



– библиотека параллельных алгоритмов для C++

Павел Якимов

[yakimov@ssau.ru](mailto:yakimov@ssau.ru)



## Thrust – это:

- STL-подобная библиотека обработки данных на GPU
  - Реализация как для GPU, так и для CPU
  - Открытый проект, поддерживаемый NVIDIA
- Разработчики: Nathan Bell и др.



## Основные возможности

- Унифицированный интерфейс для выполнения типичных задач обработки данных
- Уделяется внимание производительности
- По сравнению с CUDA C, возможности тонкого контроля (напр., разделяемая память) не так богаты
- Дизайн, схожий с STL: контейнеры, итераторы, алгоритмы



# Компоненты

## Контейнеры

- Управление памятью на host и device
- Упрощённый обмен данными

## Итераторы

- Подобны указателям
- «Следят» за областью памяти (host или device)

## Алгоритмы

- Применяются к контейнерам и итераторам



# Распространение

- Входит в состав CUDA 4+

- Хостинг кода: GitHub

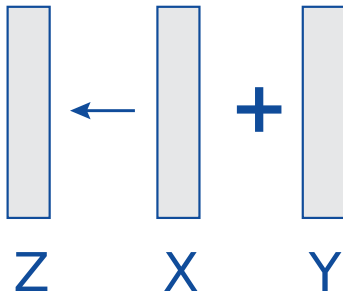
```
git clone https://github.com/thrust/thrust.git
```



## Пример №1: сложение векторов

Простое сложение векторов в Thrust:

```
for (int i = 0; i < N; ++i)  
    Z[i] = X[i] + Y[i];
```





## Пример №1: сложение векторов

```
#include <thrust/device_vector.h>
#include <thrust/transform.h>
#include <thrust/functional.h>
#include <iostream>

int main(void) {

    thrust::device_vector<float> X(3);
    thrust::device_vector<float> Y(3);
    thrust::device_vector<float> Z(3);

    X[0] = 10; X[1] = 20; X[2] = 30;
    Y[0] = 15; Y[1] = 35; Y[2] = 10;

    thrust::transform(X.begin(), X.end(),
        Y.begin(),
        Z.begin(),
        thrust::plus<float>());

    for (size_t i = 0; i < Z.size(); i++)
        std::cout << "Z[" << i << "] = " << Z[i] << "\n";

}

return 0;
```



## Пример №1: сложение векторов

```
#include <thrust/device_vector.h>
#include <thrust/transform.h>
#include <thrust/functional.h>
#include <iostream>
```

Заголовочные файлы Thrust

```
int main(void) {

    thrust::device_vector<float> X(3);
    thrust::device_vector<float> Y(3);
    thrust::device_vector<float> Z(3);

    X[0] = 10; X[1] = 20; X[2] = 30;
    Y[0] = 15; Y[1] = 35; Y[2] = 10;

    thrust::transform(X.begin(), X.end(),
        Y.begin(),
        Z.begin(),
        thrust::plus<float>());

    for (size_t i = 0; i < Z.size(); i++)
        std::cout << "Z[" << i << "] = " << Z[i] << "\n";

}

return 0;
```





## Пример №1: сложение векторов

```
#include <thrust/device_vector.h>
#include <thrust/transform.h>
#include <thrust/functional.h>
#include <iostream>
```

```
int main(void) {
```

```
    thrust::device_vector<float> X(3);
    thrust::device_vector<float> Y(3);
    thrust::device_vector<float> Z(3);
```

```
    X[0] = 10; X[1] = 20; X[2] = 30;
    Y[0] = 15; Y[1] = 35; Y[2] = 10;
```

```
    thrust::transform(X.begin(), X.end(),
        Y.begin(),
        Z.begin(),
        thrust::plus<float>());
```

```
    for (size_t i = 0; i < Z.size(); i++)
        std::cout << "Z[" << i << "] = " << Z[i] << "\n";
```

```
}
```

```
return 0;
```

3 вектора  
в памяти GPU



## Пример №1: сложение векторов

```
#include <thrust/device_vector.h>
#include <thrust/transform.h>
#include <thrust/functional.h>
#include <iostream>

int main(void) {

    thrust::device_vector<float> X(3);
    thrust::device_vector<float> Y(3);
    thrust::device_vector<float> Z(3);

    X[0] = 10; X[1] = 20; X[2] = 30;
    Y[0] = 15; Y[1] = 35; Y[2] = 10;

    thrust::transform(X.begin(), X.end(),
                     Y.begin(),
                     Z.begin(),
                     thrust::plus<float>());

    for (size_t i = 0; i < Z.size(); i++)
        std::cout << "Z[" << i << "] = " << Z[i] << "\n";

    return 0;
}
```

Поэлементные присваивания  
неэффективны, т.к. каждая  
операция влечёт копирование



## Пример №1: сложение векторов

```
#include <thrust/device_vector.h>
#include <thrust/transform.h>
#include <thrust/functional.h>
#include <iostream>

int main(void) {

    thrust::device_vector<float> X(3);
    thrust::device_vector<float> Y(3);
    thrust::device_vector<float> Z(3);

    X[0] = 10; X[1] = 20; X[2] = 30;
    Y[0] = 15; Y[1] = 35; Y[2] = 10;

    thrust::transform(X.begin(), X.end(),
                     Y.begin(),
                     Z.begin(),
                     thrust::plus<float>());

    for (size_t i = 0; i < Z.size(); i++)
        std::cout << "Z[" << i << "] = " << Z[i] << "\n";

    return 0;
}
```

Операция **transform**: начальная позиция и размер данных задаются итераторами.  
Используется стандартная операция «plus»



## Пример №1: сложение векторов

```
#include <thrust/device_vector.h>
#include <thrust/transform.h>
#include <thrust/functional.h>
#include <iostream>

int main(void) {

    thrust::device_vector<float> X(3);
    thrust::device_vector<float> Y(3);
    thrust::device_vector<float> Z(3);

    X[0] = 10; X[1] = 20; X[2] = 30;
    Y[0] = 15; Y[1] = 35; Y[2] = 10;

    thrust::transform(X.begin(), X.end(),
        Y.begin(),
        Z.begin(),
        thrust::plus<float>());

    for (size_t i = 0; i < Z.size(); i++)
        std::cout << "Z[" << i << "] = " << Z[i] << "\n";

    return 0;
```

Поэлементный вывод результата



## Пример №1: сложение векторов

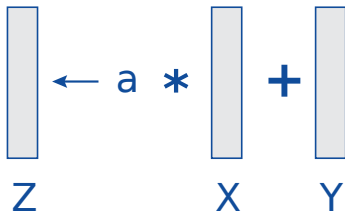
```
[dmikushin@tesla-cmc vector_addition]$ nvcc --version
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2011 NVIDIA Corporation
Built on Thu_Nov_17_17:38:12_PST_2011
Cuda compilation tools, release 4.1, V0.2.1221
[dmikushin@tesla-cmc vector_addition]$ make
nvcc vector_addition.cu -o vector_addition
[dmikushin@tesla-cmc vector_addition]$ ./vector_addition
Z[0] = 25
Z[1] = 55
Z[2] = 40
```



## Пример №2: SAXPY

SAXPY:  $z \leftarrow a * x + y$

```
for (int i = 0; i < N; ++i)  
    Z[i] = a * X[i] + Y[i];
```





## Пример №2: SAXPY – класс-функтор

```
#include <thrust/device_vector.h>
#include <thrust/transform.h>
#include <thrust/functional.h>
#include <iostream>

struct saxpy {
    float a;

    saxpy(float a) : a(a) {}

    __host__ __device__ float operator()(float x, float y) {
        return a * x + y;
    }
};
```



## Пример №2: SAXPY – класс-функтор

```
#include <thrust/device_vector.h>
#include <thrust/transform.h>
#include <thrust/functional.h>
#include <iostream>
```

```
struct saxpy {
```

```
    float a;
```

```
    saxpy(float a) : a(a) {}
```

```
    __host__ __device__ float operator()(float x, float y) {
        return a * x + y;
    }
```

```
};
```

Стандартной операции «saxpy» в Thrust нет, поэтому мы реализуем её в виде класса-**функтора** (функционального объекта)





## Пример №2: SAXPY – main

```
int main(void) {  
  
    thrust::device_vector<float> X(3), Y(3), Z(3);  
  
    X[0] = 10; X[1] = 20; X[2] = 30;  
    Y[0] = 15; Y[1] = 35; Y[2] = 10;  
  
    float a = 2.0f;  
  
    thrust::transform(X.begin(), X.end(),  
                     Y.begin(),  
                     Z.begin(),  
                     saxpy(a));  
  
    for (size_t i = 0; i < Z.size(); i++)  
        std::cout << "Z[" << i << "] = " << Z[i] << "\n";  
  
    return 0;  
}
```



## Пример №2: SAXPY – main

```
int main(void) {  
  
    thrust::device_vector<float> X(3), Y(3), Z(3);  
  
    X[0] = 10; X[1] = 20; X[2] = 30;  
    Y[0] = 15; Y[1] = 35; Y[2] = 10;  
  
    float a = 2.0f;  
  
    thrust::transform(X.begin(), X.end(),  
                     Y.begin(),  
                     Z.begin(),  
                     saxpy(a));  
  
    for (size_t i = 0; i < Z.size(); i++)  
        std::cout << "Z[" << i << "] = " << Z[i] << "\n";  
  
    return 0;  
}
```

Thrust-трансформация с функтором saxpy, определённым пользователем

Пример №3: SAXPY,  $\lambda$ -выражения (Thrust 1.5+)

```
using namespace thrust::placeholders;

int main(void) {

    thrust::device_vector<float> X(3), Y(3), Z(3);

    X[0] = 10; X[1] = 20; X[2] = 30;
    Y[0] = 15; Y[1] = 35; Y[2] = 10;

    float a = 2.0f;

    thrust::transform(X.begin(), X.end(),
                     Y.begin(),
                     Z.begin(),
                     a * _1 + _2);

    for (size_t i = 0; i < Z.size(); i++)
        std::cout << "Z[" << i << "] = " << Z[i] << "\n";

}
```

Пример №3: SAXPY,  $\lambda$ -выражения (Thrust 1.5+)

```
using namespace thrust::placeholders;

int main(void) {

    thrust::device_vector<float> X(3), Y(3), Z(3);

    X[0] = 10; X[1] = 20; X[2] = 30;
    Y[0] = 15; Y[1] = 35; Y[2] = 10;

    float a = 2.0f;

    thrust::transform(X.begin(), X.end(),
                     Y.begin(),
                     Z.begin(),
                     a * _1 + _2);

    for (size_t i = 0; i < Z.size(); i++)
        std::cout << "Z[" << i << "] = " << Z[i] << "\n";

}
```

Реализация **saxpy** с использованием  
маркеров подстановки: **\_1** и **\_2**

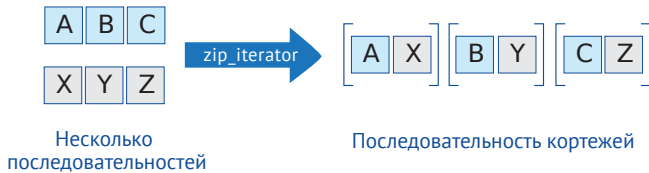


## Типы трансформаций

- Унарная:  $X[i] = f(A[i])$
- Бинарная:  $X[i] = f(A[i], B[i])$
- Тернарная:  $X[i] = f(A[i], B[i], C[i])$
- Обобщённая:  $X[i] = f(A[i], B[i], C[i], \dots)$



## zip-итераторы





## Пример №4: тернарная трансформация

```
#include <thrust/device_vector.h>
#include <thrust/transform.h>
#include <thrust/functional.h>
#include <thrust/iterator/zip_iterator.h>
#include <iostream>

struct linear_combo {
    __host__ __device__ float operator()(thrust::tuple<float,float,float> t) {

        float x, y, z;

        thrust::tie(x,y,z) = t;

        return 2.0f * x + 3.0f * y + 4.0f * z;

    }
};
```



## Пример №4: тернарная трансформация

```
#include <thrust/device_vector.h>
#include <thrust/transform.h>
#include <thrust/functional.h>
#include <thrust/iterator/zip_iterator.h>
#include <iostream>
```

```
struct linear_combo {
```

```
    __host__ __device__ float operator()(thrust::tuple<float,float,float> t) {
```

```
        float x, y, z;
```

```
        thrust::tie(x,y,z) = t;
```

```
        return 2.0f * x + 3.0f * y + 4.0f * z;
```

```
    }
```

```
};
```

Функтор, оперирующий кортежами





## Пример №4: тернарная трансформация

```
#include <thrust/device_vector.h>
#include <thrust/transform.h>
#include <thrust/functional.h>
#include <thrust/iterator/zip_iterator.h>
#include <iostream>

struct linear_combo {
    __host__ __device__ float operator()(thrust::tuple<float,float,float> t) {
        float x, y, z;
        thrust::tie(x,y,z) = t;
        return 2.0f * x + 3.0f * y + 4.0f * z;
    }
};
```

Разбиение кортежа на компоненты

thrust::tie(x,y,z) = t;

return 2.0f \* x + 3.0f \* y + 4.0f \* z;

}

};



## Пример №4: тернарная трансформация

```
#include <thrust/device_vector.h>
#include <thrust/transform.h>
#include <thrust/functional.h>
#include <thrust/iterator/zip_iterator.h>
#include <iostream>

struct linear_combo {
    __host__ __device__ float operator()(thrust::tuple<float,float,float> t) {

        float x, y, z;

        thrust::tie(x,y,z) = t;

        return 2.0f * x + 3.0f * y + 4.0f * z;
    }
};
```

Вычисление результата

return 2.0f \* x + 3.0f \* y + 4.0f \* z;



## Пример №4: тернарная трансформация

```
int main(void) {  
  
    thrust::device_vector<float> X(3), Y(3), Z(3);  
    thrust::device_vector<float> U(3);  
  
    X[0] = 10; X[1] = 20; X[2] = 30;  
    Y[0] = 15; Y[1] = 35; Y[2] = 10;  
    Z[0] = 20; Z[1] = 30; Z[2] = 25;  
  
    thrust::transform(  
        thrust::make_zip_iterator(thrust::make_tuple(X.begin(), Y.begin(), Z.begin())),  
        thrust::make_zip_iterator(thrust::make_tuple(X.end(),  
            Y.end(),  
            Z.end())),  
        U.begin(),  
        linear_combo());  
  
    for (size_t i = 0; i < Z.size(); i++)  
        std::cout << "U[" << i << "] = " << U[i] << "\n";  
  
    return 0;  
}
```



## Пример №4: тернарная трансформация

```
int main(void) {  
  
    thrust::device_vector<float> X(3), Y(3), Z(3);  
    thrust::device_vector<float> U(3);  
  
    X[0] = 10; X[1] = 20; X[2] = 30;  
    Y[0] = 15; Y[1] = 35; Y[2] = 10;  
    Z[0] = 20; Z[1] = 30; Z[2] = 25;  
  
    thrust::transform(  
        thrust::make_zip_iterator(thrust::make_tuple(X.begin(), Y.begin(), Z.begin())),  
        thrust::make_zip_iterator(thrust::make_tuple(X.end(),  
            Y.end(),  
            Z.end())),  
        U.begin(),  
        linear_combo());  
  
    for (size_t i = 0; i < Z.size(); i++)  
        std::cout << "U[" << i << "] = " << U[i] << "\n";  
  
    return 0;  
}
```

Входные данные задаются  
zip\_iterator'ами



## Пример №5: сумма

```
#include <thrust/device_vector.h>
#include <thrust/transform.h>
#include <thrust/functional.h>
#include <iostream>

int main(void) {
    thrust::device_vector<float> X(3);
    X[0] = 10; X[1] = 30; X[2] = 20;

    float result = thrust::reduce(X.begin(), X.end());

    std::cout << "sum is " << result << "\n";

    return 0;
}
```



## Пример №5: сумма

```
#include <thrust/device_vector.h>
#include <thrust/transform.h>
#include <thrust/functional.h>
#include <iostream>
```

```
int main(void) {
```

```
    thrust::device_vector<float> X(3);
```

```
    X[0] = 10; X[1] = 30; X[2] = 20;
```

```
    float result = thrust::reduce(X.begin(), X.end());
```

```
    std::cout << "sum is " << result << "\n";
```

```
    return 0;
```

```
}
```

Сумма – оператор по умолчанию  
для редукции



## Пример №6: поиск максимума

```
#include <thrust/device_vector.h>
#include <thrust/transform.h>
#include <thrust/functional.h>
#include <iostream>

int main(void) {

    thrust::device_vector<float> X(3);

    X[0] = 10; X[1] = 30; X[2] = 20;

    float init = 0.0f;

    float result = thrust::reduce(X.begin(), X.end(),
                                   init,
                                   thrust::maximum<float>());

    std::cout << "maximum is " << result << "\n";

    return 0;

}
```



## Пример №6: поиск максимума

```
#include <thrust/device_vector.h>
#include <thrust/transform.h>
#include <thrust/functional.h>
#include <iostream>

int main(void) {
    thrust::device_vector<float> X(3);

    X[0] = 10; X[1] = 30; X[2] = 20;

    float init = 0.0f;

    float result = thrust::reduce(X.begin(), X.end(),
        init,
        thrust::maximum<float>());

    std::cout << "maximum is " << result << "\n";

    return 0;
}
```

Редукция с оператором `maximum` и начальным значением 0.0 (только для неотрицательных чисел)





## Пример №7: индекс максимума

```
#include <thrust/device_vector.h>
#include <thrust/transform.h>
#include <thrust/functional.h>
#include <thrust/iterator/zip_iterator.h>
#include <iostream>

typedef thrust::tuple<int,int> Tuple;

struct max_index {

    __host__ __device__ Tuple operator()(Tuple a, Tuple b) {

        if (thrust::get<0>(a) > thrust::get<0>(b))
            return a;
        else
            return b;
    }

};
```



## Пример №7: индекс максимума

```
#include <thrust/device_vector.h>
#include <thrust/transform.h>
#include <thrust/functional.h>
#include <thrust/iterator/zip_iterator.h>
#include <iostream>
```

```
typedef thrust::tuple<int,int> Tuple;
```

```
struct max_index {
```

```
__host__ __device__ Tuple operator()(Tuple a, Tuple b) {
    if (thrust::get<0>(a) > thrust::get<0>(b))
        return a;
    else
        return b;
}
```

```
};
```

Функтор над кортежами (ключ, значение): сравнить ключи и вернуть кортеж



## Пример №7: индекс максимума

```
int main(void) {  
    thrust::device_vector<int> X(3), Y(3);  
  
    X[0] = 10; X[1] = 30; X[2] = 20; // values  
    Y[0] = 0; Y[1] = 1; Y[2] = 2;   // indices  
  
    Tuple init(X[0],Y[0]);  
  
    Tuple result = thrust::reduce(  
        thrust::make_zip_iterator(thrust::make_tuple(X.begin(), Y.begin())),  
        thrust::make_zip_iterator(thrust::make_tuple(X.end(), Y.end())),  
        init,  
        max_index());  
  
    int value, index;  
  
    thrust::tie(value,index) = result;  
  
    std::cout << "maximum value is " << value << " at index " << index << "\n";  
  
    return 0;  
}
```



## Пример №7: индекс максимума

```
int main(void) {  
    thrust::device_vector<int> X(3), Y(3);  
  
    X[0] = 10; X[1] = 30; X[2] = 20; // values  
    Y[0] = 0; Y[1] = 1; Y[2] = 2;   // indices  
  
    Tuple init(X[0],Y[0]);  
  
    Tuple result = thrust::reduce(  
        thrust::make_zip_iterator(thrust::make_tuple(X.begin(), Y.begin())),  
        thrust::make_zip_iterator(thrust::make_tuple(X.end(), Y.end())),  
        init,  
        max_index());  
  
    int value, index;  
  
    thrust::tie(value,index) = result;  
  
    std::cout << "maximum value is " << value << " at index " << index << "\n";  
  
    return 0;  
}
```

Дополнительный массив индексов



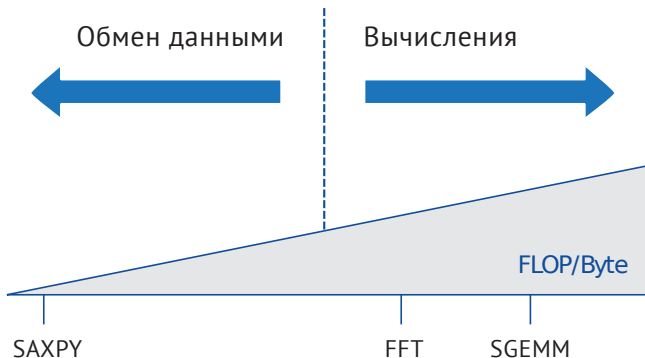
## Пример №7: индекс максимума

```
int main(void) {  
    thrust::device_vector<int> X(3), Y(3);  
  
    X[0] = 10; X[1] = 30; X[2] = 20; // values  
    Y[0] = 0; Y[1] = 1; Y[2] = 2;   // indices  
  
    Tuple init(X[0],Y[0]);  
  
    Tuple result = thrust::reduce(  
        thrust::make_zip_iterator(thrust::make_tuple(X.begin(), Y.begin())),  
        thrust::make_zip_iterator(thrust::make_tuple(X.end(), Y.end())),  
        init,  
        max_index());  
  
    int value, index;  
  
    thrust::tie(value,index) = result;  
  
    std::cout << "maximum value is " << value << " at index " << index << "\n";  
  
    return 0;  
}
```

Редукция с первым элементом в качестве начального значения и функтором `max_index`



## Вопросы производительности





## Вычислительная интенсивность

### В вышеприведённых примерах (FLOPS : byte)

vector addition      1 : 12

SAXPY                2 : 12

ternary transform    5 : 20

sum                  1 : 4

index of maximum    1 : 12

### Оптимальная интенсивность для GPU (FLOPS : byte)

GeForce GTX 280    7.0 : 1

GeForce GTX 480    7.6 : 1

Tesla C870           6.7 : 1

Tesla C1060          9.1 : 1

Tesla C2050          7.1 : 1



## Вычислительная интенсивность

### В вышеприведённых примерах (FLOPS : byte)

vector addition	1 : 12
SAXPY	2 : 12
ternary transform	5 : 20
sum	1 : 4
index of maximum	1 : 12

Реальная и оптимальная  
интенсивность  
**противоположны!**

### Оптимальная интенсивность для GPU (FLOPS : byte)

GeForce GTX 280	7.0 : 1
GeForce GTX 480	7.6 : 1
Tesla C870	6.7 : 1
Tesla C1060	9.1 : 1
Tesla C2050	7.1 : 1





## Пример №8: индекс максимума (оптимизация)

```
int main(void) {  
    thrust::device_vector<int>      X(3);  
    thrust::counting_iterator<int> Y(0);  
  
    X[0] = 10; X[1] = 30; X[2] = 20;  
  
    Tuple init(X[0],Y[0]);  
  
    Tuple result = thrust::reduce(  
        thrust::make_zip_iterator(thrust::make_tuple(X.begin(), Y)),  
        thrust::make_zip_iterator(thrust::make_tuple(X.end(), Y + X.size())),  
        init,  
        max_index());  
  
    int value, index;  
  
    thrust::tie(value,index) = result;  
  
    std::cout << "maximum value is " << value << " at index " << index << "\n";  
  
    return 0;  
}
```



## Пример №8: индекс максимума (оптимизация)

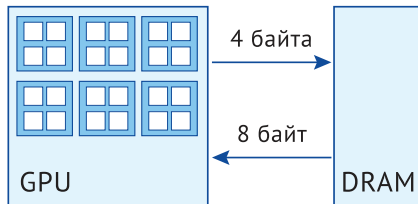
```
int main(void) {  
    thrust::device_vector<int>      X(3);  
    thrust::counting_iterator<int> Y(0);  
    X[0] = 10; X[1] = 30; X[2] = 20;  
    Tuple init(X[0],Y[0]);  
    Tuple result = thrust::reduce(  
        thrust::make_zip_iterator(thrust::make_tuple(X.begin(), Y)),  
        thrust::make_zip_iterator(thrust::make_tuple(X.end(), Y + X.size())),  
        init,  
        max_index());  
    int value, index;  
    thrust::tie(value,index) = result;  
    std::cout << "maximum value is " << value << " at index " << index << "\n";  
    return 0;  
}
```

В целях оптимизации используется  
специальный **counting\_iterator**  
вместо массива индексов



## Пример №8: индекс максимума (оптимизация)

### Исходная реализация



### Оптимизированный вариант





## Слияние (fusion) циклов

Два исходных цикла:

```
for (int i = 0; i < N; ++i)
    U[i] = F(X[i], Y[i], Z[i]);
for (int i = 0; i < N; ++i)
    V[i] = G(X[i], Y[i], Z[i]);
```

Два цикла, объединённые в один:

```
for (int i = 0; i < N; ++i) {
    U[i] = F(X[i], Y[i], Z[i]);
    V[i] = G(X[i], Y[i], Z[i]);
}
```



## Пример №9: слияние трансформаций

```
#include <thrust/device_vector.h>
#include <thrust/transform.h>
#include <thrust/functional.h>
#include <thrust/iterator/zip_iterator.h>
#include <iostream>

typedef thrust::tuple<float,float>      Tuple2;
typedef thrust::tuple<float,float,float> Tuple3;

struct linear_combo {
    __host__ __device__ Tuple2 operator()(Tuple3 t) {

        float x, y, z; thrust::tie(x,y,z) = t;

        float u = 2.0f * x + 3.0f * y + 4.0f * z;
        float v = 1.0f * x + 2.0f * y + 3.0f * z;

        return Tuple2(u,v);
    };
};
```



## Пример №9: слияние трансформаций

```
#include <thrust/device_vector.h>
#include <thrust/transform.h>
#include <thrust/functional.h>
#include <thrust/iterator/zip_iterator.h>
#include <iostream>

typedef thrust::tuple<float,float>      Tuple2;
typedef thrust::tuple<float,float,float> Tuple3;

struct linear_combo {
    __host__ __device__ Tuple2 operator()(Tuple3 t) {

        float x, y, z; thrust::tie(x,y,z) = t;

        float u = 2.0f * x + 3.0f * y + 4.0f * z;
        float v = 1.0f * x + 2.0f * y + 3.0f * z;

        return Tuple2(u,v);
    }
};
```

Входные и выходные данные  
объединяются в кортежи



## Пример №9: слияние трансформаций

```
int main(void) {  
  
    thrust::device_vector<float> X(3), Y(3), Z(3);  
    thrust::device_vector<float> U(3), V(3);  
  
    X[0] = 10; X[1] = 20; X[2] = 30;  
    Y[0] = 15; Y[1] = 35; Y[2] = 10;  
    Z[0] = 20; Z[1] = 30; Z[2] = 25;  
  
    thrust::transform(  
        thrust::make_zip_iterator(thrust::make_tuple(X.begin(), Y.begin(), Z.begin())),  
        thrust::make_zip_iterator(thrust::make_tuple(X.end(), Y.end(), Z.end())),  
        thrust::make_zip_iterator(thrust::make_tuple(U.begin(), V.begin())),  
        linear_combo());  
  
    for (size_t i = 0; i < Z.size(); i++)  
        std::cout << "U[" << i << "] = " << U[i] << " V[" << i << "] = " << V[i] << "\n";  
  
    return 0;  
}
```



## Пример №9: слияние трансформаций

```
int main(void) {
```

```
    thrust::device_vector<float> X(3), Y(3), Z(3);
```

```
    thrust::device_vector<float> U(3), V(3);
```

```
    X[0] = 10; X[1] = 20; X[2] = 30;
```

```
    Y[0] = 15; Y[1] = 35; Y[2] = 10;
```

```
    Z[0] = 20; Z[1] = 30; Z[2] = 25;
```

```
    thrust::transform(
```

```
        thrust::make_zip_iterator(thrust::make_tuple(X.begin(), Y.begin(), Z.begin())),
```

```
        thrust::make_zip_iterator(thrust::make_tuple(X.end(), Y.end(), Z.end())),
```

```
        thrust::make_zip_iterator(thrust::make_tuple(U.begin(), V.begin())),
```

```
        linear_combo());
```

```
    for (size_t i = 0; i < Z.size(); i++)
```

```
        std::cout << "U[" << i << "] = " << U[i] << " V[" << i << "] = " << V[i] << "\n";
```

```
    return 0;
```

```
}
```

Теперь и на выходе –  
последовательность кортежей



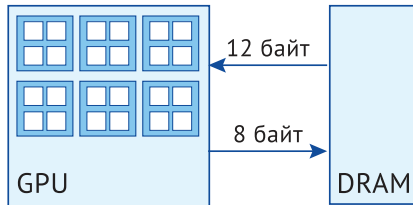


## Пример №9: слияние трансформаций

### Исходная реализация



### Оптимизированный вариант





## Слияние трансформации и редукции

Два исходных цикла:

```
for (int i = 0; i < N; ++i)
    Y[i] = F(X[i]);
for (int i = 0; i < N; ++i)
    sum += Y[i];
```

Два цикла, объединённые в один:

```
for (int i = 0; i < N; ++i)
    sum += F(X[i]);
```



## Пример №10: трансформация + редукция

```
#include <thrust/device_vector.h>
#include <thrust/transform.h>
#include <thrust/functional.h>
#include <iostream>

using namespace thrust::placeholders;

int main(void) {
    thrust::device_vector<float> X(3);

    X[0] = 10; X[1] = 30; X[2] = 20;

    float result = thrust::transform_reduce(
        X.begin(), X.end(),
        _1 * _1,
        0.0f,
        thrust::plus<float>());

    std::cout << "sum of squares is " << result << "\n";

    return 0;
}
```



## Пример №10: трансформация + редукция

```
#include <thrust/device_vector.h>
#include <thrust/transform.h>
#include <thrust/functional.h>
#include <iostream>

using namespace thrust::placeholders;

int main(void) {
    thrust::device_vector<float> X(3);

    X[0] = 10; X[1] = 30; X[2] = 20;

    float result = thrust::transform_reduce(
        X.begin(), X.end(),
        _1 * _1,
        0.0f,
        thrust::plus<float>());

    std::cout << "sum of squares is " << result << "\n";

    return 0;
}
```

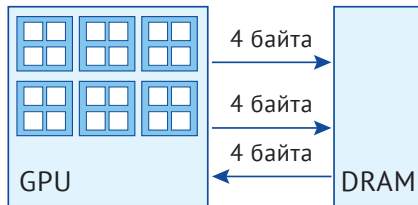
Вычисление квадратов и сложение в одной операции

```
float result = thrust::transform_reduce(
    X.begin(), X.end(),
    _1 * _1,
    0.0f,
    thrust::plus<float>());
```



## Пример №10: трансформация + редукция

Исходная реализация



Оптимизированный вариант





## Взаимодействие Thrust и CUDA

Преобразовать контейнер к обычному указателю:

```
thrust::device_vector<int> d_vec(4);  
  
int* ptr = thrust::raw_pointer_cast(&d_vec[0]);  
  
my_kernel<<< N / 256, 256 >>>(N, ptr);  
  
cudaMemcpyAsync(ptr, ... );
```



## Взаимодействие Thrust и CUDA

«Обернуть» обычный массив в спец. контейнер Thrust:

```
int *raw_ptr;  
  
cudaMalloc((void**) &raw_ptr, N * sizeof(int));  
  
thrust::device_ptr<int> dev_ptr(raw_ptr);  
  
thrust::fill(dev_ptr, dev_ptr + N, (int) 0);  
  
dev_ptr[0] = 1;  
  
cudaFree(raw_ptr);
```



## Дополнительные примеры на GitHub:

- Monte Carlo Integration
- Run-Length Encoding
- Summed Area Table
- Moving Average
- Word Count
- VoronoiDiagram
- Graphics Interop
- Stream Compaction
- Lexicographical Sort
- Summary Statistics
- Histogram
- ...





## Приложения Thrust

**CUSP** **CUSP** – операции с разреженными матрицами и вычисления на графах с поддержкой CUDA

<http://code.google.com/p/cusp-library/>



**PETSc** – параллельное решение научных задач, моделируемых дифференциальными УРЧП

<http://www.mcs.anl.gov/petsc/>



**Trilinos** – объектно-ориентированная библиотека для решения сложных научных и инженерных задач

<http://trilinos.sandia.gov/>



## Ресурсы

- ❖ Презентация Nathan Bell:  
Rapid Problem Solving Using Thrust
- ❖ Презентация Ty McKercher:  
Using Thrust to Sort CUDA FORTRAN Arrays