High-performance Scientific Computing in C++

20–21 June 2017 | Sandipan Mohanty (s.mohanty@fz-juelich.de)
Introduction
C++

Express ideas in code

- Specify actions to be executed by the machine

---

C++

Express ideas in code

- Specify actions to be executed by the machine
- Concepts to use when thinking what can be done

\(^1\text{Chapter 1, The C++ Programming Language, 4}^{th}\ \text{Edition, Bjarne Stroustrup}\)
Express ideas in code

- Specify actions to be executed by the machine
- Concepts to use when thinking what can be done
- Direct mappings of built in operations and types to hardware

---

C++

Express ideas in code

- Specify actions to be executed by the machine
- Concepts to use when thinking what can be done
- Direct mappings of built-in operations and types to hardware
- Affordable and flexible abstraction mechanisms

---

C++

### Express ideas in code

- Specify actions to be executed by the machine
- Concepts to use when thinking what can be done
- Direct mappings of built in operations and types to hardware
- Affordable and flexible abstraction mechanisms

C++ is a language for developing and using elegant and efficient abstractions

---

<sup>1</sup>Chapter 1, The C++ Programming Language, 4<sup>th</sup> Edition, Bjarne Stroustrup
## Goals

- General purpose: no specialization to specific usage areas
C++

Goals

- General purpose: no specialization to specific usage areas
- No over simplification that precludes direct expert level use of hardware
C++

Goals

- General purpose: no specialization to specific usage areas
- No over simplification that precludes direct expert level use of hardware
- Leave no room for a lower level language
C++

Goals

- General purpose: no specialization to specific usage areas
- No over simplification that precludes direct expert level use of hardware
- Leave no room for a lower level language
- What you don’t use, you don’t pay for
C++ in scientific computing

- Handle complexity and do it fast
C++ in scientific computing

- Handle complexity and do it fast
- Use the compiler to catch implementation logic errors
C++

C++ in scientific computing

- Handle complexity and do it fast
- Use the compiler to catch implementation logic errors
- Performance optimisation is very important: application return time may decide whether or not a research problem is even considered
C++

C++ in scientific computing

- Handle complexity and do it fast
- Use the compiler to catch implementation logic errors
- Performance optimisation is very important: application return time may decide whether or not a research problem is even considered
  - Smart algorithms
C++

C++ in scientific computing

- Handle complexity and do it fast
- Use the compiler to catch implementation logic errors
- Performance optimisation is very important: application return time may decide whether or not a research problem is even considered
  - Smart algorithms
  - Hardware aware translation of ideas into code
C++ in scientific computing

- Handle complexity and do it fast
- Use the compiler to catch implementation logic errors
- Performance optimisation is very important: application return time may decide whether or not a research problem is even considered
  - Smart algorithms
  - Hardware aware translation of ideas into code
  - Profiling and tuning
Know your hardware
Warm up!
Get your session ready!

- We will use the following shorthands:

```bash
G = g++ -O3 -pedantic -Wall \n   -std=c++14
A = clang++ -O3 -pedantic -Wall \n   -std=c++14 -stdlib=libc++
```

- Go to examples and open `warmup.cc`

- Compile using our short cut: `G warmup.cc`.

- What will this program do?

```c
int main()
{
    auto high = 100'000'000.0f;
    auto low = 99'999'990.0f;
    do {
        std::cout << high << "\n";
        high -= 1;
    } while (high > low);
}
```
Get your session ready!

- We will use the following shorthands:

  ```
  G = g++ -O3 -pedantic -Wall \\
  -std=c++14
  A = clang++ -O3 -pedantic -Wall \\
  -std=c++14 -stdlib=libc++
  ```

- Go to examples and open `warmup.cc`
- Compile using our short cut: `G warmup.cc`

```c
int main()
{
    auto high = 100’000’000.0f;
    auto low = 99’999’990.0f;
    do {
        std::cout << high << "\n";
        high -= 1;
    } while (high > low);
}
```

- What will this program do?
- Type `./warmup.g` to find out!
Get your session ready!

- We will use the following shorthands:

```cpp
G = g++ -O3 -pedantic -Wall \n   -std=c++14
A = clang++ -O3 -pedantic -Wall \n   -std=c++14 -stdlib=libc++
```

- Go to examples and open `warmup.cc`

- Compile using our short cut: `G warmup.cc`

```cpp
int main()
{
    auto high = 100’000’000.0f;
    auto low = 99’999’990.0f;
    do {
        std::cout << high << "\n";
        high -= 1;
    } while (high > low);
}
```

- What will this program do?
- Type `./warmup.g` to find out!
- Ready?
Floating point numbers

Area of a triangle of sides $a$, $b$ and $c$...

- Heron’s formula (Metaica, Heron of Alexandria, $\approx 60$ CE)

\[
\begin{align*}
  s &= \frac{a + b + c}{2} \\
  \Delta &= \sqrt{s \times (s - a) \times (s - b) \times (s - c)}
\end{align*}
\]
Floating point numbers

Area of a triangle of sides $a$, $b$ and $c$...

- Heron’s formula (Metrica, Heron of Alexandria, $\approx 60$ CE)

\[
s = \frac{a + b + c}{2} \quad \Delta = \sqrt{s \times (s - a) \times (s - b) \times (s - c)}
\]


\[
a \geq b \geq c \quad \Delta = \frac{1}{4} \sqrt{(a + (b + c)) \times (c - (a - b)) \times (c + (a - b)) \times (a + (b - c))}
\]
Floating point numbers

Mathematically, both calculate the same thing

```
const auto a = 5.0f;
const auto b = 4.0f;
const auto c = 3.0f;
std::cout << "Heron’s formula = "
    << area_heron(a,b,c) << "\n";
std::cout << "Kahan’s formula = "
    << area_kahan(a,b,c) << "\n";
```

Heron’s formula = 6
Kahan’s formula = 6

Example 1:

`examples/area.cc` contains an implementation of both these formulae. Change the sides of the triangles a few times. Try the values in the commented out lines.

See also: CppCon 2015: John Farrier “Demystifying Floating Point”
Floating point numbers

```cpp
const auto a = 100'000.000'00f;
const auto b = 99'999.999'79f;
const auto c = 0.000'29f;
std::cout << "Heron’s formula = "
    << area_heron(a,b,c) << "\n";
std::cout << "Kahan’s formula = "
    << area_kahan(a,b,c) << "\n";
```

- Mathematically, both calculate the same thing
- If the triangle becomes very long and thin though, weird things happen

Example 1:

`examples/area.cc` contains an implementation of both these formulae. Change the sides of the triangles a few times. Try the values in the commented out lines.

See also: CppCon 2015: John Farrier “Demystifying Floating Point”
Floating point numbers

Mathematically, both calculate the same thing

If the triangle becomes very long and thin though, weird things happen

Example 1:

`examples/area.cc` contains an implementation of both these formulae. Change the sides of the triangles a few times. Try the values in the commented out lines.

See also: CppCon 2015: John Farrier “Demystifying Floating Point”
Floating point numbers

```
const auto a = 100'000.000'00f;
const auto b = 99'999.999'79f;
const auto c = 0.000'29f;
std::cout << "Heron’s formula = "
  << area_heron(a,b,c) << "\n";
std::cout << "Kahan’s formula = "
  << area_kahan(a,b,c) << "\n";
```

- Mathematically, both calculate the same thing
- If the triangle becomes very long and thin though, weird things happen
- Correct answer is 10.

Example 1:

```
examples/area.cc contains an implementation of both these formulae. Change the sides of the triangles a few times. Try the values in the commented out lines.
```

See also: CppCon 2015: John Farrier “Demystifying Floating Point”
Representation of floating point numbers

\[-1^s \times 1.mantissa \times 2^{exponent}\]

- It is enough to store the coloured parts. We win an extra bit of precision in the mantissa by skipping the 1 before the decimal point.
Representation of floating point numbers

\[ -1^s \times 1.mantissa \times 2^{exponent} \]

- It is enough to store the coloured parts. We win an extra bit of precision in the mantissa by skipping the 1 before the decimal point.
- For a fixed exponent, there are \(2^{23}\) different floating point numbers. \(\Rightarrow\) There are as many floats between \(2^{-11}\) and \(2^{-10}\) as there are between 1024 and 2048.
Representations of floating point numbers

$-1^s \times 1.mantissa \times 2^{exponent}$

- It is enough to store the coloured parts. We win an extra bit of precision in the mantissa by skipping the 1 before the decimal point.
- For a fixed exponent, there are $2^{23}$ different floating point numbers. $\Rightarrow$ There are as many floats between $2^{-11}$ and $2^{-10}$ as there are between 1024 and 2048.
- By contrast, integral types have a uniform density throughout their range.
Representation of floating point numbers

\[ -1^s \times 1.mantissa \times 2^{exponent} \]

- Zero = all bits 0. One?
Representation of floating point numbers

\[-1^s \times 1.mantissa \times 2^{exponent}\]

- Zero = all bits 0. One?
- Exponent is stored shift-127 encoded. So,
  \[1 \equiv [0][0111111][000000000000000000000000]\]
Representation of floating point numbers

\[ -1^s \times 1.\text{mantissa} \times 2^{\text{exponent}} \]

- Zero = all bits 0. One?
- Exponent is stored shift-127 encoded. So,
  \[ 1 \equiv [0][01111111][000000000000000000000000] \]
- To maintain our sanity, we will write it as
  \[ 1 \equiv [0][2^0][00000000000000000000000000000000] \]
Mental exercise: we have two decimal numbers in scientific notation $9.78 \times 10^2$, and $1.0 \times 10^{-1}$. How will you add them?
Floating point numbers

- Mental exercise: we have two decimal numbers in scientific notation $9.78 \times 10^2$, and $1.0 \times 10^{-1}$. How will you add them?
- You shift the decimal point in one of them until the exponents are the same, and then add the mantissas: $9.78 \times 10^2 + 0.001 \times 10^2$. Digits in the smaller number are pushed to the right.
Floating point numbers

- \(1 \equiv [0][2^0][00000000000000000000000000000000]\)
Floating point numbers

- $1 \equiv [0](2^0)[000000000000000000000000000000000]$
- What is the smallest representable $n$, with $n > 1$?
Floating point numbers

- 1 \equiv [0][(2^0)][00000000000000000000000]
- What is the smallest representable \( n \), with \( n > 1 \) ?
- \[ [0][(2^0)][00000000000000000000001] \] with the mantissa changing by \( 2^{-23} \approx 0.0000001192092895507813 \)
Floating point numbers

- \[1 \equiv [0]([2^0])[000000000000000000000000]\]

- What is the smallest representable \(n\), with \(n > 1\) ?

- \([0]([2^0])[0000000000000000000000000001]\) with the mantissa changing by \(2^{-23} \approx 0.0000001192092895507813\)

- What is \(2.0\) ? \([0]([2^1])[0000000000000000000000000000000000000000]\). What if you add these two? What information about the smaller number can we retain?
Floating point numbers

- What is the smallest representable $n$, with $n > 1$?
- $[0][(2^0)][0000000000000000000000001]$. Mantissa changes by $2^{-23} \approx 0.0000001192092895507813$.
- This quantity depends on the floating point type. In C++, you can retrieve it `std::numeric_limits<T>::epsilon()`.
Floating point numbers

- What is the smallest representable $n$, with $n > 1$?
  - $[0][2^0][0000000000000000001]$. Mantissa changes by $2^{-23} \approx 0.0000001192092895507813$.

- This quantity depends on the floating point type. In C++, you can retrieve it `std::numeric_limits<T>::epsilon()`.

- Two quantities with exponent 0 cannot be distinguished in this representation, if they differ by less than `epsilon`.
Floating point numbers

- What is the smallest representable $n$, with $n > 1$?
  
  - $[0][(2^0)][000000000000000000000001]$. Mantissa changes by $2^{-23} \approx 0.0000001192092895507813$.

- This quantity depends on the floating point type. In C++, you can retrieve it `std::numeric_limits<T>::epsilon()`.

- Two quantities with exponent 0 can not be distinguished in this representation, if they differ by less than `epsilon`.

- In an expression like $(\text{big}+\text{small})-\text{big}$, if big and small differ by more than 23 in exponent, all information about small is lost, and we get a 0. $2^{23} = 8388608$. 
Floating point numbers

- Floating point numbers with all bits in the exponent field at 0, are said to be “denormalised” (remember the shift-127 encoding)
Floating point numbers

- Floating point numbers with all bits in the exponent field at 0, are said to be “denormalised” (remember the shift-127 encoding)
- Not enough bits to represent such small quantities.
Floating point numbers

- Floating point numbers with all bits in the exponent field at 0, are said to be “denormalised” (remember the shift-127 encoding)
- Not enough bits to represent such small quantities.
- All exponent bits being 1 indicate some special “numbers”:

±∞: all mantissa bits 0.
NaN: at least one mantissa bit non-zero.
Floating point numbers

- Floating point numbers with all bits in the exponent field at 0, are said to be “denormalised” (remember the shift-127 encoding).
- Not enough bits to represent such small quantities.
- All exponent bits being 1 indicate some special “numbers”:
  - $\pm\infty$: all mantissa bits 0.
Floating point numbers

- Floating point numbers with all bits in the exponent field at 0, are said to be “denormalised” (remember the shift-127 encoding)
- Not enough bits to represent such small quantities.
- All exponent bits being 1 indicate some special “numbers”:
  - $\pm\infty$: all mantissa bits 0.
  - NaN: at least one mantissa bit non-zero.
Example 2:

In examples/floating_fun.cc, there is a small program “simulating” a calculation involving some large quantities adding up to 0. Eight numbers are stored in an array of floats, and their sum evaluated and printed. The calculation is repeated by permuting the indexes of the array, so that the numbers are added in all possible orders. Observe the output!

Exercise 1: std::numeric_limits

What is epsilon for float and double on your computer? Find out by writing a small C++ program and printing out the values from std::numeric_limits. Look up the documentation of numeric_limits. What other information can you get about numeric types from that header?
**Float:**  \[1 - \text{bit}][8 - \text{bits}][23 - \text{bits}]  

- Maximum: \(3.40282e+38\)
- Minimum: \(1.17549e-38\)
- Lowest: \(-3.40282e+38\)
- Epsilon: \(1.19209e-07\)
- Rounding error: 0.5

**Double:**  \[1 - \text{bit}][11 - \text{bits}][52 - \text{bits}]  

- Maximum: \(1.79769e+308\)
- Minimum: \(2.22507e-308\)
- Lowest: \(-1.79769e+308\)
- Epsilon: \(2.22045e-16\)
- Rounding error: 0.5
Memory

- Cost of fetching one integer from the main memory can be a hundred times larger than getting it from the L1 cache.
- Fun fact: in 1 clock cycle of the CPU on my laptop, a photon travels about 10 cm!
Memory

- Cost of fetching one integer from the main memory can be a hundred times larger than getting it from the L1 cache
- Fun fact: in 1 clock cycle of the CPU on my laptop, a photon travels about 10 cm!

- Memory is fetched in “cache lines”
- Successive operations on contiguous memory locations do not incur the full cost of main memory access
std::vector<int> A(N*N, 0.0);
for (size_t i=0; i<N; ++i) {
    for (size_t j=0; j<N; ++j) {
        A[i*N+j] += j+i;
    }
}
for (size_t i=0; i<N; ++i) {
    for (size_t j=0; j<N; ++j) {
        A[j*N+i] += j+i;
    }
}
for (size_t i=0; i<N*N; ++i) {
    A[pos[i]] += i;
}

Q: Which way of accessing the “matrix” is faster, and by how much?

See also: CppCon 2016: Timur Doumler “Want fast C++? Know your hardware!”
std::vector<int> A(N*N, 0.0);
for (size_t i=0; i<N; ++i) {
    for (size_t j=0; j<N; ++j) {
        A[i*N+j] += j+i;
    }
}

for (size_t i=0; i<N; ++i) {
    for (size_t j=0; j<N; ++j) {
        A[j*N+i] += j+i;
    }
}

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

- Q: Which way of accessing the “matrix” is faster, and by how much?
- A: For N=10000, my laptop takes about 0.046 seconds for the row major pattern (top), and about 1.14 seconds for the column major pattern (middle), and 1.9 seconds for random pattern (bottom)

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

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

<table>
<thead>
<tr>
<th>Step</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.128</td>
</tr>
<tr>
<td>2</td>
<td>0.121</td>
</tr>
<tr>
<td>4</td>
<td>0.120</td>
</tr>
<tr>
<td>8</td>
<td>0.121</td>
</tr>
<tr>
<td>16</td>
<td>0.112</td>
</tr>
<tr>
<td>32</td>
<td>0.075</td>
</tr>
<tr>
<td>64</td>
<td>0.029</td>
</tr>
<tr>
<td>128</td>
<td>0.015</td>
</tr>
<tr>
<td>256</td>
<td>0.008</td>
</tr>
<tr>
<td>512</td>
<td>0.004</td>
</tr>
<tr>
<td>1024</td>
<td>0.003</td>
</tr>
<tr>
<td>2048</td>
<td>0.001</td>
</tr>
</tbody>
</table>

- 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

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.
Example 3: examples/memory_effects

In the folder examples/memory_effects you will find a few small programs illustrating 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
- `traverse1.cc` compares contiguous and non-contiguous access for different array sizes
- `war.cc` illustrates unexpected cache misses in regularly arranged data
Recommendations

- Prefer `std::array` and `std::vector` for all your container needs as a default. Any thing else needs to be carefully justified.
Recommendations

- Prefer `std::array` and `std::vector` for all your container needs as a default. Any thing else needs to be carefully justified.
- Organise code to maximise the use of any cache line that has been fetched:
Recommendations

- Prefer `std::array` and `std::vector` for all your container needs as a default. Any thing else 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
Recommendations

- Prefer `std::array` and `std::vector` for all your container needs as a default. Any thing else 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
Recommendations

- Prefer `std::array` and `std::vector` for all your container needs as a default. Any thing else 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
Functions

```
return_type function_name(parameters)
{
    // function body
}
double sin(double x)
{
    // Somehow calculate sin of x
    return answer;
}
int main()
{
    constexpr double pi=3.141592653589793;
    for (int i=0;i<100;++i) {
        std::cout << i*pi/100
        << sin(i*pi/100) <<"\n";
    }
    std::cout << sin("pi") <<"\n"; //Error!
}
```

- Logically connected reusable blocks of code
Functions

- Logically connected reusable blocks of code
- A function must be called with values called “arguments”.

```cpp
return_type function_name(parameters)  
{  
    // function body
}  
double sin(double x)  
{  
    // Somehow calculate sin of x
    return answer;
}
int main()  
{  
    constexpr double pi=3.141592653589793;  
    for (int i=0;i<100;++i) {  
        std::cout << i*pi/100  
                    << sin(i*pi/100) <<"\n";  
    }  
    std::cout << sin("pi") <<"\n";  //Error!
}
```
Functions

- Logically connected reusable blocks of code
- A function must be called with values called “arguments”.
- The type of the arguments must match or be implicitly convertible to the corresponding type in the function parameter list

```c++
return_type function_name(parameters)
{
    // function body
}
double sin(double x)
{
    // Somehow calculate sin of x
    return answer;
}
int main()
{
    constexpr double pi=3.141592653589793;
    for (int i=0;i<100;++i) {
        std::cout << i*pi/100
                   << sin(i*pi/100) <<"\n";
    }
    std::cout << sin("pi") <<"\n"; //Error!
}
```
Functions at run time

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

main()
   i:4
RP:OS

double sin(double x)
{
   // Somehow calculate sin of x
   return answer;
}
int main()
{
   constexpr double pi=3.141592653589793;
   for (int i=0;i<100;++i) {
      std::cout << i*pi/100 << sin(i*pi/100) <<"\n";
   }
   std::cout << sin("pi") <<"\n"; //Error!
}
Recursion

- Stack frame is bound to an individual call, not to the function body
- Each level of "recursion" has its own stack frame

```c
1 unsigned int factorial(unsigned int n)
2 {
3   int u=n; // u: Unnecessary
4   if (n>1) return n*factorial(n-1);
5   else return 1;
6 }
7 int someother()
8 {
9   factorial(4);
10 }
```
Recursion

- Stack frame is bound to an individual call, not to the function body
- Each level of "recursion" has its own stack frame
- Function parameters are copied to the stack frame

```c
#include <iostream>

unsigned int factorial(unsigned int n)
{
    int u=n; // u: Unnecessary
    if (n>1) return n*factorial(n-1);
    else return 1;
}

int someother()
{
    factorial(4);
}
```

---

20–21 June 2017
Sandipan Mohanty (s.mohanty@fz-juelich.de)
Recursion

- Stack frame is bound to an individual call, not to the function body
- Each level of "recursion" has its own stack frame
- Function parameters are copied to the stack frame
- Local variables at different levels of recursion live in their own stack frames, and do not interfere

```c
1 unsigned int factorial(unsigned int n) {
2     int u=n; // u: Unnecessary
3     if (n>1) return n*factorial(n-1);
4     else return 1;
5 }
6 int someother() {
7     factorial(4);
8 }
```
Recursion

- Stack frame is bound to an individual call, not to the function body
- Each level of "recursion" has its own stack frame
- Function parameters are copied to the stack frame
- Local variables at different levels of recursion live in their own stack frames, and do not interfere

```c
unsigned int factorial(unsigned int n)
{
    int u=n; // u: Unnecessary
    if (n>1) return n*factorial(n-1);
    else return 1;
}

int someother()
{
    factorial(4);
}
```
Recursion

- Stack frame is bound to an individual call, not to the function body
- Each level of "recursion" has its own stack frame
- Function parameters are copied to the stack frame
- Local variables at different levels of recursion live in their own stack frames, and do not interfere
Recursion

- Stack frame is bound to an individual call, not to the function body
- Each level of "recursion" has its own stack frame
- Function parameters are copied to the stack frame
- Local variables at different levels of recursion live in their own stack frames, and do not interfere

```c
1 unsigned int factorial(unsigned int n)
2 {
3   int u=n; // u: Unnecessary
4   if (n>1) return n*factorial(n-1);
5   else return 1;
6 }
7 int someother()
8 {
9   factorial(4);
10 }
```
Recursion

- Stack frame is bound to an individual call, not to the function body
- Each level of "recursion" has its own stack frame
- Function parameters are copied to the stack frame
- Local variables at different levels of recursion live in their own stack frames, and do not interfere

```c
1 unsigned int factorial(unsigned int n)
2 {
3     int u=n; // u: Unnecessary
4     if (n>1) return n*factorial(n-1);
5     else return 1;
6 }
7 int someother()
8 {
9     factorial(4);
10 }
```
Recursion

- Stack frame is bound to an individual call, not to the function body
- Each level of "recursion" has its own stack frame
- Function parameters are copied to the stack frame
- Local variables at different levels of recursion live in their own stack frames, and do not interfere
Functions: under the microscope!

```cpp
double f(const double x, const double y) {
    return x+y*y;
}
```

- Function parameters are passed in registers or copied to the stack

# G="g++ -std=c++14 -march=Native -O3"
G -c -g f.cc
objdump -d -M intel -S f.o

```
0000000000000000 <__Zlfdd>:   
  vmulsd xmm1,xmm1,xmm1        
  vaddsd xmm0,xmm1,xmm0        
  ret                           
```
Functions: under the microscope!

```cpp
double f(const double x, const double y) {
    return x+y*y;
}
```

```
# G="g++ -std=c++14 -march=Native -O3"
G -c -g f.cc
objdump -d -M intel -S f.o
```

- Function parameters are passed in registers or copied to the stack
- On X86_64 under Linux,
Functions: under the microscope!

```c
double f(const double x, const double y) {
    return x+y*y;
}
```

- Function parameters are passed in registers or copied to the stack
- On X86_64 under Linux,
  - XMM0..XMM7 are used for floating point arguments

```c
# G="g++ -std=c++14 -march=Native -O3"
G -c -g f.cc
objdump -d -M intel -S f.o

0000000000000000 <__Zlfdd>:
  vmulsd xmm1,xmm1,xmm1
  vaddsd xmm0,xmm1,xmm0
  ret
```
Functions: under the microscope!

```c
double f(const double x, const double y)
{
    return x+y*y;
}
```

```bash
# G="g++ -std=c++14 -march=Native -O3"
G -c -g f.cc
objdump -d -M intel -S f.o
```

- Function parameters are passed in registers or copied to the stack
- On X86_64 under Linux,
  - XMM0..XMM7 are used for floating point arguments
  - 6 integer or pointer arguments are passed in registers RDI, RSI, RDX, RCX, R8 and R9
Functions: under the microscope!

```cpp
class D {
    int nm;
    double d;
public:
    inline void val(double x) { d=x; }
    inline double val() const { return d; }
    inline auto name() const { return nm; }
    double operator+(double x1) const;
};

double D::operator+(double x) const
{
    return d+x*x;
}
```

- Object of different classes are passed as pointers (references are hidden pointers)
- Any additional arguments are passed on the stack
- Function body is executed
- Return value is written
- Execution continues at the previously stored return address
Stack

```cpp
class V3 {
    double x, y, z;
    V3 cross(const V3 &);
    double dot(const V3 &);
};

double prob(int i, const V3 & x, const V3 & y)
{
    int j = i % 233;
    V3 tmp{x};
    for (; j < i; ++j) {
        tmp = tmp.cross(y);
    }
    return tmp.dot(x);
}
```

- Heavily reused memory locations
Stack

- Heavily reused memory locations
- Likely cached, therefore, fast

```cpp
class V3 {
    double x, y, z;
    V3 cross(const V3 &);
    double dot(const V3 &);
};
double prob(int i, const V3 & x, const V3 & y)
{
    int j = i % 233;
    V3 tmp{x};
    for (; j < i; ++j) {
        tmp = tmp.cross(y);
    }
    return tmp.dot(x);
}
```
Stack

```cpp
class V3 {
    double x, y, z;
    V3 cross(const V3 &);  
    double dot(const V3 &);
};
double prob(int i, const V3 & x, const V3 & y) {
    int j = i % 233;
    V3 tmp{x};
    for (; j < i; ++j) {
        tmp = tmp.cross(y);
    }
    return tmp.dot(x);
}
```

- Heavily reused memory locations
- Likely cached, therefore, fast
- All local variables of any type
Global storage

```
double prob(int i)
{
    static int c{0};
    ++c;
    if (c%1000==0) {
        std::cout<<"Call count reached "
                  << c << "\n";
    }
    static const double L[]={3.14,2.71};
    return L[i%2];
}
```

- Variables outside any function
Global storage

```cpp
double prob(int i)
{
    static int c{0};
    ++c;
    if (c%1000==0) {
        std::cout<<"Call count reached ", << c << 
    }
    static const double L[]={3.14,2.71};
    return L[i%2];
}
```

- Variables outside any function
- Variables marked with the `static` keyword in functions
Global storage

```cpp
double prob(int i)
{
    static int c{0};
    ++c;
    if (c%1000==0) {
        std::cout<<"Call count reached "
            << c << 
        ;
    }
    static const double L[]={3.14,2.71};
    return L[i%2];
}
```

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

```cpp
void f()
{
    int *A=new int[1000000];
    // calculations with A
    delete [] A;
}
```

- Explicitly/implicitly managed memory through `new`, `delete`, `malloc` or `free`

Arrays whose size is not known at compile time. C99 style variable length arrays are not standard C++. 

20–21 June 2017 Sandipan Mohanty (s.mohanty@fz-juelich.de)
Heap

```cpp
void f()
{
    int *A = new int[1000000];
    // calculations with A
    delete [] A;
}
```

- Explicitly/implicitly managed memory through `new`, `delete`, `malloc` or `free`
- Can store very large objects which don’t fit in the stack
Heap

```cpp
void f()
{
    int *A=new int[1000000];
    // calculations with A
    delete [] A;
}
```

- Explicitly/implicitly managed memory through `new`, `delete`, `malloc` or `free`
- 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 length arrays are not standard C++.
Heap

```c
void f()
{
    int *A=new int[1000000];
    // calculations with A
    delete [] A;
}
```

- Tends to get fragmented

Objects stored one after the other may end up in very different locations

Slower than stack storage
Heap

```c
void f()
{
    int *A=new int[1000000];
    // calculations with A
    delete [] A;
}
```

- Tends to get fragmented
- Must find a suitably sized unused block

Objects stored one after the other may end up in very different locations

Slower than stack storage
Heap

```c
void f()
{
    int *A=new int[1000000];
    // calculations with A
    delete [] A;
}
```

- Tends to get fragmented
- Must find a suitably sized unused block
- Must keep track of what is and isn’t in use
Heap

```cpp
void f()
{
    int *A=new int[1000000];
    // calculations with A
    delete [] A;
}
```

- Tends to get fragmented
- Must find a suitably sized unused block
- Must keep track of what is and isn’t in use
- Must remember to free memory before accessing pointers go out of scope
Heap

```c
void f()
{
    int *A=new int[1000000];
    // calculations with A
    delete [] A;
}
```

- Tends to get fragmented
- Must find a suitably sized unused block
- Must keep track of what is and isn’t in use
- 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
Heap

```c
void f()
{
    int *A=new int[1000000];
    // calculations with A
    delete [] A;
}
```

- Tends to get fragmented
- Must find a suitably sized unused block
- Must keep track of what is and isn’t in use
- 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
Resource handles and heap allocated data
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

(a) char *data
size_t n
void f() {
    string A("ABC");
    cout << A;
}
Stack

(b) char *data
size_t n
void f() {
    string A("ABC");
    cout << A;
} Stack
Heap

(c) char *data
size_t n
~string() this
void f() {
    string A("ABC");
    cout << A;
} Stack
Heap

(d) char *data
size_t n
void f() {
    string A("ABC");
    cout << A;
} Stack
Heap
Resource handles

- STL containers (except `std::array`) are "resource" handles
Resource handles

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

```cpp
vector<int> A(32, 0);
vector<double> B(64, 0.);
vector<complex<double>> C(128);
vector<bool> D(256);
cout << sizeof(A) << "\n"
    << sizeof(B) << "\n"
    << sizeof(C) << "\n"
    << sizeof(C) << "\n";
```

Quiz

What will the program print?
Resource handles

- Can transfer ownership of the resources very cheaply
Resource handles

Move

- Can transfer ownership of the resources very cheaply
- Actual data on the heap need not be touched at all!
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

```cpp
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.
Resource handles

- In C++, objects (instances of a class) can live on the stack or on the heap.
- Putting resource handles like `vector<vector<int>>` on the heap, while allowed, incurs the cost of additional indirections.

```cpp
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
    }
}
```
Resource handles

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.
- When possible, avoid cumbersome beasts like `vector<vector<int>>`.

```cpp
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
    }
}
```
If you need your own 2D arrays, ...

- 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 to mimic 2D arrays in C

Example 4:
examples/array2d contains the template class shown here.
std::array

```cpp
#include <iostream>
#include <array>

int main()
{
  std::array<double,10> A{{0.0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.7,0.9}};
  std::cout << "Size of array on stack = " << sizeof(A) << "\n";
  std::cout << "size() = " << A.size() << "\n";
}
```

- Resembles other STL containers, but this is not just a handle.
std::array

```cpp
#include <iostream>
#include <array>

int main()
{
    std::array<double, 10> A{{0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.7, 0.9}};
    std::cout << "Size of array on stack = " << sizeof(A) << "\n";
    std::cout << "size() = " << A.size() << "\n";
}
```

- 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!
std::array

```cpp
#include <iostream>
#include <array>

int main()
{
    std::array<double,10> A{{0.0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.7,0.9}};
    std::cout << "Size of array on stack = " << sizeof(A) << "\n";
    std::cout << "size() = " << A.size() << "\n";
}
```

- 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

- 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.
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
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.
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.
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/)
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.

Example 5:

Verify the above using the example program `examples/align/alignof.cc`. 
Data structure padding

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

```cpp
class A {
    char c;
    double x;
    int d;
};
// Compiled as if it was ...
char c;
char pad[7];
double x;
int d;
char pad2[4]; // why is this here ?
// Overall alignment alignof(double)
// size of struct = 24

class B {
    double x;
    int d;
    char c;
};
// Compiled as if it was ...
double x;
int d;
char c;
char pad[3];
// Overall alignment alignof(double)
// size of struct = 16
```
Data structure padding

- Alignment requirement of members can necessitate introduction of padding between members
- Size of structures can therefore be bigger than the sum of sizes of their elements

```c
class A {
    char c;
    double x;
    int d;
};
// Compiled as if it was ...
char c;
char pad[7];
double x;
int d;
char pad2[4]; // why is this here?
// Overall alignment alignof(double)
// size of struct = 24
class B {
    double x;
    int d;
    char c;
};
// Compiled as if it was ...
double x;
int d;
char c;
char pad[3];
// Overall alignment alignof(double)
// size of struct = 16
```
Data structure padding

- Alignment requirement of members can necessitate introduction of padding between members
- Size of structures can therefore be bigger than the sum of sizes of their elements
- C++ rules do not allow the compiler to reorder elements for space
Alignment requirement of members can necessitate introduction of padding between members

Size of structures can therefore 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
alignas

alignas(64) double x[4]; // ok
alignas(64) vector<double>(4) a;
// Pointless.
// The above simply aligns the resource
// handle, not the data on the heap
alignas(64) array<double,4> A;
// This is fine, as std::array has
// real data in its struct

template <typename T, int vecsize>
struct alignas(vecsize) simd_t
{
    array<T,vecsize/sizeof(T)> data;
};

- 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
Example 6:

The examples/align/align0.cc has a concept of a template class which creates a data array of the right size to fill the vector length irrespective of the input data type. It illustrates the use of alignof and alignas.
std::align

```cpp
void* align(size_t alignment, size_t size, void*& ptr, size_t& space);
```

- Given a buffer, find a suitably aligned starting address inside it
std::align

```c
void* align(size_t alignment, size_t size, void*& ptr, size_t& space );
```

- Given a buffer, find a suitably aligned starting address inside it
- The starting value of `ptr` is used as the starting memory location
std::align

```cpp
void* align(size_t alignment, size_t size, void*& ptr, size_t& space );
```

- Given a buffer, find a suitably aligned starting address inside it.
- The starting value of `ptr` is used as the starting memory location.
- The starting value of `space` is used as the remaining number of bytes in buffer.
std::align

```c
void* align(size_t alignment, size_t size, void*& ptr, size_t& space);
```

- Given a buffer, find a suitably aligned starting address inside it
- The starting value of `ptr` is used as the starting memory location
- The starting value of `space` is used as the remaining number of bytes in buffer
- `alignment` is the sought alignment of data. `ptr` is increased until it lands on an address with this alignment.
std::align

```cpp
void* align(size_t alignment, size_t size, void*& ptr, size_t& space );
```

- Given a buffer, find a suitably aligned starting address inside it
- The starting value of `ptr` is used as the starting memory location
- The starting value of `space` is used as the remaining number of bytes in buffer
- `alignment` is the sought alignment of data. `ptr` is increased until it lands on an address with this alignment.
- `size` is the size of the object(s) intended for the aligned memory. If, after moving `ptr` we have less than `size` bytes left, the operation fails, and `nullptr` is returned
std::aligned_storage

```cpp
template <size_t Length, size_t Alignment>
struct aligned_storage {
    using type = struct { alignas (Alignment) unsigned char data[Length]; }
    // Obs: This is the idea, not necessarily the real implementation!
};
template <size_t Length, size_t Alignment>
using aligned_storage_t = typename aligned_storage<Length, Alignment>::type;
```

- Raw uninitialized storage type for use by any type with size at most `Length` and alignment a divisor of `Alignment`
std::aligned_storage

```
template <size_t Length, size_t Alignment>
struct aligned_storage {
    using type = struct {alignas (Alignment) unsigned char data[Length]; };
    // Obs: This is the idea, not necessarily the real implementation!
};
template <size_t Length, size_t Alignment>
using aligned_storage_t = typename aligned_storage<Length, Alignment>::type;
```

- Raw uninitialized storage type for use by any type with size at most `Length` and alignment a divisor of `Alignment`
- Convenient alias `aligned_storage_t` available in C++14
std::aligned_storage

```cpp
template <size_t Length, size_t Alignment>
struct aligned_storage {
    using type = struct {alignas (Alignment) unsigned char data[Length]; }
    // Obs: This is the idea, not necessarily the real implementation!
};
template <size_t Length, size_t Alignment>
using aligned_storage_t = typename aligned_storage<Length, Alignment>::type;
```

- Raw uninitialized storage type for use by any type with size at most `Length` and alignment a divisor of `Alignment`
- Convenient alias `aligned_storage_t` available in C++14
- Used with "placement new" operator and explicit destructor calls to create/destroy objects of different types
The placement new operator

- **new**: Obtain memory buffer and run a constructor on it.

```cpp
// Allocation and deallocation with new
double * d = new double[4];
delete [] d;

Protein * p = new Protein("nmrstruc.xml");
//
delete p;
```

```cpp
// Usage of "placement new"
// There exists an uninitialized buffer
// on the stack or the heap.

Protein * p
  = new(buffer) Protein("nmrstruc.xml");
// Use existing location, but initialize
// with given constructor.
p->~Protein();
// Call destructor, but don't free.
```
The placement new operator

- `new`: Obtain memory buffer and run a constructor on it.
- “Placement new” operator: We have a buffer, just run the constructor.

```cpp
// Allocation and deallocation with new
double * d = new double[4];
delete [] d;

Protein * p = new Protein("nmrstruc.xml");
//
delete p;
```

```cpp
// Usage of "placement new"
// There exists an uninitialized buffer
// on the stack or the heap.

Protein * p
    = new (buffer) Protein("nmrstruc.xml");
// Use existing location, but initialize
// with given constructor.
p->~Protein();
// Call destructor, but don’t free.
```
The placement new operator

- **new**: Obtain memory buffer and run a constructor on it.

- “Placement new” operator: We have a buffer, just run the constructor.

- Since placement `new` does not obtain memory itself, it should not be paired with a `delete` but rather an explicit destructor call at that pointer location.

```c++
// Allocation and deallocation with new
double * d = new double[4];
delete [] d;

Protein * p = new Protein("nmrstruc.xml");
//
delete p;
```

```c++
// Usage of "placement new"
// There exists an uninitialized buffer
// on the stack or the heap.

Protein * p = new (buffer) Protein("nmrstruc.xml");
// Use existing location, but initialize
// with given constructor.
p->~Protein();
// Call destructor, but don’t free.
```
Example 7:

examples/align/stdalign.cc illustrates the use of std::align through a use defined allocator class, which can allocate memory from an internal buffer, through an allocate function which takes an alignment requirement as an argument. The original example is from en.cppreference.com, which I have modified slightly and commented with additional explanations.
Example 8:

`examples/align/aligned_storage.cc` illustrates the use of 
`std::aligned_storage`, `std::forward`, the placement `new` 
operator and explicit destructor calls. The original example is 
from `en.cppreference.com`, which I have modified slightly 
and commented with additional explanations. Please study the 
code, run it, understand the output and ask if you have any 
difficulties.
Cost of various abstractions
Reading assembly code

Exercise 2:
The website 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. A few ideas on what to try are in the folder examples/assembly.

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

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

- Program execution flows through different units responsible for different work

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

- request mem \( x \)
- request mem \( y \)
- calc \( x+y \)
- calc \( \text{res} > 5 \)
- ?

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

```c
for (int i=0; i<N; ++i) {
    if (p[i] > gen()) {
        b[i] = a[i] + c[i];
        ++fwd;
    } else {
        a[i] = b[i] + c[i];
        ++rev;
    }
}
nngb=0;
while (a) {
    dist[nngb++] = distf(a, i);
}
```

- 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

```
for (int i=0; i<N; ++i) {
    if (p[i] > gen()) {
        a[i] = (b[i]>r0 && b[i] < r1 && c[i]<b[i]);
    } else {
        a[i] = b[i]+c[i];
        ++rev;
    }
}
nngb=0;
while (a) {
    dist[nngb++]=distf(a,i);
}
```

- If statements, switches, loops contain obvious branches
- The ternary operator
  
  \[ a = \text{cond} ? v1 : v2 \]
  
  (not always!) a branch

- Not so obvious branches include boolean `||` and `&&` operators:
Branch mis-prediction penalty

- If statements, switches, loops contain obvious branches
- The ternary operator
  \[ a = \text{cond ? } v1 : v2 \] is (not always!) a branch

```c
for (int i=0;i<N;++i) {
  if (p[i] > gen()) {
    a[i] = (b[i]>r0 && b[i] < r1 && c[i]<b[i]);
  } else {
    a[i] = b[i]+c[i];
    ++rev;
  }
}
```

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

- If statements, switches, loops contain obvious branches
- The ternary operator
  \[ a = \text{cond} ? v1 : v2 \] is (not always!) a branch

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

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

- Conditional assignments are often reorganised as simple sequential instructions by compilers using assembly language tricks.

```
0000000000000000 <__Z1fi>:
    cmp    edi,0x4
    setl   al
    movz x eax,al
    inc    eax
    ret

0000000000000000 <__Z1fdPd>:
    subd   xmm0,QWORD PTR [rdi]
    subd   xmm0,QWORD PTR [rdi+0x8]
    subd   xmm0,QWORD PTR [rdi+0x10]
    subd   xmm0,QWORD PTR [rdi+0x18]
    ret
```
**Not branches**

```c
double f(double x, double A[4])
{
    double a=x;
    for (int i=0;i<4;++i) a-=A[i];
    return a;
}
```

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

```assembly
0000000000000000 <__Z1fdPd>:
    subsd xmm0,QWORD PTR [rdi]
    subsd xmm0,QWORD PTR [rdi+0x8]
    subsd xmm0,QWORD PTR [rdi+0x10]
    subsd xmm0,QWORD PTR [rdi+0x18]
    ret
```
Example 9:

Branch prediction effectiveness varies a lot between CPUs. Using the example program `examples/branch_prediction.cc`, compare the Sandy bridge series of processors on your workstations with the Haswell processors on JURECA. 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?
Class hierarchies
Inheritance

- Inheriting class may add more data, but it retains all the data of the base class.

- Access of base class functions

- Access of derived class functions (qualified by private, protected etc.)
Inheritance

- 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.

**Table:**

<table>
<thead>
<tr>
<th>Base class data</th>
<th>Derived class extra data</th>
</tr>
</thead>
</table>

**Accesses:**

- Access of base class functions
- Access of derived class functions (qualified by private, protected etc.)
Inheritance

- 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.

<table>
<thead>
<tr>
<th>Base class data</th>
<th>Derived class extra data</th>
</tr>
</thead>
</table>

access of base class functions

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

- A pointer to a derived class always points to an address which also contains a valid base class object.

- Implicit downcasting is not allowed. Explicit downcasting is possible with `static_cast` and `dynamic_cast`.
Inheritance

- 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.

### Diagram

- **Base class data**
  - access of base class functions
  - access of derived class functions (qualified by private, protected etc)

- **Derived class extra data**
Inheritance

- 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)

class Base {
public:
  void f() { std::cout << "Base::f()\n"; }
protected:
  int i{4};
};
class Derived : public Base {
  int k{0};
public:
  void g() { std::cout << "Derived::g()\n"; }
};
int main()
{
  Derived b;
  Base *ptr = &b;
  ptr->g(); // Error!
  static_cast<Derived *>(ptr)->g(); // OK
}
Class inheritance with virtual functions

```cpp
int main()
{
    darray<Shape *> shape;
    shape.push_back(new Circle(0.5, Point(3,7)));
    shape.push_back(new Triangle(Point(1,2),Point(3,3),Point(2.5,0)));
    ...
    for (size_t i=0;i<shape.size();++i) {
        std::cout<<shape[i]->area()<<'
';
    }
}
```

- A pointer to a base class is allowed to point to an object of a derived class
Class inheritance with virtual functions

```cpp
int main()
{
    darray<Shape *> shape;
    shape.push_back(new Circle(0.5, Point(3,7)));
    shape.push_back(new Triangle(Point(1,2),Point(3,3),Point(2.5,0)));
...
    for (size_t i=0;i<shape.size();++i) {
        std::cout<<shape[i]->area()<<'
';
    }
}
```

- A 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()`
A 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

- For classes with virtual functions, the compiler inserts an invisible pointer member to the data and additional book keeping code.
Calling virtual functions: how it works

- 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.
Calling virtual functions: how it works

- For classes with virtual functions, the compiler inserts an invisible pointer member to the data and additional bookkeeping code.
- There is a table of virtual functions for each derived class, with entries pointing to function code somewhere.
- The `vpotr` pointer points to the `vtable` of that particular class.
Example 10:

The program examples/vptr.cc gives you a tiny class with two double data members. The main function simply creates an object of this kind and prints its size. It prints 16 (bytes) as expected. Uncomment a line containing a virtual destructor function for this class, and recompile and re-run. It will now print 24 as the size on a typical 64 bit machine. Compiling with g++ -O0 -g3 --no-inline and running it in the debugger, you can see the layout of the data structure in memory, and verify that the extra data member is indeed a pointer to a vtable for the 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.
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.

- For HPC applications, use of virtual functions in hot sections will hurt performance.
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.

- 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
Tag dispatching

```cpp
struct property1 {};  
struct property2 {};  
template <typename T>
void do_something(T && t, property1)
{
    std::cout << "Function objects with property1\n";
}
template <typename T>
void do_something(T && t, property2) {...}
//...
template <typename T>
void do_something(T t)
{
    do_something(t, typename T::tag{});
}
class Bird {
    public:
        using tag = typename property1;
    };
class SparseMatrix {
    public:
        using tag = typename property2;
    };
//...
Bird b;
do_something(b);
SparseMatrix m;
do_something(m);
```

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

```
struct property1 {};
struct property2 {};
template <typename T>
void do_something(T && t, property1)
{
    std::cout << "Function objects with property1\n";
}
template <typename T>
void do_something(T && t, property2) {...}
//...
template <typename T>
void do_something(T t)
{
    do_something(t, typename T::tag{});
}
```

```
class Bird {
public:
    using tag = typename property1;
};
class SparseMatrix {
public:
    using tag = typename property2;
};
//...
Bird b;
do_something(b);
SparseMatrix m;
do_something(m);
```

- 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
SFINAE : Substitution Failure is not an Error

Overload resolution of templates

```cpp
// Examples/sfinae0.cc
template <typename V>
void f(const V &v,
        typename V::iterator * jt=0)
{
    std::cout << "Container overload\n";
    for (auto x : v) std::cout << x <<" ";
    std::cout << "\n";
}

void f(...) {
    std::cout << "Catch all overload\n";
}

int main() {
    std::list<double> L{
            0.1, 0.2, 0.3, 0.4, 0.5, 0.6};
    int A[4]{4, 3, 2, 1};
    f(A);
    f(L);
}
```
SFINAE : Substitution Failure is not an Error

Overload resolution of templates
- If substitution fails, overload discarded

```cpp
// Examples/sfinae0.cc
template <typename V>
void f(const V &v,
    typename V::iterator * jt=0)
{
    std::cout << "Container overload\n"
    for (auto x : v) std::cout << x <<" ";
    std::cout << "\n";
}

void f(...) {
    std::cout << "Catch all overload\n";
}

int main() {
    std::list<double> L
    {0.1,0.2,0.3,0.4,0.5,0.6};
    int A[4]{4,3,2,1};
    f(A);
    f(L);
}
```
SFINAE : Substitution Failure is not an Error

- Overload resolution of templates
- If substitution fails, overload discarded
- All parameters, expressions and the return type in declarations
SFINAE : Substitution Failure is not an Error

- 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

```cpp
// Examples/sfinae0.cc
template <typename V>
void f(const V &v, typename V::iterator *jt=0)
{
    std::cout << "Container overload\n";
    for (auto x : v) std::cout << x << " ";
    std::cout << "\n";
}

void f(...) {
    std::cout << "Catch all overload\n";
}

int main()
{
    std::list<double> L
        {0.1, 0.2, 0.3, 0.4, 0.5, 0.6};
    int A[4]{4, 3, 2, 1};
    f(A);
    f(L);
}
```
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

- 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.

```cpp
template <bool B, class T>
struct enable_if;

template <class T>
struct enable_if<true, T> {
  using type = T;
};

template <bool B, class T = void>
using enable_if_t = typename enable_if<B, T>::type;

template <typename T>
enable_if_t<is_integral<T>::value, T> Power(T x, T y) {
  // Implementation suitable for integral number parameters
}

template <typename T>
enable_if_t<is_floating_point<T>::value, T> Power(T x, T y) {
  // Implementation suitable for floating point parameters
}
```
Example 11:
The tag dispatching technique is demonstrated in examples/tag_dispatch.cc.

Example 12:
examples/sfinae0.cc is a simple syntax illustration for SFINAE.

Example 13:
examples/enableif0.cc shows one use of std::enable_if. The parameter list of the two variants of the template function \( f \) are identical, and they are "templates", where the place holder typename \( T \) can take arbitrary values. Yet, we can create two versions of the function and have the compiler choose one or the other depending on the properties of the input type.
Choosing algorithm based on API

- 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

```cpp
// C++17
template <class C> size_t algo(C && x)
{
    if constexpr (hasAPI<C>) {
        x.helper();
        return x.calculateFast();
    } else {
        return x.calculate();
    }
}
```
Choosing algorithm based on API

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.

```
template <typename T> struct hasAPI_t {
    using basetype =
        typename remove_reference<T>::type;
    template <class C>
    static constexpr auto test(C * x) ->
        decltype(x->calculateFast(),
            x->helper(),
            bool{})
    {
        return true;
    }
    static constexpr bool test(...) {
        return false;
    }
    static constexpr auto value =
        test(static_cast<basetype*>(nullptr));
};
```
Choosing algorithm based on API

```cpp
template <typename T> struct hasAPI_t {
    using basetype = typename remove_reference<T>::type;
    template <class C>
    static constexpr auto test(C * x) -> decltype(x->calculateFast(), x->helper(), bool{}) {
        return true;
    }
    static constexpr bool test(...) {
        return false;
    }
    static constexpr auto value = test(static_cast<basetype*>(nullptr));
};
```

- 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

template <typename T> struct hasAPI_t {
    using basetype =
        typename remove_reference<T>::type;
    template <class C>
    static constexpr auto test(C * x) ->
        decltype(x->calculateFast(),
                  x->helper(),
                  bool{})
    {
        return true;
    }
    static constexpr bool test(...) {
        return false;
    }
    static constexpr auto value =
        test(static_cast<basetype*>(nullptr));
};

- 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.
Choosing algorithm based on API

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

- 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

```cpp
template <typename T> constexpr bool hasAPI = hasAPI_t<T>::value;
template <class C> std::enable_if_t< hasAPI<C>, size_t > algo(C && x)
{
    x.helper();
    return x.calculateFast();
}
template <class C> std::enable_if_t< !hasAPI<C>, size_t > algo(C && x)
{
    return x.calculate();
}
```

- 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.
Choosing algorithm based on API

```cpp
template <typename T> constexpr bool hasAPI = hasAPI_t<T>::value;

template <class C> std::enable_if_t< hasAPI<C>, size_t > algo(C && x)
{
    x.helper();
    return x.calculateFast();
}

template <class C> std::enable_if_t< !hasAPI<C>, size_t > algo(C && x)
{
    return x.calculate();
}
```

- 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.
Choosing algorithm based on API

```cpp
int main()
{
    Machinery obj;
    auto res = algo(obj);
    std::cout << "Result = " << res << "\n";
}
```

- Users of our great algorithm can simply call our `algo()` in their code
Choosing algorithm based on API

```cpp
int main()
{
    Machinery obj;
    auto res = algo(obj);
    std::cout << "Result = " << res << "\n";
}
```

- Users of our great algorithm can simply call our `algo()` in their code.
- If they have a `calculate` function, everything will work.
Choosing algorithm based on API

```cpp
int main()
{
    Machinery obj;
    auto res = algo(obj);
    std::cout << "Result = " << res << "\n";
}
```

- Users of our great algorithm can simply call our `algo()` in their code.
- If they have a `calculate` function, everything will work.
- If they then go on to implement `calculateFast` in their `Machinery` class, without any changes at the call site, or in the `algo` function, the compiler will make sure that we are using the (hopefully) better, `calculateFast` function.
Example 14:

The program examples/shim1.cc is an interesting application of SFINAE, where we determine whether a template argument passed to a template function is a type which has a member function called size(). Modify to detect another member function!

Example 15:

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.
Curiously Recurring Template Pattern
Curiously Recurring Template Pattern

- You need types A and B which have some properties in common, which can be calculated using similar data
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.
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.
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

```cpp
template <class D> struct Named {
  inline string get_name() const {
    // polymorphic function, simply
    // redirect to the class D given
    // as a template parameter. Wont compile
    // if D does not inherit from this class
    return static_cast<D const *>(this)
        ->get_name_impl();
  }

  inline int version() const {
    // Non-polymorphic "common" function
    return 42;
  }
};

struct Acetyl : public Named<Acetyl> {
  inline string get_name_impl() const {
    return "Acetyl";
  }
};

struct Car : public Named<Car> {
  inline string get_name_impl() const {
    return get_brand()+get_model()+
        get_year();
  }
};

int main() {
  Acetyl a;
  Car b;
  cout << "get_name on a returns : "
       << a.get_name() << '\n';
  cout << "get_name on b returns : "
       << b.get_name() << '\n';
  cout << "Their versions are "
       << a.version()<< " and "
       << b.version()<<'\n';
}
```

CRTP

- Polymorphism without virtual functions
- Faster in many cases
Expression Templates
Expression Templates

template <typename T>
class vec {
  std::vector<T> dat;
public:
  vec(size_t n) : dat(n) {}  
  T operator[](size_t i) const {
    return dat[i];
  }
  T & operator[](size_t i) {
    return dat[i];
  }
  size_t size() const{ return dat.size(); }
};
template <typename T>
vec<T> operator+(const vec<T> & v1, const vec<T> & v2) {
  assert(v1.size()==v2.size());
  auto ans=v1;
  for (size_t i=0; i<ans.size(); ++i)
    ans[i]+=v2[i];
  return ans;
}

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

- Naive implementation which elegantly expresses our intent
Expression Templates

```cpp
template <typename T>
class vec {
    std::vector<T> dat;
public:
    vec(size_t n) : dat(n) {}
    T operator[](size_t i) const {
        return dat[i];
    }
    T & operator[](size_t i) {
        return dat[i];
    }
    size_t size() const {
        return dat.size();
    }
};
template <typename T>
vec<T> operator+(const vec<T> & v1, const vec<T> & v2) {
    assert(v1.size() == v2.size());
    auto ans = v1;
    for (size_t i = 0; i < ans.size(); ++i)
        ans[i] += v2[i];
    return ans;
}
```

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

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

```cpp
template <typename T>
class vec {
    std::vector<T> dat;
public:
    vec(size_t n) : dat(n) {}  
    T operator[](size_t i) const {
        return dat[i];
    }
    T & operator[](size_t i) {
        return dat[i];
    }
    size_t size() const{return dat.size();}
};
template <typename T>
vec<T> operator+(const vec<T> & v1, const vec<T> & v2) {
    assert(v1.size()==v2.size());
    auto ans=v1;
    for (size_t i=0;i<ans.size();++i)
        ans[i]+=v2[i];
    return ans;
}
```

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

- Naive implementation which elegantly expresses our intent
- 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 $Z$
Expression templates
If only we had a special class ...

- ... which stored references to $X$, $Y$ and $Z$
- and had an `operator[]` which returns
  
  $$a \times X[i] + 2 \times a \times Y[i] + 3 \times a \times Z[i]$$

```cpp
template <typename T>
class vec {
    template <class XPR>
    vec & operator=(const XPR & r) {
        for (size_t i=0; i<size(); ++i) {
            dat[i]=r[i]; // and r[i] returns a*X[i]+2*a*Y[i]+3*a*Z[i]
        } // One single loop, no temporaries
        return *this;
    }
}
```
Expression templates
If only we had a special class ...

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

```cpp
template <typename T>
class vec {
    template <class XPR>
    vec & operator=(const XPR & r) {
        for (size_t i=0; i<size(); ++i) {
            dat[i]=r[i]; // and r[i] returns a*X[i]+2*a*Y[i]+3*a*Z[i]
        } // One single loop, no temporaries
        return *this;
    }
};
```
Expression templates
If only we had a special class ...

- ... which stored references to $X$, $Y$ and $Z$
- and had an `operator[]` which returns
  
  $$a \cdot X[i] + 2 \cdot a \cdot Y[i] + 3 \cdot a \cdot Z[i]$$

- We could equip our `vec` class with a special assignment operator taking this special class as the right hand side

```cpp
template <typename T>
class vec {
    template <class XPR>
    vec & operator=(const XPR & r) {
        for (size_t i=0;i<size();++i) {
            dat[i]=r[i]; // and r[i] returns $a \cdot X[i] + 2 \cdot a \cdot Y[i] + 3 \cdot a \cdot Z[i]$
        } // One single loop, no temporaries
        return *this;
    }
};
```

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

If we make a class like:

```cpp
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]; }

    size_t size() const { return lhs.size(); }
};
```

We can define the sum of two vecxpr objects to be a vecsum type

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

- If we try `vec1+vec2`, no evaluation happens, and we get a `vecsum<vec, vec>` object
- But, if we try `vec1+54` or `34+"dino"`, we get nonsensical compound objects
- If we write our `operator+` like:

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

, we can prevent the template from matching anything other than 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 `vecxpr<something>`
Expression templates
Design with CRTP

- By creating a base template `vecxpr` to use as a base for all expressions of `vec` objects

```cpp
template <class Derived> struct vecxpr {
    inline size_t size() const {
        return static_cast<Derived const &>(*this).size();
    }
    inline const auto operator[](size_t i) const {
        return static_cast<Derived const &>(*this)[i];
    }
    operator Derived & () {
        return static_cast<Derived&>(*this);
    }
    operator const Derived & () const {
        return static_cast<const Derived&>(*this);
    }
};
```

, we can prevent the template from matching anything other than objects which match the pattern `vecxpr<something>`
Expression templates
Design with CRTP

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

```cpp
template <typename T1, typename T2>
class vecsum : public vecxpr<vecsum<T1,T2>> {
    const T1 & lhs;
    const T2 & rhs;
public:
    using value_type=typename T1::value_type;
    vecsum(const vecxpr<T1> & l, const vecxpr<T2> & r) : lhs(l), rhs(r) {
        assert(l.size()==r.size());
    }
    const auto operator[](size_t i) const { return lhs[i] + rhs[i]; }  
    size_t size() const { return lhs.size();  }
};
```

`operator+` can now be written as:

```cpp
template <typename T1, typename T2>
vecsum<T1,T2> const operator+(const vecxpr<T1> & v1, const vecxpr<T2> & v2) {
    return vecsum<T1,T2>(v1,v2);
}
```
Expression templates
Design with CRTP

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

```cpp
template <typename T1, typename T2>
class vecsum : public vecxpr<vecsum<T1,T2>> {
  const T1 & lhs;
  const T2 & rhs;

public:
  using value_type=typename T1::value_type;
  vecsum(const vecxpr<T1> & l, const vecxpr<T2> & r) : lhs(l), rhs(r) {
    assert(l.size()==r.size());
  }
  const auto operator[](size_t i) const { return lhs[i] + rhs[i]; }
  size_t size() const { return lhs.size(); }
};
```

- `operator+` can now be written as:

```cpp
template <typename T1, typename T2>
vecsum<T1,T2> const operator+(const vecxpr<T1> & v1, const vecxpr<T2> & v2) {
  return vecsum<T1,T2>(v1,v2);
}
```
Expression templates
Design with CRTP

We also make the original `vec` class inherit from `vecxpr`

```cpp
template <typename T> class vec : public vecxpr<vec<T>> {  
std::vector<T> dat;
public:
  using value_type = T;
  vec(size_t n) : dat(n) {}
  inline const T operator[](size_t i) const { return dat[i]; }
  inline T & operator[](size_t i) { return dat[i]; }
  inline size_t size() const { return dat.size(); }
  inline size_t n_ops() const { return 0; }
  template <typename X>
  vec & operator=(const vecxpr<X> & y) {
    dat.resize(y.size());
    for (size_t i=0; i<y.size(); ++i) dat[i]=y[i];
    return *this;
  }
};
```
Expression templates
Design with CRTP

- We also make the original `vec` class inherit from `vecxpr`.

```cpp
template <typename T> class vec : public vecxpr<vec<T>> {
    std::vector<T> dat;

public:
    using value_type = T;
    vec(size_t n) : dat(n) {} 
    inline const T operator[](size_t i) const { return dat[i]; } 
    inline T & operator[](size_t i) { return dat[i]; } 
    inline size_t size() const { return dat.size(); } 
    inline size_t n_ops() const { return 0; } 
    template <typename X>
    vec & operator=(const vecxpr<X> & y) {
        dat.resize(y.size());
        for (size_t i=0; i<y.size(); ++i) dat[i]=y[i];
        return *this;
    }
};
```

- Notice the special assignment operator from an expression!
Expression templates

\[ a \times X + b \times Y + Z; \]

// Let's call this type EXPR

\[ W = a \times X + b \times Y + Z; \]
Expression templates

```cpp
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;
}
```

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

\[ W = a \times X + b \times Y + Z; \]

```cpp
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;
}
```

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

- Elegant high level syntax
Expression templates

- Elegant high level syntax
- Reduce temporaries
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
Expression templates

- Elegant high level syntax
- Reduce temporaries
- Loop fusion
- Delayed evaluation: apply algorithmic optimizations on the entire expression, e.g.,

Evaluate $\text{Matrix1} \times \text{Matrix2} \times \text{Vector}$ as $\text{Matrix1} \times (\text{Matrix2} \times \text{Vector})$

Detect and eliminate cancelling operations, e.g., $\text{Matrix} \_\text{xpr1} \cdot \text{transpose()} \cdot \text{transpose()}$

Use optimized low level kernels with assembler, intrinsics, calls to vendor libraries etc to do the work

However, can greatly increase compilation times
Expression templates

- Elegant high level syntax
- Reduce temporaries
- Loop fusion
- Delayed evaluation: apply algorithmic optimizations on the entire expression, e.g.,
  - Evaluate $\text{Matrix1} \ast \text{Matrix2} \ast \text{Vector}$ as $\text{Matrix1} \ast (\text{Matrix2} \ast \text{Vector})$
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_xprl.transpose().transpose()
Expression templates

- Elegant high level syntax
- Reduce temporaries
- Loop fusion
- Delayed evaluation: apply algorithmic optimizations on the entire expression, e.g.,
  - Evaluate $\text{Matrix1} \times \text{Matrix2} \times \text{Vector}$ as $\text{Matrix1} \times (\text{Matrix2} \times \text{Vector})$
  - Detect and eliminate cancelling operations, e.g., $\text{Matrix\_xprl\_transpose()\_transpose()}$
  - Use optimized low level kernels with assembler, intrinsics, calls to vendor libraries etc to do the work
Expression templates

- Elegant high level syntax
- Reduce temporaries
- Loop fusion
- Delayed evaluation: apply algorithmic optimizations on the entire expression, e.g.,
  - Evaluate $\text{Matrix1} \times \text{Matrix2} \times \text{Vector}$ as $\text{Matrix1} \times (\text{Matrix2} \times \text{Vector})$
  - Detect and eliminate cancelling operations, e.g.,
    $\text{Matrix}_xprl\text{.transpose()}.\text{transpose()}$
  - Use optimized low level kernels with assembler, intrinsics, calls to vendor libraries etc to do the work
- However, can greatly increase compilation times
Example 16:

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 $W = a \times X + 2 \times a \times Y + 3 \times a \times Z$, 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_vec.hh. Compile and run the program by alternating between the two headers. Study the code in xtmp_vec.hh, which illustrates the ideas presented here about expression templates.
Exercise 3:

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 $M_1 \times M_2 \times v$ is evaluated as two matrix vector products rather than a matrix-matrix product followed by a matrix vector product.
Linear Algebra
Linear algebra

- Operations on matrices, vectors, linear systems etc.
- Data parallel, simple numerical calculations
- Can be hand coded, but taking proper account of available CPU instructions, memory hierarchy etc is hard
- Libraries with standardized syntax for wide applicability
- Excellent vendor libraries are available on HPC systems
Eigen: A C++ template library for linear algebra

- Include only library. Download from http://eigen.tuxfamily.org/, unpack in a location of your choice, and use. Nothing to link.
- Small fixed size to large dense/sparse matrices
- Matrix operations, numerical solvers, tensors ...
- Expression templates: lazy evaluation, smart removal of temporaries

// examples/Eigen/eigen1.cc
#include <iostream>
#include <Eigen/Dense>
using namespace Eigen;
using namespace std;
int main()
{
    MatrixXd m=MatrixXd::Random(3,3);
m = (m+MatrixXd::Constant(3,3,1.2))*50;
    cout << "m =" << endl << m << endl;
    VectorXd v(3);
v << 1, 2, 3;
    cout <<"m * v ="<<endl<<m*v<<endl;
}

$ G -eigen eigen1.cc

- Explicit vectorization
- Elegant API
Eigen: matrix types

- MatrixXd: matrix of arbitrary dimensions
- Matrix3d: fixed size $3 \times 3$ matrix
- Vector3d: fixed size 3d vector
- Element access $m(i, j)$
- Output `std::cout << m << "\n";`
- Constant: `MatrixXd::Constant(a, b, c)`
- Random: `MatrixXd::Random(n, n)`
- Products: $m \times v$ or $m_1 \times m_2$
- Expressions: $3 \times m \times m \times v_1 + u \times v_2 + m \times m \times m$
- Column major matrix:
  `MatrixXd<\text{float}, 3, 10, Eigen::ColMajor>`
## Eigen: matrix operations

```cpp
#include <iostream>
#include <Eigen/Dense>
using namespace std;
using namespace Eigen;
int main()
{
    Matrix3f A;
    Vector3f b;
    A << 1,2,3, 4,5,6, 7,8,10;
    b << 3, 3, 4;
    cout << "Here is the matrix A:\n" << A << endl;
    cout << "Here is the vector b:\n" << b << endl;
    Vector3f x = A.colPivHouseholderQr().solve(b);
    cout << "The solution is:\n" << x << endl;
}
```

- **Blocks** `m.block(start_r, start_c, nr, nc), or m.block<nr,nc>(start_r, start_c)

```cpp
SelfAdjointEigenSolver<Matrix2f> eigensolver(A);
if (eigensolver.info() != Success) abort();
cout << "Eigenvalues << eigensolver.eigenvalues() << endl;
```
Eigen: examples

Example 17:

There are a few example programs using Eigen in the folder examples/Eigen. Read the programs eigen0.cc and eigen1.cc. To compile, use G -eigen program.cc.

Exercise 4:

The folder examples/Eigen contains a matrix multiplication example, matmul.cc using Eigen. Compare with a naive version of a matrix multiplication program, matmul_naive.cc, by compiling and running both programs. Try different matrix sizes. Then, you can use a parallel version of the Eigen matrix multiplication by recompiling with -fopenmp.
Exercise 5:

The file exercises/PCA has a data file with tabular data. Each column represents all measurements of a particular type, while each row is a different trial. In each row, the first column, $x_{i0}$, represents a pseudo-time variable. Write a program using Eigen to perform a Principal Component Analysis on this data set, ignoring the first column. Hint:

if $X_i = [x_{i1}, x_{i2}, ... x_{im}]$ is the data of row $i$, the covariance matrix is defined as,

$$C_{ab} = \frac{1}{(n-1)} \sum_k x_{ka} x_{kb}$$

The principal components of the data are obtained by right multiplying the data matrix by the matrix whose columns are the eigen vectors of the matrix $C_{ab}$, conventionally ordered by decreasing eigenvalues.
Eigen: some issues

```cpp
Eigen::Tensor<double, 3> epsilon(3,3,3);
epsilon.setZero();
Eigen::SGroup<Eigen::AntiSymmetry<0,1>, Eigen::AntiSymmetry<1,2>> sym;
sym(epsilon, 0, 1, 2) = 1;
```

- "Standard C++98": Operates within the obsolete constraints of old C++
Eigen: some issues

- "Standard C++98": Operates within the obsolete constraints of old C++
- Evolution of the language standard opens up new possibilities. Case in point: `Eigen::Tensor` (example above) elegantly leverages variadic templates
Eigen: some issues

- "Standard C++98": Operates within the obsolete constraints of old C++
- Evolution of the language standard opens up new possibilities. Case in point: `Eigen::Tensor` (example above) elegantly leverages variadic templates
- Performance lags behind vendor libraries (e.g., Intel MKL on JURECA) for strictly BLAS problems. But note: Eigen can use MKL behind the scenes.

```cpp
Eigen::Tensor<double, 3> epsilon(3,3,3);
epsilon.setZero();
Eigen::SGroup<Eigen::AntiSymmetry<0,1>, Eigen::AntiSymmetry<1,2>> sym;
sym(epsilon, 0, 1, 2) = 1;
```
Exercise 6:

Recompile the Eigen matrix multiplication example, this time to use intel MKL library (on JURECA). The procedure is described here: [http://eigen.tuxfamily.org/dox/TopicUsingIntelMKL.html](http://eigen.tuxfamily.org/dox/TopicUsingIntelMKL.html). On JURECA, using the Eigen module used in the course, you can do the following: `G -eigen-mkl matmul.cc`. Run it on one node of JURECA and compare with the previous exercise.
Threading Building Blocks
TBB: Threading Building Blocks I

- Parallel programming constructs for the end user
- Template library rather than language extensions
- Provides utilities like `parallel_for`, `parallel_reduce` to simplify the most commonly used structures in parallel programs
- Provides scalable concurrent containers such as vectors, hash tables and queues for use in multi-threaded environments
- **No direct support for vector parallelism.** But can be combined with auto-parallelisation and `#pragma simd` etc from Cilk
- Supports complex models such as pipelines, data flow and unstructured task graphs
- Scalable memory allocation, thread local storage
**TBB: Threading Building Blocks II**

- Low level synchronisation tools like mutexes and atomics
- Work stealing task scheduler
- [http://www.threadingbuildingblocks.org](http://www.threadingbuildingblocks.org)
- *Structured Parallel Programming*, Michael McCool, Arch D. Robinson, James Reinders
Using TBB

- Include `tbb/tbb.h` in your file
- Public names are available under the namespaces `tbb` and `tbb::flow`
- You indicate "available parallelism", scheduler may run it in parallel if resources are available
- Unnecessary parallelism will be ignored
parallel invoke

```cpp
void prep(Population &p);
void iomanage();
tbb::parallel_invoke(prep,iomanage,
    [other]{
        other.some_member();
    });
```

- A few adhoc tasks which do not depend on each other
- Runs them in parallel
- Waits until all of them are finished

Example 18: examples/TBB/-parallel_invoke.cc

Compile with `G -tbb parallel_invoke.cc`
TBB task groups

- Run an arbitrary number of callable objects in parallel

```cpp
struct Equation {
    void solve();
};

std::list<Equation> equations;
tbb::task_group g;
for (auto eq : equations)
    g.run([&]{eq.solve();});
g.wait();
```
TBB task groups

- Run an arbitrary number of callable objects in parallel
- In case an exception is thrown, the task group is cancelled
TBB task scheduler

```cpp
int main(int argc, char *argv[]) {
    size_t nthreads = std::stoul(argv[1]);
    /* tbb::task_scheduler_init nit; */
    tbb::task_scheduler_init nit(nthreads);
    haha();
}
void haha() {
    ...
    tbb::parallel_invoke(a, b, c, d, e);
}
void a() {
    tbb::parallel_for(...);
}
```

- Task scheduler: manages tasks, maps them to threads etc.
- Initializes the task scheduler
- Default constructor creates threads as needed while resources permit
- One scheduler is enough
Parallel for loops

- Template function modelled after the `for` loops, like many STL algorithms

```cpp
#include <tbb/concurrent_for.h>

int main()
{
    tbb::parallel_for(first, last, f);
    // parallel equivalent of
    // for (auto i=first; i<last; ++i) f(i);

    tbb::parallel_for(first, last, stride, f);
    // parallel equivalent of
    // for (auto i=first; i<last; i+=stride)
    // f(i);

    tbb::parallel_for(first, last,
                      [captures](anything){
                         //Code that can run in parallel
                      });
}
```
Parallel for loops

- Template function modelled after the `for` loops, like many STL algorithms
- Takes a **callable object** as the third argument

```cpp
#include <tbb/parallel_for.h>

void parallel_for_example()
{
    tbb::parallel_for(first, last, f);
    // parallel equivalent of
    // for (auto i=first; i<last; ++i) f(i);

    tbb::parallel_for(first, last, stride, f);
    // parallel equivalent of
    // for (auto i=first; i<last; i+=stride)
    //    f(i);

    tbb::parallel_for(first, last,
                      //Code that can run in parallel
                      [captures](anything){ ... });
}
```
Parallel for loops

- Template function modelled after the `for` loops, like many STL algorithms
- Takes a **callable object** as the third argument
- Using lambda functions, you can expose parallelism in sections of your code

```cpp
#include <tbb/parallel_for.h>

tbb::parallel_for(first, last, f);
// parallel equivalent of
// for (auto i=first;i<last;++i) f(i);

tbb::parallel_for(first, last, stride, f);
// parallel equivalent of
// for (auto i=first;i<last;i+=stride)
//   f(i);

tbb::parallel_for(first, last,
  [captures](anything){
    //Code that can run in parallel
  });
```
Parallel for with ranges

- Splits range into smaller ranges, and applies \( f \) to them in parallel.

```cpp
tbb::parallel_for(0, 1000000, f);
// One parallel invocation for each i!
tbb::parallel_for(range, f);

// A type R can be a range if the following are available
R::R(const R &);
R::~R();
bool R::is_divisible() const;
bool R::empty() const;
R::R(R & r, split);//Split constructor
```
Parallel for with ranges

- Splits range into smaller ranges, and applies \( f \) to them in parallel
- Possible to optimize \( f \) for sub-ranges rather than a single index

```
tbb::parallel_for(0, 1000000, f);
// One parallel invocation for each i!
tbb::parallel_for(range, f);

// A type R can be a range if the following are available
R::R(const R &);
R::~R();
bool R::is_divisible() const;
bool R::empty() const;
R::R(R & r, split); // Split constructor
```
Parallel for with ranges

- Splits range into smaller ranges, and applies \( f \) to them in parallel
- Possible to optimize \( f \) for sub-ranges rather than a single index
- Any type satisfying a few design conditions can be used as a range

```
tbb::parallel_for(0, 1000000, f);
// One parallel invocation for each i!
tbb::parallel_for(range, f);
// A type R can be a range if the following are available
R::R(const R &);
R::~R();
bool R::is_divisible() const;
bool R::empty() const;
R::R(R & r, split);// Split constructor
```
Parallel for with ranges

- Splits range into smaller ranges, and applies $f$ to them in parallel
- Possible to optimize $f$ for sub-ranges rather than a single index
- Any type satisfying a few design conditions can be used as a range
- Multidimensional ranges possible

```cpp
tbb::parallel_for(0, 1000000, f);
// One parallel invocation for each i!
tbb::parallel_for(range, f);

// A type R can be a range if the following are available
R::R(const R &);
R::~R();
bool R::is_divisible() const;
bool R::empty() const;
R::R(R & r, split); // Split constructor
```
Parallel for with ranges

```
tbb::blocked_range<int> r{0,30,20};
assert(r.is_divisible());
bBlocked_range<int> s{r};
//Splitting constructor
assert(!r.is_divisible());
assert(!s.is_divisible());
```

- `tbb::blocked_range<int>(0,4)` represents an integer range 0..4
Parallel for with ranges

```cpp
#include <tbb/tbb.h>

int main() {
    tbb::blocked_range<int> r{0, 30, 20};
    assert(r.is_divisible());
    blocked_range<int> s{r};
    // Splitting constructor
    assert(!r.is_divisible());
    assert(!s.is_divisible());
}
```

- `tbb::blocked_range<int>(0, 4)` represents an integer range 0..4
- `tbb::blocked_range<int>(0, 50, 30)` represents two ranges, 0..25 and 26..50
Parallel for with ranges

```cpp
tbb::blocked_range<int> r{0,30,20};
assert(r.is_divisible());
blocked_range<int> s{r};
//Splitting constructor
assert(!r.is_divisible());
assert(!s.is_divisible());
```

- `tbb::blocked_range<int>(0, 4)` represents an integer range 0..4
- `tbb::blocked_range<int>(0, 50, 30)` represents two ranges, 0..25 and 26..50
  - So long as the size of the range is bigger than the "grain size" (third argument), the range is split
Parallel for with ranges

```cpp
void dasxpcy_tbb(double a, std::vector<double> &x, std::vector<double> &y) {
    tbb::parallel_for(tbb::blocked_range<int>(0, x.size()),
        [&] (tbb::blocked_range<int> r) {
            for (size_t i = r.begin(); i != r.end(); ++i) {
                y[i] = a * sin(x[i]) + cos(y[i]);
            }
        });
}
```

- `parallel_for` with a range uses split constructor to split the range as far as possible, and then calls `f(range)`, where `f` is the functional given to `parallel_for`
Parallel for with ranges

```cpp
void dasxpcy_tbb(double a, std::vector<double> &x, std::vector<double> &y) {
    tbb::parallel_for(tbb::blocked_range<int>(0, x.size()),
        [&](tbb::blocked_range<int> r){
            for (size_t i=r.begin(); i!=r.end(); ++i) {
                y[i] = a * sin(x[i]) + cos(y[i]);
            }
        });
}
```

- `parallel_for` with a range uses split constructor to split the range as far as possible, and then calls `f(range)`, where `f` is the functional given to `parallel_for`
- It is unlikely that you wrote your useful functions with ranges compatible with `parallel_for` as arguments
Parallel for with ranges

```cpp
void dasxpcy_tbb(double a, std::vector<double> &x, std::vector<double> &y) {
    tbb::parallel_for(tbb::blocked_range<int>(0,x.size()),
                      [&](tbb::blocked_range<int> r){
                          for (size_t i=r.begin();i!=r.end();++i) {
                              y[i]=a*sin(x[i])+cos(y[i]);
                          }
                      });
}
```

- `parallel_for` with a range uses split constructor to split the range as far as possible, and then calls `f(range)`, where `f` is the functional given to `parallel_for`
- It is unlikely that you wrote your useful functions with ranges compatible with `parallel_for` as arguments
- But with lambda functions, it is easy to fit the parts!
Example 19: TBB parallel for demo

The program examples/dasxpcy.cc demonstrates the use of parallel for in TBB. It is a slightly modified version of the commonly used DAXPY demos. Instead of calculating \( y = a \times x + y \) for scalar \( a \) and large vectors \( x \) and \( y \), we calculate \( y = a \times \sin(x) + \cos(y) \). To compile, you need to load your compiler and TBB modules, and use them like this:

```
G -I$TBB_INCLUDE_DIR dasxpcy.cc -L$TBB_LIBRARY_DIR -ltbb -ltbbaumalloc
```
2D ranges

```cpp
void f(size_t i, size_t j);
tbb::blocked_range2d<size_t> r{0,N,0,N};
tbb::parallel_for(r, [&] (tbb::blocked_range2d<size_t> r){
    for (size_t i=r.rows().begin(); i!=r.rows().end(); ++i) {
        for (size_t j=r.cols().begin(); j!=r.cols().end(); ++j) {
            f(i, j);
        }
    }
});
```

- `rows()` is an object with a `begin()` and an `end()` returning just the integer row values in the range. Similarly: `cols()`...
2D ranges

void f(size_t i, size_t j);
tbb::blocked_range2d<size_t> r{0,N,0,N};
tbb::parallel_for(r,[&](tbb::blocked_range2d<size_t> r){
  for (size_t i=r.rows().begin();i!=r.rows().end();++i) {
    for (size_t j=r.cols().begin();j!=r.cols().end();++j) {
      f(i,j);
    }
  }
});

- rows() is an object with a begin() and an end() returning just the integer row values in the range. Similarly: cols() ...
- 2D range can also be split
void f(size_t i, size_t j);
tbb::blocked_range2d<size_t> r{0,N,0,N};
tbb::parallel_for(r,[&](tbb::blocked_range2d<size_t> r){
  for (size_t i=r.rows().begin();i!=r.rows().end();++i) {
    for (size_t j=r.cols().begin();j!=r.cols().end();++j) {
      f(i,j);
    }
  }
});

- rows() is an object with a begin() and an end() returning just the integer row values in the range. Similarly: cols() ...
- 2D range can also be split
- The callable object argument should assume that the original 2D range has been split many times, and we are operating on a smaller range, whose properties can be accessed with these functions.
Parallel reductions with ranges

```cpp
T result = tbb::parallel_reduce(range, identity, subrange_reduction, combine);
```

- **range**: As with parallel for
- **identity**: Identity element of type T. The type determines the type used to accumulate the result
- **subrange_reduction**: Functor taking a "subrange" and an initial value, returning reduction
- **combine**: Functor taking two arguments of type T and returning reduction over them over the subrange. Must be associative, but not necessarily commutative.
Parallel reduce with ranges

```cpp
double inner_prod_tbb(std::vector<double> & x, std::vector<double> & y) {
    return tbb::parallel_reduce(
        tbb::blocked_range<int>(0,n), // range
        double{}, // identity
        [&](tbb::blocked_range<int> &r,float in){
            return std::inner_product(x.begin()+r.begin(),x.begin()+r.end(),
                                       y.begin()+r.begin(),in);
        }, // subrange reduction
        std::plus<double>{} // combine
    );
}
```

- With TBB ranges, we can use blocked implementations with hopefully vectorisable calculations in subranges
Parallel reduce with ranges

```cpp
double inner_prod_tbb(std::vector<double> & x, std::vector<double> & y) {
    return tbb::parallel_reduce(
        tbb::blocked_range<int>(0,n), // range
        double{}, // identity
        [&](tbb::blocked_range<int> &r,float in){
            return std::inner_product(x.begin()+r.begin(),x.begin()+r.end(),
                                      y.begin()+r.begin(),in);
        }, // subrange reduction
        std::plus<double>{} // combine
    );
}
```

- With TBB ranges, we can use blocked implementations with hopefully vectorisable calculations in subranges
- Two functors are required, either of which could be lambda functions
Parallel reduce with ranges

```cpp
double inner_prod_tbb(std::vector<double> & x, std::vector<double> & y) {
    return tbb::parallel_reduce(
        tbb::blocked_range<int>(0,n), // range
        double{}, // identity
        [&](tbb::blocked_range<int> &r,float in){
            return std::inner_product(x.begin()+r.begin(),x.begin()+r.end(),
                                        y.begin()+r.begin(),in);
        }, // subrange reduction
        std::plus<double>{} // combine
    );
}
```

- With TBB ranges, we can use blocked implementations with hopefully vectorisable calculations in subranges
- Two functors are required, either of which could be lambda functions
- Important to add the contribution of initial value in subrange reductions
Example 20: TBB parallel reduce

The program `tbbreduce.cc` is a demo program to calculate an integral using `tbb::parallel_reduce`. Check how lambda functions are used to do the integral. What kind of speed up do you see relative to the serial version? Does it make sense considering the number of physical cores in your computer?
Atomic variables

- "Instantaneous" updates

```cpp
std::array<double,N> A;
tbb::atomic<int> index;

void append(double val)
{
    A[index++]=val;
}
```
Atomic variables

- "Instantaneous" updates
- Lock-free synchronization

```cpp
std::array<double,N> A;
tbb::atomic<int> index;

void append(double val)
{
   A[index++]=val;
}
```
Atomic variables

- "Instantaneous" updates
- Lock-free synchronization
- For `tbb::atomic<T>`, T can be integral, enum or pointer type

```cpp
std::array<double,N> A;
tbb::atomic<int> index;

void append(double val)
{
    A[index++]=val;
}
```
Atomic variables

- "Instantaneous" updates
- Lock-free synchronization
- For `tbb::atomic<T>`, `T` can be integral, enum or pointer type
- If `index==k` simultaneous calls to `index++` by `n` threads will increase `index` to `k+n`. Each thread will use a distinct value between `k` and `k+n`

```cpp
std::array<double,N> A;
tbb::atomic<int> index;

void append(double val)
{
    A[index++] = val;
}
```

But it is important that we use the return value of `index++` in the threads!
Enumerable thread specific

```cpp
tbb::enumerable_thread_specific<double> E;
double Eglob=0;
double f(size_t i, size_t j);
tbb::blocked_range2d<size_t> r{0,N,0,N};
tbb::parallel_for(r,[&](tbb::blocked_range2d<size_t> r){
    auto & eloc=E.local();
    for (size_t i=r.rows().begin();i!=r.rows().end();++i) {
        for (size_t j=r.cols().begin();j!=r.cols().end();++j) {
            if (j>i) eloc += f(i,j);
        }
    }
});
Eglob=0;
for (auto & v : E) {Eglob+=v;v=0;}
```

- Thread local "views" of a variable
Enumerable thread specific

```cpp
tbb::enumerable_thread_specific<double> E;
double Eglob=0;
double f(size_t i, size_t j);
tbb::blocked_range2d<size_t> r{0,N,0,N};
tbb::parallel_for(r,[&](tbb::blocked_range2d<size_t> r){
    auto & eloc=E.local();
    for (size_t i=r.rows().begin();i!=r.rows().end();++i) {
        for (size_t j=r.cols().begin();j!=r.cols().end();++j) {
            if (j>i) eloc += f(i,j);
        }
    }
});
Eglob=0;
for (auto & v : E) {Eglob+=v;v=0;}
```

- Thread local "views" of a variable
- behaves like an STL container of those views
Enumerable thread specific

```cpp
tbb::enumerable_thread_specific<double> E;
double Eglob=0;
double f(size_t i, size_t j);
tbb::blocked_range2d<size_t> r{0,N,0,N};
tbb::parallel_for(r,[&](tbb::blocked_range2d<size_t> r){
    auto & eloc=E.local();
    for (size_t i=r.rows().begin();i!=r.rows().end();++i) {
        for (size_t j=r.cols().begin();j!=r.cols().end();++j) {
            if (j>i) eloc += f(i,j);
        }
    }
});
Eglob=0;
for (auto & v : E) {Eglob+=v;v=0;}
```

- Thread local "views" of a variable
- behaves like an STL container of those views
- Member function `local()` gives a reference to the local view in the current thread
**Enumerable thread specific**

```cpp
tbb::enumerable_thread_specific<double> E;
double Eglob=0;
double f(size_t i, size_t j);
tbb::blocked_range2d<size_t> r{0,N,0,N};
tbb::parallel_for(r,[&](tbb::blocked_range2d<size_t> r){
    auto & eloc=E.local();
    for (size_t i=r.rows().begin();i!=r.rows().end();++i) {
        for (size_t j=r.cols().begin();j!=r.cols().end();++j) {
            if (j>i) eloc += f(i,j);
        }
    }
});
Eglob=0;
for (auto & v : E) {Eglob+=v;v=0;}
```

- Thread local "views" of a variable
- behaves like an STL container of those views
- Member function `local()` gives a reference to the local view in the current thread
- Any thread can access all views by treating it as an STL container
Example 21: Reduction with enumerable thread specific

You can use the `enumerable_thread_specific` and `parallel_for` to implement reduction. The program `examples/tbbreduce1.cc` demonstrates this.
TBB allocators

- Dynamic memory allocation in a multithreaded program must avoid conflicts from \texttt{new} calls from different threads
- Reduce global memory locks

TBB allocators

- Interface like \texttt{std::allocator}, so that it can be used with STL containers. E.g.,
  \texttt{std::vector\langle T, tbb::cache_aligned_allocator\langle T\rangle\rangle}
- \texttt{tbb::scalable_allocator\langle T\rangle}: general purpose scalable allocator type, for rapid allocation from multiple threads
- \texttt{tbb::cache_aligned_allocator\langle T\rangle}: Allocates with cache line alignment. As a consequence, objects allocated in different threads are guaranteed to be in different cache lines.
Concurrent containers

```cpp
#include <tbb/concurrent_vector.h>

auto v=tbb::concurrent_vector<int>(N,0);

for (int i=0; i<N; ++i)
    v[i]=i;
```

- Random access by index
Concurrent containers

```cpp
#include <tbb/concurrent_vector.h>

auto v=tbb::concurrent_vector<int>(N,0);

tbb::parallel_for(v.range(),[
    &
    ](tbb::concurrent_vector::range_type r){
       //...
    });
```

- Random access by index
- Multiple threads can grow container and add elements concurrently
Concurrent containers

```cpp
#include <tbb/concurrent_vector.h>

auto v = tbb::concurrent_vector<int>(N, 0);

tbb::parallel_for(v.range(), [&] (tbb::concurrent_vector::range_type r)
{ //...
});
```

- Random access by index
- Multiple threads can grow container and add elements concurrently
- Growing the container does not invalidate any iterators or indexes
Concurrent containers

```cpp
#include <tbb/concurrent_vector.h>

auto v=tbb::concurrent_vector<int>(N,0);

tbb::parallel_for(v.range(),[&](tbb::concurrent_vector::range_type r){
    //...
});
```

- Random access by index
- Multiple threads can grow container and add elements concurrently
- Growing the container does not invalidate any iterators or indexes
- Has a `range()` member function for use with `parallel_for` etc.
Exercise 7: N particle systems with pairwise interactions

Use the `enumerable_thread_specific` and `parallel_for` to calculate the pairwise interactions in an N-particle system.
Thrust
NVIDIA Thrust

```cpp
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/generate.h>
#include <thrust/sort.h>
#include <thrust/copy.h>
#include <cstdlib>
using namespace thrust;

int main()
{
    // generate 32 M random numbers on
    // the host
    host_vector<int> h_vec(32 << 20);
    generate(h_vec.begin(), h_vec.end(),
             rand);

    // transfer data to the device
    device_vector<int> d_vec = h_vec;
    // sort data on the device (846M keys
    // per second on GeForce GTX 480)
    sort(d_vec.begin(), d_vec.end());
    // transfer data back to the host
    copy(d_vec.begin(), d_vec.end(),
         h_vec.begin());
}
```

- Template library like STL or TBB for CUDA, with great documentation.
NVIDIA Thrust

Template library like STL or TBB for CUDA, with great documentation.

- Provides an elegant high level syntax to clearly express the intent of the programmer

```cpp
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/generate.h>
#include <thrust/sort.h>
#include <thrust/copy.h>
#include <cstdlib>
using namespace thrust;
int main()
{
    // generate 32 M random numbers on the host
    host_vector<int> h_vec(32 << 20);
    generate(h_vec.begin(), h_vec.end(),
             rand);

    // transfer data to the device
    device_vector<int> d_vec = h_vec;
    // sort data on the device (846M keys per second on GeForce GTX 480)
    sort(d_vec.begin(), d_vec.end());
    // transfer data back to the host
    copy(d_vec.begin(), d_vec.end(),
         h_vec.begin());
}
```
NVIDIA Thrust

Template library like STL or TBB for CUDA, with great documentation.

Provides an elegant high level syntax to clearly express the intent of the programmer.

The compiler translates the stated intents to efficient code for the GPU.

```cpp
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/generate.h>
#include <thrust/sort.h>
#include <thrust/copy.h>
#include <cstdlib>
using namespace thrust;
int main()
{
    // generate 32 M random numbers on the host
    host_vector<int> h_vec(32 << 20);
    generate(h_vec.begin(), h_vec.end(), rand);

    // transfer data to the device
    device_vector<int> d_vec = h_vec;
    // sort data on the device (846M keys per second on GeForce GTX 480)
    sort(d_vec.begin(), d_vec.end());
    // transfer data back to the host
    copy(d_vec.begin(), d_vec.end(), h_vec.begin());
}
```
NVIDIA Thrust

The bad

- Lag in supported C++ features => Syntax degradation to older practices.

As of CUDA 8, C++14 is not supported in .cu files! Only "experiental" support for even 6 year old C++11 version of lambda functions.
The bad

- Lag in supported C++ features $\Rightarrow$ Syntax degradation to older practices.
  - As of CUDA 8, C++14 is not supported in .cu files!
NVIDIA Thrust

The bad

- Lag in supported C++ features => Syntax degradation to older practices.
  - As of CUDA 8, C++14 is not supported in .cu files!
  - Only “experiential” support for even 6 year old C++11 version of lambda functions
NVIDIA Thrust

The bad

- Lag in supported C++ features => Syntax degradation to older practices.
  - As of CUDA 8, C++14 is not supported in .cu files!
  - Only “experiental” support for even 6 year old C++11 version of lambda functions
- NVIDIA GPUs only
NVIDIA Thrust

The bad

- Lag in supported C++ features $\Rightarrow$ Syntax degradation to older practices.
  - As of CUDA 8, C++14 is not supported in .cu files!
  - Only “experiential” support for even 6 year old C++11 version of lambda functions
- NVIDIA GPUs only

- Host exclusive C++14 code can be compiled directly with the underlying compiler, and combined inside a project with C++11 restricted .cu files containing thrust code
NVIDIA Thrust

```cpp
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/generate.h>
#include <thrust/sort.h>
#include <thrust/copy.h>
#include <cstdlib>
using namespace thrust;

int main()
{
    // generate 32 M random numbers on
    // the host
    host_vector<int> h_vec(32 << 20);
    generate(h_vec.begin(), h_vec.end(),
             rand);

    // transfer data to the device
    device_vector<int> d_vec = h_vec;
    // sort data on the device (846M keys
    // per second on GeForce GTX 480)
    sort(d_vec.begin(), d_vec.end());

    // transfer data back to the host
    copy(d_vec.begin(), d_vec.end(),
         h_vec.begin());
}
```

- Example:
  - `thrust::host_vector` and `thrust::device_vector` use the assignment operator to transfer data between the CPU and the GPU.
  - Thrust algorithms like `thrust::sort` have syntax like STL algorithms.
  - Many data parallel general operations have their own algorithms: `transform`, `reduce`, `inclusive_scan`,
**NVIDIA Thrust**

```cpp
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/generate.h>
#include <thrust/sort.h>
#include <thrust/copy.h>
#include <cstdlib>
using namespace thrust;

int main()
{
    // generate 32 M random numbers on the host
    host_vector<int> h_vec(32 << 20);
    generate(h_vec.begin(), h_vec.end(), rand);

    // transfer data to the device
    device_vector<int> d_vec = h_vec;

    // sort data on the device (846M keys per second on GeForce GTX 480)
    sort(d_vec.begin(), d_vec.end());

    // transfer data back to the host
    copy(d_vec.begin(), d_vec.end(), h_vec.begin());
}
```

- **Example:**
  - `thrust::host_vector` and `thrust::device_vector` use the assignment operator to transfer data between the CPU and the GPU
  - Thrust algorithms like `thrust::sort` have syntax like STL algorithms
NVIDIA Thrust

```cpp
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/generate.h>
#include <thrust/sort.h>
#include <thrust/copy.h>
#include <cstdlib>
using namespace thrust;

int main()
{
    // generate 32 M random numbers on
    // the host
    host_vector<int> h_vec(32 << 20);
    generate(h_vec.begin(), h_vec.end(), rand);

    // transfer data to the device
    device_vector<int> d_vec = h_vec;
    // sort data on the device (846M keys
    // per second on GeForce GTX 480)
    sort(d_vec.begin(), d_vec.end());

    // transfer data back to the host
    copy(d_vec.begin(), d_vec.end(), h_vec.begin());
}
```

- Example:
  thrust::host_vector and thrust::device_vector use the assignment operator to transfer data between the CPU and the GPU.

- Thrust algorithms like thrust::sort have syntax like STL algorithms.

- Many data parallel general operations have their own algorithms: transform, reduce, inclusive_scan.
Containers host_vector and device_vector are designed similar to std::vector, but (as of CUDA 8.0), do not have initializer list constructors or new member functions of std::vector like emplace_back.
Host and device vectors

Containers `host_vector` and `device_vector` are designed similar to `std::vector`, but (as of CUDA 8.0), do not have initializer list constructors or new member functions of `std::vector` like `emplace_back`.

- The overloaded assignment operators can copy data across devices.

```cpp
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <iostream>

int main()
{
    thrust::host_vector<int> H(4);
    for (int i=0;i<4;++i) H[i]=i;
    // resize H
    H.resize(2);
    std::cout << "H now has size "
             << H.size() << std::endl;
    // Copy host_vector H to
    // device_vector D
    thrust::device_vector<int> D = H;
    // elements of D can be modified
    D[0] = 99;
    D[1] = 88;
    // print contents of D
    for(int i = 0; i < D.size(); i++)
        std::cout << "D[" << i << "] = "
                  << D[i] << std::endl;
}
```
Other initialization options

```cpp
// initialize all ten integers to 1
thrust::device_vector<int> D(10, 1);
// set the first seven elements to 9
thrust::fill(D.begin(), D.begin() + 7, 9);
// initialize a host_vector with
// the first five elements of D
thrust::host_vector<int> H(D.begin(), D.begin() + 5);
// set elements of H to 0, 1, 2, ...
thrust::sequence(H.begin(), H.end());
```

- Many algorithms to provide initial values, to serve different purposes.
- There is also `thrust::generate` which can call a functional for every element of the vector
- The type of the iterators tell the compiler which version of the respective algorithms to use. No run-time overhead
Example 22:

The example programs examples/thrust/test0.cc and examples/thrust/test1.cc contain the thrust code in the previous slides. Run them on JURECA using the following steps:

- Find out what CUDA modules are installed and load the most recent one you can find
- Compile using the `nvcc` compiler: `nvcc test0.cu`
- Try changing the file name to `test0.cc`
- Consult the information sheet distributed in the class room about how to run your programs on JURECA GPU nodes.
Thrust algorithms

- Host and device versions

```cpp
device_vector<int> X(10), Y(10), Z(10);
// initialize X to 0,1,2,3, ....
sequence(X.begin(), X.end());
// compute Y = -X
thrust::transform(X.begin(), X.end(),
                  Y.begin(), thrust::negate<int>())
// fill Z with twos
thrust::fill(Z.begin(), Z.end(), 2);
// compute Y = X mod 2
thrust::transform(X.begin(), X.end(),
                  Z.begin(), Y.begin(),
                  thrust::modulus<int>())
// replace all the ones in Y with 10
thrust::replace(Y.begin(), Y.end(), 1, 10);
// print Y
thrust::copy(Y.begin(), Y.end(),
             std::ostream_iterator<int>(cout, "\n"));
```
Thrust algorithms

```cpp
device_vector<int> X(10), Y(10), Z(10);
// initialize X to 0,1,2,3,...
sequence(X.begin(), X.end());
// compute Y = -X
thrust::transform(X.begin(), X.end(),
    Y.begin(), thrust::negate<int>());
// fill Z with twos
thrust::fill(Z.begin(), Z.end(), 2);
// compute Y = X mod 2
thrust::transform(X.begin(), X.end(),
    Z.begin(), Y.begin(),
    thrust::modulus<int>());
// replace all the ones in Y with 10
thrust::replace(Y.begin(), Y.end(),
    1, 10);
// print Y
thrust::copy(Y.begin(), Y.end(),
    std::ostream_iterator<int>(cout, "\n"));
```

- Host and device versions
- A set of elementary functionals are available in thrust/functional.h
Thrust algorithms

- Host and device versions
- A set of elementary functionals are available in thrust/functional.h
- Notice the copy from a device vector to the ostream iterator!
Custom functionals for transforms

When pre-defined operations in thrust/functional.h do not suffice, we can write our own function objects.

The overloaded operator() must be marked with __host__ __device__
Custom functionals using placeholders

- For very simple operations, custom functionals can be generated inline using the `thrust::placeholders` namespace.

```c
void saxpy_fast(float A,
    thrust::device_vector<float>& X,
    thrust::device_vector<float>& Y)
{
    // Y <- A * X + Y
    thrust::transform(X.begin(), X.end(),
        Y.begin(), Y.begin(),
        (A*_1 +_2));
}
```

- `_1`, `_2` ... are placeholders
- Expressions involving placeholders yield a functional mapping its arguments sequentially to `_1`, `_2` ...
Custom functionals using lambda functions

- In CUDA 8, you can also use C++11 (but not C++14) style lambda functions

```c++
void saxpy_fast(float A,
    thrust::device_vector<float>& X,
    thrust::device_vector<float>& Y)
{
    // Y <- A * X + Y
    thrust::transform(X.begin(), X.end(),
        Y.begin(), Y.begin(),
        [A] __host__ __device__
        (double x, double y){
            return A * x + y;
        });
}
```

- Can have many lines of code with loops etc like a normal function

- Reference capture (thankfully!) not allowed

- Note where we mark the lambda function to be for the host and device

```
nvcc -std=c++11 --expt-extended-lambda\saxpy0.cu
```
Example 23: Placeholders and lambda functions

The example examples/saxpy0.cu shows how to use the placeholders with thrust algorithms for simple inline functionality. There is also a commented out version of the same thing done using a lambda function. The placeholder version is more compact, but the lambda version can have multiple statements, like a normal function.
Exercise 8: Mandelbrot set

The Mandelbrot set is the set of complex numbers $c$ for which the function $f(z) = z^2 + c$ does not diverge when iterated from $z = 0$. An image representing the set can be created by generating the sequence $z_n = z_{n-1}^2 + c$ for each pixel in the image, by treating the $x$ and $y$ values of the pixel as the real and imaginary components of $c$. The sequence can be taken to have diverged if the magnitude of $z$ exceeds 2. The program `exercises/mandelbrot.cc` does it, using the standard C++ library. Modify to do the computations using `thrust`. 
Reductions

Reductions require a binary operation and some initial value

For convenience, variants like count, count_if, inner_product exist

If a reduction is to follow a transform on the same data, transform_reduce offers an opportunity for "kernel fusion"

```cpp
int sum=thrust::reduce(D.begin(),D.end(),
    (int)0,thrust::plus<int>())
int sum=thrust::reduce(D.begin(),D.end(),
    (int)0);
int sum=
    thrust::reduce(D.begin(),D.end());
int result = thrust::count(vec.begin
    (),
    vec.end(), 1);
// thrust::count_if
// thrust::inner_product
float v=
    thrust::transform_reduce(d_x.begin(),
    d_x.end(),unary_op,init,binary_op) );
```
Partial sums, sorting, etc.

```c
int data[6] = {1, 0, 2, 2, 1, 3};
inclusive_scan(data, data+6, data);
exclusive_scan(data, data+6, data);
// data is now {0, 1, 1, 3, 5, 6}
thrust::sort(A, A + N);
const int N = 6;
int keys[N]={1,4,2,8,5,7};
char values[N]={'a','b','c','d','e','f'};
thrust::sort_by_key(keys, keys+N, values);
// keys is now {1,2,4,5,7,8}
// values is now {'a','c','b','e','f','d'}
thrust::stable_sort(A, A+N,
                   thrust::greater<int>());
```

- Frequently needed algorithms, which are not trivial to parallelize, have thrust implementations
Partial sums, sorting, etc.

```cpp
int data[6] = {1, 0, 2, 2, 1, 3};
inclusive_scan(data, data+6, data);
exclusive_scan(data, data+6, data);
// data is now {0, 1, 1, 3, 5, 6}
thrust::sort(A, A+N);
const int N = 6;
int keys[N] = {1, 4, 2, 8, 5, 7};
char values[N] = {'a', 'b', 'c', 'd', 'e', 'f'};
thrust::sort_by_key(keys, keys+N, values);
// keys is now {1, 2, 4, 5, 7, 8}
// values is now {'a', 'c', 'b', 'e', 'f', 'd'}
thrust::stable_sort(A, A+N,
    thrust::greater<int>());
```

- Frequently needed algorithms, which are not trivial to parallelize, have thrust implementations.
- Nicely hides low-level details and lets us work on the program logic.
Partial sums, sorting, etc.

```cpp
int data[6] = {1, 0, 2, 2, 1, 3};
inclusive_scan(data, data+6, data);
exclusive_scan(data, data+6, data);
// data is now {0, 1, 1, 3, 5, 6}
thrust::sort(A, A + N);
const int N = 6;
int keys[N]={1,4,2,8,5,7};
char values[N]={'a','b','c','d',
              'e','f'};
thrust::sort_by_key(keys, keys+N, values);
// keys is now {1,2,4,5,7,8}
// values is now {'a','c','b',
//              'e','f','d'}
thrust::stable_sort(A, A+N,
    thrust::greater<int>());
```

- Frequently needed algorithms, which are not trivial to parallelize, have thrust implementations.
- Nicely hides low-level details and lets us work on the program logic.
- The high-level syntax is parsed at compile time, and reduced to efficient system specific implementations. Overhead exists, but it is low.
Thrust iterator library

```cpp
thrust::constant_iterator<int> first(10);
first[0]  // returns 10
first[100] // returns 10

thrust::counting_iterator<int> first(10);
first[0]  // returns 10
first[1]  // returns 11
first[100] // returns 110

first = thrust::make_transform_iterator(vec.begin(), negate<int>());
...

last = thrust::make_transform_iterator(vec.end(), negate<int>());
thrust::reduce(first, last);  // returns -60 (i.e. -10 + -20 + -30)

thrust::device_vector<int> map(2);
map[0] = 3;
map[1] = 1;

thrust::device_vector<int> source(6);
source[0] = 10;
source[1] = 20;
...

int sum = thrust::reduce(thrust::make_permutation_iterator(source.begin(),
    map.begin()),
    thrust::make_permutation_iterator(source.begin(),
    map.end()));
```
Thrust zip iterator and arbitrary transforms

```cpp
struct arbitrary_functor {
    template <typename Tuple>
    __host__ __device__ void operator()(Tuple t) {
        // D[i] = A[i] + B[i] * C[i];
        thrust::get<3>(t) = thrust::get<0>(t) +
            thrust::get<1>(t) * thrust::get<2>(t);
    }
};

int main() {
    // allocate storage
    thrust::device_vector<float> A(5), B(5), C(5), D(5);
    // initialize input vectors
    A[0] = 3; B[0] = 6; C[0] = 2;
    ...
    // apply the transformation
    thrust::for_each(thrust::make_zip_iterator(
        thrust::make_tuple(A.begin(), B.begin(), C.begin(), D.begin())),
        thrust::make_zip_iterator(
            thrust::make_tuple(A.end(), B.end(), C.end(), D.end())),
        arbitrary_functor());
    // print the output
    for(int i = 0; i < 5; i++)
        std::cout << A[i] << " + " << B[i] << " * " << C[i] << " = " << D[i] << std::endl;
}
```
Thrust examples

Example 24:

Download the thrust library with examples using

git clone https://github.com/thrust/thrust.git. In the example directory you have many interesting sample programs. In the remaining time in the course room, read and run a few samples. They are well documented, but you can ask for any necessary explanations.