Paralelno računanje na grafičkim karticama

Enzo, Vituri

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Enzo Vituri

PARALELNO RAČUNANJE NA GRAFIČKIM KARTICAMA

ZAVRŠNI RAD

Varaždin, 2018.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Enzo Vituri

Matični broj: 44109-15/R

Studij: Informacijski sustavi

PARALELNO RAČUNANJE NA GRAFIČKIM KARTICAMA

ZAVRŠNI RAD

Mentor:

Izv. prof . dr. sc. Ivan Magdalenić

Varaždin, 2018.

Enzo Vituri

Izjavljujem da je moj završni rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor/Autorica potvrdio/potvrdila prihvaćanjem odredbi u sustavu FOI-radovi

_____

Sažetak

U ovome radu ću objasniti općenamjensko računanje na procesorima grafičkih kartica koje se inače izvode na centralnoj procesnoj jedinici. Opisat ću načine na koje se može slati naredbe grafičkim procesorima da izvode željeni programski kod, paralelno, na svim raspoloživim jezgrama grafičkog procesora. Opisat ću način kako podijeliti zadatke u grupe sa vlastitom memorijom, te tako ubrzati proces jer je pristupanje globalnoj memoriji na grafičkom procesoru skupo. Također ću objasniti par algoritama koje sam implementirao, te ih vremenski usporediti s njihovim implementacijama na CPU.

Sadržaj

1. Uvod

Opće je poznata činjenica da procesori grafičke kartice imaju tisuće jezgra, dok centralna procesna jedinica ima do 16. Posljedica toga je da grafičke kartice rade na manjim frekvencijama nego CPU, te se specijaliziraju za obradu slika, dok je CPU „mozak računala“, te je zadužen za centralne jednostavne zadatke. Iz toga intuitivno možemo zaključiti da je procesor grafičke kartice pogodniji za izvođenje više zadataka odjednom, odnosno paralelni rad.

Sa pojavom programabilnih shadera i podržavanjem operacija sa pomičnim zarezom na grafičkim procesorima. Tako su se problemi koji uključuju matrice i vektore mogli lako napisati za GPU zbog mogućnosti obrađivanja više podataka u isto vrijeme, jedan primjer toga je množenje matrica, u kojem je bilo moguće pomnožiti svaki redak u matrici odjednom, te na kraju zbrojiti rezultat. Tako su krenuli razvoji aplikacijskih programskih sučelja (eng. API), kao što su nVidia CUDA i OpenCL, koji su služili za direktan pristup instrukcijama grafičkog procesora i paralelnih elemenata za računanje, za pokretanje kernela za računanje, koji se još nazivaju shaderi, koji su dio koda izvršen direktno na grafičkom procesoru uz pomoć glavnog programa. [9]

U ovome radu ću pobliže objasniti programiranje grafičkih procesora uz pomoć platforme OpenCL, dekompoziciju problema da se može izvoditi na grafičkom procesoru, te neke algoritme koje sam implementirao za grafički procesor i vremenski usporedio sa njihovim implementacijama na CPU.

2. Postupak programiranja na GPU

OpenCL je okvir (eng. Framework) za pisanje programa koji se paralelno pokreću na grafičkom procesoru. Ovaj okvir također definira programski jezik u kojem se pišu spomenuti programi odnosno kerneli. Kernel možemo gledati kao funkciju koja se pokrene paralelno, broj paralelnih pokretanja ovisi o programeru, no ako je broj prevelik neće se moći izvršiti svi kerneli odjednom zbog ograničenja broja jezgri. U nastavku ću pojasniti korake za pisanje glavnog programa, koji će pokretati kernele. [1]

Slika 1 OpenCL Logo [1]

2.1. Početna podešavanja

Za početak potrebno je imati grafičku karticu koja podržava OpenCL, te instalirane upravljačke programe (eng. drivere). Također je potrebno instalirati OpenCL implementaciju za vašu grafičku karticu. nVidia koristi CUDA Toolkit, AMD kartice koriste AMD APP SDK, Intelove kartice koriste Intel OpenCL SDK, a Apple ima svoju implementaciju OpenCL-a. Moguće je instalirati više OpenCL implementacija na sistemu. [1] Ukoliko je sve instalirano dobro, pomoću komande „clinfo“ možete provjeriti verziju OpenCL-a, te grafičku karticu. Na slici 2 je dio mog ispisa funkcije „clinfo“.

```
C:\Users\Enzo>clinfo
Number of platforms:                               1
  Platform Profile:                                FULL_PROFILE
  Platform Version:                                OpenCL 2.1 AMD-APP (2671.3)
  Platform Name:                                   AMD Accelerated Parallel Processing
  Platform Vendor:                                 Advanced Micro Devices, Inc.
  Platform Extensions:                             cl_khr_icd cl_khr_d3d10_sharing cl_khr_d3d11_sharing cl_khr_dx9_media_sharing cl_amd_event_callback cl_amd_offline_devices


  Platform Name:                                   AMD Accelerated Parallel Processing
Number of devices:                                 1
  Device Type:                                     CL_DEVICE_TYPE_GPU
  Vendor ID:                                       1002h
  Board name:                                      AMD Radeon (TM) R7 M340
  Device Topology:                                 PCI[ B#1, D#0, F#0 ]
  Max compute units:                               6
  Max work items dimensions:                       3
    Max work items[0]:                             1024
    Max work items[1]:                             1024
    Max work items[2]:                             1024
  Max work group size:                             256
  Preferred vector width char:                     4
  Preferred vector width short:                    2
  Preferred vector width int:                      1
  Preferred vector width long:                     1
  Preferred vector width float:                    1
  Preferred vector width double:                   1
  Native vector width char:                        4
  Native vector width short:                       2
  Native vector width int:                         1
  Native vector width long:                        1
  Native vector width float:                       1
  Native vector width double:                      1
  Max clock frequency:                             1021Mhz
  Address bits:                                    64
  Max memory allocation:                           684510412
  Image support:                                   Yes
```

Slika 2 clinfo ispis

2.2. Dohvaćanje podataka o grafičkoj kartici

Prije nego što možemo pokrenuti bilo kakav kod na grafičkoj kartici potrebno je dohvatiti podatke o istoj. Ti podaci se spremaju u tip objekta Device u cl biblioteci. Kako bi mogli dohvatiti uređaj moramo dohvatiti trenutnu implementaciju OpenCL-a u tip objekta Platform pomoću funkcije getPlatform. Nakon toga dohvaćamo uređaj pomoću funkcije getDevices. Funkciji getDevices u prvom parametru možemo proslijediti neku od predefiniranih konstanti u cl biblioteci. U mom primjeru pomoću dodatnog argumenta pomoćnoj funkciji mogu birati između grafičke kartice (`CL_DEVICE_TYPE_GPU`), ili CPU (`CL_DEVICE_TYPE_CPU`). Slijedi kod moje pomoćne funkcije za dohvaćanje uređaja, ako želimo da kernel izvršava GPU potrebno je proslijediti 0, a za CPU 1, a povratna vrijednost funkcije je tip objekta Device. [2]

```
cl::Device getDevice(int type) {
	std::vector<cl::Platform> platforms;
	cl::Platform::get(&platforms);
	std::vector<cl::Device> devices;
	if (platforms.size() == 0) return false;
	auto platform = platforms.front();
	if (type == 0) platform.getDevices(CL_DEVICE_TYPE_GPU, &devices);
	else platform.getDevices(CL_DEVICE_TYPE_CPU, &devices);
	if (devices.size() == 0) return false;
	auto device = devices.front();
	return device;
}
```

2.3. Dohvaćanje kernela

Kod kernela se piše u datoteku ekstenzije .cl, koja označava da se radi o programu napisanom u OpenCL-u. Tu datoteku možemo dohvatiti pomoću dodatnih c++ biblioteka <string> i <ifstream>. Pomoću *string* funkcije *src* prebacujemo izvorni kod našeg kernela u *string* objekt te ga proslijeđujemo *sources* konstruktoru koja vraća tip objekta *Sources*. Prosljeđujemo ga kao tip objekta *pair*, gdje je prvi član *string*, a drugi član dužina *stringa* + 1. Prije nego što nastavimo potrebno je kreirati kontekst u kojemu ćemo izvršavati kernel. Tip objekta *Context* se kreira pomoću konstruktora *context*, te prima jedan argument – *device* koji smo kreirali. Ovo je potrebno iz razloga što OpenCL implementacija može imati više uređaja, te se kontekstu mora proslijediti s kojim ćemo uređajem raditi. [2] Nakon što imamo izvorni kod, te kontekst u varijabli, možemo napraviti tip objekta *program*, koji prima argumente *context* i *sources*. Idući korak je *compile* programa pomoću funkcije *build*. Funkcija build prima jedan argument, a to je tekst dodatnih parametara. U mome primjeru to je "-cl-std=CL1.2", što je verzija OpenCL-a, a verziju sam postavio 1.2, jer je to verzija koju moja grafička kartica podržava. Zadnji korak je kreiranje objekta tipa *Kernel*, koji vraća moja pomoćna funkcija, pomoću konstruktora koji prima parametre *program*, te naziv naše kernel funkcije. [10]

```
cl::Kernel getKernel(std::string fileName, const char functionName[], cl::Device* device,
cl::Context* context)
{
	cl_int err;
	std::ifstream exampleKernel(fileName);
	std::string src(std::istreambuf_iterator<char>(exampleKernel),
	(std::istreambuf_iterator<char>()));
	cl::Program::Sources sources(1, std::make_pair(src.c_str(), src.length() + 1));
	cl::Program program(*context, sources);
	err = program.build("-cl-std=CL1.2");
	cl::Kernel kernel(program, functionName, &err);
	return kernel;
}
```

2.4. Kernel

Sada ću opisati principe pisanja koda u OpenCL C jeziku. Funkcije u .cl datotekama započinju sa deklaracijom *__kernel*, te su tipa *void*. Potrebno je da argumenti budu tipa *global* ili *local*. Pogreška je ne definirati adresni prostor, jer se inače postavi na *private*, a to nije dozvoljeno u OpenCL jeziku. Globalnim argumentima imaju pristup sve jezgre koje se izvršavaju, no vrijeme pristupa globalnoj memoriji je veće nego lokalnoj. Lokalna memorija dostupna je samo kernelima u određenom *workgroup-u*. Slijedeći dio kod je primjer deklaracije funkcije. [10]

```
__kernel void barriers(__global int* data, __local int* localData, __global int* outData)
```

2.4.1. Workgroup

Workgroup se sastoji od *workitema*. Svaki *workitem* se može promatrati kao dretva (eng. Thread), ali je moguće da hardver pokreće više *workitema* na jednoj dretvi u svrhu paralelnog rada. Svaki *workgroup* sadrži određen broj *workitema*, čiji je maksimalni broj određen grafičkom karticom, te se može pročitati metodom *getWorkGroupInfo* u objektu *kernel*. *Workitemi* se mogu sinkronizirati uz pomoć barijera. Barijere su kao semafori, koji ne dopuštaju da se *workitemi* u svojem *workgroupu* ne nastavljaju izvršavati dokle god nisu svi došli do određene točke definirane sa barijerom. Na primjer, u mom kernelu se *workitemi* ne mogu nastaviti sa izvršavanjem petlje, dok svi nisu izvršili to ponavljanje. [3]

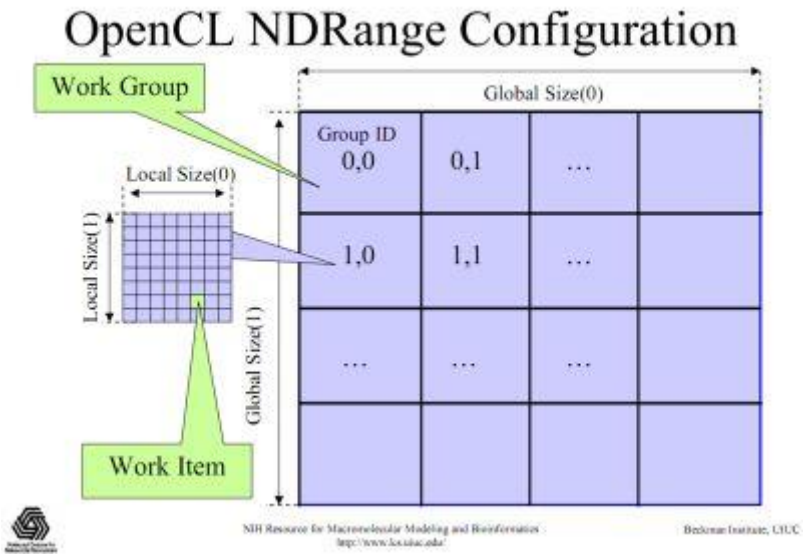
```
for(int i=(get_local_size(0)*get_local_size(1)) >> 1;i>0;i>>=1){
	if(localArrayId < i){
		localData[localArrayId] += localData[localArrayId + i];
	}
	barrier(CLK_LOCAL_MEM_FENCE);
}
```

2.4.2. Dohvaćanje globalnog i lokalnog identifikatora

Svaki kernel koji se izvršava ima svoj ID na globalnoj i lokalnoj razini. Taj ID ovisi o rasponu definiranom na pozivu kernela, koji ću obraditi kasnije, te ovisi o dimenzijama, također definiranim na pozivu. Dimenzije mogu biti jedna, dvije ili tri. Na primjer, dimenzijama se koristi kada se radi sa matricama ili trodimenzionalnim objektima, kako bi se lakše identificirali pikseli. ID možemo dohvatiti pomoću funkcije *get_global_id*, čiji je parametar ID u dimenziji koji želimo dohvatiti. Pogledajmo to na primjeru.

```
size_t arrayId = get_global_id(1)*get_global_size(0)+get_global_id(0);
```

U ovome slučaju imamo dvodimenzionalni prostor. Pošto su matrice u memoriji zapravo nizovi, pomoću opće poznatih matematičkih metoda mogu doći do pozicije u matrici. Pomoću funkcije *get_global_size(0)*, koja vraća indeks zadnjeg elementa u prvoj dimenziji, pomnoženoj sa trenutnim indeksom u drugoj dimenziji, dobivamo trenutni redak u kojem se nalazimo. Kada na to zbrojimo indeks u prvoj dimenziji dobijemo točnu poziciju elementa u matrici. Po uzoru na globalni identifikator, također postoje funkcije *get_local_id* i *get_local_size*, koje vraćaju indekse u pojedinim *workgroup-ama.* Na slici 3 je vrlo dobro ilustrirano funkcioniranje globalnih i lokalnih identifikatora. [4]



Slika 3 Identifikatori [4]

2.5. Argumenti i redovi izvršavanja

U prvom primjeru imamo vektor određene veličine pod nazivom *vec.* Ako bi željeli taj vektor proslijediti kernelu najprije moramo deklarirati varijablu tipa *Buffer* pomoću njegovog konstruktora. Prvi argument toga konstruktora je *context,* drugi argument su dozvole pristupa definirane kao konstante odvojene ili znakom. U mome prvom primjeru to su redom: *CL_MEM_READ_ONLY,* koji označava da je memorija u tom spremniku samo za čitanje, *CL_MEM_HOST_NO_ACCESS,* koji znači da glavni program nema pristup čitanja ni pisanja memorijskom prostoru, te *CL_MEM_COPY_HOST_PTR,* koji označava da će podaci definirani u trećem i četvrtom argumentu biti kopirani u memoriju. Treći argument je veličina spremnika, a četvrti pokazivač na memorijski prostor koji kopiramo u spremnik. [10]

```
cl::Buffer inBuf(context,CL_MEM_READ_ONLY | CL_MEM_HOST_NO_ACCESS | CL_MEM_COPY_HOST_PTR,
sizeof(int)*vec.size(), vec.data());
```

Također bi trebali definirati izlazni spremnik, kako bismo mogli pročitati podatke. Izlazni spremnik se definira slično kao i ulazni, sa razlikom što moramo postaviti dozvolu glavnom programu da može čitati podatke, što se može napraviti konstantom *CL_MEM_HOST_READ_ONLY,* te četvrti atribut moramo izostaviti jer još ne želimo nikakve podatke u njemu. Spremnik također može biti ulazno/izazni, stoga moramo postaviti zastavicu *CL_MEM_READ_WRITE,* da kernel može čitati i pisati podatke u njega.

Nakon što smo definirali memorijske spremnike moramo ih postaviti kao argumente kernela metodom *setArg,* čiji je prvi argument indeks argumenta u kernelu (0 za prvi argument u kernelu, 1 za drugi itd.), te drugi argument varijabla memorijskog spremnika. [10]

```
kernel.setArg(0, inBuf);
```

Kako bi pokrenuli kernel na grafičkoj kartici, moramo napraviti objekt tipa *CommandQueue,* koji kao argumente prima kontekst i uređaj koji smo dohvatili, te služi kao objekt reda čekanja izvršavanja. Također kao treći opcionalni argument prima konstantu *CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE,* s kojom možemo odrediti da se pojedine grupe kernela ne izvršavaju redom kako ih pozivamo, nego po nasumičnom redu. Na primjer ako stavimo u red čekanja prvo kernel A, pa kernel B, nema garancije da će A završiti prije B. [10]

U red čekanja naš kernel možemo staviti pomoću metode objekta *CommandQueue, enqueueNDRangeKernel*. Prvi argument toj funkciji je tipa *kernel*, drugi argument tipa *NDRange* iz cl biblioteke, te služi za definiranje odstupanja od raspona identifikatora. Objekti tipa *NDRange* primaju jedan, dva ili tri argumenta, a radi se o veličinama dimenzija s kojima želimo da kernel raspolaže. Na primjer, ako želimo da nam kernel bude dvodimenzionalan, te je svaka dimenzija veličine dva, kao treći argument poslat ćemo `cl::NDRange(2, 2)`. Četvrti argument je lokalna dimenzija, ili dimenzija *workgroup-a,* koji je također *NDRange.* Ako se želimo osigurati da je cijeli red čekanja izvršen to možemo napraviti sa funkcijom *finish.*

Čitanje iz memorijskom spremnika i zapisivanje u varijablu možemo izvršiti metodom *enqueueReadBuffer,* čiji je prvi argument memorijski spremnik koji smatramo izlaznim, drugi argument je konstanta, čije vrijednosti mogu biti *CL_TRUE* ili *CL_FALSE. CL_TRUE* znači da će glavni program nastaviti s radom tek kada su vrijednosti kopirane u varijablu. To nam može biti pogodno ako želimo čitati iz varijable odmah nakon izvršavanja metode. Dalje su argumenti odstupanje, veličina koju kopiramo, te pokazivač na varijablu u koju kopiramo. Cijeli proces stavljanja u red čekanja, te čitanje iz spremnika možete vidjeti u sljedećem kodu. [10]

```
cl::CommandQueue queue(context, device);
queue.enqueueNDRangeKernel(kernel, cl::NullRange, cl::NDRange(vec.size()));
cl::finish();
queue.enqueueReadBuffer(outBuf, CL_FALSE, 0, sizeof(int) * vec.size(), vec.data());
```

3. Dekompozicija problema

Prije nego što smo počeli razvijati paralelno rješenje nekog problema, moramo prvo provjeriti da li algoritam može biti paraleliziran. Paralelni algoritam je onaj algoritam koji se može izvršiti odjednom na više različitih procesnih uređaja, te na kraju spojiti rezultat i dobiti rješenje. Jedan od prvih koraka je da podijelimo zadatak u neovisne dijelove, koji se mogu rasporediti po različitim procesima, što se naziva dekompozicija. Dekompozicija može biti više vrsta, te se veže na to kako su nezavisni procesi definirani. Neke od vrsta dekompozicije su: dekompozicija po zadacima, dekompozicija po domeni te implicitna dekompozicija. [8]

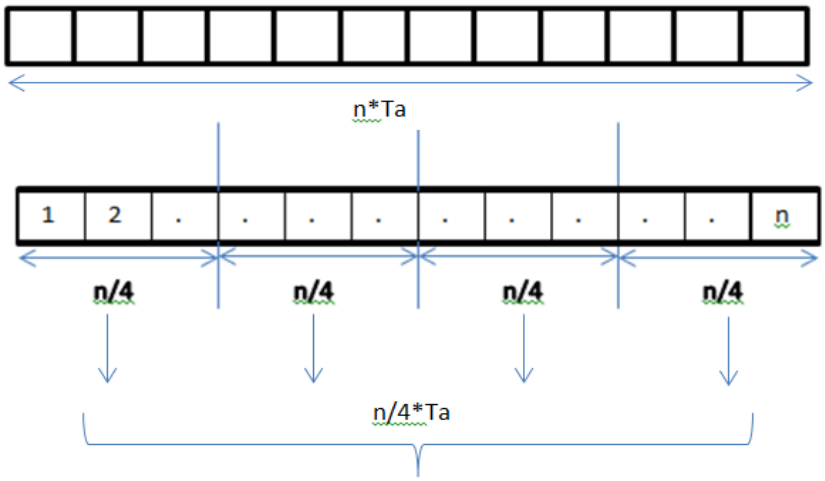
3.1. Dekompozicija po zadacima

Dekompozicija po zadacima je tip dekompozicije koji se fokusira na raspodjelu različitih zadataka na više različitih procesora. Razlika između dekompozicije po domeni i dekompozicije po zadacima je ta da dekompozicija po zadacima radi više različitih zadataka na istim podacima. Česti tip dekompozicije po zadacima je „pipelining“, što podrazumijeva vođenje skupa podataka kroz niz različitih zadataka, gdje se svaki zadatak može izvršiti neovisno od posljednjeg. Također bi se procesi trebali isključivati, npr. semaforima, monitorima ili nekim drugim načinima za međusobno isključivanje kako ne bi pristupali istom podatku odjednom.

Od mojih primjera niti jedan algoritam nije paralelan po zadacima, jer nema slučaja gdje su zadaci koji su različiti neovisni jedan o drugome. [6]

3.2. Dekompozicija po domeni

Dekompozicija po domeni ili podacima je dekompozicija koja dijeli ulazni podatak, npr. niz podataka, na više dijelova, te nakon toga svaki paralelni zadatak obrađuje svoj dio podataka. Za razliku od dekompozicije po zadacima, u dekompoziciji po domeni više istih zadataka obrađuje različite podatke. Primjer dekompozicije po podacima bi bila paralelna redukcija, koji sam primjer i implementirao, gdje sam podijelio set podataka te ih obradio paralelno na grafičkom procesoru, te ih na kraju istom metodom zbrojio. [6]



Slika 4 Dekompozicija po podacima [6]

3.3. Postupak paralelizacije

Tablica 1 Postupak paralelizacije [6]

Tip	Opis
Dekompozicija	Problem je podijeljen u najmanje jedinice koje se mogu izvršavati paralelno.
Dodjeljivanje	Zadaci su pridruženi procesima.
Orkestracija	Određivanje pristupa podacima, komunikacije i sinkronizacije između procesa.
Mapiranje	Procesi su pridruženi procesorima.

4. Primjeri GP GPU

4.1. Množenje elemenata niza sa brojem

Ovo je poprilično jednostavan algoritam, te tipičan primjer dekompozicije po podacima, koji paralelno množi svaki element niza sa određenim brojem. Iz tog razloga brzina je veća, odnosno vrijeme izvršavanja kraće, nego kod CPU zbog toga što CPU linearno prolazi kroz svaki element te ga zbraja, a GPU paralelno obavi tu operaciju u jednom prolazu. Kernel kod je vrlo jednostavan: N puta se pokrene te se za svaki element dohvati pozicija u listi te se obavi operacija i zapiše u izlazni spremnik.

```
__kernel void arrayMultiply(__global int* data, __global int* outData)
{
	size_t tid = get_global_id(0);
	outData[tid] = data[tid] * 2;
}
```

Tablica 2 Množenje Vektora Vrijeme

Veličina vektora (N)	GPU (s)	CPU (s)
10000	0.031	0.005
100000	0.03	0.05
1000000	0.03	0.471
10000000	0.038	4.729

Iz tablice 2 jasno vidimo kako vrijeme potrebno za množenje vektora na grafičkom raste jako sporo, a vrijeme na CPU raste linearno sa veličinom vektora. Također možemo vidjeti, što možemo vidjeti i u kasnijim primjerima, je da vrijeme na GPU nikada ne pada ispod 30 milisekundi, jer je to vrijeme potrebno za prvo pokretanje kernela.

4.2. Paralelna redukcija

U ovome primjeru sam pomoću paralelne redukcije paralelno zbrojio sve elemente dvodimenzionalne matrice. To sam napravio na način da sam podijelio ulaznu matricu u *workgrupe,* te primijenio algoritam na njima, te dobio izlaznu matricu. Na dobivenoj izlaznoj matrici sam opet primijenio istu metodu, dok nisam dobio matricu koja odgovara veličini jedne *workgrupe.* Tada sam paralelnom redukcijom zbrojio rezultat, te dobio krajnji rezultat. Za ove operacije red čekanja se morao izvršavati *in-order.* Algoritam paralelne redukcije za pojedini *workitem* izgleda ovako:

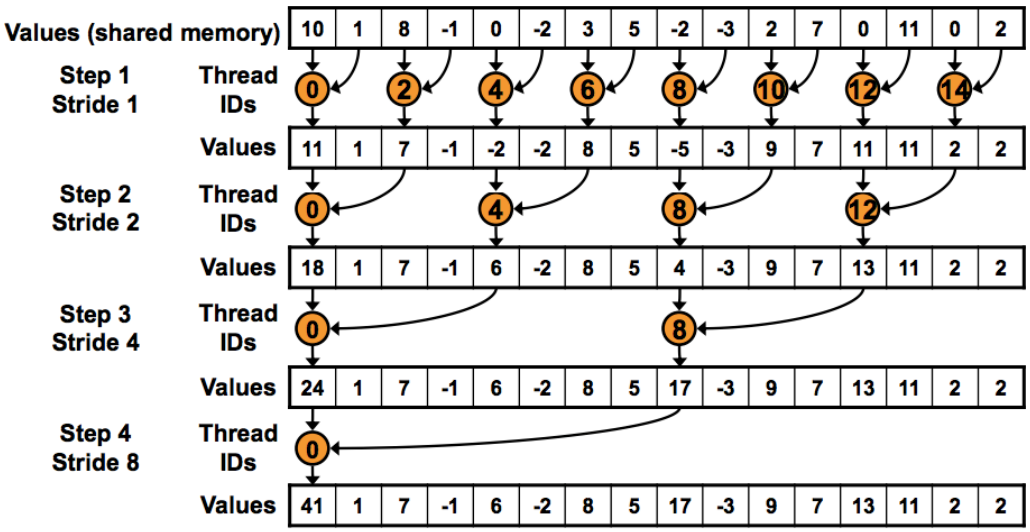
```
__kernel void barriers(__global int* data, __local int* localData, __global int*
outData)
{
	size_t arrayId = get_global_id(1)*get_global_size(0)+get_global_id(0);
	size_t localArrayId = get_local_id(1)*get_local_size(0)+get_local_id(0);

	localData[localArrayId] = data[arrayId];
	barrier(CLK_LOCAL_MEM_FENCE);

	for(int i=(get_local_size(0)*get_local_size(1)) >> 1;i>0;i>>=1){
		if(localArrayId < i){
			localData[localArrayId] += localData[localArrayId + i];
		}
		barrier(CLK_LOCAL_MEM_FENCE);
	}
	if(localArrayId == 0){

	outData[get_group_id(1)*(get_global_size(0)/get_local_size(0))+get_group_
id(0)] = localData[0];
	}
}
```

Na početku je potrebno pridružiti globalnu i lokalnu poziciju u varijablu, te postaviti lokalnu matricu, jer je brže pristupati njoj nego globalnoj memoriji. Sa postavljanjem lokalne matrice svi *workitemi* moraju biti gotovi, za što se brine funkcija *barrier*. Idući korak je obrada elemenata u matrici. Bitovno pomicanje u desno odnosno dijeljenje sa 2, služi da bi krenuli procesirat elemente matrice od polovice prema dolje. Svaki element se zbraja sa njegovom nasuprotom pozicijom na drugoj polovici. Ta petlja se izvodi dok rezultat nije završio na nultoj poziciji lokalne matrice, na principu kao na slici 6.



Slika 5 Paralelna redukcija [7]

Zadnje izvršenje kernela je malo drugačije jer nema smisla koristiti lokalnu memoriju, jer je globalna dimenzija jednaka dimenziji *workgrupe.*

```
__kernel void withoutBarriers(__global int* data, __global int* outData)
{
	size_t arrayId = get_global_id(1)*get_global_size(0)+get_global_id(0);

	for(int i=(get_global_size(0)*get_global_size(1)) >> 1;i>0;i>>=1){
		if(arrayId < i){
			data[arrayId] += data[arrayId + i];
		}
		barrier(CLK_GLOBAL_MEM_FENCE);
	}
	if(arrayId == 0){
		outData[0] = data[0];
	}
}
```

Što se tiče brzine, vidimo isti slučaj kao i kod množenja vektora, povećavanje opsega podataka ne uzrokuje povećanje vremena izvršavanja na GPU, dok na CPU raste. Na manjim podacima je CPU brži kao što vidimo na matrici 512x512, no sa povećanjem dimenzija GPU uzima prednost.

Tablica 3 Paralelna Redukcija Vrijeme

Veličina matrice (NxM)	**GPU (s)**	**CPU (s)**
512x512	0.033	0.001
1024x1024	0.034	0.003
4096x4096	0.036	0.064
8192x8192	0.033	0.263

4.3. Odd-Even Sort

Odd-Even sort je relativno jednostavan algoritam za sortiranje, sličan bubble sortu, jer su obje metode uspoređujući sortovi. Funkcionira na način da uspoređuje parove u listi, u jednom koraku počevši od parnog elementa, a u drugom koraku od neparnog elementa. Budući da su parovi potpuno neovisni jedan o drugome, moguće je primijeniti paralelizam. Svaki kernel će uspoređivati jedan par u listi, te zamijeniti ako su u krivom redoslijedu. Prvo će se izvršiti parni, pa neparni prolaz kroz listu. U glavnome programu će petlja stavljati kernele u redove čekanja, u *in-order* redoslijedu. Petlja u glavnome programu izgledati će ovako:

```
	for (int i = 0; i < elements; i++) {
		if (i % 2) {
			err = queue.enqueueNDRangeKernel(kernelEven, cl::NullRange,
cl::NDRange(elements / 2));
		}
		else {
			err = queue.enqueueNDRangeKernel(kernelOdd, cl::NullRange,
cl::NDRange(elements / 2));

		}
	}
```

Raspon je n/2 jer se kernel pokreće za polovicu elemenata u listi, a ostalu polovicu uspoređuje. Kod kernela izgleda ovako:

```
__kernel void even(__global int* data)
{
	size_t id = get_global_id(0)*2;
	int temp;
	if(get_global_id(0)<=(get_global_size(0)-1)){
		if(data[id] > data[id+1]){
			temp = data[id];
			data[id] = data[id+1];
			data[id+1] = temp;
		}
	}
}
```

Za neparne elemente, izgleda relativno isto, jedina razlika je u dohvaćanju ID-a, koji za neparne elemente iznosi `get_global_id(0)*2+1.`

Ovaj algoritam usporedio sam sa quicksortom i bubblesortom. U ovom slučaju vrijeme izvršavanja na GPU raste, što se može zaključiti iz činjenice što postoji petlja u glavnom programu koja pokreće izvršavanje kernela, te je ovisna o veličini niza. Iz tog razloga je i Quicksort brži od mog algoritma. Također možemo vidjeti u tablici primjer O(n^2) algoritma, gdje sa 10 puta povećanjem veličine niza, vrijeme izvršavanja naraste otprilike 100 puta.

Tablica 4 Odd-Even Sort Usporedba

Veličina niza (N)	**Odd-Even (s)**	**Quick (s)**	**Bubble (s)**
1000	0.032	0.001	0.01
10000	0.042	0.004	1.147
100000	0.144	0.052	110.415

5. Zaključak

Paraleliziranje algoritama koji se vrlo lako daju paralelizirati pokazalo se vrlo učinkovito. Odd-Even sort algoritam, koji na sekvencijalnim procesorima ima složenost O(n^2), ubrzao se skoro na razinu quicksorta, sa složenošću O(logn). Takvih algoritama ima mnogo, te mislim da će u budućnosti biti sve veći prijelaz na paralelno računanje, te će OpenCL i CUDA ubrzo imati konkurenciju, no i dalje smo u početnim razvojima paralelnog računanja, gdje se jezgre procesora mogu prebrojiti rukom. C

Također moramo uzeti u obzir i pojavljivanje tzv. „blockchaina“, čija je osnovna sastavnica računanje matematičkih problema kako bi se provjerila valjanost transakcije te nagradili „rudari“. Nedugo nakon nastanka „blockchaina“, otkriveno je da grafičke kartice puno brže rudare nego CPU. Stoga će potreba za brzim grafičkim karticama rasti, a samim time će rasti i potreba za dobrim API-em, koji će omogućiti što veću efikasnost tih grafičkih kartica. U tom pogledu bi mogao prednost uzeti OpenCL, jer sam u ovom vremenu koliko sam radio sa OpenCL-om, shvatio da ima vrlo široku programsku podršku, te su podržane starije grafičke kartice, što nije slučaj kod nVidie. Također je velika prednost vrlo lako prebacivanje između CPU i GPU, isti programski kod radi za oba uređaja, te iz tih razloga mislim da je budućnost OpenCL-a vrlo perspektivna.

6. Literatura

[1] Smistad E (2018) Getting started with OpenCL and GPU Computing. Preuzeto 18.9.2018. s https://www.eriksmistad.no/getting-started-with-opencl-and-gpu-computing.

[2] What do OpenCL contexts mean? Why do they make sense? (2018) Preuzeto 18.9.2018. s https://stackoverflow.com/questions/38587810/what-do-opencl-contexts-mean-why-do-they-make-sense.

[3] OpenCL: work group concept (2018) Preuzeto 18.9.2018. s https://stackoverflow.com/questions/26804153/opencl-work-group-concept.

[4] [TEST] GPU Computing – GeForce and Radeon OpenCL Test (Part 2) (2010) Preuzeto 18.9.2018. s https://www.geeks3d.com/20100115/test-gpu-computing-geforce-and-radeon-opencl-test-part-2/.

[5] OpenCL: work group concept (2018) Preuzeto 18.9.2018. s https://stackoverflow.com/questions/26804153/opencl-work-group-concept.

[6] Data Parellelism (2018) Preuzeto 18.9.2018. s https://en.wikipedia.org/wiki/Data_parallelism

[7] GPUExample (2017) Preuzeto 18.9.2018. s https://github.com/mateuszbuda/GPUExample

[8] Donald F (2016) Introduction to GPU Parallel Programming. Preuzeto 18.9.2018. s https://computing.llnl.gov/tutorials/dataheroes/GPUParallelProgramming.pdf

[9] General-purpose computing on graphics processing units (2018) Preuzeto 18.9.2018. s https://en.wikipedia.org/wiki/General-purpose_computing_on_graphics_processing_units

[10] OpenCL Documentation (2018) Preuzeto 18.9.2018. s https://www.khronos.org/registry/OpenCL/sdk/1.0/docs/man/

7. Popis tablica

8. Popis slika

9. Prilozi

Izvorni kod je moguće vidjeti na: https://github.com/evituri/zavrsniRad