Obliczenia równoległe na kartach grafinczych CUDA
Projekt
Otoczka wypukła na GPU

Hubert Obrzut

Projekt: obliczanie otoczki wypukłej na GPU za pomocą algorytmu *QuickHull*.

Specyfikacja sprzętu: • **Procesor:** *Intel Core i7-6700HQ 2.60GHz* • **RAM:** *8GB* • **Karta graficzna:** *NVidia GeForce GTX 960M*

Opis projektu: Celem projektu było zaimplementowanie algorytmu obliczającego otoczkę wypukłą na GPU oraz porównanie jego wydajności z algorytmami na CPU. Zaimplementowanie algorytmy to:

- *QuickHull* na GPU,
- *QuickHull* na CPU oraz
- *GrahamScan* na CPU.

Zbiory testowe: Zostały przygotowane trzy zbiory testowe punktów, na których uruchamiane są algorytmu: • **disc:** losowe punkty z okręgu o promieniu 1 • **ring:** losowe punkty z pierścienia o (szerokość 0.1) • **circle:** losowe punkty na okręgu o promieniu 1

Opis działania programu: Specyfikacja argumentów programu:

```
./ConvexHull --gpu|--cpu [--disc|--ring|--circle] [--seed VALUE] NUMBER_OF_POINTS...
```

Nie ma sensu opisywać znaczenia flag, bo każda z nich oznacza to, czego należy się po niej spodziewać. Zgodnie z przyjętą konwencją, flagi w nawiasach [] są opcjonalne. Flagi można podawać w dowolnej kolejności, również przeplatając je z liczbami punktów w danym teście. Testy będą uruchomione w kolejności, w jakiej zostały podane na wejściu. Domyślym zbiorem testowym jest **disc**. Możliwe jest również podanie flagi `--help`.

Przykłady użycia: `./ConvexHull --gpu 100 2000 --circle`, `./ConvexHull 10000000 --cpu`.

Po uruchomieniu, program dla każdego przypadku testowego wypisuje czas, który zajęło obliczenie otoczki wypukłej. Dodatkowo za pomocą *OpenGL* wyświetlane są punkty (po zakończeniu obliczeń dla danego przypadku testowego) wraz z zaznaczoną otoczką wypukłą. Aby przejść do następnego przypadku testowego należy nacisnąć ENTER lub SPACJĘ. Dodatkowo w każdym momencie można wyłączyć program za pomocą ESC lub wyłączenia okienka.

Dodatkowe informacje: W trybie **gpu** program całkowicie działa na karcie graficznej, tzn. bufor danych, jest alokowany za pomocą *OpenGL*, a następnie mapowany na *CUDA*. Początkowo również generowanie punktów testowych odbywało się za pomocą biblioteki *cuRAND*. Okazało się jednak, że takie generowanie jest szybsze (w porównaniu do wygenerowania na CPU i wysłania na GPU) jedynie dla dużej liczby punktów. Ostatecznie punkty są losowane za pomocą *cuRAND* dla $n \geqslant 12,000,000$. Po zunmapowaniu bufora przez *CUDA* punkty są wyświetlane za pomocą *OpenGL* - punkty leżące

na otoczce są większe i zaznaczone innym kolorem. Dodatkowo połączone są krawędziami, tworząc wielokąt wypukły. Punkty otoczki po zakończeniu działania algorytmów są ułożone w taki sposób (na początku tablicy punktów oraz w kolejności otoczki), że zarówno rysowanie wszystkich punktów, punktów otoczki i krawędzi ich łączących można zrealizować bez dodatkowych przekształceń.

Wyniki: QH - Quick Hull, GS - Graham Scan

N	**QH GPU**	**QH CPU**	**GS CPU**	**QH Speedup (CPU/GPU)**
1K	9.2	0.2	0.1	0.01
10K	9.3	1.8	1.3	0.14
100K	23.7	17.6	13.5	0.57
1M	84.7	193.9	160.4	1.89
2M	142.7	403.9	344.7	2.42
4M	256.9	815.6	711.6	2.77
10M	579	2,154.5	1875.0	3.24
15M	810.9	3,345.8	2,913.0	3.59
20M	1,064.8	4,419.6	3,977.5	3.74
25M	1,342.50	5549.6	5069.2	3.78

Tabela 1: Zbiór **circle**: Czas działania algorytmów dla **N** punktów (ms).

N	**QH GPU**	**QH CPU**	**GS CPU**	**QH Speedup (CPU/GPU)**
1K	7.9	0.2	0.1	0.0
10K	6.7	1.1	1.4	0.2
100K	9.5	12.7	14.1	1.5
1M	28.7	141.3	162.5	5.7
2M	45.1	298.2	339.7	7.5
4M	81.1	619.5	714.1	8.8
10M	184.3	1,642.4	1,883.8	10.2
15M	262.1	2,517.7	2,906.7	11.1
20M	393.3	3,404.0	3,963.0	10.1
25M	498.4	4,320.0	5,059.7	10.2

Tabela 2: Zbiór **ring**: Czas działania algorytmów dla **N** punktów (ms).

N	**QH GPU**	**QH CPU**	**GS CPU**	**QH Speedup (CPU/GPU)**
1K	4.9	0.2	0.1	0.02
10K	5.2	1.0	1.4	0.27
100K	6.9	12	13.2	1.91
1M	22.8	135	162.5	7.13
2M	35.1	277.1	342.4	9.75
4M	57.6	585.8	710.5	12.34
10M	130.5	1541.5	1896.9	14.54
15M	187.3	2,375.4	2,934.8	15.67
20M	244.3	3,221.7	3,923.2	16.06
25M	359.3	4161.7	5113.8	14.23

Tabela 3: Zbiór **disc**: Czas działania algorytmów dla **N** punktów (ms).

Konkluzje: Po pierwsze należy zauważyć, że zyski w wydajności działania programu wynikające z przeprowadzenia obliczeń za pomocą CUDA są tym większe im więcej punktów znajduje się na wejściu. Nie powinno to być niczym zaskakującym, i wynika bezpośrednio z narzutu czasowego, wynikającego z samego używania jąder CUDA, komunikacji z CPU. Zyski w wydajności zaczynają się od 100K/1M

punktów. Są one tym większe, im bardziej zbiór punktów przypomina zbiór punktów typu **disc**, gdzie możemy zaobserwować nawet 15-krotne przyspieszenie. Ogólne przypiesznie spowodowane jest dobrym wykorzystaniem architektury karty graficznej do wykonywania kolejnych kroków algorytmu *Quick Hull* - ich masywne zrównoleglenie. Dodatkowo w przypadku **disc**, znaczna większość wszystkich punktów **NIE** znajduje się na otoczce. To oraz natura algorytmu *Quick Hull* sprawia, że liczba sekwencyjnych, dużych kroków algorytmu jest mała, a na pewno mniejsza w stosunku do reszty zbiorów testowych. Im mniejsza liczba kroków sekwencyjnych, a zatem również mniejsza komunikacja i synchronizacja karty graficznej z procesorem, tym większe korzyści płyną ze zrównoleglenia algorytmy w stosunku do procesorowych odpowiedników. Nadal jednak należy zauważyć, że nawet w przypadku **circle**, gdzie znaczna liczba punktów znajduje się na otoczce, dochodzimy do około 4-krotnego przyspiesznia. Niewątpliwie wyniki te zależą w dużym stopniu od karty graficznej, na której zostały przeprowadzone, ale jednocześnie pokazują trend w zależności od liczby i rodzaju punktów oraz to, że obliczenie otoczki wypukłej na karcie graficznej może przynieść wymierne korzyści dla wydajności programu.

Dodatkowym spostrzeżeniem jest to, że algorytm *Quick Hull* na CPU w większości przypadków przebija pod względem czasu działania algorytm *Graham Scan*. Przypomnijmy, że pesymistyczna złożoność *Quick Hull* to $O(n^2)$, podczas gdy dla *Graham Scan* jedynie $O(nlogn)$. W praktyce jednak trudno jest znaleźć zbiór punktów, na którym *Quick Hull* wymagałby czasu kwadratowego, szczególnie, gdy jest się ograniczonym przez precyzję liczb zmiennoprzecinkowych. Podobnie jak w przypadku sortowania, algorytm *Quick Sort*, którego zasada działania jest nie tylko podobna, ale nawet analogiczna do algorytmu *Quick Hull*, zwykle przebija swoich *nlogn*-owych kolegów w sortowaniu, tak samo *Quick Hull* oferuje wysoką wydajność dla problemu obliczania otoczki wypukłej zbioru punktów.