Cpp Con 2022 : A brief tour of talks

10 septembre 2022 - 18 septembre 2022

Groupe De Lecture - 11/01/2023

STANDARD C++23

What’s new in C++23 ?

List of features by standard and compiler support :

https://en.cppreference.com/w/cpp/compiler_support

Useful tool to explore compilers and standard :

https://godbolt.org/

C++23 not yet released officially :

- Last features were added in February 2022 (?)
- Preview release in March 2022
- Final meeting of a standard is usually in February of the Year (and beginning of C++26)
- ISO paper should arrive in December 2023

CORE LANGUAGE FEATURES

#elifdef #elifndef, #warning, (#embed?) (also in C23)

```cpp
#include <iostream>
#include <type_traits>

#define USE_INT

#ifdef USE_DOUBLE
    using Type = double;
#elifndef USE_INT
    using Type = float;
#else
    using Type = int;
#endif

int main()
{
    std::cout << typeid(Type).name() << std::endl;
}
```

```cpp
const unsigned char RandomData[] =
{
    #embed </dev/urandom> limit(5)
};

int main()
{
    return 0;
}
```

Md subscript operator

```
nclude <iostream>

ruct Matrix

  Matrix()
  { data = new float[5 * 5](); }

  float* operator[](std::size_t i) { return data + i * 5; }

  float& operator[](std::size_t i, std::size_t j) { return *(data + j + i * 5); }

  ~Matrix() { delete[] data; }

  float* data;


t main()

  Matrix m;
  for (unsigned int i = 0; i < 5; i++) m[i, i] = 1;

  for (unsigned int i = 0; i < 5; i++)
  {
      for (unsigned int j = 0; j < 5; j++) std::cout << m[i, j] << " ";
      std::cout << "\n";
  }
```

Explicit object parameter (deducing this)

Quadruplication

```
template <typename T>
class optional {
  // ...
  constexpr T& value() & {
    if (has_value()) {
      return this->m_value;
    }
    throw bad_optional_access();
  }

  constexpr T const& value() const& {
    if (has_value()) {
      return this->m_value;
    }
    throw bad_optional_access();
  }

  constexpr T&& value() && {
    if (has_value()) {
      return move(this->m_value);
    }
    throw bad_optional_access();
  }

  constexpr T const&&
  value() const&& {
    if (has_value()) {
      return move(this->m_value);
    }
    throw bad_optional_access();
  }
  // ...
};
```

Delegation to 4th

```
template <typename T>
class optional {
  // ...
  constexpr T& value() & {
    return const_cast<T&>(
      static_cast<optional const&>(
        *this).value());
  }

  constexpr T const& value() const& {
    if (has_value()) {
      return this->m_value;
    }
    throw bad_optional_access();
  }

  constexpr T&& value() && {
    return const_cast<T&&>(
      static_cast<optional const&>(
        *this).value());
  }

  constexpr T const&&
  value() const&& {
    return static_cast<T const&&>(
      value());
  }
  // ...
};
```

Delegation to helper

```
template <typename T>
class optional {
  // ...
  constexpr T& value() & {
    return value_impl(*this);
  }

  constexpr T const& value() const& {
    return value_impl(*this);
  }

  constexpr T&& value() && {
    return value_impl(move(*this));
  }

  constexpr T const&&
  value() const&& {
    return value_impl(move(*this));
  }

private:
  template <typename Opt>
  static decltype(auto)
  value_impl(Opt&& opt) {
    if (!opt.has_value()) {
      throw bad_optional_access();
    }
    return forward<Opt>(opt).m_value;
  }
  // ...
};
```

Problem :

- Multiple overloads of the same function depending on const-ness of the object.
- Especially used for member access context
- Same code, it’s just marking the function const…

Explicit object parameter (deducing this)

C++17

```
class TextBlock {
public:
  char const& operator[](size_t position) const {
    // ...
    return text[position];
  }

  char& operator[](size_t position) {
    return const_cast<char&>(
      static_cast<TextBlock const&>
        (this)[position]
    );
  }
  // ...
};
```

Proposed

```
class TextBlock {
public:
  template <typename Self>
  auto& operator[](this Self&& self, size_t position) {
    // ...
    return self.text[position];
  }
  // ...
};
```

Python/Rust like syntax. Requires use of auto return type + template for deducing calling context.

Currently only (partially) supported by MSVC

Explicit object parameter (deducing this)

As lambdas are closures types (ie anonymous and can not be named objects), this can be applied to build recursive lambdas :

```cpp
auto gcd = [](this auto self, int a, int b) -> int {
    return b == 0 ? a : self(b, a % b);
}
std::cout << gcd(20, 30) << std::endl;
```

Assume, std::unreachable

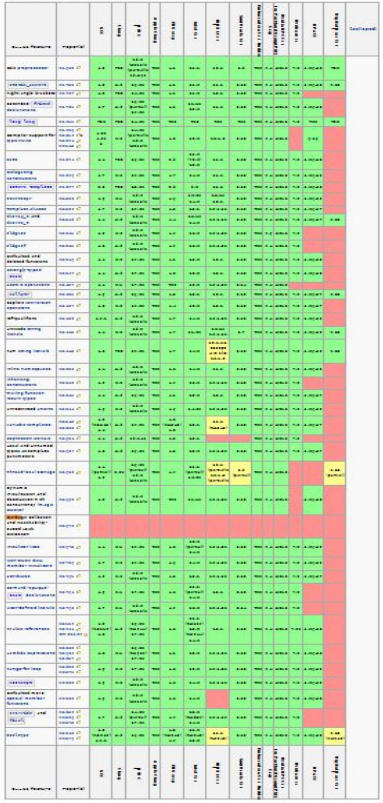
```cpp
void f(int& x, int y)
{
    void g(int);
    void h();

    [[assume(x > 0)]]; // Compiler may assume x is positive

    g(x / 2); // More efficient code possibly generated

    x = 3;
    int z = x;

    [[assume((h(), x == z))]]; // Compiler may assume x would have the same value after
                               // calling h
                               // The assumption does not cause a call to h

    h();
    g(x); // Compiler may replace this with g(3);

    h();
    g(x); // Compiler may NOT replace this with g(3);
          // An assumption applies only at the point where it appears

    [[assume((g(z), true))]]; // Compiler may assume g(z) will return
    g(z);

    [[assume(y == -10)]]; // Undefined behavior if y != -10 at this point

    [[assume((x - 1) * 3 == 12)]];

    g(x); // Compiler may replace this with g(5);
}
```

Indicates the compiler some preconditions on the code. Can help generate better code. Same idea as [[likely]] for if statements.

```cpp
#include <vector>
#include <cassert>
#include <cstddef>
#include <cstdint>
#include <utility>

struct Color { std::uint8_t r, g, b, a; };

// Assume that only restricted set of texture caps is supported.
void generate_texture(std::vector<Color>& tex, std::size_t xy)
{
    switch (xy) {
    case 128: [[fallthrough]];
    case 256: [[fallthrough]];
    case 512: /* ... */
        tex.clear();
        tex.resize(xy * xy, Color{0, 0, 0, 0});
        break;
    default:
        std::unreachable();
    }
}

int main()
{
    std::vector<Color> tex;
    generate_texture(tex, 128); // OK
    assert(tex.size() == 128 * 128);
    generate_texture(tex, 32);  // Results in undefined behavior
}
```

Removed features

C++11's garbage collector



LIBRARY ADDITIONS

<stacktrace>

```cpp
#include <stacktrace>
#include <string>
#include <iostream>

struct S { S() {
    auto trace = std::stacktrace::current();
    std::cout << std::to_string(trace) << '\n';
}};

void func2() { []() { S{}; }(); }
void func() { func2(); }

int main()
{
    func();
}
```

x86-64 gcc 12.2 -std=c++2b -lstdc++_libbacktrace

```
Program returned: 0
Program stdout
   0# S::S() at /app/example.cpp:6
   1#      at /app/example.cpp:10
   2# func2() at /app/example.cpp:10
   3# func() at /app/example.cpp:11
   4#      at /app/example.cpp:15
   5#      at :0
   6#      at :0
   7#
```

- Standards requires the stacktrace is per threads ! Not clear if includes master thread “representing approximately the initial function of the current thread of execution”
- The standards does not requires it to be exact however (some calls might be skipped).
- Might fail !

import std; import std.compat;

- Global module for standard library + global namespace function.
- Refresher :
  - Module are the new way to include / import code from multiple files
  - Compiler are required to convert #include to import for standard library headers

  - Avoid multiple compilation of header source code (copied & compiled for each TU…)
  - Kind of equivalent to precompiled headers.

  - Disable exporting macros, namespaces, …
  - Disable import forwarding by default

More on syntax and usage : https://learn.microsoft.com/en-us/cpp/cpp/modules-cpp?view=msvc-170

coroutines, views, ranges, …

Way too much additions…

- std::mdspan : non owning multidimensional view
- std::generator

- **std::ranges::to**
- std::ranges::iota
- std::ranges::shift_left, std::ranges::shift_right
- std::ranges::contains
- std::ranges::find_last, std::ranges::find_last_if, std::ranges::find_last_if_not
- **std::ranges::fold** (its reduce !!!)
- std::ranges::views::stride, std::ranges::stride_view
- **std::ranges::views::join_with**
- **std::ranges::views::chunk**, **std::views::chunk_by**
- std::ranges::views::repeat
- std::ranges::views::cartesian_product

Formatting range, std::print, std::println

```cpp
int main()
{
    std::vector<std::pair<int, int>> v { {1, 2}, {3, 4} };
    std::println("{}", v); // [(1, 2), (3, 4)]

    std::set<std::pair<int, int>> s { {1, 2}, {3, 4} };
    std::println("{}", s); // {(1, 2), (3, 4)}

    std::map<int, int> m { {1, 2}, {3, 4} };
    std::println("{}", m); // {1: 2, 3: 4};
}
```

Misc

- constexpr support for cmath and cstdlib
- auto(x), auto{x} for decay copy (obtain prvalue with auto)
- static operator ()
- flat_map, flat_set (map and set backed by std::vector)
- std::string::contains
- if consteval
- aliases (using) in init statements (for and if)
- Stable implicit move (copy-ellision)

CPP CON TALKS

What is CppCon

- CppCon is a week-long C++ conference held each year in the US (and online since Covid)
- Organized by the C++ foundation
- It is a week of talk about C++:
  - People presenting libraries, tools and deployments
  - Back-to-basics (for old and new features)
  - Best practice / patterns / commons idioms / benchmarks
- Website : https://cppcon.org/
- Github : https://github.com/CppCon
- YouTube : https://www.youtube.com/@CppCon

C++ mythbuster

Little quiz :

- Does std::move move ?

Victor Ciura : https://www.youtube.com/watch?v=bcl33-lIC70

C++ mythbuster

Little quiz :

- Does std::move move ? => No, it just changes the type, there is no guarantee for moving the object
- Does std::forward forward ?

Victor Ciura : https://www.youtube.com/watch?v=bcl33-lIC70

C++ mythbuster

Little quiz :

- Does std::move move ? => No, it just changes the ~~type~~, there is no guarantee for moving the object
- Does std::forward forward ?

Victor Ciura : https://www.youtube.com/watch?v=bcl33-lIC70

C++ mythbuster

Little quiz :

- Does std::move move ? => No, it just changes the semantic (value category), there is no guarantee for moving the object.
- Does std::forward forward ?

Victor Ciura : https://www.youtube.com/watch?v=bcl33-lIC70

C++ mythbuster

Little quiz :

- Does std::move move ? => No, it just changes the semantic, there is no guarantee for moving the object
- Does std::forward forward ? => No, it is just a conditional move
- Does std::remove remove ?

Victor Ciura : https://www.youtube.com/watch?v=bcl33-lIC70

C++ mythbuster

Little quiz :

- Does std::move move ? => No, it just changes the semantic, there is no guarantee for moving the object
- Does std::forward forward ? => No, it is just a conditional move
- Does std::remove remove ? => No, it just swaps elements
- Does std::unique makes element unique ? => No, it just swaps elements
- Is std::function a function ? => No, it is an object

Victor Ciura : https://www.youtube.com/watch?v=bcl33-lIC70

C++ mythbuster

Should everything be const-ref ? Yes and no

```
template<typename T = std::string>
struct ThreeStrings
{
    T a, b, c;

    ThreeStrings(const T& a, const T& b, const T& c);
    ThreeStrings(T&& a, T&& b, T&& c);

    ThreeStrings(T&& a, const T& b, const T& c);
    ThreeStrings(T&& a, T&& b, const T& c);
    ThreeStrings(T&& a, const T& b, T&& c);

    ThreeStrings(const T& a, T&& b, const T& c);
    ThreeStrings(const T& a, T&& b, T&& c);

    ThreeStrings(const T& a, const T& b, T&& c);
};
```

When done correctly, one would need an overload for const& and && for each parameter of a function.

- Combinatorial problem, adding a parameter doubles the amount of implementation
- All code are the same yet different (use of std::move, std::forward, …)
- Hard to maintain

While in <u>some cases</u> it may cost an additional move per parameter, it is better to take parameters by copy and moves them afterward.

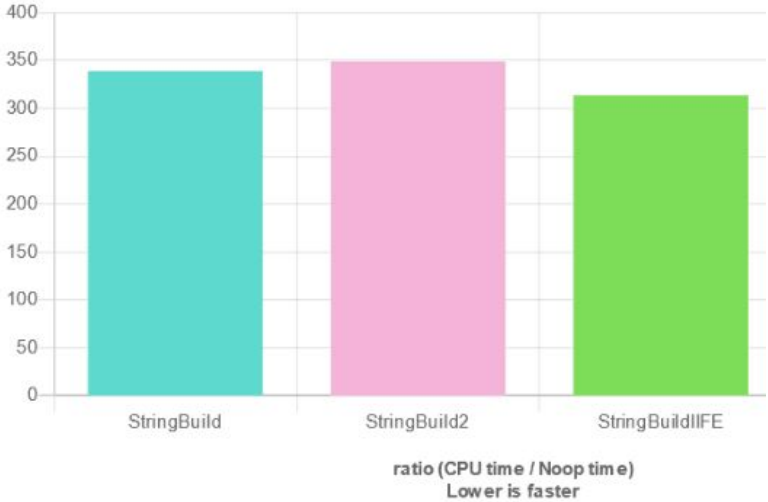
Victor Ciura : https://www.youtube.com/watch?v=bcl33-lIC70

Lambda idioms

Immediately Invoked Function Expression (IIFE):

```
1   #include <iostream>
2
3   int main()
4   {
5       bool with_world = true;
6       const std::string hello = [=]()
7       {
8           if (with_world) return "Hello World !";
9           return "Hello";
10      }();
11      std::cout << hello << std::endl;
12
13      static auto _ = []
14      {
15          std::cout << "Only called once and thread";
16          std::cout << " because variable is static";
17          return 0;
18      }();
19  }
```

- Useful for complex const initialization
- Useful for thread safe called once
- Compiler is generally very very good at generating code with this idiom !!!



Timur Doumler : https://www.youtube.com/watch?v=xBAduq0RGes

The Hidden Performance Price of C++ Virtual Functions

TLDR :

- In themselves, virtual function overhead is not high, it is the calling context that costs
  - For short and fast function: + 20% overhead
  - For long and short functions: +0.1% overhead
  - For array of pointers : less than 0.1% overhead
  - If it is possible to sort array by type : x2 speedup on virtual function calls

- What cost a lot however :
  - Virtual function cannot be inlined !!!

To avoid virtual function “Using Modern C++ to Eliminate Virtual Functions” (Jonathan Gopel) :
https://www.youtube.com/watch?v=gTNJXVmuRRA

Ivica Bogosavljevic : https://www.youtube.com/watch?v=n6PvvE_tEPk

Optimizing Binary Search

std::lower_bound on gcc :

```
template <class _Compare, class _ForwardIterator, class _Tp> _LIBCPP_CONSTEXPR_AFTER_CXX17 _ForwardIterator
__lower_bound(_ForwardIterator __first, _ForwardIterator __last, const _Tp& __value_, _Compare __comp)
{
    typedef typename iterator_traits<_ForwardIterator>::difference_type difference_type;
    difference_type __len = _VSTD::distance(__first, __last);
    while (__len != 0)
    {
        difference_type __l2 = _VSTD::__half_positive(__len);
        _ForwardIterator __m = __first;
        _VSTD::advance(__m, __l2);
        if (__comp(*__m, __value_))
        {
            __first = ++__m;
            __len -= __l2 + 1;
        }
        else
            __len = __l2;
    }
    return __first;
}
```

Sergei Slotin : https://www.youtube.com/watch?v=1RIPMQQRBWk (dispo le 17/01)

Optimizing Binary Search

```
int lower_bound(int x) {
    int l = 0, r = n - 1;
    while (l < r) {
        int m = (l + r) / 2;
        if (t[m] >= x)
            r = m;
        else
            l = m + 1;
    }
    return t[l];
}
```

std::lower_bound example implementation.

We can do better :

Sergei Slotin : https://www.youtube.com/watch?v=1RIPMQQRBWk (dispo le 17/01)

Optimizing Binary Search

```
int lower_bound(int x) {
    int l = 0, r = n - 1;
    while (l < r) {
        int m = (l + r) / 2;
        if (t[m] >= x)
            r = m;
        else
            l = m + 1;
    }
    return t[l];
}
```

std::lower_bound example implementation.

We can do better :

- No if
- Better cache locality

Sergei Slotin : https://www.youtube.com/watch?v=1RIPMQQRBWk (dispo le 17/01)

Optimizing Binary Search

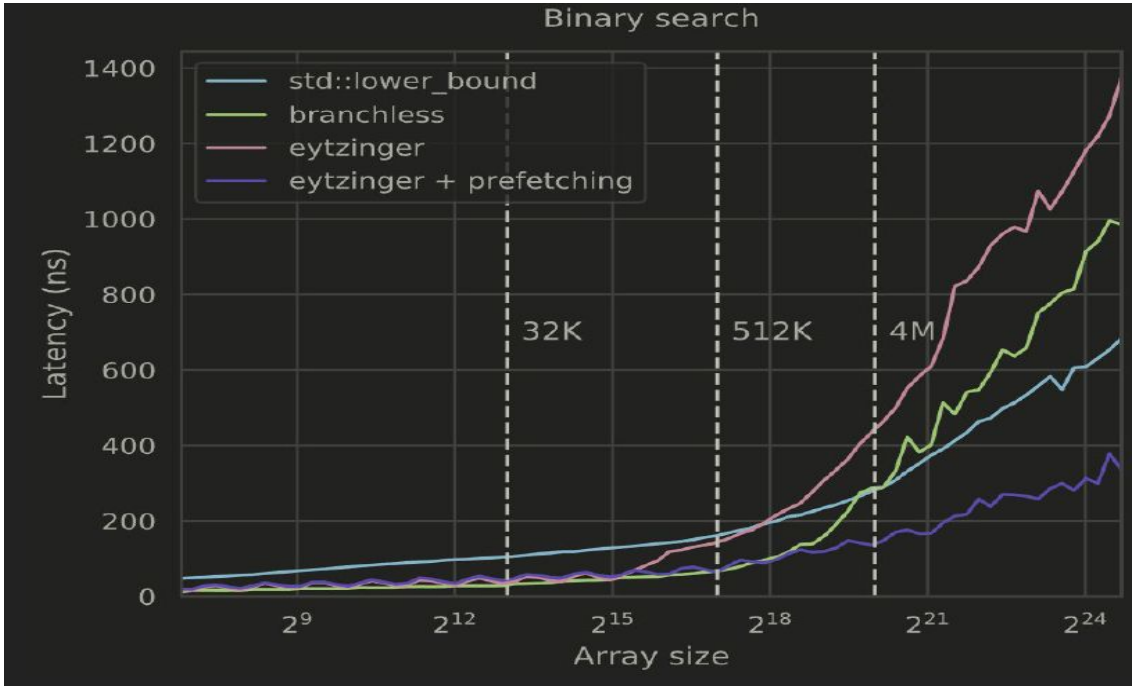
```
int lower_bound(int x) {
    int k = 1;
    while (k <= n) {
        __builtin_prefetch(&t[k * 16]);
        k = 2 * k + (t[k] < x);
    }
    k >>= __builtin_ffs(~k);
    return t[k];
}
```

More optimized version :

- Eytzinger order required
  - Storing research tree as array
  - Children at 2i and 2i+ 1
- No branch (add a boolean)
- Prefetching for faster cache usage

Sergei Slotin : https://www.youtube.com/watch?v=1RIPMQQRBWk (dispo le 17/01)

Optimizing Binary Search



Results :

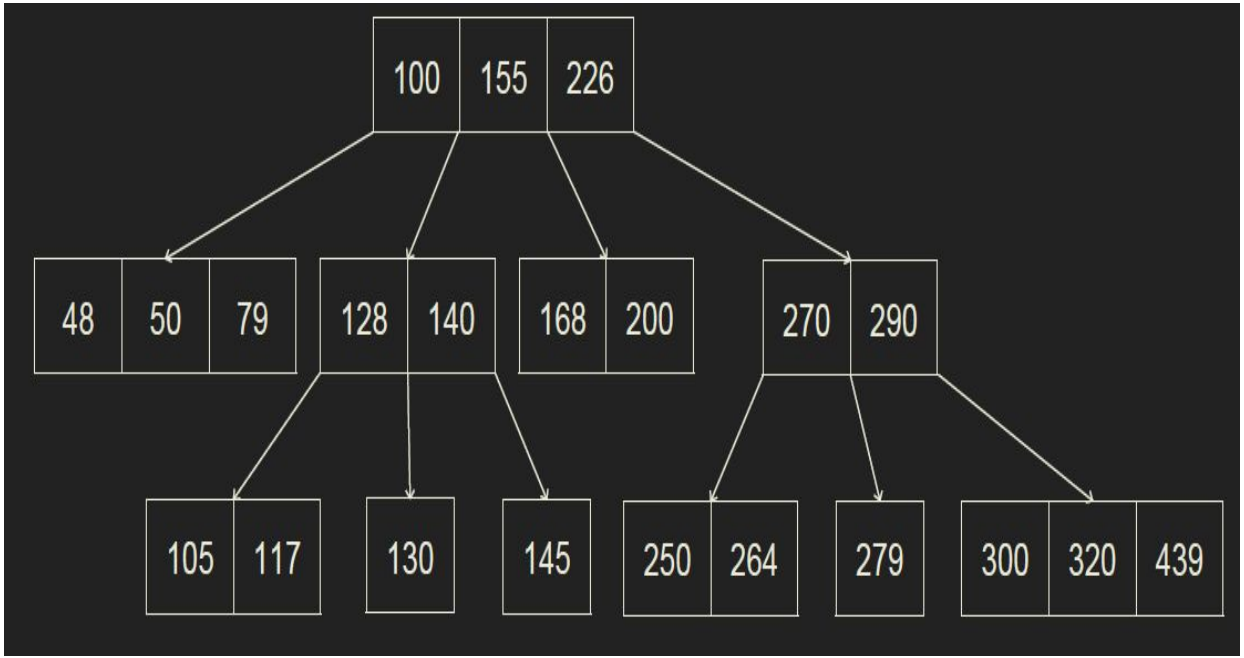
- Around 2 to 3 times faster than std::lower_bound

Can we do even better ???

- Yes ! With SIMD !

Sergei Slotin : https://www.youtube.com/watch?v=1RIPMQQRBWk (dispo le 17/01)

Optimizing Binary Search



B-Trees :

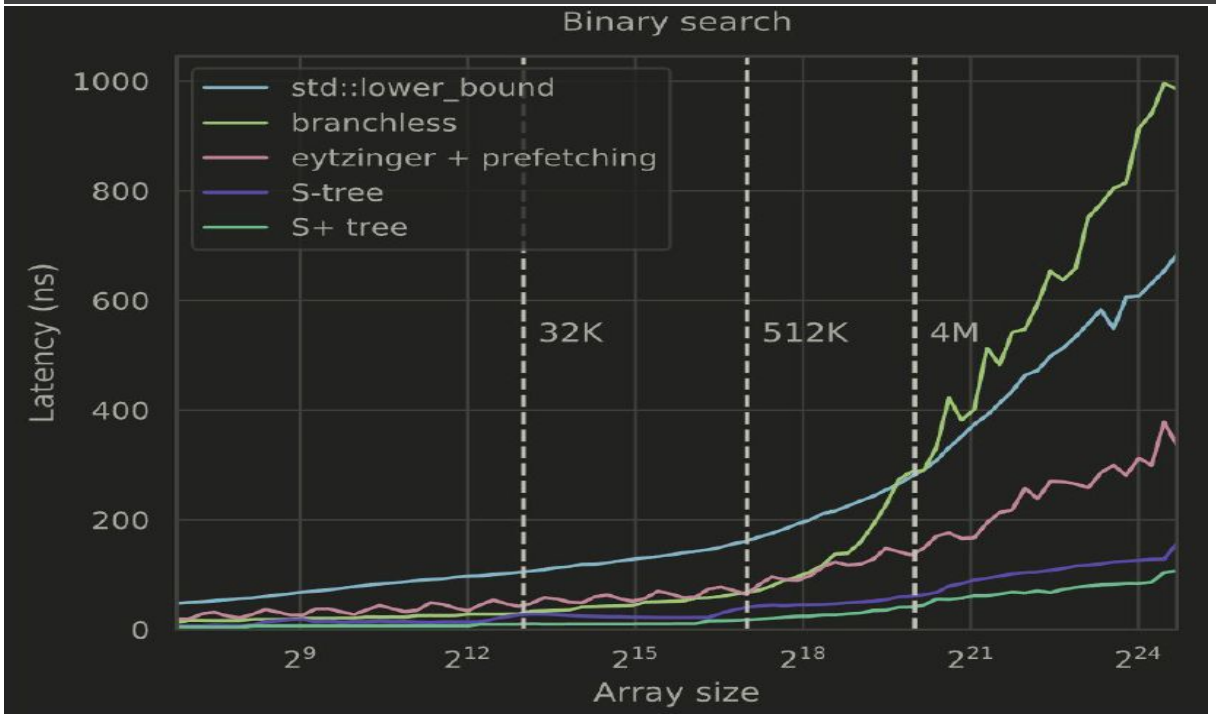
- Each node has multiple values
- Children are between values of parents

How does it works :

- Perform SIMD comparaison at first layer
- By finding the 1 in the boolean result deduce where to go.
- Repeat until reached

Sergei Slotin : https://www.youtube.com/watch?v=1RIPMQQRBWk (dispo le 17/01)

Optimizing Binary Search



Final results :

- Using a more advanced version (which is even more cache friendly)
- We get a 7 to 15 speedup compared to std::lower_bound

Sergei Slotin : https://www.youtube.com/watch?v=1RIPMQQRBWk (dispo le 17/01)

Optimization todo list

The whole talk is important !!! The summary :

At build time :

- -O3, **-O2**, -Os
- -march=native -mtune=native
- -ffast-math (/!\ Less precision)
- -fno-exceptions, -fno-rtti
- **-flto**
- -DCMAKE_UNITY_BUILD=ON
- Static linking when possible
- -fprofile-generate / -fprofile-use

C++ side:

- **const / constexpr / consteval**
- attributes (likely, assume, noexcept)
- **__restrict**
- Pure function ([[gnu::pure]])
- Take parameters properly (see talk)
- /!\ auto does not take const or ref !
- **No allocation in loops**

Hardware side:

- Data locality (cache)
- Temporal locality (pin threads)
- Avoid false sharing !
- **Branch prediction (branchless code, no indirect calls, …)**

Jan Bielak : https://www.youtube.com/watch?v=qCjEN5XRzHc

LLVM Optimization Remarks

Very technical talk. It uses ‘https://github.com/OfekShilon/optview2’, this is a tool that tells you ‘what optimizations the compiler considered, and why it failed’.

Aliasing : modifying a may modify b…

Influence of __restrict:

```
1  void foo(int* a, const int& b) {
2      for (int i=0; i<10; i++) {
3          a[i] += b;
4      }
5  }
```

```
1   foo(int*, int const&):
2         movl    (%rsi), %eax
3         addl    %eax, (%rdi)
4         movl    (%rsi), %eax
5         addl    %eax, 4(%rdi)
6         movl    (%rsi), %eax
7         addl    %eax, 8(%rdi)
8         movl    (%rsi), %eax
9         addl    %eax, 12(%rdi)
10        movl    (%rsi), %eax
11        addl    %eax, 16(%rdi)
12        movl    (%rsi), %eax
13        addl    %eax, 20(%rdi)
14        movl    (%rsi), %eax
15        addl    %eax, 24(%rdi)
16        movl    (%rsi), %eax
17        addl    %eax, 28(%rdi)
18        movl    (%rsi), %eax
19        addl    %eax, 32(%rdi)
```

Ofek Shilon : https://www.youtube.com/watch?v=qmEsx4MbKoc

LLVM Optimization Remarks

Influence of __restrict:

```
1  void foo(int* __restrict__ a,
2           const int& b) {
3      for (int i=0; i<10; i++) {
4          a[i] += b;
5      }
6  }
```

Fewer instruction : SIMD gain !

```
1   foo(int*, int const&):
2         movl    (%rsi), %eax
3         movd    %eax, %xmm0
4         pshufd  $0, %xmm0, %xmm0
5         movdqu  (%rdi), %xmm1
6         movdqu  16(%rdi), %xmm2
7         paddd   %xmm0, %xmm1
8         movdqu  %xmm1, (%rdi)
9         paddd   %xmm0, %xmm2
10        movdqu  %xmm2, 16(%rdi)
11        addl    %eax, 32(%rdi)
12        addl    %eax, 36(%rdi)
13        retq
```

Ofek Shilon : https://www.youtube.com/watch?v=qmEsx4MbKoc

LLVM Optimization Remarks

Influence of marking function pure:

```
void somefunc(const int&);
int whateva();

void f(int i, int* res) {
    somefunc(i);
    i++;
    res[0] = whateva();
    i++;
    res[1] = whateva();
    i++;
    res[2] = whateva();
}
```

i not used: but still need to update somefunc may have stored the reference in global variable used by whateva !

```
1   f(int, int*):
2       pushq   %rbx
3       subq    $16, %rsp
4       movq    %rsi, %rbx
5       movl    %edi, 12(%rsp)
6       leaq    12(%rsp), %rdi
7       callq   somefunc(int const&)
8       incl    12(%rsp)
9       callq   whateva()@PLT
10      movl    %eax, (%rbx)
11      incl    12(%rsp)
12      callq   whateva()@PLT
13      movl    %eax, 4(%rbx)
14      incl    12(%rsp)
15      callq   whateva()@PLT
16      movl    %eax, 8(%rbx)
17      addq    $16, %rsp
18      popq    %rbx
19      retq
```

Ofek Shilon : https://www.youtube.com/watch?v=qmEsx4MbKoc

LLVM Optimization Remarks

Influence of marking function pure:

```
void somefunc(const int&) __attribute__((pure));
int whateva();

void f(int i, int* res) {
    somefunc(i);
    i++;
    res[0] = whateva();
    i++;
    res[1] = whateva();
    i++;
    res[2] = whateva();
}
```

Also note : pure function that returns void => call skipped !

```
1   f(int, int*):
2           pushq   %rbx
3           movq    %rsi, %rbx
4           callq   whateva()@PLT
5           movl    %eax, (%rbx)
6           callq   whateva()@PLT
7           movl    %eax, 4(%rbx)
8           callq   whateva()@PLT
9           movl    %eax, 8(%rbx)
10          popq    %rbx
11          retq
```

Ofek Shilon : https://www.youtube.com/watch?v=qmEsx4MbKoc

One-line summary

C20-likely attribute optimizations pessimizations and unlikely consequences (Amir Kirsh & Tomer Vromen):
Between -10% and 10% improvement, should only be used when you know for sure.

Graph Algorithms and Data structures in C++20 (Phil Ratzloff & Andrew Lumsdaine)
C++20 implementation for graphs structures and algorithm (towards std::graph)

Understanding Allocator Impact on Runtime Performance (Parsa Amini)
Use std::pmr to improve runtime and memory loads !

Fast, High-Quality Pseudo-Random Numbers for Non-Crpyptographers in C++ (Roth Michaels)
Do not use <random>, use PCG, or CPRGN !

I don't remember (sorry for the author):
Use standard library more ! (ex: std::inner_product)

To be coming :
What Is an Image, Anyway ? (13/01/2023): https://www.youtube.com/watch?v=zi57OkPwzbk

The end

En france :

- CppFrug (Paris)
- Développeurs C++ à Montpellier (Montpellier)
- Qt Meetup (Montpellier)
- Nantes C++ Meetup (Nantes)
- Toulouse ?