Porównanie algorytmów sortowania na CPU i GPU

Programowanie GPU - Projekt Zaliczeniowy

Patryk Zając 297110

1 Wstęp

W ramach projektu przygotowałem program napisany w technologii CUDA C++, w którym zaimplementowałem algorytmy sortowania zarówno na CPU jak i na GPU (quicksort, ranksort, odd-even sort, mergesort i thrust::sort)

2 Algorytmy sortowania

2.1 QuickSort

Algorytm QuickSort to algorytm sortowania, który operuje na tablicach lub listach danych. Jego główną ideą jest podział zestawu danych na mniejsze fragmenty i rekursywne sortowanie tych fragmentów. Oto krótka charakteryzacja działania algorytmu QuickSort:

- Wybierz element **pivot** (punkt odniesienia) z zestawu danych. Może to być dowolny element z tablicy.
- Podziel dane na trzy części:
    - Elementy mniejsze od pivota (lewa podtablica).
    - Elementy równe pivotowi (środkowa podtablica).
    - Elementy większe od pivota (prawa podtablica).
- Rekursywnie sortuj lewą i prawą podtablicę, ponownie wybierając pivota i powtarzając proces.
- Po zakończeniu sortowania wszystkich podtablic, można je połączyć w jedną posortowaną tablicę.
- Algorytm jest kontynuowany, aż cała tablica zostanie posortowana.

Algorytm QuickSort jest wydajny i często stosowany w praktyce ze względu na swoją średnią złożoność czasową **O(n log n)**. Pomimo to, jego wydajność może być podatna na zmiany w wyborze pivota. W najgorszym przypadku, gdy pivota wybiera się nieoptymalnie, złożoność może wynieść **O(n^2)**. Istnieją jednak różne strategie wyboru pivota, takie jak wybór losowego elementu lub mediany, które pomagają zmniejszyć ryzyko wystąpienia najgorszego przypadku.

Algorytm QuickSort jest również efektywny na wielu platformach, w tym na procesorach CPU i kartach graficznych (GPU), dzięki czemu może być używany w kontekście przetwarzania równoległego.

2.2 Odd-Even Sort

Algorytm sortowania Odd-Even Sort (czasami nazywany także "**Brick Sort**") jest prostym algorytmem sortowania, który działa na zasadzie porównywania i zamiany par sąsiednich elementów w zestawie danych. Jest to algorytm porównawczy, który ma złożoność czasową **O(n^2)** w najgorszym przypadku. Oto krótka charakteryzacja jego działania:

- **Inicjalizacja**: Algorytm rozpoczyna się od pierwszego elementu zestawu danych i przechodzi po kolejnych parach elementów.
- **Porównywanie i zamiana**: Dla każdej pary sąsiednich elementów, algorytm porównuje je ze sobą. Jeśli element o niższym indeksie jest większy od elementu o wyższym indeksie, zamienia te dwa elementy miejscami.
- **Cykle nieparzyste i parzyste**: Algorytm wykona kolejno dwa cykle:
    - Cykl nieparzysty: W tym cyklu porównywane i zamieniane są elementy o nieparzystych indeksach (1, 3, 5, itd.).
    - Cykl parzysty: W tym cyklu porównywane i zamieniane są elementy o parzystych indeksach (0, 2, 4, itd.).
- **Powtarzanie cykli**: Cykle nieparzyste i parzyste są wykonywane naprzemiennie, aż cały zestaw danych zostanie posortowany. W każdym cyklu porównywane są pary sąsiednich elementów, co powoduje przesunięcie największych i najmniejszych elementów ku końcom zestawu danych.
- **Sprawdzenie warunku zakończenia**: Algorytm powtarza cykle nieparzyste i parzyste do momentu, gdy nie będzie wymagał żadnych zamian w danym cyklu. Jeśli w jednym cyklu nie została wykonana żadna zamiana, zestaw danych jest uważany za posortowany, a algorytm kończy działanie.

Algorytm Odd-Even Sort jest prosty w implementacji, ale nie jest najbardziej wydajny w porównaniu do innych zaawansowanych algorytmów sortowania, takich jak QuickSort czy MergeSort. Warto zauważyć, że jest on mniej efektywny niż te algorytmy, zwłaszcza na dużych zestawach danych. Jego główną zaletą jest prostota, co może być przydatne w niektórych prostych zastosowaniach.

2.3 MergeSort

Merge Sort to zaawansowany algorytm sortowania, który operuje na zasadzie podziału i scalania danych. Jest to algorytm sortowania stabilny, co oznacza, że zachowuje kolejność elementów o równych wartościach. Oto opis działania algorytmu Merge Sort:

- **Podział**: Pierwszym krokiem algorytmu jest podzielenie zestawu danych na dwie równe lub niemal równe części. To podejście jest rekurencyjne, więc każda z tych podtablic może być również podzielona na mniejsze podtablice, aż osiągniemy pojedyncze elementy.
- **Sortowanie podtablic**: Każda z podtablic jest rekurencyjnie sortowana za pomocą algorytmu Merge Sort. Proces ten kontynuuje się, aż osiągniemy podtablice zawierające pojedyncze elementy, które są uważane za posortowane.
- **Scalanie**: Kiedy mamy już posortowane podtablice, algorytm rozpoczyna proces scalania. W tym procesie elementy są porównywane i scalane w taki sposób, aby otrzymać jedną, posortowaną tablicę wynikową:
    - Porównywane są pierwsze elementy obu podtablic.
    - Element o mniejszej wartości jest przenoszony do tablicy wynikowej.
    - Ten proces jest kontynuowany, aż obie podtablice zostaną w pełni scalone.
- **Kontynuacja scalania**: Jeśli jedna z podtablic została już w pełni scalona, a druga jeszcze nie, pozostałe elementy z niescalonej podtablicy są bezpośrednio kopiowane do tablicy wynikowej, ponieważ są już posortowane.
- **Tablica wynikowa**: Po zakończeniu scalania wszystkich podtablic otrzymujemy jedną posortowaną tablicę wynikową.

Algorytm ten jest wydajny i może być stosowany do sortowania różnych typów danych. Jest szczególnie przydatny na zestawach danych o dużych rozmiarach, ponieważ jego złożoność czasowa wynosi **O(n log n)**, co sprawia, że jest bardziej efektywny niż niektóre inne algorytmy sortowania o złożoności czasowej O(n^2), takie jak algorytm Bubble Sort czy Insertion Sort. Jednakże, wymaga on dodatkowej pamięci do przechowywania tymczasowych podtablic, co może być wadą w przypadku bardzo dużych zestawów danych na platformach o ograniczonej pamięci.

2.4 Thrust :: sort

Jest to funkcja sortująca dostępna w bibliotece Thrust, która jest częścią NVIDIA CUDA Toolkit. Thrust jest biblioteką C++ zaprojektowaną do wydajnego programowania równoległego na kartach graficznych (GPU) przy użyciu technologii CUDA. Thrust::sort jest wygodnym

narzędziem do sortowania dużych ilości danych na GPU i wykorzystuje możliwości przetwarzania równoległego dostępne na kartach graficznych NVIDIA. Dzięki temu można uzyskać znaczący wzrost wydajności w porównaniu z sortowaniem na CPU, zwłaszcza w przypadku dużych zestawów danych.

3 Program w Cuda C++

3.1 Funkcje pomocnicze

3.1.1 Generowanie liczb

Do przygotowania tablicy używamy funkcji generateRandomData:

```
// Funkcja do generowania losowych danych wejściowych
void generateRandomData(int* data, int size)
{
    srand(static_cast<unsigned int>(time(nullptr)));
    for (int i = 0; i < size; ++i) {
        data[i] = rand() % 1000; // Zakładamy, że sortujemy liczby całkowite do 1000
    }
}
```

Przyjmuje ona pustą tablicę oraz liczbę elementów do wygenerowania, po czym wypełnia tę tablicę losowymi wartościami całkowitymi. Wywołanie funkcji wygląda następująco:

```
// Rozmiar danych wejściowych
int dataSize = 10000; // Zmień na żądany rozmiar

// Alokacja pamięci na CPU dla danych wejściowych
int* inputData = new int[dataSize];

// Generowanie losowych danych wejściowych
generateRandomData(inputData, dataSize);
```

3.1.2 Wypisywanie tablicy

W celach testowych, dodałem funkcję displayData do wypisywania elementów tablicy int*:

```
// Funkcja do wyświetlenia danych na konsoli
void displayData(int* data, int size) {
    for (int i = 0; i < size; ++i) {
        cout << data[i] << " ";
    }
    cout << endl;
}
```

3.1.3 Porównywanie tablic

Do sprawdzania czy algorytmy działają poprawnie mamy funkcję compareData. Będziemy sprawdzać czy algorytm posortował dobrze porównując jego wynik z wynikiem sortowania QuickSort na CPU:

```
// Funkcja do porównywania czy wyniki sortowań są identyczne
void compareData(int* data1, int* data2, int size) {
    bool equal = true;
    for (int i = 0; i < size; ++i) {
        if (data1[i] != data2[i]) {
            cout << "WRONG" << endl;
            equal = false;
            break;
        }
    }
    if (equal)
        cout << "CORRECT" << endl;
}
```

3.2 Algorytm QuickSort

3.2.1 CPU

Implementacja algorytmu quicksort na CPU wygląda następująco:

```cpp
// Algorytm quicksort na CPU
void quicksort(int* data, int left, int right)
{
    if (left < right)
    {
        // Wybór elementu pivot - środkowego elementu jako punktu odniesienia
        int pivot = data[(left + right) / 2];
        int i = left;
        int j = right;

        // Podział tablicy na dwie części - elementy mniejsze i większe od pivot
        while (i <= j)
        {
            // Przesuń wskaźnik 'i' w prawo, dopóki element jest mniejszy od pivot
            while (data[i] < pivot)
                i++;

            // Przesuń wskaźnik 'j' w lewo, dopóki element jest większy od pivot.
            while (data[j] > pivot)
                j--;

            // Jeśli 'i' nadal jest mniejsze lub równe 'j', zamień te dwa elementy miejscami.
            if (i <= j)
            {
                swap(data[i], data[j]);
                i++;
                j--;
            }
        }

        // Rekurencyjne wywołanie QuickSort dla lewej i prawej części tablicy
        if (left < j)
            quicksort(data, left, j);
        if (i < right)
            quicksort(data, i, right);
    }
}
```

Argumenty funkcji to:

- data: Wskaźnik do tablicy, która ma być posortowana.
- left: Indeks początkowy tablicy do posortowania.
- right: Indeks końcowy tablicy do posortowania.

Algorytm działa następująco:

1. Funkcja rozpoczyna sortowanie rekurencyjne, jeśli lewy indeks jest mniejszy od prawego, co oznacza, że są co najmniej dwie liczby do posortowania.
2. Element pivot jako środkowy element tablicy.
3. Tablica jest dzielona na dwie części: elementy mniejsze od pivot i większe od pivot.

4. Algorytm iteruje przez tablicę, przesuwając wskaźnik **i** w prawo, dopóki element jest mniejszy od pivota i wskaźnik **j** w lewo, dopóki element jest większy od pivota. Jeśli **i** jest nadal mniejsze lub równe **j**, zamieniane są miejscami te dwa elementy.

5. Po podziale tablicy, algorytm wywołuje się rekurencyjnie dla lewej i prawej części tablicy, aż cała tablica zostanie posortowana.

3.2.2 GPU

Ogólna idea tego algorytmu na GPU (jak np. wykorzystanie sortowania przez wybór i zastosowanie strumieni) jest zaczerpnięta z przykładowego kodu z NVIDIA Cuda-samples dostępnego na podanym repozytorium Github:

https://github.com/NVIDIA/cuda-samples/blob/master/Samples/3_CUDA_Features/cdpSimpleQuicksort/cdpSimpleQuicksort.cu

Kod ten został dostosowany do moich potrzeb i został rozwinięty o zastosowanie pamięci współdzielonej oraz operacji atomowych.

Funkcja do sortowania (selectionSort) przez wybieranie używana jest, gdy głębokość w algorytmie quicksort staje się zbyt duża lub liczba elementów spada poniżej pewnego progu. Jest to prosty algorytm sortowania działający na pojedynczym wątku GPU. Implementacja wygląda następująco:

```
// QUICKSORT GPU - Funkcja do sortowania przez wybieranie używana, gdy głębokość
// staje się zbyt duża lub liczba elementów spada poniżej pewnego progu.
__device__ void selectionSort(int* data, int left, int right)
{
    for (int i = left; i <= right; ++i)
    {
        int min_val = data[i];
        int min_idx = i;

        // Znajdź najmniejszą wartość w zakresie [left, right].
        for (int j = i + 1; j <= right; ++j)
        {
            int val_j = data[j];

            if (val_j < min_val)
            {
                min_idx = j;
                min_val = val_j;
            }
        }

        // Zamień wartości
        if (i != min_idx)
        {
            data[min_idx] = data[i];
            data[i] = min_val;
        }
    }
}
```

Przejdźmy teraz do głównego kernela odpowiedzialnego za nasz algorytm sortowania, jest on wywołany rekurencyjnie, na początku sprawdzamy czy nie jesteśmy zbyt głęboko lub czy nie pozostało mało elementów:

```
// Algorytm QuickSort, uruchamiający kolejny poziom rekurencji - GPU
__global__ void quicksortGPU(int* data, int left, int right, int depth)
{
    // Jeśli jesteśmy zbyt głęboko lub pozostała mała liczba elementów, używamy sortowania przez wybieranie
    if (depth >= MAX_DEPTH || right - left <= INSERTION_SORT)
    {
        selectionSort(data, left, right);
        return;
    }
```

MAX_DEPTH i INSERTION_SORT są zadeklarowane następująco:

```
#define MAX_DEPTH 16
#define INSERTION_SORT 32
```

Następnie następuję podział i partycjonowanie danych. W trakcie działania wykorzystuje pamięć współdzieloną (shared memory) na blok w celu przechowywania tymczasowych danych podczas partycjonowania:

```
// Inicjalizacja wskaźników na lewy i prawy koniec przetwarzanego fragmentu tablicy i pivot
int* low_ptr = data + left;
int* high_ptr = data + right;
int pivot = data[(left + right) / 2];

// Pamięć współdzielona na blok
extern __shared__ int sharedData[];

// Dokonanie partycjonowania
while (low_ptr <= high_ptr)
{
    // Znajdź kolejne wartości po lewej i prawej stronie do zamiany
    int low_val = *low_ptr;
    int high_val = *high_ptr;

    // Przesuń wskaźnik w lewo, dopóki element wskazywany jest mniejszy niż pivot.
    while (low_val < pivot)
    {
        low_ptr++;
        low_val = *low_ptr;
    }

    // Przesuń wskaźnik w prawo, dopóki element wskazywany jest większy niż pivot.
    while (high_val > pivot)
    {
        high_ptr--;
        high_val = *high_ptr;
    }

    // Jeśli punkty zamiany są prawidłowe, dokonaj zamiany w pamięci współdzielonej.
    if (low_ptr <= high_ptr)
    {
        sharedData[low_ptr - data] = high_val;
        sharedData[high_ptr - data] = low_val;
        low_ptr++;
        high_ptr--;
    }
}
```

Dalej wykonywana jest rekurencja. Kod posiada obsługę strumieni CUDA, co pozwala na równoległe wykonywanie sortowania w różnych strumieniach, co może przyspieszyć operację na GPU. Zastosowanie strumieni polega na uruchomieniu nowych bloków kernela do sortowania lewej i prawej części podzielonego fragmentu tablicy w nowych strumieniach CUDA. Ponadto używamy wywołania atomicExch, aby skopiować dane z pamięci współdzielonej z powrotem do globalnej pamięci:

```
    int new_right = high_ptr - data;
    int new_left = low_ptr - data;

    // Wykonaj operację w pamięci współdzielonej, aby przekopiować dane z
    // pamięci współdzielonej z powrotem do globalnej pamięci.
    for (int i = threadIdx.x; i < (new_right - left + 1); i += blockDim.x)
        atomicExch(&data[left + i], sharedData[i]);

    // Uruchom nowy blok do sortowania lewej części w nowym strumieniu.
    if (left < (high_ptr - data))
    {
        cudaStream_t stream_left;
        cudaStreamCreateWithFlags(&stream_left, cudaStreamNonBlocking);
        quicksortGPU <<<1, 1, 0, stream_left>>>(data, left, new_right, depth + 1);
        cudaStreamDestroy(stream_left);
    }

    // Uruchom nowy blok do sortowania prawej części w nowym strumieniu.
    if ((low_ptr - data) < right)
    {
        cudaStream_t stream_right;
        cudaStreamCreateWithFlags(&stream_right, cudaStreamNonBlocking);
        quicksortGPU <<<1, 1, 0, stream_right>>>(data, new_left, right, depth + 1);
        cudaStreamDestroy(stream_right);
    }
}
```

3.3 Algorytm Odd-Even Sort

3.3.1 CPU

Implementacja tego algorytmu na CPU wygląda następująco:

```
void oddEvenSortCPU(int* data, int size)
{
    bool sorted = false;
    while (!sorted)
    {
        sorted = true;

        // Sortowanie na parzystych indeksach
        for (int i = 0; i < size - 1; i += 2) {
            if (data[i] > data[i + 1]) {
                // Zamiana elementów
                int temp = data[i];
                data[i] = data[i + 1];
                data[i + 1] = temp;
                sorted = false;
            }
        }

        // Sortowanie na nieparzystych indeksach
        for (int i = 1; i < size - 1; i += 2)
        {
            if (data[i] > data[i + 1])
            {
                // Zamiana elementów
                int temp = data[i];
                data[i] = data[i + 1];
                data[i + 1] = temp;
                sorted = false;
            }
```

- Algorytm działa w pętli, w której iteruje się, dopóki tablica nie zostanie posortowana. Jeśli podczas jednej iteracji nie wystąpiły żadne zamiany elementów, algorytm uznaje tablicę za posortowaną i kończy działanie
- W każdej iteracji dwie pętle wykonują sortowanie na elementach o parzystych indeksach (pierwsza pętla) i na elementach o nieparzystych indeksach (druga pętla).
- Wewnątrz tych pętli sprawdzane są sąsiednie pary elementów, a jeśli element o niższym indeksie jest większy od elementu o wyższym indeksie, następuje ich zamiana.
- Algorytm kontynuuje iteracje, aż cała tablica zostanie posortowana.

3.3.2 GPU

Idea algorytmu pozostaje taka jak w podejściu na CPU. Funkcja implementuje algorytm sortowania parzysto-nieparzystego na GPU w sposób równoległy. Została też dodana pamięć współdzielona i operacje atomowe:

```
// Kernel funkcja do OddEvenSort
__global__ void oddEvenSortKernel(int* data, int size, int iteration, int* swapFlag)
{
    extern __shared__ int sharedData[]; // Deklaracja pamięci współdzielonej w kernelu

    // Globalny identyfikator wątku
    int tid = threadIdx.x + blockIdx.x * blockDim.x;

    // Sąsiedni wątek
    int tid2 = tid + 1;

    // Przechowuj dane w pamięci współdzielonej
    sharedData[tid] = (tid < size) ? data[tid] : INT_MAX;
    sharedData[tid2] = (tid2 < size) ? data[tid2] : INT_MAX;
    __syncthreads(); // Synchronizacja wątków

    // Sprawdź, czy obecna iteracja jest parzysta czy nieparzysta.
    if (tid2 < size)
    {
        if (iteration % 2 == 0)
        {
            if (tid % 2 == 0 && sharedData[tid] > sharedData[tid2])
            {
                // Zamiana elementów (parzyste indeksy) z operacją atomową
                int temp = sharedData[tid];
                atomicExch(&sharedData[tid], sharedData[tid2]);
                atomicExch(&sharedData[tid2], temp);
                atomicExch(swapFlag, 1); // Ustaw flagę na 1, jeśli dokonano zamiany
            }
        }
        else {
            if (tid % 2 == 1 && sharedData[tid] > sharedData[tid2])
            {
                // Zamiana elementów (nieparzyste indeksy) z operacją atomową
                int temp = sharedData[tid];
                atomicExch(&sharedData[tid], sharedData[tid2]);
                atomicExch(&sharedData[tid2], temp);
                atomicExch(swapFlag, 1); // Ustaw flagę na 1, jeśli dokonano zamiany
            }
        }
    }

    // Kopiuj zmienione dane z pamięci współdzielonej z powrotem do globalnej pamięci
    if (tid < size)
        data[tid] = sharedData[tid];
}
```

- W trakcie działania wykorzystuje pamięć współdzieloną (shared memory) w celu przechowywania tymczasowych danych podczas sortowania. Pamięć współdzielona jest znacznie szybsza niż globalna pamięć, co przyspiesza operacje odczytu i zapisu danych.
- Globalny identyfikator wątku (tid) jest obliczany na podstawie numeru wątku w bloku i indeksu bloku. Pozwala to na przetwarzanie różnych części tablicy w różnych wątkach.
- W funkcji wykonywane są dwie fazy sortowania: na parzystych i nieparzystych indeksach. To rozróżnienie jest kontrolowane przez iteration, co pozwala na przemienną pracę wątków w różnych fazach.
- W każdej fazie każdy wątek porównuje swoją parę elementów z sąsiednią parą i, jeśli zachodzi potrzeba, zamienia je miejscami. Zamiany te są wykonywane z użyciem operacji atomowych w celu uniknięcia konfliktów dostępu do pamięci współdzielonej.
- Po każdej fazie sortowania zmienione dane zostają skopiowane z pamięci współdzielonej z powrotem do globalnej pamięci.
- Wartość swapFlag jest używana do oznaczenia, czy w danej fazie sortowania dokonano jakiejkolwiek zamiany. Jeśli żadnej zamiany nie było, oznacza to, że tablica jest już posortowana, i algorytm może zakończyć działanie.

Wywołanie kernela w tym wypadku wygląda trochę inaczej więc opiszę to poniżej:

```
// Liczba iteracji (ograniczona do połowy rozmiaru danych)
int numIterations = dataSize / 2;

// Flaga do śledzenia zamiany elementów
int* devSwapFlag;
int swapFlag = 1;
```

```
for (int i = 0; i < numIterations && swapFlag == 1; ++i) {
    swapFlag = 0; // Resetowanie flagi przed każdą iteracją
    cudaMemcpy(devSwapFlag, &swapFlag, sizeof(int), cudaMemcpyHostToDevice);
    // Wywołanie kernela
    oddEvenSortKernel<<<numBlocks, threadsPerBlock, sizeof(int)* dataSize * 2 >>>(devDataOdd, dataSize, i, devSwapFlag);
    cudaMemcpy(&swapFlag, devSwapFlag, sizeof(int), cudaMemcpyDeviceToHost);
    cudaDeviceSynchronize(); // Synchronizacja po każdej iteracji
}
```

Obliczana jest liczba iteracji numIterations, która jest ograniczona do połowy rozmiaru danych (dataSize). Jest to związane z charakterystyką algorytmu sortowania parzysto-nieparzystego. Rozpoczyna się pętla, która będzie wykonywać sortowanie na GPU. Pętla działa dopóki zostanie wykonana zamiana (swapFlag == 1) lub zostanie wykonana maksymalna liczba iteracji. W wywołaniu kernela **‘i’** to numer aktualnej iteracji, który wpływa na to, czy sortowanie jest wykonywane na parzystych czy nieparzystych indeksach. Po każdej iteracji wywoływane jest cudaDeviceSynchronize(), co oznacza oczekiwanie na zakończenie pracy wszystkich wątków GPU

przed rozpoczęciem kolejnej iteracji. Jest to niezbędne, aby mieć pewność, że aktualizacje flagi swapFlag są widoczne na CPU po każdej iteracji.

3.4 Algorytm MergeSort

3.4.1 CPU

W celu implementacji tego algorytmu na CPU została stworzona funkcja scalająca merge i funkcja rekurencyjna mergeSort:

```
void merge(int* values, int* results, int l, int m, int r)
{
    int i, j, k;
    i = l;
    j = m + 1;
    k = l;

    // Pętla główna scalania podciągów.
    while (i <= m && j <= r)
    {
        // Jeśli wartość z lewego podciągu jest mniejsza lub równa, przekopiuj ją do wyników.
        if (values[i] <= values[j])
        {
            results[k] = values[i];
            i++;
        }
        // W przeciwnym razie przekopiuj wartość z prawego podciągu.
        else
        {
            results[k] = values[j];
            j++;
        }
        k++;
    }

    // Obsługa pozostałych elementów z lewego podciągu.
    while (i <= m)
    {
        results[k] = values[i];
        i++;
        k++;
    }

    // Obsługa pozostałych elementów z prawego podciągu.
    while (j <= r)
    {
        results[k] = values[j];
        j++;
        k++;
    }

    // Skopiowanie scalonych danych z tablicy wynikowej do tablicy źródłowej
    for (k = l; k <= r; k++)
    {
        values[k] = results[k];
    }
```

```
void mergeSort(int* values, int* results, int left, int right)
{
    if (left < right)
    {
        int middle = left + (right - left) / 2;

        // Wywołaj mergeSort na lewej i prawej połowie tablicy.
        mergeSort(values, results, left, middle);
        mergeSort(values, results, middle + 1, right);

        // Scal wyniki z lewej i prawej połowy
        merge(values, results, left, middle, right);
    }
}
```

- Algorytm sortowania przez scalanie jest algorytmem dziel i zwyciężaj, który polega na podziale tablicy na mniejsze podciągi, sortowaniu ich i scalaniu w jedną posortowaną tablicę.
- Funkcja merge jest odpowiedzialna za scalanie dwóch posortowanych podciągów w jedną posortowaną tablicę. Przechodzi przez oba podciągi, porównuje elementy i umieszcza je w odpowiedniej kolejności w tablicy wynikowej.
- Funkcja mergeSort jest rekurencyjną implementacją algorytmu sortowania przez scalanie. Dzieli tablicę na pół, wywołuje mergeSort na obu połowach, a następnie scalane wyniki z pomocą funkcji merge.

3.4.2 GPU

```cpp
// Sortowanie i scalanie podciagów na GPU
__device__ void CudaMerge(int* values, int* results, int l, int r, int u)
{
    // Zmienne i, j, k są indeksami do przechodzenia przez podciągi danych.
    int i, j, k;
    i = l; j = r; k = l;

    // Pętla główna scalania dwóch podciągów.
    while (i < r && j < u)
    {
        // Jeśli wartość z lewego podciągu jest mniejsza lub równa, zapisz ją i zwiększ indeksy.
        if (values[i] <= values[j])
        {
            atomicExch(&results[k], values[i]);
            i++;
        }
        // W przeciwnym razie zapisz wartość z prawego podciągu i zwiększ indeksy
        else
        {
            atomicExch(&results[k], values[j]);
            j++;
        }
        k++;
    }

    // Obsługa pozostałych elementów z lewego podciągu
    while (i < r)
    {
        atomicExch(&results[k], values[i]);
        i++;
        k++;
    }

    // Obsługa pozostałych elementów z prawego podciągu
    while (j < u)
    {
        atomicExch(&results[k], values[j]);
        j++;
        k++;
    }

    // Skopiowanie scalonych danych z tablicy wynikowej do tablicy źródłowej.
    for (k = l; k < u; k++)
    {
        values[k] = results[k];
    }
}
```

Opis funkcji CudaMerge (Sortowanie i scalanie na GPU):

- Ta funkcja jest uruchamiana na GPU jako część algorytmu sortowania przez scalanie..

- Otrzymuje jako argumenty wskaźniki do danych (values i results), zakres elementów do scalenia (indeksy l, r, u), gdzie l to początek pierwszego podciągu, r to koniec pierwszego podciągu, a u to koniec drugiego podciągu.
- Funkcja scalania działa na danych znajdujących się w pamięci globalnej na GPU.
- Pętla główna wykonuje porównania i scalanie dwóch podciągów. Wykorzystywane są operacje atomowe (atomicExch), aby uniknąć konfliktów dostępu do pamięci współdzielonej.
- Po scaleniu podciągów wyniki są zapisywane w tablicy results, a następnie skopiowane z powrotem do tablicy źródłowej values.

```
__global__ static void CudaMergeSort(int* values, int* results, int dim)
{
    // Współdzielona pamięć na blok wątków CUDA
    extern __shared__ int shared[];

    // Pobranie identyfikatora wątku w bloku.
    const unsigned int tid = threadIdx.x;

    // k - krok sortowania, i - do iteracji po danych,
    // u - pozycja, gdzie zakończy się bieżący podciąg
    int k, u, i;

    // Kopiowanie z tablicy values do współdzielonej pamięci.
    shared[tid] = values[tid];

    // Synchronizacja wątków w bloku, aby upewnić się, że wszystkie wątki skopiowały dane.
    __syncthreads();

    k = 1;
    while (k <= dim)
    {
        i = 0;
        while (i + k < dim)
        {
            u = i + k * 2;;
            if (u > dim)
            {
                u = dim + 1; // Jeśli u wykracza poza wymiar, ustawienie go na końcówkę tablicy.
            }
            // Wywołanie funkcji CudaMerge, aby posortować i scalć podciągi.
            CudaMerge(shared, results, i, i + k, u);

            // Przesuniecie do nastepnego podciagu
            i = i + k * 2;
        }

        // Zwiększamy krok sortowania dwukrotnie.
        k = k * 2;

        // Synchronizacja wątków w bloku, aby upewnić się, że wszystkie operacje są zakończone
        __syncthreads();
    }

    // Pobranie danych z wspoldzielonej pamieci
    values[tid] = shared[tid];
}
```

Opis funkcji CudaMergeSort (Kernel do MergeSort):

- Ta funkcja jest uruchamiana jako kernel na GPU i jest główną częścią algorytmu Merge Sort na GPU.

- Argumenty funkcji to wskaźnik do danych (values), wskaźnik do tablicy wynikowej (results) i rozmiar danych (dim).
- Współdzielona pamięć w bloku jest wykorzystywana do przechowywania części danych.
- Algorytm Merge Sort jest realizowany w pętli. Na każdym kroku pętli następuje sortowanie i scalanie podciągów o odpowiednich rozmiarach, a krok sortowania (k) jest dwukrotnie zwiększany.
- Funkcja CudaMerge jest wywoływana wewnątrz pętli, aby dokonać scalenia podciągów.
- Synchronizacje __syncthreads() są używane w celu zapewnienia poprawnej synchronizacji wątków w bloku.
- Po zakończeniu sortowania, dane są zapisywane w tablicy values, która zawiera już posortowane elementy.

3.5 Thrust :: sort

Poniżej jest przedstawione jak wygląda sortowanie z użyciem biblioteki Thrust:

```
// Alokacja pamięci na GPU za pomocą kontenera Thrust
thrust::device_vector<int> devDataThrust(dataSize);

// Kopiowanie danych z pamięci CPU do pamięci GPU za pomocą Thrust
thrust::copy(inputData, inputData + dataSize, devDataThrust.begin());

// Sortowanie danych na GPU za pomocą thrust::sort
thrust::sort(devDataThrust.begin(), devDataThrust.end());

// Kopiowanie wyników z pamięci GPU do pamięci CPU za pomocą Thrust
int* resultThrust = new int[dataSize];
thrust::copy(devDataThrust.begin(), devDataThrust.end(), resultThrust);
```

3.6 Bloki i wątki

Aby zagwarantować, że liczba wątków w bloku jest wielokrotnością warp (czyli ilości jednostek logiczno-arytmetycznych w multi-procesorze), najpierw obliczymy liczbę wątków w warpie i następnie dostosujemy liczbę wątków w bloku do tej wielokrotności. Moja karta graficzna GeForce GTX 960 jest starszą kartą graficzną od NVIDIA, która korzysta z architektury Maxwell. Wartość warp (liczba wątków w warpie) na tej architekturze wynosi również 32, co jest powszechne w wielu starszych i nowszych GPU od NVIDIA. Dostosowanie tych zmiennych wygląda następująco:

```
int blockSize = 32; // Wartość warp na architekturze Maxwell

// Oblicz liczbę warpów potrzebnych do przetworzenia wszystkich danych
int numWarps = (dataSize + blockSize - 1) / blockSize;

// Oblicz liczbę wątków w bloku, tak aby była wielokrotnością warp
int threadsPerBlock = numWarps * blockSize;

// Upewnij się, że liczba wątków w bloku nie przekracza maksymalnej wartości
int maxThreadsPerBlock = 1024; // Maksymalna liczba wątków w bloku na GeForce GTX 960

if (threadsPerBlock > maxThreadsPerBlock) {
    threadsPerBlock = maxThreadsPerBlock;
}

// Ustal liczbę bloków na podstawie rozmiaru danych i threadsPerBlock
int numBlocks = (dataSize + threadsPerBlock - 1) / threadsPerBlock;
```

3.7 Pomiar czasu

Czas wykonania algorytmu na CPU mierzymy w następujący sposób:

```
clock_t cpuStartMS = clock();
mergeSort(dataMS, resultsMS, 0, dataSize - 1);
clock_t cpuEndMS = clock();
double cpuTimeMS = static_cast<double>(cpuEndMS - cpuStartMS) / CLOCKS_PER_SEC;
```

Natomiast na GPU używamy cudaEvent_t oraz cudaEventElapsedTime i wygląda to tak:

```
// Pomiar czasu na GPU za pomocą cudaEvent_t
cudaEvent_t startQS, stopQS;
cudaEventCreate(&startQS);
cudaEventCreate(&stopQS);
cudaEventRecord(startQS);

// Wywołanie kernela
quicksortGPU<<<numBlocks, threadsPerBlock, sizeof(int)* dataSize * 2 >>>(d_data, left, right, 0);

cudaEventRecord(stopQS);
cudaEventSynchronize(stopQS);
float gpuTimeQS = 0;
cudaEventElapsedTime(&gpuTimeQS, startQS, stopQS);
```

```
cout << "Czas sortowania na GPU:    " << fixed << gpuTimeQS / 1000.0 << " sekundy" << endl << endl;
```

3.8 Alokacja, kopiowanie i wywoływanie

Przy każdym sortowaniu używamy tych samych funkcji CUDA i wygląda to następująco (tu przykładowo wywołanie quicksort):

```
int left = 0;
int right = dataSize - 1;

// Alokacja pamięci na GPU dla danych wejściowych
int* d_data;
cudaMalloc((void**)&d_data, dataSize * sizeof(int));

// Kopiowanie danych z CPU na GPU
cudaMemcpy(d_data, inputData, dataSize * sizeof(int), cudaMemcpyHostToDevice);

// Pomiar czasu na GPU za pomocą cudaEvent_t
cudaEvent_t startQS, stopQS;
cudaEventCreate(&startQS);
cudaEventCreate(&stopQS);
cudaEventRecord(startQS);

// Wywołanie kernela
quicksortGPU<<<numBlocks, threadsPerBlock, sizeof(int)* dataSize * 2 >>>(d_data, left, right, 0);

cudaEventRecord(stopQS);
cudaEventSynchronize(stopQS);
float gpuTimeQS = 0;
cudaEventElapsedTime(&gpuTimeQS, startQS, stopQS);

// Kopiowanie danych z GPU na CPU
int* resultQS = new int[dataSize];
cudaMemcpy(resultQS, d_data, dataSize * sizeof(int), cudaMemcpyDeviceToHost);
```

4 Analiza rozwiązania

4.1 Losowe dane

```
Ilosc danych typu INT: 200000
Liczba blokow na GPU:  196
Liczba watkow na blok: 1024

-----------QUICKSORT-----------
Poprawnosc algorytmu na CPU:   CORRECT
Poprawnosc algorytmu na GPU:   CORRECT
Czas sortowania na CPU:    0.046 sekundy
Czas sortowania na GPU:    0.000002 sekundy

-----------ODD-EVEN SORT-----------
Poprawnosc algorytmu na CPU:   CORRECT
Poprawnosc algorytmu na GPU:   CORRECT
Czas sortowania na CPU:    0.001000 sekundy
Czas sortowania na GPU:    0.000148 sekundy

-----------MERGESORT-----------
Poprawnosc algorytmu na CPU:   CORRECT
Poprawnosc algorytmu na GPU:   CORRECT
Czas sortowania na CPU:    0.022000 sekundy
Czas sortowania na GPU:    0.000002 sekundy

-----------THRUST::SORT-----------
Poprawnosc algorytmu:          CORRECT
Czas sortowania:   0.074551 sekundy
```

```
Ilosc danych typu INT: 2000000
Liczba blokow na GPU:  1954
Liczba watkow na blok: 1024

-----------QUICKSORT-----------
Poprawnosc algorytmu na CPU:   CORRECT
Poprawnosc algorytmu na GPU:   CORRECT
Czas sortowania na CPU:    0.509 sekundy
Czas sortowania na GPU:    0.000002 sekundy

-----------ODD-EVEN SORT-----------
Poprawnosc algorytmu na CPU:   CORRECT
Poprawnosc algorytmu na GPU:   CORRECT
Czas sortowania na CPU:    0.005000 sekundy
Czas sortowania na GPU:    0.000088 sekundy

-----------MERGESORT-----------
Poprawnosc algorytmu na CPU:   CORRECT
Poprawnosc algorytmu na GPU:   CORRECT
Czas sortowania na CPU:    0.258000 sekundy
Czas sortowania na GPU:    0.000002 sekundy

-----------THRUST::SORT-----------
Poprawnosc algorytmu:          CORRECT
Czas sortowania:   0.248862 sekundy
```

```
Ilosc danych typu INT: 20000000
Liczba blokow na GPU:  19532
Liczba watkow na blok: 1024

-----------QUICKSORT-----------
Poprawnosc algorytmu na CPU:   CORRECT
Poprawnosc algorytmu na GPU:   CORRECT
Czas sortowania na CPU:    5.429 sekundy
Czas sortowania na GPU:    0.000002 sekundy

-----------ODD-EVEN SORT-----------
Poprawnosc algorytmu na CPU:   CORRECT
Poprawnosc algorytmu na GPU:   CORRECT
Czas sortowania na CPU:    0.049000 sekundy
Czas sortowania na GPU:    0.000303 sekundy

-----------MERGESORT-----------
Poprawnosc algorytmu na CPU:   CORRECT
Poprawnosc algorytmu na GPU:   CORRECT
Czas sortowania na CPU:    2.948000 sekundy
Czas sortowania na GPU:    0.000002 sekundy

-----------THRUST::SORT-----------
Poprawnosc algorytmu:          CORRECT
Czas sortowania:   2.841661 sekundy
```

```
Ilosc danych typu INT: 50000000
Liczba blokow na GPU:  48829
Liczba watkow na blok: 1024

-----------QUICKSORT-----------
Poprawnosc algorytmu na CPU:   CORRECT
Poprawnosc algorytmu na GPU:   CORRECT
Czas sortowania na CPU:    14.325 sekundy
Czas sortowania na GPU:    0.000002 sekundy

-----------ODD-EVEN SORT-----------
Poprawnosc algorytmu na CPU:   CORRECT
Poprawnosc algorytmu na GPU:   CORRECT
Czas sortowania na CPU:    0.122000 sekundy
Czas sortowania na GPU:    0.000122 sekundy

-----------MERGESORT-----------
Poprawnosc algorytmu na CPU:   CORRECT
Poprawnosc algorytmu na GPU:   CORRECT
Czas sortowania na CPU:    7.661000 sekundy
Czas sortowania na GPU:    0.000002 sekundy
```

4.2 Dane posortowane

```
Ilosc danych typu INT: 200000
Liczba blokow na GPU:  196
Liczba watkow na blok: 1024

-----------QUICKSORT-----------
Poprawnosc algorytmu na CPU:   CORRECT
Poprawnosc algorytmu na GPU:   CORRECT
Czas sortowania na CPU:    0.019 sekundy
Czas sortowania na GPU:    0.000002 sekundy

-----------ODD-EVEN SORT-----------
Poprawnosc algorytmu na CPU:   CORRECT
Poprawnosc algorytmu na GPU:   CORRECT
Czas sortowania na CPU:    0.001000 sekundy
Czas sortowania na GPU:    0.000123 sekundy

-----------MERGESORT-----------
Poprawnosc algorytmu na CPU:   CORRECT
Poprawnosc algorytmu na GPU:   CORRECT
Czas sortowania na CPU:    0.032000 sekundy
Czas sortowania na GPU:    0.000002 sekundy

-----------THRUST::SORT-----------
Poprawnosc algorytmu:          CORRECT
Czas sortowania:   0.073893 sekundy
```

```
Ilosc danych typu INT: 2000000
Liczba blokow na GPU:  1954
Liczba watkow na blok: 1024

-----------QUICKSORT-----------
Poprawnosc algorytmu na CPU:   CORRECT
Poprawnosc algorytmu na GPU:   CORRECT
Czas sortowania na CPU:    0.281 sekundy
Czas sortowania na GPU:    0.000002 sekundy

-----------ODD-EVEN SORT-----------
Poprawnosc algorytmu na CPU:   CORRECT
Poprawnosc algorytmu na GPU:   CORRECT
Czas sortowania na CPU:    0.004000 sekundy
Czas sortowania na GPU:    0.000094 sekundy

-----------MERGESORT-----------
Poprawnosc algorytmu na CPU:   CORRECT
Poprawnosc algorytmu na GPU:   CORRECT
Czas sortowania na CPU:    0.362000 sekundy
Czas sortowania na GPU:    0.000002 sekundy

-----------THRUST::SORT-----------
Poprawnosc algorytmu:          CORRECT
Czas sortowania:   0.263767 sekundy
```

```
Ilosc danych typu INT: 20000000
Liczba blokow na GPU:  19532
Liczba watkow na blok: 1024

-----------QUICKSORT-----------
Poprawnosc algorytmu na CPU:   CORRECT
Poprawnosc algorytmu na GPU:   CORRECT
Czas sortowania na CPU:    2.971 sekundy
Czas sortowania na GPU:    0.000002 sekundy

-----------ODD-EVEN SORT-----------
Poprawnosc algorytmu na CPU:   CORRECT
Poprawnosc algorytmu na GPU:   CORRECT
Czas sortowania na CPU:    0.050000 sekundy
Czas sortowania na GPU:    0.000114 sekundy

-----------MERGESORT-----------
Poprawnosc algorytmu na CPU:   CORRECT
Poprawnosc algorytmu na GPU:   CORRECT
Czas sortowania na CPU:    3.028000 sekundy
Czas sortowania na GPU:    0.000002 sekundy

-----------THRUST::SORT-----------
Poprawnosc algorytmu:          CORRECT
Czas sortowania:   2.848074 sekundy
```

```
Ilosc danych typu INT: 50000000
Liczba blokow na GPU:  48829
Liczba watkow na blok: 1024

-----------QUICKSORT-----------
Poprawnosc algorytmu na CPU:   CORRECT
Poprawnosc algorytmu na GPU:   CORRECT
Czas sortowania na CPU:    8.93 sekundy
Czas sortowania na GPU:    0.000002 sekundy

-----------ODD-EVEN SORT-----------
Poprawnosc algorytmu na CPU:   CORRECT
Poprawnosc algorytmu na GPU:   CORRECT
Czas sortowania na CPU:    0.133000 sekundy
Czas sortowania na GPU:    0.000175 sekundy

-----------MERGESORT-----------
Poprawnosc algorytmu na CPU:   CORRECT
Poprawnosc algorytmu na GPU:   CORRECT
Czas sortowania na CPU:    8.271000 sekundy
Czas sortowania na GPU:    0.000002 sekundy
```