

Python tricks to speedup the code

With **numpy** and **scipy** package, Python is one of the best languages for numerics.

But, its slow!

Not, if combined with C++/Fortran!

The idea: Write most of the code in Python. Allocate all arrays in Python, to avoid annoying bookkeeping of allocation/deallocation of memory. Speed-up the innermost loop by fortran/C++.

Great tools to "glue" fortran code with Python: **f2py**.

Many tools to "glue" C++ with Python. The simplest to use comes with **scipy**: **weave**.

Others:

- Swig : very general. It can glue almost everything with everything. Looks complicated to use.
- python-cxx : smaller, intended only for C/C++ \leftrightarrow Python conversion. Looks quite simple, but very limited numpy support.
- weave : part of scipy, very simple. But code is a string, which is very clumsy for writing more than 10 lines of code.

1 f2py

- Step 1: Create fortran subroutine, and compile it with fortran compiler.
- Step 2: Add special f2py directives to fortran code for more user-friendly modules.
- Step 3: Create Python module by f2py:

```
f2py -c <source-name> -m <module-name> <libraries>
```

- Step 4: Include module in Python, and use it as python function.

1.1 Mandelbrot

```

SUBROUTINE Mandelb(data, ext, Nx, Ny, max_iterations)
  IMPLICIT NONE ! Don't use any implicit names of variables!
  ! Function arguments
  REAL*8, intent(out) :: data(Nx,Ny)
  REAL*8, intent(in)  :: ext(4)
  INTEGER, intent(in) :: max_iterations
  INTEGER, intent(in) :: Nx, Ny
  !f2py integer optional, intent(in)      :: max_iterations=1000
  ! !f2py integer intent(hide), depend(data) :: Nx=shape(data,0)
  ! !f2py integer intent(hide), depend(data) :: Ny=shape(data,1)
  ! Local variables
  INTEGER      :: i, j, itt
  COMPLEX*16   :: z0, z
  data(:, :) = 0.0
  DO i=1,Nx
    DO j=1,Ny
      z0 = dcmplx( ext(1) + (ext(2)-ext(1))*(i-1)/(Nx-1.), ext(3) + (ext(4)-ext(3))*(j-1)/(Ny-1.) )
      z=0
      DO itt=1,max_iterations
        IF (abs(z)>2.) THEN
          data(i,j) = 1./itt          ! result is number of iterations
          EXIT
        ENDIF
        z = z**2 + z0                ! f(z) = z**2+z0 -> z
      ENDDO
    ENDDO
  ENDDO
  RETURN
END SUBROUTINE Mandelb

```

Note !f2py directives. They can be used to specify optional parameters, sizes of arrays,...

To check if the code is free from grammatical errors, do

```
ifort -c mandel.f90
```

If successful, use f2py to get pythom module, mandel.so:

```
f2py -c mandel.f90 -m mandel
```

If your f2py does not find the right fortran compiler, you might need to add option

```
--fcompiler=intel or --fcompiler=intele
```

Open python interpreter ipython and check the module:

```
import mandel  
print mandel.__doc__
```

This should give you help on how the fortran subroutine was converted. Should be something like:

```
This module 'mandel' is auto-generated with f2py (version:2_4
```

```
Functions:
```

```
data = mandelb(ext,nx,ny,max_iterations=1000)
```

Now we have python function, which can be used to plot mandelbrot set:

```
from scipy import *
from pylab import *
import mandel # importing module created by f2py

# The range of the mandelbrot plot [x0,x1,y0,y1]
#ext=[-2,1,-1,1]
ext=[-1.8,-1.72,-0.05,0.05]

data = mandel.mandelb(ext,400,400).transpose()

# Using python's pylab, we display pixels to the screen!
imshow(data, interpolation='bilinear', cmap=cm.hot, origin='lower', extent=ext, aspect=1.)
colorbar()
show()
```

2 weave

The idea of weave is to write C/C++ code directly inside python script (much like we used to insert assembler code inside C code).

```
from pylab import *
from scipy import weave

# The C++ code is written in a multiline string.
# It will be compiled, when run for the first time. Next time,
# it will just use the "old" executable.
code="""
    using namespace std;

    return_val=0;
    for (int i=0; i<max_iterations; i++){
        if (norm(z)>4.) { return_val= 1./i; break; }
        z = z*z + z0;    // if |z|>2 the point is not part of mandelbrot set
    }
    """

Nx = 400
Ny = 400
data = zeros((Nx,Ny)) # allocate all arrays in Python!
max_iterations=1000
ext=[-1.8,-1.72,-0.05,0.05] # The range of the mandelbrot plot [x0,x1,y0,y1]

# The double loop is written in Python, which takes some time.
# Not the most optimal code.
for i in range(Nx):
    for j in range(Ny):
        z0 = ext[0] + (ext[1]-ext[0])*i/(Nx-1.) + 1j*(ext[2] + (ext[3]-ext[2])*j/(Ny-1.) )
        z = 0j
        # This line compiles and executed the code.
        # First argument - the code, second - all necessary variables, the rest - options.
        data[i,j] = weave.inline(code, ['max_iterations', 'z0', 'z'],
                                type_converters=weave.converters.blitz, compiler = 'gcc')
data = data.transpose() # for plotting, need to transpose

# Using python's pylab, we display pixels to the screen!
imshow(data, interpolation='bilinear', cmap=cm.hot, origin='lower', extent=ext, aspect=1.)
colorbar()
show()
```

This is faster than Python, but the double loop is still expensive in Python. We can further optimize by coding the double loop in C++:

```
from pylab import *
from scipy import weave

# The C++ code
code="""
using namespace std; // for using cout when debugging
for (int i=0; i<Nx; i++){
    for (int j=0; j<Ny; j++){
        complex<double> z0( ext(0)+(ext(1)-ext(0))*i/(Nx-1.), ext(2)+(ext(3)-ext(2))*j/(Ny-1.) );
        complex<double> z=0.0;
        for (int itt=0; itt<max_iterations; itt++){
            if (norm(z)>4.) { data(i,j)=1./itt; break; }
            z = z*z + z0;    // if |z|>2 the point is not part of mandelbrot set
        }
    }
}
"""

Nx = 400
Ny = 400
data = zeros((Nx,Ny), 'd')
max_iterations=1000
ext=array([-1.8,-1.72,-0.05,0.05]) # The range of the mandelbrot plot [x0,x1,y0,y1]

weave.inline(code, ['data', 'Nx', 'Ny', 'max_iterations', 'ext'],
             type_converters=weave.converters.blitz, compiler = 'gcc')

# Using python's pylab, we display pixels to the screen!
data = data.transpose()
imshow(data, interpolation='bilinear', cmap=cm.hot, origin='lower', extent=ext, aspect=1.)
colorbar()
show()
```

3 Second Homework

- Write a python script to compute spherical bessel functions with up and down recursion. Plot the error of your algorithm when compared to scipy version of $j_l(x)$.
- Use f2py to speed up the algorithm: write a fortran subroutine and call it from Python using f2py.
- Use weave to speed up the algorithm: write C++ inline code to speed up the evaluation of $j_l(x)$.

Hardware

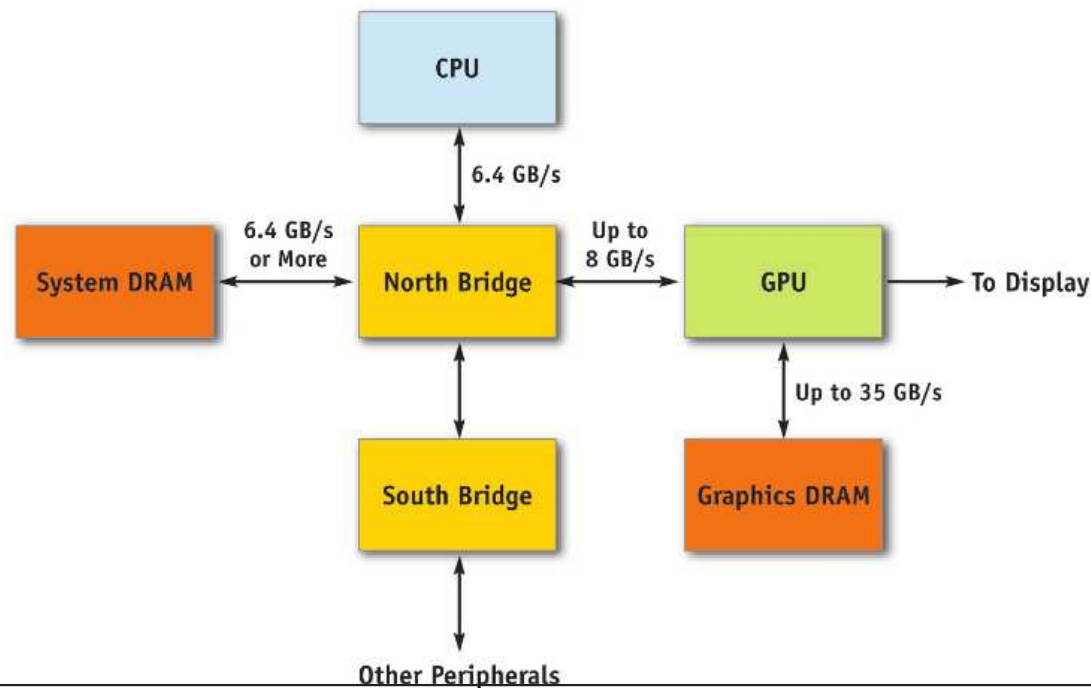
- The central unit of a computer is **CPU** (Central processing unit). It consists of a few very high speed memory units called registers. Nowadays CPU has many cores - even 16.
- Closest to the CPU is a small part of memory, called **Cache**. Nowadays, typical size of the Cache is 1MB. The Cache is very important for performance (difference between Pentium and Celeron).
- **RAM** is the central memory unit which can be accessed randomly. It is fast, but orders of magnitude slower than cache, which is orders of magnitude slower than registers.
- **Hard disc** is the slowest memory unit where the data can be (more or less) permanently stored.

ADVICE: *Use the fastest memory you can*

4 GPU

Recently, a new type of high performance computing is emerging - computing by Graphical Processing Unit (GPU). Nvidia developed the TESLA unit, and ATI and AMD developed FireStream.

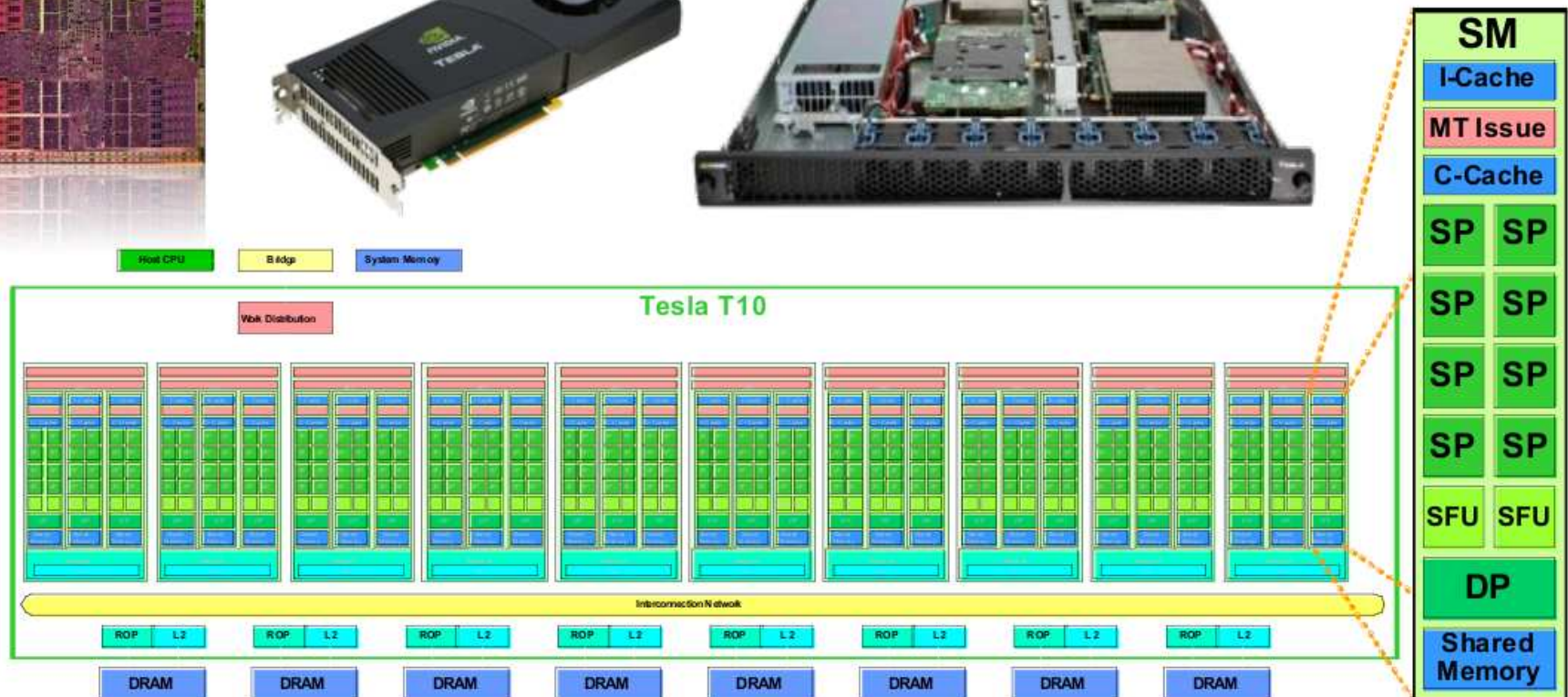
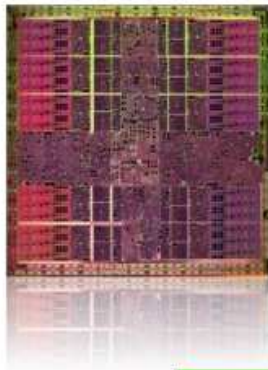
The GPU's have outperformed CPU's in terms of speed. The speed of each unit (CPU/GPU) is not increasing so much with time. But the number of processing units is increasing. GPU contains > 200 processors.





CUDA Computing with Tesla T10

- 240 SP processors at 1.45 GHz: 1 TFLOPS peak
- 30 DP processors at 1.45GHz: 86 GFLOPS peak
- 128 threads per processor: 30,720 threads total



- Do not use hard-disc for data manipulation. Keep data in RAM. If you need a lot of RAM, estimate whether it fits into RAM. Rethink your algorithm before you start writing data to hard-disc.
- Try avoiding random access of data in RAM to reduce cache misses.
- The data which you need in the innermost loop should be stored in a way that the access is maximally continuous.

Typical example is a matrix manipulation.

In C or C++, one needs to access multidimensional arrays in the following order

```
for (int i=0; i<size; i++)  
  for (int j=0; j<size; j++)  
    A[i][j] = .....
```

since the data is stored in a row major order. In Fortran, the same loop should be written in the following way

```
do i=1, size  
  do j=1, size  
    A(j,i) = .....
```

```
  enddo  
enddo
```

Notice the transposed form of $A(j,i)$ versus $A[i][j]$.

This is because Fortran uses column major storage. The figure explains it all.

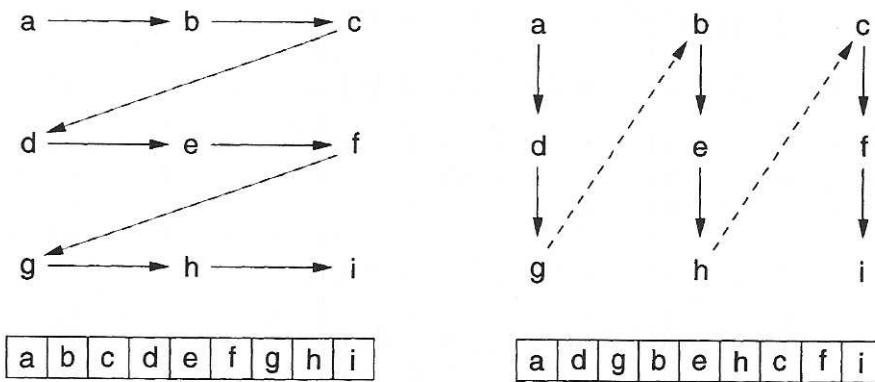


Fig. 15.2 (Left) Row-major order used for matrix storage in C and Pascal; (right) column-major order used for matrix storage in Fortran. On the bottom is shown how successive matrix elements are stored in a linear fashion in memory.

Remember: *The innermost index j runs:*

in C++ $A[...][j]$

in Fortran $A(j,....)$

Scientific programs are usually very tuned for performance. This usually goes in expense of portability and readability.

ADVICE: *Newer optimize those parts of the program which are not very crucial for speed. In typical applications, only 20% of the code spends 80% of the time. Optimize only those 20% of the code. Make the rest of the code more readable.*

Remember: *80/20 rule*

Available numeric software

Numerous libraries are available on the Web. Sometimes it is hard to decide which one to use.

The current situation: Most of the best numeric algorithms are still written in Fortran → need to learn how to use them.

One of the most comprehensive archives is **GAMS** at <http://gams.nist.gov/>:



GAMS info:

Public access repository NETLIB is located at Oak Ridge National Laboratory, Knoxville, TN and AT&T Bell Laboratories, Murray Hill, NJ

It includes 111 packages and 8792 problem-solving modules (routines)

Search for software according to

- ◆ what problem it solves.
- ◆ package name.
- ◆ module name.
- ◆ text in module abstracts.

Go straight to the problem decision tree.

- ◆ A Arithmetic, error analysis
- ◆ E Number theory
- ◆ C Elementary and special functions (search also class L5)
- ◆ D Linear Algebra
- ◆ E Interpolation
- ◆ F Solution of nonlinear equations
- ◆ G Optimization (search also classes K, L8)
- ◆ H Differentiation, integration
- ◆ I Differential and integral equations
- ◆ J Integral transforms
- ◆ K Approximation (search also class L8)
- ◆ L Statistics, probability
- ◆ M Simulation, stochastic modeling (search also classes L6 and L10)
- ◆ N Data handling (search also class L2)
- ◆ Q Symbolic computation
- ◆ P Computational geometry (search also classes G and Q)
- ◆ Q Graphics (search also class L3)
- ◆ R Service routines
- ◆ S Software development tools
- ◆ Z Other

Some of the libraries are unfortunately commercial (line NAG). Check whether they are

installed on the system (at the moment we do not have NAG license). Do *not install libraries* if they are already installed (like blas and lapack) because they are probably much more optimized than your compiled code can be.

5 Mixing Fortran and C++

Let's try to use fortran code inside C++ code.

Suppose we need to calculate erfc function of complex argument. GAMS finds many available routines. Let's pick the one from TOMS package. The header looks like this

```
C      ALGORITHM 680, COLLECTED ALGORITHMS FROM ACM.
C      THIS WORK PUBLISHED IN TRANSACTIONS ON MATHEMATICAL SOFTWARE,
C      VOL. 16, NO. 1, PP. 47.
C      SUBROUTINE WOFZ (XI, YI, U, V, FLAG)
C
C      GIVEN A COMPLEX NUMBER Z = (XI,YI), THIS SUBROUTINE COMPUTES
C      THE VALUE OF THE FADDEEVA-FUNCTION  $W(Z) = \exp(-Z^2) \operatorname{erfc}(-iZ)$ ,
C      WHERE ERFC IS THE COMPLEX COMPLEMENTARY ERROR-FUNCTION AND I
C      MEANS  $\sqrt{-1}$ .
C      THE ACCURACY OF THE ALGORITHM FOR Z IN THE 1ST AND 2ND QUADRANT
C      IS 14 SIGNIFICANT DIGITS; IN THE 3RD AND 4TH IT IS 13 SIGNIFICANT
C      DIGITS OUTSIDE A CIRCULAR REGION WITH RADIUS 0.126 AROUND A ZERO
C      OF THE FUNCTION.
C      ALL REAL VARIABLES IN THE PROGRAM ARE DOUBLE PRECISION.
C
C
C      THE CODE CONTAINS A FEW COMPILER-DEPENDENT PARAMETERS :
C      RMAXREAL = THE MAXIMUM VALUE OF RMAXREAL EQUALS THE ROOT OF
C      RMAX = THE LARGEST NUMBER WHICH CAN STILL BE
C      IMPLEMENTED ON THE COMPUTER IN DOUBLE PRECISION
C      FLOATING-POINT ARITHMETIC
C      RMAXEXP = LN(RMAX) - LN(2)
C      RMAXGONI = THE LARGEST POSSIBLE ARGUMENT OF A DOUBLE PRECISION
C      GONIOMETRIC FUNCTION (DCOS, DSIN, ...)
C      THE REASON WHY THESE PARAMETERS ARE NEEDED AS THEY ARE DEFINED WILL
C      BE EXPLAINED IN THE CODE BY MEANS OF COMMENTS
C
C
C      PARAMETER LIST
C      XI      = REAL      PART OF Z
C      YI      = IMAGINARY PART OF Z
C      U       = REAL      PART OF W(Z)
C      V       = IMAGINARY PART OF W(Z)
C      FLAG    = AN ERROR FLAG INDICATING WHETHER OVERFLOW WILL
```

First, we need to compile the Fortran code to produce object file:

```
ifort -c erfc.f
```

If it compiles, we can try to call it from C++. We need to write a prototype for Fortran routine. It is essential, to convert Fortran types to C types correctly. Here are some conversion rules

(Source: <http://www.astro.indiana.edu/~jthorn/c2f.html>)

- use pointers to arguments (in Fortran everything is pointer)
- Name Mangling
 - names of routines should be lowercase
 - in many platforms necessary to add underscore to function names
- Fortran subroutines is equivalent to C function returning void
- Conversion of types:
 - INTEGER → int
 - LOGICAL → int
 - REAL → float
 - DOUBLE PRECISION==REAL*8 → double

- Fortran subroutine or function → pointer to a function (Name Mangling applies)
- CHARACTER → usually char* (but be careful)
- Fortran starting index 1 goes to C/C++ starting index 0
- Due to column/row major convention in Fortran/C, the matrixes look transposed

Using the above rules on the fortran code

```
SUBROUTINE WOFZ (XI, YI, U, V, FLAG)
  IMPLICIT DOUBLE PRECISION (A-H, O-Z)
  LOGICAL FLAG
```

we prepare corresponding C++ prototype

```
extern "C"{
  void wofz_(double* xi, double* yi, double* ui, double* vi, int* flag);
}
```

A wrapper C++ function, hides the details of the call

```
std::complex<double> erfc(const std::complex<double>& z)
// C++ wrapper function for call to fortran routine to calculate erfc of a complex number
{
  int info;
  double rz = z.real(), iz = z.imag();
  double ru, iu;
  wofz_(&rz, &iz, &ru, &iu, &info);
  if (info!=0) std::cerr <<"Can't compute error function at "<<z<<" got "<<ru<<","<<iu<<" info="<<info<<std::endl;
  return std::complex<double>(ru,iu); // return value optimization!
}
```

Finally, we can compile C++ code and link together

```
g++ -o erfc cerfc.cc erfc.o
```

which is equivalent to

```
g++ -c cerfc.cc
```

```
g++ -o erfc cerfc.o erfc.o
```

See the available ifortran/erfc program for details.

LAPACK & BLAS

If you need to solve a standard linear algebra problem don't even think of writing your own routine and don't take it from your friend's program. It will be orders of magnitude slower. There are very optimized and FREE libraries called blas and lapack.

Blas stands for "Basic Linear Algebra". The basic information is available at <http://www.netlib.org/lapack/lug/node145.html> and also many other locations. Those two libraries are already installed on most of computers (including our cluster). If they are not, consider the following few distributions

- ATLAS: Automatically Tuned Linear Algebra Software <http://math-atlas.sourceforge.net/>

- GotoBLAS: <http://www.tacc.utexas.edu/resources/software/>

There are "Level 1", "Level 2" and "Level 3" BLAS routines.

- Level 1 implements vector-vector operations
- Level 2 implements matrix-vector operations
- Level 3 implements matrix-matrix operations

The computation time increases as N for "Level 1", as N^2 for "Level 2" and as N^3 for "Level 3". It is crucial to use "Level 3" routines because they are truly faster than an naive implementation can be. On the other hand, "Level 1" routines do not give considerable speed improvement and are therefore not mandatory to use. Typical routine from "Level 3" BLAS is `dgemm`, which implements matrix multiplication for real matrix (`zgemm` for complex matrix).

LAPACK is built on top of blas and implements more sophisticated linear algebra operations including solving

- system of linear equations
- linear least square problem

- eigenvalue problem
- singular value problem

For more information, check out <http://www.netlib.org/lapack/lapackqref.ps> and for more detailed explanation take a look at

<http://www.cs.colorado.edu/~jessup/lapack/documentation.html>

When using LAPACK or BLAS routine for the first time, it is crucial to check the answer against Mathematica or using a simple example where you know the answer. There are 100 places where something might go wrong when calling BLAS or LAPACK routine from C or C++.

A good idea is to implement wrapper routines (or member routines of your class) which call LAPACK.

Example:

In sections "Hartree-Fock" and "Density Functional Theory", we will need a routine to solve a generalized eigenvalue problem for a real symmetric matrix $\mathbf{A}x = E\mathbf{O}x$

Typing

```
man dsygvd
```

gives the following documentation

NAME

DSYGVD - compute all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form $A*x=(\lambda)*B*x$, $A*Bx=(\lambda)*x$, or $B*A*x=(\lambda)*x$

SYNOPSIS

```
SUBROUTINE DSYGVD( ITYPE, JOBZ, UPLO, N, A, LDA, B, LDB, W, WORK, LWORK, IWORK, LIWORK, INFO )
    CHARACTER      JOBZ, UPLO
    INTEGER         INFO, ITYPE, LDA, LDB, LIWORK, LWORK, N
    INTEGER         IWORK( * )
    DOUBLE          PRECISION A( LDA, * ), B( LDB, * ), W( * ), WORK( * )
```

A prototype for subroutine DSYGVD can be expressed by

```
extern "C"
void dsygvd_(const int* ITYPE, const char* JOBZ, const char* UPLO, const int* N,
            double* A, const int* LDA, double* B, const int* LDB, double* W, double* WORK, const int* LWORK,
            int* IWORK, const int* LIWORK, int* INFO);
```

and a typical call to the subroutine from a C program is

```
dsygvd_(&itype, "V", "U", &N, A, &lدا, B, &lدا, eigenval, work, &lwork, iwork, &liwork, &info);
```

Complete implementation can be found in the section on the Hartree-Fock method.

Writing makefiles

It is a good practice to write a makefile for every project. Makefile typically contains information about the default compilers, location of necessary include files and necessary libraries to link to the executable.

There are many nice tutorials available on the Web including

http://www.apl.jhu.edu/Misc/Unix-info/make/make_3.html **or**

<http://users.actcom.co.il/~choo/lupg/tutorials/writing-makefiles/writing-makefiles.html> **or**

<http://www.gnu.org/software/make/manual/>

We will briefly describe the steps in writing simple makefiles.

- The name of the makefile can be "Makefile" or "makefile" and is typically located in the same directory as other source files.
- User types "make" in the source directory and makefile is executed producing the executable file.

Lets call our project He0. The C++ source file is He0.cc. The simplest makefile contains the following two lines

```
He0 : He0.cc
    g++ -o He0 He0.cc
```

Note: Each line in the commands list must begin with a TAB character!

- The dependency rule defines under what conditions a given file needs to be recompiled, and how to compile it.

The above rule states that the executable "He0" has to be recompiled whenever "He0.cc" is modified. The rule tells us that "He0" can be obtained by the command "g++ -o He0 He0.cc".

Suppose the C++ source file contains a call to the above defined lapack routine to solve the eigenvalue problem. In this case we get an error message

```
undefined reference to `dsygvd_`
```

since we did not link lapack and blas libraries. Once we change makefile contents to

```
He0 : He0.cc
    g++ -o He0 He0.cc -llapack -lblas
```

we get an error message

```
undefined reference to `e_wsfe`
```

and many other error message for undefined references.

We need to link also g2c library which contain basic Fortran routines.

There is a very useful command to check which library is missing. Command "nm" invoked on a library, lists all undefined and defined symbols in the library. For example, the command

```
nm /usr/lib/libg2c.a|grep e_wsfe
```

gives

```
00000000 T e_wsfe
```

which means that "e_wsfe" is defined in libg2c while the line

```
U sprintf
```

tells us that "sprintf" is not defined in this library since it is in the system library libc.a .

Things that can be improved in Makefile

- We can define any variable A and use it later as \$(A).
- We can write a default rule for any Fortran and any C++ program. The rule must starts with: ".SUFFIXES: .f" and ".f.o" which tells the system how to get .o file from .f file.
- We can add rule "clean", which will clean up all object files and executable

- We can add many more rules for multiple executables or object files. By default, "make" with no arguments will execute the first rule.

Here is an improved version of the above makefile

```
C++ = g++
LIBS = -llapack -lblas -lg2c
objects = He0.o
exe = He0

$(exe) : $(objects)
    g++ $(CFLAGS) -o $@ $? $(LIBS)

clean :
    rm -f $(objects) $(exe)

.SUFFIXES : .cc
.cc.o:
    $(C++) $(CFLAGS) -c $<

.SUFFIXES : .f
.f.o:
    $(F77) $(FFLAGS) -c $<
```

Tools for detecting errors in the code

Debugger

Usually we have many debuggers available on a computer. Unfortunately not every debugger is compatible with every compiler.

- For gnu compiler `gcc` or `g77` the best choice is `gdb` (also part of gnu project and therefore fully compatible).
- For intel compiler `ifort` the appropriate debugger is `idb`
- On other non-linux system you will most probably find different kind of native compiler and corresponding debugger which is compatible with it.

To debug your program, you need to recompile the code with `-g` option added (For example: `"g++ -g -c mycode.cc"`).

It is convenient to use debugger in a tex editor like emacs. Emacs can displays the source in a separate buffer and displays a pointer to shows you which line of source code will be executed next

- To start gdb within emacs, say M-x gdb <Enter>, edit the gdb execution line and say <Enter> again.
- Set a breakpoint (for example "b main" meaning break in program main), and type "r" and <Enter> to run the program. If your program needs input arguments, add them to "r" command ("r myparam=something").
- When the program stops, a second emacs buffer window will be created containing the source code with a pointer "=>" indicating the next line of code to be executed. This is your first breakpoint.
- To learn more about available commands, you can type "help" in debugger buffer.
- Most often used commands are
 - "n[ext]" (Single-step without descending into functions)
 - "s[tep]" (Single-step descending into functions)
 - "p[rint] <variable>" (Prints the content of the variable)

- "b[reak]" <line number> (Set a breakpoint at line number)
- "b[reak]" <function name> (Set a breakpoint at function)
- "b[reak]" <class::name> (Set a breakpoint at class member function)
- "b[reak]" <class::tab> (Lists members in class)
- "i[nfo] b[reak]" (Lists breakpoints currently set)
- "d[elete] 1" (Delete breakpoint number 1)
- "dis[able] 1" (Disable breakpoint number 1)
- "en[able] 1" (Enable breakpoint number 1)
- "c[ontinue]" (Continues to next breakpoint or end)
- "fin[ish]" (Finish current function, loop, etc.)
- "x/10f <pointer to memory>" (interprets data following pointer address as 10 floating point numbers and prints them)
- "q[uit]" (Leaving debugger)

For more information, google gdb! There are hundreds of nice tutorials. (for example:

http://www-ccrma.stanford.edu/jos/pasp/Executing_gdb_Emacs.html)

If you want to use Intel debugger "idb" inside emacs, add the following lines to your ".emacs"

file

```
(load-file "/opt/intel_idb_73/bin/idb.el")
```

You need to replace the path to file "idb.el" with your path. Then say M-x idb <Enter> and continue debugging.

Profiler

Debuggers can help you locate your mistakes. But if your program is working properly but is very slow, you need to speed it up!

Profilers show you how much time is spend in each subprogram or subroutine and (if necessary) even in each line of your code.

If you still remember **80/20** rule you can understand now why profilers are so important.

A good strategy is to write a scientific program which is very readable (highly intuitive and understandable) and not necessary fast or optimized for speed. It usualy takes you much less time to write simple non-optimized code. Then it comes profiling and optimization of those few lines of code that spend most of time. You are going to be surprised many times that the actual time spent in many parts of the code, you believed are crucial for speed, is completely negligable (less than 1% of all time).

"gprof" is very simple profiler (part of gnu package fully compatible with gdb) and most of the time sufficient for our needs.

- Recompile your code with **-p** option added. If you wish more precise information, add also **-g** option. The reason is that optimization usually inlines many function calls and then they are not "seen" by gprof as separate units. The problem, however, is that program compiled with "-O3" or "-g" in general does not spend proportional time in each part of the program (some parts can be more optimized than others). Consequently, "-g" will not always show precise time spent in each subprogram in case of "-O3" execution. A good strategy is to first profile with -g and then check your results with -O3 option as well.
- **execute your program as usual.** With profiling turned on, the execution is going to be considerable slower. If your program is already very slow, consider profiling a simplified version.
- type "gprof <name of your executable>" and you will get profiled information listed on standard output.
- Go into your code and optimize subprogram which spends most of the time. Then compile for profiling again and check weather you won!
- Continue until most of the time is spent in some very optimized library like blas or you have no idea anymore how to speed up the code.

Memory leak detector

You probably need to use memory leak detector if

- your program coredumps and debugging the program does not help
- it corredumps from no obvious reason
- every time you debug corredumps at different place
- when debugging, certain variable is being changed when you really do not expect it to be changed.

The above cases usually occur because you are reading from or writing to a memory location outside of the scope of your program or outside the memory location reserved for your variable.

In addition, memory leak detectors will also detect other missusings of dynamic memory allocation and deallocation. For example: if you forget to clean up some memory reserved by new (Each "new" should be paired by a corresponding "delete" statement).

One of free available detectors is called "valgrind" (<http://valgrind.org/>).

You will most probably need to install valgrind on your computer on your own. For many common linux distributions, rpm packages are already available. Check them out first.

There are many other Memory Leak Detectors available. Some are for free but most of them are commercial. The website <http://www.linuxjournal.com/article/6556> lists and describes many of them.

More Advanced and very Useful Trick

Example is from minimization of a function but can be equally well used in numerous other applications.

Suppose a function $F(x_0, x_1, y_0, y_1, \dots)$ needs to be minimized with respect to variables x_0 and x_1 while variables y_i are kept fixed during minimization.

A typical Fortran program would define global variables y_i which are changed before the minimization routine is called. User-defined function F uses those global variables at each call.

This leads to an extremely unreadable code and always needs to be avoided. **DO NOT USE GLOBAL VARIABLES!** In large scale projects, it is practically impossible to keep track of all the statements which could change a global variable and therefore user can not easily figure out when a global variable is changed.

In C++, we have a workaround to this problem. All necessary variables y_i can be hidden in a user-defined class type and remain completely local. The code thus becomes intuitively very clear for an end user. The code that does the trick, is however slightly complicated and needs some thinking.

Let us take a very simple example of a function

$F(x_0, x_1, y_0, y_1) = (x_0 - y_0)^2 + (x_1 - y_1)^2$. The C++ statement that minimizes function F is very clear and simple

```
Parabola parab(y0,y1);  
F = Minimize(2, X, Parabola::FCN);
```

Notice that class `parab` (which is an instance of class type `Parabola`) is a local variable and will not be used after minimization (will most probably be destroyed).

A naive implementation would be

```
class Parabola{  
    float u0, u1;  
public:  
    Parabola(float u0_, float u1_) : u0(u0_), u1(u1_){};  
    double Value(double x0, double x1)  
        { return sqr(x0-u0)+sqr(x1-u1);}  
};
```

and member function `Value(x0,x1)` would be given to the Fortran subroutine. This will not work because `Value` is not a static function (not a unique function but different for every instance of a `Parabola` class).

We need a static function within the class which can be given to Fortran minimization subroutine. However, static function can not use non-static variables and therefore can not access members of a class. The reason is that static function is a common (unique)

function for all instances of a class type while class members are different for different instances. Therefore we need a static copy of the data. The simplest way to do that is to define a static pointer which will point to the instance which was last created.

The class type Parabola needs to be redefined to

```
class Parabola{
    float u0, u1;
    static Parabola* pt;
public:
    Parabola(float u0_, float u1_) : u0(u0_), u1(u1_)
    double Value(double x0, double x1)
        { return sqr(x0-u0)+sqr(x1-u1);}
    static void FCN(const int* N, const float* X, float* F);
};
```

Notice the static pointer and static function FCN. The constructor needs to point the static pointer to itself. The static member FCN can that access all data of a class through the static pointer.

```
Parabola::Parabola(float u0_, float u1_) : u0(u0_), u1(u1_)
{Parabola::pt = this;}

void Parabola::FCN(const int* N, const float* X, float* F)
{
    if (*N!=2) std::cerr<<"Not right number of variables!"<<std::endl;
    *F = pt->Value(X[0],X[1]);
    std::cout<<*F<<" "<<X[0]<<" "<<X[1]<<std::endl;
}
```

```
}
```

Finally, the function Minimize is a wrapper function to the fortran subroutine and takes the form

```
template <class functor>
float Minimize(int N, float x[], functor& Fun)
{
    float* x0 = new float[N];
    int lw = N*(N+10);
    float* w = new float[lw];
    for (int i=0; i<N; i++) x0[i]=x[i];
    float f; int info;
    uncmn_(&N,x0,Fun,x,&f,&info,w,&lw);
    delete[] w;
    delete[] x0;
    return f;
}
```

where the prototype for Fortran subroutine is

```
extern "C"  
void uncmn_(const int* N, float* X0, void (*FCN)(const int* N, const float* X, float*  
float* X, float* F, int* INFO, float* W, int* LW);
```

Fortran subroutine UNCMIN was downloaded from GAMS web page at

<http://gams.nist.gov/serve.cgi/ModuleComponent/5672/Fullsource/ITL/uncmin.f>

For details, check the available program [static_function/minimize](#) .