

Kolorowanie grafu za pomocą algorytmu genetycznego

Data	Status projektu	Uwagi
2024-03-29	Wybór tematu	Nie później niż 2024-03-31
2023-05-10	Sekwencyjny	
2023-05-22	Open MP	
2023-05-25	CUDA	
2023-05-29	Poprawki	
2023-05-29	Pomiary wydajności	
2023-05-29	Sprawozdanie	

- Kolorowanie grafu za pomocą algorytmu genetycznego
 - Autor
 - Streszczenie
 - Opis algorytmu
 - Fragmenty Kodu
 - OpenMP
 - CUDA
 - Testy wydajnościowe
 - Test 1 - rozmiar populacji
 - Test 2 - rozmiar populacji
 - Test 3 - ilość wierzchołków
 - Wnioski
 - Dane sprzętowe
 - Repozytorium z kodem źródłowym
 - Źródła

Autor

Wiktor Jordaczka 151785

Streszczenie

Projekt miał na celu przyspieszenie algorytmu genetycznego wykorzystywanego do kolorowania grafów łącząc Open MP i CUDA. Wykorzystanie OpenMP znacznie skróciło czas obliczeń. Połączenie OpenMP z CUDA jeszcze bardziej przyspieszyło algorytm.

Opis algorytmu

1. Inicjalizacja populacji: Algorytm zaczyna od losowego wygenerowania populacji początkowej. Każdy osobnik w populacji reprezentowany jest przez chromosom, gdzie każdy gen odpowiada kolorowi przypisanemu wierzchołkowi grafu.
2. Obliczanie jakości rozwiązania (fitness): Dla każdego osobnika w populacji obliczana jest jego ilość wykorzystanych kolorów oraz ilość konfliktów. Osobnik jest lepszy jeśli ma mniej konfliktów, następnie mniej kolorów.
3. Korekta kolorów: Wymuszane jest na wszystkich osobnikach, aby korzystały z mniejszej ilości kolorów niż dotychczasowy najlepszy osobnik oraz aby były to kolejne kolory numerowane od 0.
4. Selekcja turniejowa: Wybierane są pary osobników z populacji za pomocą turnieju. Dla każdej pary losowanych jest kilka osobników, a następnie wybierany jest ten z najlepszym dopasowaniem.
5. Krzyżowanie: Wybrane pary rodziców są krzyżowane jednopunktowo, aby wygenerować potomstwo.
6. Mutacja: Potomstwo jest mutowane w celu wprowadzenia różnorodności genetycznej w populacji.
7. Zastąpienie starej populacji nową populacją.
8. Algorytm powtarza punkty 2-7 przez określoną liczbę generacji lub do minięcia zadanej liczby sekund.

Fragmenty Kodu

OpenMP

Przyśpieszanie obliczeń

```
// Sprawdzamy aktualną liczbę kolorów najlepszego rozwiązania i usuwamy ich potencjalne nadwyżki u innych osobników
void correct(Specimen* population, int numOfColors, int numVertices, int populationSize, mt19937& rng){
    #pragma omp parallel for default(None) shared(populationSize, numVertices, numOfColors, population, rng)
    for (int i = 0; i < populationSize; i++) {
        if (population[i].numOfColors != numOfColors)
            for (int j = 0; j < numVertices; j++)
                if (population[i].colors[j] >= numOfColors - 1) // zastępujemy losowym kolorem
                    population[i].colors[j] = randomNumber(0, numOfColors - 2, rng);
    }
}
```

```
// Turnieje i krzyżowanie (wybór nowej populacji)
#pragma omp parallel for default(None) shared(populationSize, numVertices, population, newPopulation, rng)
for (int i = 0; i < populationSize; i+=2) {
    Specimen parent1 = tournamentSelection(population, populationSize, rng); // wybieramy rodziców z turniejów
    Specimen parent2 = tournamentSelection(population, populationSize, rng);
    Specimen* offspring1 = new Specimen; // przydzielimy pamięć dla potomków
    offspring1->colors = (int*)malloc(numVertices * sizeof(int));
    Specimen* offspring2 = new Specimen;
    offspring2->colors = (int*)malloc(numVertices * sizeof(int));
    crossover(parent1, parent2, *offspring1, *offspring2, numVertices, rng); // krzyżujemy i otrzymujemy potomków
    memcpy(newPopulation[i].colors, offspring1->colors, numVertices * sizeof(int)); // kopiujemy cechy potomków
    memcpy(newPopulation[i + 1].colors, offspring2->colors, numVertices * sizeof(int));
    free(offspring1->colors);
    free(offspring2->colors);
    delete offspring1;
    delete offspring2;
}
```

```

        delete offspring;
    }

    // Mutacje
#pragma omp parallel for default(None) shared(populationSize, numOfVertices, numOfColors, mutationChance, newPopulation, rng)
for (int i = 0; i < populationSize; i++) {
    if (randomNumber(0, 100, rng) < mutationChance) {
        mutateOld(newPopulation[i], numOfColors, numOfVertices, rng);
    }
}

// obliczamy jakość osobników w populacji
#pragma omp parallel for default(None) shared(populationSize, numOfVertices, adjacencyMatrix, newPopulation)
for (int i = 0; i < populationSize; i++) {
    calculateFitness(numOfVertices, adjacencyMatrix, newPopulation[i]); // sprawdzanie jakości rozwiązania
}

// kopiujemy nową populację w miejsce starej
#pragma omp parallel for default(None) shared(populationSize, numOfVertices, population, newPopulation)
for (int i = 0; i < populationSize; ++i) {
    memcpy(population[i].colors, newPopulation[i].colors, numOfVertices * sizeof(int));
    population[i].numOfColors = newPopulation[i].numOfColors;
    population[i].numOfConflicts = newPopulation[i].numOfConflicts;
}

```

Alokacje i dealokacje pamięci

```

// inicjalizacja populacji
void initializePopulation(Specimen* population, int populationSize, int numOfVertices, mt19937& rng) {
    #pragma omp parallel for default(None) shared(populationSize, numOfVertices, population, rng)
    for (int i = 0; i < populationSize; ++i) {
        population[i].colors = (int*)malloc(numOfVertices * sizeof(int));
        for (int j = 0; j < numOfVertices; ++j) {
            population[i].colors[j] = randomNumber(0, numOfVertices - 1, rng);
        }
    }
}

```

```

// zwalniamy pamięć populacji i macierzy
#pragma omp parallel for default(None) shared(populationSize, population)
for (int i = 0; i < populationSize; ++i) {
    free(population[i].colors);
}
free(population);
#pragma omp parallel for default(None) shared(adjacencyMatrix, numOfVertices)
for (int i = 0; i < numOfVertices; i++)
    delete[] adjacencyMatrix[i];
delete[] adjacencyMatrix;

```

CUDA

Obliczanie jakości osobnika (fitness)

```

// Kernel CUDA - każdy wątek oblicza jakość innego osobnika
__global__ void cudaCalculateFitnessKernel(int numOfVertices, int* d_adjacencyMatrix, int* d_colors, int* d_colorNum, int* d_conflictNum, int populationSize) {
    int idx = blockDim.x * blockIdx.x + threadIdx.x; // thread (specimen) number
    if (idx >= populationSize) return; // sprawdzamy czy nie jesteśmy poza populacją (ilość wątków to wielokrotność 256)
    d_conflictNum[idx] = 0; // inicjujemy na 0
    bool* colorSet = new bool[numOfVertices]; // użyte kolory
    std::memset(colorSet, 0, numOfVertices * sizeof(bool)); // inicjujemy jako false

    for (int i = 0; i < numOfVertices; i++)
    {
        for (int j = i + 1; j < numOfVertices; j++) // sprawdzamy tylko od wierzchołka do końca zakresu, by uniknąć powtórzeń
        {

```

```

        if (_adjacencyMatrix[i * numVertices + j] == 1 && _colors[idx * numVertices + i] == _colors[idx * numVertices + j]) // szukamy wierzchołka sąsiadniego z tym samym kolorem
    {
        d_conflictNum[idx]++;
    }
}
colorSet[_colors[idx * numVertices + i]] = true; // kolor użyty
}

// poprawiamy kolory, aby były kolejnymi liczbami naturalnymi od 0
while (_conflictNum[idx] == 0) { // Poprawiamy tylko dla rozwiązań bezkonfliktowych
    int maxColor = numVertices - 1;
    while (maxColor >= 0 && !colorSet[maxColor]) {
        maxColor--;
    }

    bool allColorsPresent = true; // sprawdzamy czy brakuje jakiegoś koloru
    for (int i = 0; i <= maxColor; i++) {
        if (!colorSet[i]) { // poprawiamy kolory
            allColorsPresent = false;
            int missingColor = i;

            // Obniżamy kolory o 1
            for (int j = 0; j < numVertices; j++) {
                if (_colors[idx * numVertices + j] > missingColor) {
                    _colors[idx * numVertices + j]--;
                }
            }

            colorSet[missingColor] = true;
            colorSet[maxColor] = false;
            break; // Restart sprawdzania
        }
    }
    if (allColorsPresent) { // wszystko ok
        break;
    }
}

_d_colorNum[idx] = 0; // ustawiamy liczbę kolorów
for (int i = 0; i < numVertices; i++) {
    if (colorSet[i]) {
        d_colorNum[idx]++;
    }
}

delete[] colorSet; // zwalniamy pamięć
return;
}

// obliczanie jakości osobników w populacji
void cudaCalculateFitness(int numVertices, int* _adjacencyMatrix, Specimen* population, int populationSize, int blockSize, int numBlocks) {
    // spłaszczone tablica struktur Specimen
    int* h_colors = new int[populationSize * numVertices]; // spłaszczone tablice kolorów osobników
    int* h_colorNum = new int[populationSize]; // spłaszczone ilości kolorów osobników
    int* h_conflictNum = new int[populationSize]; // spłaszczone ilości konfliktów osobników

    // spłaszczymy tablicę struktur i podtablice kolorów
    #pragma omp parallel for default(none) shared(populationSize, numVertices, h_colors, population)
    for (int i = 0; i < populationSize; i++) {
        for (int j = 0; j < numVertices; j++) {
            h_colors[(i * numVertices + j)] = population[i].colors[j];
        }
    }

    int* d_colors; // macierz kolorów GPU
    size_t colorsSize = populationSize * numVertices * sizeof(int);

```

```

cudaMalloc(&d_colors, colorsSize);
cudaMemcpy(d_colors, h_colors, colorsSize, cudaMemcpyHostToDevice);
int* d_colorNum; // wektor il. kolorów GPU
int* d_conflictNum; // wektor il. konfliktów GPU
size_t numVecSize = populationSize * sizeof(int); // rozmiar wektorów
// wartości zostaną obliczone na nowo, więc wystarczy alokacja bez kopiowania
cudaMalloc(&d_colorNum, numVecSize);
cudaMalloc(&d_conflictNum, numVecSize);

// uruchamiamy kernel
cudaCalculateFitnessKernel<<<numBlocks, blockSize>>>(numOfVertices, d_adjacencyMatrix, d_colors, d_colorNum, d_conflictNum, populationSize);
// pobieramy wynikowe dane z GPU
cudaMemcpy(h_colorNum, d_colorNum, numVecSize, cudaMemcpyDeviceToHost);
cudaMemcpy(h_conflictNum, d_conflictNum, numVecSize, cudaMemcpyDeviceToHost);
cudaMemcpy(h_colors, d_colors, colorsSize, cudaMemcpyDeviceToHost);

// kopiuujemy spłaszczone dane do populacji
#pragma omp parallel for default(None) shared(populationSize, numOfVertices, h_colors, population)
for (int i = 0; i < populationSize; i++) {
    for (int j = 0; j < numOfVertices; j++) {
        population[i].colors[j] = h_colors[(i * numOfVertices + j)];
    }
    population[i].numOfColors = h_colorNum[i];
    population[i].numOfConflicts = h_conflictNum[i];
}
// zwalniamy pamięć
delete[] h_colors;
delete[] h_colorNum;
delete[] h_conflictNum;
// zwalniamy pamięć GPU
cudaFree(d_colors);
cudaFree(d_colorNum);
cudaFree(d_conflictNum);
}

```

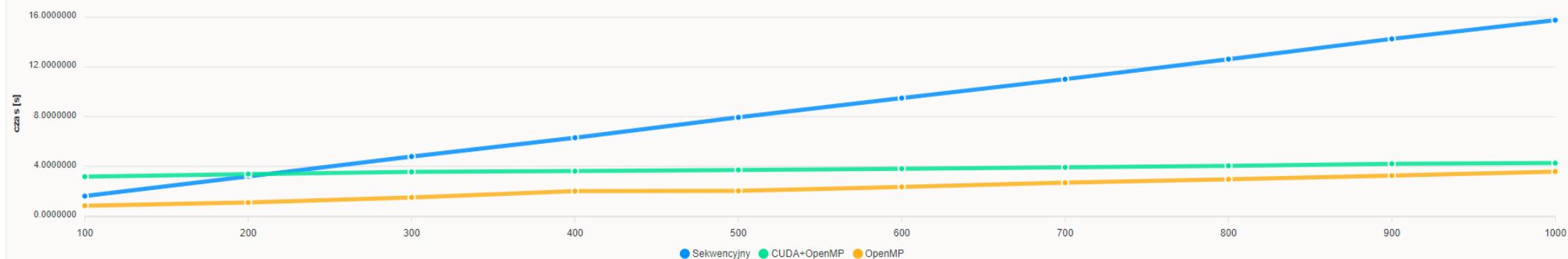
Testy wydajnościowe

Do testów wykorzystano gotową instancję z <https://mat.tepper.cmu.edu/COLOR/instances/> oraz wygenerowane grafy spójne o wypełnieniu krawędziami 50%.

Test 1 - rozmiar populacji

Instancja	Ilość wierzchołków	Liczba iteracji	rozmiar populacji
queen13.txt	169	1000	od 100 do 1000

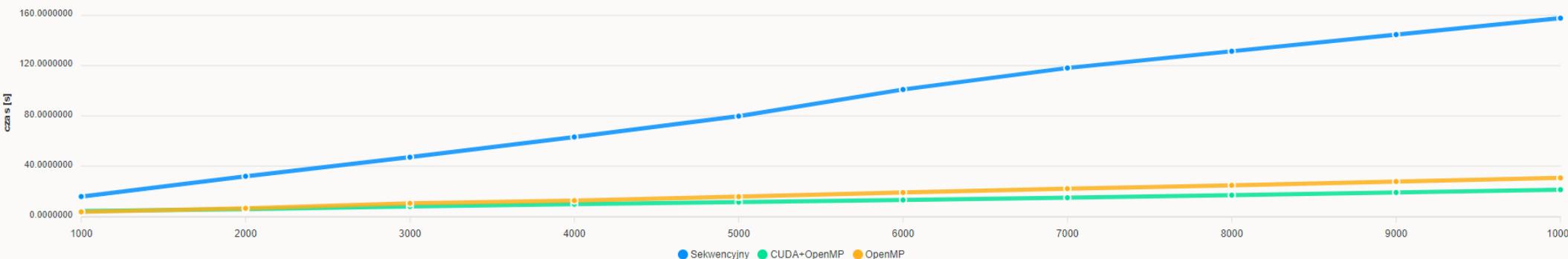
Czas wykonania algorytmu (n - rozmiar populacji)



Test 2 - rozmiar populacji

Instancja	Ilość wierzchołków	Liczba iteracji	rozmiar populacji
queen13.txt	169	1000	od 1000 do 10000

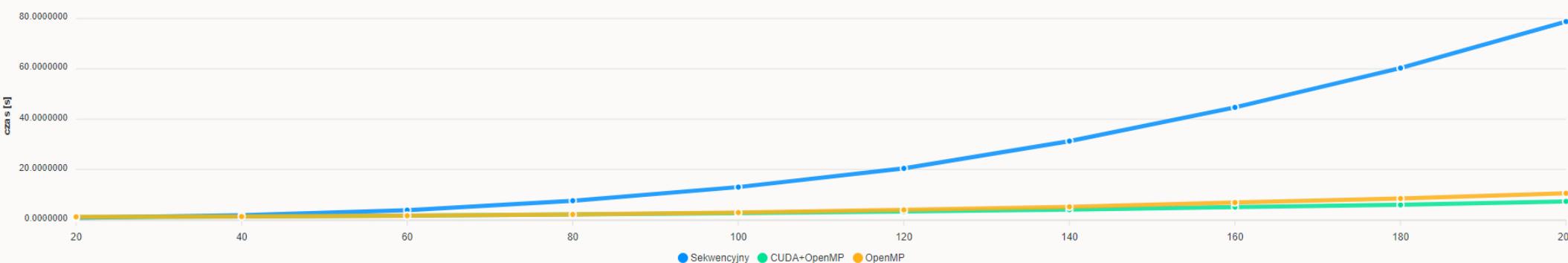
Czas wykonania algorytmu (n - rozmiar populacji)



Test 3 - ilość wierzchołków

Instancje	Rozmiar populacji	Liczba iteracji	Ilość wierzchołków
generowane grafy o wypełnieniu krawędziami = 50%	1300	1000	od 20 do 200

Czas wykonania algorytmu (n - ilość wierzchołków w grafie)



Wnioski

Dzięki zrównolegleniu algorytmu oraz wykorzystaniu karty graficznej do obliczeń można znacząco przyspieszyć jego działanie. Wraz ze zwiększeniem ilości przetwarzanych danych (rozmiaru instancji i rozmiaru populacji) zwiększa się przewaga CUDA nad OpenMP oraz wielokrotnie bardziej przewaga algorytmów równoległych nad algorytmem sekwencyjnym.

Dane sprzętowe

- System: Windows 10
- Procesor: AMD Ryzen 1600 3.2 GHz
- Karta graficzna: Nvidia GeForce GTX 1080
- Pamięć RAM: 16GB DDR4 3200 Mhz

Repozytorium z kodem źródłowym

[Github](#)

Źródła

1. https://pl.wikipedia.org/wiki/Liczba_chromatyczna
2. https://pl.wikipedia.org/wiki/Algorytm_genetyczny
3. <https://mat.tepper.cmu.edu/COLOR/instances.html>
4. [kolorowanie_grafow](#)