Równoległe przetwarzanie grafu

Katarzyna Dziewulska, Kamila Lis

W ramach projektu zostanie zaimplementowany algorytm kolorowania grafu w wersji zwykłej i równoległej. Sugerując się artykułem [1] postanowiono wykorzystać algorytm LF (Largest-First). Wersja równoległa zostanie zrealizowana w architekturze CUDA przy wykorzystaniu karty graficznej Nvidia.

1 Algorytm kolorowania

Kolorowanie wierzchołków polega na przydzielaniu kolorów wierzchołkom tak, żeby dwa sąsiednie wierzchołki otrzymały różne kolory. Najmniejsza liczba kolorów k nazywana jest liczbą chromatyczną. W celu reprezentacji pokolorowania przyjęło się oznaczać kolory kolejnymi liczbami naturalnymi. Kolorowanie grafów najczęściej wykorzystywane jest do poszukiwania rozwiązań, w których unika się konfliktów. Strategie kolorowania grafu zależą od stawianych wymagań. Celem optymalnego kolorowania jest minimalizacja liczby użytych kolorów, podczas gdy kolorowanie zbalansowane dąży do zapewnienia podobnej liczby wierzchołków w każdym kolorze.

1.1 Sekwencyjny Largest First

Algorytm LF (ang. *Largest-First*) koloruje wierzchołki w kolejności zgodnej z ich stopniami – wierzchołki są przeglądane i kolorowane zachłannie, według nierosnących stopni wierzchołkowych. Algorytm ma na celu minimalizowanie maksymalnej liczby wykorzystanych kolorów. Poniżej zamieszczono pseudokod obrazujący kroki algorytmu:

Algorithm 1: Sekwencyjny Largest-First

```
for dla każdego wierzchołka v ∈ V do
    Policz stopień d(v).
end
Posortuj zbiór V nierosnąco według stopni d wierzchołków.
for dla każdego wierzchołka v ∈ V do
    S' = S
    Aktualizuj S' - wykreśl kolory sąsiadów wierzchołka v.
    Pokoloruj v na najmniejszy kolor z S'.
end
```

V - zbiór wierzchołków grafu
n - liczba wszystkich wierzchołków grafu

$v \in 1, 2...n$ - pojedyńczy wierzchołek
S - zbiór wszystkich kolorów, liczba kolorów równa jest liczbie wierzchołków
$d(v)$ - stopień wierzchołka v

1.2 Metoda równoległa

Równoległe metody kolorowania grafów są oparte na obserwacji, że każdy niezależny zbiór wierzchołków, czyli taki w którym nie ma żadnych wierzchołków sąsiadujących ze sobą, może być kolorowany równolegle. Różnice pomiędzy metodami sprowadzają się do sposobu wyboru niezależnego zbioru i samego kolorowania wierzchołków. Niezależny zbiór wierzchołków jest konstruowany jako podgraf indukowany zawierający jedynie niepokolorowane wierzchołki. Najogólniej procedurę równoległego kolorowania można przedstawić następująco:

Algorithm 2: Kolorowanie równoległe

while $|G| > 0$ **do**
    z grafu G wybierz zbiór niezależnych wierzchołków U;
    koloruj wszystkie wierzchołki z U;
    $G := G - U$;
end

Algorytm LF w wersji zrównoleglonej polega na "równoległym" szukaniu niezależnego zbioru wierzchołków (niekoniecznie jest on maksymalny), wybierając te wierzchołki, które lokalnie są największego stopnia, a następnie kolorowaniu ich. Najpierw przydzielamy każdemu wierzchołkowi losową liczbę oraz obliczamy jego stopień. Następie równolegle sprawdzany jest stopień każdego wierzchołka (każdy wierzchołek rozpatrywany jest w oddzielnym wątku). Jeśli stopień ten jest większy od stopni sąsiadujących wierzchołków, to wierzchołek ten jest włączany do aktualnie tworzonego zbioru niezależnego. W przypadku konfliktu stopni decyduje większa wartość przydzielonej na początku liczby losowej. Po utworzeniu zbioru niezależnego wierzchołki do niego należące są równolegle kolorowane na najmniejszy kolor, który nie został użyty do pokolorowania ich sąsiadów. W kolejnych iteracjach zbiory niezależne wierzchołków są tworzone z wyłączeniem wierzchołków już pokolorowanych.

V - zbiór wierzchołków grafu
I - aktualny w danej iteracji niezależny zbiór wierzchołków grafu
U - aktualny w danej iteracji zbiór pozostałych do pokolorowania wierzchołków grafu
$d(v)$ - stopień wierzchołka v
n - liczba wszystkich wierzchołków grafu
$v \in 1, 2...n$ - pojedyńczy wierzchołek
S - zbiór wszystkich kolorów, liczba kolorów równa jest liczbie wierzchołków
x - liczba losowa

Algorithm 3: Równoległy Largest-First

for *dla każdego wierzchołka* $v \in V$ **do**
  Policz stopień $d(v)$.
  Przypisz losową liczbę x.
end
$U = V$
while $|U| > 0$ **do**
  for *dla każdego wierzchołka* $v \in U$ *wykonuj równolegle* **do**
    I = $\{v$, takie że $d(v) > d(u)$ dla każdego sąsiada u wierzchołka v, $u \in U\}$
    for *dla każdego wierzchołka* $v' \in I$ *wykonuj równolegle* **do**
      $S' = S$
      Aktualizuj S' - wykreśl kolory sąsiadów wierzchołka v'.
      Pokoloruj v' na najmniejszy kolor z S'.
    end
  end
  $U = U - I$
end

2 Struktury danych

Pierwszym krokiem do rozwiązania problemu kolorowania grafu jest znalezienie odpowiedniej struktury danych najlepiej opisującej strukturę grafu. W artykule [2] autorzy sugerują wykorzystanie macierzy sąsiedztwa jako odpowiedniej reprezentacji dla skomplikowanych algorytmów. Natomiast w [3] zauważono, że macierz sąsiedztwa marnuje dużo pamięci w przypadku „rzadkiego" grafu, dlatego lista sąsiedztwa byłaby lepszym sposobem reprezentacji takiego grafu. W GPU CUDA uzyskuje dostęp do pamięci w tablicy, więc z powodu różnej wielkości listy krawędzi trudnej jest korzystać z listy sąsiedztwa. Możliwym rozwiązaniem jest wykorzystanie biblioteki nvGRAPH [4] napisanej z myślą o algorytmach grafowych. Struktura grafowa jest zależna od wybranej topologii. Przykładowo, dla formatu CSR (*compressed sparse row*):

```
struct nvgraphCSRTopology32I_st {
  int nvertices;
  int nedges;
  int *source_offsets;
  int *destination_indices;
};
typedef struct nvgraphCSRTopology32I_st *nvgraphCSRTopology32I_t;
```

Listing 1: Struktura formatu CSR z biblioteki nvGRAPH.

gdzie

- `nvertices` – liczba wierzchołków grafu
- `nedges` – liczba krawędzi grafu
- `source_offsets` – tablica o rozmiarze $nvertices + 1$, gdzie i-ty element to numer indeksu pierwszej z krawędzi wychodzących z i-tego wierzchołka w tablicy krawędzi *destination_indices*; ostatni element przechowuje liczbę wszystkich krawędzi

- `destination_indices` – tablica o rozmiarze *nedges*, gdzie każda wartość to numer wierzchołka, do którego dochodzi i-ta krawędź

Listy wierzchołków i krawędzi pozwalają nam na określenie, które wierzchołki sąsiadują ze sobą. Dla implementacji algorytmu LF każdy z wierzchołków powinien być dodatkowo opisany przez trzy parametry:

- swój stopień $deg(v)$,
- losową wartość,
- kolor.

Dla grafów nieskierowanych stopniem wierzchołka będzie liczba wszystkich incydentnych krawędzi, a tym samym różnica między i i $i+1$ wartością w tablicy `source_offsets`. Wartość losowa może zostać wylosowana i przydzielona raz, na początku działania programu. Kolor może zostać opisany całkowitą liczbą naturalną (zakładając, że -1 oznacza brak przydzielonego koloru, rozpoczynamy od „koloru" 0, a następne określamy przez inkrementację). Parametry te można utożsamić z wierzchołkami przy wykorzystaniu funkcji `nvgraphAllocateVertexData` oraz `nvgraphSetVertexData`.

3 Projekty testów

Poprawność obu implementacji algorytmu będzie sprawdzana przez dodatkowo zaimplementowaną funkcję, uruchamianą dla wszystkich grafów, na których wykonywane będzie kolorowanie. Zarówno dla wyniku działania implementacji sekwencyjnej jak i równoległej algorytmu, funkcja sprawdzi każdy wierzchołek grafu oraz jego sąsiadów. Jeżeli znalezione zostaną jakiekolwiek dwa sąsiadujące ze sobą wierzchołki, które zostały pokolorowane na ten sam kolor oznaczać to będzie błędną implementację oraz konieczność jej poprawienia. Działanie algorytmu sekwencyjnego i równoległego przetestowane zostanie na grafach dostępnych w internecie między innymi na stronie `http://mat.gsia.cmu.edu/COLOR/instances.html`. Testowanie przeprowadzone zostanie dla grafów o liczbie wierzchołków rzędu od kilkudziesiąt do tysiąca, oraz liczbie krawędzi rzędu od kilkudziesiąt do kilkuset tysięcy (jeśli moc obliczeniowa sprzętu podoła takiej ich liczbie). Implementacje porównywane będą ze względu na czas znalezienia rozwiązania oraz liczbę użytych w rozwiązaniu kolorów. Dodatkowo implementacja równoległa sprawdzona zostanie na dwóch kartach graficznych: GForce 920M oraz GForce GT 525M.

4 Etap 3 – Implementacja i badania

4.1 Struktury danych - aktualizacja

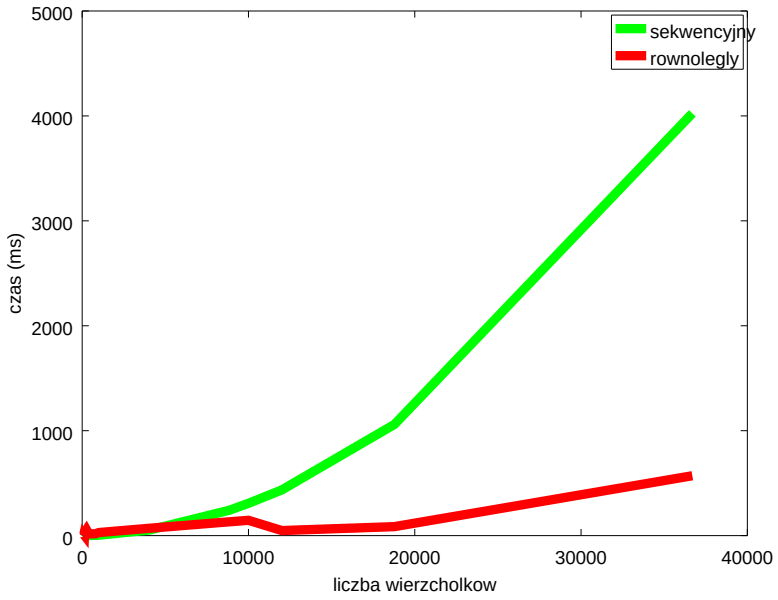
Podczas implementacji algorytmu zdecydowano się zrezygnować z biblioteki nvGRAPH do przechowywania grafu, ponieważ funkcje tej biblioteki okazały się nieprzydatne w implementacji. Zachowano jednak format CSR przechowywania grafu, dla którego powstała struktura analogiczna do tej z biblioteki nvGRAPH zamieszczonej na listingu 1. Struktura formatu CSR jest to skrócony zapis macierzy sąsiedztwa. Zawiera ona dwie zmienne przechowujące liczbę wierzchołków i liczbę krawędzi, oraz dwie tablice jednowymiarowe. Tablica *destination_indices* przechowuje zapisane w ciągu numery sąsiadów każdego z wierzchołków, rozpoczynając od najmniejszego wierzchołka. Z kolei tablica *source_offsets* zawiera indeksy tablicy *destination_indices* od których rozpoczynają się

sąsiedzi kolejnych wierzchołków. Indeksy tablicy *source_offsets* to numery wierzchołków. Tak więc dla wierzchołka np. nr 0 wartość $source\_offsets[0]$ będzie indeksem dla tablicy *destination_indices*, od którego zaczynają się numery jego sąsiadów.

4.2 Wyniki testów

Obie implementacje algorytmu były testowane na grafach o różnej liczbie wierzchołków i krawędzi. Dla każdego grafu zostało wykonane dziesięć prób kolorowania algorytmem w wersji sekwencyjnej i dziesięć prób algorytmem w wersji równoległej. Następnie została policzona średnia czasów wykonywania algorytmów oraz liczby przydzielonych kolorów. Testy algorytmów zostały wykonane na CPU (parametry). W wersji równoległej wykorzystano dodatkowo kartę graficzną GForce 920M, zdolność obliczeniowa 3.5, 2 multi-procesory. Docelowo testy miały być przeprowadzone także dla karty graficznej GForce GT 525M, jednak nie udało się uruchomić na niej potrzebnych narzędzi. Oba algorytmy porównywane były pod względem liczby wykorzystanych kolorów oraz czasu wykonywania. Wyniki testów zamieszczone zostały w tabeli poniżej. Niektóre znalezione do testów grafy zawierały informacje o optymalnym kolorowaniu, które również zamieszczono w tabeli. Poprawność kolorowania sprawdzona została specjalnie napisaną funkcją `CheckIfCorrect`. Wyniki funkcji zostały przedstawione w kolumnie *czy poprawnie?* w tabeli poniżej. Ilustrację do danych przedstawionych w tabeli stanowi rysunek 1.

wierzchołki	**krawędzie**	**LF - sekwencyjny**			**LF - równoległy**			optymalne kolorowanie
		czy poprawnie?	liczba kolorów	czas kolorowania (ms)	czy poprawnie?	liczba kolorów	czas kolorowania (ms)	
25	320	tak	7	0.136	tak	17	1.410	5
96	1368	tak	16	0.366	tak	46	7.400	12
100	2940	tak	17	0.373	tak	60	10.238	?
138	986	tak	11	0.482	tak	19	2.368	11
144	5192	tak	20	0.617	tak	84	23.456	?
169	6656	tak	20	0.870	tak	97	36.139	13
256	12640	tak	26	0.784	tak	144	57.164	?
400	8000	tak	15	0.871	tak	69	16.082	?
800	16000	tak	16	2.745	tak	66	17.639	?
1000	20000	tak	15	3.979	tak	81	29.306	?
4039	88234	tak	16	53.042	tak	136	70.0189	?
8846	185766	tak	17	241.605	tak	285	131.652	?
10000	200000	tak	16	307.984	tak	305	145.841	?
12008	118521	tak	10	435.568	tak	146	48.251	?
18772	198110	tak	11	1059.12	tak	218	85.491	?
36692	733840	tak	16	4024.31	tak	813	570.589	?



Rysunek 1: Czas kolorowania w zależności od liczby wierzchołków grafu dla algorytmu sekwencyjnego (zielony) i równoległego (czerwony).

4.3 Wnioski

Wszystkie testowane grafy zostały poprawnie pokolorowane dla obu wersji algorytmu. Zauważyć można, że algorytm w wersji sekwencyjnej wykorzystuje mniej kolorów niż algorytm w wersji równoległej, jednak nie zawsze znalezione rozwiązanie jest optymalne. Wynik taki był spodziewany, ponieważ algorytm sekwencyjny wybierając kolor dla wierzchołka bierze pod uwagę cały graf, natomiast algorytm równoległy wykonuje to samo jednak w każdej iteracji przetwarzając jedynie niezależny zbiór wierzchołków grafu, przez co wykorzystywane jest więcej kolorów. Jako że algorytm w wersji równoległej koloruje jednocześnie kilka wierzchołków, spodziewać by się mogło, że będzie on działał szybciej niż ten w wersji sekwencyjnej. Dla testowanych grafów algorytm równoległy znalazł wynik w krótszym czasie niż sekwencyjny dla grafów o liczbie wierzchołków powyżej 8000 – wykorzystanie algorytmu równoległego staję się opłacalne, gdy oszczędność czasu wynikająca z równoległego przetwarzania jest większa niż opóźnienia spowodowane dłuższym czasem dostępu do pamięci urządzenia oraz koniecznością kopiowania danych na i z pamięci GPU. Na większych grafach, czas kolorowania algorytmu równoległego jest mniejszy, ponieważ wykorzystana jest tak duża liczba wątków (każdy wierzchołek w oddzielnym wątku), że powoduje to sumarycznie szybsze wykonanie algorytmu kolorowania. Z powyższej tabeli wynika, że algorytm równoległy działa znacznie szybciej dla grafów rzadkich (o niewielkiej liczbie krawędzi). Wniosek z powyższych rozważań jest taki, że jeżeli zależy nam na jak najmniejszej liczbie kolorów wykorzystywanych do kolorowania to należy wybrać wersję sekwencyjną. Jeśli chodzi o jak najszybsze pokolorowanie wtedy w zależności od liczby wierzchołków wybieramy tą wersje algorytmu, która działa lepiej.

4.4 Perspektywy przyśpieszenia algorytmu równoległego

Jedną z możliwości dodatkowego przyśpieszenia działania algorytmu równoległego jest zastąpienie równoczesnego uruchamiania funkcji kolorującej w oddzielnym wątku, dla każdego wierzchołka na kilkukrotne wykonanie funkcji dla wielu wierzchołków z wykorzystaniem tych samych wątków. Taka zmiana powinna pozwolić na zmniejszenie czasu działania, ponieważ koszt uruchomienia nowego wątku jest stosunkowo duży. W ramach projektu nie wykonano testów dla tego podejścia dla zachowania czytelności kodu.

Literatura

[1] J R. Allwright, Rajesh Bordawekar, P D. Coddington, Kıvanç Dinçer, C L. Martin. A comparison of parallel graph coloring algorithms. 03 1995.

[2] Fengxian Shen, Xu Jian, Xianjie Xi. Implementation of graphic vertex-coloring parallel synthesis algorithm based on genetic algorithm and compute unified device architecture. *Automatic Control and Computer Sciences*, 51(1):32–41, Jan 2017.

[3] Avadhesh Pratap Singh, Dhirendra Pratap Singh. Implementation of k-shortest path algorithm in gpu using cuda. *Procedia Computer Science*, 48:5 – 13, 2015. International Conference on Computer, Communication and Convergence (ICCC 2015).

[4] Oficjalna dokumentacja bilioteki nvGRAPH. `https://docs.nvidia.com/cuda/nvgraph`.