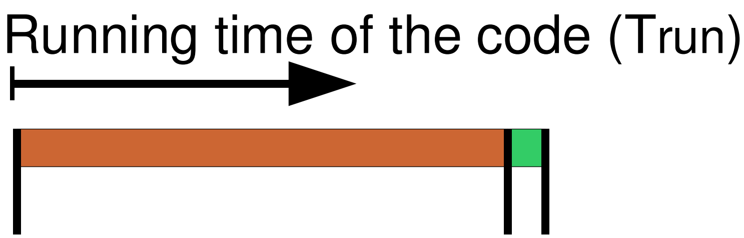


Overall Performance Strategy

Amdahl's Law



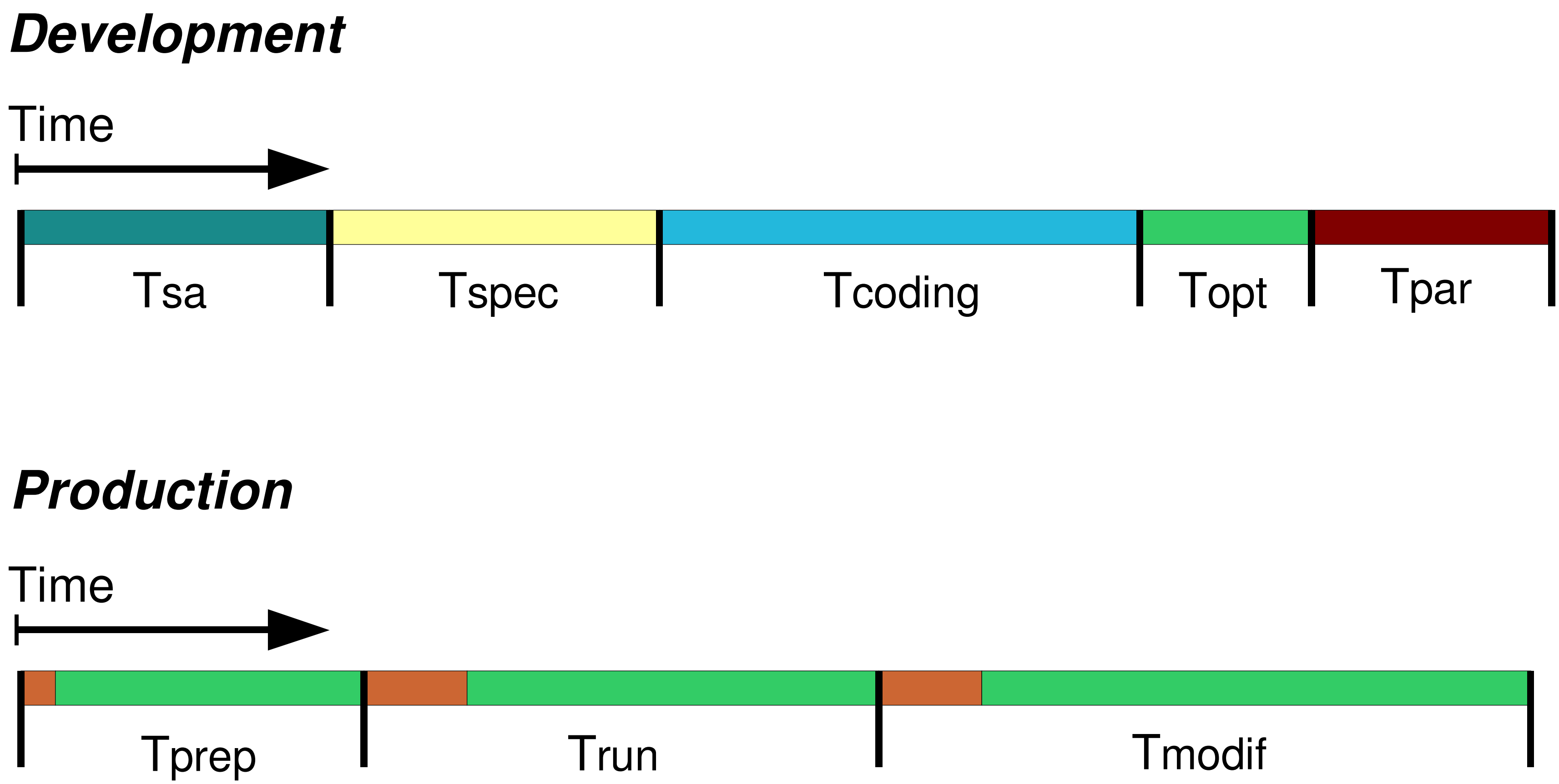
Even after an excellent parallelisation or optimisation...



.... this code will never run more than 3 times faster !!!

Overall Performance Strategy

Overview of Development and Production Time



Carol Gauthier, Centre de Calcul Scientifique (CCS), Université de Sherbrooke

Goals of Serial Optimization

1) Make your code run faster

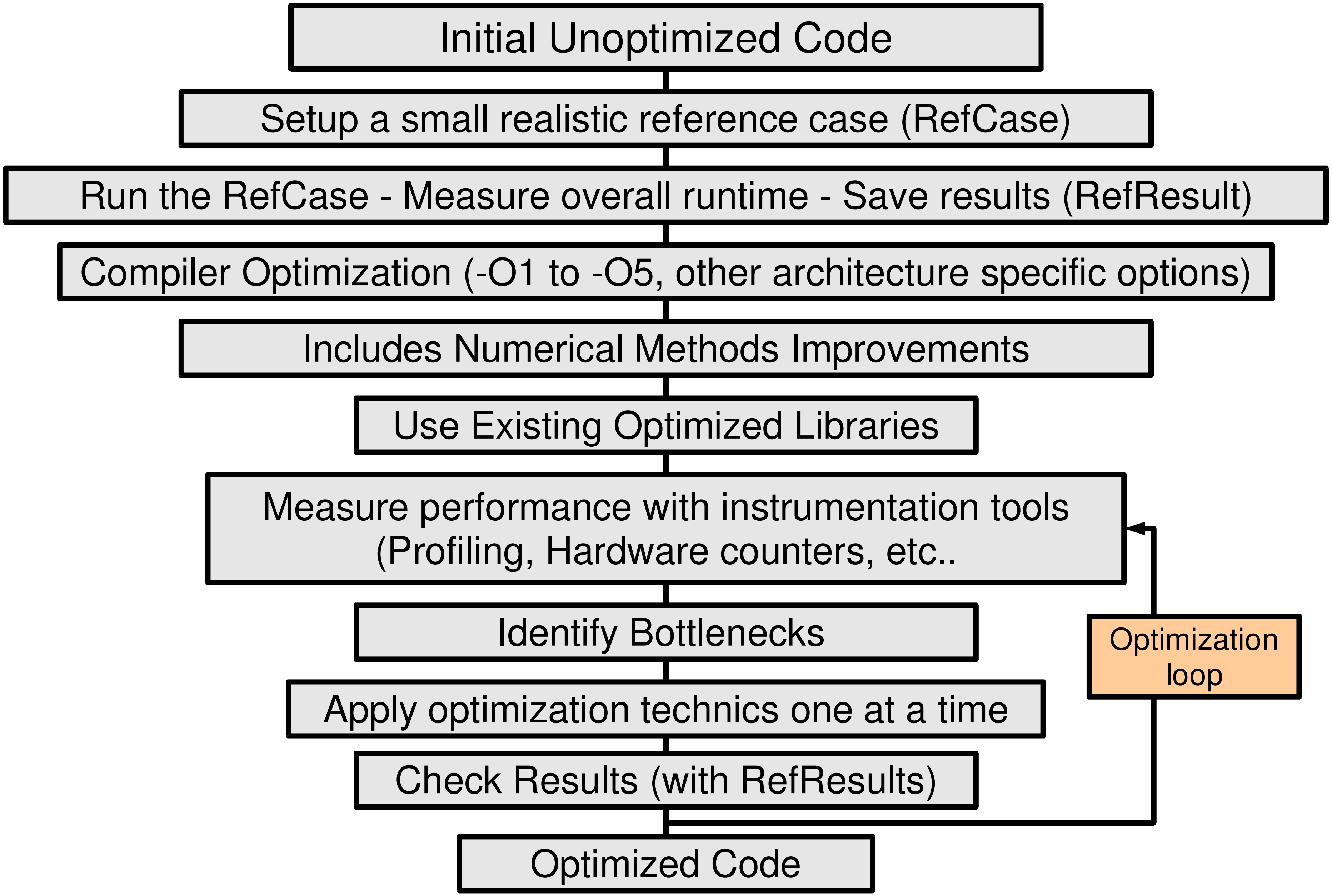
2) Make your code use less RAM

3) Make your code use less disk space

Consequently runs more and/or larger cases

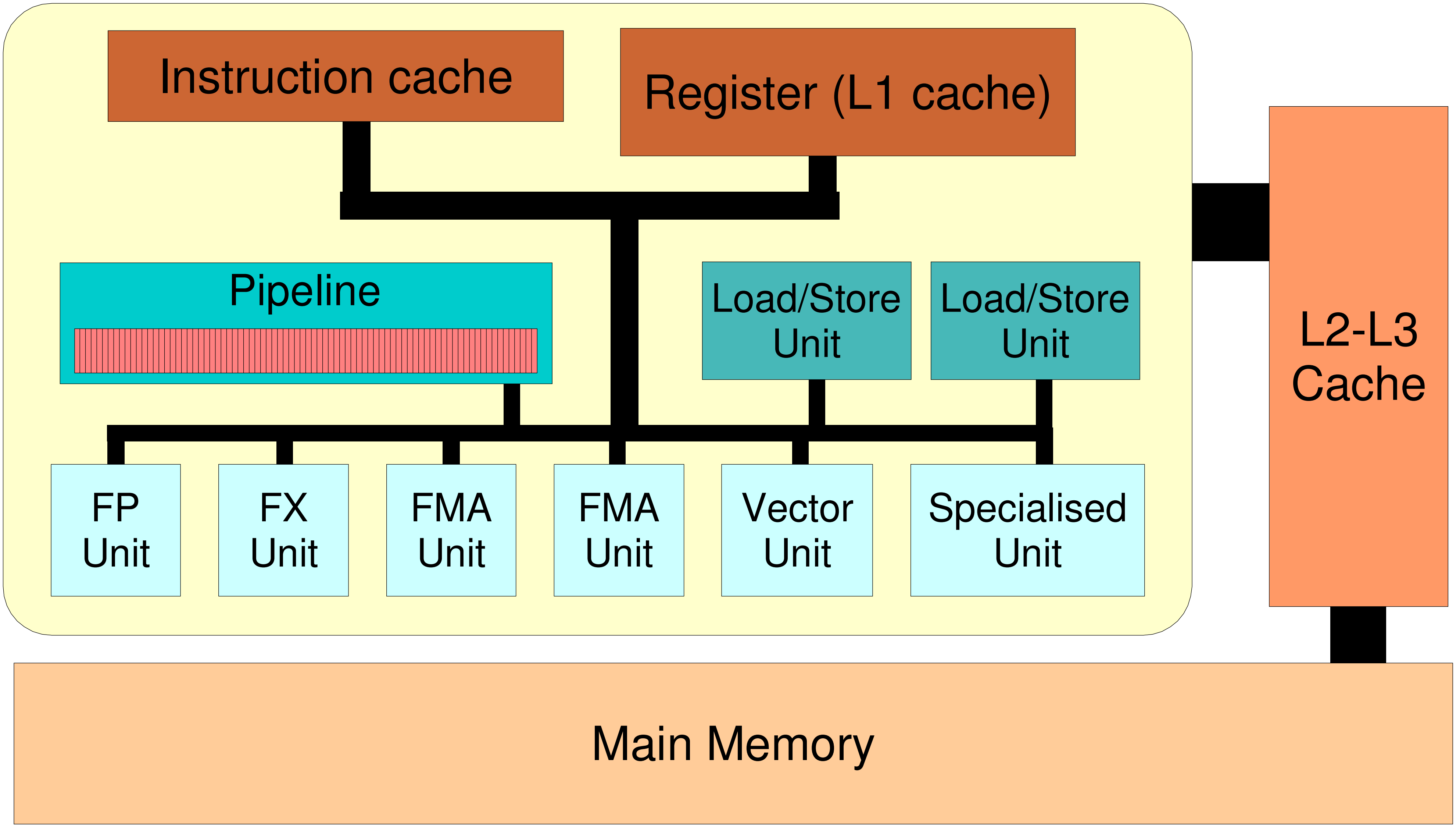
Optimization always comes before Parallezation

Optimization Strategy



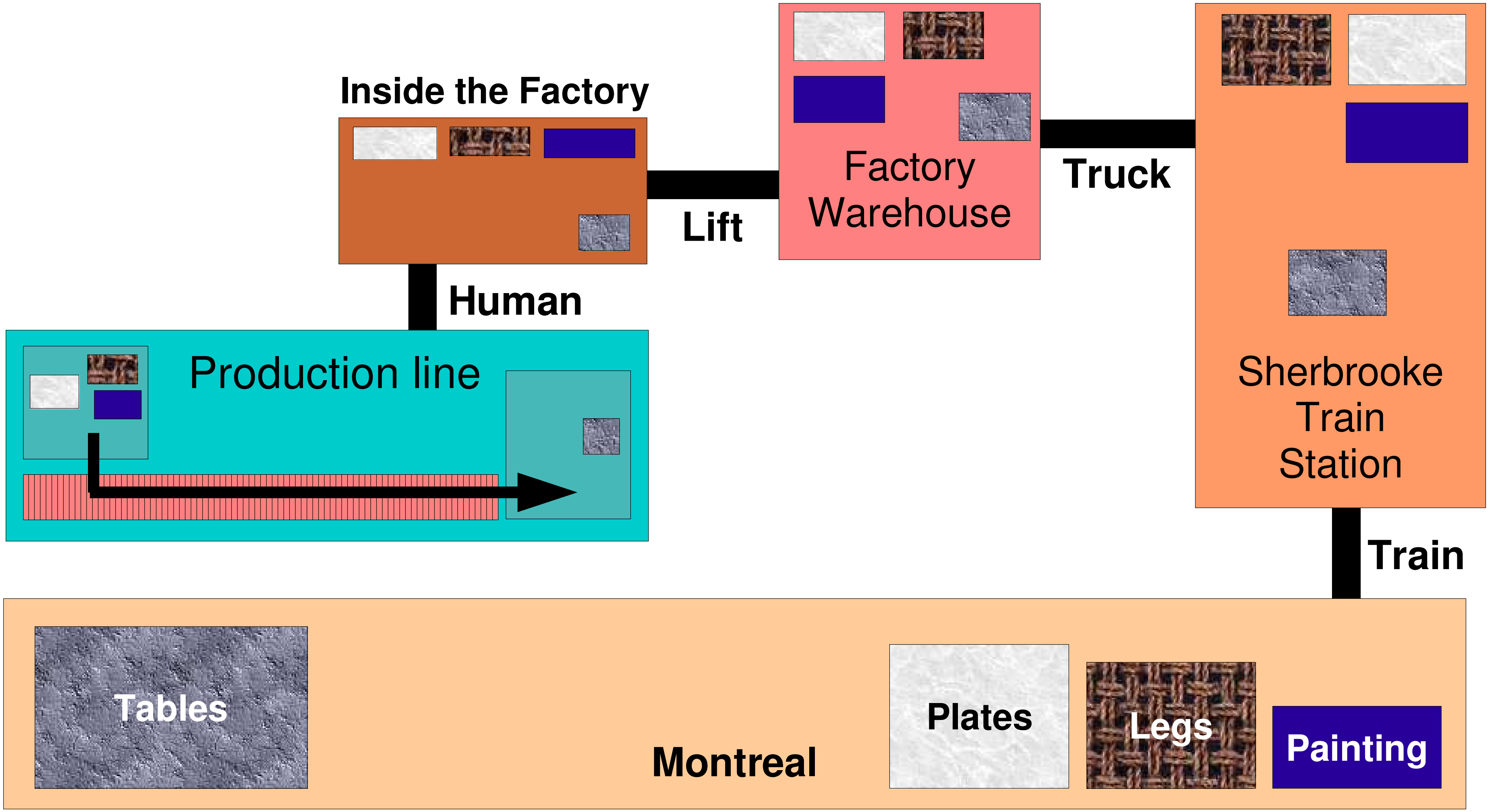
Hardware Limits and Constraints

CPU Insights



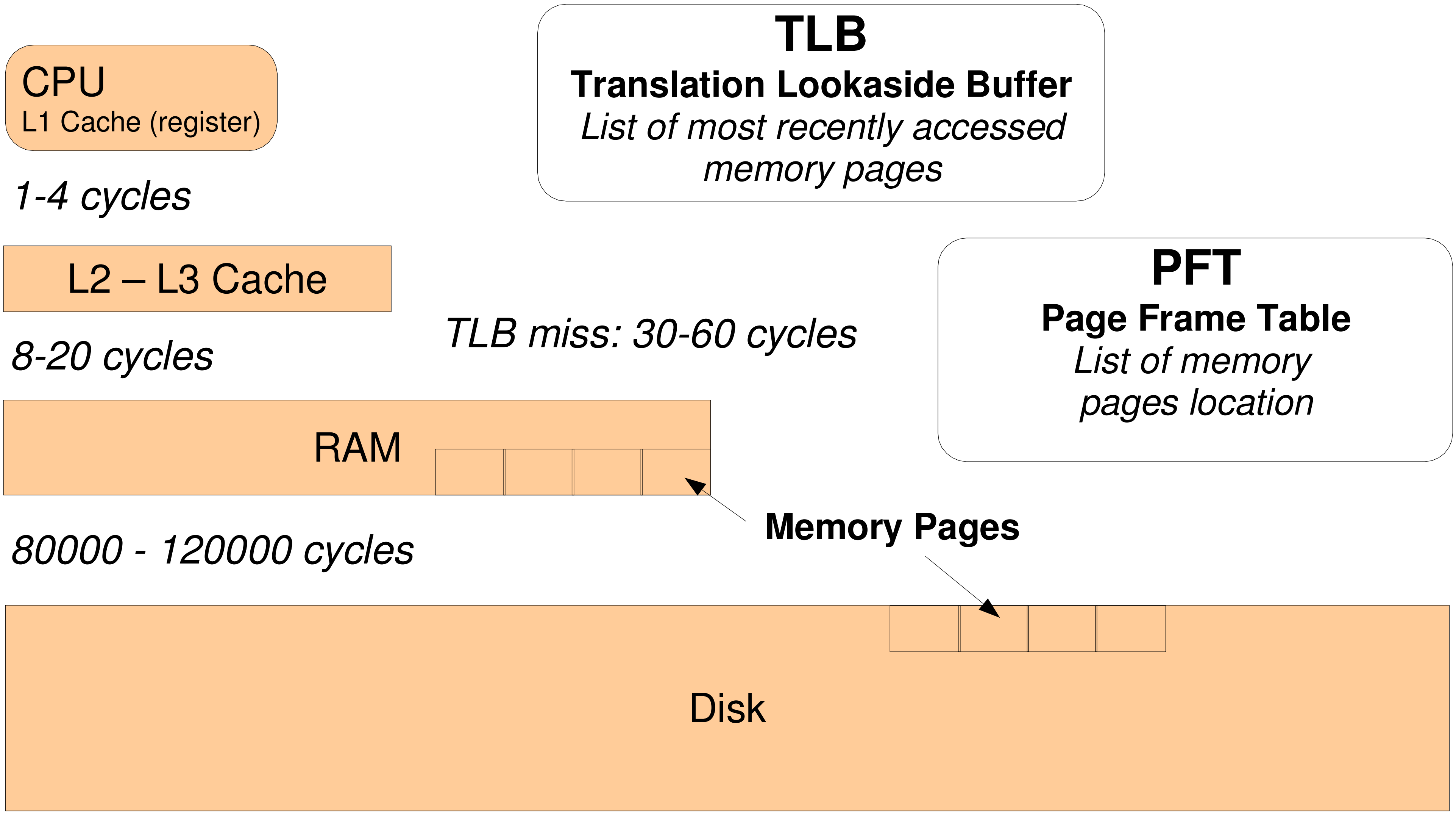
Hardware Limits and Constraints

An Analogy: The Process of Building and Delivering Tables!



Hardware Limits and Constraints

Memory Access vs Clock Cycles



Measuring Performance

1) Measurement Guidelines
2) Time Measurement
3) Profiling tools
4) Hardware Counters

Measuring Performance

Measurement Guidelines

- *Make sure your have full access to the processor*
- *Always check the correctness of the results*
- *Be prepare to instrument the code*
- *Use all the tools available*
  - *- Time measurement, Profiling, Hardware counters*
- *Watch out for overhead induced by instrumentation*
- *Compare to theoritical peak performance*

Measuring Performance

Time Measurement

Overall running time using the standard UNIX command

On Linux for example

```
[localhost /]$ time tree
3.48user 12.64system 1:50.36elapsed 14%CPU
```

Time spent in specific parts of the code

In Fortran 90 for example

```
Call SYSTEM_CLOCK(count1,count_rate,count_max)
    ... calculation
Call SYSTEM_CLOCK(count2,count_rate,count_max)
```

Measuring Performance

Profiling tools : prof, gprof, tprof, etc..

Tells you the portion of time the program spends in each of the subroutines and/or functions

Mostly useful when your program has a lot of subroutines and/or functions

Use profiling at the begining of optimization process

Measuring Performance

Hardware Counters

All modern processors has buit-in event counters

Processors may have several registers reserved for counters (meaning several types of events can be counted simultaneously)

It is possible to start, stop and reset counters at will

Many types of events can be counted (most not interresting to us!)

Software API to access counters are now available!

Using Hardware Counters is a must in Optimization!!!

Measuring Performance

Hardware Counters : PAPI

Performance **A**pplication **P**rogramming **I**nterface

A standerdized API to access hardware counters

Available on most systems
Linux, True64, Unicos, AIX, Windows NT(2000,XP), Irix, Solaris

Motivation behind PAPI (extracted from PAPI User's Guide)
- To provide solid foundation for cross platform performance analysis tools
- To present a set of standard definitions for performance metrics
- To provide a standardize API among users, vendors and academics
- To be easy to use, well documented, and freely available

For detailed documentation visit the PAPI Web site
http://icl.cs.utk.edu/projects/papi/

Optimization Technics

1) Compiler Options
2) Use Existing Libraries!
3) Numerical instabilities and convergence
4) FMA units
5) Vector Units
6) Array Considerations in C and Fortran
7) Optimization Tips & Tricks

Optimization Technics

Compiler Options

Substantial gain can be easily obtained by playing with compiler options

Optimization options are “a must”. The first and second level of optimization will rarely give no benefits!

Optimization options can range from -O1 to -O5 with some compilers. -O3 to -O5 might lead to slower code, so try them independently on each subroutine.

Always check your results when trying optimization options.

Compiler options might include hardware specifics such as accessing vector units for example.

Optimization Technics

Compiler Options

GNU C compiler

```
gcc
   -O0 -O1 -O2 -O3 -finline-functions ...
```

IBM AIX Fortran and C compilers

```
xlf and xlC
   -O1 -O2 -O3 -O4 -O5 -qstrict -qipa=level ...
```

Intel Fortran and C compiler

```
ifc and icc
   -O0 -O1 -O2 -O3 -ip -xW -tpp7(for P4) -ip ...
```

Optimization Technics

Use Existing Libraries !

Existing libraries are usually highly optimized

Try several libraries and compare if possible

Recompile libraries on the plateform your running if you have the sources (trying compiler optimization)

Vendors libraries are usually well optimized for the their platform, but comparing with other libraries is always a good thing

Some popular mathematical libraries:
BLAS, LAPACK, ESSL, ESSL, MASS, FFTW,....

Optimization Technics

Numerical instabilities and convergence problem

Specific to each problem

Could lead to much longer run time

Could even worstly lead to wrong results!

Reexamine the mathematics of the solver

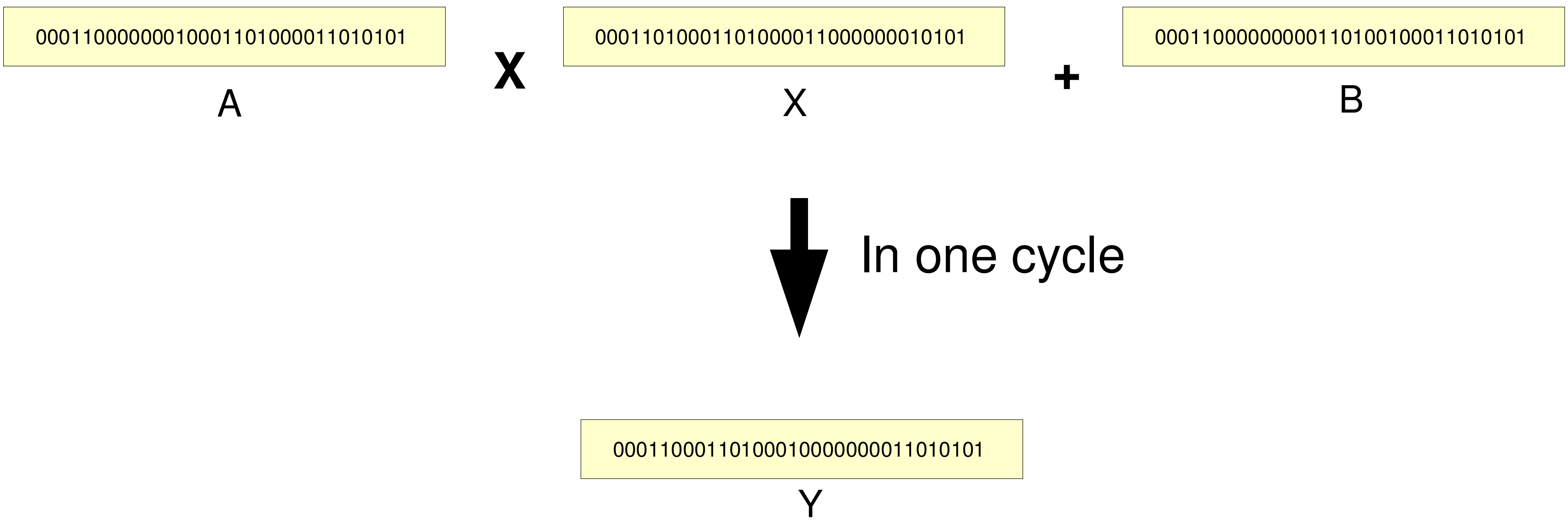
Look for operations involving very large and very small numbers

Be careful when using higher compiler optimization options

Optimization Technics

FMA Units

$$Y = A*X + B$$



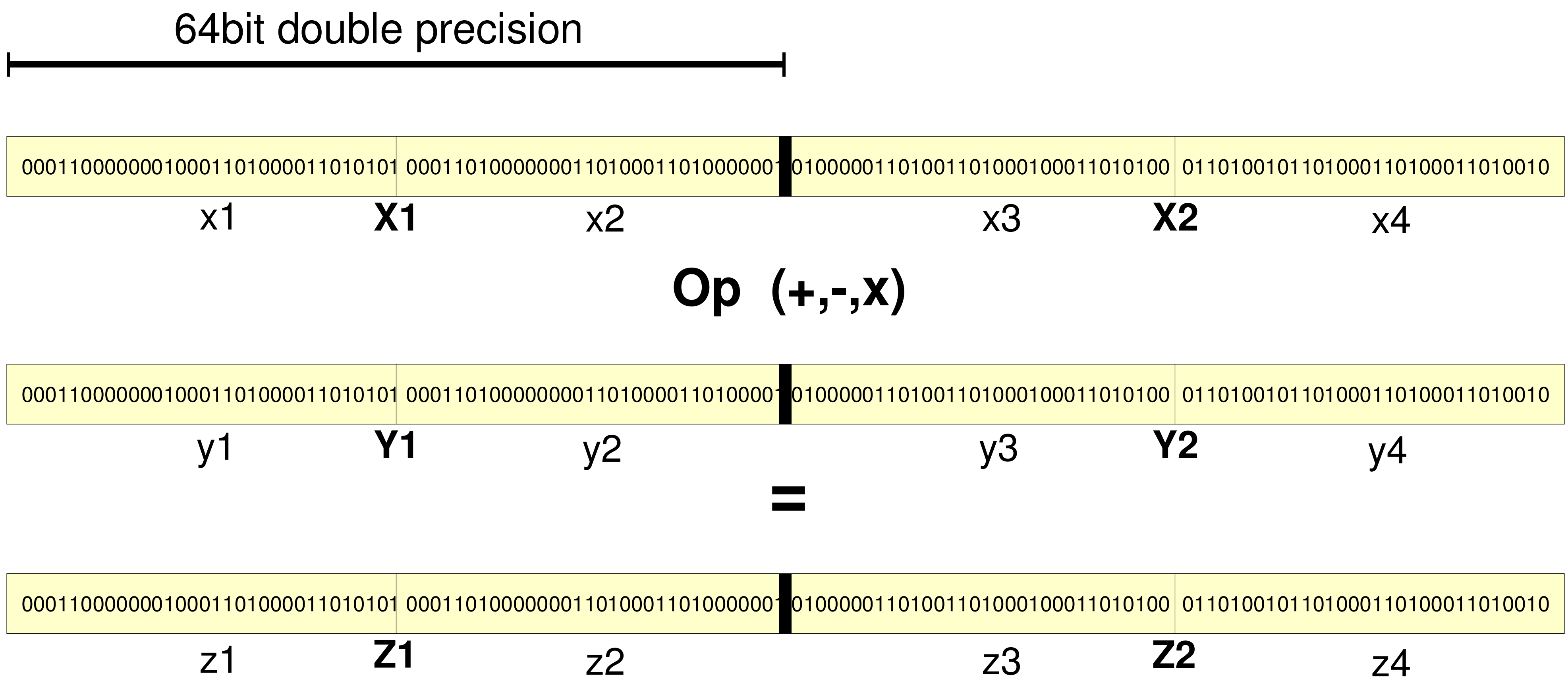
Optimization Technics

Vector Units

128bit long Vector unit (P4, Opteron)

4 single precision FLOPs / cycle
2 double precision FLOPs / cycle

32bit single precision



Optimization Technics

Array Considerations in Fortran and C/C++

In Fortran

```
do i=1,5
 do j=1,5
   a(i,j)=...
 end do
end do
```

→

```
do j=1,5
 do i=1,5
   a(i,j)=...
 end do
end do
```

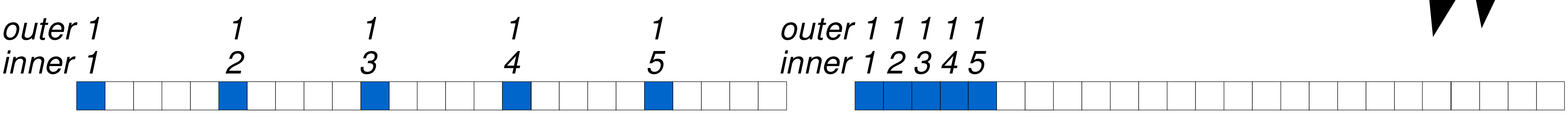
In C/C++

```
for(j=1;j<=5;j++){
 for(i=1;i<=5;i++){
   a[i][j]=...
 }
}
```

→

```
for(i=1;i<=5;i++){
 for(j=1;j<=5;j++){
   a[i][j]=...
 }
}
```

Corresponding memory representation



Optimization Technics

Tips & Tricks: Sparse Arrays

Its harder to optimize because of often unavoidable jumps when accessing memory

Try to minimize the memory jumps, they could be very costly because of cache and TLB misses

Carefully analyse the construction of the sparse arrays vs the way it is used in the code.

Lower your expectations!

Optimization Technics

Tips & Tricks: Minimize the number of operations !

When doing optimization, one of the first thing to do, is reducing the number of unnecessary operations performed by the CPU!

An obvious example!

```
do k=1,10
 do j=1,5000
  do i=1,5000
   a(i,j,k)= 3.0*m*d(k)+
             c(j)*23.1 -
             b(i)
  end do
 end do
end do
```

→

```
do k=1,10
 dtmp(k)=3.0*m*d(k)
 do j=1,5000
  ctmp(j)=c(j)*23.1
  do i=1,5000
   a(i,j,k)= dtmp(k)+
             ctmp(j)-
             b(i)
  end do
 end do
end do
```

1250 Millions of operations

500 Millions of operations

Optimization Technics

Tips & Tricks: Complex Numbers

Watch for operations on complex numbers that have Imaginary or Real part equal to zero. This is again a question of minimizing the number of operations.

```
! Real part of a elements = 0
complex*16 a(1000,1000),b
complex*16 c(1000,1000)

do j=1,1000
 do i=1,1000
  c(i,j)= a(i,j)*b
 end do
end do
```

→

```
real*8 aI(1000,1000)
complex*16 b,c(1000,1000)

do j=1,1000
 do i=1,1000
  c(i,j)=(-IMAG(b)*aI(i,j),
          aI(i,j)*REAL(b) )
 end do
end do
```

6 Millions of operations

2 Millions of operations

Optimization Technics

Tips & Tricks: Loop Overhead

```
do j=1,1000000
  do i=1,1000000
    do k=1,2
      a(i,j,k)=b(i,j)+c(k)
    end do
  end do
end do
```

→

```
do j=1,1000000
  do i=1,1000000
      a(i,j,1)=b(i,j)+c(1)
      a(i,j,2)=b(i,j)+c(2)
  end do
end do
```

Tips & Tricks: Objects declarations and instanciations

In Object-Oriented Languages ***AVOID*** objects declarations and instanciations within the most inner loops

Optimization Technics

Tips & Tricks: Function Call Overhead

```
do k=1,10000
 do j=1,10000
  do i=1,5000
   a(i,j,k)=f1(c(i),b(j),k)
  end do
 end do
end do

function f1(x,y,m)
  real*8 x,y,tmp
  integer m
  tmp=x*m - y
  return tmp
end
```

```
do k=1,10000
 do j=1,10000
  do i=1,5000
   a(i,j,k)=c(i)*k - b(j)
  end do
 end do
end do
```

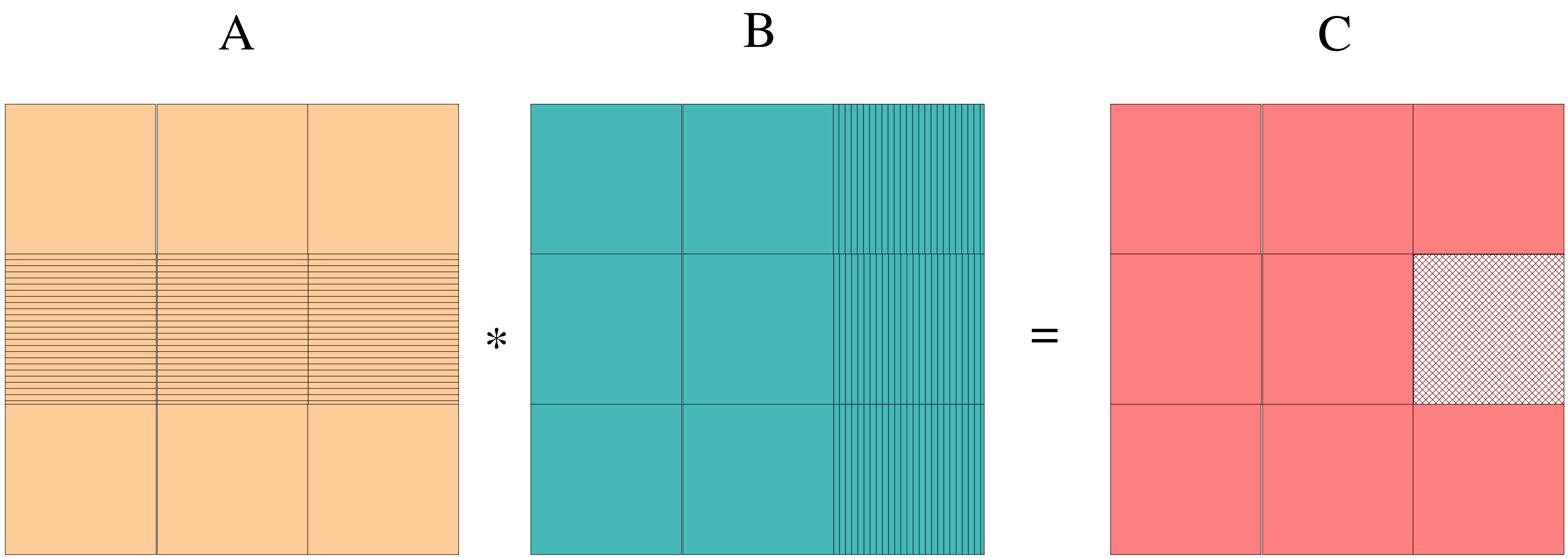
This can also be achieve with compilers inlining options. The compiler will then raplace all function calls by a copy of the function code, leading sometimes to very large binary executables.

```
xlf -qipa=inline
ifc -ip
icc -ip
gcc -finline-functions
```

Optimization Technics

Tips & Tricks: Blocking

Blocking is used to reduce cache and TLB misses in nested Matrix operations. The idea is to process as much as possible the data that is brought in the cache.




```
C(i,j)=C(i,j)+A(i,k)*B(k,j)
```

Optimization Technics

Tips & Tricks: Blocking

```
do i=1,n
 do j=1,n
  do k=1,n
   C(i,j)=C(i,j)
               +A(i,k)*B(k,j)
  end do
 end do
end do
```

```
do ib=1,n,bsize
 do jb=1,n,bsize
  do kb=1,n,bsize
   do i=ib,min(n,ib+bsize-1)
    do j=jb,min(n,jb+bsize-1)
     do k=kb,min(n,kb+bsize-1)
       C(i,j)=C(i,j)
                     +A(i,k)*B(k,j)
     end do
    end do
   end do
  end do
 end do
end do
```

Optimization Technics

Tips & Tricks: Loop Fusion

The main advantage of Loop Fusion is the reduction of cache misses when the same array is used in both loops. It also reduces loop overhead and allow a better control of multiple instructions in a single cycle, when hardware allows it (2 FMA or 2 vector operations for exemple).

```
do i=1,100000
 a = a + x(i) + 2.0*z(i)
end do

do j=1,100000
 v = 3.0*x(j) - 3.14159267
end do
```

→

```
do i=1,100000
 a = a + x(i) + 2.0*z(i)
 v = 3.0*x(i) - 3.14159267
end do
```

Optimization Technics

Tips & Tricks: Loop Unrolling

The main advantage of Loop Unrolling is to reduce or eliminate data dependencies in loops. This is particularly useful when using an architecture with 2 FMA Units (IBM Power3-4) or a Vector unit (SSE2 extensions)

```
do i=1,1000
 a = a + x(i)*y(i)
end do
```

→

```
do i=1,1000,4
 a = a + x(i)*y(i)
       + x(i+1)*y(i+1)

       + x(i+2)*y(i+2)
        +x(i+3)*y(i+3)
end do
```

1000 cycles at best ***2 FMAs*** ***750 cycles at best***

Optimization Technics

Tips & Tricks: Loop Unrolling

The main advantage of Loop Unrolling is to reduce or eliminate data dependencies in loops. This is particularly useful when using an architecture with 2 FMA Units (IBM Power3-4) or a Vector unit (SSE2 extensions)

```
do i=1,1000
 a = a + x(i)*y(i)
end do
```

→

```
do i=1,1000,4
  a=a+  x(i)*y(i)
      + x(i+1)*y(i+1)
     +  x(i+2)*y(i+2)
       +x(i+3)*y(i+3)
end do
```

2000 cycles at best ***Vecor Unit (length 2)*** ***1250 cycles at best***

Optimization Technics

Tips & Tricks: Sum Reductions

Sum Reductions is another way of reducing or eliminating data dependencies in loops. It is more explicite than the Loop Unrolling method.

```
do i=1,1000
 a = a + x(i)*y(i)
end do
```

→

```
do i=1,1000,4
 a1 = a1 + x(i)*y(i)
 a2 = a2 + x(i+1)*y(i+1)
 a3 = a3 + x(i+2)*y(i+2)
 a4 = a4 + x(i+3)*y(i+3)
end do
a = a1 + a2 + a3 + a4
```

1000 cycles at best ***2 FMAs*** ***503 cycles at best***

```
do i=1,1000
 a = a + x(i)*y(i)
end do
```

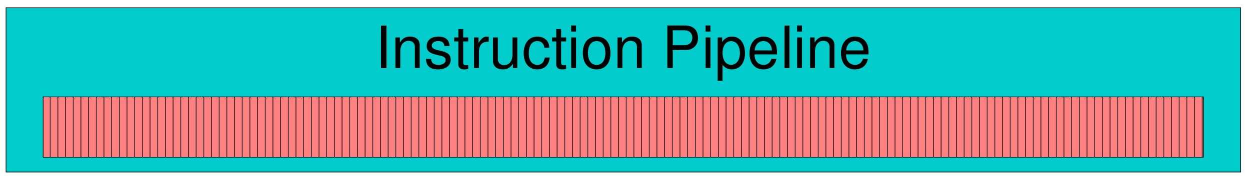
→

```
do i=1,1000,4
 a1 = a1 + x(i)*y(i)
         + x(i+1)*y(i+1)
 a2 = a2 + x(i+2)*y(i+2)
         + x(i+3)*y(i+3)
end do
a = a1 + a2
```

2000 cycles at best ***Vecor Unit (length 2)*** ***751 cycles at best***

Optimization Technics

Tips & Tricks: Branching (proper use of IFs)



Try to minimize as much as possible the use of IFs within the inner loops

The CPU will first assume a YES when it encounters a IF statment while filling up the instruction pipline

Optimization Technics

Tips & Tricks: Replace divisions by multiplications !

Contrary to Floating Point multiplications or additions or substractions,
divisions are very costly in terms of clock cycles

1 multiplication = 1 cycle
1 division = 14-20 cycles

```
do j=1,5000
 do i=1,5000
  a(i,j)=(b(i)-c(j))/D
 end do
end do
```

→

```
DD=1.0/D
do j=1,5000
 do i=1,5000
  a(i,j)=(b(i)-c(j))*DD
 end do
end do
```

Optimization Technics

Tips & Tricks: Precision Issue

The use of higher precision variables, larger than 64bit double precision, will use more cycles per operation

On the other end on certain architectures (IBM Power3 for example), the use of single precision will use more cycles per operation than double precision because of necessary type conversion

Optimization Technics

Tips & Tricks: Repeated multiplications for exponentials

Exponentiation with a small exponent should be done manually. Like divisions exponential operations use many cycles.

```
C = B**3.0
```

→

```
tmp1=B*B
C=tmp1*B
```

Coffee Break ...

Next
Introduction to
Parallel Computing