Zaawansowane zastosowania kart graficznych

Zestaw 2. Współpraca wątków

Zadania demonstrujące przetwarzanie tablic z użyciem pamięci współdzielonej i synchronizacji na poziomie bloku.

1. Napisz program który przepisuje do tablicy umieszczonej w pamięci współdzielonej tablicę z pamięci globalnej, następnie do każdego elementu (oprócz zerowego) dodaje poprzedni element z tablicy. Ostatecznie wynik powinien być przepisany z pamięci współdzielonej do pamięci globalnej. Przykładowo, dla tablicy 1,2,3 wynik powinien wyglądać następująco: 1,3,5. Ogranicz maksymalny rozmiar tablicy do 1024 tak, aby całość zadania mogła być wykonana przez jeden blok wątków.
2. Napisz program obliczający średnią kroczącą na tablicy x. Niech liczba wartości w tablicy wynosi n a wielkość okna wynosi d. Zakładamy, że wielkość okna jest liczbą nieparzystą. Średnia krocząca dla tablicy x polega na wyliczeniu tablicy y takiej, że:

$$y[i] = \frac{1}{d} \sum_{j=i-\frac{d-1}{2}}^{i+\frac{d-1}{2}} x[j]$$

   Dla indeksów j wychodzących poza zakres tablicy przyjmuje się odpowiednio pierwszą lub ostatnią wartość w tablicy x. Algorytm powinien działać dla tablicy o dowolnym rozmiarze w następujący sposób:
   1) Dla każdego bloku o rozmiarze bs należy zaalokować pamięć współdzieloną na $bs + d - 1$ liczb.
   2) Wątki w bloku o numerze b kopiują z tablicy x do pamięci współdzielonej wartości o numerach od $b * bs - \frac{d-1}{2}$ do $(b+1) * bs - 1 + \frac{d-1}{2}$. Są to wszystkie wartości potrzebne do obliczenia średniej kroczącej na pozycjach tablicy odpowiadających numerom wątków. Kopiowanie można zrobić równolegle następująco (*btid* – numer wątku w bloku, *gtid* – numer wątku w gridzie):
      a. $shared[btid] = x[gtid - (d-1)/2]$
      b. $if\ (btid < d-1)\ then\ shared[bs + btid] = x[bs + gtid - (d-1)/2]$
      c. Postaraj się zrozumieć dlaczego to zadziała. Kluczem do zrozumienia punktów a i b jest obserwacja, że wątek o numerze *btid/gtid* pobiera z tablicy *x* do pozycji *btid* w tablicy współdzielonej pierwszą potrzebną mu później (do obliczeń) wartość z okna.
      d. Zastanów się nad tym, czy konieczna jest synchronizacja pomiędzy punktem a i b, oraz po b.
      e. **Uwaga!** Powyższe linijki nie uwzględniają wychodzenia poza zakres tablicy x. Użyj *min* i *max* na obliczonych indeksach w tablicy *x* w celu zapobieżenia wyjściu poza tablicę.
   3) Wątek o numerze *gtid* oblicza w wynikowej tablicy y pozycję o numerze *gtid* na podstawie odpowiednich danych w tablicy w pamięci współdzielonej, tj. indeksów od $btid$ do $btid + d - 1$.

Zadanie drugie zaimplementowane za pomocą omówionego algorytmu nie jest realizowane zbyt wydajnie. Wiele zakresów liczb jest sumowanych niepotrzebnie wielokrotnie. Aby rozwiązać ten problem, możliwe jest zastosowanie algorytmu **inclusive scan**. **Inclusive scan** można zdefiniować następująco. Mając daną tablicę *x* należy odnaleźć tablicę *y*, w której $y[i] = \sum_{j=0}^{i} x[j]$. Obliczenia mogą być również przeprowadzone w miejscu.

Kolejne pięć zadań pokazują jak można zaimplementować algorytm obliczający **inclusive scan**. Przedstawiony algorytm jest prosty, choć nieoptymalny. Lepszy algorytm zostanie przedstawiony na wykładzie w swoim czasie ☺. Po zaimplementowaniu zastanów się, jak można by zrealizować obliczanie średniej kroczącej przy użyciu **inclusive scan**.

3. Napisz program, który dla każdej pozycji w tablicy oblicza sumę wszystkich wartości przed nią i nią samą (sekwencyjnie), np. dla tablicy 1,2,3,4 wynik powinien wynosić: 1,3,6,10. W celu optymalizacji obliczeń przepisz dane z tablicy w pamięci globalnej to tablicy w pamięci współdzielonej i na nich wykonaj obliczenia. Ogranicz algorytm do jednego bloku. Załóż, że tablica ma rozmiar będący potęgą 2. W ten sposób obliczana jest operacja scan w ramach bloku w naiwny sposób.
4. Zmodyfikuj powyższy algorytm wykorzystując algorytm Kogge-Stone'a. Algorytm działa w następujący sposób:

```
//Kogge-Stone adder
//x – tablica wejściowa
//n – rozmiar tablicy (potęga 2)

for d := 0 to log2n-1 do //pętla sekwencyjna w kernelu
    forall k in parallel do //równoległa praca wątków
        if k ≥ 2^d then x[k] := x[k – 2^d] + x[k]
```

   Zastanów się ile iteracji pętli było wykonywanych w zad. 2 a ile w 3.
5. Zmodyfikuj zadanie 3 tak, aby algorytm mógł działać na większych tablicach niż tylko wielkość bloku (ale również o rozmiarze będącym potęgą 2). Algorytm ma działać na każdym bloku niezależnie. Zakładając, że blok ma 4 wątki, to dla tablicy 1,2,3,4|2,3,4,5|3,4,5,6|4,5,6,7 wynik powinien wynosić: 1,3,6,10|2,5,9,14|3,7,12,18|4,9,15,22.
6. Uzupełnij zadanie 4 w następujący sposób. Po obliczeniu wyników z zad 4, przepisz do tablicy wejściowej ostatnie wartości wyliczone przez każdy blok. Nawiązując do wyniku zadania 4, wynikowa tablica zawierałaby 10,14,18,22. Użyj tej tablicy jako dane wejściowe dla kernela z zad 4. Dla przykładowych danych wejściowych obliczone zostanie 10,24,32,54.
7. Załóżmy, że w zadaniu 5 obliczono n wartości. Dodaj pierwsze $n-1$ wartości do wyników obliczonych w zadaniu 4 do $n-1$ ostatnich bloków. Przykładowo, dodaj wartości 10,24 i 32 do wyników bloków 1,2 i 3, czyli: 1,3,6,10 (bez zmian)|2,5,9,14 (+10)|3,7,12,18(+24)|4,9,15,22(+32). Otrzymana tablica stanowi wynik operacji scan.

---

8. Dana jest tablica z liczbami z zadanego (i znanego) przedziału. Wykorzystując instrukcje atomowe napisz program obliczający liczbę wystąpień każdej z liczb w tej tablicy. Przykładowo, dla tablicy zawierającej liczby: 0,1,3,2,5,0,0,4,3,2,1,1,5,4 (z przedziału od 0-5), wynikiem powinna być tablica o sześciu elementach, kolejno 3,3,2,2,2,2 (kolejno: trzy zera, trzy jedynki, dwie dwójki itd.).

9. Rozwiąż problem podobnie jak w 8, ale każdy blok powinien mieć histogram budowany lokalnie w pamięci współdzielonej, a dopiero potem częściowe sumy powinny być dodawane do liczników w pamięci globalnej. Zastanów się które rozwiązanie jest lepsze.
10. Napisz program przesuwający każdy element tablicy o jedną pozycję na prawo (ostatni element trafia na pierwszą pozycję). Różnica w stosunku do podobnego zadania rozwiązywanego wcześniej jest taka, że program powinien działać *in-place*, tzn. bez użycia dodatkowej tablicy w pamięci globalnej. Dopuszczalne jest natomiast wielokrotne uruchamianie różnych kerneli i ewentualne użycie pamięci współdzielonej.
    Zastanów się samodzielnie nad algorytmem. Jeżeli nie dasz rady, to przeczytaj poniżej:
    Algorytm jakiego należy tutaj użyć jest następujący:
    a. Napisz kernel który wczytuje do pamięci współdzielonej kolejne wartości z tablicy wejściowej z zadanym skokiem, wykonuje przesunięcie („zawinięcie" występuje w ramach bloku) i zapisuje z powrotem do tablicy wejściowej.
    b. Wykonaj ten kernel ze skokiem 1. Zapamiętaj liczbę użytych wątków w ramach bloku.
    c. Wykonaj ten kernel ze skokiem równym liczbie użytych poprzednio wątków w bloku.
    d. Wykonuj ten kernel za każdym razem mnożąc poprzedni skok razy liczbę wątków w bloku użytą poprzednio. Kiedy skok przekroczy rozmiar tablicy wejściowej to całość tablicy jest przesunięta.

    Przykład:

Początek	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1 wyk.	4	1	2	3	8	5	6	7	12	9	10	11	16	13	14	15
2 wyk.	16	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

---

11. Zaimplementuj atomową operację potęgowania liczb. Wykorzystaj atomicCAS.
12. Przypomnij sobie przykład na użycie bariery dostępu do pamięci z wykładu. Spróbuj rozwiązać zadanie 4 w ramach jednego uruchomienia kernela. Postępuj następująco:
    a. Każdy blok wykonuje przy użyciu pamięci współdzielonej przesunięcie ze skokiem 1 (zawinięcie w ramach bloku).
    b. Ostatni blok który to wykona (sprawdzenie, że blok jest ostatni wzorujemy na przykładzie bariery dostępu do pamięci) wykonuje w pętli „korektę" wartości granicznych. Uwaga! Wszystkie wątki w ramach bloku powinny pracować.
13. Napisz program rozwiązujący problem komiwojażera. Strukturę grafu zaimplementuj w postaci macierzy sąsiedztwa z wagami. Część równoległa programu powinna obliczać długości wszystkich ścieżek w grafie. Część sekwencyjna wybierać najkrótszą ścieżkę. Nie jest istotna optymalizacja kodu.
14. Napisz program rozwiązujący problem plecakowy. Część równoległa powinna znajdować sumaryczną wartość i wagę dla wszystkich możliwych pakowań plecaka. Część sekwencyjna wybierać największą wartość wśród dopuszczalnych wag. Nie jest istotna optymalizacja kodu. Można zastosować wielokrotne uruchamianie kernela dla różnych liczności podzbiorów pakowanych elementów.