Faster Code

Nicolas Limare

2014/11/19

faster?

one task *vs* many speeds

- one operation *vs* many algos
- one algo *vs* many codes
- one code *vs* many binaries
- one binary *vs* many runs

We want *faster* runs for the same operation.

why go fast?

- **massive data** 1 year processing too long/expensive
- **data flow** process one dataset before the next one
- **testing** try many variants, find the best

10× speedup?

Moore's law:

"*computers 2× faster every 2 years*"

. . .

10× slower → 6 years old

. . .

Wirth's law:

"*software is getting slower more rapidly than hardware becomes faster*"

cost of sluggishness

- difficult to be "*as good as others, but faster*"
  or "*as fast as others, but better*"
- can't explore a new algorithm in detail
- pay too much in computer hardware
  (or can't pay)
- can't run all tests before deadline
  (or miss deadline)
  (or fake tests)

disclaimers

- only when limited by computation time
- tradeoff, development time *vs* execution time
- some hard science: how computers work
- lots of know-how:
  read (good) books, read (good) web,
  try, retry, test and compare

disclaimers

- other good habits help:
  clean and correct code,
  well defined, well documented,
  meaningful units, ...

plan

- general method
- useful tools
- hints & examples

:(

- Q: Nice presentation. How fast is your algorithm?
- A: Well, now it's very slow, but it could probably be faster, we didn't try to be fast...

method

work on stable algorithm

perform the *same* task, but faster.

- *same* task: can't hit a moving target
- alternate algo → opti → algo → opti → ...
- or work on *stable* subsystems

it works, don't break it

- after *every* change, check you didn't break the program
- automated verification
- *small* and *fast* (seconds) computation example
- good example using every part of the code

random number generator

Add option for fake init.

. . .

parameter

```
flag_random_init = true; /* global */
if (0 == strcmp(argv[i],"--no-random-init"))
flag_random_init = false;
....
srand(flag_random_seed ? time() : 0);

$ progname --no-random-init ...
```

. . .

environment variable

```
flag_random_init = true; /* global */
if (NULL == getenv("NO_RANDOM_INIT"))
flag_random_init = false;
....
srand(flag_random_seed ? time() : 0);

$ NO_RANDOM_INIT=1 progname ...
```

```
$ export NO_RANDOM_INIT=1
$ progname ...
```

same task, different same result

The exact result may change:

- different precisions
- reordered operations
- faster approximations

May not be a problem, but need to be checked anyway.
Set target max/mean/quantile error, check and review when needed.

floating-point differences

How accurate is your result?

. . .

ULPs: Units in the Last Place

"*how many other floating-point numbers between our result and the correct result?*"

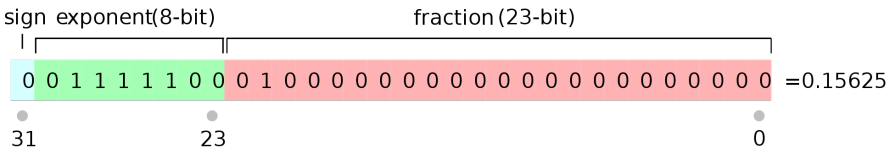


Figure 1: IEEE754 format

- difficult to compute distances correctly
- what is the acceptable distance?

floating-point differences

Number of common digits

```
def ndigits(x):
    return -int(round(log(abs(x))/log(10)))
def precision(a, b):
```

```
if a * b < 0:
    return 0
if a == b:
    return 16 # or 8
if a == 0:
    return ndigits(b)
else:
    return ndigits((b-a)/a)
```

- simple txt output: float: `"%+1.8e"`, double: `"%+1.16e"`
- precision target: maximum, mean, quantiles
- check and adjust when needed

good timer

perform the same task, but *faster*.

GHz frequency : 10^9 cycles (ops) / sec

- system/shell `time`
  low precision (msec)
  can't measure a subset of the program
  need scripting to collect and process timing results
- C `clock()` and `time()`
  low precision (msec)
- UNIX `clock_gettime()` and `gettimeofday()`
  better precision (µsec)
  Windows equivalent

wallclock time *vs* CPU time

- wallclock time: elapsed in "real world"
- CPU time: used for this process by every CPU

. . .

Ideal N CPU

- CPU time = N × wallclock time
- `OMP_WAIT_POLICY`

. . .

Ideal N→2N CPU

- CPU time unchanged
- wallclock time / 2

good time measures

- many measures, median
  (how many??)
- stable CPU frequency (laptops)

```
echo performance | /sys/devices/system/cpu/cpu0/cpufreq/scaling_governor
```

. . . or BIOS

good time measures

single (active) user, non-competing processes

```
$ ./train
TIME  [loop ] cpu:0.051391 elapsed:0.025782
$ ./train & ./train
TIME  [loop ] cpu:0.201783 elapsed:0.202419
TIME  [loop ] cpu:0.200949 elapsed:0.201396
```

→ 10× slower!!

. . .

no "virtual" CPU

```
echo 0 | /sys/devices/system/cpu/cpu0/online
```

. . . or BIOS

step by step

- sum of small accelerations
- 5% or 10% is worth taking (10× 10% speedup = 260%)
- les meilleures accelerations viennent au début

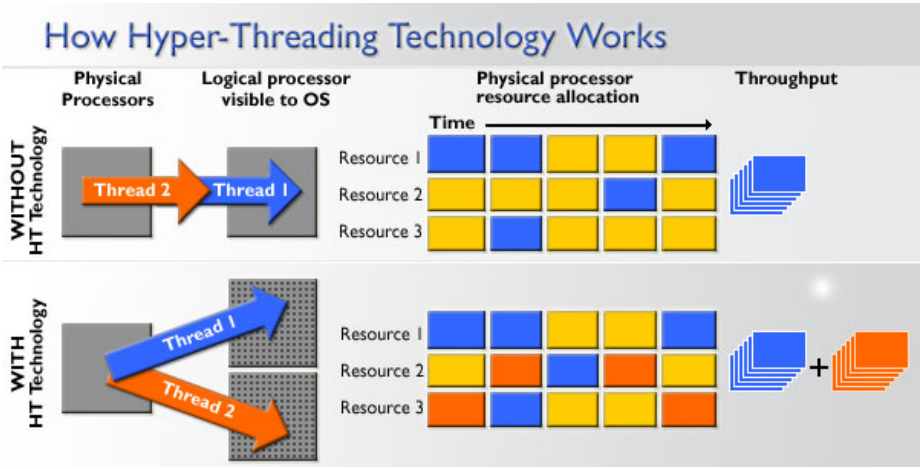


Figure 2: HyperThreading

don't get lost

- work on small and complete code changes
  one idea = one code version
- store each of them with a description
- check correctness and speed for every change
- get back, test, correct, cancel, ...

automation

makefile rules or scripts

```
$ make lint      # check language correctness
$ make test      # check result correctness
$ make timing    # look at the speed
$ make profiling # find hotspots
```

. . .

including options

```
$ make test
$ make test-memory
$ make test-regression
$ make test-largedata
$ make timing WITH_PRECISION=double
```

```
$ make timing WITH_BLAS=mkl
$ OMP_NUM_THREADS=2 make timing
```

. . .

re-run

every code version with git+make on new hardware/compiler/library

tools

basics

shell and text processing (grep, sort, cut, sed, . . .)

make with variables and beyond compilation

git commit, branch, rebase

fast (re)compilation

ccache

"*ccache is a compiler cache. It speeds up recompilation by caching previous compilations and detecting when the same compilation is being done again.*"

- https://ccache.samba.org/
- `aptitude install ccache`

. . .

```
$ ccache gcc ...
$ alias gcc="ccache gcc"; gcc ...
$ export PATH=/usr/lib/ccache/:$PATH; gcc ...
```

. . .

```
$ ccache -C
Cleared cache
$ time make -B
0m26.561s
$ time make -B
0m0.706s
```

timing

timing.c

Macros to collect wallclock time and CPU time with µsec precision (and count CPU cycles). Multiple counters, UNIX/Windows, activated by CPP macro (`-DUSE_TIMING`).

- http://dev.ipol.im/~nil/tmp/timing_20141119.tgz

. . .

```
TIMING_WALLCLOCK_START(N)
TIMING_WALLCLOCK_TOGGLE(N)
TIMING_WALLCLOCK_S(N)

TIMING_CPUCLOCK_...

TIMING_CYCLE_...

TIMING_PRINTF(...)
```

Test and examples in `timing-test.cpp`.

timing

```
$ make timing
...
TIME  [mmprc] cpu:0.013982 elapsed:0.007003
TIME  [mmT  ] cpu:0.015627 elapsed:0.008811
TIME  [mTma ] cpu:0.017627 elapsed:0.012130
TIME  [tanh ] cpu:0.003602 elapsed:0.002116
TIME  [sum  ] cpu:0.002379 elapsed:0.001198
TIME  [omsq ] cpu:0.002057 elapsed:0.001024
TIME  [rand ] cpu:0.001104 elapsed:0.000552
TIME  [patch] cpu:0.000595 elapsed:0.000308
TIME  [axpb ] cpu:0.000029 elapsed:0.000014
TIME  [crop ] cpu:0.000025 elapsed:0.000012
TIME  [down ] cpu:0.000000 elapsed:0.000000
TIME  [mosa ] cpu:0.000000 elapsed:0.000000
TIME  [loop ] cpu:0.059324 elapsed:0.034330
TIME  [gemm ] cpu:0.042940 elapsed:0.023993
```

profiling, single thread

Google profiler

Run the program in real-time, N times/sec look at which instruction is being executed, gather stats and analyze.

- https://code.google.com/p/gperftools/
- `aptitude install google-perftools`

. . .

```
$ LD_PRELOAD=/usr/lib/libprofiler.so CPUPROFILE=train.pprof
  CPUPROFILE_FREQUENCY=1000 ./train ...
$ pprof --text ./train train.pprof           # sorted functions
$ pprof --list=mtanh  ./train train.pprof    # source lines
$ pprof --disasm=mtanh  ./train train.pprof  # assembly
```

Compile with `-g` (larger, not slower).
Optimize no more than `-Og` (don't remove variables and functions).

profiling, multi-thread

Linux perf tools

"*perf is a performance analyzing tool* in *Linux.*"

- https://perf.wiki.kernel.org/
- `aptitude install linux-tools`

. . .

```
$ perf record -g -o train.perf -- ./train ...
$ perf report -i train.perf
```

Focus on one DSO (Dynamically Shared Object)
Annotate per function (source/asm)

hints & examples

use best (latest) CPU

exact same code and compilation:

- Xeon X7560 @2.27GHz
  0.465s - 0.194s/GHz
- Xeon E5 2650v2 @2.60GHz
  0.192s - 0/073s/GHz

use best (latest) CPU

exact same code and compilation:

- Xeon X7560 @2.27GHz : **2010, 24M cache, SSE4.2**
  0.465s - 0.194s/GHz
- Xeon E5 2650v2 @2.60GHz : **2013, 25M cache, AVX**
  0.192s - 0/073s/GHz

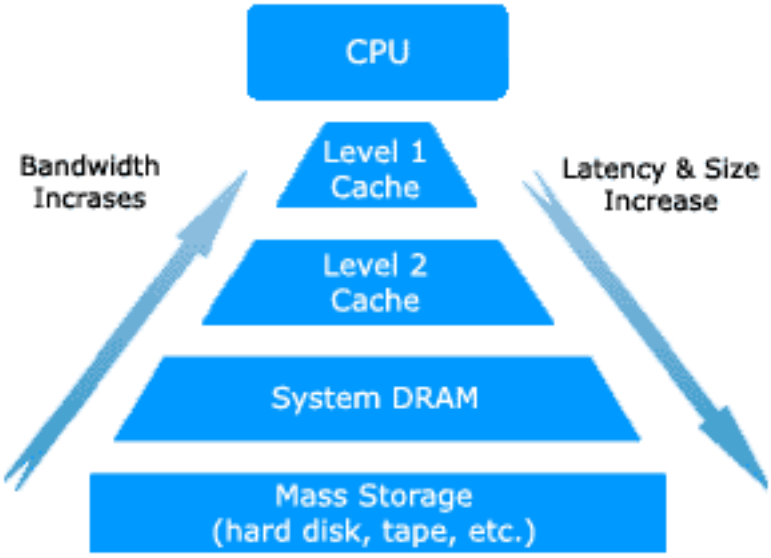
vector instructions (SIMD)

- **MMX**, **SSE**, ...
- **AVX** (2011): 256-bit vector ops on floating-point (8 float, 4 double)
- **AVX2** (2013): 256-bit vector ops on integer (8 int, 4 long)
- **AVX512** (2015): 512-bit vector ops on floats, 2^x, 1/x, 1/sqrt(x) (16 float, 8 double)

CPU cache

data access latency

L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Main memory reference	100 ns
Read 1 MB from RAM	250,000 ns
Disk seek	10,000,000 ns
Read 1 MB from disk	20,000,000 ns



use best (latest) compiler

exact same code and machine:

	gcc-4.4	gcc-4.9
tanh	0.00389s	0.00250s
sum	0.00122s	0.00117s
omsq	0.00120s	0.00118s
rand	0.00118s	0.00097s

compiler options?

- `-O2` / `-O3`
- `-ffast-math`
- `-ftree-vectorize`
- `-march=native`

use best libraries

example: linear algebra

- Eigen vs BLAS
- Eigen vs simple loops
- Blas/OpenBLAS/Atlas/MKL

cost of simple ops

```
# 100000x
# Math ops, single precision
f = .1 + fi            0.000004s  (0.000796 - 0.000916)
f = .1 * fi            0.000000s  (0.000813 - 0.000945)
f = .1 / fi            0.000044s  (0.000868 - 0.000897)
```

- +,*
- /
- sqrt()
- sin(),exp()
- tanh(),sinh()

float vs double

```
# 100000x
# Math ops, single precision
f = sqrtf(fi)          0.000058s  (0.000882 - 0.000922)
f = sinf(fi)           0.001100s  (0.001917 - 0.002163)
f = expf(fi)           0.001545s  (0.002348 - 0.002628)
f = tanf(fi)           0.002278s  (0.003091 - 0.003452)
# Math ops, double precision
d = sqrt(di)           0.000284s  (0.001108 - 0.001145)
d = sin(di)            0.002716s  (0.003527 - 0.003954)
d = exp(di)            0.002739s  (0.003534 - 0.004001)
d = tan(di)            0.004229s  (0.005035 - 0.005616)
```

- float faster than double
- implicit cast

```
f = sqrt(fi)           0.000331s  (0.001135 - 0.001284)
f = sin(fi)            0.002503s  (0.003283 - 0.003653)

out[i]=1.1 *data[i]+0.42;     0.006291s  (0.006196 - 0.006641)
out[i]=1.1f*data[i]+0.42f;    0.005619s  (0.005594 - 0.005750)
```

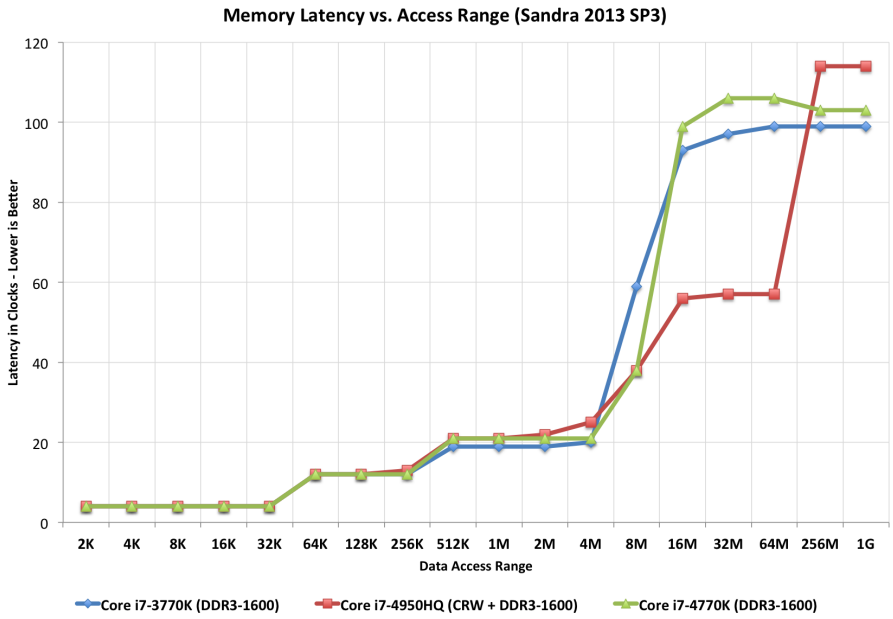
8 rules

rule 1 : smaller

- less data : fast RAM→CPU transfer, few cache miss
  single-precision: 2× smaller, 2× faster SIMD

rule 2 : simpler

- less code (DRY) : concentrate hotspots, optimize once
  simple, minimal code easier to understand, analyze and optimize
- inline core functions (not macro)



- fixed-size arrays/loops
- use system functions

```
0.005711s  for(i=0;i<size*size;i++) out[i]=data[i];
0.001774s  memcpy(out,data,sizeof(data));
0.008082s  for(i=0;i<size*size;i++)out[i]=0.;
0.000907s  memset(out,0,sizeof(out));
```

(IEEE754 floating-point zero is 0000000000...)

rule 3 : compute once

- table lookup (+ interpolation)
- enrich data with common computations

rule 4 : read once

- merge successive loops on same data to limit cache miss
- read in order
- images, interlaced or not

```
0.026045s  for(i=...) for(j=...) sum+=data[i+size*j];
0.004989s  for(j=...) for(i=...) sum+=data[i+size*j];
0.016950s  for(i=...) out[i]=data[i]; for(...) out[i]*=1.1; for(...) out[i]+=0.42;
0.006291s  for(i=...) out[i]=1.1*data[i]+0.42;
```

- compute max/min together
- store files in RAM: ramdisk or preread

rule 5 : minimal loops

- move computions and tests out of loops
- merge nested loops, 1D arrays
- no test in loops: borders, test as math

```
0.004989s  for(j=0;j<size;j++)for(i=0;i<size;i++)sum+=data[i+size*j];
0.004800s  for(i=0;i<size*size;i++)sum+=data[i];
```

- no test in loops: jump borders, test as math

```
0.011178s  for(j=1;j<size-1;j++)for(i=...) out[i+size*j]=data[i+1+size*j]+data[i-1+...]
0.008630s  for(i=size;i<size*(size-1);i++) out[i]=data[i+1]+data[i-1]...
```

```
0.012465s  for(i=...) out[i]=data[i];else out[i]=1.f-data[i];
0.005690s  for(i=...) out[i]=.5f-fabsf(data[i]-.5f);
```

rule 6 : use maths

- a/b -> a * 1/b
- monotone functions: log(x)<1 -> if x < e; squared dist
- approx. expensive functions, interpolation
- choose efficient cases: FFT

rule 7 : order tests

- cheap tests first
- order data for test prediction

```
0.009613s  for(i=...) if(data[i]>.5f) sum+=data[i]; else sum+=1.f-data[i];
0.004829s  qsort(...);for(i=...) if(data[i]>.5f) sum+=data[i]; else sum+=1.f-data[i];
```

rule 8 : parallel processing

- parallelize on large, uniform loops
- overhead and minimum block size

. . .

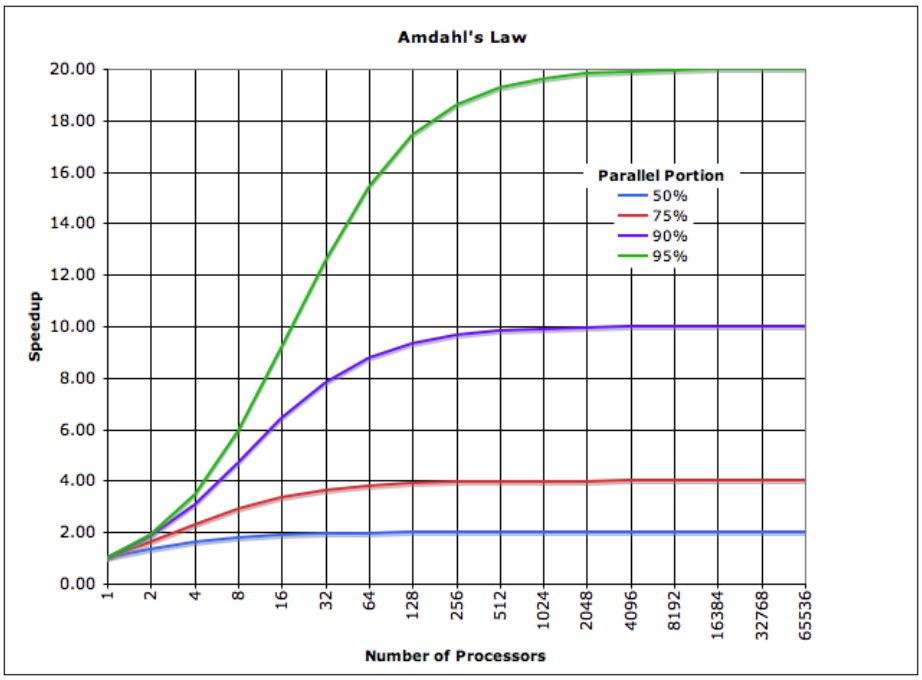


Figure 3: Amdahl's law

3 examples

example: tanh.hpp

- lookup table
- variable precision
- static variables *vs* guard
- precompute diff
- interlaced diff
- inline

example: rand.hpp

- fast, simple PRNG
- float-only
- keep useful results
- init() before main()
- inline

example: matops.cpp

- blas
- 1D array loop
- (reorder rsum)