Programowanie na platformie CUDA

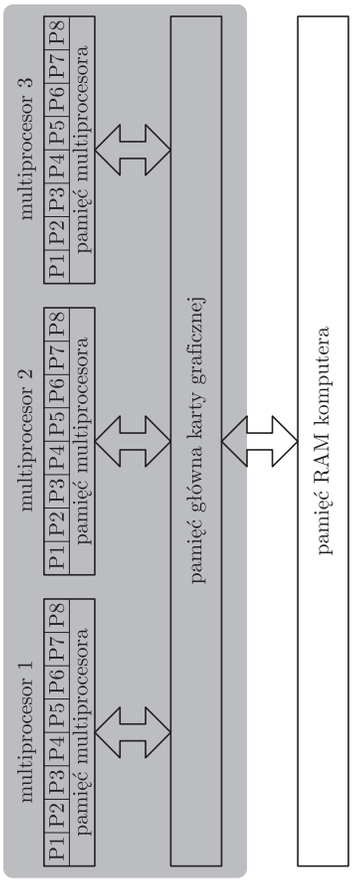
Wojciech ŚMIETANKA[*]

Dziesięć lat temu kolejne generacje procesorów charakteryzowały się wykładniczo rosnącą częstotliwością taktowania. Teraz ta sytuacja uległa zmianie. Obecnie to liczba rdzeni w jednym procesorze zaczyna rosnąć wykładniczo. W użytku są już procesory firmy Intel dla zwyczajnych PC-tów mające 8 rdzeni, a co jakiś czas pojawiają się informacje o tym, że niedługo zostanie wyprodukowany procesor o 50 rdzeniach. Niestety, pisanie programu, który wykorzystuje w pełni moc n rdzeni, nie jest dla programisty łatwym zadaniem. Jest to spowodowane dość uciążliwymi metodami synchronizacji wielu wątków i używanym modelem pamięci, który jest bardziej przystosowany do programowania jednowątkowego.

Flops to jednostka wydajności sprzętu komputerowego, oznaczająca liczbę operacji zmiennoprzecinkowych wykonywanych przez sprzęt w ciągu sekundy. Gflops to miliard takich operacji na sekundę.

Okazuje się, że w większości współczesnych komputerów znajduje się drugi układ scalony, który od początku był projektowany do obliczeń równoległych. Chodzi o kartę graficzną. Zwyczajowo karta graficzna ma za zadanie obliczyć wartości koloru pojedynczych pikseli na ekranie. Widać, że wyniki poszczególnych obliczeń są niezależne. W tym artykule chciałbym przybliżyć platformę CUDA, która służy do programowania na kartach graficznych firmy NVIDIA. O tym, że warto zastanowić się nad programowaniem na karcie graficznej, może świadczyć następujące porównanie:

- jeden z najlepszych obecnie procesorów Intela – Core i7 980X – kosztuje ok. 1000 dolarów i osiąga moc obliczeniową ok. 100 Gflopsów;
- jedna z najlepszych kart graficznych NVIDIA – GeForce GTX 580 – kosztuje ok. 500 dolarów i oferuje moc obliczeniową ok. 1500 Gflopsów.



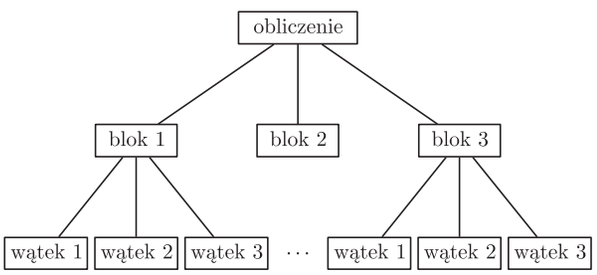
Rys. 1. Struktura karty graficznej (część zacieniona) i przepływ danych w pamięci komputera podczas obliczeń.

Struktura karty graficznej. Karta graficzna składa się (w uproszczeniu) z *multiprocesorów*, przy czym jeden multiprocesor składa się z 8 lub 16 pojedynczych procesorów i jednej niewielkiej współdzielonej pamięci na cały multiprocesor (pamięć multiprocesora jest szybka, umożliwia jednoczesny odczyt/zapis), oraz jednej dużej pamięci, która jest wspólna dla wszystkich multiprocesorów. Pamięć ta jest o rząd wielkości wolniejsza od pamięci multiprocesora, umożliwia także jednoczesny odczyt/zapis.

Standardowo program używający do obliczeń karty graficznej będzie działał według następującego schematu: skopiowanie danych z pamięci komputera do pamięci głównej karty graficznej i dalej do pamięci multiprocesora; wykonanie obliczenia na multiprocesorach; skopiowanie częściowych wyników z pamięci multiprocesora do pamięci głównej karty graficznej i skopiowanie końcowego wyniku do pamięci komputera.

Struktura obliczeń. A teraz ciekawe pytanie: w jaki sposób programista rozdziela zadania między multiprocesory i procesory? Przypuśćmy, że chcemy wykonać jakieś duże obliczenie K. Programista dzieli je na zbiór mniejszych obliczeń: $K = \{b_1, b_2, \ldots, b_n\}$. Pojedynczy element obliczenia b_i to blok, który jest z kolei złożony z pojedynczych wątków:

$$b_i = \{w_{i1}, w_{i2}, \ldots, w_{im}\}.$$



Rys. 2. Struktura obliczeń.

Specyfikacja techniczna platformy CUDA daje następujące gwarancje: każdy blok będzie wykonywany w obrębie jednego multiprocesora; nic nie wiadomo o tym, w jakiej kolejności wykonają się bloki; jeden wątek wykona się na jednym procesorze; w obrębie jednego multiprocesora wykonuje się w danym momencie co najwyżej jeden blok.

Zadaniem programisty jest napisanie kodu pojedynczego wątku. Każdy wątek wykonuje dokładnie ten sam kod, przy czym wątek może sprawdzić, w którym bloku się znajduje, a także jaki ma numer wewnątrz tego bloku. Obliczenie wykonywane przez wątek zależy od tak zdefiniowanych współrzędnych tego wątku.

[*]student, Wydział Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego

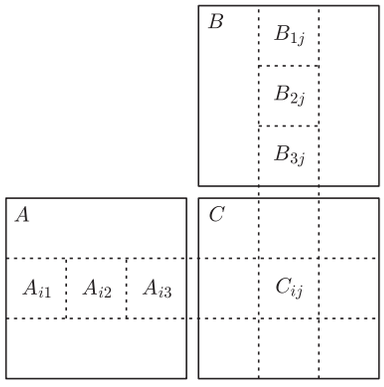
Należy jeszcze podkreślić, że programista nie specyfikuje dokładnie, na którym procesorze wykona się dany wątek. Zadaniem programisty jest zdefiniować strukturę obliczeń, a przydziałem wątków do procesorów zajmuje się CUDA.

Praktyczny przykład – mnożenie macierzy. Chcemy pomnożyć dwie macierze A i B, obie wymiaru $n \times n$. Przez C oznaczmy macierz wynikową. Obliczenie macierzy C wprost z definicji wymaga wykonania $O(n^3)$ operacji. Używając platformy CUDA, można zaproponować lepsze rozwiązanie, w którym wszystkie macierze dzielimy na podmacierze rozmiaru $k \times k$ (oznaczamy je przez A_{ij}, B_{ij}, C_{ij}). Dla uproszczenia analizy założymy, że rozmiar macierzy n jest wielokrotnością rozmiaru podmacierzy k. Jeden blok obliczenia będzie miał za zadanie obliczyć jedną z podmacierzy C_{ij}. Zauważmy, że $C_{ij} = \sum_{l=1}^{n/k} A_{il} \cdot B_{lj}$. W jednej iteracji będziemy chcieli obliczyć jeden składnik postaci $A_{il} \cdot B_{lj}$. Można to zrobić następująco.

Dla każdego $l \in \{1, 2, \ldots, n/k\}$:

- pobieramy do pamięci multiprocesora macierze A_{il}, B_{lj} – tę pracę wykonuje pierwszy wątek z każdego bloku;
- równolegle obliczamy $A_{il} \cdot B_{lj}$, każdej komórce macierzy wynikowej przyporządkowujemy jeden wątek odpowiedzialny za obliczenie tej wartości;
- dodajemy do wyniku obliczoną wartość $A_{il} \cdot B_{lj}$.

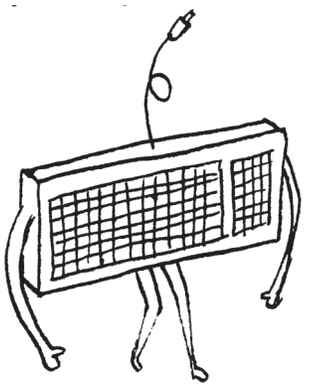
Po wykonaniu obliczeń dla wszystkich l mamy obliczoną podmacierz C_{ij}, którą możemy zapisać do pamięci głównej karty.



Rys. 3. Schemat mnożenia macierzy. Tutaj $n/k = 3$ oraz $C_{ij} = A_{i1} \cdot B_{1j} + A_{i2} \cdot B_{2j} + A_{i3} \cdot B_{3j}$.

Spróbujmy oszacować złożoność tego rozwiązania. W tym celu przez p oznaczmy liczbę procesorów wewnątrz jednego multiprocesora, a przez m – liczbę multiprocesorów.

Przyjrzyjmy się czasowi wykonywania pojedynczego bloku. Przy obliczaniu iloczynu $A_{il} \cdot B_{lj}$ najpierw pobieramy dwie macierze rozmiaru $k \times k$ do pamięci multiprocesora, co zajmuje czas $O(k^2)$, a następnie wykonujemy k^3 mnożeń, ale to zrównolegla się między p procesorów, koszt tego wynosi więc $O\big(\frac{k^3}{p}\big)$. Taki ciąg operacji należy wykonać dla $l \in \big\{1, 2, \ldots, \frac{n}{k}\big\}$. Zatem jeden blok wykonuje się w czasie $O\big(\big(\frac{k^3}{p} + k^2\big) \cdot \frac{n}{k}\big)$.

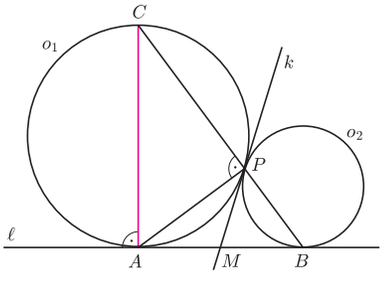


Mamy $\frac{n^2}{k^2}$ bloków, ich wykonanie zrównoleglamy między m multiprocesorów, zatem łączny czas to $\frac{n^2}{mk^2}$ razy czas wykonania pojedynczego bloku. Daje to złożoność czasową $O\big(\frac{n^3}{mk^3} \cdot \big(\frac{k^3}{p} + k^2\big)\big)$. Przyjmując, że $k = p$, co jest możliwe, gdyż to programista dobiera wartość k, daje to złożoność $O\big(\frac{n^3}{mp^3} \cdot p^2\big)$, czyli $O\big(\frac{n^3}{mp}\big)$. W ten sposób klasyczną złożoność $O(n^3)$ podzieliliśmy przez $m \cdot p$, czyli liczbę procesorów. We wspomnianej wcześniej karcie GeForce GTX 580 mamy $m = 32$, $p = 16$, czyli programista ma do dyspozycji 512 procesorów. To daje duże możliwości zrównoleglania.

Czy przyszłość programowania leży w równoległości? Jest cała lista problemów związanych z programowaniem równoległym, jak np.: mała liczba doświadczonych programistów; wyższy niż w przypadku programowania jednowątkowego poziom skomplikowania; część algorytmów ciężko się zrównolegla (np. nie jest znany dobry równoległy algorytm sprawdzania, czy w danym grafie dwudzielnym jest doskonałe skojarzenie); błędy typu *race condition*, gdy więcej niż jeden wątek jednocześnie zapisuje w danym miejscu w pamięci.

Z drugiej strony programiści chcieliby, aby ich programy działały szybko. Podnoszenie wydajności pojedynczego rdzenia nie odbywa się w takim tempie jak kiedyś, można więc przypuszczać, że do dalszego poprawiania wydajności oprogramowania potrzebne jest odrzucenie modelu programowania z jednym procesorem i myślenie w sposób równoległy. NVIDIA CUDA jest jednym z ciekawszych modeli oferujących możliwość programowania współbieżnego. Wprowadza to za cenę jednej zasadniczej nowości: programiści muszą nauczyć się dzielić obliczenia tworzonego oprogramowania na możliwie niezależne kawałki. Nie jest to takie łatwe, bo całkowicie zmienia sposób myślenia o programowaniu.

Rozwiązanie zadania M 1325.
Rozważmy wspólną styczną k okręgów o_1 i o_2 przechodzącą przez punkt P.



Przecina ona prostą ℓ w punkcie M. Ponieważ $MA = MP = MB$, więc trójkąt ABP jest prostokątny. Wobec tego AC jest średnicą okręgu o_1, jako cięciwa, na której oparty jest kąt prosty APC, a średnica okręgu jest prostopadła do stycznej w swoim końcu.