Miscellaneous

Pointer Tagging on x86-64 (1)

Virtual addresses are translated to physical addresses by the MMU

- Virtual addresses are 64-bit integers on x86-64
- On x86-64, only the lower 48 bit of pointers are actually used
- The upper 16 bit of pointers are usually required to be zero

The upper 16 bit of each pointer can be used to store useful information

- Usually called *pointer tagging*
- Tagged pointers require careful treatment to avoid memory bugs
- If portability is desired, an implementation that works without pointer tagging has to be provided (e.g. through preprocessor defines)
- Allows us to modify two values (16 bit tag and 48 bit pointer) with a single atomic instruction

Pointer Tagging on x86-64 (2)

We can store different things in the upper 16 bit of pointers

- Up to 16 binary flags
- A single 16 bit integer
- ...

Guidelines

- Always wrap tagged pointers within a suitable data structure
- Do not expose tagged pointers in raw form
- Store tagged pointers as `uintptr_t` internally
- Use bit operations to access tag and pointer parts

Pointer Tagging on x86-64 (3)

Using the upper 16 bit to store information

```
static constexpr uint64_t shift = 48;
static constexpr uintptr_t mask = (1ull << shift) - 1;
//-----------------------------------------------------------------------
uintptr_t tagPointer(void* ptr, uint64_t tag)
// Tag a pointer. Discards the upper 48 bit of tag.
{
    return (reinterpret_cast<uintptr_t>(ptr) & mask) | (tag << shift);
}
//-----------------------------------------------------------------------
uint64_t getTag(uintptr_t taggedPtr)
// Get the tag stored in a tagged pointer
{
    return taggedPtr >> shift;
}
//-----------------------------------------------------------------------
void* getPointer(uintptr_t taggedPtr)
// Get the pointer stored in a tagged pointer
{
    return reinterpret_cast<void*>(taggedPtr & mask);
}
```

Pointer Tagging on x86-64 (4)

Using the lower 16 bit to store information

```
static constexpr uint64_t shift = 16;
static constexpr uintptr_t mask = (1ull << shift) - 1;
//--------------------------------------------------------------
uintptr_t tagPointer(void* ptr, uint64_t tag)
// Tag a pointer. Discards the upper 48 bit of tag.
{
    return (reinterpret_cast<uintptr_t>(ptr) << shift) | (tag & mask);
}
//--------------------------------------------------------------
uint64_t getTag(uintptr_t taggedPtr)
// Get the tag stored in a tagged pointer
{
    return taggedPtr & mask;
}
//--------------------------------------------------------------
void* getPointer(uintptr_t taggedPtr)
// Get the pointer stored in a tagged pointer
{
    return reinterpret_cast<void*>(taggedPtr >> shift);
}
```

Vectorization

Most modern CPUs contain vector units that can exploit data-level parallelism

- Apply the same operation (e.g. addition) to multiple data elements in a single instruction
- Can greatly improve the performance of suitable algorithms (e.g. image processing)
- Not all algorithms are amenable to vectorization

Overview

- Can be used through extensions to the x86 instruction set architecture
- Commonly referred to as single instruction, multiple data (SIMD) instructions
- Can be used in C/C++ code through *intrinsic* functions
- The Intel Intrinsics Guide provides an excellent documentation

SIMD Extensions

SIMD extensions have evolved substantially over time

- MMX
- SSE, SSE2, SSE3, SSE4
- AVX, FMA, AVX2, AVX-512

Modern CPUs retain backward compatibility with older instruction set extensions

- The CPU flags exposed in `/proc/cpuinfo` indicate which extensions are supported
- We will briefly introduce AVX (`avx` flag in `/proc/cpuinfo`)
- AVX should be supported on most reasonably modern CPUs

AVX Data Types

AVX data types and intrinsics are defined in the `<immintrin.h>` header

- AVX adds 16 registers which are 256 bits wide each
- Can hold multiple data elements
- Can be used through special opaque data types

AVX data types

- `__m256`: Can hold eight 32 bit floating point values
- `__m256d`: Can hold four 64 bit floating point values
- `__m256i`: Can hold thirty-two 8 bit, sixteen 16 bit, eight 32 bit or four 64 bit integer values
- Commonly referred to as *vectors* (not to be confused with `std::vector`)

Other SIMD extensions follow similar naming conventions for data types

AVX Intrinsics

Usually, there are separate intrinsics for each data type

- AVX intrinsics usually begin with `_mm256`
- Next is a name for the instruction (e.g. `loadu`)
- Finally, the data type is indicated
  - `ps` for `__m256`
  - `pd` for `__m256d`
  - `si256` for `__m256i`
- Example: `_mm256_loadu_ps`

We will only show intrinsics for `__m256` in the following

- Intrinsics for other data types usually follow similar patterns
- Exception: AVX does not contain many arithmetic operations on integer types (added in AVX2)

Constant Values

We cannot directly modify individual data elements in AVX data types

- We have to use intrinsics for that purpose
- Intrinsics usually return the result of a modification

We can create constant vectors

- `__m256 _mm256_set1_ps(float a)`
  - Returns a vector with all elements equal to a
- `__m256 _mm256_set_ps(float e7, ..., float e0)`
  - Returns a vector with the elements `e0, ..., e7`
- `__m256 _mm256_setr_ps(float e0, ..., float e7)`
  - Returns a vector with the elements `e0, ..., e7`

Loading and Storing

Loading data from memory

- `__m256 _mm256_load_ps(const float* addr)`
  - Load eight 32 bit floating point values from memory starting at addr
  - addr has to be aligned to a 32 byte boundary
- `__m256 _mm256_loadu_ps(const float* addr)`
  - Load eight 32 bit floating point values from memory starting at addr
  - addr does not have to be aligned beyond usual `float` alignment

Storing data to memory

- `void _mm256_store_ps(float* addr, __m256 a)`
  - Store eight 32 bit floating point values in a to memory starting at addr
  - addr has to be aligned to a 32 byte boundary
- `void _mm256_storeu_ps(float* addr, __m256 a)`
  - Store eight 32 bit floating point values in a to memory starting at addr
  - addr does not have to be aligned beyond usual `float` alignment

Arithmetic Operations

AVX provides many arithmetic operations on vectors

- All the usual arithmetic operations
- Bitwise operations on integer types
- ...

Example: Adding vectors

- `__m256 _mm256_add_ps(__m256 a, __m256 b)`
  - Adds the individual elements of the vectors `a` and `b`
  - Returns the result of the addition

Example

Computing the sum of elements in an `std::vector`

```
#include <immintrin.h>
#include <vector>
//----------------------------------------------------------------------
float fastSum(const std::vector<float>& vec) {
    __m256 vectorSum = _mm256_set1_ps(0);
    uint64_t index;
    for (index = 0; (index + 8) <= vec.size(); index += 8) {
        __m256 data = _mm256_loadu_ps(&vec[index]);
        vectorSum = _mm256_add_ps(vectorSum, data);
    }

    float sum = 0;
    float buffer[8];
    _mm256_storeu_ps(buffer, vectorSum);
    for (unsigned i = 0; i < 8; ++i)
        sum += buffer[i];
    for (; index < vec.size(); ++index)
        sum += vec[index];

    return sum;
}
```

Further Operations

AVX contains many more instructions

- Comparison operations on vectors
- Masked operations

Allows vectorization of many algorithms

- Vectorization is not guaranteed to improve performance
- Generally, compute-heavy algorithms benefit greatly from vectorization
- Algorithms with a lot of fine-grained branching or many loads and stores may not benefit
- Vectorization is always an *optimization* that should not be applied prematurely

The C++20 Standard

C++20 is a major planned update of the C++ standard

- The ISO C++ committee has recently announced C++20 to be feature complete
- Nevertheless still a work-in-progress draft (pending review)
- Adds some very cool features to the C++ standard

Compiler support is still intermittent

- Most compilers already support at least some C++20 features
- GCC 8/9 in particular supports quite a few C++20 features

Constraints and Concepts (1)

We have outlined previously how templates act similar to duck typing

- Any type can be specified as an argument for a template type parameter
- Compilation will fail if type does not satisfy some *implicit* requirements
- Compiler does not know about these implicit requirements
- Compilation errors can only refer to specific cause of compilation failure (e.g. an ill-formed expression)

Constraints and concepts *explicitly* specify requirements on template parameters

- Allows the compiler to check requirements
- Allows the compiler to generate much more informative error messages
- Greatly improves safety (e.g. *explicit* concepts instead of *implicit* named requirements in the standard library)

Constraints and Concepts (2)

Sorting a range in C++17

Sort.hpp

```
#pragma once
//--------------------------------------------------
#include <utility>
//--------------------------------------------------
template <typename T>
void swap(T& a, T& b) {
    T tmp(std::move(a));
    a = std::move(b);
    b = std::move(tmp);
}
//--------------------------------------------------
template <typename T>
void sort(T* begin, T* end) {
    if (begin == end) return;

    for (T* i = begin; i != end; ++i)
        for (T* j = (i + 1); j != end; ++j)
            if (*i > *j) swap(*i, *j);
}
```

Constraints and Concepts (3)

We can easily break the code from the previous example

main.cpp

```
#include "Sort.hpp"
#include <vector>
//-----------------------------------------------------
struct Foo {
    unsigned value;

    Foo(unsigned value) : value(value) {}
    Foo(Foo&&) = delete;
    Foo& operator=(Foo&&) = delete;
};
//-----------------------------------------------------
int main() {
    Foo v[] = {3, 6, 2, 1, 4, 8, 7};
    sort(&v[0], &v[7]);
}
```

Constraints and Concepts (4)

Why does the Foo struct break our code

- We implicitly required that T is move-constructible
- We implicitly required that T is move-assignable
- We implicitly required that T implements operator>

Initial compile error by GCC 8

```
> g++-9 -std=c++17 -o main main.cpp
In file included from main.cpp:1:
Sort.hpp: In instantiation of ‘void sort(T*, T*) [with T = Foo]’:
main.cpp:15:21:   required from here
Sort.hpp:41:17: error: no match for ‘operator>’ (operand types are ‘Foo’ and
↪  ‘Foo’)
   41 |         if (*i > *j)
      |             ~~~^~~~
```

Constraints and Concepts (5)

We are not done once we implement operator> for Foo

```
> g++ -std=c++17 -o main main.cpp
In file included from main.cpp:1:
Sort.hpp: In instantiation of ‘void swap(T&, T&) [with T = Foo]’:
Sort.hpp:42:17:   required from ‘void sort(T*, T*) [with T = Foo]’
main.cpp:15:21:   required from here
Sort.hpp:28:6: error: use of deleted function ‘Foo::Foo(Foo&&)’
   28 |     T tmp(std::move(a));
      |       ^~~
main.cpp:7:4: note: declared here
    7 |     Foo(Foo&&) = delete;
      |     ^~~
In file included from main.cpp:1:
Sort.hpp:29:6: error: use of deleted function ‘Foo& Foo::operator=(Foo&&)’
   29 |     a = std::move(b);
      |     ~~^~~~~~~~~~~~~~
main.cpp:8:9: note: declared here
    8 |     Foo& operator=(Foo&&) = delete;
      |          ^~~~~~~~
In file included from main.cpp:1:
Sort.hpp:30:6: error: use of deleted function ‘Foo& Foo::operator=(Foo&&)’
   30 |     b = std::move(tmp);
      |     ~~^~~~~~~~~~~~~~~~
main.cpp:8:9: note: declared here
    8 |     Foo& operator=(Foo&&) = delete;
      |          ^~~~~~~~
```

Concepts and Constraints (6)

In C++20, we could add suitable concepts as follows

Sort.hpp

```
...
//------------------------------------------------------
template <typename T>
concept MoveConstructible = requires (T a) { T(std::move(a)); };
//------------------------------------------------------
template <typename T>
concept MoveAssignable = requires (T a, T b) { a = std::move(b); };
//------------------------------------------------------
template <typename T>
concept Comparable = requires (T a, T b) { a > b; }
//------------------------------------------------------
template <typename T>
concept Swappable = MoveConstructible<T> && MoveAssignable<T>;
//------------------------------------------------------
...
```

Concepts and Constraints (7)

Subsequently, we could impose constraints on the template parameters

Sort.hpp

```
...
//---------------------------------------------------
template <typename T> requires Swappable<T>
void swap(T& a, T& b)
// Swap two elements
{
    T tmp(std::move(a));
    a = std::move(b);
    b = std::move(tmp);
}
//---------------------------------------------------
template <typename T> requires Comparable<T> && Swappable<T>
void sort(T* begin, T* end)
// Sort a range
{
    if (begin == end) return;

    for (T* i = begin; i != end; ++i)
        for (T* j = (i + 1); j != end; ++j)
            if (*i > *j) swap(*i, *j);
}
```

Concepts and Constraints (8)

The compiler will now check that all constraints are fulfilled

```
> g++-9 -fconcepts -std=c++17 -o main main.cpp
main.cpp: In function ‘int main()’:
main.cpp:15:21: error: cannot call function ‘void sort(T*, T*) [with T = Foo]’
   15 |     sort(&v[0], &v[7]);
      |                      ^
In file included from main.cpp:1:
Sort.hpp:34:6: note:   constraints not satisfied
   34 | void sort(T* begin, T* end)
      |      ^~~~
Sort.hpp:20:9: note: within ‘template<class T> concept const bool
↪  Comparable<T> [with T = Foo]’
   20 | concept Comparable = requires (T a, T b) {
      |         ^~~~~~~~~~
Sort.hpp:20:9: note:     with ‘Foo a’
Sort.hpp:20:9: note:     with ‘Foo b’
Sort.hpp:20:9: note: the required expression ‘(a > b)’ would be ill-formed
```

Concepts and Constraints (9)

The compiler will now check that all constraints are fulfilled

```
main.cpp: In function 'int main()':
main.cpp:15:21: error: cannot call function 'void sort(T*, T*) [with T = Foo]'
   15 |    sort(&v[0], &v[7]);
      |                     ^
In file included from main.cpp:1:
Sort.hpp:34:6: note:   constraints not satisfied
   34 | void sort(T* begin, T* end)
      |      ^~~~
Sort.hpp:17:9: note: within 'template<class T> concept const bool Swappable<T> [with T =
↪  Foo]'
   17 | concept Swappable = MoveConstructible<T> && MoveAssignable<T>;
      |         ^~~~~~~~~
Sort.hpp:7:9: note: within 'template<class T> concept const bool MoveConstructible<T>
↪  [with T = Foo]'
    7 | concept MoveConstructible = requires (T a) {
      |         ^~~~~~~~~~~~~~~~~
Sort.hpp:7:9: note:     with 'Foo a'
Sort.hpp:7:9: note: the required expression '(T)(std::move(a))' would be ill-formed
Sort.hpp:12:9: note: within 'template<class T> concept const bool MoveAssignable<T> [with
↪  T = Foo]'
   12 | concept MoveAssignable = requires (T a, T b) {
      |         ^~~~~~~~~~~~~~
Sort.hpp:12:9: note:     with 'Foo a'
Sort.hpp:12:9: note:     with 'Foo b'
Sort.hpp:12:9: note: the required expression 'a = std::move(b)' would be ill-formed
```

Contracts (1)

Contracts specify preconditions, postconditions, and assertions for functions

- Precondition: `[[ expects: expression ]]`
- Postcondition: `[[ ensures: expression ]]`
- Assertion: `[[ assert: expression ]]`

Outline

- Preconditions and postconditions are function attributes
- Are evaluated immediately before evaluating the function body / immediately before returning
- Assertions can appear within a function body
- Programs can be compiled with different build levels that affect contract checking

Contracts (2)

Example (C++17)

```
void cyclicIncrease(uint64_t index, uint64_t max) {
    // preconditions
    assert(index < max);
    assert(max > 0);
    assert((max & (max - 1)) == 0);

    uint64_t result = (index + 1) & (max - 1);

    // postcondition
    assert(result < max);

    return result;
}
```

Contracts (3)

Example (C++20)

```
void cyclicIncrease(uint64_t index, uint64_t max)
    [[expects: index < max]]
    [[expects: max > 0]]
    [[expects: (max & (max - 1)) == 0]]
    [[ensures: index < max]]
{
    return (index + 1) & (max - 1);
}
```

Three-Way Comparison Operator

C++20 introduces a designated operator for three-way comparison

- Syntax: `lhs <=> rhs`
- Can be overloaded for custom types
- Default implementation provided for fundamental types

Returns an object with the following semantics

- `(a <=> b) < 0` iff `a < b`
- `(a <=> b) == 0` iff `a == b`
- `(a <=> b) > 0` iff `a > b`

`std::span` (1)

`std::span` is a straightforward extension of `std::string_view`

- Represents a contiguous sequence of zero-indexed objects
- A span can have static extent where the number of elements is encoded as a template argument
- A span can have dynamic extent where the number of elements is a member variable

Benefits

- Similar benefits as `std::string_view`
- Lightweight proxy for a range of objects
- Constant-time operations

std::span (2)

Example

```
// C++17
void foo17(unsigned* begin, unsigned* end) {
    // do something

    unsigned* mid = begin + (end - begin) / 2;
    foo(begin, mid);
    foo(mid, end);

    // do something more
}
//-----------------------------------------------------
// C++20
void foo20(std::span<unsigned> span) {
    // do something

    size_t size = span.size();
    foo(span.subspan(0, size / 2));
    foo(span.subspan(size / 2, size - (size / 2)));

    // do something more
}
```

Ranges

C++20 introduces the *range* concept

- Ranges can be seen as a generalization of the iterator concept
- Ranges support a variety of view adapters that can be chained to specify complex operations on ranges

```
#include <iostream>
#include <ranges>
#include <vector>
//--------------------------------------------------
int main() {
    std::vector<unsigned> v{0, 1, 2, 3, 4, 5, 6};
    auto even = [](unsigned i) { return (i % 2 == 0); }
    auto square = [](unsigned i) { return i * i; }

    for (auto i : v | std::view::filter(even) | std::view::transform(square))
        std::cout << i << " ";  // prints 0 4 16 36
}
```

Modules (1)

Modules help structure large amounts of code into logical parts

- A module consists of multiple translation units called *module units*
- Module units can *import* other modules
- Module units can *export* certain declarations

Facilitates encapsulation of logically independent parts

- Exported declarations are visible to name lookup in translation units that import the module
- Other declarations are not visible to name lookup

Reduces compilation overhead

- Exported definitions are compiled into easy-to-parse binary format
- No need to recursively parse transitive includes

Modules (2)

Example

greeting.cpp

```
export module greeting;

import <string>;

export std::string getGreeting() {
    return "Hello world!";
}
```

main.cpp

```
import greeting;
import <iostream>;

int main() {
    std::cout << getGreeting() << std::endl;
}
```

Coroutines

A coroutine is a function that can suspend execution to be resumed later

- Execution is suspended by returning to the caller
- Allows for sequential code that executes asynchronously
- Allows for lazily computed infinite sequences

```
generator<int> iota(int n = 0) {
    while (true)
        co_yield n++;
}
```