SZYBKIE MASYWNIE RÓWNOLEGŁE METODY OBLICZENIOWE

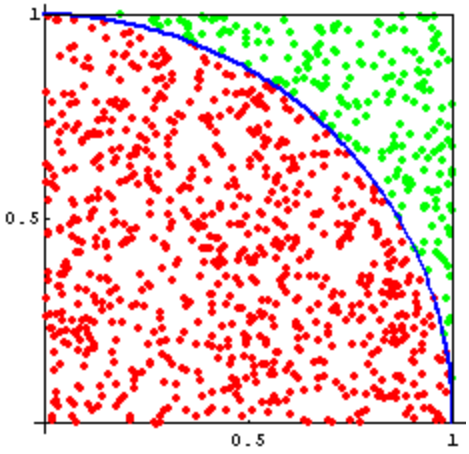
PROJEKT

OPRACOWAŁ: PAWEŁ WALCZAK

ESTYMACJA WARTOŚCI LICZBY PI ZA POMOCĄ METODY MONTE CARLO

1. **Cel projektu.**
   Celem projektu było zaimplementowanie programu estymującego wartość liczby PI za pomocą metody Monte Carlo. Należało zaimplementować oraz porównać szybkości działania klasycznej wersji sekwencyjnej CPU jak i wersji zrównoleglonych wykorzystujące różne platformy (CUDA, C++ AMP, OpenCL).

2. **Opis metody.**
   Metody Monte Carlo opierają się na wykorzystaniu w różnych celach (całkowanie, różniczkowanie, czy estymacja wartości PI) losowania. Znajdują zastosowanie w przypadkach gdy modelujemy procesy, których rozwiązania ciężko znaleźć w sposób analityczny.
   Metodę Monte Carlo do estymacji wartości PI, można opisać w kilku prostych krokach:
   - Wylosuj dowolną liczbę punktów 2D o wartościach w przedziale (0, 1),
   - Dla kolejnych punktów sprawdź, czy należą one do okręgu o promieniu o długości 1 i o środku w punkcie (0, 0).
   - Pomnóż przez 4, stosunek liczby punktów należących do wspomnianego okręgu do liczby wszystkich wylosowanych **punktów**



3. **Różnice między metodą sekwencyjną a metodą zrównoleglonąą**
   Napisanie wersji sekwencyjnej jest bardzo intuicyjne. Wykorzystujemy w tym celu jeden wątek zajmujący się wszystkimi obliczeniami. Wersja równoległa wykorzystuje do obliczeń kilka wątków (zarazem procesorów) w tym samym czasie. Kilka wątków w tym samym czasie losuje kolejne punkty i sprawdza które z nich należą do okręgu jednostkowego. Oczekujemy, że programy równolegle znacznie przyspieszą proces estymacji wartości PI.
4. **Opis systemu komputerowego.**
   Do wykonania projektu użyłem komputera PC w laboratorium CS400. Komputer składa się z CPU (Procesor Intel I7) , oraz GPU (NVidia GTX 750). Dla wersji równoległych będziemy korzystać z GPU, które swoją budową (wiele procesorów obliczeniowych) wspomagają proces programowania równoległego.
5. **Wyniki**
   Głównej analizie poddamy czasy działania programów dla różnych wersji (CPU, OpenCL, C++ AMP, CUDA). Będziemy też sprawdzać skuteczność (precyzję oszacowania wartości PI), jednakże ta powinna być niezależna od wykorzystanego programu i służyć jedynie weryfikacji poprawności działania algorytmu . Czas wykonania programu powinien być zależny od liczby wylosowanych punktów.

CPU:

Ilość wylosowanych punktów	Estymacja wartości PI	Czas wykonania programu [ms]
100	**3.16**	**0**
1000	**3.152**	**0**
10000	**3.146**	**1**
100000	**3.13572**	**5**
1000000	**3.13999**	**44**
10000000	**3.14148**	**445**
100000000	**3.1412**	**4455**
1000000000	**3.14139**	**45998**

Zauważamy, że widzimy zależność w przybliżeniu liniową między liczbą wylosowanych punktów a czasem wykonania programu. 10-krotny wzrost liczby losowanych punktów powoduje 10-krotny wzrost czasu wykonania programu. Nie można powiedzieć, że estymacja wartości PI jest idealna. Spowodowane to jest prawdopodobnie tym, że nie generator liczb losowych jest pseudolosowy, i nie zapewnia pełnej losowości.

CUDA:

CUDA to dedykowane rozwiązanie dla kart graficznych NVidia. Zapewnia ona wykorzystanie sprzętowych właściwości kart graficznych do programowania równoległego. Korzystamy tutaj także z dostarczonego przez CUDA zrównoleglonego generatora liczb losowych.

Ilość wylosowanych punktów = GridSize * BlockSize * PtsPerThread	Grid Size	Block Size	Ilość losowanych punktów i obliczeń przez 1 wątek	Estymacja PI	Czas wykonania programu [ms]
4096	1	1	4096	3.141968	137
131072	1	32	4096	3.131968	124
131072	32	1	4096	3.139293	137
4194304	32	32	4096	3.149142	124
268435456	256	256	4096	3.141582	134

268435456	128	512	4096	3.141582	146
536870912	128	1024	4096	3.141596	159
536870912	256	256	8192	3.141838	299
8388608	256	256	128	3.141520	278

Zauważamy, że dla odpowiedniego wykorzystania sprzętowych możliwości GPU, uzyskujemy wyniki dużo szybsze niż dla CPU. W wypadku gdy tak naprawdę nie stosujemy równoległości (Grid Size = 1, Block Size = 1), wynik jest wolniejszy niż dla odpowiedniej konfiguracji CPU (spowodowane kopiowaniem pamięci z CPU do GPU i odwrotnie). Jednakże w wypadku prawidłowego korzystania z GPU, czyli wykorzystania konfiguracji np. Block Size = 1024 (max), Grid Size = 128, Threads Per Block = 4096, mamy ponad 100-krotne przyspieszenie względem CPU. Zauważmy, że dla dwukrotnego zwiększenia liczby punktów przez jeden wątek, przy zachowaniu takich samych Block Size i Grid Size, mamy dwukrotne zwiększenie czasy wykonania (ponieważ, dwukrotnie zwiększyliśmy liczbę pracy (szeregowej) wykonywanej przez każdy wątek).

OpenCL

OpenCL to framework wspomagający pisanie aplikacji działających na heterogenicznych platformach składających się z różnych jednostek obliczeniowych (CPU, GPU). Główną zaletą OpenCL jest to, że można użyć jednego otwartego standradu zamiast zamkniętych wspierających sprzęt tylko jednego producenta (np. CUDA tylko dla kart graficznych NVidia).

Liczba wylosowanych punktów	**Liczba generatorów i jednostek obliczających**	**Liczba losowanych punktów i obliczeń przez 1 generator**	**Estymacja PI**	**Czas wykonania programu [ms]**	**Losowanie [ms]**	**Obliczanie PI [ms]**	**Kopiowanie [ms]**
3200	32	100	3.150000	0	0	0	0
32000	32	1000	3.136500	1	1	0	0
320000	32	10000	3.140438	8	7	1	0

3200000	32	100000	3.142184	82	68	13	0
32000000	32	1000000	3.141745	808	681	127	0
51200000	512	100000	3.141475	122	69	53	0
200000	2	100000	3.134940	54	46	8	0
800000	8	100000	3.141345	58	48	10	0
16000000	16	100000	3.142773	66	55	11	0

Zauważamy, że w porównaniu do CPU, OpenCL w najlepszej konfiguracji parametrów jest około 20-krotnie szybszy. Prawdopodobnie dla jeszcze większej liczby generatorów (de facto większej liczby osobnych wątków) wynik byłyby jeszcze szybsze. Jednakże wyniki osiągane są nieco wolniejsze od CUDA. Również i tutaj ważne było odpowiednie zrównoleglenie procesu. Najbardziej czasochłonnym zadaniem okazało się losowanie punktów. Z pewnych powodów, nie dało się też uruchomić OpenCL dla liczby punktów większej niż 50 milionów. Prawdopodobnie był to problem z dostępną dla platformy pamięcią.

C++ AMP

[Wikipedia]

'C++ AMP is a native programming model that contains elements that span the C++ programming language and its runtime library. It provides an easy way to write programs that compile and execute on data-patallel hardware, such as graphic cards (GPUs). C++ AMP is a library implemented on DirectX11 and an open specification from Microsoft for implementing data parallelism directly in C++. […]'

Korzystając z C++ AMP, wykorzystujemy specjalne struktury danych przeznaczone do przetwarzania równoległego. Posłużyłem się również algorytmem losowania punktów Mersenne Twister (efektywny obliczeniowo, przystosowany do przetwarzania równoległego i dający

wysokiej jakości losowe punktu). Wykorzystałem ogólnie dostępne parametry dla algorytmu losowania. Korzystałem także z biblioteki *AMP_ALGORITHMS* w celu efektywnego (równoległego) operowania na strukturach danych.

Liczba wylosowanych punktów	**Liczba generatorów**	**Liczba losowanych punktów i obliczeń na generator**	**Estymacja PI**	**Czas wykonania programu [ms]**
40960	4096	10	3.0418	15
409600	4096	100	3.16109	16
4096000	4096	1000	3.14183	26
40960000	4096	10000	3.14228	114
2048000	2048	1000	3.14361	23
1024000	1024	1000	3.14192	20
512000	512	1000	3.12361	18

Zauważmy, że liczba generatorów nie wpływa znacząco na czas wykonania programu (zwiększa zaś liczbę losowanych punktów), co świadczy o zrównolegleniu procesu. Wpływ natomiast ma liczba obliczeń i losowanych punktów przez jeden generator, jednakże wpływ ten zauważamy dopiero przy skoku owego parametru od wartości 100 do 1000. Prawdopodobnie dla małej liczby losowanych punktów przez jeden generator głównym kosztem jest samo tworzenie wątków i kopiowanie pamięci, nie zaś same obliczenia. Również tutaj, podobnie jak dla OpenCL, przy większej liczbie losowanych punktów niż 50 milionów, następował błąd programu. Wzrost wydajności jest praktycznie identyczny jak dla OpenCL.

WNIOSKI KOŃCOWE:

Wykorzystanie narzędzi do zrównoleglania programu, dało bardzo korzystne wyniki. Zwiększyła się znacząco wydajność programu (nawet 200-krotnie w porównaniu do wersji klasycznej CPU). Estymacje wartości PI dla takiej samej liczby losowanych punktów, są z grubsza takie same

zarówno dla sekwencyjnego przetwarzania jak i równoległego. Najwydajniejszym rozwiązaniem okazało się być platforma CUDA. Jest to framework, który jest dostosowany i optymalizowany specjalnie pod sprzęt NVidia w przeciwieństwie do uniwersalnych OpenCL oraz C++ AMP. Szybsze przetwarzanie jest kosztem mniejszej uniwersalności. Dla CUDA nie występowały również problemy dla zbyt dużej liczby losowanych punktów (powyżej 50 milionów). Stąd jeżeli zależy nam na jak najszybszym rozwiązaniu oraz dysponujemy kartą graficzną NVidia, należy wybrać CUDA. Jednakże jeśli zależy nam na uniwersalności, lub nie posiadamy GPU NVidia, najlepszym rozwiązaniem będzie OpenCL oraz C++ AMP.

6. Źródła.

- Wikipedia
- Algorytm Mersenne Twister, Makoto Matsumoto, Takuji Nishimura
- Szablon C++ AMP z wykorzystaniem Mersenne Twister, Microsoft, NVidia
- The OpenCL Programming Book, FixStars
- Internet