



HIGH PERFORMANCE SCIENTIFIC COMPUTING IN C++

PATC HPC C++ course 2021

21-24 June 2021 | Sandipan Mohanty | Forschungszentrum Jülich, Germany

Revisiting programming basics for performance

Stack execution model

```

auto main() -> int
{
    auto N = 10;
    if (f(N) < g(N)) {
        h1(N);
    } else {
        h2(N);
    }
}

auto f(int i) () -> int
{
    return (i * i) %12;
}

auto g(int i) () -> int
{
    return i % 12;
}

auto h1(int i) () -> int
{
    return h11(i);
}

auto h2(int i) () -> int
{
    return h21(i);
}

auto h11(int i) () -> int
{
    return i * i;
}

auto h21(int i) () -> int
{
    return i + h211(i);
}

auto h211(int i)
-> int
{
    return -i;
}

```

```

auto main() -> int
{
    auto N = 10;
    if (f(N) < g(N)) {
        h1(N);
    } else {
        h2(N);
    }
}

auto f(int i) () -> int
{
    return (i * i) %12;
}

auto g(int i) () -> int
{
    return i % 12;
}

auto h1(int i) () -> int
{
    return h11(i);
}

auto h2(int i) () -> int
{
    return h21(i);
}

auto h11(int i) () -> int
{
    return i * i;
}

auto h21(int i) () -> int
{
    return i + h211(i);
}

auto h211(int i)
-> int
{
    return -i;
}

```

main()

```

auto main() -> int
{
    auto N = 10;
    if (f(N) < g(N)) {
        h1(N);
    } else {
        h2(N);
    }
}

auto f(int i) () -> int
{
    return (i * i) %12;
}

auto g(int i) () -> int
{
    return i % 12;
}

auto h1(int i) () -> int
{
    return h11(i);
}

auto h2(int i) () -> int
{
    return h21(i);
}

auto h11(int i) () -> int
{
    return i * i;
}

auto h21(int i) () -> int
{
    return i + h211(i);
}

auto h211(int i)
-> int
{
    return -i;
}

```

main()

f() int i=10

```

auto main() -> int
{
    auto N = 10;
    if (f(N) < g(N)) {
        h1(N);
    } else {
        h2(N);
    }
}

auto f(int i) () -> int
{
    return (i * i) %12;
}

auto g(int i) () -> int
{
    return i % 12;
}

auto h1(int i) () -> int
{
    return h11(i);
}

auto h2(int i) () -> int
{
    return h21(i);
}

auto h11(int i) () -> int
{
    return i * i;
}

auto h21(int i) () -> int
{
    return i + h211(i);
}

auto h211(int i)
-> int
{
    return -i;
}

```

main()

```

auto main() -> int
{
    auto N = 10;
    if (f(N) < g(N)) {
        h1(N);
    } else {
        h2(N);
    }
}

auto f(int i) () -> int
{
    return (i * i) %12;
}

auto g(int i) () -> int
{
    return i % 12;
}

auto h1(int i) () -> int
{
    return h11(i);
}

auto h2(int i) () -> int
{
    return h21(i);
}

auto h11(int i) () -> int
{
    return i * i;
}

auto h21(int i) () -> int
{
    return i + h211(i);
}

auto h211(int i)
-> int
{
    return -i;
}

```

main()

g() int i = 10

```

auto main() -> int
{
    auto N = 10;
    if (f(N) < g(N)) {
        h1(N);
    } else {
        h2(N);
    }
}

auto f(int i) () -> int
{
    return (i * i) %12;
}

auto g(int i) () -> int
{
    return i % 12;
}

auto h1(int i) () -> int
{
    return h11(i);
}

auto h2(int i) () -> int
{
    return h21(i);
}

auto h11(int i) () -> int
{
    return i * i;
}

auto h21(int i) () -> int
{
    return i + h211(i);
}

auto h211(int i)
-> int
{
    return -i;
}

```

main()

```

auto main() -> int
{
    auto N = 10;
    if (f(N) < g(N)) {
        h1(N);
    } else {
        h2(N);
    }
}

auto f(int i) () -> int
{
    return (i * i) %12;
}

auto g(int i) () -> int
{
    return i % 12;
}

auto h1(int i) () -> int
{
    return h11(i);
}

auto h2(int i) () -> int
{
    return h21(i);
}

auto h11(int i) () -> int
{
    return i * i;
}

auto h21(int i) () -> int
{
    return i + h211(i);
}

auto h211(int i)
-> int
{
    return -i;
}

```

main()

h1() int i = 10

```

auto main() -> int
{
    auto N = 10;
    if (f(N) < g(N)) {
        h1(N);
    } else {
        h2(N);
    }
}

auto f(int i) () -> int
{
    return (i * i) %12;
}

auto g(int i) () -> int
{
    return i % 12;
}

auto h1(int i) () -> int
{
    return h11(i);
}

auto h2(int i) () -> int
{
    return h21(i);
}

auto h11(int i) () -> int
{
    return i * i;
}

auto h21(int i) () -> int
{
    return i + h211(i);
}

auto h211(int i)
-> int
{
    return -i;
}

```

main()

h1() int i = 10

h11() int i = 10

```

auto main() -> int
{
    auto N = 10;
    if (f(N) < g(N)) {
        h1(N);
    } else {
        h2(N);
    }
}

auto f(int i) () -> int
{
    return (i * i) %12;
}

auto g(int i) () -> int
{
    return i % 12;
}

auto h1(int i) () -> int
{
    return h11(i);
}

auto h2(int i) () -> int
{
    return h21(i);
}

auto h11(int i) () -> int
{
    return i * i;
}

auto h21(int i) () -> int
{
    return i + h211(i);
}

auto h211(int i)
-> int
{
    return -i;
}

```

main()

h1() int i = 10

```

auto main() -> int
{
    auto N = 10;
    if (f(N) < g(N)) {
        h1(N);
    } else {
        h2(N);
    }
}

auto f(int i) () -> int
{
    return (i * i) %12;
}

auto g(int i) () -> int
{
    return i % 12;
}

auto h1(int i) () -> int
{
    return h11(i);
}

auto h2(int i) () -> int
{
    return h21(i);
}

auto h11(int i) () -> int
{
    return i * i;
}

auto h21(int i) () -> int
{
    return i + h211(i);
}

auto h211(int i)
-> int
{
    return -i;
}

```

main()

Functions at run time

```
Sin(double x)
x:0.125663..
RP:<in main()>
```

```
main()
    i:4
RP:OS
```

```
1 auto sin(double x) -> int {
2     // Somehow calculate sin of x
3     return answer;
4 }
5 auto main() -> int {
6     double x{3.141592653589793};
7     for (int i = 0; i < 100; ++i) {
8         std::cout << i * x / 100
9             << sin(i * x / 100) << "\n";
10    }
11 }
```

When a function is called, e.g., when we write

`f(value1,value2,value3)` for a function `f` declared as

```
ret_type f(type1 x, type2 y, type3 z) :
```

- A "workbook" in memory called a stack frame is created for the call
- The local variables `x`, `y`, `z` are created, as if using instructions `type1 x{value1}`,
`type2 y{value2}`, `type3 z{value3}`.
- A return address is stored.
- The actual body of the function is executed
- When the function concludes, execution continues at the stored return address, and the stack frame is destroyed

Member functions

```
1 class D {  
2     int nm;  
3     double d;  
4 public:  
5     void val(double x) { d = x; }  
6     auto val() const -> double { return d; }  
7     auto name() const { return nm; }  
8     auto operator+(double x1) const -> double;  
9 };
```

```
1 auto D::operator+(double x) const -> double  
2 {  
3     return d + x * x;  
4 }
```

```
1 0000000000000000 <_ZNK1DplEd>:  
2     vmulsd xmm0,xmm0,xmm0  
3     vaddsd xmm0,xmm0,QWORD PTR [ rdi + 0x8 ]  
4     ret
```

- Object of class types are passed using their addresses.
The compiler uses the `address` of the class type variable and `offsets` to its parts to find the appropriate values to use.
- Return value is written to the type appropriate registers
- Execution continues at the previously stored return address

Aside: reading assembly code

The compiler explorer

Exercise 2.1:

The compiler explorer <https://godbolt.org> provides a great tool to quickly examine the assembly code corresponding to a code snippet. It is possible to choose different compilers, give compiler options ... Use it to quickly check the assembly code generated for simple functions. Compare different compilers. Try the examples in [examples/assembly](#). Vary the compiler and compiler options and see how the assembler changes.

- `class.cc` contains two functions doing the same thing. One operates on a bare `double` variable, and another on a `double` variable wrapped in a class with simple accessor functions. How different are the generated assembler code for the two functions ?
- `constref.cc` has two similar functions, one accepting a double by value and another by constant reference. What differences can you see ?

See also: CppCon 2016: Serge Guelton “C++ Costless Abstractions: the compiler view”

Stack

```
1  class V3 {
2      double x,y,z;
3      V3 cross(const V3 &);
4      auto dot(const V3 &) -> double;
5  };
6  auto prob(int i, const V3& x, const V3& y)
7      -> double
8  {
9      int j = i % 233;
10     V3 tmp{x};
11     for (; j < i; ++j) {
12         tmp = tmp.cross(y);
13     }
14     return tmp.dot(x);
15 }
```

- Heavily reused memory locations
- Likely cached, therefore, fast
- All local (block scope) variables of any type, which come into existence inside a block, and expire at the end of the block, i.e., with *automatically managed lifetime*.

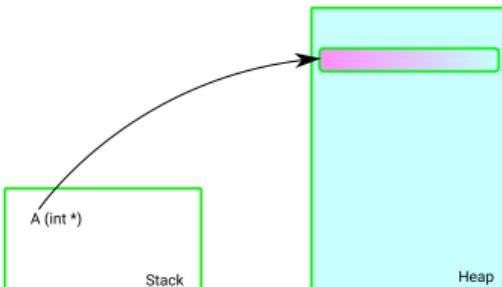
Global storage

```
1 auto prob(int i) -> double
2 {
3     static int c{0};
4     ++c;
5     if (c%1000==0) {
6         std::cout << "Call count reached "
7             << c << "\n";
8     }
9     static const double L[]={3.14, 2.71};
10    return L[i%2];
11 }
```

- Variables outside any function
- Variables marked with the `static` keyword in functions
- Floating point constants, array initializer lists, jump tables, virtual function tables

Heap

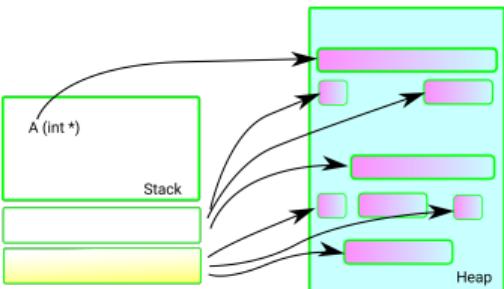
```
1 void f()
2 {
3     int *A = new int[1000000];
4     // calculations with A
5     delete [] A;
6 }
```



- Explicitly/implicitly managed memory through `new`, `delete`, `malloc` or `free`
- Objects who come into existence with a `new` call, and live until an explicit `delete` call
- Can store very large objects which don't fit in the stack
- Arrays whose size is not known at compile time. C99 style variable length arrays are not standard C++.

Heap

```
1 void f()
2 {
3     int *A = new int[1000000];
4     // calculations with A
5     delete [] A;
6 }
```



- Tends to get fragmented
- Must find a suitably sized unused block, and must keep track of what is and isn't in use → allocation and deallocation are expensive
- Must remember to free memory before accessing pointers go out of scope
- Objects stored one after the other may end up in very different locations
- Slower than stack storage

```
1 static void StringCreation(benchmark::State& state) {
2     for (auto _ : state) {
3         std::string created_string("hello");
4         benchmark::DoNotOptimize(created_string);
5     }
6 }
7 BENCHMARK(StringCreation);
8 static void StringCopy(benchmark::State& state) {
9     std::string x = "hello";
10    for (auto _ : state) {
11        std::string copy(x);
12    }
13 }
14 BENCHMARK(StringCopy);
```

Exercise 2.2:

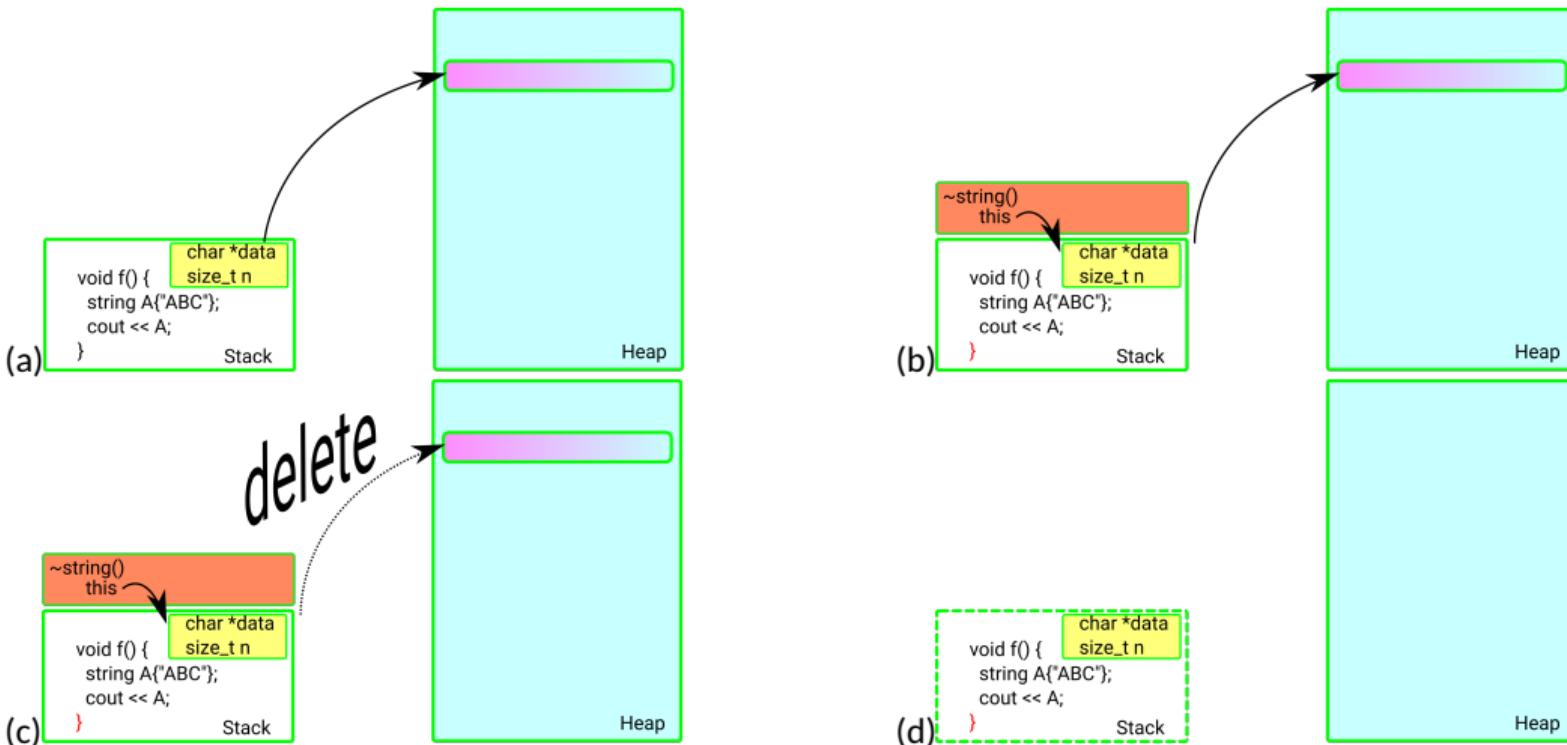
Test the cost of allocation and deallocation using the microbenchmarking site [quick-bench.com!](http://quick-bench.com) Their default example is the code given here (above), comparing string creation and copy. Note down the timings. Then add about 20 'o's at the end of the "hello" in each bench mark, i.e., "hello" \mapsto "hellooooooooooooooooooooo". Compare the timings again! To understand the timings, you need to find out about "small string optimisation".

Resource handles

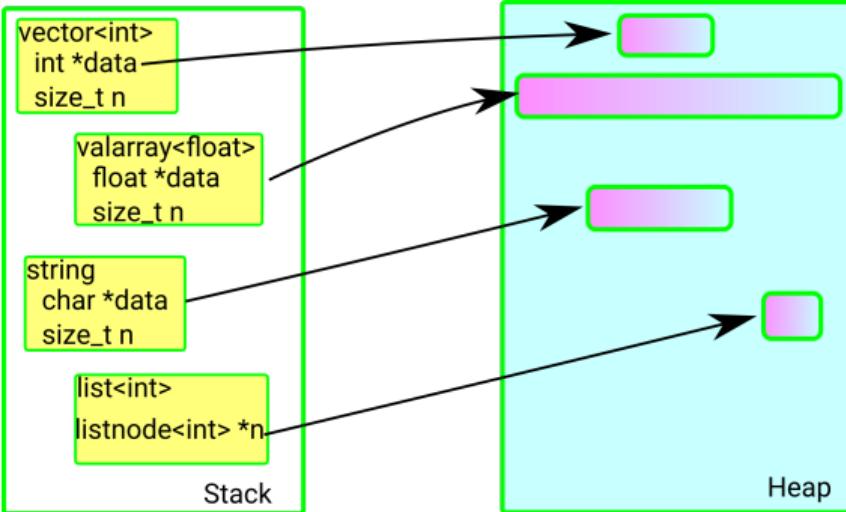
- Instead of bare heap allocation/deallocation, allocate in constructors or member functions (a)
- When the scope of the variable ends, the destructor is automatically called (b)
- Destructor should free any resources still in use (c)
- The variable can now expire (d)

The labels (a), (b), (c) and (d) refer to the figures in the following slide.

Resource handles



Resource handles



- STL containers (except `std::array`) are "resource" handles
- Memory management is done through constructors, the destructor and member functions

- No legitimate use of objects of the class should result in a memory leak
- Most data is on the heap. The objects on the stack are light-weight handles.

Resource handles

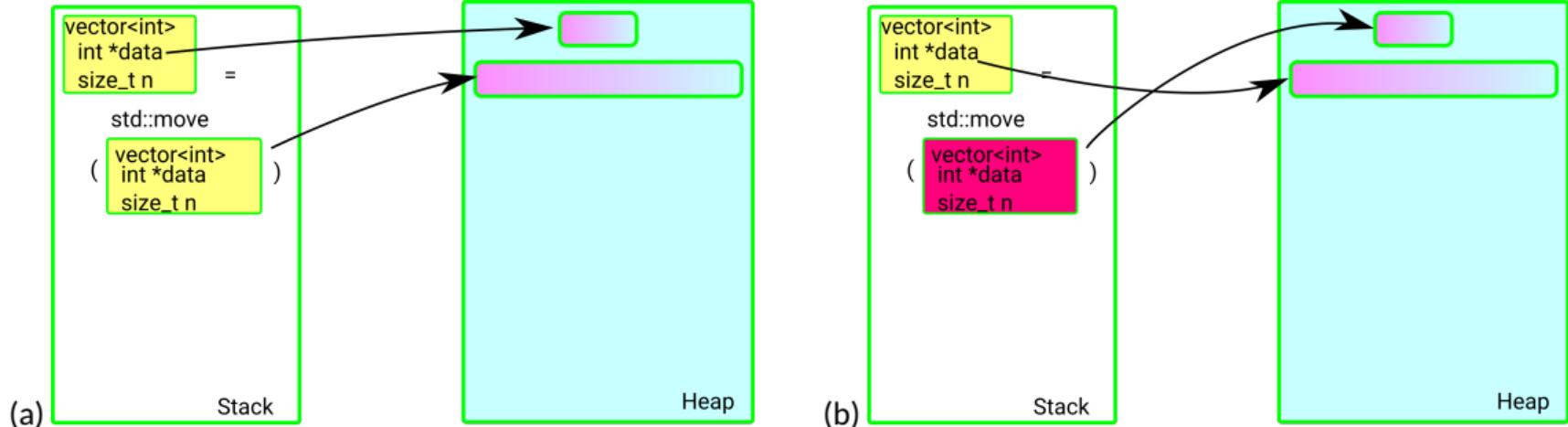
```
1 vector<int> A(32,0);
2 vector<double> B(64,0.);
3 vector<complex<double>> C(128);
4 vector<bool> D(256);
5 cout << sizeof(A) << ", "
6     << sizeof(B) << ", "
7     << sizeof(C) << ", "
8     << sizeof(D) << "\n";
```

Quiz

What will the program print ?

- A. 32, 64, 128, 256
- B. 32, 64, 256, 64
- C. 24, 24, 24, 24
- D. 24, 24, 24, *depends on the library*

Resource handles

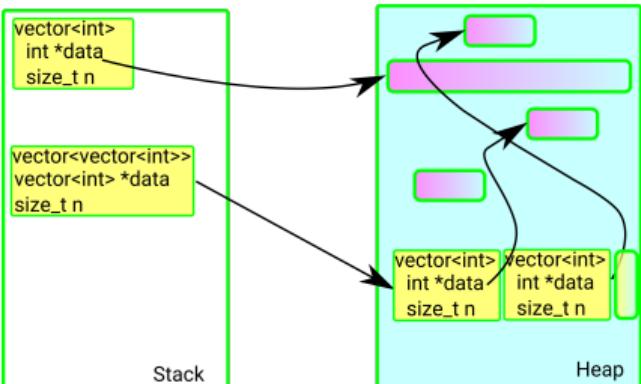


Move

- Can transfer ownership of the resources very cheaply
- Actual data on the heap need not be touched at all!
- Just some pointer re-assignments on the stack (a), (b)

Resource handles

```
vector<vector<int>> v(10, vector<int>(10, 0));  
...  
for (int i = 0; i < 10; ++i) {  
    for (int j = 0; j < 10; ++j) {  
        v[i][j] = i + j;  
        //v.operator[](i).operator[](j);  
        //(*(*(v.dat + i)).dat + j)  
    }  
}
```



- In C++, objects (instances of a class) can live on the stack or on the heap
- Putting resource handles like `vector<int>` on the heap, while allowed, incurs the cost of additional indirections
- It is almost always possible to avoid cumbersome beasts like `vector<vector<int>>`, `vector<vector<vector<vector<int>>>` or `int*****`.
- I wish I hadn't seen such "multi-dimensional arrays" in production code!

If you need your own 2D, 5D etc. arrays, ...

```
1 template <class T> class array2d {
2     vector<T> v;
3     size_t nc{0}, nr{0};
4 public:
5     auto operator()(size_t i, size_t j) const -> T { return v[i * nc + j]; }
6     auto operator()(size_t i, size_t j) -> T& { return v[i * nc + j]; }
7 };
```

- Use a wrapper class around an STL container, like `vector` or `valarray`
- Either overload the `operator()` to access a given row and column ...
- ... or use a helper class for rows and mimic 2D arrays in C

Exercise 2.3:

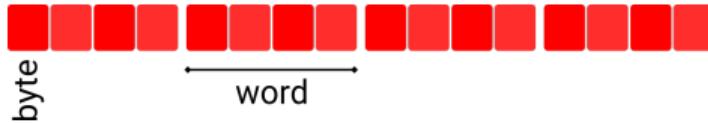
`examples/array2d` contains the class template shown here.

std::array

```
1 // examples/stdarray.cc
2 #include <iostream>
3 #include <array>
4
5 auto main() -> int
6 {
7     std::array A{0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9};
8     std::cout << "Size of array on stack = " << sizeof(A) << "\n";
9     std::cout << "size() = " << A.size() << "\n";
10 }
```

- Resembles other STL containers, but this is not just a handle.
- Does not need a data element to store the size, as the size is "part of the name" of the type!
- Moving an `std::array` has order N complexity, as each individual element needs to be moved. No pointer swapping trick can do the job for this.

Data alignment



- Data is read or written with a unit size called word. On the most common architectures, word size is 4 or 8 bytes.
- Data alignment means, putting data on memory addresses which are integral multiples of the word size
- n -byte aligned address has $\geq \log_2(n)$ least significant zeros
- Access for aligned data is fast
- If the size of a primitive type does not exceed the word size, access to aligned data of that type is also atomic

Data alignment

- The X86 architecture is tolerant of misaligned data. Programs run, even if they can't use SSE features
- PowerPC throws a hardware exception, which may be handled by the OS. For unaligned 8 byte access, a 4,610% performance penalty has been discussed
(<http://www.ibm.com/developerworks/library/pa-dalign/>)
- On other systems, crashes, data corruption, incorrect results are all possibilities

Data alignment

- Usually, primitive types are aligned by their "natural alignment": 4 byte `int` has 4 byte alignment, 8 byte `double` has alignment of 8 and so on
- A class has a natural alignment equal to the strictest requirement of its members
- The `alignof` operator can be used to query the alignment of a type
- The `alignas` keyword can be used to set a stricter alignment requirement

Exercise 2.4:

Verify the above using the example program `examples/alignof.cc`.

Data structure padding

```
class D { // alignment : 8, because of d
    int nm; // alignment requirement 4.
    double d; // Must have alignment 8.
public:
    void val(double x) { d=x; }
    auto val() const -> double { return d; }
    auto name() const { return nm; }
    auto operator+(double x1) const -> double;
};

auto D::operator+(double x) const -> double
{
    return d + x * x;
}

0000000000000000 <_ZNK1DplEd>:
    vmulsd xmm0,xmm0,xmm0
    vaddsd xmm0,xmm0,QWORD PTR [rdi+0x8]
    ret
```

- Alignment requirement of members can necessitate introduction of padding between members

Data structure padding

```
class D { // alignment : 8, because of d
    int nm; // alignment requirement 4.
    double d; // Must have alignment 8.
public:
    void val(double x) { d=x; }
    auto val() const -> double { return d; }
    auto name() const { return nm; }
    auto operator+(double x1) const -> double;
};

auto D::operator+(double x) const -> double
{
    return d + x * x;
}

0000000000000000 <_ZNK1DplEd>:
    vmulsd xmm0,xmm0,xmm0
    vaddsd xmm0,xmm0,QWORD PTR [rdi+0x8]
    ret
```

- Alignment requirement of members can necessitate introduction of padding between members
- What happens to the assembler here, if we put a comma between n and m in the name `nm` in class D ?

Data structure padding

```
class D { // alignment : 8, because of d
    int nm; // alignment requirement 4.
    double d; // Must have alignment 8.
public:
    void val(double x) { d=x; }
    auto val() const -> double { return d; }
    auto name() const { return nm; }
    auto operator+(double x1) const -> double;
};

auto D::operator+(double x) const -> double
{
    return d + x * x;
}

0000000000000000 <_ZNK1DplEd>:
    vmulsd xmm0,xmm0,xmm0
    vaddsd xmm0,xmm0,QWORD PTR [rdi+0x8]
    ret
```

- Alignment requirement of members can necessitate introduction of padding between members
- What happens to the assembler here, if we put a comma between n and m in the name `nm` in class `D` ?
- What if we make it `int n,m,p;` ? Test it using the compiler explorer! Copy and paste code from `examples/assembly/class2.cc` .

Data structure padding

```
1  class A {  
2      char c;  
3      double x;  
4      int d;  
5  };  
6  // Compiled as if it was ...  
7  char c;  
8  char pad[7];  
9  double x;  
10 int d;  
11 char pad2[4]; // why is this here ?  
12 // Overall alignment alignof(double)  
13 // size of struct = 24
```

```
1  class B {  
2      double x;  
3      int d;  
4      char c;  
5  };  
6  // Compiled as if it was ...  
7  double x;  
8  int d;  
9  char c;  
10 char pad[3];  
11 // Overall alignment alignof(double)  
12 // size of struct = 16
```

- Due to padding, size of structures can be bigger than the sum of sizes of their elements
- C++ rules do not allow the compiler to reorder elements for space
- Carefully choosing the declaration order of class members can save memory

Alignment specifiers

```
1 alignas(64) double x[4]; // ok
2
3 alignas(64) vector<double>(4) a;
4 // Pointless.
5 // The above simply aligns the resource
6 // handle, not the data on the heap
7
8 alignas(64) array<double, 4> A;
9 // This is fine, as std::array has
10 // real data in its struct
11
12 template <class T, int vecsize>
13 struct alignas(vecsize) simd_t
14 {
15     array<T, vecsize/sizeof(T)> data;
16 };
17 // We have requested that all objects
18 // of type simd_t should be aligned
19 // to vecsize bytes.
```

- The `alignas` keyword can specify alignment for variables
- Can be attached to a class declaration so that all objects of that type have a specified alignment
- Be mindful about what you are aligning when you use `alignas` for a resource handle like `vector` or `valarray`
- It is possible to attach an extended alignment specifier to the class declaration

Exercise 2.5:

The `examples/align0.cc` has an example class template, which creates a data array of the right size to fill the SIMD vector width irrespective of the input data type. It illustrates the use of `alignof` and `alignas`, and variable templates.

Exercise 2.6:

The `examples/align1.cc` shows that the usual mechanisms of dynamic allocation up to C++14 do not provide any guarantees about alignment greater than the natural alignment of the type. The behaviour changed in C++17 for types with explicitly specified extended alignment specifier like our `simd_t` class of the previous example. Finally, `examples/align2.cc` shows the use of a new version of the `new` operator introduced in C++17, which accepts an alignment argument.

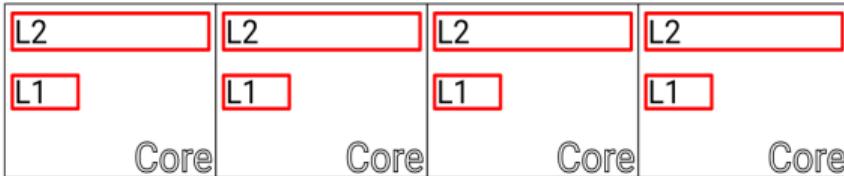
Memory effects

Memory

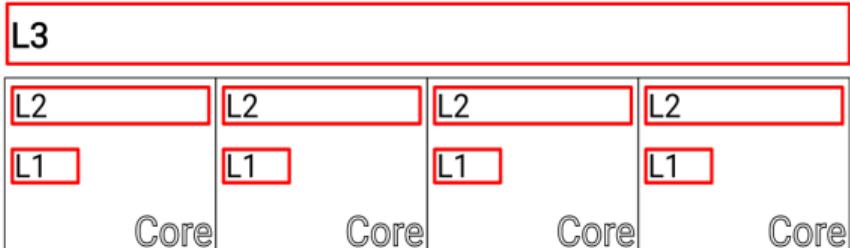
DRAM

- Fun fact: in 1 clock cycle of the CPU on my laptop, a photon travels about 10 cm in vacuum!

L3

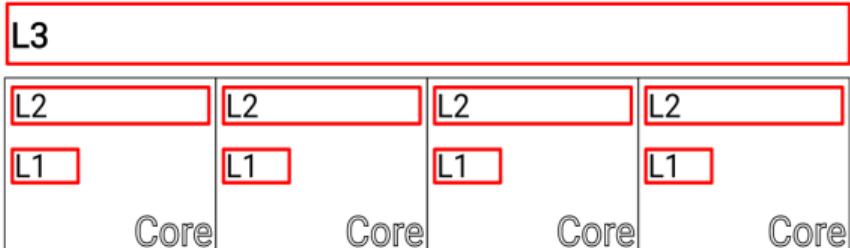


Memory



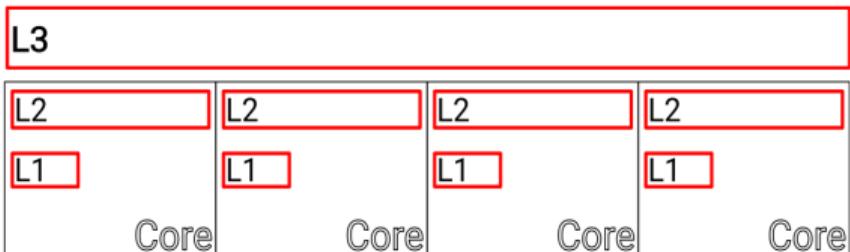
- Fun fact: in 1 clock cycle of the CPU on my laptop, a photon travels about 10 cm in vacuum!
- CPUs contain a certain amount of “cache” memory, which is faster to access, but much smaller than RAM
- Cost of fetching one integer from the main memory can be a hundred times larger than getting it from the L1 cache
- When the CPU looks for data from one address in memory, it is copied from RAM to the cache and then used.

Memory



- Fun fact: in 1 clock cycle of the CPU on my laptop, a photon travels about 10 cm in vacuum!
- CPUs contain a certain amount of “cache” memory, which is faster to access, but much smaller than RAM
- Cost of fetching one integer from the main memory can be a hundred times larger than getting it from the L1 cache
- When the CPU looks for data from one address in memory, it is copied from RAM to the cache and then used.

Memory



- Fun fact: in 1 clock cycle of the CPU on my laptop, a photon travels about 10 cm in vacuum!
- CPUs contain a certain amount of “cache” memory, which is faster to access, but much smaller than RAM
- Cost of fetching one integer from the main memory can be a hundred times larger than getting it from the L1 cache
- When the CPU looks for data from one address in memory, it is copied from RAM to the cache and then used.

- If it is immediately accessed again, it is in the cache, and can be used without the cost of fetching it from RAM
- Memory is fetched in “cache lines”. Successive operations on contiguous memory locations do not incur the full cost of main memory access

Analogy



- City with a 10000 cars, with a central "processing unit", which may require its citizens
- The administrative area has a small parking lot¹ with, say 1000 parking spaces
- When a person is required, he or she must drive to the parking lot to participate in the business
- Since, a car has 4 seats, every time one person is required, 3 others also arrive uninvited. There is no way to request the presence of one seat of the car.

[1]: What Every Programmer Should Know About Memory, (2007) Ulrich Drepper, Red Hat, Inc.

Analogy

- What happens when all 1000 spaces are taken and a new person is required ?
- How does the administration determine if a person is in the parking lot or must be summoned from the city ?

Analogy

- What happens when all 1000 spaces are taken and a new person is required ?
- How does the administration determine if a person is in the parking lot or must be summoned from the city ?
- One solution: number the parking spots 001 .. 999, and set the rule that a car with number plate ABCD can only be in the spot BCD.
 - neighbours can be in the parking lot at the same time (License plates are (obviously :-)) assigned by street address)
 - most commonly, people in the same neighbourhood are required one after the other

Analogy

- What happens when all 1000 spaces are taken and a new person is required ?
- How does the administration determine if a person is in the parking lot or must be summoned from the city ?
- One solution: number the parking spots 001 .. 999, and set the rule that a car with number plate ABCD can only be in the spot BCD.
 - neighbours can be in the parking lot at the same time (License plates are (obviously :-)) assigned by street address)
 - most commonly, people in the same neighbourhood are required one after the other
- What happens when someone is called, and arrives to find the correct parking spot already occupied ?
 - The previous occupant must leave, and drive all the way when required once again.

Analogy

- What happens when all 1000 spaces are taken and a new person is required ?
- How does the administration determine if a person is in the parking lot or must be summoned from the city ?
- One solution: number the parking spots 001 .. 999, and set the rule that a car with number plate ABCD can only be in the spot BCD.
 - neighbours can be in the parking lot at the same time (License plates are (obviously :-)) assigned by street address)
 - most commonly, people in the same neighbourhood are required one after the other
- What happens when someone is called, and arrives to find the correct parking spot already occupied ?
 - The previous occupant must leave, and drive all the way when required once again.
- What if the business involves multiple questions being asked to these two people ?

Analogy

- What happens when all 1000 spaces are taken and a new person is required ?
- How does the administration determine if a person is in the parking lot or must be summoned from the city ?
- One solution: number the parking spots 001 .. 999, and set the rule that a car with number plate ABCD can only be in the spot BCD.
 - neighbours can be in the parking lot at the same time (License plates are (obviously :-)) assigned by street address)
 - most commonly, people in the same neighbourhood are required one after the other
- What happens when someone is called, and arrives to find the correct parking spot already occupied ?
 - The previous occupant must leave, and drive all the way when required once again.
- What if the business involves multiple questions being asked to these two people ?
- Alternative: number parking spot sets from 00 to 99 and let there be 10 unnumbered spots in each set. A car with license plate number ABCD can go to any of the unoccupied spaces in set CD. This arrangement is somewhat more tolerant of conflicts such as the one in the question above, which is common in many programs with regular patterns in the data.

Finding out cache information about your CPU

```
root > dmidecode -t cache
# dmidecode 3.1
Getting SMBIOS data from sysfs.
SMBIOS 3.0.0 present.
Handle 0x0007, DMI type 7, 19 bytes
Cache Information
    Socket Designation: L1 Cache
    Configuration: Enabled, Not Socketed, Level 1
    Operational Mode: Write Back
    Location: Internal
    Installed Size: 256 kB
    Maximum Size: 256 kB
    Supported SRAM Types:
        Synchronous
    Installed SRAM Type: Synchronous
    Speed: Unknown
    Error Correction Type: Parity
    System Type: Unified
    Associativity: 8-way Set-associative}

Handle 0x0008, DMI type 7, 19 bytes
Cache Information
    Socket Designation: L2 Cache
    Configuration: Enabled, Not Socketed, Level 2
    Operational Mode: Write Back
    Location: Internal
    Installed Size: 1024 kB
    Maximum Size: 1024 kB
    Supported SRAM Types:
        Synchronous
    Installed SRAM Type: Synchronous
    Speed: Unknown
    Error Correction Type: Single-bit ECC
    System Type: Unified
    Associativity: 4-way Set-associative

Handle 0x0009, DMI type 7, 19 bytes
Cache Information
    Socket Designation: L3 Cache
    Configuration: Enabled, Not Socketed, Level 3
    Operational Mode: Write Back
    Location: Internal
    Installed Size: 8192 kB
    Maximum Size: 8192 kB
    Supported SRAM Types:
        Synchronous
    Installed SRAM Type: Synchronous
    Speed: Unknown
    Error Correction Type: Multi-bit ECC
    System Type: Unified
    Associativity: 16-way Set-associative
```

CPU cache

```
not_root> getconf -a | grep -i cache
LEVEL1_ICACHE_SIZE          32768
LEVEL1_ICACHE_ASSOC          8
LEVEL1_ICACHE_LINESIZE       64
LEVEL1_DCACHE_SIZE          32768
LEVEL1_DCACHE_ASSOC          8
LEVEL1_DCACHE_LINESIZE       64
LEVEL2_CACHE_SIZE            262144
LEVEL2_CACHE_ASSOC           4
LEVEL2_CACHE_LINESIZE        64
LEVEL3_CACHE_SIZE            8388608
LEVEL3_CACHE_ASSOC           16
LEVEL3_CACHE_LINESIZE        64
LEVEL4_CACHE_SIZE            0
LEVEL4_CACHE_ASSOC           0
LEVEL4_CACHE_LINESIZE        0
```

- Tools : `lscpu` , `dmidecode` , `lshw` , `getconf`
- Different tools may aggregate information differently (e.g., how total as opposed to per-core cache is reported)
- L1d cache is for data, L1i is for instructions (instructions must live somewhere in the cache too!)

CPU cache

..... SSSS SS11 1111

- For a 64 byte cache line, the least 6 bits of the address refer to the location inside the cache line. Not relevant in determining parking spot in the cache
- If we have 32kb of L1d cache, with a 64 byte line, we have 512 "parking spots" (lines)
- An 8 way associative cache will then have $512/8 = 64$ sets, and use the further 6 bits of a memory address to assign a set
- If we keep accessing random places in memory, it is very easy to run out of L1 cache: in the example here, we have only 64 sets!
- Address bits higher than the least 12 are not used in determining where in the cache a value is stored: any two addresses separated by 2^{12} map to the same set in the L1 cache.
- Variables with memory addresses separated by $\text{setcount} \times \text{linesize}$ compete for the same cache line
- For better performance, one should strive to write code utilising the whole cache line before it is evicted

Memory access patterns

```
1     std::vector<int> A(N * N, 0);
2     for (size_t i = 0; i < N; ++i) {
3         for (size_t j = 0; j < N; ++j) {
4             A[i * N + j] += j + i;
5         }
6     }
```

```
1     for (size_t i = 0; i < N; ++i) {
2         for (size_t j = 0; j < N; ++j) {
3             A[j * N + i] += j + i;
4         }
5     }
```

```
1     for (size_t i = 0; i < N * N; ++i) {
2         A[ pos[i] ] += i;
3     }
```

See also: CppCon 2016: Timur Doumler "Want fast C++? Know your hardware!"

- Q: Which way of accessing the “matrix” is faster, and by how much ?
- A: For N=10000, my laptop takes about 0.037 seconds for the row major pattern (top), and about 0.26 seconds for the column major pattern (middle), and 1.86 seconds for random pattern (bottom)

Memory

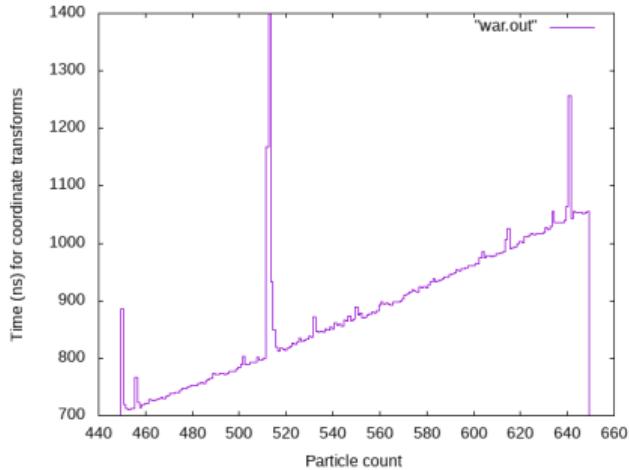
```
1 constexpr size_t size = 2 << 26;
2 std::vector< long > A(size, 0);
3 for (size_t step = 1; step <= 2048; step *= 2) {
4     for (size_t i = 0; i < size; i += step) A[i]++;
5 }
```

Step	Time
1	0.0967211
2	0.0943902
4	0.0929546
8	0.113927
16	0.137341
32	0.120449
64	0.0675447
128	0.0415029
256	0.016718
512	0.00694461
1024	0.00357155
2048	0.00178591

- For small step sizes, increasing the number of writes to the array does not change the total time.
- Multiple accesses inside a cache line has minimal extra cost.

4K aliasing

```
1 // Layout :  
2 // x0, x1, x2 ... xn-1, y0, y1 ... yn-1,  
3 // z0, z1 ... zn-1, wx0, wx1 ... wxn-1,  
4 // wy0, wy1 ... wyn-1, wz0 ... wzn-1  
5  
6 for (size_t i=0;i<npart;++i) {  
7     wx(i)=R(0,0)*x(i)+R(0,1)*y(i)+R(0,2)*z(i);  
8     wy(i)=R(1,0)*x(i)+R(1,1)*y(i)+R(1,2)*z(i);  
9     wz(i)=R(2,0)*x(i)+R(2,1)*y(i)+R(2,2)*z(i);  
10 }
```



- Innocent looking code can sometimes produce weird changes in performance based on array sizes
- The spike in required time here comes for a particle count of about 512, when the different components of the data for one particle are separated by exactly 4kB.

Exercise 2.7: Memory effects

The following examples illustrate the cache effects discussed so far:

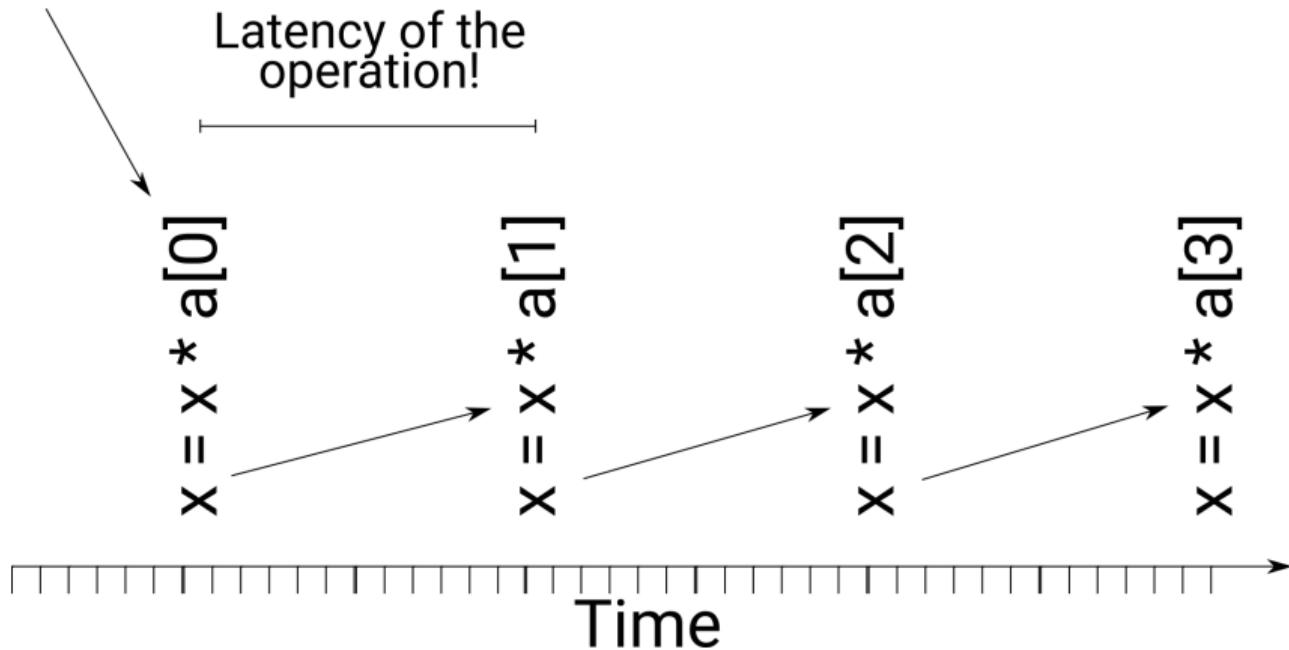
- `traverse0.cc` can be used to compare contiguous and non-contiguous access of a large array
- `every_nth.cc` compares times for accessing every n'th element, and highlights the cache line
- `transpose.cc` transpose operation on a matrix, which involves lots of non-contiguous access

Recommendations

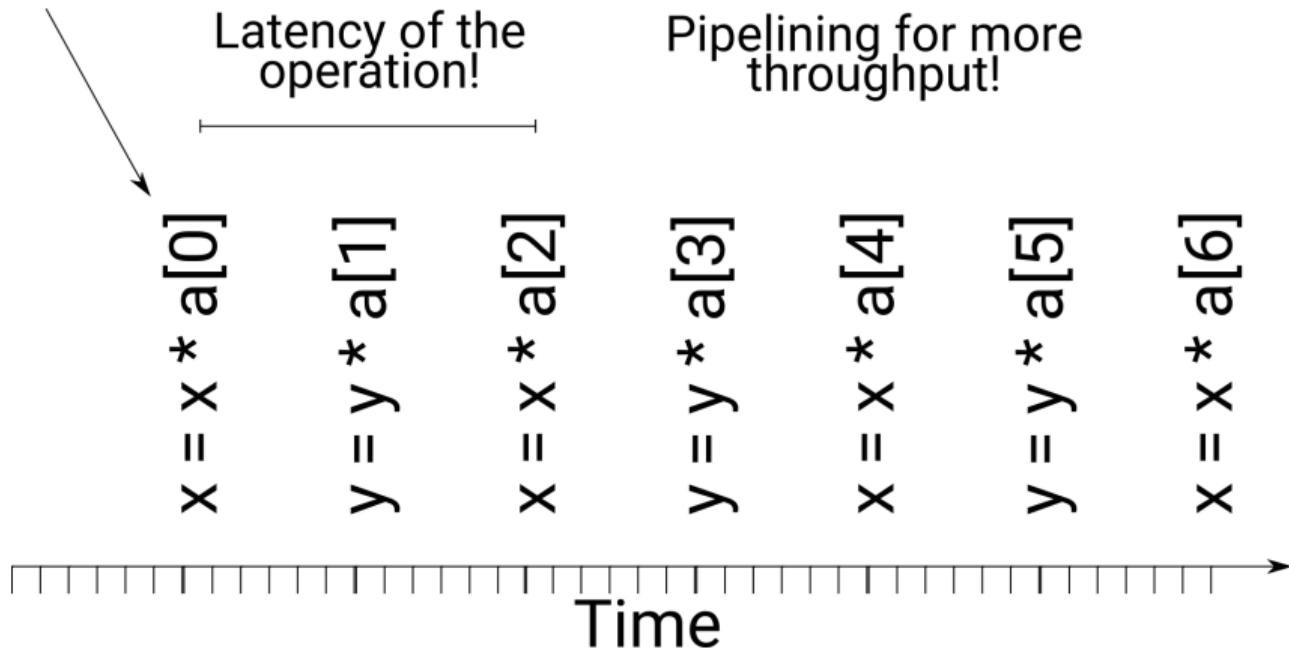
- Prefer `std::array` and `std::vector` for all your container needs as a default. Many libraries also provide other containers with contiguous storage providing advantages for specific use cases. Anything with non-contiguous storage needs to be carefully justified
- Organise code to maximise the use of any cache line that has been fetched:
 - Collate processing of nearby memory locations
 - Organise data structures so that things processed together are also stored near each other
- Keep variables as local as possible

Pipeline

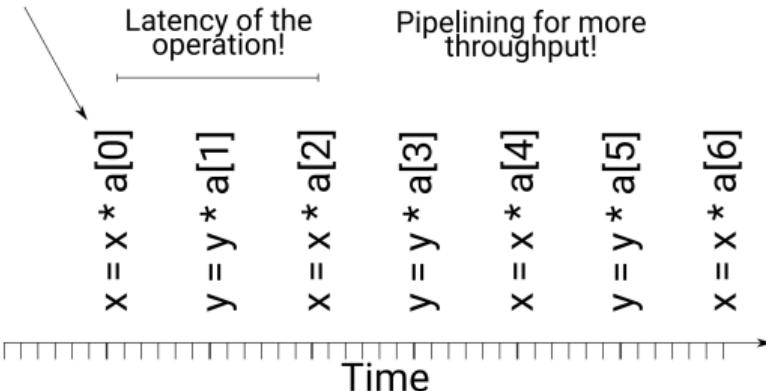
Instruction pipeline



Instruction pipeline



Instruction pipeline



- CPUs do not have to wait until one calculation is totally finished before starting another
- They “pipeline”. If the data required for another instruction is available, that can start execution before the first is finished
- Because of pipelining, the processors are able to perform more operations in time Δt than $\frac{\Delta t}{t_{latency}}$
- Data dependencies create stalls in the pipeline
- Some modern processors even execute instructions “out of order”, to keep the pipeline busy

Exercise 2.8:

The program `examples/ilp.cc` demonstrates the effects of data dependencies. Two alternative versions of a loop are given, performing the same total number of computations. One of them runs more than 5 times faster, because it avoids dependencies between successive calculations.

Pipeline and branched code



- Instruction fetch
- Instruction decode
- Instruction execute
- Memory access
- Register write back

```
1  if (x+y>5) f();  
2  else g();
```

- request mem x
- request mem y
- calc x+y
- calc res > 5
- ?

The "next instruction" depends on the outcome of an instruction.

Branch prediction

```
1  for (int i = 0; i < N; ++i) {  
2      if (p[i] > gen()) {  
3          b[i] = a[i] + c[i];  
4          ++fwd;  
5      } else {  
6          a[i] = b[i] + c[i];  
7          ++rev;  
8      }  
9  }  
10 nngb = 0;  
11 while (a) {  
12     dist[nngb++] = distf(a, i);  
13 }
```

- When branches are encountered, the CPU simply guesses which way it will go, and fetches instructions accordingly
- If the guess is right, no pipeline stall
- If it is wrong, all operations done with that guess must be purged

- For efficient execution, different units in the pipeline must be kept busy as much as possible

Branch mis-prediction penalty

```
1  for (int i = 0; i < N; ++i) {  
2      if (p[i] > gen()) {  
3          a[i] = (b[i] > r0 && b[i] < r1  
4                               && c[i] < b[i]);  
5      } else {  
6          a[i] = b[i] + c[i];  
7          ++rev;  
8      }  
9  }  
10 nngb=0;  
11 while (a) {  
12     dist[nngb++] = distf(a,i);  
13 }
```

- If statements, switches, loops contain obvious branches
- The ternary operator `a = cond ? v1 : v2` is (sometimes) a branch

- Not so obvious branches include boolean `||` and `&&` operators:
 - In a sequence of operations like `a || b || c || ...`, the operands are evaluated left to right until the first true value is obtained
 - In a sequence of operations like `a && b && c && ...`, the operands are evaluated left to right until the first false value is obtained

Not branches

```
1 auto f(int i) -> int
2 {
3     static const int a[4]={4,3,2,1};
4     int ans=0;
5     ans += (a[i]<i)?1:2;
6     return ans;
7 }
```

```
1 0000000000000000 <_Z1fi>:
2     cmp    edi,0x4
3     setl   al
4     movzx  eax,al
5     inc    eax
6     ret
```

- Conditional assignments are often reorganised as simple sequential instructions by compilers using special assembler instructions when available
- Loops with small loop counts may be automatically unrolled at compile time leaving simple linear code

```
1 0000000000000000 <_Z1fdPd>:
2     subsd  xmm0,QWORD PTR [rdi]
3     subsd  xmm0,QWORD PTR [rdi+0x8]
4     subsd  xmm0,QWORD PTR [rdi+0x10]
5     subsd  xmm0,QWORD PTR [rdi+0x18]
6     ret
```

Not branches

```
1 auto f(double x, double A[4]) -> double
2 {
3     double a=x;
4     for (int i=0;i<4;++i) a-=A[i];
5     return a;
6 }
```

- Conditional assignments are often reorganised as simple sequential instructions by compilers using special assembler instructions when available
- Loops with small loop counts may be automatically unrolled at compile time leaving simple linear code

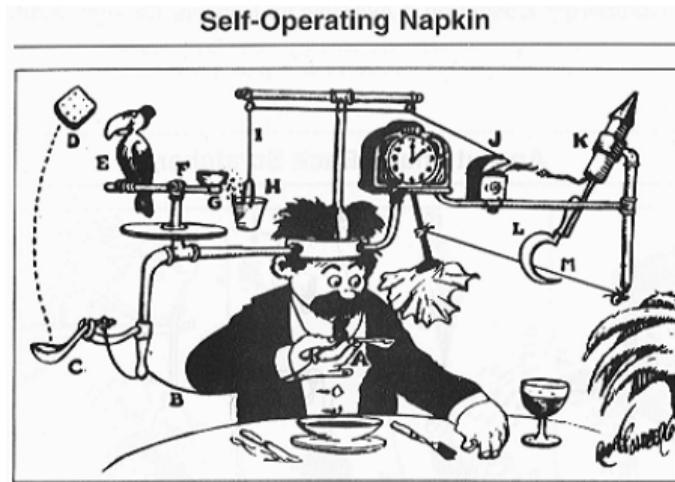
```
1 0000000000000000 <_Z1fdPd>:
2     subsd xmm0,QWORD PTR [rdi]
3     subsd xmm0,QWORD PTR [rdi+0x8]
4     subsd xmm0,QWORD PTR [rdi+0x10]
5     subsd xmm0,QWORD PTR [rdi+0x18]
6     ret
```

Exercise 2.9:

Branch prediction effectiveness using the example program `examples/branch_prediction.cc`, compare the processor on your own computer with the AMD EPYC 7742 processors on JUSUF. The program partitions an array of integers into 3 ranges. Running it with a command line argument (value ignored) causes it to first sort the array and then perform the same partitioning actions. In the sorted array, the branches are easier to predict. What do you observe ? How do different compilers compare ?

Class inheritance, virtual functions and performance

- Class hierarchies constitute a flexible and beginner friendly tool kit
- In a fairly wide variety of applications, such as graphics, and many simulations, they may form the backbone of a robust, flexible code base
- Because of their success in some areas, they have been massively overused, leading to elaborate Rube Goldberg machines, which are neither easy to read nor particularly fast
- In modern C++, we should explore alternative ways to solve our problems
- Understanding how it works can help us more easily identify situations where a deep class hierarchy will be a bad idea.



Inheritance

Base class
data

Derived class
extra data



- Inheriting class may add more data, but it retains all the data of the base
- The base class functions, if invoked, will see a base class object
- The derived class object *is a* base class object, but with additional properties

Inheritance

Base class
data

Derived class
extra data



access of base
class functions

access of derived class functions
(qualified by private, protected etc)

- A pointer to a derived class always points to an address which also contains a valid base class object.
- `baseptr=derivedptr` is called "upcasting". Always allowed.
- Implicit downcasting is not allowed. Explicit downcasting is possible with `static_cast` and `dynamic_cast`

Inheritance

Base class
data

Derived class
extra data



access of base
class functions

access of derived class functions
(qualified by private, protected etc)

```
1  class Base {
2    public:
3      void f() {std::cout<<"Base::f()\n";}
4    protected:
5      int i{4};
6    };
7  class Derived : public Base {
8    int k{0};
9  public:
10   void g() {std::cout<<"Derived::g()\n";}
11 };
12 auto main() -> int
13 {
14   Derived b;
15   Base *ptr=&b;
16   ptr->g(); // Error!
17   static_cast<Derived *>(ptr)->g(); //OK
18 }
```

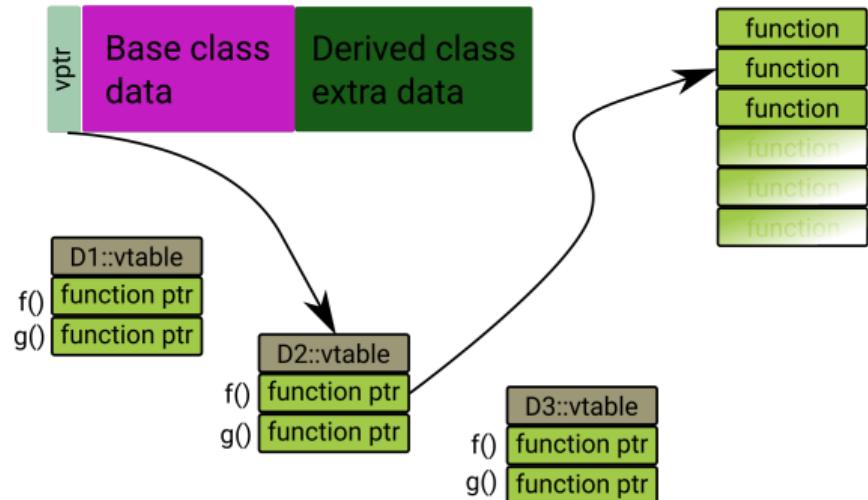
Class inheritance with virtual functions

```
1 auto main() -> int
2 {
3     vector<unique_ptr<Shape>> shapes;
4     shapes.push_back(make_unique<Circle>(0.5, Point(3,7)));
5     shapes.push_back(make_unique<Triangle>(Point(1,2),Point(3,3),Point(2.5,0)));
6     ...
7     for (auto&& shape : shapes) {
8         std::cout << shape->area() << '\n';
9     }
10 }
```

- A [smart] pointer to a base class is allowed to point to an object of a derived class
- Here, `shape[0]->area()` will call `Circle::area()`, `shape[1]->area()` will call `Triangle::area()`
- But, how does it work ?

Calling virtual functions: how it works

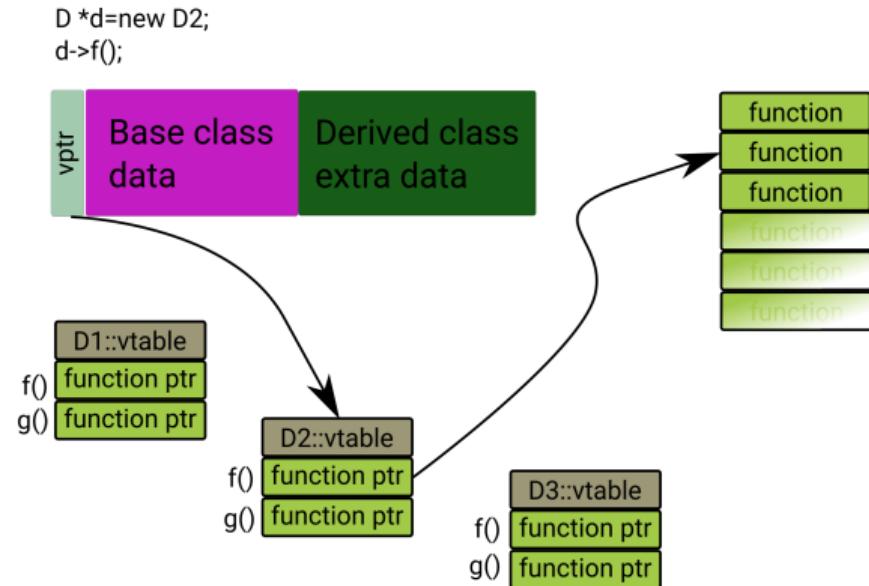
```
D *d=new D2;  
d->f();
```



- For classes with virtual functions, the compiler inserts an invisible pointer member to the data and additional book keeping code
- There is a table of virtual functions for each derived class, with entries pointing to function code somewhere
- The `vptr` pointer points to the `vtable` of that particular class

Calling virtual functions: how it works

- Virtual function call proceeds by first finding the right *vtable*, then the correct entry for the called function, dereferencing that function pointer and then executing the correct function body
- Branch mispredictions, cache misses ...
- For HPC applications, use of virtual functions in hot sections **will hurt performance**



- Often, the polymorphic behaviour sought after using virtual functions can be implemented with CRTP without the virtual function overhead

Polymorphism without virtual functions

Polymorphism without virtual functions

- We have already seen how function overloading gives us a polymorphic unit : the overload set
- Different variation of a function is picked based on the type of the input parameters, or the constraints satisfied by the input parameters
- This is one kind of *static polymorphism*

Tag dispatching

```
1  struct flying {};
2  struct swimming {};
3  template <class T>
4  void do_something(T && t, flying)
5  {
6      t.fly(a,b);
7  }
8  template <class T>
9  void do_something(T && t, swimming) {...}
10 //...
11 template <class T>
12 void do_something(T t)
13 {
14     do_something(t, typename T::preferred_mode{});
15 }
```

```
1  class Buzzard {
2  public:
3      using preferred_mode = typename flying;
4  };
5  class Whale {
6  public:
7      using preferred_mode = typename swimming;
8  };
9  //...
10 Buzzard b;
11 do_something(b);
12 Whale w;
13 do_something(w);
```

- Logically similar operations on different types, where the operations depend on certain properties of the types

Tag dispatching

```
1  struct flying {};
2  struct swimming {};
3  template <class T>
4  void do_something(T && t, flying)
5  {
6      t.fly(a,b);
7  }
8  template <class T>
9  void do_something(T && t, swimming) {...}
10 //...
11 template <class T>
12 void do_something(T t)
13 {
14     do_something(t, typename T::preferred_mode{});
15 }
```

```
1  class Buzzard {
2  public:
3      using preferred_mode = typename flying;
4  };
5  class Whale {
6  public:
7      using preferred_mode = typename swimming;
8  };
9  //...
10 Buzzard b;
11 do_something(b);
12 Whale w;
13 do_something(w);
```

- Logically similar operations on different types, where the operations depend on certain properties of the types
- “Dispatch” functions to guide the compiler to a suitable implementation based on a “tag” in the incoming type

Tag dispatching

```
1  struct flying {};
2  struct swimming {};
3  template <class T>
4  void do_something(T && t, flying)
5  {
6      t.fly(a,b);
7  }
8  template <class T>
9  void do_something(T && t, swimming) {...}
10 //...
11 template <class T>
12 void do_something(T t)
13 {
14     do_something(t, typename T::preferred_mode{});
15 }
```

```
1  class Buzzard {
2  public:
3      using preferred_mode = typename flying;
4  };
5  class Whale {
6  public:
7      using preferred_mode = typename swimming;
8  };
9  //...
10 Buzzard b;
11 do_something(b);
12 Whale w;
13 do_something(w);
```

- Logically similar operations on different types, where the operations depend on certain properties of the types
- “Dispatch” functions to guide the compiler to a suitable implementation based on a “tag” in the incoming type
- Does not tie the overload to a specific type: dispatches based on some property of the input type

SFINAE : Substitution Failure is not an Error

```
1 // Examples/sfinae0.cc
2 template <class V>
3 void f(const V &v, typename V::iterator * jt=0)
4 {
5     std::cout << "Container overload\n";
6     for (auto x : v) std::cout << x << " ";
7     std::cout << "\n";
8 }
9
10 void f(...)
11 {
12     std::cout << "Catch all overload\n";
13 }
14
15 auto main() -> int
16 {
17     std::list L{0.1, 0.2, 0.3, 0.4, 0.5, 0.6};
18     int A[4]{4, 3, 2, 1};
19     f(A);
20     f(L);
21 }
```

- Overload resolution of templates
- If substitution fails, overload discarded
- All parameters, expressions and the return type in declarations
- Substitution failure : ill-formed type or expression when a substitution is made
- Not in function body!

enable_if

```
1 // enable_if and enable_if_t are defined
2 // in the namespace std. We show them
3 // here to explain how they are used.
4 template <bool B, class T> struct enable_if;
5 template <class T> struct enable_if<true, T> {
6     using type=T ;
7 }
8 template <bool B, class T=void>
9 using enable_if_t=typename enable_if<B,T>::type;
10
11 template <class T>
12 enable_if_t<is_integral<T>::value, T>
13 Power(T x, T y) {
14     // Implementation suitable for
15     // integral number parameters
16 }
17 template <class T>
18 enable_if_t<is_floating_point<T>::value, T>
19 Power(T x, T y) {
20     // Implementation suitable for
21     // floating point parameters
22 }
```

- Only if the first parameter is true, the structure `enable_if` has a member type called `type` set to the second template parameter
- Using the `type` member of an `enable_if` struct in a declaration will lead to an ill-formed expression when the condition parameter is false. That version of the function will then be ignored

Let's not do such things any more. We have concepts now.

Exercise 2.10:

The tag dispatching technique is demonstrated in `examples/tag_dispatch.cc`.

Exercise 2.11:

`examples/sfinae0.cc` is a simple syntax illustration for SFINAE. Knowledge of history is important, but let this not be how you write your code in 2020s.

Choosing algorithm based on API

```
1 template <class C> size_t algo(C&& x)
2 {
3     if constexpr (hasAPI<C>) {
4         x.helper();
5         return x.calculateFast();
6     } else {
7         return x.calculate();
8     }
9 }
```

- We want to write a general algorithm for an operation
- In case the function argument has a certain member function, we have a neat and quick solution
- Otherwise, we have a fallback solution

Choosing algorithm based on API

```
1 template <class T> struct hasAPI_t {
2     using basetype =
3         typename remove_reference<T>::type;
4     template <class C>
5     static constexpr auto test(C * x) ->
6         decltype(x->calculateFast(),
7                   x->helper(),
8                   bool{})
9     {
10         return true;
11     }
12     static constexpr bool test(...) {
13         return false;
14     }
15     static constexpr auto value =
16         test(static_cast<basetype*>(nullptr));
17 }
```

- The “template function” `hasAPI_t` has a member `value` initialized via a `constexpr` function, which passes information about the templated type to the `test` function
- Two variants of the `test` function exist, one always returning false, to cover the “everything else” case

Choosing algorithm based on API

```
1 template <class T> struct hasAPI_t {
2     using basetype =
3         typename remove_reference<T>::type;
4     template <class C>
5     static constexpr auto test(C * x) ->
6         decltype(x->calculateFast(),
7                   x->helper(),
8                   bool{});
9     {
10         return true;
11     }
12     static constexpr bool test(...) {
13         return false;
14     }
15     static constexpr auto value =
16         test(static_cast<basetype*>(nullptr));
17 };
```

- The positive version of the `test` function defines its return type using `decltype`, but applying it to a comma separated list of necessary API expressions
- A comma separated list of expressions evaluates to the last value, but each value in the list is checked for syntax

Choosing algorithm based on API

```
1 template <class T> struct hasAPI_t {
2     using basetype =
3         typename remove_reference<T>::type;
4     template <class C>
5     static constexpr auto test(C * x) ->
6         decltype(x->calculateFast(),
7                   x->helper(),
8                   bool{});
9     {
10         return true;
11     }
12     static constexpr bool test(...) {
13         return false;
14     }
15     static constexpr auto value =
16         test(static_cast<basetype*>(nullptr));
17 };
```

- The positive version of the `test` function defines its return type using `decltype`, but applying it to a comma separated list of necessary API expressions
- A comma separated list of expressions evaluates to the last value, but each value in the list is checked for syntax

Choosing algorithm based on API

```
1 template <class T> struct hasAPI_t {
2     using basetype =
3         typename remove_reference<T>::type;
4     template <class C>
5     static constexpr auto test(C * x) ->
6         decltype(x->calculateFast(),
7                   x->helper(),
8                   bool{})
9     {
10         return true;
11     }
12     static constexpr bool test(...) {
13         return false;
14     }
15     static constexpr auto value =
16         test(static_cast<basetype*>(nullptr));
17 }
```

- The positive version of the `test` function defines its return type using `decltype`, but applying it to a comma separated list of necessary API expressions
- A comma separated list of expressions evaluates to the last value, but each value in the list is checked for syntax

- If the type of the argument does not have the member functions, the return type of the function can not be determined, and the overload is rejected

Choosing algorithm based on API

```
1 template <class T> constexpr bool hasAPI = hasAPI_t<T>::value;
2 template <class C> std::enable_if_t< hasAPI<C>, size_t > algo(C && x)
3 {
4     x.helper();
5     return x.calculateFast();
6 }
7 template <class C> std::enable_if_t< !hasAPI<C>, size_t > algo(C && x)
8 {
9     return x.calculate();
10 }
```

- What remains, is to make a nice wrapper template variable so that we can say `hasAPI<T>`, instead of `hasAPI_t<T>::value` when we need it.
- The dispatch functions are written using `enable_if_t`, so that we pick the `calculateFast` function over `calculate`, if it is available

Nah!

Choosing algorithm based on API

```
1 template <class T>
2 concept FastCalculator = requires (T rex) {
3     { rex.calculateFast()  };
4     { rex.helper()  };
5 };
6 template <FastCalculator C> auto algo(C && x)
7 {
8     x.helper();
9     return x.calculateFast();
10 }
11 template <class C> auto algo(C && x)
12 {
13     return x.calculate();
14 }
```

- Write a **concept** describing what member functions, inner types (like `value_type` for iterators) an object should have to satisfy the API
- Overload based on whether the constraints are satisfied!

Choosing algorithm based on API

```
1 auto main() -> int
2 {
3     Machinery obj;
4     auto res = algo(obj);
5     std::cout << "Result = " << res << "\n";
6 }
```

- Users of our great algorithm can simply call our `algo()` in their code
- If there is a `calculate` function, everything will work.
- If the author of the library providing `Machinery` goes on to implement `calculateFast` in the `Machinery` class, without any changes on the client side, or in the `algo` function, the compiler will make sure that the (hopefully) better, `calculateFast` function is used

Exercise 2.12:

The folder `examples/apishimming` contains the example `hasAPI` template function used in this section, with an application that uses it. By freeing the commented implementation of `calculateFast`, and recompiling, you will see that the call to `algo` automatically switches to use `calculateFast`.

Exercise 2.13:

The folder `examples/apishimming2` contains a possible logical extension of `apishimming`. Here we split the program into a provider and a client side. The “provider” develops the `Machinery` class, and in this case, also the `algo` function template. The “client” uses this library. We will use the C++20 modules feature. The provider creates a module file, and chooses what to export to the client. The client receives a “pre-compiled, binary, module file”, without the source for the templates and concepts used in the implementation. You will see that the client is still able to use the `Machinery` and `algo()` like in the previous example!

Note: The `modules` feature is not actually meant to be a new distribution tool for pre-compiled template code. The binary module interface files are allowed to contain all sorts of platform specific details which would make them unusable in the above scenario unless the provider and client are indeed on the same computing infrastructure and share the compiler and libraries. The above demo only shows that compilation of the code merely using pre-compiled modules can proceed without parsing the module sources again, as would be the case if we used header files.

```

1 template <class T> struct hasAPI_t {
2     using basetype = typename remove_reference<T>::type;
3     template <class C> static constexpr auto test(C * x) ->
4         decltype(x->calculateFast()),
5         x->helper(),
6         bool(){}
7     {
8         return true;
9     }
10    static constexpr bool test(...){}
11    {
12        return false;
13    }
14    static constexpr auto value =
15        test(static_cast<b>basetype*</b>>(nullptr));
16};
17 template <class T>
18 constexpr bool hasAPI = hasAPI_t<T>::value;
19 template <class C>
20 std::enable_if_t< hasAPI<C>, size_t > algo(C && x)
21 {
22     x.helper();
23     return x.calculateFast();
24 }
25 template <class C>
26 std::enable_if_t< !hasAPI<C>, size_t > algo(C && x)
27 {
28     return x.calculate();
29 }

```

Will get the job done.

```

1 template <class T>
2 concept FastCalculator = requires (T rex) {
3     { rex.calculateFast() };
4     { rex.helper() };
5 };
6 template <FastCalculator C> auto algo(C && x)
7 {
8     x.helper();
9     return x.calculateFast();
10 }
11 template <class C> auto algo(C && x)
12 {
13     return x.calculate();
14 }

```

Will get the job done and keep you sane.

Curiously Recurring Template Pattern

- You need types A and B which have some properties in common, which can be calculated using similar data
- There are a few polymorphic functions, but conceptually A and B are so different that you don't expect to store them in a single pointer container
- The penalty of using virtual functions seems to matter
- Option 1: implement as totally different classes, just copy and paste the common functions

Curiously Recurring Template Pattern

- You need types A and B which have some properties in common, which can be calculated using similar data
- There are a few polymorphic functions, but conceptually A and B are so different that you don't expect to store them in a single pointer container
- The penalty of using virtual functions seems to matter
- Option 1: implement as totally different classes, just copy and paste the common functions

Curiously Recurring Template Pattern

- You need types A and B which have some properties in common, which can be calculated using similar data
- There are a few polymorphic functions, but conceptually A and B are so different that you don't expect to store them in a single pointer container
- The penalty of using virtual functions seems to matter
- Option 1: implement as totally different classes, just copy and paste the common functions

Curiously Recurring Template Pattern

- You need types A and B which have some properties in common, which can be calculated using similar data
- There are a few polymorphic functions, but conceptually A and B are so different that you don't expect to store them in a single pointer container
- The penalty of using virtual functions seems to matter
- Option 1: implement as totally different classes, just copy and paste the common functions
- Option 2: try the CRTP

Curiously Recurring Template Pattern

```
1 template <class D> struct ViewInterface {
2     const D * der() const {
3         return static_cast<const D *>(this);
4     }
5     auto begin() const {
6         // Won't compile if D does not inherit from this
7         return der()->begin_impl();
8     }
9     auto version() const -> int {
10        // Non-polymorphic "common" function
11        return 42;
12    }
13 };
14 struct Atoi : public ViewInterface<Atoi> {
15     auto begin_impl() const { return bg; }
16 };
17 struct List : public ViewInterface<List> {
18     auto begin_impl() const -> string {
19         return &basenode;
20     }
21 };
```

```
1 template <class T>
2 auto proc(ViewInterface<T> v) {
3     auto b = v.begin();
4     // ...
5 }
6 auto main() -> int {
7     List H;
8     proc(H);
9     proc(Atoi{33});
10 }
```

- A function can demand that the inputs have a particular interface defined in the CRTP base
- Any input type inheriting from the CRTP base will be usable
- Polymorphism without virtual functions
- Enforces an interface at compile time
- Usually faster than virtual functions

Expression Templates

Expression Templates

```
1 template <typename T>
2 class vec {
3     std::vector<T> dat;
4 public:
5     vec(size_t n) : dat(n) {}
6     auto operator[](size_t i) const -> T {
7         return dat[i];
8     }
9     auto operator[](size_t i) -> T & {
10        return dat[i];
11    }
12     size_t size() const{return dat.size();}
13 };
14 template <typename T>
15 auto operator+(const vec<T> & v1,
16                  const vec<T> & v2) -> vec<T>
17 {
18     assert(v1.size() == v2.size());
19     auto ans = v1;
20     for (size_t i = 0;i < ans.size(); ++i)
21         ans[i] += v2[i];
22     return ans;
23 }
```

```
1 vec<double> W(N), X(N), Y(N), Z(N);
2 //..
3 W = a * X + 2 * a * Y + 3 * a * Z;
```

- Naive implementation which expresses our intent elegantly
- Each multiplication and addition creates a temporary and does a loop over elements
- Poor performance

Expression templates

If only we had a special class ...

- ... which stored references to `X`, `Y` and `Z`
- and had an `operator[]` which returns `a * X[i] + 2 * a * Y[i] + 3 * a * Z[i]`
- We could equip our `vec` class with a special assignment operator taking this special class as the right hand side

```
1 template <typename T>
2 class vec {
3     template <class XPR>
4     auto operator=(const XPR & r) ->    vec &
5     {
6         for (size_t i=0;i<size();++i) {
7             dat[i] = r[i]; // and r[i] returns a*X[i]+2*a*Y[i]+3*a*Z[i]
8         } // One single loop, no temporaries
9         return *this;
10    }
11};
```

-
- We need a different special class for every expression we have to evaluate

Expression templates

- If we make a class like :

```
template <typename LHS, typename RHS>
class vecsum {
    const LHS & lhs;
    const RHS & rhs;
public:
    vecsum(const LHS & l, const RHS & r) : lhs{l}, rhs{r} {
        assert(l.size()==r.size());
    }
    auto operator[](size_t i) const { return lhs[i] + rhs[i]; }
    auto size() const { return lhs.size(); }
};
```

- We can define the sum of two `vec` objects to be a `vecsum` type

```
template <typename LHS, typename RHS>
auto operator+(const LHS & v1, const RHS & v2) -> vecsum<LHS, RHS>
{
    return vecsum<LHS, RHS>(v1, v2);
}
```

Expression templates

- If we try `vec1+vec2`, no evaluation happens, and we get a `vecsum<vec, vec>` object, we can call `[]` on this object and cause the calculation to happen.
- But, if we try `vec1+54` or `34+"dino"`, we get nonsensical compound objects
- If we write our `operator+` like :

```
template <typename LHS, typename RHS>
auto operator+(const expr<LHS> & v1, const expr<RHS> & v2) -> vecsum<LHS, RHS> const
{
    return vecsum<LHS, RHS>(v1, v2);
}
```

, we can restrict the template to objects which match the pattern `expr<something>`

- If we further want composability of the operations, we need `vecsum<LHS, RHS>` to also match the pattern `expr<something>`

Expression templates

Design with CRTP

- CRTP: a base template `vecexpr` to use as a base for all expressions of `vec` objects

```
template <typename X> struct vecexpr {  
    X& der() noexcept { return *static_cast<X*>(this); }  
    const X& der() const { return *static_cast<const X*>(this); }  
};
```

Expression templates

Design with CRTP

- We make our expression classes like `vecsum` inherit from the template `vecxpr` instantiated on themselves:

```
1  template <typename T1, typename T2> class vecsum : public vecxpr<vecsum<T1, T2>> {
2      const T1 & lhs;
3      const T2 & rhs;
4  public:
5      using value_type = typename T1::value_type;
6      vecsum(const vecxpr<T1> & l, const vecxpr<T2> & r) : lhs{ l.der() }, rhs{ r.der() } {
7          assert(lhs.size() == rhs.size());
8      }
9      const auto operator[](size_t i) const { return lhs[i] + rhs[i]; }
10     size_t size() const { return lhs.size(); }
11 }
```

Expression templates

Design with CRTP

- `operator+` can now be written as:

```
1 template <typename T1, typename T2>
2 auto operator+(const vecexpr<T1> & v1, const vecexpr<T2> & v2) -> const vecsum<T1, T2> {
3     return vecsum<T1, T2>{ l, r };
4 }
```

Expression templates

Design with CRTP

- We also make the original `vec` class inherit from `vecexpr`

```
1 template <typename T> class vec : public vecexpr<vec<T>> {
2     std::vector<T> dat;
3 public:
4     using value_type = T;
5     vec(size_t n) : dat(n) {}
6     auto operator[](size_t i) const -> const T { return dat[i]; }
7     auto operator[](size_t i) -> T & { return dat[i]; }
8     size_t size() const { return dat.size(); }
9     size_t n_ops() const { return 0; }
10    template <typename X>
11    auto operator=(const vecexpr<X> & y) -> vec & {
12        dat.resize(y.der().size());
13        for (size_t i=0;i<y.size();++i) dat[i] = y.der()[i];
14        return *this;
15    }
16};
```

Expression templates

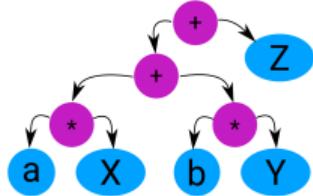
Design with CRTP

- We also make the original `vec` class inherit from `vecexpr`

```
1 template <typename T> class vec : public vecexpr<vec<T>> {
2     std::vector<T> dat;
3 public:
4     using value_type = T;
5     vec(size_t n) : dat(n) {}
6     auto operator[](size_t i) const -> const T { return dat[i]; }
7     auto operator[](size_t i) -> T & { return dat[i]; }
8     size_t size() const { return dat.size(); }
9     size_t n_ops() const { return 0; }
10    template <typename X>
11    auto operator=(const vecexpr<X> & y) -> vec & {
12        dat.resize(y.der().size());
13        for (size_t i=0;i<y.size();++i) dat[i] = y.der()[i];
14        return *this;
15    }
16};
```

- Notice the special assignment operator from an expression!

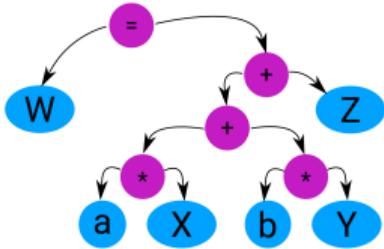
Expression templates



a * X + b * Y + Z;

```
vecsum<
    vecsum<
        vecscl<vec<double>>,
        vecscl<vec<double>>
    >,
    vec<double>> ({{a,X},{b,Y}},Z);
// Let's call this type EXPR
```

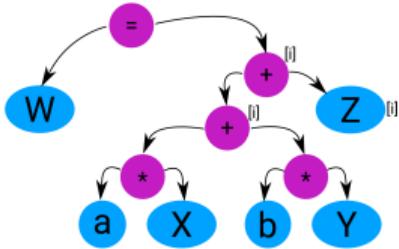
Expression templates



$W = a * X + b * Y + Z;$

```
vec<double> &  
vec<double>::operator=(const EXPR & E)  
{  
    dat.resize(E.size());  
    for (size_t i = 0; i < E.size(); ++i)  
        dat[i] = E[i];  
    return *this;  
}
```

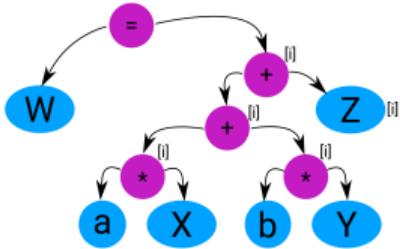
Expression templates



$W = a * X + b * Y + Z;$

```
vec<double> &  
vec<double>::operator=(const EXPR & E)  
{  
    dat.resize(E.size());  
    for (size_t i = 0; i < E.size(); ++i)  
        dat[i] = E[i];  
    return *this;  
}  
const auto vecsum<L,R>::operator[](size_t i) const {  
    return lhs[i] + rhs[i];  
}
```

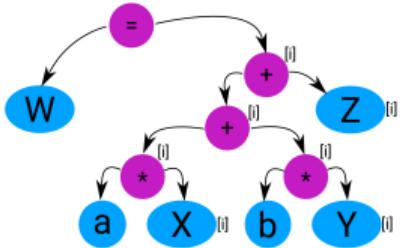
Expression templates



$W = a * X + b * Y + Z;$

```
vec<double> &
vec<double>::operator=(const EXPR & E)
{
    dat.resize(E.size());
    for (size_t i = 0; i < E.size(); ++i)
        dat[i] = E[i];
    return *this;
}
const auto vecsum<L,R>::operator[](size_t i) const {
    return lhs[i] + rhs[i];
}
const auto vecscl<T>::operator[](size_t i) const {
    return lhs * rhs[i];
}
```

Expression templates



$W = a * X + b * Y + Z;$

```
vec<double> &
vec<double>::operator=(const EXPR & E)
{
    dat.resize(E.size());
    for (size_t i = 0; i < E.size(); ++i)
        dat[i] = E[i];
    return *this;
}

const auto vecsum<L,R>::operator[](size_t i) const {
    return lhs[i] + rhs[i];
}

const auto vecscl<T>::operator[](size_t i) const {
    return lhs * rhs[i];
}

const auto vec<T>::operator[](size_t i) const {
    return data[i];
}
```

Expression templates

- Elegant high level syntax
- Reduce temporaries
- Loop fusion
- Delayed evaluation: apply algorithmic optimizations on the entire expression, e.g.,
 - Evaluate `Matrix1 * Matrix2 * Vector` as `Matrix1 * (Matrix2 * Vector)`
 - Detect and eliminate cancelling operations, e.g., `Matrix_xpr1.transpose().transpose()`
 - Use optimized low level kernels with assembler, intrinsics, calls to vendor libraries etc to do the work
- However, can greatly increase compilation times

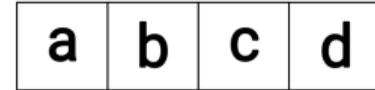
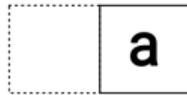
Exercise 2.14:

In `examples/xtmp0`, you will find a program which takes two numbers N and a as command line arguments, and creates 4 arrays W, X, Y, Z of size N (user defined array type `vec`). It fills X, Y and Z with random numbers and then calculates $W = a * X + 2 * a * Y + 3 * a * Z$, and times this operation by repeating the calculation 10 times. Two implementations of the user defined array type `vec` can be found: `naive_vec.hh` and `xtmp_vec0.hh`. Compile and run the program by alternating between the two headers. Study the code in `xtmp_vec0.hh`, which illustrates the ideas presented here about expression templates. The `xtmp_vec1.hh` implementation is almost the same, except using aligned allocation to store the arrays in the `vec` type. Test that as well.

Exercise 2.15:

Introduce your own matrix class in the set up used in `examples/xtmp0`, so that matrix vector multiplications can be parts of vector expressions and `M1*M2*v` is evaluated as two matrix vector products rather than a matrix-matrix product followed by a matrix vector product.

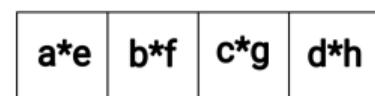
SIMD registers and operations



*



=



`vmulsd xmm0, xmm0, xmm1`

`vmulpd ymm0, ymm0, ymm1`

SIMD registers and operations

a3	a2	a1	a0
b3	b2	b1	b0
c3	c2	c1	c0

fmadd213pd source1, source2, source3

a3*b3+c3	a2*b2+c2	a1*b1+c1	a0*b0+c0
----------	----------	----------	----------

a3	a2	a1	a0
b3	b2	b1	b0
c3	c2	c1	c0
1	0	0	1

fmadd213pd source1, source2, source3, mask

a3*b3+c3	a2	a1	a0*b0+c0
----------	----	----	----------

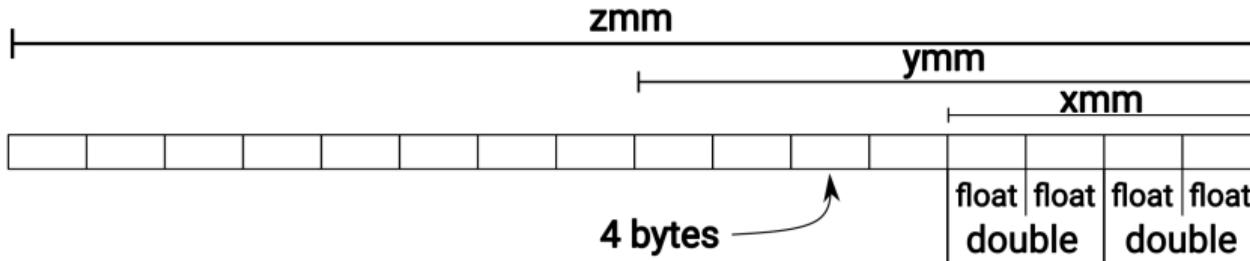
a3	a2	a1	a0
b3	b2	b1	b0
m3	m2	m1	m0

vblendpd source1, source2, mask

m3 ? b3 : a3	m2 ? b2 : a2	m1 ? b1 : a0	m0 ? b0 : a0
--------------	--------------	--------------	--------------

- Increasingly sophisticated instructions in newer CPUs
- Arithmetics, logical operations, shuffles, masked operations, trigonometry, cryptography ...

SIMD registers and operations



- `xmm0 , xmm1 ,... xmm7` (SSE)
- `xmm0 ... xmm15 , ymm0 ... ymm15` (AVX, AVX2, FMA)
- `xmm0 ... xmm31 , ymm0 ... ymm31 , zmm0 ... zmm31` (AVX512)

SIMD registers and operations



ymm0
A horizontal bar divided into 4 equal-sized boxes, all of which are empty (white).

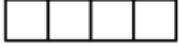
ymm1
A horizontal bar divided into 4 equal-sized boxes, all of which are empty (white).

SIMD registers and operations

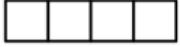


SIMD registers and operations



ymm0 

***=**

ymm1 

SIMD registers and operations



Automatic vectorization

- Compilers try to automatically identify opportunities to use SIMD instructions and generate appropriate code
- We write code exactly (or at least more or less) as before, and the vectorizer brings more speed
- Sometimes you may have to be careful about alignment of the arrays (`,
std::assume_aligned())`
- Sometimes you might need to indicate to the compiler that the multiple arrays involved in a loop do not overlap, can be assumed to be independent (`#pragma ivdep`)
- You may want to allow the compiler to proceed with the assumption that floating point arithmetic is associative (`-fassociative-math`)

Automatic vectorization

```
1 void f(double x[], double y[], unsigned N)
2 {
3     for (unsigned i=0U; i<N; ++i) x[i] = 5. * x[i] + y[i];
4 }
```

- Compiler asks : Can this loop be run in blocks of 4 or 8 for all inputs x and y ? What, if $y = x+1$!
- Then it makes careful decisions so that the results are correct for *every possible input*
- Sometimes, we don't care about every possible input. Our functions are often mere cogs in a bigger machine, and their contract is more limited

OpenMP SIMD directives

- Reorganize loop to run in chunks suitable for SIMD execution
- Syntax in C and C++ :

```
1 #pragma omp simd [clause [,clause] ...]
2 for ( ... ) {}
```

- Often possible to call straight forward inline functions or vector enabled functions

```
#pragma omp declare simd
```

- Can only be a traditional **for** loop. Loop count must be possible to determine at entry. No breaks.

```
1 template <typename T>
2 auto Vexv(T r2, T sigsa12) -> T {
3     auto sg2
4         = static_cast<T>(sqr(Lambda * sigsa12));
5     auto a
6         = static_cast<T>(icut2 * sqr(sigsa12));
7     a = a * a * a;
8     a = a * a;
9     auto b = static_cast<T>(sixdivLLcut2 * a);
10    a = 7.0 * a;
11    T r6 = sg2 / r2;
12    r6 = r6 * r6 * r6;
13    return ksa * (r6 * r6 + a + b * r2);
14 }
15 auto addup( __ ) -> double {
16     double tot{};
17     #pragma omp simd reduction(+:tot)
18     for (size_t i=vec_size; i<R2.size(); ++i)
19         tot += Vexv(R2[i], S12[i]);
20     return tot;
21 }
```

For an excellent overview, search for "Michael Klemm, Intel, SIMD Vectorization with OpenMP"



Add... More

```
C++ source #1.x
1 void f(double x[], double y[],
2     unsigned long N)
3 {
4     #pragma omp simd
5     for (auto i=0; i<N; ++i)
6         y[i] = 0.5 * x[i] + y[i];
7 }
```

x86-64 gcc 8.2 (Editor #1, Compiler #1) C++ x
x86-64 gcc 8.2 -O3

A
□11010 □.LX0: □lib.f: □.text □// □\s+ □

Libraries + Add new... Add tool...

```
1 f(double*, double*, unsigned long):
2     test rdx, rdx
3     je .L12
4     lea rcx, [rdx-1]
5     cmp rcx, 2
6     jbe .L6
7     mov rcx, rdx
8     shr rcx, 2
9     vmovupd ymm1, YMMWORD PTR .LC0[rip]
10    sal rdx, 5
11    xor eax, eax
12 .L4:
13    vmovupd ymm0, YMMWORD PTR [rdi+rax]
14    vfmadd213pd ymm0, ymm1, YMMWORD PTR [rsi+rax]
15    vmovupd YMMWORD PTR [rsi+rax], ymm0
16    add rax, 32
17    cmp rax, rcx
18    jne .L4
19    mov rax, rdx
20    and rax, -4
21    cmp rdx, rax
22    je .L14
23    vzeroupper
24 .L3:
25    vmovsd xmm0, QWORD PTR [rdi+rax*8]
26    vmovsd xmm0, QWORD PTR .LC1[rip]
27    lea rcx, [rsi+rax*8]
28    vfmadd213sd xmm1, xmm0, QWORD PTR [rcx]
29    vmovsd QWORD PTR [rcx], xmm1
30    lea rcx, [rax+1]
31    cmp rdx, rcx
32    jbe .L12
33    vmovsd xmm1, QWORD PTR [rdi+rax*8]
34    lea r8, [rsi+rax*8]
35    vfmadd213sd xmm1, xmm0, QWORD PTR [r8]
36    add rax, 2
37    vmovsd QWORD PTR [r8], xmm1
38    cmp rdx, rax
39    tbe .L12
```

x86-64 gcc 10.0.1 (Editor #1, Compiler #2) C++ x
x86-64 gcc 10.0.1 -std=c+

A
□11010 □.LX0: □lib.f: □.text □// □\s+ □

Libraries + Add new... Add tool...

```
67    lea rcx, [QWORD PTR [rsi+rax*8]] #4.29
68 .B1.19: # Preds .B1.19 ..B1.18
69    vmovsd xmm1, QWORD PTR [rdi+rax*8] #4.22
70    vfmadd213sd xmm1, xmm0, QWORD PTR [rcx+rax*8] #4.9
71    vmovsd QWORD PTR [rdi+rax*8], xmm1 #4.9
72    inc r8, 3.5
73    cmp r8, rdx #3.5
74    jb .B1.19 # Prob 82% #3.5
75    jmp .B1.27 # Prob 100% #3.5
76 .B1.21: # Preds .B1.4 ..B1.2
77    mov rcx, rdx #1.6
78    xor r8d, r8d #3.5
79    mov r9d, 1 #3.5
80    xor eax, eax #4.9
81    shr rcx, 1 #1.6
82    jb .B1.25 # Prob 9% #3.5
83    vmovsd xmm0, QWORD PTR .LC1@floatpacket.0[rip]
84 .B1.23: # Preds .B1.23 ..B1.22
85    vmovsd xmm1, QWORD PTR [rax+rdi] #4.22
86    inc rax, 3.5
87    vfmadd213sd xmm1, xmm0, QWORD PTR [rax+rsi] #4.9
88    vmovsd xmm2, QWORD PTR [rax+rax+rdi] #4.22
89    vmovsd QWORD PTR [rax+rdi], xmm1 #4.9
90    vfmadd213sd xmm2, xmm0, QWORD PTR [8+rax+rxi] #4.
91    vmovsd QWORD PTR [8+rax+rdi], xmm2 #4.9
92    add rax, 16 #3.5
93    cmp r8, rcx #3.5
94    jb .B1.23 # Prob 63% #3.5
95    lea r9, QWORD PTR [1+rax*8] #4.9
96 .B1.25: # Preds .B1.24 ..B1.21
97    lea rax, QWORD PTR [-1+r9] #3.5
98    cmp rax, rdx #3.5
99    jae .B1.27 # Prob 9% #3.5
100   vmovsd xmm1, QWORD PTR .LC1@floatpacket.0[rip]
101   vmovsd xmm0, QWORD PTR [-8+rax+r9*8] #4.22
102   vfmadd213sd xmm1, xmm0, QWORD PTR [-8+rax+r9*8] #
103   vmovsd QWORD PTR [-8+rax+r9*8], xmm1 #4.9
104 .B1.27: # Preds .B1.19 ..B1.25 ..B1.1 ..B1.17 ..B.
```

x86-64 clang (trunk) (Editor #1, Compiler #3) C++ x
x86-64 clang (trunk) -std=c++17 -O3

A
□11010 □.LX0: □lib.f: □.text □// □\s+ □ Intel □

Libraries + Add new... Add tool...

```
98    inc rax, 3.0
99    mov r8d, r10d
100   and r8d, 1
101   test r9, r9
102   je .LB88_12
103   mov ecx, 1
104   sub rcx, r10
105   lea r9, [r9 + rcx]
106   add r9, -1
107   xor ecx, ecx
108   vbroadcastsd zmm0, qword ptr [rip + .LCPI0_8] #200E
109 .LB88_14: # =>This Inner Loop Header
110   vmovupd zmm1, zmmword ptr [rdi + 8*rcx]
111   vmovupd zmm2, zmmword ptr [rdi + 8*rcx + 64]
112   vmovupd zmm3, zmmword ptr [rdi + 8*rcx + 128]
113   vmovupd zmm4, zmmword ptr [rdi + 8*rcx + 192]
114   vfmadd213pd zmm1, zmm0, zmmword ptr [rsi + 8*rcx]
115   vfmadd213pd zmm2, zmm0, zmmword ptr [rsi + 8*rcx +
116   vfmadd213pd zmm3, zmm0, zmmword ptr [rsi + 8*rcx +
117   vfmadd213pd zmm4, zmm0, zmmword ptr [rsi + 8*rcx +
118   vfmadd213pd zmm5, zmm0, zmmword ptr [rsi + 8*rcx +
119   vfmadd213pd zmm6, zmm0, zmmword ptr [rsi + 8*rcx +
120   vfmadd213pd zmm7, zmm0, zmmword ptr [rsi + 8*rcx +
121   vfmadd213pd zmm8, zmm0, zmmword ptr [rsi + 8*rcx +
122   vfmadd213pd zmm9, zmm0, zmmword ptr [rsi + 8*rcx +
123   vfmadd213pd zmm10, zmm0, zmmword ptr [rsi + 8*rcx +
124   vfmadd213pd zmm11, zmm0, zmmword ptr [rsi + 8*rcx +
125   vfmadd213pd zmm12, zmm0, zmmword ptr [rsi + 8*rcx +
126   vfmadd213pd zmm13, zmm0, zmmword ptr [rsi + 8*rcx +
127   vfmadd213pd zmm14, zmm0, zmmword ptr [rsi + 8*rcx +
128   vfmadd213pd zmm15, zmm0, zmmword ptr [rsi + 8*rcx +
129   vfmadd213pd zmm16, zmm0, zmmword ptr [rsi + 8*rcx +
130   vfmadd213pd zmm17, zmm0, zmmword ptr [rsi + 8*rcx +
131   vfmadd213pd zmm18, zmm0, zmmword ptr [rsi + 8*rcx +
132   vfmadd213pd zmm19, zmm0, zmmword ptr [rsi + 8*rcx +
133   vfmadd213pd zmm20, zmm0, zmmword ptr [rsi + 8*rcx +
134   vfmadd213pd zmm21, zmm0, zmmword ptr [rsi + 8*rcx +
135   vfmadd213pd zmm22, zmm0, zmmword ptr [rsi + 8*rcx +
136   vfmadd213pd zmm23, zmm0, zmmword ptr [rsi + 8*rcx +
137   vfmadd213pd zmm24, zmm0, zmmword ptr [rsi + 8*rcx +
138   vfmadd213pd zmm25, zmm0, zmmword ptr [rsi + 8*rcx +
139   vfmadd213pd zmm26, zmm0, zmmword ptr [rsi + 8*rcx +
140   vfmadd213pd zmm27, zmm0, zmmword ptr [rsi + 8*rcx +
141   vfmadd213pd zmm28, zmm0, zmmword ptr [rsi + 8*rcx +
142   vfmadd213pd zmm29, zmm0, zmmword ptr [rsi + 8*rcx +
143   vfmadd213pd zmm30, zmm0, zmmword ptr [rsi + 8*rcx +
144   add rcx, 64
145   add r9, 2
146   jne .LB88_14
```

Digging deeper

```
1 double pairwise(unsigned i, unsigned j,
2                  SOA * particle_record)
3 {
4     // very clever calculations
5 }
6 auto energy() -> double
7 {
8     double ans = 0.;
9     for (unsigned i=0; i < npt; ++i) {
10         #pragma omp please vectorize
11         for (unsigned j=i+1; j<npt; ++j) {
12             ans += pairwise(i, j, my_particle_record);
13         }
14     }
15     return ans;
16 }
```

Convenient. But what are we not doing ?

- Coding to load groups of 2 or 4 or 8 numbers, working with them and storing the results
- Comparing different ways to use SIMD instructions to solve the problem for our actual inputs
- Choosing where to use relaxed assumptions about floating point arithmetic at specific places in the code

Deviate only for special situations!

As HPC C++ programmers, we should know how to take full control of vectorization. But automatic or OpenMP based vectorization should be your first choice for production code. Most often they provide a cleaner, easier path. Sometimes, when the easier way does not provide enough low level access, we have ways to go beyond them.



Introduction to intrinsics

- Recognizing how numbers are stored and manipulated in the computer opens up new opportunities
- Computer arithmetic has more "fundamental" operations than normal mathematics : + , - , * , / , % , & , | , << , >>

```
1 auto morton_plain(unsigned long x,
2                     unsigned long y,
3                     unsigned long z)
4 {
5     auto ans = 0UL;
6     unsigned long i=0;
7     while (i<22) {
8         unsigned long bx = (x & (1 << i));
9         unsigned long by = (y & (1 << i));
10        unsigned long bz = (z & (1 << i));
11        auto j = 2*i;
12        ans = ans | (bx << j)
13                  | (by << (j+1))
14                  | (bz << (j+2));
15        ++i;
16    }
17    return ans;
18 }
```

```
1 auto morton(unsigned long x, unsigned long y,
2              unsigned long z)
3 {
4     constexpr unsigned long mask[] {
5         0x9249249249249249, // 0b100100100...1001001
6         0x2492492492492492, // 0b001001001...0010010
7         0x4924924924924924 // 0b010010010...0100100
8     };
9     // On x86 ...
10    return _pdep_u64(x, mask[0])
11           | _pdep_u64(y, mask[1])
12           | _pdep_u64(z, mask[2]);
13 }
```

Intrinsics : high(er) level interface to CPU instructions

Interface to SSE and AVX registers

- include "nmmintrin.h" (SSE 4.2) or "immintrin.h" (AVX)
- `__m128i` : integer register with 128 bits
- `__m128` : 128 bits with 4 packed floats
- `__m128d` : 128 bits with 2 doubles
- `__m256i` : 256 bit octint
- `__m256` : octfloat
- `__m256d` : quaddouble
- [Intel x86 optimization manual](#)
- [Intel intrinsics guide](#)

Interface to SSE and AVX operations

- `_mm_add_ps (__m128, __m128)`
- `_mm_sub_ps (, , _mm_sqrt_ps () ...)`
- `_mm256_add_pd (__m256d, __m256d)`
- Convention:
`_(sizecode)_(operation)_(suffix)`
 - sizecode is mm for SSE, mm256 for AVX and mm512 for AVX512
 - operation is "add", "sub", "mul" etc.
 - suffix indicates data type in the register arguments. ps => float, pd => double, epi32 => 32 bit signed int, epu32 => 32 bit unsigned int

Example: direct use of intrinsics

```
1 float sprod_sane(size_t n, const float a[],  
2                   const float b[]) {  
3     double res{};  
4     for (size_t i=0UL; i<n; ++i)  
5         res += a[i] * b[i];  
6     return res;  
7 }
```

- (RHS) Feels C++'ish, but commits too much to machine level details
- This is just an example to show what bare intrinsics based code looks like. It is almost never a good idea to use raw intrinsics in application code. It's lazy and dangerous, and ends up costing you more time anyway.
- Beware of persistant superstition surrounding abstractions. Overreaching advice against compile time abstractions such as static polymorphism, template or constexpr metaprogramming is usually bad advice. Always check.

```
1 float sse_sprod(size_t n, const float a[],  
2                  const float b[]) {  
3     assert(0 == n % 4); // simplifying assumption  
4     __m128 res, prd, ma, mb;  
5     res = _mm_setzero_ps();  
6     for (size_t i=0; i<n; i += 4) {  
7         ma = _mm_loadu_ps(&a[i]);  
8         mb = _mm_loadu_ps(&b[i]);  
9         prd = _mm_mul_ps(ma, mb);  
10        res = _mm_add_ps(prd, res);  
11    }  
12    prd = _mm_setzero_ps();  
13    res = _mm_hadd_ps(res, prd);  
14    res = _mm_hadd_ps(res, prd); // not a typo!  
15    float tmp;  
16    _mm_store_ss(&tmp, res);  
17    return tmp;  
18 }
```

Wrapping intrinsics in zero (/low) cost abstractions

```
1 #include <immintrin.h>
2 union alignas(32) QuadDouble {
3     __m256d mm;
4     double d[4];
5     QuadDouble(__m256d oth) : mm{oth} {}
6     constexpr QuadDouble(double x, double y, double z=0., double t=0.) : d{x, y, z, t} {}
7
8     void aligned_load(double * v) {
9         assert(get_alignment(v) >= 32);
10        mm = _mm256_load_pd(v);
11    }
12    void unaligned_load(double * v) { mm = _mm256_loadu_pd(v); }
13
14    [[nodiscard]] auto operator[](unsigned i) const -> double { return d[i%4]; }
15    auto operator[](unsigned i) -> double & { return d[i%4]; }
16
17    void operator=(double x) { mm = _mm256_broadcast_sd(&x); }
18    [[nodiscard]] auto horizontal_add() const -> double { return d[0] + d[1] + d[2] + d[3]; }
19};
```

Wrapping intrinsics in zero (/low) cost abstractions

```
1 auto get_alignment(void * var) {
2     auto n = reinterpret_cast<unsigned long>(var);
3     return (-n) & n;
4 }
5 auto operator+(QuadDouble a, QuadDouble b) -> QuadDouble {return _mm256_add_pd(a.mm, b.mm);}
6 auto operator-(QuadDouble a, QuadDouble b) -> QuadDouble {return _mm256_sub_pd(a.mm, b.mm);}
7 auto operator*(QuadDouble a, QuadDouble b) -> QuadDouble {return _mm256_mul_pd(a.mm, b.mm);}
8 auto operator/(QuadDouble a, QuadDouble b) -> QuadDouble {return _mm256_div_pd(a.mm, b.mm);}
9
10 auto main() -> int
11 {
12     QuadDouble a{3.1}, b{0.2, 5.4, 2.1, 9.8};
13     auto c = a * b - (a / b);
14     return c[2] < -1.;
15 }
```

Wrapping intrinsics in zero(/low) cost abstractions

- Notational simplification, more readable and maintainable code, at no (or rather low) run time cost
- Need to wrap all operations used by your application (but only those)
- Need to hide vendor specific differences

```
auto operator*(QuadDouble a, QuadDouble b) -> QuadDouble
{
    return _mm256_mul_pd(a.mm, b.mm);
}

1 # With Clang 7
2 operator*(QuadDouble, QuadDouble): # @operator*(QuadDouble, QuadDouble)
3     vmulpd  ymm0, ymm0, ymm1
4     ret
```

Note on alignment: Dynamically allocated arrays of our abstraction can cause unexpected crashes for C++98 ... C++14, as the `new` operator could not align “over aligned” types on the heap. This was fixed in C++17, and optionally provided for C++11 and C++14 with compiler flags (GCC: `-faligned-new` Clang: `-faligned-allocation`).

COMPILER EXPLORER

Add... More ▾

x86-64 gcc 8.2 (Editor #1, Compiler #1) C++ X

-O3 -std=c++

A ▾

11010 .LX0: lib.f: .text // \s+ Intel

Libraries + Add new... ⚙ Add tool... ▾

```

38     return _mm256_mul_pd(a.mn, b.mn);
39 }
40 inline QuadDouble operator/(QuadDouble a, QuadDouble b)
41 {
42     return _mm256_div_pd(a.mn, b.mn);
43 }
44
45 void test(QuadDouble py4, QuadDouble px4,
46           QuadDouble &oy4, QuadDouble ox4)
47 {
48     const QuadDouble xoffs4{0.55};
49     oy4 = -(py4 * px4 - px4*px4 + xoffs4);
50 }
51
52 void silly(_m256d py4, _m256d px4,
53            _m256d &oy4, _m256d ox4)
54 {
55     constexpr double cxoff = 0.55;
56     const _m256d xoffs4{_mm256_broadcast_sd(&cxoff)};
57     oy4 = _mm256_sub_pd(
58         _mm256_setzero_pd(),
59         _mm256_add_pd(
60             _mm256_sub_pd(
61                 _mm256_mul_pd(py4, py4),
62                 _mm256_mul_pd(px4, px4)
63             ),
64             xoffs4
65         )
66     );
67 }
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98

```

x86-64 clang (trunk) (Editor #1, Compiler #3) C++ X

-std=c++17 -O3

A ▾

11010 .LX0: lib.f: .text // \s+ Intel

Libraries + Add new... ⚙ Add tool... ▾

```

1 get_alignment(void const*):
2     blsi eax, edi
3     ret
4 test(QuadDouble, QuadDouble, QuadDouble&, QuadDouble):
5     vmovapd ymm1, YMMWORD PTR [rsp+8]
6     vmovapd ymm0, YMMWORD PTR [rsp+48]
7     vmulpd ymm1, ymm1, ymm1
8     vfmadd132pd ymm0, ymm1, ymm0
9     vbroadcastsd ymm1, QWORD PTR .LC0[rip]
10    vadddp ymm0, ymm0, ymm1
11    vxorpd xmm1, xmm1, xmm1
12    vsubpd ymm0, ymm1, ymm0
13    vmovapd YMMWORD PTR [rdi], ymm0
14    vzeroupper
15    ret
16 silly(double __vector(4), double __vector(4), double __vect
17     push rbp
18     vmulpd ymm0, ymm0, ymm0
19     mov rbp, rsp
20     and rsp, -32
21     vfmadd231pd ymm0, ymm1, ymm1
22     vbroadcastsd ymm1, QWORD PTR .LC0[rip]
23     vadddp ymm0, ymm0, ymm1
24     vxorpd xmm1, xmm1, xmm1
25     vsubpd ymm0, ymm1, ymm0
26     vmovapd YMMWORD PTR [rdi], ymm0
27     vzeroupper
28     leave
29     ret
30 .LC0:
31     .long 2576988378
32     .long 1071749529

```

x86-64 clang (trunk) (Editor #1, Compiler #3) C++ X

-std=c++17 -O3

A ▾

11010 .LX0: lib.f: .text // \s+ Intel

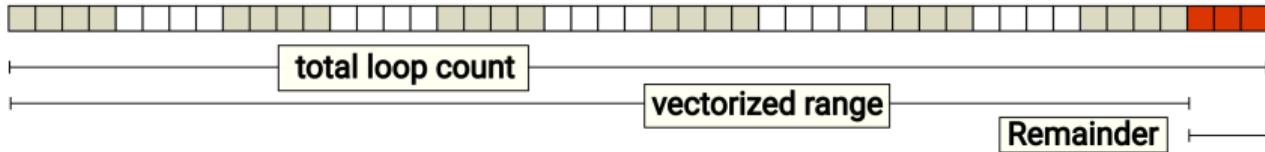
Libraries + Add new... ⚙ Add tool... ▾

```

1 get_alignment(void const*): # @get_alignment
2     blsi eax, edi
3     ret
4 .LCPI1_0:
5     .quad 4663129179135383962 # double 0.5500000000000000
6 test(QuadDouble, QuadDouble, QuadDouble&, QuadDouble):
7     vmulpd ymm1, ymm1, ymm1
8     vfmsub231pd ymm1, ymm1, ymm0 # ymm1 = (ymm0 * ymm1)
9     vadddp ymm0, ymm1, qword ptr [rip + .LCPI1_0]{1to4}
10    vxorpd xmm1, xmm1, xmm1
11    vsubpd ymm0, ymm1, ymm0
12    vmovapd ymmword ptr [rdi], ymm0
13    vzeroupper
14    ret
15 .LCPI2_0:
16     .quad 4663129179135383962 # double 0.5500000000000000
17 silly(double __vector(4), double __vector(4), double __vect
18     vmulpd ymm1, ymm1, ymm1
19     vfmsub231pd ymm1, ymm0, ymm0 # ymm1 = (ymm0 * ymm1)
20     vadddp ymm0, ymm1, qword ptr [rip + .LCPI2_0]{1to4}
21     vxorpd xmm1, xmm1, xmm1
22     vsubpd ymm0, ymm1, ymm0
23     vmovapd ymmword ptr [rdi], ymm0
24     vzeroupper
25     ret

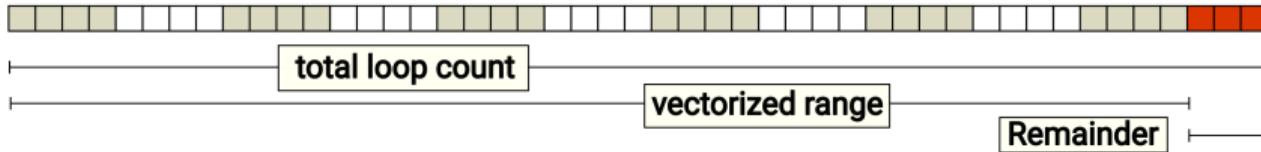
```

Using our DIY SIMD library



```
1 // examples/diy/daxpy.cc
2 void daxpy_explicit(const std::vector<double> & x, std::vector<double> & y, double a) {
3     QuadDouble bx{0.}, by{0.};
4     const QuadDouble ba{a};
5     unsigned long vsize = x.size() - x.size() % 4;
6     const double * xptr0 = x.data();
7     const double * xptr1 = x.data() + vsize;
8     double * yptr = y.data();
9     for (; xptr0 != xptr1; xptr0 += 4, yptr += 4) {
10         bx.unaligned_load(xptr0);
11         by.unaligned_load(yptr);
12         by = by + bx * ba;
13         by.unaligned_store(yptr);
14     }
15     for (auto i=vsize; i<x.size(); ++i) y[i] += a* x[i];
16 }
```

Using our DIY SIMD library



```
1 // examples/diy/sprod.cc
2 #include "QuadDouble.hh"
3 auto sprod_explicit(size_t n, const double x[], const double y[]) -> double {
4     QuadDouble bx{0.}, by{0.}, tot{0.};
5     unsigned long vsize = n - n % 4;
6     const double * xptr0 = x;
7     const double * xptr1 = x + vsize;
8     for (; xptr0 != xptr1; xptr0 += 4, y += 4) {
9         bx.unaligned_load(xptr0);
10        by.unaligned_load(y);
11        tot = tot + bx * by;
12    }
13    auto res = tot.horizontal_add();
14    for (auto i = vsize; i < x.size(); ++i) res += x[i] * y[i];
15    return res;
16 }
```

Conditional selection using masks and blend



SIMD ?

ymm0

```
if (x[i] > 3.1)
    do_one_thing();
else
    do_something_else();
```

ymm1

- Different lanes in a SIMD register can not execute different instructions \Rightarrow problems with general branched code

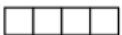
- Various kinds of conditional selection can be executed as single instructions: picking out the larger of the corresponding elements between two arrays
- Large number of “masked” instructions, e.g.,
`_mm256_mask_[op]_[type]` and
`_mm256_maskz_[op]_[type]`. A mask is a bit field of the appropriate size storing 0 or 1. The `...maskz...` variants zero out the positions in the destination corresponding to the entries where the mask is unset. The `...mask...` variants take an additional `src` argument, and copy the result from there, if the mask is unset. Both store the result of the computation if the mask bit is in fact set.
- Masked selection between two alternatives is also possible using “blend” instructions. Search for “blend” in the Intel intrinsics guide

Conditional selection using masks and blend



vmaxpd ymmX, ymmY, ymmZ

```
if (x[i] > y[i])
    z[i] = x[i];
else
    z[i] = y[i];
```

ymm0 

ymm1  _mm256_max_pd(,)

- Different lanes in a SIMD register can not execute different instructions \Rightarrow problems with general branched code

- Various kinds of conditional selection can be executed as single instructions: picking out the larger of the corresponding elements between two arrays
- Large number of “masked” instructions, e.g.,
`_mm256_mask_[op]_[type]` and
`_mm256_maskz_[op]_[type]`. A mask is a bit field of the appropriate size storing 0 or 1. The `...maskz...` variants zero out the positions in the destination corresponding to the entries where the mask is unset. The `...mask...` variants take an additional `src` argument, and copy the result from there, if the mask is unset. Both store the result of the computation if the mask bit is in fact set.
- Masked selection between two alternatives is also possible using “blend” instructions. Search for “blend” in the Intel intrinsics guide

Conditional selection using masks and blend

a3	a2	a1	a0
b3	b2	b1	b0
c3	c2	c1	c0
1	0	0	1

`fmadd213pd source1, source2, source3, mask`

a3*b3+c3	a2	a1	a0*b0+c0
1	0	0	1

- Different lanes in a SIMD register can not execute different instructions \Rightarrow problems with general branched code

- Various kinds of conditional selection can be executed as single instructions: picking out the larger of the corresponding elements between two arrays
- Large number of “masked” instructions, e.g.,
`_mm256_mask_[op]_[type]` and
`_mm256_maskz_[op]_[type]`. A mask is a bit field of the appropriate size storing 0 or 1. The
`...maskz...` variants zero out the positions in the destination corresponding to the entries where the mask is unset. The
`...mask...` variants take an additional `src` argument, and copy the result from there, if the mask is unset. Both store the result of the computation if the mask bit is in fact set.
- Masked selection between two alternatives is also possible using “blend” instructions. Search for “blend” in the Intel intrinsics guide

Conditional selection using masks and blend

a3	a2	a1	a0
----	----	----	----

b3	b2	b1	b0
----	----	----	----

m3	m2	m1	m0
----	----	----	----

`vblendpd source1, source2, mask`

m3 ? b3 : a3	m2 ? b2 : a2	m1 ? b1 : a0	m0 ? b0 : a0
--------------	--------------	--------------	--------------

- Different lanes in a SIMD register can not execute different instructions \Rightarrow problems with general branched code

- Various kinds of conditional selection can be executed as single instructions: picking out the larger of the corresponding elements between two arrays
- Large number of “masked” instructions, e.g.,
`_mm256_mask_[op]_[type]` and
`_mm256_maskz_[op]_[type]`. A mask is a bit field of the appropriate size storing 0 or 1. The `...maskz...` variants zero out the positions in the destination corresponding to the entries where the mask is unset. The `...mask...` variants take an additional `src` argument, and copy the result from there, if the mask is unset. Both store the result of the computation if the mask bit is in fact set.
- Masked selection between two alternatives is also possible using “blend” instructions. Search for “blend” in the Intel intrinsics guide

Creating and manipulating masks

- Masks are bit fields. Conceptually they are like arrays of boolean variables with the same number of elements as the corresponding SIMD register
- Many SIMD functions return mask types:
 - `_mm256_cmpeq_epi32_mask (__m256i, __m256i)` : element wise comparison. All corresponding bits of the mask set if equality comparison returns true for an element
 - `_mm256_cmplt_epi32_mask (__m256i, __m256i)` : As with cmpeq, but for “less than” comparison
- Masks can be combined with usual bitwise operations `_mm256_and_pd` , `_m256_or_pd` etc.

```
1 auto m1 = _mm256_cmpge_epi32_mask(vi, vj);
2 auto m2 = _mm256_cmpeq_epi32_mask(vk, _mm256_setzero_epi32());
3 auto mask = _mm256_and_si256(m1, m2);
4 res = _mm256_fmask_fmadd_pd(x, mask, y, z);
```

Great! Now, what about AVX512 ? Power ? ARM ?

- Application code can operate using the abstraction
- Architecture specific details can be hidden inside the SIMD library
- No run-time indirection is needed. The compiler can be made to choose one specific version (macros, template specializations ...)
- The author(s) of the SIMD library have to deal with the available capabilities in different instruction sets
- The library can also provide additional benefits: SIMD implementation of widely used functions, e.g., trigonometric, exponential functions

Open source SIMD libraries

Vector Class Library

- Agner Fog's SIMD library : Vector Class Library
- Light weight abstraction around SIMD types and functions
- Does not require a great deal of C++ knowledge to use. Uses limited metaprogramming to choose among alternative implementations
- Interface still binds you to specific vector sizes, e.g., Vec4d, although depending on the available instruction set, different implementations are provided
- Does not provide the dependent types and static constants for use with more sophisticated C++ usage (e.g., template/constexpr metaprogramming)

```
1 #include "vectorclass.h"
2
3 double sprod_vcl(size_t n, const double x[],
4                   const double y[])
5 {
6     Vec4d bx{0.}, by{0.}, tot{0.};
7     unsigned long vsize = n - n % 4;
8     const double * xptr0 = x;
9     const double * xptr1 = x + vsize;
10    for (; xptr0 != xptr1; xptr0 += 4, y += 4) {
11        bx.load(xptr0);
12        by.load(y);
13        tot = tot + bx * by;
14    }
15    auto res = horizontal_add(tot);
16    for (auto i = vsize; i < n; ++i) {
17        res += x[i] * y[i];
18    }
19    return res;
20 }
```

Vector class library

Vec16c 16 8-bit integers

Vec16uc 16 8-bit unsigned integers

Vec8s 8 16-bit integers

Vec8us 8 16-bit unsigned integers

Vec4i 4 32-bit integers

Vec4ui 4 32-bit unsigned integers

Vec8i 8 32-bit integers

Vec8ui 8 32-bit unsigned integers

Vec4q 4 64-bit integers

Vec4uq 4 64-bit unsigned integers

Vec4d 4 64-bit floating point numbers

Vec8d 8 64-bit floating point numbers

Vec4db 4 element boolean vector for use with Vec4d

```
1 int list[]{10,9,8,7,6,5,4,3,2,1,0};  
2 Vec4i a = gather4i<0,3,5,2>(list);  
3 // a = [10,7,5,8]  
4 Vec4i b;  
5 b.load_partial(2, list+4);  
6 // load 2 elements starting from the 5'th  
7 a.cutoff(3);  
8 // a = [10,7,5,0]  
9  
10 Vec4i c = (b >> 2) - a;  
11 // Usual arithmetic and bit shifts for integers  
12 Vec4ib mask = c > a;
```

XSIMD

- C++ wrappers for SIMD intrinsics from “QuantStack”. Include only. BSD-3-Clause license.

```
git clone https://github.com/QuantStack/xsimd.git
```

- Abstractions for batches of values for SIMD calculations, e.g., `xsimd::batch<double, 4>`

```
xsimd::batch<double, 4> x{1.,2.,3.,4.}, y{4.,3.,2.,1.};  
std::cout << x + y << "\n";
```

- Vectorized forms of commonly used mathematical functions, such as trigonometric, exponential functions, error functions, e.g., `xsimd::asin(xsimd::batch<double, 4>)`,

```
xsimd::exp(xsimd::batch<double, 4>)
```

- Regular arithmetic operations along with fma functions, e.g., `xsimd::fma(a, x, y)`

- Auto-detection and parametrisation based on available instruction set, e.g., based on vector width,

```
xsimd::simd_type<double>
```

- Aligned allocator:

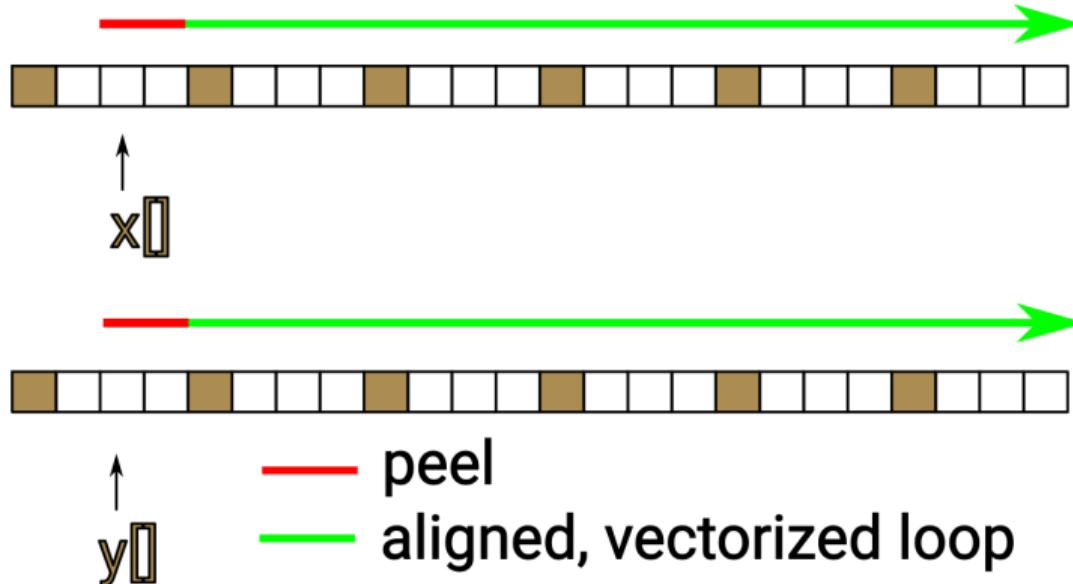
```
template <typename T> using myvector = std::vector<T,XSIMD_DEFAULT_ALLOCATOR(T)>;  
myvector<double> V(1000000, 1.2); // Aligned to cache line
```

```
1 void daxpy(double a, const std::vector<double> & x, const std::vector<double> & y,
2             std::vector<double> & res) {
3     for (size_t i = 0UL; i < x.size(); ++i) {
4         res[i] = a * x[i] + y[i];
5     }
6 }
7
8 void daxpy_xsimd(double a, const std::vector<double> & x, const std::vector<double> & y,
9                   std::vector<double> & res) {
10    using namespace xsimd;
11    constexpr auto vwidth = batch<double, 4>::size;
12    const auto vreps = x.size() - x.size() % vwidth;
13
14    for (size_t i = 0UL; i < vreps; i += vwidth) {
15        batch<double, 4> xblk = load_unaligned(&x[i]);
16        auto yblk = load_unaligned(&y[i]);
17        store_unaligned(&res[i], a * xblk + yblk);
18    }
19
20    for (size_t i = vreps; i < x.size(); ++i) { res[i] = a * x[i] + y[i]; }
21 }
```

```
1 void daxpy(double a, const std::vector<double> & x, const std::vector<double> & y,
2             std::vector<double> & res) {
3     for (size_t i = 0UL; i < x.size(); ++i) {
4         res[i] = a * x[i] + y[i];
5     }
6 }
7
8 void daxpy_xsimd(double a, const std::vector<double> & x, const std::vector<double> & y,
9                   std::vector<double> & res) {
10    using namespace xsimd;
11    constexpr auto vwidth = simd_type<double>::size;
12    const auto vreps = x.size() - x.size() % vwidth;
13
14    for (size_t i = 0UL; i < vreps; i += vwidth) {
15        simd_type<double> xblk = load_unaligned(&x[i]);
16        auto yblk = load_unaligned(&y[i]);
17        store_unaligned(&res[i], a * xblk + yblk);
18    }
19
20    for (size_t i = vreps; i < x.size(); ++i) { res[i] = a * x[i] + y[i]; }
21 }
```

```
1 void daxpy(double a, const std::vector<double> & x, const std::vector<double> & y,
2             std::vector<double> & res) {
3     for (size_t i = 0UL; i < x.size(); ++i) {
4         res[i] = a * x[i] + y[i];
5     }
6 }
7 template <typename T> using myvector = std::vector<T,XSIMD_DEFAULT_ALLOCATOR(T)>;
8 void daxpy_xsimd(double a, const myvector<double> & x, const myvector<double> & y,
9                   myvector<double> & res) {
10    using namespace xsimd;
11    constexpr auto vwidth = simd_type<double>::size;
12    const auto vreps = x.size() - x.size() % vwidth;
13
14    for (size_t i = 0UL; i < vreps; i += vwidth) {
15        simd_type<double> xblk = load_aligned(&x[i]);
16        auto yblk = load_aligned(&y[i]);
17        store_aligned(&res[i], a * xblk + yblk);
18    }
19
20    for (size_t i = vreps; i < x.size(); ++i) { res[i] = a * x[i] + y[i]; }
21 }
```

Alignment and SIMD operations



Alignment and SIMD operations



x



y

?

How many elements would you peel now ?

Alignment and SIMD operations



↑
x[]



↑
y[]

unaligned load/store : vmovupd

On Intel processors > Haswell, penalty low

SIMD with complex numbers

- `std::complex<T>` has a fixed data layout, (*real, imag*) to be compatible with C
- Arrays of complex numbers have the real parts at non-adjacent, but statically predictable, locations (same applies to the imaginary parts)
- Many ways to code vectorized operations on complex numbers
- VCL (`Complex4d` for 2 complex numbers) and XSIMD (`simd_type<complex<double>>`) both have abstractions for working with complex numbers
- Without such abstractions to aid us, explicit SIMD programming with complex number would be needlessly complicated

```
1 #include <xsimd/xsimd.hpp>
2 #include <complex>
3 #include <vector>
4 using namespace std;
5
6 void caxpy_xsimd(complex<double> a,
7                   vector<complex<double>> &x,
8                   const vector<complex<double>> &y)
9 {
10    using b_type =
11        xsimd::simd_type<complex<double>>;
12    b_type c = xsimd::set_simd(a);
13    b_type xl, yl;
14    for (size_t i=0; i<x.size(); i+=b_type::size()) {
15        xl.load_unaligned(&x[i]);
16        yl.load_unaligned(&y[i]);
17        xl = c * xl + yl;
18        xl.store_unaligned(&x[i]);
19    }
20 }
```

Exercise 2.16:

In the folder `examples/SIMD`, you will find several versions of a few short functions.

- Only one example in each directory produces an executable `exvol.cc`, the rest can be compiled to object files. They are just to demonstrate syntax.
- The DIY version does not require any libraries to compile, although it does need `immintrin.h`, which should be found in your system
- You should compile with the best available instruction set on your system (`-march=native` for GCC and Clang) and with optimization for speed
- For the examples with VCL or XSIMD, you will need to pass `-I /path/to/VCLHEADERS` or `-I /path/to/xsimdroot/include` to the compiler. There is nothing to link.

Example compile command :

```
g++ -std=c++20 -O3 -march=native -I ~/path/to/xsimd/include exvol.cc -o exvol.g
```

Summary

- Directly coding with SIMD types exposes algorithmic challenges concerning vectorization
- We are much more directly in control
- Quite often, correctly done OpenMP will bring you most of the benefits, but, knowing how to work with intrinsics gives you a fallback option when the simple approach fails. At the very least, when you try to vectorize yourself, you might see why OpenMP didn't do as you had hoped.
- If you work with C++, use its strengths: strive for zero-overhead abstractions instead of resigning to a life of verbose and error-prone misery
- Alternatively, use a SIMD library with a compatible license
 - They already exist, and others have already created the necessary abstractions
 - They support multiple instruction sets and CPU architectures
 - Often come with vectorized versions of common mathematical functions
- No substitute for play time. Take an open source SIMD library or make your own. Ask yourself, "what can I do with this?", and explore.