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

Template Metaprogramming

Templates can be used for meta-programming at compile time.

- Template specializations can be used to select different types depending on template arguments
- Recursive templates can be used for basic "control flow"
- The standard library defines several useful templates in `<type_traits>`
- All types and values are generated at compile time, so can be used as constants or template parameters

Type Traits

Type traits can be used to analyze properties of arbitrary types:

```
constexpr bool a = std::is_arithmetic_v<int>; // true
constexpr bool b = std::is_class_v<int>; // false
constexpr bool c = std::is_class_v<std::vector<int>>; // true
constexpr bool d = std::is_move_assignable_v<std::vector<int>>; // true
```

They can also be used to generate new types:

```
using T1 = std::remove_reference_t<int&>; // T1 is int
using T2 = std::add_pointer_t<int>; // T2 is int*
// T3 is const std::vector<int>&
using T3 = std::add_const_t<std::add_lvalue_reference_t<std::vector<int>>>;
// my_uintptr_t is uint64_t on systems where the size of void* is 8 bytes,
// or uint32_t otherwise.
using my_uintptr_t =
    std::conditional_t<sizeof(void*) == 8, uint64_t, uint32_t>;
```

Using Type Traits

Using type traits can prevent code duplication. Common example: const and non-const versions of an iterator.

```
template <typename T>
class Container {
    private:
    template <bool isConst>
    class Iterator {
        public:
        using reference = std::conditional_t<isConst, const T&, T&>;
        // [...]
    };

    public:
    using iterator = Iterator<false>;
    using const_iterator = Iterator<true>;
};
```

The C++20 Standard

C++20 is the latest release of the C++ standard

- Adds some very cool features to the C++ standard
- We already covered many of the well-supported new features throughout this course (e.g. concepts)
- In the following we will give an overview of additional potentially very useful features

Compiler support for these features is improving although still intermittent

- Some features (e.g. modules) are not yet implemented completely by some compilers
- Some features (e.g. coroutines) may be implemented but affected by compiler bugs
- In any case: Use the latest compiler version available to you

Implementing Concepts (1)

We can use concepts to restrict the types used by templates instead of SFINAE.

```
#include <concepts>

template <typename T> requires std::floating_point<T>
T fdiv1(T a, T b) {
    return a / b;
}

template <typename T>
T fdiv2(T a, T b) requires std::floating_point<T> {
    return a / b;
}

template <std::floating_point T>
T fdiv3(T a, T b) {
    return a / b;
}
```

Implementing Concepts (2)

Concepts can be defined by a single bool (usually computed from type traits).

```
#include <type_traits>

template <typename T>
concept signed_concept =
    std::is_integral<T>::value && std::is_signed<T>::value;

template <typename T> requires signed_concept<T>
T foo(T a) {
    return a - 1;
}

int x = 0;
foo(x); // OK, returns -1
unsigned y = 0;
foo(y); // Compile time error: constraints not satisfied
```

Implementing Concepts (3)

Concepts can be defined by a requires clause using:

- Simple requirements: Require an expression to be valid
- Type requirements: Require a type to be valid
- Nested requirements: Require nested require clauses to be satisfied
- Compound requirements: Require properties on the return type of an expression

```
template <typename Allocator>
concept IsAllocator = requires(Allocator &a) {
    a.allocate(); // Simple requirement
    typename Allocator::value_type; // Type requirement
    // Nested requirements:
    requires std::default_initializable<Allocator>;
    requires std::movable<Allocator>;
    // Compound requirements:
    { a.allocate() } -> std::same_as<typename Allocator::value_type*>;
    { a.deallocate(std::declval<typename Allocator::value_type*>()) }
        -> std::same_as<void>;
};
```

Coroutines (1)

Regular function calls are strictly nested

- A function call suspends execution of the calling function, and resumes execution at the start of the called function
- Eventually, the called function returns and execution of the calling function resumes after the function call expression

Functions have state that has to be maintained across nested function calls

- Values of any local variables
- The instruction at which to resume execution after a function call
- Strict nesting of function calls allows for highly optimized state maintenance on the stack
- Strict nesting of function calls makes implementing asynchronous operations cumbersome

Coroutines (2)

Coroutines are functions that can be suspended and resumed (almost) arbitrarily

- Suspending a coroutine transfers execution back to the caller
- Resuming a suspended coroutine continues execution at the point it was suspended
- The state of a coroutine remains alive across suspensions (e.g. local variables)

Coroutines in C++ are implemented with the help of three new keywords

- `co_await <expr>`: Suspends the coroutine and returns control to the caller
- `co_yield <expr>`: Returns a value to the caller and suspends the coroutine
- `co_return <expr>`: Returns a value to the caller and finishes the coroutine

Coroutines (3)

Unfortunately, C++ coroutines are currently quite painful to use

- There is not yet any "coroutine standard library"
- In order to actually use any of the coroutine keywords, we have to implement a lot of (boilerplate) infrastructure ourselves
- The behavior of C++ coroutines is highly configurable through the details of this infrastructure implementation
- Overall, it is quite difficult to implement working coroutines

Further complications that will (hopefully) improve over time

- Compiler bugs in the implementation of coroutines
- Suboptimal compiler error messages for coroutines
- Suboptimal debugger support for coroutines

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;
}
```

Perfect Forwarding

You can forward parameters keeping the value type with `std::forward`

```
#include <iostream>
#include <utility>

void foo(int&& n) {
    std::cout << "rvalue overload, n=" << n << "\n";
}
void foo(int& n){
    std::cout << "lvalue overload, n=" << n << "\n";
}
template <typename T>
void dispatcher(T&& n){
    foo(std::forward<T>(n));
}
dispatcher(1); // rvalue overload, n=1
int i = 2;
dispatcher(i); // lvalue overload, n=2
```

Designated Initializers

C++20 introduces *designated initializers*

- Allows explicit initialization of class members by name
- This was already possible in C and supported by many compilers
- C++20 now supports a subset of what is allowed in C

```
struct Foo {
    int a;
    int b;
};
Foo f{ .a = 1, .b = 2 };
```

Bit Manipulation

The `<bit>` header introduces several functions for bit inspection and manipulation.

- `std::bit_cast`: Inspect the object representation (instead of using `reinterpret_cast` with potential undefined behavior)
- `std::endian`: Check the endianness of the system
- `std::has_single_bit`: Check if number is power of two
- `std::bit_ceil`, `std::bit_floor`: Find the next/previous power of two
- `std::rotl`, `std::rotr`: Rotate bits
- `std::countl_zero`: Count the number of consecutive zero bits starting from the most significant bit
- `std::popcount`: Count the number of one bits
- ...

Additional atomic types

C++20 introduces an atomic specialization of `std::shared_ptr`

```
#include <atomic>
#include <memory>

struct LargeObject {
    char data[1000];
};
std::atomic<std::shared_ptr<LargeObject>> object;

void readThreadSafe() {
    auto objectPtr = object.load();
    if (objectPtr)
        objectPtr->data; /* do something with objectPtr->data */
}

void replaceThreadSafe(std::shared_ptr<LargeObject> newObject) {
    object.store(std::move(newObject));
}
```

std::format

C++20 introduces a new way to format strings: `std::format`.

```
#include <format>

std::cout << std::format("Hello {}!", "world") << std::endl;
// Prints "Hello world!"
std::cout << std::format("{} {}!", "Hello", "world", "something")
          << std::endl;
// OK, prints "Hello world!"
```

Unfortunately, still limited or no support in many compilers.

More Features

C++20 introduces further small and large features, such as:

- `std::source_location`: Stores a location in the source code. `std::source_location::current()` can be used to get the location of the current line
- `<numbers>` header: Contains mathematical constants like `std::numbers::pi` and `std::numbers::e`
- `consteval` and `constinit`: Behave like a "mandatory" `constexpr`
- More functions and classes in the standard library are `constexpr`
- Some restrictions of lambdas were removed, e.g. you can capture structural bindings
- Non-type-template arguments can have a user-defined type
- ...

The C++23 Standard

C++23 will be the next release of the C++ standard

- Will add useful features such as a coroutine library, a modularized standard library, ...

Compiler support for new features is still experimental and very limited.

- Might be not be fully implemented in a while.
- In any case: Use the latest compiler version available to you