Sortowanie z wykorzystaniem OpenMP oraz CUDA

Dawid Bachoń

Wstęp

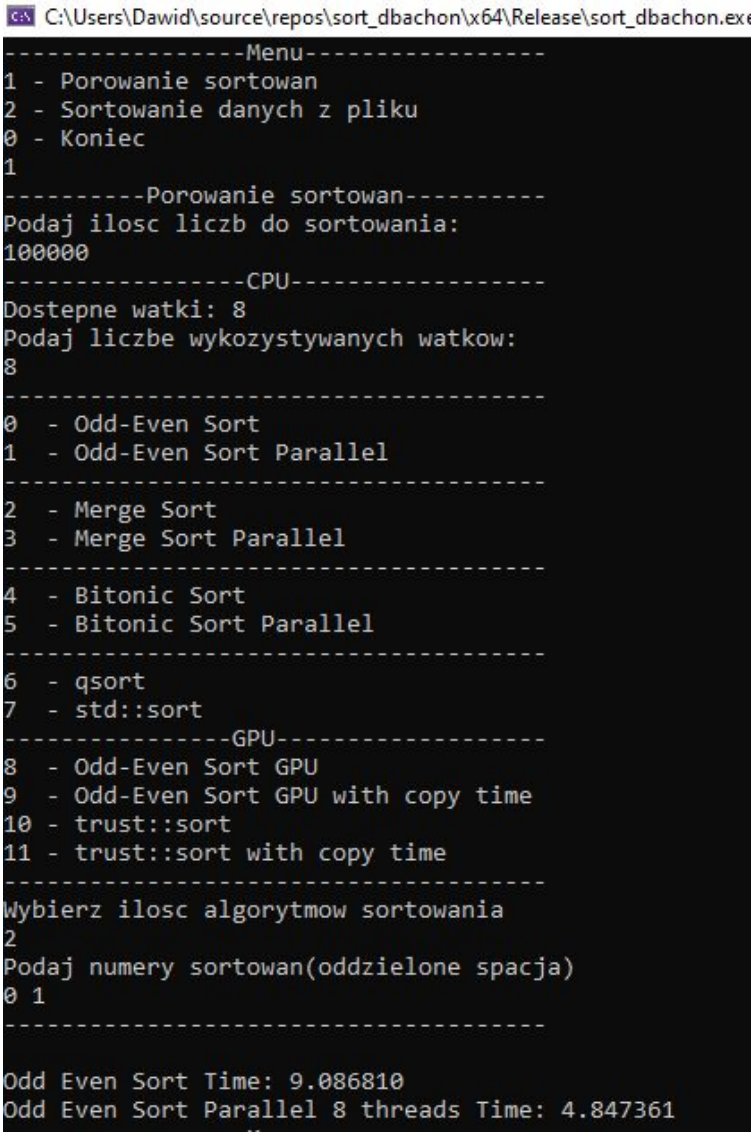
Projekt ma na celu przybliżenie oraz porównanie wybranych metod sortowania. Wyznaczenia umownych granic przy jakich uznamy że dany algorytm jest nieefektywny do wykorzystania w celu rozwiązania naszego problemu, posortowania tablicy liczb całkowitych..

Projekt posiada 2 podstawowe funkcje:

- Porównanie wybranych metod sortowania
- Pobranie liczb z pliku, posortowanie ich wybraną metodą oraz zapis wyniku do pliku.

Opis

Pierwsza funkcjonalność programu skupia się na porównaniu wybranych metod sortowania. Pozwala nam ona wybrać ilość liczb całkowitych do posortowania, liczbę wątków oraz algorytmy do wykorzystania. Na potrzeby porównań generowana jest tablica liczb pseudolosowych o stałym ziarnie oraz zadanym przez użytkownika rozmiarze.



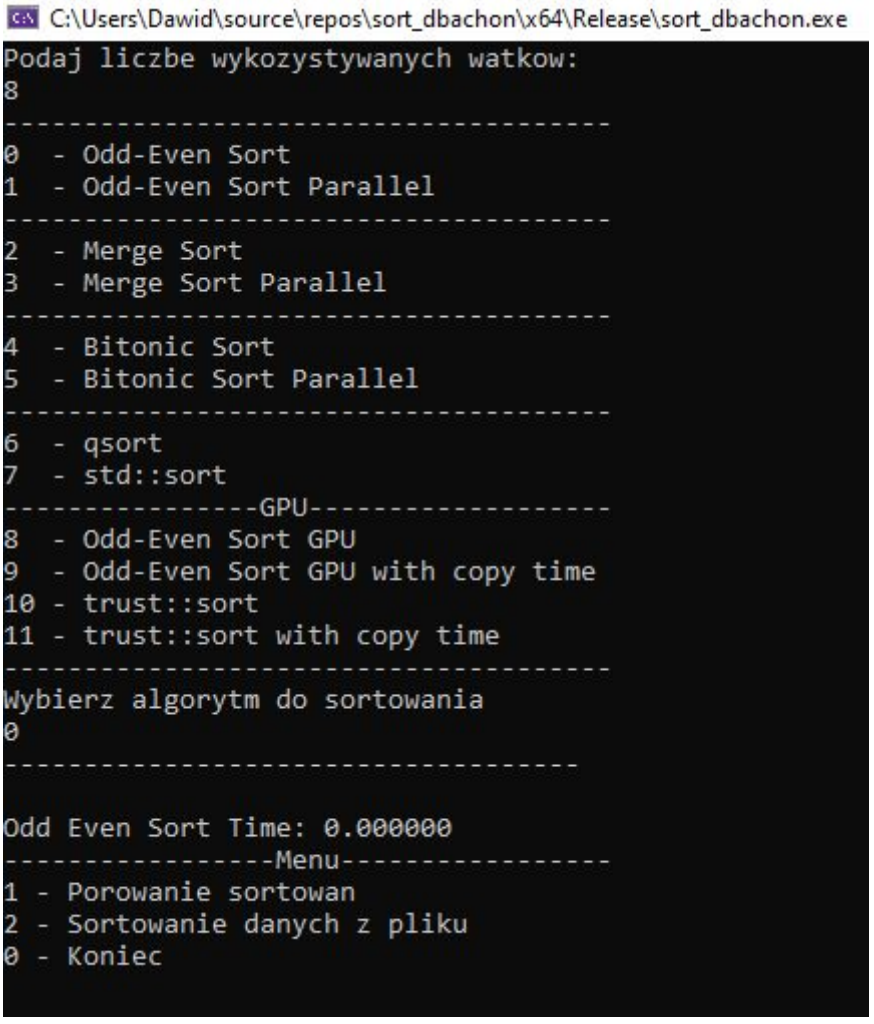
Jak widać na powyższym zdjęciu mamy do wyboru 3 algorytmy wykorzystane z bibliotek. W tym 2 na CPU oraz 1 na GPU.

Z własnych implementacji znajdziemy

- Odd-Even Sort w implementacji jedno, wielowątkowej oraz na GPU.
- Merge Sort w implementacji jedno oraz wielowątkowej

- Bitonic Sort rekurencyjny w implementacji jedno oraz wielowątkowej

Druga funkcjonalność pozwala posortować dane z pliku oraz zapisać je do odpowiedniego pliku. Oczywiście program zapewnia też sprawdzenie czy liczby zostały odpowiednio posortowane.



Odd-Even Sort

Odd-Even Sort jest to algorytmem sortowania o pesymistycznej złożoności $O(n^2)$.

Jest on uznawany za odmianę sortowania bąbelkowego. Jego zaletą jest to żę nie w danej iteracji porównania elementów w danej pętli nie zachodzą na siebie.

Zrównoleglenie tego algorytmu jest bardzo proste gdyż nie musimy martwić się o synchronizację w czasie wykonywania wewnętrznych pętli. porównania elementów odbywają się w sposób niezależny. Aby zrównoleglić wykonywanie pętli użyłam klauzuli **#pragma omp for .** Ilość wątków ustawiam wykorzystując klauzulę num_threads().

```
void oddEvenSort(int32_t* array, const size_t& size) {
    for (size_t j = 0; j < size; j++) {
        if (j & 1) {
            for (size_t i = 2; i < size; i += 2)
                if (array[i - 1] > array[i])
                    std::swap(array[i - 1], array[i]);
        }
        else {
            for (size_t i = 1; i < size; i += 2)
                if (array[i - 1] > array[i])
                    std::swap(array[i - 1], array[i]);
        }
    }
}

void oddEvenSortParallel(int32_t* array, const size_t& size, const int& threads){
    long i, j;
    int chunk = size < 1000 ? 10 : 100;
    #pragma omp parallel private(j) num_threads(threads)
    {
        for (j = 0; j < size; j++) {
            if (j & 1) {
                #pragma omp for schedule(guided, chunk)
                for (i = 2; i < size; i += 2)
                    if (array[i - 1] > array[i])
                        std::swap(array[i - 1], array[i]);
            }
            else {
                #pragma omp for schedule(guided, chunk)
                for (i = 1; i < size; i += 2)
                    if (array[i - 1] > array[i])
                        std::swap(array[i - 1], array[i]);
            }
        }
    }
}
```

Odd-Even Sort GPU

Odd-Even Sort na GPU - aby zrównoleglić algorytm podzieliłem sortowania wewnętrzną pętle sortowania z klasycznego algorytmu na bloki o rozmiarze 1024 (na tylu wątkach będzie uruchomiony algorytm), numer elementów do porównania odczytujemy z numeru_bloku*rozmiar bloku + numer_wątku_w_bloku, oraz parzystości iteracji głównej pętli. Należy mieć na uwadzę że mimo znaczącej poprawy wydajności względem wersji na CPU nadal mamy do czynienia z algorytmem o złożoności kwadratowej co dla dużych rozmiarów tablicy będzie generowało bardzo długi czas oczekiwania na rezultat. Kolejnym ograniczeniem jest rozmiar tablicy aby algorytm działał optymalnie, dane powinny w całości zmieścić w pamięci karty graficznej.

```
double oddEvenSortGPUTime = benchmark([&]() {

    for (size_t i = 0; i < size; i++) {
        oddEvenSort_GPU << < (size / 2 + 1023) / 1024, 1024 >> > (arrayOnGPU, i % 2, size);
    }

    });
```

```
template <typename T>
__device__ void inline swap(T& x, T& y) {
    T z(x); x = y; y = z;
}
__global__ void oddEvenSort_GPU(int32_t* arrayOnGPU, const unsigned int parity, const size_t size) {
    const unsigned int index = blockIdx.x * blockDim.x + threadIdx.x;
    if (parity && index * 2 + 2 < size) {
        if (arrayOnGPU[index * 2 + 1] > arrayOnGPU[index * 2 + 2]) {
            swap(arrayOnGPU[index * 2 + 1], arrayOnGPU[index * 2 + 2]);
        }
    } else if (!parity && index * 2 + 1 < size) {
        if (arrayOnGPU[index * 2] > arrayOnGPU[index * 2 + 1]) {
            swap(arrayOnGPU[index * 2], arrayOnGPU[index * 2 + 1]);
        }
    }
}
```

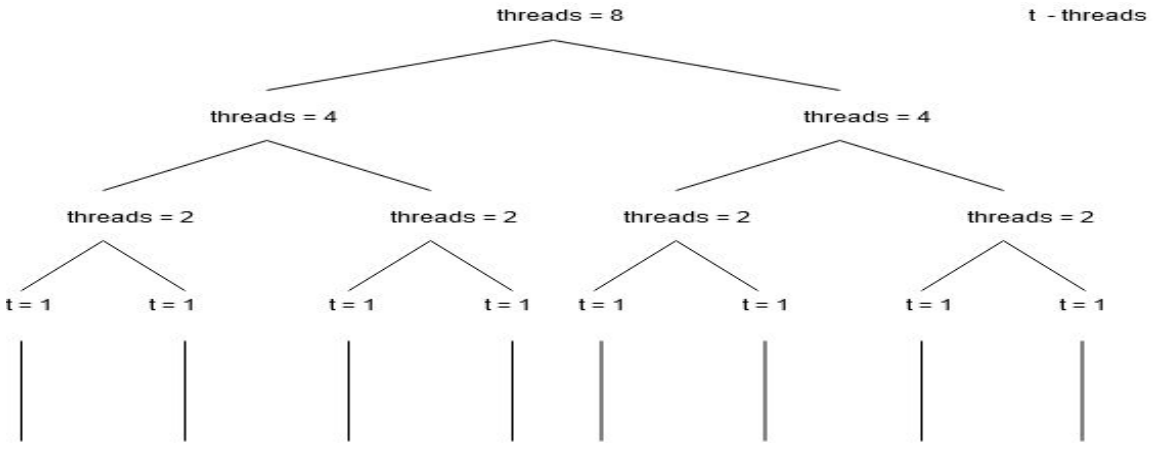
Merge Sort

MergeSort jest to algorytmem sortowania o pesymistycznej złożoności O(n*log n).

Zrównoleglenie tego algorytmu nie jest już tak proste jak Odd-Even Sort. Na danej głębokości rekurencji nie jesteśmy wstanie zrównoleglić “łączenia” dwóch posortowanych fragmentów.

Zastosowałem więc mechanizm zrównoleglenia pierwszych $\log_2$ ilość_wątków głębokości rekurencji. Dla 8 wątków pierwsze 3 wywołania rekurencji zostaną zrównoleglone.

W momencie gdy 8 wątków będzie pracować, kolejne podziały i łączenia tablicy będą już wykonywane przez klasyczny algorytm. Aby zrównoleglenie działało optymalnie liczba wątków powinna być całkowitą wielokrotnością liczby 2. Musimy też zadbać by tymczasowa tablica wykorzystana przy scalaniu była wykorzystywana na tym samym przedziale co oryginalna. Zapobiegnie to możliwości współzawodnictwa o dany element tablicy. Dodatkowo aby alokowanie tablicy nie spowalniało sortowania, tablica jest zadeklarowana statycznie(może być też dynamicznie aczkolwiek raz przed wykonaniem algorytmu, a nie w każdym wywołaniu funkcji scalania gdyż narzut czasu na alokowanie pamięci znacząco spowalnia algorytm) przed sortowaniem, o rozmiarze identycznym jak tablica liczb do posortowania. Zrównoleglania dokonałem przez uruchomienie zagnieżdżonej równoległości z wykorzystaniem mechanizmu sekcji oraz włączyłem zagnieżdżoną równoległość wykonując funkcje **omp_set_nested(1);** Tak dla ustawienia 8 wątków przedstawia się zaimplementowany wielowątkowy algorytm Merge Sort.



```cpp
void mergeSort(int32_t* array, const size_t& left, const size_t& right, int32_t* tempArray) {
    if (left < right) {
        const size_t center = (left + right) / 2;
        mergeSort(array, left, center, tempArray);
        mergeSort(array, center + 1, right, tempArray);
        mergeParallel(array, left, center, right, tempArray);
    }
}

void mergeSortParallel(int32_t* array, const size_t& left, const size_t& right, const int& threads, int32_t* tempArray) {
    if (threads == 1) {
        mergeSort(array, left, right, tempArray);
    }
    else if (threads > 1) {
        if (left < right) {
            const size_t center = (left + right) / 2;
            #pragma omp parallel sections num_threads(2)
            {
                #pragma omp section
                mergeSortParallel(array, left, center, threads / 2, tempArray);
                #pragma omp section
                mergeSortParallel(array, center + 1, right, threads - (threads / 2), tempArray);
            }
            mergeParallel(array, left, center, right, tempArray);
        }
    }
}
```

```cpp
inline void mergeParallel(int32_t* array, const size_t& left, const size_t& center, const size_t& right, int32_t* tempArray)
    size_t i = left;
    size_t j = center + 1;
    size_t current = left;

    while (i <= center && j <= right) {
        if (array[j] < array[i]) {
            tempArray[current++] = array[j++];
        }
        else {
            tempArray[current++] = array[i++];
        }
    }
    if (i <= center) {
        while (i <= center) {
            tempArray[current++] = array[i++];
        }
    }
    else {
        while (j <= right) {
            tempArray[current++] = array[j++];
        }
    }
    memcpy(array + left, tempArray + left, (right - left + 1) * sizeof(int32_t));
}
```

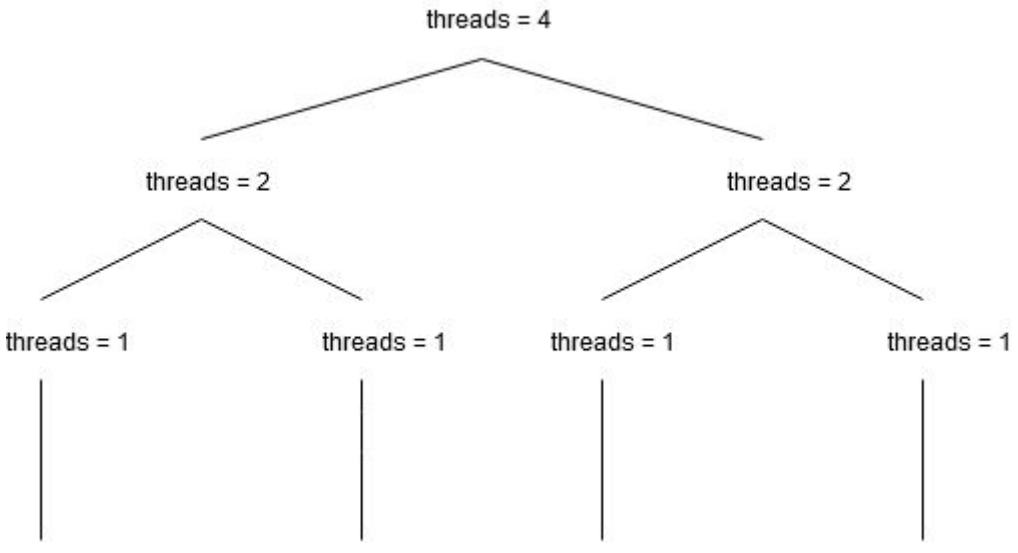
Bitonic Sort

Bitonic Sort jest to algorytmem sortowania o pesymistycznej złożoności O(n*log n).

W wersji rekurencyjnej jest trochę podobny do Merge Sort - zastosowanie metody dziel i zwyciężaj. Zasada działania jest w teorii dość prosta zaczynając od wielkości tablicy 1 tworzymy coraz dłuższe ciągi bitoniczne (ciąg niemalejący do pewnego elementu a od tego elementu ciąg nierosnący (lub odwrotnie)), w chwili gdy długość ciągu niemalejącego odpowiada długości tablicy kończymy algorytm. Zrównoleglania dokonałem w bliźniaczy sposób do merge sort, kod algorytmu jest bardzo podobny, główną różnice zobaczymy w funkcji scalania. Zasadniczo Bitonic Sort przeznaczony jest do sortowania elementów o rozmiarze 2^N, aby działał dla dowolnej ilości elementów zastosowałem pewną modyfikację, a mianowicie dzielę tablicę na 2 części, pierwsza zawsze o najwyższej potędze 2 mniejszej od rozmiaru, druga to rozmiar fragmentu - rozmiar pierwszej części fragmentu. Więc algorytm działa najszybciej dla danych o rozmiarze 2^N, ale kosztem odrobiny wydajności algorytm możemy uruchomić dla dowolnego rozmiaru danych wejściowych.

Zrównoleglania dokonałem przy użyciu sekcji, oraz zagnieżdżonej równoległości.

Gdy algorytm zużyje pule wątków, kolejne wywołania rekurencji będą wykonywane sekwencyjnie, dla 4 wątków(threads oznacza liczbę dostępnych wątków dla danego wywołania funkcji **bitonicSortParallel**) prezentuj się to tak:



Wydajność Bitonic Sort w wersji rekurencyjnej jest mniejsza od wersji iteracyjnej, spowodowane jest to przez narzut czasu potrzebny do wywołań rekurencyjnych.

W zastosowanym przeze mnie algorytmie dodatkowe lekkie spowolnienie powoduje funkcje wyznaczająca punkt podziału, nie zmienia to jednak faktu że algorytm posiada złożoność liniowo-logarytmiczną.

```
void bitonicMerge(int32_t* array, const size_t& start, const size_t& size, bool dir) {
    if (size > 1) {
        size_t center = lessPower2Than(size);
        for (size_t i = start; i < start + size - center; i++)
            if (dir == (array[i] > array[i + center]))
                std::swap(array[i], array[i + center]);
        bitonicMerge(array, start, center, dir);
        bitonicMerge(array, start + center, size - center, dir);
    }
}
```

```
void bitonicSortParallel(int32_t* array, const size_t& start, const size_t& size, bool dir, const int& threads) {
    if (threads == 1) {
        bitonicSort(array, start, size, dir);
    }
    else {
        if (size > 1) {
            size_t center = lessPower2Than(size);
            #pragma omp parallel sections num_threads(2)
            {
            #pragma omp section
                bitonicSortParallel(array, start, center, !dir, threads / 2);
            #pragma omp section
                bitonicSortParallel(array, start + center, size - center, dir, threads - threads / 2);
            }
            bitonicMerge(array, start, size, dir);
        }
    }
}

void bitonicSort(int32_t* array, const size_t& start, const size_t& size, bool dir) {
    if (size > 1) {
        size_t center = lessPower2Than(size);
        bitonicSort(array, start, center, !dir);
        bitonicSort(array, start + center, size - center, dir);
        bitonicMerge(array, start, size, dir);
    }
}
```

```
size_t lessPower2Than(const size_t& size) {
    size_t k = 1;
    while (k > 0 && k < size)
        k = k << 1;
    return k >> 1;
}
```

Obserwacje

Odd-Even Sort ma niewielki przyrost wydajności przy zwiększaniu liczby wykorzystywanych wątków na CPU, przy zmianie z 2 na 4 wykorzystane wątki przyspieszenie nie przekracza 20%. Prawdopodobnie jest to spowodowane dostępem do pamięci.

Odd-Even Sort z dynamicznym podziałem iteracji działa zdecydowanie wolniej od wersji ze statycznym podziałem oraz z “guided”. Dla odpowiednio dużego rozmiaru tablicy czas wykonania przekracza dziesięciokrotnie czas wykonania wersji iteracyjnej, stąd zrezygnowałem z dynamicznego przydzielania iteracji na rzecz “guided”

Odd-Even Sort na GPU ma prawie liniowy czas wykonania do momentu gdy rozmiar danych nie przekracza ilości dostępnych wątków.

Odd-Even Sort przez złożoność czasową nie pozwolił na przetestowanie w domowych warunkach dla danych powyżej 1 000 000, zwiększenie danych 10 powoduje wydłużenie czasu 100 krotnie wiec nawet wersja na gpu dla 10 000 000 przekroczyłaby 4500s ~75 min. W około 75 min w wersji na GPU stosunku do niecałej sekundy dla jednowątkowej wersji Merge Sort, uwydatnia wady kwadratowych algorytmów sortowania.

Merge Sort jest szybszy od rekurencyjnego Bitonic Sort, jednakże na wykresach logarytmicznych widać że posiadają tą samą klasę złożoności O(nlog)

Bitonic Sort w wersji równoległej traci na wydajności gdy dane nie są dzielone blisko połowy, cześć procesorów dostaje mało danych, jednakże pozostałe uzyskują wyższe taktowanie.

Merge Sort oraz Bitonic Sort uruchamiają teoretycznie więcej niż podana liczba wątków, przy 1 wywołaniu rekurencji 2, te w wątki kolejne 2 wiec teoretycznie podanych 4 wątków uruchamia się 6, ale w praktyce jednocześnie pracują maksymalnie 4 wątki.

Mimo posiadania 8 logicznych wątków przyspieszenie w algorytmach przy zmianie z 4 na 8 nie przekracza 10%, widoczne jest zmniejszenie taktowania dla utrzymania odpowiedniej temperatury procesora.

Trust::sort jest najwydajniejszym z testowanych algorytmów, ograniczeniem dla niego okazał się nie czas lecz pamięć karty graficznej.

Podsumowanie

Testy programu były wykonane na laptopie z procesorem **Intel® Core™ i7-7700HQ**, kartą graficzną **Nvidia GeForce GTX 1060 6GB** oraz z **32GB pamięci RAM.**

Wzrost używanych procesorów nie powoduje aż takich przyrostów szybkości algorytmów jakie można było się spodziewać w teorii.

Spowodowane jest to wieloma czynnikami takimi:

- ilość fizycznych dostępnych rdzeni to 4 ale procesor posiada 8 wątków logicznych
- zwiększenie ilości używanych rdzeni powoduje obniżenie częstotliwości pracy pojedynczego rdzenia (1 pracujący rdzeń osiąga częstotliwość do 3.80 GHz a przy 4 pracujące rdzeniach częstotliwość dla pojedynczego rdzenia to już tylko do 2.80 GHz)
- czas niezrównoleglonych fragmentów algorytmu zaczyna odgrywać większą rolę w stosunku do czasu całego algorytmu, przy zastosowaniu rozwiązań wielowątkowych.

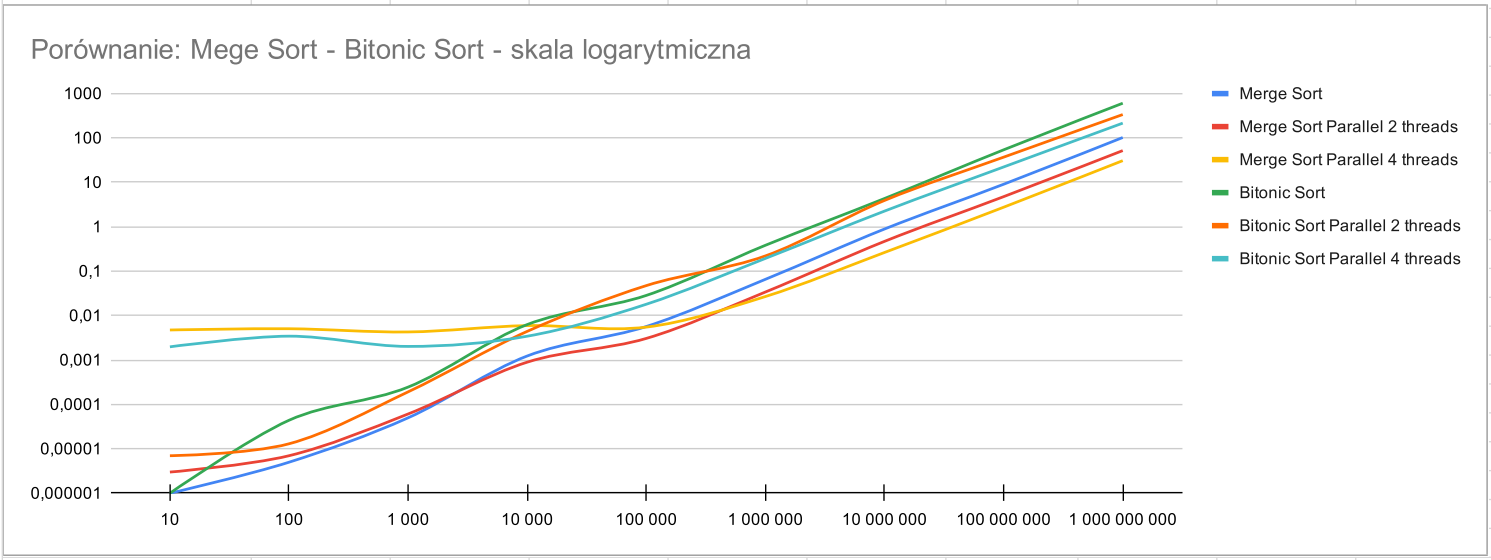
Wykorzystanie gpu do sortowania dla odpowiednich rozmiarów danych wejsciowych pozwala osiągnoć znaczne przyspieszenie działania algorytmu. Jednak wykorzystanie GPU rodzi też pewne komplikacje związane z kopiowaniem danych na urządzenie oraz z pamięcią samego urządzenia - gdy rozmiar danych przekracza rozmiar pamięci karty trzeba zadbać o odpowiednią synchronizację danych oraz zastosować dodatkowy algorytm do zarządzania danymi wysyłanymi, pobieranymi z pamięci urządzenia oraz scalania otrzymanych wyników.

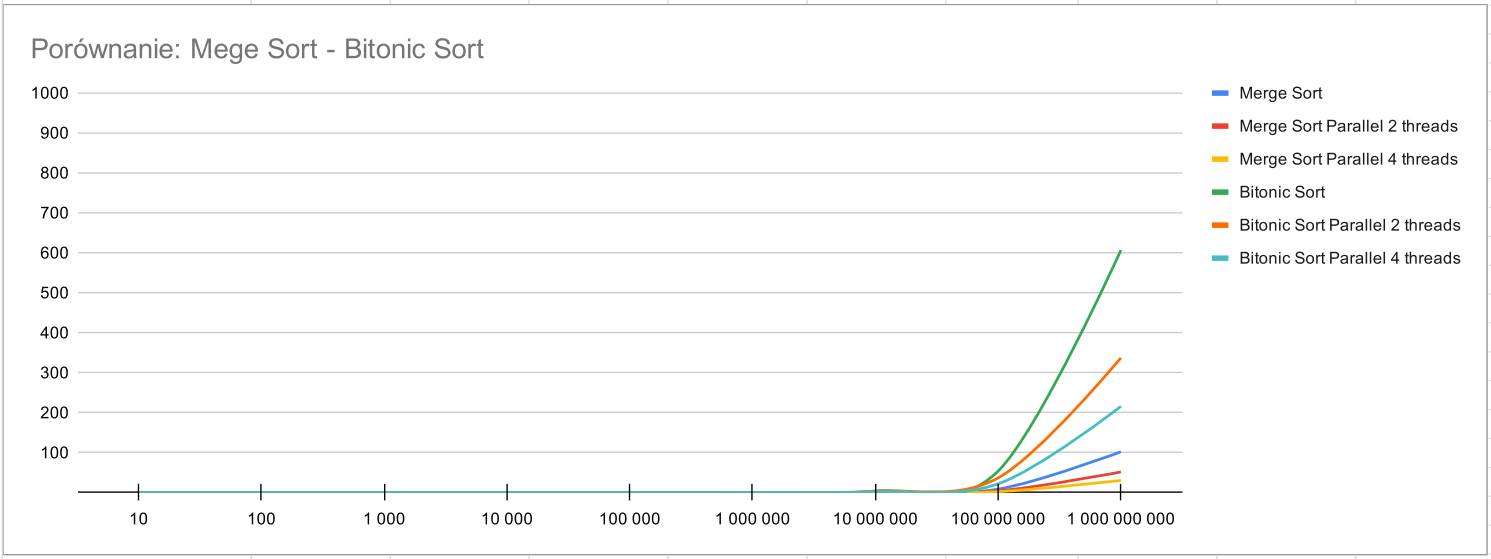
Rezultaty przeprowadzonych testów

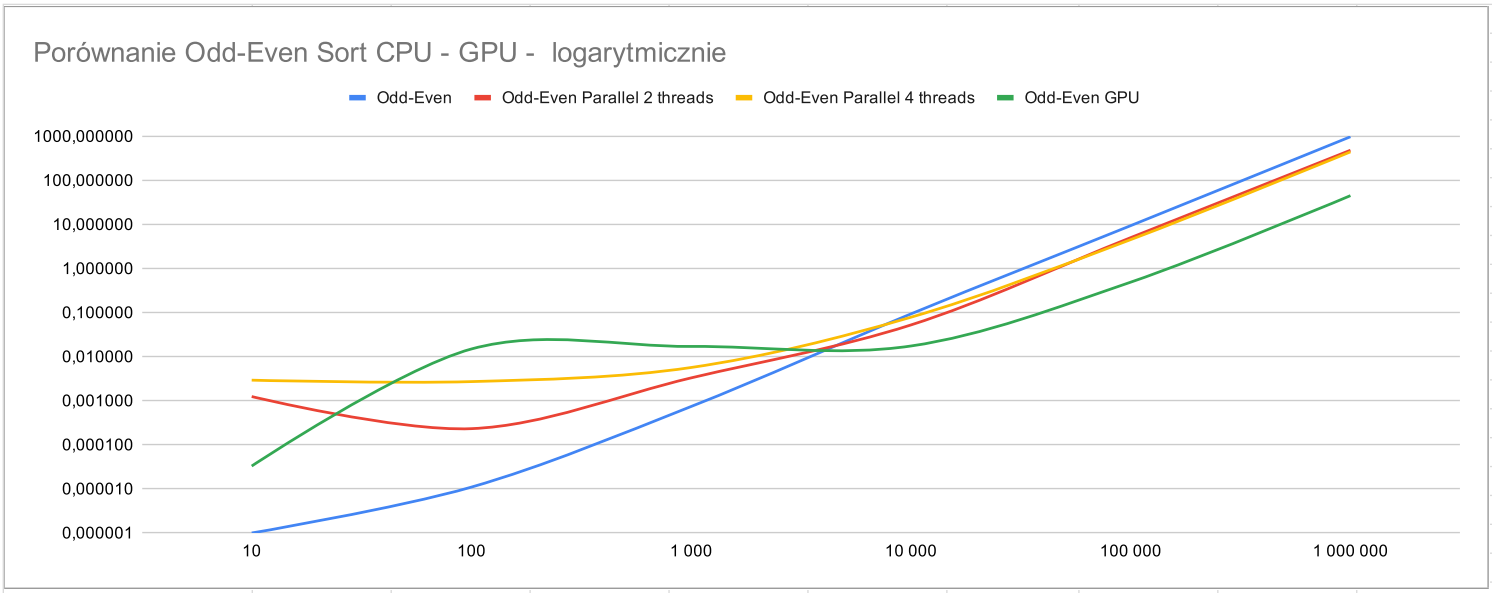
Dane sprzętu widać na poniższym zdjęciu.

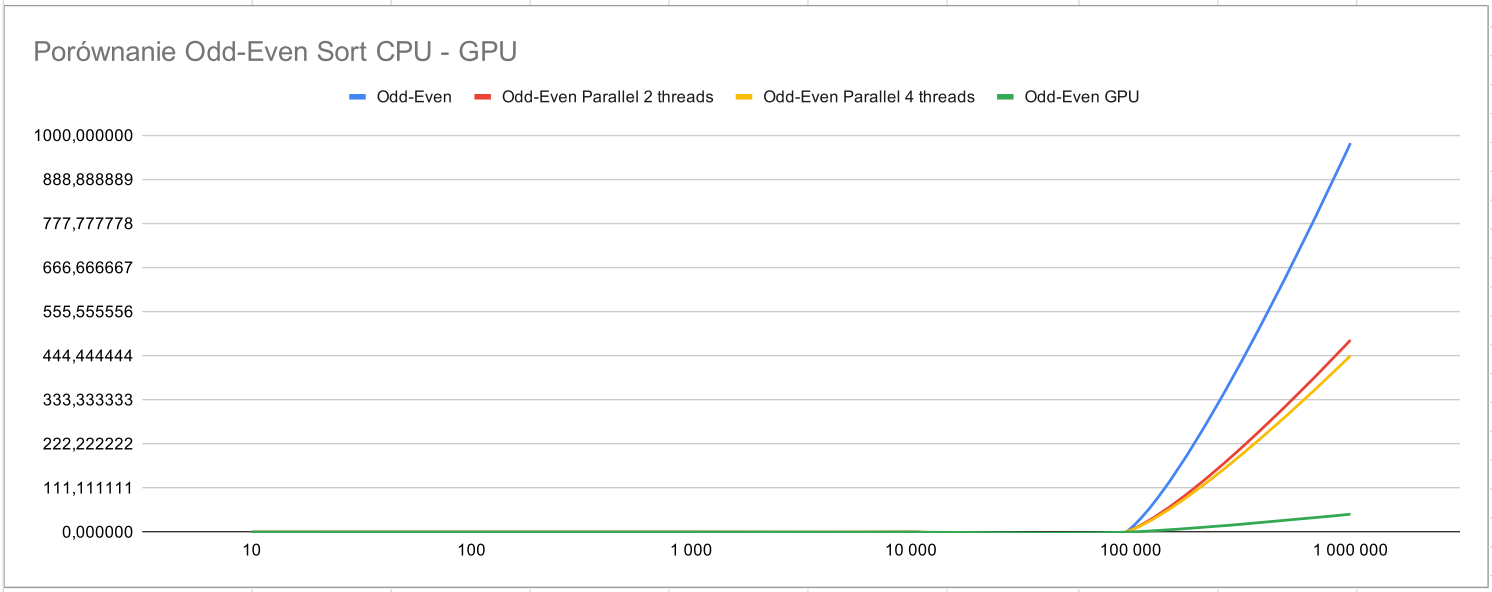
Poniżej znajdują się wyniki przeprowadzonych testów.

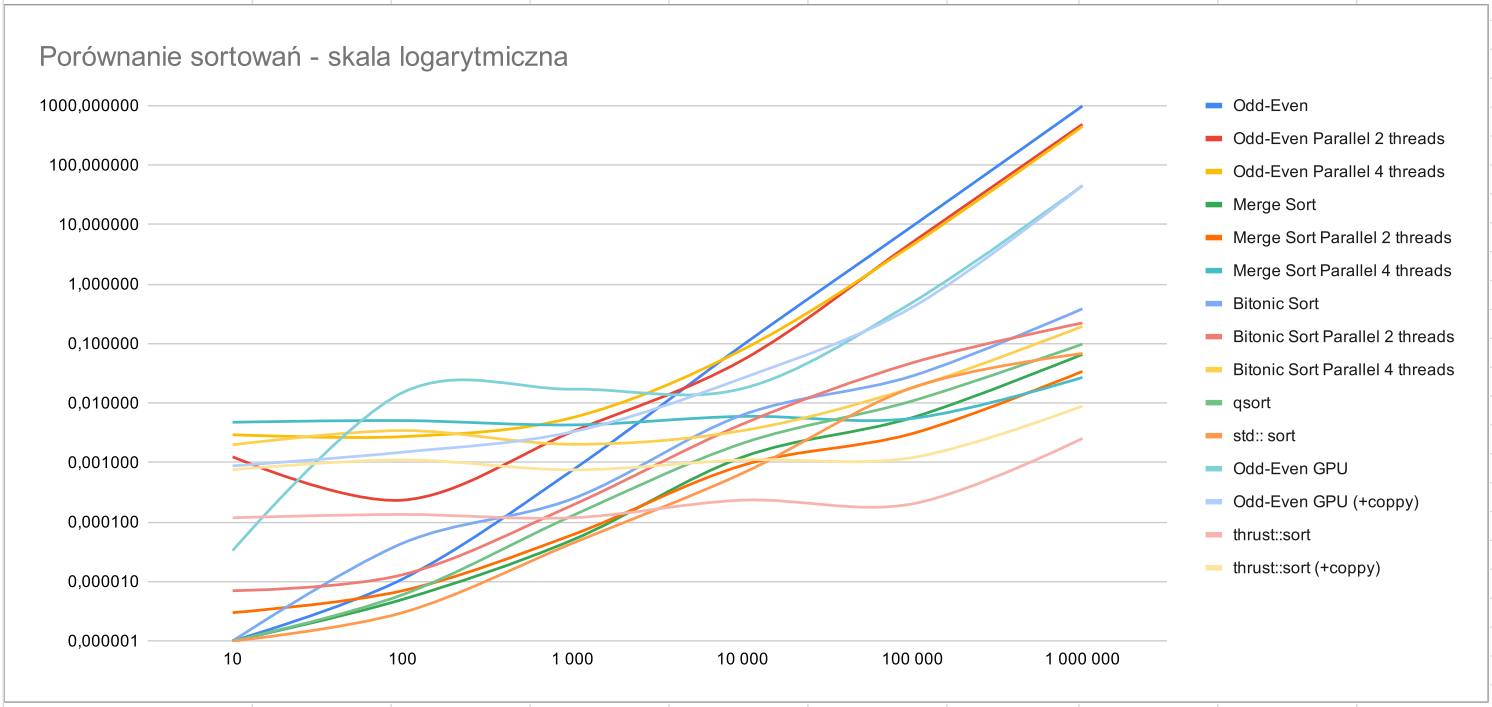
	10	100	1 000	10 000	100 000	1 000 000	10 000 000	100 000 000	1 000 000 000
Odd-Even	0,000001	0,000011	0,000743	0,093644	9,418417	980,736501	-	-	-
Odd-Even Parallel 2 threads	0,001248	0,000233	0,003309	0,052591	5,035802	484,351394	-	-	-
Odd-Even Parallel 4 threads	0,002936	0,002723	0,005693	0,078695	4,539600	444,722019	-	-	-
Merge Sort	0,000001	0,000005	0,000050	0,001233	0,005690	0,065743	0,892216	9,088763	102,623934
Merge Sort Parallel 2 threads	0,000003	0,000007	0,000061	0,000895	0,003069	0,034044	0,470419	4,776886	52,054819
Merge Sort Parallel 4 threads	0,004749	0,005067	0,004289	0,005965	0,005515	0,026918	0,264380	2,761510	30,906024
Bitonic Sort	0,000001	0,000044	0,000246	0,006295	0,028526	0,383898	4,340998	54,148530	606,939705
Bitonic Sort Parallel 2 threads	0,000007	0,000013	0,000192	0,004422	0,047442	0,221351	3,925241	37,036774	337,447652
Bitonic Sort Parallel 4 threads	0,001982	0,003447	0,002026	0,003439	0,018059	0,194276	2,259519	22,275162	216,111167
qsort	0,000001	0,000006	0,000128	0,002101	0,010869	0,097757	0,966445	10,922672	110,495514
std:: sort	0,000001	0,000003	0,000044	0,000660	0,018191	0,068001	0,668323	6,922672	70,485905
Odd-Even GPU	0,000033	0,014980	0,017104	0,017505	0,491569	45,189395	-	-	-
Odd-Even GPU (+coppy)	0,000878	0,001490	0,003280	0,026173	0,408169	45,076155	-	-	-
thrust::sort	0,000118	0,000134	0,000118	0,000233	0,000202	0,002539	0,008120	0,069288	-
thrust::sort (+coppy)	0,000759	0,001107	0,000753	0,001106	0,001210	0,008906	0,024378	0,229208	-

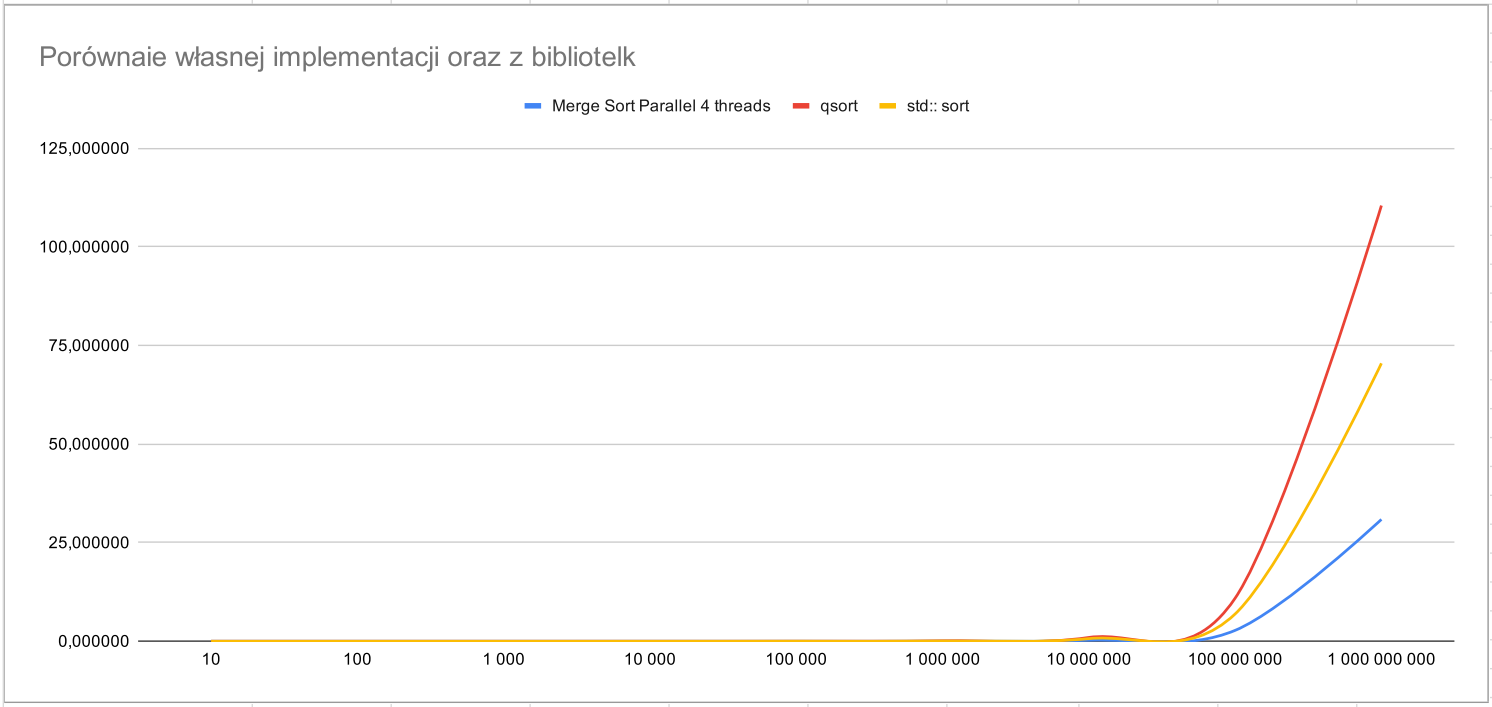


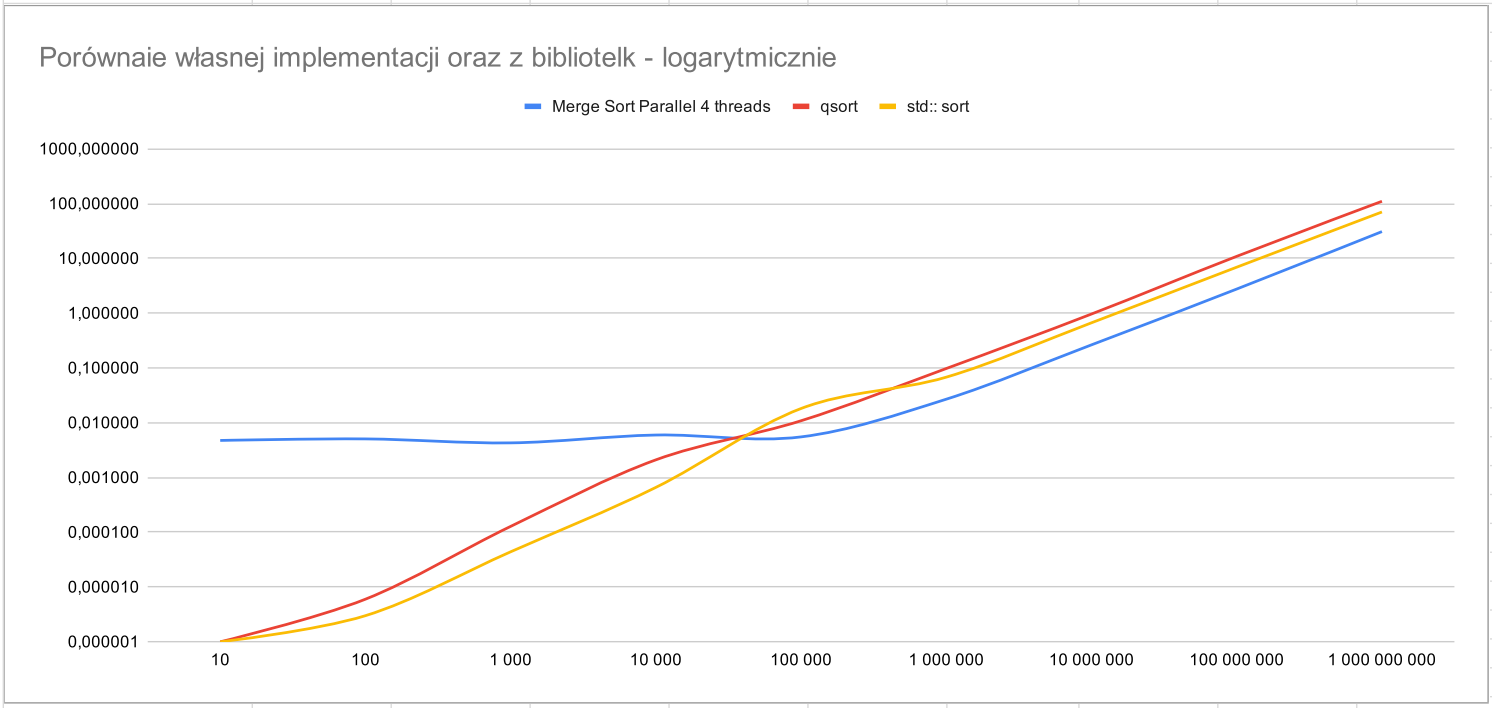



Porównanie Odd-Even Sort CPU - GPU - logarytmicznie
Odd-Even
Odd-Even Parallel 2 threads
Odd-Even Parallel 4 threads
Odd-Even GPU
1000,000000
100,000000
10,000000
1,000000
0,100000
0,010000
0,001000
0,000100
0,000010
0,000001
10
100
1 000
10 000
100 000
1 000 000


Porównanie Odd-Even Sort CPU - GPU
Odd-Even
Odd-Even Parallel 2 threads
Odd-Even Parallel 4 threads
Odd-Even GPU
1000,000000
888,888889
777,777778
666,666667
555,555556
444,444444
333,333333
222,222222
111,111111
0,000000
10
100
1 000
10 000
100 000
1 000 000


Porównanie sortowań - skala logarytmiczna
1000,000000
100,000000
10,000000
1,000000
0,100000
0,010000
0,001000
0,000100
0,000010
0,000001
10
100
1 000
10 000
100 000
1 000 000
Odd-Even
Odd-Even Parallel 2 threads
Odd-Even Parallel 4 threads
Merge Sort
Merge Sort Parallel 2 threads
Merge Sort Parallel 4 threads
Bitonic Sort
Bitonic Sort Parallel 2 threads
Bitonic Sort Parallel 4 threads
qsort
std:: sort
Odd-Even GPU
Odd-Even GPU (+coppy)
thrust::sort
thrust::sort (+coppy)


Porównaie własnej implementacji oraz z bibliotelk
Merge Sort Parallel 4 threads
qsort
std:: sort
125,000000
100,000000
75,000000
50,000000
25,000000
0,000000
10
100
1 000
10 000
100 000
1 000 000
10 000 000
100 000 000
1 000 000 000


Porównaie własnej implementacji oraz z bibliotelk - logarytmicznie
Merge Sort Parallel 4 threads
qsort
std:: sort
1000,000000
100,000000
10,000000
1,000000
0,100000
0,010000
0,001000
0,000100
0,000010
0,000001
10
100
1 000
10 000
100 000
1 000 000
10 000 000
100 000 000
1 000 000 000