Modern C++ Programming

15. Code Optimization

Federico Busato

University of Verona, Dept. of Computer Science
2019, v2.01

Agenda

(1) General Concepts

Optimization Cycle

"If you're not writing a program, don't use a programming language"

Leslie Lamport, *Turing Award*

"Inside every large program is an algorithm trying to get out"

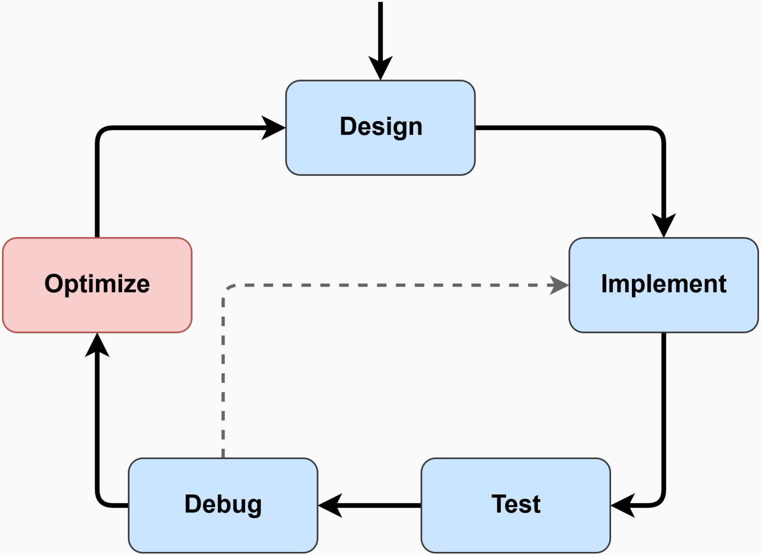
Tony Hoare, *Turing Award*

"Premature optimization is the root of all evil"

Donald Knuth, *Turing Award*

"Code for correctness first, then optimize!"

Optimization Cycle



The **asymptotic analysis** refers to estimate the execution time or memory usage as function of the input size (the *order of growing*)

The *asymptotic behavior* is opposed to a *low-level analysis* of the code (instruction/loop counting/weighting, cache accesses, etc.)

Drawbacks:

- The *worst-case* is not the *average-case*
- Asymptotic complexity does not consider small inputs
- The hidden constant can be relevant in practice
- Asymptotic complexity does not consider instructions cost and hardware details

One example out of them all is the *Strassen*'s matrix multiplication algorithm... but

`arXiv:1808.07984`: *Implementing Strassen's Algorithm with CUTLASS on NVIDIA Volta GPUs, J. Huang et. al*

Be aware of only **real-world problems** with small asymptotic complexity or small size can be solved in a *"user" acceptable time*

Two examples:

- *Sorting*: $\mathcal{O}(n \log n)$, try to sort an array of one billion elements (4GB)
- *Diameter of a (sparse) graph*: $\mathcal{O}(V^2)$, just for graphs with a few hundred thousand vertices it becomes impractical without advanced techniques

Ahmdal Law 1/2

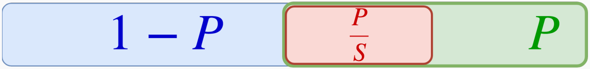
Ahmdal Law

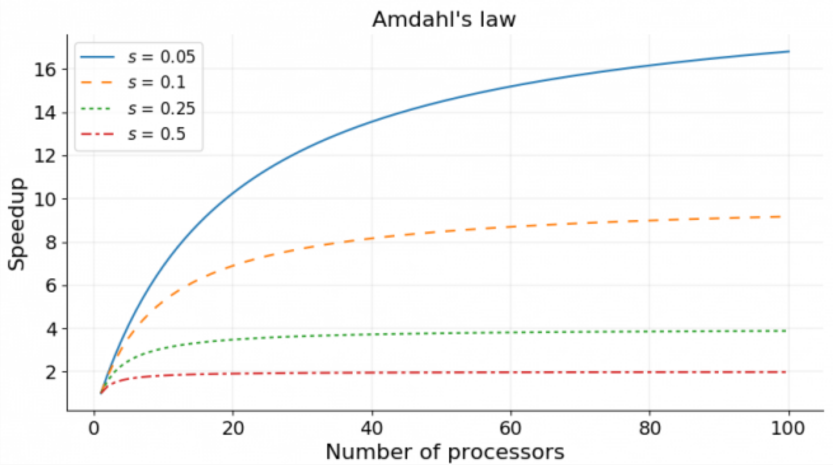
The **Ahmdal law** expresses the maximum improvement possible by improving a particular part of a system

Observation: The performance of any system is constrained by the speed or capacity of the slowest point

$$Improvement\,(S) = \frac{1}{(1 - P) + \frac{P}{S}}$$

P : portion of the system that can be improved
S : improvement factor





note: **s** is the portion of the system that cannot be improved

Performance Bounds

The performance of a program is *bounded* by one or more aspects of its computation. This is also strictly related to the underlying hardware

- **Memory-bound**. The program spends its time primarily in performing *memory accesses*. The progress is limited by the *memory bandwidth* (sometime memory-bound also refers to the amount of memory available)
- **Compute-bound**. The program spends its time primarily in computing *arithmetic instructions*. The progress is limited by the *speed of the CPU*

- **Latency-bound**. The program spends its time primarily in waiting *the data are ready* (instruction/memory dependencies). The progress is limited by the *latency of the CPU/memory*

- **I/O Bound**. The program spends its time primarily in performing *I/O operations* (network, user input, storage, etc.). The progress is limited by the *speed of the I/O subsystem*

Arithmetic Intensity

Arithmetic intensity is the ratio of total operations to total data movement (bytes)

The naive matrix multiplication algorithm requires $n^3 \cdot 2$ floating-point operations (multiplication + addition), while it involves $\left(n^2 \cdot 4\text{B}\right) \cdot 3$ data movement in bytes

$$R = \frac{ops}{bytes} = \frac{2n^3}{12n^2} = \frac{n}{6}$$

which means that for every byte accessed, the algorithm performs $\frac{n}{6}$ operations

- *Example:* $N = 10240, R = \frac{210GFlops}{1.2GB} \approx 1706$

A modern CPU performs 100 GFlops, and has about 50 GB/s memory bandwidth

Instruction-Level Parallelism (ILP)

Modern processor architectures are deeply pipelined

Instruction-level parallelism (ILP) is a measure of how many of the instructions in a computer program can be executed simultaneously by issuing independent instructions in sequence (*out-of-order*)

Instruction pipelining is a technique for implementing ILP within a single processor

```
for (int i = 0; i < N; i++) // with no optimizations the loop
    sum += A[i] * B[i];     // is exectued in sequence
```

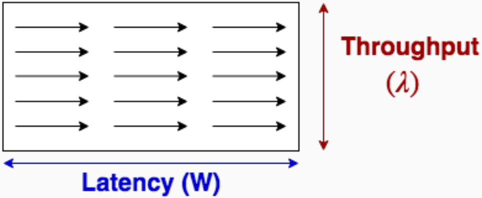
can be rewritten as:

```
for (int i = 0; i < N; i += 4) { // here, there are
    sum += A[i]     * B[i];      // four independent
    sum += A[i + 1] * B[i + 1];  // multiplications
    sum += A[i + 2] * B[i + 2];  // per iteration
    sum += A[i + 3] * B[i + 3];
}
```

The **Little's Law** expresses the relation between *latency* and *throughput*. The throughput of a system is equal to the number of elements in the system divided by the average time spent for each elements in the system:

$$L = \lambda W \quad \rightarrow \quad \lambda = \frac{L}{W}$$

- $\boldsymbol{L}$: average number of customers in a store
- $\boldsymbol{\lambda}$: arrival rate (*throughput*)
- $\boldsymbol{W}$: average time spent (*latency*)



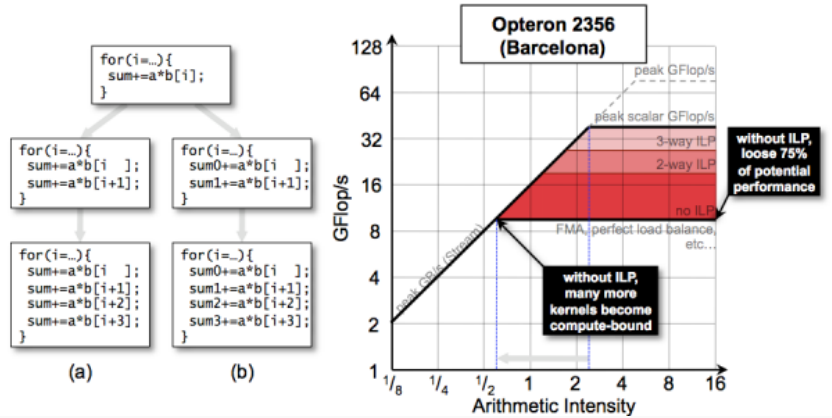
Time-Memory Trade-off

The **time-memory trade-off** is a way of solving a problem or calculation in less time by using more storage space (less often the opposite direction)

Examples:

- *Memoization* (e.g. used in dynamic programming): returning the cached result when the same inputs occur again
- *Hash table*: number of entries vs. efficiency
- *Lookup tables*: precomputed data instead branches
- *Uncompressed data*: bitmap image vs. jpeg

The **Roofline model** is a visual performance model used to provide performance estimates of a given application by showing hardware limitations, and potential benefit and priority of optimizations



(1) I/O Operations

I/O Operations

Advise: **avoid I/O**

In general, Input/Output are one of the most expensive operations

- Use **`endl`** for `ostream` only when it is strictly necessary (prefer `\n`)
- Disable *synchronization* with `printf/scanf`:
  `std::ios_base::sync_with_stdio(false)`
- Disable IO *flushing* when mixing `istream/ostream` calls:
  `<istream_obj>.tie(nullptr);`
- Increase IO *buffer size*:
  `file.rdbuf()->pubsetbuf(buffer_var, buffer_size);`

I/O Operations (Example)

```
#include <iostream>

int main() {
    std::ifstream fin;
    // --------------------------------------------------------
    std::ios_base::sync_with_stdio(false); // sync disable
    fin.tie(nullptr);                      // flush disable

                                           // buffer increase
    const int BUFFER_SIZE = 1024 * 1024;   // 1 MB
    char buffer[BUFFER_SIZE];
    fin.rdbuf()->pubsetbuf(buffer, BUFFER_SIZE);
    // --------------------------------------------------------
    fin.open(filename); // Note: open() after optimizations

    // IO operations
    fin.close();
}
```

- `printf` is faster than `ostream` (see `speed test link`)
- A printf call with the format string `%s\n` is converted to a `puts()` call

```
printf("%s\n", char_pointer);
```

- A printf call with a simple format string ending with `\n` is converted to a `puts()` call

```
printf("Hello World\n");
```

- No optimization if the string is not ending with `\n`
- No optimization if one or more `%` are detected in the format string

Reference: www.ciselant.de/projects/gcc_printf/gcc_printf.html

Memory Mapped I/O

A **memory-mapped file** is a segment of virtual memory that has been assigned a direct byte-for-byte correlation with some portion of a file

Benefits:

- Orders of magnitude faster than system calls
- Input can be "cached" in RAM memory (page/file cache)
- A file requires disk access only when a new page boundary is crossed
- Memory-mapping may bypass the page file completely
- Load and store *raw* data (no parsing/conversion)

Memory Mapped I/O (Example 1/2)

```
#if !defined(__linux__)
    #error It works only on linux
#endif
#include <fcntl.h>          //::open
#include <sys/mman.h>       //::mmap
#include <sys/stat.h>       //::open
#include <sys/types.h>      //::open
#include <unistd.h>         //::lseek
// usage: ./exec <file> <byte_size> <mode>
int main(int argc, char* argv[]) {
   size_t file_size = std::stoll(argv[2]);
   auto   is_read   = std::string(argv[3]) == "READ";
   int fd = is_read ? ::open(argv[1], O_RDONLY) :
                      ::open(argv[1], O_RDWR | O_CREAT | O_TRUNC,
                             S_IRUSR | S_IWUSR);
   if (fd == -1)
       ERROR("::open")                     // try to get the last byte
   if (::lseek(fd, static_cast<off_t>(file_size - 1), SEEK_SET) == -1)
       ERROR("::lseek")
   if (!is_read && ::write(fd, "", 1) != 1) // try to write
       ERROR("::write")
```

Memory Mapped I/O (Example 2/2)

```
    auto mm_mode = (is_read) ? PROT_READ : PROT_WRITE;

    // Open Memory Mapped file
    auto mmap_ptr = static_cast<char*>(
                ::mmap(nullptr, file_size, mm_mode, MAP_SHARED, fd, 0) );

    if (mmap_ptr == MAP_FAILED)
        ERROR("::mmap");
    // Advise sequential access
    if (::madvise(mmap_ptr, file_size, MADV_SEQUENTIAL) == -1)
        ERROR("::madvise");

    // MemoryMapped Operations
    // read from/write to "mmap_ptr" as a normal array: mmap_ptr[i]

    // Close Memory Mapped file
    if (::munmap(mmap_ptr, file_size) == -1)
        ERROR("::munmap");
    if (::close(fd) == -1)
        ERROR("::close");
}
```

Consider using optimized (low-level) numeric conversion routines:

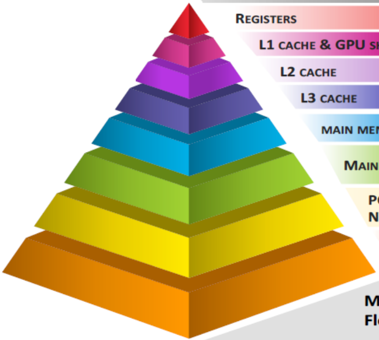
```
template<int N, unsigned MUL, int INDEX = 0>
struct fastStringToIntStr;

inline unsigned fastStringToUnsigned(const char* str, int length) {
    switch(length) {
        case 10: return fastStringToIntStr<10, 1000000000>::aux(str);
        case  9: return fastStringToIntStr< 9, 100000000>::aux(str);
        case  8: return fastStringToIntStr< 8, 10000000>::aux(str);
        case  7: return fastStringToIntStr< 7, 1000000>::aux(str);
        case  6: return fastStringToIntStr< 6, 100000>::aux(str);
        case  5: return fastStringToIntStr< 5, 10000>::aux(str);
        case  4: return fastStringToIntStr< 4, 1000>::aux(str);
        case  3: return fastStringToIntStr< 3, 100>::aux(str);
        case  2: return fastStringToIntStr< 2, 10>::aux(str);
        case  1: return fastStringToIntStr< 1, 1>::aux(str);
        default: return 0;
    }
}
```

Low-Level Parsing 2/2

```
template<int N, unsigned MUL, int INDEX>
struct fastStringToIntStr {

    static inline unsigned aux(const char* str) {
        return static_cast<unsigned>(str[INDEX] - '0') * MUL  +
               fastStringToIntStr<N - 1, MUL / 10, INDEX + 1>::aux(str);
    }

};

template<unsigned MUL, int INDEX>
struct fastStringToIntStr<1, MUL, INDEX> {

    static inline unsigned aux(const char* str) {
        return static_cast<unsigned>(str[INDEX] - '0');
    }

};
```

(3) Locality and Memory Access Patterns

Memory Locality



<table>
<tr><th>Memory hierarchies</th><th>Intel Haswell
E5-2650 v3</th><th>Intel KNL 7250
DDR5|MCDRAM</th><th>ARM
Cortex A57</th></tr>
<tr><td></td><td>10 cores
368 Gflop/s
105 Watts</td><td>68 cores
2662 Gflop/s
215 Watts</td><td>4 cores
32 Gflop/s
7 Watts</td></tr>
<tr><td>REGISTERS</td><td>16/core AVX2</td><td>32/core AVX-512</td><td>32/core</td></tr>
<tr><td>L1 CACHE & GPU SHARED MEMORY</td><td>32 KB/core</td><td>32 KB/core</td><td>32 KB/core</td></tr>
<tr><td>L2 CACHE</td><td>256 KB/core</td><td>1024 KB/2cores</td><td>2 MB</td></tr>
<tr><td>L3 CACHE</td><td>25 MB</td><td>0...16 GB</td><td>N/A</td></tr>
<tr><td>MAIN MEMORY</td><td>64 GB</td><td>384 |16 GB</td><td>4 GB</td></tr>
<tr><td>MAIN MEMORY BW</td><td>68 GB/s
5.4 flops/byte</td><td>115 | 421 GB/s
23 | 6 Flops/byte</td><td>26 GB/s
1.2 flops/byte</td></tr>
<tr><td>PCI EXPRESS GEN3x16
NVLINK</td><td>16 GB/s
23 flops/byte</td><td>16 GB/s
166 flops/byte</td><td>16 GB/s
2 flops/byte</td></tr>
<tr><td>INTERCONNECT
INFINIBAND EDR</td><td>12 GB/s
30 flops/byte</td><td>12 GB/s
221 flops/byte</td><td>12 GB/s
2.6 flops/byte</td></tr>
</table>

Memory hierarchies for different type of architectures
Flops per byte transfer (all flop rates for 64 bit operands)

Source:
"Accelerating Linear Algebra on Small Matrices from Batched BLAS to Large Scale Solvers", ICL, University of Tennessee

Memory Locality

Access to memory dominates other costs in a processor

- **Spatial Locality** refers to the use of data elements within relatively close *storage locations* e.g. scan arrays in increasing order, matrices by row
- **Temporal Locality** refers to the reuse of specific data within a relatively small *time duration*, and, as consequence, exploit lower levels of the memory hierarchy (caches)

Spatial Locality Example

A, B, C matrices of size $N \times N$

C = A * B

```
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; i++) {
        int sum = 0;
        for (int k = 0; k < N; k++)
            sum += A[i][k] * B[k][j];
        C[i][j] = sum;
    }
}
```

C = A * B^T

```
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; i++) {
        int sum = 0;
        for (int k = 0; k < N; k++)
            sum += A[i][k] * B[j][k];
        C[i][j] = sum;
    }
}
```

Spatial Locality Example

Benchmark:

N	128	256	512	1024
A * B^T				
A * B				
Speedup				

Memory-Oriented Optimizations

Head vs. Stack:

- Dynamic heap allocation is expensive: implementation dependent and interaction with the operating system
- Many small heap allocation are more expensive than one large memory allocation
- Stack memory is smaller but faster...

Maximize cache utilization:

- Prefer small data types
- Prefer `std::vector<bool>` over array of `bool`
- Prefer `std::bitset<N>` over `std::vector<bool>` if the data size is known in advance or bounded

note: modern processors have several MBs of (L1) cache

Cache Optimization Example

Speeding up a random-access function

`lemire.me/blog/2019/04/27`

Internal Structure Alignment

```
struct A1 {
    char   x1; // offset 0
    double y1; // offset 8!! (not 1)
    char   x2; // offset 16
    double y2; // offset 24
    char   x3; // offset 32
    double y3; // offset 40
    char   x4; // offset 48
    double y4; // offset 56
    char   x5; // offset 64 (byte 65)
}
```

```
struct A2 {   // internal alignment
    char   x1; // offset 0
    char   x2; // offset 1
    char   x3; // offset 2
    char   x4; // offset 3
    char   x5; // offset 4
    double y1; // offset 8
    double y2; // offset 16
    double y3; // offset 24
    double y4; // offset 32 (byte 40)
}
```

Considering an *array of structures*, there are two problems:

- We are wasting 40% of memory in the first case
- In common x64 processors the cache line is 64 bytes. For the first structure `A1`, every access involves two cache line operations

External Structure Alignment (Padding)

Considering the previous example for the structure `A2`, random loads from an array of structure `A2` leads to one or two cache line operations depending on the alignment at a specific index, e.g.

`index 0` $\rightarrow$ one cache line load

`index 1` $\rightarrow$ two cache line loads

It is possible to fix the structure alignment in two ways:

- The **memory padding** refers to introduce extra bytes at the end of the data structure to enforce the memory alignment e.g. add a `char` array of size 24 to the structure `A2`. It can be also extended to 2D (or N-D) data structures such as dense matrices
- **Align keyword or attribute** allows specifying the alignment requirement of a type or an object (next slide)

External Structure Alignment (`alignas` keyword)

C++ allows specifying the alignment requirement in three ways:

- C++03 (GCC/Clang) **`__attribute__((aligned(N)))`**
- C++11 **`alignas(N)`**
- C++17 aligned **`new`** (e.g. `new int[2, 16]`)

```
struct alignas(16) A2 { // C++11
    int x, y;
}; // __attribute__((aligned(16))); // in C++03
```

Final note: Data alignment is also important to exploit hardware vector instructions (SIMD) like SSE, AVX, etc.

(4) Arithmetic

Hardware Notes

- Instruction throughput greatly depends on processor model and characteristics
- Subtraction is implemented as an addition
- Addition, subtraction, and bitwise operations are computed by the ALU and they have very similar throughput
- Multiplication and addition are computed by the same hardware component for decreasing circuit area $\rightarrow$ multiplication and addition can be fused in a single operation
- Modern processors provide separated units for floating-point computation (FPU)

Data Types

- *Integral types are faster than floating-point* types
- *32-bit types are faster than 64-bit types*
  - 64-bit integral types are slightly slower than 32-bit integral types (modern processors widely support 64-bit operations)
  - Single precision floating-points are up to three times faster than double precision floating-points
  - In general, 32-bit type operations are hardware-implemented, while 64-bit op. requires multiple operations (both for integral and floating-point)
- *Small integral types are slower than 32-bit integer*, but they require less memory $\rightarrow$ cache/memory efficiency

Data Types

- Data type **conversions** may be expensive
  - `signed/unsigned` conversion have no cost
  - all operations on small integral type (`char`, `short`) require a conversion
  - integer to floating-point is fast, floating-point to integer is slow
- Increment `++` is faster than sum *
- Prefer **prefix** operator (`++var`) instead of the postfix operator (`var++`) *
- Use the **assignment composite** operators (`a += b`) instead of operators combined with assignment (`a = a + b`) *

* the compiler automatically applied such optimization whenever possible

Power-of-Two Multiplication/Division/Modulo

- Prefer shift for **power-of-two multiplications** (`a << b`) and **divisions** (`a >> b`) only for run-time values *
  - **`unsigned`** operations are faster than **`signed`** operations (deal with negative number)

- Prefer bitwise AND `a % b` → `a & (b - 1)` for **power-of-two modulo** operations only for run-time values *

---

* the compiler automatically applies such optimizations if `b` is known at compile-time. Bitwise operations make the code harder to read

Other Notes

- **Keep near constant values/variables** → the compiler can merge their values
- **Constant multiplication and division can be heavily optimized by the compiler, even for non-trivial values**

Other Notes

- **Multiplication is much faster than division***

`not optimized:`

```
// "value" is floating-point (dynamic)
for (int i = 0; i < N; i++)
    A[i] = B[i] / value;
```

`optimized:`

```
div = 1.0 / value;    // div is floating-point
for (int i = 0; i < N; i++)
    A[i] = B[i] * div;
```

* Multiplying by the inverse is not the same as the division
see `lemire.me/blog/2019/03/12`

Exploit Hardware Instructions

Most compilers provide **hardware-intrinsic** instructions:

`__builtin_popcount(x)`	count the number of one bits
`__builtin_clz(x)`	(`count leading zeros`) counts the number of one bits preceding the most significant zero bit
`__builtin_ctz(x)`	(`count trailing zeros`) counts the number of one bits following the least significant zero bit
`__builtin_ffs(x)`	(`find first set`) index of the least significant one bit

Usage example: `compute integer log2`

```
inline unsigned log2(unsigned x) {
    return 31 - __builtin_clz(x);
}
```

Reference: `gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html`

Low-Level Implementations

Collection of low-level implementations/optimization of common operations:

- **Bit Twiddling Hacks**
  `graphics.stanford.edu/~seander/bithacks.html`
- **The Aggregate Magic Algorithms**
  `aggregate.org/MAGIC`
- **Hackers Delight Book**
  `www.hackersdelight.org`

Low-Level Information

The same instruction/operation may take different clock-cycles on different architectures/CPU type

- **Agner Fog - Instruction tables** (latencies, throughputs)
  `www.agner.org/optimize/instruction_tables.pdf`
- **Latency, Throughput, and Port Usage Information**
  `uops.info/table.html`

(7) Control Flow

Control Flow

- **Avoid run-time recursion** (very expensive). Prefer instead *iterative* algorithms (see next slides)

- Prefer `switch` statements instead of multiple `if`. If the compiler does not use a jump-table, the cases are evaluated in order of appearance → the most frequent cases should be placed before

- Prefer **square brackets** syntax `[]` over pointer arithmetic operations for array access to facilitate compiler loop optimizations (polyhedral loop transformations)

- Prefer **signed integer** for **loop indexing**. The compiler optimizes more aggressively such loops since integer overflow is not defined

Branch

Pipelines are an essential element in modern processors. Some processors have up to 20 pipeline stages (14/16 typically)

The downside to long pipelines includes the danger of **pipeline stalls** that waste CPU time, and the time it takes to reload the pipeline on **conditional branch** operations (`if`, `while`, `for`)

Minimize Branch Overhead

- **Branch prediction**: technique to guess which way a branch takes. It requires hardware support and it is generically based on dynamic history of code executing

- **Branch predication**: a conditional branch is substituted by a sequence of instructions from both paths of the branch. Only the instructions associated to a *predicate* (boolean value), that represents the direction of the branch, are actually executed

```
int x = (condition) ? A[i] : B[i];
P = (condition) // P: predicate
@P  x = A[i];
@!P x = B[i];
```

- **Speculative execution**: execute both sides of the conditional branch to better utilize the computer resources and commit the results associated to the branch taken

Loop hoisting optimization

Wrong:

```
for (int i = 0; i < 100; i++)
    a[i] = x + y;
```

Correct:

```
v = x + y
for (int i = 0; i < 100; i++)
    a[i] = v;
```

Loop hoisting is also important in the evaluation of loop conditions

Wrong:

```
// "x" never changes
for (int i = 0; i < f(x); i++)
   a[i] = y;
```

Correct:

```
int limit = f(x)
for (int i = 0; i < limit; i++)
   a[i] = y;
```

In the worst case, `f(x)` is evaluated every iteration (especially when it belongs to another translation unit)

---

the compiler already applies such optimization when it is safe (it does not change the program semantic)

Do not hoist pointer computation!!

Example: matrix multiplication `N x N`

```
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        for (int k = 0; k < N; k++)
            C[i * N + j] += A[i * N + k] * B[k * N + j];
    }
}
```

continue...

The following code is equivalent and it apparently minimizes the number of instructions

remember `A[i] = A + i * sizeof(A_type)`

```
auto A_ptr = A;
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        auto B_ptr = B + j;
        for (int k = 0; k < N; k++) {
            *C_ptr += *(A_ptr) * (*B_ptr);
            A_ptr++;
            B_ptr += N;
        }
        C_ptr++;
    }
    A_ptr += N;
}
```

The `version 2` (with pointer hoisting) **is slower** than `version 1`

Loop unrolling (or **unwinding**) is a loop transformation technique which optimizes the code by removing (or reducing) loop iterations

The optimization produces better code at the expense of binary size

Example:

```
for (int i = 0; i < N; i++)
    sum += A[i];
```

can be rewritten as:

```
for (int i = 0; i < N; i += 8) {
    sum += A[i];
    sum += A[i + 1];
    sum += A[i + 2];
    sum += A[i + 3];
    ...
} // we suppose N is a multiple of 8
```

Loop unrolling notes:

+ Improve instruction-level parallelism (ILP)
+ Allow vector (SIMD) instructions
+ Reduce control instructions and branches
- Increase compile-time/binary size
- Require more instruction decoding
- Use more memory and instruction cache

Unroll directive The `Intel`, `IBM`, and `clang` compilers (but not `GCC`) provide the preprocessing directive `#pragma unroll` (to insert above the loop) to force loop unrolling. The compiler already applies the optimization in most cases

see `lemire.me/blog/2019/04/12`

Loop Unswitching and Fusion

Loop Unswitching

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

Loop Fusion (Jamming)

```
for (i = 0; i < 300; i++)
    a[i] = a[i] + 3;
for (i = 0; i < 300; i++)
    b[i] = b[i] + 4;
```

```
for (i = 0; i < 300; i++) {
    a[i] = a[i] + 3;
    b[i] = b[i] + 4;
}
```

Loop unswitching and loop fusion do not produce better code, but loop merging/splitting has implications on cache usage

---

In many cases, the compiler already applies these optimizations

(7) Functions

Functions

Function calls require two jumps, in addition to stack memory manipulation.

Argument Passing:

pass-by-value small data types ($\leq$ 8/16 bytes).
The data are copied into registers, instead of stack

pass-by-pointer introduces one level of indirection.
They should be used only for raw pointers (potentially `NULL`)

pass-by-reference *may* introduce one level of indirection.
`pass-by-reference` is more efficient than `pass-by-pointer` as it facilitates variable elimination by the compiler, and the function code does not require checking for `NULL` pointer

- `const` modifier applied to pointers and references help the compiler to optimize the code since the data never change and are read-only

- Keep small the number of function parameters

- Consider combining several function parameters in a structure. It allows copying parameter into stack more efficiently

- `inline` decorator: *increase inlining compiler heuristic threshold* (not force), and allow breaking the one definition rule (ODR) → inlining in multiple translation units

`restrict` pointers

GCC/Clang/Visual Code: `__restrict`

C++ Objects

Variable/Object Scope

Declare local variable in the inner most scope

- the compiler will be able to fit them into registers instead stack
- it improves readability

Wrong:

```
int i, x;
for (i = 0; i < N; i++) {
    x    = value * 5;
    sum += x;
}
```

Correct:

```
for (int i = 0; i < N; i++) {
    int x   = value * 5;
    sum    += x;
}
```

Exception! Built-in type variables and passive structures should be placed in the inner most loop, while objects with constructors should be placed outside loops

```
for (int i = 0; i < N; i++) {
    std::string str("prefix_");
    std::cout << str + value[i];
} // str call CTOR/DTOR N times
```

```
std::string str("prefix_");
for (int i = 0; i < N; i++) {
    std::cout << str + value[i];
}
```

C++ Objects

- Prefer **initializations** instead of assignments (also for variables)
- Prefer **move semantic** instead of copy constructor. Mark copy constructor as `=delete` (sometimes it is hard to see, e.g. implicit)
- Avoid multiple + operations between objects to avoid temporary storage (need example)
- Mark **final** all *virtual* functions that are not overridden
- Avoid dynamic operations: **exceptions**, **`dynamic_cast`**, **smart pointer**

Object Implicit Conversion

```cpp
#include <algorithm> // std::copy
struct A {   // big object
    int array[10000];
};

struct B {
   int array[10000];

   B(const A& a) {
      std::copy(a.array, a.array + 10000, array)
   }
};

void f(const B& b) {}

int main() {
   B b;
   f(b); // no cost
   A a;
   f(a); // very costly
}
```

(4) 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)

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

- Newer compiler produces better code
  - Effective optimizations
  - Support for newer CPU architectures
- New warnings to avoid common errors
- Better support for existing error/warnings (e.g. code highlights), less bugs, and faster compiling

Some compilers can produce better 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`

Compiler 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

`-O3` turns on all optimizations specified by -O2, plus some more. `-O3` does not guarantee to produce faster code than `-O2`

`-ffast-math` enables high level optimizations and aggressive optimizations on arithmetic calculations (like floating point re-association) $\rightarrow$ in general, it implies less floating-point accuracy (not included in `-O3`)

`-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`)

`-march=native` generate instructions for a specific machine by determining the processor type at compilation time (not included in `-O3`)

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

`-flto` enable Link Time Optimizations (Interprocedural Optimization) where the linker merges all modules into a single combined module for optimization

Note: The linker must support this feature: `GNU ld v2.21++` or gold version, to check with `ld --version`

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*

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

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

(5) Libraries and Data Structures

External Libraries

Consider using optimized *external* libraries for critical program operations

Popular libraries:

- `malloc` **replacement:** `tcmalloc`
- **Linear Algebra:** `Eigen`, `Armadillo`, `Blaze`
- **Map/Set:** `B+Tree` as replace for `std::map`
- **Hash Table:** (replace for `std::unsorted_set/map`)
  - `Google Sparse/Dense Hash Table`
  - `bytell hashmap`
  - `Facebook F14 memory efficient hash table`
- **Print and formatting:** `fmt library`
- **Random generator** `PCG` random generator

C++ Default Library

- Avoid old `C` library routines such as `qsort`, `bsearch`, etc. Prefer instead `std::sort`, `std::binary_search`
- `std::fill` applies `::memset` if inputs are continuous iterators
- 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
- Most data structures are implemented over the heap. Consider re-implement them over the stack if the number of elements to insert is small (e.g. queue)
- Prefer `lambda` expression (or `struct function`) 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

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`

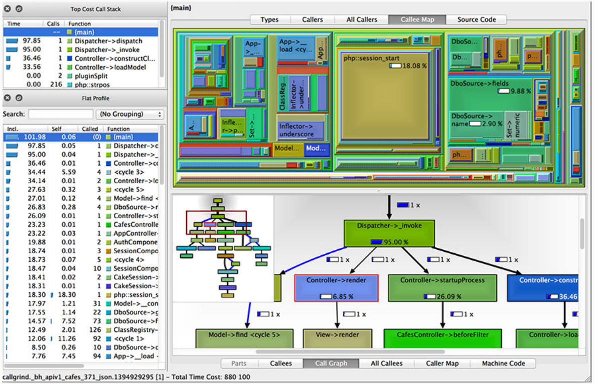
```
$ valgrind --tool cachegrind --branch-sim=yes <executable> <args>
```

- Inspect the output (cache misses and rate)
  - `I1` 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`

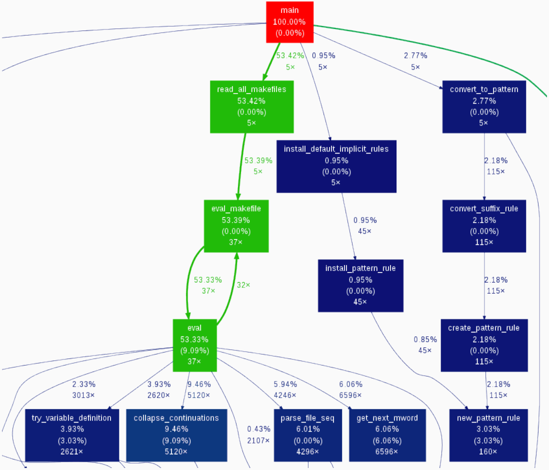
- `kcachegrind.sourceforge.net/html/Home.html`
- `sourceforge.net/projects/qcachegrindwin`



gprof2dot (View)

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

Website: `github.com/jrfonseca/gprof2dot`



Linux profiler

Linux profiler Perf- A Performance Monitoring and Analysis Tool for Linux e Performance Monitoring Unit in the CPU

Perf uses statistical profiling, where it polls the program and sees what function is working

sudo apt install linux-tools

`https://perf.wiki.kernel.org/index.php/Main_Page` man perf-subcommand

Linux profiler

```
$ perf perf record ./fib
```

[perf record: Woken up 10 times to write data] [perf record: Captured and wrote 2.336 MB perf.data (60690 samples)]

perf report

To get the call graph $ perf record -g ./fib perf record -g 'graph,0.5,caller'

perf record –call-graph dwarf – yourapp perf report -g graph –no-children

Hotspot

a GUI for the Linux perf profiler

https://www.kdab.com/hotspot-gui-linux-perf-profiler/

flare graph

http://www.brendangregg.com/FlameGraphs/cpuflamegraphs.html

(8) 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

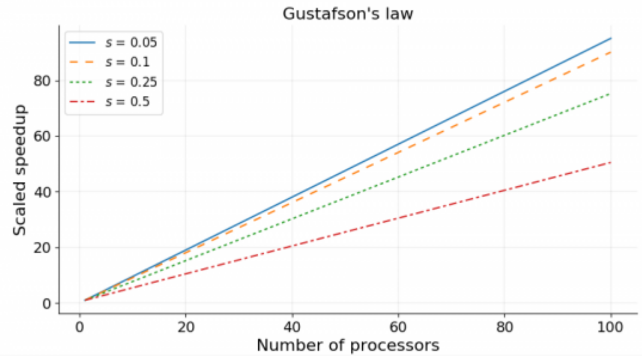
The **weak scaling** defined how the compute time decrease increasing the number of processors for a fixed total problem size per processor

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

Gustafson's Law

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

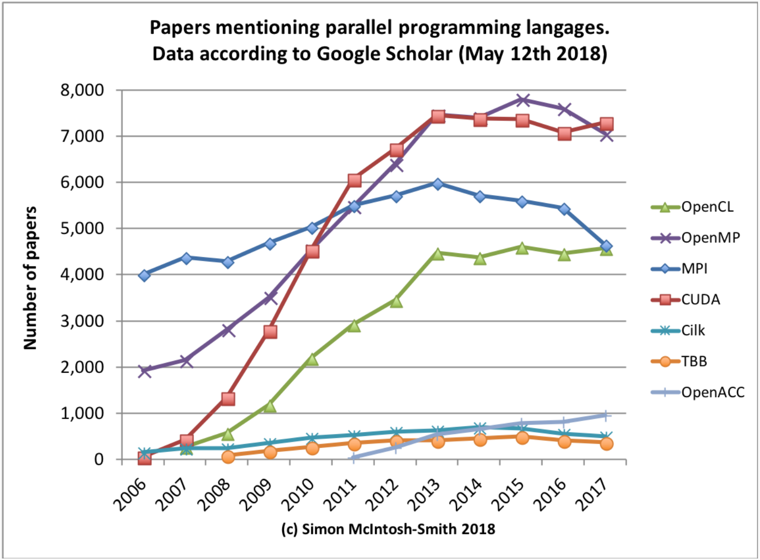
Multiple problem instances can run concurrently with more computational resources



Parallel Programming Languages

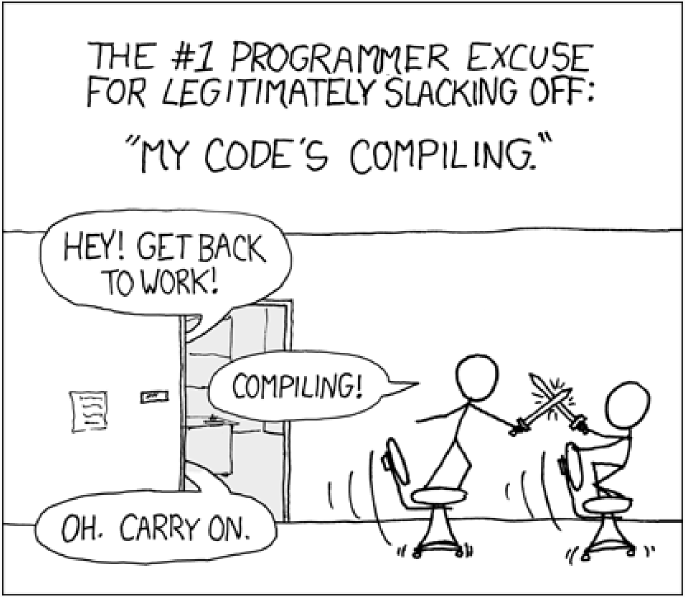
Most popular parallel programming languages based on C++:

C++11 Threads	(+ Parallel STL) (Free)
OpenMP	(Free, directive based)
OpenACC	(Free, directive based)
CUDA	(Free)
OpenCL	(Free)
Intel TBB	(Commercial)
Intel Cilk Plus	(Commercial)
KoKKos	(Free)


Papers mentioning parallel programming langages.
Data according to Google Scholar (May 12th 2018)
Number of papers
8,000
7,000
6,000
5,000
4,000
3,000
2,000
1,000
0
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
OpenCL
OpenMP
MPI
CUDA
Cilk
TBB
OpenACC
(c) Simon McIntosh-Smith 2018

Compile Time

i686-linux-gnu-ld.gold
slides.com/onqtam/faster_builds


THE #1 PROGRAMMER EXCUSE FOR LEGITIMATELY SLACKING OFF:
"MY CODE'S COMPILING."
HEY! GET BACK TO WORK!
COMPILING!
OH. CARRY ON.

Compile Time

tmpfs

ccache

precompiled header (PCH)