

# Modern C++ Programming

## 16. CODE OPTIMIZATION I

---

*Federico Busato*

University of Verona, Dept. of Computer Science  
2021, v3.11



## 1 General Concepts

- Asymptotic Complexity
- Time-Memory Trade-off
- Optimization Cycle
- Ahmdal Law
- Throughput, Bandwidth, Latency
- Performance Bounds
- Arithmetic Intensity
- Instruction-Level Parallelism
- Little's Law
- Data-Level Parallelism
- Thread-Level Parallelism

## **2 I/O Operations**

- `printf`
- Memory Mapped I/O
- Speed Up Raw Data Loading

## **3 Locality and Memory Access Patterns**

- Memory Hierarchy
- Memory Locality
- Internal Structure Alignment
- External Structure Alignment

## **4 Arithmetic**

- Data Types
- Operations
- Conversion
- Floating-Point
- Compiler Intrinsic Functions
- Value in a Range
- Lookup Table

## **5 Control Flow**

- Loop Hoisting
- Loop Unrolling
- Branch Hints

## 6 Functions

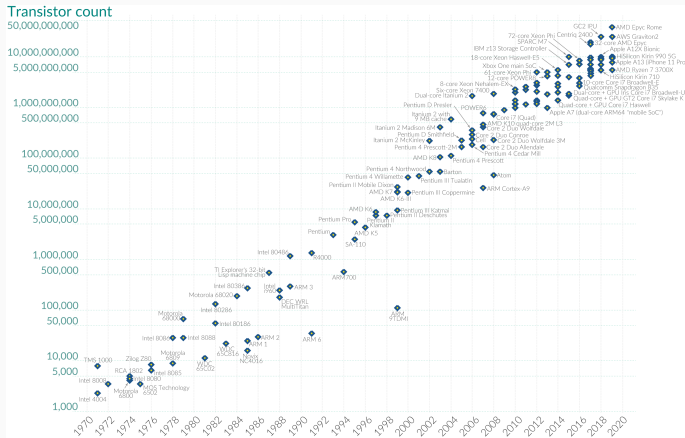
- Function Call Cost
- Argument Passing
- Function Optimizations
- Function Inlining
- Pointers Aliasing

## 7 C++ Objects

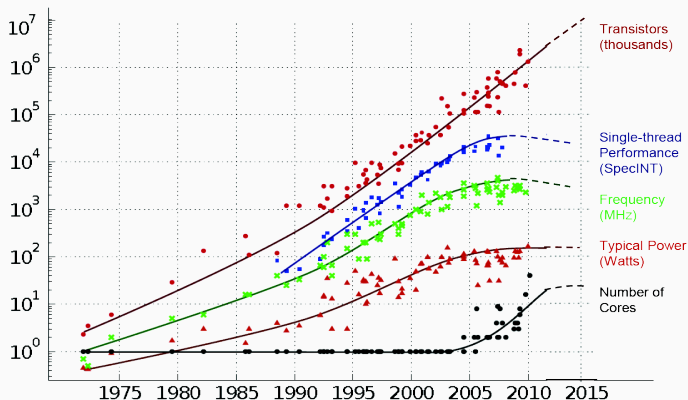
- C++ Objects Optimizations

*"The number of transistors incorporated in a chip will approximately double every 24 months." (40% per year)*

**Gordon Moore, Intel co-founder**



The Moore's Law is not (yet) dead but the same concept is not true for clock frequency, single-thread performance, and power consumption. How we can provide value?



## Reasons for Optimizing



- In the first decades, the *computer performance was extremely limited*. Low-level optimizations were essential to fully exploit the hardware
- Modern systems provide much higher performance, but *we cannot more rely on hardware improvement* on short-period
- Performance and efficiency add market value, e.g. faster program for a given task
- Optimized code uses less resources, e.g. in a program that runs on a server for months or years, a small reduction in the execution time translates in a big saving of power consumption



# The Role of Software Engineering and Algorithms

## *Forget Moore's Law. Algorithms drive technology forward*

*"Algorithmic improvements make more efficient use of existing resources and allow computers to do a task faster, cheaper, or both. Think of how easy the smaller MP3 format made music storage and transfer. That compression was because of an algorithm."*

Technology	01010011 01100011 01101001 01100101 01101110 01100011 01100101 00000000		
	<b>Software</b>	<b>Algorithms</b>	<b>Hardware architecture</b>
Opportunity	Software performance engineering	New algorithms	Hardware streamlining
Examples	Removing software bloat Tailoring software to hardware features	New problem domains New machine models	Processor simplification Domain specialization

- Forget Moore's Law
- What will drive computer performance after Moore's law?

## 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

- `Optimized C++`, *Kurt Guntheroth*
- `Awesome C/C++ performance optimization resources`, *Bartłomiej 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

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

Three 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
- *Matrix multiplication*:  $\mathcal{O}(N^3)$ , even for small sizes  $N$  (e.g. 8K, 16K), it requires special accelerators (e.g. GPU, TPU, etc.) for achieving acceptable performance

# 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

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

**Leslie Lamport**, Turing Award

*"First solve the problem, then write the code"*

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

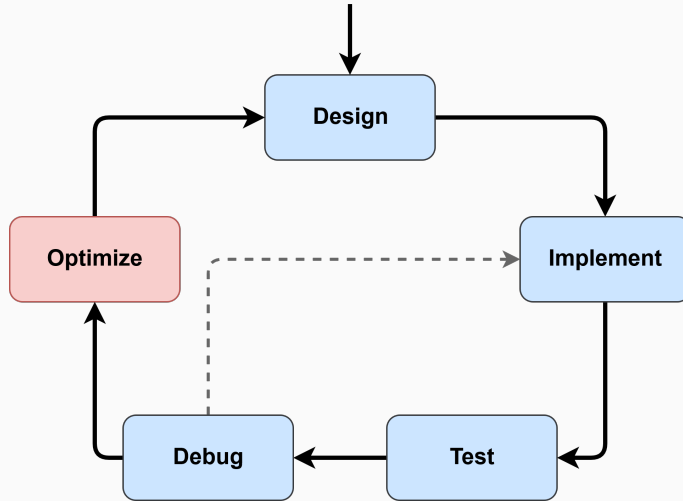
**Tony Hoare**, Turing Award

*"Premature optimization is the root of all evil"*

**Donald Knuth**, Turing Award

*"Code for correctness first, then optimize!"*





- 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**

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

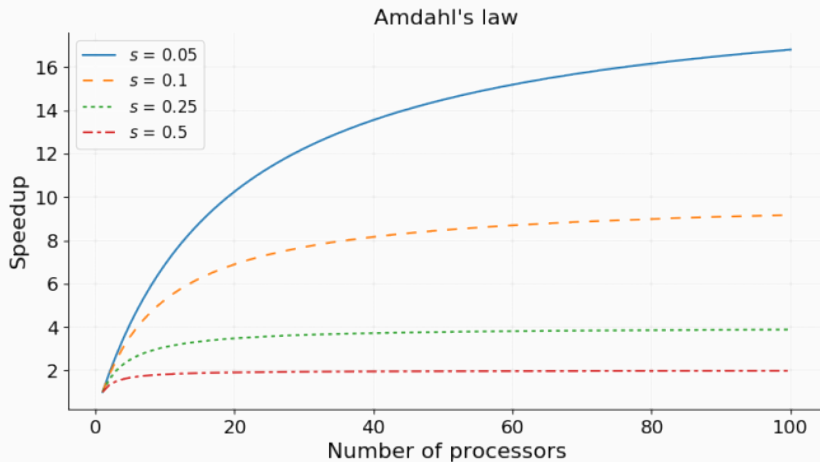
*Observation:* The performance of any system is constrained by the speed or capacity of the slowest point

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

$P$  : portion of the system that can be improved

$S$  : improvement factor





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

# Throughput, Bandwidth, Latency

The **throughput** is the rate at which operations are performed

*Peak throughput:*

$(\text{CPU speed in Hz}) \times (\text{CPU instructions per cycle}) \times$   
 $(\text{number of CPU cores}) \times (\text{number of CPUs per node})$

NOTE: modern processors have more than one computation unit

The **memory bandwidth** is the amount of data that can be loaded from or stored into a particular memory space

*Peak bandwidth:*

$(\text{Frequency in Hz}) \times (\text{Bus width in bit} / 8) \times (\text{Pump rate (memory type multiplier)})$

The **latency** is the amount of time needed for an operation to complete

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

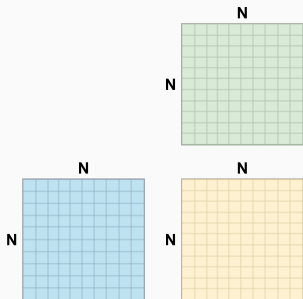
- **Memory-bound.** The program spends its time primarily in performing *memory accesses*. The progress is limited by the *memory bandwidth* (sometime memory-bound also refers to the amount of memory available)
- **Compute-bound.** The program spends its time primarily in computing *arithmetic instructions*. 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/Operational Intensity** is the ratio of total operations to total data movement (bytes or words)

The naive matrix multiplication algorithm requires  $N^3 \cdot 2$  floating-point operations (multiplication + addition), while it involves  $(N^2 \cdot 4B) \cdot 3$  data movement





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

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

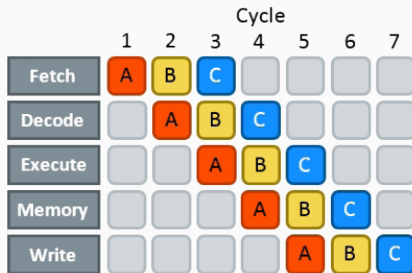
N	Operations	Data Movement	Ratio	Exec. Time
512	$268 \cdot 10^6$	3 MB	85	2 ms
1024	$2 \cdot 10^9$	12 MB	170	21 ms
2048	$17 \cdot 10^9$	50 MB	341	170 ms
4096	$137 \cdot 10^9$	201 MB	682	1.3 s
8192	$1 \cdot 10^{12}$	806 MB	1365	11 s
16384	$9 \cdot 10^{12}$	3 GB	2730	90 s

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

*Modern processor architectures are deeply pipelined → superscalar processor*

**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



Microarchitecture	Pipeline stages
Core	14
Bonnell	16
Sandy Bridge	14
Silvermont	14 to 17
Haswell	14
Skylake	14
Kabylake	14

The pipeline efficiency is affected by

- **Instruction stalls**, e.g. cache miss, an execution unit not available, etc.
- **Bad speculation**, branch misprediction

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

can be rewritten as:

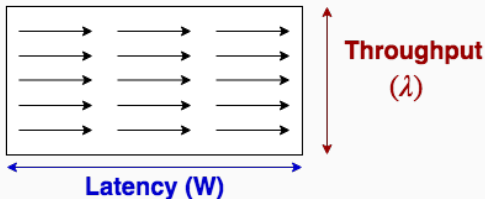
```
for (int i = 0; i < N; i += 4) { // four independent  
    C[i]      = A[i]      * B[i]; // multiplications  
    C[i + 1] = A[i + 1] * B[i + 1]; // per iteration  
    C[i + 2] = A[i + 2] * B[i + 2];  
    C[i + 3] = A[i + 3] * B[i + 3];  
}
```

## ILP and Little's Law

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

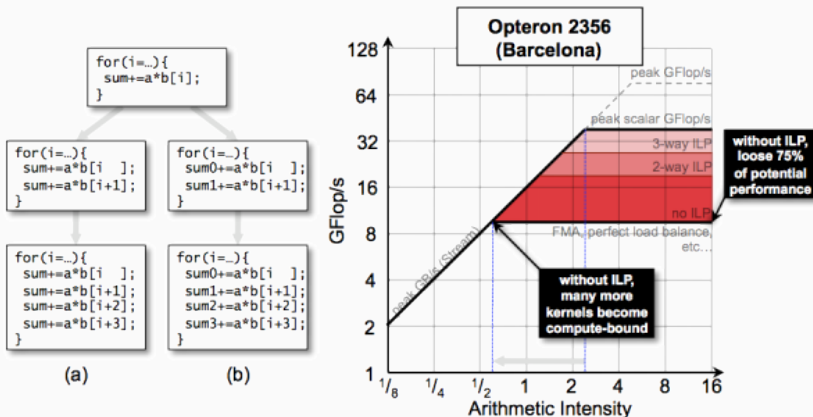
$$L = \lambda W \rightarrow \lambda = \frac{L}{W}$$

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



# 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



# Data-Level Parallelism

**Data-Level Parallelism** refers to the execution of the same operation on multiple data in parallel

*Vector processors* or *array processors* provide SIMD (*Single Instruction-Multiple Data*) or vector instructions for exploiting data-level parallelism

The popular vector instruction sets are:

**MMX** *MultiMedia eXtension*. 80-bit width (Intel, AMD)

**SSE** (SSE2, SSE3, SSE4) *Streaming SIMD Extensions*. 128-bit width (Intel, AMD)

**AVX** (AVX, AVX2, AVX-512) *Advanced Vector Extensions*. 512-bit width (Intel, AMD)

**NEON** *Media Processing Engine*. 128-bit width (ARM)

**SVE** (SVE, SVE2) *Scalable Vector Extension*. 128-2048 bit width (ARM)

# Thread-Level Parallelism

A **thread** is a single sequential execution flow within a program with its state (instructions, data, PC, register state, and so on)

**Thread-level parallelism** refers to the execution of separate computation “*thread*” on different processing units



# I/O Operations

---

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

# I/O Streams

In general, input/output operations are one of the most expensive

- 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 Streams - 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

# 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)

```
#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")
}
```

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

- Hard disk is orders of magnitude slower than RAM
- Parsing is faster than 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](https://github.com/lz4/lz4)

Another alternative is **Facebook zstd**

[github.com/facebook/zstd](https://github.com/facebook/zstd)

Performance comparison of different methods for a file of 4.8 GB of integer values

Load Method	Exec. Time	Speedup
ifstream	102 667 ms	1.0x
memory mapped + parsing (first run)	30 235 ms	3.4x
memory mapped + parsing (second run)	22 509 ms	4.5x
memory mapped + lz4 (first run)	3 914 ms	26.2x
memory mapped + lz4 (second run)	1 261 ms	81.4x

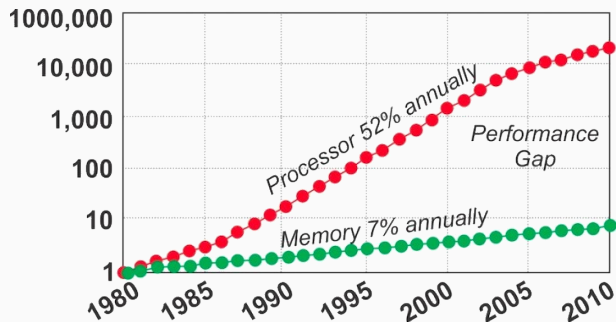
NOTE: the size of the Lz4 compressed file is 1,8 GB

# Locality and Memory Access Patterns

---

## Access to memory dominates other costs in a processor

### The Memory Wall:



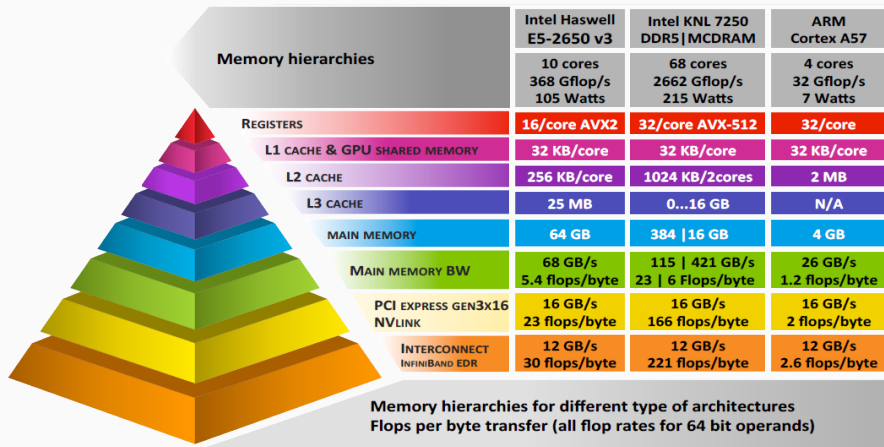
Modern architectures rely on complex memory hierarchy (primary memory, caches, registers, scratchpad memory, etc.). Each level has different characteristics and constrains (size, latency, bandwidth, concurrent accesses, etc.)



*1 byte of RAM (1946)*



*IBM 5MB hard drive (1956)*



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	~ 1.5 ns	~ 1,600 GB/s
L2 cache	1.5 MB	~ 4 ns	~ 570 GB/s
L3 cache	12 MB	~ 12 - 40 ns	~ 320 GB/s
DRAM	/	~ 60 ns	~ 40 GB/s

- [en.wikichip.org/wiki/WikiChip](https://en.wikichip.org/wiki/WikiChip)
- 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), e.g. multiple sparse accesses  
*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  $\rightarrow$  1.3x speedup (temporal locality improvement)

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

pre-sorted `hash(i)` : 135 ms  $\rightarrow$  3.2x speedup (spatial locality improvement)

# 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, it is small (cache fit), and its size is known at compile-time
- `static` stack allocations produce better code. It avoids filling the stack each time the function is reached
- `constexpr` 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};  
    static constexpr int array[] = {1,2,3,4,5,6,7,8,9};  
    return array[x];  
}
```

## 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



# 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* (AoS), 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)

## External Structure Alignment and 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 → one cache line load

index 1 → 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 different 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.**

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

struct __attribute__((aligned(16))) A2 { // compiler-specific attribute
    int x, y;
};

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

# Arithmetic

---

- 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 → 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)

- **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 → cache/memory efficiency

- 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 no 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; // slow
}
```

# Power-of-Two Multiplication/Division/Modulo

- Prefer shift for **power-of-two multiplications** (  $a \ll b$  ) and **divisions** (  $a \gg b$  ) only for run-time values \*
- Some **unsigned** operations are faster than **signed** operations (deal with negative number), e.g.  $x / 2$
- Prefer bitwise AND (  $a \% b \rightarrow a \& (b - 1)$  ) 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

Ideal divisors: when a division compiles down to just a multiplication

# 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)

# 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](https://lemire.me/blog/2019/03/12)

# Floating-Point FMA

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

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

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 intrinsics 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

- 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`

*Using intrinsic instructions is 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__`)

# Automatic Compiler Function Transformation

`std::abs` can be recognized by the compiler and transformed to an hardware instruction

In a similar way, `C++20` provides a portable and efficient way to express bit operations  
`<bit>`

```
rotate left : std::rotr
rotate right : std::rotr
count leading zero : std::countl_zero
count leading one : std::countl_one
count trailing zero : std::countr_zero
count trailing one : std::countr_one
population count : std::popcount
```

## 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))
```

## 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 (LUT)** 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]

```
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
```

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`



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](http://www.agner.org/optimize/instruction_tables.pdf)

- **Latency, Throughput, and Port Usage Information**

[uops.info/table.html](http://uops.info/table.html)

# Control Flow

---

**Computation is faster than decision**

**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` )

Not all control flow instructions involve a `jump` instructions. If the code in the branch is small, the compiler could optimize it in a conditional instruction, e.g. `ccmovl`

- 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
  - 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
- Prefer range-based loop for iterating over a container <sup>1</sup>
- Some compilers (e.g. clang) use assertion for optimization purposes: most likely code path, not possible values, etc. <sup>2</sup>
- Not all `if` statements (or branches) are translated into jump. Small code section can be optimized in different ways <sup>3</sup> (see next slide)

---

<sup>1</sup> The Little Things: Everyday efficiencies

<sup>2</sup> Andrei Alexandrescu

<sup>3</sup> Is this a branch?

# 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**, also called *loop-invariant code motion*, consists of moving statements or expressions outside the body of a loop *without affecting the semantics* of the program

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)



Loop hoisting can be applied for avoiding redundant initialization

Base case:

```
for (int i = 0; i < 100; i++) {  
    // allocation  
    std::vector v = ...  
    ... // use "v"  
}
```

Better:

```
std::vector s(max_size);  
for (int i = 0; i < 100; i++) {  
    ...  
    v.clear();  
}
```

---

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

## Branch Hints

C++20 `[[likely]]` and `[[unlikely]]` provide a hint to the compiler to optimize a conditional statement, such as `while`, `for`, `if`

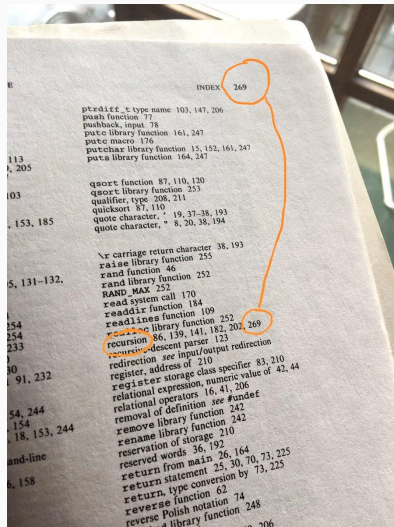
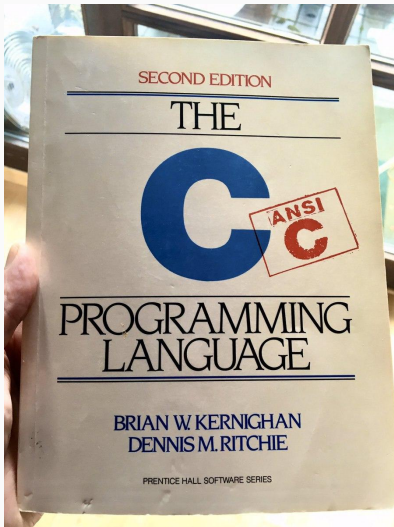
```
for (i = 0; i < 300; i++) {  
    [[unlikely]] if (rand() < 10)  
        return false;  
}
```

```
switch (value) {  
    [[likely]]   case 'A': return 2;  
    [[unlikely]] case 'B': return 4;  
}
```

**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



# 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
- The function pushes the return value on the stack
- Jump to the caller address



**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-const ref if needed) but it should be avoided for as it does not change the function signature

**by-value** Produce optimal code except for GCC (tested with GCC 9.2)

**by-reference** Could introduce a level of indirection

# Function Optimizations

- *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

### inline (internal linkage)

`inline` specifier when applied to internal linkage functions (static or anonymous namespace) is a hint for the compiler.

The code of the function can be copied where it is called (*inlining*)

```
inline void f() { ... }
```

- It is just a hint for the compiler that can ignore it ( `inline` increases the compiler heuristic threshold)
- `inline` functions increase the binary size because they are expanded in-place for every function call

## 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)

GCC/Clang extensions allow to *force* inline/non-inline functions:

```
__attribute__((always_inline)) void f() { ... }  
__attribute__((noinline))      void f() { ... }
```

- 
- 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")))`

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 can more likely 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;  
}
```

## Variable/Object Scope

**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)
- Avoid dynamic operations: **exceptions\*** (and use `noexcept`), `dynamic_cast`, **smart pointer**
- **Virtual calls** are slower than standard functions
- Mark `final` all `virtual` functions that are not overridden

---

\*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
- 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

```
struct A {    // big object
    int array[10000];
};

struct B {
    int array[10000];

    B() = default;

    B(const A& a) { // user-defined constructor
        std::copy(a.array, a.array + 10000, array);
    }
};

//-----

void f(const B& b) {}

A a;
B b;
f(b); // no cost
f(a); // very costly!! implicit conversion
```