Dugăeșescu Andrei 331CC

Tema 3 ASC - CUDA: Parallel Hashtable

În cadrul soluției propuse este descrisă implementarea unui hashtable aflat pe GPU reprezentat sub forma unui array de perechi de tipul cheie:valoare. Pentru reținerea acestor perechi a fost folosită o structură de tipul enrty_t responsabilă pentru reținerea de două numere întregi reprezentând cheia (key) și valoare (value) pentru o pereche - un entry în hashtable. Printre funcționalitățile acestui hashtable se numără: crearea hashtable-ului, distrugerea acestuia, posibilitatea inserării unui număr oarecare de perechi de tipul cheie:valoare, interogarea perechilor din cadrul hashtable-ului, precum și redimensionarea hashtable-ului.

Inserarea unor elemente în cadrul hashtable-ului este realizată în felul următor: inițial se verifică faptul ca elementele ar avea loc în interiorul hashtable-ului; ținând cont de faptul că este cunoscută încărcarea curentă, precum și spațiul total alocat, se poate determina dacă un anumit număr de elemente au sau nu loc în hashtable-ul curent. În cazul în care spațiul alocat se dovedește a fi insuficient sunt realizate următoarele operații: se alocă suficient spațiu pentru noile elemente într-un nou hashtable, se inițializează cheile și valorile din noul hashtable cu o serie de valori implicite după care fiecare element din noul hashtable este trecut printr-un proces de rehash înainte de a fi adăugat în noul hashtable. După ce toate elementele au fost transferate cu succes este eliberată memoria din vechiul hashtable. Tratarea coliziunilor se realizează în felul următor: pentru o anumită cheie dată se calculează indicele corespunzător acesteia cu ajutorul funcției de hash. Odată calculat acest indice este verificată în mod atomic cheia aflată la acel indice. Dacă acea cheie are valoarea 0 (EMPTY_KEY) sau are chiar valoarea cheii primite (este vorba de o actualizare a valorii curente) atunci valorea corespunzătoare cheii primite este inserată la acel indice din hashtable. Alternativ, în cazul în care la indicele respectiv se găsește deja o alte pereche de tipul cheie:valoare, atunci se caută liniar, în continuare, un slot liber pentru a putea insera datele primite. În cazul în care spațiul din hashtable este suficient atunci se aplică direct logica de inserție descrisă mai sus în hashtable-ul curent. O diferență între inserția din cadrul redimensionării hashtable-ului și inserția normală este reprezentată de contorizarea numărului de elemente noi adăugate. La redimensionarea hashtable-ului, numărul total de elemente nu se modifică, ci doar dimensiunea totală a hashtable-ului. În cazul inserării unor elemente (noi) atunci există două posibilități pentru fiecare pereche. Dacă la inserare o cheie dată este regăsită în cadrul hashtable-ului, atunci acea operație de inserție reprezintă defapt actualizarea valorii unei chei, deci numărul total de elemente rămâne același; dacă pentru o cheie dată este inserată la un indice la cărui cheie inițial este 0 (EMPTY_KEY) atunci se poate incrementa contorul întrucât în acest caz este vorba de adăugarea unei noi perechi în hashtable.

În ceea ce privește interogarea valorilor din hashtable, acest lucru se face în felul următor: pentru o anumită cheie primită se calculeaza de asemenea indicele corespunzător prin intermediul funcției de hash. Odată găsit acest indice este interogată valoarea cheii de la acel indice. Dacă acea cheie găsită este aceeași cu cheia pentru care se caută valoarea, atunci se returnează valoarea de la acel indice, altfel se continuă căutarea cu următoare poziție, liniar. Dacă după parcurgerea tuturor elementelor din hashtable nu a fost găsită cheia căutată, atunci se returneaza 0 (EMPTY_VALUE).

//Conectare coadă hp

qlogin -q hp-sl.q

//Adăugare modul cuda

module load libraries/cuda

În continuare se vor discuta performanțele soluției propuse precum atât pe baza metodei de implementare cât și pe baza unor rezultate obținute în urma rulării unei serii de teste. În primul rând să observăm throughput-ul soluției pentru câteva cazuri de test:

Insert	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6
Test 1	100	inf	100	inf	100	inf
Test 2	inf	inf	200	66.6667	100	inf
Test 3	66.6667	100	inf	100	inf	inf
	66.6667	50	inf	66.6667	inf	inf
Test 4	62.5	50	62.5	50	62.5	50
	50	41.6667	35.7143	50	41.6667	41.6667
	41.6667	35.7143	35.7143	35.7143	31.25	31.25
	35.7143	35.7143	27.7778	41.6667	31.25	27.7778

Get	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6
Test 1	inf	inf	100	100	100	inf
Test 2	inf	66.6667	200	100	200	100
Test 3	200	100	inf	200	inf	inf
	200	100	inf	100	inf	inf
Test 4	125	62.5	125	125	125	125
	83.3333	125	83.3333	83.3333	125	125
	83.3333	125	125	125	250	62.5
	83.3333	125	62.5	83.3333	125	125

În tabelele de mai sus se pot observa valorile de throughput obținute pentru 6 rulări diferite ale cazurilor de test. Test 1: - presupune inserarea și interogarea a 1 milion de elemente, Test 2: - similar pentru 2 milioane de elemte, Test 3: - 4 milioane de elemnte, iar Test 4: - 10 milioane de elemente. Fiecare linie reprezintă througput-ul pentru o operație de insert pentru fiecare rulare – inf presupune un throughput foarte mare. După cum se poate observa, în general, inserarea elementelor în hashtable se dovedește a fi mai lentă decât interogarea acestora. Acest lucru se poate întampla din mai multe motive. Primul și probabil cel mai important este faptul că inserarea de noi elemente poate presupune redimensionarea hashtable-ului, operație constsitoare atât din punct de vedere al memoriei folosite dar și al timpului de execuție întrucât este necesară introducereaa vechilor date în noul hashtable înainte de a trece la inserția noilor date. De asemenea inserția presupune folosirea unor operații atomice pentru a asigura integritatea și corectitudinea datelor, ceea ce poate conduce din nou la creșterea timpului de execuție. În cazul citirii datelor din hashtable, acest lucru nu presupune modificarea conținutului hashtable-ului deci gradul de paralelizare este mai mare.

Un alt lucru important care poate fi observat este faptul că throughput-ul scade la inserări consecutive dar și de la un test la următorul. Acest fapt se datorează în primul rând nevoii de a prelucra o cantitate mai mare de date, deoarce pentru redimensionarea hashmap-ului la interogări succesive este necesară introducerea tot mai multor date. De asemenea, testele devin tot mai mari și mai solicitante, ceea ce conduce la scăderea (generală) a throughput-ului programului. Un fenomen similar nu este vizibil în cazul operațiilor de interogare a hashtable-ului întrucât gradul de paralelizare este suficent pentru a avea timpi similari de la o interogare la alta.

Pentru testele de mai sus au fost măsurați timpii necesari efectuării fiecărei operații de insert și get. Rezultatele sunt următoarele:

Test 1 – 1000000 elements

Inserted 1000000 elements in 5.522208 ms (181.086976 million keys/second)
Got 1000000 elements in 2.956160 ms (338.276671 million keys/second)

Test 2 – 2000000 elements

Inserted 2000000 elements in 12.004896 ms (166.598693 million keys/second)
Got 2000000 elements in 5.893408 ms (339.362225 million keys/second)

Test 3 – 4000000 elements

Inserted 2000000 elements in 12.239008 ms (163.411934 million keys/second)
Inserted 2000000 elements in 19.431776 ms (102.924203 million keys/second)

Got 2000000 elements in 6.038624 ms (331.201288 million keys/second)
Got 2000000 elements in 5.849440 ms (341.913065 million keys/second)

Test 4 – 10000000 elements

Inserted 2500000 elements in 15.360608 ms (162.753977 million keys/second)
Inserted 2500000 elements in 24.766657 ms (100.942164 million keys/second)
Inserted 2500000 elements in 32.238014 ms (77.548201 million keys/second)
Inserted 2500000 elements in 38.189537 ms (65.462955 million keys/second)

Got 2500000 elements in 7.387072 ms (338.429068 million keys/second)
Got 2500000 elements in 7.312960 ms (341.858837 million keys/second)
Got 2500000 elements in 7.461536 ms (335.051668 million keys/second)
Got 2500000 elements in 7.512960 ms (332.758322 million keys/second)

Pentru rularea acestor teste a fost realizat un script care va înregistra timpii necesari acetor opreații pentru ficare dintre cele 4 teste întru-un fișier results.txt. Pentru a furniza rezultate, în cadrul programului este necesară setarea simbolului PERFORMANCE_VIEW pe 1, alternativ programul va rula în mod normal.

Testul 4 este probabil cel mai adecvat pentru a observa creșterea timpului necesar realizării operațiilor de insert succesive și a scăderii numărului de perechi procesate într-o unitate de timp (secundă). De asemenea, se mai poate observa faptul că timpul necesar interogării hashtable-ului rămâne aproximativ constant. Ușoara creștere a acestuia se datorează necesității de ca căuta o cheie într-o mulțime mai mare de chei posibile însă această creștere este foarte mică în raport cu numărul de elemente introduse – mai puțin de 0.1 ms pentru fiecare 2500000 elemente, ceea ce conduce către ideea că soluția propusă scalează foarte bine pentru operațiile de interogare a hashtable-ului.

Pentru a obține performanțe mai bune pentru inserări, o primă soluție ar reprezenta minimizarea numărului de redimensionări necesare. Acest lucru ar presupune desigur un compormis între memoria folosită și timpul de execuție. În cadrul soluției propuse, gradul de încărcare vizat este de 90%, ceea ce înseamnă că există o probabilitate foarte mare ca o noua serie de inserări să conducă la nevoia redimensionării hashtable-ului și astfel la diminuarea throughput-ului.