



Univerzitet u Beogradu  
Elektrotehnički fakultet  
Modul Softversko inženjerstvo

## MASTER RAD

**Optimizacija Needleman-Wunsch algoritma zasnovana  
na obradi dijagonalne pločice na grafičkom procesoru**

**MENTOR**

dr Marko Mišić, vanr. prof.  
[marko.misic@etf.bg.ac.rs](mailto:marko.misic@etf.bg.ac.rs)

**KANDIDAT**

Marko Stanojević  
indeks 2022/3037

Beograd, jun 2025. godine

# Sadržaj

<b>1</b>	<b>Uvod.....</b>	<b>5</b>
<b>2</b>	<b>Poravnanje bioloških sekvenci.....</b>	<b>6</b>
2.1	Sekvence aminokiselina.....	6
2.2	Needleman-Wunsch algoritam.....	6
<b>3</b>	<b>Arhitektura i organizacija centralnog i grafičkog procesora .....</b>	<b>9</b>
3.1	Arhitektura i organizacija CPU-a.....	9
3.2	Arhitektura GPU-a.....	9
3.3	Organizacija GPU-a.....	11
3.4	Optimizacija kernela.....	12
3.4.1	<i>Global memory coalescing .....</i>	12
3.4.2	Warp divergencija.....	12
3.4.3	Warp occupancy .....	13
3.4.4	<i>Register spill.....</i>	13
3.4.5	CUDA tokovi.....	13
3.5	Pregled korišćenih tehnologija i alata.....	13
<b>4</b>	<b>Implementacija algoritama sa poboljšanim performansama .....</b>	<b>16</b>
4.1	Pretpostavke algoritama.....	16
4.2	Objašnjenje skraćenica u nazivima algoritama.....	17
4.3	Cpu1-St-Row i Trace1-Plain.....	18
4.4	Cpu2-St-Diag algoritam.....	19
4.5	Cpu3-St-DiagRow algoritam.....	19
4.6	Cpu4-Mt-DiagRow algoritam .....	20
4.7	Gpu1-Ml-Diag algoritam.....	21
4.8	Gpu2-Ml-DiagRow2Pass algoritam.....	21
4.9	Gpu3-Ml-DiagDiag algoritam .....	22
4.10	Gpu4-Ml-DiagDiag2Pass algoritam .....	23
4.11	Gpu5-Coop-DiagDiag algoritam .....	23
4.12	Gpu6-Coop-DiagDiag2Pass algoritam .....	24
4.13	Gpu7-Mlsp-DiagDiag i Trace2-Sparse algoritam .....	24
4.14	Gpu8-Mlsp-DiagDiag algoritam.....	26

4.15 Gpu9-Mlsp-DiagDiagDiag algoritam .....	28
<b>5 Pregled postojećih rešenja .....</b>	<b>29</b>
5.1.1 <i>CUDASW++4.0: ultra-fast GPU-based Smith–Waterman protein sequence database search.....</i>	29
5.1.2 <i>Exact pairwise alignment of mega-base genome biological sequences using a novel z-align parallel strategy.....</i>	30
5.1.3 <i>SW#—GPU-enabled exact alignments on genome scale .....</i>	31
<b>6 Rezultati .....</b>	<b>32</b>
6.1 Metodologija .....	32
6.2 CPU-specifične metrike.....	32
6.3 GPU-specifične metrike .....	36
<b>7 Zaključak i dalji rad.....</b>	<b>40</b>
<b>8 Reference .....</b>	<b>41</b>

## Slike

Slika 1 – proširena skor matrica $\mathcal{A}$ i sekvene $\vec{x}$ i $\vec{y}$ .....	8
Slika 2 – optimalna sekvenca izmena.....	8
Slika 3 – pojednostavljen dijagram GPU čipa i memorija.....	10
Slika 4 – mapiranje grid-a, bloka i warp-a niti na SM-ove .....	10
Slika 5 – leva polovina dijagrama GA102 arhitekture ( <i>RTX3090</i> ) .....	14
Slika 6 – SM u arhitekturi GA102 .....	15
Slika 7 – obilazak elemenata po (sporednoj) dijagonali .....	20
Slika 8 – obilazak pločica po dijagonali i elemenata po vrsti.....	20
Slika 9 – obilazak pločica po dijagonali i elemenata same pločice po dijagonali .....	23
Slika 10 – retka reprezentacija skor matrice – <i>tileHrowMat</i> .....	25
Slika 11 – retka reprezentacija skor matrice – <i>tileHcolMat</i> .....	25
Slika 12 – tri moguća pravca prelaska u narednu pločicu tokom rekonstrukcije poravnjanja trenutne .....	25
Slika 13 – čuvanje četiri elemenata u registrima jedne niti .....	27
Slika 14 – podela pločice (pravougaonik) na podpločice oblika paralelograma.....	28
Slika 15 – mapiranje niti u <i>CUDASW++4.0</i> na elemente skor matrice.....	30

Slika 16 – podela dugačke sekvene iz baze na delove od  $k$  kolona.....30

## Tabele

Tabela 1 – maksimalan broj tera operacija u sekundi na <i>RTX3090</i> .....	16
Tabela 2 – pregled algoritama za poravnavanje u radu.....	17
Tabela 3 – pregled algoritama za rekonstrukciju poravnjanja u radu.....	18
Tabela 4 – operacija izmene i odgovarajuće pomeranje tokom rekonstrukcije poravnjanja.....	19

## Grafići

Grafik 1 – <i>end-to-end</i> vreme izvršavanja po algoritmu.....	33
Grafik 2 – ubrzanje vremena izvršavanja u odnosu na Cpu1 algoritam po algoritmu.....	33
Grafik 3 – CPU keš promašaji po algoritmu.....	34
Grafik 4 – maksimalna potrošnja RAM memorije po algoritmu .....	34
Grafik 5 – ideo faza u vremenu izvršavanja po algoritmu .....	35
Grafik 6 – warp-iskorišćenost SM-a po GPU kernelu .....	36
Grafik 7 – broj warp ciklusa po izvršenoj instrukciji po GPU kernelu .....	37
Grafik 8 – prosečan broj aktivnih niti po warp-u po GPU kernelu.....	37
Grafik 9 – broj instrukcija skoka po GPU kernelu.....	38
Grafik 10 – odnos broja instrukcija skoka sa ukupnim brojem instrukcija po GPU kernelu ...	38
Grafik 11 – potrošnja energije po GPU kernelu.....	39

# 1 Uvod

Poravnavanje sekvenci je fundamentalan problem bioinformatike, čemu svedoči ogromna količina podataka koja se poravnava u svakom trenutku i veliki broj algoritama razvijenih u tu svrhu. Ovaj rad se bavi Needleman-Wunsch algoritmom [1], jednim od prvih za poravnavanje sekvenci, koji ima primenu i danas kada je potrebno globalno poravnanje najvišeg kvaliteta.

Razmatraju se različite optimizacije paralelizacije Needleman-Wunsch algoritma za poravnavanje parova dugačkih sekvenci i određivanje optimalne sekvence izmena (engl. *edit trace-a*). Kao polazna tačka za poređenje performanse se koristi najjednostavniji CPU algoritam. Prvih nekoliko algoritama paralelizuju izračunavanje na CPU-u, a preostali na GPU-u. Performanse implementacija se poredaju detaljnije u diskusiji.

Dosadašnje GPU tehnike paralelizacije koriste podelu skor matrice na pločice, i koriste obilazak sporedne dijagonale pločica (engl. *minor diagonal*) i elemenata pločice [2][3], čak i paralelizam na nivou *warp-ova* [4]. U ovom radu je izložena nova tehnika za poravnanje parova veoma dugačkih sekvenci, gde se pločica deli na podpločice oblika paralelograma na specifičan način, koji se obilaze takođe dijagonalno.

U nastavku rada će biti:

1. detaljnije opisan problem poravnanja sekvenci Needleman-Wunsch algoritmom,
2. dat pregled arhitekture i organizacije GPU-a i CPU-a, kao i pregled tehnologija i alata korišćenih u radu,
3. dat pregled optimizacija isprobanih na Needleman-Wunsch algoritmu,
4. dat pregled algoritama iz drugih radova,
5. analizirane performanse izloženih algoritama,
6. izведен zaključak o rezultatima.

## 2 Poravnanje bioloških sekvenci

Uputstva za izgled i funkcije svih živih bića, u praktično neizmenjenoj formi još od nastanka života na planeti do danas, su hemijski lanci nukleotida (DNK i RNK) i aminokiselina (proteini). Da bi se ustanovilo koje vrste živih bića su međusobno slične, koje karakteristike je jedinka nasledila od pretka a koje je dobila mutacijom, predvidele karakteristike organizma u budućnosti, i dali odgovori na slična pitanja, treba uporediti (poravnati) ove sekvence.

Do sada je razvijen veći broj algoritama i optimizacija za poravnanje sekvenci u zavisnosti od namene. Jedna podela algoritama je na globalne, lokalne i lokalno-globalne algoritme [5]. Kod globalnih algoritama, pretpostavka je da su sekvene veoma slične, i cilj je ustanoviti optimalne biološke korake kako se od jedne sekvene može dobiti druga. Kod lokalnih algoritama, smatra se da su sekvene različite ali da imaju podsekvenci koje su slične, a cilj je odrediti te podsekvence. Lokalno-globalni algoritmi su hibridi prethodne dve vrste, koji traže optimalno parcijalno poravnanje sekvenci. Korisni su u slučaju da je početak jedne sekvene isti kao kraj druge sekvene, ili ako se dve sekvene drastično razlikuju u dužini, a kraću treba optimalno poravnati u odnosu na dužu, primer čega je poravnanje gena u hromozomu.

Broj kratkih sekvenci koje se poravnavaju u današnjici se meri u stotinama miliona po uzorku. Takođe, mogući zahtev je poravnati dve ekstremno dugačke sekvene, ukoliko je kvalitet poravnanja primaran. U interesu bioinformatike je obradu raditi u što kraćem vremenu, sa što manjom potrošnjom resursa.

### 2.1 Sekvence aminokiselina

Proteini su lanci molekula, koji se sastoje od manjih molekula aminokiselina međusobno povezanih peptidnim vezama, i to specifičnim redosledom koji u velikoj meri diktira njihovu funkciju. Ukoliko dva proteina imaju identičnu sekvenu aminokiselina, skoro sigurno se radi o proteinima identičnih svojstava<sup>1</sup>.

Poznate su 22 različite aminokiseline koje se nalaze u osnovi proteina svog poznatog živog sveta [6]. Zbog toga, aminokiseline se mogu enkodovati kao slova abecede, odnosno kodni brojevi. Protein se tada može posmatrati kao vektor kodova aminokiselina.

### 2.2 Needleman-Wunsch algoritam

Algoritam kojim se ovaj rad bavi je Needleman-Wunsch, jedan od prvih za globalno poravnanje sekvenci, objavljen 1970. godine od strane Saul B. Needleman-a i Christian D. Wunsch-a [1]. Ovaj algoritam se i danas koristi kada je potrebno da globalno poravnanje sekvenci bude najvišeg kvaliteta. Algoritam koristi dinamičko programiranje – u svojoj suštini izračunava jednu ili više matrica pravougaonog oblika, gde se vrednost polja tabele izračunava na osnovu prethodno izračunatih suseda.

Sekvene aminokiselina  $\vec{x}$  i  $\vec{y}$  se reprezentuju kao vektori dužina  $n$  i  $m$  brojeva 0-24 (22 aminokiseline i dodatni simboli). Cilj je izračunati najbolju cenu poravnanja – odnosno meru sličnosti

---

<sup>1</sup> Bitan faktor koji utiče na svojstva proteina je njegova savijenost u 3D prostoru, odnosno sekundarna, tercijarna i kvaternarna struktura. Većina proteina u prirodi se savija na jednoznačan način.

sekvenci – gde se od sekvene  $\vec{y}$  dobije sekvena  $\vec{x}$  primenom niza operacija izmene (engl. *edit* operacija). Računa se kao prosta suma cena operacija u tom nizu.

Postoji više šema, odnosno skor funkcija, kojima se određuju cene samih operacija. U radu se koristi linearna skor funkcija [1]. U tom slučaju se izračunava tačno jedna pravougaona matrica – skor matrica  $\mathcal{A}$  (slika 1). Tada su operacije izmene:

1. zamena (engl. *substitute*) – gde se element  $y_i$  menja elementom  $x_j$  prema ceni  $\mathcal{S}(y_i, x_j)$ ; u slučaju poklapanja (*match-a*) cena je pozitivna, a nepoklapanja (*mismatch-a*) nula ili negativna,
  2. praznina (*gap open*) u  $\vec{x}$  – gde se otvara praznina na poziciji  $x_j$  po ceni  $g_o$  (negativna),
  3. praznina (*gap open*) u  $\vec{y}$  – gde se otvara praznina na poziciji  $y_i$  po ceni  $g_o$  (negativna),
- gde je  $\mathcal{S}$  supstitucionu matricu dimenzija  $25 \times 25$ .

U ovom radu je korišćena BLOSUM62 supstitucionu matricu, koja pripada grupi BLOSUM matrica [7] (engl. *Block Substitution Matrix*). Ove matrice su konstruisane na osnovu poravnaja visoko očuvanih regionalnih (blokova) proteinskih sekvenci iz BLOCKS baze podataka, pri čemu su analizirani delovi bez prisustva praznina. Supstitucioni skorovi su izračunati kao  $\log_2$  odnos između realne i očekivane frekvencije supstitucije parova aminokiselina. BLOSUM62 matrica je dobijena na podskupu poravnanja sekvenci iz BLOCKS baze podataka, pri čemu su sekvene sa sličnosti 62% ili više grupisane i posmatrane kao jedna, kako bi se korigovala pristrasnost usled prisustva ovakvih grupa.

Skor matrica  $\mathcal{A}$  je dimenzije  $m \times n$ , gde svako polje  $a_{ij}$  predstavlja optimalnu cenu poravnanja podsekvene  $\vec{x}[1:j]$  sa podsekvenom  $\vec{y}[1:i]$ . Polje  $a_{ij}$  se može izračunati tek kada su izračunata polja  $a_{i-1,j-1}$ ,  $a_{i-1,j}$  i  $a_{i,j-1}$ .

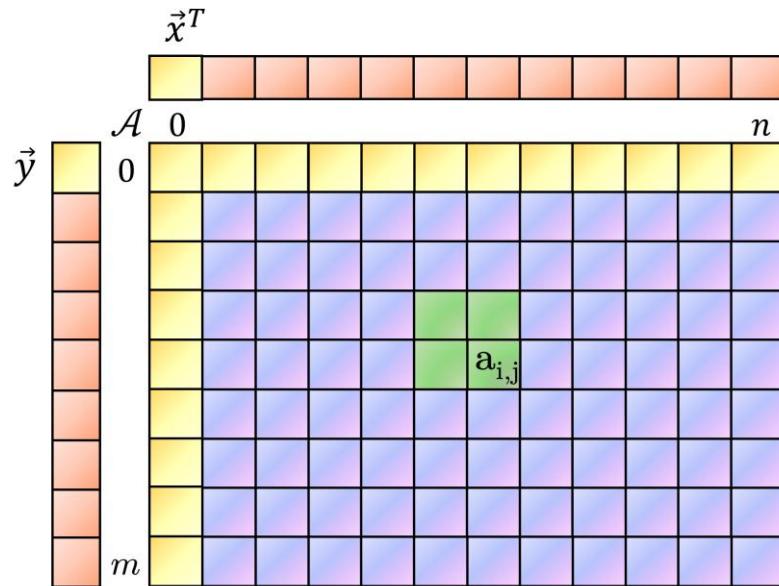
U cilju jednoobrazne obrade, skor matrici  $\mathcal{A}$  se dodaju nulta vrsta i nulta kolona gde element  $k$  ima vrednost  $k \cdot g_o$ , a vektorima  $\vec{x}$  i  $\vec{y}$  nulti element sa vrednosti 0 (slika 1).

Vrednost elementa  $a_{ij}$  se prema linearnej skor funkciji računa kao:

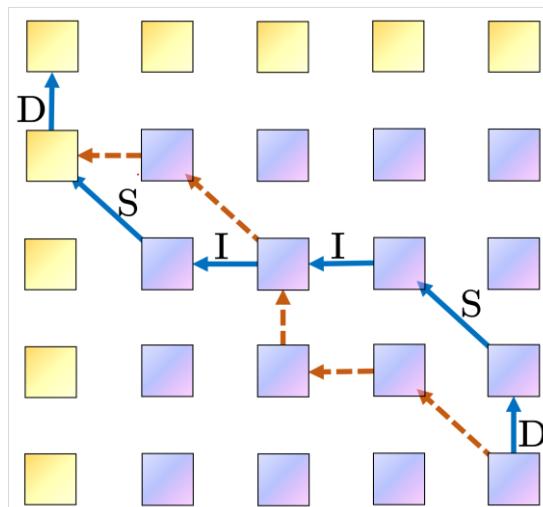
$$a_{ij} = \max \begin{cases} a_{i-1,j-1} + \mathcal{S}(y_i, x_j), & \text{poklapanje/nepoklapanje} \\ a_{i-1,j} + g_o, & \text{praznina u } \vec{x} \\ a_{i,j-1} + g_o, & \text{praznina u } \vec{y} \end{cases} \quad (1)$$

gde je  $g_o$  uzet kao  $-11$ . Kada se izračuna donje desno polje, algoritam je pronašao optimalnu cenu poravnanja  $\vec{y}$  na  $\vec{x}$ .

Moguće je rekonstruisati poravnanje (engl. *traceback*) – pronaći konkretne korake kako od  $\vec{y}$  dobiti  $\vec{x}$  po prethodno izračunatoj optimalnoj ceni (slika 2). Kao početna tačka se uzima donje desno polje sa optimalnom cenom poravnanja. Od trenutnog polja se kreće ka nekom od prethodnih – gore, levo, ili gore-levo – u zavisnosti od toga koje od njih je, u zbiru sa cenom operacije, dalo najbolji rezultat. Ako ima više takvih polja, može se odabratи bilo koje, jer sva leže na nekoj od optimalnih putanja. Čuvaju se primenjene operacije izmene. Kada se dođe do donjeg levog



Slika 1 – proširena skor matrica  $\vec{A}$  i sekvence  $\vec{x}$  i  $\vec{y}$ . Na slici je prikazan i element  $a_{ij}$  i njegova tri suseda na osnovu kojih se računa.



Slika 2 – optimalna sekvenca izmena (puna linija). Operacije su insercija (I, dijagonalna grana), delecija (D, vertikalna grana) i supstitucija (S, horizontalna grana) elemenata u  $\vec{y}$ . Smer čitanja sekvence izmena je odozgo nadole, sa leva na desno. Na slici ima više optimalnih sekvenci izmena (puna i isprekidana linija i njihove kombinacije), jedan od kojih je DSTISD.

polja, niz sačuvanih operacija se obrne, kako bi kao polaznu tačku imao gornji levi element skor matrice. Tako je dobijena optimalna sekvenca izmena (jedna od potencijalno mnogih).

### 3 Arhitektura i organizacija centralnog i grafičkog procesora

U nastavku će biti dat pregled CPU i GPU arhitektura i organizacije sa softverskog stanovišta, specifičnosti arhitektura koje utiču na performansu obrade, kao i pregled korišćenih tehnologija i alata. Poglavlje je napisano na osnovu Nvidia CUDA dokumentacije [8].

#### 3.1 Arhitektura i organizacija CPU-a

U današnje vreme CPU-ovi su veoma paralelni – tipičan CPU ima 4-16 jezgara i podržava *hyperthreading*, što znači da je broj hardverskih niti najčešće 8-32 po CPU-u. Takođe, CPU spekulativno izvršava mašinski kod koristeći prediktor grananja, i koristi veliki broj specijalizovanih keševa za ubrzavanje obrade, čime je prilagođen za brzo izvršavanje najrazličitijih problema, u velikoj meri nezavisno od frekvencije instrukcija skoka na putanji izvršavanja.

Svako jezgro CPU-a ima L0 instrukcijski keš i L0 data keš, kojem je vreme pristupa od strane CPU-a najbrže. CPU ima i L1 keš koji dele sva jezgra, koji je značajno veći od oba L0 keša ali malo sporiji za pristup. Dodatno, CPU ima pristup i L2 kešu koji je značajno veći od L1 keša i ubrzava pristup operativnoj memoriji, ali koji je najsporiji keš zbog svoje veličine i tehnologije izrade. Održavanje ovih keševa je automatsko, i oni su transparentni za programske kodove. Međutim, ukoliko je algoritam napisan tako da narušava pretpostavke keša, odnosno vremensku i prostornu lokalnost koda, on će imati drastično lošije performanse zbog kolizija keš linija.

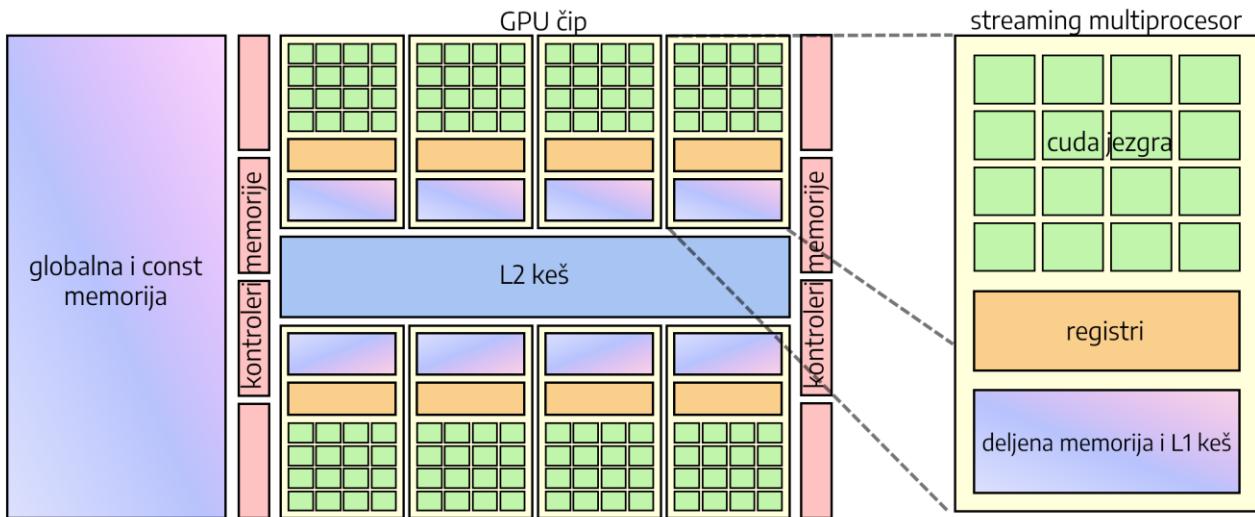
CPU ima pristup velikoj operativnoj (RAM) memoriji, u današnje vreme tipično 16-32 GB. Memorijski pristupi operativnoj memoriji tipično traju 100-300 CPU instrukcijskih ciklusa. Ukoliko neku strukturu podataka zbog njene veličine nije pogodno čuvati celu u operativnoj memoriji, može se smestiti na disk(ove), koji su najsporiji, ali čiji smeštajni prostor se može smatrati praktično neograničenim.

#### 3.2 Arhitektura GPU-a

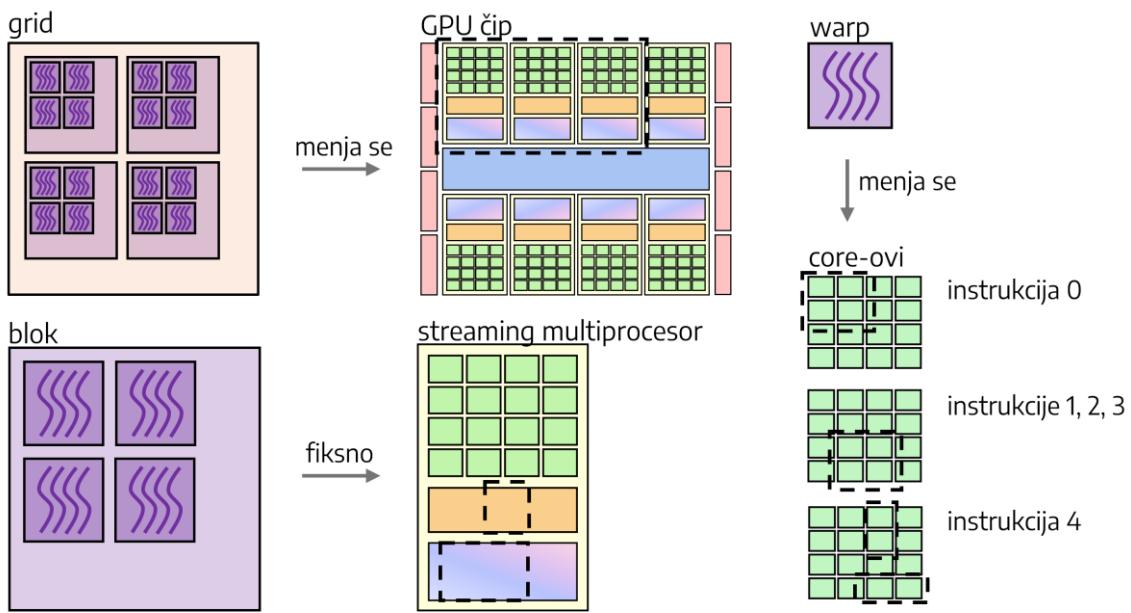
Kada je obrada dovoljno regularna, a potreba za ubrzanjem koda velika, smisleno je probati je na GPU-u. Tipičan GPU današnjice ima 1000-5000 CUDA jezgara, i daleko je paralelniji od modernog CPU-a za generalnu upotrebu.

GPU model izvršavanja je SIMT (*single instruction multiple threads*) – više hardverskih niti uvek izvršavaju istu instrukciju, ali nad različitim podacima. U CUDA arhitekturi 32 niti izvršavaju simultano istu instrukciju, odnosno u tzv. *lockstep*-u. Ta grupa se naziva *warp*.

*Streaming multiprocesor* (SM) je nezavisna osnovna jedinica paralelnog izvršavanja. Po tome je slična celom CPU-u. Tipična grafička kartica danas ih ima 20-80. SM sadrži CUDA jezgra, deljenu memoriju, kao i raznovrsna jezgra za specijalne namene. Samo CUDA jezgro sadrži aritmetičko-logičke jedinice za INT i FP operacije, registre, rasporedivač *warp*-ova, SIMT jedinice, itd. (slike 3, 5 i 6).



Slika 3 – pojednostavljen dijagram GPU čipa i memorija



Slika 4 – mapiranje grid-a, bloka i warp-a niti na SM-ove

SM-ovi imaju pristup globalnoj (video RAM, VRAM) memoriji, koja tipično ima 4-12 GB. U nju je potrebno prebaciti podatke potrebne za obradu pre izvršavanja, i rezultate iz nje prebaciti u RAM nakon obrade. Nekeširani memorijski pristupi globalnoj memoriji tipično traju 400-600 SM ciklusa, L2 keširani traju 200-300 ciklusa, dok L1 keširani traju 20-100 ciklusa.

SM sadrži deljenu memoriju, koja je drastično brža za pristup od globalne memorije, tipično 2 ciklusa na SM. Tipična veličina deljene memorije po SM-u je 48-96KB. Bitna razlika u odnosu na globalnu memoriju je da deljenoj memoriji na jednom SM-u mogu pristupiti jezgra samo sa istog SM-a. Može se posmatrati kao softverski održavani keš. Dobar način da se optimizuje protočnost programa je da se minimizuje pristup globalnoj memoriji. To se može postići čuvanjem podataka u SM deljenoj memoriji, imajući u vidu vremensku i prostornu lokalnost podataka.

SM sadrži L1 keš, koji kešira pristupe globalnoj memoriji, u cilju povećanja brzine pristupa od strane niti i smanjenja kontencije na GPU memorijskoj magistrali. Tipično vreme pristupa je 2 SM ciklusa, a veličina 16-48KB. Deljena memorija i L1 keš su hardverski deo iste memorije. Moguće je izabrati neku od dozvoljenih konfiguracija podele SM memorije između L1 keša i deljene memorije.

SM sadrži registre, tipično 256KB. Vreme pristupa im je 1 SM ciklus. Svakoj niti se efektivno dodeljuje identičan broj registara pre izvršavanja, do limita definisanog arhitekturom.

### 3.3 Organizacija GPU-a

Kernel je mašinski kod koji se izvršava na GPU-u [8]. Kernel se pokreće iz *host* koda (kod koji se izvršava na CPU-u), ali se izvršava kao *device* kod (kod koji se izvršava na GPU-u).

Nit je najmanja jedinica paralelnosti u GPU modelu izvršavanja. U okviru jednog pokretanja kernela, svaka nit izvršava identičan kod, baš taj od kernela.

Blok niti je grupacija niti u 1/2/3D rešetki, gde svaka nit ima jedinstven indeks (slika 4). Maksimalna veličina bloka je ograničena arhitekturom, najčešće 1024 niti po bloku. Dodatno, maksimalne dimenzije bloka su ograničene, tipično

$$(x_{max}, y_{max}, z_{max}) = (1024, 1024, 64) \quad (2)$$

Niti u bloku se mapiraju u *warp* redom po indeksima, prve 32 niti u prvi *warp*, itd. Ukoliko je blok 2/3D, indeksi niti se linearizuju po dimenziji *z*, a zatim po dimenziji *y*. Poželjno je odabratи broj niti u bloku tako da bude celobrojni umnožak 32 (broj niti u *warp*-u), kako poslednji *warp* ne bi imao niti koje ne rade ništa.

*Grid* blokova niti je grupacija blokova u 1/2/3D rešetki, gde svaki blok niti ima jedinstven indeks. Maksimalna veličina *grid*-a niti je tipično

$$(x_{max}, y_{max}, z_{max}) = (2^{31} - 1, 2^{16} - 1, 2^{16} - 1) \quad (3)$$

Kada se kernel prvi put pokrene sa *host*-a, na GPU se prebaci kernel kod, i zapamte se između ostalog dimenzije *grid*-a, dimenzije bloka i količina deljene memorije po bloku. Sledeća pokretanja kernela ne moraju prebacivati kernel kod ukoliko je prisutan u namenskom GPU kešu za kernel kodove.

Tokom izvršavanja, GPU blok raspoređivač posmatra SM-ove koji imaju dovoljno resursa za blok niti, i odabere jedan. SM dodeli potrebnu količinu resursa – registara, deljene memorije, itd., i pokreće izvršavanje niti u njemu. Jednom mapiran blok unutar jednog SM-a se ne može zameniti sa drugim kroz promenu konteksta, sve dok sve niti u bloku ne završe sa radom.

Blok niti se izvršava na nivou *warp*-a (slika 4). SM *warp* raspoređivač prati koji *warp*-ovi su spremni za izvršavanje, uzimajući u obzir sve blokove koji su mapirani na dati SM. Nespremni

*warp* se u nekom trenutku promenom konteksta zameni sa spremnim *warp*-om, na odgovarajuće CUDA jezgro, nakon čega spremni *warp* izvršava barem jednu instrukciju. Kada *warp* dođe do kraja kernel koda, tada je završio izvršavanje. Ideal kernel koda je da je više *warp*-ova spremno na izvršavanje u svakom trenutku, jer se time dobro sakriva čekanje.

Prilikom pisanja kernel koda treba podrazumevati da se blokovi niti i *warp*-ovi mogu izvršavati proizvoljnim redosledom.

Niti međusobno mogu da se sinhronizuju ukoliko pripadaju istom bloku ili *warp*-u. Najjednostavnija sinhronizacija je na barijeri – `_syncthreads()` u bloku, `_syncwarp()` u *warp*-u – gde sve niti čekaju dok ne dođu do iste tačke u kodu.

Niti mogu komunicirati preko globalne memorije, deljene memorije, atomičnih operacija, operacija na nivou *warp*-a, itd. Komunikaciju je potrebno sinhronizovati.

*Warp* je implicitno sinhronizovan u smislu obrade instrukcije, jer se sve niti u njemu izvršavaju u *lockstep*-u. Međutim, potencijalne memorijske operacije u okviru te instrukcije nisu implicitno sinhronizovane u *warp*-u.

## 3.4 Optimizacija kernela

U nastavku će biti date pojave koje treba imati u vidu prilikom optimizovanja kernela.

### 3.4.1 Global memory coalescing

*Memory coalescing* [8] je poželjna pojava, koja se dešava kada *warp* niti pristupa globalnoj memoriji prateći određen šablon. Tada se na hardverskom nivou broj memorijskih transakcija smanjuje na teoretski minimum grupisanjem, čime se smanjuje pritisak na globalnu memoriju i otključava veći paralelizam. Za dobre performanse, potrebno je da nulta *warp* nit pristupa lokaciji poravnatoj na 32 bajta, i da  $i$ -ta *warp* nit pristupa  $i$ -tom elementu, gde element treba biti veličine 1B, 2B, 4B, 8B ili 16B. Bitan faktor za *coalescing* je da elementi nemaju padding bajtove (prazan prostor) između sebe, u suprotnom se gube performanse.

### 3.4.2 Warp divergencija

*Warp* je obično transparentan programeru, osim kada treba maksimalno iskoristiti performanse GPU-a. Tada je potrebno minimizovati *warp* divergenciju [8], odnosno pojavu da se grananje u mašinskom kodu obilazi serijalizovano dva puta, jednom od strane niti koje bi ušle u jednu granu, a zatim drugi put od strane niti koje ne bi ušle u datu granu. To je neophodno uraditi zbog *lockstep*-a niti u *warp*-u.

Što su instrukcije uslovnog skoka ugnježdenije, to je *warp* divergencija veća, a iskorišćenost manja. U najgorem slučaju, svaka od 32 niti u *warp*-u ima sopstvenu putanju izvršavanja, i *warp* je potrebno izvršiti čak 32 puta serijalizovano, nad istim kodom. Na divergenciju utiče i vreme izvršavanja divergentnog puta u kodu. Delotvoran način optimizacije vremena izvršavanja je smanjenje *warp* divergenciju u „vrućim“ delovima koda.

### 3.4.3 Warp occupancy

*Warp occupancy* [8] oslikava procentualno maksimalan paralelizam na nivou SM-a, ukoliko se na SM-u izvršava samo posmatrani kernel, i ako je *grid* dovoljno veliki. Računa se kao odnos broja aktivnih warp-ova i maksimalnog broja warp-ova na jednom SM-u. Na metriku utiče:

1. broj warp-ova u bloku niti – što je veći blok to ih se može manje smestiti na SM,
2. veličina dodeljenih registara za blok niti – što je više registara dodeljeno, to je maksimalan broj warp-ova po SM-u manji,
3. veličina alocirane deljene memorije za blok niti – što je alokacija veća, to se manje blokova niti može smestiti na SM.

*Warp occupancy* je dobra heuristika za optimizaciju navedenih parametara. Međutim, ne mora uvek oslikavati stvarne performanse.

### 3.4.4 Register spill

Kada prevodilac prevodi kod kornela, ukoliko mu se ne navede drugačije, neće ograničiti broj registara po niti. To može limitirati paralelizam, jer su registri resurs koga po pravilu ima najmanje. Moguće je ograničiti broj registara po niti za konkretan kernel. Međutim, u slučaju da mu ponestane slobodnih registara, prevodilac je nateran da koristi globalnu memoriju kao privremene *spill-over* registre [8]. To smanjuje performanse ukoliko je već prisutan veliki pritisak na GPU memoriju magistralu, ili zbog obavezno većeg broja instrukcija u vrućem kodu. *Register spill* se može izbeći povećavanjem limita broja registara po niti ili izmenom koda.

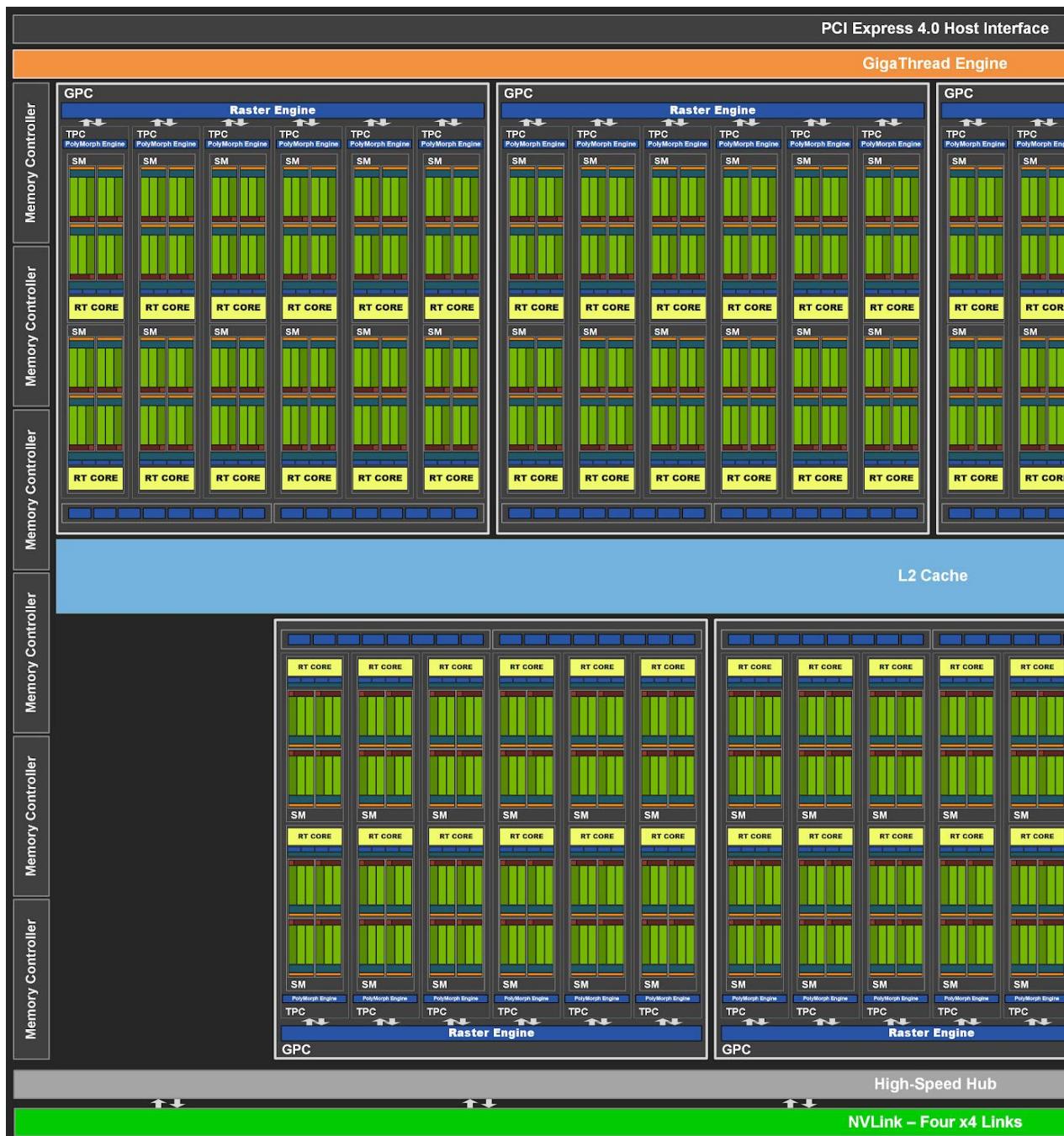
### 3.4.5 CUDA tokovi

Ukoliko se pokreće više kernela, često je moguće paralelizovati memoriske transfere i izvršavanja kernela, bez gubljenja ispravnosti algoritma. To se može postići CUDA tokovima [8] (engl. *stream*). U okviru jednog toka, svi taskovi – korneli, memoriski transferi, itd. – izvršavaju se sekvencialno, dok je izvršavanje jednog toka nezavisno od drugog. To je dobar način da se delimično ili potpuno sakrije vreme potrebno za memoriske transfere.

## 3.5 Pregled korišćenih tehnologija i alata

U ovom radu je korišćen sledeći hardver:

- *AMD Ryzen 9 7950X3D* 16-core procesor sa uključenim *hyperthreading*-om – 16 CPU jezgara odnosno 32 hardverske niti sa osnovnim taktom 4.2GHz, 1024KB L1 keš, 16MB L2 keš, 128MB L3 keš,
- *Asus ROG Crosshair X670E Hero* matična ploča sa *PCIe* 5.0, x16 128Gb/s bidirekcionim transfer ili 64Gb/s unidirekcionim,
- *HyperX Trident Z5 DDR5* operativna memorija veličine 64GB na 3600MT/s,
- *Gigabyte RTX3090* grafička kartica sa 24GB *GDDR6X* globalne memorije i 32GB deljene memorije, 82 *streaming* multiprocesora sa 128 CUDA jezgra i 64KB registara po multiprocesoru, arhitektura Ampere (CUDA 8.6).



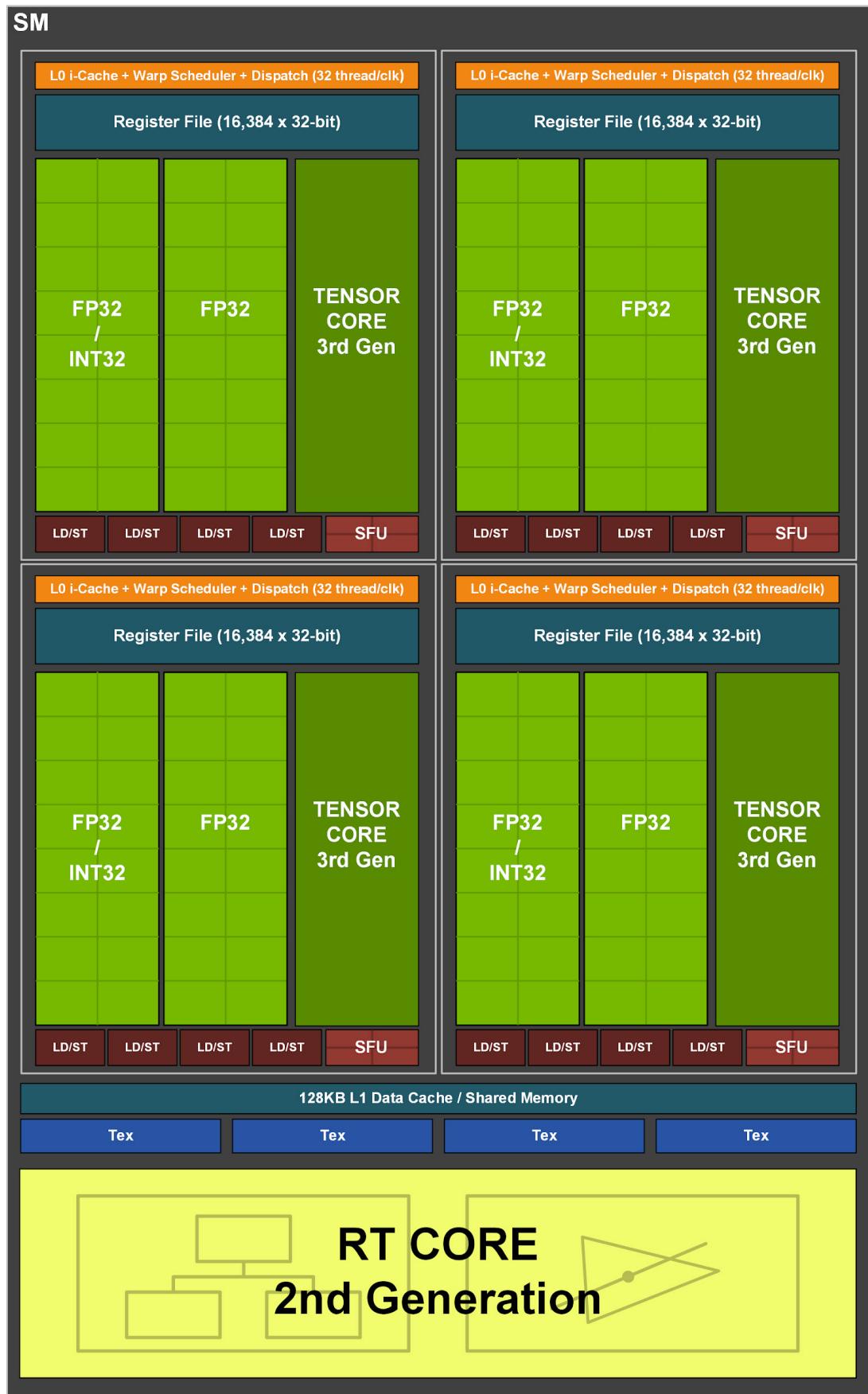
Slika 5 – leva polovina dijagrama GA102 arhitekture (RTX3090) [9]. GPU sadrži sedam GPC jedinica, svaka od kojih sadrži šest TPC jedinica, svaka od kojih sadrži dva SM-a. Na dijagramu nisu prikazana 168 FP64 jezgra (dva po SM-u).

Implementacije algoritama su rađene u CUDA proširenom C++-u. Za CPU višenitnu implementaciju i za merenje vremena faza u svakoj implementaciji se koristi *OpenMP* [10].

Za prevođenje i druge taskove projekta su korišćeni *Visual Studio Community 2022* i *Cmake*.

Izvršna okruženja su *Microsoft Windows* i *Ubuntu WSL*.

Za merenje performansi korišćeni su *Python*, *Nvidia Nsight Compute* i *NVML Python* biblioteka na *Windows*-u i *Hyperfine* na *Ubuntu WSL*.



Slika 6 – SM u arhitekturi GA102 [9]

## 4 Implementacija algoritama sa poboljšanim performansama

Jedan od ciljeva rada je isprobavanje raznih optimizacija suksesivno, kako bi se eksperimentalnim putem utvrdilo koji algoritam ima najbolje željene karakteristike. Kao polazna tačka je uzeta najjednostavnija CPU implementacija, koja je zatim korišćena za verifikaciju ispravnosti narednih algoritama.

### 4.1 Prepostavke algoritama

Fokus implementacije je na sledećem:

1. Potrebno je pronaći optimalno poravnanje, gde je brzina i racionalna upotreba resursa ključna.
2. Vraćanje cele skor matrice ili njenog dela iz GPU globalne memorije u RAM je neophodno, tj. potrebno je pronaći optimalnu sekvencu izmena.
3. Primarni cilj je poravnanje dve jako dugačke sekvene. Poravnanje velikog broja malih sekvenci na jako dugačku sekvencu nije fokus.

Algoritmi su koncipirani tako da imaju najbolje performanse ukoliko je sekvenca  $\vec{x}$  dovoljno dugačka, ili barem duža ili slične dužine sekvenci  $\vec{y}$ . Radi jednostavnosti, u nastavku rada smatraćemo da je skor matrica  $\mathcal{A}$  šira nego što je viša, što ne mora biti slučaj u realnoj upotrebni.

Radi jednostavne provere, algoritmi koriste Needleman-Wunsch način računanja trenutnog polja, kao i linearu skor funkciju.

Za čuvanje vrednosti polja u skor matrici je odabran 32-bitni *signed integer*. Generalno govoreći, grafički procesor je optimizovan da radi sa *shader-ima*<sup>1</sup>, stoga ima najbolje performanse prilikom poređenja *float-ova* (tabela 1). Međutim, radi jednostavnosti verifikacije rezultata, odabran je *integer* za smeštanje podataka.

Tabela 1 – maksimalan broj tera operacija u sekundi na RTX3090 [11]

FP32 (non-Tensor) <sup>2</sup>	20.3 TOPS
INT32 (non-Tensor)	10.2 TOPS

Ukoliko poravnavamo dve sekvene iste dužine, i koristimo linearnu skor šemu sa BLOSUM62 i konzervativnom vrednošću 11 za  $g_o$ , tada je maksimalna dužina sekvene koja čak i u najgorem slučaju ne izaziva prekoračenje (*overflow/underflow*) prilikom računanja skor matrice jednaka

$$\frac{|\min\_int|}{2 \cdot g_o} = \frac{2^{31}}{2 \cdot 11} \approx 97.5M \quad (4)$$

<sup>1</sup> Specijalizovan programski kod koji se koristi za obradu grafičkih podataka.

<sup>2</sup> Odnosi se na FP32 ili INT32 operacione jedinice koje ne barataju sa tenzorima.

U radu će biti poravnate sekvene kraće od limita diktiranog 32-bitnim smeštanjem podataka.

## 4.2 Objasnjenje skracenica u nazivima algoritama

Navedeni pojmovi će biti korišćeni u nazivima algoritama za poravnanje i rekonstrukciju poravnanja, gde ima smisla, da opišu karakteristike samog algoritma.

U nastavku rada će se termin „dijagonalala“ koristiti sa značenjem „sporedna dijagonalala“. Označava grupaciju elemenata matrice/pločice, npr. prva dijagonalala sa leva na slici (7).

Pojmovi vezani za način paralelizacije:

1. *st – single-threaded*, obrada sa jednom CPU niti.
2. *mt – multi-threaded*, višenitna CPU implementacija koja koristi *OpenMP*.
3. *ml – multi-launch*, isti kernel se više puta pokrene u istom CUDA toku, ali sa različitim ulaznim parametrima.
4. *coop – CUDA cooperative launch*, ovi algoritmi koriste *whole grid* sinhronizaciju i pokreću kernel jednom, umesto više puta kao *ml*.
5. *mlsp – multi-launch sparse*, tj. retka reprezentacija matrice ovde znači da se čuva samo deo skor matrice u GPU globalnoj i RAM memoriji. Zahteva da se koristi modifikovani algoritam za rekonstrukciju poravnanja.
6. *mlsppt – multi-launch sparse with parallel transfer*, gde se memorijski transfer vrši paralelno sa računicom. Koristi bar dva CUDA toka kako bi se to postiglo.

Pojmovi vezani za način obilaska:

1. *row – row-major* redosled računice elemenata matrice/pločica.
2. *diag – minor-diagonal* redosled računice elemenata matrice/pločica.
3. *2pass –* koriste se dva kernela, gde prvi parcijalno inicijalizuje skor matricu radeći samo računice koje ne zavise od suseda, zatim drugi kernel uradi ostatak računice.
4. *skew –* pločica/podpločica je oblika paralelograma sa uglom od  $45^\circ$ . Elementi se mogu grupisati u sporedne dijagonale *istih* dužina bez preteka, za razliku od pravougaonika.

Algoritmi u radu dati su u tabelama (2) i (3).

Tabela 2 – pregled algoritama za poravnavanje u radu

Naziv algoritma	Opis
Cpu1-St-Row	Izračunavanje skor matrice po vrstama.
Cpu2-St-Diag	Izračunavanje skor matrice po sporednim dijagonalama.
Cpu3-St-DiagRow	Matrica se deli na pravougaone pločice. Pločice se obilaze po sporednim dijagonalama, a unutar svake pločice elementi se obilaze po vrstama.

Cpu4-Mt-DiagRow	Višenitna verzija Cpu3-St-DiagRow. Jedna nit po pravougaonoj pločici. Statički raspored dodele pločica na dijagonalni odgovarajućim nitima.
Gpu1-Ml-Diag	Pokretanje kernela za svaku sporednu dijagonalu. Jedna nit po elementu.
Gpu2-Ml-DiagRow2Pass	Kao Gpu1-Ml-Diag, ali sa jednom niti po pločici. Dvoprolazna obrada – prvi prolaz radi obradu koja ne zavisi od ćelija suseda.
Gpu3-Ml-DiagDiag	Kernel se pokreće za svaku sporednu dijagonalu pločica. Više niti po pločici – po jedna nit po vrsti pločice. Niti se sinhronizuju na svakoj sporednoj dijagonalni unutar pločice.
Gpu4-Ml-DiagDiag2Pass	Kao Gpu3-Ml-DiagDiag, ali sa dvoprolaznom obradom kao u Gpu2-Ml-DiagRow2Pass.
Gpu5-Coop-DiagDiag	Kao Gpu3-Ml-DiagDiag, ali koristi <i>whole grid</i> sinhronizaciju umesto višestrukog pokretanja kernela.
Gpu6-Coop-DiagDiag2Pass	Kao Gpu4-Ml-DiagDiag2Pass, ali koristi <i>whole grid</i> sinhronizaciju umesto višestrukog pokretanja kernela.
Gpu7-Mlsp-DiagDiag	Kao Gpu3-Ml-DiagDiag, ali predstavlja skor matricu kao heder vrste i kolone pločica. Samo ti elementi se prebacuju nazad u RAM.
Gpu8-Mlsp-DiagDiag	Kao Gpu7-Mlsp-DiagDiag, ali čuva pločicu skoro kompletno u registrima umesto u deljenoj memoriji.
Gpu9-Mlsp-DiagDiagDiag	Kao Gpu8-Mlsp-DiagDiag, ali deli pravougaonu pločicu na podpločice oblika paralelograma sa uglom od $45^\circ$ . Jedna nit po vrsti podpločice. Podpločica se obilazi po sporednim dijagonalama.

Tabela 3 – pregled algoritama za rekonstrukciju poravnjanja u radu

Naziv algoritma	Opis
Trace1-Plain	Standardan algoritam za rekonstrukciju poravnjanja.
Trace2-Sparse	Rekonstrukcija poravnjanja u retkoj skor matrici, gde se čuvaju samo header vrsta i kolona pločica skor matrice. Neophodan za Gpu7-Mlsp-DiagDiag, Gpu8-Mlsp-DiagDiag i Gpu9-Mlsp-DiagDiagDiag.

## 4.3 Cpu1-St-Row i Trace1-Plain

Dati algoritam poravnjanja Cpu1-St-Row je zbog svoje jednostavnosti odabran kao polazna tačka za implementaciju i poređenje sa ostalim algoritmima. Jedna CPU nit inicijalizuje nultu vrstu i kolonu skor matrice. Zatim nit računa elemente u row-major obilasku. Kada nit stigne do donjeg

desnog elementa skor matrice, tada je pronađena cena optimalnog poravnjanja polaznih sekvenci  $\vec{x}$  i  $\vec{y}$ .

Pošto smatramo da je potrebno naći optimalnu sekvencu izmena, algoritam za rekonstrukciju poravnjanja Trace1-Plain polazi od donjeg desnog ugla matrice, i prati putanju najjeftinijih operacija (tabela 4). Na kraju se obrne dobijena sekvenca izmena, kako bi prirodno polazila iz gornjeg levog ugla ka donjem desnom.

Vreme izvršavanja ovog algoritma dato na grafiku (1) oslikava broj zaista neophodnih operacija u jedinici vremena, tj. protočnost. Zbog toga je pogodno za poređenje sa daljim algoritmima.

Mana datog algoritma poravnjanja je što ne koristi nikakvu paralelizaciju. To je limitacija samog row-major redosleda obilaska elemenata.

## 4.4 Cpu2-St-Diag algoritam

Posmatrajmo zavisnosti elemenata u skor matrici – zavisnost jednog elementa po podacima je od njegovog levog, gornjeg i gornjeg-levog elementa (slika 7). Kako bismo povećali paralelizam, potrebno je omogućiti što više nezavisnih izračunavanja. Intuitivan način da to ostvarimo je da grupišemo elemente matrice u sporedne dijagonale. Primetićemo da su svi elementi na sporednim dijagonalama međusobno nezavisni, i da zavise isključivo od odgovarajućih elemenata sa prethodne dve sporedne dijagonale (slika 7).

U ovom algoritmu jedna nit obilazi skor matricu po sporednim dijagonalama odozgo nadole, gde se u jednom trenutku obrađuje maksimalno jedna sporedna dijagonala.

Pozitivna strana algoritma je mogućnost paralelizacije izračunavanja skor matrice.

Važna ali manje očigledna mana algoritma je loša upotreba keš memorija. Pošto se matrica čuva linearizovano po vrstama, za dovoljno veliku matricu, jedna cela L1 keš linija elemenata mora biti prisutna samo da bi se pristupilo jednom elementu sa dijagonale. Samim tim je lako je izazvati kolizije keš linija. To nije slučaj u Cpu1-St-Row algoritmu (4.3).

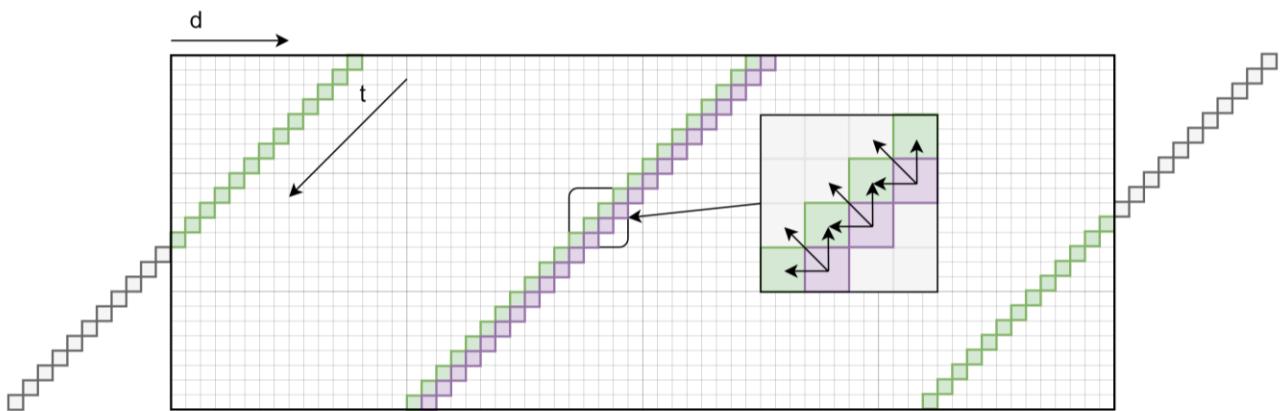
Vreme izvršavanja algoritma dato na grafiku (1) dosta zavisi od veličine L1 keša, i njegove iskorišćenosti tokom izračunavanja skor matrice. Zbog toga, algoritam neće biti paralelizovan, jer bi to učinilo vreme izvršavanja još nepredvidivijim.

## 4.5 Cpu3-St-DiagRow algoritam

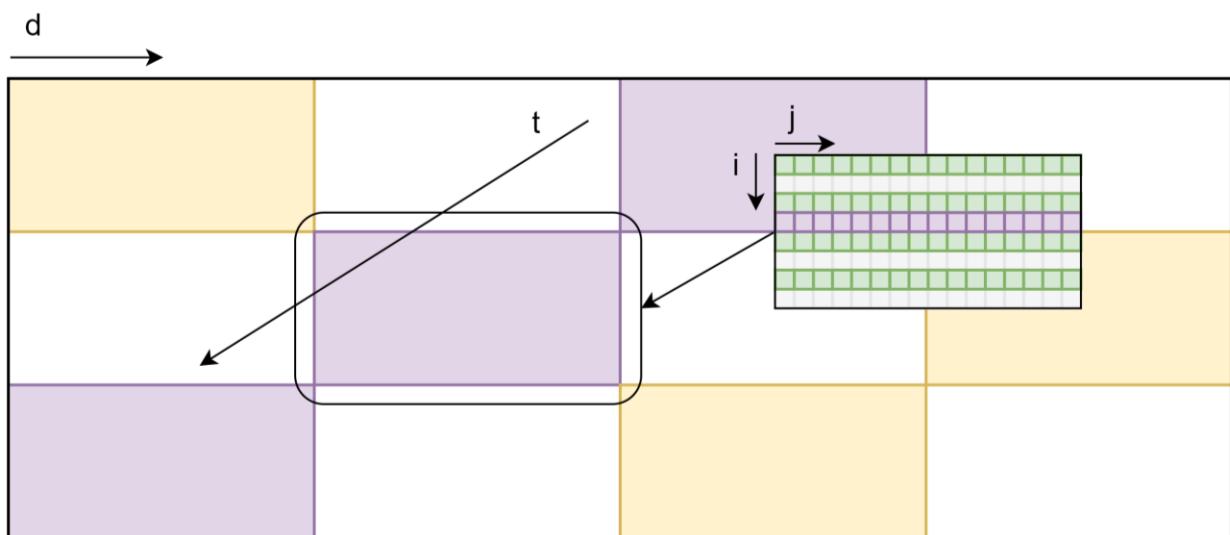
Koristeći ideje iz Cpu1-St-Diag (4.3) i Cpu2-St-Diag (4.4) algoritama, možemo smisliti sledeći obilazak (slika 8):

Tabela 4 – operacija izmene i odgovarajuće pomeranje tokom rekonstrukcije poravnjanja

praznina u $\vec{x}$	UP
praznina u $\vec{y}$	LEFT
poklapanje/nepoklapanje	UP-LEFT



Slika 7 – obilazak elemenata po (sporednoj) dijagonalni



Slika 8 – obilazak pločica po dijagonalni i elemenata po vrsti

- Podelimo matricu na pravougaonike – pločice, i obilazimo matricu pločica po dijagonalni.
- Unutar same pločice, elemente obilazimo po vrsti.

Ovakav pristup omogućava paralelizaciju, a nema loše osobine Cpu2-St-Diag algoritma. Vreme izvršavanja dato na grafiku (1) je u praksi identično Cpu1-St-Diag algoritmu.

## 4.6 Cpu4-Mt-DiagRow algoritam

Ovo je modifikovan Cpu3-St-DiagRow algoritam (4.5), koji koristi više niti za paralelno izračunavanje skor matrice. Koristi *OpenMP* za paralelizaciju, i statičku podelu posla. Za jedinicu posla je uzeta jedna pločica, i tačno jedna nit je računa. Za datu dijagonalu pločica, CPU nit  $i$  uzima pločicu koji daje ostatak  $i$  pri deljenju sa #threads, izračunava je obilaskom po vrstama, i prelazi na sledeću takvu pločicu. Kada nit izračuna sve svoje pločice na dijagonalu, sačeka sve ostale niti da to isto urade, i pređe na sledeću dijagonalu.

Vreme izvršavanja dato na grafiku (1) ovog algoritma je #cpu\_cores puta brže od Cpu1-St-Row. To je očekivano, zbog toga što *OpenMP* u praksi koristi onoliko niti koliko ima hardverskih niti (uzimajući u obzir *hyperthreading* ukoliko je uključen).

Ovim su istražene osnovne ideje paralelizacije Needleman-Wunsch algoritma.

## 4.7 Gpu1-Ml-Diag algoritam

Najjednostavnija GPU implementacija Needleman-Wunsch algoritma, inspirisana Cpu2-St-Diag algoritmom (4.4).

Pre izračunavanja na GPU-u potrebno je prebaciti vektore  $\vec{x}$ ,  $\vec{y}$  i supstitionu matricu  $\mathcal{S}$  u globalnu memoriju, kao i alocirati prostor u globalnoj memoriji za skor matricu, koja će nakon izračunavanja biti poslata u operativnu memoriju.

Skor matrica se obilazi po sporednim dijagonalama elemenata. Algoritam pokreće po jedan CUDA kernel<sup>1</sup> za svaku dijagonalu u podrazumevanom CUDA toku 0, čime je osigurana sinhronizacija svih blokova niti u kernelu. Broj niti u grid-u je broj elemenata na dijagonalni skor matrice, zaokrugljen naviše tako da bude deljiv na blokove bez ostatka.

Broj dijagonala je  $|\vec{x}| + |\vec{y}| - 1$ . Nit  $i$  na dijagonali  $d$  računa tačno jedan element skor matrice, na poziciji  $(i, d - i)$ , i završava rad.

Pozitivne strane algoritma su jednostavnost i omogućena paralelizacija računice.

Velika mana algoritma je *uncoalesced*, efektivno nasumičan, pristup memoriji. Slično algoritmu Cpu2-St-Diag na kome je baziran, problem je izazvan time što je obilazak po dijagonalama i što je jedinica posla jedan element. Međutim, pošto je obilazak po dijagonalama elemenata, a matrica se čuva po vrstama, jasno je da to nije ispunjeno.

Dodatna mana je što preovladava korišćenje globalne memorije u toku računice skor matrice. GPU poseduje mnogo brže memorije, bliže nitima prema broju ciklusa za pristup, koje bi mogле biti korišćene umesto globalne za veliki ideo računice.

Ove karakteristike značajno smanjuju protočnost algoritma. Vreme izvršavanja je dato na grafiku (1).

Korišćeni algoritam za rekonstrukciju poravnjanja je Trace1-Plain (4.3).

## 4.8 Gpu2-Ml-DiagRow2Pass algoritam

Algoritam je zasnovan na Gpu1-Ml-Diag algoritmu (4.7), i po obilasku je sličan Cpu4-Mt-DiagRow algoritmu (4.6).

Elementi skor matrice se grupišu u pravougaone pločice male veličine (npr.  $4 \times 8$ ). Matrica pločica se obilazi po dijagonalama, a elementi unutar pločice se obilaze po vrstama. Jedna nit obilazi tačno jednu pločicu nakon čega završava sa radom.

Deljena memorija omogućava značajno brži slučajan pristup od globalne memorije, i značajno je bliža nitima prema broju ciklusa potrebnih za pristup. Zbog toga što se nit, a samim tim i blok niti, duže izvršava nego u Gpu1-Ml-Diag algoritmu, ima smisla učitati cela supstitionu matricu  $\mathcal{S}$  u deljenu memoriju, čime se povećava protočnost algoritma.

---

<sup>1</sup> Kerneli se, zbog svog velikog broja, pokreću u okviru CUDA grafa, kako bi bilo izvodljivo profilisati kod sa Nvidia Nsight Compute. Ova optimizacija se primenjuje u svim *multi-launch* algoritmima.

Radi pojednostavljenja kernela koji računa skor matricu, prethodno se pokreće odvojeni kernel koji inicijalizuje nultu vrstu i nultu kolonu. To je moguće jer su u njima svi elementi međusobno nezavisni i unapred poznati.

Algoritam ima iste mane kao i Gpu1-Ml-Diag. Vreme izvršavanja je dano na grafiku (1).

## 4.9 Gpu3-Ml-DiagDiag algoritam

Algoritam je optimizacija Gpu2-Ml-DiagRow2Pass (4.8) algoritma, gde je ideja da se što više koristi deljena memorija umesto globalne.

Elementi skor matrice se grupišu u pravougaone pločice, gde je veličina pločice

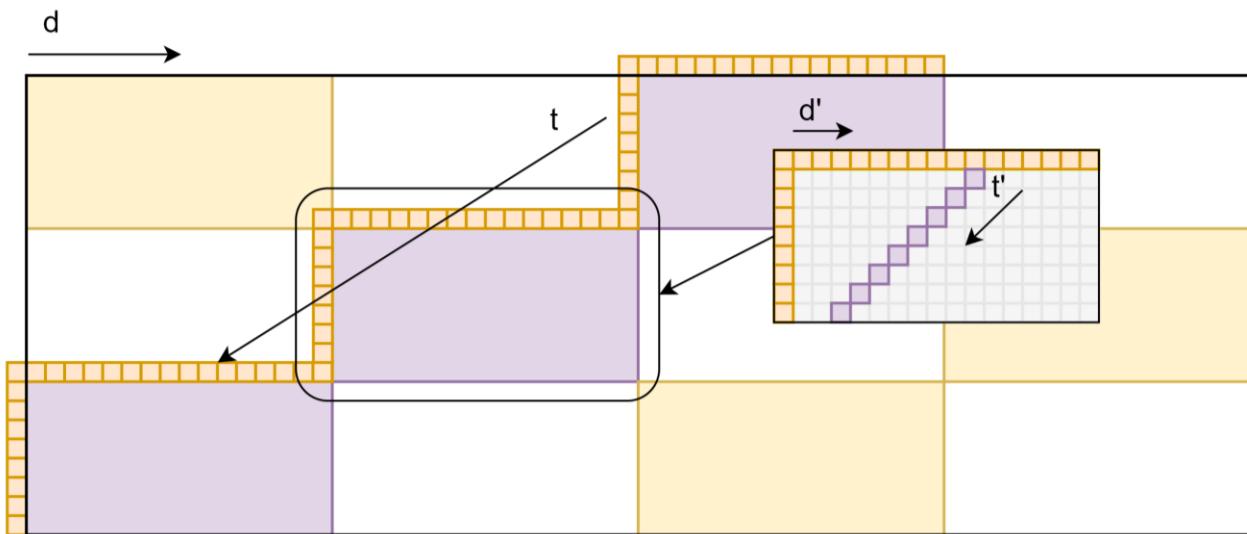
$$(height, width) = (32, t), \quad t \geq 32 \quad (5)$$

Radi pojednostavljenja kernela kao u algoritmu Gpu2-Ml-DiagRow2Pass, prvo se pokreće odvojeni kernel koji inicijalizuje nultu vrstu i nultu kolonu skor matrice u globalnoj memoriji.

Kernel se pokreće više puta, po jednom za svaku dijagonalu pločica. Blok niti sa indeksom  $i$  obrađuje jednu pločicu  $T$  sa indeksom  $i$  na dotoj dijagonali, nakon čega završava. Blok niti ima više warp-ova, kako bi se ubrzao. Pristupi globalnoj i deljenoj memoriji od strane bloka/warp-a niti su poravnati i idu redom, svuda gde je to moguće.

Obrada pločice je podeljena po fazama:

1. U deljenu memoriju se učitava supstitucionu matrica  $\mathcal{S}$  i delovi vektora  $\vec{x}, \vec{y}$  koji odgovaraju pločici  $T$ . Dodatno, alocira se prostor takođe u deljenoj memoriji gde će se čuvati rezultat obrade pločice  $T$ , sa dimenzijama  $(1 + height, 1 + width)$ .
2. Zatim se inicijalizuju nulta vrsta i kolona pločice u deljenoj memoriji, koristeći najnižu vrstu neposredno-gornje pločice i najdesniju kolonu neposredno-leve pločice (slika 9), koje su sačuvane u skor matrici u globalnoj memoriji.
3. U prvom prolazu se inicijalizuju elementi pločice  $T$  u deljenoj memoriji, slično kao u algoritmu Gpu2-Ml-DiagRow2Pass. Cilj je smanjenje maksimalnog broja korišćenih registara kernela.
4. U drugom prolazu se pločica obilazi po dijagonalama od strane jednog warp-a (32 niti), gde nit sa indeksom  $i$  računa element  $i$  na dijagonali. Koristi se `__syncwarp()` barijera za sinhronizaciju. Zbog ispravnosti sinhronizacije na dijagonali u ovoj implementaciji, potrebno je fiksirati jednu dimenziju pločice na 32.
5. Konačno, pločica  $T$  se prepisuje iz deljene u globalnu memoriju.



Slika 9 – obilazak pločica po dijagonalima i elemenata same pločice po dijagonalama; čuvanje nulte vrste i kolone elemenata i trenutne pločice u deljenoj memoriji

Algoritam je drastično brži od prethodnih CPU i GPU algoritama, ukoliko se izuzme prenos izračunate skor matrice iz globalne u operativnu memoriju. Što su sekvence duže, to više dominira to vreme prenosa. Vreme izvršavanja je dano na grafiku (1).

Mana algoritma je velika osetljivost na raspoloživost resursa. Količina deljene memorije po bloku, broj registara po niti u bloku, broj niti u bloku, veličina elementa u skor matrici – u ovom algoritmu jako utiču na *occupancy*, odnosno broj blokova koji će moći istovremeno da se izvršavaju na jednom *streaming* multiprocesoru. Lako je smanjiti maksimalnu paralelnost algoritma, ukoliko se pokreću drugi kerneli.

Kao i u slučaju prethodnih GPU algoritama, što su poravnate sekvence duže, to je prostorni pritisak na globalnu memoriju veći, i dominantniji režijski trošak prenosa skor matrice u operativnu memoriju. To je mana čuvanja cele skor matrice u globalnoj memoriji. Lako je drastično smanjiti paralelnost drugih nepovezanih kernela koji bi hteli da se izvršavaju istovremeno sa GPU poravnanjem sekvenci.

## 4.10 Gpu4-Ml-DiagDiag2Pass algoritam

Algoritam je sličan Gpu3-Ml-DiagDiag algoritmu (4.9). Razlika je u tome što se 3. korak obrade pločice izvlači u sopstveni kernel, koji se izvršava pre glavnog kernela, slično kao u Gpu2-Ml-DiagRow2Pass (4.8). Performanse su malo bolje od Gpu3-Ml-DiagDiag algoritma, jer je obrada učinjena regularnijom. Vreme izvršavanja je dano na grafiku (1).

Mane algoritma su iste kao u Gpu3-Ml-DiagDiag.

## 4.11 Gpu5-Coop-DiagDiag algoritam

Algoritam je baziran na Gpu3-Ml-DiagDiag algoritmu (4.9). Ideja je da se koristi kooperativno pokretanje kernela, kako bi se omogućila sinhronizacija blokova direktno na GPU-u. Sam način obilaska, prenosi i slično su isti kao u Gpu3-Ml-DiagDiag algoritmu.

Prednost pristupa je da se izbegava režijski trošak višestrukog pokretanja kernela, što smanjuje vreme izvršavanja.

Mane algoritma su iste kao u Gpu3-Ml-DiagDiag algoritmu.

Dodatno, kod kooperativnog pokretanja kernela, svi blokovi u *grid*-u se moraju pokrenuti istovremeno. Blokovi se tipično izvršavaju dugo. Pošto nije moguće promeniti kontekst tako da se trenutni blok zameni za neki drugi, resursi su limitirani za druge potencijalne kernele, čime se smanjuje paralelizam na nivou različitih kernela.

Vreme izvršavanja algoritma je dano na grafiku (1). Kraće je od vremena izvršavanja Gpu3-Ml-DiagDiag algoritma.

## 4.12 Gpu6-Coop-DiagDiag2Pass algoritam

Algoritam je baziran na Gpu4-Ml-DiagDiag2Pass algoritmu (4.10). Slično kao u Gpu5-Coop-DiagDiag, koristi se kooperativno pokretanje kernela.

Mane algoritma su iste kao u Gpu3-Ml-DiagDiag algoritmu (4.9), kao i mana kod kooperativnog pokretanja kernela navedena u Gpu5-Coop-DiagDiag algoritmu (4.11).

Vreme izvršavanja algoritma je dano na grafiku (1). Kraće je od vremena izvršavanja Gpu4-Ml-DiagDiag2Pass algoritma. Najbolje je od vremena izvršavanja svih algoritama do sada.

## 4.13 Gpu7-Mlsp-DiagDiag i Trace2-Sparse algoritam

Limitacija dosadašnjih algoritama je velika upotreba memorije. Što je veća površina matrice, to vreme transfera matrice u operativnu memoriju sve više zasenjuje samo vreme izračunavanja matrice, dano na grafiku (5). Dužine sekvenci  $\vec{x}$  i  $\vec{y}$  su dosta limitirane raspoloživom globalnom i operativnom memorijom, jer se skor matrica mora cela smestiti u globalnu memoriju. Prostorna složenost algoritma je  $O(mn)$ .

Poznat je CPU algoritam [12] koji ima linearnu prostornu složenost za izračunavanje skor matrice i za rekonstrukciju poravnjanja. U ovom radu nije razmatrano kako bi se ta tehnika inkorporirala u paralelnu obradu skor matrice u implementiranim GPU algoritmima.

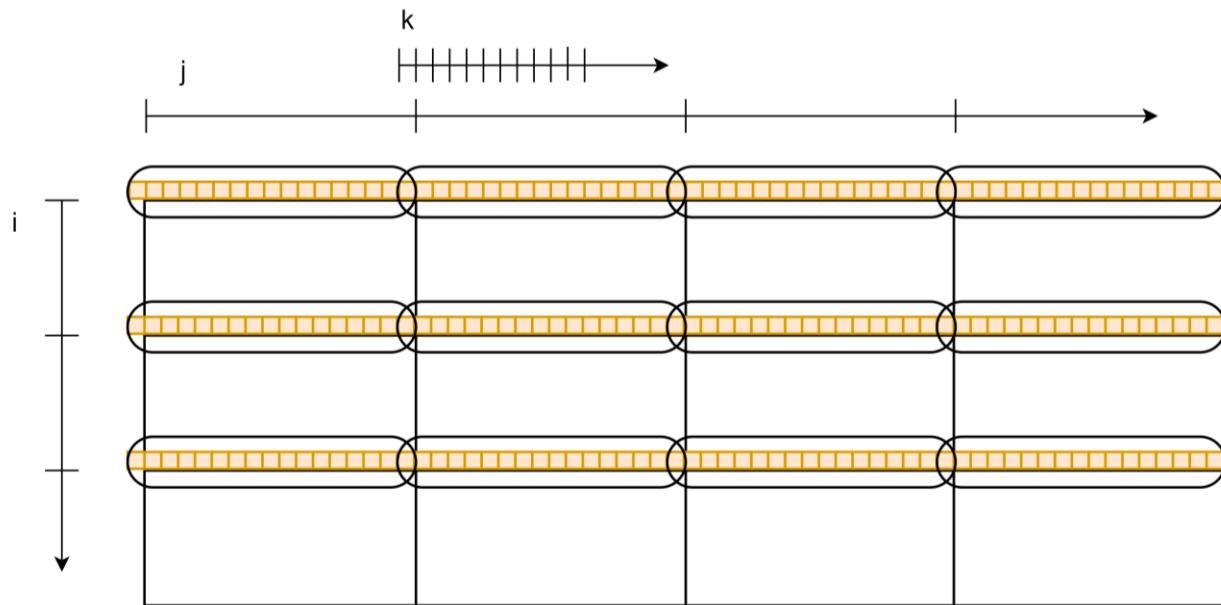
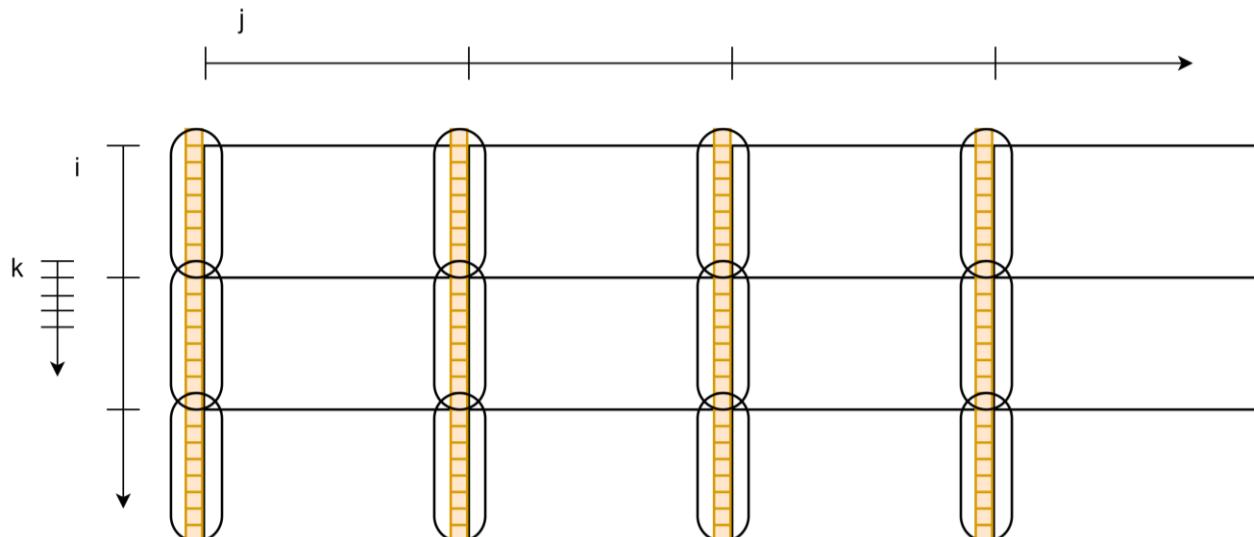
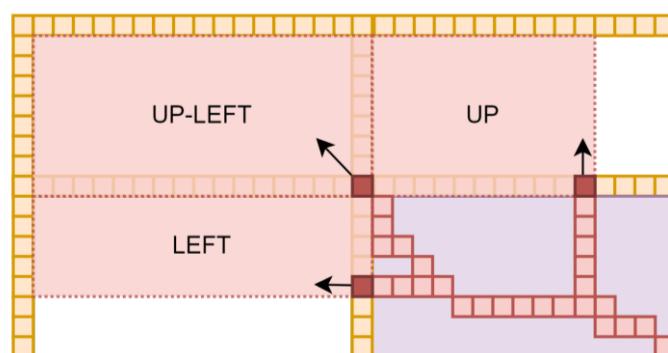
Ideja ovog algoritma je da se smanji konstantan faktor prostorne složenosti, koja je i dalje  $O(mn)$ . Time se povećavaju dužine sekvenci koje se mogu poravnati.

Algoritam je baziran na Gpu3-Ml-DiagDiag algoritmu (4.9). Jedina razlika je način pristupa skor matrici, koja se sada čuva kao dve retke matrice:

- *tileHrowMat* – matrica heder vrsta pločica (slika 10),
- *tileHcolMat* – matrica heder kolona pločica (slika 11).

Faktor redukcije prostora skor matrice u globalnoj/operativnoj memoriji je:

$$k_{reduce} = \frac{m \cdot n}{m + n} \geq \frac{\min(m, n)}{2} \quad (6)$$

Slika 10 – retka reprezentacija skor matrice – *tileHrowMat*Slika 11 – retka reprezentacija skor matrice – *tileHcolMat*

Slika 12 – tri moguća pravca prelaska u narednu pločicu tokom rekonstrukcije poravnjanja trenutne. Pri prelasku, potrebno je učitati nultu vrstu i kolonu, i izračunati odgovarajuću osećenu zonu.

Time je omogućeno poravnati sekvence koje prave  $k_{reduce}$  puta veću površinu skor matrice od prethodnih algoritama. Ukoliko su sekvence iste dužine, maksimalna dužina sekvence je  $\sqrt{k_{reduce}}$  puta veća nego pre.

Obrada pločice je slična kao u Gpu3-Ml-DiagDiag algoritmu. Razlika je da se nulta vrsta i kolona pločice  $T$  inicijalizuju direktno iz  $tileHrowMat$  i  $tileHcolMat$ . Kada se izračuna cela pločica  $T$  u deljenoj memoriji, prenose se samo poslednja vrsta i kolona u  $tileHrowMat$  sa koordinatama donje pločice i  $tileHcolMat$  sa koordinatama desne pločice od trenutne.

Za rekonstrukciju poravnjanja se koristi modifikovani algoritam (slika 12), pošto nisu sačuvani svi elementi pločice. Poznate su  $(i, j)$  koordinate elementa od kog se kreće rekonstrukcija. Algoritam ima sledeće korake:

1. Računaju se samo neophodni elementi pločice od koordinate  $(i, j)$ . Za računanje pločice je odabran Cpu4-Mt-DiagRow algoritam (4.6), radi smanjenja režijskih troškova.
2. Radi se standardna rekonstrukcija poravnjanja u okviru pločice. Kada se u okviru pločice nađe na nultu vrstu/nultu kolonu/obe istovremeno, prelazi se na gornju/levu/gornju-levu pločicu.
3. Koraci 1-2 se ponavljaju dok se ne dode do koordinate  $(0, 0)$ .

Algoritam ispoljava slične prednosti i mane kao Gpu6-Coop-DiagDiag2Pass, ukoliko se koriste sekvence maksimalne podržane dužine. To je fundamentalna limitacija čuvanja cele pločice u deljenoj memoriji bloka niti na CUDA arhitekturama današnjice.

Algoritam ima dosta kraće vreme izvršavanja u odnosu na Gpu6-Coop-DiagDiag2Pass (4.12) na istim sekvencama, zbog toga što je smanjeno vreme transfera retke skor matrice u operativnu memoriju. Vreme računanja elemenata skor matrice je nešto duže. Vreme izvršavanja algoritma je dato na grafiku (1).

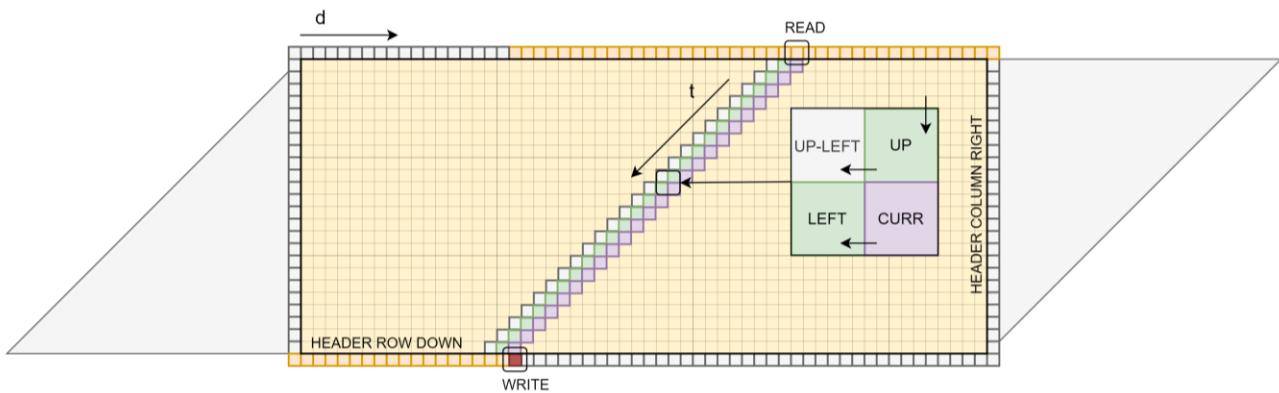
## 4.14 Gpu8-Mlsp-DiagDiag algoritam

Baziran je na Gpu7-Mlsp-DiagDiag algoritmu (4.13). Ideja je da se smanji upotreba deljene memorije i poveća occupancy.

U deljenoj memoriji se, umesto cele pločice  $T$ , rezerviše prostor za heder vrstu  $tileHrow$  i heder kolonu  $tileHcol$  pločice  $T$ , koji se inicijalizuju iz  $tileHrowMat$  i  $tileHcolMat$ .

Blok niti ima više warp-ova koji se koriste za memoriske transfere. Tačno jedan warp računa elemente pločice, slično kao u Gpu7-Mlsp-DiagDiag, odnosno Gpu3-Ml-DiagDiag algoritmu (4.9).

U tom warp-u, svaka nit čuva tri odgovarajuća elementa u svojim registrima (slika 13) – njen trenutni UP, LEFT i UP-LEFT element. Nit sa indeksom  $i$  računa samo elemente iz vrste  $i$ . Na trenutnoj dijagonali, nit računa CURR element pomoću UP, LEFT i UP-LEFT elemenata.



Slika 13 – čuvanje četiri elemenata u registrima jedne niti. Čitanje radi nit 0, upis radi nit 31. UP-LEFT i LEFT postaju UP i CURR, a UP dobija novu vrednost od niti sa manjim indeksom.

Komunikacija sa *tileHrow* i *tileHcol* u deljenoj memoriji se vrši na sledeći način:

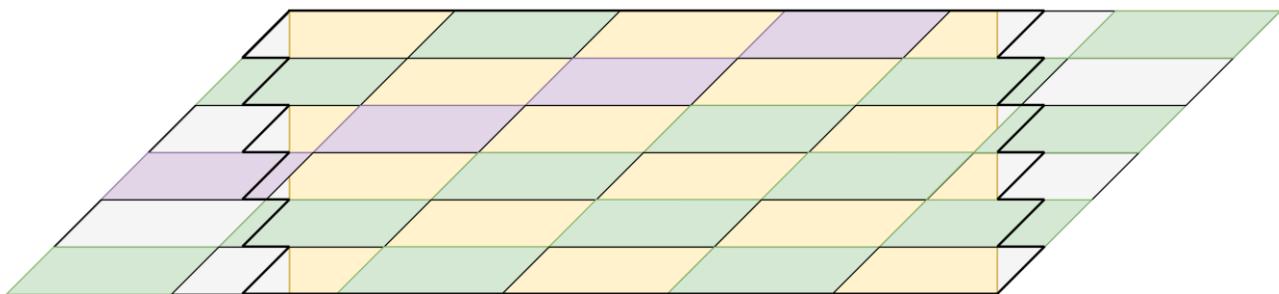
1. Pre početka računice, sve niti u warp-u prepisuju svoj LEFT element iz *tileHcol* u svoj lokalni registar.
2. Nit 0 u warp-u, svake iteracije čita svoj UP element iz *tileHrow*.
3. Nit 31 u warp-u, koja se mapira na poslednju vrstu u pločici, svake iteracije upisuje svoj CURR element u *tileHrow*. Nije narušena ispravnost rada warp niti 0 – iz elemenata *tileHrow* i *tileHcol* koji su pregaženi novom vrednosti, warp nit 0 ne čita u nastavku računice, jer ih je warp dijagonala „prošla“. Time se štedi deljena memorija u bloku.
4. Svaka warp nit, kada dođe do poslednje kolone u pločici, upisuje svoj CURR element u *tileHcol*.

Nakon što se izračunaju svi elementi pločice, *tileHrow* i *tileHcol* se prepisuju u globalnu memoriju, kao heder vrsta donje pločice u *tileHrowMat* i heder kolona desne pločice u *tileHcolMat*.

Za sinhronizaciju niti na dijagonali elemenata u pločici se koristi `_shfl_up_sync()`. Dodatno, služi da nit  $i$  prosledi vrednost svog CURR elementa niti  $i + 1$ , koja će ga koristiti kao svoj UP element u narednoj iteraciji (slika 13). UP-LEFT i LEFT elemente nije potrebno proslediti između niti – u narednoj iteraciji, oni će biti jednaki UP i CURR elementima iz trenutne iteracije.

Pozitivna strana algoritma je što se drastično manje koristi deljena memorija po bloku. Efektivno se čuvaju samo dve prethodne dijagonale pločice, i to u registrima bloka niti. Kernel nije više toliko osetljiv na raspoloživost resursa, kao u algoritmima GPU3-GPU7 (4.9, 4.10, 4.11, 4.12, 4.13).

Manja algoritma je da je maksimalna dužina sekvenci i dalje dosta ograničena – reda veličine 200Kbase. Faktor redukcije prostora skor matrice je ograničen zbog toga što se koristi tačno jedan warp da izračuna elemente pločice, i što se svaka nit mapira na tačno jednu vrstu pločice.



Slika 14 – podjela pločice (pravougaonik) na podpločice oblika paralelograma. Izračunavaju se samo elementi uokvireni crnom zubastom linijom. Blok niti se sinhronizuje na kraju računanja svake dijagonale podpločica (trenutna dijagonala je osenčena ljubičasto). Niti moraju da izračunavaju veštačke elemente izvan pločice, jer u suprotnom ne bi bilo moguće sinhronizovati blok niti.

Vreme izvršavanja je dato na grafiku (1). Kraće je od Gpu7-Mlsp-DiagDiag algoritma (4.13), zbog toga što se mnogo više pristupa registrima umesto deljenoj memoriji. Takođe, algoritam ima malu potrošnju energije uzimajući u obzir prethodne GPU algoritme, datu na grafiku (11).

## 4.15 Gpu9-Mlsp-DiagDiag algoritam

Algoritam je malo modifikovani Gpu8-Mlsp-DiagDiag algoritam (4.14), gde se koristi više warp-ova za izračunavanje pločice.

Pločica se deli na podpločice. Svaku podpločicu računa tačno jedan warp (slika 14). Podpločica je oblika paralelograma, kako bi se izbegla povećana warp divergencija. U jednom trenutku se računa jedna dijagonala podpločica u pločici, nakon čega se svi warp-ovi u bloku niti sinhronizuju standardnim `_syncthreads()`. Ovo je urađeno kako bi se smanjio neophodan broj blok sinhronizacija, koje su skupe u odnosu na warp sinhronizacije. Sama podpločica se sinhronizuje sa `_shfl_up_sync()`, na isti način kao u Gpu8-Mlsp-DiagDiag algoritmu.

Komunikacija sa `tileHrow` i `tileHcol` u deljenoj memoriji se vrši na skoro identičan način kao u Gpu8-Mlsp-DiagDiag algoritmu. Razlika je u tome što nit 31 u *svakom* warp-u, svake iteracije upisuje svoj CURR element u `tileHrow`. Ovo je neophodno, kako bi nit 0 u svakom warp-u mogla da pročita korektnu vrednost za svoj UP element. Ovo je posledica toga da nije moguće koristiti `_shfl_up_sync()` za sinhronizaciju više warp-ova.

Algoritam povećava maksimalni faktor redukcije prostora skor matrice u odnosu na Gpu8-Mlsp-DiagDiag algoritam. Međutim, limitacija nije otklonjena, zbog toga što se i dalje jedna nit mpira na tačno jednu vrstu.

Pozitivne strane i mane algoritma su, pored toga, iste kao kod Gpu8-Mlsp-DiagDiag algoritma.

Vreme izvršavanja je dato na grafiku (1).

## 5 Pregled postojećih rešenja

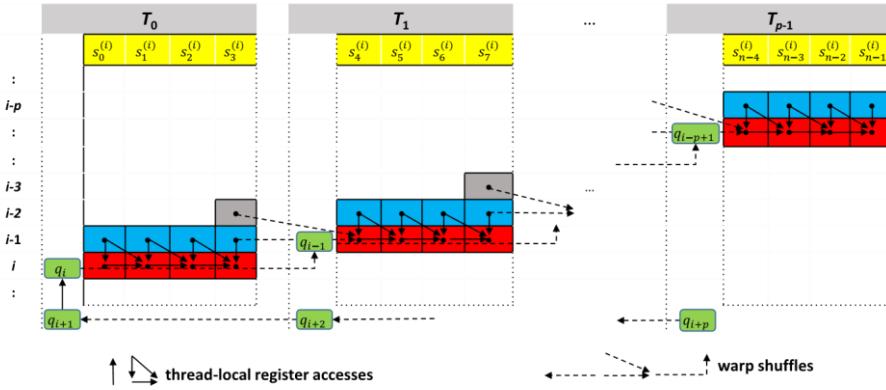
U nastavku će biti dat pregled postojećih relevantnih implementacija za Smith-Waterman [13] algoritam, pošto se sama obrada većinski odvija na isti način kao u Needleman-Wunsch algoritmu, a dati algoritam se daleko više koristi i optimizuje u praksi. Način obilaska i paralelizacija obrade skor matrice će biti poređena sa algoritmima implementiranim u ovom radu.

### 5.1.1 CUDASW++4.0: ultra-fast GPU-based Smith–Waterman protein sequence database search

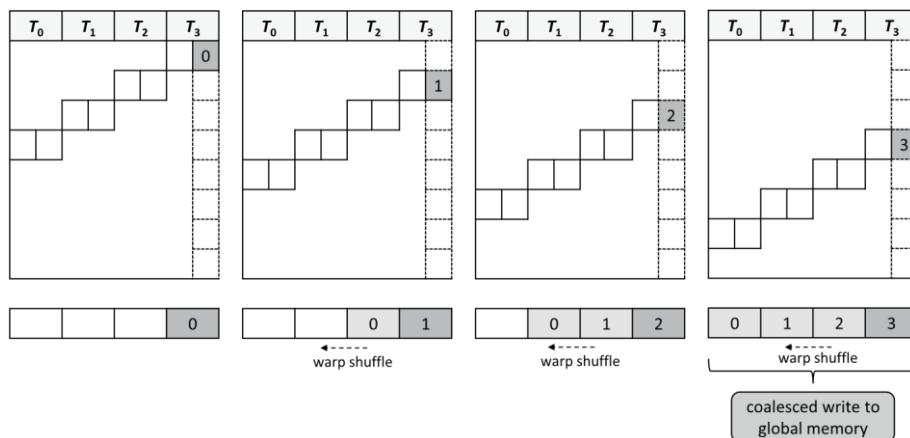
U navedenom radu [14] poravnava se upit proteinska sekvenca sa velikom bazom sekvenci korišćenjem GPU-a. Cilj je pronaći one sekvene u bazi koje su najsličnije upit sekveni, dok rekonstrukcija poravnanja nije potrebna. Na primeru *SwissProt* baze, baza poseduje ~500M sekvenci prosečne dužine ~350 i maksimalne ~35000 aminokiselina [14], dok su isprobane upit sekvene ispod ~5K aminokiselina.

Navedeni rad [14] bavi se optimizacijom poravnanja velikog broja sekveni, što je drugačiji problem od ovog rada. Međutim, ideje za optimizaciju koje su predstavljene su prenosive i na problem poravnanja dve jako dugačke sekvene ( $>1M$  elemenata). Relevantne ideje su:

- Korišćenje CUDA warp shuffle-a za razmenu podataka između niti (slika 15) – omogućava da se računanje skor matrice skoro potpuno uradi u registrima warp-a, čime se smanjuje pristup deljenoj memoriji. To dalje vodi do povećanja GCUPS-a (engl. *giga cell update per second*) i smanjenja potrošnje energije po GCUPS-u. Navedeni pristup je primenjen u algoritmima Gpu7-Mlsp-DiagDiag (4.13) nadalje.
- Korišćenje novijih CUDA DPX instrukcija za ubrzanje računanja skor funkcije. Optimizovanje vrućeg dela koda u asembleru drastično povećava protočnost algoritma. Primena ove ideje bi mogla značajno povećati performanse algoritama u ovom radu, pošto vrući delovi koda nisu optimizovani na taj način.
- Izbor tipa podataka za elemente skor matrice – `float`, `half2`, `int32`, `s16x2`. Omogućava izbor tipa koji daje najbolje performanse na konkretnoj CUDA arhitekturi. Ukoliko element skor matrice izazove overflow/underflow (prekoračenje) prilikom izračunavanja, tada se cela računica ponovi koristeći veći tip podataka. Ova ideja bi se mogla primeniti na algoritmima u ovom radu.
- Podela warp-a na manje subwarp-ove veličine  $2^n$ ,  $n \leq 5$  niti. Omogućava da se smanji pristup deljenoj memoriji za potrebe čitanja sekvenci  $\vec{x}$  i  $\vec{y}$ , korišćenjem warp shuffle-a u okviru subwarp-a. Ova ideja bi se mogla primeniti u algoritmima Gpu7-Mlsp-DiagDiag (4.13) nadalje.
- Podela sekvene duže od  $k$  aminokiselina na delove dužine  $k$  (slika 16). U jednom poravnjanju upit sekvene sa sekvencom iz baze se tada poravna  $k$  kolona skor matrice. Poslednja kolona iz skor matrice se tada koristi kao heder sekvenca za narednih  $k$  kolona. Slična ideja je primenjena nad vrstama i kolonama pločice u algoritmima Gpu7-Mlsp-DiagDiag (4.13) nadalje.



Slika 15 – mapiranje niti u *CUDASW++4.0* na elemente skor matrice. Niti koriste `_shfl_up_sync()` i `_shfl_down_sync()` kako bi međusobno razmenili elemente (isprekidane linije). Na ovom primeru, 4 niti saraduju kao deo subwarp-a.



Slika 16 – podela dugačke sekvene iz baze na delove od  $k$  kolona. Omogućava da se poravnjanje sekvene podeli na faze, gde faza kao rezultat proizvede poslednju kolonu. Ta kolona je deo ulaza u sledećoj fazi. Time se omogućava poravnjanje sekvenci dužih od ~1K aminokiselina.

Kroz primenu navedenih ideja, pokazano je da ova implementacija dobro koristi potencijal modernih GPU-ova, što se ogleda u velikom GCUPS-u i maloj potrošnji energije po jednom GCUPS-u.

### 5.1.2 *Exact pairwise alignment of mega-base genome biological sequences using a novel z-align parallel strategy*

U navedenom radu [2] poravnavaju se dve jako dugačke DNK sekvene (~3M DNK baza, maksimalno do 35M×5M) korišćenjem CPU-a. Cilj je pronaći optimalno lokalno poravnjanje i sekvencu izmena za to poravnjanje. Neke od optimizacija su:

- Korišćenje algoritma za rekonstrukciju poravnjanja koji radi u linearnom prostoru [12]. Obično skor matricu s leva na desno, odozgo na dole, i u obrnutom smeru. U oba smera dolazi do dve srednje vrste u matrici, i u preseku pronalazi element koji se sigurno nalazi na optimalnoj putanji. Zatim deli matricu na četiri dela, i ponavlja navedene korake na gornjem levom i donjem desnom delu, dok se ne dode do dovoljno malih podmatrica za koje se rekonstrukcija poravnjanja može brže izračunati standardnim algoritmom. Ukoliko bi se

poravnavale sekvence dužine od 1M DNK baza bez ove optimizacije, potpuna skor matrice bi zauzimala 1PB memorije, što je daleko od prihvatljivog. Ova optimizacija bi se mogla primeniti u CPU i GPU algoritmima u ovom radu, a posebno u rekonstrukciji poravnjanja korišćenoj u algoritmima Gpu7-Mlsp-DiagDiag (4.13) nadalje, kod kojih rekonstrukcija poravnjanja ima veliki ideo u vremenu izvršavanja (grafik 5).

- Podela matrice na velike podmatrice, i korišćenje MPI i klastera od 64 procesora za komunikaciju i sinhronizaciju izračunavanja dijagonala podmatrica. Ova ideja se lako može primeniti u algoritmima u ovom radu. Potencijalno bolja optimizacija je da se posao deli na više GPU-ova, umesto na više CPU-ova.

### 5.1.3 *SW#—GPU-enabled exact alignments on genome scale*

U navedenom radu [3] poravnavaju se dve jako dugačke DNK sekvence (~3M DNK baza, maksimalno do 33M×47M) korišćenjem GPU-a. Cilj je pronaći optimalno lokalno poravnanje i sekvencu izmena za to poravnanje. Relevantne optimizacije su:

- Korišćenje algoritma za rekonstrukciju poravnjanja koji radi u linearnom prostoru [12]. U odnosu na optimizaciju u radu [2], razlika je u tome što se poravnanje odvija većinski na GPU-u.
- Podrška za izračunavanje na više GPU-ova.

## 6 Rezultati

### 6.1 Metodologija

Pre merenja performansi su varirani parametri za svaki algoritam, kroz poravnanje dve sekvence dužine 5K aminokiselina. Izabrano je  $k$   $n$ -torki parametara koji dovode do  $k$  najkraćih vremena izvršavanja, a od njih je odabrana ona  $n$ -torka koja najbolje koristi CPU/GPU resurse. Verifikovano je da sva poravnanja istih sekvenci daju istu konačnu cenu i sekvencu izmena, kao i skor matricu. U nastavku se koriste dati parametri prilikom puštanja algoritama.

Ispravnost algoritama u radu je dodatno proverena kroz poravnanje 50 veštački generisanih sekvenci dužina između 100 i 23K aminokiselina.

Performanse algoritama su merene alatima *Hyperfine* (CPU metrike), *Nvidia Nsight Compute* (GPU metrike) i *NVML Python* biblioteka (GPU potrošnja energije). Alati su pokrenuti na benčmark kodu (engl. *benchmark*), u kom se poravnavaju dve sekvence dužine 23K aminokiselina, za pojedinačni algoritam ili za sve raspoložive algoritme odjednom.

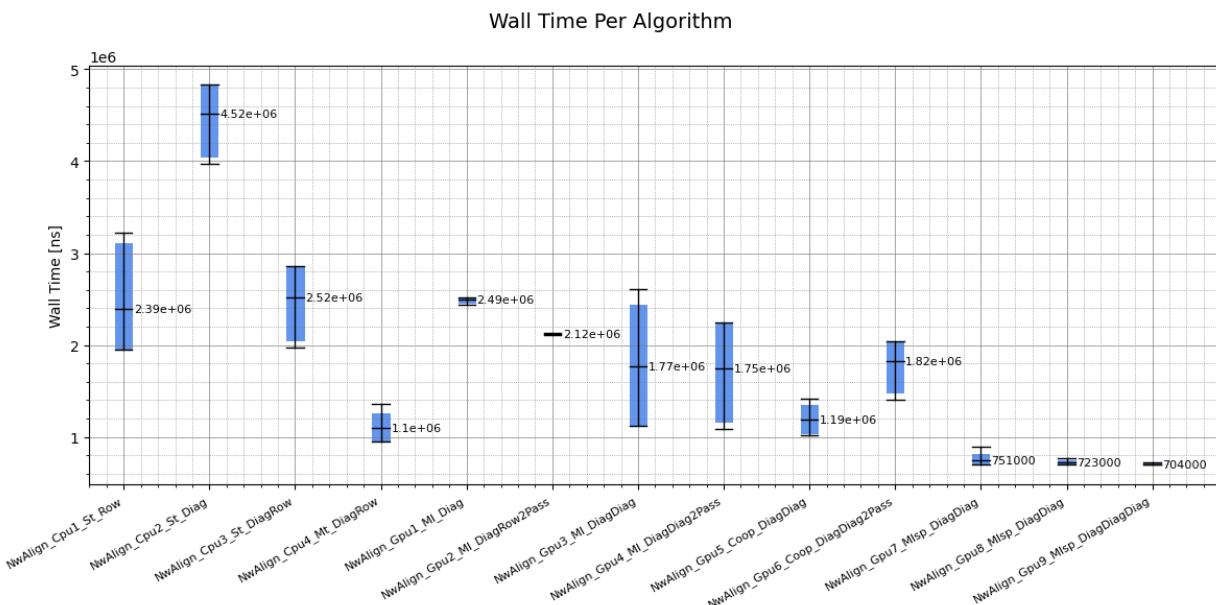
- *Nsight Compute* je pokrenut za sve algoritme odjednom u režimu ponavljanja CUDA grafa/kernela, kako bi stabilizovao performanse datih grafova/kernela u izveštaju.
- *Hyperfine* je pokrenut za svaki algoritam pojedinačno u trajanju od 5s po algoritmu, što je obuhvatilo 7-8 pokretanja po algoritmu.
- *NVML* je pokrenut za svaki algoritam pojedinačno, gde je za jedan algoritam poravnanje 23K sekvenci ponavljano u trajanju od 5s po algoritmu, kako bi se dobilo  $\sim 100$  merenja po algoritmu za Gpu1-Gpu6 ([4.7 - 4.12](#)), odnosno  $\sim 30$  merenja za Gpu7-Gpu9 ([4.13 - 4.15](#)).

Dodatno, meren je i vremenski ideo faza poravnanja u totalnom vremenu izračunavanja, u okviru frejmворка за pozivanje algoritama (engl. *framework*). Poravnate su sekvence istih dužina, gde se površina skor matrice linearno povećava u rasponu  $20^2$  do  $23700^2$  sa 30 merenja, za sve algoritme odjednom.

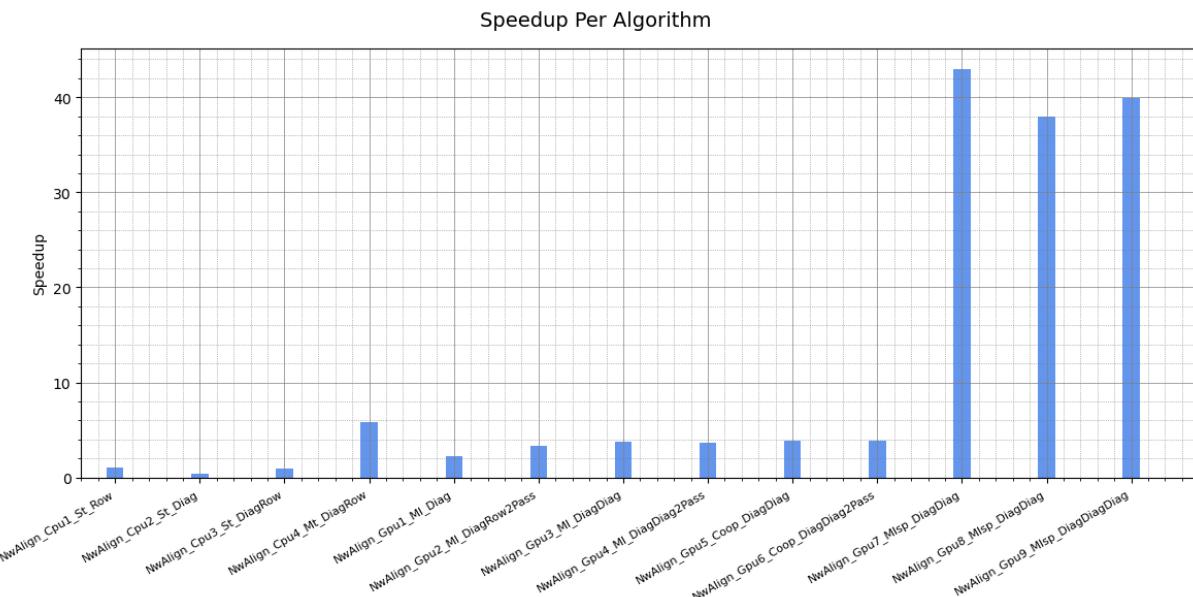
U svim poravnanjima je korišćena linearna skor funkcija sa BLOSUM62 supstitionom matricom i  $g_o = 11$ . Merenja su vršena na sistemu opisanom u ([3.5](#)).

### 6.2 CPU-specifične metrike

Vremena izvršavanja algoritama su data na grafiku ([1](#)). Dosta variraju za algoritme Cpu1-Cpu3 i Gpu3-Gpu6, dok su stabilna za algoritme Gpu1, Gpu2 i sparse algoritme Gpu7-Gpu9. Cpu2 algoritam ima najlošije performanse, jer ima loš šablon pristupa operativnoj memoriji (grafik [3](#)). Najbolja vremena izvršavanja imaju sparse algoritmi Gpu7-Gpu9, jer su efektivno računski-ograđeni za poravnavanje sekvenci dužine 23K (grafik [5](#)). Iznenađujuće je što na grafiku ([1](#)) vreme izvršavanja Cpu4 algoritma nije višestruko kraće od Cpu1 algoritma (okvirno `#cpu_threads` puta, gde je  $n$  broj hardverskih CPU niti), međutim *end-to-end* vreme izvršavanja uzima u obzir i učitavanje sekvenci i rekonstrukciju, kao što je slučaj u stvarnom okruženju. Ukoliko se režijski troškovi framework-a izbace, tada postaje jasno da sparse algoritmi imaju za red veličine bolja vremena izvršavanja u odnosu na ostale algoritme (grafik [2](#)).



Grafik 1 – end-to-end vreme izvršavanja po algoritmu (manje je bolje)<sup>1</sup>



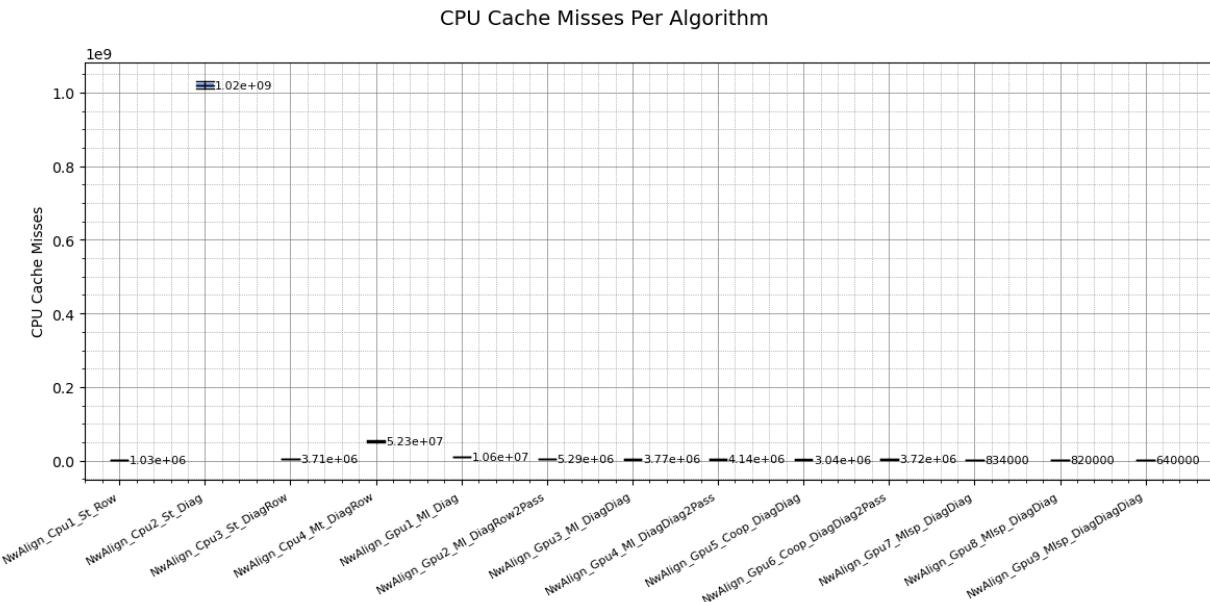
Grafik 2 – ubrzanje vremena izvršavanja u odnosu na Cpu1 algoritam po algoritmu (više je bolje).

Posmatra se vreme poravnjanja i rekonstrukcije poravnjanja, bez režijskih troškova framework-a.

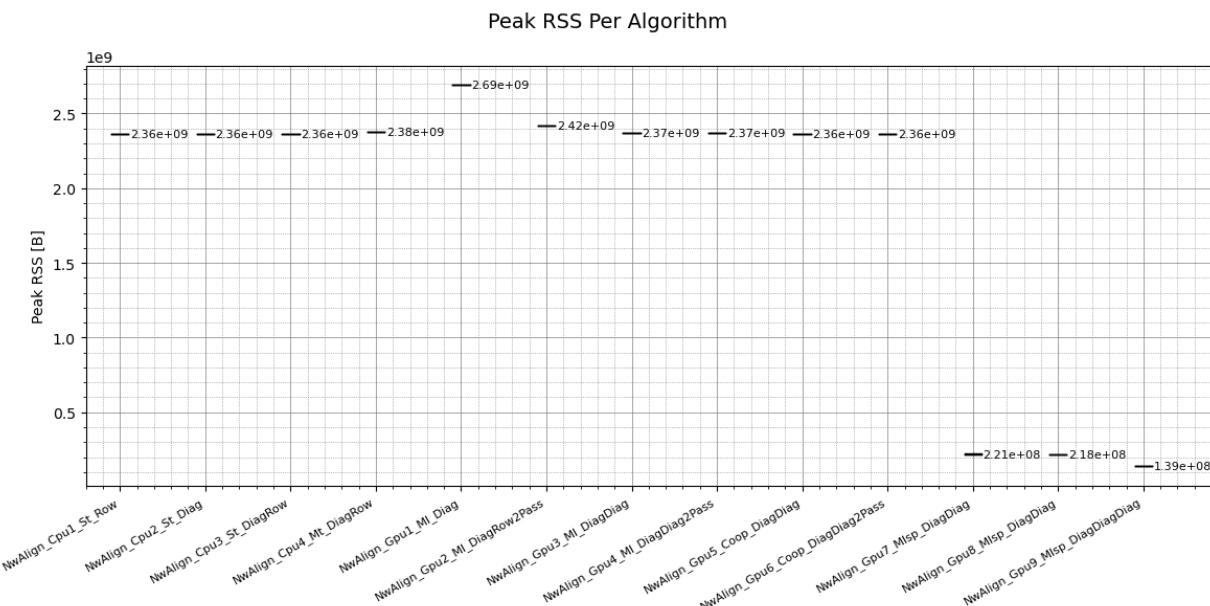
Sa povećanjem dužina sekvenci koje treba poravnati, svi režijski troškovi u okviru poravnjanja i rekonstrukcije poravnjanja praktično nestaju, osim memorijskih transfera (grafik 5). Kod poravnjanja sekvenci dužine 23K, algoritmi Cpu1-Cpu4 su računski-ograničeni, dok su Gpu1-Gpu6 algoritmi memorijski-ograničeni. Međutim, sparse algoritmi podižu granicu za memorijsko-ograđenje, koja se dostiže tek za značajno duže sekvene.

Ukoliko posmatramo korišćenje RAM memorije (grafik 4), svi ne-sparse algoritmi imaju sličnu

<sup>1</sup> Pravougaonik predstavlja standardnu devijaciju, a crne linije predstavljaju odozdo-naviše minimalnu, očekivanu i maksimalnu vrednosti. Poravnavaju se dve sekvene dužine 23K.

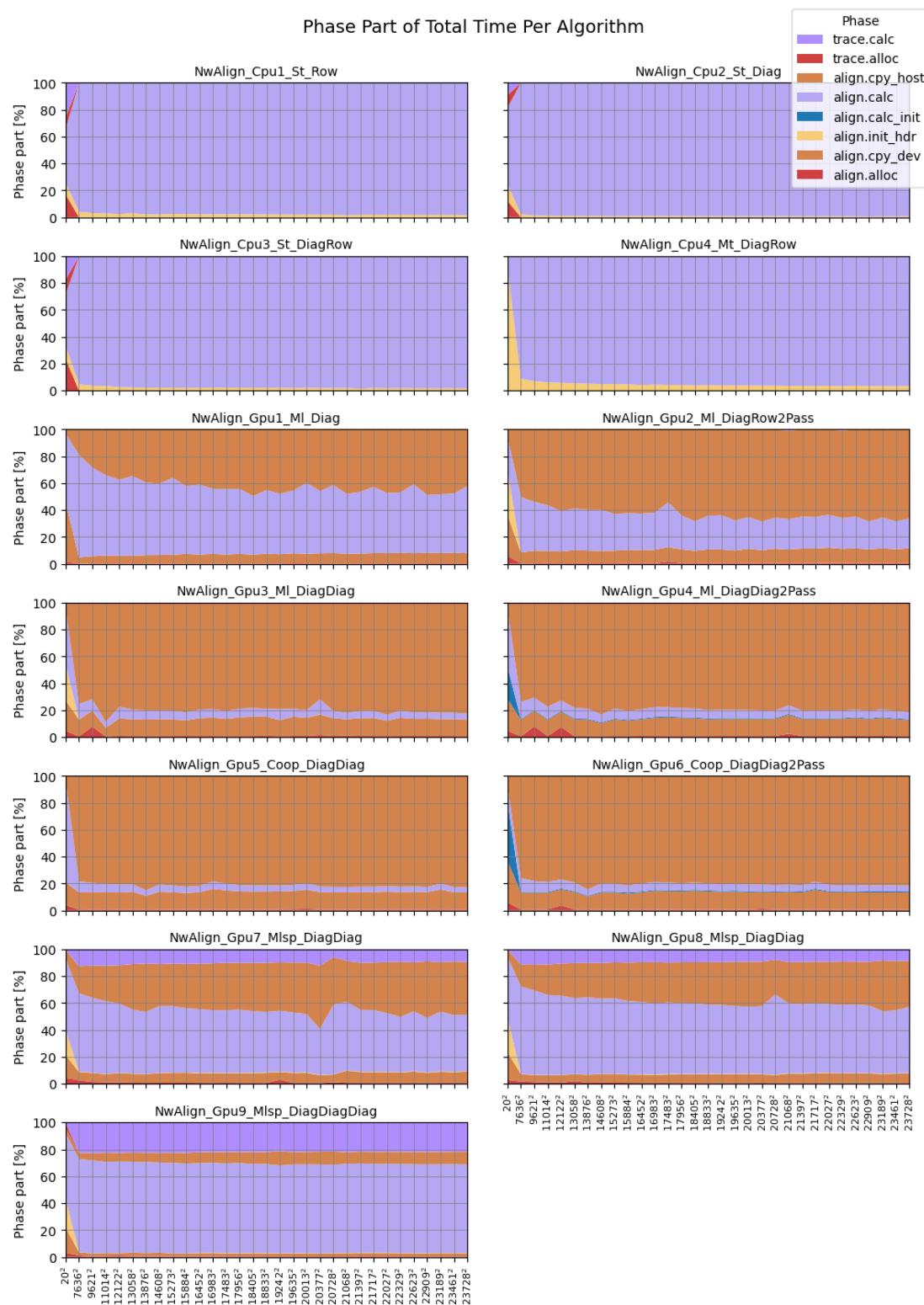


Grafik 3 – CPU keš promašaji po algoritmu (manje je bolje)



Grafik 4 – maksimalna potrošnja RAM memorije po algoritmu (manje je bolje)

potrošnju, dok *sparse* algoritmi koriste značajno manje prostora za iste dužine sekvenci. Svi algoritmi imaju kvadratnu prostornu složenost, koja bi se mogla dalje poboljšati primenom algoritma [12], i time pretvorila problem u računski-ograničen za proizvoljnu dužinu dužih sekvenci.



Grafik 5 – ideo faza u vremenu izvršavanja po algoritmu. Faze su (odozdo-naviše):

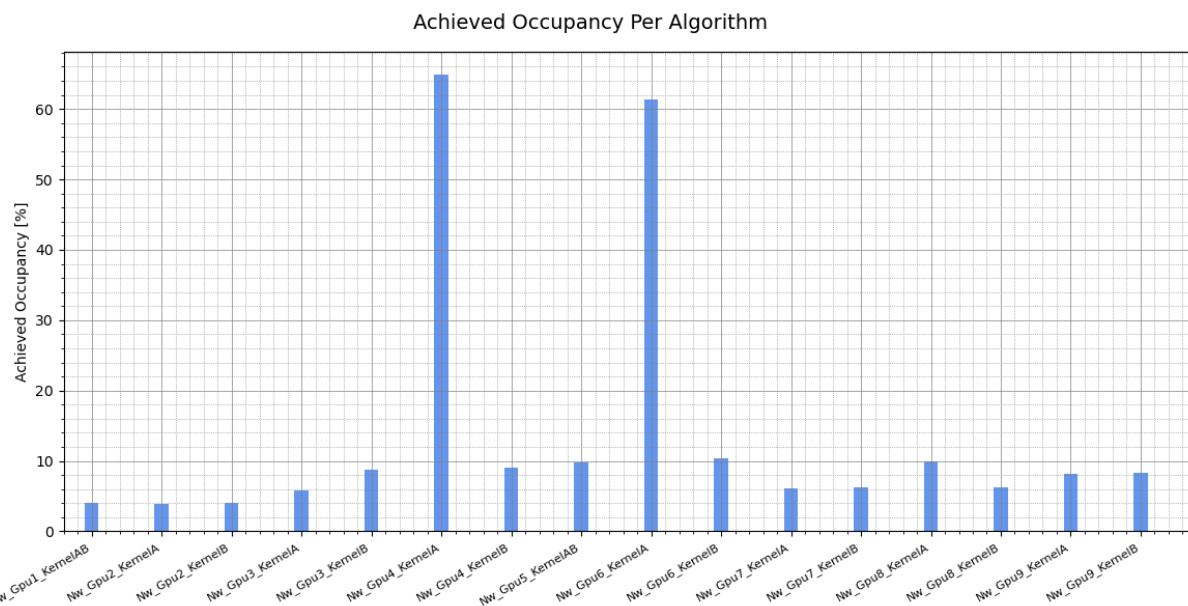
1. rekonstrukcija poravnjanja (manje je bolje),
2. alokacija memorije za rekonstrukciju poravnjanja (manje je bolje),
3. poravnjanje sekvenci (više je bolje),
  1. (prisutno u nekim algoritmima) inicijalizacija matrica pre poravnjanja,
  2. inicijalizacija hedera matrica,
4. alokacija memorije za poravnjanje (manje je bolje).

## 6.3 GPU-specifične metrike

U okviru GPU implementacija algoritama<sup>1</sup>, postoji nekoliko metrika kojima se mogu poređiti sami kerneli/grafovi kornela<sup>2</sup>. Ukoliko se posmatra postignut occupancy, odnosno warp-is-korišćenost SM-a po GPU kernelu (grafik 6), primećuje se da je relativno sličan za sve kernele sa vrednosti 5-10%. Kerneli Gpu4A i Gpu6A odstupaju i imaju veoma dobru iskorišćenost, jer inicijalizuju dovoljnu količinu podataka (celu skor matricu) na jednostavan način. Ako ih izuzmemos, zaključuje se da je dijagonalni način obilaska fundamentalna limitacija za povećanje iskorišćenosti SM-ova, ali istovremeno i jedini poznat način da se omogući veći paralelizam na GPU-u. Potencijalno je moguće povećati occupancy algoritama ukoliko bi se optimizovao kod za računanje elementa skor matrice, kroz ručnu optimizaciju NVPTX asemblerских instrukcija.

Broj warp ciklusa po izvršenoj instrukciji (grafik 7) relativno slabo varira između GPU kornela. Gpu9B kernel ima najbolju vrednost. Na tu metriku utiče šablon pristupa globalnoj memoriji, od kojih Gpu9B kernel ima najmanje frekventne pristupe i to sa najmanje transfera. Interesantno je da Gpu2A i Gpu3A imaju loše vrednosti ove metrike, iako im je obrada vrlo jednostavna, zbog nedovoljnog broja instr. sa kojima warp raspoređivač raspolaže da sakrije režijske troškove.

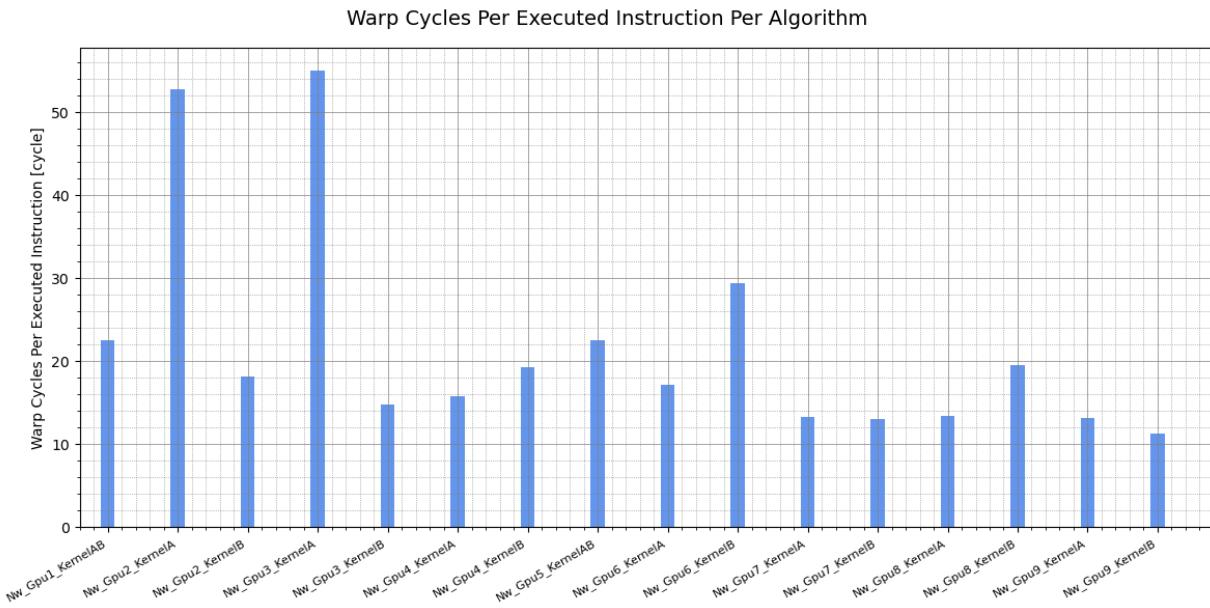
Prosečan broj aktivnih niti po warp-u (grafik 8) takođe slabo varira između GPU kornela. Postoji mali pad u odnosu na ostale vrednosti u Gpu8B kernelu. Ovo može biti rezultat usložnjavanja načina pristupa deljenoj memoriji Gpu7B kernela, kao međukorak između Gpu7 i Gpu9 algoritama, zarad malih dobitaka.



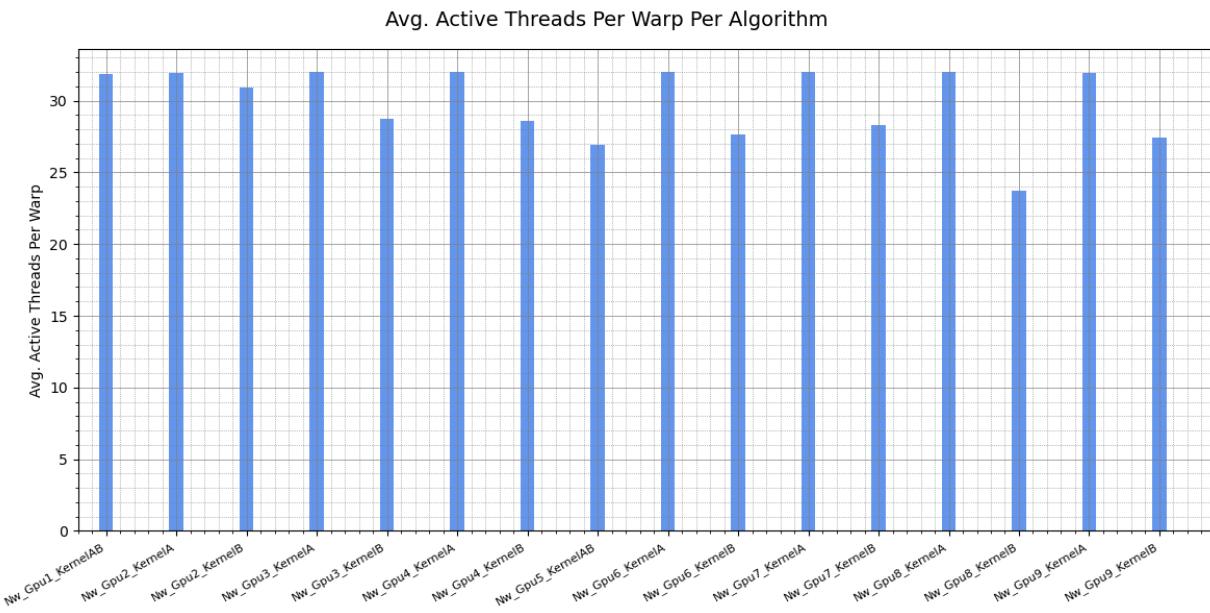
Grafik 6 – warp-iskorišćenost SM-a po GPU kernelu (više je bolje). Kerneli tipa A se kratko izvršavaju zbog obrade male količine podataka, tako da ne postižu velik occupancy iako im je obrada jednostavna.

<sup>1</sup> Svaki GPU algoritam pokreće jedan ili dva kernela, koji obavljaju fazu inicijalizacije (A), fazu izračunavanja (B), ili obe faze (AB) u okviru jednog kernela. Kerneli tipa A inicijalizuju heder vrstu i kolonu skor matrice, osim kernela Gpu6A koji dodatno inicijalizuje i celu skor matricu. Kerneli tipa B rade preostali, složeniji deo izračunavanja.

<sup>2</sup> CUDA grafovi se koriste da se odjednom pokrene niz kernela prateći zavisnosti u grafu umesto višestrukog pokretanja kernela iz host koda, kako bi bilo moguće koristiti graf-ponavljanje u Nsight Compute-u.



Grafik 7 – broj warp ciklusa po izvršenoj instrukciji po GPU kernelu (manje je bolje)

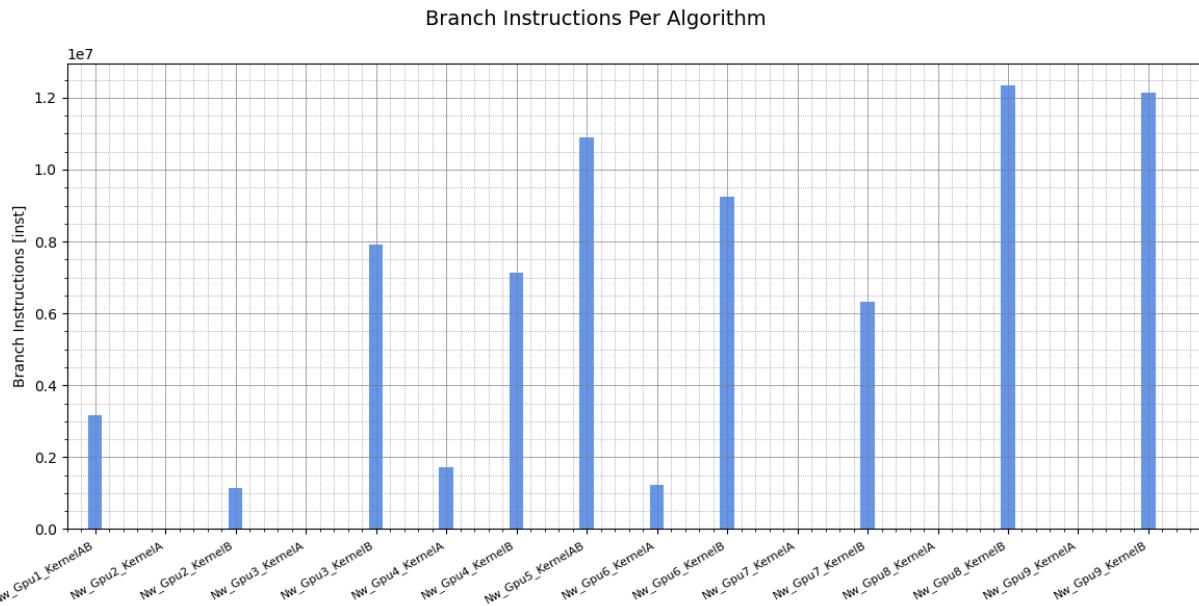


Grafik 8 – prosečan broj aktivnih niti po warp-u po GPU kernelu (više je bolje)

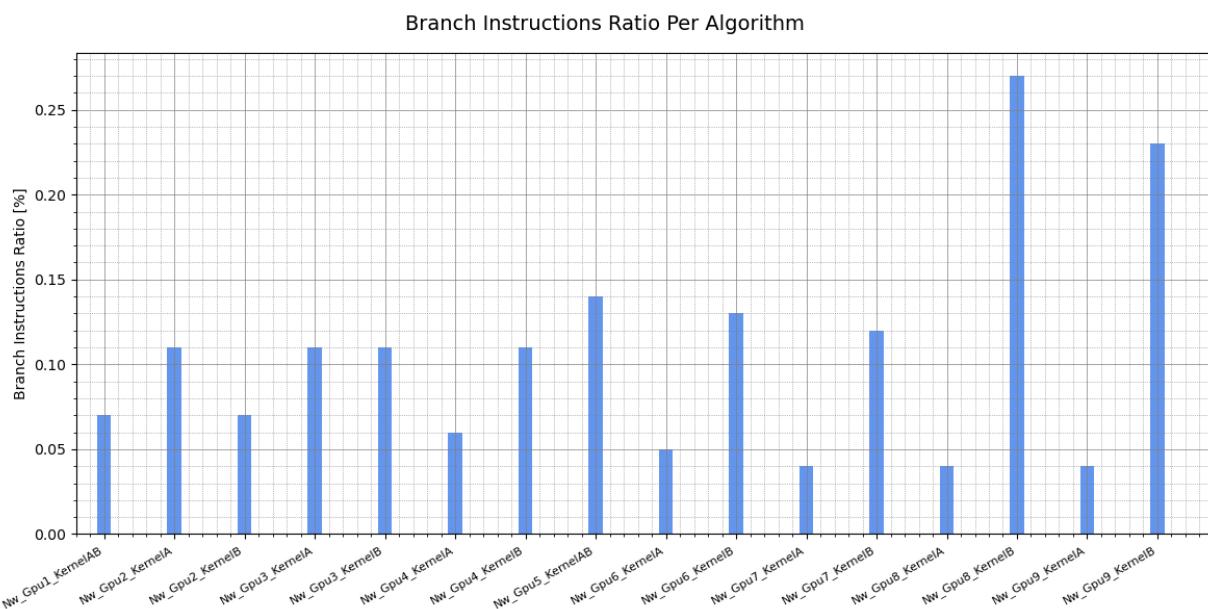
Broj instrukcija skoka (grafik 9) i odnos broja instrukcija skoka sa ukupnim brojem instrukcija (grafik 10) ukazuju na to da su Gpu8B i Gpu9B kerneli dosta složeniji u šablonu pristupa. Vruć deo koda u Gpu9B kernela bi trebalo optimizovati da bi se ova metrika poboljšala, kao što je ranije navedeno. Prema ovoj metrići, Gpu7B algoritam je bolji od Gpu8 i Gpu9.

Ako posmatramo potrošnju energije (grafik 11), najbolje se pokazuju algoritmi Gpu7-Gpu9, zbog značajno povećanog udela pristupa registrima u odnosu na druge GPU memorije.

Uzimajući navedene GPU metrike u obzir, sparse algoritmi Gpu7 i Gpu9 su najperformantniji. Gpu7 algoritam ima malo veće računske performanse od Gpu9 algoritma jer je jednostavniji,

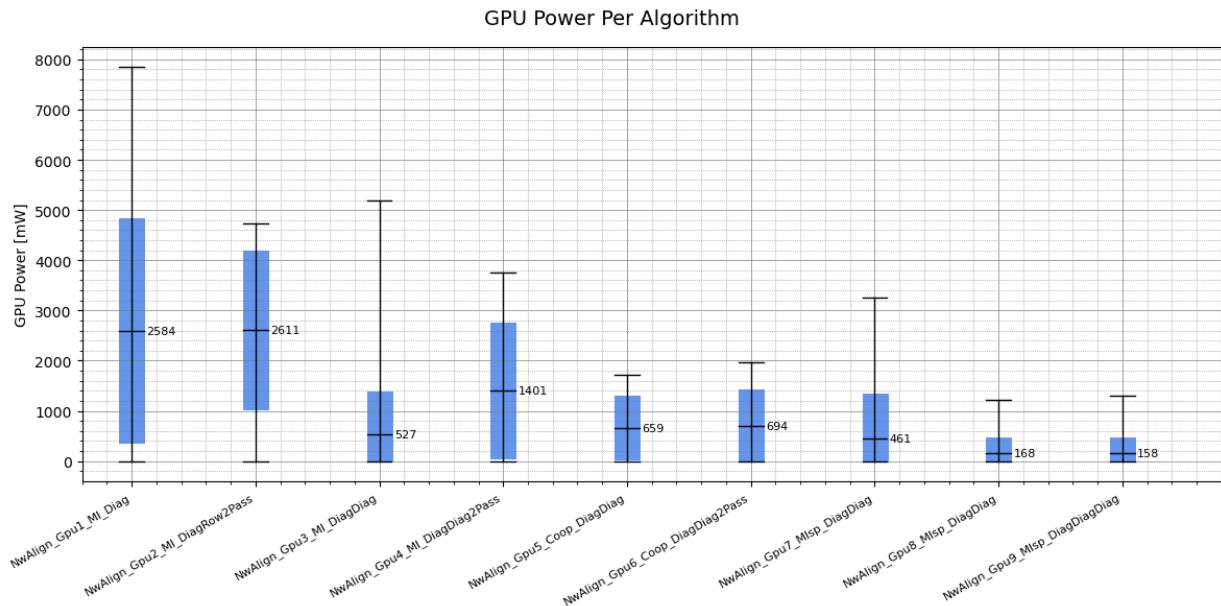


Grafik 9 – broj instrukcija skoka po GPU kernelu (manje je bolje). Kerneli koji inicijalizuju heder vrstu i kolonu skor matrice imaju jako malu vrednost, jer obrađuju malu količinu podataka.



Grafik 10 – odnos broja instrukcija skoka sa ukupnim brojem instrukcija po GPU kernelu (manje je bolje)

dok Gpu9 algoritam može da poravna sekvence dužine do 0.5M i ima manji memorijski transfer. Moguća su dalja unapredjenja Gpu9 algoritma za podršku poređenja još dužih sekvenci.



Grafik 11 – potrošnja energije po GPU kernelu (manje je bolje). Grafik sadrži šum sa relativno velikom varijansom pošto sa NVML bibliotekom nije moguće izmeriti potrošnju pojedinačnog procesa. Odabrana je NVML biblioteka umesto *nvidia-smi* alata koji može da meri potrošnju konkretnog procesa, jer ne podržava rezoluciju merenja manju od 0.5s.

## 7 Zaključak i dalji rad

U ovom radu su poređene performanse različitih implementacija Needleman-Wunsch algoritma na CPU-u i GPU-u. Uočene su prednosti i mane svake implementacije, na osnovu kojih su napravljena poboljšanja u vidu narednih algoritama. Kroz takvu genezu algoritama je otkriven nov način za podelu pločice matrice na podpločice oblika paralelograma (Gpu9-Mlsp-DiagDiagDiag algoritam ([4.15](#))), kod koga se radi manji broj sinhronizacija bloka niti u odnosu na dosadašnje poznate implementacije. Potvrđena je hipoteza da izračunavanje skor matrice – skoro potpuno u registrima niti pomoću CUDA warp intrinsic-a – dosta povećava performanse. Gpu9 algoritam se pokazao kao najbolji po više metrika.

Primećeno je da se Gpu9 algoritam može unaprediti u Gpu10 algoritam, koji bi matricu delio na super-pločice. Super-pločice bi bile računate sa modifikovanim Gpu7 algoritmom, kod koga bi se čuvale samo heder vrste i kolone pločica na trenutnoj dijagonali. Gpu10 algoritam bi tada čuvao heder vrste i kolone super-pločica u globalnoj i operativnoj memoriji. Time bi faktor prostorne uštede bio značajno veći od Gpu9 algoritma. Algoritam za rekonstrukciju poravnjanja bi pozivao modifikovani Gpu7 algoritam da bi dobio trenutnu super-pločicu, da bi zatim pozivao postojeću sparse rekonstrukciju na njoj. Prema pretpostavci, takav algoritam bi mogao da poredi sekvene mnogo duže od 1M elemenata i bio bi računski-ograničen.

Kao dalja unapređenja performansi bi se mogao ručno optimizovati NVPTX asemblerski kod u vrućoj petlji u Gpu9 algoritmu, implementirati algoritam poravnjanja zajedno sa algoritmom rekonstrukcije poravnjanja koji rade u linearnom prostoru [\[12\]](#) i dodati podrška za multi-GPU izvršavanje. Mogla bi se koristiti Afina skor funkcija jer bolje oslikava biološke procese od linearne. Konačno, poslednji algoritam bi se mogao izmeniti tako da radi lokalno (Smith-Waterman [\[13\]](#)) poravnanje.

## 8 Reference

- [1] S. B. Needleman and C. D. Wunsch, ‘A general method applicable to the search for similarities in the amino acid sequence of two proteins’, *Journal of Molecular Biology*, vol. 48, no. 3, pp. 443–453, Mar. 1970, doi: 10.1016/0022-2836(70)90057-4.
- [2] A. Boukerche, R. B. Batista, and A. C. M. A. De Melo, ‘Exact pairwise alignment of megabase genome biological sequences using a novel z-align parallel strategy’, in *2009 IEEE International Symposium on Parallel & Distributed Processing*, Rome, Italy: IEEE, May 2009, pp. 1–8. doi: 10.1109/IPDPS.2009.5161113.
- [3] M. Korpar and M. Šikić, ‘SW#—GPU-enabled exact alignments on genome scale’, *Bioinformatics*, vol. 29, no. 19, pp. 2494–2495, Oct. 2013, doi: 10.1093/bioinformatics/btt410.
- [4] B. Schmidt, F. Kallenborn, A. Chacon, and C. Hundt, ‘CUDASW++4.0: ultra-fast GPU-based Smith–Waterman protein sequence database search’, *BMC Bioinformatics*, vol. 25, no. 1, p. 342, Nov. 2024, doi: 10.1186/s12859-024-05965-6.
- [5] M. Brudno *et al.*, ‘Glocal alignment: finding rearrangements during alignment’, *Bioinformatics*, vol. 19, no. suppl\_1, pp. i54–i62, Jul. 2003, doi: 10.1093/bioinformatics/btg1005.
- [6] ‘A One-Letter Notation for Amino Acid Sequences\*: Tentative Rules’, *European Journal of Biochemistry*, vol. 5, no. 2, pp. 151–153, Jul. 1968, doi: 10.1111/j.1432-1033.1968.tb00350.x.
- [7] S. Henikoff and J. G. Henikoff, ‘Amino acid substitution matrices from protein blocks.’, *Proc. Natl. Acad. Sci. U.S.A.*, vol. 89, no. 22, pp. 10915–10919, Nov. 1992, doi: 10.1073/pnas.89.22.10915.
- [8] ‘CUDA C Programming Guide 12.9’. 2025. Accessed: May 16, 2025. [Online]. Available: <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>
- [9] ‘NVIDIA Ampere GA102 GPU Architecture Whitepaper 2.1’. 2021. Accessed: May 16, 2025. [Online]. Available: <https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.1.pdf>
- [10] ‘OpenMP API Specification 6.0’. 2024. [Online]. Available: <https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-6-0.pdf>
- [11] ‘NVIDIA RTX Blackwell GPU Architecture’. 2025. Accessed: May 16, 2025. [Online]. Available: <https://images.nvidia.com/aem-dam/Solutions/geforce/blackwell/nvidia-rtx-blackwell-gpu-architecture.pdf>
- [12] E. W. Myers and W. Miller, ‘Optimal alignments in linear space’, *Bioinformatics*, vol. 4, no. 1, pp. 11–17, Mar. 1988, doi: 10.1093/bioinformatics/4.1.11.
- [13] T. F. Smith and M. S. Waterman, ‘Identification of common molecular subsequences’, *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195–197, Mar. 1981, doi: 10.1016/0022-2836(81)90087-5.
- [14] B. Schmidt, F. Kallenborn, A. Chacon, and C. Hundt, ‘CUDASW++4.0: ultra-fast GPU-based Smith–Waterman protein sequence database search’, *BMC Bioinformatics*, vol. 25, no. 1, p. 342, Nov. 2024, doi: 10.1186/s12859-024-05965-6.