CPU optimisation

Stewart Martin-Haugh (RAL)

Efficient Computing in High Energy Physics
17 February 2020

Science and Technology Facilities Council

Overview

- Problem domain
- HEP codebases (mainly LHC experiments)
- A grab-bag of tools useful for improving CPU/memory performance
- More details/pedagogy available in lecture format

Computing domains

High throughput computing

- Can parallelise and buffer data for later processing
- LHC, SKA - **this talk**
- Maximise throughput = events/second

Low latency computing

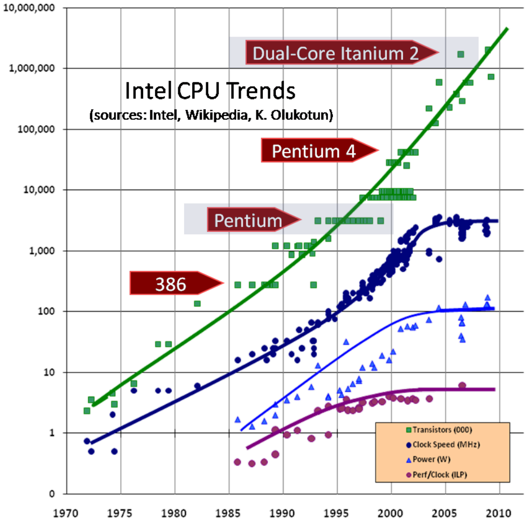
- Pointless or impossible to buffer
- High frequency trading, autonomous vehicles

High performance computing

- Problems that don't parallelise easily - supercomputer
- Climate modelling
- Fast connections between processors, lots of RAM

Hardware overview

- Moore's law stopped helping us some time ago
  - 2003 ATLAS TDAQ TDR estimated 8 GHz dual-core machines
  - In practice, ended up with multi-core 2.3 GHz machines
  - Memory per core has decreased
- Need multi-threading to make memory-efficient use of many cores



Source: Herb Sutter

Architectures

x86

- Largely build our code for x86-64 Red Hat Linux
- Using the grid restricts us to low common denominator CPU features - newer instruction sets (AVX in particular) not used

Other

- Port of ATLAS simulation to PowerPC to use Oak Ridge Summit supercomputer
- Ports of ATLAS, LHCb to ARM

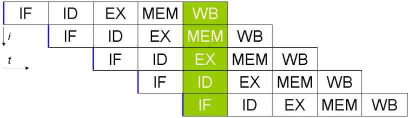
HEP codebases

Language	Files	Lines of code
CMSSW		
C++	30050	3265146
Python	13699	1453740
Athena (ATLAS)		
Language	Files	Lines of code
C++	39603	3884549
Python	10624	1148465

- 3M lines of C++, 1M lines of Python
- Multi-hour build times to build entire codebase
- Code written by 100s of people over 20+ years of development
- Barrier to
  - Determining what code is running at all
  - Determining what code is running slowly
  - Migrating to new libraries (e.g. matrix algebra)

Optimisation guidelines

- Start with data from a profiler: don't try to reason about the code
- Ensure data locality - try to read array elements in sequence `a[0], a[1], a[2]`
- CPU has long pipelines - Branch misprediction has a big penalty



Five stage Instruction pipeline

- Prefer vector to list, unordered_map to map
- Always reserve() vectors

Floating-point operations

Caveat: arithmetic usually in the shadow of memory access

- Addition is faster than multiplication (usually compiler will do this for you if needed)
- Multiplication is faster than division

```
y=x/5.0; //Bad
y=x*0.2; //Good
```

- Rearrange calculations to minimise number of operations
- Compiler won't do this for you without **-Ofast**

```
y = d*x*x*x + c*x*x + b*x + a; //Bad
y = x*(x*(x*d+c)+b) + a;       //Good
```

- Reducing operations and branching

```
if ( h >= 0.) {//Bad
  h = min( max( 0.25*h, pow((x / y), 0.25)*h), 4.*h);
} else {
  h = max( min( 0.25*h, pow((x / y), 0.25)*h), 4.*h);
}
h = h*min( max( 0.25, pow((x / y), 0.25)), 4.);//Good
```

CPU profiling

Sampling

- interrupting with a debugger and generating a stack trace
- Some measurement overhead (depending on frequency of interruption)
- Can generate various visualisations
- Intel VTune, gperftools, igprof

Emulating

- Callgrind (part of Valgrind) is only one I know of
- Emulates a basic modern CPU, with level 1, level 2 caches, branch prediction (somewhat configurable)
- Runs slowly, no measurement overhead
- Information about cache misses and branch misprediction

CPU profiling

Instrumenting

- `perf` is now the gold standard - sampling and instrumenting
- Part of Linux kernel (best results with new kernels)
- Monitor CPU performance monitoring counters
- Also possible with VTune
  - Some features require root access

```
perf stat -d program
     10 152 172 182       cycles:u                         #
        3,451 GHz                          (49,86%)
     14 584 154 073       instructions:u                   #
        1,44  insn per cycle               (62,43%)
      2 318 605 154       branches:u                       #
         788,130 M/sec                        (74,93%)
         44 768 463       branches-misses:u                #
            1,93% of all branches              (75,00%)
      4 116 170 377       L1-dcache-loads:u                #
         1399,150 M/sec                        (74,18%)
        167 821 302       L1-dcache-load-misses:u   #
           4,08% of all L1-dcache hits    (25,06%)
         45 252 042       LLC-loads:u                      #
            15,382 M/sec                         (24,89%)
```

Instrumentation

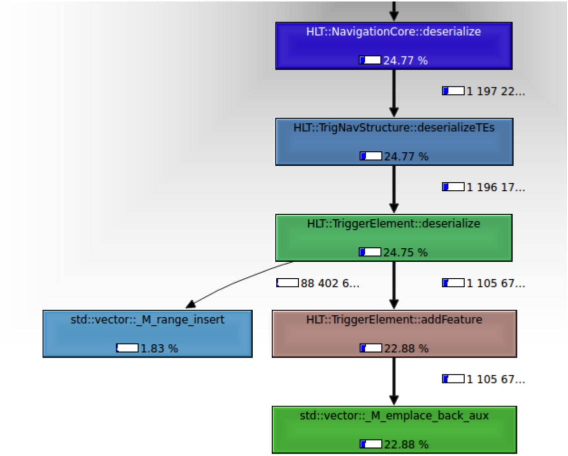
- C++ has inbuilt timing facilities:

```
using namespace std;
using namespace std::chrono;
auto start_time = high_resolution_clock::now();
doSomething();
auto end_time = high_resolution_clock::now();
cout << "Time: " << duration_cast<microseconds>(end_time -
    start_time).count() << endl;
```

- Useful, but has some overhead: shouldn't try to measure within tight loops
- Google Benchmark builds this into a useful framework to benchmark functions

Heap profilers

- **jemalloc** and **tcmalloc** both come with low-overhead profilers to analyse which functions allocate most memory
- Output can be interpreted like a call-graph



Other memory analysis

- Is your code allocating short-lived heap variables, or writing variables it never reads?
  - Experimental tools, e.g. Find Obsolete Memory (FOM)

Optimisation example/cautionary tale

- Always implement something correct and readable first
- **Then** you can have fun optimising

Courtesy of reddit

```
//I don't know what I did but it works
//Please don't modify it
private int square(int n)
{
    int k = 0;
    while (true)
    {
        if (k == n*n)
        {
            return k;
        }
        k++;
    }
}
```

Optimisation example

- GCC and Clang compilers can reduce square example[1] down to something sensible

```
int square(int n)
{
  int k = 0;
  while (true)
  {
    if(k == n*n)
    {
      return k;
    }
    k++;
  }
}
```

→

```
int square2(int n)
{
    return n*n;
}
```

- Don't second-guess the compiler: profile
- But don't keep obviously inefficient code if it will puzzle the next reader

[1] NB: Don't write a square function, just square numbers in the code

Automatic compiler optimisation

Choice of compiler

- Clang and GCC seem to give similar results on ATLAS reconstruction worklads
- Porting to other compilers (e.g. icc, Cray) not attempted for some time

Overall optimisation level

- Limited difference in performance between O2 and O3
- Moving away from standard (i.e. IEEE-754 compliant) arithmetic possible **`-Ofast`**
  - Often not appropriate for HEP workflows e.g. cannot guarantee e.g. positive input - will remove checks for `sqrt(-1)`
  - **`-freciprocal-math`** is much more benign
  - Difficult to validate algorithm behaviour under small numerical changes

Link-time optimisation

- Allow compiler to reason across function units during library linking
- Potentially large benefits for larger codebases
- Problem: linker errors for the connoisseur

```
/tmp/smh/ccyEDIFM.ltrans0.ltrans.o:(.data.rel.ro+0x588):
    undefined reference to `typeinfo for ers::Issue'
collect2: error: ld returned 1 exit status
```

Profile-guided optimisation

- The compiler doesn't know where we're spending most of our time
- **`initialize()`**, **`execute()`**, **`finalize()`** all get the same level of attention
- Profile-guided optimisation (PGO)[2] takes output from a profiler and passes it to the compiler
  - Downside: same code running simulation, reconstruction, trigger - need different PGO runs (in principle - in practice?)
  - How often do you need to run the profile as you develop the code?
  - Evaluated for Geant4

[2] also known as feedback-directed optimisation (FDO)

Auto-vectorisation

- Only enabled at **O3** in GCC
- Can be fragile - no guarantee it will apply if you reorder a loop
- Better to write using vector intrinsics, but not part of most HEP physicists' skillset
- Ideally, use intrinsics someone else has written (e.g. Eigen, VecGeom)
- Need function multi-versioning to use AVX etc

The free lunch: preloading libraries

Drop-in replacements for glibc math.h

- Intel Math Function Library (drop-in replacement for `glibc math.h`) gives over 10% improvement in ATLAS reconstruction on Intel machines - still works on AMD but not guaranteed
- AMD has a similar product (AOCL) designed for EPYC

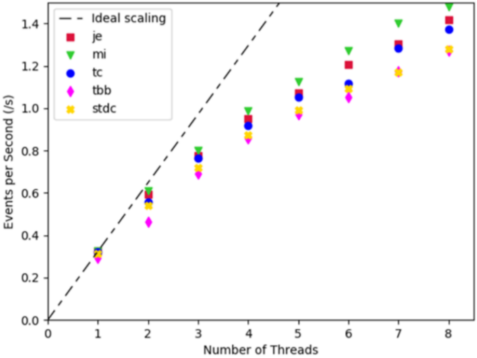
Note that there is no agreed standard for the output of trigonometric functions

- In practice, fairly close agreement
- ATLAS found differences with $\cos()$ when using IMF
- Least significant bit differs on machines with fused multiply-add instruction
- How to ensure numerical stability of HEP algorithms?

The free lunch: preloading libraries

Drop-in replacements for Linux default allocator (`glibc malloc`)

- When your program requests memory, allocator will parcel these up into larger requests
- Switching allocator can improve throughput and/or decrease total RAM footprint
- tcmalloc (Google, used by ATLAS)
- jemalloc (FreeBSD)
- mimalloc (Microsoft)



Conclusions

- Rich seam of easy and hard optimisations to apply to HEP code
- Use off-the-shelf tools as much as possible
- Needs revision as algorithms and frameworks change
- Opportunity to design new experiment software: fast and correct from the start

Time to learn Rust?