[HPC] Zadanie CUDA, rozwiązanie

Mateusz Sieniawski

W moim rozwiązaniu zaimplementowałem algorytm 3 (edge-based parallel BC) z papera podanego w treści zadania (Betweenness Centrality on GPUs and Heterogeneous Architectures). Użyłem reprezentacji grafu COO.

Opiszę pokrótce moją ścieżkę do ostatecznego rozwiązania wraz z zastosowanymi po drodze optymalizacjami. Do trenowania używałem mojej lokalnej karty graficznej GTX 1080. Do porównywania wydajności różnych implementacji brałem pod uwagę całkowity czas działania programu na grafie loc-Gowalla.

Wszystko w jednym kernelu (72,5 min)

Na samym początku zaimplementowałem cały algorytm w postaci jednego kernela. Każdy blok liczył pojedynczy wierzchołek w grafie (najbardziej zewnętrzna pętla w algorytmie). Tablice pomocnicze (dist, sigma, delta) były w pamięci karty w *n* kopiach, gdzie *n* to liczba bloków - każdy blok miał swoją kopię. Synchronizacja odbywała się poprzez pamięć wspólną dla danego bloku (pamięć __shared__).

To rozwiązanie było bardzo niewydajne - na mojej karcie liczyło się ponad godzinę.

Synchronizacja poprzez __managed__ (16 min)

W tym podejściu rozdzieliłem kod pomiędzy 3 kernele, z których każdy był wywoływany wielokrotnie. Te trzy kernele to *forward pass*, *backward pass*, *update betweenness centrality*. Każdy wierzchołek (najbardziej zewnętrzna pętla algorytmu) był liczony kolei, a zrównoleglenie odbywało się w obrębie obecnie liczonego wierzchołka.

W tym podejściu stanąłem przed problemem, aby “wyciągnąć” z pamięci karty wartość zmiennej *cont* po każdym wywołaniu kernela *forward pass.* Użyłem słowa kluczowego __managed__, dzięki któremu do pamięci można odwoływać się zarówno z pamięci karty, jak i z pamięci hosta. Gdy podczas obliczeń kernela *forward pass* zmienna *cont* była ustawiona na *True*, można było się do niej dostać bezpośrednio z hosta, nie wywołując implicite transferu pamięci z karty do hosta.

Synchronizacja poprzez cudaMemcpy (14,14 min)

Okazało się, że korzystanie ze zmiennych __managed__ wiąże się z dodatkowym narzutem czasowym, dlatego w następnej wersji programu po prostu kopiowałem zmienną *cont* z i do pamięci karty pomiędzy wywołaniami kernela *forward pass*.

Po każdym wywołaniu kerneli lub transferze pamięci wywoływałem metodę *cudaDeviceSynchronize()*.

Synchronizacja poprzez cudaMemcpy z przypiętą pamięcią (pinned memory) (14,26 min)

W następnej iteracji spróbowałem jeszcze przyspieszyć mój program używając przypiętej pamięci (pinned memory) - https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/ . W tej wersji zmienna *cont* była przypięta, ponieważ ona było wielokrotnie przesyłana z i do pamięci karty.

Czas tego rozwiązania był porównywalny z poprzednią wersją, jedynie o ~7 sekund wolniejsze.

Ostateczne rozwiązanie (13,7 min)

Używałem jednego strumienia, więc wszystkie wywołania kerneli i transfery pamięci były wykonywane sekwencyjnie i wywoływanie *cudaDeviceSynchronize()* po każdym z nich wiązało się z dodatkowym narzutem. Po usunięciu tej niepotrzebnej synchronizacji program dodatkowo przyspieszył.

Na moim lokalnym GPU najszybsze rozwiązanie liczyło zbiór loc-Gowalla 13,7 min, a **na Titanie V trwało to 4,62 min.**

Podsumowanie

- Napisałem program, który przetwarza graf loc-Gowalla na Titanie V w czasie 4,62 min
- Do profilowania mojego programu używałem nvprof
- Chciałem zaimplementować metodę usuwającą z grafu wierzchołki o stopniu 1, lecz zauważyłem, że w grafie loc-Gowalla nie ma wierzchołków o stopniu 1, zatem ta optymalizacja nie przyspiesza działania programu.
- W pliku brandes.cu znajduje się najszybsze rozwiązanie. Skompilowałem je u siebie z odpowiednimi flagami, aby uruchamiało się na Titan V. Plik wykonywalny nazywa się *brandes*, a Makefile nic nie robi. W folderze *junk*, znajdują się wcześniejsze wersje mojego programu (czasem działające, czasem niedziałające). Pierwotnie nie chciałem umieszczać tego folderu w rozwiązaniu - jeżeli sprawdzający nie chce na nie patrzeć to może patrzeć tylko na najszybsze, ostateczne rozwiązanie.