Grupa 6	Dariusz Szczupak, Kamil Wanat	14/11/2018

Zadanie 6 - Liczby pierwsze CUDA

Zadanie laboratoryjne polegało na zaimplementowaniu programu sprawdzającego czy podane liczby są liczbami pierwszymi, czy złożonymi. Program zaimplementowany został z wykorzystaniem języka programowania CUDA C umożliwiającego programowanie z wykorzystaniem procesorów graficznych GPU. Problem został rozwiązany przy pomocy algorytmu "naiwnego". Jest to metoda nieco wolniesza niż metody probabilistyczne, jednak dająca zdecydowanie lepsze wyniki. Polega ona na dzieleniu testowanego elementu przez kolejne liczby od 2 do pierwiastka kwadratowego z tejże. W naszym programie podejście naiwne zostało nieco zmodyfikowane. Zamiast sprawdzać dzielniki od 2 do pierwiastka z każdej liczby, znajdujemy najpierw liczbę maksymalną, a następnie testujemy wszystkie liczby przy pomocy dzielników od 2 do sqrt(max). Ponadto nie testujemy każdej liczby osobno dla danego dzielnika, lecz cały zbiór.

```
__global__ void primeTesting (number* tab, uint sqr, uint d)
{
uint tid=blockIdx.x;
uint i,j;
for (i=2;i<=sqr;i++) {
        for (j = tid; j <d; j+=gridDim.x) {
                if((tab[j].value%i==0)&&(tab[j].value!=i))
                    tab[j].prime=false;
        }
    }
}
```

Powyższy kod przedstawia funkcję primeTesting uruchamianą na procesorze graficznym GPU. Odpowiedzialna jest ona za testowanie pierwszości liczb. Jako argument przyjmuje wskaźnik do obszaru pamięci w którym znajdują się dane liczb do testowania, pierwiastek z wartości maksymalnej liczby, oraz rozmiar danych. Każdy z wątków odpowiada za sprawdzenie części liczb domniemanie pierwszych. Liczby do sprawdzenia przez wątek definiowane są poprzez zmienną tid - początkowo przyjmującą numer bloku, oraz zwiększaną o calkowitą ilosć bloków. Na kolejnym listingu widoczny jest fragment kopiowania danych do pamięci karty graficznej, wywołanie funkcji jądra oraz kopiowanie powrotne danych do pamięci hosta.

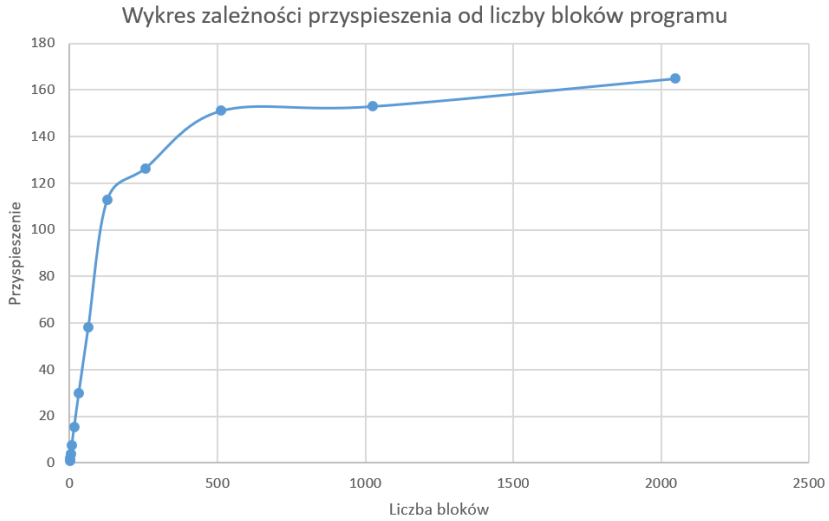
```
    number* tab2;
    number* temp = tab.data();
    cudaMalloc( (void**)&tab2, d * sizeof(number) );
    cudaMemcpy(tab2, temp, d * sizeof(number), cudaMemcpyHostToDevice);
    primeTesting <<< blockNumber, 1 >>> (tab2, sqr,d);
    number * result;
    result= (number *) malloc (d*sizeof(number));
    cudaMemcpy(result, tab2, d * sizeof(number), cudaMemcpyDeviceToHost);
```

Poniższa tabela przedstawia zalezność przyspieszenia od ilości bloków wykorzystywanych przez program.

Wykres przyspieszenia programu testującego liczby pierwsze pokazuje czy i jak udało się zrównoleglić działanie programu. Pierwszą różnicą widoczną w porównaniu do poprzednich zadań jest rozmiar bloków wykorzystanych do zrównoleglenia programu. W poprzednich zadaniach maksymalna liczba procesorów wynosila 12, w tym przypadku liczba ta

Liczba bloków	Przyspieszenie
1	1
2	1,922430365
4	3,823976317
8	7,628674784
16	15,25149128
32	30,08939713
64	58,210549
128	112,8851456
256	126,3510546
512	151,0486673
1024	152,9736427
2048	164,897505
4096	164,51219
8192	164,7069294
16384	160,7519724

Tablica 1: Zależność przyspieszenia od ilości bloków



Rysunek 1: wykres przyspieszenia

wzrasta do ponad 16000. Przyspieszenie uzyskane wynosi ponad 150 co w porównaniu do poprzednich technologii jest porażającą wartością. Wykres zaczyna spowalniać w okolicach 500 uruchomionych równolegle bloków. Związane to jest z ograniczeniami fizycznymi architektury karty graficznej. Ponadto rozmiar użytych danych ma znaczący wpływ na ilość operacji przeprowadzonych przez dany wątek. Sprowadza się to do sytuacji w której część wywołanych wątków jest bezużyteczna podczas gdy inne nadal pracują.

dane przeprowadzanych testów:

- wzięto pod uwagę średnią pomiarów czasów wykonania programu dla liczby wątków z przedziału od 1 do 16384. program był uruchamiany dla pliku pobranego z internetu.
- do mierzenia czasu wykorzystano zdarzenia CUDA.
- testy zostały wykonane na serwerze cuda.iti.pk.edu.pl .
- w czasie testów na serwerze zalogowany był jedynie użytkownik testujący.