Poročilo o preučevanju paralelnih sistemov v namizni igri Reversi pri predmetu porazdeljeni sistemi.

Avtor: Žiga Zupanec
Fakulteta za računalništvo in informatiko
Univerza v Ljubljani
Email: {zz9698@student.uni-lj.si}
Profesor: izr. prof. dr. Uroš Lotrič
Asistent: as. Davor Sluga uni. dipl. ing.

***Povzetek*—Poročilo opisuje pristope pri reševanju problema preiskovanja prostora v namizni igri reversi.**

I. Uvod

Področje tematike.

Področje tematike je raziskovanje različnih paralelnih pristopov pri reševanju problema preiskovanja prostora. V tem poročilu opisujemo problem paralelizacijo Minimax algoritma, ki simulira igranje igre Reversi do določene globine. Pri raziskovanju se opiramo predvsem na članek Parallel Minimax Tree Searching on GPU [1].

Področje je zanimivo, ker lahko delo, ki ga je možno paralelizirati opravimo na posebnih enotah, ki so veliko bolj učinkovite kot pri reševanju takih problemov kot CPE. S tem se drastično zmanjša čas izvajanja.

Opis problema.

V poročilu naslavljamo problem porazdeljenga izvajanja Minimax algoritma. V igri Reversi je približno 10^{28} možnih veljavnih stanj na igralni plošči do katerih lahko pridemo na 10^{54} različnih načinov. Algoritem Minimax simulira potek igre za določenega igralca, nato pa izmed vseh preiskanih potez izbere tako, ki je najbolj perspektivna.

V zgoraj citiranem članku se avtorja omejita na izvajanje algoritma na (do) dveh GPU napravah, naša rešitev pa omogoča izvajanje na več GPU napravah, če je le to možno.

Problem je relativno enostavno prenosljiv na več GPU naprav, ki bi, kakor je razvidno tudi iz trenda v članku, prinesel boljše rezultate.

Pristop k reševanju problema. Implementirali smo porazdeljen Minimax algoritem, ki z uporabo knjižnice Message Passing Interface (MPI) dostavi do določene globine z CPE generiranim številom igralih stanj svoji procesni enoti GPU. Procesne enote iz dobljenih stanj preiskujejo nova stanja naprej v globino. Rezultat, najboljšo oceno, nato pošljejo nazaj CPE, ki ponovno razišče igralno drevo do svoje globine, naprej pa pa uporabi rezultat GPU preiskovanja.

V razdelku V predstavimo rezultate in meritve delovanja našega algoritma. Primerjali smo delovanje:

- sekvenčnega algoritma z rekurzijo
- sekvenčnega algoritma, ki namesto rekurzije uporablja sklad
- algoritma, ki se izvaja na procesni enoti GPU
- porazdeljeno izvajanje na več enotah GPU

do globin 6 in 8.

V nadaljevanju poročila lahko preberete: Motivacijo za snovanje algoritma II, izzive s katerimi smo se soočali III, rešitve IV, rezultate V in zaključek VI.

II. Motivacija

Hoteli smo implementirati algoritem, ki bi, kar se da najbolje izkoristil procesne enote, ki smo jih imeli na razpolago. V času pisanja poročila nismo naleteli na implementacijo Minimax algoritma, ki bi izkoriščal enote GPU. Hkrati je bila to tudi lepa priložnost, da se pobliže spoznamo z tehnologijami predstavljenih pri predmetu porazdeljeni sistemi.

III. Izzivi

V tem razdelku opisujemo učinkovitejšo predstavitev igralnega polja, izboljšave evalvacijske funkcije, izboljšave preverjanja veljavne poteze ter barvanja ploščkov, uporabo sklada namesto rekurzije, kopiranje polja in porazdeljevanje bremena na porcesne enote GPU.

A. Učinkovitejša predstavitev igralnega polja.

Igralno polje je smiselno predstaviti s kar se da minimalno informacije. Minimalna predstavitev igralnega polja prineša manjšo količino prenešenih podatkov pri komuniciranju z napravami, hkrati pa omogoča hitrejšo evalvacijo stranja, kar sicer opisujemo v naslednjem delu. V povezvi s tem, smo se tudi odločili, da za polje enega igralca porabimo 128 bitov, kar je prav tako razloženo v nadaljevanju.

Za predstavitev igralnega polja enega igralca potrebujemo 64 bitov. Igralca sta dva, torej rabimo 128 bitov. Za takšno predstavitev polja smo uporabili $unsignedchar$ oziroma ekvivalenten tip $uint8\_t$. Uporabljamo tabelo, ki jo sestavlja 16 takih tipov ($16 * 8 = 128$ bitov). Posamezen plošček z uporabo pomožnih konstruktov $TILE$, $PACK$ (in $CPACK$) postavimo ali pa preberemo iz oz. na igralno porvšino.

```
#define TILE(a, player, x, y) \
((a[((player << 3) + (x))] >> (7 - (y))) & 0x1)
```

```
#define PACK(a, player, x, y) \
a[(player << 3) + (x)] |= (1 << (7 - (y))); \
a[((1 - player) << 3) + (x)] &= ~(1 << (7 - (y))) & 0xff; \
a[((player + 2) << 3) + (y)] |= (1 << (7 - (x))); \
a[((1 - player + 2) << 3) + (y)] &= ~(1 << (7 - (x))) & 0xff;
```

V naši konkretni implementaciji z 256 bitih, prvih 128 bitov predstavlja igralni polji dveh igralcev kodiranih, kot je opisano zgoraj. To omogoča hitro preverjanje in barvanje ploščkov v horizontalni smeri. Drugih 128 bitov pa sta transponirani igralni polji igralcev, kar omogoča hitro ugotavljanje števila veljavne poteze in hitro barvanje tudi v vertikalni smeri.

B. Izboljšava evalvacijske funkcije.

Ta izboljšava je povezana z opisano predstavitvijo igralnega polja. Pri evalvaciji igralno polje igralca preiščemo v osmih ciklih pri čemer si sproti zapomnimo in povečujemo število nastavljenih bitov pri predstavitvi podatka, ki je tipa $uint8\_t$. Pri ugotavljanju števila bitov v posameznem podatku smo napisali t.i. lookup tabelo, ki pohitri delovanje. Evalvacijska funkcija vrne število ploščkov enega igralca, število ploščkov drugega igralca pa ugotivimo če to število odštejemo od števila vseh ploščkov na igralni plošči, ki ga je trivialno beležiti.

C. Izboljšave preverjanja veljavne poteze ter barvanja.

Razlog, da smo se odločili, da igralno površino predstavimo z 256 biti je, da nam taka predstavitev omogoča izredno hitro ugotavljanje veljavne poteze in hitro barvanje ploščkov. Osrednje delovanje teh dveh funkcij temelji na dveh lookup tabelah. Ena je find first set, ki vrne mesto prvega najmanj pomembnega bita v podatku tipa $uint8\_t$, druga pa sporoči število ničel do prvega najbolj pomembnega bita. S tem pridobimo hitro ugotavljanje veljavnosti poteze v horizontalni in vertikalni smeri in vrnemo ugotovitev še predno je potrebno preveriti diagonale, ki so računsko bolj potratne. Obe funkciji uporabljata hitro bitno aritmetiko s kar se da malo zankami.

D. Uporaba sklada namesto rekurzije.

Da bi naš algoritem lahko uporabili na GPU napravah, je bilo potrebno rekurzivni Minimax algoritem preoblikovati, da deluje s skladomom. Skladovna rešitev deluje v neskončni zanki. Predno se spustimo nivo niže, si zapomnimo trenutno stanje v polju in ga damo na sklad. Zapomniti si je potrebno še trenutne koordinate igralnega polja, in trenutnega igralca. Izvajanje nadaljujemo naprej v neskončni zanki na novi globini in postopek ponavljamo. Ko dosežemo maksimalno globino evalviramo igralno polje, vzamemo polje iz sklada in se pomaknemo više, kjer nadaljujemo z izvajanjem tam, kjer smo končali. Ko ni več veljavnih potez ali pa ko dosežemo plitvino, prekinemo neskončno zanko in vrnemo najboljši rezultat.

E. Kopiranje polja.

Da bi lahko enak Minimax algoritem uporabili tako na CPE kot GPE smo funkcije memcpy nadomestili z lastnimi prepisovalnimi konstrukti. Pri tem smo uporabili trik, ki namesto, da bi igralno polje prepisovali tako, ko je predstavljeno (s tabelo 32 $uint8\_t$), igralno polje predstavimo kot tabelo osmih $uint32\_t$. Tako namesto z 32 cikli, tabelo prepišemo zgolj v osmih ciklih.

F. Enakomerno porazdeljevanje bremena.

Pri porazdelitvi bremena naročimo (npr osmim napravam), da preiščejo in vrnejo nekaj začetnih polj generiranih do določene globine. Število s CPE generiranih začenih polj je različno, ker je preiskovanje z algoritmom Minimax dinamičen problem. Spisali smo algoritem pri katerem gospodar dobi število generiranih polj vsakega sužnja. Gospodar nato sestavi plan porazdeljevanja bremena, ga pošlje sužnjem, ti pa nato v skladu s planom pošljejo generirana polja izbrani GPE enoti. Algoritem ima sicer slabo časovno zahtevnost, a je problem zelo majhen, po drugi strani pa močno zmanjšamo čas medsebojne sinhronizacije, torej čakanja med enotami. Algoritem je pristranski, saj za voljo minimalne komunikacije polja, ki jih je generirala enota, ki poseduje GPE ne prestavlja.

IV. Rešitve in delovanje algoritma.

Delovanje našega algoritma Minimax se začne s postavitvijo nasprotnikovega ploščka na igralno polje. Gospodar to igralno polje pošlje vsem dodeljenim enotam. Vsaka enota generira nekaj začetnik igralnih polj in jih pošlje enotam GPE. Te nadalje raziščejo igralno drevo in vrnejo rezultate za posamezno polje. Vse dodeljene enote nato spet preiščejo drevo kot na začetku, ko pridejo do maksimalne globine, pa uporabijo rezultat, ki jim ga je izračunala enota GPE. Vse enote nato pošljejo najboljše rezultate gospodarju, ta izbere potezo z najboljšo oceno in postavi plošček. Krog se ponovi dokler igre ni konec, tj. dokler ima vsaj eden od igralcev veljavne poteze ali pa, ko je vsa igralna površina pokrita.

A. Vloga gospodarja, sužnjev in naprav GPE.

Ko nasprotnik postavi plošček na igralno polje, pošlje gospodar z ukazom MPI_Bcast to polje osmim enotam. Vsaka enota razišče svoje igralno drevo do določene globine, začenši z vrstico, ki zavisi od identifikacije enote (gospodar dobi razišče prvo vrstico, suženj i, razišče $i - to$ vrstico, itd.). Ko enota konča s preiskovanjem pošlje število generiranih začetnih polj gospodarju. Ta z uporabo zgoraj predstavljenega balance algoritma particijo razdeli na dva kar se da enaka dela (dve napravi GPE) in pošlje plan pošiljanja generiranih polj vsem deležnikom. Ti v skladu s planom pošljejo svoja geneirana polja izbrani GPE napravi. Gospodar, ki ima hkrati tudi enoto GPE, pošlje celoten plan razdeljevanja še drugi enoti, ki ima GPE.

Enoti, ki imata GPE raziščeta dobljena polja naprej v globino. Dobljene rezultate pošljeta nazaj enotam, od katerih sta dobili igralna polja.

Vse enote ponovno preiščejo svoje igralno drevo. Ko pridejo do maksimalne globine, uporabijo rezultate ocene vozlišča, ki jim ga je poslala enota z GPE. Čas prvega in drugega preiskovanja je torej enak. Ko enota zaključi s preiskovanjem pošlje svojo najboljšo oceno skupaj s koordinatama gospodarju. Gospodar izmed (do osmih) dobljenih ocen izbere najboljšo in postavi plošček. Postopek se ponavlja dokler igre ni konec.

V. Rezultati

A. Testno okolje

Testiranje je potekalo na računalniku Laboratorija za adaptivne sisteme in paralelno procesiranje (LASPP) fakultete za računalništvo in informatiko. Računalnik ima 24 procesorjev Intel(R) Xeon(R) CPU E5-2620, vsak procesor ima 6 jeder. V računalniku je 64 GB delovnega pomnilnika. V računalniku

Slika 1. Čas celotnega izvajanja ene igre v posamezni globini (v sekundah).

algoritem: / globina:	6	8
sekvenčni, rekurzija	11.673658	NA
sekvenčni, sklad, bit board, eval	8.975160	886.484822
paralelni, GPE	2.504254	229.624882
porazdeljeni (8 enot), GPE	2.765659	294.380426
porazdeljeni (8 enot), 2x GPE	1.488864	154.983645

Slika 2. Povprečni čas za na potezo v posamezni globini (v nanosekundah).

algoritem: / globina:	6	8
sekvenčni, rekurzija	249.5	NA
sekvenčni, sklad, bit board, eval	191.8	18946.3
paralelni, GPE	53.5	4907.6
porazdeljeni (8 enot), GPE	59.1	6291.6
porazdeljeni (8 enot), 2x GPE	31.8	3312.4

sta dve GPU procesni enoti NVIDIA Tesla K20m. Vsaka GPU procesna enota ima 2496 jeder in 5 GB pomnilnika. Računalnik ima tudi koprocesor Inter Xeon Phi z 60 jedri s po štirimi nitmi na jedro, skupaj torej 240 niti. Koprocesor ima 6 GB delovnega pomnilnika.

B. Meritve

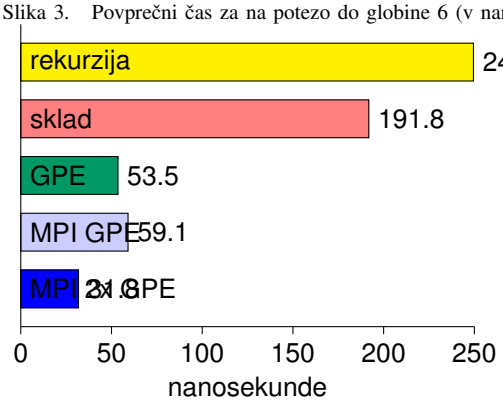
Testiranje algoritma v porazdeljenem načinu smo uporabili 8 naprav. Stvar zasnove je, da je bilo to 8 procesorjev na istem računalniku a kot implementirano bi predpostavljamo, da se procesorji obnašajo kot fizično ločene naprave. Uporabili so do dve enoti GPE. Vsi algoritmi so preizkušeni z v vnaprej znanim izvajanjem potez (rand(1)), tako, da je končni izid enak pri vseh algoritmih. Algoritmi za to izbrano stanje naredijo 46789261 postavitev ploščkov na globini 6 in 3727675942 postavitev na globini 8.

Časi izvajanja ene partije igre so predstavljeni v tabeli 1, povprečni čas izvajanja kaže tabela 2. Povprečni časi so za boljšo predstavo prikazani tudi na grafu 3 do globine 6 in na grafu 4 do globine 8.

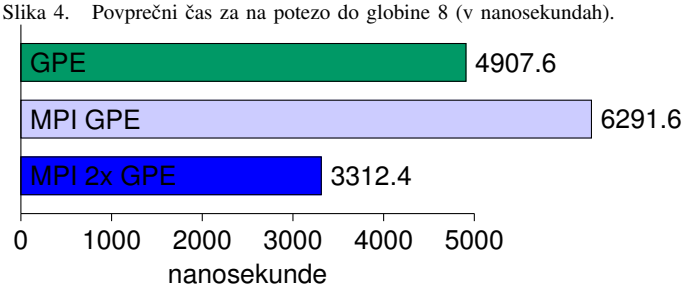
C. Analiza rezultatov

Pohitritev porazdeljenega algoritma, ki uporabi obe enoti GPE je:

Slika 3. Povprečni čas za na potezo do globine 6 (v nanosekundah).



Slika 4. Povprečni čas za na potezo do globine 8 (v nanosekundah).



$$S(n,p) = \frac{t_s(n)}{t_p(n,p)} \qquad = \frac{249.5}{31.8} = 7.8$$

Naivna ocena učinkovitosti je:

$$E(n,p) = \frac{t_s(n)}{p \times t_p(n,p)} \qquad = \frac{7.8}{8.0} = 0.975$$

VI. Ugotovitve & morebitne izboljšave

Rezultati so presenetljivo dobri, nad pričakovanji. Omeniti velja, da je čas izvajanja na porazdeljenih napravah z eno enoto GPU večji od časa izvajanja na enoti GPE brez porazdelitve ostalim napravam. Tudi to je pričakovano, saj medsebojna komunikacija med enotami zateva nekaj dodatnega dela.

Potencialne možnosti za izboljšave vidimo v implementaciji algoritma alpha-beta rezanje na GPE napravah (Parallel Alpha-Beta Algorithm on the GPU [2].), ki pa je zelo težavno za pravilno implementacijo in ne prinaša nujno dobrih rezultatov.

Literatura

[1] K. Rocki and R. Suda, "Parallel minimax tree searching on gpu," in *Parallel Processing and Applied Mathematics*. Springer, 2010, pp. 449–456.

[2] D. Strnad and N. Guid, "Parallel alpha-beta algorithm on the gpu," in *Information Technology Interfaces (ITI), Proceedings of the ITI 2011 33rd International Conference on*. IEEE, 2011, pp. 571–576.