**POLITECHNIKA OPOLSKA**

**WYDZIAŁ ELEKTROTECHNIKI, AUTOMATYKI I INFORMATYKI**

****

**PROGRAMOWANIE WSPÓŁBIEŻNE I ROZPROSZONE**

*Porównanie algorytmów sortowania*

|  |  |
| --- | --- |
| **Prowadzący:**  dr inż. Artur Pala | **Autor:**  Dawid Garncarek |
|  | |
| Rok akademicki 2024/2025 | |

Spis treści

[**PROGRAMOWANIE WSPÓŁBIEŻNE I ROZPROSZONE** 1](#_Toc200644035)

[1. Cel i zakres projektu 3](#_Toc200644036)

[2. Charakterystyka realizacji 3](#_Toc200644037)

[2.1. Opis algorytmów 3](#_Toc200644038)

[2.2. Zastosowanie i znaczenie 3](#_Toc200644039)

[2.3. Opis użytego środowiska 4](#_Toc200644040)

[3. Programy sortowania 4](#_Toc200644041)

[3.1. Sortowanie bąbelkowe 4](#_Toc200644042)

[3.2. Sortowanie szybkie 6](#_Toc200644043)

[3.3. Sortowanie przez scalanie 8](#_Toc200644044)

[3.4. Sortowanie przez wstawienie 10](#_Toc200644045)

[3.5. Sortowanie przez wybieranie 11](#_Toc200644046)

[4. Scenariusze testowe i metodologia badań 13](#_Toc200644047)

[5. Pomiary Czasowe 14](#_Toc200644048)

[5.1. Sekwencyjnie 14](#_Toc200644049)

[5.2. Nvidia CUDA 14](#_Toc200644050)

[5.3. Threads C++ oraz OpenMP 17](#_Toc200644051)

[6. Wykresy 25](#_Toc200644052)

[7. Prawo Amdahla 30](#_Toc200644053)

[7.1. Ziarnistość problemu 32](#_Toc200644054)

[7.3. Recykling danych 33](#_Toc200644055)

[8. Wnioski 33](#_Toc200644056)

[9. Bibliografia 35](#_Toc200644057)

# 1. Cel i zakres projektu

Celem projektu było zbadanie i porównanie wydajności pięciu klasycznych algorytmów sortowania: Bubble Sort, Quick Sort, Merge Sort, Insertion Sort oraz Selection Sort. Każdy z tych algorytmów został zaimplementowany w trzech wariantach:

* **sekwencyjnie**,
* z wykorzystaniem **wątków C++** (std::thread),
* CUDA Nvidia,
* oraz z użyciem **OpenMP**.

Projekt koncentruje się na analizie, które algorytmy najlepiej wykorzystują możliwości równoległego przetwarzania oraz gdzie leży granica opłacalności zrównoleglenia.

Wybrane algorytmy różnią się złożonością, strukturą działania i sposobem przetwarzania danych, co czyni je idealnymi do porównań w kontekście wielowątkowości.

# 2. Charakterystyka realizacji

Programy zostały napisane w języku **C++**, z wykorzystaniem standardowych bibliotek STL, a także technologii **OpenMP** i **std::thread**. Każdy algorytm ma trzy warianty:

* **wersja podstawowa (sekwencyjna)**,
* **wersja równoległa z OpenMP**, korzystająca z dyrektyw do automatycznego rozdzielania pracy,
* **wersja współbieżna z std::thread**, z ręcznym zarządzaniem wątkami.

Celem zrównoleglenia było skrócenie czasu sortowania, zwłaszcza przy dużych ilościach danych.

## 2.1. Opis algorytmów

* **Bubble Sort** – prosty algorytm porównujący sąsiadujące elementy i zamieniający je miejscami, dopóki cała lista nie będzie uporządkowana.
* **Quick Sort** – wydajny algorytm typu *dziel i zwyciężaj*, który wybiera element pivot i dzieli dane na dwie części, sortując je rekurencyjnie.
* **Merge Sort** – kolejny algorytm *dziel i zwyciężaj*, który dzieli tablicę na połówki, sortuje je osobno, a następnie scala je w jedną posortowaną całość.
* **Insertion Sort** – wstawia każdy element w odpowiednie miejsce już posortowanej części listy.
* **Selection Sort** – wyszukuje najmniejszy element w nieposortowanej części tablicy i przenosi go na początek.

## 2.2. Zastosowanie i znaczenie

Sortowanie jest fundamentem wielu zastosowań w informatyce – od analizy i przetwarzania danych, przez działanie baz danych, aż po systemy operacyjne. Szybkość sortowania ma bezpośredni wpływ na ogólną wydajność tych systemów, dlatego warto sprawdzić, w jakim stopniu można te operacje przyspieszyć, wykorzystując wielowątkowość i równoległość.

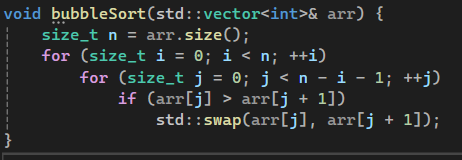
## 2.3. Opis użytego środowiska

|  |  |
| --- | --- |
| **Parametry procesora** | **Rodzina procesorów: AMD Ryzen**  **Seria procesora: 5**  **Taktowanie rdzenia: 3.6 GHz**  **Liczba rdzeni fizycznych: 6 rdzeni**  **Liczba wątków: 12 wątków** |
| **Pamięć RAM** | Pamięć RAM: **32.0 GB**  Szybkość: **3200 MHz** |
| **Dysk** | **SSD ADATA XPG SX8200 PRO 256GB**  Szybkość odczytu: **3350 MB/s**  Szybkość zapisu: **1200 MB/s** |
| **Karta graficzna** | **NVIDIA GeForce GTX 1660 SUPER** |

# 3. Programy sortowania

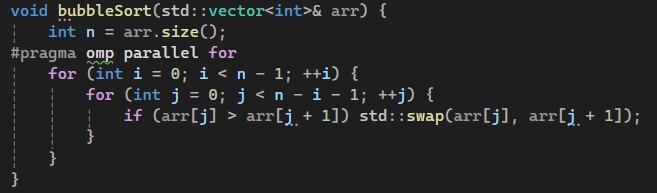
## 3.1. Sortowanie bąbelkowe

Sekwencyjnie:



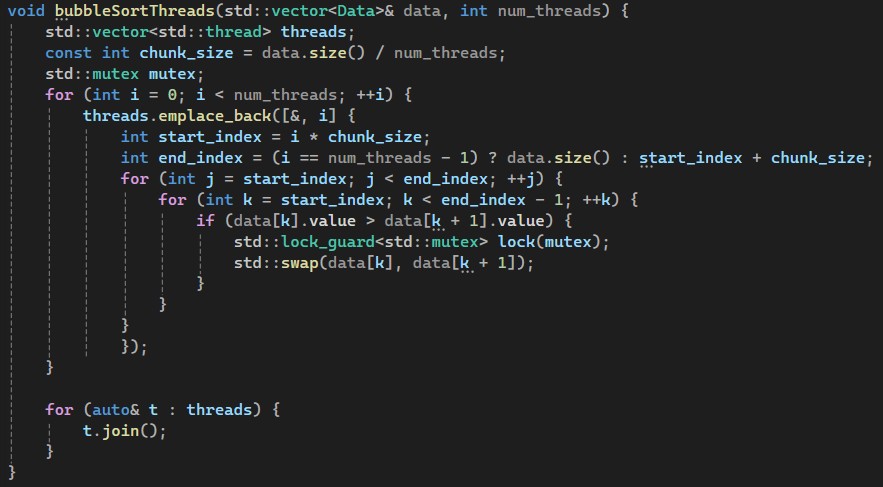
Porównuje sąsiednie elementy i zamienia je miejscami, jeśli są w złej kolejności. Proces powtarza się aż do pełnego posortowania.

OpenMP:



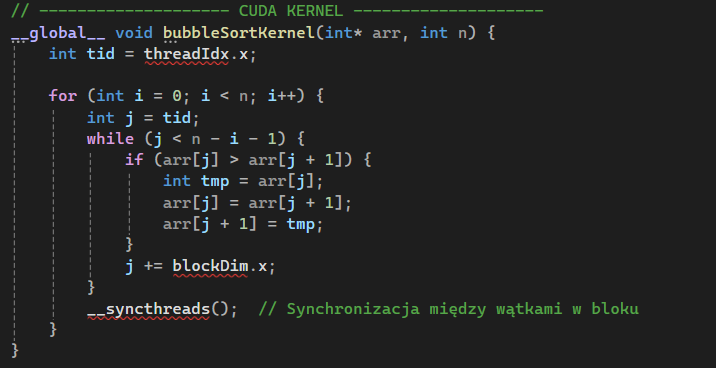
W tym algorytmie zastosowano #pragma omp parallel for, aby zrównoleglić zewnętrzną pętlę, która kontroluje liczbę przebiegów sortowania. Każdy wątek przetwarza fragment danych niezależnie, jednak ze względu na charakter zależności (sąsiednie elementy są zamieniane), może dojść do konfliktów i wyniki mogą być niedeterministyczne. Wymaga ostrożności przy użyciu.

Threads C++:



Dane wejściowe są dzielone na bloki (chunks), z których każdy jest przypisany do osobnego wątku. Każdy wątek wykonuje sortowanie bąbelkowe lokalnie w swoim zakresie. W miejscach, gdzie może dojść do konfliktu podczas zamiany elementów (np. sąsiadujące elementy w różnych blokach), stosuje się mutex, aby zapewnić bezpieczeństwo współbieżne.

CUDA:

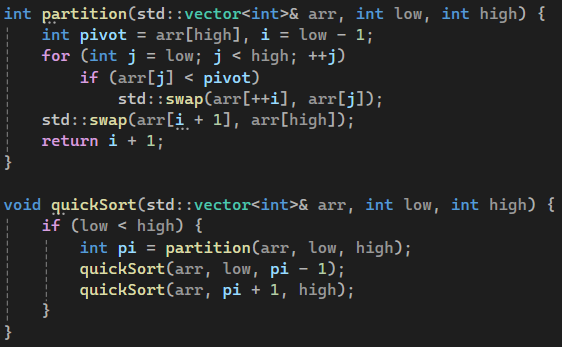


Porównuje sąsiednie elementy i zamienia je miejscami, jeśli są w złej kolejności. Powtarza to aż do całkowitego posortowania. Synchronizacja (\_\_syncthreads()) po każdej iteracji zewnętrznej, żeby zapewnić poprawność porównań.

Algorytm sortowania bąbelkowego w kontekście CUDA charakteryzuje się wysokim recyklingiem danych. Te same elementy tablicy są porównywane i zamieniane wielokrotnie podczas kolejnych przebiegów. Istnieje możliwość przechowywania fragmentów danych w pamięci współdzielonej (shared memory), co może znacząco ograniczyć koszt dostępu do globalnej pamięci. Pomimo tego, równoleglenie algorytmu jest trudne ze względu na silne zależności pomiędzy iteracjami pętli.

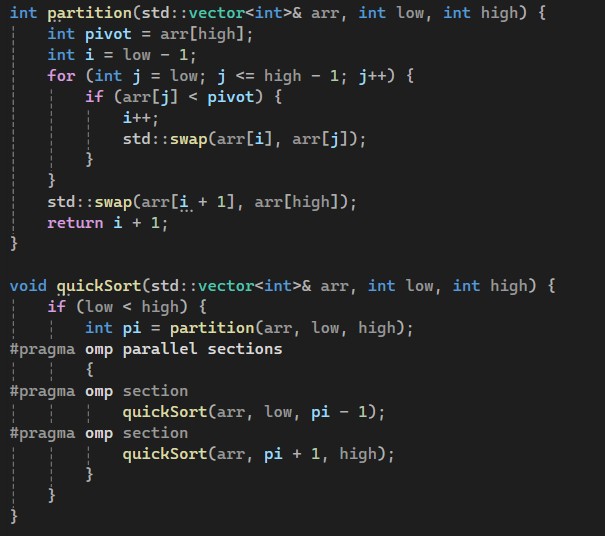
## 3.2. Sortowanie szybkie

Sekwencyjnie:



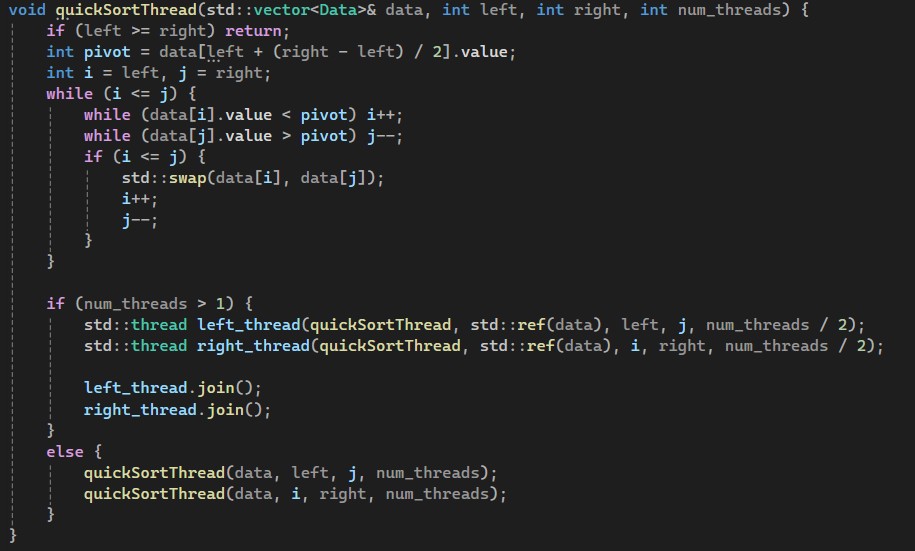
Wybiera element pivot, dzieli dane na mniejsze i większe od pivot, a następnie sortuje obie części rekurencyjnie.

OpenMP:



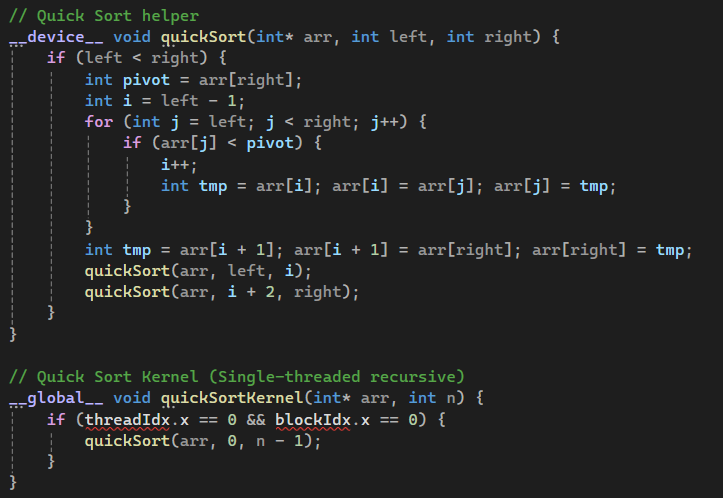
Tutaj wykorzystano #pragma omp parallel sections. Główna idea to rekurencyjne dzielenie danych wokół pivota i równoległe sortowanie lewej i prawej części. OpenMP tworzy dwa niezależne wątki (sekcje), które rekurencyjnie wywołują quickSort. To dobrze skaluje się na danych, ale wymaga dużej głębokości rekurencji dla pełnego wykorzystania wielu rdzeni.

Threads C++:



Algorytm dzieli dane wokół pivota, po czym sortuje lewą i prawą część rekurencyjnie. Jeśli liczba dostępnych wątków jest większa niż 1, tworzy się dwa nowe wątki: jeden sortuje lewą część, drugi prawą. W przeciwnym razie funkcje wywołują się sekwencyjnie.

CUDA:

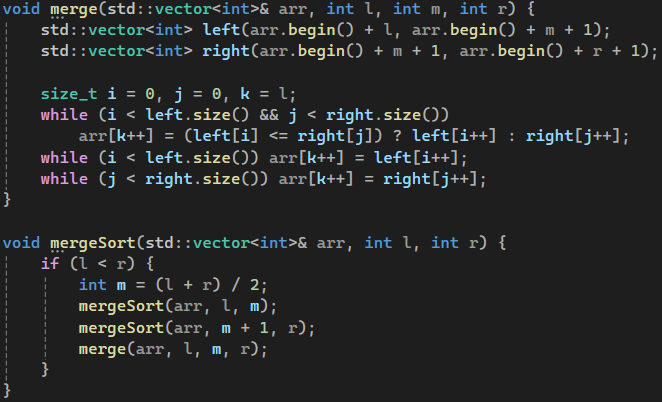


Zrealizowane jako proof-of-concept bez pełnej równoległości, z naciskiem na uruchamianie GPU i pomiar czasu.

W przypadku sortowania szybkiego dane nie są często współdzielone pomiędzy wątkami, a większość operacji odbywa się lokalnie w ramach rekurencyjnych podziałów tablicy. Możliwość recyklingu danych jest ograniczona, gdyż każdy wątek przetwarza inny zakres danych i nie korzysta z wyników innych. Z tego względu recykling danych w CUDA dla quick sorta jest oceniany jako umiarkowany.

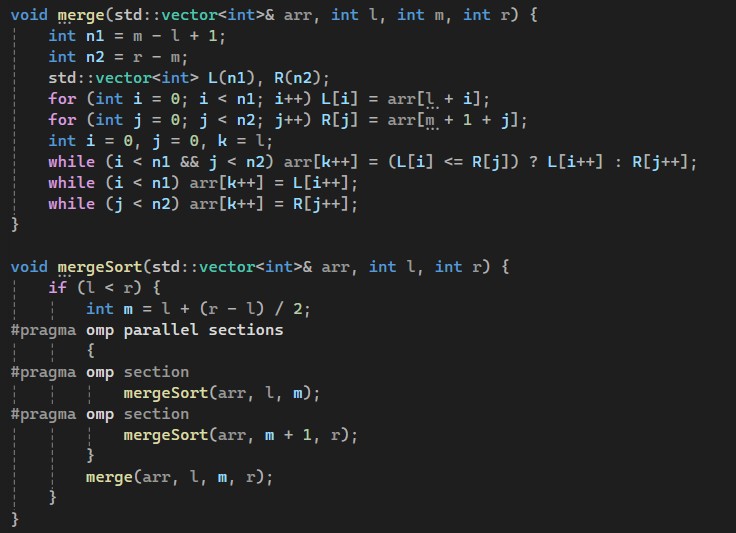
## 3.3. Sortowanie przez scalanie

Sekwencyjnie:



Dzieli dane na połowy, sortuje każdą rekurencyjnie i scala je w jedną posortowaną tablicę.

OpenMP:



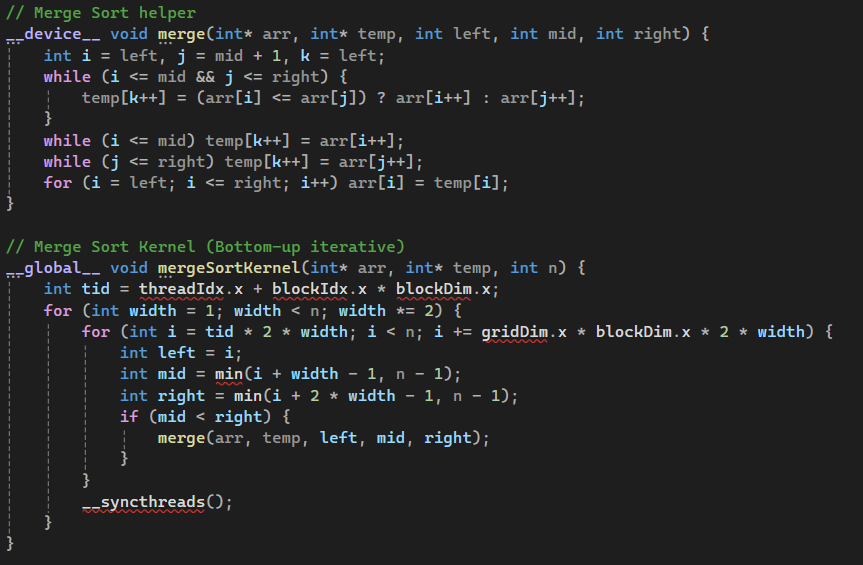
To najlepszy kandydat do zrównoleglenia. W algorytmie mergeSort za pomocą #pragma omp parallel sections rekurencyjnie uruchamiane są dwie części: lewa i prawa. Po ich wykonaniu następuje scalanie (funkcja merge). Dobrze się skaluje i przy większej liczbie rdzeni osiąga znaczące przyspieszenie. Synchronizacja odbywa się głównie po zakończeniu scalania, więc ryzyko konfliktów jest niewielkie.

Threads C++:



Działa podobnie do wersji OpenMP. Tablica jest dzielona na dwie części. Jeśli liczba wątków pozwala, tworzone są dwa osobne wątki dla sortowania lewej i prawej połowy. Po zakończeniu obie części są scalane (merge) w wątku głównym.

CUDA:

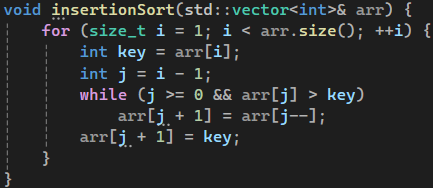


Najbardziej realistyczna i skalowalna wersja sortowania własnym kernelem; pokazuje siłę dzielenia pracy między wiele bloków.

Sortowanie przez scalanie bardzo dobrze nadaje się do implementacji na GPU i oferuje wysoki poziom recyklingu danych. W różnych etapach algorytmu te same fragmenty danych są ponownie wykorzystywane przy łączeniu podtablic. Można efektywnie wykorzystać pamięć współdzieloną do lokalnego przetwarzania, co zmniejsza liczbę odwołań do globalnej pamięci i poprawia wydajność.

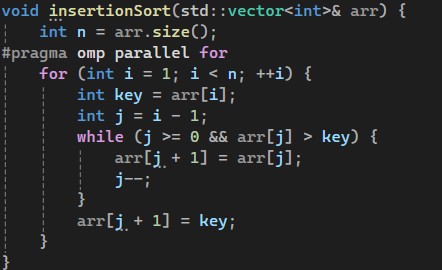
## 3.4. Sortowanie przez wstawienie

Sekwencyjnie:



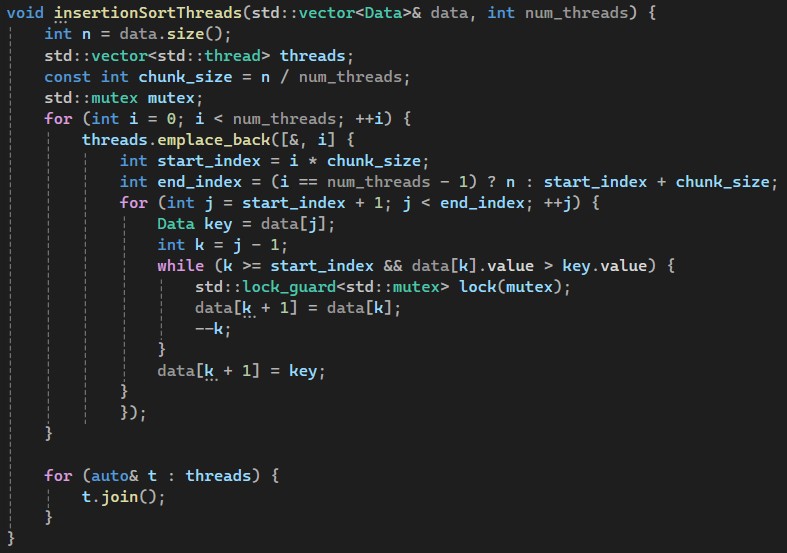
Buduje posortowaną listę element po elemencie, wstawiając każdy nowy element w odpowiednie miejsce względem poprzednich.

OpenMP:



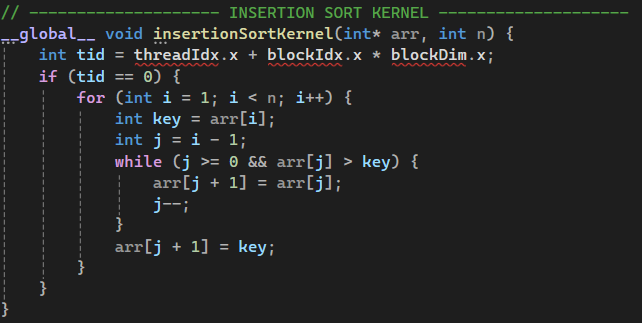
Ten algorytm został zrównoleglony przy pomocy #pragma omp parallel for. Każdy wątek wykonuje własną część sortowania, ale algorytm insertion sort wymaga przesuwania poprzednich elementów, co tworzy konflikty. W praktyce taka równoległość nie przynosi znacznego przyspieszenia, a nawet może powodować błędy, jeśli nie są zastosowane zabezpieczenia (np. mutexy).

Threads C++:



Dane są podzielone na bloki i przypisane do wątków. Każdy wątek wykonuje insertion sort na swoim zakresie. Podczas przesuwania elementów oraz ich wstawiania stosowany jest mutex, by nie dopuścić do kolizji.

CUDA:

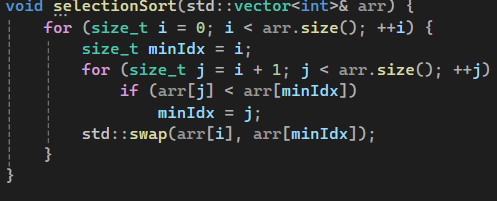


Iteruje przez tablicę i „wstawia” każdy element we właściwe miejsce w już posortowanej części. Tak jak selection sort — uruchomiona w pojedynczym wątku i też w CUDA jest to tylko jako ciekawostka.

W przypadku sortowania przez wstawianie recykling danych jest niski. Dane są przetwarzane liniowo, każdy nowy element jest porównywany z już posortowaną częścią tablicy, a ewentualne przesunięcia dotyczą tylko kilku sąsiednich elementów. Algorytm nie oferuje istotnych możliwości współdzielenia danych pomiędzy wątkami, przez co nie korzysta efektywnie z architektury CUDA.

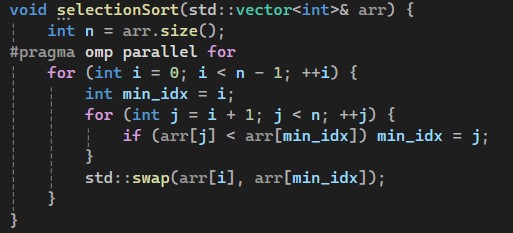
## 3.5. Sortowanie przez wybieranie

Sekwencyjnie:



Znajduje najmniejszy element i przesuwa go na początek. Powtarza to dla każdego kolejnego elementu.

OpenMP:



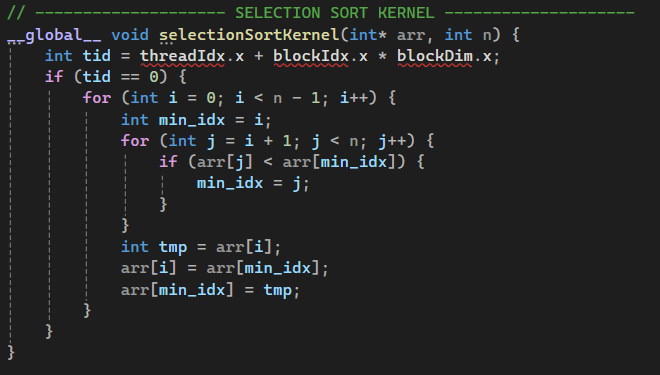
Tutaj również użyto #pragma omp parallel for. Każdy wątek może przeszukiwać swoją część tablicy, aby znaleźć minimalny element, ale zmiana pozycji (swap) wymaga synchronizacji. Jest to trudne do efektywnego zrównoleglenia bez bardziej zaawansowanej kontroli współbieżności (np. blokad na poziomie indeksów).

Threads C++:



Podobnie jak w insertion sort – dane są dzielone i przypisane do wątków. Każdy wątek szuka najmniejszego elementu w swoim fragmencie. Operacje zamiany elementów chronione są mutexem.

CUDA:



Zastosowanie CUDA jest tu bardziej edukacyjnie, pokazuje różnice między algorytmem łatwym do zrównoleglenia (jak Merge Sort), a takim, który lepiej pozostawić sekwencyjny.

Sortowanie przez wybieranie wykazuje umiarkowany poziom recyklingu danych. W każdej iteracji przeglądana jest ta sama nieposortowana część tablicy w celu znalezienia minimum. Dane są czytane wielokrotnie, co stwarza możliwość ich lokalnego buforowania w pamięci współdzielonej. Pomimo tego, synchronizacja pomiędzy wątkami może stanowić wąskie gardło i ograniczać efektywność.

# 4. Scenariusze testowe i metodologia badań

**Zakres testów**

W ramach eksperymentu przeprowadzono pomiary czasu wykonania pięciu algorytmów sortowania w różnych wariantach implementacyjnych (sekwencyjna, wątki C++, OpenMP, CUDA). Celem testów było określenie:

* jak zmienia się wydajność wraz ze wzrostem liczby wątków,
* który algorytm najlepiej reaguje na zrównoleglenie,
* kiedy równoległość przestaje być opłacalna.

**Parametry testowe**

* **Rozmiar danych (n)**: 100, 5 000, 25 000, 50 000, 100 000 (losowe liczby całkowite z zakresu 0–999 999 999). Dla 1 000 000 zostały wykonane tylko kilka pomiarów
* **Liczba wątków (p)**: 1, 2, 4, 6, 10, 14, 16, 20 (dla wersji równoległych).
* **Konfiguracja GPU**: <<<2, 4>>>, <<<4, 8>>>, <<<8, 16>>>, <<<32, 16>>>, <<<44, 32>>>

**Pomiar czasu wykonania**

Dla każdej kombinacji (n, p) mierzono czas sortowania w milisekundach. Dla porównań zbierano wyniki z:

* wersji sekwencyjnej (p = 1),
* wersji OpenMP i threads (p ≥ 2).
* wersja CUDA (GPU) - wykonywana na wielu wątkach GPU, gdzie liczba bloków i wątków w bloku dobierana jest dynamicznie. Pomiar czasu obejmował cały czas wykonania kernela oraz transfer danych między CPU a GPU.

**Analiza przyspieszenia (speedup)**

Obliczano tzw. **przyspieszenie równoległe**:

Gdzie:

* **T(1)** to czas wykonania wersji sekwencyjnej,
* **T(p)** to czas wykonania przy p wątkach.

**Ocena opłacalności**

Sprawdzano, w których przypadkach zwiększanie liczby wątków **rzeczywiście skraca czas sortowania**, a gdzie **koszty synchronizacji** (np. mutexy, bariery) zaczynają przeważać nad zyskami.

# 5. Pomiary Czasowe

## 5.1. Sekwencyjnie

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Liczba wątków** | **Technologia** | **Rozmiar** | **Algorytm sortowania** | **Czas (s)** |
| 1 | Sekwencyjnie | 100 | Bąbelkowy | 0.0000786 |
| Szybki | 0.0000111 |
| Przez scalanie | 0.0003801 |
| Przez wstawianie | 0.0000068 |
| Przez wybieranie | 0.0000599 |
| 5 000 | Bąbelkowy | 0.2031220 |
| Szybki | 0.0011894 |
| Przez scalanie | 0.0193976 |
| Przez wstawianie | 0.0005073 |
| Przez wybieranie | 0.1468880 |
| 25 000 | Bąbelkowy | 5.0900900 |
| Szybki | 0.0088364 |
| Przez scalanie | 0.1014790 |
| Przez wstawianie | 0.0033543 |
| Przez wybieranie | 3.4139300 |
| 50 000 | Bąbelkowy | 20.4790000 |
| Szybki | 0.0142580 |
| Przez scalanie | 0.2024580 |
| Przez wstawianie | 0.0078724 |
| Przez wybieranie | 13.6548000 |
| 100 000 | Bąbelkowy | 82.1745660 |
| Szybki | 0.0292025 |
| Przez scalanie | 0.4023980 |
| Przez wstawianie | 0.0150941 |
| Przez wybieranie | 55.8875760 |
| 1 000 000 | Bąbelkowy | 1165.0048180 |
| Szybki | 0.2967040 |
| Przez scalanie | 4.0861520 |
| Przez wstawianie | 0.1684690 |
| Przez wybieranie | 778.6452610 |

## 5.2. Nvidia CUDA

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Konfiguracja GPU** | **Technologia** | **Rozmiar** | **Algorytm sortowania** | **Czas (s)** | **Przyspieszenie (s)** | **Przyspieszenie (%)** |
| <<<2, 4>>> | CUDA | 100 | Bąbelkowy | 0.000780768 | -0.000702168 | -89334% |
| Szybki | 0.000702464 | -0.000691364 | -622850% |
| Przez scalanie | 0.000466208 | -0.000086108 | -2265% |
| Przez wstawianie | 0.0022231 | -0.0022163 | -3259265% |
| Przez wybieranie | 0.00349245 | -0.00343255 | -573047% |
| 5 000 | Bąbelkowy | 0.371663 | -0.168541 | -8298% |
| Szybki | 0.072689 | -0.0714996 | -601140% |
| Przez scalanie | 0.0214909 | -0.0020933 | -1079% |
| Przez wstawianie | 3.40845 | -3.4079427 | -67178054% |
| Przez wybieranie | 5.48805 | -5.341162 | -363621% |
| 25 000 | Bąbelkowy | 11.4417 | -6.35161 | -12478% |
| Szybki | 0.362491 | -0.3536546 | -400225% |
| Przez scalanie | 0.103227 | -0.001748 | -172% |
| Przez wstawianie | 84.9376 | -84.9342457 | -253210046% |
| Przez wybieranie | 139.51 | -136.09607 | -398649% |
| 50 000 | Bąbelkowy | 49.1245 | -28.6455 | -13988% |
| Szybki | 0.664103 | -0.649845 | -455776% |
| Przez scalanie | 0.210633 | -0.008175 | -404% |
| Przez wstawianie | 412.376 | -412.3681276 | -523815009% |
| Przez wybieranie | 601.2881 | -587.6333 | -430349% |
| 100 000 | Bąbelkowy | 174.12569 | -91.951124 | -11190% |
| Szybki | 1.63578 | -1.6065775 | -550151% |
| Przez scalanie | 0.486743 | -0.084345 | -2096% |
| Przez wstawianie | 1649.504 | -1649.488906 | -1092803748% |
| Przez wybieranie | 2405.152 | -2349.264424 | -420355% |
| <<<4, 8>>> | CUDA | 100 | Bąbelkowy | 0.00063712 | -0.00055852 | -71059% |
| Szybki | 0.000683214 | -0.000672114 | -605508% |
| Przez scalanie | 0.000484384 | -0.000104284 | -2744% |
| Przez wstawianie | 0.00228966 | -0.00228286 | -3357147% |
| Przez wybieranie | 0.00296378 | -0.00290388 | -484788% |
| 5 000 | Bąbelkowy | 0.0295977 | 0.1735243 | 8543% |
| Szybki | 0.000718912 | 0.000470488 | 3956% |
| Przez scalanie | 0.0180204 | 0.0013772 | 710% |
| Przez wstawianie | 3.34966 | -3.3491527 | -66019174% |
| Przez wybieranie | 5.52253 | -5.375642 | -365969% |
| 25 000 | Bąbelkowy | 0.572681 | 4.517409 | 8875% |
| Szybki | 0.351256 | -0.3424196 | -387510% |
| Przez scalanie | 0.0816915 | 0.0197875 | 1950% |
| Przez wstawianie | 93.7902 | -93.7868457 | -279601842% |
| Przez wybieranie | 138.21 | -134.79607 | -394841% |
| 50 000 | Bąbelkowy | 2.1457 | 18.3333 | 8952% |
| Szybki | 0.647235 | -0.632977 | -443945% |
| Przez scalanie | 0.205637 | -0.003179 | -157% |
| Przez wstawianie | 417.891 | -417.8831276 | -530820496% |
| Przez wybieranie | 599.303 | -585.6482 | -428895% |
| 100 000 | Bąbelkowy | 9.998 | 72.176566 | 8783% |
| Szybki | 1.55125 | -1.5220475 | -521205% |
| Przez scalanie | 0.421246 | -0.018848 | -468% |
| Przez wstawianie | 1671.564 | -1671.548906 | -1107418730% |
| Przez wybieranie | 2397.212 | -2341.324424 | -418935% |
| <<<8, 16>>> | CUDA | 100 | Bąbelkowy | 0.000636576 | -0.000557976 | -70989% |
| Szybki | 0.000618496 | -0.000607396 | -547204% |
| Przez scalanie | 0.000457984 | -0.000077884 | -2049% |
| Przez wstawianie | 0.0016919 | -0.0016851 | -2478088% |
| Przez wybieranie | 0.00333085 | -0.00327095 | -546068% |
| 5 000 | Bąbelkowy | 0.00972675 | 0.19339525 | 9521% |
| Szybki | 0.0735678 | -0.0723784 | -608529% |
| Przez scalanie | 0.0180732 | 0.0013244 | 683% |
| Przez wstawianie | 3.40966 | -3.4091527 | -67201906% |
| Przez wybieranie | 5.54309 | -5.396202 | -367368% |
| 25 000 | Bąbelkowy | 0.177548 | 4.912542 | 9651% |
| Szybki | 0.35637 | -0.3475336 | -393298% |
| Przez scalanie | 0.0981375 | 0.0033415 | 329% |
| Przez wstawianie | 95.47 | -95.4666457 | -284609742% |
| Przez wybieranie | 140.9 | -137.48607 | -402721% |
| 50 000 | Bąbelkowy | 0.565387 | 19.913613 | 9724% |
| Szybki | 0.626776 | -0.612518 | -429596% |
| Przez scalanie | 0.159809 | 0.042649 | 2107% |
| Przez wstawianie | 414.108 | -414.1001276 | -526015100% |
| Przez wybieranie | 607.279 | -593.6242 | -434737% |
| 100 000 | Bąbelkowy | 2.11509 | 80.059476 | 9743% |
| Szybki | 1.51125 | -1.4820475 | -507507% |
| Przez scalanie | 0.321294 | 0.081104 | 2016% |
| Przez wstawianie | 1656.432 | -1656.416906 | -1097393621% |
| Przez wybieranie | 2429.116 | -2373.228424 | -424643% |
| <<<32, 16>>> | CUDA | 100 | Bąbelkowy | 0.00065056 | -0.00057196 | -72768% |
| Szybki | 0.000586752 | -0.000575652 | -518605% |
| Przez scalanie | 0.000452032 | -0.000071932 | -1892% |
| Przez wstawianie | 0.0017753 | -0.0017685 | -2600735% |
| Przez wybieranie | 0.00295427 | -0.00289437 | -483200% |
| 5 000 | Bąbelkowy | 0.00513859 | 0.19798341 | 9747% |
| Szybki | 0.0712357 | -0.0700463 | -588921% |
| Przez scalanie | 0.0179203 | 0.0014773 | 762% |
| Przez wstawianie | 3.40008 | -3.3995727 | -67013063% |
| Przez wybieranie | 5.55861 | -5.411722 | -368425% |
| 25 000 | Bąbelkowy | 0.0565407 | 5.0335493 | 9889% |
| Szybki | 0.336892 | -0.3280556 | -371255% |
| Przez scalanie | 0.0976297 | 0.0038493 | 379% |
| Przez wstawianie | 91.3254 | -91.3220457 | -272253662% |
| Przez wybieranie | 142.117 | -138.70307 | -406286% |
| 50 000 | Bąbelkowy | 0.175591 | 20.303409 | 9914% |
| Szybki | 0.611356 | -0.597098 | -418781% |
| Przez scalanie | 0.165812 | 0.036646 | 1810% |
| Przez wstawianie | 419.662 | -419.6541276 | -533070128% |
| Przez wybieranie | 612.52427 | -598.86947 | -438578% |
| 100 000 | Bąbelkowy | 0.581219 | 81.593347 | 9929% |
| Szybki | 1.50012 | -1.4709175 | -503696% |
| Przez scalanie | 0.318847 | 0.083551 | 2076% |
| Przez wstawianie | 1678.648 | -1678.632906 | -1112111955% |
| Przez wybieranie | 2450.097 | -2394.209424 | -428397% |
| <<<44, 32>>> | CUDA | 100 | Bąbelkowy | 0.000669696 | -0.000591096 | -75203% |
| Szybki | 0.00070704 | -0.00069594 | -626973% |
| Przez scalanie | 0.000475936 | -0.000095836 | -2521% |
| Przez wstawianie | 0.00185078 | -0.00184398 | -2711735% |
| Przez wybieranie | 0.00365306 | -0.00359316 | -599860% |
| 5 000 | Bąbelkowy | 0.00469197 | 0.19843003 | 9769% |
| Szybki | 0.069252 | -0.0680626 | -572243% |
| Przez scalanie | 0.018592 | 0.0008056 | 415% |
| Przez wstawianie | 3.45009 | -3.4495827 | -67998870% |
| Przez wybieranie | 5.59971 | -5.452822 | -371223% |
| 25 000 | Bąbelkowy | 0.0261149 | 5.0639751 | 9949% |
| Szybki | 0.306392 | -0.2975556 | -336738% |
| Przez scalanie | 0.079646 | 0.021833 | 2151% |
| Przez wstawianie | 84.3561 | -84.3527457 | -251476450% |
| Przez wybieranie | 139.476 | -136.06207 | -398550% |
| 50 000 | Bąbelkowy | 0.0705436 | 20.4084564 | 9966% |
| Szybki | 0.593257 | -0.578999 | -406087% |
| Przez scalanie | 0.186547 | 0.015911 | 786% |
| Przez wstawianie | 415.849 | -415.8411276 | -528226624% |
| Przez wybieranie | 600.643 | -586.9882 | -429877% |
| 100 000 | Bąbelkowy | 0.216778 | 81.957788 | 9974% |
| Szybki | 1.47762 | -1.4484175 | -495991% |
| Przez scalanie | 0.412467 | -0.010069 | -250% |
| Przez wstawianie | 1663.396 | -1663.380906 | -1102007345% |
| Przez wybieranie | 2402.572 | -2346.684424 | -419894% |

## 5.3. Threads C++ oraz OpenMP

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Liczba wątków** | **Technologia** | **Rozmiar** | **Algorytm sortowania** | **Czas (s)** | **Przyspieszenie (s)** | **Przyspieszenie (%)** |
| 2 | OpenMP | 100 | Bąbelkowy | 0.000781 | -0.00070280 | -89415% |
| Thread C++ | 0.003391 | -0.00331270 | -421463% |
| OpenMP | Szybki | 0.000041 | -0.00003000 | -27027% |
| Thread C++ | 0.000932 | -0.00092070 | -829459% |
| OpenMP | Przez scalanie | 0.000239 | 0.00014160 | 3725% |
| Thread C++ | 0.001025 | -0.00064500 | -16969% |
| OpenMP | Przez wstawianie | 0.000021 | -0.00001410 | -20735% |
| Thread C++ | 0.000922 | -0.00091500 | -1345588% |
| OpenMP | Przez wybieranie | 0.000059 | 0.00000110 | 184% |
| Thread C++ | 0.000811 | -0.00075080 | -125342% |
| OpenMP | 5 000 | Bąbelkowy | 0.150316 | 0.05280600 | 2600% |
| Thread C++ | 0.385534 | -0.18241200 | -8980% |
| OpenMP | Szybki | 0.001543 | -0.00035350 | -2972% |
| Thread C++ | 0.002261 | -0.00107160 | -9010% |
| OpenMP | Przez scalanie | 0.009888 | 0.00951000 | 4903% |
| Thread C++ | 0.010150 | 0.00924810 | 4768% |
| OpenMP | Przez wstawianie | 0.019581 | -0.01907370 | -375985% |
| Thread C++ | 0.288014 | -0.28750670 | -5667390% |
| OpenMP | Przez wybieranie | 0.079240 | 0.06764770 | 4605% |
| Thread C++ | 0.086344 | 0.06054380 | 4122% |
| OpenMP | 25 000 | Bąbelkowy | 3.756220 | 1.33387000 | 2621% |
| Thread C++ | 9.633680 | -4.54359000 | -8926% |
| OpenMP | Szybki | 0.005222 | 0.00361460 | 4091% |
| Thread C++ | 0.006915 | 0.00192110 | 2174% |
| OpenMP | Przez scalanie | 0.053610 | 0.04786880 | 4717% |
| Thread C++ | 0.061893 | 0.03958620 | 3901% |
| OpenMP | Przez wstawianie | 0.471482 | -0.46812770 | -1395605% |
| Thread C++ | 6.978610 | -6.97525570 | -20794967% |
| OpenMP | Przez wybieranie | 1.995240 | 1.41869000 | 4156% |
| Thread C++ | 2.555790 | 0.85814000 | 2514% |
| OpenMP | 50 000 | Bąbelkowy | 15.192700 | 5.28630000 | 2581% |
| Thread C++ | 41.939200 | -21.46020000 | -10479% |
| OpenMP | Szybki | 0.019268 | -0.00500970 | -3514% |
| Thread C++ | 0.013857 | 0.00040110 | 281% |
| OpenMP | Przez scalanie | 0.132391 | 0.07006700 | 3461% |
| Thread C++ | 0.128216 | 0.07424200 | 3667% |
| OpenMP | Przez wstawianie | 2.362530 | -2.35465760 | -2991029% |
| Thread C++ | 34.789400 | -34.78152760 | -44181606% |
| OpenMP | Przez wybieranie | 8.950060 | 4.70474000 | 3445% |
| Thread C++ | 8.732850 | 4.92195000 | 3605% |
| OpenMP | 100 000 | Bąbelkowy | 65.515000 | 16.65956600 | 2027% |
| Thread C++ | 159.848000 | -77.67343400 | -9452% |
| OpenMP | Szybki | 0.054628 | -0.02542560 | -8707% |
| Thread C++ | 0.025994 | 0.00320900 | 1099% |
| OpenMP | Przez scalanie | 0.231007 | 0.17139100 | 4259% |
| Thread C++ | 0.268082 | 0.13431600 | 3338% |
| OpenMP | Przez wstawianie | 10.426000 | -10.41090590 | -6897335% |
| Thread C++ | 113.603000 | -113.58790590 | -75253182% |
| OpenMP | Przez wybieranie | 35.349600 | 20.53797600 | 3675% |
| Thread C++ | 41.494200 | 14.39337600 | 2575% |
| 4 | OpenMP | 100 | Bąbelkowy | 0.001131 | -0.00105250 | -133906% |
| Thread C++ | 0.004417 | -0.00433790 | -551896% |
| OpenMP | Szybki | 0.000028 | -0.00001650 | -14865% |
| Thread C++ | 0.002917 | -0.00290610 | -2618108% |
| OpenMP | Przez scalanie | 0.000206 | 0.00017410 | 4580% |
| Thread C++ | 0.002593 | -0.00221250 | -58208% |
| OpenMP | Przez wstawianie | 0.000014 | -0.00000680 | -10000% |
| Thread C++ | 0.001607 | -0.00160010 | -2353088% |
| OpenMP | Przez wybieranie | 0.000045 | 0.00001500 | 2504% |
| Thread C++ | 0.001725 | -0.00166490 | -277947% |
| OpenMP | 5 000 | Bąbelkowy | 0.090714 | 0.11240850 | 5534% |
| Thread C++ | 0.350346 | -0.14722400 | -7248% |
| OpenMP | Szybki | 0.001560 | -0.00037070 | -3117% |
| Thread C++ | 0.003476 | -0.00228670 | -19226% |
| OpenMP | Przez scalanie | 0.012103 | 0.00729460 | 3761% |
| Thread C++ | 0.021134 | -0.00173640 | -895% |
| OpenMP | Przez wstawianie | 0.008672 | -0.00816420 | -160934% |
| Thread C++ | 0.254769 | -0.25426170 | -5012058% |
| OpenMP | Przez wybieranie | 0.046153 | 0.10073530 | 6858% |
| Thread C++ | 0.058083 | 0.08880460 | 6046% |
| OpenMP | 25 000 | Bąbelkowy | 2.106020 | 2.98407000 | 5863% |
| Thread C++ | 8.030270 | -2.94018000 | -5776% |
| OpenMP | Szybki | 0.010147 | -0.00131100 | -1484% |
| Thread C++ | 0.006160 | 0.00267610 | 3028% |
| OpenMP | Przez scalanie | 0.073720 | 0.02775860 | 2735% |
| Thread C++ | 0.077156 | 0.02432340 | 2397% |
| OpenMP | Przez wstawianie | 0.140724 | -0.13736970 | -409533% |
| Thread C++ | 6.166950 | -6.16359570 | -18375207% |
| OpenMP | Przez wybieranie | 1.201550 | 2.21238000 | 6480% |
| Thread C++ | 1.305000 | 2.10893000 | 6177% |
| OpenMP | 50 000 | Bąbelkowy | 8.399440 | 12.07956000 | 5899% |
| Thread C++ | 31.501400 | -11.02240000 | -5382% |
| OpenMP | Szybki | 0.020103 | -0.00584460 | -4099% |
| Thread C++ | 0.012521 | 0.00173660 | 1218% |
| OpenMP | Przez scalanie | 0.126016 | 0.07644200 | 3776% |
| Thread C++ | 0.166350 | 0.03610800 | 1783% |
| OpenMP | Przez wstawianie | 0.550619 | -0.54274660 | -689430% |
| Thread C++ | 24.635100 | -24.62722760 | -31282998% |
| OpenMP | Przez wybieranie | 5.029500 | 8.62530000 | 6317% |
| Thread C++ | 5.379340 | 8.27546000 | 6060% |
| OpenMP | 100 000 | Bąbelkowy | 34.045000 | 48.12956600 | 5857% |
| Thread C++ | 124.596000 | -42.42143400 | -5162% |
| OpenMP | Szybki | 0.043755 | -0.01455280 | -4983% |
| Thread C++ | 0.021103 | 0.00809980 | 2774% |
| OpenMP | Przez scalanie | 0.217792 | 0.18460600 | 4588% |
| Thread C++ | 0.276738 | 0.12566000 | 3123% |
| OpenMP | Przez wstawianie | 2.137900 | -2.12280590 | -1406381% |
| Thread C++ | 89.176000 | -89.16090590 | -59070038% |
| OpenMP | Przez wybieranie | 19.698500 | 36.18907600 | 6475% |
| Thread C++ | 25.547700 | 30.33987600 | 5429% |
| 6 | OpenMP | 100 | Bąbelkowy | 0.001544 | -0.00146530 | -186425% |
| Thread C++ | 0.005567 | -0.00548830 | -698257% |
| OpenMP | Szybki | 0.000039 | -0.00002750 | -24775% |
| Thread C++ | 0.002512 | -0.00250120 | -2253333% |
| OpenMP | Przez scalanie | 0.000238 | 0.00014220 | 3741% |
| Thread C++ | 0.005164 | -0.00478370 | -125854% |
| OpenMP | Przez wstawianie | 0.000035 | -0.00002770 | -40735% |
| Thread C++ | 0.002722 | -0.00271480 | -3992353% |
| OpenMP | Przez wybieranie | 0.000031 | 0.00002850 | 4758% |
| Thread C++ | 0.002291 | -0.00223150 | -372538% |
| OpenMP | 5 000 | Bąbelkowy | 0.060619 | 0.14250340 | 7016% |
| Thread C++ | 0.282224 | -0.07910200 | -3894% |
| OpenMP | Szybki | 0.001832 | -0.00064260 | -5403% |
| Thread C++ | 0.004273 | -0.00308360 | -25926% |
| OpenMP | Przez scalanie | 0.015238 | 0.00415970 | 2144% |
| Thread C++ | 0.022152 | -0.00275390 | -1420% |
| OpenMP | Przez wstawianie | 0.004138 | -0.00363060 | -71567% |
| Thread C++ | 0.233848 | -0.23334070 | -4599659% |
| OpenMP | Przez wybieranie | 0.045296 | 0.10159210 | 6916% |
| Thread C++ | 0.039656 | 0.10723250 | 7300% |
| OpenMP | 25 000 | Bąbelkowy | 1.484200 | 3.60589000 | 7084% |
| Thread C++ | 6.891970 | -1.80188000 | -3540% |
| OpenMP | Szybki | 0.010655 | -0.00181850 | -2058% |
| Thread C++ | 0.007165 | 0.00167130 | 1891% |
| OpenMP | Przez scalanie | 0.076873 | 0.02460580 | 2425% |
| Thread C++ | 0.092609 | 0.00887010 | 874% |
| OpenMP | Przez wstawianie | 0.090404 | -0.08704930 | -259516% |
| Thread C++ | 5.558720 | -5.55536570 | -16561923% |
| OpenMP | Przez wybieranie | 0.968558 | 2.44537200 | 7163% |
| Thread C++ | 1.012760 | 2.40117000 | 7033% |
| OpenMP | 50 000 | Bąbelkowy | 5.966310 | 14.51269000 | 7087% |
| Thread C++ | 26.774700 | -6.29570000 | -3074% |
| OpenMP | Szybki | 0.026792 | -0.01253410 | -8791% |
| Thread C++ | 0.012139 | 0.00211920 | 1486% |
| OpenMP | Przez scalanie | 0.144285 | 0.05817300 | 2873% |
| Thread C++ | 0.182603 | 0.01985500 | 981% |
| OpenMP | Przez wstawianie | 0.303994 | -0.29612160 | -376152% |
| Thread C++ | 22.646200 | -22.63832760 | -28756577% |
| OpenMP | Przez wybieranie | 3.573450 | 10.08135000 | 7383% |
| Thread C++ | 4.107500 | 9.54730000 | 6992% |
| OpenMP | 100 000 | Bąbelkowy | 23.673200 | 58.50136600 | 7119% |
| Thread C++ | 108.418000 | -26.24343400 | -3194% |
| OpenMP | Szybki | 0.039889 | -0.01068630 | -3659% |
| Thread C++ | 0.020239 | 0.00896370 | 3069% |
| OpenMP | Przez scalanie | 0.238433 | 0.16396500 | 4075% |
| Thread C++ | 0.355933 | 0.04646500 | 1155% |
| OpenMP | Przez wstawianie | 1.167560 | -1.15246590 | -763521% |
| Thread C++ | 90.374200 | -90.35910590 | -59863858% |
| OpenMP | Przez wybieranie | 14.070900 | 41.81667600 | 7482% |
| Thread C++ | 16.007700 | 39.87987600 | 7136% |
| 10 | OpenMP | 100 | Bąbelkowy | 0.002698 | -0.00261980 | -333308% |
| Thread C++ | 0.006760 | -0.00668140 | -850051% |
| OpenMP | Szybki | 0.000050 | -0.00003920 | -35315% |
| Thread C++ | 0.005753 | -0.00574140 | -5172432% |
| OpenMP | Przez scalanie | 0.000278 | 0.00010200 | 2684% |
| Thread C++ | 0.008134 | -0.00775380 | -203994% |
| OpenMP | Przez wstawianie | 0.000013 | -0.00000580 | -8529% |
| Thread C++ | 0.004389 | -0.00438180 | -6443824% |
| OpenMP | Przez wybieranie | 0.000024 | 0.00003560 | 5943% |
| Thread C++ | 0.003918 | -0.00385850 | -644157% |
| OpenMP | 5 000 | Bąbelkowy | 0.053932 | 0.14919020 | 7345% |
| Thread C++ | 0.067108 | 0.13601430 | 6696% |
| OpenMP | Szybki | 0.002608 | -0.00141840 | -11925% |
| Thread C++ | 0.035766 | -0.03457630 | -290704% |
| OpenMP | Przez scalanie | 0.019426 | -0.00002820 | -15% |
| Thread C++ | 0.104466 | -0.08506840 | -43855% |
| OpenMP | Przez wstawianie | 0.002354 | -0.00184630 | -36395% |
| Thread C++ | 0.055223 | -0.05471580 | -1078569% |
| OpenMP | Przez wybieranie | 0.031563 | 0.11532470 | 7851% |
| Thread C++ | 0.057642 | 0.08924630 | 6076% |
| OpenMP | 25 000 | Bąbelkowy | 1.001480 | 4.08861000 | 8032% |
| Thread C++ | 6.152340 | -1.06225000 | -2087% |
| OpenMP | Szybki | 0.012600 | -0.00376320 | -4259% |
| Thread C++ | 0.009456 | -0.00061970 | -701% |
| OpenMP | Przez scalanie | 0.077017 | 0.02446170 | 2411% |
| Thread C++ | 0.124624 | -0.02314500 | -2281% |
| OpenMP | Przez wstawianie | 0.040102 | -0.03674780 | -109554% |
| Thread C++ | 4.708540 | -4.70518570 | -14027325% |
| OpenMP | Przez wybieranie | 0.652613 | 2.76131700 | 8088% |
| Thread C++ | 0.737793 | 2.67613700 | 7839% |
| OpenMP | 50 000 | Bąbelkowy | 3.968090 | 16.51091000 | 8062% |
| Thread C++ | 25.004900 | -4.52590000 | -2210% |
| OpenMP | Szybki | 0.024803 | -0.01054490 | -7396% |
| Thread C++ | 0.013183 | 0.00107460 | 754% |
| OpenMP | Przez scalanie | 0.171614 | 0.03084400 | 1523% |
| Thread C++ | 0.231924 | -0.02946600 | -1455% |
| OpenMP | Przez wstawianie | 0.160636 | -0.15276360 | -194050% |
| Thread C++ | 19.679100 | -19.67122760 | -24987587% |
| OpenMP | Przez wybieranie | 2.620200 | 11.03460000 | 8081% |
| Thread C++ | 2.813540 | 10.84126000 | 7940% |
| OpenMP | 100 000 | Bąbelkowy | 15.246600 | 66.92796600 | 8145% |
| Thread C++ | 99.511400 | -17.33683400 | -2110% |
| OpenMP | Szybki | 0.051673 | -0.02247020 | -7695% |
| Thread C++ | 0.017344 | 0.01185840 | 4061% |
| OpenMP | Przez scalanie | 0.243935 | 0.15846300 | 3938% |
| Thread C++ | 0.458018 | -0.05562000 | -1382% |
| OpenMP | Przez wstawianie | 0.633166 | -0.61807190 | -409479% |
| Thread C++ | 78.253700 | -78.23860590 | -51833899% |
| OpenMP | Przez wybieranie | 9.801250 | 46.08632600 | 8246% |
| Thread C++ | 11.477500 | 44.41007600 | 7946% |
| 14 | OpenMP | 100 | Bąbelkowy | 0.002696 | -0.00261740 | -333003% |
| Thread C++ | 0.009675 | -0.00959640 | -1220916% |
| OpenMP | Szybki | 0.000082 | -0.00007070 | -63694% |
| Thread C++ | 0.005953 | -0.00594200 | -5353153% |
| OpenMP | Przez scalanie | 0.000299 | 0.00008150 | 2144% |
| Thread C++ | 0.011117 | -0.01073650 | -282465% |
| OpenMP | Przez wstawianie | 0.000010 | -0.00000360 | -5294% |
| Thread C++ | 0.005343 | -0.00533580 | -7846765% |
| OpenMP | Przez wybieranie | 0.000029 | 0.00003060 | 5109% |
| Thread C++ | 0.006380 | -0.00631980 | -1055058% |
| OpenMP | 5 000 | Bąbelkowy | 0.042951 | 0.16017110 | 7885% |
| Thread C++ | 0.255884 | -0.05276200 | -2598% |
| OpenMP | Szybki | 0.001529 | -0.00033980 | -2857% |
| Thread C++ | 0.005443 | -0.00425360 | -35763% |
| OpenMP | Przez scalanie | 0.025208 | -0.00581050 | -2995% |
| Thread C++ | 0.034609 | -0.01521090 | -7842% |
| OpenMP | Przez wstawianie | 0.002710 | -0.00220300 | -43426% |
| Thread C++ | 0.205345 | -0.20483770 | -4037802% |
| OpenMP | Przez wybieranie | 0.024424 | 0.12246390 | 8337% |
| Thread C++ | 0.028776 | 0.11811190 | 8041% |
| OpenMP | 25 000 | Bąbelkowy | 0.839502 | 4.25058800 | 8351% |
| Thread C++ | 5.577540 | -0.48745000 | -958% |
| OpenMP | Szybki | 0.010234 | -0.00139770 | -1582% |
| Thread C++ | 0.009122 | -0.00028560 | -323% |
| OpenMP | Przez scalanie | 0.066251 | 0.03522760 | 3471% |
| Thread C++ | 0.132119 | -0.03064000 | -3019% |
| OpenMP | Przez wstawianie | 0.063366 | -0.06001170 | -178910% |
| Thread C++ | 3.627560 | -3.62420570 | -10804656% |
| OpenMP | Przez wybieranie | 0.591677 | 2.82225300 | 8267% |
| Thread C++ | 0.648323 | 2.76560700 | 8101% |
| OpenMP | 50 000 | Bąbelkowy | 3.207210 | 17.27179000 | 8434% |
| Thread C++ | 21.054200 | -0.57520000 | -281% |
| OpenMP | Szybki | 0.024217 | -0.00995930 | -6985% |
| Thread C++ | 0.014135 | 0.00012330 | 86% |
| OpenMP | Przez scalanie | 0.142654 | 0.05980400 | 2954% |
| Thread C++ | 0.255804 | -0.05334600 | -2635% |
| OpenMP | Przez wstawianie | 0.121690 | -0.11381760 | -144578% |
| Thread C++ | 18.353600 | -18.34572760 | -23303856% |
| OpenMP | Przez wybieranie | 2.090320 | 11.56448000 | 8469% |
| Thread C++ | 2.273630 | 11.38117000 | 8335% |
| OpenMP | 100 000 | Bąbelkowy | 12.414000 | 69.76056600 | 8489% |
| Thread C++ | 100.254000 | -18.07943400 | -2200% |
| OpenMP | Szybki | 0.053841 | -0.02463820 | -8437% |
| Thread C++ | 0.016959 | 0.01224370 | 4193% |
| OpenMP | Przez scalanie | 0.269275 | 0.13312300 | 3308% |
| Thread C++ | 0.511248 | -0.10885000 | -2705% |
| OpenMP | Przez wstawianie | 0.438489 | -0.42339490 | -280504% |
| Thread C++ | 79.125000 | -79.10990590 | -52411145% |
| OpenMP | Przez wybieranie | 8.022450 | 47.86512600 | 8565% |
| Thread C++ | 8.936360 | 46.95121600 | 8401% |
| 16 | OpenMP | 100 | Bąbelkowy | 0.003823 | -0.00374390 | -476323% |
| Thread C++ | 0.008733 | -0.00865470 | -1101107% |
| OpenMP | Szybki | 0.000042 | -0.00003120 | -28108% |
| Thread C++ | 0.012022 | -0.01201080 | -10820541% |
| OpenMP | Przez scalanie | 0.000305 | 0.00007520 | 1978% |
| Thread C++ | 0.012367 | -0.01198670 | -315356% |
| OpenMP | Przez wstawianie | 0.000021 | -0.00001370 | -20147% |
| Thread C++ | 0.006991 | -0.00698400 | -10270588% |
| OpenMP | Przez wybieranie | 0.000062 | -0.00000230 | -384% |
| Thread C++ | 0.006145 | -0.00608510 | -1015876% |
| OpenMP | 5 000 | Bąbelkowy | 0.038337 | 0.16478510 | 8113% |
| Thread C++ | 0.226714 | -0.02359200 | -1161% |
| OpenMP | Szybki | 0.001857 | -0.00066750 | -5612% |
| Thread C++ | 0.009589 | -0.00839980 | -70622% |
| OpenMP | Przez scalanie | 0.012597 | 0.00680050 | 3506% |
| Thread C++ | 0.038259 | -0.01886170 | -9724% |
| OpenMP | Przez wstawianie | 0.002191 | -0.00168400 | -33195% |
| Thread C++ | 0.174119 | -0.17361170 | -3422269% |
| OpenMP | Przez wybieranie | 0.021455 | 0.12543340 | 8539% |
| Thread C++ | 0.028153 | 0.11873550 | 8083% |
| OpenMP | 25 000 | Bąbelkowy | 0.802360 | 4.28773000 | 8424% |
| Thread C++ | 5.260630 | -0.17054000 | -335% |
| OpenMP | Szybki | 0.012195 | -0.00335820 | -3800% |
| Thread C++ | 0.025534 | -0.01669760 | -18896% |
| OpenMP | Przez scalanie | 0.107039 | -0.00556000 | -548% |
| Thread C++ | 0.139536 | -0.03805700 | -3750% |
| OpenMP | Przez wstawianie | 0.052918 | -0.04956410 | -147763% |
| Thread C++ | 4.082130 | -4.07877570 | -12159842% |
| OpenMP | Przez wybieranie | 0.533094 | 2.88083600 | 8438% |
| Thread C++ | 0.542870 | 2.87106000 | 8410% |
| OpenMP | 50 000 | Bąbelkowy | 2.854030 | 17.62497000 | 8606% |
| Thread C++ | 19.011600 | 1.46740000 | 717% |
| OpenMP | Szybki | 0.019663 | -0.00540500 | -3791% |
| Thread C++ | 0.017490 | -0.00323220 | -2267% |
| OpenMP | Przez scalanie | 0.133351 | 0.06910700 | 3413% |
| Thread C++ | 0.269085 | -0.06662700 | -3291% |
| OpenMP | Przez wstawianie | 0.132233 | -0.12436060 | -157970% |
| Thread C++ | 16.639400 | -16.63152760 | -21126375% |
| OpenMP | Przez wybieranie | 2.098410 | 11.55639000 | 8463% |
| Thread C++ | 2.129670 | 11.52513000 | 8440% |
| OpenMP | 100 000 | Bąbelkowy | 11.205500 | 70.96906600 | 8636% |
| Thread C++ | 83.198800 | -1.02423400 | -125% |
| OpenMP | Szybki | 0.034730 | -0.00552780 | -1893% |
| Thread C++ | 0.022629 | 0.00657380 | 2251% |
| OpenMP | Przez scalanie | 0.274670 | 0.12772800 | 3174% |
| Thread C++ | 0.519667 | -0.11726900 | -2914% |
| OpenMP | Przez wstawianie | 0.376984 | -0.36188990 | -239756% |
| Thread C++ | 66.468700 | -66.45360590 | -44026213% |
| OpenMP | Przez wybieranie | 8.335760 | 47.55181600 | 8508% |
| Thread C++ | 8.786850 | 47.10072600 | 8428% |
| 20 | OpenMP | 100 | Bąbelkowy | 0.004399 | -0.00432050 | -549682% |
| Thread C++ | 0.010328 | -0.01024910 | -1303957% |
| OpenMP | Szybki | 0.000038 | -0.00002660 | -23964% |
| Thread C++ | 0.011588 | -0.01157650 | -10429279% |
| OpenMP | Przez scalanie | 0.000284 | 0.00009590 | 2523% |
| Thread C++ | 0.015864 | -0.01548340 | -407351% |
| OpenMP | Przez wstawianie | 0.000044 | -0.00003690 | -54265% |
| Thread C++ | 0.007858 | -0.00785140 | -11546176% |
| OpenMP | Przez wybieranie | 0.000024 | 0.00003550 | 5927% |
| Thread C++ | 0.007490 | -0.00742960 | -1240334% |
| OpenMP | 5 000 | Bąbelkowy | 0.036818 | 0.16630430 | 8187% |
| Thread C++ | 0.154581 | 0.04854100 | 2390% |
| OpenMP | Szybki | 0.002266 | -0.00107660 | -9052% |
| Thread C++ | 0.010404 | -0.00921500 | -77476% |
| OpenMP | Przez scalanie | 0.024589 | -0.00519130 | -2676% |
| Thread C++ | 0.038793 | -0.01939490 | -9999% |
| OpenMP | Przez wstawianie | 0.002482 | -0.00197510 | -38934% |
| Thread C++ | 0.119368 | -0.11886070 | -2343006% |
| OpenMP | Przez wybieranie | 0.021675 | 0.12521300 | 8524% |
| Thread C++ | 0.030251 | 0.11663720 | 7941% |
| OpenMP | 25 000 | Bąbelkowy | 0.694582 | 4.39550800 | 8635% |
| Thread C++ | 3.498060 | 1.59203000 | 3128% |
| OpenMP | Szybki | 0.009495 | -0.00065840 | -745% |
| Thread C++ | 0.013091 | -0.00425500 | -4815% |
| OpenMP | Przez scalanie | 0.064510 | 0.03696870 | 3643% |
| Thread C++ | 0.135290 | -0.03381100 | -3332% |
| OpenMP | Przez wstawianie | 0.026152 | -0.02279750 | -67965% |
| Thread C++ | 2.704370 | -2.70101570 | -8052398% |
| OpenMP | Przez wybieranie | 0.518008 | 2.89592200 | 8483% |
| Thread C++ | 0.526596 | 2.88733400 | 8458% |
| OpenMP | 50 000 | Bąbelkowy | 2.606440 | 17.87256000 | 8727% |
| Thread C++ | 13.353400 | 7.12560000 | 3479% |
| OpenMP | Szybki | 0.014515 | -0.00025660 | -180% |
| Thread C++ | 0.026096 | -0.01183840 | -8303% |
| OpenMP | Przez scalanie | 0.162026 | 0.04043200 | 1997% |
| Thread C++ | 0.269017 | -0.06655900 | -3288% |
| OpenMP | Przez wstawianie | 0.102272 | -0.09439960 | -119912% |
| Thread C++ | 10.995500 | -10.98762760 | -13957151% |
| OpenMP | Przez wybieranie | 1.897600 | 11.75720000 | 8610% |
| Thread C++ | 2.055250 | 11.59955000 | 8495% |
| OpenMP | 100 000 | Bąbelkowy | 10.322200 | 71.85236600 | 8744% |
| Thread C++ | 56.379800 | 25.79476600 | 3139% |
| OpenMP | Szybki | 0.045617 | -0.01641400 | -5621% |
| Thread C++ | 0.022648 | 0.00655460 | 2245% |
| OpenMP | Przez scalanie | 0.298436 | 0.10396200 | 2584% |
| Thread C++ | 0.527467 | -0.12506900 | -3108% |
| OpenMP | Przez wstawianie | 0.325858 | -0.31076390 | -205884% |
| Thread C++ | 43.527900 | -43.51280590 | -28827692% |
| OpenMP | Przez wybieranie | 7.639170 | 48.24840600 | 8633% |
| Thread C++ | 8.311020 | 47.57655600 | 8513% |

# 6. Wykresy

# 7. Prawo Amdahla

Obliczono teoretyczną granicę przyspieszenia:

Gdzie:

* **f** to procentowa część kodu, której **nie da się zrównoleglić** (część szeregowa).

**Udział części szeregowej i równoległej**

Na podstawie pomiarów empirycznych oszacowano:

* jaka część działania algorytmu jest niezmiennie sekwencyjna,
* jaka część może być równolegle wykonywana,  
  co pozwoliło lepiej zrozumieć **potencjał zrównoleglenia** każdego algorytmu.

|  |  |  |
| --- | --- | --- |
| **Algorytm** | **Część szeregowa (%)** | **Część równoległa (%)** |
| Bąbelkowy | 10% | 90% |
| Szybki | 85% | 15% |
| Przez scalanie | 65% | 25% |
| Przez wstawianie | 100% | 0% |
| Przez wybieranie | 15% | 85% |

*Tabela 7.1. Procentowy udział części szeregowej i równoległej*

|  |  |  |  |
| --- | --- | --- | --- |
| **Threads&OpenMP** | | | **Przyspieszenie** |
| OpenMP | Bąbelkowy | 10.322200 | 7.96 |
| Thread C++ | Bąbelkowy | 56.379800 | 1.46 |
| OpenMP | Szybki | 0.045617 | 0.64 |
| Thread C++ | Szybki | 0.022648 | 1.29 |
| OpenMP | Przez scalanie | 0.298436 | 1.35 |
| Thread C++ | Przez scalanie | 0.527467 | 0.76 |
| OpenMP | Przez wstawianie | 0.325858 | 0.05 |
| Thread C++ | Przez wstawianie | 43.527900 | 0.00 |
| OpenMP | Przez wybieranie | 7.639170 | 7.32 |
| Thread C++ | Przez wybieranie | 8.311020 | 6.72 |

*Tabela 7.2. Przyspieszenie dla 20 wątków i rozmiarze problemu 100000 dla OpenMP i Threads.*

|  |  |  |
| --- | --- | --- |
| **CUDA** | | **Przyspieszenie** |
| Bąbelkowy | 0.216778 | 379.07 |
| Szybki | 1.47762 | 0.02 |
| Przez scalanie | 0.412467 | 0.98 |
| Przez wstawianie | 1663.396 | 0.00 |
| Przez wybieranie | 2402.572 | 0.02 |

*Tabela 7.3. Przyspieszenie dla 20 wątków i rozmiarze problemu 100000 dla CUDA.*

**Przyspieszenie (S)** mówi nam, **ile razy szybciej działa program uruchomiony na wielu wątkach w porównaniu do wersji jednowątkowej**.

Można je obliczyć, dzieląc czas wykonania programu na **jednym wątku (T₁)** przez czas wykonania tego samego programu na **wielu wątkach (Tₚ)**, gdzie *p* to liczba użytych wątków:

Wyniki tych pomiarów — dla różnych algorytmów, różnych wielkości danych i liczby wątków — zostały pokazane w tabeli 7.2. oraz 7.3 W tabeli znajdują się zarówno **czasy wykonania w sekundach**, jak i **obliczone przyspieszenia w procentach**.

W tabeli 7.4 obliczono Prawo Amdahla dla 20 wątków oraz 1408 wątków CUDA przy rozmiarze problemu równemu 100000.

|  |  |  |
| --- | --- | --- |
| **Technologia** | **Algorytm** | **Prawo Amdahla** |
| OpenMP | Bąbelkowy | 10.04500 |
| Thread C++ |
| CUDA | 10.00064 |
| OpenMP | Szybki | 1.18397 |
| Thread C++ |
| CUDA | 1.17658 |
| OpenMP | Przez scalanie | 1.55096 |
| Thread C++ |
| CUDA | 1.53864 |
| OpenMP | Przez wstawianie | 1.00000 |
| Thread C++ |
| CUDA | 1.00000 |
| OpenMP | Przez wybieranie | 6.70917 |
| Thread C++ |
| CUDA | 6.66727 |

*Tabela 7.4. Prawo Amdahla.*

**Próg opłacalności** to minimalna wielkość problemu, dla której zastosowanie programowania równoległego (np. z użyciem OpenMP, Threads lub CUDA) zaczyna przynosić rzeczywiste korzyści czasowe w porównaniu do wersji sekwencyjnej. Innymi słowy, jest to moment, w którym narzut związany z tworzeniem i zarządzaniem wątkami przestaje być większy niż zysk z równoległego przetwarzania. Dla małych danych uruchamianie wielu wątków może wydłużać czas działania programu, ale po przekroczeniu tego progu – zastosowanie wielowątkowości staje się opłacalne. Został oszacowany na bazie pomiarów czasów z rodziału 5.

|  |  |  |  |
| --- | --- | --- | --- |
| **Technologia** | **Algorytm** | **Liczba wątków** | **Próg opłacalności** |
| OpenMP | Bąbelkowy | 2 | 3500 |
| Threads | 10 | 4000 |
| CUDA | 32 | 5000 |
| OpenMP | Szybki | 2 | 22500 |
| Threads | 2 | 25000 |
| CUDA | 32 | 4500 |
| OpenMP | Przez scalanie | 2 | 500 |
| Threads | 2 | 4000 |
| CUDA | 32 | 5500 |
| OpenMP | Przez wstawianie | - | Brak opłacalności |
| Threads | - | Brak opłacalności |
| CUDA | - | Brak opłacalności |
| OpenMP | Przez wybieranie | 2 | 100 |
| Threads | 2 | 5000 |
| CUDA | - | Brak opłacalności |

*Tabela 7.5. Oszacowanie progu opłacalności.*

## 7.1. Ziarnistość problemu

Analizowano, czy dane algorytmy mają charakter:

* **gruboziarnisty** (duże, niezależne porcje danych – dobre do równoległości),
* czy **drobnoziarnisty** (częste zależności, trudne do podziału między wątki).

Większość klasycznych algorytmów sortowania, takich jak Merge Sort i Quick Sort, charakteryzuje się **średnią do grubej ziarnistości**, co pozwala na efektywne zrównoleglenie. Z kolei algorytmy takie jak Bubble Sort mają raczej **drobnoziarnistą strukturę**, przez co ich równoległość przynosi mniejsze zyski.

**Ziarnistość dla poszczególnych algorytmów:**

1. **Sortowanie bąbelkowe (Bubble Sort)**
   * **Ziarnistość: Drobnoziarnisty**
   * Sekcje krytyczne sprawiają, że efektywne zrównoleglenie jest trudne, a znacząca część czasu jest spędzana w sekcjach, które muszą być wykonywane szeregowo.
2. **Sortowanie szybkie (Quick Sort)**
   * **Ziarnistość: Gruboziarnisty**
   * Znaczna część pracy może być równoległa dzięki rekursywnemu podziałowi problemu, ale funkcja ‘partition’ pozostaje szeregowa.
3. **Sortowanie przez scalanie (Merge Sort)**
   * **Ziarnistość: Gruboziarnisty**
   * Większość pracy może być równoległa dzięki rekursywnemu podziałowi problemu, ale funkcja Przez wstawianie ‘merge’ pozostaje szeregowa.
4. **Sortowanie przez wstawianie (Insertion Sort)**
   * **Ziarnistość: Drobnoziarnisty**
   * Sekcje krytyczne w pętli wewnętrznej znacznie ograniczają efektywność zrównoleglenia. Większość pracy jest wykonywana w sekcji równoległej, ale konieczność synchronizacji zmniejsza korzyści.
5. **Sortowanie przez wybieranie (Selection Sort)**
   * **Ziarnistość: Drobnoziarnisty**
   * Sekcje krytyczne w pętli wewnętrznej znacznie ograniczają efektywność zrównoleglenia. Większość pracy jest wykonywana w sekcji równoległej, ale konieczność synchronizacji zmniejsza korzyści.

## 7.3. Recykling danych

W kontekście architektury CUDA, efektywność zrównoleglenia zależy od stopnia, w jakim algorytm może rozproszyć obliczenia między wiele wątków i bloków. Najlepiej skalujące się algorytmy to te, które mają niską część sekwencyjną (zgodnie z prawem Amdahla) oraz dobrze rozdzielają dane wejściowe między wątki.

Z przeprowadzonych testów wynika, że największy potencjał do zrównoleglenia na GPU wykazuje algorytm sortowania bąbelkowego. Pomimo że sam algorytm nie należy do najszybszych pod względem złożoności obliczeniowej, jego struktura pozwala na efektywne rozbicie operacji porównań i zamian między wątkami, co skutkuje bardzo wysokim przyspieszeniem (379x) oraz wartością prawa Amdahla przekraczającą 10.

Z kolei algorytmy takie jak sortowanie przez wstawianie czy szybkie sortowanie osiągnęły bardzo niskie przyspieszenia (0.02x), co oznacza, że większość ich wykonania nadal przebiega w sposób sekwencyjny albo struktura danych/operacji nie pozwala na ich efektywne równoleglenie w architekturze CUDA.

# 8. Wnioski

Nie każdy algorytm sortowania nadaje się dobrze do zrównoleglenia. Wydajność zależy od:

* struktury algorytmu (czy operacje są zależne czy niezależne),
* zastosowanej technologii (OpenMP, C++ Threads, CUDA),
* rozmiaru danych (małe rozmiary często nie przynoszą zysku z równoległości),
* narzutu synchronizacji i liczby rdzeni.

Zestawienie ogólnych informacji odnośnie przyspieszenia

|  |  |  |
| --- | --- | --- |
| **Algorytm** | **Najlepsza technologia** | **Powód** |
| Bąbelkowy | CUDA | Bardzo wysoki recykling danych, bardzo duże przyspieszenie (>370×). |
| Szybki | Threads C++ | |  | | --- | |  |  |  | | --- | | Lokalność przetwarzania, dobra rekurencyjna równoległość. | |
| Przez scalanie | OpenMP / CUDA | Najlepszy kandydat do zrównoleglenia – naturalne dzielenie problemu. |
| Przez wstawianie | Brak korzyści | Silne zależności między danymi – wydajność nawet się pogarsza. |
| Przez wybieranie | OpenMP | |  | | --- | |  |  |  | | --- | | Umiarkowane przyspieszenie, zależne od synchronizacji. | |

CUDA

* Daje spektakularne wyniki **tylko** dla algorytmów o dużej ziarnistości i wysokim recyklingu (np. Bubble Sort, Merge Sort).
* Dla algorytmów zależnych (Insertion, Selection) daje **negatywne przyspieszenie**, czas wykonania rośnie z powodu narzutu i synchronizacji.

Threads C++ vs OpenMP

* **OpenMP** lepiej radzi sobie przy **dużych rozmiarach danych** – prostsze zarządzanie wątkami i mniejsze opóźnienia synchronizacji.
* **std::thread** daje większą kontrolę, ale **większy narzut**, co czyni go mniej opłacalnym przy małych danych.
* W kilku przypadkach (np. szybki sort) **std::thread** wykazał się nieco lepszym przyspieszeniem, dzięki ręcznemu zarządzaniu rekurencją.

Próg opłacalności

* Dla większości algorytmów **opłaca się zrównoleglanie dopiero od rozmiarów > 5000 elementów**.
* Dla niektórych prostszych algorytmów (np. przez scalanie z OpenMP) opłacalność jest widoczna już od **500–1000** elementów.

Prawo Amdahla potwierdza ograniczenia

* Najwyższy teoretyczny speedup wg Amdahla osiągnęły:
  + **Bubble Sort (CUDA): ~10**
  + **Selection Sort (OpenMP): ~6.7**
* Sortowanie szybkie i przez wstawianie miały bliskie 1.0 → zrównoleglenie nie daje praktycznych korzyści.

Ziarnistość i recykling danych

* **Bubble Sort i Merge Sort** – wysoki recykling danych → lepsza efektywność na GPU (CUDA).
* **Quick Sort** – dobra równoległość, ale słaby recykling.
* **Insertion/Selection** – niska ziarnistość i wiele zależności → zły kandydat do CUDA i Threads.

# 9. Bibliografia

1. **Kirk, D. B., & Hwu, W. W.** (2016). *Programming Massively Parallel Processors: A Hands-on Approach* (3rd ed.). Morgan Kaufmann.
2. **OpenMP Architecture Review Board.** (2023). *OpenMP Application Programming Interface* (v5.2). <https://www.openmp.org>
3. **Sutter, H.** (2005). *The Concurrency Revolution*. C++ and Beyond.
4. **NVIDIA Corporation.** (2024). *CUDA C Programming Guide*. <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>
5. **Amdahl, G. M.** (1967). *Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities*. AFIPS Conference Proceedings.