SVEUČILIŠTE U ZAGREBU
**FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA**

ZAVRŠNI RAD br. 1537

# IZVEDBA ALGORITAMA RAČUNALNOG VIDA NA GRAFIČKIM PROCESORIMA

Ante Trbojević

Zagreb, lipanj 2010.

*Zahvala mentoru doc. dr. sc. Siniši Šegviću na stručnoj pomoći i danim sugestijama pri izradi završnog rada.*

# Sadržaj

# Poglavlje 1

# Uvod

Današnje grafičke kartice sadrže preko stotinjak, pa čak i preko tisuću procesnih elemenata. Donedavno, grafičke kartice koristile su se samo za prikaz vizualnog sadržaja. Međutim pokazalo se kako se arhitektura grafičkih kartica i ogromna procesorska snaga može iskoristi i za računske zadatke. Razvijeni su radni okviri poput CUDA-e i OpenCL-a koji pružaju mogućnost iskorištavanja grafičkih procesora za izvođenje računskih zadataka koji su se donedavno izvodili samo na procesorima opće namjene (CPU).

Arhitektura grafičkih kartica posebno je pogodna za izvođenje algoritama koji se mogu što više paralelizirati. Obrada slika i računalni vid sadrže mnoštvo algoritama s velikim paralelizacijskim svojstvima, pa tako ovaj rad opisuje izvedbu konvolucije, afine transformacije i homografije u OpenCL-u. Navedene implementacije moguće je iskoristi u složenijim algoritmima računalnog vida kako bi se iskoristile značajne vremenske uštede u odnosu na implementacije koje se izvode na procesoru opće namjene (CPU).

Poglavlje 2 detaljno opisuje arhitekturu današnjih grafičkih jedinica dva najveća proizvođača NVIDIA i ATI. Poglavlje opisuje važnije dijelove sklopovlja grafičke kartice i prikazuje razliku između arhitektura NVIDIA GT200 i ATI Evergreen. Opisuje se organizacija memorije te kako procesne jedinice grafičke kartice izvode instrukcije.

Poglavlje 3 daje kratak uvod u OpenCL, opisuje temeljne koncepte GPGPU programiranja i OpenCL-a, i daje naputke za oblikovanje algoritama kako bi se maksimalno iskoristila snaga grafičke kartice.

Poglavlje 4 opisuje izvedbu konvolucije sa separabilnim filtrima u OpenCL-u. Opisane su tri različite implementacije konvolucije.

Poglavlje 5 opisuje način izvedbe afine transformacije i homografije u OpenCL-u.

Naposljetku slijede poglavlja koja prikazuju ekperimentalne rezultate razvijenih implementacija konvolucije, afine transformacije i homografije, zajedno s komentarima dobivenih rezultata. Eksperimenti su provedeni na grafičkim karticama NVIDIA GT240 i ATI Radeon HD 5670.

# Poglavlje 2

# Arhitektura grafičkih kartica

Grafičke kartice pružaju arhitekturu koja ih čini pogodnima za obrađivanje velikih podatkovnih polja i slika. Strukture poput vektora, matrica ili slika prikazuju se pomoću podatkovnih polja. Grafičke kartice svojom arhitekturom pružaju efikasan alat za obavljanje raznih transformacija nad takvim strukturama.

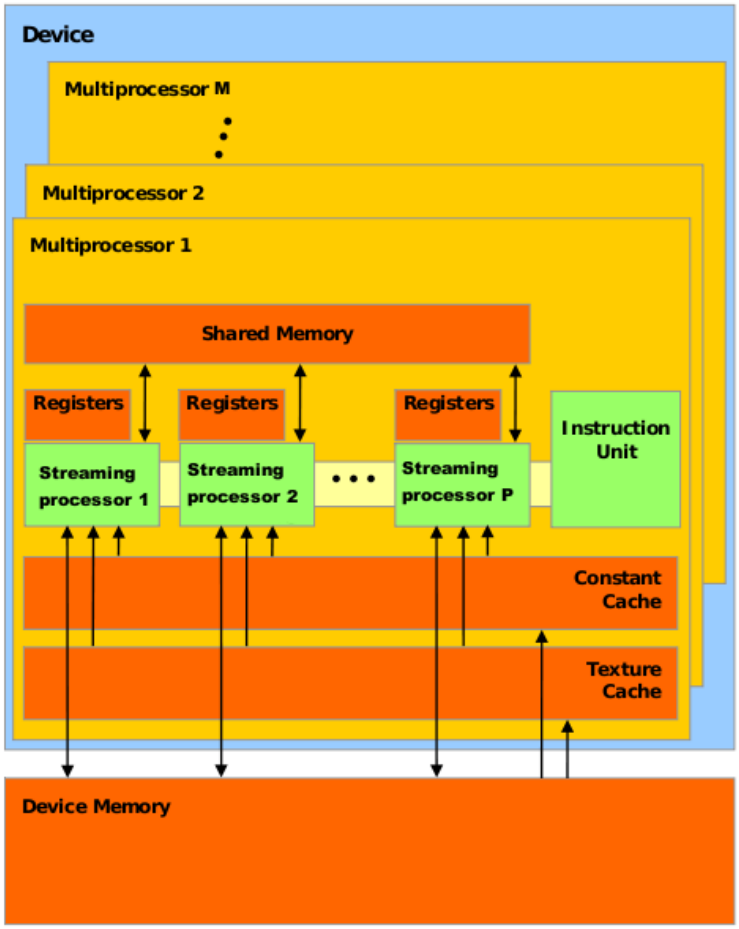
## 2.1 Organizacija sklopovlja

Grafičke kartice sadrže nekoliko razina koje su organizirane hijerarhijski. Na prvoj razini nalaze se **protočni multiprocesori** (engl. *streaming multiprocessors*) [2], u ATI-jevoj literaturi koristi se naziv **računska jedinica** (engl. *compute unit*) [7]. Svaka GPU sadrži nekoliko protočnih multiprocesora, odnosno računskih jedinica, a svaki multiprocesor (računska jedinica) sadrži nekoliko procesora. Multiprocesor je organiziran prema **SIMD** (engl. *Single Instruction Multiple Data*) arhitekturi, dakle jedna instrukcija obavlja se nad više podatkovnih elemenata, tj. nakon svakog procesorskog takta, svaki procesor multiprocesora obavlja istu instrukciju, ali nad različitim podacima [2]. Valja primijetiti kako različiti multiprocesori paralelno mogu izvoditi različite naredbe u istom periodu glavnog takta grafičke kartice, dakle međusobno su nezavisni, međutim procesori unutar određenog multiprocesora paralelno izvode istu naredbu.

## Arhitektura Nvidia GT200

Već je spomenuto kako svaki multiprocesor (računska jedinica) sadrži više procesora. Arhitektura samih procesora ovisi o proizvođaču, ali i o seriji grafičke kartice. Tako

primjerice NVIDIA, procesore unutar multiprocesora naziva **protočnim procesorima** (engl. *stream processors*), koristi se i termin **jezgra**, ali i „Cuda Core". Protočni procesori imaju mogućnost izvođenja operacija nad cjelobrojnim podacima jednostruke (32 bita) preciznosti (engl. *single-precision integer*) i operacija nad podacima s pomičnim zarezom jednostruke (32 bita) preciznosti (engl. *single-precision floating point*). Svaki multiprocesor sadrži 8 protočnih procesora, međutim pored „običnih" procesora, multiprocesor sadrži još jednu ili dvije jedinice (ovisi o seriji grafičke kartice) koje obavljaju operacije dvostruke preciznosti (64 bita) pomičnog zareza (engl. double-precision floating-point) i transcendentalne operacije[1], takve jedinice nazivaju se **jedinicama za specijalne funkcije** (engl. *special function units*) [2].

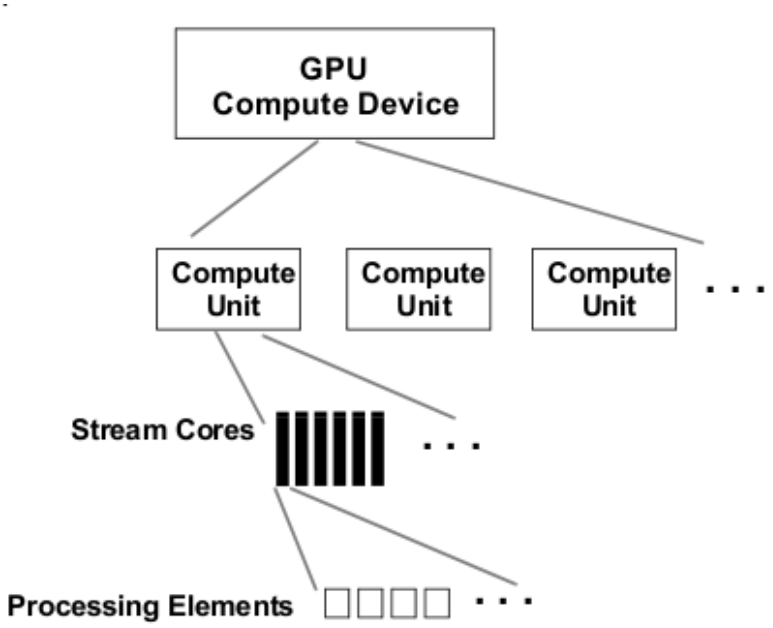



**Slika 2.1:** *Arhitektura grafičkih kartica proizvođača NVIDIA [2]. P = 8 i M = 12 na grafičkoj kartici NVIDIA GT240.*

[1]Transcendentalne operacije - složene matematičke funkcije, npr. sinus, kosinus, logaritam i sl.

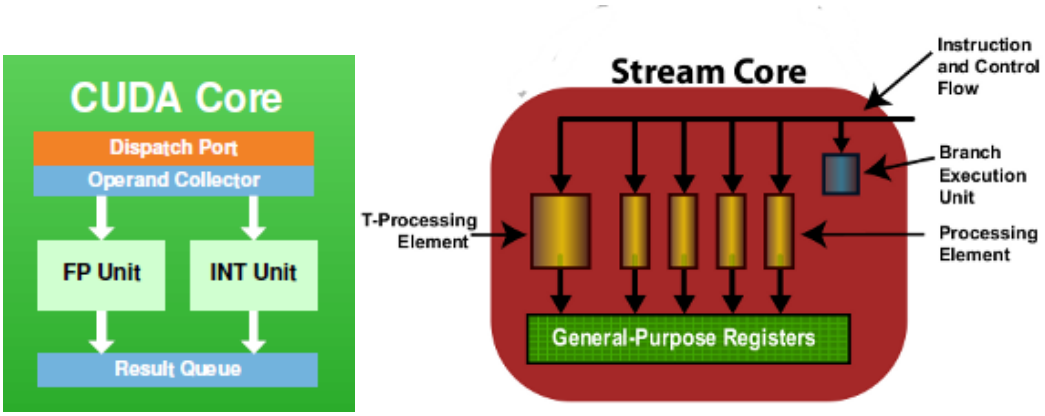
# Arhitektura ATI Evergreen

ATI-jeve grafičke kartice također sadrže jezgre, koje se nazivaju **protočnim jezgrama** (engl. *stream cores*), tj. procesore koji obavljaju istu zadaću kao i procesori NVIDIA-e, međutim svaka jezgra je sastavljena od pet procesnih elemenata. Četiri procesna elementa obavljaju cjelobrojne operacije i operacije jednostruke preciznosti s pomičnim zarezom, dok peti element, tzv. T-procesni element (engl. *T-processing element*), obavlja transcendentalne operacije. Operacija dvostruke preciznost (64 bita) s pomičnim zarezom obavlja se spajanjem dva ili četiri procesna elementa (ne ubrajajući T-procesni element) [7]. Svaka protočna jezgra sadrži vlastite registre opće namjene, a procesni elementi unutar jedne protočne jezgre zajednički koriste registre opće namjene.




**Slika 2.2:** *Arhitektura grafičkih kartica proizvođača ATI [7].*

Valja primijetiti kako je ovakva arhitektura, gdje je protočna jezgra organizirana kao peterostruki VLIW (engl. *very long instruction word*) procesor, pogodnija za vektorske tipove podataka i vektorske operacije od arhitekture NVIDIA-e (skalaran protočni procesor). Primjerice neka je potrebno izvršiti proces invertiranja slike i neka je svaka protočna jezgra (protočni procesor) zadužena za obradu jednog slikovnog elementa (engl. *pixel*). Neka je slikovni element definiran RGBA standardom, tj. vrijednost




**(a)** Protočni procesor **(b)** Protočna jezgra

**Slika 2.3:** *Protočni procesor (NVIDIA) [1] i protočna jezgra(ATI) [7]*

slikovnog elementa definirana je vektorom od četiri 8-bitne vrijednosti. Kako bi se obavio zadatak, svaku komponentu slikovnog elementa potrebno je komplementirati. Neka su na raspolaganju dvije grafičke kartice s istim brojem protočnih jezgri (protočnih procesora), ali jedna organizirana prema ATI-jevoj arhitekturi i jedna prema arhitekturi NVIDIA-e. Dakle obje grafičke kartice sadrže $n$ protočnih jezgri (protočnih procesora), međutim kod ATI-jeve grafičke kartice, svaka protočna jezgra sadrži još pet procesnih elemenata, pa će grafička kartica s ATI-jevom arhitekturom brže obraditi sliku od grafičke kartice s arhitekturom proizvođača NVIDIA. Naime svaki od četiri procesna elementa unutar protočne jezgre može obraditi jednu komponentu slikovnog elementa (T-procesni element ostaje neiskorišten jer je odgovoran za složene operacije), pa se transformacija nad slikovnim elementom obavlja paralelno na protočnoj jezgri. Kod grafičkih kartica organiziranih prema arhitekturi proizvođača NVIDIA, transformaciju slikovnog elementa nije moguće izvesti paralelno, jer protočni procesor ne sadrži više procesnih elemenata, pa se transformacija nad slikovnim elementom obavlja serijski kroz četiri koraka. Pritom se pretpostavlja da izvorni kôd koristi vektorske tipove podataka i vektorske operacije kako bi se uopće mogle iskoristi pogodnosti ATI-jeve arhitekture. Međutim, ako izvorni kôd ne koristi vektorske tipove podataka, već skalarne, tada će samo jedan procesni element u protočnoj jezgri obavljati transformaciju, dok će ostali ostati neiskorišteni, pa će obje kartice, teoretski, u jednakom vremenu obaviti invertiranje slike.

Dakle da bi se iskoristili svi procesni elementi kod ATI-jevih grafičkih kartica, poželjno je koristiti vektorske tipove podataka i operacije gdje god je to moguće. Ipak

koeficijent ubrzanja u odnosu na skalarnu arhitekturu protočnog procesora (NVIDIA) nije linearan, jer brzina izvođenja ovisi i o drugim svojstvima sklopovlja, kao što je memorijska propusnost, broj multiprocesora (računskih jedinca) itd.

**Tablica 2.1:** *Nazivi za ekvivalente sklopove NVIDIA - ATI.*

| NVIDIA | ATI |
|---|---|
| uređaj (engl. *device*) | računski uređaj (engl. *compute device*) |
| multiprocesor (engl. *multiprocessor*) | računska jedinica (engl. *compute unit*) |
| protočni procesor (engl. *stream processor*) | protočna jezgra (engl. *stream core*) |
| - | procesni element (engl. *processing element*) |

## 2.2 Organizacija memorije

Današnje grafičke kartice posjeduju nekoliko razina memorije, razine su organizirane hijerarhijski slično kao i procesne jedinice.

**Globalna memorija** Grafička kartica sadrži svoju vlastitu radnu memoriju, koja se skraćeno zove VRAM (engl. *Video Random Access Memory*), Veličina radne memorije GPU-a iznosi od nekoliko stotina megabajta do reda veličine gigabajta (iznimka su integrirane GPU koje dijele memoriju sa središnjim procesorom). U VRAM memoriju mogu se pohraniti podaci koji se prenose iz radne memorije računala domaćina (engl. *host*). Globalnom memorijom mogu se služiti svi protočni procesori, naime protočni procesori dohvaćaju iz globalne memorije željene podatke te ih pohranjuju u vlastite registre (privatna memorija) i izvršavaju zadane operacije nad podacima. Filozofija korištenja i pristupa slična je kao i kod procesora opće namjene.

**Lokalna (dijeljena) memorija** Osim VRAM memorije koja se nalazi na dnu hijerarhije (vidi sl. 2.4) i kojoj svi protočni procesori svih računskih jedinica mogu pristupati, grafičke kartice sadrže i **lokalnu memoriju**, koristi se i pojam **dijeljena memorija** [2]. Svaka računska jedinica (protočni multiprocesor) sadrži vlastitu lokalnu memoriju. Takva memorija služi kao brza memorija pomoću koje protočni procesori unutar iste računske jedinice mogu dijeliti podatke puno brže od korištenja globalne memorije. Svi protočni procesori koji se nalaze u istoj računskoj jedinici imaju omogućen

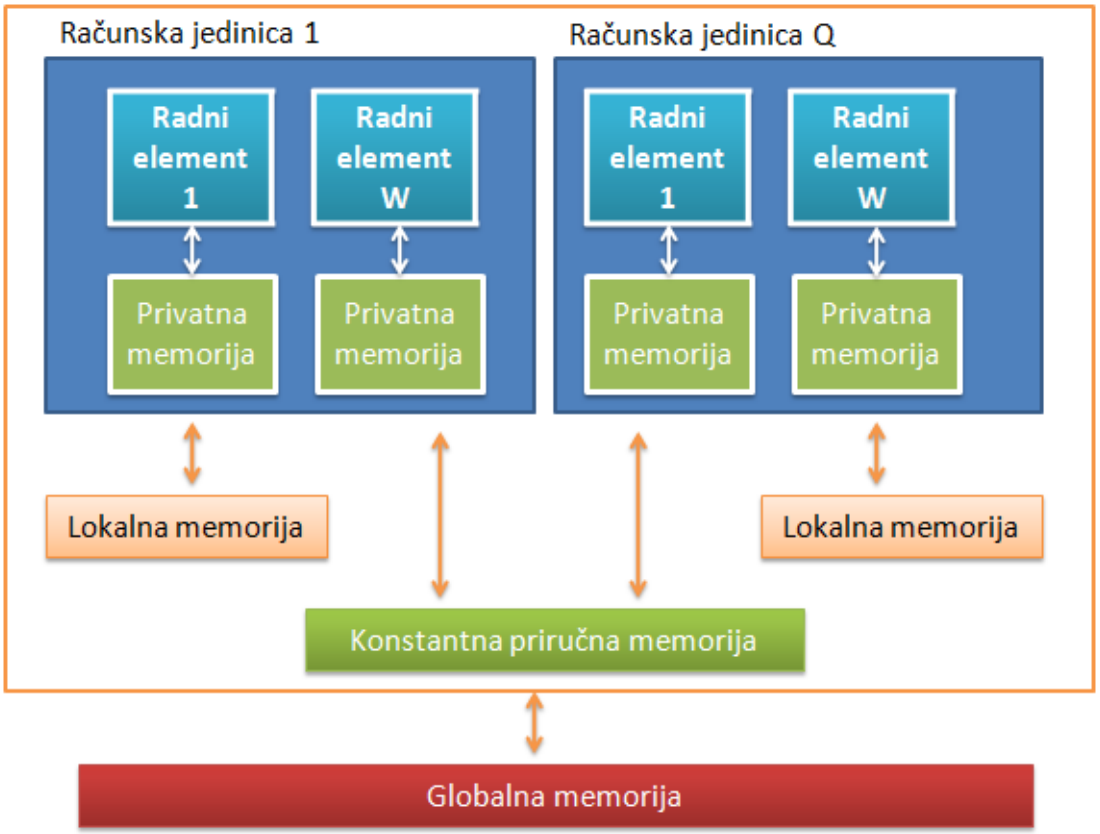
pristup lokalnoj memoriji računske jedinice u kojoj se nalaze, međutim nije moguće pristupiti lokalnoj memoriji druge računske jedinice. Pristup lokalnoj memoriji, u idealnom slučaju (vidi odjeljak 3.2), brži je i do 150× od pristupa globalnoj memoriji [4], pa ova razina memorije odgovara priručnoj memoriji (engl. *cache*) procesora opće namjene. Primjerice u lokalnu memoriju obično se pohranjuju podaci koje se više puta koriste. Priručna memorija višestruko ubrzava izvođenje programa, međutim za razliku od procesora opće namjene (CPU) koji samostalno na sklopovskoj razini manipulira priručnom memorijom, grafičke kartice ne posjeduju takav mehanizam, već ostavljaju taj dio posla programerima. Upravo je lokalna memorija prostor pomoću kojeg programer može ostvariti neku vrstu priručne memorije. Naime programeri moraju sami u programskom kôdu osmisliti način manipuliranja priručnom memorijom, tj. koje podatke spremiti u lokalnu memoriju i što je najvažnije kako ih organizirati da pristup bude optimalan (vidi odjeljak 3.2). Izuzetno je važno podatke u lokalnoj memoriji organizirati na način koji omogućava kvalitetno iskorištavanje brzine lokalne memorije, jer u suprotnom performanse mogu biti razočaravajuće.

**Privatna memorija** U odjeljku 2.1 već je spomenuto kako svaki protočni procesor sadrži vlastite 32-bitne registre opće namjene. Hijerarhijski privatna memorija nalazi se iznad lokalne memorije, također protočni procesori ne mogu pristupati privatnoj memoriji drugih protočnih procesora, već samo vlastitoj privatnoj memoriji. Privatna memorija koristi se kao stog za spremanje lokalnih varijabli prilikom izvođenja OpenCL programa.

**Konstantna memorija** Grafičke kartice sadrže i tzv. konstantnu priručnu memoriju (engl. *constant cache memory*). Konstanta priručna memorija, slična je kao i priručna memorija kod CPU-a, međutim moguće je samo predpohraniti (engl. *to cache*) podatke koje su deklarirani konstantnima, tj. oni koji se ne mijenjaju. Takvi podaci, tj. takve varijable (ili nizovi) označavaju se ključnom riječi `__constant` u OpenCL-u.

**Teksturna memorija** Teksturna memorija također je priručna memorija, ona omogućava mehanizam bržeg pristupa slikovnim podacima (nizovima podataka) koji su pohranjeni u memoriju kao 2D (slike) ili 3D objekti (volumeni). Primjerice, ako je u globalnoj memoriji pohranjena dvodimenzionalna slika, tada se prilikom pristupa slikovnom elementu (točci) iz globalne memorije dohvaća i okolina (slikovni el-

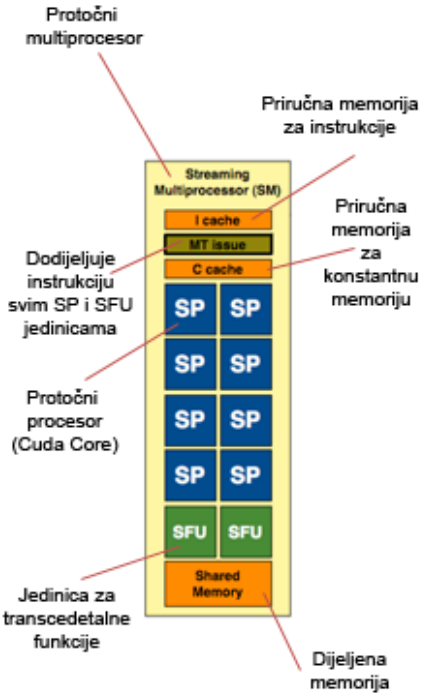
ementi oko pročitanog slikovnog elementa), gdje se zatim okolina slikovnog elementa pohranjuje u teksturnu priručnu memoriju. Naime vrlo je vjerojatno da će neposredno nakon dohvata slikovnog elementa, uslijediti i pristupati okolnim elementima, pa se oni pohranjuju u teksturnu priručnu memoriju, jer je dohvat iz priručne memorije brži od dohvata iz globalne memorije. Ova razina memorije, kod današnjih grafičkih kartici, analogna je L2 priručnoj memoriji procesora opće namjene [7]. Iz teksturne memorije, baš kao i konstante memorije, moguće je samo čitanje podatka, dok spremanje nije moguće, tj. prilikom pohrane pohrane, element neće biti predpohranjen u priručnu memoriju, pa onda automatski (posredstvom mehanizama sklopovlja) pohranjen u globalnu memoriju, već se direktno pristupa globalnoj memoriji. Podaci za koje se želi osigurati mogućnost predpohrane (engl. *caching*) u teksturnu memoriju, u OpenCL moraju se deklarirati kao tip podataka `image2d_t` (za slike) ili `image3d_t` (za 3D objekte) [17].




**Slika 2.4:** *Prikaz memorijskog sklopovlja GPU-a i prikaz dosega za radne elemente.*

## 2.3 Organizacija izvođenja instrukcija

Svaka grafička kartica sadrži i poseban sklop (ili više njih) čiji je zadatak raspoređivanje instrukcija protočnim procesorima (jezgrama). Kod grafičkih kartica arhitekture NVIDIA GT 200, svaki multiprocesor (vidi sl. 2.5) sadrži **instrukcijsku jedinicu** (engl. *multi-threaded instruction unit*) koja dodijeljuje naredbe svojim protočnim procesorima i specijalnim jedinicama za složene operacije. Grafičke kartice s arhitekturom ATI Evergreen sadrže jedan procesor koji je na razini cijele grafičke kartice odgovoran za raspoređivanje poslova svim protočnim jezgrama (engl. *ultra-threaded dispatch processor*) na svim računskim jedinicama (multiprocesorima).

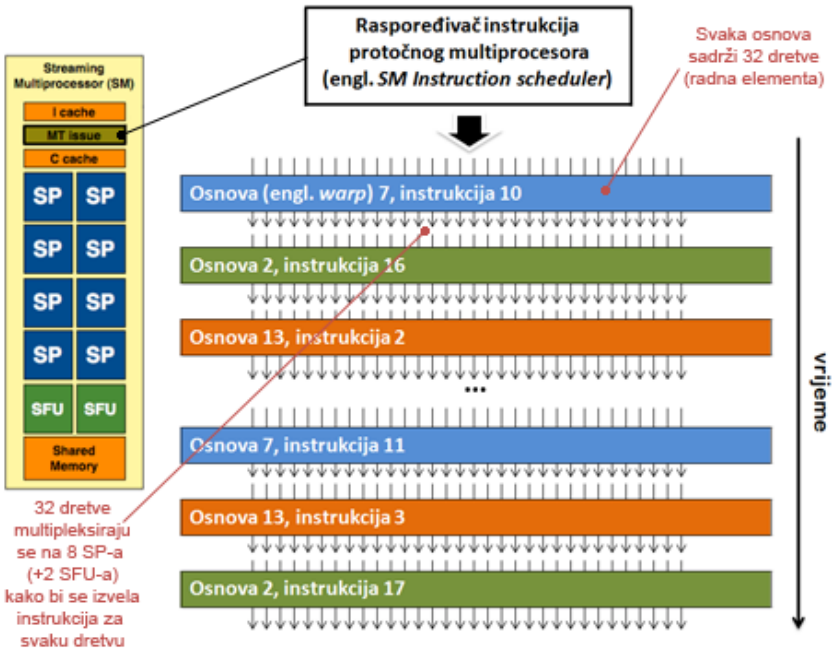



**Slika 2.5:** *Računska jedinica (protočni multiprocesor) - arhitektura NVIDIA GT200*

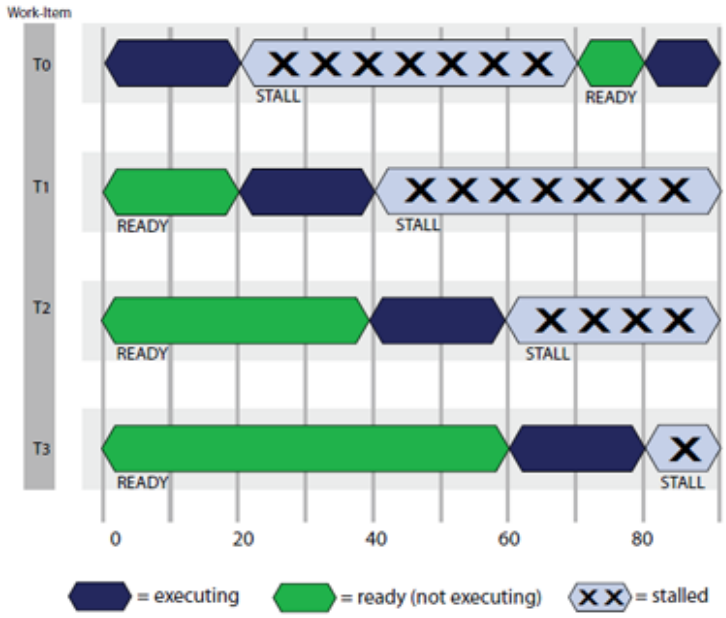
Prilikom GPGPU programiranja uz napisani program koji će se izvesti na grafičkoj kartici, programer mora definirati i broj dretvi koje će taj program izvesti. Sve dretve izvode iste instrukcije, međutim iz globalne memorije dohvaćaju različite podatke pa se stoga iste instrukcije izvode paralelno, ali manipuliraju različitim podacima. Broj dretvi može biti i veći od broja protočnih procesora koje grafička kartica posjeduje,

pa stoga grafičke kartice posjeduju jednu ili više jedinica koje su odgovorne za raspodjelu instrukcija protočnim procesorima, kao što su spomenute jedinice u prethodnom odlomku.

Mehanizam izvođenja instrukcija grupira dretve u tzv. **osnove** (engl. *warps*), u ATI-jevoj terminologiji koristi se naziv **valna fronta** (engl. *wavefront*). Svaka osnova (engl. *warp*) sadrži `N` dretvi, gdje `N` kod današnjih grafičkih kartica iznosi 32. Dakle definirani skup dretvi dijeli se u osnove od 32 dretve. Grafička kartica zatim slijedno dodijeljuje osnove računskim jedinicama, nakon čega se dretve iz osnove izvode na protočnim procesorima pripadajuće računske jedinice. Prilikom raspodjele dretvi protočnim procesorima, protočni procesori će izvesti najmanje jednu instrukciju, nakon čega se protočni procesori mogu dodijeliti drugoj osnovi (vidi sl. 2.6). Naime ukoliko dretve iz osnove pristupaju globalnoj memoriji, za što je potrebno 400 - 600 ciklusa da bi adresirani podatak bio raspoloživ, za to vrijeme dretve na protočnim procesorima ne mogu raditi ništa već nepotrebno zauzimaju raspoložive resurse. Upravo da bi se izbjegla spomenuta memorijska latencija, grafička kartica dodijeljuje protočne procesore, odnosno računsku jedinicu, drugoj osnovi, dok će prvoj osnovi resursi biti ponovno dodijeljeni nakon što zatraženi podaci iz memorije budu raspoloživi (vidi sl. 2.7).




**Slika 2.6:** *Izvođenje na računskoj jedinici.*




**Slika 2.7:** *Izvođenje radnih elemenata na jednom protočnom procesoru [7].*

Primjerice, neka je implementirano rješenje koji ima 8 aritmetičkih instrukcija, nakon čega slijedi jedna memorijska instrukcija koja zahtjeva 400 ciklusa za potpuno izvršavanje, dok je za aritmetičku instrukciju potrebno 4 ciklusa. Dakle dok se čeka obavljanje memorijske instrukcije, za to vrijeme moguće je obaviti $400/4 = 100$ aritmetičkih instrukcija. Prema tome da bi se uspješno sakrila memorijska latencija uzrokovana memorijskom instrukcijom potrebno je najmanje $\lceil 100/8 \rceil = 13$ osnova, odnosno $\lceil 100/8 * 32 \rceil = 400$ dretvi [6].

# Poglavlje 3

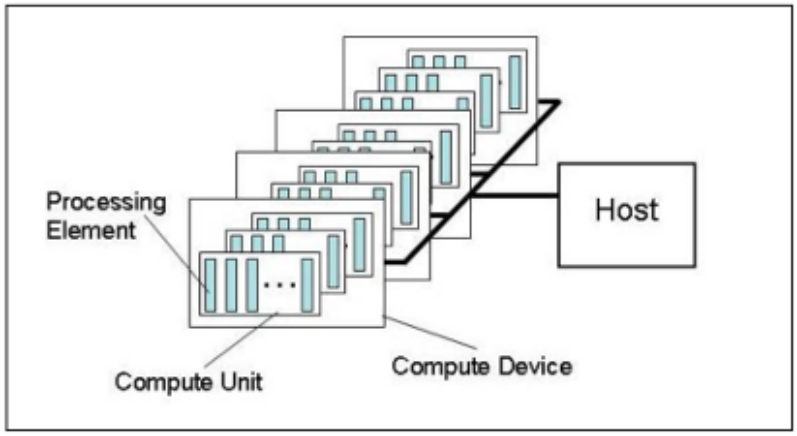
# OpenCL

OpenCL (Open Computing Language) je radni okvir za pisanje i izvršavanje programa na heterogenim platformama koje se sastoje od CPU-a, GPU-a i ostalih procesora [15]. OpenCL baš kao i CUDA, omogućava iskorištavanje performansi paralelnih arhitektura, kao što su naprimjer grafičke kartice. Za razliku od CUDA-e, koja je namijenjena samo grafičkim karticama proizvođača NVIDIA, OpenCL je otvoreni standard. Također OpenCL nastoji iskoristiti sve procesore dostupne na platformi, pa osim grafičkih kartica, OpenCL programi mogu se izvoditi i na CPU jedinicama, ali i na drugim procesorima kao što je primjerice procesor Cell.

Platforma na kojoj se izvode OpenCL programi obično se sastoji od CPU-a, radne memorije i uređaja koji podržavaju izvođenje OpenCL programa. Takav uređaj s mogućnošću izvođenja OpenCL programa može biti i sam procesor opće namjene (CPU), ali i ostale komponente računala koje imaju procesorsku moć i podršku za izvođenje OpenCL programa, a to su uglavnom današnje grafičke kartice (GPU).

Programer prilikom implementiranja programa koji će se izvoditi na računskim uređajima mora modelirati dvije komponente: **aplikaciju** (engl. *application*) i **jezgrenu funkciju** (engl. *kernel*). Zadatak aplikacije je pozivanje računskih uređaja i obavljanje pripremnih poslova za izvođenje jezgrenih funkcija na računskim uređajima. Aplikacija se programira u bilo kojem višem jeziku, dok se jezgrena funkcija programira u OpenCL-ovoj varijanti C jezika. Kontrola pokretanja i izvođenja OpenCL programa, tj. jezgrenih funkcija, kontrolira se programiranjem aplikacije. Spomenuti popratni poslovi koje je potrebno obaviti da bi se uopće OpenCL program izveo na računskom uređaju nazivaju se OpenCL zadacima, a u programskom kôdu aplikacije podržani su preko OpenCL aplikacijskog programskog sučelja. Implementirana **aplikacija**, tijekom

izvođenja, pod kontrolom je operacijskog sustava koji se izvodi na platformi, tj. na računalu domaćinu (engl. *host*). Računalo na kojem se izvodi aplikacija i preko kojeg se manipulira izvođenjem OpenCL programa naziva se **domaćinom (engl. *host*)**, a uređaji (jedinice) koji imaju sposobnost izvođenja OpenCL programa nazivaju se **računskim uređajima (engl. *compute devices*)**.




**Slika 3.1:** *Prikaz platforme [7].*

**Jezgrena funkcija** (engl. *kernel*) je najvažniji sloj OpenCL-a, naime jezgrena funkcija je programski kôd, odnosno implementacija algoritma koja će prevesti u strojni kôd te zatim izvesti na samom računskom uređaju (uređaj koji podržava OpenCL).

**OpenCL C** naziv je programskog jezika pomoću kojeg se piše izvorni kôd jezgrenih funkcija. Riječ je o varijanti jezika C99 s dodatcima kao što su npr. posebni tipovi podataka. Dakle sintaksa samog jezika slična je jeziku C, dodana su proširenja i dodatne ključne riječi pomoću kojih je moguće upravljati raznim elementima GPU-a i time potpuno iskoristiti raspoložive performanse računskih uređaja (posebice GPU-a).

**Program 3.1: 1D konvolucija implementirana u jezgrenoj funkciji OpenCL-a**

```
1 __kernel void Convolution1D(__global float* vector,
2                __global float* mask, const int radius,
3                __global float* resultVector)
4 {
5   uint idx = get_global_id(0);
6   float sum = 0.0f;
7   for(int i = - radius; i <= radius; i++){
8     float value = vector[idx + i];
9     sum += value * mask[radius - i];
```

```
10    }
11    result[idx] = sum;
12 }
```

Jedan OpenCL program sastoji se od jedne ili više jezgrenih funkcija. Ako je prisutno više jezgrenih funkcija, tada je moguće različite jezgrene funkcije proslijediti različitim računskim uređajima (engl. *compute devices*), nakon čega svaki uređaj izvodi svoju jezgrenu funkciju [4] [7].

Prevođenje izvornog kôda jezgrenih funkcija i pokretanje programa na računskim uređajima, poziva se i upravlja preko aplikacije. Već je spomenuto kako se aplikacija programira u bilo kojem višem jeziku koji ima implementirano OpenCL aplikacijsko programsko sučelje (API). U vrijeme pisanja rada definirana su aplikacijska programska sučelja za C, C++, C#, Java, Ruby i Python. Preko tih sučelja, programeru se pruža mogućnost kontrole odabira računskih uređaja na kojima će se izvršavati jezgrene funkcije, zatim prevođenje jezgrenih funkcija, tj. OpenCL programa, definiranje i alociranje ulaznih i izlaznih podataka itd.

Temeljni zadatak aplikacije može se ugrubo svesti na tri pod-zadatka:

1. prenijeti podatke do računskog uređaja

2. pokrenuti jezgrenu funkciju

3. dohvatiti podatke (rezultate) iz računskog uređaja, nakon što se izvede jezgrena funkcija

Tablica 3.1 prikazuje pseudokod, dok program 3.2 je primjer implementacije aplikacije s tipičnim OpenCL zadacima.

**Tablica 3.1:** *Primjer pseudokoda aplikacije*

| |
|---|
| (1) dohvat informacija o platformi i računskim uređajima |
| (2) kreiranje konteksta |
| (3) dohvat izvornog kôda jezgrenih funkcija |
| (4) prevođenje jezgrenih funkcija |
| (5) kreiranje naredbenog reda za odabrani računski uređaj |
| (6) kreiranje spremnika |
| (7) specificiranje jezgrene funkcije |
| (8) postavljanje argumenata jezg. funkcije |
| (9) definiranje globalnog i lokalnog radnog prostora |
| (10) dodavanje naredbe za pokretanje u naredbeni redak |
| (11) pokretanje |
| (12) dohvat rezultata |

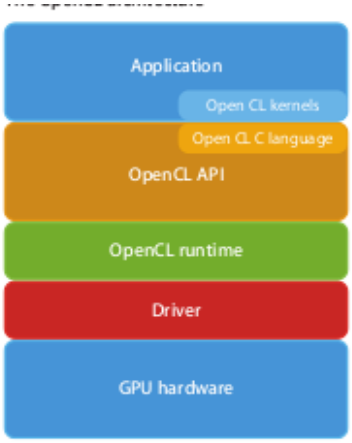
**Program 3.2: Aplikacija implementirana u C++.**

```
1  ...
2  try{
3    // dohvat informacija o platformi (1)
4    cl::vector< cl::Platform > platformList;
5    cl::Platform::get( &platformList );
6    cl_context_properties props[ 3 ] = { CL_CONTEXT_PLATFORM, (
        cl_context_properties)(platformList[0])(), 0 };
7
8    //kreiranje konteksta za GPU (2)
9    cl::Context* context = new cl::Context(CL_DEVICE_TYPE_GPU, props,
         NULL, NULL, NULL);
10
11   // dohvacanje liste OpenCL podrzanog sklopovlja
12   cl::vector<cl::Device> devices = context->getInfo<
        CL_CONTEXT_DEVICES>();
13
14   // dohvacanje izvornog koda jezgrenih funkcija iz datoteke (3)
15   // LoadKernelFile() - vlastita funkcija koja dohvaca izvorni kod
        iz datoteke
16   std::string kernelSource = LoadKernelFile();
17   cl::Program::Sources source( 1, std::make_pair(kernelSource_->
        c_str(), kernelSource_->length() + 1));
18
19   // prevodenje (4)
20   cl::Program* program = new cl::Program(*context, source);
21   program->build(devices);
22
```

```
23    //kreiranje naredbenog reda (5)
24    cl::CommandQueue* queue = new cl::CommandQueue(*context, devices
          [0], CL_QUEUE_PROFILING_ENABLE, NULL);
25
26    // kreiranje spremnika (6)
27    cl::Buffer* inputCL = new cl::Buffer(*context, CL_MEM_READ_ONLY |
           CL_MEM_COPY_HOST_PTR, inputLength * sizeof(float),  input,
          NULL);
28
29    cl::Buffer* maskCL = new cl::Buffer(*context, CL_MEM_READ_ONLY |
          CL_MEM_COPY_HOST_PTR, radius * 2 * sizeof(float),  mask, NULL)
          ;
30
31    cl::Buffer* resultCL = new cl::Buffer(*context, CL_MEM_WRITE_ONLY
          , resultLength * sizeof(float), NULL, NULL);
32
33    // specificiranje jezgrene funkcije (7)
34    cl::Kernel* kernel = new cl::Kernel(*program,                  ,
          NULL);
35
36    // postavljanje argumenata jezgrene funkcije (8)
37    kernel->setArg(0, *inputCL);
38    kernel->setArg(1, *maskCL);
39    kernel->setArg(2, radius);
40    kernel->setArg(3, *resultCL);
41
42    // definiranje NDRange-a (9)
43    cl::NDRange* globalRange = new cl::NDRange(resultLength); //
          globalni radni prostor
44    cl::NDRange* localRange = new cl::NDRange(64); // lokalni radni
          prostor
45
46    // pokretanje izvodenja (10)(11)
47    queue->enqueueNDRangeKernel(*kernel, cl::NullRange, *globalRange,
           *localRange, NULL, &event1);
48
49    // pricekaj kraj izvodenja
50    queue->finish();
51
52    // dohvat rezultata (12)
53    queue->enqueueReadBuffer(*resultCL, CL_TRUE,0, resultLength *
          sizeof(float), result);
54
55    // oslobadanje memorije
56    ...
57 }
58 catch(cl::Error err)
59 {
60    // oslobadanje memorije
61    ...
```

```
62
63    // proslijedi iznimku
64    throw err;
65 }
66 ...
```

Ispod OpenCL aplikacijskog programskog sučelja (vidi sl. 3.2), nalazi se **OpenCL runtime** (program koji izvodi OpenCL zadatke tijekom izvođenja aplikacije) čiji je zadatak prijenos podataka do GPU-a, prevođenje jezgrenih funkcija u strojni kôd, prijenos strojnog koda do GPU-a te dohvat rezultata nakon što se strojni kôd izvrši na računskom uređaju.




**Slika 3.2:** *Prikaz slojeva OpenCL-a*

Slojevitost OpenCL prikazana je na slici sl. 3.2. Na vrhu se nalazi aplikacija, aplikacija koristi OpenCL API sučelje kako bi se ostvarila veza s računskim uređajem.

## 3.1 Programiranje jezgrenih funkcija

Jezgrena funkcija izvodi se na uređaju kao funkcija višedimenzionalnih domena indeksa[1]. Svaki element domene predstavlja jednu dretvu koja će biti izvršena na protočnom procesoru i svaka dretva izvodi jezgrenu funkciju. Primjerice za jezgrenu funkciju iz programa 3.1, neka je definirana dvodimenzionalna domena $5 \times 5$. Svaka

[1] OpenCL podržava 1D, 2D i 3D domene.

od definiranih 25 dretvi izvršit će strojni kôd koji je nastao prevođenjem jezgrene funkcije. Postavlja se pitanje, postoji li uopće paralelizacija algoritma, tj. neće li sve dretve izvršiti isti kôd? Sve dretve izvode isti kôd, tj. isti skup naredbi, međutim svaka dretva će dohvatiti drugačiji skup podataka i nad takvim skupom podataka bit će primijenjene instrukcije jezgrene funkcije. Konkretno za gornji primjer, svaka od 25 dretvi dohvaća različite podatke nad kojima se zatim primjenjuju instrukcije strojnog kôda koji je zajednički svim dretvama. Takva vrsta paralelizacije naziva se **paralelizacija podataka (engl. *data-parallelism*)**.

U OpenCL terminologiji naziv za dretvu je **radni element (engl. *work-item*)** dok se ukupan broj indeksa (kardinalni broj domene ili broj radnih elemenata) naziva **globalna veličina radnog prostora (engl. *global work-size*)**. Sam prostor indeksa (dretvi, tj. radnih elemenata), tj. elemente domene moguće je grupirati u **radne grupe (engl. *work-groups*)**. U odjeljku 2.1 spomenuto je kako svaki radni element (dretva) može pristupiti samo vlastitim registrima opće namjene i također spomenuta je tzv. lokalna (dijeljena) memorija, preko koje radni elementi mogu relativno brzo izmjenjivati podatke. Upravo oni radni elementi koji se nalaze u istoj **radnoj grupi** bit će izvedeni na istoj računskoj jedinici (protočnom multiprocesoru), čime će imati osiguran pristup lokalnoj memoriji koju sadržava ta računska jedinica. Dakle jedna od prednosti grupiranja radnih elemenata u radne grupe je iskorištavanja prednosti lokalne memorije kako bi se poboljšale performanse izvođenja. Naime ukoliko se javlja potreba dijeljenja podataka između radnih elemenata, tada valja takve radne elemente smjestiti u istu radnu grupu, kako bi mogli podatke dijeliti preko lokalne memorije, koja je višestruko brža od globalne memorije.

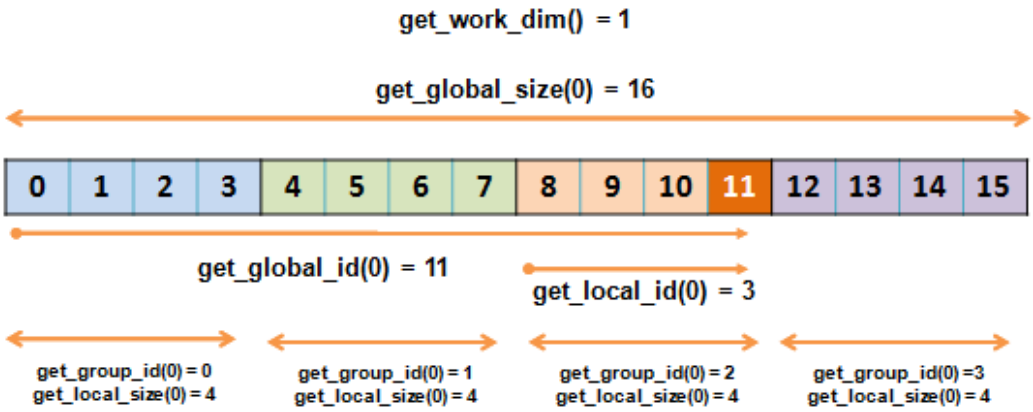
Veličina radne grupe (broj radnih elemenata u grupi) naziva se **lokalnim radnim prostorom (engl. *local work-size*)**. Valja napomenuti kako se postavljena veličina lokalnog radnog prostora odnosi na sve radne grupe, tj. nije moguće definirati radne grupe s različitim brojem radnih elemenata. Primjerice, neka je `globalRange` globalni radni prostor, neka je `globalRange` = 128 i neka je `localRange` lokalni radni prostor. Definiranjem `localRange` = 16, definirano je `globalRange`/`localRange` radnih grupa, što je 8 radnih grupa, pri čemu svaka grupa sadrži `localRange` radnih elemenata. Dakle radni elementi koje se nalaze u istoj grupi mogu ostvariti suradnju (razmjenu podataka) s drugim radnim elementima te iste grupe.

Unutar programskog kôda jezgrene funkcije moguće je jedinstveno identificirati

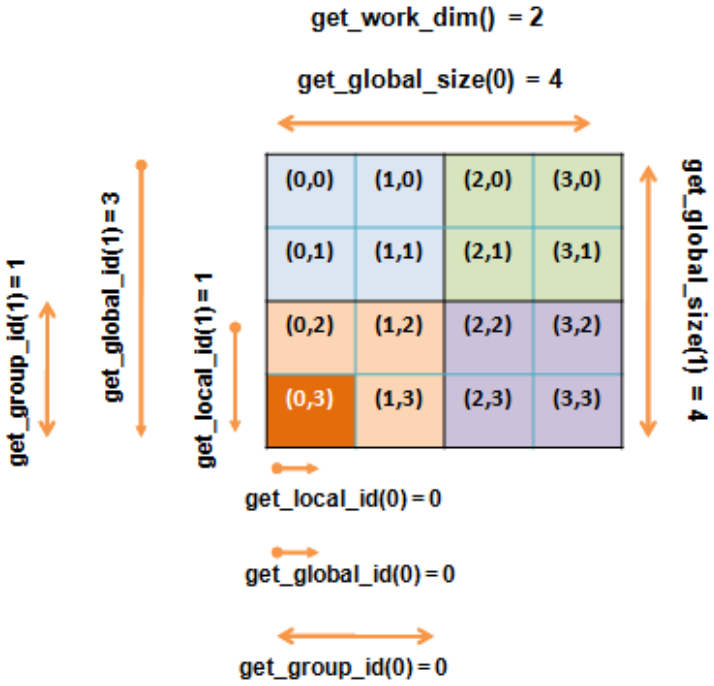
svaki radni element. Pozivom ugrađene funkcije `get_global_id(int dimension)`[2] dohvaća se indeks (identifikator) radnog elementa u globalnom radnom prostoru, dok se naredbom `get_local_id(int dimension)` dohvaća indeks (identifikator) radnog elementa u radnoj grupi, tj. unutar lokalnog radnog prostora (vidi sl. 3.3). Saznanje o indeksu radnog elemeneta (dretve) moguće je iskoristiti za izračun memorijskih adresa za dohvat skupa podataka nad kojima će radni element izvršiti jezgrenu funkciju (program 3.1).

Sl. 3.3a i sl. 3.3b slikovito prikazuju globalni radni prostor koji sadrži 16 radnih elemenata te opisuju funkcije za dohvat informacija o radnom prostoru. Svaka „kućica" prikazuje jedan radni element, a svaka radna grupa je različito obojana. Radni element s indeksom 11, odnosno (0,3), odgovara radnom elementu koji poziva funkcije (funkcije za dohvat informacija o radnom prostoru) i za kojeg vrijede vrijednosti prikazane na slici. Tablica 3.2 dodatno pojašnjava funkcije prikazane na slici 3.3.

[2]Funkcija kao argument prima broj iz skupa {0, 1, 2} koji označava željenu dimenziju (0 - prva dimenziju, 1 - druga dimenzija, 2 - treća dimenzija). Npr. neka je definiran dvodimenzionalni globalni radni prostor, tada get_global_id(0) dohvaća x-koordinatu, a get_global_id(1) y-koordinatu radnog elementa.




(a)




(b)

**Slika 3.3:** *Prikaz radnog prostora s opisom funkcija za dohvat informacija o radnom prostoru. 1D radni prostor prikazan je na (a), dok (b) prikazuje 2D radni prostor.*

**Tablica 3.2:** *Opis funkcija prikazanih na sl. 3.3.*

| | |
|---|---|
| `get_work_dim()` | dimenzija radnog prostora |
| `get_group_id(uint dimidx)` | identifikator (indeks) radne grupe |
| `get_global_id(uint dimidx)` | identifikator (indeks) radnog elementa u globalnom radnom prostoru |
| `get_local_id(uint dimidx)` | identifikator (indeks) radnog elementa u lokalnom radnom prostoru |
| `get_global_size(uint dimidx)` | veličina globalnog radnog prostora |
| `get_local_size(uint dimidx)` | veličina lokalnog radnog prostora |

## 3.2 Izvođenje OpenCL programa na GPU-u

### Izvođenje jezgrene funkcije

Nakon što se napiše izvorni kôd jezgrene funkcije (ili više njih), potrebno je pozvati prevođenje OpenCL C jezika u strojni kôd. Ako je prevođenje uspješno završeno tada je potrebno postaviti argumente jezgrenim funkcijama. Argumenti su obično pokazivači koji pokazuju na određene memorijske adrese u globalnoj memoriji. Valja napomenuti kako programer ne može znati na kojoj memorijskoj adresi se nalaze podaci u memoriji grafičke kartice, nakon što se podaci prenesu iz radne memorije računala. OpenCL runtime prenosi podatke do grafičke kartice, pa stoga ima i saznanja o točnim memorijskim lokacijama podataka te će OpenCL runtime postaviti vrijednosti pokazivača na valjane iznose. Ipak da bi OpenCL runtime znao koje podatke treba prenijeti iz radne memorije u memoriju grafičke kartice, programer u aplikaciji kreira tzv. memorijske objekte u kojima specificira pokazivač na podatak (pokazivač na prvi element ako je riječ o nizu podataka, slici i sl.), specificira veličinu podataka u oktetima i definira dodatne opcije koje definiraju kako podatke treba prenijeti do računskog uređaja (u ovom slučaju GPU-a).

U odjeljku 2.3 opisan je način izvođenja instrukcija, gdje se spominje pojam osnova (engl. *warp*), odnosno valna fronta (engl. *wavefront*). Valja istaknuti kako svi radni elementi iz iste osnove logički izvršavaju istu naredbu „paralelno” odnosno u „istom” trenutku, ipak na fizičkoj razini stvari se odvijaju malo drugačije. Nakon što je računskoj jedinici dodijeljena radna grupa, tada se računskoj jedinici dodijeljuje

po N slijednih radnih elemenata iz radnog prostora, gdje je N konstanta koja ovisi o samoj grafičkoj kartici. Skup od N slijednih radnih elemenata je osnova. U današnjim grafičkim kartica N iznosi uglavnom 32, kao npr. kod ATI Radeon HD 5670 i NVIDIA GT240. Međutim računske jedinice u današnjim grafičkim karticama ne sadrže 32 protočna procesora, već manje, pa se na sklopovskoj razini radni elementi iz osnove prilikom izvođenja multipleksiraju na raspoloživi broj protočnih procesora na računskoj jedinici. Stoga nije potpuno ispravno tvrditi da će se instrukcija za sve radne elemente unutar osnove izvršiti u istom trenutku, već će se fizički izvršiti kroz nekoliko slijednih koraka (broj koraka ovisi o broju protočnih procesora računske jedinice), ipak gledano s logičke razine osnova se može smatrati temeljnom bazom.

Važnost osnova najbolje se može vidjeti kod izvođenja grananja (if, switch ...). Ako je u jezgrenoj funkciji prisutno grananje tada može doći do divergiranja radnih elemenata. Naime moguća je situacija da određeni radni elementi iz osnove ispunjavaju uvjet grananja, dok neki ne ispunjavaju takve uvjete. U ovom radu već je spomenuto kako se ista instrukcija izvodi za sve radne elemente iz osnove, pa se postavlja pitanje kako sklopovlje rješava situaciju u kojoj bi radni elementi iz osnove trebali izvoditi različite instrukcije.

U takvim situacijama GPU se ponaša kao da svi radni elementi izvode instrukciju, međutim oni radni elementi koji ne bi trebali izvoditi instrukciju su maskirani, što znači da će fizički resursi (protočni procesori) biti dodijeljeni svim radnim elementima iz osnove, uključujući i maskirane, te će se ponašati kao da su zauzeti čak i ako je protočni procesor dodijeljen maskiranom radnom elementu. Takav proces u kojem radni elementi unutar osnove divergiraju naziva se **serijalizacija** (engl. *serialization*) [2] [4] [7]. Primjerice ako bi unutar osnove svi radni elementi divergirali, tj. ako bi svaki radni element trebao izvoditi različitu instrukciju, uz pretpostavku da osnova sadrži 32 radna elementa, tada će se osnova izvesti kroz 32 koraka (vidi program 3.3), umjesto jednog kao što bi bilo ako bi svi radni elementi izvodili istu naredbu (vidi program 3.4).

**Program 3.3: Grananje koje uzrokuje divergenciju radnih elemenata.**

```
1  ...
2
3  uint idx = get_global_id(0);
4  uint a = idx % 32;
5
6  if(a == 0) a = 0;
7  else if (a == 1) a = a + 1;
8  else if (a == 2) a = a * 2;
```

```
9   ...
10  else if (a == 32) a = 32;
11
12  ...
```

**Program 3.4: Grananje koje ne uzrokuje divergenciju radnih elemenata.**

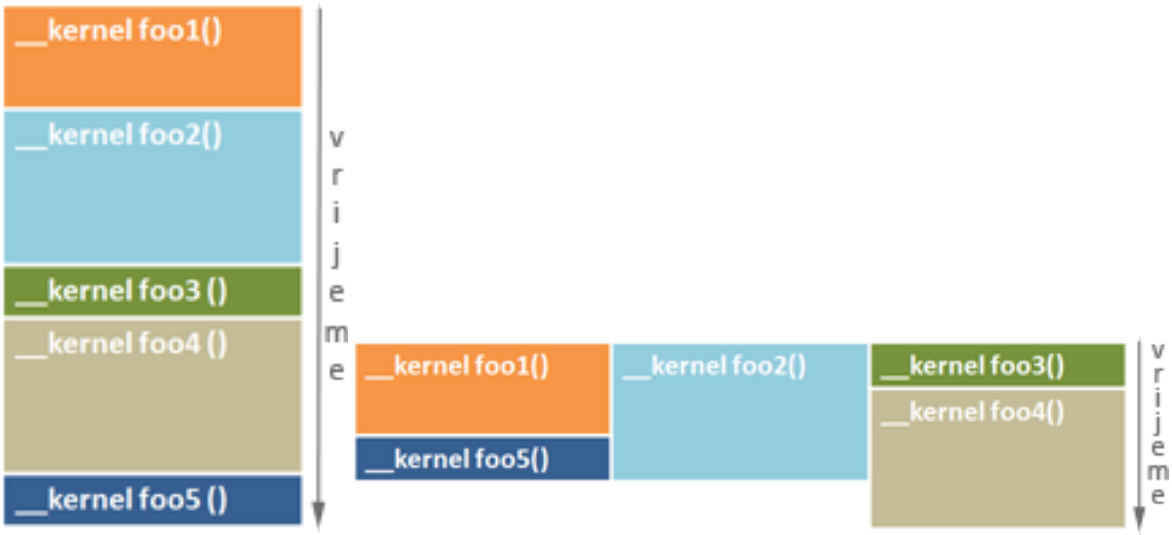
```
1   ...
2
3   uint idx = get_global_id(0);
4   uint a = idx \ 32;
5
6   if(a == 0) a = 0;
7   else if (a == 1) a = a + 1;
8   else if (a == 2) a = a * 2;
9   ...
10  else if (a == 32) a = 32;
11
12  ...
```

Nepažljivim implementiranjem algoritama koji zahtijevaju grananje mogu se značajno oslabiti performanse izvođenja, stoga programer treba izbjegavati konstrukcije grananja koje će uzrokovati divergiranje radnih elemenata unutar iste osnove.

## Konkurentno izvođenje jezgrenih funkcija

Konkurentno izvođenje jezgrenih funkcija podržavaju samo novije grafičke kartice, tj. arhitektura kodnog imena NVIDIA Fermi [1]. U arhitekturama NVIDIA GT240, ATI Evergreen i starijim, različite jezgrene funkcije izvode se serijski.

Konkurentno izvođenje različitih jezgrenih funkcija omogućava izvođenje različitih jezgrenih funkcija paralelno na različitim računskim jedinicama, čime je moguće ostvariti tzv. paralelizaciju zadataka (engl. *task-parallelism*).

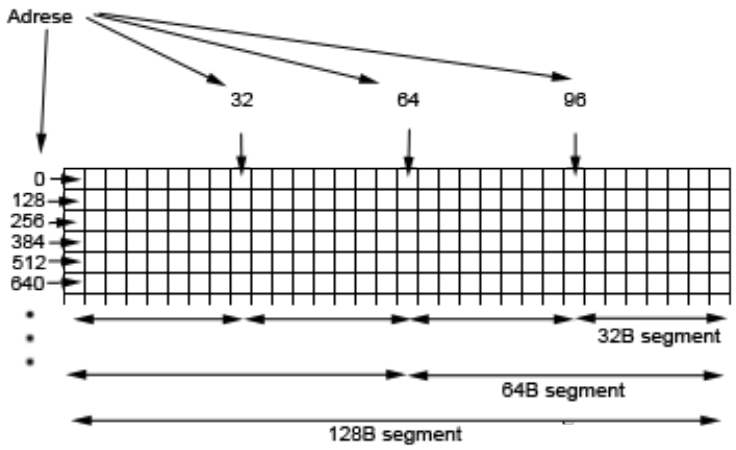



**Slika 3.4:** *Prikaz izvođenja različitih jezgrenih funkcija ovisno o arhitekturi. Serijsko izvođenje (a) na arhitekturi NVIDIA GT200 i ATI Evergreen, te konkurentno izvođenje (b) na arhitekturi NVIDIA Fermi.*

## Sjedinjavanje memorijskih pristupa

Kako bi se maksimalno iskoristile performanse grafičke kartice, prilikom programiranja jezgrene funkcije treba voditi računa o sjedinjavanju pristupa globalnoj memoriji. Ukoliko radni elementi raštrkano pristupaju memorijskim adresama, tada dolazi do značajnog gubitka performansi. Idealna situacija pristupa je kada svi radni elementi iz iste osnove pristupaju slijednim memorijskim adresama. Pažljivim modeliranjem jezgre moguće je postići idealan pristup memoriji.

Globalna memorija podijeljena je na segmente od 32, 64 i 128 okteta i svi segmenti poravnati su na adrese koji su višekratnici broja 32, 64 i 128. Najniža (prva) memorijska adresa unutar segmenta veličine 32 okteta je višekratnik broja 32, dok prva adresa segmenta veličine 64 okteta je višekratnik broja 64, a prva adresa segmenta veličine 128 okteta je višekratnik broja 128. Ovakav model memorijskog pristupa sadrže grafičke kartice proizvođača NVIDIA koje imaju podršku za „Compute Capability" 1.2 ili viši, dok ATI-jeve grafičke kartice serije Evergreen sadrže samo 128-oktetne segmente.




**Slika 3.5:** *Shematski prikaz globalne memorije s prikazom segmenata.*

Prilikom pristupa radnih elemenata (iz iste osnove) globalnoj memoriji, ukupan broj memorijskih transakcija jednak je broju segmenata kojima je pristupljeno. Ako svi radni elementi iz iste osnove pristupaju adresama koje se nalaze unutar jednog segmenta, tada će se obaviti jedna memorijska transakcija. Međutim ako radni elementi iz osnove pristupaju adresama koje su međusobno razmaknute, tada će biti potrebno onoliko transakcija koliko je različitih segmenta. Ako su adrese razmaknute primjerice 128 okteta i ako osnova sadrži 32 radna elementa, tada su potrebne 32 memorijske transakcije. Čak i ako su adrese kojima radni elementi pristupaju slijedne, ali ne padaju sve u isti segment, opet je broj memorijskih transakcija jednak broju segmenata kojima je pristupljeno (vidi sl. 3.6c). Prilikom prijenosa uvijek se prenosi najmanji mogući segment [4].

Arhitektura grafičkih kartica proizvođača NVIDIA dijeli osnovu u dvije **pod-osnove** (engl. *half-warps*), svaka pod-osnova sadrži 16 radnih elemenata. Radni elementi iz iste pod-osnove pristupaju zajedno globalnoj memoriji, ali i lokalnoj memoriji [2].

Prilikom prijenosa podataka s računala domaćina u globalnu memoriju GPU-a, primjerice nizova podataka (polja), OpenCL runtime alocira u globalnoj memoriji memorijski prostor s početnom adresom koja je višekratnik broja 128 i smješta prvi element polja na tu adresu, dok će ostali elementi polja biti poredani u slijedne memorijske adrese iza početne adrese [4].


**Slika 3.6:** *Na slici (a) pristupu su sjedinjeni za što je potrebna jedna memorijska transakcija, (b) prikazuje pomak za 2 elementa udesno, što također rezultira samo jednom transakcijom od 128 okteta, slika (c) je prikaz neporavnatog pristupa koji u ovom slučaju rezultira s dvijema memorijskim transakcijama [4].*

## Lokalna memorija

Pristup globalnoj memoriji puno je sporiji od pristupa lokalnoj memoriji, stoga valja u lokalnu memoriju pohraniti podatke koji bi se inače često dohvaćali iz globalne memorije.

Lokalna memorija organizirana je u **banke** (engl. *banks*), arhitektura ATI Evergreen sadrži 32 banke, dok arhitektura NVIDIA GT200 sadrži 16 banaka (vidi sl. 3.7). Svaka banka je veličine 1 kB, tj. svaka banka sadrži 256 redaka (memorijskih lokacija) veličine 4 okteta, što čini ukupno 16 kB na arhitekturi NVIDIA GT200 [2], odnosno 32 kB na arhitekturi ATI Evergreen [7].

Već je spomenuto kako svaka računska jedinica sadrži svoju lokalnu memoriju. Radni elementi iz iste radne grupe izvode se na istoj računskoj jedinici između ostalog i jer dijele zajedničku lokalnu memoriju. Veličina lokalne memorija prilično je oskudna i treba biti pažljiv s njenim alociranjem. Naime preveliko iskorištavanje lokalnog memorijskog prostora, također može značajno utjecati na performanse izvođenja jezgrene funkcije. U odjeljku 2.3 opisano je kako grafička kartica dodijeljuje fizičke resurse osnovama i kako skriva memorijsku latenciju. Ako su radni elementi u stanju bez posla zbog memorijske latencije, tada se računska jedinica s protočnim procesorima dodijeljuje drugoj osnovi. Za uspješno savladavanje memorijske latencije potrebno je modelirati jezgrene funkcije koje će imati što više radnih elemenata, a time i više osnova, kako bi se maksimalno iskoristili fizički resursi grafičke kartice.

Radne grupe se na računskoj jedinici izvode konkurentno, čime se također smanjuje latencija. Neka je `nWorkGroups` broj radnih grupa, a `M` broj računskih jedinica tada će na svakoj računskoj jedinici biti izvedeno `numWorkGroups/M` radnih grupa. Primjerice, neka je zadovoljena pretpostavka `numWorkGroups/M` $>$ `2`, u takvim slučajevima računska jedinica može raditi na drugoj radnoj grupi ako se prvotna radna grupa nalazi u stanju bez posla, primjerice dok čeka na barijeri (vidi program 3.5). Za ilustraciju neka je implementirana jezgrena funkcija (vidi program 3.5) koja zahtjeva 10 kB lokalne memorije (ključna riječ `__local` označava korištenje lokalne memorije). Neka maksimalna raspoloživa količina lokalne memorije, na grafičkoj kartici na kojoj će se izvoditi rješenje, iznosi 16 kB (napomena: svaka računska jedinica sadrži 16 kB lokalne memorije, dok grafička kartica ukupno sadrži $16 * M$ kB lokalne memorije).

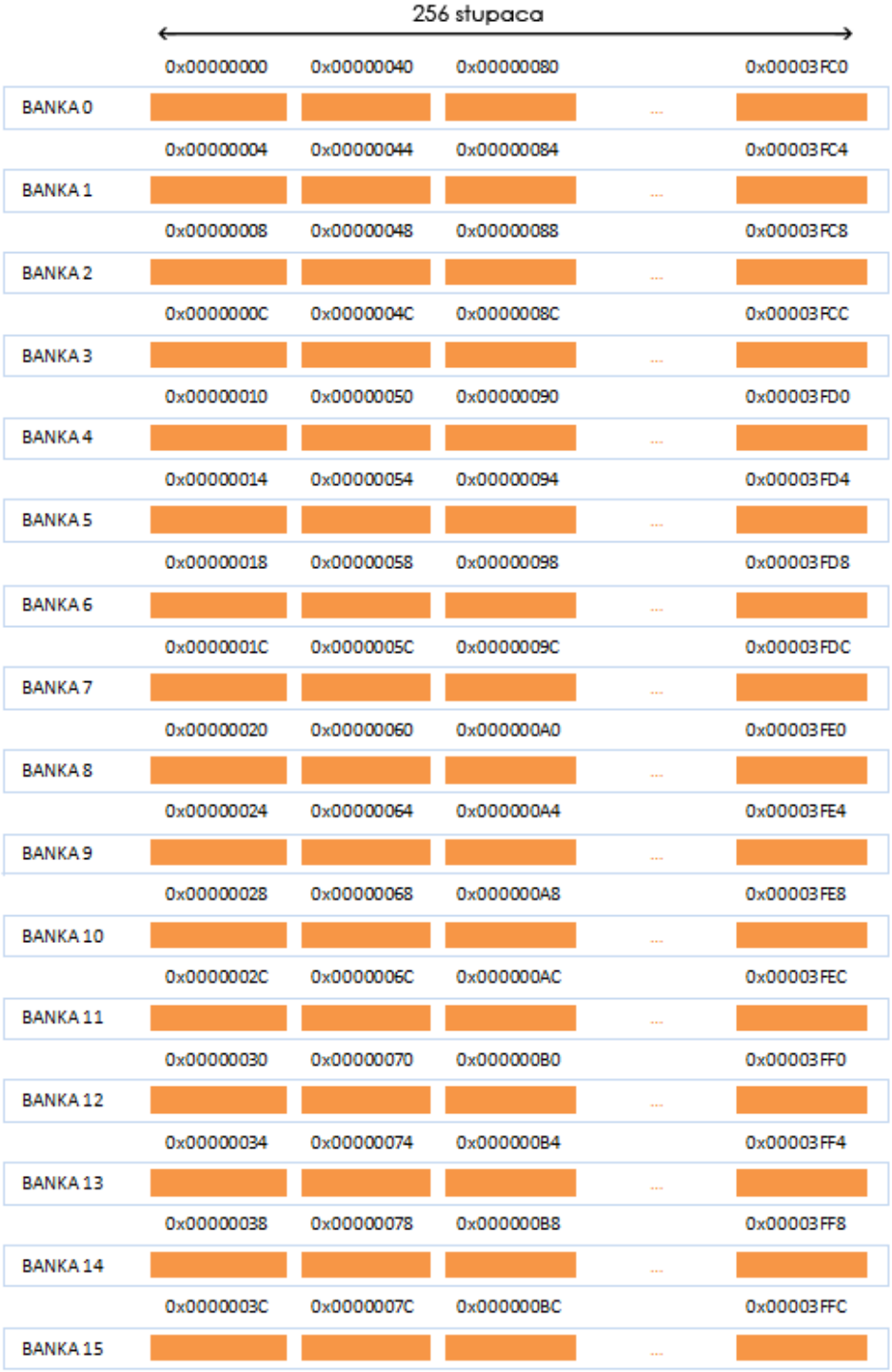
Problem jezgrene funkcije Foo() (vidi 3.5) je u tome što rezervira previše lokalne memorije, pa nije moguće izvoditi više radnih grupa konkurentno na računskoj jedinici.

Naime jezgrena funkcija zahtjeva 10 kB lokalne memorije, a na raspolaganju je najviše 16 kB lokalne memorije, prema tome na računskoj jedinici moguće je konkuretno izvoditi `w` radnih grupa, pri čemu mora vrijediti 10 kB * `w` <= 16 kB. Nejednadžba vrijedi za `w = 1`, stoga se radne grupe na računskoj jedinici izvode serijski, čime je onemogućeno sakrivanje latencija.

**Program 3.5: Primjer za lokalnu memoriju.**

```
1  __kernel Foo()
2  {
3    uint idx = get_global_id(0);
4
5    // rezerviranje potrebne lokalne memorije
6    __local int data[2560];
7
8    // inicijalizacija
9    data[idx] = (int) idx;
10
11   // barijera - radni elementi cekaju da svi radni elementi iz
         radne grupe
12   // izvedu ovu naredbu, nakon sto radni elementi izvedu ovu
         naredbu,
13   // tada mogu krenuti dalje s izvodenjem jezgrene funkcije
14   barrier(CLK_LOCAL_MEM_FENCE);
15
16   // dodatni posao
17   ...
18 }
```

Do sličnog problema može doći i zbog memorijske latencije. Primjerice neka jezgrena funkcija rezervira 10 kB lokalne memorije kao i program 3.5, međutim neka je izvornom kôdu dodana naredba koja pristupa globalnoj memoriji. Pristupi globalnoj memoriji zahtijevaju 400 - 600 ciklusa, pa računska jedinica sakriva memorijsku latenciju dodjeljivanjem resursa drugoj osnovi. Ako radna grupa sadrži malo radnih elemenata, tada radna grupa možda neće imati dovoljno osnova za potpuno sakrivanje memorijske latencije. U takvim slučajevima računska jedinica dodijeljuje svoje resurse dugoj radnoj grupi, tj. osnovama iz druge radne grupe, međutim kako radna grupa zahtjeva 10 kB lokalne memorije, tada nije moguće izvoditi više radnih grupa konkurentno, pa resursi ne mogu biti dodijeljeni drugoj radnoj grupi dok prvotna radna grupa ne obavi cijeli svoj posao. Samim time resursi grafičke kartice neće biti potpuno iskorišteni. Rješenje ovog problema je smanjiti zauzeće lokalne memorije po radnoj grupi ili povećati veličinu radne grupe (više radnih elemenata).




**Slika 3.7:** *Prikaz adresnog prostora lokalne memorije (arhitektura NVIDIA GT200).*

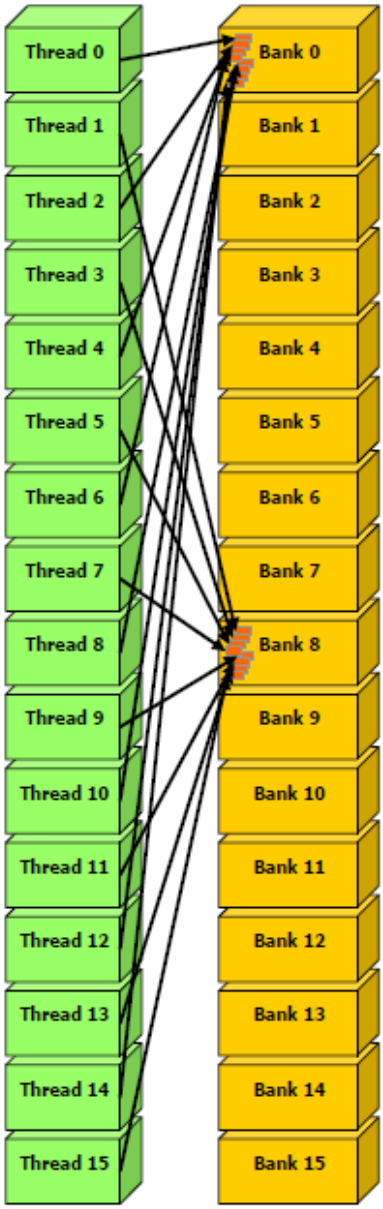
## Konflikti banaka

Ako `n` radnih elementa iz osnove pristupa (čitanje ili pisanje) `n` različitim adresama, koje padaju u `n` različitih banaka, tada su radni elementi posluženi „trenutno". Međutim ako primjerice dva radna elementa pristupaju istoj banci (adrese se nalaze u istoj banci), tada dolazi do **konflikta banaka** (engl. *bank conflict*) i tada se pristup serijalizira na onoliko pristupa koliko je potrebno da bi se razriješili konflikti [4], u ovom slučaju na dva pristupa. Ako je provedeno `n` serijaliziranih pristupa, što se naziva n-struki konflikt banaka (engl. *n-way bank conflict*), tada se propusnost smanji za faktor `n`. Do konflikta ne dolazi ako više radnih elementa zahtjeva čitanje iste riječi (engl. *word*) iz lokalne memorije, tada dolazi do razašiljana (engl. *broadcast*) podataka [4].

Kvalitetnim modeliranjem jezgrene funkcije moguće je izbjeći konflikte spremnika, u protivnom (vidi program 3.6 i sl. 3.8) resursi grafičke kartice nisu optimalno iskorišteni što rezultira padom performansi.

**Program 3.6: Primjer jezgrene funkcija koja uzrokuje osmerostruki konflikt banaka.**

```
1 __kernel BankConflict()
2 {
3   // rezerviranje potrebne lokalne memorije
4   uint idx = get_global_id(0);
5   __local int data[2560];
6
7   int dataIdx = (idx / 8) * 8 ;
8
9   // konflikt banaka
10  data[dataIdx] = 0;
11 }
```




**Slika 3.8:** *Prikaz osmerostrukog konflikta za program 3.6 (arhitektura NVIDIA GT200) [4].*

# Poglavlje 4

# Implementacija 1D konvolucije u OpenCL-u

Razvijeno je nekoliko implementacija 1D konvolucije sa separabilnim filtrima [11]. Algoritmi su optimizirani za izvođenje na GPU-u.

## 4.1 „Naivni" algoritam (Convolution_v1)

Prvi algoritam, tzv. „naivni" algoritam (vidi program 4.1), predstavlja modificirano rješenje implementacije konvolucije u jeziku C. Svaki radni element odgovoran je za izračun jednog izlaznog elementa (točke) konvolucije. Dakle svaki radni element dohvaća iz memorije vrijednosti elemenata potrebnih za izračun izlaznog elementa, ako je polumjer filtra $r$, takvih dohvata će biti $2 * r + 1$. Sam filtar nalazi se također u globalnoj memoriji pa će za dohvaćanje elemenata filtra biti potrebno još dodatnih $2 * r + 1$ pristupa. Nakon izračuna izlaznog elementa, svaki radni element pohranjuje rezultat u izlazno polje, za što je potreban jedan pristup globalnoj memoriji.

Izvođenje na grafičkom koprocesoru rezultirao je slabim performansama, upravo zbog učestalih redundantnih pristupa globalnoj memoriji.

**Program 4.1: Naivni algoritam konvolucije sa separabilnim filtrima.**

```
1 __kernel void ConvolutionRow(__global float* matrix,
2               const int nRows,
3               const int nCols,
4               __global float* horizMask,
5               const int radius,
6               __global float* result)
```

```
7  {
8    int idx = (int) get_global_id(0);
9    float sum = 0;
10   float value;
11
12   int startRowIdx = (idx / nCols) * nRows;
13
14   // horizontalno
15   for(int i = - radius; i <= radius; i++)
16   {
17     int idxElement = idx + i;
18
19     if(idxElement < startRowIdx || (idxElement > startRowIdx +
         nRows - 1)) value = 0;
20     else value = matrix[idxElement];
21
22     sum += value * horizMask[radius - i];
23   }
24
25   result[idx] = sum;
26 }
27
28 __kernel void ConvolutionColumn(__global float* matrix,
29                 const int nRows,
30                 const int nCols,
31                 __global float* vertMask,
32                 const int radius,
33                 __global float* result)
34 {
35   int idx = (int) get_global_id(0);
36   float sum = 0;
37   float value;
38
39   // vertikalno
40   for(int i = - radius; i <= radius; i++)
41   {
42     int idxElement = idx + i * nCols;
43
44     if(idxElement < 0 || (idxElement > (nRows*nCols - 1))) value =
         0;
45     else value = matrix[idxElement];
46
47     sum += value * vertMask[radius - i];
48   }
49
50   result[idx] = sum;
51 }
```

Osim učestalog pristupa globalnoj memoriji, gdje pristupi nisu sjedinjeni, implementacija

također ne iskorištava prednosti lokalne memorije, što rezultira niskom memorijskom propušnoću, što pak za posljedicu ima relativno dugo vrijeme izvođenja.

Prilikom poziva jezgrenih funkcija (`ConvolutionRow` i `ConvolutionColumn`), definiran je jednodimenzionalni radni prostor, veličina globalnog radnog prostora jednaka je veličini izlazne slike (broj točaka izlazne slike), a svakoj radnoj grupi dodijeljena su 64 radna elementa.

## 4.2 Algoritam Convolution_v3

Ova implementacija rješava probleme „naivnog" algoritma, naime uvodi se korištenje lokalne memorije. Korištenjem lokalne memorije spriječiti će se višestruki nepotrebni pristupi globalnoj memoriji pa će radni elementi umjesto pristupa globalnoj memoriji koja je spora, dohvaćati elemente ulazne slike iz lokalne memorije. Radni elementi iste radne grupe izvršavaju se na istoj računskoj jedinici pa prema tome radni elementi jedne radne grupe dijele i zajedničku lokalnu memoriju. Upravo dijeljenje lokalnog memorijskog prostora omogućava dijeljenje elemenata ulazne slike između radnih elemenata. Dakle temelja ideja je dohvatiti iz globalne memorije segment (blok) slike[1] te ga pohraniti u lokalnu memoriju, nakon čega se izračunava konvolucija nad tim segmentom slike. Kako se segment slike nalazi u lokalnoj memoriji, pristupi do memorijskih lokacija na kojima su pohranjene vrijednosti elemenata slike su puno brži od pristupa globalnoj memoriji.

### Jezgrena funkcija konvolucije po retcima

Neka lokalna veličina radnog prostora (veličina radne grupe, broj radnih elemenata radne grupe) iznosi `LINE_W` i neka je `radius` polumjer retčanog filtra. Postoji nekoliko mogućih rješenja, od koji je samo zadnje (3. način) optimalno.

**1. način** Neka svaki radni element dohvati iz globalne memorije vrijednost svog središnjeg elementa ulazne slike i pohrani tu vrijednost u lokalnu memoriju. Nakon što svi radni elemente radne grupe pohrane vrijednosti za svoje središnje elemente u lokalnu memoriju, tada će svaki radni elemente izračunati konvoluciju nad svojim

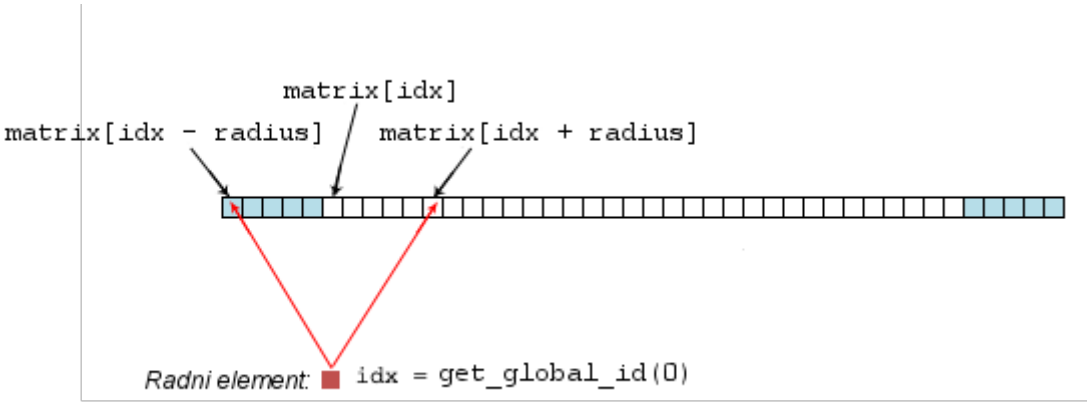
[1] Segment slike - nema veze sa fizičkim segmentima globalne memorije, već označava jedan dio ulazne slike

učitanim elementom i pohrani rezultat u izlazno polje. Problem ove ideje je u tome što nisu isključeni rubni elementi, naime konvolucija se računa i za rubne elemente, pa za rubne elemente filtar izlazi izvan prostora pohranjenog polja u lokalnoj memoriji.

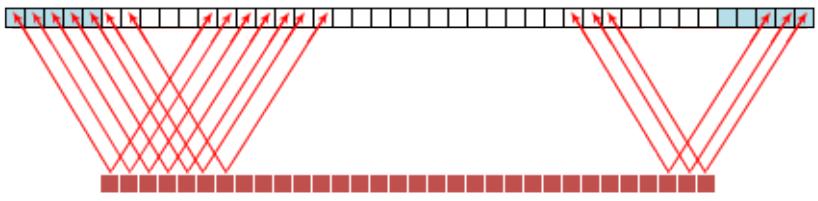
**2. način** Problem prve ideje može se riješiti preskakanjem izračunavanja konvolucije za početnih `radius` elementima i posljednjih `radius` elemenata segmenata. Odnosno ti radni elementi bi samo učitali vrijednost svog središnjeg elementa iz ulazne slike i pohranili ga u lokalnu memoriju, dok bi iz izračuna konvolucije bili isključeni. Time bi 2*`radius` radnih elemenata bilo besposleno, dok bi samo `LINE_W` - 2*`radius` bilo aktivno tijekom izračunavanja konvolucije. Ovakvo rješenje ne iskorištava sve protočne procesore, pa performanse GPU-a nisu potpuno iskorištene, jer su radni elementi, koji su odgovorni za rubne elemente segmenta ulazne slike, isključeni iz izračunavanja konvolucije, pa dolazi do divergencije radnih elemenata unutar osnove (vidi odjeljak 3.2).

**3. način** Neka je `idx` indeks radnog elementa u globalnom radnom prostoru, samim time i indeks središnjeg elementa u ulaznoj slici. Središnji element označava element ulazne slike nad kojim će radni element `idx` izračunati retčanu konvoluciju. Također `idx` određuje i indeks za pohranu rezultata u izlazno polje koje predstavlja izlaznu sliku. Radni element prije računanja retčane konvolucije dohvaća iz globalne memorije element `matrix[idx - radius]` (u daljnjem tekstu: „lijevo" čitanje) i element `matrix[idx + radius]` (u daljnjem tekstu: „desno" čitanje), gdje je `matrix` pokazivač na prvi element ulazne slike. Nakon što svi radni elementi iz iste radne grupe dohvate svoja dva elementa i pohrane ih u lokalnu memoriju, u lokalnoj memoriji će se nalaziti ukupno `LINE_W + 2*radius` slijednih elemenata ulazne slike[2], pa svaki radni element radne grupe može izračunati vrijednost izlaznog elementa za koji je odgovoran. Prvih `radius` elemenata i zadnjih `radius` elemenata polja koje predstavlja segment slike i koje je pohranjeno u lokalnoj memoriji su tzv. elementi „omotači".

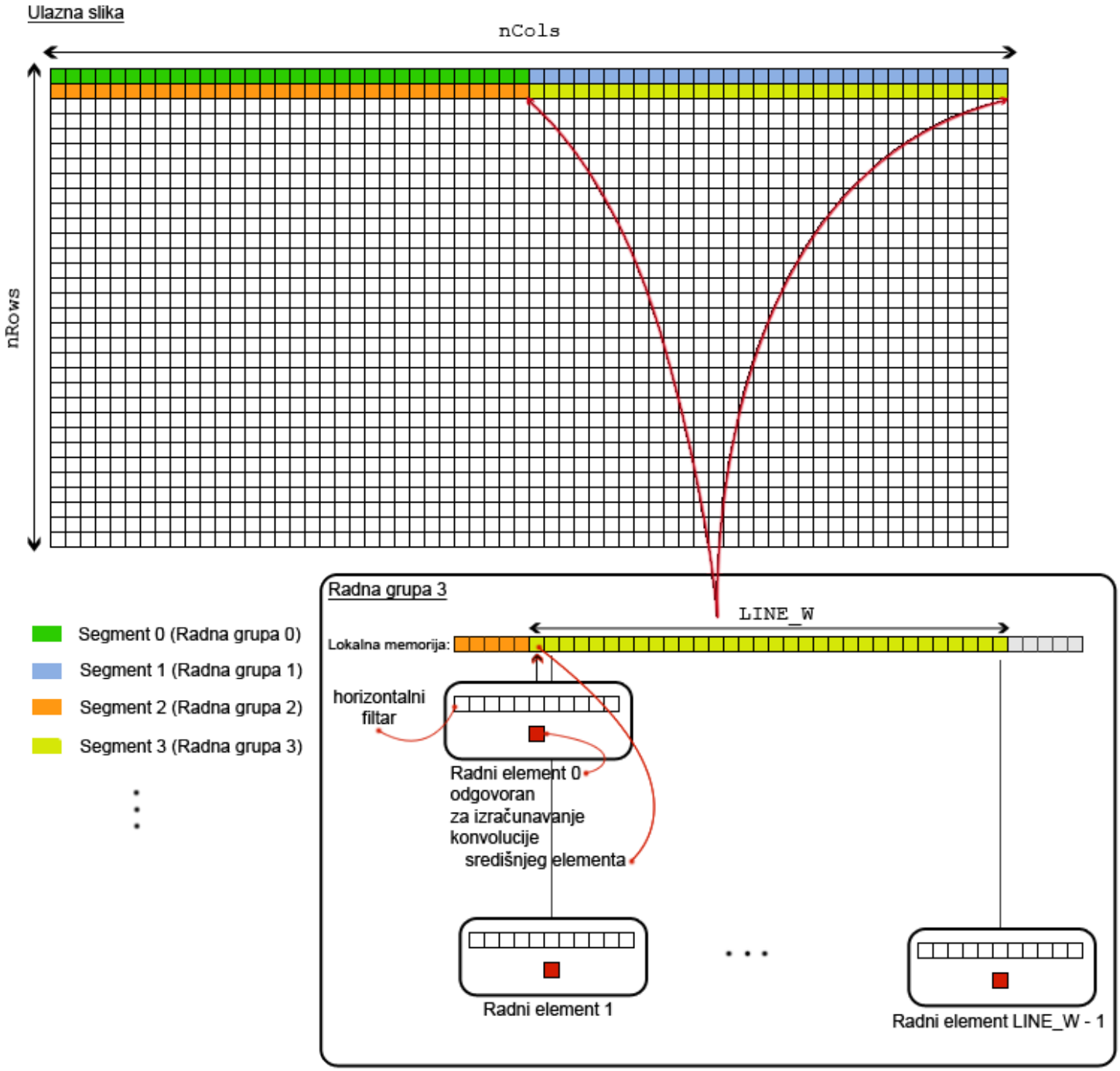
[2]Radni elementi osnove obaviti će 2*LINE_W čitanja, ali neki elementi ulazne slike učitavaju se dva puta, tj. preklapaju se čitanja između radnih elemenata. Ovakvo preklapanje nema utjecaja na performanse.




**Slika 4.1:** *Prikaz lijevog čitanja i desnog čitanja (za jedan radni element). Polumjer filtra iznosi 5.*



**Slika 4.2:** *Prikaz dohvata segmenta slike za konvoluciju po retcima. Radni elementi radne grupe dohvaćaju cijeli segment, nad kojim će obaviti konvoluciju. Veličina radne grupe je 32, a polumjer filtra iznosi 5.*




**Slika 4.3:** *Shematski prikaz rada jezgrene funkcije ConvolutionRow algoritma Convolution_v3.*

Sl. 4.3 opisuje rad jezgrene funkcije `ConvolutionRow`. Prikazan je jedan trenutak izvođenja radne grupe nazvane „Radna grupa 3". Prikazan je figurativan smještaj

ulazne slike u globalnoj memoriji, različiti segmenti slike su različito obojani te je prikazano koji radni element je odgovoran za koji središnji element iz lokalne memorije.

### Jezgrena funkcija konvolucije po stupcima

Jezgrena funkcija odgovorna za konvoluciju po stucima koristi slično rješenje kao i jezgrena funkcija za konvoluciju po retcima. Razlika je jedino u tome što se u lokalnu memoriju učitava stupac ulazne slike visine `LINE_H + 2*radius`, gdje `LINE_H` označava broj radnih elemenata radne grupe. Dakle svaki radni element učitava iz globalne memorije element ulazne slike koji se nalazi `radius` redaka iznad središnjeg elementa i element koji se nalazi `radius` redaka ispod središnjeg elementa, tj. `matrix[(iRow - radius)*nCols + iCol]` i `matrix[(iRow + radius)*nCols + iCol]`. Pri čemu `nRows` predstavlja širinu ulazne slike, `nCols` visinu ulazne slike, `iRow = idx % nRows` je redak u kojem se nalazi središnji element, a `iCol = idx % nCols` je stupac u kojem se nalazi središnji element (također i izlazni element). Nakon što radni elementi pohrane elemente segmenta ulazne slike u lokalnu memoriju, slijedi izračun izlaznih elemenata.

Izvorni kôd ovog rješenja nalazi se u datoteci Convolution_v3.cl. Koristi se jednodimenzionalni radni prostor. Veličina globalnog radnog prostora jednaka je veličini ulazne slike (broj elemenata ulazne slike), a veličina lokalnog radnog prostora (veličina radne grupe, broj radni elemenata radne grupe) iznosi 64.

**Program 4.2: Convolution_v3**

```
1  __kernel void ConvolutionRow(__global float* matrix,
2                 const uint nRows,
3                 const uint nCols,
4                 __constant float* horizMask,
5                 const int radius,
6                 __global float* result,
7                 __local float* data)
8  {
9
10   const uint LINE_W = get_local_size(0);
11
12   uint idx = get_global_id(0);
13   uint lid = get_local_id(0);
14
15   float value;
16   int startRowIdx = (idx / nCols) * nCols;
17
18   // ucitaj lijevi piksel: idx - radius
19   int idxPixel = idx - radius;
```

```
20
21    if(idxPixel < startRowIdx) value = 0;
22    else value = matrix[idxPixel];
23
24    data[lid] = value;
25
26    // ucitaj desni piksel: idx + radius
27    idxPixel = idx + radius;
28    if(idxPixel > startRowIdx + nRows - 1) value = 0;
29    else value = matrix[idxPixel];
30
31    data[lid + 2*radius] = value;
32
33    // sinkronizacija svih radnih elemenata radne grupe
34    // svi podaci ucitani u lokalnu memoriju
35    barrier(CLK_LOCAL_MEM_FENCE);
36
37    float sum = 0;
38    int startPixel = (int)lid + radius;
39
40    // horizontalno
41    for(int i = - radius; i <= radius; i++)
42    {
43      sum += data[startPixel + i] * horizMask[radius - i];
44    }
45
46    result[idx] = sum;
47 }
48
49 __kernel void ConvolutionColumn(__global float* matrix,
50                 const int nRows,
51                 const int nCols,
52                 __constant float* vertMask,
53                 const int radius,
54                 __global float* result,
55                 __local float* data)
56 {
57    const uint LINE_H = get_local_size(0);
58
59    uint idx = get_global_id(0);
60    uint lid = get_local_id(0);
61
62    float value;
63
64    int iRow = idx % nRows;
65    int iCol = idx / nRows;
66
67    // ucitaj gornji piksel: idx - radius
68    int idxPixel;
69    idxPixel = iRow - radius;
```

```
70    if(idxPixel < 0) value = 0;
71    else value = matrix[idxPixel * nCols + iCol];
72
73    data[lid] = value;
74
75    // ucitaj donji piksel: idx + radius
76    idxPixel = iRow + radius;
77    if(idxPixel > nRows - 1) value = 0;
78    else value = matrix[idxPixel * nCols + iCol];
79
80    data[lid + 2*radius] = value;
81
82    // sinkronizacija svih radnih elemenata radne grupe
83    // svi podaci ucitani u lokalnu memoriju
84    barrier(CLK_LOCAL_MEM_FENCE);
85
86    float sum = 0;
87    int startPixel = (int)lid + radius;
88
89    // vertikalno
90    for(int i = - radius; i <= radius; i++)
91    {
92      sum += data[startPixel + i] * vertMask[radius - i];
93    }
94
95    result[iRow * nCols + iCol] = sum;
96 }
```

## 4.3 Algoritam Convolution_v411 (Convolution_v412)

Algoritam Convolution_v3 koristi jednodimenzionalni radni prostor, zbog čega je potrebno u jezgrenoj funkciji za konvoluciju po stupcima računati, složenim izrazima, indekse elemenata u ulaznom polju kako bi se dohvatili elementi ulazne slike. Dvodimenzionalni radni prostor lišava jezgrenu funkciju od složenog izračunavanja indeksa, čime se uklanja nekoliko operacija množenja, dijeljenja i modulo operacija. U jezgrenoj funkciji `ConvolutionColumn` algoritma Convolution_v3 pristupi globalnoj memoriji nisu sjedinjeni, dok su u jezgrenoj funkciji `ConvolutionRow` pristupi sjedinjeni, ali nisu poravnati. Shodno tome razvijeno je poboljšano rješenje Convolution_v411 koje uz lokalnu memoriju koristi sjedinjenje i poravnate memorijske pristupe globalnoj memoriji, te dvodimenzionalni radni prostor.

## Jezgrena funkcija konvolucije po redcima

Pristupi globalnoj memoriji su sjedinjeni, naime radni elementi pristupaju slijednim adresama, međutim pristupi globalnoj memoriji će biti optimalno poravnati samo ako je polumjer retčanog separabilnog filtra jednak višekratniku broja 16.

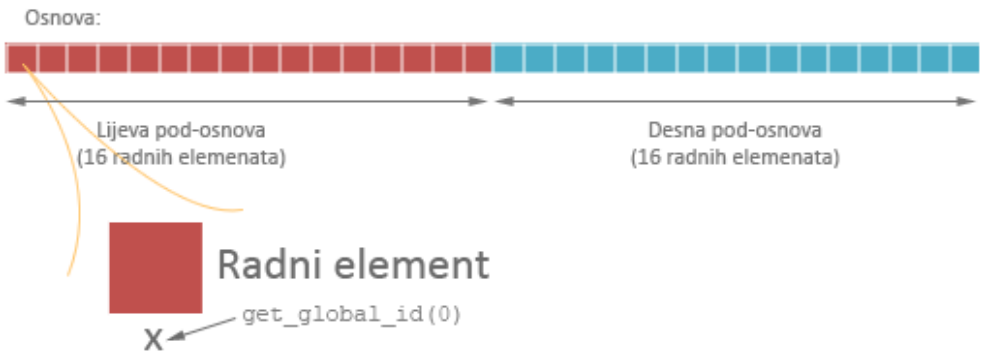
Svaki radni element u algoritmu Convolution_v3 odgovoran je za izračun konvolucije jednog središnjeg elementa ulazne slike, međutim da bi se izračunala konvolucija središnjeg elementa, potrebno je pročitati `radius` slijednih elemenata koji se nalaze lijevo od središnjeg elementa i `radius` slijednih elemenata koji se nalaze desno od središnjeg elementa. Stoga radni elementi osnove čitaju elemente ulazne slike kao što je prikazano na sl. 4.2 te ih pohranjuju u lokalnu memoriju, nakon čega svi radni elementi osnove iz lokalne memorije mogu dohvatiti potrebne elemente za izračun konvolucije svog središnjeg elementa.

Već je spomenuto kako se prilikom prijenosa polja iz radne memorije računala domaćina u globalnu memoriju GPU-a prvi element polja smješta na adresu koja je višekratnik broja 128, a ostali slijedno iza prvog elementa. Dakle adresa `matrix` sigurno je višekratnik broja 128, a prvi radni element prve osnove (onaj za kojeg vrijedi `get_global_id(0) == 0`) izračunat će konvoluciju središnjeg elementa koji se upravo nalazi na adresi `matrix` u globalnoj memoriji.

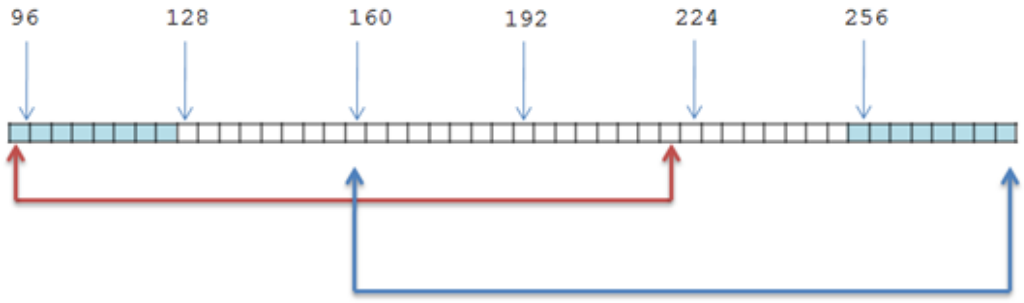
Kako osnova sadrži 32 radna elementa, a svaki radni element odgovoran je za izračun konvolucije jednog središnjeg elementa, gdje je pak element ulazne slike veličine 4 okteta (`float`), tada se središnji element prvog radnog elementa osnove `d`, gdje je `d` indeks osnove, nalazi na adresi `matrix + 32*4*d`. Dakle središnji element prvog radnog elementa bilo koje osnove `d` nalazi se uvijek na adresi koja je višekratnik broja 128. Kada bi radni elementi čitali iz globalne memorije samo vrijednosti središnjih elemenata, tj. `matrix[idx]`, tada bi pristupi bili optimalno poravnati, međutim svaki radni element iz globalne memorije dohvaća element `matrix[idx - radius]` i element `matrix[idx + radius]` (vidi odjeljak 4.2). Samim time iskače se iz optimalnog poravnanja, osim ako polumjer retčanog separabilnog filtra `radius` nije jednak višekratniku broja 16.

Problem neporavnatog pristupa globalnoj memoriji jezgrene funkcije `ConvolutionRow` algoritma Convolution_v3 ilustriran je slikama 4.8, 4.9, 4.10 i 4.11. Slike simbolički prikazuju dohvaćanje iz globalne memorije, pretpostavka je da se rješenje pokreće na grafičkoj kartici NVIDIA GT240. Slike 4.4, 4.5, 4.6 i 4.7 pojašnjavaju prikazane de-

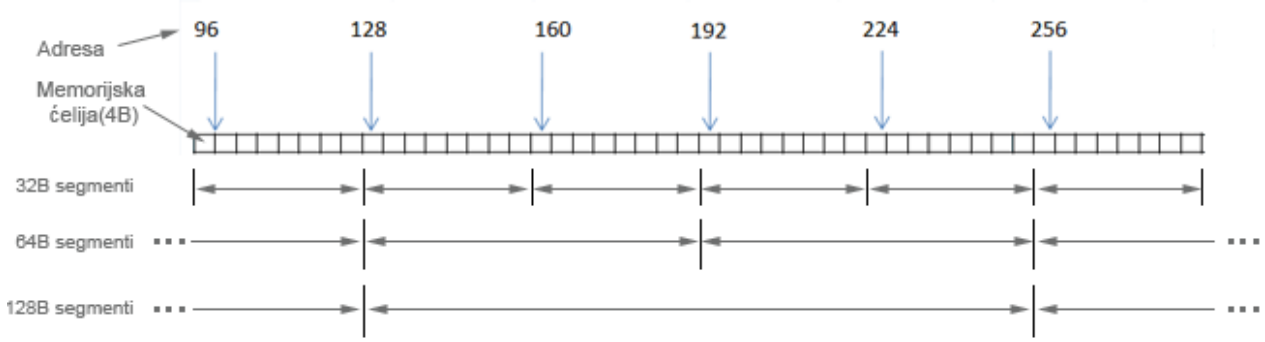
talje na slikama 4.8, 4.9 i 4.10. Prvi element ulazne slike pohranjen je na adresi 0, tj. `matrix = 0`. Prikazano je dohvaćenje radnih elementa s indeksima 32 - 63, radni elementi će dohvatiti blok ulazne slike od adrese 96 do adrese 284. Pretpostavka je da će radni elementi s indeksima 0 - 31, pročitati blok slike od adrese 0 do adrese 124. Na slikama 4.8, 4.9 i 4.10 plavom bojom označeni su tzv. elementi „omotači", polumjer retčanog separabilnog filtra iznosi 8, bijelom bojom su označeni središnji elementi za koje će radni elementi osnove izračunati konvoluciju. Elementi pohranjeni u globalnoj memoriji su tipa `float`, tj. veličine 4 okteta. Osnova sadrži 32 radna elementa, a svaka pod-osnova sadrži 16 radnih elemenata. Ispod tamno-crvenih i plavih kvadrata koji simboliziraju radne elemente, nalaze se indeksi radnih elemenata. Izraz „lijevo čitanje" simbolizira čitanje elementa `matrix[idx - radius]`, dok „desno čitanje" simbolizira čitanje elementa `matrix[idx + radius]` (vidi odjeljak 4.2).



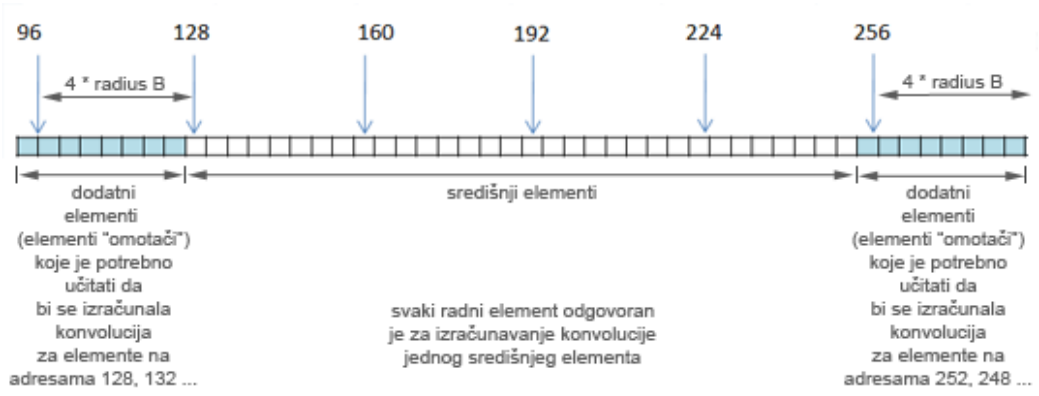

**Slika 4.4:** *Simoblizirani prikaz osnove.*



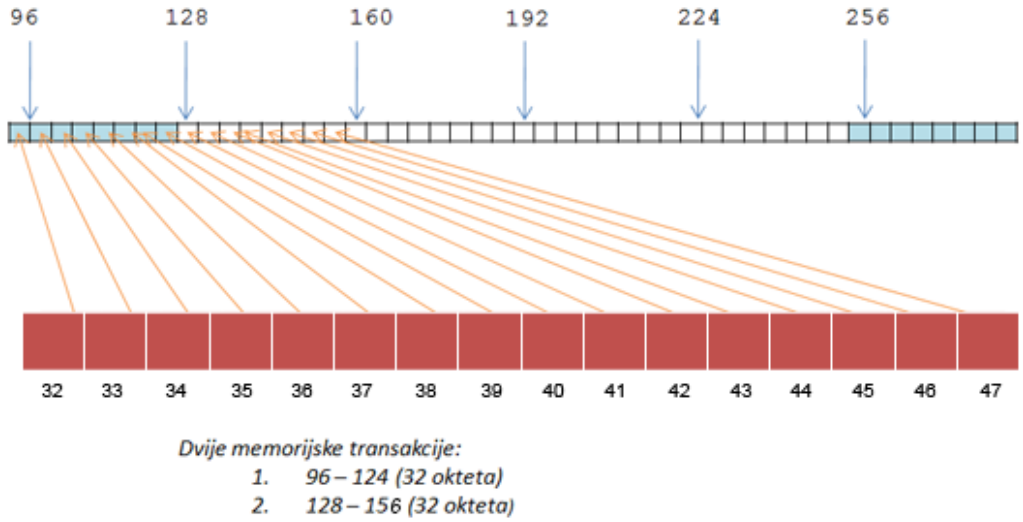

**Slika 4.5:** *Tamno-crvenom linijom su obuhvaćeni elementi koje će pročitati prva pod-osnova, a plavom elementi koje će pročitati druga pod-osnova.*



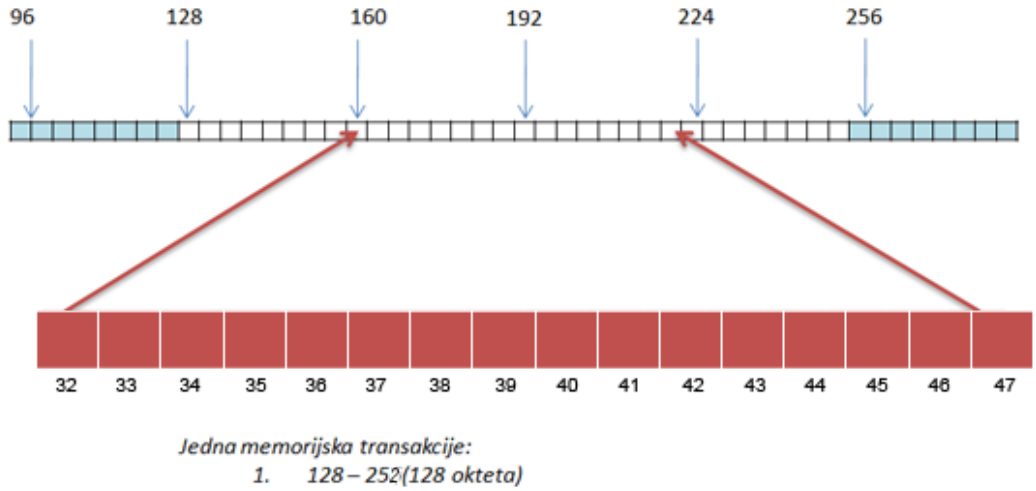

**Slika 4.6:** *Opis strukture globalne memorije.*



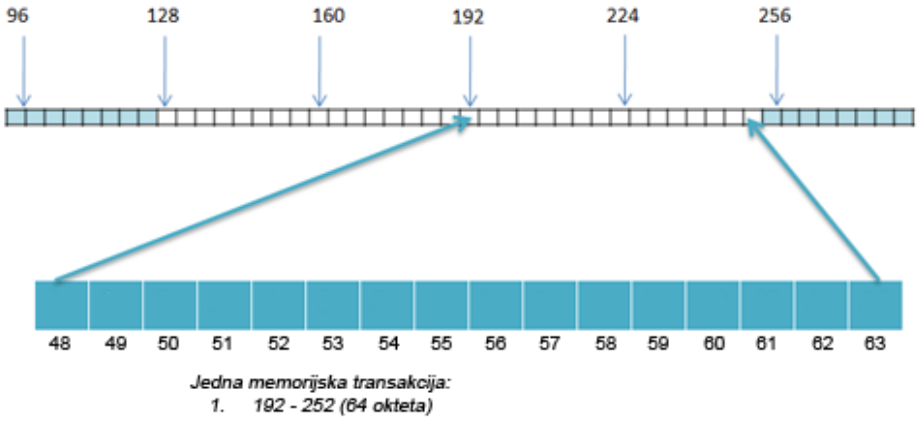

**Slika 4.7:** *Opis elemenata koji se nalaze u globalnoj memoriji.*



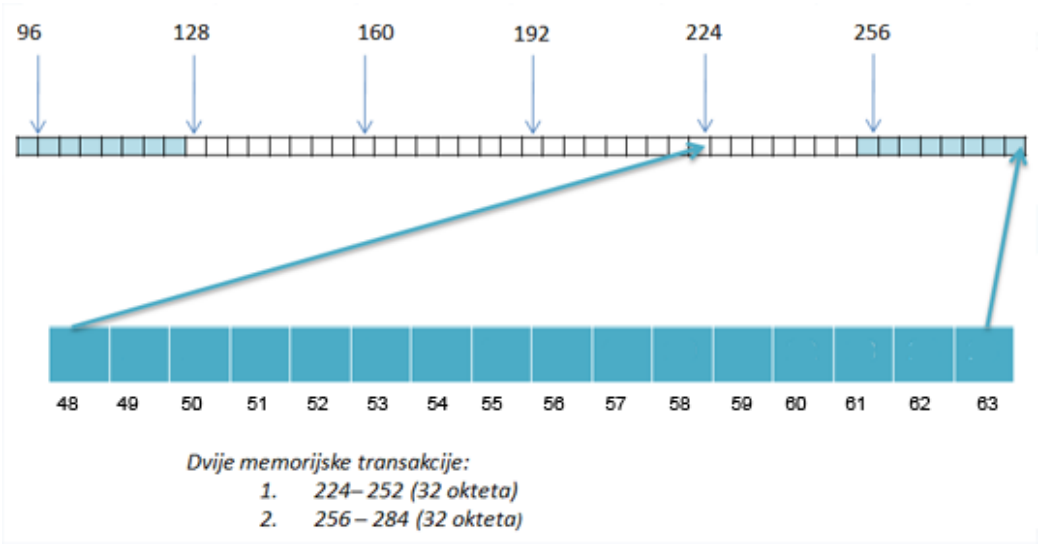

**Slika 4.8:** *Lijevo čitanje prve pod-osnove.*




**Slika 4.9:** *Desno čitanje prve pod-osnove.*




**Slika 4.10:** *Lijevo čitanje druge pod-osnove.*

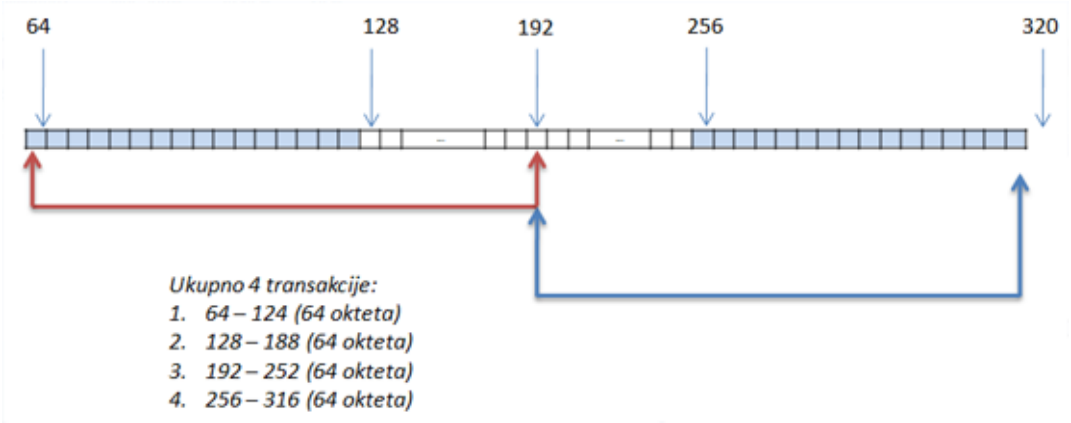



**Slika 4.11:** *Desno čitanje druge pod-osnove.*

Kao što je vidljivo sa slika 4.8, 4.9, 4.10 i 4.11, za čitanje jednog bloka potrebno je 6 memorijskih transakcija, tj. „lijevo čitanje" (sl. 4.8) prve pod-osnove i "desno čitanje" (sl. 4.11) druge pod-osnove serijalizira se u dvije memorijske transakcije, zato što adrese nisu optimalno poravnate.

Međutim ako postavimo polumjer filtra na vrijednost 16, tada su dovoljne 4 memorijske transakcije kako bi se pročitao cijeli blok (vidi sl. 4.12), tj. svakoj pod-osnovi

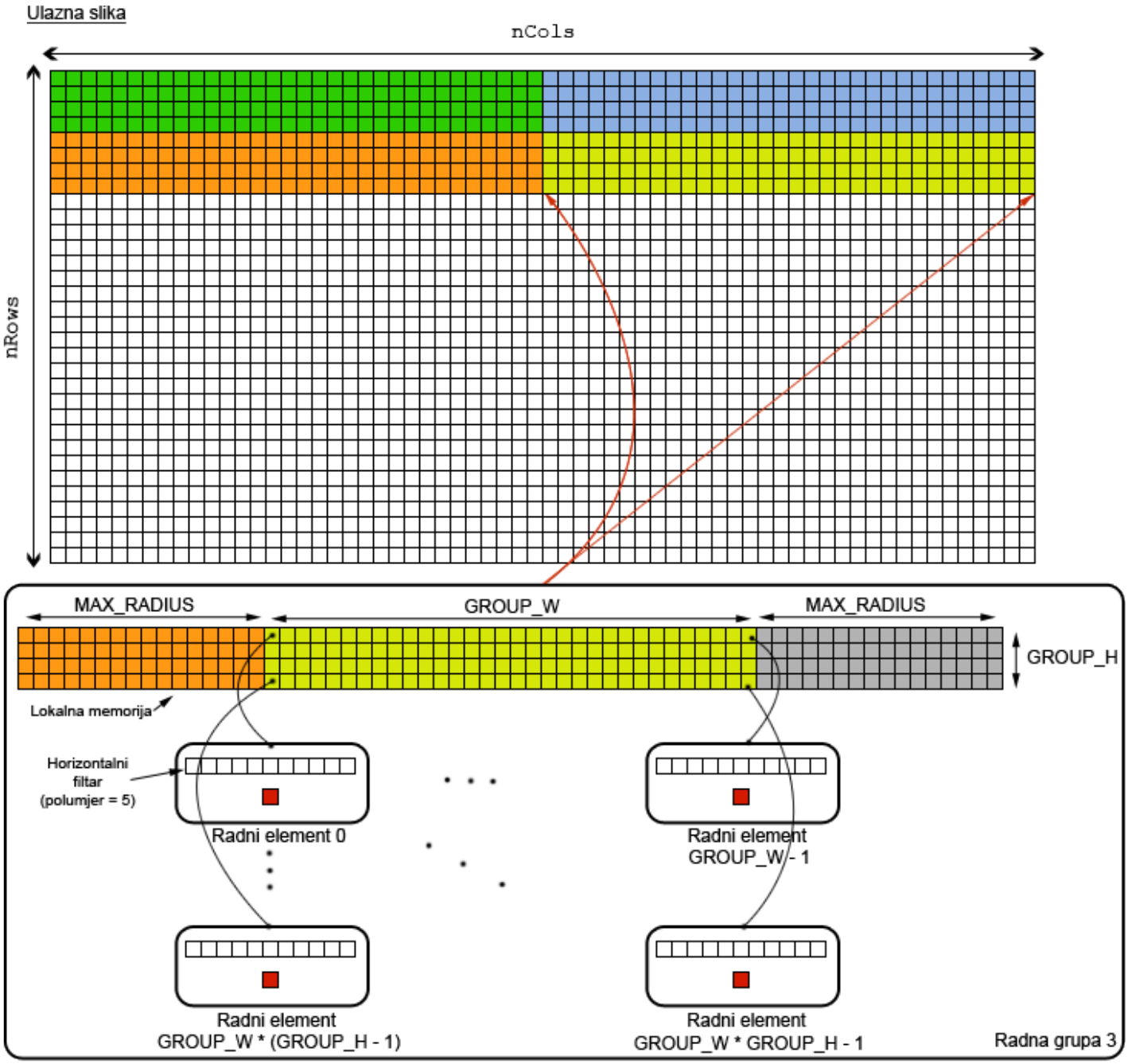
dovoljna je samo jedna memorijska transakcija.




**Slika 4.12:** *Dohvat bloka iz memorije za r = 16.*

Ušteda od dvije memorijske transakcije dovoljan je razlog za uvođenje konstante `MAX_RADIUS = 16`, i mijenjanje `input[idx - radius]` u `matrix[idx - MAX_RADIUS]` i `matrix[idx + radius]` u `input[idx + MAX_RADIUS]`. Za bilo koji polumjer filtra radna grupa će uvijek pročitati iz memorije `MAX_RADIUS` lijevih rubnih elemenata i `MAX_RADIUS` desnih rubnih elemenata, međutim prilikom obrade, tj. izračuna u obzir će ući samo `radius` elemenata „omotača" sa svake strane. Ograničenje ove implementacije je što polumjer filtra mora biti manji ili jednak od konstante `MAX_RADIUS`.

Međutim može se i bez spomenutog ograničenja, ako se unutar aplikacijskog kôda (onaj koji poziva OpenCL) doda funkcionalnost provjeravanja polumjera filtra prije prevođenja OpenCL programa. Ako je polumjer veći od `MAX_RADIUS`, tada se `MAX_RADIUS` poveća za 16 i ponovno se prevede OpenCL program. Spomenuta funkcionalnost nije implementirana za algoritam Convolution_v411, pa se pretpostavlja da polumjeri bilo retčanog bilo stupčanog separabilnog filtra neće biti veći od `MAX_RADIUS`.




**Slika 4.13:** *Shematski prikaz rada jezgrene funkcije ConvolutionRow algoritma Convolution_v411.*

Sl. 4.13 opisuje rad jezgrene funkcije `ConvolutionRow` algoritma algoritma Convolution_v411. Prikazan je jedan trenutak izvođenja radne grupe nazvane „Radna grupa

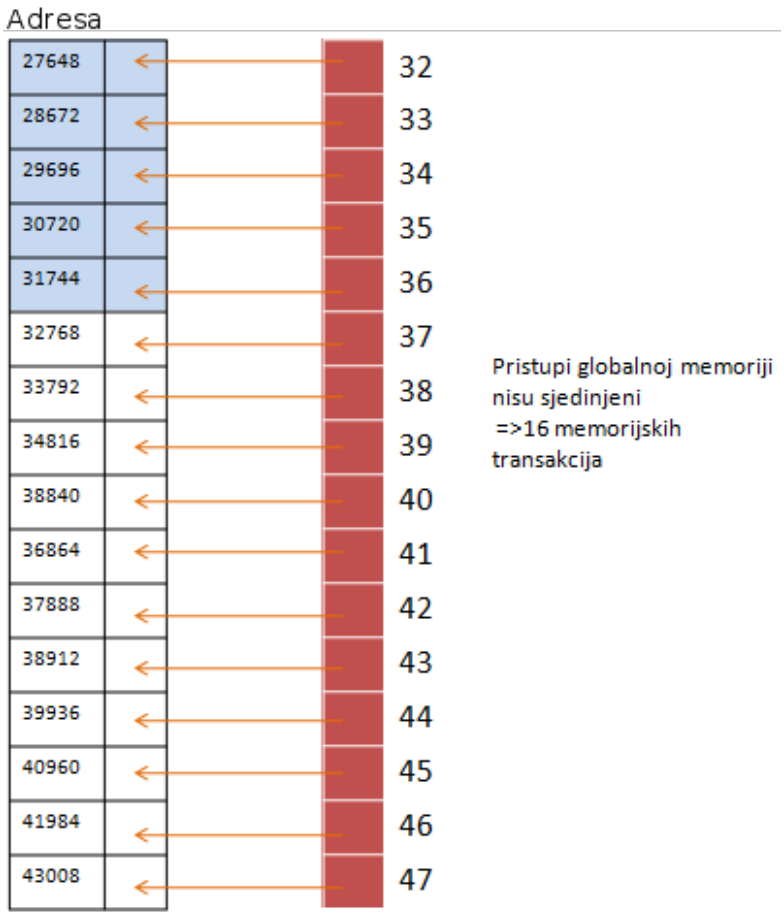
3". Prikazan je figurativan smještaj ulazne slike u globalnoj memoriji, različiti segmenti slike su različito obojani, prikazano je koji radni element je odgovoran za koji središnji element iz lokalne memorije.

Jezgrena funkcija za konovoluciju po retcima ovog algoritma koristi dvodimenzionalan radni prostor, pri čemu je globalni radni prostor jednak dimenzijama ulazne slike, a lokalni radni prostor iznosi (32, 4).

## Jezgrena funkcija konvolucije po stupcima

Za razliku od konvolucije po retcima, ovdje pristupi memoriji nisu sjedinjeni i to je glavni razlog usporenja konvolucije po stupcima u odnosu na konvoluciju po retcima. Valja još jedanput razmotriti jezgrenu funkciju `ConvolutionColumn` algoritma Convolution_v3, tj. koje to elemente radni element dohvaća: `matrix[(iRow - radius) * nCols + iCol ]` i `matrix[(iRow + radius) * nCols + iCol ]`. Uzrok usporenja je u tome što radni elementi iz iste pod-osnove prilikom pristupa globalnoj memoriji ne pristupaju slijednim adresama, odnosno ne pristupaju istim segmentima[3], već svaki radni element pristupa različitom segmentu, što rezultira serijalizacijom i `b/2` memorijskih transakcija, gdje `b/2` označava broj radnih članova pod-osnove. Ako je `b = 32`, to je ukupno 16 memorijskih transakcije za samo jednu pod-osnovu. Slika 4.14 ilustrira problematično dohvaćanje iz globalne memorije (dimenzije ulazne slike 256 $\times$ 256).

[3]Odnosi se na segmente globalne memorije, a ne na segmente slike (blokove).

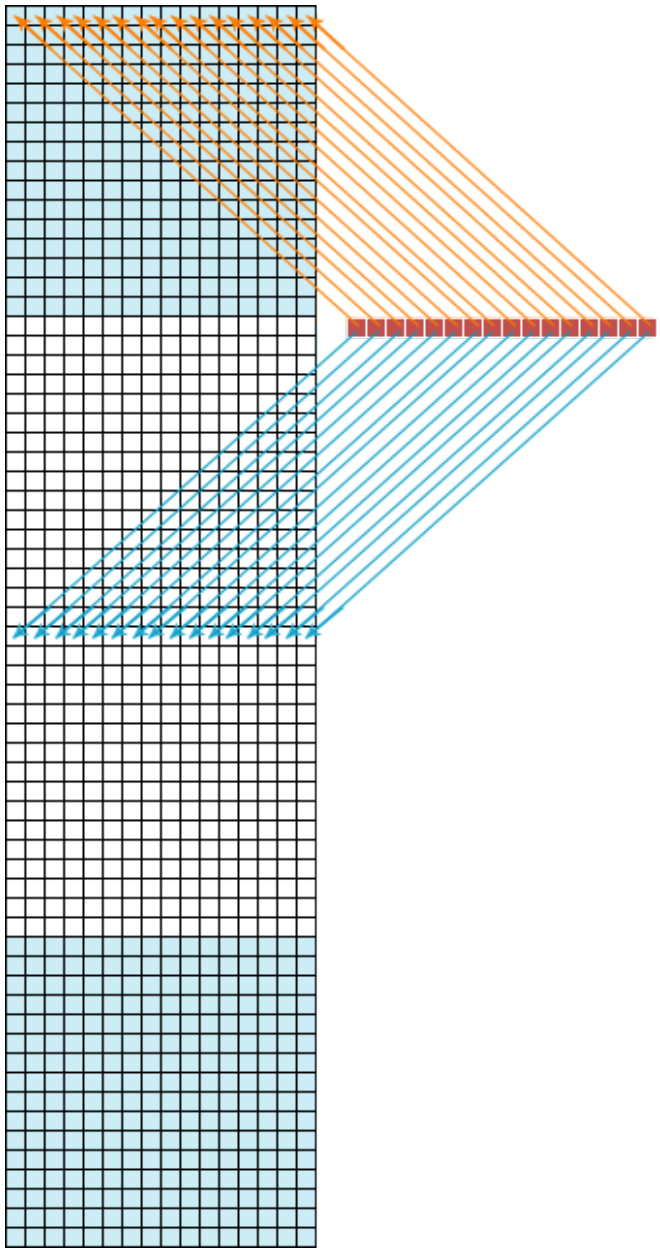



**Slika 4.14:** *„Gornje" čitanje lijeve pod-osnove*

Dakle valjalo bi jezgrenu funkciju organizirati tako da radni elementi iz iste pod-osnove pristupaju slijednim memorijskim adresama, čime bi broj memorijskih transakcija prilikom pristupa, s 16 transakcija pao na samo jednu memorijsku transakciju.

Temeljna ideja rješenja se koncipira na tome da se lokalni radni prostor definira kao dvodimenzionalan s dimenzijama `16` $\times$ `32`. Dakle u lokalnu memoriju će biti učitan blok slike veličine `16 * (MAX_RADIUS + 32 + MAX_RADIUS)`. Naime za svaki stupac potrebno je pročitati `radius` elemenata „omotača" s gornje strane i `radius` elemenata „omotača" s donje strane. Središnji element na kojem se provodi konvolucija, a time i izlazni, definiran je izrazom `globalRowShift = globalY * nRows + globalX`, gdje je `globalX = get_global_id(0)` x-koordinata radnog elementa, a `globalY = get_global_id(1)` y-koordinata radnog elementa. Sada će radni elementi unutar iste

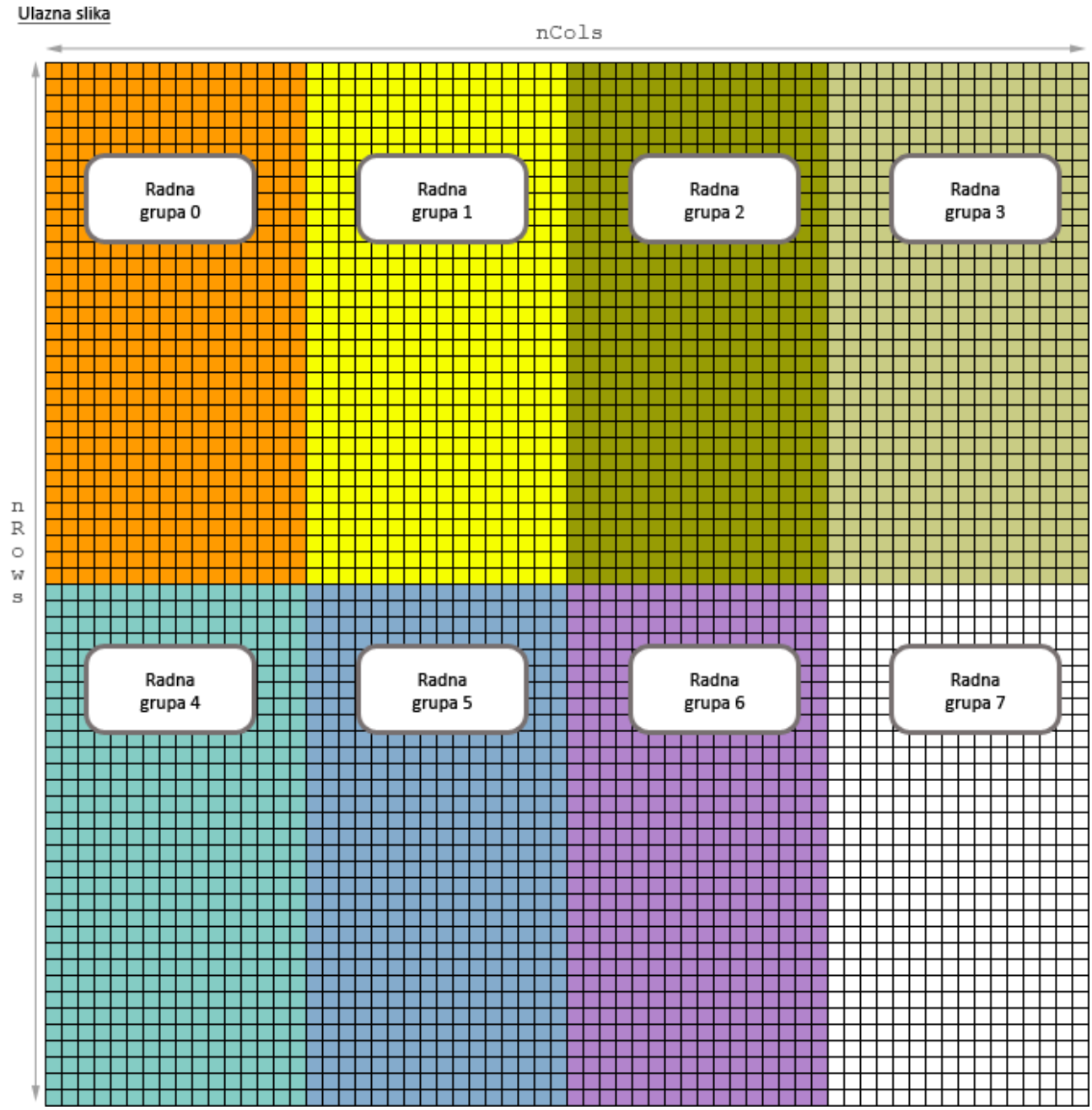
pod-osnove pristupati istom segmentu, jer je „gornje čitanje" `matrix[globalRowShift - MAX_RADIUS * nRows]` pa će radni članovi unutar pod-osnove dohvaćati slijedne memorijske adrese, naime u izrazu `matrix[globalRowShift - MAX_RADIUS * nRows]` mijenja se samo `globalRowShift`, tj. `globalX` koji označava x-koordinatu radnog elementa, a slijednim radni elementima osnove mijenja se samo vrijednost x-koordinate i to slijedno. „Donje čitanje" definirano je izrazom `matrix[globalRowShift + MAX_RADIUS * nRows]`. Čitanje iz globalne memorije za jednu pod-osnovu ilustrirano je slikom 4.15. Na slici narančaste strelice simboliziraju „gornje" čitanje, dok plave strelice simboliziraju „donje" čitanje. „Gornje" čitanje rezultira jednom memorijskom transakcijom i „donje" čitanje rezultira jednom transakcijom.



**Slika 4.15:** *„Gornje" i „donje" čitanje iz globalne memorije za jednu pod-osnovu*

Dakle pod-osnova će pročitati jedan redak i pohraniti ga u lokalnu memoriju. Nakon što se cijeli blok pročita, provodi se konvolucija nad središnjim elementima.

Sl. 4.16 prikazuje raspored radnih grupa za ulaznu sliku dimenzija $64 \times 64$. Veličina svake radne grupe je $16 \times 32$, pri čemu je svaka radna grupa odgovorna za izračun konvolucije u svom dijelu slike.




**Slika 4.16:** *Svaka radna grupa odgovorna je za svoj blok ulazne slike.*

## Dodatne optimizacije

Primijećeno je razmjerno značajno ubrzanje izvođenja jezgrenih funkcija, kada je varijabla `radius` koja se prenosi kao argument jezgrenih funkcija i koja označava polumjer filtra, zamijenjena konstantama `HORIZ_KERNEL_RADIUS` za polumjer retčanog filtra i `VERT_KERNEL_RADIUS` za polumjer stupčanog filtra. Naime prevoditelj prilikom prevođenja jezgrenih funkcija petlju koja je odgovorna za izračun konvolucije nad središnjim elementom odmota. Izvorni kôd na (vidi program 4.3) izmijenjen je u (vidi program 4.4).

**Program 4.3: Petlja jezg. fje. ConvolutionRow.**

```
1
2   for(int i = - radius; i <= radius; i++)
3   {
4     sum += data[startPixel + i] * horizMask[radius - i];
5   }
```

**Program 4.4: Petlja jezg. fje. ConvolutionRow s konstantnim polumjerom.**

```
1 #define HORIZ_KERNEL_RADIUS 8
2
3   for(int i = - HORIZ_KERNEL_RADIUS; i <= HORIZ_KERNEL_RADIUS; i++)
4   {
5     sum += data[startPixel + i] * horizMask[HORIZ_KERNEL_RADIUS - i
          ];
6   }
```

Problem ovakve modifikacije je što je potrebno svaki put ponovno prevesti OpenCL program s jezgrenim funkcijama prilikom promjene polumjera filtra, što je izuzetno vremenski zahtjevno. Međutim pretpostavlja se da korisnici, u većini slučajeva, ipak konvoluciju provode nad većim brojem slika s istim polumjerom filtra.

## Dodatne napomene

Implementacija ovog algoritma nalazi se u datoteci Convolution_v411.cl.

Veličina radne grupe može iznositi najviše 128 na grafičkoj kartici ATI Radeon HD 5670, kako algoritam Convolution_v411, tj. jezgrena funkcija `ConvolutionColumn` zahtjeva 512 radnih elemenata za jednu radnu grupu, potrebno je bilo modificirati jezgrenu funkciju za konvoluciju po stupcima. Naime veličina radne grupe smanjena

je na $8 \times 16$, što ukupno iznosi 128 radnih elemenata, shodno tome bilo je potrebno uvesti još jedno dodatno čitanje iz globalne memorije, jer dva čitanja nisu dovoljna za učitavanje cijelog bloka slike, pa svaki radni element još i dohvaća vrijednost središnjeg elementa iz globalne memorije.

Modifikacija ovog algoritma za grafičku karticu ATI Radeon HD 5670 nalazi se u datoteci Convolution_v412.cl

## Izvorni kôd jezgrenih funkcija

**Program 4.5: Jezgrena fja. konvolucije po retcima.**

```
1  #define HORIZ_KERNEL_RADIUS 8
2  #define VERT_KERNEL_RADIUS 8
3
4  #define MAX_RADIUS 16
5
6  __kernel void ConvolutionRow(__global float* matrix,
7                 const uint nRows,
8                 const uint nCols,
9                 __constant float* horizMask,
10                __global float* result,
11                __local float* data)
12 {
13   const uint GROUP_W = get_local_size(0);
14   const uint GROUP_H = get_local_size(1);
15
16   const uint globalX = get_global_id(0);
17   const uint globalY = get_global_id(1);
18
19   const uint localX = get_local_id(0);
20   const uint localY = get_local_id(1);
21
22   const uint BLOCK_WIDTH = ( MAX_RADIUS + GROUP_W + MAX_RADIUS );
23
24   float value;
25
26   const int localRowShift = localY * BLOCK_WIDTH + localX +
       MAX_RADIUS;
27   const int globalRowShift = globalY * nRows + globalX;
28
29   int leftPixelGlobalX = globalX - MAX_RADIUS;
30
31   if(leftPixelGlobalX < 0) value = 0;
32   else value = matrix[globalRowShift - MAX_RADIUS];
33
34   data[localRowShift - MAX_RADIUS] = value;
```

```
35
36   int rightPixelGlobalX = globalX + MAX_RADIUS;
37
38   if(rightPixelGlobalX > nRows - 1) value = 0;
39   else value = matrix[globalRowShift + MAX_RADIUS];
40
41   data[ localRowShift + MAX_RADIUS ] = value;
42
43   barrier(CLK_LOCAL_MEM_FENCE);
44
45   float sum = 0;
46   int startPixel = localRowShift;
47
48   for(int i = - HORIZ_KERNEL_RADIUS; i <= HORIZ_KERNEL_RADIUS; i++)
49   {
50     sum += data[startPixel + i] * horizMask[HORIZ_KERNEL_RADIUS - i
         ];
51   }
52
53   result[globalRowShift]  = sum;
54 }
```

**Program 4.6: Jezgrena fja. za konvoluciju po stupcima.**

```
1  __kernel void ConvolutionColumn(__global float* matrix,
2                 const int nRows,
3                 const int nCols,
4                 __constant float* vertMask,
5                 __global float* result,
6                 __local float* data)
7  {
8    const uint GROUP_W = get_local_size(0);
9    const uint GROUP_H = get_local_size(1);
10
11   const uint globalX = get_global_id(0);
12   const uint globalY = get_global_id(1);
13
14   const uint localX = get_local_id(0);
15   const uint localY = get_local_id(1);
16
17   const uint BLOCK_WIDTH = ( GROUP_W );
18
19   float value;
20
21   const int localRowShift = (localY + MAX_RADIUS) * BLOCK_WIDTH +
        localX;
22   const int globalRowShift = globalY * nRows + globalX;
23
24   int topPixelGlobalX = globalY - MAX_RADIUS;
```

```
25
26   if(topPixelGlobalX < 0) value = 0;
27   else value = matrix[globalRowShift - MAX_RADIUS * nRows];
28
29   data[ localRowShift - MAX_RADIUS * BLOCK_WIDTH ] = value;
30
31   int bottomPixelGlobalX = globalY + MAX_RADIUS;
32
33   if(bottomPixelGlobalX > nCols - 1) value = 0;
34   else value = matrix[globalRowShift + MAX_RADIUS * nRows];
35
36   data[ localRowShift + MAX_RADIUS * BLOCK_WIDTH ] = value;
37   )
38   barrier(CLK_LOCAL_MEM_FENCE);
39
40   float sum = 0;
41   int startPixel = localRowShift;
42
43   for(int i = - VERT_KERNEL_RADIUS; i <= VERT_KERNEL_RADIUS; i++)
44   {
45     sum += data[startPixel + i * BLOCK_WIDTH] * vertMask[
         VERT_KERNEL_RADIUS - i];
46   }
47
48   result[globalRowShift]  = sum;
49 }
```

# Poglavlje 5

# Implementacija afine transformacije i homografije

U OpenCL-u implementirana je afina transformacija i homografija. Obje transformacije često se koriste u raznim algoritmima računalnog vida, pa su stoga vremenske performanse tih transformacija od velike važnosti. OpenCL i arhitektura grafičkih kartica pružaju mogućnost ubrzanja afine transformacija i homografije u odnosu na standardne implementacije koje koriste CPU.

Neka su ulazna točka, transformacijska matrica i izlazna točka definirane kao:

$$p_{ul} = \begin{bmatrix} x_{ul} \\ y_{ul} \\ 1 \end{bmatrix}, H = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix}, p_{iz} = \begin{bmatrix} x_{iz} \\ y_{iz} \\ 1 \end{bmatrix}. \tag{5.1}$$

Tada je homografija definirana izrazom:

$$\omega \; p_{iz} = H \; p_{ul}. \tag{5.2}$$

Iz čega slijedi:

$$\omega = h_{31}x_{ul} + h_{32}y_{ul} + h_{33}, \tag{5.3}$$

$$x_{iz} = \frac{h_{11}x_{ul} + h_{12}y_{ul} + h_{13}}{\omega}, \tag{5.4}$$

$$y_{iz} = \frac{h_{21}x_{ul} + h_{22}y_{ul} + h_{23}}{\omega}. \tag{5.5}$$

Za $h_{31} = 0, h_{32} = 0, h_{33} = 1$ izvodi se afina transformacija. Afina transformacija specijalni je slučaj homografije, međutim implementirana je u OpenCL-u kao zasebna

jezgrena funkcija zbog uštede od dviju operacija dijeljenja. Dakle algoritam za obje jezgrene funkcije je sličan, osim što jezgrena funkcija homografije ima dvije dodatne operacije dijeljenja.

Jezgrene funkcije koriste teksturnu priručnu memoriju za ulaznu sliku, jer nije moguće sjediniti pristupe globalnoj memoriji prilikom dohvaćanja slikovnih elemenata iz ulazne slike. Međutim kako radni elementi unutar osnove pristupaju globalnoj memoriji donekle lokalizirano (relativno bliske memorijske adrese) tada valja iskoristiti prednosti teksturne priručne memorije.

Svaki radni element dohvaća četiri ulazne točke i računa četiri izlazne točke. Stoga globalni radni prostor sadrži $output_{width} * output_{height}/4$ radnih elemenata, pri čemu je $output_{width}$ širina izlazne slike, a $output_{height}$ visina izlazne slike. Odnosno za oba dva rješenja, definiran je dvodimenzionalan radni prostor, dimenzija $(output_{width}/4, output_{height})$. Svaki radni element obrađuje četiri izlazne točke, umjesto jedne točke, jer se pokazalo vremenski isplativije.

Nakon izračuna izraza (5.4) i (5.5) obavlja se bilinearna interpolacija te se dobivene vrijednosti pohranjuju u izlaznu sliku.

Implementirana rješenja ugrađena su u cvsh ljusku. Izvorni kôd jezgrene funkcije za afinu transformaciju nalazi se u datoteci AffineTransformation_T.cl, dok se jezgrena funkcija homografije nalazi u datoteci Homography_T.cl

# Poglavlje 6

# Rezultati implementacija konvolucije

U ovom poglavlju prikazani su rezultati testiranja nekoliko implementacija konvolucije u OpenCL-u u odnosu na implementaciju u višem programskom jeziku (C++) te u odnosu na SSE implementaciju.

Izvršena su dva testiranja OpenCL implementacija na dvama računalima, s različitim grafičkim karticama. Jedno računalo sadrži ATI Radeon HD 5670, dok drugo računalo sadrži NVIDIA GT240.

**Tablica 6.1:** *ATI Radeon HD 5670 specifikacija.*

| **ATI Radeon HD 5670** | |
|---:|---|
| Broj protočnih procesnih elemenata | **400** |
| Broj protočnih procesora | **80** |
| Broj računskih jedinica | **5** |
| Veličina lokalne memorije na računskoj jedinici | **32 kB** |
| Veličina globalne memorije | **512 MB GDDR3** |
| Propusnost memorije | **64 GB/s** |
| Takt memorije | **1.0 GHz** |
| GPU takt | **775 MHz** |

**Tablica 6.2:** *NVIDIA GT240 specifikacija.*

| **NVIDIA GT240** | |
|---|---|
| Broj protočnih procesora | **96** |
| Broj računskih jedinica | **12** |
| Veličina lokalne memorije na računskoj jedinici | **16 kB** |
| Veličina globalne memorije | **512 MB GDDR5** |
| Propusnost memorije | **54.4 GB/s** |
| Takt memorije | **1.0 GHz** |
| GPU takt | **550 MHz** |

Obje grafičke su istog cjenovnog ranga i vrlo sličnih performansi, međutim ipak različitih arhitektura.

Testiranje implementacije konvolucije realizirane SSE instrukcijama i implementacije u C++ jeziku provedeno je na trećem računalu (Tabl. 6.3).

**Tablica 6.3:** *Specifikacija "Računala 3".*

| **"Računalo 3"** | |
|---|---|
| CPU: | **Intel Core 2 Duo 1,8 Ghz** |
| Memorija: | **3 GB DDR2** |

Implementacija konvolucije SSE instrukcijama preuzeta je iz završnog rada Stjepana Hadjića [16], dok je u višem programskom jeziku (C++) realizirana vlastita implementacija. Obje implementacije koriste separabilne filtre kao i implementacije u OpenCL-u. Implementacija iz završnog rada Stjepana Hadjića ne koristi višenitnost (višedretvenost), tj. implementacija nije paralelizirana, baš kao i implementacija u višem programskom jeziku (C++).

Prije samog prikaza rezultata testiranja, valja razjasniti pojmove koji se pojavljuju u podacima. Tablica 6.4 prikazuje nazive implementacija s oznakom jezika u kojem su implementirane. Grafovi su prikazani u rezoluciji od 1 ms, a testiranja nad svim uzorcima pokrenuta su nekoliko puta (implementacije u OpenCL-u: 10 puta), nakon čega je ukupno vrijeme izvođenja podijeljeno s brojem pokretanja.

**Tablica 6.4:** *Nazivi realiziranih implementacija.*

| Implementirano u jeziku | Naziv implementacije |
|---|---|
| C++ | **ConvolutionCAlg.hpp** |
| OpenCL | **Convolution_v1.cl** |
| | **Convolution_v3.cl** |
| | **Convolution_v4.cl** |
| | **Convolution_v41.cl** |
| | **Convolution_v411.cl** |
| | **Convolution_v51.cl** |
| SSE | **SSSv3 32 bit** |

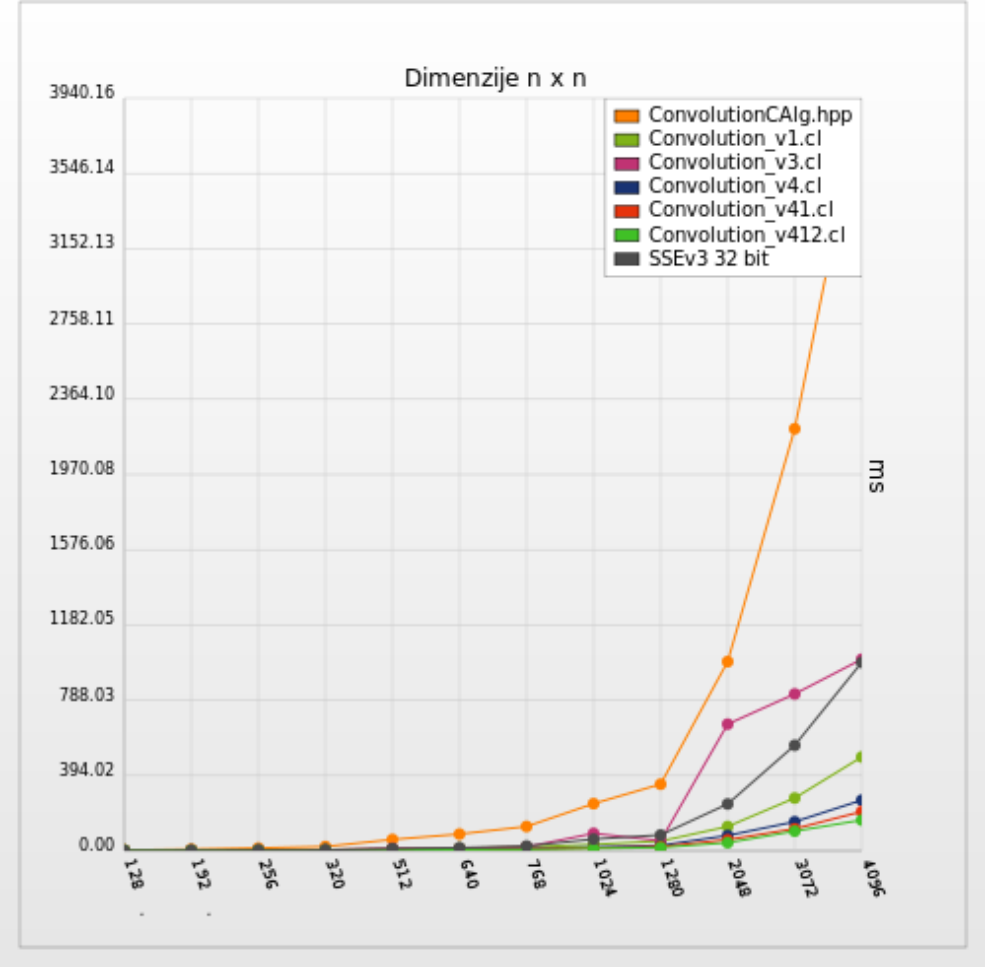
U svim implementacija ulazni i izlazni elementi su 32 bitne vrijednosti s pomičnim zarezom jednostruke preciznosti (**float**), baš kao i elementi filtra. Izuzetak je implementacija **Convolution_v51.cl**, koja je ista kao i **Convolution_v411.cl** implementacija, ali radi s 8 bitnim cjelobrojnim vrijednostima (**char**).

Implementacija **Convolution_v411.cl** zahtjeva 512 radnih članova po radnoj grupi, međutim grafička kartica ATI Radeon HD 5670 dozvoljava maksimalno 128 radnih članova po jednoj radnoj grupi, pa je prilikom testiranja na ATI Radeon HD 5670 grafičkoj kartici broj radnih članova po grupi smanjen na 128. Kako uvjeti testiranja nisu potpuno jednaki, a i sama implementacija je malo izmijenjena (jedno dodatno čitanje), implementacija za ATI Radeon HD 5670 preimenovana je u **Convolution_v412.cl**. Dakle razlika između **Convolution_v411.cl** i **Convolution_v412.cl** je jedino u veličini radnih grupa, dok je sama implementacija jezgrenih funkcija gotova pa ista.

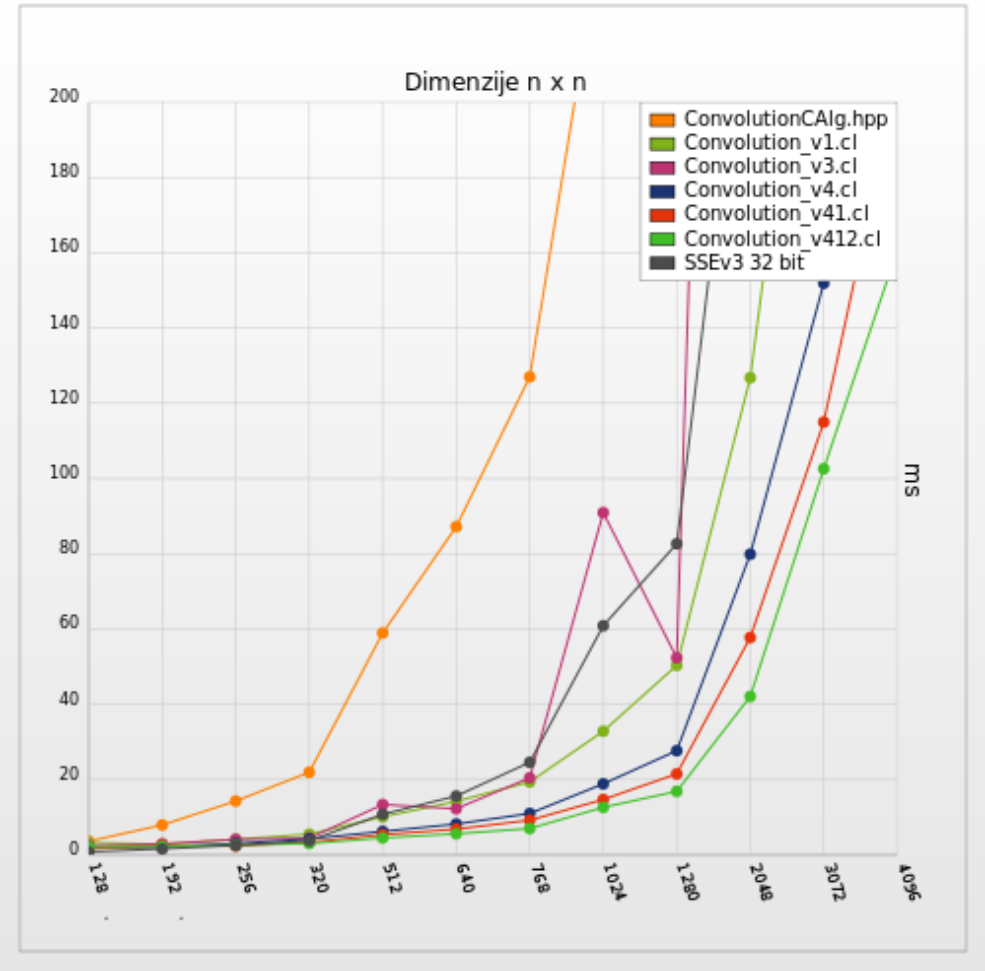
Sva vremena OpenCL implementacija odnose se na ukupno vrijeme izvođenja, tj. od trenutka inicijaliziranja spremnika do trenutka nakon što su dohvaćeni rezultati. Dakle kao vrijeme izvođenja OpenCL programa računa se: inicijalizacija spremnika s podacima koji će biti prenijeti na GPU, prijenos podataka na GPU, pokretanje i izvođenje jezgrenih funkcija na GPU, dohvat rezultata iz globalne memorije GPU-a u radnu memoriju računala.

## 6.1 Test 1 - po dimenzijama ulazne slike

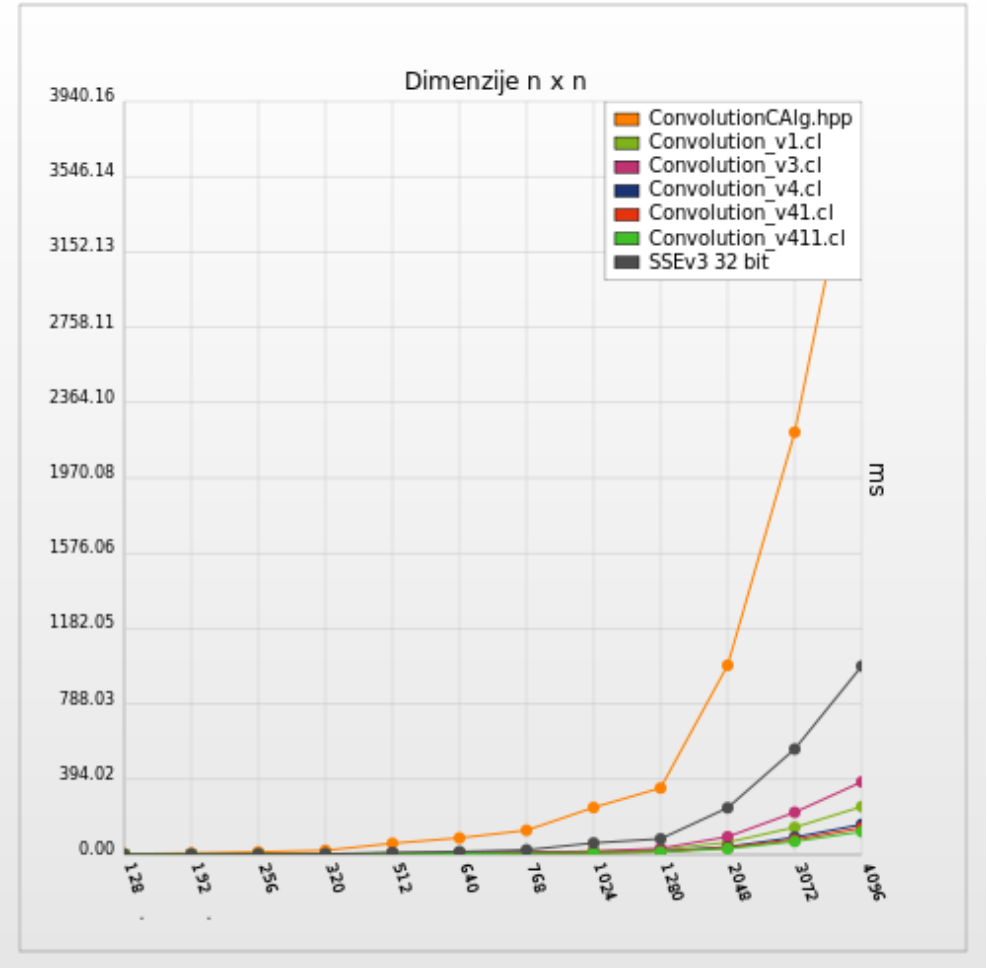
Testiranje je provedeno nad različitim veličinama ulazne slike. Ulazna slika je kvadratnih dimenzija `n` × `n`, dok je veličina filtra konstanta i iznosi `17` × `17`. Os apscisa predstavlja `n` (tj. širinu i visinu ulazne slike), dok ordinata predstavlja vrijeme izvođenja u milisekundama.



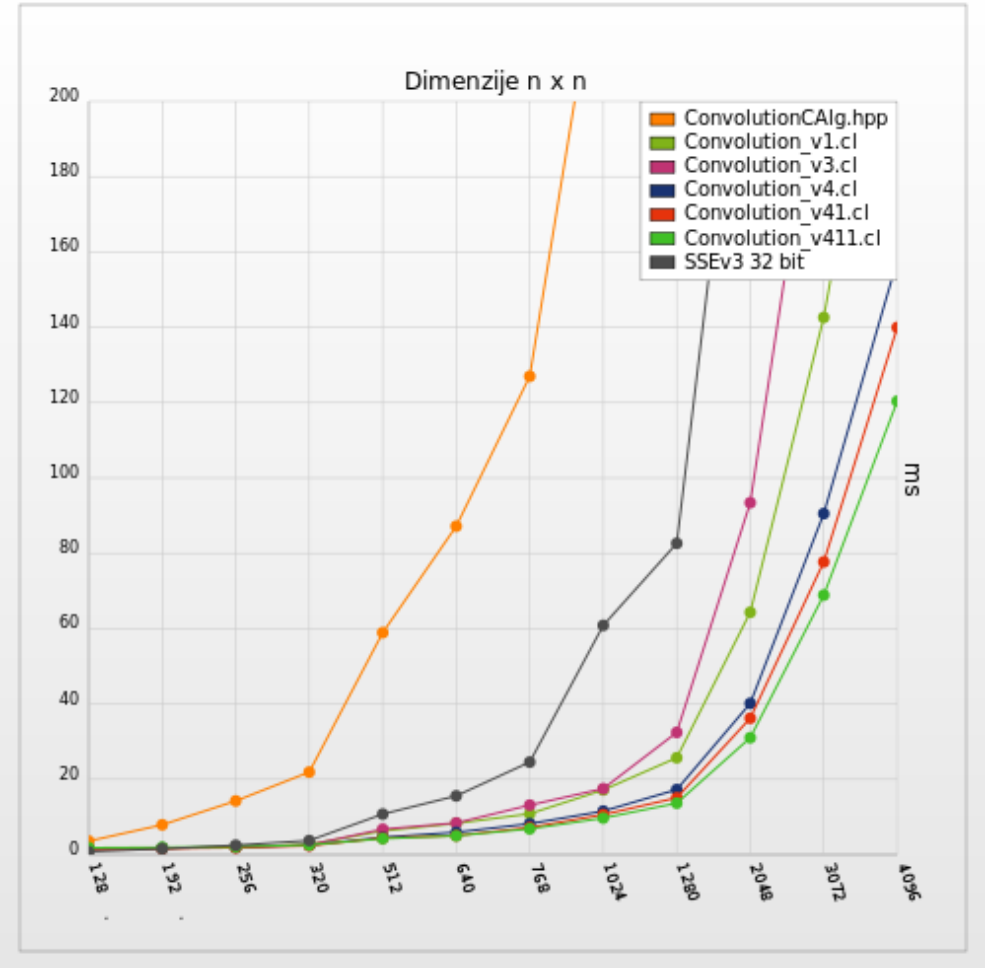

**Slika 6.1:** *Rezultati svih implementacija (ATI Radeon HD 5670).*



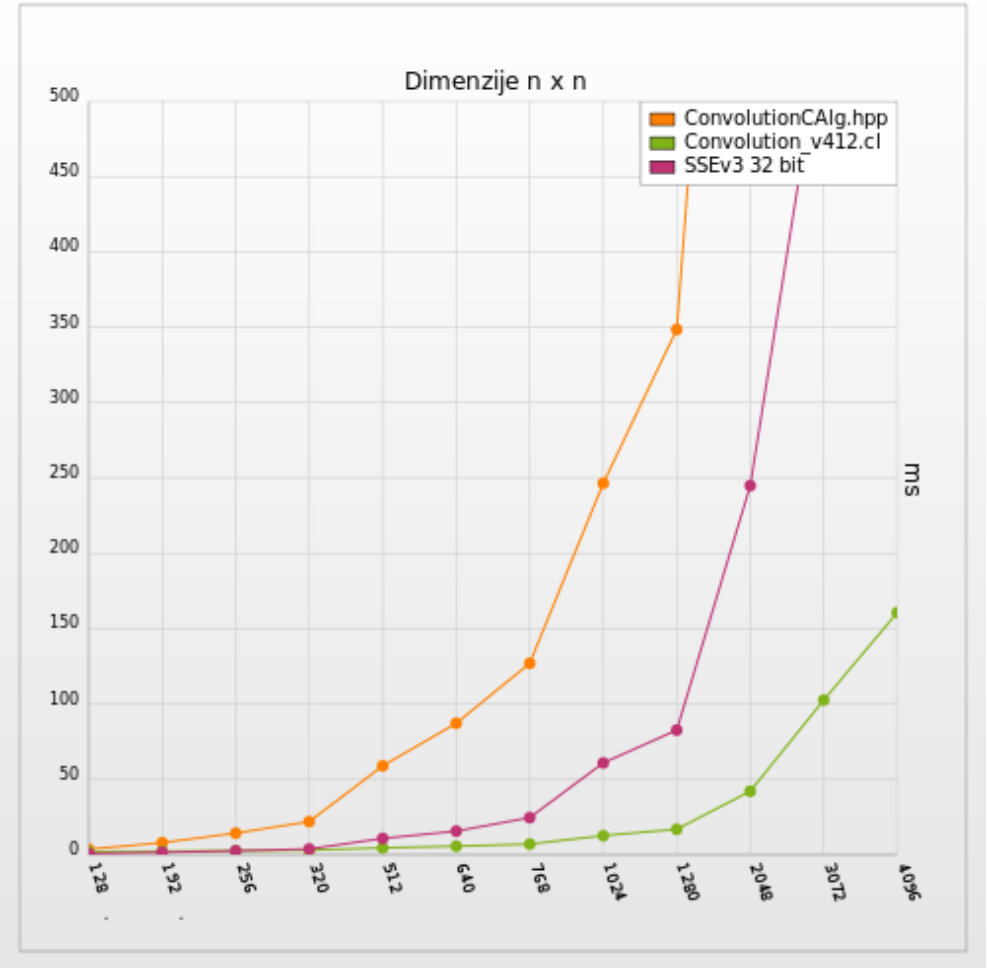

**Slika 6.2:** *Rezultati svih implementacija (ATI Radeon HD 5670) - uvećani prikaz.*



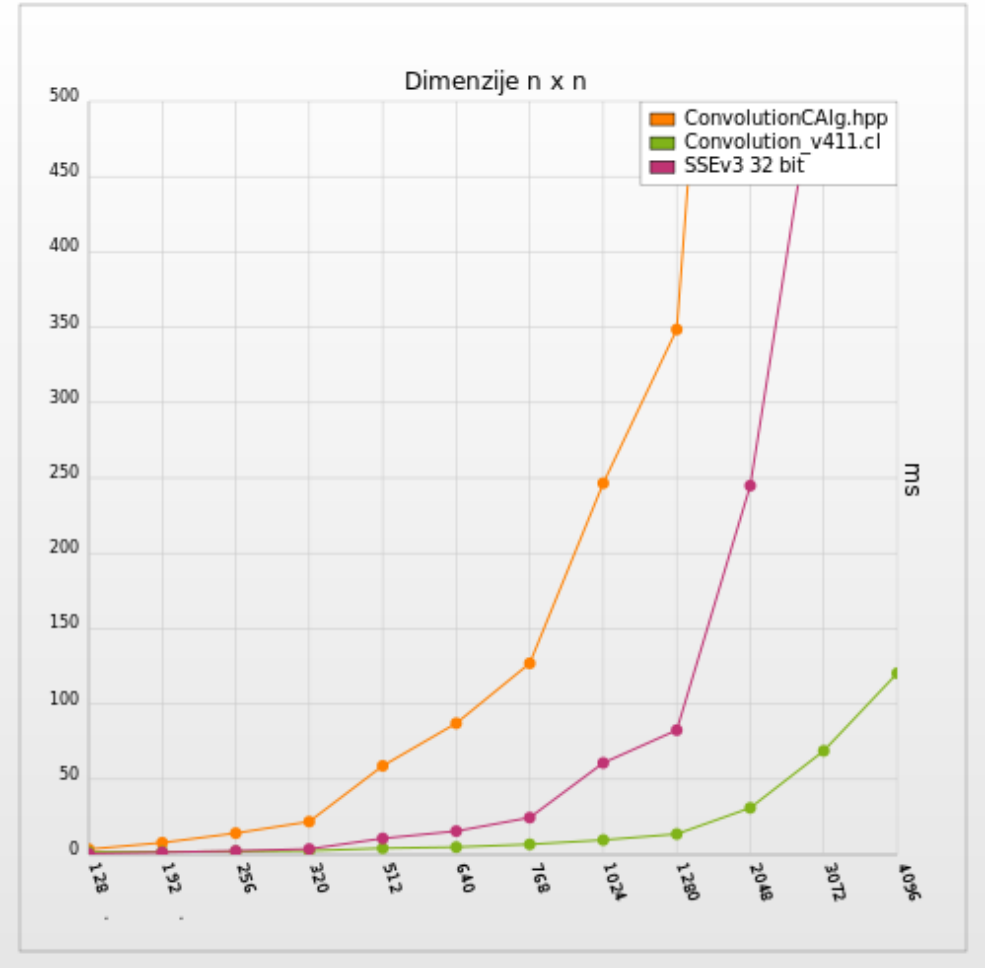

**Slika 6.3:** *Rezultati svih implementacija (NVIDIA GT240).*



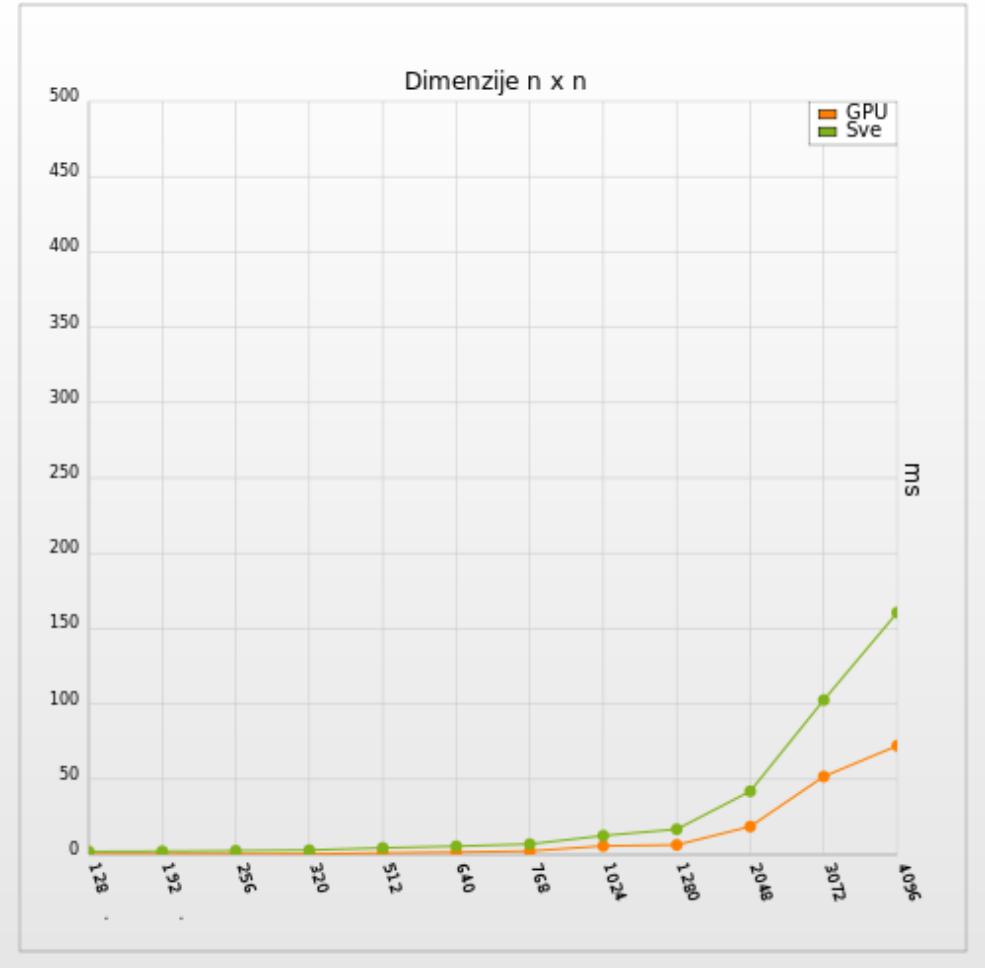

**Slika 6.4:** *Rezultati svih implementacija (NVIDIA GT240) - uvećani prikaz.*




**Slika 6.5:** *Najbrža OpenCL implementacija vs. C++ implementacija vs. SSE implementacija (ATI Radeon HD 5670).*

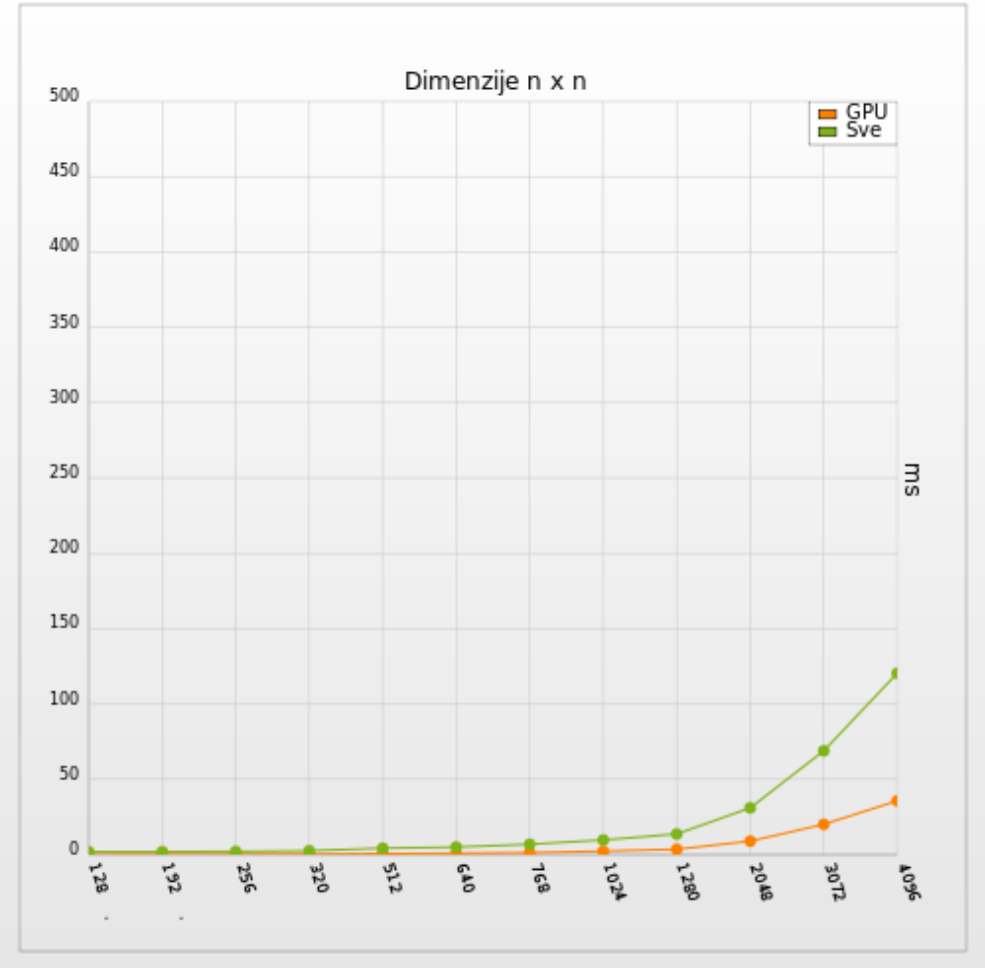



**Slika 6.6:** *Najbrža OpenCL implementacija vs. C++ implementacija vs. SSE implementacija (NVIDIA GT240).*

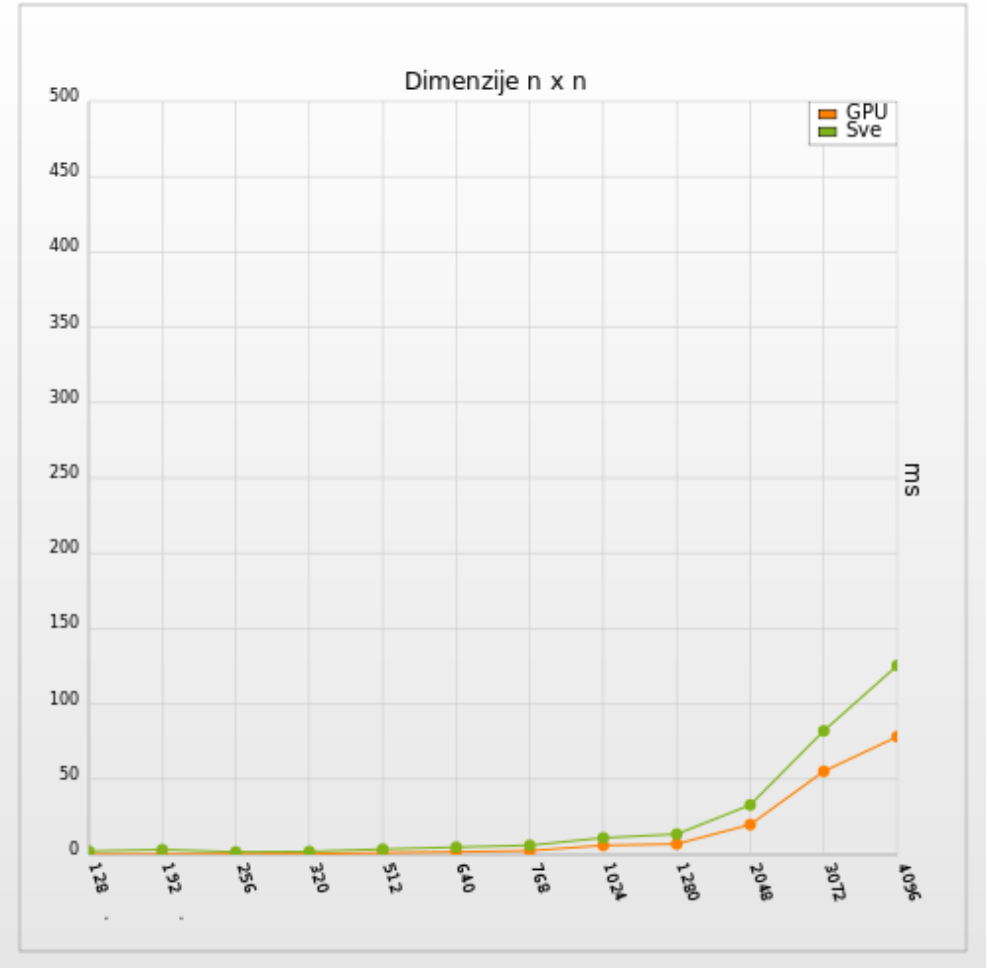



**Slika 6.7:** *Najbrža OpenCL implementacija Convolution_v412.cl (ATI Radeon HD 5670).*

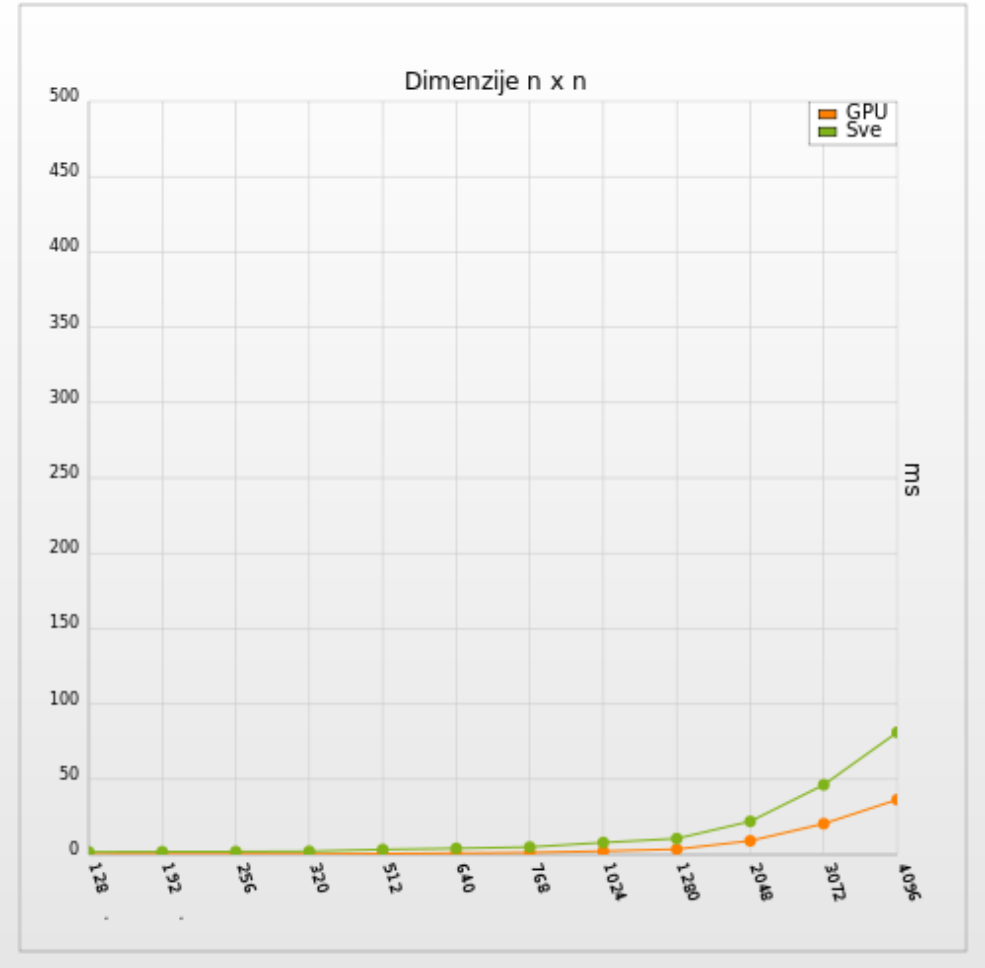
Sl. 6.7, sl. 6.8, sl. 6.9 i sl. 6.10 za svaku implementaciju prikazuju odnos vremena izvođenja na grafičkoj kartici (narančasta linija - „GPU") i cjelokupnog vremena izvođenja (zelena linija - „Sve"). U cjelokupno vrijeme izvođenja uračunato je vrijeme potrebno za stvaranje OpenCL memorijskih objekata, postavljanje argumenata jezgrenim funkcijama, definiranje radnog prostora, prijenos podataka do GPU jedinice, izvođenje jezgrenih funkcija ConvolutionRow i ConvolutionColumn, te dohvat polja koji sadrži rezultate konvolucije iz globalne memorije grafičke kartice u radnu memoriju računala domaćina.




**Slika 6.8:** *Najbrža OpenCL implementacija Convolution_v411.cl (NVIDIA GT240).*



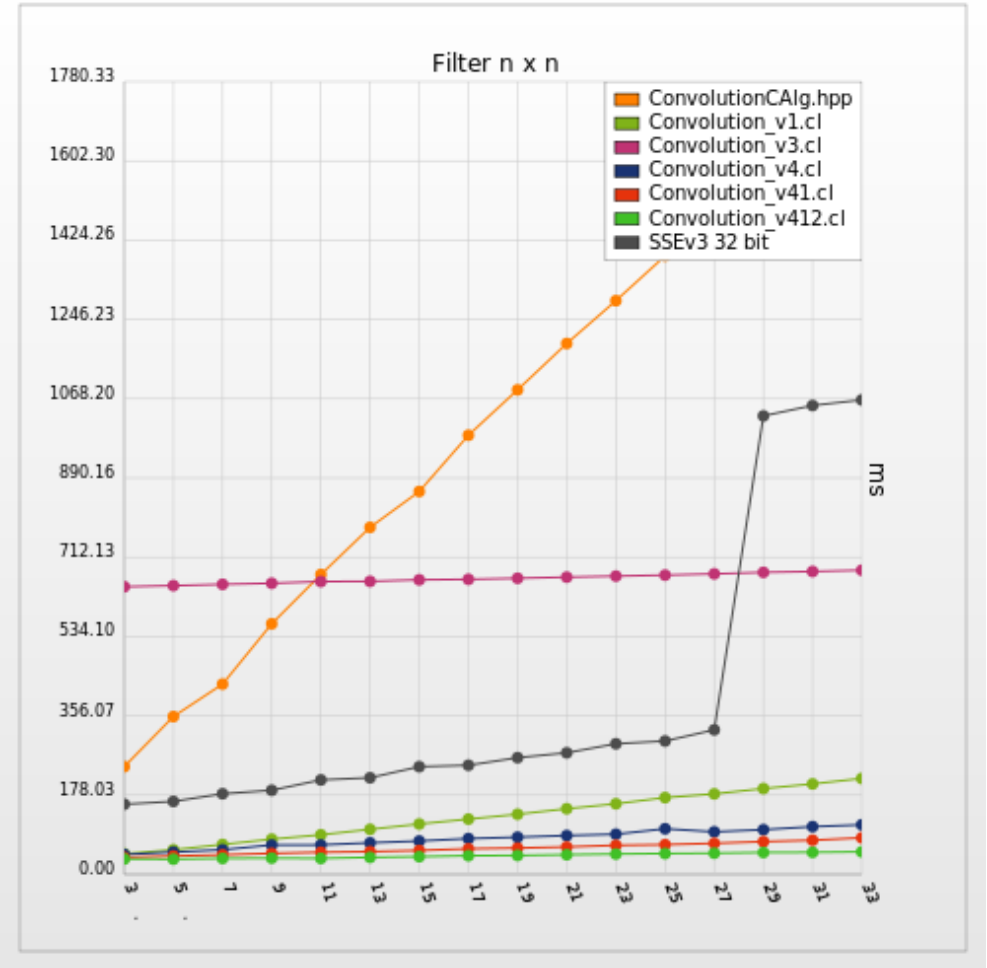

**Slika 6.9:** *Convolution_v51.cl - implementacija s 8 bitnim cjelobrojnim vrijednostima (ATI Radeon HD 5670).*



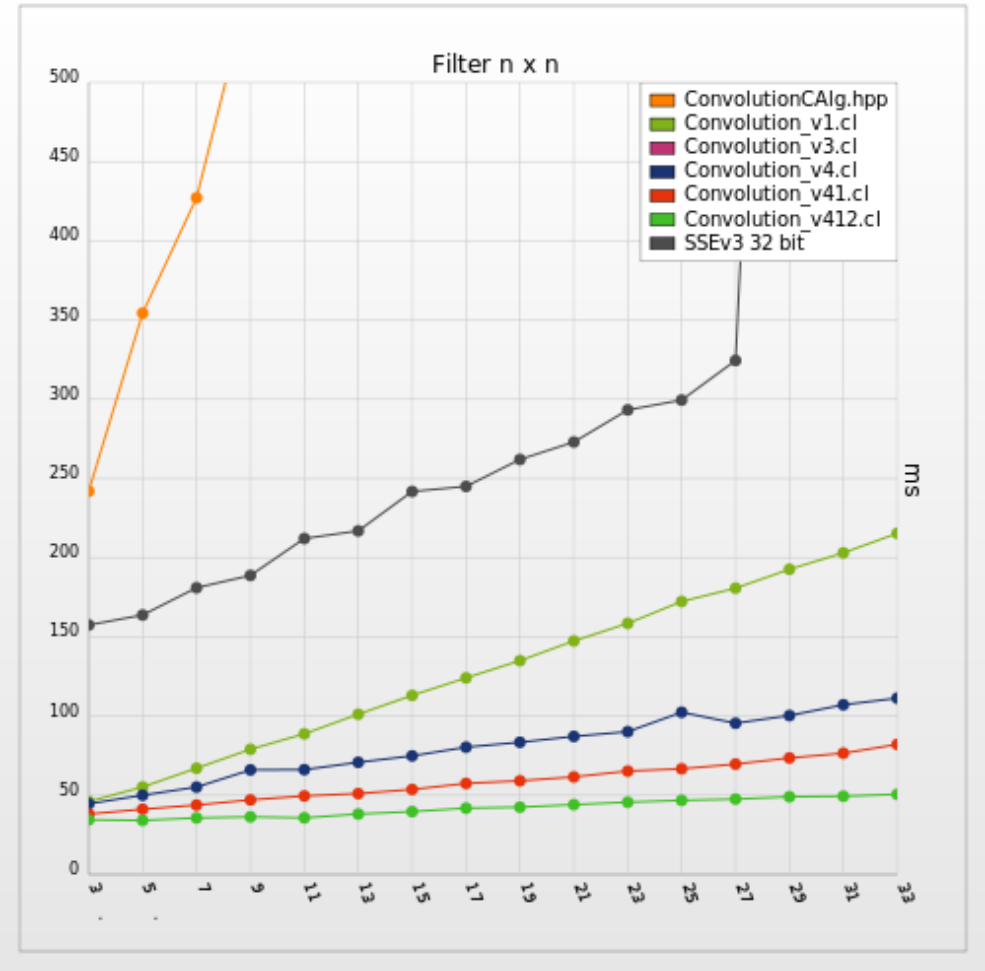

**Slika 6.10:** *Convolution_v51.cl - implementacija s 8 bitnim cjelobrojnim vrijednostima (NVIDIA GT240).*

## 6.2 Test 2 - po veličini filtra

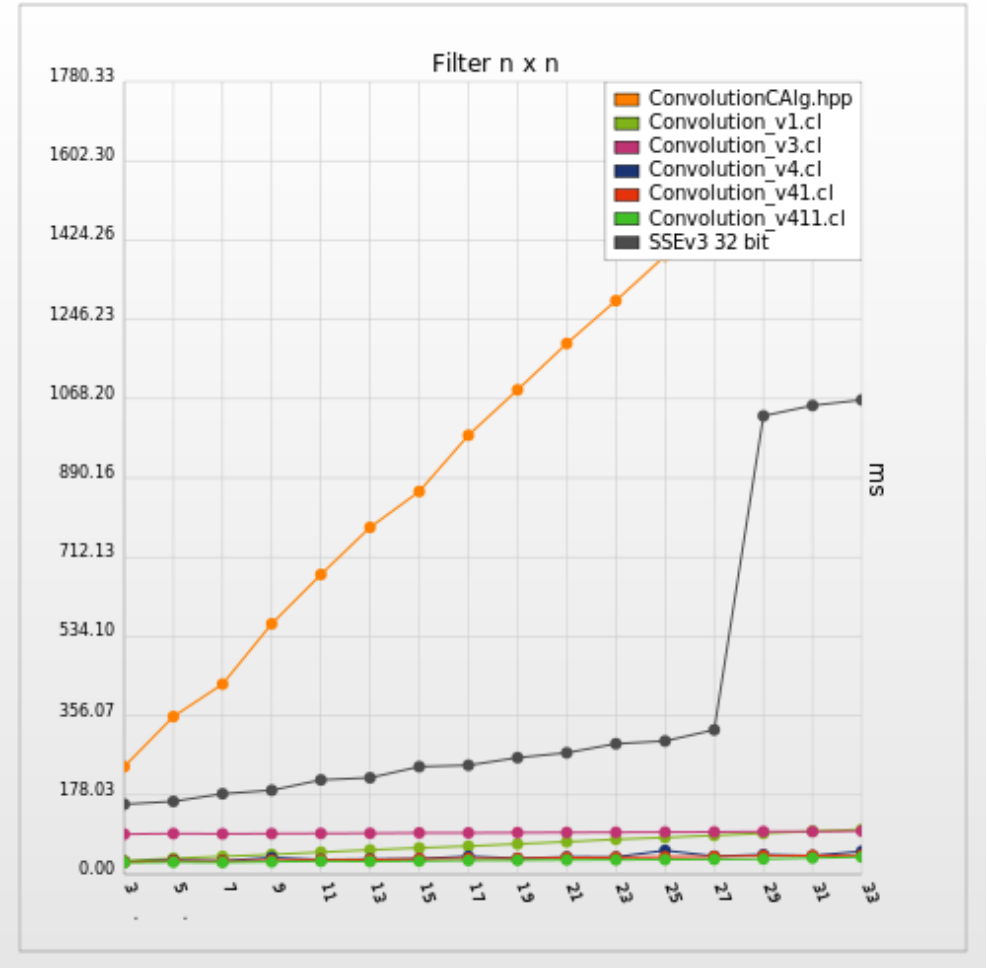
Test 2 proveden je nad različitim veličinama filtara, od $3 \times 3$ do $33 \times 33$. Ulazna slika je konstantna i dimenzije iznose $2048 \times 2048$. Apscisa predstavlja veličinu filtra, dok ordinata predstavlja vrijeme izvođenja u milisekundama.



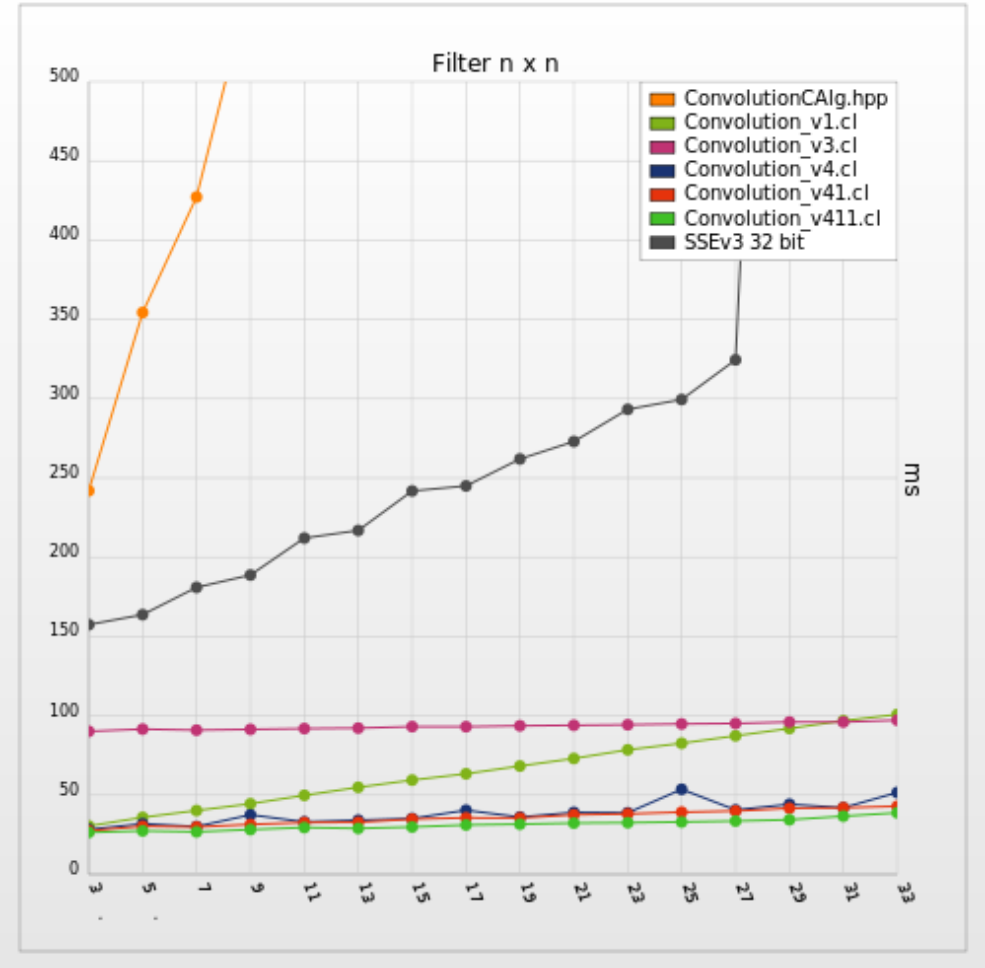

**Slika 6.11:** *Rezultati svih implementacija (ATI Radeon HD 5670).*



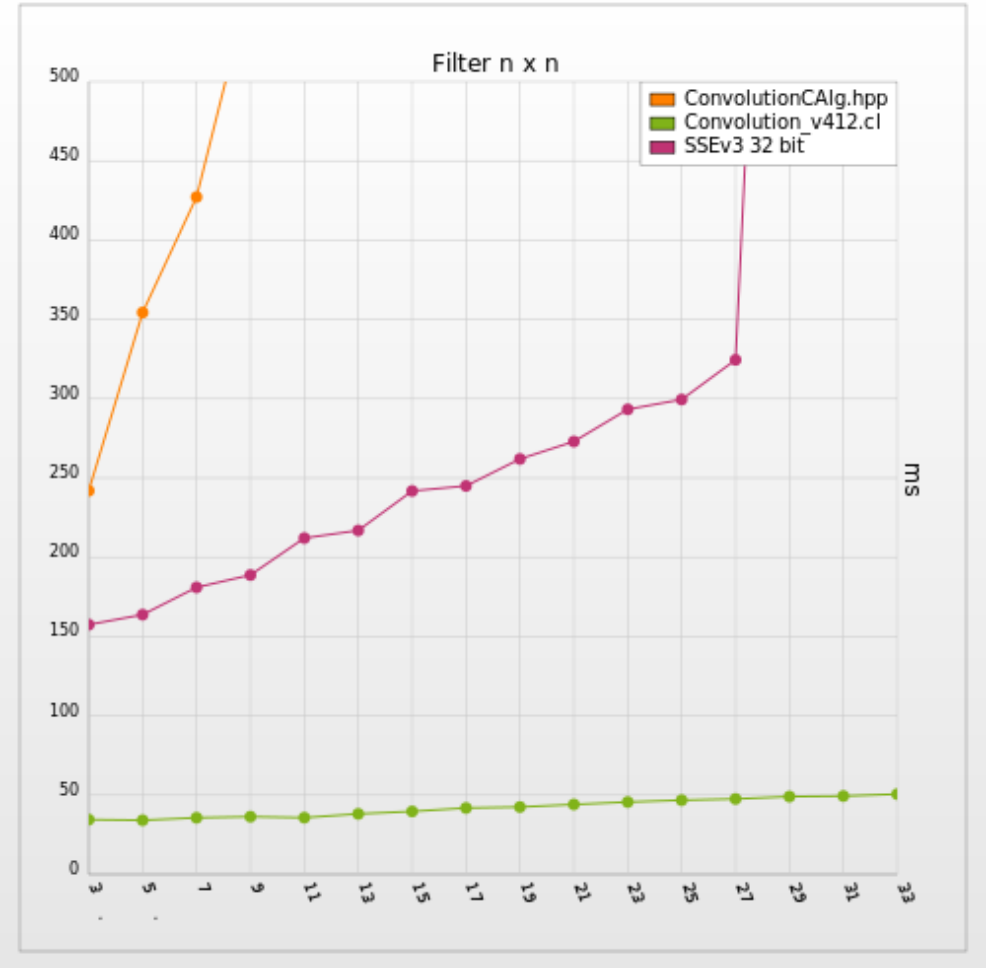

**Slika 6.12:** *Rezultati svih implementacija (ATI Radeon HD 5670) - uvećani prikaz.*



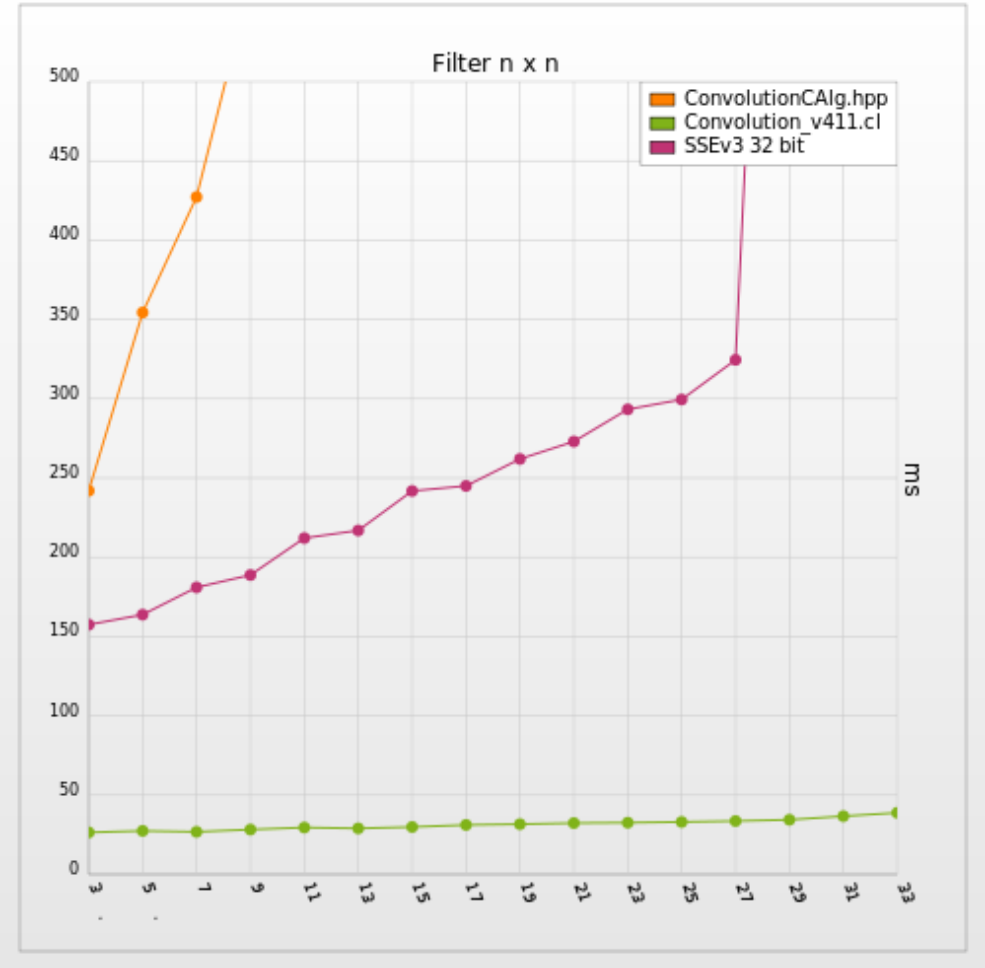

**Slika 6.13:** *Rezultati svih implementacija (NVIDIA GT240).*



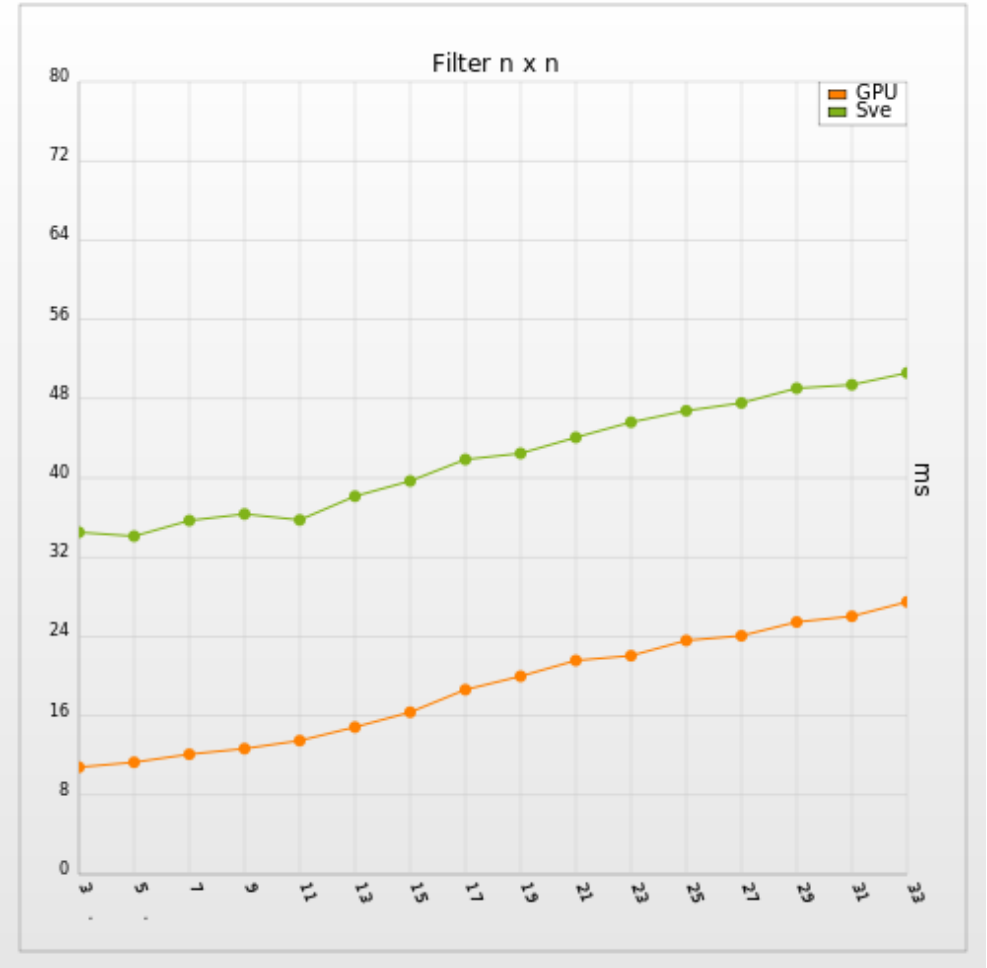

**Slika 6.14:** *Rezultati svih implementacija (NVIDIA GT240) - uvećani prikaz.*



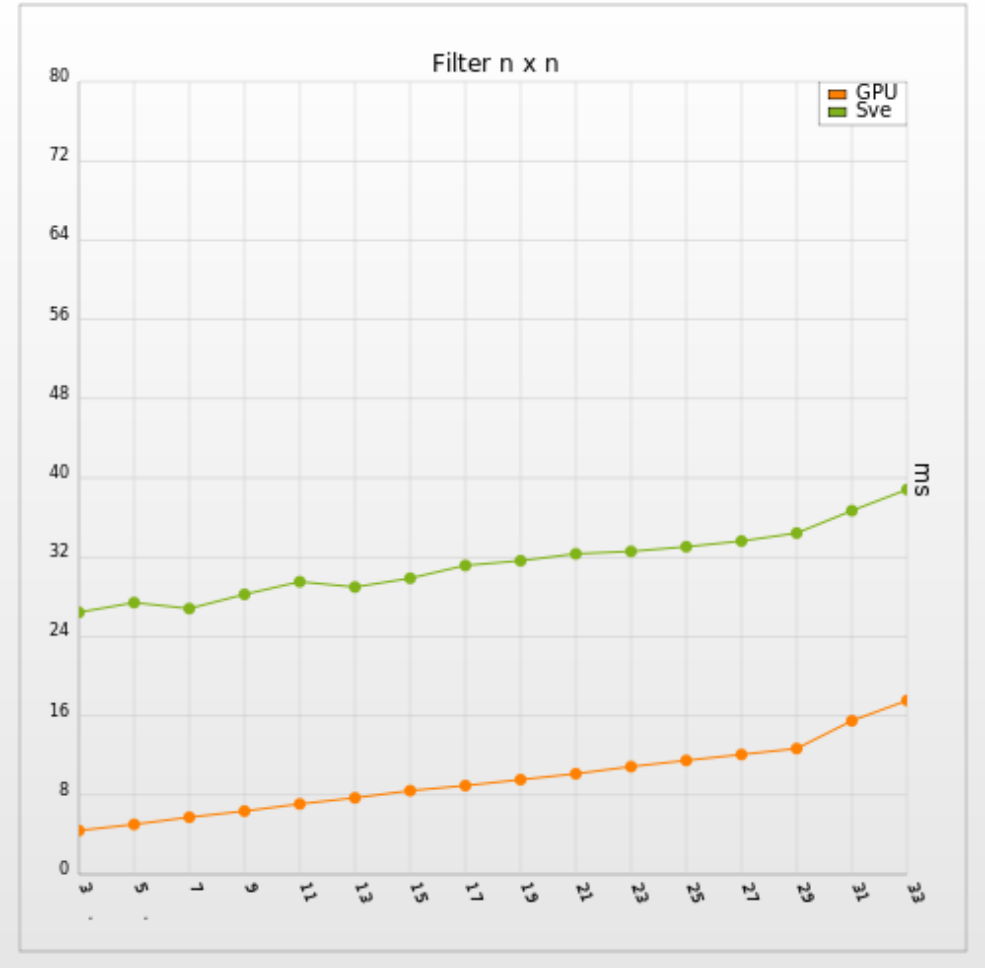

**Slika 6.15:** *Najbrža OpenCL implementacija vs. C++ implementacija vs. SSE implementacija (ATI Radeon HD 5670).*



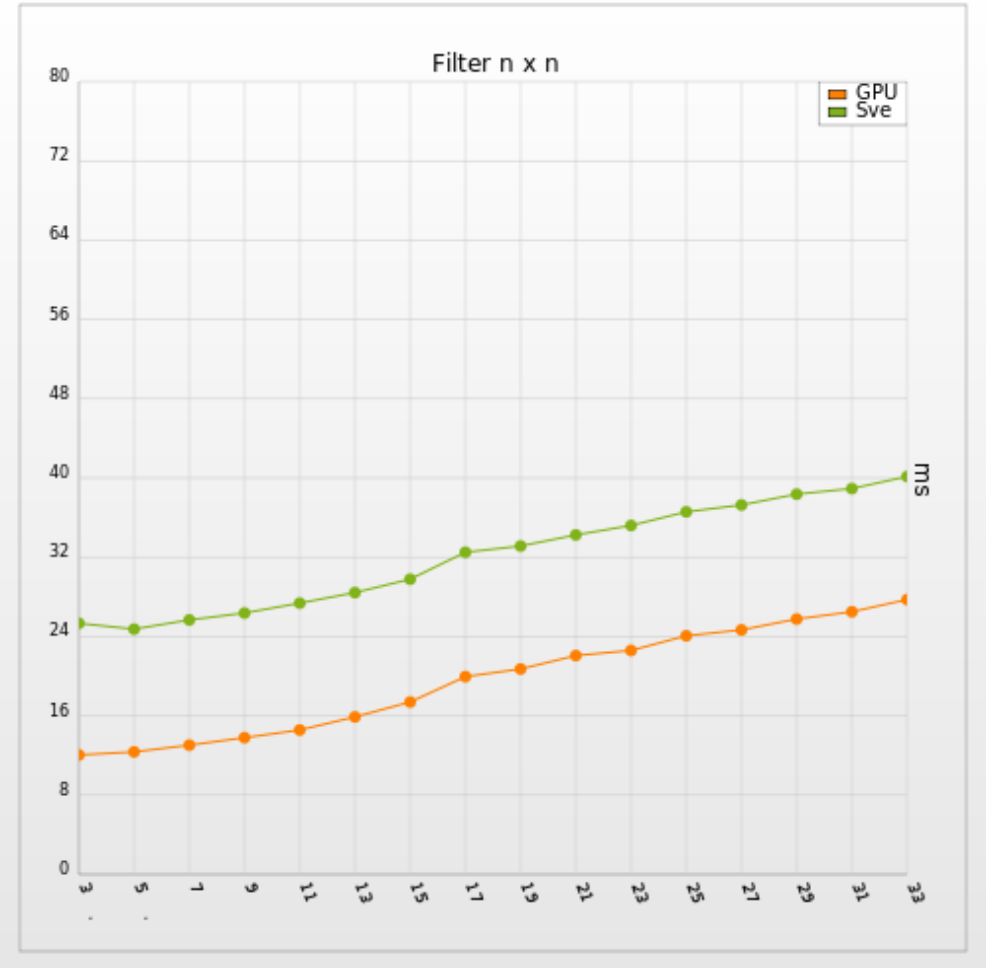

**Slika 6.16:** *Najbrža OpenCL implementacija vs. C++ implementacija vs. SSE implementacija (NVIDIA GT240).*



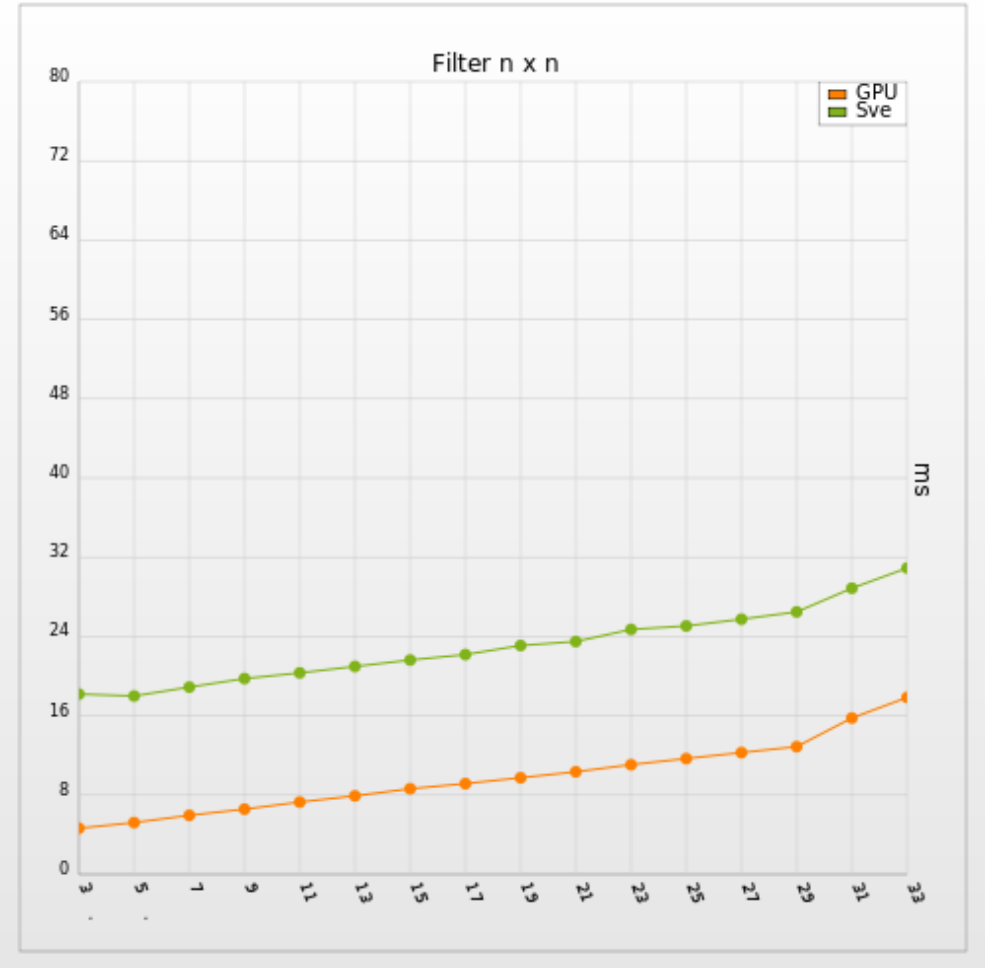

**Slika 6.17:** *Najbrža OpenCL implementacija Convolution_v412.cl (ATI Radeon HD 5670).*




**Slika 6.18:** *Najbrža OpenCL implementacija Convolution_v411.cl (NVIDIA GT240).*




**Slika 6.19:** *Convolution_v51.cl - implementacija s 8 bitnim cjelobrojnim vrijednostima (ATI Radeon HD 5670).*




**Slika 6.20:** *Convolution_v51.cl - implementacija s 8 bitnim cjelobrojnim vrijednostima (NVIDIA GT240).*

## 6.3 Rasprava o rezultatima

Ako se pogledaju grafovi implementacija Convolution_v411.cl i Convolution_v51.cl, uz pretpostavku da se razmatra vrijeme izvođenja na samom grafičkom koprocesoru (narančasta funkcija), može se ustanoviti da je NVIDIA GT240 približno 2.1× brža od ATI Radeon HD 5670 grafičke kartice. Konkretno vrijeme izvođenja, za sliku dimenzija $2048 \times 2048$ i veličinu filtra $17 \times 17$ na ATI Radeon HD 5670 iznosi 18.6 ms, dok na NVIDIA GT240 iznosi 8.9 ms. Ovakav rezultat može se smatrati donekle očekivanim jer NVIDIA GT240 ima više procesnih jezgara, a i implementirani algoritam Convolu-

tion_v411 više je naklonjen arhitekturi NVIDIA GT240, naime nisu iskorišteni vektorski procesni elementi na ATI Radeon HD 5670. Ukupno vrijeme izvođenja (prijenos podataka + izvođenje na GPU + dohvat rezultata) na NVIDIA GT240 iznosi 31.2 ms, dok na ATI Radeon HD 5670 iznosi 41.9 ms.

**Tablica 6.5:** *C++ vs. SSE vs. OpenCL (dimenzije ulazne slike: 2048 × 2048, filtra: 17 × 17).*

| **Implementacija** | **Vrijeme izvođenja (ms)** |
|---|---|
| ConvolutionCAlg | 968.2 |
| SSE v3 | 245.0 |
| Convolution_v411 | 31.2 |

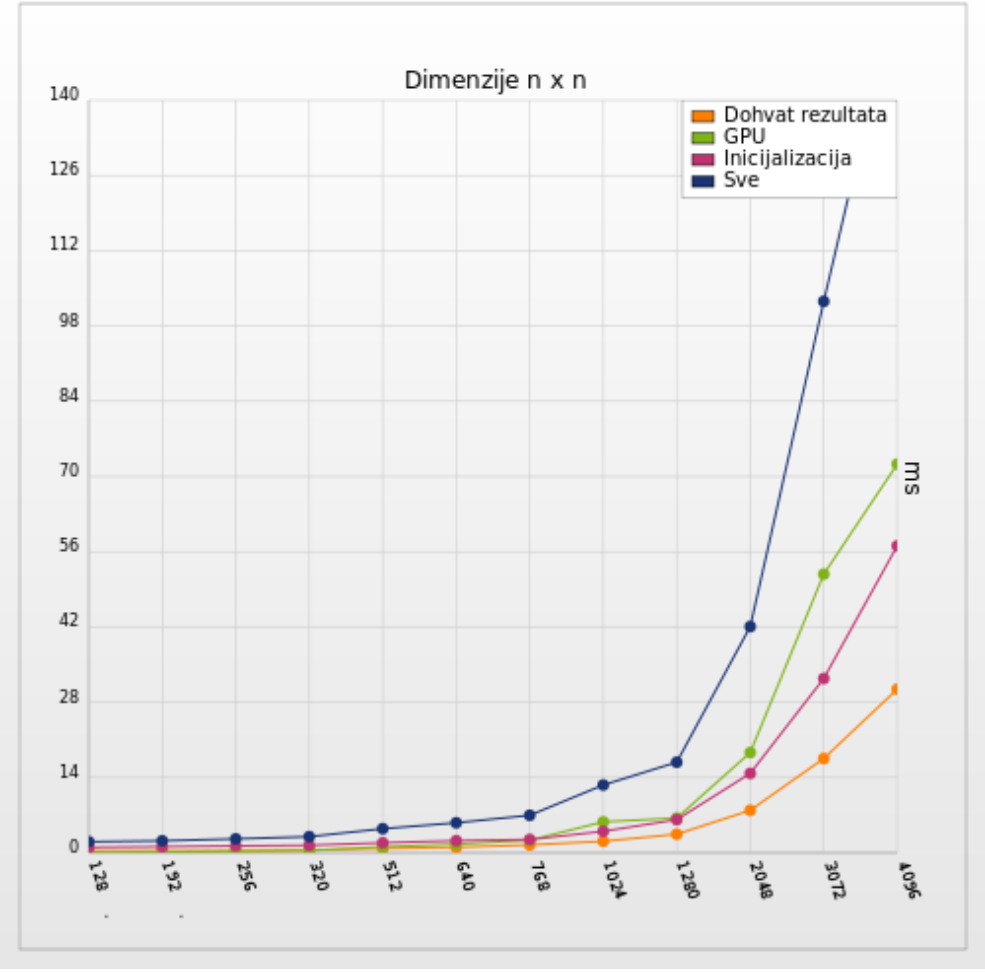
Implementacija Convolution_v411 na NVIDIA GT240 je 7.85× brža od SSE v3 implementacije, kojoj je potrebno 245 ms za obradu slike. Implementaciji u C++ potrebno je 968.2 ms za obradu slike dimenzija 2048 × 2048, s filtrom dimenzija 17 × 17, čime je algoritam Convolution_v411 izveden na NVIDIA GT240 približno brži 31×.

**Tablica 6.6:** *Vremena izvođenja po segmentima (dimenzije ulazne slike: 2048 × 2048, filtra: 17 × 17).*

| | NVIDIA GT240 (ms) | ATI Radeon HD 5670 (ms) |
|---|---|---|
| (1) Prijenos podataka (HOST $->$ GPU) | 14.2 | 14.4 |
| (2) ConvolutionRow | 3.0 | 8.4 |
| (3) ConvolutionColumn | 5.9 | 10.2 |
| (4) GPU: (2) + (3) | 8.9 | 18.6 |
| (5) Dohvat rezultata (GPU $->$ HOST) | 7.9 | 8.1 |
| (6) Ukupno | 31.1 | 41.9 |

Međutim valja primijetiti kako u rezultatima i za ATI Radeon HD 5670 i za NVIDIA GT240, prijenos podataka iz radne memorije računala domaćina u memoriju grafičke

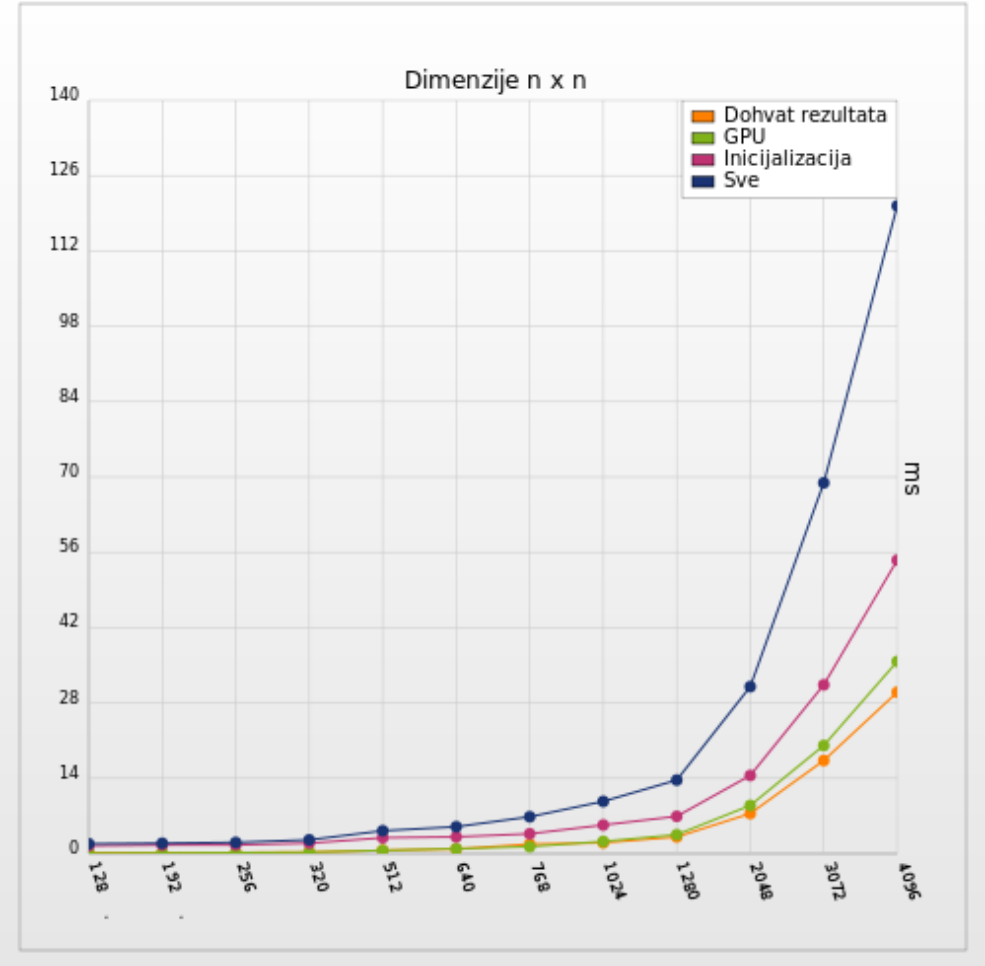
kartice uzima poprilično puno vremena, približno 2/3 od ukupnog vremena izvođenja (vidi sl. 6.21 i sl. 6.22). Ovakva poprilično velika memorijska latencija predstavlja problem, stoga bi trebalo modelirati jezgrene funkcije koje obavljaju puno složeniji posao od izračunavanja konvolucije slike.




**Slika 6.21:** *Najbrža OpenCL implementacija Convolution_v412.cl (ATI Radeon HD 5670).*

**Tablica 6.7:** *Pojašnjenje legende na slikama 6.21 i 6.22.*

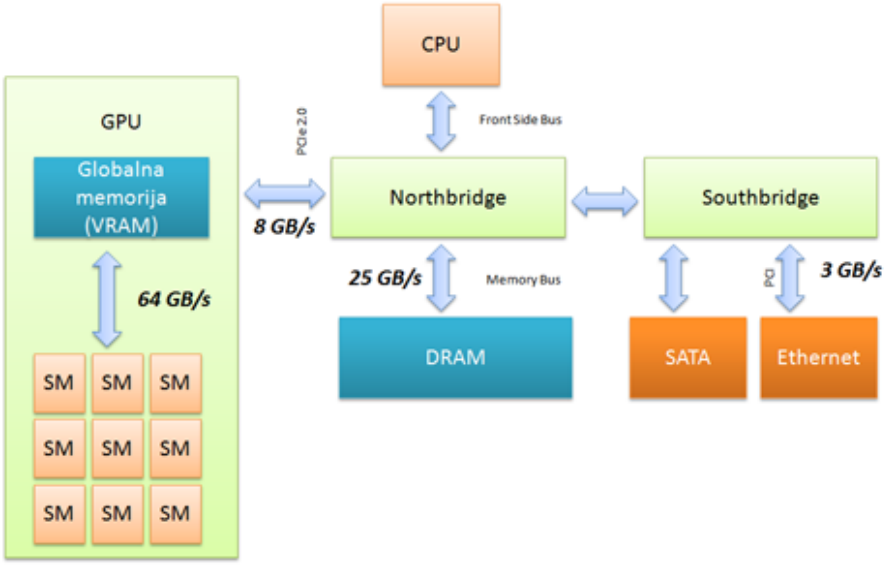
| | |
|---|---|
| (1) Inicijalizacija | **Prijenos podataka (HOST − > GPU)** |
| (2) GPU | **Izvođenje na GPU** |
| (3) Dohvat rezultata | **Dohvat rezultata (GPU − > HOST)** |
| (4) Sve | **Sveukupno trajanje izvođenja** |




**Slika 6.22:** *Najbrža OpenCL implementacija Convolution_v411.cl (NVIDIA GT240).*

Današnje matične ploče s grafičkom karticom komuniciraju kroz sučelja PCI-E 2.0, upravo taj dio je usko grlo, u obzir treba uzeti i zauzetost memorijske sabirnice

radne memorije računala domaćina (engl. *memory bus*) pa je memorijska propusnost od grafičke kartice do radne memorije manja od teoretskih 8 GB/s (vidi sl. 6.23). Stoga treba izbjegavati česte prijenose iz radne memorije računala u memoriju GPU i obratno, bolje je imati jedan veliki prijenos podataka nego više pojedinačnih malih prijenosa.




**Slika 6.23:** *Shematski prikaz računala domaćina s teoretskim vrijednostima propusnosti sabirnica.*

# Poglavlje 7

# Rezultati implementacija afine transformacije i homografije

Provedena su dva testa za afinu transformaciju i homografiju. Prvi test se provodi na ulaznoj slici (vidi sl. 7.1) dimenzija 720 × 576, dok se drugi test provodi na istoj slici (vidi sl. 7.1), ali smanjenih dimenzija, tj. 360 × 288. Algoritmi su pokrenuti 1000 puta na grafičkoj kartici NVIDIA GT240, nakon čega je uzeto prosječno vrijeme izvođenja.

**Slika 7.1:** *Ulazna slika.*

## 7.1 Afina transformacija

Na slici 7.2 prikazana je izlazna slika (720 × 576) dobivena afinom transformacijom ulazne slike (vidi sl. 7.1) dimenzija 720 × 576, pri čemu je transformacijska tablica:

$$A_1 = \begin{bmatrix} 2 & 1.5 & -800 \\ 0 & 2 & -300 \end{bmatrix}. \tag{7.1}$$

**Slika 7.2:** *Izlazna slika afine trasformacija (720 × 576, $A_1$).*

Na slici 7.2 prikazana je izlazna slika (360 × 288) dobivena afinom transformacijom

ulazne slike (vidi sl. 7.1) dimenzija 360 × 286, gdje je transformacijska tablica:

$$A_2 = \begin{bmatrix} 2 & 1.5 & -300 \\ 0 & 2 & -100 \end{bmatrix}. \tag{7.2}$$

**Slika 7.3:** *Izlazna slika afine trasformacija (360 × 288, $A_2$).*

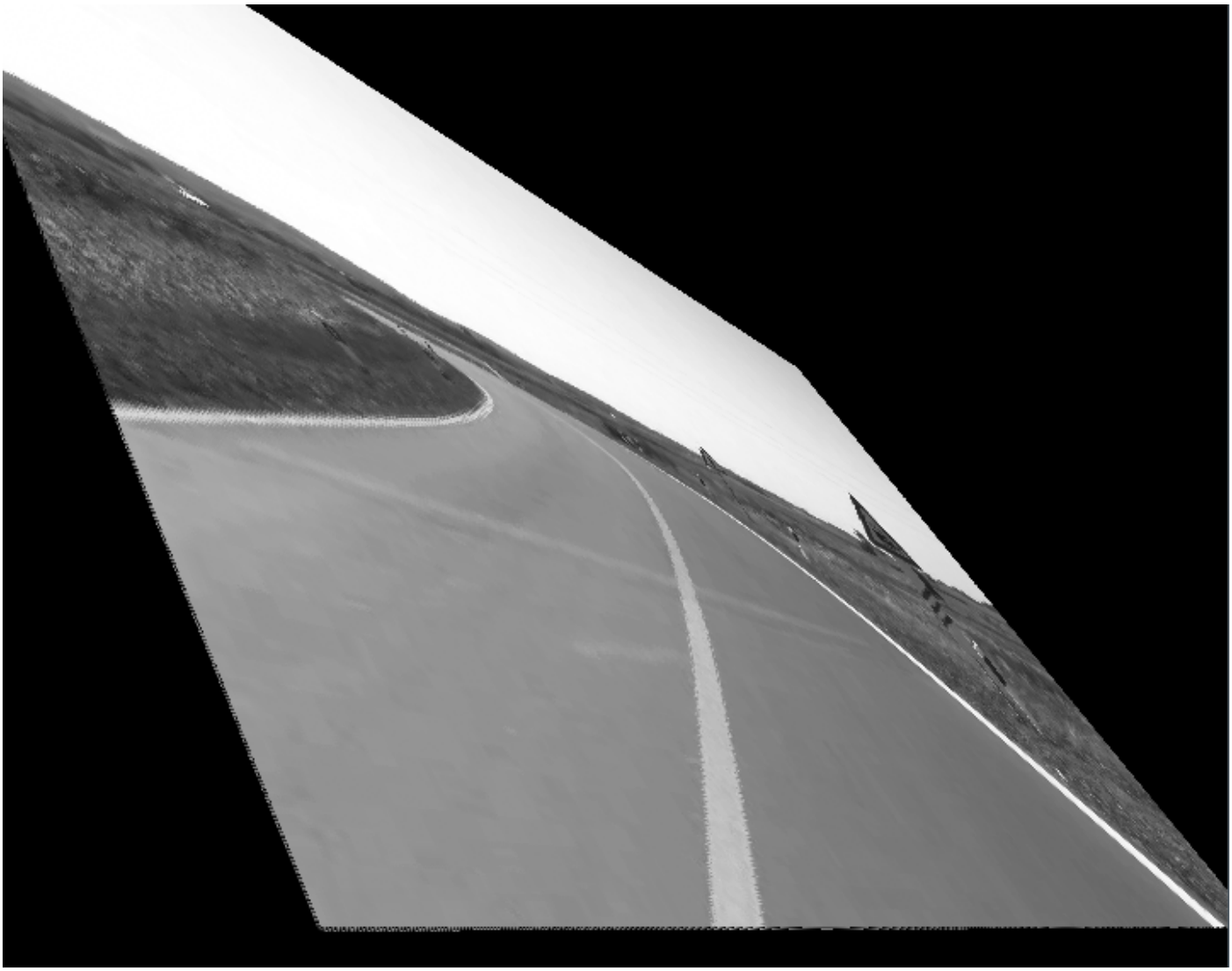
**Tablica 7.1:** *Vremena izvođenja implementacije afine transformacije.*

| | | Vremena izvođenja (ms) | |
|---|---|---|---|
| Ulazna slika | Transformacija matrica | GPU | Prijenos + GPU |
| 360 × 288 | $A_2$ | 0.09 | 1.00 |
| 720 × 576 | $A_1$ | 0.29 | 1.59 |

## 7.2 Homografija

Slika 7.4 prikazuje izlaznu sliku (720 × 576) dobivenu homografijom ulazne slike (vidi sl. 7.1) dimenzija 720 × 576, pri čemu je transformacijska tablica:

$$H_1 = \begin{bmatrix} 3 & 1.2 & -600 \\ 0 & 3 & -100 \\ -0.01 & -0.01 & 10 \end{bmatrix}. \tag{7.3}$$

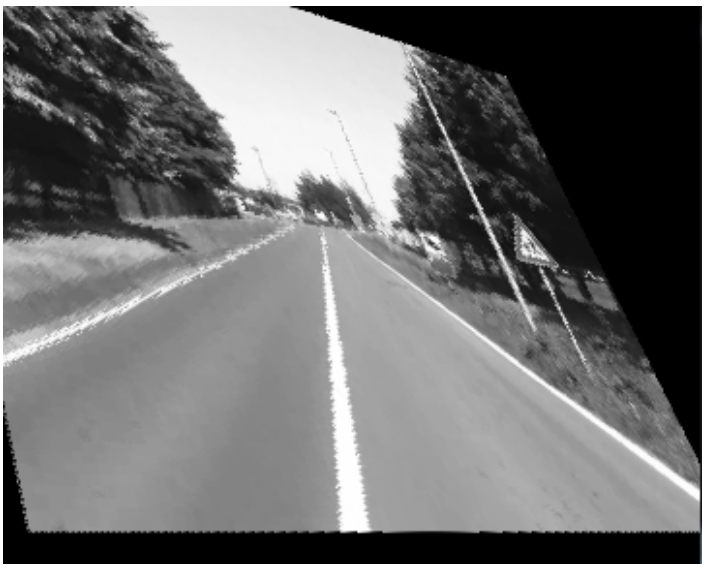


**Slika 7.4:** *Izlazna slika homografije (720 × 576, $H_1$).*

Slika 7.5 prikazuje izlaznu sliku (360 × 288) dobivenu homografijom ulazne slike

(vidi sl. 7.1) dimenzija 360 × 288, pri čemu je transformacija tablica:

$$H_2 = \begin{bmatrix} 6 & 1.2 & -100 \\ 0 & 6 & -100 \\ -0.01 & -0.01 & 10 \end{bmatrix}. \tag{7.4}$$



**Slika 7.5:** *Izlazna slika homografije (360 × 288, $H_2$).*

**Tablica 7.2:** *Vremena izvođenja implementacije homografije.*

| | | Vremena izvođenja (ms) | |
|---|---|---|---|
| Ulazna slika | Transformacija matrica | GPU | Prijenos + GPU |
| 360 × 288 | $H_2$ | 0.10 | 1.03 |
| 720 × 576 | $H_1$ | 0.35 | 1.72 |

# Poglavlje 8

# Zaključak

Rezultati eksperimenta pokazuju kako implementacija konvolucije razvijena u OpenCL-u i izvedena na grafičkoj kartici postiže višestruko povoljnije vremenske performanse u odnosu na implementacije koje se izvode na središnjoj procesorskoj jedinici (CPU). Implementacije afine transformacije i homografije također postižu zadovoljavajuće vremenske rezultate.

Shodno rezultatima može se zaključiti kako arhitektura modernih grafičkih kartica zajedno s tehnologijama poput CUDA-e i OpenCL-a otvara prostor za ubrzanje mnogih algoritama kako bi se postigla brža vremena izvođenja.

Pregledom rezultata ustanovljeno je kako značajno vrijeme otpada na prijenosa podataka iz radne memorije računala domaćina u radnu memoriju grafičke kartice i na prijenos u obrnutom smjeru. Na prijenos podataka odlazi 66% ukupnog vremena za implementaciju konvolucije, dok kod implementacija afine transformacije i homografije taj postotak penje se i do 90%.

Dakle najpogodniji algoritmi za izvođenje na GPU-u su algoritmi koji osim paralelizacijskih svojstava, sadrže i zahtjevne „poslove" kako bi se što više iskoristila procesorska snaga protočnih procesora grafičke kartice, te bi time vrijeme utrošeno na prijenos podataka postala zanemariva stavka. Implementacije afine transfomacije i homografije su primjeri prejednostavnih algoritama. Naime protočni procesori grafičke kartice relativno brzo izvedu ovakve jednostavne algoritme, pa vrijeme utrošeno na prijenos podataka dolazi do izražaja.

Veliki utjecaj prijenosa podataka na ukupno vrijeme izvođenja moguće je smanjiti uporabom dviju dretvi u aplikaciji, jedna koja bi slala podatke na GPU, dok bi druga dretva slala zahtjeve za izvođenje jezgrenih funkcija GPU jedinici. Uz pretpostavku

da je postupak moguće primijeniti na slijedu ulaznih podataka. Time bi izvođenje jezgrenih funkcija na grafičkoj kartici sakrilo latenciju uzrokovanu prijenosom podataka, jer bi slijed ulaznih podataka stizao u radnu memoriju grafičke kartice nezavisno o izvođenju jezgrenih funkcija. Ovakvo rješenje moguće je primijeniti na implementaciji konvolucije, dok kod implementacija afine transfomracije i homografije latencija je jednostavno prevelika u odnosu na vrijeme izvođenja na GPU-u.

Također primjećen je određen gubitak kvalitete slike prilikom primjene afine transformacije ili homografije. Do gubitka kvalitete dolazi zbog tzv. aliasinga, naime bilinearna interpolacija je prilično gruba interpolacija, pa bi valjalo primijeniti složenije postupke interpoliranja kako bi izlazna slika što manje gubila na kvaliteti, a time bi se dodatno zaposlili protočni procesori grafičke kartice, čime bi se smanjio omjer vremena utrošenog na prijenosa podataka u odnosu na vrijeme izvođenja jezgrene funkcije.

Koristeći GPGPU paradigmu i arhitekturu grafičkih kartica moguće je ubrzati dobar dio algoritama računalnog vida. U budućim radovima valjalo bi implementirati algoritam za detekciju lokalnih značajki na slikama, tzv. SIFT (engl. *scale-invariant feature transform*) algoritam. Algoritam za detekciju objekata na slici tzv. Viola & Jones algoritam doima se djelomično prikladnim za izvođenje na GPU-u. Naime prve faze (kaskade) Viola & Jones algoritma posebno su pogodne za izvođenje na GPU jedinici. U prvim fazama prisutno je puno prozora pa se može definirati puno radnih elemenata, dok zadnje faze nisu prikladne za izvođenje na GPU jedinici zbog divergiranja. Naime prolaskom kroz kaskade većina prozora otpada čime bi radni elementi dodijeljeni tim prozorima, u preostalima fazama, bili u stanju bez posla pa bi bespotrebno zauzimali protočne procesore. Stoga bi valjalo implementirati kombiniranu implementaciju koja bi podijelila algoritam na dva dijela. Prvi dio s relativno puno prozora i relativno malo divergiranja, izvodio bi se na GPU jedinici. Nakon što bi kroz faze preživio mali broj prozora, tada bi izvođenje preostalih faza algoritma (s malim broj preživjelih prozora), preuzeo procesor opće namjene.

# Literatura


[1] „NVIDIA Fermi Compute Architecture WhitePaper", 2. listopada 2009., *„NVIDIA's Next Generation Cuda Compute Architecture: Fermi"*, http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf, 23. svibnja 2010.

[2] „NVIDIA Cuda Programming Guide", 2. veljače 2010., *„NVIDIA Cuda Programming Guide Version 3.0"*, http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_ProgrammingGuide.pdf, 13. ožujka 2010.

[3] „NVIDIA CUDA Best Practices Guide", 4. veljače 2010., *„NVIDIA CUDA Best Practices Guide Version 3.0"*, http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_BestPracticesGuide.pdf, 13. ožujka 2010.

[4] „OpenCL Programming Guide for the CUDA Architecture", 18. veljače 2010., *„OpenCL Programming Guide for the CUDA Architecture Version 2.3"*, http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_OpenCL_ProgrammingGuide.pdf, 19. ožujka.2010

[5] „NVIDIA OpenCL Best Practices Guide", 31. kolovoza 2009., *„Optimization NVIDIA OpenCL Best Practices Guide Version 2.3"*, http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_OpenCL_BestPracticesGuide.pdf, 19. ožujka 2010.

[6] „NVIDIA OpenCL Optimization", 13. srpnja 2009., *„NVIDIA OpenCL Optimization"*, http://developer.download.nvidia.com/CUDA/training/NVIDIA_GPU_Computing_Webinars_Best_Practises_For_OpenCL_Programming.pdf, 19. ožujka 2010.

[7] „Programming Guide", 14. svibnja 2010., *„ATI Stream Computing OpenCL Programming Guide"*, http://developer.amd.com/gpu_assets/ATI_Stream_SDK_OpenCL_Programming_Guide.pdf, 20. svibnja 2010.

[8] Podlozhnyuk, V., „Image Convolution with Cuda", 23. lipnja 2007., *„Image Convolution with Cuda"*, http://www.naic.edu/~phil/hardware/nvidia/doc/src/convolutionSeparable/doc/convolutionSeparable.pdf, 14. travnja 2010.

[9] Fialka O., Čadik M., „FFT and Convolution Performance in Image Filtering on GPU", 6. lipnja 2006., http://www.cgg.cvut.cz/~cadikm/papers/cadikm-iv06-gpu.pdf, 21. svibnja 2010.

[10] Bordoloi, U., „Image Convolution Using OpenCL™", 13. listopada 2009., *„Image Convolution Using OpenCL™ - A Step-by-Step Tutorial"*, http://developer.amd.com/GPU/ATISTREAMSDK/IMAGECONVOLUTIONOPENCL/pages/ImageConvolutionUsingOpenCL.aspx, 7. travnja 2010.

[11] Ahn, S. H., „Proof of Separable Convolution 2D", *„Proof of Separable Convolution 2D"*, http://www.songho.ca/dsp/convolution/convolution2d_separable.html, 14. travnja 2010.

[12] Howes, L., „OpenCL Parallel computing for CPUs and GPUs", 1. travnja 2010., *„OpenCL Parallel computing for CPUs and GPUs"*, http://developer.amd.com/gpu_assets/OpenCL_Parallel_Computing_for_CPUs_and_GPUs_201003.pdf, 3. travnja 2010.

[13] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Skadron, K., „A Performance Study of General-Purpose Applications on Graphics Processors Using CUDA", 16. srpnja 2008., *„A Performance Study of General-Purpose Applications on Graphics Processors Using CUDA"*, http://www.cs.virginia.edu/~skadron/Papers/cuda_jpdc08.pdf, 5. travnja 2010.

[14] „OpenCL Taking the graphics processor beyond graphics", 31. srpnja 2009., *„OpenCL Taking the graphics processor beyond graphics."*, http://images.apple.com/macosx/technology/docs/OpenCL_TB_brief_20090903.pdf, 5. travnja 2010.

[15] „OpenCL", 31. srpnja 2009., *„OpenCL - Wikipedia, the free encyclopedia."*, http://en.wikipedia.org/wiki/OpenCL, 1. lipnja 2010.

[16] Hadjic, S., „Optimiranje konvolucije slike vektorskim instrukcijama u aritmetici s nepomičnim zarezom", Završni rad, Sveučilište u Zagrebu, Fakultet elektrotehnike i računarstva, 2009. godina

[17] „The OpenCL Specification", 6. listopada 2009., *„The OpenCL Specification Version 1.0"*, http://www.khronos.org/registry/cl/specs/opencl-1.0.pdf, 2. svibnja 2010.

# Izvedba algoritama računalnog vida na grafičkim procesorima


## Sažetak

Ovaj rad opisuje izvedbe konvolucije sa separabilnim filtrima, afine transformacija i homografije koristeći općenamjensko programiranje na grafičkim procesorima (GPGPU) s ciljem postizanja većih vremenskih performansi. Dizajnirani algoritmi separabilne konvolucije, afine transformacije i homografije implementirani su koristeći OpenCL radni okvir.

Ovaj rad opisuje temeljne koncepte programiranja u OpenCL-u, opisuje kako oblikovati algoritme za izvođenje na GPU-u, opisuje implementirane algoritme i prikazuje eksperimentalne rezultate.

Dobiveni eksperimentalni rezultati pokazuju da postoji potencijal za korištenje razvijenih algoritama u složenijim algoritmima računalnog vida. OpenCL implementacija separabilne konvolucije postiže ubrzanje od 30 puta u odnosu na C++ implementaciju, i 7.5 puta ubrzanje u odnosu na SSE implementaciju. Implementacije afine transformacije i homografije također postižu značajna ubrzanja.



## Ključne riječi

GPGPU, parelelizacija podataka, OpenCL, GPU, računalni vid, separabilna konvolucija, afina transformacija, homografija

# Implementation of computer vision algorithms on graphics processing units


## Abstract

This paper describes implementations of separable convolution, affine transformation and homography using a general purpose graphics hardware computing (GPGPU) in order to achieve greater time performances. The designed algorithms for separable convolution, affine transformation and homography have been implemented using OpenCL framework.

This work gives key concepts of OpenCL framework, discusses how to design algorithms for executing on a GPU, describes implemented algorithms and gives experimental results.

The experimental results show that developed implementations have a potential of using in computer vision algorithms. The OpenCL implementation of separable convolution achieves more than 30 time speed up versus the C++ implementation and 7.5 time speed up versus the SSE implementation. The affine implementation and the homography implementation also achieve a significant speed up.



## Keywords

GPGPU, data parallelism, OpenCL, GPU, computer vision, separable convolution, affine transformation, homography