UNIVERZITA KOMENSKÉHO

FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

KATEDRA INFORMATIKY

GENEROVANIE STABILNÝCH MODELOV VYUŽÍVANÍM CUDA TECHNOLÓGIE

BAKALÁRSKA PRÁCA

PETER CIEKER

Študijný odbor : 9.2.1 Informatika

Vedúci : Mgr. Michal Čertický

Bratislava, 2010

Čestné vyhlásenie

Vyhlasujem, že som túto bakalársku prácu napísal samostatne a použil iba uvedenú literatúru.

Bratislava, 11.06.2010 ....

podpis

Poďakovanie

Ďakujem vedúcemu tejto bakalárskej práce Mgr. Michalovi Čertickému za veľkú pomoc, ešte väčšiu trpezlivosť a neoceniteľnú konštruktívnu kritiku, bez ktorých by táto práca nedospela do svojej finálnej podoby.

Abstrakt

Náplňou tejto bakalárskej práce je okrem dôkladného preskúmania problematiky logického programovania so stabilnomodelovou sémantikou, hlavne návrh a implementácia generátora stabilných modelov s paralelizovaným procesom groundovania optimalizovaným pre technológiu CUDA. Samotné podklady slúžiace ako inšpirácia pre túto prácu už opisujú algoritmy pre paralelné groundovanie, sú však zamerané na bežné viacjadrové CPU, ktoré disponujú malým počtom výkonných výpočtových jednotiek umožňujúcich komplexné operácie s množinami. Jednotlivé CUDA jadrá nachádzajúce sa na grafických kartách spoločnosti NVIDIA sa nedajú porovnávať s výpočtovou silou CPU jadier, ale ich počet je niekoľkonásobne vyšší, v čom spočíva ich výpočtová sila. Predmetom tejto práce bolo preto implementovať a zjednodušiť proces groundovania, ktorý je súčasťou generovania stabilných modelov, až na takú úroveň, aby CUDA jadrá boli schopné tento proces vykonať paralelne.

Kľúčové slová : groundovanie logických programov, generovanie stabilných modelov, CUDA

Abstract

The aim of this bachelor thesis is not only an in-depth study of logic programming and stable-model semantics, but also a design and implementation of stable model generator with parallel grounding process, optimized for CUDA technology. Papers, that served as an inspiration for this work already presented an algorithms for parallel grounding. However, they are designed for use with common multi-core CPUs with small number of advanced processing units, which are capable of complex set operations. Individual CUDA cores of NVIDIA graphics processors cannot be compared to CPU cores in terms of computational strength, but their number is significantly higher, what makes them a promising option. The subject of this work was therefore a simplification and implementation of grounding process, so that CUDA cores were capable of performing it in parallel.

Key words : grounding logic programs, computing stable models, CUDA

Predhovor

Súčasťou získavania informácií k tejto téme bolo získať prehľad o existujúcich programoch, ktoré generujú SM. V súčasnosti je programov, ktoré sa venujú generovaniu stabilných modelov viac, ale ja spomeniem dva, ktoré sú pravdepodobne na najvyššej úrovni. Zo strany open source je to program SMODELS a DLV ako reprezentant druhej strany. Autori DLV projektu ale ochotne poskytli referencie k jednotlivým problémom týkajúcich sa generovania SM.

SMODELS sa skladá z dvoch častí. LPARSE, ktorý logický program zagrounduje a následne vygeneruje výstup vo formáte pre program SMODELS, ktorý z tohto výstupu vygeneruje všetky stabilné modely.

DLV je pre nekomerčné účely zadarmo ma internete dostupný. Jeho generátor je opísaný vo viacerých publikáciach, ktoré sú taktiež voľne dostupné a ktoré slúžili ako základ pre túto prácu.

Jadro problému tejto problematiky leží v zagroundovaní logického programu, t. j. dosadení konštánt za premenné všetkými možnými spôsobmi, pričom dochádza k exponenciálnemu rastu zložitosti.

Obsah

1 ASP – Answer set programming....10
2 CUDA - Compute Unified Device Architecture....12
2.1 História....12
2.2 CUDA C jazyk....12
2.2.1 Hierarchia vlákien....14
2.2.2 Hierarchia pamäte....15
3 Návrh riešenia....17
3.1 Efektívne groundovanie logických programov....17
3.1.1 Úroveň Komponentov....17
3.1.2 Úroveň Pravidiel....19
3.1.3 Úroveň pravidla....19
3.2 Implementácia....20
3.2.1 Prostredie a jazyk Implementácie....20
3.2.2 Interakcia Visual Studia a CUDA C....21
3.2.3 Závislosti výsledného zlepšenia....21
3.2.3.1 Veľkosť jednotlivých pravidiel....22
3.2.3.2 Počet pravidiel....22
3.2.3.3 Výpočtová sila G. Karty....22
4 Aplikácia....23
4.1 Stavba Aplikácie....23
4.1.1 Parser....23
4.1.2 Generovanie Dependency grafu....24
4.1.3 Generovanie Component Grafu....25
4.1.4 Procedúra Manager....27
4.1.5 Groundovanie a Generovanie Stabilných Modelov....29
4.2 Analýza Programu....29
4.2.1 Procedúra Instantiate....35
4.2.2 Kernel....37
5 Záver....40

6 Literatúra..41
7 Zoznam Príloh...42
7.1 Priložené CD...42

1 ASP – Answer set programming

V tejto kapitole si zadefinujeme syntax a sémantiku ASP. Stabilné modely vytvárame z normálnych logických programov (z angl. Normal Logic Program).

Definícia 1 (Term) je premenná alebo konštanta.

Definícia 2 (Atóm) je výraz tvaru $p(t_1, \ldots, t_n)$, vytvorený z predikátového symbolu p a termov $t_1, \ldots, t_n$. Atóm, ktorý neobsahuje žiadnu premennú, sa nazýva základný (groundovaný) .

Definícia 3 (Literál) je atóm *a*, alebo jeho negácia *not a*. Literál, ktorý neobsahuje žiadnu premennú sa nazýva základný(groundovaný).

Definícia 4 NLP je každá množina pravidiel *r* tvaru

$$a \leftarrow b_1 \wedge b_2 \wedge \ldots \wedge b_k \wedge not\, b_{k+1} \wedge \ldots \wedge not\, b_m$$

kde *a* a $b_1, \ldots, b_m$ sú atómy a $n \geq 0, m \geq k \geq 0$. Atóm *a* je hlava *r (Head(r))* a konjunkcia $b_1 \wedge \ldots \wedge b_k \wedge not\, b_{k+1} \wedge \ldots \wedge not\, b_m$ je telo *r (Body(r))*. Pravidlo bez hlavy sa nazýva integrity constraint. Ak je telo prázdne (t. j. k=m=0), tak ho nazývame faktom.

Pravidlo r je bezpečné, ak každá premenná v r sa vyskytuje aj v nejakom pozitívnom literály $p \in Body(r)$.

ASP program P je konečná množina bezpečných pravidiel. Term, atóm, literál a pravidlo tvoria základ pre syntax ASP programov.

Nasleduje definícia sémantiky, s ktorou sa pri generovaní SM budeme stretávať.

Definícia 5 Majme pravidlo *r* vyskytujúce sa v programe *P*. Ground inštancia *r* je pravidlo získané z *r* nahradením každej premennej *X* v *r* konštantou *f(x)*, kde funkcia *f* je substitúcia, ktorá zobrazuje premenné vyskytujúce sa v *r* na konštanty v Herbrandovskom univerze (ďalej *U(p)*). *Ground(P)* označuje množinu všetkých ground inštancií pravidiel vyskytujúcich sa v *P*.

Herbrandovské univerzum jazyka *L* je množina všetkých základných termov jazyka.

Herbrandovská báza jazyka *L* je množina všetkých základných atómov *L*.

Definícia 6 (Herbrandovská interpretácia) logického programu P je nejaká podmnožina Herbrandovskej bázy jazyka L programu P.

Definícia 7 (Herbrandovský model) logického programu *P* je Herbrandovská interpretácia I programu *P*, v ktorej je každé pravidlo z *P* splnené. t. j.

$$\forall \; pozitívne \; p \in Body(r) \, , \, p \subseteq I \Rightarrow Head(r) \in I$$

Definícia 8 Pre každú interpretáciu *I* existuje programový redukt P^I , ktorý dostaneme z *P* vymazaním

1. každého pravidla *r*, ktoré obsahuje literál $not \; a \in Body(r) \wedge a \in I$
2. každého literálu *not a* takého, že $a \notin I$

Definícia 9 Každá interpretácia *I* normálneho logického programu P je stabilný model práve vtedy, keď I je minimálny model programov reduktu P^I . Množinu všetkých stabilných modelov programu P označujeme ANS(P).

Definíciu 10 podľa databázovej terminológie nazývame predikát, ktorý sa vyskytuje iba vo faktoch ako *EDB* predikát a všetky ostatné ako *IDB* predikáty. Množinu všetkých faktov logického programu *P* budeme označovať ako *EDB(P)*.

2 CUDA - Compute Unified Device Architecture

V roku 2006, spoločnosť NVIDIA predstavila $CUDA^{TM}$, paralelnú výpočtovú architektúru, ktorá využíva NVIDIA grafické jadrá na riešenie komplexných výpočtových problémov oveľa efektívnejšie ako je to možné na jadrách procesora (CPU).

2.1 História

História CUDY siaha až do čias, keď spoločnosť AGEIA ešte nebola súčasťou NVIDIA spoločnosti. AGEIA a ich PhysX herná technológia priniesla novú myšlienku do sveta vývoja hier. Nevýhodou bolo však, že ku klasickej grafickej karte bolo treba zapojiť ďalšie zariadenie, ktoré samostatne počítalo fyziku hier t. j. gravitáciu objektov, detekciu kolízií, a pod. Táto myšlienka sa však nestretla hneď s úspechom vo svete, no neskôr sa stala inšpiráciou pre nové generácie grafických kariet.

PhysX bol prepísaný neskôr do CUDY a v roku 2008 sa stala AGEIA súčasťou NVIDIE.

[9] CUDA projekt bol uvedený na trh v novembri 2006 spolu s grafickou kartou G80. Verejná beta verzia SDK (softvérového vývojárskeho balíku) vyšla vo februári 2007. O rok neskôr vyšla ďalšia verzia, s malými zmenami, ako CUDA 1.1, ktorá uviedla CUDA funkcie na obyčajných NVIDA ovládačoch. To znamená, že každý CUDA program potreboval na spustenie len grafickú kartu z rady GeForce 8 a vyššie a k tomu ovládač verzie 169.xx a vyššie. Toto bolo veľmi dôležité pre vývojárov, pretože tieto podmienky zaručovali beh CUDA programov na hocijakom počítači.

Najnovšie vyšla CUDA 3.0 z radou nových vylepšení, ktoré rastúca komunita vývojárov s nadšením privítala.

2.2 CUDA C jazyk

Pre CUDU môžeme programovať príkazy vo viacerých jazykoch. Najrozšírenejšie je programovanie v jazyku C, v ktorom je aj táto práca naprogramovaná. Iné sú napr. CUDA FORTRAN, OpenCL alebo Direct Compute. C bol vybraný z dôvodu, že umožňuje ľahkú prácu so smerníkmi a je veľmi rozšírený v

informatickom svete a teda má dobrú dokumentáciu. Pozrime sa bližšie na to ako sa programuje v CUDA C.

Sú tri najzákladnejšie úrovne: hierarchia vlákien, zdielanej pamäte a synchronizácie [3]. Tieto úrovne nás vedú k tomu, aby sme problémy delili na podproblémy, ktoré je možné riešiť nezávisle v blokoch skladajúcich sa z vlákien a tieto podproblémy ešte na menšie, ktoré je možno riešiť v rámci blokov súčasne, využívajúc vlákna vo vnútri každého bloku.

CUDA C využíva jazyk C aby umožnila programátorovi definovať C funkcie, nazývané kernely, ktoré keď sú zavolané, sú spustené N krát paralelne N rôznymi CUDA vláknami. Ináč ako v klasickom C, kde je funkcia spustená len raz.

Toto je základný recept na programovanie využívajúc CUDA technológiu. Pre programátora, ktorý sa už stretol s paralelným programovaním, nie je CUDA C ničím výnimočná a preto si veľmi rýchlo osvojí jednotlivé ovládacie prvky. Keďže aj programovacie prostredie je možné si vybrať, netreba predpokladať problém pri štartovaní. NVIDIA navyše uverejnila ukážkové projekty, ktoré sú uverejnené na ich stránke pre vývojárov. Majú veľmi jednoduchú syntax, sú dobré okomentované a pokrývajú široké spektrum možných problémov, ktoré sa dajú riešiť paralelne oveľa efektívnejšie ako sekvenčne.

Veľmi jednoduchý príklad takého problému, kde je dobre využiť CUDU, je súčet dvoch matíc veľkosti n x n. Políčko na pozícii *i,j* (*i* značí pozíciu riadka, *j* pozíciu stĺpca) v prvej matici pripočítame k políčku na tej istej pozícii v druhej matici. Je ľahké vidieť, že budeme mať n^2 operácií súčtu, ktoré sa dajú vykonať v hocijakom poradí bez toho, aby to ovplyvnilo výslednú maticu. Preto je možné vytvoriť *n x n* vlákien v CUDE a tieto nám spočítajú v jednom kroku tento súčet dvoch matíc, narozdiel od CPU, kde sú tieto operácie vykonávané postupne za sebou.

Prejdime na syntax programovania v CUDA C. Vysvetlili sme si, že kernel je C funkcia spustená každým vláknom. Kernel je definovaný využívajúc __global__ deklaračný špecifikátor a počtom CUDA vlákien, ktoré majú tento kernel pri volaní spustiť. Toto číslo sa špecifikuje využívaním novej konfiguračnej syntaxe <<< … >>> . Každé vlákno, ktoré spúšťa kernel dostane priradené jedinečné identifikačné číslo slúžiace ako ID vlákna a ktoré je dostupné vo vnútri kernelu cez threadIdx premennú. V následujúcej ilustrácií vidíme použitie tejto premennej pri sčitovaní dvoch vektorov.

Príklad 1:

```
__global__ void VecAdd( float* A, float* B, float* C)
{
       int i = threadIdx.x;
       C[i] = A[i] + B[i];
}
int main()
{
      …
      VecAdd<<<1,N>>>(A, B, C);
}
```

Každé z N vlákien vykonáva sčitovanie prvkov i z A,B a výsledok uloží do i -tého prvku v C.

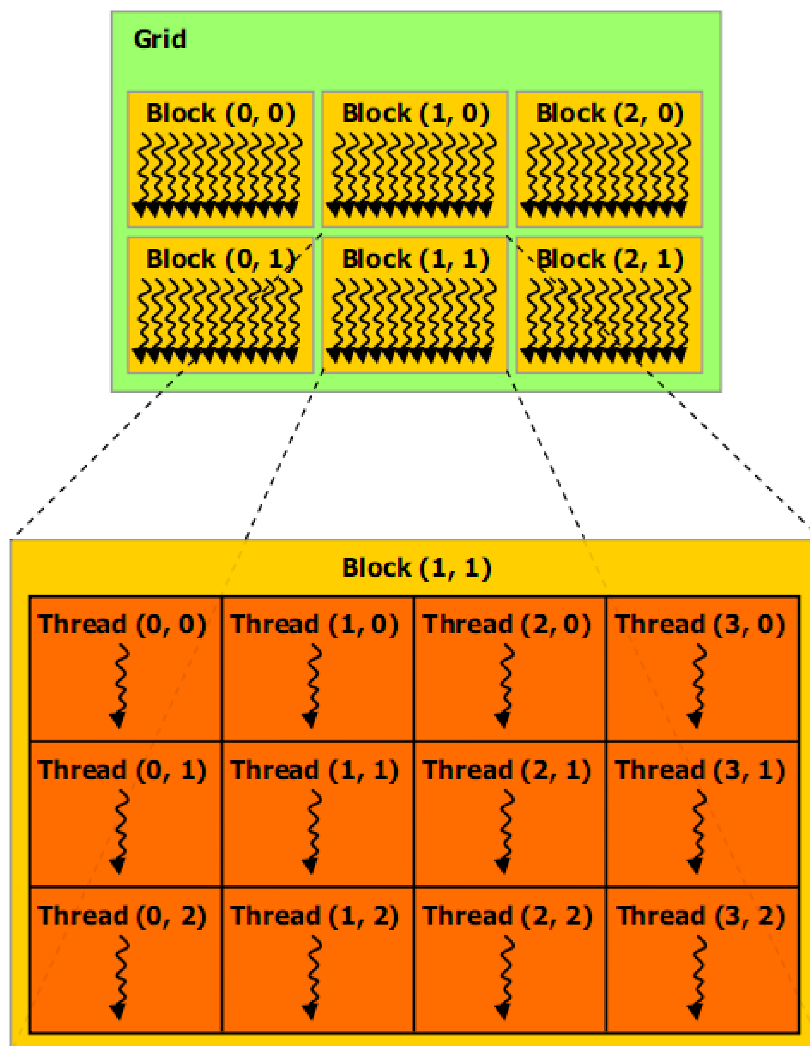
2.2.1 Hierarchia vlákien

Najnižšie v hierarchii stojí samotné vlákno. Vlákna, ktoré spúšťajú ten istý kernel sa spájajú do blokov. Očakáva sa, že všetky vlákna budú pridelené jednému jadru a teda musia si deliť obmedzené prostriedky tohto jadra. Preto je limit 512 vlákien na jeden blok. Vlákna vo vnútri bloku môžu byť usporiadané jedno, dvoj alebo trojrozmerne formujúc tak 1D, 2D alebo 3D blok. Každé vlákno, ktoré je pri spúštaní kernelu (kernel call) vytvorené, obdrží svoje ID podľa vstupných parametrov, ktoré sa týkajú hlavne formy blokov. V tejto práci bude blok jednoduchý, jednorozmerný a bude obsahovať 512 vlákien.

Bloky sa ďalej formujú do 1D alebo 2D gridov. Každý blok vo vnútri gridu dostane pri spustení vlastný *blockIdx* a *blockDim* ako ID slúžiac na manipuláciu s ním. Každý blok musí byť samostatne spustiteľný a na poradí, v akom budú jednotlivé bloky spustené nesmie záležať. Táto podmienka nám umožňuje priradiť bloky dostupným jadrám a tak písať kód, ktorého rýchlosť je závislá (škálovateľná)od počtu jadier.

Vlákna vo vnútri bloku sú schopné komunikovať medzi sebou pomocou *shared memory*. *Shared Memory* je tzv. Low- latency pamäť s rýchlou odozvou nachádzajúca sa blízko výpočtového jadra.

Funkcia *__syncthreads()* slúži na koordináciu vlákien vo vnútri bloku. Tvári sa ako bariéra, kde všetky vlákna čakajú, až kým sa každé vlákno dostane na túto pozíciu v kóde. Až keď každé vlákno v bloku zavolá túto funkciu, môžu všetky pokračovať. Je dôležité zaručiť, aby funkcia *__syncthreads()* bola vždy dosiahnuteľná každým vláknom, ináč hrozí zaseknutie výpočtu.



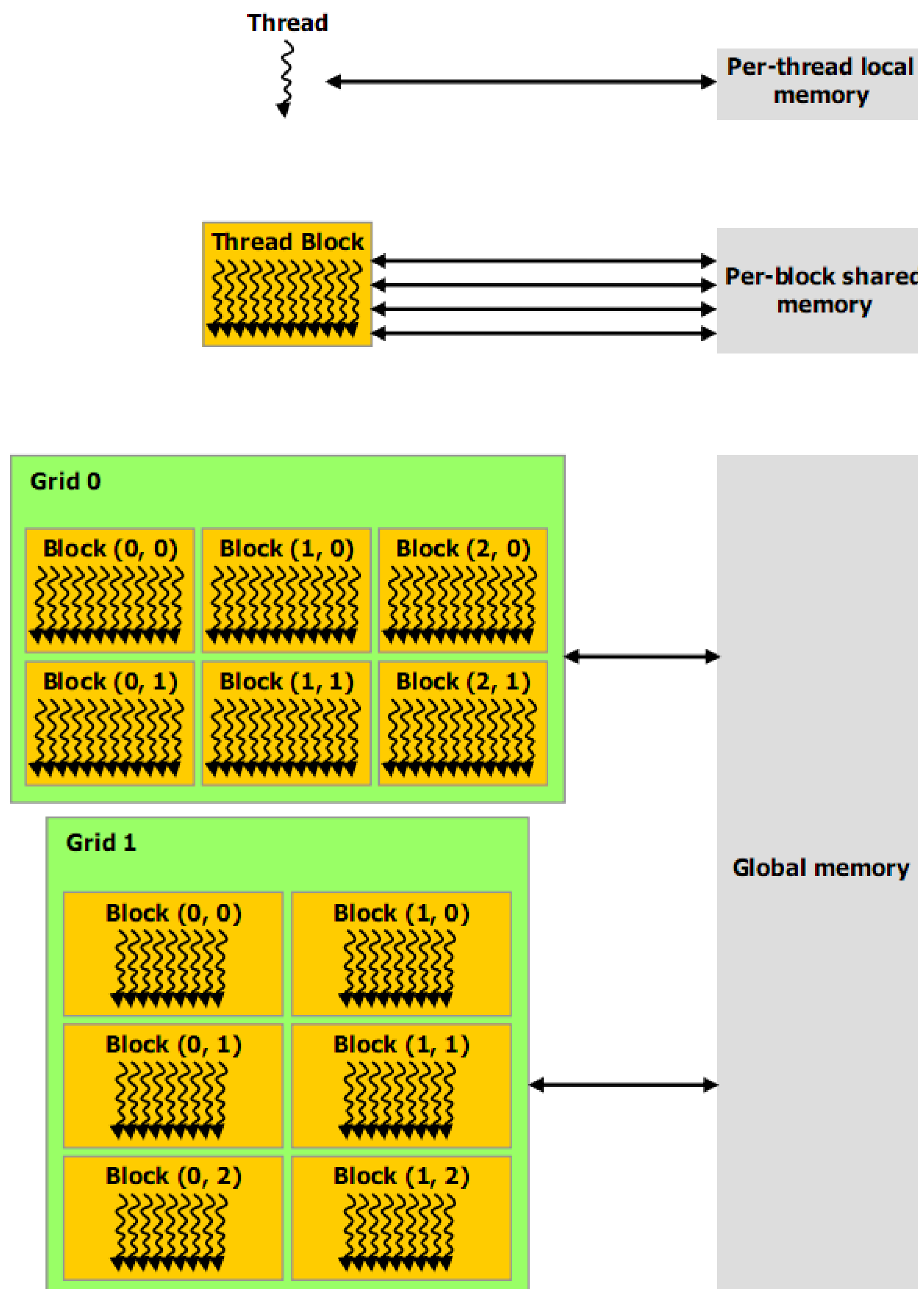
OBRÁZOK 1. Hierarchia vlákien v CUDE

2.2.2 Hierarchia pamäte

CUDA vlákna môžu pristupovať k viacerým typom pamätí počas svojho behu. Každé vlákno ma vlastnú súkromnú pamäť. Všetky vlákna vo vnútri toho istého bloku majú tú istú zdielanú pamäť (shared memory) ku ktorej môžu pristupovať. Táto pamäť má „životnosť" rovnakú ako jej blok. Všetky vlákna majú prístup ku *global memory.*

Okrem týchto troch typov existujú ešte iné, špecifické pamäte, ale v rámci tejto práce nie sú využité.

Programovací model CUDA predpokladá, že vlákna sú spúšťané na fyzicky odlišnom zariadení (*device*) ako je to, na ktorom je spúšťaný C program (*host*). Predpokladá sa, že obe zariadenia sa starajú o svoju vlastnú pamäť a preto je potrebné, aby program mohol zabezpečiť viditeľnosť určitých častí pamäti medzi *host* a *device*. Na toto sú vytvorené špeciálne funkcie starajúce sa o alokáciu, dealokáciu a prenos dát medzi týmito dvoma typmi pamäti.



OBRAZOK 2. Hierarchia pamäťe v CUDE

3 Návrh riešenia

V tejto časti práce si prejdeme dôležité vlastnosti, ktoré chceme, aby výsledná aplikácia implementovala.

3.1 Efektívne groundovanie logických programov

Všeobecne známy problém efektívneho groundovania SM je kľúčová premenná pri výpočte odhadovaného času zložitosti. Keby sme naivne preberali všetky možnosti, tak nám zložitosť rastie exponenciálne. Napríklad, keby sme mali jednoduchý program, v ktorom sa vyskytuje 5 rôznych premenných a 4 termy, tak s kombinatoriky vieme, že množina všetkých možností sa rovná 4^5 = 1024 možností.

Na efektívne groundovanie sa môžeme dívať ako na snahu túto zložitosť znížiť až na absolútne minimum potrebných zagroundovaných inštancií. V roku 2007 vyšiel [1] úspešný pokus posunúť groundovanie SM dopredu. Autori sa vo svojej práci venujú rozširujúcemu sa trendu viacjadrových procesorov a spôsobu, ako túto novú výpočtovú silu naplno využiť. Opisujú inštanciovanie spôsobom ako to robí DLV a využívajú ho v novej, vlastnej implementácií. V roku 2009 a 2010 vydali doplnenia[2] [3] k tejto práci a poukazujú na tri vrstvy v procese groundovania, kde je možné využiť viacjadrové výpočtové štruktúry.

3.1.1 Úroveň Komponentov

Prvým je tzv. Components level. Na tejto úrovni vieme podľa vzájomnej závislosti IDB predikátov určiť, ktoré moduly možno paralelne groundovať, bez toho, aby sme ovplyvnili výsledný proces [3]. Je možné určiť poradie modulov také, že modul C_i závisí len od informácií získaných inštanciovaním modulov C_j , pričom j<i.

Pre vstupný program P je vytvorený Dependency graf a z toho potom Komponent graf. Relácia medzi týmito grafmi je taká, že keby sme v Dependency grafe každý cyklus skomprimovali do jediného vrcholu (hrany vo vnútri cyklu odoberieme a vrcholy vymažeme až na jeden, ktorý bude obsahovať všetky zvyšné hrany vymazaných vrcholov), tak sa tieto dva grafy rovnajú.

Modul C je možné vybrať na inštanciovanie, ak nemá žiadne pozitívne hrany vedúce do neho, alebo keď všetky hrany vedúce do neho sú negatívne. Z definície Komponent grafu vyplýva, že vždy je možné vybrať nejaký vrchol, lebo keby nebolo,

znamenalo by to podľa definície SCC to, že Component graf bol nesprávne vytvorený z dependency grafu a že všetky vrcholy v grafe majú pozitívnu hranu vchádzajúcu do seba t. j. existuje aspoň jeden cyklus medzi dvoma modulmi. Dokážeme to sporom.

Majme Component Graf (CGp) , ktorý bol správne vytvorený z Dependency grafu(DGp). To, že CGp obsahuje cyklus znamená, že existujú dva komponenty $C, D \in CG_p$ a dve hrany $e_1, e_2 \in CG_p$; $e_1 = (C, D) \wedge e_2 = (D, C)$. To ale znamená, že v DGp existovali dve hrany spájajúce aspoň jeden vrchol z C s nejakým (aspoň jedným) vrcholom z D. Vytvorme nový graf H, ktorý bude mať ako vrcholy množinu vrcholov komponentov {C} a {D} a hrany H budú všetky hrany medzi týmito vrcholmi a pridáme ešte hrany e_1, e_2 . Každý vrchol z C je dosiahnuteľný vrcholom z C podľa definície SCC a to platí aj pre vrcholy z D. Treba ukázať, že každý vrchol z C je dosiahnuteľný z D a naopak. Vieme, že existujú nejaké vrcholy pre naše dve hrany e_1, e_2 také, že $(x_1, x_2 \in C) \wedge (y_1, y_2 \in D); (e_1 = (x_1, y_1)) \wedge (e_2 = (y_2, x_2))$. Čiže vyberme ľubovoľný štartovný vrchol $v_1 \in C$ a ľubovoľný cieľový vrchol $v_2 \in D$. Z definície C vieme, že x_1 je dosiahnuteľný z v_1 . Z x_1 vedie hrana do y_1 . Z definície D vieme, že v_2 je dosiahnuteľný z y_1 .

Takisto využijúc hranu e_2 sa vieme dostať z v_1 do v_2 . Čiže v tomto grafe existuje cyklus, ktorý obsahuje všetky vrcholy. Lenže to je spor s definíciou SCC, že C a D sú maximálne množiny vrcholov také, že každý vrchol je dosiahnuteľný z každého vrcholu.

Tento dôkaz nám zaručuje, že vždy budeme môcť vybrať aspoň jeden vrchol z cG_p , ktorý nebude mať žiadnu pozitívnu hranu vedúcu do neho čo značí, že komponent, ktorý tento vrchol v cG_p reprezentuje, nie je priamo závislý na informáciach získaných z inštaciacie iného komponentu a teda môže byť vybraný na tento proces.

Dokázali sme, že teda jeden vrchol bude vždy možné vybrať. V praxi je však takýchto vrcholov viac. Keďže každý jeden spĺňa hore uvedenú podmienku, je možné ich inštanciovať paralelne, ideálne všetky naraz.

Toto je prvá úroveň paralelizmu, ktorú možno využiť pri groundovaní LP. Logická implementácia týchto faktov je vrámci groundovania LP spustiť *n* vlákien, pričom *n* reprezentuje počet nezávislých vrcholov získaných hore uvedeným postupom.

Keď sa vlákno spustí, je priradené jednému procesoru. Je zrejmé, že počet vlákien ktoré sa efektívne vykonávajú naraz je obmedzený počtom jadier vo výpočtovej jednotke počítača. V dnešnej dobe už sú dostupné viacjadrové procesorové architektúry, no ich počet jadier je stále ešte malý v porovnaní z množinou vlákien, ktoré je možné spustiť naraz.

Tento postup okrem toho, že nám vytvára nezávislé komponenty, pomáha určiť „potrebné“ informácie na groundovanie LP, čím je možné obísť kombinatorickú explóziu zložitosti, ktorá vzniká pri inštanciácií LP.

3.1.2 Úroveň Pravidiel

Druhá úroveň je tzv. Rules Level čo doslovne znamená úroveň pravidiel. Na tejto úrovni delíme jednotlivé pravidlá do dvoch skupín, rekurzívne a „exit" pravidlá. Rekurzívne sú tie, v ktorých sa vyskytuje predikát $p \in C$ v pozitívnom tele pravidla $r \in C$, pričom $C \in G_p$. Všetky ostatné sú exit rules. Podľa nasledujúcej techniky vieme paralelne spracovať najprv všetky exit rules a potom postupne každé rekurzívne pravidlo využívaním semi - naívnej evaluačnej techniky[3].

Pri každej iterácií *n*, inštanciácia rekurzívnych pravidiel je vykonávaná súbežne a využívajúc iba dôležité informácie získané počas iterácie *n-1*. Toto je realizované rozdelením dôležitých atómov do troch skupín: S_1, S_2 a NS. V NS sú atómy získané počas prebiehajúcej iterácie (*n*) . V S_1 sú atómy získané v priebehu predošlej iterácie (*n-1*) a S_2 obsahuje tie získané doteraz (až po iteráciu *n-2*).

Prvotne, S_1 a NS sú prázdne a S_2 obsahuje všetky informácie získané v doterajšom priebehu inštanciácie. Na začiatku každej novej iterácie, NS je pridaná do S_1, t .j. informácie získané počas iterácie *n* sú pokladané za dôležité pre iteráciu *n+1*. Potom sú rekurzívne pravidlá spracované naraz a každá z nich využíva informácie z S_1. Na konci iterácie, keď je výpočet každého pravidla ukončený, množina S_1 je pridaná do S_2. Výpočet končí vtedy , keď nie je žiadna nová informácia získaná v aktuálnom kroku (t. j. $NS = \emptyset$).

3.1.3 Úroveň pravidla

Tretia úroveň je tzv. Single Rule Level (SRL). Pretože túto techniku nevyužívame, opíšeme ju len veľmi stručne. Dve doteraz uvedené techniky paralelnej

inštanciácie sú efektívne, keď pracujeme s veľkým programom. Ale pri krátkych programoch sú tieto techniky v najhoršom prípade dokonca časovo náročnejšie, ako sekvenčná inštanciácia.

V SRL takéto programy umelo „zväčšujeme“ (tzv. Split) a tým dokážeme sekvenčnú prácu paralelizovať. Ale nie vždy sa nám podarí pravidlo dobre rozdeliť. Ideálne sa pravidlo rozdelí na *n* pravidiel s rovnakou náročnosťou a ktorých výsledná inštanciácia je rovnaká ako pôvodného pravidla. Na toto sú využívané špeciálne heuristické metódy na výpočet toho pravidla, ktoré je možné „dobre“ rozdeliť na *n* iných, pričom hodnota *n* je taktiež vypočítaná heuristickou meódou. Viac informácií si o SRL sa dá nájsť v [3].

3.2 Implementácia

3.2.1 Prostredie a jazyk Implementácie

Rozhodol som sa pre Microsoft Visual C++ 2008, aj keď som ešte nemal žiadne skúsenosti s programovaním v tomto vývojovom prostredí. Spoločnosť Microsoft ho poskytuje pre nekomerčné účely zadarmo na svojej internetovej stránke. Spoločnosť NVIDIA založila programovanie pre CUDU aj na jazyku C a v C++ je možné zavolať vyexportovanú funkciu napísanú v C.

Ďalej jazyk C++ ponúka aj dátové štruktúry typu *vector* na prácu s poliami. Keďže v tejto práci budeme pracovať s veľkým množstvom údajov, je pre prehľadnosť kódu lepšie, keď sa čo najviac posnažíme vyhnúť práci so smerníkmi. Aplikácia bude obsahovať v sebe prvé dve úrovne paralelizmu, aj keď ich naplno využívať nebudeme.

Je tu otázka, prečo teda robiť tieto operácie pred inštanciáciou, keď aj tak budeme postupovať sekvenčne počas prvej/druhej úrovne. Je to preto, aby sme sa vyhli naívnemu groundovaniu a to nám tieto dve úrovne dokážu zabezpečiť. Druhou motiváciou je aplikáciu navrhnúť tak, aby ju bolo možné v budúcnosti rozšíriť. CUDA 2.3 podporuje súčasný beh len jediného kernelu, čo je nevýhodou, no mysliac do budúcnosti určite napraviteľnou.

V C++ teda spracujeme vstup a prerobíme ho do fázy, keď budeme mať jednotlivé pravidlá rozdelené na rekurzívne a exit pravidlá. Podľa definície, je proces groundovania mapovanie termov na premenné v tele literálov. Majme predikáty *{ foo, bar ,d }* a konštanty *{ a, b, c }* a nasledujúce pravidlo:

```
foo(X)<- d(X), not bar(X).
```

Potom groundovanie je postupné mapovanie konštánt *{ a, b, c }* na premennú *X*. V tomto prípade groundovaný program obsahuje tri pravidlá. V hore uvedených dvoch technikách, je tento proces realizovaný sekvenčne. Postupne sa vezme každý literál a podľa mapovacieho pravidla je premenným v jeho tele dosadená konštanta.

NVIDIA CUDA nám umožňuje tento proces urobiť paralelne. Podľa počtu litrálov v pravidle, je vytvorený počet aktívnych vlákien (threadov) a tieto podľa počtu premenných v literále dokážu namapovať na každú premennú určenú konštantu.

3.2.2 Interakcia Visual Studia a CUDA C

Čo sa na pohľad zdá byť obyčajná, samozrejmá vec, nie je v skutočnosti až také jednoduché. Svedčí o tom aj fakt, že NVIDIA poskytla k stiahnutiu vzorové príklady tzv. „samples“, kde sú pokryté všetky triviálne prípady vsunutia CUDA kódu do klasického VS projektu s príponou *.cpp .

Táto aplikácia „beží“ väčšiu časť kódu na CPU a až pri groundovaní využívame výpočtovú silu GPU. Preto v súbore main.cpp v niektorom momente potrebujeme zavolať kernel (kernel.cu) napísaný v CUDA C.

CUDA C kód musí byť napísaný v osobitnom súbore s príponou *.cu a musí mať nastavený Build Rule v Properties na Cuda Build Rule (+ verzia, aktuálne v2.3.0). Aby VS projekt úspešne “buildlo”, treba nastaviť ešte radu dôležitých ciest (pathov) k rozličným súborom. Odporúčam skopírovať celý projekt do priečinku so vzorovými príkladmi, čím sa dá obísť nastavovanie týchto ciest v konkrétnom projekte.

3.2.3 Závislosti výsledného zlepšenia

Tým, že sekvenčný postup groundovania logických programov prerábame na paralelný, je pri určitých podmienkach očakávaný rast efektivity. Výsledné zlepšenie závisí od troch vecí:

1. počet pravidiel
2. veľkosť jednotlivých pravidiel
3. výpočtová sila grafickej karty podporujúcej CUDA technológiu

3.2.3.1 Veľkosť jednotlivých pravidiel

Tento postup transformuje sekvenčné dosádzanie premenných v literáloch na paralelné. Z toho vyplýva, že čím viac literálov v pravidlách, tým viac operácií sa vykoná v jednom kroku a tým viac času ušetreného.

3.2.3.2 Počet pravidiel

Keďže čas je ušetrený pri každom pravidle veľkosti väčšej ako 1, tak potom logicky z toho vyplýva, že čím viac takýchto pravidiel máme, tým efektívnejšie je možné paralelizmus rozvinúť.

3.2.3.3 Výpočtová sila G. Karty

Grafická karta je súčasťou počítača, ktorý sa stará predovšetkým o grafický výstup a spracovanie obrazu. Dôležitým parametrom je veľkosť pamäte, takt a frekvencia čipu. Na vykonanie kódu však potrebujeme grafickú kartu podporujúcu CUDU, t. j. Karta s CUDA jadrami.

NVIDIA grafické karty podporujúce CUDA technológiu sa líšia počtom CUDA jadier.. Jedno CUDA jadro sa skladá zo shaderu a pamäťového bloku. Shader je zložený z viacerých procesorov.

Všetky tieto zložky grafickej karty sú dôležité a ich parametre majú dopad na rýchlosť výpočtu CUDA programov.

Grafické karty podporujúce CUDU sú rady GeForce 8, 9, 100, 200, 400 GPU s minimálnou lokálnou pamäťou 256MB a vybrané rady QUADRO, ION a TESLA.

4 Aplikácia

4.1 Stavba Aplikácie

1.Časť: Parser

2.Časť: Generovanie Dependency Grafu

3.Časť: Generovanie Komponent Grafu

4.Časť: Manažér

5.Časť: Groundovanie Využívaním CUDA technológie

6.Časť: Generovanie Stabilných modelov

4.1.1 Parser

Parser je z angličtiny prebraný názov a znamená syntaktický analyzátor. Ako názov prezrádza, parser vezme vstupný súbor a podľa svojej vnútornej syntaktickej štruktúry hľadá význam v obsahu tohto vstupného súboru.

Inými slovami, rozpoznáva stavbu konkrétneho vstupného programu *P*, počet pravidiel, ich jednotlivé hlavy a telá, rozpoznáva aritu predikátových symbolov, premenné a konštanty.

Aplikácia ako vstup akceptuje textový súbor s názvom „ InputFile.txt “ s nasledovnou syntaxou:

$$L_0(a_1, \ldots, a_{n_a}); ?\, L_{i+1}(c_1, \ldots, c_{n_c}); \ldots; L_k(d_1, \ldots, d_{n_d}); !\, L_{k+1}(e_1, \ldots, e_{n_e});$$
$$\ldots; L_n(f_1, \ldots, f_{n_f}); \quad \%$$

1. Všetky literály môžu mať ako prvý znak „ ! “ ktorý označuje negáciu.
2. Všetky literály začínajú postupnosťou znakov reprezentujúcich meno alebo označenie literálu.
3. Každý literál má konečný počet termov medzi zátvorkami.
4. Každý literál je ukončený znakom “ ; “
5. Hlava pravidla *r* je literál L_0 rešpektujúci body 1. - 4. od začiatku riadku po znak „ ? “.

6. Telo pravidla *r* je (v zmysle definície 4) konjunkciou literálov rešpektujúcich body 1. - 4. od znaku „ ? “ po znak „ . “.
7. Konštanty vo vnútri literálu začínajú malým písmenom a premenné veľkým.
8. Konštanty nevyskytujúce sa v žiadnom pravidle radíme do špeciálneho literálu s názvom „ constants ” ako jeho termy (3. bod).

Program *P* je množina pravidiel od začiatku až po koniec súboru rešpektujúc pri tom body 1.-8.

Aplikácia automaticky rozoznáva fakty už pri parsovaní a vytvorí množinu EDB(P) a IDB(P) v zmysle definície (doplň).

Príklad 2.

p(X);?s(X);!r(X);.
r(X);?t(X);!p(X);.
s(a);?.
t(a);?.
s(b);?.
p(a);?.
p(b);?.

Táto ukážka reprezentuje správny vstup pre aplikáciu.

4.1.2 Generovanie Dependency grafu

Táto časť čerpá námet z publikácií [1][2][3] uvedených v zozname použitej literatúry. Idea je rozdelenie programu *P* do podprogramov, podľa závislosti medzi IDB predikátmi v *P* a identifikovanie tých podprogramov, ktoré sa dajú spustiť paralelne (naraz).

Pre každý program vieme vyrobiť reprezentáciu v tvare Grafu závislosti . Tento Graf je orientovaný, t. j. Gp=<N,E>, kde N je množina vrcholov a E je množina hrán. N obsahuje vrchol pre každý IDB predikát v *P*. E obsahuje hranu *e=(p,q)*, ak existuje pravidlo $r \in P$ také, že *q* sa vyskytuje v *Head(r)* a *q* sa vyskytuje v kladnom /pozitívnom literály v *Body(r)*.

Takýto graf nám rozdelí P do Podprogramov(modulov), čo nám umožní modulárne spracovanie/ vyhodnocovanie. Hovoríme, že pravidlo $r \in P$ definuje predikát p, ak $p \in Head(r)$.

4.1.3 Generovanie Component Grafu

Definícia 11 Majme Graf G=<N,E>. N je množina vrcholov a E je relácia medzi dvoma vrcholmi. $(e=(x,y)) \in E; x, y \in N$ značí, že existuje hrana medzi vrcholmi x a y v G. Hovoríme, že s je cesta z x do y práve vtedy, keď existuje postupnosť $xe_1 v_1 \ldots e_n y$ taká, že pre

$$\forall v_i \in N; \forall e_n \in E; ((v_i = v_j) \Leftrightarrow (i = j)) \wedge ((e_i = e_j) \Leftrightarrow (i = j))$$

Definícia 12 Hovoríme, že dva vrcholy *x,y* sú dosiahnuteľné v G, ak existuje cesta z x do y a naopak.

Definícia 13 (Strongly connected component) C grafu G je maximálna množina vrcholov taká, že každý vrchol v C je dosiahnuteľný každým iným vrcholom v C.

Ináč: Majme G_p príslušný dependency graf. Komponent graf P je orientovaný označený graf CG_p = (N, E, lab), kde N je množina vrcholov, E je množina hrán, a lab: E ->{ +, - } je funkcia priraďujúca každej hrane označenie + alebo - . N obsahuje vrchol reprezentujúci každý maximálny SSC z G_p ; E obsahuje hranu e (B, A), s lab(e) = „+“, ak existuje pravidlo r v P také, že $q \in A$ sa vyskytuje v hlave r a $p \in B$ sa vyskytuje v pozitívnom literály v tele r; navyše, E obsahuje hranu e = (B, A), s lab(e) = „-“, ak obe z nasledujúcich podmienok platia:

(i) existuje pravidlo r v P také, že $q \in A$ sa vyskytuje v hlave r a $p \in B$ sa vyskytuje v negatívnom literály v tele r, a

(ii) (ii) neexistuje hrana e' = (B, A) s lab(e')= „+“.

Definícia 14. Pre každý pár vrcholov A, B z CG_p , A predchádza B ($A \subseteq B$) ak existuje cesta v CG_p z A do B; a A priamo predchádza B ($A \subset B$), ak A predchádza B a zároveň B nepredchádza A.

Pre každý SCC(Strongly Connected Component) C grafu Gp, množina pravidiel definujúcich všetky predikáty v C je nazývaná modulom C. Pravidlo r vyskytujúce sa v module componentu C (t. j. definujúce nejaký predikát $q \in C$)

nazývame rekurzívne, ak existuje predikát $p \in C$, ktorý sa vyskytuje v kladnom *Body(r)*, ináč vravíme, že *r* je exit rule(pravidlo).

V tejto časti programu je nutné rozpoznať všetky SCC. Na toto rozpoznávanie je použitý nasledovný algoritmus, ktorý vymyslel Robert Tarjan[8].

```
Input: Graph G = (V, E)
index = 0
S = empty
forall v in V do  if (v.index is undefined)
               tarjan(v)

procedure tarjan(v)
  v.index = index
  v.lowlink = index
  index = index + 1
  S.push(v)
  forall (v, v') in E do
    if (v'.index is undefined)
        tarjan(v')
        v.lowlink = min(v.lowlink, v'.lowlink)
    else if (v' is in S)
        v.lowlink = min(v.lowlink, v'.index )
  if (v.lowlink == v.index)
    print "SCC:"
    repeat
      v' = S.pop
      print v'
    until (v' == v)
```

Je to depth-first search algoritmus s časovou zložitosťou $O(| V | + | E |)$ (algoritmus ktorý prehľadáva do hĺbky). Konkrétna implementácia tohoto algoritmu je trochu zložitejšia nakoľko tento algoritmus nevracia množinu všetkých SCC v Grafe, ale ju iba krok po kroku vypisuje na výstup.

4.1.4 Procedúra Manager

Ako jadro programu si podľa vzoru článku [1] vezmeme procedúru Manager.

```
Procedure Manager ( P:Program;  CG_p  : ComponentGraf; T:
SetOfAtoms; II: GroundProgram );
begin
  var U, D,R:SetOfComponents; I: SetOfAtoms;
C:SetOfPredicates;
  D= Ø , R= Ø ;U=vrcholy( CG_p )
  T:=EDB( P ); I = EDB ( P ); II:=  Ø ;
  while ( U≠Ø )
     for all  C∈U ;
        if(canBeRun(C,U,R, CG_p ))
           begin
              R =  R∪C ;
              Instantiator(P,C,U,R,D,T,I,II);
            end;
  end.
```

```
Procedure Instantiator(P:Program; C: Komponent; U,R,D:
PoleKomponentov; T,I:PoleAtómov;II:GroundovanýProgram);
begin
  InstantiateComponent(P,C,T,I,II);
  D =  D∪S ;
  R = R - {C};
  U = U - {C};
end
```

Procedúra manager vezme Program P a jeho komponent graf a podľa toho vygeneruje T, pole pravdivých atómov, a II, pole zagroundovaných pravidiel, ako výstup. Ďalej platí, že ANS($T \cup II$) = *ANS(P)*. Najprv sú inicializované polia T,I a II ako v štandardnom DLV Instantiatore. Ďalej vytvoríme tri nové polia komponentov: U, (čo značí *nedokončený* z angl. Undone) pole nedokončených komponentov P, ktoré treba ešte spracovať, D, (čo značí *dokončený* z angl. Done) pole komponentov P ktoré už sú

spracované a R, (čo značí *bežiaci* z angl. Runinng) pole tých komponentov, ktoré sú spracovávané.

Na začiatku sú D a R prázdne, pokým U obsahuje všetky vrcholy z CG_p . Manager kontroluje pomocou funkcie *canBeRun*, či komponenty v U môžu byť inšanciované. Hneď ako je nejaké C spracovateľné, je pridané do R, a nové *Instantiator* vlákno je spustené na inštanciovanie C funkciou *InstantiateComponent*. Keď je inštanciácia C dokončená, C je presunuté z R do D a vymazané z U. Funkcia manager beží až dovtedy, pokým všetky komponenty sú dokončené (t. j. $U=\emptyset$). Funkcia *canBeRun* zisťuje, či je možné komponent C spustiť podľa nasledovnej definície:

Definícia 15. [1] Nech je U pole komponentov, ktoré treba ešte spracovať. Vravíme, že $C\in U$ môže byť spustené, ak pre $\forall A\in U$ platí aspoň jedna z nasledovných podmienok :

(i) $C\subset A$;

(ii) $C\subseteq A$ a $(\neg\exists)A'\in U$ také, že existuje hrana e= (A' , C) z CG_p s lab = „+“, a $\forall K\in R$, neexistujú hrany $e', e''\in CG_p$ také, že e' = (R, C) a e'' = (C, R).

Táto definícia zaisťuje zhruba to, že (i) komponent C nie je spracovaný skôr, ako všetky komponenty priamo predchádzajúce C; a (ii) ak C vyzerá byť v cykle, potom je vybrané len vtedy, keď nemá pozitívne hrany vedúce do seba a priamo nezávisí od nejakých práve bežiacich komponentov. Tie dve podmienky z definície 15 kontrolované funkciou *canBeRun* zaručujú správnosť, keďže rešpektujú závislosti G_p (ako štandardné algoritmy na inštanciaciu).Navyše funkcia *canBeRun* vyberá komponenty, ktoré môžu byť spracované paralelne, bez využívaní tzv. „Mutex lockov“ v dátových štruktúrach ktoré implementujú T a I. Toto nám umožňuje ušetriť prostriedkami a redukovať čas strávený v lock-kolízií. Je ľahké vidieť, že dva komponenty C_1 a C_2 sú vybrané, ak žiadny predikát *p* vyskytujúci sa v tele nejakého pravidla z C_1 sa nevyskytuje v hlave nejakého pravidla z C_2 a naopak. Keď navyše dátové štruktúry implementujúce polia T a I ukladajú atómy v rôznych políčkach, nie je potrebné, aby sme implementovali „mutex locky“ na ich ochranu. Nie je možné, aby dve vlákna zapisovali do toho istého políčka.

4.1.5 Groundovanie a Generovanie Stabilných Modelov

Groundovaním sa budeme vo veľkom zaoberať v ďalšej časti práce. Z groundovaného logického programu P vytvárame SM podľa postupu popísanom v definícii, preto ho nebudeme podrobne popisovať tak, ako zvyšok aplikácie.

4.2 Analýza Programu

Groundovaniu využívaním CUDA technológie sa venujeme v zvyšnej časti práce. Prejdeme si postupne stavbu aplikácie, ako je navrhnutá, prácu jednotlivých objektov pričom budeme implementovať znalosti z predchádzajúcich kapitol. Dá sa povedať, že až teraz prechádzame na aktívnu časť tejto práce. Dá sa to rozdeliť do viacerých častí.

1. analýza programu, jeho vnútornej stavby a hľadanie možností, kde sa dá efektívne implementovať CUDA
2. navrhnutie riešenia ku konkrétnemu rozpoznanému problému.
3. analýza časovej zložitosti pred implementáciou a po nej.

Program sa delí na nasledujúce časti: Parser, vytváranie Dependency grafu, vytvorenie komponent grafu, groundovanie, vytváranie SM. Pokiaľ máme ako vstup jeden súbor a v ňom všetky údaje, nemáme ako ináč čítať z neho ako sekvenčne. Tuto sa časová zložitosť nedá zlepšiť. Nech je v súbore „InputFile.txt“ n znakov, potom aj n znakov postupne prečítame.

Na uchovávanie informácií z tohto procesu, nám slúži objekt *rule.* Reprezentuje práve jedno pravidlo zo vstupného programu P a nejaké pole pravidiel nám určuje presne celý P. Rule objekt sa delí podľa definície pravidla na RuleHead a RuleBody, čo sú reprezentované poliami *predicate* objektov.

Jeden predicate objekt reprezentuje presne jeden predikát z P a skladá sa z mena (*name*), množiny literálov v sebe (*values*) a atribút *negation* nám, indikuje, či je predikát negovaný, alebo nie. Na prácu s atribútmi sú podľa štandardného postupu vytvorené funkcie na operáciu s ich hodnotami t. j. *getName( )*, *setName( )*, *getValue( )*, *setValue()*, atď.

Hlavná premenná sa volá *ProgRules* a reprezentuje celý vstupný program. Vnútorne je reprezentovaná ako pole objektov *rule*, presne tak, ako to je v skutočnosti. Všetky ďalšie operácie budeme vykonávať nad týmto poľom.

Pred tým, ako sa posunieme ďalej teda začneme s generovaním Dependency grafu, treba si uvedomiť čo je na ďalší krok nevyhnutne potrebné. Dependecy graf obsahuje podľa definície vrchol reprezentujúci každý IDB predikát v *P* a hrany podľa závislosti medzi nimi.

Funkcia *IdbFinder* ako názov prezrádza, hľadá tieto IDB predikáty. Moja konkrétna implementácia si vezme ako vstup *ProgRules*, reprezentujúcu *P*, a po ukončení sú prázdne polia *EdbPredicates*, IdbPredicates, *constants* a *variables* naplnené príslušnými hodnotami.

EdbPredikáty sú tie, ktoré sa vyskytujú len vo faktoch. Preto prechádzame *P* a priraďujeme všetky predikáty do poľa *EdbPredicates*. Predikát *p* priradíme do *EdbPredicates* vtedy, ak sa tam ešte nenachádza a jeho telo je prázdne. Predikát *p* priradíme do *IdbPredicates* ak sa tam ešte nenachádza. Následne prechádzame *P* ešte raz a keď predikát *p* nemá prázdne telo (*RuleBody*) tak ho z *EdbPredicates* vymažeme, ak sa tam nachádza. Po druhom prejdení *P* máme správne vytriedené EDB predikáty. Keďže IDB sú všetky ostatné, tak prejdeme *IdbPredicates* a ak je nejaký predikát $p \in IdbPredicates$ zhodný s predikátom $q \in EdbPredicates$, tak ho z poľa IdbPredicates vymažeme.

Predikát *p* je zhodný s predikátom *q* práve vtedy, ak sa rovnajú v mene a pozícii každej konštanty a premennej (konštanta/premenná na i tej pozícií v *p* sa rovná konštante/premennej na i tej pozícii v *q*).

Vlastnosť „*nachádza sa*“ v pravidle je časovo náročná na zistenie. Majme pole A veľkosti *n* prvkov a chceme zistiť, či prvok *x* sa nachádza v *A*. Rozdelíme to na dva prípady: prvok $x \in A$, a prvok $x \notin A$.

V prvom prípade môže byť *x* ideálne na prvej pozícií v *A* a naopak najhoršie bude, ako sa nachádza na konci. Keďže poradie prvkov nie je záväzne dané, môžeme predpokladať, že prvky v *A* sú vždy náhodne usporiadané a teda v priemernom čase nám to bude trvať *n/2* operácií porovnávania.

Je však možné túto priemernú časovú zložitosť zmenšiť využitím viacerých vlákien. Ideálne by bolo mať *n* vlákien na prácu s každým prvkom. Vlákno n_i

porovnáva prvok A_i s prvkom *x* a ak sa rovná, tak nastaví boolovskú premennú *isInA* na true. Ak na začiatku nastavíme túto premennú na false, tak sa môže zmeniť na true iba, ak jeden prvok z *A* sa rovná *x*. Ak predpokladáme, že všetky kroky uvedené trvajú konštantný čas, tak potom vieme na dva kroky zistiť, či $x \in A$.

V skutočnosti je ale nutné sa zaoberať s klasickým problémom, ktoré vlákno kedy môže zapisovať. Keby sme explicitne dovolili vláknam zapisovať naraz, tak výsledný čas je znova dva výpočtové kroky. Väčšina systémov ale nedovoľuje súčasne zapisovať dvom vláknam na tú istú pozíciu. Teda vlákna potom ako zistia, či prvok *x* je rovný tomu i-tému políčku v *A*, musia čakať na právo zápisu do premennej *isInA*. Lenže predpokladajme, že pole vzniklo tak, že nie sú v ňom dva rovnaké prvky. Vtedy môže byť najviac jedno políčko rovnaké ako *x*. Teda najviac jedno vlákno bude žiadať o zápis v druhom kroku.

Čiže vidíme, že s tejto načrtnutej myšlienky dôkazu vyplýva, že na dva kroky vieme zistiť túto vlastnosť prvku *x*. Lenže znova, aj CUDA má obmedzenú výpočtovú silu. Ak je *n* dostatočne veľké, tak nedokážeme v jednom kroku pristúpiť na každé políčko, ale len na *k* políčok, pričom *k* označuje počet výpočtových jednotiek na grafickej karte schopných obslúžiť vlákno. Preto na *n* políčok dokážeme pristúpiť na *(n/k) +1* krokov. Prvý krok teda má časovú zložitosť $\Theta((n/k)+1)$. Druhý krok ostáva ale nezmenený. Stále iba jedno vlákno sa bude snažiť zapisovať do premennej *isInA*. Predpokladajme, že zápis trvá konštantný čas a že zistenie hodnoty *isInA* na konci funkcie trvá tiež len $\Theta(1)$. Potom výsledná časová zložitosť je $\Theta((n/k)+1)$.

Pre architektúry s viac ako 2 jadrami to už prinesie zrýchlenie. Ďalej si treba uvedomiť, že týmto spôsobom sme natesno ohraničili výpočtovú zložitosť z priemernej $O(n/2)$ priemernej na $\Theta((n/k)+1)$.

Vrámci druhého prechádzania *P* sa vnárame do každého predikátu a teda môžeme túto príležitosť využiť a vytvoriť polia *constants* a *variables*. Štandardne, konštanty začínajú malým písmenom a premenné veľkým.

Po funkcii *IdbFinder* môžeme už začať s vytváraním dependency grafu. Celý graf sa dá reprezentovať ako jeden objekt s dvoma poliami *gNode* a *gArcs.* g*Node* podľa definície obsahuje vrchol za každý IDB predikát a *gArcs* podľa závislostí medzi nimi obsahuje hrany tohto grafu.

Vnútorne reprezentujeme *gNode* ako pole objektov *node*. Tento objekt má v sebe tri atribúty *name*, *index* a *lowlink*. *Name* určuje meno IDB predikátu, ktorý daný vrchol zastupuje. Atribúty *index* a *lowlink* využijeme v ďalšom procese generovania komponent grafu. Objekt *node* obsahuje okrem štandardných objektových metód funkciu compare, ktorá slúži na porovnávanie dvoch objektov tohto typu.

Objekt *gArcs* je vytvorený ako pole objektov typu *arc*. *Arc* obsahuje nasledovné tri atribúty: *nodeA*, *nodeB* a *lab*. Vrámci šetrenia zdrojov nám stačí, aby *nodeA* a *nodeB* boli len pointrami na objekt typu *node*. Atribút lab určuje negáciu hrany.

Pri vytváraní (konštrukcií) DG berieme ako vstup program P a množinu IDB predikátov. K nim sledujúc definíciu DG vytvoríme hrany. Hrana e = (x, y) $e=(x, y) \in DG\, p$ vtedy, ak sa predikát *x* nachádza v tele pravidla *r* v ktorom sa *y* nachádza v hlave. Preto prechádzame každú hlavu *P* a takto postupne vytvárame a pridávame do DG_p hrany. Ako výstup vznikne objekt *dependencyGraph*, ktorý je vyššie opísaný.

K časovej zložitosti. Prechádzame v Body(P) každý objekt q typu *predicate* raz a potom pre každý $p \in Head(P)$ vytvoríme hranu e = (q, p). Ak rátame, že proces vyvárania hrany sa skladá z troch krokov trvajúcich konštantný čas $O(1)$, potom nech k je najväčšie číslo také, že veľkosť Head(P) = *k*; *m* je počet literálov v Head(P); *n* je počet všetkých literálov v P; vieme ohraničiť tento proces zhora $O(k*m*3)$. Pričom k,m nikdy neprekročí n, takže definitívne si vystačíme s časovou zložitosťou $O(n^2)$, čo je polynomiálna zložitosť.

DG_p obsahuje kvôli neskorším operáciám metódu findNode(), ktorá dostane ako vstup meno vrchola a na výstup vyhodí index, na ktorom sa vrchol s týmto menom v poli *gNodes* nachádza. Keďže vrcholy sú usporiadané podľa prvého výskytu v *P*, podobne ako pri určovaní vlastnosti „nachádza sa“, keď dodržíme podmienku, že meno hľadaného prvku je náhodne vybrané, priemerný čas na nájdenie prvku v poli je $O(n/2)$. Tu je možné implementovať hocijakú známu stromovú štruktúru, kde by sme mohli túto zložitosť zmenšiť aj bez paralelného prístupu. Napr. V binárnom vyhľadávacom strome je garantovaný čas log(n) na nájdenie určeného prvku. Na jeho vytvorenie ale potrebujeme v najhoršom prípade $O(n^2)$. Ale analogicky môžeme aplikovať postup, ktorý je uvedený pri vlastnosti „nachádza sa“ a zmenšiť čas na

$\Theta((n/k)+1)$, kde n je počet vrcholov v DG_p a k je počet jadier na grafickej karte.

Vrcholy pre komponent graf CG_p dostaneme z DG_p zavolaním funkcie *sscComponentsFinder*. Táto funkcia implementuje Tarjanov algoritmus opísaný v kapitole 4.1.3 s malými zmenami. Ako vstup príjme DG_p a na výstupe dostaneme množinu SSC komponentov. Tento výstup obsahuje každý vrchol z DG_p a spolu s hranami dependency grafu slúži ako vstup pre konštruktor *componentsGraph*.

ComponentsGraph generuje zo vstupných parametrov CG_p . Na začiatku vygeneruje pre každý SCC C vlastný vrchol reprezentujúci C vo vnútri CG_p . Vrcholy sú zaradom očíslované a *posComp* vlastná metóda objektu componentsGraph zabezpečuje prehľadávanie vo vnútri každého komponentu, keď je treba nájsť komponent, v ktorom sa daný vrchol z DG_p nachádza.

Konštruktor postupne prechádza hrany z DG_p a podľa nich vytvára nové, ktoré slúžia na určenie poradia v ktorom sa budú komponenty od grafu odstraňovať a spracovávať procedúrou Instantiate.

Keď už je vytvorený komponent graf vytvoríme množinu EDB(P) jediným zavolaním funkcie EdbFinder.

Teraz už môžeme zavolať procedúru manager, ktorá je opísaná v 4.1.4 , preto sa ňou už nebudem zaoberať.

Hlavnou časťou tejto práce je implementácia procedúry InstantiateComponent. Námet čerpá z programu DLV a preto sa bude v mnohých častiach s ňou zhodovať. Z článku [1] je táto procedúra nasledovne navrhnutá:

```
  DLV InstantiateComponent
  Procedure Instantiate ( P: Program;  DG_p : Dependencygraf;
II:GroundProgram;
  T: poleAtómov);
  begin
       I: poleAtómov;
       C: polePredikátov;
       T := EDB( P );  I = EDB( P ); II :=  Ø ;
       while  DG_p≠Ø  do
            Odstráň SCC C z  DG_p bez vchádzajúcich hrán;
```

```
        InstantiateComponent( P, C, T, I, II );
    end while
end Procedure;
```

Procedúra *Instantiate* zobere ako vstup program *P* a jeho Dependency graf DG_p a ako výstup slúžia polia T, pole pravdivých atómov, a II, pole tých pravidiel, ktoré sa dajú dostať z *P* tak, že ANS ($T \cup II$) = ANS (P). Ako už viac krát bolo spomenuté, vstupný program *P* je rozdelený do modulov podľa maximálnych SCC (Strongly Connected Component) dependency grafu DG_p . Takéto moduly môžu byť spracované po jednom, počnúc tými, ktoré nezávisia od ostatných komponentov, podľa poradia stanovené DG_p .

Bližšie k tomuto. Algoritmus na začiatku vytvorí nové pole atómov I, ktoré bude obsahovať podmnožinu Herbrandovskej bázy relevantnej pre inštanciáciu. Pri štarte, T = EDB (P), I = EDB (P), a $II = \emptyset$. Potom nejaký SCC C bez vchádzajúcich hrán je odstránený z DG_p , a modul programu *P* korešpondujúci s C je spracovaný volaním procedúry *InstantiateComponent*, ktorá využíva semi-naívnu techniku na inštanciovanie rekurzívnych pravidiel popísanú v kapitole 3.1.2.

InstantiateComponent zoberie ako vstup komponent C, ktorý má byť inštanciovaný, T a I, a pre každý atóm $a \in C$, a pre každé pravidlo definujúce *{a}* vyrobí groundované pravidlá *r* obsahujúce len atómy, ktoré je možné dostať z *P*. Počas tejto procedúry aktualizuje množinu T novo vygenerovanými groundovanými atómami, ktoré už sú rozpoznané ako pravdivé a množinu I atómami vyskytujúcimi sa v hlavách pravidiel z II. Algoritmus beží, až kým všetky komponenty DG_p nie sú spracované. Dá sa ukázať, že k danému programu *P*, Ground($II \cup T$) vygenerovaný týmto algoritmom je taký, že *P* a $II \cup T$ majú rovnakú ANS.

Jadrom tejto práce je navrhnúť spôsob, ako využiť výpočtovú schopnosť grafických kariet NVIDIA čoho dôsledkom môže nastať zrýchlenie procesu generovania SM. Keď sa pozrieme na časové zložitosti navrhnutého programu, najnáročnejšia časť je proces groundovania pravidiel *P*. Z až exponenciálnou časovou zložitosťou reprezentuje najvyšší „kopec“ zložitosti, ktorý treba na ceste ku generovaniu *SM* prekonať. Očakávam, že paralelným prístupom sa táto prekážka podstatne zmenší. Cieľom je

navrhnúť CUDA kernel, ktorý bude spustiteľný viacerými CUDA vláknami paralelne a pre daný program P vygeneruje groundované pravidlá.

4.2.1 Procedúra Instantiate

Keď je komponent C vybraný procedúrou *canBeRun*, stačí jedno vykonanie *Instatiate* na to, aby komponent C bol zagroundovaný. Pre všetky komponenty $a \in C$ sú najprv groundované všetky exit rules a až potom v nasledujúcom kroku rekurzívne pravidlá využívaním semi-naívnej evaluačnej metódy popísanej v kapitole 3.1.3.

V oboch prípadoch, je zavolaná funkcia *GroundRule*. Táto slúži ako prepis z jazyka C++ do C. Dátové štruktúry v ktorých sú uschované informácie dôležité na inštanciáciu, musia byť prepísané do nových, pre jazyk C akceptovateľných. Tento krok si vyžaduje znalosti v oblasti práce so smerníkmi, keďže polia v jazyku C sú reprezentované takýmto „ukazovateľom“ na prvý prvok v poli. Vrámci tohto postupu je vytvorené pomocné pole A, ktoré reprezentuje pravidlo, ktoré je nutné groundovať.

Je nutné vyrobiť nové pole B, ktoré obsahuje každý predikátový symbol spolu s konštantami a symbolmi pre premenné. Definujme si funkciu *f*, ktorá pre daný prvok $b \in B$ vráti pozíciu *b* v B. Napr. Pre pole *B={ p, q, s}* a prvok *b*= „s”, f(b) = 2, lebo číslujeme od nuly.

Keď aplikujeme *f* na každý prvok z A, vznikne nám nové pole A', kde platí, $a_i' = f(b_i) \quad ; \quad \forall i ; 0 \leq i < sizeof(A')$.

Pre ďalšiu prácu s A' je nutné zapamätať si indexy, kde končí číslovanie predikátov a premenných. S týmito dvoma indexmi vieme zistiť, neskôr kam a_i' patrí a tým pádom aj čo predstavuje. Rezervujeme pozície 0 a 1 v poli A' pre hodnoty true, false a zvyšok posunieme o dve miesta.

Aplikácia dáva na prvé miesto meno predikátu, na druhé informáciu o jeho negácií, potom všetky termy, ktoré obsahuje.

Mysliac dopredu na prácu v CUDE, je nutné zaviesť v poli A' padding, aby bolo možné pristúpiť k jednotlivým literálom v A' naraz a nie postupne počítať začiatočnú pozíciu literálu podľa predchádzajúceho.

Je rozumné uvažovať o „výške“ daného literálu. Intuitívne, výška literálu *p* je počet znakov, ktoré bude *p* potrebovať na reprezentáciu v poli A' . Majme literál *p*=

p(a,X), pole $A = \{false, true, p, X, a\}$. K nemu vytvoríme A' podľa vyššie uvedeného princípu, z čoho nám vznikne $A'_p = \{2, 1, 4, 3\}$. Výška tohoto literálu je 4.

Padding, ktorý je v tejto práci použitý, je vždy výška najvyššieho literálu z pravidla *r*, ktoré budeme groundovať. Preto je nutné, aby sme si ku každemu literálu v reprezentácií A' pamätali, koľko termov mal v sebe. Toto číslo môže byť umiestnené na viacerých miestach v A' reprezentácií pričom výsledný efekt nám to nezmení. Aplikácia toto číslo umiestňuje za menom literálu, čiže na tretej pozícií relatívne od začiatku konkrétneho literálu.

Celkový padding teda môžeme definovať ako *height(r) = Max(výška(p))+1*; $\forall p \in r$.

CUDA podporuje prácu z 1D, 2D aj 3D poliami, pričom si 2D a 3D polia vnútorne reprezentuje ako 1D pole. Preto aj v tejto práci budeme dávať ako vstup pre CUDA kernel 1D pole. Znova je tu nutné, si pamätať veľkosť(šírku) pravidla *r* (=width(r)), čiže koľko literálov obsahuje.

Funkcia *startGrounding* dostane na vstupe pole A' , width(r), height(r), indexy kde končia predikáty, premenné a termy v A' , výslednú veľkosť. Výslednú veľkosť vypočítame ako počet rôznych termov umocnený na počet rôznych premenných v pravidle a na výstupe dostaneme smerník na výsledné pole.

Funkcia *startGrounding* musí alokovať miesto pre každú premennú v pamäti zariadenia podporujúce CUDU.

Využívajú sa tu CUDA C funkcie *cudaMalloc(premenná, veľkosť)* a *cudaMemcpy(...)* (ktorá je trochu zložitejšia). *CudaMalloc* vyhradí miesto pre premennú na prvom mieste veľkosti špecifikovanej na druhom mieste v argument liste a vráti smerník na túto premennú. *CudaMemcpy* sa stará o prenos dát z hosťovskej pamäti do pamäti zariadenia. Špecifikuje sa odkiaľ kam sa koľko dát prenáša a typ prenosu.

StartGrounding vytvorí ďalšie dôležité pole premenných menom var_Array, ktoré bude slúžiť neskôr na rozlíšenie, ktorý term sa na danú premennú mapuje. var_Array má veľkosť podľa počtu premenných t. j. size(var_Array) = size(variables)*2+1.

Prvé dve pozície sú vyhradené na neskoršie operácie v rámci práce na GPU. Ďalej ale po jednom je index premennej a jej aktuálna hodnota v procese groundovania *r*. Na začiatku, je každá premenná nastavená na prvú konštantu t. j. všetky prvky majú na *i+1* pozícií index prvej konštanty. Zoberme si predošlý príklad. Pre neho by vyzeralo

pole premenných nasledovne: *var_Array={?,?, 3,4}*, kde 3 značí premennú *X* a 4 konštantu *a*.

Keď sa alokuje miesto v pamäti zariadenia pre všetky dôležité premenné, je spustený kernel, ktorý je vykonávaný na GPU paralelne.

Treba špecifikovať počet vlákien, ktoré sa majú spustiť. Maximálny počet vlákien v jednom CUDA bloku je 512, preto je výhodné urobiť to aj pre túto aplikáciu maximum. Každé vlákno obsluhuje jeden predikát v pravidle, teda z toho vyplýva aj obmedzenie 512 predikátov v jednom pravidle. Toto obmedzenie je však možné úplne odstrániť, no na väčšinu reálnych problémov nám táto veľkosť stačí.

4.2.2 Kernel

Jeho úlohou je vziať pravidlo *r* a vygenerovať všetky možné *ground(r)*. Kernel úzko spolupracuje s poľom *var_Array*, v ktorom uchováva informáciu o každom kroku a o celkovom stave procesu. Preto je žiadúce si toto pole uchovávať v shared pamäti, ktorá je zo všetkých dostupných najrýchlejšia. Túto nám príde vhod skutočnosť, že všetky vlákna zdieľajú tú istú shared pamäť.

Ďalej potrebuje kernel informáciu o tom, ako pravidlo *r* vyzerá. Z dôvodu, že shared memory je veľmi obmedzená veľkosťou a nie je na každom zariadení rovnako veľká, je možnosť, že by nám nestačila na uchovanie informácie o konštrukcii pravidla *r*. Preto je táto informácia uložená na device memory, v ktorej je aj *singleGrule* premenná uložená, ktorá slúži ako výsledok pre dané kolo iterácie.

Groundované pravidlá sú vytvárané po jednom, pričom každý literál obsluhuje jedno vlákno. Každé vlákno osobitne vyhľadá vo *var_Array* premenné v tele svojho literálu a priradí im hodnoty určené na aktuálne kolo. Postupne vpisujú do prototypu groundovaného pravidla (*singleGrule*) tieto hodnoty. Padding nám umožňuje toto vykonávať bez „mutex lockov" a tým ušetríme čas, ktorý by bol strávený čakaním na to, aby sa jednotlivé vlákna dostali na radu v zapisovaní.

Po tomto máme pravidlo správne zagroundované a potrebujeme urobiť zmenu v poli *var_Array* tak, aby sme prešli všetky správne možnosti ako substituovať konštantu za premennú. Aplikácia má implementované nasledovné riešenie:

1. Na konci každého kola je hodnota prvej premennej vo var_Array zvýšená o jedna.

2. Ak presahuje niektoré číslo, reprezentujúce hodnotu premennej v ďalšom kole, maximálne možné číslo pre konštantu(max(const) a analogicky aj min(const)), tak je smerom doprava (rastúci index) v poli hľadaná prvá premenná, ktorej hodnota pre ďalšie kolo je menšia ako max(const) a táto je zvýšená o jedna. Hodnoty všetkých premenných vľavo od nej (s menším indexom) sú nastavené na min(const).
3. Ak všetky premenné vľavo od prvého prvku majú hodnotu rovnú max(const), tak ukonči groundovanie.

Myšlienka je podobná systému ako keď zvyšujeme číslo v binárnej sústave o jedna. Je preto ľahké vidieť, že takto naozaj prejdeme všetky možnosti, ako dané pravidlo *r* možno zagroundovať. Navyše tento systém nie je závislý od vedľajších premenných a dokáže bez uchovania si informácie o predošlom kroku vyrobiť ďalší.

Lenže aj tuto je možné využiť neuveriteľnú výpočtovú silu GPU, ak zaručíme paralelnú implementáciu tohto systému.

Prvý krok nie je možné ešte zefektívniť. V druhom kroku však potrebujeme zistiť, kde je najľavejší prvok, ktorého hodnota je odlišná od max(const). Túto využívať len jedno vlákno je hlúposť a mrhanie časom aj prostriedkami. Aplikácia má preto implementované nasledovné riešenie problému:

Využívame pri tom dve premenné *x* a *y*, kde si budeme uschovávať indexy dvoch polí. Bližšie k algoritmu:

1. Každé vlákno obsluhujúce premennú vpravo od prvej porovná svoju hodnotu s max(const) .
2. Ak je táto hodnota menšia ako max(const), zapíše index premennej do *x*.
3. Ak sa $x=y$, tak ukonči algoritmus.
4. Ak sa $x\neq y$ tak *y:=x*.

Táto myšlienka je trochu zložitejšia. Nevieme totiž poradie v akom budú v druhom kroku jednotlivé vlákna zapisovať svoju hodnotu do *x*. Vieme ale definovať nasledujúci invariant: ak nie je *var_Array[x]* najľavejší prvok rôzny od prvého prvku v

poli *var_Array*, taký, že *var_Array[x]* < max(const), tak *x* klesá v každom prebehnutí tohto algoritmu.

Dôkaz Sporom: *var_Array[x]* nespĺňa podmienku implikácie, ale v tomto kroku *x* nekleslo. *Var_Array[x]* musel niekedy predtým splniť 1. bod algoritmu, čo znamená, že je menší ako max(const). To , že *x* nekleslo znamená, že žiadny prvok naľavo od neho nesplnil 1. bod algoritmu. Z toho vyplýva, že všetky prvky naľavo od neho sú rovné max(const). Z toho vyplýva ale, že *var_Array[x]* je najľavejší prvok rôzny od prvého taký, že *var_Array[x]* < max(cons) . Čo je spor s predpokladom.

Ďalej, nech *var_Array[x]* spĺňa podmienku implikácie, ale *x* kleslo aj tak. To znamená, že nejaký iný prvok *var_Array[i]* musel splniť 1. bod algoritmu. To znamená, že $\exists i\,;1<i<x$ < size(*var_Array*) také, že hodnota prvku *var_Array[i]*(čo je *var_Array[i+1]*) je menšia ako max(const). To je ale spor s predpokladom, že *x* je najľavejší prvok s touto vlastnosťou.

Máme dokázané, že tento invariant platí počas behu. Stačí ešte *y* a *x* inicializovať na čísla väčšie ako je veľkosť poľa *var_Array*, aby platil od začiatku.

Máme zaručené, že *x* na konečne veľa zbehnutí algoritmu dosiahne minimum. Keďže postupnosť, v akej vlákna zapisujú je náhodná, môžeme predpokladať, že v priemernom prípade sa nám *x* zmenší o strednú hodnotu medzi minimálnou možnou a aktuálnou hodnotou *x*. Preto priemerná časová zložitosť je $O(\log(n))$.

Nakoniec ešte je nutné vykonať tzv. „údržbárske“ priradenia premenným, aby ďalšie kolo algoritmu zbehlo v poriadku. Na vykonanie tohto stačí jedno vlákno, ktoré je možné špecifikovať premennou threadIdx.

Jednu dôležitú vec nesmieme zabudnúť spomenúť, a to je synchronizácia vlákien počas behu. V časoch, keď sa vykonávajú „údržbárske“ práce, je potrebné zastaviť výpočet ostatných vlákien a počkať na to jedno pracujúce.

Po skončení kernelu je nutné preniesť výsledok práce do hosťovskej pamäte a uvolniť premenné v pamäti zariadenia.

Po ukončení Manager procedúri máme už vstupný program P zagroungovany a podľa postupu uvedeného v definícii SM generujeme stabilné modely P.

5 Záver

V tej to práci sa nám podarilo navrhnúť a úspešne implementovať paralelnú procedúru, ktorá správne zagrounduje vstupný program P.

Efektivita paralelizmu závisí od jeho konkrétnej implementácie. V tejto práci sme sa zamerali na zefektívnenie groundovania logických programov využívaním CUDA technológie na zmaximalizovanie paralelných výpočtov počas tohto procesu. Toto za určitých podmienok môže priniesť zrýchlenie v celkovom procese generovania stabilných modelov.

V tejto aplikácií sa na mnohých iných miestach dá pokračovať v zrýchlovaní procesu. Často treba pridať prvok do poľa, ak sa v tom poli ešte nenachádza. Sekvenčne prechádzame všetky prvky a porovnávame. Na túto procedúru sa dá použiť CUDA, čo prinesie zrýchlenie.

Keď bola táto práca vymýšľaná, bola aktuálna verzia CUDA 2.3, ktorá podporovala len beh jednej kernel procedúry súčasne. Vo verzií 3.0 už je ale povolený beh viacerých kernelov naraz.

Časti tejto práce je teda možné prerobiť na procedúry tak, aby boli využité jednotlivé úrovne paralelného groundovania opísané v [1][2][3] spolu s paralelným procesom opísaný v tejto práci.

6 Literatúra

[1] F. Calimeri - S. Perri - F. Ricca. *Experimenting with Parallelism for the Insantiation of ASP Programs*. Journal of Algorithms. 2007.

[2] F. Calimeri - S. Perri - F. Ricca. *Increasing Parallelism while Instantiating ASP Programs. 2009.*

[3] F. Calimeri - S. Perri - F. Ricca. *A Parallel ASP Instantiator Based on DLV* . Annual Symposium on Principles of Programming Languages .2010.

[4] Tommi Syrjänen . *Lparse 1.0 User's Manual.*

[5] NVIDIA *„NVIDIA CUDA Programming Guide version 3.0"*

[6] NVIDIA *„CUDA Technical Training, Volume 1: Introduction to CUDA Programming"*

[7] NVIDIA *„CUDA Technical Training, Volume 2: CUDA Case Studies"*

[8] Robert Tarjan. Tarjan Algorithm for computing stable models. Design and Analysis of Algorithms. http://www.ics.uci.edu/~eppstein/161/960220.html#sca . 1996

[9] iXBT Labs. Non-graphic computing with graphics processors. http://ixbtlabs.com/articles3/video/cuda-1-p3.html . 2010

[10] M. Čertický. *IK-STRIPS Formalism for Fluent-free Planning with Incomplete Knowledge*. Technical Reports, Comenius University, Bratislava. 2010.

7 Zoznam Príloh

7.1 Priložené CD

Na priloženom CD sa nachádza:

1. Spustiteľný súbor aplikácie
2. Zdrojové kódy aplikácie