Modern C++ Programming

17. Code Optimization II

Federico Busato

University of Verona, Dept. of Computer Science

2020, v3.01

Table of Context

1 Compiler Optimizations

- About the Compiler
- Architecture Flags
- Optimization Flags
- Help the Compiler to Produce Better Code
- Profile Guided Optimization (PGO)

2 Libraries and Data Structures

- External Libraries
- `Std` Library

Table of Context

3 Profiling

- `gprof` 1/2
- `uftrace`
- `callgrind`
- `cachegrind`
- `perf` Linux profiler

4 Parallel Computing

- Concurrency vs. Parallelism
- Performance Scaling
- Gustafson's Law
- Parallel Programming Languages

Compiler Optimizations

About Compiler Optimizations

> *”I always say the purpose of optimizing compilers is not to make code run faster, but to prevent programmers from writing utter **** in the pursuit of making it run faster“*
>
> ***Rich Felker***, *`musl-libc` (`libc` alternative)*

Overview on compiler code generation and transformation:

- `Optimizations in C++ Compilers`
  Matt Godbolt, *ACM Queue*

About the Compiler 1/2

Important advise: **Use an updated version of the compiler**

- Newer compiler produces **better/faster code**
  - Effective optimizations
  - Support for newer CPU architectures
- **New warnings** to avoid common errors and better support for existing error/warnings (e.g. code highlights)
- **Faster compiling, less memory usage**
- **Less compiler bugs**: compilers are very complex and they have many bugs

Which compiler?

Answer: It dependents on the code and on the processor
example: `GCC 9 vs. Clang 8`

Some compilers can produce optimized code for specific architectures:

- **Intel Compiler** (commercial): Intel processors
- **IBM XL Compiler** (commercial): IBM processors/system
- **Nvidia PGI Compiler** (free/commercial): Multi-core processors/GPUs

- `gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`
- `Intel Blog: gcc-x86-performance-hints`

Architecture Flags

32-bits or 64-bits?

`-m64` In 64-bit mode the number of available registers increases from 6 to 14 general and from 8 to 16 `XMM`. Also all 64-bits x86 architectures have `SSE2` extension by default. 64-bit applications can use more than 4GB address space

`-m32` 32-bit mode. It should be combined with `-mfpmath=sse` to enable using of `XMM` registers in floating point instructions (instead of stack in x87 mode). 32-bit applications can use less than 4GB address space

It is recommended to use 64-bits for High-Performance Computing applications and 32-bits for phone and tablets applications

Optimization Flags 1/4

`-O0` Disables any optimization

- default behavior
- fast compile time

`-O1` Enables basic optimizations

`-O2` Enables advanced optimizations

- some optimization steps are expensive
- can increase the binary size

`-O3` Turns on all optimizations specified by -O2, plus some more

- `-O3` does not guarantee to produce faster code than `-O2`
- it could break floating-point IEEE764 rules on some non-traditional compilers

`-O4` For some compilers, it is an alias of `-O3`. In other cases can refers to inter-procedural optimization

Optimization Flags (floating-point) 2/4

In general, enabling the following flags implies less floating-point accuracy, breaking the IEEE764 standard, and it is implementation dependent (not included in `-O3`)

`-fno-trapping-math` Disable floating-point exceptions

`-ffinite-math-only` Disable special conditions for handling `inf` and `NaN`

`-funsafe-math-optimizations`

Allows breaking floating-point associativity and enables reciprocal optimization

`-ffast-math` Enables aggressive floating-point optimizations. All the previous, flush-to-zero denormal number, plus others

Optimization Flags 3/4

`-Ofast` Provides other aggressive optimizations that may violate strict compliance with language standards. It includes `-O3 -ffast-math`

`-Os` Optimize for size. It enables all `-O2` optimizations that do not typically increase code size

`-funroll-loops` Enables loop unrolling (not included in `-O3`)

`-fwhole-program` Assume that all non-extern functions and variables belong only to their compilation unit (see Link-time-optimization) (only recent compilers)

Optimization Flags 4/4

`-march=native` Generates instructions for a specific machine by determining the processor type at compilation time (not included in `-O3`) (e.g. `SSE2`, `AVX512`, etc.)

`-mtune=native` Generates instructions for a specific machine and for earlier CPUs in the architecture family (may be slower than `-march=native`)

`-flto` Enables *Link Time Optimizations* (Interprocedural Optimization). The linker merges all modules into a single combined module for optimization

- the linker must support this feature: `GNU ld v2.21++` or gold version, to check with `ld --version`
- it can significantly improve the performance
- in general, it is a very expensive step, even longer than the object compilations

Matrix Multiplication Example

`A * B`

N	128	256	512	1024
V0				
V1				
V2				
V3				
V4				
Speedup				

V0 `-O0`

V1 `-O3`

V2 `-O3` + restrit pointers

V3 `-O3 -march=native` + restrit pointers

V4 `-O3 -march=native -funroll-loops` + restrit pointers

Help the Compiler to Produce Better Code

Grouping related variables and functions in same translation units

- *Private* functions and variables in the same translation units
- Define every *global variable* in the translation unit in which it is used more often
- Declare in an *anonymous namespace* the variables and functions that are global to translation unit, but not used by other translation units
- Put in the same translation unit all the function definitions belonging to the same *bottleneck*

Static library linking helps the linker to optimize the code across different modules (link-time optimizations). Dynamic linking prevents these kind of optimizations

Profile Guided Optimization (PGO) 1/2

Profile Guided Optimization (PGO) is a compiler technique aims at improving the application performance by reducing instruction-cache problems, reducing branch mispredictions, etc. *PGO provides information to the compiler about areas of an application that are most frequently executed*

It consists in the following steps:

(1) Compile and *instrument* the code

(2) *Run* the program by exercising the most used/critical paths

(3) *Compile again* the code and exploit the information produced in the previous step

The particular options to instrument and compile the code are compiler specific

Profile Guided Optimization (PGO) 2/2

GCC

```
$ gcc -fprofile-generate my_prog.c my_prog # program instrumentation
$ ./my_prog # run the program (most critial/common path)
$ gcc -fprofile-use -O3 my_prog.c my_prog  # use instrumentation info
```

Clang

```
$ clang++ -fprofile-instr-generate my_prog.c my_prog
$ ./my_prog
$ xcrun llvm-profdata merge -output default.profdata default.profraw
$ clang++ -fprofile-instr-use=default.profdata -O3 my_prog.c my_prog
```

e.g. Firefox and Google Chrome support PGO building

Libraries and Data Structures

External Libraries 1/2

Consider using optimized *external* libraries for critical program operations

- `malloc` **replacement:**
  - `tcmalloc` (Google),
  - `mimalloc` (Microsoft)

- **Linear Algebra:** `Eigen`, `Armadillo`, `Blaze`

- **Map/Set:** `B+Tree` as replacement for red-black tree (`std::map`) (better locality, less pointers)
  - `STX B+Tree`
  - `Abseil B-Tree`

External Libraries 2/2

- **Hash Table:** (replace for `std::unsorted_set/map`)
  - `Google Sparse/Dense Hash Table`
  - `bytell hashmap`
  - `Facebook F14 memory efficient hash table`
  - `Abseil Hashmap` (2x-3x faster)

- **Print and formatting:** `fmt library` instead of `iostream` or `printf`

- **Random generator:** `PCG` random generator instead of `Mersenne Twister` or `Linear Congruent`

- **Non-cryptographic hash algorithm:** `xxHash` instead of `CRC`

- **Cryptographic hash algorithm:** `BLAKE3` instead of `MD5` or `SHA`

`std` Library 1/3

- Avoid old C library routines such as `qsort`, `bsearch`, etc. Prefer instead **`std::sort`**, **`std::binary_search`**
    - `std::sort` is based on a hybrid sorting algorithm. Quick-sort / head-sort (introsort), merge-sort / insertion, etc. depending on the `std` implementation

- **`std::fill`** applies `::memset` and **`std::copy`** applies `::memcpy` if the input/output are continuous in memory

- Set **`std::vector`** size during the object construction (or use the `reserve()` method) if the number of elements to insert is known in advance

- Prefer **`std::array`** instead of dynamic heap allocation

`std` Library 2/3

- Prefer `std::find()` for small array, `std::lower_bound`, `std::upper_bound`, `std::binary_search` for large sorted array
- Use `std` container member functions (e.g. `obj.find()`) instead of external ones (e.g. `std::find()`). Example: `std::set` $O(log(n))$ vs. $O(n)$
- Be aware of container properties, e.g. `vector.push_vector(v)`, instead of `vector.insert(vector.begin(), v)`
- Use `noexcept` decorator → program is aborted if an error occurred instead of raising an exception. see `Bitcoin: 9% less memory: make SaltedOutpointHasher noexcept`

`std` Library 3/3

- Consider *unordered* containers instead of the standard one, e.g. `unorder_map` vs. `map`
- Most data structures are implemented over the heap memory. Consider re-implement them by using the stack memory if the number of elements to insert is small (e.g. queue)
- Prefer **`lambda`** expression (or `function object`) instead of `std::function` or function pointer

Profiling

Overview

A **code profiler** is a form of *dynamic program analysis* which aims at investigating the program behavior to find performance bottleneck. A profiler is crucial in saving time and effort during the development and optimization process of an application

Code profilers are generally based on the following methodologies:

- **Instrumentation** Instrumenting profilers insert special code at the beginning and end of each routine to record when the routine starts and when it exits. With this information, the profiler aims to measure the actual time taken by the routine on each call.
  `Problem:` The timer calls take some time themselves
- **Sampling** The operating system interrupts the CPU at regular intervals (time slices) to execute process switches. At that point, a sampling profiler will record the currently-executed instruction

gprof

gprof is a profiling program which collects and arranges timing statistics on a given program. It uses a hybrid of instrumentation and sampling programs to monitor *function calls*

Website: `sourceware.org/binutils/docs/gprof/`

Usage:

- Code Instrumentation

```
$ g++ -pg [flags] <source_files>
```

Important: `-pg` is required also for linking and it is not supported by `clang`

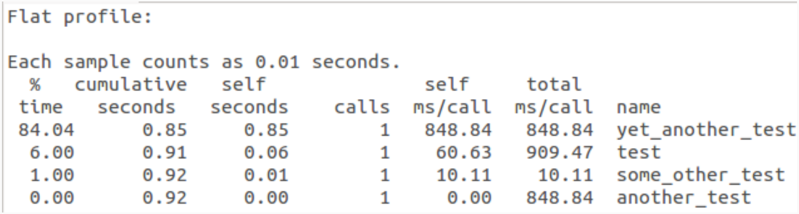
- Run the program (it produces the file `gmon.out`)
- Run `gprof` on `gmon.out`

```
$ gprof <executable> gmon.out
```

- Inspect `gprof` output

gprof 2/2

gprof output



```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 84.04      0.85     0.85        1   848.84   848.84  yet_another_test
  6.00      0.91     0.06        1    60.63   909.47  test
  1.00      0.92     0.01        1    10.11    10.11  some_other_test
  0.00      0.92     0.00        1     0.00   848.84  another_test
```

gprof can be also used for showing the call graph statistics

```
$ gprof -q <executable> gmon.out
```

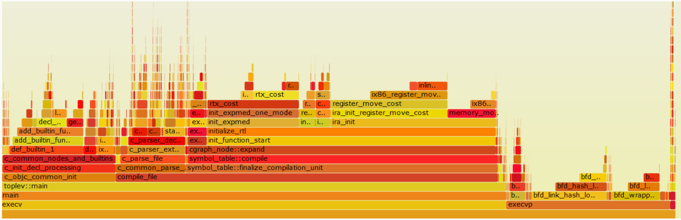
uftrace

The uftrace tool is to trace and analyze execution of a program written in C/C++

Website: github.com/namhyung/uftrace

```
$ gcc -pg <program>.cpp
$ uftrace record <executable>
$ uftrace replay
```

Flame graph output in html and svg



callgrind

callgrind is a profiling tool that records the call history among functions in a program's run as a call-graph. By default, the collected data consists of the number of instructions executed

Website: `valgrind.org/docs/manual/cl-manual.html`

Usage:

- Profile the application with `callgrind`

```
$ valgrind --tool callgrind <executable> <args>
```

- Inspect `callgrind.out.XXX` file, where `XXX` will be the process identifier

cachegrind

cachegrind simulates how your program interacts with a machine's cache hierarchy and (optionally) branch predictor

Website: `valgrind.org/docs/manual/cg-manual.html`

Usage:

- Profile the application with `cachegrind`

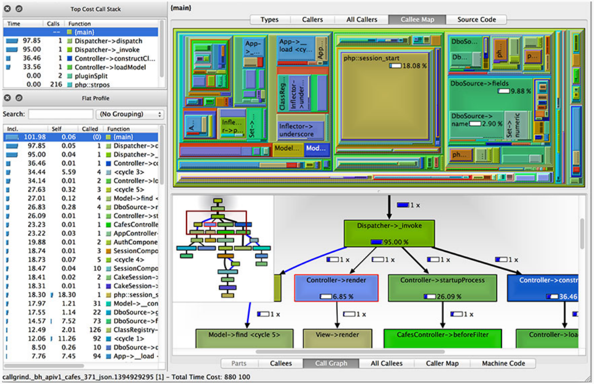
```
$ valgrind --tool cachegrind --branch-sim=yes <executable> <args>
```

- Inspect the output (cache misses and rate)
    - `l1` L1 instruction cache
    - `D1` L1 data cache
    - `LL` Last level cache

kcachegrind and qcachegrindwin (View)

KCachegrind (linux) and Qcachegrind (windows) provide a graphical interface for browsing the performance results of callgraph

- kcachegrind.sourceforge.net/html/Home.html
- sourceforge.net/projects/qcachegrindwin



Top Cost Call Stack

Time	Calls	Function
	--	{main}
97.85	1	Dispatcher->dispatch
95.00	1	Dispatcher->_invoke
36.46	1	Controller->constructCl...
33.56	1	Controller->loadModel
0.00	2	pluginSplit
0.00	216	php::strpos

Flat Profile

Search: (No Grouping)

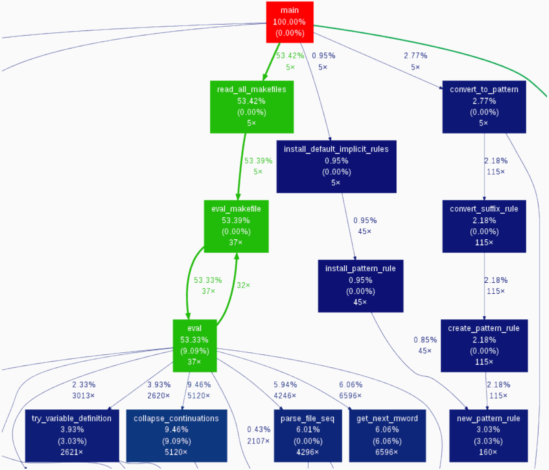
Incl.	Self	Called	Function
101.98	0.06	(0)	{main}
97.85	0.05	1	Dispatcher->c
95.00	0.04	1	Dispatcher->_
36.46	0.01	1	Controller->cc
34.44	5.59	4	<cycle 3>
34.14	0.01	2	Controller->lo
27.63	0.32	3	<cycle 5>
27.01	0.12	4	Model->find <
26.83	0.28	4	DboSource->r
26.09	0.01	1	Controller->st
23.23	0.01	1	CafesControlle
23.22	0.03	1	AppController-
19.88	0.01	2	AuthCompone
18.74	0.01	3	SessionCompo
18.73	0.07	5	<cycle 4>
18.47	0.04	10	SessionCompo
18.41	0.02	2	CakeSession->
18.31	0.01	1	CakeSession->
18.30	18.30	1	php::session_s
17.97	1.21	31	Model->__con
17.55	1.14	22	DboSource->g
14.57	7.52	73	DboSource->f
12.49	2.01	126	ClassRegistry-
12.06	11.26	92	<cycle 1>
8.50	0.26	10	DboSource->c
7.76	7.45	94	App->__load -

callgrind._bh_apiv1_cafes_371_json.1394929295 [1] – Total Time Cost: 880 100

gprof2dot (View)

gprof2dot is a Python script to convert the output from many profilers into a dot graph

Website: `github.com/jrfonseca/gprof2dot`

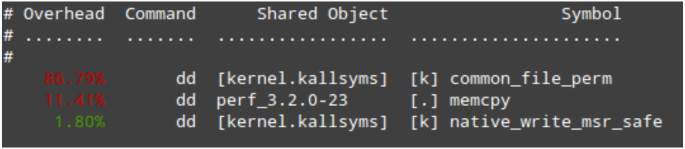


perf Linux profiler

Perf is performance monitoring and analysis tool for Linux. It uses statistical profiling, where it polls the program and sees what function is working

Website: perf.wiki.kernel.org/index.php/Main_Page

```
$ perf record -g <executable> <args> // or
$ perf record --call-graph dwarf <executable>
$ perf report // or
$ perf report -g graph --no-children
```



```
# Overhead  Command      Shared Object                Symbol
# ........  .......  .................  .....................
#
    86.79%       dd  [kernel.kallsyms]  [k] common_file_perm
    11.41%       dd  perf_3.2.0-23      [.] memcpy
     1.80%       dd  [kernel.kallsyms]  [k] native_write_msr_safe
```

Other Profilers

Free profiler:

- `Hotspot`

Proprietary profiler:

- `Intel VTune`
- `AMD CodeAnalyst`

Parallel Computing

Concurrency vs. Parallelism

Concurrency

A system is said to be **concurrent** if it can support two or more actions in progress at the same time. Multiple processing units work on different tasks independently

Parallelism

A system is said to be **parallel** if it can support two or more actions executing simultaneously. Multiple processing units work on the same problem and their interaction can effect the final result

Note: parallel computation requires rethinking original sequential algorithms (e.g. avoid race conditions)

Performance Scaling

Strong Scaling

The **strong scaling** defined how the compute time decreases increasing the number of processors for a fixed total problem size

Weak Scaling

The **weak scaling** defined how the compute time decrease increasing the number of processors for a fixed total problem size per processor

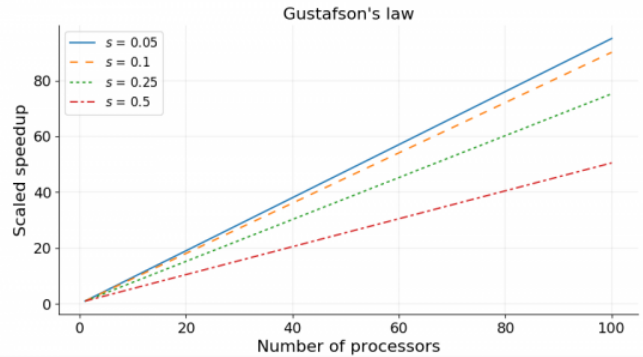
Strong scaling is hard to achieve because of computation units communication. *Strong scaling* is in contrast to the Amdahl's Law

Gustafson's Law

Gustafson's Law

Increasing number of processor units allow solving larger problems in the same time (the computation time is constant)

Multiple problem instances can run concurrently with more computational resources



Parallel Programming Platforms and APIs 1/3

C++11 Threads (+ Parallel STL) free, multi-core CPUs

OpenMP free, directive-based, multi-core CPUs and GPUs (last versions)

OpenACC free, directive-based, multi-core CPUs and GPUs

OpenCL free, multi-core CPUs and GPUs

Nvidia CUDA free, Nvidia GPUs

AMD ROCm free, AMD GPUs

HIP free, heterogeneous-compute Interface for AMD/Nvidia GPUs

Parallel Programming Platforms and APIs 2/3

SyCL free, abstraction layer for OpenCL, OpenMP, C/C++ libraries, multi-core CPUs and GPUs

KoKKos (Sandia) free, abstraction layer for multi-core CPUs and GPUs

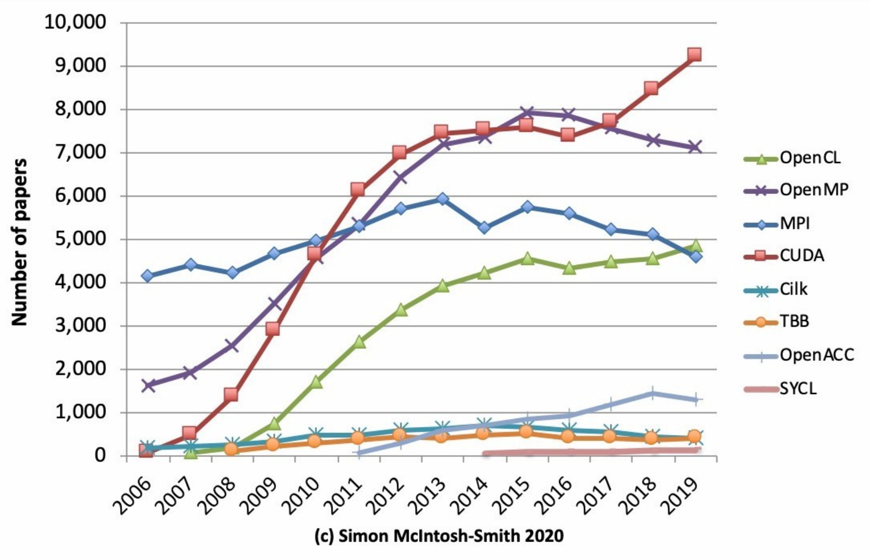
Raja (LLNL) free, abstraction layer for multi-core CPUs and GPUs

Intel TBB commercial, multi-core CPUs

Intel Cilk Plus commercial, multi-core CPUs

MPI free, de-facto standard for distributed system

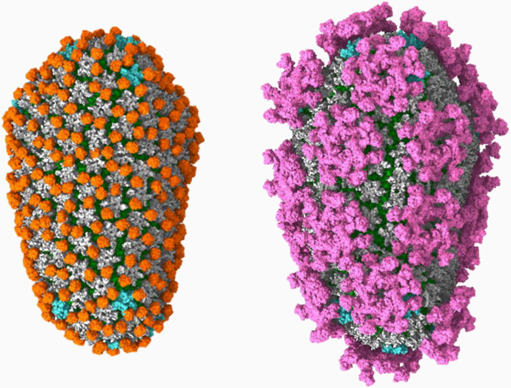
Parallel Programming Platforms and APIs 3/3



(c) Simon McIntosh-Smith 2020

A Nice Example

Accelerates computational chemistry simulations from 14 hours to 47 seconds with OpenACC on GPUs ($\sim 1,000x$ Speedup)



link: `Accelerating Prediction of Chemical Shift of Protein Structures on GPUs`