Mateusz Comblik 175827

Współczesne Środowiska Programowania

Projekt – Algorytm A*

1. Celem ćwiczenia było zapoznanie się z programowaniem kart graficznych firmy Nvidia i napisanie programu obliczającego najkrótszą drogę na grafie pomiędzy dwoma punktami odpowiednio na CPU oraz na GPU.

2. Testy zostały wykonane na karcie graficznej GTX 1050 Ti, program został napisany w języku python, do wykorzystania środowiska CUDA posłużyła biblioteka numba.

3. Biblioteka numba tłumaczy funkcje Pythona na kod PTX, który wykonuje się na sprzęcie CUDA. Dekorator jit jest stosowany do funkcji Pythona napisanych w naszym kodzie Pythona dla CUDA. Numba współdziała z CUDA Driver API w celu załadowania PTX na urządzenie CUDA i wykonania. Semantyka wykorzystywana przez bibliotekę numba przypomina tą na C++, jesteśmy w stanie pisać kernele oraz przekazywać informacje z hosta na device i na odwrót. Nie musimy natomiast alokować pamięci poszczególnych zmiennych, ponieważ odbywa się to automatycznie (pamięć na GPU jest alokowana tak długo jak jest potrzebna).

3. Operacje wykonywane przez program:

(CPU)

- generowanie macierzy przechowujących wartości reprezentujących drogę do przebycia przez algorytm

- wyznaczanie drogi od startu do końca algorytmem A*

- animacja końcowego wyniku za pomocą biblioteki matplotlib i mierzenie czasu pracy algorytmu

(GPU)

- wyznaczanie drogi od startu do końca zrównoleglonym algorytmem A* (więcej informacji w następnych punktach)

- mierzenie czasu pracy algorytmu

4. Opis algorytmu A*:

Algorytm A* jest algorytmem heurystycznym do znajdowania najkrótszej ścieżki w grafie ważonym z dowolnego wierzchołka do wierzchołka spełniającego określony warunek.

Algorytm jest zupełny i optymalny, w tym sensie, że znajduje ścieżkę, jeśli tylko taka istnieje, i przy tym jest to ścieżka najkrótsza. Stosowany głównie w dziedzinie sztucznej inteligencji do rozwiązywania problemów i w grach komputerowych do imitowania inteligentnego zachowania.

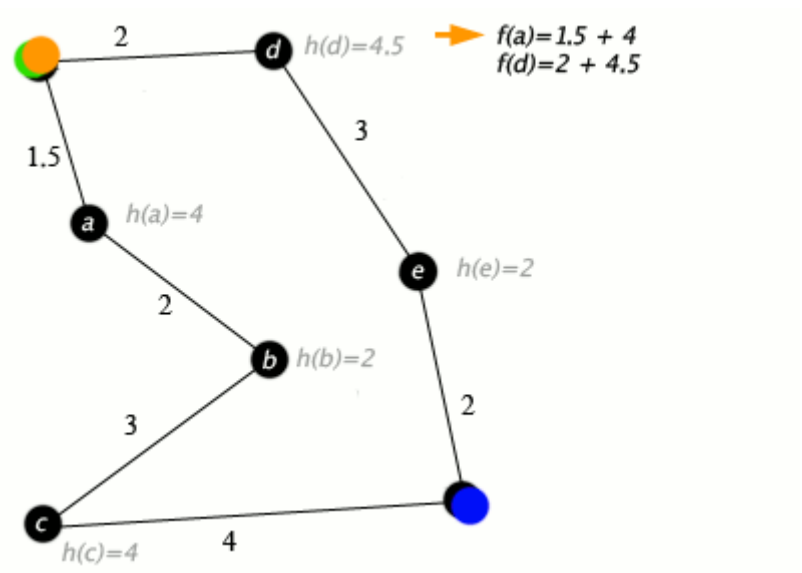
Algorytm A* od wierzchołka początkowego tworzy ścieżkę, za każdym razem wybierając wierzchołek x z dostępnych w danym kroku wierzchołków tak, by minimalizować funkcję f(x) postaci:

f(x)=g(x)+h(x),

gdzie:

g(x) – droga pomiędzy wierzchołkiem początkowym a x

h(x) – przewidywana droga pomiędzy aktualnym wierzchołkiem a celem, często określana za pomocą funkcji



Przykład działania algorytmu A, źródło: wikipedia*

Pseudokod algorytmu A*:

```
function A*(start,goal)
    closedset := the empty set                 % Zbiór wierzchołków przejrzanych.
    openset := set containing the initial node % Zbiór wierzchołków nieodwiedzonych,
sąsiadujących z odwiedzonymi.
    g_score[start] := 0                        % Długość optymalnej trasy.
    while openset is not empty
        x := the node in openset having the lowest f_score[] value
        if x = goal
            return reconstruct_path(came_from,goal)
        remove x from openset
        add x to closedset
        foreach y in neighbor_nodes(x)
            if y in closedset
                continue
            tentative_g_score := g_score[x] + dist_between(x,y)
            tentative_is_better := false
            if y not in openset
                add y to openset
                h_score[y] := heuristic_estimate_of_distance_to_goal_from(y)
                tentative_is_better := true
            elseif tentative_g_score < g_score[y]
                tentative_is_better := true
            if tentative_is_better = true
                came_from[y] := x
                g_score[y] := tentative_g_score
                f_score[y] := g_score[y] + h_score[y] % Przewidywany dystans od startu do celu
przez y.
    return failure

 function reconstruct_path(came_from,current_node)
    if came_from[current_node] is set
        p = reconstruct_path(came_from,came_from[current_node])
        return (p + current_node)
    else
        return the empty path
```

5. Implementacja algorytmu A* na CPU:

Algorytm A* zazwyczaj jest prezentowany za pomocą grafu z wierzchołkami połączonymi krawędziami z odpowiednimi wagami. W zaimplementowanym rozwiązaniu postanowiłem użyć reprezentacji za pomocą siatki punktów w postaci kwadratów, taka forma znacząco ułatwia wizualizację oraz implementację – wartości konkretnych punktów mogą być reprezentowane za pomocą macierzy dwuwymiarowej z odpowiednimi wartościami.

W celu przejrzystości zdecydowałem się na uproszczenie wag, każdy punkt posiada domyślnie wagę równą jeden. Dzięki temu podczas wizualizacji jesteśmy od razu w stanie stwierdzić czy algorytm działa poprawnie (w wypadku zastosowania losowych wag wizualizacja byłaby mniej oczywista).

Reprezentacja za pomocą kwadratowej siatki punktów oznacza również wymuszenie znalezienia odpowiedniego algorytmu do obliczania odległości pomiędzy nimi. W tym celu został wybrany algorytm *Manhattan Distance*:

h = abs(x_aktualnego_punktu – x_punktu_celu) + abs(y_aktualnego_punktu – y_punktu_celu)

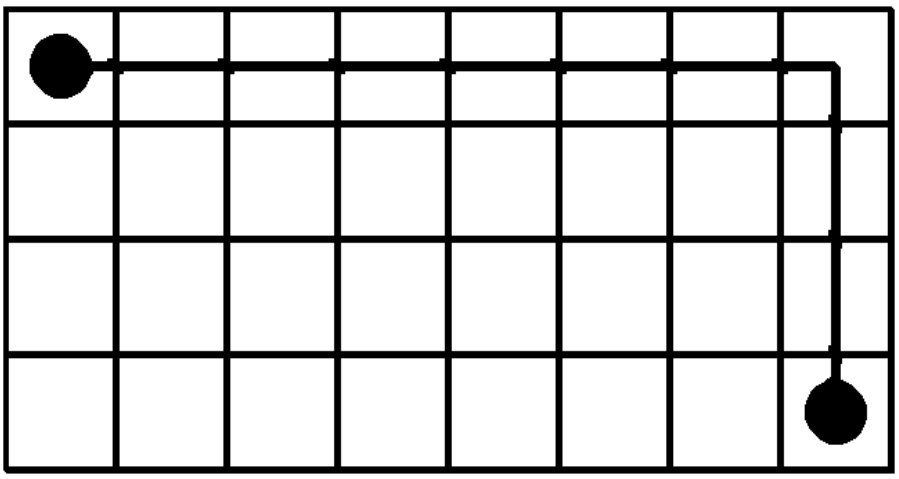
Jest to nic innego jak suma wartości bezwzględnych różnic współrzędnych x i y celu oraz współrzędnych x i y bieżącego punktu.

Kiedy używać tej heurystyki? - Gdy wolno nam poruszać się tylko w czterech kierunkach (prawo, lewo, góra, dół); dlatego również został wybrany w tym wypadku.

Napisany kod:

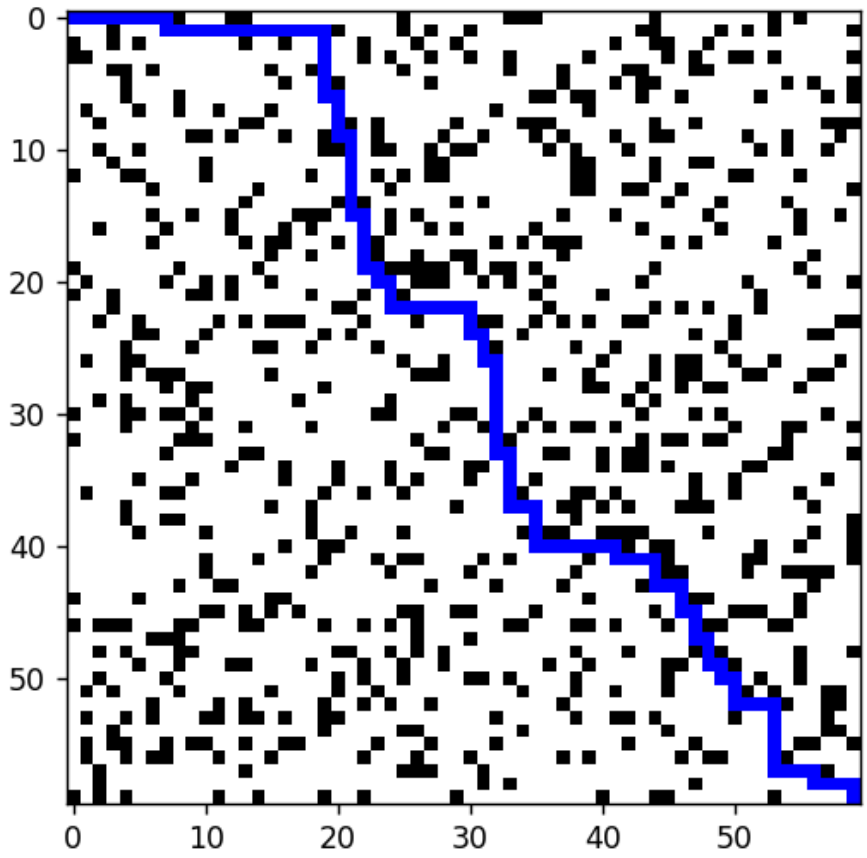
```
def h(komorka1,komorka2):
    x1,y1=komorka1
    x2,y2=komorka2
    return abs(x1-x2) + abs(y1-y2)
```

heurystyka, cpu



Wizualizacja Manhattan Distance (lewy punkt – start, prawy - cel)

Po implementacji kodu w języku python, każda ścieżka wyglądałaby podobnie do tej na rysunku powyżej. Dla urozmaicenia prezentacji, dodałem punkty, po których algorytm nie może się poruszać (ściany).



Obrazek przedstawia wykonaną wizualizację algorytmu A.*

Białe pola – teren, po którym algorytm może się poruszać

Czarne pola - ściany

Niebieski – gotowy algorytm; najkrótsza droga od lewego górnego rogu do dolnego

Na powyższej wizualizacji widać tendencję algorytmu do odnajdywania najkrótszej ścieżki od góry zamiast po diagonali. Dzieje się tak z powodu właśnie zastosowanego algorytmu *Manhattan Distance*.

Obliczenia na procesorze korzystają z biblioteki numpy w celu zminimalizowania czasu obliczeń; nie użyto zbędnych pętli ‘for’, które spowolniłyby czas działania algorytmu.

6. Wstęp teoretyczny do równoleglenia algorytmu A*:

W niektórych zastosowaniach wyszukiwania A*, wyznaczanie funkcji heurystycznych jest bardzo kosztowne i staje się bottleneckiem dla całego algorytmu. Pierwszym krokiem w procesie paralelizacji wyszukiwania A* na GPU w takich zastosowaniach jest paralelizacja obliczania funkcji heurystycznych. Ten krok jest prosty, a rozumowanie opiera się na obserwacji, że obliczanie funkcji heurystycznych dla każdego stanu rozszerzonego jest wzajemnie niezależne.

Przykładem dla powyższej sytuacji może być projektowanie białek (protein design); jest to istotny problem w biologii obliczeniowej, który można sformułować jako wyznaczanie najbardziej prawdopodobnego wyjaśnienia modelu grafowego z grafem zupełnym. Aby rozwiązać ten problem, stworzono przeszukiwanie drzew A*.

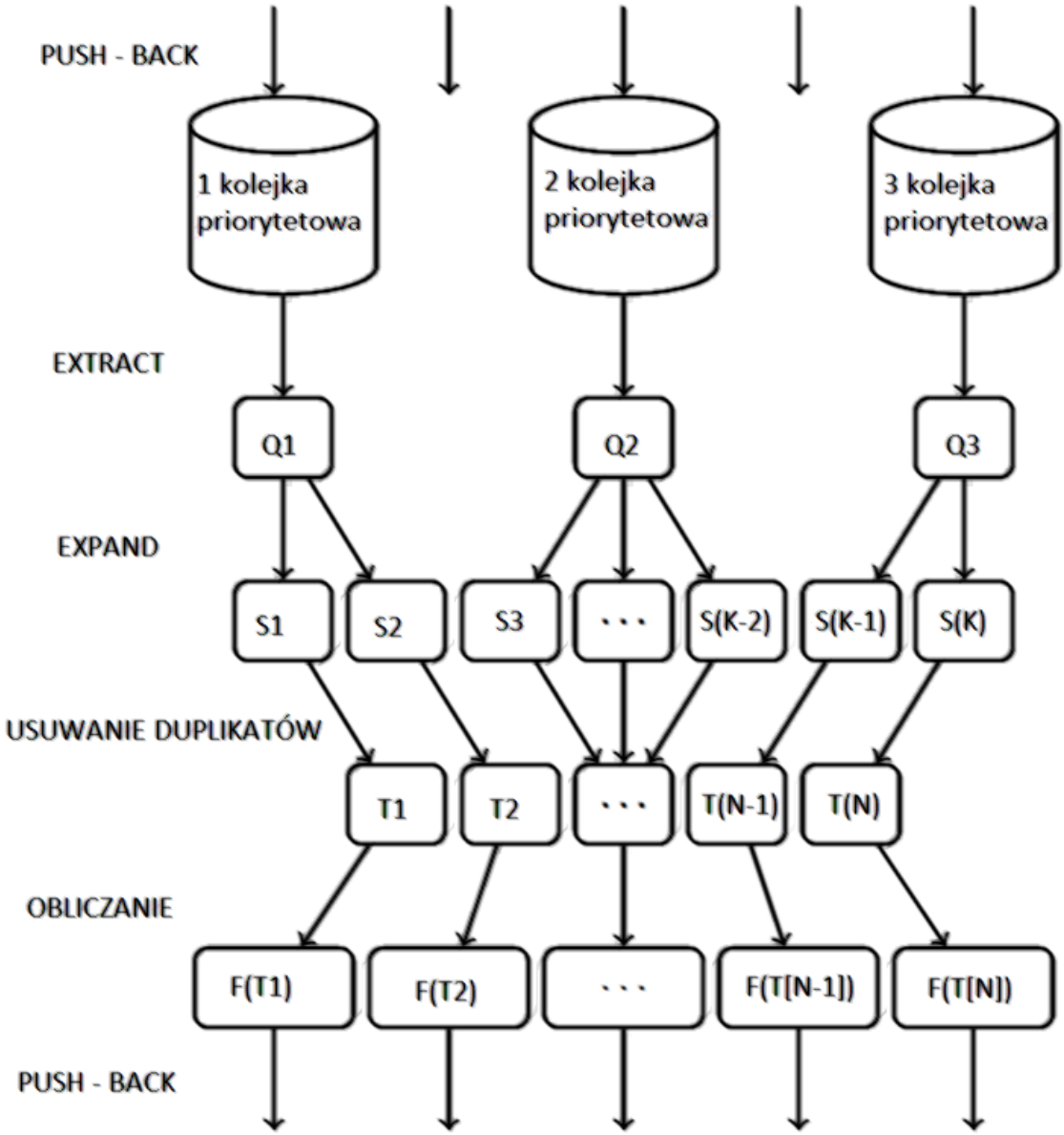
7. Równoległe kolejki priorytetowe:

Po zastosowaniu poprzedniej procedury, otrzymujemy prosty algorytm równoległy. Jednakże, na platformie obliczeniowej GPU, ten algorytm A* jest nadal nieefektywny. W tym miejscu pojawiają się dwa problemy. Po pierwsze, stopień równoległości jest ograniczony przez zewnętrzny stopień każdego węzła w grafie wyszukiwania. Procesor graficzny ma zazwyczaj tysiące rdzeni. Jednak w niektórych aplikacjach, takich jak wyznaczanie ścieżek w grafie siatki, stopień węzła jest zwykle mniejszy niż dziesięć, co ogranicza stopień równoległości na platformie GPU. Po drugie, wiele sekwencyjnych części tego prostego algorytmu nie zostało jeszcze w pełni sparalelizowanych.

Na przykład, używając sterty binarnej, operacje EXTRACT i PUSH-BACK dla listy otwartej zajmują O(log N) czasu, gdzie N jest całkowitą liczbą elementów na liście. Funkcje kolejki priorytetowej staną się najbardziej czasochłonnymi częściami wyszukiwania A* w zastosowaniach, w których obliczanie funkcji heurystycznych jest stosunkowo tanie. Operacje sekwencyjne są nieefektywne dla procesora graficznego, ponieważ wykorzystują tylko niewielką część jego możliwości, a wydajność pojedynczego wątku w procesorze graficznym jest znacznie niższa niż na procesorze.

Aby zwiększyć stopień równoległości w obliczaniu funkcji heurystycznych, pierwszy problem może zostać rozwiązany poprzez wyodrębnienie wielu stanów z kolejki priorytetowej. Z drugiej strony, operacje na kolejce priorytetowej nadal działają w trybie sekwencyjnym. Drugi problem może zostać rozwiązany poprzez zastosowanie współbieżnej struktury danych dla kolejki priorytetowej. Istniejące bezblokowe współbieżne kolejki priorytetów, nie mogą działać wydajnie w architekturze Single Instruction and Multiple Data Stream (SIMD) nowoczesnego procesora graficznego, ponieważ wymagają użycia operacji compare-and-swap (CAS).

Aby rozwiązać oba problemy, możemy stworzyć równoległy algorytm A*, który wykorzystuje równoległość sprzętu GPU. Zamiast używać pojedynczej kolejki priorytetowej dla listy otwartej (lista sprawdzanych wierzchołków/punktów), możemy podczas wyszukiwania A* przydzielić dużą liczbę (przykładowo tysiące) kolejek priorytetowych. Za każdym razem możemy wyodrębnić wiele stanów z poszczególnych kolejek priorytetowych, co powoduje paralelizację sekwencyjnej części oryginalnego algorytmu. Może to również zwiększyć liczbę rozszerzających się stanów w każdym kroku, poprawiając stopień równoległości dla obliczania funkcji heurystycznych, jak omówiono wcześniej.



Przepływ danych dla listy otwartej

8. Pseudokod dla równoległego algorytmu A*:

```
1 procedure ParallelAstar(s, t, k)
2 {Find the shortest path from s to t with k queues}
3       Let {Qi} where i=1 to k be the priority queues of the open list
4       Let H be the closed list
5       PUSH(Q1, s)
6       m <- nil {m stores the best target state}
7       while Q is not empty do
8               Let S be an empty list
9               for i 1 to k in parallel do
10                      if Qi is empty then
11                              continue
12                      end if
13                      qi <- EXTRACT(Qi)
14                      if qi.node = t then
15                              if m = nil or f(qi) < f(m) then
16                                      m <- qi
17                              end if
18                              continue
19                      end if
20                      S <- S + EXPAND(qi)
21              end for
22              if m =/= nil and f(m) ≤ minq BELONGS TO Q f(q) then
23                      return the path generated from m
24              end if
25              T <- S
26              for s' BELONGS TO S in parallel do
27                      if s'.node BELONGS TO H and H[s'.node].g < s'.g then
28                              remove s' from T
29                      end if
30              end for
31              for t' BELONGS TO T in parallel do
32                      t'.f <- f(t')
33                      Push t' to one of priority queues
34                      H[t'.node] <- t'
35              end for
36      end while
37 end procedure
```

Powyższy pseudokod nie jest w pełni oczywisty, dlatego wytłumaczenie znajduje się poniżej.

Dla każdego otwartego lub zamkniętego stanu listy s, s.node reprezentuje ostatni punkt w ścieżce zdefiniowanej przez s, s.f i s.g przechowują wartości odpowiednio f(s) i g(s), a s.prev przechowuje wskaźnik do poprzedniego stanu, który rozszerzył s, który jest używany do odbudowania ścieżki z określonego stanu. Lista S zawiera rozszerzone punkty (sąsiadów), zaś lista T zawiera punkty po usunięciu zduplikowanych punktów. Linie 25-30 wykrywają zduplikowane punkty, gdzie H[n] definiuje stan listy zamkniętej, w którym ostatnim punktem na jej ścieżce jest punkt n. Punkty pochodzące od tego samego rodzica możemy umieścić w różnych kolejkach za pomocą operacji synchronizacji, które są obliczeniowo tanie na układach GPU (linia 33), ponieważ węzły o tym samym rodzicu mają podobne wartości priorytetu.

Możemy zwiększyć stopień równoległości, przydzielając więcej kolejek priorytetowych (tj. zwiększając parametr k), aby jeszcze bardziej wykorzystać moc obliczeniową sprzętu GPU. Nie możemy jednak w nieskończoność zwiększać stopnia równoległości, ponieważ równoległe wyodrębnianie wielu stanów zamiast jednego stanu o najlepszej dotychczas wartości f może spowodować narzut na liczbę rozszerzanych stanów. Im wyższy stopień paralelizmu, tym więcej stanów musi wygenerować równoległy algorytm A*, aby znaleźć optymalną ścieżkę. Aby wykorzystać moc obliczeniową sprzętu GPU, musimy zrównoważyć kompromis pomiędzy narzutem dodatkowych stanów rozszerzonych a stopniem równoległości.

9. Wykrywanie duplikacji punktów na GPU:

W przeszukiwaniu grafu A* możemy próbować rozszerzyć stan, którego punkt został już odwiedzony. Jeśli wartość f nowego stanu jest nie mniejsza niż wartość f stanu istniejącego na zamkniętej liście, to można bezpiecznie zapobiec ponownemu odwiedzeniu tego stanu. Jest to znane jako wykrywanie duplikacji punktów. Trudność wykrywania duplikacji punktów zależy od aplikacji. Ten krok nie jest wymagany w przeszukiwaniu drzewa A*. Aby wykonać przeszukiwanie grafu A*, w którym cały graf mieści się w pamięci, możemy po prostu użyć tablicy 1D.

Wykrywanie duplikacji węzłów wymaga użycia struktury danych, która może obsługiwać zarówno operacje INSERT jak i QUERY. Polecenie INSERT dodaje parę klucz-wartość do struktury danych. QUERY określa, czy klucz istnieje w tej strukturze danych, czy nie. Jeśli tak jest, zwraca wartość z nim związaną. Często używamy połączonej tablicy haszującej lub zrównoważonego drzewa wyszukiwania binarnego na platformie CPU. Jednak rozszerzenie tych struktur danych do równoległego wykrywania duplikacji węzłów na GPU jest niezwykle trudne.

Musimy więc uciec się do bardziej odpowiedniej struktury danych dla efektywnego wykrywania duplikacji węzłów na GPU, takiej jak:

- Równoległy algorytm cuckoo hashing, który jest sparalelizowaną wersją tradycyjnego algorytmu cuckoo hashing

- Równoległe haszowanie z wymianą, które jest probabilistyczną strukturą danych zaprojektowaną specjalnie z myślą o prostocie na układach GPU

Równoległe haszowanie z wymianą. Kluczową obserwacją jest to, że nie musimy w nim gwarantować, że wszystkie zduplikowane punkty muszą zostać wykryte. Jest to dopuszczalne, ponieważ posiadanie na liście otwartych stanów zduplikowanymi punktami na otwartej liście nie wpływa na poprawność całego algorytmu. Jeśli pozwolimy sobie na pominięcie niektórych punktów, to proces wykrywania kolizji nie jest ściśle wymagany. W zasadzie, równoległe haszowanie z wymianą jest podobne do cuckoo hashing, z tą różnicą, że gdy nowy punkt ma zająć pozycję starego punktu, zamiast przesuwać stary punkt na na inną pozycję, równoległe haszowanie z wymianą po prostu usuwa stary punkt. Ta modyfikacja czyni go prostszym, szybsze i łatwiejsze do paralelizacji. Szczegółowy pseudokod równoległego haszowania z wymianą można znaleźć w Algorytmie A3 w dodatku (Zhou i Zeng 2014). Wybór pomiędzy równoległym cuckoo hashing a równoległym haszowaniem z wymianą zależy od indywidualnych zastosowań. Jeśli głównym problemem jest ograniczenie przestrzeni, można wybrać równoległe haszowanie z wymianą, dzięki czemu może ono wykryć wszystkie zduplikowane punkty. Z drugiej strony, równoległe haszowanie z wymianą może być wybrane, jeśli szybkość lub prostota jest głównym czynnikiem.

10. Implementacja równoległego algorytmu A*:

```
@cuda.jit
def agwiazdka_gpu(grid,dim,k,result):

    tx = cuda.threadIdx.x

    poprzedni_punkt=cuda.shared.array(shape=(36),dtype=int64)
    najnizszy_koszt=cuda.shared.array(shape=(36),dtype=int64)
    array=cuda.shared.array(shape=(30),dtype=int64)
    heuristics=cuda.shared.array(shape=(30),dtype=int64)
    sizes=cuda.shared.array(shape=(5),dtype=int64)
    flag=cuda.shared.array(shape=(1),dtype=int64)
    odkryte_pkt=cuda.shared.array(shape=(20),dtype=int64)
```

Powyższy przykład wykonano dla tablicy punktów 6x6 (natomiast pomiary zostały już wykonane dla większych, wymagających rozmiarów)

Tutaj odbywa się inicjalizacja zmiennych shared int; k jest liczbą kolejek (queue)

poprzedni_punkt służy do przetrzymywania znalezionej drogi

najnizszy_koszt przetrzymuje koszty przechodzenia konkretnych punktów, na początku ustalane na ‘inf’

array symuluje działanie kolejki priorytetowej

sizes przetrzymuje obecne rozmiary kolejek priorytetowych

odkryte_pkt to k * 4 z powodu czterech możliwości ruchu

```
    flag=0
    najnizszy_koszt[(dim*dim)-1]=0
    poprzedni_punkt[(dim*dim)-1]=-5

    if tx==0:
        for i in range(dim*k):
            array[i]=2
        for i in range(dim*dim):
            najnizszy_koszt[i]=213640
        for i in range(k):
            sizes[i]=0
        for i in range(4*k):
            odkryte_pkt[i]=2

    cuda.syncthreads()
```

w kolejnym wycinku kodu następują wstępne ustawienia (na przykład ustawienie najniższych kosztów wszystkich punktów na wysoką wartość jako symulacja ‘inf’)

```
    array[0]=0
    sizes[0]=1
    najnizszy_koszt[0]=0

    while(checkIfQueueEmpty(array,dim,k)!=0):
        cuda.syncthreads()
        if flag==1:
            break
```

na początku algorytmu, tak samo jak w jego sekwencyjnej implementacji, następuje sprawdzenie, czy kolejki są puste, czy posiadają jeszcze punkty do sprawdzenia

```
        if array[dim*tx]!=-1:
            extracted=array[dim*tx]
            shiftqueue(array,dim,tx)
            sizes[tx]-=1

            if extracted==((dim*dim)-1) or flag==1:
                flag=1
                continue
```

w tym momencie zaczyna się uzupełnianie listy odkryte_pkt, najpierw sprawdzamy czy aktualna kolejka prioryterowa jest pusta, następnie ustawiamy odkryty punkt na początek

```
            top=extracted-dim
            bottom=extracted+dim
            left=extracted-1
            right=extracted+1
            if (extracted%dim)==0:
                left=-1
            if extracted%dim==dim-1:
                right=-1
```

dodawanie do odkryte_pkt

```
            odkryte_pkt[tx*4+0]=top
            odkryte_pkt[tx*4+1]=bottom
            odkryte_pkt[tx*4+2]=left
            odkryte_pkt[tx*4+3]=right
```

wyrzucanie sąsiednich punktów do tablicy

```
            for i in range(4):
                aktualny_numer=odkryte_pkt[tx*4+i]

                if(aktualny_numer<0 or aktualny_numer>dim*dim or
grid[aktualny_numer]==0 or
najnizszy_koszt[extracted]+1>najnizszy_koszt[aktualny_numer]):
                    odkryte_pkt[tx*4+i]=-1
                    continue

                if(odkryte_pkt[tx*4+i]!=-1):
                    najnizszy_koszt[aktualny_numer]=najnizszy_koszt[extracted]
+1
                    poprzedni_punkt[aktualny_numer]=extracted
```

sprawdzanie sąsiednich punktów, usuwanie duplikatów, droga robi się krótsza, dlatego nastepuje zaktualizowanie kosztów oraz odkrytych punktów

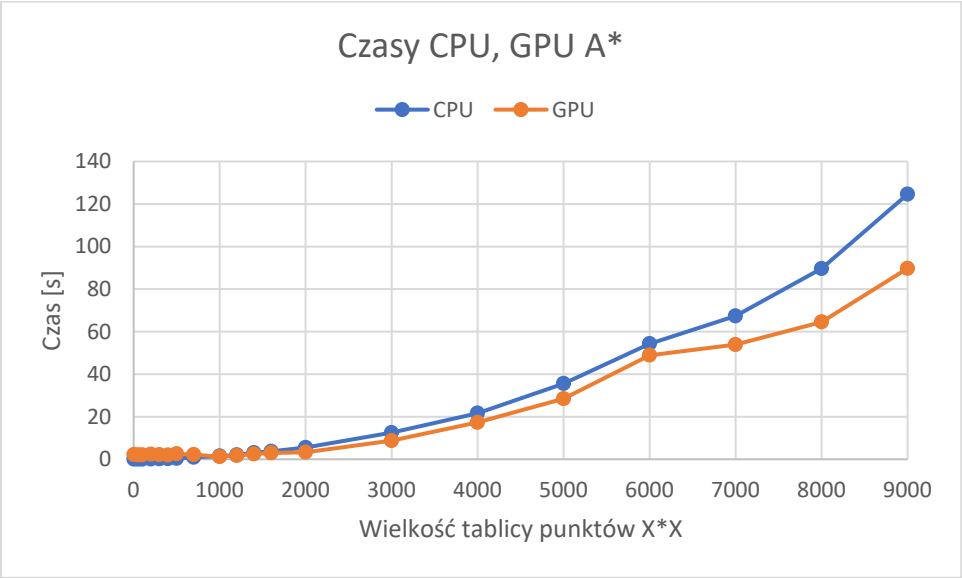
```
            for i in range(4):
                r=duplicateAdjacents(odkryte_pkt,odkryte_pkt[tx*4+i],k,tx*4+i)
                if r!=2:
                    cuda.atomic.exch(odkryte_pkt,r,2)
                if(odkryte_pkt[tx*4+i]!=2):
                    h=najnizszy_koszt[odkryte_pkt[tx*4+i]]+heuristic(dim,odkry
te_pkt[tx*4+i],(dim*dim)-1)
                    check=0
                    while(check==0):
                        targetLocation=findNextInsertionPoint(sizes,k)
                        if(cuda.atomic.compare_and_swap(array[targetLocation*d
im+sizes[targetLocation]],2,odkryte_pkt[tx*4+i])==2):
                            heuristics[targetLocation*dim+sizes[targetLocation
]]=h
                            sizes[targetLocation]+=1
                            organizeQueue(array,targetLocation,heuristics,h,si
zes[targetLocation],dim)
                            check=1
```

w powyższym wycinku odbywa się obliczanie manhattan distance/heurystyki dla punktów, które nie mają wartości 2, czyli wartości odkrytej drogi

koniec tego kawałku kodu oznacza, że w tym momencie algorytm powinien znaleźć już trasę

```
    if tx==0:
        aktualny=dim*dim-1
        while(aktualny!=0):
            if poprzedni_punkt[aktualny]==-5:
                break
            grid[aktualny]=2
            aktualny=poprzedni_punkt[aktualny]
            if aktualny==0:
                grid[aktualny]=2
            else:
                print('brak drogi')
```

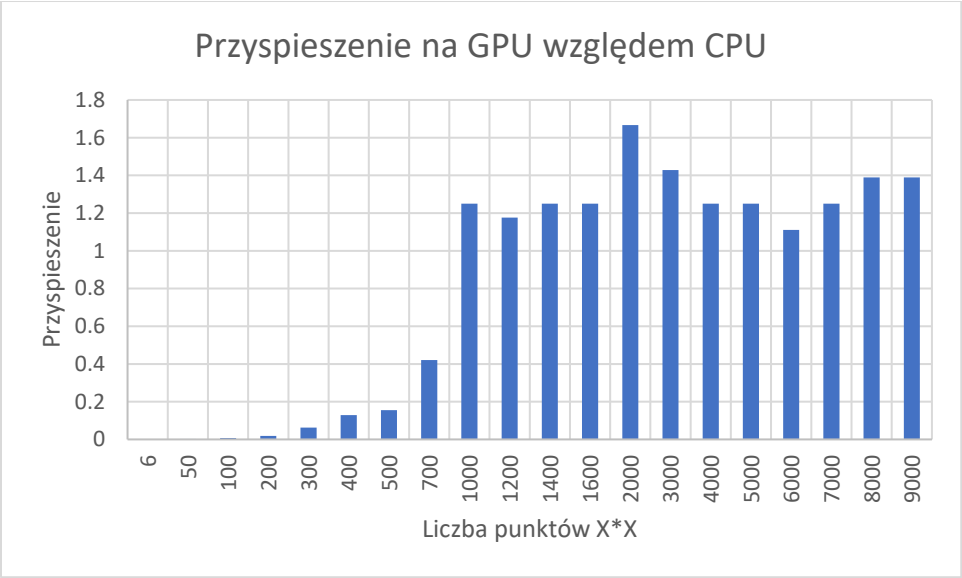
11. Osiągniete wyniki, czasy, porównania:



Osiągnięte wyniki potwierdzają poprawność implementacji jak i wszystkie wnioski z poprzednich laboratoriów – więcej danych oznacza lepszą optymalizację za pomocą programowania na karcie graficznej. Na powyższym wykresie konkretne przyspieszenie zaczynało się od około tysiąca punktów (każdy punkt posiada swoją wartość).

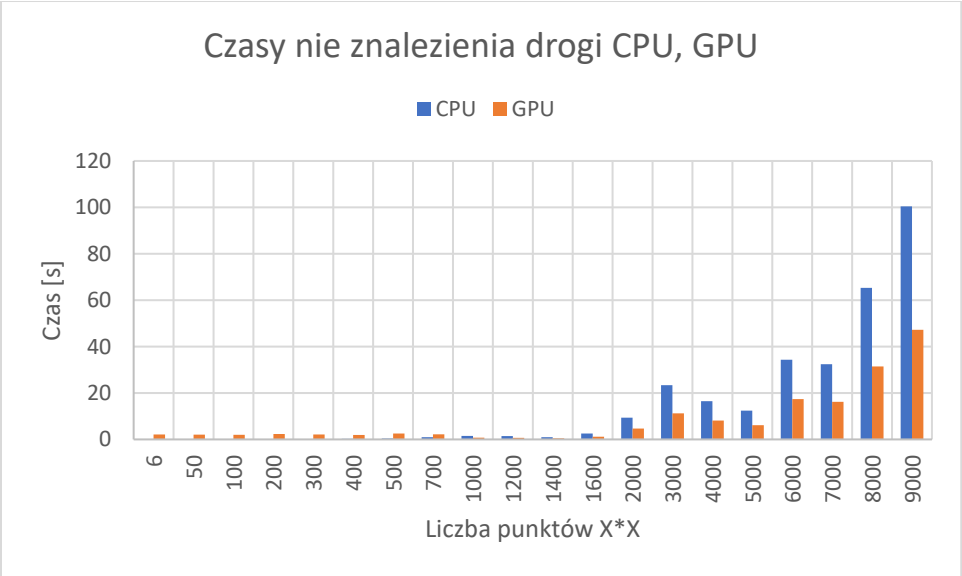
0.000763	6	2.154554
0.004269	50	2.054354
0.015149	100	2.00435
0.04403	200	2.335454
0.134009	300	2.134535
0.252675	400	1.954346
0.393787	500	2.532535
0.935694	700	2.224344
1.50898	1000	1.207184
2.006877	1200	1.705845
3.091629	1400	2.473304
3.723392	1600	2.978713
5.524304	2000	3.314582
12.49895	3000	8.749263
21.66973	4000	17.33579
35.56517	5000	28.45213
54.32147	6000	48.88932
67.35601	7000	53.88481
89.64767	8000	64.54632
124.6259	9000	89.73066

Wyniki w postaci tabeli.

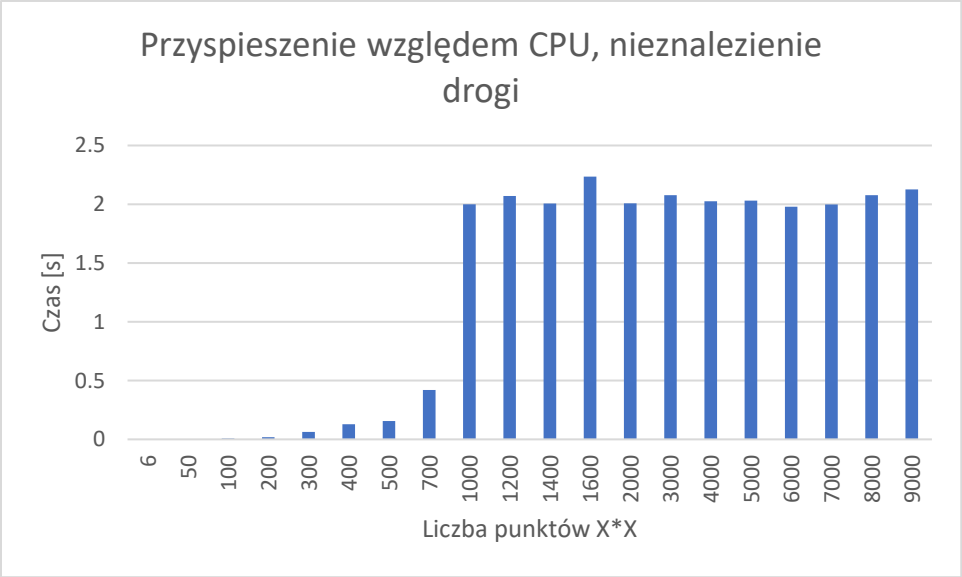


W tym wypadku widzimy wyraźne przyspieszenie od 1000 punktów, warto też zauważyć, że przyspieszenie nie rośnie niesamowicie szybko w porównaniu do liczby punktów; utrzymuje się natomiast na dość konkretnej wartości. Przyspieszenie wynosi mniej więcej 20/30%, ma to sens, ponieważ algorytm na karcie graficznej musi rozwiązywać również wiele innych pomniejszych problemów (tak jak opisano to w poprzednich punktach) co jest wciąż zajmującym procesem. Niemniej jednak, sekwencyjny algorytm A* bardzo szybko zaczyna zajmować wiele czasu do znalezienia optymalnej ścieżki, nic dziwnego więc, że wciąż są poszukiwane nowe rozwiązania do skrócenia jej złożoności czasowej; 20/30% przyspieszenia może wydawać się słabym/średnim wynikiem, jednak w rzeczywistości jest to znaczne osiągnięcie.

Złożoność czasowa A* zależy od zastosowanej heurystyki. W najgorszym przypadku, gdy przestrzeń poszukiwań jest nieograniczona, liczba rozszerzanych węzłów jest wykładnicza od głębokości rozwiązania (najkrótszej ścieżki) d: O(bd), gdzie b jest współczynnikiem rozgałęzienia (średnia liczba następników na stan).



Algorytm pokazuje również kiedy drogi nie odnaleziono – to znaczy, że cel jest otoczony ścianami i nie ma możliwości dotarcia do niego. Oczywiście w tym momencie algorytm w większości wypadków wykonuje się szybciej, w innych wypadkach stara się znaleźć drogę naokoło i dopiero po chwili okazuje się, że drogi nie ma. W porównaniu do CPU, na karcie graficznej w wypadku nieznalezienia drogi, nieznalezienie następuje szybciej. Dzieje się tak z powodu używania wielu wątków, kiedy jeden wątek dostanie informację o tym, że drogi nie znaleziono, reszta wątków kończy szukanie.



Widać, że nieznalezienie drogi jest o wiele szybsze na GPU, około 50%.

12. Wnioski:

Wszystkie wykresy podkreślają jak GPU sprawdza się do obliczania większej ilości danych. Biblioteka numba do obsługi CUDA w języku Python jest zdecydowanie dobrym rozwiązaniem oraz bardzo wygodnym dla osób, które dopiero zaczynają programowanie na kartach graficznych. Spodziewałem się o wiele gorszych wyników czasowych spowodowanych programowaniem w Pythonie.

Wykonanie projektu pozwoliło mi docenić możliwości optymalizacji programowania na karcie graficznej.

13. Przemyślenia, pomysły na ulepszenie programu:

Wykonany program jest jedynie konceptem, który możnaby w kolejnych krokach uzupełnić o przydatne zastosowania. Ciekawym byłoby generowanie labiryntu, a następnie uruchamianie algorytmu w celu sprawdzenia jak dobrze sobie radzi. Przedstawiona wizualizacja nie jest najlepsza – wartoby stworzyć implementację grafową zamiast punktowej z różnymi wagami na krawędziach, a także dodać wizualizację odkrywanych punktów wraz z odkrywaną drogą, zamiast prostej animacji odkrytej najkrótszej drogi. Poza wymienionymi, heurystyka mogłaby być bardziej skomplikowana od Manhattan Distance, heurystyka, która pozwalałaby algorytmowi na poruszanie się po diagonali pomogłaby w lepszym przedstawieniu problemu.

30. Źródła:

https://pl.wikipedia.org/wiki/Algorytm_A*

https://people.csail.mit.edu/rholladay/docs/parallel_search_report.pdf

https://www.geeksforgeeks.org/a-search-algorithm/

https://medium.com/analytics-vidhya/parallel-a-search-on-gpu-ceb3bfe2cf51