Projekt: Rozwiązywanie Sudoku algorytmem kolorowania grafu

Data	Status projektu	Uwagi
2025-03-28	Wybór tematu	
2025-04-16	Rozpoczęty	
2025-05-02	Wykonanie programów	
2025-05-06	Wykonanie testów	

Autorka

Agnieszka Maleszka, 155941

Streszczenie

Celem projektu jest stworzenie solvera do Sudoku dowolnego rozmiaru , gdzie , który wykorzystuje algorytm kolorowania grafu do rozwiązania plansz Sudoku. Zaimplementowano trzy warianty: sekwencyjny, z użyciem OpenMP oraz z wykorzystaniem CUDA. Celem projektu jest porównanie tych podejść pod względem czasu działania i jakości rozwiązań.

Słowa kluczowe

- Sudoku
- Kolorowanie grafu
- OpenMP
- CUDA
- GPU
- Heurystyka
- Równoległość danych
- Problem NP-zupełny

Podstawy teoretyczne

- Modelowanie Sudoku jako problem kolorowania grafu:
  - Każde pole planszy Sudoku jest reprezentowane jako wierzchołek w grafie.
  - Krawędzie łączą pary pól znajdujących się w tym samym wierszu, kolumnie lub kwadracie .
  - Rozwiązanie Sudoku odpowiada poprawnemu kolorowaniu grafu przy użyciu kolorów – bez konfliktów między sąsiadującymi wierzchołkami.

- Klasa złożoności:
  - Sudoku należy do klasy problemów NP.
  - Kolorowanie grafu jest problemem NP-trudnym.

- Wersje implementacyjne:
  - **Sekwencyjna** – podstawowe kolorowanie grafu.
  - **OpenMP** – przyspieszenie przez wykorzystanie wielu rdzeni CPU.
  - **CUDA** – implementacja na GPU dla dużych plansz i populacji rozwiązań.

- Podejścia heurystyczne:
    - Kolorowanie zachłanne: kolejność odwiedzania wierzchołków wg stopnia lub losowa permutacja.
    - Algorytmy wspinaczkowe i inne metaheurystyki – rozważane jako rozszerzenie.

Format danych

- Plansze Sudoku wczytywane i zapisywane w formacie TXT.
- Puste pole jest reprezentowane jako 0.

0 6 0 0 0 0 0 0 0
7 9 1 0 0 8 0 4 0
4 0 0 6 0 0 1 3 5
0 0 0 4 8 0 0 9 0
3 0 0 0 0 0 0 0 2
8 2 6 3 9 0 0 7 0
0 0 0 0 5 0 9 0 0
0 0 0 7 0 1 0 6 4
0 0 5 0 0 6 3 0 0

Model matematyczny problemu

Planszę Sudoku rozmiaru (gdzie) można modelować jako graf nieskierowany , gdzie:

- — zbiór wierzchołków, odpowiadających polom planszy (),
- — zbiór krawędzi między parami pól, które nie mogą mieć tej samej wartości (czyli należą do tego samego wiersza, kolumny lub bloku).

Cel: Pokolorować wierzchołki grafu przy użyciu dokładnie kolorów (liczb od 1 do), tak aby żadne dwa sąsiednie wierzchołki nie miały tego samego koloru.

Zmienna decyzyjna:

Ograniczenia:

1. **Dokładnie jedna wartość w każdej komórce:**

2. **Każde sąsiedztwo ma różne wartości (kolory):**

3. **Dane wejściowe (początkowo wypełnione pola):**

Dla każdego wierzchołka o znanej wartości : (i dla).

Funkcja celu: (opcjonalna – jeśli np. chcemy minimalizować liczbę kolorów, co w Sudoku nie ma zastosowania, bo kolorów jest z góry)

Słownik pojęć

- **Sudoku** – łamigłówka logiczna na planszy , podzielonej na podkwadraty (gdzie), w której celem jest wypełnienie pól liczbami od 1 do tak, aby w każdym wierszu, kolumnie i podkwadracie każda liczba występowała dokładnie raz.

- **Graf** – struktura matematyczna składająca się ze zbioru wierzchołków i zbioru krawędzi

między nimi. W kontekście Sudoku, każdy wierzchołek reprezentuje jedno pole planszy, a krawędź łączy pola, które nie mogą mieć tej samej wartości.

- **Kolorowanie grafu** – przypisanie kolorów (liczb) wierzchołkom grafu tak, aby żadne dwa sąsiednie wierzchołki nie miały tego samego koloru. W Sudoku odpowiada to przypisaniu liczb do pól planszy zgodnie z regułami.

- **Problem NP-zupełny** – klasa problemów obliczeniowych, dla których nie znamy algorytmu rozwiązującego je w czasie wielomianowym, ale dla których można łatwo sprawdzić poprawność danego rozwiązania.

- **OpenMP** – biblioteka do równoległego przetwarzania danych na procesorach wielordzeniowych w języku C/C++ lub Fortran.

- **CUDA** – platforma programistyczna firmy NVIDIA pozwalająca na wykonywanie obliczeń równoległych na kartach graficznych (GPU).

- **Równoległość danych** – technika przetwarzania danych, w której wiele operacji jest wykonywanych jednocześnie na różnych fragmentach danych.

Opis problemu

Rozważany problem polega na rozwiązaniu dowolnej instancji Sudoku , gdzie , przy użyciu metod grafowych. Sudoku można sformalizować jako problem kolorowania grafu, w którym:

- Każde pole Sudoku reprezentowane jest jako wierzchołek grafu.
- Krawędź łączy dwa pola, jeśli należą do tego samego wiersza, kolumny lub bloku .
- Celem jest przydzielenie dokładnie kolorów (liczb 1..) w taki sposób, aby żadne dwa połączone wierzchołki nie miały tej samej wartości.

Problem ten zostanie rozwiązany z wykorzystaniem trzech podejść:

- **Sekwencyjnego** – przeszukiwanie i kolorowanie grafu bez równoległości.
- **Z użyciem OpenMP** – przyspieszenie przez równoległe rozpatrywanie wierzchołków.
- **Z użyciem CUDA** – masowe równoległe kolorowanie wielu kandydatów rozwiązania na GPU.

Celem projektu jest ocena skuteczności i wydajności każdej z metod w kontekście różnych instancji Sudoku.

Spis zaimplementowanych algorytmów

Lp	Algorytm / Narzędzie (KOD)	Kategoria	Przeznaczenie	Uwagi
1	[SUDOKU_SEQ](https://github.com/AgnieszkaMaleszka/Sudoku-solver/blob/main/src/seq.cpp)	Sekwencyjny (CPU)	Rozwiązanie Sudoku przez kolorowanie grafu	Działa dla plansz , gdzie
2	[SUDOKU_OMP](https://github.com/AgnieszkaMaleszk	OpenMP	Równoległe	Przyspieszeni

	a/Sudoku-solver/blob/main/src/omp.cpp)	(CPU)	kolorowanie planszy z użyciem wielu wątków	e przez równoległe kolorowanie wierszy/komórek
3	[SUDOKU_CUDA](https://github.com/AgnieszkaMaleszka/Sudoku-solver/blob/main/src/cuda.cu)	CUDA (GPU)	Masowe rozwiązywanie instancji Sudoku na GPU	Jeden blok GPU = jedna plansza
4	[SUDOKU_GEN](https://github.com/AgnieszkaMaleszka/Sudoku-solver/blob/main/src/sudoku_generator.cpp)	Generator danych	Generowanie losowych instancji Sudoku z kontrolowaną liczbą podpowiedzi	Format danych: TXT
5	[SUDOKU_VIS](https://github.com/AgnieszkaMaleszka/Sudoku-solver/blob/main/src/GUI.py)	Wizualizacja (Python)	Rysowanie planszy Sudoku (wejściowej i rozwiązania) oraz porównywanie wyników	Użyto biblioteki matplotlib / seaborn
6	[SUDOKU_TO_GRAPH_JSON](https://github.com/mojuser/sudoku-solver/blob/main/tools/sudoku_to_graph_json.cpp)	Konwersja danych	Przekształcanie instancji Sudoku do postaci grafowej w formacie JSON	Przydatne do analizy i wizualizacji grafów

Przykładowe dane wejściowe i wyniki działania algorytmu

Input grid	Output grid

Algorytm generowania danych testowych

Algorytm generuje poprawną planszę Sudoku rozmiaru (gdzie), a następnie usuwa z niej cyfry w kontrolowany sposób tak, aby wynikowa plansza miała dokładnie jedno rozwiązanie.

Proces przebiega następująco:

- **Wypełnienie przekątnych bloków** losowymi permutacjami liczb od 1 do – bloki te nie mają ze sobą zależności.
- **Rekurencyjne wypełnienie pozostałych komórek** – klasyczny backtracking, sprawdzane są warunki poprawności: wiersz, kolumna, blok.
- **Usuwanie cyfr (maskowanie)** – losowe zerowanie komórek z zachowaniem jednoznaczności (czyli po każdej próbie sprawdzana jest liczba możliwych rozwiązań).
- **Eksport planszy do pliku tekstowego** – wartości oddzielone spacją, puste pola zapisane jako `0`.

```
bool checkIfSafe(const vector<vector<int>> &grid, int i, int j, int num) {
```

```
  return (unUsedInRow(grid, i, num) &&
        unUsedInCol(grid, j, num) &&
        unUsedInBox(grid, i - i % boxSize, j - j % boxSize, num));
}
```

Założenia:

- Działa dla Sudoku , gdzie (np. 4, 9, 16).
- Każda wygenerowana plansza ma **dokładnie jedno rozwiązanie**.
- Możliwość sterowania liczbą pustych pól.

Złożoność czasowa:

- Generowanie pełnej planszy: (rekurencyjne wypełnianie).
- Sprawdzanie jednoznaczności po każdej zmianie: .

—

Przekształcanie planszy sudoku na graf

Plansza Sudoku jest reprezentowana jako graf nieskierowany, w którym: - Każde pole planszy to jeden wierzchołek grafu, - Krawędź łączy dwa pola, jeśli nie mogą zawierać tej samej wartości (czyli leżą w tym samym wierszu, kolumnie lub bloku), - Wartości początkowe (ustalone) są przechowywane w osobnej tablicy `fixedValues`.

Dla planszy o wymiarze `dim x dim` (np. 9×9), graf zawiera `dim * dim` wierzchołków. Dla każdego wierzchołka tworzone są listy sąsiadów (pola, z którymi współdzieli wiersz, kolumnę lub blok). Tworzony w ten sposób graf zawiera wiele klik — w każdej linii, kolumnie i bloku wszystkie wierzchołki są połączone między sobą (grafy całkowite w lokalnych grupach).

Przykład: Pole (0,0) jest połączone z: - wszystkimi polami z tego samego wiersza, - wszystkimi z tej samej kolumny, - wszystkimi z tego samego bloku 3×3 (dla dim=9).

Taka reprezentacja umożliwia wykorzystanie kolorowania grafu (graph coloring) jako modelu do rozwiązywania Sudoku metodami metaheurystycznymi (np. algorytmem genetycznym).

Dane grafu są zapisywane do pliku JSON w postaci: {

```
 "dim": 9,
 "size": 81,
 "fixedValues": [...],
 "adjacency": [
    [1,2,3,...],
    [0,2,4,...],
    ...
 ]
}
```

Gdzie: - `fixedValues[i]` to wartość początkowa w polu i (0 = puste), - `adjacency[i]` to lista sąsiadów pola i (pola, z którymi i nie może mieć tej samej liczby).

Testowanie rozwiązań

Dla każdej wygenerowanej planszy wykonywane jest porównawcze testowanie trzech wersji

solvera: sekwencyjnej, z użyciem OpenMP oraz CUDA. Proces testowy obejmuje:

1. Iteracja po numerach testów

- Dla testów generowane są kolejne pliki wejściowe w katalogu `./input/`.

2. Uruchamianie solverów

- Dla każdego testu uruchamiane są:
  - `seq.exe` – wersja sekwencyjna (CPU),
  - `omp.exe` – wersja równoległa OpenMP (CPU),
  - `cuda.exe` – wersja GPU z wykorzystaniem CUDA.
- Parametry wejściowe:
  - Plik wejściowy z planszą (`.txt`),
  - Plik wyjściowy z rozwiązaniem,
  - Plik pomiarowy zawierający czasy i parametry,
  - Parametry algorytmu: populacja, iteracje, szansa mutacji.

3. Nazewnictwo wyników

- Nazwy plików wynikowych zawierają zakodowane parametry testu, np.: `output/openmp_pop512_iter1500_mut0_05.txt`

Złożoność rozwiązywania:

- Sekwencyjnie: –
- OpenMP: , gdzie – liczba wątków
- CUDA: na jeden blok GPU (dla jednej planszy)

—

Założenia testów:

- Każdy algorytm przetwarza tę samą planszę i te same parametry.
- Wyniki zapisywane w osobnych plikach.

Algorytm sekwencyjny (CPU)

1. Reprezentacja Sudoku jako grafu

- Plansza modelowana jest jako graf nieskierowany , gdzie:
  - – pola planszy,
  - – połączenia między polami, które nie mogą mieć tej samej wartości (wiersz, kolumna, blok).
- Krawędzie są ustalane na podstawie pozycji wiersz/kolumna/blok.

```
for (int i = 0; i < dim; ++i) {
   for (int j = 0; j < dim; ++j) {
      int idx = i * dim + j;
      // Dodajemy sąsiadów z wiersza, kolumny i bloku
      ...
```

```
        sort(adjacency[idx].begin(), adjacency[idx].end());
        adjacency[idx].erase(unique(adjacency[idx].begin(), adjacency[idx].end()),
adjacency[idx].end());
    }
}
```

2. Inicjalizacja populacji rozwiązań

- Dla każdego osobnika tworzony jest losowy przydział wartości do zmiennych (komórek), z zachowaniem pól z wartościami ustalonymi.

```
if (graph.fixedValues[idx] != 0) {
    colors[idx] = graph.fixedValues[idx];
} else {
    // Wybierz losowo wartość nieużywaną przez sąsiadów
    ...
    colors[idx] = options[dist(rng)];
}
```

3. Ocena jakości rozwiązania (fitness)

- Liczba konfliktów (takich samych wartości u sąsiadów) liczona jest dla każdego osobnika. Celem jest osiągnięcie `fitness == 0`.

```
for (int u = 0; u < graph.size; ++u) {
    for (int v : graph.adjacency[u]) {
        if (!visited[v] && colors[u] == colors[v])
            fitness++;
    }
    visited[u] = true;
}
```

4. Krzyżowanie osobników

- Tworzenie potomka na podstawie dwóch rodziców – wybierana jest wartość od rodzica, który ma mniejszą liczbę konfliktów w danym polu.

```
child.colors[i] = (conflicts1 <= conflicts2) ? parent1.colors[i] : parent2.colors[i];
```

5. Mutacja osobnika

- Z określonym prawdopodobieństwem (`mutationRate`) zmieniane są pola powodujące konflikty.
- Wybierana jest losowo wartość, która nie powoduje kolizji (jeśli istnieje).

```
if (conflictCount[i] > 0 && dist(rng) < mutationRate) {
    ...
    colors[i] = options[valueDist(rng)];
}
```

6. Główna pętla algorytmu genetycznego

- Iteracyjny proces ewolucji osobników: selekcja, krzyżowanie, mutacja i ocena.
- Zatrzymanie przy `fitness == 0` lub po osiągnięciu limitu iteracji.

```
while (generation < maxGenerations) {
   sort(population.begin(), population.end(), ...);
   if (population[0].fitness == 0) break;

   // Selekcja, krzyżowanie, mutacja
   ...
   generation++;
}
```

7. Zapis rozwiązania i pomiar czasu

- Jeśli znaleziono poprawne rozwiązanie (bez konfliktów), zapisujemy je do pliku.
- Dodatkowo zapisywany jest czas wykonania i rozmiar planszy.

```
auto duration = chrono::duration<double>(endTime - startTime).count();
if (solution.fitness == 0) {
   measurmentOutFile << dim <<" "<< duration << endl;
   printSudoku(solution, dim, outFile);
}
```

Złożoność czasowa

- Inicjalizacja i ocena populacji:
- Krzyżowanie i mutacja:
- Całość: , gdzie:
  - – liczba osobników,
  - – liczba iteracji (generacji),
  - – wymiar planszy Sudoku.

Algorytm równoległy (OpenMP)

Wersja solvera Sudoku z równoległym przetwarzaniem za pomocą OpenMP oparta jest na algorytmie genetycznym. Celem jest przyspieszenie najbardziej kosztownych operacji: inicjalizacji populacji, oceny dopasowania (fitness) oraz tworzenia nowej generacji (krzyżowanie + mutacja).

1. Inicjalizacja populacji osobników (równoległa)

- Każdy osobnik w populacji otrzymuje losowe wartości w zmiennych (polach planszy), przy zachowaniu wartości z pól ustalonych.
- Równoległość osiągnięta poprzez `#pragma omp parallel for`.
- Każdy wątek korzysta z własnego generatora liczb losowych.

```
#pragma omp parallel for default(none) shared(population, graph) private(i)
```

```
for (int i = 0; i < populationSize; ++i) {
	mt19937 thread_rng(rng());
	population[i].initialize(graph, thread_rng);
	population[i].evaluate(graph);
}
```

2. Selekcja i tworzenie nowej populacji (równoległe)

- Najlepszy osobnik jest kopiowany (elitaryzm).
- Pozostali osobnicy są tworzeni równolegle przez:
  - Selekcję turniejową z populacji,
  - Krzyżowanie pary rodziców,
  - Mutację osobnika,
  - Ocena fitness.
- Każdy wątek tworzy własne dzieci.

```
#pragma omp parallel for
#pragma omp parallel for
for (int i = 1; i < populationSize; ++i) {
		mt19937 thread_rng(rng());
		Individual parent1 = tournamentSelection(population, 3);
		Individual parent2 = tournamentSelection(population, 3);
		Individual child = Individual::crossover(parent1, parent2, graph, thread_rng);
		child.mutate(graph, thread_rng, mutationRate);
		child.evaluate(graph);
		newPopulation[i] = move(child);
}
```

3. Mutacja i lokalne naprawy

- Dla każdego konfliktowego pola wykonywana jest próba zmiany koloru.
- Jeśli dostępne są kolory niepowodujące konfliktów – wybierany jest jeden z nich.
- W przeciwnym wypadku przypisywana jest losowa wartość.

```
if (graph.fixedValues[i] == 0 && conflictCount[i] > 0 && dist(rng) < mutationRate) {
   ...
   colors[i] = options[valueDist(rng)];
}
```

4. Zatrzymanie ewolucji

- Algorytm kończy działanie jeśli:
  - znaleziono rozwiązanie (`fitness == 0`),
  - osiągnięto maksymalną liczbę pokoleń (`maxGenerations`).

```
if (population[0].fitness == 0) break;
```

5. Pomiar czasu i zapis rozwiązania

- Pomiar czasu całkowitego działania przeprowadzany jest przy użyciu `chrono`.
- Jeśli rozwiązanie jest poprawne – zapisywane jest do pliku wynikowego.
- Dodatkowo zapisywana jest liczba sekund i rozmiar planszy.

```
if (solution.fitness == 0) {
   measurementOutFile << dim << " " << duration.count() << endl;
   printSudoku(solution, dim, outFile);
}
```

Złożoność czasowa

- Teoretyczna złożoność podobna jak w wersji sekwencyjnej:
  - – gdzie to liczba osobników, liczba generacji.
- W praktyce – dzięki równoległemu przetwarzaniu osobników – realny czas jest skrócony do około:
  - , gdzie to liczba dostępnych wątków.

Różnice względem wersji sekwencyjnej

- Wersja OpenMP znacząco skraca czas działania poprzez:
  - równoległe tworzenie i ocenę osobników,
  - równoległą mutację i krzyżowanie,
  - niezależne generatory RNG na każdy wątek.

Algorytm równoległy (CUDA)

Wersja CUDA przyspiesza najkosztowniejszy etap algorytmu genetycznego – mutację – poprzez przeniesienie tej operacji na GPU. Każdy osobnik z populacji reprezentuje jedno możliwe rozwiązanie Sudoku, a jego „genotyp" to przydział kolorów do wierzchołków grafu. Implementacja wykorzystuje strukturę grafu do przechowywania sąsiedztwa, a na GPU działają dedykowane kernele do mutacji oraz inicjalizacji stanu pseudolosowego.

1. Przetwarzanie równoległe (GPU) – mutacja osobników

- Mutacja to najintensywniejsza operacja – każda komórka analizuje swoje konflikty z sąsiadami i próbuje zmienić wartość, jeśli występuje kolizja.
- Każda komórka planszy jest przetwarzana przez jeden wątek.
- Warunek `fixedValues[idx] != 0` chroni ustalone wartości przed zmianą.

```
__global__ void mutateKernel(int* colors, const int* adjacency, const int* degrees, const int*
fixedValues,
                             int dim, int size, double mutationRate, curandState* states) {
   int idx = blockIdx.x * blockDim.x + threadIdx.x;
   if (idx >= size || fixedValues[idx] != 0) return;


   ...
   if (conflicts > 0 && curand_uniform_double(&states[idx]) < mutationRate) {
```

```
        ...
        colors[idx] = newVal;
    }
}
```

2. Losowanie na GPU (curand)

- Dla każdego wątku na GPU inicjalizowany jest stan pseudolosowy (curand).
- Generatory są niezależne, co zapewnia zróżnicowanie mutacji.

```
__global__ void setupKernel(curandState* states, unsigned long seed, int size) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < size) curand_init(seed, idx, 0, &states[idx]);
}
```

3. Inicjalizacja GPU – przygotowanie danych i pamięci

- Dane grafowe (sąsiedzi, stopnie, wartości ustalone) kopiowane są na GPU.
- Po stronie CPU generowana jest wersja spłaszczona list sąsiedztwa (`adjacency`).

```
CUDA_CALL(cudaMemcpy(d_adjacency, flatAdj.data(), sizeof(int) * size * maxDegree,
cudaMemcpyHostToDevice));
CUDA_CALL(cudaMemcpy(d_fixed, graph.fixedValues.data(), sizeof(int) * size,
cudaMemcpyHostToDevice));
```

4. Ocena rozwiązania (fitness) – wersja CPU lub GPU

- Obliczana liczba konfliktów – dla każdej pary sąsiadów, które mają identyczną wartość.

```
if (colors[idx] == colors[neighbor] && idx < neighbor)
    conflicts++;
```

W wersji CUDA kernel `evaluateKernel()` może być uruchamiany zbiorczo dla wielu osobników.

5. Integracja z algorytmem genetycznym

- GPU wykorzystywane jest w funkcji `mutate()` klasy `GPUMutator`, która zarządza pamięcią GPU oraz wywołaniem kernela.
- Po każdej generacji CUDA wykonuje mutację wektorów kolorów osobników, po czym wynik wraca do CPU w celu dalszego przetwarzania.

```
gpu.mutate(child.colors, mutationRate);
child.evaluate(graph);
```

6. Przykład tworzenia obiektu GPUMutator i użycia mutacji

```
GPUMutator gpu(graph);

for (...) {
```

```
    ...
    gpu.mutate(child.colors, mutationRate);
}
```

Złożoność i korzyści z CUDA

- Złożoność czasowa algorytmu pozostaje bez zmian:
- CUDA przyspiesza najbardziej kosztowny etap – mutację – zmniejszając jej czas z do na wątek.
- Całkowity czas generacji jest znacząco zredukowany przy większych populacjach lub planszach ().
- Mutacja na GPU minimalizuje wąskie gardło przetwarzania danych – każdy osobnik mutowany równolegle.

Różnice względem wersji OpenMP

- OpenMP używa CPU do równoległości poziomu osobnika, CUDA używa GPU do przetwarzania **komórek**.
- CUDA wymaga alokacji i transferu danych na GPU, ale zapewnia większe przyspieszenie przy dużych instancjach.
- Wersja CUDA lepiej się skaluje dla dużych populacji (np. >1024 osobników).

Środowisko testowe

Testy porównawcze przeprowadzono w jednolitych warunkach, aby zapewnić spójność pomiarów czasów wykonania i jakości rozwiązań. Poniżej przedstawiono parametry sprzętu i oprogramowania użytego w eksperymentach.

System operacyjny

- **Windows 11 Pro**

Kompilatory

- **g++**: 11.2.0 (MinGW)
- **CUDA Toolkit**: 12.8.93
- **nvcc** (CUDA compiler): release 12.8, build cuda_12.8.r12.8/compiler.35583870_0

CPU

- **Model**: AMD Ryzen 7 5800H with Radeon Graphics
- **Liczba rdzeni/wątków**: 8 rdzeni fizycznych, 16 wątków (SMT)
- **Cache**:
  - L1: 512 KB
  - L2: 4 MB
  - L3: 16 MB
- **Architektura**: x86_64 (64-bitowa)

RAM

- **Pamięć operacyjna**: 16 GB

GPU

- **Dedykowany GPU**: NVIDIA GeForce GTX 1650 Laptop GPU
  - CUDA Compute Capability: 7.5
  - Liczba rdzeni CUDA: 1024
  - Max threads per block: 1024
  - Liczba multiprocesorów (SM): 16
  - Wersja CUDA: 12.8
  - Sterownik NVIDIA: 572.61

Skrypt testujący Sudoku Solver

Skrypt `run_tests.cpp` automatyzuje proces **testowania rozwiązywacza Sudoku** w trzech wariantach:

- `sekwencyjny` (`seq.exe`)
- `OpenMP` (`omp.exe`)
- `CUDA` (`cuda.exe`)

W ramach działania:

- Generowane są losowe instancje Sudoku (`sudoku_generator.exe`)
- Każda implementacja rozwiązuje te same plansze
- Wyniki i czasy działania zapisywane są do plików

Parametry

Parametry konfiguracyjne ładowane są z pliku `config.json`:

```
{
  "tests": 10,
  "grid_size": 9,
  "empty_ratio": 0.62,
  "population": 512,
  "iterations": 1500,
  "mutation": 0.05
}
```

Parametr	Znaczenie
`tests`	Liczba testów
`grid_size`	Rozmiar planszy (np. 9 dla 9×9)
`empty_ratio`	Ułamek pustych pól (np. 0.62 = 62%)
`population`	Rozmiar populacji algorytmu genetycznego
`iterations`	Liczba iteracji GA
`mutation`	Szansa mutacji (np. 0.05 = 5%)

Fragment kodu

```
json config = loadConfig("config.json");
int N = config["tests"];
string populationSize = to_string(config["population"]);
string mutationChance = to_string(config["mutation"].get<double>());
...
string cmd = executable + " " + inFile + " " + outFile + " " + measFile +
          " " + populationSize + " " + iterations + " " + mutationChance;
system(cmd.c_str());
```

Dane wyjściowe

- Rozwiązania → `output/solution/`
- Czasy i parametry → `output/sequential_*.txt`, `openmp_*.txt`, `cuda_*.txt`

Każda implementacja działa na identycznym zestawie danych wejściowych, co umożliwia uczciwe porównanie wydajności.

Pomiary czasu

Plan testów i pomiarów

Nr	Cel testu	Rozmiar planszy	Puste pola (%)	Populacja	Iteracje	Mutacja	Liczba testów	Algorytmy do porównania
1	Porównanie SEQ / OMP / CUDA	9×9	50	512	1500	0.05	10	SEQ, OMP, CUDA
2	Wpływ pustych pól (skala)	9×9	30 / 40 / 50 / 60 / 70	512	1500	0.05	5–10/test	SEQ, OMP, CUDA
3	Wpływ wielkości planszy	4×4 / 9×9 / 16×16 / 25×25 / 36×36	50	1024	3000	0.05	3–5/test	SEQ, OMP, CUDA
4	Wpływ rozmiaru populacji	9×9	50	256 / 512 / 1024 / 2048	1500	0.05	5/test	OMP, CUDA
5	Wpływ liczby iteracji	9×9	50	512	500 / 1000 / 2000 / 3000	0.05	5/test	OMP, CUDA
6	Wpływ prawdopodobieństwa mutacji	9×9	50	512	1500	0.01 / 0.05 / 0.1	5/test	OMP, CUDA

Uwagi

- Wszystkie testy są wykonywane na tych samych instancjach Sudoku (ta sama liczba i rozkład pustych pól).
- W testach CUDA używana jest karta NVIDIA GTX 1650.
- W testach OMP maksymalna liczba wątków: 16.
- Wyniki pomiarów zapisywane są do osobnych plików `.txt` i mogą być analizowane przez skrypt w Pythonie.
- Ponieważ wersja sekwencyjna nie wykorzystuje w pełni zalet parametrów GA (brak równoległości, skalowalność) nie uwzględniamy jej w testach 4-6.

Wyniki testów

2) Wpływ pustych pól (skala)

Rozmiar	Liczba testów	Średni czas SEQ	Średni czas OMP	Średni czas CUDA	Populacja	Iteracje	Mutacja	Puste pola (%)
9×9	5	0.14	0.03	0.16	512	1500	0.05	30
9×9	5	0.57	0.04	0.93	512	1500	0.05	40
9×9	5	0.97	0.07	1.00	512	1500	0.05	50
9×9	5	1.00	0.13	1.33	512	1500	0.05	60
9×9	5	1.51	0.23	2.50	1024	1500	0.05	70

Wnioski:

- Im więcej pustych pól w planszy Sudoku, tym większa liczba potencjalnych konfliktów, co skutkuje rosnącym czasem obliczeń.
- SEQ i CUDA wykazują wyraźny wzrost czasu wraz ze wzrostem trudności instancji (liczby pustych pól).
- OMP pozostaje znacznie szybszy, zachowując niskie czasy nawet dla plansz z 70% pustych komórek.
- CUDA zaczyna tracić przewagę wydajnościową przy bardziej „niedookreślonych” planszach — być może z powodu większego narzutu synchronizacji GPU.

3) Wpływ wielkości planszy

Rozmiar	Liczba testów	Średni czas SEQ	Średni czas OMP	Średni czas CUDA	Populacja	Iteracje	Mutacja	Puste pola (%)
4×4	5	0.017	0.023	0.196	1024	1500	0.05	50
9×9	5	0.97	0.07	1.00	1024	1500	0.05	50
16×16	5	8.48	6.28	5.01	1024	1500	0.05	50
25×25	3	11.50	9.58	7.86	1024	1500	0.05	50

Wnioski:

- Czas obliczeń rośnie wraz z rozmiarem planszy — najbardziej zauważalnie w implementacji sekwencyjnej.
- CUDA zyskuje przewagę przy większych rozmiarach (16×16, 25×25), lepiej skalując się przy wzroście liczby danych.
- OpenMP wypada najlepiej przy średnich rozmiarach (9×9), dzięki niskiemu narzutowi

uruchamiania i efektywnej pracy wielowątkowej.

Wszystkie testy w 4–6 były wykonane na tym samym zestawie planszy.

4) Wpływ rozmiaru populacji

Rozmiar	Liczba testów	Średni czas OMP	Średni czas CUDA	Populacja	Iteracje	Mutacja	Puste pola (%)
9×9	5	0.06	0.62	256	1500	0.05	50
9×9	5	0.07	1.37	512	1500	0.05	50
9×9	5	0.18	1.60	1024	1500	0.05	50
9×9	5	0.29	2.72	2048	1500	0.05	50

Wnioski:

- Większe populacje poprawiają jakość rozwiązań, ale zwiększają czas działania.
- CUDA traci efektywność przy bardzo dużych populacjach z powodu ograniczeń sprzętowych.
- OpenMP skaluje się bardziej przewidywalnie w tym zakresie.

5) Wpływ liczby iteracji

Rozmiar	Liczba testów	Średni czas OMP	Średni czas CUDA	Populacja	Iteracje	Mutacja	Puste pola (%)
9×9	5	0.09	0.81	512	500	0.05	50
9×9	5	0.07	0.95	512	1000	0.05	50
9×9	5	0.10	0.90	512	2000	0.05	50
9×9	5	0.11	0.99	512	3000	0.05	50
9×9	5	0.10	0.79	512	6000	0.05	50
9×9	5	0.11	0.84	512	10000	0.05	50

Wnioski:

- W OMP czas działania stabilizuje się po ~1000 iteracjach – możliwe lepsze dopasowanie wcześniej.
- CUDA wykazuje zmienność, ale generalnie zyskuje na większej liczbie iteracji.
- Najlepszy kompromis czas/jakość to zwykle 1500–3000 iteracji.

6) Wpływ prawdopodobieństwa mutacji

Rozmiar	Liczba testów	Średni czas OMP	Średni czas CUDA	Populacja	Iteracje	Mutacja	Puste pola (%)
9×9	5	0.07	0.89	512	1500	0.01	50
9×9	5	0.10	0.94	512	1500	0.05	50
9×9	5	0.12	2.64	512	1500	0.10	50

Wnioski:

- Niskie wartości mutacji dają stabilny czas wykonania.
- CUDA silnie traci wydajność przy zbyt wysokiej mutacji – zbyt duży chaos w populacji.
- OMP bardziej odporne na wahania tego parametru.

Wnioski efektywnościowe (speedup)

Współczynniki przyspieszenia (speedup) obliczono względem wersji sekwencyjnej dla tego samego testu:

Rozmiar	Parametry	SEQ [s]	OMP [s]	CUDA [s]	Speedup OMP	Speedup CUDA
9×9	pop=512, iter=1500, mut=0.05	1.00	0.08	1.89	12.5×	0.53×
9×9	pop=512, iter=1500, mut=0.05, empty=30%	0.14	0.23	0.16	0.61×	0.88×
9×9	pop=1024, iter=1500, mut=0.05, empty=70%	1.51	0.023	2.50	65.65×	0.60×
4×4	pop=1024, iter=1500, mut=0.05	0.017	0.023	0.196	0.74×	0.09×
16×16	pop=1024, iter=1500, mut=0.05	8.48	6.28	5.01	1.35×	1.69×
25×25	pop=1024, iter=1500, mut=0.05	11.50	9.58	7.86	1.20×	1.46×

Wnioski:

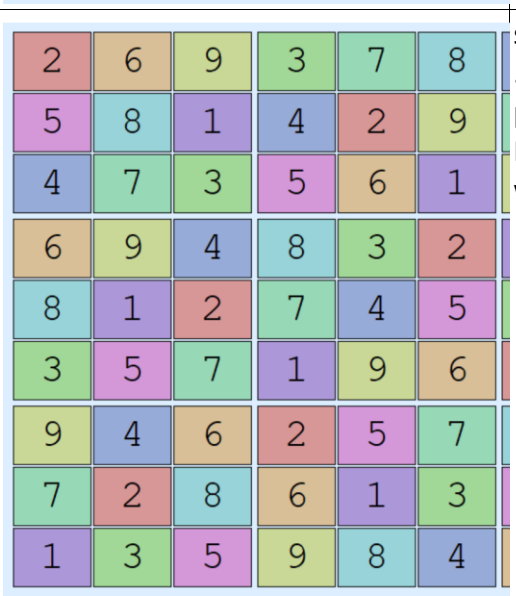
- CUDA zyskuje znacząco dopiero przy większych planszach – najlepiej skalując się dla 16×16 i 25×25.
- OMP osiąga imponujący przyspieszenie przy małych i średnich instancjach (nawet 12.5× na 9×9 z domyślnymi parametrami).
- Dla przypadków z bardzo dużą trudnością (np. 70% pustych pól) CUDA nie zawsze daje zysk czasowy – przeciążenie GPU i trudności z konwergencją zwiększają czas.
- CUDA zyskuje efektywność dopiero przy odpowiednio dużej populacji i liczbie iteracji oraz większej planszy – wtedy wyraźnie pokonuje OMP i SEQ.
- OpenMP zapewnia dobrą wydajność bez dużych narzutów uruchomieniowych – sprawdza się jako rozwiązanie uniwersalne.

Testy na planszach sudoku znalezionych w innych materiałach

Testowanie wykonano dla parametrów: populacja = 1024, iteracje = 3000, mutacja = 5%.

Sudoku	Rozwiązanie	Źródło	SEQ [s]	OMP [s]	CUDA [s]

		Logi-Mix nr 201, marzec 2025	41.81	1.35	3.20
		sudoku.com – poziom Trudny	1.88	2.42	—

		sudoku.com – poziom Ekspert	1.31	2.98	—
		sudoku.com – poziom Mistrzowski	1.79	2.30	—

Podsumowanie projektu

Projekt dotyczył rozwiązania Sudoku dowolnego rozmiaru przy użyciu algorytmu kolorowania grafu. Zrealizowano trzy wersje implementacji:

- **Sekwencyjną** (CPU),
- **OpenMP** (CPU, wielowątkowa),
- **CUDA** (GPU, masywnie równoległa).

W ramach projektu opracowano również:

- generator plansz z kontrolą liczby pustych pól i jednoznacznością rozwiązania,

- GUI do wizualizacji działania solvera i interakcji z użytkownikiem,
- narzędzia do automatyzacji testów i pomiarów wydajności.

Podsumowanie projektu

Projekt dotyczył rozwiązywania Sudoku dowolnego rozmiaru `n×n` z wykorzystaniem **algorytmu kolorowania grafu**, zaimplementowanego w trzech wariantach:

- **Wersja sekwencyjna** (jednowątkowa, CPU),
- **Wersja równoległa OpenMP** (wielowątkowa, CPU),
- **Wersja CUDA** (masywnie równoległa, GPU).

W ramach prac wykonano również:

- **Generator plansz** z kontrolą liczby pustych pól oraz zapewnieniem jednoznaczności rozwiązania,
- **GUI** do wizualizacji działania algorytmu oraz interaktywnego rozwiązywania Sudoku,
- **System testów i pomiarów wydajności**, pozwalający porównać implementacje i przeanalizować wpływ parametrów.

—

Wnioski końcowe

- **OpenMP** zapewnia największy zysk czasowy dla plansz 9×9 — w wielu przypadkach **2–12× szybszy** niż wersja sekwencyjna, dzięki niskim narzutom uruchomieniowym.
- **CUDA** osiąga przewagę przy **większych instancjach** (16×16, 25×25) oraz przy dużych populacjach — czas skraca się nawet o **30–40%** względem CPU.
- **Liczba pustych pól** silnie wpływa na trudność problemu — im więcej pustych komórek, tym trudniejszy i wolniejszy proces znajdowania poprawnego rozwiązania.
- **Parametry algorytmu GA** (rozmiar populacji, liczba iteracji, współczynnik mutacji) mają kluczowy wpływ zarówno na **czas działania**, jak i **jakość rozwiązań**.
- **Model grafowy** (kolorowanie wierzchołków) jest uniwersalny i dobrze adaptuje się zarówno do środowisk sekwencyjnych, jak i równoległych.
- Projekt potwierdza, że **wykorzystanie równoległości obliczeniowej** (zarówno CPU, jak i GPU) pozwala efektywnie rozwiązywać problemy kombinatoryczne klasy **NP**.

Repozytorium z kodem źródłowym

https://github.com/AgnieszkaMaleszka/Sudoku-solver/tree/main

Literatura

- https://medium.com/code-science/sudoku-solver-graph-coloring-8f1b4df47072
- https://www.geeksforgeeks.org/graph-coloring-set-2-greedy-algorithm/