

Атомарные операции. Поток.
Параллельное копирование и
выполнение ядра.
Интероперабельность с OpenGL
Библиотека thrust

Лекторы:

[Боресков А.В. \(ВМК МГУ\)](#)

План

- Атомарные операции
- Потоки, их использование
- Реализация одновременного копирования данных и выполнения ядра
- Интероперабельность с OpenGL
- thrust

Атомарность операции

Рассмотрим традиционную операцию инкремента

`x++`

На практике она переводится в следующие операции

`r = x; load into register`

`inc r; increment value in register`

`x = r; store incremented value`

Атомарность операции

Теперь рассмотрим ситуацию когда две нити пытаются одновременно выполнить операцию инкремента над одной и той же переменной

```
; Thread 1           ; Thread 2
r1 = x;              r2 = x;
inc r1;              inc r2;
x = r1;              x = r2;
```

Тут уже возможны варианты наложения операций друг на друга

Атомарность операции. Конфликт

```
; Thread 1
```

```
; x = 0
```

```
r1 = x; r1 = 0
```

```
inc r1; r1 = 1
```

```
x = r1; x = 1
```

```
; Thread 2
```

```
; x = 0
```

```
r2 = x; r2 = 0
```

```
inc r2; r2 = 1
```

```
x = r2; x = 1
```

Операция прошла некорректно - значение переменной было инкрементировано всего один раз

Атомарные операции

CUDA поддерживает специальные операции, гарантирующие атомарность

- Они выполняются медленнее
- CC 1.1 поддерживает целочисленные атомарные операции в глобальной памяти
- CC 1.1 поддерживает целочисленные атомарные операции в разделяемой памяти

Атомарные операции

- Над 64-битовыми целыми с CС 2.0
- `atomicAdd` для `float` - CС 2.0

Атомарные операции

```
// возвращают старое значение
int atomicAnd ( int * addr, int value );
uint atomicAnd ( uint * addr, uint value );
unsigned long long atomicAdd ( unsigned long long * addr, unsigned long long value );
float atomicAdd ( float * addr, float value );

int atomicSub ( int * address, int value );
uint atomicSub ( uint * address, uint value );

// записывает значение по адресу, возвращает старое значение
int atomicExch ( int * addr, int value );
uint atomicExch ( uint * addr, uint value );
unsigned long long atomicExch ( unsigned long long * addr, unsigned long long value );

// записывает результат операции, возвращает старое значение
int atomicMin ( int * addr, int value );
uint atomicMin ( uint * addr, uint value );
int atomicMax ( int * addr, int value );
uint atomicMax ( uint * addr, uint value );

uint atomicInc ( uint * addr, uint value );
uint atomicDec ( uint * addr, uint value );

int atomicAnd ( int * addr, int value );
uint atomicAnd ( uint * addr, uint value );
int atomicOr ( int * addr, int value );
uint atomicOr ( uint * addr, uint value );
int atomicXor ( int * addr, int value );
uint atomicXor ( uint * addr, uint value );
```


Потоки (Streams)

- GPU умеют выполнять многие вещи параллельно
 - Выполнение ядер и копирование памяти между CPU и GPU может выполняться параллельно
 - GPU с CC 2.x умеют выполнять до 16 ядер одновременно
- Для использования этой параллельности вводятся потоки
 - Поток (stream) в CUDA представляет собой очередь запросов, которые должны быть выполнены в заданном порядке
 - Внутри потока все выполняется последовательно
 - По умолчанию используется всегда существующий поток 0
 - Если вы не задали явно поток - это поток 0
 - Однако можно создать несколько потоков, тогда операции из разных потоков могут выполняться параллельно
 - Но последовательно внутри каждого потока

Потоки

- Поток (stream) в CUDA представляет собой очередь запросов, которые должны быть выполнены в заданном порядке
- По умолчанию используется всегда существующий поток 0
- Однако можно создать несколько потоков, тогда операции из разных потоков могут выполняться параллельно

Потоки

- Создание и уничтожение потоков
 - Поток имеет тип `cudaStream_t`
 - Поток должен явно создаваться и уничтожаться
 - Кроме потока 0 - он всегда существует и его не надо уничтожать

```
cudaStream_t stream;
```

```
cudaStreamCreate ( &stream );
```

```
cudaStreamDestroy ( &stream );
```

Потоки (пример)

- Рассмотрим следующую задачу - есть операция, берущая на вход два массива и по каждой паре элементов из соответствующих массивов строящая элемент третьего (выходного) массива
 - $c[i] = \text{foo}(a[i], b[i])$
- Традиционный способ
 - Сперва целиком копируем оба массива CPU->GPU
 - Выполняем ядро
 - Целиком копируем выходной массив GPU->CPU
 - В каждый момент времени мы выполняем что-то одно
 - Все остальное простаивает

Потоки (пример)

- Для того, чтобы воспользоваться возможностью параллельного копирования и выполнения
 - Выделим pinned-память
 - Разобьем массивы на блоки
 - Одновременно будем копировать два входных блока, выполнять ядро и копировать результат обратно
 - Должны *вроде-бы* получить перекрывание по времени для всех операций
 - Выполнения ядра
 - Копирование на GPU
 - Копирования с GPU

ПОТОКИ

- Как мы ХОТИМ получить
 - По очереди для каждого потока
 - Copy A
 - Copy B
 - Kernel
 - Copy C

Stream 0

memcpy A to GPU
memcpy B to GPU
kernel
memcpy C to GPU
memcpy A to GPU
memcpy B to GPU
kernel
memcpy C to GPU

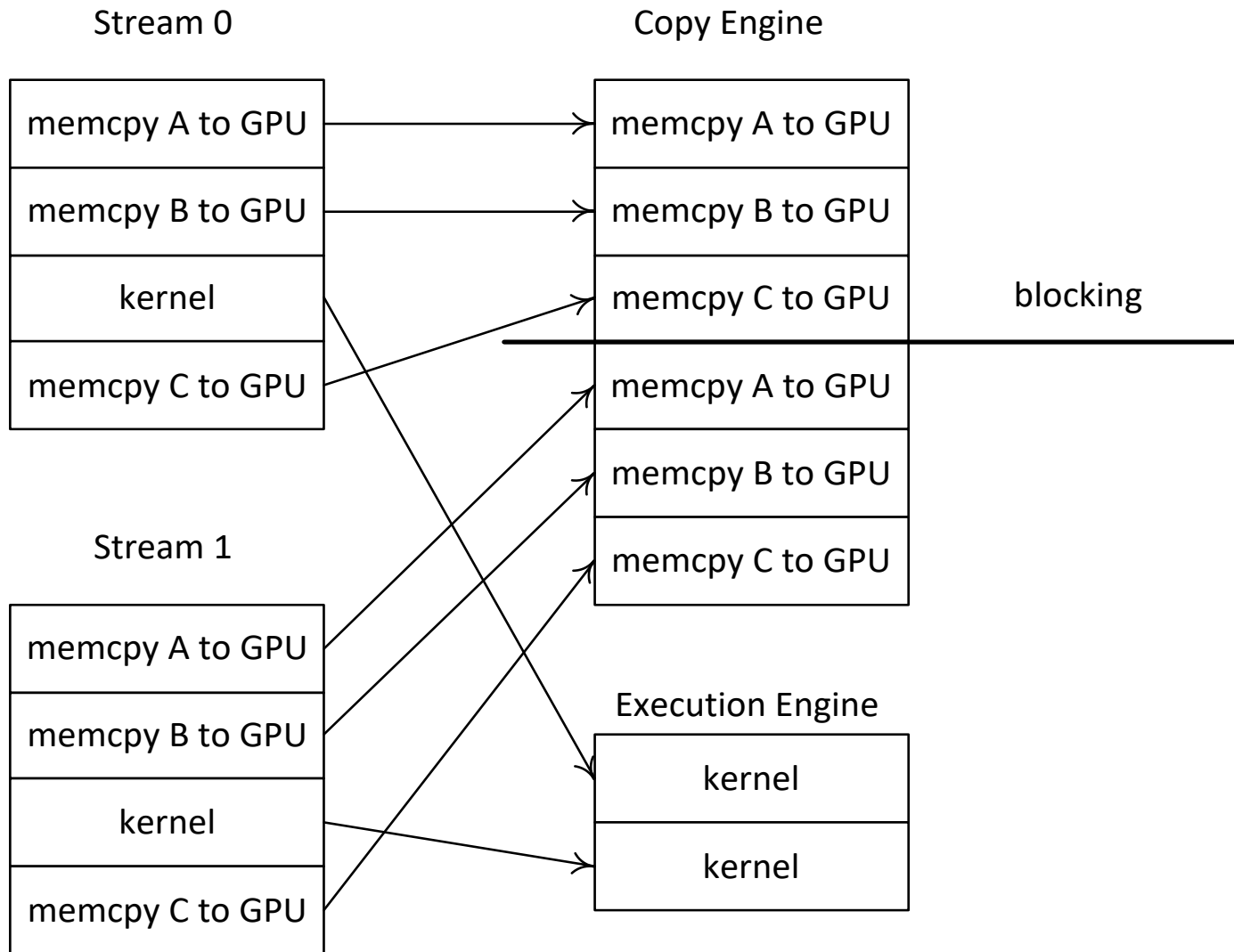
Stream 1

memcpy A to GPU
memcpy B to GPU
kernel
memcpy C to GPU
memcpy A to GPU
memcpy B to GPU
kernel
memcpy C to GPU

Потоки, как все на самом деле

- На самом деле на GPU есть независимые подсистемы
 - Подсистема расчета (выполнения ядер)
 - Подсистема копирования
 - Про потоки эти подсистемы вообще ничего не знают
- Таким образом мы в каждом потоке ставим в очередь в подсистему копирования
 - Сору А
 - Сору В
 - Сору С
- В каждом потоке ставим в подсистему расчета
 - Kernel
- Для подсистем “Сору С” зависит от ядра соответствующего потока - так драйвер отслеживает зависимости в потоке

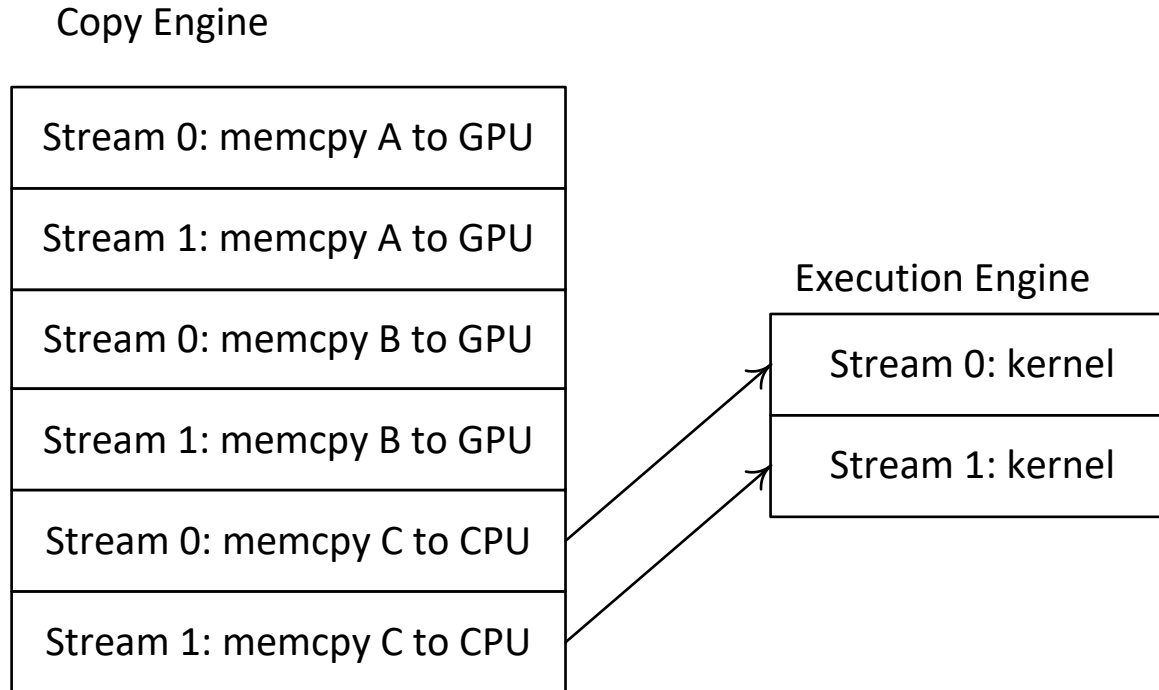
ПОТОКИ - как все работает



Потоки - как нужно

- Раньше вы ставили задачи по принципу «в глубину»
- Теперь будем ставить «в ширину»
 - Copy A (stream 0)
 - Copy A (stream 1)
 - Copy B (stream 0)
 - Copy B (stream 1)
 - Kernel (stream 0)
 - Kernel (stream 1)
 - Copy C (stream 0)
 - Copy C (stream 1)

Потоки - как нужно



Интероперабельность с OpenGL

- Возможность непосредственно в CUDA использовать данные OpenGL без необходимости их копировать
 - Поддерживаются текстуры
 - Поддерживаются VBO (Vertex Buffer Object)
 - Соответствующий ресурс необходимо зарегистрировать вначале
 - И разрегистрировать в конце
 - Для отображения ресурса в адресное пространство CUDA используется соответствующие функции отображения
 - При этом ресурс не может одновременно использоваться и CUDA и OpenGL - мы отображаем его в CUDA, работаем, потом закрываем отображение и OpenGL может снова его использовать
 - Довольно легко завернуть в классы C++
 - Изменения в работе с текстурами в последних версиях CUDA

Интероперабельность с OpenGL - VBO

- Регистрируем буфер в CUDA
 - Получаем указатель на `cudaGraphicsResource`
 - В конце работы нужно разрегистрировать
 - Можно задавать флаги через `cudaGraphicsResourceSetMapFlags`
 - `cudaGraphicsMapFlagsReadOnly`
 - `cudaGraphicsMapFlagsWriteDiscard`
- Когда нужны данные
 - Вызываем `cudaGraphicsMapResources`
 - Получаем указатель на глобальную память буфера через `cudaGraphicsResourceGetMappedPointer`
 - Используем указатель
 - Чтобы сделать буфер снова доступным OpenGL вызываем `cudaGraphicsUnmapResources`

Интероперабельность с OpenGL - VBO

```
class      CudaGlbuffer                                // VBO
{
    cudaGraphicsResource * resource;
    VertexBuffer          * buffer;
    GLenum                target;

public:
    CudaGlbuffer ( VertexBuffer * buf, GLenum theTarget,
                  unsigned int flags = cudaGraphicsMapFlagsWriteDiscard )
    {
        buffer = buf;
        target = theTarget;

        buffer -> bind ( target );
        cudaGraphicsGLRegisterBuffer ( &resource, buffer -> getId (), flags );
        buffer -> unbind ();
    }
    ~CudaGlbuffer ()
    {
        cudaGraphicsUnregisterResource ( resource );
    }
    bool mapResource ( cudaStream_t stream = 0 )
    {
        return cudaGraphicsMapResources ( 1, &resource, stream ) == cudaSuccess;
    }
    bool unmapResource ( cudaStream_t stream = 0 )
    {
        return cudaGraphicsUnmapResources ( 1, &resource, stream ) == cudaSuccess;
    }
}
```

Интероперабельность с OpenGL - VBO

```
void * mappedPointer ( size_t& numBytes ) const
{
    void * ptr;

    if ( cudaGraphicsResourceGetMappedPointer ( &ptr, &numBytes, resource )
        != cudaSuccess )
        return NULL;

    return ptr;
}

GLuint getId () const
{
    return buffer -> getId ();
}

GLenum getTarget () const
{
    return target;
}

cudaGraphicsResource * getResource () const
{
    return resource;
}
};
```

Интероперабельность с OpenGL - Текстуры

- Регистрируем текстуру в CUDA
 - Получаем указатель на `cudaGraphicsResource`
 - В конце работы нужно разрегистрировать
 - Можно задавать флаги через `cudaGraphicsResourceSetMapFlags`
 - `cudaGraphicsMapFlagsReadOnly`
 - `cudaGraphicsMapFlagsWriteDiscard`
- Когда нужны данные
 - Вызываем `cudaGraphicsMapResources`
 - Доступ к текстуре идет через
 - `cudaGraphicsResourceGetMappedMipmappedArray`
 - `cudaGraphicsSubResourceGetMappedArray (cudaArray)`
 - Используем указатель
 - Чтобы сделать буфер снова доступным OpenGL вызываем `cudaGraphicsUnmapResources`

Интероперабельность с OpenGL - Текстуры

- Либо получаем доступ ко всей текстуре со всеми уровнями пирамидального фильтрования
 - `cudaMipmappedArray_t`
 - Является полем `mipmap` структуры `cudaResourceDesc`
 - Можем создать `texture object` для доступа
- Либо получаем доступ к одному слою/уровню текстуры через `cudaArray` (сразу задаем их)
 - Можем создать `textureObject` для доступа

Интероперабельность с OpenGL - Текстуры

```
class    CudaGlImage
{
    GLuint          image;
    GLenum          target;
    cudaGraphicsResource * resource;

public:
    CudaGlImage ( GLuint theImage, GLenum theTarget,
        unsigned int flags = cudaGraphicsMapFlagsWriteDiscard )
    {
        image = theImage;
        target = theTarget;
        cudaGraphicsGLRegisterImage ( &resource, image, target, flags );
    }
    ~CudaGlImage ()
    {
        cudaGraphicsUnregisterResource ( resource );
    }

    bool    mapResource ( cudaStream_t stream = 0 )
    {
        return cudaGraphicsMapResources ( 1, &resource, stream ) == cudaSuccess;
    }

    bool    unmapResource ( cudaStream_t stream = 0 )
    {
        return cudaGraphicsUnmapResources ( 1, &resource, stream ) == cudaSuccess;
    }
}
```

Интероперабельность с OpenGL - Текстуры

```
cudaArray * mappedArray ( unsigned int index = 0, unsigned int mipLevel = 0 ) const
{
    cudaArray * array;

    if ( cudaGraphicsSubResourceGetMappedArray ( &array, resource, index,
                                                mipLevel ) != cudaSuccess )
        return NULL;

    return array;
}
```

Библиотека thrust

- Полностью на шаблонах
- Напоминает STL
- Свой namespace thrust
- Свои контейнеры, итераторы, алгоритмы (почти как STL)
- Поддерживает функторы и лямбды
 - Для лямбд нужна опция nvcc --expt-extended-lambda
- Многие операции над данными записываются в одну строку
- Все алгоритмы работают и для CPU и для GPU

Библиотека thrust - контейнеры

- Всего два типа контейнеров (самые эффективные с точки зрения доступа к памяти) - `thrust::host_vector`, `thrust_device_vector`
- Управляют памятью, можно динамически менять размер
- По интерфейсу напоминают STL
- Произвольный доступ к элементам (даже для `device_vector` на стороне CPU)
- Простое копирование (`c-tor`, `copy`)
- Типизированные
- Можно получить указатель на область глобальной памяти для `device_vector`

Библиотека thrust - контейнеры

```
thrust::host_vector<int>    hv ( 1000 );
thrust::device_vector<int> dv ( 1000 );
    // заполняем данными,
    // sequence - функция, функтор, лямбда
thrust::generate ( hv.begin (), hv.end (), sequence );
    // копирование
dv = hv;
    // операция над данными
int sum = thrust::reduce ( dv.begin (), dv.end () );
    // обращение на стороне CPU
int value = dv [10];
```

Библиотека thrust - контейнеры

- Можно получить указатель на данные контейнера

```
thrust::device_vector<int> v (1000);  
int * rawPtr = thrust::raw_pointer_cast(v.data());  
myKernel<<<threads, blocks>>> ( rawPtr, v.size () );
```

- Если есть готовый указатель в память GPU, то для получения нужного типа итератора следует обернуть его в класс device_ptr

```
int * rawPtr;  
cudaMalloc ( (void**)&rawPtr, N * sizeof ( int ) );  
thrust::device_ptr ptr ( rawPtr );  
// some operations on ptr  
cudaFree ( rawPtr );
```

Библиотека thrust - итераторы

- `begin()`, `end()`, `++`, `--`, арифметика указателей
- Тип итератора несет в себе информацию о том, в чьей памяти
 - Используется при компиляции для выбора алгоритма
- Обычный указатель - тоже итератор, но только в память CPU
 - Поэтому указатель на глобальную память нельзя напрямую передавать в thrust
 - Перевод device-указателя в итератор

```
thrust::device_ptr<int> ptr ( rawPtr );
```
- Есть специальные классы итераторов

Библиотека thrust

Простейшие операции, работают с памятью CPU и GPU

```
// заполнение заданным значением
thrust::fill ( d.begin (), d.end (), 77 );

// заполнение арифметической прогрессией
thrust::sequence ( d.begin (), d.end (), 7, -2 );

// использование функции для генерации
thrust::generate ( d.begin (), d.end (), rand );

// копирование
thrust::copy ( src.begin (), src.end (), dst.begin () );
```

Библиотека thrust - алгоритмы

- Принимают на вход функторы и итераторы
- Есть набор стандартных функторов
- Можно делать свои функторы как в STL
 - указываем `__device__` (`__host__`)
- Есть «хитрая замена» лямбд через `thrust::placeholders`
- Можно использовать лямбды с описателем `__device__`
 - Это расширение языка - нужен флаг компиляции `--expt-extended-lambda`

Библиотека thrust - алгоритмы

```
// преобразование одного массива в другой
thrust::transform ( a.begin (), a.end (), b.begin (),
                   output.begin (), thrust::multiplies<float> () );

// редукция с заданной операцией
int sum = thrust::reduce( a.begin (), a.end (), 0,
                          thrust::plus<int> () );

// задаем свой функтор
template<typename T> struct square
{
    __device__ __host__ T operator ()( const T& x ) const
    { return x*x; }
};

// используем функтор, reduce для преобразованных значений
float s = thrust::transform_reduce ( v.begin (),
                                     v.end (), square<float>(), 0, thrust::plus<float>() );
```

Библиотека thrust - алгоритмы

```
// первый вариант префиксной суммы
thrust::inclusive_scan ( x.begin (), x.end (), x.begin () );
    // второй вариант префиксной суммы
thrust::exclusive_scan ( a.begin (), a.end (), b.begin () );
    // сортировка (не устойчивая)
thrust::sort ( v.begin (), v.end () );
    // устойчивая - не меняет порядок равных элементов
thrust::stable_sort ( v.begin (), v.end () );
    // ключи сортировки отдельно от данных
thrust::sort_by_key ( keys.begin (), keys.end (),
                    data.begin (), compare );
```

Библиотека thrust - итераторы

- `thrust::constant_iterator<T>` - всегда возвращает одно значение
- `thrust::counting_iterator<T>` - арифметическая последовательность
- `thrust::transform_iterator<T>` - объединение итератора и преобразования
 - `thrust::make_transform_iterator (a.begin (), thrust::negate<int> ())`
- `thrust::zip_iterator` - объединение нескольких итераторов в один (возвращающий tuple)

Библиотека thrust

Аналог лямбд

```
#include <thrust/functional.h>
using namespace thrust::placeholders;
thrust::for_each(x.begin(), x.end(), _1++);

int a = 42;
thrust::transform(x.begin(), x.end(), y.begin(),
                 x.begin(), a * _1 + _2);

thrust::for_each(x.begin(), x.end(), _1++);
```

Библиотека thrust - лямбды

- Выставляем флаг компиляции
- Задаем `__device__` в описании
- Используем где хотим

```
// сортируем по критерию - сначала те элементы
// для которых true
thrust::partition ( dv2.begin (), dv2.end (),
    [] __device__ ( int x ) { return x % 2 == 0; } );
```

Библиотека thrust - еще алгоритмы

- `bool flag = thrust::all_of (a.begin (), a.end (), _1 < 0);`
- `bool flag = thrust::any_of (a.begin (), a.end (), _1 > 10);`
- `bool flag = thrust::none_of (a.begin (), a.end (), _1 < 0);`
- `thrust::copy_if (a.begin(), a.end (), result.begin (), _1 % 2 != 0);`
- `thrust::for_each`
- `int c = thrust::count_if (a.begin (), a.end (), _1 > 7);`

Библиотека thrust

Есть еще много различных алгоритмов

Есть много специальных итераторов,
например `transform_iterator` или
`zip_iterator`