Modern C++ Programming

17. Code Optimization II

Federico Busato

University of Verona, Dept. of Computer Science
2021, v3.04

Table of Context

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)*

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

`-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) 2/5

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

`-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`)

Optimization Flags

`-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

`-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

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
```

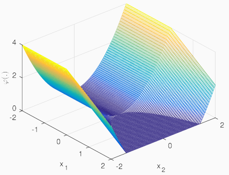
Clang

```
$ 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

Polyhedral Optimizations

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`

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`

- **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 y = y * z + k; // y * z is redundant
```

- **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"
```

- **Dense 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

- **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 and perform additional optimizations

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

```
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++)
    b[i] = b[i] + sqrt(i);
```

```
for (i = 0; i < 300; i++) {
    a[i] = a[i] + sqrt(i); // sqrt(i) is computed only
    b[i] = b[i] + sqrt(i); // one time
}
```

Loop Fission

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

```
for (i = 0; i < 300; i++)
    a[i] = a[i] + sqrt(i);
for (i = 0; i < 300; i++)
    b[i] = b[i] + sqrt(i);
```

```
for (i = 0; i < 300; i++) {
    a[i] = a[i] + sqrt(i); // sqrt(i) is computed only
    b[i] = b[i] + sqrt(i); // one time
}
```

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++) {
```

In many cases, the compiler already applies these optimizations

Libraries and Data Structures

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`

- **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` awesome HPP

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

`std` Library - From C to C++

- 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
  - Prefer `std::find()` for small array, `std::lower_bound`, `std::upper_bound`, `std::binary_search` for large sorted array

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

- Prefer **`lambda`** expression (or `function object`) instead of `std::function` or function pointers

`std` Library - Containers

- 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(), value)`
- Consider *unordered* containers instead of the standard one, e.g. `unorder_map` vs. `map`
- Prefer **`std::array`** instead of dynamic heap allocation
- 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)

- Set `std::vector` size during the object construction (or use the `reserve()` method) if the number of elements to insert is known in advance
- Use the same type for initialization in functions like `std::accumulate()`

```
auto array = new int[size];
...                              // 0u != 0 => no memset
auto sum = std::accumulate(array, array + size, 0u);
```

- Use `noexcept` decorator → program is aborted if an error occurred instead of raising an exception. see `Bitcoin: 9% less memory: make SaltedOutpointHasher noexcept`

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 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
```

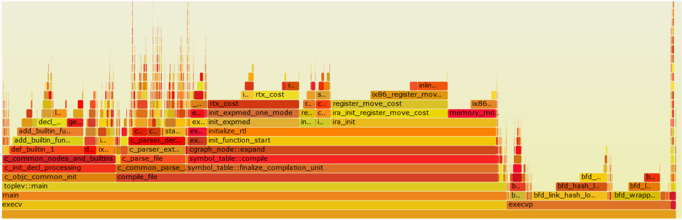
uftrace

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`

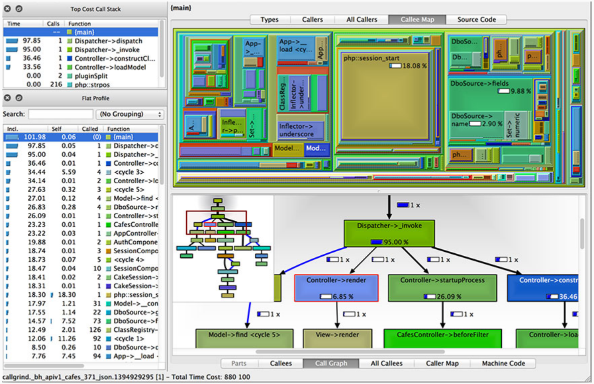
```
$ valgrind --tool cachegrind --branch-sim=yes <executable> <args>
```

- 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`

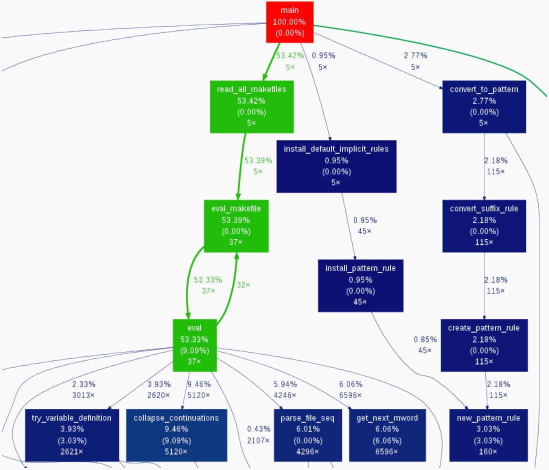
- `kcachegrind.sourceforge.net/html/Home.html`
- `sourceforge.net/projects/qcachegrindwin`



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
```

Linux perf for Qt developers

Other Profilers

Free profiler:

- `Hotspot`

Proprietary profiler:

- `Intel VTune`
- `AMD CodeAnalyst`

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

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 (FLOPS)
    - Graph traversing: Edge per Second (EPS)

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, 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`

Cache Behavior

- 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
- *There is no a systematic way to flush the cache*. A good technique to ensure reliable performance results is to overwrite all data involved in the computation between each runs

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

Stable CPU Performance

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.

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`"

- *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*

```
sudo cpupower frequency-set -g performance
```

- *Set CPU affinity* (CPU-Program binding)

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

- *Set process priority*

```
sudo nice -n -5 taskset -c <cpu_id> <process>
```

- *Disable address space randomization*

```
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>
```

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

Program Memory Layout

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

- **Linking**
  - link order $\rightarrow$ 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

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

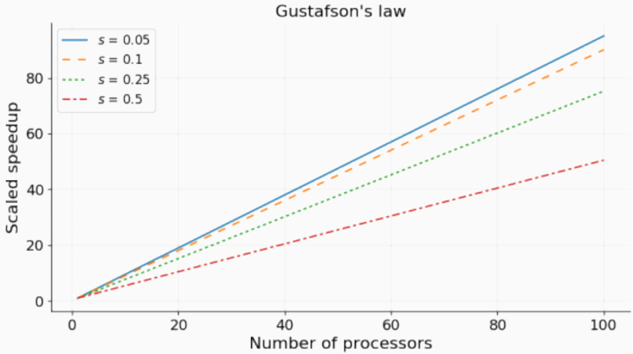
Strong scaling is hard to achieve because of computation units communication. *Strong scaling* is in contrast to the Amdahl's Law

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

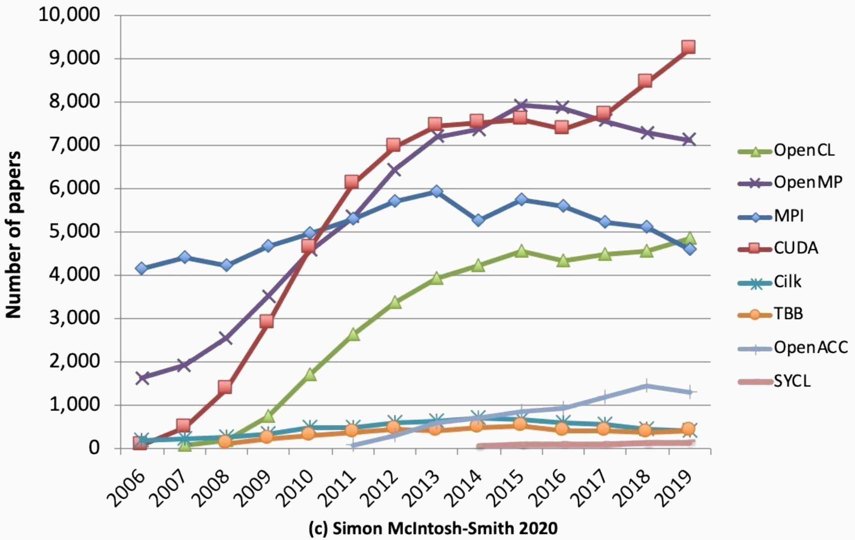
KoKKos (Sandia) free, abstraction layer for multi-core CPUs and GPUs

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

Intel TBB commercial, multi-core CPUs

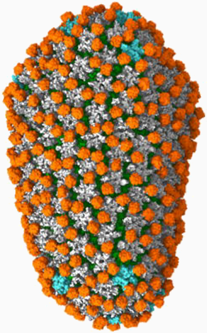
OneAPI free, Data Parallel C++ (DPC++) built upon C++ and SYCL, CPUs, GPUs, FPGA, accelerators

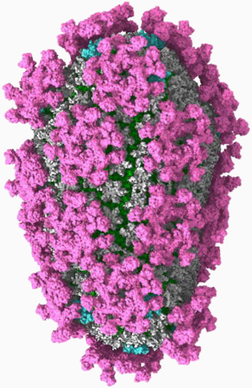
MPI free, de-facto standard for distributed system


10,000
9,000
8,000
7,000
6,000
5,000
4,000
3,000
2,000
1,000
0
Number of papers
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
OpenCL
OpenMP
MPI
CUDA
Cilk
TBB
OpenACC
SYCL
(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`