UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Klemen Pravdič

Množenje redkih matrik na arhitekturi CUDA

DIPLOMSKO DELO NA
UNIVERZITETNEM ŠTUDIJU

Mentor: doc. dr. Tomaž Dobravec

Ljubljana, 2013

Univerza *v Ljubljani*
Fakulteta *za računalništvo in informatiko*

Tržaška 25
1001 Ljubljana, Slovenija
telefon: 01 476 84 11
01 476 83 87
faks: 01 426 46 47
01 476 87 11
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 01891/2013
Datum: 07.01.2013

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **KLEMEN PRAVDIČ**

Naslov: **MNOŽENJE REDKIH MATRIK NA ARHITEKTURI CUDA**
SPARSE MATRIX MULTIPLICATION ON CUDA

Vrsta naloge: Diplomsko delo univerzitetnega študija

Tematika naloge:

Predstavite pojem redke matrike ter opišite CSR, CSC in COO načine zapisa redkih matrik v računalniku. Zapise med seboj primerjajte glede na porabo pomnilnika.
Opišite vrstično-stolpični ter vrstično-vrstični algoritem za množenje redkih matrik. Oba algoritma predstavite v zaporedni izvedbi, ki je primerna za izvajanje na CPE, ter v vzporedni izvedbi, ki je primerna za izvajanje na GPE. Vse predstavljene algoritme implementirajte, pri tem za GPE implementacijo uporabite arhitekturo CUDA.
S pomočjo različnih testov primerjajte hitrosti izvajanja vseh predstavljenih algoritmov. Za vhodne matrike uporabite tipične predstavnike redkih matrik iz znanih zbirk (recimo Matrix Market) ter množico sistematično ustvarjenih matrik različnih razsežnosti in gostot. Predstavite rezultate testiranja in ugotovite, pri kakšnih pogojih je izvajanje na GPE hitrejše od izvajanja na CPE.

Mentor:

doc. dr. Tomaž Dobravec

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Dekan:

prof. dr. Nikolaj Zimic

I Z J A V A O A V T O R S T V U

diplomskega dela

Spodaj podpisani Klemen Pravdič,

z vpisno številko 63060272,

sem avtor diplomskega dela z naslovom:

Množenje redkih matrik na arhitekturi CUDA

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Tomaža Dobravca,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki ''Dela FRI''.

V Ljubljani, dne 28.6.2013 Podpis avtorja:

Zahvala

Iskreno se zahvaljujem mentorju doc. dr. Tomažu Dobravcu za trud, dosegljivost in strokovno pomoč pri izdelavi diplomske naloge.

Zahvalil pa bi se tudi družini za vsestransko podporo in spodbudo pri študiju.

Kazalo

Seznam uporabljenih kratic in simbolov

GPE – grafična procesna enota (angl. GPU – graphics processing unit)

CPE – centralna procesna enota (angl. CPU – central processing unit)

nnz – število neničelnih elementov (angl. number of non-zeros)

snop – skupina (trenutno 32-ih) implicitno sinhroniziranih niti na GPE (angl. warp)

ščepec – funkcija, ki jo vzporedno izvajajo niti na GPE (angl. kernel)

$A(:, j)$ – MATLAB notacija za j-ti stolpec matrike A

$A(i, :)$ – MATLAB notacija za i-ti vrstico matrike A

$A(i, j)$ – MATLAB notacija za element na preseku i-te vrstice in j-tega stolpca matrike A

Povzetek

Množenje redkih matrik je pogosta operacija v linearni algebri in pomemben sestavni del algoritmov. Redka matrika pa je matrika, ki vsebuje večino elementov z vrednostjo 0. Diplomsko delo opisuje dva algoritma za množenje redkih matrik, vrstično-stolpični algoritem in vrstično-vrstični (imenovan tudi vrstični) algoritem. Za oba algoritma smo opisali zaporedno izvedbo, primerno za izvajanje na CPE, ter vzporedno izvedbo, primerno za izvajanje na GPE. Vse izvedbe algoritmov smo implementirali v programskem jeziku C, za vzporedno implementacijo pa smo uporabili GPE arhitekturo CUDA. V algoritmih smo uporabljali različne zapise redkih matrik (CSR, CSC, COO), katere smo v diplomi tudi opisali. V okviru potreb razumevanja diplomske naloge smo opisali tudi arhitekturo CUDA.

Algoritmom smo merili čas izvajanja ter jih med seboj primerjali. Kot vhodne testne matrike smo uporabili redke matrike iz zbirke Matrix Market in tudi redke matrike, ki smo jih ustvarili sami ter jim določili gostoto in razsežnost. Produkt množenja matrik smo pri vzporednih algoritmih na arhitekturi CUDA zapisovali kot redko in gosto matriko. Ugotovili smo, da je vrstični algoritem hitrejši od vrstično-stolpčnega, vzporedna izvedba pa pod določenimi pogoji deluje hitreje od zaporedne izvedbe vrstičnega algoritma. Uspešnost vzporedne izvedbe je odvisna od gostote in razsežnosti vhodnih matrik. Izkaže se, da je za uspešno izvajanje vzporednega vrstičnega algoritma pri manjših razsežnostih potrebna višja gostota vhodnih matrik in obratno. Vzporedni vrstični algoritem na arhitekturi CUDA deluje najučinkoviteje, ko uporabimo pristop z implicitno sinhroniziranimi nitmi – snopi.

Ključne besede:

Množenje redkih matrik, redka matrika, GPE procesiranje, CUDA

Abstract

Sparse matrix multiplication is a common operation in linear algebra and an important element of other algorithms. Sparse matrix is a matrix populated primarily with zeros. This thesis presents two algorithms for sparse matrix multiplication, row-column algorithm and row-row (also known as row-wise) algorithm. It describes sequential implementation on CPU and parallel implementation on GPU for both algorithms. Algorithms were implemented in C programming language. For parallel implementation we used GPU with CUDA architecture. We described different formats of storage for sparse matrices (CSR, CSC, and COO) that are used in implementation of algorithms. For the purpose of understanding parallel implementations of algorithms CUDA architecture is described.

Timings for all implementations were measured and compared against each other. For testing purposes we used sparse matrices from Matrix Market repository along with sparse matrices with different densities and dimensions that we generated ourselves. On GPU we stored product as both, a sparse and a dense matrix. We determined that row-row algorithm is faster than row-column algorithm and under certain conditions parallel implementation outperforms sequential implementation of a row-row algorithm. Performance of parallel row-row algorithm depends on density and dimensions of input matrices; for efficient performance input matrices with smaller dimensions should be denser. Row-row algorithm on CUDA performs better when groups of implicitly synchronized threads (warps) are used.

Key words:

Sparse matrix multiplication, sparse matrix, GPU processing, CUDA

Poglavje 1

1 Uvod

Pri reševanju računalniških oz. matematičnih problemov se pogosto srečamo z matriko kot podatkovno strukturo. Poseben primer matrike je redka matrika, katere lastnost je, da ima večino vrednosti enakih nič. Za redke matrike zato obstajajo posebni algoritmi, ki za bolj učinkovito izvajanje izrabljajo prav to lastnost in so učinkovitejši od algoritmov za navadne – goste matrike. Pri tem igra pomembno vlogo izbira strukture oz. zapisa hranjenja redke matrike.

Ena izmed pogostih operacij nad matrikami je množenje dveh matrik. V primeru, da so operandi redke matrike, pa lahko s primernimi postopki množenje izvedemo hitreje od navadnega množenja matrik. Množenje redkih matrik je osnovna operacija v linearni algebri in eden ključnih gradnikov zmogljivih algoritmov nad grafi.

Operacijo pa lahko poskusimo izboljšati še tako, da jo prilagodimo za vzporedno izvajanje. Grafične procesne enote se že nekaj časa ne uporabljajo le za procesiranje računalniške grafike, temveč so zaradi velike računske zmogljivosti primerne za splošno namensko vzporedno procesiranje. Vzporedni sistemi z visoko računsko zmogljivostjo so z razvojem grafičnih procesnih enot in njihovo več-jedrno arhitekturo postali razširjeni in cenovno bolj dosegljivi kot kadarkoli prej. Ena izmed takšnih arhitektur za grafične procesne enote je CUDA, katero bomo uporabili v diplomskem delu.

V diplomski nalogi bomo predstavili in implementirali dva algoritma, ki uporabljata različen pristop k množenju redkih matrik ter ju prilagodili za vzporedno izvajanje na GPE z arhitekturo CUDA. Vse implementacije bomo med seboj tudi primerjali glede na čas množenja redkih matrik različnih razsežnosti in redkosti, ter raziskali, pod kakšnimi pogoji se uporaba GPE izplača. Za potrebe razumevanja diplomskega dela bomo predstavili tudi pojem redke matrike, njenih zapisov ter arhitekture CUDA.

Poglavje 2

2 Redke matrike

Matrika je pravokotna struktura z elementi, navadno številčnimi vrednostmi, urejenimi v vrstice in stolpce. Redka matrika pa je definirana kot matrika, ki vsebuje večino ničelnih elementov oz. elementov z vrednostjo nič [4]. Preostalim elementom pravimo tudi neničelni elementi. Rečemo lahko, da gre za matriko, ki je do določene stopnje »prazna«. V splošnem poznamo strukturirane in nestrukturirane redke matrike. Pri strukturiranih matrikah so neničelni elementi razporejeni po matriki v določenem vzorcu, pri nestrukturiranih matrikah pa naključno. Redka matrika se kot podatkovna struktura pogosto uporablja pri operacijah z grafi, saj lahko vsak graf opišemo z matriko [2].

Primer

Primer redke matrike A z razsežnostmi 4×4:

$$A = \begin{bmatrix} 1 & 0 & 2 & 3 \\ 0 & 4 & 0 & 0 \\ 5 & 0 & 6 & 0 \\ 0 & 0 & 0 & 7 \end{bmatrix}$$

2.1 Zapis redke matrike

Za izvajanje algoritmov nad redkimi matrikami moramo le-te hraniti v primerni podatkovni strukturi. Za hranjenje redke matrike obstajajo številni zapisi, vsak s svojimi lastnostmi glede velikosti v pomnilniku in dostopnosti elementov. Glavna ideja shranjevanja redkih matrik je, da v podatkovno strukturo shranimo le relativno majhno število neničelnih elementov, ničelnih vrednosti pa ne zapisujemo. Prostorska zahtevnost običajnega zapisa matrike razsežnosti $n \times n$ je namreč kar n^2. Na velikost strukture redke matrike pa naj bi tako vplivalo le število neničelnih elementov, ne pa tudi razsežnosti matrike. Različni zapisi ponujajo različne možnosti dostopanja do posameznih elementov v podatkovno strukturi. Podatkovna struktura, ki ustreza nekemu algoritmu, je lahko popolnoma neprimerna za nek drug. Na izbiro zapisa lahko vpliva tudi sama strukturiranost redke matrike. Ker se diplomsko delo dotika splošnega množenja redkih matrik z nestrukturiranimi matrikami, bomo v nadaljevanju poglavja predstavili tri splošne zapise, ki smo jih uporabili v implementiranih algoritmih. Lastnosti zapisov so povzete po [6, 10].

2.1.1 CSR zapis

CSR zapis (angl. Compressed Sparse Row) je poleg CSC zelo pogosto uporabljen zapis za redke matrike. Na CSR lahko gledamo kot na polje zaporednih vrstic matrike, ki vsebuje samo neničelne vrednosti, ničelne vrednosti pa so izpuščene. Podatkovno strukturo redke matrike razsežnosti $m \times n$ v zapisu CSR sestavljajo 3 polja:

- polje `val` dolžine nnz z neničelnimi vrednostmi,
- polje `col` dolžine nnz z indeksi stolpcev neničelnih vrednosti in

- polje `ptr` dolžine $m + 1$ s kazalci na prve elemente posameznih vrstic v polju vrednosti. Zadnji element po dogovoru vsebuje število nnz. Če je i-ta vrstica v celoti prazna pa velja `ptr[i]=ptr[i+1]`.

Vrednost in indeks stolpca posameznega elementa se nahajata v poljih `val` in `col` pod istim indeksom. Zapis je nadgradnja COO zapisa in omogoča neposreden dostop do neničelnih elementov i-te vrstice ter enostaven izračun števila neničelnih elementov v posamezni vrstici. Elementi znotraj posameznih vrstic niso nujno urejeni tudi po stolpcih. Ta zapis je uporabljen tudi v izvirni predstavitvi vrstičnega algoritma za množenje redkih matrik, ki ga obravnavamo tudi v diplomskem delu. Zaradi neposrednega dostopa do poljubne vrstice je zapis primeren za vrstični algoritem za množenje redkih matrik, kjer dostopamo do vrstic obeh vhodnih matrik. Za hranjenje matrike v tem zapis potrebujemo le $2nnz + m + 1$ mest v podatkovni strukturi, med tem ko jih potrebujemo v običajnem zapisu n^2.

Primer

Matrika A, predstavljena v CSR zapisu:

val = [1 2 3 4 5 6 7]

col = [0 2 3 1 0 2 3]

ptr = [0 3 4 6 7]

2.1.2 CSC zapis

Zapis CSC (angl. Compressed Sparse Column) je zelo podoben CSR zapisu, le da so tukaj v skrčeni obliki zapisani indeksi stolpcev elementov, elementi pa si v strukturi sledijo kot v zaporednih stolpcih matrike. Zapis smo uporabili pri vrstično-stolpični metodi za množenje redkih matrik, kjer potrebujemo pri drugi vhodni matriki neposreden dostop do želenega stolpca. Za hranjenje matrike razsežnosti $m \times n$ v zapisu CSC potrebujemo $2nnz + n + 1$ mest, podatkovno strukturo pa, podobno kot CSR, sestavljajo 3 polja:

- polje `val` dolžine nnz z neničelnimi vrednostmi,
- polje `row` dolžine nnz z indeksi vrstic neničelnih vrednosti,
- polje `ptr` dolžine $n + 1$ s kazalci na prve elemente stolpca v polju vrednosti. Kot pri CSR, tudi tukaj zadnje polje vsebuje število neničelnih elementov.

Primer

Matrika A, predstavljena v CSC zapisu:

val = [1 5 4 2 6 3 7]

row = [0 2 1 0 2 0 3]

ptr = [0 2 3 5 7]

2.1.3 Koordinatni ali COO zapis

Koordinatni zapis je enostaven splošen način predstavitve oz. shranjevanja redkih matrik. Vsak od nič različen element matrike razsežnosti $m \times n$ je predstavljen s trojico vrednosti: številčno vrednostjo, indeksom vrstice in indeksom stolpca elementa v matriki. Podatkovna struktura tako vsebuje 3 polja:

- polje `val` dolžine nnz z vrednostmi neničelnih elementov,
- polje `col` dolžine nnz z indeksi stolpcev neničelnih elementov ter
- polje `row` dolžine nnz z indeksi vrstic neničelnih elementov.

Vrednost, indeks vrstice in indeks stolpca posameznega elementa se nahajajo v za to namenskih poljih pod enakim indeksom. Elementi so lahko v zapisu še dodatno urejeni po vrsticah in/ali stolpcih, ni pa to pogoj in je implementacija odvisna od aplikacije. Prednost takšne predstavitve je predvsem v enostavnosti ter v neposredno izraženih indeksih, ki jih ni potrebno računati. Slabost zapisa pa je, da ne moremo neposredno dostopati do želenih vrstic oz. stolpcev, četudi je struktura urejena. Začetke vrstic ali stolpcev moramo iskati. Za hranjenje matrike v tem zapisu potrebujemo $3nnz$ mest.

Primer

Matrika A, predstavljena v COO zapisu:

```
val = [1 2 3 4 5 6 7]

col = [0 2 3 1 0 2 3]

row = [0 0 0 1 2 2 3]
```

Poglavje 3

3 Množenje redkih matrik

Množenje redkih matrik je ena od osnovnih in ključnih operacij v numerični linearni algebri in zelo pogosta in pomembna operacija pri raznih algoritmih. Vsak graf lahko namreč zapišemo kot matriko in obratno, kar pomeni, da lahko operacije iz linearne algebre uporabljamo kot sestavne dele algoritmov nad grafi [2]. V tem poglavju bomo opisali delovanje in implementacijo dveh algoritmov z različnima pristopoma za množenje splošnih oz. nestrukturiranih redkih matrik, kasneje pa ju bomo prilagodili tudi za vzporedno izvajanje.

Preden nadaljujemo z množenjem redkih matrik si najprej poglejmo klasično množenje matrik. Naj bo A matrika razsežnosti $m \times k$ in B matrika razsežnosti $k \times n$. Produkt matrik A in B je matrika C razsežnosti $m \times n$, katere elementi so definirani kot:

$$C(i,j) = \sum_{l=1}^{k} A(i,l) * B(l,j), \ 1 \leq i \leq m, 1 \leq j \leq n \quad (1)$$

Element na preseku i-te vrstice in j-tega stolpca produkta C je enak skalarnemu produktu i-te vrstice matrike A in j-tega stolpca matrike B [5]. Množenje matrik kot tako zahteva $m * k * n$ delnih produktov $A(i,l) * B(l,j)$, ki jih lahko razporedimo v štiri razrede (povzeto po [1]):

1. $A(i,l) = 0$ in $B(l,j) = 0$
2. $A(i,l) = 0$ in $B(l,j) \neq 0$
3. $A(i,l) \neq 0$ in $B(l,j) = 0$
4. $A(i,l) \neq 0$ in $B(l,j) \neq 0$

Rezultat delnih produktov prvih treh razredov je 0, saj je vsaj eden od operandov, bodisi $A(i,l)$ iz matrike A bodisi $B(l,j)$ iz matrike B, enak 0. Če sta matriki A in B redki, bo največ delnih produktov pripadalo razredom 1, 2, 3, najmanj pa jih bo v 4. razredu. Le delni produkti iz 4. razreda pa vplivajo na končni produkt, matriko C. Za učinkovito množenje redkih matrik želimo uporabiti algoritem s številom operacij N, sorazmernim številu neničelnih delnih produktov. Zanj velja:

$$0 \leq N \leq m * k * n \quad (2)$$

Za množenje redkih matrik torej želimo algoritem, ki je časovno odvisen od števila operacij nad neničelnimi elementi. V diplomskem delu bomo predstavili in implementirali dva algoritma za množenje redkih matrik, nato pa ju prilagodili za vzporedno izvajanje na GPE. Algoritma izvajata množenje redkih matrik, kjer delni produkti vključujejo le neničelne elemente. V nadaljevanju poglavja sledi opis zaporednih različic obeh omenjenih algoritmov.

3.1 Vrstično-stolpični algoritem za množenje redkih matrik

Vrstično-stolpični algoritem [9] za množenje redkih matrik je algoritem za množenje redkih matrik, ki spominja na klasični algoritem za množenje matrik, pri tem pa izkorišča dejstvo, da uporabljamo redke matrike.

$$C(i,j) = \sum_{l=1}^{k} A(i,l) * B(l,j) \text{ za } A(i,l) \neq 0, B(l,j) \neq 0 \quad (3)$$

Matematični opis (3) je podoben klasičnemu množenju gostih matrik (1); pri množenju matrike A razsežnosti $m \times k$ in matrike B razsežnosti $k \times n$, je element na preseku i-te vrstice in j-tega stolpca produkta C enak skalarnemu produktu i-te vrstice matrike A in j-tega stolpca matrike B, pri čemer zadošča, da v delnih produktih nastopajo le neničelni elementi.

```
foreach C(i,j) in C
      vrstica_A= A(i,:);
      stolpec_B= B(:,j);
      A(i,k)= naslednji_nz(vrstica_A);
      B(l,j)= naslednji_nz(stolpec_B);
      while(A(i,k)!= null && B(i,k) != null)
            if(k=l)
                  vsota  = vsota + A(i,k) * B(l,j);
                  A(i,k) = naslednji_nz(vrstica_A);
                  B(l,j) = naslednji_nz(stolpec_B);
            else if(k<l)
                  A(i,k) = naslednji_nz(vrstica_A);
            else
                  B(l,j) = naslednji_nz(stolpec_B);
      if(vsota!==0)
            C(i,j)=vsota;
```

Psevdokoda ideje: funkcija naslednji_nz() vrne naslednji neničelni element vrstice oz. stolpca.

Algoritem lahko s pomočjo psevdo kode opišemo v spodaj naštetih točkah:

- Po vrsti se premikamo skozi elemente izhodne matrike C.
- Za trenuten element $C(i,j)$ izberemo i-to vrstico matrike A in j-ti stolpec matrike B.
- V zanki se sprehajamo skozi elemente izbrane vrstice in stolpca. Med seboj pomnožimo elemente vrstice A in stolpca B z enakim indeksom v vrstici oz. stolpcu in delni produkt prištejemo k vsoti.
- Neničelen rezultat kot trojico `(i,j,vsota)` zapišemo v strukturo matrike C in se premaknemo na naslednji element $C(i,j)$.

Za učinkovito izvajanje algoritma potrebujemo dostop do želene vrstice matrike A ter želenega stolpca matrike B. Zato uporabimo za matriko A zapis CSR in matriko B zapis CSC. Obe matriki sta znotraj vrstic oz. stolpcev urejeni še po stolpičnih oz. vrstičnih indeksih. Produkt oz. matriko C zapišemo v COO zapisu. Elementi bodo v strukturi že urejeni po stolpcih ali vrsticah, odvisno od vrstnega reda, v katerem se pomikamo skozi elemente matrike C.

Primer

Množenje matrik A in B. Rezultat je matrika C.

$$A = \begin{bmatrix} 1 & 0 & 2 & 3 \\ 0 & 4 & 0 & 0 \\ 5 & 0 & 6 & 0 \\ 0 & 0 & 0 & 7 \end{bmatrix} \qquad B = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 3 & 0 \\ 0 & 0 & 4 \\ 5 & 0 & 0 \end{bmatrix}$$

Po vrsti računamo elemente matrike C:
$C(1,1) = [1\ 0\ 2\ 3] * [1\ 2\ 0\ 5] = 1 * 1 + 3 * 5 = 16$
$C(1,2) = [1\ 0\ 2\ 3] * [0\ 3\ 0\ 0] = 0$
$C(1,3) = [1\ 0\ 2\ 3] * [0\ 0\ 4\ 0] = 2 * 4 = 8$
...

$$C = \begin{bmatrix} 16 & 0 & 8 \\ 8 & 12 & 0 \\ 5 & 0 & 24 \\ 35 & 0 & 0 \end{bmatrix}$$

Komentar: Vsak element $C(i,j)$ matrike C računamo kot produkt i-te vrstice A in j-tega stolpca B, med seboj pa množimo le neničelne elemente z enakimi indeksi v vektorju vrstice oz. stolpca.

3.1.1 Implementacija

Celoten postopek programa :

1. Branje datotek z vhodnimi matrikami A in B in hranjenje v pomnilniku v CSR zapisu.
2. Pretvorba matrike B v zapis CSC.
3. Klic funkcije z algoritmom.
4. Zapis v datoteko.

Tako za uporabo CSR zapisa kot tudi za pretvorbo v CSC zapis smo delno uporabili programsko knjižnico, dostopno na [12]. Funkcija z algoritmom sprejme naslednje argumente:

- strukturo za matriko A v CSR zapisu,
- strukturo za matriko B v CSC zapisu,
- kazalec na izhodno strukturo za matriko C v COO zapisu.

Za podatkovno strukturo, ki hrani matriko, smo uporabili strukturo programskega jezika C (`struct`) , ki vsebuje ustrezna polja (v primeru CSR zapisa so to polja `val`, `ptr`, `col`) ter podatke o številu neničelnih elementov, višini in širini matrike (`nnz`, `height`, `width`).

```
void rowColCPU(const CSR A, const CSC B, COO *C) {
  int aColIt,aColEnd,bRowIt,bRowEnd,colIdxA,rowIdxB,count=0;
  float sum;
  for (col = 0; col < B.width; col++){ /* sprehod med elementi produkta */
    for (row = 0; row < A.height; row++){
      aColIt  = A.ptr[row];            /* kazalec na začetek vrstice A */
      aColEnd = A.ptr[row+1];          /* konec vrstice A */
      bRowIt  = B.ptr[col];            /* kazalec na začetek stolpca B */
      bRowEnd = B.ptr[col+1];          /* konec stolpca B */
      sum=0;
      while(aColIt<aColEnd && bRowIt<bRowEnd){
        colIdxA = A.col[aColIt];       /* prebere stolpični indeks vrstice A*/
        rowIdxB = B.row[bRowIt];       /* prebere vrsticni indeks stlpca A*/
        if(colIdxA == rowIdxB){        /* če sta enaka, pomnoži vrednosti */
          sum+=A.val[aColIt]*B.val[bRowIt];
          aColIt++;
          bRowIt++;
        }else if( colIdxA< rowIdxB){
          aColIt++;
        }else{
          bRowIt++;
        }
      }
      if(sum!=0){                      /* shranimo le element različen od 0*/
        if(count>=C->nnz-1)
          extendCOO(C);
        C->val[count]=sum;
        C->row[count]=row;
        C->col[count]=col;
        count++;
      }
    }
  }
  C->nnz=count;
}
```

Izsek kode: funkcija z vrstično-stolpičnim algoritmom za množenje redkih matrik.

Implementacija algoritma spominja na klasično množenje matrik. Z dvema gnezdenima zankama se sprehodimo skozi vse elemente produkta oz. matrike C. Za vsak element matrike C izberemo vrstico matrike A glede na njegov vrstični indeks ter stolpec matrike B glede na njegov stolpični indeks. Programska logika poskrbi, da množimo le neničelne elemente trenutne vrstice A in stolpca B, ki so poravnani po indeksih i in j. Sestavlja jo zanka, ki se sprehodi skozi neničelne elemente trenutne vrstice A in stolpca B. Zanka se ponavlja, dokler kažeta kazalca na veljavna elementa znotraj vrstice A oz. stolpca B. Tedaj preberemo njuni vrednosti in indeksa stolpca oz. vrstice. Pogoj za množenje elementov je, da je indeks stolpca elementa matrike A enak indeksu vrstice elementa matrike B. Produkt se prišteje k delni vsoti. Nato se v obeh matrikah premaknemo na naslednji neničelni element. Kadar pa do množenja ne pride, in je npr. indeks elementa iz matrike A višji od indeksa elementa matrike B, se na naslednji element pomaknemo le v B in obratno.

Ker je produkt redka matrika, katere velikosti vnaprej ne poznamo, izračunane elemente matrike C shranjujemo v COO strukturo, katere velikost po potrebi dinamično povečujemo. Elementi so shranjeni v enakem vrstnem redu kot so izračunani, zato potrebe po urejanju ni.

3.2 Vrstični algoritem za množenje redkih matrik

Vrstični algoritem (tudi vrstično-vrstični) za množenje redkih matrik je leta 1978 predstavil Gustavson. Algoritem uporablja drugačen pristop kot prej predstavljen algoritem. Splošno formulo za množenje *(1)* lahko zapišemo kot:

$$C(i,:) = \sum_{a(i,l)\neq 0} A(i,l) * B(l,:) \quad (4)$$

Algoritem po korakih računa vrstice matrike produkta C. Iz enačbe (4) vidimo, da je i-ta vrstica matrike C linearna kombinacija tistih l-tih vrstic B, za katere je element $A(i,l)$ i-te vrstice matrike A različen od nič [1]. Ker sta vhodni matriki shranjeni v CSR zapisu, ki hrani po vrsticah dosegljive neničelne elemente, vsi delni produkti spadajo v 4. razred (glej zgoraj), kar pomeni, da množimo le neničelne elemente in je število operacij množenja proporcionalno neničelnim produktom. V splošnem lahko algoritem za vrstično množenje matrike A razsežnosti $m \times k$ in matrike B razsežnosti $k \times n$ zapišemo kot [6]:

```
for i =1 to m
      for l where A(i,l) != 0
          C(i,:) = C(i,:) + A(i,l) · B(l,:)
```

Naj tukaj omenimo tudi zelo podoben stolpični algoritem za množenje matrik, ki uporablja enak pristop kot vrstični algoritem. V tem primeru računamo po vrsti stolpce matrike C; j-ti stolpec matrike C je v tem primeru izračunan kot linearna kombinacija tistih l-tih stolpcev A, za katere so elementi $B(l,j)$ j-tega stolpca B neničelni. V tem primeru uporabljamo za hranjenje vhodnih matrik CSC zapis, ki omogoča dostop do želenih stolpcev matrik. Ta različica se uporablja tudi v programskem paketu orodij MATLAB. V diplomskem delu pa bomo opisali in implementirali v zaporedni in vzporedni izvedbi prvotno, vrstično različico algoritma, bi pa lahko v primeru stolpične različice uporabljali iste principe pri implementaciji.

Zaradi dostopa do želene vrstice obeh vhodnih matrik je CSR struktura idealna za hranjenje vhodnih podatkov. Algoritem kot vhodne podatke sprejme vhodni matriki A in B razsežnosti $m \times k$ ter $k \times n$ v redkem CSR zapisu. Kot rezultat dobimo matriko C razsežnosti $m \times n$ v CSR zapisu, ki predstavlja produkt vhodnih matrik A in B.

```
foreach vrstica_A= A(i,:) in A
      ponastavi(medpomnilnik); //vrednosti medpomnilnika na 0
      foreach element_A=A(i,l) in vrstica_A
            foreach element_B=B[l,j] in vrstica_B=B(l,:)
                  medpomnilnik[j]+= A(i,l)*B(l,j);
      foreach nz=medpomnilnik[j]!=0 in medpomnilnik
            nz->C(i,j);
```

Psevdokoda ideje vrstičnega algoritma.

Delovanje algoritma lahko poleg psevdokode povzamemo v naslednjih ključnih točkah:

- Za vsako vrstico matrike C se po vrsti sprehajamo skozi vrstice matrike A.
- Vrednosti medpomnilnika ponastavimo na nič.
- Za vsak neničelni element $A(i,l)$ vrstice $A(i,:)$ določimo pripadajočo l-to vrstico matrike B.

- Vsak neničelni element $B(l,j)$ vrstice $B(l,:)$ pomnožimo z $A(i,l)$.
- Delni produkt $A(i,l) * B(l,j)$ prištejemo na j-to mesto v medpomnilnik.
- Ko je trenutna vrstica izračunana, prepišemo neničelne elemente iz medpomnilnika v strukturo matrike C.
- Nadaljujemo z naslednjo vrstico.

Algoritem uporablja za računanje trenutne izhodne vrstice medpomnilnik v velikosti n (širina matrike C). Da pa se izognemo m-kratnemu ponastavljanju medpomnilnika za vsako vrstico matrike C, uporablja algoritem posebno tehniko z dvema poljema dolžine n; poljem celih števil `xb` in poljem realnih števil `x` [1]. Polje `xb` služi kot indikator in ga je potrebno ponastaviti samo na začetku. Polje `x` pa hrani izračunane vrednosti i-te vrstice. Ko izračunamo nov delni produkt $A(i,l) * B(l,j)$, preverimo j-to mesto v polju `xb`. Če se v njem nahaja celo število, ki je različno od številke računane vrstice i, vemo, da na tem indeksu še nismo izračunali delnega produkta in delni produkt vpišemo na j-to mesto v polje `x` in shranimo indeks j v strukturo matrike C. V nasprotnem primeru je delni produkt že zasedel j-to mesto v trenutni vrstici ter moramo slednjega prišteti na j-to mesto polja `x`. Iz njega po končanem računanju vrstice prepišemo vrednosti glede na indekse j, ki smo jih sproti shranjevali v podatkovno strukturo matrike C.

Časovna zahtevnost vrstičnega algoritma s takšnim pristopom je $O(max(m,n,nnz(A),N))$ [1]. Obrazložitev za takšno časovno zahtevnost je sledeča: skozi medpomnilnik dolžine n se moramo vsaj enkrat sprehoditi. Prav tako se sprehodimo skozi vseh m vrstic matrike C. N predstavlja število delnih produktov iz enačbe (2), $nnz(A)$ pa število neničelnih elementov v celotni matriki A. Za vsakega od njih pa preverimo matriko B za pripadajočo vrstico.

Primer

Množenje matrik A in B. Rezultat je matrika C.

$$A = \begin{bmatrix} 1 & 0 & 2 & 3 \\ 0 & 4 & 0 & 0 \\ 5 & 0 & 6 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \qquad B = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 3 & 0 \\ 0 & 0 & 4 \\ 5 & 0 & 0 \end{bmatrix}$$

1. vrstica C:
$C(1,:) = 1 * [1\ 0\ 0] + 2 * [0\ 0\ 4] + 3 * [5\ 0\ 0] = [1\ 0\ 0] + [0\ 0\ 8] + [15\ 0\ 0] = [16\ 0\ 8]$
2. vrstica C:
$C(2,:) = 4 * [2\ 3\ 0] = [8\ 12\ 0]$
3. vrstica C:
$C(3,:) = 5 * [1\ 0\ 0] + 6 * [0\ 0\ 4] = [5\ 0\ 0] + [0\ 0\ 24] = [5\ 0\ 24]$
4. vrstica C:
$C(4,:) = 7 * [5\ 0\ 0] = [35\ 0\ 0]$

$$C = \begin{bmatrix} 16 & 0 & 8 \\ 8 & 12 & 0 \\ 5 & 0 & 24 \\ 35 & 0 & 0 \end{bmatrix}$$

Komentar: Za izračun i-te vrstice C pomnožimo neničelne elemente i-te vrstice A z neničelnimi elementi pripadajočih vrstic B.

3.2.1 Implementacija

Postopek celotnega programa obsega:

1. Branje datotek z vhodnimi matrikami A in B in hranjenje v pomnilniku v CSR zapisu.
2. Klic funkcije z algoritmom za vrstično množenje.
3. Urejanje produkta in zapis v datoteko.

Funkcijo z algoritmom pa smo implementirali s pomočjo psevdokode in opisa iz izvornega besedila o algoritmu [1]. Pri klicu funkcije podamo 3 argumente:

- strukturo matrike A v CSR zapisu,
- strukturo matrike B v CSR zapisu,
- kazalec na strukturo matrike C v CSR zapisu.

Podatkovna struktura za hranjenje matrike je implementirana enako kot pri vrstično-stolpičnem algoritmu.

```
void spmmCPU(const CSR A, const CSR B, CSR *C) {
  /*
    izpuščeno: deklaracija in inicializacija spremenljivk
  */
  for(i=0;i<n;i++)  /*ponastavitev polja širine n, ki sluzi kot indikator*/
        xb[i]=-1;
  for(i=0;i<m;i++){   /*sprehod skozi m vrstic A*/
    C->I[i]=ip;
    for(jp=A.ptr[i];jp<A.ptr[i+1];jp++){ /*sprehod skozi trenutno vrstico A*/
      j=A.col[jp];
      for(kp=B.ptr[j];kp<B.ptr[j+1];kp++){/*sprehod skozi vrstico B*/
        k=B.col[kp];
        if(xb[kd4]!=i){          /*ali je delni produkt prvič na tem indeksu?*/
                                 //DA:
          xb[k]=i;                       /* označi v polju indikatorjev  */
          C->col[ip]=k;                  /* shrani indeks v strukturo C */
          x[k]=A.val[jp]*B.val[kp];      /* izračuna produkt ter ga zapise*/
          ip++;                          /* poveča števec nz */
          if(ip>=C->nnz-1) /* preveri, če je potrebno povečati strukturo C*/
            extend(C);
        }else{                   //NE:
          x[k]+=A.val[jp]*B.val[kp];  /*prišteje delni produkt*/
        }
      }
    }
    for(vp=C->ptr[i];vp<ip;vp++) {     /* v C strukturo prepise še vrednosti */
      C->val[vp]=x[C->col[vp]];
    }
  }
  C->ptr[p]=ip;                        /*zapišemo se končno število nnz*/
  C->nnz=ip;
}
```

Izsek kode: funkcija z vrstičnim algoritmom za množenje redkih matrik.

Algoritem sestavljajo tri gnezdene zanke. Zunanja zanka poskrbi za sprehod skozi vseh m vrstic matrike A. Po vsakem obhodu te zunanje zanke je izračunana vrstica v C. Sredinska zanka se

sprehodi med neničelnimi elementi trenutne vrstice A. Program vstopi v to zanko za vsak neničelen element matrike A, ne glede na to, ali vsebuje pripadajoča vrstica B neničelne elemente. Notranja zanka pa se sprehodi skozi elemente pripadajoče vrstice B glede na trenutni element iz A. Če pripadajoča vrstica ne vsebuje neničelnih elementov, program v notranjo zanko ne vstopi. V notranji zanki se izvede množenje delnih produktov. V njej je vejitev, v kateri pregledamo, ali je mesto v medpomnilniku, kamor zapišemo delni produkt, že bilo zasedeno; v tem primeru delni produkt prištejemo na to mesto. Če pa za to vrstico na to mesto zapisujemo prvič, prejšnjo vrednost prepišemo, mesto označimo v polju indikatorjev, shranimo stolpični indeks novega elementa v strukturo C ter povečamo števec. Na tem mestu tudi preverimo, ali je potrebno strukturo za matriko C povečati. Ko se izteče sredinska zanka, je izračunana celotna trenutna vrstica matrike C in iz medpomnilnika prepišemo neničelne elemente v strukturo C. To storimo s pomočjo števca neničelnih elementov in že shranjenih stolpičnih indeksov.

Izhodna matrika C je po izvajanju urejena po vrsticah, elementi posameznih vrstic pa niso urejeni tudi po stolpičnih indeksih. Pred zapisom v datoteko lahko strukturo uredimo z uporabo vgrajene funkcije iz standardne C knjižnice (urejanje po metodi »`quicksort`« iz knjižnice `stdlib.h`); strukturo uredimo najprej po stolpičnih indeksih, nato vrstičnih indeksih. Takšno obliko namreč uporablja zapis vhodnih datotek.

Poglavje 4

4 CUDA

CUDA (Compute Unified Device Architecture) je splošno namenska visoko-vzporedna računska arhitektura. Uporablja se na zmogljivejših grafičnih procesnih enotah podjetja NVIDIA in zaradi visoke vzporedne narave in velike računske moči omogoča reševanje zahtevnih računskih problemov. Poleg računanja grafičnih podob aplikacij se uporablja tudi za splošno namensko vzporedno računanje. Uporaba CUDE je primerna na številnih področjih: izrisovanje 3D slik, manipulacija slik in videa, fizikalne simulacije, procesiranje signalov, finančno preračunavanje in drugo.

Za razvoj programske opreme za GPE z arhitekturo CUDA so na voljo programska orodja CUDA SDK in razširjena različica visoko nivojskega programskega jezika C. Za prevajanje programov je na voljo prevajalnik `nvcc`. Za poganjanje programov pa je dovolj osebni računalnik, ki vsebuje GPE z arhitekturo CUDA. Poleg programskega jezika C nudi CUDA podporo tudi drugim programskim jezikom.

Visoka računska zmogljivost GPE izhaja iz dejstva, da je večina tranzistorjev na čipu GPE namenjena procesiranju podatkov, manjši del pa je namenjen kontrolni enoti in predpomnilniku (slika 1). Zato je GPE z arhitekturo CUDA pisana na kožo problemom, katerih reševanje zahteva pretežno aritmetične operacije in dopušča vzporedno obdelavo podatkov. V takšnem primeru lahko isti program izvedemo vzporedno na veliki količini podatkov, zato ni potrebe po zapleteni kontrolni enoti. Latenca dostopa do pomnilnika pa se zakrije z veliko vzporednimi računskimi operacijami.

Slika 1: Razporeditev tranzistorjev na CPE (levo) in GPE (desno). Zelena polja predstavljajo delež tranzistorjev, namenjen procesiranju, rumena predstavlja kontrolno enoto, oranžna pa pomnilniški delež [8].

Ker je za hitro delovanje tranzistorjev potrebna velika moč, kar pa ob velikem številu tranzistorjev na čipu sprošča velike toplote in onemogoča delovanje, je hitrost procesorjev težko povečevati z višanjem frekvence ure. Višjo učinkovitost delovanja lahko zato iščemo v vzporednosti: grafične procesne enote sestavlja veliko število energijsko učinkovitih in preprostejših procesorjev, ki niso med najhitrejšimi, vendar lahko zaradi njihove številčnosti in vzporednega izvajanja hitreje izvršijo zadano nalogo. Visoka prepustnost operacij odtehta relativno nizko hitrost posamezne operacije in prispeva k hitri izvršitvi celotnega programa. Programski model in arhitektura CUDA je v nadaljevanju podrobneje opisana v okviru potreb za razumevanje diplomskega dela. Opis je povzet po [8].

4.1 Programski model

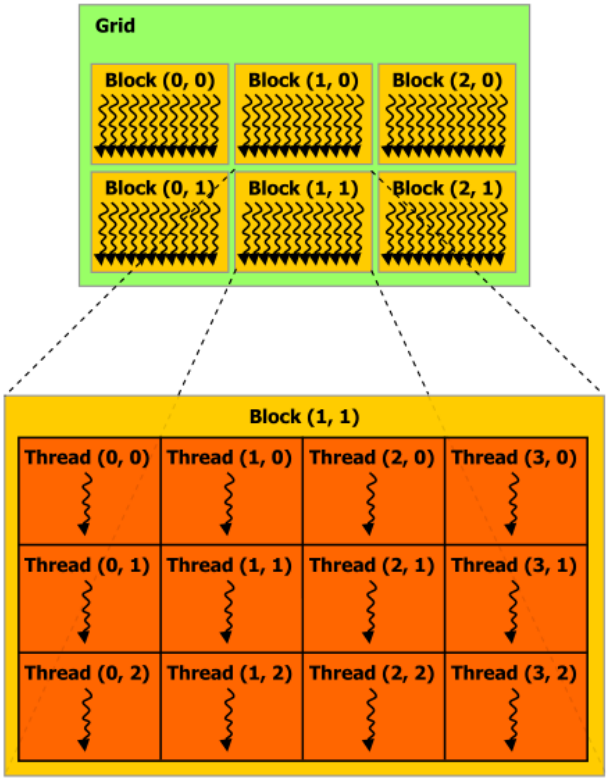
Tipičen program, ki se izvaja na GPE, je sestavljen iz zaporednega dela, ki se izvaja na CPE ter vzporednega dela, ki se izvaja na GPE. GPE deluje kot koprocesor CPE oz. gostitelju. Podatke, s katerimi računamo vzporedno, pa moramo pred tem prenesti v pomnilnik GPE ter po izvršitvi prenesti nazaj. Podrobnosti izvajalnega okolja, kot je delo s pomnilnikom in sinhronizacija, so opisani v skladu s potrebami implementacije.

4.1.1 Ščepci

Vzporedne dele programa, ki se izvajajo na GPE, zapišemo v posebnih funkcijah, imenovanimi ščepci. Zapišemo jih podobno kot običajno C funkcijo. Razlika med C funkcijo in ščepcem pa je v tem, da slednjega vzporedno izvajajo niti.

4.1.2 Hierarhija niti

Pri klicu ščepca določimo število niti, ki bodo ščepec izvajale. Niti pri klicu ščepca organiziramo v bloke, bloke pa v mrežo (slika 2). Bloki lahko imajo 3, mreža blokov pa do 2 razsežnosti. Blok navadno vsebuje do 512 oz. do 1024 niti na najnovejših GPE. Za vsako nit lahko v ščepcu izračunamo njeno unikatno številko preko posebnih spremenljivk, ki hranijo lego niti znotraj bloka, lego bloka v mreži ter razsežnosti bloka. Na tak način lahko vsaki niti določimo podatke, nad katerimi izvaja programsko kodo v ščepcu. Dimenzije in število blokov niti zato navadno določimo glede na vrsto problema, na izbiro pa lahko vpliva tudi zgradba GPE.



Slika 2: Mreža blokov in blok niti. Mrežo sestavljajo bloki, vsak blok pa vsebuje mnogo niti [8].

4.1.3 Izvajanje programa na GPE

Pred opisom vzporednih algoritmov in njihove implementacije si najprej poglejmo potek tipičnega vzporednega programa, ki se izvaja na GPE z arhitekturo CUDA. Kot že omenjeno, deluje GPE kot koprocesor CPE. Pred izvajanjem vzporednih operacij na napravi lahko CPE izvaja zaporedni del programa, za izvajanje vzporednega dela na GPE pa je potrebno sledeče:

1. Rezervacija prostora in prenos vhodnih podatkov v pomnilnik GPE.
2. Klic in izvajanje ščepca.
3. Prenos izhodnih podatkov v pomnilnik CPE in sprostitev pomnilnika na GPE.

CPE lahko nato nadaljuje z izvajanjem zaporednega dela programa ali pa kliče naslednji ščepec za izvajanje na GPE.

4.1.3.1 Rezervacija prostora in prenos podatkov na GPE

Tako CPE kot GPE uporabljata svoj pomnilniški prostor. Na napravo je zato potrebno pred klicem ščepca prenesti vhodne podatke. Za te podatke je potrebno najprej rezervirati prostor, prav tako pa moramo rezervirati prostor tudi za izhodne podatke, kamor zapisujemo rezultat računanja na napravi. Za našteta opravila poskrbimo s klici funkcij, ki jih nudi izvajalno okolje CUDA.

4.1.3.2 Klic in izvajanje ščepca

Preden CPE prepusti izvajanje GPE, mora izvesti klic ščepca. Pred klicem ščepca se navadno določi še:

- razsežnosti bloka niti, ki bodo izvajale ščepec na GPE in
- razsežnosti mreže blokov.

Klic ščepca zgleda kot klic navadne funkcije, le da pri klicu poleg argumentov podamo še parametre, ki določajo organizacijo niti, ščepec pa deklariramo z dodatkom rezervirane besede `__global__`. Ščepec nato vzporedno izvajajo niti. Ko se ščepec konča, se nadzor nad izvajanjem programa vrne CPE.

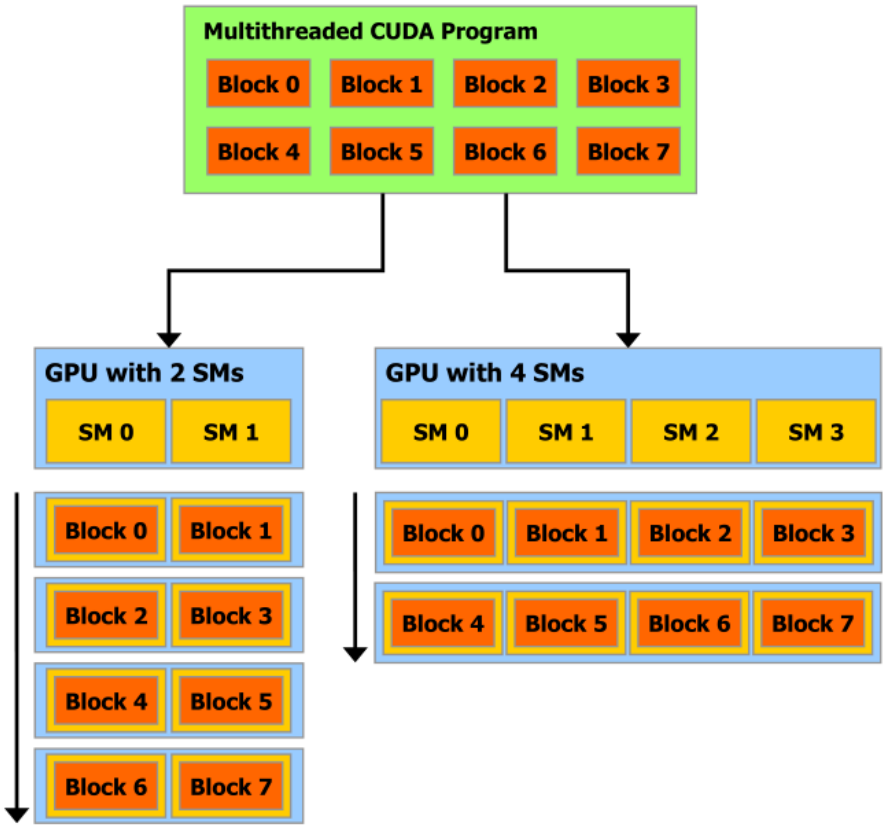
4.1.3.3 Prenos podatkov iz GPE

Prenos podatkov z naprave izvedemo s klicem funkcije izvajalnega okolja CUDA funkcije ter s parametrom določimo smer prenosa. Na koncu pa je potrebno sprostiti uporabljen pomnilnik na GPE.

4.2 Zgradba GPE z arhitekturo CUDA

CUDA je zasnovana kot polje t.i. »streaming multi-procesorjev«, na kratko SM. Število SM je navadno odvisno od zmogljivostnega in cenovnega razreda GPE. Vsak SM sestavljajo preprostejši, t.i. skalarni procesorji (navadno 8), v nadaljevanju SP, na katerih se vzporedno izvajajo niti. Na vsakem SM se nahaja skupni pomnilnik, ki omogoča komunikacijo niti znotraj bloka in je hitrejši od glavnega, globalnega pomnilnika GPE. Pomnilniška hierarhija bo opisana v nadaljevanju.

GPE je v celoti odgovorna za dodeljevanje in razporejanje blokov med SM - programerjem za to ni potrebno skrbeti. Pred klicem ščepca je potrebno določiti razsežnosti mreže blokov in blokov niti, GPE pa jih sama dodeli med SM. Tukaj je potrebno omeniti, da lahko število blokov močno preseže število SM. GPE takrat porazdeli več blokov na isti SM in zagotovi, da se niti istega bloka izvajajo na istem SM ter da se vsi bloki izvedejo pred klicem naslednjega ščepca. Komunikacija in sinhronizacija med nitmi je mogoča le znotraj bloka, bloki pa se med seboj izvajajo neodvisno - vrstnega reda blokov ne moremo določiti. Zagotoviti moramo, da lahko bloke niti GPE razporeja in izvaja neodvisno med seboj in v poljubnem vrstnem redu, ne da bi to vplivalo na rezultat. Tako je zagotovljena skalabilnost; program se lahko izvaja na GPE z različnim številom SM (slika 3), ne glede na število in velikost blokov. GPE lahko tako tudi učinkovito upravlja z bloki – ko se določen blok konča, se mesto na SM nemudoma zagotovi naslednjemu bloku.



Slika 3: Program, ki ga izvaja 8 blokov niti, se lahko izvede na različnih GPE. Na napravi z 2 SM (levo) se bloki razporedijo tako, da vsak SM izvede 4 bloke, na napravi s 4 SM pa 2 bloka [8].

4.2.1 SIMT arhitektura in snopi

Da se lahko hkrati na SM izvaja veliko število niti, uporablja GPE posebno arhitekturo SIMT (angl. Single Instruction Multiple Thread), ki narekuje, da isti ukaz izvrši mnogo niti. GPE razdeli, razporeja in izvaja niti na SM znotraj bloka v skupinah po 32 niti, imenovane snopi. Izvajalni konteksti snopov (vsebina registrov niti, programski števec niti) so shranjeni na dodeljenem SM v registrih. V primeru, da niti snopa zahtevajo časovno potraten dostop do pomnilnika, lahko MP brez časovne izgube začne izvajati drug snop.

Ker vse niti znotraj snopa izvajajo isti ukaz, je učinkovitost najvišja, če se potek vseh niti v snopu ujema. Niti znotraj snopa so tudi implicitno sinhronizirane – če pride do vejitve poteka izvajanja niti znotraj snopa, tiste niti, ki ne izpolnjujejo pogoja za vejitev, pričakajo ostale.

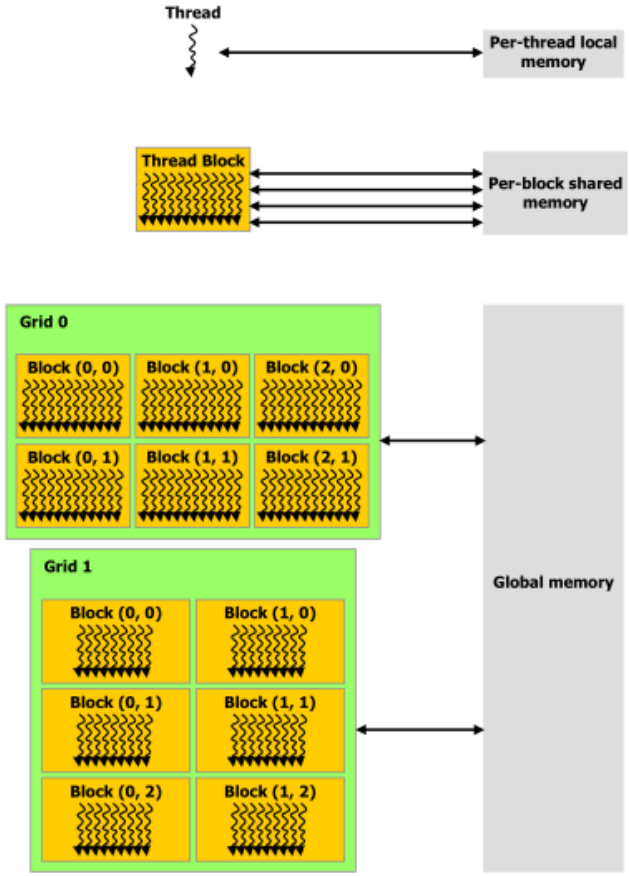
Dostopi do globalnega pomnilnika se izvajajo v segmentih po 32, 64 ali 128 bajtov, hkrati pa lahko dostop do pomnilnika izvede 16 niti oz. t.i. pol-snop (angl. »half-warp«). Najučinkovitejše delovanje dosežemo, če so naslovi pomniških zahtev niti istega pol-snopa zaporedni – tako se z 1 dostopom do globalnega pomnilnika postreže ves pol-snop. V nasprotnem primeru se dostop izvede za vsako nit snopa posebej. Za opisan pristop starejše GPE zahtevajo, da morajo zaporedne niti pol-snopa dostopati do zaporednih pomnilniških lokacij. Novejše naprave pa vrstnega reda pomnilniških lokacij ne predpisujejo. CUDA izvajalno okolje nudi funkcije za delo s snopi, ki pa bodo za potrebe razumevanja algoritma, opisane pri vzporedni implementaciji vrstičnega algoritma.

4.2.2 Računska združljivost

Vsaka CUDA naprava je označena z revizijsko številko. Naprave, z isto »major« revizijsko številko imajo isto osnovno arhitekturo in so med seboj združljive. Pri prevajanju programa moramo to upoštevati, ter prevesti program za primerno arhitekturo. Pri izdelavi diplomske dela sem uporabljal izdelek z revizijsko številko 1.3.

4.3 Pomnilniška hierarhija

Na GPE se srečamo z več vrstami pomnilnika (slika 4), ki se med seboj razlikujejo po hitrosti dostopa in velikosti. Za razumevanje tega diplomskega dela je potrebno predvsem razlikovati med globalnim in skupnim pomnilnikom GPE.



Slika 4: Pomnilniška hierarhija: posamezna nit ima na voljo lastne registre, niti istega bloka lahko dostopajo do lokacij v skupnem pomnilniku. Vse niti ščepca pa lahko dostopajo do globalnega pomnilnika GPE [8].

4.3.1 Registri in lokalni pomnilnik

Pri izvajanju ščepca se vedno uporabljajo registri. Vsaka nit ima na voljo svoje zasebne registre, ki nudijo najhitrejši dostopni čas. Namenjeni so za lokalne spremenljivke, ki jih določimo znotraj ščepca. Če zmanjka registrov, se spremenljivke shranijo znotraj lokalnega pomnilnika, ki se nahaja poleg globalnega pomnilnika.

4.3.2 Skupni pomnilnik

Vse niti znotraj bloka imajo dostop do hitrega skupnega pomnilnika, ki je majhen v primerjavi z globalnim pomnilnikom in se nahaja neposredno na SM. Življenjska doba podatkov v skupnem pomnilniku je enaka trajanju procesiranja celotnega bloka niti. Skupni pomnilnik lahko uporabimo za komunikacijo med nitmi znotraj istega bloka. Prav tako pa lahko v določenih primerih dosežemo hitrejše delovanje, če podatke prenesemo iz globalnega v skupni pomnilnik. Pri tem smo omejeni z relativno majhno velikostjo skupnega pomnilnika. Skupni pomnilnik rezerviramo kot spremenljivko oz. polje znotraj ščepca s pripisom rezervirane besede `__shared__`.

4.3.3 Globalni pomnilnik

Glavni pomnilnik na GPE se imenuje tudi globalni pomnilnik. Nahaja se na napravi, vendar ne na čipu skupaj s SM, zato so dostopni časi veliko daljši od dostopa do skupnega pomnilnika. Pomnilnik rezerviramo pred klicem ščepca. Med izvajanjem ščepca globalnega pomnilnika ne moremo dinamično rezervirati (razen na napravah z računsko združljivostjo 2.0 ali več). Podatki se v globalnem pomnilniku ohranijo tudi po izteku ščepca. Do globalnega pomnilnika lahko dostopajo vse niti iz vseh blokov, ki izvajajo ščepec. Ker vrstnega reda izvajanja blokov niti ne poznamo, moramo paziti, kadar več niti bere ali piše na isto mesto v glavnem pomnilniku. CUDA izvajalno okolje zato nudi t.i. atomske operacije, ki preprečujejo podatkovne nevarnosti v glavnem pomnilniku.

4.3.4 Konstantni in teksturni pomnilnik

Poleg opisanih pomnilnikov naprava CUDA vsebuje še konstantni in teksturni pomnilnik, ki omogočata branje vsem nitim. Prvi vsebuje gostiteljsko programski kodo, ki se vanj shrani pred izvajanjem ščepca. Nahaja se izven čipa, za hitrejše delovanje pa se uporablja v kombinaciji s predpomnilnikom SM. Teksturni pomnilnik je namenjen hranjenju tekstur v posebnih podatkovnih strukturah. V tem diplomskem delu ga ne bomo uporabljali.

Poglavje 5

5 Vzporedni algoritem in implementacija

Oba opisana zaporedna algoritma smo implementirali vzporedno na GPE z arhitekturo CUDA. V tem poglavju bomo predstavili idejo pristopa k vzporednosti, opis algoritmov in implementacijo.

5.1 Vzporedni vrstično-stolpični algoritem

Ta algoritem uporablja pristop, podoben klasičnemu, »šolskemu« množenju gostih matrik na arhitekturi CUDA, ki je opisan tudi v [8].

5.1.1 Ideja

Vhodni matriki sta matriki A in B razsežnosti $m \times k$ in $k \times n$, izhodna matrika C pa je razsežnosti $m \times n$. Ideja vzporedne implementacije tega algoritma je, da vsaka nit računa enega od $m \times n$ elementov izhodne matrike C. Sprehod skozi elemente produkta C v zaporedni izvedbi lahko namreč izvedemo vzporedno. Tako se, za razliko od zaporedne izvedbe, vsi elementi računajo hkrati.

```
foreach C(i,j) in C (vzporedno)
      vrstica_A= A(i,:);
      stolpec_B= B(:,j);
      A(i,k)= naslednji_nz(vrstica_A);
      B(l,j)= naslednji_nz(stolpec_B);
      while(A(i,k)!= null && B(i,k) != null)
            if(k=l)
                  vsota  = vsota + A(i,k)*B(l,j);
                  A(i,k) = naslednji_nz(vrstica_A);
                  B(l,j) = naslednji_nz(stolpec_B);
            else if(k<l)
                  A(i,k) = naslednji_nz(vrstica_A);
            else
                  B(l,j) = naslednji_nz(stolpec_B);
      if(vsota!==0)
            C(i,j)=vsota;
```

Psevdo-koda ideje. Izvedba spominja na zaporedno implementacijo, le da elemente računamo vzporedno.

Ključne lastnosti algoritma lahko ob psevdo-kodi povzamemo v naslednjih točkah:

- Nit, ki izvaja ščepec, izračuna pozicijo svojega elementa $C(i,j)$ znotraj produkta.
- Za element $C(i,j)$ izberemo i-to vrstico matrike A in j-ti stolpec matrike B.
- V zanki se sprehajamo skozi elemente izbrane vrstice in stolpca. Med seboj pomnožimo elemente vrstice A in stolpca B z enakim indeksom v vrstici oz. stolpcu in delni produkt prištejemo k vsoti.
- Neničelen rezultat kot trojico `(i,j,vsota)` zapišemo v strukturo matrike C.

5.1.2 Celoten potek programa z vzporednim vrstično-stolpičnim algoritmom

V nadaljevanju bodo podrobneje opisane operacije in algoritem, ki jih izvedemo na GPE (točke 3, 4 in 5), spodnji potek pa naj služi lažjemu pregledu nad izvajanjem celotnega programa:

1. Branje datotek z vhodnimi matrikami A in B in hranjenje v pomnilniku v zapisu CSR.
2. Pretvorba matrike B v zapis CSC.
3. Rezervacija prostora za vhodne in izhodne podatke na GPE in prenos vhodnih podatkov v globalni pomnilnik GPE iz pomnilnika CPE.
4. Določitev mreže blokov z nitmi in klic ščepca, ki izvede algoritem za množenje matrik.
5. Prenos izhodnih podatkov iz globalnega pomnilnika GPE v pomnilnik CPE.
6. Urejanje produkta in zapis v datoteko.

5.1.3 Rezervacija prostora in prenos vhodnih podatkov

Za izvajanje vzporednega algoritma moramo na napravi najprej rezervirati prostor:

- za matriko A v zapisu CSR; prostor rezerviramo za vsa 3 polja zapisa CSR,
- za matriko B v zapisu CSC; prostor rezerviramo za vsa 3 polja zapisa CSC,
- za izhodno matriko C v zapisu COO; prostor rezerviramo za vsa 3 polja zapisa COO,
- za podatek o trenutnem številu neničelnih elementov izhodne matrike, do katerega dostopajo vse niti.

Na napravo moramo nato prenesti še vnaprej znane vhodne podatke, matriko A in B. Za rezervacijo pomnilnika poskrbi funkcija `cudaMalloc()`, za prenos podatkov pa `cudaMemcpy()`. Spodnji izsek kode prikazuje rezervacijo prostora in prenos vhodne matrike A v globalni pomnilnik na napravi.

```
CSR d_A; /*spremenljivka za strukturo na napravi*/

d_A.width=A.width;
d_A.height=A.height;
d_A.nnz=A.nnz;

size_t sizeI=(A.height+1)*sizeof(int); /*velikost polja kazalcev*/
size_t sizeN=A.nnz*sizeof(int);        /*velikost polja indeksov stolpec*/
size_t sizeNV=A.nnz*sizeof(float);     /*velikost polja vrednost*/

cudaError_t err = cudaMalloc(&d_A.ptr, sizeI);/*rezervacija prostora za ptr*/
err = cudaMemcpy(d_A.ptr, A.ptr, sizeI, cudaMemcpyHostToDevice); /* prenos */
err = cudaMalloc(&d_A.col, sizeN);     /*rezervacija prostora za j indekse */
err = cudaMemcpy(d_A.col, A.col, sizeN, cudaMemcpyHostToDevice); /* prenos */
err = cudaMalloc(&d_A.val, sizeNV); /*rezervacija prostora za vrednosti */
err = cudaMemcpy(d_A.val, A.val, sizeNV, cudaMemcpyHostToDevice); /*prenos*/
```

Izsek kode: Rezervacija prostora za matriko A in prenos podatkov na napravo.

Podobno storimo tudi za matriko B. Matrika C se na napravi izračuna, zato moramo zanjo le rezervirati prostor. Zaradi narave operacije število neničelnih elementov produkta vnaprej ni znano, kar povzroči problem pri rezervaciji prostora za produkt. Problem smo rešili tako, da smo na napravi rezervirali velik del preostalega globalnega pomnilnika, z naprave pa prenesli le del, ki ga dejansko zasede produkt. Po izteku ščepca je namreč znano število `nnz` produkta.

5.1.4 Klic ščepca

Algoritem se izvede znotraj ščepca, ki ga pokličemo s tolikšnim številom niti, kolikor je vseh elementov v matriki produkta. Mrežo niti razdelimo v bloke velikosti `BLOCK_SIZE` ×

BLOCK_SIZE. Za konstanto BLOCK_SIZE smo uporabili število 16, saj je pri 256 nitih v bloku glede na preizkušanje algoritem deloval najhitreje. V 2D mrežo blokov vključimo toliko blokov, da je število niti tolikšno, kot je vseh elementov v matriki C. Če razsežnosti produkta C niso deljive z BLOCK_SIZE, število blokov zaokrožimo navzgor, odvečne niti pa nato predčasno zapustijo ščepec. Sledi klic ščepca, kjer kot argumente podamo kazalce na uporabljene strukture (vhodni matriki, prostor za izhodno matriko ter števec neničelnih elementov).

```
dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
dim3 dimGrid((B.width+dimBlock.x-1)/dimBlock.x,(A.height+dimBlock.y-
1)/dimBlock.y);
spmmKernelRowColLarge<<<dimGrid, dimBlock>>>(d_A, d_B, d_C, d_nnz);
```

Izsek kode: določanje razsežnosti blokov niti in mreže niti in klic ščepca.

5.1.5 Izvajanje ščepca z algoritmom

Vsaka nit najprej izračuna svoji koordinati v mreži niti, ki predstavljata i in j koordinate v matriki produkta. Niti, katere indeksi v mreži so izven razsežnosti matrike, takoj končajo izvajanje.

```
int row = blockIdx.y*blockDim.y+threadIdx.y;
int col = blockIdx.x*blockDim.x+threadIdx.x;
if(!(row<A.height && col<B.width))
      return;
```

Izsek kode: nit izračuna svojo pozicijo v mreži, ki predstavlja indeks i in j računanega elementa v C. Niti, z indeksi izven razsežnosti matrike, takoj zapustijo izvajanje.

Nit računa svoj element v izhodni matriki C na poziciji $C(i,j)$ kot produkt i-te vrstice matrike A in j-tega stolpca matrike B. Vsaka nit izračuna kazalca na začetek elementov pripadajoče vrstice in stolpca ter njun konec.

```
int aColIt = A.ptr[row];
int aColEnd = A.ptr[row+1];
int bRowIt= B.ptr[col];
int bRowEnd = B.ptr[col+1];
```

Izsek kode: nit poišče začetek in konec vrstice v A in stolpca v B, katerih produkt je računan element.

Sledi zanka, v kateri se bere in množi neničelne elemente pripadajoče vrstice A oz. stolpca B. Ta del algoritma je enak zaporednemu. Vsaka nit preveri, če je izračunan pripadajoč element različen od nič. V tem primeru sledi zapis v strukturo matrike C na naslednje prosto mesto z uporabo atomske operacije.

```
while(aColIt<aColEnd && bRowIt<bRowEnd){
  colIdxA = A.col[aColIt];
  rowIdxB = B.row[bRowIt];
  if(colIdxA == rowIdxB){
    sum+=A.val[aColIt]*B.val[bRowIt];
    aColIt++;
    bRowIt++;
  }else if( colIdxA< rowIdxB){
    aColIt++;
  }else{
    bRowIt++;
  }
}
if(sum!=0){       /* zapis neničelnih elementov v podatkovno strukturo COO  */
  count =atomicAdd(nnz, plusOne);
  C.val [count]=sum;
  C.row [count]=row;
  C.col [count]=col;
}
```

Izsek kode: Množenje istoležnih elementov pripadajoče vrstice in stolpca. Nit zapiše produkt v strukturo matrike C v zapisu COO z uporabo atomske operacije, če je le-ta različen od 0.

5.1.6 Prenos podatkov iz GPE in sprostitev pomnilnika

Po izvedbi ščepca je potrebno izhodno matriko in števec neničelnih elementov prenesti iz pomnilnika GPE v pomnilnik CPE, za kar ponovno uporabimo funkcijo izvajalnega okolja `cudaMemcpy()`. Prenašamo le neničelne elemente. Sledi še sprostitev rezerviranega pomnilnika za uporabljene strukture z uporabo funkcije `cudaFree()`.

```
err = cudaMemcpy(&C.nnz, d_nnz, sizeof(int), cudaMemcpyDeviceToHost);
sizeNV=C.nnz*sizeof(float);
sizeN=C.nnz*sizeof(int);
err = cudaMemcpy(C.rowIndeces, d_C.rowIndeces, sizeN, cudaMemcpyDeviceToHost);
err=cudaMemcpy(C.coloumnIndeces,d_C.coloumnIndeces,sizeN,cudaMemcpyDeviceToHost;
err = cudaMemcpy(C.values, d_C.values, sizeNV, cudaMemcpyDeviceToHost);

cudaFree(d_C.values);
cudaFree(d_C.rowIndeces);
cudaFree(d_C.coloumnIndeces);
/*
  izpuščeno: sproščanje struktur A, B
*/
```

Izsek kode: Primer prenosa rezultata iz naprave ter sproščanja virov.

5.2 Vzporedni vrstični algoritem

5.2.1 Ideja

Zaporedni algoritem po vrsti računa vrstice matrike produkta C, razsežnosti $m \times n$. Ideja vzporedne izvedbe algoritma pa je, da se hkrati računajo vse vrstice.

```
foreach vrstica_A=A(i,:) in A (vzporedno)
      foreach element_A=A(i,l) in vrstica_A
            vrstica_B=B(l,:);
            foreach element_B=B(l,j) in vrstica_B
                  medpomnilnik[j]+= A(i,l)*B(l,j);
      foreach nz=medpomnilnik[j]!=0 in medpomnilnik
           nz->C(i,j);
```

Psevdokoda ideje vzporednega vrstičnega algoritma.

Idejo algoritma lahko ob psevdo-kodi povzamemo v naslednjih točkah:

- Hkrati se računa vse vrstice matrike C.
- Za vsako izhodno vrstico rezerviramo medpomnilnik dolžine n, kamor bomo zapisovali delne produkte.
- Za vsak neničelni element $A(i,l)$ vrstice $A(i,:)$ določimo pripadajočo l-to vrstico matrike B.
- Vsak neničelni element $B(l,j)$ vrstice $B(l,:)$ pomnožimo z $A(i,l)$.
- Delni produkt $A(i,l) * B(l,j)$ prištejemo na j-to mesto v medpomnilnik.
- Ko je trenutna vrstica izračunana, prepišemo neničelne elemente iz medpomnilnika v strukturo matrike C.

Algoritem smo najprej implementirali v enostavnejši različici, kjer vsako vrstico produkta računa 1 nit. Nato pa smo implementacijo poskušali izboljšati tako, da vsako vrstica produkta računa snop 32 niti [3, 7]. Opis te implementacije sledi v nadaljevanju. Snop niti računa hkrati natanko eno vrstico produkta. V primeru, ko je vrstic v produktu več kot snopov, se snop po izračunu trenutne vrstice premakne na naslednjo. Prednost takšnega pristopa je, da so vse niti znotraj snopa že implicitno sinhronizirane, vsak ukaz se hkrati izvede za vse niti snopa naenkrat, pol-snop niti pa lahko do zaporednih pomnilniških lokacij pristopi z enim branjem oz. pisanjem pomnilnika.

5.2.2 Implementacija

Vzporedni program sprejme enake vhodne podatke kot serijski: poti do dveh vhodnih datotek. Izhod, ki ga vrne program, pa je matrika produkta, zapisana v datoteki. Potek celotnega programa je naslednji:

1. Branje datotek z vhodnimi matrikami A in B in hranjenje v pomnilniku v zapisu CSR.
2. Rezervacija prostora za vhodne in izhodne podatke na GPE in prenos vhodnih podatkov v globalni pomnilnik GPE iz pomnilnika CPE.
3. Določitev mreže blokov z nitmi in klic ščepca, ki izvede algoritem za vrstično množenje matrik.
4. Prenos izhodnih podatkov iz globalnega pomnilnika GPE v pomnilnik CPE.
5. Urejanje produkta in zapis v datoteko.

Sledijo podrobneje opisani koraki 2, 3 in 4 kateri so specifični za poganjanje programa na GPE. Zaradi podobnosti z vzporedno implementacijo vrstično-stolpčnega algoritma smo nekatere dele opisa in primere iz kode izpustili. Algoritem smo implementirali s pomočjo psevdokode in opisa v [3] in [7].

5.2.3 Rezervacija prostora in prenos vhodnih podatkov

Za izvajanje vzporednega algoritma moramo na napravi najprej rezervirati prostor:

- za matriko A v zapisu CSR; prostor rezerviramo za vsa 3 polja zapisa CSR,
- za matriko B v zapisu CSR ; prostor rezerviramo za vsa 3 polja zapisa CSR,
- za izhodno matriko C v zapisu COO; prostor rezerviramo za vsa 3 polja zapisa COO,
- za podatek o trenutnem številu neničelnih elementov izhodne matrike, do katerega dostopajo vse niti,
- za medpomnilnik, kamor zapisujemo delno izračunane vrstice izhodne matrike C.

Na napravo moramo nato prenesti še vnaprej znane vhodne podatke, to sta matriki A in B v CSR zapisu. Algoritem potrebuje za računanje vsake vrstice medpomnilnik, kamor shranjujemo elemente računane vrstice glede na stolpični indeks. Medpomnilnik torej hrani vrstico v gosti, ne-redki obliki, ko pa je vrstica izračunana, v strukturo matrike C prepišemo neničelne elemente medpomnilnika. Ker se hkrati izvaja veliko snopov in vsak snop potrebuje svoj medpomnilnik, moramo njegovo velikost omejiti. Če je vrstica daljša od medpomnilnika, le-to računamo po odsekih. Velikost medpomnilnika mora zaradi načina implementacije biti večkratnik velikosti snopa, velikost pa je omejena na 102400 števil v plavajoči vejici oz. na širino matrike B pri manjših matrikah. Pri tej velikosti smo glede na preizkušanje dosegli najhitrejši čas.

5.2.4 Klic ščepca

Pred klicem ščepca moramo določiti razsežnosti mreže blokov. Ker vsako vrstico računa 32 niti snopa, ščepec izvaja `NUM_WARPS* 32` niti (`NUM_WARPS` je konstanta, s katero določimo število snopov, ki jih izvajajmo hkrati). Niti prerazporedimo v bloke razsežnosti `WARP_SIZE* WARPS_IN_BLOCK` (`WARP_SIZE` predstavlja število niti v snopu, `WARPS_IN_BLOCK` pa število snopov v bloku). Algoritem smo optimizirali za GPE, ki smo jo uporabljali pri testiranju in je opisana v nadaljevanju. Med preizkušanjem različnih konfiguracij se je izkazalo, da algoritem deluje najhitreje kadar je `WARPS_IN_BLOCK` konstanta 8, `NUM_WARPS` pa 240. Takrat ščepec izvaja natanko 30 blokov niti, kolikor je SM v uporabljeni napravi, vsak blok pa sestavlja 8 snopov niti. Z izvajanjem več blokov višje pohitritve ni. Mreža blokov je enorazsežna in skupaj vsebuje toliko snopov niti, kolikor se hkrati računa vrstic, v našem primeru 30. Ko določimo razsežnosti blokov in mreže niti, kličemo ščepec z algoritmom, ki mu kot argumente podamo kazalce na prej rezervirane pomnilniške lokacije podatkov: vhodnih matrik, izhodne matrike, medpomnilnika in števca neničelnih elementov produkta.

```
dim3 dimBlock(WARP_SIZE, WARPS_IN_BLOCK);
dim3 dimGrid((NUM_WARPS + dimBlock.y - 1) / dimBlock.y);
spmmKernelWarpLargeGlobal<<<dimGrid,
dimBlock>>>(d_A,d_B,d_C,d_nnz,d_medpomnilnik);
```

Izsek kode: določanje razsežnosti mreže in blokov ter klic ščepca.

5.2.5 Izvajanje ščepca z algoritmom

Vsaka nit ščepca najprej izračuna zaporedno številko svojega snopa ter svojo pozicijo v snopu 32 niti.

```
int blockId = blockIdx.x+ blockIdx.y * gridDim.x;
int threadId = blockId * blockDim.x*blockDim.y
+threadIdx.y*blockDim.x+threadIdx.x;
int warpId=threadId / WARP_SIZE;        //ostevilcimo snope
int laneId=threadIdx.x % WARP_SIZE;     //ostevilcimo niti snopa -> 0 … 31
```

Izsek kode: vsaka nit izračuna številko snopa, h kateremu pripada ter svojo pozicijo znotraj snopa niti.

Za vsak snop rezerviramo mesto v skupnem pomnilniku, kamor se zapiše vrednost in j indeks trenutnega neničelnega elementa matrike A. Ker se skupni pomnilnik uporablja za potrebe komunikacije med nitmi v snopu, uporabimo pri deklaraciji rezervirano besedo `volatile`. To prepreči morebitno optimizacijo prevajalnika, ki lahko v nekaterih primerih prestavi vrednosti v registre niti.

```
__shared__ volatile int   sColA[nWarps];
__shared__ volatile float sValA[nWarps];
```

Izsek kode: rezervacija skupnega pomnika. Vsak snop zahteva po 1 celoštevilsko mesto za hranjenje indeksa ter eno mesto za število v plavajoči vejici za hranjenje vrednosti elementa.

Vsak snop niti računa svojo vrstico produkta. Če je število vrstic produkta večje od števila snopov, se snop niti po izračunu trenutne vrstice premakne na naslednjo. Kot pri zaporednem algoritmu, tudi tukaj potrebujemo medpomnilnik za izračun posamezne vrstice produkta. Ker potrebuje vsak snop svoj medpomnilnik in je velikost globalnega pomnilnika omejena, računamo izhodne vrstice po odsekih. V prvi zanki se (vzporedno) sprehajamo skozi vrstice matrike produkta, druga pa služi izračunavanju vrstic po odsekih. Prva zanka se izvaja vzporedno za `nWarps` vrstic hkrati, za vsako vrstico je potrebno ponastaviti vrednosti v medpomnilniku na nič. Vsaka nit izračuna svoj kazalec na neničelni element iz vrstice A. Dokler kazalec katerekoli niti v snopu kaže na element znotraj pripadajoče vrstice, niti nadaljujejo z izvajanjem. To preverimo s posebno funkcijo `__any()`, ki preveri pogoj za vse niti snopa in v primeru, da ta drži za vsaj eno nit snopa, ovrednoti pogoj kot resničen. Takšen pristop je potreben, ker potrebujemo za nadaljnje izvajanje vse niti snopa. Niti nato preberejo vrednost in stolpični indeks svojega elementa, če pa je njihov element neveljaven (kazalec kaže izven trenutne vrstice) pa preberejo kot stolpični indeks vrednost -1, in ga kasneje ne upoštevamo.

```
/*
deklaracija spremenljivk,
nit izračuna svojo pozicijo, warpId, laneId,
rezervacija SM
*/
for(row=warpId;row < C.height;row+=nWarps){     /*1. Zanka -> skozi vrstice C*/
  for(partRow=0;partRow<B.width; partRow+=PART_ROW_GLOBAL){       /*2. zanka */
    startIdx=partRow;                                  /*začetni j-indeks odseka*/
    endIdx=startIdx+ buffer_length;                    /* končni j-indeks odseka*/
    j=0;
    while(j< buffer_length){/*inicializacija medpomnilnika*/
      buffer.values[offsetLane+j]=0;
      j+=WARP_SIZE;
    }
    aColIt  = A.ptr[row] + laneId;
    aColEnd = A.ptr[row + 1];
    while(__any(aColIt<aColEnd)){
      colA = aColIt < aColEnd ? A.col[aColIt] : -1;
      valA = aColIt < aColEnd ? A.val[aColIt] : 0.0;
      /*
         notranja zanka
      */
      aColIt+=WARP_SIZE;
    }
    /*
      prepis iz medpomnilnika v strukturo
    */
  }
}
```

Izsek kode: zunanja zanka posrbi, da vsak snop računa svojo vrstico C. Če je vrstic več kot snopov, izvede vsak snop več iteracij algoritma. V naslednji zanki niti snopa odsekih `PART_ROW_GLOBAL` računajo pripadajočo vrstico. Vsaka nit snopa prebere po en neničelni element matrike A.

V notranji zanki se po vrsti premikamo skozi vseh `WARP_SIZE` prebranih elementov trenutne vrstice A. Element shranimo v skupni pomnilnik ter tako omogočimo dostop do elementa ostalim nitim snopa. Nato vse niti za trenuten element A izračunajo kazalec na neničelni element v pripadajoči vrstici (glede na j-ti indeks trenutnega elementa A) v matriki B. Vse niti imajo dostop do tega podatka preko skupnega pomnilnika. Če kaže kazalec znotraj pripadajoče vrstice B, nit prebere vrednost in stolpični indeks elementa iz B. Nato še preverimo, če leži element znotraj trenutno računanega odseka, in ga v tem primeru pomnožimo z vrednostjo trenutnega elementa A (ta se nahaja v skupnem pomnilniku). Tako se hkrati lahko izvede 32 množenj. Produkt se zapiše na primerno mesto medpomnilnika.

```
for(k = 0; k < WARP_SIZE; k++){
  if(laneId == k){
    sColA[warpId] = colA;
    sValA[warpId] = valA;
  }
  if(sColA[warpId]==-1)
    break;
  bRowIt  = B.ptr[sColA[warpId]    ]+laneId;
  bRowEnd = B.ptr[sColA[warpId] + 1];
  while(bRowIt<bRowEnd){
    colB = B.col[bRowIt];
    valB = B.val[bRowIt];
    if(colB>=startIdx && colB<endIdx )
       medpomnilnik.values[offset+colB%PART_ROW_GLOBAL]+=sValA[warpId]*valB;
    bRowIt += WARP_SIZE;
  }
}
```

Izsek kode: notranja zanka algoritma

Nazadnje prepišemo neničelne vrednosti iz medpomnilnika v COO strukturo matrike C. Vsaka od 32 niti ščepca prepiše en neničelni element na naslednje prosto mesto v strukturi ter se premakne na naslednji element. To ponavljamo dokler ne prepišemo vseh neničelnih elementov iz medpomnilnika. Nato se snop niti premakne na naslednji odsek oz. naslednjo vrstico, če le-ta obstaja. Posebnost pri zapisu elementov v strukturo matrike C je uporaba t.i. atomskih operacij. Ker vse niti vseh snopov hkrati zapisujejo vrednosti v strukturo na naslednje prosto mesto, lahko pride do izgube podatkov. Z uporabo atomskih operacij zagotovimo, da vsaka nit izračuna prosto mesto in pri tem ne prepiše ostalih vrednosti .

```
while(j<PART_ROW_GLOBAL){
  curVal=medpomnilnik.values[offsetLane+j];
  if(curVal != 0){
    count =atomicAdd(nnz, plusOne);
    C.values[count]=curVal;
    C.rowIndeces[count]=row;
    C.coloumnIndeces[count]=laneId+j+ partRow;
  }
  j+=WARP_SIZE;
}
```

Izsek kode: Prepis vrednosti iz medpomnilnika v strukturo matrike C z uporabo atomskih operacij.

5.2.6 Prenos podatkov iz GPE in sprostitev pomnilnika

Izhodno matriko v zapisu COO in števec neničelnih elementov prenesemo iz pomnilnika GPE v pomnilnik CPE, kot smo to storili pri vrstično-stolpičnem algoritmu. Sledi še sprostitev rezerviranega pomnilnika za uporabljene strukture.

5.3 Alternativni način zapisa produkta

V obeh opisanih vzporednih algoritmih smo produkt na GPE shranili v COO strukturo matrike C, kar pa zahteva uporabo časovno zahtevnih atomskih operacij. Teh namreč ne gre izvajati vzporedno.

Produkt pa lahko shranimo tudi neposredno v gost zapis, ki lahko vsebuje tudi ničelne vrednosti. Tako se znebimo atomskih operacij. Pri tem načinu v pomnilniku GPE rezerviramo prostor za vrednosti matrike C v velikosti $m \times n$. Matrika je v pomniku shranjena v obliki polja, v katerem si sledijo elementi zaporednih vrstic matrike. Ima pa takšen pristop tudi pomanjkljivosti; iz naprave moramo prenesti celotno matriko v gostem zapisu, kar lahko predstavlja visok časovni strošek. Prav tako takšen zapis zasede dosti več prostora v globalnem pomnilniku GPE. Zato smo omejeni na matrike relativno majhnih razsežnosti.

Pri vrstično-stolpičnem algoritmu ni večjih sprememb v implementaciji. Namesto, da neničelne elemente produkta s pomočjo atomskih operacij shranjujemo v COO strukturo, jih zapišemo na točno določeno mesto v matriki. Pri vrstičnem algoritmu pa v tem primeru ne potrebujemo medpomnilnika, delne produkte pa prištevamo neposredno v izhodno matriko. Pri tem je potrebno najprej ponastaviti vrednosti matrike produkta C. To smo storili s pomočjo preprostega ščepca, kjer vsaka nit zapiše 0 na eno mesto v matriki, izvedemo pa ga pred množenjem.

```
__global__ void initKernel(Matrix C){
    int row = blockIdx.y*blockDim.y+threadIdx.y;
    int col = blockIdx.x*blockDim.x+threadIdx.x;
    if( row<C.height && col<C.width) {
        C.elements[row * C.width + col]=0;
    }
}
```

Izsek kode: ščepec za inicializacijo matrike C. Vsaka nit izračuna pozicijo svojega elementa v matriki ter nanjo zapiše 0.

Poglavje 6

6 Testiranje in rezultati

6.1 O testiranju

Pri testiranju smo kot obe vhodni matriki uporabili kvadratno matriko razsežnosti $n \times n$, ki smo jo pomnožili samo s seboj, pri tem pa merili čase izvajanja posameznih delov programa. Testiranje smo izvedli v dveh korakih. Najprej smo čase izvajanja vseh algoritmov in njihovih različic preizkusili na različnih primerkih redkih matrik, dostopnih v spletni zbirki redkih matrik Matrix Market [11]. V nadaljevanju pa smo čas izvajanja najhitrejših implementacij algoritmov merili še na redkih matrikah, ki smo jih ustvarili sami in jim določili lastnosti: razsežnost n in gostoto neničelnih elementov.

6.1.1 Strojna oprema

Pri testiranju smo za potrebe vzporednega procesiranja uporabljali GPE NVIDIA Tesla C1060 [13] z arhitekturo CUDA z naslednjimi lastnostmi:

- 30 SM,
- 240 SP,
- urina frekvenca 1.296 GHz,
- 32 niti/snop,
- 512 niti/blok,
- 4GB globalnega pomnilnika,
- 16KB skupnega pomnilnika na SM,
- računska združljivost 1.3.

Gostiteljski sistem:

- CPE: Intel Core i5 650 z urino frekvenco 3.20 GHz,
- 4 GB RAM pomnilnika in 4 MB predpomnilnika.

6.1.2 Poganjanje programa

Program smo poganjali na operacijskem sistemu Linux. Ker smo pri vrstičnem algoritmu uporabili funkcijo izvajalnega okolja `__any()`, ki je na voljo s CUDA napravami 1.3 in naprej, moramo program prevesti z naslednjim ukazom:

```
nvcc -o program program.cu -arch=sm_13
```

Program za množenje redkih matrik nato poženemo s spodnjim ukazom, kjer s *st_algoritma* izberemo želen algoritem, vhod1 in vhod 2 pa predstavljata poti do datotek z vhodnimi matrikami.

```
./program vhod1 vhod2 st_algoritma
```

Za preverjanje pravilnosti algoritma lahko datoteko z rezultatom zaporednega algoritma primerjamo z datoteko rezultata vzporedne izvedbe. Na operacijskem sitemu Linux je to mogoče

z uporabo ukaza `grep`, ki mu z argumenti naročimo iskanje in izpis razlik v vhodnih datotekah. Podobno bi lahko storili tudi z ukazom `diff`. Pri pravilnem izvajanju ukaz ne vrne ničesar.

```
grep -Fxvf datoteka1 datoteka2
```

6.2 Testiranje algoritmov na različnih matrikah

Matrike, ki smo jih uporabili pri tem testu, so zapisane v datoteki v t.i. »matrix market« zapisu. Ta zapis je zelo podoben COO zapisu redke matrike, kjer vsaka vrstica v datoteki vsebuje indekse in vrednost enega neničelnega elementa. Elementi so urejeni najprej po stolpičnih, nato vrstičnih indeksih. Sledeči seznam navaja merjene algoritme ter oznake, s katerimi se bomo sklicevali na njih:

1. Vrstično-stolpični algoritem na CPE, na kratko VSC.
2. Vrstično-stolpični algoritem na GPE, kjer rezultat shranjujemo v redko matriko, na kratko VSG-R.
3. Vrstično-stolpični algoritem na GPE, kjer rezultat shranjujemo v gosto matriko, na kratko VSG-G.
4. Vrstični algoritem na CPE, na kratko VVC.
5. Vrstični algoritem na GPE, kjer vrstico produkta računa ena nit in rezultat shranjujemo kot redko matriko, na kratko VVG-RN.
6. Vrstični algoritem na GPE, kjer vrstico produkta računa snop niti in rezultat shranjujemo kot redko matriko, na kratko VVG-RS.
7. Vrstični algoritem na GPE, kjer vrstico produkta računa snop niti in kjer rezultat shranjujemo kot gosto matriko, na kratko VVG-GS.
8. Urejanje rezultata oz. UR.

Algoritmi pod točkami 2, 4, 5 in 6 izračunajo matriko produkta v neurejenem COO zapisu. Če želimo urejen rezultat, je potrebno strukturo še dodatno urediti. Algoritmi pod točkami 1, 3. in 7. tega urejanja ne potrebujejo. Kadar želimo urejen rezultat, je za pošteno primerjavo potrebno čas urejanja (UR) prišteti k času algoritmov. Pri algoritmih, ki potekajo na GPE, smo merili tudi čase prenosa podatkov na GPE ter nazaj na CPE. V čas izvajanja algoritmov je vključen tudi ta čas. Ker sta vhodni matriki prebrani v pomnilnik v CSR zapisu, algoritmi 1,2 in 3 pa potrebujejo matriko B v zapisu CSC, smo v čas izvajanja vključili tudi čas pretvarjanja iz CSR v CSC, čeprav se je, v primerjavi s trajanjem operacije množenja, izkazal kot zanemarljiv. Tabela 1 podaja čase izvajanja vseh implementiranih algoritmov pri vhodnih matrikah različnih razsežnosti n in gostota neničelnih elementov. Uporabljene redke matrike se med seboj razlikujejo tudi po strukturi – neničelni elementi matrik niso enakomerno razporejeni.

Tabela 1: Časi trajanja poteka algoritmov na različnih matrikah. Tabela je urejena po naraščajočih nnz, časi pa so izmerjeni v milisekundah. Algoritmi, ki proizvedejo neurejen rezultat, so označeni z *. Krepko pa so zapisani časi najhitrejše implementacije množenja pri določenem vhodu.

matrika	n	gostota [%]	nnz	VSC	VSG-R*	VSG-G	VVC*	VVG-RN*	VVG-RS*	VVG-GS	UR
bcsstm05	153	0,6535	153	0.33	0.47	0.58	**0.01**	0.6	0.54	0.51	0.11
bcsstk01	48	9,7222	224	0.16	0.55	0.44	**0.02**	0.75	0.56	0.44	0.21
bcsstk03	112	2,9974	376	0.42	0.73	0.59	**0.03**	0.86	0.7	0.57	0.22
bcsstk04	132	10,8471	1890	2.32	2.45	0.93	**0.39**	2.52	2.3	0.69	1.54
bcsstm21	3600	0,0277	3600	165	10.1	49	**0.19**	29.6	5.6	41	0.9
bcsstm12	1473	0,4869	10566	124	15.25	18.3	**1.41**	13	9	7.93	7.7
bcsstk34	588	3,1824	11003	47.82	15.62	7.3	2.7	13.15	11.7	**2.19**	9.05
bcsstm13	2003	0,2984	11973	138	23.57	25.3	2.87	23	14.6	13.8	11.62
bcsstk21	3600	0,1165	15100	446	32.61	65	**1.6**	36.11	14.9	40.41	11.1
bcsstk13	2003	1,0703	42943	574	106	69	14.71	68.34	59.1	**14.5**	52.7
bcsstk18	11948	0,0564	80519	7510	2325	1019	**12.48**	446	82.6	420	82.42
nasa2910	2910	4,55	88603	1827	272	198	37	192.54	121	**29**	114
bcsstk16	4884	0,6189	147631	4598	552	498	54	205	161	76.42	149
bcsstk38	8032	0,2817	181746	9644	1098	1151	**53**	436	192	196.9	198
CAG_mat	1916	5,34	195985	4447	754	527.7	238.6	649.7	303	**36.4**	342
bcsstk17	10974	0,1825	219812	15291	1681	1902	**59.93**	590	213	360	197
msc10848	10848	0.564	620313	50555	4794	4725	**301**	1314	575	366	611
bcsstk39	46772	0,0488	1068033	325101	101886	-	**286**	6566	1381	-	916
nd3k	9000	2,030	1644345	125940	11050	12184	2303	5481	2492	**367**	2943
Si34H36	97569	0,0275	2626974	1860759	206281	-	**1896**	40038	8967	-	10545
nd6k	18000	1,0671	3457658	527911	41660	50356	5235	12674	5761	**1224**	7163
nd12k	36000	0,5500	7128473	2141898	154464	-	**11130**	27625	12425	-	16106
af_shell4	504855	0,0035	9046865	6h+	6h+	-	**1867**	1724595	34877	-	7482
nd24k	72000	0,2776	14393817	7942816	572800	-	**23672**	66205	29051	-	35571

V testu se je za najhitrejšega med vsemi v večini primerov izkazal zaporedni vrstični algoritem VVC. Med vzporednimi algoritmi na GPE pa sta bili najhitrejši obe implementaciji vrstičnega algoritma s snopi, VVG-RS in VVG-GS. Različica VVG-RN, ki za izračun vrstice uporablja eno nit, se izvaja občutno počasneje. Pristop s snopi je torej učinkovit. Algoritem VVG-RN pa lahko že kar takoj označimo kot manj primeren. V nekaterih primerih deluje VVG-GS hitreje od zaporednega algoritma VVC. Če upoštevamo, da je produkt nato že urejen, je ta algoritem še toliko hitrejši. Opazili smo, da se to zgodi pri nekoliko gostejših vhodnih matrikah oz. pri primerkih z gostoto nad 1%. Matrike iz zbirke Matrix Market so namreč zelo redke, gostota neničelnih elementov pa redko preseže 1%. Glavni časovni strošek algoritma VVG-GS je komunikacija oz. prenos produkta v gosti obliki (tabela 2), je pa sam ščepec veliko hitrejši kot pri VVG-RS. Algoritem VVG-RS pa se zaradi atomskih operacij izvaja počasneje, prenos produkta na CPE pa je hitrejši. V nekaterih primerih je na GPE hitrejši VVG-RS, v nekaterih pa VVG-GS. Zaporedno izvedbo pa je (nekajkrat) prehitel samo slednji. Čeprav obe različici računski del algoritma izvedeta hitro, se njuna pomanjkljivost izkaže pri shranjevanju produkta.

Tabela 2: Primerjava deležev v skupnem času vrstičnih implementacij na primeru matrike iz testa. VVG-GS je na tem vhodu najhitrejši, saj se ščepec izvede zelo hitro, veliko časa pa v nasprotju z algoritmom VVG-RS potrebuje prenos podatkov na CPE. VVG-RS potrebuje zaradi atomskih operacij približno 18x več časa za izvedbo ščepca ter skoraj 7x manj časa za prenos.

VVG-GS	VVG-RS	VVC
vhod: nd6k.mtx n: 18000 **celoten čas: 1237 ms** od tega: čas prenosa na GPE CUDA: 14.82 **čas izvajanja ščepca: 301.47** **čas prenosa rezultata iz GPE CUDA: *920.88***	vhod: nd6k.mtx n: 18000 **celoten čas: 5764 ms** od tega: čas prenosa na GPE CUDA: 14.65 **čas izvajanja ščepca: 5599.49** **čas prenosa rezultata iz GPE CUDA: *150.03***	vhod: nd6k.mtx n: 18000 **celoten čas: 5226 ms**

Vrstično-stolpčni algoritem je občutno počasnejši od vrstičnega; že zaporedni algoritem VSC se izvaja veliko počasnejše od zaporednega algoritma VVC. Zaporedna različica VSG-R je na matrikah z nekaj tisoč neničelnimi elementi že dosti hitrejša od zaporedna VSC (tudi, če prištejemo čas urejanja), kar pomeni, da je uporaba vzporednosti sicer učinkovita in izvede algoritem hitreje. Še vedno pa je tudi vzporedna različica počasnejša od zaporednega vrstičnega algoritma VVC. Pri vzporednem vrstično-stolpičnem algoritmu način shranjevanja rezultata ni bistveno vplival na čas izvajanja; v nekaterih primerih je bil hitrejši prvi (VSG-R), v nekaterih drugi način (VSG-G). Pri nobenem od njiju pa nismo izmerili časa, ki bi se približal delovanju hitrejših implementacij. Razlog za počasno delovanje lahko leži v dejstvu, da je veliko časa potrebnega za iskanje istoležnih elementov iz vrstice matrike A in stolpca matrike B, četudi množimo le neničelne elemente. Zato je tudi zaporedna izvedba veliko počasnejša od zaporedne izvedbe vrstičnega algoritma. Vrstično-stolpični algoritem smo zaradi počasnega izvajanja zato ovrednotili kot manj primeren za moženje redkih matrik, saj razen enostavnejše implementacije ne predstavlja nikakršnih prednosti pred ostalimi.

6.3 Nadaljnje testiranje implementacij vrstičnega algoritma

Vzporedni različici vrstičnega algoritma s snopi in CPE implementacija vrstičnega algoritma so na pri izvajanju na raznovrstnih redkih matrikah iz spletne zbirke delovali najhitreje. Zaporedno izvedbo je v nekaterih primerih prehitela vzporedna, ki shranjuje v redko matriko. V teh primerih se je tudi hitrost vzporedne različice, ki shranjuje produkt v redko matriko, približala zaporedni implementaciji. To se zgodi pri množenju matrik, katerih gostota neničelnih elementov je nekoliko nižja. Matrike v spletni zbirki Matrix Market so v večini zelo redke – gostota je pogosto veliko manjša od 1%. Ker se vzporedni algoritmi obnašajo drugače pri različnih gostotah, si za potrebe testiranja želimo vhodnih matrik, katerim lahko določimo poljubno gostoto in razsežnosti. Za potrebe nadaljnjega testiranja smo tako zapisali program, s katerim poženemo najhitrejše implementacije vrstičnega algoritma na vhodnih redkih matrikah poljubnih razsežnosti in gostot. Program podamo razpon razsežnosti n in število nnz ali gostoto vhodne matrike, določimo pa tudi korak, s katerim n in nnz oz. gostoto postopoma večamo. Neničelni elementi v generiranih redkih matrikah so, za razliko od matrik s prvotnega testa, enakomerno porazdeljeni. Pri vzporedni implementaciji lahko zato pričakujemo boljše delovanje, saj bodo nnz produkta enakomerneje razporejeni po vrsticah. Namen tega testiranja pa ni primerjava s prvotnim, ampak predvsem nadaljnja primerjava implementacij. V izhodno datoteko s programom beležimo vse pomembne podatke merjenih algoritmov:

- lastnosti vhodne matrike (n, nnz, gostota),
- čas izvajanja celotnega algoritma (brez urejanja) vseh treh implementacij,
- čas za morebitno urejanje,
- čas prenosa podatkov na in iz GPE,
- čas izvajanja ščepcev,
- pohitritev vzporedne izvedbe v primerjavi z zaporedno.

Podatki iz datoteke nam omogočajo grafični prikaz s pomočjo grafov, za njihov izris pa smo uporabili programski paket Octave.

6.3.1 Izvajanje algoritmov odvisnosti od gostote in razsežnosti

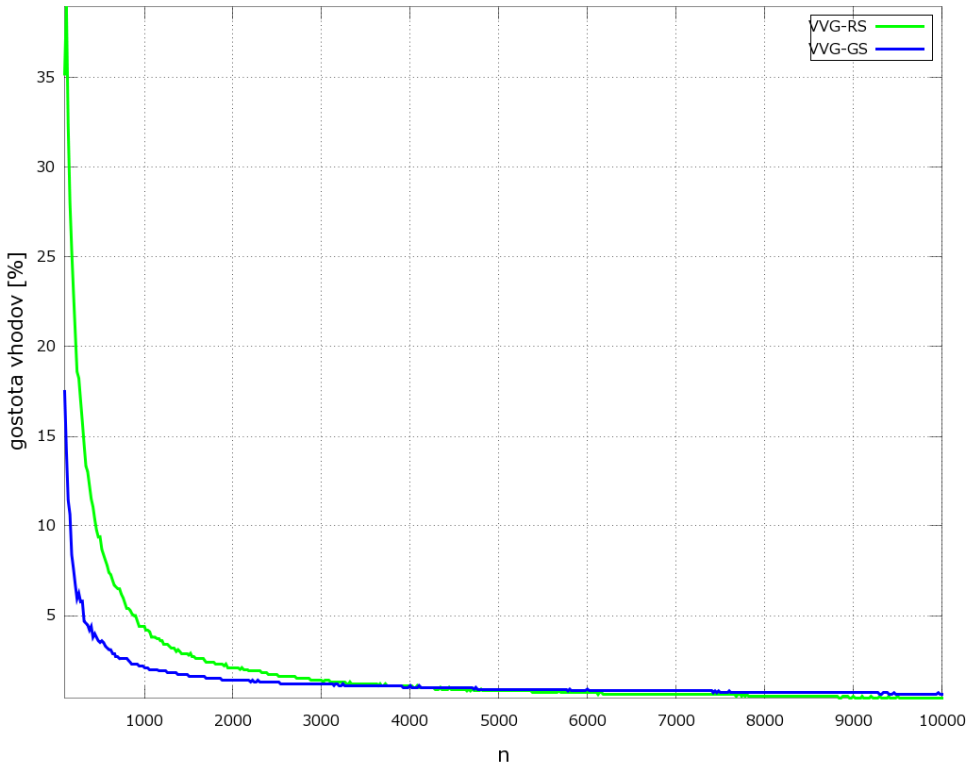
Pri testiranju smo opazovali obnašanje vseh treh implementacij pri vhodnih matrikah, katerim smo spreminjali gostoto neničelnih elementov in razsežnost n.

Gostota, kjer je čas izvajanja na GPE enak času izvajanja na CPE

Pri izvajanju algoritmov smo opazili, da je do določene gostote vhodnih matrik hitrejši zaporedni algoritem, pri višjih gostotah pa vzporedni različici algoritma. Ta gostota je odvisna od n:

- Pri $n = 100$ algoritem VVG-RS prehiti VVC pri gostoti 49.8% , VVG-GS pa pri 21%.
- Pri $n = 500$ algoritem VVG-RS prehiti VVC pri gostoti 9.2%, VVG-GS pa pri 3.5%.
- Pri $n = 1000$ algoritem VVG-RS prehiti VVC pri gostoti 4.4%, VVG-GS pa pri 2.1%.
- Pri $n = 10000$ algoritem VVG-RS prehiti VVC pri gostoti 0.36%, VVG-GS pa pri 0.59%.
- Pri $n = 100000$ algoritem VVG-RS prehiti VVC pri gostoti 0.05%, VVG-GS pa pri tej razsežnosti ne moremo izvajati.

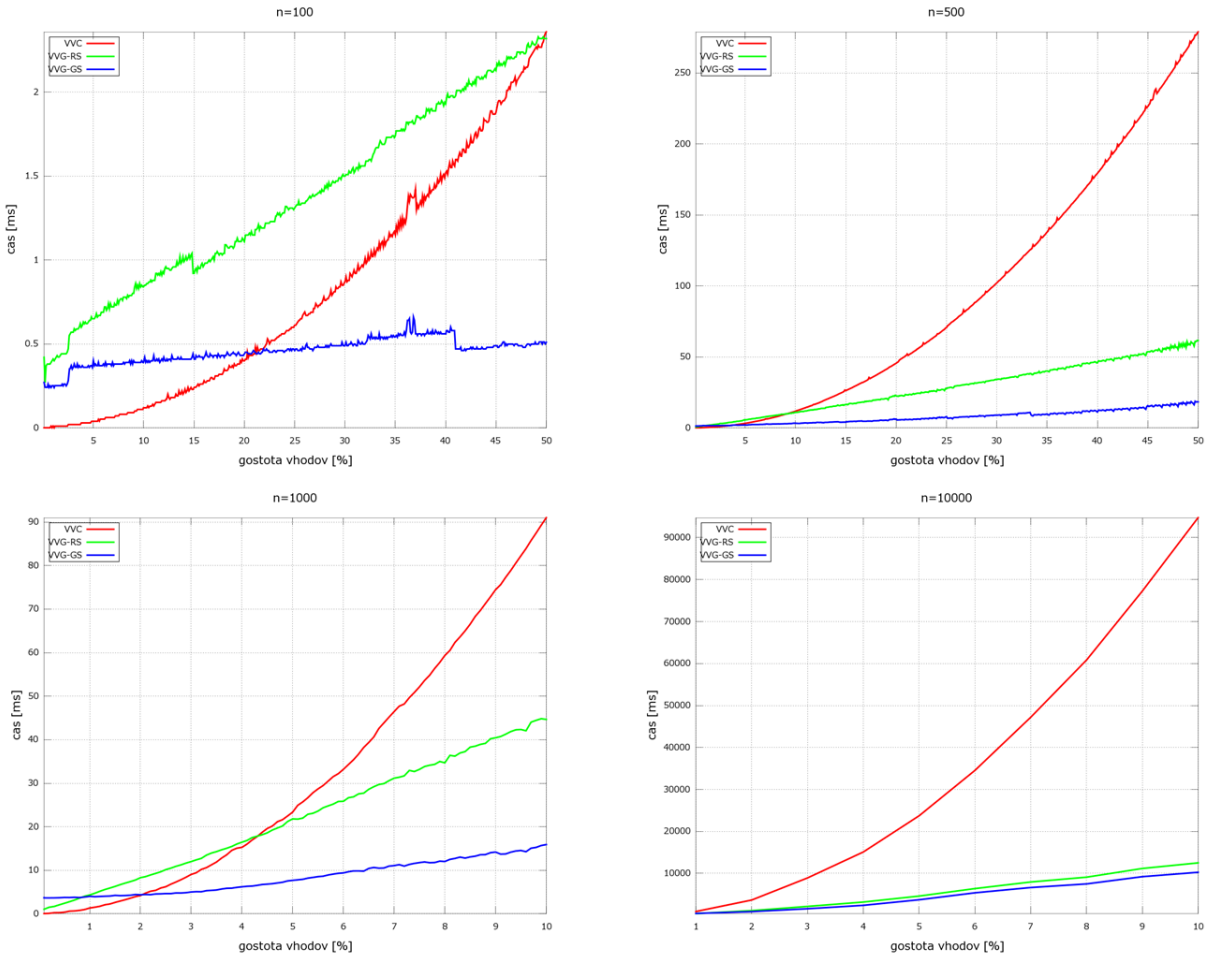
Graf na sliki 5 prikazuje gostoto, pri kateri se vzporedna algoritma izvedeta enako hitro kot zaporedni algoritem. Iz grafa je razvidno, da je na vhodnih matrikah manjših razsežnosti potrebna višja gostota neničelnih elementov, da se algoritem na GPE izplača. Takrat višjo gostoto zahteva algoritem VVG-RS. Z večanjem n mejni gostoti algoritmov padata ter se približujeta druga drugi. Pri večjih n pa malenkost višjo gostoto zahteva VVG-GS. Čas prenosa produkta z GPE algoritma VVG-GS z večanjem razsežnosti narašča, zato je za večjo učinkovitost algoritma potreben večji delež računskih operacij, ki se izvajajo vzporedno.



Slika 5: Graf gostote vhodnih matrik, pri kateri algoritem na GPE dohiti zaporedni algoritem, v odvisnosti od razsežnosti.

Čas izvajanja v odvisnosti od gostote

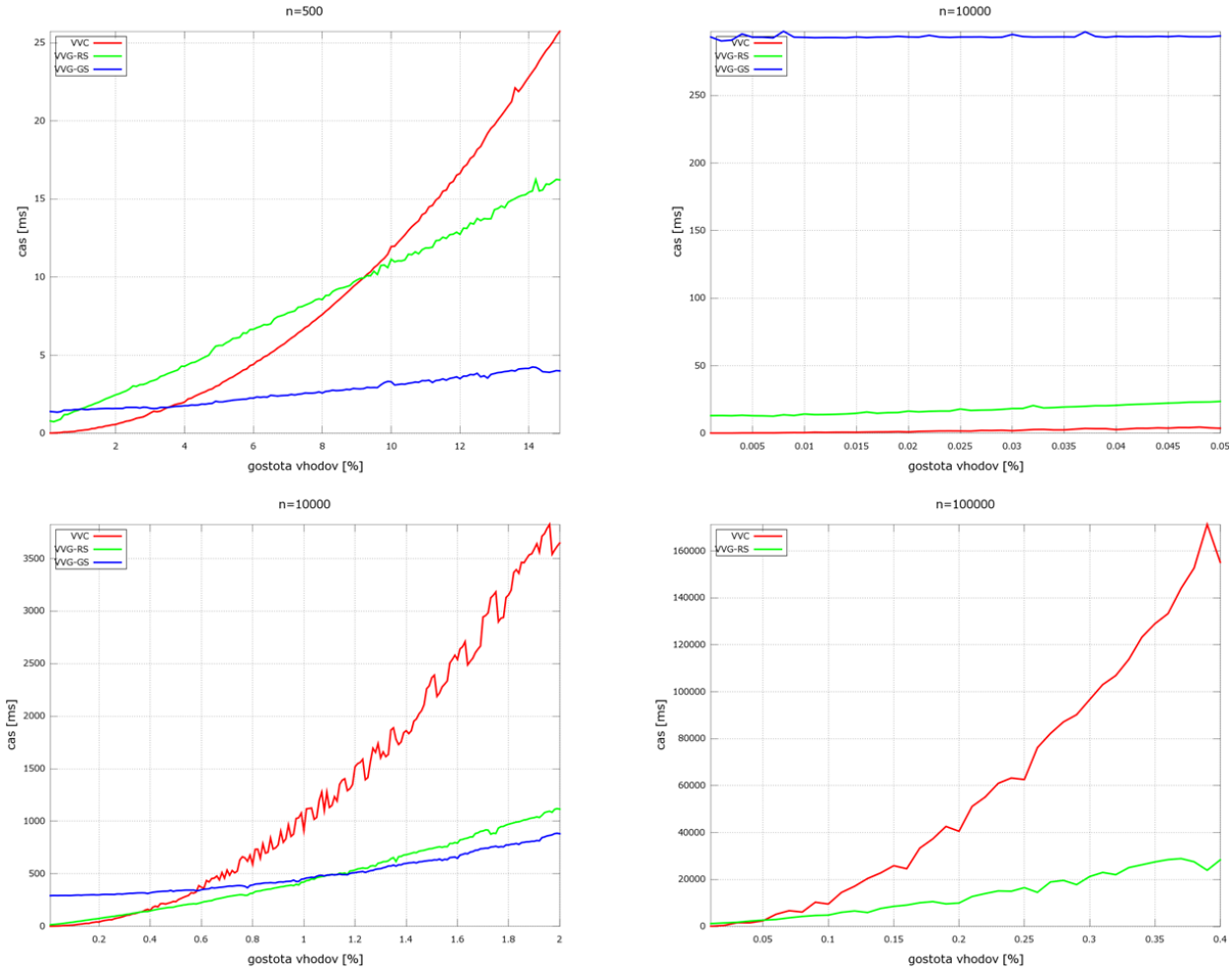
Grafi s slike 6 prikazujejo čas izvajanja algoritmov v odvisnosti od gostote vhodnih matrik pri fiksnih razsežnostih n. Iz splošnega obnašanja grafov je razvidno, da je pri nižjih gostotah hitrejši zaporedni, pri višjih pa vzporedna algoritma. Čas izvajanja zaporednega algoritma VVC z večanjem gostote vhodnih matrik strmo narašča, manj pa čas izvajanja vzporednih algoritmov, kar kaže na učinkovitost vzporednega izvajanja. Izkoristek vzporednega izvajanja pri nekem n je večji pri gostejših vhodnih matrikah. Pri večjih n pa je pohitritev pri določeni gostoti večja, kar lahko vidimo na grafih, ki prikazujeta odvisnost časa izvajanja pri gostoti do 10% za $n = 1000$ in $n = 10000$. Pri gostoti 10% in $n = 1000$ je pohitritev parelelnih algoritmov približno 2-kratna (VVG-RS) in 6-kratna (VVG-GS), pri $n = 10000$ pa že približno 9-kratna za oba vzporedna algoritma. Pri manjših n, npr. $n = 100$, pa se vzporedna algoritma izplačata šele pri matrikah, kjer je skoraj polovica (VVG-RS) oz. četrtina (VVG-GS) elementov neničelnih. Vzporedna algoritma sta torej uspešnejša na vhodnih matrikah višjih razsežnosti. Vzporedni algoritem VVG-GS se je pri meritvah do $n = 10000$ izkazal za hitrejšega od vzporednega algoritma VVG-RS. Pri višjih n pa ta algoritem ne deluje, saj smo omejeni s pomnilnikom GPE. Razlika v pohitritvi je opaznejša pri manjših n, pri $n = 10000$ pa je že dosti manjša.



Slika 6: Grafi časa izvajanja algoritmov v odvisnosti od gostote pri razsežnostih n=100 (levo zgoraj), n=500 (desno zgoraj), n=1000 (levo spodaj), n=10000 (desno spodaj).

Čas izvajanja pri nizkih gostotah

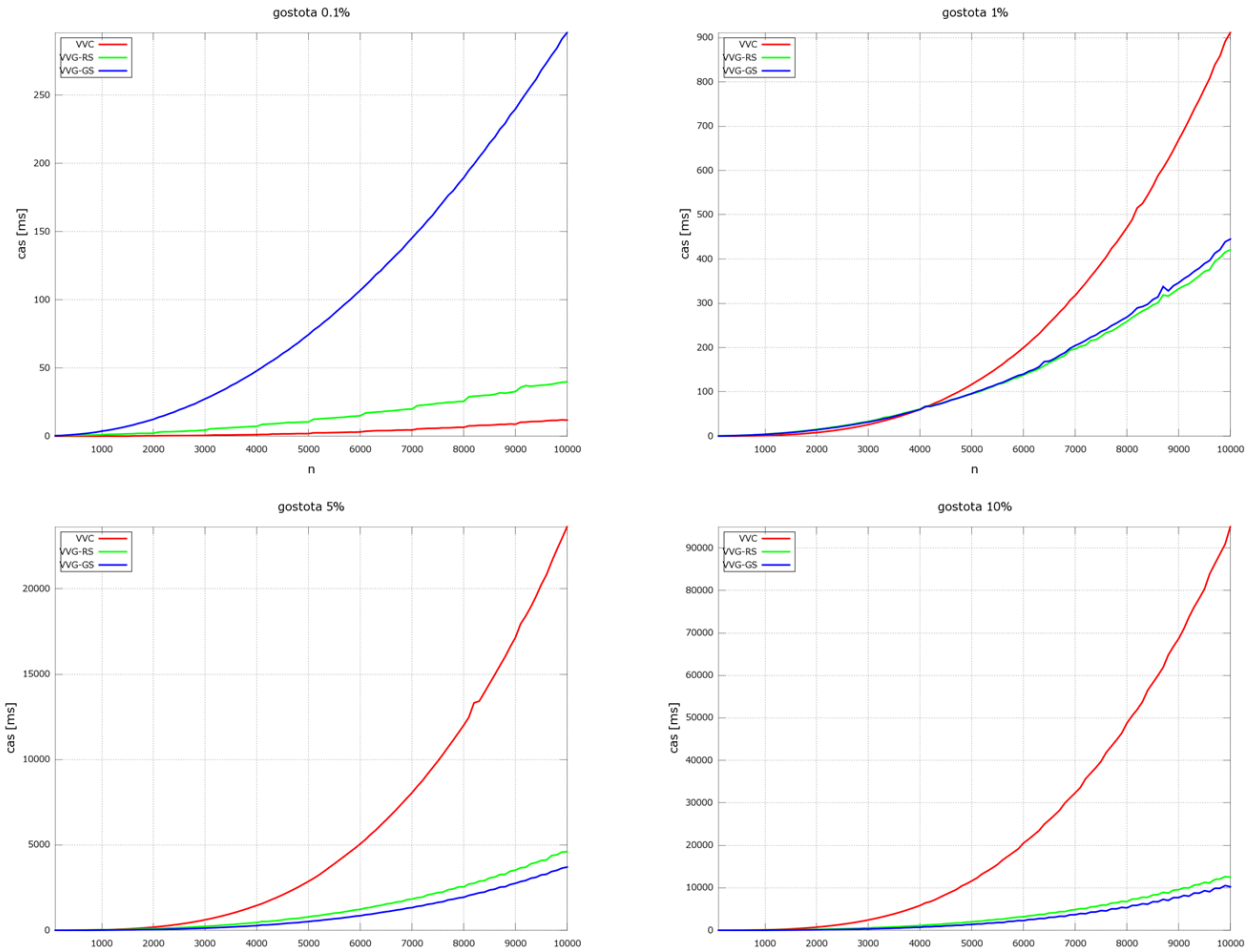
Grafi s slike 7 prikazujejo čas izvajanja algoritmov v odvisnosti od gostote pri fiksnih razsežnostih pri nekoliko nižjih gostotah vhodnih matrik. Pri nizkih gostotah se najhitreje izvaja zaporedni algoritem VVC. Pri $n = 10000$ in gostoti do 0.05% vidimo, da je zaporedna izvedba občutno najhitrejša. Tako redke matrike so sicer pogoste v zbirki matrik Matrix Market. Algoritem VVG-GS je pri zelo redkih matrikah najpočasnejši, saj velik časovni strošek predstavlja prenos produkta v gosti obliki iz GPE, vzporednih operacij množenja pa je relativno malo. Pri nekoliko višjih gostotah pa je ta algoritem že najhitrejši. Krivulja grafov časa izvajanja ni gladka – število delnih produktov niha z višanjem gostote, vendar se njihovo število dolgoročno povečuje. Zadnji graf s slike 6 prikazuje čas izvajanja pri $n = 100000$ v odvisnosti od gostote vhodnih matrik. Pri večjem n je pohitritev algoritma VVG-RS relativno velika že pri manjših gostotah vhodnih matrik (približno 5-kratna pri gostoti 0.4%). Ker algoritem shranjuje rezultat v redko matriko, smo pri tej implementaciji omejeni le številom neničelnih elementov vhodnih matrik in produkta. Algoritma VVG-GS pa pri takšni razsežnosti ne moremo izvajati.



Slika 7: Grafi odvisnosti časa izvajanja algoritmov od gostote pri razsežnostih $n = 500$ (levo zgoraj), $n = 10000$ (desno zgoraj in levo spodaj), $n = 100000$ (desno spodaj).

Čas izvajanja v odvisnosti od razsežnosti

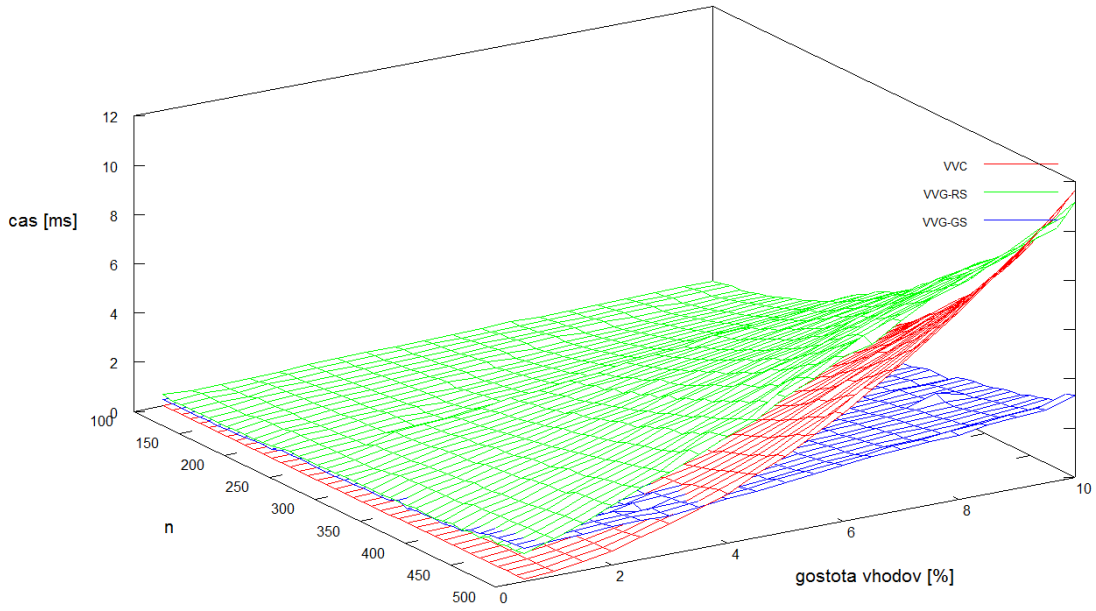
Grafi s slike 8 prikazujejo odvisnost časa izvajanja algoritmov od razsežnosti vhodnih matrik pri fiksnih gostotah. Vidimo, da je 0.1% premajhna gostota vhodnih matrik, da bi vzporedni implementaciji učinkovito delovali, saj je pri razsežnostih vse do $n = 10000$ zaporedni algoritem najhitrejši. Pri gostoti 1% vzporedna algoritma že delujeta hitreje od zaporednega pri razsežnosti $n = 4000$ in naprej, pri gostoti 5% pa sta hitrejši še pri manjšem n (približno $n = 900$ za VVG-RS in $n = 400$ za VVG-GS). Pri 10% gostoti pa za večjo uspešnost vzporednih algoritmov zadostujejo zgolj razsežnosti $n = 500$ (VVG-GS) in $n = 200$ (VVG-RS). Pohitritev vzporednih algoritmov je pri gostejših vhodnih matrikah višja. Množenje takrat zahteva več delnih produktov, ki se izvedejo vzporedno.



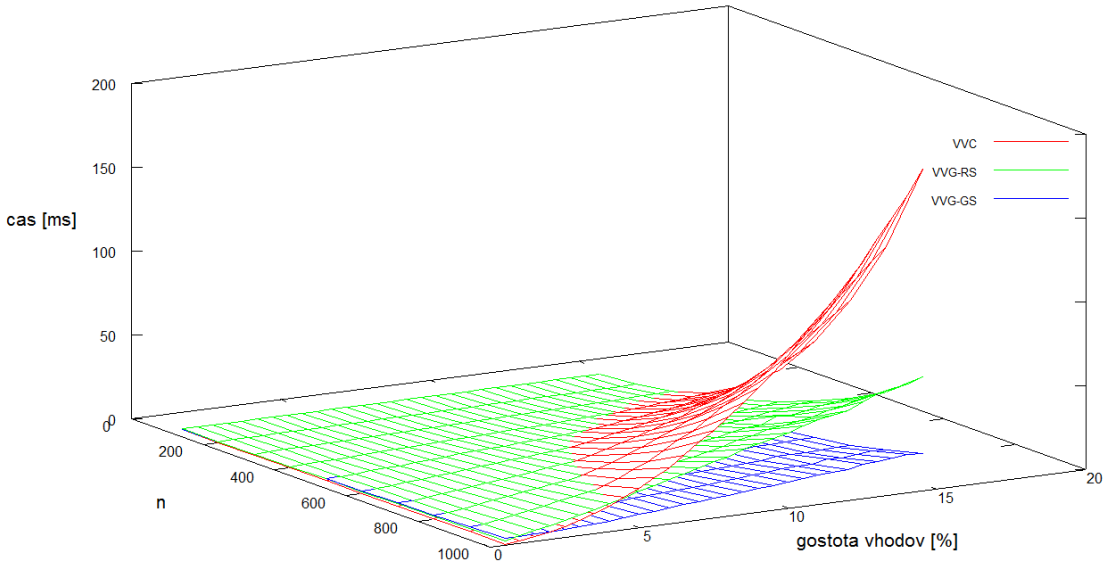
Slika 8: Grafi časov izvajanja algoritmov odvisnosti od razsežnosti vhodnih matrik n pri fiksni gostoti vhodnih matrik 0.1% (zgoraj levo), 1% (zgoraj desno), 5% (spodaj levo) in 10% (spodaj desno).

3D prikaz časa izvajanja v odvisnosti od gostote in razsežnosti

Na trirazsežnostnem prikazu s slike 9 vidimo potek časa izvajanja v odvisnosti od razsežnosti in gostote vhodnih matrik algoritmov VVC, VVG-RS in VVG-GS. Graf s slike 10 pa je nadaljevanje istega grafa z večjim razponom gostote in razsežnosti vhodnih matrik. Z grafa vidimo značilnosti algoritmov, ki smo jih opazili že pri dvorazsežnostih grafih. Z večanjem gostote vhodnih matrik se pohitritev vzporednih implementacij viša, saj čas njune izvedbe narašča počasneje od zaporednega algoritma. Prav tako je pri določeni gostoti vhodov izkoristek vzporednega izvajanja višji pri večjih razsežnostih. Ploskve pa se sekajo tam, kjer so algoritmi enako hitri.



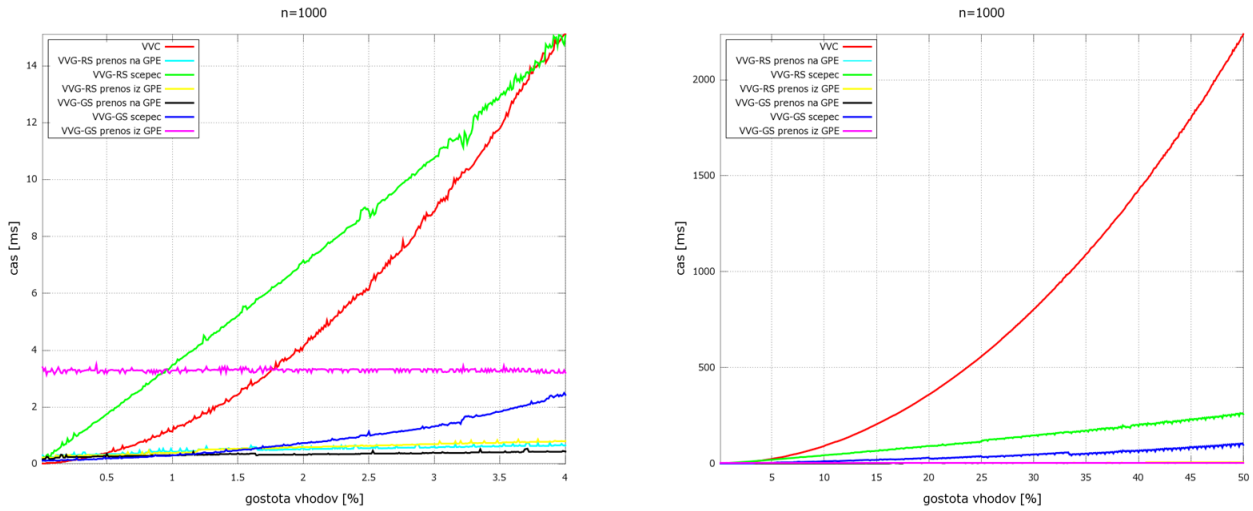
Slika 9: 3D graf predstavlja odvisnost časa vseh treh algoritmov od gostote in razsežnosti n.



Slika 10: 3D graf predstavlja odvisnost časa vseh treh algoritmov od gostote in razsežnosti n in je nadaljevanje grafa na sliki 9.

6.3.2 Časovni deleži posameznih delov algoritmov

Pri nižjih gostotah vhodov (slika 11, levi graf) je čas ščepca algoritma VVG-RS višji od časa izvedbe zaporednega algoritma VVC. Atomske operacije predstavljajo velik del časa ščepca, operacij množenja, ki jih lahko izvajamo vzporedno, pa je manj. Algoritem VVG-GS izvede ščepec hitro, vendar traja prenos produkta dlje časa, saj prenašamo celotno matriko. Na desnem grafu vidimo, da se z višjimi gostotami čas zaporednega algoritma strmo veča. Vzporedna algoritma sta zato učinkovita od določene gostote vhodnih matrik naprej. Ščepec algoritma VVG-GS takrat predstavlja večji delež celotnega časa in izvajanje je bolj učinkovito. Tudi ščepec VVG-RS se z višanjem gostote izvaja hitreje, vendar je počasnejši od VVG-GS. Čas izvajanja obeh ščepcev se z večanjem gostote povečuje manj strmo od časa izvajanja zaporedne izvedbe, torej je vzporedno izvajanje učinkovito; ozko grlo obeh različic predstavlja shranjevanje rezultata. Prenosi vhodnih matrik na GPE pa zasedajo le majhen delež časa.



Slika 11: Trajanje posameznih delov algoritmov v odvisnosti od gostote vhodov pri $n = 1000$.

Kljub vsemu pa uspešnost vzporednega algoritma ne moremo natančno napovedati, saj je uspešnost odvisna tudi od sestave produkta. Ščepec se bo izvajal uspešno, če bo množenje matrik vsebovalo veliko delnih produktov, ki jih je mogoče izvajati vzporedno. Pri gostejših vhodnih matrikah je to navadno res, ni pa pogoj. Prav tako je izvajanje uspešnejše, če bodo delni produkti enakomerno razporejeni med vsemi snopi, ki se izvajajo. V testiranju smo množili vhodne matrike z enakomerno razporejenimi elementi in tako dobili tudi enakomerneje porazdeljeno matriko produkta. Trajanje ščepca in s tem uspešnost celotnega algoritma pa je težko napovedati na primeru poljubnih vhodnih matrik.

Poglavje 7

7 Zaključek

V diplomskem delu smo opisali in implementirali algoritma za množenje redkih matrik ter ju tudi prilagodili za vzporedno izvajanje na GPE z arhitekturo CUDA. Pri tem smo uporabili različne pristope pri shranjevanju produkta ter sami izvedbi vrstičnega algoritma. Izkazalo se je, da je vzporedni vrstični algoritem za množenje redkih matrik pod določenimi pogoji hitrejši od zaporedne izvedbe. V primerjavi z različico, ki namesto snopa niti za računanje vrstice uporablja posamezno nit, se je pristop s snopi izkazal kot hitrejši. Tako zaporedna kot vzporedna izvedba vrstično-stolpičnega algoritma za množenje redkih matrik pa sta delovali občutno počasneje, čeprav je slednja hitrejša od zaporedne.

Čeprav se sam računski del vzporednega vrstičnega algoritma izvaja hitreje od zaporedne izvedbe, se problem pri izvajanju množenja redkih matrik na GPE izkaže pri shranjevanju produkta. Pristop, kjer rezultat shranimo in prenašamo kot gosto matriko, je omejen na manjše razsežnosti matrik, prav tako pa velik čas izvajanja algoritma porabimo za prenos rezultata iz GPE. Uporaba atomskih operacij prihrani na času prenosa, saj prenašamo redko matriko, prav tako pa nismo omejeni z razsežnostjo produkta. Izkaže pa se, da so same atomske operacije časovno zelo zahtevne in lahko opazno povečajo čas izvajanja ščepca. Čas je odvisen predvsem od števila neničelnih produktov matrike, katerega v večini primerov množenja matrik ne poznamo, zato se lahko čas takšnega pristopa zelo spreminja. V splošnem se je za hitrejšega izkazal pristop z gostim produktom. Omeniti je potrebno tudi, da imamo pri takšnem načinu shranjevanja produkt že urejen in ni potrebe po dodatnem urejanju.

Pri testiranju vzporednih vrstičnih algoritmov smo opazili, da na učinkovitost vzporednih implementacij vpliva tudi gostota in razsežnost vhodnih redkih matrik. Na zelo redkih vhodnih matrikah je vzporedni vrstični algoritem deloval počasneje v primerjavi z zaporednim. Pri manjših vhodnih matrikah je za učinkovito delovanje potrebna višja gostota kot pri večjih vhodnih matrikah. Na vhodnih matrikah iz spletne zbirke so se algoritmi obnašali slabše, saj so relativno redke. Z generiranimi vhodnimi matrikami pa se je od neke gostote naprej vzporedni vrstični algoritem izvajal hitreje od zaporednega. Na to gostoto pa vpliva razsežnost vhodnih matrik.

Zaključimo lahko, da je uspešnost vzporednega množenja matrik težko napovedati, saj je ta precej odvisna od lastnosti vhodnih redkih matrik, kot so velikost, gostota in sama struktura. Pri tem se lahko pri določanju uspešnosti do neke mere opremo na gostoto vhodnih matrik. Ta naj bo za uspešno delovanje pri večjih razsežnostih vhodnih matrik višja. Pri zelo redkih vhodnih matrikah, sploh manjših razsežnosti, vzporedni algoritem ne bo hiter. Na uspešnost vzporednih algoritmov vpliva tudi GPE. Če imamo na voljo zadosti globalnega, produkt shranjujemo v gosti obliki, saj takšna različica deluje hitreje pri redkejših vhodnih matrikah. Še vedno pa smo tudi s shranjevanjem samo neničelnih elementov produkta omejeni z globalnim pomnilnikom GPE. Kaj kmalu ga lahko zapolnimo z neničelnimi elementi vhodnih matrik.

7.1 Nadaljnje delo

Z nadaljnjim delom lahko rešitev omejenosti z globalnim pomnilnikom morda iščemo v bločnem vzporednem množenju redkih matrik, kjer bi vsak blok obravnavali kot posamezno vzporedno vrstično množenje redkih matrik, vendar bi bilo pri takem načinu veliko prenosov med GPE in CPE. Zanimivo pa bi bilo raziskati tudi algoritme za množenje strukturiranih redkih matrik (npr. diagonalne, tridiagonalne, pasovne matrike) ter njihove vzporedne izvedbe.

Literatura

[1] Fred G. Gustavson, *Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition*, v ACM Transactions on Mathematical Software, vol 4, no. 3, pp. 250-269, 1978.

[2] Jack J. Dongarra and others, *Graph Algorithms in the Language of Linear Algebra*, Society for Industrial and Applied Mathematics, Philadelphia, 2011, str. 3-12.

[3] Julien Demouth, *Sparse Matrix-Matrix multiplication on GPU*, predstavitev s konference *GPU Techology Conference*, NVIDIA, 2012. Dostopno na: http://on-demand.gputechconf.com/gtc/2012/presentations/S0285-GTC2012-Sparse-Matrix-Multiplication.pdf

[4] Yousef Saad, *Iterative Methods for Sparse Linear Systems*, Society for Industrial and Applied Mathematics, Philadelphia, 2003, str. 75-79.

[5] Ivan Vidav, *Algebra*, DMFA - založništvo, Ljubljana, 2003, str. 123 – 126.

[6] Aydın Buluç, *Linear Algebraic Primitives for Parallel Computing on Large Graphs*, PhD thesis, University of California, Santa Barbara, 2010, str. 23-63.

[7] Kiran Kumar Matam, Siva Rama Krishna Bharadwaj, Kishore Kothapalli, *Sparse Matrix Matrix Multiplication on Hybrid CPU+GPU Platforms*, Center for Security, Theory and Algorithmic Research (CSTAR), Hyderabad, 2012.

[8] NVIDIA Corporation, *CUDA C Programming Guide*, version 5.0, 2012.

[9] Vrstično-stolpični algoritem. Dostopno na: http://ellard.org/dan/www/Q-97/HTML/root/node20.html#smmAlg.

[10] Zapisi redkih matrik. Dostopno na: http://netlib.org/utk/papers/templates/node91.html

[11] Zbirka redkih matrik Matrix Market. Dostopno na: http://math.nist.gov/MatrixMarket/

[12] Programska knjižnica za pretvarjanje med zapisi. Dostopno na: http://crd-legacy.lbl.gov/~yunhe/cs267/final/source/utils/convert/

[13] Specifikacija GPE Tesla C1060. Dostopno na: http://www.nvidia.com/docs/IO/56483/Tesla_C1060_boardSpec_v03.pdf