2-operations

June 3, 2024

1 The volatile price of basic operations

Nowadays, it is hazardous to estimate the time taken by an isolated arithmetic operation, in particular because modern processors apply **instruction parallelism**, which covers two aspects : * the use of a "pipeline", which allows to overlap the various stages of the elementary operations, sometimes reducing an operation to an apparent execution time of 1 clock cycle. * the use of several arithmetic units in parallel, when the processor detects independent instructions.

Similarly to data tranfers, one may consider that computation has some kind of **latency**, seen at the beginning of a programm execution, when pipelines and caches are empty, and some kind of **throughput**, which is the mean speed reached when all the hardware facilities are fully operating.

Here are some relative measures from 2015, that give clues about what is expensive and is cheap: - ADD, SUB, MUL, FM: ~0.5 cycle - DIV, SQRT: ~3-4 cycles - EXP, LOG: ~5-6 cycles - SIN, COS: ~10-11 cycles - ATAN: ~22 cycles

1.1 Few tools

The two files below will help to monitor the mean execution time of a given function.

[3]:
```
%%file tmp.time.h

// This works with any function whose return type is void

#include <chrono>
#include <string_view>
#include <iostream>

template< typename Fonction, typename... ArgTypes >
void time( std::string_view title, Fonction f, ArgTypes&&... args )
 {
  using namespace std::chrono ;
  auto t1 {steady_clock::now()} ;

  f(std::forward<ArgTypes>(args)...) ;

  auto t2 {steady_clock::now()} ;
  auto dt {duration_cast<microseconds>(t2-t1).count()} ;
  std::cout<<"("<<title<<" time: "<<dt<<" us)"<<std::endl ;
```

```
}
```

```
Overwriting tmp.time.h
```

[4]:
```
%%file tmp.time.py
#!/usr/bin/env python3

import os, sys
import re
import subprocess
import statistics

NB_RUNS = int(sys.argv[1])
SRC_FILE = sys.argv[2]
RUN_ARGS = ' '.join(sys.argv[3:])

exe_file = SRC_FILE.replace(".cpp",".exe")
compile_cmd = "rm -f {} && g++ -std=c++17 -O3 -march=native {} -o {}".
 ↪format(exe_file,SRC_FILE,exe_file)
run_cmd = "./{} {}".format(exe_file,RUN_ARGS)

# Utility fonction

def run(cmd):
    proc = subprocess.run(
      cmd, stdout=subprocess.PIPE, stderr=subprocess.STDOUT, shell=True,
      executable='bash', universal_newlines=True, check=True
    )
    return proc.stdout.rstrip().split('\n')

# Compile & first run

expr_times = re.compile("^\((.*) time: (.*) us\)$")

times = {}

os.system(compile_cmd)
for line in run(run_cmd):
    match = expr_times.match(line)
    if expr_times.match(line):
        times[match.groups()[0]] = []
    else:
        print(line)

# Repeat timing

for irun in range(NB_RUNS):
```

```
    for line in run(run_cmd):
        match = expr_times.match(line)
        if expr_times.match(line):
            times[match.groups()[0]].append(int(match.groups()[1]))

# Display mean times

for ktime in times:
    #print("({} times: {} s)".format(ktime,times[ktime]))
    print("({} mean time: {} s)".format(ktime,round(statistics.
 ↪mean(times[ktime])/1000000.,3)))
```

```
Overwriting tmp.time.py
```

1.2 The division cost

In the following code, we compare two functions, one which is actively using `x*0.1` and the other which is doing `x/10.`instead. We should get the same result. What do you observe regarding the computation time ?

[5]:
```
%%file tmp.division.1.h

#include "tmp.time.h"
#include <cstdlib>
#include <array>

int const SIZE = 1024 ;
int const REPEAT = 100000 ;

using Array = std::array<double,SIZE> ;

void randomize( Array & x ) {
  srand(1) ;
  for ( int i=0 ; i<SIZE ; ++i )
    x[i] = std::rand()/(RAND_MAX+1.)-0.5 ;
}
```

```
Writing tmp.division.1.h
```

[6]:
```
%%file tmp.division.2.h

void multiply( Array const & x, Array & y ) {
  for ( int r=0 ; r<REPEAT ; ++r )
    for ( int i=0 ; i<SIZE ; ++i )
      y[i] += x[i]*.1 ;
}

void divide( Array const & x, Array & y ) {
```

```
  for ( int r=0 ; r<REPEAT ; ++r )
    for ( int i=0 ; i<SIZE ; ++i )
      y[i] += x[i]/10. ;
}

void reduce( Array const & y ) {
  double res {0.} ;
  for ( int i=0 ; i<SIZE ; ++i ) {
    res += y[i] ;
  }
  std::cout<<(res/SIZE)<<std::endl ;
}
```

```
Writing tmp.division.2.h
```

[7]:
```
%%file tmp.division.cpp

#include "tmp.division.1.h"
#include "tmp.division.2.h"

int main( int argc, char * argv[] )
 {
  Array x, y ;
  randomize(x) ;

  y.fill(0.) ;
  time("multiply",multiply,x,y) ;
  reduce(y) ;

  y.fill(0.) ;
  time("divide",divide,x,y) ;
  reduce(y) ;
 }
```

```
Writing tmp.division.cpp
```

[8]:
```
!python3 tmp.time.py 10 tmp.division.cpp
```

```
67.5054
67.5054
(multiply mean time: 0.098 s)
(divide mean time: 0.118 s)
```

1.2.1 Tips and tricks

When a division by a constant value is applied many times within a loop, you can precompute the inverse of the constant value, and replace the division by a mutiplication with the inverse.

1.3 The conditions cost

In a naive processor, a simple `if` would cost **1 cycle**.

But **modern processors speculate**: they execute some instructions in advance, reversibly. When they finally know if those instructions should have been done, they write resuls in memory or through them away.

A typical speculation is the **branch prediction** for `if`. The processor makes a bet on the conditional value to come. If the bet is false, the *super-pipeline* must be emptied and the cost of this mess can go up to **hundreds cycles**.

Let's introduce two fake conditions in our example.

[24]:
```
%%file tmp.if.1.h

#include "tmp.time.h"
#include <cstdlib>
#include <array>

int const SIZE = 1024 ;
int const REPEAT = 100000 ;

using Array = std::array<double,SIZE> ;

void randomize( Array & x, Array & pos, Array & neg ) {
  srand(1) ;
  for ( int i=0 ; i<SIZE ; ++i ) {
    x[i] = std::rand()/(RAND_MAX+1.)-0.5 ;
    pos[i] = (x[i]>=0.) ;
    neg[i] = (x[i]<0.) ;
  }
}
```

```
Overwriting tmp.if.1.h
```

[25]:
```
%%file tmp.if.2.h

void multiply( Array const & x, Array & y ) {
  for ( int r=0 ; r<REPEAT ; ++r )
    for ( int i=0 ; i<SIZE ; ++i )
      y[i] += x[i]*.1 ;
}

void multiply_if
 ( Array const & x,
   Array const & pos, Array const & neg,
   Array & ypos, Array & yneg ) {
  for ( int r=0 ; r<REPEAT ; ++r ) {
    for ( int i=0 ; i<SIZE ; ++i ) {
```

```
      if (pos[i]) ypos[i] += x[i]*.1 ;
      if (neg[i]) yneg[i] += x[i]*.1 ;
    }
  }
}
```

```
Overwriting tmp.if.2.h
```

[26]:
```
%%file tmp.if.3.h

void reduce( Array const & y ) {
  double res {0.} ;
  for ( int i=0 ; i<SIZE ; ++i ) {
    res += y[i] ;
  }
  std::cout<<(res/SIZE)<<std::endl ;
}

void reduce( Array const & ypos, Array const & yneg ) {
  double res {0.} ;
  for ( int i=0 ; i<SIZE ; ++i ) {
    res += ypos[i] ;
    res += yneg[i] ;
  }
  std::cout<<(res/SIZE)<<std::endl ;
}
```

```
Overwriting tmp.if.3.h
```

[27]:
```
%%file tmp.if.cpp

#include "tmp.if.1.h"
#include "tmp.if.2.h"
#include "tmp.if.3.h"

int main( int argc, char * argv[] )
 {
  Array x, pos, neg, y, ypos, yneg ;
  randomize(x,pos,neg) ;

  y.fill(0.) ;
  time("multiply",multiply,x,y) ;
  reduce(y) ;

  ypos.fill(0.) ; yneg.fill(0.) ;
  time("multiply if",multiply_if,x,pos,neg,ypos,yneg) ;
  reduce(ypos,yneg) ;
```

```
}
```

```
Overwriting tmp.if.cpp
```

[28]:
```
!python3 tmp.time.py 10 tmp.if.cpp
```

```
67.5054
67.5054
(multiply mean time: 0.008 s)
(multiply if mean time: 0.16 s)
```

1.3.1 Tips and tricks

When you know that a given condition is hardly predictable, it may proves efficient to replace it with a raw-force computation.

[29]:
```
%%file tmp.if.2.h

void multiply_if
 ( Array const & x,
   Array const & pos, Array const & neg,
   Array & ypos, Array & yneg ) {
  for ( int r=0 ; r<REPEAT ; ++r ) {
    for ( int i=0 ; i<SIZE ; ++i ) {
      if (pos[i]) ypos[i] += x[i]*.1 ;
      if (neg[i]) yneg[i] += x[i]*.1 ;
    }
  }
}
void multiply_bool
 ( Array const & x,
   Array const & pos, Array const & neg,
   Array & ypos, Array & yneg ) {
  for ( int r=0 ; r<REPEAT ; ++r ) {
    for ( int i=0 ; i<SIZE ; ++i ) {
      ypos[i] += pos[i]*x[i]*.1 ;
      yneg[i] += neg[i]*x[i]*.1 ;
    }
  }
}
```

```
Overwriting tmp.if.2.h
```

[32]:
```
%%file tmp.if.cpp

#include "tmp.if.1.h"
#include "tmp.if.2.h"
#include "tmp.if.3.h"
```

```
int main( int argc, char * argv[] )
 {
  Array x, pos, neg, ypos, yneg ;
  randomize(x,pos,neg) ;

  ypos.fill(0.) ; yneg.fill(0.) ;
  time("multiply if",multiply_if,x,pos,neg,ypos,yneg) ;
  reduce(ypos,yneg) ;

  ypos.fill(0.) ; yneg.fill(0.) ;
  time("multiply bool",multiply_bool,x,pos,neg,ypos,yneg) ;
  reduce(ypos,yneg) ;
 }
```

```
Overwriting tmp.if.cpp
```

[33]:
```
!python3 tmp.time.py 10 tmp.if.cpp
```

```
67.5054
67.5054
(multiply if mean time: 0.579 s)
(multiply bool mean time: 0.062 s)
```

1.4 The function call cost

Note the significant additional cost of a virtual function, and therefore the interest of avoiding object polymorphism in the lowest layers of applications : * ordinary function call : 3 cy * **virtual function call : ~90 cy !**

[9]:
```
%%file tmp.virtual.1.h

#include "tmp.time.h"
#include <cstdlib>
#include <vector>
#include <memory>

int const SIZE = 1024 ;
int const REPEAT = 100000 ;

struct XY1 {
  double x, y = 0. ;
  void saxpy( double a ) { y += x*a ; }
 } ;

struct XY2 {
  double x, ypos = 0., yneg = 0. ;
  virtual void saxpy( double ) = 0 ;
```

```
 } ;

struct XY2pos : public XY2 {
  void saxpy( double a ) { ypos += x*a ; }
 } ;
struct XY2neg : public XY2 {
  void saxpy( double a ) { yneg += x*a ; }
 } ;

using XY1s = std::vector<std::unique_ptr<XY1>> ;
using XY2s = std::vector<std::unique_ptr<XY2>> ;
```

```
Writing tmp.virtual.1.h
```

[10]:
```
%%file tmp.virtual.2.h

#include <memory>

void randomize( XY1s & xys ) {
  srand(1) ;
  for ( int i=0 ; i<SIZE ; ++i ) {
    xys.emplace_back(std::make_unique<XY1>()) ;
    xys[i]->x = std::rand()/(RAND_MAX+1.)-0.5 ;
  }
}

void randomize( XY2s & xys ) {
  srand(1) ;
  for ( int i=0 ; i<SIZE ; ++i ) {
    double x = std::rand()/(RAND_MAX+1.)-0.5 ;
    if (x>=0)
      xys.emplace_back(std::make_unique<XY2pos>()) ;
    else
      xys.emplace_back(std::make_unique<XY2neg>()) ;
    xys[i]->x = x ;
  }
}
```

```
Writing tmp.virtual.2.h
```

[11]:
```
%%file tmp.virtual.3.h

template< typename Array >
void saxpy( Array const & xys ) {
  for ( int r=0 ; r<REPEAT ; ++r )
    for ( int i=0 ; i<SIZE ; ++i )
      xys[i]->saxpy(.1) ;
}
```

```
void reduce( XY1s const & xys ) {
  double res {0.} ;
  for ( int i=0 ; i<SIZE ; ++i ) {
    res += xys[i]->y;
  }
  std::cout<<(res/SIZE)<<std::endl ;
}

void reduce( XY2s const & xys ) {
  double res {0.} ;
  for ( int i=0 ; i<SIZE ; ++i ) {
    res += xys[i]->ypos ;
    res += xys[i]->yneg ;
  }
  std::cout<<(res/SIZE)<<std::endl ;
}
```

```
Writing tmp.virtual.3.h
```

[12]:
```
%%file tmp.virtual.cpp

#include "tmp.virtual.1.h"
#include "tmp.virtual.2.h"
#include "tmp.virtual.3.h"

int main( int argc, char * argv[] )
 {
  XY1s xy1s ;
  randomize(xy1s) ;
  time("static call",saxpy<XY1s>,xy1s) ;
  reduce(xy1s) ;

  XY2s xy2s ;
  randomize(xy2s) ;
  time("virtual call",saxpy<XY2s>,xy2s) ;
  reduce(xy2s) ;
 }
```

```
Writing tmp.virtual.cpp
```

[13]:
```
!python3 tmp.time.py 10 tmp.virtual.cpp
```

```
67.5054
67.5054
(static call mean time: 0.204 s)
(virtual call mean time: 0.417 s)
```

1.4.1 Tips and tricks

If you do not need the run-time flexibility and extensibility offered by the object-oriented polymophism, prefer the use of generic programming features: templates, `std::variant`, etc.

1.5 The cost of mathematical functions

Regarding functions such as `sin`, `cos`, `exp`, `log`, … their costs are specific to the math library used (which itself can use fast hardware instructions and/or slow software refinement depending on the precision that is required). At the back of this pack, **`pow` costs really a lot**. And **even worse, the *special* functions** (bessel, ellipticals, polylog, dzeta, hypergeometric …).

For all of these functions, it is important to note that they do not necessarily provide the expected accuracy of the float type used, especially with the universally used library `libm`. When maximum precision is needed on these functions, **alternative libraries should be used** (`libultim`,`libmcr`, `CRlibm`) should be used.

On the contrary, if accuracy is not so important in your use-case, and if you do not care too much about standards compliance, then the compiler option **`-Ofast` may boost your performance**.

2 Take away

We must confess we had a very hard time to write code examples which demonstrate the expected relative cost between multiplication, division, if, etc. The CPU advanced features, combined with compiler advanced features, make the execution time hardly predictable. If you take previous examples, and **mix them all**, with different compiler options and hardware, **you will be often puzzled**.

The only way to know the best options, for your own application and platform, is to **test and compare alternatives**.

This is tedious, and the optimization tricks deteriorate the code readibility, portability, robustness. **You must only optimize the code section where it is really worth**.

So, **first and foremost, you must profile your code**, and identify the bottlenecks.

3 Questions ?

4 Exercice

Let's assume that we always run the program below with a `degree`which is even. 1. Replace `val *= data` with `val *= data*data`, and `j<degree` with `j<(degree/2)`. What is the effect on the execution time ? 2. Ask for `std::cout.precision(18)`, and compare the original and the previous implementation. Why the result is slightly differing ? 3. Try to compile with `-Ofast`.

[24]:
```
%%file tmp.operations.cpp

#include <valarray>
#include <cstdlib>
#include <cassert>
```

```cpp
#include <iostream>

std::valarray<double> generate( int size )
 {
  std::valarray<double> datas(size) ;
  for ( double & data : datas )
   { data = std::rand()/(RAND_MAX+1.) ; }
  return datas ;
 }

double pow_reduce( std::valarray<double> const & datas, int degree )
 {
  double res = 0 ;
  for ( double data : datas )
   {
    double val = 1 ;
    for ( int j=0 ; j<degree ; ++j )
      val *= data ;
    res += val ;
   }
  return res ;
 }

int main( int argc, char * argv[] )
 {
  assert(argc==3) ;
  int size {atoi(argv[1])} ;
  int degree {atoi(argv[2])} ;

  auto datas = generate(size) ;
  auto res = pow_reduce(datas,degree) ;
  std::cout << res << std::endl ;
 }
```

```
Overwriting tmp.operations.cpp
```

[30]:
```
%%file tmp.operations.sh

rm -f tmp.operations.exe
g++ -std=c++17 -O3 tmp.operations.cpp -o tmp.operations.exe
\time -f "(%U s)" ./tmp.operations.exe $*
```

```
Overwriting tmp.operations.sh
```

[31]:
```
!bash -l ./tmp.operations.sh 1024 65536
```

```
0.656155
(1.05 s)
```

4.1 References

- Processeurs super-scalaires
- Agner Fog Instruction Tables
- Branch predictor
- As if rule