Thrust

Stanisław Kaźmierczak, MiNI PW

Agenda

- wprowadzenie
- wektory
- algorytmy
- iteratory
- optymalizacja

Czym jest 

?

- biblioteka szablonów C++ dla CUDA
- bazuje na STL-u
- pozwala implementować wysokiej wydajności aplikacje równoległe
- minimalizuje nakład pracy programisty (szybki rozwój złożonych aplikacji)
- interfejs wysokiego poziomu w pełni współpracujący z CUDA C/C++
- dostarcza bogatą kolekcję równoległych operacji takich jak skanowanie, sortowanie, redukcja, które razem mogą implementować złożony algorytm napisany zwięzłym, czytelnym kodem

cd..

- typy generyczne
- automatyczny wybór najbardziej efektywnej implementacji
- możliwość wykorzystania w prototypowaniu aplikacji na CUDA, gdzie wydajność programisty jest najważniejsza
- możliwość wykorzystania w produkcji, gdzie solidność i kompletne wykonanie są kluczowe
- Thrust v1.3.0 jest kompatybilny z CUDA 3.2 (preferowana wersja), CUDA 3.1 i CUDA 3.0
- Thrust v1.4.0 będzie zamieszczony w CUDA 4.0 Toolkit

Hello World

```
#include <thrust/version.h>
#include <iostream>

int main(void)
{
    int major = THRUST_MAJOR_VERSION;
    int minor = THRUST_MINOR_VERSION;

    std::cout << "Thrust v" << major << "." << minor << std::endl;

    return 0;
}
```

Hello World

```
#include <thrust/version.h>
#include <iostream>

int main(void)
{
    int major = THRUST_MAJOR_VERSION;
    int minor = THRUST_MINOR_VERSION;

    std::cout << "Thrust v" << major << "." << minor << std::endl;

    return 0;
}
```

wyjście: Thrust v1.3

Wektory

- wektor jest strukturą danych funkcjonującą jak dynamiczna tablica
- `host_vector` – przechowywany w pamięci hosta
- `device_vector` – przechowywany w pamięci karty
- deklaracja analogiczna do STL-a (`std::vector`):
  - `thrust::host_vector`
  - `thrust::device_vector`
- są typami generycznymi, których rozmiar może być zmieniany dynamicznie

Wektory – przykład 1

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <iostream>

int main(void){
    thrust::host_vector<int> H(4);

    H[0] = 14;
    H[1] = 20;
    H[2] = 38;
    H[3] = 46;

    std::cout << "H has size " << H.size() << std::endl; //4

    for(int i = 0; i < H.size(); i++)
        std::cout << "H[" << i << "] = " << H[i] << std::endl;    //14, 20, 38, 46
```

Wektory – przykład 1

```
    H.resize(2);

    std::cout << "H now has size " << H.size() << std::endl; //2

    thrust::device_vector<int> D = H;

    D[0] = 99;
    D[1] = 88;

    for(int i = 0; i < D.size(); i++)
        std::cout << "D[" << i << "] = " << D[i] << std::endl;    //99, 88

    // H and D are automatically deleted when the function returns
    return 0;
}    //main
```

Wektory – przykład 2

```
// initialize all ten integers of a device_vector to 1
thrust::device_vector<int> D(10, 1);

// set the first seven elements of a vector 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, 3, ...
thrust::sequence(H.begin(), H.end());

// copy all of H back to the beginning of D
thrust::copy(H.begin(), H.end(), D.begin());
```

Wektory cd.

- ukryte `cudaMalloc`, `cudaMemcpy`, `cudaFree` podczas tworzenia, kopiowania wektorów
- `thrust::copy i std::copy`
- w wektorze, który jest *de facto* tablicą, iterator należy traktować jako wskaźnik na element tablicy, np. `H.end(), D.begin() + 5`
- `v.end()` wskazuje na jeden element po ostatnim elemencie wektora v
- iteratory zawierają więcej informacji niż wskaźniki – mówią o typie wektora
- stąd iterator zwrócony przez `H.begin()` jest innego typu niż `D.begin()`

Wektory cd.

- na podstawie typu iteratora używana jest odpowiednia implementacja funkcji (hosta lub karty)
- proces ten nazywany jest *static dispatching*
- łatwa integracja z kontenerami z STL-a
- podobnie jak STL, Thrust dopuszcza użycie “surowych” wskaźników; spowoduje to wykonanie wersji hosta
- jeżeli wskaźnik jest w rzeczywistości wskaźnikiem do pamięci karty, potrzebne jest przed wywołaniem funkcji “opakowanie” go przy pomocy `thrust::device_ptr`

Iterator - opakowanie

```
size_t N = 10;

// raw pointer to device memory
int * raw_ptr;
cudaMalloc((void **) &raw_ptr, N * sizeof(int));

// wrap raw pointer with a device_ptr
thrust::device_ptr<int> dev_ptr(raw_ptr);

// use device_ptr in thrust algorithms
thrust::fill(dev_ptr, dev_ptr + N, (int) 0);
```

Iterator - rozpakowanie

```
size_t N = 10;


// create a device_ptr
thrust::device_ptr<int> dev_ptr = thrust::device_malloc<int>(N);


// extract raw pointer from device_ptr
int * raw_ptr = thrust::raw_pointer_cast(dev_ptr);
```

Iteratory cd.

```
// allocate device vector
thrust::device_vector<int> d_vec(4);

// obtain raw pointer to device vector’s memory
int * ptr = thrust::raw_pointer_cast(&d_vec[0]);

// use ptr in a CUDA C kernel
my_kernel<<<N/256, 256>>>(N, ptr);
```

Iteratory cd.

```
// create an STL list with 4 values
std::list<int> stl_list;

stl_list.push_back(10);
stl_list.push_back(20);
stl_list.push_back(30);
stl_list.push_back(40);

// initialize a device_vector with the list
thrust::device_vector<int> D(stl_list.begin(), stl_list.end());

// copy a device_vector into an STL vector
std::vector<int> stl_vector(D.size());
thrust::copy(D.begin(), D.end(), stl_vector.begin());
```

Algorytmy

- Thrust dostarcza wiele powszechnie znanych algorytmów równoległych, np. transformacje, redukcje, skany, sortowanie
- wiele z nich ma bezpośrednią analogię w STL, np. `thrust::sort` i `std::sort`
- wszystkie algorytmy posiadają implementację zarówno dla hosta, jak i dla karty (rodzaj iteratora determinuje rodzaj implementacji)
- wszystkie argumenty algorytmów powinny być po tej samej stronie: hosta lub karty; wyjątkiem jest tu funkcja `copy`
- typy generyczne

Transformacje

- są to algorytmy, które stosują pewną operację do każdego elementu w pewnym przedziale i umieszczają wynik w docelowym przedziale
- `thrust::{fill, sequence, replace, transform}`

Transformacje - przykład

```
#include <thrust/device_vector.h>
#include <thrust/transform.h>
#include <thrust/sequence.h>
#include <thrust/copy.h>
#include <thrust/fill.h>
#include <thrust/replace.h>
#include <thrust/functional.h>
#include <iostream>

int main(void){
    // allocate three device_vectors with 10 elements
    thrust::device_vector<int> X(10);
    thrust::device_vector<int> Y(10);
    thrust::device_vector<int> Z(10);

    // initialize X to 0,1,2,3, ....
    thrust::sequence(X.begin(), X.end());
```

Transformacje - przykład

```
    // 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 tens
    thrust::replace(Y.begin(), Y.end(), 1, 10);

    // print Y
    thrust::copy(Y.begin(), Y.end(), std::ostream_iterator<int>(std::cout, "\n"));

    return 0;
}       //main
```

Funktory - przykład

```
void saxpy_slow(float A, thrust::device_vector<float>& X, thrust::device_vector<float>& Y)
{
    thrust::device_vector<float> temp(X.size());

    // temp <- A
    thrust::fill(temp.begin(), temp.end(), A);

    // temp <- A * X
    thrust::transform(X.begin(), X.end(), temp.begin(), temp.begin(), thrust::multiplies<float>());

    // Y <- A * X + Y
    thrust::transform(temp.begin(), temp.end(), Y.begin(), Y.begin(), thrust::plus<float>());
}
```

Funktory - przykład

```
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));
}
```

Redukcja

- Algorytm redukcji używa binarnej operacji w celu *zredukowania* sekewncji wejściowej do pojedynczej wartości, np. :

  - suma tablicy liczb jest uzyskiwana poprzez redukcję tablicy z operacją dodawania

  - maksimum tablicy jest uzyskiwane poprzez redukcję z operatorem dwuargumentowym zwracającym większą z liczb

```
int sum = thrust::reduce(D.begin(), D.end(), (int) 0, thrust::plus<int>());

int sum = thrust::reduce(D.begin(), D.end(), (int) 0);

int sum = thrust::reduce(D.begin(), D.end())
```

Redukcja cd.

- thrust::{count, count_if, min_element, max_element, is_sorted, inner_product} i kilka innych

```
#include <thrust/count.h>
#include <thrust/device_vector.h>
...
thrust::device_vector<int> vec(5,0);
vec[1] = 1;
vec[3] = 1;
vec[4] = 1;

int result = thrust::count(vec.begin(), vec.end(), 1);  //3
```

thrust::transform_reduce

```
// square<T> computes the square of a number f(x) -> x*x
template <typename T>
struct square
{
    __host__ __device__
        T operator()(const T& x) const {
            return x * x;
        }
};
```

thrust::transform_reduce

```
int main(void){
    float x[4] = {1.0, 2.0, 3.0, 4.0};

    // transfer to device
    thrust::device_vector<float> d_x(x, x + 4);

    // setup arguments
    square<float>        unary_op;
    thrust::plus<float> binary_op;
    float init = 0;

    // compute norm
    float norm = std::sqrt( thrust::transform_reduce(d_x.begin(), d_x.end(), unary_op, init, binary_op) );

    return 0;
}
```

Inclusive scan

```
#include <thrust/scan.h>

int data[6] = {1, 0, 2, 2, 1, 3};

thrust::inclusive_scan(data, data + 6, data); // in-place scan

// data is now {1, 1, 3, 5, 6, 9}
```

Exclusive scan

```
#include <thrust/scan.h>

int data[6] = {1, 0, 2, 2, 1, 3};

thrust::exclusive_scan(data, data + 6, data); // in-place scan

// data is now {0, 1, 1, 3, 5, 6}
```

Reordering

- `copy_if` – kopiuje elementy spełniające predykat
- `partition` – zmienia kolejność elementów wg. predykatu:

  elementy spełniające poprzedzają elementy niespełniające predykat
- `remove, remove_if` – usuwa elementy niespełniające predykatu
- `unique` – usuwa duplikaty w ramach kolekcji

Sortowanie

- Thrust dostarcza kilka funkcji do sortowania i przeorganizowywania danych według danych kryteriów
- `thrust::sort` i `thrust::stable_sort` są bezpośrednimi odpowiednikami `sort` i `stable_sort` z STLa

```
#include <thrust/sort.h>

...

const int N = 6;

int A[N] = {1, 4, 2, 8, 5, 7};

thrust::sort(A, A + N);

// A is now {1, 2, 4, 5, 7, 8}
```

Sortowanie cd.

```
#include <thrust/sort.h>

...

const int N = 6;

int    keys[N] = {  1,   4,   2,   8,   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'}
```

```
#include <thrust/sort.h>

#include <thrust/functional.h>

...

const int N = 6;

int A[N] = {1, 4, 2, 8, 5, 7};

thrust::stable_sort(A, A + N, thrust::greater<int>());

// A is now {8, 7, 5, 4, 2, 1}
```

Fancy iterators

- idea zaczerpnięta z Boost Iterator Library
- służą jako narzędzie do rozwiązywania szerszej klasy problemów przy użyciu algorytmów
- algorytmy “nie widzą” różnicy między zwykłymi iteratorami
- `constant_iterator`, `counting_iterator`, `transform_iterator`, `permutation_iterator`, `zip_iterator`

constant_iterator

- zwraca zawsze tą samą wartość
- iterator reprezentujący wskaźnik na przedział stałych wartości
- użyteczny przy tworzeniu przedziału wypełnionego tą samą wartością bez jawnego przechowywania w pamięci

```
#include <thrust/iterator/constant_iterator.h>

...

// create iterators

thrust::constant_iterator<int> first(10);

thrust::constant_iterator<int> last = first + 3;

first[0]   // returns 10

first[1]   // returns 10

first[100] // returns 10

// sum of [first, last)

thrust::reduce(first, last);   // returns 30 (i.e. 3 * 10)
```

counting_iterator

- gdy potrzebna jest sekwencja rosnących wartości
- zachowuje się jak tablica, ale nie wymaga tyle pamięci
- wartość generowana jest *on-the-fly* podczas wywołania

```
#include <thrust/iterator/counting_iterator.h>

...

// create iterators

thrust::counting_iterator<int> first(10);

thrust::counting_iterator<int> last = first + 3;

first[0]   // returns 10

first[1]   // returns 11

first[100] // returns 110

// sum of [first, last)

thrust::reduce(first, last);   // returns 33 (i.e. 10 + 11 + 12)
```

transform_iterator

- iterator reprezentujący wskaźnik na przedział wartości po transformacji przez funkcję

```
#include <thrust/iterator/transform_iterator.h>

thrust::device_vector<int> vec(3);

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

// create iterator (type omitted)

... first = thrust::make_transform_iterator(vec.begin(), negate<int>());

... last  = thrust::make_transform_iterator(vec.end(),   negate<int>());


first[0]   // returns -10

first[1]   // returns -20

first[2]   // returns -30


// sum of [first, last)

thrust::reduce(first, last);   // returns -60 (i.e. -10 + -20 + -30)
```

transform_iterator cd.

- specyfikowanie pełnego typu iteratora jest niewygodne, np.

```
thrust::device_vector<float>::iterator FloatIterator
```

- z tego powodu powszechną praktyką stało się wywoływanie `make_transform_iterator` w argumentach algorytmu
- poniższy przykład pozwala uniknąć tworzenia zmiennych do przechowywania `first` i `last`:

```
// sum of [first, last)

thrust::reduce(thrust::make_transform_iterator(vec.begin(), negate<int>()),
               thrust::make_transform_iterator(vec.end(),   negate<int>()));
```

permutation_iterator

```
// gather locations
thrust::device_vector<int> map(4);
map[0] = 3; map[1] = 1; map[2] = 0; map[3] = 5;

// array to gather from
thrust::device_vector<int> source(6);
source[0] = 10; source[1] = 20; source[2] = 30; source[3] = 40; source[4] = 50; source[5] = 60;

// fuse gather with reduction: sum = source[map[0]] + source[map[1]] + … = 130
int sum = thrust::reduce(thrust::make_permutation_iterator(source.begin(), map.begin()),
                         thrust::make_permutation_iterator(source.begin(), map.end()));
```

zip_iterator

```
// initialize vectors
thrust::device_vector<int>  A(3);
thrust::device_vector<char> B(3);
A[0] = 10;  A[1] = 20;  A[2] = 30;
B[0] = 'x'; B[1] = 'y'; B[2] = 'z';
// create iterator (type omitted)
first = thrust::make_zip_iterator(thrust::make_tuple(A.begin(), B.begin()));
last  = thrust::make_zip_iterator(thrust::make_tuple(A.end(),   B.end()));
first[0]   // returns tuple(10, 'x')
first[1]   // returns tuple(20, 'y')
first[2]   // returns tuple(30, 'z')
// maximum of [first, last)
thrust::maximum< tuple<int,char> > binary_op;
thrust::tuple<int,char> init = first[0];
thrust::reduce(first, last, init, binary_op); // returns tuple(30, 'z')
```

zip_iterator

- dostaje wiele sekwencji, zwraca sekwencję *krotek* (uporządkowany ciąg stałych wartości różnych typów danych)
- wygląda jak tablica struktur, jest strukturą tablic
- wykorzystywany w wielu algorytmach, które przeważnie przyjmują jako parametr jedną, max. 2 sekwencje
- Pozwala implementować programy bardziej efektywniej

  Przykład: punkty 3d

  - przechowywanie w tablicy typu float3 jest nieefektywne (brak *coalesced memmory access)*

  - dzięki iteratorowi możemy przechowywać współrzędne w trzech oddzielnych tablicach (zasada: struktura tablic, a nie tablica struktur)

  - za pomocą iteratora tworzymy wirtualną tablicę wektorów 3d, które można użyć jako parametrów do algorytmów Thrusta

Optymalizacja

- fusion

  - łączenie powiązanych operacji razem
- struktura tablic

  - zapewnienie memmory coalescing
- niejawne sekwencje

  - eliminuje odwoływanie się do pamięci

Fusion – wersja nieoptymalna

```
// define transformation f(x) -> x^2
struct square
{
    __host__ __device__
    float operator()(float x)
    {
        return x * x;
    }
};

float snrm2_slow(device_vector<float>& x)
{
    // without fusion
    device_vector<float> temp(x.size());
    transform(x.begin(), x.end(), temp.begin(), square());
    return sqrt( reduce(temp.begin(), temp.end()) );
}
```

Fusion – wersja optymalna (3.8x szybciej)

```
// define transformation f(x) -> x^2
struct square
{
    __host__ __device__
    float operator()(float x)
    {
        return x * x;
    }
};

float snrm2_fast(device_vector<float>& x)
{
    // with fusion
    return sqrt( transform_reduce(x.begin(), x.end(), square(), 0.0f, plus<float>());
}
```

Struktura tablic

- tablica struktur
  - często nie przestrzega *coalescing rules*
  - `device_vector<float3>`
- struktura tablic
  - spełnia *coalescing rules*
  - `device_vector<float> x, y, z`
- przykład: rotacja wektora 3d
  - struktura tablic jest 2.8 raza szybsza

Rotacja wektora – tablica struktur

```
struct rotate_float3
{
    __host__ __device__
    float3 operator()(float3 v)
    {
        float x = v.x; float y = v.y; float z = v.z;
        float rx = 0.36f*x + 0.48f*y + -0.80f*z;
        float ry =-0.80f*x + 0.60f*y + 0.00f*z;
        float rz = 0.48f*x + 0.64f*y + 0.60f*z;
        return make_float3(rx, ry, rz);
    }
};

...
device_vector<float3> vec(N);
transform(vec.begin(), vec.end, vec.begin(), rotate_float3());
```

Rotacja wektora – struktura tablic

```
struct rotate_tuple{
    __host__ __device__
    tuple<float,float,float> operator()(tuple<float,float,float> v){
        float x = get<0>(v); float y = get<1>(v); float z = get<2>(v);
        float rx = 0.36f*x + 0.48f*y + -0.80f*z;
        float ry =-0.80f*x + 0.60f*y + 0.00f*z;
        float rz = 0.48f*x + 0.64f*y + 0.60f*z;
        return make_tuple(rx, ry, rz);
    }
};


...
device_vector<float> x(N), y(N), z(N);
transform(make_zip_iterator(make_tuple(x.begin(), y.begin(), z.begin())),
          make_zip_iterator(make_tuple(x.end(), y.end(), z.end())),
          make_zip_iterator(make_tuple(x.begin(), y.begin(), z.begin())),
          rotate_tuple());
```

Niejawne sekwencje

- unikać jawnego przechowywania sekwencji
  - stałe sekwencje: [1, 1, 1, ...]
  - przyrostowe sekwencje: [0, 1, 2, ...]
- niejawne sekwencje nie wymagają magazynowania
  - `constant_iterator`
  - `counting_iterator`
- przykład: znajdowanie indeksu najmniejszego elementu
  - przy użyciu iteratorów algorytm działa 3.4 raza szybciej

Najmniejszy indeks

```
// return the smaller of two tuples
struct smaller_tuple
{
    tuple<float,int> operator()(tuple<float,int> a, tuple<float,int> b)
    {
        if (a < b)
            return a;
        else
            return b;
    }
};
```

Najmniejszy indeks – jawna sekwencja

```
int min_index(device_vector<float>& vec)
{
    // create explicit index sequence [0, 1, 2, ... )
    device_vector<int> indices(vec.size());
    sequence(indices.begin(), indices.end());
    tuple<float,int> init(vec[0],0);
    tuple<float,int> smallest;

    smallest = reduce(make_zip_iterator(make_tuple(vec.begin(), indices.begin())),
                      make_zip_iterator(make_tuple(vec.end(), indices.end())),
                      init,
                      smaller_tuple());
    return get<1>(smallest);
}
```

Najmniejszy indeks – niejawna sekwencja

```
int min_index(device_vector<float>& vec)
{
    // create implicit index sequence [0, 1, 2, ... )
    counting_iterator<int> begin(0);
    counting_iterator<int> end(vec.size());
    tuple<float,int> init(vec[0],0);
    tuple<float,int> smallest;

    smallest = reduce(make_zip_iterator(make_tuple(vec.begin(), begin)),
                      make_zip_iterator(make_tuple(vec.end(), end)),
                      init,
                      smaller_tuple());
    return get<1>(smallest);
}
```

Podsumowanie optymalizacji

- fusion – 3.8 razy szybciej
- struktura tablic – 2.8 razy szybciej
- niejawne sekwencje – 3.4 razy szybciej

Bibliografia

- http://code.google.com/p/thrust/

Dziękuję za uwagę