Paralelizacija operatora konvolucije nad slučajnim varijablama u programskom jeziku Bend

Krstačić, Rafael

SVEUČILIŠTE JURJA DOBRILE U PULI
FAKULTET INFORMATIKE

Rafael Krstačić

Paralelizacija operatora konvolucije nad slučajnim varijablama u programskom jeziku Bend

DIPLOMSKI RAD

Pula, rujan, 2024. godine

SVEUČILIŠTE JURJA DOBRILE U PULI
FAKULTET INFORMATIKE

Rafael Krstačić

Paralelizacija operatora konvolucije nad slučajnim varijablama u programskom jeziku Bend

DIPLOMSKI RAD

JMBAG: 0303092283, redoviti student
Studijski smjer: Informatika

Kolegij: Raspodijeljeni sustavi
Znanstveno područje: Društvene znanosti
Znanstveno polje: Informacijske i komunikacijske znanosti
Znanstvena grana: Informacijsko i programsko inženjerstvo

Mentor: doc. dr. sc. Nikola Tanković

Pula, rujan, 2024. godine

Sažetak

Ovaj diplomski rad istražuje računalne aspekte i tehnike optimizacije za operator konvolucije nad slučajnim varijablama u programskom jeziku Bend, s fokusom na pristupe paralelnog računanja. Konvolucija, temeljna operacija u raznim područjima kao što su obrada signala, obrada slike i teorija vjerojatnosti, može biti računalno intenzivna, osobito u kontekstu velikih skupova podataka. Ovaj rad istražuje različite metode za računalnu konvoluciju, fokusirajući se na tehnike diskretne konvolucije, i evaluira njihovu izvedbu u paralelnom računalnom okruženju.

Implementirana su dva različita pristupa za izračunavanje konvolucije: naivni algoritam kliznog skalarnig produkta i pristup redukcije stabla koji je paralelno orijentiran. Obje su metode implementirane u programskom jeziku Bend, koji je funkcijski jezik visoke razine dizajniran za paralelno izvođenje. Izvedba ovih implementacija analizirana je pomoću Bend-ovih Rust i C runtime-ova.

Rezultati pokazuju da je naivni pristup jednostavniji za implementaciju i još uvijek je bolji od tehnike redukcije stabla čak i u paralelnim okruženjima, budući da algoritam još uvijek nije optimiziran i ima usko grlo. Ovaj diplomski rad daje uvid u praktičnu primjenu paralelnog računanja u optimizaciji matematičkih operacija i doprinosi širem razumijevanju dizajna paralelnog algoritma.

Ključne riječi : Konvolucija, Paralelno Računanje, Bend programski jezik, Klizni Skalarni Produkt, Redukcija Stabla, CPU Višejezgreno Ubrzanje.

Abstract

This thesis investigates computational aspects and optimization techniques for the convolution operator over random variables in the Bend programming language, with a focus on parallel computing approaches. Convolution, a fundamental operation in fields as diverse as signal processing, image processing, and probability theory, can be computationally intensive, especially in the context of large data sets. This paper explores different methods for computational convolution, focusing on discrete convolution techniques, and evaluates their performance in a parallel computing environment.

Two different approaches are implemented to compute the convolution: a naive sliding dot product algorithm and a parallel-oriented tree reduction approach. Both methods are implemented in the Bend programming language, which is a high-level functional language designed for parallel execution. The performance of these implementations was analyzed using Bend's Rust and C runtimes.

The results show that the naive approach is simpler to implement and still outperforms the tree reduction technique even in parallel environments, since the algorithm is still not optimized and has a bottleneck. This thesis provides insight into the practical application of parallel computing in the optimization of mathematical operations and contributes to a broader understanding of parallel algorithm design.

Keywords : Convolution, Parallel Computing, Bend Programming Language, Sliding Dot Product, Tree Reduction, CPU Multi-Core Acceleration.

Sadržaj

1 Uvod

Konvolucija je temeljna matematička operacija koja ima ključnu ulogu u brojnim poljima, uključujući obradu signala, obradu slike, statistiku i inženjering. Koristi se za određivanje načina na koji oblik jedne funkcije mijenja druga, što ga čini važnim u raznim analitičkim i računalnim primjenama. Unatoč širokoj upotrebi, operacija konvolucije može biti računalno skupa, osobito kada se primjenjuje na velike skupove podataka.

Kako tehnologija napreduje, potražnja za bržim i učinkovitijim računanjem raste [1], a to ujedno čini motivaciju ovoga rada. Paralelno računanje omogućuje istovremeno izvođenje više izračuna, čime se smanjuje vrijeme potrebno za izvođenje tih izračuna. Ovo je osobito korisno u slučajevima kada se problem može rastaviti na manje, nezavisne zadatke koji se mogu rješavati paralelno. Paralelno računalstvo postiglo je velik uspjeh u projektiranju aeroprofila (optimiziranje uzgona, otpora, stabilnosti), motora s unutarnjim izgaranjem (optimiziranje distribucije naboja, sagorijevanje), brzih krugova (rasporedi za kašnjenja te kapacitivne i induktivne učinke), struktura (optimiziranje strukturni integritet, parametri dizajna, cijena itd.), a u novije vrijeme dizajn mikroelektromehaničkih i nanoelektromehaničkih sustava (MEMS i NEMS) privukao je značajnu pozornost [2].

Ovaj rad fokusira se na implementaciju i optimizaciju operacije diskretne konvolucije u kontekstu paralelnog računanja. Cilj je istražiti kako različiti algoritamski pristupi konvoluciji rade kada su podvrgnuti paralelnom izvođenju, specifično u programskom jeziku Bend. Bend pojednostavljuje paralelno računanje, dopuštajući programerima pisanje čistog i jednostavnog koda bez razmišljanja o uobičajenim problemima prisutnim u razvoju paralelnih algoritama.

U sljedećim poglavljima dotaknut će se matematičkih temelja konvolucije, principa paralelnog računanja i specifičnosti programskog jezika Bend. Zatim će se predstaviti dva pristupa implementaciji konvolucije: metodu naivnog kliznog skalarnog umnoška i tehniku redukcije stabla. Ove će se implementacije ocjenjivati na temelju njihovih performansi u paralelnim računalnim okruženjima, s fokusom na to kako iskorištavaju mogućnosti modernih hardverskih arhitektura.

GitHub repozitorij projekta: https://github.com/rkrstacic/bend-parallel

2 Konvolucija

U ovom poglavlju će se govori o teoriji konvolucije i objasniti kako je izračunati, dati nekoliko primjera primjene i istražiti neke od algoritama. Postoje dva oblika operacije, a to su kontinuirani i diskretni oblici. Ovaj rad se prvenstveno fokusira na diskretni oblik, no generalni koncept je sličan.

Konvolucija je matematička operacija nad dvije funkcije f i g koja proizvodi treću funkciju $(f * g)$. Točnije, definirana je na sljedeći način [3]:

$$(f * g)(x) = \int_0^x f(x)g(x-t)dt$$

ili, zbog komutativnosti operacije, formulom:

$$(f * g)(x) = \int_0^x f(x-t)g(x)dt$$

Taj je oblik kontinuirane konvolucije, to jest konvolucije nad kontinuiranim konstruktima kao što su funkcije. Kontinuirani oblik definiran je kao integral umnoška dviju funkcija nakon što se jedna odrazi oko y-osi i pomakne. Može se vizualizirati kao da se jedna funkcija isprepliće kroz okrenutu verziju duž y-osi druge i za svaki vremenski korak vrijednost konvolucije bila bi integral presjeka nakon što se dotaknu. Simbol t koji se koristi u formuli, ne mora nužno predstavljati vremensku domenu. Na svakom t, konvolucijska formula može se opisati kao površina ispod funkcije f(x) ponderirana funkcijom g(-x) pomaknuta za iznos t (gdje ulaz predstavlja neku apstraktnu ulaznu vrijednost).

Osim kontinuiranog oblika konvolucije postoji i diskretni oblik, koji radi nad nizovima, sekvencama, torkama i sličnim, a definira sljedećom formulom:

$$(S_1 * S_2)[x] = \sum_{i=0}^{n-1} S_1[i] * S_2[x-i]$$

ili, zbog komutativnosti operacije, formulom:

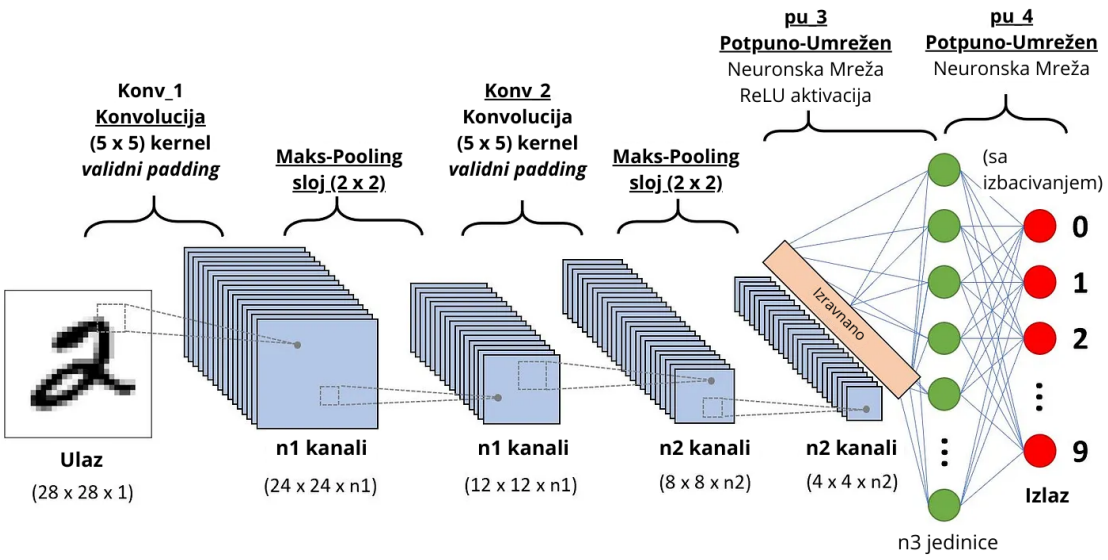
$$(S_1 * S_2)[x] = \sum_{i=0}^{n-1} S_1[x-i] * S_2[i],$$

Radi li se s torkama, rezultat konvolucije će isto tako biti torka, a veličina torke će iznositi 2n - 1, gdje je n veličina ulaznih torki. x-ti element rezultirajuće torke se može tumačiti kao skalarni produkt podskupa torki određene veličine gdje je jedna torka invertirana. Više o ovome govorit će se u odjeljku o kliznom skalarnom produktu. Diskretni oblik konvolucije se javlja, na primjer, kada se računa umnožak dvaju polinoma. Za dane polinome p_1 i p_2, koeficijenti umnoška ta dva su konvolucija koeficijenata od p_1 i koeficijenata od p_2. Treba skrenuti pozornost na to da ako nijedna funkcija nije obrnuta, izvodi se slična operacija, koja se naziva unakrsna korelacija [4].

Inače, konvolucija se koristi za filtriranje, uklanjanje šuma, otkrivanje rubova u slici, korelaciju, kompresiju, dekonvoluciju, simulaciju i za mnoge druge primjene

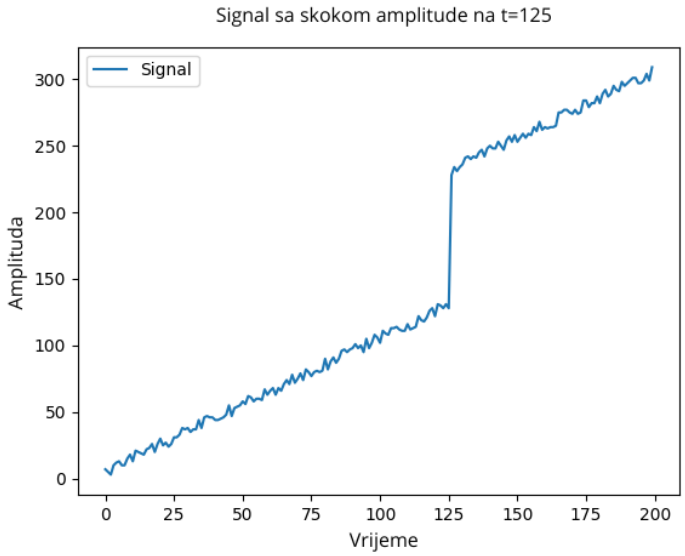
[5]. Na primjer, uzme li se konvolucija neke slike i takozvanog kernela određene veličine (na primjer 3 piksela širok sa 3 piksela visok), ono što će se dobiti jest nova slika koja će biti zamućena ovisno o strukturi kernela. Svaki rezultirajući piksel zamućene slike će u svom koraku izračuna imati vrijednost ovisnu o svojim susjednim pikselima a kernel daje težinu, to jest naglasak koliko su ti susjedi bitni za njegovu promjenu. Konvolucija također ima primjenu u vjerojatnosti, statistici, akustici, spektroskopiji, obradi signala i, geofizici, inženjerstvu, fizici, računalnom vidu i diferencijalnim jednadžbama... Predstavlja kako oblik jedne funkcije mijenja druga [6]. Može se također opisati kao radnja promatračkog instrumenta kada uzima ponderiranu srednju vrijednost neke fizičke veličine u uskom rasponu neke varijable [4].

Kao što je Pavle naveo, jedna od učestalih primjena konvolucije jest u računalnom vidu. Specifično, konvolucijska neuronska mreža koja ima sposobnosti pronaći obrasce u slici da se prepozna objekt. Djeluje slično kao već spomenut efekt zamućivanja slike. Na danu sliku se primjeni konvolucijski sloj određene veličine kernela koji onda pomaže neuronima mreže da znaju nešto više o svojoj okolini i na taj način prepoznaju određene značajke slike koje pomažu mreži da donese odluku [7]. Slika ispod prikazuje prikaz rada konvolucijske neuronske mreže gdje se na ulazu pojavljuje slika koja putuje kroz kernele veličina 5x5 i max-pooling slojeva kako bi na kraju dali težinu brojevima od 0-9:

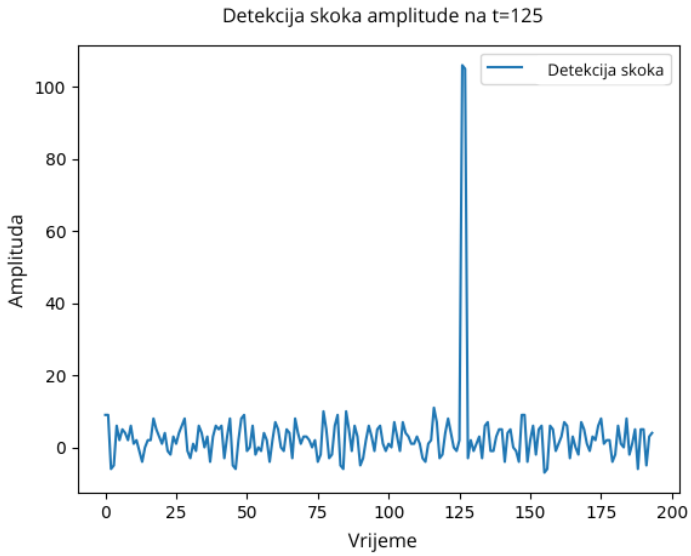


Slika 1: Primjer konvolucijske neuronske mreže [8]

Ako se uzme za primjer detekcija rubova u signalu (slično vrijedi za sliku), konvolucija signala i kernela za detekciju ruba će dati niske vrijednosti gdje nema rubova i signal nema velikih promjena u amplitudi u odnosu na dijelove gdje će konvolucija pokazati visoke vrijednosti ako postoji veća promjena u amplitudi signala.



Slika 2: Primjer signala nad kojim će se primijeniti konvolucija za detekciju rubova



Slika 3: Konvolucija signala sa slike iznad i kernela (-1, 0, 1) za detektiranje rubova

Na slici 2 prikazan je primjer signala koji naglo raste za 100 jedinica amplitude na vremenskom uzorku 125. Na slici 3 prikazana je konvolucija tog signala s jednostavnim kernelom (-1, 0, 1) koji detektira rub (to jest skok). Kao rezultat, jedino vremenski uzorak 125 ima visoku vrijednost veću od 100, dok su svi ostali uzorci značajno manji, s vrijednostima ispod 10. Kernel u ovom slučaju funkcionira tako da za odabrani uzorak negira njegovu vrijednost i računa razliku između susjednih vrijednosti s lijeve i desne strane. Ako je razlika velika, rezultat konvolucije je visok, dok kod malih razlika rezultat ostaje nizak.

Obično je vremenska složenost operacije konvolucije $O(n^2)$ [6, 9], ali postoje algoritmi koji izračunavaju konvoluciju u $O(n * log(n))$. Ti se algoritmi baziraju

na teoremu konvolucije, definiran formulom:

$$a * b = DFT_{2n}^{-1}(DFT_{2n}(a) * DFT_{2n}(b))$$

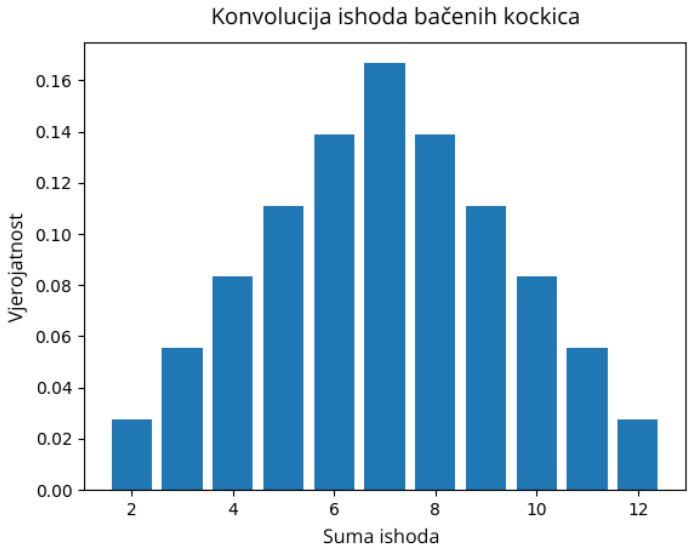
Teorem glasi da za dane dvije torke (sekvence) duljine n, koja je potencija baze 2, konvolucija tih torki jest izračun inverza diskretne Fourierove transformacije od produkta diskretne Fourierove transformacije obje torke [9]. Više o Fourierovim transformacijama u odjeljku o algoritmima temeljenim na brzoj Fourierovoj transformaciji.

2.1 Problem vjerojatnosti zbroja bačenih kocki

Ovaj odjeljak će pokazati klasični primjer upotrebe konvolucije u statistici. Radi se o problemu dodavanja dviju diskretnih nezavisnih slučajnih varijabli.

Slučajna varijabla jest neka vrijednost koja ovisi o slučajnom događaju. Za diskretnu varijablu to je neka vrijednost iz skupa mogućih ishoda gdje je zbroj njihovih vjerojatnosti da se dogode jednak 100% [10].

U ovom primjeru koriste se dvije pravilne kocke, svaka s 6 stranica, pri čemu svaka stranica ima jednaku vjerojatnost da padne, što znači da je distribucija uniformna. Neovisnost između kocki znači da ishod bacanja jedne kocke nema nikakav utjecaj na ishod bacanja druge kocke.



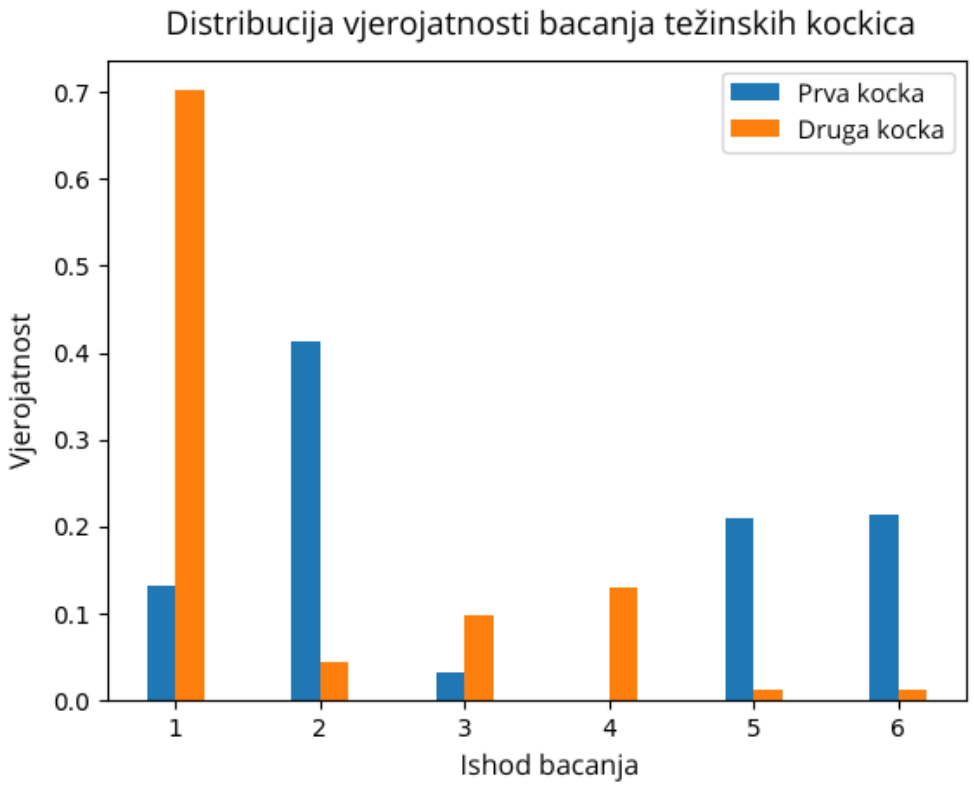
Slika 4: Distribucija vjerojatnosti za zbroj dva bacanja kockica

Ako se baci jedna standardna kocka sa šest stranica, vjerojatnost da padne broj 3 iznosi 1/6, budući da je svaka od šest stranica jednako vjerojatna. Isto vrijedi i za broj 4, s vjerojatnošću od 1/6. Kada se bace dvije kocke i njihovi brojevi zbroje, vjerojatnost dobivanja određenog zbroja može se izračunati analizom svih mogućih kombinacija rezultata. Ukupno postoji 6 x 6 = 36 mogućih kombinacija. Da bi se dobio zbroj 8, mogući parovi su: 2+6, 3+5, 4+4, 5+3 i 6+2. Budući da postoji 5 takvih kombinacija, vjerojatnost dobivanja osmice je 5/36. Slično

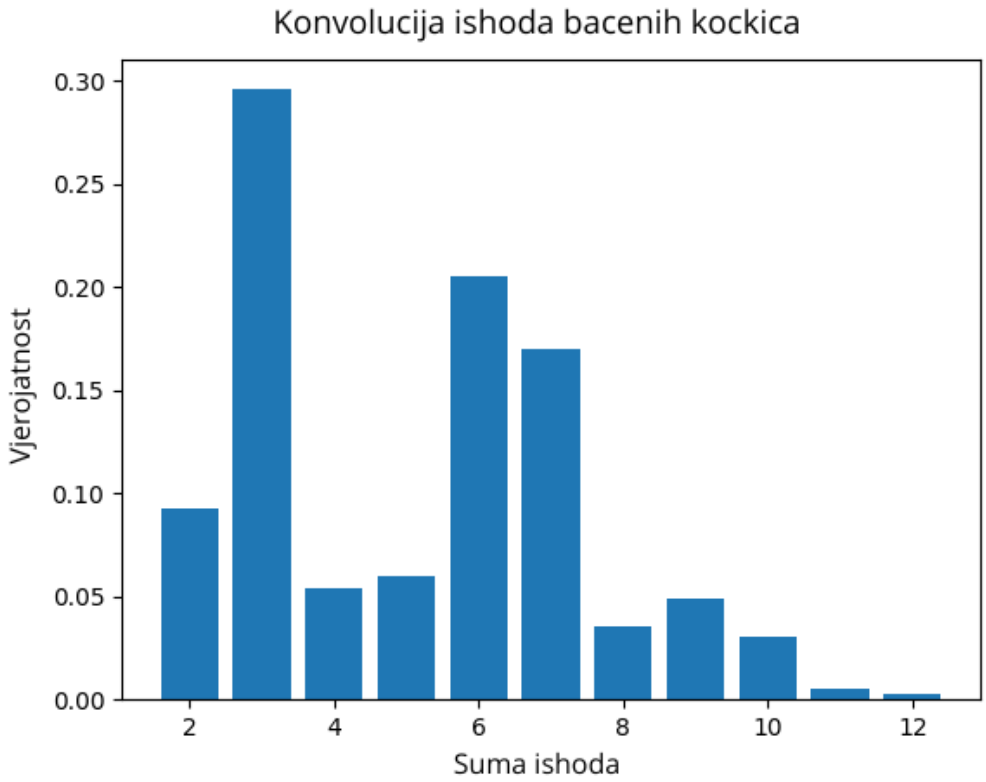
tome, za zbroj 5, moguće kombinacije su: 1+4, 2+3, 3+2 i 4+1, što daje ukupno 4 kombinacije, pa je vjerojatnost dobivanja petice 4/36.

Ovaj primjer postaje kompleksniji kada kocke nemaju uniformnu distribuciju, odnosno kada vjerojatnost ispadanja svakog broja na kockama nije jednaka. Naime, umjesto standardnih vjerojatnosti od 1/6 za svaki broj, vjerojatnosti su zadane slijedom: [0,132, 0,413, 0,032, 0,001, 0,209, 0,213] za prvu kocku i [0,702, 0,045, 0,099, 0,130, 0,012, 0,012] za drugu kocku.

Kako bi se riješio ovaj problem, koristi se konvolucija. Konvolucija dva niza vjerojatnosti daje niz koji predstavlja distribuciju zbrojeva svih mogućih ishoda dviju kocki. Rezultantni niz nakon primjene konvolucije ovih dvaju skupova vjerojatnosti je: [0,093, 0,296, 0,054, 0,06, 0,205, 0,17, 0,036, 0,049, 0,03, 0,005, 0,003]. Kako bi se dobila vjerojatnost zbroja pet, čita se četvrti element ovog niza (budući da prvi element predstavlja vjerojatnost za zbroj dva), što znači da je vjerojatnost dobivanja zbroja pet 0,06. Ako se analiziraju distribucije vjerojatnosti prikazane na slikama 5 i 6, može se primijetiti da konvolucija pokazuje visoku vrijednost već na zbroju tri. Razlog tome leži u visokim vjerojatnostima da prva kocka padne na jedinicu, a druga na dvojku. Sljedeći značajni zbrojevi su 6 i 7, što se može objasniti time da postoji mnogo kombinacija koje daju ove zbrojeve, čineći ih čestim ishodima. Međutim, od zbroja 8 nadalje, vjerojatnosti opadaju.



Slika 5: Distribucije vjerojatnosti za bacanje kockica



Slika 6: Distribucija vjerojatnosti za zbroj dva bacanja pravilnih nepravilnih kockica

Osmica se može dobiti kombinacijama 2+6, 3+5, 4+4, 5+3 i 6+2. S obzirom na to da druga kocka ima visoku vjerojatnost za dobivanje jedinice, te nešto manju za trojku i četvorku, zbroj 8 se može postići jedino korištenjem trojki ili četvorki, koje imaju vjerojatnost od oko 10%. Na prvoj kocki, da bi se dobio ovaj zbroj, potrebna je četvorka ili petica. Četvorka ima vrlo malu vjerojatnost (0,1%), dok petica ima veću (20%). Dakle, jedina realna kombinacija koja daje zbroj 8 je petica na prvoj kocki i trojka na drugoj. Slično objašnjenje vrijedi i za zbrojeve 11 i 12, budući da druga kocka gotovo nema šansu da dobije peticu ili šesticu, koji su potrebni za postizanje tih zbrojeva, što objašnjava gotovo nultu vjerojatnost za te ishode.

2.2 Klizni skalarni produkt

Skalarni produkt neka dva niza S_1 i S_2 duljine n iznosi:

$$produkt(S_1, S_2) = \sum_{i=0}^{n-1} S_1[i] * S_2[i]$$

Za konvoluciju diskretnog slučaja, dana dvaju niza S_1 i S_2, oba duljine n, x-ti element algoritma kliznog skalarnog produkta definiran je sljedećom formulom:

$$(S_1 * S_2)[x] = \sum_{i=0}^{n-1} S_1[i] * S_2[x-i]$$

$$x \in \{0, 1, 2, ..., 2n-2\}$$

gdje algoritam zanemaruje elemente kada S_1 ili S_2 izađu izvan raspona. Iz formule je jasno da algoritam ima vremensku složenost $O(n^2)$. Rezultat konvolucije biti će nova sekvenca duljine 2n - 1. Za konvoluciju mora izračunati svaki element, a računanje svakog elementa zahtjeva sumu koja prolazi ponovno po svim elementima, tako da je to O(n*n) to jest $O(n^2)$ (prvi n se dobije zbog broja output elemenata koji se moraju izračunat, a drugi n se dobije zbog sume za svaki od tih output elemenata). Ovo je raspisana verzija sume za svaki x:

$x = 0 : S_1[0] * S_2[0] + S_1[1] * S_2[-1] + S_1[2] * S_2[-2] + S_1[3] * S_2[-3] + S_1[4] * S_2[-4] + S_1[5] * S_2[-5]$
$x = 1 : S_1[0] * S_2[1] + S_1[1] * S_2[0] + S_1[2] * S_2[-1] + S_1[3] * S_2[-2] + S_1[4] * S_2[-3] + S_1[5] * S_2[-4]$
$x = 2 : S_1[0] * S_2[2] + S_1[1] * S_2[1] + S_1[2] * S_2[0] + S_1[3] * S_2[-1] + S_1[4] * S_2[-2] + S_1[5] * S_2[-3]$
$x = 3 : S_1[0] * S_2[3] + S_1[1] * S_2[2] + S_1[2] * S_2[1] + S_1[3] * S_2[0] + S_1[4] * S_2[-1] + S_1[5] * S_2[-2]$
$x = 4 : S_1[0] * S_2[4] + S_1[1] * S_2[3] + S_1[2] * S_2[2] + S_1[3] * S_2[1] + S_1[4] * S_2[0] + S_1[5] * S_2[-1]$
$x = 5 : S_1[0] * S_2[5] + S_1[1] * S_2[4] + S_1[2] * S_2[3] + S_1[3] * S_2[2] + S_1[4] * S_2[1] + S_1[5] * S_2[0]$
$x = 6 : S_1[0] * S_2[6] + S_1[1] * S_2[5] + S_1[2] * S_2[4] + S_1[3] * S_2[3] + S_1[4] * S_2[2] + S_1[5] * S_2[1]$
$x = 7 : S_1[0] * S_2[7] + S_1[1] * S_2[6] + S_1[2] * S_2[5] + S_1[3] * S_2[4] + S_1[4] * S_2[3] + S_1[5] * S_2[2]$
$x = 8 : S_1[0] * S_2[8] + S_1[1] * S_2[7] + S_1[2] * S_2[6] + S_1[3] * S_2[5] + S_1[4] * S_2[4] + S_1[5] * S_2[3]$
$x = 9 : S_1[0] * S_2[9] + S_1[1] * S_2[8] + S_1[2] * S_2[7] + S_1[3] * S_2[6] + S_1[4] * S_2[5] + S_1[5] * S_2[4]$
$x = 10 : S_1[0] * S_2[10] + S_1[1] * S_2[9] + S_1[2] * S_2[8] + S_1[3] * S_2[7] + S_1[4] * S_2[6] + S_1[5] * S_2[5]$

Ovo je raspisana verzija sume za svaki x ako se ignoriraju indeksi koji izlaze iz raspona (uvijek će izaći iz raspona u drugom nizu, prvo će izaći na indeksima manjima od nule, a zatim na indeksima većim od duljine samog niza):

$x = 0 : S_1[0] * S_2[0]$
$x = 1 : S_1[0] * S_2[1] + S_1[1] * S_2[0]$
$x = 2 : S_1[0] * S_2[2] + S_1[1] * S_2[1] + S_1[2] * S_2[0]$
$x = 3 : S_1[0] * S_2[3] + S_1[1] * S_2[2] + S_1[2] * S_2[1] + S_1[3] * S_2[0]$
$x = 4 : S_1[0] * S_2[4] + S_1[1] * S_2[3] + S_1[2] * S_2[2] + S_1[3] * S_2[1] + S_1[4] * S_2[0]$
$x = 5 : S_1[0] * S_2[5] + S_1[1] * S_2[4] + S_1[2] * S_2[3] + S_1[3] * S_2[2] + S_1[4] * S_2[1] + S_1[5] * S_2[0]$
$x = 6 : S_1[1] * S_2[5] + S_1[2] * S_2[4] + S_1[3] * S_2[3] + S_1[4] * S_2[2] + S_1[5] * S_2[1]$
$x = 7 : S_1[2] * S_2[5] + S_1[3] * S_2[4] + S_1[4] * S_2[3] + S_1[5] * S_2[2]$
$x = 8 : S_1[3] * S_2[5] + S_1[4] * S_2[4] + S_1[5] * S_2[3]$
$x = 9 : S_1[4] * S_2[5] + S_1[5] * S_2[4]$
$x = 10 : S_1[5] * S_2[5]$

Ideja za nazivom kliznog skalarnog produkta je ta da se u konvoluciji računa puno skalarnih produkata u nekim rasponima indeksa koji "klize" indeks po indeks, to jest mijenjaju početni ili završni indeks računanja produkta za svaki novi izlaz.

Važno je napomenuti da svaki od izlaza za x = 0 pa sve do x = 10, skalarni produkti ne ovise jedan o drugom, to jest rezultat jednog nije potreban da bi se izračunao rezultat drugog, što je ključno za kasniju paralelizaciju kod implementacije.

2.3 Algoritmi temeljeni na brzoj Fourierovoj transformaciji

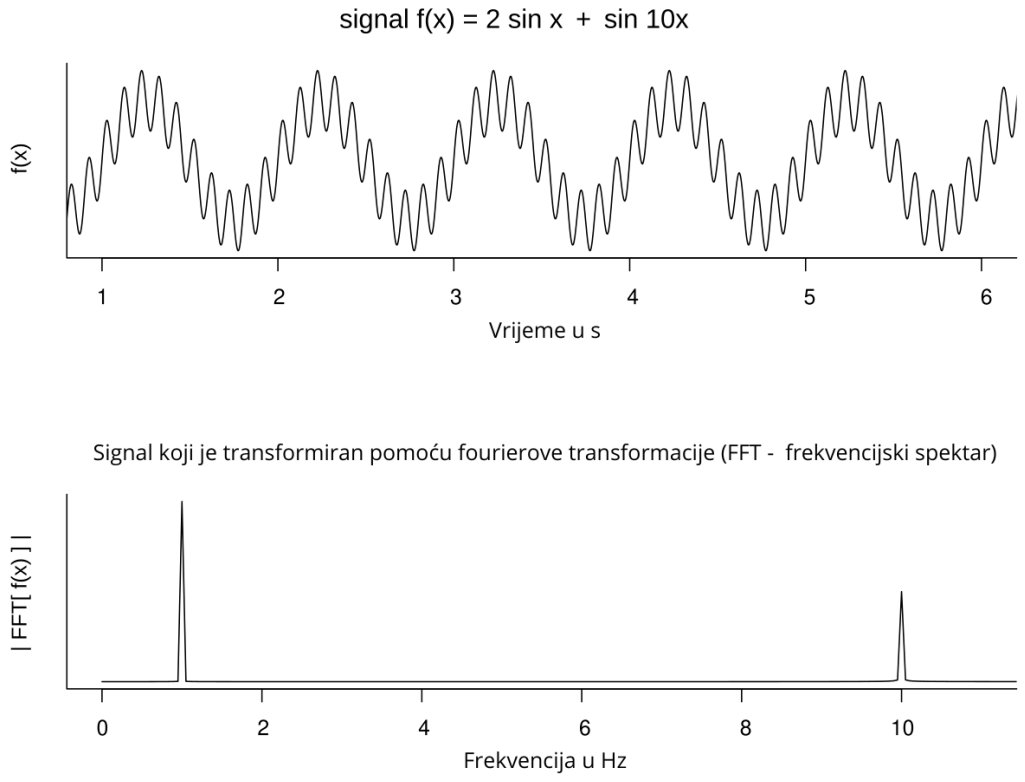
Algoritmi za izračun konvolucije koji se temelje na brzoj Fourierovoj transformaciji (FFT) imaju vremensku složenost od $O(n*log(n))$ u odnosu na standardne $O(n^2)$ [9] što je velika optimizacija pogotovo nad većim podacima.

Cormen kaže [9] da algoritmi temeljeni na FFT-u izvode sljedeće 3 operacije kako bi postigli konvoluciju:

1. Izračunaju diskretnu Fourierovu transformaciju (DFT) obje sekvence (pretvorite njihovo vrijeme u frekvenciju)
2. Pomnože nizove po elementima
3. Izračunaju inverznu diskretnu Fourierovu transformaciju (inverz DFT) sekvence (pretvoriti frekvenciju natrag u vrijeme)

Matematički, za dane torke a i b, njihova konvolucija je zapisana na sljedeći način:

$$a * b = DFT_{2n}^{-1}(DFT_{2n}(a) * DFT_{2n}(b))$$



Slika 7: Transformacija signala iz vremenske domene u frekvencijsku domenu [11]

Najčešća uporaba Fourierove transformacije, a time i FFT-a, je u obradi signala. Signal je dan u vremenskoj domeni kao funkcija koja preslikava vrijeme na amplitudu. Fourierova analiza izražava signal kao težinski zbroj fazno pomaknutih sinusoida različitih frekvencija. Težine i faze povezane s frekvencijama karakteriziraju signal u frekvencijskom području. [4, 9, 12, 13]. Drugim riječima, za dan signal koji je zapisan u vremenskoj domeni, to jest svaki uzorak predstavlja pojedinu amplitudu u toj vremenskoj jedinici, Fourierova analiza, to jest Fourierova transformacija (u ovom slučaju diskretna) će dati signal u drugom obliku gdje se može iščitati koja frekvencija ima koju težinu u zbroju s ostatkom frekvencija da bi se dobio originalni signal.

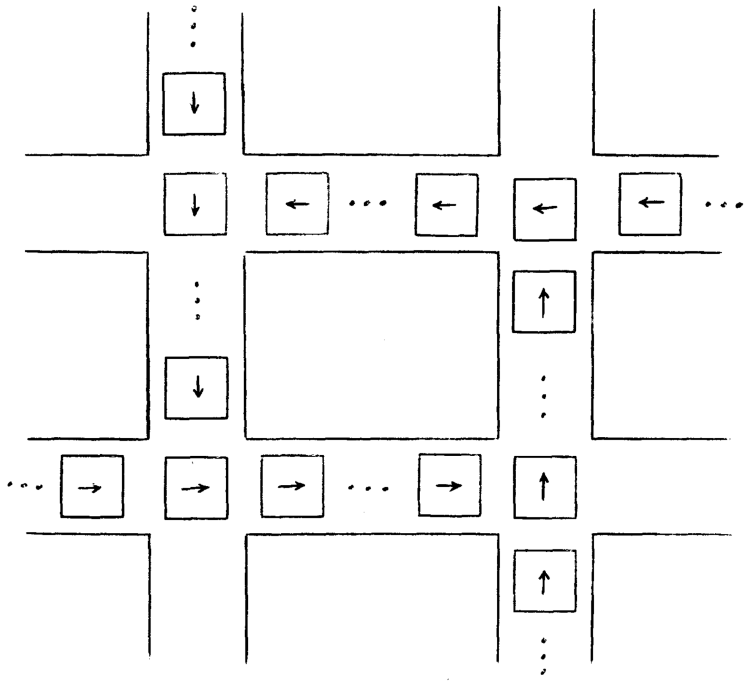
Funkcije koje nisu periodične (ali čije je područje ispod krivulje konačno) mogu se izraziti kao integral sinusa i/ili kosinusa pomnoženih težinskom funkcijom. Važna je karakteristika da se funkcija, izražena Fourierovom transformacijom, može potpuno rekonstruirati putem inverznog procesa, bez gubitka informacija [13].

3 Paralelno računanje

Paralelno izračunavanje je vrsta izračunavanja gdje se više izračunavanja ili zadataka računa istovremeno. Paralelno računanje obično je zgodno kada se problem može rastaviti na manje nezavisne probleme koji se zatim paralelno rješavaju. Ne smije se miješati s pojmom istodobnog izvršavanja, što je kada se proces koji izvršava neki zadatak prebacuje između zadataka, a da ih nužno ne dovrši [2].

Kada je problem sam po sebi trivijalan za paralelno izvođenje, naziva se neugodno paralelan ili prirodno paralelan. Takvi problemi su primjerice manipulacije na slici poput pomicanja, skaliranja, rotiranja i izrezivanja, prikazivanje Mandelbrotovog skupa i Monte Carlo metode [1].

Kao što je primijećeno, samo zato što algoritam ima suboptimalnu vremensku složenost, on može biti bolja opcija na paralelnom sustavu. Zato programeri moraju osmisliti algoritme za već riješene probleme koji će raditi na paralelnom sustavu, čime će se poboljšati performanse. U takvom inženjerstvu pojavljuju se neki izazovi. Zastoj je jedan takav izazov gdje više procesa čeka na resurse zaključane od strane drugog procesa koji je također u zastoju te u konačnici se nikad ne izvrše. Zastoj je posebno problematičan jer, jednom kada se dogodi, nema jednostavnog načina da se riješi bez vanjskog uplitanja ili prekida nekih procesa, što može dovesti do gubitka podataka ili drugih problema u sustavu [11]. Prepoznavanje, prevencija i oporavak od deadlocka su ključni aspekti dizajna sustava koji se bave višeprocesnim dizajnom programa [14].



Slika 8: Primjer zastoja u prometu, svi čekaju da se oslobodi raskrižje [14]

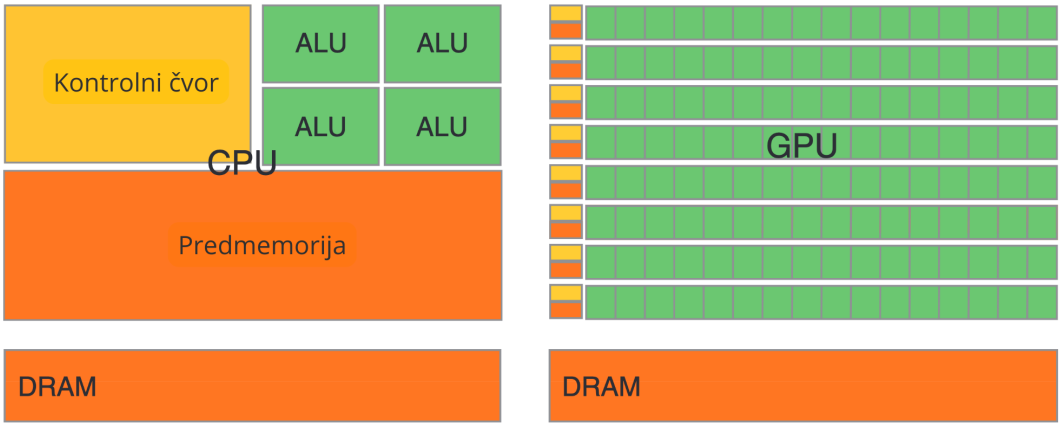
Još jedan uobičajeni problem je takozvani uvjet utrke je jedan od njih, gdje ishod programa ovisi o vremenu događaja koji se ne mogu kontrolirati, kao što

je povećanje vrijednosti u zajedničkoj memoriji budući da operacije "čitanje", "povećanje", "pisanje" rade na istoj memoriji višestruko procesi mogu čitati istu vrijednost, povećavati je i pisati isti broj, dakle ne povećavajući je više puta. CUDA je, na primjer, riješila ovaj problem uvođenjem nečega što se zove "atomic add" što čini sve ove tri operacije kao jednu [15].

3.1 Paralelno Procesiranje na jednom računalu

CPU (centralna procesorska jedinica) je primarna komponenta računala odgovorna za izvršavanje instrukcija iz programa. Zbog njihove ključne uloge, oni se stalno istražuju i poboljšavaju u dizajnu omogućujući veću procesorsku snagu. Njihovi dizajni primjenjuju neke od tehnika kao što su multithreading, višejezgrenost, korištenje SIMD-a (jedna instrukcija, više podataka) i druge.

Grafička procesorska jedinica (GPU) je komponenta koja se obično koristi u kontekstu obrade digitalne slike i ubrzane računalne grafike. Međutim, GPU se ne koriste samo u grafičke svrhe. Također postoje GPU-ovi opće namjene, kao što im ime govori, koji koriste GPU hardver za zadatke opće namjene (obično se nazivaju GPGPU) [16]. Dobavljači takvog hardvera omogućuju razvojnim programerima da iskoriste računalnu snagu svojih GPU-ova putem API-ja koje su oni dostavili. U NVIDIA-inom slučaju to je CUDA API koji proširuje C jezik [17], a u AMD-ovom slučaju to je AMD APP SDK koji radi kroz OpenCL [18].



Slika 9: Razlike u filozofiji dizajna CPU-a i GPU-a [19]

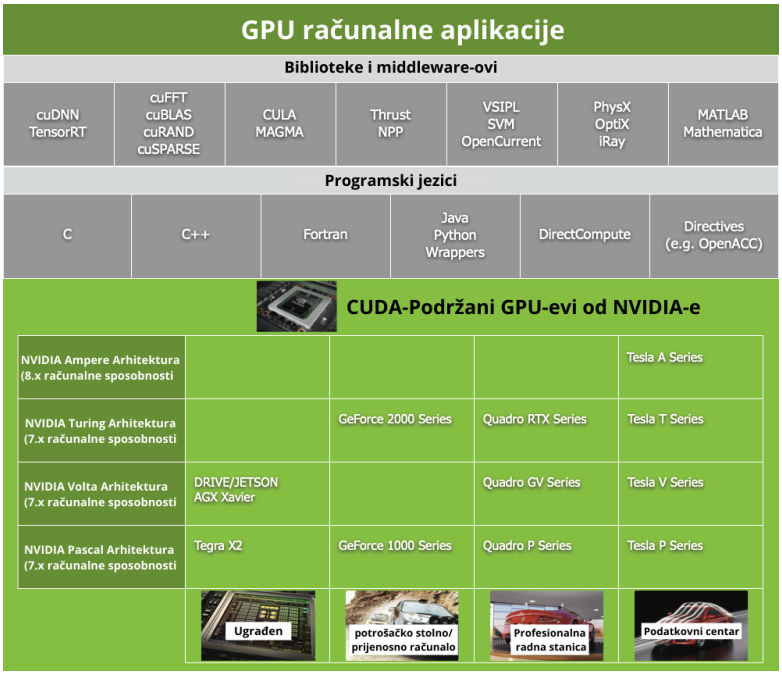
CPU i GPU imaju temeljno različite arhitekture koje su optimizirane za različite vrste zadataka, a time i za različite pristupe paralelnom procesiranju. CPU (Central Processing Unit) je dizajniran za generalno računalstvo i sposoban je izvršavati niz različitih zadataka, ali s relativno ograničenim brojem paralelnih operacija u jednom trenutku. CPU-ovi su često opremljeni s nekoliko jezgri (obično između 4 i 16), od kojih svaka može izvršavati vlastiti slijed instrukcija, što im omogućuje istovremeno obavljanje različitih zadataka. Svaka CPU jezgra ima složenu strukturu s velikim brojem jedinica za predviđanje, grananje i keširanje (engl. caching), što je optimizirano za serijsko izvršavanje instrukcija [19].

S druge strane, GPU (Graphics Processing Unit) je posebno dizajniran za visoko paralelne zadatke. GPU-ovi imaju tisuće jednostavnijih jezgri koje su optimizirane za istovremeno izvršavanje iste operacije na velikim skupovima podataka. Ova arhitektura je izuzetno učinkovita za zadatke koji zahtijevaju masivni paralelizam, kao što su grafički renderiranje, simulacije i strojno učenje. GPU jezgre su manje složene od CPU jezgri i dizajnirane su za obradu velikih blokova podataka u paralelnim nitima, što omogućuje brzo izvođenje ponavljajućih operacija na velikim količinama podataka. Glavna razlika u arhitekturi između CPU-a i GPU-a je u balansu između kontrolne logike i jedinične logike za obradu podataka [19].

3.1.1 CUDA API

CUDA (Compute Unified Device Architecture) API je platforma za paralelno programiranje koju je razvila NVIDIA. Dizajnirana je kako bi omogućila programerima korištenje grafičkih procesnih jedinica (GPU) za opće svrhe (GPGPU) računalnih zadataka. CUDA API omogućuje programerima da pišu kod u jezicima poput C, C++, Fortran i Python, što se zatim može izvršavati na NVIDIA GPU-ima. Ovaj pristup omogućuje izravnu kontrolu nad resursima GPU-a [20].

Programeri koriste CUDA API za definiranje kernel funkcija koje se paralelno izvršavaju na GPU-u, dok se sinkronizacija i upravljanje memorijom odvijaju putem specifičnih CUDA funkcija [21].



Slika 10: Primjene CUDA-e, podržani jezici i biblioteke [15]

3.1.2 AMD APP SDK

AMD APP SDK (Accelerated Parallel Processing Software Development Kit) je alat razvijen od strane AMD-a koji omogućuje programerima stvaranje aplikacija

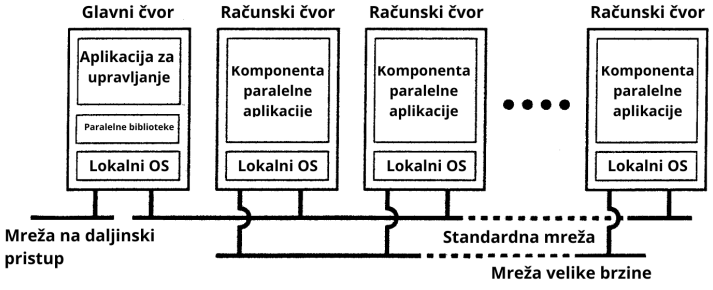
koje koriste prednosti paralelnog procesiranja na AMD GPU-ima i CPU-ima. Ovaj SDK podržava OpenCL standard, što znači da omogućuje pisanje prijenosnog koda koji se može izvršavati na različitim platformama, uključujući procesore različitih proizvođača, a ne samo AMD-ove [22].

Kod razvijen sa AMD APP SDK-a prvo se piše u OpenCL jeziku, gdje se kernel definira za izvršavanje na GPU-u ili CPU-u. Zatim se OpenCL kernel kompajlira i pokreće iz host programa, koristeći OpenCL API funkcije za upravljanje kontekstom, redovima zadataka i resursima na dostupnim računalnim jedinicama [22].

3.2 Paralelno Procesiranje na više računala

Paralelno procesiranje se ne mora ograničiti na jedan sustav ili uređaj poput GPU-a ili multicore CPU-a, već se može postići s više računala s prijenosom poruka ili mrežnih klastera [1]. Moguće je proširiti paralelizam na više računala koja rade zajedno kao mreža. Ovaj pristup, poznat kao distribuirano računanje, omogućuje obradu velikih količina podataka i složenih računalnih zadataka koristeći snagu više uređaja, često povezanih putem brze mreže [23]. Distribuirano računanje zahtjeva podjelu zadatka na manje dijelove koji se distribuiraju između više računala te svako računalo obrađuje svoj dio zadatka paralelno s ostalim računalima u mreži. Međutim, iako ovaj model može značajno smanjiti vrijeme obrade velikih zadataka, zahtijeva učinkovitu koordinaciju i komunikaciju između svih sustava u mreži [24].

Jedan od ključnih izazova u implementaciji distribuiranog računalstva je komunikacijska latencija između sustava. Brzina mreže i kašnjenja u prijenosu podataka mogu postati usko grlo koje utječe na ukupne performanse sustava. Stoga je važno optimizirati protokole i algoritme za prijenos podataka kako bi se smanjio utjecaj latencije [24].



Slika 11: Primjer računalnog sustava organiziranog u klaster [24]

Klasičan primjer distribuiranog računalstva je klaster računalstvo, gdje skupina povezanih računala radi zajedno kako bi obavila zajednički zadatak. Klasteri se obično koriste u znanstvenim istraživanjima, financijskim modelima i simulacijama, gdje je potrebna velika procesorska snaga [25]. U klasterima, sustavi često koriste tehnologije kao što su MPI (Message Passing Interface) za koordinaciju i razmjenu podataka između čvorova [26]. Grid računalstvo je još jedan oblik distribuiranog računanja, ali se razlikuje od klaster računalstva po tome što koristi distribuirane geografski na velikim udaljenostima. Grid računala omogućuju

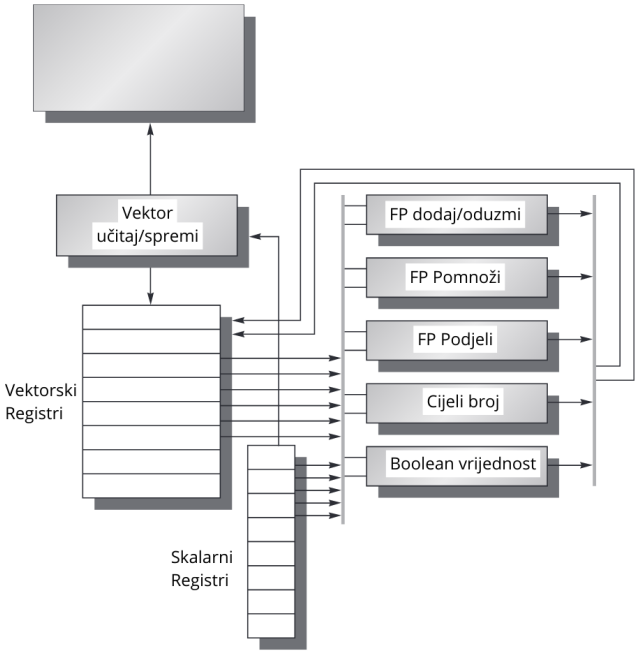
pristup resursima kao što su procesorsko vrijeme, memorija ili pohrana podataka s više organizacija ili institucija, stvarajući virtualni superračunalni sustav [23]. Ovaj model je posebno koristan u projektima poput analize podataka s velikih eksperimenata ili globalnih simulacija.

Cloud računalstvo, najnoviji oblik distribuiranog računalstva, koristi slične principe, ali se fokusira na pružanje računalnih resursa kao usluge putem interneta. Korisnici mogu dinamički alocirati i de-alocirati resurse prema potrebi, što čini cloud računalstvo izuzetno fleksibilnim rješenjem za paralelno procesiranje [27].

3.3 SIMD Arhitektura

SIMD (Single Instruction, Multiple Data) je arhitektura u paralelnom računalstvu koja, kako sam naziv govori, omogućuje izvršavanje jedne instrukcije na više podatkovnih točaka istovremeno. Ovaj pristup razlikuje se od tradicionalnog SISD (Single Instruction, Single Data) modela, gdje svaka instrukcija djeluje na jednu podatkovnu točku. SIMD je posebno učinkovit u zadacima koji zahtijevaju istu operaciju na velikim skupovima podataka, kao što su obrada vektora, multimedijalne aplikacije i znanstveni izračuni [28] [29].

U SIMD arhitekturi, jedna instrukcija se prenosi na više procesnih elemenata, od kojih svaki obrađuje različite dijelove podataka. Ova konfiguracija je idealna za radne zadatke koji su inherentno paralelni, kao što su matrične operacije, obrada slika i određene vrste simulacija. Na primjer, u zadatku množenja matrica, SIMD procesor može istovremeno množiti odgovarajuće elemente dviju matrica, značajno smanjujući vrijeme obrade.



Slika 12: Arhitektura VMIPS vektorskog procesora [30]

Izvršavanje SIMD-a može se vizualizirati kao vektor podatkovnih elemenata

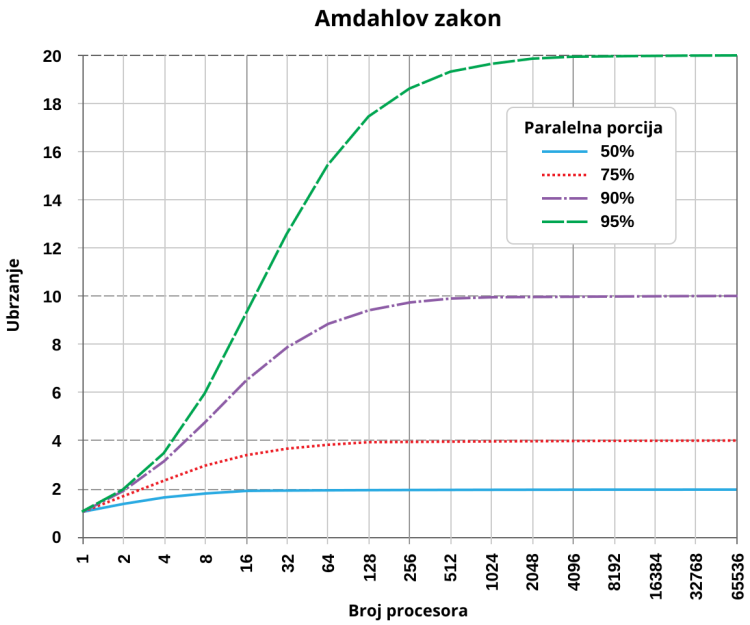
koji se obrađuju zajedno. Moderni procesori postižu SIMD paralelizam putem vektorskih registara, koji drže više podatkovnih elemenata. Instrukcije se zatim primjenjuju na ove vektorske registre, obrađujući sve podatkovne elemente u registru odjednom [30].

Moderni CPU-ovi i GPU-ovi intenzivno koriste SIMD kako bi poboljšali performanse, posebno u multimedijalnim, gaming i znanstvenim aplikacijama. Intelovi SSE (Streaming SIMD Extensions) i AVX (Advanced Vector Extensions) su poznati primjeri SIMD skupova instrukcija u CPU-ima. Ovi dodaci omogućuju procesorima da u jednom taktu obrade više podatkovnih točaka, što značajno poboljšava performanse u zadacima koji se mogu vektorizirati [31].

GPU-ovi su, po svojoj prirodi, dizajnirani oko SIMD modela. Sadrže tisuće jednostavnih, paralelnih procesnih jedinica sposobnih za izvršavanje iste instrukcije na velikom broju podatkovnih točaka. Treba imati na umu da SPMD (jedan program, više podataka) nije isto što i SIMD (jedna instrukcija, više podataka). U SPMD sustavu jedinice za paralelnu obradu izvršavaju isti program na više dijelova podataka. Međutim, ove procesorske jedinice ne moraju izvršavati istu instrukciju u isto vrijeme. U SIMD sustavu sve procesorske jedinice izvršavaju istu instrukciju u bilo kojem trenutku [19].

3.4 Amdahlov zakon

Samo zato što postoje ogromne količine resursa za paralelno izvođenje, to ne znači da će ubrzanje biti linearno korelirano. Sve ovisi o postotku paralelnog dijela zadatka koji se izračunava.



Slika 13: Količina ubrzanja u paralelnom izvođenju u odnosu na broj procesora i paralelni dio programa [32]

Na primjer. Ako je 50% zadatka paralelne prirode i kada se koristi dovoljno

resursa, tih 50% bit će dovršeno gotovo odmah, a preostalo vrijeme potrošit će se na taj sekvencijalni dio zadatka. Dakle, ubrzanje će biti dvostruko. Ako je paralelni dio 33,33%, ubrzanje je trostruko, i tako dalje. Opća formula za to glasi $(1-p)^{-1}$ gdje je p postotak paralelnog dijela. Ovo je također poznato kao Amdahlov zakon [33].

Kao što se može vidjeti sa slike iznad, za program koji ima paralelnu porciju od 50%, dakle 50% programa je sklono paralelnom izvršavanju, on će dobiti na maksimalnoj brzini na otprilike 16 procesa, a ona će biti dvostruko veća od sekvencijalnog izvršavanja. Za program of 75% paralelne porcije on će dobiti na maksimalnoj brzini na otprilike 256 procesa, a ona će biti četverostruka u odnosu na sekvencijalno izvršavanje. Svakako, ni u kojem slučaju ne može n procesa postići više od n puta brže izvršavanje.

3.5 Primjer dodavanja dva niza

Kako bi se prikazalo paralelno računanje na djelu, postoji klasični problem zbrajanja brojeva od 1 do n. Standardni pristup je zbrajanje jedan i dva, zatim zbrajanje tri prethodnom zbroju, zatim zbrajanje 4 itd. Paralelni pristup bio bi podijeliti cijelu duljinu od 1 do n u dva jednaka niza (ili približno jednaka) a zatim svaki podijeliti na još dva i tako dalje sve dok na listovima te strukture nalik stablu ne budu samo pojedinačni elementi. Zapisano to dvoje izgleda ovako:

$$Sekvencijalni\_zbroj = (((((((1+2)+3)+4)+5)+6)+7)+8)$$

$$Paralelni\_zbroj = (((1+2)+(3+4))+((5+6)+(7+8)))$$

U sekvencijalnom pristupu (1 + 2) uzima jednu jedinicu vremena, +3 uzima drugu jedinicu vremena +4 uzima još jednu i tako dalje. Ukupno je potrebno 7 jedinica vremena. Problem je u tome što pri paralelnom izvođenju također treba 7 otkucaja budući da sve prethodne operacije moraju biti izračunate prije izračunavanja, na primjer +7. Dakle, ni u jednom trenutku više od jednog procesa nije aktivno i ne obavlja posao.

Međutim, paralelni pristup unatoč uzimanju isto 7 jedinica vremena kada se izvodi na jednom procesu, potrebno je manje jedinica vremena kada je dostupno više resursa. U prvoj jedinica vremena će se računati (1 + 2), (3 + 4), (5 + 6) i (7 + 8) istovremeno jer su svi neovisni jedni o drugima. U drugoj jedinici vremena će se izračunati prva dva zbroja (3 + 7) i druga dva zbroja (11 + 15), u trećoj jedinici vremena će se izračunati konačni zbroj (10 + 26) = 36.

Paralelni dio algoritma je približno $1 - \frac{log_2(n)}{n}$ gdje je n broj elemenata koje treba zbrojiti. Kako broj elemenata raste, paralelni dio gotovo postaje 100%, kao što je prikazano ovim ograničenjem:

$$\lim_{n\to\infty} 1 - \frac{log_2(n)}{n} = 1$$

3.6 Primjer paralelnog sortiranja

Bitonic sort, kojeg je predložio Batcher [34] je efikasan algoritam za paralelno sortiranje, posebno pogodan za implementaciju na paralelnim računalnim arhitekturama poput GPU-ova i paralelnih procesora. To je algoritam s razmjenom podataka, što znači da radi usporedbom i zamjenom elemenata na određeni način kako bi dobio konačno sortirani niz.

Bitonic sort se temelji na konceptu bitonskih nizova. Bitonski niz je niz koji prvo monoton raste, a zatim monoton opada, ili obrnuto. Na primjer, niz [1, 3, 7, 8, 6, 4, 2] je bitonski jer prvo raste (1, 3, 7, 8), a zatim opada (8, 6, 4, 2).

Cilj bitonic sorta je pretvoriti proizvoljan niz u bitonski, a zatim ga sortirati korištenjem bitonskog spajanja (engl. bitonic merge). Proces sortira niz tako da koristi više razina usporedbi i zamjena, gdje se na svakoj razini postiže djelomično sortiranje elemenata niza.

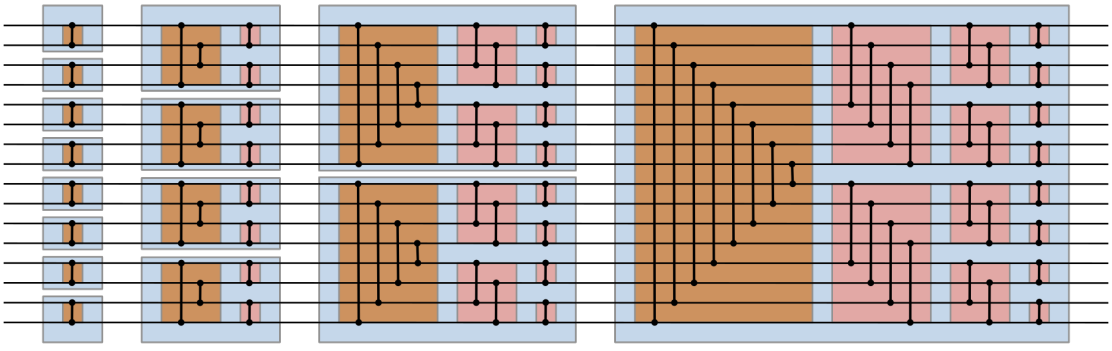
Prvi korak u bitonic sortiranju je formiranje bitonskog niza iz neuređenog niza. To se postiže korištenjem podnizova koji su sortirani u suprotnim smjerovima. Primjerice, ako je dan niz [a, b, c, d], najprije se mogu sortirati prva dva elementa u rastućem poretku i druga dva u opadajućem poretku, čime se dobiva bitonski niz [min(a,b), max(a,b), max(c,d), min(c,d)].

Nakon što je bitonski niz formiran, bitonic sort koristi operaciju spajanja kako bi stvorio potpuno sortirani niz. Spajanje započinje uspoređivanjem elemenata u prvoj polovici bitonskog niza s odgovarajućim elementima iz druge polovice i njihovim zamjenama ukoliko su izvan redoslijeda.

Na primjer, ako se sortiraju dva podniza [a, b] i [c, d] gdje je [a, b] rastući niz, a [c, d] opadajući niz, uspoređuju se a i c, te b i d, kako bi se stvorio novi rastući niz [min(a, c), min(b, d), max(a, c), max(b, d)].

Proces spajanja se ponavlja rekurzivno, pri čemu se svaka polovica niza ponovo dijeli i sortiraju se nove bitonske sekvence dok cijeli niz ne bude potpuno sortiran. Ovaj rekurzivni proces omogućava visoku razinu paralelizacije jer se svaka usporedba može izvoditi nezavisno od drugih.

Slika ispod prikazuje cijeli taj postupak vizualno.



Slika 14: Bitonično sortiranje prikazuje usporedbe za svaku obradu [35]

Budući da se nizovi mogu dijeliti na manje podnizove i svaka usporedba može biti neovisna, algoritam se može izvršavati na velikom broju procesora bez potrebe za komunikacijom između procesora tijekom većine koraka [34].

Efikasnost bitonic sorta, mjerena brojem potrebnih usporedbi i zamjena, je $O(n * log^2(n))$, što ga čini manje efikasnim u odnosu na najbolje sekvencijalne algoritme poput Quicksorta koji imaju složenost $O(n * log(n))$. Međutim, kad se govori o paralelnom izvršavanju, bitonic sort je daleko bolji pošto je velik dio programa sklono paralelnom izvršavanju.

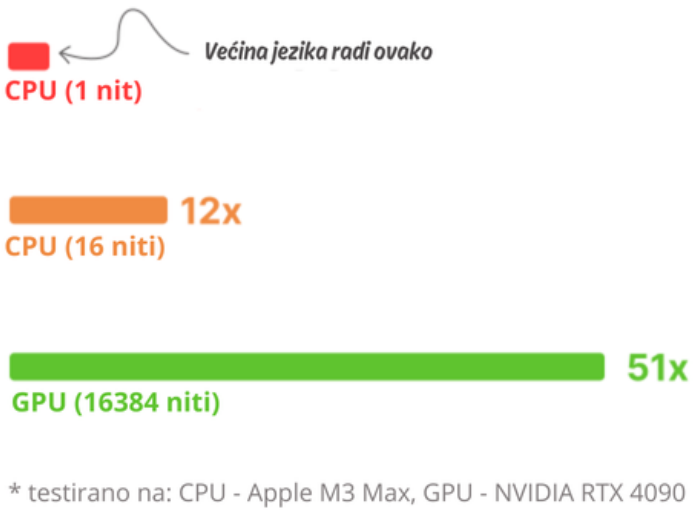
Jedan nedostatak ovog sortiranja je što zahtjeva da ulazni skup bude veličine 2^n zbog svoje prirode dijeljenja podataka na dva djela [34].

3.7 Bend programski jezik

Bend je funkcijski programski jezik visoke razine koji je orijentiran paralelnom računanju, a razvila ga je tvrtka Higher Order Company. Također su razvili runtime nad kojim se Bend izvršava, nazvan HVM. I Bend i HVM izgrađeni su u programskom jeziku Rust. Oba su projekta otvorili za zajednicu na platformi GitHub.

Ovako to funkcionira. Programer prvo napiše program u Bend jeziku, Bend zatim pretvara taj kod u HVM kod, a zatim HVM procesira HVM kod kako bi ga pokrenuo u Rust, C ili CUDA. HVM je iza zavjesa zapravo evaluator za interakcijski kombinator, koji je univerzalni model za distribuirana izračunavanja, kako navodi Lafont [36].

Pokretanje koda može se izvršiti putem jednog od runtime-a. Može ga pokrenuti Rust runtime koji je standardno sekvencijalno izvršavanje jedne dretve, može ga pokrenuti C runtime koji omogućuje višedretvenost CPU-a i može imati neke paralelne prednosti ili ga može pokrenuti CUDA runtime koji će koristiti samu CUDA-u za korištenje resursa GPU-a, maksimizirajući paralelne koristi.



Slika 15: Različite prednosti ubrzanja na Bendu [37]

Obično bi se paralelni algoritam prvo implementirao sekvencijalno kako bi se smanjili svi trivijalni bugovi, budući da je rješavanje bugova u paralelnom algo-

ritmu puno teže od onih na standardnim sekvencijalnim tokovima [1]. Međutim, način na koji se programira u Bendu uklanja potrebu za tim, budući da te probleme rješava sam Bend.

3.7.1 Struktura Benda

Jezik je izgrađen imajući na umu jednostavnost. Kod je čist i jednostavan, smanjujući kognitivno opterećenje programera. Također ima automatizirano upravljanje memorijom čime se smanjuje vjerojatnost curenja memorije ili "prelijevanja međuspremnika". Njegov interni model je dizajniran za izbjegavanje uvjeta utrke i zastoja, kao što je spomenuto, koji su obično prisutni u paralelnom programiranju. Bend koristi automatsko prepoznavanje obrazaca u kodu koji su pogodni za paralelizaciju. Na primjer, kada se u kodu pojavi račvanje rekurzije na dva nova poziva funkcija, Bend automatski prepoznaje mogućnost paralelizacije i raspoređuje rad na više jezgri procesora ili druge resurse. Ovo omogućuje programerima da postignu gotovo linearno ubrzanje s povećanjem broja jezgri (ili drugih resursa).

Bend je funkcijski jezik koji ne dopušta nikakvu globalnu ili zajedničku memoriju. To je ključno pravilo koje omogućuje paralelno izvršavanje bez potrebe za eksplicitnim paralelnim napomenama poput stvaranja kernela, zaključavanja, muteksa ili atomika. Čisti funkcijski jezici su prirodno pogodni za paralelizaciju.

Iako postoji funkcijska i imperativna sintaksa, imperativni koncepti nisu baš podržani za vrijeme pisanja ovog rada. na primjer, petlje ne postoje (jedino u obliku rekurzije, i konstrukata "fold" i "bend").

3.7.2 Uvod u Bend

Ova podsekcija će dati kratki uvod u sam jezik i mora se naglasiti kako sve što je trenutno pisano je sklono promjenama pošto je jezik u fazi razvoja (naravno, i ova rečenica je sklona promjenama pošto je u fazi je razvoja za vrijeme pisanja, što ne mora biti slučaj za godinu, dvije, ili deset).

Bend je još uvijek u izradi, tako da je trenutno svaki Bend program svojevrsna Bend skripta. Svaki Bend program ima .bend ekstenziju, a skripta mora sadržavati "main" funkciju koja je polazišna točka programa. Program se pokreće kroz naredbu u terminalu, a kroz argumente se mogu prosljeđivati vrijednost koje program može koristiti. Ovaj mehanizam je jedini mehanizam za uvođenje dinamike u program (dobivanje različitog rezultata, bez mijenjanja samog koda). U jeziku ne postoji naredba ispisa kao u većini jezika, tako da je jedina povratna informacija programa ona vrijednost koja vrati funkcija "main".

Većinom, program je podijeljen na definiranje tipova i funkcija. Tipovi su nešto kao struct-ovi u C++ jeziku. Ovdje tipovi imaju "verzije" tipova. Na primjer: Tip je "Geometrijski oblik", a verzije su mu "Kocka" i "Kugla". Kod definiranja tipova i verzija, također se definiraju i konstruktori za verzije, ali samo kao nazivi vrijednosti koje će posjedovati ta verzija tipa kad se instancira. Vrijednost može također biti pokazivač na neki drugi tip pa se tako mogu dobiti vezane liste i stabla kao rekurzivni tip.

```
# Tip Lista, sa verzijama Cons (element) i Nil (kraj)
```

```
# Verzija Nil nema konstruktor
# Verzija Cons ima konstruktor head kao vrijednost i ~tail
# kao pokazivač na novu Listu
type List:
  Nil
  Cons { head, ~tail }
```

Konstrukt "match" je konstrukt koji radi nad tipovima. Match djeluje kao "if" za tipove. Kroz match mogu se staviti uvjeti, ukoliko je dani tip određene verzije, izvrši neki kod. Ovo je primjer gdje program vrati 0 ako je lista prazna, ili prvi element ako nije prazna:

```
def main:
  lista = [1, 2, 3]
  match lista:
    case List/Cons:
      return lista.head
    case List/Nil:
      return 0
```

Bend, kako je naglašeno, podržava imperativnu i funkcijsku sintaksu. Poželjno je pisati u jednoj sintaksi na razini funkcije. Funkcija bi se trebala pisati isključivo imperativno ili isključivo funkcijsko, ne miješano.

Funkcije u Bendu su jednostavne po definiciji. Postoji naziv funkcije, argumenti i povratna vrijednost (nezgrapno je ukoliko se ne vraća vrijednost). Argumenti ne mogu biti opcionalni, pa je pojam definiranja "pomoćne funkcije" zgodan ukoliko je potrebna neka dodatna vrijednost kod rekurzija, najčešće za postavljanje inicijalnih ili baznih slučaja rekurzije. Ukoliko se koristi funkcijska sintaksa, dobivaju se nove pogodnosti za definiciju funkcija. Bend voli raditi sa račvanjima i detektirati slučajeve (engl. case) pa tako kada se koristi funkcijska sintaksa, funkcije se mogu definirati takozvanim "pattern-matching"-om, gdje se u definiciji definira kojeg je tipa određeni argument i onda se izvrši ta određena definicija funkcije. Primjer uklanjanja prvog elementa liste sa "patter-matching"-om:

```
# List/pop_front:
#   Removes and discards the first item of list l.
#   The new list is returned, or [] if the list is empty.
# Example:
#   [1, 2, 3] -> [2, 3]
List/pop_front (List/Cons x xs) = xs
List/pop_front (List/Nil) = List/Nil
```

Funkcija List/pop_front uzima listu kao ulaz i uklanja prvi element tako da provjerava je li lista kao tip verzija elementa ili verzija prazne liste. Ako je verzija elementa, vrati rep što je druga lista, a ako je prazna lista, vrati praznu listu pošto se ništa ne može ukloniti.

Za iterirati rekurzivne tipove koriste se "fold", a za generirati "bend" konstrukti. Fold je sličan rekurziji sa "match"-om. Primjer folda nad stablom:

```
def sum(tree):
  fold tree:
    case Tree/Node:
      return tree.left + tree.right
    case Tree/Leaf:
      return tree.value

def main:
  tree = ![![!1, !2],![!3, !4]]  # Skraćena sintaksa stabla
  return sum(tree)  # Rezultat: 10
```

Bend će automatski izvoditi ovu rekurzivnu prirodu konstrukcije paralelno. Važno je napomenuti da su će Bend izvršavati paralelno sve što može, dakle ako je algoritam paralelne prirode, on će se paralelno izvršiti. Ako funkcija poziva dvije funkcije i one su neovisne, one će se izvršavati paralelno (naravno, ako postoje dostupni resursi). Upravo je to ovdje primjer. tree.left će nastaviti sljedeći poziv folda kao što će i tree.right nastaviti na svoj sljedeći poziv folda, pa će se oni pri paralelnom izvršavanju izvoditi paralelno. Slično je i sa "bend" konstruktom:

```
def main():
  bend x = 0:
    when x < 3:
      tree = ![fork(x + 1), fork(x + 1)]
    else:
      tree = !7
  return tree
```

Svaki fork će stvoriti svoju granu koja će se izvršavati paralelno pošto jedna grana stvara 2 nove nezavisne grane.

3.7.3 Priprema okruženja u Bendu

Kako bi netko programirao u Bendu i pisao kod koji se pokreće paralelno, mora učiniti nekoliko koraka:

1. Instalirati Rust da se dobije pristup Cargo menadžeru paketa [38] (za MacOS i Linux: curl https://sh.rustup.rs -sSf — sh)
2. Instalirati Bend programski jezik preko Cargo menadžera: cargo install hvm@2.0.19
3. Instalirati HVM preko Carko menadžera: cargo install bend-lang@0.2.36
4. Instalirati GCC kompajler [39]
5. (opcionalno) Podesiti CUDA okruženje ukoliko je prisutan "CUDA podržan GPU" za paralelno izvršavanje preko grafičke kartice

6. (opcionalno) Instalirati neki uređivač koda (IDE) kao na primjer: VSCode [40]

Ovo je priprema okruženja koja je potrebna za krenuti programirati u Bendu, a pisana je 27. kolovoza 2024. godine. Moguće je da će ova lista nakon nekog vremena biti polovična ili potpuno nevažeća no to su neki vanjski faktori na koje se ne može utjecati.

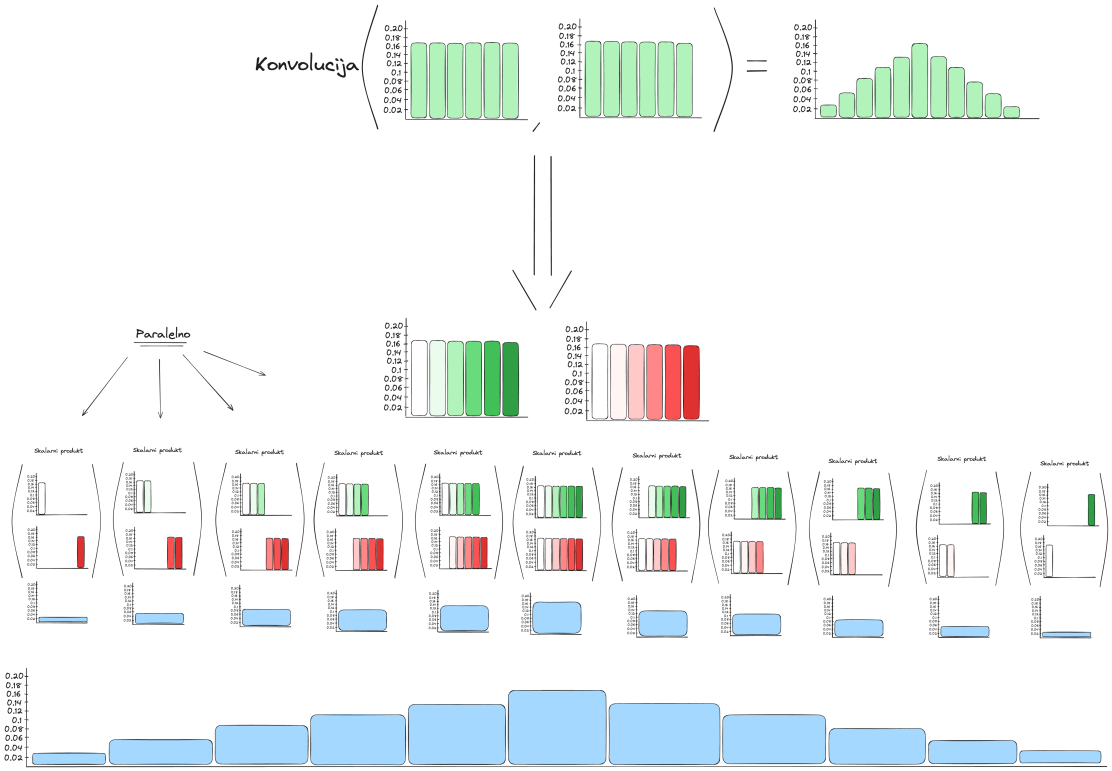
4 Implementacija

Ovo poglavlje će objasniti implementaciju i proces računanja konvolucije dvaju nizova u programskom jeziku Bend.

Prvi korak je implementirati konvoluciju na bilo koji način. Dakle, ignoriranje paralelizma za sada. Pošto se ovdje radi s operacijom diskretnog slučaja, implementirat će se algoritam kliznog skalarnog produkta. Samo kao podsjetnik, konvolucija će uzeti skalarni produkt prvog i zadnjeg elementa, zatim prva dva i posljednja dva, zatim prva tri i zadnja tri i tako dalje.

Nakon tog naivnog pristupa, tehnika redukcije stabla koristit će se kao druga metoda s ciljem povećanja paralelne porcije programa za učinkovitije izvršavanje u paralelnom okruženju, a onda će se te dvije strategije zasebno evaluirati u sljedećem poglavlju.

U oba slučaja, radi se o programu koji za ulaz prima dva niza veličine n i za izlaz vraća jedan niz veličine 2n - 1 (pošto konvolucija dvaju nizova veličine n daje niz veličine 2n - 1).

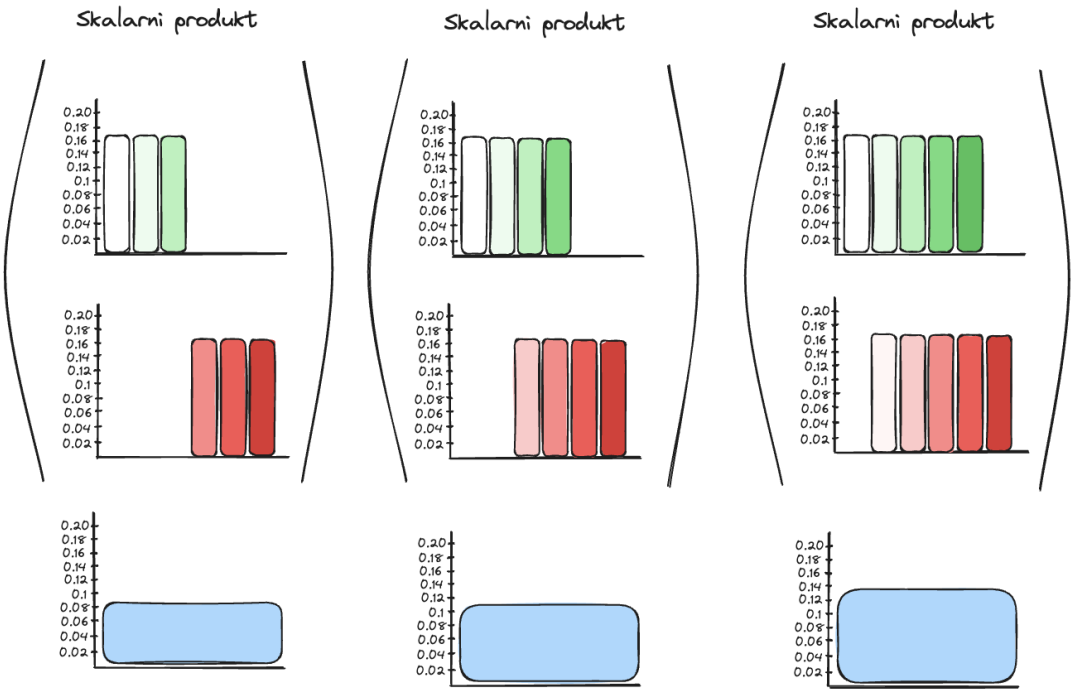


Slika 16: Vizualizirana implementacija algoritma

Slika iznad pokazuje skicu naivnog pristupa algoritma i uvid u podrijetlo ideje za paralelizam kod tehnike redukcije stabla. Cijela skica bazira se na vizualizaciji u kontekstu konvolucije dviju diskretnih slučajnih varijabli bacanja kockica. Na vrhu skice nalaze se ulazi i izlazi algoritma. Ulaz programa jesu dva niza vrijednosti, u ovom primjeru to su (1/6, 1/6, 1/6, 1/6, 1/6, 1/6) i (1/6, 1/6, 1/6, 1/6, 1/6, 1/6), to jest (0.166, 0.166, 0.166, 0.166, 0.166, 0.166) i (0.166, 0.166, 0.166, 0.166, 0.166, 0.166) prikazani u obliku grafa distribucije bacanja kockica, a izlaz jest

rezultirajući niza konvolucije, u ovom primjeru to je niz (0.028, 0.056, 0.083, 0.111, 0.139, 0.167, 0.139, 0.111, 0.083, 0.056, 0.028).

Ispod ulaza i izlaza algoritma nalazi se strelica koja pokazuje na postupak dobivanja tog izlaza. Prvi (lijevi) ulazni niz je sada obojen u gradijente zelene boje, a drugi (desni) u gradijente crvene boje za distinkciju nizova po bojama i distinkciju elemenata po jačini boje što je nužno za razumijevanje sljedećeg koraka.



Slika 17: Vizualizirana implementacija algoritma - zumirana

Slika iznad je zumirana na porciju nekoliko skalarnih produkta iz originalne slike cijele implementacije radi čitljivosti potrebne za idući korak. Sljedeći korak opisuje kako se dobivaju izlazni elementi (prikazani plavom bojom) jedan po jedan:

- **Prvi element** izlaznog niza iznosi rezultat skalarnog produkta **prvog** elementa prvog niza (najsvjetliji element u zelenom nizu) i **zadnjeg** elementa drugog niza (najtamniji element u crvenom nizu).
- **Drugi element** izlaznog niza iznosi rezultat skalarnog produkta **prva dva** elementa prvog niza (najsvjetliji i drugi najsvjetliji elementi u zelenom nizu) i **zadnjih dvaju** elementa drugog niza (najtamniji i drugi najtamniji elementi u crvenom nizu).
- **Treći element** izlaznog niza iznosi rezultat skalarnog produkta **prva tri** elementa prvog niza (najsvjetliji, drugi najsvjetliji i treći najsvjetliji elementi u zelenom nizu) i **zadnjih triju** elementa drugog niza (najtamniji, drugi najtamniji i treći najtamniji elementi u crvenom nizu).

  ...

- **Šesti element** izlaznog niza iznosi rezultat skalarnog produkta **prvih šest** elementa prvog niza i **zadnjih šest** elementa drugog niza. U ovom slučaju to su svi elementi prvog i svi elementi drugog niza, tako da se samo uzima skalarni produkt prvog i drugog niza (što su ujedno i ulazni nizovi programa). Formulacija je ista kao i: Šesti element izlaznog niza iznosi rezultat skalarnog produkta **zadnjih šest** elementa prvog niza i **prvih šest** elementa drugog niza.

- **Sedmi element** izlaznog niza iznosi rezultat skalarnog produkta **zadnjih pet** elementa prvog niza i **prvih pet** elementa drugog niza.

- **Osmi element** izlaznog niza iznosi rezultat skalarnog produkta **zadnjih četiri** elementa prvog niza i **prvih četiri** elementa drugog niza.

  ...

- **Jedanaesti element** izlaznog niza iznosi rezultat skalarnog produkta **zadnjeg** elementa prvog niza (najtamniji element prvog niza) i **prvog** elementa drugog niza (najsvjetliji element drugog niza).

Ovdje se također prikazuje kako su tih jedanaest izračuna nezavisni jedan o drugom, te se mogu paralelno računati.

Za vrijeme implementacije navedenih pristupa, Bend nije podržavao način, to jest nije efikasno podržavao učitavanja nizova u program. Jedini unos dinamike u program je bio kroz CLI (command line interface) argumente koji su bili limitirani na argument po argument, a unutar samog programa se morao vaditi argument po argument i to je bilo zahtjevno za odraditi, tako da je program morao samom sebi izgenerirati dva niza određene veličine s kojima bi mogao baratati. Kako to vrijeme izvršavanja za generiranje nizova nije od značajnog utjecaja na cjelokupni izračun, radi jednostavnosti program generira ulazne podatke. Pravi ulazni podatak je k, koji označava eksponencijalni stupanj koliko elemenata se mora generirati. Za ulaz k=3, program računa konvoluciju od dva niza veličine $2^3 = 8$ elemenata, za k=6 računa konvoluciju dva niza veličine $2^6 = 64$ elemenata, a za k=10 računa konvoluciju dva niza veličine $2^10 = 1024$ elemenata.

4.1 Naivni pristup

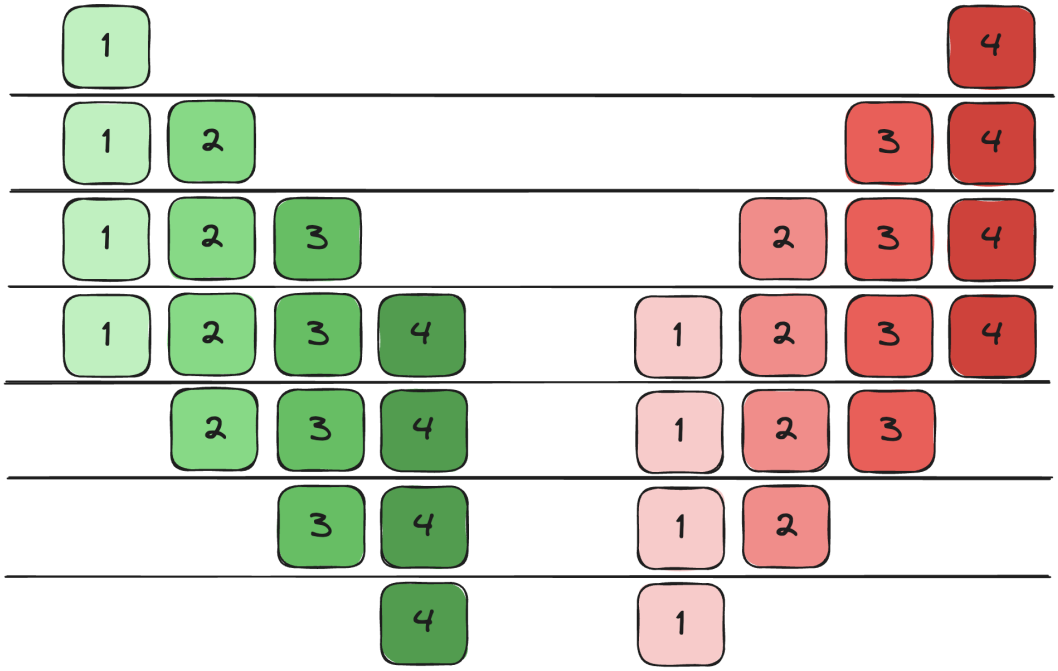
Prvo opažanje kod naivnog pristupa bilo je uočavanje uzorka indeksa za koje su rađeni skalarni produkti. Fokusirajući se samo na indekse prvog niza (obojenih u zeleno na skici sa početka poglavlja), pojavljuje se sljedeći obrazac:

$$[1], [1, 2], ..., [1, 2, 3, ..., n], [2, 3, ..., n], ..., [n-2, n-1, n], [n-1, n], [n]$$

Fokusiranjem na indekse drugog niza (obojenih u crveno na skici sa početka poglavlja) pojavljuje se isti uzorak, ali obrnut. [n], [n, n - 1] ... Imajući tu informaciju, prvi zadatak je dobiti sve moguće glave i repove danog niza pošto njihova konkatenacija elemenata daje upravo taj uzorak.

Glave niza su svi mogući podnizovi koji počinju u prvom elementu i slijedno prate elemente originalnog niza. Dakle [1, 2, 3] jest glava, dok [2, 3, 4] nije glava

pošto ne počinje brojem jedan, [1, 3, 2] nije glava pošto drugi element nije broj 2, [1, 2, 4] nije glava pošto treći element nije 3. Isto tako vrijedi i za repove, samo što nemaju fiksiran prvi, već zadnji element. Dakle [n - 3, n - 2, n - 1, n] je validni rep, dok [n - 3, n - 1, n] nije. Nakon što se dobiju indeksi svih glava (sortiranih od onih sa najmanje elemenata, do onih sa najviše) dobiva se niz [1], [1, 2], [1, 2, 3], ..., [1, 2, 3,..., n]. Nakon što se dobiju indeksi svih repova (sortiranih na isti način) dobiva se niz [1, 2, 3, ..., n], [2, 3,..., n], ..., [n - 2, n - 1, n], [n - 1, n], [n]. Zadnje što preostaje je spojiti ta dva niza u jedan. Primijetiti da se element [1, 2, 3, ..., n] pojavljuje u oba slučaja, tako da se prilikom spajanja mora ukloniti jedan od tih dvaju elementa (nije bitno koji pošto su identični). Ovaj finalni niz će se nazvati trokut niza.



Slika 18: Vizualizacija indeksa potrebnih za skalarne produkte konvolucije

Slika iznad vizualizira dobivanje tih trokuta za nizove od 4 elementa. Svaki redak označava izlazni element, dok stupac označava koji elementi su uključeni u skalarne produkte. Za prvi red, računa se skalarni produkt prvog elementa prvog niza i zadnjeg elementa zadnjeg niza, za drugi element izlaznog niza računa se skalarni produkt prvih dvaju elemenata prvog niza i zadnjih dvaju drugog niza, i tako dalje. Ako se fokusira na zeleni niz i njegovu progresiju kroz "vrijeme", to jest kroz redove, može se vidjeti kako broj elemenata raste ([1], [1, 2], [1, 2, 3], [1, 2, 3, 4]) i kad se obuhvate svi elementi, broj elemenata počinje opadat ([2, 3, 4], [3, 4], [4]). Upravo to su glave niza (prva 4 retka) i repovi niza (sljedećih 3 retka). Ista je situacija kod crvenog niza, samo što je slika zrcaljena po y osi, što znači da je redoslijed obrnut. Prvo zadnji elementi (repovi), pa onda prvi (glave).

Brojčano raspisani, uzme li se trokut prvog niza, i trokut drugog niza te obrne redoslijed tog drugog trokuta, dobit će se sljedeći nizovi:

$$[1], [1,2], [1,2,3], ..., [1,2,3,..., n-1], [1,2,3,..., n], ..., [n-1, n], [n]$$

$$[n], [n-1, n], [n-2, n-1, n], ..., [2,3,..., n], [1,2,3,..., n], ..., [1,2], [1]$$

Budući da je niz u Bendu predstavljen kao (head, tails), dobivanje trokuta nije tako teško. Taj algoritam je podijeljen na dobivanje svih mogućih glava i svih mogućih repova. Repovi su jednostavni, ali za izračunavanje svih glava koristi se trik s obrnutim nizom. Budući da sama konvolucija treba izračunati trokute za 2 nezavisna niza, a svaki trokut izračunava vrhove i repove neovisno, moguće je x4 ubrzanje kada se radi paralelno.

Sljedeći zadatak je uzeti te nizove trokuta T_1 i T_2 (T_2 je obrnut) i izračunati sve skalarne produkte: $dot\_product(T_1[i], T_2[i])$ a kada se svi skalarni produkti izračunaju i kombiniraju u jedan niz, to je rezultat konvolucije. Pseudokod algoritma izgleda ovako:

```
function konvolucija(ulaz_a, ulaz_b)
{
    trokut_a := stvori_trokut(ulaz_a)
    trokut_b := obrni_redosljed_elemenata(stvori_trokut(ulaz_b))

    izlaz := []
    for( i = 0; i < veličina(trokut_a); i++) {
        produkt := skalarni_produkt(trokut_a[i], trokut_b[i])
        izlaz.add_element(produkt)
    }
    return izlaz
}

function skalarni_produkt(niz_a, niz_b)
{
    izlaz := 0
    for( i = 0; i < veličina(niz_a); i++) {
        izlaz += niz_a[i] * niz_b[i]
    }
    return izlaz
}
```

No, kako u Bendu nisu podržane petlje, implementacija se mora izvršiti rekurzijama. Prilagođen pseudokod koji se implementirao u Bendu je:

```
// Glavni dio programa
funkcija main(k)
{
    ulaz_a = stvori_niz_duljine(2^k)
    ulaz_b = stvori_niz_duljine(2^k)
```

```
    return konvolucija(ulaz_a, ulaz_b)
}

// Naivni izračun konvolucije
funkcija konvolucija(ulaz_a, ulaz_b)
{
    trokut_a := stvori_trokut(ulaz_a)
    trokut_b := stvori_trokut(ulaz_b)

    return primjeni_funkciju(
        skalarni_produkt,
        trokut_a,
        obrni_redosljed_elemenata(trokut_b),
    )
}

// Funkcija koja računa skalarni produkt dvaju nizova
funkcija skalarni_produkt(niz_a, niz_b)
{
    množenje := funkcija (a, b) => { return a * b }
    umnošci := primjeni_funkciju(množenje, niz_a, niz_b)

    return suma(umnošci)
}

// Funkcija koja primjenjuje danu funkciju nad elementima niza a i b
// u parovima. funk(a[0], b[0]), funk(a[1], b[1]), ...
funkcija primjeni_funkciju(funk, niz_a, niz_b)
{
    return Lista({
        glava: funk(niz_a.glava(), niz_b.glava()),
        rep: zip_with(funk, niz_a.rep(), niz_b.rep()),
    })
}
```

4.2 Metoda redukcije stabla

Kako se naivni pristup bazira na izračun takozvanih "trokuta", analizom se došlo do informacije da algoritam većinu svog vremena troši nakon što se izračunaju trokuti, dakle kad je vrijeme za računanje skalarnih produkata. Pošto se računa puno skalarnih produkata, došlo se do ideje da se paralelizira taj dio programa.

Ovdje se koristi poznata strategija organiziranja podataka u obliku stabla tako da se svaka grana izvodi zasebno. Ova tehnika se češće klasificira kao tehnika dekompozicije podataka gdje se podaci dijele na manje dijelove, od kojih svaki

računa sličnu operaciju. Postoji specifičan slučaj dekompozicije podataka gdje se svaki element izlaza može izračunati neovisno i naziva se dekompozicija podataka na temelju particioniranja izlaznih podataka, i to je upravo ono što se ovdje koristi. Općenitije, kada se program razvija na paralelan način, postoji nekoliko tehnika dekompozicije za rastavljanje izračuna na manje neovisne zadatke koji se mogu izvoditi paralelno. Osim tehnike dekompozicije podataka, postoji i rekurzivna dekompozicija (ili funkcionalna particija [1]), istraživačka dekompozicija i spekulativna dekompozicija [2], ali ne odgovara svaka tehnika svakom problemu.

Dakle, nakon što se izračunaju trokuti niza (na isti način kao i u naivnom pristupu) primjenjuje se tehnika reduciranja niza u jednu vrijednost kroz stablastu strukturu. Najčešća redukcija niza jest suma niza. Suma niza kao ulaz uzima niz vrijednosti i za izlaz vrati jednu vrijednost. Suma uzima element po element i zbraja ih. Ovo je sekvencijalna vrsta redukcije gdje se funkcija zbrajanja primjenjuje na akumulator (trenutni zbroj do određenog elementa) i sljedeći element. Pseudokod takve redukcije izgleda ovako:

```
// Glavni dio programa
funkcija main()
{
    lista := [1, 2, 3, 4]
    zbrajanje := funkcija (a, b) => { return a + b }

    return redukcija_liste(zbrajanje, 0, lista)  // Output: 10
}

// Imperativni
funkcija redukcija_liste(funk, akumulator, lista)
{
    res := akumulator
    foreach (lista as element) {
        res = funk(res, element)
    }
    return res
}

// Funkcijski (prvi način)
funkcija redukcija_liste(funk, akumulator, lista)
{
    if (lista.prazna()) {
        return akumulator
    }

    return redukcija_liste(
        funk,
        funk(akumulator, lista.glava()),
        lista.rep()
```

```
    )
}

// Funkcijski (drugi način)
funkcija redukcija_liste(funk, inicijalna_vrijednost, lista)
{
    if (lista.prazna()) {
        return inicijalna_vrijednost
    }

    return funk(
        redukcija_liste(funk, inicijalna_vrijednost, lista.rep()),
        lista.glava()
    )
}
```

Kao što se može vidjeti, to je obična for petlja u jezicima koji podržavaju for petlju. Stablasta vrsta redukcije je primjena funkcije nad rezultatom lijevog stabla i rezultatom desnog stabla, gdje su rezultati dobiveni daljnjom rekurzivnom redukcijom. Pseudokod izgleda ovako:

```
// Glavni dio programa
funkcija main()
{
    stablo := {
              o
      Left  /   \  Right
          o       o
         / \     / \
        1   2   3   4
    }
    zbrajanje := funkcija (a, b) => { return a + b }

    return redukcija_stabla(zbrajanje, stablo)  // Output: 10
}

funkcija redukcija_stabla(funk, stablo)
{
    if (stablo.je_list()) {
        return stablo.dohvati_vrijednost()
    }

    return funk(
        redukcija_stabla(funk, stablo.lijevo()),
        redukcija_stabla(funk, stablo.desno()),
```

```
    )
}
```

Ovo je pristup redukcije pomoću stabla. Kod paralelnog izvršavanja suma ima vremensku složenost od O(log(n)), umjesto O(n).

Prvi zadatak je isti. Dobiti trokute obaju nizova. Drugi zadatak je pretvoriti prvi trokut niz u strukturu nalik stablu gdje listovi sadrže vrijednosti (elementi trokuta). Pošto Bend ima automatske sposobnosti paralelnog izvršavanja, kod ovakve rekurzije prethodnog pseudokoda, kod će se automatski paralelizirati. Na ovaj način, Bend će automatski dodijeliti resurse za paralelno izvođenje za svaku granu. Budući da je stablo binarno, ono će dodijeliti 2 resursa, zatim 4 resursa i tako dalje, dok se ne dosegne dubina listova (ili preostane bez resursa, što će staviti grane na listu čekanja). Posljednji zadatak je pretvoriti stablo natrag u niz kako bi se dobila konvolucija kao prikaz u nizu, a ne u stablu, što samo iziskuje da se vrijednosti listova stabla spreme u niz od najljevijeg do najdesnijeg lista.

Sekcija programa s trokutima ima x4 paralelno ubrzanje, a sekcija sa skalarnim umnoškom trebala bi imati ubrzanje na temelju broja dostupnih paralelnih procesa (u teoriji gotovo 100% paralelni dio). Kako se trebaju izračunati 2 trokuta niza, a svaki trokut zahtjeva da se izračunaju glave i repovi, to su 4 nezavisne operacije koje se moraju izračunat (glave prvog niza, repovi prvog niza, glave drugog niza i repovi drugog niza), pa je zato ubrzanje x4. Skalarni produkti se računaju sa skoro maksimalnom iskoristivošću resursa pa je ubrzanje u teoriji jednako broju dostupnih resursa.

5 Evaluacija

Ovo poglavlje daje procjenu rezultata implementacije, kako naivnog pristupa tako i pristupa koji se temelji na redukciji stabla. Evaluacija se provodi nizom izvršavanja programa sa različitim ulazima gdje se Bendova ugrađena statistika izvođenja programa koristi za metrike kao što su vrijeme, operacije u sekundi itd. Konačno, u poglavlju će se usporediti i raspravljati o rezultatima oba pristupa i utvrditi prednosti i ograničenja svakog.

5.1 Metodologija

Za procjenu izvedbe predloženih metoda u poglavlju 4 korištene su sljedeće metrike:

- Vrijeme izvršenja: Ukupno vrijeme potrebno za dovršetak operacija konvolucije u paralelnim i sekvencijalnim izvođenjima.
- Ubrzanje: Omjer vremena izvršenja sekvencijalne implementacije u odnosu na paralelnu implementaciju, demonstrirajući učinkovitost paralelizacije.

Algoritam se procjenjuje na ulazu od 2 niza cijelih brojeva veličine n. Kako je već objašnjenog u 4. poglavlju, same nizove generira program, ali to ne utječe na ukupnu izvedbu. Testovi se izvode nad veličinama ulaza $2^6, 2^7, 2^8, 2^9, 2^{10}, 2^{11}, 2^{12}$ i 2^{13}. Dakle, ulaz programa je varijabla k, koja se prosljeđuje kao parametar prilikom pokretanja Bend skripte. Varijabla k označava nad kojim veličinama niza će program računati konvoluciju. Npr, za k=4, program će računati konvoluciju dva niza od 2^4 elemenata. Formula po kojoj se određuje veličina dvaju nizova nad kojima se vrši operacija jest 2^k.

Evaluacija se odvija na MacBooku M2 Pro (16 inča) s 12 jezgri, koji u VSCode-u pokreće Bend skriptu za izračun konvolucije koristeći Bendov CLI za izvođenje i profiliranje testova.

5.2 Rezultati

Tablica u nastavku prikazuje rezultate evaluacije naivnog pristupa i pristupa redukcije stabla nad testiranim veličinama unosa:

	Naivni pristup			Pristup redukcije stabla		
Veličina unosa	Sekvencijalno vrijeme	Paralelno vrijeme	Ubrzanje	Sekvencijalno vrijeme	Paralelno vrijeme	Ubrzanje
64	0,01 s	0,01 s	x1,00	0,02 s	0,01 s	x2,00
128	0,03 s	0,02 s	x1,50	0,05 s	0,02 s	x2,5
256	0.12s	0.06s	x2.00	0.18s	0.09s	x2.00
512	0.46s	0.23s	x2.00	0.72s	0.36s	x2.00
1024	1.80s	0.90s	x2.00	2.83s	1.42s	x1.99
2048	7.47s	4.16s	x1.80	12.44s	-	-
4096	29.57s	-	-	-	-	-
8192	-	-	-	-	-	-

Ćelije označene sa "-" nikada nisu završile rad. Vodoravna linija nakon 4 testa je samo vizualne prirode, ne znači ništa. Ubrzanje se izračunava dijeljenjem sekvencijalnog vremena s paralelnim vremenom.

U ovom odjeljku bit će razmotreno nekoliko ključnih zapažanja: razlike u pristupima, teorijska vremenska složenost, ubrzanje kod oba pristupa, pad ubrzanja pri posljednjoj evaluaciji te testovi koji se ne izvrše do kraja.

Na prvi pogled moglo bi se pretpostaviti da bi pristup temeljen na redukciji stabla bio brži prilikom paralelnih izvođenja, no rezultati su pokazali da je ovaj pristup zapravo nešto sporiji.

Analizom vremena potrebnog za obradu svake nove ulazne veličine, uočava se da s povećanjem ulaznih podataka za jednu jedinicu (to jest veličina za koju se računa za faktor 2), vrijeme izvođenja raste za faktor 4, što je u skladu s očekivanom vremenskom složenošću od $O(n^2)$.

Iako ubrzanja oba pristupa izgledaju neujednačeno, to nije slučaj. U prvim dvjema iteracijama, vrijeme izvođenja je toliko kratko da preciznost na dvije decimale nije dostatna za prikaz stvarnog ubrzanja. Stoga, čini se da nema ubrzanja kod prve veličine unosa u naivnom pristupu, no brojke bi lako mogle biti, primjerice, 0,01455 i 0,005, što bi rezultiralo ubrzanjem od x2,91, dok zaokruživanje daje rezultat od 0,1 u oba slučaja što bi prikazivalo ubrzanje od x1. Posljednji relevantan test naivnog pristupa prikazuje ubrzanje od x1,8, što je neuobičajeno te bi zahtijevalo dodatnu analizu.

Važno je napomenuti da, iako pristupu temeljenom na redukciji stabla treba više vremena zbog dodatnih koraka poput pretvaranja polja u stablo i obrnuto, ubrzanje ostaje nepromijenjeno. To ukazuje na to da je skalarni umnožak značajno ubrzao ovaj pristup, budući da sam proces pretvorbe popisa u stablo nije paraleliziran, što bi inače rezultiralo smanjenjem ubrzanja. Ovo sugerira da bi pristup temeljen na redukciji stabla mogao biti brže rješenje, no trenutno je ograničen uskim grlom koje predstavlja pretvorba polja u stablo.

Kao što je vidljivo, neke ćelije označene su s "-" što označava "nikad nije dovršeno". Ovakvo ponašanje nije očekivano. Čini se da Bend ne može evaluirati program koji traje dulje od jedne minute ili dolazi do greške prilikom rada s većim brojevima ili veličinama nizova, pošto ni jedan test koji je trajao dulje od minute nije okončao unatoč izvršavanju dulje od 30 minuta. Uzrok ovog problema trenutačno nije poznat.

6 Ograničenja

Iako ovaj rad predstavlja istraživanje algoritama paralelne konvolucije implementiranih u programskom jeziku Bend, važno je napomenuti ograničenja koja su mogla utjecati na rezultate. Ova ograničenja proizlaze iz različitih čimbenika, uključujući hardverska ograničenja, softverska ograničenja, metodološke izbore.

Programski jezik Bend, iako moćan i fleksibilan za paralelno računanje, relativno je nov i možda mu nedostaju značajke zrelosti i optimizacije koje se nalaze u poznatijim jezicima. Određene jezične značajke, poput upravljanja memorijom, jezgrene paralelne obrade i optimizacije prevoditelja, možda neće biti toliko rafinirane, što potencijalno dovodi do neoptimalne izvedbe u nekim scenarijima. Na primjer, neki testovi s većim zahtjevima nikada nisu završili izvršenje.

Zbog vremenskih ograničenja, algoritam paralelne konvolucije nije iscrpno optimiziran za izvedbu. Iako je implementacija u Bendu pokazala značajno ubrzanje i skalabilnost, vjerojatno bi daljnja optimizacija mogla dati bolje rezultate.

7 Zaključak

Ovaj diplomski rad istraživao je operaciju konvolucije, njezine teorijske osnove i praktične primjene, s posebnim naglaskom na diskretnu konvoluciju. Raspravljalo se o različitim pristupima konvoluciji računanja, uključujući naivne metode i metode bazirane na stablu, te su te metode implementirane u programskom jeziku Bend. Sve to imajući na umu paralelno računanje, ključni aspekt u modernom računanju za poboljšanje performansi.

Početna implementacija konvolucijskog algoritma korištenjem jednostavnog pristupa pokazala je osnovnu funkcionalnost i pružila mjerilo za daljnja poboljšanja. Ova metoda, iako jednostavna, nije bila optimizirana za paralelno izvođenje i pokazala je ograničenja u učinkovitom rukovanju velikim veličinama podataka.

Kako bi se riješile neučinkovitosti uočene u jednostavnoj metodi, implementiran je pristup redukcije temeljen na stablu. Ova je metoda iskoristila mogućnosti paralelne obrade modernog hardvera, s ciljem smanjenja vremena izvršenja pri radu na paralelnom sustavu.

Programski jezik Bend, dizajniran imajući na umu paralelizam, olakšao je učinkovitu implementaciju ovih algoritama za konvoluciju. Apstrahirajući mnoge složenosti povezane s paralelnim programiranjem, Bend je olakšao fokusiranje na optimizaciju samog algoritma. Funkcionalna priroda jezika i nedostatak zajedničke memorije smanjili su rizik od uobičajenih problema s paralelnim programiranjem, kao što su uvjeti utrke i zastoji.

Učinkovitost oba pristupa procijenjena je u različitim veličinama unosa. Rezultati su pokazali da je početni pristup pružio bolju izvedbu od pristupa koji se temelji na stablu unatoč ne tako učinkovitoj upotrebi paralelizma. Međutim, pokazalo se da je zapravo došlo do ubrzanja u odnosu na pristup temeljen na stablu, ali je uveo nova uska grla koja su malo usporila pristup.

Ovaj diplomski rad doprinosi razumijevanju i praktičnoj primjeni programskog jezika Bend. Demonstrirajući kako se Bend može koristiti za implementaciju i optimizaciju paralelnih algoritama, ovaj rad pridonosi rastućoj količini znanja o ovom jeziku u nastajanju i njegovom potencijalu za računalstvo visokih performansi.

Zaključno, ovaj diplomski rad pokazao je potencijal paralelnog računanja u poboljšanju računalnih performansi. Korištenjem programskog jezika Bend prikazana je implementacija i procjena različitih algoritama za konvoluciju, što je u konačnici pridonijelo širem razumijevanju dizajna i implementacije paralelnog algoritma. Rad pruža temelj za buduća istraživanja i optimizaciju u dizajnu algoritama i paralelnom računanju, posebno u kontekstu novih programskih jezika kao što je Bend.

7.1 Budući rad

Kao što je prethodno spomenuto, stroj korišten za demonstraciju prednosti paralelnog izvođenja za konvoluciju bio je MacBook M2 Pro (16-inčni) koji ima 12 jezgri i proces evaluacije je pokrenut na tih 12 jezgri. Uz odgovarajući hardver, dostupne su i druge mogućnosti za daljnja istraživanja. Jedna takva opcija je NVIDIA-in CUDA API koji daje kontrolu nad grafičkim karticama s omogućenom

CUDA putem API-ja. Druga opcija je AMD-ov APP SDK koji radi s AMD-ovim grafičkim karticama putem OpenCL-a.

Budući da Bend nije mogao dovršiti programe s većim ulaznim vrijednostima (kao što su 2^{13} i više), treba istražiti ostaje li paralelno ubrzanje isto ili počinje opadati?

Vjerojatno bi daljnja optimizacija implementacije pristupa temeljenog na stablu mogla polučiti bolje rezultate. Postoji li bolji način za pretvaranje popisa u stablo u paralelnoj prirodi? Druga mogućnost za istraživanje su algoritmi koji se temelje na brzoj Fourierovoj transformaciji i procjena prednosti paralelne obrade na tim pristupima.

Ova je teza pokazala kako se prirodno paralelna priroda algoritma može paralelizirati, ali ostatak algoritma koji je inherentno sekvencijalan također može imati neke paralelne prednosti. Herlihy detaljno istražuje tehnike za takav problem [41] i može biti dobar izvor za istraživanje jesu li neki pristupi primjenjivi na programski jezik Bend.

Dodatak

Ova sekcija sadrži informacije o poboljšanjima koda i kako je vrijeme sekvencijalnog izvođenja poboljšano za 40 puta, a paralelno vrijeme za 12 puta od početne verzije. Početni pristup je trajao 5,15 sekundi u sekvencijalnom izvođenju i 1,07 sekundi u paralelnom izvođenju za izračun konvolucije nad nizovima od 256 elemenata koristeći naivni pristup, dok je za pristup s reducirajućim stablom sekvencijalno izvođenje trajalo 17,55 sekundi, a paralelno oko 6 sekundi.

Bilo je kontraintuitivno da rješenje sa stablastom redukcijom bude toliko sporo. Nakon istraživanja, poput provjere kako se algoritam skalira, otkriveno je da ima lošiju vremensku složenost od $O(n^2)$. Ispostavilo se da algoritam gotovo cijelo vrijeme provodi pretvarajući listu u stablo. Funkcija je zapravo raspakiravala cijelu listu samo kako bi dohvatila pokazivač na prvi element "repova" niza (drugim riječima, kako bi dohvatila drugi element). Nakon određenih izmjena, kreirana je nova funkcija koja je učinkovitije obradila tu konverziju, smanjujući vrijeme s 17 sekundi na 1,53 sekunde. Međutim, paralelno vrijeme i dalje je bilo samo 0,65 sekundi, što je poboljšanje od samo 2,35 puta.

Daljnjim istraživanjem, otkriveno je da je bilo bolje pisati funkcije u Bend jeziku koristeći stil prepoznavanja uzoraka umjesto "Pythonovskog" stila. Na taj način, funkcija može prepoznati strukturu i dodijeliti njezine komponente argumentima. Tako, dohvaćanje drugog elementa niza može jednostavno prepoznati uzorak (glava, (glava, rep)) i drugi element glave predstavlja drugi element niza. Ovom tehnikom, vrijeme izvođenja kako naivnog, tako i algoritma sa stablastom redukcijom smanjeno je na 0,12 sekundi za sekvencijalno izvođenje i 0,06 sekundi za paralelno u naivnom pristupu, te 0,18 sekundi za sekvencijalno, a 0,09 sekundi za paralelno izvođenje u stablastoj redukciji.

Dalje, otkriveno je da rješenje sa stablastom redukcijom troši oko 1,6 puta više vremena zbog konverzije liste u stablo. Budući da ta konverzija ne može iskoristiti prednosti paralelizma, ona je sporija od naivnog pristupa. Također je otkriveno da trokutasti dio algoritma zauzima oko 1/3 vremena obrade za naivni pristup i postiže maksimalno ubrzanje od x4 jer se proces dijeli na dva nezavisna izračuna trokuta, a svaki trokut računa glavu i rep nezavisno. Za stablastu redukciju, taj dio iznosi oko 22%. Slijedi dekompozicija zadataka kako bi se vidjelo odakle dolaze paralelne prednosti:

Pristup temeljen na listama: 100% - (37% / 4 + 63% / 1,54) = 50% ubrzanja
37% - Izračun trokuta
63% - Skalarni produkti

Nije potpuno jasno gdje skalarni produkti dobivaju svoje performanse, ali moglo bi biti da slučajno dolazi do odvajanja logike u kodu. S obzirom na to da se zna da izračun trokuta dobiva x4 ubrzanje, ostatak mora biti kod skalarnog produkta.

Pristup temeljen na stablu: 100% - (39% + 22% / 4 + 39% / 10) = 50% ubrzanja
39% - Konverzija liste u stablo
22% - Izračun trokuta

39% - Skalarni produkti

Kao što se vidi, skalarni produkti dobivaju ubrzanje od x10, trokuti standardno ubrzanje x4, a za konverziju nema ubrzanja, što dovodi do istog paralelnog ubrzanja od 50%, unatoč tome što ima 1,6 puta više iteracija za izračun.

Budući da konverzija liste u stablo dolazi nakon što se izračunaju trokuti, sljedeća ideja za buduće istraživanje mogla bi biti provjera može li funkcija za izračun trokuta biti "pametnija" i odmah spremiti trokute u stablastu strukturu, preskačući fazu konverzije. Možda bi trokuti i konverzija mogli biti jedna operacija, a ne dvije.

Literatura

[1] Barry Wilkinson and Michael Allen. *Parallel programming: techniques and applications using networked workstations and parallel computers*. Pearson, Upper Saddle River, NJ, 2 edition, March 2004.

[2] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. *Introduction to Parallel Computing*. Addison Wesley, January 2003.

[3] Hari Mohan Srivastava and R. G. Buschman. Theory and applications of convolution integral equations. In *Theory and Applications of Convolution Integral Equations*. Springer Netherlands, 1992.

[4] Ronald N Bracewell. *Fourier Transform and Its Applications*. McGraw-Hill electrical and electronic engineering series. McGraw-Hill, New York, NY, August 1978.

[5] Caleb Sneath and Eduardo Colmenares. Comparative sequential and parallel discrete signal convolution algorithms: A case study. *J. Comput. Sci. Coll.*, 38(7):55–64, apr 2023.

[6] Karas Pavel and Svoboda Davi. *Algorithms for Efficient Computation of Convolution*. InTech, January 2013.

[7] Yann LeCun and Yoshua Bengio. *Convolutional networks for images, speech, and time series*, page 255–258. MIT Press, Cambridge, MA, USA, 1998.

[8] A comprehensive guide to convolutional neural networks. `https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53`. Accessed: 25.08.2024.

[9] Thomas H Cormen and Charles E Leiserson. *Introduction to Algorithms, fourth edition*. MIT Press, London, England, April 2022.

[10] Joseph K Blitzstein and Jessica Hwang. *Introduction to Probability*. Chapman & Hall/CRC Texts in Statistical Science. Chapman & Hall/CRC, Philadelphia, PA, July 2014.

[11] Fast fourier transform wikipedia. `https://en.wikipedia.org/wiki/Fast_Fourier_transform`. Accessed: 15.08.2024.

[12] Alan V Oppenheim, Ronald W Schafer, Mark A Yoder, and Wayne T Padgett. *Discrete-time signal processing*. Pearson, Upper Saddle River, NJ, 3 edition, August 2009.

[13] Rafael C Gonzalez and Richard E Woods. *Digital Image Processing*. Pearson, Upper Saddle River, NJ, 4 edition, March 2017.

[14] E. G. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *ACM Comput. Surv.*, 3(2):67–78, jun 1971.

[15] Cuda features and technical specifications. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#features-and-technical-specifications. Accessed: 15.08.2024.

[16] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, March 2007.

[17] Nvidia developer zone. http://developer.nvidia.com/category/zone/cuda-zone. Accessed: 18.08.2024.

[18] Amd developer central. https://www.amd.com/en/developer.html. Accessed: 18.08.2024.

[19] David B Kirk and Wen-Mei W Hwu. Introduction. In *Programming Massively Parallel Processors*, pages 1–21. Elsevier, 2013.

[20] Jason Sanders and Edward Kandrot. *CUDA by example*. Addison-Wesley Educational, Boston, MA, July 2010.

[21] Cuda features and technical specifications. https://developer.nvidia.com/hpc. Accessed: 30.08.2024.

[22] Benedict Gaster, Lee Howes, David R. Kaeli, Perhaad Mistry, and Dana Schaa. *Heterogeneous Computing with OpenCL*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2011.

[23] Ian Foster and Carl Kesselman, editors. *The grid: blueprint for a new computing infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.

[24] Andrew S Tanenbaum and Maarten van Steen. *Distributed systems*. Pearson, Upper Saddle River, NJ, 2 edition, October 2006.

[25] Rajkumar Buyya. *High Performance Cluster Computing: Architectures and Systems*. Prentice Hall PTR, USA, 1999.

[26] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI*. Scientific and Engineering Computation. MIT Press, London, England, 2 edition, November 1999.

[27] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, apr 2010.

[28] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[29] Ken Kennedy and John R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. 2001.

[30] David A Patterson and John L Hennessy. *Computer Organization and Design MIPS Edition*. The Morgan Kaufmann Series in Computer Architecture and Design. Morgan Kaufmann, Oxford, England, 5 edition, September 2013.

[31] Intel® 64 and ia-32 architectures software developer's manual. `https://www.cos.ufrj.br/~daniel/CL/Intel-Architectures-Manual.pdf`. Accessed: 29.08.2024.

[32] Amdahl's law wikipedia. `https://en.wikipedia.org/wiki/Amdahl%27s_law`. Accessed: 15.08.2024.

[33] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference on - AFIPS '67 (Spring)*, AFIPS '67 (Spring). ACM Press, 1967.

[34] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), page 307–314, New York, NY, USA, 1968. Association for Computing Machinery.

[35] Bitonic sorter wikipedia. `https://en.wikipedia.org/wiki/Bitonic_sorter`. Accessed: 15.08.2024.

[36] Yves Lafont. Interaction combinators. *Inf. Comput.*, 137:69–101, 1997.

[37] Higher order company official website. `https://higherorderco.com/`. Accessed: 15.08.2024.

[38] Install rust. `https://www.rust-lang.org/tools/install`. Accessed: 27.08.2024.

[39] Install gcc. `https://gcc.gnu.org/install/`. Accessed: 27.08.2024.

[40] Visual studio code. `https://code.visualstudio.com/`. Accessed: 27.08.2024.

[41] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.

Popis slika

Dodatak

Ova sekcija sadrži informacije o poboljšanjima koda i kako je vrijeme sekvencijalnog izvođenja poboljšano za 40 puta, a paralelno vrijeme za 12 puta od početne verzije. Početni pristup je trajao 5,15 sekundi u sekvencijalnom izvođenju i 1,07 sekundi u paralelnom izvođenju za izračun konvolucije nad nizovima od 256 elemenata koristeći naivni pristup, dok je za pristup s reducirajućim stablom sekvencijalno izvođenje trajalo 17,55 sekundi, a paralelno oko 6 sekundi.

Bilo je kontraintuitivno da rješenje sa stablastom redukcijom bude toliko sporo. Nakon istraživanja, poput provjere kako se algoritam skalira, otkriveno je da ima lošiju vremensku složenost od $O(n^2)$. Ispostavilo se da algoritam gotovo cijelo vrijeme provodi pretvarajući listu u stablo. Funkcija je zapravo raspakiravala cijelu listu samo kako bi dohvatila pokazivač na prvi element "repova" niza (drugim riječima, kako bi dohvatila drugi element). Nakon određenih izmjena, kreirana je nova funkcija koja je učinkovitije obradila tu konverziju, smanjujući vrijeme s 17 sekundi na 1,53 sekunde. Međutim, paralelno vrijeme i dalje je bilo samo 0,65 sekundi, što je poboljšanje od samo 2,35 puta.

Daljnjim istraživanjem, otkriveno je da je bilo bolje pisati funkcije u Bend jeziku koristeći stil prepoznavanja uzoraka umjesto "Pythonovskog" stila. Na taj način, funkcija može prepoznati strukturu i dodijeliti njezine komponente argumentima. Tako, dohvaćanje drugog elementa niza može jednostavno prepoznati uzorak (glava, (glava, rep)) i drugi element glave predstavlja drugi element niza. Ovom tehnikom, vrijeme izvođenja kako naivnog, tako i algoritma sa stablastom redukcijom smanjeno je na 0,12 sekundi za sekvencijalno izvođenje i 0,06 sekundi za paralelno u naivnom pristupu, te 0,18 sekundi za sekvencijalno, a 0,09 sekundi za paralelno izvođenje u stablastoj redukciji.

Dalje, otkriveno je da rješenje sa stablastom redukcijom troši oko 1,6 puta više vremena zbog konverzije liste u stablo. Budući da ta konverzija ne može iskoristiti prednosti paralelizma, ona je sporija od naivnog pristupa. Također je otkriveno da trokutasti dio algoritma zauzima oko 1/3 vremena obrade za naivni pristup i postiže maksimalno ubrzanje od x4 jer se proces dijeli na dva nezavisna izračuna trokuta, a svaki trokut računa glavu i rep nezavisno. Za stablastu redukciju, taj dio iznosi oko 22%. Slijedi dekompozicija zadataka kako bi se vidjelo odakle dolaze paralelne prednosti:

Pristup temeljen na listama: 100% - (37% / 4 + 63% / 1,54) = 50% ubrzanja
37% - Izračun trokuta
63% - Skalarni produkti

Nije potpuno jasno gdje skalarni produkti dobivaju svoje performanse, ali moglo bi biti da slučajno dolazi do odvajanja logike u kodu. S obzirom na to da se zna da izračun trokuta dobiva x4 ubrzanje, ostatak mora biti kod skalarnog produkta.

Pristup temeljen na stablu: 100% - (39% + 22% / 4 + 39% / 10) = 50% ubrzanja
39% - Konverzija liste u stablo
22% - Izračun trokuta

39% - Skalarni produkti

Kao što se vidi, skalarni produkti dobivaju ubrzanje od x10, trokuti standardno ubrzanje x4, a za konverziju nema ubrzanja, što dovodi do istog paralelnog ubrzanja od 50%, unatoč tome što ima 1,6 puta više iteracija za izračun.

Budući da konverzija liste u stablo dolazi nakon što se izračunaju trokuti, sljedeća ideja za buduće istraživanje mogla bi biti provjera može li funkcija za izračun trokuta biti "pametnija" i odmah spremiti trokute u stablastu strukturu, preskačući fazu konverzije. Možda bi trokuti i konverzija mogli biti jedna operacija, a ne dvije.