Разработка на CUDA с использованием Thrust

Михаил Смирнов

```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>

int main(void)
{
    // сгенерировать 32M случайных чисел на хосте
    thrust::host_vector<int> h_vec(32 << 20);
    thrust::generate(h_vec.begin(), h_vec.end(), rand);

    // скопировать данные на устройство
    thrust::device_vector<int> d_vec = h_vec;

    // отсортировать (846M ключей в секунду на GeForce GTX 480)
    thrust::sort(d_vec.begin(), d_vec.end());

    // скопировать обратно на хост
    thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());

    return 0;
}
```

Thrust

- Библиотека шаблонов для CUDA
  - похожа на STL
- Контейнеры
  - основная память и память устройства
- Алгоритмы
  - transform, sort, reduce, scan, …
- Итераторы

Контейнеры

- Автоматическое управление памятью

```
thrust::device_vector<int> d_vec1 (10);
// доступ к памяти устройства с хоста
d_vec1[0] = 11;
d_vec1[1] = 12;
// вектор в основной памяти
thrust::host_vector<int> h_vec (10);
// копирование данных в память устройства
thrust::device_vector<int> d_vec2 = h_vec;
// чтение данных из памяти устройства
std::cout << d_vec1[5] << " " << d_vec2[5] << std::endl;
```

Контейнеры

- Совместимость с STL

```
// STL список в основной памяти
std::list<int> h_list;
h_list.push_back (10);
h_list.push_back (11);

// вектор в памяти устройства
thrust::device_vector<int> d_vec;
// инициализация значениями из списка
thrust::copy(h_list.begin(), h_list.end(), d_vec.begin());

// альтернативный подход: использование конструктора
thrust::device_vector<int> d_vec(h_list.begin(), h_list.end());
```

Отдельное пространство имен

- Позволяет избежать конфликтов с STL

```
// вектор в основной памяти
thrust::host_vector<int> h_vec (10);

// сортировка средствами Thrust
thrust::sort (h_vec.begin(), h_vec.end());
// сортировка средствами STL
std::sort (h_vec.begin(), h_vec.end());

// получить доступ к именам из пространства thrust
using namespace thrust;

// вызов без использования квалификатора
int sum = reduce (d_vec.begin(), d_vec.end());
```

Итераторы

- Пара итераторов задает диапазон

```
// выделить память устройства
device_vector<int> d_vec(10);
// объявление итераторов
device_vector<int>::iterator begin  = d_vec.begin();
device_vector<int>::iterator end    = d_vec.end();
device_vector<int>::iterator middle = begin + 5;

// просуммировать половинки вектора отдельно
int sum_half1 = reduce(begin , middle);
int sum_half2 = reduce(middle, end);

// пустой диапазон
int empty = reduce(begin, begin);
```

Итераторы

- Работают как указатели

```
// переменные типа iterator
thrust::device_vector<int>::iterator begin = d_vec.begin ();
thrust::device_vector<int>::iterator end   = d_vec.end ();

// арифметические действия как с указателями
begin++;

// разыменование на хосте итераторов для памяти устройства
int a = *begin;
int b = begin[3];

// длина отрезка [begin, end)
int size = end - begin;
```

Итераторы

- Содержат информацию о расположении памяти (хост/устройство)
  - Автоматический выбор алгоритма

```
// генерация случайных чисел на хосте
host_vector<int> h_vec (10);
generate (h_vec.begin (), h_vec.end (), rand);
// скопировать данные на устройство
device_vector<int> d_vec = h_vec;

// посчитать сумму на хосте
int h_sum = reduce (h_vec.begin (), h_vec.end ());
// посчитать сумму на устройстве
int d_sum = reduce (d_vec.begin (), d_vec.end ());
```

Алгоритмы

- Поэлементные операции
  - for_each, transform, gather, scatter, …
- Редукции
  - reduce, inner_product, reduce_by_key, …
- Префиксные суммы
  - inclusive_scan, inclusive_scan_by_key, …
- Сортировка
  - sort, stable_sort, sort_by_key, …

Алгоритмы

- Работают с диапазонами
  - один или несколько

```
// выделить память на устройстве
device_vector<int> A(10);
device_vector<int> B(10);
device_vector<int> C(10);
// сортировка A (in-place)
sort(A.begin(), A.end());
// A -> B
copy(A.begin(), A.end(), B.begin());
// A + B -> C
transform(A.begin(), A.end(), B.begin(), C.begin(), plus<int>());
```

Алгоритмы

- Стандартные операторы

```
// выделить память
device_vector<int> A(10);
device_vector<int> B(10);
device_vector<int> C(10);

// A + B -> C
transform(A.begin(), A.end(), B.begin(), C.begin(), plus<int>());
// A - B -> C
transform(A.begin(), A.end(), B.begin(), C.begin(), minus<int>());

// произведение всех элементов (редукция с умножением)
int product = reduce(A.begin(), A.end(), 1, multiplies<int>());
```

Алгоритмы

- Стандартные типы данных

```
// выделить память
device_vector<int> i_vec= ...
device_vector<float> f_vec= ...

// сумма целых чисел
int i_sum= reduce(i_vec.begin(), i_vec.end());

// сумма вещественных чисел одинарной точности
float f_sum= reduce(f_vec.begin(), f_vec.end());
```

Алгоритмы

- Пользовательские типы данных
- Пользовательские операции

Алгоритмы

```
// вычислить коэффициент наклона прямой
struct GetSlope {
    __host__ __device__
    float operator() (float2 vec) {
        return vec.x / vec.y;
    }
};

// выделить память
device_vector<float2> d_vec;
Init (d_vec);
device_vector<float2> d_slopes (d_vec.size ());
// создать функтор
GetSlope func;
// преобразовать каждый элемент
transform (d_vec.begin (), d_vec.end (), d_slopes.begin (), func);
```

Алгоритмы

```
// сравнить компоненту x двух структур float2
struct CompareFloat2 {
    __host__ __device__
    bool operator() (float2 a, float2 b) {
        return a.x < b.x;
    }
};

// выделить память
device_vector<float2> vec = ...
// создать функтор
CompareFloat2 comp;
// сортировать элементы по компоненте x
sort(vec.begin(), vec.end(), comp);
```

Взаимодействие с обычным кодом

- Преобразование итераторов в указатели

```
// выделение памяти
thrust::device_vector<int> d_vec(4);

// получение указателя на память вектора
int* ptr = thrust::raw_pointer_cast(&d_vec[0]);

// использование ptr в обычном CUDA C
my_kernel<<< N / 256, 256 >>>(N, ptr);

// Указатель ptr не может быть разыменован на хосте!
```

Взаимодействие с обычным кодом

- device_ptr - обертка для указателей

```
// обычный указатель на память устройства
int* raw_ptr;
cudaMalloc((void**) &raw_ptr, N * sizeof(int));
// обертка
device_ptr<int> dev_ptr(raw_ptr);

// использование device_ptr в алгоритмах Thrust
fill(dev_ptr, dev_ptr + N, (int) 0);
// доступ к памяти через device_ptr
dev_ptr[0] = 1;

// освободить память
cudaFree(raw_ptr);
```

Thrust

- Контейнеры
  - автоматическое управление памятью
- Итераторы
  - знают, в какой памяти расположены данные
- Алгоритмы
  - работают с диапазонами
  - пользовательские типы данных
  - пользовательские операции

Thrust

- Тщательно тестируется
- Открытый код
  - репозиторий на Google Code
  - Apache v2
- Активное сообщество
  - 300+ участников в cusp-users (Google Groups)

Ссылки

- Thrust на Google Code
- thrust-users на Google Groups
- Основано на презентации “High-Productivity CUDA Development with the Thrust Template Library”, Nathan Bell (NVIDIA Research), GTC 2010