High-performance Scientific Computing in C++

27–28 June 2016  |  Sandipan Mohanty (s.mohanty@fz-juelich.de)
Jülich Supercomputing Centre

Day 1
Introduction

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

---

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++

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
C++

- Modern C++ provides many high level abstractions to facilitate fast development
- An awareness of low level features of the language, as well as the hardware environment, is nevertheless useful in writing code for HPC

Revisiting aspects of C++
Functions

```cpp
return_type function_name(parameters)
{
    // function body
}
double sin(double x)
{
    // Somehow calculate sin of x
    return answer;
}
int main()
{
    constexpr const 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!
}
```

- All executable code is in 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

Functions at run time

```
double sin(double x)
{
    // Somehow calculate sin of x
    return answer;
}
int main()
{
    constexpr const 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!
}
```

- When a function is called, it is given a "workbook" in memory called a stack frame
- Arguments are copied to registers or the stack as well as a return address
- (Non-static) local variables are allocated in the stack
- When the function concludes, execution continues at the return address, and the stack frame is destroyed
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

```cpp
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 }
```

Functions: under the microscope!

- 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

```cpp
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
```

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

- 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

```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;
};
```

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

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

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);
}
```
Global storage

```cpp
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
- Variables in functions marked with the `static` keyword
- 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`
- Can store very large objects which don’t fit in the stack
- In C++14, the only standard way to create arrays whose size is not known at compile time. C99 style variable length arrays are not standard C++.
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 no longer 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

- STL containers 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.

```
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

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)

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>>`
Dynamically sized 2D arrays

```cpp
template <typename T> class array2d {
    vector<T> v;
    size_t nc{0},nr{0};
public:
    T & operator()(size_t i, size_t j) {
        return v[i*nc+j];
    }
    // and a const version of the above
    template <typename U> struct row_t {
        vector<U> & orig;
        size_t offst, strd;
        explicit row_t(vector<U> &vv, size_t i, size_t st) :
            orig(vv), offst(i), strd(st) {}
        U & operator[](size_t i) {
            return orig[offst+i*strd];
        }
    };
    row_t<T> operator[](size_t i) {
        return row_t(v,i*nc,1);
    }
};
```

- 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 1.1:
examples/array2d contains the template class shown here.

---

STD::ARRAY

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

int main()
{
    std::array<double,10> A={0.0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9});
    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 contain a data element containing its size. But can always tell its size, as the size is "part of the name" of its type!
- Moving an `std::array` has order N complexity, as each individual element needs to be moved. No pointer swapping trick can do the job for this.
Data alignment

- Data is read or written with a unit size called word. On the most common architectures, word size is 4 or 8 bytes.
- Data alignment means, putting data on memory addresses which are integral multiples of the word size
- $n$-byte aligned data 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

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, an 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 1.2:

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

---

Data structure padding

```
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
```

- 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
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 1.3:

The `examples/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
- 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`
- Convenient alias `aligned_storage_t` available in C++14
- Used with "placement new" operator and explicit destructor calls to create/destruct objects of different types
### The placement new operator

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

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

- If buffer is obtained using a different mechanism, and we want to use those bytes as an object of a certain class, we must run a constructor on a specified location in memory. This is the role of the "placement new" operator.

- 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.

```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.
```

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

---

**Example 1.4:**

`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 `in.cppreference.com`, which I have modified slightly and commented with additional explanations.
Example 1.5:

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 in.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.

Branching
pipeline

- Program execution flows through different units responsible for different work

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

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

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

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

---

Branch prediction

```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

```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;
  }
}
```

- If statements, switches, loops contain obvious branches
- The ternary operator
  
  ```c
  a = cond ? v1 : v2
  ```
  is (not always!) a branch

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

Not branches

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

- 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

```c
double f(double x, double A[4])
{
  double a = x;
  for (int i=0;i<4;++i) a -= A[i];
  return a;
}
```
Example 1.6:

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.
- 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.

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

Class inheritance with virtual functions

- 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
- There is a table of virtual functions for each derived class, with entries pointing to function code somewhere
- The vptr pointer points to the vtable of that particular class

Example 1.7:
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++ -00 -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.

- 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.

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.
- 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.

```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();
    }
};
```

```cpp
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
Example 1.8: CRTP

The file examples/crtpl.cc demonstrates the use of CRTP to implement a form of polymorphic behaviour. The function `version()` is inherited without changes in Acetyl and Car. `get_name()` is inherited, but behaves differently for the two types. All this is without using virtual functions, i.e., in the call to the polymorphic function, the actual version of the function is determined at compile time.

SFINAE
// 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 `true` set to the second template parameter
- Using the `true` 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
Example 1.9:

examples/sfinae0.cc and examples/sfinae1.cc demonstrate SFINAE. The first is a simple syntax illustration. The second is an interesting application where we determine whether a template argument passed to a template function is a type which has a member function called size(). Try them out and modify to detect another member function!

Example 1.10:

examples/enableif0.cc shows one use of std::enable_if. The parameter list of the two variants of the template function \texttt{f} are identical, and they are "templates", where the place holder typename \texttt{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.
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;
}
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 $Z$
- 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

```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(); }
};
```

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:

```
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

```
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);
}
```

- 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

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

\begin{verbatim}
vec<
    vecsum<
        vecsum<
            vecscl<vec<double>>,
            vecscl<vec<double>>
        >,
    vec<double>
    > ({{a,X},{b,Y}},Z);
// Let’s call this type EXPR
\end{verbatim}

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;
}

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

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

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

- 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
Example 1.11:

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 1.1:

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.
Day 2

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 $m1 \times m2$
- **Expressions**: $3 \times m \times m \times v1 + u \times v2 + m \times m \times m$
- **Column major matrix**:
  - `Matrix<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;
  ```
Example 2.1:

There are a few example programs using Eigen in the folder examples/Eigen. Read the programs eigen0.cc and eigen1.cc. To compile, first load the Eigen module (module load Eigen/3.3-beta) and then use G $EIGEN$ program.cc. To use clang++ or icpc replace G with A or I to use the pre-defined aliases. For reference, A=’clang++ -stdlib=libc++ --std=c++14 -pedantic -Wall -O3 -march=native’ I=’icpc -std=c++14 -fast -march=native’ and $EIGEN"=-I /path/to/Eigen/headers"

Exercise 2.1:

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 (or -qopenmp for the Intel compiler).
Exercise 2.2:

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 \( X_i = [x_{i1}, x_{i2}, \ldots x_{im}] \) 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;
```

- No AVX until quite recently. We are using a beta version for that reason.
- "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.
Exercise 2.3:

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. On JURECA, using the Eigen module used in the course, you can do the following:

```
I $EIGEN_MKL matmul.cc. Run it on one node of JURECA and compare with the previous exercise. For reference,
```

```
EIGEN_MKL="-DEIGEN_USE_MKL_ALL -DNDEBUG -openmp -I/path/to/mkl/headers
-Wl,--start-group /path/to/libmkl_intel_lp64.a /path/to/libmkl_core.a
/path/to/libmkl_intel_thread.a -Wl,--end-group -lpthread -lm -ldl -mkl"
```
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 Plus
- Supports complex models such as pipelines, data flow and unstructured task graphs

TBB: Threading Building Blocks II

- Scalable memory allocation, avoidance of false sharing, thread local storage
- Low level synchronisation tools like mutexes and atomics
- Work stealing task scheduler
- `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
- Load the TBB module in your environment with `module load tbb` to compile the examples

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

parallel invoke

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

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

- Run an arbitrary number of function 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
- Takes a callable object as the third argument
- Using lambda functions, you can expose parallelism in sections of your code

```cpp
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);

// Code that can run in parallel
```

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
// 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

```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

```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 2.2: 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:

```
module load tbb
G $TBB_INCLUDES $TBB_LIBRARIES dasxpcy.cc -o dasxpcy_gcc
```

The TBB_* environment variables are defined by our TBB modules.

2D ranges

```c++
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
      &[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 2.3: TBB parallel reduce

The program `tbbreduce.cc` rewrites the program used to calculate the integral using async, now 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
- 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`
Enumerable thread specific

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 2.4: 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 `new` calls from different threads
- Global memory lock

**TBB allocators**

- Interface like `std::allocator`, so that it can be used with STL containers. E.g.,
  
  ```cpp
class std::vector<T, tbb::cache_aligned_allocator<T>>;
```

- `tbb::scalable_allocator<T>`: general purpose scalable allocator type, for rapid allocation from multiple threads
- `tbb::cache_aligned_allocator<T>`: 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);
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 2.4: 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

- Template library like STL or TBB for CUDA, with great documentation. The examples below are from thrust documentation
- Provides an elegant high level syntax to clearly express the intent of the programmer
- The language mechanisms of C++ leave enough room for the stated intents to be translated to efficient code by the compiler

Example:

```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());
}
```

- 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
### Host and device vectors

```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;
}
```

- **Containers** `host_vector` and `device_vector` are designed similar to `std::vector`, but as of CUDA 7.5, 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

---

### 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 the 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 2.5:

The example programs examples/thrust/demo0.cc and examples/thrust/demo1.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 demo0.cu`
- Try changing the file name to `demo0.cc`
- There is an alias for `salloc` called `getgpunode` defined in your `.bashrc` file containing the options about our reservation: `getgpunode --time=00:05:00`
- Run your code like any other executable

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"));
```
Custom functionals for transforms

To do data transformations which can not be represented by simple pre-defined operations in thrust/functional.h, we have to write our own function objects.

The overloaded operator() must be marked with __host__ __device__

```
struct saxpy_functor {
    const float a;
    saxpy_functor(float _a) : a(_a) {}  
    __host__ __device__
    float operator()(const float& x, const float& y) const {
        return a * x + y;
    }
};
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(), saxpy_functor(A));
}
```

Reductions

Reductions require a binary operation and some initial value.

Convenience variants with names such as 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".

```
int sum=thrust::reduce(D.begin(),D.end(),(int)0,thrust::plus<int>());
int sum=thrust::reduce(D.begin(),D.end(), (int)0);
int result = thrust::count(vec.begin(), vec.end(), 1);
// thrust::count_if
// thrust::inner_product
template <typename T>
struct square {
    __host__ __device__
    T operator()(const T& x) const {
        return x * x;
    }
};
float v=
    thrust::transform_reduce(d_x.begin(), d_x.end(),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, 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

```c
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 2.6:

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.