Akceleracja obliczeń w przetwarzaniu danych - Projekt
Zrównoleglone algorytmy sortowania

Dawid Waligórski
Michał Dziedziak
Jakub Łazorko
Marcin Sitarz
Michał Zychowicz

Grudzień 2024

Spis treści

1 Wprowadzenie

1.1 Cel

Celem projektu była implementacja algorytmów sortujących *Odd-even* i *Bitonic* w wersjach wielowątkowych przeznaczonych do wykonywania na jednostce centralnej (ang. *Central Processing Unit*, CPU) oraz procesorze graficznym (ang. *Graphics Processing Unit*, GPU) komputera. Dodatkowo postanowiono zbadać czas działania implementacji wskazanych algorytmów, aby porównać ze sobą efektywność wykonywania obliczeń ogólnego przeznaczenia na obu rozważanych architekturach.

1.2 Zagadnienia teoretyczne

Do momentu omawiania działania algorytmów sortowania Odd-Even oraz Bitonic, część teoretyczna dokumentacji opiera się na wiedzy zdobytej podczas wykładów z kursu „Akceleracja obliczeń w przetwarzaniu danych" oraz na materiałach przygotowanych przez dr inż. Marka Wodę, które zostały stworzone na potrzeby tych wykładów. Jeśli chodzi o algorytm Odd-Even, wiedzę na jego temat zdobyliśmy z materiałów dostępnych na stronie `https://www.geeksforgeeks.org/odd-even-sort-brick-sort/`, natomiast o algorytmie Bitonic z `https://www.geeksforgeeks.org/bitonic-sort/`.

1.2.1 Obliczenia równoległe

Wykonywanie wielu instrukcji jednocześnie, w przeciwieństwie do tradycyjnego sekwencyjnego podejścia, w którym to instrukcje przetwarzane są jedna po drugiej. Zyskały na popularności ze względu na ograniczenia fizyczne dotyczące zwiększania częstotliwości taktowania procesorów jakimi były
i nadal są:

- wzrost użycia energii,
- nagrzewanie procesorów,
- spadek stabilności wraz ze wzrostem częstotliwości taktowania.

Równoległe przetwarzanie danych umożliwia znaczące skrócenie czasu obliczeń przy odpowiedniej implementacji, jak i odpowiednio dużej wielkości przetwarzanych danych – w przypadku zbyt małych zbiorów danych równoległość może być nieopłacalna, ponieważ czas potrzebny na przesyłanie i zarządzanie danymi przewyższa korzyści wynikające z równoległego przetwarzania. Powiązane pojęcia:

- Wielowątkowość – wykonywanie kilku zadań (wątków) w ramach jednego procesu. Została wprowadzona, aby umożliwić przetwarzanie współbieżne, do którego potrzebne jest co najmniej wiele procesorów jednordzeniowych lub jeden procesor wielordzeniowy. Największymi przeszkodami w jej stosowaniu są:
  - *Deadlock* – dwa lub więcej procesy czekają na siebie nawzajem i w efekcie żaden nie może się rozpocząć.
  - *Race condition* – zachowanie systemu zależy od niekontrolowanych zdarzeń, co w efekcie może dostarczać niepożądane wyniki.
  - *Starvation* – złe zarządzanie dostępem procesu do zasobów może doprowadzić do sytuacji kiedy ten dostęp nigdy nie nadejdzie.
- Modele przetwarzania równoległego – strategie partycjonowania i przetwarzania danych. Wyróżniamy:
  - SIMD (ang. *Single Instruction Multiple Data*),
  - MIMD (ang. *Multiple Instruction Multiple Data*),
  - SIMT (ang. *Single Instruction, Multiple Threads*).

Dla obliczeń równoległych istotna jest również efektywna komunikacja pomiędzy współpracującymi ze sobą jednostkami. Kluczowe znaczenie ma optymalizacja dostępu do pamięci, aby zminimalizować opóźnienia w dostępie do danych (ang. *latency*) oraz odpowiednia synchronizacja danych, która zapewnia poprawność i szybkość obliczeń.

Warto pamiętać, że nie każdy algorytm można zrównoleglić. Równoległość jest możliwa w przypadku zadań niezależnych od siebie, które mogą być wykonywane jednocześnie. Maksymalne przyspieszenie obliczeń opisuje prawo Amdahla, które określa granice wzrostu wydajności w zależności od udziału części algorytmu, którą można zrównoleglić w kontekście całego programu.

1.2.2 Charakterystyka CPU

CPU (ang. *Central Processing Unit*) posiada kilka mocnych rdzeni, co pozwala na obsługę kilkudziesięciu wątków jednocześnie, ponieważ każdy rdzeń może obsłużyć określoną liczbę wątków. Dzięki temu różne zadania mogą być wykonywane równolegle na wątkach, jednak w praktyce często zdarza się, że liczba wątków przekracza możliwości rdzeni – w takich sytuacjach CPU dynamicznie przełącza kontekst między wątkami, aby obsłużyć wszystkie zadania w sposób poprawny i wydajny.

CPU kładzie duży nacisk na szybką pamięć podręczną (*cache*), ponieważ jest ona kluczowa dla szybkiego dostępu do najczęściej używanych danych i instrukcji, co minimalizuje opóźnienia wynikające z odwołań do wolniejszej pamięci RAM. Programowanie na CPU jest prostsze w porównaniu do GPU, ponieważ deweloperzy korzystają z bardziej wszechstronnych języków programowania, takich jak C++, Python czy Java, a także nie muszą zarządzać złożoną strukturą pamięci, co upraszcza proces tworzenia oprogramowania.

CPU jest wykorzystywane przede wszystkim do obsługi systemu operacyjnego, przetwarzania danych o różnorodnym charakterze oraz zarządzania logiką aplikacji. Jeżeli zachodzi potrzeba wykonania dużej liczby prostych, krótkich obliczeń, CPU często wykorzystuje GPU jako wsparcie w celu odciążenia i przyspieszenia procesów.

1.2.3 Charakterystyka GPU

GPU (ang. *Graphics Processing Unit*) posiada kilka tysięcy rdzeni, z których każdy zazwyczaj obsługuje tylko jeden wątek, co umożliwia wykonywanie ogromnej liczby obliczeń równolegle, jednakże trzeba pamiętać, że tyczy się do wykonywania stosunkowo prostych operacji. Zadania są przetwarzane w ten sposób, że dane są przesyłane z CPU do GPU, gdzie są przetwarzane, a następnie odsyłane z powrotem do CPU.

GPU korzysta z hierarchii pamięci, która obejmuje rejestry, pamięć lokalną, pamięć współdzieloną oraz pamięć globalną. Optymalizacja zależy od konkretnej implementacji i rodzaju obliczeń – w zależności od potrzeb, odpowiednie dane przechowywane są w różnych typach pamięci, aby zmaksymalizować wydajność i wykorzystać atuty poszczególnego typu pamięci.

Watki w GPU są organizowane w grupy po 32 wątki, zwane *warpami*, które następnie są grupowane w bloki, a bloki w *gridy*. Każdy wątek posiada swój własny indeks, co pozwala na efektywne przetwarzanie dużych zbiorów danych.

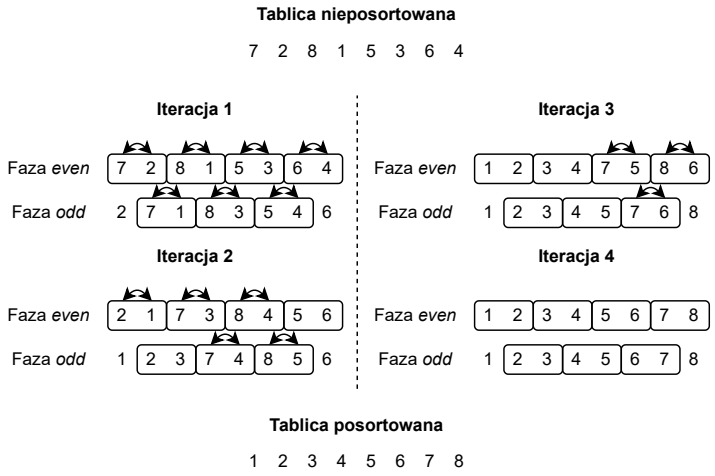
GPU znajduje zastosowanie w renderowaniu filmów i grafiki 3D, kopaniu kryptowalut, trenowaniu modeli uczenia maszynowego oraz przetwarzaniu obrazu. Do programowania GPU wykorzystuje się specjalistyczne technologie, takie jak CUDA (dla kart NVIDIA) lub OpenCL, które wymagają zaawansowanej wiedzy o zarządzaniu wątkami, synchronizacji i pamięcią przez co proces pisania kodu staje się bardziej skomplikowany.

1.2.4 Algorytm sortujący *Odd-even*

Algorytm podobny do algorytmu *sortowania bąbelkowego*, który przechodzi przez listę elementów, zamieniając (ang. *swap*) sąsiadujące elementy będące w nieodpowiedniej kolejności, do momentu kiedy nie musi wykonywać już żadnych zmian. Algorytm *Odd-even* posiada przy tym dwie fazy: *even* oraz *odd*. W fazie *even* porównuje pary elementów zaczynające się od indeksów parzystych, a w fazie *odd* pary elementów zaczynające się od indeksów nieparzystych. Jeżeli w ramach takiego porównania pierwszy element w parze okaże się większy niż jego następnik, to następuje zmiana kolejności w parze (rys. 1). Jeżeli w danej iteracji nie wykonano żadnej zamiany w obu fazach, to tablica zostaje uznana posortowaną a algorytm może zakończyć swoje działanie. W przeciwnym wypadku obie fazy są wykonywanie jeszcze raz.

Złożoność obliczeniowa sekwencyjnego algorytmu *Odd-even* wynosi $O(n^2)$. Wynika to z faktu, że zarówno w fazie *odd* jak i *even* porównuje on przez maksymalnie $\frac{n}{2}$ par w tablicy, a uruchomienie obu faz w najgorszym wypadku powtarzane jest $\frac{n}{2}$ razy. Dzieje się tak momencie kiedy element musi zostać przemieszczony z jednego skraju tablicy na drugi. Wówczas w ramach każdej iteracji każda z faz przesuwa go o 1 pozycję w „dobrym kierunku", a jego pełne przemieszczenie zajmie właśnie $\frac{n}{2}$ iteracji.

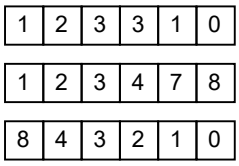
Rozważany algorytm może być przyspieszony poprzez równoległe wykonywanie porównań par w ramach każdej z faz. Jest to możliwe, gdyż wszystkie takie porównania są między sobą niezależne w przeciwieństwie do faz, których działanie zależne jest od wyników faz je poprzedzających.



Rysunek 1: Przebieg sortowania algorytmem *Odd-even* na przykładzie 8-elementowej tablicy liczb

1.2.5 Algorytm sortujący *Bitonic*

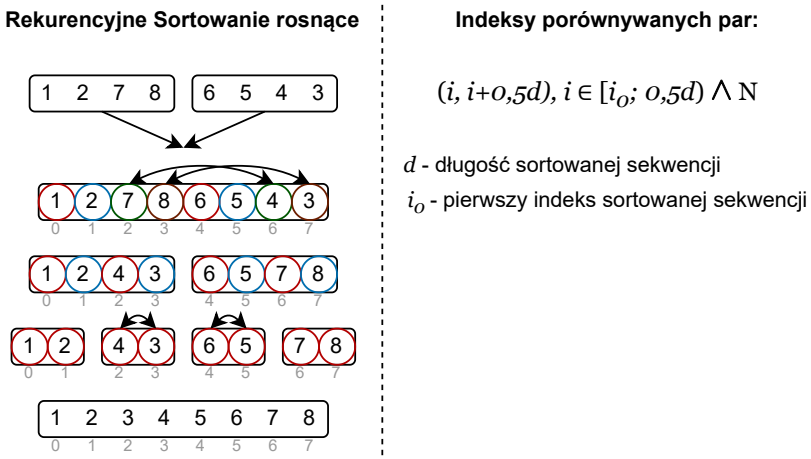
Algorytm sortujący, który polega na tworzeniu z fragmentów sortowanej tablicy tzw. *sekwencji bitonicznych*. Są to sekwencje, które do pewnego momentu są rosnące a potem malejące (lub odwrotnie). Warto przy tym pamiętać, że za *bitoniczne* uznaje się również sekwencje całkowicie rosnące jak i całkowicie malejące. Przykłady *sekwencji bitonicznych* zawarto na rysunku 2.



Rysunek 2: Przykłady poprawnych sekwencji bitonicznych

Pierwszy etap sortowania polega na rekurencyjnym dzieleniu tablicy na równe dwie części, z których „lewa" jest oznaczana jako sortowana rosnąco a „prawa" malejąco (główna tablica oznaczana jest jako sortowana rosnąco). Podział tablicy kończy się w momencie uzyskania 1-elementowych pod-tablic. Wówczas następuje ponownie łączenie rozdzielonych wcześniej sekwencji. W ramach takiego łączenia

następuje sortowanie oparte na ciągu porównań par elementów na znajdujących się na odpowiednich indeksach. Jeżeli kolejność w parze jest niezgodna z tą przyjętą dla danej pod-tablicy, to są one zamieniane miejscami (*swap*). Sortowania w ramach łączenia również wykonywane są rekurencyjnie i znów do momentu uzyskania 1-elementowych, posortowanych odpowiednio pod-tablic. Porównania układają się przy tym we wzorzec, który został przedstawiony na rysunku 11.



Rysunek 3: Przebieg sortowania odbywającego się podczas łączenia pod-tablic w alrotymie *Bitonic*, na przykładzie łączenia pod-tablic 4-elementowych w tablicę 8-elementową. Przedstawiono także wzorzec określający pary porównywane na każdym etapie rekurencji odbywającej się w ramach łączenia

Ograniczeniem algorytmu *Bitonic* jest wymóg, aby sortowane przy jego pomocy tablice miały rozmiar będący potęgą liczby 2. Wynika to bezpośrednio z ciągłym, wykonywanym w nim rekurencyjnym dzieleniu sortowanej tablicy na dwie równe części.

Złożoność obliczeniowa rozważanego algorytmu wynosi $O(log^2(n))$, gdyż rekurencyjnie dzieli on tablice na pod-tablice a następnie w ramach każdego takiego podziału rekurencyjnie sortuje on kolejne pod-tablice.

Sortowanie *Bitonic* nadaje się idealnie do przyspieszenia poprzez przetwarzanie równoległe. Jego rekurencyjna, dzieląca na połowy tablicę natura sprawa bowiem, że porównania wykonywane są w nim zawsze w tej samej kolejności, niezależnie od wyników przednich porównań. Można zatem wykonywać je równolegle wykorzystując wzorzec określający ich kolejność.

2 Koncepcja projektu

Według założeń projektowych miały zostać zaimplementowane oraz porównane ze sobą implementacje algorytmów sortowania *Odd-even* oraz *Bitonic* wykonywane GPU oraz CPU. Wspomniane badania były realizowane według procedury badawczej opisanej w sekcji 2.1. Wszystkie pomiary były przeprowadzone przy pomocy stworzonej do tego celu aplikacji badawczej, której działanie opisano w sekcji 2.2.

2.1 Procedura badawcza

2.1.1 Hipoteza badawcza

(1) Procesor graficzny jest sprzętowo przystosowany do wykonywania równoległych obliczeń, najlepiej na dużych ilościach danych. Z tego względu spodziewanym jest, że jego efektywność pod względem czasu obliczeń w porównaniu do CPU będzie rosnąć wraz ze wzrostem rozmiaru problemu.

(2) Napisanie efektywnie wykorzystującego zasoby sprzętowe i szybkiego kodu dla procesora graficznego jest zadaniem nietrywialnym. Często osiągnięcie zadowalających efektów (przyspieszenia względem CPU) wymaga przejścia przez proces odpowiedniej optymalizacji kodu. Po przejściu przez taki proces spodziewane jest jednak znaczące skrócenie się czasu wykonywania i zwiększenie uzyskanego przyspieszenia względem CPU.

(3) Rozrzut uzyskanych wyników będzie większy w przypadku implementacji na GPU ze względu na fakt, że są one zależne od operacji transferu danych między pamięciami hosta i urządzenia.

2.1.2 Plan badania

W ramach projektu zostaną wykonane pomiary zależności między czasem wykonywania się w sekundach danych implementacji równoległych algorytmów sortowania (t) a rozmiarem problemu (n). Przeprowadzone zostaną one dla 4 następujących, napisanych uprzednio i zweryfikowanych pod kątem poprawności implementacji:

1. *Bitonic* wykonywane na CPU,
2. *Bitonic* wykonywane na GPU,
3. *Odd-even* wykonywane na CPU,
4. *Odd-even* wykonywane na GPU.

Całość będzie odbywać się przy pomocy aplikacji badawczej (patrz sekcja 2.2), uruchomionej na komputerze stacjonarnym. Specyfikacja wspomnianej maszyny zawierająca parametry kluczowe w badaniu została zamieszczona w tabeli 1.

W celu uzyskania odpowiedzi na punkty hipotezy badawczej postawionej w sekcji 2.1.1 ustalono odpowiedni harmonogram pomiarów, który będzie składać się z dwóch etapów.

- **Etap I:** Przeprowadzenie pomiarów zależności $t(n)$ dla pierwszej, niezoptymalizowanej wersji wszystkich implementacji.
- **Etap II:** Przejście przez proces optymalizacji implementacji przeznaczonych do wykonywania na GPU, a następnie powtórzenie pomiarów zależności $t(n)$.

W obu etapach badaniu zostaną poddane zestawy danych, które będą składać się z ciągów losowych instancji problemu sortowania o rosnącym n. Zestawy zostaną zdefiniowane tak, aby w obu etapach zbadano taki sam ciąg rozmiarów problemu. Dane pomiarowe zostały dokładniej opisane w sekcji 2.1.3.

Dla każdego rozmiaru problemu pomiary t implementacji zostaną powtórzone wielokrotnie, tak aby uzyskać wiarygodną wartość średnią oraz odpowiadające jej odchylenie standardowe.

Tabela 1: Parametry CPU, karty graficznej oraz pamięci głównej maszyny, na której wykonano pomiary

Parametr maszyny badawczej	**Wartość**
Model CPU	AMD Ryzen 7 5700X
Częstotliwość zegara CPU	3.4 GHz
Liczba rdzeni CPU	8
Liczba wątków CPU	16
Ilość pamięci cache L1 CPU (na rdzeń)	64 KB
Ilość pamięci cache L2 CPU (na rdzeń)	512 KB
Ilość współdzielonej pamięci cache L3 CPU	32 MB
Ilość pamięci głównej	32 GB
Częstotliwość zegara pamięci głównej	3600 MHz
Model GPU	NVIDIA GeForce RTX 4070 Ti SUPER
Częstotliwość zegara GPU	2.61 GHz
Liczba rdzeni GPU	8448
Liczba wątków GPU	135 168
Ilość pamięci globalnej GPU	16 GB
Częstotliwość zegara pamięci globalnej GPU	10501 MHz
Ilość pamięci współdzielonej GPU na blok wątków	48 KB
Ilość pamięci rejestrowej GPU na wątek	256 B
Ilośc pamięci cache L2 GPU	48 MB

2.1.3 Dane pomiarowe

Badaniu poddane zostaną dwa zestawy instancji problemu sortowania. Pierwszy z nich zostanie zbadany w etapie I, a drugi w etapie II badania (patrz sekcja 2.1.2). Oba zestawy nie będą składać się z tych samych instancji, gdyż te generowane będą losowo (sposób generowania opisano w sekcji 2.2.3). Identyczne między zestawami będą jedynie przebadane rozmiary problemu, liczby przebadanych instancji danego rozmiaru i liczby powtórzeń pomiarów wykonywanych dla poszczególnych rozmiarów.

Przebadane rozmiary problemu wraz przypisaną im ilością powtórzeń pomiarów wykonanych dla każdej instancji rozwiązywanej przy pomocy każdej z implementacji zawarto w tabeli 2. Ze względu na specyfikę algorytmu *Bitonic* (patrz sekcja 1.2.5) musiały być one potęgami liczby 2. Jako, że czasy działania algorytmów rosną wraz ze wzrostem rozmiaru problemu, to liczba powtórzeń była przypisywana tak, aby zachować czas poświęcany na pomiar pojedynczego n dla danej implementacji (przed optymalizacją) w rozsądnych ramach. Za ich górną granicę przyjęto pułap 20 minut bądź wykonanie łącznie 100 powtórzeń pomiaru. Warto przy tym wspomnieć, że ramach każdego n badane jest 5 losowo wygenerowanych instancji.

Tabela 2: Zbadane rozmiary problemu sortowania wraz z przypisaną do nich liczbą powtórzeń pomiaru, wykonywanych dla każdej badanej instancji tego rozmiaru

Rozmiar problemu	**Liczba powtórzeń dla pojedynczej instancji**
2^{15}	100
2^{16}	100
2^{17}	100
2^{18}	100
2^{19}	60
2^{20}	25
2^{21}	10
2^{22}	1

2.2 Aplikacja badawcza

Szczegółowe informacje na temat narzędzia pomiarowego można znaleźć w pliku README.md znajdującym się w repozytorium projektowym pod adresem `https://github.com/CenturionTheMan/ParallelSortingAlgorithms`.

2.2.1 Zasada działania

Po uruchomieniu aplikacji ładowana jest konfiguracja z pliku `configuration.ini`, która definiuje parametry pomiarowe, takie jak rodzaje algorytmów, środowisko sprzętowe (CPU/GPU), liczba powtórzeń dla każdego rozmiaru instancji oraz tryb weryfikacji wyników. Następnie aplikacja generuje losowe lub wczytuje zdefiniowane instancje problemu, które mają zostać posortowane. Każda instancja jest przesyłana do odpowiednich implementacji algorytmów na CPU i GPU, gdzie wykonywane są pomiary czasu sortowania. Po zakończeniu sortowania dla danej instancji aplikacja zapisuje wyniki do pliku `result.csv`. Jeżeli włączony jest tryb weryfikacji, aplikacja sprawdza poprawność posortowanego wyniku – w przypadku wykrycia błędu działanie aplikacji zostaje przerwane, a szczegóły błędu są zapisywane w pliku `error.log`. Proces jest powtarzany dla każdej instancji problemu, a po zakończeniu pomiarów generowany jest raport w konsoli z wynikami oraz statystykami (rys. 4).

```
>>> STARTING BENCHMARK...

#==============#========================#========================#========================#========================#
| Instance size |       CPU Bitonic      |       GPU Bitonic      |       CPU Odd-Even     |       GPU Odd-Even     |
#==============#========================#========================#========================#========================#
|           128 |   2.35e-03 (4.64e-04) s |   2.60e-03 (1.26e-02) s |    3.62e-03 (3.57e-04) s |    2.58e-03 (8.35e-04) s |
|           256 |   2.43e-03 (2.20e-04) s |   1.28e-03 (1.65e-04) s |    8.34e-03 (6.11e-04) s |    3.47e-03 (2.93e-04) s |
|           512 |   2.48e-03 (2.88e-04) s |   1.52e-03 (1.74e-04) s |    2.94e-02 (2.01e-03) s |    6.72e-03 (6.77e-04) s |
|          1024 |   2.66e-03 (3.22e-04) s |   2.56e-03 (5.21e-04) s |    6.26e-02 (4.63e-03) s |    1.85e-02 (2.81e-03) s |
#==============#========================#========================#========================#========================#

>>> BENCHMARK COMPLETE!
>>> Press ENTER to continue...
```

Rysunek 4: Zrzut ekranu z raportu generowanego przez aplikacje badawczą

2.2.2 Pomiar czasu

Dla każdej instancji problemu aplikacja mierzy czas wykonania algorytmów *Bitonic* i *Odd-Even* w implementacjach na CPU oraz GPU. Aby zapewnić wiarygodne wyniki, dla każdego rozmiaru instancji przeprowadzana jest określona liczba powtórzeń, a na ich podstawie obliczana jest średnia oraz odchylenie standardowe. Wyniki pomiarów są wyświetlane w formie tabeli w konsoli, a szczegóły zapisywane do pliku `result.csv` – w przypadku błędu w działaniu któregoś z algorytmów aplikacja generuje odpowiedni komunikat i zapisuje szczegóły błędu w pliku `error.log`. Pomiar czasu opiera się na wykorzystaniu precyzyjnego zegara systemowego, który pozwala na rejestrowanie szczegółowych wyników czasów wykonania.

2.2.3 Generowanie instancji

Aplikacja umożliwia generowanie instancji problemu w dwóch trybach: losowym oraz zdefiniowanym przez użytkownika. W przypadku trybu losowego instancje są tworzone na podstawie rozmiaru zadanego w pliku `configuration.ini`, a wartości elementów są generowane pseudolosowo. W przypadku zdefiniowanych instancji użytkownik ma możliwość dokładnego określenia ich zawartości, podając elementy w konfiguracji.

3 Implementacja i optymalizacja algorytmów

3.1 Implementacja algorytmu *Odd-even* (CPU)

3.1.1 Struktura implementacji

Implementację algorytmu *Odd-even* przeznaczoną do wykonywania na CPU podzielono na 2 następujące części:

1. `CpuOddEvenSort()`: Główna funkcja pełniąca rolę punktu wejście do procesu sortowania. Koordynuje ona również wykonywanie wątków realizujących fazy *odd* oraz *even* algorytmu. Wykonywana sekwencyjnie.

2. `runPhasesOnArrayChunk()`: Funkcja realizująca na wskazanym fragmencie tablicy naprzemiennie fazy zarówno *odd* oraz *even* algorytmu. Wykonywana równolegle przez wiele wątków.

3.1.2 Funkcja `CpuOddEvenSort()`

Główna funkcja realizująca algorytm *Odd-even* wykonywany na CPU, która jest również jego punktem wejścia. Jej odpowiedzialnością jest koordynacja całego procesu sortowania, które wykonywane jest na poszczególnych, rozłącznych fragmentach tablicy przez uruchamiane przez nią wątki.

Jej zadaniem jest dynamiczne wyznaczenie liczby zaprzęgniętych do wykonania sortowania wątków. W tych obliczeniach wykorzystywana jest zarówno liczba wskazująca na liczbę wątków sprzętowych procesora jak i długość sortowanej tablicy. Dzięki temu algorytm jest w stanie efektywnie i w pełni wykorzystywać dostępne zasoby sprzętowe do osiągnięcia jak największych zysków z przetwarzania równoległego.

Listing 1: Ustalanie liczby wątków przez funkcję `CpuOddEvenSort()`

```
void sorting::CpuOddEvenSort(std::vector<int>& arr) {
    const int THREADS_COUNT = std::min(
        int(std::thread::hardware_concurrency()), std::max(1, int(std::log2(arr.size())) - 5)
    );
    std::vector<std::thread> threads(THREADS_COUNT);

    // ...
}
```

W ramach rozważanej funkcji deklarowana jest również współdzielona przez wszystkie wątki flaga logiczna (`is_sorted`). Ta flaga pozwala na określenie czy w danym momencie tablica może zostać uznać za posortowaną, a tym samym wyznacza warunek kończący działanie algorytmu. Mechanizmem chroniącym dostęp do tego współdzielonego zasobu jest deklarowany w rozważanej funkcji mutex (`is_sorted_mutex`) przekazywany każdemu z pracujących wątków.

Listing 2: Deklaracja współdzielonego przez wątki zasobu przez funkcję `CpuOddEvenSort()`

```
void sorting::CpuOddEvenSort(std::vector<int>& arr) {
    // ...

    bool isSorted = false;
    std::mutex isSortedMutex;

    // ...
}
```

Najistotniejszym elementem funkcji jest pętla uruchamiająca wątki realizujące naprzemiennie fazy algorytmu *Odd-even*. Wykonuje się ona tak długo, aż tablica zostanie uznana za posortowaną, czyli aż w każdym z wątków nie wystąpi żadna zamiana w ramach przetwarzanych par. W ramach jej każdej iteracji uruchamiana jest wcześniej ustalana liczba wątków, z których każdy realizuje fazy *odd* i *even* na przydzielonym sobie fragmencie sortowanej tablicy. Warto jednak zauważyć, że w celu zachowania poprawności wyników każdy wątek korzysta także z początkowego elementu fragmentu tablicy wątku sąsiedniego. W teorii może to prowadzić do wystąpienia zjawiska wyścigu krytycznego w wypadku, gdy

współdzielące taki element wątki nie wykonują w danym momencie takiej samej fazy. Szanse na jego wystąpienie oceniono jednak na na tyle marginalne, że zrezygnowano z wprowadzania mechanizmu synchronizacji, który spowolniłby wykonywanie się algorytmu.

Opisane wykorzystanie wątków odbiega od klasycznego podejścia równoległego wykonywania algorytmu *Odd-even*, które zakłada, że wątki powinny wykonywać równolegle tylko i wyłącznie poszczególne porównania w ramach każdej z faz. Wybór obecnego rozwiązania podyktowany był faktem, że w przypadku zastosowania podejścia klasycznego same koszty uruchomienia i przełączania się między tak liczbą grupą wątków zniwelowałyby wszelkie zyski z równoległego wykonywania porównań.

Listing 3: Pętla uruchamiająca wątki realizujące fazy algorytmu *Odd-even*

```
void sorting::CpuOddEvenSort(std::vector<int>& arr) {
    // ...

    while (!isSorted) {
        isSorted = true;
        start = 0;

        for (int i = 0; i < THREADS_COUNT - 1; i++) {
            threads[i] = std::thread(std::bind(
                runPhasesOnArrayChunk,
                std::ref(arr),
                start,
                start + ELEMENTS_PER_THREAD,
                std::ref(isSortedMutex),
                std::ref(isSorted)
            ));
            start += ELEMENTS_PER_THREAD;
        }
        threads[THREADS_COUNT - 1] = std::thread(std::bind(
            runPhasesOnArrayChunk,
            std::ref(arr),
            start,
            arr.size(),
            std::ref(isSortedMutex),
            std::ref(isSorted)
        ));

        for (std::thread& t : threads)
            t.join();
    }
}
```

3.1.3 Funkcja `runPhasesOnArrayChunk()`

Wykonywana przez wątki realizujące naprzemiennie fazy kolejno *odd* oraz *even* na przydzielonym sobie fragmencie tablicy. Fazy powtarzane są do momentu kiedy nie nastąpie w nich żadna zamiana elementów w ramach sortowanych par (na co wskazuje zmienna boolowska `swap_performed`). W czasie wykonywania funkcji wątki mogą zmienić wartość współdzielonej przez siebie flagi `is_sorted` określającej czy cała tablica jest posortowana. Dostęp do niej jest chroniony przez dostępny dla wszystkich wątków mutex `is_sorted_mutex`.

Listing 4: Wykonywana przez wątki funkcja realizująca fazy algorytmu *Odd-even*

```
inline void runPhasesOnArrayChunk(
    std::vector<int>& arr,
    const int start_point,
    const int end_point,
    std::mutex& is_sorted_mutex,
    bool& is_sorted
) {
    const int ODD_END = std::min(end_point + 1, int(arr.size() - 1));

    while (true) {
        bool needs_to_change_is_sorted = false;
```

```
        bool swap_performed = false;

        for (int i = start_point + 1; i < ODD_END; i += 2) {
            // Odd
            if (arr[i] > arr[i + 1]) {
                std::swap(arr[i], arr[i + 1]);
                if (!swap_performed) {
                    std::unique_lock<std::mutex> lock(is_sorted_mutex, std::defer_lock);
                    if (lock.try_lock()) {
                        is_sorted = false;
                        swap_performed = true;
                        needs_to_change_is_sorted = false;
                    }
                    else {
                        needs_to_change_is_sorted = true;
                    }
                }
            }
            // Even
            if (arr[i - 1] > arr[i]) {
                std::swap(arr[i - 1], arr[i]);
                if (!swap_performed) {
                    std::unique_lock<std::mutex> lock(is_sorted_mutex, std::defer_lock);
                    if (lock.try_lock()) {
                        is_sorted = false;
                        swap_performed = true;
                        needs_to_change_is_sorted = false;
                    }
                    else {
                        needs_to_change_is_sorted = true;
                    }
                }
            }
        }

        if (arr[ODD_END - 1] > arr[ODD_END]) {
            std::swap(arr[ODD_END - 1], arr[ODD_END]);
            if (!swap_performed) {
                std::unique_lock<std::mutex> lock(is_sorted_mutex, std::defer_lock);
                if (lock.try_lock()) {
                    is_sorted = false;
                    swap_performed = true;
                    needs_to_change_is_sorted = false;
                }
                else {
                    needs_to_change_is_sorted = true;
                }
            }
        }

        if (needs_to_change_is_sorted) {
            std::unique_lock<std::mutex> lock(is_sorted_mutex);
            is_sorted = false;
            swap_performed = true;
        }

        if (!swap_performed)
            break;
    }
}
```

3.2 Wstępna implementacja algorytmu *Odd-even* (GPU)

3.2.1 Struktura implementacji

Wstępną implementację algorytmu *Odd-even* przeznaczoną do wykonywania na GPU podzielono na 3 następujące funkcje:

1. `GpuOddEvenSort()`: Funkcja wykonywana po stronie hosta pełniąca rolę punktu wejścia i koor-

dynatora procesu sortowania.

2. `Odd()`: Jądro z kodem przeznaczonym do wykonywania na GPU. Realizuje ono fazę *even* algorytmu sortowania *Odd-even*.

3. `Even()`: Jądro z kodem przeznaczonym do wykonywania na GPU. Realizuje ono fazę *even* algorytmu sortowania *Odd-even*.

3.2.2 Funkcja `GpuOddEvenSort()`

Funkcja `GpuOddEvenSort()` odpowiada ona za uruchamianie odpowiednich jąder z kodem przeznaczonym do wykonywania na GPU, koordynacje ich działania oraz transfery między pamięcią główną a pamięcią globalną urządzenia (inne rodzaje pamięci urządzenia nie zostały wykorzystane). Najważniejszym, wykonywanym w jej ramach transferem jest ten dotyczący sortowanej tablicy `arr`, która to jest zapisywana pod adresem `d_arr`.

Listing 5: Zarządzanie pamięcią GPU w funkcji wykonwanej po stronie hosta

```
void sorting::GpuOddEvenSort(std::vector<int>& arr) {
    int half = arr.size() / 2;
    int* d_arr;
    cudaMalloc(&d_arr, arr.size() * sizeof(int));
    cudaMemcpy(d_arr, arr.data(), arr.size() * sizeof(int), cudaMemcpyHostToDevice);

    // ...

    cudaMemcpy(arr.data(), d_arr, arr.size() * sizeof(int), cudaMemcpyDeviceToHost);
    cudaFree(d_arr);
}
```

Zajmuje się ona także ustaleniem liczby wątków GPU, które mają zostać wykorzystane w procesie sortowania. Długość sortowanej tablicy jest przy tym wykorzystywana do dynamicznego wyznaczenia odpowiedniej liczby bloków wątków zaprzęgniętych do realizacji zadania. Założono, że każdy taki blok składać się będzie z 32 wątków, z których to każdy będzie odpowiadać za wykonanie jednego z porównań w fazie *even* oraz *odd* algorytmu.

Listing 6: Ustalenie liczby wątków w funkcji wykonwanej po stronie hosta

```
void sorting::GpuOddEvenSort(std::vector<int>& arr) {
    //...
    int threads = 32;
    int blocks = (int)ceil(half / (double)threads);
    // ...
}
```

Wątki realizujące porównania w ramach obu faz algorytmu *Odd-even* uruchamiane są zawsze $\frac{n}{2}$ razy. Jest to pewne odstępstwo od klasycznej implementacji tegoż algorytmu, który zakłada, że zakończenie sortowania nastąpi w po pierwszej iteracji, w której w obu fazach nie została wykonana żadna zamiana elementów. Takie działanie wymaga jednak istnienia współdzielonej przez wszystkie wątki zmiennej, która wskazywałaby, że żadna zamiana nie została wykonana. Taka zmienna musiałaby być w każdej iteracji pętli przekopiowywana z pamięci globalnej GPU do pamięci głównej, w celu jej odczytania, co wiązałoby się z niezwykle dużym narzutem czasowym. Z tego względu zrezygnowano z takiego rozwiązania i postanowiono zawsze wykonywać maksymalną (patrz sekcja 1.2.4) liczbę faz algorytmu.

Listing 7: Pętla uruchamiająca wątki działające na GPU w funkcji wykonwanej po stronie hosta

```
void sorting::GpuOddEvenSort(std::vector<int>& arr) {
    // ...
    for (int i = 0; i < half; i++)
    {
        sorting::Even<<<blocks, threads>>> (d_arr, arr.size());
        sorting::Odd<<<blocks, threads>>>(d_arr, arr.size());
        cudaDeviceSynchronize();
    }
```

```
9
10      // ...
11  }
```

3.2.3 Jądra `Even()` oraz `Odd()`

Wątki wykonujące jądra `Even()` oraz `Odd()` wykonują jednocześnie porównania w ramach wszystkich par dla odpowiednio faz *even* oraz *odd* algorytmu. Rozpoczynają one swoje działanie od ustalenia indeksu pary, w ramach której mają wykonać porównanie. Wykorzystują do tego swój identyfikator w ramach bloku wątków, do którego należą, a także identyfikator tegoż bloku. Możliwym jest przy tym, że do posortowania tablicy zostanie przypisanych zbyt wiele wątków. Działanie takich nadmiarowych wątków jest kończone przedwcześnie dzięki przeprowadzanej walidacji wyliczanego indeksu.

Listing 8: Jądro wykonywane przez wątki GPU, realizujące fazę *odd*

```
__global__ void sorting::Odd(int* arr, int length) {
    int index = 2 * (blockIdx.x * blockDim.x + threadIdx.x) + 1;
    if (index >= length - 1) return;

    if (arr[index] > arr[index + 1])
    {
        int tmp = arr[index];
        arr[index] = arr[index + 1];
        arr[index + 1] = tmp;
    }
}
```

Listing 9: Jądro wykonywane przez wątki GPU, realizujące fazę *even*

```
__global__ void sorting::Even(int* arr, int length) {
    int index = 2 * (blockIdx.x * blockDim.x + threadIdx.x);
    if (index >= length - 1) return;

    if (arr[index] > arr[index + 1])
    {
        int tmp = arr[index];
        arr[index] = arr[index + 1];
        arr[index + 1] = tmp;
    }
}
```

3.3 Optymalizacja algorytmu *Odd-even* (GPU)

Po wykonaniu wstępnej implementacji algorytmu *Odd-even* przeznaczonego do wykonywania na GPU nastąpiło przejście do prób jego optymalizacji przy pomocy narzędzia *NVIDIA Nsight Compute*. Wykonano łącznie 2 iteracje jego optymalizacji w wyniku czego powstały dwie wersje kolejne wersje implementacji algorytmu:

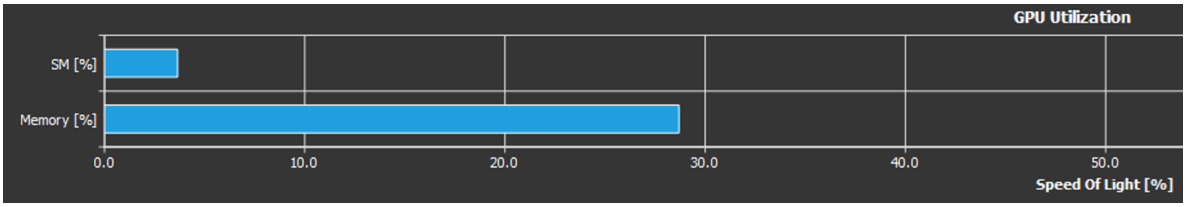
- **V1:** Implementacja uzyskana w wyniku pierwszej iteracji optymalizacji. Została ona dokładnie przebadana w ramach etapu II procedury pomiarowej opisanej w sekcji 2.1.2.
- **V2:** Implementacja uzyskana w wyniku drugiej iteracji optymalizacji. Została ona jedynie pobieżne przebadana w celu uzyskania przybliżonej informacji o zyskach z przeprowadzonych zmian.

Opisy iteracji optymalizacji w ramach których uzyskano wersje *V1* i *V2* zamieszczono odpowiednio w sekcjach 3.3.1 oraz 3.3.2.

3.3.1 Pierwsza iteracja optymalizacji

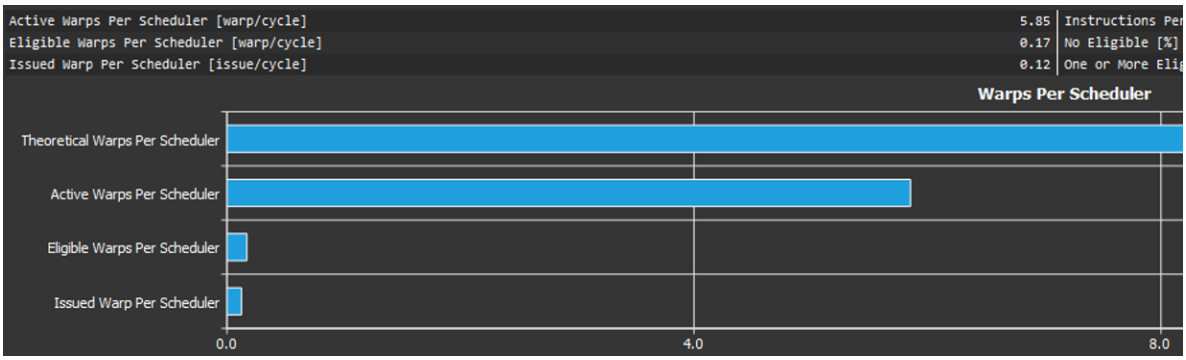
Po przebadaniu wstępnej wersji implementacji przez narzędzie *Nsight* uzyskano raport wskazujący na występujące w niej problemy wpływające negatywnie na wydajność.

Pierwsza część raportu dotyczyła wysokopoziomowego przeglądu procentowego wykorzystania zasobów GPU w odniesieniu do teoretycznego maksimum. Dowiedziano się z niego, że rozważana implementacja wykorzystuje 28,69% zasobów pamięciowych i jedynie 3,65% jednostek obliczeniowych (rys. 5). Narzędzie wskazało, że wąskim gardłem powodującym tak niskie wykorzystanie zasobów jest zbyt mała liczba bloków w siatce wątków.



Rysunek 5: Wykres z fragmentu raportu z narzędzia *Nsight*, wskazujący na wykorzystanie zasobów obliczeniowych i pamięciowych GPU przez wstępną implementacje algorytmu *Odd-even*

Innym wykryty przy pomocy narzędzia *Nsight* problem dotyczył efektywności działania schedulerów sterujących harmonogramem wykonywania instrukcji przez jednostki wykonawcze procesora graficznego. Każdy scheduler w teorii może rozdysponować 2 instrukcje w każdym cyklu zegara. Jednakże w rozważanej implementacji rozdysponowanie jednej instrukcji zajmuje mu aż 8 cykli zegara przez co zasoby dostępne sprzętowe nie są efektywnie wykorzystywane. Dalsza część raportu (rys. 6) wskazała bowiem, że średnio dla każdego schedulera wykorzystywane (ang. *active*) jest jedynie 5,85 z 16 podlegających mu warpów. Co więcej w każdym cyklu jedynie 0,17 warpa jest gotowe (ang. *eligble*) do otrzymania kolejnych instrukcji, czyli nie jest zablokowane (ang. *stalled*) synchronizacją lub dostępem do pamięci globalnej urządzenia.



Rysunek 6: Wykres z fragmentu raportu z narzędzia *Nsight* dla wstępnej implementacji, dotyczący efektywności działania schedulerów

W odpowiedzi na znalezione problemy wprowadzono w implementacji niezbędne zmiany. Pierwszą z nich było połączenie jąder `Odd()` oraz `Even()` w jedno jądro `OddEven()`, które jest w stanie wykonywać obie fazy algorytmu. To jaka faza ma być obecnie wykonywana jest decydowane przy pomocy wartości parametru `phase`. Tak wykonane złączenie wymusiło także zmianę działania pętli uruchamiającej jądra w funkcji hosta `GpuOddEvenSort()`.

Listing 10: Jądro `OddEven()` powstałe w ramach optymalizacji po złączeniu jąder `Odd()` oraz `Even()`

```
__global__ void OddEven(int* arr, int array_length, int phase) {
    int pair_index = 2 * (blockIdx.x * blockDim.x + threadIdx.x) + phase;
    if (pair_index >= array_length - 1) return;

    int first = arr[pair_index];
    int second = arr[pair_index + 1];

    if (first > second)
    {
        arr[pair_index] = second;
```

```
        arr[pair_index + 1] = first;
    }
}
```

Inna zmiana dotyczyła sposobu wyznaczania liczby bloków wątków używanych w sortowaniu, która od tego momentu jest wykonywana przez funkcję `CalculateThreadsBlocksAmount()` wykonywanej po stronie hosta. W niej liczba bloków oraz liczba bloków w wątku obliczania jest dynamicznie w oparciu o dane na temat dostępnych zasobów sprzętowych takich jak: rozmiar wrapów, maksymalna liczba bloków wątków na procesor strumieniowy czy maksymalna liczba wątków na blok. Chciano w ten sposób osiągnąć efektywniejsze wykorzystanie zasobów procesora graficznego.

Listing 11: Nowa funkcja wyznaczająca liczbę wątków i bloków używanych w obliczeniach, powstała w ramach optymalizacji

```
void CalculateThreadsBlocksAmount(int& threads, int& blocks, int length)
{
    cudaDeviceProp deviceProp;
    cudaGetDeviceProperties(&deviceProp, 0);

    const int threadsAmountMin = deviceProp.warpSize;
    const int blocksPerMultiMax = deviceProp.maxBlocksPerMultiProcessor;
    const int multiMax = deviceProp.multiProcessorCount;
    const int maxThresPerBlock = deviceProp.maxThreadsPerBlock;

    blocks = multiMax * blocksPerMultiMax;
    threads = (
        length / (float)blocks < threadsAmountMin
        ? threadsAmountMin
        : RoundUpToMultiple(length / (float)blocks, threadsAmountMin)
    );

    if (threads > maxThresPerBlock)
    {
        threads = maxThresPerBlock;
        blocks = std::ceill(length / (float)threads);
    }
}
```

Ostatnią dokonaną zmianą jest usunięcie konieczności synchronizowania i sekwencyjnego wywoływania jąder w ramach każdej iteracji głównej pętli algorytmu wykonywanej po stronie hosta. Zamiast tego jądra przekazywane są do tzw. strumienia CUDA (`cudaStream_t`, który pozwala na ich asynchronicznie wywoływanie. Takie rozwiązanie pozwala zaś na potencjalne nałożenie się na siebie czasu spędzanego na właściwych obliczeniach z czasem przeznaczonym na operacje pamięciowe.

Listing 12: Zastąpienie sekwencyjnego wywoływnaia jąder wywoływaniem przy pomocy strumienia CUDA, zastosowane w wyniku optumalizacji

```
void sorting::GpuOddEvenSort(std::vector<int>& arr)
{
    int* deviceArr;
    cudaMalloc(&deviceArr, arr.size() * sizeof(int));
    cudaMemcpy(deviceArr, arr.data(), arr.size() * sizeof(int), cudaMemcpyHostToDevice);

    int blocks, threads;
    CalculateThreadsBlocksAmount(threads, blocks, std::ceill(arr.size() / 2.0));

    cudaStream_t stream;
    cudaStreamCreate(&stream);

    for (int i = 0; i < arr.size(); i++)
    {
        OddEven<<<blocks, threads, 0, stream>>>(deviceArr, arr.size(), i % 2);
    }
    cudaMemcpy(arr.data(), deviceArr, arr.size() * sizeof(int), cudaMemcpyDeviceToHost);

    cudaFree(deviceArr);
    cudaStreamDestroy(stream);
```

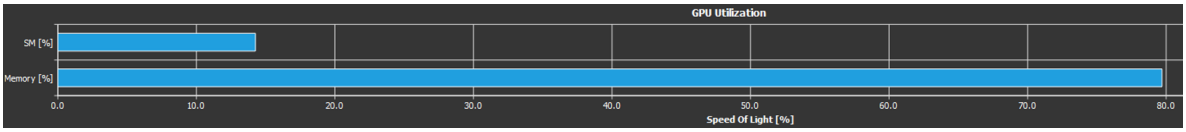
```
21 }
```

Tak zmieniona implementacja *Odd-even* sort została poddana pełnym (rozplanowanym w sekcji 2.1.2) pomiarom zależności $t(n)$, których wyniki zamieszczono w sekcji 4.2. W tejże sekcji porównano również efektywność implementacji *V1* z implementacją wstępną.

3.3.2 Druga iteracja optymalizacji

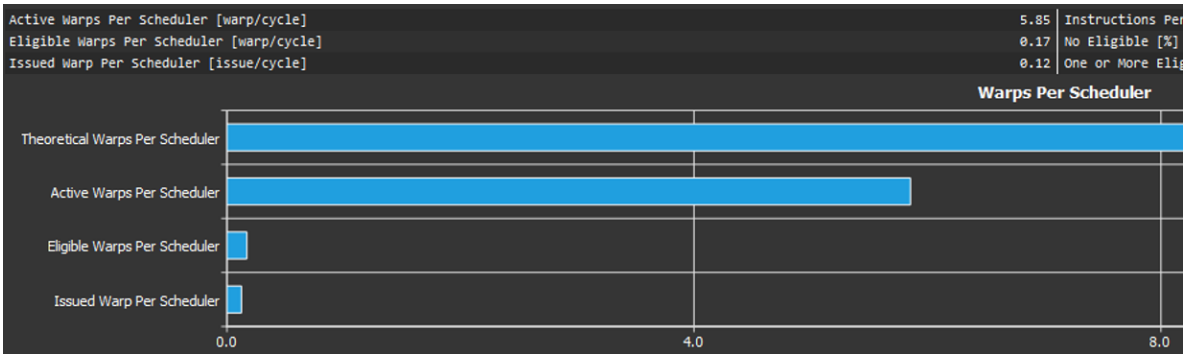
W drugiej iteracji optymalizacji również posłużono się narzędziem *Nsigh*, które tym razem użyto do przeanalizowania wersji *V1* implementacji.

Fragment raportu (rys. 7) z narzędzia tyczący się wykorzystania zasobów sprzętowych GPU wykazał pewną poprawę względem implementacji wstępnej. Okazało się bowiem, że implementacja *V1* wykorzystuje już 79,7% zasobów pamięciowych oraz 14,25% zasobów obliczeniowych. Podążając za radami narzędzia za niepokojącą uznano jednak zważalną dominację użycia zasobów pamięciowych nad użyciem zasobów obliczeniowych. Wskazuje ona na fakt, że wydajność rozważanej implementacji cierpi przede wszystkim ze względu na dostęp do pamięci, a nie ze względu na złożoność i intensywność wykonywanych obliczeń. Proponowanymi przez *Nsight* rozwiązaniami są w takim wypadku próby łączenia obliczeń wykonywanych w aktualnie w ramach wielu jąder (ang. *kernel fusion*) lub zwiększenie efektywności koalescencji pamięci urządzenia przez wątki.



Rysunek 7: Wykres z fragmentu raportu z narzędzia *Nsight*, wskazujący na wykorzystanie zasobów obliczeniowych i pamięciowych GPU przez implementacje *V1* algorytmu *Odd-even*

W przypadku efektywności schedulerów także widoczna jest poprawa względem implementacji bazowej. Dla *V1* bowiem rozdysponowanie instrukcji nie zajmuje już 8 a 6,2 cykle zegara, liczba wykorzystywanych warpów wzrosła do 9,02 z 16 a średnia liczba warpów gotowych w każdym cyklu do otrzymania kolejnych instrukcji zwiększyła się do wartości 0,23 (rys. 8). Wciąż nie jest to jednak wynik bliski idealnemu.



Rysunek 8: Wykres z fragmentu raportu z narzędzia *Nsight* dla implementacji *V1*, dotyczący efektywności działania schedulerów

W celu rozwiązania znalezionych problemów ponownie wprowadzono zmiany w implementacji. Tym razem dotyczyły one jedynie jądra `OddEven()`. Najistotniejszym elementem wprowadzonych poprawek było wykorzystanie pamięci współdzielonej GPU (ang. *shared memory*) do buforowania wykorzystywanych podczas sortowania elementów tablicy. Miało poskutkować zredukowaniem liczby odwołań do wolnej, globalnej pamięci urządzenia.

Listing 13: Zmiany w jądrze `OddEven()`, kótre przystosowują je do użycia pamięci współdzielonej GPU

```
1 __global__ void OddEven(int* arr, int length, int phase, int threadsAmount) {
```

```
    extern __shared__ int sharedMem[];

    int threadIndex = threadIdx.x;
    int globalIndex = blockIdx.x * blockDim.x + threadIdx.x;
    if (
        globalIndex + 1 >= length
        || (blockIdx.x != 0 && threadIndex == 0 && globalIndex % 2 != phase)
    ) {
        return;
    }

    sharedMem[threadIndex] = arr[globalIndex];

    bool isBlockEdge = (
        globalIndex % 2 == phase && (globalIndex + 2 >= length || threadIndex + 1 == threadsAmount)
    );

    if (isBlockEdge)
    {
        sharedMem[threadIndex + 1] = arr[globalIndex + 1];
    }
    __syncthreads();

    if (globalIndex % 2 == phase)
    {
        int current = sharedMem[threadIndex];
        int next = sharedMem[threadIndex + 1];
        if (current > next)
        {
            sharedMem[threadIndex] = next;
            sharedMem[threadIndex + 1] = current;
        }
    }
    __syncthreads();

    arr[globalIndex] = sharedMem[threadIndex];
    if (isBlockEdge)
    {
        arr[globalIndex + 1] = sharedMem[threadIndex + 1];
    }
}
```

Ze względu na ograniczony czas przeprowadzono jedynie pobieżne pomiary zależności $t(n)$, jedynie częściowo zgodne z procedurą opisaną w sekcji 2.1.2. Ich wyniki w formie tabelarycznej oraz graficznej zamieszczono w sekcji 4.2.2.

3.4 Implementacja algorytmu *Bitonic* (CPU)

3.4.1 Struktura implementacji

Implementacja algorytmu *Bitonic* w wersji możliwej do wykonania na CPU składa się z następujących elementów:

1. `CpuBitonicSort()`: Punkt wejścia do algorytmu.
2. `bitonicSortRecurrence()`: Funkcja realizująca sortowanie *Bitonic* w klasyczny, rekurencyjny sposób.
3. `bitonicSortThreaded()`: Funkcja realizująca sortowanie *Bitonic* zastępując rekurencyjne wywołania uruchomieniem wątków.
4. `bitonicMerge()`: Funkcja realizująca proces łączenia pod-tablic.
5. `comparePairs()`: Funkcja realizująca proces sortowania elementów w ramach łączenia pod-tablic.

3.4.2 Funkcja `CpuBitonicSort()`

Główna funkcja realizująca algorytm *Bitonic* wykonywany na CPU, która stanowi punkt wejścia procesu sortowania. Na podstawie liczby dostępnych wątków sprzętowych oraz długości tablicy określa ona liczbę podziałów (`divisions_left`), w ramach których rekurencyjne, działające w ramach tego samego wątku, wywołania przetwarzań pod-tablic zostaną zastąpione uruchomieniami wątków realizujących takie przetwarzanie. Pozwala to na efektywne i pełne wykorzystanie dostępnych zasobów sprzętowych w procesie sortowania.

Listing 14: Główna funkcja i punkt wejścia do algorytmu *Bitonic* sort wykonywanego na CPU

```
void sorting::CpuBitonicSort(std::vector<int> &arr) {
    int divisions_left = std::min(
        std::log2(std::thread::hardware_concurrency()) + 1, std::log2(arr.size() - 1)
    );
    bitonicSortThreaded(arr, 0, arr.size(), 1, divisions_left);
}
```

3.4.3 Funkcja `bitonicSortRecurrence()`

Funkcja realizująca sortowanie pod-tablicy sekwencyjnie w tradycyjny, rekurencyjny sposób. Jeżeli aktualnie przetwarzana pod-tablica nie jest 1-elementowa, to funkcja dokonuje jej ponownego podziału i uruchamia rekurencyjnie przetwarzanie tak powstałych pod-tablic. Następnie następuje złączenie i odpowiednie posortowanie (wyznaczane przez zmienną `sort_direction`) tychże pod-tablic.

Listing 15: Kod funkcji realizującej rekurencyjne sortowania pod-tablic

```
void bitonicSortRecurrence(std::vector<int>& a, int start, int sequence_size, int sort_direction) {
    if (sequence_size == 1)
        return

    int split_point = sequence_size / 2;
    bitonicSortRecurrence(a, start, split_point, 1);
    bitonicSortRecurrence(a, start+split_point, split_point, 0);
    bitonicMerge(a, start, sequence_size, sort_direction);
}
```

3.4.4 Funkcja `bitonicSortThreaded()`

Funkcja realizująca sortowanie pod-tablicy wielowątkowo. Jeżeli aktualnie przetwarzana pod-tablica nie jest 1-elementowa, to funkcja dokonuje jej ponownego podziału i w zależności od ilości podziałów, w których możliwe jest uruchomienie wątków (ograniczane wartością zmiennej `divisions_left`) przetwarza powstałe pod-tablice na różne sposoby. Jeżeli nie są dostępne już żadne podziały, w których można uruchomić kolejne wątki, to przetwarzanie pod-tablic odbywa się w tradycyjny, sekwencyjny i rekurencyjny sposób. W przeciwnym wypadku uruchamiane są kolejne dwa nowe wątki, z których pierwszy przetwarza „lewą" pod-tablicę a drugi „prawą" pod-tablicę. Po tym obecny wątek jest usypiany w oczekiwaniu na wyniki nowo stworzonych wątków.

Po zakończeniu przetwarzani pod-tablic i niezależnie od wybranego sposobu, następuje ich sekwencyjne złączenie.

Listing 16: Kod funkcji realizującej wielowątkowe sortowania pod-tablic

```
void bitonicSortThreaded(
    std::vector<int>& array, int start, int sequence_size, int sort_direction, int divisions_left
) {
    if (sequence_size == 1)
        return

    int split_point = sequence_size / 2;
    if (divisions_left > 0) {
        std::thread aT (std::bind(
            bitonicSortThreaded,
```

```
            std::ref(array),
            start,
            split_point,
            1,
            divisions_left - 1
        ));
        std::thread bT (std::bind(
            bitonicSortThreaded,
            std::ref(array),
            start+split_point,
            split_point,
            0,
            divisions_left - 1
        ));

        aT.join();
        bT.join();
    }
    else {
        bitonicSortRecurrence(array, start, split_point, 1);
        bitonicSortRecurrence(array, start+split_point, split_point, 0);
    }
    bitonicMerge(array, start, sequence_size, sort_direction);
}
```

3.4.5 Funkcja `bitonicMerge()`

Funkcja realizująca sekwencyjne łącznie dwóch pod-tablic zgodnie z klasycznym działaniem algorytmu *Bitonic*. Na początku dokonuje ona porównania i posortowania odpowiednich par elementów w złączonej pod-tablicy zgodnie przyjętym dla niej kierunkiem sortowania (`sort_direction`). Następnie sortuje ona rekurencyjni wszystkie pod-tablice zgodnie z tymże kierunkiem.

Listing 17: Kod funkcji realizującej łączenie pod-tablic

```
void bitonicMerge(std::vector<int>& a, int start, int sequence_size, int sort_direction) {
    if (sequence_size == 1)
        return;

    int split_point = sequence_size/2;
    comparePairs(a, start, split_point, sort_direction);
    bitonicMerge(a, start, split_point, sort_direction);
    bitonicMerge(a, start+split_point, split_point, sort_direction);
}
```

3.4.6 Funkcja `comparePairs()`

Funkcja realizująca porównywanie ze sobą par elementów odpowiednich dla aktualnie łączonych pod-tablic. Porównania są w jej ramach wykonywane sekwencyjnie.

Listing 18: Kod funkcji realizującej porównania i sortowania par elementów w ramach łączenia pod-tablic

```
tatic inline void comparePairs(
    std::vector<int>& arr, const int start, const int pair_distance, const int sort_direction
) {
    for (int i = start; i < start + pair_distance; i++)
        if (sort_direction == (arr[i] > arr[i+pair_distance]))
            std::swap(arr[i],arr[i+pair_distance]);
}
```

3.5 Wstępna implementacja algorytmu *Bitonic* (GPU)

3.5.1 Struktura implementacji

Wstępna implementacja algorytmu *Bitonic*, która jest przeznaczona do wykonywania na GPU składa się z dwóch elementów:

1. `GpuBitonicSort()`: Funkcja wykonywana po stroni hosta. Pełni rolę punktu wejścia i koordynatora procesu sortowania.

2. `comparePair()`: Funkcja Jądro z kodem do wykonywania na procesorze graficznym. Wykonuje ono pojedynczy etap sortowania dla zadanej głębokości podziału sortowanej tablicy.

3.5.2 Funkcja `GpuBitonicSort()`

Funkcja `GpuBitonicSort()` odpowiada za uruchamianie jądra z kodem, który ma zostać wykonany na GPU a także za transfery wykonywane między pamięcią główną a pamięcią globalną GPU (inne rodzaje pamięci urządzenia nie zostały wykorzystane). Najważniejszym wykonywanym przez nią transferem jest ten dotyczący sortowanej tablicy.

Listing 19: Zarządzanie pamięcią GPU przez funkcje hosta koorydującą realizację algorytmu *Bitonic* na GPU

```
void sorting::GpuBitonicSort(std::vector<int>& arr) {
    int *device_array;
    size_t size = arr.size() * sizeof(int);

    cudaMalloc(&device_array, size);
    cudaMemcpy(device_array, arr.data(), size, cudaMemcpyHostToDevice);

    // ...

    cudaMemcpy(arr.data(), device_array, size, cudaMemcpyDeviceToHost);
    cudaFree(device_array);
}
```

Odpowiedzialnością funkcji jest również ustalenie liczby struktury siatki wątków, która będzie wykonywać sortowanie. Założono, że wspomniana siatka będzie składać się z bloków wątków, z których każdy zawsze grupować będzie 512 wątków (wartość `THREADS_PER_BLOCK`). Liczba bloków wątków jest ustalana dynamicznie na podstawie rozmiaru tablicy tak, aby ilość uruchomionych wątków była większa bądź równa maksymalnej liczbie wykonywanych w algorytmie porównań. W ten sposób każde porównanie zawsze zostanie przypisane do jednego z wątków.

Listing 20: Zarządzanie ilością używanych w obliczeniach wątków GPU przez funkcje hosta koorydującą realizację algorytmu *Bitonic* na GPU

```
void sorting::GpuBitonicSort(std::vector<int>& arr) {
    // ...

    const int THREADS_PER_BLOCK = 512;
    dim3 blocks((arr.size() + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK);
    dim3 threads(THREADS_PER_BLOCK);

    // ...
}
```

Najważniejszym fragmentem całości funkcji są dwie pętle w ramach których wykonywane jest właściwe sortowanie przy pomocy algorytmu *Bitonic*. Zadaniem tych pętli jest odzwierciedlenie ciągu porwań i sortowań, które odbywają się w ramach łączenia pod-tablic wydzielonych w ramach algorytmu (patrz sekcja 1.2.5, zwłaszcza rysunek 11). Pętla zewnętrzna koordynuje aktualną długością sortowanej sekwencji (`sequence_size`), a pętla wewnętrzna koordynuje odległością między indeksami porównywanych elementów (`pair_distance`). Dla tychże dwóch wartości funkcja uruchamia wątki wykonujące jądro `comparePair()` (więcej o nim w sekcji 3.5.3). Następnie przed przejściem do kolejnej iteracji następuje odczekanie na zakończenie pracy wszystkich uruchomionych wątków.

Listing 21: Główna pętla wywołujące jądra GPU wykonujące sortowania odpowiednich par w ramach realizacji algorytmu *Bitonic* na GPU

```
void sorting::GpuBitonicSort(std::vector<int>& arr) {
    // ...

    int pair_distance, sequence_size;
    for (sequence_size = 2; sequence_size <= arr.size(); sequence_size <<= 1) {
        for (
            pair_distance = sequence_size >> 1; pair_distance > 0; pair_distance = pair_distance >> 1
        ) {
            comparePairs<<<blocks, threads>>>(device_array, pair_distance, sequence_size);
            cudaDeviceSynchronize();
        }
    }

    // ...
}
```

3.5.3 Jądro `comparePair()`

Wątek wykonujący jądro odpowiada za wykonanie porównania pary elementów odpowiedniej dla aktualnego (wyznaczanego przez parametry `pair_distance` i `sequence_size`) kroku algorytmu *Bitonic*.

Listing 22: Wyznaczanie pierwszego elementu porównywanej pary w jądrze realizującym sortowanie pary dla algorytmu *Bitonic*

```
__global__ void comparePair(int *array, int pair_distance, int sequence_size, int array_size) {
    unsigned int first = threadIdx.x + blockDim.x * blockIdx.x;
    // ...
}
```

Na początku wątek na podstawie swojego identyfikatora w ramach bloku określa indeks pierwszego z porównywanych elementów. Jako, że wątków przypisanych do sortowania może być więcej niż aktualnie wykonywanych porównań, to następuje walidacja, czy aktualny wątek nie jest nadmiarowy. Jeżeli taki się okaże to jego działanie zostaje zakończone. Po walidacji wyznaczany jest indeks drugiego z porównywanych elementów przy pomocy operacji `XOR` wykonanej na pierwszym elemencie porównania i dystansie jaki dzieli go od drugiego elementu. Ta operacja zastępuje tutaj arytmetyczną sumę $(+)$ indeksu pierwszego elementu z dystansem do drugiego elementu, która jest od niej wolniejsza.

Listing 23: Walidacja pierwszego i wyznaczanie drugiego elementu porównywanej pary w jądrze realizującym sortowanie pary dla algorytmu *Bitonic*

```
__global__ void comparePair(int *array, int pair_distance, int sequence_size, int array_size) {
    // ...
    if (first >= array_size)
        return;
    unsigned int second = first ^ pair_distance;

    // ...
}
```

Kolejnym krokiem jest określenie czy obecny wątek nie wykonuje redundantnego porównania i zakończenie go jeżeli w rzeczywiście tak jest. Taka sytuacja jest bowiem możliwa do zaistnienia w momencie kiedy wątek ma wykonać porównanie pary indeksów (j, i) po tym jak inny wątek wykonał już porównanie pary indeksów (i, j).

Listing 24: Określenie czy potencjalne porównanie jest redundantne w jądrze realizującym sortowanie pary dla algorytmu *Bitonic*

```
__global__ void comparePair(int *array, int pair_distance, int sequence_size, int array_size) {
    // ...
```

```
3
4       if (second <= first)
5           return;
6       // ...
7   }
```

Porównanie wykonywane przez wątek wymaga na początku określenia kierunku sortowania pod-tablicy, na której elementach wątek aktualnie działa. Wykorzystano przy tym efektywną operację logiczną `AND` wykonywaną na indeksie pierwszego elementu oraz rozmiarze pod-tablicy, do której należy porównywana para (`sequence_size`). Jest to efektywniejszy od tradycyjnego rozgałęzienia strukturą `if` sposób na sprawdzenie, czy aktualnie sprawdzana para znajduje się w „prawej" (sortowanej malejąco), czy też „lewej" (sortowanej rosnąco) pod-tablicy.

W końcu wykonywane jest właściwe porównanie elementu w ramach pary i ewentualna zamiana ich kolejności.

Listing 25: Określenie kierunku sortowania i wykonanie porównania elementów w jądrze realizującym sortowanie pary dla algorytmu *Bitonic*

```
__global__ void comparePair(int *array, int pair_distance, int sequence_size, int array_size) {
    // ...

    const int descending = first & sequence_size;
    const bool first_greater = array[first] > array[second];
    if ((!descending && first_greater) || (descending && !first_greater)) {
        int temp = array[first];
        array[first] = array[second];
        array[second] = temp;
    }

    // ...
}
```

3.6 Optymalizacja algorytmu *Bitonic* (GPU)

Niestety ze względu na ograniczony czas przeznaczony na realizację projektu nie udało się przejść przez proces optymalizacji dla implementacji algorytmu *Bitonic* wykonywanego na GPU.

4 Wyniki pomiarów

Po przeprowadzeniu pomiarów zależności $t(n)$ dla wszystkich implementacji algorytmów *Odd-even* i *Bitonic* przy pomocy zaimplementowanej aplikacji badawczej uzyskano wyniki, które zostały zaprezentowane w formie graficznej jak i tabelarycznej w podsekcjach 4.1 (pomiary sprzed optymalizacji dla GPU) oraz 4.2 (pomiary po optymalizacji dla GPU).

4.0.1 Wykorzystywane symbole

- n - Rozmiar problemu.
- t - Średni, dla danego n, wynik pomiaru czasu w sekundach.
- σ - Względne, wyrażane w procentach, odchylenie standardowe liczone względem danego t.
- S - Względne przyspieszenie implementacji GPU względem implementacji CPU dla danego n oraz algorytmu. Liczone jako iloraz t dla implementacji GPU i t dla implementacji CPU.
- O - Względne przyspieszenie wynikające z optymalizacji implementacji GPU dla danego n oraz algorytmu. Liczone jako iloraz t dla zoptymalizowanej implementacji GPU i t dla niezoptymalizowanej implementacji GPU.

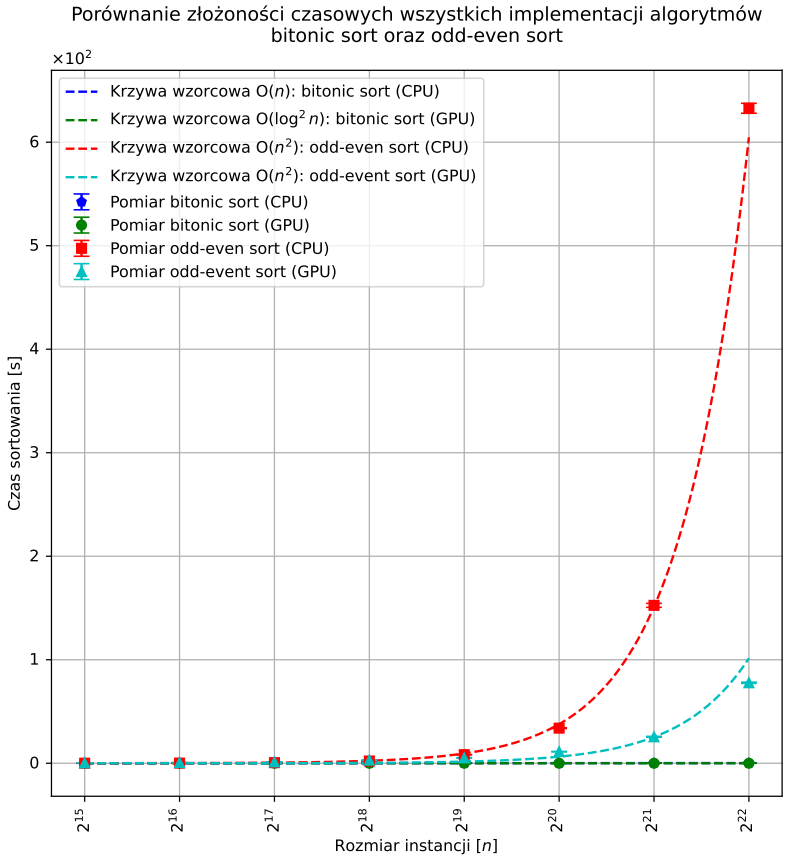
4.1 Wyniki dla wersji przed optymalizacji

4.1.1 Wszystkie algorytmy

Rysunek 9 wyraźnie pokazuje, że algorytmy *Odd-Even* zarówno dla CPU, jak i GPU, okazały się znacznie mniej wydajne w porównaniu z algorytmem *Bitonic*, nawet przy zestawieniu wersji GPU dla *Odd-Even* z wersją CPU dla *Bitonic* – podkreśla to znaczną przewagę wydajności algorytmu *Bitonic*, szczególnie w przypadku większych rozmiarów danych.

Tabela 3: Wyniki pomiarów zależności $t(n)$ dla wszystkich implementacji algorytmów *Odd-even* oraz *Bitonic* przed optymalizacją ich wersji na GPU

n	**Bitonic (CPU)**		**Bitonic (GPU)**		**Odd-even (CPU)**		**Odd-event (GPU)**	
	t	σ	t	σ	t	σ	t	σ
2^{15}	3,90E-03	6,33%	3,61E-03	157,66%	4,40E-02	6,17%	3,36E-01	16,13%
2^{16}	4,79E-03	6,03%	3,49E-03	43,78%	1,62E-01	4,00%	6,68E-01	11,66%
2^{17}	6,42E-03	1,87%	3,77E-03	44,85%	5,90E-01	2,45%	1,29E+00	6,94%
2^{18}	1,07E-02	2,18%	4,47E-03	43,88%	2,18E+00	1,33%	3,00E+00	6,25%
2^{19}	2,08E-02	2,69%	4,19E-03	45,59%	8,29E+00	0,96%	5,14E+00	6,27%
2^{20}	4,18E-02	2,49%	5,45E-03	42,90%	3,39E+01	1,12%	1,12E+01	2,18%
2^{21}	8,66E-02	2,73%	7,49E-03	33,10%	1,53E+02	1,25%	2,55E+01	1,69%
2^{22}	1,87E-01	2,11%	1,00E-02	8,33%	6,33E+02	0,85%	7,78E+01	0,86%



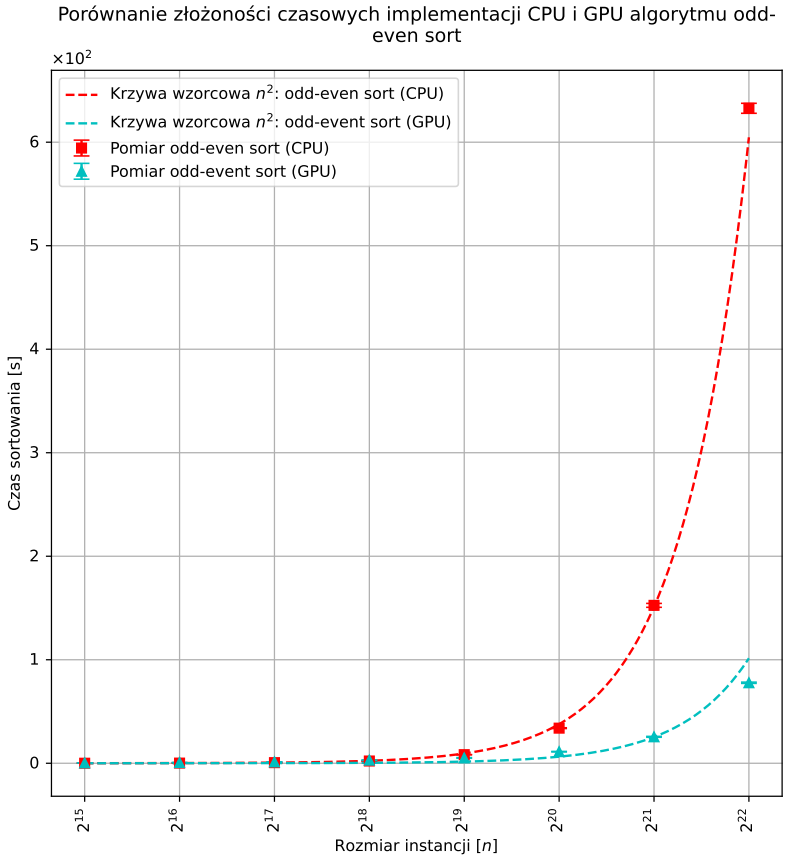
Rysunek 9: Wyniki pomiarów zależności $t(n)$ dla wszystkich implementacji algorytmów *Odd-even* oraz *Bitonic* przed optymalizacją ich wersji na GPU

4.1.2 Algorytm *Odd-even*

Na podstawie rysunku 10 można zauważyć, że wersja algorytmu *Odd-Even* zaimplementowana na GPU zaczyna zauważalnie przewyższać wydajnością wersję na CPU dla instancji danych o rozmiarze większym od 2^{18}, konsekwentnie pogłębiając różnicę na korzyść GPU.

Tabela 4: Porównanie wyników pomiarów zależności $t(n)$ dla implementacji algorytmu *Odd-even* przed optymalizacją jego wersji na GPU

n	**Odd-even (CPU)**		**Odd-event (GPU)**		**S**
	t	σ	t	σ	
2^{15}	4,40E-02	6,17%	3,36E-01	16,13%	0,13
2^{16}	1,62E-01	4,00%	6,68E-01	11,66%	0,24
2^{17}	5,90E-01	2,45%	1,29E+00	6,94%	0,46
2^{18}	2,18E+00	1,33%	3,00E+00	6,25%	0,73
2^{19}	8,29E+00	0,96%	5,14E+00	6,27%	1,61
2^{20}	3,39E+01	1,12%	1,12E+01	2,18%	3,03
2^{21}	1,53E+02	1,25%	2,55E+01	1,69%	5,97
2^{22}	6,33E+02	0,85%	7,78E+01	0,86%	8,14



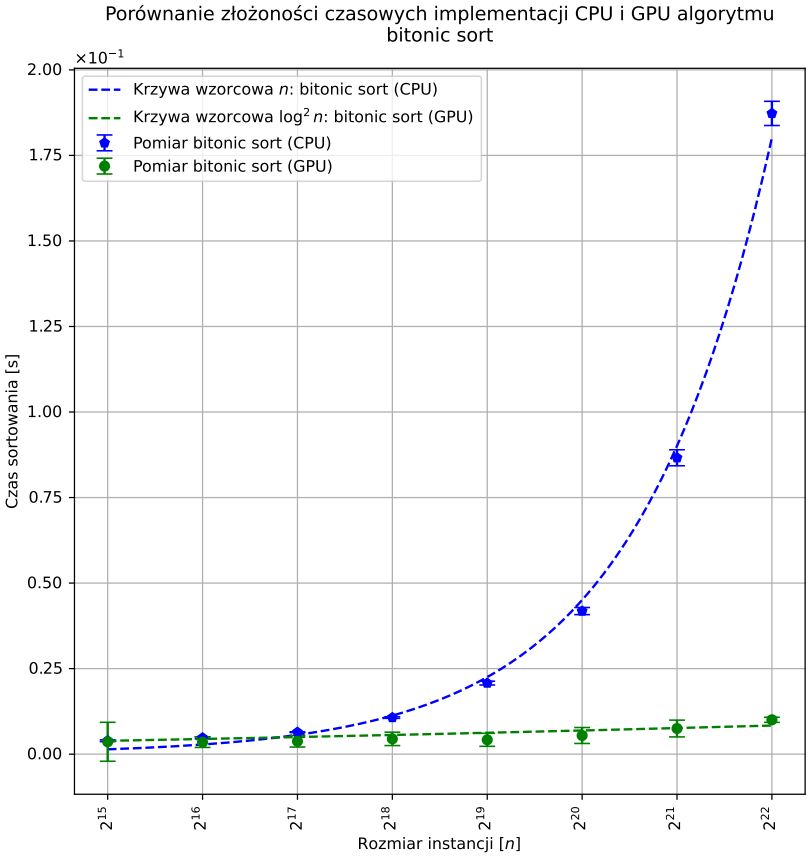
Rysunek 10: Porównanie wyników pomiarów zależności $t(n)$ dla implementacji algorytmu *Odd-even* przed optymalizacją jego wersji na GPU

4.1.3 Algorytm *Bitonic*

Na podstawie rysunku 11 można zauważyć, że wersja algorytmu *Bitonic* zaimplementowana na GPU zaczyna zauważalnie przewyższać wydajnością wersję na CPU dla instancji danych o rozmiarze większym od 2^{17}, konsekwentnie pogłębiając różnicę na korzyść GPU tak jak to miało miejsce w przypadku algorytmu *Odd-Even*.

Tabela 5: Porównanie wyników pomiarów zależności $t(n)$ dla implementacji algorytmu *Bitonic* przed optymalizacją jego wersji na GPU

n	**Bitonic (CPU)**		**Bitonic (GPU)**		S
	t	σ	t	σ	
2^{15}	3,90E-03	6,33%	3,61E-03	157,66%	1,08
2^{16}	4,79E-03	6,03%	3,49E-03	43,78%	1,37
2^{17}	6,42E-03	1,87%	3,77E-03	44,85%	1,70
2^{18}	1,07E-02	2,18%	4,47E-03	43,88%	2,40
2^{19}	2,08E-02	2,69%	4,19E-03	45,59%	4,96
2^{20}	4,18E-02	2,49%	5,45E-03	42,90%	7,67
2^{21}	8,66E-02	2,73%	7,49E-03	33,10%	11,57
2^{22}	1,87E-01	2,11%	1,00E-02	8,33%	18,66



Rysunek 11: Porównanie wyników pomiarów zależności $t(n)$ dla implementacji algorytmu *Bitonic* przed optymalizacją jego wersji na GPU

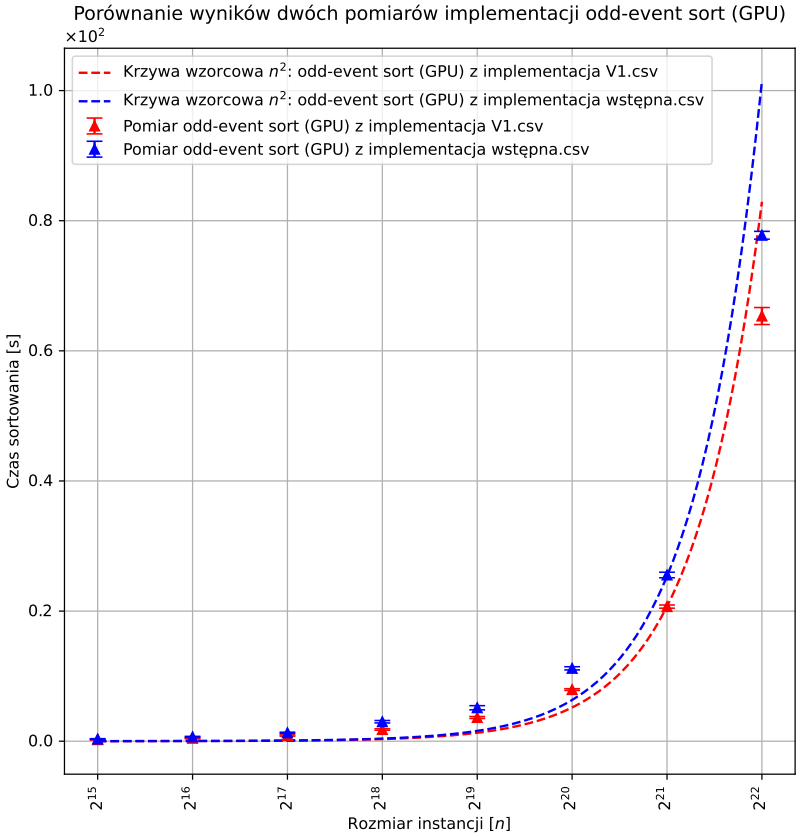
4.2 Wyniki dla wersji po optymalizacji

4.2.1 Algorytm *Odd-even* (implementacja *V1*)

Przyglądając się wynikom zależności $t(n)$ dla implementacji $V1$ algorytmu *Odd-even* można stwierdzić, że pierwsza iteracja optymalizacji tegoż algorytmu zakończyła się pewnym sukcesem. Zarówno na rysunku 12 jak i w tabeli 6 dobrze widoczne są zyski O z zastosowanych zmian (średnio implementacja zoptymalizowana jest 1,4 raza szybsza od bazowej). Widoczne jest jednak, że zyski z zastosowanych optymalizacji maleją wraz ze wzrostem rozmiaru problemu.

Tabela 6: Porównanie wyników pomiarów zależności $t(n)$ dla wstępnej i optymalizowanej ($V1$) implementacji algorytmu *Odd-even*

n	**Odd-even (GPU) Niezoptymalizowane**		**Odd-even (GPU) *V1***		O
	t	σ	t	σ	
2^{15}	3,36E-01	16%	2,34E-01	24%	1,44
2^{16}	6,68E-01	12%	4,50E-01	20%	1,48
2^{17}	1,29E+00	7%	8,73E-01	11%	1,48
2^{18}	3,00E+00	6%	1,80E+00	6%	1,66
2^{19}	5,14E+00	6%	3,64E+00	4%	1,41
2^{20}	1,12E+01	2%	7,94E+00	2%	1,41
2^{21}	2,55E+01	2%	2,07E+01	1%	1,23
2^{22}	7,78E+01	1%	6,54E+01	2%	1,19



Rysunek 12: Porównanie wyników pomiarów zależności $t(n)$ dla wstępnej i optymalizowanej ($V1$) implementacji algorytmu *Odd-even*

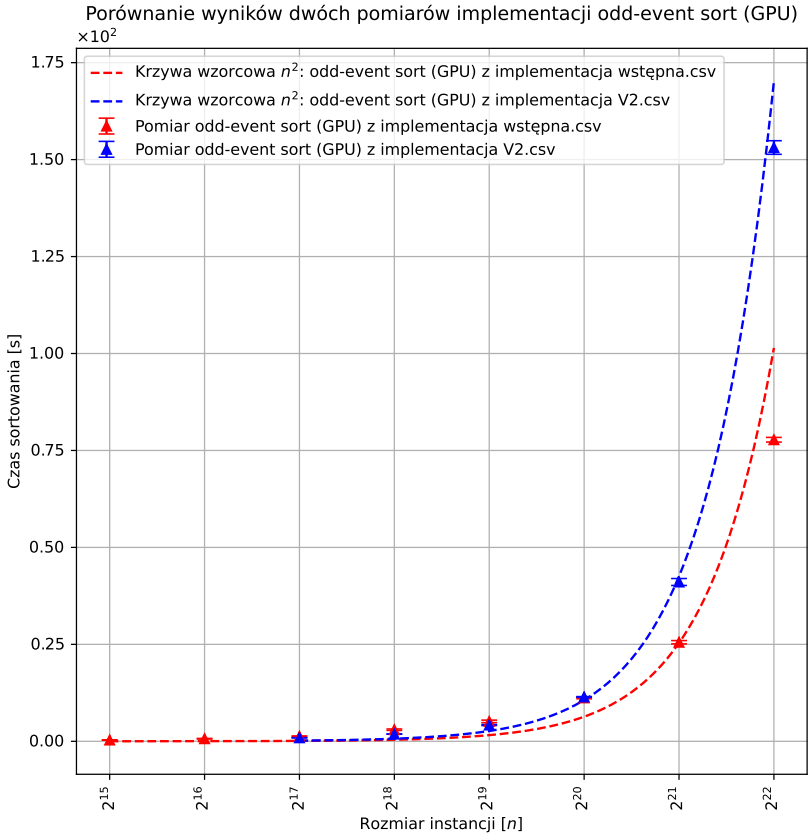
4.2.2 Algorytm *Odd-even* (implementacja *V2*)

Po przeprowadzeniu pobieżnych pomiarów implementacji *V2* opracowano uzyskanie w nim wyniki, które przedstawiono na rysunku 12 i 13 a także w tabeli 7. Widać w nich, że wykonane w ramach drugiej iteracji optymalizacji zmiany nie poprawiły efektywności zgodnie z oczekiwaniami. Zysk O wynikający ze zmian spada bowiem wraz ze wzrostem rozmiaru problemu, a przyglądając się wykresom ze wcześniej wspomnianych rysunków mozna zauważyć, że zależność $t(n)$ dla implementacji $V2$ jest

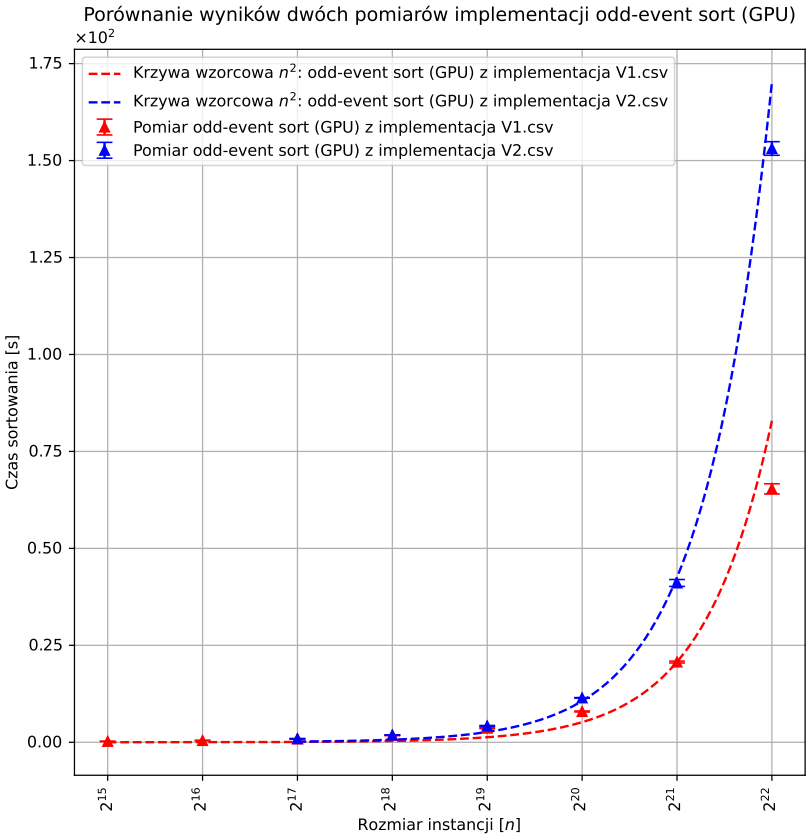
mniej korzystna nie tylko od implementacji $V2$ ale także od implementacji wstępnej.

Tabela 7: Porównanie wyników pomiarów zależności $t(n)$ dla implementacji *V1* oraz *V2* algorytmu *Odd-even*

n	**Odd-even (GPU) Niezoptymalizowane**		**Odd-even (GPU)** $V2$		O
	t	σ	t	σ	
2^{15}	3,36E-01	16%	-	-	-
2^{16}	6,68E-01	12%	-	-	-
2^{17}	1,29E+00	7%	9,12E-01	3%	1,42
2^{18}	3,00E+00	6%	1,87E+00	2%	1,60
2^{19}	5,14E+00	6%	4,20E+00	3%	1,22
2^{20}	1,12E+01	2%	1,14E+01	1%	0,98
2^{21}	2,55E+01	2%	4,11E+01	2%	0,62
2^{22}	7,78E+01	1%	1,53E+02	1%	0,51



Rysunek 13: Porównanie wyników pomiarów zależności $t(n)$ dla implementacji wstępnej oraz zoptymalizowanej (*V2*) algorytmu *Odd-even*



Rysunek 14: Porównanie wyników pomiarów zależności $t(n)$ dla implementacji *V1* oraz *V2* algorytmu *Odd-even*

5 Podsumowanie

Zgodnie z założeniami hipotezy, algorytmy w wersjach na GPU okazały się znacznie szybsze od ich odpowiedników na CPU, szczególnie w miarę wzrostu rozmiaru instancji problemu. Wynika to bezpośrednio z charakterystyki obu jednostek – GPU, zoptymalizowany pod kątem masowych obliczeń równoległych, zyskuje znaczącą przewagę przy większych zbiorach danych.

Interesującym odkryciem było to, że algorytm *Odd-Even* wypadł znacznie gorzej w porównaniu z *Bitonic*, nawet w przypadku porównania wersji GPU dla *Odd-Even* z wersją CPU dla *Bitonic*. Wyniki te podkreślają, że dla większych instancji problemu algorytm *Bitonic* jest lepszym wyborem. Co więcej, nawet bez dodatkowych optymalizacji, algorytm *Bitonic* przewyższał zoptymalizowaną wersję *Odd-Even* na GPU. Wynika to z prostszej struktury algorytmu *Bitonic*, który lepiej nadaje się do równoległego przetwarzania, w przeciwieństwie do *Odd-Even*, który ze względu na swoją charakterystykę ma ograniczenia w zakresie równoległości.

Projekt wyraźnie pokazał, jak istotne jest odpowiednie dobranie jednostki obliczeniowej do charakterystyki problemu, a także w jaki sposób można efektywnie optymalizować kod na GPU. Otworzył również szerokie możliwości do zgłębiania wiedzy na temat pracy z danymi w sposób równoległy.