

HPC - CUDA

Tomasz Kępa

28 kwietnia 2019

1 Struktura plików

Rozwiązanie podzielone jest na moduły w następujący sposób:

1. *astar_gpu.cu* – główny moduł zbierający wszystko w całość, parsujący flagi i uruchamiający odpowiedni problem
2. *config.h* – struktura przechowująca flagi przekazane do programu
3. Generyczna implementacja algorytmu GA* znajduje się w pliku *solver.cuh*. Klasa ta trzyma rozwijane stany i odpowiada na konstruowanie pomocniczych struktur danych
4. Detale związane z konkretnym problemami znajdują się w następujących plikach:
 - (a) *pathfinding.cuh* oraz *pathfinding.cu* – implementacja problemu *pathfinding*
 - (b) *slidingpuzzle.cuh* oraz *slidingpuzzle.cu* – implementacja problemu *slidingpuzzle*
5. Struktury danych wykorzystywane przez algorytm:
 - (a) *hashtable.cuh* – hashtablica opisana w artykule
 - (b) *queues.cuh* – struktura przechowująca zadaną liczbę kolejek o ustalonym maksymalnym rozmiarze
 - (c) *lock.cuh* – prosty mutex wykorzystany do stworzenia sekcji krytycznej "per blok"
6. Mniej ważne pliki pomocnicze
 - (a) *errors.h* – pomocnicze funkcje obsługujące błędy alokacji i błędy zwrócone przez funkcje CUDY
 - (b) *memory.h* – pomocnicze funkcje do zwalniania pamięci

2 Opis rozwiązania

Zamknięte stany trzymane są w statycznej tablicy przechowywanej przez klasę *Solver*. W celu zaoszczędzenia pamięci, zamiast wskaźników, używane są indeksy poprzednich stanów. Synchronizacja polega na zwiększaniu przez wątki zmiennej trzymającej następny wolny indeks (za pomocą *atomicAdd*) Typ przechowywanych stanów definiowany jest przez klasy odpowiadające odpowiednim problemom w ich plikach nagłówkowych.

Kolejki mają statyczne rozmiary i jest ich dokładnie tyle samo co wątków. Synchronizacja przebiega w sposób następujący:

1. Pobieranie z kolejki – każdy wątek ma swoją kolejkę i w trakcie czytania zapewniam, że żaden inny wątek w tym samym nie będzie próbował ani nic wpisywać ani wyciągać z kolejki
2. Wpisywanie do kolejek – tutaj zapewniam synchronizację "per blok", czyli w momencie gdy piszę coś do kolejek to pracuje w tym czasie tylko jeden blok, a każdy wątek wpisuje coś wyłącznie do kolejek odpowiadających wątkom o tym samym indeksie w bloku, tak aby nie była konieczna synchronizacja pomiędzy wątkami w tym samym bloku

Algorytm wykonywany jest w wyraźnych krokach opisanych w artykule. Wszystkie wątki zaczynają krok razem i kończą go razem. Dodatkowe synchronizacje opisane są w poszczególnych punktach. W skrócie – fazy :

1. Extract – każdy wątek wyciąga jeden ze swojej kolejki. W tym etapie wątek przygotowuje sobie miejsce w głównej tablicy stanów na zapisanie swoich nowych stanów (przez użycie *atomicAdd*)
2. Expand – każdy wątek ekspanduje swój stan i sprawdza, czy taki lub lepszy (czyli o tym samym *node* ale lepszej wartości *g*) jest już w tablicy. Jeśli nie to wpisuje do tablicy rozwiniętych stanów. Jeśli już znany jest lepszy stan to ten stan jest pomijany. Przygotowane miejsce nie marnuje się – będzie użyte w dalszym kroku. Wartości *f* i *g* liczone są już w tym etapie.
3. Deduplicate – etap ten również wykonywany jest bez dodatkowych synchronizacji ze względu na to, że pozwala na to nasza tablica hashująca. Jeśli jakiś stan zostanie zdeduplikowany dopiero na tym etapie (bo inny wątek dopisał w międzyczasie lepszy stan) to już nie zwalniam pamięci tego stanu, tylko oznaczam go jako usunięty.
4. Compute – ten etap jest połączony z etapem Expand, ze względu na to, że używane są proste heurystyki i niewiele by pomogło ich liczenie dopiero po deduplikacji. Dodatkowo, etapy po deduplikacji są u mnie synchronizowane ”per blok”, przez co tym bardziej korzystne jest policzyć te wartości od razu
5. Push-Back – W tym etapie następuje synchronizacja ”per blok” (aktywny jest tylko jeden blok na raz). Wszystkie wątki z bloku zbierają w jednej tablicy indeksy nowo rozwiniętych stanów, a następnie wpisują stany do odpowiednich kolejek tak jak opisano wyżej.
6. Check Final Conditions – jest to etap nie nazwany explicite w artykule. Podzielony jest na dwa podetapy:
 - (a) – sprawdzenie czy znaleziono już ścieżkę – w trakcie ekspansji stanów każdy wątek zapisuje najlepszy napotkany przez siebie stan końcowy, tutaj następuje zebranie tych wyników najpierw w najlepszy stan ”per blok” a następnie najlepszy globalny stan. Następnie, jeśli spełniony jest warunek końcowy (wszystkie stany w kolejkach mają nie lepsze wartości *f*) to algorytm kończy działanie
 - (b) – sprawdzenie czy nie przeszukano wszystkich stanów – jeden z wątków sprawdza rozmiary wszystkich kolejek. Jeśli wszystkie kolejki są puste to rozwiązanie nie istnieje. Pętla ta jest wykonywana przez tylko jeden wątek ponieważ i tak przez większość czasu wszystkie kolejki są niepuste

3 Heurystyki

3.1 Pathfinding

W tym przypadku zaproponowana w zadaniu metryka jest niepoprawna, ponieważ dopuszczamy ruchy ”na ukos”, przez co bardzo łatwo o przypadki, kiedy odległość policzona przez heurystykę jest większa niż rzeczywista odległość.

Heurystyka zaimplementowana przeze mnie, pozbawiona powyższego błędu, jest następująca:

$$h = \max(|x_1 - x_2|, |y_1 - y_2|)$$

3.2 Sliding puzzle

Tutaj zaproponowana heurystyka jest w porządku, trzeba tylko pamiętać, żeby nie doliczać odległości pustego fragmentu.