Modern C++ Programming

16. Code Optimization I

Federico Busato

University of Verona, Dept. of Computer Science
2020, v3.02

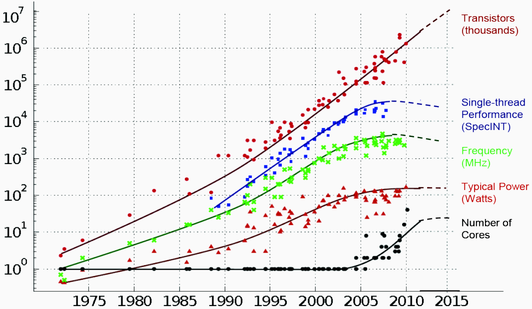
1 General Concepts

Table of Context

Table of Context

Reasons for optimizing:

- In the first decades, the *computer performance was extremely limited*. Low-level optimizations were required to fully exploit the hardware
- Modern systems provide much higher performance, but *we cannot more rely on hardware improvement* on short-period



Going the Other Way

- Computing systems are unfathomably complex
- Optimization is complicated and surprising
- Doing something sensible had opposite effect
- We often try clever things that don't work

- How about trying something silly then?

25 / 72

from *"Speed is Found in the Minds of People"*,
Andrei Alexandrescu, CppCon 2019

References

- `Optimized C++`, *Kurt Guntheroth*
- `Awesome C/C++ performance optimization resources`, *Bartlomiej Filipek*
- `Optimizing C++`, *wikibook*
- `Optimizing software in C++`, *Agner Fog*
- `Hacker Delight` (2nd), *Henry S. Warren*

General Concepts

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 (think to *insertion sort*)
- 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 that only **real-world problems** with a 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

"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"

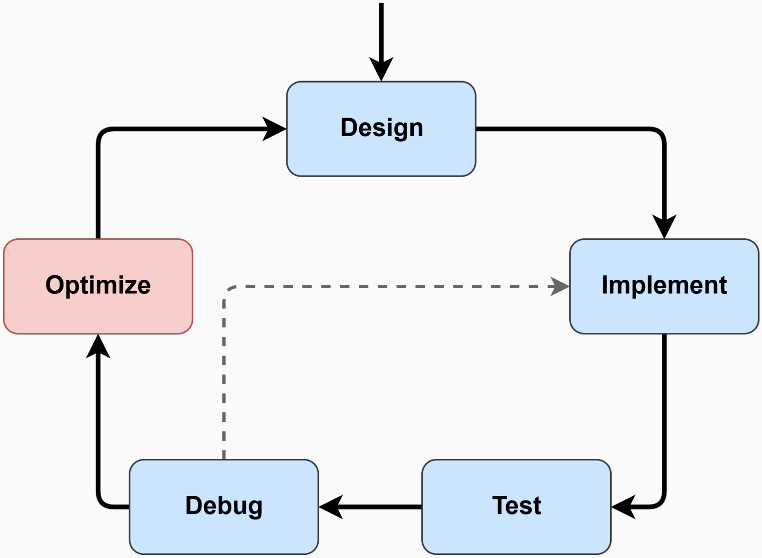
Tony Hoare, *Turing Award*

"Premature optimization is the root of all evil"

Donald Knuth, *Turing Award*

"Code for correctness first, then optimize!"

"First solve the problem, then write the code"


Design
Implement
Test
Debug
Optimize

- One of the most important phase of the optimization cycle is the **application profiling** for finding regions of code that are *critical for performance* (**hotspot**)
  - Expensive code region (absolute)
  - Code regions executed many times (cumulative)

- Most of the times, **there is no the perfect algorithm for all cases** (e.g. insertion, merge, radix sort). Optimizing refers also in finding the correct heuristics for different program inputs instead of modifying the existing code

Ahmdal Law 1/2

Ahmdal Law

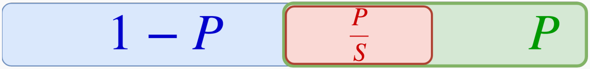
The **Ahmdal law** expresses the maximum improvement possible by improving a particular part of a system

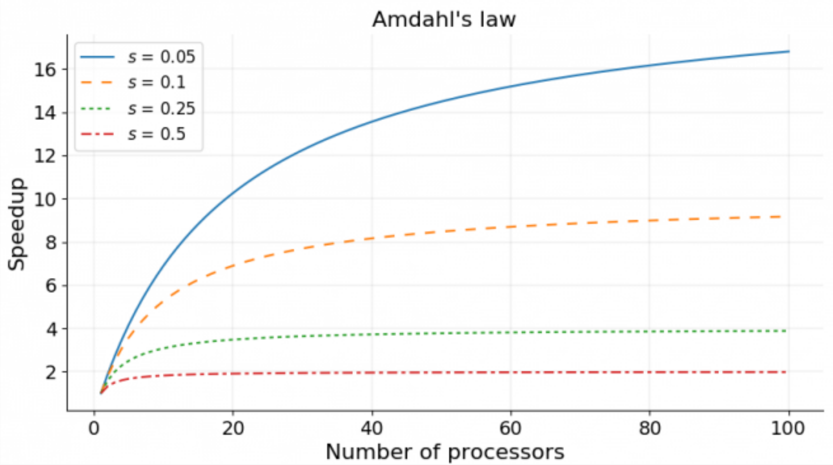
Observation: The performance of any system is constrained by the speed or capacity of the slowest point

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

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*

Performance Bounds 2/2

- **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/Operational 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

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

which means that for every byte accessed, the algorithm performs $\frac{n}{6}$ operations $\rightarrow$ **compute-bound**, but...

- *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 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:

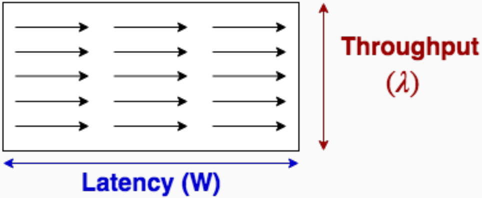
```
for (int i = 0; i < N; i += 4) { // here, there are
    sum += A[i]     * B[i];      // four independent
    sum += A[i + 1] * B[i + 1];  // multiplications
    sum += A[i + 2] * B[i + 2];  // per iteration
    sum += A[i + 3] * B[i + 3];
}
```

ILP and Little's Law

The **Little's Law** expresses the relation between *latency* and *throughput*. The throughput of a system $\boldsymbol{\lambda}$ is equal to the number of elements in the system divided by the average time spent $\boldsymbol{W}$ 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

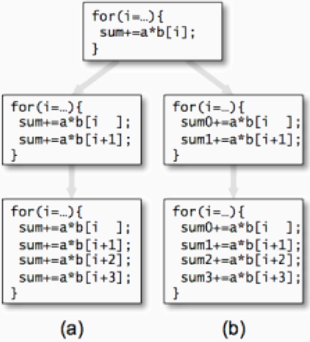
The **time-memory trade-off** is a way of solving a problem or calculation in less time by using more storage space (less often the opposite direction)

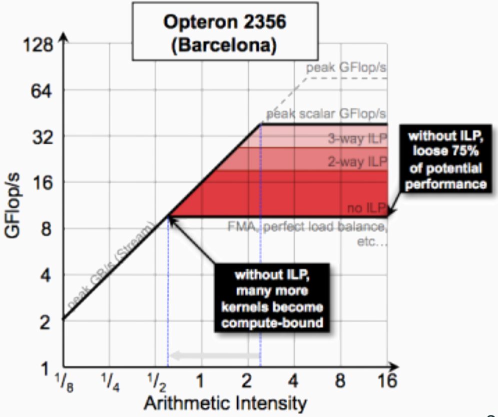
Examples:

- *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

Roofline Model

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





I/O Operations

I/O Operations are orders of magnitude slower than memory accesses

I/O Operations

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

- `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", string);
```

- 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;
    }
}
```

```
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');
    }
};
```

Faster parsing: lemire.me/blog/tag/simd-swar-parsing

Speed Up Raw Data Loading

- Hard disk is orders of magnitude slower than RAM
- Parsing is faster than hard disk data reading
- Parsing can be avoided by using *binary* storage and `mmap`
- Decreasing the number of hard disk accesses improves the performance → **compression**

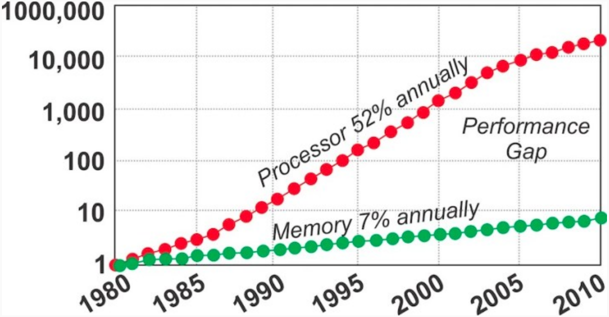
LZ4 is lossless compression algorithm providing *extremely fast decompression* up to 35% of `memcpy` and good compression ratio

`github.com/lz4/lz4`

Locality and Memory Access Patterns

Access to memory dominates other costs in a processor

The Memory Wall:



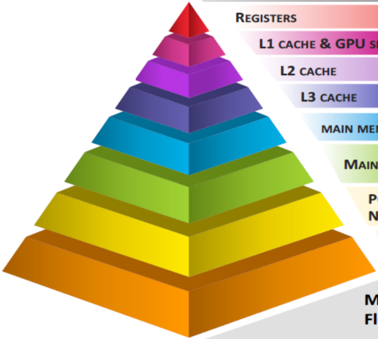
Memory Hierarchy

Modern architectures rely on complex memory hierarchy (primary memory, caches, registers, scratchpad memory, etc.). Each level has different characteristics and constrains (size, latency, bandwidth, concurrent accesses, etc.)

1 byte of RAM (1946)

IBM 5MB hard drive (1956)

twitter.com/MIT_CSAIL



Memory hierarchies	Intel Haswell E5-2650 v3	Intel KNL 7250 DDR5|MCDRAM	ARM Cortex A57
	10 cores 368 Gflop/s 105 Watts	68 cores 2662 Gflop/s 215 Watts	4 cores 32 Gflop/s 7 Watts
Registers	16/core AVX2	32/core AVX-512	32/core
L1 cache & GPU shared memory	32 KB/core	32 KB/core	32 KB/core
L2 cache	256 KB/core	1024 KB/2cores	2 MB
L3 cache	25 MB	0...16 GB	N/A
Main memory	64 GB	384 |16 GB	4 GB
Main memory BW	68 GB/s 5.4 flops/byte	115 | 421 GB/s 23 | 6 Flops/byte	26 GB/s 1.2 flops/byte
PCI express gen3x16 NVLink	16 GB/s 23 flops/byte	16 GB/s 166 flops/byte	16 GB/s 2 flops/byte
Interconnect InfiniBand EDR	12 GB/s 30 flops/byte	12 GB/s 221 flops/byte	12 GB/s 2.6 flops/byte

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

Intel Coffee Lake Core-i7-8700 example:

Cache level	Size	Latency	Bandwidth
L1 cache	192 KB	$\sim$ 1.5 ns	$\sim$ 1,600 GB/s
L2 cache	1.5 MB	$\sim$ 4 ns	$\sim$ 570 GB/s
L3 cache	12 MB	$\sim$ 12 - 40 ns	$\sim$ 320 GB/s
DRAM	/	$\sim$ 60 ns	$\sim$ 40 GB/s

en.wikichip.org/wiki/WikiChip
www.forrestthewoods.com/blog/memory-bandwidth-napkin-math/

Memory Locality

- **Spatial Locality** refers to the use of data elements within *relatively close storage locations* e.g. scan arrays in increasing order, matrices by row. It involves mechanisms such as *memory prefetching* and *access granularity*

- **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)

  Heavily used memory locations can be accessed more quickly than less heavily used locations

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

`C = A * B`^{T}

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

Benchmark:

N	64	128	256	512	1024
A * B	$<$ 1 ms	5 ms	29 ms	141 ms	1,030 ms
A * B^T	$<$ 1 ms	2 ms	6 ms	48 ms	385 ms
Speedup	/	2.5x	4.8x	2.9x	2.7x

Cache Optimization Example

Speeding up a random-access function

```
for (int i = 0; i < N; i++)       // V1
    out_array[i] = in_array[hash(i)];
```

```
for (int i = 0; i < N; i++) {    // V2
    for (int K = 0; K < N; K += CACHE) {
        auto x = hash(i);
        if (x >= K && x < K + CACHE)
            out_array[i] = in_array[x];
    }
}
```

`V1`: 436 ms, `V2`: 336 ms → 1.3x speedup

.. but it needs a careful evaluation of `CACHE` and it can even decrease the performance for other sizes

pre-sorted `hash(i)`: 135 ms → 3.2x speedup

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

Heap Memory

- *Dynamic heap allocation is expensive*: implementation dependent and interaction with the operating system
- *Many small heap allocations are more expensive than one large memory allocation*
  The default page size on Linux is 4KB. For smaller/multiple sizes, C++ uses a suballocator
- *Allocations within the page size is faster than larger allocations* (suballocator)

Stack Memory

- *Stack memory is faster than heap memory*. The stack memory provides high locality

- `static` stack allocations produces better code. It avoids filling the stack each time the function is reached

- `constexpr` for arrays with dynamic indexing produces very inefficient code with GCC. Use `static constexpr` instead

```
void f(int x) { // bad performance with GCC
    constexpr int array[] = {1,2,3,4,5,6,7,8,9};
    return array[x];
}
```

Memory-Oriented Optimizations

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

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 (`A1`)
- In common x64 processors the cache line is 64 bytes. For the first structure `A1`, every access involves two cache line operations (2x slower)

see also `#pragma pack(1)`

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)

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

- C++11 `alignas(N)` only for variable / struct declaration
- C++17 aligned `new` (e.g. `new int[2, N]`)
- Compiler Intrinsic only for variables / struct declaration
  - GCC/Clang: `__attribute__((aligned(N)))`
  - MSVC: `__declspec(align(N))`
- Compiler Intrinsic for dynamic pointer
  - GCC/Clang: `__builtin_assume_aligned(x)`
  - Intel: `__assume_aligned(x)`

Data alignment is essential for exploiting hardware vector instructions (SIMD) like SSE, AVX, etc.

External Structure Alignment in C++ 2/2

```
struct alignas(16) A1 { // C++11
    int x, y;
};

struct __attribute__((aligned(16))) A2 { // require compiler
    int x, y;                            // support
};

auto ptr1 = new int[100, 16]; // 16B alignment
auto ptr2 = new int[100];     // 4B alignment guarantee
auto ptr3 = __builtin_assume_aligned(ptr2, 16);
// require compiler support
```

Arithmetic

Hardware Notes

- Instruction throughput greatly depends on processor model and characteristics
- *Addition*, *subtraction*, and *bitwise operations* are computed by the ALU and they have very similar throughput
- In modern processors, *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 `fma` (floating-point) and `mad` (integer)
- Modern processors provide separated units for floating-point computation (FPU)

Data Types

- **32-bit integral vs. floating-point**: in general, integral types are faster, but it depends on the processor characteristics

- **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 native 64-bit instructions for most operations, otherwise they require multiple operations
  - Single precision floating-points are up to three times faster than double precision floating-points

- **Small integral types are slower than 32-bit integer**, but they require less memory $\rightarrow$ cache/memory efficiency

Operations

- In modern architectures, arithmetic increment/decrement `++` / `--` has the same performance of `add` / `sub`
- **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`) *
- **Keep near constant values/variables** → the compiler can merge their values

* the compiler automatically applies such optimization whenever possible (this is not ensured for object types)

Integer Multiplication

- Integer multiplication requires double the number of bits of the operands
- Cast one of the two operands to a bigger integer has not cost

```
// gcc -m32 (32-bit system)
int f1(int x, int y) {
    return x * y; // efficient
}
int64_t f2(int x, int y) {
    return x * static_cast<int64_t>(y); // efficient!!
}
int64_t f3(int64_t x, int64_t y) {
    return x * y; // very slow
}
```

Power-of-Two Multiplication/Division/Modulo

- Prefer shift for **power-of-two multiplications** (`a << b`) and **divisions** (`a >> b`) only for run-time values *

- Some **`unsigned`** operations are faster than **`signed`** operations (deal with negative number), e.g. `x / 2`

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

- **Constant multiplication and division** can be heavily optimized by the compiler, even for non-trivial values

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

Conversion

From	To	Cost
`Signed`	`Unsigned`	no cost, bit representation is the same
`Unsigned`	`Larger Unsigned`	no cost, register extended
`Signed`	`Larger Signed`	1 clock-cycle, register + sign extended
`Integer`	`Floating-point`	4-16 clock-cycles Signed → Floating-point is faster than Unsigned → Floating-point (except `AVX512` instruction set is enabled)
`Floating-point`	`Integer`	fast if `SSE2`, slow otherwise (50-100 clock-cycles)

Reference: `Optimizing software in C++`, *Agner Fog*

Floating-Point Division

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`

Floating-Point FMA

CPU processors perform computations with a larger register size than the original data type (e.g. 48-bit for 32-bit floating-point) to achieve this result

Modern processors allow performing `a * b + c` in a single operation, called **fused multiply-add** (`std::fma` in C++11). This implies better performance and accuracy

Compiler behavior:

- GCC 9 and ICC 19 produce a single instruction for `std::fma` and for `a * b + c` with `-O3 -march=native`
- Clang 9 and MSVC 19.* produce a single instruction for `std::fma` but not for `a * b + c`

FMA: solve quadratic equation

FMA: extended precision addition and multiplication by constant

Compiler intrinsics are highly optimized functions directly provided by the compiler instead of external libraries

Advantages:

- Directly mapped to hardware functionalities if available
- Inline expansion
- Do not inhibit high-level optimizations and they are portable contrary to `asm` code

Drawbacks:

- Portability is limited to a specific compiler
- Some intrinsics do not work on all platforms
- The same instricics can be mapped to a non-optimal instruction sequence depending on the compiler

Most compilers provide intrinsics **bit-manipulation functions** for `SSE4.2` or `ABM` (Advanced Bit Manipulation) instruction sets for Intel and AMD processors

GCC examples:

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

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

- Compute integer log2

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

- Check if a number is a power of 2

```
inline bool is_power2(unsigned x) {
    return __builtin_popcount(x) == 1;
}
```

- Bit search and clear

```
inline int bit_search_clear(unsigned x) {
    int pos = __builtin_ffs(x); // range [0, 31]
    x      &= ~(1u << pos);
    return pos;
}
```

Example of intrinsic portability issue:

`__builtin_popcount()` GCC produces `__popcountdi2` instruction while Intel Compiler (ICC) produces 13 instructions

`_mm_popcnt_u32` GCC and ICC produce `popcnt` instruction, but it is available only for processor with support for `SSE4.2` instruction set

More advanced usage

- Compute CRC: `_mm_crc32_u32`
- AES cryptography: `_mm256_aesenclast_epi128`
- Hash function: `_mm_sha256msg1_epu32`

Reference: `software.intel.com/sites/landingpage/IntrinsicsGuide/`

Using intrinsic instructions are extremely dangerous if the target processor does not natively support such instructions

Example:

> *"If you run code that uses the intrinsic on hardware that doesn't support the `lzcnt` instruction, the results are unpredictable" - MSVC*

on the contrary, GNU and clang `__builtin_*` instructions are always well-defined. The instruction is translated to a non-optimal operation sequence in the worst case

The instruction set support should be checked at *run-time* (e.g. with `__cpuid` function on MSVC), or, when available, by using compiler-time macro (e.g. `__AVX__`)

Value in a Range

Checking if a non-negative value *x* is within a range *[A, B]* can be optimized if *B* $>$ *A* (useful when the condition is repeated multiple times)

```
if (x >= A && x <= B)

// STEP 1: subtract A
if (x - A >= A - A && x - A <= B - A)
// -->
if (x - A >= 0 && x - A <= B - A) // B - A is precomputed

// STEP 2
//   - convert "x - A >= 0" --> (unsigned) (x - A)
//   - ensure that "B - A" is not less than zero
if ((unsigned) (x - A) <= (unsigned) (B - A))
// works even if A, B are negative (B >= A)
```

Value in a Range Examples

Check if a value is an uppercase letter:

```
uint8_t x = ...
if (x >= 'A' && x <= 'Z')
    ...
```

→

```
uint8_t x = ...
if (x - 'A' <= 'Z')
    ...
```

A more general case:

```
int x = ...
if (x >= -10 && x <= 30)
    ...
```

→

```
int x = ...
if ((unsigned) (x + 10) <= 40)
    ...
```

The compiler applies this optimization only in some cases (tested with GCC/Clang 9 -O3)

Lookup Table

Lookup table is a *memoization* technique which allows replacing *runtime* computation with precomputed values

Example: a function that computes the logarithm base 10 of a number in the range [1-100]

```
#include <array> // the code requires C++17
#include <cmath>

template<int SIZE, typename Lambda>
constexpr std::array<float, SIZE> build(Lambda lambda) {
    std::array<float, SIZE> array{};
    for (int i = 0; i < SIZE; i++)
        array[i] = lambda(i);
    return array;
}
float log10(int value) {
    constexpr auto lamba = [](int i) { return std::log10f((float) i); };
    static constexpr auto table = build<100>(lambda);
    return table[value];
}
```

Basic Bit Manipulation

```
x ^ 0  == x
x ^ ~0 == ~x

-x == two-complement == ~x + 1

x & -x            // get the least significant bit
x | -x            // set one after the least significant bit

(x & (x - 1))     // clear the least significant bit
x ^ (x & (x - 1)) // get the least significant bit
```

Low-Level Optimizations

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`

Control Flow

Computation is faster than decision

Branches are expensive 2/2

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`)

Control Flow

- Prefer `switch` statements instead of multiple `if`
  - If the compiler does not use a jump-table, the cases are evaluated in order of appearance $\rightarrow$ the most frequent cases should be placed before
  - Some compilers (e.g. `clang`) are able to translate a sequence of `if` into a `switch` statement

- 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

- Some compilers (e.g. `clang`) use assertion for optimization purposes: most likely code path, not possible values, etc. *

* Andrei Alexandrescu on Twitter

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

Base case:

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

Better:

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

Loop hoisting is also important in the evaluation of loop conditions

Base case:

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

Better:

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

In the worst case, `f(x)` is evaluated at 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)

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;
}
```

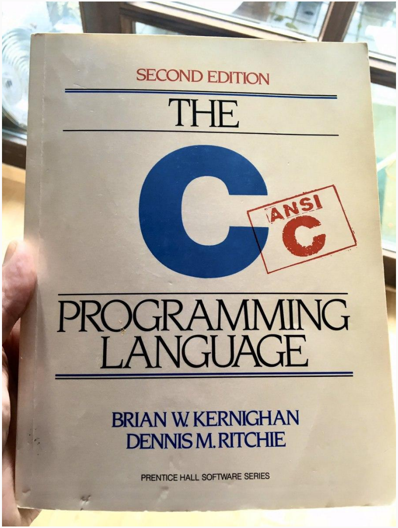
Loop unswitching and loop fusion do not produce better code, but loop merging/splitting has implications on cache usage

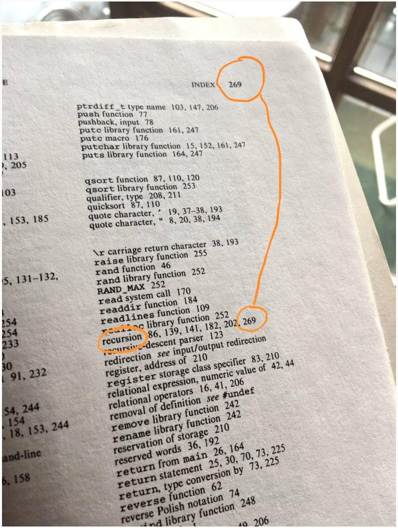
In many cases, the compiler already applies these optimizations

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

Recursion cost: The program must store all variables (snapshot) at each recursion iteration on the stack, and remove them when the control return to the caller instance

The **tail recursion** optimization avoids to maintain caller stack and pass the control to the next iteration. The optimization is possible only if all computation can be executed before the recursive call





Via Twitter - Jan Wildeboer

Functions

Function Call Cost

Function call methods:

Direct: Function address is known at compile-time

Indirect: Function address is known only at run-time

Inline: The function code is fused in the caller code

Function call cost:

- The caller pushes the arguments on the stack in reverse order
- Jump to function address
- The caller clears (pop) the stack

Reference: `The True Cost of Calls`

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

Most compilers optimize **pass by-value** with **pass by-reference** for *passive* data structures

For *active* objects with non-trivial (and expensive) copy constructor or destructor:

by-value	Expensive, hard to optimize
by-pointer/reference	ok. Prefer pass-by-`const`-X (`const` overloading can also be cheaper)

For *passive* objects with trivial copy constructor *and* destructor:

by-const-value	Always produce the optimal code (converted in `pass-by-ref` if needed)
by-value	Produce optimal code except for GCC (tested with GCC 9.2)
by-reference	Could introduce a level of indirection

- *Pass by-value built-in types and passive data structured (no side-effect*. The compiler already applies heuristics to determine the most efficient way to pass the parameter (by-value or by-reference). Pass by-reference does not allow the compiler to optimize in pass by-value (if not inline)

- *Keep small the number of function parameters*. The parameters can be passed by using the registers instead filling and emptying the stack

- Consider *combining several function parameters* in a structure

- `const` modifier applied to pointers and references *does not produce better code* in most cases, but it is useful for ensuring read-only accesses

- `const` applied to pass by-value *does not change the function signature* and, for this reason, should be avoided in function declaration

Inlining Constrains

Compilers have different heuristics for function inlining:

- Number of lines (even comments: `How new-lines affect the Linux kernel performance`)
- Number of assembly instructions
- Inlining depth (recursive)

A function is integrated into the caller if:

- It satisfies the compiler heuristics
- The function must have *internal linkage* (`static` or anonymous namespace)
- The `inline` keyword helps to decrease the heuristic threshold
- Compiler-specific decorators (`__attribute__((always_inlinee))` , `__forceinline`) allow skipping compiler heuristics

---

- `An Inline Function is As Fast As a Macro`
- `Inlining Decisions in Visual Studio`

Local Functions

All compilers, except MSVC, export all function symbols → slow, the symbols can be used in other translation units

Alternatives:

- Use `static` functions
- Use `anonymous namespace` (functions and classes)
- Use GNU extension (also `clang`)
  `__attribute__((visibility("hidden")))`

gcc.gnu.org/wiki/Visibility

Consider the following example:

```
// suppose f() is not inline
void f(int* input, int size, int* output) {
    for (int i = 0; i < size; i++)
        output[i] = input[i];
}
```

- The compiler cannot *unroll* the loop (sequential execution, no ILP) because `output` and `input` pointers can be **aliased**, e.g. `output = input + 1`

- The aliasing problem is even worse for more complex code and *inhibits all kind of optimization* from code re-ordering to common sub-expression elimination

Most compilers (included `GCC/Clang/MSVC`) provide **restricted pointers** (`__restrict`) so that the programmer asserts that the pointers are not aliased

```
void f(int* __restrict input,
       int             size,
       int* __restrict output) {
    for (int i = 0; i < size; i++)
         output[i] = input[i];
}
```

Potential benefits:

- Instruction-level parallelism
- Less instructions executed
- Merge common sub-expressions

Benchmarking matrix multiplication

```
void matrix_mul_v1(const int* A,
                   const int* B,
                   int        N,
                   int*       C) {
```

```
void matrix_mul_v2(const int* __restrict__ A,
                   const int* __restrict__ B,
                   int                     N,
                   int*       __restrict__ C) {
```

Optimization	-O1	-O2	-O3
v1	1,030 ms	777 ms	777 ms
v2	513 ms	510 ms	761 ms
Speedup	2.0x	1.5x	1.02x

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];
}
```

- Prefer **direct initialization** and *full object constructor* instead of two-step initialization (also for variables)

- Prefer **move semantic** instead of copy constructor. Mark copy constructor as `=delete` (sometimes it is hard to see, e.g. implicit)

- Mark **final** all *virtual* functions that are not overridden

- Avoid dynamic operations: **exceptions*** (and use `noexcept`), **`dynamic_cast`**, **smart pointer**

- **Virtual calls** are slower than standard functions

*Investigating the Performance Overhead of C++ Exceptions

- Use `static` for all members that do not use instance member (avoid passing `this` pointer)
- Avoid multiple `+` operations between objects to avoid temporary storage (need example)
- Prefer `++obj` / `--obj` (return `&obj`), instead of `obj++`, `obj--` (return old `obj`)
- Prefer `x += obj`, instead of `x = x + obj` → avoid the object copy

Object Implicit Conversion

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