CODE OPTIMIZATION

HOW FAST IS IT

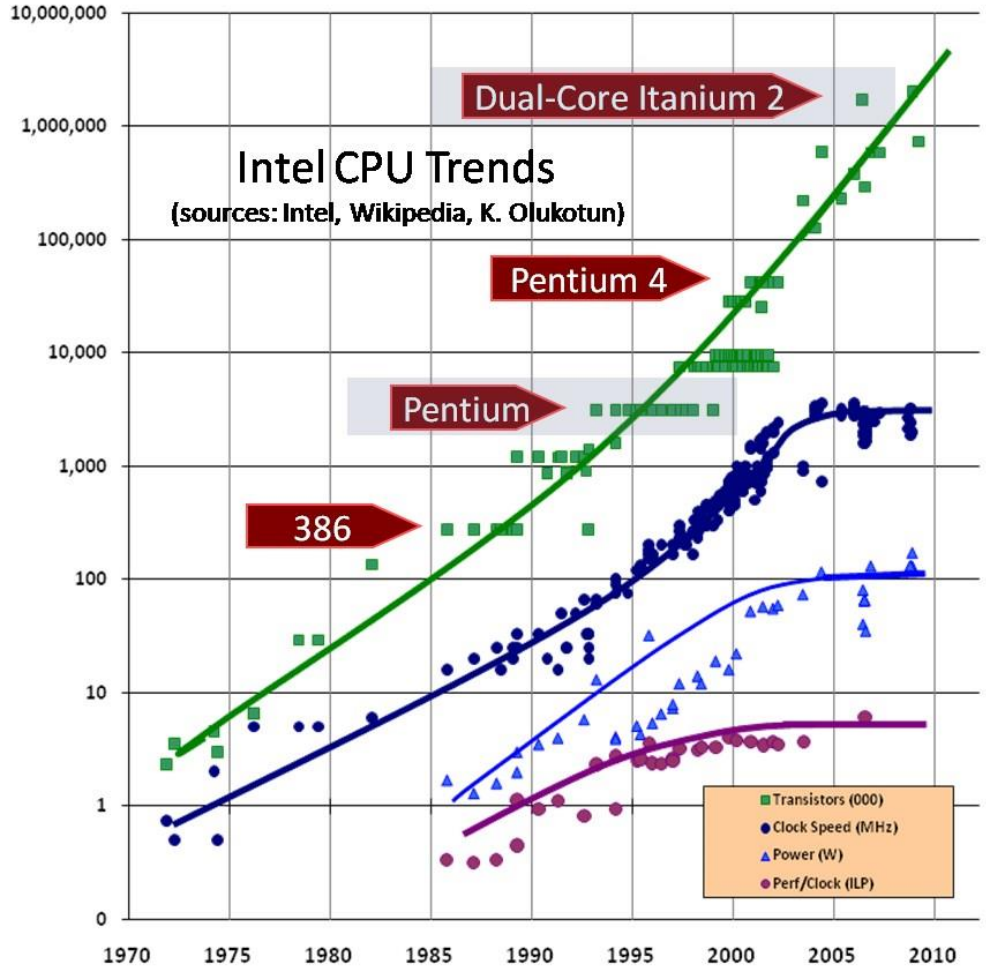
THE NEED FOR SPEED

- In High-Performance Computing we try to make code run as fast as possible (given the constraints we always experience)
- "Make it right, then make it fast." Be sure your code is correct before you begin to optimize.
- The first step of parallelization is to make our serial code run efficiently. This is especially important for threading parallelization.

MOORE'S LAW AND DENNARD SCALING

- Moore's Law: transistor density doubles every 18-24 months.
- Dennard Scaling: as transistors shrink their power density remains constant=>voltage and current go down as a function of length scale. When Dennard Scaling held the performance per watt grew exponentially.
- We have reached the end of Dennard Scaling (this happened circa 2005) but not (yet) of Moore's Law. To compensate for the loss of Dennard Scaling, chip manufacturers introduced multiple cores per cpu.

INCREASE OF CLOCK SPEED WITH TIME



PERFORMANCE AND TUNING

CORRECTNESS VERSUS PERFORMANCE

- “Make it right, then make it fast”
- It does no good to optimize or parallelize an incorrect program
- It does no good if the optimizations or parallelization break correctness
- Correctness is not necessarily the same as giving the same answer
  - Random (stochastic) applications
  - Problems with numerical precision

MEASUREMENT

- In addition to execution times – keep metadata
  - Version of the application
  - Machine configuration
    - Architecture, speed, memory, cache, motherboard
    - Operating system
  - Which compiler, compiler flags, etc.
  - Other factors, load, other users, time of day
  - Inputs

MEASUREMENT TOOLS

- Time
  - `time` is the simplest Unix tool.

```
time ./mycode .01 output
  real    0m14.435s
  user    0m14.424s
  sys     0m0.011s
```

  - Using four cores

```
  real    0m6.400s
  user    0m21.195s
  sys     0m0.061s
```

INTERPRETING TIME

- Real time is the total walltime
- User time is the total cpu time (over all cores used)
- System time is time spent in system calls.

PROFILING—COMPILED LANGUAGES

- The profiler most widely used is gprof
- Compiler and linker flags must be added.

  gcc –pg myprog.c –o myprog

  gfortran –pg myprog.f90 –o myprog
- After the binary is prepared run it as usual

  myprog inputs
- This will generate a file gmon.out. Run gprof

  gprof myprog > profile.out
- This generates a call graph and timing for each subprogram.

VALGRIND

- Another useful tool is valgrind. Valgrind is a free debugging/profiling tool particularly useful for finding memory leaks (which can slow a code tremendously) and cache inefficiencies.
- Valgrind has several tools including memcheck (for memory checking) and cachegrind (for cache efficiency verification).
- Valgrind.org is the site for documentation.
- Example: run to check memory
  - Compile with `-g` as for debuggers
  - Do not optimize beyond `O1`
  - Run your binary
  - Run valgrind

```
valgrind --leak-check=yes ./myprog
```

PROFILING–PYTHON

- There are two versions in Python 2.7
  - cProfile
  - profile (in Python 3 profile uses cProfile by default).
- They work similarly to gprof.
- We will use cProfile as our example.

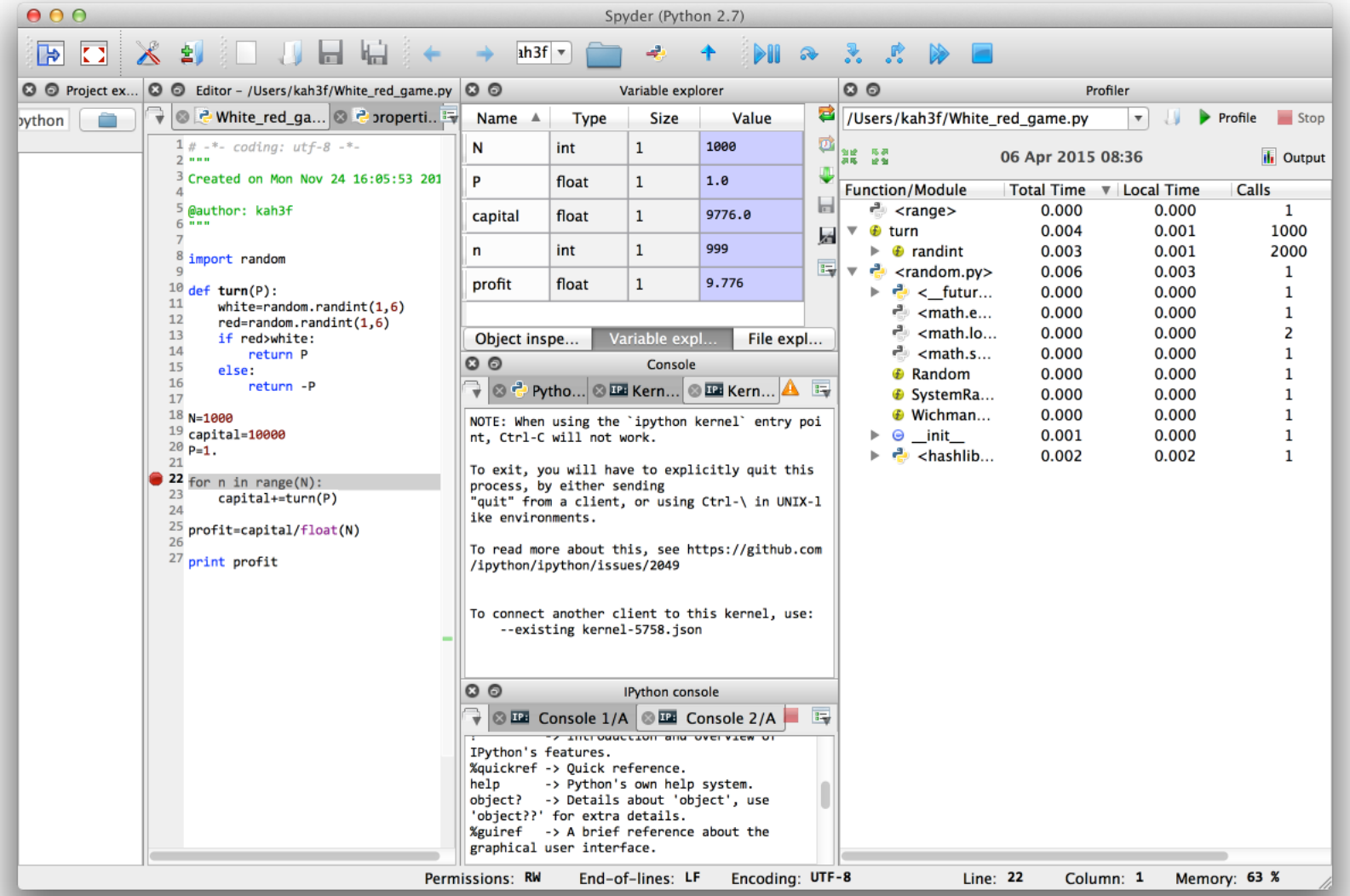
```
import pstats, cProfile

import myModule

cProfile.runctx("MyModule.myFunc()", globals(),\
     locals(), "Profile.prof")

s = pstats.Stats("Profile.prof")
s.strip_dirs().sort_stats("time").print_stats()
```

PROFILING WITH SPYDER

- Spyder offers the ability to run the profiler from within the IDE.
- Run->Profile
- Results will be shown in the Profiler pane
  - View->Panes->Profiler
- The profiler can also be run from the command line using the –m (module) flag:

```
python -m cProfile -s cumtime myscript.py
```

- Run with cProfile and sort the result by time.


Spyder (Python 2.7)
Project ex...
Editor - /Users/kah3f/White_red_game.py
White_red_ga...
-*- coding: utf-8 -*-
"""
Created on Mon Nov 24 16:05:53 201
@author: kah3f
"""
import random
def turn(P):
white=random.randint(1,6)
red=random.randint(1,6)
if red>white:
return P
else:
return -P
N=1000
capital=10000
P=1.
for n in range(N):
capital+=turn(P)
profit=capital/float(N)
print profit
Variable explorer
Name Type Size Value
N int 1 1000
P float 1 1.0
capital float 1 9776.0
n int 1 999
profit float 1 9.776
Object inspe...
Variable expl...
File expl...
Console
Pytho...
Kern...
Kern...
NOTE: When using the `ipython kernel` entry poi
nt, Ctrl-C will not work.
To exit, you will have to explicitly quit this
process, by either sending
"quit" from a client, or using Ctrl-\ in UNIX-l
ike environments.
To read more about this, see https://github.com
/ipython/ipython/issues/2049
To connect another client to this kernel, use:
--existing kernel-5758.json
IPython console
Console 1/A
Console 2/A
IPython's features.
%quickref -> Quick reference.
help -> Python's own help system.
object? -> Details about 'object', use
'object??' for extra details.
%guiref -> A brief reference about the
graphical user interface.
Profiler
/Users/kah3f/White_red_game.py
Profile
Stop
06 Apr 2015 08:36
Output
Function/Module Total Time Local Time Calls
<range> 0.000 0.000 1
turn 0.004 0.001 1000
randint 0.003 0.001 2000
<random.py> 0.006 0.003 1
<__futur... 0.000 0.000 1
<math.e... 0.000 0.000 1
<math.lo... 0.000 0.000 2
<math.s... 0.000 0.000 1
Random 0.000 0.000 1
SystemRa... 0.000 0.000 1
Wichman... 0.000 0.000 1
__init__ 0.001 0.000 1
<hashlib... 0.002 0.002 1
Permissions: RW
End-of-lines: LF
Encoding: UTF-8
Line: 22
Column: 1
Memory: 63 %

MORE DETAILED TIMING

- Profilers are statistical in nature. They query the program to find out what it's doing at a snapshot of the run, then compile the results into a kind of frequency chart.
- Often we want to know exactly how long a particular code, or part of the code, ran.
- Languages provide different ways to obtain cpu time within a running program.

C++

```
#include <ctime>
using namespace std;
clock_t start_t=clock();
 {do stuff}
float etime=(float)(clock()-
     start_t)/(float)CLOCKS_PER_SEC;
cout<<"Elapsed time "<<etime<<endl;
```

FORTRAN

- Intrinsic subroutine `cpu_time`

```
real    ::   start_t, end_t
call cpu_time(start_t)
do stuff
call cpu_time(end_t)
write(*,*) "Elapsed time is ",end_t - start_t
```

For parallel code use a library-provided intrinsic or `system_clock`

```
integer :: start_t,end_t, rate
real      :: elapsed_time
call system_clock(count_rate=rate)
call system_clock(start_t)
…
call system_clock(end_t)
elapsed_time=(end_t - start_t)/real(rate)
```

PYTHON

- In Python 2.7 use the time package

```
import time
start_time=time.clock()
do stuff
end_time=time.clock()
print "Elapsed time is ",end_time-start_time
```

- In Python 3.3+ use timeit
  - Must instantiate a Timer object
  - On some systems may provide only wallclock time,not cpu time.

REMEMBER: WALLCLOCK TIME IS EVERYTHING!

- The only metric that ultimately matters is Wallclock time to solution.
  - Wallclock is how long it takes to run your simulation.
  - Wallclock is how long your code is blocking other users from using the machine.

- **Lies, Damn Lies and Statistics…**
  - TFlops/PFlops numbers are ‘worse’ than statistics.
  - You can easily write a code that gets very high Flops numbers but has a longer time to solution.

OPTIMIZATION STRATEGIES

WHEN TO OPTIMIZE?

Premature optimization is the root of all evil.

-Donald Knuth

- Code optimization is an iterative process requiring time, energy and thought.
- Performance tuning is recommended for:
  1) production codes that are heavily used in your research community
  2) projects with limited allocation (to maximize available compute hours).
  3) faster turnaround for computations enabling more research results.

OPTIMIZATION STRATEGY: COMPILED LANGUAGES

- Aim: To try to minimize the amount of work the compiler is required to do.

  DO NOT rely on the compiler to optimize bad code.

Use a two pronged approach:

1. Write easily optimizable code to begin with.
2. Once the code is debugged and working try more aggressive optimizations.

WORK WITH YOUR COMPILER

- The first step is to use the optimization capabilities of your compiler.
- Normally there are several levels available. See your compiler's manual. The default is specified with -O
- The –g flag used for debugging usually inhibits optimization (or is incompatible with it) depending on the compiler.
  - Fortran: array bounds checking is essential for debugging but will slow the code down *considerably*. See your compiler manual for the flag (-CB for Intel)
- IDEs
  - If you are using an Integrated Development Environment such as Code::Blocks or Eclipse, typically the default build is Debug. Always build a Release version once your code is working.

WHAT COMPILERS WILL *NOT* DO.

- They will not reorder large memory arrays.
- They will not factor/simplify large equations.
- They will not replace inefficient logic.
- They will not vectorize large complex loops.
- They will not fix inefficient memory access.
- They will not detect and remove excess work (unless they think there is *no* work in a segment).
- They may not automatically invert divisions.
- …

CACHE

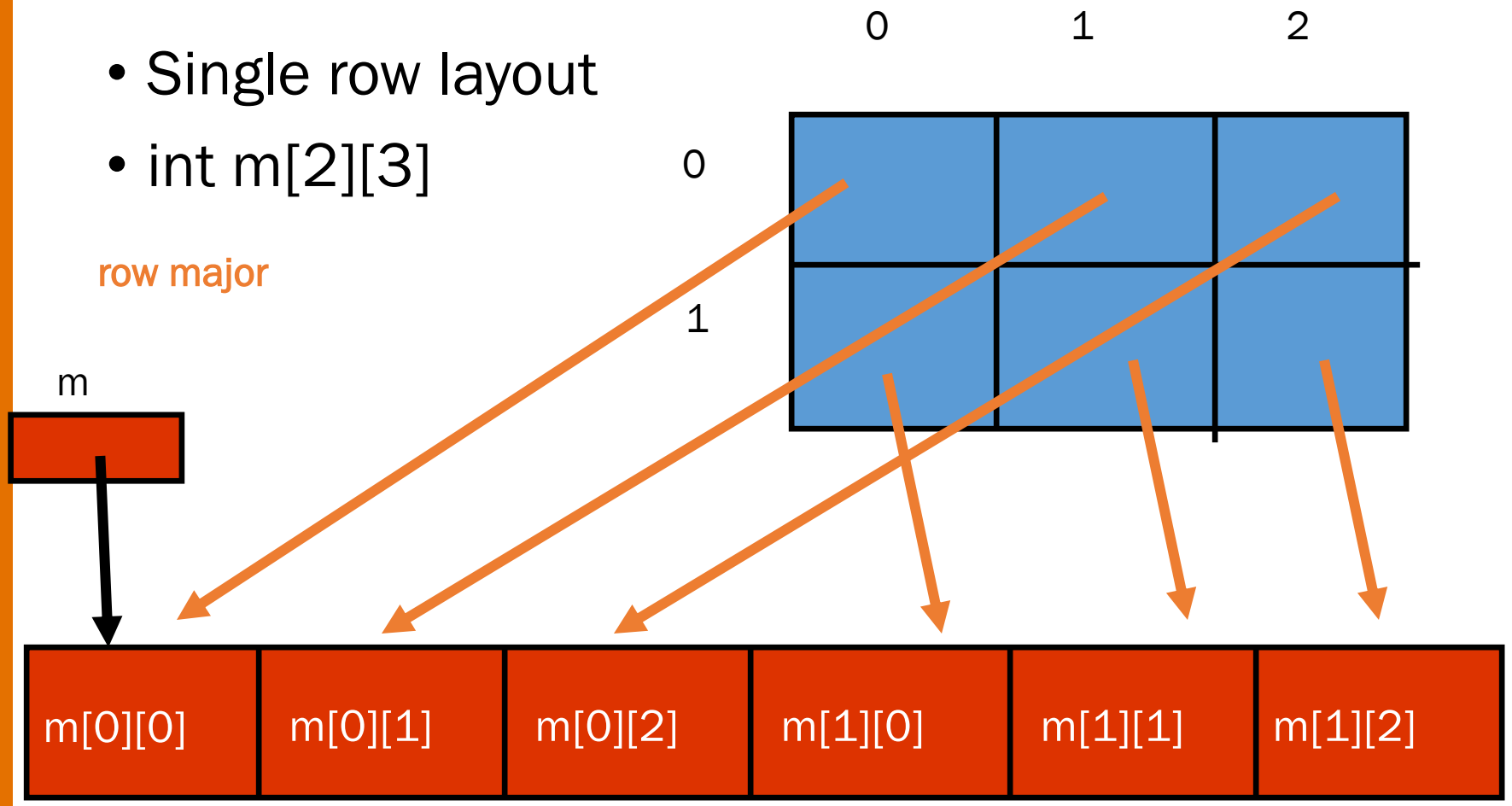
- When a reference to a variable is made, the processor fetches an entire **cache line** from memory, starting at the variable accessed. The size is limited by the hardware and there is usually more than one of them.
- If the next variable referenced is in one of the lines, it will be rapidly available. If it is *not* in the line, an entire line is dumped and a new line is loaded.
- Working within cache is a **cache hit.** Dumping the line and reloading is a **cache miss.**

CACHE EFFICIENCY

- In compiled languages, cache efficiency is critical to optimization.
- Remember that all arrays are just one long block of memory. So order matters!
- Fortran: arrays are guaranteed to be contiguous in memory regardless of the dimension.
- C: N-dimensional arrays A[N][M] must be contiguous but many operations are not defined in the standard.
- C++ we may want to use the STL <array> which is guaranteed to be contiguous in memory.

C (AND C++) ARRAY IMPLEMENTATION

- Single row layout
- int m[2][3]

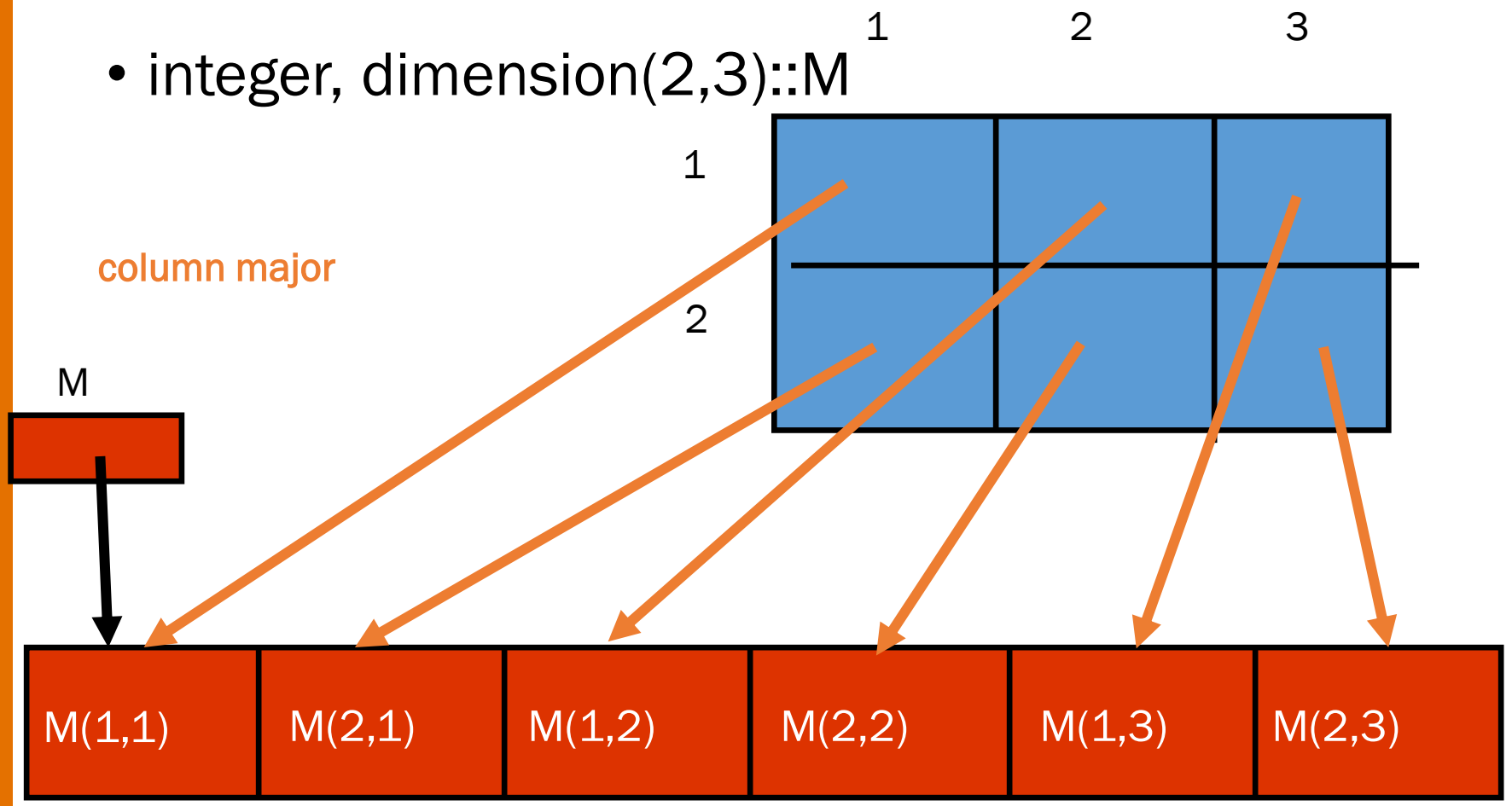


```
int main (void) {
  int array[1024][1024];
  for ( int i = 0; i < 1024; i++ )
       for ( int j = 0; j < 1024; j++ )
            array[i][j] = 0;
  for ( int c = 0; c < 1024; c++ )
       for ( int i = 0; i < 1024; i++ )
            for ( int j = 0; j < 1024; j++ )
                 array[i][j]++;
  int sum = 0;
  for ( int i = 0; i < 1024; i++ )
       for ( int j = 0; j < 1024; j++ )
            sum += array[i][j];

  printf ("%d\n", sum);
}
```

What if we swap the i and j loops?

FORTRAN ARRAY IMPLEMENTATION

- integer, dimension(2,3)::M



```
program cache
implicit none
  integer, dimension(1024,1024) ::  array
   integer                                          ::  sum_A
   integer                                          ::  i, j
  do j=1,1024
       do i=1,1024
            array(i,j)=0
       enddo
   enddo
   do j=1,1024
        do i=1,1024
             array(i,j)=array(i,j)+1
        enddo
   enddo
  sum_A = 0;
  do j=1,1024
        do i=1,1024
              sum_A=sum_A+array(i,j)
        enddo
   enddo
  write(*,*) sum_A
end program cache
```

What if we swap the i and j loops?

EVEN BETTER FOR FORTRAN

- Use array operations.
- Modern compilers optimize them correctly and there is no chance of an array bounds error.
- Note: very early on, compilers were stupid and often carried out Fortran array operations in C order. This has been corrected! On all compilers we've tested, array operations are at least the same speed as correctly ordered loops and are usually faster, sometimes much faster.
- Nevertheless there are *still* Web pages that erroneously claim array operations are slow.

```
program aocache
implicit none
  integer, dimension(1024,1024) ::  array
   integer                      ::  sum_A
   integer                      ::  i, j

   A=0        ! Unneeded, here only for timing consistency
   A=1
   sum_A=sum(A)
  write(*,*) sum_A
end program aocache
```

Results using Unix time command
(wrongcache.f90 uses incorrect looping order):

```
gfortran 4.9.2, -O optimization level:
cache.f90              0.003 sec
wrongcache.f90  0.027 sec
aocache.f90          0.001 sec
```

PYTHON NUMPY ARRAYS

- Row-major ordered by default.
- Can be created to use column-major ordering.
- NumPy Ndarrays are actually objects and so optimization techniques for cache efficiency are less important.
- The major optimization strategy for arrays is to avoid for loops *entirely*.
- The best strategy is to use array operations and "vectorized" built-in functions.

```
import numpy as np

array=np.zeros((1024,1024),dtype=int)

for i in xrange(1024):
    for j in xrange(1024):
        array[i,j]+=1

sum_A=0
for i in xrange(1024):
    for j in xrange(1024):
        sum_A+=array[i,j]

print sum_A
```

```
import numpy as np

array=np.zeros((1024,1024),dtype=int)

array+=1

sum_A=np.sum(array)

print sum_A
```

Results with Python 2.7.8

cache.py 2.37 sec
aocache.py 0.068 sec

OPTIMIZING OPERATIONS

- After aligning to cache as best we can, next we should look for savings in operations.
- Avoid expensive mathematical operations as much as possible.
  - Addition, Subtraction, Multiplication = Very Fast
  - Division and Exponentiation = Slow
  - Mathematical functions such as Square Root = Very Slow
- Example: comparing a value to a square root.
  - Exponentiation is much faster (and in Fortran always use an integer exponent if that's what you mean). Could we compare the squares instead? This works if we don't have to worry about the negative square root.

SIMPLIFY EXPRESSIONS

- This is why you learned algebra!
- Example: combine exponents
- $e^a e^b = e^{a+b}$
- This replaces 2 expensive exponentials with just 1.
- The same is true for excessive divisions, square roots etc.
- Try expanding the equation that your code represents and then refactorizing it with the intention of avoiding as many square roots, exponentials and divisions as possible.

DO NOT RECOMPUTE ("BOOSTING")

- Do not keep adding the same constant in a loop when we know that
- $\sum_{i=1}^{n}(i+a) = na + \sum_{i=1}^{n} i$
- Especially pull constant *multiplicative* factors out of a loop.
- Even more importantly, convert a division into a multiplication whenever possible even if you must defined a new variable.
  - Not necessary to do it for an entire array unless it will be used many times in a loop, but Rinv=1./R is common.
- Do not apply a function over and over to the same data, especially within an inner loop.

COMBINING TWO OPTIMIZATIONS

- Pseudocode:

```
for i=1,100
    for j=1,100
        a[i,j]=d[i,j]/r[i]
```

- Better:

```
for i=1,100
    one_o_r=1./r[i]
     for j=1,100
        a[i,j]=one_o_r*d[i,j]
```

IN NEARLY ALL CASES THE TRADEOFF IS:

- Memory versus speed
  - More memory usage=more speed UNTIL you use enough memory that you slow down (or crash) the entire system
  - More memory usage=fewer options for running (less of a problem nowadays)
  - But if you do use enough memory to slow down the system it will be MUCH slower.
    - Please try to avoid 100000x100000 arrays! Especially in interpreted languages!

SPECIAL ISSUES FOR INTERPRETED LANGUAGES

- You have much less control over cache performance than a programmer using a compiled language.
- You have less control over the data structures; e.g. in Python *everything*, including "primitive" floats, is an object and has overhead associated with that.
- Memory utilization is often much less efficient than in compiled languages.
- If native performance Is unacceptable, write core numerical procedures in C/C++ or Fortran and use built-in tools provided by Python to call these code from within your scripts. Or use Cython or Numba.

PYTHON OPTIMIZATION WITH NUMPY

- `for` loops are very slow in Python (and interpreted languages in general).
- You can often use NumPy to avoid loops.
- However, if you still have loops NumPy may actually be *slower* than lists. You must use the "vectorized" NumPy functions for maximum speed.
- Vectorized operations include array operations as well as most of the many built-in NumPy functions.

PYTHON NUMPY EXAMPLE

```
for i in range(1,n-1):
    for j in range(1,n-1):
        u[i,j]=0.25*(u[i+1,j]+u[i-1,j] \
                     +u[i,j+1]+u[i,j-1])
```

- Can be replaced by

```
u[1:-1,1:-1]=0.25*(u[2:,1:-1]+u[:-2,1:-
1]+\
                   u[1:-1,2:]+u[1:-1,:-2]
```

- This can result in a speedup of up to 250x.

OTHER OPTIONS FOR PYTHON

- Write in a compiled language and use f2py
  - Written for Fortran but can use with C with right bindings
- Write in C/C++ and use Boost Python wrappers or pybind11.
- Cython
  - https://cython.org
- Numba
  - In Anaconda
  - http://numba.pydata.org

BONUS FOR FORTRAN STUDENTS

- A Python user tried f2py on

```
subroutine pairwise_fort(X,D,m,n)
        integer :: n,m
        double precision, intent(in) :: X(m,n)
        double precision, intent(out) :: D(m,m)
        integer :: i,j,k
        double precision :: r
        do i = 1,m
           do j = 1,m
              r = 0
              do k = 1,n
                 r = r + (X(i,k) - X(j,k)) * (X(i,k) - X(j,k))
              end do
              D(i,j) = sqrt(r)
           end do
        end do
     end subroutine pairwise_fort
```

And found it was 2x slower than Numba or Cython. What did he do wrong?