Programowanie współbieżne i rozproszone

WYKŁAD 5

Jan Kazimirski

Programowanie GPU 2/2

Thrust

- Język CUDA C bazuje na języku C – języku o dosyć niskim poziomie abstrakcji („macro assembler)
- Implementowanie złożonych problemów w CUDA C jest kłopotliwe
- CUDA C wymaga podejmowania decyzji związanych z „niskopoziomową” reprezentacją problemu (bloki, wątki, sposoby alokacji pamięci itd.)

Thrust c.d.

- **Thrust** jest biblioteką wysokiego poziomu opartą o system szablonów C++.
- Używa modelu znanego z biblioteki STL (kontenery i algorytmy)
- Ukrywa szczegóły sprzętowe i niskopoziomowe
- Pozwala skoncentrować się na problemie, a nie na szczegółach jego implementacji
- Pozwala na szybkie tworzenie kodu

Thrust c.d.

- **Thrust** jest częścią tool-kitu dostępnego na stronie NVIDIA
- Kompilacja nie wymaga dodatkowych narzędzi
- Programy wykorzystujące Thrust muszą włączać odpowiednie pliki nagłówkowe
- Dokumentację API można znaleźć pod adresem:

  http://docs.thrust.googlecode.com/hg/index.html

Thrust - Przykład programu suma wektorów 1/2

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <stdio.h>
#include <thrust/functional.h>
#include <thrust/fill.h>
#include <thrust/transform.h>

#define N 100

void show(thrust::host_vector<int> v) {
    int i;
    for(i=0;i<N;i++) printf("%d ",v[i]);
    printf("\n");
}
```

Thrust - Przykład programu suma wektorów 1/2

```
int main(void) {
    thrust::device_vector<int> d_vA(N);
    thrust::device_vector<int> d_vB(N);
    thrust::device_vector<int> d_vC(N);
    thrust::fill(d_vA.begin(),d_vA.end(),1);
    thrust::fill(d_vB.begin(),d_vB.end(),2);

    thrust::plus<int> op;
    thrust::transform(
                d_vA.begin(),
                d_vA.end(),
                d_vB.begin(),
                d_vC.begin(),op);

    thrust::host_vector<int> h_vC = d_vC;
    show(h_vC);

    return 0;
```

Elementy Thrust

- **Kontenery** – pozwalają na przechowywanie danych dowolnego typu
- **Iteratory** – udostępniają dane kontenera
- **Algorytmy** – przetwarzają dane z kontenerów
- **Obiekty funkcyjne** – wykorzystywane do definiowania operacji używanych w algorytmach
- **Generator pseudolosowy**

Kontenery

- Thrust definiuje jeden typ kontenera w dwóch wersjach. Kontener ten odpowiada STL-owemu kontenerowi vector.
  - Klasa **thrust::host_vector** – kontener alokowany w pamięci hosta
  - Klasa **thrust::device_vector** – kontener alokowany w pamięci urządzenia
- Funkcjonalność tych kontenerów odpowiada funkcjonalności STL-owego wektora

Kontenery c.d.

- Kopiowanie danych między hostem i urządzeniem jest bardzo łatwe – używa się operatora przypisania:

```
thrust::host_vector<int>  hVec(100);
thrust::device_vector<int> dVec(100);
dVec = hVec;
// ...
hVec = dVec;
```

Kontenery c.d.

- Klasy wektorów Thrust definiują wiele wygodnych metod do manipulowania danymi (tworzenie, usuwanie oraz dostęp do elementów).
- Elementy wektorów dostępne są poprzez operator indeksowania [] (Uwaga! Dla vector_device niewydajne).
- Wektorami zwykle manipulujemy za pomocą iteratorów.

Iteratory

- Przetwarzanie danych z kontenerów wymaga zwykle podanie zakresu.
- Metody begin() i end() wektora zwracają iterator do pierwszego i ostatniego elementu wektora.
- Iteratory implementują również metody pozwalające na dostęp do dowolnego elementu i wyliczanie odstępu między elementami.

Iteratory - przykład

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

int main(void) {
    thrust::device_vector<int>::iterator iter;

    thrust::host_vector<int>  hVec(10,5);
    thrust::device_vector<int> dVec(10);

    dVec = hVec;
    for(iter=dVec.begin();iter<dVec.end();iter++)
                    std::cout << *iter << std::endl;

    return 0;
}
```

Iteratory c.d.

- Iteratory zawierają informację o typie wskaźnika (device, host) – te same algorytmy mogą działać na obu typach
- Thrust zawiera metody do konwersji „gołego” wskaźnika CUDA (cudaMalloc) na iterator i odwrotnie.

Wybrane algorytmy

- Thrust implementuje zbiór algorytmów analogicznych do algorytmów STL.
- Algorytmy Thrust mogą działać na wektorach na hoście, jak i na urządzeniu (zależnie od użytego iteratora)
- Sposób korzystania z algorytmów Thrust bardzo przypomina używanie algorytmów STL.

Wybrane algorytmy c.d.

- Wyszukiwanie
- Kopiowanie
- Redukcja
- Sortowanie
- Partycjonowanie
- Łączenie
- Operacje teorii mnogości
- Transformacje

Wyszukiwanie

- Funkcje związane z wyszukiwaniem:
  - **find** – wyszukuje podany element w zbiorze.
  - **find_if**, **find_if_not** – wyszukuje w zbiorze element spełniający lub niespełniający danego warunku
  - **mismatch** – porównuje dwa zbiory elementów i zwraca pozycję pierwszej różnicy elementów

find

```
01: #include <thrust/host_vector.h>
02: #include <thrust/device_vector.h>
03: #include <iostream>
04:
05: int main(void) {
06:     int arr[] = {1,3,5,2,6,9,11,32,4,99};
07:     thrust::host_vector<int> hVec(arr,arr+10);
08:     thrust::device_vector<int> dVec = hVec;
09:     thrust::device_vector<int>::iterator iter;
10:
11:     iter = thrust::find(dVec.begin(),dVec.end(),5);
12:     std::cout << iter-dVec.begin() << std::endl;
13:     iter = thrust::find(dVec.begin(),dVec.end(),33);
14:     std::cout << iter-dVec.begin() << std::endl;
15:
16:     return 0;
17: }
```

find_if

```
01: #include <thrust/host_vector.h>
02: #include <thrust/device_vector.h>
03: #include <iostream>
04:
05: struct pred {
06:   __device__ bool operator()(int x) {
07:     return x>10;
08:   }
09: };
10:
11: int main(void) {
12:     int arr[] = {1,3,5,2,6,9,11,32,4,99};
13:     thrust::host_vector<int> hVec(arr,arr+10);
14:     thrust::device_vector<int> dVec = hVec;
15:     thrust::device_vector<int>::iterator iter;
16:
17:     iter = thrust::find_if(dVec.begin(),dVec.end(),pred());
18:     std::cout << iter-dVec.begin() << std::endl;
19:
20:     return 0;
21: }
```

mismatch

```
01: #include <thrust/host_vector.h>
02: #include <thrust/device_vector.h>
03: #include <iostream>
04:
05: int main(void) {
06:     int arr1[] = {1,3,5,2,6,9,11,32,4,99};
07:     int arr2[] = {1,3,5,2,6,11,8,30,22,15};
08:     thrust::device_vector<int> dVec1(arr1,arr1+10);
09:     thrust::device_vector<int> dVec2(arr2,arr2+10);
10:     typedef thrust::device_vector<int>::iterator Iterator;
11:     thrust::pair<Iterator,Iterator> result;
12:     result = thrust::mismatch(dVec1.begin(), dVec1.end(), dVec2.begin());
13:
14:     std::cout << *result.first << std::endl <<
15:                  *result.second << std::endl <<
16:                  result.first-dVec1.begin() << std::endl;
17:
18:     return 0;
19: }
```

Kopiowanie

- Funkcje związane z kopiowaniem danych:
  - **copy** – kopiuje elementy z jednego zbioru do drugiego
  - **copy_n** – kopiuje n elementów z jednego zbioru do drugiego
  - **swap_ranges** – Wymienia elementy dwóch zbiorów między sobą.

copy

```
01: #include <thrust/host_vector.h>
02: #include <thrust/device_vector.h>
03: #include <iostream>
04:
05: int main(void) {
06:     int arr1[] = {1,3,5,2,6,9,11,32,4,99};
07:     thrust::device_vector<int> dVec1(arr1,arr1+10);
08:     thrust::device_vector<int> dVec2(10);
09:
10:     thrust::copy(dVec1.begin(),dVec1.end(),dVec2.begin());
11:
12:     for(int i=0;i<10;i++) std::cout << dVec2[i] << " ";
13:     std::cout << std::endl;
14:
15:     return 0;
16: }
```

swap_ranges

```
01: #include <thrust/host_vector.h>
02: #include <thrust/device_vector.h>
03: #include <iostream>
04:
05: int main(void) {
06:     int arr1[] = {1,3,5,2,6,9,11,32,4,99};
07:     int arr2[] = {1,1,1,1,1,1,1,1,1,1};
08:     thrust::device_vector<int> dVec1(arr1,arr1+10);
09:     thrust::device_vector<int> dVec2(arr2,arr2+10);
10:
11:     thrust::swap_ranges(dVec1.begin()+2,dVec1.begin()+5,dVec2.begin()+2);
12:
13:     for(int i=0;i<10;i++) std::cout << dVec2[i] << " ";
14:     std::cout << std::endl;
15:
16:     return 0;
17: }
```

Kopiowanie c.d.

- Funkcje **gather** i **scatter** pozwalają na kopiowanie elementów pomiędzy zbiorami zgodnie z określoną mapą
- Alternatywne wersje powyższych funkcji **gather_if** i **scatter_if** pozwalają dodatkowo okreslić warunek jaki muszą spełniać kopiowane elementy

Redukcja

- Algorytmy związane z redukcją możemy podzielić na kilka grup:
  - Redukcja za pomocą określonej operacji
  - Zliczanie elementów
  - Redukcja oparta o porównanie
  - Wyszukiwanie wartości ekstremalnych
  - Redukcje z transformacją
  - Redukcje oparte o operacje logiczne
  - Redukcje na podstawie własności zbioru

Redukcja za pomocą określonej operacji

- Funkcja **reduce** pozwala przekształcić zbiór wejściowy w pojedynczą wartość
- Zależnie od wariantu funkcji możemy zastosować sumowanie lub inną operację
- Uwaga! Wektoryzacja powoduje, że elementy mogą być przetwarzane w innej niż podana kolejności.

reduce

```
01: #include <thrust/host_vector.h>
02: #include <thrust/device_vector.h>
03: #include <iostream>
04:
05: int main(void) {
06:     int arr1[] = {1,2,3,4,5,6,7,8,9,10};
07:     thrust::device_vector<int> dVec1(arr1,arr1+10);
08:
09:     std::cout << thrust::reduce(dVec1.begin(),dVec1.end()) << std::endl;
10:     std::cout << thrust::reduce(dVec1.begin(),dVec1.end(),10) << std::endl;
11:
12:     return 0;
13: }
```

Zliczanie elementów

- Funkcja **count** zlicza ile elementów w podanym zakresie jest równe danej wartości
- Funkcja **count_if** zlicza dla ilu elementów w podanym zakresie podane wyrażenie jest prawdziwe.

count

```
01: #include <thrust/host_vector.h>
02: #include <thrust/device_vector.h>
03: #include <thrust/count.h>
04: #include <iostream>
05:
06: int main(void) {
07:     int arr1[] = {1,2,3,4,5,6,7,8,9,1};
08:     thrust::device_vector<int> dVec1(arr1,arr1+10);
09:
10:     std::cout << thrust::count(dVec1.begin(),dVec1.end(),1) << std::endl;
11:     std::cout << thrust::count(dVec1.begin(),dVec1.end(),11) << std::endl;
12:
13:     return 0;
14: }
```

Redukcja oparta o porównanie

- Funkcja **equal** pozwala sprawdzić czy elementy dwóch podanych zakresów są takie same.
- Funkcja jest też dostępna w wariancie umożliwiającym podanie funkcji stosowanej do porównywania elementów.

Wyszukiwanie wartości ekstremalnych

- Dostępne funkcje:
  - **min_element** – znajduje najmniejszy element zbioru
  - **max_element** – znajduje największy element zbioru
  - **minmax_element** – znajduje parę elementów: największy i najmniejszy
- Funkcje dostępne są też w wersji pozwalającej na dostarczenie własnej funkcji porównującej.

minmax_element

```
01: #include <thrust/host_vector.h>
02: #include <thrust/device_vector.h>
03: #include <thrust/count.h>
04: #include <iostream>
05:
06: int main(void) {
07:     int arr1[] = {1,2,3,4,5,6,7,8,9,1};
08:     thrust::device_vector<int> dVec1(arr1,arr1+10);
09:     typedef thrust::device_vector<int>::iterator Iterator;
10:
11:     thrust::pair<Iterator,Iterator> result =
thrust::minmax_element(dVec1.begin(),dVec1.end());
12:
13:     std::cout << *result.first << " " << result.first-dVec1.begin() << std::endl;
14:     std::cout << *result.second << " " << result.second-dVec1.begin() << std::endl;
15:
16:     return 0;
17: }
```

Redukcje z transformacją

- Funkcja **inner_product** liczy iloczyn skalarny dla danego zbioru
- Funkcja **transform_reduce** pozwala połączyć redukcję z transformacją zbioru za pomocą określonej operacji

Redukcje oparte o operacje logiczne

- Wykorzystują operacje logiczne na zbiorach. Dostępne funkcje:
  - **all_of** – sprawdza czy wszystkie elementy spełniają dany warunek
  - **any_of** – sprawdza czy jakikolwiek element spełnia dany warunek
  - **none_of** – sprawdza czy żaden element nie spełnia danego warunku.

any_f, all_of

```
01: #include <thrust/host_vector.h>
02: #include <thrust/device_vector.h>
03: #include <thrust/count.h>
04: #include <thrust/logical.h>
05: #include <iostream>
06:
07: struct is_one {
08:   __device__ bool operator()(int x) {
09:     return x==1;
10:   }
11: };
12:
13: int main(void) {
14:     int arr1[] = {1,2,3,4,5,6,7,8,9,1};
15:     thrust::device_vector<int> dVec1(arr1,arr1+10);
16:
17:
18:     std::cout << thrust::any_of(dVec1.begin(),dVec1.end(),is_one()) << std::endl;
19:     std::cout << thrust::all_of(dVec1.begin(),dVec1.end(),is_one()) << std::endl;
20:
21:     return 0;
22: }
```

Redukcje na podstawie własności zbioru

- Funkcje:
  - **is_partitioned** – sprawdza czy zbiór jest podzielony na elementy spełniające i niespełniające danego warunku
  - **is_sorted** – sprawdza czy zbiór jest posortowany
  - **is_sorted_until** – zwraca liczbę elementów zbioru, które są posortowane.

Sortowanie

- Algorytm sortowania zbioru występuje w dwóch wariantach **sort** i **stable_sort**.
- Funkcja **stable_sort** zachowuje porządek elementów równorzędnych
- Funkcja **sort_by_key** pozwala sortować dane w postaci par klucz-wartość.
- Zamiast domyślnego operatora porównania można do sortowania użyć własnej funkcji porównującej.

sort

```
01: #include <thrust/host_vector.h>
02: #include <thrust/device_vector.h>
03: #include <thrust/sort.h>
04: #include <iostream>
05:
06:
07: int main(void) {
08:     int arr1[] = {1,2,3,4,5,6,7,8,9,1};
09:     thrust::device_vector<int> dVec1(arr1,arr1+10);
10:
11:     thrust::sort(dVec1.begin(),dVec1.end());
12:
13:     for(int i=0;i<10;i++) std::cout << dVec1[i] << " ";
14:     std::cout << std::endl;
15:
16:     return 0;
17: }
```

Partycjonowanie

- Algorytmy te pozwalają na częściowe uporządkowanie zbioru tzn. osobno elementy spełniające i niespełniające warunku
- Funkcje:
- **partition** – podział zbioru zgodnie z warunkiem
- **partition_copy** – podział zbioru i umieszczenie rezultatu w innym miejscu
- **stable_partition**, **stable_partition_copy** – stabilne wersje powyższych algorytmów

Łączenie

- Funkcja **merge** pozwala połączyć dwa posortowane zbiory w jeden zbiór wynikowy
- Dostępna jest wersja funkcji **merge** pozwalająca dostarczyć własny operator porównania zamiast domyślnego

Operacje teoriomnogościowe

- Dostępne operacje teoriomnogościowe:
  - **set_difference** – różnica zbiorów
  - **set_intersection** – cześć wspólna zbiorów
  - **set_symmetric_difference** – różnica symetryczna
  - **set_union** – suma zbiorów
- Powyższe funkcje wymagają posortowanych zbiorów.
- Alternatywne wersje funkcji pozwalają dostarczyć własną funkcję porównującą.

Transformacje

- Thrust dysponuje algorytmami pozwalającymi na różnorodne transformacje zbioru wejściowego
- Podstawowe grupy transformacji:
  - Generowanie zbioru
  - Wypełnianie zbioru
  - Transformacje zbioru wejściowego w wyjściowy
  - Przetwarzanie każdego elementu zbioru
  - Zastępowanie elementów zbioru

Generowanie zbioru

- Funkcja **generate** – wypełnia kontener w podanym zakresie wartościami wyliczonymi za pomocą podanej funkcji generującej.
- Funkcja **generate_n** – j.w. ale podaje się ile elementów ma być utworzonych.
- Funkcja **sequence** – wypełnia kontener wartościami tworzącymi sekwencję.

generate

```
01: #include <thrust/host_vector.h>
02: #include <thrust/device_vector.h>
03: #include <thrust/generate.h>
04: #include <iostream>
05:
06: struct five {
07:   __device__ int operator()() {
08:     return 5;
09:   }
10: };
11:
12: int main(void) {
13:     thrust::device_vector<int> dVec1(10);
14:
15:     thrust::generate(dVec1.begin(),dVec1.end(),five());
16:
17:     for(int i=0;i<10;i++) std::cout << dVec1[i] << " ";
18:     std::cout << std::endl;
19:
20:     return 0;
21: }
```

sequence

```
01: #include <thrust/host_vector.h>
02: #include <thrust/device_vector.h>
03: #include <thrust/sequence.h>
04: #include <iostream>
05:
06: int main(void) {
07:     thrust::device_vector<int> dVec1(10);
08:
09:     thrust::sequence(dVec1.begin(),dVec1.end());
10:
11:     for(int i=0;i<10;i++) std::cout << dVec1[i] << " ";
12:     std::cout << std::endl;
13:
14:     return 0;
15: }
```

Wypełnianie zbioru

- Funkcja **fill** – Wypełnia kontener w podanym zakresie określoną wartością
- Funkcja **fill_n** – Jak **fill**, ale podaje się liczbę elementów do wypełnienia.

fill

```
01: #include <thrust/host_vector.h>
02: #include <thrust/device_vector.h>
03: #include <thrust/fill.h>
04: #include <iostream>
05:
06: int main(void) {
07:     thrust::device_vector<int> dVec1(10);
08:
09:     thrust::fill(dVec1.begin(),dVec1.end(),11);
10:
11:     for(int i=0;i<10;i++) std::cout << dVec1[i] << " ";
12:     std::cout << std::endl;
13:
14:     return 0;
15: }
```

Transformacje zbioru wejściowego w wyjściowy

- Funkcja **transform** – dokonuje transformacji zbioru (lub 2 zbiorów) wejściowych na zbiór wyjściowy według zadanej funkcji.
- Funkcja **transform_if** – Podobna do **transform**, ale pozwala dokonać transformacji warunkowej na podstawie danych wejściowych lub dodatkowej mapy.

transform

```
01: #include <thrust/host_vector.h>
02: #include <thrust/device_vector.h>
03: #include <thrust/transform.h>
04: #include <iostream>
05:
06: int main(void) {
07:     int arr1[] = {1,3,5,2,6,9,11,32,4,99};
08:     int arr2[] = {1,1,1,1,1,1,1,1,1,1};
09:     thrust::device_vector<int> dVec1(arr1,arr1+10);
10:     thrust::device_vector<int> dVec2(arr2,arr2+10);
11:     thrust::device_vector<int> dVec3(10);
12:
13:     thrust::plus<int> op;
14:     thrust::transform(dVec1.begin(),dVec1.end(),dVec2.begin(),dVec3.begin(),op);
15:
16:     for(int i=0;i<10;i++) std::cout << dVec3[i] << " ";
17:     std::cout << std::endl;
18:
19:     return 0;
20: }
```

Przetwarzanie każdego elementu zbioru

- Funkcja **for_each** – wykonuje określoną operację na zbiorze w podanym zakresie.
- Funkcja **for_each_n** – Podobna do **for_each**, ale podaje się liczbę elementów do przetworzenia

Zastępowanie elementów zbioru

- Funkcja **replace** – pozwala zastąpić w zbiorze podany element nową wartością.
- Funkcja **replace_if** – jw. ale podaje się warunek jaki muszą spełnić zastępowane wartości.
- Funkcja **replace_copy**, **replace_copy_if** – ten wariant funkcji **replace** wykonuje zastępowanie z jednoczesnym kopiowaniem do innego zbioru.

Zaawansowane iteratory

- Thrust definiuje dodatkowe „specjalne” iteratory pozwalające zwiększyć możliwości algorytmów.
- Algorytm nie widzi różnicy między zwykłym a specjalnym iteratorem.
- Specjalne iteratory:
  - constant_iterator
  - counting_iterator
  - transform_iterator
  - permutation_iterator
  - zip_iterator

constant_iterator

- Reprezentuje „zbiór” złożony z jednej wartości
- Może być wykorzystany do wypełniania kontenerów danymi.
- Funkcja **make_constant_iterator** może być użyta do łatwego utworzenia iteratora jako parametru algorytmu (nie wymaga podania dokładnego typu iteratora)

counting_iterator

- Reprezentuje „zbiór” będący sekwencją rosnących wartości.
- Pozwala na łatwe tworzenie sekwencji lub wykonywanie operacji z udziałem sekwencji
- Dostępna jest funkcja **make_counting_iterator**

transform_iterator

- Reprezentuje „zbiór” będący przedziałem wartości po przetworzeniu przez funkcję (transformacja)
- Połączony z innym algorytmem pozwala połączyć wywołania jądra zwiększając wydajność
- Dostępna jest funkcja **make_transform_iterator**

permutation_iterator

- Reprezentuje „zbiór” danych będący podzbiorem podanego przedziału, ale z wartościami przestawionymi zgodnie z podanym schematem
- Wygodny przy operacjach typu scatter/gather
- Dostępna jest funkcja generująca **make_permutation_iterator**

zip_iterator

- Łączy kilka sekwencji wejściowych w jedną sekwencję
- Użyteczny gdy wymagana jest operacja na bardziej złożonych danych (np.. wielowymiarowych).
- Umożliwia algorytmom pracę na wielu strumieniach danych z użyciem tylko jednego iteratora.
- Dodatkowe funkcje: **make_tuple**, **make_zip_iterator**

Obiekty funkcyjne

- Większość algorytmów pozwala na dostarczenie własnej wersji funkcji wykorzystywanej do przetwarzania danych.
- Thrust udostępnia predefiniowane funkcje w postaci tzw. obiektów funkcyjnych dla różnych typów operacji: arytmetycznych, porównania, logicznych, bitowych i innych.
- Predefiniowane obiekty funkcyjne dostarczane są w postaci szablonów.

Predefiniowane obiekty funkcyjne

- Operatory arytmetyczne: **plus**<T>, **minus**<T>, **multiplies**<T>, **divides**<T>, **modulus**<T>, **negate**<T>
- Operatory porównania: **equal_to**<T>, **not_equal_to**<T>, **greater**<T>, **less**<T>, **greater_equal**<T>, **less_equal**<T>
- Operatory logiczne: **logical_and**<T>, **logical_or**<T>, **logical_not**<T>
- Operatory bitowe: **bit_and**<T>, **bit_or**<T>, **bit_xor**<T>