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

1) Overall Performance Strategy
2) Goals of Serial Optimization
3) Optimization Strategy
4) Hardware Limits and Constraints
5) Measuring Performance
6) Optimization Technics
Code Optimization

Overall Performance Strategy

Scientific Application Development Steps

- **T_{sa}** (1-12 months) → **Scientific Analysis**
  - Theory, Maths, Algorithms, Data Structures, etc.
- **T_{spec}** (1-6 months) → **Writing Specifications**
- **T_{code}** (1-12 months) → **Coding**
- **T_{opt}** (1-6 weeks) → **Serial Optimization**
- **T_{par}** (1-6 months) → **Parallelization (if necessary)**
- **T_{prep}** (1 msec - 60 min / run) → **Prepare cases**
- **T_{run}** (1 min – x months / run) → **Run cases**
- **T_{mod}** (1-12 months) → **Code Modifications**

Carol Gauthier, Centre de Calcul Scientifique (CCS), Université de Sherbrooke
Amdahl's Law

Running time of the code (Trun)

Portion of the code that cannot be parallelize/optimize

Portion of the code that can be parallelize/optimize

Even after an excellent parallelisation or optimisation...

Running time of the code (Trun)

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

Overall Performance Strategy

Overview of Development and Production Time

Development

Time

Tsa  
Tsuc  
Tcoding  
Topt  
Tpar

Production

Time

Tprep  
Trun  
Tmodif
Code Optimization

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 Parallelization
Code Optimization

Optimization Strategy

```
Initial Unoptimized Code
Setup a small realistic reference case (RefCase)
Run the RefCase - Measure overall runtime - Save results (RefResult)
Compiler Optimization (-O1 to -O5, other architecture specific options)
Includes Numerical Methods Improvements
Use Existing Optimized Libraries
Measure performance with instrumentation tools (Profiling, Hardware counters, etc..)
Identify Bottlenecks
Apply optimization technics one at a time
Check Results (with RefResults)
Optimized Code
```

Optimization loop

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

Hardware Limits and Constraints

CPU Insights

Instruction cache
Register (L1 cache)

Pipeline
Load/Store Unit
Load/Store Unit

FP Unit
FX Unit
FMA Unit
FMA Unit
Vector Unit
Specialised Unit

L2-L3 Cache

Main Memory

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

Hardware Limits and Constraints

An Analogy: The Process of Building and Delivering Tables!
Memory Access vs Clock Cycles

- **CPU**
  - L1 Cache (register)
  - 1-4 cycles

- **L2 – L3 Cache**
  - 8-20 cycles

- **RAM**
  - 80000 - 120000 cycles

- **Disk**

**TLB**
- Translation Lookaside Buffer
- List of most recently accessed memory pages
- TLB miss: 30-60 cycles

**PFT**
- Page Frame Table
- List of memory pages location

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

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

Measuring Performance

Measurement Guidelines

- Make sure you 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 theoretical 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)
```
Code Optimization

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 beginning of optimization process*
Measuring Performance

Hardware Counters

All modern processors has built-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 interesting to us!)

Software API to access counters are now available!

Using Hardware Counters is a must in Optimization!!!
Code Optimization

Measuring Performance

Hardware Counters : PAPI

Performance Application Programming Interface

A standardized 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/

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

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
Code Optimization

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.
Code Optimization

Optimization Technics

Compiler Options

**GNU C compiler**

```
gcc
  -00 -01 -02 -03 -finline-functions ...
```

**IBM AIX Fortran and C compilers**

```
xlf and xlC
  -01 -02 -03 -04 -05 -qstrict -qipa=level ...
```

**Intel Fortran and C compiler**

```
ifc and icc
  -00 -01 -02 -03 -ip -xW -tpp7(for P4) -ip ...
```
Code Optimization

Optimization Technics

Use Existing Libraries!

Existing libraries are usually highly optimized

Try several libraries and compare if possible

Recompile libraries on the platform you are running if you have the sources (trying compiler optimization)

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

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

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
Code Optimization

Y = A*X + B

In one cycle

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

Optimization Technics

Vector Units

128bit long Vector unit (P4, Opteron)

4 single precision FLOPs / cycle

2 double precision FLOPs / cycle

32bit single precision

64bit double precision

00011000000100011010000110101010
000110100000011010001101000001
0100001101001100010011010001101010
0110100101101000110100011010010

Op (+,-,x)

x1 X1 x2 x3 X2 x4

y1 Y1 y2 y3 Y2 y4

z1 Z1 z2 z3 Z2 z4

Carol Gauthier, Centre de Calcul Scientifique (CCS), Université de Sherbrooke
Array Considerations in Fortran and C/C++

**In Fortran**

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

**In C/C++**

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

**Corresponding memory representation**

<table>
<thead>
<tr>
<th>outer 1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>inner 1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

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

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!
Tips & Tricks: Minimize the number of operations!

When doing optimization, one of the first things 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
```

1250 Millions of operations

```
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
```

500 Millions of operations
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
Code Optimization

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

Tips & Tricks: Objects declarations and instanciations

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

**Optimization Techniques**

**Tips & Tricks: Function Call Overhead**

```fortran
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
```

```fortran
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 achieved with compilers inlining options. The compiler will then replace all function calls by a copy of the function code, leading sometimes to very large binary executables.

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

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

\[ C(i, j) = C(i, j) + A(i, k) \times B(k, j) \]
Code Optimization

Optimization Technics

Tips & Tricks: Blocking

```fortran
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
```

```fortran
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
```

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

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

v = 3.0*x(i) - 3.14159267
```

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

```fortran
v = 3.0*x(i) - 3.14159267
```

```fortran
do j=1,100000
  v = 3.0*x(j) - 3.14159267
end do
```
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 \(i=1,1000\)
\[a = a + x(i) \cdot y(i)\]
end do

1000 cycles at best

2 FMAs

750 cycles at best
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).

```latex
do i=1,1000
    a = a + x(i) \times y(i)
end do
```

2000 cycles at best

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

1250 cycles at best
Sum Reductions is another way of reducing or eliminating data dependencies in loops. It is more explicite than the Loop Unrolling method.

```fortran
! Original loop
do i=1,1000
   a = a + x(i)*y(i)
end do

1000 cycles at best

! Sum Reductions
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

503 cycles at best

! Vector Unit (length 2)
2000 cycles at best

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

751 cycles at best

Carol Gauthier, Centre de Calcul Scientifique (CCS), Université de Sherbrooke
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 statement while filling up the instruction pipeline
Tips & Tricks: Replace divisions by multiplications!

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

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

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

Optimized Code:
```fortran
DD=1.0/D
do j=1,5000
  do i=1,5000
    a(i,j)=(b(i)-c(j))*DD
  end do
end do
```
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.
Exponentiation with a small exponent should be done manually. Like divisions exponential operations use many cycles.

\[
C = B^{**3.0}
\]

\[
tmp1 = B^2
\]

\[
C = tmp1 * B
\]
Coffee Break ...

Next

Introduction to Parallel Computing