High performance scientific computing in C++

HPC C++ Course 2023

29 May – 02 June 2023 | Sandipan Mohanty | Forschungszentrum Jülich, Germany

What happens on line 9 of this code?

```
1  #include <iostream>
2  #include <Eigen/Dense>
3  using namespace Eigen;
4  using namespace std;
5  auto main() -> int
6  {
7      MatrixXd m = MatrixXd::Random(3,3);
8      auto result = (m + MatrixXd::Constant(3, 3, 1.2)) * 50;
9      m = result;
10     cout << "m =" << "\n" << m << "\n";
11
12 }
```

- (A) The result matrix calculated in the previous line is copied to `m`
- (B) The calculation specified in the previous line happens due to the call to `operator=`
- (C) SEGFAULT

Assume `CountingIterator` to be an iterator which iterates over a conceptual sequence of integers. What do you expect to be printed from this code?

```
auto main() -> int
{
    auto tot = 0.;
    constexpr auto L = 100.;
    constexpr auto N = 1000000UL;
    std::for_each(std::execution::par,
                  CountingIterator(0UL),
                  CountingIterator(N),
                  [&](auto i) {
                      auto dx = 2. * L / N;
                      auto x = -L + dx * (i + 0.5);
                      tot += dx / (1 + x * x);
                  });
    std::cout << tot << "\n";
}
```

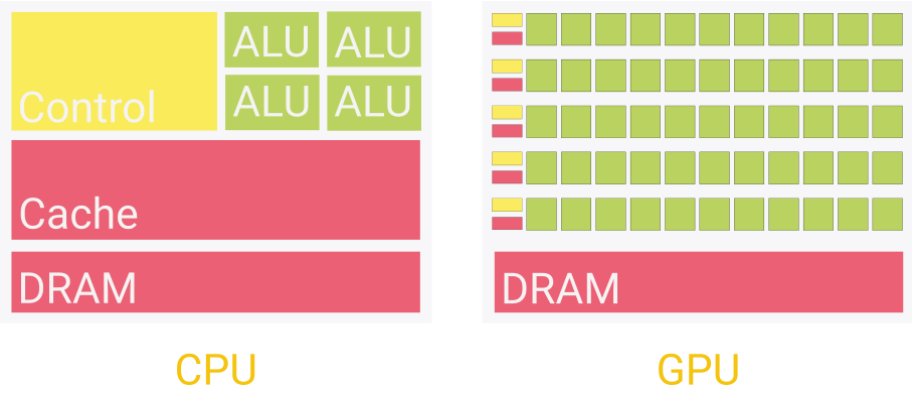
- (A) Crude approximation of the base of natural logarithm e
- (B) Crude approximation of π
- (C) Meaningless random value

GPU programming with CUDA

Data parallelism

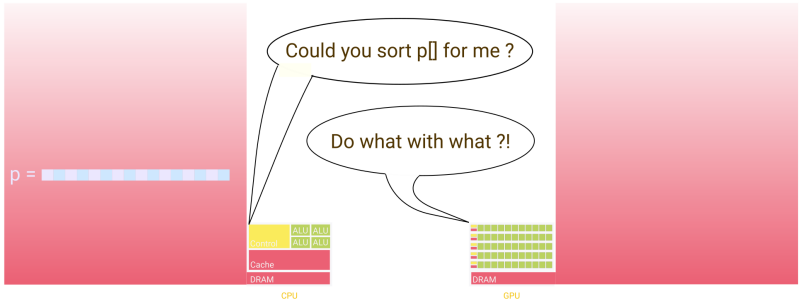
Bus/motorcycle analogy and figure stolen from GPU course/lecture slides by Andreas Herten (JSC)

Priorities



- CPU: faster clock speed, more cache, more sophisticated instructions and scheduling
- GPU: More chip area dedicated to floating point computations

A separate device



- Separate memories. GPU does not automatically know the state of any object in the memory of the CPU.
- Must transfer data.
- Must tell what to do with the data.
- Must retrieve results with another information transfer.

Can run C++ functions

- A program running on a CPU can call special functions designed to run on the GPU
- The GPU understands a different set of hardware instructions than the CPU, so any human readable function meant for the GPU must be compiled to a different kind of hardware instructions than code compiled for the CPU.
- A set of function "execution space specifiers" are provided as language extensions : `__global__`, `__device__` and `__host__`. These indicate to a CUDA aware compiler which parts to translate to the CPU language and which parts to the GPU language.
- A function running on the GPU can call other functions compiled for the GPU, leading to a call tree on the device side.

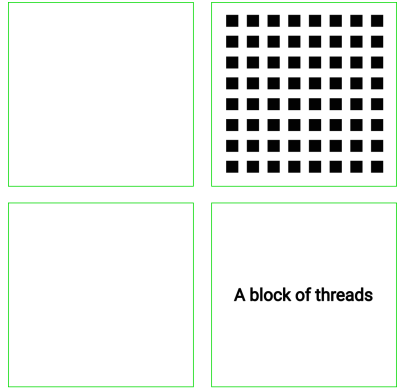
```
1  __device__ auto shuf(int id)
2  {
3      return (id + 1723) % 2000;
4  }
5  __global__
6  void gpufunc(int *ids, unsigned N)
7  {
8      // ...
9      ids[i] = shuf(ids[i]);
10     // ...
11 }
12 auto cpufunc() -> int
13 {
14     gpufunc<<<1, 100>>>(p, 3000);
15 }
16
```

Execution space specifiers

- `__device__` : the function runs on the device, and it can only be called from the device
- `__host__` : the function runs on the host, and it can only be called form the host
- `__global__`: the function is a "kernel". It runs on the device, and can be called from the host, or from device (compute capability >= 3.2)
  - Must have `void` return type
  - Can not be a member function
  - It is asynchronous : the function returns before the device performs its work
  - Must be called along with an "execution configuration" e.g., `gpufunc<<<1, 100>>>(p, 3000)`
- `__device__` and `__host__` can both be used for a function, in which case, it is compiled for both the host and the device.

Kernel call syntax

- Kernel functions are called with the `<<<GridSpec, BlockSpec>>>` notation, i.e., potentially in a large number of threads, arranged in blocks
- `BlockSpec` denotes a 3 dimensional object, 3 integers, specifying the arrangement of threads in a thread block
- `GridSpec` denotes a 3 dimensional object, 3 integers, specifying how blocks are arranged in a grid
- Each thread running a kernel function has a built in variable, `threadIdx`, specifying the position of the thread in its block, and another variable `blockIdx` to identify the block in the grid, and `blockDim` = number of threads in a block
- Overall x index: `blockIdx.x * blockDim.x + threadIdx.x` etc.



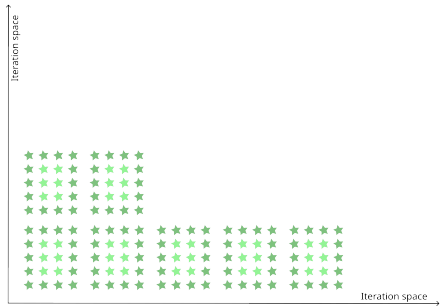
Kernel call syntax

```
1  __global__ void MatAdd(float A[N][N], float B[N][N],
2                         float C[N][N])
3  {
4      int i = blockIdx.x * blockDim.x + threadIdx.x;
5      int j = blockIdx.y * blockDim.y + threadIdx.y;
6      if (i < N && j < N)
7          C[i][j] = A[i][j] + B[i][j];
8  }
9
10 auto main() -> int
11 {
12     ...
13     // Kernel invocation
14     dim3 threadsPerBlock{16, 16};
15     dim3 numBlocks{N / threadsPerBlock.x,
16                    N / threadsPerBlock.y};
17     MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
18     ...
19 }
```

- The block and grid properties are often chosen to reflect properties of the problem being solved.
- In this example, the threads are organized in a 2D lattice: a natural fit for a matrix sum
- Each thread only needs to process one element!
- There is a maximum number of threads allowed in a block: a limit coming from hardware properties
- It is therefore necessary to arrange blocks into a grid

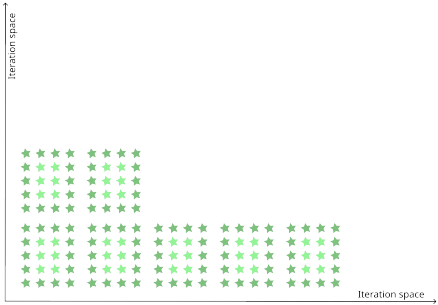
Kernel call syntax

- Remember how you had to process an array of double, 4 elements at a time and a stride of 4, when using AVX style SIMD register?



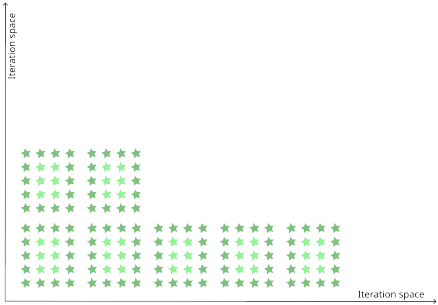
Kernel call syntax

- Remember how you had to process an array of double, 4 elements at a time and a stride of 4, when using AVX style SIMD register?
- Think of a block in CUDA as a potentially 3-dimensional stencil of tiny computations which you repeat to cover the iteration space



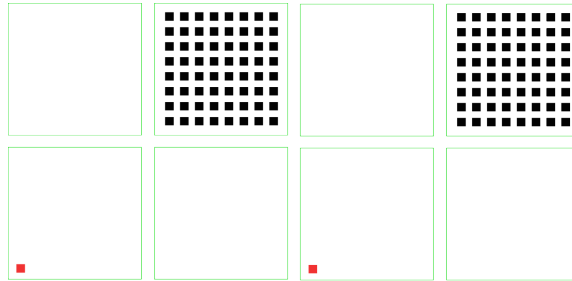
Kernel call syntax

- Remember how you had to process an array of double, 4 elements at a time and a stride of 4, when using AVX style SIMD register?
- Think of a block in CUDA as a potentially 3-dimensional stencil of tiny computations which you repeat to cover the iteration space
- We can use the grid to describe the iteration space in terms of the blocks. Depending on hardware availability, multiple blocks will run in parallel



Kernel call syntax

- Remember how you had to process an array of double, 4 elements at a time and a stride of 4, when using AVX style SIMD register?
- Think of a block in CUDA as a potentially 3-dimensional stencil of tiny computations which you repeat to cover the iteration space
- We can use the grid to describe the iteration space in terms of the blocks. Depending on hardware availability, multiple blocks will run in parallel



stride = blockDim.x * gridDim.x

- If the iteration space is much larger then the total number of GPU threads, it is sometimes helpful to do *grid stride loops* in the kernels. You have to take into account that `gridDim._ * blockDim._` GPU threads in the whole grid, which are processing lots of indexes together. That's how many indexes you would now jump over as a "stride"

Information transfer to and from the device

- Any data the kernel needs to process must be transferred using CUDA memory transfer functions
- Pointer/reference values received as input parameters in a function are interpreted on the same side of the host-device boundary

```
1  float *d_A, *d_B, *d_C;
2  auto size = N * sizeof(float);
3  cudaMalloc(&d_A, size);
4  cudaMalloc(&d_B, size);
5  cudaMalloc(&d_C, size);
6
7  cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
8  cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
```

```
1  __device__ float devData;
2  float value = 3.14f;
3  cudaMemcpyToSymbol(devData, &value, sizeof(float));
```

Information transfer to and from the device

- Any data the kernel needs to process must be transferred using CUDA memory transfer functions
- Pointer/reference values received as input parameters in a function are interpreted on the same side of the host-device boundary
- Allocations on unified memory are accessible from both the host and the device.
- Any data transfer required between the physically separate host and device memory happens automatically when using unified memory

```
1  float *u_A, *u_B, *u_C;
2  auto size = N * sizeof(float);
3  cudaMallocManaged(&u_A, size);
4  cudaMallocManaged(&u_B, size);
5  cudaMallocManaged(&u_C, size);
```

```
1  template <class T>
2  auto malloc_usm(size_t N,
3          std::optional<T> init = std::nullopt) -> T*
4  {
5      T* ans{};
6      cudaMallocManaged(&ans, N * sizeof(T));
7      if (init) {
8          for (size_t i = 0UL; i < N; ++i)
9              ans[i] = *init;
10     }
11     return ans;
12 }
```

Device side memory hierarchy and memory space specifiers

- "Local memory" -> per thread memory
- "Shared memory" -> private to a block, but shared among the threads inside a block
- "Global memory" -> visible from all threads in all blocks
- "Constant memory" -> also in the device space, and cached in the constant cache

- Memory address specifier `__device__` declares a variable which lives on the device
- `__constant__` declares a variable to be stored in constant cache
- `__shared__` : variable for the shared memory inside a block, and has the lifetime of the block
- `__managed__`: A variable declared with managed storage specifier can be accessed from both the host and the device, We can determine its address, and it can be read/written from both the host and the device. Since the host and device memories are physically separate, this behaviour is achieved by transferring memory implicitly

Example

```
1  __global__ void mul(const double *A, const double *B, double *C, size_t N) {
2      auto i = threadIdx.x + blockIdx.x * blockDim.x;
3      auto j = threadIdx.y + blockIdx.y * blockDim.y;
4      double res{};
5      if (i < N && j < N)
6          for (size_t k = 0ul; k < N; ++k)
7              res += A[N * i + k] * B[N * k +j];
8      C[N*i + j] = res;
9  }
10 auto main(int argc, char *argv[]) -> int {
11     const unsigned N = (argc > 1) ? std::stoul(argv[1]) : 2048u;
12     auto a = malloc_usm<double>(N * N);
13     auto b = malloc_usm<double>(N * N);
14     auto c = malloc_usm<double>(N * N);
15     for (size_t i = 0UL; i < N * N; ++i) { a[i] = b[i] = 1.1; }
16     auto t0 = std::chrono::high_resolution_clock::now();
17     dim3 ThreadsPerBlock{16, 16};
18     dim3 NumBlocks{N / ThreadsPerBlock.x, N / ThreadsPerBlock.y};
19     mul<<<NumBlocks, ThreadsPerBlock>>>(a, b, c, N);
20     cudaDeviceSynchronize();
```

This is simply a syntax demonstration! Not a particularly clever implementation!

Compiling CUDA code

- With `nvcc` :

```
nvcc [--expt-extended-lambda] [-std=__] source.cu
```

- With `clang++` :

```
clang++ [-std=__] source.cc --cuda-gpu-arch=____ -I /path/to/CUDA/include \
    -L /path/to/CUDA/lib64 -lcudart_static -ldl -lrt -lpthread
```

CUDA and C++

- Except in some ancient versions, CUDA is parsed by the rules of the C++ language. Many perfectly valid code in C, e.g., using `class`, `new`, `using` etc. as variable names can not be part of CUDA programs
- Valid C++ code, can often not be used, for a variety of reasons:
  - Generally, the NVIDIA implementation of newer language features arrives a few years after standardization
  - Some language features may have to be modified for use in the context of GPUs
  - CUDA 11 does support most of C++17. Our working environment is based on CUDA 11.7.
- Execution space specifiers, execution configuration etc. are language extensions
- This sometimes means additional rules are necessary before a new language feature can be used with CUDA. E.g., how do we make a lambda function `__device__` ? Should `__host__` etc. be considered parts of the functions signature or not ? `nvcc` and `clang++` disagree !

Thrust

NVIDIA Thrust

```
1  #include <thrust/host_vector.h>
2  #include <thrust/device_vector.h>
3  #include <thrust/generate.h>
4  #include <thrust/sort.h>
5  #include <thrust/copy.h>
6  #include <cstdlib>
7  using namespace thrust;
8  auto main() -> int
9  {
10     // generate 32 M random numbers on
11     // the host
12     host_vector<int> h_vec(32 << 20);
13     generate(h_vec.begin(), h_vec.end(), rand);
14
15     // transfer data to the device
16     device_vector<int> d_vec = h_vec;
17     sort(d_vec.begin(), d_vec.end());
18     // transfer data back to the host
19     copy(d_vec.begin(), d_vec.end(), h_vec.begin());
20 }
```

- Template library like STL or TBB for CUDA
- Elegant high level syntax (STL like iterator interface for algorithms, clever use of operator overloading ...) to clearly express the intent of the programmer
- The compiler translates the stated intents to efficient code for the GPU
- Primarily NVIDIA GPUs

NVIDIA Thrust

```
1  #include <thrust/host_vector.h>
2  #include <thrust/device_vector.h>
3  #include <thrust/generate.h>
4  #include <thrust/sort.h>
5  #include <thrust/copy.h>
6  #include <cstdlib>
7  using namespace thrust;
8  auto main() -> int
9  {
10     // generate 32 M random numbers on
11     // the host
12     host_vector<int> h_vec(32 << 20);
13     generate(h_vec.begin(), h_vec.end(), rand);
14
15     // transfer data to the device
16     device_vector<int> d_vec = h_vec;
17     sort(d_vec.begin(), d_vec.end());
18     // transfer data back to the host
19     copy(d_vec.begin(), d_vec.end(), h_vec.begin());
20 }
```

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

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;
auto main() -> int
{
    // 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(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

```
1  #include <thrust/host_vector.h>
2  #include <thrust/device_vector.h>
3  #include <thrust/generate.h>
4  #include <thrust/sort.h>
5  #include <thrust/copy.h>
6  #include <cstdlib>
7  using namespace thrust;
8  auto main() -> int
9  {
10     // generate 32 M random numbers on
11     // the host
12     host_vector<int> h_vec(32 << 20);
13     generate(h_vec.begin(), h_vec.end(), rand);
14
15     // transfer data to the device
16     device_vector<int> d_vec = h_vec;
17     sort(d_vec.begin(), d_vec.end());
18     // transfer data back to the host
19     copy(d_vec.begin(), d_vec.end(), h_vec.begin());
20 }
```

- 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`

Host and device vectors

```
1  #include <thrust/host_vector.h>
2  #include <thrust/device_vector.h>
3  #include <iostream>
4  auto main() -> int
5  {
6      thrust::host_vector<int> H(4);
7      for (int i = 0; i < 4; ++i) H[i] = i;
8      // resize H
9      H.resize(2);
10     std::cout << "H now has size "
11         << H.size() << "\n";
12     // Copy host_vector H to
13     // device_vector D
14     thrust::device_vector<int> D = H;
15     // elements of D can be modified
16     D[0] = 99;
17     D[1] = 88;
18     // print contents of D
19     for(int i = 0; i < D.size(); ++i)
20         std::cout << "D[" << i << "] = "
21                   << D[i] << "\n";
22 }
```

- Containers `host_vector` and `device_vector` are designed similar to `std::vector`, but, do not have initializer list constructors or new member functions of `std::vector` like `emplace_back`

Host and device vectors

```
1  #include <thrust/host_vector.h>
2  #include <thrust/device_vector.h>
3  #include <iostream>
4  auto main() -> int
5  {
6      thrust::host_vector<int> H(4);
7      for (int i = 0; i < 4; ++i) H[i] = i;
8      // resize H
9      H.resize(2);
10     std::cout << "H now has size "
11         << H.size() << "\n";
12     // Copy host_vector H to
13     // device_vector D
14     thrust::device_vector<int> D = H;
15     // elements of D can be modified
16     D[0] = 99;
17     D[1] = 88;
18     // print contents of D
19     for(int i = 0; i < D.size(); ++i)
20         std::cout << "D[" << i << "] = "
21                   << D[i] << "\n";
22 }
```

- Containers `host_vector` and `device_vector` are designed similar to `std::vector`, but, 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

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

Exercise 4.1:

The example programs `examples/thrust0.cu` and `examples/thrust1.cu` contain the thrust code in the previous slides. Run them on JUSUF using the following steps:

- Check that the NVidia CUDA modules are loaded: `ml`
- Compile using the `nvcc` compiler: `nvcc thrust0.cu`
- Try changing the file name to `thrust0.cc` and compiling

Thrust algorithms

```
1   device_vector<int> X(10),Y(10),Z(10);
2   // initialize X to 0,1,2,3, ....
3   sequence(X.begin(), X.end());
4   // compute Y = -X
5   thrust::transform(X.begin(), X.end(),
6      Y.begin(), thrust::negate<int>());
7   // fill Z with twos
8   thrust::fill(Z.begin(), Z.end(), 2);
9   // compute Y = X mod 2
10  thrust::transform(X.begin(), X.end(),
11                    Z.begin(), Y.begin(),
12                    thrust::modulus<int>());
13  // replace all the ones in Y with 10
14  thrust::replace(Y.begin(),Y.end(), 1, 10);
15  // print Y
16  thrust::copy(Y.begin(), Y.end(),
17  std::ostream_iterator<int>(cout,"\n"));
```

- Host and device versions

Thrust algorithms

```
1   device_vector<int> X(10),Y(10),Z(10);
2   // initialize X to 0,1,2,3, ....
3   sequence(X.begin(), X.end());
4   // compute Y = -X
5   thrust::transform(X.begin(), X.end(),
6      Y.begin(), thrust::negate<int>());
7   // fill Z with twos
8   thrust::fill(Z.begin(), Z.end(), 2);
9   // compute Y = X mod 2
10  thrust::transform(X.begin(), X.end(),
11                    Z.begin(), Y.begin(),
12                    thrust::modulus<int>());
13  // replace all the ones in Y with 10
14  thrust::replace(Y.begin(),Y.end(), 1, 10);
15  // print Y
16  thrust::copy(Y.begin(), Y.end(),
17  std::ostream_iterator<int>(cout,"\n"));
```

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

Thrust algorithms

```
1   device_vector<int> X(10),Y(10),Z(10);
2   // initialize X to 0,1,2,3, ....
3   sequence(X.begin(), X.end());
4   // compute Y = -X
5   thrust::transform(X.begin(), X.end(),
6      Y.begin(), thrust::negate<int>());
7   // fill Z with twos
8   thrust::fill(Z.begin(), Z.end(), 2);
9   // compute Y = X mod 2
10  thrust::transform(X.begin(), X.end(),
11                    Z.begin(), Y.begin(),
12                    thrust::modulus<int>());
13  // replace all the ones in Y with 10
14  thrust::replace(Y.begin(),Y.end(), 1, 10);
15  // print Y
16  thrust::copy(Y.begin(), Y.end(),
17  std::ostream_iterator<int>(cout,"\n"));
```

- 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!

Universal vectors

```cpp
// examples/thrust_usm.cc
#include <thrust/universal_vector.h>
#include <thrust/sort.h>
#include <iostream>
auto main() -> int
{
    thrust::universal_vector<int> h_vec(1 << 22);
    std::cout << "Filling host vector with random numbers\n";
    thrust::generate(thrust::host, h_vec.begin(), h_vec.end(), rand);
    std::cout << "Done.\n";

    std::cout << "Sorting vector on device\n";
    thrust::sort(thrust::device, h_vec.begin(), h_vec.end());
    std::cout << "Done.\n";
}
```

- `thrust::universal_vector` is similar to `thrust::host_vector` and `thrust::device_vector`, but uses unified memory for storage

Universal vectors

```cpp
// examples/thrust_usm.cc
#include <thrust/universal_vector.h>
#include <thrust/sort.h>
#include <iostream>
auto main() -> int
{
    thrust::universal_vector<int> h_vec(1 << 22);
    std::cout << "Filling host vector with random numbers\n";
    thrust::generate(thrust::host, h_vec.begin(), h_vec.end(), rand);
    std::cout << "Done.\n";

    std::cout << "Sorting vector on device\n";
    thrust::sort(thrust::device, h_vec.begin(), h_vec.end());
    std::cout << "Done.\n";
}
```

- `thrust::universal_vector` is similar to `thrust::host_vector` and `thrust::device_vector`, but uses unified memory for storage
- Data does not need to be moved explicitly between host and device

Universal vectors

```cpp
// examples/thrust_usm.cc
#include <thrust/universal_vector.h>
#include <thrust/sort.h>
#include <iostream>
auto main() -> int
{
    thrust::universal_vector<int> h_vec(1 << 22);
    std::cout << "Filling host vector with random numbers\n";
    thrust::generate(thrust::host, h_vec.begin(), h_vec.end(), rand);
    std::cout << "Done.\n";

    std::cout << "Sorting vector on device\n";
    thrust::sort(thrust::device, h_vec.begin(), h_vec.end());
    std::cout << "Done.\n";
}
```

- `thrust::universal_vector` is similar to `thrust::host_vector` and `thrust::device_vector`, but uses unified memory for storage
- Data does not need to be moved explicitly between host and device
- Algorithms need to be told whether they are meant for host or device explicitly

Custom functionals for transforms

```
1  struct saxpy_functor {
2    const float a;
3    saxpy_functor(float _a) : a(_a) {}
4    __host__ __device__
5      auto operator()(const float& x,
6                const float& y) const -> float {
7        return a * x + y;
8      }
9  };
10 void saxpy_fast(float A,
11       thrust::device_vector<float>& X,
12       thrust::device_vector<float>& Y)
13 {
14   // Y <- A * X + Y
15   thrust::transform(X.begin(), X.end(),
16                     Y.begin(), Y.begin(),
17                     saxpy_functor(A));
18 }
```

- 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 placehoders

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

```
1  void saxpy_fast(float A,
2        thrust::device_vector<float>& X,
3        thrust::device_vector<float>& Y)
4  {
5      // Y <- A * X + Y
6      thrust::transform(X.begin(), X.end(),
7                        Y.begin(), Y.begin(),
8                        (A*_1 +_2));
9  }
```

- `_1`, `_2` ... are placehoders
- Expressions involving placeholders yield a functional mapping its arguments sequencially to `_1`, `_2` ...

Custom functionals using lambda functions

```
1  void saxpy_fast(float A,
2        thrust::device_vector<float>& X,
3        thrust::device_vector<float>& Y)
4  {
5      // Y <- A * X + Y
6      thrust::transform(X.begin(), X.end(), Y.begin(), Y.begin(),
7                        [A] __host__ __device__ (double x, double y) {
8          return A * x + y;
9      });
10 }
```

```
nvcc --extended-lambda  saxpy0.cu
```

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

Exercise 4.2: Placeholders and lambda functions

The example `examples/saxpy0.cu` shows how to use the placehoders 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 4.3: 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_cpu.cc` does it, using the standard C++ library. A modified version using `thrust`, `mandelbrot_gpu.cu` is also present. Build `nvcc` and `clang++` and run. Figure out the code differences and why they are needed.

Reductions

```
1  int sum=thrust::reduce(D.begin(),D.end(), 0,
2                         thrust::plus<int>());
3  int sum=thrust::reduce(D.begin(),D.end(), 0);
4  int sum=thrust::reduce(D.begin(),D.end());
5  int result = thrust::count(vec.begin(),
6                             vec.end(), 1);
7  // thrust::count_if
8  // thrust::inner_product
9  float v= thrust::transform_reduce(d_x.begin(),
10    d_x.end(),unary_op,init,binary_op) );
```

- 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"

Partial sums, sorting, etc.

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

- Frequently needed algorithms, which are not trivial to parallelize, have thrust implementations

Partial sums, sorting, etc.

```
1  int data[6] = {1, 0, 2, 2, 1, 3};
2  inclusive_scan(data,data+6,data);
3  exclusive_scan(data,data+6,data);
4  // data is now {0, 1, 1, 3, 5, 6}
5  thrust::sort(A, A + N);
6  const int N = 6;
7  int    keys[N]={1, 4, 2, 8, 5, 7};
8  char values[N]={'a', 'b', 'c', 'd', 'e', 'f'};
9  thrust::sort_by_key(keys, keys + N, values);
10 // keys is now   {1, 2, 4, 5, 7, 8}
11 // values is now {'a', 'c', 'b', 'e', 'f', 'd'}
12 thrust::stable_sort(A, A+N,
13           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.

```
1  int data[6] = {1, 0, 2, 2, 1, 3};
2  inclusive_scan(data,data+6,data);
3  exclusive_scan(data,data+6,data);
4  // data is now {0, 1, 1, 3, 5, 6}
5  thrust::sort(A, A + N);
6  const int N = 6;
7  int    keys[N]={1, 4, 2, 8, 5, 7};
8  char values[N]={'a', 'b', 'c', 'd', 'e', 'f'};
9  thrust::sort_by_key(keys, keys + N, values);
10 // keys is now   {1, 2, 4, 5, 7, 8}
11 // values is now {'a', 'c', 'b', 'e', 'f', 'd'}
12 thrust::stable_sort(A, A+N,
13           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

```
1  thrust::constant_iterator<int> first(10);
2  first[0]   // returns 10
3  first[100] // returns 10
4  thrust::counting_iterator<int> first(10);
5  first[0]   // returns 10
6  first[1]   // returns 11
7  first[100] // returns 110
8  first = thrust::make_transform_iterator(vec.begin(), negate<int>());
9  ...
10 last  = thrust::make_transform_iterator(vec.end(),   negate<int>());
11 thrust::reduce(first, last);   // returns -60 (i.e. -10 + -20 + -30)
12
13 thrust::device_vector<int> map(2);
14 map[0] = 3;
15 map[1] = 1;
16 thrust::device_vector<int> source(6);
17 source[0] = 10;
18 source[1] = 20;
19 ...
20 int sum = thrust::reduce(thrust::make_permutation_iterator(source.begin(), map.begin()),
21                           thrust::make_permutation_iterator(source.begin(), map.end()));
```

Thrust zip iterator and arbitrary transforms

```
1  struct arbitrary_functor {
2      template <class Tuple>  __host__ __device__ void operator()(Tuple t) {
3          // D[i] = A[i] + B[i] * C[i];
4          thrust::get<3>(t) = thrust::get<0>(t) + thrust::get<1>(t) * thrust::get<2>(t);
5      }
6  };
7  auto main() -> int {
8      using namespace thrust;
9      device_vector<float> A(5), B(5), C(5), D(5);
10     // initialize input vectors
11     A[0] = 3;  B[0] = 6;  C[0] = 2;
12     A[1] = 4;  B[1] = 7;  C[1] = 5;
13     ...
14     // apply the transformation
15     for_each(make_zip_iterator(make_tuple(A.begin(), B.begin(), C.begin(), D.begin())),
16              make_zip_iterator(make_tuple(A.end(), B.end(), C.end(), D.end())),
17              arbitrary_functor());
18     for(int i = 0; i < 5; ++i)
19         std::cout << A[i] << " + " << B[i] << " * " << C[i] << " = " << D[i] << "\n";
20 }
```

Thrust examples

Exercise 4.4:

`examples/thrust/matmuls.cu` and `examples/thrust/matmuld.cu` demonstrate using CUDA blas library, CUDA random number generator and thrust together. Using thrust for the memory management parts significantly improves the readability of such programs. Learn the steps necessary to compile them, and check performance on one of the GPU nodes.

STDPAR: standard C++ for GPUs

- NVC++, the NVIDIA HPC SDK C++ compiler
- No `<<< >>>`, no `__device__` etc. Just plain C++ written with STL algorithms
- `std::execution::par` regions automatically translated into GPU code!
- There are restrictions, but they will likely be fewer and fewer in the future

```
std::transform_reduce(std::execution::par, R2.begin(),
    R2.end(), S12.begin(), 0., std::plus<double>{},
    [](auto r2, auto s12){
        return Vexv(r2, s12);
    });
```

```
nvc++ -O3 -std=c++17 -stdpar exvol.cc -o exvol.nv
```

STDPAR: standard C++ for GPUs

- Only inline functions or function templates. `nvc++` selects functions for GPU execution on its own, and that only works if it can see the definitions
- CUDA Unified Memory for all data movement between CPU and GPU: presently, only heap allocated objects in CPU code compiled by `nvc++ -stdpar` can be automatically managed. Stack and global storage not accessible. Even heap allocations from portions of CPU code not compiled by `nvc++ -stdpar` are not visible.
- Pointers dereferenced in the parallel algorithms must point to heap locations. References used must be of heap objects.
- Lambda captures by references can often entail pointer dereferencing for stack entities, which should not occur in parallel algorithm regions
- No function pointers: functions are compiled for CPU and GPU. Pointer can only point to one. Inside GPU code, there will then be the danger of accessing a pointer to a function with CPU code. Pass function objects or lambdas as arguments to the algorithms instead.
- Only random access iterators
- `catch` clauses in GPU code ignored. Fine inside CPU code.

Exercise 4.5:

The programs `stdpardemo0.cc` and `stdpardemo1.cc` are simple short programs using parallel algorithms. The second one is a slighly modified version of the `exvol.cc` program we used in connection with SIMD programming. Compile them with `nvc++` and run them on a GPU node on JUSUF.

Exercise 4.6:

Two versions of a program `jacobi_cxx17.cc` and `jacobi_cxx20.cc` are in your examples folder. Identify the part which can be parallelized using STL parallel algorithms, and do the necessary code changs. The C++17 version can be compiled for the GPU using `nvc++`, without any code changes. Try this new way of CPU/GPU programming where the exact same code runs on both!

Unfinished business

Exercise 4.7:

The travelling salesman example from the first day is conceptually easily parallelisable problem. In `examples/TSP-pstl/` you will find a version which uses parallelisation with PSTL. The parallelised version allows us to search all possible paths for 15 cities in about 30 seconds on JUSUF CPU nodes. But we used far too much C++20 there for it to be compatible with current NVidia compilers. There is a different version `TSP-stdpar/`, in which I have removed all unsupported C++20 features. It no longer uses modules, and is in a single file. Although the `nvc++` compiler does not emit any errors, it can't generate code for the GPU using this version. Can you fix it?