Implementacija algoritama na grafovima pomoću GPU

Musap, Loredana

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Loredana Musap

IMPLEMENTACIJA ALGORITAMA NA GRAFOVIMA POMOĆU GPU

Diplomski rad

Voditelj rada:
Doc. dr. sc. Zvonimir Bujanović

Zagreb, Srpanj, 2016

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Sadržaj

Uvod

Krajem 1980-tih interes javnosti za operacijskim sustavima koji koriste grafička korisnička sučelja (GUI) potakao je proizvođače (uvijek u korak s tržišnim zahtjevima) na razvoj specijaliziranih procesora koji će poboljšati prikaz i pomoći ostale grafičke procese na računalu. To su bili začetci nove vrste procesora - GPU[1].

Jedan od značajnijih događaja u razvoju GPU bilo je prvo programsko sučelje za grafički procesor koje je, u siječnju 1992. godine, na tržište izbacila tvrtka Silicon Graphics u obliku biblioteke nazvane OpenGL. Tada je već bila popularizirana i upotreba 3D grafike koja je svoje mjesto našla u mnogim područjima primjene od vojske do znanosti.

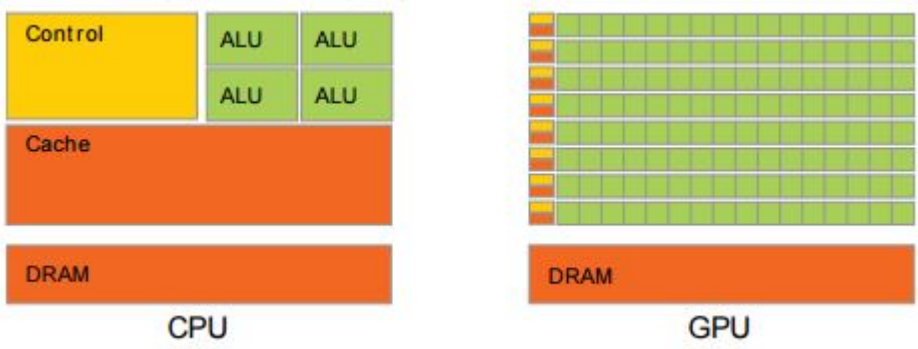
Tržište je sve više raslo, 3D igre su postajale sve popularnije, zbog čega su i grafički procesori postajali sve bolji i napredniji kako bi išli u korak s programskim napretkom. Tada se dogodio drugi značajni pomak - na tržištu se pojavila NVIDIA GeForce serija 3 i novi standard kojeg su te grafičke kartice podržale - DirectX 8.0. Njime su programeri po prvi put dobili kontrolu nad točnim izračunima koje su se odvijale na GPU.

GPU se do tog trenutka razvio u izrazito jaki procesor - morao je podržavati brojne izračune i renderiranja da bi podržao prikaz složenih trodimenzionalnih objekata, zajedno s njihovim teksturama. Ne čudi da su programeri htjeli isprobati njegovu moć i na drugim izračunima, ne samo onima vezanim uz grafiku. Ali, takvo programsko sučelje nije postojalo. Zato su oni posebno znatiželjni programirali na GPU koristeći postojeća sučelja (OpenGL ili DirectX), na način da su ih pokušavali zavarati prikazujući svoje podatke kao da su podatci koje treba renderirati. To naravno nije funkcioniralo savršeno, jer postojeća sučelja nisu bila namijenjena za to, ali je dalo uvid u to da bi GPU, s prikladnim sučeljem, u budućnosti zaista mogao služiti i za efikasno računanje nad podacima koji nemaju veze sa samom grafikom.

Prava snaga GPU je u izrazitom paralelizmu. Razlog je što je GPU upravo prilagođen

[1]GPU (Graphics Processing Unit) - Grafička procesna jedinica

za intenzivne, izrazito paralelne izračune u grafici pa je i dizajniran na način da je puno više tranzistora posvećeno procesuiranju podataka naspram dohvatu i kontroli toka podataka (što je slučaj kod CPU[2]). Ako pogledamo Sliku 0.1 lijevo vidimo pojednostavljeni prikaz četverojezgrenog CPU. Svaka od tih jezgri zapravo je jedna aritmetičko logička jedinica (engl. *arithmetic logic unit*) ili skraćeno ALU. ALU je jedinica zadužena za aritmetičke i logičke operacije na procesoru. Desno na istoj slici nalazi se pojednostavljeni prikaz GPU, koji se sastoji od velikog broja ALU naspram CPU. Svaka od ovih jedinica sposobna je raditi izračune (aritmetičke i logičke operacije) neovisno o ostalim jedinicama, iz čega proizlazi paralelizam na GPU.

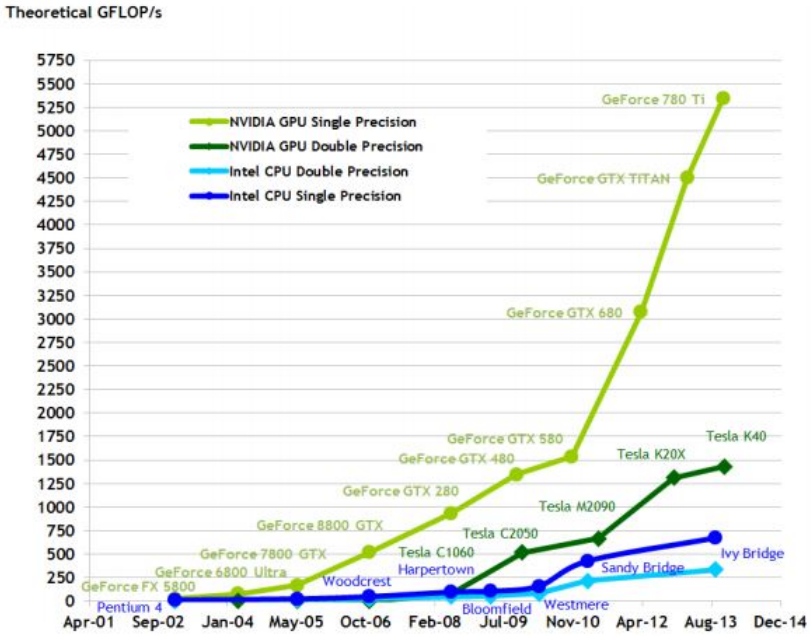


Slika 0.1: Različitost u građi CPU i GPU.

Prilikom mjerenja performansi procesora u FLOPS-ima [3] primjećuje se velika prednost GPU nad CPU, kao što vidimo na Slici 0.2. Razlog je u već spomenutom paralelizmu kojeg GPU nudi (zbog prilagođene arhitekture), te odgovarajućim strukturama podataka nad kojima GPU pokazuje pravu moć.

[2]CPU (Central Processing Unit - centralna procesna jedinica

[3]FLOPS (Floating point operations per second) - Broj operacija pomičnog zareza po sekundi je mjera performansi procesora, korisna u poljima znanstvenog računanja koje koriste aritmetiku pomičnog zareza.



Slika 0.2: Usporedba FLOPS u CPU i GPU.

Zadnji i najznačajniji događaj u razvoju programiranja za GPU dogodio se u studenom 2006. godine kad je NVIDIA predstavila CUDA® platformu i programski model. Time je NVIDIA uvela revoluciju u smislu da je prvi put omogućeno vršiti izračune opće namjene na GPU. Programer više ne treba pokušavati varati GPU na način da svoje podatke prikazuje kao npr. piksele koje treba renderirati. Također, programer ne mora poznavati specifične grafičke jezike (OpenGL ili DirectX) koje nazivamo jezici za sjenčanje, zato što CUDA sadrži softversku okolinu koja omogućava korištenje programskog jezika C kao jezika više razine. Dakle, dovoljno je znati standardni programski jezik C te usvojiti neke dodatne funkcionalnosti koje CUDA pruža kako bi se direktno komuniciralo s GPU.

Iako je programski model CUDA model koji ćemo u daljnjem tekstu obrađivati i detaljnije objašnjavati i koji, kako smo spomenuli, radi na NVIDIA grafičkim karticama, spomenimo i jedan jako sličan programski model - OpenCL. Osnovna razlika je to što je OpenCL razvijen za AMD grafičke kartice. Prva je verzija objavljena nekoliko godina nakon CUDA-e, preciznije u kolovozu 2009. godine. Ideja, organizacija, pa čak i izgled koda jako su slični CUDA-i što je korisno jer znači da kad naučimo koristiti jedan model brzo i lako se može snaći i u drugom.

Većina današnjih grafičkih kartica podržava CUDA ili OpenCL programski model što

znači da je ovakav način programiranja na dohvat ruke i da gotovo svatko od nas može iskoristiti prednosti koje nam pruža. Iz tog razloga sljedeća su poglavlja posvećena objašnjenju zašto i kako koristiti CUDA programski model.

Poglavlje 1

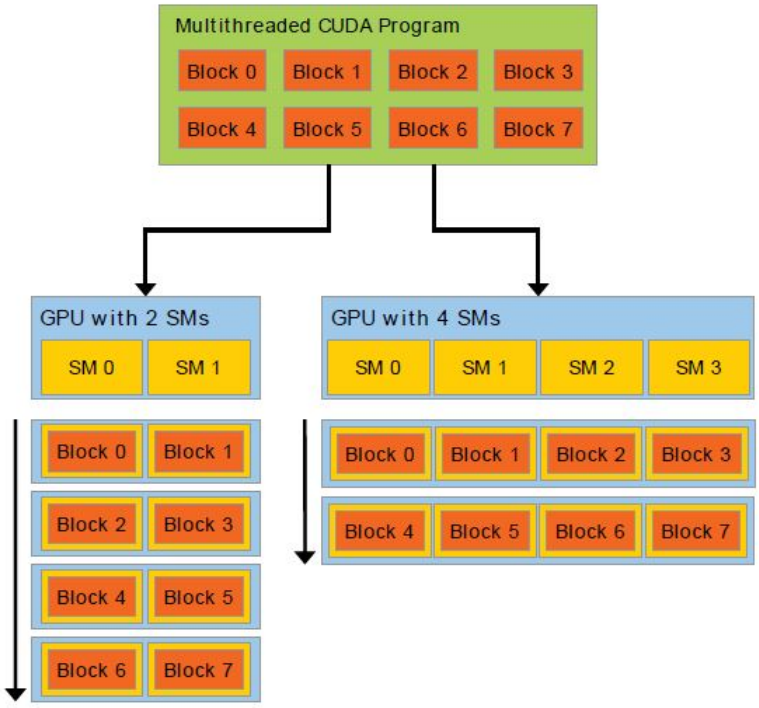
Programski model CUDA®.

Intenzivan razvoj GPU-a te paralelnog programiranja na istom nije napravljen s ciljem da izgura sekvencijalno programiranje na CPU. Neke zadatke za koji su namijenjeni za CPU, ovako dizajniran GPU ne bi niti mogao izvesti. Nameće se zaključak da bi idealno rješenje bilo da se CPU i GPU objedine u programskom modelu gdje će biti moguće određene probleme adresirati na CPU, a neke druge na GPU. Upravo ta paradigma heterogenog programiranja implementirana je u CUDA-i. U CUDA terminologiji CPU i pripadnu memoriju nazvat ćemo domaćin (engl. host), a GPU i pripadnu memoriju uređaj (engl. device).

Paralelizam kojeg smo spomenuli kod CUDA-e je takozvani podatkovni paralelizam, a znači da paralelizam postiže tako da se objekt (složeni podatak) nad kojim provodimo algoritam razloži na sastavne dijelove nad kojima se mogu operacije izvoditi paralelno. Jednostavan primjer je niz odnosno vektor čijim elementima pristupamo na jednostavan način, preko indeksa. Paralelizam možemo postići tako da, na primjer, operaciju koja se izvršava nad jednim elementom niza dodijelimo jednoj dretvi. Ako su operacije nad pojedinim elementima međusobno nezavisne, dretve će moći izvršavati istovremeno, ali neovisno jedna o drugoj, svaka nad svojim dijelom podatka i time postići (za prikladne probleme) i do 10 puta brže izvršavanje (na GPU) nego kod klasičnog sekvencijalnog programiranja (na CPU).

Još jedno važno svojstvo paralelnog programiranja kojeg CUDA pruža je automatska skalabilnost. Arhitektura CUDE-a je organizirana kao niz skalabilnih jedinica koje se nazivaju streaming multiprocesori (u daljnjem tekstu SM). Prilikom pokretanja jednog bloka dretvi (dretve su particionirane u skupove koje zovemo blokovi i koji se paralelno izvršavaju posve neovisno jedan o drugom; kasnije će to biti detaljnije objašnjeno) taj blok se dodjeljuje onom SM-u (engl. *streaming multiprocesoru*) koji trenutno ima kapacitet za obraditi ga i tako redom. Prilikom pisanja programa nebitno je koliko GPU na kojem ćemo

kod izvršavati ima SM-ova. GPU s više SM-ova isti će kod izvršiti brže, ali to je potpuno automatizirano. Bolji uvid u situaciju imamo ako pogledamo Sliku 1.1. [1]



Slika 1.1: Skalabilnost obzirom na broj SM-a na GPU.

1.1 Deklaracije funkcija

Kao što smo već spomenuli, CUDA omogućava heterogeno programiranje - programiranje i na domaćinu (CPU) i na uređaju (GPU). Dakle, na neki način kompajleru treba dati do znanja gdje želimo da se naša funkcija izvršava te želimo li možda da se ista funkcija izvršava paralelno u više dretvi. Kako bi to omogućila, CUDA nudi proširenje klasičnih funkcija na način da ih označimo s ključnom riječju `__global__`, `__device__` te `__host__`.

S `__device__` označene su funkcije koje se izvršavaju na uređaju, a s `__host__` one koje će se izvršavati na domaćinu. U kasnijem tekstu kad bude riječi više o memoriji i pristupu istoj detaljnije će biti objašnjeno čemu je dopušten pristup u kojoj funkciji.

[1] Sve slike u ovom radu preuzete su iz izvora [1], [2], [3], [4], [5] te [6].

Ukoliko funkciju definiramo s obje ključne riječi istovremeno - i `__device__` i `__host__` neće doći do greške. U tom slučaju isti će se kod kompajlirati i za uređaj i za domaćina.

S `__global__` su označene posebne vrste funkcija - kerneli. One su drugačije i važne jer omogućuju cijelu paradigmu paralelnog programiranja. Izvršavaju se također na uređaju, ali ono što ih razlikuje od prije spomenutih funkcija `__device__` jest njihovo izvršavanje koje ide u paralelnim dretvama. Programeru je dana potpuna sloboda da definira samu funkciju te da odredi raspored paralelnih dretvi (i neke dodatne parametre). Kako je ovaj tip funkcija poseban, dolazi i s nekoliko ograničenja u odnosu na klasične funkcije. Povratni tip ove funkcije uvijek mora biti `void`. Odnosno, ovakva funkcija ne može vratiti neki podatak. Ne podržava `static` varijable niti pokazivače. Iz koda tih funkcija moguće je pristupati samo memoriji uređaja.

1.2 Paralelizacija i kerneli

Algoritmi nad vektorima, počevši već i od jednodimenzionalnih, često su izrazito pogodni za paralelizaciju. Pogledajmo primjer jednostavne funkcije u programu C (na CPU) u kojoj zbrajamo dva vektora A i B.

```
void VectAdd(float *A, float *B, float *C)
{
    for(int i = 0 ; i < N; i++)
    {
        C[i] = A[i] + B[i];
    }
}
```

Što se dogodilo? N puta smo napravili istu operaciju, jednu za drugom. Uočimo da su operacije koje izvodimo u petlji neovisne jedna o drugoj i da bi se mogle izvoditi paralelno.

Kako paralelizirati ovaj jednostavan primjer? Svako od pridruživanja (njih N) pridijelit ćemo jednoj od N dretvi koje bi se izvršavale paralelno, neovisno jedna o drugoj pa sljedeća operacija ne bi čekala prethodnu da se izvrši kao kod sekvencijalnog koda na CPU. Time bi za velike N-ove postigli veliko ubrzanje čak i ovakve jednostavne funkcije.

Dakle, prema definiciji kernel funkcije, ta jedna operacija, zbroj i-tog člana vektora bila bi jedna kernel funkcija, označena s ključnom riječju `__global__`. Pogledajmo primjer:

```
__global__ void VectAdd(float *A, float *B, float *C, int N)
{
    int i = threadIdx.x;
```

```
    if(i < N)
    {
        C[i] = A[i] + B[i];
    }
}
```

Možemo zamisliti da GPU pozivom jedne ovakve kernel funkcije napravi N kopija kernela koji se izvršavaju paralelno. Svaka od dretvi izvršiti će kod unutar te funkcije, a `threadIdx.x` predstavlja indeks dretve koji nam omogućava da svakoj dretvi pridijelimo njezin element u polju C kojeg će ona izračunati.

Ako zamislimo da imamo N multiprocesora od kojih svaki izvrši jednu spomenutu kopiju kernela (izračuna jedan element vektora C), cijeli izračun vektora C (svih N elemenata) odvijat će se paralelno na N multiprocesora koji su međusobno neovisni. U tom slučaju bi vremenska složenost izračunavanja zbroja dvaju vektora pala sa O(N) na O(1)! Naravno, u praksi za dovoljno velike N nemamo tako mnogo multiprocesora, ali je faktor ubrzanja u odnosu na sekvencijalni algoritam i dalje upravo jednak broju multiprocesora. Ovo je jedan jako jednostavan primjer podatkovnog paralelizma (iste operacije se u paraleli izvršavaju na različitim podacima) koji je kao što smo već spomenuli osnova tehnologije CUDA.

1.3 Dretve, blokovi i poziv kernela

Pitanje koje preostaje je kako pozvati tu kernel funkciju. Pogledajmo kako izgleda jedan jednostavan primjer poziva, za kernel iz gornjeg primjera.

```
VecAdd<<<1, N>>>(A, B, C);
```

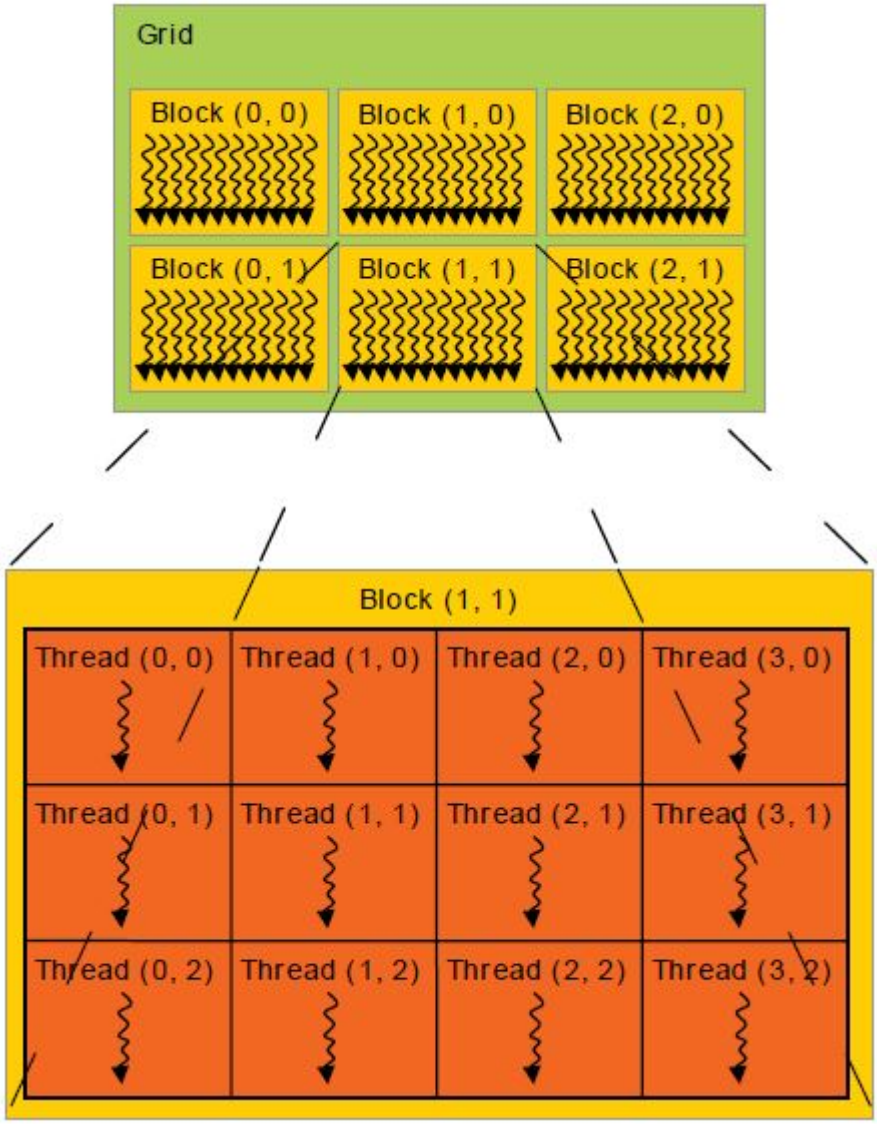
Razlika između sintakse poziva funkcije u standardnom C-u i u CUDA-i jesu trostruke šiljaste zagrade te dva parametra unutar njih, što nazivamo konfiguracija izvršavanja kernela. To nisu parametri koji se šalju uređaju kao što su oni unutar običnih zagrada nego parametri koji određuju na koji način će se pozvati uređaj, odnosno, koliko dretvi će se pozvati za izvršavanje koda.

Kako bi definirali parametre te način organizacije dretvi pri pozivanju potrebno je pojasniti hijerarhiju dretvi. Ta hijerarhija ima dvije razine - blokovi dretvi te mreža blokova.

Sve dretve pokrenute jednim pozivom kernela čine *mrežu* (engl. grid). Sve dretve u mreži dijele istu globalnu memoriju. Mreža se sastoji od više *blokova* (engl. block) dretvi. Blok je skupina dretvi koje mogu međusobno komunicirati. Dakle, dretve koje se nalaze u

različitim blokovima ne mogu međusobno komunicirati. Svaki blok u mreži ima isti broj dretvi.

Radi bolje vizualizacije proučit ćemo Sliku 1.2.

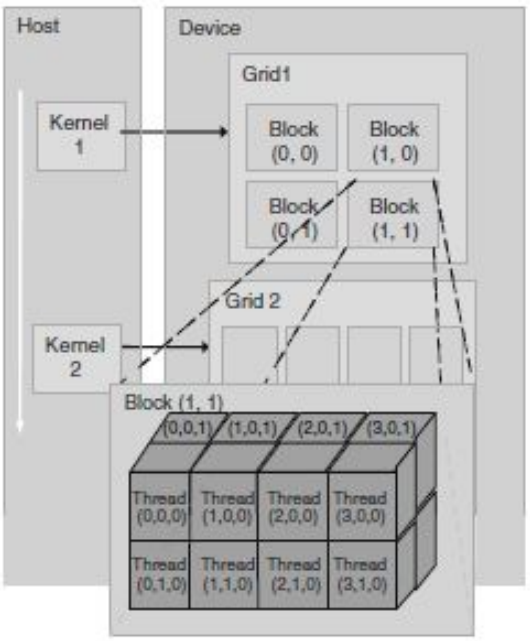


Slika 1.2: Hijerarhija dretvi.

Sada kada su poznati pojmovi mreže i bloka moguće je definirati i što prije spomenuti parametri (u šiljastim zagradama) u pozivu kernela znače. Prvi parametar je broj blokova u mreži, drugi parametar je broj dretvi u bloku. Dakle, pri svakom pozivu kernelu programer zadaje broj blokova u mreži te broj dretvi po bloku, a time posredno i točan broj dretvi koje će se moći paralelno izvršavati (na uređaju). Točan broj dretvi u mreži u jednom pozivu kernelu biti će:

$$ukupan\_broj\_dretvi\_u\_mrezi = broj\_blokova\_u\_mrezi \times broj\_dretvi\_po\_bloku$$

Prilikom poziva kernelu (posredno) se definiraju i dimenzije parametara, odnosno, dimenzija mreže te dimenzija blokova dretvi. Svi blokovi u jednoj mreži biti će iste dimenzije. Pogledajmo sada Sliku 1.3 koja prikazuje mrežu blokova i dretvi. Primijetit ćemo kako je mreža dimenzije 2, a svaki blok u toj mreži dimenzije 3. Maksimalne dimenzije koje se mogu postaviti, za oba parametra su 3. Minimalna dimenzija za oba parametra je 1. U praksi se najčešće koristi upravo dvodimenzionalna mreža blokova te trodimenzionalni blokovi dretvi, baš kao u primjeru sa slike. Dakle, prilikom kodiranja i postavljanja parametara treba izabrati i njihovu dimenziju (1, 2 ili 3).



Slika 1.3: Mreža blokova i dretvi.

Samo su dva tipa podataka kojeg parametri mogu biti, `int` ili `dim3`, i to vrijedi za oba parametra. Jednodimenzionalni parametar će biti tipa `int`, a u slučaju parametra dimenzije 2 ili 3 koristimo tip `dim3`. Ime tog tipa sugerira da se radi (samo) o trodimenzionalom parametru, pa može biti zbunjujuće kako iskoristiti tip `dim3` za dvodimenzionalan parametar. Takvo nešto omogućava CUDA jer će ako ostavimo "suvišnu" dimenziju praznom neupotrijebljena polja automatski biti inicijalizirana na 1 i zanemarena. Pogledajmo sada i primjer poziva nekog kernela koji bi prema rasporedu blokova i dretvi odgovarao Slici 1.3.

```
__global__ void kernel_Function(float *A, float *B, float *C)
{
    ...
}

int main()
{
    ...
    dim3 threadsPerBlock(4, 2, 2);
    dim3 numBlocks(2,2);
    kernel_Function<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

Koju ćemo dimenziju izabrati za blok dretvi ovisi o tipu podataka kojeg obrađujemo u kernelu. Sjetimo se primjerice jednostavnog zbrajanja dva jednodimenzionalna vektora. Tada smo imali i jednodimenzionalne blokove. U slučaju matrica, trebati će nam dvodimenzionalni blokovi, i dalje analogno.

Prije spomenutu jednostavnu formulu broja dretvi u mreži sada možemo nadopuniti, tako da uzmemo u obzir i dimenzije. S *numBlocks* označen je broj blokova u mreži, a s *threadsPerBlocks* broj dretvi u svakom bloku. Slovo nakon točke odnosi se na neku od dimenzija parametara. Prisjetimo se, ako je neka od dimenzija nije navedena (u ovom slučaju kod *numBlocks*), ona se postavlja automatski na 1.

$$broj\_blokova\_u\_mrezi = numBlocks.x \times numBlocks.y \times numBlocks.z$$
$$broj\_dretvi\_u\_mrezi = threadsPerBlocks.x \times threadsPerBlocks.y \times threadsPerBlocks.z$$
$$ukupan\_broj\_dretvi\_u\_mrezi = broj\_blokova\_u\_mrezi \times broj\_dretvi\_po\_bloku$$

Prema ovoj formuli, u gornjem primjeru broj blokova je $2 \times 2 \times 1 = 4$. Broj dretvi u jednom bloku je $4 \times 2 \times 2 = 16$. Dakle, ukupan broj dretvi u mreži jest $4 \times 16 = 64$.

Za sada smo definirali parametre te njihove dimenzije, postavili formule po kojima se može izračunati ukupan broj dretvi, ali što je s samim vrijednostima parametara i kako

odabrati odgovarajuće za naše konkretne programe koje pišemo? Dodatno pitanje je i možemo li postaviti bilo koju vrijednost parametara koju želimo? Za odgovor se treba osvrnuti na fizička ograničenja GPU-a koji je na raspolaganju, jer bi bilo nerealno očekivati da za bilo koji odabrani parametar stroj ispravno radi.

Ograničenja broja dretvi po bloku te blokova u mreži ovise o uređaju na kojem izvršavamo program. Za računalo Fermi, odnosno grafičku karticu Nvidia Tesla S2050 (na njemu će se pokretati svi algoritmi u ovom radu), maksimalan broj dretvi po bloku je 1,024, a maksimalan broj blokova u mreži (1-dim) je 65,536. Zamislimo teoretski u ovakvoj situaciji primjer u kojem želimo zbrojiti dva vektora dimenzije 33,554,432 (na GPU). Ako postavimo dimenzije bloka na (maksimalnih) 1,024 dretvi, broj blokova u mreži morati će biti $33554432/1024 = 32768 < 65536$. Oba broja zadovoljavaju zahtjeve našeg uređaja. Stavimo li dimenzije bloka na 512, sada će broj blokova u mreži biti $33554432/512 = 65536 \not< 65536$, što već nije zadovoljavajuće. Daljnjim smanjivanjem broja dretvi po bloku broj blokova u mreži biti će također prevelik. Dakle, u ovom konkretnom slučaju moguće je postaviti broj dretvi po bloku na 1024. No, ako na izbor imamo više odgovarajućih vrijednosti izabrat ćemo onaj koji prilikom testiranja algoritma daje bolje rezultate.

Ono što je zanimljivo, a često može predstavljati i problem, je da se prilikom pokretanja programa s vrijednostima parametara koje prelaze dopuštene naizgled čini da program dobro radi. Razlog tome je što se program neće zaustaviti niti izbaciti poruku o grešci kao što bi možda bilo za očekivati. Iako CUDA ne javlja nikakvu grešku, nikako ne znači da program dobro radi. Ono što se događa je da CUDA zanemaruje sve što izlazi izvan granica. Ako bi u takvoj situaciji imali niz kojem pridijeljujemo vrijednosti unutar kernela, dio niza bismo imao neke nasumične vrijednosti, a ne one koje bi očekivali. To može biti jako nezgodno jer se često može dogoditi da niti ne primijetimo da smo pogriješili. CUDA ipak ima implementirane funkcije koje nas upozoravaju o prelasku dopuštenih granica, ali, moramo ih pozvati u kodu.

Poziv je jako jednostavan, radi se o jednoj jedinoj funkciji:

```
cudaError_t err = cudaGetLastError();
printf("Error %d : %s",err,cudaGetErrorString(err));
```

Važno je taj dio koda postaviti nakon poziva kernelu (s mogućim krivim parametrima) kako bi dobili ovakvu poruku o grešci:

```
Dimenzija vektora koje zbrajamo: 33554430

Max broj blokova u mreži: 65536
Max broj dretvi po bloku: 1024

Broj blokova u mreži: 65536
Broj dretvi po bloku: 512

Error 9 : invalid configuration argument
```

Slika 1.4: Ispis greške.

U slučaju dobrog izbora parametara odgovarajuća poruka izgleda ovako:

```
Dimenzija vektora koje zbrajamo: 33554430

Max broj blokova u mreži: 65536
Max broj dretvi po bloku: 1024

Broj blokova u mreži: 32768
Broj dretvi po bloku: 1024

Error 0 : no error
```

Slika 1.5: Ispis bez greške.

Ali, što ako se pokrećemo ili pišemo program na nekom uređaju kojeg ne poznajemo i ne znamo njegove mogućnosti niti specifikacije, kako znati koja su ograničenja ako ne želimo nasumce pogađati brojeve? Rješenje je jednostavno - CUDA pruža mogućnost da ispitamo GPU o njegovim specifikacijama. Pogledajmo primjer koda te odgovarajućeg ispisa za računalo Fermi.

```
int dev = 0;
cudaDeviceProp deviceProp;
cudaGetDeviceProperties(&deviceProp, dev);

printf("Device %d: \"%s\"\n\n", dev, deviceProp.name);

printf(" Max Texture Dimension Size (x,y,z)  1D=(%d), 2D=(%d,%d), 3D
    =(%d,%d,%d)\n\n",
        deviceProp.maxTexture1D ,
```

```
        deviceProp.maxTexture2D[0], deviceProp.maxTexture2D[1],
        deviceProp.maxTexture3D[0], deviceProp.maxTexture3D[1],
           deviceProp.maxTexture3D[2]);

printf(" Max number of threads per block: %d\n\n", deviceProp.
    maxThreadsPerBlock);
```

Ispis daje tražene podatke o uređaju na kojem je kod pokrenut:

```
Device 0: "Tesla S2050"

 Max Texture Dimension Size (x,y,z)  1D=(65536), 2D=(65536,65535), 3D=(2048,2048,2048)

 Max number of threads per block: 1024
```

Slika 1.6: Ispis specifikacija o uređaju.

Ovo nisu svi podatci koje možemo saznati iz strukture `cudaDeviceProp`. Još jedna korisna informacija koju preko ovih svojstava možemo saznati jest broj aktivnih GPU koji su na raspolaganju za korištenje (moguće je da je ponuđeno više od jednog uređaja) te broj multiprocesora na svakom od njih. To je korisno jer kod biranja na kojem želimo izvršiti kod možemo izabrati onaj GPU s najviše multiprocesora.

1.4 Predefinirane varijable

U prethodnom tekstu spomenute su kernel funkcije, objašnjen je način pozivanja te hijerarhija dretvi. Sljedeći je korak detaljnije pogledati sadržaj tih funkcija. Ono što je preostalo objasniti je kako dretva preko univerzalnog poziva kernela zna nad kojim će ona točno dijelom podatka računati te gdje će rezultat spremati. Sve to definirano je u samom kodu kernela.

Kako bismo identificirali svaku dretvu te dimenzije zadane u pozivu kernela CUDA na raspolaganje stavlja četiri predefinirane varijable - `threadIdx`, `blockIdx`, `blockDim` te `grimDim`.

Svaka je dretva koja se stvori jedinstveno određena svojim identifikatorom. Prilikom pokretanja prva će dobiti vrijednost identifikatora jednaku 0, sljedeća 1, i tako do $N - 1$ (gdje je N broj dretvi definiran pozivom kernela).

Ukoliko pozivamo kernel s jednim blokom i N dretvi u tom bloku:

```
VectAdd<<<1, N>>>(A, B, C);
```

Vrijednost varijable `threadIdx.x` u kernelu imati će vrijednost 0 do $N-1$ na temelju čega ćemo znati o kojoj je dretvi riječ i razlikovati ih međusobno. Dakle, ta varijabla predstavlja identifikator dretve unutar bloka. To bi značilo da bi (jako jednostavan) kernel koji odgovara ovom pozivu izgledao ovako:

```
__global__ VectAdd(double *A, double *B, double *C)
{
    int i = threadIdx.x;

    C[i] = A[i] + B[i];
}
```

Važno je napomenuti da bi u slučaju dvodimenzionalnog bloka uz `threadIdx.x` imali još i `threadIdx.y` odnosno još i `threadIdx.z` u slučaju trodimenzionalnog bloka.

U ovom kodu, kod postavljanja konfiguracije kernela, kompajleru smo rekli da pozove samo jedan blok i u njemu svih N dretvi, koliko je potrebno. Program radi korektno, ali za jako male N-ove. Bez organizacije dretvi u blokove bilo bi nemoguće izvršavati operacije nad nizovima koji imaju više elemenata nego što je maksimum broja dretvi. Taj maksimum je, sjetimo se, čak za jače strojeve (kao što je naš) još uvijek jako nizak - 1024 dretve. To nisu toliko veliki nizovi i izračuni gdje bi GPU mogao iskazati svoje prave performanse.

Sjetimo se da kod kombinacije s maksimalnim dopuštenim brojem blokova i dretvi (na našem stroju) možemo optimalno izvoditi aritmetičke operacije nad nizovima reda veličine nekoliko desetaka milijuna. To su značajne količine podataka gdje CPU postaje izrazito spor te GPU može pokazati svoje performanse (primjenjen na odgovarajuće algoritme).

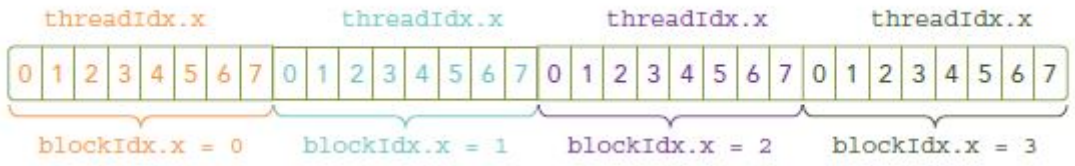
Što se tiče identifikacije bloka analogno je dretvi. Svaki je blok određen svojim identifikatorom (vrijednosti od 0 do $N-1$ gdje je N broj blokova u mreži). Ako bismo imali specifičnu situaciju gdje pozivamo kernel s N blokova i po jednom dretvom u svakom od njih, gore definirani kernel bio bi primjenjiv s jednom jedinom izmjenom - `threadIdx.x` zamijeniti s `blockIdx.x`.

Kernel s takvom konfiguracijom u praksi se ne koristi jer su blokovi važni baš za one situacije kad duljina niza nad kojim se radi prelazi ograničenje broja dretvi u jednom bloku.

Zamislimo da je N maksimalan broj dretvi po bloku, a niz nad kojim želimo izvršiti operaciju je duljine $4N$. Zaključujemo da je potrebno 4 bloka za pohraniti te dretve, po

N u svaki od 4 bloka. Indeksi dretvi unutar svakog bloka kretat će se od 0 do $N-1$. Da bismo pristupili vrijednostima niza, na indeksima od N do $2N-1$, $2N$ do $3N-1$ i $3N$ do $4N-1$ morati ćemo za svaki novi blok "pomaknuti" indeksiranje dretvi za po N. Kako bi to postigli, koristimo ostale dvije predefinirane varijable `blockIdx` te `blockDim`. Sada ćemo dretvu identificirati ovom formulom:

```
int tid = threadIdx.x + blockIdx.x * blockDim.x;
```



Slika 1.7: Linearna organizacija podataka u memoriji.

Opisani postupak nalikuje na stardardno konvertiranje iz dvodimenzionalnog (jedna dimenzija blokovi, jedna dretve) u jednodimenzionalno indeksiranje niza.

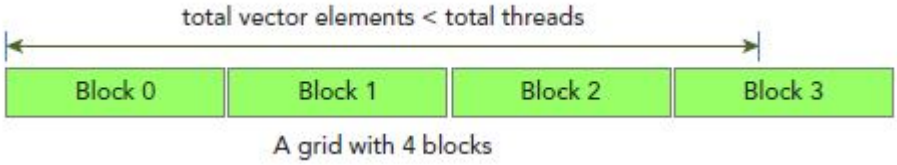
Sada kada je jasno kako se unutar kernela identificiraju podaci, dretve i blokovi s kojima se radi ono što je iduće za definirati je kako odrediti parametre s kojima ćemo pozivati kernel u ovisnosti o duljini niza kojeg predajemo kao parametar funkcije.

Prilikom poziva kernela za rad s nizom od N elemenata trebamo odrediti oba parametra, broj dretvi po bloku i blokova u mreži na način da pozovemo ukupno N paralelnih dretvi (niz je od N elemenata). Prvo što ćemo izabrati je broj dretvi po blokovima. To će biti neki od brojeva potencije 2 kao što su 128, 256, ... Broj koji izaberemo ne smije biti veći od ograničenja kojeg pretpostavlja GPU na kojem pokrećemo. Nazovimo taj broj `threadsPerBlock`. Želimo dretve rasporediti u dovoljan broj blokova tako da postoji po jedna dretva za svaki $i = 1...N$, pa ćemo broj blokova u mreži, nazovimo ga `blocksPerGrid` odredit kao

$$\texttt{blocksPerGrid} = (N + \texttt{threadsPerBlock} - 1)/\texttt{threadsPerBlock}.$$

Prva ideja koju bi programer mogao imati jest podijeliti $N/$`threadsPerBlock`. To nije ispravno, a pogledajmo na jednostavnom primjeru i zašto. Neka je $N = 7$, a `threadsPerBlock` $= 8$, tada za `blocksPerGrid` dobivamo $7/8 = 0$ što očito nije dobro. Ono što zapravo želimo izračunati je najveće cijelo, a navedena formula se jako često koristi u programima kako bi se izbjeglo korištenje funkcije `ceil()`.

S gornjom formulom osigurali smo dovoljan broj dretvi za bilo koju duljinu niza N, ali moguće je da smo pozvali veći broj dretvi od onog koji je potreban. Uzmimo primjer s jako malim brojevima kako bi jasno opisali situaciju. Neka je $N = 5$, a `threadsPerBlock` $= 8$, tada je `blocksPerGrid` $= (5 + 8 - 1)/8 = 1$, a pozvali smo ukupno `threadsPerBlock` $\times$ `blocksPerGrid` $= 8$ dretvi. Pozvali smo $8 - 5 = 3$ dretve viška. Pogledajmo i sliku 1.8 radi lakše vizualizacije problema.



Slika 1.8: Broj pozvanih dretvi veći od broja elemenata.

Kako ne želimo ilegalno pisanje po memoriji (alocirali smo samo N mjesta u nizu) kernelu ćemo kao parametar predati N te u svakom pozivu provjeriti je li gore spomenuti `tid` manji od N i tek tada izvršiti kod samog kernela.

Za kraj poglavlja o kernelima pogledajmo sljedeći kod, koji se sastoji od jednog jako jednostavnog kernela te njegovog poziva, i koji bi u ovom trenutku trebao biti potpuno jasan.

```
__global__ void kernel_sum(float *A, float *B, float *C, int N)
{
    int tid = threadIdx.x * blockDim.x + blockIdx.x;

    if(tid < N)
    {
        C[i] = A[i] + B[i];
    }
}

int main()
{
    ...
    int threadsPerBlock = 1024;
    int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
    kernel_sum<<<blocksPerGrid, threadsPerBlock>>>(A, B, C, N);
    ...
}
```

Sada kada je na osnovnim primjerima razjašnjena cijela ideja kernela - sintakse, definiranja, pozivanja, nije teško to postupno poopćiti na veće dimenzije ili neke kompliciranije primjere.

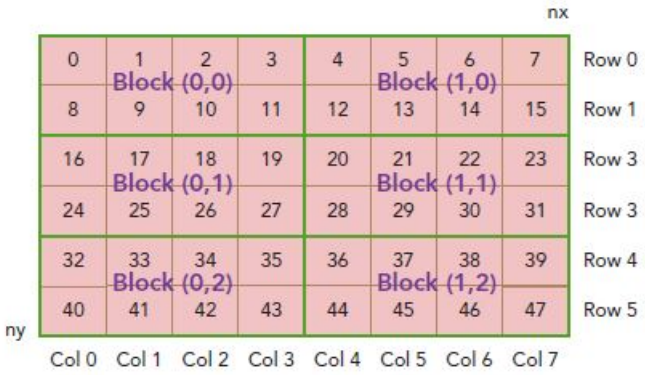
Pogledajmo još kratki primjer s većim dimenzijama, u kojem izračunavamo zbroj dvije matrice a i b dimenzije $N \times N$, te rezultat spremamo u matricu c.

```
__global__ void addMatrix(float *a,float *b,
 float *c, int N)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    int index = i + j * N;

    if (i < N && j < N)
        c[index]= a[index] + b[index];
}

void main()
{
     dim3 threadsPerBlock (256, 256);
     dim3 blocksPerGrid ((N + threadsPerBlock.x - 1) /
        threadsPerBlock.x, (N + threadsPerBlock.y - 1) /
        threadsPerBlock.y);
     addMatrix<<<blocksPerGrid, threadsPerBlock>>>(a, b, c, N);
}
```

Možemo primijetiti da su koraci analogni. Kod izgleda kao da ponavljamo dva puta ono što bi napravili da imamo samo jednu dimenziju, kako je prije opisano. Situacija postaje malo drugačija kod identifikacije dretve unutar kernela. Nizovi su u računalu kao što smo i rekli spremljeni linearno. I isto kao što dvodimenzionalnu mrežu koju predstavljaju jednodimenzionalni blok i dretva moramo ”prevesti” u jednu dimenziju, odnosno prilagoditi linearnom zapisu u memoriji, tako sada još dodatno trebamo dvije dimenzije koje blok ima po definiciji prevesti u jednu dimenziju, odnosno, linearni zapis. Tome služi dodatna linija `i + j * N` u kodu. Kako bismo lakše predočili numeriranje dretvi, pogledajmo Sliku 1.9.

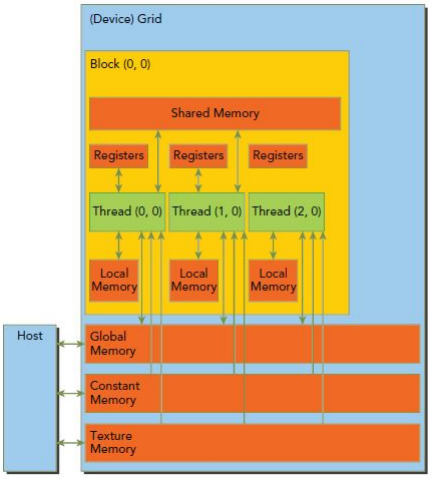


Slika 1.9: Mreža dvodimenzionalnih blokova.

Svaka ćelija u tablici na slici predstavlja dretvu, a broj upisan unutar te ćelije je `index` kojeg mi računamo u gornjem kodu. Sada bi bilo jako lako primjer proširiti i na više dimenzija.

1.5 Memorija

CUDA na raspolaganje stavlja razne vrste memorije koje se razlikuju u kapacitetu, brzini pristupa te dosegu.



Slika 1.10: Model memorije na GPU.

Ako pogledamo Sliku 1.10 vidimo jako puno različitih vrsta memorije, ali na slici se isto tako jasno vidi koji dio koda ima pristup kojoj od njih.

Sada ćemo reći ponešto o svakoj.

Registri (engl. registers)

Registri su najbrža memorija na GPU koja sprema varijable i neke nizove definirane unutar kernela. Može im se pristupiti samo iz kernela u kojem su definirani (doseg kernela) te nakon što se kernel izvrši tim se varijablama više ne može pristupiti (kažemo da imaju životni vijek dretve). Fermi ima kapacitet od 64 "riječi" po dretvi za registre. Svaka riječ je 4 bytea. Dakle, u registre stane najviše 64 inta ili 32 doublea. Ukoliko se napuni kapacitet registra dolazi do tzv. izlijevanja pa se varijabla ipak spremi u lokalnu memoriju. Izlijevanje memorije treba izbjegavati jer je lokalna memorija bitno sporija kao što ćemo vidjeti u sljedećem odjeljku. Što manje mjesta u registru zauzimamo to će se više blokova moći pridijeliti jednom SM-u što generalno ubrzava program. Dretve ne dijele registre, nego pojedina dretva ima pristup samo svojim registrima.

Lokalna memorija (engl. local memory)

Kao što smo već rekli, lokalna memorija služi za pohranjivanje podataka kad se dogodi tzv. izlijevanje iz registara. Samo ime nas navodi na krivi trag - naime ta memorija nalazi se u globalnoj memoriji pa ne pruža maksimalnu brzinu pristupa (sjetimo se, registri su najbrža memorija na GPU, a globalna memorija kako ćemo vidjeti nije toliko brza) tako da alokaciju prevelikih količina podataka u kernelu treba izbjeći ako je ikako moguće, radi brzine izvođenja programa.

Dijeljena memorija (engl. shared memory)

Dijeljena memorija smještena je na samom čipu, pa je shodno tome i jako brza, puno brža od primjerice globalne memorije. Ova memorija ima doseg kernela, ali životni vijek bloka (za razliku od registra koji ima životni vijek dretve). To znači da varijablu dijeljene memorije deklariramo unutar kernela, ali da ju (kako joj samo ime kaže) dijele sve dretve unutar jednog bloka. Kada su sve dretve jednog bloka gotove s izvršavanjem, dijeljena memorija tog bloka se dealocira. Ova memorija je omogućava da dretve (unutar bloka) međusobno komuniciraju. Način na koji kompajler tretira te varijable je da stvori novu kopiju za svaki novi blok koji se pozove, odnosno stvori.

Kako bi varijable unutar kernela bile spremljene u dijeljenu memoriju moramo prilikom deklaracije navesti atribut `__shared__`.

Pogledajmo primjer deklaracije. Kasnije ćemo pokazati i objasniti i jedan jako koristan primjer upotrebe ovakve varijable.

```
__global__ void kernel_example(int *A, int N)
{
    __shared__ int sdata[1000];
    ...
}
```

Ovo je statička alokacija dijeljenje memorije. Broj 1000 u primjeru nasumce je izabran broj. Kasnije će na primjeru biti pokazano kako dinamički alocirati dijeljenu memoriju.

Kao što smo rekli sve dretve jednog bloka imaju pristup istoj varijabli spremljenoj u dijeljenoj memoriji, alociranoj za taj blok. U većini slučajeva u kojima se koristi ovakva memorija bitno je kontrolirati način pristupa dretvi toj memoriji. Za to se koristi funkcija.

```
void __syncthreads();
```

Ova funkcija osigurava da sve dretve u bloku s izvršavanjem dođu upravo do te linije u kojoj je poziv te funkcije. Scenarij u kojem je funkcija `__syncthreads()` potrebna je sljedeći:

1. korak dretva 1 nešto zapiše u dijeljenu memoriju

2. korak poziva se `__syncthreads()`

3. korak sve dretve čitaju iz dijeljene memorije

Bez 2. koraka, odnosno korištenja `__syncthreads()`, moglo bi se dogoditi da neka od dretvi iz 3. koraka čita iz dijeljene memorije prije nego je dretva 1 zapisala pretpostavljenu vrijednost. Funkcija `__syncthreads()` osigurava da sve dretve dođu do te linije, uključujući i dretvu 1. Dakle, sigurni smo da je vrijednost zapisana u dijeljenu memoriju pa se onda može i čitati.

Bilo bi najbolje da dretve uvijek međusobno komuniciraju preko ove memorije (jer je brza), ali je zbog ograničenja veličine ove memorije nekad ipak potrebna komunikacija preko globalne memorije.

Konstantna memorija (engl. constant memory)

Konstantne varijable prepoznajemo po atributu `__constant__` prije deklaracije. Ta memorija biti definirana u globalnom dosegu, izvan bilo kojeg kernela. Ograničena je na samo 64 KB. Statički je deklarirana i vidljiva svim kernelima u istoj kompilacijskoj jedinici. Tim kernelima je dopušteno samo čitanje iz te memorije. Dakle, konstantna memorija mora biti inicijalizirana na domaćinu i to CUDA funkcijom:

```
cudaError_t cudaMemcpyToSymbol(const void* symbol, const void* src,
size_t count);
```

Memorija za teksture (engl. texture memory)

Ovo je još jedna memorija koju se može samo čitati, baš kao i konstantna. Nalazi se na samom čipu što znači da će uglavnom imati veliku brzinu jer se ne treba pristupati memoriji izvan čipa. Koristi se za specifične slučajeve kad uzastopne dretve pristupaju podacima koji su blizu po memorijskoj lokaciji, što je korisno kako bi se ubrzalo izvođenje.

Globalna memorija (engl. global memory)

Globalna memorija je najvećeg kapaciteta, te najčešće korištena memorija. Riječ globalna odnosi se na njezin doseg i životni vijek. Toj memoriji može pristupiti bilo koji SM, u bilo kojem trenutku. Pristup ovoj memoriji jako je spor, no i dalje dosta brži no pristup klasičnog procesora RAM-u.

Jedna stvar je jako bitna u vezi globalne memorije - treba paziti da različite dretve ne pristupaju istoj memorijskoj lokaciji istovremeno jer je u ovom slučaju nemoguće sinkronizirati dretve. Sinkronizacija je moguća samo UNUTAR blokova, a ova memorija obuhvaća sve dretve, a ne samo one unutar jednog bloka. Zato se za probleme u kojima je potrebna sinkronizacija koristi isključivo dijeljena memorija i njena mogućnost sinkronizacije na razini bloka.

Varijabla u glavnoj memoriji može biti deklarirana statički ili dinamički. Statički deklarirana globalna varijabla ima prije deklaracije naveden atribut `__device__`. U sljedećem ćemo se poglavlju detaljnije osvrnuti na dinamički alociranu memoriju.

Kako je globalna memorija najčešće korištena pogledajmo jedan kratki primjer upotrebe (statički deklarirane):

```
__device__ float devData;
```

```
__global__ void checkGlobalVariable()
{
    printf("The value of global variable is: %f\n", devData);
    devData += 2.0f;
}

int main()
{
    float value = 3.14f;
    // kopiranje u globalnu memoriju, host -> device
    cudaMemcpyToSymbol(devData, &value, sizeof(float));
    printf("Host: copied %f to the global variable\n", value);
    // poziv kernelu
    checkGlobalVariable <<<1, 1>>>();
    // kopiranje globalne varijable, device -> host
    cudaMemcpyFromSymbol(&value, devData, sizeof(float));
    printf("Host: the value changed by the kernel to %f\n", value);
}
```

Memoriju smo statički alocirali, zatim joj pridružili vrijednost CUDA funkcijom `cudaMemcpyToSymbol()`. Ova funkcija kao parametre prima adresu varijable (s CPU) iz koje kopiramo vrijednost (`&value`), globalnu varijablu (na GPU) u koju kopiramo, koja je već statički deklarirana (`devData`), te veličinu jedne, odnosno, druge varijable (moraju se podudarati inače CUDA neprevidljivo reagira). Globalnoj varijabli pristupamo iz kernela, gdje ju možemo čitati i mijenjati ju (pisati na tu memorijsku lokaciju). U slučaju da želimo provjeriti rezultat izračuna u kernelu (izvan kernela) moramo rezultat s GPU kopirati na CPU kako bi mogli pristupiti toj vrijednosti (ispisati ju). Tome služi funkcija `cudaMemcpyFromSymbol()` koja kopira vrijednost iz globalne varijable na GPU na CPU na analogan način kao što `cudaMemcpyToSymbol()` kopira s CPU na GPU. Tek kada vrijednost imamo spremljenu u varijabli na CPU moguće je ispisati rezultat kernela.

Upravljanje memorijom

Upravljanje memorijom grana se na dvije bitne stavke - upravo spomenuto alociranje (dinamičke) memorije te kopiranje. Za obje stavke CUDA runtime pruža lako upotrebljive funkcije.

Što se tiče dinamički alocirane globalne memorije situacija je slična kao kod CPU. CUDA ima integrirane dvije funkcije `cudaMalloc()` te `cudaFree()` koje je moguće pozvati samo s domaćina.

Pogledajmo prototipe tih dviju funkcija kako bismo stekli dojam kako je minimalna

razlika od klasične alokacije memorije na CPU.

```
int *devPtr;
cudaError_t cudaMalloc(void **devPtr, size_t count);
...
cudaError_t cudaFree(void *devPtr);
```

Funkcija `cudaMalloc` alocira `count` bajtova globalne memorije (na uređaju) i sprema lokaciju te memorije u pokazivač devPtr, kojeg smo prethodno deklarirali baš kao što bi to napravili na CPU.

Iako imamo pokazivač na alociranu memoriju za niz, niz nema nikakve vrijednosti tj. nije inicijaliziran. Dva su načina za inicijalizaciju niza.

Prvi način je pozivanjem funkcije `cudaMemset()` kojoj osim pokazivača (`devPtr`) koji pokazuje na niz prosljeđujemo i inicijalnu vrijednost `value` koju će poprimiti svaki element niza. Treća vrijednost koju funkcija prima je broj bajtova koji zauzima niz na koji pokazuje pokazivač `devPtr`.

```
cudaError_t cudaMemset(void *devPtr, int value, size_t count);
```

Drugi način inicijalizacije niza je kopiranje niza (kojem su pridružene vrijednosti) s CPU. CUDA ponovno ima implementiranu odgovarajuću funkciju - `cudaMemcpy()`.

Poziv ove funkcije izgleda ovako:

```
cudaError_t cudaMemcpy(void *dst, const void *src, size_t count,
enum cudaMemcpyKind kind);
```

Ta funkcija kopira `count` bajtova s lokacije `src` na lokaciju `dst`. Ono što je jako bitno jest ispravno definirati vrstu prijenosa. U ovom slučaju kad želimo kopirati niz s CPU (host) na GPU (device) varijabla `kind` treba imati vrijednost `cudaMemcpyHostToDevice`.

Analogno tome, ukoliko kopiramo niz s GPU na CPU varijablu `kind` ćemo postaviti na `cudaMemcpyDeviceToHost`. Naime, kao što smo već spominjali i kod statički deklarirane globalne memorije, izvan kernela nije moguće direktno pristupiti nekom elementu polja alociranom na GPU (dakle nije moguće niti pisati niti čitati elemente niza). Ako izvan kernela želimo pristup elementima takvog niza potrebno je s `cudaMemcpyDeviceToHost` kopirati niz na CPU. Primjer iz prakse u kojem se to koristi je kad želimo rezultat izračuna kernela kopirati na CPU kako bi ga mogli jednostavno ispisati ili provjeriti.

Pogledajmo sada jednostavan primjer u kojem se koriste gore navedene funkcije i varijable. Sastoji se od dinamičke alokacije globalne memorije kopiranjem s CPU, korištenjem funkcije `cudaMemcpy()`, poziva jednostavnom kernelu koji inkrementira svaki element niza, te vraćanju rezultata na CPU kako bi se mogao ispisati, ponovno korištenjem funkcije `cudaMemcpy()` ali s drugim parametrima.

```
__global__ void kernel_example(dA, N)
{
    if( i < N) dA[i] += 1;
}

void main(
{
    //deklaracija, alokacija, inicjalizacija vektora na CPU
    int *A = (int*) malloc(N * sizeof(int));
    for(int i = 0; i < N; i++) A[i] = i;

    //deklaracija, alokacija vektora na GPU
    int *dA;
    DEVALLOC( dV, int, sizeof(int)*N);

    //inicjalizacija vektora na GPU (kopiranje host -> device)
    cudaMemcpy( dA, A, sizeof(int)*BR_V, cudaMemcpyHostToDevice );

    kernel_example<<blocksPerGrid, threadsPerBlock>>(dA, N);

    //kopiranje device -> host
    cudaMemcpy(A, dA, sizeof(int)*BR_V, cudaMemcpyDeviceToHost);

    //ispis rezultata kernela
    for(int i = 0; i < N; i++) printf("A[%d] = %d", i, A[i]);
}
```

Važno je paziti da pokazivači `src` i `dst` odgovaraju definiranoj vrsti prijenosa, jer u slučaju da pogriješimo ponašanje funkcije cudaMemcpy je nedefinirano.

Još su dvije vrijednosti koje gore spomenuta varijabla `kind` može imati: `cudaMemcpyHostToHost` te `cudaMemcpyDeviceToDevice`.

Funkcija `cudaMemcpy()` ne može biti korištena za kopiranje memorije s jednog GPU na drugi (u sustavima gdje imamo više GPU na raspolaganju).

Pregled svih memorija

Primjećujemo da CUDA na raspolaganje stavlja puno različitih memorija. Rezimirajmo osnovna svojstva svih spomenutih memorija tablicom s osnovnim podatcima:

MEMORIJA	NA ČIPU?	Čitanje - R Pisanje - W	Doseg (engl. scope)	Životni vijek (engl. lifetime)
Registri	Da	R/W	1 dretva	dretva
Lokalna	Ne	R/W	1 dretva	dretva
Dijeljena	Da	R/W	Sve dretve u bloku	blok
Globalna	Ne	R/W	Sve dretve + domaćin	domaćin
Konstantna	Ne	R	Sve dretve + domaćin	domaćin
Za teksture	Ne	R	Sve dretve + domaćin	domaćin

Slika 1.11: Vrste memorije na GPU i njihova svojstva..

1.6 Mjerenje vremena

Kako ćemo u nastavku rada uspoređivati brzine izvođenja algoritma na CPU i GPU sada ćemo uvesti funkcije potrebne za mjerenje vremena.

Mjerit ćemo koristeći funkcije na CPU (primjenjiv i na mjerenje vremena na GPU). Definirajmo funkciju `cpuSecond()` koja koristi funkcije definirane u `sys/time.h`.

```
double cpuSecond()
{
    struct timeval tp;
    gettimeofday(&tp,NULL);
    return ((double)tp.tv_sec + (double)tp.tv_usec*1.e-6);
}
```

Ako bismo sada pomoću ove funkcije htjeli mjeriti vrijeme izvođenja na CPU jednostavno prije funkcije (ili dijela koda) kojeg želimo mjeriti stavimo:

```
double iStart = cpuSecond();
```

Nakon tog koda ubacujemo:

```
double iElaps = cpuSecond() - iStart;
```

U varijablu `iElaps` spremljeno je izmjereno vrijeme - od prvog poziva `cpuSecond()` do idućeg poziva te funkcije.

Situacija s mjerenjem brzine izvršavanja koda na GPU je skoro ista, uz jednu nadopunu. Naime, kako se poziv kernela izvodi asinkrono moramo osigurati da prije nego zaustavimo mjerenje vremena sve dretve budu gotove odnosno da se sve kopije kernela izvrše.

Za to nam služi funkcija `cudaDeviceSynchronize()` koja se poziva s domaćina. Ona će osigurati da se ne izvrši kod nakon nje dok god ima aktivnih dretvi.

Konačno, mjerenje vremena izvršavanja svih kopija kernela odnosno koda na GPU izgledat će ovako:

```
double iStart = cpuSecond();
kernel_name<<<grid, block>>>(argument list);
cudaDeviceSynchronize();
double iElaps = cpuSecond() - iStart;
```

1.7 Asinkrono izvršavanje koda

Što znači netom spomenuto asinkrono izvršavanje koda? Ono se odnosi na način na koji CPU tretira pozive kernelu. Čim se kernel pozove GPU preuzima izvršavanje kernela, a CPU nastavlja s kodom koji se nalazi nakon tog poziva. Pogledajmo primjer kako bi bilo lakše shvatiti tijek izvršavanja.

```
kernel1<<<blocksPerGrid, threadsPerBlock>>>(argument list);
cpu_function1();
kernel2<<<blocksPerGrid, threadsPerBlock>>>(argument list);
cudaDeviceSynchronize();
cpu_function2();
kernel3<<blocksPerGrid, threadsPerBlock>>>(argument list);
...
```

Odmah nakon poziva prvog kernela, CPU je krenuo s izvršavanjem prve funkcije, bez obzira što paralelno s tim GPU izvršava prvi kernel. Nakon što je izvršio prvu funkciju CPU poziva drugi kernel. Ako je prvi kernel gotov GPU će pokrenuti izvršavanje drugog kernela. Ali, ako se prvi kernel još uvijek izvršava, GPU će staviti drugi kernel u red

kernela koji čekaju izvršavanje i neće ga pokrenuti dok god ne završi s prvim. CPU će svejedno nastaviti s svojim kodom, bez obzira na to što se događa na GPU. Ali, nakon drugog kernela, pozivom funkcije `cudaDeviceSynchronize()` CPU je dobio naredbu da čeka. Mora čekati da se svi dotad pozvani kerneli izvrše - i prvi, i drugi. Tek nakon što sve dretve završe s radom, CPU poziva drugu funkciju i nastavlja s izvršavanjem. Nakon izvršavanja cijele druge CPU funkcije poziva se i treći kernel koji nastavlja s radom paralelno s bilo čim što se izvršava na CPU nakon poziva tog kernela.

Iz ovog primjera vidljivo je da se sa strane CPU-a dva uzastopna poziva kernelu tretiraju asinkrono - ne čeka se završetak prvog za poziv drugog, ali sa strane GPU-a dva uzastopna poziva kernelu (osim ako nije drugačije naznačeno) ne izvršavaju se asinkrono nego sinkrono - drugi će čekati da sve dretve pozvane prvim kernelom završe s radom. Ali, CPU rutine neće čekati završetak kernela pozvanih prije njih i upravo to je razlog za korištenje `cudaDeviceSynchronize()` kako bi se sinkronizirao rad GPU-a s rutinama koje slijede na CPU.

Zaključujemo da prilikom pisanja koda koristeći CUDA tehnologiju treba razmišljati istovremeno s gledišta CPU-a i s gledišta GPU-a jer su to dvije odvojene jedinice koje imaju drugačiju logiku rada.

Poglavlje 2

Implementacija Dijkstrinog algoritma na GPU

Grafovi su struktura podataka široko rasprostranjena u znanosti i inženjerstvu. Shodno tome, osnovne operacije odnosno algoritmi na grafovima kao što su pretraga u dubinu, širinu ili izračun najkraćeg puta također su jako često korišteni. U praktičnim primjenama (npr. socijalne mreže) često se javljaju vrlo veliki grafovi. Ovo znači da je klasični CPU pristup nespretan za korištenje jer često bez obzira na maksimalne optimizacije algoritama i dalje prisiljava znanstvenike i inženjere da na neke izračune čekaju satima, često i danima, dok računala završe s radom.

2.1 Grafovi

Podsjetimo se ukratko pojmova iz teorije grafova.

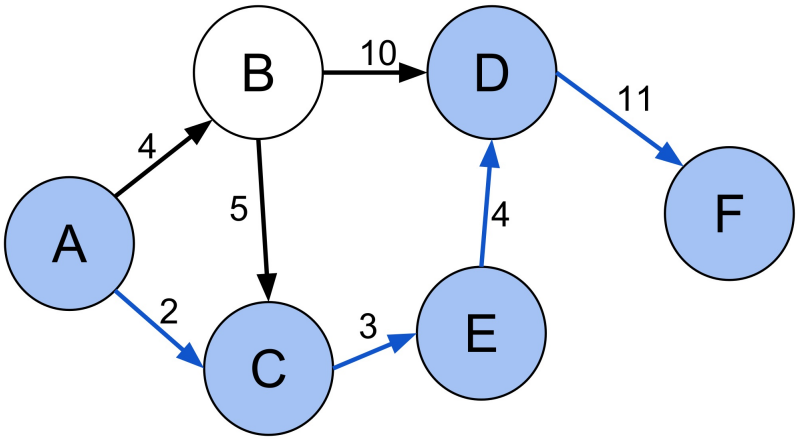
Formalno, **graf** definiramo kao uređeni par (V, E), pri čemu je V skup **vrhova**, a E podskup skupa svih dvočlanih podskupova od V, koje zovemo *bridovi*. Za vrhove $x, y \in V$ kažemo da su susjedni ako je $\{x, y\} \in E$.

Usmjereni graf ili digraf je uređeni par (V, E) pri čemu je V skup **vrhova**, a E skup uređenih parova elemenata skupa V, koje nazivamo usmjereni bridovi ili **lukovi**. Vrhovi $x, y \in V$ su susjedni ako je $(x, y) \in V$. Kažemo da je orijentacija luka $l = (x, y)$ od vrha x prema vrhu y, te da je x početni vrh, a y krajnji vrh tog luka. Kod luka (x, y) vrh x nazivamo prethodnikom vrha y, a vrh y sljedbenikom vrha x.

Sada ćemo definirati neke načine obilaska vrhova grafa. Pretpostavimo da između vrhova x i y postoji samo jedan brid, $\{x, y\}$ odnosno (x, y).

Šetnja u grafu je niz
$(v_0, v_1, v_2, \ldots, v_n)$, pri čemu brid e_i spaja vrhove v_{i-1} i v_i, za $i = 1, \ldots, n$. Kažemo da je to šetnja od v_0 do v_n. Jedna od specijalnih vrsta šetnji je *put* - šetnja u kojoj su svi vrhovi različiti (osim eventualno prvog i zadnjeg). Analogno definiramo put za usmjeren graf, $e_i = (v_{i-1}, v_i)$ je luk za za $i = 1, \ldots, n$ u gornjoj definiciji.

Pogledajmo prikaz jednog grafa (usmjerenog) gdje su plavo označeni vrhovi i bridovi koji čine jedan od putova u tom grafu.



Slika 2.1: Put u usmjerenom grafu.

Podgraf grafa $G = (V, E)$ je graf kojemu su skup vrhova i skup bridova podskupovi od V i E, redom.

Definirajmo relaciju ekvivalencije $\equiv$ na skupu vrhova V grafa G: $x \equiv y$ ako postoji put od x do y.

Ova relacija ekvivalencije definira jednu particiju skupa V. Sada definiramo komponente povezanosti (ili kraće, komponente) grafa kao podgrafove inducirane klasama ekvivalencije.

Kažemo da je graf **povezan** ako postoji samo jedna komponenta.

Drugim riječima, graf je povezan ako postoji put između svaka dva vrha u grafu.

Stupanj (ili valencija) vrha x grafa G definira se kao broj bridova grafa G koji sadrže vrh x. Za usmjerene grafove stupanj vrha x je broj lukova grafa G za koje je x početni vrh.

Katkada će naši grafovi sadržavati i dodatne informacije. Formalno takve grafove opisujemo uz pomoć težinskih funkcija. Težinska funkcija na skupu X je funkcija $t : X \to \mathbb{R}$ (ili češće, $t : X \to \mathbb{R}_0^+$). Vršno–težinski, te bridno–težinski graf je graf s težinskom funkcijom na skupu vrhova, odnosno bridova, redom. Bridno–težinski grafovi su mnogo češći u primjenama. U daljnjem tekstu će se pojam težinski graf odnositi se na bridno-težinski.

Usmjeren težinski graf često se naziva **mreža**. Sada primjećujemo da je na gornjoj Slici 2.1 prikazan jedan težinski usmjeren graf, odnosno mreža.

Dakle, mreža $M = (V, E, \delta)$ je usmjereni graf $G = (V, E)$ (gdje je $|V| = n$, $|E| = e$) na čijem je skupu bridova definirana realna funkcija $\delta : E \to \mathbb{R}$.

U tako definiranoj mreži duljina puta $w = v_1, \ldots, v_m$ definirana je kao:

$$l(w) = \sum_{i=1}^{m-1} \delta(v_i, v_{i+1}).$$

Definirajmo sada za taj isti graf odnosno mrežu najkraći put od vrha x do vrha y ($x, y \in V$):

$$Min(x, y) = inf\{l(w) : w \text{ put od } x \text{ do } y\}$$

2.2 Problem najkraćeg puta u grafu (engl. SSSP problem)

Kao što smo već spomenuli jedan od čestih zahtjeva u praktičnim primjenama je izračunati najkraći put između dva vrha u grafu. U engleskoj literaturi općenitiji problem od ovoga, izračun najkraćih putova od jednog do svih ostalih vrhova u grafu, naziva se "single source shortest path" ili SSSP kako stoji i u naslovu. Formalno ga definiramo ovako (koristeći gore definiranu funkciju Min):

Neka je $N = (V, E, \delta)$ usmjereni graf (mreža) i neka je s neki element iz V (vrh). Izračunaj $Min(s, x)$ za svaki $x \in V$.

2.3 SSSP problem u praksi

Sada kada je potpuno jasno što je graf te što je formalno SSSP problem pokazat ćemo nekoliko primjena iz (više ili manje) svakodnevnog života u kojima se zahtjeva efikasno rješenje

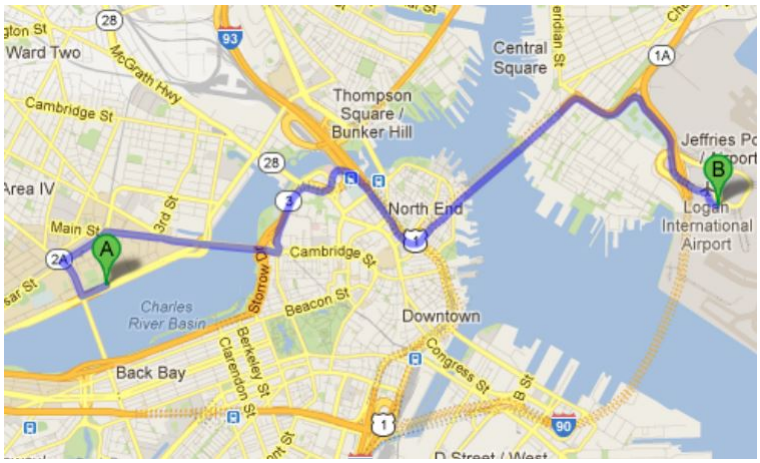
SSSP problema što će dati motivaciju za razvoj algoritama koji rješavaju taj problem, ali i optimizaciju istih.

Primjena - Navigacija po kartama

Danas, kada je zahvaljujući pametnim telefonima navigacija dostupna svakome tko ih posjeduje, sve više je ljudi koji u svakodnevnome životu koriste digitalne karte i prepuštaju svojim pametnim telefonima da im računaju putove kojima će se kretati.

Ako mapu zamislimo kao graf gdje su vrhovi točke na karti (mjesta), bridovi su ceste koje povezuju dva susjedna mjesta, a težine duljine tih cesta, SSSP problem je određivanje kako najkraćim mogućim putem doći od mjesta u kojem se netko trenutno nalazi do nekog drugog mjesta na karti. Ukoliko bi uključili komponentu javnog prijevoza ili osobnih automobila onda kao težine bridova (putova) nije dovoljno uzeti udaljenosti. Tada kao težinu uzimamo vrijeme koje nam treba za prijeći taj put, što je i dalje isti SSSP problem samo s drugom težinom na lukovima. Istina, postoji mala razlika s gore navedenom formalnom definicijom. U ovakvim primjenama uglavnom će nas zanimati samo najkraći put između točno dva vrha, ali to nije ništa drugo nego podskup rješenja SSSP problema jer ne gledamo rješenje za sve nego samo za jedan vrh.

Pogledajmo primjer jedne karte na Slici 2.2. Pojedini segmenti cesta predstavljaju bridove, a vrhovi su mjesta na kojima se sijeku dvije ili više cesta. Plavom bojom je naznačen najkraći put od točke (vrha) A do točke B.



Slika 2.2: Aplikacija Google Maps i prikaz najkraćeg puta između dvije točke na karti.

Zaključak je, da bi sve te pametne aplikacije dobro i efikasno radile potrebno je implementirati dobar i efikasan algoritam koji će računati najkraće putove u grafovima.

Primjena - Protokoli za usmjeravanje (engl. routing protocols)

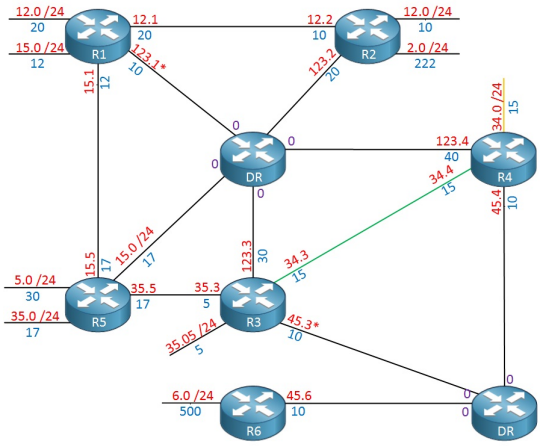
Ova primjena izračuna najkraćeg puta događa se u pozadini svakog našeg pristupa internetu, a da toga možda nismo niti svjesni.

Usmjernici (engl. routeri) povezuju mreže koristeći internet protokol (engl. IP protocol). OSPF (Open Shortest Path First) je protokol za usmjeravanja razvijen za IP mreže, koji izračunava najkraći put za pakete koji putuju kroz različite mreže. Metrika OSPF-a (težina brida) računa se po formuli:

$$C = \frac{10^3}{\text{Pojasna širina (bit/s)}}$$

Iz formule je očigledno da je cijena puta obrnuto proporcionalna pojasnoj širini neke veze. Dakle, veza 100 Mb/s ima veću cijenu, nego veza 1 Gb/s, a paket će biti usmjeren na put s manjom cijenom.

Ako pogledamo Sliku 2.3 jasno je kako usmjernici i promet na mreži čine graf, gdje su usmjernici vrhovi, bridovi spajaju usmjernike koji mogu komunicirati direktno jedan s drugim, a težine tih bridova izračunate prema gore navedenoj formuli (upisane plavom bojom).



Slika 2.3: Primjer mreže usmjernika.

U ovom slučaju, algoritam koji računa najkraći put pomaže u svakodnevnom životu na način da ubrzava svaku aktivnost na mreži.

2.4 Ideja algoritma

Edsger W. Dijkstra, danski matematičar i programer, 1959. godine riješio je spomenuti problem SSSP-a, i objavio ga u članku "A note on two problems in connexion with graphs". Ovaj algoritam, koji je po njemu dobio ime Dijkstrin algoritam, u osnovnoj verziji radi u složenosti $O(V^2)$ gdje je V broj vrhova u grafu, a uz optimizaciju i poboljšanja može se postići složenost $O((E + V) \times \log(V))$ gdje je E broj bridova odnosno lukova u grafu.

Ideja algoritma je sljedeća:

- U svakom koraku algoritma skup vrhova je podijeljen u dva disjunktna skupa: posjećenih i neposjećenih vrhova.
- Posjećeni vrhovi su oni vrhovi za koje je već određen najkraći put od početnog vrha, neposjećeni su svi ostali vrhovi. Na početku algoritma, početni vrh je jedini posjećen.
- Svaki vrh u svakom trenutku ima pridijeljenu trenutnu vrijednost duljine najkraćeg puta od početnog vrha do samog sebe (za posjećene vrhove ova je vrijednost ujedno i konačna vrijednost, za ostale je podložna promjenama, ovisno o sljedećim koracima).
- U svakom koraku jedan vrh (označimo ga s f) postaje posjećen i to onaj vrh koji ima najmanju trenutnu vrijednost u skupu neposjećenih vrhova.
- U sljedećem koraku za sve za sve neposjećene susjede vrha f ponovno se računa (osvježava se) njihova trenutna vrijednost.
- Algoritam završava u trenutku kad više nema neposjećenih vrhova odnosno kad su vrijednosti duljina najkraćih putova konačne za sve vrhove u grafu.

Prilikom izvođenja algoritma, ako je potrebno, u dodatni niz može se pohraniti redoslijed posjećivanja vrhova na način da za svaki vrh pohranimo prethodnika tog vrha u najkraćem putu.

Ono što već iz same ideje vidimo jest da je Dijsktrin algoritam primjer jednog pohlepnog (engl. *g*reedy) algoritma zato što u svakom koraku biramo trenutno najbliži vrh kao

vrh na najkraćem putu (lokalno rješenje smatramo ujedno i globalnim rješenjem). U ovom slučaju pohlepan algoritam funkcionira i uvijek daje točno rješenje.

2.5 Pseudokod algoritma

U prošlom smo odjeljku naveli ideju algoritma. Kako bi ga mogli dodatno analizirati, odnosno dokazati da ovaj algoritam stvarno daje točno rješenje, treba navesti pseudokod algoritma.

Algoritam 1: Dijkstrin algoritam na CPU (sekvencijalni).

```
1  za svaki vrh i grafa G čini
2      C[i] ← ∞;
3      U[i] ← true;
4  C[0] ← 0;
5  U[0] ← false;
6  f ← 0;
7  mssp ← 0;
8  dok mssp ≠ ∞ čini
9      za svaki neposjećeni vrh j susjedan vrhu f čini
10         C[j] ← Min(C[j], C[f] + W[f, j]);
11     mssp ← ∞;
12     za svaki neposjećeni vrh j čini
13         ako C[j] < mssp onda
14             mssp ← C[j];
15             f ← j;
16     U[f] ← false
```

Prisjetimo se ideje algoritma. Spomenuli smo skup posjećenih i neposjećenih vrhova koji su međusobno disjunktni. Analogno, u pseudokodu postoji niz U koji za posjećene vrhove na odgovarajućem indeksu ima pohranjenu vrijednost *true*, a za neposjećene *false*. U nizu C pohranjene su trenutne najmanje vrijednosti duljine puta do svakog vrha, što znači da će taj niz ujedno predstavljati konačno rješenje nakon što algoritam završi s radom. U pseudokodu s f je označen onaj vrh kojeg u svakom koraku proglasimo kao najmanje udaljenog, birajući od neposjećenih vrhova te ga idućeg posjećujemo. $W[f, j]$ označava

težinu brida između vrhova f i j.

Kao što vidimo, glavni dio algoritma (nakon početne inicijalizacije) sastoji se od petlje. Tijelo te petlje možemo logički podijeliti na 3 dijela, u zagradama su navedene linije koda u algoritmu.

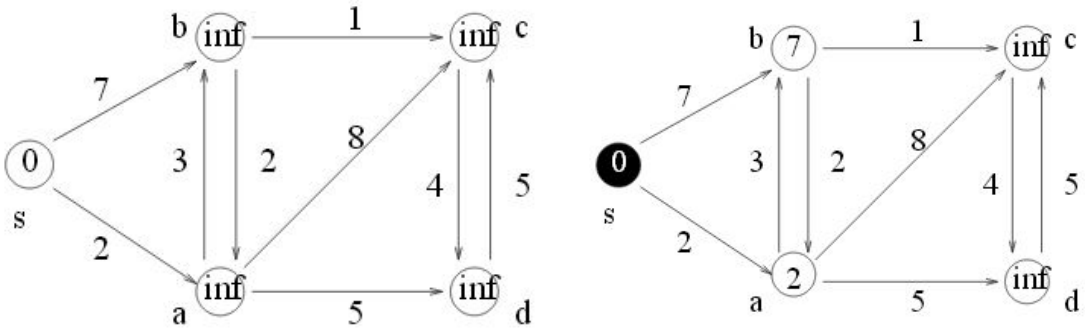
Relax (9-10) U ovom koraku ponovno se izračunavaju sve vrijednosti ne posjećenih vrhova (koji su susjedni izabranom vrhu f), uzevši u obzir vrijednost vrha f koji je u tom trenutku "vodeći" vrh.

Minimum (12-15) U ovom koraku tražimo najmanju vrijednost vrha (duljina najkraćeg puta od ishodišta) među neposjećenim vrhovima.

Update (16) Onaj vrh koji zadovoljava prethodno svojstvo postaje novi vodeći vrh, dodajemo ga u posjećene vrhove.

Kako bismo dodatno pojasnili algoritam pogledajmo jednostavan primjer grafa te korake u kojima Dijkstrin algoritam računa najkraći put do svakog vrha tog grafa.

Početni vrh označen je slovom s, a ostali vrhovi slovima a, b, c, d. Težina svakog luka upisana je pokraj luka, a trenutna duljina najkraćeg puta od s do određenog vrha upisana je u unutar tog vrha. Vrhovi za koji se završi računanje i uklone se iz skupa U obojani su crnom bojom.

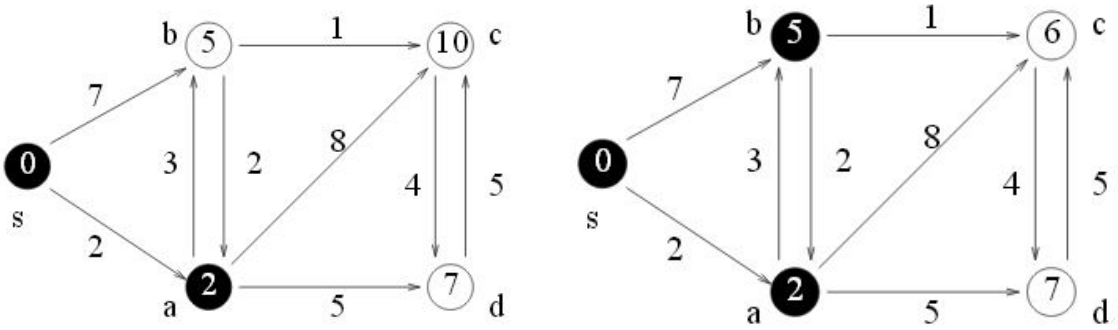


(a) (b)

Slika 2.4: Inicijalizacija Dijkstrinog algoritma (lijevo), te stanje nakon što vrh s postane posjećen (desno).

Na Slici 2.4a vidimo stanje nakon inicijalizacije gdje je vrijednost vrha s postavljena na 0, a vrijednosti svih ostalih vrhova na beskonačno. Trenutno je $U = \{s, a, b, c, d\} = V$.

Kako je s vrh s najmanjom vrijednošću od svih vrhova u U, tako vrh s uklanjamo iz U i postaje f. Na Slici 2.4b vidimo da je s zato obojan crno. Sada mijenjamo vrijednosti svih vrhova susjednih vrhu s, koji su u U. To su u ovom slučaju vrhovi a i b. Njihova vrijednost postaje težina luka koji vodi od s jer je ta težina manja od inf. Primjerice za vrh a, $C[a] = \infty > C[s] + W[s, a] = 2$.

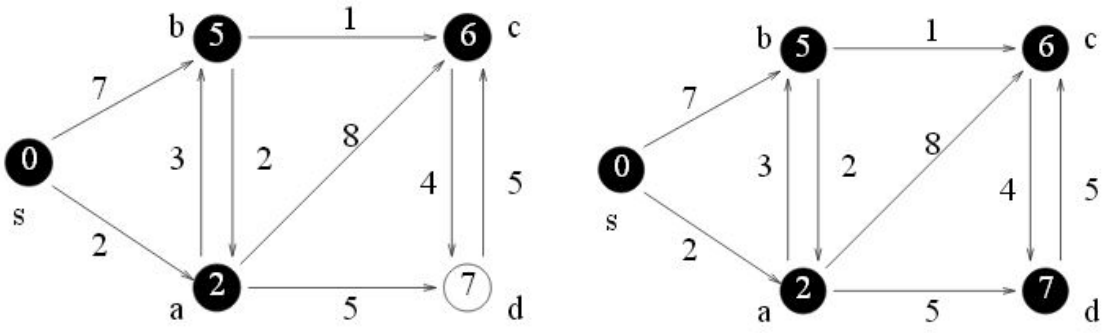


(a) (b)

Slika 2.5: Situacija nakon posjećivanja vrha a (lijevo), te vrha b (desno).

Od preostalih vrhova u $U = \{a, b, c, d\}$ najmanju vrijednost ima vrh a. Zato je na Slici 2.5a a označen crnom bojom, i on postaje vodeći vrh (f) u narednom koraku i uklanjamo ga iz U. Na istoj slici vidimo da su zato promijenjene vrijednosti svih susjednih vrhova od a (koji su u U), a to su vrhovi b, c i d. Primjerice vrhu b promijenila se vrijednost iz 7 u 5 jer je $C[b] = 7 > C[a] + W[a, b] = 2 + 3 = 5$. Ostali vrhovi analogno.

Kako je sada vrh b najmanje vrijednosti u skupu $U = \{b, c, d\}$, tako b postaje vodeći vrh, postaje vodeći vrh f te ga uklanjamo iz U. Na Slici 2.5b također vidimo da taj vrh postaje crne boje. Mijenjamo vrijednosti vrhova c i d u odnosu na b. Primjerice c mijenja vrijednost iz 10 u 6 jer je $C[c] = 10 > C[b] + W[b, c] = 5 + 1 = 6$.



Slika 2.6: Situacija nakon posjećivanja vrha c (lijevo), te na kraju algoritma (desno).

U preostalom skupu $U = \{c, d\}$ najmanju vrijednost ima vrh c pa ga uklanjamo iz U, on postaje vodeći vrh f, a na Slici 2.6a vidimo da je i obojan crno iz tog razloga. Jedini susjed od c koji je još uvijek u U je vrh d, a on je ujedno i jedini preostali vrh u U. Provjeravamo njegovu vrijednost s obzirom na vrijednost od c, ali nema promjena jer je $C[d] = 7 < C[c] + W[c, d] = 6 + 4 = 10$.

Sada uklanjamo d iz U jer je vrh najmanje vrijednosti u U, a ujedno i jedini preostali. U ostaje prazan što znači da je algoritam gotov.

2.6 Dokaz točnosti

Iako na primjeru vidimo da algoritam dobro rješava zadani problem, želimo li to generalizirati i reći da taj algoritam uvijek dobro radi, potreban je matematički dokaz.

Teorem 2.6.1. *U svakom koraku algoritma za svaki čvor $x \in S$ gdje je $S = V \backslash U$ vrijedi: $C[x]$ je duljina najkraćeg puta od s do x.*

Dokaz. Dokaz provodimo indukcijom po broju vrhova u skupu S. Neka je s početni vrh. Neka je $T(v)$ vrijednost puta (definiranog algoritmom) od vrha s do v koja može i ne mora biti minimalna.

Baza indukcije je sam početak algoritma gdje je $S = s$. U tom slučaju tvrdnja očito vrijedi jer je $C[s] = 0$. Pretpostavimo da tvrdnja vrijedi kada S ima $k \geq 1$ čvorova. Pokazat ćemo da tvrdnja vrijedi i za skup kojem je dodan još jedan čvor, označimo ga s x.

Neka je P_x najkraći put od s do x. Neka je luk (u, x) posljednji luk na putu P_x. Kako je $u \in S$, po indukcijskoj je pretpostavci $C[u]$ duljina najkraćeg puta od s do u.

Želimo dokazati da je P_x najkraći put od s do x u grafu. Neka je P neki drugi put u grafu od s do x. Cilj je dokazati da duljina puta P nije manja od dužine puta P_x.

Kako je čvor x izvan S, put P mora negdje izaći iz skupa S. Neka je y prvi čvor na putu P koji nije u S, a neka je čvor $v \in S$ neposredno prije y na putu P.

Označimo s P' dio puta P od s do v. Kako je čvor $v \in S$, po indukcijskoj pretpostavci duljina puta P' nije manja od duljine najkraćeg puta od s do v, odnosno vrijedi $l(P') \geq C[v]$. Iz ovoga slijedi da za put s do y vrijedi $l(P') + \delta(v, y) \geq C(v) + \delta(v, y) \geq T(y)$. Kako je u Dijkstrinom algoritmu čvor x izabran kako bi bio dodan skupu S, a $y \notin S$ vrijedi $T(y) \geq T(x) = C[x] = l(P_x)$. Zbog pretpostavljene nenegativnosti težina lukova u Dijkstrinom algoritmu, vrijedi da duljina cijelog puta P nije manja od dijela puta P od s do y.

Slijedi da je $l(P) \geq l(P') + \delta(v, y) \geq T(y) \geq T(x) = C[x] = l(P_x)$. Ovime je tvrdnja dokazana pomoću indukcije. □

2.7 Paralelizacija algoritma

U opisanom algoritmu moglo se dogoditi da možemo birati između više vodećih vrhova f, a kako je algoritam sekvencijalan odabrali smo samo jednog kandidata u svakom koraku. Iz ovoga slijedi da za svaki od tih kandidata ukoliko ih je više treba dodatni korak umjesto da ih sve obradimo u istom koraku (paralelno). Polazišna točka za paralelizaciju je stoga istovremeno procesiranje svih kandidata za vodeći vrh.

Paralelni algoritam, dakle, omogućava da imamo skup vodećih vrhova, označimo ga, analogno, s $F \subseteq V$. Ovaj skup možemo obrađivati odjednom jer vrijedi:

1. Trenutno izračunata vrijednost za sve vodeće vrhove podudara se s duljinom najkraćeg puta do tih vrhova.

2. Kada računamo novu vrijednost nekog vrha $j \in U$ (`relax` dio), najkraći put do tog vrha ne može prolaziti kroz više vrhova iz skupa $V \backslash U$ (u ovom slučaju više vrhova iz skupa F). Zato ćemo prilikom biranja prethodnika (vrha koji u najkraćem putu od s do j dolazi odmah prije vrha j), prirodno, izabrati onog za kojeg se postiže $min_{f \in F} \{C[f] + W[f, j]\}$.

Sada kada je iznijeta osnovna ideja paralelnog algoritma te ključna promjena u odnosu na sekvencijalni algoritam pogledajmo još kako i koliko ova ideja mijenja implementaciju algoritma.

Algoritam 2: Dijkstrin algoritam na GPU (paralelni).

```
1 initialize(C, F, U);
2 mssp ← 0;
3 dok mssp! = ∞ čini
4     relax(C, F, U);
5     mssp ← minimum(C, U);
6     update(C, F, U, mssp);
```

Osnovni dijelovi algoritma su isti, samo se način na koji se izvršavaju ipak malo razlikuje. Kao što vidimo algoritam se sastoji od 3 glavne procedure:

1. `relax`(C, F, U) - izračunava trenutni najkraći put za sve U-vrhove do kojih postoji brid iz nekog F-vrha. Dakle, mora izračunati $C[j] = min\{C[j], C[f] + W[f, j]\}$ za svaki par vrhova $f \in F, j \in U$.

2. `minimum`(C, U) - pronalazi najmanju duljinu puta među svim neposjećenim vrhovima. Tu duljinu spremamo u varijablu $mssp$.

3. `update`$(C, F, U, mssp)$ - uklanja iz U sve one vrhove čija je trenutna vrijednost najkraćeg puta jednaka vrijednosti $mssp$, i time ujedno stvara novi skup F.

Procedura initialize

Procedura `initialize(C, F, U)` gotovo je identična kao u sekvencijalnoj verziji:

```
1 initialize (C, F, U)
2     za svaki vrh i u grafu G čini
3         C[i] ← ∞;
4         F[i] ← false;
5         U[i] ← true;
6     C[0] ← 0;
7     F[0] ← true;
8     U[0] ← false;
```

Procedura relax

Procedura `relax`(C, F, U) bi u paralelnoj verziji (kada se skup F sastoji od više vrhova) bila posve analogna onoj iz sekvencijalne (kada skup F ima samo jedan vrh) da nema jedne inkonzistentnosti koja se može pojaviti u nekim slučajevima.

```
1 relax_krivo (C, F, U)
2     za svaki vrh f za koji je F[f] = true paralelno čini
3         za svaki vrh j sljedbenik od f za koji je U[j] = true čini
4             C[j] ← min(C[j], C[f] + W[f, j]);
```

Upravo je ovaj algoritam korišten u članku [5] i vjerovalo se da je ispravan, dok autori članka [6] nisu pokazali zašto nije ispravan te kako napisati ispravan algoritam.

Promotrimo dio s računanjem vrijednosti za C u gornjem kodu. Ukoliko dva vrha iz F pristupaju istom vrhu iz U (paralelno) i izvršavaju spomenuti izračun $C[j] = min\{C[j], C[f]+W[f, j]\}$ može se dogoditi da vrijednost koja ostane upisana bude veća vrijednost, odnosno, kriva vrijednost. Nemamo utjecaja na to kojim će redoslijedom odnosno u kojem trenutku paralelne dretve pristupiti podatku, a ne možemo sinkronizirati na klasičan način jer dretve nisu nužno unutar istog bloka. Dakle, potrebna je promjena pristupa.

Postoje dva načina za rješenje problema. Prvi je korištenjem ugrađene funkcije koju pruža CUDA, o kojoj će biti riječi kasnije kod same implementacije.

Drugi pristup uključuje ispitivanje prethodnika svakog vrha (umjesto sljedbenika) i traženje onih prethodnika koji su u F. Za takve vrhove računa se nova vrijednost prema istoj formuli. U ovom slučaju ne može doći do spomenute kolizije jer se svaki vrh brine sam za sebe, odnosno, vrijednost $C[i]$ će moći promijeniti samo vrh i, za razliku od prvotne formulacije u kojoj su to mogli napraviti svi njegovi prethodnici koji su u skupu F. Pogledajmo kako izgleda ova verzija u pseudokodu:

```
1 relax (C, F, U)
2     za svaki vrh i za koji je U[i] = true paralelno čini
3         za svaki vrh f prethodnik od i za koji je F[f] = true čini
4             C[i] ← min(C[i], C[f] + W[f, i]);
```

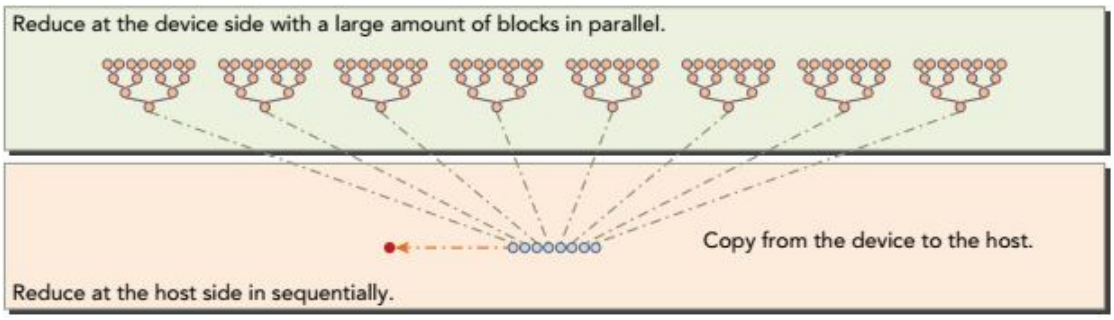
Ipak, radi jednostavnosti generiranja samog grafa (ne moramo generirani graf modificirati u varijantu koja memorira prethodnike vrhova umjesto sljedbenika), izabrati ćemo prvi pristup.

Od tri navedene procedure, `minimum()` je najteža za paralelizaciju iz razloga što je po prirodi sekvencijalna (tražimo minimalnu vrijednost u nekom nizu brojeva). Ovaj se algoritam u literaturi naziva redukcijom te se osim za nalaženje najmanjeg elementa u vektoru ista metoda primjenjuje i za zbrajanje svih vrijednosti u vektoru, a onda i za računanje skalarnog produkta dvaju vektora.

Komplikacija prilikom implementacije redukcije je činjenica da je nemoguće sinkronizirati sve dretve međusobno (sjetimo se, moguća je sinkronizacija samo unutar bloka). Zato ćemo sinkronizaciju izvesti na sljedeći način:

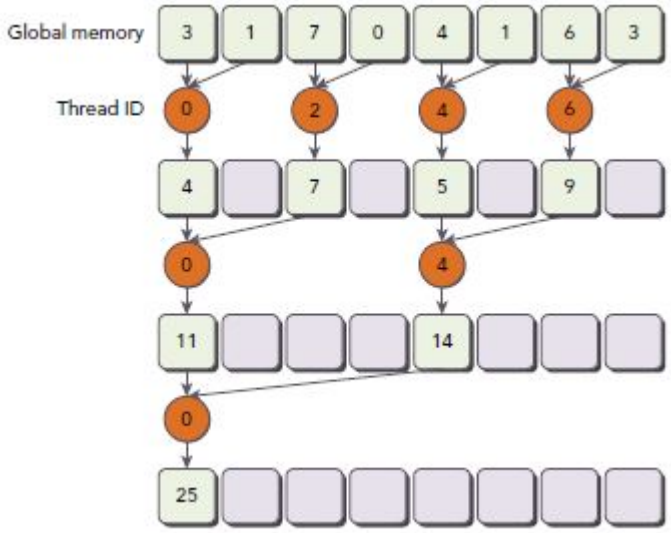
1. Podijelit ćemo vektor u manje dijelove - blokove.
2. Sinkronizirat ćemo dretve na razini bloka, odnosno izračunati minimum na razini bloka (parcijalni rezultat: minimum svih elemenata koji su u danom bloku).
3. Izračunati minimum (na CPU) svih parcijalnih rezultata.

Iako nismo postigli sinkronizaciju na cijelom setu podataka ipak smo dobili djelomičnu paralelizaciju i ubrzanje.



Slika 2.7: Redukcija.

Na Slici 2.7 također vidimo osnovnu ideju redukcije, dok Slika 2.8 pokazuje korake na razini bloka, u ovom primjeru za zbrajanje (sasvim analogno će ići i minimum).



Slika 2.8: Redukcija - koraci na razini bloka.

Na razini bloka, zbrajanje se izvršava kroz nekoliko koraka, koji su međusobno sinkronizirani, odnosno, sve dretve prelaze u idući korak istovremeno. Time se postiže da neka dretva neće čitati vrijednost zbroja iz prethodnog koraka prije nego su svi zbrojevi iz prethodnog koraka izračunati i upisani na odgovarajuća mjesta. Vrijednosti se zbrajaju u parovima, po jedna dretva zbraja vrijednost s dva indeksa niza te zbroj sprema u isti niz. U svakom koraku dvostruko se povećava razmak između dva indeksa niza čije vrijednosti jedna dretva zbraja. Broj dretvi koje računaju u svakom koraku postaje dvostruko manji.

Kao što vidimo na gornjem primjeru, u početnom koraku razmak među indeksima niza je 1, i 4 dretve zbrajaju vrijednosti (8 je elemenata niza, a svaka dretva zbraja dvije vrijednosti). Izračunate vrijednosti spremljene su na svaki drugi indeks niza. Iz tog razloga je u sljedećem koraku razmak među indeksima 2 i potrebne su dvije dretve kako bi zbrojile ta 4 rezultata prethodnog koraka. Dretve spremaju svoje rezultate na svaki četvrti indeks niza. U zadnjem koraku jedna dretva zbraja 2 preostale vrijednosti i sprema ih na indeks 0 u nizu. Sada kada su svi koraci gotovi ta će dretva ujedno spremiti tu konačnu vrijednost bloka u globalni niz. Ovakav isti proces napraviti će svaki blok. Blokovi rade neovisno jedni o drugima, svaki svom dijelu jednog većeg niza.

Ovo je osnovna ideja koja se može još dosta optimizirati, ali princip uvijek ostaje isti. U implementaciji algoritma koja slijedi koristit će se optimizirana verzija redukcije.

Procedura update

Za razliku od prethodnih, procedura `update(C, F, U, mssp)` jednostavna je kao i u sekvencijalnom algoritmu, jer se može izvesti neovisno i paralelno za svaki vrh grafa.

```
1 update (C, F, U, mssp)
2     za svaki vrh i paralelno čini
3         F[i] ← false;
4         ako C[i] = mssp onda
5             U[i] ← false;
6             F[i] ← true;
```

2.8 Implementacija

Algoritme pokrećemo na težinskim usmjerenim grafovima takvim da postoji put od početnog vrha do svakog drugog vrha grafa. Težine lukova su pozitivne jer Dijkstrin algoritam inače ne bi ispravno radio.

Struktura podataka

Grafovi se uobičajeno u algoritmima predstavljaju kao **matrice susjedstva**. Neka je $G(V, E, C)$ graf koji zadovoljava postavljene uvjete. Neka je $n = |V|$. Tada je matrica susjedstva grafa G kvadratna matrica A dimenzija $n \times n$, zadana formulom po članovima:

$$A_{ij} = \begin{cases} W[i, j], & , e_{ij} \in E, \\ 0 & , \text{inače.} \end{cases}$$

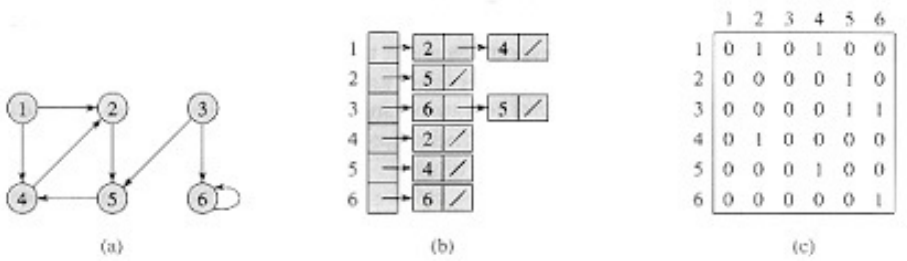
U praksi se često događa da je ovakav izbor prikaza grafa nije efikasan jer su ove matrice uglavnom rijetko popunjene (imaju jako puno nula), a zauzimaju puno prostora: ako su težine cijeli brojevi (dužine 4 bytea), onda za pohranu matrice susjedstva reda n treba $4n^2$ byteova bez obzira na to koliko bridova ima graf.

Zato je alternativa korištenje **lista susjedstva**. Kod ovakvog prikaza za svaki čvor v grafa se pohranjuje lista u kojoj se nalaze svi čvorovi u takvi da $e_{vu} \in E$, odnosno, postoji luk od v do u. Dakle, lista susjedstva se sastoji od n lista vrhova. U slučaju težinskog grafa

neće biti dovoljna samo oznaka susjednog čvora nego i odgovarajuća težina luka. To se lako rješava korištenjem strukture koja ima oznaku čvora i težinu luka od prethodnog do tog čvora. Iako bi nas oznaka luka mogla navesti da mislimo da ta struktura predstavlja čvor, bilo bi ispravnije reći da ta struktura predstavlja jedan luk.

Liste susjedstva osobito su pogodne za usmjerene grafove jer kod neusmjerenih nastaje redundancija iz razloga što brid između vrha u i v zapisujemo dva puta - u zapisujemo u listu od v te v zapisujemo u listu od u.

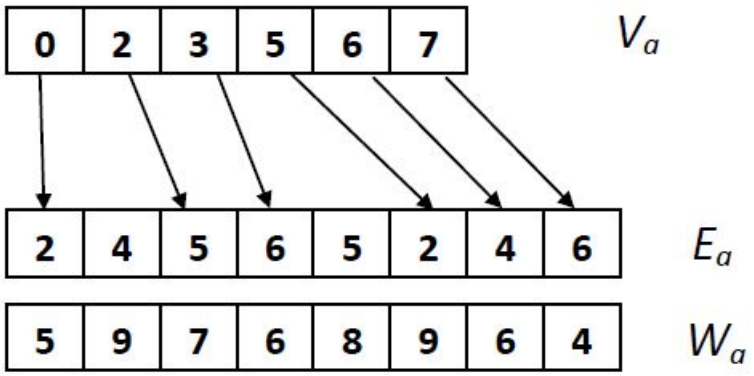
Na Slici 2.9 imamo primjer matrice i liste susjedstva za jedan usmjereni graf (ovo je primjer bez težina, radi jednostavnosti).



Slika 2.9: Primjer matrice i liste susjedstva za prikazani graf.

Često korišten način implementacije liste susjedstva jest kompaktni prikaz gdje je cijela lista susjedstva smještena u jedan (veliki) niz, uz dodatnu listu koja identificira vrhove i pokazuje na taj niz.

Vrhovi grafa predstavljeni su nizom V_a, koji ima n elemenata, a lukovi nizom E_a koji ima $|E| = e$ elemenata. Indeks i u nizu V_a odgovara i-tom vrhu grafa. Svaki element u nizu V_a (vrijednost na određenom indeksu) predstavlja indeks (pokazivač) nekog elementa u nizu E_a. To je pokazivač na listu susjednih čvorova (sjetimo se upravo objašnjene strukture lista susjedstva). Broj čvorova koji pripadaju vrhu i određen je vrijednošću na koju pokazuje $V_a[i+1]$. Vrhu i susjedni su svi oni čvorovi koji se nalaze u nizu E_a na indeksima od $E_a[V_a[i]]$ do $E_a[V_a[i+1]-1]$. Ukoliko se radi o težinskom grafu imat ćemo još jedan dodatan niz W_a koji je također duljine $|E| = e$ i u kojem je spremljena odgovarajuća težina za određeni luk. Sljedeća slika ilustrira kompaktni prikaz liste susjedstva za primjer sa slike 2.9, s dodatkom nasumice odabranih težina za svaki luk, kako bi ilustrirali i niz W_a na istom primjeru.



Slika 2.10: Primjer kompaktnog prikaza liste susjedstva.

Komentirajmo kratko primjer radi boljeg shvaćanja. Pogledajmo prvo niz V_a i prva dva elementa. Na indeksu 0 u nizu V_a nalazi se vrijednost 0 što znači da vrh v_0 pokazuje na indeks 0 u nizu E_a. Na indeksu 1 u nizu V_a nalazi se vrijednost 2 što znači da vrh v_1 pokazuje na indeks 2 u nizu E_a. Dakle, prvom vrhu susjedni su vrhovi koji se nalaze na indeksu $E_a[0] = 8$ i $E_a[2-1] = E_a[1] = 6$. U nizu W_a čitamo da težina luka od vrha 0 do 2 iznosi 5, a da je težina luka od vrha 0 do 4 vrijednosti 9. Analogno gledamo za svaki sljedeći vrh.

CPU implementacija sekvencijalnog algoritma

Dakle, u samom kodu implementirat ćemo 3 niza koji će predstavljati graf kao listu susjedstva na gore prikazan i objašnjeni kompaktan način. Niz `V` s `BR_V` (predstavlja broj vrhova) elemenata, te nizove `E` i `W` s `BR_E` (predstavlja broj bridova) elemenata.

Osim nizova koji predstavljaju graf potrebni su i pomoćni nizovi za izvođenje samog algoritma, koji su već spomenuti i kod navođenja pseudokoda.

- Niz `U` s `BR_V` elemenata predstavljat će spomenuti skup U (skup neposjećenih vrhova, odnosno vrhova za koje još nije izračunata konačna duljina najkraćeg puta). Niz će na i-tom indeksu imati vrijednost 1 ako je vrh i posjećen, a vrijednost 0 ako vrh i nije posjećen.
- Niz `C` s `BR_V` elmenata predstavljat će trenutnu vrijednost najkraćeg puta do svakog vrha. Kao što smo rekli, nakon što algoritam završi s radom (svi vrhovi postanu

posjećeni) ovaj će niz sadržavati rezultat algoritma, odnosno `C[i]` će biti duljina najkraćeg puta od početnog vrha do vrha i.

Ideju i pseudokod algoritma već smo objasnili, kao i strukturu svih podataka nad kojima se algoritam izvodi. Kako smo uveli kompaktni prikaz liste susjedstva, u odnosu na pseudokod promijenjen je način pristupa podacima.

U sljedećim primjerima uzimamo da je početni vrh $s = 0$.

```
//initialize
for( int i= 0; i < BR_V; i++ ) {
    C[i] = INT_MAX;
    U[i] = 1;
}

C[0] = 0;
U[0] = 0;
f = 0;
mssp = 0;

while( mssp < INT_MAX ) {

    //relax
    int current = V[f];

    int next;
    if( (f+1) == BR_V )
        next = BR_E;
    else
        next = V[f+1];

    for(int j = current; j < next; j++) {
        int n = E[j];

        if( U[n] ) {
                if( C[n] > ( C[f] + W[j] ) ) {
                        C[n] = C[f] + W[j];
                }
        }
    }

    mssp = INT_MAX;

    //minimum
    for( int i = 0; i < BR_V; i++ ) {
            if( U[i] ) {
```

```
                    if( C[i] < mssp ) {
                            mssp = C[i];
                            f = i;
                    }
            }
    }

    //update
    U[f] = 0;
}
```

GPU implementacija paralelnog algoritma

Graf prikazujemo s tri niza, dV, dE i dW. Nizovi su analogni nizovima V, E i W u sekvencijalnom algoritmu, ali smo svakom od njih dodali prefiks d iz jednostavnog razloga - radi lakšeg razlikovanja u čitanju koda.

Pomoćni nizovi dU i dC analogni su gore opisanim nizovima U odnosno C . U ovom algoritmu koristimo još jedan dodatni niz dF koji predstavlja skup F, skup svih vodećih vrhova u nekom koraku (detaljno objašnjeno prilikom analize pseudo koda). Pripadnost vrha i skupu F označavamo zastavicom 1 na odgovarajućem indeksu, dakle, $i \in F$ ako i samo ako dF[i] = 1.

Nizove koji predstavljaju graf prvo je potrebno kopirati iz RAM memorije na CPU u globalnu memoriju na GPU, koristeći cudaMemcpy() na način na koji je objašnjeno u prvom poglavlju, inače im ne možemo pristupati s GPU (a na kojem se izvodi ovaj algoritam).

Prije poziva bilo kojeg kernela treba odrediti vrijednost varijablama opisanima u prvom poglavlju, a to su varijable koje određuju parametre za konfiguraciju kernela nazvane threadsPerBlock i blocksPerGrid.

```
int threadsPerBlock = 512;
int blocksPerGrid   = (BR_V + threadsPerBlock - 1) / threadsPerBlock;
```

Spomenute pomoćne nizove, dU, dC i dF treba inicijalizirati. Za to služi funkcija initialize koja se pokreće jednom, na početku samog algoritma. Inicijalizaciju smo također paralelizirali, pa iz te funkcije pozivamo kernel koji vrši pridjeljivanje elementima niza.

```
__global__ void initialize_kernel(int *dC, int *dF, int *dU, int BR_V
    )
```

```
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;

    if(i >= BR_V) return;

    if(i == 0)
    {
        dC[i] = 0;
        dF[i] = 1;
        dU[i] = 0;
    }
    else
    {
        dC[i] = INT_MAX;
        dF[i] = 0;
        dU[i] = 1;
    }
}
```

```
__host__ int initialize(int *dC, int *dF, int *dU, int BR_V, int
    blocksPerGrid, int  threadsPerBlock)
{
    initialize_kernel<<<blocksPerGrid, threadsPerBlock>>>(dC, dF, dU,
        BR_V);
    return 0;
}
```

Kostur algoritma čini `while` petlja koja se vrti sve dok ne izračunamo konačne najkraće puteve do svih vrhova, odnosno sve dok uvedena varijabla `mssp` ne završi neki korak s vrijednošću `INT_MAX` (računalo ne poznaje vrijednost beskonačno pa uzimamo najveću moguću vrijednost koju `int` može poprimiti, a koja je pohranjena u ugrađenoj varijabli `INT_MAX`). Varijabla `mssp` završit će korak s vrijednošću `INT_MAX` ukoliko u tom koraku vrijednost niti jednog vrha ne bude promijenjena, što znači da je algoritam posjetio sve vrhove grafa do kojih postoji put od početnog vrha i time završio s izvršavanjem.

```
while( mssp < INT_MAX )
{
  relax(dV, dE, dW, dC, dF, dU, BR_V, BR_E, blocksPerGrid,
      threadsPerBlock);

  mssp = minimum(dC, dU, g_odata, BR_V, blocksPerGrid,
      threadsPerBlock);

  update(dC, dF, dU, BR_V, mssp, blocksPerGrid, threadsPerBlock);
}
```

U svakom koraku while petlje pozivaju se 3 funkcije koje smo već idejno objasnili - `relax()`, `minimum()` i `update()`. Ovo su domaćin (host) funkcije, iz kojih ćemo pozvati odgovarajuće kernele. Razlog zašto kernel nije pozvan direktno je samo zbog preglednosti i organizacije koda. Pogledajmo sada svaku od njih.

U **relax** dijelu, kao što smo već rekli, za sve vrhove koji su još uvijek u dU označeni zastavicom 1, a kojima je neki element iz dF sljedbenik, mijenjamo vrijednost tog elementa s obzirom na vrijednost odgovarajućeg elementa iz dF, koju čitamo iz niza dC.

U **relax** dijelu, kao što smo već rekli, za sve vrhove koji su sljedbenici vrhova iz dF, a još uvijek su u dU označeni zastavicom 1 što znači da su neposjećeni, mijenjamo vrijednost tih elemenata s obzirom na vrijednost odgovarajućeg elementa iz dF, koju čitamo iz niza dC.

```
__global__ void relax_kernel(int *dV, int *dE, int *dW, int *dC, int
    *dF, int *dU, int BR_V, int BR_E)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if(i >= BR_V) return;

    if(dF[i] == 1)
    {
        int current = dV[i];

        int next;
        if( (i+1) == BR_V)
            next = BR_E;
        else
            next = dV[i+1];

        for(int j = current; j < next; j++)
        {
            int n = dE[j];

            if(dU[n] == 1)
                atomicMin(&dC[n], dC[i] + dW[j]);
        }
    }
}
```

U prošlom smo poglavlju objasnili koliziju koja može nastati ukoliko minimum vrijednosti tražimo na klasičan način, kao u CPU kodu. Prisjetimo se, kako je nemoguće međusobno sinkronizirati sve dretve (moguća sinkronizacija samo na razini blokova) moguće je da dvije dretve u isto vrijeme računaju minimum i da onda u konačnici ostane

izračunata veća vrijednost, a ne minimum. Zato smo ovdje upotrijebili funkciju `atomicMin` koju nam CUDA stavlja na raspolaganje. Ovo je specifična funkcija koja pročita vrijednost s prve adrese, usporedi s drugom vrijednošću te zapiše minimum te dvije vrijednosti ponovno na prvu adresu. Dakle, točno ono što nama treba. Uz to, ova funkcija osigurava da u tom postupku niti jedna druga dretva neće pristupati istoj adresi, zbog čega je i nazvana atomskom. Točno taj zahtjev rješava spomenuti problem kolizije.

```
__host__ int relax(int *dV, int *dE, int *dW, int *dC, int *dF, int *
    dU, int BR_V, int BR_E, int blocksPerGrid, int threadsPerBlock)
{
    relax_kernel<<<blocksPerGrid, threadsPerBlock>>>(dV, dE, dW, dC,
        dF, dU, BR_V, BR_E);
    return 0;

}
```

Način na koji se poziva kernel uobičajen je, te objašnjen detaljno u prvom poglavlju.

Ono što je također vidljivo iz poziva činjenica je da koliko ima vrhova grafa toliko se dretvi pokreće istovremeno (paralelno) te svaka radi sa svojom kopijom kernela. To znači da istovremeno mijenjamo sve one vrhove koji zadovoljavaju uvjete u tom trenutku. Ubrzanje paralelnog algoritma je utoliko što istovremeno mijenjamo sljedbenike više od jednog vodećeg vrha, što u konačnici daje manji broj koraka u kojem će algoritam izračunati konačne vrijednosti.

U `minimum` dijelu želimo odrediti minimum vrijednosti C svih vrhova koji imaju zastavicu 1 u nizu dU. Već smo spomenuli da je minimum nemoguće tražiti istovremeno po svim dretvama jer ih je nemoguće sve međusobno sinkronizirati pa se minimum traži na razini svakog bloka. Princip je isti kao onaj opisan prilikom objašnjavanja relax procedure, samo se razmak među indeksima niza u svakom koraku određuje na malo optimalniji način. Dretve su i ovdje sinkronizirane po koracima i to funkcijom `__syncthreads()`.

Kako bi sve dretve bloka međusobno komunicirale u zajedničkom traženju minimuma, potrebna je dijeljena memorija preko koje će to i činiti. U ovom kernelu dinamički alociramo niz `sdata[]` u dijeljenoj memoriji što označavamo ključnom riječju `extern` neposredno prije `__shared__`. Broj elemenata koji će biti alociran definira se izvan samog kernela, u pozivu, odnosno konfiguraciji kernela. Svaki blok imati će svoju kopiju takvog niza. Nakon što sve dretve bloka završe s radom prvi element tog dijeljenog niza će biti traženi minimum. Izabiremo jednu dretvu koja će to zapisati u globalnu memoriju (niz `g_odata[]`) vidljivu svim dretvama unutar svih blokova.

```
__global__ void reduce_min(int *g_idata, int *dU, int *g_odata,
    unsigned int n)
{

    extern __shared__ int sdata[];

    unsigned int tid = threadIdx.x;
    unsigned int blockSize = blockDim.x;
    unsigned int i = blockIdx.x*blockSize*2 + threadIdx.x;
    unsigned int gridSize = blockSize*2*gridDim.x;

    int myMin = INT_MAX;

    while (i < n)
    {
        if(dU[i])
            myMin = (myMin<g_idata[i]) ? myMin : g_idata[i];

        if (i + blockSize < n)
            if(dU[i + blockSize])
                myMin = (myMin<g_idata[i+blockSize]) ? myMin :
                    g_idata[i+blockSize];

        i += gridSize;
    }

    sdata[tid] = myMin;
    __syncthreads();

    if (blockSize >= 512)
    {
        if (tid < 256)
        {
            sdata[tid] = myMin = (myMin<sdata[tid+256]) ? myMin :
                sdata[tid + 256];
        }
        __syncthreads();
    }

    if (blockSize >= 256)
    {
        if (tid < 128)
        {
            sdata[tid] = myMin = (myMin<sdata[tid+128]) ? myMin :
                sdata[tid + 128];
        }
        __syncthreads();
```

```
    }

    if (blockSize >= 128)
    {
        if (tid <  64)
        {
            sdata[tid] = myMin = (myMin<sdata[tid+64]) ? myMin :
                sdata[tid + 64];
        }
        __syncthreads();
    }

    if (tid < 32)
    {
        if (blockSize >=  64)
        {
            smem[tid] = myMin = (myMin<smem[tid+32]) ? myMin : smem[
                tid + 32];
        }

        if (blockSize >=  32)
        {
            smem[tid] = myMin = (myMin<smem[tid+16]) ? myMin : smem[
                tid + 16];
        }

        if (blockSize >=  16)
        {
            smem[tid] = myMin = (myMin<smem[tid+8]) ? myMin : smem[
                tid + 8];
        }

        if (blockSize >=   8)
        {
            smem[tid] = myMin = (myMin<smem[tid+4]) ? myMin : smem[
                tid + 4];
        }

        if (blockSize >=   4)
        {
            smem[tid] = myMin = (myMin<smem[tid+2]) ? myMin : smem[
                tid + 2];
        }

        if (blockSize >=   2)
        {
            smem[tid] = myMin = (myMin<smem[tid+1]) ? myMin : smem[
```

```
                tid + 1];
        }
    }

    if (tid == 0)
        g_odata[blockIdx.x] = sdata[0];

}
```

```
__host__ int minimum(int *dC, int *dU, int *g_odata, int BR_V, int
    blocksPerGrid, int threadsPerBlock)
{

        minimum_kernel<<<blocksPerGrid, threadsPerBlock,
            threadsPerBlock*sizeof(int)>>>(dC, dU, g_odata, BR_V);

        int *rezM;
        MALLOC(rezM, int, blocksPerGrid*sizeof(int));
        SC( cudaMemcpy( rezM, g_odata, blocksPerGrid * sizeof(int),
            cudaMemcpyDeviceToHost ) );

        int min = rezM[0];
        for(int i = 1; i < blocksPerGrid; i++)
        {
                if(rezM[i] < min) min = rezM[i];
        }

        return min;

}
```

Nakon izvršavanja kernela minimum svakog bloka zasebno zapisan je u `g_odata[]` i preostaje naći minimum tog niza. To radimo sekvencijalno, u funkciji na domaćinu.

Osvrnimo se još samo na konfiguraciju izvršavanja kernela (parametri unutar 3 šiljaste zagrade). Primjećujemo promjenu u odnosu na objašnjeni klasični način postavljanja parametara. Promjena je dodatni treći parametar. Taj parametar odnosi se na veličinu niza `sdata[]` koji se deklarira u dijeljenoj memoriji. Ovaj parametar obavezan je prilikom korištenja takvog niza u kernelu. Dimenzija će biti jednaka broju dretvi po bloku jer je memorija dijeljena na razini bloka.

Zadnja funkcija koju u svakom koraku pozivamo je `update`. U toj funkciji jednostavno želimo sve vrhove koji su u `dU` označeni s 1, s trenutno najmanjom vrijednosti (čak i ako ih ima više) ukloniti iz U na način da ih označimo s nulama u dU te postaviti zastavice u

dF koje kažu da su u idućem koraku ti vrhovi vodeći.

```
__global__ void update_kernel(int *dC, int *dF, int *dU, int BR_V,
    int mssp)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;

    if(i >= BR_V) return;

    dF[i] = 0;

    if(dC[i] == mssp)
    {
        dU[i] = 0;
        dF[i] = 1;
    }
}
```

```
__host__ int update(int *dC, int *dF, int *dU, int BR_V, int mssp,
    int blocksPerGrid, int threadsPerBlock)
{
    update_kernel<<<blocksPerGrid, threadsPerBlock>>>(dC, dF, dU,
        BR_V, mssp);
    return 0;
}
```

Kada while petlja završi s radom rezultat se nalazi u nizu dC. Ako želimo provjeriti rezultat algoritma i ispisati niz trebamo ga iskopirati u CPU koristeći funkciju cudaMemcpy() na način koji smo objasnili u prvom dijelu.

2.9 Testiranje

Testiranje je provedeno na računalu Fermi, na grafičkoj kartici Nvidia Tesla S2050.

Kako bi usporedili brzinu izvođenja Dijkstrinog algoritma za CPU i GPU potrebno je prvo generirati graf nad kojim će oba algoritma računati najkraće puteve.

Generiranje grafa

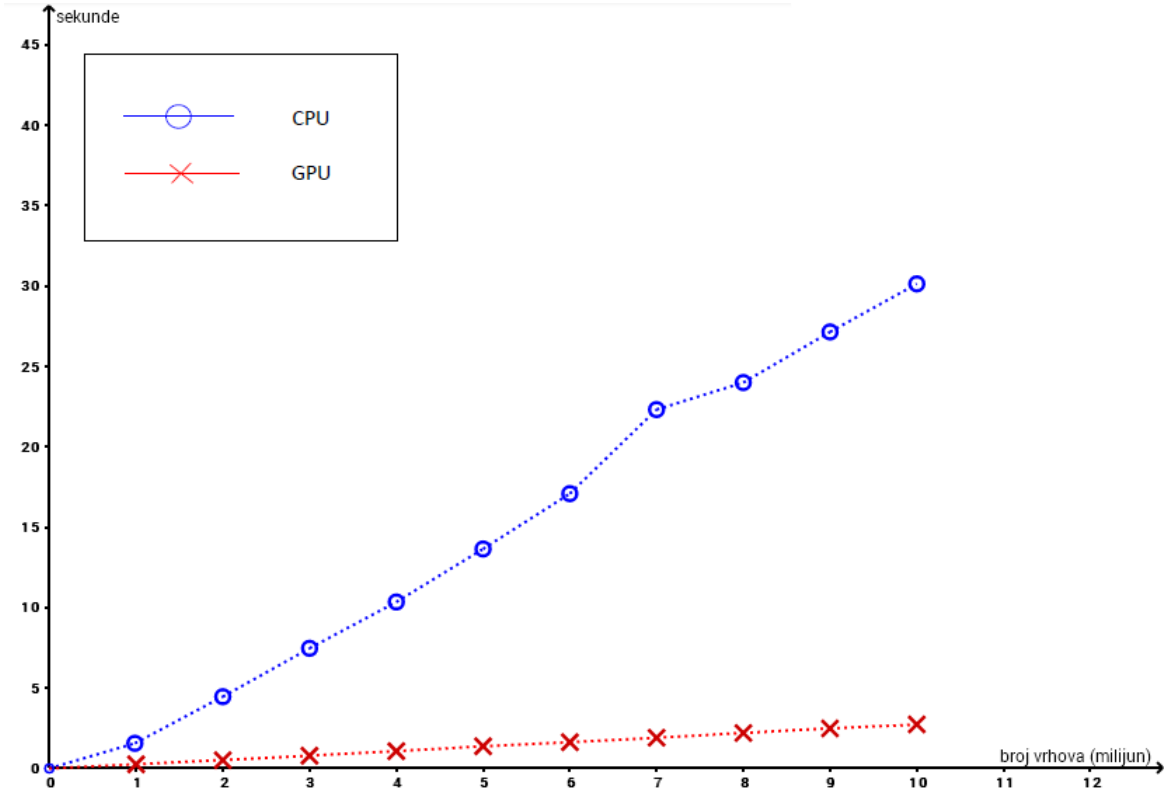
Prvi zahtjev grafa jest da bude povezan, što znači da postoji put između početnog vrha i svakog drugog vrha grafa. Kako bi taj zahtjev osigurali potrebno je u prvom koraku generirati po jedan luk iz svakog vrha tako da osiguramo povezanost grafa u prvom koraku.

U sljedećem koraku generirati ćemo ostale lukove, tako da pazimo da svaki vrh bude povezan s istim brojem vrhova. To je potrebno kako bi graf bio što uravnoteženiji i time pogodniji za provođenje testiranja.

Za stupanj svakog vrha grafa (broj vrhova s kojima je povezan) uzeta je vrijednost 7, a za broj vrhova grafa od 1 milijuna, pa do 10 milijuna. Ove specifikacije su preuzete iz članka [6] kako bi mogli pratiti rezultate u usporebi s njihovima.

Rezultati

Generirat ćemo grafove s varijabilnim brojem vrhova, te ćemo na svakom grafu pokrenuti prvo CPU pa GPU algoritam kako bi usporedili brzinu. Za svaki odabrani broj vrhova graf je generiran 10 puta. Kao brzinu za taj broj vrhova za određeni algoritam uzeta je aritmetička sredina tih izračuna.

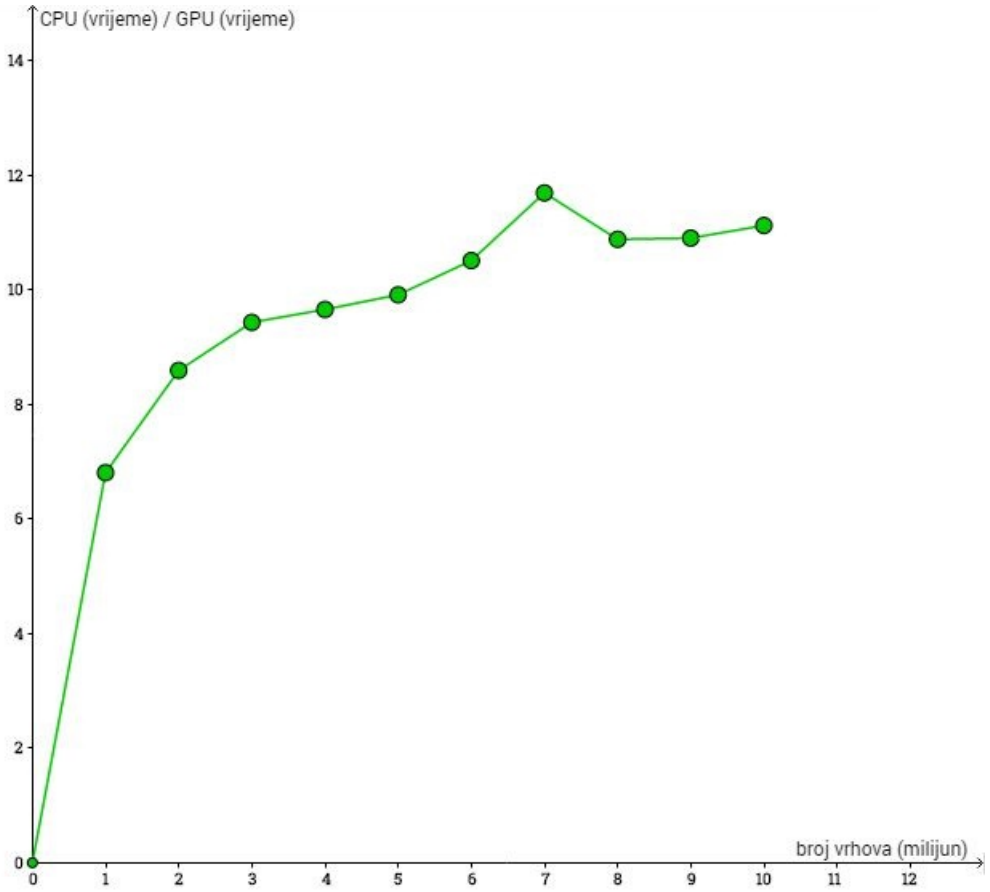


Slika 2.11: Rezultati CPU i GPU

Primjećujemo iz grafa na Slici 2.11 da je GPU konstantno izrazito brz, a čak ni CPU za ove brojeve ne traje toliko dugo da stvori probleme. Ono što je uzelo najviše vremena u izvođenju jest samo generiranje grafa, i zato je graf od 10 milijuna vrhova gornja granica kod našeg testiranja. Na grafu su prikazana samo vremena izvođenja samog Dijkstra algoritma na CPU te na GPU, a ne i vrijeme generiranje grafa.

Iz istog grafa jasno vidimo da GPU postaje izrazito bolji u odnosu na CPU kako se veličina grafa povećava. Ako slijedimo aproksimacijsku funkciju za ova dva algoritma možemo zamisliti da bi za određenu veličinu grafa CPU postao jako spor za razne izračune u praksi dok bi GPU i dalje Dijkstrin algoritam izvršavao u razumnom vremenu.

Također na Slici 2.12 vidimo kako omjer vremena izvršavanja algoritma na CPU i algoritma na GPU raste.



Slika 2.12: Omjer vremena CPU i GPU

Poglavlje 3

Zaključak

Rezultati ovog rada pokazali su nam da su paralelni algoritmi na GPU zaista za određene probleme pogodniji od tradicionalnih sekvencijalnih algoritama na CPU (u ovom slučaju za algoritme na grafovima, specifično Dijkstrin algoritam). Pogodniji su u smislu da su puno brži za velike setove podataka, što je uvijek poželjno jer ubrzava puno poslova kakvi se u zadnje vrijeme naročito često javljaju u praksi (obrada podataka u socijalnim mrežama i ostale big data aplikacije).

Razlog koji bi u široj upotrebi mogao predstavljati prepreku korištenju GPU je naravno potpuno nov način programiranja koji je potrebno usvojiti. Moguće je da će za puno ljudi prvi pogled na program u CUDA-i izgledati neshvatljivo i možda ih odbiti. Ali, uz dobru literaturu i malo strpljenja programiranje u CUDA-i jako brzo postaje jednako prirodno kao programiranje u programskom jeziku C odnosno C++. Programeri koji su stvorili ovaj programski model zaista su se potrudili da bude što jednostavniji za korištenje ljudima koji imaju dobru podlogu u programskom jeziku C.

Kombinacija sekvencijalnog i paralelnog programiranja koju nam programski model CUDA omogućava idealni je spoj jer u istom programu možemo iskoristiti najbolje od CPU i GPU, ovisno što nam za koji dio koda odgovara. Jako puno govori i činjenica da treuntno najmoćnija računala svijeta koriste hibridnu arhitekturu koja se sastoji od nekoliko desetaka tisuća CPUova i nekoliko desetaka tisuća GPUova.

Mišljenja sam da omjer dobrih strana i nedostataka ovog pristupa ide u korist heterogenom programiranju i će se u budućnosti CUDA i drugi slični programski modeli sve više razvijati i primjenjivati za složene analize velikih količina podataka.

Bibliografija

[1] J. Sanders, E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*, Addison-Wesley Professional, Boston, 2010 .

[2] J. Cheng, M. Grossman, T. McKercher, *Professional CUDA C Programming*, John Wiley and Sons, Inc., Indianapolis, 2014.

[3] NVIDIA, *CUDA C PROGRAMMING GUIDE*, dostupno na www.nvidia.com (svibanj 2015.).

[4] D. B. Kirk, W. W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*, Morgan Kaufmann Publishers Inc., San Francisco, 2010.

[5] P. Harish, P. J. Narayanan, *Accelerating Large Graph Algorithms on the GPU Using CUDA*, High Performance Computing - HiPC 2007: 14th International Conference, Goa, India, December 18-21, 2007 Proceedings, LNCS 4873 (S. Aluru, M. Parashar, M. Badrinath, V. K. Prasanna) Springer, Berlin, 2007, 197–208.

[6] P. J. Martin, R. Torres, A. Gavilanes, *CUDA Solutions for the SSSP Problem*, Computational Science – ICCS 2009: 9th International Conference Baton Rouge, LA, USA, May 25-27, 2009 Proceedings, Part I, LNCS 5544 (G. Allen, J. Nabrzyski, E. Seidel, G. D. van Albada, J. Dongarra, P. Sloot) Springer, Berlin, 2009, 904–913.

Sažetak

U ovom radu se analizira potencijal korištenja visoko paralelnih grafičkih procesora za implementaciju algoritama na grafovima, s naglaskom na Dijkstrin algoritam za nalaženje najkraćih puteva u grafu. Dan je opis tehnologije CUDA tvrtke NVidia, zajedno s pripadnim programskim modelom. Diskutiramo aspekte paralelizacije Dijkstrinog algoritma u okvirima ovog modela na GPU. Provedeni numerički testovi pokazuju značajno ubrzanje u odnosu na klasičnu implementaciju na CPU.

Summary

In this work we analyze the potential of using a highly parallel graphical processor in order to implement graph algorithms, in particular, Dijkstra's algorithm for finding shortest paths in a graph. We describe the CUDA technology which was introduced by NVidia, together with its programming model. We discuss aspects of parallelizing Dijkstra's algorithm within this GPU framework. Numerical experiments show that this approach achieves significant speedup compared to the classical CPU implementation.

Životopis

Rođena sam 14. rujna 1991. godine u Zadru. Nakon završene osnovne škole u Splitu, upisala sam matematičku gimnaziju u Splitu. Godine 2010. sam upisala preddiplomski studij Matematike i informatike na Prirodoslovno-matematičkom fakultetu u Splitu. Preddiplomski studij sam završila 2013. godine. Iste godine sam upisala diplomski sveučilišni studij Računarstvo i matematika na Prirodoslovno-matematičkom fakultetu u Zagrebu.