Modern C++ Programming

25. Performance Optimization III

Non-Coding Optimizations and Benchmarking

Federico Busato

2026-01-06

Table of Contents

1 Compiler Optimizations

- About the Compiler
- Compiler Optimization Flags
- Floating-point Optimization Flags
- Linker Optimization Flags
- Architecture Flags
- Help the Compiler to Produce Better Code
- Compiler Optimization Remarks
- Profile Guided Optimization (PGO and AutoFDO)
- Post-Processing Binary Optimizer
- Polyhedral Optimizations

Table of Contents

2 Compiler Transformation Techniques

- Basic Compiler Transformations
- Loop Unswitching
- Loop Fusion
- Loop Fission
- Loop Interchange
- Loop Tiling

Table of Contents

Table of Contents

5 Stable Performance Measurement

- Cache Behavior
- Stable CPU Performance
- Multi-Threads Considerations
- Operating System Considerations
- Program Memory Layout
- Measurement Overhead
- Compiler Optimizations

Table of Contents

6 Profiling

- `gprof`
- `uftrace`
- `callgrind`
- `cachegrind`
- `perf` Linux profiler

Table of Contents

7 Parallel Computing

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

Compiler Optimizations

About Compiler Optimizations 1/3

> *"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)

About Compiler Optimizations 2/3

```
bool isEven(int number) {
    int  numberCompare = 0;
    bool even          = true;
    while (number != numberCompare) {
        even = !even;
        numberCompare++;
    }
    return even;
}
```

→

```
bool isEven(int number) {
    return number & 1u;
}
```

About Compiler Optimizations 3/3

On the other hand, having a good compiler does not mean that it can fully optimize any code:

- The compiler does not *“understand”* the code, as opposed to human
- The compiler is *conservative* and applies optimizations only if they are safe and do not affect the correctness of computation
- The compiler is full of *models and heuristics* that could not match a specific situation
- The compiler *cannot spend large amount of time* in code optimization
- The compiler could consider *other targets* outside performance, e.g. binary size

About the Compiler 1/2

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

- Newer compiler produces **better/faster code**
  - Effective optimizations, for example, VS2026 provides %6 better performance
  - 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

Use an updated version of the linker: enable *Link Time Optimization*, `gold linker` or LLVM linker `lld`

About the Compiler 2/2

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 NVC++ Compiler** (free/commercial): Multi-core processors/GPUs

- gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html
- Intel Blog: gcc-x86-performance-hints
- Advanced Optimization and New Capa-bilities of GCC 10

Compiler Optimization Flags 1/3

`-O0`, `/Od` Disables any optimization

- default behavior
- fast compile time

`-O1`, `/O1` Enables basic optimizations

`-O2`, `/O2` Enables advanced optimizations

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

`-O3` Enable aggressive optimizations. 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 IEEE754 rules in some non-traditional compilers (`nvc++`, `IBM xlc`)

* Performance (Really) Matters, Emery Berger

Compiler Optimization Flags 2/3

`-O4` / `-O5` It is an alias of `-O3` in some compilers, or it can refer to `-O3` + inter-procedural optimizations (basic, full) and high-order transformation (HOT) optimizer for specialized loop transformations

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

`-Os` , `/Os` Optimize for size. It enables all `-O2` optimizations that do not typically increase code size (e.g. loop unrolling)

`-Oz` Aggressively optimize for size

Compiler Optimization Flags 3/3

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

`-fprefetch-loop-arrays` Emit prefetch instructions in loops (not included in `-O3`)

`-fopt-info` Describes optimization passes and missed optimizations
`-fopt-info-missed`

Floating-point Optimization Flags 1/2

In general, enabling the following flags implies less floating-point accuracy, breaking the IEEE754 standard, and it is implementation dependent (not included in `-O3`)

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

`-mfma -ffp-contract=fast` Force floating-point expression contraction such as forming of fused multiply-add operations

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

`-fassociative-math` Assume floating-point associative behavior

Floating-point Optimization Flags 2/2

`-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

Linker Optimization Flags

`-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

`-fwhole-program` Assume that the current compilation unit represents the whole program being compiled → Assume that all non-extern functions and variables belong only to their compilation unit

Ubuntu 21.04 To Turn On LTO Optimizations For Its Packages

Architecture Flags - 32-bits or 64-bits?

Architecture-oriented optimizations are not included in other flags (`-O3`).

`-m64` **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-bit mode for high-performance computing applications, while 32-bit mode for embedded platforms.

Architecture Flags 2/5

`-march=<arch>` Generates instructions for a specific processor to exploit exclusive hardware features, including SIMD instructions, pipelines depth, and L1/L2/L3 cache sizes. `<arch>` represents the minimum hardware supported by the binaries (not portable).

`-mtune=<tune_arch>` Specifies the target microarchitecture. Generates optimized code for a class of processors without exploiting specific hardware features. Binaries are still compatibles with other processors, for example, earlier CPUs in the architecture family. It may be slower than `-march`.

`-mcpu=<tune_arch>` Deprecated synonym for `-mtune` for `x86-64` processors, optimizes for both a particular architecture and microarchitecture on `Arm`.

Architecture Flags 2/5

`-mfpu<fp_hw>` (Arm) Optimize for a specific floating-point hardware.

`-m<instr_set>` (x86-64) Optimize for a specific instruction set.

Architecture examples:

`<arch>`	`armv9-a`, `armv7-a+neon-vfpv4`, `znver4`, `core2`, `skylake`
`<tune_arch>`	`cortex-a9`, `neoverse-n2`, `generic`, `intel`
`<instr_set>`	`see2`, `avx512`
`<fp_hw>`	`neon`, `neon-fp-armv8`

Architecture Flags

- `<tune_arch>` should be always greater than `<arch>`.
- In general, `-mtune` is set to `generic` if not specified.
- `-march=native`, `-mtune=native`, `-mcpu=native`: Allows the compiler to determine the processor type (not always accurate).
- Prefer compiler **automatic vectorization** over manual vector intrinsics, especially with new compilers.

- GCC Arm options, GCC X86-64 options
- Compiler flags across architectures: -march, -mtune, and -mcpu
- NVIDIA Grace CPU Benchmarking Guide, Arm Vector Instructions: SVE and NEON

Architecture Flags - Function Targets 5/5

GCC and Clang provide the attributes `target` and `target_clones` to automatically generate different instruction set backends that are dispatched at runtime

- `target` accepts different target options than specified on the command line. The original target command-line options are ignored
- `target_clones` accepts different targets in addition to the options specified on the command line

```
__attribute__ ((__target__ ("sse4.1,arch=core2")))
void f1() {}

__attribute__ ((__target_clones__ ("sse4.1,avx,default")))
void f2() {}
```

GCC documentation

Clang documentation

Help the Compiler to Produce Better Code

- Grouping variables and functions related to each other in the same translation unit
- Define *global variables* and *functions* in the translation unit in which they are used more often
- *Global variables* and functions that are not used by other translation units should have *internal linkage* (*anonymous namespace*/ `static` function)

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

Compiler Optimization Remarks 1/3

Recent compilers can provide insights, called **optimization remarks**, into the optimizations applied (or not applied) during the compilation process.

The reports help developers understand *how the compiler transforms code* and *identify areas where optimizations are not applied*. Such information is useful to improve performance, memory usage, and binary size.

Typical examples of **optimization passes** include:

- `licm` : Loop invariant code motion
- `loop-vectorize` : Loop vectorization optimization
- `size-info` : number of IR instructions
- `gvn` : Global value numbering
- `inline` : Function inlining
- `loop-unroll` : Loop unrolling

Compiler Optimization Remarks 2/3

GCC (`documentation`)

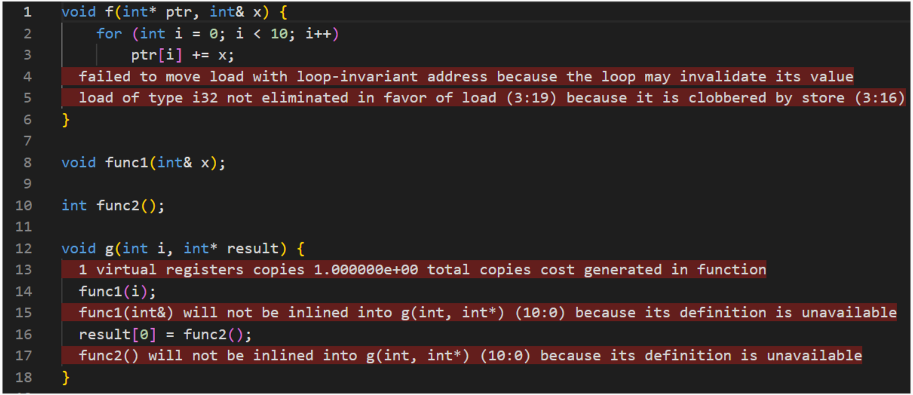
- `-fopt-info`, `-fopt-info-<filter>` Print applied and missed optimization passes. `<filter>` can be `missed`, `optimized`, `all`, `note` (verbose)
- `-fopt-info-<optimization>-<filter>` Print a specific optimization pass

Clang (`documentation`)

- `-Rpass`, `-Rpass=<optimization>` Print applied optimization passes
- `-Rpass-missed`, `-Rpass-missed=<optimization>` Print missed optimization passes
- `-Rpass-analysis`, `-Rpass-analysis=<optimization>` Print applied and missed optimization passes
- `-fsave-optimization-record` Save optimization reports. Then use `opt-viewer.py` (llvm suite) or `opt-viewer2` to generate a html file for a better visualization

Compiler Optimization Remarks 3/3

Optimization remarks are also available on Compiler Explorer



```
void f(int* ptr, int& x) {
    for (int i = 0; i < 10; i++)
        ptr[i] += x;
  failed to move load with loop-invariant address because the loop may invalidate its value
  load of type i32 not eliminated in favor of load (3:19) because it is clobbered by store (3:16)
}

void func1(int& x);

int func2();

void g(int i, int* result) {
  1 virtual registers copies 1.000000e+00 total copies cost generated in function
  func1(i);
  func1(int&) will not be inlined into g(int, int*) (10:0) because its definition is unavailable
  result[0] = func2();
  func2() will not be inlined into g(int, int*) (10:0) because its definition is unavailable
}
```

- Loop Optimizations: interpreting the compiler optimization report
- Generating compiler optimization remarks in LLVM
- Optimization Remarks - "Remarks Helping the Compiler Generate Better Code"

Profile Guided Optimization (PGO) 1/2

Profile Guided Optimization (PGO), also called **Feedback-Directed Optimization (FDO)**, 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
```

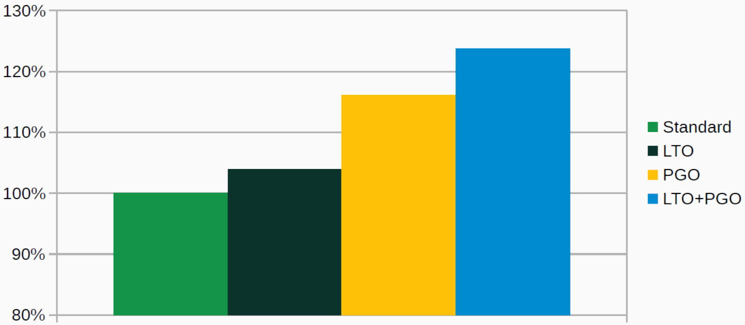
Clang

```
$ 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
```

Clang PGO can be combined with the optimization remark flag `-fshow-diagnostics-hotness` to show code chucks heavily optimized by the compiler (*hotness*)

Firefox and Google Chrome support PGO building

PGO and LTO Performance



SPEC 2017 built with GCC 10.2 and -O2

Automatic Feedback-Directed Optimization (AutoFDO)

Feedback-Directed Optimization (PGO/FDO) often shows high runtime overhead, require to recompile the binaries, and present difficulties in generating representative training data set

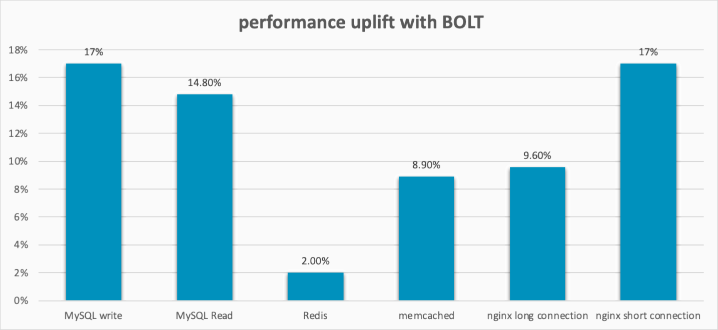
Automatic Feedback-Directed Optimization (AutoFDO) instead works by sampling hardware performance monitors on production machines and using those profiles to guide optimization

Some applications showed up to 10% performance improvements. Clang AutoFDO has been also used to optimize the Linux kernel resulting in 5%-10% performance speedup

- AutoFDO: Automatic Feedback-Directed Optimization for Warehouse-Scale Applications
- AutoFDO tutorial
- Clang AutoFDO & Propeller Optimization Support Sent In For Linux 6.13

Post-Processing Binary Optimizer 1/2

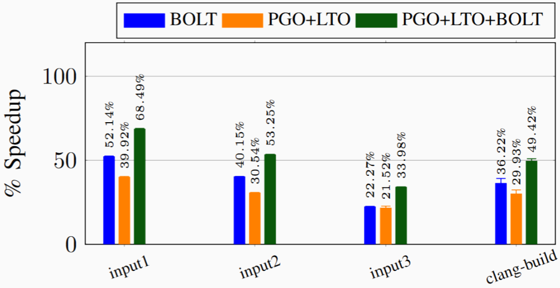
The code layout in the final binary can be further optimized with a **post-link binary optimizer** and **layout optimization** like `BOLT` or `Propeller` (sampling or instrumentation profile)

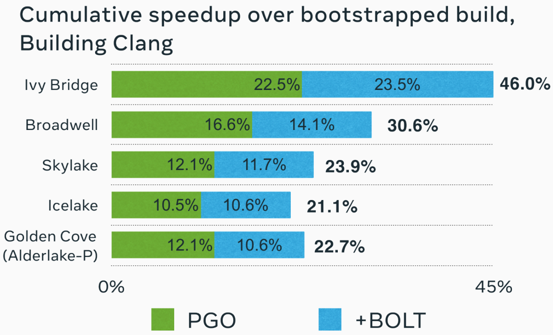


`BOLT: A Practical Binary Optimizer for Data Centers and Beyond`

`LLVM-project/BOLT`

Post-Processing Binary Optimizer

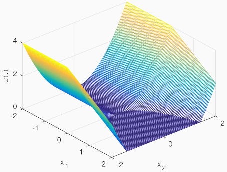



Cumulative speedup over bootstrapped build, Building Clang

2022 LLVM Dev Meeting: Optimizing Clang with BOLT using CMake

BOLT optimization technology could bring obvious performance uplift on arm server

Polyhedral Optimizations 1/2



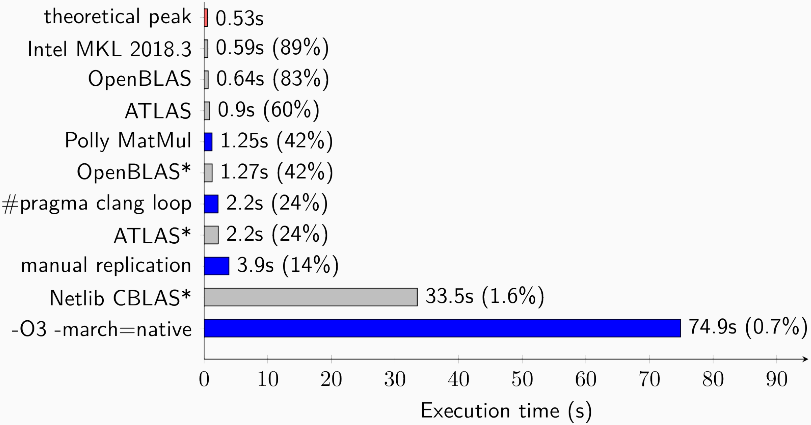
Polyhedral optimization is a compilation technique that rely on the representation of programs, especially those involving nested loops and arrays, in *parametric polyhedra*. Thanks to combinatorial and geometrical optimizations on these objects, the compiler is able to analyze and optimize the programs including *automatic parallelization*, *data locality*, *memory management*, *SIMD instructions*, and *code generation for hardware accelerators*

`Polly` ↗ is a high-level loop and data-locality optimizer and optimization infrastructure for LLVM

`PLUTO` ↗ is an automatic parallelization tool based on the polyhedral model

see also `Using Polly with Clang`

Polyhedral Optimizations 2/2



* Pre-compiled from Ubuntu repository

Compiler Transformation Techniques

Help the Compiler to Produce Better Code

Overview on compiler code generation and transformation:

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

Basic Compiler Transformations 1/3

- **Constant folding**. Direct evaluation constant expressions at compile-time

```
const int K = 100 * 1234 / 2;
```

- **Constant propagation**. Substituting the values of known constants in expressions at compile-time

```
const int K = 100 * 1234 / 2;
const int J = K * 25;
```

- **Common subexpression elimination**. Avoid computing identical and redundant expressions

```
int x = y * z + v;
int w = y * z + k; // y * z is redundant
```

Compiler Transformations 2/3

- **Induction variable elimination**. Eliminate variables whose values are dependent (induction)

```
for (int i = 0; i < 10; i++)
    x = i * 8;
// "x" can be derived by knowing the value of "i"
```

- **Dead code elimination**. Elimination of code which is executed but whose result is never used, e.g. dead store

```
int a = b * c;
... // "a" is never used, "b * c" is not computed
```

Unreachable code elimination instead involves removing code that is never executed

Compiler Transformations 3/3

- **Use-define chain**. Avoid computations related to a variable that happen before its definition

```
x = i * k + l;
x = 32; // "i * k + l" is not needed
```

- **Peephole optimization**. Replace a small set of low-level instructions with a faster sequence of instructions with better performance and the same semantic. The optimization can involve pattern matching

```
imul    eax, eax, 8 // a * 8
sal     eax, 3      // a << 3 (shift)
```

Loop Unswitching

- **Loop Unswitching**. Split the loop to improve data locality, reduce loop instructions (especially branches), and allow additional optimizations

```c
for (i = 0; i < N; i++) {
    if (x)
        a[i] = 0;
    else
        b[i] = 0;
```

```c
if (x) {
    for (i = 0; i < N; i++)
        a[i] = 0; // use memset
}
else {
    for (i = 0; i < N; i++)
        b[i] = 0; // use memset
}
```

Loop Fusion

- **Loop Fusion** (jamming). Merge multiple loops to improve data locality and perform additional optimizations

```
for (i = 0; i < 300; i++)
    a[i] = a[i] + sqrt(i);
for (i = 0; i < 300; i++)
for (i = 0; i < 300; i++) {
    auto tmp = sqrt(i);    // called once, we suppose sqrt is a pure function
    a[i]     = a[i] + tmp; // -> no side effects, no global state dependencies
    b[i]     = b[i] + tmp;
}
```

Loop Fission

- **Loop Fission** (distribution). Split a loop in multiple loops to

```
for (i = 0; i < size; i++) {
    a[i] = b[rand()]; // cache pollution
    c[i] = d[rand()];
for (i = 0; i < size; i++)
    a[i] = b[rand()]; // better cache utilization
for (i = 0; i < size; i++)
    c[i] = d[rand()];
```

Loop Interchange

- **Loop Interchange**. Exchange the order of loop iterations to improve data locality and perform additional optimizations (e.g. vectorization)

```
for (i = 0; i < 1000000; i++) {
    for (j = 0; j < 100; j++)
        a[j * x + i] = ...; // low locality
for (j = 0; j < 100; j++) {
    for (i = 0; i < 1000000; i++)
        a[j * x + i] = ...; // high locality
}
```

Loop Tiling

- **Loop Tiling** (blocking, nest optimization). Partition the iterations of multiple loops to exploit data locality

```
for (i = 0; i < N; i++) {
    for (j = 0; j < M; j++)
        a[j * N + i] = ...; // low locality
for (i = 0; i < N; i += TILE_SIZE) {
    for (j = 0; j < M; j += TILE_SIZE) {
        for (k = 0; k < TILE_SIZE; k++) {
            for (l = 0; l < TILE_SIZE; l++) {
```

Libraries and Data Structures

External Libraries and Data Structures 1/4

Consider using optimized *external* libraries for critical program operations

Compressed Bitmask: set algebraic operations

- `BitMagic Library`
- `Roaring Bitmaps`

Ordered Map/Set: `B+Tree` as replacement for red-black tree

- `STX B+Tree`
- `Abseil B-Tree`

External Libraries and Data Structures 2/4

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)
- Robin Hood Hashing
- Comprehensive C++ Hashmap Benchmarks 2022
- An Extensive Benchmark of C and C++ Hash Tables

External Libraries and Data Structures 3/4

- **Probabilistic Set Query:** Bloom filter, ‘`XOR filter`, `Facebook’s Ribbon Filter`, `Binary Fuse filter`

- **Scan, print, and formatting:** `fmt library`, `scn library` instead of `iostream` or `printf/scanf`

- **Random generator:** `PCG`/`Xoshiro`/`DualMix128` random generators instead of `Mersenne Twister` or `Linear Congruent`

- **Integer hash function** instead of a random generator if the period length is not a concern `hash-prospector`

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

- **Cryptographic hash algorithm:** `BLAKE3` instead of `MD5` or `SHA256`

External Libraries and Data Structures 4/4

- **Search:** Performance comparison: linear search vs binary search

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

- **Sort:**
  - Beating Up on Qsort. Radix-sort for non-comparative elements (e.g. `int`, `float`)
  - Vectorized and performance-portable Quicksort

- **Compression:** LZ4 (very fast/medium compression), zstd (fast/ high compression) instead of zlib or lzma

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

Libraries and `Std` replacements

- **`Folly`**: Performance-oriented `std` library (Facebook)
- **`Abseil`**: Open source collection of C++ libraries drawn from the most fundamental pieces of Google's internal codebase
- **`Frozen`**: Zero-cost initialization for immutable containers, fixed-size containers, and various algorithms.

awesome HPP

A curated list of awesome header-only C++ libraries

Performance Benchmarking

Performance Benchmarking

Performance benchmarking is a non-functional test focused on measuring the efficiency of a given task or program under a particular load

Performance benchmarking is hard!!

Main reasons:

- What to test?
- Workload/Dataset quality
- Cache behavior
- Stable CPU performance
- Program memory layout
- Measurement overhead
- Compiler optimizations
- Metric evaluation

What to Test?

1. **Identify performance metrics**: The metric(s) should be strongly related to the specific problem and that allows a comparison across different systems, e.g. elapsed time is not a good metric in general for measuring the throughput
   - Matrix multiplication: FLoating-point Operation Per Second (FLOP/s)
   - Graph traversing: Edge per Second (EP/s)

2. **Plan performance tests**: Determine what part of the problem is relevant for solving the given problem, e.g. excluding initialization process
   - Suppose a routine that requires different steps and ask a memory buffer for each of them. Memory allocations should be excluded as a user could use a memory pool

Workload/Dataset Quality

1. **Stress the most important cases**: Rare or edge cases that are not used in real-world applications or far from common usage are less important, e.g. a graph problem where all vertices are not connected

2. **Use datasets that are well-known in the literature and reproducible**. Don't use "self-made" dataset and, if possible, use public available resources

3. **Use a reproducible test methodology**. Trying to remove sources of "noise", e.g. if the procedure is randomized, the test should be use with the same seed. It is not always possible, e.g. OS scheduler, atomic operations in parallel computing, etc.

see also `Reproducibility in artificial intelligence`

Metric Evaluation 1/6

After extracting and collecting performance results, it is fundamental to report/summarize them in a way to fully understand the experiment, provide interpretable insights, ensure reliability, and compare different observations, e.g. codes, algorithms, systems, etc.

Metric Evaluation

Metric	Formula	Description
Arithmetic mean	$\bar{x} = \frac{\sum_{i=1}^{n} x_i}{n}$	For summarizing costs, e.g. exec. times, floating point ops, etc.
Harmonic mean	$\frac{n}{\sum_{i=1}^{n} \frac{1}{x_i}}$	For summarizing rates, e.g. flop/s
Geometric mean	$\sqrt[n]{\prod_{i=1}^{n} x_i}$	For summarizing rates. Harmonic mean should be preferred. Commonly used for comparing speedup
Standard deviation	$\sigma = \frac{\sum_{i=1}^{n} (x_i - \overline{x})^2}{n-1}$	Measure of the spread of normally distributed samples
Coefficient of Variation	$\frac{std.dev}{arith.mean}$	Represents the stability of a set of normally distributed measurement results. Normalized standard deviation

Metric Evaluation

Metric	Formula	Description
Confidence intervals of the mean	$z = t\left(n-1, \frac{\alpha}{2}\right)$ $CI = \left[\bar{x} - \frac{z\sigma}{\sqrt{n}}, \bar{x} + \frac{z\sigma}{\sqrt{n}}\right]$	Measure of reliability of the experiment. The concept is interpreted as the probability (e.g. $\alpha = 95\%$) that the observed confidential interval contains the true mean
Median	value at position $n/2$ after sorting all data	Rank measures are more robust with regard to outliers but do not consider all measured values
Quantile: Percentile/Quartile	value at a given position after sorting all data	The percentiles/quartiles provide information about the spread of the data and the skew. It indicates the value below which a given percentage of data falls
Minumum/ Maximum	$\min / \max_{i=1}^{n}(x_i)$	Provide the lower/upper bounds of the data, namely the range of the values

Metric Evaluation

Confidence Interval	Z
80%	1.282
85%	1.440
90%	1.645
95%	1.960
99%	2.576
99.5%	2.807
99.9%	3.291

Some metrics assume a normal distribution → the arithmetic mean, median and mode are all equal

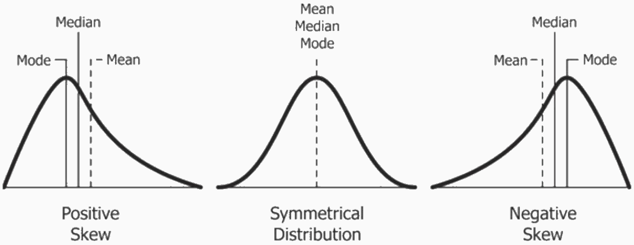
$$\frac{|\bar{x} - median|}{\max\left(\bar{x}, median\right)}$$

If the *relative difference between the mean and median* is larger than 1%, values are probably not normally distributed

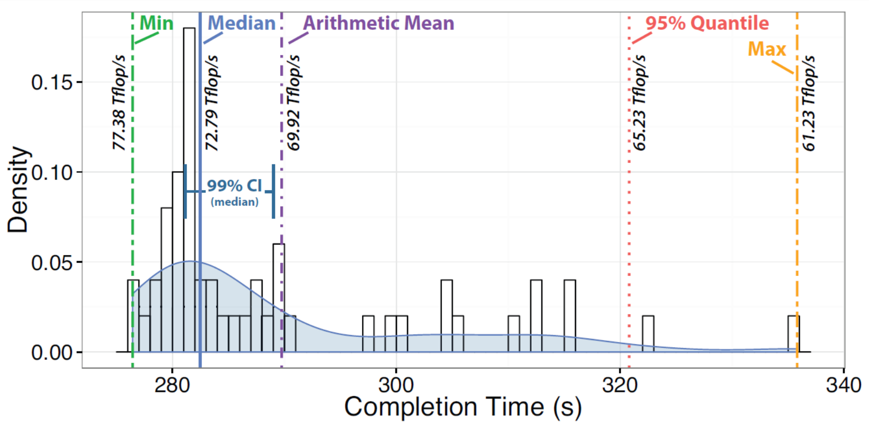
Metric Evaluation

Minimum/Maximum vs. Arithmetic mean. The minimum/maximum could be used to get the best outcome of an experiment, namely the measure with the least noise. On the other hand, the arithmetic mean considers all values and could better represent the behavior of the experiment.

If the *skewness* of the distribution is *symmetrical* (e.g. normal, binomial) then the arithmetic mean is a superior statistic, while the minimum/maximum could be useful in the opposite case (e.g. log-normal distribution)



Metric Evaluation



- Benchmarking: minimum vs average
- Scientific Benchmarking of Parallel Computing Systems
- Benchmarking C++ Code

Stable Performance Measurement

Cache Behavior 1/2

- *Cache behavior is not deterministic*. Different executions lead to different hit rates
- After a data is loaded from the main memory, it remains in the cache until it expires or is evicted to make room for new content
- Executing the same routine multiple times, the first run is much slower than the other ones due to the cache effect (warmup run)

Cache Behavior

There is no a systematic way to flush the cache. Some techniques to ensure more reliable performance results are

- overwrite all data involved in the computation between each runs
- read/write between two buffers of size at least the size of the largest cache
- some processors, such as ARM, provide specific instructions to *invalidate* the cache `__builtin___clear_cache()`, `__clear_cache()`

Note: manual cache invalidation must consider cache locality (e.g. L1 per CPU core) and compiler optimizations that can remove useless code (solution: use global variables and `volatile`)

see: Is there a way to flush the entire CPU cache related to a program?

Stable CPU Performance 1/4

One of the first source of fluctuation in performance measurement is due to unstable CPU frequency

Dynamic frequency scaling, also known as *CPU throttling*, automatically decreases the CPU frequency for:

- Power saving, extending battery life
- Decrease fan noise and chip heat
- Prevent high frequency damage

Modern processors also comprise advanced technologies to automatically **raise CPU operating frequency when demanding tasks are running** (e.g. Intel® Turbo Boost). Such technologies allow processors to run with the *highest possible frequency* for limited amount of time depending on different factors like *type of workload*, *number of active cores*, *power consumption*, *temperature*, etc.

Stable CPU Performance 2/4

Get CPU info:

- *CPU characteristics*:

  ```
  lscpu
  ```

- *Monitor CPU clocks in real-time*:

  ```
  cpupower monitor -m Mperf
  ```

- *Get CPU clocks info*:

  ```
  cpupower frequency-info
  ```

  see “`cpufreq governors`”

Stable CPU Performance 3/4

- *Disable Turbo Boost*

  ```
  echo 1 » /sys/devices/system/cpu/intel_pstate/no_turbo
  ```

- *Disable hyper threading*

  ```
  echo 0 > /sys/devices/system/cpu/cpuX/online
  ```

  or through BIOS

- *Use* “`performance`” *scaling governor* and max frequency and use ‘*userspace*‘ governor to specify a fixed frequency

  `sudo cpupower frequency-set -g performance` or

  `sudo cpufreq-set -f <frequency>`, e.g. 3200000 (3.2 GHz)

Stable CPU Performance 4/4

- *Use ‘userspace‘ governor* to specify a fixed frequency

  ```
  sudo cpufreq-set -g userspace
  ```

  It is recommended to use 80% of the maximum frequency to reduce thermal throttling and dynamic frequency scaling.

  `sudo cpufreq-set -u <frequency>`, e.g. 3200000 (3.2 GHz)

- *Set CPU affinity*: CPU (set of cores) ↔ Program binding

  ```
  taskset -c <cpu_id> <program>
  ```

- *Set process priority* in the range $[-20, 19]$ (highest-lowest)

  ```
  sudo nice -n -15 <process>
  ```

Multi-Threads Considerations

- `numactl -interleave=all`
  NUMA: Non-Uniform Memory Access (e.g. multi-socket system)
  The default behavior is to allocate memory in the same node as a thread is scheduled to run on, and this works well for small amounts of memory. However, when you want to allocate more than a single node memory, it is no longer possible. This option sets interleaved memory allocations among NUMA nodes

- `export OMP_NUM_THREADS=96` Set the number of threads in an OpenMP program

Operating System Considerations

- *Disable unnecessary system services*

  List all enabled services:

  ```
  systemctl list-unit-files --type=service --state=enabled
  ```

  Stop a specifi service: `sudo systemctl stop <service_name>`

- *Disable address space layout randomization (ASLR)*

  ```
  echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
  ```

- *Drop file system cache* (if the benchmark involves IO ops)

  ```
  echo 3 | sudo tee /proc/sys/vm/drop_caches; sync
  ```

- *CPU isolation*

  don't schedule process and don't run kernels code on the selected CPUs. GRUB options: `isolcpus=<cpu_ids>,rcu_nocbs=<cpu_ids>`

References

- NVIDIA Grace Performance Tuning Guide
- How to get consistent results when benchmarking on Linux?
- How to run stable benchmarks
- Best Practices When Benchmarking CUDA Applications

Program Memory Layout 1/2

A small code change modifies the memory program layout
→ large impact on cache (up to 40%)

- **Linking**
  - link order → changes function addresses
  - upgrade a library
- **Environment Variable Size**: moves the program stack
  - run in a new directory
  - change username

▪ `Performance Matters`, *E. Berger*, CppCon20

▪ `Producing Wrong Data Without Doing Anything Obviously Wrong!`, *Mytkowicz et al.*, ASPLOS'09

Program Memory Layout 2/2

The compiler provides options that can be used to align functions, jumps, labels, and loops, as well as mitigate program memory layout issues. The flags are already enabled with their default values with `-O2`.

`-falign-functions=<N>` Align the start of functions to `N` bytes.

`-falign-jumps=<N>` Align branch targets reachable with jump instructions to `N` bytes.

`-falign-loops=<N>` Align loops to `N` bytes.

`-falign-labels=<N>` Align all branch targets, including jumps and loops, to `N` bytes.

Measurement Overhead

Time-measuring functions could introduce significant overhead for small computation

`std::chrono::high_resolution_clock::now()` / `std::chrono::system_clock::now()` rely on library/OS-provided functions to retrieve timestamps (e.g. `clock_gettime`) and their execution can take several clock cycles

Consider using a **benchmarking framework**, such as `Google Benchmark` or `nanobench` (`std::chrono` based), to retrieve hardware counters and get basic profiling info

Compiler Optimizations

Compiler optimizations could distort the actual benchmark

- *Dead code elimination:* the compiler discards code that does not perform “useful” computation
- *Constant propagation/Loop optimization:* the compiler is able to pre-compute the result of simple codes
- *Instruction order:* the compiler can even move the time-measuring functions

Other Considerations

The actual numerical values for a benchmark could significantly affect the results. For instance, a dense matrix multiplication (GEMM) could show 2X performance between matrices filled with zeros and random values due to the effect on power consumption.

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

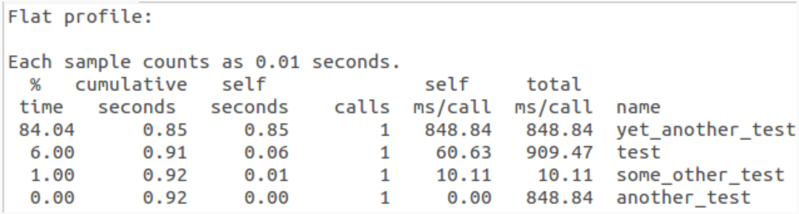
  ```
  $ g++ -pg [flags] <source_files>
  ```

  Important: `-pg` is also required 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
```

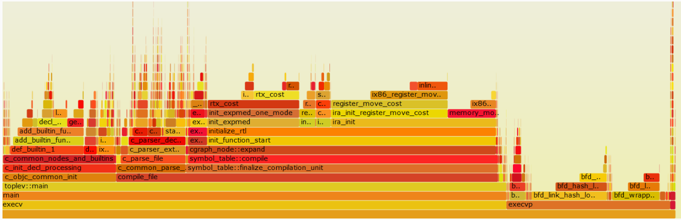
uftrace

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

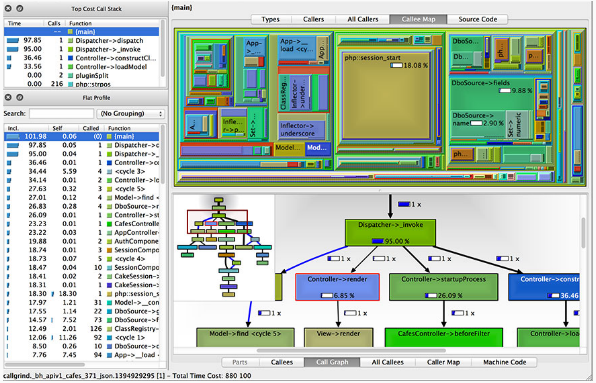
```
$ valgrind --tool cachegrind --branch-sim=yes <executable> <args>
```

- 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)

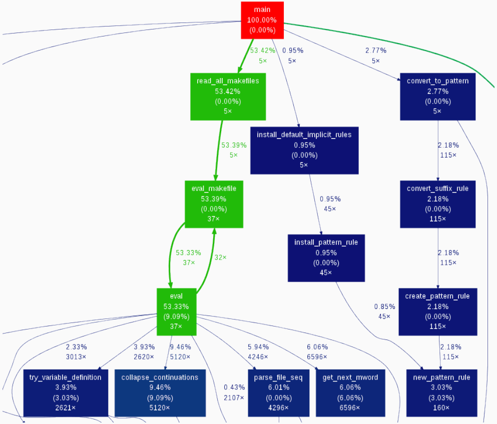
Incl.	Self	Called	Function
101.98	0.06	(0)	{main}
97.85	0.05	1	Dispatcher->c
95.00	0.04	1	Dispatcher->_
36.46	0.01	1	Controller->cc
34.44	5.59	4	<cycle 3>
34.14	0.01	2	Controller->lc
27.63	0.32	3	<cycle 5>
27.01	0.12	4	Model->find <
26.83	0.28	4	DboSource->r
26.09	0.01	1	Controller->st
23.23	0.01	1	CafesControlle
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`

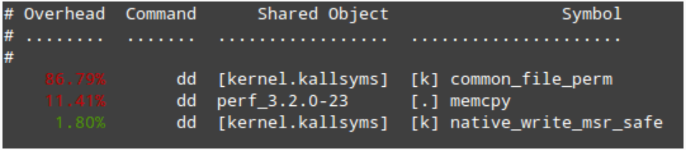


perf Linux profiler 1/2

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

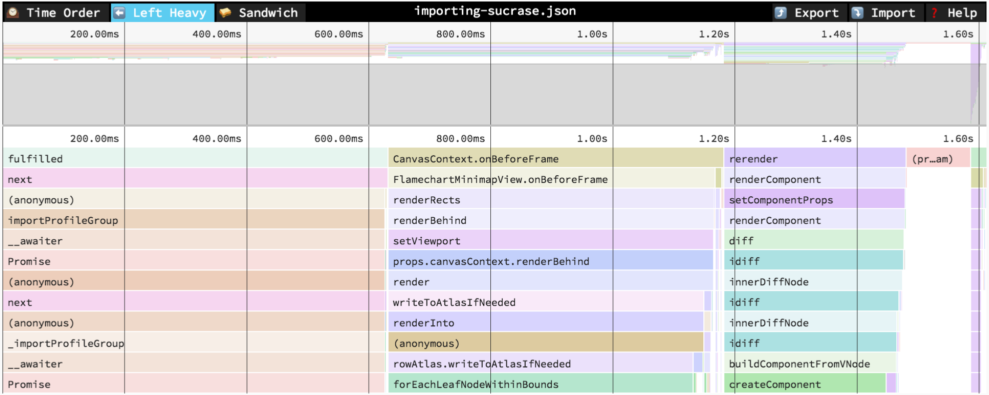
```
$ perf record -g <executable> <args> // or
$ perf record --call-graph dwarf <executable>
$ perf report // or
$ perf report -g graph --no-children
```



```
# 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
```

Data collected by **perf** can be visualized by using flame graphs, see:

`Speedscope: visualize what your program is doing and where it is spending time`



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

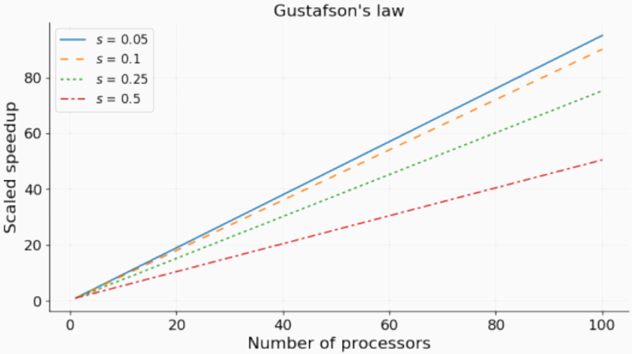
Strong scaling is hard to achieve because of computation units communication. *Strong scaling* is in contrast to the Amdahl's Law

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
Khronos OpenCL	free, multi-core CPUs, GPUs, FPGA
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

Khronos 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

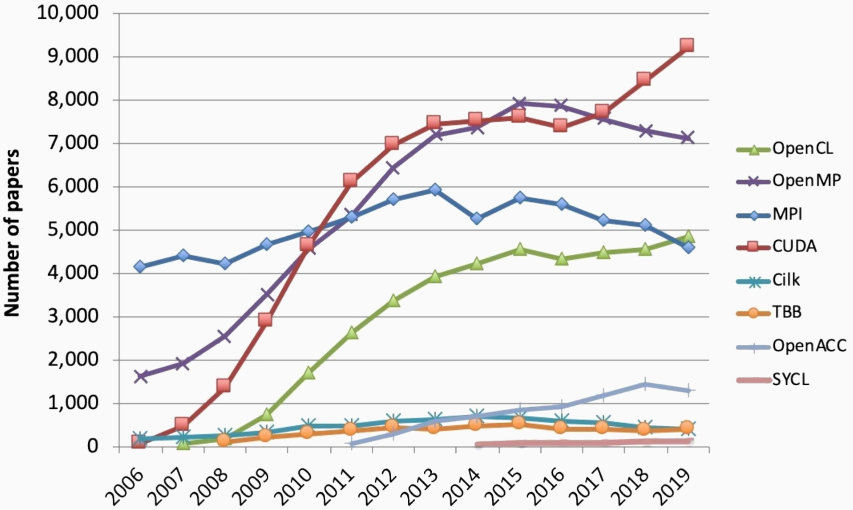
Raja (LLNL) free, abstraction layer for multi-core CPUs and GPUs

Intel TBB commercial, multi-core CPUs

OneAPI free, Data Parallel C++ (DPC++) built upon C++ and SYCL, CPUs, GPUs, FPGA, accelerators

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)