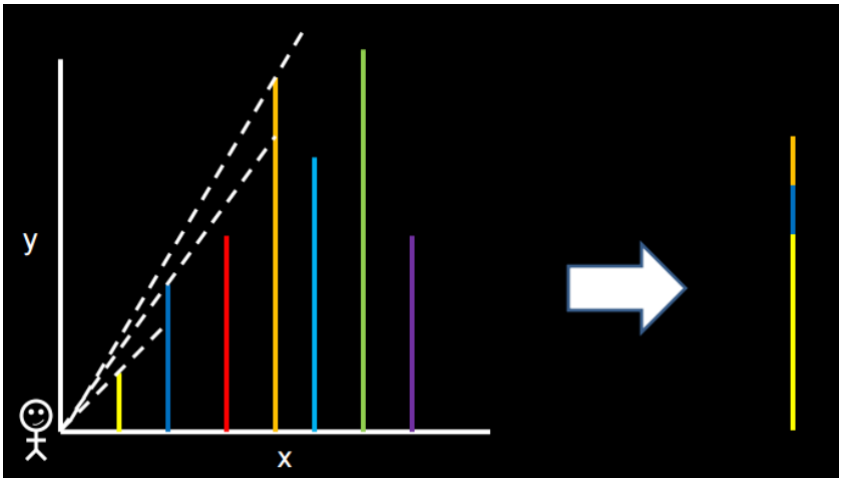
Seminar: Paralelizacija določanja vidnih odsekov črt

Primož Lavrič (63130133)

14. marec 2016

1 Uvod

Cilj seminarja je bil zasnovati kar se da optimalni paralelni algoritem za reševanje problema vidnih črt na izbranih arhitekturah. Gre za geometrijski problem, pri katerem določamo vidni del vertikalnih črt iz opazovalčeve perspektive (v našem primeru se opazovalec lahko nahaja le v izhodišču koordinatnega sistema, zaradi poenostavitve problema). Za vse črte velja, da so neprekrivajoče in se nahajajo v prvem kvadrantu koordinatnega sistema, vsaka izmed črt ima eno končno točko na **x** osi koordinatnega sistema in je pravokotna na **x** os. Zaradi poenostavitve problema lahko vidni del posamezne črte izračunamo v odvisnosti od predhodnih s pomočjo kotne funkcije tangens. Velja $\tan\alpha = \frac{nasprotna\_kateta\ (Y)}{prilezna\_kateta\ (X)}$ torej za vsako črto, ki je vidna velja: $\frac{Y_{i-s}}{X_{i-s}} < \frac{Y_i}{X_i}$ za $1 <= s <= i$. Vidni odsek črte i pa je definiran z $vis_i = Y_i - \frac{X_{i-1}}{Y_{i-1}} * X_i$



Slika 1: Shema problema.

2 Izbrane arhitekture in pristopi

2.1 Rešitev z uporabo PThread

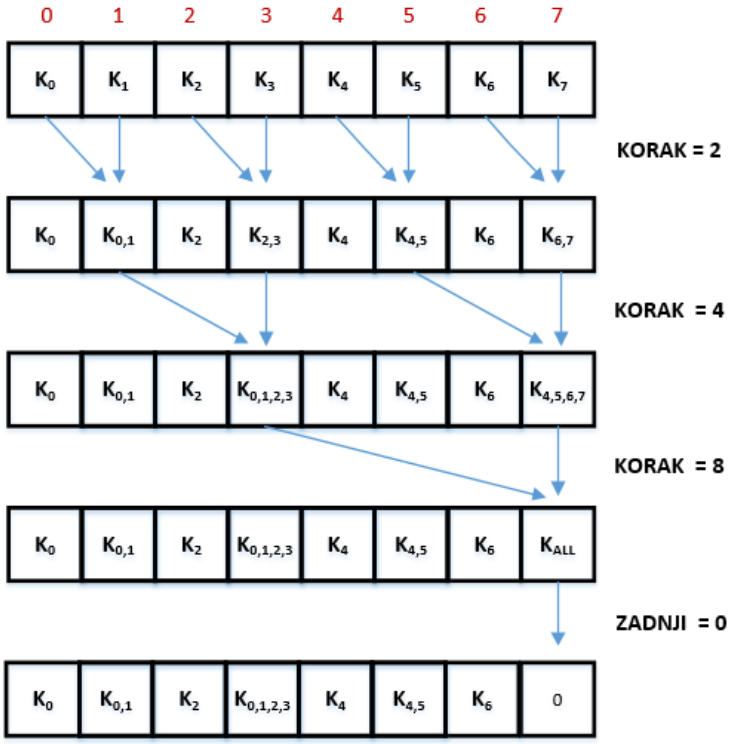
Sama implementacija s pomočjo PThread je nastala kot prototip GPU implementacije. Cilj implementacije je tem bolj optimizirati računanje maksimalnega elementa predpone (z njim določimo, če je zadnja črta, ki ustreza zadnjemu elementu predpone vidna oz. kakšen delež črte je viden). Pri snovanju učinkovitega algoritma za računanje predpon sem se zgledoval po Mark Harrisovem članku, ki opisuje učinkovito implementacijo "prefix sum"algoritma na arhitekturi Cuda[1].
Glavna ideja te implementacije je zgraditi uravnoteženo binarno drevo čez vhodne podatke. Čez vpeto drevo nato izračunamo maksimalni element na poti do vsakega izmed vozlišč iz listov do korena in nazaj. Ta dva koraka imenujemo "up-sweep"2 in "down-sweep"3. Ker je binarno drevo uravnoteženo, ima tako vpeto drevo $log(n)$ nivojev, na vsakem izmed nivojev pa 2^d vozlišč. Torej za izvršitev enega koraka ("up-sweep", "down-sweep") potrebujemo O(n-1) operacij (za vsako vozlišče enkrat izračunamo maksimalno vrendost). Za izračun maksimalnih predpon torej potrebujemo $2 * O(n-1)$ operacij.
Glede na zgornji opis je vredno omeniti, da se pri dani implementaciji ne uporablja dejansko binarno drevo, ampak to služi zgolj kot koncept, ki določa delo niti v posameznih korakih (izračuni na posameznem nivoju konceptualnega drevesa se lahko izvršijo paralelno).
Celotno implementacijo sestavljajo 4 koraki:

1. Realociranje vektorja podanih višin na velikost $2^{ceil(log2(N))}$. Implementacija namreč zahteva, da je dvojiški logaritem velikosti podanega vektorja celo število. Čez vektor namreč želimo vpeti polno uravnoteženo dvojiško drevo.

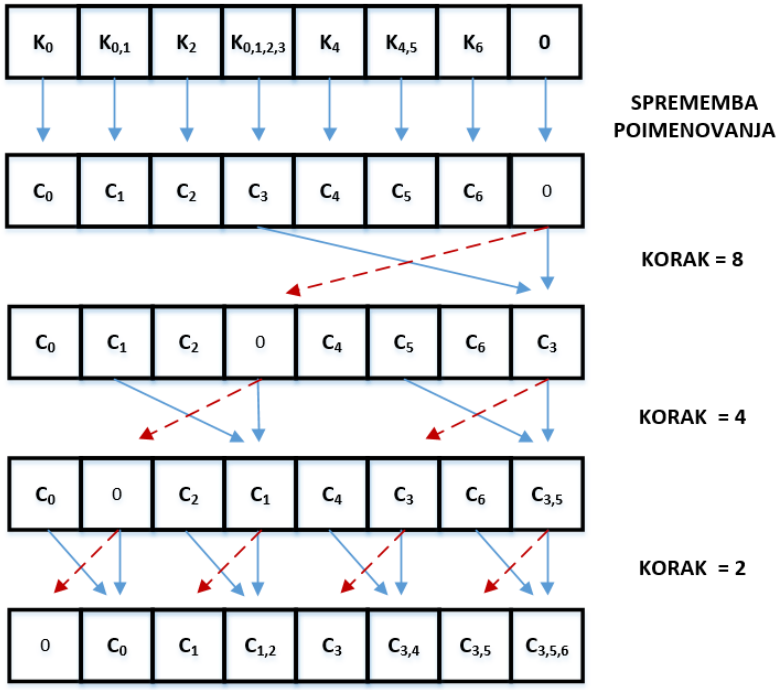
2. Izračun razmerij višin črt in njihove oddaljenosti od opazovalca.

3. Nad izračunanim vektorjem razmerij $\vec{r}$ nato izvedemo zgoraj opisan algoritem ("prefix max scan"). Ta algoritem nam vrne vektor $\vec{m}$ za katerega velja $\vec{m_i} = max(\vec{r_0}, \vec{r_1}, ..., \vec{r_{i-1}})$ za $i > 0$ in $\vec{m_0} = 0$.

4. V zadnjem koraku se izračunajo vidne višine s pomočjo vektorja $\vec{m}$, izračunanega v 3. koraku, vektorja višin $\vec{y}$ in vektorja oddaljenosti črt od opazovalca $\vec{x}$. Vektor vidnih višin je torej definiran kot $vis_i = \vec{y} - \vec{m} * \vec{x}$. Tu je vredno omeniti, da se ta korak izvede v zadnjem koraku down-sweepa (zaradi optimizacije dostopa do pomnilnika).



Slika 2: Primer "up-sweep"koraka.



Slika 3: Primer "down-sweep"koraka.

2.2 Rešitev z uporabo OpenCL

Implemetacija s pomočjo OpenCL se nekoliko razlikuje od implementacije s pomočjo pthread. Glavni razlog za razliko je sama arhitektura (ščepec se namreč izvaja na grafični procesni enoti GPE). Sam ščepec izvajajo delovne skupine, ki vsebujejo **N** niti. Tu se pojavi problem sinhronizacje, saj je na GPE mogoče sinhronizirati samo niti, ki se nahajajo znotraj ene delovne skupine, samih delovnih skupin pa ni mogoče sinhronizirati. Ker zgoraj opisana implementacija zahteva, da se nivoji vpetega drevesa poračunajo zaporedno, ni mogoče izračunati maksimalnih predpon vektorja večjega od maksimalne velikosti delovne skupine. Dani problem sem rešil tako, da probleme večje od velikosti delovne skupine razdelim na več podproblemov velikosti ene delovne skupine. Primer: za problem velikosti $4 * W$, kjer je W velikost delovne skupine, razdelim glavni problem na 4 podprobleme, ki jih rešujejo različne delovne skupine.
Tu nastane naslednji problem, maksimalne predpone so namreč poračunane samo za posamezne podprobleme, zato je potrebno maksimalni element posameznega podproblema upoštevati v vseh naslednjih podproblemih, da bo globalni problem pravilno rešen. Tega sem se lotil tako, da shranim začasne maksimalne elemente posameznih podproblemov in nato v naslednjem ščepcu izračunam maksimalne predpone teh elementov. To rekurzivno ponavljam, dokler ni število maksimalnih elementov manjše od velikosti ene delovne skupine. Tedaj se rekurzivno vračam čez vse maksimalne elemente, pri katerih upoštevam maksimalne elemente njihovih predhodnih podproblemov (enostaven primer je prikazan na sliki 4). Po končanem propagiranju predhodnih podproblemov lahko sedaj poračunamo vidne višine posameznih črt.
Celotno implementacijo sestavlja 5 korakov (in 4 ščepci):

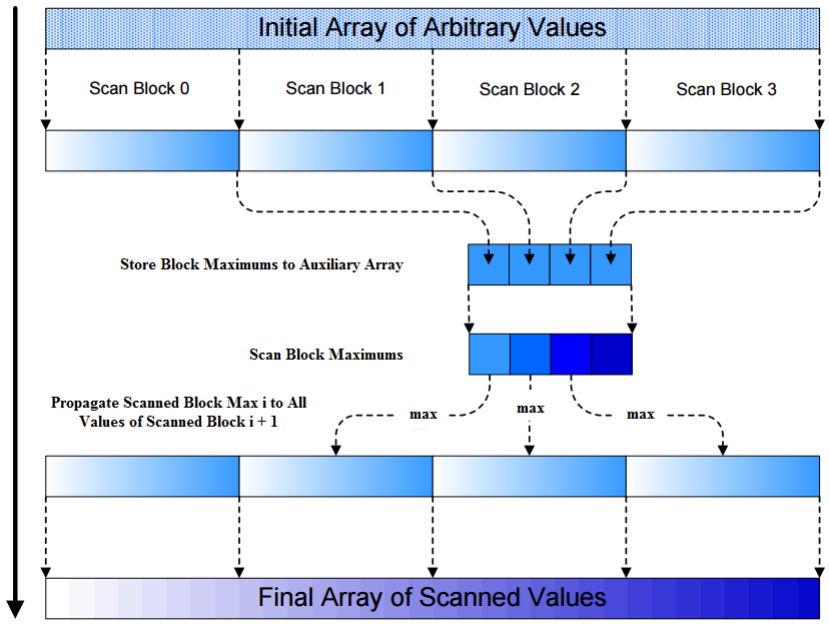
1. Razčlenitev problema, računanje globine rekurzije in inicializacija potrebnih ščepcev.

2. Izvršitev prvega ščepca, ki izračuna razmerja, lokalno izvrši izračun maksimalnih predpon ter v vektor maksimalnih vrednosti podproblemov zapiše maksimalno vrednost poračunanega podproblema.

3. Izvršitev drugega ščepca (glede na velikost problema se lahko izvrši večkrat ali nikoli), vhod tega ščepca je vektor maksimalnih elementov predhodnega ščepca. Njegova naloga je, da nad tem vektorjem izvrši izračun maksimalnih predpon in v primeru, da je ščepec izvršilo več skupin, vsaka izmed skupin zapiše svojo maksimalno vrednost v naslednji vektor maksimalnih vrednosti podproblemov.

4. Izvršitev tretjega ščepca (glede na velikost problema se lahko izvrši večkrat ali nikoli), vhod tega ščepca je vektor maksimalnih predpon podproblemov. Naloga tega ščepca je, da pravilno propagira te predpone.

5. Izvršitev četrtega ščepca, naloga katerega je, da propagira zadnji vektor maksimalnih predpon podproblemov in nato izračuna dejanske vidne višine črt.



Slika 4: Primer maksimalnih predpon na problemu večjem od velikosti delovne skupine.

2.3 Rešitev z uporabo OpenMPI

Implementacije algoritma s pomočjo OpenMPI sem se lotil nekoliko drugače. Zgoraj opisani pristop namreč zahteva veliko komunikacije, katero želimo pr uporabi OpenMPI minimizirati (zaradi veliko počasnejšega komunicranja med procesi čez omrežje). Glavna ideja implementacije je, da vsak izmed procesov dobi približno enak kos podatkov, katerega najprej sam sprocesira (izračuna kvociente in maksimalne predpone kvocientov). Ker je vsak kos podatkov odvisen od vseh predhodnih kosov, je potrebno upoštevati maksimalne kvociente vseh predhodnikov. Ta problem rešim tako, da vsak proces pošlje svoj maksimalni kvocient vsem naslednikom, ki nato poiščejo maksimalnega od vseh prejetih kvocientov. Ta kvocient nato propagirajo čez svoje kvociente (dokler ne naletijo na večji kvocient od prejetega). Vsak izmed procesov izračuna vidne višine, katere posreduje glavnemu procesu, ki jih združi v končno rešitev.

Celotno implementacijo sestavlja 7 korakov:

1. Izračun razdelitve dela med procesi (izračun velikosti kosov, ki jih dobijo posamezni procesi).
2. Razpošiljanje kosov med vse delovne procese.
3. Izračun kvocientov in računanje maksimalnih predpon.
4. Posredovanje lokalnega maksimuma vsem naslednikom in sprejemanje lokalnih maksimumov vseh predhodnikov (izračun maksimalnega lokalnega maksimuma predhodnikov).
5. Propagiranje izračunanega maksimuma čez lokalne maksimalne predpone (dokler ne naleti do predpone večje od propagiranega maksimuma).

6. Vsak izmed procesov izračuna višine s pomočjo izračunanih maksimalnih predpon.

7. Združevanje rezultatov v glavnem procesu.

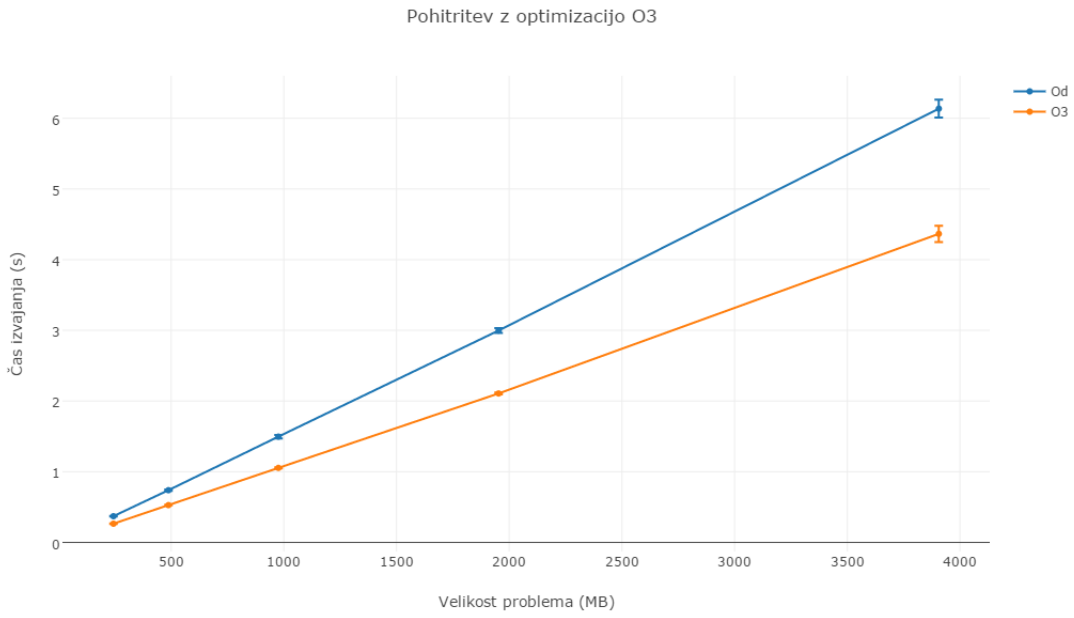
3 Rezultati

3.1 Sekvenčni algoritem

Sekvenčno implementacijo sem stestiral z različnimi optimizacijami prevajalnika. Najbolj zanimva je O3 optimizacija, saj je sam faktor pohitritve 1.4, kar je primerljivo s paralelnimi implementacijami.

Velikost problema	Optimizacija	Čas izvajanja (s)	STD (s)
244 MB	Od	0.3712	0.0036
	O3	0.2652	0.0032
488 MB	Od	0.7391	0.0086
	O3	0.5282	0.0041
976 MB	Od	1.4967	0.0248
	O3	1.0549	0.0058
1953 MB	Od	2.9973	0.0326
	O3	2.1071	0.0122
3906 MB	Od	6.1352	0.1273
	O3	4.3646	0.1152

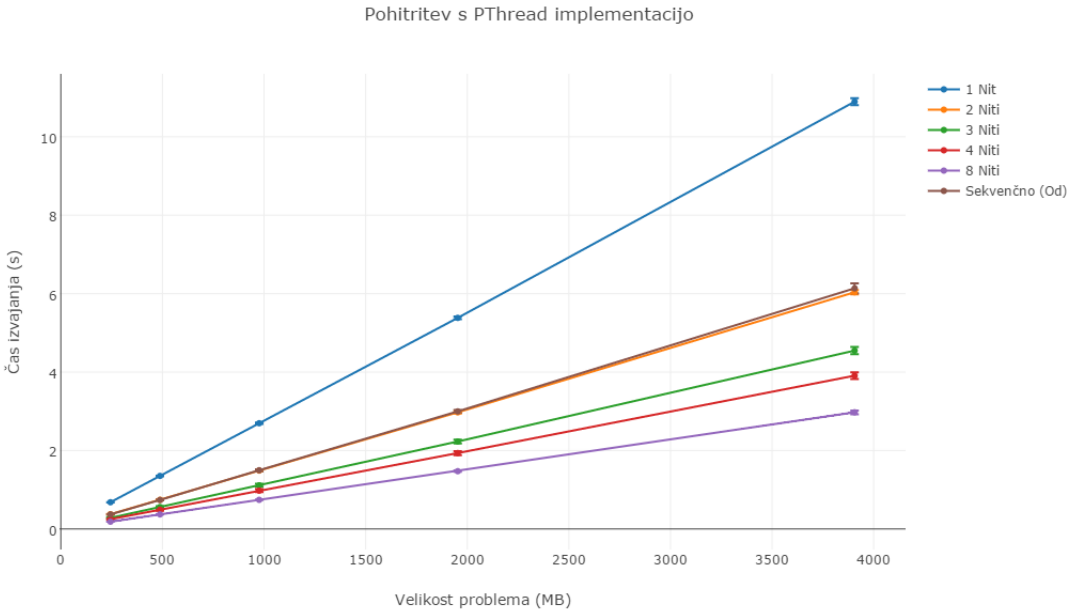
Tabela 1: Časi izvajanja sekvenčnega algoritma brez in z optimizacijo O3



Slika 5: Pohitritev sekvenčnega algoritma z uporabo O3 optimizacije.

3.2 PThread

Hitrost implementacije z uporabo PThread je na spodnji sliki6 primerjanja z ne optimirano implementacijo sekvečnega algoritma. Implementacija postane učinkovita šele pri uporabi 3 ali več niti in doseže faktor pohitritve okoli 2 na večjih problemih.



Slika 6: Pohitritev sekvenčnega algoritma z uporabo PThread implementacije.

Velikost problema	Število niti	Čas izvajanja (s)	STD (s)	Faktor pohitritve
255 MB	1	0.6807	0.0021	0.55
	2	0.3751	0.0062	0.99
	3	0.2823	0.0181	1.31
	4	0.2487	0.0150	1.49
	8	0.1847	0.0027	2.01
488 MB	1	1.3520	0.0075	0.55
	2	0.7434	0.0087	0.99
	3	0.5582	0.0237	1.32
	4	0.4907	0.0238	1.51
	8	0.3711	0.0044	1.99
976 MB	1	2.6954	0.0170	0.56
	2	1.4887	0.0180	1.01
	3	1.1120	0.0378	1.35
	4	0.9760	0.0366	1.53
	8	0.7419	0.0083	2.02
1953 MB	1	5.3816	0.0347	0.56
	2	2.9733	0.0298	1.01
	3	2.2292	0.0458	1.34
	4	1.9343	0.0523	1.55
	8	1.4729	0.0173	2.03
3906 MB	1	10.8940	0.0903	0.56
	2	6.0351	0.0554	1.02
	3	4.5477	0.0910	1.35
	4	3.9081	0.0844	1.57
	8	2.9739	0.0481	2.06

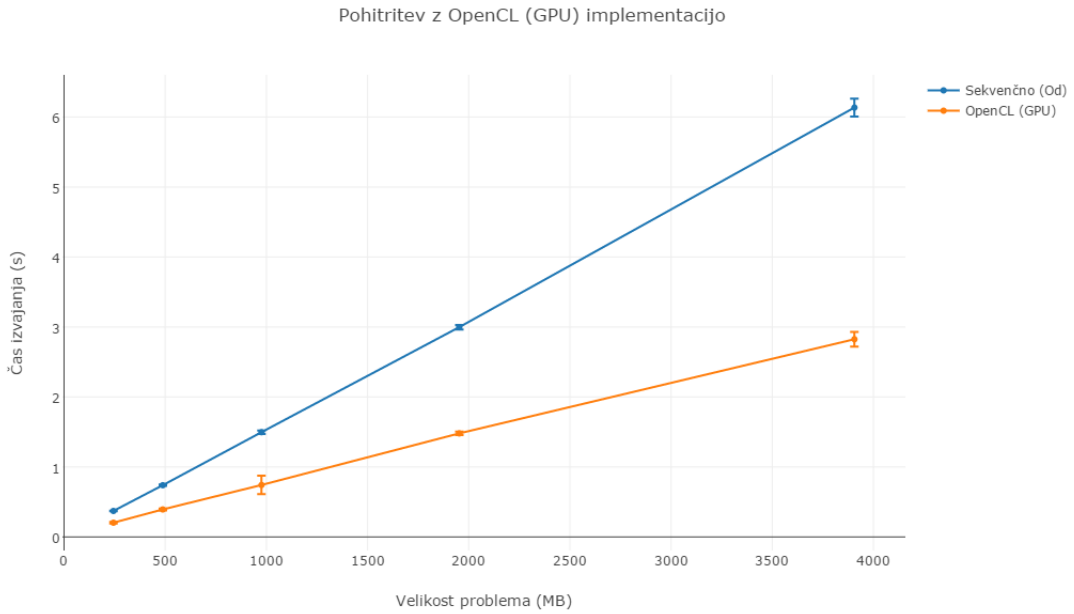
Tabela 2: Časi izvajanja PThread implementacije in faktor pohitritve

3.3 OpenCL (GPU)

Hitrost implementacije z uporabp OpenCL (GPU) je na spodnji sliki7 primerjana z ne optimirano implementacijo sekvenčnega algoritma. Sama implementacija dosega podobne pohitritve kot implementacija s pomočjo PThread z uporabo 8 niti. Implementacijo bi bilo mogoče še izbolšati z optimiranjem ščepca za računanje maksimalnih predpon (preprečevanje pomnilniških kolizij).

Velikost problema	Čas izvajanja (s)	STD (s)	Faktor pohitritve)
244 MB	0.2033	0.0097	1.83
488 MB	0.3929	0.0122	1.88
976 MB	0.7423	0.1323	2.02
1953 MB	1.4794	0.0251	2.03
3906 MB	2.8249	0.1055	2.17

Tabela 3: Časi izvajanja OpenCL (GPU) implementacije ter faktor pohitritve



Slika 7: Pohitritev sekvenčnega algoritma z uporabo OpenCL GPU implementacije.

3.4 OpenMPI (Gruča)

Implementacija s pomočjo OpenMPI se ni izkazala za najboljšo. Sama hitrost izračuna se je namreč zmanjšala v primerjavi z sekvenčnim algoritmom (pri tem velja omenita, da imata testna sistema različne performanse). Časi izvajanja so prikazani v spodnji tabeli 4

4 Testni sistem

Sekvenčna, PThread in OpenCL implementacije so bile testirane na sistemu:

- Procesor: Intel i7 6700K 4.2 GHz
- Ram: 16GB
- GPU: Nvidia gtx 980ti

OpenMPI implementacija pa je bila testirana na IJS-jevi gruči (podana je specifikacija računalnikov):

- Procesor: Intel Xeon E5520 2.27 GHz
- Ram: 8GB

Velikost problema	Št. procesov	Čas izvajanja (s)	STD (s)	Faktor pohitritve
244 MB	4	2.6416	0.0043	/
	8	2.5953	0.0819	/
	16	2.8431	0.1636	/
	32	3.2984	0.1675	/
488 MB	4	5.2897	0.0109	/
	8	5.1806	0.0750	/
	16	5.1641	0.7982	/
	32	5.7784	0.2177	/
976 MB	4	10.5799	0.0155	/
	8	10.3392	0.0882	/
	16	10.3010	0.1194	/
	32	10.5566	0.2565	/
1953 MB	4	21.1703	0.0325	/
	8	20.6369	0.1676	/
	16	20.3945	0.1292	/
	32	20.5796	0.2002	/

Tabela 4: časi izvajanja implementacije z uporabo OpenMPI na IJS gruči

Literatura

[1] Harris, Mark, Shubhabrata Sengupta, and John D. Owens. "Parallel prefix sum (scan) with CUDA."GPU gems 3.39 (2007): 851-876.